OS MMXXIII

MIT ;)

https://pdos.csail.mit.edu/6.828/2023

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 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 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 5.3 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 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 6.0-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

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	5174	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

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

Hodnotenie (2)

- Bonusy a malusy podľa ľubovôle

- 10 x 2b bleskovka na cvičení (napr. začiatok)
- 1 x 8b, 2 x 16b (nejestvujúci) zápočet
  - 3. týždeň (8b)
  - 7. týždeň (16b)
  - 12. týždeň (16b)
- 1x 40b skúška

- Minimum z cvičení 20b (zo 60b, bez bonusov)
- Čo keď nevyhovuje termín cvičenia?

3. časť: ÚVOD DO OS

Očakávania od OS

Očakávania od OS

- Podpora viacerých vecí SÚČASNE

Očakávania od OS

- Podpora viacerých vecí SÚČASNE
- Zdieľanie a prerozdeľovanie zdrojov hw (CPU pre procesy, pamäť, disk, tlačiareň, myš, monitor, sieťová karta...)

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)

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

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“)

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

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()`)

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?

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

Užívateľský priestor

- Silná izolácia si vyžaduje presne definované API (prístupové body k službám jadra)

Užívateľský 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

Užívateľský 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

User / Kernel space

- Moderné CPU poskytujú minimálne 2 režimy činnosti:
  - **Kernel** mód (všetky inštrukcie CPU povolené), napríklad priamy prístup k zariadeniam

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 užívateľa (**„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 užívateľa, 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

Systémové volania

- Príklady systémových volaní ukázané v xv6
- Xv6 má tradičný návrh podľa UNIX

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?

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 ;)

SV read/write

- Prvý príklad systémových volaní: `copy.c`
- Kopíruje vstup na výstup ;)
- Napísaný v jazyku C

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*)

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!!!

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, …)

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

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

SV read/write

- Druhý argument je pamäťové miesto
  - `read()`: kam sa uložia údaje načítané **ZO** súboru

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

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

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 užívateľskom programe získa/nastaví hodnota premennej `fd`)?

SV open

- Príklad: `open.c`, vytvorenie súboru

SV open

- Príklad: `open.c`, vytvorenie súboru
  - `./open`
  - `./cat output.txt`

SV open

- Príklad: `open.c`, vytvorenie súboru
  - `./open`
  - `./cat output.txt`
- FD je malé celé číslo, štartuje sa od 0

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)

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)

