Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Implementácia virtuálneho počítača na x86 hardvéri

Diplomová práca

Milan Plžík

Vedúci:
RNDr. Jaroslav Janáček

Bratislava, 2010

Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Implementácia virtuálneho počítača na x86 hardvéri

(Diplomová práca)

Bc. Milan Plžík

Študijný program: Informatika
Študijný odbor: 9.2.1 Informatika
Vedúci: RNDr. Jaroslav Janáček

Bratislava, 2010

Čestne prehlasujem, že som túto diplomovú prácu vypracoval(a) samostatne s použitím citovaných zdrojov.

. . . . . .

Obsah

Zoznam tabuliek

Poďakovanie

V prvom rade by som sa chcel poďakovať RNDr. Jaroslavovi Janáčkovi, za jeho pomoc pri tejto práci, hlavne pri jej finalizácii, ktorá si vyžiadala veľa času vo veľmi krátkom intervale. Ďalšie poďakovanie patrí mojim rodičom, ktorí ma počas celého štúdia podporovali a pomáhali mi so všetkým, čo bolo v ich silách; a takisto aj mojej priateľke Martinke Patašiovej, za jej trpezlivosť a podporu vždy, keď ju bolo treba.

Ďakujem:)

Abstrakt

Autor:	Bc. Milan Plžík
Názov práce:	Implementácia virtuálneho počítača na x86 hardvéri
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Školiteľ:	RNDr. Jaroslav Janáček
Rok:	2010

Predmety zaoberajúce sa operačnými systémami sú v súčasnosti povinným obsahom každého študijného odboru zameraného na informatiku. Často pokrývajú veľký a rôznorodý obsah učiva, pričom na cvičeniach sa používa softvérové riešenie, ktoré študentov prakticky oboznámi s najdôležitejšími témami. Cieľom tejto práce je vytvoriť sadu programov, ktoré umožnia efektívne a jednoducho vytvárať rôzne cvičenia, ako tiež demonštrovať niektoré časti fungovania bežných operačných systémov. Prístup, ktorý je rozoberaný v tejto práci, vytvorenie virtuálneho počítača so simulovaným zjednodušeným hardvérom. Samotné riešenie je zložené z emulačného jadra zabezpečujúceho základnú funkcionalitu a správcovských programov, ktoré simulujú samotný hardvér. Prílohou ku práci je implementácia jadra aj ukážkových správcovských programov.

Kľúčové slová: virtuálny počítač, emulácia, operačný systém, hardvér

Kapitola 1

Úvod

Zrejme na všetkých vysokých školách so študijnými odbormi príbuznými informatike sa vyučuje predmet s tematikou operačných systémov, ktorého cieľom je poskytnúť študentom prehľad o základných vlastnostiach a úlohách operačných systémov. Táto tematika je veľmi rozsiahla a poskytuje množstvo rôznorodých tém, či už sa jedná o komunikáciu s hardvérom, správu pamäte, procesorového času, operácie nad súborovým systémom, alebo iné. Podľa charakteristiky predmetu je ťažisko kladené na niektoré z nich.

S prednáškami operačných systémov sú veľmi často spojené aj cvičenia, ktoré sa zameriavajú na praktickú ukážku ťažiskových častí preberaného učiva. Väčšinou sa využívajú existujúce softvérové riešenia, ako napríklad *Nachos*[3] alebo *MINIX*[1], ktoré sú vhodné na štandardné typy úloh.

1.1 Motivácia

Vyššie popísané riešenia nepokrývajú dobre úplne celý rozsah tematiky. Ich najväčšou nevýhodou je, že sú vždy buď viazané na konkrétnu oblasť alebo zadania (napríklad niekoľko konkrétnych príkladov), alebo svojou komplexnosťou už tvoria samotný (hoci jednoduchý) operačný systém.

Nižšie sú uvedené krátke analýzy jednotlivých riešení, ktoré sa v súčasnosti používajú na výuku operačných systémov. Zoznam samozrejme nie je kom-

pletný obsahuje iba vybrané príklady.

1.1.1 Nachos

Nachos je vyvíjaný na University of California, Berkeley, kde sa taktiež používa na výuku. Existujú dve implementácie prostredia, jedna napísaná v jazyku C++, druhá v Jave . Programy, ktoré sú *userspace*, sú kompilované pre architektúru MIPS, ktorej emulátor je súčasťou Nachos-u.

Skladá sa zo štyroch fáz, z ktorých každá sa zameriava na jednu špecifickú tému:

- Vlákna
- Multiprogramovanie
- Caching a virtuálna pamäť
- Sieť a distribuované systémy

Úlohou v jednotlivých fázach je upraviť a doplniť kód Nachosu tak, aby spĺňal požadované vlastnosti.

Výhodou Nachosu je, že poskytuje pripravený balík, ktorý sa dá priamo bez ďalších úprav použiť na cvičeniach. Takisto je k dispozícii rozhranie pre automatické ohodnocovanie odovzdaných študentských prác, takže jediná potrebná aktivita je pripraviť si vlastnú sadu testov, ktorými budú skúšané odovzdané študentské práce.

Naopak značnou nevýhodou Nachos-u je jeho „statickosť“. Pripravené fázy nie sú navrhnuté tak, aby umožňovali variácie úloh. Riešenia jednotlivých úloh, ktoré boli raz vypracované, sa dajú opäť použiť v nasledujúcich rokoch.

1.1.2 MINIX

Operačný systém pre akademické účely vyvíjaný už dlhú dobu Andrewom S. Tanenbaumom (Vrije University), momentálne v tretej verzii. Dokumentáciou k nemu je [1], ktorá zároveň v prílohe obsahuje veľkú časť zdrojového kódu.

MINIX sám o sebe je písaný hlavne ako nástroj na učenie, hoci najnovšie verzie 3.x už majú za cieľ byť aj plnohodnotným operačným systémom pre zariadenia s obmedzenými prostriedkami.

Pre študentov je MINIX vhodný hlavne na štúdium toho, ako funguje mikrojadro a systémy na ňom postavené. Veľkým pozitívom je dostupnosť dobre dokumentovaných zdrojových kódov, ako tiež samostatnej knihy. Problematický ale môže byť samotný fakt, že MINIX je operačný systém – je navrhnutý tak, aby bežal priamo na reálnom hardvéri, ktorého ovládanie je často dosť komplikované.

Demonštrácia niektorých nízkoúrovňových vlastností hardvéru môže byť vďaka tomuto dosť náročná – na jeho pochopenie je potrebné naštudovať si netriviálne množstvo dokumentácie.

1.1.3 TOPSy

Teachable OPerating System[4] je ďalší systém vytvorený pre akademické účely, vyvinutý na ETH v Zürichu. Prvá verzia bola určená pre architektúru MIPS, ďalšie boli portované na viacero platforiem. Jeho využitie bolo podobné ako pri MINIXe.

V celom systéme existujú dva procesy – jeden pre jadro a jeden pre používateľské vlákna, pričom je podporovaný multithreading. Systém podporuje správu pamäte, medziprocesovú komunikáciu (message passing) a preemptívny multitasking. Pri implementácii bol dôraz kladený na jednoduchosť a čitateľnosť kódu.

1.2 Cieľ

Základným cieľom tejto práce bolo vytvoriť sadu programov, ktorá by dostatočne jednoduchým a flexibilným spôsobom umožňovala vytváranie materiálu (cvičení, ukážok) zameraného na tematiku operačných systémov.

Cieľom tejto práce nebolo nahradiť všetky doteraz vytvorené nástroje na výučbu operačných systémov – tieto nástroje sú oveľa vyspelejšie, prepracov-

anejšie a vyvíjajú sa dlhšie, ako existuje táto práca. Cieľom bolo poskytnúť alternatívne riešenie pokrývajúce témy, ktoré boli súčasnými riešeniami menej podporované (v tomto prípade sa jedná napríklad o prácu s hardvérom a nízkoúrovňové operácie), ako tiež poskytnúť nový pohľad na problematiku a nástroj na prípravy nových úloh.

Dôležitou motiváciou za touto snahou bolo nevytvárať len sadu programov a zadaní, ktoré budú študenti a cvičiaci môcť využívať – na tento účel postačujú vyššie uvedené programy. Hlavnou myšlienkou bolo vytvoriť základ, nad ktorým bude možné jednoducho vytvárať cvičenia upravené pre konkrétne prednášky (prípadne dynamicky rozširovať alebo meniť obsah podľa záujmu študentov).

