8. Zarządzanie procesami

Wstęp

Proces pełni rolę podstawowej jednostki pracy systemu operacyjnego. System wykonuje programy użytkowników pod postacią procesów. Omówienie procesów z punktu widzenia użytkownika systemu Linux zostało zamieszczone w wykładzie 4. Bez wnikania w budowę wewnętrzną samego systemu, przedstawiliśmy tam podstawowe pojęcia oraz metody uruchamiania procesów i sterowania ich działaniem.

Wykład 8 poświęcamy omówieniu, w jaki sposób jądro systemu Linux realizuje zarządzanie procesami. Początkowo prezentujemy podstawowe zagadnienia dotyczące zarządzania procesami w dowolnym systemie operacyjnym. Następnie przedstawiamy sposób reprezentacji procesów w systemie Linux oraz podstawowe atrybuty każdego procesu. Opisujemy działanie planisty przydziału procesora. Omawiamy mechanizmy tworzenia nowych procesów, kończenia procesów oraz uruchamiania programów, a także obsługi sygnałów w procesie. Prezentujemy również funkcje systemowe realizujące wspomniane operacje na procesach.

8.1. Podstawowe zagadnienia

W wielozadaniowym systemie operacyjnym może działać jednocześnie wiele procesów. Podstawowe zadania związane z zarządzaniem tymi procesami obejmują:

- tworzenie i usuwanie procesów,
- wstrzymywanie i wznawianie procesów,
- planowanie kolejności wykonania procesów,
- dostarczanie mechanizmów synchronizacji i komunikacji procesów.

Zagadnienia te zostaną omówione w ramach wykładów 8, 12 i 13.

Procesy

Pojęcie procesu zostało wprowadzone w wykładzie 4. Przypomnijmy, że proces to wykonujący się program. Pełni on rolę podstawowej jednostki pracy systemu operacyjnego.

Kontekst procesu obejmuje zawartość logicznej przestrzeni adresowej procesu, rejestrów sprzętowych oraz struktur danych jądra związanych z procesem. W systemach UNIX i Linux można wyróżnić trzy poziomy kontekstu:

- kontekst poziomu użytkownika, na który składa się:
    - obszar kodu (instrukcji),
    - obszar danych,
    - obszar stosu,
    - obszary pamięci dzielonej,
- kontekst poziomu rejestru zawierający:
    - licznik rozkazów,
    - rejestr stanu procesora,
    - wskaźnik stosu,
    - rejestry ogólnego przeznaczenia,
- kontekst poziomu jądra zawierający:
    - struktury danych jądra opisujące proces,
    - stos jądra.

Proces zawsze wykonuje się w swoim kontekście. Jeśli proces wywoła funkcję systemową, to kod jądra realizujący tę funkcję również wykona się w kontekście bieżącego procesu.

Procesor przerywa wykonywanie kodu bieżącego procesu w sytuacji, gdy:

- wystąpi przerwanie z czasomierza informujące o wykorzystaniu przydzielonego kwantu czasu procesora,
- wystąpi dowolne przerwanie sprzętowe lub programowe (pułapka),
- proces wywoła funkcję systemową.

Zachodzi wtedy konieczność zapamiętania kontekstu przerwanego procesu, aby można było później wznowić jego wykonywanie od miejsca, w którym zostało przerwane. Czynność tę określa się zachowaniem kontekstu procesu.

Każdy proces reprezentowany jest w systemie operacyjnym przez specjalną strukturę danych, określaną zwykle jako blok kontrolny procesu. Struktura ta przechowuje atrybuty procesu takie, jak:

- aktualny stan procesu,
- informacje związane z planowaniem procesów,

- informacje o pamięci logicznej procesu,
- informacje o stanie operacji wejścia - wyjścia,
- informacje do rozliczeń,
- zawartość rejestrów procesora.

Procesy działające współbieżnie w systemie wielozadaniowym mogą współpracować lub być niezależne od siebie. Współpracujące procesy wymagają od systemu dostarczenia mechanizmów komunikowania się i synchronizacji działania. Problematyka ta zostanie przedstawiona w wykładach 12 i 13.

8.2. Planowanie procesów

W problemie planowania procesów wyróżnia się trzy zagadnienia:

1. planowanie długoterminowe (planowanie zadań),
2. planowanie krótkoterminowe (planowanie przydziału procesora),
3. planowanie średnioterminowe (wymiana procesów).

Planowanie długoterminowe polega na wyborze procesów do wykonania i załadowaniu ich do pamięci. Stosowane jest przede wszystkim w systemach wsadowych do nadzorowania stopnia wieloprogramowości. W systemach wielozadaniowych (z podziałem czasu) w zasadzie nie jest stosowane.

Planowanie krótkoterminowe polega na wyborze jednego procesu z kolejki procesów gotowych do wykonania i przydzieleniu mu dostępu do procesora. Ten typ planowania dominuje w systemach z podziałem czasu, takich jak Unix, Linux, Windows NT.

Planowanie średnioterminowe polega na okresowej wymianie procesów pomiędzy pamięcią operacyjną i pomocniczą, umożliwiając w ten sposób czasowe zmniejszenie stopnia wieloprogramowości. Stanowi pośredni etap w planowaniu procesów, często stosowany w systemach wielozadaniowych.

Planowanie przydziału procesora

Planowanie przydziału procesora nazywane jest również planowaniem krótkoterminowym lub szeregowaniem procesów. W dalszej części podręcznika nazwy te stosowane są zamiennie.

Istotą planowania krótkoterminowego jest sterowanie dostępem do procesora procesów gotowych do wykonania. Dokonując wyboru jednego procesu planista może stosować różne kryteria np.:

- wykorzystanie procesora,
- przepustowość systemu,
- średni czas cyklu przetwarzania procesów,
- średni czas oczekiwania procesów,
- średni czas odpowiedzi procesów,
- wariancja czasu odpowiedzi procesów.

Opracowano wiele różnych algorytmów planowania przydziału procesora. Najważniejsze z nich opisano w tablicy 8.1. Większość z przedstawionych tam algorytmów planowania może być zrealizowana w wersji wywłaszczającej lub niewywłaszczającej.

Wywłaszczanie procesów polega na przerywaniu wykonywania bieżącego procesu w celu dokonania ponownego wyboru procesu do wykonania. Jedynie algorytm FCFS nie może być wywłaszczający. Z kolei algorytm rotacyjny zakłada przymusowe wywłaszczenie procesu po upłynięciu przydzielonego kwantu czasu procesora.

Koncepcja wielopoziomowego planowania kolejek zakłada istnienie wielu kolejek procesów gotowych do wykonania. Procesy oczekujące na przydział procesora ustawiane są w konkretnej kolejce na podstawie ich priorytetu statycznego. Każda kolejka grupuje procesy z jednakową wartością priorytetu. Wszystkie procesy z kolejki o wyższym priorytecie statycznym muszą być wykonane wcześniej niż procesy z kolejek i niższym priorytecie. Każda kolejka ma ustaloną własną strategię szeregowania procesów. Niektóre algorytmy mogą korzystać z priorytetów dynamicznych procesów ustalanych na podstawie wykorzystania czasu procesora. Większość współczesnych systemów operacyjnych realizuje w pewien sposób koncepcję wielopoziomowego planowania kolejek.

