SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY
A INFORMATIKY

Operačné systémy

Poznámky ku prednáškam

Bratislava 2024

Publikované na internetovej stránke Ústavu informatiky a matematiky Fakulty elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave
https://uim.fei.stuba.sk/wp-content/uploads/2018/03/operacne-systemy-2023.pdf

ISBN 978-80-570-5636-2

Operačné systémy

(poznámky ku prednáškam)

Stránky tohto dokumentu predstavujú pracovný text ku prednáškam predmetu Operačné systémy[1] v bakalárskom štúdiu študijného programu Aplikovaná informatika na Fakulte elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave.

Aj keď sa pracovný text hypertextovými odkazmi vzťahuje na prednášky, ktoré odzneli v zimnom semestri akademického roka 2023/2024, je aplikovateľný na štrukúru predmetu, ktorá vychádza z dlhoročno zavedených cvičení predmetu Operating System Engineering[2] na MIT (Massachusetts Institute of Technology).

Široké nasadenie výukového operačného systému xv6 v univerzitnom prostredí[3] je jasným znakom dobrej koncepcie predmetu. Ďakujeme pánovi Fransovi Kaashoekovi za povolenie použiť ich študijné materiály pri príprave predmetu Operačné systémy pre študentov nášho študijného programu. Je nutné poznamenať, že predmet na našej fakulte sa nemôže vyrovnať ani obsahovo, ani rozsahovo svojmu vzoru na MIT. Pri polovičnom počte prednášok sme sa obmedzili na nasledovné témy:

1. systémové volania,
2. organizácia OS,
3. virtuálna pamäť,
4. prechod z používateľského priestoru do jadra a späť,
5. výpadky stránok,
6. vlákna,
7. zámky,
8. súborový systém,
9. obnova súborového systému po zlyhaní a
10. prerušenia.

Dúfame, že texty uvedené v tomto pracovnom zošite pomôžu poslucháčom predmetu sústredenejšie sledovať prednášky a uľahčia zápis prípadných poznámok.

Všetkým aktuálnym aj budúcim študentom predmetu Operačné systémy prajeme jeho úspešné absolvovanie.

Château de la Mûre
18. decembra A. D. MMXXIII

1 https://uim.fei.stuba.sk/predmet/b-os/
2 https://pdos.csail.mit.edu/6.828
3 Viď časť “Educational use” na https://en.wikipedia.org/wiki/Xv6

Prednáška 1

Úvod do predmetu Operačné systémy

Obsah prednášky

- 1. motivácia
- 2. štruktúra predmetu
- 3. úvod do os
- 4. systémové volania

1. časť: MOTIVÁCIA

Ciele predmetu

- Detailnejšie pochopenie OS
- Ako:
  - Návrh
  - Implementácia
- malilinkatý OS ;)

Načo je dobrý OS

- Aplikácie (izolácia ↔ zdieľanie)
- Služby (hw → app, spravodlivosť)
- Hardvér (výkon)

Čo chcú aplikácie od OS

- Prístup k hardvéru, čo vyžaduje:
  - Abstrakciu hardvéru (problém ovládačov)
  - Multiplexovanie hw medzi aplikáciami
- Izolovanie poškodených aplikácií od zvyšku OS
- Komunikáciu medzi sebou

Aké služby od OS očakávame

- Procesy
- Pamäť
- Prístup ku súborom
- Adresárová štruktúra
- Bezpečnosť
- Sieť
- Viacerí používatelia
- Komunikácia medzi aplikáciami (IPC)
- …

Čo znamená abstrakcia hw?

- Prístup ku hw spravovaný jadrom na základe žiadosti aplikácie
- Jednotné rozhranie k rôznym zariadeniam toho istého typu (zbernica)
- Programátor ovládača sa môže sústrediť iba na jednu časť OS

Ako vyzerá abstrakcia OS?

- Aplikácie využívajú služby jadra cez tzv. systémové volania (*syscalls*)

- Príklad zo systémov typu UNIX:

```
fd = open("subor.txt",1);
write(fd, "ahoj svjete!\n", 13);
pid = fork();
```


Prečo sa venovať takejto oblasti?

- Skĺbenie vedomostí: PROG, PT, OOP
- Cieľom je vidieť reálny program (jadro), ktorý:
  - Musí byť efektívny (rýchlosť), ale na druhej strane dostatočne abstraktný (prenositeľný na iné platformy)
  - Musí byť výkonný (veľa služieb a možností), ale pritom jednoduchý (zložený z ľahko vymeniteľných blokov)

Prečo sa venovať takejto oblasti?

- Naučiť sa myslieť, správne navrhovať a implementovať algoritmy, ktoré medzi sebou spolupracujú

- Využiť všetky doteraz získané vedomosti a skĺbiť ich

- OS je azda najkomplexnejší program vôbec

Linux 4.8-rc7 19.9.2016

```
y@ellYah:/usr/src/linux-4.8-rc7$ cloc .
   55460 text files.
   54971 unique files.
    9923 files ignored.

http://cloc.sourceforge.net v 1.56  T=192.0 s (235.7 files/s, 106196.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                            23425        2258325        2074816       11503172
C/C++ Header                 17771         433230         741250        2835111
Assembly                      1433          48266          63102         294019
make                          2217           8254           7915          34510
Perl                            49           5183           3760          27188
Bourne Shell                   201           2199           3513          11559
Python                          44           1579           1917           9262
yacc                             8            655            355           4327
HTML                             3            512              0           4289
lex                              8            299            289           1894
ASP.Net                         19            133              0           1615
Bourne Again Shell              46            355            260           1585
C++                              1            231             58           1581
awk                             10            132            131           1138
NAnt scripts                     2            128              0            475
Pascal                           3             49              0            231
Lisp                             1             63              0            218
Objective C++                    1             55              0            189
m4                               1             15              1             95
XSLT                             6             13             27             71
vim script                       1              3             12             27
CSS                              1             12             22             25
sed                              2              0             25             16
Teamcenter def                   1              0              2              6
-------------------------------------------------------------------------------
SUM:                         45254        2759691        2897455       14732603
-------------------------------------------------------------------------------
```

Linux 4.13.1 10.9.2017

```
y@ellYah:/usr/src/linux-4.13.1$ cloc .
   60545 text files.
   60002 unique files.
   11247 files ignored.

http://cloc.sourceforge.net v 1.56  T=211.0 s (232.3 files/s, 108178.4 lines/s)
```

Language	files	blank	comment	code
C	25241	2468666	2227885	12531579
C/C++ Header	19482	484063	907454	3641969
Assembly	1434	49263	64821	298076
make	2362	8582	8151	36990
Perl	50	5061	3707	26293
Bourne Shell	246	3091	4180	15853
Python	72	2099	2426	12018
HTML	3	565	0	4730
yacc	9	682	357	4530
lex	8	302	301	1906
C++	7	287	71	1838
ASP.Net	32	158	0	1808
Bourne Again Shell	47	384	312	1713
awk	12	185	170	1510
NAnt scripts	2	158	0	588
Pascal	3	49	0	231
Objective C++	1	55	0	189
m4	1	15	1	95
XSLT	5	13	26	61
CSS	1	14	23	35
vim script	1	3	12	27
Teamcenter def	1	0	2	6
sed	1	2	5	5
SUM:	49021	3023697	3219904	16582050

Linux 5.3 16.9.2019

```
y@ellYah:/mnt/data1/skola/os/2019/prednasky/01/linux-5.3$ cloc .
   65218 text files.
   64765 unique files.
   13172 files ignored.

github.com/AlDanial/cloc v 1.70  T=227.84 s (228.6 files/s, 110043.8 lines/s)
```

Language	files	blank	comment	code
C	27407	2706938	2262021	13772088
C/C++ Header	19656	523220	932408	4108616
Assembly	1327	47356	101761	230264
JSON	246	1	0	154515
Bourne Shell	543	11429	8786	45784
make	2508	9291	10602	40883
Perl	56	5590	4069	27896
Python	108	4374	4082	23575
YAML	115	2076	659	11118
HTML	5	656	0	5446
yacc	9	698	361	4627
lex	8	332	306	2013
C++	8	300	82	1871
Bourne Again Shell	51	354	296	1748
awk	11	171	148	1387
NAnt script	2	145	0	545
Markdown	2	133	0	423
Windows Module Definition	2	15	0	109
m4	1	15	1	95
CSS	1	27	28	72
XSLT	5	13	26	61
vim script	1	3	12	27
Ruby	1	4	0	25
INI	1	1	0	6
sed	1	2	5	5
SUM:	52075	3313144	3325653	18433199

Linux 4.19-rc3 10.9.2018

```
y@ellYah:/mnt/data1/skola/os/2018/__prednasky/01/linux-4.19-rc3$ cloc .
   61684 text files.
   61262 unique files.
   12218 files ignored.

http://cloc.sourceforge.net v 1.56  T=225.0 s (219.2 files/s, 104909.5 lines/s)
```

Language	files	blank	comment	code
C	26078	2586063	2268480	13128894
C/C++ Header	18861	492135	903746	3659127
Assembly	1323	47331	61174	278057
make	2386	8738	9443	37957
Bourne Shell	376	6571	5894	28200
Perl	55	5426	4004	27407
Python	102	2775	2975	15979
HTML	5	670	0	5497
yacc	9	701	375	4648
lex	8	326	315	2006
ASP.Net	38	191	0	1981
C++	7	286	77	1844
Bourne Again Shell	51	352	318	1722
awk	11	170	155	1386
NAnt scripts	2	155	0	588
Teamcenter def	2	14	2	100
m4	1	15	1	95
XSLT	5	13	26	61
CSS	1	18	27	44
vim script	1	3	12	27
Ruby	1	4	0	25
sed	1	2	5	5
SUM:	49324	3151959	3257029	17195650

Linux 5.9-rc6 21.9.2020

```
y@ellYah:~/Downloads/linux$ cloc .
   69961 text files.
   69491 unique files.
   13491 files ignored.

github.com/AlDanial/cloc v 1.70  T=244.08 s (231.5 files/s, 111430.5 lines/s)
```

Language	files	blank	comment	code
C	29197	2899589	2366428	14779106
C/C++ Header	21121	588338	1022657	4640386
Assembly	1275	45959	98965	225130
JSON	287	0	0	165799
YAML	1088	18476	4963	83572
Bourne Shell	657	16812	11349	65983
make	2592	9806	10864	44176
Perl	60	6686	4767	34393
Python	123	5345	4606	27918
yacc	9	695	354	4755
lex	9	349	304	2130
C++	8	326	87	1934
Bourne Again Shell	52	338	297	1750
awk	10	139	116	1051
NAnt script	2	143	0	549
Windows Module Definition	2	15	0	109
m4	1	15	1	95
CSS	1	28	29	80
XSLT	5	13	26	61
vim script	1	3	12	27
Ruby	1	4	0	25
INI	1	1	0	6
sed	1	2	5	5
SUM:	56503	3593082	3525830	20079040


```
y@ellYah:~/Downloads/linux$
```

Linux 5.15-rc2 19.9.2021

```
y@ellYah:/usr/src/linux$ cloc .
   73560 text files.
   73073 unique files.
   10743 files ignored.

github.com/AlDanial/cloc v 1.86  T=70.66 s (889.5 files/s, 424910.6 lines/s)
```

Language	files	blank	comment	code
C	30406	3062423	2455985	15622292
C/C++ Header	22019	622613	1125991	5432335
reStructuredText	2956	147017	59060	404328
Assembly	1285	45995	98168	223163
JSON	357	2	0	183332
YAML	2000	35042	9129	158071
Bourne Shell	750	21692	14969	84887
make	2657	10199	11267	45992
SVG	59	78	1159	37555
Perl	65	7103	4967	35850
Python	139	5756	5442	29318
yacc	9	693	355	4761
PO File	5	791	918	3077
lex	9	345	303	2104
C++	10	349	138	1935
Bourne Again Shell	51	297	247	1304
awk	11	155	126	1111
⋮	⋮	⋮	⋮	⋮
SUM:	62849	3960906	3788347	22273460

Linux 6.6-rc2 18.9.2023

```
y@ellYah:/usr/src/linux$ cloc .
   81709 text files.
   81138 unique files.
   11479 files ignored.

github.com/AlDanial/cloc v 1.86  T=90.32 s (777.9 files/s, 386428.3 lines/s)
```

Language	files	blank	comment	code
C	32958	3360500	2624829	17352062
C/C++ Header	23808	716438	1379822	7133514
reStructuredText	3381	167497	69513	457456
JSON	585	2	0	387710
YAML	3715	68503	17204	323240
Assembly	1339	48724	102041	235663
Bourne Shell	986	30888	21035	121001
make	2924	11199	12126	52086
SVG	74	90	1171	48177
Python	197	9595	8399	47955
Perl	69	7569	5474	37706
Rust	60	1409	9001	8623
yacc	10	710	412	4953
PO File	6	948	1088	3733
lex	10	360	312	2202
C++	6	335	126	1788
Bourne Again Shell	57	392	310	1619
awk	12	195	118	1076
⋮	⋮	⋮	⋮	⋮
SUM:	70257	4425759	4252968	26222818

Linux 6.0-rc6 19.9.2022

```
y@ellYah:/usr/src/linux$ cloc .
   77966 text files.
   77409 unique files.
   11105 files ignored.

github.com/AlDanial/cloc v 1.86  T=83.37 s (802.3 files/s, 398018.7 lines/s)
```

Language	files	blank	comment	code
C	31796	3220241	2543177	16507334
C/C++ Header	23209	696041	1323006	6827231
reStructuredText	3201	155905	62456	426890
JSON	452	2	0	371620
YAML	2870	51988	13143	237709
Assembly	1311	46799	99794	226041
Bourne Shell	852	25874	17642	100685
make	2754	10628	11629	48591
SVG	65	81	1162	41845
Perl	66	7373	5026	36470
Python	151	6426	6169	31958
yacc	9	698	409	4912
PO File	5	791	918	3077
lex	9	346	309	2110
C++	10	372	138	2016
Bourne Again Shell	54	355	314	1476
awk	13	217	157	1323
⋮	⋮	⋮	⋮	⋮
SUM:	66893	4224522	4085691	24873470

S čím sa potýkame v B-OS? xv6

```
y@ellYah:/usr/src/xv6-riscv/kernel$ cloc .
      46 text files.
      46 unique files.
       1 file ignored.

github.com/AlDanial/cloc v 1.86  T=0.02 s (2062.6 files/s, 291928.7 lines/s)
```

Language	files	blank	comment	code
C	24	691	806	3484
C/C++ Header	17	139	166	758
Assembly	4	32	78	215
SUM:	45	862	1050	4457

2. časť: ŠTRUKTÚRA PREDMETU

Štruktúra predmetu

- Stránka: https://uim.fei.stuba.sk/predmet/b-os
- Kurz podľa MIT: https://pdos.csail.mit.edu/6.828

- Práca na predmete:
  - Priebežné programovanie úloh počas semestra
- Práca na doma (DU) pozostáva z 2 častí:
  - Príprava na ďalšie cvičenie podľa pokynov
  - Vypracovanie úloh pre daný týždeň

Štruktúra predmetu

- Prednášky:
  - Základná teória OS
  - Pohľad na implementáciu systému xv6

- Cvičenia:
  - Systémové programovanie (systémové volania)
  - Základné veci OS (napr. správa pamäte)
  - Dopĺňanie funkcionality do xv6

Pracovné prostredie

- Virtuálka (odporúčame VirtualBox)
- Debian (alebo Ubuntu alebo iná distro)
- Qemu (vlastné zostavenie vítané)
- Xv6 (cieľové prostredie)

- Vlastné stroje! (alebo nejaké v prenájme)

Hodnotenie (1)

- Akýkoľvek identifikovaný pokus o podvod:
  - Disciplinárna komisia FEI STU
  - FX hodnotenie z predmetu
- Nutná (**nie postačujúca**) podmienka získania hodnotenia lepšieho než FX
  - Vyplnenie evaluácie
  - Kontrola vyplnenia evaluácie môže byť (individuálne) na konci semestra

3. časť: ÚVOD DO OS

Očakávania od OS

- Podpora viacerých vecí SÚČASNE
- Zdieľanie a prerozdeľovanie zdrojov hw (CPU pre procesy, pamäť, disk, tlačiareň, myš, monitor, sieťová karta...)
- Izolácia procesov (aby nemohol jeden ničiť druhý len tak, z čírej zloby)
- Komunikácia procesov je však tiež potrebná

Požiadavky na OS

1) Multiplex

2) Izolovanie

3) Interakcia

Prečo multiplex?

- Čo keby sme nemali OS, ale každá aplikácia by pomocou nejakej knižnice priamo pristupovala k hw, ktorý potrebuje?
- Viď niektoré vnorené *(embedded)* aplikácie

Prečo multiplex?

- Ak by bola aplikácia jediná, OK
- Ak je ich viac, musela by byť každá „slušná“, museli by spolupracovať (kooperatívny multitasking, dnes často označovaný ako „asynchrónne programovanie“)
- Aplikácie sú často plné chýb… a nie sú vždy slušné!

Prečo izolácia?

- Aby sa zachovala „spravodlivosť“ pri prideľovaní hw, oddelíme aplikácie
- Prístup aplikácií nie k hw, ale k abstrakcii hw cez službu OS (príklad: prístup na disk cez systém. volania `open()`, `read()`, `write()`, `close()`)
- Výhody:
  - Abstraktnejší prístup aplikácie (nemusí používať čísla sektorov, ale mená súborov)
  - Nepríde ku chybe pri práci s diskom

Prečo interakcia?

- Ak máme izolovaný beh aplikácie, ako by medzi sebou mohli priamo komunikovať?
- Nepriamo cez služby OS:
  - Súborové popisovače (tzv. *deskriptory*)
  - Mapovanie pamäte
  - Sieť

Používateľský priestor

- Silná izolácia si vyžaduje presne definované API (prístupové body k službám jadra)
- Aplikácia by nemala byť schopná pristupovať k interným údajovým štruktúram a inštrukciám jadra
- CPU poskytuje hw podporu takejto ochrany

User / Kernel space

- Moderné CPU poskytujú minimálne 2 režimy činnosti:
  - **Kernel** mód (všetky inštrukcie CPU povolené), napríklad priamy prístup k zariadeniam
  - **User** mód (nie všetky inštrukcie CPU povolené); ak aplikácia v tomto móde skúsi vykonať privilegovanú inštrukciu, vyvolá sa výnimka a OS má možnosť „prehovoriť do duše“ tejto apke. Zväčša tým, že zneutralizuje jej existenciu ;)

User / Kernel space

Aplikácia bežiaca v režime „user mód“ beží v tzv. priestore používateľa (**„user space“**)

Program bežiaci v režime CPU „kernel mód“ sa vykonáva v tzv. priestore jadra (**„kernel space“**)

V prípade OS sa takémuto programu vraví „jadro“, angl. *„kernel“*

Čo všetko má bežať v jadre?

- Všetky sys. volania? → **monolitické jadro**
  - Všetko potrebné v jadre
  - Rôzne časti jadra môžu priamo komunikovať (napr. spoločný *buffer* pre virtuálnu pamäť a súborový systém)
  - Problém je práve to, že všetko je v jadre a všetko môže/chce komunikovať so všetkým → komplexné rozhrania, šírenie chýb naprieč celým jadrom
  - V prípade chyby hrozí pád celého jadra a tým aj systému, vyžaduje sa reštart počítača

Čo všetko má bežať v jadre?

- Minimálna funkcionalita → **mikrokernel**
  - Služby OS bežia v priestore používateľa, vtedy sa takéto aplikácie volajú *servery*
  - Na využívanie služieb serverov je definované rozhranie, tzv. *posielanie správ* serverom
  - Funkcionalita mikrojadra: spúšťanie aplikácií, posielanie správ, sprístupňovanie hw

Príklady z praxe

- Linux
  - Monolitické jadro
  - Niektoré služby v user space (grafický subsystém)
- Xv6
  - Monolitické jadro
  - Tak málo služieb, že jadro je menšie ako niektoré mikrojadrá ;)

4. časť: Systémové volania

Systémové volania

- Príklady systémových volaní ukázané v xv6
- Xv6 má tradičný návrh podľa UNIX, ale úplne jednoduchý; **nutné preštudovať** a porozumieť knihe o xv6 (ako a prečo)

Systémové volania UNIX

- Príklady systémových volaní ukázané v xv6
- Xv6 má tradičný návrh podľa UNIX, ale úplne jednoduchý; **nutné preštudovať** a porozumieť knihe o xv6 (ako a prečo)

- Prečo UNIX?
  - Dobrá dokumentácia
  - Jednoduchý návrh
  - Široké nasadenie

Systémové volania UNIX

- Xv6 beží na architektúre RISC
- Preto ho treba spúšťať pomocou virtualizácie nástrojom Qemu

- (ukážka spustenia: `make qemu`)

SV read/write

- Prvý príklad systémových volaní: `copy.c`
- Kopíruje vstup na výstup ;)
- Napísaný v jazyku C
- Využíva 2 systémové volania: `read` a `write`

SV read/write

- Prvý argument je FD (angl. *file deskriptor*)
  - Informuje jadro, o ktorý súbor sa jedná
  - Súbor, na ktorý odkazuje, musí byť už otvorený!!!
  - V systéme UNIX je takmer „všetko“ súbor (súbor, myš, disk, adresár, sieťový rámec, …)
  - Proces (čo to je?) môže mať otvorených veľa súborov, používať veľa deskriptorov

  - !!! **FD 0** *„standard input“*, **FD 1** *„standard output“*, **FD 2** *„standard error output“* !!!

SV read/write

- Druhý argument je pamäťové miesto
  - `read()`: kam sa uložia údaje načítané **ZO** súboru
  - `write()`: skadiaľ sa zoberú údaje pri zápise **DO** súboru
- Tretí argument je veľkosť (počet bajtov!!!)
  - Pri čítaní (`read`) možno načítať menej, ale nie viac!
  - Podobne pri zápise (`write`)

SV read/write

- Návratová hodnota systémového volania `read`/`write`: počet načítaných/zapísaných bajtov
- `read()` a `write()` neriešia formát údajov; UNIX I/O sú v binárnej forme reťazce bajtov (1B = 8b)
- Interpretácia prúdu bajtov ostáva na aplikácii

SV read/write

(ukážka behu aplikácie a kód aplikácie `copy.c`)

SV open

- Skadiaľ sa berú súborové popisovače (ako sa v používateľskom programe získa/nastaví hodnota premennej `fd`)?

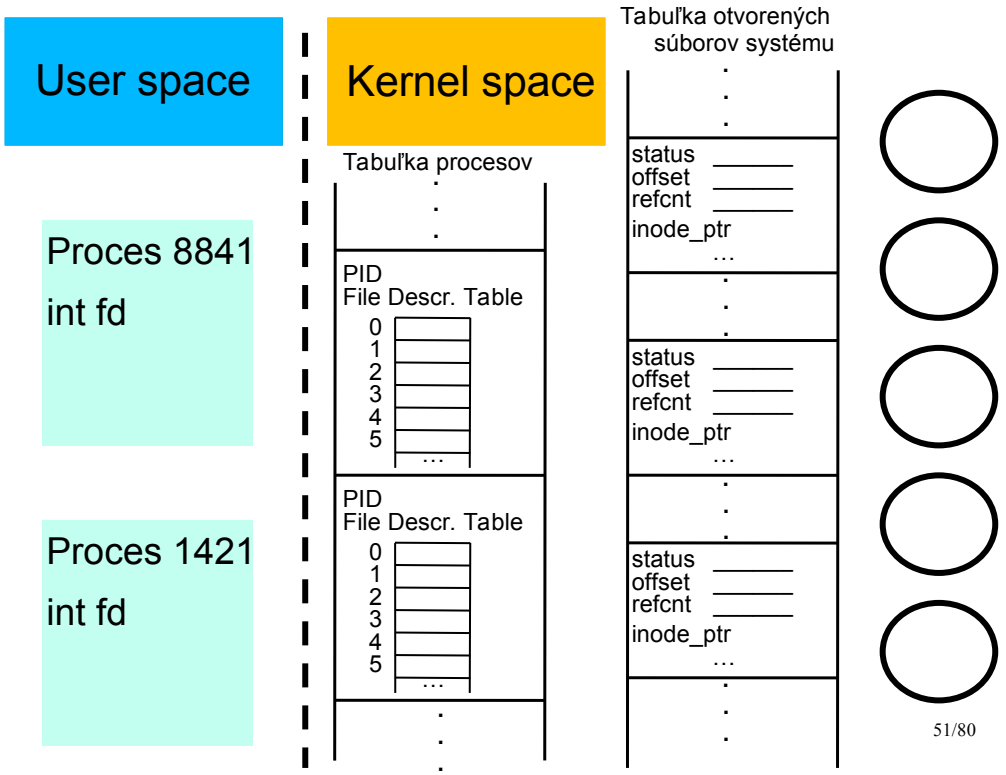
SV open

- Príklad: `open.c`, vytvorenie súboru
  - `./open`
  - `./cat output.txt`
- FD je malé celé číslo, štartuje sa od 0
- FD je indexom do tabuľky procesu (?) (o túto tabuľku sa stará jadro OS)
- Každý proces má vlastnú tabuľku! (vo všeobecnosti `fd 3` ukazuje na rozličné súbory v rôznych procesoch)

SV open

- Pozor na kontrolu chýb!!!
- Príkladíky na prednáške túto kontrolu neobsahujú (pre jednoduchosť a názornosť jadra funkcionality)

- `man 2 open`
- Príkladíky `open.c`, `cat.c`
- Obrázok 1.2 v knižke xv6 na str. 11



SV open

- Čo všetko sa udeje pri vyvolaní systémového volania (v našom prípade `open`)?

