Rozdział 2

Zarządzanie procesami

Białynicka-Birula Iwona	Kwiatek Michał
Czarnocka Joanna	Leśniewski Radosław
Demusiak Piotr	Nieżurawski Marek
Doroś Anna	Otto Rafał
Gajda Marcin	Paśko Stanisław
Gajda Michał	Pietrak Przemysław
Kałuża Rafał	Rycharski Marek
Karpiuk Piotr	Waleń Tomasz
Kozieradzki Piotr	Wawruch Konrad
Krajewski Tomasz	Wrzesińska Małgorzata
Kulik Bartosz	

Spis treści

Linux jest systemem wielozadaniowym, a więc może się w nim jednocześnie wykonywać wiele procesów. Możemy uruchomić kilka programów, które mogą się ze sobą komunikować, synchronizować się, wpływać wzajemnie na swoje wykonanie. W rozdziale tym opiszemy moduł jądra Linuksa 2.2.12 obsługujący mechanizmy związane z zarządzaniem procesami. Zaczniemy od przedstawienia istotnych struktur danych, wyjaśnienia w jaki sposób procesy wykonywane są na procesorach. Przedstawimy też: mechanizm umożliwiający synchronizacje procesów w postaci semaforów, sygnały Uniksowe, funkcje systemowe służące do obsługi procesów oraz tzw. lekkie procesy - wątki.

2.1 Struktury danych do opisu procesu

2.1.1 Tablica procesów

Linux do przechowywania danych o wszystkich uruchomionych procesach używa tablicy `task`. Jej deklaracja znajduje się w pliku *include/linux/sched.h*:

- **struct task_struct *task[NR_TASKS]**

Jak widać liczba procesów jest ograniczona przez stałą `NR_TASKS`, wynoszącą standardowo 512. Związane są z nią dwie inne wartości:

- **MAX_TASKS_PER_USER** - ograniczenie liczby procesów, jaką może uruchomić zwykły użytkownik, stanowi połowę `NR_TASKS`
- **MIN_TASKS_LEFT_FOR_ROOT** - minimalna liczba wolnych miejsc na procesy, którą może wykorzystać tylko root, standardowo wynosi 4

Stałe te są zdefiniowane w pliku *include/linux/tasks.h*.

Pierwsze pole tablicy wskazuje na zmienną `init_task`. Jej wartość jest na stałe zaszyta w pliku *sched.h*, są to dane pierwszego procesu tworzonego przy starcie systemu, zwanego **idle**. Jest to w zasadzie pseudoproces, nie odpowiada mu żaden plik wykonywalny. Jako jedyny powstaje on bez pomocy funkcji *fork()*. Zadaniem tego procesu jest inicjalizacja niektórych struktur systemowych i stworzenie procesu `init` przy pomocy funkcji *clone()*, później wykonuje nieskończoną pętlę. Ponieważ jego priorytet jest najniższy w systemie, jest wykonywany tylko wtedy, gdy nie ma żadnych procesów gotowych. Nigdy nie ginie i jest zawsze gotowy do działania - wartością jego pola `state` jest `TASK_RUNNING`, jednak nie jest ono nigdy sprawdzane. Jego PID jest równy 0. Proces `init` ma PID równy 1 i wykonuje kod z pliku *sbin/init*.

Nowo powstałemu procesowi przydzielane jest pierwsze wolne miejsce w tablicy, które zatrzymuje do końca istnienia. Kończenie się starszych procesów może powodować powstawanie dziur, dlatego procesy powiązane są w dwukierunkową listę cykliczną (pola `next_task` i `prev_task` w `task_struct`) o początku i końcu w `init_task`. Dzięki temu przy przeglądaniu procesów nie trzeba przechodzić całej tablicy.

2.1.2 Struktura task_struct

Wiadomości wstępne

Uwaga: struktury task_struct nie można łatwo zmieniać przez dopisywanie, zamienianie kolejności czy usuwanie gdyż niektóre funkcje asamblerowe odwołują się do części ze składników przez adresy względne.

```
struct task_struct {
volatile long state;
```

Zmienna state zawiera kod obecnego stanu procesu zmienne te mogą przyjmować następujące wartości

```
TASK_RUNNING 0
TASK_INTERRUPTIBLE 1
TASK_UNINTERRUPTIBLE 2
TASK_ZOMBIE 4
TASK_STOPPED 8
TASK_SWAPPING 16
```

-1 jako kod stanu oznacza błąd. Więcej na temat stanów procesu zostanie napisane w rozdziale 2.2 warto jednak zauważyć, że TASK_RUNNIG może oznaczać zarówno proces gotowy do wykonania, jak i oczywiście aktualnie wykonywany przez procesor. Stan TASK_SWAPPING jest faktycznie nie wykorzystany; tylko w dwóch miejscach w kodzie jądra wystepują do niego odwołania (przy sprawdzaniu stanu i przy jednym ustalaniu maski bitowej poprawiającym ewentualne nieprawidłowe wartości zmiennej `state`) nie jest on jednak nigdy nadawany.

```
long counter;
long priority;
cycles_t avg_slice;
```

Zmienna `counter` przechowuje czas mierzony w tyknięciach przez jaki może jeszcze działać. Natomiast zmienna `priority` jest podstawą do wyznaczenia początkowej wartości `counter` przez funkcję szeregującą `schedule`. Zmienna `avg_slice` pozwala w architekturach wieloprocesorowych oceniać czas wykorzystania procesora przez proces.

```
unsigned long flags;
```

Zmienna `flags` zawiera połączenie flag stanu systemu. `PF_PTRACED, PF_TRACESYS, PF_STATRTING i PF_EXITING. PF_TRACED i PF_TRACESYS` oznaczają, iż proces jest monitorowany przez inny proces za pomocą wywołania systemowego ptrace. `PF_STARTING i PF_EXITING` oznacza, że proces jest akurat rozpoczynany lub kończony. Dostępne są też inne flagi:
`PF_ALIGNWARN`
Niewykorzystana flaga mająca służyć przy drukowaniu komunikatów o błędach przy szeregowaniu, dotychczas nie zaimplementowana, a ma mieć wykorzystanie tylko na procesorach i486.
`PF_FORKNOEXEC`
proces potomny po operacji fork, który nie był jeszcze wykonany.
`PF_SUPERPRIV`
proces ma uprawnienia super użytkownika.
`PF_DUMPCORE`
proces "zrzuca" obszar pamięci.
`PF_SIGNALED`
uśmiercony sygnałem.
`PF_MEMALLOC`
proces w czasie alokacji pamięci.

```
struct exec_domain *exec_domain;
```

W systemie Linux mogą być uruchamiane programy z innych systemów dla procesorów i386 zgodnych ze standardem iBCS2. Procesy zgodne ze standardem iBCS2 różnią się nieznacznie, a zmienna exec_domain przechowuje opis mówiący, który typ systemu UNIX ma być emulowany.

Procesy

```
struct task_struct *next_task, *prev_task;
struct task_struct *next_run, *prev_run;
```

Zmienne `next_task` i `prev_task` pozwalają powiązać wszystkie dostępne w systemie procesy w dwukierunkową listę cykliczna. Zmienne `next_run` i `prev_run` są używane przy „wiązaniu” procesów w kolejki zadań oczekujących.

```
struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr,*p_osptr;
```

Jak już może wiesz Czytelniku proces nie powstaje z niczego. Oprócz procesu `init_task` każdy ma swego ojca oraz, co prawdopodobne, braci i synów. Zmienne `*p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr` opisane bliżej w rozdziale 2.1.5 pokazują na:
`*p_opptr` orginalnego rodzica
`*p_pptr` rodzica
`*p_cptr` najmłodszego potomka
`*p_ysptr` młodszego brata
`*p_osptr` starszego brata

```
struct task_struct **tarray_ptr;
```
Wskaźnik do tablicy `task[]`. Więcej na temat tej struktury napisane jest w rozdziale 2.1.1.

```
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;
```

Wskaźniki te wykorzystywane są przez tablicę z hashowaniem, w której funkcja hashujaca korzysta z pidów procesów.

Pamięć

```
struct mm_struct *mm;
```

Elementy struktury mm_struct opisują początek i koniec segmentów danych obecnie wykonywanego programu. Istnieją też zmienne umożliwiające korzystanie z pamięci wirtualnej. Więcej na ten temat napisane jest w rozdziale 3.6.

Pliki

```
struct fs_struct *fs;
```

W strukturze fs_struct znajdują się dane związane z systemem plików. Składa się ona z czterech składników: `count, umask, root, pwd`. Proces może wpływać na tryb dostępu do nowo utworzonych plików za pośrednictwem wywołania systemowego umask. Wartości ustawione przy użyciu umask są przechowywane w `umask`. Atrybut `pwd` określa aktualny katalog, a `root` wskazuje katalog główny używany przy wyznaczaniu bezwzględnych ścieżek dostępu. Zmienna `count` zarezerwowana jest dla dalszych rozszerzeń.

```
struct files_struct *files;
```

najważniejsze pola tej struktury to:

```
atomic_t count;
fd_set close_on_exec_init;
fd_set open_fds_init;
struct file * fd_array[NR_OPEN_DEFAULT];
struct file ** fd;
```

Zmienna `count` używana jest jako licznik odniesienia (określa liczbę otwartych plików). Proces otwierający plik otrzymuje deskryptor pliku, który jest niewielką liczbą całkowitą mniejszą od `max_fds`, który jest nie mniejszy od `NR_OPEN_DEFAULT` czyli 32. Używając deskryptora jako indeksu w tablicy `fd` uzyskujemy wskaźnik do strukrury `file` która opisuje plik. W jądrze 2.2 zmienna `fd` dla procesu `init` równa się `fd_array`. Jednak obecnie nie używa się praktycznie `fd_array` i wszystkie operacje wykonuje się na dynamicznej tablicy `fd`, która może osiągnąć rozmiary maksymalnie `NR_OPEN` czyli 1024*1024. Zmienne `close_on, exec_init` i `open_fds_init` przechowują maski bitowe odpowiednio wszystkich używanych deskryptorów plików, które mają być zamknięte gdy wywołana zostanie funkcja systemowa exec i wszystkich używanych deskryptorów plików. Typ `fd_set` umożliwia przechowywanie 1024 bitów.

Sygnały

```
int pdeath_signal;
```

Numer sygnału, który będzie wysłany do siebie gdy zakończy się proces ojca.

```
struct signal_struct *sig;
```

Wskaźnik do tablicy przechowującej informacje na temat sposobu obsługi sygnału. Dokładniej jest ona opisana w rozdziale 2.5.1 omawiającym sygnały.

```
sigset_t signal, blocked;
```

Zmienne `signal` i `blocked` zawierą maski bitowe sygnałów odpowiednio otrzymywanych przez proces oraz tych, które zamierza obsłużyć później, a które są obecnie zablokowane.

Czas

```
unsigned long start_time;
```

Moment, w którym proces został wygenerowany.

```
long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
```

Tablice te przechowują informacje ile czasu proces spędził odpowiednio w trybie użytkownika i w trybie systemowym na poszczególnych procesorach

Identyfikator procesu

`pid_t pid;`
Numer indentyfikacyjny procesu
`pid_t pgrp;`
Numer indentyfikacyjny grupy procesów
`pid_t session;`
Identyfikator sesji.
`int leader;`
Proces będący liderem sesji.
`int ngroups;`
Ilość grup do których należy proces.
`gid_t groups[NGROUPS];`
Tablica w której kolejnymi elementami są struktury opisujace grupy do których proces należy.

Różne

```
unsigned long personality;
int dumpable:1;
int did_exec:1;
```

Flaga dumpable wskazuje, czy obecny proces ma dokonać zrzucenia obszaru pamięci jeśli pojawią się określone sygnały. Niezbyt jasna składnia obowiązująca w standardzie POSIX wymaga aby podczas wywołania `setpgid` dokonano rozróżnienia czy proces wykonuje ciągle oryginalny program czy też załadował nowy program przy pomocy wywołania systemowego `execve`. Informacje te można monitorować przy pomocy flagi `did_exec`.

```
struct linux_binfmt *binfmt;
```

Ta zmienna umożliwia uruchamianie pod Linuksem prawie każdego programu przez napisanie jego nazwy. Obejmuje to na przykład programy napisane i skompilowane w językach: Java(TM) i Python. By to osiągnąć należy podać jaki interpreter będzie użyty. Struktura `linux_binfmt` zawiera wskaźniki do funkcji używanych do ładowania programu i bibliotek dzielonych oraz „zrzucania” obszaru pamięci.

```
int exit_code, exit_signal;
```

Kod zakończenia procesu i sygnał wyjścia. Zmienne te mogą być użyteczne dla procesu „ojca”.

```
struct wait_queue *wait_chldexit;
struct semaphore *vfork_sem;
```

Atrybuty te wykorzystywane są przy wywołaniach systemowych odpowiednio `wait4` i `vfork` jako kolejki procesów uśpionych do których proces wstawia sam siebie.

```
struct sem_undo *semundo;
```

Gdy proces zostanie przerwany powinien pozwalniać zajmowane przez siebie semafory. Potrzebne do tego informacje przechowywane są w zmiennej semundo.

```
struct sem_queue *semsleeping;
```

Atrybut ten wykorzystywany jest w momencie gdy proces jest „zatrzymywany" przy przechodzeniu przez semafor jako kolejka procesów uśpionych, do których proces wstawia sam siebie.

```
long need_resched;
int processor;
int last_processor;
unsigned long policy, rt_priority;
```

Zmienne te zostały opisane w rozdziale 2.3.1.

Uwagi do poprzednich wersji jądra: Dawniej były dostępne zmienne `errno` i `debugreg[8]`. Przechowywały one odpowiednio: kod błędu kopiowany w momencie wychodzenia z wywołania systemowego na zmienną globalną errno; natomiast debugreg[8] przechowywała rejestry z informacjami do usuwania błędów. Poprzednio były wykorzystywane tylko w wywołaniu systemowym ptrace.

2.1.3 Lista wszystkich procesów

Lista wszystkich procesów jak sama nazwa wskazuje zawiera wszystkie procesy działające w systemie. Jest to cykliczna lista dwukierunkowa zaimplementowana za pomocą następujących elementów struktury `task_struct`:

`struct task_struct * prev_task` - poprzedni element
`struct task_struct * next_task` - następny element

Początkowo lista zawiera jeden proces — *idle*[1]. Z listą związana jest zmienna globalna `nr_tasks`, która zawiera liczbę działających procesów (nie licząc procesu *idle*).

Do uproszczenia opracji na liście wszystkich procesów w pliku w pliku `include/linux/sched.h` zdefiniowano następujące makra:

- `SET_LINKS(p)` dołączenie procesu `p` do listy wszystkich procesów (poprawienie wskaźników tak by lista zawierała `p`)

- `REMOVE_LINKS(p)` usunięcie `p` z listy wszystkich procesów (tak jak w poprzednim przypadku, chodzi jedynie o zmianę wskaźników, tak by nie wskazywały na `p`, a lista dalej tworzyła spójna całość)

- `for_each_task(p)` wykonanie na wszystkich procesach (wyłączając proces *idle*) następującego po tym makrze fragmentu kodu, zmienna `p` oznacza zmienną lokalną pętli

Pierwsze dwa makra są wykorzystywane bardzo sporadycznie (głównie do tworzenia/usuwania procesów), natomiast makro `for_each_task(p)` jest bardzo często używane w źródłach jądra.

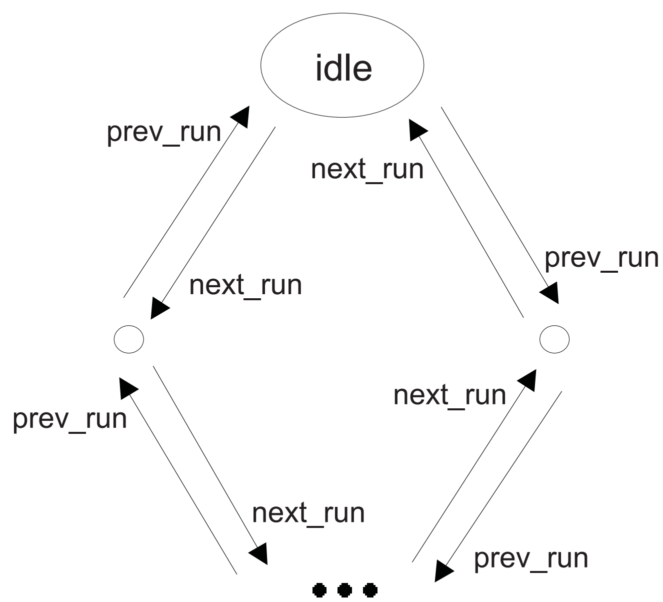
[1] *idle*

2.1.4 Kolejka procesów gotowych

Kolejka procesów gotowych jest listą wszystkich procesów ubiegających się o procesor. Jest to cykliczna lista dwukierunkowa połączona za pomocą następujących elementów struktury `task_struct`:

`struct task_struct * prev_run` - poprzedni element
`struct task_struct * next_run` - następny element

Początkowo znajduje się na niej jedynie proces *idle* [2]. Program szeregujący wyszukuje proces, któremu zostanie przydzielony procesor przeglądając tę właśnie listę. Z kolejką procesów gotowych związana jest jest zmienna globalna `nr_running`, której wartością jest liczba procesów ubiegających się o przydział procesora (nie licząc procesu *idle*).



Rysunek 2.1: Kolejka procesów gotowych

Na kolejce procesów gotowych można wykonywać następujące operacje (zdefiniowane w pliku *kernel/sched.c*):

- `void add_to_runqueue (struct task_struct * p)` - dodanie do kolejki nowego elementu oraz zwiększenie liczby procesów gotowych (`nr_running`).

  UWAGA: Nowy proces wstawiany jest na początek kolejki procesów gotowych - tzn. stoi bezpośrednio za procesem *idle*. Umożliwia to przyjmowanie przerwań w czasie, gdy funkcja `schedule()` wybiera proces, który zacznie wykonywać się jako następny. Wstawienie nowych procesów na początek listy nie zmieni struktury przeglądanego fragmentu kolejki procesów gotowych, a zatem nie doprowadzi do sytuacji błędnej.

[2] *idle* to proces wskazywany przez zmienną globalną `init_task`, jego identyfikator `pid=0`

- `void del_from_runqueue (struct task_struct * p)` - usunięcie procesu z kolejki oraz zmniejszenie liczby procesów gotowych (`nr_running`).
- `void move_last_runqueue (struct task_struct * p)` - przesunięcie procesu na koniec kolejki - tzn. bezpośrednio przed proces *idle*.
- `void move_first_runqueue (struct task_struct * p)` - przesunięcie procesu na początek kolejki - tzn. bezpośrednio za proces *idle*.

W systemie istnieje dokładnie jedna kolejka procesów gotowych, zatem wszystkie procesy ubiegające się o procesor przechowywane są w jednej strukturze danych. Kolejki koncepcyjne dla procesów czasu rzeczywistego realizowane są przez nadanie priorytetom tych procesów odpowiednio wysokich wartości - wynoszą one `rt_priority + 1000` . Jest to wielkość nieosiągalna dla zwykłych procesów (patrz część o szeregowaniu procesów).

2.1.5 Powiązania rodzinne procesu

W Linuksie wszystkie procesy należą do jednego drzewa, którego korzeniem jest proces `init` (ma on `pid` równy 1). Osierocone procesy (tzn. takie, których ojciec zakończył się przed nimi) przejmuje proces `init` i w ten sposób każdy proces ma swojego ojca. Jedynym wyjątkiem jest `init_task` (ma on `pid` równy 0) - zawartość odpowiadającej mu struktury `task_struct` jest wprost zapisana w pliku `include/linux/sched.h` - ma on powiązania tylko do samego siebie.

Drzewo procesów można wyświetlić (w trybie tekstowym) przy pomocy polecenia `pstree`.

Oto wynik jego przykładowego działania:

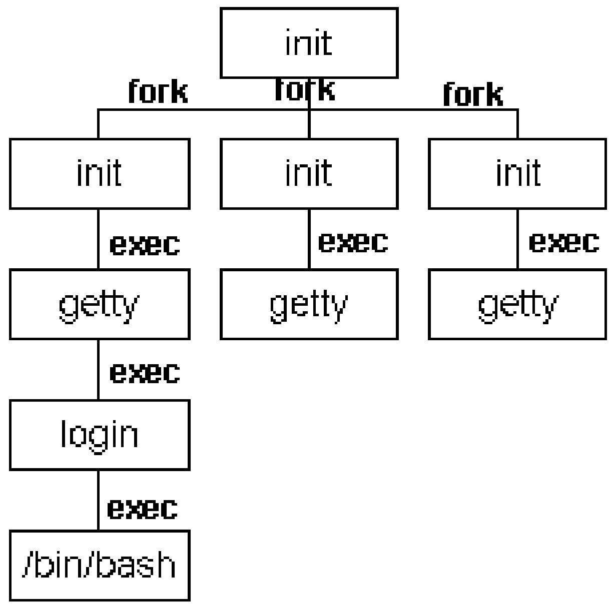
```
init-+-atd
     |-crond
     |-gpm
     |-inetd
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-login---bash---pstree
     |-lpd
     |-5*[mingetty]
     |-sendmail
     |-syslogd
     `-update
```

Polecenie `pstree` ma kilka pouczających opcji, np. -u wyświetla w nawiasie nazwę użytkownika (`uid`) procesu, jeżeli nastąpiła jego zmiana. `-p` wyświetla także identyfikator (`pid)` procesu. Polecam przeczytanie `man pstree.`

Drzewo procesów zaraz po zalogowaniu się jednego użytkownika może wyglądać np. tak jak na rysunku 2.2 - *Drzewo procesów getty*([Stevens 95]).

Powiązania rodzinne procesu są reprezentowane przez wskaźniki - pola struktur `task_struct`:

- `struct task_struct* p_opptr` - wskaźnik do oryginalnego ojca



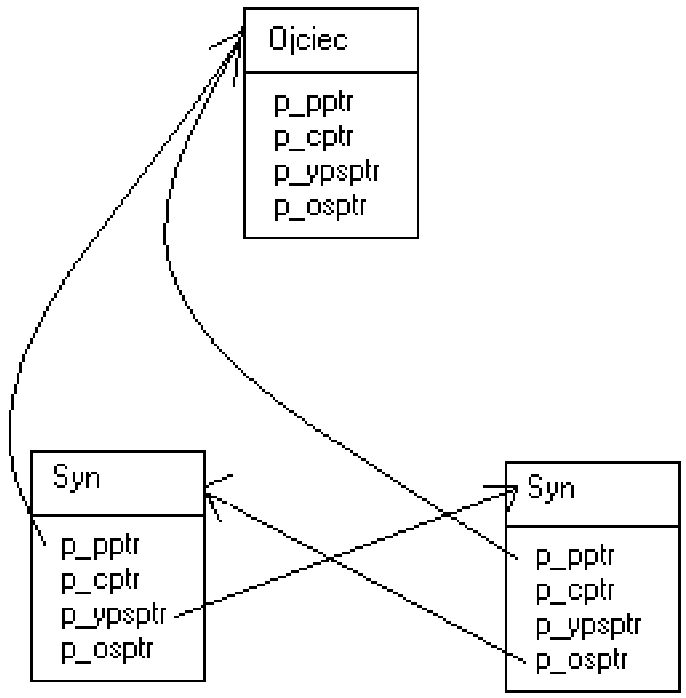
Rysunek 2.2: Drzewo procesów getty

- `struct task_struct* p_pptr` - wskaźnik do ojca
- `struct task_struct* p_cptr` - wskaźnik do najmłodszego syna
- `struct task_struct* p_ysptr` - wskaźnik do młodszego brata
- `struct task_struct* p_osptr` - wskaźnik do starszego brata

Na rysunku 2.3 (*Przykładowe drzewo procesów*) widać te wskaźniki dla drzewa procesów zawierającego ojca i dwóch synów. Gdy jakieś pole (na tym rysunku) na nic nie wskazuje, oznacza to, że jego wartością jest `NULL` albo wskazuje na obiekt spoza tego drzewa.

