FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Návrh API pro exokernel
Student:	Bc. Dorian Řehák
Vedoucí:	Ing. Ladislav Vagner, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

1. Nastudujte si koncept exokernelu OS a modelové možnosti jeho integrace s userspace.
2. Vyberte typické userspace aplikace, které lze díky konceptu exokernelu implementovat efektivněji.
3. Navrhněte programové rozhraní (API) exokernelu. V návrhu se soustřeďte zejména na aplikace z bodu (2).
4. Implementujte demonstrační exokernel a prakticky ověřte jeho schopnosti. Při implementaci vycházejte z vlastního hotového základu kernelu. Dále při realizaci využijte vhodné existující open-source komponenty, které vývoj demonstračního kernelu usnadní (ACPICA, ...).
5. Proveďte empirická měření výkonu vybraných aplikací nad realizovaným exokernelem.
6. Porovnejte změřené výsledky s výkonem obdobných aplikací nad konvenčním kernelem. Diskutujte výsledky.

Diplomová práce

NÁVRH API PRO EXOKERNEL

Bc. Dorian Řehák

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Ladislav Vagner, Ph.D.
11. ledna 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především svému vedoucímu za nekonečnou trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 citovaného zákona.

V Praze dne 11. ledna 2024 .....

Abstrakt

Tato diplomová práce se zabývá návrhem API mezi exokernelem a userspacem. Hodnotí předchozí řešení, jejich výhody a problémy a přichází s inovativními řešeními. Práce se dále zabývá implementací takového exokernelu a jednoduchého demonstračního userspacu. Na závěr empiricky měří dopad takové architektury operačního systému.

Klíčová slova exokernel, API, kernel, operační systém, systémové volání

Abstract

This master's thesis looks into design of API between an exokernel and userspace. It analyzes previous solutions, their advantages and problems, and proposes new solutions. This thesis also addresses implementation of such an exokernel and of a simple userspace. Finally, it empirically measures practical impact of this architecture of operating system.

Keywords exokernel, API, kernel, operating system, system call

Seznam zkratek

API	application programming interface
OS	operační systém
I/O	input output
DMA	direct memory access
ACPI	Advanced Configuration and Power Interface
UEFI	Unified Extensible Firmware Interface
GOP	Graphics Output Protocol
ACL	access control list
TCB	thread control block
CDT	capability derivation tree
IPC	inter-process communication
CT	capability table
MRS	memory register set
MMU	memory management unit
CPU	central processing unit
APIC	advanced programmable interrupt controller
APIT	APIC timer
TLC	translation lookaside buffer
SMP	symmetric multiprocessing
SPT	shadow page table
PRPL	physical region page list

Kapitola 1
Úvod

Účelem operačního systému je ovládání hardwaru a poskytování funkcionality hardwaru aplikacím. Skrze historii počítačů pro tento zdánlivě jasný úkol bylo vynalezeno mnoho různých architektur, které tohoto cíle dosahují jinými způsoby a s jinými trade-offy. Jednou z mála prozkoumaných oblastí je architektura exokernel. Protože této oblasti přes 20 let nebyla věnována žádná pozornost, rozhodl jsem se ji blíže prozkoumat a pokud možno prohloubit.

1.1 Exokernel

Přesná definice exokernelu, popisující jeho nutné znaky, není zatím nikým zcela jasně dána. Práce v této oblasti bylo provedeno minimum. Průkopníkem konceptu exokernel, a víceméně jediným člověkem se mu hlouběji zabývajícím, je Dawson R. Engler, který se svým týmem vytvořil exokernel jménem Aegis a nad ním postavený OS jménem ExOS[1].Následně ještě druhý exokernel jménem Xok, který se od prvního značně liší[2].

Jako motivaci označuje, podle něho, chybný směr historického vývoje architektur operačních systémů. Dvěma hlavními typy kernelů se staly monolitické kernely a mikrokernely, které spolu soupeří v otázkách trade-offu stability a praktičnosti. Tyto dvě kategorie pokrývají všechny široce užívané kernely - Linux (monolitický), XNU (mikrokernel), NT kernel (hybrid mezi monolitem a mikrokernelem) a další. Obě kategorie však typicky mají společnou následující vlastnost: kernel kombinuje ovládání hardwaru s vysokoúrovňovými abstrakcemi operačního systému. Abstrakcemi OS rozumíme koncepty jako jsou soubory, procesy, signály, uživatelská oprávnění, sockety a podobně. U mikrokernelů tyto abstrakce mohou být vyčleněny do samostatných "serverů"a ne součástí samotného základního mikrokernelu, ovšem z pohledu obyčejných aplikací v tomto není rozdíl. Aplikace mají k dispozici pouze vysokoúrovňové abstrakce a nemohou jim uniknout. Dále tyto běžné kernely budu nazývat termínem "tradiční kernely".

Engler tvrdí, že toto je fundamentální omyl a že kernel by měl provádět pouze první skupinu činností (ovládání hardwaru) a druhou skupinu (poskytování abstrakcí OS) vůbec. Jinými slovy, exokernel by měl poskytovat obyčejným aplikacím rozhraní, které co nejblíže kopíruje rozhraní poskytované hardwarem. Abstrakce OS by byly implementovány čistě v userspace a hlavně by byly zcela volitelné. Různé aplikace by si mohly implementovat své vlastní abstrakce (nebo si je nechat poskytnout jinými aplikacemi, tzv. exoservery) nebo používat exokernelem poskytnuté nízkoúrovňové rozhraní přímo a využít plně specifik konkrétního hardwaru pro splnění konkrétního úkolu.

Hlavní myšlenkou této úvahy je fakt, že žádná jiná komponenta operačního systému nerozumí, co aplikace chce a potřebuje, více než aplikace samotná. Logicky následuje hypotéza, že vyššího výkonu aplikace dosáhneme, pokud kernel předá více svojí

zodpovědnosti za správu zdrojů aplikaci.

Tradiční kernely poskytují vysokoúrovňové abstrakce a "hádají"různými heuristikami, co aplikace bude dále dělat. Sledují její systémová volání čtení souborů a snaží se odhadnout, zda je potřeba už předem přednačítat další bloky z disku (tzv. read-ahead). Sledují její přístupy do paměti (skrze tzv. access bit ve stránkovacích tabulkách) a podle toho s pomocí least-recently-used algoritmů odhadnout, které paměťové rámce odswapovat na disk. A tak dále. Přitom aplikace může být schopna dobře vědět (lépe než odhad kernelu), jaké operace budou následovat a kterou část paměti bude dále využívat - ale nemá jak takovou informaci tradičnímu kernel předat. Jedná se zde o jakousi informační bariéru. Tuto bariéru exokernel odstraňuje.

Na druhou stranu, nalezení lokálních optim aplikací nemusí vést k nalezení globálního optima celého systému. Například u operací magnetického disku potřebujeme centralizovanou autoritu, která bude požadavky aplikací na čtení souborů řadit tak, aby se čtecí hlava nepohybovala zbytečně velkou vzdálenost mezi čtenými bloky. Takovou centralizovanou autoritou je tradiční kernel, ale u exokernelu toto může být problematické, protože autority jsou decentralizované ve formě aplikací, které se spolu pravděpodobně koordinovat nebudou.

Nelze tady na první pohled jasně prohlásit, že exokernel vždy poskytuje vyšší výkon. Bude záležet, jak moc hardware principiálně vyžaduje takovou centralizovanou autoritu.

Kapitola 2

Současný stav operačních systémů

2.1 Je to užitečné?

Poptávka po exokernelovém přístupu mezi vývojáři aplikací dle mě evidentně existuje. Můžeme ji vidět v případě Linuxu.

První vlaštovkou potřeby samosprávy zdrojů aplikací samotnou bylo přidání příznaku O_DIRECT u systémového volání open()[3]. Tento příznak vyjadřuje touhu aplikace spravovat čtení a zápis diskových bloků více manuálně, konkrétně dojde úplně k obejití page cache v kernelu. DMA (direct memory access) operace pak hardware provádí přímo do soukromé paměti aplikace a je zodpovědností aplikace provádět I/O efektivně. Tato možnost byla přidána pro aplikace jako jsou například relační databázové systémy[4, str. 668], které si chtějí spravovat vlastní diskovou cache a ne záviset na slepě hádajících heuristikách diskové cache v kernelu.

Dalším krokem je systémové volání sendfile()[5]. Protože u systémových volání read() a write() nemůžeme vyjádřit účel těchto volání uvnitř konkrétní aplikace, bude někdy docházet k neefektivním až zbytečným operacím ze strany kernelu. Pokud aplikace chce jen zkopírovat kus jednoho souboru do jiného kusu souboru, je zbytečné kopírovat při read() data z kernelspace do userspacového bufferu a pak z něj zase kopírovat do kernelspace při write(). sendfile() umožňuje tzv. zero-copy přesun dat mezi file descriptory. Jinými slovy, sendfile() skládá read() a write() efektivně dohromady pro účely kopírování. Exokernelové API můžeme považovat za jakousi generalizaci sendfile(), protože umožňuje skládat libovolné základní operace do větších celků; kdežto sendfile() je limitován jen pro kopírování z jednoho souboru do druhého.

Za zmíňku stojí i systémové volání madvise()[6][7]. Pomocí něho může aplikace říct Linuxu, že k určitému rozsahu virtuální paměti bude v blízké budoucnosti sekvenčně přistupovat (parametr MADV_SEQUENTIAL). Nebo že určitý rozsah virtuální paměti už nebude příliš používán (parametry MADV_DONTNEED, MADV_FREE, MADV_COLD, MADV_PAGEOUT). I toto demonstruje limitace tradičního kernelu. Takové API bude totiž nevyhnutelně poněkud vágní - co to znamená "v blízké budoucnosti"? Aplikace může někdy s jistotou vědět, že bude potřebovat stránku X za milisekundu a pak až stránku Y za 10 milisekund. Toto ale nelze rozumně reprezentovat ve vysokoúrovňovém API poskytovaném tradičním kernelem. Na to je spíše potřeba, aby aplikace měla moc nad mapováním virtuální paměti a nad diskem přímo.

Dalším zajímavým krokem ukazujícím zájem vývojářů o obcházení vysokoúrovňových abstrakcí v systémových API je Vulkan.

2.2 Možnosti nápravy tradičních kernelů

Pokud je tedy pravdou, že aplikace ví lépe, co aplikace bude dělat a co potřebuje, než kernel - nestačí pak jen přidat pár nových systémových volání do tradičních kernelů? Aplikace by mohly tímto přidaným API kernelu předat tyto dodatečné informace. Tradiční kernel by pak nemusel tolik nepřesně hádat a výkon by se zvýšil. Vývoj se již tímto směrem ubíhá, jak popisuje předchozí podkapitola.

Ačkoliv by toto zlepšilo situaci, nemyslím si, že je to uspokojující řešení. Problém je zde dvojí:

- Vysokoúrovňové rozhraní někdy prostě nemůže poskytovat nebo využívat nízkoúrovňové informace

  Například pokud tradiční kernel poskytuje správu virtuální paměti aplikace, kde kernel transparentně odstraňuje a přidává paměťové rámce a stránkovací tabulky. Aplikaci by se mohlo hodit sledovat svoje vlastní vzorce přístupu do paměti, ale k tomu by aplikace potřebovala být schopná přímo číst strankovací tabulky (procesor tam ukládá dirty a access bity). I kdyby ji kernel zpřístupňoval stránkovací tabulky, tak by integrace s takovou transparentní správou byla problematická, protože kernel by aplikaci měnil stránky "pod rukou". Aplikace by si musela být schopna spravovat vlastní virtuální paměť sama. Pak se posouváme směrem k exokernelu.

- Rozmanitost hardwaru

  Představme si diskový řadič. Snad všechny takové řadiče budou poskytovat operaci typu "přečti X bloků od bloku Y do paměťového rozsahu od adresy Z". Ale různé řadiče mohou poskytovat specifické rozšíření takové operace. Například v I/O požadavcích u NVMe můžeme poznačit, že víme, že čtená data jsou komprimovatelná[8], a disk toto může zkusit využít pro vyšší přenosovou rychlost. Můžeme poznačit, že chceme malou latenci, nebo že je pro nás přijatelné si trochu počkat - a disk může vykonávat paralelně zpracovávané požadavky ve vhodnějším pořadí. Nebo můžeme poznačit, že do čtené oblasti budeme brzy zapisovat, a disk může lépe využít svojí zabudovanou volatilní cache pro jiné čtecí požadavky. NVMe dokonce umí i operaci porovnávání obsahu paměti počítače a obsahu bloků na disku, aniž by bylo nutné nejdříve přečíst bloky do paměti počítače a softwarově porovnávat dvě oblasti paměti pro shodnost/rozdílnost. Jiné typy řadičů budou poskytovat jiné funkcionality.

V praxi nelze vymyslet skutečně univerzální vysokoúrovňové API (jako read(), write()), které by bylo schopno plně reprezentovat všechny schopnosti všeho rozmanitého hardwaru a skrze které by mohla aplikace věrně reprezentovat svoje budoucí přístupové vzorce. Pokud bychom se místo univerzálního rozhraní rozhodli do Linuxu přidat speciální systémová volání pro každé takové specifikum hardwaru, výrazně bychom se přesunuli blíže ke konceptu exokernelu, avšak nedokonale.

Engler se odkazuje na tzv. end-to-end princip[1]. **Je žádoucí přeskočit překážejícího prostředníka (vrstva abstrakcí OS v kernelu) a komunikovat přímo mezi dvěma koncovými aktéry - mezi aplikací a ovladačem hardware. Tím se neztrácí informace po cestě a oba konce mohou pracovat s největším množstvím a přesností navzájem si předávaných informací.**

Exokernel jde cestou poskytování jakési stavebnice, jejíž prvky jsou nízkoúrovňové operace a které aplikace může libovolně skládat a mezi sebou propojovat. Z této stavebnice si aplikace může sestavit libovolné vysokoúrovňové operace a abstrakce. A výsledek může být výkonnější, protože kombinování operací provádí aplikace samotná sobě na míru a ne jen heuristicky odhadující kernel.

2.3 Typy aplikací benefitujících z použití exokernelu

Obecně by z exokernelu benefitovaly aplikace, kde je i malý nárůst výkonu velmi ceněný, a kde se tedy vyplatí investovat více programátorské práce do manuálnější správy prostředků.

Představme si například:

- Vědecké výpočetní aplikace

  Takové aplikace mohou pracovat s enormním množstvím dat, které se nevejdou celé do hlavní paměti. Exokernel by jim dovolil, na rozdíl od tradičních kernelů, spravovat si vlastní virtuální paměť, chytat svoje page faulty a implementovat si vlastní logiku swapování na disk. Nebo dokonce víceúrovňové swapování mezi různými diskovými poli s rozdílnými rychlostmi - podle toho, jak aplikace usoudí, za jak dlouhou dobu které úseky paměti bude znovu potřebovat.

- Webové aplikace

  Nejvíce by z exokernelového API získaly aplikace kombinující operace různých kategorií hardwaru dohromady. Engler ve své práci o svém druhém exokernelu jménem Xok[2] prezentuje HTTP server (Cheetah), který čte z disku soubory a posílá je síťovou kartou jinému klientovi. Exokernel zde umožnil userspacové aplikaci provádět DMA operaci čtení z disku do paměti, přičemž na disku byly uloženy tyto serverem poskytované soubory jako předem rozkouskované do TCP paketů. Jakmile byla DMA operace čtení dokončena, exokernel informoval aplikaci a ta jen rychle upravila síťové adresy v paketech a dále řekla exokernelu, aby provedl DMA operaci do síťové karty. Celý HTTP server takto mohl posílat data bez jediného kopírování bufferů - ani v userspace, ani v síťovém stacku v kernelu. Což vedlo ke značnému nárůstu výkonu u některých typů zátěže.

- Multimediální streamování

  Podobné jako předchozí případ. Je zde žádoucí kombinovat různé hardwarové operace přes sebe pro zero-copy operace. Rozdíl je, že zde nás nemusí zajímat jen propustnost, ale spíš stabilita latence. U tradičního operačního systému se toto těžko garantuje, protože síťové operace jsou reprezentovány jen vysokoúrovňovými abstrakcemi jako jsou sockety. Aplikace nemohou rozumně implementovat vlastní síťový stack či mít moc nad pořadím operací síťové karty (ve smyslu I/O plánovače). Dále u této kategorie aplikací budeme dobře vědět přístupové vzorce při čtení disku a tyto informace chtít předat diskovému řadiči (pokud je dostatečně sofistikovaný).

- Hry

  V podkapitole 2.1 jsem zmínil nedávný vývoj v oblasti API pro vykreslování obrazu na grafické kartě; přesun směrem k nízkoúrovňovým API je zde jasný. Nejedná se ale jen o Vulkan nebo Metal. Hry potřebují mít určitou moc i nad plánovačem úloh pro efektivní multithreading. Některé moderní procesory poskytují různé typy jader, některé pro větší výkon a některé pro menší spotřebu, v rámci jednoho procesoru zároveň. K dobrému výkonu her je třeba přiřazovat vhodná vlákna na vhodná procesorová jádra. Tradiční operační systémy poskytují systémová volání pro určení tzv. procesorové afinity a některých dalších atributů pro plánovač. Platí zde teoretická omezení vysvětlena v podkapitole 2.2. Exokernel by mohl umožnit hře lépe si spravovat a rozdělovat procesorový čas, dokonce si možná i vytvořit vlastní plánovač.

Aplikace, které by z exokernelu nebenefitovaly, jsou ty, kde nám nárůst výkonu za tu větší práci při vývoji aplikace nestojí. Například asi nebudeme používat API exokernelu pro přímý přístup k diskovým operacím u jednoduchého shellového programu jako je grep; tam využijeme pouze příjemnějších vysokoúrovňových abstrakcí poskytujících soubory a page cache. Zde je třeba jasně říci, že u operačního systému na bázi exokernelu nemusí všechny aplikace používat

nízkoúrovňové exokernelové API. Abstrakce operačního systému mohou být implementovány ve formě knihoven, ke kterým je naše aplikace přilinkována, a ve formě jiných userspacových vláken, které naslouchají požadavkům odpovídajícím systémovým voláním u např. monolitického kernelu (”prověď pro mě readdir()”a podobně). Taková aplikace je programována podobně jako u tradičních operačních systémů, klidně by mohla používat například jen POSIXová API.

Exokernel tedy nenutí aplikaci spravovat si vlastní zdroje; exokernel dává jen tuto možnost navíc, pokud ji chceme využít.

2.4 Co je to tedy exokernel?

Jak jsem zmínil na začátku textu, definice exokernelu není jasně daná. Pro zbytek práce budu stavět na následující, mé vlastní, definici exokernelu.

Exokernel je typ kernelu, který má následující dvě vlastnosti:

1. Poskytuje téměř přímý, ale bezpečný, přístup k HW neprivilegovaným aplikacím

   Každá funkcionalita, kterou hardwarové zařízení poskytuje, by měla být dostupná userspacu. Žádné činnosti, které hardware nebrání, by neměl bránit kernel - ale jen pokud tato činnost nenarušuje stabilitu nebo bezpečnost systému. Například diskový řadič může zapisovat data kamkoliv do fyzické paměti, ale určitě nechceme dovolit aplikaci přepisovat datové struktury kernelu nebo jiné aplikace. Omezení zajišťující stabilitu a bezpečnost systému jsou jediná žádoucí a nelze u nich dělat kompromis.

   Nejedná se tedy o doslova přímý přístup k hardwaru. Userspace nebude typicky manipulovat přimo s MMIO oblastmi zařízení, protože pak by nešlo vynutit bezpečnost. Hardware bude manipulován ovladačem v exokernelu a exokernel bude poskytovat userspacu API, které vypadá velmi podobně jako rozhraní daného hardwaru. Exokernel pro každý požadavek z userspacové aplikace ověří, že aplikace má právo tuto operaci provést, a pak je požadavek předán ovladači. Pokud aplikace právo nemá, operace není provedena.

2. Přístup k HW je granulární

   API exokernelu by mělo být navrženo tak, aby přístup k zařízení nebyl nutně exklusivní pro jednu aplikaci. Například určitě nechceme, aby nešlo rozdělit přístup k jednomu disku mezi více aplikací. Princip granularity nám bude diktovat, že každý zdroj může být používán více aplikacemi - ale tak, že si navzájem ”nelezou do zelí”(pokud si to nepřejí) a že se nemusí navzájem koordinovat (či o sobě vůbec vědět). Zdrojem zde můžeme rozumět procesorový čas, rozsah fyzické paměti, rozsah diskových bloků, zlomek bandwidth diskového řadiče, zlomek bandwith síťové karty, právo přijímat síťové pakety s určitými hodnotami polí (UDP porty) a podobně.

   Ne vždy je praktické takovou vlastnost dodržet. Někdy by to mohlo vést k příliš vysokému paměťovému overheadu při správě oprávnění. A někdy to prostě nedává smysl; například umožnit aplikacím nastavovat aktuální datum v CMOS RTC na takové granulární úrovni, že jedna aplikace může nastavit jen aktuální rok a druhá jen aktuální měsíc, naprosto postrádá smysl. Zde je rozumné poskytovat přístup negranulárně ve formě manipulace s CMOS RTC jako s jedním nedělitelným celkem.

Tyto dvě vlastnosti můžeme kontrastovat s jinými architekturami operačního systému:

- DOS

  Operační systému rodiny DOS také poskytovaly aplikacím přímý přístup k hardwaru. Jednalo se o exokernely? Rozhodně ne, protože DOS neposkytoval takový přístup bezpečně. Neexistovala žádná vrstva mezi aplikací a hardwarem, které by ověřovala, zda je takový požadavek

bezpečnostně v pořádku. Aplikace mohla sahat kamkoliv na disku či v paměti a klidně narušit vnitřní stav DOSu samotného. To u exokernelu není možné, bezpečnost je vždy zaručena.

- Linux

  I Linux často poskytuje výše zmíněnou první vlastnost. Například pro již dříve zmíněné diskové řadiče NVMe, u kterých můžeme na HW úrovni specifikovat přístupové vzorce, Linux poskytuje takové přímé API skrze systémové volání ioctl() nad souborem /dev/nvmeXnY.[9] Toto ioctl() API umožňuje aplikaci posílat NVMe příkazy přímo do ovladače a tedy jedná se o dříve zmíněný exokernelový end-to-end princip. Takové API zdánlivě splňuje požadavek na nenarušování bezpečnosti systému - pouze superuživatel má oprávnění takový přímý přístup použít. Jedná se o exokernel?

  Není tomu tak. Sice pouze superuživatel může provádět takové operace, ale jakmile má aplikace práva superuživatele, může stejně jako u DOSu narušit bezpečnost systému. Je to tím, že ioctl() umožňuje posílat přímo "syrové"NVMe příkazy; Linux je nijak neinterpretuje, jen je zařadí do fronty v ovladači. Takže opět může aplikace zapisovat i do fyzické paměti, která ji nepatří.

  Další důvod, proč toto nelze považovat za API ve stylu exokernelu je fakt, že toto API není granulární. Není zde žádná možnost rozdělit rozsahy bloků disků a různým aplikacím přiřadit rozsah, mimo který nemůže aplikace sáhnout. Aplikace má u tohoto API buď totální moc nad diskem nebo vůbec žádnou.

  U exokernelu nechceme mít potřebu "speciálních"privilegií jako je superuživatel. Protože smyslem exokernelu je umožnit aplikacím efektivněji využít hardware, je klíčové aby k hardwaru mohly nízkoúrovňově přistupovat všechny aplikace; i aplikace, kterým nemusíme "důvěřovat". Chceme být nuceni důvěřovat pouze ve smyslu, že naše důvěra je vkládána jen v granulárně definovatelné podmnožiny zdrojů (např. rozsahy diskových bloků), které aplikaci svěříme; a můžeme si být jisti, že nehledě na záměrnou škodlivost či závadovost aplikace tato aplikace nemůže manipulovat mimo svěřené rozsahy. U tohoto Linuxového API to takto není, zde musíme naprosto věřit aplikaci běžící pod superuživatelskými právy, že například nepřepíše oddílovou tabulku. I když třeba smysl té aplikace je jen zjistit S.M.A.R.T. stav disku.

- Mikrokernely

  Mikrokernely se mohou jevit jako exokernely. Též jsou minimalistické a delegují funkcionalitu do userspace. Někdy i se sofistikovaným capability systémem, který brání userspace komponentám sahat, kam nemají.

  Rozdíl je v tom, co je cílem. Ačkoliv tyto userspacové komponenty v operačních systémech na bázi mikrokernelu nejsou součástí kernelu, obyčejné aplikace před nimi stále nemohou uniknout. Mikrokernelové servery poskytují ovladače i abstrakce operačního systému jako kompletní celek obyčejným aplikacím. Obyčejným aplikacím není dovoleno komunikovat s ovladačem disku nebo síťové karty přímo, protože zde není žádná dostatečně granulární vrstva oprávnění, a tak je komunikace s/mezi mikrokernelovými servery povolena jen privilegovaným komponentám (jako je ovladač souborového systému či pager). Smyslem vyčlenění funkcionality do userspace je spíše snížení komplexity kernelu a zvýšení bezpečnosti v případě bugu v ovladačích.

  Engler tento rozdíl popisuje výstižně: [*We define application-level software as software that can be changed and/or avoided by any application; this is in contrast to software at user-level (or in user-space), which may require very high privileges to adapt or replace (e.g., replacing a device driver often requires root privileges). Much of the fixation micro-kernels have with putting pieces of the kernel into user-space comes from a confusion between user and application level.*][10].

2.5 O tomto exokernelu

Závěrem úvodu představuji svůj exokernel jménem **FleK**. Zkratka je ze slov ”flexible kernel”, protože jakožto exokernel klade minimum limitací na možnosti projektů nad ním postavených. Můžeme si nad ním postavit Unix-like systém či něco zcela zvláštního.

Cílovou architekturou je PC. Jedním z důvodů je fakt, že toto je jediná mně přístupná architektura, která zároveň má velmi dobrou dokumentaci a mnoho konkurenčních projektů k inspiraci a porovnání. Trade-offy a hardwarová podpora jsou orientovány na typický v současnosti vyráběný laptop.

Zvolenou procesorou architekturou je AMD64. IA-32 podporována není, protože je dnes už příliš zastaralá. Některá volitelná rozšíření AMD64 jsou podporována (například 1-GiB stránky), ale většina není (například AVX). Firmwarem musí být UEFI[11], BIOS podporován není. K video výstupu je použito UEFI GOP[11], který umožňuje konfiguraci rozlišení (dle mého testování někde i 4K) a poskytnutí memory-mapped pixelového framebufferu. Pro uživatelský vstup je použito PS/2 klávesnice a myši. Takový hardware na moderních stolních počítačích už vůbec neexistuje, ale i na moderních laptopech jsou tímto rozhraním připojeny zabudované klávesnice a touchpady[12]. Pro persistentní data je implementována podpora diskových řadičů NVMe ve verzi 1.0 (jsou zpětně kompatibilní), i s některými volitelnými rozšířeními. Je implementována podpora PCI a PCI Express, včetně rozšíření MSI a MSI-X. Podpora ACPI je poskytnuta knihovnou ACPICA od Intelu. Jedná se o jedinou komponentou mého projektu, která nebyla napsána mnou. Nad ACPICA byly implementovány některá rozhraní pro vypnutí počítače, detekci stavu baterie[1] a podobně.

FleK byl otestován na dvou PC, v Qemu a ve VirtualBoxu.

Konkrétní detaily implementace jsou popsány veskrze práci.

[1] Ovladač je ve FleKu funkční, ale nestihl jsem ho začlenit do capability systému

Kapitola 3

Návrh API

3.1 Předchozí řešení

3.1.1 Secure bindings

Jestliže chceme reprezentovat aplikacím umožňovat granulární přístup k hardwaru, musíme mít v exokernelu nějaký systém oprávnění, který bude držet informace nutné k vyhodnocení, zda určitá operace může být provedena či ne.

Engler ve své první práce popisuje koncept jménem secure binding[1]. Secure binding je bezpečnostní prvek, který reprezentuje oprávnění manipulovat s nějakým zdrojem. Definující vlastností secure binding je, že ověření oprávnění je provedeno při vytvoření secure binding příslušného k danému zdroji. Ne při manipulaci se zdrojem; při manipulaci je pouze ověřena existence secure bindingu, protože jeho existence samotná je důkaz oprávnění manipulace se zdrojem (jinak by mu nebylo v minulosti dovoleno vzniknout). Toto je důležité pro výkon. Je naprosto nepřijatelné při každé jednotlivé operace složitě vypočítavat nebo procházet nějaký velký strom historie předávání a rozdělování oprávnění mezi aplikacemi. Chceme, aby tato operace ověření secure binding proběhla v konstantním čase a co nejrychleji. Vytvoření secure binding může být pomalé, protože nejspíš nebudeme nikdy muset vytvářet nová oprávnění v tight loop.

Secure binding je tedy jakýsi token, který jednou vznikne při vytvoření nového granulárního oprávnění a aplikace ho pak při operacích jen opakovaně ukazuje exokernelu.

Aegis implementuje tyto secure bindings různými způsoby. Secure bindings k paměťovým rámcům jsou implementovány databází v exokernelu, která popisuje, kdo je současným vlastníkem. Secure bindings k přijímané síťové komunikaci jsou implementovány jako userspacem do kernelu uploadovaný kód, který je vyhodnocován virtuálním strojem v exokernelu pro každý přijatý paket.

Zajímaly mě konkrétní implementační detaily secure bindings k paměťovým rámcům a diskový blokům v Aegis, ale bohužel toto není nikde popsáno. Ovšem nalezl jsem nějaké implementační detaily k Englerovu druhému exokernelu, Xok, kde je databáze secure bindings k paměťovým rámcům popsána jako pole, přičemž každý objekt v poli popisuje jeden rámec[13]. Oprávnění jsou v každém objektu reprezentovány jako ACL seznam.

Takový systém mi přišel příliš neflexibilní. Například (pravděpodobně) nesleduje historii svěřování oprávnění mezi aplikacemi. Jak potom implementovat svěření zdroje od aplikace A aplikaci B, které ho dále svěří aplikaci C? Chceme, aby aplikace A stále mohla se zdrojem manipulovat a ideálně ho i odebrat od B (což by ho odebralo i C), pokud se tak A rozhodne. To vyžaduje složitější databázi oprávnění než takového systém secure bindings.

V práci o Aegis je popsán protokol odebírání (revokace) oprávnění. Práce mluví o revokaci secure binding patřící aplikaci exokernelem [We view the revocation process as a dialogue between an exokernel and a library operating system.][1], ale ne o revokaci mezi aplikacemi. To mě utvrdilo v mém odhadu (předchozí odstavec) a začal jsem hledat alternativní řešení. Schopnost aplikací svěřovat podmnožiny svých zdrojů jiným aplikacím a možnost si je násilně vzít zpět revokací považuji totiž za důležitou pro možnost svěřování prostředků aplikacím, v jejichž korektnost úplně nevěříme.

Engler svoje dva exokernely implementoval na konci 90. let a od té doby došlo ke značnému pokroku na poli capability-based kernelů. Především v rodině mikrokernelů L4, jako například seL4 a Barrelfish. Capability systémem seL4 a jeho klonů jsem se silně inspiroval a blíže ho popisuji v kapitole Capability systém3.2.

3.1.2 Disková cache

Po dostatečně dlouhém běhu operačního systému nic jako nevyužitá paměť neexistuje. Hlavní paměť, kterou nepoužívají kernel a aplikace k uložení svých datových struktur, totiž chceme použít jako cache disku. Je otázkou, kým je taková paměť spravována u operačního systému na bázi exokernelu.

U tradičních kernelů je spravována kernelem.

- Linux ji využívá pro page cache.
- Exokernel Xok spravuje nevyužitou paměť jako buffer cache[14].

Informace o exokernelu Aegis v tomto ohledu nejsou k dispozici.

Xok poskytoval systémová volání, kterým se aplikace mohla zeptat, zda buffer cache obsahuje daný blok (podle identifikátoru), a pokud nikomu daný paměťový rámec nepatří, si ho může i namapovat do své virtuální paměti. Aplikace mohou do této globální buffer cache vkládat paměťové rámce s libovolným obsahem a tvrdíc, že jde o určité diskové bloky a že jsou či nejsou dirty. Musí ale mít secure bindings k daným paměťovým rámcům a diskovým blokům.

Takové řešení jsem viděl jako problémové ze dvou důvodů:

- Implementace page cache

  Pro implementaci page cache nestačí mít jen bloky v paměti, je třeba je mít ve správných místech v paměti. Například jeden soubor může být na disku náhodně fragmentovaný s blocích o velikosti 512 B. Ale v page cache musí být bloky po sobě jdoucí v rámci jednoho paměťového rámce. Tedy soubor, který se skládá z 12 po sobě **ne**jdoucích bloků na disku, musí být v page cache jako 8 po sobě jdoucích bloků v jednom paměťovém rámci ($8 * 512B = 4096B$) a pak znovu 4 po sobě jdoucích bloků v jiném rámci. Jen pak je možné soubory mapovat do virtuální paměti (ekvivalent POSIXového mmap()[15]).

  Problém spočívá v tom, jak toto reprezentovat v takové globální buffer cachi ve stylu Xoku. Stejné bloky totiž mohou být součástí různých "pohledů"na obsah disku a tudíž muset být jinak poskládané uvnitř paměťových rámců. Například pokud na Linuxu provedeme mmap() souboru /dev/sda1 a souboru /home/user/test.txt (který je fragmentovaně uložen uvnitř /dev/sda1), pak v Linuxové page cachi budou ty samé bloky dvakrát, právě z tohoto důvodu[4, str. 601–602]. U sofistikovanějších souborových systémů dochází k deduplikaci obsahu souborů[16] a stejný problém tedy nastává i u memory-mappingu jen obyčejných souborů. Pak nestačí asociovat bloky v exokernelové buffer cachi jen podle čísla disku a čísla bloku, pokud chceme mít page cache.

- Policy

  Jednou z myšlenek exokernelu je, že se snažíme vyhýbat globálním heuristikám a necháváme aplikace dělat rozhodnutí. Ale pokud máme diskovou cache v kernelu, ať už je to block cache, buffer cache nebo page cache, jsou takové heuristiky nevyhnutelné. Poskytovatel cache musí nějak rozhodovat, kdy a která data vyřadit, když je třeba vložit nová. Skrze jakou heuritisku by ale mohl exokernel o takové věci rozhodovat, když se poskytováním jen nízkoúrovňového API (na rozdíl od tradičních kernelů) zbavil zdrojů informací o chování aplikací? Tyto cache také bývají úzce propojené s ovladačem souborového systému, protože pokud je blok/stránka dirty, je třeba říct ovladači souborového systému, at je nejdříve zapíše. Ale ovladače souborového systému jsou v userspacu, exokernel o těchto věcech vůbec neví. Exokernel by mohl dirty bloky předat jen kernelspacovému diskovému ovladači, ale to by bylo problémové pro userspacové ovladače souborových systémů, který by na základě toho potřebovaly aktualizovat čas přístupu souboru nebo provést deduplikaci (což by zde nešlo vůbec).

Rozhodl jsem se tedy nemít žádnou cache v kernelu a místo toho nechat userspace, ať si page cache nebo jiné typy cachi implementuje u sebe, stejně jako jakoukoliv jinou abstrakci operačního systému. Toto je první radikální vzdálení se mého exokernelu od exokernelů Englera.

Důsledky jsou dalekosáhlé. **Veškerá** "nevyužitá"paměť musí patřit userspacu, aby ji mohl využít pro cache. To znamená, že kernel nemá žádný prostor k expanzi, když potřebuje alokovat paměť pro svoje datové struktury. Tradiční kernel (a předchozí exokernely) mohou při nedostatku paměti při alokaci prostě zahodit nějakou stránku v cachi obsahující ne-dirty bloky a alokovat

v tomto prostoru. To ale nejde, pokud veškerá ”nevyužitá”paměť je vlastně využitá userspacem. Zdá se, že jediná možnost takového exokernelu je poprosit skrze nějaký protokol nějakou aplikaci, aby darovala trochu fyzické paměti kernelu.

3.1.3 Alokace paměti exokernelem

Protože je veškerá kernelem nevyužitá fyzická paměť po bootu předána userspacu, kernel bude v potížích, pokud při obsluze hardwaru bude potřebovat alokovat paměť. Řešení je očividné - stačí nechat kernelu nějakou rezervu volnou, například 10 MiB. Můj kernel v současnosti nepodporuje plug-n-play mechanizmy a tudíž by nějaká malá fixní rezerva měla ovladačům stačit na neomezeně dlouhou dobu běhu systému. O problematice plug-n-play piši na konci práce.

Co ovšem dělat, pokud bude potřeba alokovat paměť při obsluze systémových volání? Co když aplikace budou schválně mapovat svoji virtuální paměť tak, že kernel bude muset alokovat stovky MiB jen na stránkovací tabulky? Co když aplikace požádá o vytvoření tisíce nových vláken? Tady ani větší rezerva nebude stačit a škodlivá aplikace by vždy mohla exokernel paměťově vyčerpat.

Aegis řeší podobný problém dvěma kroky[1]:

- Požádání nějaké aplikace o zdroj

  Vybraná aplikace dostane od exokernelu zprávu, že musí odevzdat určité množství paměti do určitého časového limitu.

- Násilné odebrání zdroje

  Pokud aplikace včas nesplnila požadavek, bude zdroj násilně odebrán. Toto může aplikaci rozbít, ale exokernel může běžet dál.

  Toto řešení se mi nelíbilo, protože odděluje ”zločin”a ”trest”.

Jedna aplikace může provádět systémová volání, která někde v hloubi exokernelu způsobují alokace; třeba i nepřímo, jako zaplňováním hashovací tabulky nějakého ovladače, která udržuje load factor realokací na větší tabulku. A nějaká jiná aplikace bude zvolena k odevzdání paměti. Nebo dokonce zničena, pokud zrovna nebude ve stavu umožňující splnění tohoto požadavaku.