- Z pohľadu používateľa ide o akoby funkciu, ale v podstate sa jedná o špeciálnu inštrukciu procesora s nejakými parametrami

SV open

- Hw uchová registre procesora
- Hw zvýši úroveň oprávnení procesora
- Hw zabezpečí vyvolanie vstupného bodu do jadra OS
- Vyvolá sa funkcia na spracovanie `open()` (môže to chvíľku trvať – práca s diskom, aktualizácia dát jadra ako tabuľka FD, vyrovnávacia pamäť)
- Obnovia sa registre procesora pre proces
- Zníži sa úroveň oprávnení procesora
- Obnoví sa beh procesu v používateľskom režime

SV open

- Viď inštrukcia `ecall` v *.asm súboroch priečinka `user/`
- Napríklad `open.asm`

Do **do prvého cvičenia PREČÍTAŤ** kapitolu 1 z knižky:

https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

SV fork

- Zrod nového procesu
- Čo je to `shell`?
- Program (utilitka) príkazového riadku určená na komunikáciu s OS
- Veľmi veľa systémových volaní je dosiahnuteľných (vykonateľných) pomocou tohto programu
- Jedná sa o pôvodné rozhranie s OS pre systém UNIX

SV fork

- Pre každý príkaz napísaný v programe `shell` sa vytvorí nový proces (napr. `echo ahoj`)
- Ako? Pomocou systémového volania „vidlička“
- Prečo vidlička? Lebo sa proces rozdvojí! Skopírujú sa inštrukcie, údaje, registre, tabuľka FD, ostatné interné údaje o procese…
- Bude jestvovať iba (takmer) jediná odlišnosť, a to PID; novo vytvorený proces dostane nový identifikátor v OS

SV fork

- Ako sa potom odlíšia rodič a dieťa, ako vôbec začne potomok „žiť“? Ako sa „narodí“?
- Zvláštnosťou systémového volania `fork` je to, že jedenkrát je vyvolané (rodič vyvolá „funkciu“ `fork()`), ale dvakrát sa vykoná návrat z jadra OS do *user space* !!!!!!!!!!!!!
- Raz pre rodiča, druhý raz pre potomka
- Ako ich odlíšime, keď vykonávajú rovnaký kód (keďže sa potomkovi skopíroval kód rodiča?)

SV fork

- Odlišuje ich návratová hodnota volania `fork()`
- `<0` znamená chybu
- `>0` znamená rodiča
- `=0` znamená potomka

```
pid = fork()
```

`if (pid == 0)` → dieťa

`if (pid > 0)` → rodič

SV fork

(ukážka `fork.c` a `forkwait.c`)

SV fork

- Pomocou `fork()` vieme zduplikovať proces programu `shell`... ale ako spustíme nejaký program?
- Ako spustiť aplikáciu, ktorá má iný programový kód než rodičovský proces?

SV exec

- Nahradenie aktuálneho procesu iným zo súboru na disku

SV exec

- Ako `shell` spustí príkaz '`echo ahoj`'?

SV exec

- Ako `shell` spustí príkaz '`echo ahoj`'?
- Program `echo` je uložený niekde na disku (inštrukcie programu spolu s inicializovanými údajmi od prekladača a linkovacieho programu)

SV exec

- Volanie `exec()` nahradí v pamäti (RAM) aktuálny proces inštrukciami nového procesu; tieto inštrukcie sa načítajú z disku; ako sa to deje?
  - Zrušia sa inštrukcie a dáta aktuálneho procesu, ktorý vyvolal `exec()`
  - Nahrajú sa do pamäte RAM inštrukcie a dáta novo spúšťaného procesu z disku
  - Zachovajú sa niektoré interné údaje patriace k procesu, ktorý vyvolal `exec()` (napr. tabuľka FD)

SV exec

- `exec(prog_na_disku, argum_programu)`
- `argum_programu` sa odovzdajú funkcii `main()`
- Viď `echo.c` alebo `cat.c`, ako sa argumenty používajú v programe

SV exec

- Príklad `forkexec.c`
  - `fork()` vytvorí kópiu rodiča
  - `exec()` sa spustí v potomkovi
  - `wait()` v rodičovi čaká na ukončenie potomka
- `shell` vykonáva sekvenciu *fork-exec-wait* pre každý zadaný príkaz (`wait` sa môže vynechať, ak chceme spúšťať príkaz „na pozadí“, t. j. asynchrónne)

Príklad redirect.c

- Presmerovanie výstupu programu do súboru na disku (miesto vypísania na monitore)
- Čo sa deje pri ‘`echo ahoj > output.txt`’?
  - `fork` v rodičovi (`shell`); `wait` v rodičovi
  - **Zmena FD 1**, `exec ‘echo’` v potomkovi

Príklad redirect.c

- Opakovanie: fd 0, fd 1, fd 2 !!!!!!
- Poučenie: SV `open()` použije vždy **NAJMENŠÍ** voľný (t. j. aktuálne nepoužitý) index do tabuľky FD !!!!!!!

Príklad redirect.c

- Dôsledok 1:
  - Ak chceme nahradiť `fd` 1 naším súborom `output.txt`, musíme vykonať operácie v tomto poradí (za predpokladu otvoreného `fd` 0):
    1. `close(1)`
    2. `open('output.txt')`
- Dôsledok 2:
  - Program `echo` nič „netuší“ o presmerovaní!!!!!!
  - Všetko sa deje na úrovni programu `shell`

Príklad redirect.c

- `echo` pracuje s deskriptorom 1: zapisuje vždy do `fd` 1
- Vďaka zachovaniu tabuľky FD pri SV `exec()` vieme vymeniť to, kam „ukazuje“ FD 1; `echo` stále bude používať na výstup FD 1, ale v skutočnosti pôjde výstup do súboru

Príklad redirect.c

(príklad `redirect.c`, `echo.c`)

Komunikácia medzi procesmi

Komunikácia medzi procesmi

- Pomocou rúry: má 2 konce; jeden na čítanie, druhý na zápis
- Ako je rúra „vyrobená“ pomocou programu `shell`?

```
$ ls | grep x
```


Komunikácia medzi procesmi

- Ak chcem použiť rúru na komunikáciu medzi procesmi (na čo iné by sme ju použili, že?), musíme správne skombinovať SV `fork()` a `pipe()`
- Príklad ‘`ls | grep x`’:
  - Shell vytvorí rúru
  - Shell urobí `fork` (2x!!!! Jeden pre `ls`, druhý pre `grep`)
  - Nahradí `fd` 1 vo forku pre `ls` za zapis. koniec rúry
  - Nahradí `fd` 0 vo forku pre `grep` za čítací koniec rúry
  - Vykoná `exec` pre `ls` a `grep`

Komunikácia medzi procesmi

- Pomocou systémového volania `pipe()`
- Vytvorí sa pár FD (pri `open` iba jeden FD, pri `pipe` dva FD) !!!!!
  - Číta sa z prvého
  - Zapisuje sa do druhého

(príklad `pipe1.c`)

Komunikácia medzi procesmi

- Obrázok prepojenia '`ls | grep x`'

(ukážka `pipe2.c`)

Čítanie obsahu priečinka

- Ako `ls` získa zoznam súborov (položiek) adresára?
- Adresár je iba špeciálny typ súboru; vieme ho otvoriť a čítať

(viď príklady `list.c`, `ls.c`)

Čítanie obsahu priečinka

PREČÍTAŤ kapitolu **1** z knižky:

https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Prednáška 2

Organizácia OS

Štruktúra prednášky

Úvod do OS

Štart prvého procesu xv6

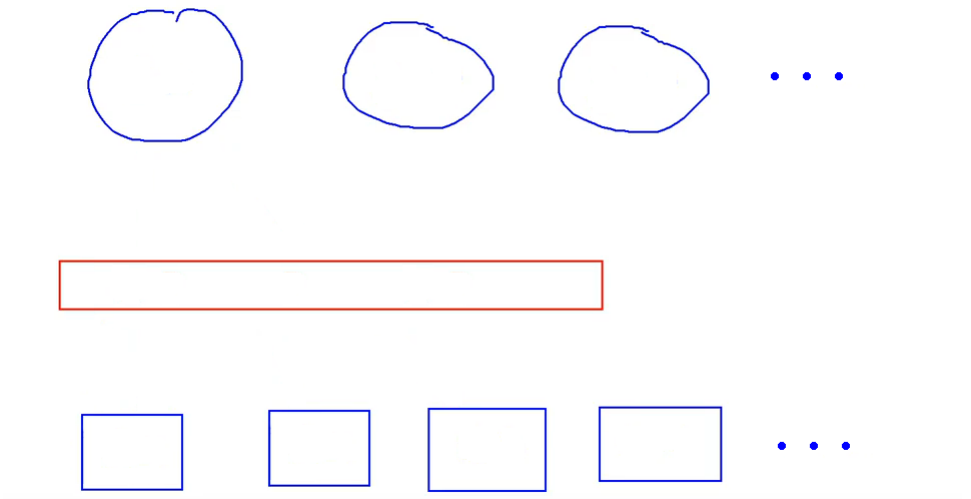
Izolácia procesov

- Systémové volania
- Virtuálna pamäť
  - Segmentácia
  - Stránkovanie

Opakovanie

- Načo slúži OS?

Opakovanie



Opakovanie

- Ciele OS

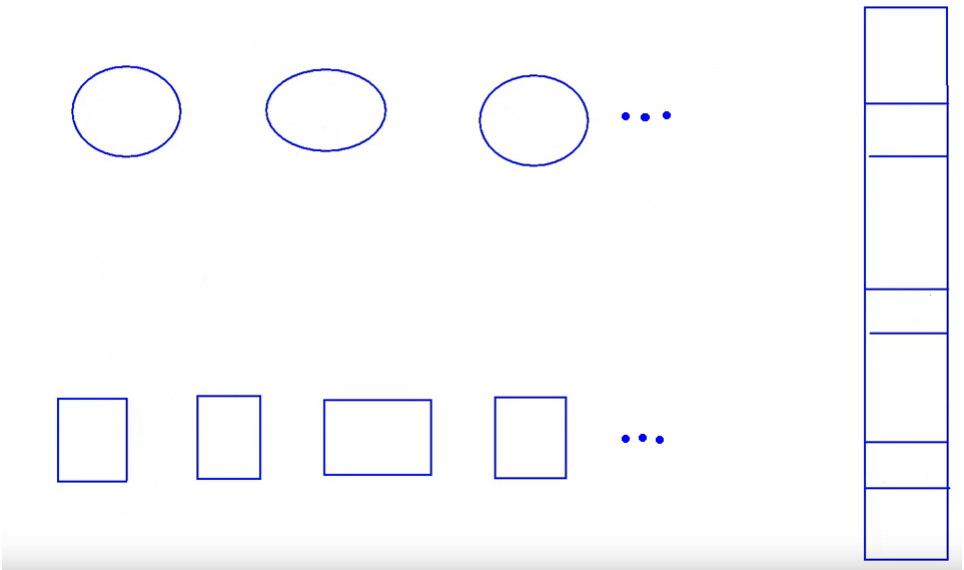
1. Umožniť beh viacerým aplikáciám (súčasne)
2. Izolovať aplikácie
3. Umožniť aplikáciám komunikovať
4. Efektívne využívať hardvér

Môžu bežať app bez OS?

- Obrázok 2

- Aplikácie priamo interagujú s hw
  - CPU
  - RAM
  - HDD
  - …

Môžu bežať app bez OS?



Môžu bežať app bez OS?

- Problém multiplexingu

- Aplikácia sa sama musí vzdať CPU
  - Ak to programátor zabudne urobiť, iná app sa k CPU nedostane
  - Ak sa app dostane do nekonečného cyklu, iná app sa už k CPU nedostane
  - Nie je možné ukončiť beh inej aplikácie (`kill`)
- Tento prístup sa používa pri OS reálneho času (*kooperatívne* plánovanie procesov)

Môžu bežať app bez OS?

- Problém vzájomného prístupu do pamäte
- Nejestvuje izolácia, všetky aplikácie majú priamy prístup do pamäte
- Aplikácia môže prepísať údaje inej aplikácie
- Aplikácia môže prepísať kód zdieľanej knižnice pre všetky aplikácie
- Každá aplikácia má prístup ku všetkým údajom iných aplikácií

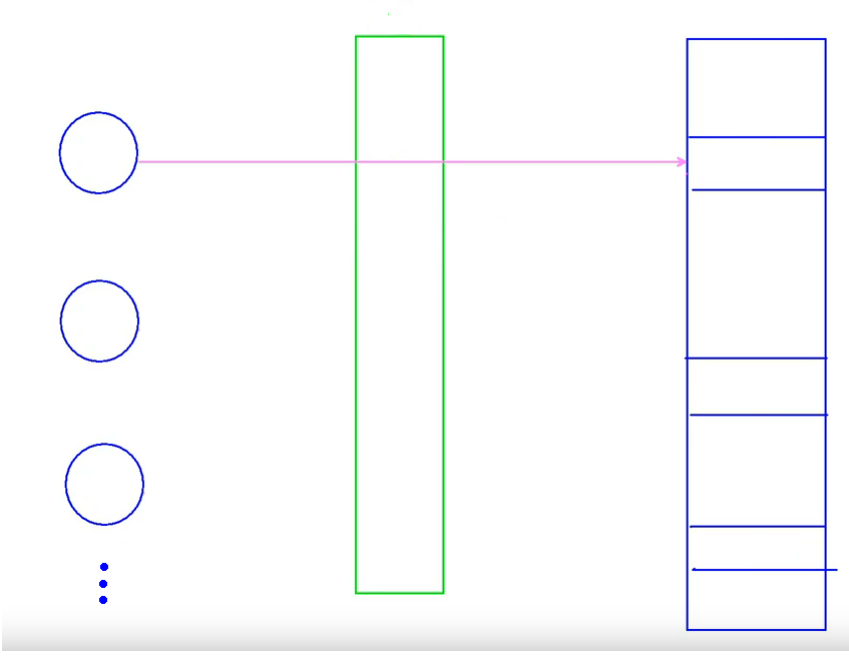
Rozhranie UNIX-like OS

- OS vytvára abstrakciu hardvéru
- Procesy ⟷ jadrá CPU (`fork`)
- Pamäť ⟷ fyzická pamäť (`exec`)
- Súbory ⟷ diskové bloky (`open`, `read`, …)
- Rúry ⟷ zdieľaná fyzická pamäť (`pipe`)
- …

Rozhranie UNIX-like OS

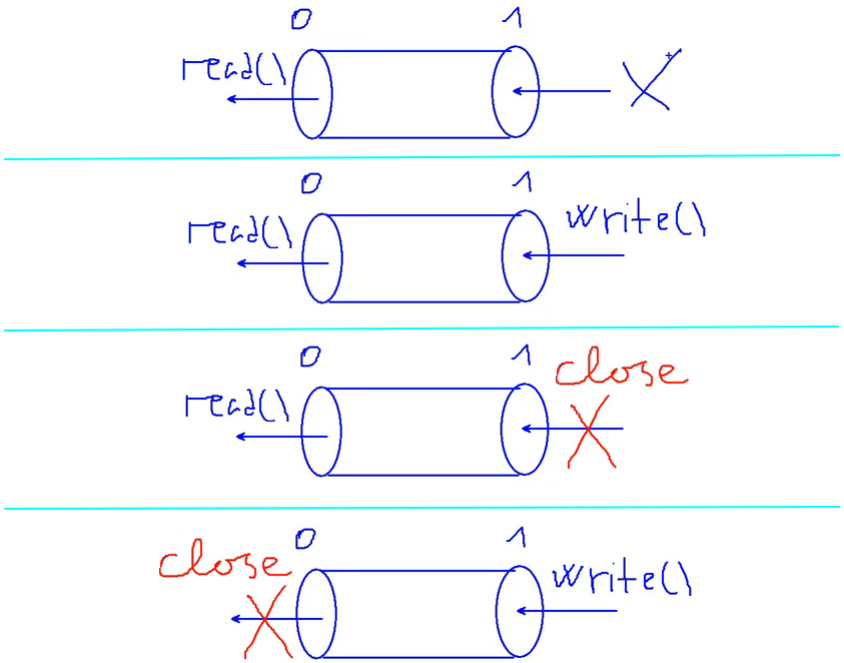
- Procesy ⟷ jadrá CPU (`fork`)
  - OS obsadzuje jadrá pre beh procesov (ukladá a obnovuje registre)
  - OS vynucuje výmenu procesov na CPU
- Pamäť ⟷ fyzická pamäť (`exec`)
  - Každý proces má svoju „vlastnú” pamäť (obrázok 3)
  - OS rozhoduje, kam sa do ramky umiestni app

Rozhranie UNIX-like OS



Rozhranie UNIX-like OS

- Súbory ⟷ diskové bloky (`open`, `read`, ...)
  - OS poskytuje pohodlné používanie mien súborov a adresárovú štruktúru
  - OS môže umožniť zdieľanie súborov medzi procesmi/používateľmi systému
- Rúry ⟷ zdieľaná fyzická pamäť (`pipe`)
  - OS riadi synchronizáciu prenosu (ak je rúra prázdna, čitateľ musí čakať) (obrázok 4)
  - OS môže kedykoľvek ukončiť činnosť čitateľa či zapisovateľa



Rola OS

- OS musí byť defenzívny
- Aplikácia by nemala byť schopná spôsobiť pád OS
- Aplikácia by nemala byť schopná preraziť bariéru izolácie a zasahovať priamo do inej aplikácie

Ako?

- Keďže aj OS aj aplikácie sú sw, väčšinou sa to robí pomocou hw

1. Hw podpora rôznych úrovní vykonávania inštrukcií na CPU (user / kernel mód)
2. Hw podpora virtuálnej pamäte (vytvorenie zdania „vlastnej” pamäte pre proces)

User / Kernel mód

- Už v minulej prednáške sme spomínali, že *kernel* mód znamená možnosť vykonávania všetkých inštrukcií, aj tzv. *privilegovaných*
  - napr. nastavenie módu procesora
  - alebo priamy prístup k hw
- OS beží v *kernel* móde, procesy používateľa v *user* móde

User / Kernel mód

- V *user* móde nie je možné privilegované inštrukcie vykonať
- Pri pokuse sa vyvolá hw výnimka, ktorú má možnosť obslúžiť programový kód v *kernel* režime

Virtuálna pamäť

- SW používa adresy (smerníky, angl. *pointers*)
  - Nazývame ich virtuálne
  - Platí to ako pre OS, aj pre aplikácie používateľa
- HW poskytuje tzv. tabuľky stránok, pomocou ktorých sa robí (hardvérovo!) preklad virtuálnej adresy na fyzickú (napr. do ramky)

Virtuálna pamäť

- OS nastavuje tieto tabuľky pre proces tak, aby nemal prístup k dátam iného procesu
- Tabuľky určujú, do ktorej fyzickej pamäte má tá-ktorá aplikácia prístup

Komunikácia app s OS

- Ak sa využije hw na takto silnú izoláciu, je potrebné vymyslieť mechanizmus, pomocou ktorého môže aplikácia **kontrolovaným spôsobom** pristúpit k hw (alebo k údajom inej aplikácie)
- V podstate sa tento problém redukuje na kontrolovaný prechod z používateľského režimu procesora do kernel módu procesora
- Na architektúre RISC-V sa to deje pomocou inštrukcie `ecall`

ecall

- Vyvolanie služby OS presne definovaným spôsobom
- Procesy nemajú prístup k funkciám jadra OS priamo!

- Systémové volania nie sú „klasické” funkcie
  - `user/forktest.asm` hľadaj `fork`
  - `user/usys.pl` →`user/usys.S`
  - hw prepnutie do kernel módu; OS určuje, kde sa po prepnutí začne vykonávať kód jadra OS

Štruktúra xv6

- Monolitický kernel
  - Rozhranie user/kernel space: systémové volania
- Zdrojové kódy sú usporiadané modulárne
  - `user/` → aplikácie v user móde
  - `kernel/` → kód v kernel móde
- Samotný kernel pozostáva zo samostatných častí
  - Viď `kernel/defs.h` (proc, fs, …)
- Kód je vynikajúco dokumentovaný

Použitie xv6

- Súbor zostavenia `Makefile` riadi
  - Vytvorenie programu jadra (priečinok `kernel/`)
  - Vytvorenie používateľských programov (priečinok `user/`)
  - Vytvorenie disku (priečinok `mkfs/`) pre xv6

- Príkaz `` `make qemu` ``
  - Spúšťa xv6 vo virtualizačnom nástroji qemu
  - Qemu emuluje počítač architektúry RISC-V

RISC-V doska

- Ide o reálny hw → možnosť skutočného spustenia xv6!
- Veľmi jednoduchá základná doska (bez grafického výstupu)
  - CPU má 4 jadrá
  - RAM 128 MiB
  - Podpora prerušení
  - Podpora UART (sériová konzola/klávesnica)
  - Podpora sieťovej karty e1000 (cez zbernicu PCIe)
- Qemu emuluje presne túto konfiguráciu

Prečo qemu a nie hw?

- Museli by ste si ho zakúpiť (a vzhľadom na dodaciu lehotu predmet robiť o rok)
- Pohodlnejšie je využívať služby qemu, nakoľko
  - Qemu emuluje viacero implementácií RISC-V
  - Xv6 využíva počítač “virt” (https://github.com/riscv/riscv-qemu/wiki)
  - Táto implementácia je dosť blízka hw doske SiFive, ale navyše má podporu pre rozhranie VirtIO (https://www.sifive.com/boards)

Čo znamená emulácia?

- Qemu je sw (program), ktorý implementuje RISC-V procesor

```
for (;;) {
  1) read next instruction
  2) decode instruction
  3) execute instruction (updating
  processor state)
}
```

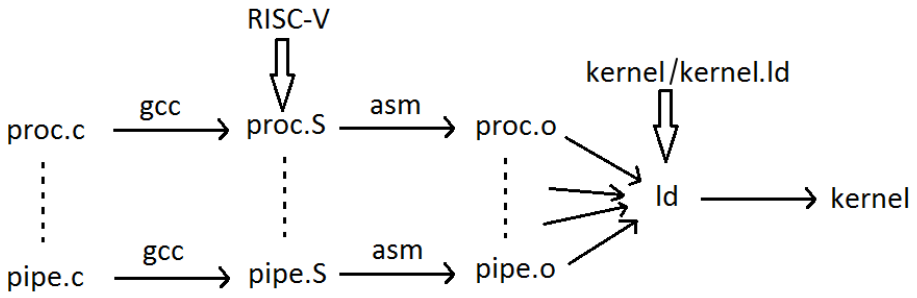

Štart systému xv6

- Využijeme gdb
- Spustíme xv6 s jedným jadrom (nie ako je prednastavená hodnota 3), aby sme mohli sledovať jedno vlákno v gdb

```
$ make CPUS=1 qemu-gdb
```

- Qemu začne vykonávať kód xv6, ktorý sa nachádza v `kernel/entry.S`
  - Viď `kernel/kernel.ld` symbol `_entry` (riadok 2)
  - Čo je `kernel.ld`?

Zostavenie xv6



prevzaté a upravené z https://pdos.csail.mit.edu/6.828/2020/lec/l-os-boards.pdf [21. september 2020]

`Makefile` je nastavený tak, aby vytvoril `asm` súbory (viď napríklad `kernel.asm`)

Štart systému xv6

- `b _entry`
  - Porovnajme inštrukcie so súborom `kernel.asm`
  - `info reg, x/10i _entry`
- `b main`
  - si, continue, next next next next…, tui enable
- step do funkcie `userinit()`
  - Next cez funkciu
  - Viď `proc.h`
  - Step do `allocproc()`
  - Viď `initcode.S/initcode.asm` a `od -t xC initcode`

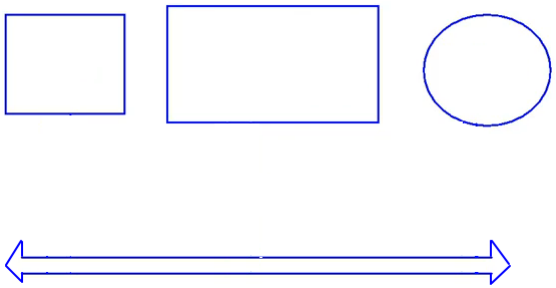
Štart systému xv6

- `b forkret`
  - Next po `usertrapret()`
- `b syscall`
  - `print num`
  - Step do `syscalls[num]()`
  - Nachádzame sa v `` `exec "/init"` ``
- `symbol-file user/init.o`
  - `b main`
  - `continue`

Nejaké otázky?

