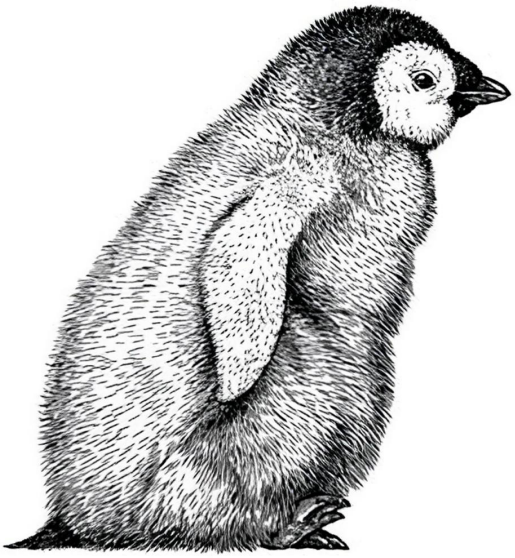


Systemy Operacyjne

Wybrane Zagadnienia

---

p *„Możecie zrobić Fork Bombę i wysłać na satori, nawet powinniście”*

Popełnione przez

Bajtocjanin
Zosia

Kraków
Anno Domini 2024

Spis treści

Licencja

Rozdział 1

POSIX

1.1 Główne zadania systemu operacyjnego

- obsługa plików
- poziom abstrakcji nad handlerem, tak aby nie dawać użytkownikowi bezpośredniego dostępu do hardware'u

1.2 Nalot, cechy POSIXa

- Jest tylko specyfikacją, a nie konkretną implementacją
- Oparty na C
- Nie ma superużytkownika ani administracji systemu
- Łatwy w implementacji (relatywnie)
- Historycznie jedynie niewielkie znaczące zmiany

1.3 Procesy

Proces - program w trakcie wykonywania

1.4 Wywołania systemowe

Wywołania systemowe stanowią interfejs między wykonywanym programem a (posiadającym zwykle wyższe uprawnienia) jądrem systemu operacyjnego.
Przykłady wywołań systemowych (syscalli)

- exit - kończy proces i opróżnia wszystkie bufory oraz sprząta to, czego wymaga biblioteka standardowa, _exit po prostu zakańcza proces
- fork - tworzy nowy proces równoległy z aktualnym, wykonujący ten sam kod, ma on jednak unikalny pid, inny ppid, i własną kopię deskryptorów rodzica oraz zablokowane te same sygnały
- exec* rodzinka syscalli - 1.4.1
- wait - blokuje proces, póki wszystkie dzieci żyją albo otrzymano sygnał
- kill 1.7.3
- waitpid - blokuje proces, dopóki proces o podanym pidzie się nie skończy. Dozwolone jest czekanie tylko na swoje dzieci.

1.4.1 Syscall execve

Podmienia wykonywany przez proces program, np. po zforkowaniu, aby dziecko wykonywało inny kod. Podajemy mu ścieżkę do binarki lub, jeśli coś jest dodane do kontekstu, nazwę.
Efekty uboczne:

- **domyślnie deskryptory plików pozostają otwarte!**
- przekierowania sygnałów do handlerów są zresetowane
- mapowania pamięci nie są zachowywane, są odmapowywane
- deskryptory do do POSIXowych wiadomości systemowych są zamykane.
- zamykane są wszystkie POSIXowe semafory
- wszystkie pozostałe wątki procesu są zamykane

1.5 Deskryptory plików

Deskryptory plików to unikalne dla procesów dodatnie inty używane do identyfikacji otwartych plików. Przyjmują wartości od 0 do OPEN_MAX, w POSIXie standardowo jest to max 16 otwartch plików. Co ciekawe, plik może być zreferowany przez dwa różne deskryptory, ale jeden deskryptor zawsze wskazuje na tylko jeden plik, czyli pomiędzy deskryptorami a open file description jest relacja many-to-one. Jedyną znaną nam flagą deskryptora pliku jest FD_CLOEXEC.

Domyślnie otwarte są deskryptory: 0 - stdin, 1 - stdout, 2 - stderr.

1.5.1 Otwieranie nowych deskryptorów

Nowe deskryptory otwieramy przy pomocy syscala **open**, który przyjmuje ścieżkę do pliku oraz flagi:

- O_RDONLY, otwarcie tylko do czytania
- O_WRONLY, otwarcie tylko do pisania
- O_RDWR, otwarcie do czytania i pisania
- O_NONBLOCK, wyłącza domyślne blokowanie
- O_APPEND, każdy write pisze na końcu pliku
- O_CREAT, stwórz plik, jeśli nie istnieje
- O_EXCL, wyrzuca błąd kiedy plik już istniał

Aby połączyć dwie flagi używamy logicznego lub $\rightarrow |, np.$ $(O\_CREAT \mid$ _EXCL). $Funkcja open tworzy nowy$ **open file description**, $w ktrej przechowywane s flagi.$

1.5.2 Czytanie z pliku

Czytamy przy pomocy syscalla **read**, który przyjmuje deskryptor, wskaźnik gdzie zapisać dane i liczbę byte'ów do przeczytania. Zwraca liczbę byte'ów, które udało mu się przeczytać lub 0, jeśli przytrafi się EOF.

1.5.3 Pisanie do pliku

Piszemy do pliku przy pomocy syscalla **write**, który przyjmuje deskryptor, wskaźnik na dane do zapisania i liczbę byte'ów do zapisania. Zwraca liczbę byte'ów, które udało mu się zapisać.

Domyślnie read i write są blokujące. Jeśli zostaną przerwanę przez jakiś sygnał zanim przeczytają / zapiszą cokolwiek to zwrócą -1 i ustawią errno na EINTR. Jeśli przeczytają trochę danych, ale niekoniecznie tyle, o ile prosiliśmy, to zwrócą rozmiar opracowanych danych i się przerwą.

1.5.4 Seek

Możemy zmienić aktualną pozycje w pliku bez czytania za pomocą **seek**, który przyjmuje deskryptor, *d* liczbę byteów o którą mamy się przesunąć i flagę, która ustala, odkąd liczymy przesunięcie: **SEEK_SET** przesuwa na *d* pozycji względem początku pliku, **SEEK_CUR** względem aktualnej pozycji, a **SEEK_END** względem końca pliku.
Przykład

```
int fd = open("foo", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
lseek(fd, 10000000000L , SEEK_CUR ); // ~10GB
write(fd, "a", 1);
close(fd);
```

Mimo że plik będzie miał rozmiar 1000000001, to będzie wyświetlane tylko "a".

1.6 Pipe'y

Pipe'y to kanały komunikacji, które pozwalają przesyłać informacje pomiędzy procesami.

```
int pipe(int fildes[2])
```

Pipe'a tworzymy, podając syscallowi **pipe** dwuelementową tablicę intów. Syscall stworzy pipe i ustawi fildes[0] na koniec do czytania (read descriptor) i fildes[1] na

koniec do pisania (write descriptor). Dane zapisane do write końca są buforowane przez kernel, póki nie zostaną przeczytane z read końca. Pipe ma maksymalny rozmiar, jeśli jest dużo nieprzeczytanych danych, to write się zablokuje.

