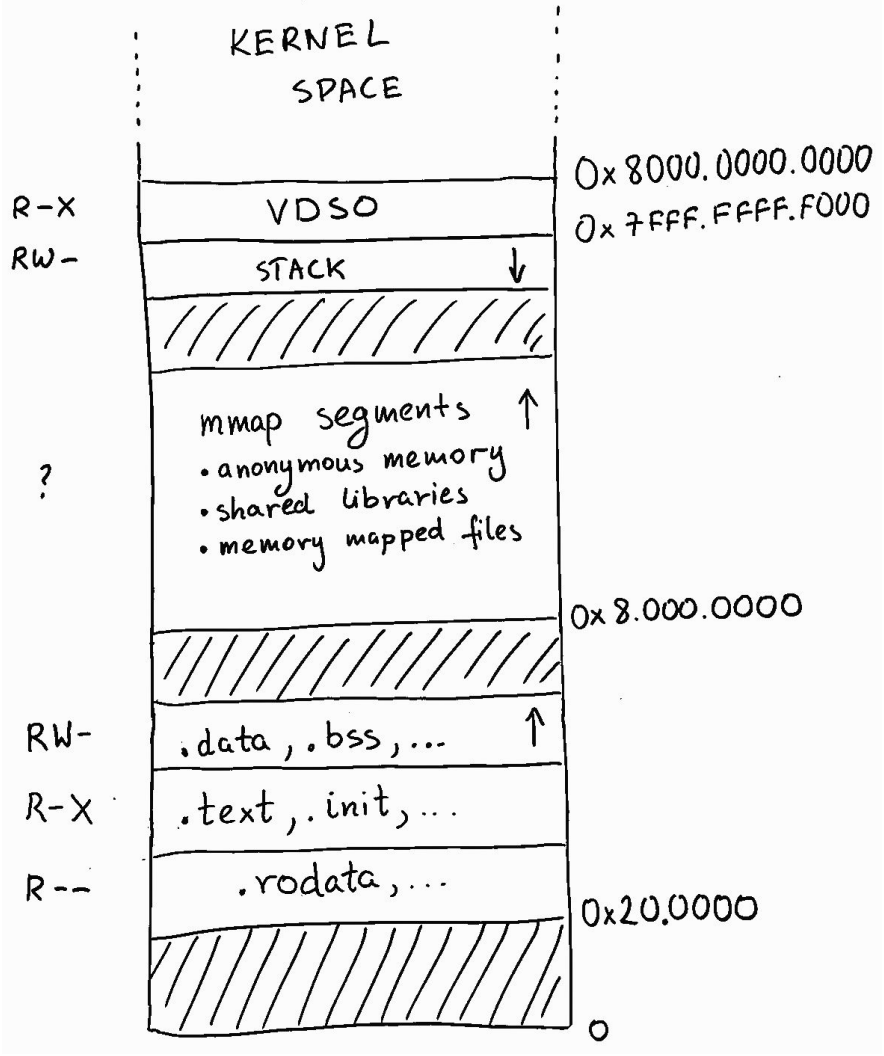
Struktura jądra UNIX

Wykład 10 i 11: Procesy

Przestrzeń adresowa procesu: **FreeBSD x86-64**

Jądro żyje w górnej połowie przestrzeni adresowej każdego procesu!

- Mapa pamięci procesu: **`procstat -v`**
- Pod koniec BSS koniec programu (ang. *segment break*).
- Sterta przydzielana **`mmap`**.
- Tyle stosów ile wątków!
- Biblioteka współdzielona jądra vdso.



