

Mechanizmy IPC

Opis funkcji wykorzystanych w programie:

```
int shmget(key_t key, int size, int shmflg);
```

Zwraca deskryptor segmentu pamięci wspólnej, skojarzonego z wartością (kluczem) przekazaną w parametrze key.

IPC_CREAT, aby utworzyć nowy segment. Jeśli ten znacznik nie zostanie ustawiony, to shmget() spróbuje znaleźć segment skojarzony z key i sprawdzić, czy użytkownik ma uprawnienia dla dostępu do segmentu.

IPC_EXCL przekazane łącznie z IPC_CREAT zapewnia sygnalizację błędu, jeśli segment już istnieje. mode_flags (9 najmniej znaczących bitów) określa prawa dostępu do segmentu dla jego właściciela, grupy oraz reszty świata. Prawa uruchamiania nie są obecnie przez system używane.

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Funkcja

shmat dołącza segment pamięci wspólnej o deskryptorze shmid do przestrzeni adresowej procesu, który ją wywołał. Adres, pod którym segment ma być widoczny jest przekazywany parametrem shmaddr, przy czym system może przetworzyć ten adres w następujący sposób:

Jeśli shmaddr jest równy NULL, wówczas system sam wybierze odpowiedni (nieużywany) adres, pod którym segment będzie widoczny.

```
void *shmdt(int shmid);
```

Funkcja

shmdt wyłącza segment pamięci wspólnej odwzorowany pod adresem podanym w shmaddr.

```
int semctl(int semid, int semnum, int cmd, ...);
```

Funkcja semctl wykonuje operację sterującą określoną przez cmd na zestawie semaforów określonym przez semid lub na semnum-tym semaforze tego zestawu. (Numeracja semaforów zaczyna się od 0.)

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

Funkcja semop wykonuje operacje na wybranych semaforach z zestawu wskazywanego przez semid. Każdy z nsops elementów tablicy wskazywanej przez parametr sops określa operację, która ma być wykonana na semaforze. Struktura struct sembuf zawiera następujące pola:

```
unsigned short sem_num;      /* numer semafora */
short sem_op;               /* operacja na semaforze */
short sem_flg;              /* dodatkowe znaczniki operacji */
```

Problem czytelników i pisarzy:

Dowolna liczba procesów czyta lub pisze do jednego pliku. Dowolne z nich mogą jednocześnie czytać. Jeśli jakiś pisze to inne nie piszą ani nie czytają. W rozwiązaniu nie wolno zagłodzić żadnego wątku.

```

int main(void) {
    int reader_id = 0, writer_id = 0;
    int mux = semget(IPC_PRIVATE, 2, 0600 | IPC_CREAT | IPC_EXCL);
    if (mux < 0)
        perror("semget");
    semval.val = 1;
    if (semctl(mux, 0, SETVAL, semval) < 0)
        perror("semctl 0");
    if (semctl(mux, 1, SETVAL, semval) < 0)
        perror("semctl 1");
    int shmid = shmget(IPC_PRIVATE, 4096, 0600 | IPC_CREAT | IPC_EXCL);
    if (shmid < 0)
        perror("shmget");
    srand(time(NULL));
    int i;
    for (i = 0; i < NR_PROC; i++) {
        int choice = rand() % 2;
        if (choice)
            reader_id++;
        else
            writer_id++;
        int pid = fork();
        if (pid == -1)
            perror("fork");
        if (pid == 0) {
            if (choice)
                reader(reader_id, mux, shmid);
            else
                writer(writer_id, mux, shmid);
            return 0;
        }
    }
    for (i = 0; i < NR_PROC; i++)
        if (wait(NULL) < 0)
            perror("wait");
    if (shmctl(shmid, IPC_RMID, NULL) < 0)
        perror("shmctl");
    if (semctl(mux, 0, IPC_RMID, NULL) < 0)
        perror("semctl IPC_RMID");
    return 0;
}

