

Procesy

Procesy umożliwiają w systemach operacyjnych (quasi)równoległe wykonywanie wielu zadań. Mogą być wykonywane naprzemiennie i/lub jednocześnie.

Zatem system operacyjny musi to (quasi)równoległe wykonywanie zapewnić.

Poza tym musi umożliwić procesom pewne operacje, których nie mogą one sobie zapewnić same, np. dostęp do globalnych zasobów systemu jak port komunikacyjny.

Idąc dalej, systemy operacyjne mogą również dostarczać procesom dalszych usług, jak np. komunikacja międzyprocesowa, a także pewnych funkcji synchronizacji, jak blokady, semafony, itp.

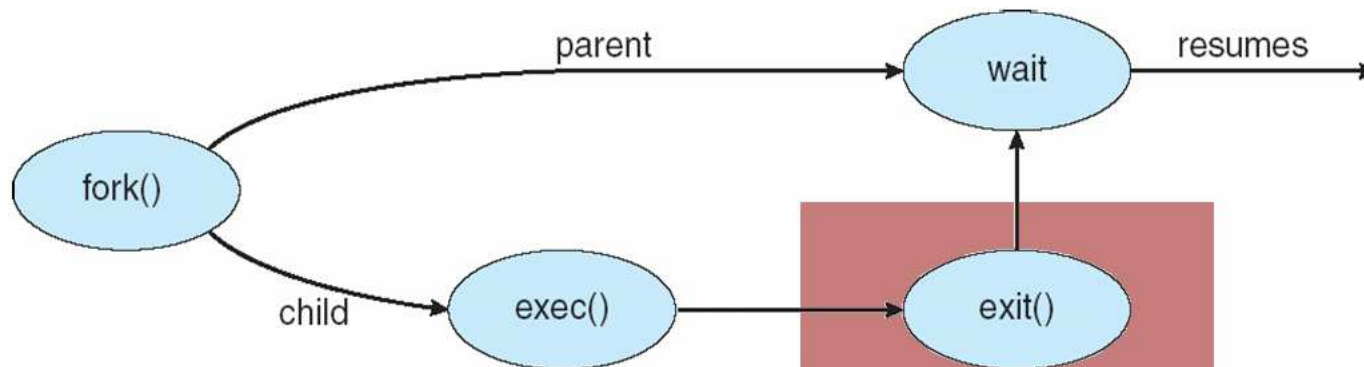
Model procesów systemu UNIX (standard POSIX)

- Proces uniksowy:
 - kod programu w pamięci (obszar programu i danych)
 - **środowisko** (zestaw zmiennych i przypisanych im wartości)
 - zestaw dalszych informacji utrzymywanych przez jądro Uniksa o wszystkich istniejących procesach, m.in. tablicę otwartych plików, maskę sygnałów, priorytet, i in.
- Funkcja `main()` wywoływana przez procedurę startową z następującymi argumentami:
 - liczbą argumentów wywołania: `int argc`
 - wektorem argumentów wywołania: `char *argv[]`
 - środowiskiem: `char **environ`
rzadko używanym ponieważ dostęp do środowiska możliwy jest również poprzez funkcje: `getenv()/putenv()`
- Proces charakteryzują: `pid`, `ppid`, `pgrp`, `uid`, `eid`, `gid`, `egid`.
Wartości te można uzyskać funkcjami `get...()`

Tworzenie procesów

- Procesy tworzone są przez klonowanie istniejącego procesu funkcją `fork()`, zatem każdy proces jest podprocesem (*child process*) jakiegoś innego procesu nadrzędnego, albo inaczej rodzicielskiego (*parent process*).
- Jedynym wyjątkiem jest proces numer 1 — **init** — tworzony przez jądro Uniksa w chwili startu systemu. **Wszystkie inne procesy w systemie są bliższymi lub dalszymi potomkami inita.**
- Podproces dziedziczy i/lub współdzieli pewne atrybuty i zasoby procesu nadrzędnego, a gdy kończy pracę, Unix zatrzymuje go do czasu odebrania przez proces nadrzędny statusu podprocesu (wartości zakończenia).

