Pamięć dzielona i kolejki komunikatów

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/
oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

POSIX vs. System V (IPC)

- W systemach Unix dostępne są 2 interfejsy do programowania komunikacji międzyprocesowej: POSIX i IPC
- Oba oferują tę sama funkcjonalność: semafory, pamięć dzieloną oraz kolejki komunikatów.
- W dalszej części zajęć będą dokładniej omówione wybrane mechanizmy komunikacji z jednego lub drugiego interfejsu, które uznano za wygodniejsze.

Definicja semafora

- Semafor S jest obiektem systemu operacyjnego z którym związany jest licznik L zasobu przyjmujący wartości nieujemne. Na semaforze zdefiniowane są *atomowe* operacje **sem_init**, **sem_wait** i **sem_post**.

Operacja	Oznaczenie	Opis
Inicjacja semafora S	sem_init(S,N)	Ustawienie licznika semafora S na początkową wartość N .
Zajmowanie	sem_wait(S)	Gdy licznik L semafora S jest dodatni (L > 0) zmniejsz go o 1 (L = L – 1). Gdy licznik L semafora S jest równy zero (L= 0) zablokuj proces bieżący.
Sygnalizacja	sem_post(S)	Gdy istnieje jakiś proces oczekujący na semaforze S to odblokuj jeden z czekających procesów. Gdy brak procesów oczekujących na semaforze S zwiększ jego licznik L o 1 (L=L+1).

Semafory nazwane POSIX (1)

Semafory nazwane identyfikowane są w procesach poprzez ich nazwę.

Funkcja tworząca semafor:

```
sem_t *sem_open(const char *name,  int oflag, mode_t mode , int value);
sem_t *sem_open(const char *name, int oflag);
```

Parametry:
name – nazwa semafora w systemie, powinna zacząć się od znaku „/”.
oflag – jest ustawiana na O_CREAT gdy chcemy utworzyć semafor (jeśli dodamy flagę O_EXCL funkcja zwróci błąd, w przypadku, gdy taki semafor już istnieje).
mode_t – ustalenie kontroli dostępu do semafora
value – ustalenie wartości początkowej semafora.

Funkcja zwraca „uchwyt” do semafora lub błąd SEM_FAILED z zapisaną odpowiednią wartością w zmiennej errno. Od tej chwili w programie dostęp do semafora odbywa się za pomocą tego „uchwytu”.

Pojedyncze wywołanie tworzy semafor, inicjalizuje go i ustala zasady dostępu do niego.

Semafory nazwane POSIX (2)

Funkcja inicjalizująca semafor:

int sem_init(sem_t *sem, int pshared, unsigned int value);

pshared : 0-semofor jest dzielony pomiędzy wątki, >0 może być dzielony pomiędzy procesy

value: początkowa wartość semafora

Funkcja „opuszczająca" semafor:

int sem_wait(sem_t *sem);

Funkcja „podnosząca" semafor:

int sem_post(sem_t *sem);

Funkcja zamykająca dostęp do semafora:

int sem_close(sem_t *sem);

Funkcja usuwająca semafor z sytemu:

int sem_unlink(const char *name);

Uwagi: Po zakończeniu korzystania z semafora należy zamknąć do niego dostęp wywołując funkcję sem_close(). Usunięcie semafora z sytemu odbywa się po wywołaniu funkcji sem_unlink(). Jeśli jakiś inny proces lub wątek mają dostęp do tego semafora, to wywołanie funkcji sem_unlink() nie da żadnego efektu.

