

# PROCESY

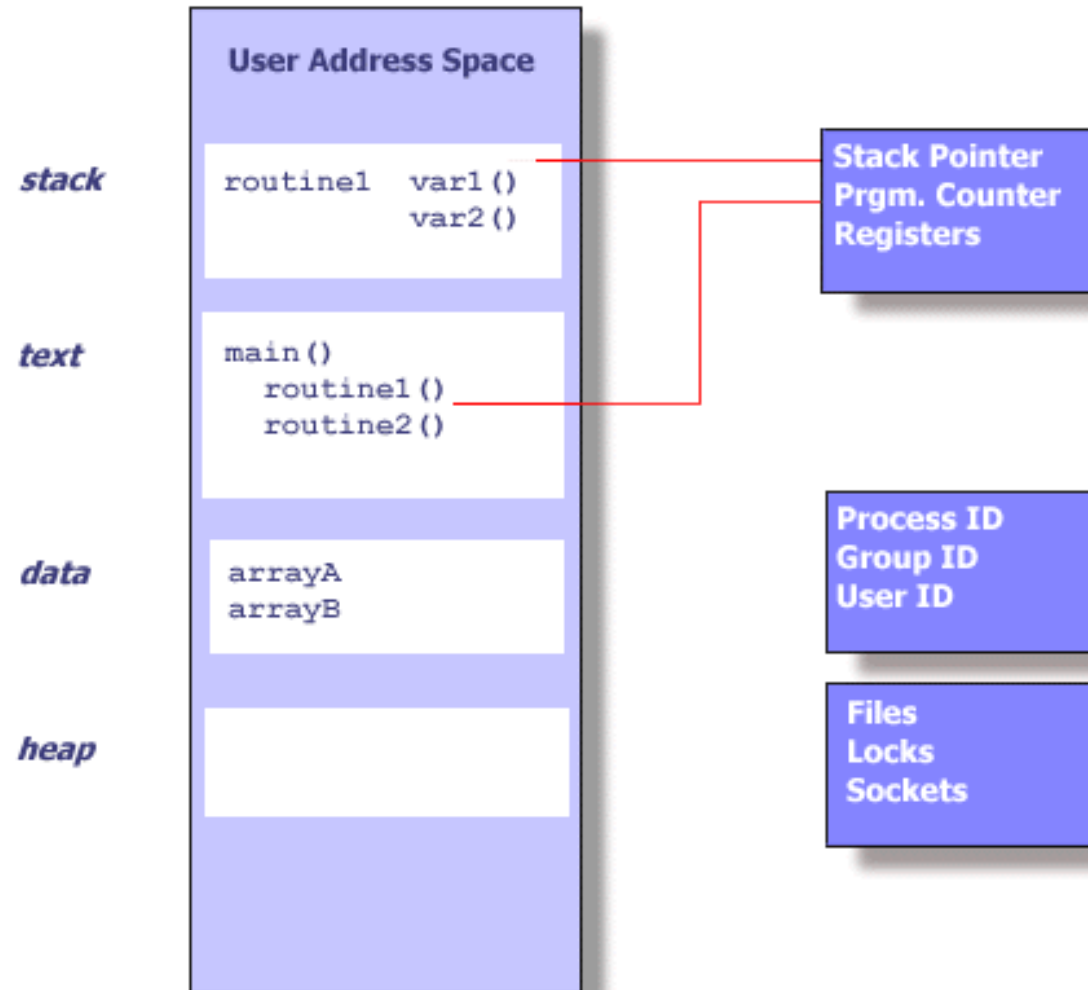
# Co to jest proces? 1/2

**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.:

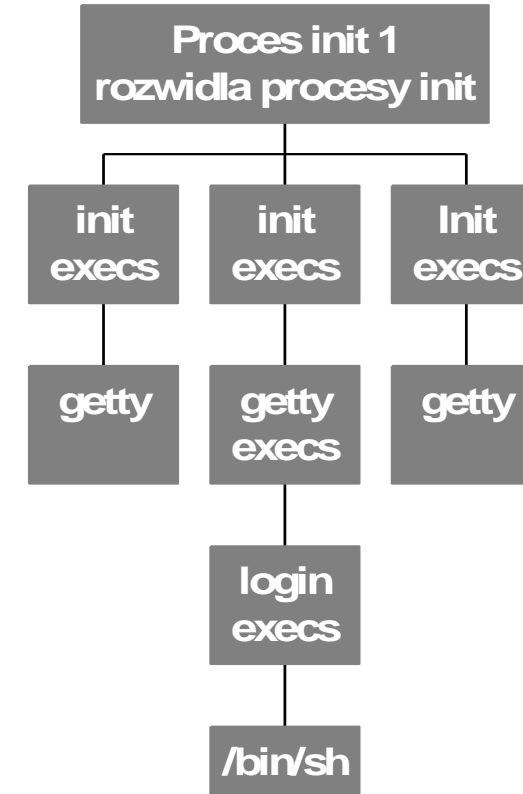
- identyfikator procesu, identyfikator grupy procesów, identyfikator użytkownika, identyfikator grupy użytkowników
- środowisko
- katalog roboczy
- kod programu
- rejestry
- stos
- sarta
- deskryptory plików
- akcje sygnałów
- biblioteki współdzielone
- narzędzia komunikacji międzyprocesowej