Tworzenie procesów — funkcja fork



```
switch (pid = fork()) {
  case -1:
    fprintf(stderr, "Bład, nieudane fork, errno=%d\n", errno);
    exit(-1);

  case 0:
    /* jestem potomkiem, teraz sie przeobraze ... */
    execlp("program", "program", NULL);
    fprintf(stderr, "Bład, nieudane execlp, errno=%d\n", errno);
    _exit(0);

  default:
    /* jestem szczesliwym rodzicem ... */
    fprintf(stderr, "Sukces fork, pid potomka = %d\n", pid);
    wait(NULL);
}
```

Tworzenie procesów — funkcja `fork` (cd.)

- Funkcja `fork()` tworzy nowy podproces, który jest kopią procesu macierzystego.
- Od momentu utworzenia oba procesy pracują równolegle i kontynuują wykonywanie tego samego programu. Jednak funkcja `fork()` zwraca w każdym z nich inną wartość. Poza tym różnią się identyfikatorem procesu PID (rodzic zachowuje oryginalny PID, a potomek otrzymuje nowy).
- Po sklonowaniu się podproces może wywołać jedną z funkcji grupy `exec` i rozpocząć w ten sposób wykonywanie innego programu.
- Po sklonowaniu się oba procesy współdzielą dostęp do plików otwartych przed sklonowaniem, w tym do terminala (plików `stdin`, `stdout`, `stderr`).
- Funkcja `fork()` jest jedyną metodą tworzenia nowych procesów w systemach uniksowych.

Własności procesu i podprocesu

Dziedziczone (podproces posiada kopię atrybutu procesu):

- real i effective UID i GID, proces group ID (PGRP)
- kartoteka bieżąca i `root`
- środowisko
- maska tworzenia plików (`umask`)
- maska sygnałów i handlery
- ograniczenia zasobów

Wspólne (podproces współdzieli atrybut/zasób z procesem):

- terminal i sesja
- otwarte pliki (deskrytory)
- otwarte wspólne obszary pamięci

Różne:

- PID, PPID
- wartość zwracana z funkcji `fork`
- blokady plików nie są dziedziczone
- ustawiony alarm procesu nadrzędnego jest kasowany dla podprocesu
- zbiór wysłanych (ale nieodebranych) sygnałów jest kasowany dla podprocesu

Atrybuty procesu, które nie zmieniają się po `exec`:

- PID, PPID, UID(real), GID(real), PGID
- terminal, sesja
- ustawiony alarm z bieżącym czasem
- kartoteka bieżąca i `root`
- maska tworzenia plików (`umask`)
- maska sygnałów i oczekujące (nieodebrane) sygnały
- blokady plików
- ograniczenia zasobów

Kończenie pracy procesów

- Zakończenie procesu uniksowego może być **normalne**, wywołane przez zakończenie funkcji `main()`, lub funkcję `exit()`. Wtedy:
 - następuje wywołanie wszystkich handlerów zarejestrowanych przez funkcję `atexit()`,
 - następuje zakończenie wszystkich operacji wejścia/wyjścia procesu i zamknięcie otwartych plików,
 - można spowodować normalne zakończenie procesu bez kończenia operacji wejścia/wyjścia ani wywoływania handlerów `atexit`, przez wywołanie funkcji `_exit()`.
- Zakończenie procesu może też być **anormalne** (ang. *abnormal*), przez wywołanie funkcji `abort()`, lub otrzymanie sygnału (funkcje `kill()`, `raise()`).

Status procesu

- W chwili zakończenia pracy proces generuje kod zakończenia, tzw. **status**, który jest wartością zwróconą z funkcji `main()` albo `exit()`. W przypadku śmierci przez otrzymanie sygnału system generuje dla procesu status o wartości $128 + \text{nr_sygnału}$.
- Rodzic normalnie powinien po śmierci potomka odczytać jego status wywołując funkcję systemową `wait` (lub `waitpid()`). Aby to ułatwić (w nowszych systemach) w chwili śmierci potomka jego rodzic otrzymuje sygnał `SIGCLD` (domyślnie ignorowany).
- Gdy proces nadrzędny żyje w chwili zakończenia pracy potomka, i nie wywołuje funkcji `wait()`, to **potomek pozostaje (na czas nieograniczony) w stanie zwanym zombie**. Może to być przyczyną wyczerpania jakichś zasobów systemu, np. zapelnienia tablicy procesów, otwartych plików, itp.
- Istnieje mechanizm adopcji polegający na tym, że **procesy, których rodzic zginął przed nimi (sieroty), zostają adoptowane przez proces numer 1, init**. Init jest dobrym rodzicem (choć przybranym) i wywołuje okresowo funkcję `wait()` aby umożliwić poprawne zakończenie swoich potomków.

