Systemy operacyjne – studia stacjonarne 2022/23

Lab 6. Procesy: atrybuty procesu, sygnały, funkcje systemowe fork(), exec(), wait(), exit(). Projekt nr1.

1. Atrybuty procesu, stany procesu, .

Z każdym procesem związanych jest wiele atrybutów. Najważniejszymi z nich to:

PID identyfikator procesu (process ID),

PPID identyfikator procesu przodka,

UID identyfikator użytkownika, który proces uruchomił (jest on właścicielem procesu),

EUID efektywny identyfikator użytkownika (effective user ID), który określa jakie prawa przysługują danemu procesowi,

RUID rzeczywisty identyfikator użytkownika (real user ID), który rozpoczął proces; RUID różni się od EUID, jeżeli uruchomiony program miał ustawione rozszerzone prawa dostępu – SUID (s zamiast x na pozycji użytkownika),

RGID rzeczywisty identyfikator grupy użytkownika (real group ID),

EGID efektywny identyfikator grupy użytkownika (effective group); Effective Group ID – różni się od RGID, jeśli uruchomiono program z ustawionym prawem SGID (s zamiast x na pozycji grupy),

TIME czas trwania,

TTY terminal,

COM, **CMD** faktyczne polecenie, które uruchomiło proces,

NI Liczba **nice** mająca wpływ na priorytet procesu, określa poziom uprzejmości procesu,

SIZE wielkość pamięci wirtualnej procesu,

RSS wielkość użytej pamięci rzeczywistej,

STAT aktualny stan procesu; może on przybrać wartość:

- **R** – run (działający),
- **S** – sleep (uśpiony),
- **D** – oczekujący na operację dyskową,
- **T** – stopped lub traced (zatrzymany lub śledzony),
- **Z** – zombie (proces, który zakończył swoje działanie, zwolnił wszystkie używane zasoby, ale nie otrzymał potwierdzenia przyjęcia sygnału zakończenia od procesu rodzica).

Status dodatkowo może być oznaczony symbolami:

- **W**– nie ma strony zaalokowanej w pamięci operacyjnej,
- **L** – ma stronę w pamięci,
- **<** – wysoki priorytet,
- **N** – obniżony priorytet,

PRI aktualny priorytet procesu (obliczany dynamicznie),

STIME czas rozpoczęcia procesu.

2. Informacje o procesach - polecenia ps, pstree, top.

Polecenie **ps** wyświetla informacje (PID, nazwę) o aktualnie wykonywanych procesach. Np. polecenie:

ps -aux wyświetla informacje o działających procesach wszystkich użytkowników,

ps –U nazwa_użytkownika wyświetla informacje dotyczące procesów konkretnego użytkownika.

ps –eo pid,user,ruid,rgid,cmd --sort pid

Jeśli stosuje się opcję **-o** dla zdefiniowania własnego formatu wyświetlanych informacji, należy po niej umieścić oddzielone przecinkami atrybuty. Pełna ich lista dostępna jest w manualu i zawiera m.in. **cmd, pid, user, ruid, ruser, euid, euser, rgid, rgroup, egid, egroup**. Opcja **--sort** umożliwia posortowanie procesów wg danego klucza.

Polecenie **pstree** umożliwia wyświetlenie informacji o procesach w postaci drzewa. Procesy potomne zostają wyświetlone z prawej strony procesów macierzystych.

Polecenie **top** wyświetla informacje o aktywnych procesach z uwzględnieniem najbardziej obciążających. Możliwa jest praca interaktywna. Poprzez linię komend możliwa jest ingerencja w stan procesu:

r zmiana wartości nice dla wskazanego zadania (użytkownik nieuprzywilejowany może ją jedynie podnieść, a tym samym zmniejszyć priorytet procesu),

h lub **?** wyświetlenie pomocy,
k wysłanie sygnału do procesu (trzeba podać PID procesu),
q zakończenie.

Ćwiczenie 1.
Używając polecenia **top** zmień priorytet procesowi top ustawiając atrybut (liczbę) NI na wartość 15.