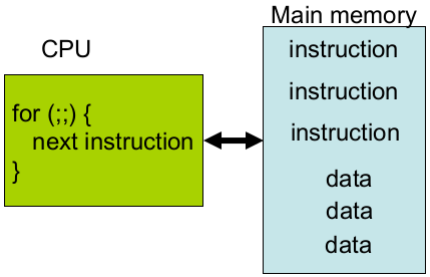
Von Neumann architektúra

- CPU
- Memory
- I/O
- Bus



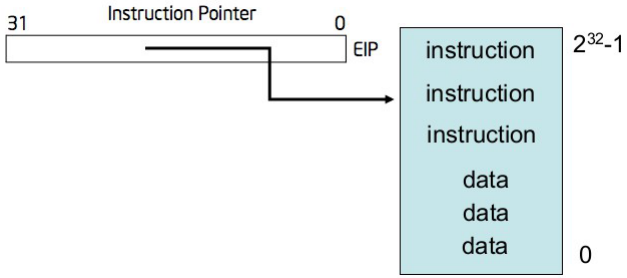
Vykonávanie programu

- Pamäť
  - Inštrukcie
  - Údaje
- CPU
  - Interpretácia
  - Manipulácia

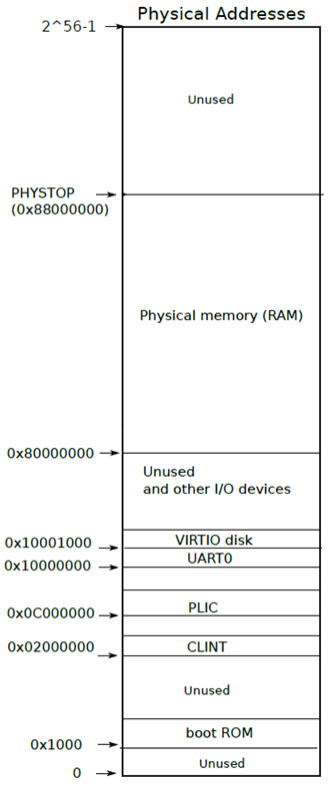


Vykonávanie programu

- Ako sa CPU dostane k ďalšej inštrukcii?
  - Špeciálny register CPU, ktorý sa zvyšuje po každej inštrukcii
  - Automaticky modifikovaný niektorými inštrukciami (volanie procedúry, návrat z procedúry, skok)



Fyzické adresy: organizácia RISC-V



prevzaté a upravené z

https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf

Opakovanie

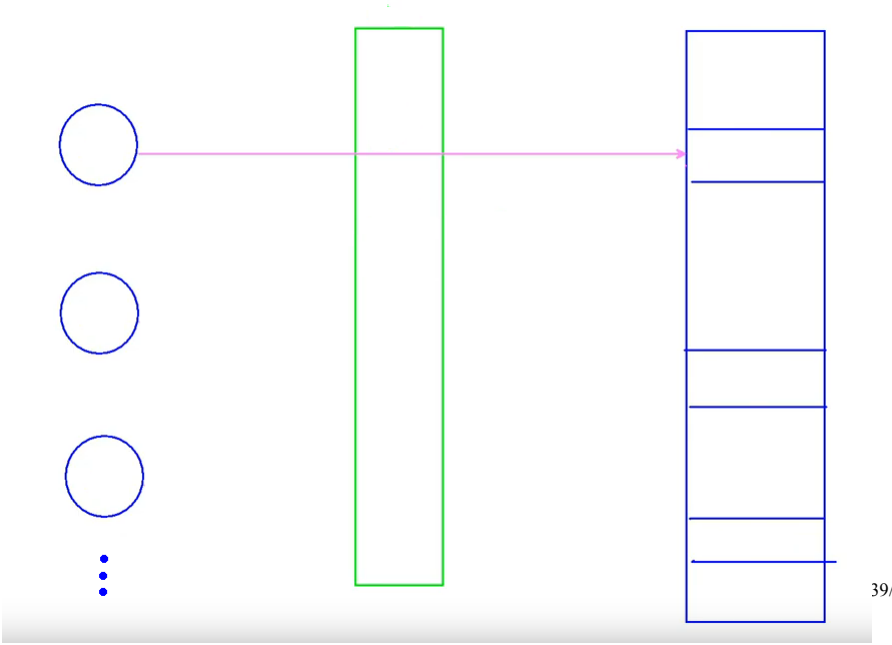
- Dve hw technológie umožňujúce izoláciu procesov od jadra a vzájomne medzi sebou

1. Systémové volania (umožňujú procesom presne definovaným spôsobom vykonať obsluhu služby v jadre OS)
2. Virtuálna pamäť (vytvára zdanie vlastnej pamäte pre proces; znemožňuje zasahovať do pamäte iného procesu alebo jadra)

Pamäť

- Majme v pamäti umiestnené procesy (a kód OS)
- Nech sa v nejakej aplikácii nachádza chyba, ktorá spôsobuje občasné zapísanie na náhodnú adresu v RAM-ke
- Ako zabezpečíme, aby sa neprepísali údaje/kód nejakej aplikácie alebo jadra OS?

Pamäť



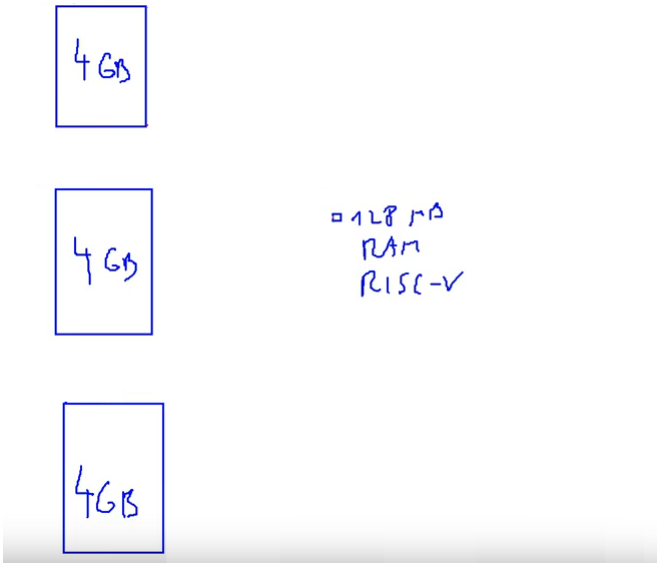
Adresný priestor

- Izoláciou prístupu procesu k pamäťovému priestoru
- Každý proces bude mať „vlastnú” pamäť
- Čo to znamená?
  - Môže do nej zapisovať a čítať z nej
  - Nemôže zapisovať a čítať z pamäte iného procesu alebo jadra OS

Adresný priestor

- Otázka
- Ako vtesnať (multiplexovať) viacero takýchto samostatných pamäťových boxov do jednej RAM-ky a zachovať izoláciu medzi nimi?

Adresný priestor



Adresný priestor

- Jestvujú viaceré možnosti, najrozšírenejšie sú

1.Segmentácia

2.Stránkovanie

3.Kombinácia seg+str alebo str+seg

Schéma prekladu adresy

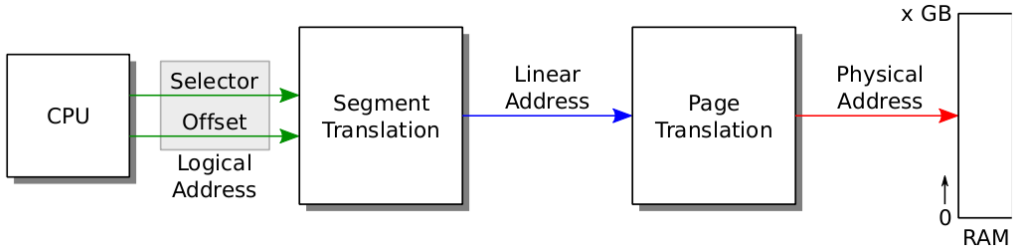
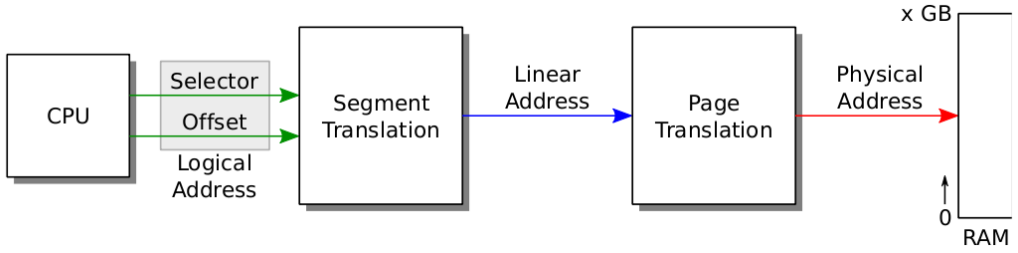
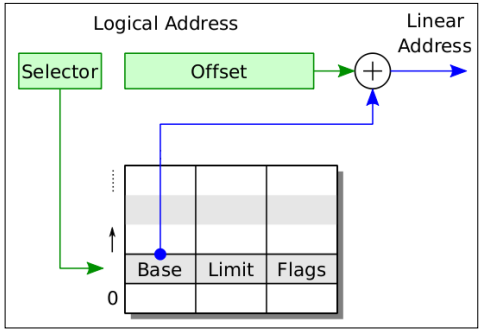
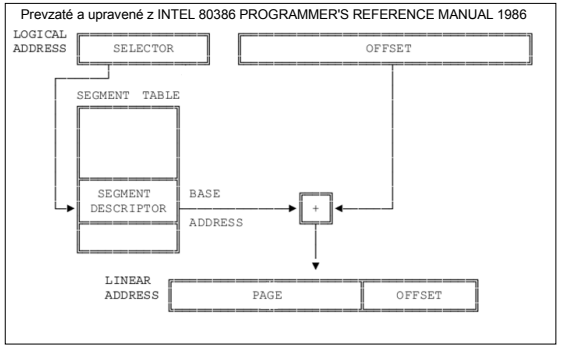


Schéma prekladu adresy







Stránkovanie

- Ide o mechanizmus nepriamej adresácie

Segmentácia

Zhrnutie

- CPU používa logické (virtuálne) adresy, VŽDY!
- Výsledok segmentácie je preklad logickej (virtuálnej) adresy na tzv. lineárnu adresu

- Prečo lineárnu?

Stránkovanie

- Ide o mechanizmus nepriamej adresácie
- Odkaz na aktuálnu tabuľku stránok (tabuľku, ktorá sa pri preklade používa), sa nachádza v špeciálnom registri CPU
- Meniť hodnotu tohto registra CPU je možné iba v privilegovanom režime (*kernel* mód)

Stránkovanie

- CPU štartuje v režime, kedy je stránkovanie vypnuté
  - Prečo? Ak by bolo zapnuté, tak máme problém sliepky a vajca…
- Inicializačný kód jadra
  - Vyplní tabuľku stránok
  - Nastaví register, ktorý ukazuje na túto tabuľku
  - Zapne stránkovací hardvér

Stránkovanie

- Jadro OS v privilegovanom režime môže meniť obsah registra, ktorý ukazuje na tabuľku stránok
- Teda iba jadro OS môže meniť mapovanie, ktorá fyzická pamäť je dostupná z ktorej virtuálnej adresy (či už samotného jadra alebo procesu)
- Používateľský proces beží v *user* móde CPU, nemôže meniť obsah tohto registra – pokus o jeho zmenu vyvolá výnimočný stav

Stránkovanie

- Každá tabuľka stránok definuje vlastné mapovanie, vlastný adresný priestor, ktorý nazývame VIRTUÁLNY
- Obrázok 7 s mapovaním procesov do RAM

Stránkovanie

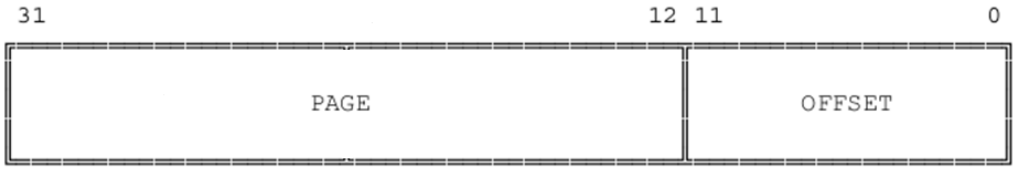
- Stránkovanie umožňuje (podobne ako segmentácia, ale nespomínali sme)
  - Rozlíšiť oprávnenia typu prístupu (Read / Write)
  - Určiť, či mapovanie jestvuje (Present)
  - Rozlíšiť oprávnenia prístupu (Kernel / User)
  - Zistiť, či prístup k určitej adrese vyvolal zmenu hodnôt v pamäti (Dirty)
  - Zistiť, či bolo vôbec niekedy pristúpené k nejakej adrese (Access)
- Triky s virtuálnou pamäťou

Stránkovanie – čo nám k tomu treba

- Rámec (*page frame*) versus stránka (*page*)
- Lineárna adresa
- Tabuľka stránok
- Položka tabuľky stránok
- Ako sa robí preklad

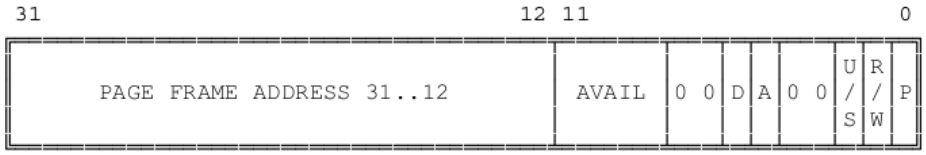
Prevzaté a upravené z INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Formát lineárnej adresy



Prevzaté z INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Položka tabuľky stránok (Intel)