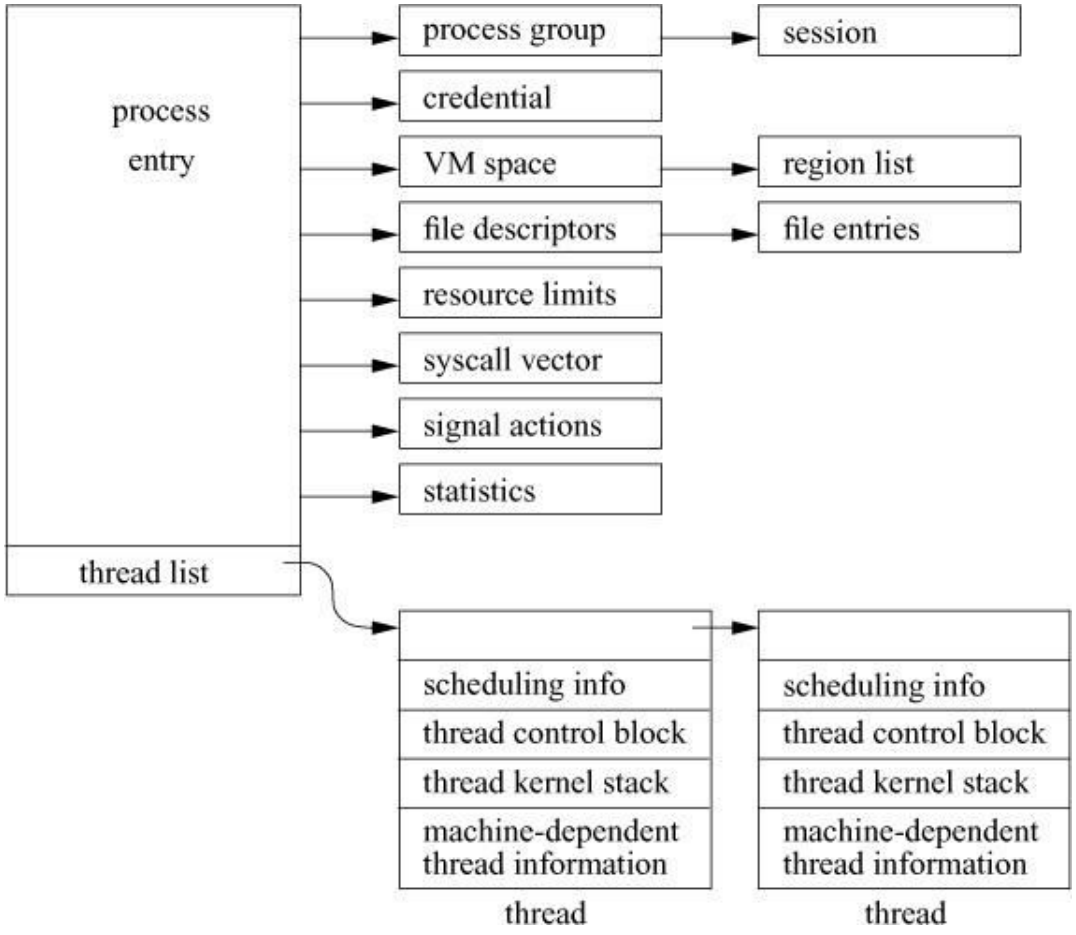
FreeBSD: Zasoby procesu

Proces `proc` posiada co najmniej jeden aktywny wątek `thread`.

Rozróżnienie między zasobami, a kontekstami wykonania.

`pcb` (process control block) to sprzętowy kontekst wątku (historyczna nazwa)

Model wątków 1:1.



FreeBSD: Stan procesu vs. stan wątku

Procesy i wątki to różne byty! Stany procesu:

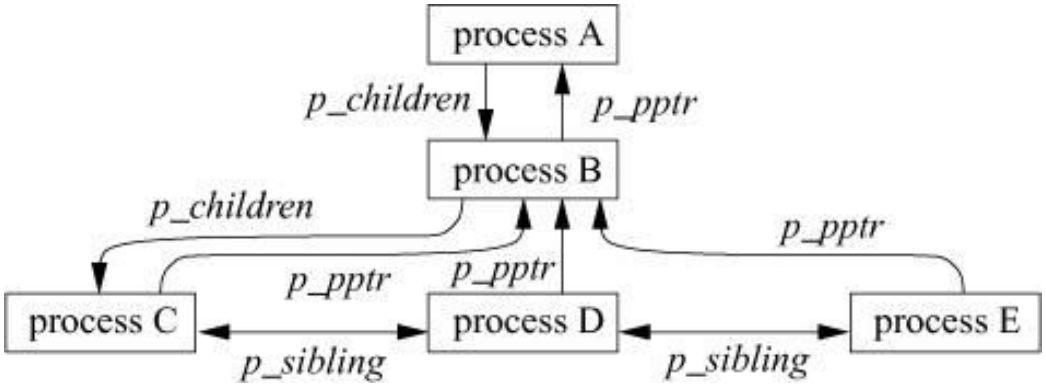
Stan	Opis
`PRS_NEW`	proces jest w trakcie tworzenia
`PRS_NORMAL`	istnieje co najmniej jeden wątek, który jest w stanie `RUNNABLE`, `SLEEPING` lub `STOPPED`
`PRS_ZOMBIE`	proces jest w trakcie odśmiecania

Dodatkowo mamy cały szereg flag w `p_flag` rejestrujących pewne fakty o procesie na potrzeby implementacji wywołań systemowych (np. `P_EXEC` dla `ptrace`).

Ciekawsze zasoby procesu `proc`

- Tożsamość `p_ucred` (identyfikatory)
- Tabela deskryptorów plików i bieżący katalog `p_fd`
- Statystyki `p_stats` (zużyty czas i zasoby)
- Limity na zasoby `p_limit` (pamięć, czas procesora, …)
- Maski sygnałów i bieżące akcje `p_sigacts`
- Opis przestrzeni adresowej `p_vmspace`
- Wektor wywołań systemowych `p_sysent`
- V-węzeł pliku wykonywalnego `p_textvp`
- Grupa procesów `p_pgrp`
- Czasomierze `p_realtimer` i `p_itimers`

FreeBSD: Hierarchia procesów



W nagłówku pliku `proc.h` mowa o konwencji zakładania blokad:
(c) `proc::p_mtx` (d) `allproc_lock` (e) `proctree_lock`
(f) `session::s_mtx` (g) `pgrp::pg_mtx`.

Każdy proces jest liście **`allproc`** albo **`zombproc`** chronionej blokadą **`allproc_lock`** (`sx`). Kiedy trawersujemy drzewo należy trzymać blokadę **`proctree_lock`**. Kiedy członków grupy to **`pg_mtx`**.

Komentarz: Dodatkowo każdy proces posiada **żniwiarza** (ang. *reaper*).

FreeBSD: Tworzenie procesów

1. **«fork»** kompletna (?) kopia procesu
2. **«vfork»** nie klonuje przestrzeni adresowej, rodzic jest wznawiany wtw. gdy dziecko zawoła `execve` lub `exit`
3. **«rfork»** tworzy proces potomny, który współdzieli pewne zasoby z procesem tworzącym (`RFMEM`, `RFSIGSHARE`)

Używając jądra `Linux` tworzymy nowe procesy przy pomocy clone(2) → brak wyraźnego podziału na procesy i wątki.

We `FreeBSD` osobne wywołania systemowe: `thr_new(2)`, `thr_exit(2)`, `thr_kill(2)` i uniwersalny mechanizm konstrukcji środków synchronizacji: `_umtx_op(2)`.

`fork1`: tworzenie nowego procesu

Po przydzieleniu pamięci na strukturę `proc` (**`newproc`**):

1. Tworzymy nowy wątek `thread_alloc` (**`td2`**).
2. Podłączamy wątek do procesu `proc_linkup`.
3. Klonujemy przestrzeń adresową `vmspace_fork` (**`vm2`**).
4. Współdzielimy kredencjały z rodzicem.
5. Inicjujemy księgowanie zasobów `racct_proc_fork`.

