API usług jądra systemu operacyjnego UNIX*

Dariusz Wawrzyniak
Dariusz.Wawrzyniak@cs.put.poznan.pl

11 stycznia 2017

Wszystkie opisane niżej funkcje z wyjątkiem funkcji z biblioteki *pthread* (patrz punkt 2) w przypadku błędu zwracają wartość -1, w związku z czym fakt ten nie jest wyszczególniony przy opisie poszczególnych funkcji.

1 Procesy

W odniesieniu do procesów wymagana jest znajomość następujących terminów i zagadnień: *proces*, hierarchia procesów i sposoby ich identyfikacji (*identyfikator procesu* — pid, *identyfikator procesu macierzystego* — ppid, *efektywny identyfikator użytkownika/grupy* i *rzeczywisty identyfikator użytkownika/grupy*), proces systemowy *init*, *program* wykonywany przez proces, *proces macierzysty*, *proces potomny*, *proces-sierota*, adoptowanie sierot, *proces-zombi*.

`fork()` — utworzenie procesu potomnego. W procesie macierzystym funkcja zwraca identyfikator (pid) procesu potomnego (wartość większą od 0, w praktyce większą od 1), a w procesie potomnym wartość 0.

`getpid()` — zwrócenie własnego identyfikatora procesu. Funkcja zwraca własny identyfikator (pid) procesu, który ją wywołał.

`getppid()` — zwrócenie identyfikatora przodka. Funkcja zwraca identyfikator przodka procesu wywołującego (identyfikator procesu macierzystego).

`exit(int status)` — zakończenie procesu. Funkcja kończy proces i powoduje przekazanie w odpowiednie miejsce tablicy procesów wartości *status*, która może zostać odebrana i zinterpretowana przez proces macierzysty.

`wait(int *status)` — oczekiwanie na zakończenie potomka. Funkcja zwraca identyfikator (pid) procesu, który się zakończył. Pod adresem wskazywanym przez *status* umieszczany jest status zakończenia, który zawiera albo numer sygnału (najmniej znaczące 7 bitów), albo status właściwy (bardziej znaczący bajt). Najbardziej znaczący bit młodszego bajta wskazuje, czy nastąpił zrzut `core`'a.

`execl(const char* path, const char* arg, ...)` — zmiana programu wykonywanego przez proces. Proces rozpoczyna wykonywanie nowego programu, którego kod znajduje się w pliku wskazywanym przez *path*. Jeśli *path* nie jest ścieżką absolutną, to znaczy, że ścieżka rozpoczyna się od katalogu bieżącego. Pomimo zmiany wykonywanego programu pewne atrybuty procesu (pid, ppid, tablica otwartych plików) nie ulegają zmianie.

*W przypadku wykrycia jakichkolwiek błędów proszę o mail na podany adres.

`execlp(const char* file, const char* arg, ...)` — funkcja działa podobnie jak `execl`, jednak plik z programem poszukiwany jest w katalogach wyszczególnionych w zmiennej środowiskowej `PATH`.

Analogiczne jest działanie funkcji systemowych `execv` i `execvp`. Różnica jest w przekazywaniu parametrów: w przypadku funkcji `execv`/`execvp` parametry przekazywane są przez tablicę łańcuchów znaków.

2 Wątki

Wymagana jest znajomość zagadnień dostępu wątków do współdzielonej przestrzeni adresowej, szczególnie zasad współdzielenia segmentu danych i segmentu stosu.

Funkcje opisane w tym punkcie zwracają wartość 0 w przypadku poprawnego zakończenia lub kod błędu, który jest wartością różną od 0.