Odpowiadający interfejs semaforów IPC

```
#include <sys/sem.h>
// Bezpośrednie sterowanie semaforem:
int semctl(int sem_id, int sem_num, int command, ...);

// Utworzenie semafora/semaforów lub dowiązanie do istniejącego:
int semget(key_t key, int num_sems, int sem_flags);

//Zmiana wartości semafora:
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

Przykładowa aplikacja z zastosowaniem semaforów IPC (1)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
 union semun {
 int val;                    /* value for SETVAL */
 struct semid_ds *buf;    /* buffer for IPC_STAT, IPC_SET */
unsigned short int *array;  /* array for GETALL, SETALL */
struct seminfo *__buf;      /* buffer for IPC_INFO */    };

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;

int main(int argc, char *argv[])
{    int i;     int pause_time;    char op_char = 'O';
     srand((unsigned int)getpid());
     sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
    if (argc > 1) {
       if (!set_semvalue()) {
          fprintf(stderr, "Failed to initialize semaphore\n");
          exit(EXIT_FAILURE);}
       op_char = 'X';
       sleep(2);}
```

```
   for(i = 0; i < 10; i++) {
      if (!semaphore_p()) exit(EXIT_FAILURE);
      printf("%c", op_char);fflush(stdout);
      pause_time = rand() % 3;
      sleep(pause_time);
      printf("%c", op_char);fflush(stdout);
      if (!semaphore_v()) exit(EXIT_FAILURE);
      pause_time = rand() % 2;
      sleep(pause_time);}

   printf("\n%d - finished\n", getpid());
   if (argc > 1) { sleep(10);   del_semvalue();}
   exit(EXIT_SUCCESS); }
```

Przykładowa aplikacja z zastosowaniem semaforów IPC (2)

```
static int set_semvalue(void)
{   union semun sem_union;
    sem_union.val = 1;
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1)
                  return(0);
    return(1);}

static void del_semvalue(void)
{    union semun sem_union;
     if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");}
```

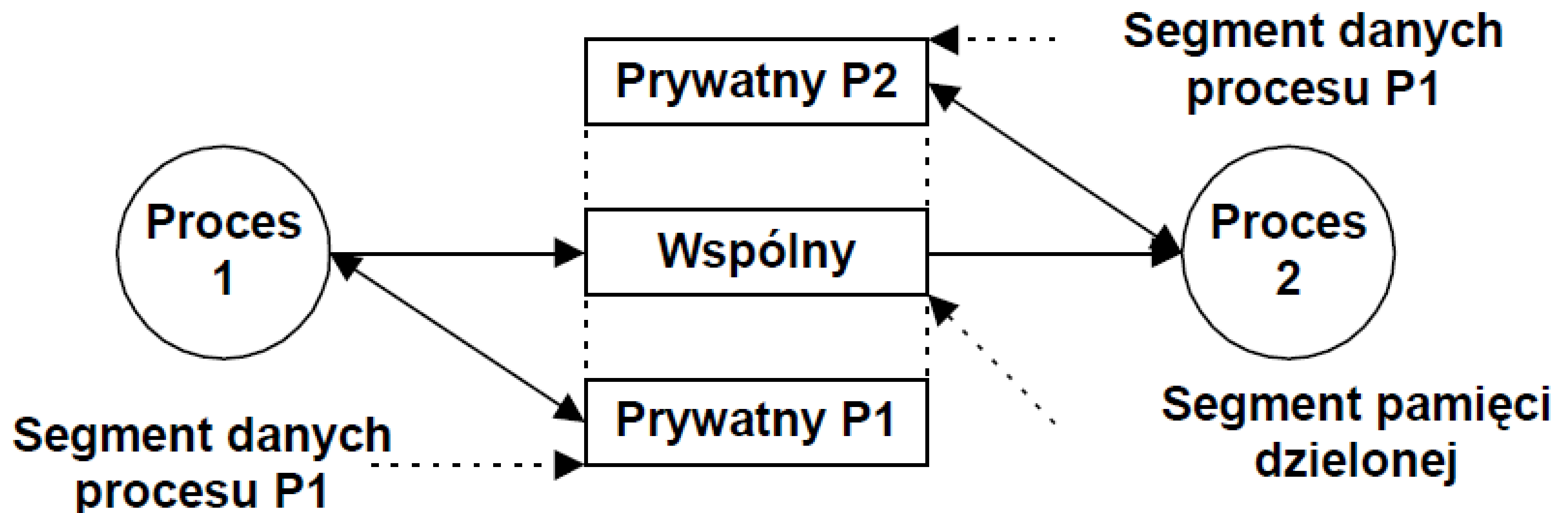
```
static int semaphore_p(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
       fprintf(stderr, "semaphore_p failed\n");
       return(0);
    }
    return(1);
}

static int semaphore_v(void)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
       fprintf(stderr, "semaphore_v failed\n");
       return(0);
    }
    return(1);
}
```