1.2.1 Návrh riešenia

Základné ciele, ktoré si kládla táto práca, sú:

- Jednoduchosť pre študentov aj cvičiacich
- Modifikovateľnosť/rozšíriteľnosť pre nové druhy úloh

Na dosiahnutie tohto cieľa sme využili koncept *virtuálneho počítača*. Ten pozostáva z prideleného pamäťového priestoru a procesorového času. Vonkajšie prostredie virtuálneho počítača (napríklad zariadenia namapované v pamätovom priestore) je emulované iným programom. Toto umožňuje vytvoriť si vlastný zjednodušený hardvér, nad ktorým už prebieha konkrétne cvičenie.

Inštrukčná sada virtuálneho stroja je taká istá, ako natívneho počítača, avšak hardvér tohto počítača bude oproti reálnemu zjednodušený natoľko, aby bolo možné sústrediť sa na podstatné časti cvičení. Praktická implementácia pozostáva z komponent, ktoré sú popísané nižšie.

Základným komponentom architekúry je emulátor *Qemu* (ktorý je ale v prípade potreby možné nahradiť iným emulátorom, prípadne skutočným hardvérom). Nad *Qemu* beží emulačné jadro, ktoré má na starosti správu prostriedkov pre procesy a virtuálne počítače.

Nad emulačným jadrom bežia samotné virtuálne počítače a procesy. Virtuálny počítač je proces, o ktorého výnimky (v prípade tejto práce výpadok stránky) sa stará správcovský proces. Správcovský proces môže delegovať ošetrenie výnimky ďalej na nejaký iný proces.

Vďaka takejto architektúre je možné mať kód, ktorý zabezpečuje emuláciu hardvéru, dostatočne oddelený, čo jednak zvyšuje jeho znovupoužiteľnosť, a tiež pomáha pri ladení – v porovnaní s monolitickým správcom procesu, ktorý obsluhuje všetok hardvér, je oblasť kódu, v ktorom mohla nastať chyba, oveľa menšia.

V ďalších častiach sú detailnejšie popísane jednotlivé časti systému.

1.2.2 Emulačné jadro

Jeho úloha bude podobná, ako je úloha jadra operačného systému, pričom návrh berie za vzor mikrojadrový dizaj. Implementovaná teda bude hlavne jednoduchá správa pamäte a procesora. Navyše bude poskytovať systémové volania umožňujúce spravovať procesy a virtuálne počítače a odchytávať operácie, ktoré je nutné špeciálne ošetriť (napríklad výnimky spomínané vyššie).

Jadro je pre maximálnu jednoduchosť implementované hlavne pre emulátor PC – *Qemu*. Toto umožňuje jednak jednoduchý vývoj bez nutnosti podporovať širokú škálu hardvéru, a tiež zjednodušuje neskoršie používanie – emulátor je obyčajná aplikácia, ktorú môžu študenti bez väčších zásahov nainštalovať na ľubovoľný počítač.

Návrh založený na dizajne mikrojadra bol zvolený hlavne z nasledovných dôvodov:

- Rozdelenie vývoja na dostatočne malé (a relatívne jednoducho kontrolovateľné) komponenty s jasne špecifikovaným rozhraním umožňuje jednoduché testovanie komponent.

- Oproti monolitickému riešeniu, kde by všetky emulačné komponenty zdieľali jeden adresný priestor, v tomto prípade vieme vo väčšine prípadov

jednoznačne určiť komponentu zodpovednú za nekorektné správanie sa systému. Toto môže byť veľká pomoc pri vývoji alebo upravovaní virtuálnych zariadení.

- Jedná sa o projekt určený pre akademické účely, kde nižší výkon, ktorý býva spájaný s mikrojadrami, nie je podstatný.

1.2.3 Virtuálny hardvér

V tomto bode je lepšie hovoriť o programe, resp. sade programov, ktoré implementujú zjednodušený hardvér využívaný virtuálnym počítačom. Zároveň sa jedná o časť, ktorá je určená na modifikáciu alebo rozširovanie pre účely konkrétnych cvičení.

Hoci samotný virtuálny hardvér je zjednodušený, dôležité je, aby sa nestratili niektoré jeho základné črty, ktoré sú dôležité pre vlastné cvičenia. Napríklad samotný procesor okrem vykonávania inštrukcií (ktoré budú v tomto prípade vykonávané natívne) má v sebe integrovanú jednotku správy pamäte, podporu pre prerušenia a veľa ďalších komponent.

Virtuálny hardvér môže využívať (z pohľadu operačného systému na virtuálnom počítači) také isté prostriedky, ako súčasný reálny hardvér – registre zariadení mapované do adresného priestoru, generovanie prerušení, DMA, ... Závisí na konkrétnej implementácii hardvéru, ktoré z poskytovaných možností využije.

Opäť platí podmienka, že virtualizovaný hardvér by mal byť vytvorený na základe už existujúceho, reálneho, ale mal by byť jednoduchý na ovládanie. V rámci tejto práce bol implementovaný jednoduchý jednosmerný sériový port[1], ktorý ukazuje, ako je možné simulovať hardvér mapovaný do pamäťového priestoru.

[1] `examples/polling-master.c`

1.2.4 Príklady na možné zadania

Predchádzajúce dve časti vytvárajú akýsi základ, na ktorom je možné stavať vlastné zadania úloh, pričom tieto úlohy je možné rozdeliť do viacerých tematických celkov:

- **Základy práce s hardvérom** – cvičenia zamerané osvojenie si elementárnych techník používaných pri práci s hardvérom (MMIO, prerušenia, DMA, ...).
- **Koordinácia paralelne bežiacich procesov** – táto časť je podobná *Nachos Phase 2* a zaoberá sa paralelne bežiacimi systémami.
- **Oddelenie kernel- a user-space** – ťažisko je na implementácii správy pamäte, procesora a prepínaní úloh, ako tiež implementácii systémových volaní
- **Správa súborov** sa zaoberá implementáciou jednoduchého súborového systému a zodpovedajúcich systémových volaní.
- **Práca v sieti** používa virtuálne počítače prepojené navzájom sériovými linkami do kruhu na vytvorenie jednoduchej siete typu token-ring.

Tieto príklady zadaní majú slúžiť iba ako inšpirácia na to, čo je možné spraviť so systémom a ich účel je hlavne byť štartovacím bodom pre vlastné modifikácie a zadania, ktoré budú prispôsobené konkrétnym prednáškam a cvičeniam.

1.2.5 Dokumentácia

Táto časť je zameraná na zdokumentovanie všetkých častí, ktoré boli v rámci tejto práce implementované. Keďže hlavným cieľom je vytvoriť prostredie, ktoré bude ďalej modifikované, je dokumentácia nutnou pomôckou pre každého používateľa.

1.2.6 Iné možnosti riešenia

Vyššie uvedené riešenie samozrejme nie je jediné, ktorým sa dá pôvodný cieľ práce dosiahnuť. Zvažované boli rôzne riešenia, napríklad:

- Úprava samotného *qemu* namiesto písania vlastného emulačného riešenia. Tento prístup bol zavrhnutý kvôli jeho neflexibilnosti – virtuálny hardvér by bol buď neoddeliteľnou súčasťou emulátora, alebo dynamicky načítavaný. V obidvoch prípadoch by bol súčasťou monolitického emulátora a ťažšie opravovateľný.

  Navyše zostáva komplikovanosť PC architektúry a pridáva sa nutnosť udržiavania upraveného kódu *qemu*. API *qemu* má oveľa väčší potenciál na zmenu, ako rozhranie v samotnom emulovanom počítači (toto je v konečnom dôsledku stabilné už niekoľko desiatok rokov). Jednou možnosťou by bolo začlenenie kódu priamo do *qemu*, čo však kvôli dosť rozdielnej povahe (*qemu* je emulátor reálneho PC, zatiaľ čo táto práca navrhuje v reále neexistujúci hardvér). Ďalej je možnosť vytvorenia vlastnej vetvy *qemu*, pričom je ale potrebné pravidelne udržiavať úpravy, aby boli použiteľné na nové verzie.

- Návrh kompletného vlastného virtuálneho počítača. Pri tomto prístupe by sa väčšina času vývoja bola spojená s vyvíjaním procesora a potrebných nástrojov. Bolo by tiež možné použiť existujúce emulátory procesorov (prípadne bytecode, ako napríklad LLVM[5][2], avšak zrejme najjednoduchšie a najdostupnejšie sú nástroje pre natívnu architektúru. Navyše, takýmto spôsobom za znižuje závislosť na type kompilátora a s istou dávkou úsilia je možné používať aj iné nástroje.