1.6.1 FIFO - named pipe'y

Nazwane pipe'y działają trochę jak pipe'y, trochę jak pliki. Tworzymy je syscallami mknod lub mkfifo, przy czym mknod jest zarezerwowany tylko dla superużytkownika. Tworząc je, musimy podać ścieżkę, pod którą będą się znajdować, tak aby procesy bez wcześniejszego porozumiewania się mogły je znaleźć, zwykle najlepiej je tworzyć w folderze */tmp/*. Potem otwieramy je jak pliki i czytamy z nich lub piszemy do nich jak do zwyczajnych pipe'ów.

```
int mknod(const char *path, mode_t mode, dev_t dev)
int mkfifo(const char *path, mode_t mode)
```

Otwieranie nazwanego pipe'a do pisania (O_WRONLY), w przeciwieństwie do otwierania go do czytania (O_RDONLY), które jest blokujące, nie powiedzie się, jeśli po drugiej stronie nie ma czytelnika (przynajmniej na linuxie) i zwróci -1.

1.6.2 Uwagi co do równoległego czytania

Równoległe odczyty z pipe'ów, fifo i terminalu to działania niezdefiniowane. Może zadziała, a może nie.
Operacje I/O powinny być atomowe, czyli wszystkie bajty z danej operacji powinny być obok siebie, nie przeplecione bajtami z innych operacji. Dla Pipe'ów, FIFO i plików to zwykle działa, ale terminale są prawie zawsze wyłączone od tej zasady, tam nie można na tym polegać.
Write'y o rozmiarze mniejszym niż bufor pipe'a nie powinny być przeplecione write'ami z innych pipe'ów, ale jeśli są większe nie ma takiej gwarancji.

1.6.3 fcntl

fcntl pozwala wykonać komendę podaną w argumencie cmd na deskryptorze pliku

```
int fcntl(int fd, int cmd, [data])
```

1.7 Sygnały

Sygnały informują o asynchronicznym zdarzeniu, często o błędzie.

1.7.1 Źródła sygnałów

1. Ctrl-C wysyła **SIGINT** do wszystkich procesów z foregroundu
2. Ctrl-\ wysyła **SIQUIT** do wszystkich procesów z foregroundu
3. Dzielenie przez 0 powoduje wysłanie sygnału **SIGFPE** do procesu
4. Procesy mogą wysyłać sygnały poprzez syscall kill 1.7.3

Do adresata nie docierają sygnały $SIGKILL$ i $SIGSTOP$, odbiera je kernel i odpowiednio zabija / pauzuje proces.

1.7.2 Ważne sygnały

- **SIGINT** - żądanie przerwania od użytkownika
- **SIGKILL** - absolutne zakończenie procesu bez sprzątania
- **SIGTERM** - domyślny sygnał wysyłany przez funkcję kill
- **SIGSTOP**
- **SIGILL**
- **SIGALRM** - sygnał wykorzystywany w syscallu alarm
- **SIGCHLD**

Sygnały SIGSTOP i SIGKILL nie mogą być ignorowane ani blokowane. SIGTERM, SIGINT i SIGQUIT mają jako domyślną obsługę zakończenie procesu, przy czym SIGTERM być może trochę posprząta, zanim całkowicie zakończy proces, a SIGQUIT zrobi core dump.
Prośba o wysłanie SIGALARM anuluje wszystkie poprzednie alarmy. Podanie 0 jako arguemnt do funkcji alarm anuluje wszystkie alarmy. Przenoszą się przez execve, ale nie są dziedziczone przez procesy stworzone przez fork.

1.7.3 Syscall kill

Wysyła sygnał do danego procesu o tym samym ID użytkownika.

```
int kill(pid_t pid, int sig);
```

Adresaci:

- pid>0 proces, którego id jest równe pid
- pid=0 procesy z tej samej grupy co wysyłający
- pid=-1 wszystkie procesy (do których można wysyłać)
- pid< -1 wszystkie procesy z grupy o ID równym |pid|

Syscall raise pozwala wysłać sygnał do samego siebie

```
int raise(int sig);
```

1.7.4 Przekierowywanie sygnałów

1.7.4.1 Signal - ISO C

Signal pozwala nam zmienić domyślne działanie sygnału. Nie jest częścią POSIXa, jest częścią standardu C.

```
void (*signal(int signo, void (*func)(int)))(int);
```

Jako drugi argument możemy podać wskaźnik do funkcji, która ma obsługiwać sygnał albo ustawić flagę na **SIG_IGN**, co pozwala na całkowite zignorowanie sygnału. Możemy też przywrócić domyślną obsługę poprzez ustawienie flagi na **SIG_DFL**. Funkcja signal ustawia obsługę sygnału tylko na jeden raz, a później przywracana jest domyślna.

1.7.4.2 Sigaction - POSIX

Sigaction także pozwala nam zmienić domyślne działanie sygnału, jest zdefiniowany w POSIXie.

```
int sigaction(int sig, const struct sigaction *restrict act,
        struct sigaction *restrict oact);
```

Struktura sigaction jest zdefiniowana jako

```
struct sigaction {
  void    (*sa_handler)(int); // wskaźnik do funckji, która
        będzie obsługiwać sygnał lub SIG_IGN / SIG_DFL
  void    (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t  sa_mask; // zbiór sygnałów, które chcemy zablokować
  int    sa_flags; // flagi
  void    (*sa_restorer)(void);
};
```

Flagi:

- **SA_SIGINFO** w sigaction pozwala nam na uzyskanie dodatkowych informacji o procesie.
  Jeśli flaga SA_SIGINFO jest ustawiona, to musimy zdefiniować funkcję obsługującą sygnały jako:

  ```
  void func(int signo, siginfo_t *info, void *context);
  ```

- **SA_RESTART**, po jej ustawieniu, jeśli odebranie sygnału przerwało jakąś funkcje blokującą, to wznowi on działanie po obsłudze procesu, zamiast się zakończyć z błędem. Przydatne np. do reada i write'a.

- **SA_NOCLDSTOP** wyłącza generowanie sygnałów SIGCHLD, kiedy proces dziecko się zakończy. Przydatne, gdyż ustawienie wprost obsługi SIGCHLD na SIG_IGN jest niezdefiniowane w POSIXie i mogą się dziać złe rzeczy. W Linuxie jednak można ignorować SIGCHLD, aby zapobiec powstawaniu procesów zombie.

UWAGA! takie same sygnały w POSIXie nie są kolejkowane, to znaczy kiedy dostaniemy wiele takich samych sygnałów na raz to nie ma gwarancji, że handler wywoła się tyle razy, ile sygnałów dostaliśmy. Jeśli jednak dostaliśmy wiele różnych sygnałów, to mamy gwarancję, że handler dla każdego typu sygnału wykona się co najmniej raz.