Wynik przykładowej sesji:

```
$ ./sem1 1 &
[1] 1082
$ ./sem1
OOXXOOXXOOXXOOXXOOXXOOOOXXOOXXOOXXOOXXXX
1083 - finished
1082 - finished
$
```

Pamięć dzielona

- Do komunikacji między rozdzielnymi procesami można zastosować mechanizm pamięci dzielonej – specjalnie zaalokowanego obszaru pamięci w obszarze procesu, w którym zastosowana jest wydzielona konwencja nazewnicza.



Pamięć dzielona IPC (1)

```
#include <sys/shm.h>
// Podłączenie segmentu pamięci dzielonej do przestrzeni adresowej procesu:
void *shmat(int shm_id, const void *shm_addr, int shmflg);

// Funkcja bezpośrednio sterująca pamięcią dzieloną:
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);

// Funkcja odłączająca pamięć dzieloną od bieżącego procesu:
int shmdt(const void *shm_addr);

// Funkcja tworząca pamięć dzieloną
int shmget(key_t key, size_t size, int shmflg);
```

shmget()

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

- key – „nazwa” dzielonego segmentu w pamięci
- size – rozmiar dzielonego obszaru pamięci
- shmflg – znaczniki zezwoleń
- funkcja zwraca: identyfikator pamięci dzielonej (liczba dodatnia) lub -1 w przypadku błędu.

shmat()

void *shmat(int shm_id, const void *shm_addr, int shmflg);

- shm_id – identyfikator pamięci dzielonej (zwracany przez shmget)
- shm_addr – wskazanie, gdzie pamięć ma być przyłączona do procesu (zwykle wskaźnik pusty)
- shmflg – znaczniki zezwoleń (zwykle 0)

Pamięć dzielona IPC – przykład – program 1

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define TEXT_SZ 2048
struct shared_use_st {
   int written_by_you;
   char some_text[TEXT_SZ];};

int main()
{   int running = 1;
   void *shared_memory = (void *)0;
   struct shared_use_st *shared_stuff;
   int shmid;
   srand((unsigned int)getpid());
   shmid = shmget((key_t)1234,
   sizeof(struct shared_use_st), 0666 | IPC_CREAT);
   if (shmid == -1) {  fprintf(stderr, "shmget failed\n");
                               exit(EXIT_FAILURE);
   }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);
   }
```

```
printf("Memory attached at %X\n", (int)shared_memory);

shared_stuff = (struct shared_use_st *) shared_memory;
shared_stuff->written_by_you = 0;
 while(running) {
      if (shared_stuff->written_by_you) {
         printf("You wrote: %s", shared_stuff->some_text);
         sleep( rand() % 4 );
         shared_stuff->written_by_you = 0;
         if (strncmp(shared_stuff->some_text,
                  "end", 3) == 0) {
            running = 0;
         }
      }
   }
   if (shmdt(shared_memory) == -1) {
      fprintf(stderr, "shmdt failed\n");
      exit(EXIT_FAILURE);
   }

