Procesy POSIX/UNIX/Linux

POSIX 1003.1

„**Process** – an address space with one or more threads executing within that address space, and the required system resources for those threads”.

Note: Many of the system resources defined by IEEE Std 1003.1-2001 are shared among all of the threads within a process. These include the process ID, the parent process ID, process group ID, session membership, real, effective, and saved set-user-ID, real, effective, and saved set-group-ID, supplementary group IDs, current working directory, root directory, file mode creation mask, and file descriptors.

Procesy w systemie Unix

- Proces jest wykonującym się programem.
- Procesy są rozróżniane za pomocą identyfikatorów procesów (*process identifier, PID*), które są liczbami całkowitymi.
- Każdy proces działa na rzecz (konta) użytkownika (o numerycznym identyfikatorze oznaczanym symbolicznie przez UID) oraz grupy użytkownika (identyfikator numeryczny oznaczany przez GID). UID i GID określają prawa dostępu do zasobów systemowych.
- W systemie Linux zdefiniowano „*personality identifier*”, który pozwala modyfikować semantykę funkcji systemowych (zgodność z wariantami Unixa)

Przykład. Wyświetlanie parametrów procesów użytkownika lopalski

$ **ps -lu lopalski**

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
8	O	1253	14457	14379	0	50	20	?	142		pts/24	0:00	ps
8	S	1253	14379	14364	0	50	20	?	423	?	pts/24	0:00	zsh

Procesy w systemie Unix – c.d.

Wszystkie procesy użytkowników są potomkami jednego pierwotnego procesu o PID=1 (tradycyjna nazwa: **init**, obecnie jest to zwykle **systemd**)

Przykład. Linux - procesy systemowe

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  1 16:40 ?        00:00:01 init [2]
root         2     1  0 16:40 ?        00:00:00 [ksoftirqd/0]
root         3     1  2 16:40 ?        00:00:02 [events/0]
root         4     3  0 16:40 ?        00:00:00 [khelper]
root         5     3  0 16:40 ?        00:00:00 [kacpid]
root        29     3  0 16:40 ?        00:00:00 [kblockd/0]
root        39     3  0 16:40 ?        00:00:00 [pdflush]
root        40     3  0 16:40 ?        00:00:00 [pdflush]
root        41     1  0 16:40 ?        00:00:00 [kswapd0]
root        42     3  0 16:40 ?        00:00:00 [aio/0]
.........
daemon    1419     1  0 16:40 ?        00:00:00 /sbin/portmap
root      1753     1  0 16:40 ?        00:00:00 /sbin/syslogd
root      1756     1  0 16:40 ?        00:00:00 /sbin/klogd
.........
root      1797     1  0 16:40 ?        00:00:00 /usr/sbin/inetd
lp        1801     1  0 16:40 ?        00:00:00 /usr/sbin/lpd -s
root      1808     1  0 16:40 ?        00:00:00 /usr/sbin/sshd

root      1857     1  0 16:40 tty2     00:00:00 /sbin/getty 38400 tty2
.........
```

Procesy w systemie Unix – c.d.

Liczba i nazwy procesów systemowych różnią się pomiędzy systemami.

Wyjątkiem jest proce o PID==1 (**init/systemd**), którego nazwa, PID i misja są ustalone

Przykład. SunOS 5.9 – wybrane procesy systemowe

```
    UID   PID  PPID  C    STIME TTY      TIME CMD
    root     0     0   0  04:52:16 ?      0:01  sched
    root     1     0   0  04:52:16 ?      0:03  /etc/init -
    root     2     0   0  04:52:16 ?      0:00  pageout
    root     3     0   1  04:52:16 ?     24:56  fsflush
...
   root   282     1   0  04:52:46 ?      0:01  /usr/sbin/rpcbind