## Co to jest proces? 2/2



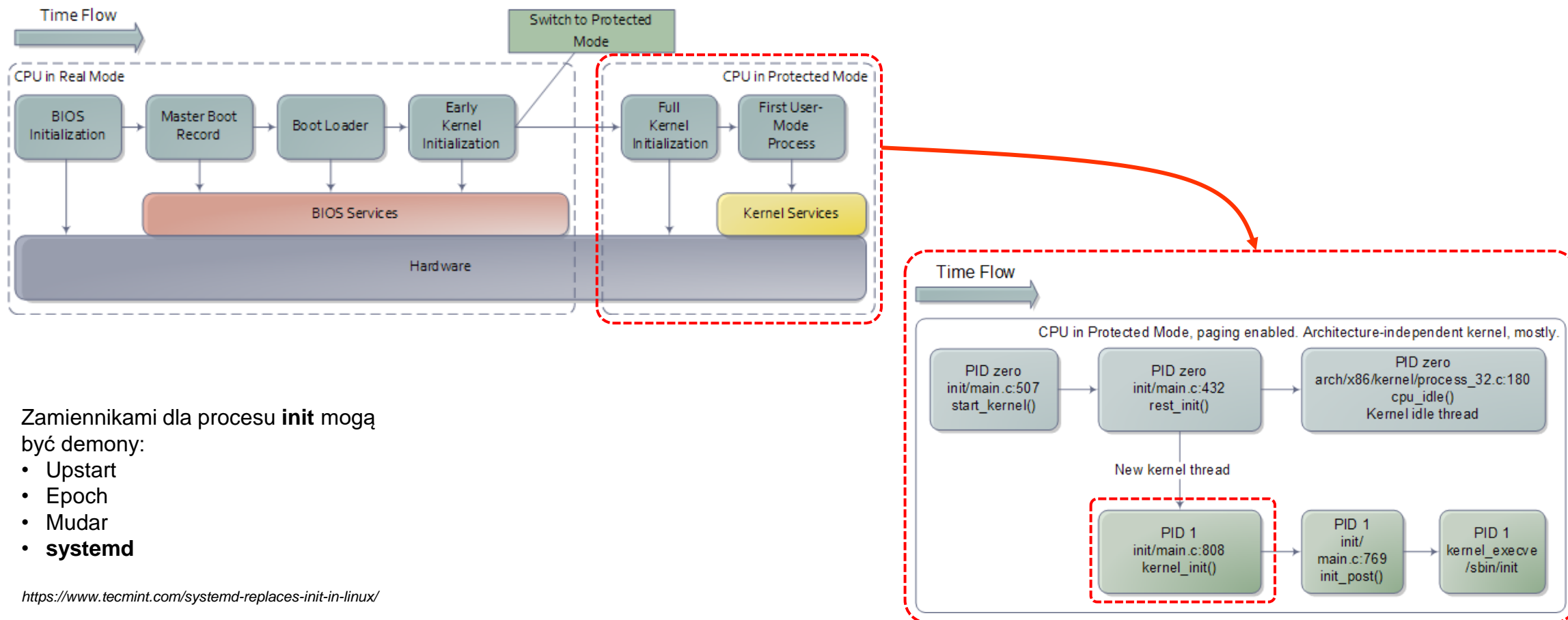
# Hierarchia procesów 1/3

- 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.
- Nowe identyfikatory nadawane liniowo, unikalne w pewnej jednostce czasu.
- Podczas życia procesu za pomocą wywołania funkcji systemowej **exec** można zmienić jego kod i dane
- Procesy kończą się zwykle po wywołaniu **exit**



# Hierarchia procesów 2/3

## Start systemu i „pierwszy” proces



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

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

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

# Hierarchia procesów 2/3

## systemd vs init

„**systemd** ... jest demonem, który został zaprojektowany do uruchamiania procesów równoległe, redukując w ten sposób czas startu systemu i związane z nim narzut obliczeniowy. Posiada wiele innych funkcji w porównaniu do **init**.”

- szybszy boot (równoległe uruchamianie serwisów)
- możliwość ponownego uruchamiania uszkodzonych serwisów
- integracja z SELinux
- automatyczne zarządzanie zależnościami między serwisami
- bardziej zaawansowane mechanizmy zbierania logów (mogą zastąpić lub rozszerzyć działanie syslog)
- Ale np. ... mniej modułowa budowa, cięższy kod