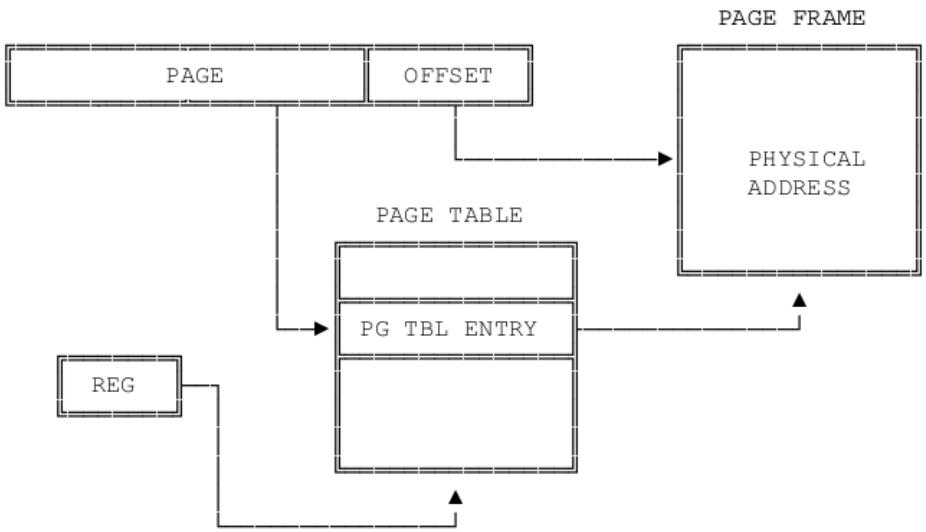



```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

Prevzaté a upravené z INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Preklad lineárnej adresy na fyzickú

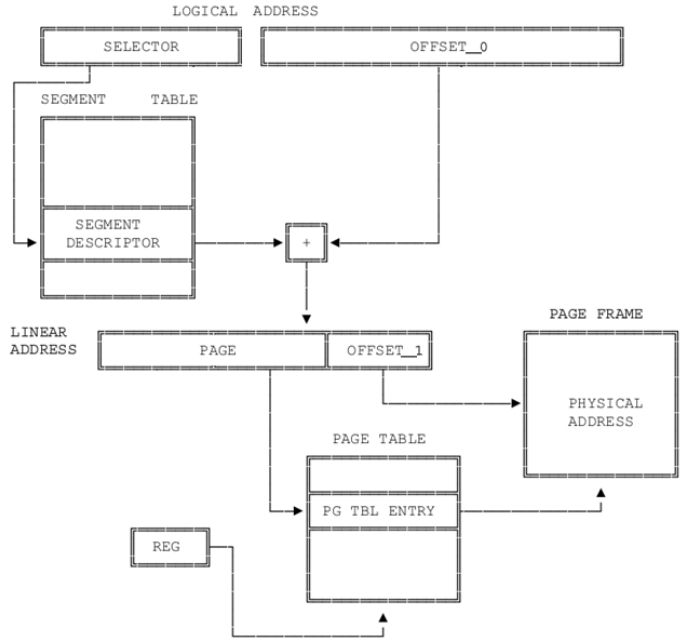


Stránkovanie…

- Aby svet spel k lepšiemu, v takejto podobe (ako bolo na prednáške) sa stránkovanie nikde nepoužíva
- Viac o stránkovaní a virtuálnej pamäti na budúcej prednáške

Prevzaté a upravené z INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986

Kombinácia Seg a Str



Domáca úloha

- Prečítať kapitolu 2 a 4.3, 4.4
- *Operating system organization*
- https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Prednáška 3

Virtuálna pamäť

Štruktúra prednášky

- Adresné priestory
- Stránkovací hardvér
- Programový kód xv6 pre správu VM

Téma

- Majme program, ktorý z času na čas zapíše niečo na náhodnú adresu v pamäti
- Ako udržať takýto program „na uzde”?
- Izolujme adresné priestory procesov (a jadra)

Izolácia adresného priestoru

- Každý proces má vlastný adresný priestor
- Môže čítať/zapisovať iba v rámci svojho priestoru (nemôže čítať ani zapisovať v inom)
- **Ako SÚČASNE implementovať viac adresných priestorov v rámci jedinej fyzickej pamäte (RAM) a zabezpečiť izoláciu medzi nimi?**

Izolácia adresného priestoru

- Odpoveď: stránkovací hardvér
- Xv6 využíva hardvér procesora RISC-V

Stránkovanie

- Poskytuje *<u>nepriamu</u>* adresáciu
  - program používa smerníky (adresy); tieto adresy využíva CPU na prístup k pamäti
    - Nie sú to však (vo všeobecnosti) adresy operandov vo fyzickej pamäti!
    - Ide o tzv. virtuálne adresy!
  - Hardvér správy pamäte (MMU) „prekladá” virtuálne adresy na fyzické adresy
    - V literatúre sa často spomína aj termín „lineárna adresa” (v rámci predmetu nerozlišujeme)
    - Až fyzická adresa je adresou operandu na adresnej zbernici (nemusí to byť iba RAM!!!)

Stránkovanie

- Program dokáže pracovať iba s virtuálnymi adresami, nie fyzickými (čo sa týka menenia/čítania obsahu RAM)
- Jadro OS má za úlohu riadiť mapovanie každej VA na PA (nastaviť údaje pre toto mapovanie)

Stránkovanie

- MMU používa na „preklad” tabuľku, v ktorej
  - VA je indexom (zjednodušene)
  - PA je hodnotou na indexe (zjednodušene)
- Tabuľka má názov „tabuľka stránok” (angl. *Page Table*)
- Jedna tabuľka popisuje jeden adresný priestor (mapovanie VA na PA)
- MMU dokáže obmedziť, ktoré VA sú prístupné v *user* móde CPU

Stránkovanie

- Odkaz na aktuálnu tabuľku stránok (tabuľku, ktorá sa pri preklade používa), sa nachádza v špeciálnom registri CPU
- Meniť hodnotu tohto registra je možné iba v privilegovanom režime CPU (*kernel/supervisor* mód)
  - Pokus o zmenu v *user* móde vedie ku výnimke

Stránkovanie

- CPU štartuje v režime, kedy je stránkovanie vypnuté
  - Prečo?
- Inicializačný kód jadra
  - Vyplní tabuľku
  - Nastaví register, ktorý ukazuje na tabuľku stránok
  - Zapne stránkovanie CPU

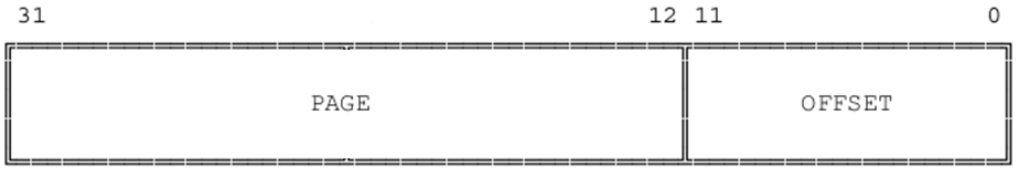
Stránkovanie

- Stránkovanie umožňuje
  - Rozlíšenie oprávnenia typu prístupu (*Read / Write*)
  - Určiť, či mapovanie jestvuje (*Present*)
  - Rozlíšenie oprávnenia prístupu (*Supervisor / User*)
  - Zistiť, či prístup k určitej adrese vyvolal zmenu hodnôt v pamäti (D – *dirty bit*)
  - Zistiť, či bolo vôbec niekedy pristúpené k nejakej adrese v stránke (A – *access bit*)
- Triky s virtuálnou pamäťou

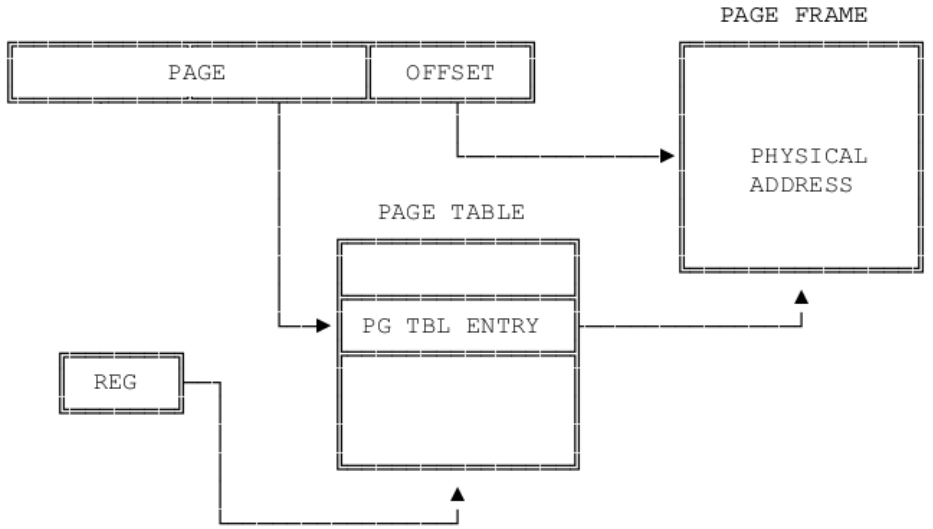
Stránkovanie – čo nám k tomu treba

- Rámec (*page frame*) versus stránka (*page*)
- Lineárna adresa
- Tabuľka stránok
- Položka tabuľky stránok
- Ako sa robí preklad

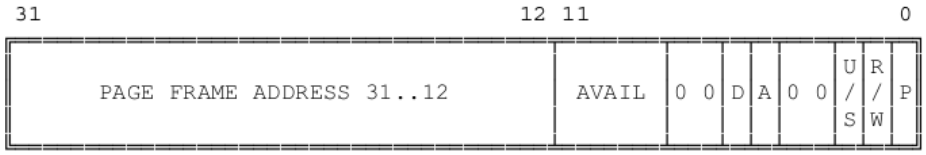
Formát lineárnej adresy



Preklad lineárnej adresy na fyzickú



Položka tabuľky stránok (Intel)



P - PRESENT
R/W - READ/WRITE
U/S - USER/SUPERVISOR
D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

RISC-V

- Mapuje stránky o veľkosti 4 KiB
- Zarovnané na hranicu 4 KiB (0, 4 Ki, 8 Ki, 12 Ki, …)
- Koľko bitov potrebujeme na indexovanie (adresovanie) v rámci jednej stránky 4 KiB? 12

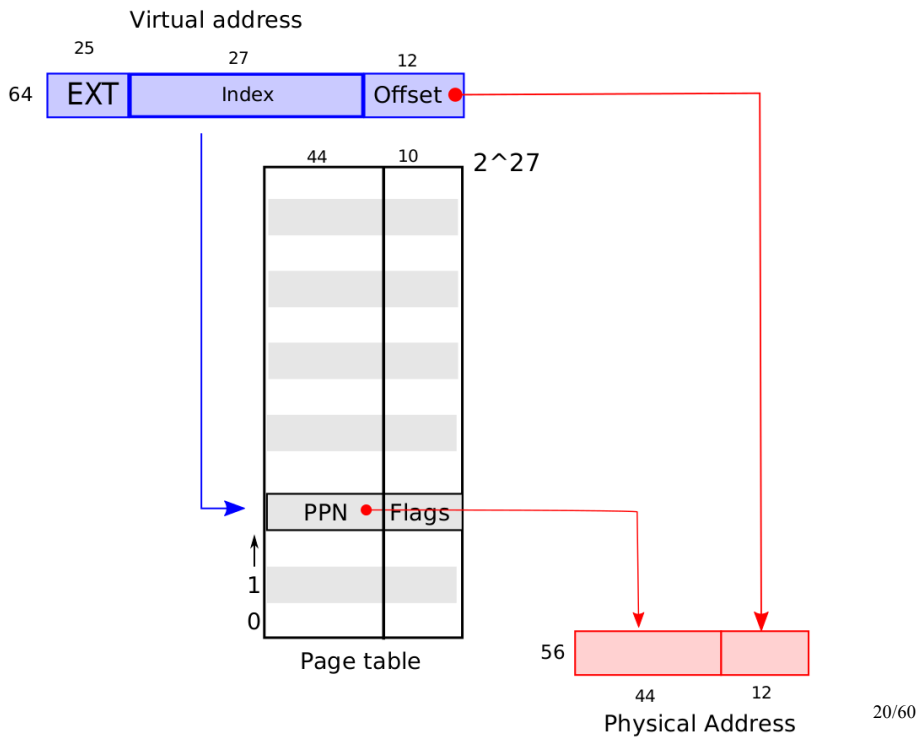
RISC-V

- Xv6 využíva RISC-V v 64-bitovom adresnom režime
- Na index do PT sa využíva 64−12 = 52 bitov z VA
- Nie tak celkom, horných 25 bitov z týchto 52 sa nevyužíva, takže index má reálne 27 bitov

RISC-V

- Máme 27 bitov na index, koľko indexov je to spolu? 2^27=2^7 * 2^20 = 128 * 1 Mi = 128 Mi
- Každá položka tabuľky má 64 b, čo je 8 B
- Koľko teda bude zaberať celá PT v pamäti? 128 Mi * 8 B = **1 GiB** ;)
- Ak PT pre 1 proces zaberá 1 GiB v RAM, koľko procesov asi tak môže bežať v OS? :D :D :D
- Xv6 má 128 MiB RAM… Tak tu **niečo nesedí**…

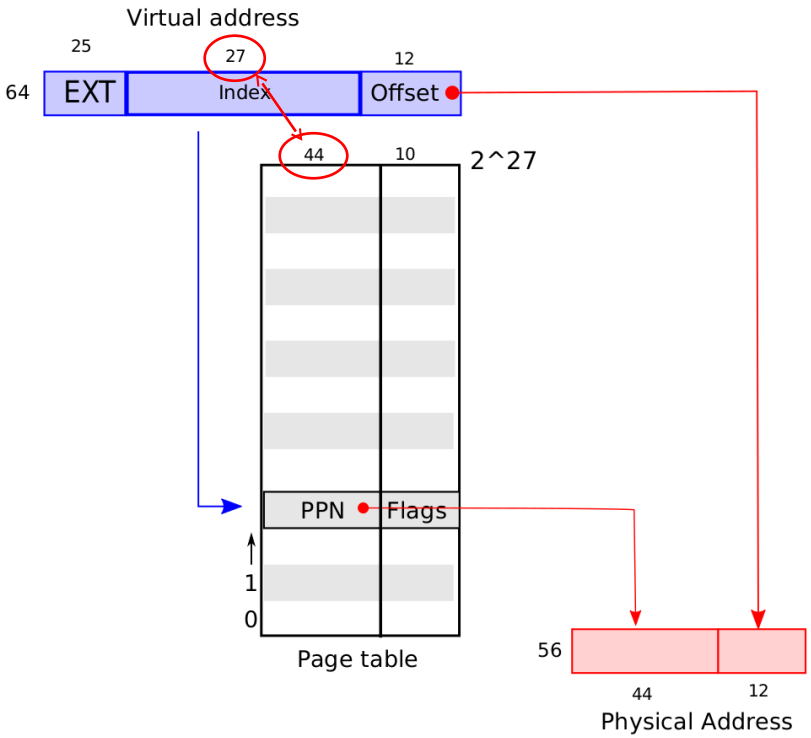
Preklad MMU



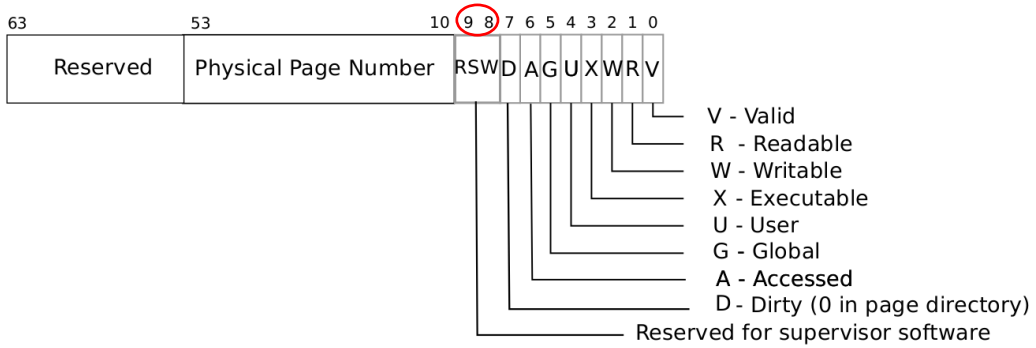
Preklad MMU

- Záznam v tabuľke PT sa nazýva PTE (*Page Table Entry*)
  - Má 64 b (8 B), ale využíva sa „iba” 54
  - Horných 44 bitov PTE tvoria horných 44 bitov fyzickej adresy (*Physical Page Number* = PPN)
  - Spodných 10 sú tzv. príznaky
    - *Present*, *Writeable*, *User*, *Accessed*, *Dirty*…

- !POZOR! **veľkosť VA != veľkosť PA**



PTE



PT

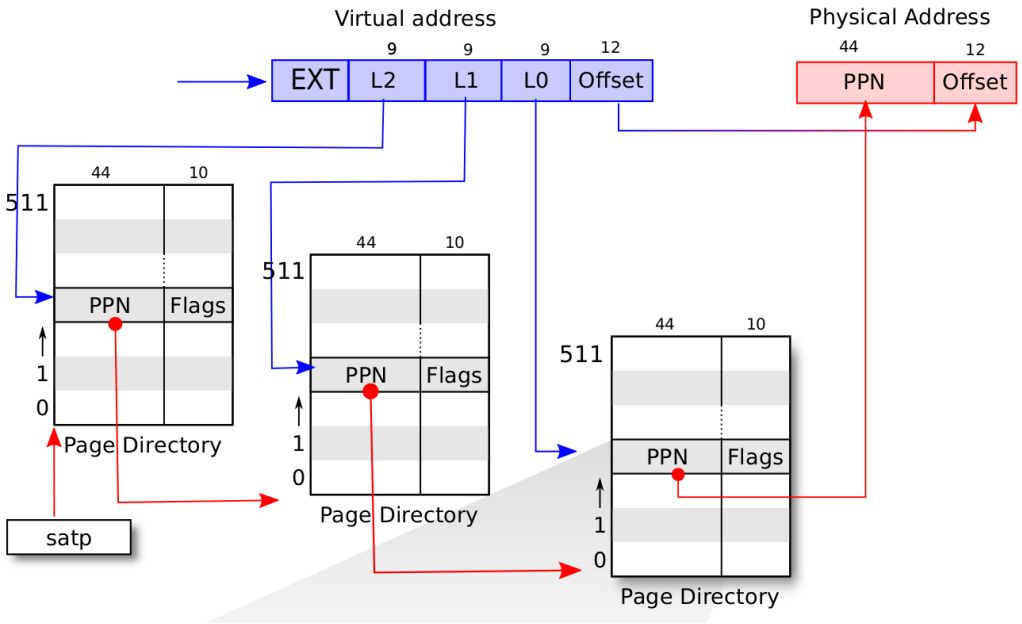
- Kde je PT uložená? V RAM

- MMU dokáže manipulovať so záznamami PTE
- Podobne to dokáže aj OS

PT

- Ako sme už vypočítali, veľkosť PT je 1 GiB!
  - 1 PT na 1 adresný priestor
  - 1 adresný priestor zväčša pre 1 aplikáciu (proces)
- Avšak proces často potrebuje iba pár KiB/MiB pamäte, takže iba pár Ki/Mi záznamov PTE bude využitých, ostatné budu prázdne!

PT RISC-V



PT

- RISC-V 64 využíva na ušetrenie miesta tzv. **trojúrovňovú** tabuľku stránok

PT

- V každej úrovni (L2, L1, L0) máme k dispozícii 9 bitov na index → 512 položiek v PD (*Page Directory*)
- 512*512*512 = 2^9 * 2^9 * 2^9 = 2^(3*9) = 2^27
- PTE môže byť neplatné (bit *Valid*); nejestvuje prepojenie t. j. mapovanie s RAM)
- Preto môže byť PT pre proces malá!

PT

- V jednej úrovni máme 512 položiek PTE; jedna má veľkosť 64 b, takže celkovo je to 512 * 8 B = 4 KiB (čírou „náhodou” to sedí na veľkosť stránky?)
- 512 záznamov môže ukazovať na 512 stránok v RAM; 1 tabuľkou vieme „pokryť” max. 512 * 4 KiB RAM, t. j. 2048 KiB, t. j. 2 MiB
- Na pokrytie mapovania niekoľko desiatok MiB nám stačia desiatky (stovky) KiB tabuľky PT namiesto 1 GiB

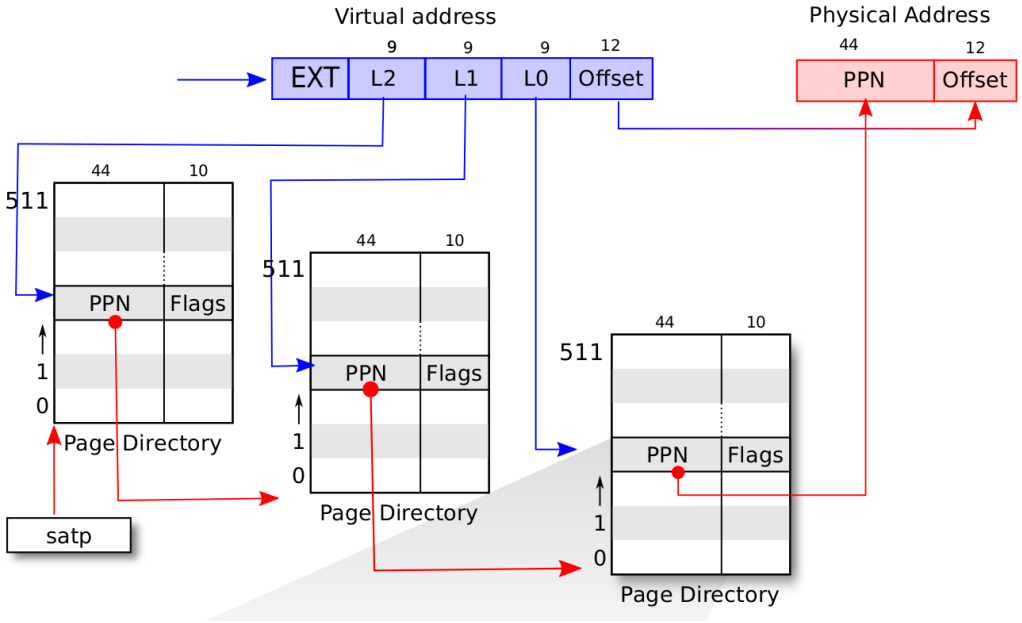
PT

- Ako MMU „vie”, kde v RAM sa PT nachádza?
- Na RISC-V je **FYZICKÁ** adresa hornej časti PT (pre index L2) v registri `satp`
- Prepísaním `satp` sa prepínajú adresné priestory!
- Stránky PT môžu byť voľne roztrúsené v RAM, nemusí ísť o súvislú oblasť RAM!!!

Preklad va2pa

- MMU musí nájsť správny záznam PTE zodpovedajúci danej VA; ako?
  - Z registra `satp` vieme PA pre obsah tabuľky PD_{L2}
  - Horných 9 bitov indexu VA (L2) ukazuje do PD_{L2}; z PD_{L2}[L2] získame PA pre obsah PD_{L1}
  - Ďalších 9 bitov indexu VA (L1) ukazuje do PD_{L1}; z PD_{L1}[L1] získame PA pre obsah PD_{L0}
  - Posledných 9 bitov indexu VA (L0) ukazuje do PD_{L0}; z PD_{L0}[L0] získame PA pre PTE
  - PA = PPN z PTE plus spodných 12 bitov VA

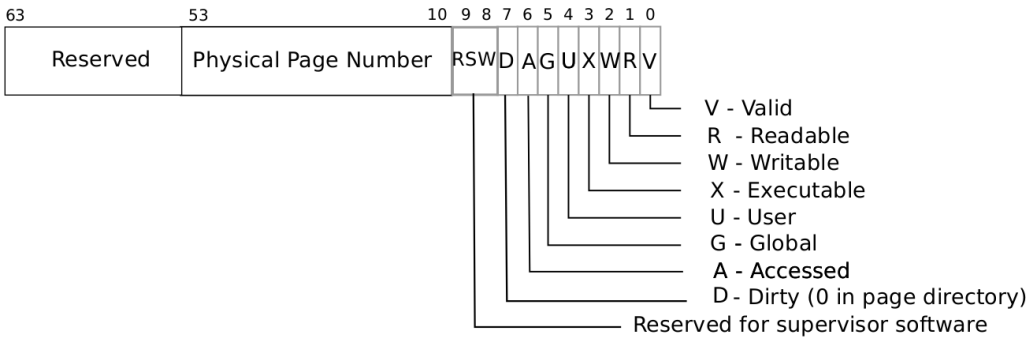
PT RISC-V



Príznaky PTE

- Xv6 využíva V, R, W, X, U
- V cvičení 3 musíte využiť príznak A
- Neskôr využijeme aj niektorý z tých, ktoré sú voľne dostupné pre OS

Príznaky PTE



Príznaky PTE

- Čo ak bit V nie je nastavený? Alebo zapisujeme a nie je nastavený W?
- Výpadok stránky (angl. *Page Fault*)
  - Vynútený presun do jadra (trap.c)
  - Jadro buď vypíše chybovú správu a ukončí proces, ktorý chybu spôsobil („usertrap(): unexpected scause…”) (viď ukážka cat.c)
  - Alebo môže nainštalovať chýbajúci PTE a obnoviť beh procesu (napr. ak sa používa *swapovanie* pamäte RAM na disk)

Výhody stránkovania

- Spojitý virtuálny adresný priestor nevyžaduje spojitý fyzický adresný priestor! (vôbec neprichádza ku ***externej*** fragmentácii!!!)
- „*lazy allocation*”; alokácia pamäte až pri jej prvom použití (nastane výpadok, jadro alokuje PTE a proces zopakuje inštrukciu)
- „*copy-on-write fork*”; kópia stránky až pri prvom pokuse o zápis

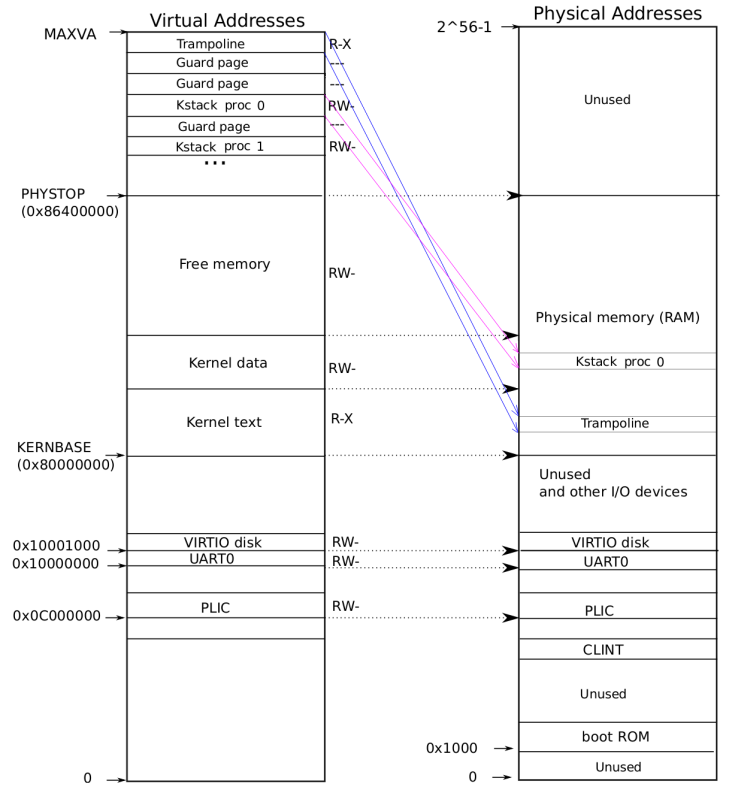
Virtuálna pamäť v jadre

- Pre procesy je využitie VM v poriadku, ale prečo ju využívať aj v jadre?
- Môže bežať jadro iba s fyzickou pamäťou? ÁNO, môže!

Virtuálna pamäť v jadre

- Väčšina jadier OS však využíva VA; prečo?
- Je (časovo) drahé vypínať/zapínať stránkovanie
- Uľahčuje prechod medzi *user*/*kernel* (tým istým mapovaním tej istej stránky – viď *trampoline*)
- Uľahčuje to hľadanie chýb
  - Text (kód) jadra označíme X, údaje nie
  - Ponechanie pamäťovej „diery” pod zásobníkom

Virtuálna pamäť **JADRA** xv6



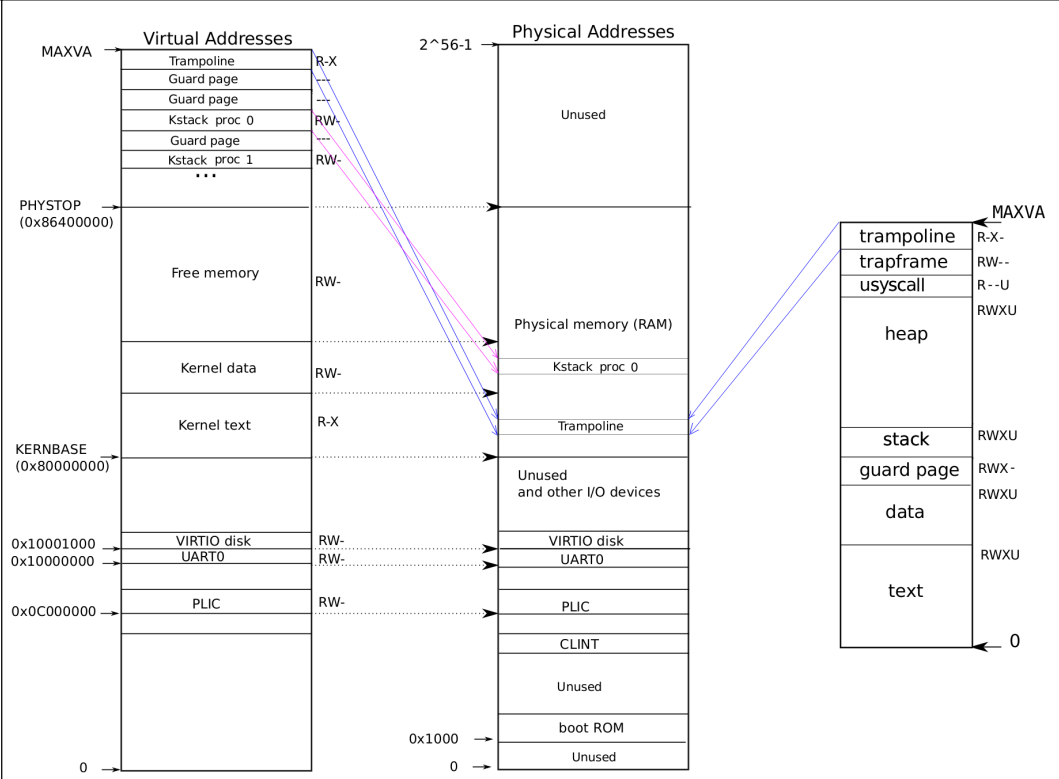
Virtuálna pamäť JADRA xv6

- Jednoduché mapovanie virtuálnej pamäte na fyzickú jedna-k-jednej
- Prečo sa mapujú aj zariadenia?
- Viď oprávnenia rôznych oblastí...

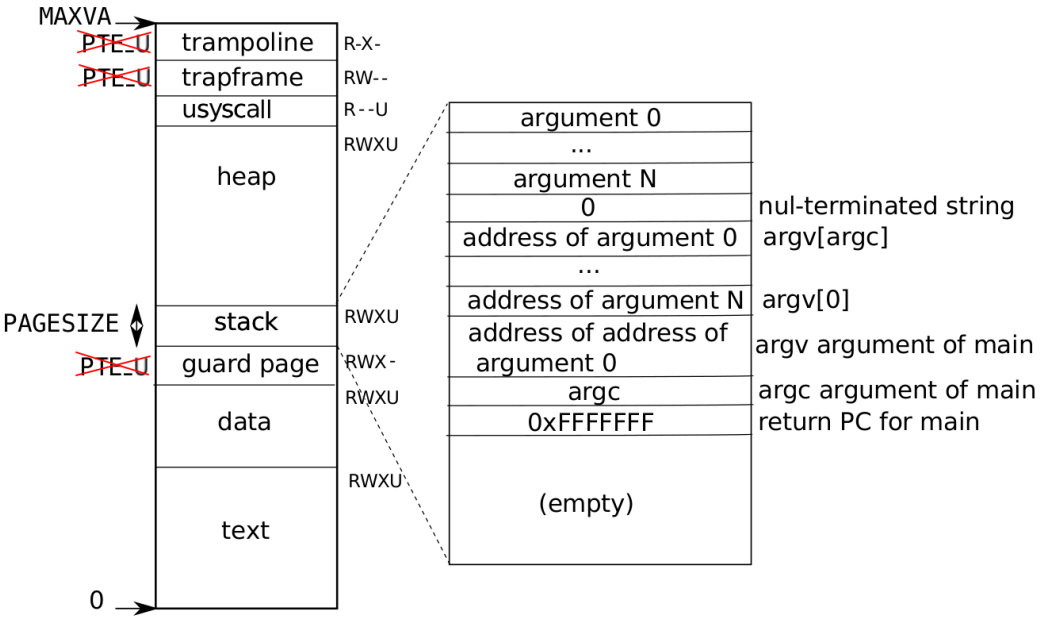
Virtuálna pamäť **POUŽÍVATEĽA** xv6

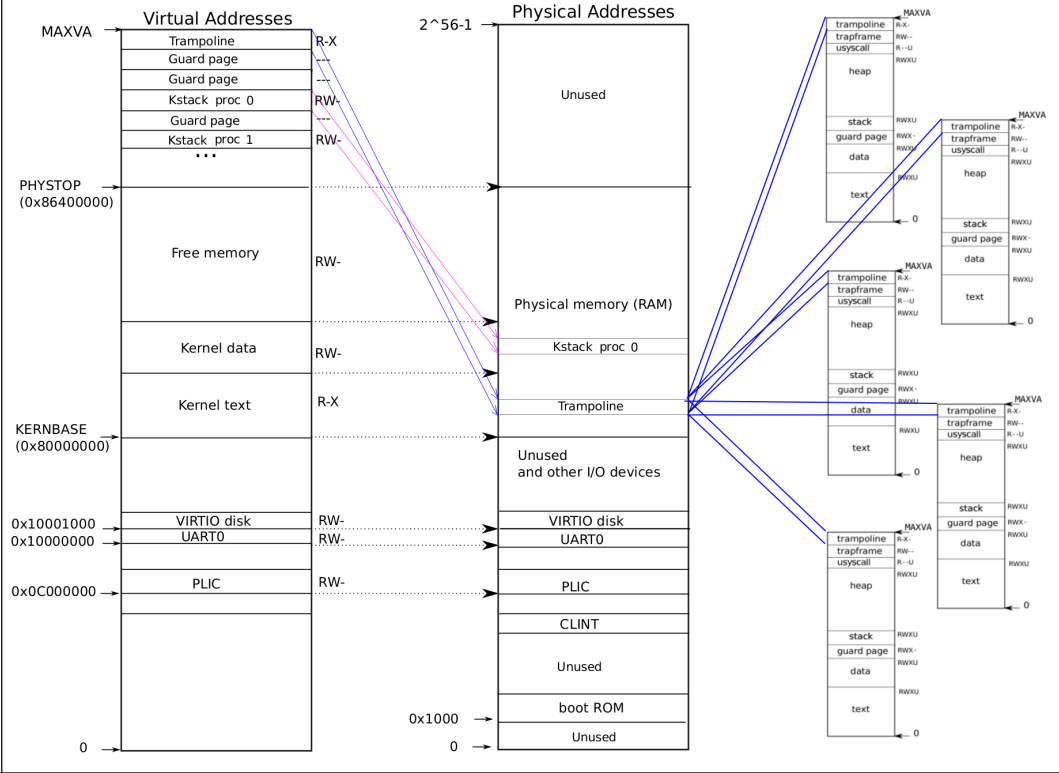
- Každý proces má vlastný adresný priestor
- Vlastnú tabuľku stránok
- Viď `trampoline` a `trapframe` – nie sú zapisovateľné pre používateľský proces!!!

- Jadro OS nastavuje register `satp` pri prepínaní procesov (`usertrapret()` `kernel/trap.c`:123)



Virtuálna pamäť **POUŽÍVATEĽA** xv6





Virtuálna pamäť xv6

- Vlastnosti takejto organizácie adresného priestoru
  - VA používateľa začína od 0 (avšak v každom procese je VA používateľa od 0 mapovaná na inú RAM – vo všeobecnosti)
  - 256 GiB halda používateľa ;) (viď `MAXVA`)
  - Jednoduchý prechod *user* ↔ *kernel* mapovaním trampolíny a *trapframe* (o tom nabudúce)
  - Neľahký prístup jadra k pamäti používateľa!!!
  - Ľahký prístup jadra k fyzickej pamäti: `pa(x)` mapovaná na `va(x)`

Kód VM xv6

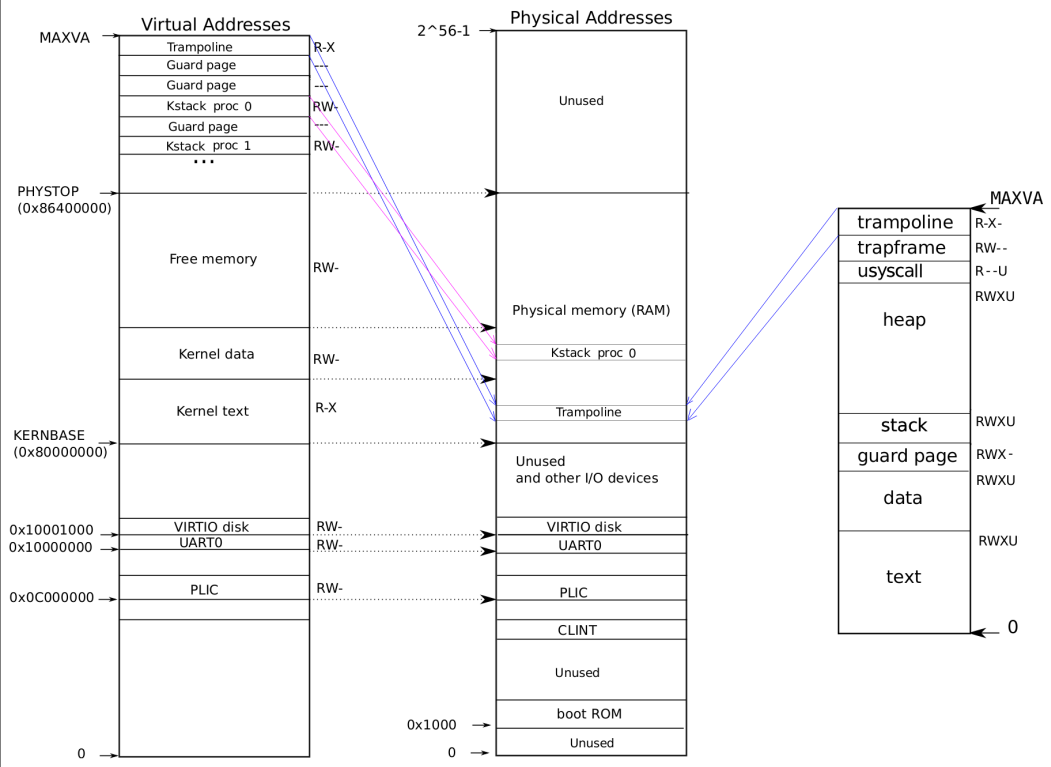
- Inicializácia adresného priestoru jadra
- kernel/memlayout.h a kernel/vm.c kvminit()
  - Koľko adresného priestoru vie obsiahnuť 1 L0 položka? 4 KiB
  - Koľko 1 L1 položka? 2 MiB
  - Koľko 1 L2 položka? 1 GiB
  - Ako veľký je celý adresný priestor? 512 GiB

Kód VM xv6

- Koľko pamäte (stránok) je použitých na reprezentáciu PT (t. j. mapovania adresného priestoru) po prvom volaní `kvmmap()`?
  - `kernel/vm.c`:28 `kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);`
- Koľko pamäte (stránok) sa týmto volaním mapuje?
- Ukážka `vmprint()` za týmto prvým volaním

Kód VM xv6

- Koľko položiek v PT jadra zaberá `VIRTIO`
  - `kvmmap(kpgtbl, VIRTIO0, …);`
- Koľko položiek v PT jadra zaberá `PLIC`
  - `kvmmap(kpgtbl, PLIC, …, 0x400000, …);`
- Je trampolína mapovaná v `kpgtbl` iba raz?
- A čo zásobníky procesov?



Kód VM xv6

- `mappages()`
  - Argumenty: top PD, va, size, pa, perm
  - Do PT zaznamenáva mapovanie <va; va+size) na príslušný interval <pa; pa+size)
- walk()
  - Napodobňuje činnosť MMU; pre danú VA a PT nájde príslušný PTE záznam v PT
  - Makro `PX(level, va)` extrahuje 9 bitov indexu na úrovni `level`

Kód VM xv6

- `kvminithart()`
  - `w_satp()`
  - `sfence_vma()`
- Prečo po nastavení PT musí nasledovať inštrukcia `sfence_vma()`? (kernel/riscv.h)
  - TLB (*Translation Lookaside Buffer*) – vyrovnávacia pamäť pre preklad VA→PA

Kód VM xv6

- `walk(pagetable, va)` algoritmus:

1)PTE_addr = &pagetable[PX(level, va)]

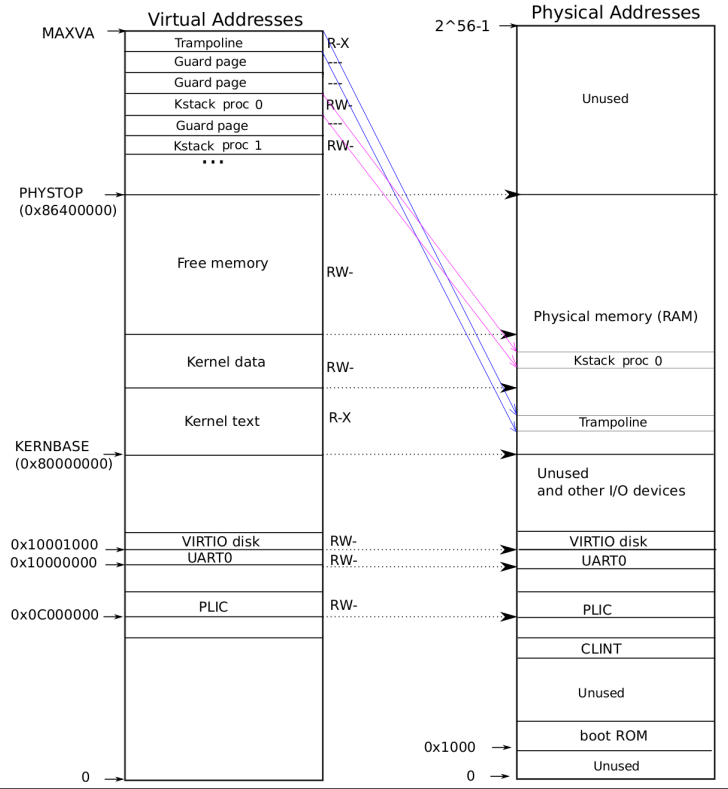
2)If is set PTE_V in *PTE_addr
- Príslušná tabuľka v PT jestvuje
- PTE2PA extrahuje PPN zo záznamu PTE

3)If not set PTE_V in *PTE_addr
- Alokuj tabuľku ďalšej úrovne
- Vyplň *PTE_addr s PPN alokovanej tabuľky (PA2PTE)

4)Vráť adresu PTE z (vytvorenej/jestvujúcej) tabuľky L0

Kód VM xv6

- `procinit()` v `kernel/proc.c`
- Statické pole procesov
- Každému procesu alokuj v ramke stránku na zásobník jadra (veľkosť `PGSIZE`) a namapuj do VA kernelu
- Každý proces má vlastný zásobník v jadre
- Každý zásobník má „*guard page*”!!! (obrázok)



Kód VM xv6

- Inicializácia používateľského adresného priestoru
  - `allocproc()` v `kernel/proc.c`
  - `fork()` v `kernel/proc.c`
  - `exec()` v `kernel/exec.c`

Kód VM xv6

- Inicializácia používateľského adresného priestoru
  - `allocproc()` alokuje prázdnu PT najvyššieho stupňa
  - `fork()` robí uvmcopy()
  - `exec()` prepíše PT procesu novou
    - `uvmalloc()`
    - `loadseg()`
- Ukážka `vmprint()` pre procesy `init` a `sh`

Kód VM xv6

- Ak chce (používateľský) proces viac pamäte (alokácia z haldy), vyvolá systémové volanie `sbrk(n)`; o n sa zväčší pamäť procesu
- Viď `user/umalloc.c` volanie `sbrk()`

Kód VM xv6

- Každý proces má svoju veľkosť; volanie `sbrk()` pridáva procesu na konci pamäť, zväčšuje veľkosť procesu (`kernel/sysproc.c`)
- Alokuje fyzickú pamäť (RAM)
- Mapuje ju do PT procesu
- Vracia počiatočnú adresu tejto novej pamäte

Kód VM xv6

- `growproc()` v `kernel/proc.c`
  - `proc->sz` je aktuálna veľkosť procesu
  - `uvmalloc()` obsahuje hlavnú funkcionalitu
  - Pri prepnutí z jadra do *user space* sa do `satp` uloží adresa aktualizovanej PT
- `uvmalloc()` v `kernel/vm.c`
  - Prečo je tam `PGROUNDUP`?
  - Prečo `mappages(…, PTE_W|PTE_X|PTE_R|PTE_U)`?

Čítanie na večer / nad ránom

- Prečítať kapitolu 3
- Page Tables
- https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Prednáška 4

Prechod user ←→kernel pomocou systémových volaní

Téma

- Ako funguje prechod do/z jadra pri systémových volaniach
- Ponoríme sa do jadra OS

write(fd, buf, n)

- Môžeme použiť v používateľskom priestore funkciu (z pohľadu jazyka C) na [za|vy] volanie (uskutočnenie) služby jadra?
- Bolo by to rýchle a efektívne
- Taktiež flexibilné na prenos komplexnejších údajových typov
- A išlo by o programátorom známy mechanizmus

write(fd, buf, n)

- Žiaľ, nemôžeme použiť takýto mechanizmus!
- Dôvodom je IZOLÁCIA procesov
- Izolácia ovplyvňuje väčšinu návrhu jadra OS

Čo to je izolácia

- **Vynútená separácia z dôvodu zapúzdrenia dôsledkov zlyhaní**

Čo to je izolácia

- **Vynútená separácia z dôvodu zapúzdrenia dôsledkov zlyhaní**
- Predmetom izolácie je zvyčajne proces
- Zabraňuje
  - procesu X modifikovať/čítať údaje o procese Y (napr. modifikovať tabuľku deskriptorov, r/w prístupy do pamäte…)
  - procesu miešať sa do úloh a služieb OS

Hlavné nástroje izolácie

1)Virtuálny adresný priestor procesu

2)Privilegovaný režim činnosti CPU (používateľský proces nemôže robiť I/O operácie a meniť dôležité systémové registre CPU)

- Vynútená **kontrola toku riadenia** pomocou systémových volaní; prechod user→ kernel!!!

Privilegovaný režim CPU

- RISC-V CPU **3** režimy: *supervisor*, *user*, *machine*
- Kernel (jadro OS) beží v režime *supervisor*
- Bežný používateľský program v režime *user*

Privilegovaný režim CPU

- RISC-V CPU **3** režimy: *supervisor*, *user*, *machine*
- Kernel (jadro OS) beží v režime *supervisor*
- Bežný používateľský program v režime *user*
- Čo môže *supervisor* oproti *user* naviac:
  - I/O operácie (prístup k zariadeniam)
  - Nastavovanie adresného priestoru (virtuálna pamäť pomocou zmeny registra *satp*)
  - R/W prístup k špeciálnym registrom CPU
- Takmer každý CPU využíva tento princíp

Čo sa deje pri systémovom volaní?

- Procesor je nastavený na beh *user* programu
- Čo je potrebné na procesore RISC-V urobiť:
  - Zmeniť mód CPU z *user* na *supervisor* (pre jadro)
  - Uložiť 32 *user* registrov a register PC
  - Zmeniť tabuľku stránok z *user* na *kernel*
  - Zmeniť zásobník z *user* na *kernel*
  - Skočiť na vykonávanie kódu jadra v C

Ciele riadeného prechodu

1. nedovoliť zasahovať *user* programu do prechodu *user* → *kernel*
2. prechod musí byť pre *user* program transparentný (bez akéhokoľvek zásahu používateľa)

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?

1) Prístup k špeciálnym registrom CPU
   - `satp` – tabuľka stránok
   - `stvec` – inštrukcia `ecall` skočí na addr uloženú v tomto registri (v xv6 ukazuje na kód trampolíny)
   - `sepc` – inštrukcia `ecall` sem uloží user PC
   - `sscratch` – pomocný register; xv6 ho využíva na uloženie adresy pamäťovej oblasti trapframe
2) *Supervisor* má prístup ku stránkam, ktoré nemajú nastavený príznak `PTE_U`

Prečo prepínať cpu mód?

- Prečo umožniť prechod do *supervisor* módu?

3) *Supervisor* má priamy prístup k hw

- Nič iné nemá naviac…
- To znamená, že ani v *supervisor* móde nie je možné pristupovať k adresám, pre ktoré nie je platné mapovanie v tabuľke stránok

Kód xv6 prechodu user→kernel