To mi také přišlo silně nežádoucí a tak jsem se znovu poohlédl po alternativních řešeních. Zaujalo mě řešení tohoto problému u mikrokernelu seL4. seL4 a jeho klony taktéž nemají žádnou diskovou cache v kernelu.

Z pohledu vývoje exokernelu je mikrokernel seL4 pozoruhodný. Poskytuje určité druhy hardwarových zdrojů přímo a granulárně userspacu, konkrétně procesorový čas, fyzickou pamět a stránkovací tabulky[17]. Čímž stojí v kategorii mikrokernelů blízko exokernelům.

3.1.4 Deterministická alokace paměti

Alokace paměti při obsluze systémových volání u seL4 funguje tak, že každé systémové volání má deterministicky dané množství paměti, které potřebuje. To znamená, že když userspace chce po mikrokernelu seL4 provést nějaké systémové volání, tak userspace dopředu ví, kolik paměti bude potřeba, a toto množství paměti při systémovém volání předá kernelu. Kernelu tedy nikdy nedojde paměť při obsluze systémových volání; buď je mu dostatek paměti darován přímo při tom volání, nebo není a pak kernel odmítne volání splnit. Protože potřebné množství paměti je z pohledu userspace deterministické, k tomuto odmítnutí může dojít jen v případě bugu v aplikaci. Aplikace nikdy nemusí hádat, zda a kolik paměti musí předat při systémovém volání. Je to jasné z povahy systémového volání a nezávislé na předchozích voláních.

Kupříkladu, pokud aplikace pod mikrokernelem seL4 chce vytvořit a spustit nové vlákno, seL4 bude potřebovat nějaké množství paměti pro objekt reprezentující vlákno, kam preemptivně multitaskingový plánovač bude ukládat hodnoty registrů při přepnutí úloh. Aplikace bude mít

capabilitu reprezentující rozsah fyzické paměti a provede systémové volání, kde ukáže tuto capabilitu a požádá o vytvoření[18] nové capability reprezentující vlákno (TCB). seL4 v tomto rozsahu fyzické paměti inicializuje takovou datovou strukturu. Nic alokovat nemusí, fyzickou paměť dodala aplikace. Nyní aplikace provede druhé systémové volání, kde požádá seL4 o spuštění vlákna. Při tomto volání ukáže (v předchozím kroku vytvořenou) capabilitu reprezentující datovou strukturu vlákna. seL4 ani tentokrát nepotřebuje alokovat paměť, aplikace dodala datovou strukturu vlákna. seL4 tuto strukturu jen prováže do spojových seznamů plánovače.

V tomto duchu fungují i všechna ostatní systémová volání. Například při mapování virtuální paměti je třeba dodat capabilitu reprezentující stránkovací tabulku (pokud je třeba). seL4 nic nealokuje.

Toto je velmi elegantní, ale zároveň složité na implementaci. Kernel musí zajistit, že množství potřebné paměti je vždy z pohledu userspacu deterministické (tj. neexistuje žádný "skrytý stav"v kernelu). To například znamená, že nelze používat dynamicky realokovaná pole nebo dynamicky se nafukující hashovací tabulky. Pak by userspace nemohl vědět, zda je při systémovém volání potřeba dodat paměť (a kolik) či ne. I přes tuto velkou výzvu z hlediska implementace jsem se rozhodl adoptovat toto řešení pro svůj exokernel.

3.2 Capability systém

V předchozí kapitole jsem vyzdvihl nedostatky předchozích exokernelů:

- **Nedostatek č. 1**: příliš limitovaný systém oprávnění (secure bindings)

  Chceme, aby systém capabilit sledoval historii odvozování a předávání oprávnění, aby aplikace byly schopny věnovat zdroje jiným i nedůveryhodným aplikacím bez rizika nemožnosti si tyto zdroje vzít zpět od takové potenciálně nespolupracující aplikace.

  Dále se mi nelíbí fakt, že aplikace pod Aegis a Xok si prostě mohou prostě přivlastnit např. libovolnou fyzickou paměť, která zrovna není využívána. Jako lepší volba mi přijde systém, kde všechny zdroje někomu musí patřit a ten se pak rozhoduje je delegovat jiným aplikacím. Tím lze implementovat policy.

- **Nedostatek č. 2**: problém alokace paměti kernelem, pokud všechna ostatní paměť patří userspacu

  Tento problém je umocněn mým rozhodnutím mít diskovou cache v userspacu a ne v kernelspacu. Potřeba exokernelu násilně odebírat paměť (skrze *abort protocol*) víceméně náhodným aplikacím je velmi nežádoucí.

Tyto nedostatky budeme chtít při návrhu capability systému vzít v úvahu. Před samotným návrhem jsem se rozhodl formulovat požadavky. Následující invarianty musí nesmlouvavě platit. A to pro všechny typy zdrojů.

- **Invariant č. 1**: capabilita odvozená od jiné capability musí být její podmnožinou.

  Z toto plyne tranzitivita. Jestliže A je odvozené od B a B je odvozené od C, pak víme, že A je podmnožinou C.

  Jestliže bychom aplikaci svěřili nějaký zdroj (např. rozsah fyzické paměti) a ona by si mohla toto oprávnění rozšířit i na něco, co jí nebylo původně svěřeno, nastalo by hrubé narušení bezpečnosti systému.

  Co přesně je myšleno výrazem podmnožina závisí na typu capability.

- **Invariant č. 2**: capabilita nemůže existovat, pokud nějaký její předek přestal existovat.

  Capabilita reprezentuje oprávnění aplikace nakládat s nějakým zdrojem. Jestliže capabilita přestane existovat, aplikace ztratila právo s daným zdrojem nakládat. Je nepřípustné, aby nějaké odvozená capabilita i nadále existovala, protože pak by aplikace mohla nakládat se zdrojem, ke kterému by už teď neměla mít oprávnění.

  Tento invariant je svázán s dodatečným požadavkem č. 1 (vizte dále). Pokud capability nelze revokovat, je tento invariant nepodstatný.

- **Invariant č. 3**: capabilita nemůže vzniknout z ničeho za běhu systému.

  Každá capabilita má svého rodiče, ze kterého byla odvozena. Toto tvrzení má jedinu výjimku - prapůvodní capability vytvořené exokernelem (resp. ovladači hardwaru) při bootu. Ty žádného rodiče nemají. Všechny capability v systému jsou odvozeny (tj. podmnožiny) od těchto prapůvodních capabilit.

  Například pro každý disk bude vytvořena při bootu jedna prapůvodní capabilita reprezentující celý rozsah bloků. Po bootu už může systém vytvářet jen odvozené capability (podmnožiny rozsahů bloků) této prapůvodní capability či jiných odvozených capabilit.

Předchozí invarianty dohromady zajišťují, že **množina capabilit předaná aplikaci je tranzitivním uzávěrem oprávnění této aplikace**. To je velmi hezká vlastnost. Pokud svěříme capability aplikaci, které úplně nevěříme, pak máme jistotu, že nemůže napáchat škody mimo svěřené zdroje.

O následující vlastnosti máme zájem, ale kompromisy jsou přijatelné v případě problémů při implementaci:

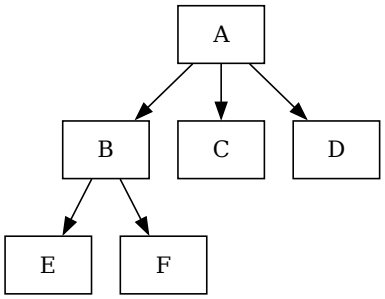
- **Dodatečný požadavek č. 1**: capability lze zrušit (revokovat).

  Revokace je nástroj násilného odebrání oprávnění ke zdroji. Protože u exokernelu chceme dovolit nízkoúrovňový přístup k hardwaru i obyčejným (tj. ne speciálně privilegovaným a důvěřovaným) aplikacím, musíme počítat s tím, že aplikace může odmítat se svěřených zdrojů vzdát. Potom musíme mít nějaký mechanismus, skrze který původní vlastník oprávnění může oprávnění zlobivé a nespolupracující aplikaci zrušit.

  Toto není tak jednoduché, jak se zdá. Komplikací jsou zde DMA operace. Co když do úseku fyzické paměti reprezentované právě revokovanou capabilitou zrovna zapisuje nějaké zařízení? Pokud se rozhodne původní vlastník tohoto úseku paměti po revokaci použít tento úsek paměti pro jiné účely, zařízení mu tu paměť může dál přepisovat. Přerušit takovou hardwarovou operace ze strany exokernelu nelze. Má operace revokace selhat nebo počkat? Jak vůbec bude exokernel schopen zjistit takovou kolizi? Tato problematika je dále rozvedena v podkapitole 3.5.4.

Z těchto požadavků vyplývá, že capability budou tvořit jakési stromy dědičnosti. Prohlédněme si moji první iteraci návrhu.

3.2.1 První iterace



Obrázek 3.1 Strom capabilit odvozených od A

V předchozím grafu je každý uzel *capability slot*, tedy kernelspace datová struktura popisující jednu capabilitu (tj. oprávnění aplikace k nějakému zdroji). Každý potomek slotu je slot reprezentující, v souladu s invariantem č. 1, podmnožinu zdroje reprezentovaného rodičovským slotem. Například pokud rodič popisuje rozsah fyzické paměti 0x1000 - 0x5000, pak potomek může reprezentovat například 0x2000 - 0x3000. Takovou capabilitu budeme ve zbytku textu nazývat jako odvozenou capabilitu.

Povšimněme si, že žádný invariant neříká, že by capability odvozené od stejného rodiče nemohly reprezentovat navzájem se překrývající rozsahy zdrojů. Specifické typy reprezentovaných

zdrojů si mohou definovat dodatečné invarianty, které mohou vytváření odvozených capabilit dále omezovat, podle potřeby. Toto zatím nemusíme neřešit.

Dodatečný požadavek č. 1 můžeme splnit tak, že při revokaci nějaké capability nejdříve provedeme revokaci všech jejích potomků. Ti opět rekurzivně provedou revokaci svých potomků. Výsledek je ukončení existence capability, kterou jsme chtěli revokovat, a celého jejího podstromu odvozených capabilit.

První iterace splňuje všechny 3 invarianty i dodatečný požadavek č. 1. Pro zajímavost, invariant č. 3 implikuje, že kořen stromu je vždy prapůvodní capabilita vytvořená při bootu. Napravuje tím tedy výše zmíněný nedostatek č. 1. Co nedostatek č. 2? Odpověď záleží na implementačních detailech.

Výpis kódu 3.1 První iterace capability slotu

```
struct cap_slot{
  struct cap_slot *parent;
  struct cap_slot *children;
  size_t children_count;
  uint8_t capability_type;
  // more fields would go here to
  // describe memory ranges, disk block ranges etc.
  // ...
};
```

Pokud bychom se rozhodli strom capabilit implementovat jako ukazatel na rodiče a pole ukazatelů na potomky pevné velikosti zabudované uvnitř capability slotu, měli bychom horní omezení na počet capabilit odvozených od jedné rodičovské capability. Pokud bychom se rozhodli implementovat ukazatele na potomky jako dynamicky alokované pole (či jiná zvětšující se datová struktura), nedostatek č. 2 by nebyl napraven, protože k alokacím paměti by docházelo (z pohledu userspacových systémových volání) nedeterministickým způsobem. Jak již jsem zmínil dříve v podkapitole 3.1.4, pro řešení nedostatku č. 2 bych chtěl použít řešení ve stylu seL4.

Implikací je, že paměť pro uložení spojení mezi rodičem a potomkem musí být poskytnuta uvnitř potomka samotného. Potom bychom neměli horní omezení na počet potomků, protože s každý přidaným potomkem máme další paměť pro reprezentaci nového spojení. Zároveň by nedocházelo k nedeterministickým alokacím, protože pro každé přidání potomka potřebuje exokernel vždy stejné množství paměti - velikost datové struktury capability slotu. Nehledě na stav stromu a sekvenci předchozích operací.

To znamená, že strom dědičnosti bude muset být reprezentovaný nějakou formou spojového seznamu. Jen tak můžeme mít kardinalitu 1:n mezi rodičem a potomky bez růstu potřebné paměti uvnitř rodiče.

3.2.2 Druhá iterace

Výpis kódu 3.2 Druhá iterace capability slotu

```
struct dlist{
  struct dlist *next;
  struct dlist *prev;
};
struct cap_slot{
  struct cap_slot *parent;
  struct dlist children;
  struct dlist self_as_child;
  uint8_t capability_type;
  // more fields would go here to
  // describe memory ranges, disk block ranges etc.
  // ...
```

```
};
```

Parent ukazuje na rodičovskou capabilitu (*NULL* u prapůvodních capabilit), *children* je dvojitý spojový seznam jehož prvky jsou *self_as_child* v odvozených capabilitách. *capability_type* a následující prvky struktury už popisují samotný reprezentovaný zdroj.

Tato možnost implementace umožňuje splnění všech našich invariant a napravuje nedostatek č. 1 i 2. Má už jen jednu malou vadu na kráse - capability slot je vcelku velký. Při použití 64-bitových ukazatelů výše popsaná struktura využívá 41 bajtů. Na jednu stranu bychom mohli reprezentaci ukazatelů trochu optimalizovat, protože fyzický adresní prostor AMD64 je omezen na maximálně 52 bitů, přičemž v současnosti skutečně existující hardware typicky podporuje 40 či 48 bitů. Na druhou stranu struktura zatím neobsahuje žádné prvky popisující reprezentovaný zdroj.

Počet capabilit v systému může být obrovský, protože všechny zdroje budou reprezentovány v capability systému. Pokud například jedna aplikace předá druhé aplikaci množinu capabilit pro přímý přístup k souboru na disku, počet capabilit může být ve statisících u velkého fragmentovaného souboru. Podobný problém bude u správy virtuální paměti, který si kvůli revokacím musí pamatovat každý namapovaný paměťový rámec, aby při revokaci byl odmapován ze stránkovacích tabulek. Nezapomeňme, že počet virtuálních adresních prostorů roste s počtem běžících procesů. V systému tedy mohou být v jeden moment dohromady klidně i miliony capabilit.

Podíval jsem se na řešení jiných capability-based kernelů a seL4 a jeho klony tento problém vyřešily elegantním způsobem. Strom dědičnosti capabilit není implementovaný tolik jako strom jako mé první řešení, ale je tzv. linearizovaný do jednoho spojového seznamu.

3.2.3 Třetí iterace (CDT)

Řešení stromu dědičnosti používané mikrokernely seL4 a Barrelfish je nazýváno *capability derivation tree*, dále bude označováno jen jako CDT.

Výpis kódu 3.3 Capability derivation tree

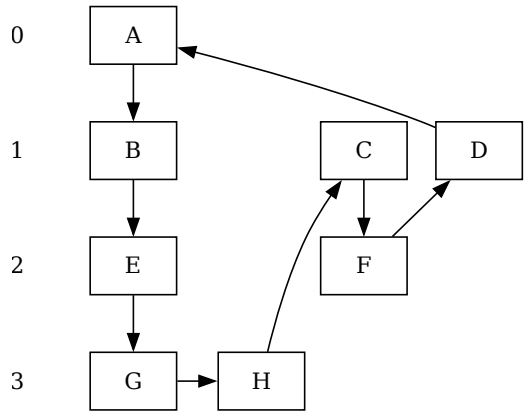
```
struct cap_slot{
  struct cap_slot *prev;
  struct cap_slot *next;
  uint8_t depth;
  uint8_t capability_type;
  // more fields would go here to
  // describe memory ranges, disk block ranges etc.
  // ...
};
```

Jeden dvojitý spojový seznam propojuje všechny capability (tranzitivně) odvozené od prapůvodní capability. Informaci o rodičích a potomcích udržuje prvek *depth*, který popisuje hloubku capability ve stromu dědičnosti. Potomci budou mít *depth* o 1 vyšší než rodič. Prapůvodní capability bude mít *depth* s hodnotou 0.

Potomci leží v CDT vždy směrem *next* od rodiče. Směrem *next* tedy najdeme buď svého potomka, sourozence, sourozence rodiče (či obecně sourozence nějakého předka) nebo prapůvodní capabilitu. Směrem *prev* najdeme buď svého rodiče, sourozence nebo sourozencovi potomky (či jejich potomky). Zvláštním případem je prapůvodní capabilita bez žádných potomků, kde pak *next* a *prev* ukazují na sebe (tj. prázdný dvojitý spojový seznam).

Každý uzel v grafu je capability slot. Číslo vlevo je hodnota hloubky capabilit na stejné vodorovné úrovni jako dané číslo. Hrana ukazuje šipečkou ve směru *next*.

Capabilita A je prapůvodní. Od A byly odvozeny 3 capability: B, C a D. Od B bylo odvozeno E. Od C bylo odvozeno F. D žádného potomka nemá. Od E byly odvozeny 2 capability, G a H.



Obrázek 3.2 Capability derivation tree (seL4)

Oproti druhé iteraci nám odpadlo několik ukazatelů a tato struktura využívá jen 18 bajtů. Tedy zhruba polovinu. Některé vlastnosti jsou ale horší. Například nelze v konstatním čase najít svého rodiče. K tomu je třeba opakovaně procházet ukazatel *prev*, dokud nenalezneme capabilitu s nižším *depth* než naše. Takovou operaci ale v praxi potřebovat nebudeme.

Úryvek kódu výše není skutečnou strukturou používanou seL4, jedná se jen o ilustraci principu. Skutečná implementace v seL4 nemá samostatné pole pro hloubku, ta je uložena v dolních bitech *prev* a *next*, které jsou jinak nevyužity díky zarovnání slotu v paměti[19]. Místo *depth* a *capability_type* má dva 4-bajtové nebo 8-bajtové prvky, *objptr* a *cap_data*[19]. První obsahuje data specifická pro typ reprezentovaného zdroje a druhý obsahuje typ a další informace, jako například jaká konkrétní oprávnění z reprezentovanému zdroji tato capabilita drží. To může znamenat například právo mapovat paměťový úsek do virtuální paměti jen pro čtení nebo i pro zápis a podobně.

Při srovnání mých 2 iterací a řešení seL4 (CDT), jsem se rozhodl pro implementaci capability systému pokračovat v CDT z důvodu snížení paměťového overheadu.

V tento moment už můj kernel nabírá určitou podobu. Na rozdíl od seL4 není mikrokernelem, tedy ovladače jsou monoliticky součástí kernelu. Systém capabilit bude silně inspirován seL4 (avšak ne identický). Celkový cíl je poskytnutí exokernelového API, tak jak bylo definováno v podkapitole 2.4. To je něco, čím se bude fundamentálně lišit od seL4 a jeho klonů; seL4 či Barrelfish rozhodně nereprezentují přístup ke klávesnici nebo rozsahům diskových bloků či zlomku diskového bandwidth ve svém capability systému. Reprezentují pouze základní prvky jako CPU čas, fyzickou paměť a např. rozsahy I/O portů. Ovladače jsou implementovány jako userspace programy, které s vnějškem komunikují po vlastních IPC protokolech. Tedy produkt ovladačů stojí mimo capability systém.

Ale můj exokernel, na rozdíl od těchto mikrokernelů, bude reprezentovat všechny hardwarové zdroje a operace v jednotném capability systému, díky čemuž lze delegovat přístup ke všemu hardware granulárně a s revokacemi, což u rodiny seL4 nelze.

Můj projekt je tedy jakési unikátní skloubení capability systému rodiny seL4 a myšlenek a cílů exokernelu. Myslím si, že se navzájem, z uvedených důvodů v předchozích stránkách, až překvapivě dobře doplňují.

3.2.4 Implementace slotu

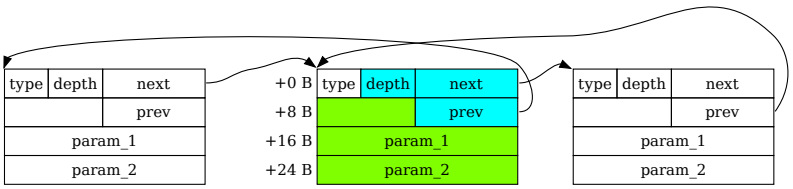
Výpis kódu 3.4 Finální verze capability slotu

```
struct cdt_slot{
  uint64_t next;
  uint64_t prev;
} __attribute__((aligned((16))));
struct cap_slot{
  struct cdt_slot cdt_slot;
  uint64_t param_1;
  uint64_t param_2;
};
```

Mé řešení je rozděleno do dvou struktur.

První, *cdt_slot*, reprezentuje pouze uzel v CDT. *next* a *prev* tvoří dvojitý spojový seznam. Hloubka capability v CDT je stejně jako u seL4 uložena v nevyužitých bitech spojového seznamu. Konkrétní implementace je ale velmi jiná. Protože můj kernel se omezil na podporu fyzického adresního prostoru o velikosti 1 TiB, mám k dispozici agresivnější optimalizaci. Libovolná adresa se vejde do 40 bitů a *cdt_slot* je vždy zarovnán na 16 bajtů, což znamená, že ať už je *cdt_slot* kdekoliv v paměti, jeho fyzická adresa se vejde do 36 bitů. Takže do pole *next* se vejde ještě 10 bitů hloubky v CDT a 18 bitů pro číselný kód typu capability. V *prev* zůstane $64 - 36 = 28$ bitů nevyužitých. Tyto mohou být využity pro účely specifické pro typ reprezentovaného zdroje.

Prvky *param_1* a *param_2* jsou celé volné pro typově specifické účely. Například capabilita reprezentující úsek fyzické paměti by si do *param_1* mohla uložit fyzickou adresu začátku úseku a do *param_2* délku úseku v bajtech a oprávnění (čtení, zápis); capabilita reprezentující vlákno by v *param_1* měla ukazatel na kernelspace objekt vlákna. Bylo otázkou, jak velký prostor takto rezervovat. Na jednu stranu chceme, aby byl co nejmenší, protože bude zabírat místo i v capabilitách, jejichž typ tento prostor nevyužívá. Na druhou stranu chceme, aby co nejvíce typům capabilit stačil, protože jinak by takové typy capabilit musely používat externí datové struktury. Což by v kombinaci s deterministickou alokací paměti 3.1.4 znamenalo složitější API pro userspace.



Obrázek 3.3 Capability slot

Modrá pole jsou dvojitý spojový seznam, který spolu s *depth* tvoří uzel v CDT. Zelená pole jsou volná a využitelná k ukládání typově specifických dat.

Prázdný slot má *type* nastaven na speciální hodnotu tento fakt indikující a jeho *prev* a *next* nejsou zapojeny do žádného CDT.

Zajímavou náhodou je fakt, že velikost stránky na AMD64 je 4096 bajtů, což znamená, že adresa zarovnaná na stránku má dolních 12 bitů nevyužitých. Spolu s faktem, že můj kernel je omezen na 40-bitový fyzický adresní prostor, to znamená, že adresa libovolného paměťového rámce se vejde do $40 - 12 = 28$ bitů. Výsledkem je, že do jednoho 64-bitového ukazatele mohu vměstnat jeden ukazatel na *cdt_slot* a jeden ukazatel na paměťový rámec zároveň, protože $36 + 28$ je přesně 64. Celkem se do *cap_slot* vejdou 2 ukazatelé v rámci spojového seznamu CDT, hloubka,

typ, a například 3 ukazatelé na paměťový rámec a 2 ukazatelé na *cdt_slot* či jiný objekt zarovnaný alespoň na 16 bajtů. To je celkem až 7 ukazatelů, hloubka a typ v jednom pouze 32-bajtovém objektu. Takový prostor byl dostačující pro všechny typy capabilit v tomto projektu; některé ho využily naplno.

Pro srovnání, velikosti capability slotů různých capability-based kernelů:

	IA-32	AMD64
seL4	16 B	32 B
Barrelfish	64 B	64 B
FleK	nepodporováno	32 B

[17, str. 11]

U mého exokernelu byl tlak na množství uložených dat mnohem vyšší z důvodu reprezentace složitějších typů hardwarových zdrojů, které mikrokernely neřeší. Například capabilita reprezentující rozsah bloků na NVMe disku potřebuje uložit ukazatel na kernelspace objekt příslušný k danému disku, počáteční blok rozsahu, počet bloků rozsahu a 4 bity oprávnění. Stejně tak byl tlak na co nejmenší capability slot vyšší než u mikrokernelů, protože exokernel bude operovat s větším počtem capabilit než mikrokernely.

V případě, že v budoucnosti bude potřeba podporovat více než 1 TiB fyzické paměti, bude nutné toto optimalizované kódování změnit a *cap_slot* zvětšit. V současnosti není třeba plýtvat paměť velkými sloty, když 1 TiB paměti jsou pro PC stále v nedohlednu, a tak je toto skvělý trade-off.

Závěrem, důvod, proč je *cdt_slot* vyčleněn z *cap_slot*, bude objasněn v kapitole 3.10.6. Prozatím postačí říct, že je to forma optimalizace paměťového overheadu v mém systému správy virtuální paměti, protože *param_1* a *param_2* tam jsou nevyužité.

3.2.5 Operace v CDT

Někdy budeme potřebovat CDT procházet. Nejdříve se podívejme na časovou náročnost různých operací. Proměnná n zde značí počet capabilit v příslušném CDT.

- Ověření, zdá má potomky: O(1)

  Stačí se podívat slot v *next*. Pokud je jeho hloubka vyšší než naše, je to náš nejnovější potomek a odpověď je ano. Pokud má stejnou nebo nižší hloubku, víme, že žádné potomky nemáme a odpověd je ne.

- Vytvoření potomka: O(1)

  Potomek je vložen do CDT hned za rodiče ve směru *next*, s hloubkou vyšší o 1.

- Nalezení rodiče: O(n)

  Musíme procházet ve směru *prev* než nalezneme první capabilitu s hloubkou nižší než naše.

- Zjištění, zda si jsou dvě capability sourozenci

  - Pokud nejsou sousedící: O(n)

    Zde prostě musíme procházet CDT ve směru *prev*, dokud nenajdeme capabilitu s nižší hloubkou, a ověřovat každou mezitím nalezenou capabilitu se stejnou hloubkou. A pak znovu to samé ve směru *next*. Toto je velmi pomalé, protože musíme procházet i skrze potomky (a jejich potomky rekurzivně) sebe a sourozenců.

  - Pokud jsou sousedící: O(1)

    Stačí ověřit, že pomocí *next* či *prev* na sebe obě capability přímo ukazují. Další podmínkou je, že mají stejnou hloubku.

V praxi je cokoliv horší než O(1) nepřijatelné, protože škodlivá aplikace by mohla vytvářením odvozených capabilit v určitých vzorech provádět *denial of service* proti jiným aplikacím i proti kernelu.

Důvod, proč popisuji ověření sourozenectví sousedících jako speciální případ je fakt, že v praxi budeme chtít testovat relaci sourozenectví jen pro sousedy v CDT. Tak je třeba mít konstantní časovou náročnost této operace na paměti.

Tato operace je často užívána mým kernelem pro "spojování"dvou sousedících capabilit. Například pokud máme capabilitu reprezentující diskové bloky 10-19, kterou rozdělíme na dvě capability 10-14 a 15-19. Pokud je následně budeme chtít spojit do původní 10-19, tak skrze právě tuto grafovou operaci ověříme, že mají sousedící rozsahy bloků a zároveň stejného rodiče. Stejného rodiče potřebujeme ověřit, jinak by spojením byl narušen invariant č. 1. Relace sourozenectví implikuje stejného rodiče. Vzpomeňme, že přímé zjištění rodiče je O(n), a tak je hezké dvě takové operace obejít jednou O(1).

Kdyby byly sourozenci, ale nesousedily v CDT, tak by buďto nesousedily ani jejich rozsahy bloků nebo by levý sourozenec měl potomky. V prvním případě není problém, protože pak by nebylo, co vlastně spojovat (výsledek musí být souvislý rozsah bloků). Druhý případ je defektem v mém návrhu, protože taková operace by nenarušovala žadný invariant, ale v praxi nemůžeme riskovat O(n) operace. Za zmíňku stojí fakt, že moje *druhá iterace* 3.2.2 návrhu capability systému by si zde vedla lépe než toto řešení inspirované seL4, protože by nebyla nucena zbytečně procházet i potomky levého sourozence. Mohla by sáhnout do levého sourozence přímo pomocí *prev* v *self_as_child* v O(1) čase.

3.2.5.1 Počítání referencí

V předchozích podkapitolách byla nakousnuta možnost capabilit reprezentovat kernelspace datové struktury, které jsou stvořeny v darované paměti. Konkrétní detaily těchto vytvářecích operací budou vysvětleny v kapitole 3.5. Prozatím stačí říct, že capabilita reprezentující nějaký kernelspace objekt jednoduše drží ukazatel na ten objekt třeba v *param_1*.

Potom ale vyvstává otázka. Pokud máme capabilitu ukazující na kernelspace objekt a z této capability vytvoříme odvozenou capabilitu (tj. reprezentující stejný zdroj), pak budeme mít dvě různé capability odkazující na stejný objekt. Jak potom ale vědět, kdy je třeba daný kernelspace objekt destruovat, tedy odstranit reference na něj v jiných kernelspace strukturách? Kupříkladu, pokud aplikace revokuje capabilitu reprezentující TCB objekt (vlákno), a původní fyzickou paměť (kde tato TCB existovala) se rozhodne použít pro jiné účely. Kernel musí zajistit, že daný TCB objekt byl nejdříve odpojen z datových struktur plánovače úloh.

Musíme nějakým způsobem zjistit, zda revokovaná capabilita je poslední capabilita ukazující na daný objekt. Tehdy and jen tehdy je třeba objekt deinicializovat. Možností by bylo držet čítač referencí uvnitř samotného objektu. Při vytvoření odvozené capability by se čítač zvýšil o 1, při revokaci by se snížil o 1. Při poklesu na 0 víme, že už neexistuje žádná capabilita reprezentující tento objekt.

Existuje ale ještě jednoduší řešení. Díky invariantu č. 1 víme, že všechny capability odvozené od capability reprezentující nějaký kernelspace objekt budou také reprezentovat ten stejný objekt. Potom tedy při revokaci stačí ověřit, že zda náš soused v CDT ve směru *prev* má stejný typ a ukazuje na stejný objekt. Pokud ne, jsme (z hlediska dědičnosti) první capabilitou reprezentující daný objekt. Kombinace předchozího zjištění a invariantu č. 2 implikuje, že jsme poslední capabilitou ukazující na daný objekt a musí tedy teď dojít k destrukci objektu. Invariant č. 2 nám totiž zajišťuje, že všechny z nás odvozené capability již byly revokovány před námi. Pokud jsme první a zároveň poslední, jsme jediní a s naší revokací tedy končí poslední reference na daný objekt.

Podobné řešení používá seL4 a jemu podobné kernely, čímž jsem se inspiroval. Tam je situace trochu složitější v důsledku možnosti vytváření odvozených capabilit takovým způsobem, že nová capabilita může být potomkem či sourozencem původní. Aplikace si může vybrat z těchto dvou

možností odvození. Můj capability systém, na rozdíl od seL4, nedovoluje vytváření odvozenin jako sourozenců, protože výsledný systém bez této možnosti považuji za mnohem elegantnější a pro člověka předvídatelnější a přehlednější.

3.2.6 Překlad indexů

V předchozí kapitole byl popsán způsob reprezentace databáze capabilit v kernelu, ale jakým způsobem budou capability prezentovat aplikace při systémových voláních? Pochopitelně nemohou kernelu předávat paměťovou adresu capability slotu, protože kernel musí být ve virtuálním adresním prostoru izolován z bezpečnostních důvodů. Budeme proto potřebovat nějaký zástupný číselný identifikátor.

Můžeme se podívat, jak stejný problém řeší Unix. Aplikace při systémových voláních např. *read()* v určitém procesorovém registru nastaví tzv. *file descriptor*. To je jen číselný identifikátor, který je kernelem přeložen na ukazatel na kernelspace datovou strukturu popisující otevřený soubor.

Stejné řešení použijeme i zde. Aplikace budou na své capability při systémových voláních odkazovat číselnými identifikátory, dále označované jako *capability index*.

Zbývající otázkou je, kde bude exokernel ukládat datové struktury (sloty), jak je bude vytvářet a jak bude překládat indexy na sloty. Když se znovu poohlédneme na Un*xové systémy, tentokrát na Linux do verze 2.2, uvidíme, že v datové struktuře reprezentující proces (*struct task*) bylo pole pevně dané velikosti (NR_OPEN, tehdy 1024)[20]. Prvky pole byly ukazatele na datové struktury popisující otevřené soubory (*struct file*) nebo *NULL*. File descriptory potom byly jednoduše indexy v tomto poli[4, str. 480].

Od verze 2.2 je pole ukazatelu v Linuxu dynamicky realokované, když se zaplní. Nyní si vzpomeňme na podkapitolu 3.1.4. Pokud bychom naše pole během vkládání při plném stavu realokovali, spotřeba paměti by nebyla z pohledu userspace deterministická - spotřeba paměti pro splnění systémového volání by závisela na userspacu skrytém vnitřním stavu kernelu. Pak by nešlo použít řešení ve stylu seL4, že kernel nic nealokuje a aplikace daruje paměť při systémovém volání.

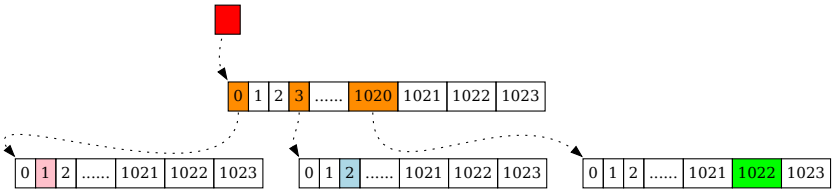
seL4 tento problém řeší tak, že na rozdíl od file descriptorů je za správu capability indexů zodpovědná aplikace a ne kernel. Na Unix-like systémech např. při systémovém volání *open()* je nový file descriptor nalezen kernelem a předán aplikaci v návratovém procesorovém registru. seL4 místo toho nové indexy nikdy nevrací. Při vytváření nějaké nové capability musí aplikace říct kernelu, kam ji má uložit. Na Unix-like systémech neříkáme kernelu, jaký nový deskriptor použít (s výjimkou *dup2()*). Aplikace pod seL4 si musí pamatovat, které indexy jsou zatím nevyužité.

U seL4 překlad indexu na capability slot nefunguje tak jako u Unixu (index v poli ukazatelů), ale jedná se o průchod několika úrovní tabulek (v terminologii seL4 *CNode*), kde každá tabulka je pole capability slotů (ne ukazatelů na capability sloty)[17, str. 15]. Jde o velmi podobný proces jako překlad adresy ve virtuálním adresním prostoru na adresu ve fyzickém adresním prostoru pomocí stránkovacích tabulek (*page tables*). Rozsah možných indexů vlastně tvoří adresní prostor a v terminologii seL4 je nazýván *capability space*, neboli *CSpace*. Dva různé capability space spolu mohou sdílet podmnožinu svého prostoru tím, že mají oba někde ve stromu tabulek namapovanou stejnou tabulku. Výsledek je něco jako sdílená paměť u virtuální paměti.

Alokace nových capability slotů má formu alokace jedné nové tabulky a její vložení do tabulky vyšší úrovně. Jakým způsobem je ale nová tabulka vlastně "alokována"? Pamatujme, že seL4 paměť nikdy nealokuje (což je pro náš exokernel s userspace disk cachí velmi žádoucí), takže nemůže existovat systémové volání "alokuj mi novou tabulku a vlož ji do X". Jak již bylo dříve vysvětleno, pokud systémové volání potřebuje pro svoje vykonání paměť, musí aplikace prezentovat jinou capabilitu reprezentující potřebné množství paměti (jakýsi dar kernelu). Existuje tedy systémové volání "zde máš kus fyzické paměti, vytvoř v ní novou tabulku a vlož ji do X". Capabilita reprezentující novou tabulku je pak sama součástí stromu dědičnosti jako odvozená capabilita, jako potomek capability fyzické paměti.

U mikrokernelu Barrelfish je velikost tabulky proměnná a aplikace si tedy může vytvářet velké či malé tabulky dle uvážení[21, str. 6]. Malé tabulky znamenají vyšší maximální hloubku stromu tabulek, což obecně znamená menší paměťový overhead ale vyšší výpočetní overhead. Cache miss je zde problémem. Tabulky různých velikostí mohou být libovolně kombinované v rámci jednoho stromu tabulek. Toto přidává výpočetní overhead, jak v kernelu při překladu capability indexu tak v userspace, přičemž benefit je diskutabilní.

A tak jsem se u svého exokernelu rozhodl pro tabulky pevně dané velikosti. Konkrétně 1024 (2^{10}) capability slotů v jedné tabulce a strom tabulek je dvouúrovňový. To znamená, že adresní prostor capability indexů je 20-bitový, protože máme 2 úrovně po 10 bitech indexace v úrovni. Vyšších 10 bitů určuje capability slot v tabulce nejvyšší úrovně. Tento capability slot (pravděpodobně) reprezentuje další tabulku a v té znovu indexujeme nižšími 10 bity. Výsledný capability slot je ten, na který se userspace odkazuje. Maximální platná hodnota capability indexu je proto $2^{10+10} - 1 = 1.048.575$. Jedna aplikace pod mým exokernelem tedy nemůže mít více capabilit než 1.048.576, toto je absolutní strop.



Obrázek 3.4 Capability space FleKu

Předcházející obrázek popisuje capability space skládající se ze 4 capability tables (dále *CT*). Jeden CT vyšší úrovně a tři CT nižší úrovně, dohromady tvořící dvouúrovňový strom. Každé políčko je jeden capability slot, jak byl popsán v 3.2.4. Oranžová políčka a červené políčko jsou sloty obsahující capability typu CT. Tečkované hrany ilustrují cíl ukazatele v prvku *param_1*. Červené políčko je capability slot též typu CT, sloužící jako kořen našeho capability space. Slot ilustrovaný červeným políčkem by mohl být například zabudovaný do datové struktury vlákna (TCB), čímž by určoval capability space toho vlákna. Červený slot si můžeme představit jako ekvivalent procesorového registru *CR3* u virtuální paměti.