Każdy proces w Linuksie należy do pewnej grupy procesów. W strukturze `task_struct` procesu jest pamiętany jej identyfikator - `pgrp`. Jest to identyfikator (`pid`) procesu - założyciela grupy. Podział na grupy jest pomocny na przykład przy wysyłaniu sygnałów: `kill -n` (gdzie n - liczba naturalna) wysyła sygnał do procesów, których `pgrp` jest równy n. Proces może założyć nową grupę przez **setpgrp**. Wówczas jej identyfikatorem staje się jego `pid`. Przyłączyć się do już istniejącej grupy proces może tylko jako potomek procesu - założyciela grupy.

Oprócz tego w strukturze `task_struct` są jeszcze takie atrybuty realizujące powiązania rodzinne procesu, jak:



Rysunek 2.3: Przykładowe drzewo procesów

- `pid_t session` - `pid` procesu - lidera sesji
- `int leader` - czy proces jest liderem grupy terminalowej

2.2 Stany procesu, diagram przejść międzystanowych

W strukturze `task_struct` znajduje się pole `volatile long state`.

Pole to określa stan, w jakim aktualnie znajduje się dany proces. Proces może, reagując na pewne zdarzenia i okoliczności, zmienić swój stan. Pole to może przyjąć jedną z następujących wartości:

- `TASK_RUNNING` (0) - proces jest procesem, który aktualnie wykorzystuje procesor lub czeka na jego przydzielenie (jest na liście procesów gotowych do wykonywania).
- `TASK_INTERRUPTIBLE` (1) - proces czeka na określone zdarzenie. Różnica pomiędzy tym stanem, a `TASK_UNINTERRUPTIBLE` jest taka, że tutaj proces obsługuje nadchodzące sygnały. Proces w tym stanie jest po prostu uśpiony i po nadejściu sygnału lub zajściu zdarzenia, na które czekał, umieszczany jest w kolejce procesów gotowych do wykonania.

- `TASK_UNINTERRUPTIBLE` (2) - podobnie jak `TASK_INTERRUPTIBLE`, ale w tym stanie proces nie obsługuje nadchodzących sygnałów. Proces znajduje się w tym stanie zwykle w oczekiwaniu na spełnienie jakiegoś warunku związanego ze sprzętem.

- `TASK_STOPPED` (4) - proces został wstrzymany po otrzymaniu odpowiedniego sygnału (`SIGSTOP`, `SIGSTP`, `SIGTTIN`, `SIGTTOU`) lub będąc monitorowanym przez inny proces przy użyciu wywołania systemowego `ptrace` przekazał sterowanie do procesu monitorującego. Opuszczenie tego stanu możliwe jest jedynie po otrzymaniu sygnału `SIGCONT`.

- `TASK_ZOMBIE` (8) - proces został przerwany (skończył swój komputerowy żywot), ale jego struktura znajduje się jeszcze w tabeli procesów.

- `TASK_SWAPPING` (16) - stała ta jeszcze nie jest używana w wersji systemu 2.2. Żaden proces nie znajduje się nigdy w tym stanie.

O procesie, który znajduje się aktualnie w stanie `TASK_RUNNABLE` i jest tym, który ma aktualnie przydzielony procesor, mówimy, że się właśnie wykonuje. Proces taki może wykonywać się w jednym z dwóch następujących trybów:

- Tryb użytkownika - tryb podstawowy procesu.

- Tryb jądra - w tym trybie możliwe jest wywoływanie funkcji systemowych.

Rysunek 2.4 przedstawia diagram możliwych przejść procesu (nad strzałkami podane są funkcje i zdarzenia, pod wpływem których proces zmienia stan):

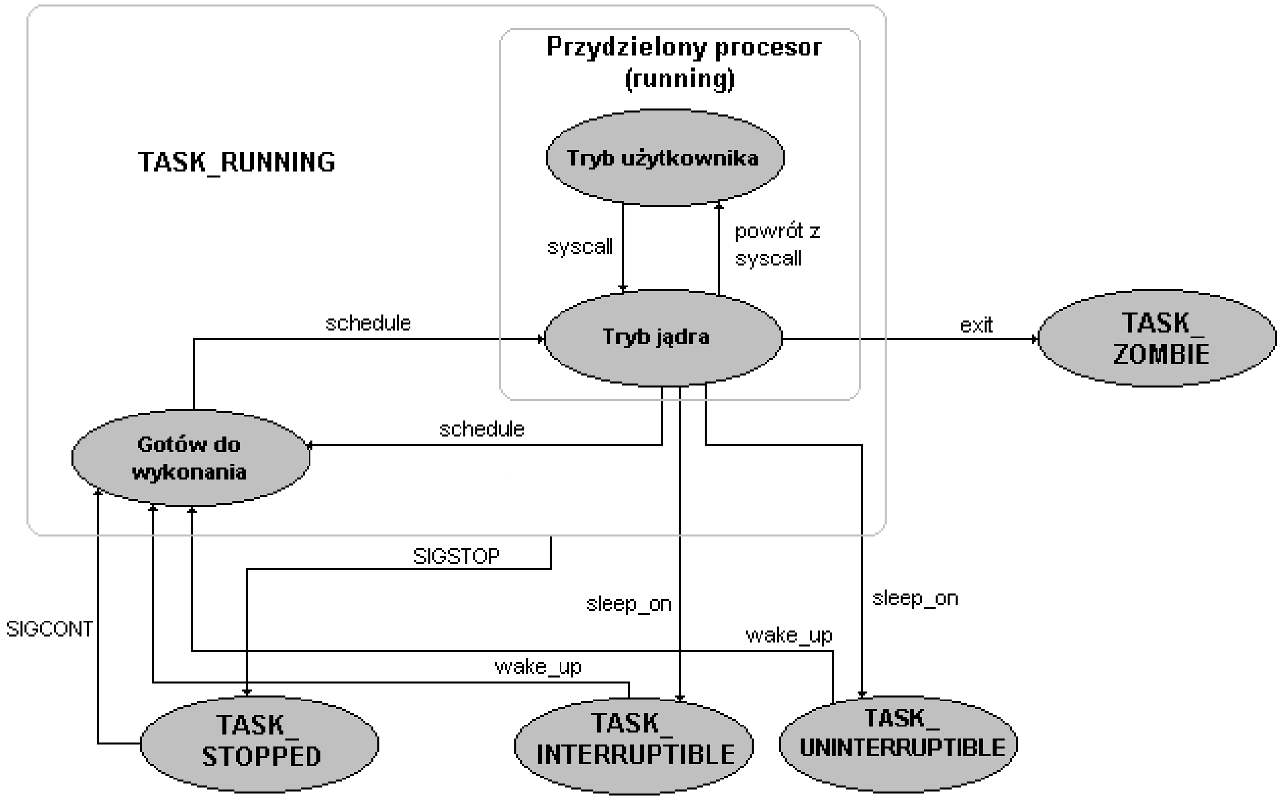
2.3 Szeregowanie procesów

Wiadomo iż użytkownik doznaje wrażenia, że kilka zadań jest wykonywanych jednocześnie, dzięki dzieleniu zasobu w postaci czasu procesora, przez poszczególne procesy. Każdy z nich działa na procesorze w krótkich przedziałach czasowych mierzonych w tyknięciach zegara systemowego. Wiaże się z tym konieczność częstych podmian aktualnie wykonywanego procesu (tzw. zmiany kontekstu). O tym kiedy, jak długo i który proces ma zostać wybrany jako następny do wykonywania na procesorze decydują funkcje opisane w tym podrozdziale.

Algorytm zgodnie z którym procesor jest przydzielany poszczególnym procesom jest zaimplementowany w pliku `kernel/sched.c`.

Szereg funkcji i procedur służących do ustalania kolejności w jakiej procesy będą korzystały z procesora jest nazywany „schedulerem”. Zaś główną funkcją odpowiedzialną za te właśnie zadania jest funkcja `schedule()`.

Linux daje możliwość szeregowania procesów zgodnie z trzema następującymi metodami: **SCHED-_FIFO, SCHED_RR, SCHED_OTHER**. Informacje o dwóch pierwszych z tych trybów szeregowania zawarte są w rozdziale 2.3.2. Dla procesów zwykłych używana jest strategia **SCHED-_OTHER**.



Rysunek 2.4: Diagram przejść międzystanowych

2.3.1 Pola w task_struct związane z szeregowaniem

- `volatile long state;`

  stan procesu, algorytm szeregowania wybiera jeden z gotowych procesów (w stanie **TASK_RUNNING**).

- `struct task_struct *next_run, *prev_run;`

  wskaźniki na następny i poprzedni proces gotowy.

- `unsigned long policy;`

  tryb szeregowania procesu (**SCHED_FIFO**, **SCHED_RR**, **SCHED_OTHER**)

- `long priority;`

  tzw. priorytet statyczny, który jest podstawą do obliczenia kwantu czasu na jaki procesowi przyznawany będzie procesor.

- `long counter;`

pozostała część bieżącego kwantu czasu (ile tyknięć zegara pozostało do wywłaszczenia).

- `rt_priority;`

  numer kolejki koncepcyjnej do jakiej proces należy (0 dla trybu **SCHED_OTHER**, 1..99 dla pozostałych trybów szeregowania)

- `long need_resched;`

  ustawienie tej flagi oznacza, że należy wykonać algorytm szeregowania (funkcja `schedule()`). Jest ustawiana, gdy bezpośrednie wywołanie tego algorytmu mogłoby trwać za długo - np. w procedurze obsługi przerwania zegarowego.

- `int processor;`

  procesor na jakim proces się wykonuje lub **NO_PROC_ID**, jeśli się nie wykonuje.

- `int last_processor;`

  procesor na jakim proces się ostatnio wykonywał lub **NO_PROC_ID**, jeśli się jeszcze nie wykonywał. Scheduler wykorzystuje tę informację przy wyborze zadania dla wolnego procesora. Faworyzowany jest procesor na którym zadanie było ostatnio wykonywane, gdyż w jego pamięci podręcznej mogą jeszcze znajdować się dane procesu, co przyspieszy jego wykonywanie.

2.3.2 Klasy priorytetowe procesów

Procesy dzielimy na trzy klasy:

- Procesy czasu rzeczywistego
- Procesy zwykłe
- Proces `idle`

Procesy czasu rzeczywistego (ang. *RealTime*) wymagają dostępu do procesora w ściśle określonym przedziale czasowym - są to na przykład procesy zarządzające mechanizmami na taśmie produkcyjnej, gdzie wszelkie opóźnienia są niedopuszczalne. System Linux nie jest specjalizowany jako system czasu rzeczywistego, jednak stworzony został na tyle dobrze, iż radzi sobie z tym niewiele gorzej od systemów specjalizowanych. Oczywiste jest, iż procesy czasu rzeczywistego muszą mieć wyższy priorytet od pozostałych, jeżeli chodzi o przydział procesora. W praktyce jeśli podczas działania procesu zwykłego pojawi się gotowy proces czasu rzeczywistego, to działający proces zostaje natychmiast wywłaszczony, a procesor przydzielony procesowi czasu rzeczywistego. Istnieje tu pewne niebezpieczeństwo - po wywołaniu pętlącego się procesu czasu rzeczywistego system sprawiał będzie wrażenie, jakby się „zawiesił" - procesor wciąż przydzielany będzie procesowi czasu rzeczywistego. Wstrzymana zostanie w takim wypadku praca wszystkich procesów zwykłych - w tym systemu okien, interpretatora poleceń itp. W celu ochrony systemu przed podobnymi wypadkami procesy czasu rzeczywistego uruchamiać może jedynie root.

Proces Idle ma z kolei najniższy priorytet w systemie. Jest procesem szczególnym - jako jedyny tworzony jest na samym początku bez udziału funkcji `fork()`, nie odpowiada mu także żaden plik wykonywalny. Dokonuje on inicjacji wielu systemowych struktur danych, a następnie tworzy proces init za pomocą funkcji `clone()`. Na tym jednak jego rola się nie kończy - proces ten nigdy nie ginie i wykonuje nieskończoną pętlę. Nie wpływa to na obniżenie wydajności systemu, gdyż proces ten, jako posiadający najniższy priorytet w systemie, zostanie wybrany wtedy i tylko wtedy, gdy nie ma innych procesów gotowych. Monitorowanie stopnia obciążenia procesora odbywa się właśnie przy pomocy tego procesu - wystarczy bowiem sprawdzić jak często jest on wykonywany.

2.3.3 Szeregowanie procesów rzeczywistych

Wstęp

W jądrze Linuksa możemy wyróżnić dwa typy procesów: procesy zwykłe (nie omawiane w niniejszej pracy) oraz procesy czasu rzeczywistego. Podstawową rożnicą między nimi jest to że procesy czasu rzeczywistego muszą mieć dostępny procesor w ściśle określonym przedziale czasowym. Znajdują one zastosowanie w systemach, gdzie czas reakcji na określone zdarzenia musi być szybki i precyzyjny, na przykład systemy podtrzymujące życie pacjenta w szpitalu lub program obsługujący chłodzenie reaktora atomowego.

Pierwsze implementacje procesów czasu rzeczywistego zostały dołączone do jądra Linuksa w wersji 1.3.8. Jednak standartowe dystrybucje omawianego systemu nie posiadają w pełni profesjonalnej ich obsługi. Między innymi dlatego że przyjęto 5ms za standard rozpoczęcia wykonywania procesu czasu rzeczywistego. W klasycznej dystrybucji systemu czas reakcji na zdarzenie może wynosić do 10ms, co jest niewystarczające. Dla w pełni profesjonalnej obsługi procesów czasu rzeczywistego istnieją specjalne dystrybucje jądra.

Jak szeregowane są procesy czasu rzeczywistego

Ze wzgledów bezpieczeństwa procesy czasu rzeczywistego mogą być tworzone tylko przez administratora systemu. Wynika to z faktu, że algorytm wybierający proces do wykonania wybierze zawsze proces czasu rzeczywistego przed procesami zwykłymi. W przypadku stworzenia procesu pętlącego się lub wymagającego dużo czasu procesora system może sprawiać wrażenie, że zawiesił.

Procesy czasu rzeczywistego pogrupowane są w `99` kolejkach koncepcyjnych o numerach od `1` do `99`. Im wyższy numer kolejki tym proces szybciej dostanie procesor.

Przykład

W systemie są dwa procesy czasu rzeczywistego: pierwszy proces w kolejce o numerze 33 i drugi w kolejce 24. Algorytm szeregujący procesy będzie wybierał proces z kolejki o numerze 33 dopóki ten nie zakończy się lub zostanie uśpiony. Dopiero potem zostanie wybrany proces z kolejki o numerze 24.
Wyrożniamy dwa rodzaje trybów szeregowania procesów czasu rzeczywistego:

- `SCHED_RR:` Jest to metoda bardzo zbliżona do metody `SCHED_OTHER`, opisanej w następnym rozdziale dla procesów zwykłych. Proces dostaje procesor na kwant czasu zapisany w jego `task_struct` w polu `counter`. Po upłynięciu tego kwantu czasu procesor zostaje mu wywłaszczony i uruchomiony algorytm szeregujący procesy.

- SCHED_FIFO: Proces dostaje procesor do chwili gdy sam go nie zwolni lub pojawi się proces z kolejki koncepcyjnej o wyższym numerze. W chwili pojawienia się procesu bardziej uprzywilejowanego procesowi zostaje odebrany procesor i zostaje umieszczony na początku kolejki procesów gotowych, tak by w chwili gdy procesy z kolejek o wyższych numerach skończą domagać się procesora został wybrany proces, który był poprzednio wykonywany.

Proces czasu rzeczywistego może oddać procesor w trzech sytuacjach:

- wywoła funkcję `exit()`
- pojawi się proces z kolejki koncepcyjnej o wyższym numerze
- proces wywoła funkcję `sched_yield()`

Funkcja `sched_yield ()` powoduje, że procesowi zostaje wywłaszczony procesor i zostaje umieszczony na koncu kolejki procesów gotowych do wykonania, żeby algorytm szeregujacy mógł wybrać inny proces o tym samym priorytecie dynamicznym, czyli wartości zwróconej przez funkcję `goodness()`.

Przykład

W systemie są dwa procesy czasu rzeczywistego A i B w tych samych kolejkach koncepcyjnych. Procesem wykonywanym jest proces A i wywołuje funkcję `sched_yield()` wtedy proces A zostaje odłożony na końcu kolejki procesów gotowych do wykonania i algorytm szeregujący wybiera jako następny do wykonania proces B.

Do tworzenia procesów czasu rzeczywistego służy funkcja `sched_setscheduler()` zdefiniowana `sched.h`. Ma ona następującą postać:

```
sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)
```

Gdzie `pid` oznacza pid procesu, na którym ma zostać wykonana operacja. W przypadku podania `pid==0` zostanie zmieniony sposób szeregowania procesu, który wywołał tę funkcję. Zmienna `policy` oznanacza tryb szeregowania procesu dla procesów czasu rzeczywistego może ona przyjmować wartości `SCHED_RR` i `SCHED_FIFO`. Struktura `sched_param` składa się z pola `int sched_priority`, które oznacza numer kolejki koncepcyjnej do której ma należeć proces, czyli może przyjmować wartości od 0 do 99.

Przykład

Jeżeli mielibyśmy stworzyć proces czasu rzeczywistego należący do drugiej kolejki koncepcyjnej i mający tryb szeregowania `SCHED_FIFO` to wyglądałoby to mniej więcej tak:

```
param.sched_priority=2;
sched_setscheduler(0,SCHED_FIFO,*param);
```

Jeżeli wykonanie funkcji się nie powiedzie to zwróci ona na wyniku -1 i zapisze następujące wartości na zmiennej errno:

- `ESRCH` - nie istnieje proces o takim pidzie
- `EPERM` - użytkownik nie ma uprawnień
- `EINVAL` - podane wartości nie mają sensu

Krótki opis implementacji

Bardziej szczegółową informację na ten temat będzie można znaleźć w rozdziale o funkcji `schedule()` i `goodness()`.

Za szeregowanie wszystkich procesów (tych zwykłych i czasu rzeczywistego) odpowiada funkcja `schedule()` zaimplementowana w pliku `sched.c`. Przeszukuje ona listę procesów gotowych, zaimplementowaną jako dwukierunkową listę struktur `task_struct`. Dla każdego elemetu listy `schedule()` wywołuje funkcję `goodness()`, która z kolei zwraca dynamiczny priorytet procesu. Im wiekszą wartość zwróci `goodness()` dla procesu tym szybciej dostanie on procesor, w szczególności przed tym dla którego funkcja zwróciła mniejszą wartość. Funkcja `goodness()` stosuje następujący algorytm wartościowania: dla procesu zwykłego zwraca wartość pola `counter` czyli ilość kwantów czasu na jaki proces ma dostać procesor zwiększoną o wartość pola `priority`, natomiast dla procesu czasu rzeczywistego wartość pola `rt_priority` zwiększonego o 1000. Należy teraz pokazać, że dla procesu czasu rzeczywistego funkcja `goodness()` zwróci zawsze większą wartość niż dla procesu zwykłego.

Wartość pola `priority` w strukturze `task_struct` nie może przekroczyć wartości `2*DEF_PRIORITY`, gdzie `DEF_PRIORITY` jest stałą, której wartość wynosi standardowo 20. Algorytm szeregujący przydziela nowe priorytety dynamiczne zgodne z zasadą: `counter = counter / 2 + priority`, po rozwinięciu tej fomuły w szereg geometryczny otrzymamy, że `counter <= 2 * priority`, a stąd `counter <= 80` (wartość całkowita jest nie większa od 100) co jest znacznie mniejsze od `rt_priority +1000`.

Funkcja `schedule()` sprawdza czy procesowi któremu skończył się kwant czasu, a był przed chwilą wykonywany, nie jest procesem szeregowanym `SCHED_RR` jeśli tak to otrzymuje automatycznie nowy kwant czasu nie czekając aż wszystkie procesy bedą miały wartość pola `counter` równą zeru.

2.3.4 Szeregowanie procesów zwykłych

Procesy zwykłe znajdują się w kolejce koncepcyjnej o numerze 0. Skoro więc procesy te nie są rozróżniane ze względu na kolejki, kolejność ich wyboru jest uzależniona głównie od wartości atrybutu `counter`.

Najważniejsze funkcje odpowiedzialne na szeregowanie to `schedule()` i `goodness()`.

Funkcja schedule()

Jest to główna funkcja szeregująca. Wybiera ona proces, któremu zostanie przydzielony procesor. Oto schemat jej działania:

- Gdy proces bieżący jest szeregowany w trybie **SCHED_RR** i skończył mu się kwant czasu (counter = 0), to przyznaj mu nowy kwant i przesuń na koniec jego kolejki koncepcyjnej.

- Jeśli bieżący proces jest w stanie **TASK_INTERRUPTABLE** i nadszedł dla niego sygnał, którego nie blokuje, to zmień jego stan na **TASK_RUNNING**, w przeciwnym przypadku usuń go z kolejki procesów gotowych.
- Przejdź po kolejce procesów gotowych i wybierz pierwszy z procesów dla których funkcja `goodness()` zwróciła maksymalną wartość.
- Gdy wszystkie procesy gotowe wykorzystały swoje kwanty czasu i nie ma procesów czasu rzeczywistego wykonaj dynamiczne przeliczanie priorytetów
- Przełącz kontekst na wybrany proces (`switch_to`), o ile wybrany został proces różny od bieżącego.

Funkcja goodness()

Funkcja goodness ocenia na ile dany proces jest odpowiednim do przydzielenia mu czasu procesora. Im jej wartość jest większa dla danego procesu, tym lepiej byłoby przydzielić mu procesor. Działa ona w następujący sposób:

- dla procesów czasu rzeczywistego zwracana jest wartość 1000 + rt_priority
- dla procesów zwykłych wartość pola counter +
+ **PROC_CHANGE_PENALTY**, jeśli proces wykonywał się ostatnio na tym samym procesorze (jeśli mamy w systemie więcej niż 1 procesor)
+ 1, jeśli jest on bieżącym procesem

Dynamiczne przeliczanie priorytetów

- **Priorytet statyczny**

  Priorytet statyczny procesu zapisany jest w strukturze `task_struct` w polu `priority`. Jest to liczba z przedziału -20..20 (-20 to najwyższy). Jest on zmieniany np. przez funkcję nice(). Używa się go do wyliczania priorytetu dynamicznego procesu (przydzielania kwantu czasu).

- **Priorytet dynamiczny**

  Priorytet dynamiczny procesu zapisany jest w strukturze `task_struct` w polu `counter`. Służy on do wybierania procesu do wykonania. Jeśli proces jest procesem aktualnie wykonywanym, `counter` jest zmniejszany wraz z każdym tyknięciem zegara (jeśli jest większy od 0). Jeśli `counter` = 0 , oznacza to, że proces wyczerpał czas przydzielony mu przez jądro i, jeśli jeszcze się nie zakończył, jest wstawiany do kolejki procesów gotowych i wywoływana jest funkcja schedule().