1.3 Základné pojmy

Tieto pojmy sa budú vyskytovať v ďalšom texte vždy v nasledujúcom význame:

[2]Jednoduchý nízkoúrovňový jazyk navrhnutý tak, aby z neho bolo jednoduché generovať kód pre rôzne architektúry

- **jadro** – pokiaľ nie je uvedené ináč, jedná sa o emulačné jadro
- **kontext** – adresný priestor spolu s priradenými stránkami pamäte a stavom procesora
- **virtuálny počítač** – kontext, ktorý je spravovaný iným procesom..
- **proces** – kontext, ktorý nemá správcu.
- **správca virtuálneho počítača** – proces, ktorý spravuje určitý virtuálny počítač, stará sa o všetky akcie programu bežiaceho v kontexte virtuálneho počítača, ktoré potrebujú externý zásah.

Kapitola 2

Emulačné jadro

2.1 Inicializácia

Emulačné jadro je prvá komponenta systému, ktorá sa načítava a pripravuje prostredie pre beh ďalších komponent, čo na architektúre x86 znamená hlavne prepnutie do chráneného režimu, nastavenie tabuľky globálnych deskriptorov, tabuľky deskriptorov prerušení a stránkovania.

Hoci väčšinu z vyššie popísaných úloh je možné zabezpečiť ručne, pre účely tejto práce bolo zvolené využitie zavádzača GRUB2[7] a zavádzanie pomocou *Multiboot*[8][1]. Takéto riešenie má viacero výhod:

- Zavádzač podporuje prácu s binárnym formátom ELF[6], čo značne zjednodušuje prácu a umožňuje využívať priamo staticky linkovaný ELF výstup z kompilátora, bez potreby vytvárania špeciálneho obrazu.
- Multiboot zavádzač automaticky nahrá do pamäte celý spustiteľný ELF obraz (prípadne ďalšie moduly), umožňuje jednoduchým spôsobom predávať parametre jednotlivým komponentom
- Umožňuje nahrávanie ELF obrazu uloženého na rôznymi spôsobmi,

[1] špecifikácia pre uľahčenie štartovania operačných systémov – určuje formát obrazu spustiteľného kódu, ktorý sa má vykonávať, stav prostredia, v ktorom bude kód bežať a tiež spôsob predania základných informácií o stave systému

napríklad v súborovom systéme (FAT, EXT2/3, ...), z TFTP servera na sieti.

- ELF obraz je spúšťaný v dobre definovanom prostredí (procesor je prepnutý do chráneného režimu, 32-bitové prostredie, vypnuté prerušenia).

Po zavedení jadro vytvára segmentové deskriptory pre seba a pre používateľské aplikácie, v tabuľke deskriptorov prerušení nastavuje ošetrovanie výnimiek a prerušení, vytvára tabuľku stránok a zapína stránkovanie a prerušenia časovača. Zároveň inicializuje alokačné algoritmy pre pamäť (fyzickú pamäť aj virtuálny adresný priestor).

V tomto momente je jadro pripravené poskytovať svoje funkcie ďalším komponentám, ktorými sú moduly načítané pomocou multiboot-u. Pre každý z nich jadro vytvára samostatný kontext, v do ktorého sa v prvom rade nahrá spustiteľný obraz načítaný z ELF súboru a registre procesora sa inicializujú na správne hodnoty. Následne sa kontext nastaví do stavu, kedy ešte nemôže byť vykonávaný. Po spracovaní všetkých modulov sa kontextu s prvým načítaným modulu predá riadenie (ostatné kontexty zostávajú zastavené).

2.2 Prehľad komponent jadra

Interné rozhrania jadra používajú navzájom všetky komponenty jadra, navyše niektoré z nich exportujú ďalej používateľským procesom. Tieto rozhrania sa navyše rozdeľujú na platformovo závislé a nezávislé. V prvej kateogórií sa nachádzajú napríklad ovládače na nevyhnutnú množinu hardvéru (napríklad obsluha prerušení, jednotka správy pamäte), v druhej sú potom knižničné funkcie používané vnútri jadra.

2.2.1 Inicializácia systému

V nasledujúcich častiach sú uvedené funkcie, ktoré sa starajú o inicializáciu celého systému. Ich poradie je zhodné s poradím ich volania počas štartovania.

`multiboot_init` Spracúva informácie z prostredia, ktoré boli dodané podľa Multiboot špecifikácie – jednak informácie o dostupnej pamäti, ako aj informácie o moduloch prítomných v pamäti.

`mmu_init` (`arch/x86/mmu.c`) Zapne stránkovanie a nastaví počiatočný adresár stránok.

`interrupt_init` (`arch/x86/interrupts.c`) Inicializuje obsluhu prerušení na procesore – nastaví popisovače prerušení v Interrupt Descriptor Table tak, aby smerovali na IDT trampolíny (`arch/x86/idt_trampolines.S`).

`kmem_init` (`kernel/kmem.c`) Inicializuje najjednoduchšiu správu pamäte pomocou `kheap_request`, ktorá postupne posúva hranicu využitej pamte.

`physmem_init` (`kernel/physmem.c`)(`kernel/physmem.c`) inicializuje alokátor fyzickej pamäte. Po inicializácii je možné používať volania ako `physmem_alloc`, `physmem_free`, ako tiež `physmem_mark_used` na označenie už využitej pamäte (napríklad pamäť zabratá vlastným jadrom alebo nahratými modulmi.

`kmem_iomem_init` a `kmem_heap_init` (`kernel/kmem.c`) Inicializujú alokáciu I/O pamäte používanej na mapovanie stránok a haldy používanej na normálnu alokáciu pamäte.

2.2.2 Správa pamäte

V tejto časti sú dve základné rozhrania, správa fyzickej pamäte a správa virtuálneho adresného priestoru a jedno rozhranie zabezpečujúcu štandardnú `malloc`/`free` funkcionalitu. Prvé dve interne používajú jednduchú bitmapu na označovanie alokovaného alebo voľného bloku pamäte, pričom najmenšia jednotka na alokovanie je určená veľkosťou stránky. Rozhranie `kmalloc`/`kfree` postupne posúva hranicu voľnej pamäte o alokované miesto, uvoľnovanie pamäte je prázdna operácia.

Začiatok	Koniec	Popis
0x100000	kernel_end	Kód a dáta emulačného jadra
kernel_end	kernel_end	vyhradené pre dynamickú
	+ 0x7fffff	alokáciu pamäte jadra
kernel_end	kernel_end	vyhradené pre dynamickú
+ 0x800000	+ 0xffffff	alokáciu pamäťových stránok,
		prípadne iba adresného priestoru
		pre ručné mapovanie stránok.
0x7ffff000	0x7fffffff	zásobník jadra
0x80000000	0x8fffefff	kód kontextu
0x8ffff000	0x8fffffff	zásobník kontextu

Tabuľka 2.1: Mapa adresného priestoru

Alokácia fyzickej pamäte je spoločná pre všetky kontexty, pri virtuálnom adresnom priestore má každý kontext svoj vlastný adresný priestor (avšak adresný priestor jadra je pre všetky kontexty spoločný). Pri oboch rozhraniach je použitý jednoduchý alokačný algoritmus, ktorý nájde prvý výskyt dostatočne dlhého súvislého bloku pamäte a ten použije.

Alokácia virtuálneho adresného priestoru je rozdelená na niekoľko častí – virtuálny adresný priestor pre aplikácie sa alokuje v zápornej časti[2] adresného priestoru, na alokovanie I/O pamäte je vyhradených 8MB a na haldu pre `malloc/free` ďalších 8MB. Celkový rozloženie adresného priestoru je uvedené v tabuľke 2.2.2.

2.2.3 Rozhrania závislé na architektúre

Tieto rozhrania sú väčšinou umiestnené v adresári `arch/` a obsahujú kód viazaný na architektúru x86, jej hardvér a špecifickú inicializáciu pri spúšťaní.

[2] najvyšší bit je nastavený na 1

Inicializácia Inicializačný kód je umiestnený v `setup.S`, na neskoršie fázy sa používa aj `multiboot.c` a ďalšie. `setup.S` inicializuje behové prostredie, nastaví potrebné deskriptory, spracuje multiboot informácie, nastaví prerušenia a jednotku správy pamäte. Nakoniec inicializácia pokračuje v `kernel/init.c`.

Grafický adaptér Implementácia sa nachádza v súbore `vga_console.c`. Toto rozhranie má jedinú funkciu `putchar`, ktorá na obrazovku vypíše jeden znak, alebo interpretuje riadiaci znak. Zároveň pre potreby ladenia vypisuje táto funkcia znaky aj na prvý sériový port počítača.