Grupy procesów

- **Grupa procesów:** wszystkie podprocesy uruchomione przez jeden proces nadrzędny. Każdy proces w chwili utworzenia automatycznie należy do grupy procesów swojego rodzica.
- Pod pewnymi względami **przynależność do grupy procesów jest podobna do przynależności do partii politycznych**. Każdy proces może:
 - założyć nową grupę procesów (o numerze równym swojemu PID),
 - wstąpić do dowolnej innej grupy procesów,
 - włączyć dowolny ze swoich podprocesów do dowolnej grupy procesów (ale tylko dopóki podproces wykonuje kod rodzica).
- Grupy procesów mają głównie znaczenie przy wysyłaniu sygnałów.

Zasoby procesu

System operacyjny może i powinien ograniczać wielkość zasobów zużywanych przez procesy, w celu realizacji swoich funkcji zapewnienia sprawnej i efektywnej pracy.

Systemy uniksowe definiują szereg zasobów które mogą być kontrolowane. Na poziomie interpretera poleceń ograniczenia można ustawiać poleceniem `limit` (C-shell) lub `ulimit` (Bourne shell). Programowo do kontrolowania zasobów służą funkcje: `getrlimit()/setrlimit()`.

ulimit	limit	nazwa zasobu	opis zasobu
-c	coredumpsize	RLIMIT_CORE	plik core (zrzut obrazu pamięci) w kB
-d	datasize	RLIMIT_DATA	segment danych procesu w kB
-f	filesize	RLIMIT_FSIZE	wielkość tworzonego pliku w kB
-l	memorylocked	RLIMIT_MEMLOCK	obszar, który można zablokować w pamięci w kB
-m	memoryuse	RLIMIT_RSS	zbiór rezydentny w pamięci w kB
-n	descriptors	RLIMIT_NOFILE	liczba deskryptorów plików (otwartych plików)
-s	stacksize	RLIMIT_STACK	wielkość stosu w kB
-t	cputime	RLIMIT_CPU	wykorzystanie czasu CPU w sekundach
-u	maxproc	RLIMIT_NPROC	liczba procesów użytkownika
-v	vmemoryuse	RLIMIT_VMEM	pamięć wirtualna dostępna dla procesu w kB

Dla każdego zasobu istnieją dwa ograniczenia: miękkie i twarde. Zawsze aktualnie obowiązujące jest ograniczenie miękkie, i użytkownik może je dowolnie zmieniać, lecz tylko do maksymalnej wartości ograniczenia twardego. Ograniczenia twarde można również zmieniać, lecz **procesy zwykłych użytkowników mogą je jedynie zmniejszać**. Ograniczenia zasobów są dziedziczone przez podprocesy.

Priorytety procesów

W zakresie dostępu do procesora **obowiązuje podstawowa zasada równości (sprawiedliwości), zapewniająca quasi-równoległe wykonywanie procesów.**

Procesy uniksowe mają również **priorytety** określające kolejność ich wykonywania w procesorze. Priorytety mają znaczenie nadrzędne nad zasadą równości. To znaczy, **proces o niższym priorytecie zawsze musi czekać na zakończenie lub wstrzymanie procesu o priorytecie wyższym.**

Wykonywanie procesów o równych priorytetach realizowane jest zgodnie z zasadą równości.

Dynamiczna modyfikacja priorytetów

System realizuje algorytm kolejowania wykonywania procesów, zwany **planistą** (*scheduler*). Planista uniksowy opiera się na priorytetach procesów, ale jednocześnie może je zmieniać. Ogólnie, procesy interakcyjne zyskują wyższe priorytety, ponieważ mają związek z pracą człowieka, który nie chce czekać na reakcję komputera, ale jednocześnie pracuje z przerwami, które zwykle pozwalają na wykonywanie innych procesów.

Wszelkie ingerencje w pracę planisty są zaawansowanymi operacjami, dla zwykłych użytkowników trudnymi i niebezpiecznymi. Jednak w systemach uniksowych istnieje prosty mechanizm pozwalający użytkownikowi wspomagać planistę w wyznaczaniu priorytetów procesów. Tym mechanizmem są liczby **nice** dodawane do priorytetów. Użytkownik może uruchamiać procesy z zadaną liczbą nice (służy do tego polecenie **nice**), lub zmieniać liczbę nice wykonujących się procesów (polecenie **renice**). Co prawda **zwykli użytkownicy mogą tylko zwiększać domyślną wartość nice co obniża priorytet procesu**. Jednak uruchamiając procesy „obliczeniowe” z priorytetem obniżonym można efektywnie jakby zwiększyć priorytet procesów „interakcyjnych”.