```
User space     write()        návrat z write()
-------------------------------------------------------
Kernel space   uservec()      userret()    (trampoline.S)
               usertrap()     usertrapret()      (trap.c)
               syscall()         ^            (syscall.c)
               sys_write() ------|
   (sysfile.c)
```


Tok riadenia

write()
  trampoline.S / uservec
    trap.c / usertrap()
      syscall.c / syscall()
        sysfile.c / sys_write()
      syscall.c / syscall()
    trap.c / usertrapret()
  trampoline.S / userret
write()

Ako *syscall* vstúpi do jadra

- napríklad xv6 *shell* vypisujúci *prompt* (‘$’)
  - user/sh.c:137 `write(2, “$ “, 2)`
  - user/usys.S:29
    - `li a7, SYS_write` (→kernel/syscall.h:17)
    - `ecall` (vyvolá **RIADENÝ** prechod user→kernel)
  - user/sh.asm:22, adresa `write` je 0xe16
  - user/sh.asm:1939, adresa `ecall` je 0xe18

Ako *syscall* vstúpi do jadra

$ make CPUS=1 qemu-gdb …

(gdb) c

Ctrl+C

(gdb) b *0xe16

(gdb) c

V konzole qemu-gdb zadáme Enter a vrátime sa do okna s gdb

Nachádzame sa pred systémovým volaním `write()`

Ako *syscall* vstúpi do jadra

(gdb) add-symbol-file user/sh.o

(gdb) ctrl-x 2

(gdb) ctrl-x 2

V hornom okne máme zobrazené registre CPU

V strednom vidíme asm kód

V dolnom zadávame príkazy gdb

Ako *syscall* vstúpi do jadra

(gdb) b *0xe1c (adresa inštrukcie ret v sh.asm)

→ pozri okno behu xv6!

(gdb) c

→ pozri okno behu xv6!

Na konzolu sa vypíše *prompt*.

Ako *syscall* vstúpi do jadra

(gdb) si (posun o 1 inštrukciu ďalej)

→ user/sh.asm:30 adresa 0x24 (`memset()`)

(gdb) c

→ nič sa nedeje… vstup z klávesnice!

→ sme na adrese 0xe16 (syscall `write()`)

→ viď okno behu xv6

Ako *syscall* vstúpi do jadra

(gdb) c

→ sme späť z kernelu na konci write()

→ viď okno behu xv6

→ poďme znovu pred syscall write

(gdb) c (v okne Qemu nezabudnime dať Enter)

Ako *syscall* vstúpi do jadra

- Registre $pc a $sp sú na nízkych adresách
- Argumenty funkcií sú v registroch a0, a1, a2…
- Argumenty systémového volania write()
  - a0 je fd
  - a1 je buf
  - a2 je n

(gdb) x/2c $a1 (vypíš 2 znaky od addr v reg a1)

sh vypisuje prompt '$ '

Ako *syscall* vstúpi do jadra

- Vykonajme inštrukciu `ecall`

(gdb) si 2

- Kde sa nachádzame?

(gdb) print $pc → veľmi vysoká adresa!

(gdb) x/6i $pc

  - 6 inštrukcií, ktoré sa budú vykonávať
  - Viď `uservec` v kernel/trampoline.S
  - Ide o začiatok kódu trampolíny do jadra

Ako *syscall* vstúpi do jadra

- Všetky registre majú hodnoty z *user* programu (okrem $pc)

(gdb) info reg

- Ešte stále sa používa tabuľka stránok *user* procesu

(gdb) p/x $satp

Ako *syscall* vstúpi do jadra

- Čo sa udialo inštrukciou `ecall`?

- Procesor sa prepol do *supervisor* módu
- Ako to vieme? Vykonáva sa inštrukcia na adrese danej registrom $stvec (`ecall` zoberie hodnotu v $stvec a nastaví podľa nej $pc)
- Trampolína **nie je** prístupná pre *user* program (nie je nastavený príznak `PTE_U`)!

(gdb) p/x $stvec

Ako *syscall* vstúpi do jadra

- Čo sa udeje inštrukciou `ecall`?

1. Zmení sa mód procesora z *user* na *supervisor*
2. Uloží sa hodnota $pc do $sepc

   (gdb) p/x $sepc
3. Vykonávanie skočí na adresu uloženú v $stvec (register $pc sa nastaví na hodnotu, ktorá je v $stvec)

Ako *syscall* vstúpi do jadra

- Kernel musel nastaviť $stvec pred prechodom do *user* priestoru!!!

- `ecall` umožňuje zmeniť mód CPU z *user* na *superuser*
- Ale iba *kernel* má kontrolu nad tým, ČO sa začne v tomto režime vykonávať, a to na základe obsahu registra $stvec
- Program používateľa NEDOKÁŽE zmeniť $stvec

Ako *syscall* vstúpi do jadra

- Čo ďalšie sa musí urobiť? Zatiaľ máme „iba” zmenený mód CPU

- Uložiť registre CPU, ktoré využíva *user* (aby sa mohli neskôr obnoviť pri prechode z jadra späť k vykonávaniu kódu používateľského programu)
- Zmeniť tabuľku stránok z *user* na *kernel*
- Nastaviť zásobník pre C-čkovský kód jadra
- Skočiť na vykonávanie C kódu jadra

Ako *syscall* vstúpi do jadra

- Prečo je `ecall` tak navrhnutý, že nerobí tieto ostatné činnosti? Aby bolo možné implementovať veľmi rýchlu obsluhu výnimiek:
  - Obslúžiť niektoré výnimky bez nutnosti zmeny tabuľky stránok
  - Ak by *kernel* a *user* používali to isté mapovanie, netreba meniť tabuľku stránok
  - Ak je obsluha písaná v asm, nie je nutné používať zásobník

Ako *syscall* vstúpi do jadra

- Aké možnosti máme na RISC-V pre uloženie stavu CPU (registrov používateľa)?
- Môžeme ich len tak zapísať niekam do fyzickej pamäte? Nie, lebo je zapnuté stránkovanie!
- Môžeme najprv zmeniť $satp na *kernel* stránky?
  - *Supervisor* mód to umožňuje
  - Ale nepoznáme ADRESU tabuľky jadra úrovne L2!
  - A $satp pred prepísaním musí byť tiež odložený!

Ako *syscall* vstúpi do jadra

- Uloženie registrov pozostáva z 2 častí
  1. Z dostupného pamäťového miesta, kam sa uložia
  2. Z adresy tohto pamäťového miesta
- Xv6 mapuje do *user* stránok 2 stránky jadra, ku ktorým *user* nemá prístup: trampolínu (úplne posledná stránka virtuálneho priestoru) a pod ňou tzv. *trapframe*
- RISC-V poskytuje 1 pracovný register, ku ktorému nemá *user* prístup: $sscratch

Ako *syscall* vstúpi do jadra

- *Trapframe*
  - 1 stránka je dostatočne veľká na uloženie 32 registrov RISC-V CPU
  - Každý proces má mapovaný *trapframe* na tú istú virtuálnu adresu, ale vždy do iného rámca v RAM
  - Virtuálna adresa je 0x3ffffe000
  - Viď štruktúru *trapframe* v kernel/proc.h
- Pamäťové miesto máme, ale potrebujeme mať niekde uloženú adresu tohto miesta!

Ako *syscall* vstúpi do jadra

- Register `sscratch`
  - Pred prechodom do *user* priestoru *kernel* nastaví nielen $stvec, ale aj $sscratch register
  - Uloží do neho odkaz na *trapframe* štruktúru
  - RISC-V poskytuje inštrukciu na výmenu ľubovoľného registra s $sscratch (prehodenie hodnôt medzi registrom a $sscratch)

- Prvá inštrukcia `uservec()` v trapframe.S:

```
csrrw sscratch, a0
```


Ako *syscall* vstúpi do jadra

(gdb) p/x $sscratch

(gdb) p/x $a0

(gdb) si

(gdb) p/x $sscratch

(gdb) p/x $a0

Ďalej kód ukladá 32 registrov CPU do štruktúry *trapframe* pomocou nepriamej adresácie cez register $a0

Ako *syscall* vstúpi do jadra

až po inštrukciu `csrr t0, sscratch` (30x si)

(gdb) si

- ešte je potrebné uložiť pôvodnú hodnotu $a0 (tá sa nachádza v $sscratch)

(gdb) si

(gdb) si

- po uložení registrov je potrebné pripraviť zásobník a skočiť na vykonávanie C kódu

Ako *syscall* vstúpi do jadra

- Pozrime sa na štruktúru *trapframe* do kernel/proc.h
- Kde sa v nej nachádza adresa zásobníka jadra? V štruktúre *trapframe* od 8. bajtu
  - `ld sp, 8(a0)`
  - Nezabúdajme, že v $a0 je adresa začiatku *trapframe*; takže po pripočítaní hodnoty 8 k hodnote v registri $a0 dostaneme adresu, z ktorej sa načíta hodnota do registra $sp
  - Všetky údaje potrebné pre obnovenie behu kódu jadra sú k dispozícii v štruktúre *trapframe*

Ako *syscall* vstúpi do jadra

- Viď komentáre v kernel/trampoline.S pre ďalšie inštrukcie pri krokovaní pomocou „si”
  - Obnoví sa hw ID vlákna
  - Načíta sa adresa funkcie v C, ktorou bude pokračovať spracovanie po ukončení asm kódu
  - Načíta sa adresa tabuliek jadra do $satp, vyčistí sa TLB – prečo neprišlo ku výnimke po výmene tabuliek stránok?
    - Lebo na rovnakú virtuálnu adresu je mapovaná trampolína aj v tabuľkách jadra!!!
    - (gdb) p/x $satp

Ako *syscall* vstúpi do jadra

- Napokon možno skočiť do funkcie usertrap()
  - Jej adresa sa nachádza v registri $t0

(gdb) p/x $t0

(gdb) x/4i $t0

(gdb) si

(gdb) tui enable

- A konečne sa nachádzame v C kóde
- `usertrap()` v kernel/trap.c

Ako *syscall* vstúpi do jadra

- Úloha `usertrap()`
  - Dispečing rôznych typov prerušení, chybových stavov a systémových volaní prichádzajúcich z *user* módu CPU
  - Zistenie príčiny výnimočného stavu pomocou hodnoty registra $scause
  - Viď obrázok 10.3 na strane 102 manuálu „The RISC-V Reader”
  - $scause rovné hodnote 8 je systémové volanie (*environment call from U-mode*, čiže `ecall`)

0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store address misaligned
7	Store access fault
8	Environment call from U-mode
9	Environment call from S-mode
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault
15	Store page fault

Prevzaté a upravené z: Patterson, D.; Waterman, A. The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon LLC: San Francisco, CA, USA, 2017. Obrázok 10.3 na strane 102.

Ako *syscall* vstúpi do jadra

- Až po funkciu `syscall()`: (gdb) n

  (gdb) s

  (gdb) n

- Nachádzame sa vo funkcii `syscall()` v súbore kernel/syscall.c

Ako *syscall* vstúpi do jadra

- Funkcia `syscall()`
  - p→trapframe ukazuje na pamäťovú oblasť, v ktorej sú uchované VŠETKY registre user programu
- p→trapframe→a7 uchováva číslo systémového volania (v našom prípade hodnotu 16, čiže `SYS_write`)
- p→trapframe→a0 uchováva 1. argument (`fd`)
- p→trapframe→a1 uchováva 2. argument (`buf`)
- p→trapframe→a2 uchováva 3. argument (n)

Ako *syscall* vstúpi do jadra

(gdb) n

(gdb) p num

(gdb) n

(gdb) s

- A sme vo funkcii sys_write()
- Ďalší kód je nezaujímavý – štandardné C
- Pozrime sa, ako sa vrátime späť do *user* módu

Ako *syscall* vystúpi z jadra

(gdb) finish

- Vrátime sa späť do funkcie `syscall()`
- Funkcia `sys_write()` vykonala výpis na monitor
- Dôležité: PO vykonaní systémového volania sa výsledok uloží do p→trapframe→a0, aby sa pri obnovení obsahu registrov dostala návratová hodnota volania do registra $a0 !!!!!!!!
- Prečo $a0? Konvencia volaní C na RISC-V

Ako *syscall* vystúpi z jadra

- Z pohľadu používateľského programu je vyvolanie systémového volania volaním „obyčajnej funkcie”
- Konvencia volaní C definuje, že návratová hodnota volanej funkcie sa u volajúceho musí objaviť v registri $a0
- Preto sa návratová hodnota systémového volania v jadre „injektážou” cez `trapframe` dostane až k používateľskému kódu tiež v registri $a0

Prechod kernel→user

(gdb) fin — vrátime sa do `usertrap()`

(gdb) n

(gdb) s

Nachádzame sa vo funkcii usertrapret(), ktorá rieši návrat späť do používateľského programu

V prvom rade treba prichystať všetky údaje potrebné pre ďalší prechod z user→kernel

Prechod kernel→user

- Príprava pre ďalší prechod user→kernel

- $stvec ← uservec() kvôli ďalšiemu `ecall`
- tf->satp ←kernel page table kvôli ďalšiemu volaniu uservec()
- tf->sp ←vrchol zásobníka kernelu
- tf-> trap ←adresa funkcie usertrap()
- tf->hartid ←id vlákna (nachádza sa v $tp)

Prechod kernel→user

- Prechod do *user* módu spôsobuje inštrukcia **sret** (na architektúre RISC-V)
- Táto inštrukcia využíva na svoju činnosť viaceré registre (podobne ako `ecall`)
  - $sstatus (pomocou neho sa nastaví bit „predošlého módu CPU” na *user*)
  - $sepc (adresa inštrukcie *user* programu, ktorou sa bude pokračovať vykonávanie kódu – tú máme uloženú v p→trapframe→epc)
- Pomocou ‘n’ prejdime na posledný riadok (č. 129) funkcie

Prechod kernel→user

- Teraz by bolo vhodné zmeniť tabuľku stránok späť na používateľskú
- To sa však NEDÁ v `usertrapret()`, pretože nie je mapovaná do používateľského priestoru!
- Preto sa v `usertrapret()` na konci vypočíta adresa funkcie `userret()` v stránke trampolíny, a tam sa odovzdá riadenie

(gdb) tui disable

Pomocou ‘si’ sa v gdb presuňme na 0x3ffffff09c

(gdb) x/8i 0x3ffffff09c

Prechod kernel→user

- $a0 obsahuje adresu stránok používateľského priestoru
- Inštrukciou `csrw satp` sa nastaví tabuľka stránok používateľského procesu
- Prečo pri vykonávaní ďalšieho kódu nenastane výnimka?

(gdb) si

(gdb) si

Prechod kernel→user

- Do registra $a0 sa nastaví adresa *trapframe* a do *trapframe* sa uložia všetky registre okrem $a0

(gdb) si

(gdb) si

- Nasleduje 30 inštrukcií obnovy hodnôt registrov z *trapframe*

Prechod kernel→user

- Nasleduje 30 inštrukcií obnovy hodnôt registrov z *trapframe*
- Nakoniec sa obnoví hodnota registra $a0 a pomocou inštrukcie `sret a0` sa obnoví beh používateľského procesu v režime *user*:
  - $sstatus má nastavený *user* mód
  - $sepc obsahuje adresu inštrukcie, ktorá sa má v používateľskom procese vykonávať

Prechod kernel→user

- Nakoniec sa nachádzame pred inštrukciou `sret`, ktorá podľa registra $sstatus nastaví mód CPU a podľa $sepc obnoví hodnotu PC

(gdb) p/x $pc

(gdb) si

(gdb) p/x $pc

Prechod kernel→user

- A sme späť v používateľskom programe, po vyvolaní systémového volania `write()`
- Návratovú hodnotu volania máme v registri $a0

Zhrnutie

- Systémové volanie cez entry/exit je oveľa komplexnejšie ako obyčajné volanie fnc…
- Takáto komplikovaná vec je v dôsledku požiadavky izolácie procesov
- Natíska sa legitímna otázka: nedá sa to nejako jednoduchšie?
- Odpoveď… treba hľadať ;)

TF

- Slúži na uchovavanie stavu CPU pri prechode user→ kernel a naopak
- Viď kernel/proc.h
  - Kernel page table
  - Kernel stack pointer
  - Address of usertrap() fnc in kernel
  - User pc CPU register
  - User 32 CPU registers

Niekoľko RISC-V registrov

- Iba niekoľko najdôležitejších
- Ďalšie viď kapitola 10 v https://github.com/Lingrui98/RISC-V-book/blob/master/rvbook.pdf

Niekoľko RISC-V registrov

- stvec – *supervisor trap-vector base addr reg*
  - addr v jadre, kam skáče `ecall`; addr trampolíny
- sepc – *supervisor exceptional instruction pc*
  - `ecall` vyvolá výnimku; v tomto reg. sa uchová adresa inštrukcie, ktorá vyvolala výnimku; v našom prípade tu bude hodnota *user* pc (`ecall` je inštrukcia v používateľskom programe)
- scause – supervisor cause
  - Uložený kód príčiny vyvolanej výnimky; v prípade xv6 tam bude hodnota 8 (*system call*)

Niekoľko RISC-V registrov

- sscratch – *supervisor scratch*
  - Jedno slovo (*word*) údajov na dočasné použitie; v prípade xv6 tam je adresa *trapframe*
- satp – *supervisor address translation and protection* (riadi stránkovanie)
  - Aktuálna tabuľka stránok

Domáce čítanie

- Chapter 4: Operating system organization knižky „xv6: a simple, Unix-like teaching operating system”
- Okrem časti 4.6; tej sa budeme zvlášť venovať budúci týždeň

Prednáška 5

Výpadky stránok

Téma

- Implementácia rôznych trikov pomocou VM
- Špeciálne zameranie na výpadky stránok
  - Oneskorená (lenivá) alokácia – *lazy allocation*
  - Alokácia pri zápise – *fork COW (copy-on-write)*
  - Mapovanie súboru do pamäte – *memory mapping*

Téma

- Načo sú tieto triky dobré
- Zlepšenie výkonu/efektivity
  -
  -
  -
- Nová funkcionalita systému
  -

Téma

- Načo sú tieto triky dobré
- Zlepšenie výkonu/efektivity
  - Oneskorená alokácia
  - Jedna nulová stránka pre celý systém
  - COW fork
- Nová funkcionalita systému
  - Mapovanie pamäte (`mmap`)

Reakcia xv6 na výpadok stránky

- Ako reaguje xv6 na výpadky stránok?
- Ako je to v priestore používateľa?
- A ako v priestore jadra?

Reakcia xv6 na výpadok stránky

- `usertrap(): unexpected scause …`
- Viď príklad `user/echo.c`
- Ukončenie procesu

- Čo výpadok stránky v jadre?
- Napríklad `kernel/sysproc.c sys_exit()`

Opakovanie

- Výhody VM

1. Izolácia – každý proces má svoj vlastný adresný priestor
2. Nepriamočiarosť (*level-of-indirection*) – preklad VA na FA umožňuje triky
   - Zdieľanie dát v ramke (trampolína)
   - Stráženie pretečenia zásobníka (*guard page*)
   - …

Statické / dynamické mapovanie

- Jadro OS má kontrolu nad prekladom VA→FA

- Doteraz sme sa venovali pevne nastavenému mapovaniu (v zmysle, že mapovanie musí byť nastavené PRED prístupom k VA)

Statické / dynamické mapovanie

- VM však umožňuje aj mapovanie „za chodu”
- Pri pokuse o prístup na VA sa generuje výnimka
- Jadro OS pre danú VA vytvorí mapovanie a reštartuje proces od inštrukcie, ktorá spôsobila výnimku (proces platne pristúpi k údajom na VA)

Terminológia RISC-V

- SiFive Interrupt Cookbook
- https://sifive.cdn.prismic.io/sifive/0d163928-2128-42be-a75a-464df65e04e0_sifive-interrupt-cookbook.pdf

Terminológia RISC-V

- *Exception* (výnimka) – výnimočný stav vyvolaný inštrukciou vykonávaného programu
- *Trap* (presun riadenia) – **synchrónny** prenos riadenia do kódu obsluhy spôsobený výnimočným stavom, ktorý zapríčinil vykonávaný program
- *Interrupt* (prerušenie) – externá udalosť, ktorá sa vyskytne **asynchrónne** voči vykonávanému kódu

The RISC-V Instruction Set ManualVolume II: Privileged ArchitectureDocument Version 1.12-draft, 2020, Table 4.2, p.66

Supervisor Cause (scause) register

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Available for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

Cieľ tohto týždňa

- Pri výpadku stránky v používateľskom programe
- Zistiť, či sa jedná o legitímnu adresu programu
- Zistiť, či sa jedná o legitímny nárok prístupu
- Ak áno, aktualizovať tabuľku stránok procesu
- A napokon reštartovať proces od inštrukcie, ktorá spôsobila výpadok stránky (vykonanie inštrukcie sa musí zopakovať)

Čo potrebujeme

1. VA, ktorá spôsobila výpadok (*faulting VA*) (na RISC-V je táto hodnota uložená v registri `$stval`)
2. Typ výpadku stránky (pre RISC-V viď tabuľku 4.2 v „RISC-V privileged.pdf", hodnota registra `$scause` – *read*, *write*, *instruction*)

Supervisor Cause (scause) register

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Available for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

Čo potrebujeme

1. VA, ktorá spôsobila výpadok (`$stval`)
2. Typ výpadku stránky (`$scause`)
3. Mód CPU a inštrukciu, kde k výpadku prišlo
   - U/S mód: implicitne sa vyvolá `usertrap` alebo `kerneltrap`
   - PC: `$sepc`, hodnota uložená v `tf→epc` pri móde U

Zdroj informácií

- pre RISC-V viď tabuľka 4.2 v „RISC-V privileged.pdf" na strane 66
- https://uim.fei.stuba.sk/wp-content/uploads/2018/02/riscv-privileged.pdf

- https://uim.fei.stuba.sk/predmet/b-os/
- záložka Literatúra

Čo potrebujeme

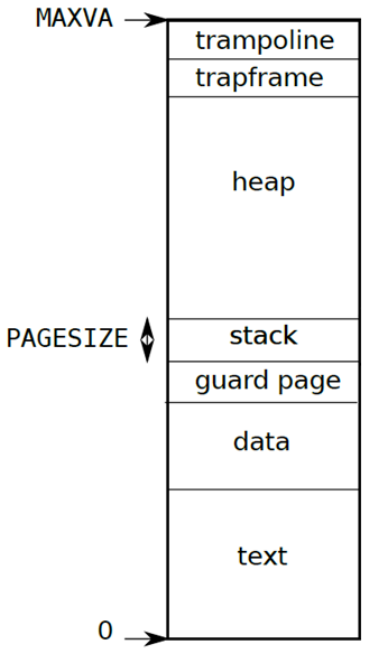
- Načo nám treba register PC?

- Aby kód jadra vedel reštartovať používateľský program presne od tej inštrukcie, ktorá vyvolala výpadok
- Inštrukcia sa musí zopakovať (jej vykonanie)
- Ak jadro všetko dobre nastavilo, pri opakovaní inštrukcie sa už výnimka neobjaví

Zoznam trikov s VM

- alokácia pamäte na žiadosť (*lazy/on-demand*)
- jedna stránka núl (*zero-filled page*)
- COW fork (*copy-on-write*)
- VA väčšia než RAM (*swap*)
- mapovanie súborov do pamäte (*mmap*)
- zdieľanie pamäte medzi procesmi (*shared*)

`sbrk`, `fork` a `exec`



Lazy alokácia v1 (pre `sbrk`)

- `sbrk()` v xv6 je primitívny – vždy procesu dá, čo chce
- Aplikácia často nevyužije všetku pridelenú pamäť
  - Napr. alokuje *buffer* na načítanie vstupu, ale väčšinou sa využije iba prvých pár bajtov
- *sbrk()* tak často alokuje pamäť, ktorá sa NIKDY v procese nevyužije, ale nikto iný ju nemôže využívať!

Lazy alokácia v1 (pre `sbrk`)

- Moderné OS alokujú pamäť oneskorene (lenivo, *lazy*)
- Ako?

Lazy alokácia v1 (pre `sbrk`)

1) Fyzická ram sa alokuje až vtedy, keď je treba
2) Pri volaní `sbrk()` sa zväčší `p->sz`, ale nerobí sa žiadna alokácia (mapovanie)
3) Keď sa proces pokúsi pristúpiť k „alokovanej” pamäti, nastane výpadok stránky!!!
4) Jadro v obsluhe výpadku alokuje a namapuje potrebnú pamäť
5) Aplikácia sa reštartuje od miesta výpadku

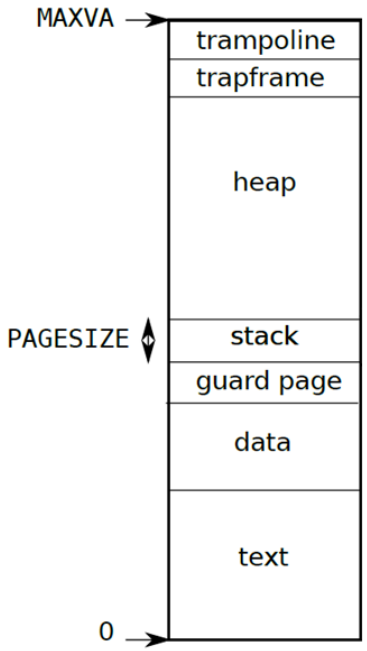
Lazy alokácia v1 (pre `sbrk`)

- Aké sú výhody takéhoto prístupu?
- Alokuje sa menej fyzickej pamäte (ak sa k pamäti z *user* programu nepristúpi, nepríde k žiadnemu výpadku, žiadna pamäť sa nebude alokovať)
- Cenou tohto prístupu je strata istej efektivity – pamäť sa musí alokovať nie pri jednom systémovom volaní (`sbrk`), ale pri každom výpadku stránky

Lazy alokácia v1 (pre `sbrk`)

- Ako na to prakticky v xv6?

1. sbrk(): p→sz = p→sz+n
2. pgfault():
   - Kontrola VA v intervale <……; ……..)
   - Alokuj rámec v ram-ke
   - Vynuluj rámec (prečo?)
   - Namapuj príslušnú stránku VA na rámec
   - Reštartuj *user* program od inštrukcie, ktorá spôsobila výpadok



Lazy alokácia v1 (pre `sbrk`)

- Ošetrenie chýb alokácie RAM pri štandardnom (nie *lazy*) `sbrk()` je triviálne
- Ako je to v prípade oneskorenej alokácie?

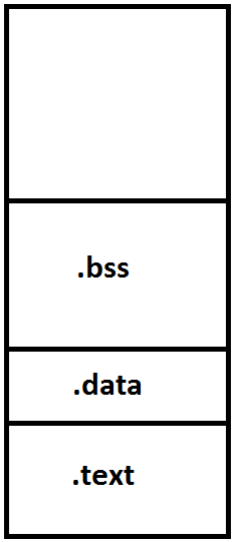
- Čo robí obsluha výpadku stránky *pgfault()?*

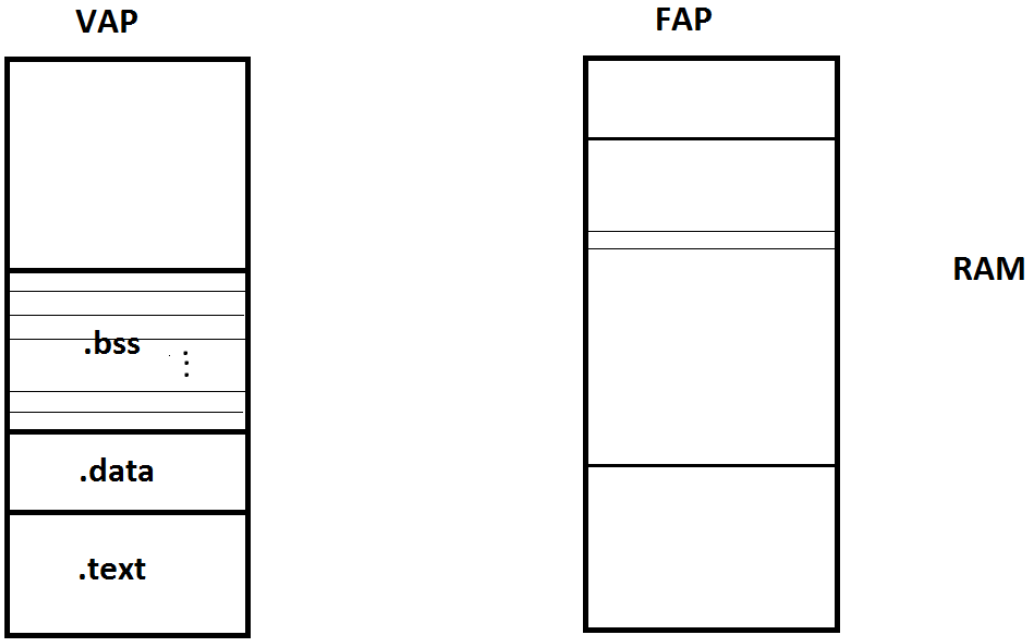
Lazy alokácia v1 (pre `sbrk`)

- Skontroluj VA v intervale <……; …….)
- Alokuj rámec v ramke
- Vynuluj rámec (prečo?)
- Namapuj príslušnú stránku VA na rámec
- Reštartuj *user* program od inštrukcie, ktorá spôsobila výpadok

Stránka núl na požiadanie

- *Zero-fill on demand*





Stránka núl na požiadanie

- Čo má robiť obsluha `pgfault()`

- Overiť, či VA ukazuje do *zero-page*
- Alokovať nový rámec
- Vynulovať ho / skopírovať ho
- Vytvoriť mapovanie do rámca pre VA (`PTE_W`)
- Reštartovať inštrukciu

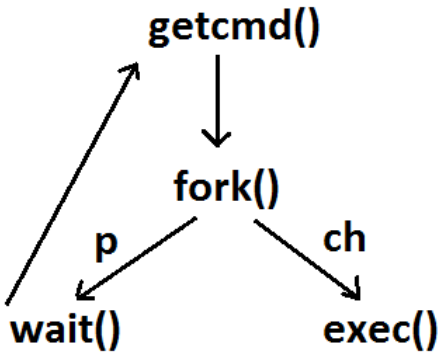
Stránka núl na požiadanie

- Načo je to dobré?

1. Program využíva iba toľko pamäte, koľko aktuálne potrebuje – podobne ako pri *lazy* alokácii
2. `exec()` je efektívnejší – netrvá tak dlho (žiadna nulová stránka sa reálne nealokuje)

Copy-on-write fork

- Implementácia je predmetom cvičenia tento týždeň
- Ako funguje spustenie príkazu v sh?



Copy-on-write fork

- `fork()` skopíruje celý VAP rodiča
- `exec()` celý VAP procesu zruší a nahradí ho VAP nového procesu

- Brutálny obrázok

Copy-on-write fork

- Cieľ COW forku – všetky VA detského procesu (totožné s VA rodiča) budú ukazovať na tie isté údaje v RAM

- Pozor na stránky, ktoré majú PTE_W!
  - Aj v rodičovi, aj v potomkovi musíme odstrániť PTE_W
  - Dôsledok: pri pokuse o zápis sa vygeneruje výnimka

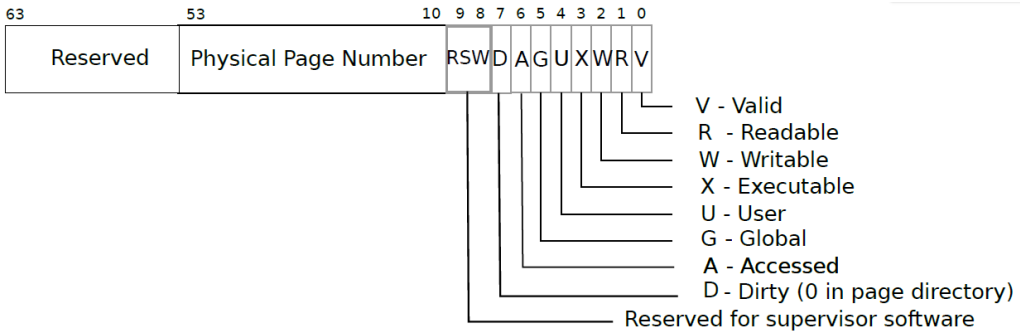
Copy-on-write fork

- Čo má urobiť `pgfault()`

- Skontroluj, či stránka zodpovedajúca VA má „nárok” byť zapisovateľná
- Alokuj rámec v RAM
- Skopíruj do neho údaje z PTE_R stránky
- Vytvor do neho mapovanie pre VA (PTE_W)
- Reštartuj inštrukciu

Copy-on-write fork

- Skontroluj, či stránka zodpovedajúca VA má „nárok” byť zapisovateľná. Ako?
- Využijeme jeden z RSW bitov PTE záznamu



Copy-on-write fork

- Skontroluj, či stránka zodpovedajúca VA má „nárok” byť zapisovateľná. Ako?
- Využijeme jeden z RSW bitov PTE záznamu
  - Pri kopírovaní VAP rodiča vo `fork()` všetkým mapovaniam `PTE_W` odstránime `PTE_W` a pridáme nami využitý bit (označme ho napr. `PTE_COW`)
  - V obsluhe `pgfault()` skontrolujeme, či mapovanie danej VA obsahuje `PTE_COW`: ak áno, `pgfault()` urobí, čo má (alokácia, kópia, mapovanie, reštart inštrukcie)

Copy-on-write fork

- Prečo musíme robiť zbytočnú operáciu alokácie/kopírovania/mapovania v prípade, že po poslednom `pgfault()` nám ostane iba jeden jediný proces, ktorý ukazuje na pôvodné data vo FAP? Viď obrázok…
- Nedá sa to urobiť nejako tak, aby sme túto situáciu vyriešili?

Copy-on-write fork

- Pri COW musíme byť opatrní na viacerých miestach, zvlášť pri uvoľňovaní stránok
- Napríklad systémové volanie `exit()`
  - Môže jadro uvoľniť stránky procesu?
  - Ani pri PTE_COW, ani pri PTE_R nevieme, koľko procesov má rámec FAP namapovaný do svojho VAP!

Copy-on-write fork

- Ako vyriešiť uvoľňovanie stránok?

- Počítadlo počtu referencií na rámec FAP
- Pri každom namapovaní sa počítadlo zvýši
- Pri každom odmapovaní sa počítadlo zníži

- Keď hodnota počítadla klesne na 0, rámec sa uvoľní

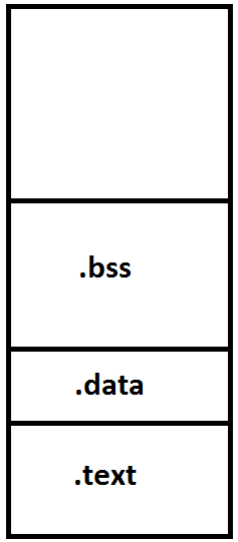
Copy-on-write fork

- Na zavedenie počítadla pre každý rámec potrebujeme vhodné datové štruktúry

- COW fork je predmetom tohto týždňa v rámci cvičení

Stránkovanie na žiadosť v2

- *On-demand paging* (`exec`)



Stránkovanie na žiadosť v2

- *On-demand paging* (`exec`)

- `exec()` nahráva kompletne celý súbor do pamäte (viď `kernel/exec.c`), čo je extrémne náročná operácia
- Disk je totiž o dosť rádov pomalší než CPU, takže CPU nemôže efektívne využiť čas na činnosť

Stránkovanie na žiadosť v2

- Cieľom je pri sys. volaní `exec()` načítať iba najnutnejšie údaje z disku, alokovať štruktúru stránok, ale nenahrávať údaje do RAM
- príslušné záznamy PTE nebudú obsahovať PTE_V

Stránkovanie na žiadosť v2

- Čo má robiť `pgfault()`
- Skontroluje, či má ísť o mapovanie do súboru
- Ak áno, alokuje rámec, načíta príslušné údaje do rámca, upraví PTE záznam pre VA, reštartuje inštrukciu

Stránkovanie na žiadosť v2

- Činnosť `pgfault()` vyžaduje nejaké metadáta o tom, kde sa stránka na disku nachádza
- Tieto informácie sú zvyčajne uložené v štruktúre nazývanej VMA (*Virtual Memory Area*)

swap

- Využitie väčšieho VAP než je RAM
- Cieľom je umožniť aplikáciám zdanie takej veľkej RAM, ako potrebujú (bez ohľadu na veľkosť samotnej RAM)

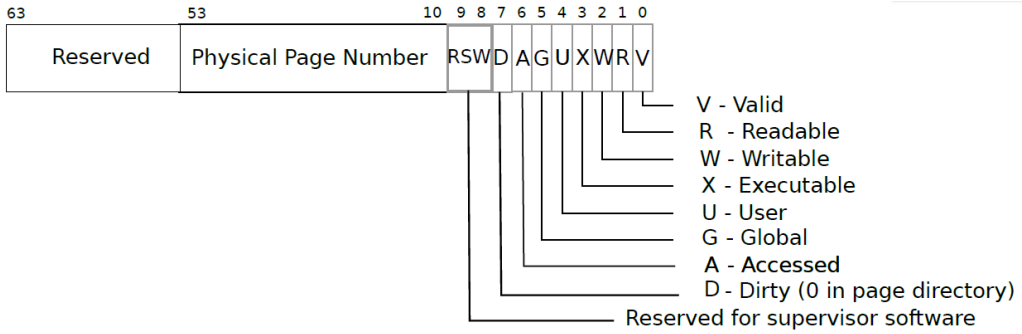
swap

- Ako to urobiť?
- Málo frekventované stránky procesu odložiť na disk, a toto miesto použiť na nové stránky, ktoré chce proces používať
- Odloženie stránky na disk a jej nahratie pri ďalšom prístupe musí byť pre proces transparentné

swap

- Ako vybrať „obeť" (*victim*), ktorá bude odložená na disk?
- Rôzne algoritmy, najčastejšie sa používa LRU (*Least Recently Used*)
- Jednoduchá verzia tohto algoritmu sa implementuje s podporou hw
  - Bit „A" v PTE zázname
  - Bit „D" v PTE zázname

swap



- Jednoduchá verzia tohto algoritmu sa implementuje s podporou hw
  - Bit „A" v PTE zázname
  - Bit „D" v PTE zázname

swap

- Bit „A" v PTE zázname
- Pri každom prístupe MMU ku stránke sa tento bit nastaví (to už vieme)
- Algoritmus výberu obete
  - Každých N tikov vynuluje bit A
  - V procese výberu obete prehľadáva stránky, a tú, ktorá nemá nastavený bit A, označí za obeť
  - Algoritmus môže zohľadňovať aj bit D

swap

- Podobne ako pri „*on-demand*" stránkovaní sú potrebné pomocné dátové štruktúry, aj v tomto prípade treba vedieť, ktoré údaje na disku zodpovedajú ktorej stránke vo VAP
- Kedy sa hľadá obeť?
- Keď nie je voľný rámec RAM, t. j. keď `kalloc()` vráti 0 (*null*)

swap

- Ako vyzerá náčrt fungovania?
- Ak nie je voľná RAM
  - Nájdi kandidáta na vyhodenie z RAM (napr. LRU algoritmus)
  - Ulož dáta „*victim*" stránky na disk
  - Zneplatni mapovanie „*victim*" stránky
  - Použi voľný rámec

Mapovanie súboru do pamäte

- `mmap()` – *memory mapped files*
- Cieľom je manipulovať s obsahom súboru nie pomocu explicitných volaní `read()`, `seek()`, `write()`
- ale pomocou inštrukcií `ld` (*load*), `st` (*store*), ktoré manipulujú s pamäťou RAM
- `mmap(va, len, protection, flags, fd, offset)`

Mapovanie súboru do pamäte

- Jadro načítava údaje zo súboru do pamäte technikou „*on-demand*" (v obsluhe `pgfault()`)
- Ak je RAM plná, napr. algoritmom LRU môže nepoužívané časti súboru uvoľniť

Mapovanie súboru do pamäte

- Systémové volanie `unmap(va, len)` zapíše do súboru iba pozmenené (využije sa „D” bit v PTE zázname) časti súboru

Mapovanie súboru do pamäte

- Na činnosť mapovania sa znovu využívajú pomocné údaje VMA (*virtual memory area*)
- Čo v prípade, že viacero procesov chce manipulovať súčasne s obsahom súboru?
- Podobne, ako keď viacero procesov súčasne používa systémové volania `read()` alebo `write()` nad tým istým súborom

Zdieľaná virtuálna pamäť

- Cieľom je umožniť procesom na rôznych uzloch siete zdieľať virtuálnu pamäť
- Vytvoriť zdanie zdieľania fyzickej pamäte

Domáce čítanie

- Chapter 4: Operating system organization knižky „xv6: a simple, Unix-like teaching operating system”
- So zameraním na časť 4.6: „Page-fault exceptions”

Prednáška 7

Zámky

Prečo nás má táto téma zaujímať?

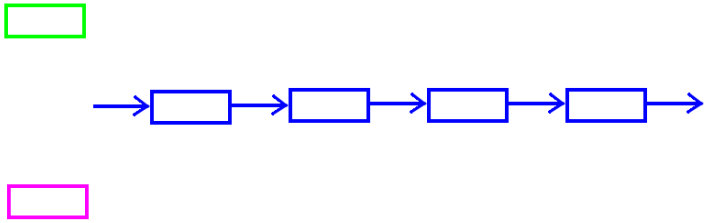
- Používateľ spúšťa viacero programov naraz
- Jadro by malo vedieť obsluhovať viacero systémových volaní súčasne
- Takže viacero tokov vykonávania súčasne môže pristupovať k rôznym dátovým štruktúram

Prečo nás má táto téma zaujímať?

- Zámky nám pomáhajú túto situáciu zvládať (spoločne to zvládneme?)
- Zámky znižujú efektivitu paralelného vykonávania (nie vždy to spoločne zvládneme!)

Prečo nás má táto téma zaujímať?

- Ukážka: zmažme `acquire()`/`release()` v `kfree()`
  - xv6 naštartuje normálne
  - Zbehnú všetky usertesty, okrem stratenia nejakých pamäťových rámcov... `` `usertests reparent2` ``
- Prečo príde k strate rámcov? Obrázok so súbehom
- Potrebujeme zámky pre zabezpečenie správnosti fungovania, ale zároveň strácame výkon (`kfree()` beží sériovo)



ADT Lock

- Už sme o ňom hovorili na minulej prednáške
- `lock l`
- `acquire(l)`
- `  x = x + 1` // KO (*critical section*)
- `release(l)`

ADT Lock

- Ak viacero vlákien (súčasne) vyvolá `acquire(l)`
  - Iba jednému vláknu sa podarí z funkcie vrátiť a pokračovať vo vykonávání ďalšieho kódu
  - Ostatné musia čakať na uvoľnenie (`release()`)
- Zámok nie je automaticky spätý so žiadnou premennou – je na programátorovi, aby samotným kódom určil, na čo bude zámok slúžiť

Kedy použiť v kóde zámok?

- Ak viac tokov vykonávania môže súčasne pristúpiť k spoločnému pamäťovému miestu a aspoň jeden z týchto tokov vykonáva operáciu zápisu na danom pamäťovom mieste
- **Nikdy** v programe **nepristupuj** k zdieľaným údajom **bez** použitia správneho zámku (každý údaj môže mať vlastný zámok)

Automatické uzamykanie

- Vyššie programovacie jazyky poskytujú ADT s automatickou podporou uzamykania
- Tento prístup nie je možné použiť na všetky prípady použitia

Automatické uzamykanie

- Tento prístup nie je možné použiť na všetky prípady použitia
  - Napr. `rename("dir1/file1", "dir2/file2")`
    - `lock(dir1), erase(file1), unlock(dir1)`
    - `lock(dir2), add(file2), unlock(dir2)`
  - Problém: súbor istý časový interval nejestvuje
  - `rename()` musí byť **atomická** operácia

Automatické uzamykanie

- Tento prístup nie je možné použiť na všetky prípady použitia
  - Napr. `rename("dir1/file1", "dir2/file2")`
    - `lock(dir1), erase(file1), unlock(dir1)`
    - `lock(dir2), add(file2), unlock(dir2)`
  - Problém: súbor istý časový interval nejestvuje
  - `rename()` musí byť **atomická** operácia:
    - `lock(dir1), lock(dir2)`
    - `erase(file1), add(file2)`
    - `unlock(dir2), unlock(dir1)`
- Programátor musí mať kontrolu nad použitím

Na čo sú zámky dobré

1. Aby sme sa vyhli stratám údajov pri ich aktualizácii
2. Aby sme dokázali z viackrokovej operácie urobiť atomickú operáciu (ukryť interný nekonzistentný medzistav)

Na čo sú zámky dobré

3. Vo všeobecnosti na zachovanie *invariantov* pri rôznych ADT
   - Invariant je platný pred začatím operácie
   - Zámky „ukryjú" dočasné porušenie invariantu
   - Na konci operácie pred uvoľnením zámku sa platnosť invariantu obnoví

Uviaznutie

- Majme verziu `rename()` s 2 zámkami

vlákno A:	vlákno B:
rename("a/1","b/2")	rename("b/3","a/4")
lock(a)	lock(b)
lock(b)	lock(a)
…	…

- Aké môže byť riešenie?

Uviaznutie

- Uzamykanie v tom istom poradí
  - Musíme vedieť, o ktoré zámky sa jedná
  - Zoradiť ich, a až potom volať `acquire()`
  - Príliš komplexné riešenie
- Programátor2 musí poznať kód, význam jednotlivých zámkov a miesta ich použitia

Zámky versus paralelné vykonávanie

- Zámky **ZNEMOŽŇUJÚ** paralelné vykonávanie
- Na umožnenie (čiastočného) paralelizmu je často nutné rozdeliť dátovú množinu (ktorú zámok chráni) na menšie časti
- Každá menšia časť bude mať vlastný zámok
- Viď úlohy cvičenia
- V zmysle delenia dátovej množiny hovoríme o granularite (hrubá versus jemná)

Zámky versus paralelné vykonávanie

- Nájdenie optimálnej granularity je netriviálna úloha
  - Celý FS / per adresár a súbor / per diskový blok
  - Celé jadro systému / každý subsystém / každý objekt
- Úloha 1 z cvičenia
  - 1 zoznam voľných rámcov RAM rozdeliť na toľko zoznamov, koľko je CPU
  - Každé CPU bude využívať svoj zoznam
  - Cieľ: môžu pristupovať k zoznamom súčasne

Granularita zámkov

- Ako nájsť správnu granularitu?
- Návrh začneme jedným veľkým zámkom (tzv. *big lock*)
  - Menšia šanca uviaznutia
  - Menej dokazovania správnosti invariantov
- Zmeriame efektivitu riešenia
  - Často riešenie s veľkými zámkami postačuje
- Jemnejšiu granularitu riešime, iba ak je to nutné (s ohľadom na zmeranú efektivitu riešenia)

Implementácia zámkov