Tak uzbrojeni wchodzimy do `do_fork`, gdzie:

1. Kopiujemy pola `p_startcopy`…`p_endcopy` z rodzica,
   np. nazwę procesu, priorytet, ograniczenie na zużycie czasu.
2. Czyścimy pola `p_startzero`…`p_endzero`:
   np. zużycie zasobów, ustawienia czasomierzy.

do_fork: główna procedura klonowania procesu

1. Stan procesu ustala na `PRS_NEW`.
2. Przydziela nowy identyfikator `fork_findpid`.
3. Wstawienia proces na listę wszystkich procesów `allproc` oraz odpowiedniego kubełka tablicy mieszającej `pidhashtbl`.
4. Kopiuje tablicę deskryptorów `fdcopy`.
5. Kopiuje ustawienia planisty `sched_fork` (w
6. Klonuje ustawienia sygnałów `sigacts_copy`.
7. Kopiuje lub zeruje limity `lim_fork` i statystyki `pstats_fork`.
8. Wstawia proces do grupy procesów rodzica `p_pgrp` i podłącza proces potomny pod rodzica `p_pptr`.
9. Przygotowuje przestrzeń adresową i kontekst na pierwsze wejście nowego wątku vm_forkproc (kopiuje stos, `fork` ma zwrócić zero).
10. Oznacza proces `PRS_NORMAL` i dodaje wątek do planisty `sched_add`.

Szybkie wyszukiwanie identyfikatorów `pid`

Żeby przyspieszyć wyszukiwanie procesów względem numeru `pid` uciekamy się do dobrze znanego pomysłu → kubełki procesów: `pidhashtbl[]` i `pidhashtbl_lock[]`.

Problem: Jak szybko znaleźć nieużywany numer `pid`?

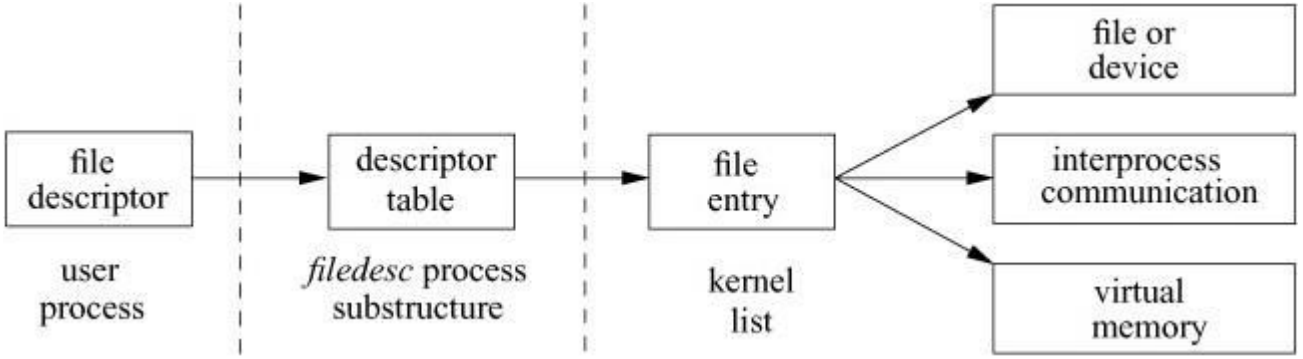
Uwaga! We FreeBSD §4.5 nieaktualny opis!

Trzymamy bitmapę `proc_id_pidmap` zajętych numerów `pid`. Podobnie dla `grpid` i `sessid`. Procedura `fork_findpid` przeszukuje ją używając generatora liczb pseudolosowych.

Do randomized PIDs bring more security?

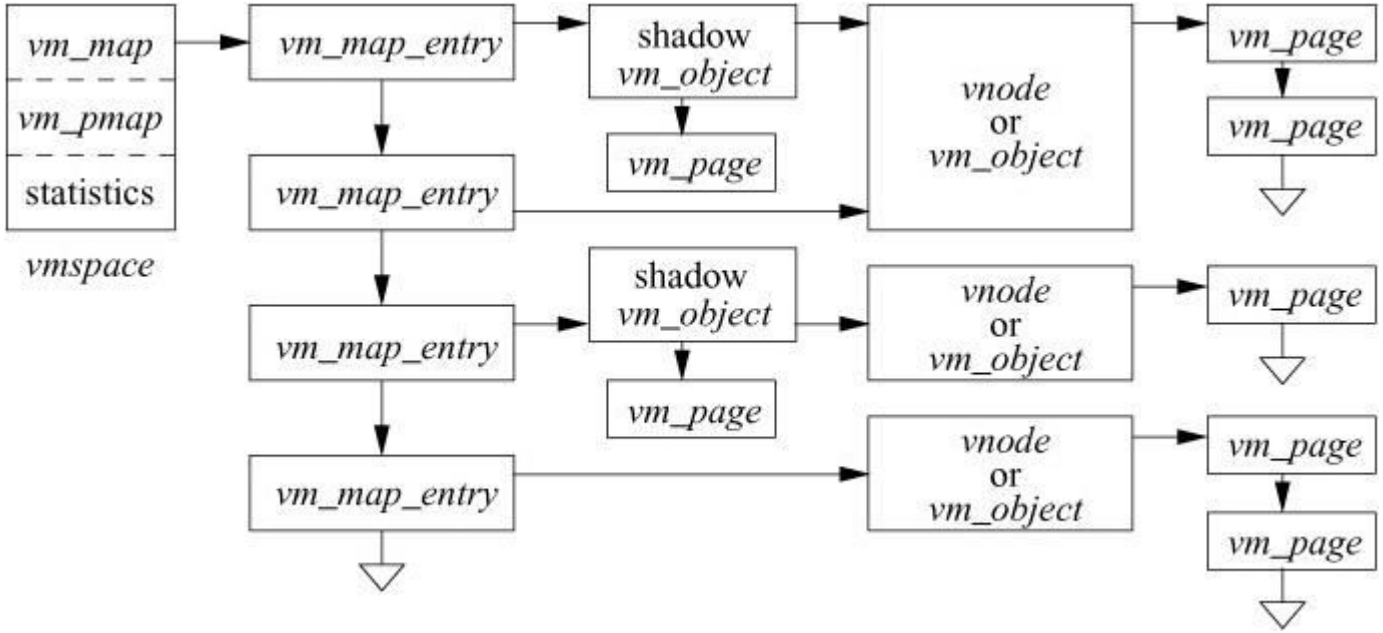
Kopiowanie tablicy deskryptorów `filedesc(9)`

Kopiowanie tablicy deskryptorów (`fdcopy`) inkrementuje licznik referencji (`fhold`) wszystkim plikom (`file`), na które wskazują deskryptory plików.