Jednotka správy pamäte Spravuje adresáre a tabuľky stránok, pričom má na starosti aj to, aby všetky mapovania virtuálnej pamäte mali zhodne mapované prvé 2GiB adresného priestoru. `mmu_init` inicializuje tabuľku stránok a zapne stránkovanie. Funkcia `mmu_mapping_init` inicializuje štruktúru obsahujúcu mapovanie pamäte, a funkcie `mmu_map_range` a `mmu_unmap_range` sa spravujú tieto mapovania. Funkcia `mmu_virt_to_phys` vykonáva preklad z virtuálnej pamäte na fyzickú.

Práca s procesorom Súbor `cpu.h`, ktorý obsahuje funkcie potrebné na ukladanie a obnovovanie stavu procesora (`cpu_state_save`, `cpu_state_restore`), prepnutie do používateľského režimu (`cpu_resume_userspace`).

2.2.4 Zdieľané knižničné funkcie

Implementácia týchto funkcií je `lib/` – nemajú funkciu, ktorá je systémová alebo priamo závislá na hardvéri. Implementované sú napríklad:

- `printf`,
- základné funkcie na prácu s pamäťou `strcpy`, `strncpy`, `memcpy`, `memset`
- jednoduchý bitmapový alokátor. Okrem jadra sú využívané aj v používateľských aplikáciách.

2.3 Rozhrania poskytované pre emuláciu

Inicializované emulačné jadro tvorí základné rozhranie medzi hardvérom a emulačným prostredím, a po zavedení a inicializácii poskytuje nasledovné služby:

- správa kontextov
- správa pamäte
- správa procesora
- medziprocesová komunikácia

Niektoré z týchto službieb sú ostatným komponentom spokytované pomocou sád systémových volaní. Na rozdiel od skutočných operačných systémov sú uvedené funkcie implementované iba na jedndouchej úrovni tak, aby splnili svoju základnú funkciu – ich hlavným účelom nie je efektivita, ale jednoduchosť a čitateľnosť.

Dôležitým bodom pri určovaní rozsahu vlastností emulačného jadra je schopnosť umožniť emulačnému softvéru vytvoriť čo najvernejšiu simuláciu. Toto napríklad znamená umožniť ošetrenie výnimiek, ktoré sa môžu vyskytnúť pri práci procesora, napríklad výpadok stránky, neplatná inštrukcia, ... Toto bolo zabezpečené pomocou medziprocesovej komunikácie, v tomto prípade ale jadrom a správcom virtuálneho počítača.

Rovnako podstatné je umožniť používateľom prípadné ladenie programov a hľadanie chýb, aspoň na najjednoduchšej úrovni. Správca virtuálneho počítača má teda mať plný prístup do kontextu virtuálneho počítača – do jeho pamäťového priestoru, aj do uloženého stavu procesora. Takisto je podstatné, aby správca mohol zastavovať a spúšťať virtuálny počítač.

Samotné systémové volania, ktoré sú poskytnuté kontextom, sú pre x86 implementované pomocou prerušení, a ich popis je uvedený ďalej v tejto kapitole.

V nasledujúcich častiach sú popísané jednotlivé rozhrania, ktoré jadro poskytuje.

2.3.1 Správa kontextov

Ľubovoľný program musí byť vykonávaný v rámci nejakého kontextu (jadro systému je vždy zdieľané medzi všetkými kontextmi), v ktorom sa drží všetko, čo je potrebné na jeho vykonávanie – mapovanie pamäte, stav registrov, stav potrebný pre plánovač úloh a buffer na prijatie správy.

`context_create ()` Toto volanie alokuje nový kontext a vráti pointer na novo inicializovanú štruktúru, ktorá ho popisuje. Neobsahuje žiadne mapované stránky, všetky registre sú nastavené na štandardné hodnoty.

`context_switch (`*`context_id`*`)` Uloží stav starého kontextu a prepne sa kontext *context_id*.

`context_resume_current ()` Prepne sa z režimu jadra do používateľského režimu a pokračuje vo vykonávaní aktuálne nastaveného kontextu.

`context_set_ip (`*`ip`*`)` Nastaví hodnotu IP[3] na požadovanú adresu.

`context_set_sp (`*`sp`*`)` Nastaví hodnotu SP[4] na požadovanú adresu.

`context_get_by_id(`*`id`*`)` Vráti štruktúru pre kontext určený jednoznačným identifikátorom *id*.

`context_get_id(`*`context`*`)` Vráti jednoznačný identifikátor pre kontext *context*.

2.3.2 Správa pamäte

Každý kontext musí mať pre svoju korektnú činnosť priradenú pamäť (minimálne pamäť, v ktorej je uložený vykonávaný program), ktorá je namapovaná do

[3]Instruction Pointer
[4]Stack Pointer

jeho adresného priestoru. Navyše je niekedy nutné, aby isté časti pamäte boli dostupné iba na čítanie, prípadne pre zápis. Navyše, keďže toto rozhranie sa bude používať aj na prideľovanie pamäte procesom (a potenciálne ovládačom reálneho hardvéru), je nutné pri alokácii pamäte zohľadniť aj požiadavky na DMA:

- Pamäť musí byť alokovaná v iba istom rozsahu.
- Rozsah alokovanej pamäte musí byť súvislý nielen vo virtuálnej pamäti, ale aj vo fyzickej.
- Jadro musí poskytovať možnosť prekladu virtuálnej adresy na fyzickú, čo je nevyhnutný predpoklad na fungovanie DMA.

Tento problém je najjednoduchšie riešiť priamou alokáciou fyzických stránok v pamäti a ich následným mapovaním do userspace. Takéto riešenie jednak umožní používateľským programom vytvárať si medzi sebou zdieľanú pamäť, zároveň to však je aj akási forma základnej podpory pre multithreading v prípade, že mapovania pamäte dvoch kontextov sú úplne rovnaké.

Zahŕňajúc tieto požiadavky, správa pamäte má o čosi zložitejšiu štruktúru ako správa kontextov a je rozdelená na správu fyzickej pamäte a správu mapovaní.

`physmem_alloc (size, flags)` Alokuje stránky vo fyzickej pamäti počítača. Navyše, tieto stránky sú súvislé. Návratová hodnota je počiatočná adresa fyzickej stránky. *size* bude interne zaokrúhlená na najbližší vyšší násobok veľkosti stránky. *flags* je vyhradené pre budúce rozšírenie a musia byť nastavené na 0.

`physmem_free (physaddr)` Uvoľní stránky alokované volaním `physmem_alloc`. Pokiaľ stránky boli mapované, je treba ich ručne uvoľniť zo všetkých mapovaní.

Ďalej nasleduje popis rozhrania na mapovanie pamäte. Funkcie pracujú so štruktúrou *mmu_mapping_t*, v ktorej sú uložené všetky záznamy o mapo-

vaniach virtuálnej pamäte na fyzickú. Jej presný tvar závisí na architektúre, na x86 obsahuje priamo adresár stránok a odkazy na gq tabuľky stránok.

`mmu_map_range (`*`mapping`*`, `*`virt`*`, `*`phys`*`, `*`size`*`, `*`flags`*`)` Namapuje stránky v *phys* o veľkosti *size* do mapovania *mapping* na adresu *virt*. Pokiaľ by sa mapovanie prekrývalo s už existujúcim mapovaním, stránky z mapovania budú najprv odstránené. Hodnota vo *flags* vzniká operáciou OR medzi nasledovnými konštantami:

- `PAGE_KERNEL` alebo `PAGE_USER` podľa toho, či stránka má byť dostupná iba jadru alebo aj používateľovi.
- `PAGE_READABLE` alebo `PAGE_WRITABLE` podľa toho, či stránka má byť iba čitateľná, alebo aj zapisovateľná.

Ak niektoré z rozsahu boli dostupné už predtým, funkcia ich pred zmenou odmapuje.

`mmu_unmap_range (`*`mapping`*`, `*`address`*`, `*`size`*`)` Uvoľní mapovanie stránok dĺžky *size* v *mapping*, ktoré sú namapované od adresy *address*. Nenamapované stránky budú ignorované.

`mmu_virt_to_phys (`*`mapping`*`, `*`virt`*`)` Vykoná preklad adries z virtuálnej na fyzickú. Pokiaľ stránka nie je mapovaná, vráti -1 .

2.3.3 Správa procesora

Prideľovanie procesorového času prináša so sebou isté problémy – správcovia kontextov totiž musia mať kontrolu nad behom virtuálnych počítačov, teda sú nutné systémové volania na nastavenie stavu jednotlivých kontextov.