# Kontekst procesu 1/3

- 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

# Kontekst procesu 2/3

## 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.



# Kontekst procesu 3/3

## U-obszar i tablica Proc

### U-obszar

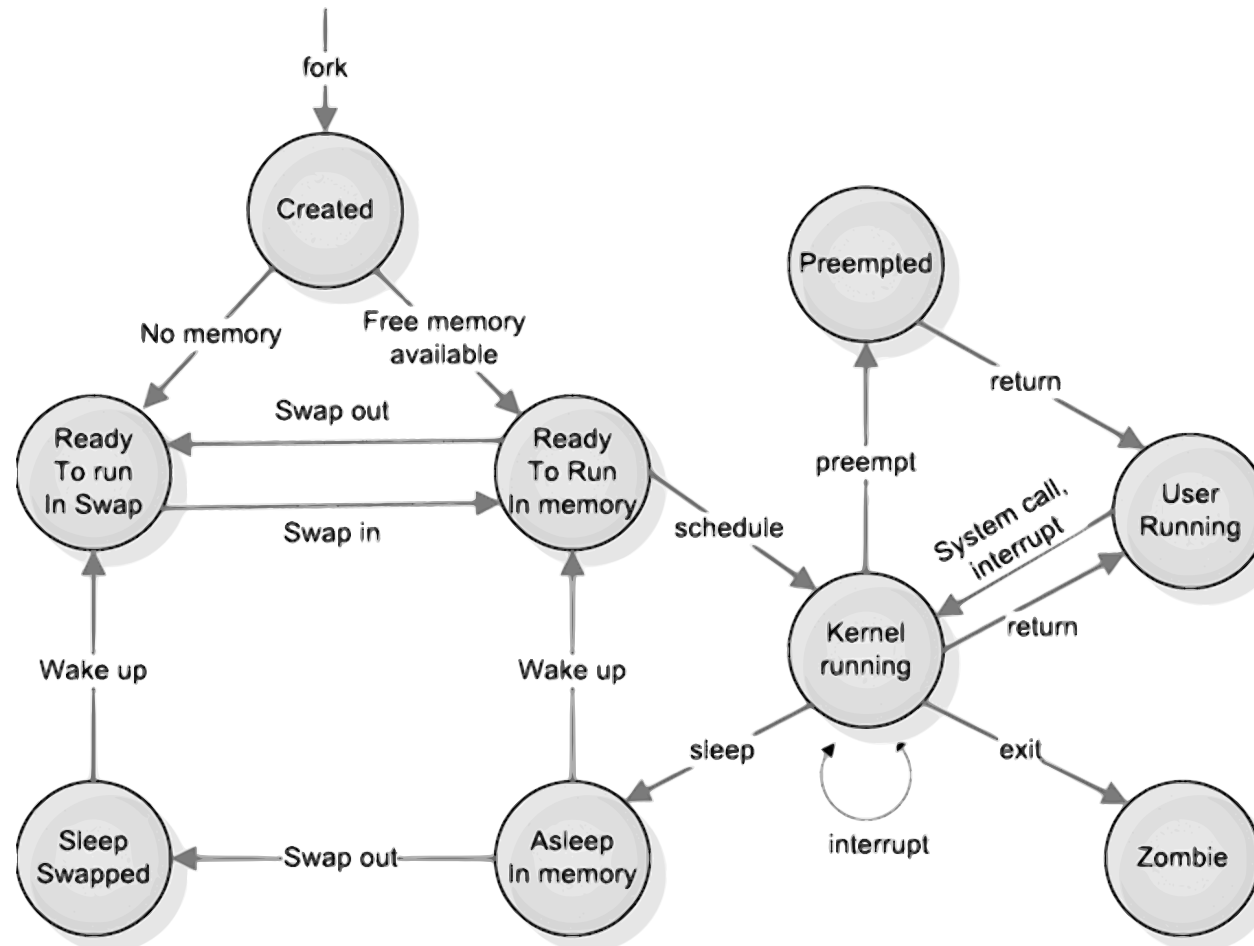
- **Wskaźnik** do pozycji w tablicy Proc
- Rzeczywisty/efektywny **UID**
- **argumenty**, zwracane wartości lub **błędy** bieżącego wywołania funkcji systemowej.
- Tablica reakcji na **sygnały**
- Tabela deskryptorów **plików**
- Bieżący katalog i bieżący korzeń.

### 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 1/2

## UNIX - podstawowe stany i tranzycje



# Uproszczony graf stanu procesów 2/2

## Linux – stany z punktu widzenia użytkownika

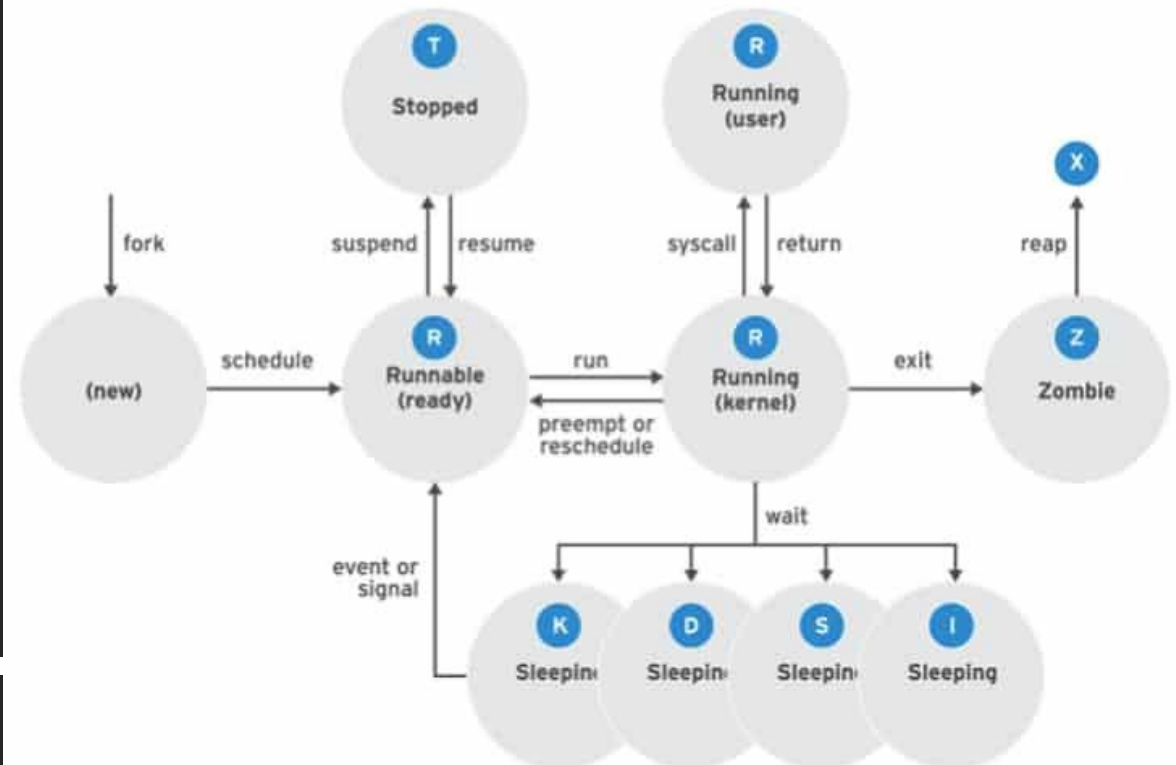
D uninterruptible sleep (usually IO) man ps  
 I Idle kernel thread  
 R running or runnable (on run queue)  
 S interruptible sleep (waiting for an event to complete)  
 T stopped by job control signal  
 t stopped by debugger during the tracing  
 W paging (not valid since the 2.6.xx kernel)  
 X dead (should never be seen)  
 Z defunct ("zombie") process, terminated but not reaped by its parent

For BSD formats and when the stat keyword is used, additional characters may be displayed:

< high-priority (not nice to other users)  
 N low-priority (nice to other users)  
 L has pages locked into memory (for real-time and custom IO)  
 s is a session leader  
 l is multi-threaded (using CLONE\_THREAD, like NPTL pthreads do)  
 + is in the foreground process group

```

$ ps -o "pid,stat,command"
PID STAT COMMAND
191 Ss -bash
290 S ./testWait
291 R ./testWait
376 R+ ps -o pid,stat,command
  