Tablica 8.1 Charakterystyka różnych algorytmów planowania przydziału procesora

Algorytm	Charakterystyka
planowanie metodą "pierwszy zgłoszony - pierwszy obsłużony" (ang. First Came First Served - FCFS)	1. Do wykonania wybierany jest pierwszy proces z kolejki, czyli proces, który pierwszy zgłosił gotowość do wykonania.
planowanie metodą "najpierw najkrótsze zadanie" (ang. Shortest Job First - SJF)	1. Każdy proces ma określoną długość następnej fazy procesora. 2. Do wykonania wybierany jest proces, który ma najkrótszą następną fazę procesora. 3. Jest to algorytm optymalny ze względu na minimalny średni czas oczekiwania procesów. Jest bardzo trudny do zrealizowania.
planowanie priorytetowe	1. Wszystkie procesy mają przydzielone priorytety. 2. Do wykonania wybierany jest proces o najwyższym priorytecie.
planowanie rotacyjne (ang. round robin - RR)	1. Wszystkie procesy gotowe do wykonania tworzą kolejkę cykliczną. 2. Do wykonania wybierany jest pierwszy proces z kolejki. 3. Po upłynięciu przydzielonego kwantu czasu procesora proces jest przerywany i umieszczany na końcu kolejki.
wielopoziomowe planowanie kolejek	1. Procesy przydzielane są do wielu kolejek. 2. Każda kolejka ma ustalony stały priorytet. 3. Dla każdej kolejki ustalony jest odrębny algorytm planowania procesów. 4. Określona jest metoda przydziału gotowego procesu do konkretnej kolejki.
wielopoziomowe planowanie kolejek ze sprzężeniem zwrotnym	1. Procesy przydzielane są do wielu kolejek. 2. Każda kolejka ma ustalony stały priorytet. 3. Dla każdej kolejki ustalony jest odrębny algorytm planowania procesów. 4. Określona jest metoda przydziału gotowego procesu do konkretnej kolejki. 5. Określona jest metoda awansu procesu do kolejki o wyższym priorytecie. 6. Określona jest metoda degradacji procesu do kolejki o niższym priorytecie.

8.3. Reprezentacja procesów w systemie Linux

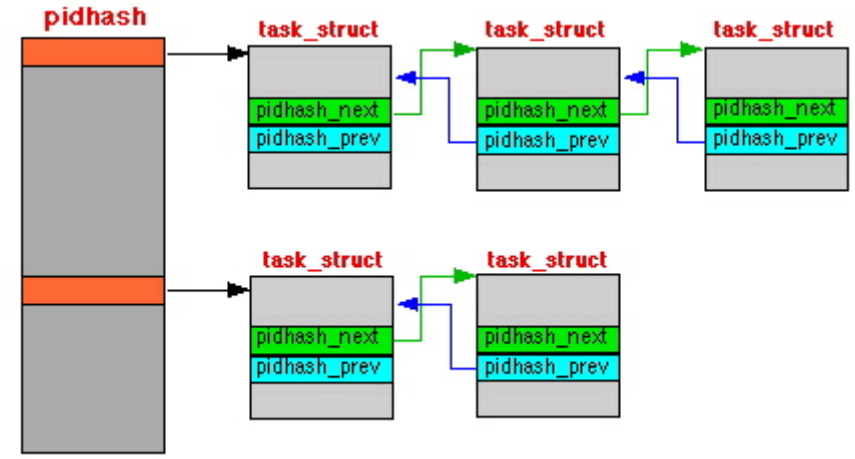
Reprezentacja procesu

Każdy proces w systemie Linux jest reprezentowany przez strukturę **task_struct**. Struktura jest dynamicznie alokowana przez jądro w czasie tworzenia procesu. Jedynym odstępstwem jest tu proces INIT_TASK o identyfikatorze PID=0, który jest tworzony statycznie w czasie startowania systemu. Struktury **task_struct** reprezentujące wszystkie procesy zorganizowane są na dwa sposoby:

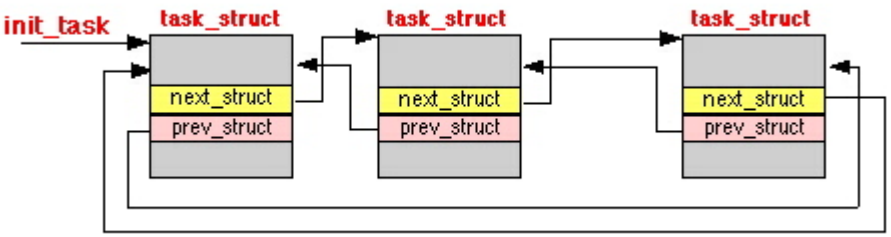
1. w postaci tablicy,
2. w postaci dwukierunkowej listy cyklicznej.

Starsze wersje jądra systemu (do wersji 2.2 włącznie) używały statycznej tablicy procesów o nazwie **task[]**, zawierającej wskaźniki do struktur **task_struct**. Rozmiar tablicy był z góry ustalony, co powodowało ograniczenie maksymalnej liczby procesów w systemie. W nowszych wersjach jądra (od wersji 2.4) stosowana jest specjalna tablica o nazwie **pidhash[]** (Rys.8.1). Jest to tzw. tablica skrótów (ang. *hash table*). Jej rozmiar jest również ograniczony, ale każda pozycja może wskazywać jeden lub listę kilku procesów. Dzięki temu liczba procesów w systemie jest ograniczona tylko rozmiarem dostępnej pamięci. Równomiernym rozmieszczaniem procesów w tablicy zajmuje się specjalna funkcja, która oblicza indeks tablicy wykonując zestaw operacji logicznych na wartości identyfikatora PID procesu. Procesy, dla których funkcja zwróci ten sam indeks tablicy, łączone są w listę. Ta sama funkcja umożliwia szybkie odnalezienie procesu w tablicy na podstawie wartości PID, unikając przeglądania wszystkich indeksów.

Dodatkowo struktury **task_struct** wszystkich procesów połączone są w dwukierunkową listę cykliczną (Rys.8.2). Wskaźniki do poprzedniego i następnego procesu z listy zapisane są jako pola **prev_task** i **next_task** w strukturze każdego procesu.



Rys. 8.1 Reprezentacja procesów w systemie Linux w postaci tablicy skrótów



Rys. 8.2 Reprezentacja procesów w systemie Linux w postaci listy dwukierunkowej

Atrybuty procesu

Atrybuty procesu zapisane w strukturze **task_struct** można podzielić na kilka kategorii:

stan procesu - stan, w którym znajduje się proces w obecnej chwili,

powiązania z procesami pokrewnymi - wskaźniki do procesu macierzystego, procesów rodzeństwa oraz do sąsiednich procesów na liście dwukierunkowej,

identyfikatory - numery identyfikatorów procesu, grupy, właściciela,

parametry szeregowania - informacje dla planisty procesów dotyczące polityki szeregowania, priorytetu i zużycia czasu procesora,

atrybuty związane z systemem plików - opis plików używanych aktualnie przez proces,

parametry pamięci wirtualnej procesu - opis odwzorowania pamięci wirtualnej procesu w pamięci fizycznej systemu,

odmierzanie czasu - informacje o czasie utworzenia procesu, wykorzystaniu czasu pracy procesora, czasie spędzonym w trybie jądra i w trybie użytkownika oraz o ustawieniach zegarów w procesie,

obsługa sygnałów - informacje o nadesłanych sygnałach oraz o blokowaniu i sposobach obsługi poszczególnych sygnałów w procesie,

kontekst poziomu rejestru - zawartość rejestrów sprzętowych procesora w chwili przerwania bieżącego procesu.