Z obrázku vidíme, že capability index o hodnotě 1 bude vyhodnocen jako ukazující na růžový capability slot. Modrému slotu přísluší index $3 * 1024 + 2 = 3074$. Zelenému $1020 * 1024 + 1022 = 1.045.502$.

Můj exokernel nijak nebrání do capability slotů v tabulce vyšší úrovně vkládat capability i jiného typu než tabulka (nižší úrovně). Pokus překládat index skrze takový slot potom selže a aplikace dostane chybový kód. Stejně tak dopadne pokus překládat skrze prázdný slot v tabulce vyšší úrovně (”prázdný slot”je ve skutečnosti prostě jen typ capability).

3.3 Systémová volání

Formát systémových volání se lehce podobá Unixovým - jeden registr popisuje operaci k provedení a několik dalších registrů obsahuje argumenty operace. Od Unixových syscallů se liší tím, že registr popisující operaci neobsahuje pouze identifikační číslo operace, ale funguje na jakémsi objektovém principu. Obsahuje capability index a identifikátor operace, relativně k typu capability, která se má provést nad určenou capabilitou. Některé operace existují pro všechny capability, ty mají stejná čísla nehledě na typ capability.

Registr popisující operaci bude nadále nazýván *command registrem*. Je jím registr *RDI*. Registrů obsahujících argumenty je pět - *RSI*, *RDX*, *R10*, *R8* a *R9*, v tomto pořadí.

Výběr těchto registrů není náhodný, je důsledkem hardwarového návrhu instrukce *syscall*[22, str. 174], která na AMD64 nahrazuje starší způsob skoku do *ring 0* pomocí softwarového vyvolání přerušení instrukcí *int*. Instrukce *syscall* zkopíruje registr *RIP* (instruction pointer) do registru *RCX* a registr *RFLAGS* do registru *R11*. Tudíž obsah těchto registrů je ztracen.

Dále je výběr registrů důsledkem ABI System V[23] (standard Unixových systémů), který jsem vybral pro svůj OS. SysV ABI definuje registry *RAX*, *RDI*, *RSI*, *RDX*, *RCX*, *R8*, *R9*, *R10*, *R11* jako tzv. scratch registry. Tyto jsou při volání funkce považovány volající funkcí za ztracené. Registry *RBX*, *RSP*, *RBP*, *R12*, *R13*, *R14*, *R15* jsou tzv. callee saved. Volaná funkce musí jejich obsah zachovat. I pokud je volající funkce nepoužívá – to volaná funkce nemůže vědět. Proto jsou scratch registry vhodné pro systémová volání. Userspace funkce obalující systémová volání pro vysokoúrovňovější kód si ušetří několik *push/pop* instrukcí.

Někdy, vzácně, je potřeba i mnohem více než 5 argumentů. K tomu slouží tzv. *memory register set* (dále jen *MRS*), též inspirovaný mikrokernely rodiny L4[24, str. 50]. V mém kernelu se jedná o 4 KiB paměti zarovnané na 4 KiB (1 stránka), které se chovají jako další argumentové registry při systémových voláních. Protože registry mají 8 bajtů, je toto prostor pro dalších 512 registrů. Ovšem formát command registru FleKu omezuje počet registrů na 511 celkem, tedy při 5 argumentových registrech v CPU zbývá na MRS jen 506 registrů.

Skrze MRS lze předávat i datové struktury z/do kernelu; některá systémová volání tohoto využívají. V Unixových systémech se v takových případech předává skrze argumentový registr ukazatel do virtuální paměti procesu, kam kernel data zapíše (např. u fstat()). U našeho systému toto nelze provést, protože může dojít k page faultu a exokernel tyto neumí vyřešit, protože jsou řešeny v userspacu. MRS je speciálním místem, jehož **fyzická** adresa je uložena přímo v TCB 3.9, a tudíž při zápisu kernelem nemůže dojít k page faultu (kernel má namapovaný celý fyzický adresní prostor). Vlákno nemá povinnost MRS poskytovat, pak ale mohou systémová volání, která ho potřebují, selhat.

Konkrétní číselné kódy jsou poskytovány kernelem pomocí hlavičkového souboru obsahující makra, který mohou použít userspacové programy. Takové identifikátory budou v textu níže někdy použity.

Konkrétní formát command registru je následující, od nejnižších bitů po nejvyšší:

- Cílový capability index (20 bitů)

  Určuje, na kterou capability bude operace provedena. Interpretace capability indexu byla vysvětlena v kapitole 3.2.6.

- *Moving capability* (dále jen jako *movcap*) index (20 bitů)

  Je možné posílat svoje capability jiným vláknům. Toto pole specifikuje index capability k poslání (na straně odesílatele), resp. index, kam bude přijata (na straně přijímatele)[1].

- Movcap bit (1 bit)

  Pokud je nastaven, bude capabilita z movcap pole poslána, resp. přijata. Jinak je movcap pole ignorováno.

[1] Tuto funkcionalitu jsem nakonec nestihl implementovat. Nicméně je součástí návrhu.

- Počet argumentů (9 bitů)

  Počet argumentových registrů, které při systémovém volání budou vzaty v úvahu. Ačkoliv většina syscallů má jasně daný počet argumentů, některé mají proměnlivý. Proto je toto pole nezbytné.

- Label (10 bitů)

  Identifikátor toho, co chceme, aby bylo s capabilitou provedeno.

- Operace (4 bity)

  Tento formát systémového volání je zároveň použit i pro IPC, proto je toto pole mimojiné nutné k odlišení, zda jen posíláme zprávu jinému vláknu skrze endpoint nebo je to systémové volání manipulující se samotným endpointem (protože v obou případech je index capability tato endpoint capabilita). Bude blíže vysvětleno v 3.8.

V případě skutečného systémového volání (tj. ne IPC) je hodnota pole *operace* nastavena na *FLEK_SYSCALL_OP_METHOD* (0) a pole *label* na identifikátor syscallu. Kernel v případě úspěšného provedení systémového volání ponechá hodnotu command registru, jak byl nastaven userspacem, a v argumentových registrech vrátí hodnoty, pokud je to žádoucí. V případě neúspěchu nastaví pole *operace* na hodnotu *FLEK_SYSCALL_OP_ERROR* (15) a pole *label* na návratový kód označující chybu. Existuje 6 dalších hodnot pole *operace*, které jsou použity výhradně při interakci s capabilitou typu endpoint. Ty jsou vysvětleny v sekci 3.8.

3.4 Společná systémová volání

Jak bylo zmíněno v předcházejícím textu, existuje množina systémových volání, která je společná pro všechny typy capabilit.

3.4.1 Get type

label	FLEK_SYSCALL_ANY_GET_TYPE (3)
argumenty userspacu	žádné
návratové hodnoty	číslo typu capability, která byla nastavena jako cíl systémového volání

Jednoduše vrátí číslo typu capability. Toto číslo je identické s číslem v poli *type* popsaném v podkapitole 3.2.4.

3.4.2 Has children

label	FLEK_SYSCALL_ANY_HAS_CHILDREN (4)
argumenty userspacu	žádné
návratové hodnoty	boolean

Odpoví, zda capability má či nemá odvozené capability.

3.4.3 Revoke

label	FLEK_SYSCALL_ANY_REVOKE (1)
argumenty userspacu	žádné
návratové hodnoty	žádné

Toto systémové volání smaže (revokuje) všechny odvozené capability od cílové capability. Samotná cílová capabilita smazána nebude.

Ne vždy je možné capability mazat. Například pokud capabilita reprezentuje userspacu přístupnou fyzickou paměť, do které zrovna zapisuje diskový řadič, pak nelze takovou capabilitu revokovat. Pokud bychom to umožnili, mohlo by nastat narušení stability a bezpečnosti systému - userspace by po revokaci mohl požádat v této fyzické paměti zkonstruovat kernelový objekt (například stránkovací tabulku) a diskový řadič by ji z druhé strany přepsal.

CDT je nejdříve procházen od cílové capability skrze její potomky až k poslední capabilitě, která je stále potomkem. Následně je CDT procházen v opačném směru, zpět k cílové capabilitě. Při druhém procházení už jsou capability revokovány. Tento průchod zpět totiž zajišťuje, že současná capabilita nikdy nemůže mít potomky. Pokud je měla, už jsme je při zpětném průchodu navštívili. Je to tedy iterativní algoritmus a ne rekurzivní, což je pro kernel důležité kvůli pevně dané velikosti zásobníku. Bohužel se jedná o algoritmus s lineární časovou náročnosti vzhledem k počtu odvozených capabilit.

Při prvním pokusu o revokaci capability, který selže, se celé systémové volání ukončí a je nastavena chybová hodnota toto značící.

3.4.4 Delete

label	FLEK_SYSCALL_ANY_DELETE (2)
argumenty userspacu	žádné
návratové hodnoty	žádné

Stejné jako FLEK_SYSCALL_ANY_REVOKE, ale navíc i cílová capabilita bude smazána.

3.5 Správa fyzické paměti

Fundamentálním zdrojem systému je fyzická pamět. Fyzickou paměť alokujeme pro mnohé účely - paměť užívaná userspacovým vláknem, paměť užívaná kernelovými datovými strukturami i paměť užívaná hardwarem (stránkovací tabulky, DMA buffery).

Klíčový rozdíl v přístupu k fyzické paměti mezi capability-based mikrokernelem jako seL4 a exokernelem jako FleK je fakt, že mikrokernel implementuje ovladače hardwaru v userspacu a proto nemusí jeho capability systém tolik předcházet nebezpečným situacím. Pokud se userspacový ovladač začne chovat chybně či škodlivě, seL4 nemůže systém ochránit a ani se o to nesnaží a není pro to navržen. Mikrokernely jako seL4 tedy například neřeší ochranu před revokováním capability paměti a znovupoužití této paměti, do které právě zapisuje DMA hardware; kernel prostě předá rozsahy portů a fyzického adresního prostoru userspacovým ovladačům a ty si v rámci jejich rozsahů mohou dělat cokoliv. Dokonce ovladače mohou říci diskovému řadiči, aby zapisoval mimo rozsahy fyzické paměti, které byly ovladači svěřeny kernelem[2]. *[I/O devices capable of DMA present a security risk because the CPU's MMU is by-passed when the device accesses memory. In seL4, device drivers run in user space to keep them out of the trusted computing base. A malicious or buggy device driver may, however, program the device to access or corrupt memory that is not part of its address space, thus subverting security]*[17, str. 65]. Takový mikrokernel o žádných discích a DMA operacích ani nic neví. Vyžaduje to tedy nemalou míru důvěryhodnosti userspacu, přinejmenším v některé privilegovanější z userspacových komponent.

Kdežto exokernel implementuje ovladače v kernelspace a musí být schopen poskytovat téměř přímý přístup k hardwaru **běžným** userspace programům, které nehledě na to co dělají, nesmí mít možnost narušit stabilitu a bezpečnost systém. Exokernel nemůže záviset na důvěryhodnosti žádné userspace komponenty. Pokud by exokernel vyžadoval důvěryhodnost, podkopávalo by to jeho účel - tedy umožnit **naprosto obyčejným** programům optimální využití hardwaru.

To výrazně zesložiťuje návrh a implementaci mého exokernelu v porovnání s mikrokernely.

Připomeňme si koncept deterministické alokace paměti kernelem. Pokud chce aplikace vytvořit nové vlákno, musí kernelu předat fyzickou paměť, kterou bude kernel používat pro uložení datové struktury vlákna (TCB = thread control block). Tedy aplikace musí mít capabilitu reprezentující fyzickou paměť a vytvořit z této capability odvozenou capabilitu typu TCB. Samotná datová struktura TCB bude uvnitř rozsahu fyzické paměti rodičovské capability. Toto vyžadují invarianty CDT č. 1 a č. 2. Navíc k invariantům CDT musíme přidat následující invarianty pro správu fyzické paměti:

- **Požadavek č. 1**: userspace nesmí mít přístup do existujících kernelových objektů

  Pokud v rozsahu fyzické paměti vytvoříme kernelové objekty, nesmí být možné tuto fyzickou paměť namapovat do dolní poloviny virtuálního adresního prostoru. V horní polovině virtuálního adresního prostoru je kernelspace a tam pochopitelně namapovaná být musí, aby objekty kernel mohl použít.

  Pokud by userspace mohl číst nebo dokonce zapisovat do fyzické paměti, ve které si nechal vytvoři kernelové objekty, došlo by k narušení bezpečnosti systému.

- **Požadavek č. 2**: nesmí být možné překrývat kernelové objekty čímkoliv jiným

  Pokud v rozsahu fyzické paměti vytvoříme kernelové objekty, není přijatelné, abychom například ve stejné paměti vytvořili ještě jiné kernelové objekty. Jinými slovy, pokud máme capabilitu reprezentující fyzickou paměť, a z ní odvodíme capabilitu typu TCB (což vytvoří TCB objekt v té paměti), pak nemůžeme z této capability reprezentující fyzickou paměť odvodit ještě druhého potomka, například capabilitu typu stránkovací tabulka. To by způsobilo přepsání objektu TCB a narušilo stabilitu systému.

[2]Tomuto lze zabránit použitím IOMMU, jenž seL4 používá, pokud ho počítač obsahuje.

- **Požadavek č. 3**: kernel nesmí nechávat citlivé informace v paměti darované userspacem po jejím navrácení

  Jestliže userspace zlikviduje capability reprezentující kernelové objekty a začne jejich fyzickou paměť zase používat pro svoje potřeby (tj. namapování do virtuální paměti), tak v této paměti nesmí zbýt žádné citlivé informace z doby, kdy ji ještě používal kernel.

  Toto lze zajistit jednoduše. Pokud je poslední capabilita reprezentující určitý kernelový objekt revokována, paměť, ve které objekt existuje, vynulujeme. To lze zajistit implicitním počítáním referencí, vizte kapitola 3.2.5.1. Dále se už o tomto požadavku nebudeme bavit.

Na první pohled se zdá, že splnění požadavku č. 1 a 2 by vyžadovalo obrovský výpočetní overhead. Jestliže z capability reprezentující fyzickou paměť mohu odvozovat libovolný počet dalších capabilit reprezentujících fyzickou paměť či kernelové objekty, tak jak potom může kernel při vytvoření nové capability reprezentující kernelový objekt ověřit, že se s ničím nepřekrývá? Pokud aplikace požádá o namapování capability reprezentující fyzickou paměť do virtuální paměti, jak může kernel efektivně ověřit, že v té fyzické paměti není žádný kernelový objekt? Zdánlivě je nevyhnutelné procházení potenciálně obrovského počtu uzlů v CDT pro ověření výše zmíněných požadavků.

Efektivní řešení existuje a dokonce má konstatní časovou náročnost. Spočívá v rozdělení konceptu "capabilita reprezentující fyzickou paměť"na několik podtypů capabilit reprezentujících fyzickou paměť. Následně operacemi popsanými v sekci 3.2.5 můžeme zajistit nemožnost porušení výše zmíněných požadavků jednoduchou kontrolou např. typů rodiče, existence potomků a podobně. Není pak potřeba procházet seznam úplně všech capabilit v systému ke zjištění, zda nedochází k nějaké kolizi. Tuto techniku používají capability-based kernely jako mikrokernel seL4 či multikernel Barrelfish.

3.5.1 Aegis

Englerovy exokernely nepoužívaly deterministickou alokaci paměti, takže userspace nikdy neřešil vytváření kernelových objektů. Všechny 3 výše zmíněné požadavky se tedy na ně nevztahovaly.

Správa fyzické paměti byla velmi jednoduchá. Každý paměťový rámec měl svoje identifikační číslo. Pro každý rámec byla vedena informace, komu zrovna patří. Jakékoliv rámce, které nikomu nepatřily, si mohly userspacové komponenty volně vzít.

Kernel mohl revokovat rámce, ale v důsledku tohoto jednoduchého řešení nemohly userspacové aplikace revokovat rámce jiným aplikacím.

3.5.2 seL4

Mikrokernel seL4 reprezentuje základní fyzickou paměť capabilitou typu *untyped memory*. Ta obsahuje počátek reprezentovaného rozsahu paměti a délku rozsahu; dále "počítadlo". Pokaždé, když je vytvořena nová odvozená capabilita, je její počátek offsetován počítadlem a počítadlo zvyšeno o její velikost. Počítadlo je resetováno, pokud všechny odvozené capability přestanou existovat[18].

Tím je zajištěn požadavek č. 2. Pak má druhý typ capability pro fyzickou paměť, *page*, který na rozdíl od všech ostatních typů capabilit může být mapován do virtuálního adresního prostoru. Page[17, str. 42] má velikost paměťového rámce (na AMD64 je to 4096 bajtů) a je odvozován z untyped memory postupem popsaným v předchozích větách. Tím je zajišten i požadavek č. 1.

3.5.3 FleK

Pro svůj kernel jsem vybral podobné řešení jako seL4, ale trochu pozměnené. FleK také reprezentuje fyzickou paměť typem capability jménem *untyped memory*. Tato capabilita nemůže být ma-

pována do virtuálního adresního prostoru, ale může být odvozována do dalších untyped memory (stejného rozsahu; podrozsahy nejsou povoleny), do capabilit reprezentujících kernelové objekty, či dělena a spojována. Ovšem pouze tehdy, pokud žádné odvozené capability zatím nemá. **Z toho vyplývá důležitý invariant: capabilita typu *untyped memory* nikdy nemůže mít potomky, které by se vzájemně překrývaly v rozsazích fyzické paměti.** Pokud chceme odvodit capability reprezentující kernelové objekty, dělá se to hromadně (tj. "vytvoř mi v sobě 50 instancí TCB"). Díky kontrole toho, že zatím žádné odvozené capability ještě nemá, víme, že nemůžeme narušit požadavky č. 1 a 2. A toto dokážeme zajistit v konstantním čase, protože se jedná o operaci ověření existence potomka v CDT (podkapitola 3.2.5). Takto si vystačíme i bez "počítadla", což zjednodušuje implementaci.

Stejně jako u seL4 má i FleK druhý typ capability pro fyzickou paměť a tím je *userspace memory*. Stejně jako Page u seL4 je vlastností *userspace memory* možnost jejího mapování do virtuálního adresního prostoru. Klíčovým rozdílem je však fakt, že *userspace memory* nemá fixní velikost a to velikost paměťového rámce, ale stejně jako u *untyped memory* je to rozsah fyzické paměti. Tento jeden velký rozsah pak může být mapován do virtuální paměti opakovaně, s různými offsety do něj. To znamená, že se FleK vyhne značnému paměťovému overheadu při mapování paměti, protože na rozdíl od seL4 nepotřebuje celou novou capabilitu pro každý jednotlivý namapovaný paměťový rámec. To je rozdíl i oproti exokernelu Aegis, jehož secure bindings databáze také reprezentovala paměťové rámce na individuální úrovni. Od capability typu *userspace memory* můžeme odvozovat capability reprezentující libovolné podrozsahy, i navzájem se překrývající. Tím se liší od *untyped memory* – zde si to už můžeme dovolit a neporušit přitom invarianty č. 1 a č. 2, protože kernelové objekty a *userspace memory* lze odvozovat pouze od *untyped memory*. Takže pokud už máme k dispozici *userspace memory*, tak už jen z její existence víme, že se její rozsah nemůže překrývat s nějakým kernelovým objektem.

Protože *userspace memory* lze mapovat do virtuálního adresního prostoru, je to vhodné místo pro nastavení oprávnění přístupu. Chceme mít možnost aplikaci předat oprávnění číst určitý rozsah fyzické paměti, ale přitom nemoci do něj zapisovat. Proto má *userspace memory* 3 bity oprávnění: pro čtení, pro zápis, pro vykonávání jako program. Tyto bity reflektují bity oprávnění ve stránkovacích tabulkách. Aplikace může namapovat *userspace memory* do stránkovací tabulky maximálně s těmi oprávněními, které jsou v té *userspace memory* capabilitě. Může si je ponížit samozřejmě.

3.5.4 DMA operace

Jak již bylo zmíněno v kapitole 3.2 u dodatečného požadavku č. 1 (v té kapitole), revokace capabilit a DMA operace jsou v konfliktu. Pokud exokernel umožní revokaci capability paměti, zatímco do této paměti probíhá DMA, tak dojde k narušení bezpečnosti i stability systému, protože userspace mezitím může použít stejný rozsah paměti např. pro vytvoření kernelových objektů.

Exokernel musí být schopen jednak detekovat, že daný úsek paměti zrovna podléhá DMA operaci, a dále musí odmítat revokace, dokud DMA neskončí. Jak toto zajistit a efektivně (ideálně O(1))?

Mikrokernely jako seL4 tuto otázku vůbec neřeší. Mikrokernely dávají přístup k HW zdrojům privilegovaným komponentám jako ovladače a tam se implicitně předpokládá rozumné chování; ne obyčejným nedůvěryhodným aplikacím.

Exokernel Aegis má v kernelspace pole deskriptorů paměťových rámců. Akademické práce nepopisují implementační detaily; předpokládám, že zrovna tam drží např. nějaký příznak o probíhajícím DMA do tohoto rámce. Exokernel Xok to takto řešil[13].

Exokernel FleK ale nemá žádné takové pole; právo nakládat s pamětí je reprezentováno capabilitami obsahujícími rozsahy adres. Mohli bychom držet nějaký takový příznak uvnitř capability slotu rozsahu paměti, kde zrovna probíhá DMA? Ne, protože stejný rozsah paměti může být reprezentován více než jednou capabilitou (např. odvozené podrozsahy). Pak by nebylo možné

(efektivně) udržovat tento příznak synchronizován ve všech takových capability slotech.

Přes několik iterací řešení tohoto problému (které nebudu popisovat) jsem eventuálně dosáhl perfektního řešení. Pokaždé, když si userspace požádá o DMA operaci (k čemuž musí přiložit capabilitu typu *userspace memory*), tak si příslušný hardwarový ovladač vytvoří odvozeninu té capabilitu paměti a tuto odvozeninu si uloží někam do sebe. Tato odvozená capabilita paměti má svůj vlastní typ capability (indikující DMA účel). Vzpomeňme, že revokace capability nejdřív automaticky způsobí revokaci všech jejích odvozenin. Pokud dojde k revokaci původní capability paměti, dojde nejdřív k revokaci i této odvozené DMA capability. **Revokace DMA capability vždy selže.** V ten moment je revokace původní capability paměti přerušena a chybový kód je navrácen tomu, kdo se o revokaci pokusil. Jakmile ovladač dostane od hardwaru indikaci, že DMA operace skončila, ovladač tuto vnitřní DMA capabilitu smaže. Pak už může capabilita paměti být zase revokována.

Toto řešení je velmi efektivní, protože revokace potomků v CDT se musí stát, tak jako tak, takže je toto v podstatě zadarmo. Zcela přirozeně zapadá do myšlenky CDT a jeho operací.

Ještě tady nastává jeden menší problém: škodlivá aplikace by mohla schválně v nekonečném cyklu vyvolávat DMA operace na skoro všechnu paměť, která ji byla svěřena. Výsledkem je forma *denial of service* útoku, protože takové aplikaci nikdy nebudeme schopni odebrat svěřenou pamět revokací. Mým řešením je prosté přidání nového bitu oprávnění capabilitě paměti (typu *userspace memory*), který umožňuje (či jeho absence znemožňuje) takovou paměť používat pro DMA operace. Problém to neřeší úplně, ale aspoň můžeme škodlivost aplikace nějakým způsobem omezit. Capabilita typu *untyped memory* má také tento bit; je to kvůli tomu, abychom měli jak zakázat aplikaci vytvářet *userspace memory* s povolujícím DMA bitem ze svěřené *untyped memory*.

Sekundární mechanismus potlačující tento problém je odebírání bitu oprávnění k DMA operacím všem paměťovým capabilitám, které jsou během revokace procházené. Tedy samozřejmě stále nelze aplikaci sebrat paměť, do které probíhá DMA operace, ale potom, co DMA skončí, už aplikace nebude schopna tuto paměť znovu použít k dalšímu DMA a opakovaná revokace někdy později už uspěje. Myslím, že v praxi by tato dvě opatření byly dostačující.

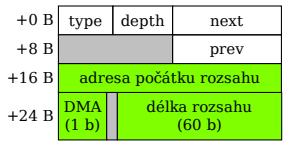
3.5.5 Nepřímé operace

Englerův operační systém na bázi exokernelu se snažil předejít potřebě exoserverů tím, že sdílený stav nebyl uzavřen uvnitř služby, ale otevřen všem aplikacím/libOS sdílenou pamětí. Protože přístupová práva ke sdílené paměti ověřuje hardware s granularitou velikosti stránky (na AMD64 je to 4096 bajtů), bylo toto příliš nepraktické pro sdílení datových struktur, tak aby aplikace mohly číst/zapisovat jen do určitých polí.

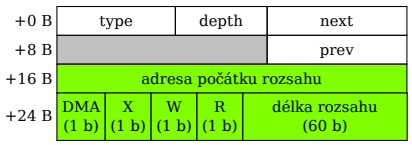
Proto jeho exokernel poskytoval nepřímé operace čtení a zápisu, kde aplikace mohla např. zapsat 4 bajty na určitou adresu pomocí systémového volání a tak exokernel byl schopen ověřit oprávnění pro takto drobnou granularitu. To je sice pomalejší než přímý zápis, ale rychlejší než IPC se službou, a tak někdy opodstatněno.

Moje implementace userspacu je silně vychýlena od konceptu libOS ke konceptu exoserveru a tak takovou funkcionalitu nevyužívá. Nicméně tyto operace jsou ve FleKu i tak implementovány, pro úplnost ideí exokernelu.

3.5.6 Capability slot



Obrázek 3.5 Capability slot untyped memory



Obrázek 3.6 Capability slot userspace memory

3.5.7 Systémová volání, untyped memory

label	FLEK_SYSCALL_UTMEM_GET_FIELDS
argumenty userspacu	žádné
návratové hodnoty	adresa počátku rozsahu ve fyzické paměti
	délka rozsahu v bajtech
	bity oprávnění

Dotazovací systémové volání vracející hodnoty v capability slotu. Jediný bit oprávnění existující u *untyped memory* je oprávnění k DMA. Přitom *untyped memory* nemůže být použito k DMA; tento bit jen omezuje možnost odvození *userspace memory* s DMA bitem, pokud si to nepřejeme. Více v kapitole 3.5.4.

label	FLEK_SYSCALL_UTMEM_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
	bity oprávnění
návratové hodnoty	žádné

Odvodí novou capabilitu typu *untyped memory* reprezentující stejný fyzický rozsah. Argument s bity oprávnění může snížit, ale ne zvýšit, oprávnění.

label	FLEK_SYSCALL_UTMEM_SPLIT
argumenty userspacu	capability index, kam bude vytvořena odštěpená capabilita
	délka levého úseku v bajtech
	bity oprávnění levého úseku
	bity oprávnění pravého úseku
návratové hodnoty	žádné

Rozdělí fyzický rozsah na dvě capability. Levá část bude reprezentována adresovanou capabilitou, pravá část bude zapsána do předaného capability indexu. Části budou mít bity oprávnění nastavené podle argumentů (za předpokladu, že nežádají něco, co originál neměl).

label	FLEK_SYSCALL_UTMEM_MERGE
argumenty userspacu	capability index ukazující na pravého sourozence
návratové hodnoty	žádné

Spojí dvě capability typu *untyped memory* do jedné. Adresovaná capabilita musí být levá část úseku fyzické paměti, capabilita v argumentu pravá část. Capability musí být sourozenci v CDT, tedy ukazatel *next* v capability slotu adresované capability musí ukazovat na capabilitu v argumentu, a obě musí mít stejnou hloubku. Není možné spojovat dva "vnuky", i když jejich rozsahy fyzické paměti na sebe navazují. Také jsou zamítnuty pokusy o spojení capabilit, které mají potomky.

Výsledné bity oprávnění jsou společnou množinou bitů oprávnění obou capabilit.

Po úspěšném skončení se capability slot obývaný pravou capabilitou stane prázdným.

Capabilita *untyped memory* má mnoho dalších systémových volání pro odvozování capabilit, jejichž kernelové objekty pak existují v jejím rozsahu fyzické paměti. Tyto budou popsány v ostatních podkapitolách.

3.5.8 Systémová volání, userspace memory

adresovaná capability	untyped memory
label	FLEK_SYSCALL_UTMEM2USMEM
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
	bity oprávnění
návratové hodnoty	žádné

Jedno z nejdůležitějších systémových volání FleKu. Vytvoří odvozenou capabilitu typu *userspace memory*, kterou pak můžeme používat pro DMA operace a pro mapování do virtuálního adresního prostoru.

Bity oprávnění jsou bity pro *userspace memory*, tedy právo číst, zapisovat, spouštět jako kód a provádět DMA.

Capabilita *userspace memory* má též systémová volání pro dotazování, odvozování, štěpení a spojování. Funguje naprosto stejně jako u *untyped memory*, jen s jinými bity oprávnění. Proto nebudou popisovány.

label	FLEK_SYSCALL_USMEM_SUBSET
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
	offset od začátku rozsahu
	délka rozsahu odvozené capability
	bity oprávnění odvozené capability
	bity oprávnění
návratové hodnoty	žádné

Na rozdíl od *untyped memory*, *userspace memory* umožňuje mít překrývající se rozsahy fyzické paměti napříc odvozenými capabilitami. Smysl *untyped memory* je zajistit disjunktnost rozsahů odvozenin; jakmile už jsme prošli rozhodnutím, že úsek paměti bude mapovatelný do userspace virtuální paměti, tak už se nepotřebujeme omezovat disjunktností.

Toto systémové volání vytváří pod rozsah, přičemž původní capabilita je nedotčena.

label	FLEK_SYSCALL_USMEM_INDIRECT_READ
argumenty userspacu	offset od začátku rozsahu
	počet bajtů k přečtení
návratové hodnoty	hodnota přečtená ze zadané fyzické adresy

Toto systémové volání implementuje nepřímý bezpečný zápis do fyzické paměti (vizte 3.5.5). Může být užitečné, pokud chceme sdílet modifikovatelný stav mezi aplikacemi a granularita stránek je nedostačující. U tohoto volání řeší oprávnění k přístupu ne MMU CPU, ale manuálně exokernel, podle poskytnuté capability. Overhead by měl být menší, než poslání zprávy přes IPC do exoserveru, protože zde nedojde k přepnutí virtuálního adresního prostoru.

Druhý argument musí mít hodnotu 1, 2, 4 nebo 8.

label	FLEK_SYSCALL_USMEM_INDIRECT_WRITE
argumenty userspacu	offset od začátku rozsahu
	počet bajtů k přečtení
	hodnota k zapsání
návratové hodnoty	žádné

Stejná operace jako USMEM_INDIRECT_READ, ale pro zápis.

3.6 Capability table

Capability table je kernelový objekt obsahující pole capability slotů. V seL4 je jeho ekvivalent nazýván *capability node* nebo-li *CNode*. Na rozdíl od seL4, FleK nepoužívá guard bits[17, str. 15], a na rozdíl od Barrelfish, velikost capability table je fixní (1024 slotů).

Překlad capability indexu byl již detailně vysvětlen a ilustrován v kapitole 3.2.6.

Capability table je klíčová komponenta mého capability-based kernelu. Je to právě capability table, která poskytuje kernelu fyzickou paměť, ve které kernel udržuje databázi capabilit.

Pokud je poslední capabilita odkazující na nějakou capability table revokována, jsou zároveň revokovány všechny capability držené ve slotech této capability table. To je nevyhnutelné, protože právě v těchto slotech držíme databázi CDT a tudíž by tyto capability neměly, kde dál existovat.

3.6.1 Rekurzivní mapování

Budování capability spacu aplikace z individuálních capability tables je zodpovědností aplikace. Jak ale může aplikace referencovat capability tables vyšší úrovně, jestliže capability indexy v command registru jsou kernelem automaticky překládány skrze obě úrovně? Je nutné mít možnost adresovat tabulky první úrovně, aby aplikace mohla vložit novou capability table nižší úrovně.

V oblasti virtuální paměti existuje technika jménem *recursive mapping*. Jedná se o situaci, kdy stránkovací tabulka referencuje jako stránkovací tabulku nižší úrovně sama sebe. Výsledkem je, že překlad virtuální adresy, který prochází touto stránkovací tabulkou, zdánlivě přeskočí jednu úroveň překladu ve stromu stránkovacích tabulek. Dokonce lze procházet touto tabulkou opakovaně pro několik úrovní, a tím přeskočit i více než jednu úroveň překladu. Touto technikou lze přistupovat do obsahu stránkovacích tabulek, aniž by všechny tabulky v systému byly namapovány v nejnižší úrovni jako obyčejné paměťové rámce. Nevýhodou je, že musíme obětovat část virtuálního paměťového prostoru pro toto rekurzivní mapování.

Jelikož překlad virtuální adresy ve stromu stránkovacích tabulek a překlad capability indexu ve stromu capability tables je velmi podobný, můžeme se inspirovat touto technikou.

Pokud první slot v capability table vyšší úrovně bude obsahovat odvozenou capabilitu reprezentující tuto tabulku vyšší úrovně (smyčka), potom capability indexy 0 - 1023 budou vlastně překládány FleKem na sloty v tabulce vyšší úrovně.

Mít toto rekurzivní mapování není povinné, ale každá aplikace, která si chce manipulovat vlastním capability prostorem, ho bude potřebovat.

Implementace kernelu si musí dát pozor na tuto smyčku při revokaci. Vzpomeňme, že při revokaci capability table jsou revokovány i všechny sloty v ní. Při smyčce ale jeden ze slotů obsahuje capabilitu typu capability table reprezentující opět tu samou capability table, ve které existuje. Je třeba zde předejít zacyklení.

3.6.2 Systémová volání

adresovaná capabilita	netypovaná paměť
label	FLEK_SYSCALL_UTMEM2CT
argumenty userspacu	capability index první vytvořené capability table
	počet vytvořených capability table
návratové hodnoty	žádné

Vytvoří jeden či více capability table v daném rozsahu fyzické paměti.

Selže, pokud capabilita netypované paměti velikostně nestačí. Capability tables jsou vytvářeny postupně. Pokud už je capability index pro nějakou capability table zabraný,

systémové volání se ukončí s chybou. Dosud vytvořené capability tables ale zůstanou platné a použitelné.

label	FLEK_SYSCALL_CAP_TABLE_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu reprezentující stejnou capability table. V současnosti nemají capability typu capability table žádné *access bity*, kterými bychom mohli například omezit přístup ke slotům na read-only. Do budoucna bych toto chtěl přidat. Mohlo by to být užitečné, pokud bychom chtěli s nedůvěřovanou komponentou sdílet naše capability tables, ale tak aby jejich obsah nemohla narušit.

label	FLEK_SYSCALL_CAP_TABLE_MOVE
argumenty userspacu	capability index vyhodnocen v adresovaném capability table
	capability index cílové capability table
	capability index vyhodnocen v cílové capability table
návratové hodnoty	žádné

Účelem tohoto systémového volání je přesun capability z jednoho slotu do jiného slotu. Je to jeden ze dvou mechanismů přesunu capability u FleKu; tím druhým je movcap.

Z podstaty věci zde musíme referencovat dvě různé capability tables a dva různé capability index – jeden pár pro nalezení přesouvaného capability slotu a druhý pár pro nalezení cílového capability slotu. Po úspěšném přesunu se původní capability slot stane prázdným. Trochu bizarně se může zdát potřeba explicitního argumentu obsahující capability index přesouvané capability – vždyť přece capability index už předáváme v command registru. Překlad přesouvané capability funguje následovně:

1. Capability index v command registru je přeložen v capability space volajícího vlákna. Výsledkem překladu by měl být capability slot obsahující capabilitu typu capability table.

2. V této adresované capability table je přeložen capability index předaný prvním argumentem systémového volání. Výsledkem překladu je capability slot obsahující capabilitu, kterou chceme přesunout.

3. Capability index v druhém argumentu systémového volání je přeložen v capability space volajícího vlákna. Výsledkem překladu by měl být capability slot obsahující capabilitu typu capability table.

4. Capability index ve třetím argumentu systémového volání je přeložen v capability table získané předchozím krokem. Výsledkem překladu je capability slot, kam bude přesouvaná capabilita přesunuta.

Tento trochu těžkopádný algoritmus nám umožní přesouvat capability mezi capability tables, které jsou samy přímo adresovatelné v capability space volajícího vlákna – ale jejich vnitřní sloty už nejsou (protože indexy překládáme ve dvou úrovních tabulek a tohle už by byla třetí úroveň). Jinými slovy, jedná se o jakousi nepřímou adresaci pro ulehčení práce userspacu.

Některé typy capabilit přesouvat nelze. Jde o ty, které využívají slabé reference, například capability typu shadow page table a hardware page table, pokud jsou ve stavu *coupled* a mají odvozené capability (vysvětleno v kapitole 3.10.8). Přesunem by se slabé reference narušily, protože kernel nemůže vědet, v jakých jiných capabilitách se na přesouvanou capabilitu slabou referencí odkazujeme.

label	FLEK_SYSCALL_CAP_TABLE_CYCLE
argumenty userspacu	index (0-1023) uvnitř adresované capability table
návratové hodnoty	žádné

Vytvoří smyčku na zadaném indexu.

Čistě technicky, pokud už vlákno ve svém capability prostoru smyčku má, tak další smyčky může vytvořit pomocí systémového volání FLEK_SYSCALL_CAP_TABLE_DERIVE. Toto volání je pro ulehčení práce userspacu a pro vlákna, které smyčku ještě nemají (ačkoliv jejich tvůrce jim ji asi měl vytvořit).

3.7 CPU slice

CPU slice je capabilita reprezentující nárok na procesorový čas. Není reprezentována žádným kernelovým objektem, všechny informace jsou uloženy přímo v capability slotu.

Skládá se ze dvou hodnot:

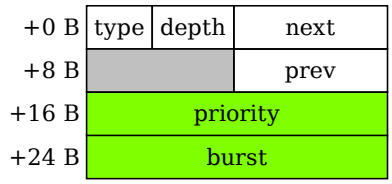
- **Priorita**

  Určuje, v jaké frontě plánovače úloh bude vlákno čekat. Rozsah hodnoty je 1 až 252.