   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      fprintf(stderr, "shmctl(IPC_RMID) failed\n");
      exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}
```

Pamięć dzielona IPC –przykład - program 2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define TEXT_SZ 2048
struct shared_use_st {
   int written_by_you;
   char some_text[TEXT_SZ];};
int main()
{
   int running = 1;
   void *shared_memory = (void *)0;
   struct shared_use_st *shared_stuff;
   char buffer[BUFSIZ];
   int shmid;
   shmid = shmget((key_t)1234,
       sizeof(struct shared_use_st), 0666 | IPC_CREAT);
   if (shmid == -1) {
      fprintf(stderr, "shmget failed\n");
      exit(EXIT_FAILURE);    }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);    }

   printf("Memory attached at %X\n",
(int)shared_memory);

shared_stuff = (struct shared_use_st *)
                                   shared_memory;
   while(running) {
      while(shared_stuff->written_by_you == 1) {
         sleep(1);
         printf("waiting for client...\n");
      }
      printf("Enter some text: ");
      fgets(buffer, BUFSIZ, stdin);

      strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
      shared_stuff->written_by_you = 1;

      if (strncmp(buffer, "end", 3) == 0) {
            running = 0;
      }
   }
   if (shmdt(shared_memory) == -1) {
      fprintf(stderr, "shmdt failed\n");
      exit(EXIT_FAILURE);
   }
   exit(EXIT_SUCCESS);
}
```

Uruchomienie i rezultat działania programów:

```
$ ./shm1 &
[1] 294
Memory attached at 40017000
$ ./shm2
Memory attached at 40017000
Enter some text: hello
You wrote: hello
waiting for client...
waiting for client...
Enter some text: Linux!
You wrote: Linux!
waiting for client...
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
$
```

Jak to działa?

- Pierwszy program (shm1) tworzy segment pamięci dzielonej i dołącza do swojej przestrzeni adresowej
- Na pierwszą część pamięci dzielonej rzutowana jest struktura typu shared_use_st, w strukturze znajduje się znacznik written_by_you, który jest ustawiany, kiedy dane staną się dostępne
- Po wykryciu, że znacznik jest ustawiony program odczytuje tekst, wyświetla go, a następnie zeruje znacznik, aby wskazać, że odczytał dane
- Jeśli przekazano tekst „end” następuje zamknięcie pętli odczytującej teksty
- Drugi program (shm2) pobiera i dołącza ten sam segment pamięci (korzystając z tego samego klucza 1234)
- Następnie prosi użytkownika o wprowadzenie tekstu.
- Jeśli znacznik written_by_you jest ustawiony, to oznacza, że drugi proces nie odczytał jeszcze poprzednio zapisanych danych i czeka na wyzerowanie znacznika
- Kiedy pierwszy proces to uczyni, to drugi zapisuje nowe dane do pamięci dzielonej i ustawia znacznik written_by_you
- Po odebraniu tekstu „end” również i ten program kończy działanie
- **Rozwiązanie komunikacji jest nieeleganckie wato do sygnalizacji gotowości do odczytu zastosować sygnały, semafory, potoki lub kolejki komunikatów…**

Uwagi

- Należy pamiętać, że mechanizm pamięci dzielonej nie zawiera w sobie metod synchronizacji i należy je dodać.
- W systemach Unix proponuje się stosowanie pamięci dzielonej do współużywania dużych ilości danych przez wiele procesów
- To współużywanie uzupełnia się semaforami lub kolejkami do wymiany mniejszych komunikatów sterujących dostępem do pamięci dzielonej.

Funkcje API pamięci współdzielonej standardu POSIX

Funkcja	Zastosowanie
shm_open()	Tworzy obszar pamięci współdzielonej lub dołącza nowy obszar do już istniejącego. Obszary pamięci współdzielonej są reprezentowane przez nazwę, a funkcja shm_open zwraca deskryptor pliku.
shm_unlink()	Usuwa obszar pamięci współdzielonej dostępnego przez deskryptor pliku. Obszar pamięci nie jest usuwany dopóki wszystkie procesy z nim powiązane z niego nie zrezygnują. Jednak po wywołaniu tej funkcji żaden nowy proces nie może z obszaru skorzystać.
mmap()	Odwzorowuje plik w pamięci danego procesu. Funkcja zwraca wskaźnik do właśnie odwzorowanej pamięci.
munmap()	Usuwa odwzorowanie obszaru pamięci odwzorowanego wcześniej za pomocą wywołania mmap.
msync()	Synchronizuje dostęp do obszaru pamięci odwzorowanej za pomocą wywołania mmap, zapisując wszystkie buforowane dane do pamięci fizycznej w sposób umożliwiający pozostałym procesom dostęp do zmian
ftruncate()	Ustala rozmiar pliku do podanego rozmiaru.