1.7.5 sigprocmask

sigprocmask pozwala nam zablokowywać/odblokowywać procesy zgodnie z podaną maską.

```
int sigprocmask(int mode, const sigset_t *restrict set,
  sigset_t *restrict oset);
```

Dostępne tryby

- SIG_BLOCK zablokuj podane sygnały
- SIG_SETMASK zbiór zablokowanych sygnałów ma być podanym zbiorem (wcześniejsze ustawienia są zapominane)
- SIG_UNBLOCK odblokuj podane sygnały (nawet jeśli były zablokowane przed zmianami, nie przywraca poprzedniego stanu)

1.7.6 sigsuspend

Zmienia maskę na podaną, po czym zawiesza się aż do otrzymania niezablokowanego sygnału, po czym przywraca poprzednią maskę.

```
int sigsuspend(const sigset_t *sigmask);
```

1.7.7 sigpending

sigpending zapisuje w podanym sigsecie wszystkie sygnały, które zostały zablokowane i oczekują na wywołanie, przydatne gdy chwilowo musieliśmy zablokować sygnały podczas wykonywania krytycznych operacji

```
int sigpending(sigset_t *set);
```

1.7.8 Czekanie na wiele deskryptorów na raz

1.7.8.1 select

select pozwala nam na czekanie na wiele deskryptorów na raz, aż będzie można do nich pisać / czytać. Podajemy tablicę deskryptorów do readfs / writedfs / errordfs w zależności od rodzaju. Później możemy sprawdzić, czy konkretny deskryptor jest gotowy przez FD_ISSET. Gotowość niestety może czasami okazać się sytuacją, kiedy po drugiej stronie nie ma nikogo... Jeśli działa, to trzeba go resetować przez FD_CLEAR w każdym wywołaniu, bo zmienia on readfs.

```
int select(int nfds, fd_set *restrict readfds,
  fd_set *restrict writefds, fd_set *restrict errorfds,
  struct timeval *restrict timeout);
  void FD_CLR(int fd, fd_set *fdset);
  int FD_ISSET(int fd, fd_set *fdset);
  void FD_SET(int fd, fd_set *fdset);
  void FD_ZERO(fd_set *fdset);
```

1.7.8.2 pselect

to samo co select, ale ma mniej ograniczeń i można mu podać dodatkowo maskę sygnałów, które chcemy zablokować podczas czekania i nie trzeba go resetować.

```
  int pselect(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask);
```

1.7.8.3 poll

Rozdział 2

OS-internals

2.1 Multiprogramming

2.2 Scheduler

Scheduler zarządza dostępem procesów do czasu procesora, zatrzymując co niektóre i wznawiając inne.

2.2.1 Schedulery czasu rzeczywistego

Procesy mają przypisane priorytety i pierwszy wykona się ten o najmniejszym priorytecie spośród gotowych.

2.2.1.1 Round Robin

Każdy proces dostaje kwant czasu i jest ustawiony w kolejce. Proces może działać tak na jak długo wystarczy mu kwantu, a kiedy go zużyje trafia na koniec kolejki i otrzymuje nowy. Jeśli proces zablokuje się w oczekiwaniu na zasoby przed zużyciem kwantu, to po odblokowaniu trafia na początek kolejki z czasem, jaki mu pozostał.

Bardzo istotny jest dobór wielkości kwantu, za duży powoduje, że system ma zbyt długi czas reakcji, zbyt mały powoduje dużo zmarnowanej pracy na zmiany

kontekstu. Zwykle używa się między 20-50ms.

2.2.1.2 Priority Scheduling

Bazuje na round robin, kwanty czasu zależą od priorytetu, procesy o niższym priorytecie mają pierwszeństwo.

2.2.1.3 Guaranteed Scheduling / Fair share scheduling

Każdy proces ma zagwarantowane x% procesora, scheduler dobiera procesy tak, aby dopełnić zobowiązania.

2.2.1.4 Lottery Scheduling

Każdy proces dostaje kilka żetonów z puli. Scheduler losowo dobiera żeton i przydziela procesor jego właścicielowi. Współpracujące procesy mogą sobie przekazywać żetony.

2.2.1.5 A jak to jest w minixie?

W minixie używany jest priority scheduling z dodatkową degracją (zwiększeniem) priorytetu. Procesy użytkownika zwiększają swój priorytet, jeśli zużyły cały kwant i dwa razy z rzędu dostały procesor, zmniejszają w przeciwnym przypadku.

2.2.2 Przerwania

Scheduler pracuje podczas przerwań, w tym w szczególności przerwań zegarowych. Im więcej przerwań zegarowych, tym efektywniej można zarządzać pracą procesów, ale z drugiej strony tym więcej czasu procesora jest wykorzystane przez scheduler i na zmianę kontekstu. W minixie jest 60 przerwań na sekundę

2.2.3 Wątki

Każdy proces posiada przynajmniej jeden wątek. Procesy mają oddzielne przestrzenie adresowe, a wątki mogą je współdzielić. Wątki jednego procesu mają wspólne między innymi:

- Wskaźnik do segmentu tekstowego (PM)
- Wskaźnik do segmentu danych (PM)
- Wskaźnik do segmentu bss (PM)
- Wskaźnik do stosu (K)
- Stan procesu (K)
- Licznik programu (K)

K - kernel, PM - process management

2.2.4 Zachowanie Procesów

Dzielimy procesy na kategorie i każda z nich jest traktowana inaczej. Scheduler dba o to, żeby podobne procesy były podobnie traktowane i utrzymuje równe obciążenie wszystkich części systemu.

- **Interaktywne** - Mają najwyższy priorytet. Scheduler stara się, aby szybko reagowały na input. W zależności od czasu trwania zadania scheduler pozwala sobie na opóźnienie reakcji (nie zrobi nikomu różnicy, jeśli ls wypisze coś w 0.05 sekundy zamiast 0.01).
- **Wsadowe** - wykonywanie serii zadań (programów) przez komputer. Zazwyczaj kolejne zadania są ze sobą powiązane: dane wyjściowe jednego programu przekazywane są kolejnemu programowi, któremu służą jako dane wejściowe itd.. Scheduler stara się maksymalizować zadania na godzine (throughput), minimalizować czas między zakolejkowaniem a zakończeniem zadania i maksymalizować obciążenie procesora.
- **Czasu rzeczywistego** - są ograniczone przez CPU. Mają najniższy priorytet, a scheduler stara się w nie nie wtryniać i jeśli to możliwe, szczególnie w systemach wielowątkowych, oddaje im wątek na wyłączność.

2.3 Komunikacja pomiędzy procesami

2.3.1 Sekcje krytyczne

To fragmenty programu z odwołaniami do współdzielonej pamięci. Ważne jest, żeby

- maksymalnie jeden proces/wątek na raz znajdował się w sekcji krytycznej
- żaden proces poza sekcją krytyczną nie blokował innego procesu
- żaden proces nie czekał w nieskończnoność na wejście do sekcji krytycznej
- wszystko działało niezależnie od szybkości i liczby procesorów

2.3.2 Spin Lock

Spin Lock to aktywne czekanie, w kółko sprawdzamy czy już możemy dalej pracować. Zużywa dużo czasu procesora, zwykle nieefektywne.

2.3.3 Rozwiązanie Peterson'a

Rozwiązanie gwarantuje, że dwa procesy nigdy nie będą w sekcji krytycznej na raz, każdy proces kiedyś doczeka się na zasoby i czas oczekiwania nie będzie nieskończony. Jeśli tylko wątki nie robią podejrzanych rzeczy...

```
int turn, interested[2]; // shared
void enter_region(int process){
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while ((turn==process) && (interested[other]==TRUE));
}
void leave_region(int process){
  interested[process] = FALSE;
}
```

2.3.4 Priority Inversion Problem

Spin-locki mogą mieć uzasadnienie, kiedy czas oczekiwania jest krótki, więc usypianie i budzenie wątku miało by stosunkowo duży narzut. W systemach czasu rzeczywistego może to jednak być problematyczne, ponieważ proces o wysokim priorytecie (i bliskim deadlinie) będzie pracował, zapętlając się w oczekiwaniu na zasób, który jest zablokowany przez proces o niskim priorytecie.