Wybrane kategorie zostaną szczegółowo omówione w dalszej części podręcznika.

Identyfikatory

Każdy proces przechowuje w strukturze **task_struct** następujące identyfikatory procesów:

pid - własny identyfikator PID,

ppid - identyfikator swojego procesu macierzystego PPID,

pgid - identyfikator grupy procesów, do której należy,

sid - identyfikator sesji, do której należy.

Wartości te można odczytać przy pomocy funkcji systemowych:

```
pid_t getpid(void);

pid_t getppid(void);

pid_t getpgrp(void);

pid_t getpgid(pid_t pid);
```

Funkcja **getpgid**() zwraca identyfikator grupy procesu wskazanego argumentem **pid**.

Wartość **pid** pozostaje oczywiście niezmienna przez cały okres działania procesu. Identyfikator **ppid** może ulec zmianie jedynie w wyniku zakończeniu procesu macierzystego i adopcji przez proces **init**. Identyfikatory grupy i sesji mogą być zmienione przez proces.

Funkcja **setpgid**() umieszcza proces wskazany przez **pid** w grupie o identyfikatorze **pgid**, przy czym bieżący proces można wskazać wartością 0.

```
int setpgid(pid_t pid, pid_t pgid);
```

Funkcja **setsid**() tworzy nową grupę i nową sesję procesów oraz mianuje proces wywołujący ich liderem.

```
pid_t setsid(void);
```

Identyfikatory użytkownika i grupy użytkowników określają uprawnienia procesu. Proces przechowuje cztery pary takich identyfikatorów:

uid gid - identyfikatory rzeczywiste,

euid egid - identyfikatory obowiązujące,

suid sgid - identyfikatory zapamiętane,

fsuid fsgid - identyfikatory dostępu do systemu plików.

Identyfikatory rzeczywiste określają właściciela procesu, czyli użytkownika, który uruchomił proces. Można je odczytać przy pomocy funkcji:

```
uid_t getuid(void);

gid_t getgid(void);
```

Identyfikatory zapamiętane służą do przechowania wartości identyfikatorów rzeczywistych na czas zmiany ich oryginalnych wartości dokonanych za pomocą funkcji systemowych. Zapamiętanie identyfikatorów następuje automatycznie i nie wymaga dodatkowych działań ze strony procesu.

Identyfikatory obowiązujące wykorzystywane są do sprawdzania uprawnień procesu i z tego względu mają największe znaczenie. Pobranie wartości identyfikatorów obowiązujących umożliwiają odpowiednio funkcje:

```
uid_t geteuid(void);

gid_t getegid(void);
```

Zazwyczaj po utworzeniu procesu identyfikatory **rzeczywiste**, **zapamiętane** i **obowiązujące** przyjmują te same wartości równe identyfikatorom właściciela procesu. W trakcie działania każdy proces może zmienić swoje identyfikatory obowiązujące. Jednak tylko procesy administratora działające z **euid** równym 0 mogą dokonać dowolnej zmiany. Procesy zwykłych użytkowników mogą jedynie ustawić takie wartości, jakie przyjmują ich identyfikatory rzeczywiste lub zapamiętane. Jedyną możliwość zmiany identyfikatorów obowiązujących w zwykłych procesach stwarza wykorzystanie specjalnych bitów w atrybucie uprawnień dostępu do pliku z programem. Jeżeli właściciel programu ustawi bit ustanowienia użytkownika **setuid** lub bit ustanowienia grupy **setgid** w prawach dostępu do pliku, to każdy użytkownik będzie mógł uruchomić program z identyfikatorem obowiązującym właściciela programu i własnym identyfikatorem rzeczywistym. W ten sposób uzyska uprawnienia właściciela programu i zachowa możliwość przywrócenia własnych uprawnień oraz przełączania się pomiędzy nimi przy pomocy funkcji systemowych.

Funkcja **setreiud**() ustawia identyfikator rzeczywisty użytkownika na **ruid**, a obowiązujący na **euid**.

```
int setreuid(uid_t ruid, uid_t euid);
```

Funkcja **setuid**() wywołana w procesie zwykłego użytkownika, ustawia identyfikator obowiązujący na **uid**. W procesie administratora funkcja ustawia wszystkie trzy identyfikatory na **uid**.

```
int setuid(uid_t uid);
```

Funkcja **seteuid**() ustawia identyfikator obowiązujący użytkownika na **euid**.

```
int seteuid(uid_t euid);
```

Funkcje **setregid**(), **setgid**() i **setegid**() działają analogicznia na identyfikatorach grupy.

```
int setregid(gid_t rgid, gid_t egid);

int setgid(gid_t gid);

int setegid(gid_t egid);
```

Identyfikatory dostępu do systemu plików stanowią jakby uboższą wersję identyfikatorów obowiązujących. Używane są tylko do sprawdzania uprawnień dostępu procesu do systemu plików. Nie dają natomiast żadnych innych uprawnień, przez co ich użycie stwarza mniejsze zagrożenie dla bezpieczeństwa procesów i całego systemu. Identyfikatory te przyjmują nowe wartości przy każdej zmianie identyfikatorów obowiązujących. Są dzęki temu praktycznie niewidoczne dla większości programów. Istnieje ponadto możliwość niezależnej zmiany ich wartości za pomocą funkcji:

```
int setfsuid(uid_t fsuid);

int setfsgid(uid_t fsgid);
```

Informacje dotyczące systemu plików

W strukturze **task_struct** procesu zdefiniowane są dwa atrybuty dotyczące plików wykorzystywanych przez proces:

files - wskaźnik do tablicy deskryptorów otwartych plików,

fs - wskaźnik do struktury zawierającej:

- maskę uprawnień do tworzonych plików **umask**,
- wskazanie na katalog główny (korzeniowy) procesu,
- wskazanie na katalog bieżący procesu.

8.4. Szeregowanie procesów w systemie Linux

W systemie Linux nie stosuje się planowania długoterminowego. Podstawowym mechanizmem jest planowanie krótkoterminowe, wykorzystujące różne algorytmy szeregowania dla różnych typów procesów.

Klasy szeregowania, wybór procesu i przełączanie kontekstu

Planista systemu Linux stosuje trzy klasy szeregowania:

- klasa procesów czasu rzeczywistego (rt_sched_class),
- klasa procesów zwykłych (fair_sched_class),
- klasa procesów idle (idle_sched_class).

Każdej klasie przypisany jest pewien priorytet, który decyduje o kolejności przeglądania klas przez planistę. Oznacza to, że wszystkie gotowe procesy z klasy o najwyższym priorytecie będą wykonywane przed procesami z klas o niższych priorytetach. Ponadto, poszczególne klasy obsługiwane są przez różne moduły planisty, implementujące różne algorytmy szeregowania procesów. Przy każdym wywołaniu planista przegląda klasy szeregowania zgodnie z malejącym priorytetem i uruchamia moduł z algorytmem szeregowania dla klasy z najwyższym priorytetem, w której są jakieś procesy gotowe do wykonania (znajdujące się w stanie TASK_RUNNING).