- **Przeliczanie**

  Jeśli w systemie nie ma żadnych gotowych procesów czasu rzeczywistego, a wszystkie procesy zwykłe wykorzystały swoje kwanty czasu tzn. dla każdego procesu `counter` = 0, jądro wykonuje przeliczenie dynamicznych priorytetów wszystkich procesów w systemie. Procesom gotowym zwyczajnie przydzielany jest następny kwant czasu tzn.

  counter := **priority**,

natomiast procesom śpiącym priorytet ten jest dodatkowo podwyższany (tzw. postarzanie procesów):

$$\mathbf{counter} := \frac{\mathbf{counter}}{\mathbf{2}} + \mathbf{priority}.$$

Powoduje to, że po obudzeniu procesy te będą faworyzowane w przydzielaniu im czasu procesora.

Cechy algorytmu szeregowania

Procesy czasu rzeczywistego są obsługiwane wcześniej niż procesy zwykłe, w związku z czym procesy zwykle nie spowodują zagłodzenia.

- **Wady algorytmu szeregującego procesy zwykłe:**
  - jeśli w systemie działają procesy długie i wyczerpią one swój kwant czasu to muszą czekać na wyczerpanie się kwantów czasu innych procesów, co powoduje urywaną pracę procesów (jest to niekorzystne dla niektórych klas procesów np. dla procesów interakcyjnych)
  - Nie rozróżnia procesów nastawionych na obliczenia lub interakcję. Jeśli działają 2 procesy: A interakcyjny i B obliczeniowy, to A nie wykorzystuje całego przydzielonego sobie czasu i usypia, B dostaje procesor i trzyma go cały swój kwant. Dlatego obciążenie systemu procesami obliczeniowymi znacznie utrudnia pracę z programami interakcyjnymi.
- **Zalety:**
  - algorytm jest sprawny, nie powoduje zagłodzenia procesów oraz wyklucza możliwość blokady, a także jest łatwy do zrealizowania,
  - kosztowne przeliczanie priorytetów nie odbywa się zbyt często,
  - algorytm jest sprawiedliwy i nie faworyzuje żadnego procesu w obrębie grupy procesów o tym samym priorytecie.

2.4 Synchronizacja procesów na poziomie jądra

2.4.1 Usypianie i budzenie procesów

Często zdarza się, że działający proces musi zostać wstrzymany i może być kontynuowany dopiero po zajściu konkretnego zdarzenia. Takim zdarzeniem może być zwolnienie się zasobu, podniesienie semafora, pojawienie się informacji w łączu lub zakończenie pracy procesu potomnego. W Linuksie procesy oczekujące na konkretne zdarzenie umieszczane są w osobnych kolejkach. Elementy tych kolejek mają następującą strukturę, zdefiniowaną w pliku `wait.h`:

```
struct wait_queue
{
    struct task_struct *task;
    struct wait_queue *next;
};
```

W jądrze Linuksa (pliki: `sched.h` i `sched.c`) zdefiniowane są następujące funkcje:

- `void interruptible_sleep_on(struct wait_queue **p)`
- `long interruptible_sleep_on_timeout(struct wait_queue **p, long timeout)`
- `void sleep_on(struct wait_queue **p)`
- `long sleep_on_timeout(struct wait_queue **p, long timeout)`
- `void wake_up(struct wait_queue **q)`
- `void wake_up_interruptible(struct wait_queue **q)`

Poniżej przedstawione są dwa szkice algorytmów. Pierwszy opisuje sposób działania pierwszych czterech funkcji, służących do usypiania procesów, a drugi — dwóch pozostałych.

Algorytm 1 Funkcje `sleep_on`

Dane wejściowe: `p` - wskaźnik do kolejki procesów oczekujących, w której umieszczony zostanie dany proces

ustaw stan procesu na uśpiony (`TASK_INTERRUPTIBLE` względnie `TASK_UNINTERRUPTIBLE`)
wstaw proces do kolejki `p`
wywołaj szeregowanie (`schedule` względnie `schedule_timeout`)
usuń proces z kolejki `p`

Algorytm 2 Funkcje `wake_up`

Dane wejściowe: `q` - wskaźnik do kolejki procesów oczekujących na dane zdarzenie

dopóki są nieprzetworzone procesy `p` w kolejce `q` (względnie rozpatrujemy tylko te będące w stanie `TASK_INTERRUPTIBLE`) **wykonuj**
  zmień stan procesu `p` na `TASK_RUNNING`
  uaktualnij kolejkę procesów gotowych
koniec dopóki

Uwagi:

1. Funkcja `wake_up` nie usuwa procesu z kolejki procesów oczekujących. Dzieje się to w odpowiedniej funkcji `sleep_on` w momencie gdy proces ponownie dostaje sterowanie po wykonaniu `schedule`.
2. Funkcja `schedule_timeout` i inne z nią związane zostały zaimplementowane w jądrze Linuksa w listopadzie 1998.

2.4.2 Implementacja semaforów systemowych

W jądrze Linuxa zaimplementowane są semafory dostępne dla procedur systemowych (mogą być wołane tylko z poziomu jądra). Nie mają one nic wspólnego z semaforami IPC.

Struktura reprezentująca atrybuty semafora systemowego:

```
struct semaphore {
atomic_t count;
int waking;
struct wait_queue * wait;
};
```

1. `count` jest to po prostu licznik typu int, na którym wszystkie niezbędne operacje (dodawanie, odejmowanie, sprawdzanie wartości oraz ustawianie wartości) są zdefiniowane jako operacje atomowe (patrz plik atomic.h); jeżeli `count` jest ujemne, to moduł z `count` mówi o ilości procesów wiszących na semaforze, a dodatnia wartośc `count` jest po prostu wartością semafora.

2. `waking` jest zmienną pomocniczą, która mówi ile procesów w danej chwili przejdzie pod semaforem; wartość tej zmiennej jest zawsze dodatnia, i powinna zawsze być inicjowana na 0.

3. `wait` jest kolejką procesów wiszących na semaforze.

Funkcje i makra obsługujące operacje na semaforach

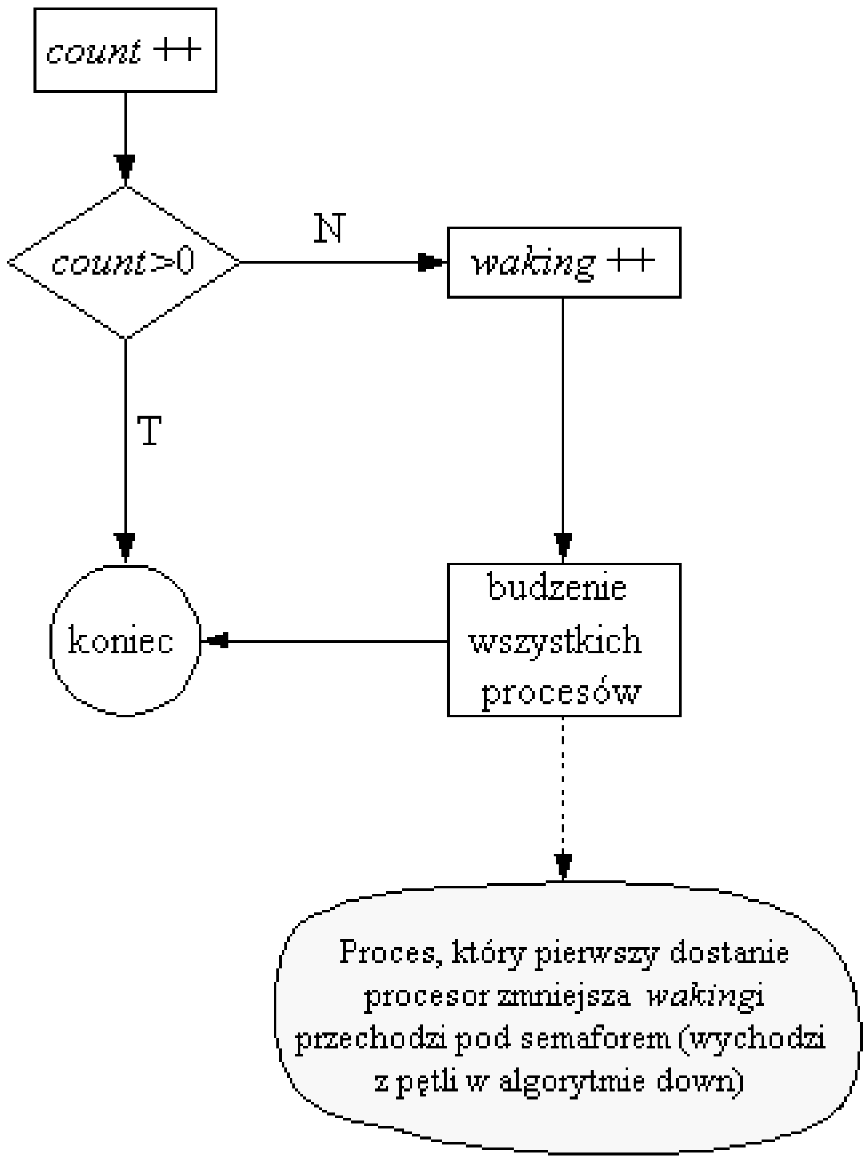
Tablica 2.1: Funkcje i makra obsługujące operacje na semaforach

Operacja	semaphore.h	semaphore.S	sched.c
up	void up(struct semaphore * sem)	ENTRY(__up_wakeup)	void __up(struct semaphore * sem)
down	void down(struct semaphore * sem)	ENTRY(__down_failed)	void __down(struct semaphore * sem)
down_interruptible	int down_interruptible(struct semaphore * sem)	ENTRY(__down_failed_interruptible)	int __down_interruptible(struct semaphore * sem)
down_trylock	int down_trylock(struct semaphore * sem)	ENTRY(__down_failed_trylock)	int __down_trylock(struct semaphore * sem)

Operacje na semaforze powodują wywołanie wyżej wymienionych funkcji (makr) w następującej kolejności pokazanej na rysunku 2.5.

Rysunek 2.5:

Dokładniejszy opis znajduje się w opisie poszczególnych operacji.



Rysunek 2.6: Schemat operacji podniesienia semafora

Podniesienie semafora

Algorytm podniesienia semafora obrazuje rysunek 2.6. Gdy semafor jest podnoszony, wywoływana jest funkcja asemblerowa `void up(struct semaphore * sem)` z pliku `semaphore.h`. Funkcja ta zwiększa `count` o jeden, a następnie sprawdza jaką `count` ma wartość. Jeżeli jest mniejsza lub równa zero, oznacza to, że jakiś proces czekał na semaforze. Wówczas zwiększana jest wartość atrybutu `waking` o jeden (jeden proces będzie mógł przejść pod semaforem) i budzone są wszystkie procesy wiszące na semaforze (wszystkie znajdujące się w kolejce `wait`). Jeden z obudzonych procesów (ten, który najszybciej otrzyma dostęp do procesora) zmniejsza wartość zmiennej `waking` i przechodzi pod semaforem.

Opuszczenie semafora

Operację tę, przedstawioną na rysunku 2.7, realizują dwie funkcje: `void down(struct semaphore * sem)` oraz `int down_interruptible(struct semaphore * sem)`. Pierwsza z nich opuszcza semafor i usypia proces, w sposób w którym nie przyjmuje sygnałów, druga pozwala na odebranie sygnału. Każda z tych funkcji zmniejsza początkowo wartość atrybutu `count` (jeden proces więcej będzie wisiał na semaforze lub jeden mniej może jeszcze przejść pod semaforem). Jeżeli wartość `count` jest mniejsza od zera, oznacza to, że proces nie może przejść pod semaforem i musi się zawiesić. W tym przypadku proces wstawia się do kolejki procesów wiszących na semaforze `wait`, a następnie próbuje w pętli odjąć jeden od atrybutu `waking` (tylko wtedy, gdy `waking` jest większe od 0). W każdym obrocie pętli po nieudanym zmniejszeniu `waking` proces wywołuje funkcję `schedule()` (`sched.c`). Proces wychodzi z pętli dopiero, gdy uda mu się zmniejszyć `waking` lub (tylko w przypadku `__down_interruptible`) gdy dostanie sygnał. W tym drugim przypadku funkcja `__down_interruptible` zwraca wartość, która różna od zera mówi o tym, że proces został przerwany przez sygnał. Po wyjściu z kolejki proces usuwa się z kolejki `wait`.

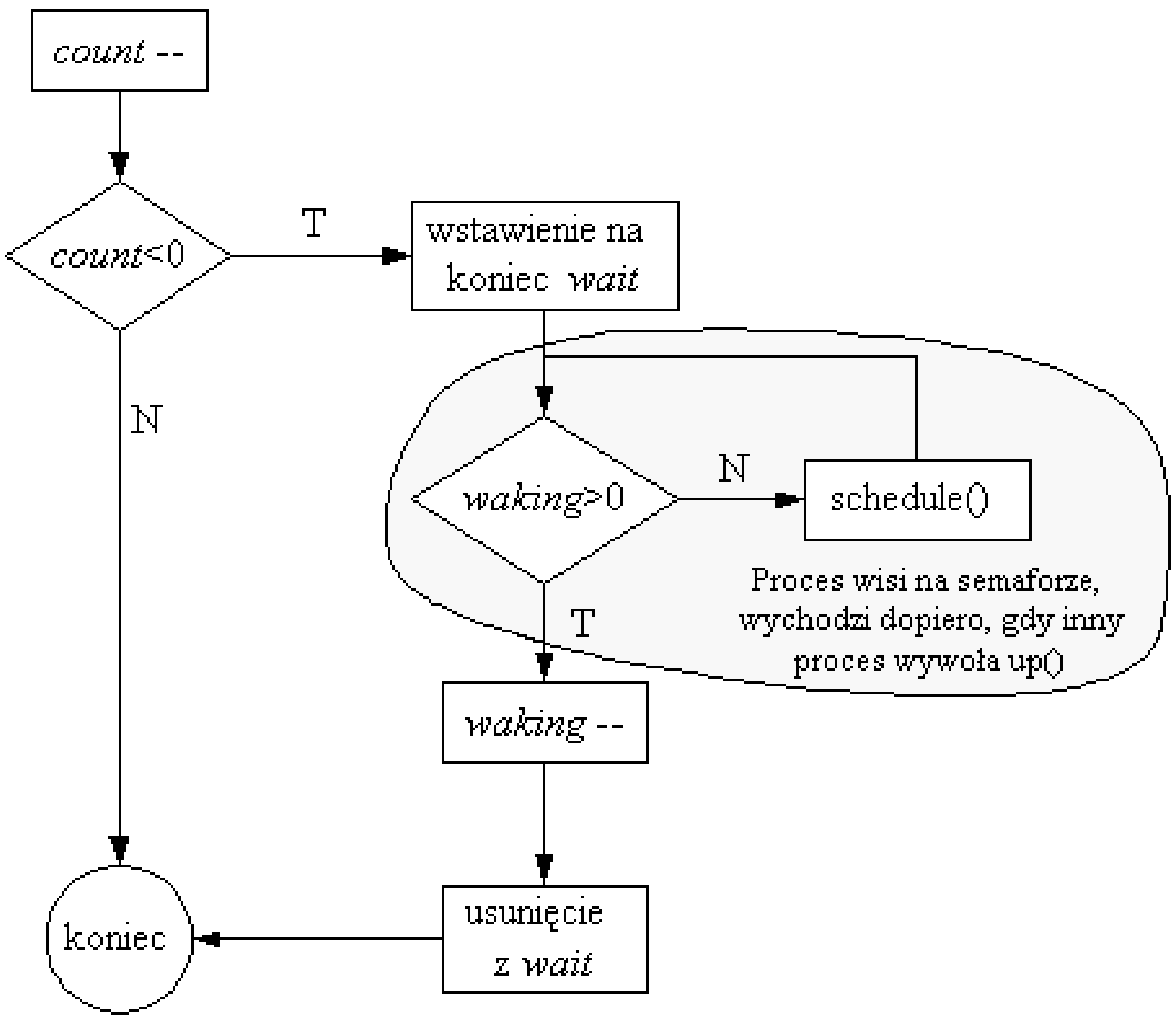
Warunkowe opuszczenie semafora(Rysunek 2.8)

Operacja ta nie jest klasyczną operacją na semaforach, natomiast w jądrze 2.2 została zaimplementowana. Operacja ta próbuje opuścić semafor, jednak w przypadku niepowodzenia nie wiesza procesu, tylko rezygnuje z dostępu do semafora. Operację tę realizuje funkcja `int down_trylock (struct semaphore * sem)`. Jej działanie jest proste. Sprawdza, czy `count` jest większy od zera, jeżeli jest, to zmniejsza ten atrybut o jeden oraz przechodzi pod semaforem. Jeżeli `count` jest mniejsze lub równe zero, oznacza to, ze proces ten musiałby zawisnąć na semaforze. W tym przypadku proces rezygnuje z dostępu. Gdy udało się przejść pod semaforem funkcja zwraca 1, w przeciwnym przypadku 0.

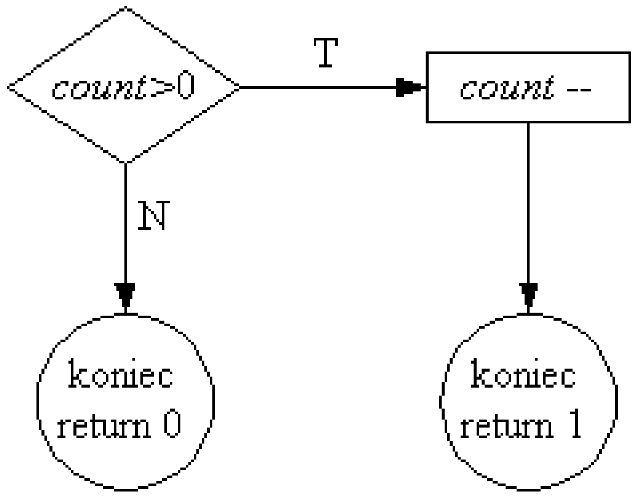
2.5 Obsługa sygnałów

2.5.1 Ogólnie o sygnałach

Sygnały to jeden ze sposobów komunikacji między procesorami. Zostały one stworzone głównie w celu zapewnienia poprawnej obsługi sytuacji wyjątkowych. Sygnał może być wysłany przez jądro lub przez proces, a jego odbiorcą jest zawsze proces (może być on jednocześnie nadawcą). Główne cechy komunikacji sygnałami to:



Rysunek 2.7: Schemat operacji opuszczenia semafora



Rysunek 2.8: Schemat operacji warunkowego opuszczenia semafora

- asynchroniczność - wysyłający nie czeka, aż proces-adresat odbierze sygnał
- jedyną informacją przekazywaną przez sygnał jest jego numer
- nadawca sygnału jest nieznany
- ilość przesłanych sygnałów danego typu (o tym samym numerze) jest nieznana, proces-odbiorca wie tylko, że sygnał się pojawił
- proces nie musi odsługiwać sygnału czy reagować na niego (nie dotyczy części sygnałów)

Zatem sygnały mają dużo mniejsze możliwości niż mechanizmy IPC takie jak np. kolejki komunikatów. W wielu sytuacjach są jednak bardzo potrzebne.
Sygnały można podzielić na:

- sygnały wysyłane przez jądro, gdy proces spowoduje błąd działania systemu czy sprzętu: SIGILL, SIGIOT, SIGBUS, SIGFPE, SIGSEGV, SIGXCPU, SIGXFSZ, SIGPWR
- sygnały nakazujące procesowi zakończenie, wstrzymanie lub wznawianie działania, lub informujące o tym: SIGINT, SIGQUIT, SIGKILL, SIGTERM, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP
- sygnały terminala: SIGHUP, SIGTTIN, SIGTTOU, SIGWINCH
- sygnały wejścia / wyjścia: SIGPIPE, SIGURG, SIGIO
- sygnały użytkowe: SIGUSR1, SIGUSR2, SIGALRM, SIGVTALRM
- sygnały pojawiające się, gdy program jest śledzony: SIGTRAP, SIGPROF

Jak już wcześniej napisałem, proces może sam decydować o obsłudze większości sygnałów. Oprócz obsługi domyślnej proces może ignorować dany sygnał, lub "podczepić" pod sygnał własną funkcję jego obsługi (ang. *handler*). Nie można jedynie zmienić obsługi sygnału o numerze 9: SIGKILL. Domyślną obsługą dla niektórych sygnałów jest ich ignorowanie, dla innych zakończenie działania procesu z ewentualnym zrzuceniem obrazu pamięci na dysk.

Sygnał do procesu może być wysyłany przez:

- jądro
- inny proces (funkcja systemowa kill())
- bezpośrednio przez użytkownika (polecenie systemowe kill)

2.5.2 Struktury danych

Najważniejsze pola w strukturze `task_struct` (rekordzie opisującym proces) używane w mechanizmie przesyłania sygnałów to:

- `sigset_t signal` - długie słowo, w którym każdy z bitów oznacza, czy dany sygnał został przesłany do procesu
- `sigset_t blocked` - analogicznie, bit określa, czy sygnał jest zablokowany
- `struct signal_struct *sig` - tablica, której głównym celem jest przechowywanie informacji o zachowaniu procesu po odebraniu danego sygnału.

Wysłanie sygnału do procesu polega na ustawieniu odpowiedniego bitu w polu `signal` jego struktury `task_struct`. Proces dowiaduje się o przyjjściu sygnału odczytując w pewnym momencie zawartość tego pola. Dokładniejszy opis sposobu obierania sygnałów przez procesy jest zamieszczony dalej.