2.1 Tworzenie wątków

`pthread_create(pthread_t *thread, pthread_attr_t *attr, void* (*start_routine) (void*), void *arg)` — utworzenie wątku. Wątek wykonuje funkcję wskazywaną przez parametr *start_routine*. Parametry funkcji muszą być przekazane przez wskaźnik na obszar pamięci (strukturę), który zawiera odpowiednie wartości. Wskaźnik ten jest przekazywany przez parametr *arg* i jest dalej przekazywany jako parametr aktualny do funkcji wykonywanej przez wątek. Parametr *attr* wskazuje na atrybuty wątku, a przez wskaźnik *thread* zwracany jest identyfikator wątku.

`pthread_exit(void *retval)` — zakończenie wątku. Funkcja powoduje zakończenie wątku i przekazanie *retval*, jako wskaźnika na wynik. Wskaźnik ten może zostać przejęty przez inny wątek, który będzie wykonywał funkcję `pthread_join`.

`pthread_join(pthread_t th, void **thread_return)` — oczekiwanie na zakończenie wątku. Funkcja umożliwia zablokowanie wątku w oczekiwaniu na zakończenie innego wątku, identyfikowanego przez parametr *th*. Jeśli oczekiwany wątek zakończył się wcześniej, funkcja zakończy się natychmiast. Funkcja przekazuje przez parametr *thread_return* wskaźnik na wynik wątku (wykonywanej przez niego funkcji), przekazany jako parametr funkcji `pthread_exit` wywołanej w zakończonym wątku.

`pthread_cancel(pthread_t thread)` — zakończenie wykonywania innego wątku. Funkcja umożliwia wątkowi

usunięcie z systemu innego wątku, identyfikowanego przez parametr *thread*.

2.2 Synchronizacja wątków

2.2.1 Wzajemne wykluczanie

Do zapewnienia wzajemnego wykluczania używana jest zmienna (o przykładowej nazwie `mutex`), zadeklarowana następująco:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

`pthread_mutex_lock(pthread_mutex_t *mutex)` — zajęcie zamka. Funkcja powoduje zajęcie zamka wskazywanego przez parametr *mutex*. (zajęcie sekcji krytycznej) poprzedzone ewentualnym zablokowaniem wątku do czasu zwolnienia zamka, jeśli został on wcześniej zajęty przez inny wątek.

`pthread_mutex_unlock(pthread_mutex_t *mutex)` — zwolnienie zamka. Funkcja powoduje zwolnienie zamka wskazywanego przez parametr *mutex* (zwolnienie sekcji krytycznej), umożliwiając jego zajęcie innemu wątkowi.

`pthread_mutex_trylock(pthread_mutex_t *mutex)` — próba zajęcia zamka. Funkcja powoduje zajęcie zamka wskazywanego przez parametr *mutex*, jeśli nie jest zajęty przez inny wątek. W przeciwnym przypadku zwraca błąd, nie blokując tym samym procesu.

2.2.2 Zmienne warunkowe

Synchronizacja za pomocą zmiennych warunkowych polega na usypianiu i budzeniu wątku w sekcji krytycznej. W tym celu używana jest zmienna warunkowa (o przykładowej nazwie `cond`), zadeklarowana następująco:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` — oczekiwanie na sygnał. Funkcja powoduje uśpienie wątku na zmiennej warunkowej, wskazywanej przez parametr *cond*. Na czas uśpienia wątek zwalnia zamek, wskazywany przez parametr *mutex*, udostępniając tym samym sekcję krytyczną innym wątkom. Po obudzeniu i wyjściu z funkcji (na skutek odebraniu sygnału wysłanego przez `pthread_cond_signal`) zamek zajmowany jest ponownie.

`pthread_cond_signal(pthread_cond_t *cond)` — wysłanie sygnału (obudzenie) do jednego z oczekujących wątków. Funkcja powoduje wysłanie sygnału do jednego z wątków śpiących na zmiennej warunkowej wskazywanej przez *cond* w celu obudzenia. Jeśli na wskazanej zmiennej warunkowej nie śpi żaden wątek, sygnał ginie bez żadnego efektu.

