

LINUX PROGRAMOWANIE SYSTEMOWE [1/3]

Plan wykładu

- Procesy
- Sygnały
- Pamięć

Procesy

- **Procesy**
- Sygnały
- Pamięć

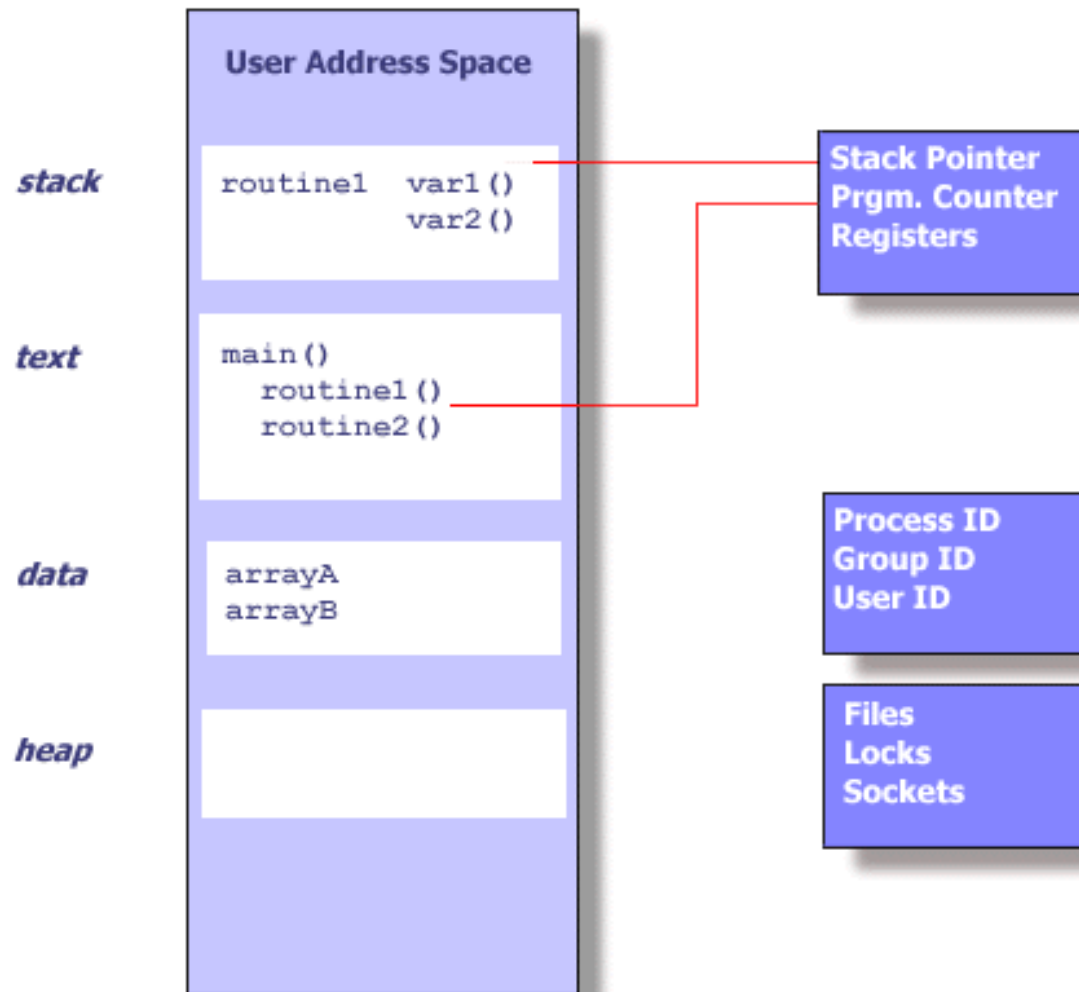
Co to jest proces? 1/2

Proces to egzemplarz wykonywanego **programu**. Należy odróżnić proces od wątku - każdy proces posiada własną przestrzeń adresową, natomiast wątki posiadają wspólną sekcję danych.

Na **kontekst** procesu składają się m.in.:

- identyfikator procesu, identyfikator grupy procesów, identyfikator użytkownika, identyfikator grupy użytkowników
- środowisko
- katalog roboczy
- kod programu
- rejestry
- stos
- sarta
- deskryptory plików
- akcje sygnałów
- biblioteki współdzielone
- narzędzia komunikacji międzyprocesowej

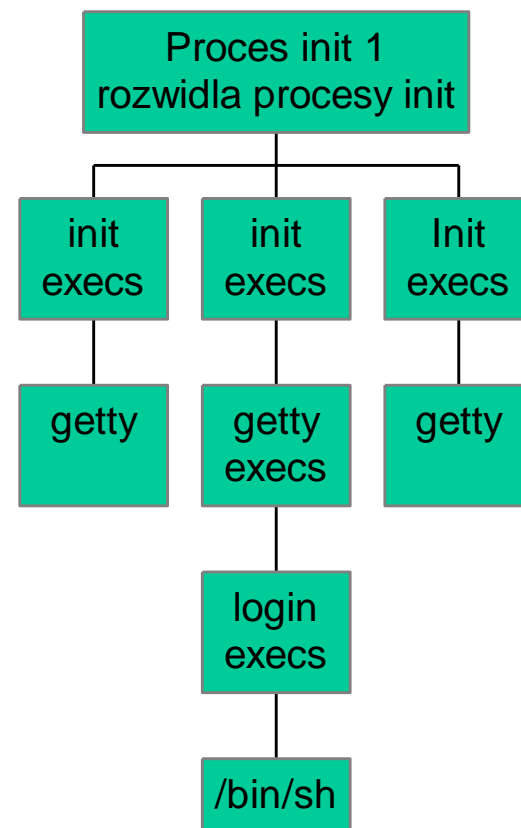
Co to jest proces? 2/2



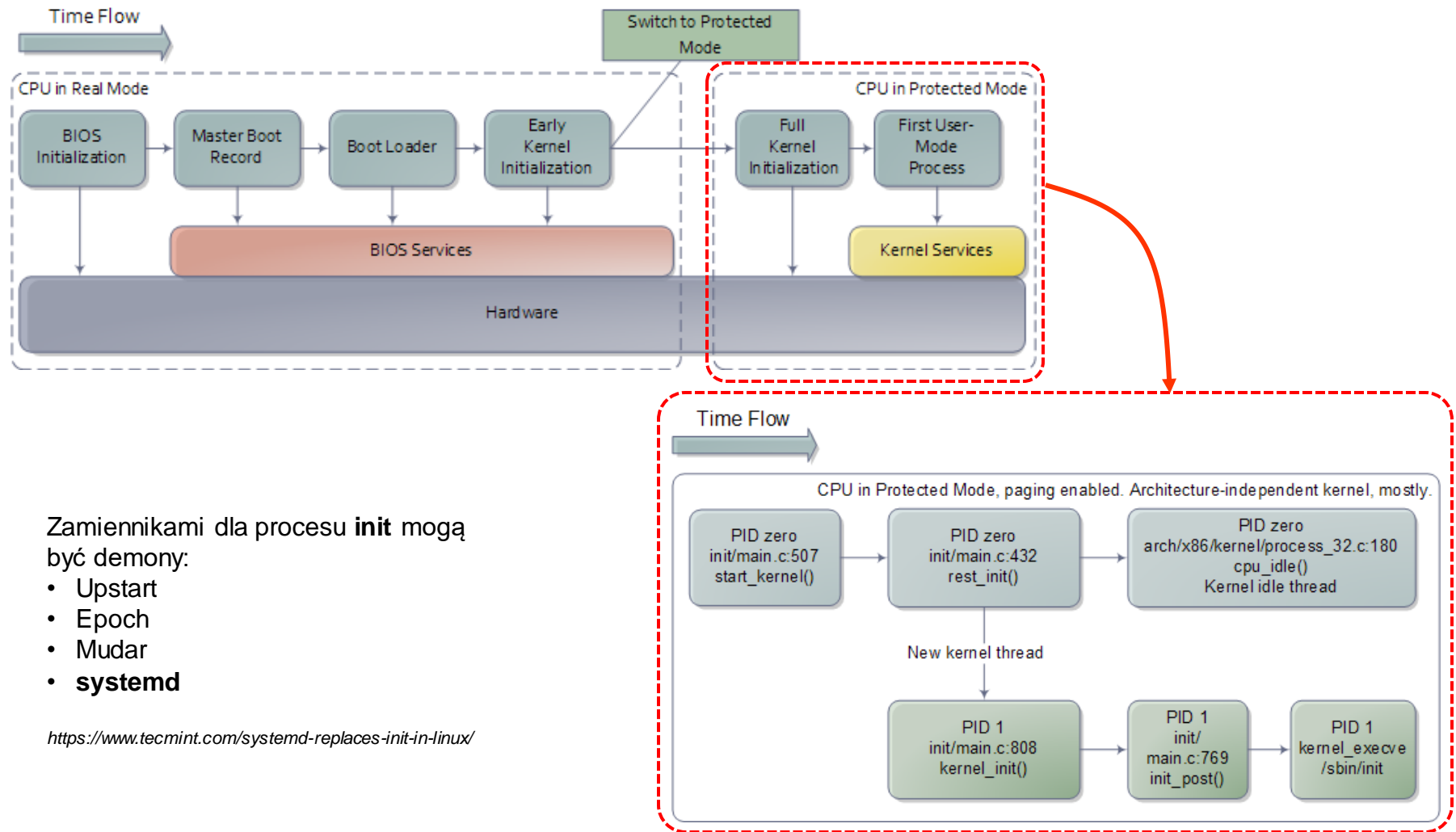
Hierarchia procesów 1/2

- Nowe procesy zwykle tworzone za pomocą **fork** lub **vfork**
- Dobrze zdefiniowana hierarchia procesów: **jeden rodzic** i zero lub więcej procesów potomnych. Proces **init** jest korzeniem tego drzewa.
- Podczas życia procesu za pomocą wywołania funkcji systemowej **exec** można zmienić jego kod i dane
- Procesy kończą się zwykle po wywołaniu **exit**

Generacje procesów



Hierarchia procesów 2/2



Zamiennikami dla procesu **init** mogą być demony:

- Upstart
- Epoch
- Mudar
- **systemd**

<https://www.tecmint.com/systemd-replaces-init-in-linux/>

Kontekst procesu

- Przestrzeń adresowa
 - kod, dane, stos, pamięć współdzielona, ...
- Informacje kontrolne (u-obszar, tablica procesów **proc**)
 - u-obszar, tablica procesów, odwzorowania
 - odwzorowania translacji adresów
- Dane uwierzytelniające
 - ID użytkownika i grupy (rzeczywiste i efektywne)
- Zmienne środowiskowe
 - **zmienna=wartość**
 - zwykle przechowywane na spodzie stosu

Informacje kontrolne procesu

- **U-obszar**

- Część przestrzeni użytkownika (powyżej stosu).
- Zwykle odwzorowywany w ustalony adres.
- Zawiera informacje niezbędne podczas wykonywania procesu.
- Może być wymieniany (ang. *swapped*)

- **Tablica procesów Proc**

- Zawiera informacje konieczne m.in. wtedy, gdy proces nie wykonuje się.
- Nie może być wymieniany (ang. *swapped*).
- Tradycyjna tabela o ustalonym rozmiarze.