2.3.5 Semafory

To chroniona zmienna służąca do kontroli dostępu przez wiele procesów do wspólnego zasobu. Zlicza ona, ile elementów ma dostęp do zasobu. Przykładem semafora jest mutex, jest to semafor binarny (zajęte / niezajęte).

2.3.6 Monitory

To obiekty, które mogą być bezpiecznie używana przez kilka wątków. Metody monitora chronione są przez mutexy, dzięki czemu w dowolnym momencie czasu z dowolnej metody może korzystać tylko jeden wątek.

2.3.7 Przesyłanie komunikatów

2.4 Architektura

2.4.1 Monolithic Kernel

Procesy użytkownika pracują w dwóch trybach: **trybie użytkownika** i **trybie jądra**. Przechodzimy z trybu użytkownika do trybu jądra poprzez przerwania systemowe i syscalle. Wiele procesów może pracować równocześnie w trybie jądra.

2.4.2 Micro Kernel

Mikrokernele minimalizują ilość kodu pracującego w trybie uprzywilejowanym. Funkcjonalności systemu są zaimplementowane jako serwery, z którymi komunikujemy się przez wiadomości.

Jądro jest odpowiedzialne za multiprogramming, przesyłanie wiadomości oraz obsługę urządzeń. Procesy użytkownika nigdy nie pracują w trybie jądra.

2.5 Kernel w Minixie

2.5.1 Message Parsing

W Minixie jest system wymiany wiadomości między procesami poprzez:

- SEND - wysyła wiadomość i blokuje się, dopóki nie zostanie dostarczona (block on full buffer)
- RECEIVE - blokuje proces, póki nie dostanie wiadomości (block on empty buffer)
- SENDREC - wysyła wiadomość i blokuje się, póki nie dostanie odpowiedzi
- NOTIFY

Wiadomości mogą być wysyłane do zadań jądra tylko i wyłącznie poprzez SENDREC.
Każdy proces ma maskę operacji wiadomościowych, z których może korzystać. Procesy użytkownika mogą używać wyłącznie SENDREC. Wskaźnik do pamięci musi znajdować się w pamięci procesu wołającego. Wiadomości mogą być buforowane jak notify i wtedy można wysyłać / oczekiwać wiadomości od wielu procesów - wiadomości od któregokolwiek z potencjalnych odbiorców / nadawców. Mogą też nie być buforowane (rendezvouz message-passing) jak w send i receive. Wtedy procesy blokują się w oczekiwaniu na wysłanie / odbiór, więc jest ryzyko zakleszczenia.

2.5.2 Zegar

clock handler() - obsługa przerwań zegara - aktualizacja czasu, „księgowość", sprawdzanie alarmów i potrzeby wywłaszenia.
clock task() - wywłaszczanie procesów, których czas się wyczerpał, uruchamianie alarmów.

2.5.3 Kernel Calle

Każdemu syscallowi odpowiada adekwatny kernel call, funkcja działająca wewnątrz jądra. Ich nazwy tworzymy dodając sys_ przed nazwą syscalla np sys_read().

2.5.4 Serwery

Serwery w minixie realizują kernel calle. Wchodzą w ich skład

- **pm (process manager)** – zarządza procesami, obsługuje funkcje takie jak: fork(), exec(), exit(), kill()
- **sched (scheduler server)** – bierze udział w szeregowaniu procesów
- **vfs (virtual file system)** – obsługuje funkcje związane z systemem plików, na przykład: chdir(), open(), read(), select(), write()
- **vm (virtual memory manager)** – zarządza pamięcią wirtualną
- **rs (reincarnation server)** – przechowuje i udostępnia informacje o endpointach serwerów, uruchamia serwery i sterowniki, które nie są uruchamiane w tym samym czasie co jądro, ale w kolejnej fazie uruchamiania systemu, nadzoruje ich działanie i w przypadków błędów uruchamia je ponownie. Jest to możliwe dzięki temu, że proces rs jest rodzicem uruchamianych przez siebie procesów serwerów i sterowników.
- **is (information server)** – pomocniczy, dostarcza informacji na temat działania sterowników i serwerów
- **ds (data store server)** – pomocniczy, informuje procesy o zmianie konfiguracji po wznowieniu działania serwera, spełnia rolę małej bazy danych

```
1 PUBLIC void sys_task()
2 {
3 /* Main entry point of sys_task.  Get the message and dispatch on type. */
4   /.../
5
6   initialize();
7
8   while (TRUE) {
9       /* Get work. Block and wait until a request message arrives. */
10      receive(ANY, &m);
11      call_nr = (unsigned) m.m_type - KERNEL_CALL;
12      caller_ptr = proc_addr(m.m_source);
13
14      /* See if the caller made a valid request and try to handle it. */
15      if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
16    kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
17    result = ECALLDENIED;       /* illegal message type */
18      } else if (call_nr >= NR_SYS_CALLS) {   /* check call number */
19    kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
20    result = EBADREQUEST;       /* illegal message type */
21      }
22      else {
23          result = (*call_vec[call_nr])(&m);  /* handle the kernel call */
24      }
25
26      /* Send a reply, unless inhibited by a handler function. Use the kernel
27       * function lock_send() to prevent a system call trap. The destination
28       * is known to be blocked waiting for a message.
29       */
30      if (result != EDONTREPLY) {
31      m.m_type = result;          /* report status of call */
32          if (OK != (s=lock_send(m.m_source, &m))) {
33              kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
34          }
35      }
36  }
37 }
```

Ten krótki wycinek kodu został załączony za zachętą Osoby Serdecznie Zaangażowanej w Badania nad Kodem Minixa i z dedykacją dla niej

2.6 Organizacja pamięci

Rodzaj Pamięci	czas dostępu
Rejestry	1-3 ns
Level 1 Cache	2-8 ns
Level 2 Cache	5-12 ns
Memory	10-60 ns
Hard Disk	3 000 000-10 000 000 ns

Tabela 2.1: Hierarchia pamięci

2.6.1 Strategie alokacji

Alokacja w większości systemów jest lazy, tzn. możemy sobie zażądać, ile chcemy pamięci, nawet więcej niż jest fizycznie dostępne, ale system de facto jej nam nie przydziela, dopóki jej nie dotkniemy.