Planista może zostać wywołany w następujących sytuacjach:

- po zakończeniu wykonywania bieżącego procesu,
- po przejściu bieżącego procesu do stanu uśpienia w wyniku oczekiwania na operację wejścia/wyjścia,
- przed zakończeniem funkcji systemowej lub obsługi przerwania, przed powrotem procesu z trybu jądra do trybu użytkownika.

W dwóch pierwszych przypadkach uruchomienie planisty jest konieczne. W trzecim przypadku konieczność taka występuje, gdy bieżący proces wykorzystał już przydzielony czas procesora lub pojawił się proces czasu rzeczywistego o wyższym priorytecie. W przeciwnym przypadku proces może być wznowiony. Za każdym uruchomieniem planista wybiera jeden proces spośród procesów gotowych i przydziela mu czas procesora. W przypadku wyboru nowego procesu pojawia się konieczność przełączenia kontekstu. Operacja ta polega na zachowaniu kontekstu bieżącego procesu i załadowaniu kontekstu procesu wybranego przez planistę.

Algorytm planowania procesów czasu rzeczywistego

Procesy czasu rzeczywistego muszą uzyskać pierwszeństwo wykonania przez zwykłymi procesami, które działają z podziałem czasu procesora. Z tego względu otrzymują zawsze wyższe priorytety statyczne z zakresu od 0 do 99. Planista nie wylicza dla nich priorytetu dynamicznego. Procesy oczekujące na przydział procesora ustawiane są w kolejce. O kolejności ich wykonania zadecyduje wartość priorytetu statycznego, a przy równych priorytetach - strategia szeregowania. Planista czasu rzeczywistego dopuszcza dwie strategie szeregowania procesów:

SCHED_FIFO - planowanie metodą "pierwszy zgłoszony - pierwszy obsłużony" (FCFS), w którym procesy wykonywane są w kolejności zgłoszenia do kolejki bez ograniczeń czasowych,

SCHED_RR - planowanie rotacyjne, w którym procesy wykonywane są w kolejności zgłoszenia do kolejki, ale po upływie przydzielonego kwantu czasu proces zwalnia procesor i jest ustawiany na końcu kolejki.

W obydwu przypadkach wykonywany proces może być wywłaszczony przez proces z wyższym priorytetem statycznym.

Algorytm planowania procesów zwykłych

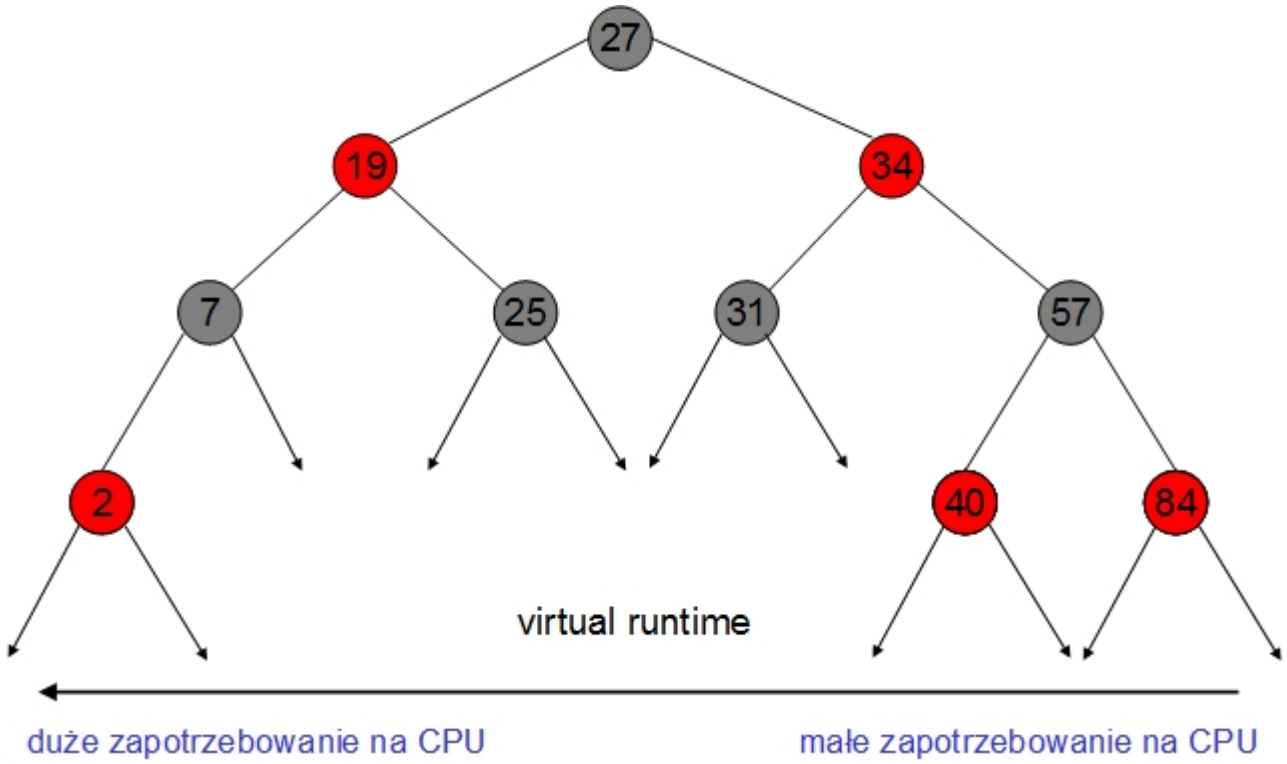
W klasie procesów zwykłych domyślną strategią szeregowania jest:

SCHED_NORMAL - planowanie metodą sprawiedliwego szeregowania procesów (CFS).
(SCHED_OTHER)

Planista CFS przydziela procesom proporcjonalny udział w czasie procesora. W ustalonym przedziale czasu wszystkie procesy gotowe dostają do wykorzystania w przybliżeniu 1/n czasu procesora, gdzie n oznacza liczbę procesów gotowych do wykonania. Czas przydzielany wybranemu procesowi jest zatem odwrotnie proporcjonalny do liczby procesów gotowych. Dodatkowo, wartości parametru **nice** procesów stosowane są jako wagi zmieniające proporcje przydziału. Algorytm wprowadza też minimalny kwant czasu procesora o długości 1 ms, aby zapobiec zbyt częstemu przełączaniu kontekstu przy bardzo dużej liczbie procesów gotowych w systemie.

Procesy gotowe do wykonania nie są ustawiane w kolejce tylko w drzewie czerwono-czarnym (drzewie RB). Indeksem procesu w drzewie jest parametr **vruntime**, czyli wirtualny czas wykonania procesu (virtual runtime), obliczany na bieżąco dla każdego procesu. Każdy węzeł drzewa reprezentuje jeden proces gotowy. Na lewo od danego węzła wstawiane są procesy o mniejszej wartości a na prawo – procesy o większej wartości parametru **vruntime**.