U-obszar i tablica Proc

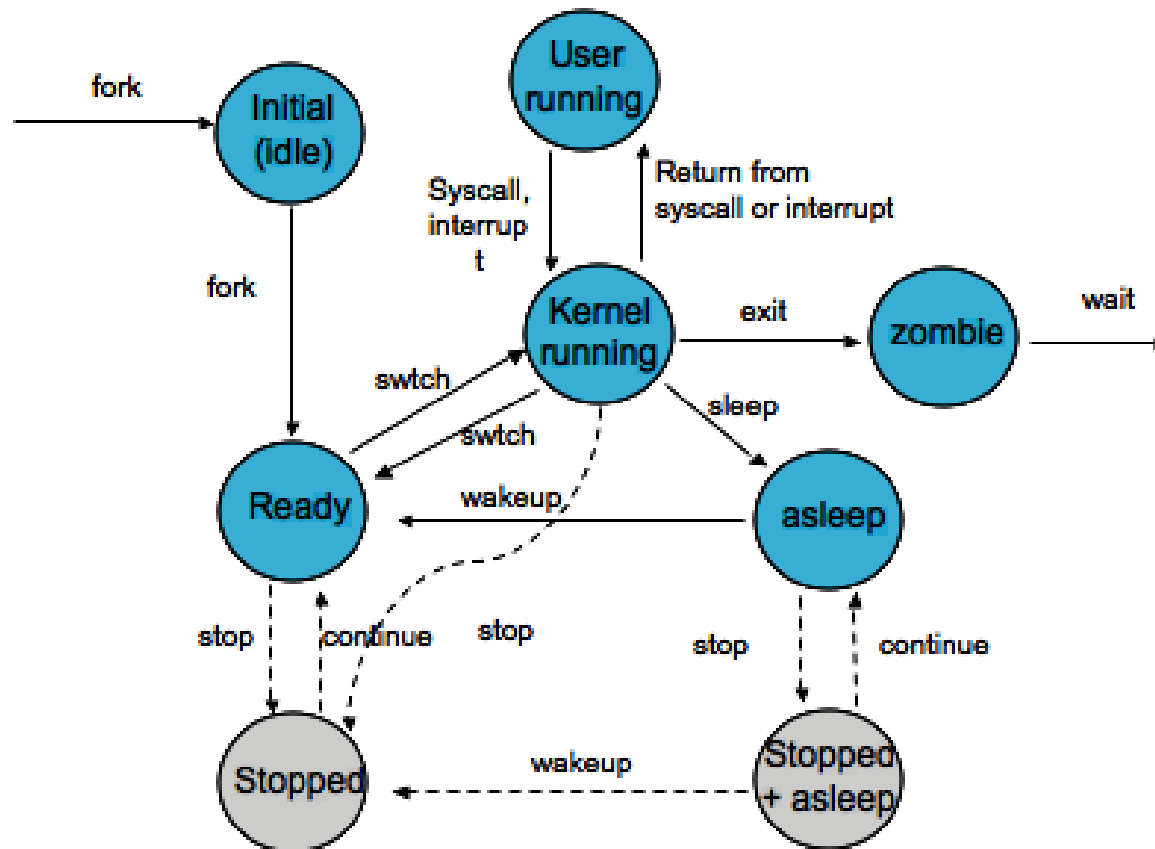
U-obszar

- **Wskaźnik** do pozycji w tablicy Proc
- Rzeczywisty/efektywny **UID**
- **argumenty**, zwracane wartości lub **błędy** bieżącego wywołania funkcji systemowej.
- Tablica reakcji na **sygnały**
- Tabela deskryptorów **plików**
- Bieżący katalog i bieżący korzeń.

Tablica Proc

- ID procesu i grupy procesów
- Wskaźnik na U-obszar
- Stan procesu
- Wskaźniki na kolejki – planowania, uśpione, etc.
- Priorytet
- Informacja zarządzania pamięcią
- Flagi (znaczniki)
- Tablica odebranych i nieobsłużonych sygnałów

Uproszczony graf stanu procesów



Szeregowanie procesów UNIX

- Unix może jednocześnie (współbieżnie) wykonywać **wiele procesów**
- Algorytm planowania **round-robin z wywłaszczzeniami**
- Każdy proces posiada przydzielony, ustalony **kwant czasu**
- Procesy w trybie jądra mają przypisane **priorytet jądra**, który jest wyższy niż **priorytety użytkownika**.
- Priorytety: **jądra 0-49, użytkownika 50-127**.

Planowanie procesów w systemie Linux (1/2)

- Dwa oddzielne algorytmy planowania procesów:
 - Algorytm z **podziałem czasu** - do sprawiedliwego planowania z wywłaszczeniami działania wielu procesów.
 - Algorytm dla zadań **czasu rzeczywistego**, gdzie priorytety bezwzględne są ważniejsze niż sprawiedliwość.
- W skład **kontekstu każdego procesu** wchodzi **klasa planowania**, określająca, który z algorytmów ma być użyty dla procesu.

Planowanie procesów w systemie Linux (2/2)

- Dla **zwykłych** procesów z podziałem czasu stosowany jest algorytm priorytetowy oparty na **kredytowaniu**: $\text{kredyt}_n = \text{kredyt}_{n-1}/2 + \text{priorytet}$
 - Do wykonania wybierany jest proces o **największym kredycie**.
 - Kredyt wykonywanego procesu jest **obniżany** o jedną jednostkę przy **każdym przerwaniu zegara** - kiedy kredyt spadnie do zera, proces jest zawieszany.
 - Kiedy żaden z procesów gotowych do działania nie ma kredytu, następuje **wtórne kredytowanie** każdego procesu w systemie (a nie tylko procesów gotowych).
- Dla procesów **czasu rzeczywistego** realizowane są dwie klasy planowania:
 - Algorytm **FCFS**;
 - Algorytm **RR** (rotacyjny).
 - Każdy proces **posiada priorytet** - planista uruchamia ten o najwyższym priorytecie!
 - Kod jądra **nie może** być wywłaszczony przez kod trybu użytkownika!

Tworzenie procesu

- **fork**

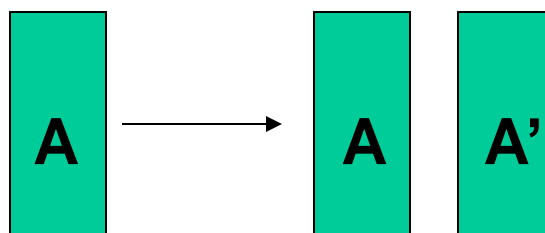
- Tworzy nowy proces
- Kopiuje pamięć wirtualną rodzica
- Kopiuje katalog roboczy i deskryptory otwartych plików
- Zwraca procesowi rodzicielskiemu wartość PID potomka
- Zwraca procesowi potomnemu wartość 0

- **exec**

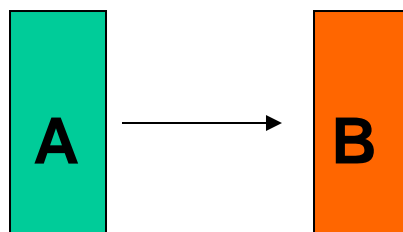
- Nadpisuje istniejący proces nowym kodem z podanego programu