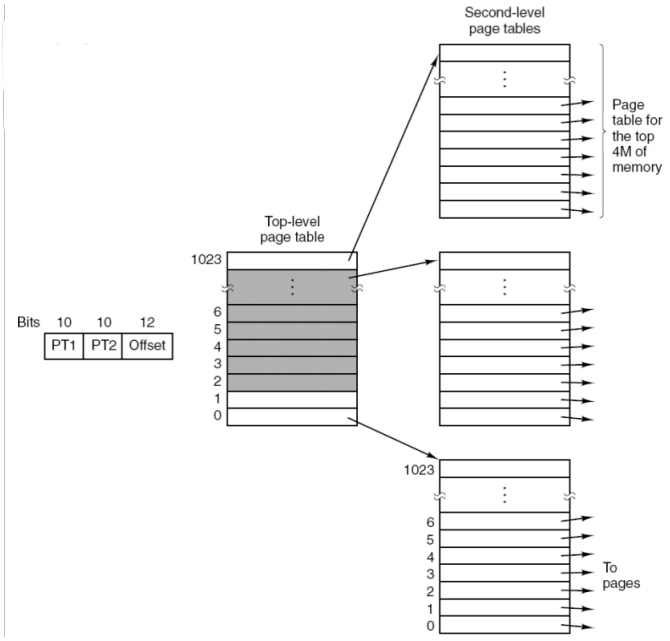
- First Fit - ani zły, ani dobry
- Next Fit - podobnie jak first fit
- Best Fit - wrzuca do najmniejszego w jakim się zmieści, niby brzmi fajnie, ale zostawia małe dziury, przez co potem ciężko coś wcisnąć
- Worst Fit - wrzuca na początku największego dostępnego fragmentu wolnego, fajny bo zostawia duże dziury

2.6.2 Pamięć wirtualna

Można tak gospodarować pamięcią RAM, żeby przydzielać procesom więcej pamięci niż jest fizycznej pamięci w komputerze. Nieużywane akurat dane przenosi się w inne miejsce (np. na dysk), a jeśli są potrzebne, to przywraca się je w jakimś dostępnym miejscu. Jeśli go zabraknie, to inne dane przenosi się na dysk. W UNIXowych systemach jest na to specjalnie wydzielona partycja na dysku (partycja „swap").

2.6.3 Stronicowanie

Obraz procesu i pamięć fizyczna dzielone są na strony, wszystkie o tym samym rozmiarze. Dla odróżnienia stron z obrazem procesu od stron pamięci fizycznej te drugie nazywamy ramkami. Strony mają rozmiar od kilku do kilkudziesięciu kilobajtów (zwykle 4kB w systemie 32-bitowym). Procesowi można przydzielić ramki w różnych miejscach, niekoniecznie obok siebie. Żeby odczytać zawartość strony, trzeba przetłumaczyć adres logiczny (numer strony, przesunięcie) na adres fizyczny.
Stronicowanie może być wielopoziomowe. W takim przypadku adres logiczny to pozycja tablicy w tablicy tablic, pozycja strony w tablicy stron, przesunięcie na stronie.



https://lass.cs.umass.edu/ shenoy/courses/spring20/lectures/Lec11.pdf

2.6.3.1 Strategie ładowania stron

- Utopijna **strategia optymalna** - zastąp stronę, która będzie najpóźniej ładowana. Nie do zrealizowania online.
- **FIFO** - Wybierz ostatnio nieużywaną, mamy dodatkowe bity w tabeli stron,

accessed i dirty. Bierzemy w kolejności (not referenced, not modified) → (not referenced, modified) → (referenced, not modified) → (referenced, modified), w ramach klasy losowo.

- **Second Chance / Clock PRA** - modyfikacja FIFO, usuwamy tylko, jeśli referenced = 0, wpp czyścimy referenced i przesuwamy na koniec kolejki. Zamiast kolejki lepiej używać bufora cyklicznego (clock).
- **Least recently used PRA** - zastąp stronę, która była używana najdawniej. Implementacja wymaga wsparcia od hardware'u.
- **Not frequently used PRA** - po każdym przerwaniu kasuj bit referenced i zwiększaj licznik stronom, które mają ten bit ustawiony. Usuwaj te z najmniejszą wartością licznika.
- **Aging PRA** - po każdym przerwaniu dla każdej strony przesuń bity licznika o 1 w prawo, uzupełniając bitem referenced, po czym skasuj bit referenced. Usuwaj te z najmniejszą wartością licznika.

2.6.3.2 Rozmiar strony

Przyjęło się używać wzoru:

$$p = \sqrt{2se} \tag{2.1}$$

Gdzie:
s - średni rozmiar procesu
e - rozmiar wiersza w tabeli stron
p - rozmiar strony

W większości systemów pracujących w 32 bitowej architekturze przyjmuje się:
s = 1MB, e = 8 $\implies$ $p = 4KB$

2.6.4 Segmentacja

Segmenty mogą mieć różne długości, dlatego adres logiczny to numer segmentu (mapowany na adres bazowy, czyli adres początku segmentu) i długość danych do

przeczytania. Podział na segmenty jest mniej więcej podziałem na logiczne części programu, więc niektóre z nich mogą być całkiem długie.

2.6.5 W połączeniu

Może być trudno znaleźć spójny fragment pamięci dla długiego segmentu, dlatego łączy się stronicowanie i segmentację i mamy segmenty składające się z ramek. Przesunięcie wewnątrz segmentu traktowane jest jako adres w pamięci stronicowanej (numer strony, przesunięcia na stronie). Zamiast adresu bazowego segmentu używany jest adres tablicy stron tego segmentu, w której można znaleźć numer ramki.

2.6.6 Jak to działa w Linuxie?

Jest tam więcej stronicowania, ponieważ łatwiej sobie z tym radzić, ale nie tylko stronicowanie.

2.6.7 A w Minixie?

Wersje Minixa starsze niż 3.1 nie miały stronicowania ani wirtualnej pamięci i cały proces musiał być załadowany do RAMu. Mogła się pojawić fragmentacja. Nowsze mają coś ze stronicowania i serwer VM, który jest oddzielony od PMa. Procesy były podzielone na segmenty.

2.7 Drivery

Troszkę się zapędziliśmy, bo tego chyba nie przerabialiśmy, ale jest na slajdach. Sekcja niekompletna. Anyway, postanowiliśmy tego nie usuwać, a nóż komuś się przyda.

Drivery służą do interakcji z innymi urządzeniami. Dzielą się na:

Urządzenia blokowe - mają swobodny dostęp do danych, czas dostępu do każdego fragmentu zbliżony, Dane zorganizowane w bloki, (np. CD/DVD-ROM)

Urządzenia znakowe - operują na ciągach znaków, dostęp sekwencyjny (np. klawiatura, mysz, modem)

2.7.0.1 I/O port

W architekturze x86 procesor ma do dyspozycji 216 8-bitowych portów. Porty mogą być grupowane dla zwiększenia transferu. Urządzenia mają oddzielną przestrzeń adresową.

2.7.0.2 Memory mapped I/O

Można też zmapować rejestry urządzeń do przestrzeni adresowej pamięci procesora. Dostęp jak do pamięci (np. MOV).

2.7.0.3 Spooling

Tryb realizowania operacji wejścia-wyjścia w odniesieniu do urządzeń o dostępie wyłącznym. Cała zawartość strumienia jest buforowana zanim zostanie przesłana w miejsce docelowe.