```

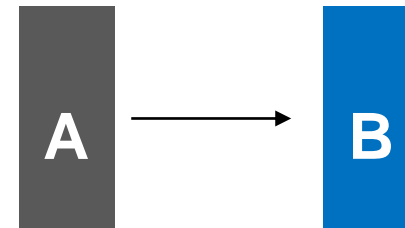
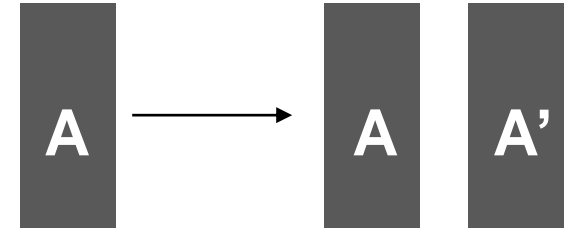


# Szeregowanie procesów Linux

- Od jądra 2.6.3 **CFS (Completely Fair Scheduler)**
- Punktem wyjścia nie są stałe kwanty czasu ale **TSL** (Target Scheduling Latency) – jest to jednostka czasu procesora dzielony proporcjonalnie między procesy (zależnie m.in. od ich priorytetów).
- **TSL** nie może być dzielne w nieskończoność – zabezpieczenie **minimal granularity**.
- Z każdym procesem związany jest **vruntime** (normalizowany czas dotychczasowego użycia procesora przez proces).
- Procesy gotowe do wykonania umieszczane są w drzewie czerwono czarnym (kluczami są wartości **vruntime**).
- Do wykonania wybierany jest najbardziej lewy węzeł (i usuwany z drzewa).

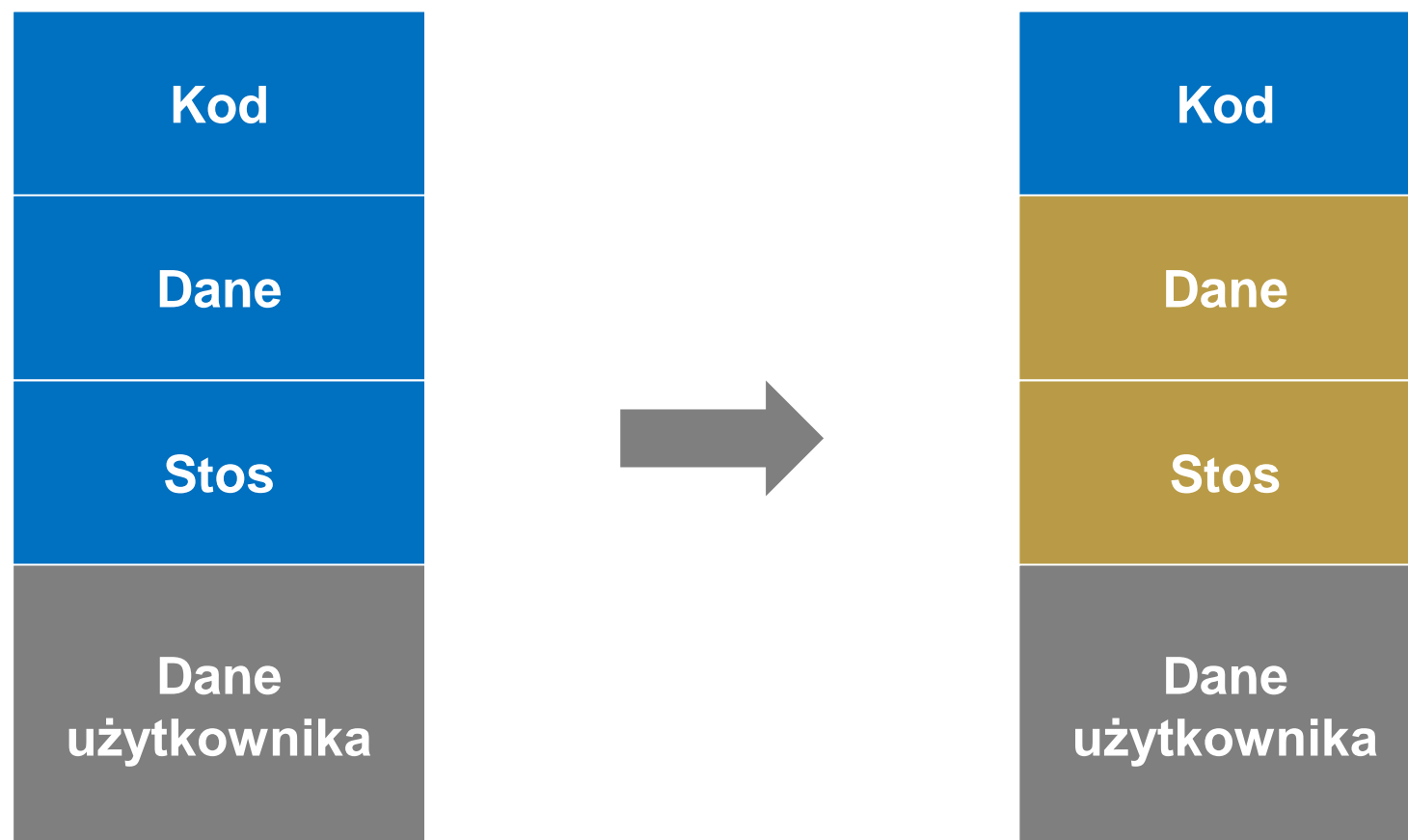
# Tworzenie procesu 1/3

- **fork** (klonuje aktualny proces)
  - 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** (zastępuje bieżący proces)
  - Nadpisuje istniejący proces nowym kodem z podanego programu