    root   305     1   0  04:52:47 ?      0:00  /usr/sbin/inetd -s
    root   372     1   0  04:52:49 ?      0:55  /usr/lib/autofs/automountd
    root   367     1   0  04:52:48 ?      0:00  /usr/lib/nfs/lockd
 daemon   369     1   0  04:52:48 ?      0:00  /usr/lib/nfs/statd
   root   400     1   0  04:52:49 ?      0:00  /usr/sbin/cron
...
    root  1183  1135 0 15:59:16 ?      0:00  dtgreet -display :17
...
```

Środowisko wykonania procesu

- Częścią środowiska wykonania procesu UNIX są **zmienne środowiskowe** (*environment variables*). Mają one postać:

  nazwa=wartość

Często spotykane (predefiniowane) zmienne środowiskowe:

PATH - lista ścieżek dostępu do plików wykonywalnych realizujących polecenia powłoki
HOME - katalog macierzysty użytkownika
PWD - aktualny katalog
PS1,PS2 - pierwszy i drugi tekst zachęty
TERM - nazwa (typ, model) używanego terminala
SHELL - używana powłoka
LOGNAME -nazwa użytkownika
RANDOM - liczba losowa
EDITOR - edytor użytkownika
PPID - nr procesu rodzicielskiego

- Zmienne środowiskowe są **dziedziczone** przez proces potomny uzyskany przez wywołanie fork(). Przy wywoływaniu execle(), execve() zbiór zmiennych środowiskowych jest definiowany na nowo, dla pozostałych funkcji grupy exec – zmienne środowiskowe nie zmieniają się.

Środowisko wykonania procesu – c.d.

- Nadawanie wartości zmiennym środowiskowym
  - Powłoka bash, ksh, zsh itp.

    name=value; **export** *name*
  - Program w języku C

    putenv(”*name*=value”);

  Uwaga: zmiana dotyczy procesu, wykonujące powyższe czynności (i nowo utworzonych procesów potomnych – jeśli dziedziczą środowisko).
- Pobieranie wartości zmiennych środowiskowych
  - Powłoka sh, bash, zsh itp.

    echo $*name* # wyświetlanie wartości zmiennej *name*
  - Program w języku C