3. Sygnały – polecenie kill

Sygnały są metodą zawiadamiania procesu o zajściu jakiegoś zdarzenia. Pojawienie się sygnałów powoduje przerwanie pracy procesu i wymusza natychmiastową ich obsługę. W pewnym sensie przypominają one przerwania. Różnica polega na sposobie generowania. Przerwania tworzone są przez sprzęt, natomiast sygnały mogą być wysłane:

- z procesu do procesu
- z systemu operacyjnego do procesu

Z tego względu nazywamy je czasami przerwaniami programowalnymi. Sygnały są asynchroniczne.
Każdy sygnał posiada swój unikalny numer i pojawia się w określonej sytuacji. W Linuksie sygnały są zdefiniowane w pliku /usr/include/bits/signum.h.
Sposoby generowania sygnałów:

3.1. Skróty klawiszowe (najczęściej stosowane)- pewne kombinacje klawiszy powodują generowanie sygnałów.
CTRL+C - wciśnięcie tych klawiszy powoduje wysłanie przez system operacyjny sygnału SIGINT do bieżącego procesu. Domyślnie sygnał ten powoduje natychmiastowe zakończenie procesu.
CTRL+\ - powoduje zakończenie bieżącego procesu i generowanie obrazu pamięci tego procesu; generowany jest sygnał SIGQUIT
CTRL+Z - powoduje zawieszenie procesu poprzez wysłanie sygnału SIGSTOP

3.2. Funkcja systemowa kill() zezwala, aby proces wysłał sygnał do drugiego procesu albo do samego siebie (funkcja raise()). Składnia wygląda następująco:

```
#include <sys/types.h>
#include <signal.h>
int kill(int idproc, int sig);
int raise(int sig);
```

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=raise

3.3. Polecenie kill:

```
kill -<signal> <PID>
```

kill –l wyświetla listę sygnałów, które można wysłać. Najważniejsze z nich to:

2	SIGINIT	Ctrl+c	– przerwanie wykonywania procesu,
3	SIGQUIT	Ctrl+\	– zakończenie procesu oraz zrzut pamięci,
9	SIGKILL		– zabicie procesu,
15	SIGTERM		– miękkie zakończenie procesu (domyślny),
19	SIGSTOP	Ctrl+z	– zatrzymanie procesu.

Ćwiczenie 2.
Uruchom dwa procesy bash (w dwóch oknach). Używając polecenia kill wyślij do drugiego procesu bash sygnał nr 9 (SIGKILL).

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=kill

3.4. Pewne sytuacje wykrywane sprzętowo są przyczyna generowania sygnałów. Przykładowo odwołanie do niewłaściwego adresu pamięci generuje sygnał SIGSEGV. Błąd przy wykonywaniu operacji na liczbach zmiennoprzecinkowych jest wskazywane sygnałem SIGFPE.

3.5. Pewne sytuacje wykrywane przez oprogramowanie systemowe, o których jest powiadamiane jądro, powoduje generowanie sygnałów. Przykładem jest sygnał SIGURG, który pojawia się gdy na połączeniu sieciowym pojawią się dane wysokopriorytetowe, lub sygnał SIGALRM generowany przy przekroczeniu terminu licznika zegarowego ustawionego w procesie.

3.6. „Przechwytywanie" sygnałów.

Do zmieniania akcji, którą obiera proces po odebraniu określonego sygnału możemy użyć funkcji: signal() lub sigaction(). Możemy za ich pomocą sprawdzać lub modyfikować akcję związaną z konkretnym sygnałem.

Funkcja signal().

```
#include <signal.h>
       typedef void (*sighandler_t)(int);
       sighandler_t signal(int signum, sighandler_t handler);
```

Funkcja systemowa signal() instaluje nową obsługę sygnału dla sygnału o numerze signum. Obsługa sygnału ustawiana jest na sighandler, który może być funkcją podaną przez użytkownika lub SIG_IGN albo SIG_DFL.

Przykład użycia funkcji signal():

```
void handler(int sig)
{
        printf("Sygnal SIGINT!\n");
        return;
}
...
signal(SIGINT, handler);
...
```

Funkcja sigaction().

```
#include <signal.h>
struct sigaction
{
     void (*sa_handler)(int);
     sigset_t sa_mask;
     int sa_flags;
};
int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact);
```

Pierwszym jej parametrem jest numer sygnału. Następne dwa to wskaźniki do struktur sigaction, w których najważniejszym polem jest sa_handler. Może ono przyjmować jedną z trzech wartości:

- SIG_IGN - powoduje ignorowanie sygnału
- SIG_DFL - powoduje wywołanie akcji domyślnej
- Wskaźnik do funkcji obsługującej sygnał.

Argument signum jest numerem sygnału, dla którego chcemy sprawdzić lub zmodyfikować dyspozycje.