```

W funkcji main() pobieramy deskryptor do 2-elementowej tablicy semaforów, a następnie za pomocą struktury union semun zdefiniowanej w <sys/sem.h> i funkcji semctl() przypisujemy semaforom początkową wartość 1. Pobieramy deskryptor wspólnej pamięci. W pętli for, która wykonuje się podaną liczbę razy u nas 16, tworzymy nowe procesy czytelników albo pisarzy, o czym decyduje zmienna choice (wynik funkcji rand()). Po przejściu całej pętli, wywołujemy wait(), który zmusza do czekania na zakończenie procesu potomnego i sprzątamy po sobie zaznaczając segmenty do usunięcia.

W funkcjach writer() i reader() jest umieszczony kod, który wykonują procesy pisarzy i czytelników odpowiednio. Kod działa zgodnie z podanym przykładem:

```

do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    /* ... */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);

```

Figure 5.12 The structure of a reader process.

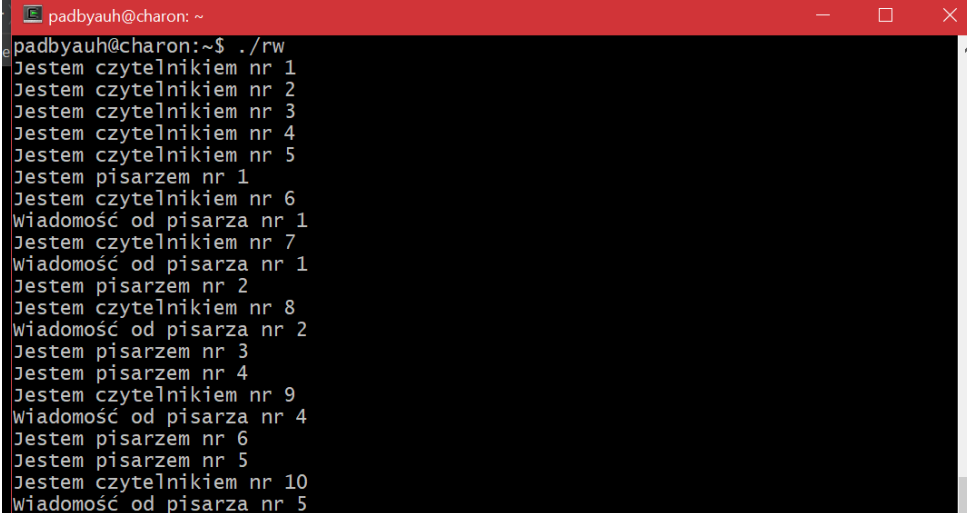
```

do {
    wait(rw_mutex);
    /* ... */
    /* writing is performed */
    /* ... */
    signal(rw_mutex);
} while (true);

```

Figure 5.11 The structure of a writer process.

Przykład działania programu:

A terminal window with a red title bar containing the text 'padbyauh@charon: ~'. The terminal output consists of 20 lines of text. The first 10 lines are 'Jestem czytelnikiem nr' followed by numbers 1 through 10. The next 10 lines are 'wiadomość od pisarza nr' followed by numbers 1 through 5, each appearing twice.

```
padbyauh@charon:~$ ./rw
Jestem czytelnikiem nr 1
Jestem czytelnikiem nr 2
Jestem czytelnikiem nr 3
Jestem czytelnikiem nr 4
Jestem czytelnikiem nr 5
Jestem pisarzem nr 1
Jestem czytelnikiem nr 6
wiadomość od pisarza nr 1
Jestem czytelnikiem nr 7
wiadomość od pisarza nr 1
Jestem pisarzem nr 2
Jestem czytelnikiem nr 8
wiadomość od pisarza nr 2
Jestem pisarzem nr 3
Jestem pisarzem nr 4
Jestem czytelnikiem nr 9
wiadomość od pisarza nr 4
Jestem pisarzem nr 6
Jestem pisarzem nr 5
Jestem czytelnikiem nr 10
wiadomość od pisarza nr 5
```