Planista wybiera do wykonania proces z najmniejszą wartością wirtualnego czasu wykonania, który jest reprezentowany przez lewy skrajny węzeł w drzewie RB. Przykładowy wygląd drzewa RB przewdstawia Rys. 8.3.



Rys. 8.3 Drzewo czerwono-czarne (drzewo RB) z podanymi w węzłach wartościami parametru **vruntime**

Funkcje systemowe

Funkcje systemowe umożliwiają zmianę niektórych parametrów szeregowania procesu: priorytetu **statycznego** i **dynamicznego** oraz strategii szeregowania. Uprawnienia zwykłych użytkowników są tu jednak poważnie ograniczone i sprowadzają się do obniżenia priorytetu dynamicznego własnych procesów. Procesy z uprawnieniami administratora mogą również podwyższać priorytet dynamiczny oraz zmieniać swój priorytet statyczny i strategię szeregowania.

Aktualną wartość priorytetu dynamicznego procesu można uzyskać funkcją **getpriority**(). Funkcje **nice**() i **setpriority**() umożliwiają zmianę tego priorytetu.

```
int nice(int inc);

int getpriority(int which, int who);

int setpriority(int which, int who, int prio);
```

Funkcja **sched_getparam**() odczytuje priorytet a funkcja **sched_getscheduler**() strategię szeregowania dowolnego procesu.

```
int sched_getparam(pid_t pid, struct sched_param *p);

int sched_getscheduler(pid_t pid);
```

Funkcje **sched_setparam**() i **sched_setscheduler**() pozwalają ustawić powyższe parametry szeregowania.

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

int sched_setparam(pid_t pid, const struct sched_param *p);
```

Opisane wyżej funkcje korzystają ze struktury **sched_param** do przekazania lub odczytania parametrów szeregowania:

```
struct sched_param {
    ...
    int sched_priority;
    ...
};
```

Wartość **sched_priority** traktowana jest jako priorytet statyczny lub dynamiczny w zależności od używanej strategii szeregowania.

8.5. Podstawowe operacje na procesach

Tworzenie procesu

Pierwszy proces w systemie o identyfikatorze PID = 0 zostaje utworzony przez jądro podczas inicjalizacji systemu. Wszystkie pozostałe procesy powstają jako kopie swoich procesów macierzystych w wyniku wywołania jednej z funkcji systemowych: **fork**(), **vfork**() lub **clone**().

```
pid_t fork(void);

pid_t vfork(void);
```

Jądro realizuje funcję **fork**() w następujący sposób:

- przydziela nowemu procesowi pozycję w tablicy procesów i tworzy nową strukturę **task_struct**,
- przydziela nowemu procesowi identyfikator PID,
- tworzy logiczną kopię kontekstu procesu macierzystego:
    - segment instrukcji jest współdzielony,
    - inne dane są kopiowane dopiero przy próbie modyfikacji przez proces potomny - polityka kopiowania przy zapisie (ang. copy-on-write),
- zwiększa otwartym plikom liczniki w tablicy plików i tablicy i-węzłów,
- kończy działanie funkcji w obydwu procesach zwracając następujące wartości:
    - PID potomka w procesie macierzystym,
    - 0 w procesie potomnym.

Po zakończeniu funkcji **fork**() obydwa procesy, macierzysty i potomny, wykonują ten sam kod programu. Proces potomny rozpoczyna a proces macierzysty wznawia wykonywanie od instrukcji następującej bezpośrednio po wywołaniu funkcji **fork**(). Różnica w wartościach zwracanych przez **fork**() pozwala rozróżnić obydwa procesy w programie i przeznaczyć dla nich różne fragmenty kodu. Ilustruje to poniższy przykład.

Różnica w działaniu funkcji **vfork**() polega na tym, że proces potomny współdzieli całą pamięć z procesem macierzystym. Ponadto proces macierzysty zostaje wstrzymany do momentu, gdy proces potomny wywoła funkcję **systemową _exit**() lub **execve**(), czyli zakończy się lub rozpocznie wykonywanie innego programu. Funkcja **vfork**() pozwala zatem zaoszczędzić zasoby systemowe w sytuacji, gdy nowy proces jest tworzony tylko po to, aby zaraz uruchomić w nim nowy program funkcją **execve()**.

Funkcja **clone**() udostępnia najogólniejszy interfejs do tworzenia procesów. Pozwala bowiem określić, które zasoby procesu macierzystego będą współdzielone z procesem potomnym. Głównym zastosowaniem funkcji jest implementacja wątków poziomu jądra. Bezpośrednie użycie funkcji **clone**() we własnych programach nie jest jednak zalecane. Funkcja ta jest specyficzna dla Linux-a i nie występuje w innych systemach uniksowych, co zdecydowanie ogranicza przenośność programów. Dlatego zaleca się stosowanie funkcji z bibliotek implementujących wątki np. biblioteki Linux Threads.

Przykład
Poniżej prezentowany jest kod programu implementującego typowy schemat tworzenia procesu potomnego z wykorzystaniem funkcji fork().

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
  pid_t pid;
  if ((pid = fork()) == -1)
  {
    perror("fork");
    exit(1);
  }

  if (pid == 0)
  {
    printf("Proces potomny: funkcja fork() zwrocila wartosc %d\n", pid);
    pid = getpid();
    printf("Proces potomny: PID = %d\n", pid);
    exit(0);
  }
  else
  {
    printf("Proces macierzysty: funkcja fork() zwrocila wartosc %d\n",
pid);
    pid = getpid();
    printf("Proces macierzysty: PID = %d\n", pid);
    exit(0);
  }

  return(0);
}
```

Kończenie procesu

Proces może zakończyć swoje działanie w wyniku wywołania jednej z funkcji:

```
void _exit(int status);

void exit(int status);
```

gdzie:

status - status zakończenia procesu.

Funkcja **_exit**() jest funkcją systemową, która powoduje natychmiastowe zakończenie procesu. Funkcja **exit**(), zdefiniowana w bibliotece, realizuje normalne zakończenie procesu, które obejmuje dodatkowo wywołanie wszystkich funkcji zarejestrowanych przez **atexit**() przed właściwym wywołaniem **_exit**().

```
int atexit(void (*function)(void));
```

Jeżeli nie zarejestrowano żadnych funkcji, to działanie **exit**() sprowadza się do **_exit**().

Funkcja **exit**() jest wołana domyślnie przy powrocie z funkcji **main**(), tzn. wtedy, gdy program się kończy bez jawnego wywołania funkcji.

Realizacja funkcji przez jądro wygląda następująco:

- wyłączana jest obsługa sygnałów,
- jesli proces jest przywódcą grupy, jądro wysyła sygnał zawieszenia do wszystkich procesów z grupy oraz zmienia numer grupy na 0,
- zamykane są wszystkie otwarte pliki,
- następuje zwolnienie segmentów pamięci procesu,
- stan procesu zmieniany jest na zombie,
- status wyjścia oraz sumaryczny czas wykonywania procesu i jego potomków są zapisywane w strukturze task_struct,
- wszystkie procesy potomne przekazywane są do adopcji procesowi **init**,
- sygnał śmierci potomka SIGCHLD wysyłany jest do procesu macierzystego,
- następuje przełączenie kontekstu procesu.