Tworzenie procesu

- Wywołanie systemowe **fork** klonuje aktualny proces

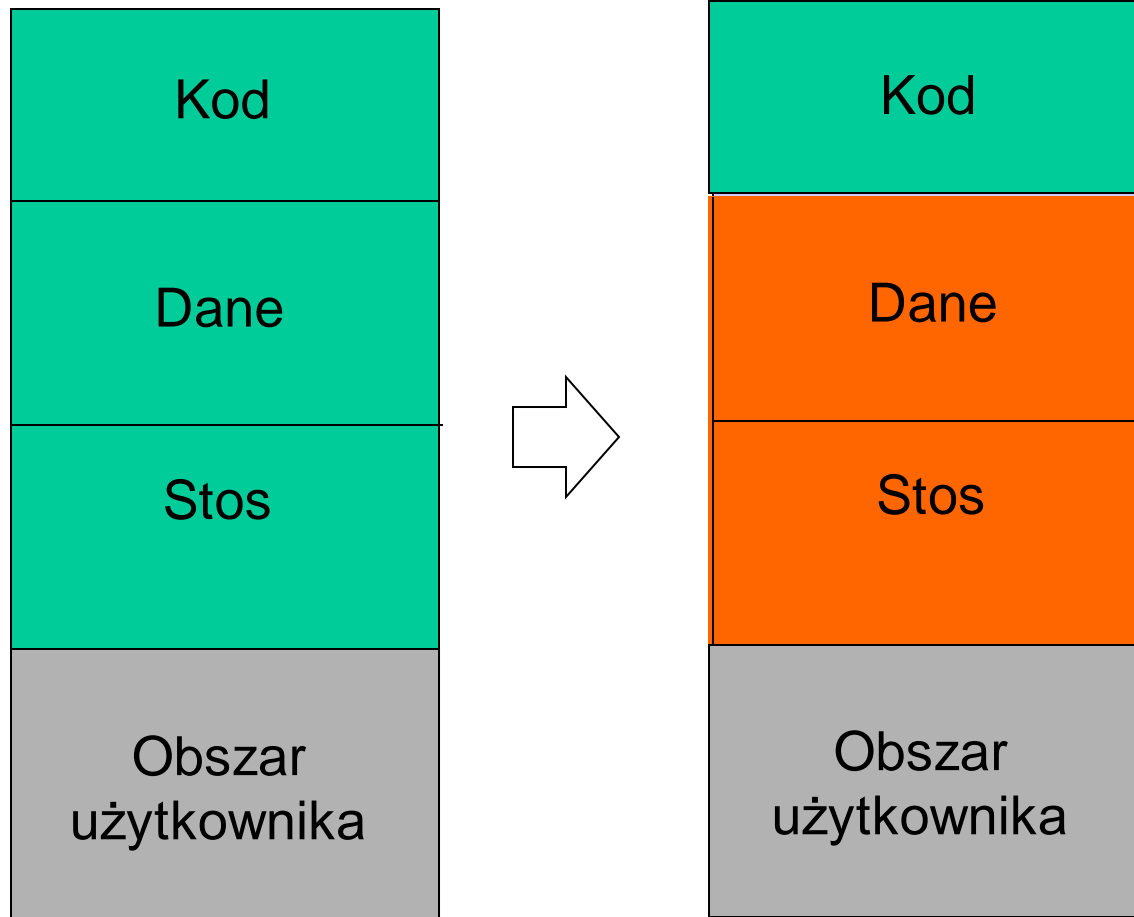


- Wywołanie systemowe **exec** zastępuje bieżący proces



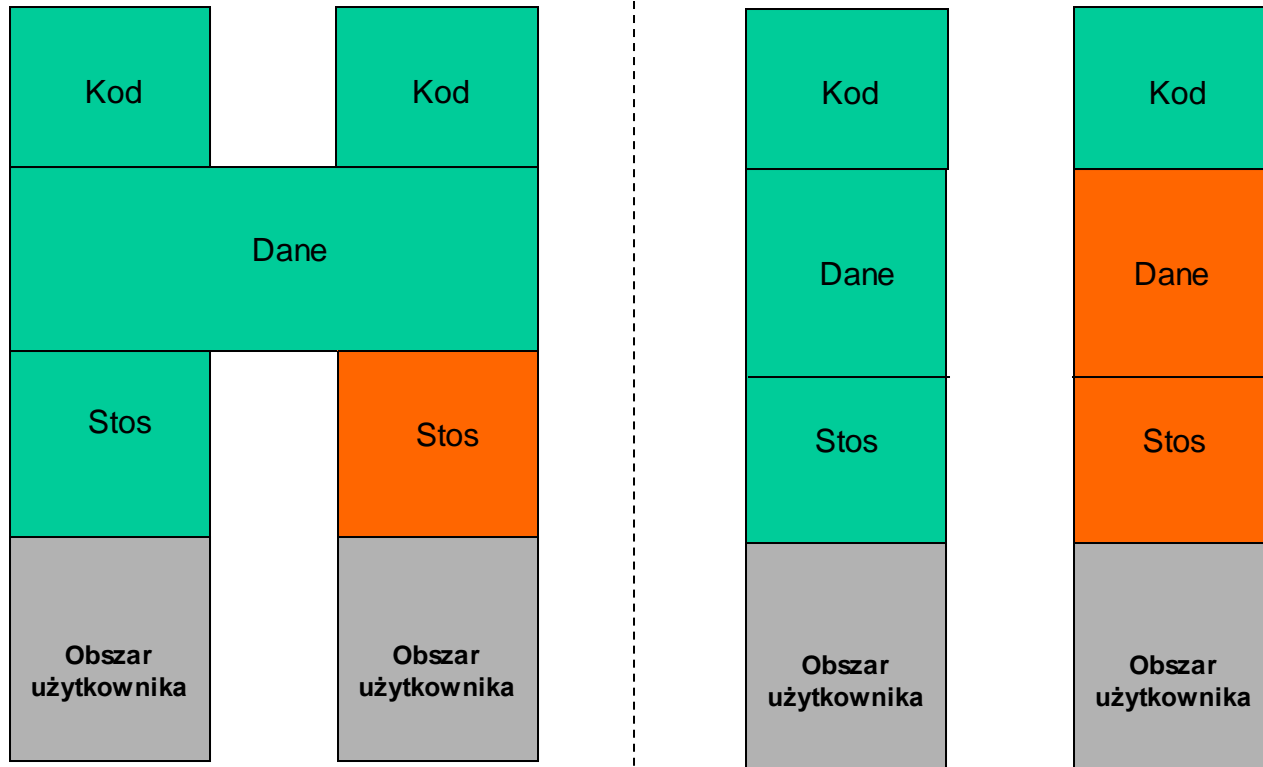
- Zwykle **exec** jest wywoływany zaraz po **fork**

Działanie fork



Efektywna implementacja fork

Kopiuj po zapisie



Przed zapisem do obszaru **Dane**

Po zapisie do obszaru **Dane**

Kończenie procesu

- Wywoływana jest funkcja **exit**
 - Zamyka otwarte pliki
 - Zwalnia inne zasoby
 - Zapisuje statystyki użytkownika zasobów i status powrotu w tablicy procesów
 - Budzi rodzica (jeśli czeka)
 - Wywołuje **swtch**
- Proces jest w stanie **zombie**
- Rodzic gromadzi przez funkcję **wait** status zakończenia procesu i statystyki użytkownika zasobów

Utrzymywana informacja o procesie 1/2

- Katalog roboczy
- Tablica deskryptorów plików
- ID procesu
 - Liczba używana do identyfikacji procesu
- ID grupy procesów
 - Liczba używana do identyfikacji zbioru procesów
- ID procesu rodzica
 - ID procesu, który utworzył proces

Utrzymywana informacja o procesie 2/2

- Efektywny ID użytkownika i grupy
 - Użytkownik i grupa, którzy mają prawo uruchomienia procesu
- Rzeczywisty ID użytkownika i grupy
 - Użytkownik i grupa, którzy wywołali proces
- **umask**
 - Domyślne (negatywne) prawa dostępu do nowo tworzonego pliku
- Zmienne środowiskowe

Dane uwierzytelniające użytkownika

- Każdy użytkownik posiada przypisany unikalny ID użytkownika (**uid**) oraz ID grupy (**gid**).
- Superużytkownik (**root**) ma **uid == 0** i **gid == 0**
- Każdy proces posiada zarówno rzeczywisty jak i efektywny ID.
 - Efektywny ID => tworzenie plików i dostęp,
 - Rzeczywisty ID => rzeczywisty właściciel procesu. Stosowany podczas wysyłania sygnałów.
 - Efektywny lub rzeczywisty ID nadawcy musi być równy rzeczywistemu ID odbiorcy.

Środowisko procesu

- Zbiór par nazwa-wartość związanych z procesem
- Klucze i wartości są napisami
- Przekazywane procesom potomnym
- Nie mogą być zwracane z powrotem
- Typowe przykłady:
 - **PATH**: gdzie szukać programów
 - **TERM**: typ terminala

Identyfikacja procesów

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

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

```
pid_t getpid(void) ;
```

Zwraca identyfikator bieżącego procesu

```
pid_t getppid(void) ;
```

Zwraca identyfikator procesu macierzystego

Identyfikacja właścicieli procesów

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

```
uid_t getuid(void);
```

Zwraca rzeczywisty ID użytkownika dla aktualnego procesu

```
uid_t geteuid(void);
```

Zwraca efektywny ID użytkownika dla aktualnego procesu

```
gid_t getgid(void);
```

Zwraca rzeczywisty ID grupy bieżącego procesu

```
gid_t getegid(void);
```

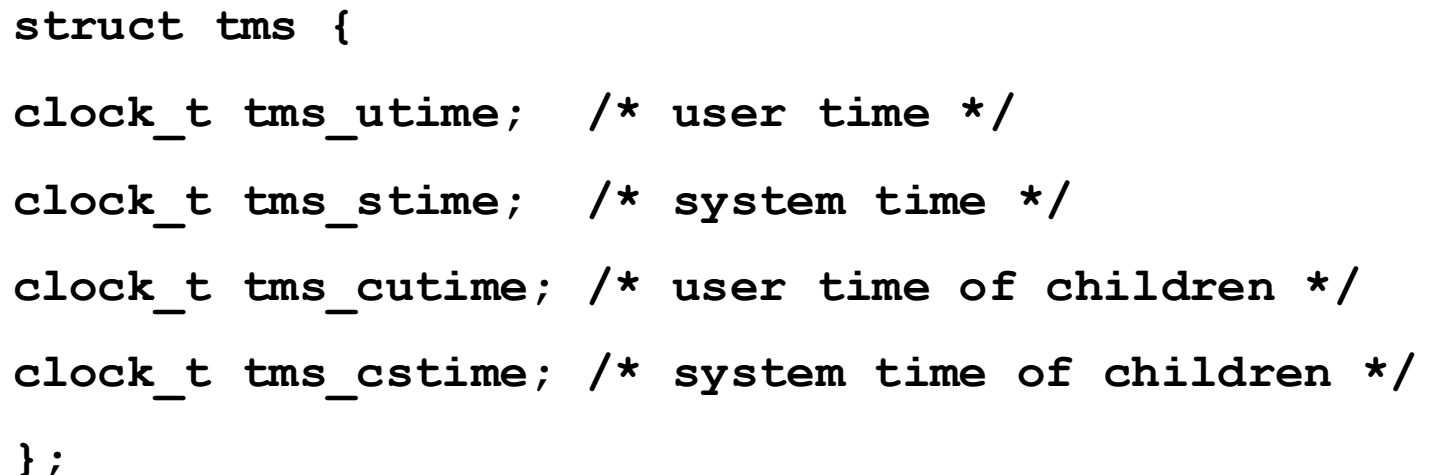
Zwraca efektywny ID grupy bieżącego procesu

Czas działania procesu

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buf);
```

Zwraca zużyty czas zegarowy liczony w taktach zegara



```
struct tms {  
    clock_t tms_utime; /* user time */  
    clock_t tms_stime; /* system time */  
    clock_t tms_cutime; /* user time of children */  
    clock_t tms_cstime; /* system time of children */  
};
```

Tworzenie nowego procesu: `fork` 1/3

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

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

```
pid_t fork(void);
```

Funkcja tworzy proces potomny różniący się zasadniczo od procesu macierzystego jedynie wartościami **PID** i **PPID**.

W systemie **Linux** funkcja implementuje mechanizm **copy-on-write** (stały narzut czasowy jedynie na skopiowanie **tablicy stron rodzica** i utworzenie nowej struktury procesu)

Funkcja zwraca **PID potomka** do procesu macierzystego, wartość 0 do procesu potomnego i wartość -1 w przypadku błędu.

Tworzenie nowego procesu: `fork` 2/3

test-fork.c

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

int main ()
{
    pid_t child_pid;
    printf ("Program główny przed fork() , PID = %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid > 0) {
        printf ("To jest proces macierzysty o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID potomka wynosi %d\n", (int) child_pid);
    }
    else if (child_pid == 0) {
        printf ("To jest proces potomny o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID procesu macierzystego wynosi %d\n", (int) getppid());
    }
    return 0;
}
```

Tworzenie nowego procesu: `fork` 3/3

Proces macierzysty

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

int main ()
{
    pid_t child_pid;
    printf ("Program główny przed fork(), PID = %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid > 0) {
        printf ("To jest proces macierzysty o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID potomka wynosi %d\n", (int) child_pid);
    }
    else if (child_pid == 0) {
        printf ("To jest proces potomny o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID procesu macierzystego wynosi %d\n", (int) getppid());
    }
    return 0;
}
```

Proces potomny

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

int main ()
{
    pid_t child_pid;
    printf ("Program główny przed fork(), PID = %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid > 0) {
        printf ("To jest proces macierzysty o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID potomka wynosi %d\n", (int) child_pid);
    }
    else if (child_pid == 0) {
        printf ("To jest proces potomny o numerze PID = %d\n", (int) getpid ());
        printf ("Numer PID procesu macierzystego wynosi %d\n", (int) getppid());
    }
    return 0;
}
```

Tworzenie nowego procesu: `vfork`

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

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

```
pid_t vfork(void) ;
```

Funkcja różni się od `fork` tym, że wykonanie rodzica jest zawieszane do momentu wykonania przez potomka `execve` lub `_exit`. Potomek **nie kopiuje tablicy stron rodzica**.