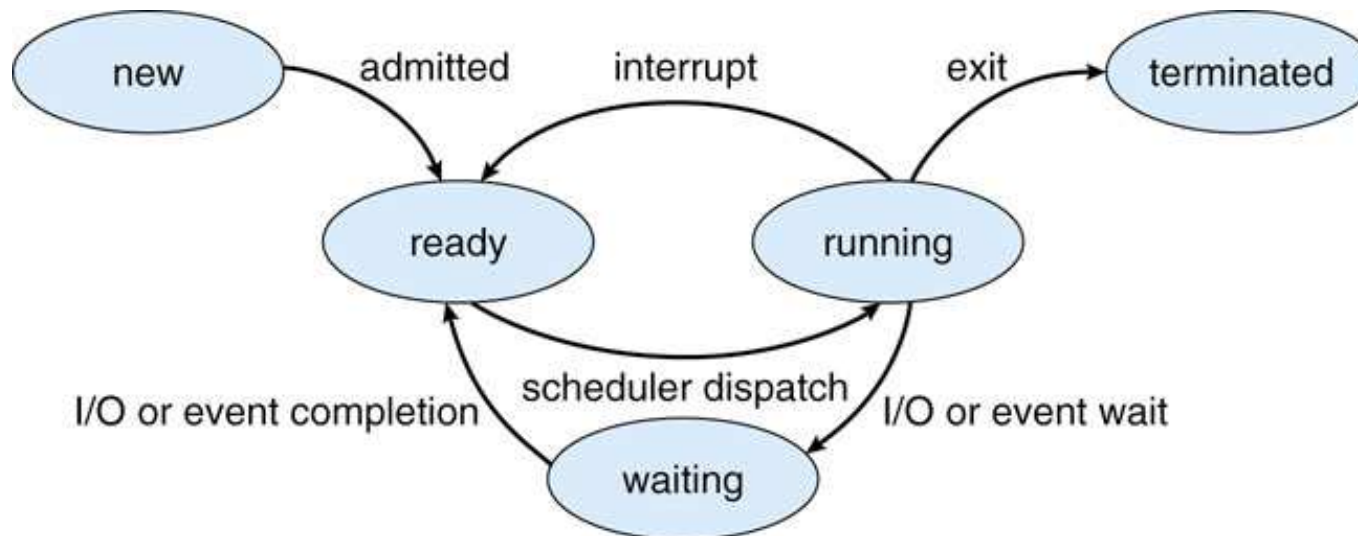
Stany procesów

wykonywalny (runnable) — proces gotowy do wykonania, może być:

wykonywany (running) — wykonujący się na procesorze

gotowy (ready) — oczekujący w kolejce na wykonanie

oczekujący/uśpiony (waiting/sleeping) — proces na coś czeka, np. na dane do przeczytania, na sygnał, na dostęp do zasobu, lub dobrowolnie zawiesił się