Pomimo zakończenia proces pozostaje w stanie zombie dopóki proces macierzysty nie odczyta jego statusu zakończenia jedną z funkcji **wait**().

Oczekiwanie na zakończenie procesu potomnego

Proces macierzysty nie traci kontaktu ze stworzonym procesem potomnym. Może wstrzymać swoje działanie w oczekiwaniu na jego zakończenie i odebrać zwrócony status. Pozwalają na to funkcje systemowe:

```
pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int flags);
```

gdzie:

status - status zakończenia procesu potomnego,

pid - identyfikator PID procesu potomnego,

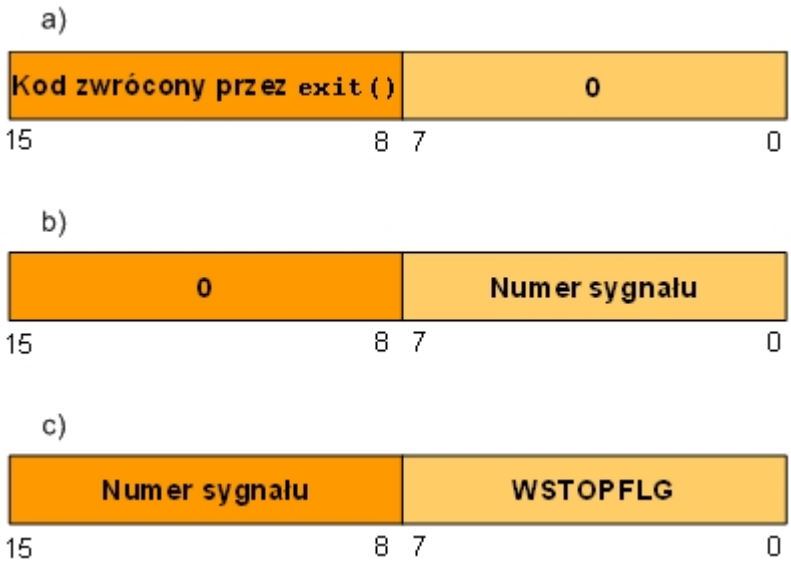
flags - flagi.

Funkcja **wait**() wstrzymuje działanie procesu dopóki nie zakończy się dowolny z jego procesów potomnych. Funkcja **waitpid**() powoduje oczekiwanie na zakończenie konkretnego potomka wskazanego przez argument **pid**.

Realizacja funkcji przez jądro obejmuje trzy przypadki:

1. Jeżeli nie ma żadnego procesu potomnego, to funkcja zwraca błąd.
2. Jeżeli nie ma potomka w stanie zombie, to proces macierzysty jest usypiany i oczekuje na sygnał śmierci potomka SIGCHLD. Reakcja procesu na sygnał jest uzależniona od ustawionego sposobu obsługi. Jeśli ustawione jest ignorowanie, to proces zostanie obudzony dopiero po zakończeniu ostatniego procesu potomnego. Przy domyślnej lub własnej obsłudze proces zostanie obudzony od razu, gdy tylko zakończy się któryś z procesów potomnych.
3. Jeżeli istnieje potomek w stanie zombie, to jądro wykonuje następujące operacje:
    - dodaje czas procesora zużyty przez potomka w strukturze **task_struct** procesu macierzystego,
    - zwalnia strukturę **task_struct** procesu potomnego,
    - zwraca identyfikator PID jako wartość funkcji oraz status zakończenia procesu potomnego.

Zwracany status jest liczbą całkowitą. Jeśli proces potomny zakończył się normalnie, to starszy bajt liczby przechowuje wartość zwróconą przez funkcję **exit**(), a młodszy bajt wartość 0. Jeśli proces został przerwany sygnałem, to w młodszym bajcie zostanie zapisany jego numer, a w starszym bajcie wartość 0. Ilustruje to Rys. 8.3.



Rys. 8.4 Status zakończenia procesu potomnego odebrany za pomocą funkcji wait() w procesie macierzystym, gdy a) proces zakończył się normalnie przez wywołanie exit(), b) proces został zakończony sygnałem, c) proces został zatrzymany sygnałem

Znając ten sposób reprezentacji, wartość odczytaną przez **wait**() można samodzielnie przesunąć o 8 bitów, jak w przykładzie poniżej, lub wykorzystać specjalne makrodefinicje do prawidłowego odczytania przyczyny zakończenia procesu oraz zwróconej wartości statusu. Szczegółowy opis można znaleźć w dokumentacji funkcji **wait**().

Przykład
Przedstawiamy kod programu, w którym proces macierzysty uruchamia 10 procesów potomnych, a następnie oczekuje na ich zakończenie i wypisuje status zakończenia każdego z potomków.

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

main(int argc, char *argv[])
{
        int i,p;
        int pid,status;

        for (i=0; i<10; i++)
        {
                if ((p=fork())==0)
                {
                        printf("PID = %d\n", getpid());
                        sleep(5);
                        exit(i);
                }
                else if (p<0)
                        perror("fork");
        }
```