`pthread_cond_broadcast(pthread_cond_t *cond)` — wysłanie sygnału (obudzenie) do wszystkich oczekujących wątków. Funkcja działa podobnie jak `pthread_cond_signal` z taką różnicą, że sygnał budzący wysyłany jest do wszystkich wątków śpiących na zmiennej warunkowej wskazywanej przez *cond*.

3 Pliki

Wymagana jest znajomość zasad tworzenia i dostępu do plików w hierarchicznej strukturze katalogów.

3.1 Tworzenie i otwieranie plików

`creat(const char *pathname, mode_t mode)` — utworzenie nowego pliku lub usunięcie jego zawartości, gdy już istnieje oraz otwarcie go do zapisu. Funkcja tworzy plik, którego lokalizację wskazuje parametr *pathname*. Prawa dostępu do utworzonego pliku ustawiane są zgodnie z parametrem *mode*. Jeśli plik o takiej nazwie już istnieje, a proces wywołujący funkcję `creat` ma prawo do zapisu tego pliku, to jego zawartość jest usuwana. Plik wskazywany przez *pathname* otwierany jest w trybie do zapisu, a funkcja zwraca jego deskryptor.

`open(const char *pathname, int flags)` — otwarcie pliku. Funkcja otwiera plik o nazwie *pathname* w trybie określonym przez parametr *flags* (np. `O_WRONLY`, `O_RDONLY`, `O_RDWR`). Funkcja zwraca deskryptor otwartego pliku.

`close(int fd)` — zamknięcie deskryptora pliku. Funkcja zamyka deskryptor pliku przekazany przez parametr *fd*. Jeśli jest to ostatni (jedyny) deskryptor pliku, następuje zamknięcie pliku.

`dup(int oldfd)` — powielenie deskryptora. Funkcja powoduje utworzenie dodatkowego deskryptora otwartego wcześniej pliku, identyfikowanego przez parametr (deskryptor) *oldfd*. Nowy deskryptor tworzony jest na pierwszej wolnej pozycji (otrzymuje najmniejszą możliwą wartość). Funkcja zwraca wartość nowo utworzonego deskryptora.

`dup2(int oldfd, int newfd)` — powielenie deskryptora we wskazanym miejscu. Podobnie jak w przypadku funkcji `dup`, tworzony jest nowy deskryptor otwartego pliku, identyfikowanego przez *oldfd*, w miejscu wskazanym przez *newfd*. *newfd* staje się nowym, dodatkowym deskryptorem, a jeśli przed wywołaniem `dup2` identyfikował on inny otwarty plik, następuje zamknięcie tego deskryptora przed powieleniem *oldfd*. Funkcja zwraca wartość nowego deskryptora.

`unlink(const char *pathname)` — usunięcie dowiązania do pliku. Funkcja usuwa wskazaną przez parametr *pathname* nazwę pliku, a jeśli było to jedyne dowiązanie tego pliku, następuje usunięcie pliku z systemu.

3.2 Operacje na plikach

`read(int fd, void *buf, size_t count)` — odczyt z pliku. Funkcja powoduje odczyt *count* bajtów z otwartego pliku, identyfikowanego przez deskryptor *fd*, począwszy od bieżącej pozycji i umieszczenie ich pod adresem *buf* w przestrzeni adresowej procesu. Funkcja zwraca liczbę bajtów, na której udało się wykonać operację.

`write(int fd, const void *buf, size_t count)` — zapis do pliku. Funkcja powoduje zapis *count* bajtów, począwszy od bieżącej pozycji, w otwartym pliku

identyfikowanym przez deskryptor *fd*. Zapisywane wartości pobierane są spod adresu *buf* w przestrzeni adresowej procesu. Funkcja zwraca liczbę bajtów, na której udało się wykonać operację.

`lseek(int fd, off_t offset, int whence)` — przesunięcie wskaźnika bieżącej pozycji. Funkcja powoduje zmianę wskaźnika bieżącej pozycji w otwartym pliku. Nowa pozycja jest bajtem o numerze *offset* liczonym odpowiednio względem

- początku pliku, gdy *whence* = `SEEK_SET`,
- końca pliku, gdy *whence* = `SEEK_END`,
- bieżącej pozycji, gdy *whence* = `SEEK_CUR`.