Funkcja ma większe znaczenie w systemach, w których `fork` nie implementuje mechanizmu **copy-on-write**.

Proces potomny a proces macierzysty 1/2

Proces potomny **dziedziczy** po procesie macierzystym:

- rzeczywiste i efektywne identyfikatory użytkownika i grupy,
- deskryptory plików (i pozycje w plikach)
- identyfikatory dodatkowych grup, identyfikator sesji, terminal sterujący, sygnalizator ustanowienia identyfikatora użytkownika oraz sygnalizator ustanowienia identyfikatora grupy, bieżący katalog roboczy, katalog główny, maskę tworzenia plików,
- maskę sygnałów oraz dyspozycje obsługi sygnałów,
- sygnalizator zamykania przy wywołaniu funkcji exec (close-on-exec) dla wszystkich otwartych deskryptorów plików,
- środowisko,
- przyłączone segmenty pamięci wspólnej,
- ograniczenia zasobów systemowych.

Proces potomny a proces macierzysty 2/2

Różnice między procesem potomnym i macierzystym:

- wartość powrotu z funkcji **fork**,
- różne identyfikatory procesów,
- inne identyfikatory procesów macierzystych - w procesie potomnym jest to identyfikator procesu macierzystego; w procesie macierzystym identyfikator procesu macierzystego nie zmienia się,
- w procesie potomnym wartości **tms_utime**, **tms_cutime** i **tms_ustime** są równe **0**,
- potomek nie dziedziczy **blokad plików**, ustalonych w procesie macierzystym,
- w procesie potomnym jest zerowany **zbiór zaległych sygnałów**.

Uruchamianie programów: `exec` . . . 1/4

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

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

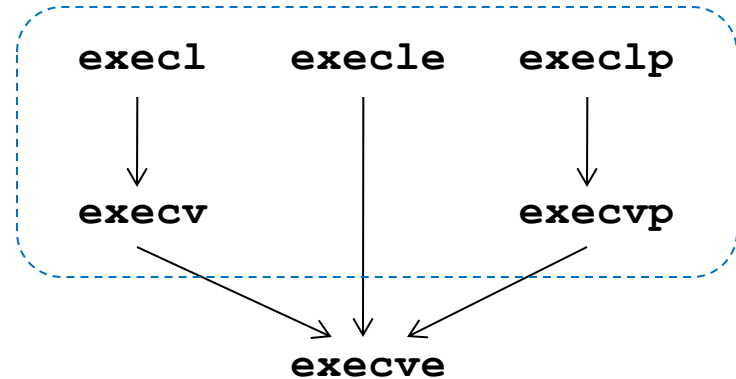
Funkcja uruchamia program wskazany przez *filename*

- **argv** jest tablicą łańcuchów przekazywanych jako argumenty nowego programu, pierwszym argumentem jest powtórzona nazwa programu.
- **envp** jest tablicą łańcuchów postaci **klucz=wartość**, która jest przekazywana jako środowisko do nowego programu.
- **argv** i **envp** muszą być zakończone wskaźnikiem pustym (**NULL**).
- tablica argumentów oraz środowisko są dostępne w funkcji **main** wywoływanego programu.

Uruchamianie programów: **exec** . . . 2/4

- **execve** nie powraca po pomyślnym wywołaniu,
- segmenty **text**, **data**, **bss** oraz **segment stosu** procesu wywołującego zostają nadpisane przez odpowiedniki ładowanego programu,
- wywoływany program dziedziczy **PID** procesu wywołującego i wszelkie deskryptory otwartych plików, które nie są ustawione jako **close-on-exec**,
- sygnały oczekujące na proces wywołujący zostają wyczyszczone,
- sygnałom, które były przechwytywane przez proces wywołujący, zostaje przypisana ich domyślna obsługa,
- Jeżeli plik programu wskazywany przez **filename** ma ustawiony bit **set-uid**, to efektywny identyfikator użytkownika procesu wywołującego jest ustawiany na właściciela pliku programu

Uruchamianie programów: **exec** . . . 3/4



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

```
extern char **environ;
```

```
int execl(const char *path, const char *arg, ...);  
int execlp(const char *file, const char *arg, ...);  
int execle(const char *path, const char *arg, ...,  
           char * const envp[]);  
int execv(const char *path, char *const argv[]);  
int execvp(const char *file, char *const argv[]);
```

Uruchamianie programów: `exec` . . . 4/4

program1

```
int main()
{
    printf( "\n %d %d", getpid(), getppid());
    execl( "program2", "program2", NULL);
    printf("\n koniec 1");
}
```

program2

```
int main()
{
    printf( "\n %d %d", getpid(), getppid());
    printf( "\n koniec 2");
}
```

```
$ ./program1
```

```
100 95
```

```
... i co dalej ?
```

Kończenie procesu: `exit`

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

```
void exit(int status) ;
```

Funkcja powoduje **normalne zakończenie** programu i zwraca do procesu macierzystego wartość `status`. Wszystkie funkcje zarejestrowane za pomocą `atexit` są wykonywane w kolejności odwrotnej niż zostały zarejestrowane, a wszystkie otwarte strumienie są zamykane po opróżnieniu ich buforów.

Taki sam efekt daje wywołanie instrukcji `return` funkcji `main`.

Kończenie procesu: `_exit`

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

```
void _exit(int status);
```

Funkcja “**natychmiast**” kończy proces, z którego została wywołana. Wszystkie przynależące do procesu otwarte deskryptory plików są zamykane; wszystkie jego procesy potomne są przejmowane przez proces 1 (`init`), a jego proces macierzysty otrzymuje sygnał `SIGCHLD`.

`_exit` nie wywołuje żadnych funkcji zarejestrowanych za pomocą funkcji `atexit`

Kończenie procesu: `atexit` 1/2

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

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

Procedura rejestruje bezargumentową funkcję wskazaną przez `func`. Wszystkie funkcje zakończenia zarejestrowane za pomocą `atexit` przy zakończeniu będą wywoływane w odwrotnej kolejności.

Kończenie procesu: `atexit` 2/2

`test-atexit.c`

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

void fnExit1 ( void)
{
    puts ( "Exit function 1.");
}

void fnExit2 ( void)
{
    puts ( "Exit function 2.");
}

int main ()
{
    atexit ( fnExit1);
    atexit ( fnExit2);
    puts ( "Main function.");
    return 0;
}
```

```
$/test-atexit
Main function.
Exit function 2.
Exit function 1.
```


Synchronizacja procesów: `wait`

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

Funkcja zatrzymuje wykonywanie bieżącego procesu aż do zakończenia procesu potomka lub aż do dostarczenia sygnału kończącego bieżący proces lub innego, dla którego wywoływana jest funkcja obsługi sygnału.

Jeśli potomek zakończył działanie przed wywołaniem tej funkcji ("zombie"), to funkcja kończy się natychmiast. Wszelkie zasoby potomka są zwalniane.

Jeśli *status* nie jest równe **NULL** to funkcja zapisuje dane o stanie zakończonego potomka w buforze wskazywanym przez *status*.

Zwracany jest PID zakończonego potomka albo -1 w przypadku błędu

Synchronizacja procesów: `waitpid` 1/2

