6
Pamięć współdzielona

6.1 Dostęp do pamięci współdzielonej

1. Utwórz blok pamięci współdzielonej korzystając z poniższego kodu:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
  int id;
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  printf("Blok utworzony: %d\n", id);
  return 0;
}
```

2. Po utworzeniu bloku sprawdź jego istnienie komendą `ipcs`:

```
# ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000111 3342338    sobaniec   700        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

3. Usuń blok pamięci komendą `ipcrm`:

```
# ipcrm -M 0x111
```

lub:

```
# ipcrm -m 3342338
```

4. Dołącz blok pamięci współdzielonej do przestrzeni adresowej procesu i wpisz dowolny napis do tego bloku:

```
char* ptr;
...
ptr = shmat(id, NULL, 0);
strcpy(ptr, "Ala ma kota");
```

5. Napisz program, który będzie odczytywał zawartość łańcucha tekstowego z pamięci współdzielonej.
6. Odłącz blok pamięci współdzielonej od przestrzeni adresowej procesu:

```
char* ptr;
...
shmdt(ptr);
```

7. Usuń blok pamięci współdzielonej:

```
id = shmget(0x111, 1024, 0);
...
shmctl(id, IPC_RMID, NULL);
```

6.2 Synchronizacja procesów

1. Napisz dwa programy, zapisujący i odczytujący z bloku pamięci współdzielonej. Program piszący powinien zapisywać do pamięci naprzemiennie, w to samo miejsce napisy: „xxxxxx" i „oooooo". Program czytający odczytuje zawartość bufora i sprawdza czy jest to jeden z wpisywanych napisów. W przypadku błędu, wypisuje niepoprawny łańcuch na ekranie.
2. Przeanalizuj działanie programu z przykładu 6.1 przesyłającego napisy pomiędzy procesami. Czy zaproponowana synchronizacja jest wystarczająca? Porównaj przedstawione rozwiązanie z przykładami 6.2 i 6.3. Przetestuj działanie programu z przykładu 6.2 po dodaniu dodatkowego wywołania `sleep(1)` zaraz za wywołaniem funkcji `printf` w procesie macierzystym.

Przykład 6.1: Synchronizacja procesow — wersja 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

void obsluga(int sig)
{
  printf("sygnał w procesie %d\n", getpid());
}

int main()
{
  int id;
  char *ptr;
  int pid;
  signal(SIGHUP, obsluga);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    sleep(1);
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      sleep(1);
      kill(ppid, 1);
      pause();
      strcpy(ptr, "oooooo");
      sleep(1);
      kill(ppid, 1);
      pause();
    }
  }
  else
  {
    while(1)
    {
      pause();
      printf("%s\n", ptr);
      sleep(1);
      kill(pid, 1);
    }
  }
}
```

Przykład 6.2: Synchronizacja procesow — wersja 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int ok = 0;

void obsluga(int sig) {
  ok = 1;
}

int main()
{
  int id;
  char *ptr;
  int pid;
  signal(SIGHUP, obsluga);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      ok = 0;
      kill(ppid, 1);
      while(!ok);
      strcpy(ptr, "oooooo");
      ok = 0;
      kill(ppid, 1);
      while(!ok);
    }
  }
  else
  {
    while(1)
    {
      while(!ok);
      printf("%s\n", ptr);
      ok = 0;
      kill(pid, 1);
    }
  }
}
```

Przykład 6.3: Synchronizacja procesow — wersja 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

void obsluga(int sig)
{
}

int main()
{
  int id;
  char *ptr;
  int pid;
  sigset_t mask, mask_old;
  signal(SIGHUP, obsluga);
  sigemptyset(&mask);
  sigaddset(&mask, SIGHUP);
  sigprocmask(SIG_BLOCK, &mask, &mask_old);
  id = shmget(0x111, 1024, 0600 | IPC_CREAT);
  if (id==-1)
  {
    perror("shmget");
    exit(1);
  }
  ptr = (char*)shmat(id, NULL, 0);
  if ((pid=fork())==0)
  {
    int ppid = getppid();
    while(1)
    {
      strcpy(ptr, "xxxxxx");
      kill(ppid, 1);
      sigsuspend(&mask_old);
      strcpy(ptr, "oooooo");
      kill(ppid, 1);
      sigsuspend(&mask_old);
    }
  }
  else
  {
    while(1)
    {
      sigsuspend(&mask_old);
      printf("%s\n", ptr);
      kill(pid, 1);
    }
  }
}
```

7 Semafory

7.1 Obsługa semaforów

1. Poniższy program tworzy semafor i wykonuje na nim podstawowe operacje.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

int main()
{
  int sid;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  op.sem_num = 0;
  op.sem_flg = 0;

  /* zwi kszenie warto ci semafora o 1 */
  op.sem_op = 1;
  semop(sid, &op, 1);

  /* zmniejszenie warto ci semafora o 1 */
  op.sem_op = -1;
  semop(sid, &op, 1);

  return 0;
}
```