Samotný kontext vie byť v týchto stavoch:

1. `NEW` – čerstvo vytvorený kontext, ktorý sa nesmie spúšťať, iný kontext alebo jadro ho môže prepnúť do stavy `READY`

2. `READY` – kontext vo fronte pripravený na spustenie
3. `RUNNING` – aktuálne bežiaci kontext
4. `STOPPED` – zastavený kontext, ktorý sa nesmie spúšťať (iný kontext ho ale môže prepnúť do stavu `READY`).

`context_set_sched_state (context_id, state)` Volanie nastaví stavu kontextu *context_id* na jednu z možných hodnôt:

- `STATE_READY` – kontext bude normálne zaraďovaný do fronty procesov čakajúcich na beh a bude mu prideľovaný procesorový čas.
- `STATE_STOPPED` – kontextu nebude prideľovaný procesorový čas. Z tohto stavu ho môže dostať jedine externá udalosť.

`sched_schedule_next()` Funkcia prepne bežiaci proces do stavu `READY` a pomocou algoritmu Round-Robin vyberie ďalší (v najhoršom prípade ten istý) proces schopný behu a prepne ho do stavu `RUNNING`.

2.3.4 Medziprocesová komunikácia

Táto časť jadra nemá žiadne priame spojenie s virtuálnymi počítačmi, používa sa však na komunikáciu medzi procesmi (napríklad správca kontextu a ovládače na jednotlivé zariadenia). Samotná komunikácia je založená na predávaní správ a na tejto úrovni je bezstavová (jednotlivé procesy si medzi sebou môžu posielať správy) a asynchrónna.

```
typedef struct message_header {
        unsigned int length;
        unsigned long to;
        unsigned long from;
} message_header_t;
```

Listing 2.1: struct *message_header*

Štruktúra správy je navrhnutá tak, aby jej hlavička už obsahovala všetky potrebné dáta na odoslanie; nie je teda nutné uvádzať ich ako ďalšie parametre pre systémové volania.

Dôležitá je tu aj otázka bezpečnosti – je dôležité, aby ľubovoľný proces nemohol posielať dáta zamaskovaný za iný kontext. Preto platí pravidlo, že zdrojové *context_id* môže byť nastavené iba na aktuálny kontext, v opačnom prípade bude jadrom prestavené na ID aktuálneho kontextu.

Interne je medziprocesová komunikácia implementovaná pomocou jediného prijímacieho buffera, to znamená, že pre jeden kontext nemôže naraz dostať viacero správ. Pokiaľ by sa nejaký kontext mal dostať správu, hoci už jedna v prijímacom bufferi je, funkcia `send` vráti hodnoti -1.

Správy sú tiež obmedzené ich maximálnou veľkosťou na `MSG_MAX_LENGTH` (štandardne 1024) včetne hlavičky. Ľubovoľná väčšia správa bude funkciou `send` odmietnutá.

`send (`*`message`*`)` Zaradí správu do fronty na odoslanie a vráti 0, ak všetko prebehlo úspešne, v opačnom prípade vráti -1.

`recv (`*`message, length`*`)` Ak existuje správa vo fronte a jej dĺžka je najviac *length*, skopíruje ju na adresu *message*, a vráti 0. Pokiaľ neexistuje, štruktúru *message* nevyplní a vráti nenulovú hodnotu. V prípade chyby funkcia vráti -1.

2.4 Systémové volania

Systémové volania sú pre x86 implementované ako obsluha prerušenia $0x80$, pričom parametre pre systémové volanie sú uložené v registroch procesora:

- Vstupné parametre:
  - `EAX` – číslo systémového volania
  - `EBX` – parameter číslo 1

  - `ECX` – parameter číslo 2
  - `EDX` – parameter číslo 3
  - `ESI` – parameter číslo 4
  - `EDI` – parameter číslo 5

- Výstupné parametre:
  - `EAX` návratová hodnota systémového volania

Zoznam všetkých systémových volaní aj s ich kódmi je uložený v súbore `include/core/syscall.h`, vlastný kód na zavolanie je potom napríklad v `include/user/syscalls.h`.

Funkcionalita systémových volaní má dosť veľký prienik s tým, čo poskytujú funkcie popisované v predchádzajúcej časti, alebo s nimi úzko súvisia. Z tohto dôvodu je pri systémových volaniach uvádzaný aj odkaz na príslušnú predchádzajúcu časť.

`0x00` `recv(`*`msg, len`*`)` Prijme správu určenú pre kontext, viď 2.3.4.

`0x01` `send(`*`msg`*`)` Odošle správu, viď 2.3.4.

`0x10` `pmalloc(`*`size, flags`*`)` Alokuje rozsah fyzickej pamäte, viď 2.3.2.

`0x11` `pmfree(`*`addr`*`)` Uvoľní rozsah fyzickej pamäte, viď 2.3.2.

`0x20` `v2p(`*`ctx, addr`*`)` Preloží virtuálnu adresu *addr* v kontexte *ctx* na adresu fyzickej pamäte, viď 2.3.2.

`0x21` `map(`*`ctx,vaddr,paddr,len,flags`*`)` Namapuje na adresu *vaddr* rozsah fyzických stránok začínajúcich na *paddr* a dĺžky *len*. Flags môže nadobúdať hodnoty `PAGE_READ` alebo `PAGE_WRITE` podľa toho, či je stránka dostupná iba na čítanie, alebo aj na zápis, viď 2.3.2.

`0x22` `unmap(`*`ctx, vaddr, len`*`)` Vymaže mapovanie virtuálnej adresy *vaddr* v kontexte *ctx* dĺžky *len*, viď 2.3.2.

`0x30` `getid()` Vráti ID aktuálneho kontextu.

`0x31` `nextid(id)` Vráti ID kontextu, ktoré nasleduje po *id*. Volanie funguje cyklicky nad všetkými existujúcimi kontextmi – keď vráti ID aktuálneho kontextu, boli prejdené všetky existujúce ID.

`0x32` `setstate(ctx, state)` Nastaví stav kontextu pre plánovač úloh – možné hodnoty pre *state* sú `STATE_READY` a `STATE_STOPPED`, viď 2.3.3.

`0x33` `setmanager(ctx, manager)` Nastaví správcu kontextu *ctx* na *manager*.

`0x34` `getcpu(ctx, cpu, len)` Získa štruktúru s aktuálnymi nastaveniami registrov procesora v kontexte *ctx* a uloží ju na adresu *cpu*. *len* musí byť nastavené na veľkosť pamäte, ktorá je vyhradená pre štruktúru. V prípade úspechu vráti počet uložených bajtov, v prípade neúspechu -1.

`0x34` `setcpu(ctx, cpu, len)` Kontextu *ctx* nastaví registre na hodnoty uložené na adrese *cpu*, ktorý má dĺžku *len*. V prípade úspechu vráti 0, v prípade neúspechu (napr. zlá dĺžka štruktúry) -1.

`0x47` `switchtask()` Vynúti si prepnutie úlohy viď 2.3.3

`0xfe` `mputchar(c)` Vypíše jeden znak na terminál monitora, viď 2.2.3.

`0xff` `puts(s)` Vypíše na terminál monitora signatúru kontextu a reťazec *s*.

2.4.1 Obsluha výnimiek

Na rozdiel od systémových volaní popísaných v predchádzajúcej časti sú výnimky generované virtuálnymi počítami čisto asynchrónne udalosti. Správca virtuálneho počítača je preto na ne upozornený pomocou mechanizmu medziprocesovej komunikácie, pomocou správy od kontextu s ID 0 – emulačného jadra.

Všeobecný formát správy, ktorou je správcovský proces upozorňovaný na jednotlivé výnimky, je nasledovný (všetky potrebné údaje sú uvedené v `include/core/manager_msg.h`):

```
        typedef struct {
        message_header_t hdr;
        manager_msg_type_t type;
        unsigned long context;
} manager_msg_header_t;
```

Obr. 2.1: struct manager_msg_header_t

```
typedef struct {
        manager_msg_header_t mhdr;
        unsigned long address;
        page_fault_op_t operation;
} manager_msg_page_fault_t;
```

Obr. 2.2: struct manager_msg_page_fault_t

V tejto štruktúre, *context* je kontext, ktorého sa týka udalosť a *type* je typ udalosti, ktorý nastal. V súčasnosti sú podporované nasledovné možnosti:

- `NONE` – táto hodnota sa v správe nesmie vyskytnúť.
- `PAGE_FAULT` – označuje výpadok stránky, prijatá je štruktúra:

  address určuje adresu, kde došlu ku výpadku a *operation* môže mať hodnotu buď `PF_OP_READ`, ak výpadok nastal pri čítaní a `PF_OP_WRITE`, ak pri zápise.