```
#include <sys/types.h>
#include <sys/wait.h>
```

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

Funkcja zawiesza wykonywanie bieżącego procesu dopóki potomek określony przez ***pid*** nie zakończy działania lub dopóki nie zostanie dostarczony sygnał, którego akcją jest zakończenie procesu lub wywołanie funkcji obsługującej sygnały.

Synchronizacja procesów: `waitpid` 2/2

Wartość ***pid*** może być:

- < -1 oczekiwanie na dowolny proces potomny, którego ID grupy procesów jest równy wartości bezwzględnej *pid*
- 1 oczekiwanie na dowolny proces potomny (takie samo zachowanie, jakie wykazuje `wait`)
- 0 oczekiwanie na każdy proces potomny, którego ID grupy procesu jest równe ID grupy procesu wywołującego funkcję.
- > 0 oczekiwanie na potomka, którego ID procesu jest równy wartości *pid*.

Ustawienie ***options*** na **WNOHANG** oznacza natychmiastowy powrót z funkcji, jeśli potomek nie zakończył pracy. Funkcja zwraca wtedy wartość **0**.

Przedwczesne zakończenie i 'zombie'

1. Proces potomny kończy się w czasie, gdy jego proces rodzicielski nie wykonuje funkcji **wait**

*Proces potomny staje się procesem zombie i umieszczany jest w stanie zawieszenia. Nadal zajmuje pozycję w tablicy procesów jądra, ale nie używa innych zasobów jądra. Pozycja w tablicy procesów zostanie zwolniona po wywołaniu przez rodzica funkcji **wait**.*

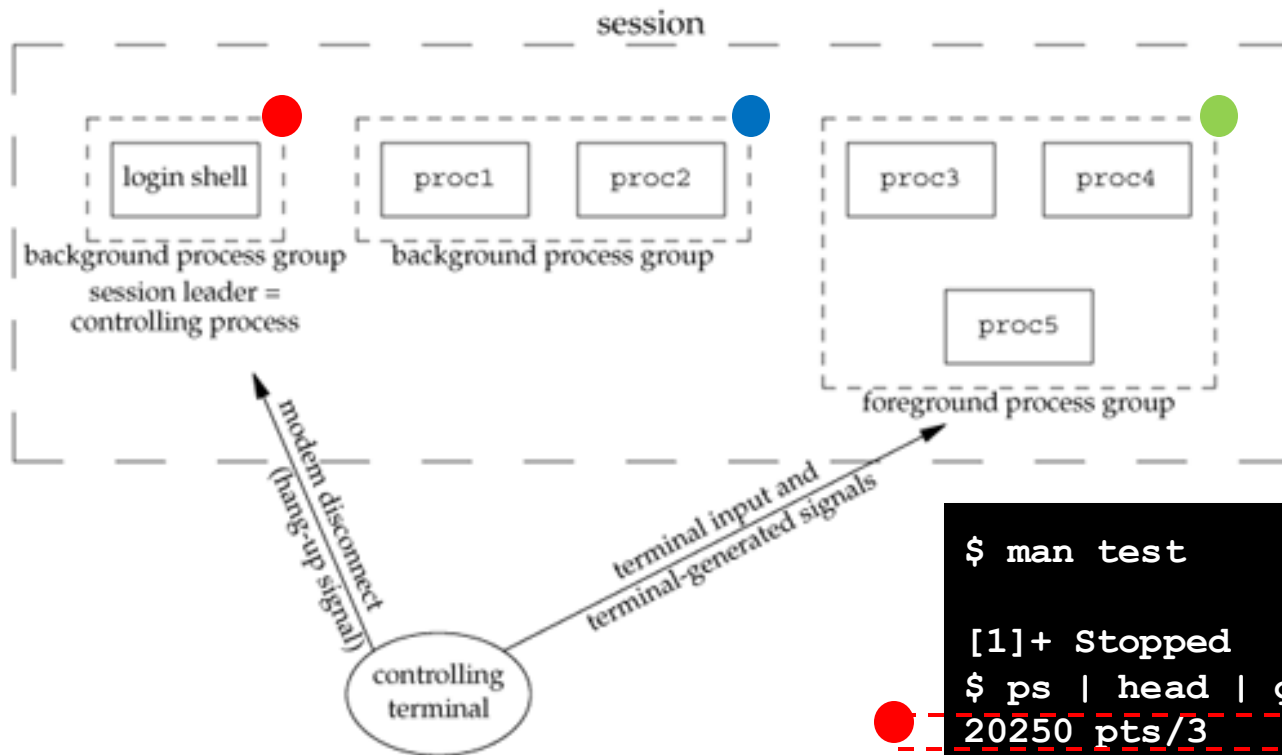
2. Proces rodzicielski kończy się, gdy jeden lub więcej procesów potomnych ciągle działa

*Procesy potomne (w tym potencjalne procesy zombie) są adoptowane przez proces **init**.*

Grupowanie procesów 1/5

- Każdy proces należy do pewnej **grupy procesów**. W każdej grupie jest **lider** (główny proces) oraz opcjonalnie inne procesy. Identyfikator grupy procesów (**PGID** – **uwaga! nie mylić z identyfikatorem grupy GID!!!**) jest równy identyfikatorowi PID lidera. Grupowanie pozwala na zarządzanie zbiorem procesów (np. polecenia **jobs**, **fg**, **bg** dotyczą grup, które są równoważne „pracom” - job).
- Grupy procesów działają w ramach **sesji**. Sesja zazwyczaj powiązana jest z **terminalem** sterującym (wyjątek – demon). W systemie może istnieć wiele sesji. Sesja zawiera co najmniej jedną grupę procesów posiadającą co najmniej jeden proces. Identyfikator sesji (**SID**) jest tożsamy z identyfikatorem PID **lidera sesji**.

Grupowanie procesów 2/5



```
$ man test
```

```
[1]+ Stopped
```

```
man test
```

```
$ ps | head | grep "^[0-9]"
```

```
20250 pts/3 00:00:00 bash
```

```
20914 pts/3 00:00:00 man
```

```
20926 pts/3 00:00:00 pager
```

```
21563 pts/3 00:00:00 ps
```

```
21564 pts/3 00:00:00 head
```

```
21565 pts/3 00:00:00 grep
```

Grupowanie procesów 3/5

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

```
int setsid(void);
```

Utworzenie nowej sesji (aktualny proces zostaje jej liderem). Nie można wykonać dla procesu będącego liderem grupy.

```
pid_t getsid(pid_t pid);
```

Pobranie identyfikatora sesji (czyli identyfikatora PID jej lidera) dla proces wskazanego jako argument. Podanie argumentu **0** oznacza, że jesteśmy zainteresowani identyfikatorem sesji procesu bieżącego.

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

Przeniesieni procesu wskazanego przez **pid** do grupy o numerze **pgid**. Podanie **pid** równego **0** oznacza, że odnosimy się do procesu bieżącego. Obie grupy muszą znajdować się w tej samej sesji. Podanie **pgid** równego **0** oznacza, że ma zostać stworzona nowa grupa, której liderem zostanie bieżący proces. Nie można zmieniać grupy liderowi sesji.


```
pid_t getpgid(pid_t pid);
```

Pobranie identyfikatora grupy (czyli identyfikatora PID jej lidera) dla proces wskazanego jako argument. Podanie argumentu **0** oznacza, że jesteśmy zainteresowani identyfikatorem grupy procesu bieżącego.

Grupowanie procesów 4/5

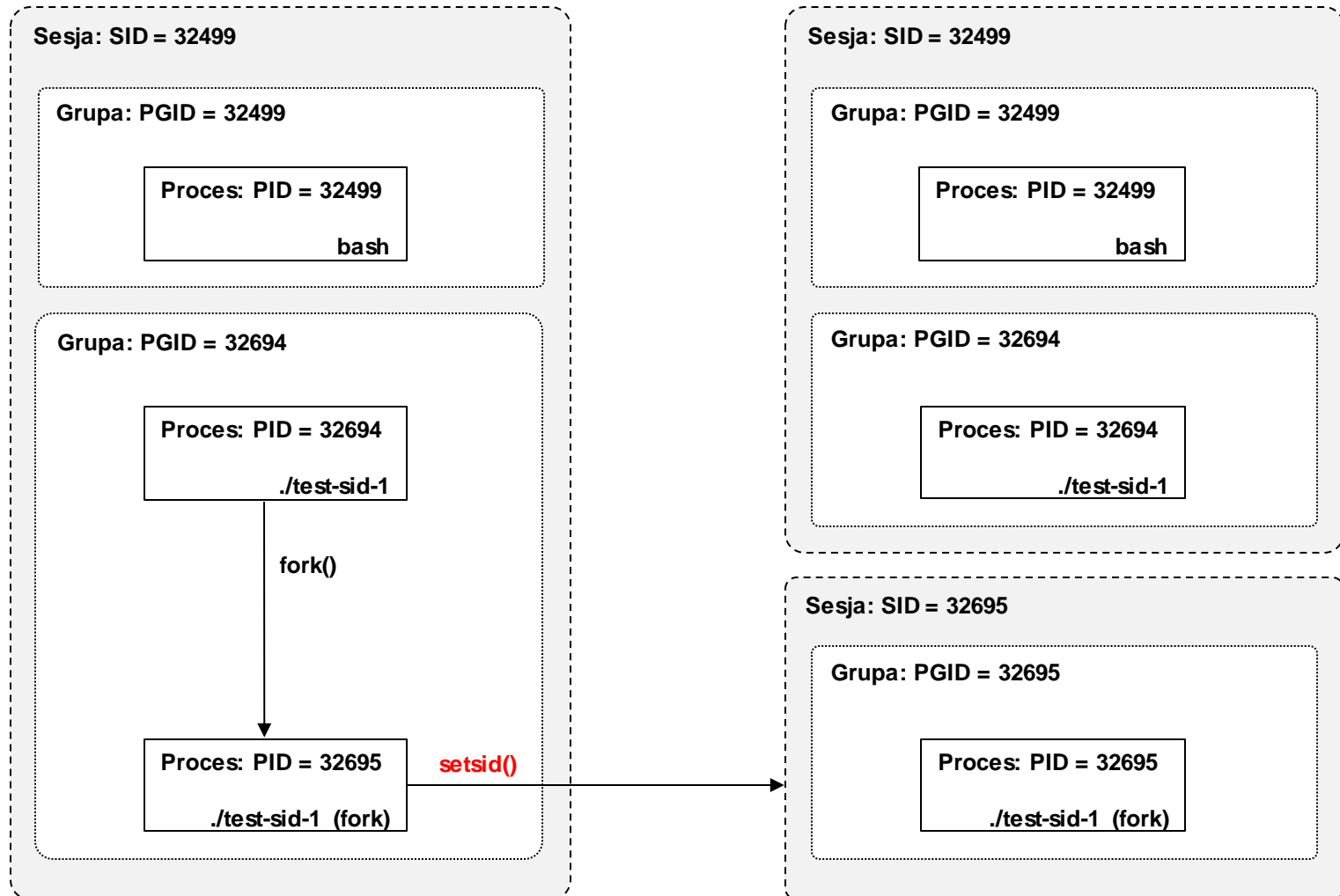
test-sid-1.c

```
...  
  
int show(void) {  
    printf("ppid %d  pid %d  sid %d  pgid %d  \n", getppid(), getpid(), getsid(0), getpgid(0));  
}  
  
int main() {  
    show();  
    if(fork() == 0) {  
        show();  
        if ( -1 == setsid())  
            perror("");  
        else  
            show();  
    }  
    return 0;  
}
```



```
$ ps  
  PID TTY          TIME CMD  
 32499 pts/7        00:00:00 bash  
 32653 pts/7        00:00:00 ps  
  
$ ./test-sid-1  
ppid 32499  pid 32694  sid 32499  pgid 32694  
ppid 32694  pid 32695  sid 32499  pgid 32694  
ppid 32694  pid 32695  sid 32695  pgid 32695
```


Grupowanie procesów 5/5



Demony

- Demon – proces niepodłączony do żadnego terminala sterującego, posiadający swoją sesję, działający w tle. Musi być bezpośrednim potomkiem procesu **init**.
- Aby utworzyć demona:
 - tworzymy normalny proces (proces rodzica),
 - wewnątrz rodzica za pomocą wywołania **fork()** tworzymy proces potomny,
 - kończymy działanie procesu rodzica – proces potomny zostaje zaadoptowany przez proces **init**,
 - wołamy **setsid()** tworząc dla procesu nową sesję i odcinając go od terminala,
 - zmieniamy katalog roboczy na katalog główny systemu plików,
 - zamykamy wszystkie deskryptory otwartych plików (w tym **stdin**, **stdout** i **stderr**, czyli deskryptory 0, 1 i 2),
 - ponownie otwieram deskryptory 0,1 i 2, ale tak, aby wskazywały **/dev/null**
 - uruchamiamy logikę demona.

Inne funkcje

```
#include <sched.h>
```

```
int sched_yield (void);
```

Proces rezygnuje z użycia procesora i zostaje przeniesiony na koniec kolejki swojego statycznego priorytetu (zostaje uruchomiony kolejny proces).

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

```
int nice (int inc);
```

Zmiana wartości **nice** procesu o wartość **inc**. Zwraca nową wartość **nice**. Jedynie proces należący do roota może zmniejszyć wartość nice, czyli podwyższyć priorytet procesu.