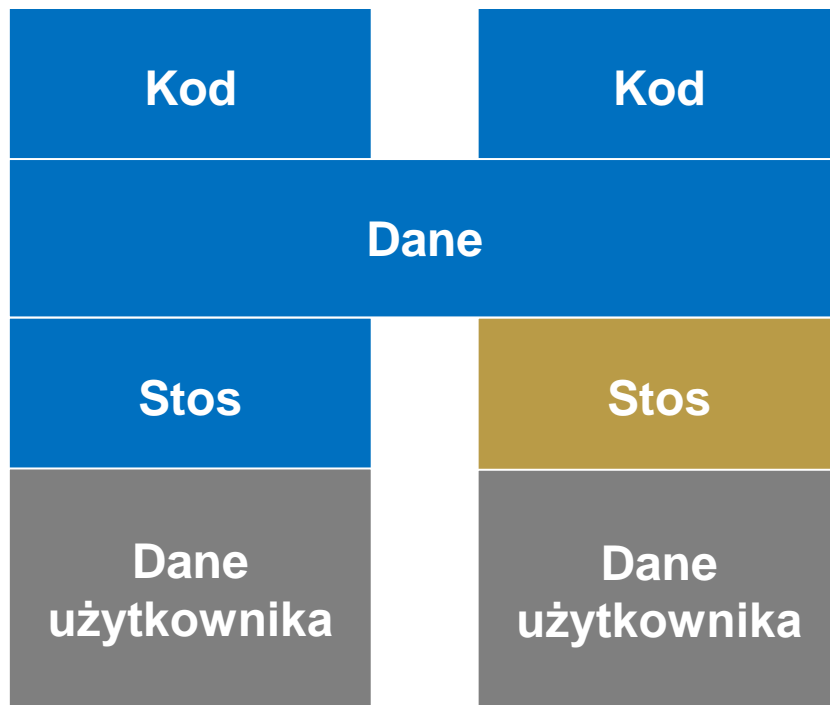
# Tworzenie procesu 2/3

Działanie fork

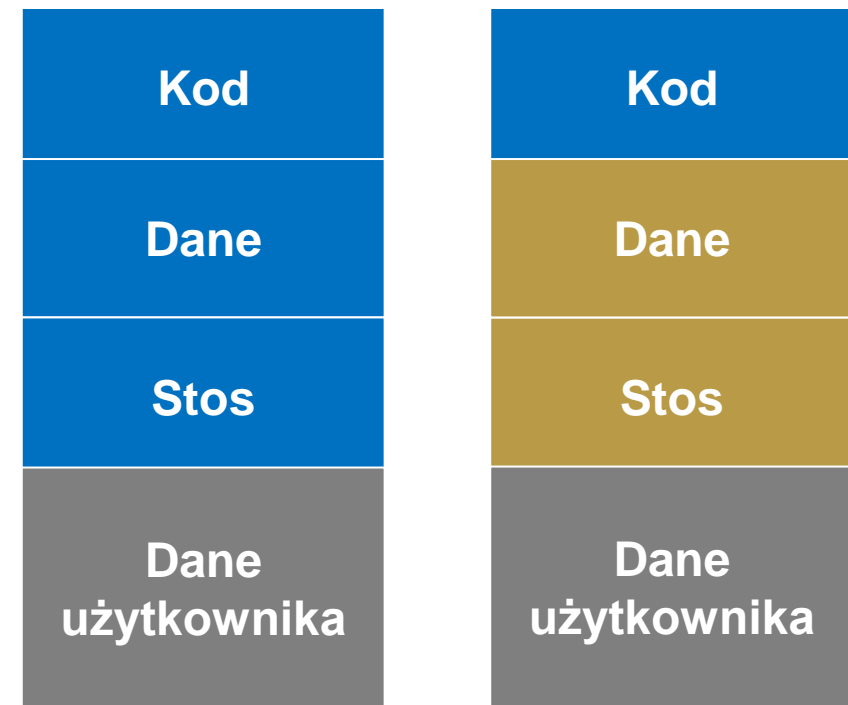


# Tworzenie procesu 2/3

Efektywna implementacja fork: kopiowanie po zapisie (**copy-on-write**)



Przed zapisem do obszaru **Dane**



Po zapisie do obszaru **Dane**

## Kończenie procesu

- Wywoływana jest funkcja `exit`
  - Zamyka otwarte pliki
  - Zwalnia inne zasoby
  - Zapisuje statystyki użytkowania 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 użytkowania zasobów



# 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);
```

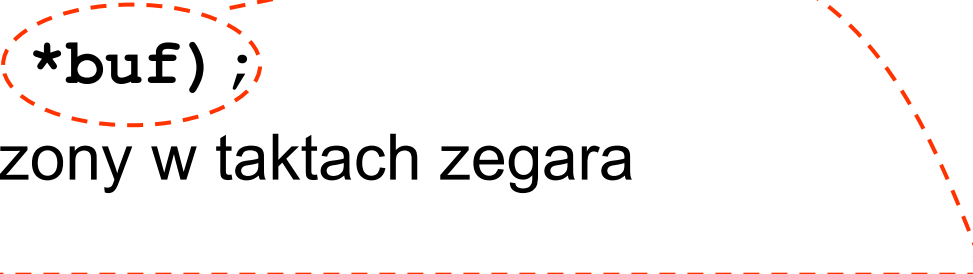
Zwraca efektywny ID grupy bieżącego procesu

# 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 */  
};
```

A dashed red oval highlights the `*buf` parameter in the `times` function signature. A dashed red arrow originates from this oval and points to the `struct tms` definition, which is enclosed in a larger dashed red box.

# Tworzenie nowego procesu

## Funkcja 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 używa w tym systemie wywołania **clone**.

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

# Tworzenie nowego procesu

## Funkcja 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

## Funkcja fork 3/3

Ppid: 99

Pid: 100

child\_pid: 101

### 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;
}
```

PC/IP

Ppid: 100

Pid: 101

child\_pid: 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;
}
```

PC/IP

# Tworzenie nowego procesu

## Funkcja 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**. Czyli obecnie właściwie niewielkie ...

# Tworzenie nowego procesu

## 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.



# Tworzenie nowego procesu

## 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** ani **pamięci** ustalonych w procesie macierzystym,
- w procesie potomnym jest zerowany **zbiór zaległych sygnałów**.