Rozdział 3

Rozwiązania egzaminów

3.1 Egzamin 2013/2014

Zadanie 1
konieczność wielokrotnego przełączania procesów spowodowana dużą liczbą wywołań systemowych

Zadanie 2
Mamy takie zależności (x → y ozn x pojawi się przed y):
c1 → p1, c2 → p2, p1 → p2
1. nie
2. nie
3. tak
4. tak

Zadanie 3
mniejszy koszt przełączania procesora pomiędzy wątkami niż pomiędzy procesami

Zadanie 4
1. micro
2. żaden – w mikrojądrze tylko jądro pracuje w trybie uprzywilejowanym, a w systemie monolitycznym procesy mogą wykonywać jedynie kod jądra w uprzywi-

lejowaniu

3. mono – procesy same przełączają się na bycie driverami, bo nie ma serwerów

4. mono – na przykład żeby wykonać syscall

Zadanie 5

1. tak 2. nie 3. nie 4. nie

Zadanie 6

1. nie

2. tak – kiedy procesy równolegle zawieszą się na semaforach AB oraz BA

3. nie – przykład notify

4. tak – przykład send, receive

Zadanie 7

1. pag

2. seg, pag

3. seg, pag

4. pag

5. seg

6. pag

7. pag

Zadanie 8

1. nie

2. tak

3. tak

4. tak

Zadanie 9

1. tak

2. tak

3. 4. nie

3.2 Egzamin 2014/2015

Zadanie 1
5

Zadanie 2
Jeśli otwieramy pipe O_RDONLY lub O_WRONLY i po drugiej stronie nikogo nie ma.

Zadanie 3
Jeśli write jest przerwany, to zwróci rozmiar tego, co udało się zapisać lub -1, jeśli nic.

Zadanie 4
Sygnały się nie stackują.

Zadanie 5
select

Zadanie 6
send
Nie jest buforowany, więc jest blokujący.
Przenosi dane.
notify
Jest buforowany i nieblokujący.
Powiadomienie nie jest koniecznie od konkretnego procesu.
Nie przenosi danych.

Zadanie 7
3.1.0: FS, Disc Driver, System Task
3.2.2: VFS, ???, Disc Driver, System Task

Zadanie 8
throughput - ?
turnaround time - zmniejszy (mniej czasu traconego na zmianę kontekstu)
response time - zwiększy (dłuższy czas czekania w kolejce)

Zadanie 9
Prawdopodobnie porty

Zadanie 10
Część pamięci może być przeniesiona na dysk bez wiedzy procesu.

3.3 Egzamin 2015/2016

Zadanie 1
i-node, tablica tablic, tablica wskaźników, node z plikiem

Zadanie 2
Jeśli plik jest odpowiednio duży, to cat zapełni cały pipe i dziecko zawiesi się na blokującym write. W tym samym czasie rodzic czeka na jego zakończenie i takczekają, i czekają.nikt nie czyta z pipe'a. Jeśli plik jest mały, to dziecko nie zablo-kuje się przy pisaniu i zakończy, więc rodzic będzie mógł odnotować to i przejśćdo dalszej pracy.

Zadanie 3
Signaled
Dziecko najprawdopodobniej zostanie zakończone sygnałem podczas wykonywania cat. Handlery nie przenoszą się przez exec.

Zadanie 4
Pod rm wywołany jest unlink, ale w systemie mogło być więcej linków do tego

pliku.

Zadanie 5

W kernelu jest tabela, w której zapisane są uprawnienia. Wszystkie procesy użytkownika mają jeden wspólny rekord, a pozostałe procesy oddzielne. Są tam zapisane maski kernel calli, które dany proces może wywoływać.

Zadanie 6

Jeśli proces nie wykorzystał w pełni swojego kwantu czasu, to po odblokowaniu zostaje dodany na początek swojej kolejki. Dodatkowo procesy, które dwa razy z rzędu dostały czas procesora i wykorzystały swój kwant są degradowane.

Zadanie 7

send
Nie jest buforowany, więc jest blokujący.
Przenosi dane.
notify
Jest buforowany i nieblokujący.
Powiadomienie nie jest koniecznie od konkretnego procesu.
Nie przenosi danych.

Zadanie 8

nie?
tak
tak
tak?
nie
nie
nie

Zadanie 9

fx, fy, fz
fz, fx, fy

Zadanie 10
Tak działa cache'owanie.

3.4 Egzamin 2016/2017

Zadanie 1
Write nie jest buforowany, więc zapisanie każdego znaku wymaga osobnego syscalla.

Zadanie 2
Jeśli plik jest odpowiednio duży, to cat zapełni cały pipe i dziecko zawiesi się na blokującym write. W tym samym czasie rodzic czeka na jego zakończenie i tak czekają, i czekają.nikt nie czyta z pipe'a. Jeśli plik jest mały, to dziecko nie zablokuje się przy pisaniu i zakończy, więc rodzic będzie mógł odnotować to i przejść do dalszej pracy.

Zadanie 3
To procesy, które czekają na osierocone procesy, żeby nie stały się zombie, kiedy rodzic już sie zakończył i nie odnotował ich stanu. Liczba 1613 to reaper, a 1 na Minixie to init.

Zadanie 4
Jeśli wyczerpał swój kwant, trafi na koniec, a jeśli zostało mu jeszcze troszkę czasu, to na początek.

Zadanie 5
komunikacja z (fizycznym) dyskiem
message-passing pomiędzy driverem a procesem

Zadanie 6

Można wykorzystać funkcję select albo poll, żeby dowiedzieć się, że któryś z pipe'ów jest gotowy na wpisanie do niego danych.

Zadanie 7

Handler obsługujący przerwania od klawiatury dodaje hdd do kolejki schedulera.

Zadanie 8

Do tego służą przerwania zegarowe, które budzą scheduler i pozwalają mu robić porządki.

Zadanie 9

Może się tak stać, jeśli strona w pamięci jest dwukrotnie zmapowana w przestrzeni procesu, czyli arr[42] w pamięci fizycznej odpowiada dwóm miejscom w pamięci wirtualnej. Można to zrobić funkcją mmap.

Zadanie 10

To sposób organizacje miejsca na dysku. Polega na tym, że dane upychane są w dostępnych miejscach pamięci, które mogą być zupełnie porozrzucane, ale co jakiś czas reorganizujemy dane tak, żeby fragmenty pliku były względnie blisko siebie. Pozwala to zmniejszyć czas odczytu.

3.5 Egzamin 2017/2018

Zadanie 1

Możliwe odpowiedzi: 1. read może wczytać fragmentaryczne dane 2. może pojawić się sygnał i błąd EINTR 3. exec może się nie udać i dziecko się zapętli Program dostaje zupełnie nieprzewidywalny argument.

Zadanie 2

Chodzi o to, że handlery nie zachowują się przy execu, a maska tak (SIG_IGN działa jak ustawianie maski). Dlatego proces dziecka będzie miał blokowane SIGUSR2

i SIGINT, ale SIGUSR1 nie. Program wypisze 1 kropkę, bo SIGUSR1 zakończy proces potomny.

Zadanie 3

To procesy, które czekają na osierocone procesy, żeby nie stały się zombie, kiedy rodzic już sie zakończył i nie odnotował ich stanu. Liczba 1613 to reaper, a 1 na Minixie to init.

Zadanie 4

1. utworzenie nowego wątku
2. zablokowanie procesu
3. zakończenie procesu
4. wyczerpanie kwantu
5. przerwanie zegarowe
6. przerwanie IO

Zadanie 5

?

Zadanie 6

1. współdzielona pamięć
2. porty IO
3. przerwania

Zadanie 7

Zalety:

1. Pamięć nie jest rezerwowana na zapas - mniejszy narzut.
2. Pojedyńcza operacja jest szybsza.
3. Procesy uruchamiają się szybciej.

Wady:

1. Opóźnienie przy pierwszym dostępie do pamięci związane z obsługą page faulta przez system operacyjny.
2. Bardziej złożona obsługa.