```
#include <sys/time.h>
```

```
#include <sys/resource.h>
```

```
int getpriority (int which, int who);
```

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

Funkcje operują na procesie, grupie procesów o zadany identyfikatorze bądź grupie procesów należących do danego użytkownika (argument **which** odpowiednio PRIO_PROCESS, PRIO_PGRP, or PRIO_USER). Argument **who** zawiera odpowiedni identyfikator. Funkcja **get...** zwraca najwyższy priorytet procesu ze wskazanej grupy, funkcja **set...** ustawia nowe priorytety (ograniczenia podobne jak w przypadku f-cji **nice**).

Sygnały

- Procesy
- **Sygnały**
- Pamięć

Sygnały

- Zdarzenia asynchroniczne i wyjątki
- Sygnały generowane są za pomocą funkcji systemowej `kill()`
- Operacje domyślne lub specyficzne programy obsługi napisane przez użytkownika
- Ustawiają bit w masce sygnałów procesu w tablicy procesów

Sygnały

- **Sygnal**: wiadomość, którą proces może przesłać do procesu lub grupy procesów, jeśli tylko ma odpowiednie uprawnienia (zdarzenia asynchroniczne i wyjątki).
- Typ wiadomości reprezentowany jest przez **nazwę symboliczną**
- Dla każdego sygnału otrzymujący go proces może:
 - Jawnie **zignorować** sygnał (bit w masce sygnałów w tablicy procesów nie jest ustawiany – brak śladu odebrania sygnału)
 - Realizować specjalne działania za pomocą tzw. **signal handler** (np. funkcja użytkownika)
 - W przeciwnym przypadku realizowane jest **działanie domyślne** (zwykle proces kończony)
- Wybrane sygnały mogą być również **blokowane**, tzn. ich nadejście jednorazowo jest odznaczane w masce sygnałów w tablicy procesów, ale obsługa jest odkładana do momentu zdjęcia blokady.

Przykład sygnałów 1/3

- Po zakończeniu potomka wysyłany jest sygnał SIGCHLD do rodzica (wartość 20, 17 lub 18 zależna od architektury – dla **i386** i **ppc** 17).
- Jeśli rodzic chce czekać na zakończenie potomka, to powiadamia system, że chce przechwycić sygnał SIGCHLD
- Jeśli nie zasygnalizuje oczekiwania, to sygnał SIGCHLD jest przez niego ignorowany (standardowa obsługa)
- W większości systemów dokładny opis obsługiwanych sygnałów wraz z ich domyślną obsługą i dodatkowymi informacjami umieszczany jest w **7 manualu signal** (**man 7 signal**)

Przykład sygnałów 2/3

- **SIGINT** Sygnał przerwania generowany zwykle, gdy użytkownik naciśnie klawisz **przerwania** na terminalu.
- **SIGQUIT** Sygnał przerwania generowany gdy użytkownik naciśnie na terminalu klawisz zakończenia pracy. Sygnał **SIGQUIT** jest podobny do sygnału **SIGINT**, ale dodatkowo generuje **obraz pamięci**.
- **SIGKILL** Bezwarunkowe zakończenie procesu (proces odbierający ten sygnał **nie może** go ani zignorować, ani przechwycić).
- **SIGTSTP** Sygnał wysyłany do procesu po naciśnięciu klawisza zawieszenia (na ogół CTRL+Z) lub klawisza zawieszenia z opóźnieniem (CTRL+Y). Proces zawieszony (zatrzymany), można wznowić sygnałem SIGCONT.
- **SIGILL** Sygnał ten jest generowany po wystąpieniu wykrywanej sprzętowo sytuacji wyjątkowej, spowodowanej przez niewłaściwą implementację systemu.
- **SIGSTOP** Sygnał ten zatrzymuje proces. Podobnie jak sygnał **SIGKILL** **nie może** zostać zignorowany lub przechwycony. Działanie zatrzymanego procesu można wznowić sygnałem **SIGCONT**.

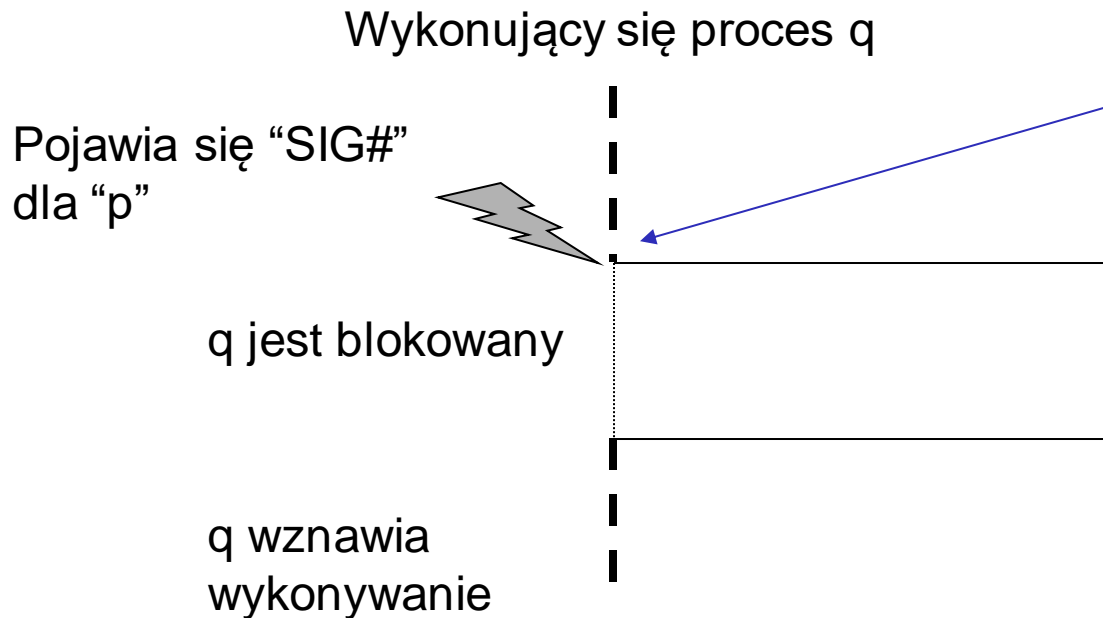
Przykład sygnałów 3/3

- **SIGSEGV** Sygnał generowany po wystąpieniu **błędu sprzętowego** spowodowanego przez niewłaściwą implementację systemu. Sygnał naruszenia segmentacji pojawia się na ogół wtedy, kiedy proces odnosi się do takiego adresu w pamięci, do którego nie ma dostępu.
- **SIGALARM** Sygnał budzika generowany przez f-cję `unsigned int alarm(unsigned int sec);`
- **SIGCHLD** Sygnał wysyłany do procesu, kiedy jego potomny proces się skończył.
- **SIGUSR1, SIGUSR2** Sygnały definiowane przez użytkownika, których można używać do komunikacji między procesami.
- **SIGTERM** Domyślny sygnał wysyłany przez komendę **kill**. Wymusza zawieszenie procesu.

Obsługa sygnałów

```
/* kod procesu p */  
.  
.  
.  
signal(SIG#, sig_hndlr);  
.  
.  
.  
/* DOWOLNY KOD */
```

```
void sig_hndlr(...) {  
    /* DOWOLNY KOD */  
}
```



tzw. **wolne funkcje** systemowe (w odróżnieniu od f-cji szybkich) mogą zostać przerwane i zwrócić błąd EINTR

sig_hndlr wykonuje się w przestrzeni adresowej procesu p

Wolne funkcje systemowe

- „*Funkcje systemowe podzielono na funkcje wolne i szybkie. Z grubsza wolne funkcje systemowe to te, które mogą długo wstrzymywać proces (np. read z konsoli), a szybkie to te, które nie będą wstrzymywać procesów długo (read z pliku).*”
- Przykłady wolnych funkcji systemowych:
 - blokujące wywołanie `read` na pustym potoku
 - `pause`
 - `wait`
- Wolne funkcje systemowe są **przerywane** przez sygnały. W zależności od wersji systemu:
 - kończą się one wówczas zwracając `-1` i przekazując w wyniku błąd **EINTR**
 - lub po obsłudze sygnału są automatycznie wznowiane.
- POSIX nie określa, które z powyższych zachowań jest słuszne. W systemie Linux domyślne jest zachowanie pierwsze, możemy jednak zmienić je na drugie używając flagi **SA_RESTART** w funkcji `sigaction`.

Wysyłanie sygnałów: `kill` 1/2

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

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

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

Funkcja systemowa `kill` może służyć do przesłania dowolnego sygnału do dowolnego procesu lub do dowolnej grupy procesów.

Jeśli ***pid*** ma **wartość dodatnią**, to sygnał ***sig*** jest przesyłany do procesu ***pid***.

Jeśli ***pid*** jest **równy 0**, to ***sig*** jest przesyłany do wszystkich procesów należących do tej samej grupy, co proces bieżący.

Jeśli ***pid*** jest **równy -1**, to sygnał jest przesyłany do wszystkich procesów, oprócz procesu nr 1 (init).

Jeśli ***pid*** jest **mniejszy niż -1**, to sygnał jest przesyłany do wszystkich procesów należących do grupy procesów o numerze ***-pid***.

Wysyłanie sygnałów: `kill` 2/2

Linux pozwala procesowi wysłać sygnał **do samego siebie**, ale wywołanie `kill(-1,sig)` pod Linuksem nie powoduje wysłania sygnału do bieżącego procesu.

Aby proces miał prawo wysłać sygnał do procesu *pid*:

- Proces wysyłający musi mieć uprawnienia roota, albo
- **rzeczywisty lub efektywny ID użytkownika** procesu wysyłającego musi być równy rzeczywistemu ID lub zachowanemu set-UID procesu otrzymującego sygnał.

Zmiana obsługi sygnału: `signal` 1/2

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