- **Burst**

  Určuje, jak dlouho bude vlákno na procesoru běžet, než plánovač přepne na další vlákno. Dáno v mikrosekundách, maximální hodnota je $2\hat{3}2$.

3.7.1 Capability slot



Obrázek 3.7 CPU slice capability slot

3.7.2 Plánovač

Jelikož FleK je exokernel, bylo snahou nemít v kernelu žádnou policy. Například Unixové kernely mají typicky dynamickou prioritu procesů, kde jsou dlouho čekající procesy dočasně povýšeny na vyšší prioritu s cílem dosáhnout větší férovosti [*The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it.*][25]. Takové funkce chceme mít v userspacu.

Přesto však kernel musí nějak rozhodovat, které vlákno příště poběží. Chceme mít plánovač, nad kterým by si userspace mohl vytvořit složitější policy. Takový, který nejméně omezuje možnosti userspacu.

Zvolil jsem algoritmus round robin s prioritami.

Plánovač má 256 front vláken, jak userspacových tak kernelspacových. Nejvyšší 3 priority a nejnižší 1 priorita jsou vyhrazeny jen kernelovým vláknům (například ovladače hardwaru) a userspacová vlákna takové priority nikdy nemohou mít. Proto je rozsah hodnot priority v CPU slice 1 až 252. Plánovač vždy vybírá pouze vlákna nejvyšší priority. Dokud více prioritnější vlákno chce procesorový čas, tak se méně prioritnější vlákno nedostane ke slovu.

V prioritě 0 běží speciální kernelové vlákno, jenž provádí jen nekonečnou smyčku. Takové vlákno je nezbytné, protože na procesoru vždy musí něco běžet - a přitom v operačním systému zrovna mohou všechna skutečná vlákna na něčem blokovat. Protože má prioritu 0, bude se provádět pouze, pokud žádnou skutečnou úlohu nemáme. Ve smyčce se provádí instrukce *hlt*[22, str. 181], která způsobí zastavení procesorových hodin. Teprve až hardwarové přerušení jejich

činnost obnoví. Cílem je snížít spotřebu elektřiny, když zrovna operační systém nemá, co dělat. Pokročilejší techniky jako nastavování *ACPI P states*[26, str. 475] nejsou implementovány.

Pokud chce userspace implementovat určitou policy, lze jí dosáhnout prostým manipulováním priority a burstu každého spravovaného vlákna v nějakých intervalech. Exokernely Aegis a Xok implementovaly userspace plánovače pomocí IPC, kde po upršení kvóty procesorového času byla zaslána zpráva přiřazenému vláknu, jenž sloužilo jako plánovač.

Jako alternativu s nižším overheadem autoři též ukazuji na kód uploadovatelný userspacem do kernelspace virtual machine, jehož vykonávání by umožňovalo vybírat další úlohu. Touto cestou jsem se chtěl ubrat, jelikož se předejde přebytečnému přepínání kontextu, ale už jsem ji nestihl implementovat.

3.7.3 Systémová volání

label	FLEK_SYSCALL_CPU_SLICE_GET_FIELDS
argumenty userspacu	žádné
návratové hodnoty	priorita
	burst v mikrosekundách

Dotazovací systémové volání, kterým může userspace zjistit hodnoty v capabilitě.

label	FLEK_SYSCALL_CPU_SLICE_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu. Ta bude mít stejnou prioritu a burst jako rodičovská capabilita.

label	FLEK_SYSCALL_CPU_SLICE_SPLIT
argumenty userspacu	capability index, kam bude uložena druhá vzniknuvší capabilita
	priorita první capability
	priorita druhé capability
	burst první capability
návratové hodnoty	žádné

Rozdělí adresovanou capabilitu na dvě.

Tento mechanismus dělení zajišťuje, že odvozené capability jsou disjunktní v burstu. Tedy že nelze z capability s burstem 1 sekunda vytvořit odvozené capability s burstem 0,75 sekundy a 0,5 sekundy (za předpokladu nesnížené priority). Tím je zachován invariant CDT č. 1.

label	FLEK_SYSCALL_CPU_SLICE_MERGE
argumenty userspacu	capability index pravého sourozence
návratové hodnoty	žádné

Spojí dvě capability typu CPU slice do jedné. Capability musí být v CDT sourozenci a to tak, že levý sourozenec je adresovaná capabilita a pravý sourozenec je argumentem systémového volání. Oba také nesmí mít děti, jinak by došlo k narušení invariantu CDT č. 2 3.2). Při nesplnění těchto požadavků je vrácen chybový kód.

Při úspěšném spojení bude levý sourozenec obsahovat výsledek spojení a pravý sourozenec se stane prázdným capability slotem. Výsledná capabilita bude mít prioritu takovou, která je nižší z priorit původních dvou capabilit. A její burst bude součtem burstů původních capabilit. Toto nenarušuje invariant CDT č. 1.

Povšimněme si, že lze původní jednu capabilitu typu CPU slice rozdělit na dvě s nižší prioritou či burstem, pak je spojit zase v jednu – a tato jedna může mít nižší prioritu či burst než ta původní capabilita. To není problémem, protože si můžeme prostě ”zazálohovat”původní capabilitu odvozením, odvozeninu dělit a pak odvozeniny revokovat. Tím nám zůstane původní capabilita s původními hodnotami.

3.8 IPC

Exokernely Aegis a Xok implementovaly mezivláknovou komunikaci velmi nízkoúrovňovým způsobem. Jejich mechanismus připomínal hardwarová přerušení, kdy vlákno mělo specifikovanou adresu handleru a při příchozí komunikaci exokernel pouze přepnul program counter na tuto adresu[1]. Aplikace samotné měly zodpovědnost za zálohování registrů, když k takovému přepnutí došlo.

Autoři hlásili vysoký výkon takového řešení.

Já jsem se od tohoto řešení vzdálil z několika důvodů:

- Stav aplikace v moment komunikace

  Nedává moc smysl, aby aplikace povinně přijímala komunikaci, i když zrovna není ve stavu, kdy je schopna ji rovnou zpracovat (je uprostřed dělání něčeho jiného). Samotná komunikace skokem je jistě rychlá, ale overhead nějaké synchronizace (event loop?) na straně přijímající aplikace nemusí být hned zřejmý. Typicky budeme chtít přijímat zprávy až v určitý vhodný moment, kdy pak budeme blokovat na přicházející zprávu.

- Kernel může zálohovat registry efektivněji

  Architektura AMD64 má sice velkou sadu registrů, ale také v dnešní době má velmi optimalizované instrukce pro jejich ukládání a načítání do/z paměti. Některé tyto instrukce jsou privilegované a userspace je nemůže použít, např. *XSAVES*: [*XSAVES can be executed only if CPL = 0*[3] *(...) if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the modified optimization). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.)*][27]. Tím bychom se stříleli do nohy při použití řešení Aegis/Xok.

Ve výsledku jsem pro IPC převzal, téměř beze změny, řešení rodiny mikrokernelů L4. Ačkoliv výpočetní overhead zde bude jistě větší, jedná se stále o velmi jednoduché řešení.

IPC je zpřístupněno capabilitou typu endpoint, která reprezentuje kernelový objekt. V tomto kernelovém objektu jsou jednoduše dvě fronty (dvojité spojové seznamy), jedna pro přijímající vlákna a jedna pro odesílající.

Při synchronní komunikaci se zkontroluje, zda už čeká nějaké vlákno na opačné frontě (tj. na přijímající pokud posíláme); pokud ano, překopírují se hodnoty registrů (počet registrů ke kopírování je indikovaný v syscallu) ze současného vlákna do tohoto vlákna (nebo naopak) a vlákno je probuzeno. Pokud ne, znamená to, že opačná fronta je prázdná, a my se přidáme do příslušné fronty a uspíme se.

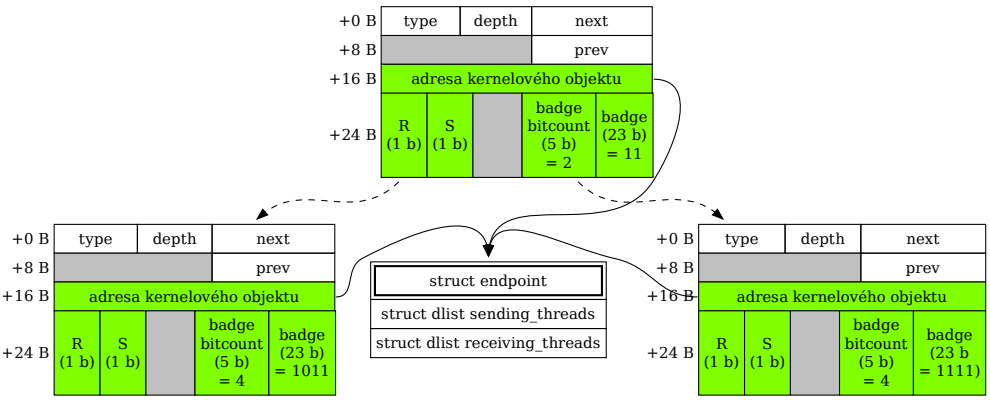
Při asynchronní komunikaci se děje to samé, kromě přidání se do fronty v případě, že opačná fronta je prázdná. Místo toho systémové volání rovnou vrátí chybový kód.

Nikdy se nemůže stát, že by na obou frontách čekala vlákna zároveň.

Toto řešení je velmi komfortní na použití a přitom dostatečně jednoduché, aby mohlo být rychlé. Nicméně stále se jedná o odklon od původních exokernelů. Myslím, že spíše než se mermomocí snažit vše implementovat na co nejnižší úrovni, je důležitejší držet se centrální myšlenky exokernelového API kopírujícího rozhraní zařízení, které je přístupné běžným aplikacím.

[3] Také známý jako *ring 0*, čili kernelspace.

3.8.1 Capability slot



Obrázek 3.8 Capability slot a kernelový objekt endpointu

Šafrované hrany reprezentují vztah mezi capabilitami – šipečka vede směrem od rodičovské capability k odvozené capabilitě.

Od seL4 jsem také převzal mechanismus tzv. *badges*. Ty fungují jako identifikace odesílatele z pohledu přijímatele. Když vytváříme odvozenou capabilitu již existující capability typu *endpoint*, tak můžeme do odvozené capability přidat dalších pár bitů *badge*. Můžeme bity pouze přidávat, ty z rodičovské capability jsou vždy překopírovány. Toto lze pozorovat v ilustraci výše.

Pokaždé, když je *endpointem* poslána zpráva, je do prvního argumentového registru přijímajícího vlákna zakódován badge té capability, skrze kterou byla tato zpráva poslána.

Kombinace těchto dvou vlastností umožňuje aplikaci vytvořit si nový *endpoint* s určitým badge a vytvořit od něj 5 odvozených capabilit typu *endpoint*, kde každá si přidá své unikátní bity badge. Následně těchto 5 odvozených *endpointů* může rozdat nějakým 5 cizím vláknům a začít poslouchat na své původní capabilitě typu *endpoint*. Protože bity badge lze pouze přidávat, je zcela jedno, že tato vlákna si mohou vytvářet další odvozeniny darovaného *endpointu*. Naše aplikace i tak bude bude vědět, od koho zpráva přichází.

Posledním prvkem capability slotu typu *endpoint* jsou dva bity oprávnění, jeden pro posílání zpráv a jeden pro přijímání. Je tedy možné například aplikaci svěřit *endpoint* tak, že skrze něj bude moci pouze naslouchat.

3.8.2 Způsoby komunikace

Existuje celkem 6 komunikačních operací nad *endpointem*: synchronní přijímání, asynchronní přijímání, synchronní posílání, asynchronní posílání, synchronní call, asynchronní call. Operace mají stejný formát procesorových registrů jako systémová volání, pouze políčko *operace* má jinou hodnotu než při systémovém volání (vizte 3.3) a to právě nastavenou na konstantu příslušející jedné z těchto šesti možností. Adresovanou capabilitou je nějaký endpoint.

Přijímající operace ještě využívají první argumentový registr pro metadata o přijaté zprávě, jako např. badge odesílatele. Přijaté argumenty zprávy jsou pak v procesorových registrech a MRS posunuty o jednu pozici. FleK a libsyscall toto vše řeší automaticky.

Problémem na závěr je, jak implementovat synchronní komunikaci a nenechat se tak vystavit *denial of service* útokům. Aplikace může poslat požadavek exoserveru, ten blokuje dokud

nepřijme nějaký takový požadavek. Požadavek obslouží a pak potřebuje poslat odpověď aplikaci. Pokud aplikace odmítne blokujícím způsobem čekat na přijetí zprávy, pak má exoserver dvě možnosti: poslat odpověď asynchronně a okamžitě selhat nebo poslat odpověď synchronně a tam se navždy zaseknout, blokujíc bez přijímatele. Druhá možnost je očividně nepřijatelná, ale první možnost též není reálná. I nevinná aplikace se může do takové situace dostat v důsledku nepříznivého přepínání úloh a třeba se ještě nedostala do bodu synchronního přijímání (tj. race condition).

Předchozí řešení tohoto problému v rodině kernelu L4 používala synchronní posílání a přijímaní s konfigurovatelnými timeouty. Toto se neosvědčilo[28] a bylo nahrazeno cally. Jednou z kritik byla obtížnost nalezení vhodné délky timeoutu.

Call je atomická kombinace (a)synchronního poslání a synchronního přijetí. Pokud ji aplikace použije, pak nemůže dojít k výše zmíněnému race conditionu mezi posláním a přijetím. Exoserver posílá odpověď asynchronním posíláním: nevinná aplikace používající call nebude mít problém a škodlivá aplikace, která nečeká na odpověď, exoserver nezablokuje.

3.8.3 Systémová volání

label	FLEK_SYSCALL_ENDPOINT_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
	badge k připojení k badge od rodičovské capability
	počet bitů připojovaného badge
	access bity
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu typu *endpoint* ukazující na ten samý kernelový objekt.

Druhý parametr je nutný, protože v prvním parametru můžeme chtít předávat jen nulové bity, a tak by nebylo možné dopočítat délku badge k přidání. Třetí parametr nastavuje bity oprávnění; pochopitelně musí být nižší nebo stejné jako v rodičovské capabilitě.

label	FLEK_SYSCALL_ENDPOINT_GET_FIELDS
argumenty userspacu	žádné
návratové hodnoty	badge
	počet bitů badge
	access bity

Vrátí výše popsané a ilustrované hodnoty z capability slotu typu *endpoint*.

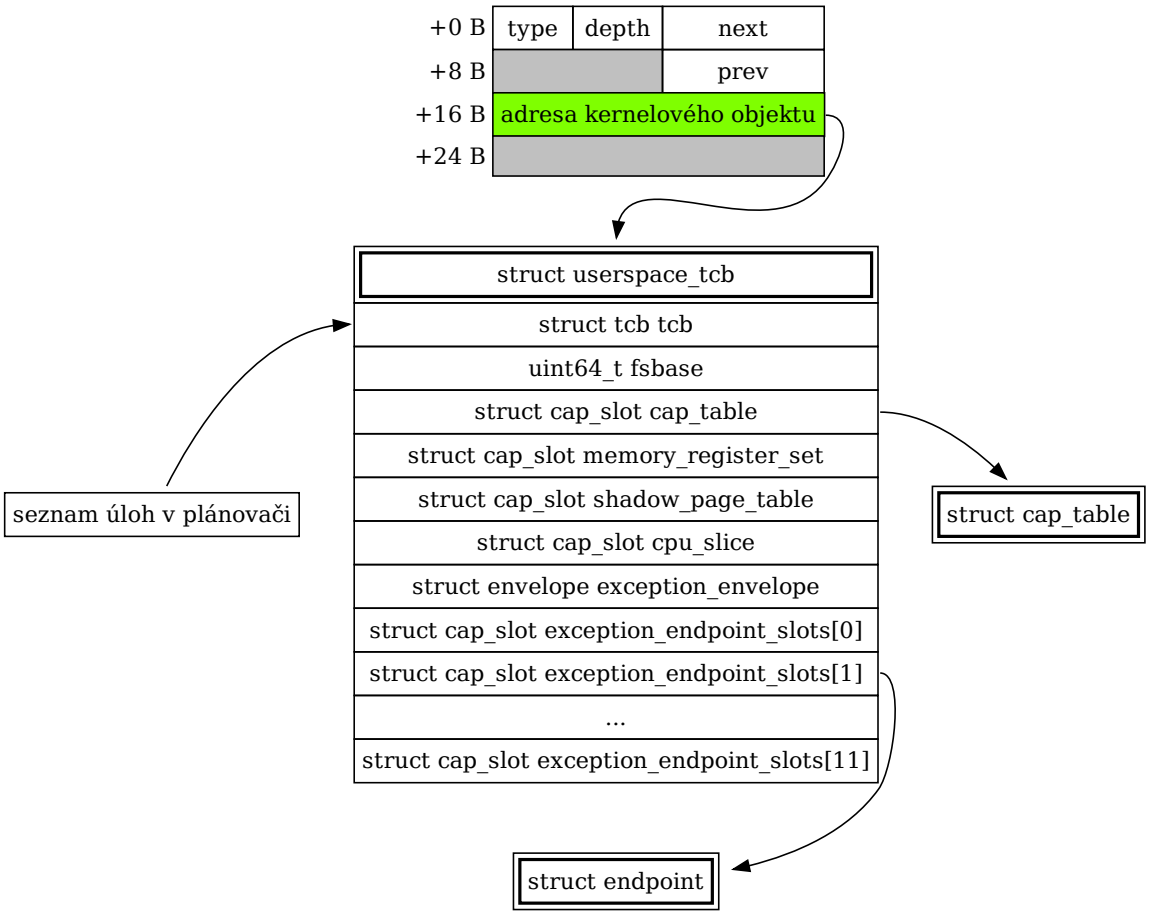
3.9 Vlákna

Userspacové vlákno je reprezentováno capabilitou typu TCB (*thread control block*), jenž reprezentuje kernelový objekt (struct userspace_tcb). Kernelový objekt je nutný pro ukládání hodnot registrů vlákna při přepínání vláken plánovačem úloh. Dále musí držet reference na některá nezbytná nastavení, jako například priorita procesu nebo kořenová stránkovací tabulka.

Objekt TCB je relativně velký, přibližně 1400 bajtů.

Revokace TCB, které patří právě běžícímu vláknu, selže s chybovým kódem. Přinejmenším současná implementace kernelu neumožňuje vláknu smazat samo sebe.

3.9.1 Capability slot



Obrázek 3.9 TCB capability slot

Políčka struct userspace_tcb byla zjednodušena pro ilustraci. Člen *struct tcb* obsahuje prostor pro zálohování registrů a různé pomocné informace pro plánovač úloh. Tyto nejsou relevantní z pohledu návrhu API pro userspace.

3.9.2 Návrhový vzor connector

Během návrhu mého kernelu jsem se setkával s opakujícím se řešením problému propojování capabilit a správné reakce capability na revokaci jiné propojené capability.

Představme si situaci, kde máme capabilitu typu CPU slice a capabilitu typu TCB. Vláknu přiřadíme daný slice a vlákno spustíme. Někdy později však bude capabilita typu CPU slice revokována. Pokud bude vlákno běžet dál, jedná se o narušení bezpečnosti systému, protože používáme zdroj (procesorový čas), ke kterému již nemáme práva.

Kernel musí být schopen toto zjistit při revokaci a výpočetně efektivně, jinak by se revokace obecně staly pomalé.

Řešením je vložení capability slotu dovnitř kernelového objektu, ke kterému nějakou capabilitu přiřazujeme. Tento capability slot je plnohodnotý, chová se stejně jako sloty v capability table. Na rozdíl od slotů v capability tables ale není přímo adresovatelný pomocí capability indexů (tj. není součástí capability prostoru). V moment přiřazení capability se do tohoto vnitřního slotu vytvoří odvozená capabilita přiřazované capability. Důsledkem je, že v případě revokace přiřazované capability se revokuje i odvozená capabilita v tomto vnitřním slotu. Je to naprosto obyčejný průběh algoritmu revokace, který revokuje nejdříve všechny potomky revokované capability. A tím je capabilita "odpřiřazena"a nezbývá žádná dangling reference na revokovaný zdroj nebo něco podobného.

Tím máme vyřešeno "odpřiřazení"při revokaci, ale co když při takové události musíme ještě provést nějaký obsluhující kód? Vraťme se k situaci v druhém odstavci této podkapitoly. Obsluhujícím kódem by zde byl myšlen kód uspávající dané vlákno při odpřiřazení. Jak může ale algoritmus revokace vědět, že když revokuje capabilitu typu CPU slice, tak že tento capability slot je zrovna uvnitř TCB objektu a že tedy ještě musí uspat toto vlákno? Řešením tohoto problému je použití podtypů capability. Můžeme si představit capabilitu typu "CPU slice uvnitř TCB", která se chová úplně stejně jako capabilita typu "CPU slice"a která může být v CDT potomkem pouze capability typu "CPU slice". To umožní algoritmu revokace rozpoznat, že likviduje CPU slice, který zrovna existuje v capability slotu uvnitř TCB objektu. A protože bajtový offset tohoto slotu uvnitř TCB objektu je známý při kompilaci, je možné vypočítat adresu obalujícího TCB objektu. A to je vše, co potřebujeme, aby kernel mohl automaticky uspat vlákno při revokaci jeho CPU slice.

Tato technika je používána kernelem seL4 u právě takovéto situace (TCB a procesorový čas). Protože FleK je exokernel a musí v capability systému reprezentovat mnohem více typů capabilit, které jsou typicky nějak vzájemně propojované, je tohoto návrhového vzoru v mém kernelu velmi široce užito. Nikde jsem nenašel žádný popis či název tohoto vzoru (nejspíše protože v seL4 se používá jen párkrát), ale protože se na něj musím ve zbytku práce opakovaně odkazovat, dal jsem mu tento název – *connector*.

3.9.3 Procesorové výjimky

Architektura AMD64 používá tabulku jménem *interrupt descriptor table* pro nalezení interrupt handlerů. Tato tabulka má 256 prvků a operační systém může na jednotlivá čísla přerušení napojovat hardware, jako například APIC či PCI MSI. Prvních 32 prvků tabulky je rezervováno procesorovými výjimkami. Některé z těchto procesorových výjimek mají význam pouze pro kernel, některé ale chceme propagovat do userspacu. Userspace v operačním systému na bázi exokernelu musí být schopen řešit pagefaulty, floating point exceptions a podobně.

Exokernel FleK poskytuje API pro informování userspacu o 12 různých typech procesorových výjimek. Toto je implementováno 12 instancemi capability slotu uvnitř kernelového objektu TCB, do kterých se odvozují capability typu endpoint. To znamená, že různé výjimky jednoho vlákna mohou způsobovat poslání zprávy do různých endpointů.

Protože procesorové výjimky mohou být parametrizované (procesor předává určité informace ve speciálních registrech), je třeba tyto také poslat do příslušného endpointu. K tomu slouží

jedna instance envelope uvnitř kernelového objektu TCB, která udržuje tyto parametry, dokud si zprávy z endpointu nějaké vlákno nevyzvedne.

Proč stačí pouze 1 envelope a není jich také 12? Protože když vlákno vyvolá procesorou výjimku, FleK toto vlákno uspí. Vlákno musí být userspacem probuzeno a to bude dělat ta userspacová komponenta, která čeká na zprávu o procesorové výjimce, aby ji mohla obsloužit. Protože do momentu obsloužení je vlákno uspáno, není možné, aby byla způsobena další procesorová výjimka na tomto vlákně dřív, než je zpráva o předchozí přijata. Proto stačí jeden envelope.

Pokud přecejen něco probudí toto uspané vlákno, aniž by to nejdříve přijalo zprávu o procesorové výjimce, bude tato zpráva nadále čekat v endpointu. Pokud pak nastane další procesorová výjimka, další zpráva už nebude vygenerována (protože envelope stále drží první zprávu). Ale toto se může stát jen v případě vadně naprogramovaného userspacu.

Pokud dojde k procesorové výjimce a příslušný capability slot je prázdný, pak FleK vlákno uspí, ale žádná zpráva userspacu poslána není. Proto by v praxi všechny typy výjimek měly být userspacem pozorovány, ačkoliv FleK si toto nevynucuje.

Seznam výjimek je v tabulce 3.1.

3.9.4 Systémová volání

label	FLEK_SYSCALL_TCB_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu reprezentující stejný kernelový objekt TCB.

label	FLEK_SYSCALL_TCB_ASSIGN_MRS
argumenty userspacu	capability index odkazující na capabilitu typu *userspace memory*
	offset uvnitř této paměti
návratové hodnoty	žádné

Použije poskytnutou paměť jako MRS oblast pro toto vlákno. Účel MRS je popsán v kapitole 3.3.

Offset umožňuje userspacu v jednom velkém rozsahu určit konkrétní pozici MRS. Tím se předejde potřebě userspacu pro každé vlákno vytvořit novou capabilitu typu *userspace memory* reprezentující jen konkrétní úsek; takto může pro několik vláken použít jen jednu capabilitu typu *userspace memory* s různými offsety. To snižuje paměťový overhead.

Capabilita typu *userspace memory* a offset v ní musí popisovat oblast o velikosti alespoň paměťového rámce (4096 bajtů). Tato oblast musí být navíc zarovnaná na velikost paměťového rámce. Smyslem tohoto požadavku je fakt, že skrze MRS chceme někdy předávat i datové struktury a ty mohou mít libovolné potřeby zarovnání. Zarovnání na velikost (tzv. přiřozené zarovnání) je v této situaci to nejvyšší technicky možné a tudíž splňující všechny potřeby userspacu, které mohou nastat.

Nakonec, protože MRS je použita mimojiné pro přijímání a posílání zpráv s velkým počtem argumentů, musí mít poskytnutá capabilita typu *userspace memory* mít oprávnění pro čtení i zápis. Bez tohoto požadavku by mohlo najít k narušení bezpečnosti. Vlákno by mohlo použít paměť, ke které má jen právo pro čteni, jako MRS a poslat si velkou zprávu, čímž by mohlo přepsat obsah paměti, aniž by potřebovalo právo na zápis.

Nastaví danou *shadow page table* jako kořen stromu stránkovacích tabulek. Jedná se tedy o přiřazení virtuálního adresního prostoru.

label	FLEK_SYSCALL_TCB_ASSIGN_SPT
argumenty userspacu	capability index odkazující na capabilitu typu *shadow page table*
návratové hodnoty	žádné

Shadow page table musí být úrovně PML4 (nejvyšší úroveň stránkovací tabulky na AMD64) a musí být ve stavu *coupled*. Stav *coupled* je vyžadován, protože jen tehdy má kernel přístup k adrese asociované hardwarové stránkovací tabulky – a ta je to, co kernel musí v procesoru nastavit. Pokud tyto podmínky nejsou splněny, je vrácen chybový kód.

Jestliže se již přiřazená *shadow page table* přepne ze stavu *coupled* na *decoupled*, bude její přiřazení zrušeno a vlákno uspáno (pokud běželo). To je důsledek faktu, že toto systémové volání vytváří odvozenou capabilitu poskytnuté *shadow page table* do vnitřního capability slotu uvnitř TCB objektu a změna stavu SPT do *decoupled* způsobuje automatickou revokaci všech odvozených capabilit té SPT. A v důsledku návrhového vzoru connector dojde k uspání vlákna. To samé se stane, pokud je revokována přímo *shadow page table* nebo její asociovaná hardwarová stránkovací tabulka. Takto nám to všechno hezky do sebe zapadá a vlákno nikdy nemůže běžet bez virtuálního adresního prostoru, nebo s virtuálním adresním prostorem, ke kterému ztratilo práva.

Více vláken může používat stejný virtuální adresní prostor.

label	FLEK_SYSCALL_TCB_ASSIGN_CPU_SLICE
argumenty userspacu	capability index odkazující na capabilitu typu *CPU slice*
návratové hodnoty	žádné

Přiřadí kvótu procesorového času.

label	FLEK_SYSCALL_TCB_ASSIGN_CT
argumenty userspacu	capability index odkazující na capabilitu typu *capability table*
návratové hodnoty	žádné

Přiřadí capability table, která bude sloužit jako capability table vyšší úrovně při vyhodnocování capability indexů při systémových voláních tohoto vlákna. Určuje tedy capability prostor vlákna.

Více vláken může používat stejný capability prostor.

label	FLEK_SYSCALL_TCB_ACTIVATE
argumenty userspacu	žádné
návratové hodnoty	žádné

Probudí vlákno. Vlákno může být probuzeno pouze, pokud splňuje následující požadavky:

- Má capability prostor.

  To znamená, že vláknu byla přiřazena capability table. Toto je povinné, protože systémová volání FleKu vždy adresují nějakou capabilitu. A to pomocí capability indexu, který musí být vyhodnocen ve stromu capability tables (= capability prostor). Bez capability prostoru by vlákno sice teoreticky mohlo běžet, ale nemohlo by nikdy provést žádné systémové volání. A takový program je k ničemu.

- Má shadow page table úrovně PML4.

  Běžící program vždy sahá do virtuálního adresního prostoru. I kdyby nic nečetl a nezapisoval, instruction fetching stále probíhá ve virtuální paměti.

  To znamená, že každý program musí mít nějaký strom stránkovacích tabulek.

- Má CPU slice.

  Běžící program spotřebovává procesorový čas. Aby vlákno mohlo běžet, musí mu být alokován nějaký procesorový čas. To je provedeno přiřazením capability typu CPU slice.

Všimněte si, že MRS není povinný.

Pokud nějaká podmínka není splněna, bude vrácen chybový kód popisující, co přesně není v pořádku.

Pokud nějaká z podmínek přestane platit, zatímco vlákno již běží, bude vlákno automaticky uspáno. Po nápravě můžeme tímto systémovým voláním vlákno zase probudit.

Probuzení již běžícího vlákna nic nedělá.

label	FLEK_SYSCALL_TCB_DISABLE
argumenty userspacu	žádné
návratové hodnoty	žádné

Uspí vlákno. Vlákno bude znovu vykonáváno až po uspěšném provedení systémového volání FLEK_SYSCALL_TCB_ACTIVATE.

Vlákno může být tímto voláním uspáno, i pokud zrovna blokuje (např. na endpointu nebo na conditional locku). V tom případě se okamžitě nic nezmění, ale jakmile blokování pomine, tak vlákno bude nadále uspané.

label	FLEK_SYSCALL_TCB_ASSIGN_EXCEPTION_ENDPOINT
argumenty userspacu	capability index endpointu
	číslo procesorové výjimky
návratové hodnoty	žádné

Přiřadí endpoint k dané procesorové výjimce adresovaného vlákna. Více v podkapitole 3.9.3.

Capabilita typu endpoint musí mít oprávnění na posílání zpráv. Oprávnění na přijímání zpráv není potřeba, protože kernel v při procesorové výjimce zprávu posílá, nic z endpointu nepřijímá.

Pokud už byl nějaký endpoint k této procesorové výjimce přiřazen, je původní přiřazený endpoint odpojen. Pouze jeden endpoint může být připojen na jednu procesorovou výjimku vlákna.

Procesorové výjimky, na které může userspace pod FleKem naslouchat, jsou následující:

Tabulka 3.1 Seznam procesorových výjimek reportovaných userspacu

#	název výjimky	procesorové registry poslané jako argumenty zprávy
0	Divide by zero	RIP
1	Debug	RIP[4]
2	Breakpoint	RIP
3	Overflow	RIP
4	Bound range exceeded	RIP
5	Invalid opcode	RIP
6	Stack segment fault	RIP
7	General protection fault	RIP
8	Page fault	RIP, CR2 (virtuální adresa přístupu), error code (příznaky typu přístupu)
9	x87 floating point exception	RIP, x87 RIP, FSW
10	alignment check	RIP
11	SIMD floating point exception	RIP, MXCSR

[4]Debug registry v součanosti posílané nejsou, i když by měly být.

label	FLEK_SYSCALL_TCB_GET_GP_REGISTERS
argumenty userspacu	žádné
návratové hodnoty	hodnoty 17 procesorových registrů adresovaného vlákna

Vrátí hodnoty "běžných"registrů vlákna. Protože se jedná o 17 64-bitových hodnot, nemůže je kernel vrátit ze systémového volání v procesorových registrech. Je nutné použít MRS. Pokud vlákno, provádějící toto systémové volání, nemá přiřazený MRS, volání selže.

label	FLEK_SYSCALL_TCB_SET_GP_REGISTERS
argumenty userspacu	hodnoty 17 procesorových registrů adresovaného vlákna
návratové hodnoty	žádné

Nastaví běžné procesorové registry adresovaného vlákna na poskytnuté hodnoty v MRS. Stejně jako u FLEK_SYSCALL_TCB_GET_GP_REGISTERS, pokud vlákno, provádějící toto systémové volání, nemá přiřazený MRS, volání selže.

Kombinace těchto dvou systémových volání umožňuje upravovat hodnoty registrů jiného vlákna. Tuto funkcionalitu budou využívat hlavně exoservery, například pro nastavení počátečního stavu vlákna.

label	FLEK_SYSCALL_TCB_SET_FS
argumenty userspacu	hodnota registru FSBASE
návratové hodnoty	žádné

Registr FSBASE určuje offset segmentu FS. Instrukce přistupující k paměti mohou specifikovat, že přístup má proběhnout v segmentu FS, což způsobí, že procesor příčte hodnotu registru FSBASE k přistupované adrese. Tohoto je typicky používáno pro implementaci *thread local storage*[29, str. 14].

label	FLEK_SYSCALL_TCB_GET_FS
argumenty userspacu	žádné
návratové hodnoty	hodnota registru FSBASE

Získá současnou hodnotu registru FSBASE.

3.10 Správa virtuální paměti

Virtuální pamětí rozumíme paměťový adresní prostor, v němž je každý přístup do paměti překládán do fyzického adresního prostoru. Na AMD64 je podpora segmentace paměti v *long mode* téměř odstraněna a budeme se tedy zabývat pouze stránkováním.

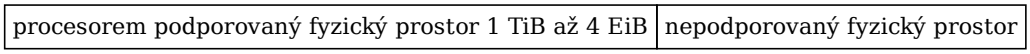
3.10.1 Chování hardware

Architektura AMD64 v *long mode* pří stránkování překládá virtuální adresy pomocí stromu o hloubce čtyř stránkovacích tabulek. Stránkování je v *long mode* povinné; při jeho vypnutí se procesor navrátí do *protected mode* (32bitový režim). Fyzické adresy mají délku 40 až 52 bitů[22, str. 154]; konkrétní hodnota je v runtime k dispozici instrukcí *cpuid*. Vyšší bity musí být nulové, jinak při při použití stránkovací tabulky obsahující vyšší než podporovanou fyzickou adresu dojde k procesorové výjimce.

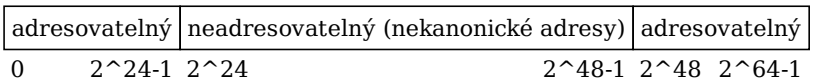
Virtuální adresy mají délku 48 bitů – každá úroveň stránkovacích tabulek přeloží 9 bitů a zbývajích dolních 12 bitů je offset uvnitř 4 KiB paměťového rámce. To platí při použití běžných 4KiB stránek, AMD64 dále podporuje 2MiB stránky. V tom případě má strom pouze tři úrovně stránkovacích tabulek. Tabulka třetí úrovně neukazuje na tabulku čtvrté úrovně, ale přímo na destinaci překladu (jehož adresa musí být zarovnána na 2 MiB).

Některé procesory podporují i 1GiB stránky. Ty fungují obdobně, ale pouze s dvěmi úrovněmi stránkovacích tabulek. FleK používá 1GiB stránky pro mapování celého fyzického adresního prostoru do kernelspace. Pokud procesor toto rozšíření nepodporuje, FleK použije 2MiB stránky, což má vyšší paměťový overhead.

Protože registry AMD64 mají délku 64 bitů, tedy delší než virtuální adresa, a protože velikost virtuálního adresního prostoru se může do budoucnosti rozšiřovat, rozhodli se inženýři AMD reprezentovat virtuální adresy v *sign-extended* formátu. Tedy dolních 48 bitů drží informaci a horních 16 bitů má stejnou hodnotu jako nejvyšší bit držící informaci. Tomuto formátu se říká *canonical address*. Důsledek je ten, že uprostřed virtuálního adresního prostoru je jakási "díra", jenže nemůže být reprezentována kanonickou (tj. platnou) virtuální adresou. Pokus o přístup do takové adresy způsobí procesorovou výjimku.



Obrázek 3.10 Fyzický adresní prostor



Obrázek 3.11 Virtuální adresní prostor

Některé novější procesory podporují i strom o hloubce pěti tabulek*amd64pml5*, což rozšiřuje velikost virtuální adresy na 57 bitů. Touto volitelnou funkcionalitou se nebudeme zabývat. -pcid

3.10.2 Požadavky

- **Invariant č. 1**: Do virtuálního prostoru nelze mapovat fyzickou paměť, ke které nemáme capabilitu

  Narušením tohoto invariantu bychom ztratili bezpečnost systému. Navíc nestačí jen mít capabilitu k mapovanému rozsahu fyzické paměti, ale též musí tato capabilita mít příslušná oprávnění. Například pokud máme capability fyzické paměti jen s oprávnění ke čtení, nelze ji namapovat do virtuální paměti s oprávněním k zápisu.

- **Invariant č. 2**: Revokace capability fyzické paměti musí vyvolat odmapování z virtuální paměti

  Dalším závažným narušením bezpečnosti systému by bylo, pokud bychom revokovali capabilitu fyzické paměti, ale ta by zůstala ve stránkovacích tabulkách stále namapovaná. Po revokaci capability fyzické paměti bychom totiž z její rodičovské capability typu netypovaná paměť mohli odvodit capability kernelových objektů. V ten moment by userspace měl možnost číst nebo i dokonce zapisovat do paměti užívané kernelem. To je naprosto nepřípustné.