3. Duża częstotliwość występowania page faultów

Zadanie 8

System utworzy stos dla procesu, na którym na początku jest są dane z programu. Na szczycie zostaną ułożone argumenty, a potem uruchomiony proces.

Zadanie 9

Może się tak stać, jeśli strona w pamięci jest dwukrotnie zmapowana w przestrzeni procesu, czyli arr[42] w pamięci fizycznej odpowiada dwóm miejscom w pamięci wirtualnej. Można to zrobić funkcją mmap.

Zadanie 10

Mógł być wykorzystywany przez handler.
Mógły być tam ułożone informacje o procesie.

3.6 Egzamin 2018/2019

Zadanie 1

14

Zadanie 2

Select poinformuje o gotowości do odczytu zbyt wcześnie, ponieważ odczyt z pipe'a zamkniętego po drugiej stronie jest traktowany jako otwarty. Więc po zamknięciu pipe'a przez producenta powstaje aktywne czekanie.

Zadanie 3

Jeśli przez 3 sekundy nie pojawi się input, to przyjdzie alarm, który nie jest obsługiwany i proces się zakończy.

Zadanie 4

To procesy, które czekają na osierocone procesy, żeby nie stały się zombie, kiedy

rodzic już sie zakończył i nie odnotował ich stanu. Liczba 1613 to reaper, a 1 na Minixie to init.

Zadanie 5

Użycie printf i exit w handlerze jest ryzykowne - co jeśli się nie powiodą? On ne fait pas des choses comme ça

Zadanie 6

Zwykły użytkownik nie może wywołać sys_fork, bo komunikację z kernelem wykonują serwery i drivery (np. PM czy VFS). Dla użytkowników przeznaczone są zwykłe syscalle, które komunikują się przez serwery.

Zadanie 7

1. powstanie nowego wątku
2. wyczerpanie kwantu
3. doczekanie się odpowiedzi lub dostępu do zasobów

Zadanie 8

Driver chce doczytać bliższe sektory na dysku, by przyśpieszyć przyszłe dostępy do pamięci w tych okolicach (pewnie też będą niedługo potrzebne - założenie często słuszne).

Zadanie 9

Dostęp do początku pliku jest od razu, a żeby dostać się do końca trzeba przeczytać pośrednie bloki. Asymptotycznie czas jest logarytmiczny (mamy drzewo i-nodów), praktycznie stały rzędu 3.

Zadanie 10

page fault frequency mówi jak często programy odwołują się do pamięci obiecanej przez system, ale nie przydzielonej. Jeśli program nigdy nie odwołał się albo nie odwoływał przez dłuższy czas do jakiegoś miejsca, to może być przeniesione na dysk i kiedy nastąpi page fault, to trzeba je ściągnąć z powrotem do pamięci podręcznej. Jeśli pojawia się bardzo dużo page faultów, to może tak być, że system

obiecał zbyt wielu zbyt wiele i brakuje pamięci pod ręką, więc trzeba swapnąć któryś proces na dysk.

Zadanie 10

Printf jest buforowany, a dzieciom kopiuje się bufor rodzica, w którym już są kropki.

3.7 Egzamin 2019/2020

Zadanie 1

Wykorzystamy oznaczenia p – pid programu
q – pid dziecka
r – pid reapera
x – pid rodzica programu
„Zwykle" (wszystko się powiedzie, bufor się nie skopiuje) mamy tylko trzy opcje:
Dziecko pracuje pierwsze:
p x
0 q p
q p x

Rodzic pracuje pierwszy, ale kończy się później niż dziecko:
p x
q p x
0 q p

Rodzic pracuje pierwszy i kończy się wcześniej niż dziecko:
p x
q p x
0 q r

Zadanie 2

dup
Kopiuje deskryptor, mamy nowy odnośnik do pliku

Miejsce z którego czytamy to to, w którym skończyliśmy ostatni odczyt
open
Tworzy nowe open file description
Odczyt zaczyna się na początku pliku

Zadanie 3

Alarmy dotyczą tylko procesu, który o niego prosił i przenoszą się przez exec. Dziecko wypisze dwie kropki w warunku i dwie tuż przed wyjściem. Zależnie od czasu alarmu i spanka rodzic dopisze swoją trzecią kropkę lub nie.

Zadanie 4

Sygnały się nie kolejkują, więc handler może wywołać się jeden raz i powstaną zombie.

Zadanie 5

exit w handlerze, dobranoc (jeśli koniecznie (dalece niezalecane) chcemy wyleźć z programu w handlerze to używamy _exit)

Zadanie 6

Jeśli proces nie wykorzystał w pełni swojego kwantu czasu, to po odblokowaniu zostaje dodany na początek swojej kolejki. Dodatkowo procesy, które dwa razy z rzędu dostały czas procesora i wykorzystały swój kwant są degradowane.

Zadanie 7

1. komunikacja między procesami
2. obsługa urządzeń oraz przerwań
3. obsługa IO
4. obsługa zegara

Zadanie 8

open musi znaleźć odpowiedni i-node (długość ścieżki = liczba katalogów po drodze).
lseek tylko zmienia ustawienie w odpowiedniej tablicy

read czyta z dysku parę sektorów

Zadanie 9

Stara się podnosić i restartuje serwery, kiedy się popsują
Przechowuje i udziela informacji o endpointach serwerów

Zadanie 10

page fault frequency mówi jak często programy odwołują się do pamięci obiecanej przez system, ale nie przydzielonej. Jeśli program nigdy nie odwołał się albo nie odwoływał przez dłuższy czas do jakiegoś miejsca, to może być przeniesione na dysk i kiedy nastąpi page fault, to trzeba je ściągnąć z powrotem do pamięci podręcznej. Jeśli pojawia się bardzo dużo page faultów, to może tak być, że system obiecał zbyt wielu zbyt wiele i brakuje pamięci pod ręką, więc trzeba swapnąć któryś proces na dysk.

3.8 Egzamin 2020/2021

Zadanie 1

Rodzi zrobi fork, wywoła main, wypisze kropkę i zakończy pracę, ponieważ f będzie równe 1. Z kolei dziecko wywoła ten sam program, ale z użyciem execa, więc f z powrotem będzie równe 0 i tak zaobserwujemy potencjalnie nieskończenie wiele kropek. Chyba że któryś fork po drodze się nie uda, wtedy program się nie zapętli.

Zadanie 2

..
..
Pierwsze dwie kropki wypisze exec i proces się zakończy. Kolejne dwie zostaną wypisane w nahlerze dziecka i rodzica po otrzymaniu SIGUSR1. Na koniec SIGUSR2 pozbędzie się wszystkich procesów.

Zadanie 3

FD_CLOEXEC gwawrantuje, że deskryptor będzie zamknięty przed exec. Dlatego wypisanie się nie uda, bo stdout zostanie zamknięty.

Zadanie 4

TICTIC

Tryb O_RDWR oznacza, że proces ma otwarty i koniec do pisania i do czytania, a nie da się odróżnić, kto co napisał. Dziecko zapisze TIC, po czym je przeczyta, kiedy rodzic będzie spał i zostawi pusty pipe. Przed wyjście wypisze TICTIC. Rodzic nie będzie miał nic do czytania, ale ponieważ sam trzyma koniec do pisania, to read będzie blokujący i tak się zawiesi w oczekiwaniu.