Doplnenie obsluhy ďalších výnimiek je možné pomocou úpravy súborov `arch/x86/interrupts.c`, pridania príslušných ošetrovacích funkcií do súborov `kernel/handlers.c`, štruktúr a typov do `include/core/manager_msg.h` a `include/core/handlers.h`.

2.4.2 Ďalšie možné rozhrania

Rozhrania a systémové volania popísané vyššie poskytujú systému elementárnu funkcionalitu, avšak nezahŕňajú veľa funkcií, ktoré by bežný používateľ očakával,

napríklad zisťovanie podrobnejších informácií o mapovaniach. Tieto pre účely tejto práce nebolo potrebné implementovať, ale môže byť námetom na ďalšie rozširovanie.

Na rozširovanie funkcionality samotného jadra je dobré použiť adresár `arch/` v prípade kódu závislého na architektúre a `kernel/` pre všeobecný kód. V prípade rozširovania sady systémových volaní je treba upraviť súbory `include/core/syscall.h`, `include/user/syscalls.h` a samotná obsluha systémových volaní začína v `kernel/syscall.c`.

Kapitola 3

Správca virtuálneho počítača

Tento komponent je kľúčový pre vlastný beh virtuálneho počítača. Stará sa o dve časti života virtuálneho počítača. Zvyčajne je ním prvý kontext ktorý je do systému nahraný a spustený. V tomto čase sa stará o prvotnú inicializáciu, ako namapovanie správnych častí pamäte pre seba aj pre virtuálny počítač, a v konečnom dôsledku aj o spustenie.

Druhou úlohou je vlastná podpora behu virtuálneho počítača, simulácia potrebného hardvéru a ošetrovanie výnimiek. Simulácia hardvéru pritom môže byť „posunutá“ aj ďalším procesom – nemusí byť vykonávaná iba v rámci procesu správcu, ale pomocou medziprocesovej komunikácie alebo zdieľaných pamäťových stránok môže byť hardvér simulovaný aj v samostatných procesoch.

Samotná simulácia hardvéru využíva zdieľanú pamäť a mechanizmy jadra, ktoré informujú správcu o vypadku pamäťovej stránky, prípadne jej zmene. Tieto informácie slúžia na zistenie zmien v registroch zariadení mapovaných do pamäte.

3.1 Simulovaný hardvér

V tejto časti je navrhnutý hardvér, ktorý je možné implementovať vo virtuálnom počítači, spolu s prostriedkami, ktoré využíva. V popisoch sú použité adresy

relatívne ku základnej adrese, kam je zariadenie mapované. Ukážkový popis hardvéru má slúžiť ako inšpirácia pre návrh vlastného hardvéru

3.1.1 Sériový terminál

Prostriedky: 2 bajty MMIO, voliteľne IRQ

So sériovým terminálom sa komunikuje prostredníctvom MMIO registrov o veľkosti 2 bajty, z čoho jeden je konfiguračný a druhý slúži na vstupno/výstupné operácie.

Bit	Popis	Po resete
0	Enable	0
1	RX acknowledge	0
2	TX acknowledge	0
3	Data in queue	0
4	TX busy	0
5	RX overrun	0
6	TX overrun	0
7	IRQ enabled	0

Tabuľka 3.1: Riadiaci register sériového terminálu na adrese $0x0$

Bit	Popis	Po resete
0-7	Dáta	nedef.

Tabuľka 3.2: Dátový register sériového terminálu na adrese $0x1$

- **Enable** bit slúži na zapnutie sériového terminálu – ľubovoľný zápis alebo čítanie počas nastavenej 0 nemá žiaden efekt.
- **RX acknowledge** zapísaním 1 sa radiču oznámi že byte z dátového registra bol úspešne prečítaný. Po spracovaní ho radič nastaví na 0.
- **TX acknowledge** zapísaním 1 sa radiču oznámi, že môže poslať bajt z dátového registra. Po spracovaní ho radič nastaví na 0.

- **Data in queue** *Iba na čítanie.* hovorí, že vo fronte sériového portu sú nejaké dáta, ktoré je možné čítať.
- **TX busy** *Iba na čítanie.* je nastavený na 1, pokiaľ sa práve odosielajú dáta. Zápis do dátového registra v tomto stave môže byť úspešný, ak je voľné miesto vo fronte; v opačnom prípade sa nastaví príznak **TX overrun** a znak sa neodošle.
- **RX overrun** *Iba na čítanie.* bude nastavený na 1 v prípade, že nastalo čítanie a vo fronte neboli žiadne dáta. Po prečítaní je nutné zapísať na jeho miesto 0 na vynulovanie.
- **TX overrun** *Iba na čítanie.* Bude nastavené na 1, pokiaľ sa preplnila fronta sériového portu. Po prečítaní je nutné zapísať 0 na vynulovanie.
- **IRQ enabled** určuje, či bude sériový terminál po prijatí bajtu generovať prerušenie.

Čítanie z dátového registra vracia hodnotu z fronty prijatých dát, pokiaľ, pokiaľ nejaké sú. V opačnom prípade vracia nedefinovanú hodnotu. Nastavenie *TX Acknowledge* spôsobí odoslanie znaku uloženého v dátovom registri.

Využitie Prijímacia časť (z vonkajšieho pohľadu; terminál dáta prijíma z vonka a program ich iba číta) sériového terminálu bez obsluhy prerušení je implementovaná v `examples/polling-master.c`. Správca najskôr namapuje do svojho adresného priestoru, aj do priestoru virtuálneho počítača na predom dohodnutú adresu zdieľanu pamäťovú stránku, inicializuje pamäť a spustí virtuálny počítač.

Správca potom čaká na zapnutie sériového terminálu, a postupne vysiela reťazec `Hello, world`. Virtuálny počítač má komunikovať s radičom terminálu a postupne všetky znaky prečítať. Ukážka programu pre virtuálny počítač je v `examples/polling-slave.c`.

Možné problémy

1. Je potrebné zapnúť radič, ináč nefunguje žiadna komunikácia
2. Pri čakaní na prijaté dáta je treba kontrolovať nielen bit *Data in queue*, ale aj *Rx acknowledge*, ktorý odlišuje či radič spracoval Rx Acknowledge.
3. Pri čakaní na dáta sa nesmie použiť konštrukcia, ktorá dvakrát priamo číta zo stavového registra – raz pre overenie prítomnosti Data in queue, druhý raz na Rx acknowledge. Takouto konštrukciou je napríklad (berúc ako základ ukážkový program pre virtuálny počítač:

   ```
   while (!(base[SR_OFFSET] & SR_DATA_IN_QUEUE) ||
           (base[SR_OFFSET] & SR_RX_ACK));
   ```

   Medzi prvým a druhým čítaním vzniká dostatočný časový priestor pre race condition, a v tomto prípade je možné, že prvé čítanie bude mať ešte nastavené oba bity (ake keďže aj *Data in queue*, prvá podmienka je splnená), a druhé čítanie žiaden bit (teda ani *Rx acknowledge*, a teda je splnená aj druhá podmienka a čakanie sa skončí, hoci v dátovom registri ešte nie sú platné dáta.

3.1.2 Systém prerušení

Prostriedky: 2*8 + 64*4 bajtov MMIO .

Slúži ako základný subsystém pre všetky zariadenia využívajúce prerušenia. Poskytuje 64 prerušení, ktoré môžu byť spojené s ľubovoľným hardvérom.

- **Interrupt mask** slúži ako príznakové registre pre maskovanie prerušení. Pokiaľ je na *i*-tom bite nastavená jednotka, pri výskyte prerušenia sa nevyvolá program ošetrujúci prerušenie, ale iba sa nastaví bit v **Interrupt pending**. Pokiaľ je prerušenie odmaskované, postupne od najnižšieho sa volajú všetky prerušenia, ktoré majú nastavený bit v **Interrupt pending**.

Adresa	Popis	Po resete
0x0 – 0x7	Interrupt mask	111...
0x8 – 0xf	Interrupt pending	0...
0x10 – 0x110	Interrupt vectors	0...

Tabuľka 3.3: Registre prerušovacieho systému

- **Interrupt pending** má na i-tom bite hodnotu 1 práve vtedy, ak nastalo prerušenie i. Po spracovaní je nutné bit manuálne nastaviť na 0 zápisom 0 na danú pozíciu.
- **Interrupt vectors** obsahuje 4-bajtové adresy obslužných programov prerušení od 0 po 63, na ktoré sa skáče v prípade prerušenia.

Prerušovací systém musí tiež poskytovať rozhranie ostatným procesom v systéme, ktoré chcú simulovať hardvér, preto musí poskytovať aj rozhranie na úrovni medziprocesovej komunikácie.