```
...
char *p=getenv(”name”);
if(p) printf (”name=%s\n”,p);
else printf(„Nie zdefiniowano zmiennej name\n”);
...
```

Środowisko wykonania procesu – c.d.

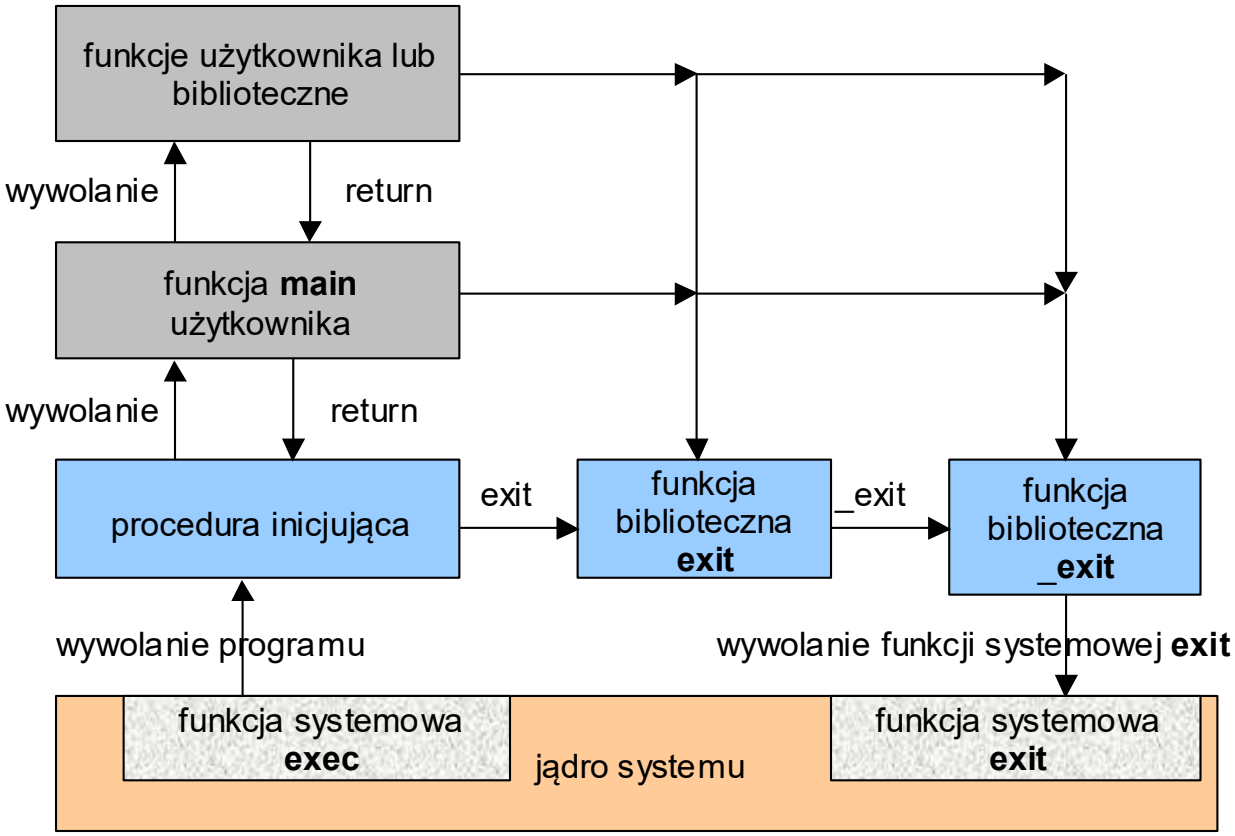
POSIX:

Przy tworzeniu procesu, tworzone są trzy **strumienie** (*streams*):

- **standardowe wejście** (*standard input)*
- **standardowe wyjście** (*standard output*) oraz
- **standardowe wyjście diagnostyczne** (*standard error output*).

Po otwarciu strumień diagnostyczny nie jest w pełni buforowany (*is not fully buffered*); pozostałe strumienie są w pełni buforowane (*are fully buffered*) - jeśli tylko strumień nie jest związany z urządzeniem interakcyjnym.

Cykl życia procesu



Droga: od utworzenia procesu funkcją systemową **`exec`**
do jego zakończenia funkcją systemową **`exit`**

Cykl życia procesu – c.d.

- Funkcje systemowe związane z tworzeniem i kończeniem procesów:
  - **`fork`** tworzy nowy proces (logiczną kopię procesu macierzystego)
  - Funkcja rodziny **`exec`** (tu `execl`) jest używana po rozwidleniu procesu (za pomocą **`fork`**), aby zastąpić przestrzeń adresową wołającego nowym programem
  - Funkcja **`_exit`** kończy działanie procesu
  - Proces może oczekiwać (funkcja **`wait`**) na zakończenie procesu potomka; **`wait`** udostępnia `PID` zakończonego procesu, aby proces rodzic mógł określić który potomek zakończył działanie, a także status (określający przyczynę zakończenia).