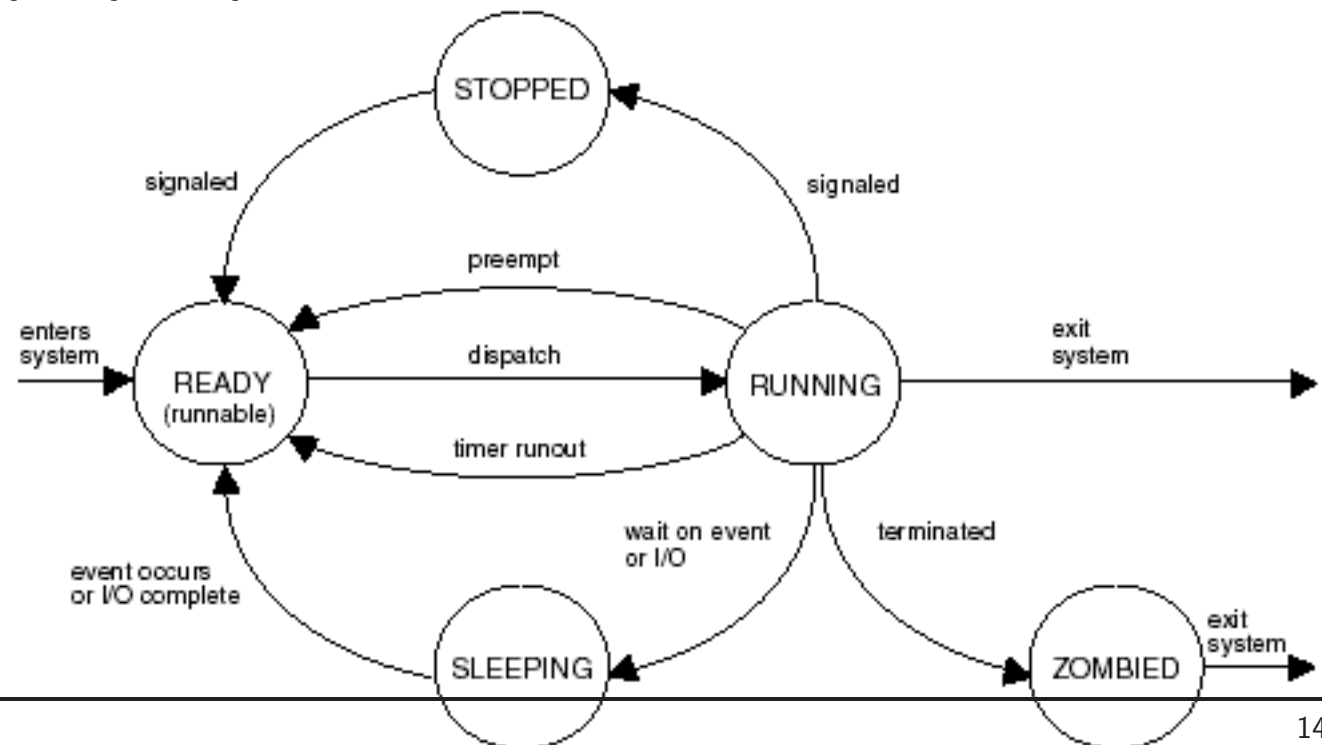


System przenosi proces w stan uśpionia jeśli nie może mu czegoś dostarczyć, np. dostępu do pliku, danych, itp. W stanie uśpionia proces nie zużywa czasu procesora, i jest *budzony* i wznawiany w sposób przez siebie niezauważony.

zatrzymany (stopped) — proces gotowy do wykonywania lecz zatrzymany na skutek otrzymania sygnału **SIGSTOP** lub **SIGTSTP**, może to być skutek: dobrowolnego żądania procesu, wysłania sygnału (np. przez użytkownika), lub odwołania się procesu pracującego w tle do terminala sterującego; wznowienie pracy procesu następuje po otrzymaniu sygnału **SIGCONT**

nieusuwalny (zombie) — proces nie może się zakończyć

wymieciony (swapped out) — proces usunięty okresowo z kolejki procesów gotowych do wykonywania wskutek działania algorytmu obsługi pamięci wirtualnej; stan wymiecenia nie jest właściwym stanem procesu — pozostaje on w jednym z powyższych właściwych stanów; wymiecenie procesu jest skutkiem braku dostatecznej ilości pamięci fizycznej na jednoczesne skuteczne wykonywanie wszystkich procesów wykonywalnych



Konta użytkownika

Konta użytkownika istnieją dla zapewnienia izolacji i zabezpieczenia działających procesów. W systemach uniksowych istnieje jedno główne konto użytkownika o numerze 0 zwane **root**. Główne programy zapewniające funkcjonowanie systemu, i uruchamiane przez jądro, działają w ramach konta użytkownika **root**.

Może istnieć dowolna liczba innych kont, zarówno dla ludzi-użytkowników systemu, jak i dla określonych programów lub ich grup, których uruchamianie wymaga izolacji lub zabezpieczenia przez innymi użytkownikami.

Konto użytkownika jest chronione hasłem, które system przechowuje w postaci **zahaszowanej**, tzn. zaszyfrowanej w sposób nieodwracalny. Weryfikacja hasła jest możliwa przez zahaszowanie podanego przez użytkownika stringa i porównanie z hasłem posiadanym przez system. Nie jest natomiast możliwe odtworzenie wersji oryginalnej hasła.