Tworzenie obszaru pamięci dzielonej

int shm_open(char *name, int oflag, mode_t mode)
name - nazwa segmentu pamięci
oflag - flaga specyfikująca tryb utworzenia (jak dla plików), np. O_RDONLY, O_RDWR, O_CREAT
mode - specyfikacja trybu dostępu (jak dla plików).

Gdy funkcja zwraca liczbę nieujemną jest to uchwyt identyfikujący segment w procesie. Segment widziany jest jako plik specjalny w katalogu /dev/shmem

Ustalanie rozmiaru pamięci dzielonej (pliku)

int ftruncate(int fildes, off_t length);
filedes – uchwyt identyfikujący segment pamięci w procesie
length - rozmiar do którego ma być zawężony lub zwiększony plik.

Funkcja zwraca 0 w przypadku sukcesu, lub -1 w przypadku błędu. Zmienna errno zawiera kod wykrytego błędu.

Odwzorowanie segmentu pamięci wspólnej w obszar procesu

void *mmap(void * addr, size_t len, int prot, int flags, int fdes, off_t off)
addr - zmienna wskaźnikowa w procesie, której wartość będzie przez funkcję zainicjowana. Może być 0.
len - wielkość odwzorowywanego obszaru.
prot - specyfikacja dostępu do obszaru opisana w <sys/mman.h>. Może być PROT_READ|PROT_WRITE
flags - specyfikacja użycia segmentu, np. MAP_SHARED.
fdes - uchwyt segmentu wspólnej pamięci.
off - początek obszaru we wspólnej pamięci (musi to być wielokrotność strony 4K)

Odłączenie się od segmentu pamięci

shm_unlink(char *name)

name - nazwa segmentu pamięci.

Każde wywołanie tej funkcji zmniejsza licznik udostępnień segmentu. Gdy osiągnie on wartość 0 czyli segment nie jest używany już przez żaden proces, segment jest kasowany.