Využitie Virtuálnemu počítaču sú zo správcovského kontextu generované prerušenia, a to rýchlosťou, ktorá môže spôsobiť výskyt ďalšieho prerušenia už v čase, keď systém jedno obsluhuje. Ťažisko úlohy je v zabránení dvojnásobného vyvolania obsluhy prerušenia a zároveň v obslúžení všetkých prerušení, implementácia kombinácie obsluhy prerušenia a pollingu.

3.1.3 Jednotka správy pamäte

Jednotka správy pamäte (ďalej len MMU[1]) umožňuje stránkovanie a ochranu pamäte vo virtuálnom počítači, čo je základom pre veľké množstvo potenciálnych úloh zameraných na správu pamäte. Dôležitou súčasťou jednotky správy pamäte je TLB.

TLB (*Translation Lookaside Buffer* je dátová štruktúra, v ktorej sú uložené najčastejšie používané záznamy na preklad virtuálnej pamäte na fyzickú.

[1]Memory Management Unit

Adresa	Popis	Po resete
0x0 – 0x3	MMU Control Word	000...
0x10 – 0x210	TLB	000...

Tabuľka 3.4: Registre MMU

Bit	Popis	Po resete
0x0	Paging enable	0...
0x1	Protection enable	0...
0x2 – 0x1f	Reserved	0...

Tabuľka 3.5: MMU Control word

Zvyčajne má malý počet záznamov, ktoré je možné použiť. MMU môžeme podľa tobo, či sprístupňujú TLB, rozdeliť na dve kategórie

- jednotka schopná pracovať s tabuľkou stránok pre celý adresný priestor procesora (a interne spravujúca svoj TLB, ktorý aplikácie nevidia) – obsahuje dostatočný počet položiek pre celý adresný priestor
- jednotka umožňujúca priamu prácu s TLB – obsahuje obmedzený počet položiek, ktorý nepostačuje pokryť celý adresný priestor a generuje sa TLB miss. Proužívateľ je zvyčajne zodpovedný za uchovanie si celého mapovania virtuálnej na fyzickú pamäť, pretože do TLB sa nezmestí.

Z pedagogických dôvodov je lepšie použitie MMU s priamym prístupom do TLB – dôvodom tohto rozhodnutia je užitočnosť priameho prístupu na demonštráciu napríklad Beladyho paradoxu pri práci so stránkami. Nutné je tiež, aby systém podporoval prerušenia, ktoré sa používajú na signalizáciu výnimiek, ako napríklad neprítomnosť záznamu v TLB, porušenie ochrany pamäte, ...

MMU sa skladá z dvoch základných častí – riadiacich registrov a vlastnej TLB (viď 3.4). Detailnejší popis častí je v tabuľkách 3.5 a 3.6.

Pokiaľ sa v TLB vyskytne viac ako jeden záznam pre jednu virtuálnu adresu, použije sa prvý nájdený (t.j. ten s najnižším indexom). Ostatné

Adresa	Popis
0x3f – 0x2c	Virtual address
0x2b	Readable
0x2a	Writable
0x29	Executable
0x28	Used
0x27	Dirty
0x1f – 0x0c	Physical address
0x0b	Page size (0 = 4KiB, 1 = 4MiB)

Tabuľka 3.6: Položka TLB (8 bajtov)

záznamy v TLB nebudú kontrolované a nebudú ani aktualizované ich *Used* a *Dirty* príznaky.

Využitie Emulovaná MMU je užitočná hlavne na demonštráciu algoritmov na nahrádzanie stránok; príkladom môže byť nasledovné prostredie:

- Kontext, v ktorom beží používateľský program pracujúci s väčším množstvom dát, napr. algoritmus *quicksort*. Mapovanie jeho virtuálnej pamäte na fyzickú je spravované emulovanou MMU.
- Kontext, ktorý spravuje priestor používateľského programu, dostáva informácie o výpadkoch stránok a nahrádza a upravuje obsah TLB.
- Kontext, ktorý simuluje vlastný TLB – akákoľvek zmena v TLB sa prejaví v zmene mapovania kontextu, kde beží používateľský program.

3.1.4 Diskový radič

Prostriedky: MMIO + 1 IRQ

Umožňuje prístup ku fiktívnemu zariadeniu na ukladanie dát, pričom možnosti práce s ním sú:

- polling

Adresa	Popis	Po resete
0x0 – 0x3	Control Word	000...
0x4 – 0x7	Status Word	000...
0x8 – 0xb	Block address register	000...
0xc – 0xf	Data FIFO	000...
0x10 – 0x13	DMA address register	000...

Tabuľka 3.7: Registre diskového radiča

Bit	Popis	Po resete
0	Read(0)/Write(1)	0...
1	Interrupt enable	0...
2	DMA Enable	0...
3	Start transfer	0...
4	Buffer empty	0...

Tabuľka 3.8: Disk Controller Control Word

- signalizovanie dostupnosti dát pomocou prerušení
- DMA

Zariadenie je zložené z 512 bajtových blokov a používa lineárnu adresáciu, na riadenie a oznamovanie stavu používa oblasť MMIO (tabuľka 3.1.4), voliteľne doplnenú prerušeniami a DMA.

Významy jednotlivých položiek MMIO priestoru sú vysvetlné nižšie:

Radič umožňuje naraz pracovať s iba jedným blokom.

Po nastavení adresy bloku a parametrov prenosu (povolenie prerušení, DMA) sa zapísaním bitu *Start transfer* začne požadovaná akcia. Programátor (pokiaľ sa nepoužíva DMA) kontroluje príznaky v stavovom registri a periodicky číta/zapisuje dáta z/do FIFO registra.

Pri použití DMA je potrebné do *DMA Address Register* zapísať fyzickú adresu miesta v pamäti, kam sa uložia načítané dáta, resp. z kadiaľ sa budú čitať zapisované dáta. FIFO register sa v tomto prípade nepoužíva a čítanie z neho je nedefinované.

Bit	Popis	Po resete
0	Busy	0...
1	Buffer overrun	0...
2	Buffer underrun	0...
3	Buffer full	0...
4	Buffer empty	0...
5	Transfer complete	0...

Tabuľka 3.9: Disk Controller Status Word

Využitie Diskový radič sa môže využiť viacerými spôsobmi:

1. Ako samostatný hardvér vo virtuálnom počítači, kde je úlohou precvičiť si základné ovládanie a rôzne spôsoby prenosu dát

2. Ako hardvér pre kontext, ktorý dohodnutým protokolom dovoľuje ďalšiemu kontextu pracovať s diskom (a dovoľuje napr. merať rýchlosť práce s diskom), prípadne interpretuje obsah (ovládač na súborový systém).

3.1.5 Časovač

Prostriedky: MMIO + IRQ

Komponenta vygeneruje po ubehnutí určeného času (dosiahnutí rovnosti *Delay register* a *Counter*) prerušenie. Čas sa udáva v milisekundách. Nastavením bitu *Auto-reload* sa prerušenie bude vykonávať periodicky až do jeho vypnutia, alebo vypnutia časovača. Pri každom zápise do *Delay register* sa zároveň nuluje *Counter*.

Adresa	Popis	Po resete
0x0 – 0x3	Control Word	000...
0x4 – 0x7	Delay register	000...
0x8 – 0xb	Counter	000...

Tabuľka 3.10: Registre časovača

Bit	Popis	Po resete
0	Enabled	0
1	Auto-reload	0

Tabuľka 3.11: Timer Control Word

Využitie Časovač je možné využiť pri v kombinácii s hardvérom, ktorý potrebuje používa polling, na zníženie záťaže systému – virtuálny počítač sa môže vďaka časovaču na istú dobu zastaviť a po prerušení skontrolovať stav zariadenia.

Kapitola 4

Príklady

V tejto kapitole sú popísané námety cvičenia, na ktoré sa dá využiť emulačné jadro. Jednotlivé návrhy majú pomerne široký tematický záber, sú preto rozčlené na menšie časti. Často tiež vyžadujú implementáciu ďalšieho emulovaného hardvéru, čo už presahuje rámec tejto práce.

4.1 I/O

Do tejto časti patria aj príklad uvedené v kapitole 3, v odstavcoch o využití navrhnutého hardvéru.

1. Naprogramujte čítanie sériového portu, ktoré pomocou pollingu prečíta vstupné dáta a vypíše ich na konzolu; to isté zopakujte pre riadenie prerušením.

2. Napíšte program na preposielanie dát medzi dvoma sériovými portami (výstup prvého je vstupom druhého a naopak).

4.2 Správa pamäte

1. Jednoduchá ukážka ošetrenia výpadku stránky v pamäti, ktorá je už implementovaná v ukážkových cvičeniach, `examples/pagefault-master.c`.