2. Po uruchomieniu sprawdź istnienie semafora w systemie komendą `ipcs`:

```
# ipcs
------ Shared Memory Segments --------
```

```
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000123 32768      sobaniec   700        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

3. Usuń semafor z systemu komendą `ipcrm`:

```
# ipcrm -S 0x123
```

lub:

```
# ipcrm -s 32768
```

4. Odczytaj bieżącą wartość semafora i ustaw ją na wybraną wartość:

```
int num;
...
num = semctl(sid, 0, GETVAL, 0);
printf("semaphore value: %d\n", num);
num = 1;
semctl(sid, 0, SETVAL, num);
```

5. Poniższy program umożliwia wykonywanie operacji *P* i *V* na semaforze. Sprawdź zachowanie programu w przypadku zmniejszania wartości semafora poniżej zera.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

int main(int argc, char* argv[])
{
  int sid;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  printf("przed: %d\n", semctl(sid, 0, GETVAL, NULL));
  op.sem_num = 0;
  op.sem_flg = 0;
  op.sem_op = atoi(argv[1]);
  semop(sid, &op, 1);
  printf("po:    %d\n", semctl(sid, 0, GETVAL, NULL));

  return 0;
}
```

6. Uruchom kilkukrotnie poniższy program, który wykonuje operacje w sekcji krytycznej. Pierwsza kopia powinna być uruchomiona z dodatkowym argumentem powodując w ten sposób zainicjowanie semafora.

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/sem.h>

int main(int argc, char* argv[])
{
  int sid;
  int num;
  struct sembuf op;

  sid = semget(0x123, 1, 0600 | IPC_CREAT);
  if (sid == -1)
  {
    perror("semget");
    exit(1);
  }

  if (argc>1)
  {
    printf("Inicjacja semafora...\n\n");
    num = 1;
    semctl(sid, 0, SETVAL, num);
  }

  op.sem_num = 0;
  op.sem_flg = 0;
  while(1)
  {
    printf("Wchodzę do sekcji krytycznej...\n");
    op.sem_op = -1;
    semop(sid, &op, 1);
    printf("Jestem w sekcji krytycznej ...\n");
    sleep(5);
    op.sem_op = 1;
    semop(sid, &op, 1);
    printf("Wyszedłem z sekcji krytycznej ...\n");
    sleep(1);
  }
  return 0;
}
```

7.2 Problem producenta-konsumenta

1. Zrealizuj wymianę danych pomiędzy dwoma procesami (producentem i konsumentem) synchronizując ich pracę semaforami. Dane powinny być przechowywane w pamięci współdzielonej w postaci wieloelementowej tablicy. Praca procesów może się odbywać współbieżnie o ile dotyczy rozłącznych bloków pamięci.

   Przykład 7.1 przedstawia przykładową implementację procesu producenta. Na podstawie kodu opracuj implementację procesu konsumenta.

2. Czy implementacja procesu producenta z przykładu 7.1 jest wystarczająca w przypadku współbieżnej pracy wielu procesów produkujących dane? Jeżeli nie, to jakie zmiany należałoby wprowadzić?

Przykład 7.1: Proces producenta

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define N 10         /* rozmiar wspó dzielonej tablicy */

int main()
{
  int shmid;         /* systemowy identyfikator bloku pami ci dzielonej */
  int semid;         /* systemowy identyfikator semafora */
  int *tab;          /* wska nik na tablic  z danymi */
  int pos;           /* pozycja w tablicy */
  int num;           /* aktualnie zapisywana warto   */
  struct sembuf op;  /* struktura dla operacji na semaforze */

  shmid = shmget(0x123, sizeof(int)*N, 0600 | IPC_CREAT);
  if (shmid == -1)
  {
    perror("shmget");
    exit(1);
  }
  tab = (int*)shmat(shmid, NULL, 0);
  if (tab == NULL)
  {
    perror("shmat");
    exit(1);
  }

  /* utworzenie nowego dwuelementowego semafora
   * 0 - semafor producenta
   * 1 - semafor konsumenta
   */
  semid = semget(0x234, 2, 0600 | IPC_CREAT);
  if (semid == -1)
  {
    perror("semget");
    exit(1);
  }
  /* ustawienie warto ci pocz tkowych semaforów */
  num = N;
  semctl(semid, 0, SETVAL, num);
  num = 0;
  semctl(semid, 1, SETVAL, num);

  op.sem_flg = 0;
  pos = 0;
  num = 1;
  while(1)
  {
    /* opuszczenie semafora producenta */
    op.sem_num = 0;
    op.sem_op = -1;
    semop(semid, &op, 1);

    /* zapis elementu do tablicy */
    tab[pos] = num;
    printf("Na pozycji %d zapisałem %d\n", pos, num);
    pos = (pos + 1) % N;
    num++;
    sleep(1);

    /* podniesienie semafora konsumenta */
    op.sem_num = 1;
    op.sem_op = 1;
    semop(semid, &op, 1);
  }
}
```