```
pid_t pid1, pid2;
pid1 = fork(); /* duplikowanie bieżącego procesu */
if (pid1 < 0) { perror("Fork failed"); /* błąd wykonania fork .... */ exit(1);
} else if ( pid1 == 0 ) { /* proces potomny  */
        execl("/bin/ls", "ls", „-l", NULL); /* Kod wyjścia tego procesu zależy od nowej zawartości procesu */
        perror(”execl”); /* ta linia nigdy się nie wykona, jeśli execl() się wykona */
        exit(2);
} else { /* proces macierzysty */
        if( (pid2=wait (&status)) > 0 ){/* oczekiwanie na proces potomny */
        /* wykorzystanie PID zakończonego procesu (pid2) i informacji o przyczynie zakończenia (status) */
                ...
        } else {/* zakończenie awaryjne */
                perror(”wait”);  /* Błąd wykonania wait */ exit(3);
        }
}
```

Cykl życia procesu – c.d.

- Proces potomny (uzyskany przez wywołanie `fork()`) dziedziczy od procesu macierzystego m.in.:
  - Zawartość przestrzeni adresowej użytkownika (za wyjątkiem wartości zwracanej przez funkcję `fork()`)
  - Deskryptory plików (łączy, gniazd)
  - Zmienne środowiskowe (modyfikowalne przez `execle()`, `execve()`)
  - Odwzorowania plików w pamięci (`mmap()`)
  - Obsługę sygnałów
  - Politykę planowania i priorytet
- Proces potomny nie dziedziczy m.in.:
  - Stanu budzików
  - Sygnałów oczekujących
  - Operacji asynchronicznych
  - Innych wątków niż ten, który wywołał `fork()`

Cykl życia procesu – c.d.

- Proces uzyskany przez wywołanie funkcji systemowej **exec** dziedziczy m.in.:
  - PID, PPID,PGID, RUID, RGID, zawartość liczników alarmów
  - Katalog bieżący i domowy, **umask**, **ulimit**
  - Deskryptory plików (łączy, gniazd), za wyjątkiem tych oznaczonych jako **close-on-exec**
  - Zmienne środowiskowe (o ile nie są modyfikowalne przez wywołanie funkcji bibliotecznych **execle**(), **execve**())
  - Maskę sygnałów i sygnały oczekujące
  - Terminal sterujący
  - Ograniczenia zasobów
- Sygnały obsługiwane domyślnie (`SIG_DFL`) oraz ignorowane (`SIGN_IGN`) nie zmieniają dyspozycji, za wyjątkiem `SIGCHLD` (jeśli był ignorowany – po przeistoczeniu dyspozycja nie jest określona: `SIGN_DFL` albo `SIGN_IGN`). Sygnały obsługiwane przez funkcję obsługi (handler) zmieniają dyspozycję na domyślną.
- Stan jednostki zmiennoprzecinkowej staje się początkowy (jak po inicjalizacji).
- Licznik zużycia czasu procesora nie jest zerowany.

Cykl życia procesu – c.d.

- Są dwa typy kończenia procesu:
  - Normalny – przez powrót z main(), czy wywołanie **exit()** , **_exit()**
  - Awaryjny – przez wywołanie **abort()** , albo wskutek doręczenia pewnych sygnałów
- Przy kończeniu procesu zamykane są deskryptory i strumienie tego procesu.
- Jeśli rodzic kończonego procesu ustawił flagę SA_NOCLDWAIT albo reakcję sygnału SIGCHLD na SIG_IGN, to:
  - Informacja o statusie potomka jest gubiona, a jego czas życia się kończy
  - W przeciwnym razie tworzona jest informacja o statusie, potomek przekształcony jest w : „*proces nieżywy*” , a do rodzica jest wysłany sygnał **`SIGCHLD`** .