Z kontem użytkownika jest związany katalog dyskowy, do którego wyłączone prawo ma użytkownik. Jest możliwe przyznawanie praw dostępu procesom innych użytkowników. Poza tym istnieje szereg dalszych ustawień związanych z kontem użytkownika, pozwalających konfigurować jego uprawnienia, parametry pracy, itp.

Konta użytkownika — zabezpieczenia

Procesy jednego użytkownika nie mogą ingerować w procesy innego użytkownika. To znaczy, nie mogą ich zatrzymywać, wysyłać im sygnałów, czytać/zapisywać ich pamięci, zmieniać ich ustawień, zasobów, priorytetów, tworzyć ani usuwać blokad, itp. Nie dotyczy to konta użytkownika `root`, który jest użytkownikiem uprzywilejowanym, i jego procesy mogą ingerować w procesy innych użytkowników systemu.

Procesy różnych użytkowników mogą brać udział w komunikacji międzyprocesowej (np. wysyłać sobie komunikaty), o ile dobrowolnie taką komunikację podejmą.

Warto dodać, że mechanizm zabezpieczeń systemu Unix, składający się ze zbioru zwykłych użytkowników, których procesy są przed sobą nawzajem zabezpieczone, oraz użytkownika `root`, przed którym nic nie jest zabezpieczone, **jest uproszczony, dość ubogi i często niewystarczający**. Występują sytuacje pośrednie, kiedy proces potrzebuje pewnych zwiększonych uprawnień, ale nie powinien posiadać nieograniczonej władzy w systemie. Ten problem jest rozwiązywany przez szereg dodatkowych mechanizmów.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jaka jest rola procesów w systemach operacyjnych?
2. W jaki sposób tworzone są procesy w modelu Unix/POSIX?
3. Co to jest status procesu, jak jest tworzony, i co się z nim potem dzieje?
4. Jakie związki istnieją pomiędzy procesem rodzicielskim a potomkiem?
5. W jakim celu ustawiane są ograniczenia zasobów procesu?
6. W jaki sposób na priorytet procesu wpływa ustawienie liczby nice?
7. Jakie mechanizmy lub zjawiska powodują przechodzenie procesu między stanami: wykonywalnym i uśpionym?

Sygnaly

- Sygnaly są mechanizmem asynchronicznego powiadamiania procesów przez system o błędach i innych zdarzeniach.
- **Domyślną reakcją procesu na otrzymanie większości sygnałów jest natychmiastowe zakończenie pracy, czyli śmierć procesu.** Oryginalnie sygnały miały służyć do sygnalizowania błędów i sytuacji nie dających się skorygować. Niektóre sygnały powodują również, tuż przed uśmierceniem procesu, wykonanie zrzutu jego obrazu pamięci do pliku dyskowego o nazwie `core`.
- **Proces może zadeklarować własną procedurę obsługi, ignorowanie sygnału, albo czasowe wstrzymanie otrzymania sygnału** (z wyjątkiem sygnałów `SIGKILL` i `SIGSTOP`, których nie można przechwycić, ignorować, ani wstrzymać). Proces może również przywrócić reakcję domyślną.
- W trakcie wieloletniego rozwoju systemów uniksowych mechanizm sygnałów wykorzystywano do wielu innych celów, i wiele sygnałów nie jest już w ogóle związanych z błędami. Zatem niektóre sygnały mają inne reakcje domyślne, na przykład są domyślnie ignorowane.