Wskaźnik act zawiera nowe dyspozycje: sa_handler – wskaźnik do funkcji obsługi sygnału
sa_mask - zbiór dodatkowych sygnałów do zablokowania
sa_flags - zbiór dodatkowych opcji.

Przykład użycia funkcji sigaction()

```
void sighandler(int signum, siginfo_t *info, void *ptr)
{
        printf("Sygnal SIGINT!\n");
        return;
}
        struct sigaction sa;
        sa.sa_sigaction = sighandler;
...
        sigaction(SIGINT, &sa, NULL);
...
```

Przykład.
Skopiuj do swojego katalogu domowego, pliki signal.c, sigaction.c znajdujące się w katalogu:
/home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/SYGNALY
Przeanalizuj sposób działania funkcji signal(), sigaction().

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=signal
http://www.linux.pl/man/index.php?command=sigaction

4. Uruchamianie procesów w trybie pierwszoplanowym i w tle.

Uruchamianie poleceń w tle odbywa się przez umieszczenie znaku **&** na końcu wiersza poleceń. Po uruchomieniu polecenia w tle wyświetlany jest numer zadania użytkownika oraz PID procesu, np. [1] 543. W tło można przenieść więcej niż jedno polecenie. Listę zadań działających w tle wypisuje polecenie **jobs**. Każda pozycja na liście składa się z numeru zadania w nawiasach kwadratowych, informacji, czy zadanie jest zatrzymane, czy działa oraz jego nazwy. Znak „+" oznacza, że zadanie jest właśnie wykonywane, a znak „-", że czeka w kolejce do wykonania. System nie informuje o zakończeniu zadania. Zadanie działające w tle można przenieść na plan pierwszy poleceniem **fg**. Jeśli w tle działa więcej niż jedno zadanie, należy podać jego numer po symbolu % jako argument polecenia, np. **fg %2**.

Zadanie działające na pierwszym planie można umieścić w tle. Aby było to możliwe, aktualnie wykonywane zadanie trzeba przerwać (zatrzymać). Kombinacja klawiszy [**CTRL+Z**] powoduje wstrzymanie procesu, dopóki nie zostanie on wznowiony. Przerwane zadanie poleceniem **bg** można umieścić w tle, np. **bg %2**.

% – oznacza bieżące zadanie, **%n** – oznacza zadanie o numerze n.

Zadanie działające w tle można zatrzymać wymuszając jego zakończenie, dzięki poleceniu **kill**. Polecenie **kill** przyjmuje jako argument numer zadania użytkownika lub pid procesu. Numer zadania użytkownika musi być poprzedzony znakiem **%** (np. **kill %2**).

Ćwiczenie 3.
Uruchom dwa procesy sleep: jeden w tle (sleep 300), drugi w trybie pierwszoplanowym (sleep 200). Przenieś proces sleep 200 w tło. Wypisz listę zadań wykonywanych w tle. Przywołaj proces sleep 300 na pierwszy plan, a następnie ponownie przenieś w tło. Zakończ procesy używając polecenia kill (w wersji z PID i nr zadania).

5. System plikowy proc.

Katalog **/proc** to punkt montowania wirtualnego system plików - który znajduje się w pamięci operacyjnej. Daje on dostęp do danych na temat uruchomionych procesów i jądra systemu. Znajdują się tam pliki: kcore, będący obrazem pamięci systemu, cpuinfo, zawierający dane na temat procesora, meminfo, przechowujący informacje o pamięci operacyjnej oraz katalogi odpowiadające wszystkim uruchomionym procesom, których nazwy są identyfikatorami (PID) procesów. Tekstowe pliki znajdujące się w tych katalogach zapewniają dostęp do danych o procesach i można z nich odczytać atrybuty procesów.

6. Identyfikatory związane z procesami.

Podstawowe identyfikatory związane z procesami oraz funkcje systemowe służące do ich uzyskiwania:

Nazwa	Funkcja systemowa	Opis
PID	pid_t getpid(void)	identyfikator procesu
PPID	pid_t getppid(void)	identyfikator procesu macierzystego
UID	uid_t getuid(void)	identyfikator użytkownika (rzeczywisty)

GID	gid_t	getgid(void)	identyfikator grupy użytkownika (rzeczywisty)
PGID	pid_t pid_t	getpgid(void) getpgrp(void)	identyfikator grupy procesów getpgid(0)

Identyfikatory te przyjmują wartości liczb całkowitych nieujemnych. Do wywołania powyższych funkcji niezbędne są następujące pliki nagłówkowe: <sys/types.h>, <unistd.h>.