- Proces, który zakończył działanie, ale oczekuje na pobranie przez proces macierzysty przyczyny tego zakończenia (**`wait`**) to „*proces nieżywy*” (***zombie***).
- Proces staje się ***sierotą***, gdy jego proces macierzysty kończy się; obowiązki procesu macierzystego przejmuje wówczas systemowy proces o ustalonym PID (**`init/systemd,`**…). Uwaga: W SO UNIX PID=1; POSIX nie narzuca PID=1.

Funkcje do tworzenia procesów

`pid_t` **`fork`**`(void)`

tworzy logiczną kopię bieżącego procesu; zwraca:

–1 - gdy się nie powiedzie (kod błędu w **`errno`**)

0 - w procesie potomnym

>0 - w procesie macierzystym (PID procesu potomnego)

`pid_t vfork(void)`

tworzy kopię bieżącego procesu (dla procesu wielowątkowego – kopię wątku wołającego **`vfork`**()), która współdzieli przestrzeń adresową z procesem macierzystym. Proces macierzysty (wątek wołający **`vfork`**()) jest wstrzymany, aż proces potomny wywoła **`exec`** () bądź **`_exit`**(). Proces potomny nie powinien wołać **`exit`**(), gdyż funkcja ta opróżnia bufory i zamyka (współdzielone z procesem macierzystym) standardowe kanały wejścia/wyjścia. Używanie **`vfork`**() dla dużych procesów powoduje znaczne oszczędności czasu procesora, ale jest potencjalnie niebezpieczne. We współczesnych systemach z pamięcią wirtualną funkcja **`fork`**() jest realizowana efektywnie (z użyciem „copy-on-write”) => używanie funkcji **`vfork`**() nie jest uzasadnione.

Funkcje do tworzenia procesów – c.d.

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

tworzy nowy proces, który może współdzielić część kontekstu procesu wołającego. Nowy proces wykonuje kod ***fn(arg)*** (a nie jak w przypadku **fork**() - kod procesu wołającego), a kiedy kod ten wykonuje return – proces potomny się kończy (może się również zakończyć wywołując **exit**(), bądź wskutek doręczenia sygnału).
child_stack musi wskazywać na odpowiednio długi obszar pamięci, który będzie wykorzystywany jako stos nowego procesu. Bity parametru **flags** określają co jest współdzielone. Symboliczne oznaczenia bitów tego parametru

- **CLONE_PARENT** – współdzielenie PPID procesu macierzystego
- **CLONE_FS** – współdzielenie informacji o korzeniu systemu plików: katalogu bieżącym u-masce.
- **CLONE_FILES** – współdzielenie tablicy deskryptorów plików
- **CLONE_SIGHAND** – współdzielenie tablicy obsługi sygnałów
- **CLONE_VM** – współdzielenie pamięci wirtualnej

Ponadto:

- **CLONE_VFORK** - włączenie wstrzymywania procesu macierzystego aż do zakończenia procesu potomnego

Funkcja zwraca identyfikator wątku (TID) procesu potomnego, lub –1 (errno)

clone() nie jest funkcją zgodną ze standardem POSIX.

Funkcje do nadzoru procesów

`pid_t` **`wait`**`(int *pstatus)`

czeka na zakończony podproces, zwracając jego PID.Funkcja zwraca –1 w przypadku wykrycia błędu (np. jeśli nie ma podprocesu wykonującego się ani zakończonego, bądź wystąpił sygnał). Dla pomyślnego wykonania, gdy status!=NULL => ***`status=*pstatus`*** zawiera informację o przyczynie zakończenia. Następujące makro pozwalają skorzystać z wartości status:

`WIFEXITED``(status)`	1, gdy podproces zakończony przez **`exit`**`()`, 0 w przeciwnym przypadku
`WEXITSTATUS``(status)`	argument wywołania **`exit`**(), gdy podproces został zakończony przez **`exit`**()
`WIFSIGNALED``(status)`	1, gdy podproces został zakończony wskutek doręczenia sygnału, 0 w przeciwnym przypadku
`WTERMSIG``(status)`	numer sygnału, który zakończył proces
`WIFSTOPPED``(status)`	1, gdy podproces zatrzymano, bądź 0
`WSTOPSIG``(status)`	nr sygnału, który zatrzymał proces

Funkcje do nadzoru procesów – c.d.

`pid_t` **`waitpid`**`(pid_t` *`pid`*`, int *`*`pstatus`*`. int` *`options`*`)`

czeka na zakończony podproces:

`pid == -1` => na dowolny podproces

`pid < -1` => na dowolny podproces należący do grupy procesów pgid=-pid

`pid == 0` => na dowolny podproces należący do tej samej grupy procesów co proces wołający

`pid > 0` => na podproces o **`PID==pid`**

Parametr *options* jest sumą logiczną 0 i jednej lub dwóch wartości:

`WNOHANG` – **`waitpid`** powraca natychmiast jeśli nie ma zakończonego potomka

`WUNTRACED` - **`waitpid`** powraca również, gdy któryś z potomków został wstrzymany.

`waitpid` zwraca PID podprocesu, bądź –1 w przypadku błędu

Funkcje do nadzoru procesów – c.d.

`void _exit(int status)`

funkcja biblioteczna powodująca „bezzwłoczne zakończenie” procesu przez wywołanie funkcji systemowej **`exit`**. Uwaga: otwarte pliki są zamykane bez opróżniania buforów; procesy potomne są „adoptowane” przez proces **`init`**, a proces macierzysty otrzymuje sygnał **`SIGCHLD`**.

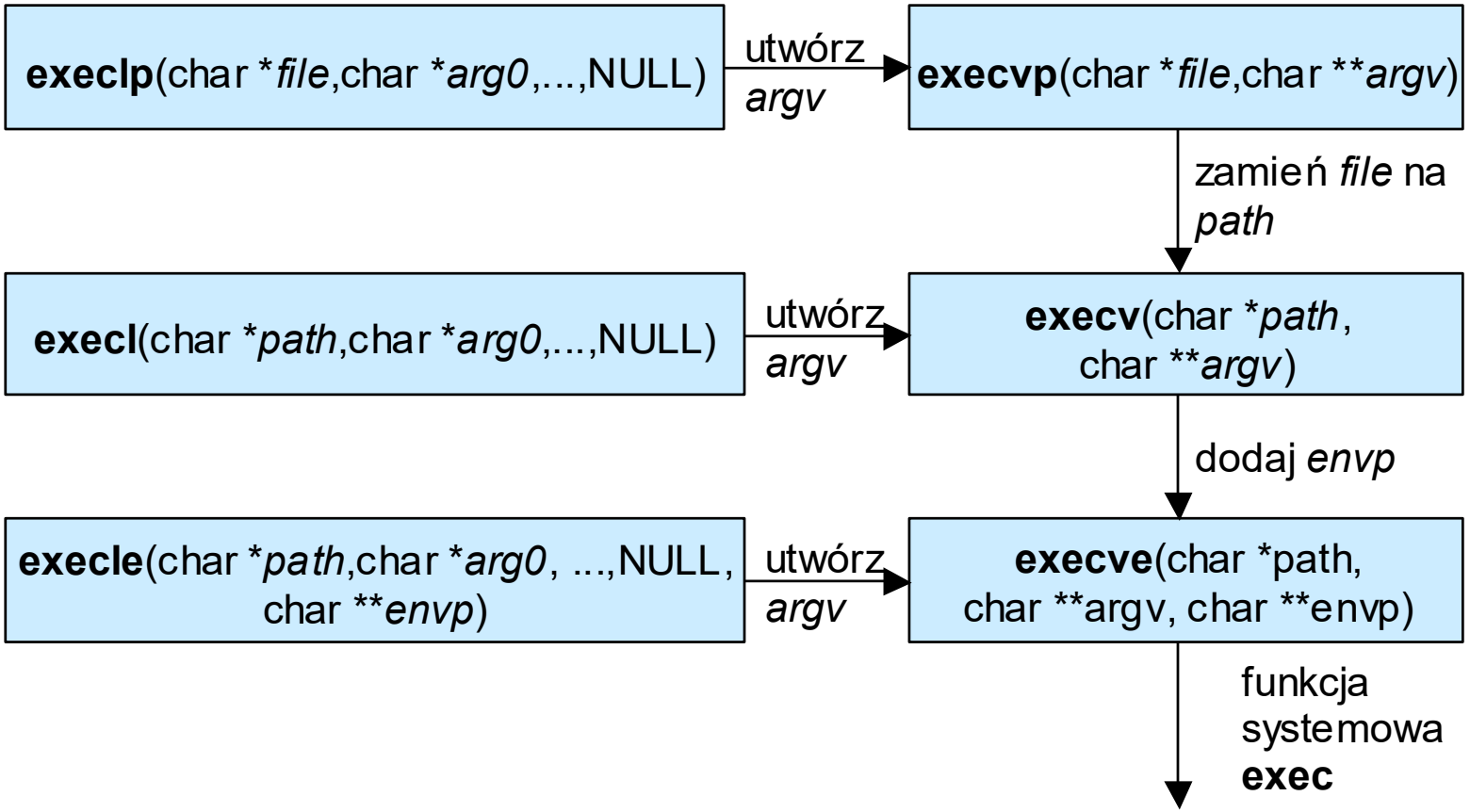
`void exit(int status)`

funkcja biblioteczna wywołująca funkcję **`_exit`** po wywołaniu funkcji zarejestrowanych przez **`atexit()`** (bądź **`on_exit`**()), po wypisaniu danych wszystkich częściowo zapisanych buforów oraz po zamknięciu plików tymczasowych utworzonych za pomocą **tmpfile**(). Jeśli kończony proces jest liderem sesji terminalowej, to każdy pierwszoplanowy proces grupy procesów związanych z tym terminalem dostaje sygnał **`SIGHUP`**; znika też dotychczasowy związek terminala sterującego z sesją terminalową.

Wartość zmiennej *status* może być równa **0, EXIT_SUCCESS, EXIT_FAILURE**, albo inna wartość, przy czym tylko 8 najmniej znaczących bitów tej liczby (t.j. **status & 0377**) będzie dostępne oczekującemu procesowi.

Funkcje do tworzenia/kończenia/nadzoru procesów – c.d.

Funkcje biblioteczne związane z funkcją systemową **exec**



Modyfikacja praw dostępu procesów

- Funkcja **`fork`** nie zmienia identyfikatorów UID i GID procesu (rzeczywistych, obowiązujących, ani zapamiętanych)
- Proces użytkownika o (obowiązującym) **UID==0** może, za pomocą funkcji `setuid()`, zmienić (rzeczywiste, obowiązujące oraz zapamiętane) UID (a za pomocą **`setgid`()** GID) procesu na dowolne wartości zarejestrowane w systemie.