```
1 struct lock { int locked; }
2 acquire(l) {
3    while(1) {
4        if (l->locked == 0) {
5            l->locked = 1;
6            return;
7        }
8    }
9 }
```


Implementácia zámkov

- Ako vyriešiť súbeh?
- Potrebujeme atomickú operáciu
  - Výmeny obsahu pamäťového miesta
  - Môže byť aj zložitejšia: CAS (*compare-and-swap*)
- Atomická inštrukcia `swap(addr, reg)`
  1. lock addr (globálne zamkne adresu (zbernicu))
  2. temp := *addr
  3. *addr := reg
  4. reg := temp
  5. unlock addr

Implementácia zámkov

- Implementácia ADT Spinlock pomocou CAS

```
acquire(l) {
     while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
          ;
   }
```

  - Ak `lk->locked` bolo rovné 1, inštrukcia znovu nastaví hodnotu 1; zároveň vráti hodnotu 1 (pôvodnú hodnotu pred zmenou)
  - Ak `lk->locked` bolo rovné 0, inštrukcia nastaví hodnotu 1 a vráti hodnotu 0 (pôvodnú hodnotu pred zmenou)

Poradie inštrukcií

- Doteraz sme predpokladali, že program sa vykonáva sekvenčne
  - T. j. majme sekvenciu ASM inštrukcií
  - Inštrukcie sa vykonávajú za sebou

- Toto nie je pravda
  - GCC môže optimalizáciou poprehadzovať inštr.
  - CPU môže pri vykonavaní poprehadzovať inštr.

Poradie inštrukcií

Program 1:

locked = 1

x = x + 1

locked = 0

Program 2:

while(locked == 1)

….

locked = 1

x = x + 1

locked = 0

==========

locked = 1

locked = 0

x = x + 1

Poradie inštrukcií

- Aby sme zabránili poprehadzovaniu inštrukcií narábajúcich s pamäťou (ukazateľmi), využívame ďalšiu gcc *intrinsic* funkciu `__sync_synchronize()`

- Kompilátor nemôže presunúť inštrukcie pracujúce s pamäťou za volanie tejto „funkcie”
- Kompilátor by mal vygenerovať inštrukciu „*memory barrier*” pre cieľové CPU, aby ani CPU neprehodilo žiadnu pamäťovú inštrukciu spred bariéry za ňu

Načo je dobrý ADT Spinlock

- Neustále vyťažuje na 100% CPU; je potrebný?
- Využíva sa pri čakaní na „krátky” čas
  - Krátky čas znamená kratšie, ako je réžia nutná k preplánovaniu vlákna pri ADT Sleeplock
- Zároveň pri ADT Spinlock **NESMIE** prísť ku preplánovaniu
  - Pri možnosti preplánovania by mohlo nastať uviaznutie

Uviaznutie pri držaní ADT Spinlock

- Počas `sched()` P1 drží *spinlock* L1
- Obnoví sa beh procesu P2, a ten sa pokúsi vykonať `acquire(L1)`
- Keďže `acquire()` beží s vypnutými prerušeniami
  - tak časovač nemôže doručiť prerušenie,
  - preto sa P2 nemôže vzdať CPU,
  - takže P1 nemôže byť naplánovaný na beh,
  - a teda nemôže byť uvoľnený zámok L1.
- Nastáva uviaznutie (*deadlock*)

Načo je dobrý ADT Sleeplock

- Ak je čakanie dlhšie než réžia nutná k použitiu ADT Sleeplock
- Ak je nutné preplánovanie
- Čakajúce vlákno uvoľní CPU
- Zväčša sa čaká na externé udalosti (údaje zo zariadení)
- Klávesnica, disk, sieťová karta…

Domáce čítanie

Chapter 6

Locking

xv6: a simple, Unix-like teaching operating system

Implementácia ADT Sleeplock samoštúdium

Prednáška 8

Súborový systém

Prečo potrebujeme FS?

- Perzistencia medzi reštartami systému
- Hierarchia, organizácia, pomenovanie dát
- Zdieľanie (medzi aplikáciami a používateľmi)

Čo je na návrhu FS zaujímavé

- Obnova po zlyhaní (*crash recovery*)
- Výkon (konkurentné vykonávanie)
- Zdieľanie
- Bezpečnosť
- Abstrakcia jednotného prístupu
  - /proc, /sys, …
  - rúry, schránky
  - zariadenia

Príklad systémov typu UNIX

```
fd = open("x/y.txt", …)
write(fd, "abc", 3)
link("x/y.txt", "x/z.txt")
write(fd, "def", 3)
close(fd)
```

- Súbor „y.txt” aj „z.txt” (ten istý súbor, dve rôzne cesty k jeho obsahu) obsahuje „abcdef“

UNIX FS API

- Objekty: súbory
- Obsah (údaje): pole bajtov
- Pomenovanie: ľudsky čitateľné
- Organizácia: hierarchia mien
- Synchronizácia: nie je

Niektoré dôsledky API

- fd odkazuje na niečo, čo sa zachováva, aj keď
  - sa meno súboru zmení
  - sa meno zmaže (pokým ostáva súbor otvorený)

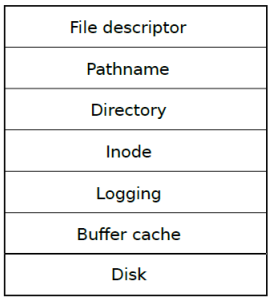


Figure 8.1: Layers of the xv6 file system.

Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Niektoré dôsledky API

- fd odkazuje na niečo, čo sa zachováva, aj keď
  - sa meno súboru zmení
  - sa meno zmaže (pokým ostáva súbor otvorený)
- Súbor môže mať viacero liniek
  - T. j. údaje sú dostupné pomocou viacerých ciest
  - Žiadna cesta nemá prioritu
- Takže info o súbore musí byť niekde inde ako v adresári

Niektoré dôsledky API

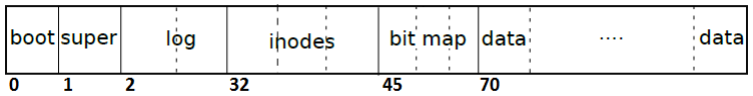
- FS uchováva info o súbore v tzv. i-uzle (*inode*) na disku
- FS odkazuje na i-uzol pomocou i-čísla (*inumber*) – ide o niečo podobné ako FD
- i-uzol musí obsahovať počítadlo referencií (liniek)
- i-uzol musí udržiavať počet otvorených FD
- Uvoľnenie i-uzla je odložené, pokým počet liniek a počet otvorených FD neklesne na 0

Disk

- Údaje sa uchovávajú na perzistentnom médiu
- Najpoužívanejšie médiá
  - HDD (veľká kapacita, ale pomalé; lacné)
  - SSD (menšia kapacita, ale rýchle; drahé)
- Najmenšia adresovateľná jednotka pri práci s diskom je 1 sektor = 512 B (viď rámec v RAM)
- OS pracuje s blokmi (násobok sektora)
  - xv6 používa bloky o veľkosti 2 sektorov (1 KiB)

Štruktúra disku xv6

- Blok 0: nepoužitý
- Blok 1: super blok (veľkosť fs, počet i-uzlov)
- Blok 2: začiatok blokov pre transakcie FS
- Blok 32: pole i-uzlov
- Blok 45: blok bitmapy využitia blokov (0=voľný blok, 1=použitý blok)
- Blok 70: začiatok blokov pre obsah súborov a adresárov



Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Štruktúra disku xv6

- Kto/čo vytvára túto štruktúru?
- Program `mkfs` (priečinok `mkfs`) (`rm fs.img`, `make`)

Štruktúra disku xv6

- Kto/čo vytvára túto štruktúru?
- Program `mkfs` (priečinok `mkfs`)
- Čo sú metadáta?
- Všetko na disku okrem samotného obsahu súborov
  - Superblok
  - i-uzly
  - Bitmapy
  - Obsah adresárov

I-uzol na disku

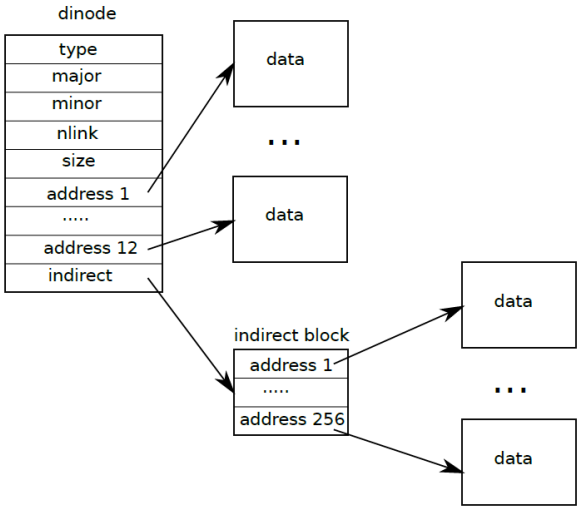


Figure 8.3: The representation of a file on disk.

Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

I-uzol na disku

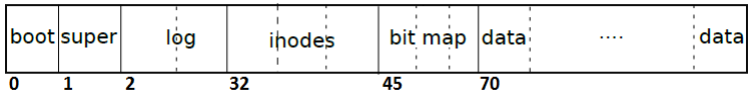
- Čo obsahuje i-uzol na disku?
  - Typ (voľný, súbor, adresár, …) (`type`)
  - Počet odkazov na súbor z adresárovej štruktúry (`nlink`)
  - Veľkosť súboru (`size`)
  - Odkazy na údaje súboru (`addrs[12+1]`)
- Obrázok priamych, nepriamych, dvojito nepriamych blokov

Výpočet logického bloku

- Ako nájsť adresu bloku disku, v ktorom sa nachádza konkrétny bajt súboru?
- Príklad: 8000-ci bajt súboru
  - Logický blok = 8000/`BSIZE` = 8000/1024 = 7
  - Položka v poli adries blokov súboru (`addr`) s ind. 7
- Príklad: 20134-tý bajt súboru
  - Logický blok = 20134/`BSIZE` = 19
  - Kde sa nachádza adresa logického bloku súboru číslo 19?

I-uzol xv6 na disku

- Ako nájsť i-uzol na disku?
- Každý i-uzol má svoje číslo **inum** (index do poľa i-uzlov na disku)
- i-uzol má veľkosť 64 bajtov
- Adresa i-uzla na disku: 32*`BSIZE` + 64*inum



Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

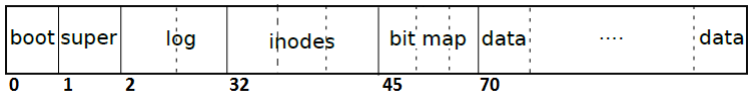
I-uzol xv6 na disku

- Štruktúra i-uzla v xv6: 64 B

- `type`: typ súboru, 2 B
- `major`, `minor`: označenie zariadenia, 2 B a 2 B
- `nlink`: počet liniek na i-uzol vo fs, 2 B
- `size`: veľkosť súboru v bajtoch, 4 B
- `addrs[NDIRECT+1]`: údajové bloky súboru, 4 B*(12+1) = 52 B

Obmedzenia FS v xv6

- Maximálny počet súborov/adresárov
- Maximálna veľkosť súboru/adresára
- Maximálna veľkosť súborového systému (disku)



Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Obsah adresára

- Adresár je tiež len súbor (t. j. pole bajtov)
  - Ale toto pole bajtov dokáže interpretovať jadro OS
  - K tomuto poľu bajtov nemá priamy prístup používateľ (iba pomocou systémových volaní na prácu s FS)

Obsah adresára

- Adresár je tiež len súbor (t. j. pole bajtov)

- Ale toto pole bajtov dokáže interpretovať jadro OS
- K tomuto poľu bajtov nemá priamy prístup používateľ (iba pomocou systémových volaní na prácu s FS)

Obsah adresára

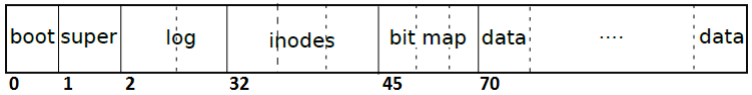
- Súbor adresára sa v xv6 interpretuje ako pole štruktúr `dirent` (kernel/fs.h)
- Veľkosť štruktúry je 16 B
  - Meno položky má iba 14 B
  - 2 B sú pre číslo i-uzla, ktorý popisuje obsah súboru
- Štruktúra je voľná, ak je `inum` rovné 0

Spracovanie cesty v xv6

- Tzv. *pathname lookup* (napr. "/x/y")
- "/" *root* → natvrdo dané číslo *root* i-uzla (1)
- V obsahu súboru, ktorý popisuje i-uzol 1, hľadaj `direntry` s menom rovným reťazcu "x"
- Ak nájdeš položku "x", v obsahu súboru, ktorý popisuje i-uzol tejto položky (v blokoch súboru "x"), hľadaj `direntry` s menom "y"
- Ak nájdeš položku "y", našiel si aj číslo i-uzla, ktorý popisuje súbor "y"

Ukážka xv6

- Zameriame sa na zápis blokov na disk
- Vytvorenie súboru
- Zapísanie do súboru
- Zmazanie súboru
- `rm fs.img && make qemu`



Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

$ echo hi > x