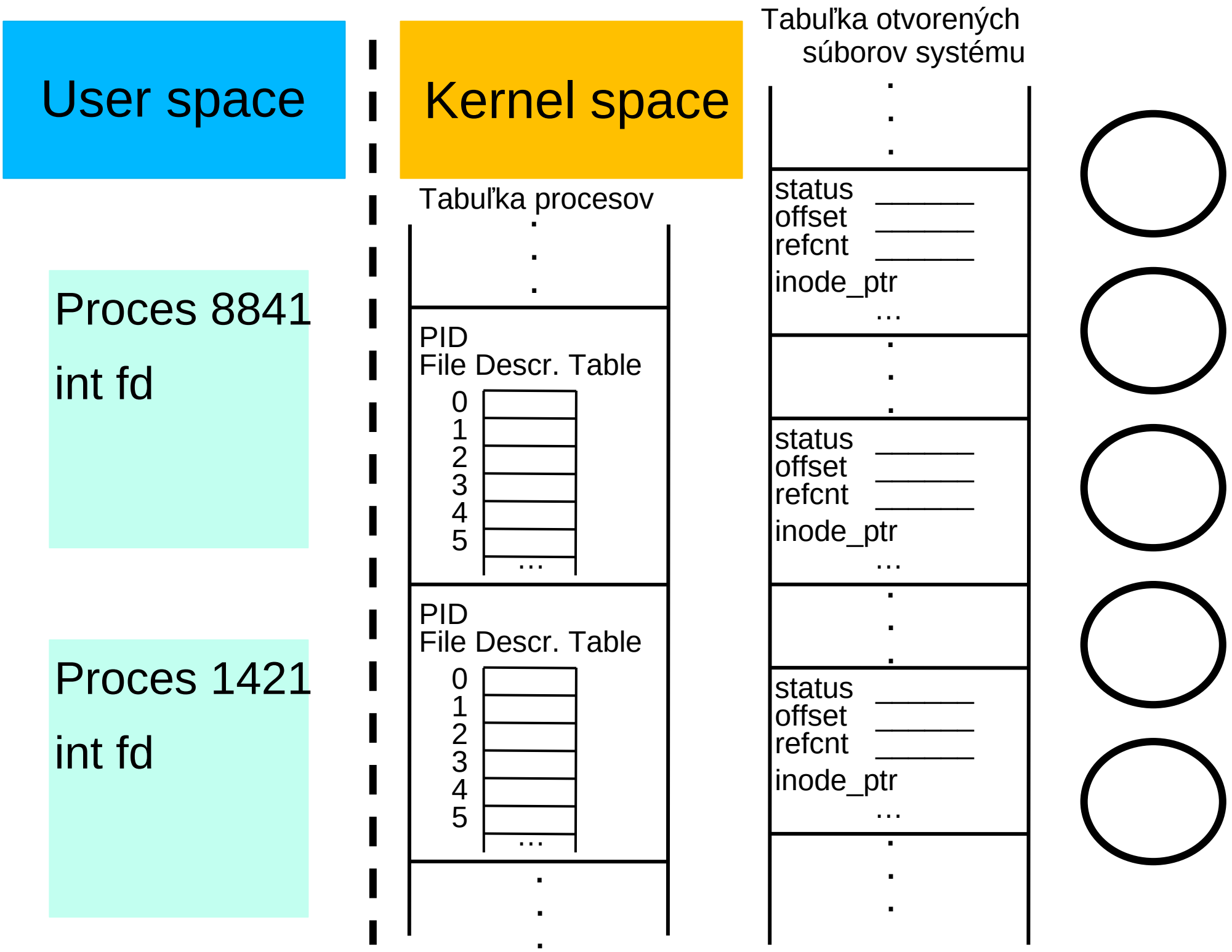
SV open

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

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


User space
Kernel space
Tabuľka otvorených súborov systému
Tabuľka procesov
Proces 8841
int fd
Proces 1421
int fd
PID
File Descr. Table
0
1
2
3
4
5
...
PID
File Descr. Table
0
1
2
3
4
5
...
status
offset
refcnt
inode_ptr
...
status
offset
refcnt
inode_ptr
...
status
offset
refcnt
inode_ptr
...

SV open

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

SV open

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

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

SV open

- Hw uchová registre procesora

SV open

- Hw uchová registre procesora
- Hw zvýši úroveň oprávnení procesora

SV open

- Hw uchová registre procesora
- Hw zvýši úroveň oprávnení procesora
- Hw zabezpečí vyvolanie vstupného bodu do jadra OS

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)

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

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

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 užívateľskom 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

SV fork

- Zrod nového procesu
- Čo je to `shell`?

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`)

SV fork

- Pre každý príkaz napísaný v programe `shell` sa vytvorí nový proces (napr. `echo ahoj`)
- Ako? Pomocou systémového volania „vidlička“

SV fork

- Pre každý príkaz napísaný v programe `shell` sa vytvorí nový proces (napr. `echo ahoj`)
- Ako? Pomocou systémového volania „vidlička“
- Prečo vidlička?

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…

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“?

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* !!!!!!!!!!!!!

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

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()`

SV fork

- Odlišuje ich návratová hodnota volania `fork()`

- `<0` znamená chybu
- `>0` znamená rodiča
- `=0` znamená potomka

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?

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)`

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

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)

Príklad redirect.c

- Presmerovanie výstupu programu do súboru na disku (miesto vypísania na monitore)

- Čo sa deje pri ‘`echo ahoj > output.txt`’?

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 !!!!!!

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')`

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

Komunikácia medzi procesmi

- Pomocou rúry: má 2 konce; jeden na čítanie, druhý na zápis

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

- Pomocou systémového volania `pipe()`

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

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

- Ak chcem použiť rúru na komunikáciu medzi procesmi (na čo iné by sme ju použili, že?), musíme správne skombinovať SV `fork()` a `pipe()`

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

- Obrázok prepojenia ‘`ls | grep x`’

(ukážka `pipe2.c`)

Čítanie obsahu priečinka

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`)

PREČÍTAŤ kapitolu **1** z knižky:

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

MIT ;)