Głównym polem struktury `signal_struct` jest struktura `sigaction` zawierająca pola:

- `sa_handler` - adres funkcji obsługi sygnału (handlera); wartość `SIG_IGN` (ignorowanie sygnału) lub `SIG_DFL` (obsługa domyślna)
- `sa_mask` - maska sygnałów, które są blokowane w trakcie działania handlera sygnału
- `sa_flags` - flagi określające różne sprawy związane z obsługą sygnału

2.5.3 Wysyłanie sygnałów

Jak wiemy, wysłanie sygnału do procesu polega na odpowiednim zmodyfikowaniu pola signal w jego strukturze `task_struct`. Jednak część sygnałów można lub trzeba obsłużyć w momencie ich nadawania. Przykładowo przy wysłaniu sygnału SIGCONT proces powinien być obudzony. Nie ma sensu też informować procesu o wystąpieniu sygnału, który on ignoruje. Takie i podobne czynności wykonuje funkcja wewnętrzna jądra `send_sig_info()`, która bezpośrednio odpowiada za wysłanie konkretnego sygnału do danego procesu:

```
int send_sig_info(int sig, struct siginfo *info, struct task_struct *t)
```

Algorytm 3 `send_sig_info`

Dane wejściowe: `sig` - numer sygnału
`info` - informacje o wysyłającym
`t` - proces docelowy
Wynik: numer błędu lub 0

```
jeśli błędny numer sygnału to
  return -EINVAL;
koniec jeśli
jeśli wysyłający nie ma uprawnień do wysyłania sygnału to
  return -EPERM;
koniec jeśli
jeśli sygnał = 0 lub proces jest zombie to
  return 0;
koniec jeśli
jeśli sig in (SIGKILL, SIGCONT) to
  jeśli proces był zastopowany to
    budź go: wake_up_process(t);
  koniec jeśli
  wyzeruj w polu signal bity odpowiadające sygnałom SIGSTOP, SIGTSTP, SIGTTOU,
  SIGTTIN
koniec jeśli
jeśli sig in (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) to
  wyzeruj bit odpowiadający SIGCONT;
koniec jeśli
jeśli sygnał może być dalej zignorowany to
  jeśli proces jest w stanie TASK_INTERRUPTIBLE i sygnał jest “w toku” to
    wake_up_process(t);
  koniec jeśli
  return 0;
koniec jeśli
ustaw odpowiedni bit w polu signal;
jeśli sygnał nie jest zablokowany to
  ustaw: sygnał „w toku”;
koniec jeśli
jeśli proces jest w stanie TASK_INTERRUPTIBLE i sygnał jest “w toku” to
  wake_up_process(t);
koniec jeśli
return 0;
```

Sygnał może być przesłany do procesu, gdy:

- wysyłający jest superużytkownikiem, lub
- wysyłany jest sygnał SIGCONT oraz proces wysyłający i odbierający są z tej samej sesji, lub
- identyfikator wysyłającego (uid lub euid) jest równy identyfikatorowi użytkownika procesu odbierającego (uid, euid), lub
- wartość uprzywilejowania w parametrze `info` funkcji `send_sig_info` jest niezerowa

Z punktu widzenia programisty lub użytkownika systemu do wysyłania sygnałów służy funkcja systemowa `kill()`. Zależnie od swych parametrów wywołuje ona funkcje `kill_proc()`, `kill_sl()` lub `kill_pg()`, które dopiero korzystają z funkcji `send_sig_info()`. Zdarza się, że sygnał musi być wysłany bezwarunkowo - służy do tego wewnętrzna funkcja jądra `force_sig_info()`:

```
int force_sig_info(int sig, struct siginfo *info, struct task_struct *t)
```

Algorytm 4 `force_sig_info`

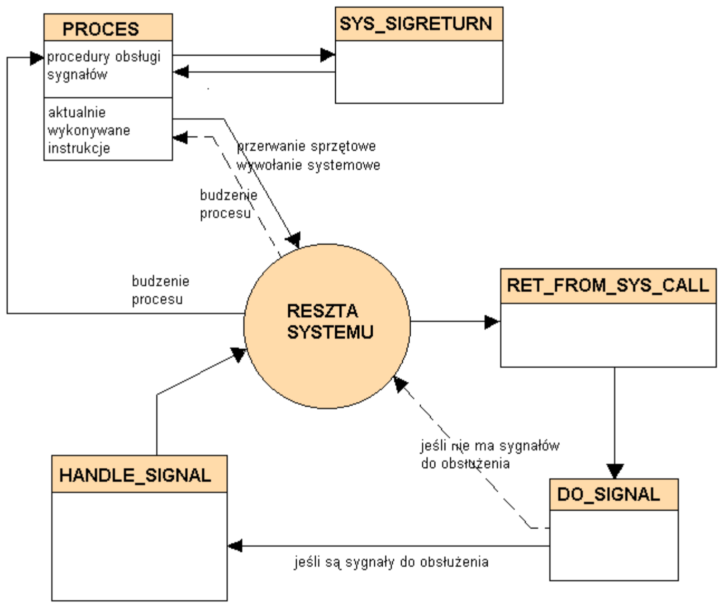
Dane wejściowe: analogicznie jak w funckji `send_sig_info`
Wynik: numer błędu lub 0

jeśli proces jest zombie **to**
  `return -ESRCH;`
koniec jeśli
jeśli obsługa sygnału jest ustawiona na `SIG_IGN` (ignorowanie) **to**
  zmień obsługę na `SIG_DFL` (domyślną);
koniec jeśli
zeruj bit sygnału w polu `blocked`;
`send_sig_info(...);`

2.5.4 Odbieranie sygnałów

Odbieraniem sygnałów zajmuje się funkcja jądra `do_signal()`[3]. Jest ona wywoływana z procedury `ret_from_sys_call`[4], a ta z kolei uruchamiana jest przy powrotach z funkcji systemowych i z procedur obsługi niektórych przerwań (np. przerwania zegarowego) – czyli dość często. Z sygnałami obsługiwanymi w sposób domyślny `do_signal()` radzi sobie samodzielnie, natomiast obsługę sygnałów, dla których proces ma zdefiniowaną własną procedurę obsługi zleca procedurze `handle_signal()`[5]. Co robi `handle_signal()`? Nie może po prostu wykonać zdefiniowanej przez proces procedury obsługi, gdyż wtedy wykonałaby się ona w trybie jądra. Zamiast tego dokonuje sprytnych manipulacji na stosie i wskaźniku instrukcji procesu tak, żeby po obudzeniu rozpoczął on swoje działanie właśnie od wykonania procedury obsługi dla otrzymanego sygnału. Co dokładnie dzieje się na stosie – patrz niżej. Rysunek 2.9 przedstawia schemat przepływu sterowania przy obsłudze sygnałów.

[3] jej implementację można znaleźć w /usr/src/linux/arch/i386/kernel/signal.c
[4] jej implementację można znaleźć w /usr/src/linux/arch/i386/kernel/entry.S (koszmarek w asemblerze)
[5] jej implementację można znaleźć w /usr/src/linux/arch/i386/kernel/signal.c



Rysunek 2.9: Schemat przepływu sterowania

Czas na szczegóły.

Funkcja `handle_signal` zajmuje się sygnałami, dla których proces zdefiniował własne procedury obsługi. Stos i wskaźnik instrukcji procesu modyfikowane są tak, żeby proces po otrzymaniu procesora wykonał najpierw procedurę obsługi dla danego sygnału, a następnie wywołał funkcję jądra `sys_sigreturn`[6]. Co się dzieje ze stosem? `handle_signal` wkłada tam (licząc od dołu):

- adres powrotu, czyli adres instrukcji, którą proces właśnie miął wykonać
- adres funkcji `sys_sigreturn`

Wskaźnik instrukcji procesu ustawiany jest na pierwszej instrukcji procedury obsługi. Jeśli jest kilka sygnałów do obsłużenia, to tworzonych jest kilka ramek na stosie.

Procedura obsługi sygnału standardowo nie może być przerwana przez obsługiwany właśnie sygnał – jest on blokowany. Oczywiście użytkownik może to zmienić ustawiając flagę `SA_NODEFER` w strukturze `sigaction` dla danego sygnału. Blokowaniem tego sygnału zajmuje się właśnie procedura `handle_signal`. Odblokowuje go procedura `sys_sigreturn`

Przy definiowaniu nowej procedury obsługi sygnału użytkownik może zażądać, żeby po pierwszym wywołaniu tej procedury została przywrócona obsługa domyślna (należy ustawić flagę `SA_ONESHOT` w procedurze `sigaction`). Przywracaniem domyślnej obsługi zajmuje się także procedura `handle_signal`.

[6]jej implementację można znaleźć w /usr/src/linux/arch/i386/kernel/signal.c

Algorytm 5 do_signal

```
dopóki są sygnały do obsłużenia wykonuj
  wybierz sygnał
  jeśli proces jest śledzony i sygnał to nie SIGKILL to
    zatrzymaj proces
    powiadom debugger ( funkcja notify_parent )
    wywwołaj funkcję szeregującą
    jeśli sygnał został zablokowany przez debugger to
      wyślij go jeszcze raz do siebie
    koniec jeśli
  w przeciwnym przypadku
    jeśli sygnał jest ignorowany to
      jeśli to SIGCHLD to
        posprzątaj po zmarłych potomkach ( funkcja sys_wait4 )
      koniec jeśli
    w przeciwnym przypadku
      jeśli sygnał jest obsługiwany standardowo to
        wykonaj odpowiednie działania*)
      w przeciwnym przypadku
        wywołaj handle_signal
      koniec jeśli
    koniec jeśli
  koniec jeśli
koniec dopóki
```

*)czyli np.:

- SIGCONT,SIGCHLD - (ignorowane) przejdź do obsługi kolejnego sygnału
- SIGSTOP - (zatrzymanie procesu) zatrzymaj proces, powiadom o tym jego proces macierzysty, wywołaj funkcję szeregującą
- SIGKILL,SIGSEGV - (zakończenie procesu) zakończ proces, ewentualnie wykonaj zrzut pamięci na dysk (SIGSEGV)

Jak można wpływać na sposób obsługi sygnałów

- Blokowanie sygnału

  Służy do tego funkcja `sys_sigprocmask`[7] (`sigprocmask` dla programisty). Do sprawdzania, które z blokowanych sygnałów nadeszły służy funkcja `sys_sigpending`[8] (`sigpending` dla programisty). Obie funkcje operują na polu `blocked` w strukturze `task_struct`.

- Własna procedura obsługi

  Do ustawiania własnej procedury obsługi służy funkcja `sys_sigaction`[9] (`sigaction` dla programisty). Modyfikuje pole `sig` w strukturze `task_struct`

2.6 Funkcje systemowe do obsługi procesów

2.6.1 Funkcja `fork()` - tworzenie nowego procesu

Wprowadzenie

Zadaniem funkcji `fork()` jest tworzenie nowego procesu, będącego potomkiem procesu wywołującego. Wszystkie procesy w systemie powstają za jej pomocą, oprócz procesu o pidzie 0 (idle), który jest tworzony wewnętrznie przez jądro przy ładowaniu systemu do pamięci.

Podstawowy interfejs programisty: `pid_t fork()` gdzie wartość zwracana jest liczbą określającą tożsamość procesu:

- ojciec otrzymuje pid potomka
- potomek otrzymuje zero

lub informującą o błędzie w przypadku niepowodzenia przy tworzeniu nowego procesu: -1. Stąd typowy sposób użycia fork:

```
main(argv,argc)
{
  switch(fork())
  {
    case -1 : /* zgłoszenie błędu */
    case  0 : /* bezpośrednio kod potomka lub wywołanie funkcji exec */
    default : /* kod ojca */
  }
}
```

Proces potomny dziedziczy po macierzystym cały jego kontekst, z jedynym wyjątkiem - właśnie wartością zwracaną przez `fork()`. Powstanie nowego procesu jest niemożliwe w przypadku braku pamięci lub przy przekroczeniu limitu na liczbę procesów.

[7] jej implementację można znaleźć w /usr/src/linux/kernel/signal.c
[8] jej implementacje można znaleźć w /usr/src/linux/kernel/signal.c
[9] jej implementację można znaleźć w /usr/src/linux/arch/i386/kernel/signal.c

Szkic algorytmu dla funkcji `fork()`

Jądro wykonuje następujące działania:

- Sprawdza dostępność zasobów:
  - czy nie została przekroczona maksymalna dostępna liczba zadań w systemie, określona przez stałą `NR_TASKS` (wynoszącą domyślnie 512, w wersji jądra 2.2.12),
  - czy pozostanie wystarczająca liczba możliwych procesów dla administratora (roota) ustalona przez `MIN_TASKS_LEFT_FOR_ROOT` (wynoszącą 4 w tejże wersji),
  - czy dany użytkownik nie przekroczy swojego limitu liczby procesów określonego przez stałą `MAX_TASKS_PER_USER` (wynoszącą przykładowo `NR_TASKS`/2).
- Przydziela nowemu procesowi:
  - wolną pozycję w tablicy procesów,
  - unikatowy pid.
- Inicjuje wartości struktury `task_struct`:
  - kopiując większość danych ze struktury procesu macierzystego,
  - zaznaczając, że nowy proces nie wykonał jeszcze exec i że nie można go przerwać ani wyrzucić z pamięci.
- Tworzy logiczną kopię kontekstu ojca:
  - kopiując informacje o deskryptorach plików, procedury obługi sygnałów oraz - za pomocą techniki *kopiowanie przy zapisie* (ang. *copy-on-write*) (kopiuj dopiero przy próbie zapisu) - segmenty pamięci,
  - uwzględniając ewentualne żądania współdzielenia pewnych zasobów przez ojca i syna (wtedy nie kopiuje zasobów, lecz zwiększa odpowiednie liczniki dostępu).
- Zwiększa plikom, związanym z rozmnożonym procesem, liczniki w tablicy plików oraz i-węzłów.
- Wstawia nowy proces do kolejki procesów gotowych do wykonania.
- Przekazuje:
  - pid potomka - procesowi macierzystemu,
  - 0 - potomkowi.

Implementacja

Funkcja `fork` dostępna dla programisty jest zaimplementowana za pomocą ukrytej dla niego funkcji `do_fork()`, której funkcje składowe, znajdujące się w pliku `kernel/fork.c` omówione są poniżej:

- Funkcja `find_empty_process()`: sprawdza, czy uruchomienie kolejnego procesu nie spowoduje przekroczenia któregoś z limitów na liczbę procesów. Następnie przeszukiwana jest globalna tablica zadań, w celu znalezienia indeksu pozycji, która ma wartość `NULL`. Jeśli nie ma wolnej pozycji, to zwracaną wartością jest EAGAIN, wpp. znaleziony indeks.

- Funkcja `get_pid()`: zwraca unikatowy (wyjątek - funkcja `clone()`) identyfikator procesu. Nowy pid jest równy zwiększonemu o jeden ostatnio przydzielonemu pidowi, przy czym jeśli tak obliczony pid przekroczy wartość 32768, to zostaje zredukowany do 1. Następnie jest sprawdzane, czy nowy pid nie koliduje z żadnym wykorzystywanym aktualnie pidem lub identyfikatorem grupy (pgrp). Jeżeli jest kolizja to bierzemy kolejny pid.

- Funkcja `dup_mmap()`: funkcja składowa funkcji `copy_mm()`. Kopiuje strony pamięci procesu macierzystego do przestrzeni potomka za pomocą techniki *copy-on-write*, tzn. fizyczne kopiowanie strony odbywa się dopiero przy próbie zapisu na niej przez jeden z tych procesów.

- Funkcja `copy_mm()`: przydziela miejsce na strukturę opisującą pamięć i wywołuje powyższą funkcje - `dup_mmap()`. Zeruje statystyki dotyczące dostępu do pamięci. W przypadku żądania współdzielenia pamięci przez ojca i potomka segmenty nie są kopiowane, lecz są zwiększane odpowiednie liczniki odwołań.

- Funkcja `copy_fs()`: inkrementuje liczniki odwołań do i-węzłów katalogu głównego i aktualnego oraz tworzy strukturę `fs_struct`

- Funkcja `copy_files()`: kopiuje prywatną tablicę deskryptorów plików procesu oraz zwiększa liczniki odwołań do i-węzłów tych plików.

- Funkcja `copy_sighand()`: kopiuje procedury obsługi sygnałów.

Zbierzmy wszystko razem: funkcja `fork()` Jest to główna funkcja algorytmu `fork`, wywołująca wszystkie wyżej wspomniane funkcje. Jej dużą zaletą jest, że w razie niepowodzenia na jakimkolwiek etapie usuwa ona wszelkie utworzone przez siebie struktury i wycofuje wprowadzone ustalenia. Kolejne czynności:

1. alokacja pamięci na:
   - strukturę `vtask_struct`, do której wskaźnik zostanie umieszczony w tablicy procesów.
   - stos jądra
2. wywołanie funkcji `find_empty_process()`
3. inicjalizacja różnych wartości struktury nowego procesu:
   - zaznaczenie, że proces jest nowy i nie wykonał jeszcze exec, oraz, że procesu nie można wyrzucić z pamięci oraz go przerwać
   - przypisanie pidu procesu
   - wyzerowanie zegara
4. wstawianie procesu do tablicy procesów
5. utworzenie dowiązań sąsiadów i przodków w tablicy procesów

6. kopiowanie informacji o deskryptorach plików: `copy_files()`
7. wywołanie funkcji `copy_fs()`
8. wywołanie funkcji `copy_sighad()`
9. wywołanie funkcji`copy_mm()`
10. wywołanie niskopoziomowej funkcji `copy_thread()`, która powoduje 'fizyczne rozdzielenie' procesów ojca i potomka. Z punktu widzenia procesu potomnego sytuacja wygląda tak jakby wywołał on funkcję systemową `fork()`, a następnie otrzymał kod powrotu 0.
11. wywołanie funkcji `wake_up_process`, która zmienia flagę nowego procesu na `TASK_RUNNING` oraz wstawia go do kolejki procesów gotowych do wykonania.

Pokrewne funkcje systemowe

Interfejs ukrytej dla programisty funkcji `do_fork` stanowią trzy funkcje:

- `fork()`
- `vfork()`
- `clone()`

Opiszemy teraz dwie ostatnie z nich.

- Nadmiarowa funkcja `vfork()`. Funkcja ta pochodzi z systemu BSD, gdzie powstała aby przyśpieszyć tworzenie nowego procesu. Było to osiągnięte poprzez to, że potomek operował w przestrzeni adresowej ojca, więc kopiowanie było zbędne. Jednak wadą funkcji `vfork` było założenie, że natychmiast po niej musiała być wywołana funkcja exec, która sama przydzieli nowemu procesowi pamięć. W *Linuksie* rozwiązano to inaczej: za pomocą wspomnianej techniki copy-on-write, czyli poprzez współdzielenie pamięci, aż do próby zapisu na niej. Podejście to czyni zbędnym założenie o wywoływaniu exec i zapewnia, że nie będzie niepotrzebnego kopiowania. Zatem w *Linuksie* `fork` i `vfork` są synonimami.
- Funkcja `clone()` - wsparcie wątków[10]. Funkcja ta umożliwia współdzielenie przez proces macierzysty i potomny rozmaitych zasobów:
  - pamięci
  - tablicy deskryptorów
  - tablicy obsługi sygnałów
  - pidu, co ułatwia implementację wątków

  Funkcja staje się dostępna po skompilowaniu jądra z ustawionym symbolem `CLONE_ACTUALLY_WORKS_OK`.

[10] więcej na temat funkcji `clone()` w rodziale 2.7.

2.6.2 Funkcja `exec()`

Przeznaczenie

Każda spośród sześciu funicji rodziny `exec` zmienia kontekst poziomu użytkownika danego procesu na podstawie programu wykonywalnego umieszczonego w pliku dyskowym (nie jest tworzony żaden nowy proces; w szczególności nie zmienia się identyfikator i miejsce procesu w hierarchii procesów). Innymi słowy, kiedy proces uruchamia (z powodzeniem) funkcję `exec`, istniejący proces zostaje pokryty nowym kodem programu (mowa o obrazach kodu w pamięci operacyjnej: segmentach kodu, danych i stosu). W większości programów funkcja `exec` występuje w parze z `fork`, co pozwala uruchomić nowy proces z wnętrza już istniejącego (w niektórych systemach operacyjnych istnieje wywołanie systemowe `spawn` łączące role obu tych funkcji).

Sygnatury funkcji z kategorii `exec` przedstawiają się następująco:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, char *);
int execv(const char *path, char *const argv[]);
int execls(const char *path, char *const arg0, ..., const char *argn, char *,
  char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn, char *);
int execvp(const char *file, char *const argv[]);
```

Drobne różnice w sposobie działania tych funkcji są sugerowane przez sufiksy poszczególnych nazw, rozpoczynające się za prefiksem exec. I tak, pierwsza litera sufiksu określa czy argumenty będą przekazywane do programu w postaci listy (l) czy tablicy (v) (w obu przypadkach ostatnim parametrem powinien być wskaźnik pusty (null)). Jeżeli sufiks nazwy funkcji posiada dalej jakąś literę, to określa ona, że: (e) programista skonstruuje (w formacie tablicowym) grupę zmiennych środowiskowych, którą przekaże do funkcji (wskaźnik na tę grupę stanie się trzecim argumentem funkcji `main()`), (p) podczas szukania plików wykonywalnych system operacyjny powinien korzystać z bieżącej wartości zmiennej PATH i w związku z tym parametr `path` nie musi być dokładną ścieżką dostępu.

Algorytm

Wymienione wyżej funkcje biblioteczne języka odwołują się do funkcji `sys_execve`, która po wykonaniu drobnych czynności organizacyjnych realizuje powierzone jej zadanie za pomocą systemowej funkcji `do_execve`. W kolejnych krokach algorytm realizuje następujące czynności:

- W pierwszej kolejności znajdowany jest i-węzeł związany ze wskazanym w argumencie plikiem, po czym następuje sprawdzenie czy operacja jest wykonalna (czy wskazany plik nie jest katalogiem lub urządzeniem zewnętrznym, czy można go wykonać) i gdy proces ma uzyskać uprawnienia administratora, wykonywane są testy bezpieczeństwa (czy proces nie jest śledzony, czy użytkownik jest administratorem, czy proces dzieli jakieś struktury z innymi procesami).

- Argumenty wywołania oraz zmienne środowiskowe kopiowane są do tablicy `page`, aby odpowiednie fragmenty obu kontekstów mogły bez przeszkód współistnieć w pamięci.

- Na podstawie pierwszych 128 bajtów pliku ustalany jest jego format (standardowo rozpoznawane są formaty: script, aplikacje i aplety javy, a.out, oraz najbardziej uniwersalny i wygodny - ELF; jeżeli format nie jest standardowy, a dostępny jest `kerneld`, to ładowany jest odpowiedni moduł i następuje ponowne sprawdzenie listy dostępnych formatów).
- Otwierany jest plik wskazany w wywołaniu funkcji.
- Zwalniane zostają segmenty danych, kodu i stosu starego procesu (o ile nikt inny z nich nie korzystał).
- Przywracana jest standardowa obsługa sygnałów, albowiem dotychczasowe adresy przechwytujące te sygnały są już nieaktualne. Uwaga ta nie dotyczy sygnałów ignorowanych przed wywołaniem funkcji, albowiem będą one ignorowane nadal.
- Zamykane są odpowiednie pliki (każdy deskryptor otwartego pliku posiada flagę określającą czy należy go zamknąć przy wywolaniu `exec`.
- Modyfikowana jest struktura `task_struct`. Dołączone i inicjowane zostają nowe segmenty danych i stosu, przyłączane strony z argumentami wywołania i zmiennymi środowiskowymi. Modyfikowane są wartości rejestrów oraz licznik instrukcji.
- Jeżeli któraś z powyższych operacji się nie udała, zwalniany jest i-węzeł i w zmiennej `errno` zwracany jest kod błędu.

Tablica 2.2: Możliwe (ważniejsze) błędy zgłaszane przez funkcję `exec`

Nazwa	**Przyczyna**
ENOENT	Błędna ścieżka dostępu do pliku
EINTR	Przechwycony sygnał podczas wywołania systemowego
E2BIG	Argumenty i zmienne powłoki nowego procesu zajmują zbyt dużo pamięci
ENOEXEC	Plik nowego procesu zawiera błędną wartość magiczną
EAGAIN	Ilość pamięci podczas operacji IO jest niewystarczająca
ENOMEM	Nowy proces wymaga zbyt dużo pamięci
EACCES	Zabroniony dostęp do pliku
ENOLINK	`path` wskazuje na nieaktywny zdalny system

2.6.3 Funkcja `exit()`

W systemie Linux procesy kończą się wykonując funkcję systemową `exit()`. Proces wywołujący `exit ()` przechodzi do stanu `ZOMBIE`, zwalnia swoje zasoby i oczyszcza kontekst, z wyjątkiem pozycji w tablicy procesów. Sposób wywołania funkcji jest następujący: `exit (status)`; przy czym `status` jest wartością całkowitą przekazywaną procesowi macierzystemu do zbadania. Przyjmuje się zasadę, że liczba zero oznacza poprawne zakończenie się procesu, a wartość statusu różna od zera wskazuje na wystąpienie błędu. Procesy mogą wołać `exit ()` jawnie bądź niejawnie na zakończenie programu; funkcja inicjująca dołączana do wszystkich programów w C wywołuje `exit ()` po powrocie programu z funkcji `main ()`. Z drugiej strony jądro może wywołać funkcję `exit`

`()` na potrzeby procesu po nadejściu nie przechwytywanego sygnału. W tym przypadku wartością `status` jest numer sygnału.

Algorytm 6 Exit

Dane wejściowe: kod zakończenia procesu
Wynik: brak

Zapisz dane o pracy procesu do pliku rozliczeniowego;
jeśli używał semaforów **to**
  Odwróć efekty wykonania operacji semaforowych;
koniec jeśli
Zwolnij pamięć przydzieloną procesowi na segmenty danych, kodu, tablice stron;
Zamknij otwarte pliki; Zwolnij i-węzły bieżącego i głównego katalogu;
Zamknij tablice obsługi sygnałów;
Ustaw stan procesu na `ZOMBIE`;
Ustaw wartość `exit_code`;
Powiadom proces macierzysty o swojej śmierci;
dopóki jest potomek **wykonuj**
  Uczyń proces init (1) procesem macierzystym wszystkich procesów potomnych. Jeśli któreś z dzieci było w stanie `ZOMBIE` to wyślij do `init` sygnał śmierci potomka;
koniec dopóki
jeśli lider grupy procesów związanej z terminalem sterującym **to**
  Wyślij sygnał zawieszenia do wszystkich procesów związanych z tym terminalem
koniec jeśli
Wywołaj szeregowanie procesów (bez powrotu)

Zapisywanie danych do pliku rozliczeniowego

Funkcja `acct_process()` zdefiniowana w pliku `sys.c` służy do zapisania danych o pracy procesu do pliku rozliczeniowego:

```
struct acct {
  char ac_comm[ACCT_COMM];// nazwa programu
  time_t ac_utime;        // czas w trybie użytkownika
  time_t ac_stime;        // czas w trybie jądra
  time_t ac_etime;        // czas pracy w sek.
  time_t ac_btime;        // czas rozpoczęcia pracy
  uid_t ac_uid;           // User ID
  gid_t ac_gid;           // Group ID
  dev_t ac_tty;           // identyfikator terminala
  char ac_flag;           /* info o zakończonym procesie:
     #define AFORK 0001 - proces wykonał fork bez exec'a
     #define ASU 0002   - proces używał przywilejów super-user'a
     #define ACORE 0004 - został wykonany zrzut pamięci (core dumped)
     #define AXSIG 0010 - proces został zabity przez sygnał */
```

```
  long ac_minflt;         // ilość małych i dużych błędów stron
  long ac_majflt;         // (w celu rozliczenia zużycia pamięci przez proces)
  long ac_exitcode;       // Kod zakończenia programu
};
```

Zegar procesu

Funkcja `del_timer()` zdefiniowana jest w pliku `shed.c`. Służy do usunięcia prywatnego zegara procesu. Zegar ten znajduje się w strukturze `timer_list` zadeklarowanej w pliku `time.h`. Wskaźnik do zegara procesu wykonującego 'exit' przechowywany jest w polu `real_timer` w strukturze `task_struct` procesu.

Modyfikacja wartości używanych semaforów

Funkcja `sem_exit()` zdefiniowana jest w pliku `sem.c`. Najpierw usuwa ona proces wykonujący 'exit' z listy procesów czekających na semaforze. Wskaźnik do elementu tej listy przechowywany jest w polu `semsleeping` w strukturze `task_struct` procesu. Następnie cofane są operacje semaforowe wykonywane ze znacznikiem `SEM_UNDO`. Lista tych operacji przechowywana jest w polu `semundo` w strukturze `task_struct` procesu.

Kolejka komunikatów jądra

Funkcja `kerneled_exit()` zdefiniowana jest w pliku `msg.c`. Obsługuje odłączenie procesu od kolejki komunikatów jądra. Jeżeli ta kolejka istnieje to zmniejszany jest licznik odwołań. W przypadku, gdy nie ma procesu korzystającego z kolejki, zostaje ona usunięta.

Struktury utworzone przez fork()

Funkcja `exit()` musi m.in. posprzątać to co kopiowała i utworzyła funkcja `fork()`. Wywołaniom z tej ostatniej odpowiadają następujące odwrotne funkcje w `exit()`:

konstruktory do_fork():	**destruktory do_exit():**	**opis**
`dup_mmap()`,`copy_mm()`	`exit_mm()`	zwolnienie pamięci przydzielonej procesowi na segmenty danych, kodu i tablice stron.
`copy_fs()`	`exit_fs()`	zamyka (zwalnia) ona i-węzły katalogu głównego oraz katalogu bieżącego
`copy_files()`	`exit_files()`	zamyka otwarte przez proces pliki
`copy_sighand()`	`exit_sighand()`	zwolnienie pamięci przydzielonej na tablicę obsługi sygnałów

Powiadamianie procesów spokrewnionych o swojej śmierci

Funkcja `exit_notify()` służy do usunięcia procesu wykonującego `exit()` z drzewa procesów, oraz wysłania sygnałów o śmierci procesu do odpowiednich procesów. Wszystkie dzieci bieżącego procesu adoptuje proces `INIT`. Dodatkowa funkcja `audit_ptree()` jest wywoływana jedynie wtedy, gdy zdefiniowana jest flaga `DEBUG_PROC_TREE`. Funkcja ta sprawdza poprawność drzewa procesów. W razie odkrycia błędu wyświetlane są odpowiednie komunikaty na konsoli.

Pliki źródłowe Linuksa

1. `kernel/exit.c` - implementacja funkcji exit
2. <`include/linux/sched.h`> - definicja struktury procesu (`task_struct`) i pamięci (`mm_struct`)
3. <`include/linux/acct.h`> - struktura rozliczeniowa (`acct`)
4. <`kernel/sys.c`> - zapisywanie do pliku rozliczeniowego (`acct_process`)
5. <`ipc/sem.c`> - modyfikacja wartości semaforowych (`sem_exit`)
6. <`drivers/char/tty_io.c`> - śmierć procesu kierującego terminalem sterującym (`disassociate_ctty`)

2.6.4 Funkcja `wait()`

Informacje ogólne

Czasami proces macierzysty potrzebuje synchronizacji swego działania. Funkcje `wait()` oraz `waitpid()` (pierwsza jest okrojoną wersją drugiej) powodują zawieszenie aktywności procesu nadrzędnego do czasu gdy jakiś (lub konkretny) proces zatrzyma się (przejdzie w stan STOPPED) lub zakończy (stan ZOMBIE). W dalszych punktach obydwie funkcje zostaną dokładniej omówione.

`wait()`

Wywołanie funkcji jest następujące:

```
pid_t wait(int *addr_stat)
```

Jak widać, funkcja wymaga podania jednego argumentu, którym jest wskaźnik do zmiennej całkowitej typu `pid_t` (`<sys/types.h>`). Będzie tam zwracana informacja o stanie złapanego potomka (parametrem funkcji może być wartość 0, co oznaczać będzie, że informacją tą nie jesteśmy zainteresowani).

Funkcja `wait()` oczekuje na *dowolnego* (pierwszego) potomka który osiągnie wymagany stan i kończy działanie natychmiast, gdy:

- proces wywołujący nie ma żadnych procesów potomnych; wówczas zwracana jest wartość -1, a zmienna `errno` przyjmuje wartość ECHILD (patrz dalej)
- któryś z procesów potomnych osiągnął wymagany stan *zanim* wywołana została funkcja `wait()`

W każdym innym wypadku `wait()` zawiesza proces macierzysty w oczekiwaniu na potomka.

Zwracane wartości

Kiedy przestanie działać proces potomny na którego przodek czeka, do procesu macierzystego zostaną zwrócone informacje o stanie potomka oraz identyfikator wychwyconego procesu.

Informacje o stanie zapisywane są w zmiennej `addr_stat`, dokładniej w pierwszych jej szesnastu bitach (pozostałe mają wartość 0).

- W razie *poprawnego zakończenia*, młodszy bajt będzie miał wartość 0, a starszy będzie zawierał kod powrotu (od 0 do 255).
- Jeżeli natomiast proces potomny został *zakończony z powodu nieprzechwycenia sygnału*, to młodszy bajt będzie zawierał numer sygnału (5 najmłodszych bitów), a starszy wartość 0. Jeżeli dodatkowo wygenerowany został plik zrzutu, ustawiony zostanie skrajny lewy bit młodszego bajtu.
- Gdy proces potomny został *wstrzymany*, wartość `exit_code` umieszczana jest w bajcie starszym, a w młodszym pojawia się wartość 0x7f.

`waitpid()`

```
pid_t waitpid (pid_t pid, int *stat_loc, int options)
```

Funkcja `waitpid()` jest rozszerzoną wersją funkcji `wait()`. Parametr `pid` określa grupę numerów identyfikacyjnych procesów potomnych, na które funkcja ma czekać.

Tablica 2.3: Dopuszczalne wartości parametru `pid` funkcji `waitpid()`

Wartość pid	**Oczekiwanie na**
$<$ -1	Dowolny proces potomny, którego gid == abs(pid)
-1	Dowolny proces potomny (tak jak w przypadku `wait()`)
0	Dowolny proces potomny, którego gid jest równy gid procesu wywołującego `waitpid()`
$>$ 0	Proces potomny o takim identyfikatorze

Trzeci argument funkcji jest wyrażeniem zerowym (nie ma wtedy znaczenia) lub złożonym z połączonych operacją OR znaczników wymienionych w kolejnej tabeli (zdefiniowanych w pliku `sys/wait.h`).

Tablica 2.4: Dopuszczalne wartości parametru `options` funkcji `waitpid()`

Znacznik	**Znaczenie**
WCONTINUED	Dowolny działający proces potomny, o którego stanie nie pobrano jeszcze informacji
WNOHANG	Natychmiastowe wyjście z funkcji z wynikiem 0 gdy nie można podać infomacji o stanie
WNOWAIT	Utrzymuje proces w stanie oczekiwania (będzie można na niego poczekać ponownie)
WUNTRACED	Dowolny zatrzymany proces potomny, o którego stanie nie pobrano jeszcze informacji

W razie gdy działanie `waitpid()` się nie powiedzie, zwrócona zostanie wartość -1 a do zmiennej `errno` wpisany zostanie kod błędu (patrz następny punkt).

Zgłaszane błędy

Funkcja `waitpid()` może zgłosić następujące rodzaje błędów:

Tablica 2.5: Możliwe błędy zgłaszane przez funkcję `waitpid()`

Nazwa	Przyczyna
EINTR	Przerwane wywołanie systemowe (przechwycony sygnał)
ECHILD	Proces określony parametrem `pid` nie istnieje
EINVAL	Nieprawidłowa wartość parametru `options`

Rzecz jasna jedynym błędem jaki zgłosić może funkcja `wait()` jest ECHILD w sytuacji gdy nie istnieją procesy na które można byłoby czekać.

Algorytm

Implementacją funkcji `wait()` jest funkcja systemowa `sys_wait4()`, której algorytm zaprezentowano w 7:

Ignorowanie sygnału śmierci

Proces może ustawić obsługę sygnału SIGCHLD na SIG_IGN. Oznacza to, że proces macierzysty nie jest zainteresowany stanem końcowym swoich procesów potomnych, zatem nie ma sensu przechowywać w systemie procesów w stanie ZOMBIE. Stąd też w momencie ogsługi sygnału SIGCHLD, jeśli proces go ignoruje, jądro samo (bez udziału procesu) będzie wywoływało `sys_wait4()` z opcją WNOHANG do skutku, tj. do momentu gdy nie będzie już żadnych procesów ZOMBIE.

2.7 Wątki w Linuksie

2.7.1 Nazewnictwo

Słowniczek

Blokujące funkcje systemowe - zatrzymują proces na czas wykonywania zleconego zadania (na przykład operacji wejścia/wyjścia), ułatwia to pisanie programów, ale często *nie daje* wystarczającej *efektywności* lub *reaktywności* (czyli program zawiesza się i nie odpowiada na polecenia użytkownika)

Asynchroniczne funkcje systemowe - powracają natychmiast do wywołującego procesu, a o wykonaniu zleconego zadania informuje pózniej, w dowolnym momencie, na przykład za pomocą sygnałów. Potrafi być istotnie trudniejsza w użyciu.

Granica ochrony - chroni jeden podsystem oprogramowania przed drugim. Na przykład między jądrem i różnymi procesami. Ogranicza komunikację między do jawnie dozwolonych metod. Przejście przez granicę ochrony jest **drogie**.

Wątki - jednocześnie wykonujące się procesy(czyli *sekwencyjne przepływy sterowania*), korzystające z wspólnego śródowiska systemowego(pamięci, tablicy deskryptorów plików). Komunikacja i przełączanie między wątkami jednego procesu Unix'wego jest dużo ta/nsza, niż pomiędzy różnymi procesami Unix'owymi.

Algorytm 7 sys_wait4()

```
sprawdź poprawność argumentów i ewentualnie zwróć błąd
wstaw proces do swojej kolejki wait_chldexit
dopóki są jeszcze jakieś dzieci, dla każdego z nich wykonuj
  jeśli proces nie należy do interesującego nas zbioru to
    przejdź do następnego potomka
  koniec jeśli
  jeśli stan == STOPPED to
    jeśli proces był już wychwycony to
      przejdź do następnego (exit_code == 0)
    koniec jeśli
    zapisz dane statystyczne o procesie
    wyzeruj exit_code
    usuń kolejkę wait_chldexit
    zwróć pid zatrzymanego procesu
  koniec jeśli
  jeśli stan == ZOMBIE to
    zwiększ czas wykorzystania procesora przez proces wywołujący wait o czas zużyty potomka
    zapisz dane statystyczne o procesie
    jeśli wskaźniki do p_opptr i p_pptr są różne to
      powiadom prawdziwego ojca o śmierci
    w przeciwnym przypadku
      zwolnij task_struct i miejsce w tablicy procesów
    koniec jeśli
    usuń kolejkę wait_chldexit
    zwróć pid zakończonego procesu
  koniec jeśli
koniec dopóki
jeśli nie ma dzieci to
  zwróc błąd
koniec jeśli
jeśli jest ustawiona opcja WHOANG to
  zwróc 0
koniec jeśli
jeśli nadszedł nieoczekiwany sygnał to
  zwróć błąd
koniec jeśli
zaśnij dopuszczając przerwania
wróć do 3
```

Czasem ze względu na róznice w narzutach czasowych (na komunikację i przełączanie procesora między kontekstami) określa się zwykłe procesy Unix'owe, jako **procesy ciężkie**, a wątki jako **procesy lekkie**. ęndsubsubsection

Rodzaje watków (według implementacji)

:

- **wątki poziomu jądra** - korzystają ze wsparcia jądra systemu Linux (sys_clone()), pozwalają wykorzystać wiele procesorów, operacje wejścia-wyjścia w jednym wątku nie blokują drugiego, zajmują miejsca w tablicy procesów, dają większe narzuty czasowe
- **poziomu użytkownika** - niewidoczne dla systemu (nie używaja wsparcia ze strony systemu), korzystaja z sygnalów (na przykład `SIGALRM` do przełaczania między wątkami, `SIGUSR1` i/lub `SIGUSR2` do mutex'ów i wątków), wykorzystują tylko jeden procesor, blokująca funkcja systemowa blokuje wszystkie wątki, mniejsze narzuty czasowe
- **hybrydowe** - łączą zalety powyższych rodzajów wątków tworząc pulę wątków-robotników (jest ich zazwyczaj N = liczba równoległych operacji wejscia-wyjscia + liczba procesorów) poziomu jądra przypisywąnych wątkom poziomu użytkownika, których może być dużo więcej

Inny sposó określania powyższych rodzajów wątków polega na wskazaniu ilu wątkom widzianym przez użytkownika odpowiada ile wątków widzianych przez jądro systemu:

- **1-1** - **wątki poziomu jądra**
- **1-n** - **wątki poziomu użytkownika**
- **m-n** - **wątki hybrydowe**

Oba rodzaje wątków moga mieć *ten sam interfejs*, jak na przykład **pthreads***(wątki poziomu użytkownika)* i **LinuxThreads** *(wątki poziomu jądra)* napisane przez Xavier'a Leroy'a. Zazwyczaj implementuje się *standard IEEE POSIX 1003.1c*, którego specyfikacja jest dostępna od IEEE za odpłatnoscią.

- Tworzenie wątków: `pthread_create()`
- Czekanie na zako/nczenie innego wątku: `pthread_join()`
- Blokowanie dostępu do sekcji krytycznych(semafory): `pthread_mutex_lock()`, `pthread_mutex_unlock()`
- Zawieszanie i oczekiwanie na wznowienie przez inny wątek(zmienne warunkowe): `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_bcast()`
- Mechanizm **cancellation points** - sprzatanie po wątku „nagle" przerwanym przez obsługę błędów lub przez inny wątek (`pthread_kill()`). W szczególności zwalnianie zajętych semaforów *(działa podobnie do obsługi wyjątków)*. `pthread_cleanup_push()`, `pthread_cleanup_pop()`

Mechanizm *cancellation points* gwarantuje że wątek nie może zostać przerwany w całkowicie dowolnym sposób. Może to być błąd który kończy wykonywanie zwykłego programu (na przyklad odebranie `SIGSEGV`). W przeciwnym razie musi się to zdarzyć w wybranych funkcjach biblioteki wątków lub funkcjach systemowych.

Obecnie *wsparcie w jądrze systemu* jest bardzo niewielkie i niskopoziomowe dla watków: funkcja `sys_clone()` umozliwia tworzenie procesów dzielących pamieć i środowisko. Mechanizmy synchronizacji*(semafory, zmienne warunkowe)* musza być zapewnione przez bibliotekę wątków.

Ograniczenia

Czego należy wymagać od *kodu*, który ma dobrze współpracować z wątkami:

- **reentrant** - wielokrotne *wejście* do kodu(na przykład funkcji) bez wychodzenia ze/n jest bezpieczne; kod nie moze używac żadnych zmiennych globalnych, ani statycznych lub musi to robic bardzo ostrożnie (w sekcji krytycznej lub operacjami atomowymi)
- **async-safe** - kod może byc wykonywany z procedury obsługi sygnalu, powinien byc **reentrant** i blokować sygnały przed blokowaniem jakichkolwiek semaforów
- **mt-safe** *(czyli multithread-safe)* - kod może być używany w programie wielowątkowym, zapewniając jednocześnie rozsądny poziom współbieżności

Ograniczenia obecnego jądra *Linux'a*:

- dla watków korzystających ze wsparcia jądra(funkcja `sys_clone()`):
- stała NR_TASKS w jądrze *(w pliku „sched.h”)*
  - standardowo 512
  - ograniczenia architektury Intel x86: `NR_TASKS=<4090`
- algorytmy w jadrze dzialaja w czasie liniowym względem liczby watków - w praktyce należy unikać większej ilości, niz 100 wątków poziomu jądra

Ograniczenia biblioteki *LinuxThreads*:

- tablica wątków jest wielkości `PTHREADS_MAX = 1024` (można zrekompilować bibliotekę)
- kazdy watek rezerwuje 2MB pamieci wirtualnej na stos, a na Intelu tylko pierwsze 1GB jest przeznaczone na stosy procesow (mozna zmienic wielkosc rezerwowanego obszaru uzywajac atrybutu setstackaddr)

2.7.2 Funkcja `__clone()`

Systemowa funkcja `__clone()` umożliwia zaimplementowanie wątków na poziomie jądra. Można za jej pomocą przekazać do funkcji jądra `do_fork()` kilka parametrów wpływających na współdzielenie zasobów (pamięć, deskryptory plików). Dzięki takiemu wsparciu, biblioteka wątków nie musi zajmować się przełączaniem zadań, wykorzystuje też wieloprocesorowość systemu (Linux będzie rozdzielał wątki między procesorami).

libc: `__clone()`

`__clone(int (*fn) (void *arg), void *child_stack,int flags,void *arg)`

Funkcji `__clone()` należy podać adres funkcji która ma być wykonana. Funkcja ta powinna z kolei przyjmować jeden argument typu `void*`, którego wartość przekazuje się do `__clone()` za pomocą parametru `arg`.

W przypadku współdzielenia pamięci, należy zaalokować stos dla nowego procesu i przekazać początkowy wskaźnik stosu tej funkcji `child_stack`.

Wsparcie dla wątków zapewniają flagi: CLONE_VM, CLONE_FS, CLONE_SIGHAND, CLONE_FILES. Są one bezpośrednio przekazywane funkcji `sys_clone` a ich działanie zostanie wyjaśnione poniżej.

Działanie:

- Wykonuje funkcję jądra sys_clone.
- Powrót w procesie potomnym: wywołuje podaną funkcję `fn`, której przekazywany jest wskaźnik na jej argumenty `arg`.
- Powrót w procesie ojca - działanie podobne jak przy zwykłym fork - wykonywanie kontynuowane jest w miejscu wywołania, tak samo też jest zwracany numer nowego procesu.

Po zakończeniu działania funkcji `fn` proces potomny umiera przekazując wynik jej działania jako swój kod wyjścia. Proces może też sam zakończyć działanie wywołując `exit()`.

kernel: `sys_clone()`

Pobiera z rejestrów flagi (`eax`) oraz nowy wskaźnik stosu (`ecx`). W przypadku gdy nowy wskaźnik stosu nie zostanie podany, używany jest adres stosu procesu-ojca, co jest niewskazane przy współdzieleniu pamięci (poprawne działanie takich procesów jest prawie niemożliwe). Następnie wywoływany jest`do_fork()`.

kernel: `do_fork()`

`int do_fork(unsigned long clone_flags, unsigned long usp, struct pt_regs *regs)`

Ta funkcja jest odpowiedzialna za stworzenie nowego procesu. Bezpośrednio w niej jest inicjalizowana tylko podstawowa struktura `task_struct` nowego procesu. Pozostałe struktury są obsługiwane w wyspecjalizowanych funkcjach. Tam też jest zaimplementowane właściwe wsparcie dla wątków.

- Kopiuje strukturę `task` bieżącego procesu.
- Przyznaje procesorowi nowy PID, o ile nie została przekazana flaga `CLONE_PID`. W takim wypadku wykorzystywany jest PID aktualnego procesu.
- Inicjalizuje pola stanu sygnałów, statystyk, etc.
- Wywołuje funkcje kopiujące pozostałe struktury (`copy_files(), copy_fs(), copy_sighand(), copy_mm()`) - opisane poniżej.

- ustawia parametry **lekkiego wątku** (ang *lightweight thread*) - rejestry procesora, segment stosu etc.
- Dodaje nowy proces do struktur jądra.
- Aktywuje nowy proces.

kernel: `copy_files()`

Ta funkcja inicjalizuje struktury zawierające dane o plikach otwartych przez proces. Aby współdzielić pliki należy użyć flagi `CLONE_FILES`. W takim przypadku, operacje na plikach wykonane w jednym z wątków będą miały skutki w pozostałych - dostępne będą otwierane pliki, zmieniać się będzie pozycja w plikach otwartych, etc. Dotyczy to w zasadzie wszystkich operacji na plikach które modyfikują w jakiś sposób ich stan.

- Jeżeli przekazano flagę `CLONE_FILES` zwiększa tylko licznik odwołań do struktury `files` i wychodzi.
- W przeciwnym wypadku tworzy nową strukturę `files` i kopiuje wszystkie struktury `file` z bieżącego procesu - Dostępne będą te same pliki ale operacje na deskryptorach będą niezależne.

kernel: `copy_fs()`

Inicjalizuje strukturę opisującą system plików na którym wykonuje się proces. Są w niej zawarte informacje o bieżącym katalogu oraz korzeniu (ang. *root*) drzewa katalogów czy też parametru `umask` bieżącego procesu. Aby współdzielić tę strukturę należy użyć flagi `CLONE_FS`, wtedy funkcje `chroot(), umask(), chdir()` będą zmieniać odpowiednie parametry dla wszystkich wątków.

- Jeśli przekazano flagę `CLONE_FS`, zwiększa licznik odwołań do bieżącej struktury `fs` i wychodzi.
- Jeśli nie, tworzy nową strukturę `fs` i inicjalizuje ją wartościami z bieżącego zadania.

kernel: `copy_sighand()`

Ustawia obsługę sygnałów dla nowo stworzonego procesu. Aby współdzielić obsługę sygnałów, należy posłużyć się flagą `CLONE_SIGHAND`. Operacje ustawiania obsługi sygnałów będą wtedy zmieniać ją we wszystkich wątkach. Raczej należy używać tego w połączeniu z CLONE_VM, w przeciwnym wypadku łatwo ustawić procedurę obsługi nieistniejąca w innym procesie.

Należy zauważyć, że dzielona jest obsługa sygnałów, same sygnały docierają jednak tylko do tych samych procesów do których dotarłyby bez współdzielenia. Nie są również współdzielone maski sygnałów. Jednakże, zgodnie ze specyfikacją wątków POSIX, sygnał wysłany do programu wielowątkowego powinien dotrzeć do dowolnego z wątków który nie ma zablokowanego danego sygnału. W związku z tym korzystając np. z biblioteki LinuxThreads należy używać sygnałów tylko gdy jest to konieczne, korzystając z funkcji LinuxThreads 'obchodzących' ograniczenia jądra. Albo nie korzystać z sygnałów w ogóle .

- Jeśli przekazana flaga `CLONE_SIGHAND`, zwiększa licznik odwołań do struktury `sig` i wychodzi.
- W przeciwnym wypadku tworzy nową strukturę `sig` i inicjalizuje ją wartościami z bieżącego procesu.

kernel: `copy_mm()`

Inicjalizuje struktury pamięci wirtualnej procesu. Jeśli użyjemy flagi `CLONE_VM` pamięć będzie współdzielona - tzn. wszystkie procesy będą się wykonywać na tym samym kodzie i z tymi samymi danymi. Każdy z tych procesów powinien mieć jednak osobny stos. Lokalizacja stosu nie jest parametrem struktur pamięci tylko lekkiego wątku i jest ustawiana później.

Dzielona pamięć oznacza, że bardzo łatwe jest przekazywanie danych między wątkami (choć to nie wystarcza do synchronizacji - mechanizmy synchronizacji powinny być i są zaimplementowane w bibliotece poziomu użytkownika np. LinuxThreads). Oznacza też, że funkcje mogą zostać wykonane wiele razy jednocześnie i należy o tym pamiętać - w szczególności jeśli korzystamy z jakichkolwiek danych które nie zostały zaalokowane na stosie, należy bądź blokować do nich dostęp za pomocą mechanizmów synchronizacji, bądź brać pod uwagę to, że mogą się nagle zmienić.

- Czyści informacje statystyczne nowego procesu
- Jeśli podana została flaga CLONE_VM, ustawia strukturę `mm` na tą z bieżącego procesu i wychodzi.
- W przeciwnym wypadku: tworzy nową strukturę `mm`, kopiuje informacje o segmentacji (`copy_segments()`), kopiuje mapy pamięci (`dup_mmap()`). Linux korzysta z techniki copy-on-write przy tworzeniu nowego procesu, więc pamięć zostanie skopiowana dopiero przy modyfikacji w którymś z procesów.

2.7.3 Biblioteka LinuxThreads

Oba rodzaje wątków moga mieć *ten sam interfejs*, jak na przykład **pthreads***(wątki poziomu użytkownika)* i **LinuxThreads** *(wątki poziomu jądra)* napisane przez Xavier'a Leroy'a. Zazwyczaj implementuje się *standard IEEE POSIX 1003.1c*, którego specyfikacja jest dostępna od IEEE za odpłatnoscią.

Ten krótki opis interfejsu programisty dotyczy zasadniczo LinuxThreads, ale funkcje tu opisane są identyczne z tymi w innych implementacjach, poza tymi zaznaczonymi jako nieprzenośnie.

Tworzenie wątków i operacje na nich

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
void *(*start_routine)(void *), void * arg)
```