Innymi słowy kopiujemy referencje do otwartych plików, a nie struktury opisujące otwarte pliki. Zatem współdzielimy kursory plików (`f_offset`) i flagi (`f_flags`).

Klonowanie przestrzeni adresowej



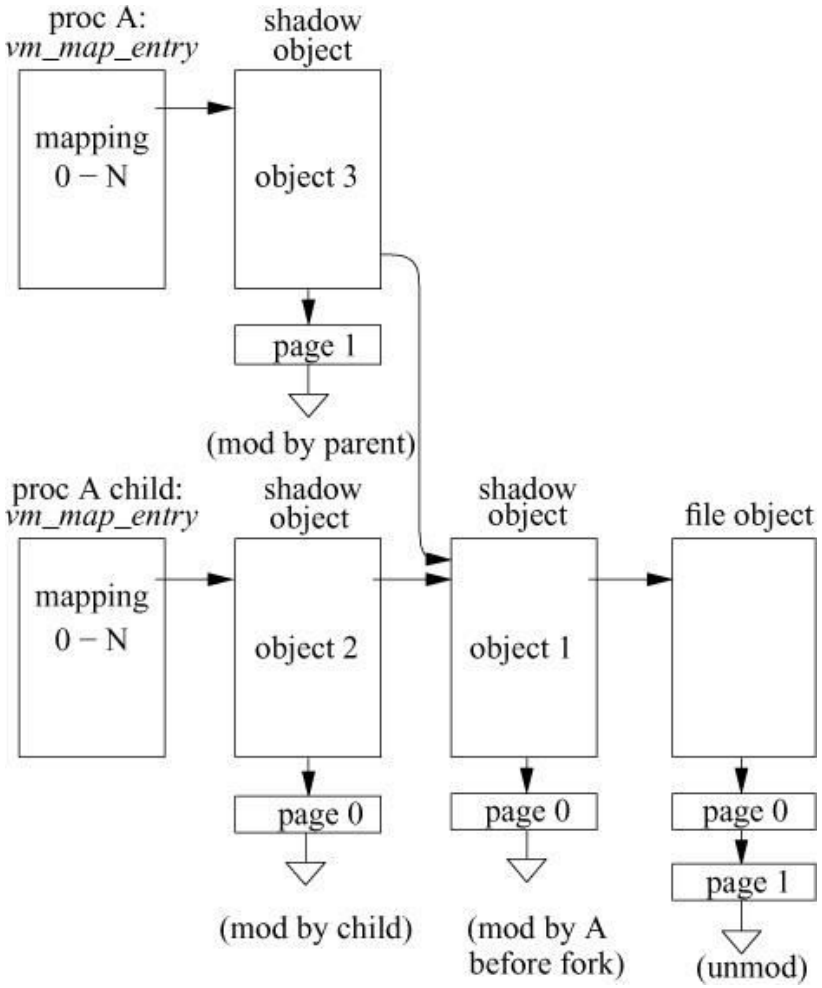
Sklonujemy tylko i wyłącznie segmenty (`vm_map_entry`) bez kopiowania ich zawartości (`vm_object`).

Zawartość będzie trzeba oznaczyć tak, żeby zapis wygenerował kopię modyfikowanej strony (ang. copy on write).

Obiekty przesłaniające i kopiowanie przy zapisie

Każdy obiekt zawiera strony, które zmodyfikował względem kolejnego w łańcuchu obiektu wspierającego.

Po `fork` tworzymy obiekty przesłaniające w obu procesach. W tablicy stron `pmap` oznaczamy wszystkie strony `read-only`. Obsługa błędy strony skopiuje stronę i zmieni uprawnienia.



FreeBSD: Kończenie pracy procesów

Faktycznie tylko dwa sposoby na zakończenie procesu:

- normalne zakończenie pracy (`exit`)
- otrzymanie sygnału uśmiercającego (awaria lub `kill`)

Można oczekiwać na zakończenie jednego z bezpośrednich potomków `wait4(2)`. Odbierzemy informację czy zakończył się normalnie, z błędem, czy w wyniku awarii. Można też odczytać zużycie zasobów `rusage` (czas CPU, pamięć, operacje I/O). W przypadku `wait6(4)` możemy też dostać `siginfo_t` (np. błędny adres wiodący do `SIGSEGV`).

`exit1`: Śmierć procesu (1)

1. Najpierw należy zabić pozostałe wątki w procesie:
   a. wątek wchodzący z przestrzeni użytkownika do przestrzeni jądra zostanie przekierowany do `thread_exit`
   b. wątki w jądrze zamiast pójść spać wrócą z `EINTR` lub `EAGAIN`
   c. **pytanie:** co z tymi które śpią? `thread_suspend_check`
2. Potem zwolnić zasoby procesu:
   a. anulowanie terminów czasomierzy
   b. zwolnienie przestrzeni adresowej `vmspace_exit`
   c. zamknięcie wszystkich deskryptorów plików `fdescfree`
3. I wreszcie zatroszczyć się o zależne procesy:
   a. wycofanie się z roli lidera sesji `killjobc`
   b. zabicie zatrzymanych lub śledzonych procesów potomnych
   c. przepięcie procesów potomnych pod żniwiarza `proc_reparent`

exit1: Śmierć procesu (2)

Zanim `exit1` zakończy działanie musi:

1. Zapisać kod wyjścia procesu w `p_xexit`.
2. Dodać statystyki procesu z `p_ru` do statystyk rodzica `ruadd`.
3. Wypiąć się z listy procesów (zostaje w `pidhashtbl`).
4. Wybudzić rodzica czekającego na potomków `wakeup`.
5. Wysłać sygnał `SIGCHLD` do rodzica `childproc_exited`.
6. Przejść w stan `PRS_ZOMBIE`.