- **Invariant č. 3**: Revokace capability stránkovací tabulky musí vyvolat její odmapování ze stránkovací tabulky vyšší úrovně

  V podstatě invariant č. 2, ale pro stránkovací tabulky samotné.

- **Invariant č. 4**: Nelze přímo zapisovat do stránkovací tabulky

  Exokernel nesmí umožnit userspacu namapovat stránkovací tabulku jako přímo adresovatelnou paměť s oprávněním pro zápis. Jinými slovy, userspace vlákno nemůže zápisem do virtuální paměti něco přepsat ve stránkovací tabulce. Pokud by toto někdy bylo možné, userspace by mohl obcházet veškeré bezpečnostní kontroly exokernelu a porušovat tyto invarianty.

- **Invariant č. 5**: Paměť zapisovatelná userspacem nesmí být nikdy interpretována jako stránkovací tabulka

  Stejný problém jako invariant č. 4, ale v opačném pořadí. Exokernel musí zabránit userspacu v propojování stránkovacích tabulek takovým způsobem, že by procesor interpretoval "obyčejnou"paměť jako stránkovací tabulku. V opačném případě by userspace mohl získat schopnost zapisovat do kernelspace paměti.

- **Invariant č. 6**: Translation lookaside buffer (TLB) cache musí být vždy synchronizována s obsahem stránkovacích tabulek

  Tento požadavek je nejobtížnější na implementaci. Userspace si může mapovat a odmapovávat stránky a stránkovací tabulky všelijak (i sdíleně v několika adresních prostorech), ale exokernel musí vždy nějak vědět, jaká část jakého virtuálního adresního prostoru je tím dotčena, aby mohl flushnout relevantní část TLB cache. Nejen to, je klíčové toto implementovat efektivně – neflushovat cache více, než je třeba (ideálně vůbec, pokud možno). A zároveň mít minimální paměťový overhead při udržování k tomuto potřebných informací, které navíc musejí následovat princip deterministicky alokované paměti, jako zbytek capability systému. Což, jak si předvedeme dále, není vůbec jednoduché.

  Nesplnění tohoto invariantu by narušilo bezpečnost systému, protože by došlo k narušení dodatečného požadavku č. 1 v sekci 3.2, ovšem bez ohlášení neúspěšného provedení. Aplikace by například mohla nadále používat paměť, která jí byla úspěšně odebrána, protože by mapování zůstalo v TLB. Dále by takto bylo možno narušit výše popsané invarianty č. 4 a č. 5 podobným způsobem.

  Dodatečné požadavky:

- **Dodatečný požadavek č. 1**: Stránkovací tabulky lze mapovat do virtuálního prostoru pro čtení

  Tento požadavek není v rozporu s invariantem č. 4. Je to stejná situace, ale tentokrát je stránkovací tabulka namapovaná s oprávněním pouze pro čtení. Tato vlastnost je praktická, protože procesor ve stránkovacích tabulkách nastavuje bity (*accessed* a *dirty*), které jsou pro operační systém velmi užitečné. Userspace by se na hodnoty těchto bitů mohl zeptat exokernelu systémovým voláním, ale přímé čtení je jistě rychlejší. Proto je tento požadavek pouze doplňující.

  Engler ve své práci[1] zmiňuje: [*As dictated by the exokernel principle of exposing kernel book-keeping structures, the page table should be visible (read only) at application level.*].

- **Dodatečný požadavek č. 2**: Schopnost kombinovat stránky různých velikostí

  AMD64 podporuje 4KiB a 2MiB stránky. Některé procesory i 1GiB stránky. Exokernel by měl umožňovat vše, co podporuje hardware. Tudíž i kombinování různých velikostí stránek v rámci jednoho virtuálního adresního prostoru by mělo být přístupné userspacu.

Posledním cílem je subsystém správy virtuální paměti s co nejmenším paměťovým overheadem. Velikost základního typu stránky na AMD64 je 4 KiB a jeden paměťový rámec může být mapován několikrát do různých virtuálních adresních prostorů. I desítky bajtů spotřebované capability systémem na držení informace o mapování jedné stránky mohou na moderním PC s několika GiB paměti vést k promrhání stovek MiB jen na book-keepingu virtuální paměti.

Při návrhu virtuální paměti jsem nejdříve analyzoval řešení jiných capability-based kernelů.

3.10.3 Aegis

Aegis byl implementován pro procesor, který obsluhoval pagefaulty softwarově. Aegis proto svým secure bindings systémem chránil obsah TLB: [*A simple hardware secure binding is a TLB entry: when a TLB fault occurs the complex map-ping of virtual to physical addresses in a library operating system's page table is performed and then loaded into the kernel (bind time) and then used multiple times (access time)*][1].

Architektura AMD64 obsluhuje pagefaulty na hardwarové úrovni. Proto je pro mě jejich řešení nepoužitelné a capability systém mého exokernel musí chránit obsah hardwarových stránkovacích tabulek.

Engler toto ve své práci zmiňuje: [*If the underlying hardware defines a page-table interface, then an exokernel must guard the page table instead of the TLB*][1].

3.10.4 seL4

Mapování fyzické paměti do virtuálního adresního prostoru v mikrokernelu seL4 funguje následujícím způsobem. Nejdříve je potřeba odvodit capabilitu typu *page* z capability typu *untyped memory*. A to pro každý jednotlivý paměťový rámec, který chceme mapovat. Následně lze odvodit capabilitu reprezentující hardwarovou stránkovací tabulku z capability typu *untyped memory*.

Nad capabilitou reprezentující hardwarovou stránkovací tabulku lze provést systémové volání žádající namapovaní capability typu *page*. Dojde k zápisu do hardwarové stránkovací tabulky a do capability slotu capability *page* je zapsán ASID klíč a virtuální adresa. První identifikuje virtuální adresní prostor a druhé pozici v něm, kde je tato *page* namapována.

Pokud je potom capabilita *page* revokována, kernel na základě těchto dvou informaci obsažených v jeho slotu může provést odmapování z hardwarové virtuální tabulky a invalidovat správnou adresu v TLB.

ASID klíč identifikuje adresní prostor, a tím umožní kernelu najít hardwarovou stránkovací tabulku nejvyšší úrovně. Kernel poté prochází strom stránkovacích tabulek podle poznačené

virtuální adresy, až nalezne stránkovací tabulku, kde byla revokovaná *page* namapována. Položku v tabulce vynuluje. Protože kernel zná poznačenou virtuální adresu, může smazat konkrétní mapování v TLB (např. na AMD64 instrukcí invlpg[22, str. 182]).

Představme si situaci, kdy není revokována *page* ale místo ní ta capabilita reprezentující hardwarovou stránkovací tabulku. V tom případě se děje to samé (i tato capabilita v sobě drží ASID klíč a virtuální adresu). Ale zbyde nám tu dangling reference uvnitř capability slotu *page*, protože tato capabilita o ničem nebude vědet a bude dále obsahovat ASID klíč a virtuální adresu, i když už byla reálně odmapována z virtuálního adresního prostoru. To však nic nerozbije, při revokaci *page* podle předchozího odstavce kernel předčasně ukončí procházení stromu stránkovacích tabulek (protože ta revokovaná tabulka už v něm chybí) a nemusí nic dělat.

Položka v ASID tabulce musí obsahovat počítadlo referencí, aby nebylo možné znovupoužít ASID klíč pro nový adresní prostor zatímco se na něj ještě nějaké capability odkazují. Při každém namapování stránkovací tabulky či *page* se počítadlo zvýší, při odmapování sníží. Znovupoužití je možné, pokud je počítadlo na nule.

Výhody:

- **Výhoda č. 1**: Malá spotřeba paměti

  Toto řešení nepotřebuje žádné dodatečné capability pro sledování mapování do virtuální paměti. Postačí si s pouze již existujícími capabilitami reprezentující stránkovací tabulky a paměťové rámce k namapování. Samotné sledování mapování je poznačováné do těchto existujících capabilit. Z toho zároveň plyne nevýhoda č. 1.

  Nicméně pokud je paměťový rámec namapován dvakrát, je třeba mu vytvořit i druhou capabilitu *page*.

Nevýhody:

- **Nevýhoda č. 1**: Stránkovací tabulky nelze sdílet

  S tímto řešením nelze prakticky namapovat jednu stránkovací tabulku do vícero stromů stránkovacích tabulek. Ačkoliv nic nebrání možnosti vytvoření sourozeneckých capabilit ukazujících na stejnou stránkovací tabulku, neexistuje způsob, jak při odmapování stránky rozhodnout o invalidaci TLB v ovlivněných virtuálních adresních prostorech. Jak bylo vysvětleno výše, virtuální adresa je uložena v capability slotu reprezentující namapovaný paměťový rámec. Pokud je ale paměťový rámec namapován do stránkovací tabulky a tato stránkovací tabulka hypoteticky namapována do dvou stránkovacích tabulek vyšší úrovně, potom by tento paměťový rámec měl vlastně dvě různé virtuální adresy (v jednom či dvou virtuálních adresních prostorech). Prostor v capability slotu paměťového rámce je konstantní, potřeba paměti by ale rostla s počtem namapování stránkovacích tabulek. Toto tedy není možné.

  Takové omezení je možná přijatelné pro mikrokernel, ale protože exokernel se snaží zpřístupnit userspace softwaru vše, co hardware umožňuje, je toto omezení neatraktivní.

- **Nevýhoda č. 2**: Existence ASID tabulky

  ASID tabulka má pevně danou velikost.Indexy v ASID tabulce jsou tedy omezeným zdrojem a proto jsou spravované capability systémem, skrze který jsou rozdávána oprávnění indexy používat. Pokud je ASID tabulka vyčerpána, nelze vytvářet žádné další virtuální adresní prostory.

  Položku v ASID tabulce nelze znovupoužít, dokud nepřestanou existovat všechny capability reprezentující stránkovací tabulky a paměťové rámce na tuto ASID položku se odkazující. To je tím, že položka v ASID tabulce nemá žádný seznam na ni odkazujících se capabilit typu *page*; obsahuje pouze počítadlo referencí, podle kterého pozná, že už žádná nezbývá. Není tedy možné capabilitu reprezentující rozsah v ASID tabulce násilně odebrat revokací a rovnou použít původním majitelem. Spolu s omezeným počtem ASID je toto velký problém

pro operační systém na bázi exokernelu, protože chceme, aby i nedůvěryhodné aplikace mohly upravovat svůj virtuální adresní prostor. A přitom si chceme zachovat možnost revokovat cokoliv, co jsme této nedůvěryhodné aplikaci svěřili.

3.10.5 Barrelfish

Výzkumný multikernel Barrelfish je též capability-based kernel používající CDT pro reprezentaci capabilit. Zaujala mě především modifikace provedená Simonem Gerberem v jeho disertační práci[30], která používá radikálně jiné schéma než seL4. Jeho řešení budu níže popisovat.

Problematický koncept ASID tabulky byl opuštěn a místo toho se ukládají dodatečné informace o virtuální paměti do capability slotů.

Typ capability reprezentující mapovatelnou fyzickou pamět je pojmenován *frame*. Na rozdíl od capability *page* u seL4, *frame* nereprezentuje jeden paměťový rámec, ale rozsah paměťových rámců. Je tedy podobný mému *userspace memory*, ovšem moje řešení má bajtovou granularitu a *frame* rámcovou (4 KiB).

Prostor v capability slotu je použit pro držení druhého nezávislého stromu, a to pouze u capabilit typu *mapping*. Capabilita typu *mapping* je odvozená (ve smyslu CDT) z capability *frame* nebo z capability stránkovací tabulky a vzniká pro každé mapování těchto capabilit do virtuální paměti. Sekundární strom je typu AA strom, což je forma vyváženého binárního stromu. V capability slotu je uložen ukazatel na rodiče uzlu stromu a dva ukazatele na potomky. Tedy capability jsou stále normálně propojeny v CDT, kde CDT strom reprezentuje historii odvozování capabilit; ale navíc jsou ještě propojeny tímto sekundárním stromem, jenž je použit kernelem k zajištění výše definovaných invariantů. Kořenem stromu je *mapping* stránkovací tabulky, všechny ostatní uzly jsou capability *mapping* (rámce nebo stránkovací tabulky nižší úrovně) do této stránkovací tabulky namapovány. Díky tomuto stromu může kernel obsluhovat revokace *frame* i stránkovacích tabulek, protože jsou tímto stromem provázány a kernel je tak může libovolně procházet.

Při revokaci capability typu *mapping* kernel v této capabilitě vyšplhá sekundárním stromem ke kořenu. Kořenem je *mapping* odvozený od capability typu stránkovací tabulky, kam je revokovaná capabilita namapována, tudíž stačí odstranit záznam v tabulce a provést TLB invalidaci.

Poslední informace v dodatečném prostoru capability slotu u capabilit typu *mapping* je offset ve stránkovací tabulce a počet záznamů. To umožňuje jednou capabilitou *mapping* reprezentovat namapování do několika záznamů ve stránkovací tabulce. Toto lze provést pouze, pokud je mapovaná paměť fyzicky souvislá (protože *mapping* může mít v CDT jen jednoho rodiče) a i virtuálně souvislá (protože máme offset a počet).

- **Výhoda č. 1**: Sdílené tabulky

  Na rozdíl od seL4, toto schéma umožňuje sdílet stránkovací tabulky.

- **Výhoda č. 2**: Souvislé mapování používá jen jeden capability slot

  Protože capabilita *mapping* obsahuje offset do stránkovací tabulky a počet záznamů, lze fyzicky souvislý rozsah namapovaný do virtuálně souvislého rozsahu reprezentovat jen jednou instancí *mapping*. Tato technika umožňuje extrémní nízký paměťový overhead při mapování souvislé paměti.

- **Nevýhoda č. 1**: Velké capability sloty

  Toto řešení potřebuje v capability slotu mnoho informací, které by se jistě do mého současného 32B formátu nevešly.

- **Nevýhoda č. 2**: Cache unfriendly

  Při každém mapování nebo odmapování je potřeba procházet AA strom. Ve stránkovací tabulce je až 512 položek, tedy strom i při perfektním vyvážení může mít hloubku až 9. Toto

pravděpodobně bude způsobovat mnoho cache missů při procházení ke kořenu. Strom musí být navíc vyvažován.

3.10.6 FleK

Jako hlavní problém seL4 a Barrelfish jsem viděl fakt, že každé mapování vytváří novou capability. Přitom jediné, co kernel potřebuje v capability systému zaznamenat (aby mohl správě ošetřovat revokace), je provázání capability stránkovací tabulky a capability paměťového rámce. K tomu stačí dvojitý spojový seznam tvořící CDT, netřeba zbytek prostoru capability slotu.

S touto myšlenkou jsem se dostal ke konceptu, který je někdy označen jako *shadow page table*. Jako shadow page table se označuje forma stránkovací tabulky, která slouží softwaru, a jejíž obsah je synchronizován se její asociovanou hardwarovou stránkovací tabulkou, která slouží procesoru. Tento termín se používá také u virtualizačních technologií; je to naprosto nesouvisející s touto prací. Ve FleKu je *shadow page table* kernelový objekt, který je jednoduše pole 512 instancí *struct cdt_slot*, tedy 16B struktury uzlu v CDT (vizte 3.2.4). Tyto jsou poloviční oproti plnohodnotným capability slotům. Tato *shadow page table* je asociována s hardwarovou stránkovací tabulkou a pro každý rámec namapovaný do hardwarové stránkovací tabulky existuje příslušný záznam v *shadow page table*.

3.10.7 Porovnání

Pro porovnání řešení si vypočtěme jejich paměťový overhead v následujících modelových situacích. U všech je předpokládáno použití 4 KiB stránek na AMD64. Jako overhead se zde počítá jeden capability slot reprezentující celý rozsah fyzické paměti k namapování, capability reprezentující stránkovací tabulky a pomocné capability specifické pro kernel. Hardwarové stránkovací tabulky se nepočítají, protože ty jsou vždy vyžadovány procesorem.

1. Osamocená stránka

   Hypotetická situace, kdy je v celém virtuálním prostoru namapována jen jedna stránka. Situace demonstruje overhead při řídkém mapování, například první přístup po Unixovém mmap().

2. Souvislý rozsah 16 MiB

   V této situaci je souvislý rozsah ve virtuální paměti o velikosti 16 MiB namapován na 16 MiB souvislých i ve fyzické paměti. Virtuální adresa počátku je zarovnána na 512 GiB, aby nedocházelo k přelomům napříč stránkovacími tabulkami vyšších úrovní.

3. Fragmentovaný rozsah 16 MiB

   I v této situaci máme souvislý rozsah ve virtuální paměti o velikosti 16 MiB, ale žádné 2 stránky sousedící ve virtuální paměti nesousedí i ve fyzické paměti. Virtuální adresa počátku zarovnána na 512 GiB.

   Výsledky:

Tabulka 3.2 Overhead správy virtuální paměti

Situace	seL4	Barrelfish[5]	FleK	hardware
Osamocená stránka	ASID + 160 B	576 B	32 992 B	16 384 B
Souvislý rozsah 16 MiB	ASID + 131 424 B	1 920 B	90 672 B	45 056 B
Fragmentovaný rozsah 16 MiB	ASID + 131 424 B	263 552 B	90 672 B	45 056 B

[5] ve formě Simona Gerbera

Postup výpočtu je v příloze.

Problematická revokace a nemožnost sdílet stránkovací tabulky diskvalifikují řešení seL4 pro účely exokernelu a proto dále nebude diskutováno.

Při první situaci má mé řešení masivní overhead v porovnání s těmito capability-based kernely. Je třeba si ale uvědomit, že tato situace je velmi vzácná. Operační systémy mapují virtuální paměť ne jako osamocené stránky, ale jako velké rozsahy. V nich mohou být osamocené stránky (např. po prvním pagefaultu po mmap()), ale to je jen dočasný stav věci.

Řešení modifikovaného Barrelfishe je zdaleka nejefektivnější z porovnávaných kernelů, pokud mapujeme souvislý úsek fyzické paměti do souvislého úseku ve virtuálním adresním prostoru. To se může hodit v některých usecasech, ale v typickém operačním systému je paměť mapována zcela fragmentovaně. Po chvíli běhu operačního systému žádné velké nevyužité úseky fyzické paměti neexistují a alokace paměti pro userspace probíhá jako alokace jednotlivých rámců, namapovaná do souvislého úseku až ve virtuální paměti. Page cache klasických operačních systému invaliduje rámce podle LRU algoritmu prostorově náhodně a tyto uvolněné rámce jsou zaplňovány obsahem jiných souborů, než jaké jsou v sousedních rámcích, takže ani paměťové mapování souborů (.text a .rodata segmenty na Unixech) není fyzicky souvislé. Proto nás efektivita v druhé situaci příliš nezajímá.

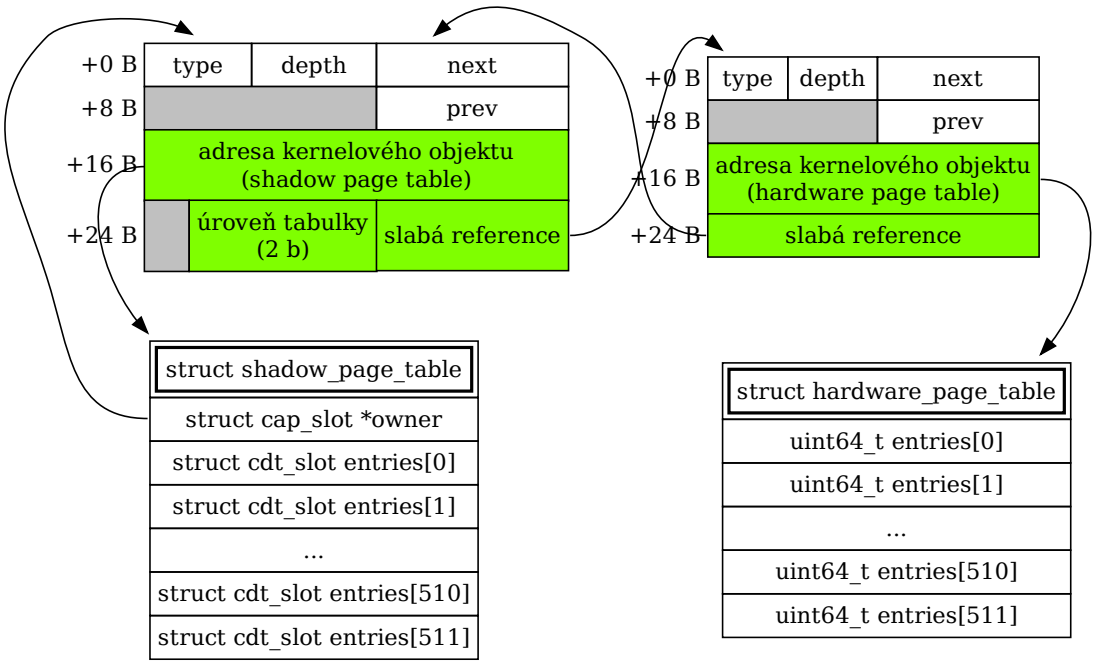
Další důvod, proč řešení Barrelfishe nelze použít u mého exokernelu je fakt, že exokernel bude mít velký počet capabilit v běžícím systému, protože v capability systému (na rozdíl od mikrokernelu) spravuje veškeré hardwarové zdroje a služby ovladačů. Zvětšení capability slotu znamená, že tato paměť bude spotřebovávána i u capabilit, které nemají s virtuální pamětí nic společného.

Nakonec benefit Barrelfishe z nízkého overheadu při fyzicky souvislém mapování je vlastně zbytečný, protože v takové situaci můžeme rovnou použít 2MiB stránky, které u seL4 a FlekU overhead sníží přibližně 512x.

Poslední situace je nejblíže tomu, co by se dělo, kdybychom chtěli implementovat Unix-like OS na bázi exokernelu. Ve výsledku z tohoto důvodu považuji svoje řešení za nejvhodnější, pokud chceme stavět operační systém pro PC. Všimněme si, že FleK zde má nejmenší overhead.

3.10.8 Implementační detaily

Existují dvě capability, jedna reprezentující hardwarovou stránkovací tabulku a jedna reprezentující *shadow page table*. Obě jsou vytvořené odvozením od *untyped memory*.



Obrázek 3.12 Capability sloty a kernelové objekty shadow page table a hardwarové stránkovací tabulky

Kernelový objekt capability hardwarové stránkovací tabulky je 4096B přirozeně zarovnané pole 512ti 8B záznamů, tak jak ho vyžaduje architektura AMD64. Kernelový objekt *shadow page table* obsahuje 512 instancí *struct cdt_slot* a ukazatel (slabá reference) na capability slot *shadow page table*. Protože tyto dva kernelové objekty musí být synchronizovány v obsahu, musejí na sebe mít také nějaké reference. Ty existují ve formě ukazatele v capability slotu ukazujícím na druhý capability slot. Po spárovaní je capabilita hardwarové stránkovací tabulky z pohledu userspacu pasivním prvkem; systémová volání se provádějí adresovány na *shadow page table*.

Když dojde k mapování capability typu *userspace memory* do *shadow page table*, dojde k zápisu do spárované hardwarové stránkovací tabulky a vytvoření odvozené "capability"v příslušném *struct cdt_slot* v *shadow page table*. Tímto si kernel drží informaci o tom, které paměťové capability jsou vlastně kde mapovány.

Trik mého kernelu spočívá v tom, že existuje 512 různých konstant typu capability (*mapping0* až *mapping511*), každá používána pro jeden index v poli *struct cdt_slot* v *shadow page table*. Pokud je revokován paměťový rámec, je třeba ho odmapovat z hardwarové stránkovací tabulky a zjistit jeho virtuální adresu, abychom mohli invalidovat TLB. To probíhá díky návrhovému vzoru connector, kde při revokaci paměťového rámce dojde automaticky k revokaci od něj odvozených capabilit, což způsobí například revokaci capability typu *mapping17*. Díky tomuto typu víme, že existuje na indexu 17 v poli *struct cdt_slot* v kernelovém objektu *shadow page table*. Z adresy capability slotu revokované capability typu *mapping17* jsme tedy schopni vypočítat počátek objektu *shadow page table* a díky v něm existujícím ukazateli na jeho capability slot tedy i capability slot *shadow page table*. Odtud díky provázání capability *shadow page table* a capability hardwarové stránkovací tabulky dojdeme i k samotné hardwarové stránkovací tabulce a teď už můžeme smazat příšlušný záznam užívaný procesorem.

Ještě zbývá invalidace TLB a k tomu musíme zrekonstruovat virtuální adresu odma-

povávaného rámce. Protože jakožto exokernel chceme podporovat vše, co podporuje hardware, FleK (na rozdíl od seL4) umí mapovat jednu stránkovací tabulku do vícero stránkovacích tabulek. **To znamená, že jeden namapovaný paměťový rámec může mít dokonce několik virtuálních adres. Každá z nich musí být zrekonstruována a podle potřeby invalidována.** Rekonstrukce je přímočarý úkol – z instance *shadow page table* z předchozího odstavce najít všechny *shadow capability table*, kde je tato *shadow page table* namapována. Eventuálně dojdeme k *shadow page table* nejvyšší úrovně a podle zatím zapamatovaných pozic (které víme z prošlých *mapping⟨číslo⟩*) zpětně zrekonstruovat všechny virtuální adresy čehokoliv, co zrovna odmapováváme. Samotné *shadow page table* jsou do *shadow page tables* vyšší úrovně mapovány zcela stejným způsobem, tedy také jsou od nich odvozovány capability typu *mapping⟨číslo⟩*. Tímto mechanismem je zajištěn invariant č. 2, č. 3 a č. 6.

V kontrastu s modifikovaným Barrelfish jsme tedy schopni přejít od odmapovávaného paměťového rámce ke stránkovací tabulce, kde je namapován, v konstantním čase nehledě na počet rámců namapovaných do této tabulky.

Samotná invalidace TLB používá instrukci invlpg[22, str. 182] pro jednotlivé odmapovávané položky; jinak znovunastavení registru *CR3*, což způsobí zahození celé TLB.

Invariant č. 1 je zajištěn triviálně – při systémovém volání k namapování paměťového rámce musí aplikace předložit capabilitu k tomuto rámci. Invarianty č. 4 a č. 5 jsou zajištěny mechanismem popsaným v kapitole *Správa fyzické paměti paměti3.5*, kde odvozovat od capability *untyped memory* lze pouze, pokud zatím nemá potomky, což zajišťuje disjuktní rozsahy fyzické paměti. Tedy capabilita *userspace memory* nikdy nemůže reprezentovat stejný rozsah paměti, jako kde je stránkovací tabulka. K zajištění invariantu č. 5 ještě musíme zabránit aplikaci, aby namapovala příliš "plochý"strom stránkovacích tabulek, protože pak by obyčejná paměť namapovaná na nejnižší úrovni byla interpretována procesorem jako stránkovací tabulka a to by umožnilo aplikaci například zapisovat do kernelspace. K tomuto slouží informace uvnitř capability slotu *shadow page table* popisující její úroveň ve stromu (například PML4). A stránkovací tabulky pak mohou být mapovány jen o jednu nižší úroveň do vyšší.

Poslední rozdíl mého návrhu oproti seL4 a Barrelfish je způsob reprezentace úrovně stránkovací tabulky. seL4 a Barrelfish mají každou úroveň samostatný typ capability stránkovací tabulky. To je jednoduchý způsob, jak zabránit chybnému propojování stránkovacích tabulek škodlivým userspacem (vizte invariant č. 5). Problém je, že userspace si bude chtít tyto capability vytvářet ve větších počtech najednou a držet je v nějaké cachi/alokátoru. To by znamenalo mít jednu samostatnou cache pro capability každé úrovně. Aplikace může chtít alokovat stránkovací tabulku úrovně PML4, ale její cache je prázdná a potřebuje doplnit, i když cache stránkovacích tabulek úrovně PDP má velké zásoby. To je neefektivní, z pohledu spotřeby paměti i potřeby provádět systémová volání. Ve svém návrhu jsem se rozhodl mít jen jeden typ capability hardwarové stránkovací tabulky, nehledě na její úroveň. To umožňuje userspacu mít jen jednu cache pro capability stránkovacích tabulek, a tím pádem menší paměťový overhead. Úroveň musí být userspacem rozhodnuta před prvním mapováním, je zapsána do capability slotu a pak už není dovoleno ji měnit (až zase do posledního odmapování).

Speciální situací je možnost namapovat capabilitu typu hardwarová stránkovací tabulka do *shadow page table*. To potřebujeme ke splnění dodatečného požadavku č. 1. K zajištění invariantu č. 4 poslouží jednoduché omezení na mapování pouze pro čtení. Z pohledu *mapping⟨číslo⟩* a revokací funguje stejně jako dříve popsané situace.

A nakonec: proč jsou *shadow page table* a hardwarová stránkovací tabulka oddělené objekty a ne jeden objekt? Protože architektura AMD64 vyžaduje, aby jeho stránkovací tabulka byla zarovnána na stránku. Velikost pole *struct cdt_slot* v *shadow page table* je násobkem velikost stránky, ale ještě je tam ukazatel na svůj reprezentující capability slot. Userspace bude chtít vytvářet jedním systémovým voláním mnoho kernelových objektů z jedné capability *untyped memory*. Důsledkem je, že kdybychom chtěli takto vytvářet sjednocený objekt, mnoho paměti by bylo vyplýtváno na padding.

Ačkoliv *shadow page table* a hardwarová stránkovací tabulka jsou dvě capability, userspace si

je před prvním použitím na sebe naváže a dále už se k nim chová jako k jedné capabilitě (manipuluje jen s *shadow page table*). Protože tyto dvě capability jsou provázány slabými referencemi, kdykoliv je jedna z nich revokována, z druhé se slabá reference taky smaže.

Nelze provádět mapování paměťových rámců před navázáním těchto dvou capabilit. Pokud je navázání zrušeno po namapování, je mapování zrušeno. Prostě platí invariant, že nemůže existovat mapování v *shadow page table* nebo hardwarové stránkovací tabulce, aniž by jedna byla svázána s tou druhou.

3.10.9 Systémová volání shadow page table

label	FLEK_SYSCALL_SPT_COUPLE
argumenty userspacu	úroveň tabulky
	capability index ukazující na capabilitu typu hardwarová stránkovací tabulka
návratové hodnoty	žádné

Propojí *shadow page table* s hardwarovou stránkovací tabulku. Zde je také určena úroveň této stránkovací tabulky. Úroveň je pak už neměnná, dokud nejsou SPT a hardwarová stránkovací tabulka opět rozpojeny. Neměnnost úrovně stránkovací tabulky je součástí mechanismu zajišťujícího invariant č. 5.

label	FLEK_SYSCALL_SPT_DECOUPLE
argumenty userspacu	žádné
návratové hodnoty	žádné

Rozpojí propojené *shadow page table* a hardwarovou stránkovací tabulku. Veškerá namapování do a z této tabulky jsou zrušena (dojde k automatické revokaci capabilit *mapping*⟨*číslo*⟩). Také dojde k revokace odvozených capabilit typu *shadow page table*.

K rozpojení dojde automaticky, pokud je jeden z propojeného páru revokován.

label	FLEK_SYSCALL_SPT_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu, reprezentující stejnou *shadow page table*. Odvozeniny nelze vytvářet, pokud tato *shadow page table* není provázána s hardwarovou stránkovací tabulkou. To je nutné omezení k zachování korektního stavu slabých referencí.

label	FLEK_SYSCALL_SPT_MAP_SPT
argumenty userspacu	index (0–511) ve stránkovací tabulce, kam budeme prvek mapovat
	bity oprávnění
	capability index odkazující na capabilitu typu *shadow page table*
návratové hodnoty	žádné

Namapuje stránkovací tabulku nižší úrovně do tabulky vyšší úrovně. Selže pokud se jejich úrovně liší o více než 1, pro zachování invariantu č. 5.

U stránkovací tabulky úrovně PML4 je navíc omezení, že index může mít maximální hodnotu 255. Je to tím, že v horní polovině virtuálního adresního prostoru existuje kernel.

label	FLEK_SYSCALL_SPT_MAP_USMEM
argumenty userspacu	index (0–511) ve stránkovací tabulce, kam budeme prvek mapovat
	bity oprávnění
	capability index odkazující na capabilitu typu *userspace memory*
návratové hodnoty	žádné

Namapuje fyzickou paměť do stránkovací tabulky. Lze provést jen nad tabulkami úrovně PT (4 KiB stránka), PD (2 MiB stránka) a PDPT (pokud má CPU podporu 1GiB stránek, jinak vrátí chybový kód). Tím splňujeme dodatečný požadavek č. 2.

label	FLEK_SYSCALL_SPT_MAP_HWPT
argumenty userspacu	index (0–511) ve stránkovací tabulce, kam budeme prvek mapovat
	capability index odkazující na capabilitu typu hardwarová stránkovací tabulka
návratové hodnoty	žádné

Namapuje hardwarovou stránkovací tabulku do stránkovací tabulky, jako kdyby byla obyčejná fyzická paměť k přímému přístupu. To umožňuje aplikaci číst *accessed* a *dirty* bity vyplňované procesorem. Tím splňujeme dodatečný požadavek č. 1.

Povšimněme si, že nelze specifikovat bity oprávnění. To je tím, že userspacu můžeme dovolit přistupovat do obsahu stránkovacích tabulek pouze jako read-only. Tím splňujeme invariant č. 4.

label	FLEK_SYSCALL_SPT_REMAP
argumenty userspacu	index (0–511) prvku ve stránkovací tabulce, jehož mapování chceme pozměnit
	nové hodnoty bitů oprávnění
návratové hodnoty	žádné

Na indexu bude namapován stále ten samý prvek. Změní se jen jeho bit oprávnění (čtení, zápis, spustitelnost). Pochopitelně nelze žádat o zvýšení oprávnění nad rámec toho, co capabilita (např. *userspace memory*) předložená při mapování měla dovoleno.

Toto systémové volání je jen takové *nice-to-have*. Můžeme samozřejmě ke stejnému účelu použít SPT_MAP, ale tam musíme znovu předložit capabilitu prvku k namapování. Zde nemusíme.

To funguje, protože capability typu *mapping⟨číslo⟩* v sobě drží bity oprávnění použity pro vzniku mapování. Z toho plyne trochu překvapivý fakt, že lze pomocí SPT_REMAP snížit bity oprávnění namapovaného prvku – a pak znovu pomocí SPT_REMAP je zvýšit na původní úroveň, bez potřeby znovu předkládat capabilitu namapovaného prvku. Toto se může hodit při implementaci copy-on-write mechanik (resp. toto je skutečně užito PALem).

Dojde k updatu TLB, ale jen tehdy pokud je to potřeba; tedy při snížení oprávnění.

label	FLEK_SYSCALL_SPT_UNMAP
argumenty userspacu	index (0–511) prvku ve stránkovací tabulce, který chceme odmapovat
návratové hodnoty	žádné

Odmapuje prvek z virtuálního adresního prostoru.

3.11 Envelope ring

Zatím byla popsána komunikace mezi userspacem a exokernelem pouze jedním směrem, a to pomocí systémových volání. Někdy je ale nutné iniciovat komunikaci i v opačném směru, tedy z kernelu do userspacu.

V Unixových systémech je toto implementováno pomocí signálů. Každý proces má ještě druhý zásobník, na kterém Unixový kernel spouští příslušný signal handler[31]. Po návratu ze signálu Unixový kernel obnoví původní hodnoty registrů a program pokračuje na hlavním zásobníku tam, kde předtím skončil. Toto řešení se u exokernelu nehodí, protože exokernel nezná nic jako procesy. Zná pouze vlákna v nejjednodušším slova smyslu a nezajímá se o relativně vysokoúrovňovou abstrakci operačního systému jako jsou signály.

Jelikož už máme implementovaný dosti univerzální mechanismus zasílání zpráv mezi vlákny (vizte kapitola IPC 3.8), můžeme posílat zprávy z exokernel skrze něj. Deterministická alokace paměti kernelem nám tady ale háže klacek pod nohy: exokernel chce poslat zprávu, zatímco userspace vlákno nemusí být schopno zprávu okamžitě přijmout. Exokernel ale musí běžet dál - nemůže jako synchronně posílající userspacové vlákno spát, dokud nebude přijímající vlákno připraveno. To znamená, že posílaná zpráva musí být někde dočasně odložena, exokernel bude pokračovat a přijímající vlákno si odloženou zprávu později přečtě.

Kernelu s deterministickou alokací paměti nemůžeme dynamicky alokovat nějaké pole. Prostor pro takové zprávy musí být poskytnut userspacem.

Jelikož jsme koncept deterministické alokace paměti převzali od mikrokernelu seL4, je vhodné podívat se, jak je tento problém řešen tam. seL4, jakožto mikrokernel implementující ovladače hardwaru v userspacu, potřebuje komunikovat směrem z kernelu do userspace pouze při obsluze hardwarového přerušení. Tímto se liší od mého exokernelu, který implementuje ovladače v kernelspace, a proto je komunikace směrem z kernelu do userspacu mnohem bohatší. seL4 poskytuje capability typu tzv. *notification object* a *IRQHandler*[17], které lze napojit na capability typu endpoint. Protože obyčejná hardwarová přerušení nemají žádné "parametry", není u seL4 potřeba žádného místa pro uložení zprávy. Některá speciální přerušení jako například pagefault "parametry"mají, pomocí kterých procesor sdělí, na které virtuální adrese došlo k faultu. U takových stačí uložit tuto informaci dovnitř toho TCB, které takové speciální přerušení svým vykonáváním vyvolalo (toto TCB bude pozastaveno, dokud pagefault nevyřešíme).