```
typedef void (*sig_handler_t) (int);
```

```
sig_handler_t signal(int signum, sig_handler_t handler);
```

Funkcja instaluje nową obsługę sygnału **signum**. Obsługa sygnału ustawiana jest na **handler**, który może być funkcją podaną przez użytkownika lub **SIG_IGN** albo **SIG_DFL**.

Po przyjściu sygnału do procesu:

- jeśli obsługa odpowiedniego sygnału została ustawiona na **SIG_IGN**, to sygnał jest ignorowany.
- jeśli obsługa została ustawiona na **SIG_DFL**, to podejmowana jest domyślna akcja skojarzona z sygnałem.
- jeśli jako obsługa sygnału została ustawiona funkcja **sig_handler** to wywoływana jest funkcja **sig_handler** z argumentem **signum**.

Sygnały **SIGKILL** i **SIGSTOP** nie mogą być ani przechwycone, ani zignorowane.

Funkcja zwraca poprzednią wartość obsługi sygnału, lub **SIG_ERR** w przypadku błędu.

Zmiana obsługi sygnału: `signal` 2/2

test-signal-1.c

```
...  
void (*f)( int);  
f=signal(SIGINT,SIG_IGN); /* ignorowanie sygnału sigint*/  
signal(SIGINT,f); /*przywrócenie poprzedniej reakcji na syg.*/  
signal(SIGINT,SIG_DFL); /*ustaw. standardowej reakcji na syg.*/  
...
```

test-signal-2.c

```
void moja_funkcja(int s) {  
    printf("Został przechwycony sygnał %d\n", s); return 0; }  
  
main() {  
    signal(SIGINT, moja_funkcja); /* przechwycenie sygnału */  
    ...  
}
```

Zmiana obsługi sygnału: `sigaction` 1/4

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

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

Wywołanie systemowe używane do zmieniania akcji, którą obiera proces po odebraniu określonego sygnału. ***signum*** określa sygnał i może być dowolnym prawidłowym sygnałem poza **SIGKILL** i **SIGSTOP**. Jeśli ***act*** jest niezerowe, to nowa akcja dla sygnału ***signum*** jest brana z ***act***. Jeśli ***oldact*** też jest niezerowe, to poprzednia akcja jest w nim zachowywana.

```
struct sigaction {  
    void (*sa_handler) (int);  
    void (*sa_sigaction) (int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_restorer) (void);  
}
```


Zmiana obsługi sygnału: `sigaction` 2/4

`sa_handler` podaje akcję, związaną z sygnałem *signal* i może to być m.in. **SIG_DFL** dla akcji domyślnej, **SIG_IGN** dla akcji ignorowania lub wskaźnik do funkcji obsługującej sygnał. Funkcja ta ma tylko jeden argument, w którym będzie przekazany numer sygnału.

`sa_sigaction` podaje akcję zamiast *sa_handler* jeżeli w *sa_flags* ustawiono **SA_SIGINFO**. Funkcja ta otrzymuje numer sygnału jako pierwszy argument, wskaźnik do *siginfo_t* jako drugi argument oraz wskaźnik do *ucontext_t* (zrzutowany na void *) jako jej trzeci argument.

`sa_mask` podaje maskę sygnałów, które powinny być blokowane podczas wywoływania handlera sygnałów. Dodatkowo, sygnał, który wywołał handler będzie zablokowany, chyba że użyto flagi **SA_NODEFER**.

`sa_flags` podaje zbiór flag, które modyfikują zachowanie procesu obsługi sygnałów. Jest to zbiór wartości połączonych bitowym OR (np. flaga **SA_RESETHAND** odtwórz akcję sygnałową do stanu domyślnego po wywołaniu handlera sygnałów a **SA_SIGINFO** określa, że handler sygnałów pobiera 3 argumenty, a nie jeden, natomiast **SA_RESTART** pozwala na automatyczne wznowienie przerwanej sygnałem *wolnej funkcji* systemowej).

Zmiana obsługi sygnału: `sigaction` 3/4

Parametr `siginfo_t` z `sa_sigaction` jest strukturą zawierającą następujące elementy:

```
siginfo_t {  
int si_signo; /* Numer sygnału */  
int si_errno; /* Wartość errno */  
int si_code; /* Kod sygnału */  
pid_t si_pid; /* Id procesu wysyłającego */  
uid_t si_uid; /* Rzeczywisty ID użytkownika wysyłającego  
procesu */  
int si_status; /* Kod zakończenia lub sygnał */  
clock_t si_utime; /* Czas spędzony w przestrzeni użytkownika */  
clock_t si_stime; /* Czas spędzony w przestrzeni systemu */  
sigval_t si_value; /* Wartość sygnału */  
int si_int; /* sygnał POSIX.1b */  
void * si_ptr; /* sygnał POSIX.1b */  
void * si_addr; /* Adres pamięci, który spowodował błąd */  
int si_band; /* Zdarzenie grupy (band event) */  
int si_fd; /* Deskryptor pliku */ }
```

Zmiana obsługi sygnału: `sigaction` 4/4

`test-sigaction-1.c`

```
void obslugaint(int s) { printf("... nie przerwiesz!\n"); }

int main(void)
{
    int x = 1;
    sigset_t iset;
    struct sigaction act;

    sigemptyset(&iset);
    act.sa_handler = &obslugaint;
    act.sa_mask = iset;
    act.sa_flags = 0;
    sigaction(SIGINT, &act, NULL);

    while (x != 0)
    {
        printf("Skoncze sie dopiero kiedy wprowadzisz 0\n");
        scanf("%d", &x);
    }
    return 0;
}
```

Blokowanie sygnałów: `sigprocmask` 1/2

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

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

Funkcja zmienia maskę blokowanych sygnałów procesu. Jej zachowanie zależy od ustawionej opcji **how**:

SIG_BLOCK dodanie do aktualnej maski sygnałów z zestawu **set**

SIG_UNBLOCK usunięcie z aktualnej maski sygnałów z zestawu **set**

SIG_SETMASK ustawienie aktualnej maski na zbiór sygnałów z zestawu **set**

Jeżeli **oldset** nie jest ustawione na **NULL**, to jest pod nim zapisywany zestaw sygnałów sprzed zmiany.

Jeżeli **set** jest ustawione na **NULL**, to maska sygnałów pozostaje niezmienną, ale ustawienia aktualnej maski są zapisywane w **oldset** (o ile różne od **NULL**).

Blokowanie sygnałów: `sigprocmask` 2/2

`test-sigprocmask.c`

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

int main(void)
{
    int i = 0;
    sigset_t iset;

    sigemptyset(&iset);
    sigaddset(&iset, SIGALRM);
    sigaddset(&iset, SIGINT);
    sigprocmask(SIG_BLOCK, &iset, NULL);
    alarm(5);
    while (1)
        printf("%d\n", i++);
    return 0;
}
```

Zbiory sygnałów 1/2

Uwaga! Poniższe funkcje służą tylko do grupowania zestawów sygnałów. Ich wywołanie nie powoduje żadnych zmian w obsłudze sygnałów, wysłania sygnałów czy ich blokowania.

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

Funkcja ustawia pusty zbiór sygnałów **set**

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

Funkcja ustawia kompletny zbiór sygnałów **set**

```
int sigaddset(sigset_t *set, int signum);
```

Funkcja dodaje do zbioru sygnałów **set** sygnał **signum**

```
int sigdelset(sigset_t *set, int signum);
```

Funkcja usuwa ze zbioru sygnałów **set** sygnał **signum**

```
int sigismember(const sigset_t *set, int signum);
```

Funkcja sprawdza czy sygnał **signum** jest zawarty we wskazanym zbiorze **set**.

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

Funkcja zwraca do zmiennej **set** zbiór sygnałów, które zostały wysłane do procesu i zablokowane.

Zbiory sygnałów 2/2

test-block-sig.c

```
...

sigset_t empty, set1, set2, set3;
sigemptyset(&empty);
sigemptyset(&set1); sigaddset(&set1, SIGUSR1);

sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

sigprocmask(SIG_SETMASK, &set1, NULL);

sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

kill(getpid(), SIGUSR1);
sigpending(&set2);
sigprocmask(SIG_BLOCK, &empty, &set3);
printf("%d %d\n", sigismember(&set2, SIGUSR1), sigismember(&set3, SIGUSR1));

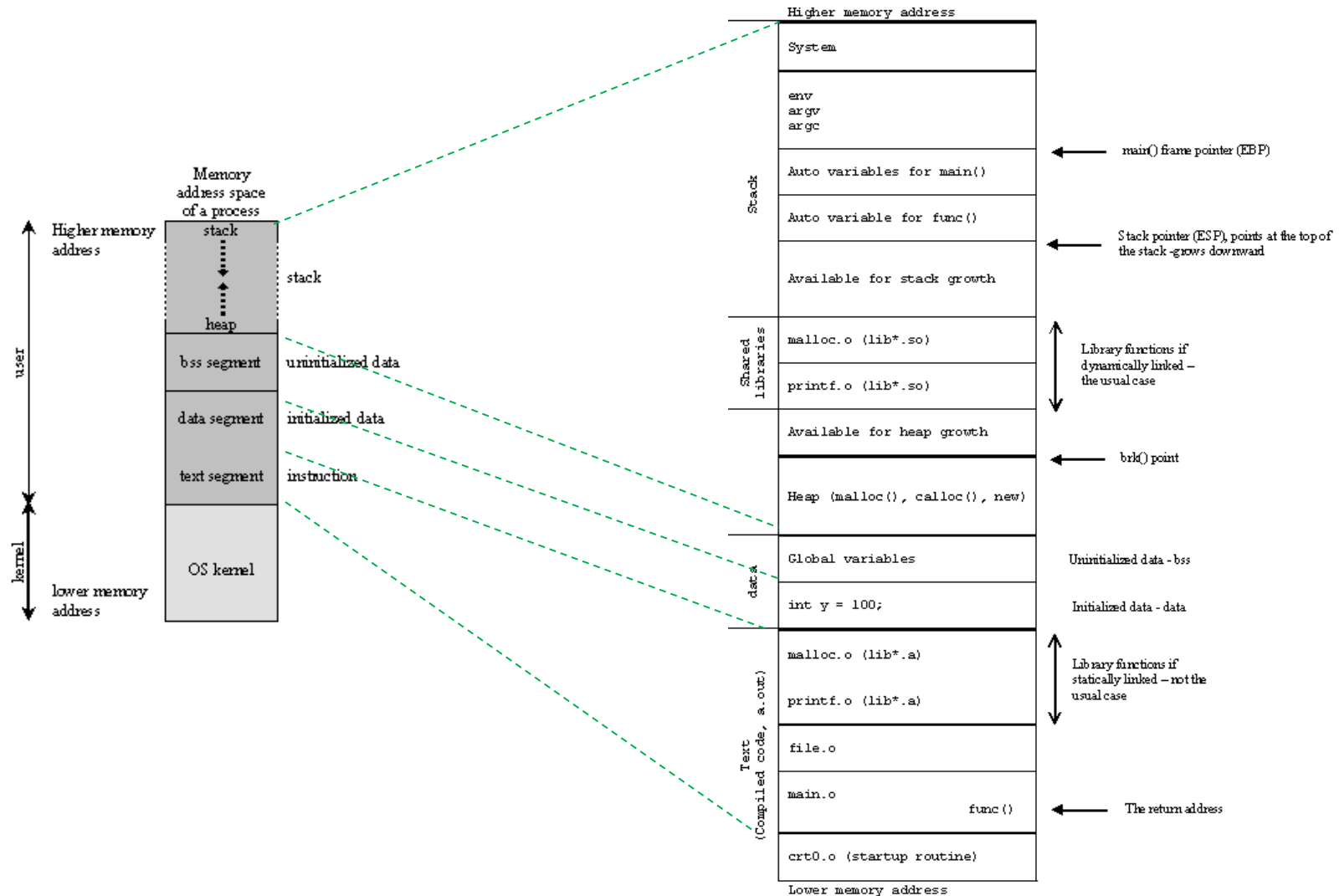
...
```

```
$/test-block-sig
0 0
0 1
1 1
```