Ostatni wątek należący do procesu popełnia `thread_exit`, a w rezultacie `sched_throw`, które wybiera kolejny proces do uruchomienia `choosethread` i porzuca kontekst bieżącego wątku `cpu_throw` na rzecz nowo wybranego.

Stan ZOMBIE czyli życie po śmierci

Proces w stanie ZOMBIE jest nadal w kubełku pidhashtbl i na liście potomków procesu p_children.

Wywołanie systemowe wait* ostatecznie trafia do kern_wait6. Jeśli na liście p_children procedura proc_to_reap znajdzie zombiaka to przystępujemy do pogrzebu proc_reap:

- Informacje p_ru, p_xexit, p_xsigno i pochodne są przeznaczone do skopiowania do user-space.
- Usuwamy proces z listy potomków i pidhashtbl.
- Proces opuszcza grupę procesów leavepgrp.
- Zwalniamy identyfikator procesu proc_id_clear.
- Zwalniamy pozostałe elementy struktury proc i ją samą.

execve(2): Ładowanie programów

Używamy pary `fork` i `execve`, alternatywnie `posix_spawn`.

1. Ustalamy zasoby widoczne dla uruchamianego programu:
   a. deskryptory nieoznaczone FD_CLOEXEC → przechodzą
   b. sygnały: ignorowane → przechodzą, wyłapywane → ustawiane `SIG_DFL`
   c. katalog główny i roboczy → przechodzą
   d. przestrzeń adresowa → całkowicie wymazywana
2. Wołamy `execve(path,argv,envp)`:
   a. `path` → ścieżka absolutna
   b. `argv` → lista parametrów
   c. `envp` → lista zmiennych środowiskowych w formacie "`key=value`".

Pliku programu nie można modyfikować → `ETXTBSY` !

Niektórzy (z M$) twierdzą, że `fork` to zły pomysł: A `fork()` in the road, HotOS 2019

`sys_execve`: obsługa wywołania systemowego

Obraz procesu to skonfigurowana przestrzeń adresowa.

- Zatrzymujemy wszystkie wątki użytkownika w przestrzeni jądra `thread_single` (podobnie jak przy fork).
- Przygotowujemy argumenty przekazywane do obrazu procesu `image_args` (w stronicowalnej pamięci jądra) za pomocą `exec_copyin_args`.
- Wołamy właściwą procedurę `do_execve`, która ładuje nowy obraz procesu i modyfikuje odpowiednio zasoby.
- Jeśli udało się utworzyć nowy obraz procesu, to post_execve każe starym wątkom popełnić wyjście przy powrocie do przestrzeni użytkownika i kasuje stary obraz procesu.

do_execve: właściwa część ładowania obrazu

1. Wykrywa format pliku wykonywalnego (image activators):
   a. plik interpretowany (**`#!`** aka shebang, albo ELF(5) z segmentem `INTERP`) → ładuje interpreter (**`/bin/sh`** i jemu przekazuje program i parametry),
   b. plik wykonywalny (`ELF` o typie `DYN` lub `EXEC`)
2. Tworzy nową przestrzeń adresową:
   a. tworzy stos i wrzuca na niego `argv` i `envp`, oraz `auxiliary vector`
3. Ładuje program:
   a. odczytuje listę segmentów typu `PT_LOAD`
   b. odwzorowuje w pamięć strony pliku wykonywalnego według informacji ze struktury `elf64_phdr`, tj. `p_vaddr`, `p_filesz`, `p_memsz`, `p_align`
   c. jeśli ELF dla Linuksa to zmienia `sysentvec`
4. Zwalnia zasoby nieużywane przez proces.
5. Tworzy kontekst startowy na podstawie `e_entry`.

ELF: Nagłówek pliku wykonywalnego

```
# file /bin/sh
/bin/sh: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
dynamically linked, interpreter /libexec/ld-elf.so.1,
for FreeBSD 12.0 (1200086), FreeBSD-style, stripped

# readelf -h /bin/sh
ELF Header:
  ...
  OS/ABI:                            FreeBSD
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices x86-64
  Version:                           0x1
  Entry point address:               0x20b000
  ...
```

ELF: Nagłówki segmentów pliku wykonywalnego

```
# readelf -l /bin/sh
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flg    Align
  ...
  INTERP         0x0000000000000270 0x0000000000200270 0x0000000000200270
                 0x0000000000000015 0x0000000000000015  R      0x1
      [Requesting program interpreter: /libexec/ld-elf.so.1]
  LOAD           0x0000000000000000 0x0000000000200000 0x0000000000200000
                 0x000000000000a2a0 0x000000000000a2a0  R      0x1000
  LOAD           0x000000000000b000 0x000000000020b000 0x000000000020b000
                 0x000000000001bc70 0x000000000001bc70  R E    0x1000
  LOAD           0x0000000000027000 0x0000000000227000 0x0000000000227000
                 0x0000000000001190 0x00000000000033ac  RW     0x1000
```

ELF: Konsolidator dynamiczny

```
# file /libexec/ld-elf.so.1
/libexec/ld-elf.so.1: ELF 64-bit LSB shared object, x86-64,
version 1 (FreeBSD), dynamically linked, stripped

# /libexec/ld-elf.so.1 -h
Usage: /libexec/ld-elf.so.1 [-h] [-f <FD>] [--] <binary> [<args>]
...

# procstat -v $$ | cut -f 2,3,4,11 -w
0x200000   0x20b000   r-- /bin/sh
0x20b000   0x227000   r-x /bin/sh
0x227000   0x228000   rw- /bin/sh
...
0x800227000    0x80022f000    r-- /libexec/ld-elf.so.1
0x80022f000    0x800248000    r-x /libexec/ld-elf.so.1
0x800248000    0x800249000    rw- /libexec/ld-elf.so.1
```

Sygnały, grupy, sesje

FreeBSD: Sygnały

Sygnały są programowym odzwierciedleniem przerwań:

Hardware Machine	Software Virtual Machine
instruction set	set of system calls
restartable instructions	restartable system calls
interrupts/traps	signals
interrupt/trap handlers	signal handlers
blocking interrupts	masking signals
interrupt stack	signal stack