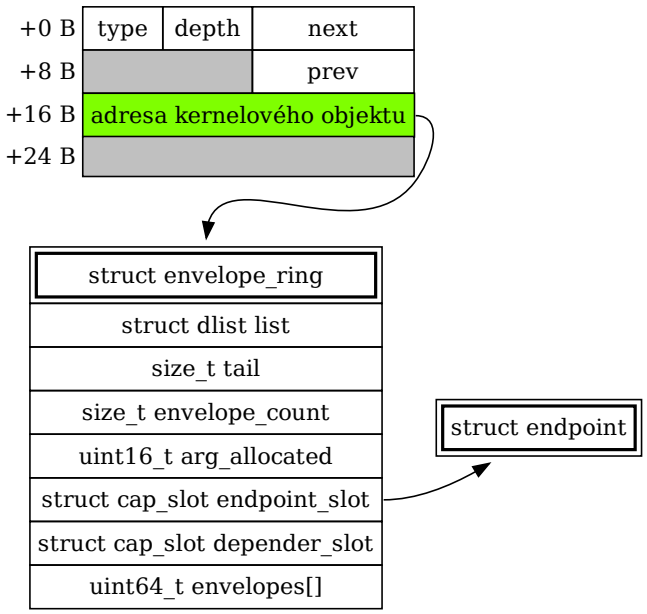
Řešení tohoto typu nelze u mého exokernelu použít. FleK potřebuje předávat userspacu téměř arbitrární počet parametrizovaných zpráv s arbitrárním načasováním. Tyto zprávy jsou generovány ovladači hardwaru víceméně nezávisle na aktivitách userspacu, a proto nelze použít staticky alokovaný prostor v TCB.

Proto jsem musel vymyslet nové řešení a tím je tzv. *envelope ring*. Jedná se o typ capability, která reprezentuje kus fyzické paměti věnovaný userspacem exokernelu. Do této paměti může exokernel ukládat zprávy pro userspace, dokud si je userspace nevyzvedne.

Prostor pro jednu zprávu jsem nazval *envelope*. Reprezentovat jeden envelope jako samostatnou capabilitu by mělo příliš vysoký paměťový overhead v CDT, proto je větší počet envelopes reprezentován jednou capabilitou nazvanou *envelope ring*. Protože zprávy jsou mým exokernelem posílané sekvenčně a userspacem sekvenčně také odebírány (tj. v pořadí odeslání), jedná se o ring buffer, kde exokernel je *producer* a userspace *consumer*. Pokud všechny envelopes v envelope ringu už čekají s obsahem zprávy v endpointu a zároveň exokernel potřebuje poslat další zprávu, pak je nejstarší čekající zpráva recyklována pro tuto další zprávu. Tedy pokud je userspace příliš pomalý, jsou staré zprávy zahazovány. Proto je dobrý nápad vytvářet větší envelope ringy jako buffer.

Z pohledu userspacu a sémantiky IPC operací můžeme envelope považovat za jakési virtuální vlákno synchronně posílající zprávu skrze endpoint.

3.11.1 Capability slot



Obrázek 3.13 Envelope ring capability slot a kernelový objekt

Políčko *envelopes* je ve skutečnosti pole instancí typu envelope. Ale protože envelope má proměnnou velikost (proměnný počet argumentů) a samotný počet envelopes v envelope ring je také proměnný, tak z hlediska syntaxe jazyka C musíme použít VLA pole primitivního typu a na příslušné pozici přetypovávat na envelope.

envelope_count drží počet envelopes a *arg_allocated* drží počet argumentů v každém envelope. Dohromady z těchto dvou hodnot můžeme vypočítat velikost pole *envelopes*.

tail je pozice producera (tj. kernelu) v ring bufferu.

endpoint_slot obsahuje odvozenou capabilitu typu endpoint přiřazenou systémovým voláním. Do tohoto endpointu budou posílány zprávy.

Posledními dvěma políčky k vysvětlení jsou *list* a *depender_slot*. Účelem těchto dvou políček je přiřazení envelope ringu nějakému zdroji událostí, který poté skrze tento envelope ring generuje zprávy. Není to tak jednoduché, jak se zdá; musí být splněny následující dva požadavky

- Požadavek č. 1: revokace envelope ringu musí být detekovatelná zdrojem událostí

  Jestliže přiřadíme envelope ring zdroji událostí a pak proběhne revokace envelope ringu, zdroj událostí už pochopitelně nebude schopen posílat další zprávy. Ale to nestačí, někdy potřebuje zdroj událostí být okamžitě informován, že k tomuto došlo. Např. aby mohl provést nějaké uklizení zdrojů.

- Požadavek č. 2: revokace zdroje událostí musí odpojit zdroj od envelope ringu

  Představme si situaci, kdy přiřadíme envelope ring ke zdroji událostí. Ale pak je capabilita zdroje událostí revokována. V ten moment jsme ztratili oprávnění zdroj událostí používat. Pokud by náš envelope ring i nadále generoval zprávy z tohoto zdroje, bylo by to narušení bezpečnosti systému.

Oba požadavky můžeme splnit vhodným použitím návrhového vzoru connector. Odvozenou capabilitu zdroje událostí zapíšeme do vnitřního slotu envelope ringu (*depender_slot*) a odvozenou capabilitu envelope ringu zapíšeme do kernelového objektu zdroje událostí. Druhý zápis slouží k tomu, aby zdroj událostí vlastně měl přístup k objektu envelope ringu; první zápis bude mít svůj vlastní "podtyp"(vizte 3.9.2). Protože *depender_slot* bude obsahovat potomka capability zdroje událostí, kterou jsme připojili, tak bude při revokace této capability zdroje událostí také automaticky revokována. Podle podtypu víme, kterou cleanup akci (vzhledem ke zdroji událostí) provést, čímž splníme požadavek č. 1. Mimojiné tato reakce bude moci obsahovat revokaci vnitřního slotu objektu zdroje událostí (obsahující odvozeninu envelope ringu), čímž splníme i požadavek č. 2.

Některé zdroje událostí nemají žádné kernelové objekty. Příkladem je capabilita typu PS/2 klávesnice, v jejímž slotu není žádný ukazatel na žádný objekt. PS/2 klávesnice je v systému pouze jedna.

V tom případě se použije políčko *list*, které se prováže do ovladače PS/2 klávesnice. Políčko *depender_slot* je stále používáno, pro splnění obou požadavků výše. Odvozenina capability typu PS/2 klávesnice bude zapsána do *depender_slotu*, revokace capability PS/2 klávesnice skrze návrhový vzor connector způsobí odpojení políčka *list* envelope ringu od ovladače PS/2.

Protože *depender_slot* musí vždy obsahovat odvozeninu zdroje událostí, důsledkem je, že jeden envelope ring může naslouchat jen jednomu zdroji. Toto není žádným problémem, protože několik envelope ringů může generovat zprávy do stejného endpointu. Jedno vlákno tedy může na jednom endpointu naslouchat na události z několika zdrojů, přičemž přijímané zprávy si zachovávají správné pořadí generování.

3.11.2 Systémová volání

adresovaná capabilita	netypovaná paměť
label	FLEK_SYSCALL_UTMEM2ER
argumenty userspacu	capability index prvního vytvořeného envelope ringu
	počet ringů
	počet argumentů v envelope
návratové hodnoty	žádné

Vytvoří jeden či více envelope ringů v daném rozsahu fyzické paměti. Každý envelope bude mít daný počet argumentů.

Selže, pokud capabilita netypované paměti velikostně nestačí. Envelope ringy jsou vytvářeny postupně. Pokud už je capability index pro nějaký ring zabraný, systémové volání se ukončí s chybou. Dosud vytvořené ringy ale zůstanou platné a použitelné.

Počet argumentů v envelope je něco, co userspace musí nějakým způsobem předvídat. Pokud bude exokernel potřebovat poslat zprávu s větším počtem argumentů, než s jakým jsme příslušný envelope ring vytvořili, exokernel zprávu zahodí.

label	FLEK_SYSCALL_ENVELOPE_RING_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu reprezentující stejný envelope ring. Toto může být užitečné, například pokud chce používat stejný envelope ring objekt napříč vlákny.

label	FLEK_SYSCALL_ENVELOPE_RING_ASSIGN_ENDPOINT
argumenty userspacu	capability index ukazující na capabilitu typu endpoint
návratové hodnoty	žádné

Přiřadí endpoint tomuto envelope ringu. Když bude exokernel posílat zprávu skrze tento envelope ring, bude zpráva poslána do tohoto endpointu. Badge zprávy bude badge endpoint capability použité v tomto systémovém volání.

V jeden moment může mít envelope ring přiřazen pouze jeden endpoint. Pokud toto systémové volání provedeme, když už byl dříve endpoint přiřazen, je původní endpoint odpojen.

Pokud je endpoint revokován, je automaticky odpojen od envelope ringu. Veškeré envelopy čekající v tomto endpointu jsou považovány za přečtené a připravené k dalšímu použití exokernelem.

Pokud exokernel pošle zprávu, zatímco envelope ring nemá přiřazený endpoint, zpráva bude zahozena (protože envelope by pak neměl v jaké frontě čekat).

Pokud jsou všechny capability typu envelope ring odkazující na stejný kernelový objekt revokovány, jsou všechny jeho envelopy čekající v endpointu z endpointu odebrány.

3.12 PS/2

PS/2 rozhraní je použito k přístupu ke klávesnici a myši/touchpadu.

Capabilita typu *envelope ring* je přiřazena capabilitě reprezentující PS/2 klávesnici, čímž tento *envelope ring* generuje zprávy pro každý stisk a uvolnění klávesy.

3.12.1 Systémová volání

label	FLEK_SYSCALL_PS2_KEYBOARD_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu reprezentující stejnou klávesnici.

label	FLEK_SYSCALL_PS2_KEYBOARD_ASSIGN_ENVELOPE_RING
argumenty userspacu	capability index capability typu *envelope ring*
návratové hodnoty	žádné

Přiřadí *envelope ring* k této klávesnici. Od tohoto momentu bude envelope ring generovat zprávy. Pokud je revokována capabilita typu *PS/2 klávesnice*, které byla použita během tohoto přiřazení, tak dojde k odpojení *envelope ringu* a generování zpráv přestane.

Formát zprávy je jednoduchý. Label je nastaven na konstantu FLEK_PS2_KEYBOARD_LABEL_SCANCODE (0) a počet argumentů je 1. Jediný argument je číslo události, které kóduje identifikátor klávesy a zda došlo ke stisku či uvolnění.

3.12.2 Myš/touchpad

Myš a touchpad jsou z pohledu PS/2 stejným zařízením. FleK je reprezentuje společnou capabilitou. Její rozhraní je zcela identické s capabilitou klávesnice, tudíž její systémová volání nebudu dále rozvádět.

Jediný rozdíl je formát zprávy. Capabilita myši skrze *envelope ring* předává zprávy o třech argumentech; dva jsou signed integery popisující posun myši a jeden obsahuje přiznaky se stavem tlačítek (levé, střední, pravé).

Je naprosto klíčové, aby aplikace byla schopna přijímat události klávesnice i myši v jednom proudu zpráv ve správném vzájemném pořadí[32]. Aplikace toto může zajistit snadno: klávesnici i myši přiřadit po jednom *envelope ringu* a oba *envelope ringy* napojit na stejný endpoint. Teď už stačí jen přijímat zprávy na tomto endpointu. Pak se nemůže stát, že by například zpráva o stisku klávesy na klávesnici nesprávně předběhla zprávu o stisku tlačítka na myši.

3.13 NVMe

NVM Express (NVMe) je standard definující hardwarové rozhraní řadiče disku. FleK implementuje základní verzi 1.0[8].

3.13.1 Chování hardware

Základem komunikace s NVMe jsou páry kruhových bufferů. Jeden slouží ke vkládání požadavků (*submission queue*) a druhý slouží k přijímání výsledků (*completion queue*). Kernel zapíše nový požadavek do submission queue a nastaví memory-mapped register na poslední vloženou položku. Tímto zápisem se zařízení dozví o nových požadavcích. Zařízení poté vyplňuje completion queue a když je nový výsledek vložen, je vyvoláno přerušení, čímž se kernel dozví o novém výsledku.

Existují minimálně dva páry, jeden pro ovládání samotného řadiče (*admin queue*) a jeden pro samotné I/O operace (*I/O queue*). Párů může být více, NVMe byl navržen pro efektivnější práci na SMP počítačích, kde každé jádro procesoru může mít svůj pár, čímž je redukován lock contention v kernelu. FleK tohoto nevyužívá, protože nepodporuje SMP.

Zajímavostí je fakt, že NVMe řadič neprovádí zadané požadavky v pořadí vkládání, ale podle svého uvážení. Je efektivní vložit mnoho požadavků najednou a nechat řadič je vykonávat v ideálním pořadí či paralelně.

Řadič NVMe provádí *scatter-gather* s granularitou paměťových rámců (může a nemusí podporovat i jiné). Jen v první stránce může přenos začít s offsetem od začátku stránky; jen v poslední stránce může být přenos splněn před koncem stránky. U malých přenosů může být stránka pod těmito kritérii chápana jako první a poslední zároveň.

NVMe 1.0 definuje i několik volitelných funkcionalit. Mezi ně patří například hardwarově akcelerovaná implementace round robin algoritmu s volitelnými váhami pomocí několika kruhových bufferů. Toto není FleKem podporováno. Jediná volitelná funkcionalita podporovaná FleKem je hardwarově akcelerované porovnávání bloků s pamětí (tj. bez potřeby čtení do paměti a porovnávání na CPU).

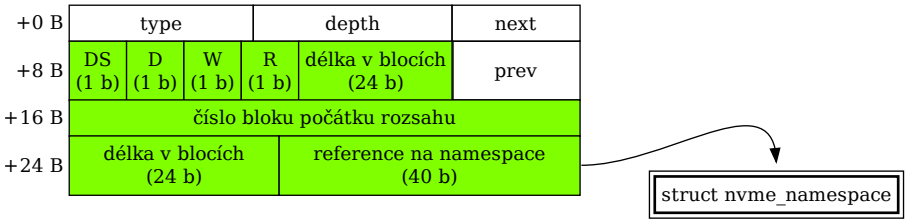
3.13.2 Požadavky

3.13.3 Implementace

Implementace exokernelového API pro NVMe sestává ze 4 typů capabilit. Jedna capabilita, *NVMe block range*, reprezentuje rozsah bloků na disku a bity oprávnění. Druhá capabilita, *NVMe slice*, reprezentuje zlomek bandwidthu diskového řadiče. Třetí capabilita, *NVMe ring*, reprezentuje kruhový buffer pro vkládání požadavků. Závěrem, *NVMe shadow PRPL*, je čistě kernelu vnitřní typ capability, jenž je užíván ovladačem NVMe pro zabránění revokací paměťových capabilit, do kterých právě probíhá DMA.

3.13.4 NVMe block range

NVMe block range je jednoduchý typ capability popisující rozsah bloků na disku.



Obrázek 3.14 NVMe block range capability slot

Je podobný capabilitám popisujícím rozsah fyzické paměti – obsahuje počátek rozsahu a jeho délku, ovšem v blocích a ne v bajtech. Stejně tak obsahuje bity oprávnění a to čtyři: pro čtení, zápis, DMA a změnu datasetu.

Oprávnění ke čtení a zápisu nepotřebují vysvětlení. Oprávnění k DMA zní zvláštně, protože k čemu by byla capabilita k rozsahu bloků bez možnosti provádět DMA operace?

Tento bit oprávnění je součástí DMA systému vysvětleného v kapitole 3.5.4. Pokud se aplikace prokáže capabilitou typu *NVMe block range* při požadavku např. o čtení z disku, bude ovladačem NVMe vytvořena odvozená DMA capabilita. Tím se zabrání aplikaci revokovat *NVMe block range*, z/do kterého právě probíhá DMA operace, což by jinak mohlo způsobovat nemilá překvapení v userspacu. Pokud takto revokace selže, je ale stále odebrán bit oprávnění k DMA, což zabrání aplikaci zahájit další DMA operaci po pokusu o revokaci (což je prevence *denial of service* proti revokacím).

Bit oprávnění ke změně datasetu je úzce svázán s nízkoúrovňovými detaily NVMe. Ve zkratce umožňuje userspacové aplikace specifikovat informace o svých budoucích přístupových vzorech, což umožní diskovému řadiči pracovat efektivněji. Toto však mutuje globální stav diskového řadiče a proto je tato možnost chráněna bitem oprávnění v této capabilitě.

Interně ještě tato capabilita drží referenci na kernelspace strukturu popisující, kterého NVMe disku se capabilita týká. Proto nelze použít capabilitu k rozsahu bloků 5-10 pro disk A ke čtení bloku 7 z disku B. Tato struktura není reprezentována touto capabilitou ve smyslu, že bychom ji museli alokovat z *untyped memory*. Vytváří si ji ovladač.

3.13.5 Systémová volání NVMe block range

Systémová volání zahrnují dotazovací volání, odvození celého rozsahu, dělení, spojení a odvození podmnožin. Jedná se o stejnou množinu volání, jako u *userspace memory* (vizte kapitola 3.5.8).

Rozdílem je jen fakt, že zde pracujeme v rozlišení ne bajtů, ale bloků. Jiné jsou také bity oprávnění, vizte výše.

Z těchto důvodů zde nebudu téměř duplikovat popisy volání *userspace memory*.

Jeden problém, kterému bude userspace čelit, je zjišťování, ke kterému disku vlastně capabilita rozsahu bloků patří. K tomu má *NVMe block range* následující systémové volání.

label	FLEK_SYSCALL_NVME_BLOCK_RANGE_GET_IDS
argumenty userspacu	žádné
návratové hodnoty	namespace ID
	PCI kód výrobce
	sériové číslo (20 B)
	modelové číslo (40 B)

V podstatě navrací hodnoty, které ovladač získá provedením příkazu *Identify*[8, str. 71]. PCI kód výrobce, sériové číslo a modelové číslo jsou vráceny skrze MRS. Volání selže, pokud vlákno nemá přiřazené MRS.

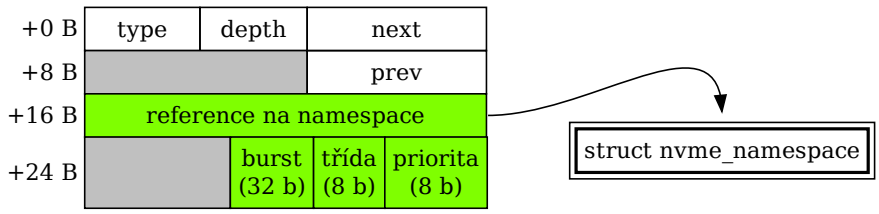
Namespace ID je identifikátor disku v rámci jednoho diskového řadiče.

label	FLEK_SYSCALL_NVME_BLOCK_RANGE_GET_FEATURES
argumenty userspacu	žádné
návratové hodnoty	příznaky volitelných funkcionalit
	maximální velikost požadavku (ve stránkách)
	velikost bloku

Tato metoda nám umožňuje zjistit, jaké dodatečné funkcinality příslušný diskový řadič podporuje. V současnosti lze přes příznaky zjistit podpora hardwarově akcelerované porovnávací operace a podpora *dataset management* příkaz[6].

Maximální velikost požadavku nám říká, jak velký přenos diskový řadič zvládne jako jednu operaci. A velikost bloku je jasná; typicky 512 B, ale NVMe může podporovat i jiné velikosti.

3.13.6 NVMe slice



Obrázek 3.15 NVMe slice capability slot

Tato capabilita je v podstatě je ekvivalentem capability *CPU slice*. Ovladač NVMe ve FleKu používá plánovač round robin s prioritami, stejně jako plánovač úloh pro CPU. V důsledku toho má capabilita *NVMe slice*, stejně jako *CPU slice*, prioritu a burst.

Různé aplikace mohou provádět skoropřímé NVMe operace skrze exokernelové API a my chceme mít možnost umožnit nějaké aplikaci prioritní přístup. Proto *NVMe slice* obsahuje prioritu. Požadavky s vyšší prioritou mají vždy přednost před požadavky s nižší prioritou.

Na co je však potřeba burst? U plánovače úloh pro CPU používáme burst, protože přepínání úloh má nějaký výpočetní overhead a tak ho nechceme provádět častěji než je nutné. U NVMe operací čelíme podobnému problému, protože aplikace může specifikovat přístupové vzory. Pokud bychom příliš často kombinovali proudy požadavků od dvou aplikací, specifikace přístupových vzorů by se často přebíjely a jejich benefit by byl ztracen. Burst je vyjádřen v počtu paměťových rámců, které byly přeneseny (”spotřebovány”aplikací).

Plánovač FleKu pro NVMe tedy neustále krmí diskový řadič požadavky z nějakého *NVMe ringu*, dokud mu nedojde burst. Poté plánovač přepne na další *NVMe ring* a jeho požadavky předává diskovému řadiči. A tak stále dokola. Pokud se *NVMe ring* vyprázdní, je odejmut z fronty plánovače, aby uvolnil místo dalším čekajícím *NVMe ringům*, dokud do něj aplikace zase nevloží nějaký požadavek.

[6] FleK toto prozatím nepodporuje, nicméně je schopen to detekovat

Vhodným nastavením burstů ve stejné prioritě vlastně alokujeme zlomky bandwidthu diskového řadiče různým aplikacím.

Dále musí mít referenci na *struct nvme_namespace*, aby kernel věděl, jakého disku se tato capabilita týká.

Zajímavostí je políčko *třída*. NVMe volitelně podporuje vážený round robin na hardwarové úrovni. Pokud by FleK podporoval tuto funkci, existovaly by 4 třídy: low, medium, high, urgent a admin. Protože FleK tuto funkcionalitu nepodporuje, existují zatím jen 2 třídy: urgent a admin.

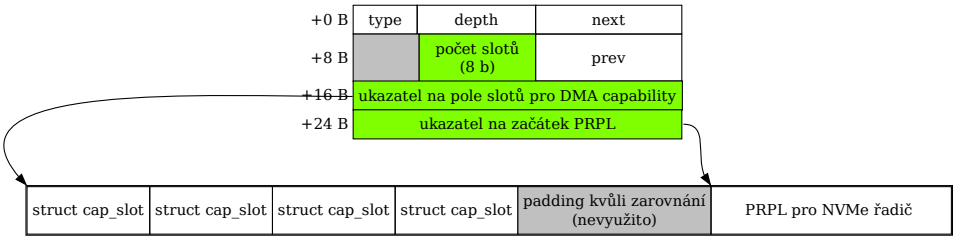
Třída admin je užívaná pro požadavky, které jsou směřovány přímo na řadič (např. čtení stavu S.M.A.R.T.), kdežto třída urgent pro ty, které jsou směřovány na disk samotný.

3.13.7 Systémová volání NVMe slice

NVMe slice má ekvivalentní systémová volání, jako *CPU slice*, tedy: odvození celého rozsahu, rozdělení, sloučení. Stejně jako *NVMe block range* podporuje GET_IDS.

3.13.8 NVMe shadow PRPL

NVMe shadow PRPL[7] je něco podobné capability table. Jde o pole capability slotů avšak proměnné délky, do kterých jsou vložené DMA capability (vizte 3.5.4) odvozené od všech paměťových capabilit zúčastňujících se DMA operace. Vzpoměňme si, že NVMe provádí scatter-gather operace. Počet rozsahů paměti je tedy proměnný a proto je i prostor pro vytváření DMA capabilit proměnný. Důsledkem je potřeba takovéhoto zvláštního typu capability. Za polem capability slotů je vymezena část paměti, kde je konstruován samotný scatter-gather list pro NVMe řadič. Tento celý rozsah fyzické paměti je reprezentován jednou touto *NVMe shadow PRPL* capabilitou.



Obrázek 3.16 NVMe shadow PRPL capability slot a kernelový objekt

Protože i samotný scatter-gather list musí existovat po celou dobu probíhající operace, chová se tento typ capability jako DMA capabilita – tj. nelze ji revokovat. Pouze ovladač NVMe ji dokáže destruovat, a to až když je operace dokončena řadičem. Jinak by userspace mohl revokovat fyzickou paměť, ve které scatter-gather existuje, a použít ji jako *userspace memory* a tím donutit řadič zapisovat například to kernelspace paměti.

Userspace se s tímto typem capability nikdy nesetká a tak není součástí exokernelového API. Jedná se o interní bookkeeping. Nicméně userspace musí při poslání požadavku k NVMe operaci exokernelu poskytnout capabilitu *untyped memory*, od které bude *NVMe shadow PRPL* odvozen (a tedy v jím reprezentovaném rozsahu fyzické paměti konstruován).

Protože ho userspace nikdy nemůže adresovat, *NVMe shadow PRPL* nemá žádná systémová volání.

[7] Zkratka PRPL ve standardu NVMe znamená *physical region page list*

3.13.9 NVMe ring

Tato capabilita propojuje vše dohromady. Reprezentuje kernelový objekt typu *struct cap_nvme_ring*, jenž obsahuje kruhový buffer s požadavky a některá pomocná data. Systémovými voláními adresovanými právě této capabilitě může aplikace požadovat čtení či zápis z/do NVMe disku.

Tento kernelový objekt dále obsahuje dva vnitřní capability sloty; jeden, do kterého je připojen *envelope ring*, a druhý, do kterého je připojen *NVMe slice*. Skrze *envelope ring* jsou generovány zprávy o dokončení operací. Povšimněme si, že *NVMe slice* není přiřazován jednotlivým požadavkům, ale celému *NVMe ringu*. To znamená, že plánovač NVMe pracuje nad *NVMe ringy*. Aplikace posílá požadavky do *NVMe ringu* a plánovač odebírá požadavky *NVMe ringu* (a předává je řadiči), dokud tomuto *NVMe ringu* nedojde burst. Pak se dostane ke slovu jiný *NVMe ring*.

Je možné *NVMe slice* vyměňovat a tím měnit prioritu či burst *NVMe ringů*. Pokud aplikace odebere *NVMe slice* (nebo je revokován), jsou všechny požadavky, které jsou odeslány do exokernelu, ale ještě ne do řadiče, automaticky zrušeny. Toto je důležitý bezpečnostní prvek, protože tomu tak nebylo, mohlo by opět dojít k *denial of service* útoku ze strany zlobivé aplikace. Aplikace by mohla vložit požadavek na DMA operaci, pak odebrat *NVMe slice*, čímž by se *NVMe ring* nikdy v plánovači nedostal ke slovu, a paměťové capability a capability rozsahu disku by byly zablokovány DMA systémem navždy[8].

Protože ovladač NVMe drží ukazatele na *NVMe ringy*, jejichž požadavky jsou diskem prováděny, nelze *NVMe ring* revokovat, dokud stále čeká na zpráva o dokončení požadavků.

Jak bylo řečeno, kernelový objekt *NVMe ring* obsahuje kruhový buffer požadavků. Co taková struktura jednoho požadavku (*struct nvme_request*) obsahuje? Především obsahuje informace, které jsou později překopírovány do kruhového bufferu řadiče. Dále obsahuje capability slot, do kterého je odvozena DMA capabilita rozsahu bloků. A závěrem dva capability sloty, do kterých jsou odvozeny DMA capability paměťových capabilit, z/do kterých jsou data čteny. Poslední zmíněné jsou dva, protože tak vypadá rozhraní NVMe řadiče – DMA operace se dvěma paměťovými rámci nepotřebuje scatter-gather seznam. Pokud aplikace vyžádá operaci přesouvající velké množství dat, bude druhý ze zmíněných slotů použit pro capabilitu typu *NVMe shadow PRPL*. Ta již byla vysvětlena výše.

3.13.10 Systémová volání

label	FLEK_SYSCALL_NVME_RING_ASSIGN_ENVELOPE_RING
argumenty userspacu	capability index ukazující na capabilitu typu envelope ring
návratové hodnoty	žádné

Přiřadí *envelope ring* tomuto *NVMe ringu*. To znamená, že *NVMe ring* bude generovat zprávy o dokončených I/O operacích skrze tento *envelope ring*, a tedy že zprávy budou posílány do *endpointu*, který je s tímto *envelope ringem* asociován.

Zprávy generované *NVMe ringem* mají 3 argumenty.

1. Identifikátor požadavku

   S každým požadavkem aplikace do *NVMe ringu* vkládá i identifikátor tohoto požadavku. Takto aplikace ví, ke kterému požadavku zpráva o dokončení patří.

2. Posun v kruhovém bufferu

[8] Alternativně by šlo vytvářet DMA odvozeniny až v moment poslání požadavku do řadiče, ale to by vyžadovalo komplikovanější implementaci

Toto číslo znamená, kolik pozic v kruhovém bufferu se uvolnilo v důsledku dokončení požadavků řadičem disku. Tato hodnota může být i nula, pokud diskový řadič dokončil požadavky v jiném než sekvenčním pořadí. O to větší pak bude někdy později.

3. Chybový kód od diskového řadiče

Nějaká konstanta, tak jak je definovaná ve standardu NVMe. Hodnota nula znamená úspěch.

Pokud má *envelope ring* předaný tomuto systémovému volání ve svých *envelopes* méně argumentů než 3, toto systémové volání selže.

label	FLEK_SYSCALL_NVME_RING_ASSIGN_NVME_SLICE
argumenty userspacu	capability index ukazující na capabilitu typu NVMe slice
návratové hodnoty	žádné

Přiřadí *NVMe slice* tomuto *NVMe ringu*. Tím je ringu přiřazena priorita a burst v rámci plánovače – a také třída. Přiřazená třída (vzpoměňme: admin nebo urgent) omezuje *NVMe ring* na vykonávání určitých typů požadavků.

Pokud je přiřazený *NVMe slice* někdy revokován, je *NVMe ring* automaticky odpojen z front plánovače a jeho zatím

label	FLEK_SYSCALL_NVME_RING_READ
argumenty userspacu	capability index ukazující na *NVMe block range*
	číslo počátečního bloku
	počet bloků
	identifikátor požadavku
	atributy
	seznam capabilit typu *userspace memory* s offsety a délkami
návratové hodnoty	žádné

Přečte rozsah bloků do paměti. Třída přiřazeného *NVMe slice* musí být *urgent*.

Zajímavé je, jakým způsobem aplikace vyplňuje informace of scatter-gather. Informace o rozsazích fyzické paměti se předávají skrze MRS (protože rozsahů může být velký počet) a to ve formě trojic hodnot. První hodnota je capability index ukazující na nějakou capabilitu typu *userspace memory* s oprávněním pro zápis a DMA. Druhá hodnota je offset uvnitř tohoto rozsahu fyzické paměti. Třetí hodnota je délka od tohoto offsetu, kde se má přenos dat ukončit. Pokud zbývají nějaká data v rozsahu bloků, vezme se další trojice a tak dokola.

FleK z tohoto seznamu rozsahů fyzické paměti sestaví PRPL pro řadič. Zároveň pro každou užitou capabilitu typu *userspace memory* vytvoří odvozenou DMA capabilitu a vloží ji do *NVMe shadow PRPL*. *NVMe shadow PRPL* je potřeba právě tehdy, pokud řadič potřebuje PRPL, což lze z pohledu aplikace rozhodnout. V tom případě aplikace musí za všechny ty trojice v MRS vložit ještě jednu hodnotu – capability index ukazující na capabilitu typu *untyped memory* s oprávněním pro DMA. V této *untyped memory* totiž bude zkonstruován *NVMe shadow PRPL*.

Argument s atributy říká NVMe řadiči, jaké přístupové vzory má očekávat[8, str. 99]. Lze zde specifikovat, jakou chceme latenci, zda budeme častěji číst či zapisovat, zda se jedná o jednorázové čtení, zda bude oblast brzy přepsána...

label	FLEK_SYSCALL_NVME_RING_WRITE
argumenty userspacu	capability index ukazující na *NVMe block range*
	číslo počátečního bloku
	počet bloků
	identifikátor požadavku
	atributy
	seznam capabilit typu *userspace memory* s offsety a délkami
návratové hodnoty	žádné

Zápis na disk. Funguje úplně stejně jako NVME_RING_READ, ale vyžadovaná oprávnění u *NVMe block range* je zápis a u *userspace memory* čtení.

label	FLEK_SYSCALL_NVME_RING_GET_LOG_SMART
argumenty userspacu	identifikátor požadavku
	seznam capabilit typu *userspace memory* s offsety a délkami
návratové hodnoty	žádné

Přečte S.M.A.R.T. informace disku. Jedná se o operaci podobnou čtení z disku, ovšem nespecifikujeme rozsah bloků a třída našeho *NVMe slice* musí být *admin*. Velikost přenesených dat je pevně dána jako 512 bajtů.

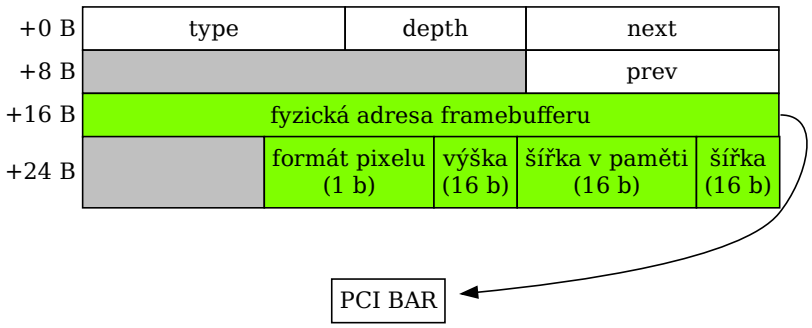
3.14 Framebuffer

Capabilita typu *framebuffer* zpřístupňuje framebuffer grafické karty. Framebuffer má určité rozlišení, formát pixelu a oblast ve fyzickém adresním prostoru. Čtení a zápisy do této oblasti jsou posílány skrze PCI/PCIe do grafické karty, která obsah zobrazuje na monitoru počítače.

Protože grafické karty jsou dnes velmi komplikovaným hardwarem, není realistické v rámci této diplomové práce napsat skutečný ovladač. FleK proto používá *UEFI GOP*[11, str. 425], což je standardizované API poskytované firmwarem UEFI. GOP umožňuje mému kernelu dotazovat podporovaná rozlišení obrazovky, nějaké z nich nastavit a získat adresu framebufferu. Změna rozlišení po raném bootu (po ukončení *boot services*) však již není možná. Podpora více monitorů zde také není, stejně tak 3D akcelerace.

FleK vybírá nejvyšší podporované rozlišení; na některém mnou testovaném hardwaru bylo podporováno i rozlišení 3840x2160. Bootovacím parametrem ze zavaděče lze FleKu říct, aby vybral konkrétní rozlišení.

3.14.1 Capability slot



Obrázek 3.17 Framebuffer capability slot

Šířka a výška jsou v pixelech. Šířka v paměti popisuje fiktivní šířku v pixelech, kterou jeden řádek skutečně v paměti zabírá. Z pohledu grafického hardwaru je žádoucí mít adresy řádků zarovnané na určitých násobcích. Formát pixelu říká, jak je jeden pixel kódován. Ačkoliv UEFI GOP podporuje mnoho formátů, FleK zatím podporuje jen B8G8R8X8 a R8G8B8X8 (kde X označuje nevyužité bity). Všechny mnou testované PC používaly jen tyto dva formáty.

Zajímavý je způsob, jakým se userspace může dostat k paměti framebufferu. Aby aplikace mohla vykreslovat obraz, musí si namapovat framebuffer někam do svého virtuálního adresního prostoru. To znamená, že z capability typu framebuffer musíme nějakým způsobem vytvořit capabilitu typu *userspace memory*. A ta pak už může být libovolně napamována.

Z pohledu návrhu API je třeba zajistit, že když je capabilita typu framebuffer (jenž jsme si namapovali) revokována, tak veškerá mapování ve stránkovacích tabulkách jsou zrušena, včetně invalidace TLB. Jinak by aplikace mohla vykreslovat na obrazovku i po ztrátě oprávnění ke framebufferu. To znamená, že výše zmíněná *userspace memory* musí být v CDT vedena jako potomek capability typu framebuffer. Vzpoměňme, že mapování *userspace memory* ve virtuálním adresním prostoru vytváří capability typu *mapping*, které jsou potomci mapované *userspace*. Důsledkem je tedy rodičovský vztah v CDT: framebuffer –¿ *userspace memory* –¿ mapping.

Revokace capability typu framebuffer způsobí rekurzivně revokaci *userspace memory* a ta revokaci mappingu. Ten, díky návrhovému vzoru connector (vizte 3.9.2), způsobí správné a úplné provedení odmapování z virtuálního adresního prostoru.

Vytváření paměťové capability z něčeho jiného než *untyped memory* může vypadat zvláštně, ale z výše uvedeného důvodu je to elegantní způsob přímého zpřístupňování memory-mapped oblastí userspacu.

Povšimněme si, že tento typ capability nereprezentuje žádný kernelový objekt. Je to tím, že UEFI GOP framebuffer může být v systému jen jeden a tak je to kernelspace singleton.

3.14.2 Systémová volání

label	FLEK_SYSCALL_FRAMEBUFFER_GET_FIELDS
argumenty userspacu	žádné
návratové hodnoty	šířka
	šířka v paměti
	výška
	formát pixelu

Dotazovací systémové volání, jímž zjistíme potřebné informace o capabilitě. Bez těchto informací nemůžeme framebuffer použít.

label	FLEK_SYSCALL_FRAMEBUFFER_DERIVE
argumenty userspacu	capability index, kam bude vytvořená odvozená capability
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu. Všechny její vlastnosti budou identické.

label	FLEK_SYSCALL_FRAMEBUFFER_GET_USMEM
argumenty userspacu	capability index, kam bude vytvořená odvozená capabilita typu *userspace memory*
návratové hodnoty	žádné

Vytvoří odvozenou paměťovou capabilitu, jak bylo vysvětleno v minulé podkapitole. Paměťová capabilita bude reprezentovat rozsah přesně o velikosti framebufferu. Přístupová práva jsou čtení i zápis, ne však oprávnění k DMA.

Aplikace si pak výslednou capabilitu může namapovat do virtuální paměti, tam psát a tak vykreslovat na obrazovku.

label	FLEK_SYSCALL_FRAMEBUFFER_KERNEL_USE
argumenty userspacu	volba možnosti
návratové hodnoty	žádné

Protože i FleK používá obrazovku pro svoje účely (výpis logu, chyb), musí se userspace s kernelem domlouvat na předávání autority. Toto systémové volání svým argumentem přepne autoritu na userspace (tj. kernel přestane vykreslovat) nebo na kernel (tj. kernel začne vypisovat). FleK, pokud to uživatel bootovacím parametrem nezakáže, vždy vypisuje log i na sériový port, tudíž se k logu ve virtualizovaném prostředí vždy dostaneme nehledě na toto nastavení.