Zadanie 5

Procesy dzieci bedą się kończyć prawie równo, a sygnały się nie kolejkują. Przybędzie parę zombie.

Zadanie 6

dirty - czy była modyfikowana i trzeba to będzie zapisać na dysku
accessed - czy była niedawno odczytywane, bit jest co jakiś czas zerowany przez system

Zadanie 7

HARD_INT oznacza Hardware Interrupt. W tym przypadku jako przerwanie systemowe pochodzą pewnie od urządzenia pełniącego rolę zegara i oznaczają, że przesunąć przesunąć czas o kolejny tick.

Zadanie 8

Pierwszy odczyt jest powolny bo wymaga sięgnięcia na dysk, a fragmenty pliku mogą nie być w spójnym fragmencie pamięci. Linux cache'uje sobie w RAMie zawartość tego pliku, dzięki czemu kolejne odczyty są szybciutkie.

Zadanie 9

VFS ma stały endpoint, a IPC nie, więc za każdym razem trzeba pytać RSa który proces to IPC.

Zadanie 10

Jeśli wszystko byłoby przekazywane przez wskaźnik to proces, który odbiera wiadomość musiałby sam odczytać/skopiować napis spod wskaźnika, który wskazuje na pamięć innego procesu. Taki mechanizm jest bardziej kosztowny niż po prostu odczytanie z wiadomości, która należy do odbiorcy.

3.9 Egzamin 2021/2022

Zadanie 1

1. AB
2. ABB
3. ABC
4. ABBC
5. ABBCC
6. ABCB
7. ABCBC

Na początku na pewno pojawią się litey A i B. Przy czym B może się pojawić dwa razy. Litera C pojawi się maksymalnie 2 razy (zanim dzieci obu równoległych procesów zabiją całą grupę poprzez kill 0). Jeśli form się nie uda, to kill -1 zakończy wszystkie procesy (w tym wypadku dotrze do tej samej grupy co kill 0).

Zadanie 2

```
int main() {
  int fd[2];
  pipe(fd);
  while(1) {
    write(fd[1], "A", 1);
  }
```

```
}
```

Kiedy w buforze skończy się miejsce, to write się zablokuje.

Zadanie 3
FD_CLOEXEC gwawrantuje, że deskryptor będzie zamknięty przed exec. Dlatego wypisanie się nie uda, bo stdout zostanie zamknięty.

Zadanie 4
Problem polega na tym, że deskryptory są kopiowane (dup2), ale nie są nigdzie zamykane, a jest górne ograniczenie na ich liczbę (16 w Linuxie), więc po pewnym czasie program zakończy się z błędem. Trzeba by zamykać deskryptor po skopiowaniu.

Zadanie 5
W kernelu każdy deskryptor reprezentowany jest przez strukturę struct_fd, która przechowuje wskaźnik do open file description. W ofd sa takie informacje jak tryb otwarcia, flagi, pozycja kursora, etc. Wiele fd może wskazywać na to samo ofd, kazdy fd może wskazywać tylko na jeden ofd. Dla ofd jest co najmniej kilka flag, np O_APPEND lub O_TRUNC, a dla fd tylko jedna nam znana, FD_CLOEXEC.

Zadanie 6
ready – w oczekiwaniu na przydzielenie czasu procesora
blocked – w oczekiwaniu na zdarzenie, np. odpowiedź na wiadomość lub zwolnienie zasobów
swapped – zatrzymany, ze stanem zapisanym na dysku twardym (bo zabrakło pamięci) zombie – zakończony, co nie zostało odnotowane przez rodzica

Zadanie 7
To znaczy, że najpewniej brakuje pamięci, więc trzeba swapować jakiś proces na dysk.

Zadanie 8

Lepiej pozwolić najpierw pracować dziecku, ponieważ nie trzeba kopiować pamięci, dopóki któryś z procesów (dziecko lub rodzic), nie zaczną po niej pisać. Jest duże prawdopodobieństwo, że dziecko wywoła execa, więc i tak trzeba będzie zorganizować dla niego pamięć. Gdyby rodzic zaczął pracę jako pierwszy, pewnie szybko przeszedłby do pisania po pamięci i dwukrotnie trzeba by organizować pamięć dziecka - skopiować tę od rodzica i przygotować nową do exec.

Zadanie 9

Stara się podnosić i restartuje serwery, kiedy się popsują
Przechowuje i udziela informacji o endpointach serwerów

Zadanie 10

Strony będą miały adresy od 4095 do 8000 i trochę, a kolejna do 12000 i trochę, więc x znajdzie się na jednej stronie a y i z razem na innej. Strony mogą być przechowywane w pamięci w dowolnej kolejności, stąd są dwa możliwe scenariusze:
x y z
y z x

3.10 Egzamin 2022/2023

Zadanie 1

3
setsid tworzy nową, niezależną grupę procesów. Powstaną trzy grupy, a w każdej tylko najszybszy z procesów wypisze kropkę, bo potem zakończy pracę wszystkich w grupie.

Zadanie 2

a c d
Printf jest buforowany _exit kończy program bez wcześniejszego opróżniania buforów.

Zadanie 3

Kod biblioteki jest ładowany dynamicznie, czyli jest jakby współdzielony. Dzięki temu zajmuje mniej miejsca i można łatwo uaktualniać go dla wszystkich procesów, ale z drugiej strony trzeba zadbać o jego bezpieczeństwo.

Zadanie 4

SIGKILL oraz SIGSTOP

Zadanie 5

To są procesy zombie.

Zadanie 6

Nie ustawiamy ani act.sa_mask, ani act.sa_flags i nie wiadomo, jak wyglądają, więc nie można oczekiwać, że zachowanie programu będzie zdefiniowane.

Zadanie 7

Jeśli proces nie wykorzystał w pełni swojego kwantu czasu, to po odblokowaniu zostaje dodany na początek swojej kolejki. Dodatkowo procesy, które dwa razy z rzędu dostały czas procesora i wykorzystały swój kwant są degradowane.

Zadanie 8

Prawdopodobnie plik jest pusty poza jakimś znakiem na odległej pozycji. I-nody nie przechowują pustej zawartości, więc zajmują mniej miejsca niż rozmiar pliku.

Zadanie 9

Pamięć wirtualna ma oddzielne adresowanie dla każdego procesu, więc potrzebne są identyfikatory procesów, żeby określić fizyczne adresy.

Zadanie 10

System cache'uje plik. Pierwsze użycie czyta z dysku, kolejne dwa czytają z RAMu.

Bibliografia

[1] Operating Systems: Design and Implementation - Andrew S. Tanenbaum, Albert S. Woodhull

[2] pl.wikipedia.org, zbyt dużo, żeby linkować pojedyńczo

[3] en.wikipedia.org, zbyt dużo, żeby linkować pojedyńczo

[4] Wykłady Prof. Jakuba Kozika z Systemów Operacyjnych na TCS UJ.

[5] man7.org

[6] Strona o systemach operacyjnych mimuw

[7] Materiały Uniwersytetu Massachusetts

[8] SO-Suffering, materiały TCSu

[9] Materiały opracowane przez Weronikę Lorenczyk, Wojciecha Węgrzynka i Jędrzeja Hodora