Wysłanie sygnału może się zakończyć:

1. Zignorowaniem go.
2. Zakończeniem wszystkich wątków w procesie.
3. Wstrzymaniem / wznowieniem wszystkich wątków.

Procedura obsługi sygnału

Ustawiana przez wywołanie `sigaction(2)` i przechowywana w `p_sigacts`. Wywoływana przez jądro w wyniku zdarzenia:

- programowego → np. naciśnięcie `CTRL+C` na klawiaturze
- sprzętowego → np. użycie nieznanej instrukcji

```
struct sigacts {
  sig_t    ps_sigact[_SIG_MAXSIG];    // Disposition of signals
  sigset_t ps_catchmask[_SIG_MAXSIG]; // Signals to be blocked
  ...
};
```

Za tłumaczenie zdarzeń sprzętowych na sygnały odpowiada procedura `trap` (`MIPS`).

NetBSD: Właściwości i domyślne akcje sygnałów

Tablica sigprop dla każdego sygnału przechowuje logiczną sumę:

```
SA_KILL      : terminates process by default
SA_CORE      : ditto and coredumps
SA_STOP      : suspend process
SA_TTYSTOP   : ditto, from tty
SA_IGNORE    : ignore by default
SA_CONT      : continue if suspended
SA_CANTMASK  : non-maskable, catchable
SA_NORESET   : not reset when caught
SA_TOLWP     : to LWP that generated, if local
SA_TOALL     : always to all LWPs
```

Pytanie: Jakie właściwości mają: SIGCHLD, SIGINT, SIGTRAP, SIGTSTP, SIGKILL, SIGSEGV, SIGSTOP, SIGCONT?

FreeBSD: Wysyłanie sygnału psignal(9)

1. **Sygnały synchroniczne.**
   trap rejestruje błąd w ksiginfo i dostarcza sygnał trapsignal. Proces nieśledzony → sygnał dostarczony do wątku tdsendsignal bez opóźnienia, w p.p. skierowany do procesu śledzącego (gdb).
2. **Sygnały asynchroniczne.**
   Sprawdzenie uprawnień p_cansignal (SIGCONT!).
   Do procesu kern_psignal, a grupy pgsignal.
   Sprawdza czy sygnał ignorowany p_sigignore, czy wyłapywany p_sigcatch. Dodaje do listy sygnałów oczekujących td_siglist i wybudza wątek, jeśli to możliwe.

Dostarczanie sygnału

Sygnały sprawdzane w `ast` przed powrotem do przestrzeni użytkownika i przed uśpieniem `sleepq_wait_sig`.

Przy wyznaczaniu sygnału do dostarczenia `issignal`, wątek sumuje oczekujące sygnały procesu `p_sigqueue` i `td_sigqueue`. Kiedy znamy już numer sygnału odznaczamy go w obu kolejkach.

W pętli odpytujemy `cursig` i dostarczamy je `postsig`, które:

1. albo zabija proces → może utworzyć zrzut pamięci,
2. albo przygotowuje kontekst wątku, z użyciem `sendsig` (MD), na wejście do procedury obsługi sygnału; dodatkowo maskuje sygnał na czas obsługi.