7. Funkcja biblioteczna perror(), zmienna errno.

- Wywołanie funkcji systemowej lub bibliotecznej, które zakończy się błędem zwraca wartość -1 (czasami NULL) i przypisuje zmiennej zewnętrznej errno wartość wskazującą na rodzaj błędu. Plik nagłówkowy <sys/errno.h> definiuje zmienną errno, odpowiadające kody (stałe) błędów oraz odpowiadające im komunikaty błędów.
- Funkcja biblioteczna **perror()** wypisuje komunikat błędu poprzedzony napisem *s i znakiem ':'

8. Tworzenie procesów potomnych – funkcja fork(), exec(), wait().

Typowe wywołanie fork()

```
switch (fork())
{   case -1;
        perror(„fork error”);
        exit(1);
    case 0;
         /* akcja dla procesu
potomnego */
    default;
        /* akcja dla procesu
macierzystego */
}
```

Typowe wywołanie fork i exec

```
switch (fork())
{   case -1;
        perror(„fork error”);
        exit(1);
    case 0; /* proces potomny */
         execl(„./nowy_program.x”,
„nowy_program.x”,NULL);
         exit(2);
    default;  /* proces macierzysty
*/
}
```

pliki nagłówkowe	<sys/types.h>, <unistd.h>		
prototyp	pid_t fork(void)		
zwracana wartość	sukces	porażka	zmiana errno
	0 w procesie potomnym PID procesu potomnego w procesie macierzystym	-1	tak
prototyp	int execl(cont char *path, const char *arg0,..., const char*argn, char *null)		
zwracana wartość	sukces	porażka	zmiana errno
	nic nie zwraca	-1	tak

Argumenty funkcji exec

path	- ścieżkowa nazwa pliku (wykonawczego) zawierającego program
arg0	- argument zerowy (nazwa pliku zawierającego program
arg1,...,argn	- argumenty wywołania programu
null	- wskaźnik NULL
argv[ ]	- adres tablicy wskaźników na ciągi znaków będące argumentami przekazywanymi do wykonywanego programu (ostatnim elementem powinien być NULL)

Rodzina funkcji exec():

```
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[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

Znaczenie poszczególnych liter, używanych w nazwach funkcji exec():

l - argumenty wywołania programu podane w postaci listy napisów zakończonej zerem (NULL);

p - przeszukiwanie zmiennej środowiskowej PATH w celu znalezienia programu;

v - argumenty wywołania programu w postaci tablicy napisów (tak jak argument argv funkcji main)

e - oznacza, że środowisko jest przekazywane ręcznie jako ostatni argument wywołania funkcji

pliki nagłówkowe	<sys/types.h>, <sys/wait.h>		
prototyp	pid_t wait(int *status)		
zwracana wartość	sukces	porażka	zmiana errno
	PID procesu potomnego	-1	tak

status - wskaźnik do zmiennej zawierającej kod zakończonego procesu potomnego.

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=fork
http://www.linux.pl/man/index.php?command=execl
http://www.linux.pl/man/index.php?command=waitpid

Ćwiczenie 4.

Skopiuj do swojego katalogu domowego, pliki fork_exec.c, f-test.c oraz e-test.c znajdujące się w katalogu: /home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/PROCESY
Przeanalizuj sposób działania funkcji fork() i exec() w oparciu o programy zawarte w pliku fork_exec.c .
Zmodyfikuj program e-test.c tak, aby w procesie potomnym uruchomić program sl z parametrem -a.

PROJEKT NR1

Napisać program wypisujący identyfikatory PID, PPID, UID, GID dla danego procesu (program1_1).

Wywołać funkcję fork() 3 razy np. w pętli i wypisać powyższe identyfikatory dla wszystkich procesów potomnych. Na podstawie wyników narysować „drzewo genealogiczne" tworzonych procesów. Ile powstaje procesów i dlaczego (program1_2).

Zmodyfikować poprzedni program tak, aby komunikaty procesów potomnych były wypisywane przez program uruchamiany przez funkcję exec(). W procesie macierzystym zaimplementuj funkcję wait() (proces macierzysty czeka na zakończenie wszystkich procesów potomnych, wypisuje komunikat, który proces potomny się zakończył i z jakim statusem). Ile teraz powstaje procesów i dlaczego? Dla wszystkich funkcji systemowych (fork(), execl(), wait()) zaimplementuj obsługę błędów używając funkcji bibliotecznej perror() (program1_3).