```
// create
bwrite: block 33 by ialloc()      // alloc inode in inode block 33
bwrite: block 33 by iupdate()     // update nlink in inode
bwrite: block 70 by writei()      // write direntry (‘x’ by dirlink())
bwrite: block 32 by iupdate()     // update size of dir inode
bwrite: block 33 by iupdate()     // itrunc() new inode
// write
bwrite: block 45 by balloc()          // alloc a block in bitmap block 45
bwrite: block 648 by bzero()      // zero the allocated block
bwrite: block 648 by writei()     // write ‘hi’ to it
bwrite: block 33 by iupdate()     // update size in inode
// write
bwrite: block 648 by writei()     // write ‘\n’ to it
bwrite: block 33 by iupdate()     // update size in inode
```

Graf volaní

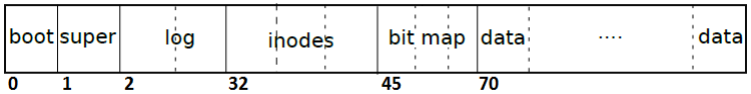
sys_open	sysfile.c	
create	sysfile.c	
ialloc	fs.c	X
iupdate	fs.c	X
dirlink	fs.c	
writei	fs.c	X
iupdate	fs.c	X
itrunc	sysfile.c	
iupdate	fs.c	X

Graf volaní

sys_unlink	sysfile.c	
writei	fs.c	X
iupdate	fs.c	X
iupdate	fs.c	X
iunlockput	fs.c	
iput	fs.c	
itrunc	fs.c	
bfree	fs.c	X
iupdate	fs.c	X
iupdate	fs.c	X

$ rm x

bwrite: block 70 by writei() // from sys_unlink; directory content
bwrite: block 32 by iupdate() // from writei of directory content
bwrite: block 33 by iupdate() // from sys_unlink; link count of file
bwrite: block 45 by bfree() // from itrunc; from iput
bwrite: block 33 by iupdate() // from itrunc; zeroed length
bwrite: block 33 by iupdate() // from iput; marked free



Zdroj: xv6: a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf

Domáce čítanie

Chapter 8

File System

xv6: a simple, Unix-like teaching operating system

Okrem kapitol a častí, ktoré hovoria o logovaní transakcií FS

Prednáška 9

Obnova FS po zlyhaní

Zlyhanie FS

- Napríklad príde k výpadku napájania
- FS môže byť v nekonzistentnom stave
- Prečo? Zápis na disk je viackroková operácia
- Riešenie – logovanie (žurnál)

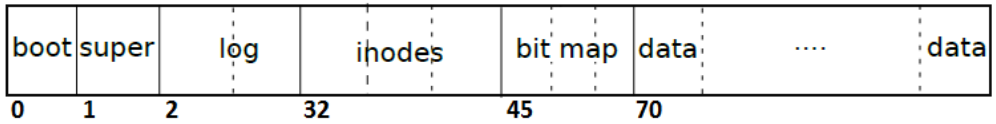
Riziká FS

- Operácie FS pozostávajú z viacerých krokov (nie sú atomické)
  - Vytvorenie adresára alebo súboru
  - Zmazanie súboru
  - …
- Zlyhanie pri vykonávaní operácie nad FS môže zanechať FS v nekonzistentnom stave (porušený invariant FS)

Riziká FS

- Čo sa môže diať po reštarte (po zlyhaní)
- Môže prísť znovu k zlyhaniu
- Nemusí prísť k zlyhaniu, ale používateľ pomocou operácií čítania/zápisu nemusí dostať tie údaje, ktoré požaduje

Príklad



$ echo “hi” > x

1: write: 33 allocate inode for x

2: write: 33 init inode x

3: write: 70 record x in / directory’s data block

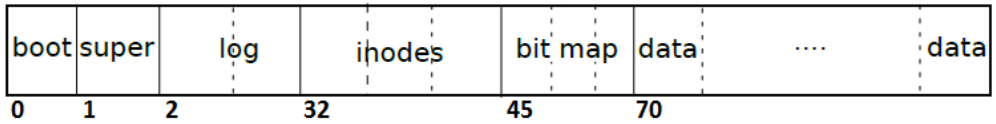
4: write: 32 update root inode

5: write: 33 update inode x

Pozorovanie

- Prehodenie poradia vykonávania zápisov nerieši problém
- Prehodením vyriešime jeden problém, ale vytvoríme ďalší (minimálne jeden)

Príklad



Zapísanie "hi\n" do súboru x

1: write: 45 set alloc bit in bitmap block

2: write: 644 write "hi" into file’s data block

3: write: 644 write "\n" into file’s data block

4: write: 33 update inode x (size, addr[0])

Pozorovanie

- Prehodenie poradia vykonávania zápisov nerieši problém
- Prehodením vyriešime jeden problém, ale vytvoríme ďalší (minimálne jeden)
- Riešenie: logovanie (žurnál)

Logovanie

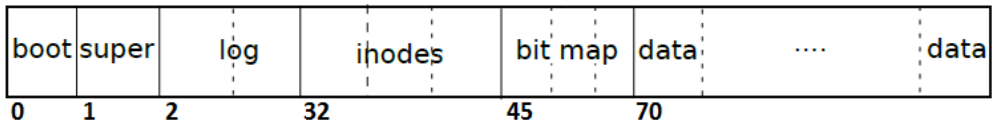
- Atomicita systémového volania nad FS
- Rýchla obnova po zlyhaní
- Zachovanie výkonu FS
- Logovanie xv6 rieši prvé dva body, tretí nie

Logovanie xv6

- Zachováva atomicitu komplexnejšej operácie nad FS – buď sa urobia všetky zápisy alebo žiaden
- Kde sa nachádza začiatok blokov logu xv6?
- Aká je štruktúra logu? (obrázok na disku a v pamäti)

Logovanie v xv6

1. Logujú sa bloky, ktoré sa zapisujú
2. Na začiatok blokov sa vkladá potvrdenie (*commit*) operácie (počet blokov, ktoré sa majú zapísať)
3. V treťom kroku sa robí inštalácia blokov, ktoré sa nachádzajú v logu
4. Na záver sa log vyčistí (počet blokov, ktoré sa majú zapísať, sa nastaví na 0)



Princíp logovania xv6

- Pri zápise (`bwrite`) na disk sa pridá číslo bloku do poľa čísiel blokov v logu v pamäti (nie na disku!)
- Samotné údaje bloku sú v `bcache` (`pin`/`unpin`)

Princíp logovania xv6

- Čo sa deje pri vyvolaní operácie *commit*

- Zapísanie bufs z bcache do logu na disku
- WAIT na dokončenie zápisov (synchrónnosť)
- Zapísanie hlavičky logu (1 blok) na disk
  - Nenulové N
  - Čísla blokov, ktoré sa majú inštalovať

Princíp logovania xv6

- Po operácii *commit*

- Inštalácia (zápis) obsahu blokov na ich miesta na disku
- Zavolanie `unpin()` na tieto bloky
- Zapísanie N = 0 do hlavičky logu na disku

Princíp logovania xv6

- Hodnota ‘N’ v hlavičke logu na disku znamená platný *commit*

- 0 znamená, že k potvrdeniu transakcie neprišlo, a teda nič sa nebude inštalovať
- Nenulová hodnota znamená, že potvrdenie nastalo, obsah logovacích blokov je platný, a je potrebné dokončiť transakciu

Kód xv6

- Viď výstup xv6 po modifikácii `bwrite`

- Viď kód `sysfile.c:sys_write()`
- Viď kód `file.c:filewrite()`
  - Vypočíta maximálny počet blokov na 1 zápis (aby sa ich počet zmestil do logu)
  - `begin_op()`
  - `writei()`
  - `end_op()` volá samotný `commit()`

Kód xv6

- Čo robí `filewrite()` spolu s `writei()`

`begin_op()`

`bmap()` – môže zapísať bitmap, indir block

– `bzero()` vyvolá `log_write()`

`bread()`

modifikácia `bp->data`

`log_write()` absorbovanie `bzero`

`log_write()`

`iupdate()` – zápis i-uzla

`end_op()`

Kód xv6

- `begin_op()`

- Kontrola prebiehajúceho commitu
- Kontrola miesta v logu na disku
- Označenie bloku operácií, ktoré musia byť atomické (zvýšením hodnoty premennej `outstanding`)

Kód xv6

- `log_write()`

- Ak je blok v logu, nepridáva sa (**absorbovanie**)
- `pin()` bloku, ktorý sa má pridať do logu
- Pridanie čísla bloku do poľa v hlavičke v pamäti
- Zvýšenie počtu blokov na zápis v hlavičke v pamäti

Kód xv6

- `end_op()`

- Ak neprebieha žiadna ďalšia operácia, ktorá patrí do atomickej transakcie (`outstanding` je 0), robí sa `commit()`

Kód xv6

- `commit()`

- Zapísanie blokov z `bcache` do logu na disk
- Zapísanie hlavičky logu (čísla blokov a N) na disk
- Inštalácia blokov z logu na disku na príslušné miesto na disku
  - V rámci inštalácie `bunpin()` blokov
- Vymazanie hlavičky logu na disku (vynulovanie N)

Logovanie v xv6

- Je primitívne a pomalé
- Predpokladá, že zariadenie disku je bezchybné

- Výzvy, s ktorými sa treba vysporiadať pri programovaní aj takto jednoduchého logovania

Výzvy návrhu logovania

- Blok, ktorý treba logovať, sa nesmie vyhodiť z `bcache`
- Veľkosť dát systémového volania sa musí zmestiť do log záznamov na disku
- Umožniť konkurentné vykonávanie viacerých systémových volaní
- Jeden blok môže byť zapisovaný na disk počas jednej transakcie viackrát

Výzvy návrhu logovania

- Blok, ktorý treba logovať, sa nesmie vyhodiť z `bcache`
  - Pred pridaním do transakcie sa zistí dostatok voľného miesta v blokoch logu
  - Urobí sa `pin()` (zvýši sa počet referencií v `bcache`)
  - Po operácii `commit` sa robí `unpin()`
- Veľkosť dát systémového volania sa musí zmestiť do log záznamov na disku
  - Ak sa dáta nezmestia do transakcie, čaká sa
  - Ak sa nezmestia do jednej transakcie, dáta sa rozdeľujú na viac atomických zápisov → **?????**

Výzvy návrhu logovania

- Umožniť konkurentné vykonávanie viacerých systémových volaní
  - Vstúpiť do transakcie je možné, počet aktívnych sys. volaní v transakcii udržiava premenná `outstanding`
  - Ak nie je miesto v logu, sys. volanie nemôže vstúpiť do transakcie; čaká na ukončenie predošlej
- Jeden blok môže byť zapisovaný na disk počas jednej transakcie viackrát
  - Technika absorbovania zápisu
  - Blok v `bcache` odráža stav (viacerých) transakcií bez operácie `commit`; inštalácia bloku však nastáva až po op. `commit`, keď nie je žiadna transakcia aktívna

Pozitíva návrhu logovania xv6

- Korektnosť vďaka pravidlu *write-ahead*: nezapisuj zmeny na disk, kým nepotvrdíš transakciu v logu
- Priemerná priepustnosť disku: vďaka logovaniu sú zápisy na disk dávkované (nie sú po jednom)
- Konkurentné vykonávanie systémových volaní je obmedzené veľkosťou dát, ktoré sa pomocou nich zapisuje

Negatíva návrhu logovania xv6

- Problém operácií, ktoré sa nezmestia na 1x do logu (zápis veľkých súborov NIE je atomický)
- Celková efektivita riešenia je slabá
  - Každý blok sa na disk zapisuje 2x (raz do logu, druhý raz pri inštalácii na svoje miesto)
  - Logujú sa celé bloky, aj keď sa zmení pár bitov
  - Zapisovanie logu prebieha synchrónne (čaká sa na dokončenie operácie zápisu na disk)

Domáce čítanie

Chapter 8

File System

xv6: a simple, Unix-like teaching operating system

Tie časti, ktoré hovoria o logovaní transakcií FS

Prednáška 10

Prerušenia

Téma

- Stlačenie klávesy
- Pohyb myšou
- Tik časovača
- Údaje pripravené na vstupe sieťovej karty
- …

- HW si vyžaduje OKAMŽITÚ pozornosť!

Téma

- CPU musí
  - Odložiť aktuálnu činnosť (uložiť aktuálny stav)
  - Obslúžiť hw (obslúžiť prerušenie)
  - Obnoviť vykonávanie činnosti pred prerušením
- Na spracovanie prerušení na RISC-V sa používa ten istý mechanizmus ako pre
  - Systémové volania (*syscalls*)
  - Výnimky (*exceptions*)

Komplikácie prerušení

- Prerušenia sú **asynchrónne**

- Viac **konkurentne** vykonávaných vecí

- Programovanie **zariadení**

Komplikácie prerušení

- Prerušenia sú **asynchrónne**
  - Kód vykonávaný na CPU pred príchodom prerušenia nijako nesúvisí s prerušením! Nie je medzi ním a prerušením žiadna kauzalita
  - Kód obsluhy prerušenia nebeží v „kontexte" procesu (v xv6 nemá zmysel využívať myproc())
- Viac **konkurentne** vykonávaných vecí
  - CPU vykonáva kód, zároveň sa niečo deje na zariadení
- Programovanie **zariadení**
  - Môže byť značne zložité naprogramovať obsluhu

Zdroje prerušení

- Vodiče zo zariadení napojené na špeciálnu zbernicu (buď priamo do CPU alebo cez čip, ktorý ďalej spracúva zdroj prerušenia)
- Na SiFive základnej doske (RISC-V CPU) prerušenia zariadení idú cez obvod PLIC
- PLIC ďalej smeruje vzniknuté prerušenie na to jadro CPU, ktoré môže prerušenie obslúžiť
  - CPU môže mať vypnuté spracovanie prerušení
  - Ak nie je žiadne CPU dostupné, PLIC uchováva prerušenie, pokým nejaké CPU nezapne obsluhu

Mechanizmus obsluhy prerušení

- Prerušenie informuje jadro o tom, že nejaký hw vyžaduje pozornosť
- Ovládač (kód v jadre) vie, ako obsluhu zariadenia uskutočniť
- Najjednoduchšia obsluha je priame volanie ovládača z obsluhy prerušenia (tak to robí xv6), ale je možná aj sofistikovanejšia schéma – pre obsluhu sa vytvorí a naplánuje vlastné vlákno jadra, prípadne sa obsluha viacerých prerušení spojí do jednej, atď.

Mechanizmus obsluhy prerušení

- Obsluha prerušenia NEBEŽÍ vždy v kontexte procesu
- Čo to znamená pre xv6
  - `myproc()` môže vrátiť `0`
  - `copyin()`, `copyout()` sa nedajú použiť
  - Prečo?

Programovanie zariadenia

- Zväčša sa používa mapovanie pamäte
- Pomocou virtuálnych adries je možné pristupovať priamo k interným registrom (pamäti) samotného zariadenia
- Priamo sa používajú inštrukcie `load`/`store`
- Programovanie UART viď napr. na: byterunner.com/16550.html

Prípadová štúdia xv6: $ ls

- Výpis znaku $ na konzolu
  - Ovládač pošle znak do FIFO **odosielacej** fronty UART zariadenia
  - UART vygeneruje prerušenie, keď sa znak pošle, čím informuje ovládač, že môže poslať ďalší znak
- Načítanie a výpis 'ls'
  - Používateľ stlačí klávesu, čo spôsobí prerušenie UART
  - Ovládač načíta znak z FIFO **prijímacej** fronty UART

Ako jadro rozozná zariadenia

- Každé zariadenie má jedinečné číslo zdroja IRQ (*Interrupt ReQuest*)
- IRQ je definované hw platformou (medzi platformami sa zväčša IRQ čísla líšia)
  - V Qemu má `UART0` pridelené IRQ 10 (viď `kernel/memlayout.h`)
  - Na doske SiFive má `UART0` iné IRQ číslo

Podpora prerušení na RISC-V CPU

- `sie` (*supervisor interrupt enable register*)
  - bity pre sw prerušenie, externý zdroj a časovač
- `sip` (*supervisor interrupt pending register*)
  - bity pre sw prerušenie, externý zdroj a časovač
- `sstatus` (*supervisor status register*)
  - jeden bit určujúci, či sú prerušenia zapnuté

Podpora prerušení na RISC-V CPU

- `scause` (*supervisor cause register*)
  - Číslo prerušenia
- `stvec` (*supervisor trap vector register*)
  - Adresa kódu obsluhy prerušení
- `mideleg` (*machine interrupt delegate register*)
  - Všetky výnimky (teda aj prerušenia) sa štandardne obsluhujú obsluhou v **M-móde**
  - Register `mideleg` umožňuje nastaviť automatické smerovanie prerušení do obsluhy v **S-móde**

Inicializácia prerušení v xv6

kernel/start.c start()

  w_sie(r_sie() | SIE_SEIE|SIE_STIE|SIE_SSIE)

kernel/main.c main()

  consoleinit()

    uartinit()

  plicinit()/plicinithart()

  scheduler()

    intr_on()

      w_sstatus(r_sstatus() | SSTATUS_SIE)

Zobrazenie ‘$’

- user/init.c main()
  - Init otvára fd 0, 1, 2 pre konzolový vstup/výstup
  - Shell ich pomocou mechanizmu fork() zdedí
- Všetky zariadenia sa v systémoch typu UNIX interpretujú ako súbory
  - printf() → putc() → write()

Zobrazenie ‘$’

sys_write()

  filewrite()

    consolewrite() v kernel/console.c

      uartputc()

- Vloženie znaku do FIFO UART
- Návrat do *userspace*
- Zároveň v tom istom čase UART posiela znak na konzolu

Zobrazenie ‘$’

- Shell v getcmd() vyvolal sys_read(), čaká na vstup
- UART po dokončení posielania znaku na konzolu vygeneruje prerušenie
- PLIC posunie prerušenie nejakému jadru CPU
- Čo urobí CPU?

Čo robí CPU pri prijatí prerušenia?

1. Ak ide o *trap* zo zariadenia a SIE bit je 0, nepokračuj v spracovaní
2. Vypni spracovanie prerušení vynulovaním SIE
3. Skopíruj `$pc` do `$sepc`
4. Uchovaj aktuálny mód (*user* alebo *supervisor*) do bitu SPP v `$sstatus`
5. Nastav `$scause` podľa zdroja prerušenia
6. Nastav mód CPU na *supervisor*
7. Skopíruj `$stvec` do `$pc`
8. Pokračuj vykonávaním inštrukcie podľa `$pc`

Čo robí CPU pri prijatí prerušenia?

- `$stvec` obsahuje buď adresu `kernelvec()` alebo `uservec()` (podľa toho, či je prerušenie vyvolané z *user* alebo *kernel* priestoru)
- Rovnaký mechanizmus sa používa aj pre výnimky a inštrukciu `ecall` (systémové volanie)

Čo robí CPU pri prijatí prerušenia?

kerneltrap()/usertrap() volajú devintr()

ak ide o externé prerušenie

plic_claim() zistí, o ktoré zariadenie ide

ak UART, uartintr()

pokým je znak na vstupe, vypíš ho

pošli na výstup aj znaky z vyrov. pamäte

plic_complete()

return z kernelvec()/uservec() obnoví prerušené vykonávanie kódu

Viaceré prerušenia súčasne

- Čo keď sa v jednom čase vyskytne viacero prerušení?
- PLIC zabezpečuje, že každé zariadenie môže vygenerovať iba 1 prerušenie, pokým nie je obsluha dokončená
- To znamená, že súčasne sa môžu vyskytnúť prerušenia od rôznych zariadení

Viaceré prerušenia súčasne

- PLIC dokáže prideliť (alebo skôr CPU si dokáže prevziať od PLIC vo funkcii plic_claim()) spracovanie prerušenia rôznym jadrám CPU
- Takže spracovanie viacerých prerušení MÔŽE prebiehať súčasne! (t. j. **paralelne**)
- Ak žiadne jadro CPU neprevezme prerušenie na spracovanie, prerušenie ostáva nespracované (angl. *pending*), pokým ho niektoré jadro nespracuje

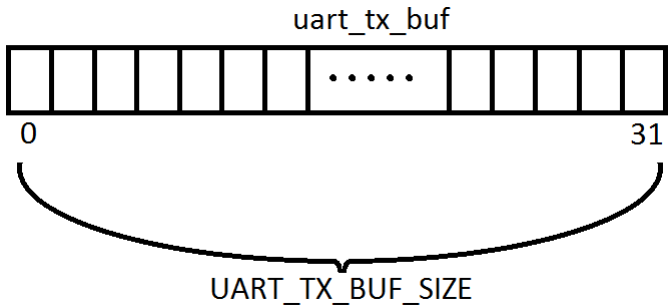
Prerušenia a konkurentnosť

- Prerušenia vnášajú problematiku viacerých typov konkurencie (angl. *concurrency*) vykonávania činnosti

1. Medzi zariadením a CPU (problém producent/konzument)
2. Prerušenie používateľského programu OK, ale čo prerušenie kódu jadra? Napr. userret() (riešenie – vypínanie prerušení, aby sme dosiahli atomicitu operácie)
3. Paralelné vykonávanie rôznych častí kódu využívajúce tú istú pamäť (riešenie – zámky)

1. Producent/konzument

- Napríklad vypisovanie na monitor
  - Shell je producent
  - Zariadenie UART je konzument



1. Producent/konzument

- sys_write()→…→uartputc()→uartstart()
  - Vkladá do `uart_tx_buf`
  - Ak je *buf* plný, čaká (stav procesu *SLEEPING*)
  - **Beží v kontexte procesu**!!! (systémové volanie)

- devintr()→…→uartintr()→uartstart()
  - Vyberá z `uart_tx_buf` a posiela na zariadenie
  - Ak je *buf* prázdny, nič neurobí
  - Budí z čakania producentov!
  - Vyvolané z devintr(), **nebeží v kontexte procesu**!!!

1. Producent/konzument

- Už vieme, ako sa vypíše prompt '$ '
- Teraz sa pozrieme na načítavanie znakov (napr. 'ls')

- Načítanie znakov z klávesnice
  - *shell* je konzument (`sys_read()`)
  - klávesnica je producent (pri stlačení sa generuje hw prerušenie)
  - Príslušný kód xv6 v kernel/console.c

1. Producent/konzument

- sys_read()→fileread()→consoleread()
  - Využíva cons.buf
  - Ak je *buffer* prázdny, proces sa uspí
  - **Volá sa v kontexte procesu**!!!

- devintr()→…→uartintr()→consoleintr()
  - Vždy **mimo kontext procesu**!!!
  - Vkladá do cons.buf (čo ak je *buf* plný?)
  - Budí konzumentov čakajúcich na vstup z klávesnice
    - Za akej podmienky nastane prebudenie?

2. Prerušenie preruší bežiaci kód

- Čo v prípade, keď nejaké prerušenie preruší vykonávaný kód?

- Majme napríklad kód, ktorý alokuje na zásobníku miesto a uloží tam návratovú adresu:
  1. addi sp, sp, -48
  2. sd ra, 40(sp)
- Môže sa vykonať nejaký kód MEDZI riadkami 1 a 2?
  - Áno, obsluha prerušení! Napríklad časovač, uart...

2. Prerušenie preruší bežiaci kód

- Čo „hrozí” v takom prípade používateľskému procesu?
  - Vykonávanie obsluhy pobeží v priestore jadra
  - Stav používateľského programu bude obnovený v tej podobe, v akej bol pri vyvolaní výnimočného stavu spôsobeného prerušením
- Čo „hrozí” kódu jadra? Je to podobne jednoduché ako v prípade používateľského kódu?

2. Prerušenie preruší bežiaci kód

- Majme nasledovné kódy jadra: bežiaci kód jadra a kód vykonávaný obsluhou prerušenia

```
bežiaci_kód:            obsluha_prerušenia:
    x = 0
    if (x == 0)             x = 1
       f()
```

- Ako je to s volaním funkcie **f()**?

2. Prerušenie preruší bežiaci kód

- Aby sa funkcia f() vždy zaručene vykonala, nesmie nastať prerušenie!
- Prečo?
  - Obsluha prerušenia môže byť vyvolaná iba na hranici inštrukcií, nie uprostred vykonávania nejakej inštr.
  - Predpokladáme, že nastavenie premennej (x = 0) a testovanie premennej (if x == 0) sú minimálne 2 inštrukcie
  - Ak je na CPU povolená obsluha prerušení, medzi týmito dvoma riadkami kódu sa môže spustiť obsluha prerušenia, ktorá zmení hodnotu premennej x!

2. Prerušenie preruší bežiaci kód

- Aby sa funkcia f() vždy zaručene vykonala, nesmie nastať prerušenie!
- Ako zabezpečiť „atomické” vykonanie nejakého bloku inštrukcií (riadkov kódu)?
- Vypnutím spracovania inštrukcií
  - Viď funkcia kernel/riscv.h: intr_off()
  - w_sstatus(r_sstatus() & ~SSTATUS_SIE);
- Kde v kóde sa táto funkcia využíva? Môže jadro obsluhovať prerušenie v kóde trampolíny?

2. Prerušenie preruší bežiaci kód

- Vráťme sa k príkladu načítania vstupu z klávesnice. V akej funkcii sa nachádza kód jadra?

$ (shell je v sys_read(), aby získal vstup z kláv.)

usertrap() – vyvolané systémovým volaním (`ecall`)

w_stvec((uint64)kernelvec) !!!!!!!

…

consoleread()

sleep()

scheduler()

intr_on() // kde sa volá?

2. Prerušenie preruší bežiaci kód

$ l (používateľ stlačil klávesu ‘l’, UART prerušenie)

kernelvec() – pretože $stvec obsahuje túto adresu!!!

– na aký zásobník sa uložia registre CPU?

2. Prerušenie preruší bežiaci kód

kernelvec()

kerneltrap()

devintr()

uartintr()

c = uartgetc()

consoleintr(c)

obsluha špeciálnych sekvencií (ctrl)

poslanie znaku ‘l’ na výstup (uartputc_sync() v consputc())

vloženie c do cons.buf

zobudenie konzumentov čakajúcich v consoleread()

návrat z consoleintr()

návrat z uartintr()

návrat z devintr()

návrat z kerneltrap()

obnovenie stavu registrov CPU

sret

KAM sa vráti tok vykonávania inštrukciou `sret`?

2. Prerušenie preruší bežiaci kód

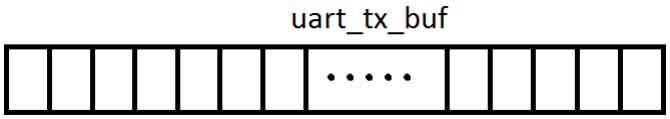
- Kam sa vráti tok riadenia inštrukciou `sret`?
- Tam, kde prišlo k prerušeniu bežiaceho kódu pri príchode prerušenia
- V našom prípade to je cyklus vo funkcii `scheduler()`
- Viď pomocou `make CPUS=1 qemu-gdb`

3. Konkurentný prístup k údajom

- Poslednou úrovňou konkurencie je prístup k tým istým pamäťovým oblastiam konkurentne/paralelne z rôznych tokov vykonávania kódu

3. Konkurentný prístup k údajom

- Príklad
  - majme dva používateľské programy, každý sa vykonáva na samostatnom jadre CPU
  - nech sa oba programy v tom istom časovom okamihu pokúsia vykonať printf(“ahoj %d\n”, pid)
  - kernel/uartc.c: uartputc()



3. Konkurentný prístup k údajom

- Riešenie – použitie zámkov (angl. *lock*)
- Viď funkcie `acquire()` a `release()` v kernel/uart.c: `uartputc()`
- Opakovanie: zámky sa využívajú na vynútenie vykonávania istej časti kódu iba jediným tokom riadenia

Vývoj prerušení

- Kedysi bol tento prístup navrhnutý a vyvinutý, aby urýchlil činnosť CPU
- V súčasnosti sú prerušenia príliš pomalé pre niektoré zariadenia
  - Napr. gigabit ethernet dokáže preniesť 1.5 milióna paketov za sekundu
  - To je viac než 1 za mikrosekundu
  - Spracovanie prerušenia trvá rádovo v mikrosekundách
  - Ako potom takéto zariadenia obsluhovať?

Vývoj prerušení

- Ak je obsluha prerušenia príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa”, tzv. *polling*
- CPU neustále v cykle kontroluje, či niektoré zariadenie nevyžaduje pozornosť
  - Toto čakanie v cykle je neefektívne (nevyužije sa CPU naplno), ak je zariadenie pomalé
  - Jeden príklad v xv6: uartputc_sync()
  - Ale ak je zariadenie mega super rýchle, šetrí sa čas CPU (žiadna zmena kontextu atď.)

Vývoj prerušení

- Ak je obsluha prerušenia príliš pomalá klasickým prístupom, je možné využiť techniku „dopytovania sa”, tzv. *polling*
- Prečo alebo kedy používať túto techniku?
- Ak je generovanie udalostí tak rýchle, že musia neustále čakať na spracovanie – vtedy nie je nutné o vygenerovaní udalosti informovať, pretože vieme, že vždy je k dispozícii nejaká udalosť čakajúca na spracovanie

Prerušenia vs *polling*

- Pre zariadenia, ktoré chŕlia udalosti – *polling*
- Pre pomalé zariadenia (typu klávesnica) – *irq*
- Automatické prepínanie medzi oboma módmi činnosti
- Presmerovanie spracovania prerušení do používateľského priestoru
  - Výpadky stránok
  - Obsluha nejakých zariadení (napr. disk, sieť)

Domáce čítanie a pozeranie

Chapter 5

Interrupts and device drivers

xv6: a simple, Unix-like teaching operating system

https://www.youtube.com/watch?v=Fcjychg4Tvk