Wartość parametru *offset* < 0 oznacza przesunięcie w kierunku początku pliku (niższych indeksów), a wartość *offset* > 0 oznacza przesunięcie w kierunku końca pliku (wyższych indeksów). Funkcja zwraca aktualną wartość wskaźnika bieżącej pozycji (po przesunięciu), liczoną względem początku pliku.

4 Łącza (potoki i kolejki FIFO)

Wymagana jest znajomość zagadnień komunikacji strumieniowej, w szczególności różnicy pomiędzy komunikacją strumieniową a komunikacją „pakietową".

4.1 Tworzenie i otwieranie potoków (łączy nienazwanych)

`pipe(int filedes[2])` — utworzenie potoku i otwarcie go do zapisu i do odczytu. Funkcja tworzy i zarazem otwiera potok, i przekazuje przez tablicę *filedes* dwa deskryptory. *filedes*[0] zawiera deskryptor potoku do odczytu, a *filedes*[1] zawiera deskryptor potoku do zapisu.

4.2 Tworzenie i otwieranie klejek FIFO (łączy nazwanych)

`mkfifo(const char *pathname, mode_t mode)` — utworzenie kolejki FIFO. Funkcja tworzy (ale nie otwiera) plik typu *kolejka FIFO* w katalogu i pod nazwą zawartą w parametrze *pathname*, z prawami dostępu przekazanymi w parametrze *mode*.

`open(const char *pathname, int flags)` — otwarcie kolejki FIFO. Funkcja otwiera kolejkę FIFO o nazwie wskazanej przez *pathname*, podobnie jak otwierany jest plik. Kolejka może być otwierana tylko w dwóch trybach: *tylko do odczytu* (*flags* = `O_RDONLY`) lub *tylko do zapisu* (*flags* = `O_WRONLY`). Ponadto kolejka musi zostać otwarta w trybie komplementarnym, tzn. muszą być dwa procesy, z których jeden otwiera kolejkę w trybie `O_RDONLY`, a drugi w trybie `O_WRONLY`. W przeciwnym razie jeden z procesów jest blokowany. Funkcja zwraca deskryptor kolejki.

4.3 Operacje na łączach

`read(int fd, void *buf, size_t count)` — odczyt z łącza. Dane z łącza są odczytywane tak, jak z pliku. Jeśli łącze jest puste, ale jest otwarty jakiś deskryptor do zapisu (potencjalnie w potoku mogą pojawić się jakieś dane), to proces jest blokowany. Jeśli danych jest mniej niż wartość parametru *count*, odczytane zostaną tylko dostępne dane. Jeśli wszystkie deskryptory do zapisu są zamknięte i łącze jest puste, zostanie zwrócona wartość 0. Dane będą odczytywane w kolejności, w której zostały zapisane, a po odczytaniu zostaną usunięte z łącza.

`write(int fd, const void *buf, size_t count)` — zapis do łącza. Dane zapisywane są tak, jak w przypadku zapisu w pliku z wyjątkiem sytuacji, gdy brak jest wystarczającej ilości wolnego miejsca. Wówczas proces jest blokowany. Funkcja zapisze w potoku *count* bajtów w całości i będą one stanowiły ciągły strumień danych, tzn. nie będą się przeplatać z danymi pochodzącymi z innych zapisów (innych wywołań funkcji `write`).

`close(int fd)` — zamknięcie deskryptora łącza. Funkcja działa analogicznie, jak w przypadku zamykania deskryptora zwykłego pliku.