Tworzy nowy wątek. Rozpocznie on wykonywanie w funkcji `start_routine`, `*arg` zostanie jej przekazane jako argument wywołania. Informacje o wątku zostaną zachowane w miejscu wskazywanym przez `*thread`. Wątek zakończy się w momencie wywołania funkcji `pthread_exit`, lub po wyjściu z funkcji `start_routine`. Wątek może też zostać **anulowany** (ang. *cancel*) przez inny wątek za pomocą funkcji `pthread_cancel()`. Argument `*attr` wskazuje na strukturę atrybutów wątku (opisane poniżej)

```
int pthread_attr_init (pthread_attr_t *attr)
int pthread_attr_destroy (pthread_attr_t *attr)
```

Tworzy lub niszczy strukturę atrybutów wątku. W przypadku tworzenia, atrybutom nadawane są wartości domyślne.

```
int pthread_attr_setPARAM (pthread_attr_t *attr, int value)
int pthread_attr_getPARAM (pthread_attr_t *attr, int *value)
```

Ustawia lub pobiera wartość parametru `PARAM` w sktrukturze `*attr`. Dostepne sa nastepujace parametry:

- **detachstate**

  Kontroluje czy można czekać na zakonczenie wątku (funkcja `pthread_join()`.

  - `PTHREAD_CREATE_JOINABLE`

    Można czekać na zakończenie wątku. Jesli nie użyje sie `pthread_join()`, zasoby zajmowane przez wątek, zaalokowane przez bibliotekę nie zostaną zwolnione. Jest to wartość domyślna. Można zmienic stan na drugi za pomocą funkcji `pthread_detach()`

  - `PTHREAD_CREATE_DETACHED`

    Nie można czekać na zakończenie takiego wątku, ale za to zasoby są zwalniane automatycznie.

- **schedpolicy**

  Ustawia **tryb szeregowania** (ang *scheduling policy*) dla danego wątku. Można zmieniac tryb szeregowania później, używając funkcji `pthread_setschedparam()`

  - `SCHED_OTHER` - zwyczajne, domyślne
  - `SCHED_RR` - czasu rzeczywistego, **round-robin**
  - `SCHED_FIFO` - czasu rzeczywistego, **FIFO**

- **schedparam**

  Ustawia priorytet wątku (tylko przy domyślnym szeregowaniu). Domyślnie 0.

- **inheritsched**

  Kontroluje czy parametry dotyczące szeregowania są dla nowego wątku pobierane z atrybutów czy kopiowane z wątku-ojca.

  - `PTHREAD_EXPLICIT_SCHED` - pobierane z atrybutów
  - `PTHREAD_INHERIT_SCHED` - kopiowane od ojca

- **scope**

  Kontroluje kontekst parametrów szeregowania.

  - `PTHREAD_SCOPE_SYSTEM`

    Wątki są widziane przez system jako osobne procesy. To oznacza, że ich priorytety/tryb szeregowania są rozpatrywane relatywnie do pozostałych procesów. To jest jedyny kontekst obsługiwany przez LinuxThreads.

  - `PTHREAD_SCOPE_PROCESS`

    Interpretacja parametrów szeregowania jest lokalna, tzn. relatywnie tylko do pozostałych wątków.

`int pthread_exit (void *retval)`

Kończy działanie wątku, wartość `*retval` jest ustawiana jako kod wyjścia (moze zostać pobrana przez `pthreadjoin()`

`int pthread_join(pthread_t thread, void **thread_return)`

Zawiesza bieżący wątek i czeka na zakończenie wątku opisanego parametrem `thread`. Musi on zezwalać na przyłaczenie (patrz opis atrybutów). `**thread_return` wskazuje na wartość wyjśnia. Ta funkcja jest punktem anulowania (patrz dalej), w przypadku otrzymania żadania anulowania wątek `thread` nie zostanie zwolniony.

Anulowanie wątków

Anulowanie (ang. *cancellation*) pozwala jednemu wątkowi zażądać zakończenia wykonywania innego wątku. Jeśli odbiorca honoruje takie żądanie, to działanie jest identyczne jak wywołanie przez niego `pthread_exit()`. Jako wartość wynikowa jest wtedy przekazywana stała `PTHREAD_CANCELLED`.

`int pthread_cancel (pthread_t thread)`

Wysyła żadanie anulowania do wątku `thread`.

`int pthread_setcancelstate (int STATE, int *OLDSTATE)`

Kontroluje czy bieżacy wątek ma honorować żadania anulowania. Stara wartość jest przechowywana w `OLDSTATE`. Parametr `STATE` może przybierac dwie wartości

- `PTHREAD_CANCEL_ENABLE` - domyślne, honoruje żądania.
- `PTHREAD_CANCEL_DISABLE` - ignoruje żądania anulowania.

`int pthread_setcanceltype(int TYPE, int *OLDTYPE)`

Jesśli wątek honoruje żądania anulowania, ustawia w jaki sposób są one obsługiwane. Jesli parametr `TYPE` ma wartość `PTHREAD_CANCEL_ASYNCHRONOUS` - tryb natychmiastowy - wątek zostanie zakończony natychmiast po odebraniu żadania. Wartość `PTHREAD_CANCEL_DEFERRED` - tryb opóźniony - powoduje, że żądanie zostanie rozpatrzone gdy wątek osiągnie **punkt anulowania** (ang. *cancellation point*). Takie punkty to prawie wszystkie funkcje biblioteki wątków powodujące zawieszenie się bieżącego wątku, jak również funkcje systemowe powodujące zawieszenie. Stara wartość jest przechowywana w `OLDSTATE`.

`void pthead_testcancel (void)`

Sprawdza, czy nie przyszło żądanie zakończenia, a jeśli tak, to kończy wątek.

Obsluga zakończenia wątku (*Cleanup handlers*)

Funkcje obsługi zakończenia wątku są wykonywane w momencie gdy wątek zakończy działanie, w porządku odwrotnym do tego w jakim były dodawane. Programista może za ich pomocą zwalniać zasoby zajęte przez wątek, na przyklad odblokowywać mutexy.

```
pthread_cleanup_push (void (*routine) (void *), void *arg)
pthread_cleanup_push_defer_np (void (*routine) (void *), void *arg)
```

Dodaje funkcję na stos funkcji obsługi, przy wykonaniu funkcji `routine` przekazany zostanie parametr `arg`. Druga funkcja jest nieprzenośna (tylko Linuxthreads) i oprócz wstawienia funkcji na stos, ustawia tryb odbierania żądań anulowania na opóźniony.

```
void pthread_cleanup_pop(int execute)
void pthread_cleanup_pop_defer_np(int execute)
```

Usuwa ostatnią funkcje ze stosu, jeśli `execute` różne od 0, także ją wykonuje. Druga z tych funkcji powinna być użyta dla odpowiadającej drugiej funkcji wstawiającej na stos - przywraca jednocześnie tryb reagowania na anulowanie.

Mutexy

Mutex (skrót od *MUTual EXclusion* - wzajemne wykluczanie) jest to semafor binarny służący do ochrony struktur danych przed równoległą modyfikacja. Używany jest też do implementacji sekcji krytycznych i monitorów.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mutexattr)
int pthread_mutex_destroy(pthread_mutex_t *mutex)
int pthread_mutexattr_init(pthread_mutexattr_t *attr)
int pthread_mutexattr_destroy(pthread_mutexattr *attr)
int pthread_mutexattr_setkind_np(pthread_mutexattr *attr, int kind)
int pthread_mutexattr_getkind_np(pthread_mutexattr *attr, int *kind)
```

Dwie pierwsze funkcje tworzą i niszczą mutex, dwie następne tworzą i niszczą strukturę atrybutów mutexa, dwie ostatnie (nieprzenośne, tylko LinuxThreads) ustawiają i pobierają rodzaj mutexa w jego atrybutach. Parametr kind moze miec wartosci:

- `PTHREAD_MUTEX_FAST_NP`

  „szybki”, domyślny. Próba zamknięcia już zamkniętego mutexa zawsze kończy się zawieszeniem wątku.

- `PTHREAD_MUTEX_RECURSIVE_NP`

  rekurencyjny, wątek który już „posiada” (tzn. zamknął go) mutexa może jeszcze raz go zamknąć. Aby otworzyć mutexa i udostępnić go innym wątkom trzeba go otworzyć tyle razy ile był zamykany.

- `PTHREAD_MUTEX_ERRORCHECK_NP`

  ze sprawdzaniem, próba zamknięcia mutexa który się już posiada zwraca błąd.

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
int pthread_mutex_trylock (pthread_mutex_t *mutex)
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Pierwsza funkcja zamyka mutexa i zawiesza się gdy nie jest w stanie, druga próbuje go zamknać i zwraca błąd gdy jest już zamknięty. Ich zachowanie na różnych typach mutexow zostało opisane wyżej. Ostatnia funkcja otwiera mutexa. Jeśli mutex jest ze sprawdzaniem, to nie można go otworzyć, chyba, że samemu się go zamknęło. Rekurencyjny mutex jest otwierany naprawdę dopiero gdy zostanie otworzony tyle razy co zamknięty, ale wątki mogą otwierać takie mutexy nawet jeśli same ich nie zamknęły. *Zamykanie mutexów nie jest punktem anulowania*, nie należy więc zajmować takich zmiennych na dłuższy czas. Jest to konieczne, żeby można było przewidzieć stan mutexów przy anulowaniu wątku. Dlatego też zalecane jest ustawianie trybu reagowania na anulowanie na opóźnione, jeśli korzysta się z mutexów (lub wykorzystanie nieprzenośnej wersji).

Zmienne warunkowe

Zmienne warunkowe to inna metoda synchronizacji wątków. Zapewniają one operacje: czekania (na spełnienie warunku) i sygnalizowanie (warunek prawdziwy). Zmienne takie muszą być powiązane z mutexami, aby uniknąć możliwych błędów.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_attr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond)
int pthread_condattr_init(pthread_condattr_t *attr)
int pthread_condattr_destroy(pthread_condattr_t *attr)
```

Tworzy lub niszczy zmienną warunkową. LinuxThreads nie obsługuje żadnych atrybutów dla zmiennych warunkowych, funkcje inicjalizujące takie struktury zapewnione są tylko dla zgodności ze standardem.

```
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Sygnalizuje warunek i budzi czekające na niego wątki. Pierwsza z tych funkcji budzi jeden (nieokreślone który) wątek, druga wszystkie. Jeśli nikt nie czeka, nic się nie dzieje.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec* abstime)
```

Odblokowuje `mutex` i zasypia w oczekiwaniu na sygnał. Po obudzeniu, zamyka z powrotem mutex. Operacja zasypiania i blokowania jest wykonywana atomowo, tzn. nie można zasygnalizować warunku pomiędzy tymi operacjami (co spowodowałoby zignorowanie sygnału), co pozwala na wykonanie przygotowania na oczekiwanie na warunek przy zamkniętym mutexie i prawidłową synchronizację. Mutex musi być zamknięty przez bieżący wątek przed wywołaniem tej funkcji. Druga funkcja czeka na sygnał, ale tylko przez czas określony parametrem `abstime`. Jeśli sygnał nie nadejdzie w tym czasie, mutex jest przejmowany z powrotem i zwracany jest błąd. `abstime` oznacza czas absolutny (systemowy) a nie relatywny, tzn. należy podac koniec oczekiwania. Czekanie na sygnał jest punktem anulowania.

Semafory

Semafory POSIX to normalne semafory uogólnione. Służą oczywiście do synchronizacji i kontroli równoległego dostępu.

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem)
```

Inicjalizuje lub usuwa semafor. Zmienna `pshared`, jeśli różna od 0 oznacza, że semafor ma być dzielony między wieloma procesami (tzn. nie tylko wątkami dzialającymi w tej samej pamięci). Zmienna `value` określa początkową wartość semafora. Wartość semafora nie może przekroczyć `SEM_VALUE_MAX` (zazwyczaj równe `INT_MAX`). LinuxThreads nie obsługuje w aktualnej wersji (glibc 2.1) semaforów dzielonych między wieloma procesami i `pshared` musi mieć wartość 0.

```
int sem_wait(sem_t *sem)
int sem_trywait(sem_t *sem)
```

Czeka, aż wartość semafora `sem` będzie większa od 0, w tym momencie budzi sie i jednocześnie ją zmniejsza o 1. Druga funkcja wychodzi z błędem jeśli wartość semafora wynosi zero, w przeciwnym przypadku ją zmniejsza. `sem_wait` jest punktem anulowania.

```
int sem_post(sem_t *sem)
```

Zwiększa wartość semafora.

```
int sem_getvalue(sem_t *sem, int *val)
```

Zapisuje wartość semafora w `val`

2.8 Zadania

1. W systemie Linux istnieje możliwość zapisu stanu pamięci procesu do specjalnego pliku `core` w momencie wystąpienia w programie np. błędu segmentacji. Możliwe jest następnie przejrzenie tego pliku i ewentualne wykrycie błędów.

   W związku z powyższym nasuwa się myśl, że przydatna też może być możliwość zrzucenia na dysk do specjalnego pliku (lub kilku plików) całej informacji o procesie (tzw. **checkpointing**): stanu pamięci, deskryptorów plików, dowiązanianń do bibliotek, itd. Dałoby to możliwość np. zatrzymania pracy procesu i wznowienia wykonywania w późniejszym czasie, gdy system będzie mniej obciążony, bez konieczności wykonywania obliczeń od nowa.

   Oczywiście niektóre informacje mogą być szczególnie ciężkie do odtworzenia, inne wręcz niemożliwe (proszę sobie wyobrazić odtwarzanie połączeń sieciowych!).

   Spróbuj napisać zestaw funkcji jądra umożliwiających realizację checkpointingu, w być może niepełnym zakresie (jednak jak największym). [11]

[11] Autor: Marcin Gajda

2. Program szeregujący procesy (**scheduler**) w przypadku braku procesów zwykłych i czasu rzeczywistego, które są w stanie `TASK_RUNNING` przydziela procesor procesowi o numerze 0 (jest to tzw. proces **idle**). Proces ten nie robi w tym czasie nic - realizuje pustą pętlę, zajmuje jedynie procesor.

   Spróbuj tak przerobić definicję tego procesu, aby wykonywał on w tym czasie jakąś pożyteczną dla systemu czynność (porządkował pamięć lub dysk, testował urządzenia zewnętrzne, liczył rozwinięcie dziesiętne liczby pi lub po prostu grał np. w życie). [12]

3. W schedulerze za wybór procesu, któremu zostanie przydzielony procesor odpowiada funkcja jądra `goodness()` zwracająca efektywny priorytet procesu. Jest ona tak zbudowana, że zawsze wybiera proces czasu rzeczywistego, jeśli taki oczekuje na procesor (jest zresztą rzecz normalna i sensowna). Spróbuj tak zmodyfikować funckję `goodness()`, by zwracała inne wartości (możesz np. tak zmiejszyć stałą dodawaną do priorytetu procesu czasu rzeczywistego, by możliwe było wywłaszczenie go przez zwykły proces o szczególnie dużym priorytecie, możesz jeszcze bardziej faworyzować proces, który wykonywał się na tym samym procesorze, możesz odebrać fory procesowi bieżącemu ...).

   Dokładnie opisz zmiany i przeprowadź testy na takim i na normalnym jądrze. Wyciągnij odpowiednie wnioski. [13]

4. Udowodnij, że wartość pola `counter` procesu nie może być większa niż `2*priority` procesu [14]

5. Dlaczego procesy czasu rzeczywistego mogą być uruchamiane tylko przez administratora systemu? [15]

6. Wyjaśnij dlaczego obsługa sygnału nie przedefiniowana przez użytkownika lub ignorowana odbywa się w trybie jądra w funkcji `do-signal` natomiast przedefiniowana w trybie użytkownika?[16]

7. Procesy oczekujące na zajście pewnego zdarzenia (takiego jak zwolnienie zasobu) umieszczane są w skojarzonej z tym zdarzeniem kolejce `wait_queue`. Wskaż sytuacje, kiedy informacje o procesie usuwane są z tej kolejki mimo, że dane zdarzenie nie zaszło. Omów mechanizmy odpowiedzialne za usunięcie z kolejki informacji o procesie w każdej z takich sytuacji.

   Źródła:

   `sched.c`, w szczególności funkcje:

   - `schedule()`
   - `schedule_timeout(...)`
   - `process_timeout(...)`

---

[12] Autor: Marcin Gajda
[13] Autor: Marcin Gajda
[14] Autor: Marek Nieżurawski
[15] Autor: Marek Nieżurawski
[16] Autor: Marek Nieżurawski

- wake_up_process(...)

signal.c, w szczególności funkcja:

- send_sig_info(...)

[17]

8. Używając funkcji opisanych w tym rozdziale, zaimplementuj mechanizm numeratora i licznika zdarzeń, czyli struktury: sequencer i eventcount oraz funkcje: int ticket(sequencer), void await(eventcount, int) i void advance(eventcount). Można założyć, że funkcje te będą wykonywane w sposób atomowy.[18]

9. Grupa procesów wykonuje następujący algorytm współbieżnie, korzystając z funkcji, które zostały podane jako rozwiązanie zadania w poprzednim podpunkcie:

```
sequencer s;
eventcount e;

while(1)
{
    sekcja_lokalna;
    await(e, ticket(s));
    sekcja_krytyczna;
    advance(e);
}
```

Co się stanie, jeżeli jeden z procesów, podczas gdy jest w stanie TASK_INTERRUPTIBLE, otrzyma sygnał SIGKILL? Jaki będzie to miało wpływ na poprawność działania algorytmu?
[19]

10. Oto fragment tabeli sygnałów:

Nazwa sygnału	Numer	Standardowa obsługa
SIGCONT	18	ignorowany
SIGSTOP	19	zatrzymanie procesu
SIGCHLD	17	ignorowany

Dane są procesy A i B. Wszystkie bity w polu signal struktury task_struct procesu A są wyzerowane. Proces B wysyła do procesu A następujące sygnały (w tej kolejności) SIGCONT, SIGSTOP,SIGCHLD. Podaj wartość pola signal struktury task_struct procesu A (które bity są zapalone) w następujących przypadkach:

(a) Proces obsługuje wszystkie sygnały standardowo (SIG_DFL)

(b) Proces ignoruje SIGCHLD (SIG_IGN), pozostałe obsługuje standardowo (SIG_DFL)

[17] Autor: Michał Kwiatek
[18] Autor: Iwona Białynicka-Birula
[19] Autor: Iwona Białynicka-Birula

(c) Proces jest śledzony

(d) Proces obługuje wszystkie sygnały standardowo (SIG_DFL), ale blokuje SIGCHLD

We wszystkich przypadkach zakładamy, że proces B ma odpowiednie prawa do wysłania sygnałów oraz, że proces A nie zdążył jeszcze odebrać żadnego z nich.[20]

11. W algorytmie wysyłania sygnału `send_sig_info` , jeśli proces-odbiorca ignoruje dany sygnał, to odpowiedni bit w polu `signal` struktury `task_struct` tego procesu nie jest ustawiany. W ten sposób funkcja `do_signal` ma potem mniej roboty – nie musi obsługiwać tego sygnału. Jednak w przypadku procesów śledzonych rezygnuje się z tej optymalizacji. Dlaczego?[21]

12. W systemie działa pięć procesów. W tabeli 2.6 przedstawione są: wartość pola `rt_priority` struktury `task_struct` każdego z procesów, czas potrzebny na wykonanie danego procesu na procesorze oraz moment pojawienia się w systemie (czyli kiedy proces zostaje dodany do kolejki procesów gotowych).

Tablica 2.6: Opis działania procesów

proces	`rt_priority`	czas procesora	czas pojawienia się
p1	6	7	0
p2	5	6	2
p3	5	4	4
p4	5	3	?
p5	4	1	6

Dodatkowo wiadomo, że:

- proces *p4* powstaje w wyniku wywołania przez proces *p3* funkcji systemowej `fork()` po 1 jednostce czasu wykonywania się tego procesu.
- proces *p2* usypia na 2 jednostki czasu po 2 jednostkach czasu wykonywania się.

Zakładamy, że system dysponuje jednym procesorem oraz, że czas potrzebny na przełączenie kontekstu, wybranie procesu do wykonywania, itp. jest zaniedbywalnie mały.
Kiedy pojawi się proces *p4*?
Przedstawić wykorzystanie procesora przez dane procesy oraz opisać jak zmienia się kolejka procesów gotowych w tym czasie. [22]

13. W systemie działają trzy procesy: A, B i C, przy czym proces C powstał w wyniku wywołania funkcji systemowej `fork()` przez proces B. Działanie procesu C jest śledzone przez proces A (zostało to zrealizowane przy pomocy funkcji systemowej `ptrace()` wywołanej z argumentem `PTRACE_ATTACH`). Zatem w strukturze `task_struct` procesu C pole `p_opptr` wskazuje na proces B, natomiast pole `p_pptr` wskazuje na proces A.
Który proces zostanie poinformowany o śmierci procesu C (tzn. może otrzymać odpowiedni

[20] Autor: Małgorzata Wrzesińska
[21] Autor: Małgorzata Wrzesińska
[22] Autor: Anna Doroś

sygnał), a który usunie proces C z tablicy procesów oraz zwolni jego strukturę `task_struct` w sytuacji, gdy:
(a) proces C kończy działanie jako pierwszy?
(b) procesy kończą działanie w kolejności: proces B, proces C, proces A?
(c) procesy kończą działanie w kolejności: proces A, proces C, proces B?
Zakładamy, że procesy A i B wykonują funkcję systemową `wait()`[23]

14. Zadanie polega na przerobieniu implementacji funkcji down_interruptible (semafory systemowe) tak, żeby funkcja ta reagowała tylko na sygnały wysyłane przez jądro. W tym celu należy przerobić implementację sygnałów tak aby niosły ze sobą informację o tym, czy sygnał został wysłany przez jądro.[24]

15. Proszę przedstawić w jakiej kolejności i przez jakie okresy czasu podanym w tabelce 2.7 procesom zostanie przyznany procesor. Załóż, że wykonywane są na jednym procesorze. [25]

Tablica 2.7: Charakterystyka procesów

Proces	`policy`	`priority`	`rt_priority`	początkowe `counter`	całkowity czas wykonania	moment startu procesu
A	FIFO	5	20	5	15	0
B	RR	5	30	5	10	5
C	RR	5	30	5	5	10
D	FIFO	5	20	5	5	25
E	OTHER	10	0	10	20	15
F	OTHER	5	0	5	10	30
G	OTHER	10	0	10	5	55

16. Scheduler[26]

    Jak zostaną przeliczone priorytety i który proces zostanie wybrany do wykonania w następującym przypadku:

    Są dwa procesy użytkownika: proces **A** jest właśnie wykonywany i wartość jego pola `counter` to 0, proces **B** ma także wartość pola `counter` 0, oba mają wartość pola `priority` równą 10. Mamy tylko jeden procesor i na początku żadnych procesów czasu rzeczywistego.

    Co będzie, gdy pojawią się też dwa procesy czasu rzeczywistego **C** i **D** - oba z `rt_priority` = 10 i `policy = SCHED_FIFO` (co będzie, gdy `policy = SCHED_RR`)?

17. Semafory systemowe [27]

[23] Autor: Anna Doroś
[24] Autor: Rafał Otto
[25] Autor: Rafał Otto
[26] Autor: Piotr Demusiak
[27] Autor: Piotr Demusiak

Jak będą się zmieniać wartości zmiennych `counter` i `waking` semafora systemowego `sem`, gdy na początku obie mają wartość 0, procesy **A** i **B** opuszczają semafor (wykonują `down(&sem)`), następnie procesy **C** i **D** go podnoszą (wykonują `up(&sem)`)?

18. Swieżo upieczony absolwent MIMUW-u dostał zadanie napisania programu, który ma działać jako proces czasu rzeczywistego. Obawia się on jednak przetestować napisany przez siebie program. W jaki sposób może zminimalizować ryzyko zablokowania systemu w razie zapętlenia się testowanego procesu czasu rzeczywistego? [28]

19. W systemie działa proces **P1**, który ma zdefiniowaną następującą obsługę sygnałów **S1**, **S2** i **S3**: **S1** obsługiwany jest przez funkcję `s1_handler()`, **S2** i **S3** są ignorowane(**SIG_IGN**). W trakcie swojego działania **P1** wywołuje funkcje `fork()` i `exec()` tworząc proces **P2**, który zmienia obsługę sygnału **S3** na funkcję `s3_handler()` i w podobny sposób wywołuje `fork()` i `exec()` tworząc **P3**.

    Jaką obsługę sygnałów **S1**, **S2** i **S3** będzie miał zdefiniowaną powstały w ten sposób proces **P3**. [29]

20. W momencie początkowym w systemie działają tylko dwa procesy **P1** i **P2** o parametrach zadanych tabelką 2.8. Korzystają one z semafora `s`, który w momencie początkowym jest w

	P1	P2
rt_priority	SCHED_OTHER	SCHED_OTHER
priority	10	12
counter	7	5

Tablica 2.8: Pola struktury `task_struct` dla procesów **P1**, **P2** w momencie startowym

stanie (`count=0, waking=0`). Jedyny obecny w systemie procesor wykonuje kolejno następujące instrukcje:

- P1: P(s)
- P2: P(s)
- idle: nop
- pojawia się proces P3
- P3: V(s) i kończy swoje działanie

gdzie V(s) - podniesienie semafora s, P(s) - opuszczenie semafora. Załóż, że wykonanie operacji P(s) zajmuje procesowi 3 takty zegara (tyle zmniejsza się dla niego `counter`).

Któremu z procesów **P1**, czy **P2** uda się przejść pod semaforem s? [30]

21. Sygnały

[28] Autor: Przemysław Pietrak
[29] Autor: Marek Rycharski
[30] Autor: Marek Rycharski

Funkcja *send_sig_info* służy do „powiadamiania” procesu o otrzymanym sygnale. Tryb powiadamiania wygląda różnie w zależności od stanu procesu i jego sposobu obsługi sygnału - może go np. blokować, ignorować. Zadanie polega na dodaniu nowego sposobu zachowania funkcji, np. niech zaistnieje możliwość zignorowania sygnału przez proces tylko określoną ilość razy, potem obsługa byłaby ustawiana na domyślną, lub niech pamiętana będzie informacja o ilości sygnałów każdego typu otrzymanych przez proces. [31]

22. **Wprowadzenie**

    W funkcji schedule należy wybrać kolejny proces, który ma być uruchomiony. Można uwzględnić następujące sytuacje które mogą zwiększać czas reakcji lu "sprawiedliwość" podziału czasu:

    - Zdarza się często, że proces korzysta z podobnych zasobów jak jego potomkowie lub przodek o ile po funkcji fork nie wykonano funkcji exec
    - Korzystnym też wydaje się zwiększenie czasu reakcji systemu w momencie uruchamiania procesu
    - Zdarzyć się może, iż użytkownik uruchomi wiele procesów które ze sobą współpracują; Należy zwiększyć prawdopodoieństwo przydzielenia procesora innemu użytkownikowi.

    Zauważyć trzeba, że punkty pierwszy i trzeci stoją ze sobą w sprzeczności.

    Polecenie

    Spróbuj wykorzystać informacje zawarte w strukturze `task_struct` by funkcja schedule uwzględniała powyższe sytuacje. W rozwiązaniu wprowadź parametry i na końcu omów swoje rozwiązanie w zależności od nich. [32]

23. Zadanie polega na zasymulowaniu działania funkcji szeregującej dla ustalonej grupy procesów z których część jest w kolejce oczekujących na jakieś zdarzenie. Wszystkie procesy należą do klasy `SCHED_OTHER`
    p1 zużyje 20 jednostek czasu; `counter== 5`; `priority== 5`;
    p2 zużyje 10 jednostek czasu; `counter== 5`; `priority== 5`;
    p3 zużyje 15 jednostek czasu; `counter== 5`; `priority== 5`;
    p4 zużyje 20 jednostek czasu; `counter== 5`; `priority== 5`;
    w kolejce oczekujacych
    p5 zużyje 10 jednostek czasu; `counter== 3`; `priority== 5`;
    p6 zużyje 10 jednostek czasu; `counter== 2`; `priority== 5`;
    p7 zużyje 15 jednostek czasu; `counter== 3`; `priority== 5`;
    p8 zużyje 15 jednostek czasu; `counter== 4`; `priority== 5`;
    Zdarzenia które zwonilą procesy z kolejek nastąpią w 8, 17, 21 i 32 jednostce czasu. Przerwania zegarowe następują co 1 jednostkę czasu. Procesy są powiązane na początku od p1 do p4 a w kolejce od p5 do p8. Można wprowadzić uproszczające założenie że dla każdych dwóch procesów p(i) i p(j) zachodzi warunek p(i)->mm != p(j)->mm. [33]

24. O przydzielenie procesora konkurują procesy. Każdy z nich posiada atrybut w postaci własnej temperatury, która podwaja się co kwant czasu. Procesor może zająć się pojedynczym procesem przez kwant czasu i obniżyć jego temperaturę do 2 stopni. Zadanie polega na obliczeniu

[31] Autor: Radosław Leśniewski
[32] Autor: Piotr Kozieradzki
[33] Autor: Piotr Kozieradzki

średniej temperatury jaką procesor jest w stanie utrzymać wśród n procesów czekających na „ochłodzenie”. Po ochłodzeniu temperatura procesu znów zaczyna się podwajać. [34]

25. Zaimplementuj funkcję systemową umożliwiającą wysłanie sygnałów do procesu oraz wszystkich jego procesów potomnych. [35]

26. **Eksperyment.** [36] Zaprojektować zestaw eksperymentów, których celem będzie oszacowanie następujących współczynników wydajności działania algorytmu szeregowania procesora:

    - **wykorzystanie procesora** - procentowa zajętość czasu procesora
    - **średnia przepustowość** - liczba procesów, które zakończyły wykonanie w jednostce czasu
    - **średni czas obrotu** - średni czas zużywany na wykonanie procesu
    - **średni czas oczekiwania** - średni czas spędzony przez proces w kolejce procesów gotowych
    - **średni reakcji** - średni czas pomiędzy zgłoszeniem żądania a uzyskaniem odpowiedzi

    Następnie wprowadzić proponownaą niżej zmianę w jądrze i ponownie przeprowadzić eksperymenty. Porównać nowe oszacowania współczynników z otrzymanymi poprzednio wynikami. Wyciągnąć wnioski.
    Proponowane modyfikacje:

    - odebebrać dodatkowe bonusy procesom, które „długo śpią” - tzn. zmienić sposób przeliczania priorytetu dynamicznego - Jakie niesie to zagrożenie?
    - przeciwnie: zwiększyć preferencje dla długo oczekujących procesów

27. **Charakter procesu.**[37] Zaprojektować eksperyment mający dawać wyobrażenie o preferencjach algorytmu szeregującego dla specyficznych procesów: typowo obliczeniowych, silnie zorientowanych na wejście-wyjście, interakcyjnych oraz wymagających CPU:

    - często i na bardzo krótkie (tzn. krótsze niż takt procesora)/długie odcinki czasu
    - rzadko i na bardzo krótkie/długie odcinki czasu

    Uzasadnić wyniki w oparciu o wiadomości o algorymie szeregowania.

28. **Wieloprocesorowość.**[38] Począwszy od wersji 2.0 *Linuksa* przewidziana jest możliwość wykorzystywania wieloprocesorowości. Odnaleźć w źródłach główne stałe, struktury i funkcje implementujące ten mechanizm. Omówić model wieloprocesorowości w rozważanej wersji jądra.

[34] Rafał Kałuża
[35] Tomasz Waleń
[36] Autor zadania: Joanna Czarnocka <jc167261@students.mimuw.edu.pl>
[37] Autor zadania: Joanna Czarnocka <jc167261@students.mimuw.edu.pl>
[38] Autor zadania: Joanna Czarnocka <jc167261@students.mimuw.edu.pl>

29. **Symulacja.**[39] Przeprowadzić (teoretyczną) symulację algorytmu szergowania dla następującego ciągu procesów:
ozn. P_i(*time,p_ real,p_ static*), gdzie:
i - kolejność zgłoszenia procesu,*time* - wymagany przez proces czas CPU w milisekundach, *p_ real* - priorytet określony przez pole `rt_priority` w `task_struct` , *p_ static* - priorytet statyczny procesu - pole `priority` jw.:
P_0(270,4,20),P_1(300,11,20),P_2(30,99,40).P_3(150,0,20),P_4(3,0,40),P_5(100,44,60),P_6(20,99,20)
Zakładamy, że kolejne procesy zgłaszają się co trzy tyknięcia, gdzie długość tyknięcia wynosi 10 milisekund. Czy można znaleźć takie uporządkowanie zgłaszania się powyższych procesów, aby łączny czas wykonania był minimalny? Jeżeli tak to spóbować podać ogólną metodę.

30. **7 plików.**[40] Program przetwarza dane z trzech plików wejściowych, zapisując wyniki do czterech plików wyjściowych. W systemie są dostępne cztery procesory (bardzo popularne rozwiązanie, choć teraz „magiczna” granica szybkiego i łatwego w zastosowaniu SMP na wspólnych zasobach przesunęła się trochę dalej).

    (a) Utworzenie ilu wątkow trybu jądra polecałbyś i dlaczego?

    (b) (dla ambitnych) Czy (i jeżeli tak to jakie) dodatkowe czynniki należałoby uwzględnić i dlaczego?

31. **Zagadka 1.** [41] Co się stanie po wykonaniu w wierszu poleceń polecenia `exec ls` ? (Odpowiedź: wykonane zostanie polecenie `ls`, ale wyniku jego działania prawdopodobnie nie zdążysz zobaczyć, ponieważ natychmiast po tym zostaniesz wylogowany.)

32. **Zagadka 1.** [42] W jaki sposób proces potomny może ustalić, czy jego proces macierzysty jeszcze działa? (Odpowiedź: Ojcem osieroconych procesów staje się proces `init`. Należy więc sprawdzić czy identyfikator procesu macierzystego jest równy 1.)

33. **Counter.** [43] Policzyć, ile maksymalnie może wynosić task->counter dla priorytetu 10.

34. **Wątki.** [44] W jaki sposób należy napisać program wielowątkowy na maszynie wieloprocesorowej (n procesorów), aby optymalnie wykonywały się:

    (a) programy wielowątkowe takie jak dodatkowy stos obsługi protokołu sieciowego

    (b) programy wielowątkowe, w których ładujemy generowane w trakcie działania programu moduły

35. Co się dzieje, gdy sygnał nie powodujący przerwania, zastopowania ani wznowienia działania programu (np. użytkowy) jest wysyłany do procesu w stanie `TASK_UNINTERRUPTIBLE`? Czy i kiedy taki proces go odbiera i obsługuje? [45]

[39] Autor zadania: Joanna Czarnocka <jc167261@students.mimuw.edu.pl>
[40] Autor zadania: Michał Gajda <korek@charybda.icm.edu.pl>
[41] Piotr Karpiuk
[42] Piotr Karpiuk
[43] Stanisław Paśko
[44] Konrad Wawruch
[45] Autor: Sebastian Łopieński

36. W różnych zadaniach pojawiał się problem: co trzeba zrobić, aby proces odbierający sygnał znał jego nadawcę i/lub wiedział, ile razy dany sygnał był wysyłany. Można jednak wyobrazić sobie sytuację, kiedy proces odbierający chciałby znać czas ostatniego wysłania danego sygnału. Umożliwiłoby to np. odbieranie sygnałów w kolejności zgodnej z ich nadawaniem. Można by też wprowadzić czas, po jakim dany sygnał traci ważność. Wysyłający określałby ten czas w momencie wysyłania sygnału. Proszę zaimplementować co najmniej "datowanie" sygnałów. [46]

37. W systemie działa 5 procesów z trybem szeregowania `SCHED_OTHER` o priorytetach (pole `priority` w strukturze procesu) odpowiednio 2, 5, 10, 6 i 13 oraz zapotrzebowaniu na procesor odpowiednio 10, 15, 16, 8 i 27 tyknięć. Procesy te nie wykonują operacji WE/WY (są np. procesami obliczeniowymi) ani nie wpływają na siebie nawzajem tzn. nie mogą się przerywać. Działają, aż do wyczerpania czasu sobie przyznanego. Podaj średni czas oczekiwania na zakończenie się tych procesów. Uwaga: procesy są jedynymi działającymi w systemie.[47]

38. W systemie działa 5 procesów czasu rzeczywistego, wszystkie w obrębie jednej kolejki koncepcyjnej (mające taką samą wartość `rt_priority`), o zapotrzebowaniu na procesor odpowiednio 17, 20, 16, 8 i 21 tyknięć oraz o priorytetach odpowiednio 4, 9, 6, 8 i 7. Jaki będzie średni czas oczekiwania na zakończenie wykonywania się tych procesów jeżeli w polu `policy` mają wartość `SCHED_RR` a jaki jeżeli `policy = SCHED_FIFO` ? Zakładamy, że procesy nie są przerywane przez inne procesy, nie wykonują operacji WE/WY oraz są jedynymi działającymi w systemie. [48]

2.9 Rozwiązania zadań

1. Rozwiązanie zadania 1.

2. Rozwiązanie zadania 2.

3. Rozwiązanie zadania 3.

4. Rozwiń równanie `counter(n) = counter(n - 1) / 2 + priority` w szereg geometryczny.

5. Algorytm szeregujący zawsze wybierze proces czasu rzeczywistego przed procesami zwykłymi.

6. Rozważ funkcję obsługi sygnału, która sięga pod adres spoza przestrzeni adresowej procesu lub wywołuje siebie nieskończenie wiele razy rekurencyjnie.

7. Możliwe są dwie takie sytuacje:

   (a) Proces jest w stanie `TASK_INTERRUPTIBLE` i otrzyma sygnał.

   (b) Proces został umieszczony w kolejce `wait_queue` w wyniku funkcji `sleep_on_timeout(p,x)` (względnie `interruptible_sleep_on_timeout(p,x)`) i upłynął czas `x`.

   Objaśnienie:

[46] Autor: Sebastian Łopieński
[47] Tomek Krajewski
[48] Tomek Krajewski

(a) Pod koniec wykonania funkcji `send_sig_info(...)` wywoływana jest funkcja `wake_up_process(...)` dla procesu odbierającego sygnał. Po ponownym przełączeniu kontekstu (w wyniku działania funkcji systemowej `schedule()`), proces sam usuwa się z kolejki `p`. Może się jednak zdarzyć, że proces otrzyma sygnał bezpośrednio przed wywołaniem funkcji `schedule()`. Jest to sprawdzane na początku tej funkcji; jeżeli rzeczywiście tak jest, to stan procesu jest zmieniany na `TASK_RUNNING` a on sam nie jest usuwany z kolejki procesów gotowych.

(b) W chwili wykonania funkcji `schedule_timeout(...)`, w systemie tworzony jest timer, którego zadaniem jest wywołanie funkcji `process_timeout(...)` po upływie czasu `x`. W wyniku działania tej funkcji, stan procesu zmieniany jest na `TASK_RUNNING`, a on sam zostaje dodany do kolejki procesów gotowych. Po ponownym przełączeniu kontekstu (w wyniku działania funkcji systemowej `schedule()`), proces sam usuwa się z kolejki `p`.

8.
```
struct sequencer
{
    int val;
};

struct eventcount
{
    int val;
    wait_queue *q;
};

int ticket(sequencer s)
{
    s.val = s.val + 1;
    return s.val;
};

void await(eventcount e, int x)
{
    while(e.val < x)
        interruptible_sleep_on(&e.q);
};