3.15 Synchronizační primitiva

Historicky byl v operačních systémech základním synchronizačním primitivem semafor. Například v Unix System V implementováno systémovým voláním semget()[33].

Z pohledu teorie můžeme nad semafory implementovat všechna synchronizační primitiva, jako například mutex, bariéra, podmíněná proměnná. Z hlediska výkonu je toto však velmi problémové. Některá primitiva jako např. podmíněná proměnná musí být implementovány několika semafory, přičemž dochází k opakovaným systémovým voláním a opakovaným probouzením a uspáváním – v rámci jedné operace[34].

Další problém stavění primitiv nad semafory je fakt, že objekt semaforu bude kernelspace. To znamená, že každý pokus o uzamčení mutexu vyžaduje provést systémové volání, i když ve většině případů bude mutex již odemčen.

Moderní operační systémy přešly na jednodušší a univerzálnější alternativu. Na Linuxu se jí říká *futex* (zkratka *fast userspace mutex*). Princip je jednoduchý: futex je kernelspace frontou spících úloh a systémovým voláním je můžeme probudit; uspávání nad futexem je prováděno systémovým voláním, které porovná a uspí volající proces. Proces kernelu sdělí adresu a očekávanou hodnotu na ní, kernel proces uspí pouze pokud je hodnota na adrese stejná jako očekávaná[35]. Tímto lze předejít race conditionům (tzv. lost wake up problem) a implementovat složitější primitiva.

Velkou výhodou je fakt, že samotné systémové volání lze např. při zamykání mutexu ve většině situací opomenout chytrou kontrolou porovnávané hodnoty před systémovým voláním[36].

Windows používá ekvivalent futexu od verze 8[37].

Rozhodl jsem se implementovat něco podobného futexu spíše než semafor. Tím je capabilita typu conditional lock. Tato capabilita reprezentuje kernelový objekt, který zastřešuje frontu čekajících úloh.

Pokud je poslední capabilita typu conditional lock revokována a zároveň existují vlákna blokující na tomto zámku, jsou všechna tato vlákna probuzena (tj. navracejí se ze systémového volání). Jejich systémová volání skončí s chybovým kódem indikujícím, že conditional lock byl během jejich čekání zničen.

3.15.1 Capability slot

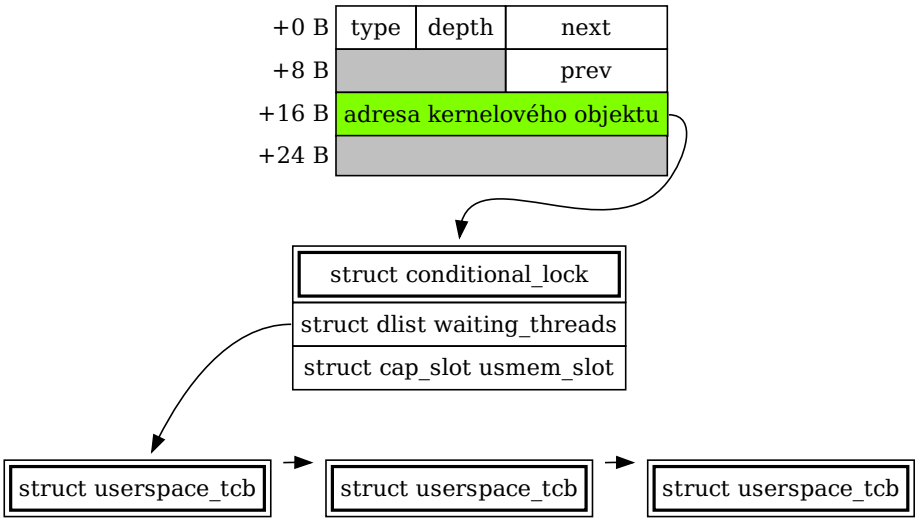
Kernelový objekt conditional locku obsahuje pouze dvě políčka: seznam userspacových vláken čekajících na tomto locku a oblast paměti k porovnávání při systémovém volání FLEK_SYSCALL_CONDITIONAL_LOCK_LOCK.

3.15.2 Paměť k porovnání

Jak již bylo zmíněno, exokernel FleK nikdy od userspacu nepřijímá adresy do virtuálního adresního prostoru. Pouze adresy do fyzického adresního prostoru, protože neumí řešit pagefaulty (řeší je až userspace). To znamená, že při operaci "porovnání a uspání"musíme poskytnout fyzickou paměť, která bude porovnávána, ačkoliv aplikace běží ve virtuálním adresním prostoru. A to je capabilita typu *userspace memory*.

Proč je potřeba vnitřní capability slot držící odvozenou capabilitu typu *userspace memory*? Potřeba jistě není, mohli bychom samozřejmě při operaci "porovnání a uspání"prostě poskytnout capability index odkazující na danou paměť v argumentu systémového volání. Ušetřili bychom navíc paměťový overhead způsobený existencí tohoto vnitřního capability slotu.

To bylo moje původní řešení, ale později se ukázalo jako vadné. Userspace aplikace musí nějakým způsobem vědět, který úsek fyzické paměti používat s jakým conditional lockem. Toto se může jevit jako triviální překážka (pokud musíme přiřadit paměť, tak už stejně musíme nějak vědět jakou), ale je to problém u aplikací, jejichž virtuální paměť je spravována externě (např. exoserverem).



Obrázek 3.18 Capability slot a kernelový objekt conditional locku

Pokud exoserver spravuje virtuální paměť aplikaci, tak aplikace ví jen o úsecích virtuální paměti, ne o tom, která fyzická paměť je jejich backingem. To by znamenalo, že pokud by aplikace chtěla používat například implementaci mutexů, musely by požadavky o zamykání a odemykání mutexu jít skrze IPC do exoserveru. Jelikož jen ten ví, která fyzická paměť je za kterou virtuální adresou. A až tento exoserver, na popud IPC požadavku o zamknutí mutexu, by provedl příslušná systémová volání nad conditional lock capabilitou.

Toto je jednak zbytečně pomalé, ale také to jde proti cílům exokernelu. Chceme, aby aplikace mohla používat synchronizační primitiva přímo komunikací s exokernelem – a to i tehdy, pokud je její virtuální paměť spravována exoserverem. Finálním řešením je právě toto přiřazení paměti k objektu condition locku. Pokud aplikace chce vytvořit mutex, požádá exoserver. Ten vytvoří capabilitu typu conditional lock, přiřadí příslušnou fyzickou paměť a pak předá zcela připravenou capabilitu typu condition lock aplikaci. Aplikace potom může provádět operaci ”porovnání a uspání”rovnou systémovým voláním do exokernelu, aniž by musela nějakým způsobem vědět, s jakou fyzickou pamětí se porovnání provádí.

3.15.3 Systémová volání

label	FLEK_SYSCALL_UTMEM2CL
adresovaná capabilita	netypovaná paměť
argumenty userspacu	capability index prvního vytvořeného conditional locku
	počet vytvořených conditional locků
návratové hodnoty	žádné

Vytvoří jeden či více kernelových objektů typu conditional lock v daném rozsahu fyzické paměti.

Selže, pokud capabilita netypované paměti velikostně nestačí. Conditional locky jsou vytvářeny postupně. Pokud už je capability index pro nějaký conditional lock zabraný, systémové volání se ukončí s chybou. Dosud vytvořené conditional locky ale zůstanou platné a použitelné.

label	FLEK_SYSCALL_CONDITIONAL_LOCK_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu reprezentující stejný conditional lock.

label	FLEK_SYSCALL_CONDITIONAL_LOCK_ASSIGN_USMEM
argumenty userspacu	capability index nějaké capability typu userspace memory
	offset od počátku rozsahu této paměti
návratové hodnoty	žádné

Přiřadí paměť k tomuto conditiona locku. Capability paměti musí mít access bit pro čtení, jinak toto volání selže. Pokud by FleK toto nekontroloval, mohl by userspace takto (velmi neefektivně) číst paměť, ke které nemá právo ke čtení.

Argument s offsetem říká, kde uvnitř přiřazené paměti má docházet k porovnávání. Tento argument by vlastně nebyl potřeba; stačilo by od capability paměti vytvořit odvozenou capabilitu paměti začínající na tom offsetu a tu přiřadit conditional locku. Tato capabilita by ale způsobovala zcela zbytečný paměťový overhead. Kernelový objekt conditional locku v sobě tak či tak musí obsahovat capability slot popisující přiřazenou paměť (aby správně fungovaly revokace), a tak je vlastně tato hypotetická odvozená capabilita vytvořena interně tímto voláním rovnou.

Pokud byla původní capabilita paměti revokována, je pochopitelně zrušeno i přiřazení ke conditional locku. Nicméně vlákna již blokující na tomto conditional locku probuzena nejsou; potřeba číst paměť existuje pouze při operaci podmíněného uspání. Stejně tak lze změnit přiřazení prostě opakovaným voláním tohoto systémového volání. Tato vlastnost je důležitá: pokud je paměť používaná conditional lockem spravovaná nějakým způsobem externě z pohledu aplikace (např. exoserverem), tak ten může potřebovat měnit backing paměť (např. swapování) a nechceme, aby toto probouzelo již spící vlákna.

label	FLEK_SYSCALL_CONDITIONAL_LOCK_LOCK
argumenty userspacu	očekávaná 64-bitová hodnota v přiřazené paměti
návratové hodnoty	žádné

Toto systémové volání provádí samotnou operaci podmíněného uspání. Jedná se o ekvivalent Linuxového FUTEX_WAIT[35].

Z pohledu userspacu se jedná o atomickou operaci, která porovná argument systémového volání s obsahem přiřazené paměti; pokud se rovnají, vlákno je uspáno. Pokud se nerovnají, systémové volání vrátí chybový kód toto indikující. Pokud je toto volání provedeno, když ještě nebyla přiřazena paměť, je vrácen chybový kód (protože operace nedává smysl).

label	FLEK_SYSCALL_CONDITIONAL_LOCK_WAKE
argumenty userspacu	počet blokujících vláken k probuzení
návratové hodnoty	počet probuzených vláken

Probudí vlákna čekající na tomto conditional locku, a to v počtu až do specifikovaného argumentem. Jedná se o ekvivalent Linuxového FUTEX_WAKE[35].

Protože jich může čekat méně a tato informace může být užitečná, je skutečný počet uspaných vláken vrácen. Může se stát, že vlákno se uspalo na conditional locku a během spánku mu byla odpřiřazena nějaká povinná capabilita, jako například kořenová stránkovací tabulka nebo CPU slice. Takové vlákno pochopitelně nemůže být probuzeno. Pokud se ho tímto voláním pokusíme

probudit, nebude probuzeno, ale stále bude z fronty conditional locku odstraněno a v počtu probuzených vláken započítano.

Implikací výše zmíněného je, že např. odstranění CPU slicu nezpůsobí odebrání vlákna z fronty čekání v conditional locku. Toto umožňuje exoserverům na pozadí libovolně prohazovat tyto vláknům-přiřazené capability bez rozbíjení jejich čekání v conditional locku. Očekávaná sekvence systémových volání ze strany exoserveru je odebrání CPU slicu, nastavení nového a aktivování vlákna. To bude fungovat správně, ať už na začátku vlákno běží nebo čeká v conditional locku.

3.15.4 Implementace synchronizačních primitiv nad conditional lockem

Pro zjednodušení práce s *conditional lockem* jsem implementoval C knihovnu libsynchro. Obsahuje efektivní implementaci mutexu, podmíněné proměnné a read-write mutexu.

Tuto knihovnu používá můj demonstrační exoserver PAL.

3.16 Čas

Capabilita typu *system time* umožňuje získání a nastavení aktuálního času a data.

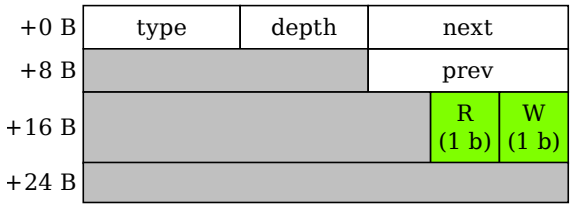
Kernel při bootu přečte aktuální datum a čas z *real-time clock* (RTC) na základní desce a tyto přepočítá na počet vteřin od *Unix epoch* (1.1.1970). Od tohoto momentu dál počítá uplynulý čas počítadlem inkrementovaným obsluhou přerušení časovače v procesoru (APIT) v mikrosekundách.

Capability používá bity oprávnění, pro čtení a zápis času. Typicky chceme programu svěřit jen oprávnění ke čtení času a oprávnění k zápisu ponechat nějaké privilegované komponentě.

Obyčejná aplikace nic praktického nezíská použitím tohoto exokernelového API oproti použití API exoserveru, existuje spíše pro exoservery.

Pokud FleK nedetekuje RTC nebo je mu to bootovacím parametrem zakázáno, tak předpokládá, že čas bootu je Unix epoch.

3.16.1 Capability slot



Obrázek 3.19 System time capability slot

Pole R je bitový příznak označující oprávnění ke čtení, W k zápisu. Šedá pole jsou prostor k dispozici capabilitě, ale zde nevyužitý.

3.16.2 Systémová volání

label	FLEK_SYSCALL_SYSTEM_TIME_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
	požadované bity oprávnění v odvozené capabilitě
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu.

Vyžaduje alespoň stejná oprávnění, jaká jsou požadována pro odvozenou capabilitu.

label	FLEK_SYSCALL_SYSTEM_TIME_GET_TIMES
argumenty userspacu	žádné
návratové hodnoty	moment bootu v sekundách od Unix epoch
	počet mikrosekund od bootu

Navrátí aktuální čas. Pokud chce userspace zjistit datum, musí si ho zpětně vypočítat z Unix time.

Vyžaduje oprávnění ke čtení.

label	FLEK_SYSCALL_SYSTEM_TIME_SET_CURR_TIME
argumenty userspacu	nový aktuální čas v sekundách od Unix epoch
návratové hodnoty	žádné

Nastaví aktuální datum a čas do RTC. Dále se zpětně přepočítá čas bootu oproti změřenému času od bootu, aby GET_TIMES vracel správnou hodnotu.

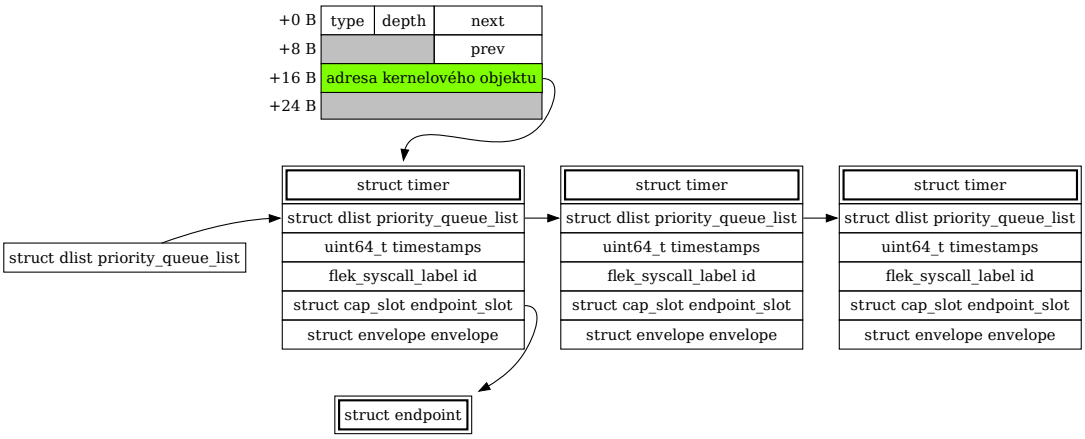
Vyžaduje oprávnění pro zápis.

3.17 Časovač

Časovač má jednoduchý účel: poslat zprávu po uplynutí nastaveného času.

Typicky toto budeme používat pro implementaci timeoutů. Vlákno nastaví časovač a provede synchronní přijetí IPC na endpointu. To vlákno uspí a teprve zpráva od časovače ho probudí.

3.17.1 Capability slot



Obrázek 3.20 Capability slot a kernelový objekt časovače

Kernelový objekt časovače obsahuje capability slot, jenž bude obsahovat odvozenou capabilitu endpointu. Do tohoto endpointu bude odeslána zpráva po uplynutí nastaveného času.

Zpráva jako taková je uložena v envelope (vizte 3.11), který je také součástí tohoto kernelového objektu. Jeden envelope stačí, protože časovač vždy čeká pouze na jeden čas. Není tedy třeba používat plnohodnotný envelope ring.

Všechny kernelové objekty časovačů, které jsou nastaveny, jsou registrované ve frontě čekajících časovačů. V této frontě jsou seřazeny podle políčka *timestamps*, které obsahuje počet *jiffies* od bootu, specifikujíc čas své aktivace. Při každém ticku hardwarového časovače (APIT[38, str. 10-16], nastaven FleKem na 1 kHz) je porovnáno políčko *timestamps* časovače na vrcholu fronty s počítadlem jiffies. Pokud je políčko vyšší, ještě nenastal čas aktivace. Protože jsou objekty seřazeny, znamená to, že žádný další časovač také ještě nemusíme řešit.

V moment aktivace je časovač z fronty odstraněn a zpráva poslána.

Co se týče implementace, výše popsaná činnost je vykonávána speciálním kernelovým vláknem. Při každém přerušení od hardwarového časovače se zkontroluje, zda registrovaný časovač na vrcholu fronty je už potřeba zpracovat. Pokud ano, je toto vlákno probuzeno, a to zpracuje jeden či více registrovaných časovačů (tj. pošle jejich envelopes).

3.17.2 Systémová volání

label	FLEK_SYSCALL_TIMER_DERIVE
argumenty userspacu	capability index, kam bude vytvořena odvozená capabilita
návratové hodnoty	žádné

Vytvoří odvozenou capabilitu, která bude odkazovat na stejný kernelový objekt.

label	FLEK_SYSCALL_TIMER_CONFIGURE
argumenty userspacu	capability index endpointu
	počet mikrosekund k počkání
	label zprávy
návratové hodnoty	žádné

Nastaví časovač. Časovači je přiřazen specifikovaný endpoint, do kterého bude časovačem poslána zpráva. Capabilita endpointu musí mít oprávnění pro odesílání, jinak toto systémové volání selže. Druhý argument říká kernelu, za jak dlouhou dobu se má zpráva poslat. Třetí argument je label zprávy; to umožňuje aplikaci rozeznat, která zpráva patřila kterému časovači, např. pokud jich používá několik.

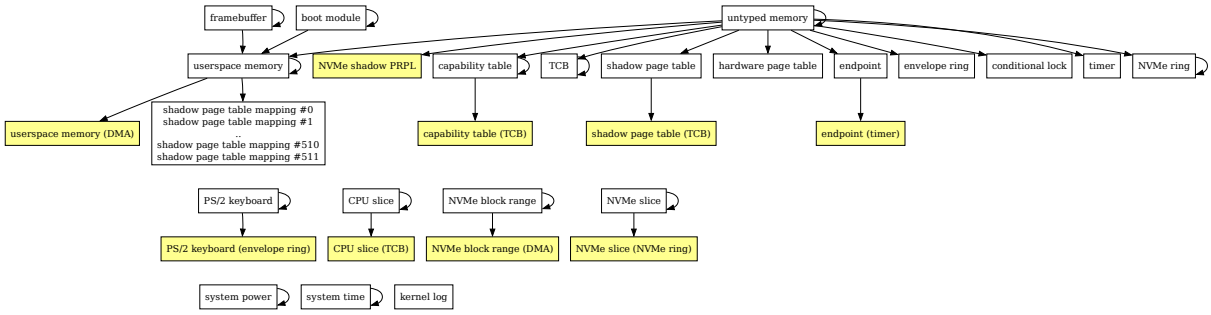
Pokud je endpoint revokován až po té, co je časovač nastaven, je časovač zrušen (jakoby byl zavolán TIMER_CANCEL, vizte dále).

label	FLEK_SYSCALL_TIMER_CANCEL
argumenty userspacu	žádné
návratové hodnoty	žádné

Zruší nastavení časovače, čímž ho vrátí do počátečního stavu. Vždy skončí úspěšně.

Kernelový objekt časovače je odebrán z fronty.

3.18 Přehled



Obrázek 3.21 Typy capabilit exokernelu FleK

Každý uzel v grafu je typ capability, šipečky označuje možnost odvozování. Takže například vidíme, že když máme capabilitu typu *untyped memory*, můžeme z ní odvodit capabilitu typu *TCB*.

Žluté uzly jsou capability, které nejsou adresovatelné userspacem. Je to tím, že tyto typy capabilit existují pouze v kernelových objektech a ovladačích zařízení. Nikdy neexistují v capability tables, tudíž nejsou součástí capability prostoru aplikace a tudíž žádný capability index nemůže odkazovat na capabilitu ilustrovanou žlutým uzlem.

Uzly, které mají text v závorce, jsou prvky návrhového vzoru connector, který byl vysvětlen v kapitole 3.9.2. Text v závorce znamená, že tato capabilita existuje pouze uvnitř capabilitu slotu, který existuje uvnitř kernelového objektu patřícího ke capabilitě typu jména v závorce. Například capabilita typu "CPU slice (TCB)"je capabilita odvozená od *CPU slice* a existující uvnitř capability slotu v kernelovém objektu capability *TCB*. Pokud je v závorce DMA, jedná se o DMA capabilitu (vizte kapitola 3.5.4).

Uzel s popiskem "shadow page table mapping"by měl být ve skutečnosti 512 uzlů v ilustraci. Účel tohoto rozdělení je popsán v kapitole 3.10.8. Pro přehlednost ilustrace jsou reprezentovány jen jedním uzlem.

Všimněme si, že některé typy capabilitu nejsou odvozovány od jiného typu capability. Například *untyped memory* a *framebuffer*. Odkud je tedy aplikace může získat? Capability těchto typů jsou vytvořeny kernelem při bootu a předány prvnímu spuštěnému vláknu. Detaily vysvětluje následující podkapitola.

3.19 Spuštění prvního programu

Kernel běžící bez userspacu je k ničemu. Tradiční kernely jednoduše spustí určený userspacový program, například na GNU/Linux to bývá /sbin/init.

Jak ale může exokernel spustit userspace, jestliže exokernel neobsahuje žádné ovladače souborových systémů? FleK bootuje ze zavaděče podporujícího protokol Multiboot 2[39], konkrétně GRUB 2. Součástí tohoto protokolu je i předání některých načtených souborů, které je zavaděč schopen přečíst (jinak by ani nenašel kernel).

FleK tedy obdrží první spustitelný soubor stejným způsobem, jako Linux obdrží *initrd*. Tento soubor musí být spustitelný soubor ve formátu ELF. Po dokončení své bootovací sekvence ho FleK rozparsuje, zkonstruuje pro něj virtuální adresní prostor, zkopíruje do něj jeho segmenty a spustí ho jako userspace vlákno.

Jelikož však tento první program běží pod exokernelem, stále nemá k dispozici žádné abstrakce operačního systému. Musí si je poskytnout sám, ale k tomu potřebuje základní zdroje jako fyzická

paměť. Kde je ale získá?

První program obdrží veškeré capability ke všemu hardware, který byl FleKem detekován, do svého capability prostoru. Dále obdrží capability k "sobě", tedy například TCB reprezentující právě toto vlákno a *shadow page table* nejvyšší úrovně jeho virtuálního adresního prostoru. Díky tomu si může první program manipulovat se svým stavem a například dále rozvíjet svůj virtuální adresní prostor. FleK na vrchol jeho zásobníku vyplní datovou strukturu s různými informacemi o předaných capabilitách. Protože podle SysV ABI[23] musí být zásobník při vstupu do funkce vhodně zarovnán, FleK musí do procesorového registru pro první argument ještě předat ukazatel na počátek této struktury.

Od prvního programu se očekává, že poskytne vhodné prostředí a spustí zbytek operačního systému. Zbytku systému pak deleguje podmnožiny capabilit, které mu předal FleK.

Samotný protokol předávání capabilit a datové struktury na zásobníku nevyužívá žádného speciálního exokernelového API. Toto je důležité pro zachování composability. Jestliže protokol mezi exokernelem a prvním programem nevyužívá speciálních systémových volání, pak může jeden program "simulovat"tento protokol mezi sebou a druhým programem. Lze si představit situaci, kde exoserver spuštěný exokernelem vytvoří podmnožinu předaných capabilit a následně je stejným způsobem předá dalšímu exoserveru. Druhý exoserver nemusí vědět, jestli byl spuštěn exokernelem nebo jiným exoserverem, a bude fungovat stejně. To umožní provozování zdánlivě několika operačních systémů pod jedním exokernelem.

Kapitola 4
Userspace

4.1 Ukázka použití API FleKu

Následující kusy kódu demonstrují, jakým způsobem userspacová aplikace vlastně používá API exokernelu FleK. Aby aplikace nemusely psát svoje vlastní implementace provádění systémových volání v assembly, napsal jsem knihovnu *libsyscall*, která poskytuje C funkce pro celé exokernelové API. Tyto funkce jsou užité v následujících úsecích kódu.

Ukázka kódu má za úkol provést čtení z NVMe disku. Programu jsou dána jen práva k nějaké fyzické paměť, práva k NVMe blokům a právo k bandwidthu NVMe disku.

Výpis kódu 4.1 Rozdělování untyped memory

```
// we need to obtain the rights to resources from somewhere
// (probably another app)
flek_cap given_untyped_cap = give_me_untyped_memory();
flek_cap nvme_block_range_cap = give_me_block_range();
flek_cap nvme_slice_cap = give_me_bandwidth();

// lets not touch the original memory cap given to us
// so we can revoke everything at once when we're done
// lets create a copy of it that we will later split
flek_cap untyped_cap = new_cap();
syscall_utmem_derive(
    given_untyped_cap, untyped_cap, FLEK_UTMEM_ACCESS_PINNABLE);

// lets determine how much memory we received
uint64_t untyped_base; // beginning of the physical memory
uint64_t untyped_length; // length in bytes
syscall_utmem_get_fields(
    &untyped_base, &untyped_length, NULL, untyped_cap);

// lets split the memory that was given to us
// megabyte for each range
flek_cap untyped_for_endpoint_cap = new_cap();
flek_cap untyped_for_envelope_ring_cap = new_cap();
flek_cap untyped_for_nvme_ring_cap = new_cap();
flek_cap untyped_for_buffer = new_cap();
uint64_t megabyte = 1024 * 1024;
syscall_utmem_split(
    untyped_cap, untyped_for_endpoint_cap,
    untyped_length -= megabyte,
```

```
    FLEK_UTMEM_ACCESS_PINNABLE, 0);
syscall_utmem_split(
    untyped_cap, untyped_for_envelope_ring_cap,
    untyped_length -= megabyte,
    FLEK_UTMEM_ACCESS_PINNABLE, 0);
syscall_utmem_split(
    untyped_cap, untyped_for_nvme_ring_cap,
    untyped_length -= megabyte,
    FLEK_UTMEM_ACCESS_PINNABLE, 0);
syscall_utmem_split(
    untyped_cap, untyped_for_buffer,
    untyped_length -= megabyte,
    FLEK_UTMEM_ACCESS_PINNABLE, FLEK_UTMEM_ACCESS_PINNABLE);
```

Pro zjednodušení není prováděna žádná obsluha chybových kódu navrácených kernelem, pokud nastanou. Každá funkce v ukázce vrací chybový kód. Dále předpokládáme, že capability programu předané jsou všechny správné.

Funkce *new_cap()* je fiktivní funkcí pro zjednodušení. Ve skutečné aplikaci bychom si museli nějak spravovat capability prostor, tj. umět alokovat ještě nepoužívané capability indexy, což je zde reprezentováno touto funkcí.

Kód dále zjednodušuje rozdělování paměti pro kernelové objekty. Megabajt pro každý z nich je plýtvání, ale zde není prostor pro řešení velikostí objektů a jejich zarovnání[1].

Následující ukázka navazuje na předchozí.

Výpis kódu 4.2 Vytváření capabilit reprezentující kernelové objekty

```
// lets create kernel objects and buffer for the contents
// in the previously split memory
flek_cap endpoint_cap = new_cap();
flek_cap envelope_ring_cap = new_cap();
flek_cap nvme_ring_cap = new_cap();
flek_cap buffer_cap = new_cap();
syscall_utmem2ep(untyped_for_endpoint_cap, endpoint_cap, 1);
syscall_utmem2er(
    untyped_for_envelope_ring_cap, envelope_ring_cap,
    1, 16, FLEK_NVME_RING_ENVELOPE_ARG_COUNT);
syscall_utmem2nr(untyped_for_nvme_ring_cap, nvme_ring_cap, 1, 16);
syscall_utmem2usmem(
    untyped_for_buffer, buffer_cap,
    FLEK_USMEM_ACCESS_R | FLEK_USMEM_ACCESS_W |
    FLEK_USMEM_ACCESS_PINNABLE);

// lets connect the capabilities together
syscall_envelope_ring_assign_endpoint(envelope_ring_cap, endpoint_cap);
syscall_nvme_ring_assign_envelope_ring(nvme_ring_cap, envelope_ring_cap);
syscall_nvme_ring_assign_nvme_slice(nvme_ring_cap, nvme_slice_cap);
```

Zde jsme vytvořili *envelope ring* ve fyzické paměti, která nám byla dána, pro 16 envelopes (prostor pro obsah zpráv) s 3 argumenty v každé envelope. A dále jsme vytvořili *NVMe ring* pro až 16 souběžných NVMe operací.

Následující ukázka navazuje na předchozí.

Výpis kódu 4.3 Čtení z disku (DMA operace)

```
// some idenfitier of the operation for us, when it finishes
flek_nvme_ring_request_id_t id = 777;
```

[1] K tomu slouží konstanty v hlavičkovém souboru FleKu exportovaném userspacu, *flek/syscall.h*.

```
// now this is the whole point of an exokernel:
// kernel mirroring the interface of hardware
flek_nvme_operation_attr_t attr =
    FLEK_NVME_OPERATION_ATTR_ACCESS_FREQUENCY_ONE_TIME |
    FLEK_NVME_OPERATION_ATTR_INCOMPRESSIBLE;
// we are telling the disk controller, that we dont
// intend to read these blocks anytime soon again and that theyre
// incompressible, so it doesnt waste time compressing them
// during internal data transfer

// MRS contains a scatter gather list of userspace memory capabilities
// and offsets and lengths of the range to be read/written within them
uint64_t *mrs = give_me_MRS();
mrs[0] = buffer_cap;
mrs[1] = 0; // offset within buffer
mrs[2] = 512; // assume the NVMe disk's block is 512 bytes

// we will read 1 block, the 17th block
syscall_nvme_ring_read(
    nvme_ring_cap, nvme_block_range_cap,
    17, 1, id, attr, 3);
// this system call ^ reads MRS for additional arguments

// wait for the result!
// this is synchronous so it will block us until the disk is done
// but if the disk was fast and it already finished, the message
// will still wait for us
flek_syscall_label label;
unsigned int arg_count;
flek_syscall_badge badge;
uint64_t arg1;
uint64_t arg2;
uint64_t arg3;
uint64_t arg4;
syscall_endpoint_receive_sync(
    endpoint_cap, &label, &arg_count, &badge,
    &arg1, &arg2, &arg3, &arg4);

// now the buffer memory contains contents of the disk block!
// we could map it to virtual memory or something...
// arg1 contains the id of 777 we supplied
// arg3 contains NVMe error code given directly by the hardware

// lets revoke all the capabilities we created so far
syscall_any_revoke(given_untyped_cap)
// and everything is in the original state now
// as if we didnt do anything
```

4.2 libOS a exoservery

Cílem exokernelu je nenutit aplikace do používání abstrakcí operačního systému, které se jim nehodí. Na druhou stranu by bylo těžce nepraktické, kdyby každý program musel používat výše ilustrované, velmi komplikované, exokernelové API pro běžné úkony.

Většina programů bude chtít mít abstrakce operačního systému již poskytnuté. Kde se ale budou tyto abstrakce nacházet? V kernelu to být nemůže.

Nic nám nebrání použít řešení operačních systémů na bázi mikrokernelu a implementovat abstrakce zcela zapouzdřeně v userspacových serverech obsluhujících požadavky přes IPC. Tím bychom ale zahodili výhody exokernelu: [*exokernel architecture attempts to avoid shared servers (especially trusted shared servers), since they often limit extensibility. For example, it is difficult to change the buffer management policy of a shared file server. In many ways, servers can be viewed as fixed kernel subsystems that run in user-space*][1]. Určitě nechceme takový typ serveru, který zapouzdří prostředky jemu svěřené, protože pak by aplikace nemohla používat exokernelové API podle end-to-end principu.

Řešení specifické pro operační systém na bázi exokernelu je tzv. *libOS* (library operating system). Protože aplikace získavají capability nakládat s různým hardwarem s dostatečnou granularitou, je možné implementovat abstrakce operačního systému jednoduše jako knihovnu přilinkovanou k aplikaci. Výhodou tohoto řešení je nesmírná flexibilita: různé aplikace si mohou vybírat implementaci abstrakcí podle své chuti, či je zcela opomenout. Problematické je sdílení stavu s jinými aplikacemi. Při použití sdílené paměti lze těžko vynutit, aby jiná aplikace zapisovala do datových struktur jen správně a ne špatně. Stránkování má granularitu pouze 4096 bajtů.

Sdílený stav tedy musíme nějak chránit, buď použitím nepřímých zápisů (vizte 3.5.5) či použitím serveru: [*How can system state be shared? Trusted (or at least highly accountable) servers can be used to manage shared, fault-isolated caches of system objects such as file-buffers. This methodology is has been explored in the context of microkernels; many of the lessons learned are directly applicable to exokernels.*][10].

Praktickým řešením je tedy něco mezi. Sdílený stav (např. implementace page cache) ve formě serveru, ale takového, aby tento server na vyžádání předal syrové capability k relevantním zdrojům aplikaci. Aplikace si pak nad těmito zdroji může implementovat vlastní abstrakce dle libosti, aniž by to ohrozilo sdílený stav či bezpečnost systému, protože capability systém je granulární a revokovatelný. A při používání těchto svěřených zdrojů nekomunikuje s tím serverem po IPC, ale napřímo s exokernelem pomocí exokernelového API. To je klíčový rozdíl proti mikrokernelovým serverům. Takovému druhu serveru říkáme exoserver[40].

4.3 PAL

Demonstrační implementaci userspacu jsem pojmenoval PAL – *Primitive abstraction layer*. Jak název napovídá, je jednoduchá a jejím cílem je ukázat, jak se API FleKu používá. Drží sdílený stav mezi aplikacemi (např. page cache, mountovací tabulky, video framebuffer), jenž musí být před nimi chráněný, a jedná se tedy o exoserver a ne libOS.

PAL je první vlákno spuštěné FleKem a proto jsou mu předány capability ke všemu detekovanému hardwaru.

PAL implementuje VFS vrstvu a v ní read-only ovladač souborového systému ext2. Dále implementuje správu virtuální paměti pro jiné aplikace a pagecache, která je s tím úzce spojená. Umí spouštět spustitelné soubory ve formátu ELF. Jsou podporovány jen nejzákladnější funkcionality, jež jsou potřeba pro interakci s uživatelem, čtení souborů, fork() a exec() s obsluhou copy-on-write a podobně.

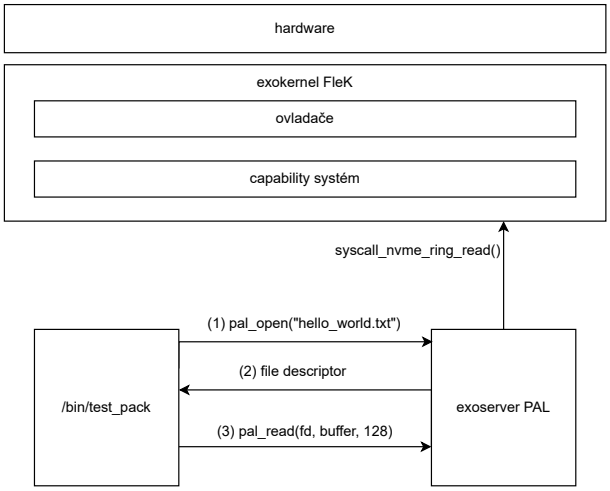
Hlavním prvkem je jeho správa kernelem svěřených capabilit. Aplikace si mohou od PALu vyžádat capability a pak mohou využívat těchto prostředků napřímo komunikací pouze s exokernelem – bez PALu jako prostředníka. Tím je zachován exokernelový *end-to-end* princip. Přesto však PAL má možnost si tyto prostředky vzít kdykoliv zpět od aplikace pomocí revokačního mechanismu.

4.3.1 PALcall

Tímto termínem jsem označil mechanismus, kterým aplikace žádají PAL o obsluhu. Jednotlivé PALcally budou přibližně na stejné úrovni abstrakce, jako systémová volání do kernelu u tradičních operačních systémů. Je to tím, že teprve až PAL implementuje klasické abstrakce operačního systému, jako jsou soubory a správa procesů.

Tato volání jsou implementována jako IPC skrze capabilitu typu *endpoint*. PAL vytvoří nové vlákno pro každý proces pod jeho správou (tj. vlákno, ve kterém aplikace běží) a k němu ještě druhé, interní, vlákno (tzv. watcher thread), jehož úkolem je jen v nekonečné smyčce čekat na požadavky od aplikace. Watcher vlákna kolektivně tvoří PAL a existují v jednom virtuálním adresním prostoru, spolu s některými dalšími vlákny. Exoserver PAL je tedy multithreaded programem.