nr	nazwa	domyślnie	zdarzenie
1	SIGHUP	śmierć	rozłączenie terminala sterującego
2	SIGINT	śmierć	przerwanie z klawiatury (zwykle: Ctrl-C)
3	SIGQUIT	zrzut	przerwanie z klawiatury (zwykle: Ctrl-\)
4	SIGILL	zrzut	nielegalna instrukcja
5	SIGTRAP	zrzut	zatrzymanie w punkcie kontrolnym (breakpoint)
6	SIGABRT	zrzut	sygnał generowany przez funkcję abort
8	SIGFPE	zrzut	nadmiar zmiennoprzecinkowy
9	SIGKILL	śmierć	bezwarunkowe uśmiercenie procesu
10	SIGBUS	zrzut	błąd dostępu do pamięci
11	SIGSEGV	zrzut	niepoprawne odwołanie do pamięci
12	SIGSYS	zrzut	błąd wywołania funkcji systemowej
13	SIGPIPE	śmierć	błąd potoku: zapis do potoku bez odbiorcy
14	SIGALRM	śmierć	sygnał budzika (timera)
15	SIGTERM	śmierć	zakończenie procesu
16	SIGUSR1	śmierć	sygnał użytkownika
17	SIGUSR2	śmierć	sygnał użytkownika
18	SIGCHLD	ignorowany	zmiana stanu podprocesu (zatrzymany lub zakończony)
19	SIGPWR	ignorowany	przerwane zasilanie lub restart
20	SIGWINCH	ignorowany	zmiana rozmiaru okna
21	SIGURG	ignorowany	priorytetowe zdarzenie na gniazdku
22	SIGPOLL	śmierć	zdarzenie dotyczące deskryptora pliku
23	SIGSTOP	zatrzymanie	zatrzymanie procesu
24	SIGTSTP	zatrzymanie	zatrzymanie procesu przy dostępie do terminala
25	SIGCONT	ignorowany	kontynuacja procesu
26	SIGTTIN	zatrzymanie	zatrzymanie na próbie odczytu z terminala
27	SIGTTOU	zatrzymanie	zatrzymanie na próbie zapisu na terminalu
30	SIGXCPU	zrzut	przekroczenie limitu CPU
31	SIGXFSZ	zrzut	przekroczenie limitu rozmiaru pliku

Mechanizmy generowania sygnałów

- wyjątki sprzętowe: nielegalna instrukcja, nielegalne odwołanie do pamięci, dzielenie przez 0, itp. (`SIGILL`, `SIGSEGV`, `SIGFPE`)
- naciskanie pewnych klawiszy na terminalu użytkownika (`SIGINT`, `SIGQUIT`)
- wywołanie komendy `kill` przez użytkownika (`SIGTERM`, i inne)
- funkcje `kill()` i `raise()`
- mechanizmy software-owe (`SIGALRM`, `SIGWINCH`)

Sygnały mogą być wysyłane do konkretnego pojedynczego procesu, do wszystkich procesów z danej grupy procesów, albo wszystkich procesów danego użytkownika. W przypadku procesu z wieloma wątkami sygnał jest doręczany do jednego z wątków procesu.

Reakcje na sygnał

Proces może zadeklarować jedną z następujących możliwości reakcji na sygnał:

- ignorowanie,
- wstrzymanie doręczenia mu sygnału na jakiś czas, po którym odbierze on wysłane w międzyczasie sygnały,
- obsługa sygnału przez wyznaczoną funkcję w programie, tzw. handler,
- przywrócenie domyślnej reakcji na dany sygnał.

W przypadku skryptów zestaw możliwych reakcji na sygnał jest bardziej ograniczony (np. Bourne shell: polecenie `trap`).

Dla sygnałów `SIGKILL` i `SIGSTOP/SIGCONT` proces nie może w żaden sposób zmienić domyślnej reakcji na sygnał.

Obsługa sygnałów

Jakkolwiek procedura obsługi sygnału może być napisana dowolnie i podjąć dowolne akcje, typowa funkcja handlera jest krótka i ogranicza się do wykonania minimalnych, niezbędnych w związku z powstałym zdarzeniem czynności, np. zamknięcia plików zapisywanych, skasowania plików roboczych, itp.

Po ich wykonaniu handler ma do wyboru następujące opcje:

- zakończenie procesu,
- kontynuowanie procesu od miejsca przerwania, jednak niektórych funkcji systemowych nie można kontynuować, i zostają one przedterminowo zakończone (ale proces kontynuuje),
- wznowienie procesu od określonego zapamiętanego wcześniej punktu (funkcje `setjmp()/longjmp()`); przy czym zapamiętany zostaje jedynie stos obliczeń, ale nie wartości zmiennych globalnych itp. stan (np. deskryptory plików nie cofają wykonanych operacji wejścia/wyjścia).

W przypadku kontynuowania lub wznowienia procesu istnieje możliwość przekazania informacji o fakcie obsługi sygnału przez ustawienie jakiejś zmiennej globalnej.

Sygnały — funkcje obsługi (tradycyjne)

Istnieje kilka różnych modeli generowania i obsługi sygnałów. Jednym z nich jest tzw. model tradycyjny, pochodzący z wczesnych wersji Uniksa. Jego cechą charakterystyczną jest, że zadeklarowanie innej niż domyślna obsługi sygnału ma charakter jednorazowy, tj., po otrzymaniu pierwszego sygnału przywracana jest reakcja domyślna.