void advance(eventcount e)
{
    e.val = e.val + 1;
    wake_up(&e.q);
};
```

9. Skoro proces był w stanie `TASK_INTERRUPTIBLE`, to znaczy, że musiał wykonać operację `interruptible_sleep_on`, która jest w treści funkcji `await`. Ponieważ proces zostaje zabity właśnie wtedy, zmienia się bilans wykonanych operacji `ticket` i `advance`. Efekt tego jest taki, że od tej pory w sekcji krytycznej będzie mógł przebywać o jeden proces mniej niż

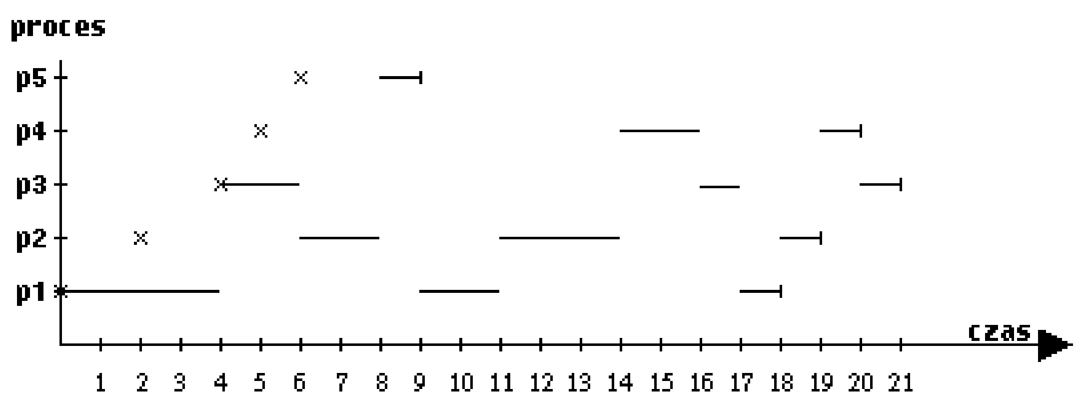
wcześniej.

10. (a) 19
    (b) 17,19
    (c) 17,19
    (d) 17,19

11. Zanim proces odbierze sygnał, śledzący go proces (debugger) może ten sygnał zablokować. Do czasu odblokowania sygnału proces może już zmienić sposób jego obsługi, np. ustawić własną procedurę obsługi. Z tego samego powodu powyższej optymalizacji nie stosuje się w przypadku sygnałów blokowanych przez proces-odbiorcę.

12. Wykorzystanie procesora przez dane procesy przedstawione zostało na rysunku 2.10 , natomiast zmiany kolejki procesów gotowych - w tabeli 2.9.

Rysunek 2.10: Wykorzystanie procesora.



Jako pierwszy zaczyna wykonywać się proces *p1*. Gdy po 2 jednostkach czasu pojawia się proces *p2* - system musi zdecydować, czy proces *p1* trzeba wywłaszczyć. W tym celu wywoływana jest funkcja `goodness()` dla obu procesów, a zwrócone przez nią wartości wyznaczają, czy należy pozwolić dalej wykonywać się procesowi *p1* (funkcja `preemption_goodness()`). Ponieważ dla obu procesów wartość pola `counter` struktury `task_struct` jest większa od 0, funkcja `goodness()` zwraca wartość `counter + priority +1`(jeśli to proces który do tej pory wykonywał się). Zatem wartości zwrócone przez funkcję `goodness()` dla procesów *p1* i *p2* wynoszą odpowiednio 4+6+1 i 5+5, a ponieważ 11>10 proces *p1* kontynuuje wykonywanie się.
W chwili 4 pojawia się proces *p3*. Tym razem wartość funkcji `goodness()` dla procesu *p1* (2+6+1) jest mniejsza od tej wartości dla *p3* (5+5). Następuje więc szeregowanie procesów, w wyniku czego proces *p3* zaczyna wykonywać się (wprawdzie *p2* i *p3* były tak samo dobre,

Tablica 2.9: Opis zmian kolejki procesów gotowych

czas	procesy w kolejności występowania w kolejce procesów gotowych
0	*idle, p1*
2	*idle, p2, p1*
4	*idle, p3, p2, p1*
5	*idle, p4, p3, p2, p1*
6	*idle, p5, p4, p3, p2, p1*
8	*idle, p5, p4, p3, p1*
9	*idle, p4, p3, p1*
10	*idle, p2, p4, p3, p1*
18	*idle, p2, p4, p3*
19	*idle, p4, p3*
20	*idle, p3*

ale *p3* występował przed *p1* na kolejce procesów gotowych - nowe procesy dostawiane są na początek kolejki).
Po 1 jednostce czasu wykonywania się proces *p3* tworzy proces *p4* (proces *p4* powstaje więc w 5 jednostce czasu). Pola `counter` obu procesów przyjmują wartość (`counter` procesu *p3*)/2. Proces *p3* jest dodawany do kolejki procesów gotowych, a następnie podejmowana jest decyzja o dalszym wykonywaniu *p3* (bo 2+5+1 dla *p3* jest większe od 2+5 dla *p4*).
W chwili 6 pojawia się proces *p5*. Wartości wyznaczone przez funkcję `goodness` dla procesów *p3* i *p5* wynoszą odpowiednio 1+5+1 i 4+4. Następuje szeregowanie procesów, w wyniku którego do wykonywania zostaje wybrany proces *p2*, który w chwili 8 usypia. Po nim działa proces *p5*.
W chwili 9 *p5* zostaje zakończony, a zaczyna działać *p1*.
W chwili 10 *p2* budzi się, a sprawdzenie wartości funkcji `goodness` (dla *p1* to 1+6+1, dla *p2* 3+5) pozwala dalej działać procesowi *p1*.
W chwili 11 pole `counter` dla procesu *p1* osiąga wartość 0, kolejnym działającym procesem jest *p2*.
W chwili 14 zaczyna działać *p4*, a w chwili 16 - *p3*.
W chwili 17 pole `counter` dla wszystkich procesów z kolejki procesów gotowych wynosi 0, następuje więc przeliczenie priorytetów dynamicznych. Dalej działać będą kolejno *p1*, *p2*, *p4* i *p3*, każdy 1 jednostkę czasu.

13. (a) Gdy proces kończy działanie (wykonywana jest funkcja `do_exit`) informowany jest o tym proces wskazywany przez pole `p_pptr` struktury `task_struct` - czyli w przypadku procesu C będzie to proces A. Następnie (w funkcji systemowej `wait()` wywołanej dla procesu A) informowany jest oryginalny proces macierzysty procesu C - czyli B. To właśnie oryginalny rodzic usuwa proces potomny z tablicy procesów oraz zwalnia jego strukturę `task_struct`, więc dla procesu C wykona te czynności proces B.
(b) Gdy proces B kończy działanie wtedy w strukturze `task_struct` procesu C pole `p_opptr` jest modyfikowane tak, aby wskazywało na proces *init* (wszystkie inne pola tej struktury po-

zostają niezmienione). Zatem gdy proces C kończy działanie informowany jest o tym najpierw proces A (gdyż jest wskazywany przez pole `p_opptr` struktury `task_struct`), a następnie oryginalny rodzic, czyli proces *init*.
Usunięcie procesu C z tablicy procesów oraz zwolnienie jego struktury `task_struct` wykona w tym przypadku proces *init*.
(c) Gdy proces A kończy działanie wtedy odpowiednie wskaźniki struktur `task_struct` procesów C, B oraz najmłodszego procesu potomnego procesu B (o ile taki istnieje) zostają przekierowane tak, aby proces C stał się najmłodszym potomkiem procesu B. Zatem procesem macierzystym dla procesu C będzie proces B, a nie proces *init* (ale wszystkie procesy, dla których proces A był oryginalnym procesem macierzystym zostatną „odziedziczone" przez proces *init*).
Gdy proces C zakończy działanie poinformowany zostanie o tym proces B. Proces B usunie również proces C z tablicy procesów oraz zwolni jego strukturę `task_struct`.

14. • Należy przerobić strukturę `task_struct` dodając do niej pole `sigset_t signalsender`. Każdy z bitów tego nowego słowa oznaczać będzie, czy sygnał został wysłany przez jądro.
    • Następnie należy przerobić funkcję `send_sig_info(...)` tak aby ustawiała odpowiedni bit nowego pola `signalsender`. Może się to wiązać także z przerobieniem funkcji: `kill()`, `kill_proc`, `kill_st` lub `kill_pg`. Należy także pamiętać o np. takiej sytuacji: jądro wysyła sygnał X do procesu A. Proces A nie zdąży odebrać tego sygnału, a proces B (nie jądro) wyśle do niego sygnał X. Wtedy tracona jest informacja, że jądro wysłało wcześniej sygnał X, a `down_interruptible` nie zareaguje. Należy dać sygnałom jądra wyższy priorytet w ten sposób, że jeżeli proces B wysyła sygnał X do procesu A, to sprawdza w polu `signal`, czy sygnał ten został przez kogoś wcześniej wysłany, a następnie sprawdza, czy tym kimś było jądro. Jeżeli to było jądro, to wysłanie sygnału przez proces B jest ignorowane.
    • Należy przerobić funkcję `signal_pending()` oraz kolejno funkcje obsługujące odbieranie sygnałów tak, aby przekazywały jeszcze informację, czy sygnał przyszedł od jądra (np. przez jakiś parametr typu `int` przekazywany przez referencję).
    • Przerobić funkcję `waking_non_zero_interruptible` tak żeby ignorowała sygnały przychodzące nie od jądra.

15. Zachowanie procesów przedstawia tabela 2.10.

16. Oba procesy będą miały nową wartość `counter` równą `priority`, czyli 10. Jednak zostanie wybrany proces **A**, ponieważ funkcja `goodness()` w jego przypadku zwróci 10+1 (1 to premia za to, że jest procesem aktualnie wykonywanym i po wybraniu właśnie jego nie trzeba będzie przełączać kontekstu, co zajmuje trochę czasu) a w przypadku procesu **B** - tylko 10.

    Jeden z procesów czasu rzeczywistego (np. **C**) otrzyma kwant czasu, ponieważ funkcja `goodness()` w jego przypadku zwróci wartość `1000+rt_priority`, czyli 1010. Gdy skończy się jego kwant czasu, znowu on otrzyma procesor (taka jest natura `SCHED_FIFO`). Dopiero gdy zakończy się (wywoła `exit()`) lub dobrowolnie odda procesor (wywoła `sched_yield()`), do głosu dojdzie proces **D**. Gdyby było `policy = SCHED_RR` w obu procesach, otrzymywałyby one na przemian procesor na kwant czasu. W obu przypadkach procesy **A** i **B** dostaną procesor dopiero, gdy procesy **C** i **D** zakończą działanie.

Tablica 2.10: Zachowanie procesów

Czas	Proces, który dostał procesor	Wydarzenie
0	A	Jest tylko jeden proces - A, na dodatek jest procesem czasu rzeczywistego. Dostaje więc procesor.
5	B	Startuje proces B. Ponieważ ma on wyższy numer kolejki koncepcyjnej (`rt_priority`) niż proces A, dlatego dochodzi do wywłaszczenia procesu A na rzecz procesu B.
10	C	Startuje proces C. Teraz sš dwa procesy czasu rzeczywistego w tej samej kolejce koncepcyjnej. Ponieważ oba te procesy działają w trybie szeregowania Round Robin, stąd proces B oddaje procesor procesowi C.
15	B	Proces C wykonuje `exit()`. Funkcja `goodness()` wybiera proces B.
20	A	Proces B wykonuje `exit()`. Kolej teraz na proces A - ostatni proces czasu rzeczywistego.
25	A	Proces A nadal ma procesor, startuje natomiast proces D, ale musi chwilowo czekać.
30	D	Proces A wykonuje `exit()`. Startuje proces F, ale musi czekać, gdyż nie jest procesem czasu rzeczywistego.
35	E	Proces D wykonuje `exit()`. Dopiero w tym momencie nie ma wśród procesów gotowych procesów czasu rzeczywistego i dostęp do procesora dostajš procesy zwykłe. Funkcja `goodness()` wybiera proces E.
45	F	Proces E oddaje procesor, gdyż wykorzystał już swój kwant czasu. Teraz procesor otrzymuje proces F.
50	E	Teraz wszystkie procesy mają pole `counter` równe 0. Potrzebne jest dynamiczne przeliczenie priorytetów. Dla obu procesów wykonywana jest operacja `counter=priority`. Teraz funkcja `goodness()` wybiera proces E.
55	E	Proces E nadal ma procesor, startuje natomiast proces G, ale musi chwilowo czekać.
60	G	Proces F wykonuje `exit()`. Funkcja `goodness()` wybiera proces G.
65	F	Proces G wykonuje `exit()`. Został już tylko jeden proces w kolejce procesów gotowych - proces F.
70		Proces F wykonuje `exit()`. Nie ma już procesów gotowych.

17. Na początku proces **A** zmniejszy wartość `counter` do -1. Po sprawdzeniu, że `counter` < 0, proces **A** zawiesi się na kolejce `wait`. Podobnie uczyni proces **B**. Wówczas `counter` $=$ -2, `waking` $= 0$.

    Proces **C** zwiększy wartość `counter` do -1, `waking` do 1, obudzi wszystkie czekające w kolejce `wait`, czyli **A** i **B**. Gdy np. proces **A** najpierw dostanie procesor, zmieni wartość `waking` na 0 i wyjdzie ze swojej funkcji `down()`. Proces **B** nie może jeszcze wyjść, bo on zastanie `waking` $= 0$. Dopiero, gdy **D** podniesie semafor i zwiększy wartość `waking` do 1, to gdy **B** dostanie procesor, będzie mógł wyjść z funkcji `down()` i zmienić `waking` na 0. W przypadku, gdy **D** zwiększy `waking` do 2, zanim **A** i **B** dostaną procesor, oba będą mogły przejść pod semaforem zaraz po dostaniu procesora.

    Jak widać na końcu wszystko jest tak, jak być powinno: `counter` $= 0$, `waking` $= 0$, **A** i **B** czekały, aż zostanie podniesiony semafor.

18. Powinien najpierw uruchomić interpretator poleceń jako proces czasu rzeczywistego, z priorytetem wyższym niż testowany proces. Pozwoli to w razie zapętlenia się testowanego procesu na „zabicie" go poleceniem kill. Testowanie powinien przeprowadzić oczywiście w środowisku tekstowym, gdyż system okienek przestanie funkcjonować w razie zapętlenia się procesu czasu rzeczywistego.

19. Rozwiązanie przedstawione jest w tablicy 2.11. Przy wywołaniu **exec()** obsługa sygnałów kopiowana jest z prcesu macierzystego do potomka z wyjątkiem własnych procedur obsługi, które zastępowane są przez **SIG_DFL**.

	S1	S2	S3
P1	s1_handler()	SIG_IGN	SIG_IGN
P2	SIG_DFL	SIG_IGN	s3_handler()
P3	**SIG_DFL**	**SIG_IGN**	**SIG_DFL**

Tablica 2.11: Obsługa sygnałów **S1**, **S2** i **S3** dla procesów **P1**, **P2** i **P3**

20. Pod semaforem przejdzie proces **P2**, ponieważ po podniesieniu semafora będzie on miał większy parametr `counter`, a zatem pierwszy zostanie wybrany przez scheduler i to jemu uda się zmniejszyć `count` semafora s. Dokładnie obrazuje to tabela 2.12. Zauważmy, że

instrukcja	P1->counter	P2->counter
start	7	5
P1: P(s)	4	5
P2: P(s)	4	2
schedule()	$(4/2) + 10$	$(2/2) + 12$
P3: V(s)	12	13

Tablica 2.12: Wartości parametrów `counter` procesów **P1** i **P2** po kolejnych instrukcjach.

kiedy **P1** i **P2** zawieszą się na semaforze, w systemie nie będzie gotowych procesów i zostanie wywołany scheduler, w wyniku czego parametry `counter` tych procesów zostaną przeliczone zgodnie ze wzorem:

```
counter = counter/2 + priority
```

21. Do struktury *sigaction* można dodać dodatkowe pole, służące jako licznik. Od jego wartości można uzależnić działanie funkcji *send_sig_info*. Należy zadbać też o to, by istniał sposób ustawienia go przez użytkownika.

22. Zmień funkcję goodness by sprawdzała czy poprzedni

```
static inline int goodness (struct task_struct * prev,
 struct task_struct * p, int this_cpu)
{
int weight;

if (p->policy != SCHED_OTHER) {
weight = 1000 + p->rt_priority;
goto out;
}

weight = p->counter;
if (!weight)
goto out;

#ifdef \_\_SMP\_\_
if (p->processor == this_cpu)
weight += PROC_CHANGE_PENALTY;
#endif

if (p->mm == prev->mm)
weight += 1;
/* jeśli po operacji fork nie była wykonana funkcja exec to
 * jest duża szansa że procesy korzystają z tych samych bibliotek
 */
if (!did_exec)
{
if ( (p->opptr==prev)||(p->cptr==prev)||
(p->ysptr==prev)||(p->osptr==prev) )
    weight += a; // pierwszy parametr
}
if (p->flags & FORKNOEXEC)
    weight += b; //drugi parametr
if (p->uid != prev->uid ) // ten sam numer użytkownika
    weight += c; //trzeci parametr

weight += p->priority;
```

```
out:
return weight;
}
```

- duża wartość parametru a może spowodować ”przejęcie” procesora przez jednego z użytkowników.
- parametr b może być duży zapewni nam to tylko szybsze uruchamianie programu.
- duża wartość parametru c nie jest zalecana gdyż może ona wpłynąć na uruchamianie procesów dotychczas oczekujacych w kolejkach procesów oczekujących, a które zostały właśnie ”obudzone”.

23. Bardzo ważne by zachować kolejność w kolejce procesów gotowych do wykonania oraz by pamiętać, że przeliczanie odbywa się dopiero gdy nie ma już procesów które mogły by się wykonać, to znaczy funkcja goodness zwraca dla wszystkich 0.
Wyniki podam w formie : proces (czas zakończenia wykonywania)
p1 (5) –> p2 (8) –> p3 (13) –> p4 (17) –> p5 (20) –> p6 (21) –> p7 (24) –> p2 (26) –> p6 (27) –> p4 (28) –> p1 (31) –> p8 (38) –> p3 (43) –> p5 (48) –> p7 (53) –> p2 (58) –> p6 (63) –> p1 (65) –> p8 (70) –> p3 (75) –> p5 (77) –> p7 (82) –> p6 (85) –> p4 (90) –> p1 (95) –> p8 (98) –> p3 (103) –> p7 (105) –> p1 (110)
Mam nadzieję Czytelniku, że uzyskałeś podobne wyniki.

24. Łatwo jest udowodnić, że istnieje taka maksymalna średnia temperatura, którą prcesor jest wstanie zapewnić. Obliczmy jaka jest minimalna średnia temperatura. Po pewnym czasie wśród procesów istnieje taki, który był ochłodzony 1 kwant czasu temu i posiada temperaturę 2 stopnie, istnieje taki który był chłodzony 2 kwanty czasu temu i posiada temperaturę 4 stopnie, tak aż do n-tego który był chłodzony n kwantów czasu temu i posiada temperaturę 2n̂. Wszystko to oczywiście jeżeli, nie istnieje jakikolwiek „zaniedbany” proces, którego temperatura przekraczałaby sumę temperatur wszystkich procesów. Stąd też minimalna średnia temperatura to

$$\frac{(2 + 2^2 + 2^3 + ... + 2^n)}{n}$$

Łatwo też zauważyć strategię jaką powinien kierować się procesor podczas wyboru procesów do ochłodzenia. Wystarczy jeżeli będzie wybierał ten o najwyższej temperaturze.

25. Można po otrzymaniu żądania wysłania sygnału przeglądnąć całe drzewo podprocesów i do każdego z osobna wysłać sygnał, innym rozwiązaniem może być “kaskadowe” wysyłanie sygnału.

26. Sugestie: przygotować program uruchamiający i usypiający w pętli ciąg procesów i mierzyć czasy wykonania poszczególnych etapów (z maksymalną dokładnością, np. co do milisekund). Zaproponowane modyfikacje w algorytmie szeregowania wymagają prostych zmian dotyczących m.in. ustalania wartości pól `counter` oraz `priority`. Wystarczą niewielkie zmiany w plikach:
    - kernel/sched.c

- kernel/fork.c

27. Dokonujemy oszacowania współczynników wymienionych w poprzednim zadaniu, zmieniając stopniowo charakter procesów - np. poprzez wstawianie do programów nowych pętli lub zwiększanie liczby obrotów pętli wykonujący specjalistyczne zadania: same operacje odczytu, pierwiastkowanie, czy też budzące się na bardzo krótki okres czasu - mozna tu użyc ciekawego chwytu z wykorzystaniem funkcji `select()`, który umożliwia uśpienie procesu na czas określony z dokładnością do milisekund:
`select(0,NULL,NULL,NULL,NULL,czas_w_mili_s)` ,
czego nie oferuje funkcja `sleep()`

28. Np.:

    - stała `NR_CPUS` - opisująca liczbę procesorów,
    - tablica `task_struct *current_set[NR_CPUS]` - procesy aktualnie wykonywane
    - pola `processor` oraz `last_processor` w strukturze zadania `task_struct` opisujące odpowiednio działający procesor oraz numer procesora, który ostatnio wykonywał to zadanie
    - funkcja `smp_init()`- uaktywniająca przy inicjalizacji jądra wszystkie pozostałe procesory, po uruchomieniu przez funkcję `start_kernel()` głównego procesora rozruchowego

    Jest to symetryczna wieloprocesorowość (SMP) - wszystkie procesy współdzielą tę samą pamięć główną oraz podsystem wejścia/wyjścia.

29. Obliczenia te, choć pouczające - są jednak mozolne. Ich wykonanie pozostawiamy dociekliwemu czytelnikowi.

30. (a) 3+4+4, Czyli 3 na wczytywanie wejścia, 4 na przetwarzanie na wszystkich procesorach i 4 na pisanie wyników do pliku. Taka ilość jest najmniejsza, która gwarantuje że jednocześnie można wysłać zadania czytania i pisania wszystkich używanych plików, a jednocześnie efektywnie wykorzystywać moc wszystkich dostępnych procesorów do przetworzenia wcześniej uzyskanych danych. (Jeżeli zostanie to dobrze uzasadnione, to użycie jednego dodatkowego wątku koordynującego może być uznane za poprawne]

    (b) Bardzo trzeba uważać, by dane w miarę równomiernie obciążały różne wątki/procesory. Jeżeli tylko wyróżniony procesor ma prawo do czytania i mamy coś data-intensive to może tylko 3 procesory przeznaczyć na obliczenia i dać 3+3+4 wątki? (wbrew pozorom często spotykane - niektórzy zauważają, że: jeżeli we/wy nie stanowi ponad 50to prawdopodobnie nadal stanowi ponad 50

31.

32.

33. 19 (przy wykorzystaniu zawsze jednego cyklu – dodaje zawsze połówkę).

34. (a) Najlepiej przy użyciu bibliotek wątków przestrzeni użytkownika, ze względu na konieczność bardzo częstych zmian kontekstu

(b) Używając wątków przestrzeni jądra – możliwa jest taka implementacja programu, że wyjątek jednego wątku nie powoduje wyjścia z całego programu (nie zawsze mamy zaufanie do ładowanych, a więc niekoniecznie własnych modułów)

35. Informacja o wysłaniu sygnału zapisywana jest w odpowiednim bicie pola `signal` struktury `task_struct`. W momencie wysyłania nic więcej się nie dzieje - proces w tym momencie nie odbiera sygnału. Natomiast gdy proces zostanie obudzony (dokładnie: przy pierwszym powrocie z funkcji systemowej), zostanie uruchomiona funkcja `do_signal`, która obsłuży sygnał.

36. Trzeba dodać do struktury `task_struct` tablicę czasów ostatniego wysłania sygnału. Czas ten mogłby się równać wartości zmiennej `jiffies` w momencie wysyłania danego sygnału - zatem być typu `unsigned long`. Należałoby w tym celu odpowiednio zmodyfikować funkcję `send_sig_info`. Aby zmienić kolejność odbierania sygnałów, trzeba z kolei zmodyfikować funkcję `do_signal` - powinna ona (dla pierwszej wersji zadania) przegladać tablicę czasów wysłania sygnałów, i zacząć od obsługiwania najstarszego z nich. Nie ma konieczności zmieniania czasu zapisanego w tablicy, gdyż przy nastepnym wysłaniu danego sygnału i tak zostanie on nadpisany.

    Aby zaimplementować mozliwość „przedawnienia" sygnału, należałoby albo ustanowiść stałe czasy ważności dla każdego sygnału, albo napisać funkcje analogiczne do grupy funkcji `kill_xxx()`, które z numerem sygnału dostawałyby także czas jego "ważności". Funkcja `send_sig_info` zamiast aktualnego czasu (wartości `jiffies`) do tablicy czasów powinna wpisywać wartość tę powiększoną o czas ważności sygnału. Natomiast funkcja `do_signal` odbierając każdy sygnał musi najpierw porównywać aktualnł wartość `jiffies` z czasem zapisanym w tablicy - jeśli `jiffies` jest większe, nie trzeba obsługiwać sygnału. W przeciwnym wypadku sygnał powinien byc odebrany i obsłużony w standardowy sposób.

37. Procesom zostaną przyznane na początku wartości pola `counter` zgodne z ich priorytetami. Pierwszeństwo mają procesy o najwyższej wartości w polu `counter` w związku z czym one będą wykonywane najpierw dopóki nie wyczerpią przyznanych im kwantów czasu (`counter = 0`). Kiedy wszystkie procesy wyczerpią swoje kwanty nastąpi dynamiczne przeliczenie priorytetów i wszystko zacznie się od nowa z tym, że procesy będą już miały odpowiednio mniejsze zapotrzebowania na procesor. Trzeci proces zakończy się jako pierwszy po 55 tyknięciach, a następnie czwarty po 57, piąty po 66, drugi po 71 i na końcu czwarty po 77. Daje to średni czas oczekiwania na zakończenie procesu 65,2 tyknięcia.

38. Jeśli chodzi o strategię FCFS (`policy = SCHED_FIFO`) to procesy będą się wykonywały do końca w kolejności przyjścia tzn. pierwszy proces zakończy się po 17 tyknięciach, drugi po 17+20=37 tyknięciach, trzeci po 53, czwarty po 61, a piąty po 82 tyknięciach. Zatem średni czas oczekiwania będzie wynosił 50. Jeśli chodzi o strategię Round Robin (`policy = SCHED_RR`) to jest ona trochę podobna do szeregowania zwykłych procesów z tą różnicą, że na następny proces wybierany jest następny w kolejce, a nie ten o najwyższym priorytecie. Natomiast ten, który właśnie oddał procesor dostaje nowy kwant zgodnie ze swoim priorytetem i jest wstawiany na koniec kolejki. Z podanych pięciu procesów jako pierwszy zakończy się proces czwarty po 27 tyknięciach, następnie procesy drugi po 66, trzeci po 70, czwarty po 77 oraz pierwszy po 82. Średni czas oczekiwania na zakończenie się procesu wynosi zatem 64,4.

Literatura

[Rochkind 93] Rochkind M. J.: *Programowanie w systemie Unix dla zaawansowanych.* Warszawa, WNT 1993.

[Stevens 95] Stevens W. R.:*Programowanie zastosowań sieciowych w systemie Unix.* Warszawa, WNT 1995.