# Uruchamianie programów

## Funkcja 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.

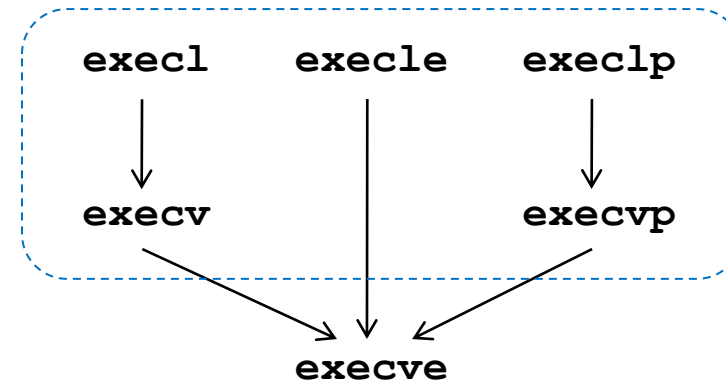
# Uruchamianie programów

## Funkcja 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

## Funkcja 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

## Funkcja 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

## Funkcja 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

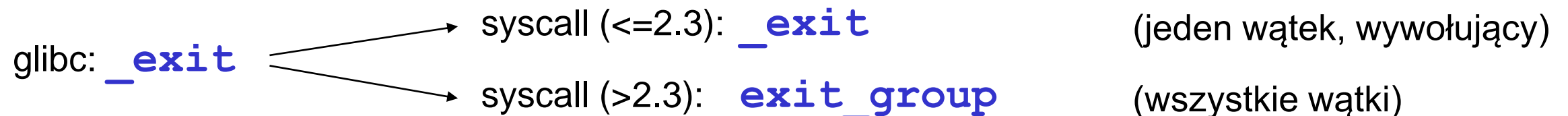
## Funkcja `_exit`

```
#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 funkcji zarejestrowanych za pomocą funkcji `atexit`



# Kończenie procesu

## Funkcja 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.

Alternatywa: **on\_exit**



# Kończenie procesu

## Funkcja atexit 1/2

### 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

## Funkcja wait

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

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

Funkcja zatrzymuje wykonywanie bieżącego procesu aż do zakończenia procesu potomka (sygnał SIGCHLD) 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

## Funkcja 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.

alternatywa: `waitid`

przestarzałe: `wait3`, `wait4`

# Synchronizacja procesów

## Funkcja waitpid 1/2

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**.

# Synchronizacja procesów

## 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**.*

# Synchronizacja procesów

## `wait` a zmiana stanu procesu potomnego 1/2

`wait, waitpid, waitid`

Sygnał **SIGCHLD** (lub inny zarejestrowany przy wywołaniu `clone`)

Inne sygnały, które trafią do procesu podczas działania jednej z funkcji **`wait`**

Funkcja kończy się poprawnie w związku ze **zmianą stanu** procesu potomnego.

Funkcja kończy się błędem (bez związku ze stanem procesu potomnego).

- Zmianą stanu może być nie tylko przejście do stanu ZOMBIE, ale również przejście w stan T (wstrzymanie, debug) lub powrót do wykonania.
- Żeby śledzić również takie zmiany musimy ustawić dodatkowe flagi (`waitpid, waitid`): **WUNTRACED, WCONTINUED**

# Synchronizacja procesów

## `wait` a zmiana stanu procesu potomnego 2/2

```
int main() {
    if( !fork()) {
        printf("cpid: %d\n", getpid());
        while(1);
    }
    int s;
    int cpid;
    do {
        cpid = waitpid(-1, &s, WUNTRACED | WCONTINUED);
        printf("on wait > cpid: %d, stopped: %d, continued: %d, exited: %d, signaled: %d, termSig: %d, stopSig %d \n",
               cpid, WIFSTOPPED(s), WIFCONTINUED(s), WIFEXITED(s), WIFSIGNALED(s), WTERMSIG(s), WSTOPSIG(s));
    } while(cpid == -1 || (WIFEXITED(s) == 0 && WIFSIGNALED(s) == 0));
    return 0;
}
```

```
$ testWait&
cpid: 649

$ kill -SIGTSTP 649
on wait > cpid: 649, stopped: 1, continued: 0, exited: 0, signaled: 0, termSig: 127, stopSig: 20

$ kill -SIGCONT 649
on wait > cpid: 649, stopped: 0, continued: 1, exited: 0, signaled: 0, termSig: 127, stopSig: 127

kill -SIGKILL 649
on wait > cpid: 649, stopped: 0, continued: 0, exited: 0, signaled: 1, termSig: 9, stopSig: 127
```

SIGSTOP, SIGTSTP

SIGCONT

SIGKILL, SIGTERM, ...

```

#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <unistd.h>
#include <stdio.h>

#if __WORDSIZE == 64
#define AX(reg) reg.orig_rax
#else
#define AX(reg) reg.orig_eax
#endif

int main( int argc, char* argv[]) {
    if ( argc > 1) {
        int cpid = fork();
        if( cpid == 0) {
            ptrace( PTRACE_TRACEME, 0, NULL, NULL);
            execlp( argv[1], argv[1], NULL);
        } else {

            int s;
            while( waitpid(cpid, &s, 0) && !WIFEXITED(s)) {
                struct user_regs_struct regs;
                ptrace( PTRACE_GETREGS, cpid, NULL, &regs);
                fprintf( stderr, "system call %llu from pid %d\n", AX(regs), cpid);
                ptrace( PTRACE_SYSCALL, cpid, NULL, NULL);
            }
        }
    }
    return 0;
}

```

# Synchronizacja procesów

## wait i ptrace

Jeżeli śledzimy działanie procesu za pomocą funkcji `ptrace`, to `wait` dodatkowo posłuży do oczekiwania na szczegółowe informacje o stanie procesu.