Funkcje tradycyjnego modelu obsługi sygnałów:

- `signal()` – deklaruje jednorazową reakcję na sygnał: ignorowanie (`SIG_IGN`), obsługa, oraz reakcja domyślna (`SIG_DFL`)
- `kill()` – wysyła sygnał do innego procesu
- `raise()` – wysyła sygnał do własnego procesu
- `pause()` – czeka na sygnał
- `alarm()` – uruchamia „budzik”, który przyśle sygnał `SIGALRM`

Handlery sygnałów

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Zwróćmy uwagę:

- deklaracja handlera: funkcja typu `void` z pojedynczym argumentem `int`
- powrót z handlera — wznowienie pracy programu

Handlery sygnałów — alarm

```
#include <signal.h>
#include <setjmp.h>

jmp_buf stan_pocz;
int obudzony = 0;

void obudz_sie(int syg) {
    signal(syg, SIG_HOLD);
    fprintf(stderr,
            "Sygnal %d!\n",
            syg);
    obudzony = 1;
    longjmp(stan_pocz, syg);
}

int main() {
    int syg;

    ...
    syg = setjmp(stan_pocz); /* np.poczatek petli */
    signal(SIGTERM, obudz_sie);
    signal(SIGINT, obudz_sie);
    signal(SIGALRM, obudz_sie);

    ...
    if (obudzony == 0) {
        alarm(30); /* limit 30 sekund */
        DlugieObliczenia();
        alarm(0); /* zdazyliśmy */
    }
    else
        printf("Program wznowiony po ");
        printf("przerwaniu sygnałem %d\n", syg);
}
```

Obsługa sygnałów — inne modele

Tradycyjny model generowania i obsługi sygnałów jest uproszczony i nieco niewygodny (istnieje okno czasowe w czasie którego brak jest zadeklarowanego handlera). Dlatego powstały alternatywne modele funkcjonowania sygnałów. Niestety, są one wzajemnie niekompatybilne.

Model BSD nie zmienia deklaracji handlera po otrzymaniu sygnału, natomiast w trakcie wykonywania handlera kolejne sygnały tego samego typu są automatycznie blokowane. Deklaracji handlera w tym modelu dokonuje się funkcją `sigvec()`.

Niezawodny model Systemu V ma podobne własności do modelu BSD, tzn. również pozwala na trwałą deklarację handlera. Wprowadza więcej funkcji i opcji obsługi sygnałów. Do deklaracji handlera w tym modelu służy funkcja `sigset()`.

Pomimo iż powyższe modele istnieją w wielu systemach uniksowych, ich stosowanie nie ma sensu w programach, które mają być przenośne. Sytuację uporządkował dopiero nowy model generowania i obsługi sygnałów — model POSIX.

Obsługa sygnałów — model POSIX

Standard POSIX zdefiniował nowy model generowania sygnałów i zestaw funkcji oferujący możliwość kompleksowego i bardziej elastycznego deklarowania ich obsługi:

- funkcja `sigaction()` – deklaruje reakcję na sygnał (trwale)
- automatyczne blokowanie na czas obsługi obsługiwanego sygnału i dowolnego zbioru innych sygnałów
- dodatkowe opcje obsługi sygnałów, np. handler sygnału może otrzymać dodatkowe argumenty: strukturę informującą o okolicznościach wysłania sygnału i drugą strukturę informującą o kontekście przez odebraniem sygnału

rozszerzony prototyp handlera:

```
void handler(int sig, siginfo_t *sip, ucontext_t *uap);
```

- `sigprocmask()/sigfillset()/sigaddset()` – operacje na zbiorach sygnałów

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

void joj(int sig, siginfo_t *sip, ucontext_t *uap) {
    printf("Dostalem signal numer %d\n", sig);
    if (sip->si_code <= 0)
        printf("Od procesu %d\n", sip->si_pid);
    printf("Kod sygnalu %d\n", sip->si_code);
}

void main() {
    struct sigaction act;

    act.sa_handler = joj;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGINT, &act, 0);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Co to są sygnały i jak są generowane?
2. Jakie są możliwe reakcje procesu na otrzymanie sygnału?
3. W jaki sposób proces może przeprowadzić obsługę sygnału?