5 Mechanizmy IPC

5.1 Pamięć współdzielona

`shmget(key_t key, int size, int shmflg)` — utworzenie lub pobranie identyfikatora segmentu pamięci współdzielonej. Funkcja tworzy segment współdzielonej pamięci o rozmiarze *size*, identyfikowany przez klucz *key* lub znajduje segment o takim kluczu, jeśli segment już istnieje. Funkcja zwraca identyfikator, który służy do odwoływanie się do segmentu w pozostałych funkcjach operujących na pamięci współdzielonej. Parametr *shmflg* umożliwia przekazanie praw dostępu do kolejki oraz pewnych dodatkowych flag definiujących sposób jej tworzenia (np. `IPC_CREAT`), połączonych z prawami dostępu operatorem sumy bitowej (np. `IPC_CREAT|0660`).

`shmat(int shmid, const void *shmaddr, int shmflg)` — włączenie segmentu pamięci współdzielonej w przestrzeń adresową procesu. Funkcja przydziela segmentowi współdzielonej pamięci, identyfikowanemu przez *shmid*, adres w przestrzeni adresowej procesu i zwraca ten adres jako wynik.

`shmdt(const void *shmaddr)` — wyłączenie segmentu pamięci współdzielonej z przestrzeni adresowej procesu. Funkcja powoduje odłączenie segmentu pamięci współdzielonej, umieszczonego pod adresem *shmaddr*.

`shmctl(int shmid, int cmd, struct shmid_ds *buf)` — operacje kontrolne na segmencie pamięci współdzielonej. Funkcja umożliwia wykonywanie operacji kontrolnych na segmencie pamięci współdzielonej, np. usunięcie tego segmentu.

5.2 Semafory

`semget(key_t key, int nsems, int semflg)` — utworzenie lub pobranie identyfikatora tablicy semaforów.

Funkcja tworzy tablicę składającą się z *nsems* semaforów, jeśli tablica o kluczu *key* jeszcze nie istnieje i zwraca na podstawie klucza identyfikator tej tablicy. Parametr *semflg* umożliwia przekazanie praw dostępu do tablicy semaforów oraz pewnych dodatkowych flag definiujących sposób jej tworzenia (np. `IPC_CREAT`), połączonych z prawami dostępu operatorem sumy bitowej (np. `IPC_CREAT|0660`).

`semop(int semid, struct sembuf *sops, unsigned nsops)` — wykonanie operacji semaforowej. Operacja semaforowa może być wykonywana jednocześnie na kilku semaforach w tej samej tablicy identyfikowanej przez *semid*. *sops* jest wskaźnikiem na tablicę operacji semaforowych, a *nsops* jest liczbą elementów w tej tablicy. Każdy element tablicy opisuje jedną operację semaforową i ma następującą strukturę:

```
struct sembuf {
   short sem_num;  /* numer semafora */
   short sem_op;   /* operacja semaforowa */
   short sem_flg;  /* flagi operacji */
};
```

Pole `sem_op` zawiera wartość, która zostanie dodana do zmiennej semaforowej pod warunkiem, że zmienna semaforowa nie osiągnie w wyniku tej operacji wartości mniejszej od 0. W przeciwnym razie nastąpi blokada procesu lub błąd wykonania funkcji `semop`, zależnie od ustawienia flagi `IPC_NOWAIT` i żadna z operacji semaforowych zdefiniowanych w tablicy *sops* nie zostanie wykonana. `sem_op` = 0 oznacza oczekiwanie na osiągnięcie wartości 0 przez zmienną semaforową. Flagi określają dodatkowe cechy operacji (np. `IPC_NOWAIT`, `SEM_UNDO`).

`semctl(int semid, int semnum, int cmd, union semun arg)` — operacje kontrolne na tablicy semaforów.

5.3 Kolejki komunikatów

`msgget(key_t key, int msgflg)` — utworzenie lub pobranie identyfikatora kolejki komunikatów. Funkcja tworzy kolejkę komunikatów, jeśli kolejka o kluczu *key* jeszcze nie istnieje, i zwraca identyfikator tej kolejki. Parametr *msgflg* umożliwia przekazanie praw dostępu do kolejki oraz pewnych dodatkowych flag definiujących sposób jej tworzenia (np. `IPC_CREAT`), połączonych z prawami dostępu operatorem sumy bitowej (np. `IPC_CREAT|0660`).

`msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg)` — wysłanie komunikatu przez przekazanie go do kolejki. Funkcja umieszcza w kolejce identyfikowanej przez *msgid* komunikat, którego treść znajduje się pod adresem *msgp* w przestrzeni adresowej procesu i zawiera *msgsz* bajtów we właściwej treści komunikatu oraz wartość typu `long` określającą typ komunikatu. Ogólna struktura komunikatu zdefiniowana jest następująco:

```
struct msgbuf {
   long mtype;     /* typ komunikatu, > 0 */
   char mtext[1];  /* treść komunikatu */
};
```

Treść komunikatu może mieć w rzeczywistości inny rozmiar i inną strukturę. Jeśli w kolejce nie ma miejsca to proces jest blokowany w funkcji `msgsnd` lub — w przypadku ustawienia flagi `IPC_NOWAIT` w parametrze *msgflg* — zwracana jest wartość -1.

`msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)` — odebranie komunikatu przez pobranie go z kolejki. Funkcja pobiera z kolejki komunikat który spełnia kryteria określone przez *msgtyp* (typ komunikatu), pmsgsz (rozmiar bufora w przestrzeni adresowej procesu) i *msgflg* (flagi specyfikujące zachowanie się funkcji w warunkach ńietypowych"). Wybór komunikatu dokonuje się według następujących zasad: jeśli parametr *msgflg* ma ustawioną flagę `MSG_NOERROR`, komunikat przekraczający rozmiar bufora jest ucinany przy odbiorze. W przeciwnym razie odbierane są tylko komunikaty, których treść ma mniejszy rozmiar niż *msgsz*. Drugim kryterium wyboru jest typ komunikatu, zgodnie z poniższą regułą:

- *msgtyp* > 0 — wybierany jest komunikat, którego typ jest dokładnie taki, jak *msgtyp*,
- *msgtyp* < 0 — wybierany jest komunikat, którego który ma najmniejszą wartość typu mniejszą lub równą bezwzględnej wartości z *msgtyp*,
- *msgtyp* $= 0$ — typ komunikatu nie jest brany pod uwagę przy wyborze.

Komunikaty spełniające kryteria pobierane są w kolejności, w której zostały umieszczone w kolejce. Jeśli w kolejce nie ma wymaganego komunikatu i w parametrze *flag* ustawiona jest flaga `IPC_NOWAIT`, funkcja zwraca -1. Jeśli flaga nie jest ustawiona, proces jest blokowany aż do czasu pojawienia się komunikatu. Funkcja zwraca rozmiar treści odebranego komunikatu (liczbę bajtów zajmowaną przez `mtext`).

`msgctl(int msqid, int cmd, struct msqid_ds *buf)` — operacje kontrolne na kolejce komunikatów.

6 Sygnały

`kill(pid_t pid, int signum)` — wysłanie sygnału do procesu. Funkcja powoduje wysłanie sygnału *signum* do procesu o identyfikatorze *pid*.

`void (*signal(int signum, void (*handler)(int)))(int)` — zdefiniowanie sposobu reakcji procesu na sygnał. Funkcja umożliwia ustawienie ignorowania sygnału o numerze *signum* (jako *handler* przekazywana jest stała `SIG_IGN`), ustawienie domyślnej reakcji (jako *handler* przekazywana jest stała `SIG_DFL`) lub przechwytywanie sygnału. W przypadku przechwytywania jako wartość aktualna parametru *handler* przekazywany jest wskaźnik na funkcję, która będzie wywoływana asynchronicznie w reakcji na otrzymanie sygnału *signum*. Funkcja zwraca wskaźnik na poprzednią funkcją obsługi sygnału lub odpowiednią stałą (`SIG_DFL`, `SIG_IGN`).