```
$ ./testPtrace1 ls
```

```

...
system call 5 from pid 1334      fstat
system call 5 from pid 1334      fstat
system call 1 from pid 1334      write
testPtrace testPtrace.c
system call 1 from pid 1334      fstat
system call 3 from pid 1334      close
system call 3 from pid 1334      close
system call 3 from pid 1334      close
system call 3 from pid 1334      close
system call 231 from pid 1334    exit_group

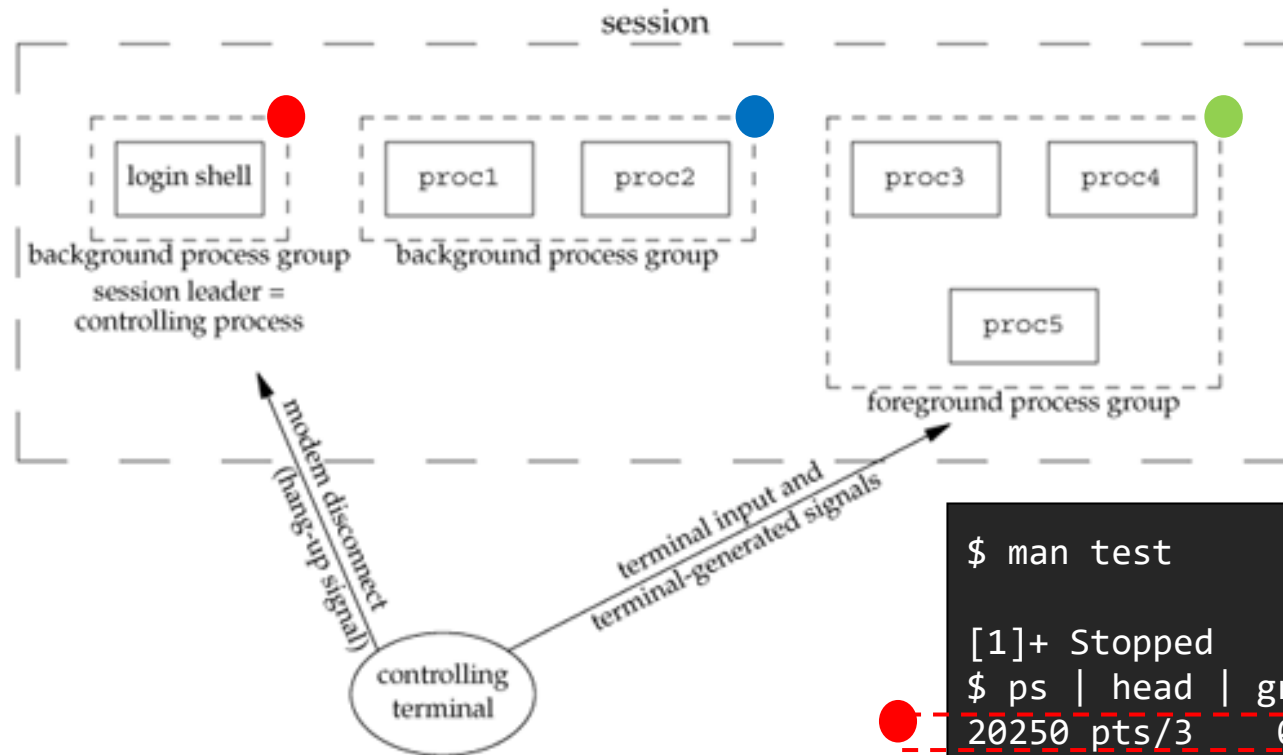
```



# 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 procesu wskazanego jako argument. Podanie argumentu **0** oznacza, że jesteśmy zainteresowani identyfikatorem sesji procesu bieżącego.

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

Przeniesienie 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.

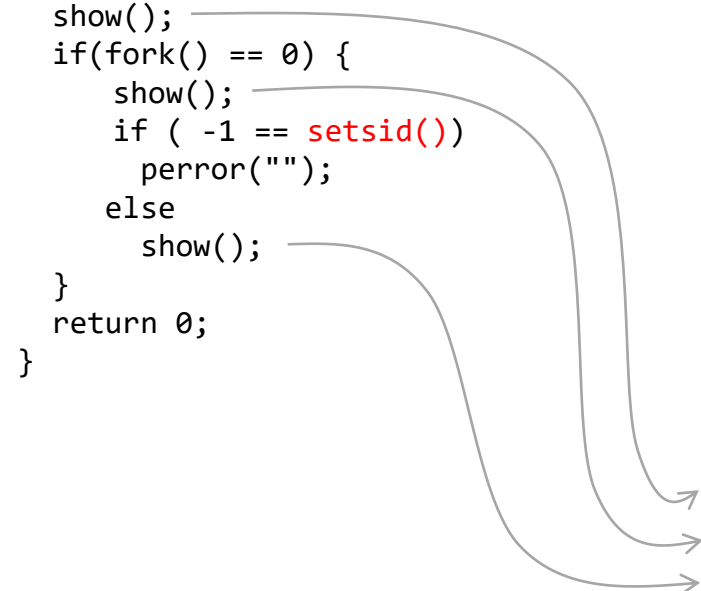
```
pid_t getpgid(pid_t pid);
```

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

# Grupowanie procesów 4/5

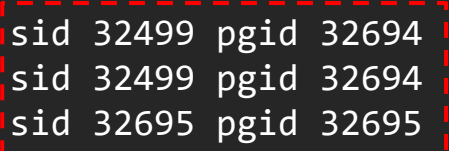
test-sid-1.c

```
...  
  