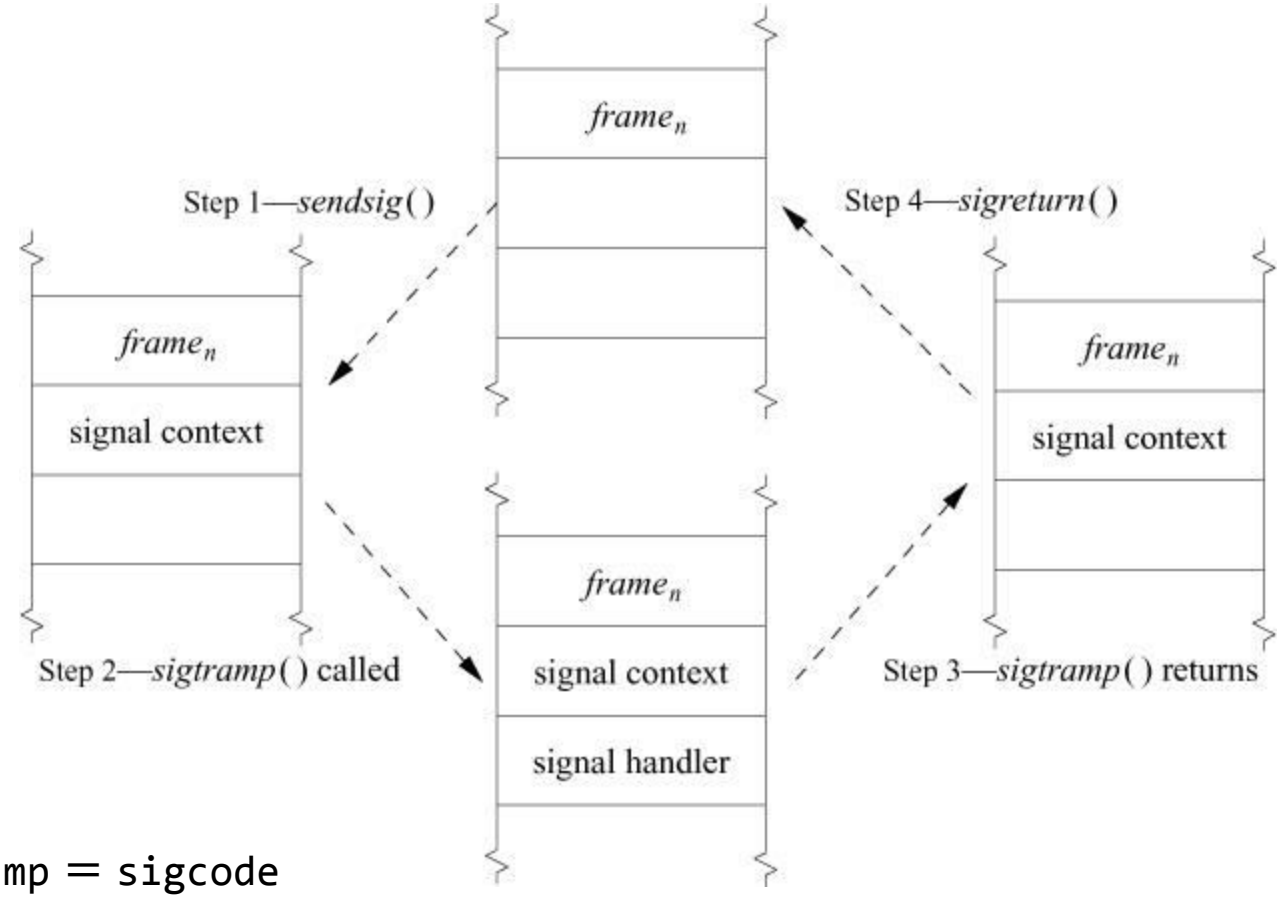
Obsługa sygnału

Procedura **`sendsig`** kopiuje na stos procedury obsługi sygnału (pożyczony od wątku użytkownika albo ustawiony `sigaltstack`):

1. Bieżący kontekst użytkownika `sigframe`.
2. Argumenty procedury obsługi sygnału.
3. Kod*, którego zadaniem jest wywołanie procedury obsługi sygnału i wywołanie `sigreturn` po zakończeniu.

Zadaniem **`sigreturn`** jest atomowo odblokować sygnały i przywrócić kontekst ze stosu użytkownika.

Obsługa sygnałów



`sigtramp` = `sigcode`

Dynamiczna biblioteka jądra

Jak w trybie użytkownika zawołać `sigcode`, skoro jednak stos nie jest wykonywalny? Na **`amd64`** ustawiamy **`%rip`** na `sigcode` umieszczony na współdzielonej stronie.

Na sam koniec przestrzeni adresowej `exec_new_vmspace` wrzuca odpowiednik linuksowego `vdso(8)`, czyli współdzieloną stronę z kodem i danymi przygotowaną przez `exec_sysvec_init`.

Oprócz `sigcode` trzymamy na niej procedury do odczytu czasu (`__vdso_gettimeofday`) dla biblioteki `libc`.

Czas aktualizowany w dogodnych dla jądra momentach, ale nie rzadziej niż co takt zegara systemowego.

Grupy procesów

Zarządzanie procesami zgrupowanymi w zadania (ang. *jobs*):

1. potoki w `shell`'u
2. wysyłanie sygnałów `SIGSTOP`, `SIGCONT`, `SIGINT`, ...

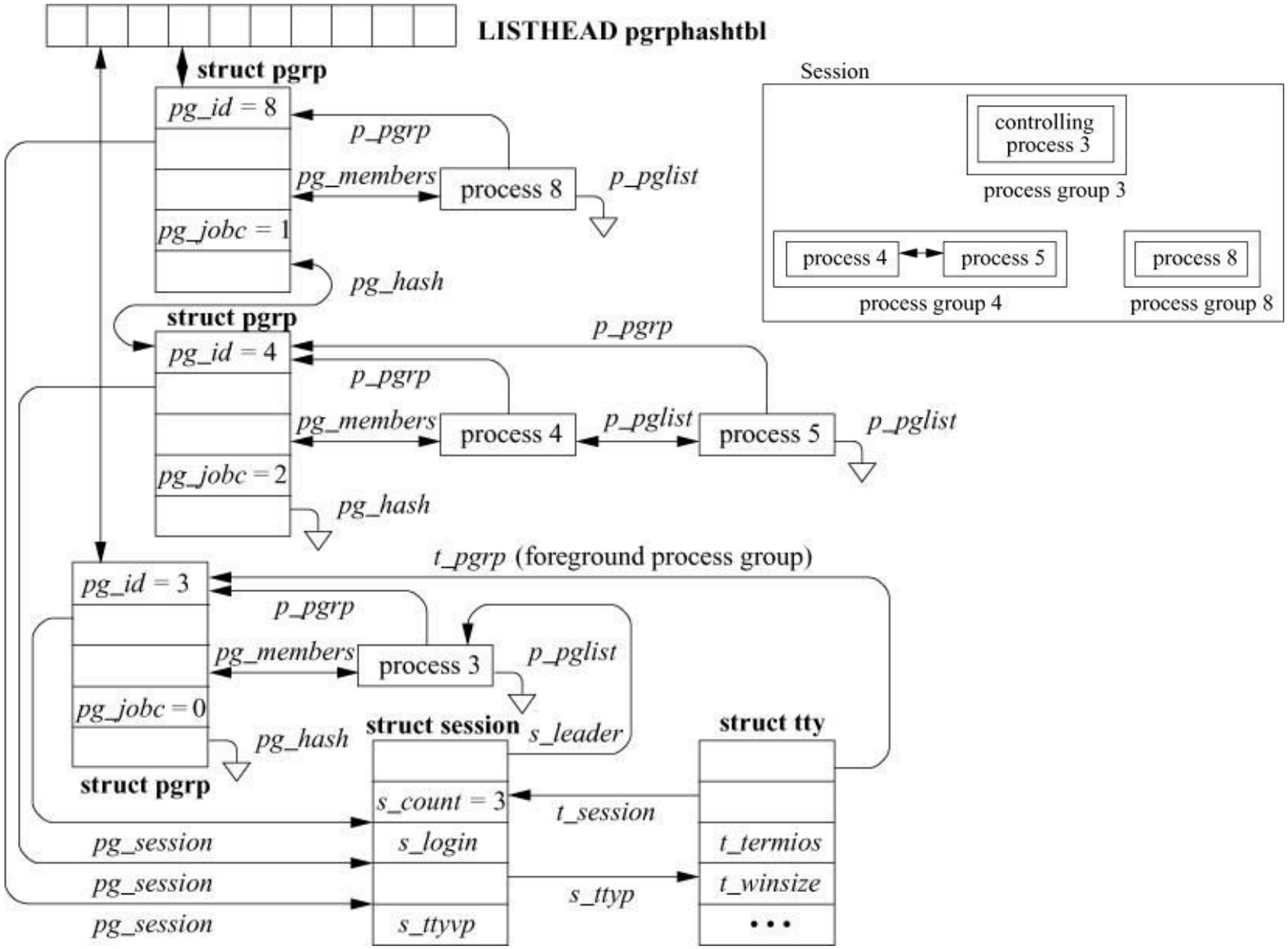
Każdy proces należy do dokładnie jednej grupy procesów `pgrp` zapisanej w `p_pgrp`! Identyfikator grupy **`pgid`** jest tożsamy z identyfikatorem **`pid`** procesu, który ją utworzył. Domyślnie grupa dziedziczona po rodzicu.