8
Kolejki komunikatów

1. Utwórz kolejkę komunikatów wywołując funkcję `msgget(2)`:

   ```
   mid = msgget(0x123, 0600 | IPC_CREAT);
   ```

   Sprawdź komendą `ipcs` czy kolejka komunikatów faktycznie została utworzona. Zadbaj o obsługę ewentualnych błędów poprzez sprawdzenie zwracanej przez funkcję wartości.

2. Zadeklaruj w programie strukturę do reprezentowania wiadomości. Pierwsze pole w strukturze służy do przechowywania informacji o typie komunikatu. Pole to musi być typu `long`. Pozostałe pola mogą być dowolnego typu. Poniżej przedstawiono przykładową strukturę do przesyłania wiadomości tekstowych:

   ```
   typedef struct
   {
     long type;
     char text[1024];
   } MSG;
   ```

3. Wyślij wiadomość korzystając z funkcji `msgsnd(2)`:

   ```
   MSG msg;
   ...
   msgsnd(mid, &msg, strlen(msg.text)+1, 0);
   ```

   Po wysłaniu wiadomości sprawdź komendą `ipcs` czy kolejka faktycznie zawiera nową wiadomość.

4. Odbierz wiadomość z kolejki korzystając z funkcji `msgrcv(2)`:

   ```
   MSG msg;
   ...
   msgrcv(mid, &msg, 1024, 0, 0);
   ```

   Trzeci argument funkcji `msgrcv` reprezentuje rozmiar przekazanej struktury reprezentującej komunikat z pominięciem początkowego pola typu `long`. Czwarty argument wskazuje na typ wiadomości, która powinna zostać pobrana z kolejki:

   - wartość 0 oznacza, że ma być pobrana pierwsza wiadomość z kolejki,
   - wartość dodatnia oznacza, że ma być pobrana pierwsza wiadomość z kolejki o typie równym argumentowi,

- wartość ujemna oznacza, że ma być pobrana pierwsza wiadomość z kolejki o typie mniejszym lub równym wartości bezwzględnej argumentu.

Ostatni argument to dodatkowe flagi doprecyzowujące tryb pracy funkcji `msgrcv`.

5. Usuń kolejkę komunikatów korzystając z funkcji `msgctl`:

```
msgctl(mid, IPC_RMID, NULL);
```

Usuwanie można również przeprowadzić poleceniem `ipcrm`.

6. Napisz program serwera, który w pętli nieskończonej odbiera komunikaty z kolejki i wyświetla je na ekranie. Z programem tym komunikuje się program klienta, który wysyła wiadomość przekazaną jako argument w linii poleceń.
7. Zmodyfikuj implementację serwera z poprzedniego przykładu tak, aby serwer odpowiadał na żądanie klienta odsyłając tą samą wiadomość tekstową, ale z zamienionymi wszystkimi literami na wielkie.

   Podpowiedź: komunikację dwustronną można przeprowadzić poprzez wykorzystanie różnych typów komunikatów. Pierwszy argument dla programu klienta powinien więc określać typ komunikatów odbieranych przez tego klienta.
8. Przetestuj odbiór wiadomości w trybie nieblokującym.
9. Sprawdź czy są jakieś wiadomości w kolejce bez ich odbierania.