2. K dispozícii je proces s pevným množstvom pamäte. Z vonkajšieho prostredia bude dostávať príkazy na alokáciu/dealokáciu pamäte a prípadné čítanie a zápis. Implementujte alokačný algoritmus, a čo najviac znížte pamäťový overhead.

3. Využitie popísané v podčasti 3.1.3.

4.3 Multitasking

1. Implementujte algoritmus vzájomného vylúčenia pomocou činného čakania.

2. Implementujte producer/consumer problém

3. V systéme je zariadenie, ku ktorému potrebujú pristupovať dva procesy, ktoré medzi sebou môžu komunikovať zdieľanou pamäťou; zabezpečte, aby zariadenie používal vždy iba jeden z nich

4. Pomocou nastavovania procesov do stavu `READY` a `STOPPED` sa dá simulovať jednoduchý scheduler, a tiež modelovať problém inverzie priorít.

4.4 Sieť

Pri návrhu cvičení zameraných na sieť je možné využiť to, že virtuálne počítače je možné pomocou správneho správcu pospájať do takmer ľubovoľnej logickej topológie, či simulovať stratovosť liniek/preusporiadanie rámcov.

1. Vytvorte pomocou kontextov sieť typu *ring* s vami zvoleným tvarom rámca; každý kontext má svoje jednoznačné ID. Skúste si po sieti poslať rámec, ktorý obehne celú kružnicu a vráti sa naspäť. Navrhnite, ako čo najjednoduchšie a najrýchlejšie zistiť kontext s najnižším ID.

2. Dva virtuálne počítače komunikujú pomocou správcu, ktorý simuluje spojenie s nenulovou chybovosťou. Navrhnite a implememntujte protokol, ktorý použije čo najmenej správ na prenesenie údajov.

4.5 Systémové programovanie

Jednoduché cvičenia a algoritmy v assembleri, C, prípadne analýza fungovania emulačného jadra, teoreticky alebo pomocou Qemu a gdb.

1. Zmeňte scheduler emulačného jadra tak, aby podporoval priority, a pridajte systémové volanie na ich zmenu

2. Vylepšite mechanizmy emulačného jadra na alokáciu pamäte

3. Rozšírte schopnosti VGA ovládača emulačného jadra o farebný výstup pomocou escape sekvencií (napr. xterm).

4. Implementujte vyvolanie prerušenia vo virtuálnom počítači pomocou systémových volaní `setstate`, `getcpu` a `setcpu`, a manipuláciou so zásobníkom. Čo je potrebné na to, aby mohol program vo virtuálnom počítači po skončení obsluhy prerušenia pokračovať ďalej vo vykonávaní?

Kapitola 5

Záver

V tejto práci sme navrhli a implementovali jednoduchú architektúru použiteľnú na demonštráciu rôznych problémov, ktoré je treba riešiť pri návrhu a činnosti operačných systémov. Pomocou emulačného jadra a virtuálnych počítačov je možné pokryť dostatočne veľké rozpätie tém, hoci na niektoré konkrétne úzke tematické celky existujú elegantnejšie riešenia.

Obsah tejto práce nie je v žiadnom prípade nemenný – samotná práca poskytuje iba jednoduchý základ pre rozširovanie a vytváranie nových modulov. Témy na ďalšie rozširovanie sú napríklad:

- presunutie všetkých nepotrebných ovládačov z jadra do používateľského prostredia
- vytvorenie väčšej sady ukážkových cvičení
- odstránenie platformových závislostí emulačného jadra a pridanie podpory pre ďalšie architektúry
- vylepšenie posielania správ medzi procesmi
- pridanie možnosti používateľskej interakcie s kontextmi

Dodatok A

Implementácia

Ku tejto práci je priložené CD s nasledovným obsahom:

- `kernel.tar.gz` – samotná implementácia
- `grub-sources.tar.gz` zdrojové kódy GRUB2 použité pre tento projekt

V tomto súbore je GIT[12] repozitár obsahujúci všetky revízie implementačnej časti tejto práce. V adresári `grub/` sa nachádzajú binárne súbory GRUB-u spolu s konfiguračným súborom (`grub/grub.cfg`) nastaveným na štartovanie dodávaných príkladov.

Na samotné spustenie je treba mať nainštalovaný emulátor Qemu-kvm[10] (vývoj prebiehal na verzii 0.12.3) alebo Qemu[9]. Štart prebieha pomocou štartovania z internej siete Qemu – pomocou vstavaného TFTP[11] servera sa stiahne emulačné jadro spolu s modulmi a predá sa riadenie jadru. Potrebné parametre pre Qemu sú:

```
qemu -boot n -net nic -net user,tftp=.,bootfile=grub/pxegrub
```

Voliteľne je možné pridať parameter `-serial stdio`, ktorý presmeruje výstup zo sériového portu na konzolu, z ktorej bolo Qemu spustené. Qemu musí byť spúšťané z rovnakého adresára, ako ten, v ktorom je súbor `kimage`, adresár `grub` a ďalšie. Pre uľahčenie spúšťania je v kernel.tar.gz súbor `run.sh`,

ktorý po nastavení cestu ku Qemu spustí emulátor so správnymi parametrami.

A.1 Príklady

v Adresári `examples/` je implementovaných niekoľko ukážkových správcovských programov, ako aj programov pre virtuálny počítač, ktoré sú dostupné cez prednastavené položky štartovacom menu GRUB-u.

A.1.1 Príklad `polling`

V súbore `examples/polling-master.c` je kód, ktorý pri inicializácii vyrobí zdieľanú pamäťovú stránku v adresnom priestore oboch kontextov. V tejto stránke potom simuluje jednosmernú sériovú komunikáciu. Viac informácií nájdete v 3.1.1

A.1.2 Príklad `pagefault`

Kód v `examples/pagefault-master.c` sa pri inicializácii nastaví ako správca virtuálneho počítača pre `examples/pagefault-slave.c`, vďaka čomu bude upozorňovaný na ním spôsobené výnimky. Virtuálny počítač sa po spustení pokúsi vypísať na obrazovku obsah pamäte na adrese `0xa0000000`, ktorá nie je zatiaľ nikam namapovaná. Toto vyvolá výnimku odchytenú správcom, ktorý na túto adresu namapuje predpripravenú stránku s jednoduchým textom a znova spustí virtuálny počítač, ktorý môže pokračovať vo vykonávaní a úspešne vypísať požadovaný reťazec.

A.1.3 Príklad `interrupt`

V `examples/interrupt-slave.c` je program, ktorý by za normálnych podmienok poslal druhému kontextu správu s adresou funkcie `interrupt`, vypísal úvodnú správu a zostal stáť na mieste. V `examples/interrupt-master.c` je ale kód, ktorý druhý kontext po určitom čase zastaví a vyvolá v ňom

prerušenie – na zásobník uloží starú hodnotu registra EIP a nastaví ho na adresu, ktorá mu bola doručená na začiatku. Takto upravený kontext začne vykonávať inštrukcie na zaslanej adrese, a pri výskyte inštrukcie `ret` bude pokračovať vo vykonávaní pôvodného kódu. Rozširujúcou úlohou pre tento príklad môže byť odstránenie inštrukcií `pusha` a `popa` vo funkcii `interrupt` bez zmenenia funkčnosti.

Literatúra

[1] Tanenbaum, Andrew S.; Woodhull Albert S. *Operating Systems: Design and implementation, 3rd Edition*. Prentice Hall, 2006

[2] Nachos 4.0, `http://www.cs.washington.edu/homes/tom/nachos/`

[3] Nachos 5.0j, `http://www.eecs.berkeley.edu/~kubitron/courses/cs162-F05/Nachos/`

[4] Dr. Fankhauser, George; Dr. Ruf, Lukas; *Topsy – A Teachable Operating System*, `http://www.topsy.net/`

[5] Low Level Virtual Machine, `http://www.llvm.org/`

[6] Executable and Linking Format specification, verzia 1.2, `http://refspecs.freestandards.org/elf/elf.pdf`

[7] GRUB 2, `http://www.gnu.org/software/grub/grub-2.en.html`

[8] Multiboot specification, `http://www.gnu.org/software/grub/manual/multiboot/`

[9] Qemu, open source processor emulator, `http://www.qemu.org/`

[10] Kernel Based Virtual Machine, `http://www.linux-kvm.org/page/Main_Page`

[11] The TFTP protocol, `http://tools.ietf.org/html/rfc1350`

[12] Git, The fast version control system, `http://git-scm.com/`