Aplikace, která nestojí o možnost přímé manipulace s hardwarem, může se nechat zcela obsluhovat PALem:



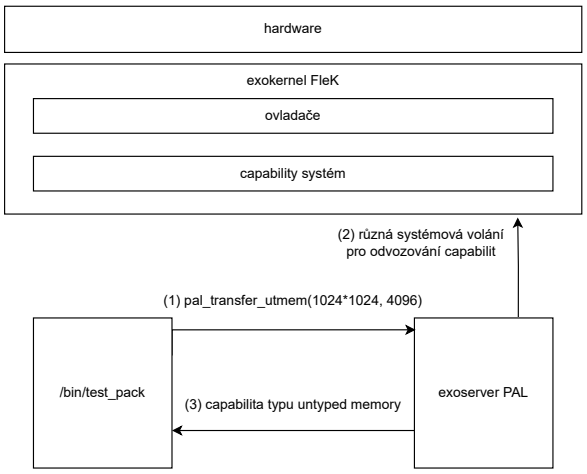
Obrázek 4.1 Obsluha PALcallu

To je na první pohled situace identická s operačními systémy na bázi mikrokernelu. Aplikace posílá IPC zprávy a server je obsluhuje. Žádná získaná flexibilita pro aplikace zde není.

Protože PAL je ale exoserver, tak má eso v rukávu...

4.3.2 Předávání capabilit

Aplikace si může od PALu vyžádat capabilitu k úseku fyzické paměti o nějaké velikosti (a zarovnání) pomocí PALcallu přes IPC.



Obrázek 4.2 Darování paměti PALem aplikaci

Pomocí API FleKu si dále může aplikace ve fyzické paměti, která ji náleží, vytvářet mnoho různých druhů capabilit: nová vlákna, stránkovací tabulky, endpointy, téměř cokoliv.

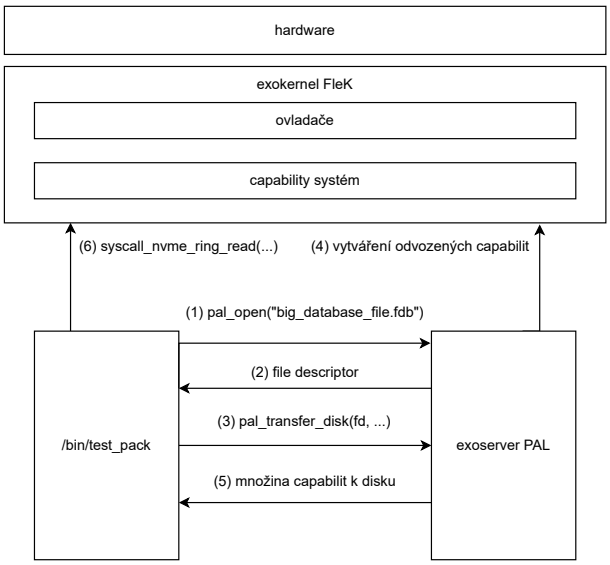
Podobným způsobem by si mohla aplikace od PALu vyžádat různé typy zdrojů, například capabilitu typu *CPU slice*. Tu si pak může aplikace rozštěpit a přiřadit vláknům, které si sestrojila z výše darované *untyped memory*. V ten moment už si provozuje vlastní vlákna, jímž figuruje jako plánovač, protože může manipulovat těmito *CPU slice* capabilitami. Může si ve svěřené fyzické paměti vytvořit *endpoint* a ten nastavit jako přijímač výjimek těchto vláken, a tím pádem si například implementovat svoji vlastní pagecache naslouchání na pagefaulty.

A to vše je možné s tím, že PAL rozhoduje o tom, zda tyto požadavky o převod capabilit uspokojí nebo odmítne podle nějaké policy. Nejen to, PAL může tyto předané capability kdykoliv revokovat, pokud by je aplikace odmítala vrátit, protože jsou odvozené od těch, které PAL interně stále drží.

Řešení tedy považuji za velmi svobodné pro aplikace; zároveň však není ani trochu ztracena moc systému nad aplikacemi.

Můžeme zajít ještě mnohem dál: představme si situaci, kdy PAL obsluhuje několik aplikací. Tyto aplikace chtějí získat capability k přímému přístupu na disk pro určité soubory. Protože PAL obsahuje ovladač souborového systému, ví, které bloky souborům patří. Může pak, za použití exokernelového API FleKu, vytvořit odvozené capability pro všechny potřebné rozsahy diskových bloků a ty předat aplikacím.

Aplikace pak mohou za pomocí exokernelového API provádět diskové DMA operace – a přitom nijak nenarušovat sdílený stav v PALu, nijak rozbíjet metadata filesystemu ani lézt do bloků, které jejich souboru nenáleží. Dokonce při tom aplikace ani nemusí chápat, jak daný souborový systém vůbec funguje. PAL si jen musí poznačit, že v rozsazích bloků těchto souborů může probíhat DMA a odmítat PALcally vyžadující například zkrácení délky takového souboru (alespoň dokud aplikace ty capability nevrátí PALu, nebo je PAL nerevokuje).



Obrázek 4.3 Darování capabilit k diskovým rozsahům souboru a k diskovému bandwidth

4.3.3 Privilegované komponenty

PAL není žádným způsobem privilegovaná komponenta, na rozdíl od mikrokernelových serverů. Je to obyčejná aplikace, která jen náhodou běží jako první v systému, a tak sdědí všechny capability od kernelu. Pokud si aplikace vyžádá od PAL zdroje všeho druhu, může si klidně implementovat svůj exoserver a nebo si implementovat libOS.

Pak může vzniknout nezvyklá situace, kdy by běželo na jednom počítači několik ”operačních systémů”pod jedním kernelem. Ani to by nepředstavovalo žádný problém, capability systém by je bezpečně multiplexoval.

Kapitola 5
Empirická měření

5.1 Testovací aplikace - FleK

Jako testovací aplikaci jsem implementoval jednoduchý program, jenž manipuluje s ext2 soubory. V podstatě kombinuje operace ilustrované v předchozí kapitole.

Provádí čtení souboru do paměti; dále zero-copy kopírování obsahu souboru z jednoho do druhého souboru. Poslední testovanou operací je porovnávání obsahu souborů, a to za pomoci speciální hardwarové akcelerace NVMe či jen softwarově. Výrazem zero-copy se rozumí operace bez potřeby kopírování dat do dočasných bufferů.

Na této testovací aplikaci je demonstrováno mnoho aspektů exokernelového API.

Aplikace si spravuje fyzickou paměť a provádí si své vlastní DMA operace s diskem, čímž umožňuje skládat buffery těchto DMA operací a tím implementovat zero-copy kopírování souborů. Diskovému řadiči volitelně poradí, že bude číst sekvenčně a jednorázově, skrze atributy NVMe operací.

Při porovnávání souborů si i spravuje část svého virtuální adresního prostoru – přestože jiná část virtuálního adresního prostoru je zároveň spravována PALem (např. memory-mapped části ELF souboru této aplikace). Pokud je vybrán test se softwarovým porovnáváním souborů, použije i 1GiB stránky pro namapování bufferů, což by mělo mírně snížit čas na konstrukci virtuální adresního prostoru pro buffery.

5.2 Testovací aplikace - Linux

Soupeřem testovací aplikace nad exokernelem je testovací aplikace běžící nad GNU/Linux, provádějící stejné úkoly, avšak s omezeními plynoucími architekturou operačního systému.

Linuxový analog provádí čtení stejných souborů v ext2 pomocí systémového volání read() ve výchozím nastavení a read() za použití příznaku O_DIRECT.

Ve výchozím nastavení file descriptoru je prováděno čtení z disku do paměťových rámců spravovaných page cachí a jejich obsah je následně kopírován do paměťových rámců namapovaných do virtuální adresního prostoru aplikace. To má výpočetní overhead. Pro aplikace, které nestojí o cachování jejich souborů v page cachi (protože vlastní data umí cachovat lépe po svém, např. databáze), byl přidán příznak O_DIRECT. Ten způsobuje, že kernel nakonfiguruje DMA operace tak, že se provedou rovnou do paměťových rámců namapovaných v adresním prostoru aplikace. Page cache je zcela obejita, kromě počátečního zeptání se, zda již v ní požadovaná data neexistují.

Kopírování obsahu souborů provádí Linuxová testovací aplikace třemi způsoby. Opět pomocí obyčejného read() a write(); dále pomocí read() a write() s O_DIRECT a nakonec pomocí systémového volání sendfile(). Poslední dvě možnosti by teoreticky měly být zero-copy.

Ale na rozdíl od testovací aplikace nad FleKem, userspace tradičních kernelů nemá jak využít konkrétních detailů hardwarového rozhraní (specifikace budoucích přístupových vzorů a podobně).

Operaci porovnání obsahu souborů provádí obyčejným read() a read() s O_DIRECT. Samotné porovnání probíhá na procesoru funkcí memcmp().

Buffery jsou zarovnány na megabajt, aby nehrozila penalizace operací s O_DIRECT v důsledku nevhodného zarovnání bufferu pro DMA.

5.3 Výsledky

Cílem testu je zjištění či vyvrácení existence nějakého zlepšení v situacích, kde je žádoucí samospráva zdrojů (tj. usecasy pro které byl vytvořen Unixový O_DIRECT). Jinak by Linuxové testy těchto I/O operací zdegenerovaly čistě na kopírování z paměti pagecache.[1] FleK má též pagecache v PALu a souboj odladěnosti implementací dvou pagecachí není smyslem tohoto testu.

Obě aplikace byly zkompilovány kompilátorem clang a s -O3.

Test probíhal na skutečném hardwaru, konkrétně na laptopu Lenovo 20YG003VCK. Procesorem je AMD Ryzen 7 5700U, NVMe diskem je SK Hynix HFM512GD3HX015. Disk podporuje rozšíření pro hardwarově akcelerované porovnávání bloků. Na disku byl vytvořen oddíl se souborovým systémem ext2 (revision 0, velikost bloku 1024 B).

V souborovém systému byly vytvořeny páry identických souborů a to ve velikostech: 512 KiB, 32 MiB a 1 GiB. Nad každým párem byl proveden každý test, oběma testovacími aplikacemi.

Výsledky jsou měřeny v milisekundách, jelikož to je největší rozlišení času dostupné v mém kernelu[2]. Testy byly opakovány 10x a výsledky průměrovány[3].

5.3.1 Test čtení

Tabulka 5.1 Výsledky čtecího testu

Pár souborů	Linux read()	Linux read() + O_DIRECT	FleK bez atributů	FleK s atributy
Malé (512 KiB)	2 ms	1 ms	2 ms	2 ms
Střední (32 MiB)	51 ms	42 ms	36 ms	36 ms
Velké (1 GiB)	1 562 ms	1 254 ms	969 ms	974 ms

Z výsledků čtení je vidět očekávaný výsledek u Linuxových testů – O_DIRECT je rychlejší, protože není třeba po DMA operaci dělat ještě kopii navíc (z page cache do bufferu aplikace).

Velmi mile mě překvapil fakt, že FleK je zde výrazně rychlejší než Linux. Očekával jsem u testu bez atributů stejné časy jako u Linuxu s O_DIRECT a s atributy nižší časy. Je možné, že FleK je rychlejší než Linux s O_DIRECT díky tenčí vrstvě mezi kernelovým API a ovladačem disku.

Nemilé je ale zjištění, že poskytnutí atributů přístupových vzorů nemá vůbec žádný dopad. Zde jsem očekával hlavní zrychlení. Je možné, že levný hardware v tomto laptopu není tak sofistikovaný, aby jich vůbec nějak využíval.

[1] echo 1 > /proc/sys/vm/drop_caches
[2] FleK zatím neumí používat HPET.
[3] Odchylky byly zanedbatelné.

5.3.2 Test kopírování

Tabulka 5.2 Výsledky kopírovacího testu

Pár souborů	Linux read() + write()	Linux read() + write() + O_DIRECT	Linux sendfile()	FleK bez atributů	FleK s atributy
Malé (512 KiB)	2 ms	2 ms[4]	2 ms	2 ms	2 ms
Střední (32 MiB)	62 ms	70 ms	52 ms	50 ms	47 ms
Velké (1 GiB)	2 101 ms	2 263 ms	1 853 ms	1 585 ms	1 606 ms

sendfile() je dle očekávání nejrychlejší varianta na Linuxu. Jedná se o kombinaci čtení a zápisu celou probíhající jen v kernelu, tudíž lze jistě obejít nějaký overhead oproti dvěma samostatným operacím čtení a zápisu.

Samostatné čtení a zápis s O_DIRECT je překvapivě mírně pomalejší, než bez tohoto příznaku.

FleK je výkonější u velkých souborů než všechny formy tohoto úkonu na Linuxu, včetně zero-copy sendfile(), což považuji za dobrý výsledek.

5.3.3 Test porovnávání

Porovnávací test spočívá v rozhodnutí, zda mají dva soubory identický obsah. Smyslem testu bylo vyzkoušet funkcionalitu NVMe pro hardwarově akcelerované porovnávání obsahu paměti s obsahem na disku. Tradiční kernely ve svých abstrakcích OS neposkytují aplikacím takovou nízkoúrovňovou specifickou funkcionalitu, a tak se je mi jevilo srovnání s exokernelem jako relevantní.

Softwarové porovnávání spočívá v přečtení obou souborů a porovnání dvou bufferů na procesoru. Jako implementace memcmp() byl použit __builtin_memcmp() clangu u obou testovacích aplikací, aby testovací aplikace pod FleKem nebyla penalizována pro svoji naivní implementaci memcmp() proti vysoce optimalizované implementaci v glibc.

Tabulka 5.3 Výsledky porovnávacího testu

Pár souborů	Linux read() + memcmp()	Linux read() + O_DIRECT + memcmp()	FleK HW akcelerovaný s atributy	FleK memcmp() s atributy
Malé (512 KiB)	4 ms	3 ms	245 ms	3 ms
Střední (32 MiB)	105 ms	87 ms	15 108 ms	60 ms
Velké (1 GiB)	3 066 ms	2 489 ms	453 586 ms	1 727 ms

Bohužel hardwarově akcelerované porovnávání se u tohoto disku ukázalo jako doslova nepoužitelné. Vysvětlení takového chování nemám. Nerozumím, proč zařízení hlásí, že tuto volitelnou funkcionalitu podporuje, pokud je implementována takovýmto způsobem.

Softwarové porovnávání testovací aplikace pod FleKem je výrazně rychlejší než testovací aplikace pod Linuxem. Protože v případě softwarového porovnávání se jedná o dvě operace čtení a FleK již v testu čtení byl znatelně rychlejší, tak se zde rozdíl ještě prohloubil. Nějaký overhead v neprospěch Linuxu bude způsoben tím, že tradiční kernel bude konstruovat virtuální adresní

[4] Zde stojí za zmíňku, že tato konfigurace testu konzistentně trvala 8 milisekund, pokud nebyla vyčištěna page cache. Důvod je neznámý.

prostor aplikace pro buffer z mnoha malých stránek. Testovací aplikace pod FleKem použila pro buffer 1GiB stránku, což je velmi rychlé na nakonfigurování.

5.3.4 Závěry z testů

Absurdně neefektivní hardwarová implementace hardwarově akcelerovaného porovnávání souborů zhatila situaci, kde by zrovna exokernel mohl zazářit. Ačkoliv výsledky nenaplnily má očekávání, alespoň v některých testech si FleK vede lépe než Linux.

V retrospektu se testy na malých souborech zdají být zbytečné, protože jejich výsledky jsou stejné a to s hodnotami pár milisekund, tedy s nedostačující přesností měření. Smyslem jejich testování bylo detekování případného neúměrného konstantního overheadu v nějaké situaci. Nic takového se nakonec neukázalo, což je dobrá zpráva.

Testovací aplikace pro FleK zde mohla být v drobné nevýhodě, protože PAL musel vytvořit odvozenou capabilitu *NVMe block range*, pro každý rozsah bloků souboru. Souborový systém ext2 ale neoperuje nad extenty, pouze nad seznamem bloků. PAL tedy musí extenty vypočítat ze seznamu bloků. Kdyby se jednalo o souborový systém ext4, který už sám reprezentuje obsazená data jako rozsahy, tento overhead by odpadl.

Kapitola 6
Co zbývá?

6.1 SMP

FleK v současnosti běží jen na jednom jádře procesoru. To je v době, kdy se v PC běžně používají i 16-jádrové procesory, těžko přijatelné.

Podpora SMP je však větším problémem, než u tradičních kernelů. Operace nad capabilitami provádí procházení CDT v obou směrech, což by komplikovalo zamykání (deadlocky). V současnosti není žádné zamykání potřeba, protože kernelová vlákna mají vždy vyšší prioritu než userspace vlákna a plánovač je během obsluhy systémových volání vypnut. Synchronizace mezi kernelovými vlákny navzájem se provádí vypnutím plánovače v kritických sekcích. Synchronizace mezi kernelovými vlákny a obsluhou přerušení se provádí vypnutím přerušení v kritických sekcích. FleK tedy v kernelu nepotřebuje ani mutexy, ani spinlocky.

Alternativu fine-grained zamykání kernelspace objektů představuje málo prozkoumaná oblast tzv. *multikernelů*. Ty zamítají koncept jednoho kernelu běžícího na více procesorech a místo toho na každém procesoru běží zdánlivě samostatný kernel mající svůj vlastní stav[41].

Příkladem multikernelu je v této práci často zmiňovaný Barrelfish, jenž také používá jako svůj capability systém strukturu CDT převzatou od seL4. Na každém jádře existuje samostatný CDT graf a capability musejí být explicitně přesouvány mezi procesorovými jádry[42, str. 11].

Je otázkou zda implementovat SMP tradičním způsobem, tedy jeden globální stav, který je složitě zamykán; či cestou multikernelu.

6.2 Kombinované operace

Protože aplikace běžící pod exokernelem bude pravděpodobně vykonávat mnohem větší počet systémových volání, je větší tlak na snížení overheadu systémového volání. Některé operace často vykonávané ve smyčce by možná bylo záhodno kombinovat do jedné operace.

Můžeme si představit situaci, kdy si aplikace vyplňuje celou vlastní stránkovací tabulku. V současnosti musí provést 512 systémových volání FLEK_SYSCALL_SPT_MAP_USMEM. Tyto by šly zkombinovat do jednoho volání a těchto mnoho parametrů by mohlo být předáno přes MRS.

Dále by bylo vhodné prozkoumat možnost implementace ekvivalentu Linuxového io_uring. Ten je novým mechanismem komunikace aplikace a kernelu bez nutnosti provádění systémového volání pro každou jednotlivou požadovanou operaci[43]. Jedná se o kruhový buffer, kde roli *producera* má aplikace a vkládá do něj popisy operací pro kernel. Kernel má roli *consumera* a popisy odebírá a provádí. V druhém kruhovém bufferu jsou role otočené a kernel jím předává aplikaci informace o výsledcích provedených operací.

Nutnost takového mechanismus může ukázat či vyvrátit až praktická zkušenost.

6.3 Více zdrojů událostí

Některé operace prováděné skrze API FleKu mutují stav systému, ale negenerují se o tom zprávy. Příkladem je změna RTC času (capabilita *system time*). Jestliže jedna aplikace změní aktuální čas, jiné aplikace se o změně nedozví a byly by nuceny občas pollovat aktuální čas.

Do budoucna by bylo lepší umožnit na takové události napojit capabilitu typu *envelope ring*, aby i nezúčastněné aplikace mohly být informovány o mutacích globálního stavu.

6.4 Síť

Mým původním záměrem bylo implementovat i síťové funkcionality s tím, že testovací aplikace by demonstrovala zero-copy operace mezi diskem a síťovou kartou, jako bylo demonstrováno u Xoku[2].

Z důvodu limitovaného dostupného času jsem od tohoto musel upustit. Nicméně FleK obsahuje částečně funkční implementaci ovladače Realtek 8139, který nebyl dokončen.

Exokernelové API by se velmi podobalo NVMe capabilitám: jistě bychom měli nějaký ”slice”reprezentující zlomek bandwidth síťové karty; místo ”block range”bychom měli nějakou reprezentaci filtrů packetů. A jistě bychom měli ”ring”capabilitu, do které by se vkládaly požadavky, a envelope ring, ze kterého by přicházely výsledky.

6.5 Hibernace

Nevyřešeným problémem jsou některé změny power state. Při hibernaci je nutné uložit všechny paměťové rámce userspacu i kernelu na disk a pak počítač vypnout.

Userspace pod exokernelem FleK sice může ukládat svoji paměť na disk, ale nemůže číst paměť kernelu. A už rozhodně ne zapisovat ji při probuzení.

Exokernel má zase jiný problém: nechápe nic jako souborový systém či oddíly na disku, aby mohl paměť uložit do swap oddílu či souboru.

Tedy ani jedna komponenta sama o sobě není schopna provést hibernaci. Řešení, pokud existuje, by vyžadovalo hibernaci provedenou exokernelem, ale s nějakou asistencí userspacu, který by směroval zápisy na disk. Příslušná userspacová komponenta by musela mít capability k daným diskovým blokům. Při dalším bootu, pokud by došlo k běžnému startu a ne probuzení z hibernace (čtení paměti z disku), exokernel by musel nějakým způsobem zabránit všem userspacovým komponentám získat přístup k těmto blokům. Jinak by si nějaká aplikace mohla přečíst obsah paměti jiné aplikace z předchozího běhu, což by mohl být bezpečnostní problém. To může být komplikované na implementaci.

V případě úspěšně provedené hibernace je zde ještě problém desynchronizace capability systému a skutečného stavu počítače. Při příštím bootu mohou být rozsahy fyzické paměti nakonfigurované firmwarem jinak, či i dostupný hardware může být fyzicky trochu jiný. Aplikace ale už mají svoje capability prostory nějak nakonfigurované, stránkovací tabulky nějak sestavené. Exokernel by vždy mohl revokovat cokoliv, co mu přijde nadále neplatné. Ale userspace by si s takovou náhle narušenou konfigurací těžko poradil.

Dobré řešení mi není známé.

6.6 Hotplugging

Hotplugging je schopnost některých zařízení být připojeno za běhu počítače. Jiná zařízení mohou být konfigurována jen při spuštění počítače.

Možnost připojení nových zařízení za běhu je problematická pro exokernel. Obsluha zařízení bude potřebovat alokaci většího množství paměti pro datové struktury ovladače. U tradičního kernelu toto není problém, protože všechna fyzická paměť a správa virtuálních adresních prostorů aplikací je v kompetenci kernelu. Může tedy transparentně zahodit paměťové rámce z page cache a podobně. U exokernelu je toto problémové, protože částěčně správa fyzické paměti a plně správa virtuálních adresních prostorů je v kompetenci userspacu.

Exokernel Aegis měl tzv. *abort protocol*, kterým mohl násilně sebrat paměť userspacu. Exokernel FleK používá deterministickou alokaci paměti kernelem v reakci na systémová volání, aby předešel potřebě destruktivního a těžko obslužitelného chování jako je abort protocol. V důsledku mého rozhodnutí mít page cache v userspace není ani možnost zahodit cachovaný obsah disku.

FleK si drží určitou rezervu fyzické paměti, kterou při bootu nepředal userspacu (a zbytek fyzické paměti v systému předal). Je jen otázkou počtu zařízení připojených přes Plug and Play, USB či obdobné technologie, než rezerva dojde. Tedy toto není řešení.

Řešením, kterým bych se vydal, by byl mechanismus informování o nově připojeném zařízení. Pravděpodobně pomocí capabilit typu *endpoint* a *envelope ring*, která by byly vytvořeny při bootu a předány první aplikaci. Skrze ně by přicházely zprávy o nově připojeném zařízení a kolik fyzické paměti jeho ovladač potřebuje. Následně by existovalo API pro darování capability typu *untyped memory* exokernelu, která by reprezentovala rozsah fyzické paměti potřebné velikosti. Musí to být *untyped memory* a ne *userspace memory*, aby byly splněny invarianty, že si aplikace nebude moci namapovat tuto paměť používanou kernelspace ovladačem.

Problém by byl s revokacemi této paměti, protože si nelze představit, že by prakticky bylo možné bezpečně zdestruovat stav ovladače a paměť vrátit aplikaci, kdykoliv si tak aplikace usmyslí. Nejspíše by takové dary byly trvalé.

6.7 Transparentní kernelspace paměť

Původní exokernel Aegis zpřístupňoval některé své datové struktury ve virtuálním adresním prostoru i obyčejným aplikacím[1]. Tyto oblasti byly z pohledu userspacu namapované pouze pro čtení samozřejmě. Smyslem bylo umožnit aplikacím vědět o globálním stavu systému a vhodně reagovat.

Jakákoliv informace takto získána může být získána i obyčejným systémovým voláním, ale s vyšším výpočetním overheadem.

Podobnou techniku používají i mainstreamové operační systémy. Například Linux poskytuje *vDSO (virtual dynamic shared object)*. Jedná se o fyzickou paměť vyplňovanou kernelem, ale mapovanou do virtuálního adresního prostoru každého userspace procesu. Jejím obsahem je i spustitelný kód, ale jsou v ní kernelem vyplňovány i data jako je aktuální systémový čas. Aplikace si pak může přečíst aktuální čas přímým paměťovým čtením a není nutné provést pomalé systémové volání.

Do budoucna by bylo dobré toto implementovat i ve FleKu.

6.8 Coalescing

NVMe má funkci, která snižuje počet přerušení, jménem *interrupt coalescing*[8]. V moment, kdy by řadič normálně provedl přerušení, ho neprovede a spustí časovač, jehož periodu nastavil kernel. Přerušení se provede až po uplynutí periody – pokud mezitím nastaly další události normálně vyvolávající přerušení, tak tato přerušení se neprovádí. Ve výsledku se tedy "sesbírá"několik přerušení do jednoho na konci periody.

Cílem je minimalizovat plýtvání procesorového času přepínáním kontexu v situacích, kdy je prováděn velký počet rychlých operací. Cenou za to je vyšší latence.

Podobnou techniku bychom možná mohli zobecnit, a to na úrovni *envelope ring*. Tím by aplikace mohly volitelně snížit počet context switchů, pokud jim nevadí latence.

Kapitola 7
Závěr

Smyslem práce bylo prozkoumat možnosti návrhu operačního systému na bázi exokernelu. Při návrhu API byla prošlapána nová cesta kombinující cíle původního exokernelu Aegis s pokroky v oblasti moderních capability-based kernelů jako seL4.

Obyčejné nedůvěryhodné aplikace postavené nad exokernelem FleK mohou používat API, které blízce mimikuje hardwarová rozhraní, a přitom není bezpečnost systému narušena. Byla vymyšlena originální řešení otázek správy virtuální paměti, DMA operací, přístupu k NVMe diskům a naslouchání na události v kernelu (*envelope ring*). A to s důrazem na maximální schopnost revokací capabilit vzájemně mezi aplikacemi. Dále byla odstraněna potřeba existence *abort protocolu* původního exokernelu Aegis návrhem exokernelu s deterministickou alokací paměti, čímž byla zvýšena robustnost architektury.

Práci považuji za úspěšnou, avšak mnoho jejích částí je implementováno absolutně minimalisticky pro splnění demonstračních účelů. V projektu plánuji nadále pokračovat.

Příloha A

Výpočet overheadu správy virtuální paměti

Z tabulky 3.2.

A.1 Situace č. 1

A.1.1 seL4

Je potřeba ASID prvku. Dále 4 tabulky (1 pro každou úroveň), to jsou 4 sloty. Dále *page* slot, který je mapován. Celkový overhead je

$$ASID + 5 * 32 = 160$$

.

A.1.2 Gerberův Barrelfish

Je potřeba 4 tabulky a pro jejich propojení 3 sloty (*mapping*). Následně je potřeba vytvořit *frame* a jeho *mapping*. Celkový overhead je

$$4 * 64 + 3 * 64 + 1 * 64 + 1 * 64 = 576$$

.

A.1.3 FleK

Je potřeba 4 sloty pro hardwarové stránkovací tabulky, 4 sloty pro SPT a 1 slot pro *userspace memory*, která bude mapována. Celkový overhead je

$$4 * 32 + 4 * 8208 + 32 = 32992$$

.

A.2 Situace č. 2

Situace vychází na 8 stránek nejnižší úrovně a 4096 paměťových rámců k namapování.

A.2.1 seL4

Je potřeba ASID prvku. Dále 8 tabulky nejnižší úrovně a 3 do zbývajících úrovní (po jedné), to je 11 slotů. Dále 4096 *page* slotů, které jsou mapovány. Celkový overhead je

$$ASID + 11 * 32 + 4096 * 32 = 131424$$

.

A.2.2 Gerberův Barrelfish

Je potřeba 11 tabulek (8 + 3) a pro jejich propojení 10 slotů (*mapping*). Následně je potřeba vytvořit *frame* a 8 jeho *mapping* (pro každou stránku nejnižší úrovně). Celkový overhead je

$$11 * 64 + 10 * 64 + 1 * 64 + 8 * 64 = 1920$$

.

A.2.3 FleK

Je potřeba 11 slotu pro hardwarové stránkovací tabulky, 11 SPT slotů a 1 slot of *userspace memory*. Celkový overhead je

$$11 * 32 + 11 * 8208 + 32 = 90672$$

.

A.3 Situace č. 3

A.3.1 seL4

Stejné jako u předchozí situace. Fragmentace způsob mapování zde nic nemění.

A.3.2 Gerberův Barrelfish

Je potřeba 11 tabulek (8 + 3) a pro jejich propojení 10 slotů (*mapping*). Následně je potřeba vytvořit *frame* a jeho *mappingy*. Tentokrát bude počet *mappingů* masivní. Celkový overhead je

$$11 * 64 + 10 * 64 + 1 * 64 + 4096 * 64 = 263552$$

.

A.3.3 FleK

Stejné jako u předchozí situace. Fragmentace způsob mapování zde nic nemění.

Bibliografie

1. ENGLER, Dawson R.; KAASHOEK, M. Frans; JR., James O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. 1995. https://pdos.csail.mit.edu/6.828/2008/readings/engler95exokernel.pdf.
2. ENGLER, Dawson R. *The Exokernel Operating System Architecture*. 1998. https://dspace.mit.edu/bitstream/handle/1721.1/16713/42430053-MIT.pdf.
3. *open(2) – Linux manual page*. [B.r.]. https://man7.org/linux/man-pages/man2/open.2.html.
4. BOVET, Daniel P.; CESATI, Marco. *Understanding the Linux kernel*. Third edition. O'Reilly, 2006. ISBN 978-0-596-00565-8.
5. *sendfile(2) – Linux manual page*. [B.r.]. https://man7.org/linux/man-pages/man2/sendfile.2.html.
6. *The Open Group Base Specifications Issue 6: posix_madvise specification*. IEEE a The Open Group, 2004. https://pubs.opengroup.org/onlinepubs/007904875/functions/posix_madvise.html.
7. *madvise(2) – Linux manual page*. [B.r.]. https://man7.org/linux/man-pages/man2/madvise.2.html.
8. HUFFMAN, Amber. *Non-volatile Memory Host Controller Interface 1.0e*. Intel Corporation, 2013. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_10e_specification.pdf.
9. *nvme-cli*. [B.r.]. https://github.com/multi-stream/nvme-cli/blob/master/nvme-ioctl.c.
10. ENGLER, Dawson R.; KAASHOEK, M. Frans. *Exterminate All Operating System Abstractions*. MIT Laboratory for Computer Science, [b.r.]. https://people.eecs.berkeley.edu/~brewer/cs262b/hotos-exokernel.pdf.
11. *Unified Extensible Firmware Interface (UEFI) Specification*. UEFI Forum, Inc., 2022. https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf.
12. *Synaptics PS/2 TouchPad Interfacing Guide*. Synaptics Incorporated, 2011. http://blog.amigas.ru/wp-content/uploads/2014/03/touchpad_RevB.pdf.
13. PINCKNEY, Thomas. *Xok and physical pages*. 1998. https://pdos.csail.mit.edu/archive/exo/exo-internals/node7.html.
14. PINCKNEY, Thomas. *Xok: Kernel Buffer Cache*. 1998. https://pdos.csail.mit.edu/archive/exo/exo-internals/node11.html.

15. *The Open Group Base Specifications Issue 7, 2018 edition: mmap specification*. IEEE a The Open Group, 2018. https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html.

16. *Dokumentace BTRFS*. [B.r.]. https://btrfs.readthedocs.io/en/latest/Deduplication.html.

17. GROSVENOR, Matthew; WALKER, Adam. *seL4 Reference Manual*. 2021. https://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

18. *seL4 tutorial: Retyping*. 2020. https://docs.sel4.systems/Tutorials/untyped.html#retyping.

19. JONES, Mark P. *seL4 - capabilities in practice*. Portland State University, 2018. https://web.cecs.pdx.edu/~mpj/llp/slides/LLP08-seL4-6up.pdf.

20. CHEN, Shuo. *Evolution of File Descriptor Table in Linux Kernel*. 2021. https://github.com/chenshuo/notes/blob/master/docs/kernel/file-descriptor-table.md.

21. SINGHANIA, Akhilesh; KUZ, Ihor; NEVILL, Mark; GERBER, Simon. *Capability Management in Barrelfish*. Systems Group, Department of Computer Science, ETH Zurich, 2017. https://barrelfish.org/publications/TN-013-CapabilityManagement.pdf.

22. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices Inc., 2023. Rev. 3.4. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

23. MATZ, Michael; HUBIČKA, Jan; JAEGER, Andreas; MITCHELL, Mark. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. Advanced Micro Devices Inc., 2012. https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf.

24. DANNOWSKI, Uwe; LEVASSEUR, Joshua; SKOGLUND, Espen; UHLIG, Volkmar; STOESS, Jan. *L4 eXperimental Kernel Reference Manual*. System Architecture Group, Dept. of Computer Science, Karlsruhe Institute of Technology, 2011. https://www.l4ka.org/l4ka/l4-x2-r7.pdf.

25. JONES, M. *Inside the Linux 2.6 Completely Fair Scheduler*. 2018. https://developer.ibm.com/tutorials/l-completely-fair-scheduler/.

26. *Advanced Configuration and Power Interface (ACPI) Specification, Release 6.5*. UEFI Forum, Inc., 2022. https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf.

27. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture*. Intel Corporation, 2023. https://cdrdv2.intel.com/v1/dl/getContent/671436.

28. HEISER, Gernot; ELPHINSTONE, Kevin. *L4 Microkernels: The Lessons from 20 Years of Research and Deployment*. NICTA a UNSW, 2023. https://trustworthy.systems/publications/nicta_full_text/8988.pdf.

29. DREPPER, Ulrich. *ELF Handling For Thread-Local Storage*. Red Hat Inc., 2005. https://www.uclibc.org/docs/tls.pdf.

30. GERBER, Simon. *Authorization, Protection, and Allocation of Memory in a Large System*. ETH Zurich, 2018. https://barrelfish.org/publications/gerbesim-phd.pdf.

31. *The GNU C Library Reference Manual, for version 2.38: Using a Separate Signal Stack*. Free Software Foundation, Inc., 2023. https://www.gnu.org/software/libc/manual/html_node/Signal-Stack.html.

32. CARMACK, John. *9fans mailing list: Graphics issues*. 1995. https://marc.info/?l=9fans&m=111558698816997.

33. *The Open Group Base Specifications Issue 6: XSI Interprocess Communication*. IEEE a The Open Group, 2004. https://pubs.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_07.html#tag_02_07.
34. BIRRELL, Andrew D. *Implementing Condition Variables with Semaphores*. Microsoft Research, 2003. http://birrell.org/andrew/papers/ImplementingCVs.pdf.
35. *futex(2) — Linux manual page*. [B.r.]. https://man7.org/linux/man-pages/man2/futex.2.html.
36. DREPPER, Ulrich. *Futexes Are Tricky*. Red Hat, Inc., 2004. https://dept-info.labri.fr/~denis/Enseignement/2008-IR/Articles/01-futex.pdf.
37. *WaitOnAddress function (synchapi.h)*. Microsoft, 2021. https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitonaddress.
38. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide, Part 1*. Intel Corporation, 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.
39. OKUJI, Yoshinori K.; FORD, Bryan; BOLEYN, Erich Stefan; ISHIGURO, Kunihiro; SERBINENKO, Vladimir; KIPER, Daniel. *The Multiboot2 Specification version 2.0*. 2016. https://www.gnu.org/software/grub/manual/multiboot2/multiboot.pdf.
40. PULIDO, Hector Manuel Briceno. *Decentralizing UNIX Abstractions in the Exokernel Architecture*. Massachusetts Institute of Technology, 1997. http://dspace.mit.edu/bitstream/handle/1721.1/42781/38545045-MIT.pdf.
41. BAUMANN, Andrew; BARHAM, Paul; DAGAND, Pierre-Evariste; HARRIS, Tim; ISAACS, Rebecca; PETER, Simon; ROSCOE, Timothy; SCHÜPBACH, Adrian; SINGHANIA, Akhilesh. *The Multikernel: A new OS architecture for scalable multicore systems*. Symposium on Operating Systems Principles, 2009. https://people.inf.ethz.ch/troscoe/pubs/sosp09-barrelfish.pdf.
42. *Barrelfish Architecture Overview*. Systems Group, Department of Computer Science, ETH Zurich, 2013. https://barrelfish.org/publications/TN-000-Overview.pdf.
43. CORBET, Jonathan. *The rapid growth of io_uring*. LWN.net, 2020. https://lwn.net/Articles/810414/.

Obsah přiloženého média

- `flek` — zdrojový kód kernelu FleK
- `font` — kompilátor fontu
- `image` — generátor obrazu disku pro Qemu
- `libalgo` — zdrojový kód obecné algoritmické knihovny
- `libmonospace` — zdrojový kód knihovny pro vykreslování fontu
- `libsynchro` — zdrojový kód knihovny se synchronizačními primitivy
- `libsyscall` — zdrojový kód knihovny obalující API FleKu
- `linux` — zdrojový kód Linuxové testovací aplikace
- `pal` — zdrojový kód exoserveru PAL
- `pdf` — zdrojová forma práce ve formátu LaTeX
  - `DP_Rehak_Dorian_2023.pdf` — text práce ve formátu PDF
- `utils` — skripty pro sestavení kompilačního prostředí
- `README.txt` — návod pro zprovoznění všeho