```
        while ((pid=wait(&status)) > 0)
                printf("PID = %d, status = %d\n", pid, (status>>8));
        exit(0);
}
```

Uruchamianie programów

Programy wykonywane są w postaci procesów. Program do wykonania określony jest już w momencie tworzenia procesu. Jest to ten sam program, który wykonywał proces macierzysty z tym, że proces potomny kontynuje wykonywanie od miejsca, w którym wystąpiła funkcja **fork**(). Możliwość wykonania innego programu daje rodzina funkcji **exec**(). Rodzina składa się z sześciu funkcji, ale tylko jedna z nich **execve**() jest funkcją systemową.

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

gdzie:

path - pełna nazwa ścieżkowa programu,

argv[] - tablica argumentów wywołania programu,

envp[] - tablica zmiennych środowiska w postaci ciągów **zmienna=wartość**.

Pozostałe funkcje rodziny zdefiniowane zostały w bibliotece w oparciu o funkcję **execve**(). Podstawowe różnice, dotyczące sposobu wywołania i interpretacji argumentów, zebrano w tablicy 6.1.

Tablica 6.1 Charakterystyka rodziny funkcji exec()

Funkcja	Przekazywanie argumentów	Przekazywanie zmiennych środowiska	Wskazanie położenia programu
execv	tablica	zmienna globalna environ	nazwa ścieżkowa pliku
execve	tablica	tablica	nazwa ścieżkowa pliku
execvp	tablica	zmienna globalna environ	ścieżka poszukiwań PATH
execl	lista	zmienna globalna environ	nazwa ścieżkowa pliku
execle	lista	tablica	nazwa ścieżkowa pliku
execlp	lista	zmienna globalna environ	ścieżka poszukiwań PATH

Ze względu na zmienną długość, zarówno tablice jak i listy argumentów we wszystkich funkcjach muszą być zakończone wskaźnikiem NULL.

Pomimo różnic w wywołaniu, wszystkie funkcje realizują to samo zadanie. Polega ono na zastąpieniu kodu bieżącego programu w procesie kodem nowego programu. Realizacja tego zadania przebiega następująco:

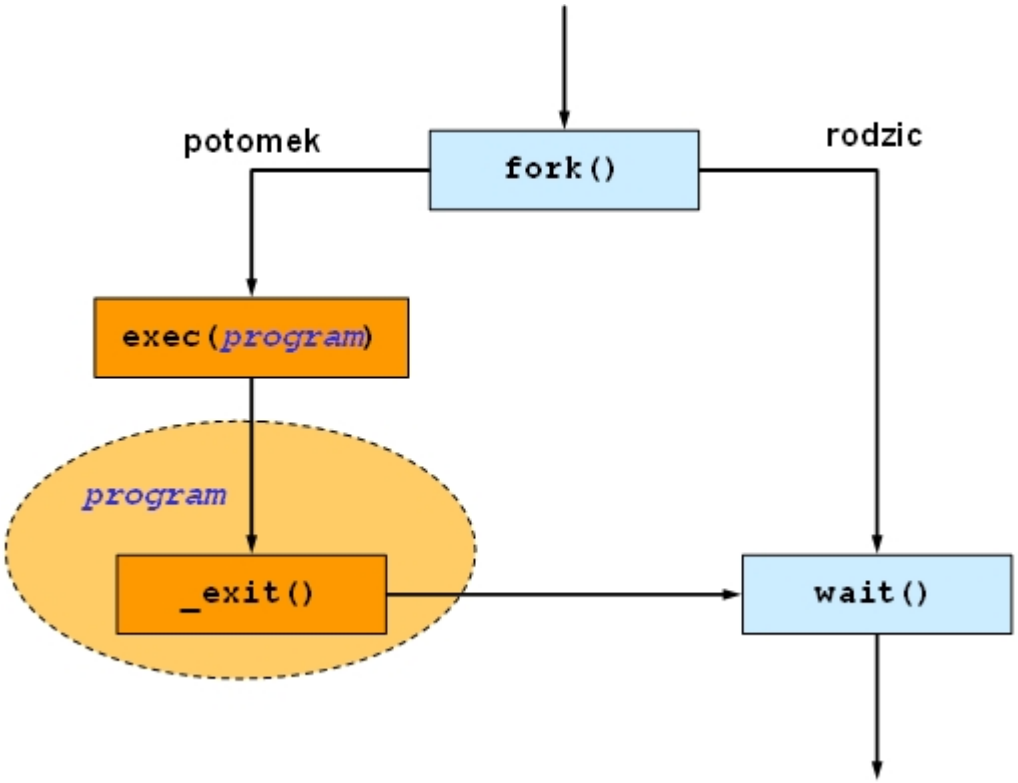
- odszukanie i-węzła pliku z programem wykonywalnym,
- kontrola możliwości uruchomienia,
- kopiowanie argumentów wywołania i środowiska,
- rozpoznanie formatu pliku wykonywalnego:
    - skrypt powłoki

    - Java
    - a.out
    - ELF
- usunięcie poprzedniego kontekstu procesu (zwolnienie segmentów pamięci kodu, danych i stosu),
- załadowanie kodu nowego programu,
- uruchomienie nowego programu (wznowienie wykonywania bieżącego procesu).

Po zakończeniu wykonywania nowego programu proces się kończy, gdyż kod starego programu został usunięty z pamięci i nie ma już możliwości powrotu.

8.6. Prosty interpreter poleceń - przykład realizacji

Zadaniem interpretera jest wykonywanie poleceń użytkownika. Jeżeli wprowadzone polecenie zostanie rozpoznane jako nazwa programu, to powłoka uruchamia ten program jako nowy proces potomny. Rys. 8.4 przedstawia schemat działania powłoki. Najpierw powłoka tworzy proces potomny wywołując funkcję **fork**() a następnie proces potomny wywołuje funkcję z rodziny **exec**() z nazwą programu do wykonania. Funkcja **exec**() zamienia w procesie dotychczasowy kod programu (czyli kod powłoki) na kod nowego programu i wznawia proces. Wykonywanie kodu kończy się funkcją **_exit**(), wywołaną jawnie lub niejawnie przez proces potomny. Jeżeli proces potomny ma działać na pierwszym planie, to proces macierzysty wywołuje funkcję z rodziny **wait**(), aby poczekać na zakończenie potomka i odebranie jego statusu zakończenia (kodu wyjścia). Jeżeli potomek działa w tle, to rodzic nie czeka na jego zakończenie, czyli nie woła funkcji **wait**().



Rys. 8.5 Schemat uruchamiania programów przez powłokę

Przykład
Poniżej prezentujemy kod programu **msh**, który jest prostym interpreterem poleceń. Powłoka **msh** analizuje wprowadzone polecenia i wykonuje je w pierwszym planie lub w tle. Ponadto, powłoka **msh** realizuje polecenie wbudowane exit(). Program pokazuje praktyczne zastosowanie funkcji systemowych fork(), execvp(), waitpid().

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main()
{
  char bufor[80];
  char* arg[10];

  int bg;

  while (1)
  {
```

```
    printf("msh $ ");
    fgets(bufor, 80, stdin);
    bg = AnalizujPolecenie(bufor, arg);

    if (arg[0] == NULL)
      continue;
    else if (strcmp(arg[0], "exit")==0)
      exit(0);
    else
      Wykonaj(arg, bg);
  }
}


int AnalizujPolecenie(char *bufor, char *arg[])
{
  int counter=0;
  int i=0, j=0;

  while (bufor[counter] != '\n')
  {
    while (bufor[counter] == ' ' || bufor[counter] == '\t') counter++;
    if (bufor[counter] != '\n')
    {
      arg[i++] = &bufor[counter];
      while (bufor[counter] != ' ' && bufor[counter] != '\t' && bu-
for[counter] != '\n') counter++;
      if (bufor[counter] != '\n') bufor[counter++] = '\0';
    }
  }
  bufor[counter] = '\0';
  arg[i]=NULL;
  if (i>0)
    while (arg[i-1][j] != '\0')
    {
      if (arg[i-1][j] == '&')
      {
        if (j == 0)
          arg[i-1] = NULL;
        else
          arg[i-1][j] = '\0';
        return 1;
      }
    j++;
  }
  return 0;
}


int Wykonaj(char **arg, int bg)
{
  pid_t pid;
  int status;

  if ((pid=fork()) == 0)
  {
    execvp(arg[0],arg);
    perror("Blad exec");
    exit(1);
```

```
  }
  else if (pid > 0)
  {
    if (bg == 0)
      waitpid(pid, &status, 0);
    return 0;
  }
  else
  {
    perror("Blad fork");
    exit(2);
  }
}
```

8.7. Obsługa sygnałów w procesach.

Atrybuty procesu

Informacje dotyczące sygnałów przechowywane są w postaci trzech atrybutów procesu (trzy pola struktury **task_struct** procesu):

signal - maska nadesłanych sygnałów w postaci liczby 32-bitowej, w której każdy sygnał jest reprezentowany przez 1 bit,

blocked - maska blokowanych sygnałów w postaci liczby 32-bitowej,

sig - wskaźnik do struktury **signal_struct** zawierającej tablicę 32 wskaźników do struktur **sigaction** opisujących sposoby obsługi poszczególnych sygnałów.