- Wywołanie funkcji **`exec`** zazwyczaj zachowuje (rzeczywiste, obowiązujące i zapamiętane) UID i GID wołającego procesu. Jeśli jednak ustawić bit **`setuid`** w i-węźle pliku wykonywalnego, to obowiązującym identyfikatorem (*effective user identifier*) i zapamiętanym (*saved user identifier)* procesu wykonującego ten plik staje się identyfikator właściciela pliku, podczas gdy rzeczywisty identyfikator użytkownika (*real user identifier*) pozostaje niezmienny. Podobnie bit **setgid** zmienia obowiązujący i zapamiętany identyfikator grupy użytkownika wykonującego program z tym atrybutem (na czas wykonywania programu). W kodzie programu można przełączać identyfikator obowiązujący pomiędzy identyfikatorami: rzeczywistym i zapamiętanym.
- Bity **`setuid/setgid`** nadają więc "zwykłym użytkownikom" prawa dostępu (np. do plików) takie jakie mają właściciele programów, którzy te bity ustawili.
- Bity **`setuid`** i **`setgid`** dla niezbyt starannie napisanych programów mogą zmniejszać bezpieczeństwo systemu.

Grupy procesów

- **Grupa procesów** to zbiór procesów współpracujących ze sobą przy wykonywaniu wspólnych zadań. Proces dziedziczy przynależność do grupy procesów po przodku, ale może też utworzyć nową grupę (stając się jej przywódcą). Identyfikatorem grupy procesów jest PID przywódcy.
- **Sesja** – kolekcja grup procesów, utworzona dla realizacji sterowania zadaniami (*job control*) przy pomocy terminala sterującego. Operacje: suspend/resume, fg/bg, kontrola użycia terminala przez procesy. Każda grupa procesów należy do jakiejś sesji. Przynależność procesu do sesji jest dziedziczona
- Grupy procesów są używane przez powłoki do nadzorowania pracy wielu zadań..
  - Procesy pierwszoplanowe (**foreground proces group**) mają nieograniczony dostęp do terminala sterującego sesji. Pozostałe procesy (drugoplanowe, **background**) – nie.
  - Sekwencje sterujące terminala mogą powodować przesłanie sygnałów do procesów grupy pierwszoplanowej.
- **Zadanie (*job*)** – zbiór procesów jednego potoku powłoki (i ich potomstwo), które są w tej samej grupie procesów. Zadanie dziedziczy terminal sterujący po procesie – rodzica.
- .

Powłoka zgłoszeniowa klasycznego UNIXa

- Proces **`init`** tworzy proces potomny **`getty`** (czy t.p.) który staje się przywódcą **sesji**, otwiera dostęp do terminala, oczekuje na **nazwę rejestracyjną (*login name*)** użytkownika, po czym wywołuje program **login** z parametrem - nazwą użytkownika
- **`login`** pobiera hasło użytkownika, wyznacza skrót i porównuje go z wartością pamiętaną w `/etc/shadow` (albo w innym miejscu systemu). Przy pomyślnym porównaniu ustawia dla procesu identyfikator użytkownika (UID) i jego grupy (GID) np. wg zawartości pliku **`/etc/passwd`** (albo NIS) oraz rozpoczyna sesję terminalową przez wywołanie **powłoki zgłoszeniowej** (**login shell**) użytkownika (określonej w **`/etc/passwd`**). Powłoka ta pozwala użytkownikowi na pracę interakcyjną.