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;  
}
```

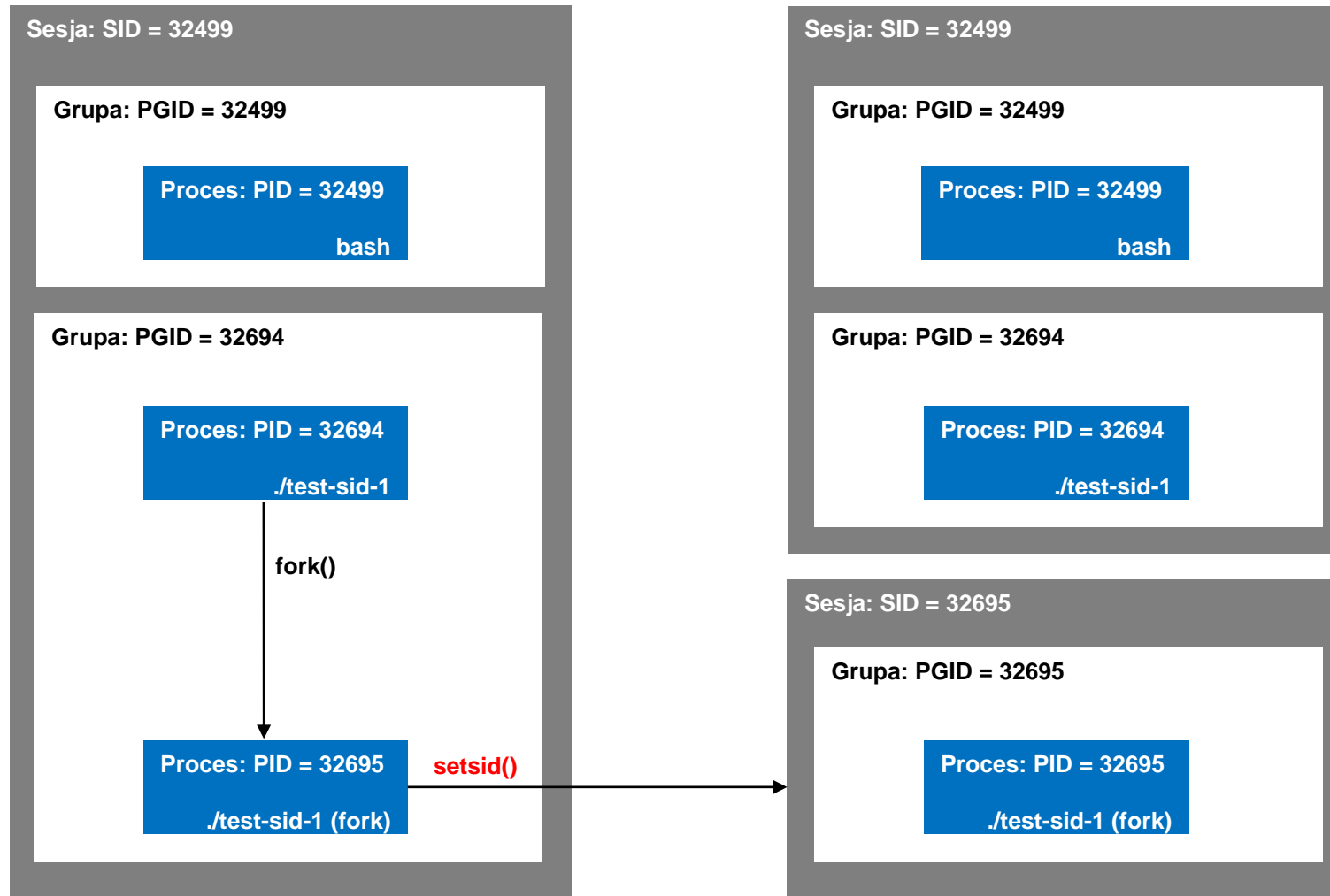


```
$ ps  
  PID TTY          TIME CMD  
 32499 pts/7        00:00:00 bash  
 32653 pts/7        00:00:00 ps
```

```
$ ./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



# Zamknięcie terminala 1/2

pts/0

```
$ ./testSighup 2> testSighup.txt
```

!!! Zamknięcie terminala pts/0

```
int main() {  
    while(1) {  
        sleep(1);  
        fprintf( stderr, "x\n");  
    }  
    return 0;  
}
```

pts/1

```
$ ps ax | grep testSighup  
1793 pts/3    S+   0:00 tail -f testSighup.txt  
1794 pts/0    S+   0:00 ./testSighup  
1796 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ ps ax | grep testSighup  
1793 pts/3    S+   0:00 tail -f testSighup.txt  
1745 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ ps ax | grep testSighup  
1704 pts/3    S+   0:00 tail -f testSighup.txt  
1748 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ less testSighup.txt  
...  
x  
x  
x  
x
```

Wysyłany sygnał **SIGHUP** do wszystkich procesów związanych z zamykanym terminaliem (sesja). Sygnał powoduje natychmiastowe przerwanie działania procesu, który go otrzymał.

# Zamknięcie terminala 1/2

pts/0

```
$ ./testSighup 2> testSighup.txt
```

```
!!! Zamknięcie terminala pts/0
```

```
void handler(int sigNo) {
    fprintf( stderr, "--- signal %d\n", sigNo);
}

int main() {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&(sa.sa_mask));
    sa.sa_flags = 0;

    sigaction(SIGHUP, &sa, NULL);

    while(1) {
        sleep(1);
        fprintf( stderr, "x\n");
    }

    return 0;
}
```

pts/1

```
$ ps ax | grep testSighup
1704 pts/3    S+   0:00 tail -f testSighup.txt
1741 pts/0    S+   0:00 ./testSighup
1743 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ ps ax | grep testSighup
1704 pts/3    S+   0:00 tail -f testSighup.txt
1741 ?         S    0:00 ./testSighup
1745 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ killall -SIGKILL testSighup
```

```
$ ps ax | grep testSighup
1704 pts/3    S+   0:00 tail -f testSighup.txt
1748 pts/1    S+   0:00 grep --color=auto testSighup
```

```
$ less testSighup.txt
...
x
--- signal 1
x
--- signal 1
x
```

# 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**).

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

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



wywołania systemowe bez implementacji w **glibc**