Dla każdego nadesłanego sygnału jądro ustawia odpowiadający mu bit w atrybucie **signal**. Jeżeli odbieranie sygnału jest blokowane przez proces poprzez ustawienie bitu w atrybucie **blocked**, to jądro przechowuje taki sygnał do momentu usunięcia blokady i dopiero wtedy ustawia właściwy bit. Proces okresowo sprawdza wartość atrybutu **signal** i obsługuje nadesłane sygnały w kolejności ich numerów.

Funkcje systemowe

Funkcja kill() umożliwia wysyłanie sygnałów do procesów.

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

gdzie:

pid - identyfikator PID procesu,

sig - numer lub nazwa sygnału.

Argument **pid** może przyjąć jedną z następujących wartości:

pid > 0 Sygnał wysyłany jest do procesu o identyfikatorze pid

pid = 0 Sygnał wysyłany jest do wszystkich procesów w grupie procesu wysyłającego

pid = -1 Sygnał wysyłany jest do wszystkich procesów w systemie z wyjątkiem procesu init (PID=1)

pid < -1 Sygnał wysyłany jest do wszystkich procesów w grupie o identyfikatorze pgid = -pid

Funkcje **signal**() i **sigaction**() umożliwiają zmianę domyślnego sposobu obsługi sygnałów w procesie. Pozwalają zdefiniować własną funkcję obsługi lub spowodować, że sygnały będą ignorowane. Oczywiście obowiązuje tu ograniczenie, podane w lekcji 2, że sygnały SIGKILL i SIGSTOP zawsze są obsługiwane w sposób domyślny.

Funkcja **signal**() ma zdecydowanie prostszą składnię:

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

gdzie:

signum - numer sygnału,

sighandler - sposób obsługi sygnału.

Ustawia jednak tylko jednokrotną obsługę wskazanego sygnału, ponieważ jądro przywraca ustawienie SIG_DFL przed wywołaniem funkcji obsługującej sygnał.

Argument **sighandler** może przyjąć jedną z następujących wartości:

- SIG_DFL,
- SIG_IGN,
- wskaźnik do własnej funkcji obsługi.

Ustawienie SIG_DFL oznacza domyślną obsługę przez jądro. Wartość SIG_IGN oznacza ignorowanie sygnału.

Funkcja **sigaction**() ma znacznie większe możliwości. Nie tylko zmienia obsługę sygnału na stałe (do następnej świadomej zmiany), ale również ustawia jego blokowanie na czas wykonywania funkcji obsługi. Pozwala ponadto zablokować w tym czasie inne sygnały.

```
int sigaction(int signum, const struct sigaction *act, struct sigac-
tion *oldact);
```

gdzie:

signum - numer sygnału,

act - struktura typu sigaction opisująca nowy sposób obsługi,

oldact - struktura typu sigaction przeznaczona do zapamiętania starego sposobu obsługi.

Struktura **sigaction** zdefiniowana jest następująco:

struct sigaction {

void (*sa_handler)(int);	- sposób obsługi sygnału
sigset_t sa_mask;	- maska sygnałów blokowanych na czas obsługi odebranego sygnału
int sa_flags;	- flagi

}

Funkcja **sigaction**() blokuje wybrane sygnały jedynie na czas obsługi nadesłanego sygnału. Istnieje możliwość ustawienia maski blokowanych sygnałów na dowolnie długi okres aż do ponownej zmiany przy pomocy funkcji:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

gdzie:

how - sposób zmiany maski,

set - zbiór sygnałów przeznaczony do zmiany aktualnej maski,

oldset - zbiór sygnałów przeznaczony do zapamiętania starej maski.

Argument **how** określa operację, jaka będzie wykonana na aktualnej masce blokowanych sygnałów i może przyjmować trzy wartości:

SIG_BLOCK - sygnały ustawione w zbiorze **set** są dodawane do maski,

SIG_UNBLOCK - sygnały ustawione w zbiorze **set** są usuwane z maski,

SIG_SETMASK - wyłącznie sygnały ustawione w zbiorze **set** są ustawiane w masce.

Sygnały blokowane nie są dostarczane do procesu, ale nie giną bezpowrotnie. Jądro przechowuje takie **niezałatwione** sygnały do momentu, gdy proces odblokuje ich odbieranie. Uzyskanie informacji o niezałatwionych sygnałach umożliwia funkcja:

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

Proces może zawiesić swoje działanie w oczekiwaniu na sygnał. Funkcja **pause**() usypia proces do momentu nadejścia dowolnego sygnału.

```
int pause(void);
```

Z kolei funkcja **sigsuspend**() czasowo zmienia maskę blokowanych sygnałów i oczekuje na jeden z sygnałów, które nie są blokowane. Dzięki temu umożliwia oczekiwanie na konkretny sygnał poprzez zablokowanie wszystkich pozostałych.

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

Przykład
Prezentowany poniżej program demonstruje sposób zmiany standardowej procedury obsługi sygnału SIGINT na procedurę określoną przez programistę. W tym przypadku jest to zliczenie odebranych sygnałów realizowane przez funkcję `Obsługa()`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>

void Obsluga(int);
int licznik;

int main(void)
{
  struct sigaction nowaAkcja;
  sigset_t maska;

  sigfillset(&maska);
  nowaAkcja.sa_handler = Obsluga;
  nowaAkcja.sa_mask = maska;
  if (sigaction(SIGINT, &nowaAkcja, NULL))
    perror("Blad sigaction");
&n sigdelset(&maska, SIGINT);

  while(1)
    sigsuspend(&maska);

  return(0);
}

void Obsluga(int numerSygnalu)
{
  licznik++;
  printf("Odebralem %d sygnal SIGINT (%d).\n", licznik, numerSygnalu);
}
```

9. Bibliografia

1. Silberschatz A., Galvin P.B.: Podstawy systemów operacyjnych, WNT 2000
2. Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 12.4, 13.3, 13.6)
3. Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdziały: 9, 13, 14)
4. Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdziały: 5, 9)

Słownik

Termin	Objaśnienie
kontekst procesu	zawartość wirtualnej przestrzeni adresowej procesu, rejestrów sprzętowych oraz struktur danych jądra związanych z procesem
niezałatwiony sygnał	sygnał wysłany do procesu i przechowywany przez jądro do momentu, gdy proces odblokuje jego odbieranie
planowanie przydziału procesora	wybór jednego procesu z kolejki procesów gotowych do wykonania i przydzieleniu mu dostępu do procesora

Zadania do wykładu 8

Zadanie 1

Uzupełnić program interpretera poleceń (msh.c), dodając polecenia wbudowane:

- **wait** - czekaj na zakończenie dowolnego z działających procesów drugoplanowych i wypisz kod wyjścia na terminalu (stdout),
- **exec program** - wykonaj program w bieżącym procesie,
- **kill sig pid** - wyślij sygnał o numerze **sig** do procesu o identyfikatorze **pid.**

Zadanie 2

Napisz program, w którym proces macierzysty uruchamia kilka procesów potomnych wykonujących identyczne obliczenia (np. zliczanie w pętli). Gdy jeden z procesów potomnych zakończy swoje obliczenia, pozostałe też muszą przerwać i wypisać informację o stanie swoich obliczeń. Proces macierzysty oczekuje na zakończenie wszystkich potomków i kończy działanie.