`setpgid`: proces może zmienić swoją grupę tworząc nową, albo zmienić grupę swoich procesów potomnych*.

Sesje

Tworzona wraz z nową grupą procesów przy pomocy `setsid(2)`.

1. Sesja może posiadać urządzenie **terminala sterującego**.
2. Lider sesji może ustanowić połączenie z terminalem
   → staje się wtedy **procesem sterującym**.
3. Grupy procesów dzielą się na **pierwszoplanowe**
   (ang. *foreground*) i **drugoplanowe** (ang. *background*).
4. Kody sterujące interpretowane przez urządzenie terminala
   (np. `Ctrl+c`, `CTRL+\`) są zamieniane na sygnały (`SIGINT`,
   `SIGQUIT`) i dostarczane do grupy pierwszoplanowej.
5. Tylko członkowie sesji są w stanie zmieniać przypisanie
   terminala sterujące do innej grupy procesów w obrębie sesji.


LISTHEAD pgrphashtbl
struct pgrp
pg_id = 8
p_pgrp
pg_members
process 8
p_pglist
pg_jobc = 1
pg_hash
struct pgrp
pg_id = 4
p_pgrp
p_pgrp
pg_members
process 4
p_pglist
process 5
p_pglist
pg_jobc = 2
pg_hash
t_pgrp (foreground process group)
pg_id = 3
p_pgrp
pg_members
process 3
p_pglist
pg_jobc = 0
pg_hash
struct session
s_leader
struct tty
struct pgrp
s_count = 3
t_session
pg_session
s_login
t_termios
pg_session
s_ttyp
t_winsize
pg_session
s_ttyvp
• • •
Session
controlling process 3
process group 3
process 4
process 5
process group 4
process 8
process group 8

Problem z osieroconymi grupami procesów

Grupa procesów jest **osierocona**, jeśli żaden proces należący do tej grupy nie posiada rodzica, który należy do innej grupy w obrębie tej samej sesji.

Jeśli proces sterujący (np. powłoka) się zakończy, to system odbiera procesom w sesji dostęp do urządzenia terminala, a do grupy pierwszoplanowej wysyła `SIGHUP`. W grupach drugoplanowych nadal możemy zatrzymane procesy.

- Grupa pierwszoplanowa przechowywana w `tty::t_pgrp`.
- Liczba potomków procesu sterującego w `pgrp::pg_jobc`.

Osieroconym grupom procesów jądro wysyła `SIGHUP` i `SIGCONT`. Zapobiega ponownemu wysyłaniu `SIGTTIN` i `SIGTTOU`.

Śledzenie procesów

Śledzenie wykonania procesów

```
int ptrace(int request, pid_t pid, caddr_t addr, int data);
```

Z użyciem wywołania `ptrace` można:

- podłączyć się do istniejącego procesu by go odpluskwiać
- czytać i modyfikować stan procesu (rejestry, pamięć)
- przechwytywać sygnały wysyłane do procesu
- nasłuchiwać na zdarzenia (**`fork`**, **`exec`**)
- śledzić wywołania systemowe
- przełączyć wykonywanie programu w tryb krokowy

Śledzenie wywołań systemowych z użyciem narzędzia `strace` (Linux) lub `truss` (FreeBSD)!

Debugowanie procesu

Śledzenie (ang. *tracing*) i instrukcje breakpoint → `SIGTRAP`.

1. Debugowany proces zamiast odbierać sygnał,
   zostaje zatrzymany (stan `STOPPED`).
2. Debugger zostaje poinformowany `SIGCHLD`.
3. Debugger sprawdza stan procesu wywołaniem `wait6(2)`.
4. Podjęcie decyzji o debugowanym procesie:
   a. **`PT_CONTINUE:`** wznawia pracę pod zadanym adresem
      i ew. dostarcza sygnał o podanym numerze
   b. **`PT_STEP:`** j.w. ale w trybie krokowym
   c. **`PT_KILL:`** `PT_CONTINUE` + `SIGKILL`

Startowanie sesji debugowej

Można się podłączyć do istniejącego procesu: **`PT_ATTACH`**.
Co jeśli chcemy zatrzymać się na pierwszej instrukcji?

1. Debugger tworzy dziecko.
2. Te oznacza się flagą `P_TRACED` przy użyciu `PT_TRACE_ME`.
3. Woła `execve(2)` ze ścieżką do pliku wykonywalnego.
4. Jądro po załadowaniu obrazu wysyła `SIGTRAP`.

Możemy odłączyć się od debugowanego procesu **`PT_DETACH`**.

Dostęp do danych debugowanego procesu

1. Odczyt i zapis rejestrów: **`PT_GETREGS`**, **`PT_SETREGS`**
2. Pojedyńcze słowa maszynowe: **`PT_READ_*`**, **`PT_WRITE_*`**
3. Kopiowanie pamięci: **`PT_IO`**
4. Odczyt stanu procesu: **`PT_LWPINFO`**, ...

W przypadku kopiowania dużych ilości pamięci z / do debugowanego procesu `PT_IO` niewydajne!

Z pomocą przychodzi system plików `/proc`!

Oprócz tych samych funkcji co **`ptrace`**, umożliwa odwzorowanie przestrzeni adresowej debugowanego procesu (**`/proc/pid/map`**) przy użyciu `mmap(2)`.

Pytania?