Pamięć

- Procesy
- Sygnały
- **Pamięć**

Pamięć procesu



Alokacja na sterpie

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

```
void* malloc (size_t size);
```

Funkcja alokuje obszar pamięci o rozmiarze **size** bajtów. W przypadku powodzenia zwracany jest wskaźnik do początku zaalokowanego regionu (w przeciwnym przypadku NULL). Zawartość zaalokowanego obszaru jest nieokreślona (nie musi być wyzerowana).

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

```
void* calloc (size_t nr, size_t size);
```

Funkcja alokuje obszar pamięci, który pomieści **nr** elementów o rozmiarze **size**. W przypadku powodzenia zwracany jest wskaźnik do początku zaalokowanego regionu (w przeciwnym przypadku NULL). Zaalokowana pamięć jest wyzerowana.

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

```
void* realloc (void *ptr, size_t size);
```

Funkcja zamienia wielkość regionu pamięci wskazanego przez **ptr**, ustawiając go na **size** bajtów. W przypadku powodzenia zwraca nowy wskaźnik do regionu pamięci. Dotychczasowa zawartość regionu jest zachowana. Podanie wartości 0 jako drugiego argumentu powoduje, że f-cja zachowuje się tak jak **free**.

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

```
void free (void *ptr);
```

Funkcja zwalnia pamięć wskazaną przez **ptr**, zaalokowaną wcześniej np. przez **malloc**. Funkcja nie daje możliwości zwolnienia fragmentu pamięci.

Anonimowe odwzorowanie w pamięci

- **malloc** w bibliotece **glibc** używa sterty przy alokacji niewielkich obszarów pamięci (standardowo do 128kB)
- Przy alokacji większych obszarów wykorzystywany jest mechanizm **anonimowego odwzorowania w pamięci** (alokacja poza stertą, rozmiar jest całkowitą wielokrotnością strony systemowej, alokacja bardziej czasochłonna niż na stercie)
- Anonimowe odwzorowanie można wymusić za pomocą f-cji **mmap** (z parametrem **MAP_ANONYMOUS** bądź mapując plik **/dev/zero**) i zwolnić za pomocą **unmap**

```
#include <sys/mman.h>

void * mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap (void *start, size_t length);
```

anonymous-memory.c

```
...
void *p;
p = mmap (NULL, /* nieważne, w jakim miejscu pamięci */
          512 * 1024, /* 512 kB */
          PROT_READ | PROT_WRITE, /* zapis/odczyt */
          MAP_ANONYMOUS | MAP_PRIVATE, /* odwzorowanie anonimowe i prywatne */
          -1, /* deskryptor pliku (ignorowany) */
          0); /* przesunięcie (ignorowane) */
if (p == MAP_FAILED)
    perror ("mmap");
else
    /* 'p' wskazuje na obszar 512 kB anonimowej pamięci... */
...

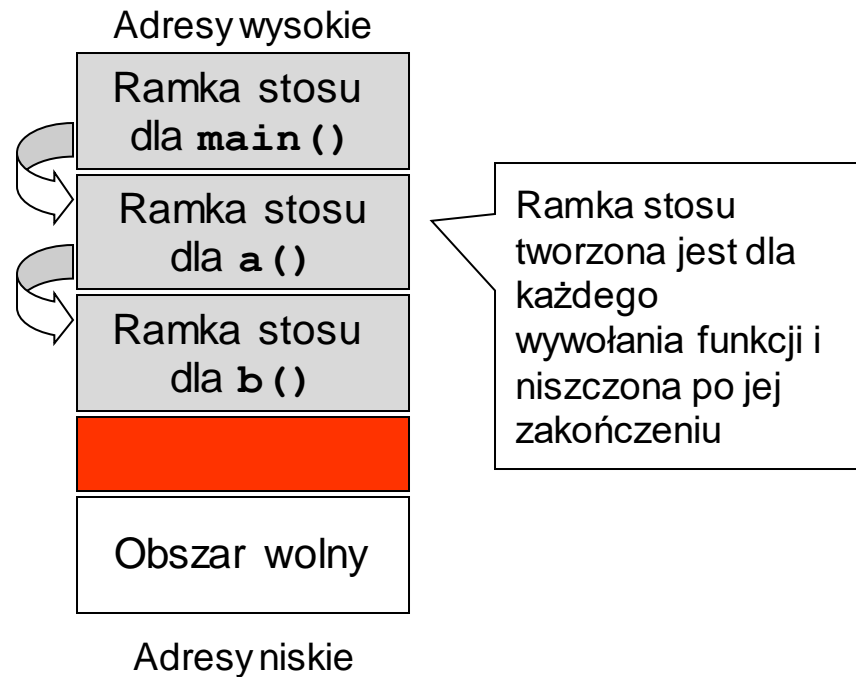
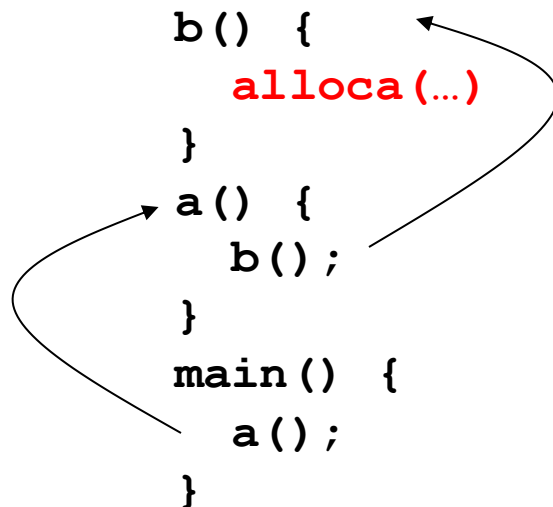
```

Alokacja na stosie

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

```
void* malloc (size_t size);
```

Funkcja alokuje obszar pamięci o rozmiarze **size** bajtów, ale nie na sterpie a na stosie. **Pamięci zaalokowanej w ten sposób nie zwalniamy manualnie!** Zostaje ona zwolniona automatycznie po zakończeniu wykonywania się funkcji, w której wywołano **malloc** (podczas zdejmowania ze stosu ramki tej funkcji i powrotu do funkcji wywołującej wskaźnik wierzchołka stosu przesuwa się do pozycji sprzed wywołania).



Tablice o zmiennej długości

- Zgodnie ze standardem **C99** można w języku C używać tablic o zmiennej długości, dla których pamięć alokowana jest automatycznie.
- Pamięć zaalokowana w ten sposób istnieje do momentu, gdy **zmienna**, która ją reprezentuje, znajdzie się **poza zakresem widoczności** (czyli potencjalnie wcześniej niż przy alokacji za pomocą **alloca**)

```
VLA.c
...
for (i = 0; i < n; ++i)
{
    char foo[i + 1];
    /* tu można użyć 'foo'... */
}
...
```

Alokacja 0 bajtów 1/3

- Zachowanie **malloc(0)** jest zależne od implementacji, najczęściej jedno z dwóch:
 - AIX, Solaris i Tru64 UNIX powinny zwrócić **wskaźnik NULL** podczas próby alokacji 0 bajtów
 - Darwin, FreeBSD, IRIX, Linux (z domyślnymi ustawieniami) i Windows zwrócą **wskaźnik różny od zera**, wskazujący na bufor zerowej długości
- ... ale jeszcze większe zamieszanie powoduje **realloc(...,0)**

Alokacja 0 bajtów 2/3

	returns	ptr	errno
AIX			
<code>realloc(NULL, 0)</code>	Always NULL		unchanged
<code>realloc(ptr, 0)</code>	Always NULL	freed	unchanged
BSD			
<code>realloc(NULL, 0)</code>	only gives NULL on alloc failure		ENOMEM
<code>realloc(ptr, 0)</code>	only gives NULL on alloc failure	unchanged	ENOMEM
glibc			
<code>realloc(NULL, 0)</code>	only gives NULL on alloc failure		ENOMEM
<code>realloc(ptr, 0)</code>	always returns NULL	freed	unchanged

Alokacja 0 bajtów 3/3

```
char *p2;  
char *p = malloc(100) ;  
...  
if ((p2 = realloc(p, 0)) == NULL) {  
    if (p)  
        free(p) ;  
    p = NULL;  
    return NULL;  
}  
p = p2;
```

... nie powinno się zwalniać pamięci
za pomocą **realloc**