Pamięć dzielona POSIX – przykład

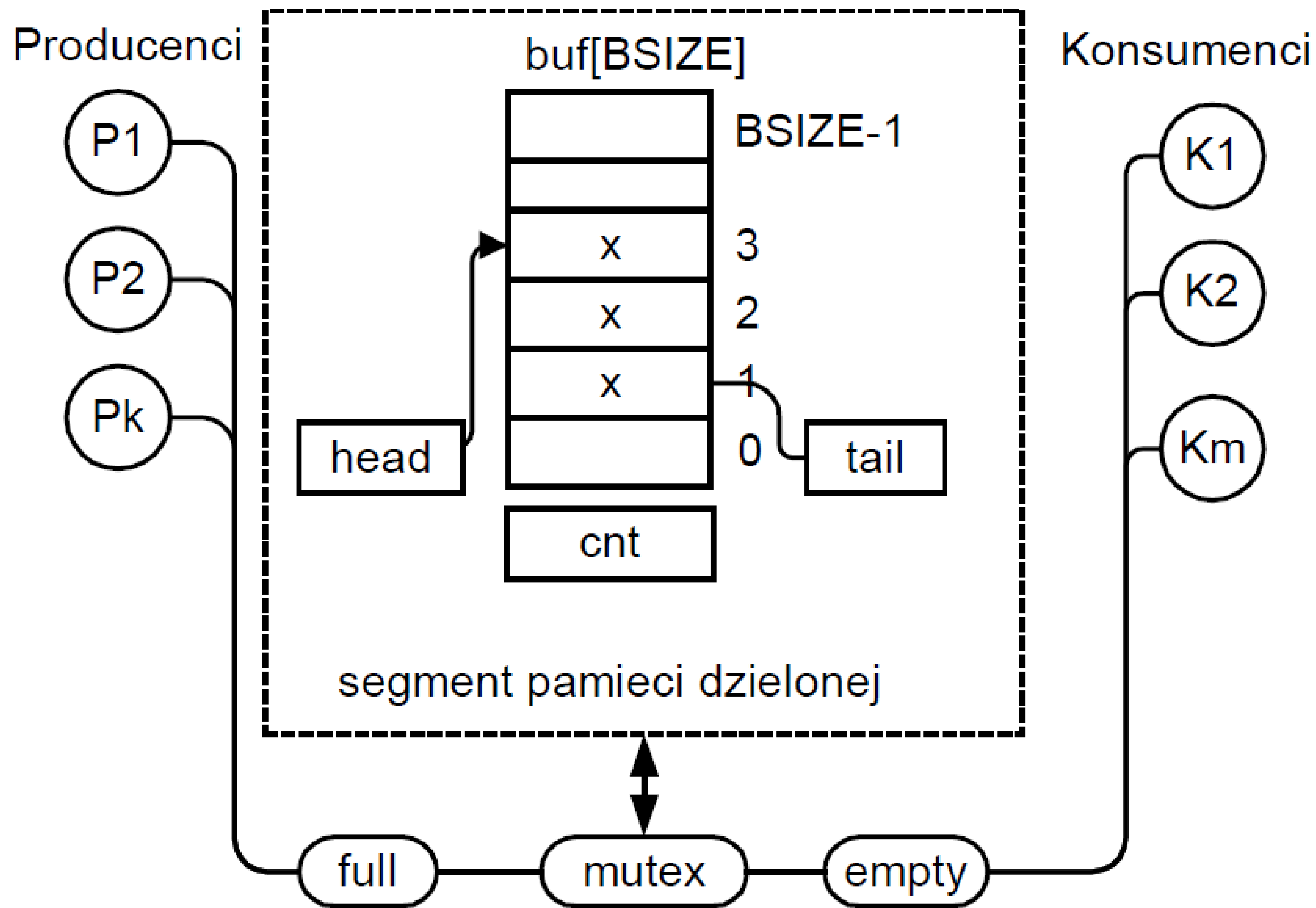
```
/* posix-shm.c : gcc -o posix posix.c -lrt */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>        // POSIX
#include <sys/file.h>      // Pulls in open(2) and friends.
#include <sys/mman.h> // Pulls in mmap(2) and friends.
#include <sys/wait.h>

void error_out(const char *msg)
{ perror(msg);
 exit(EXIT_FAILURE);}

int main(int argc, char *argv[])
{ int r;
  const char *memname = "/mymem";
  const size_t region_size = sysconf(_SC_PAGE_SIZE);
  int fd = shm_open(memname, O_CREAT | O_TRUNC |
O_RDWR, 0666);
  if (fd == -1) error_out("shm_open");
  r = ftruncate(fd, region_size);
  if (r != 0) error_out("ftruncate");
  void *ptr =
  mmap(0, region_size, PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
 if (ptr == MAP_FAILED) error_out("mmap");
 close(fd);
```

```
pid_t pid = fork();
  if (pid == 0) {
  u_long *d = (u_long *) ptr;
  *d = 0xdeadbeef;
  exit(0);
 }
 else {
 int status;
 waitpid(pid, &status, 0);
 printf("child wrote %#lx\n", *(u_long *) ptr);
 }
 r = munmap(ptr, region_size);
 if (r != 0) error_out("munmap");
 r = shm_unlink(memname);
 if (r != 0) error_out("shm_unlink");
 return 0;
 }
```

Wynik przykładowej sesji:

```
$ gcc -o sysv-shm sysv-shm.c
$ ./sysv-shm
child wrote 0xdeadbeef
```

Rozwiązanie problemu producent-konsument (pamięć dzielona)



Rozwiązanie – pamięć dzielona (1)

```
##include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <pthread.h>
#include <semaphore.h>
#include <assert.h>
#include <sys/stat.h>
#include <sys/file.h>

#define BSIZE 4 // Rozmiar bufora
#define LSIZE 80 // Dlugosc linii
typedef struct { // Obszar wspólny
char buf[BSIZE][LSIZE];
int head;
int tail;
int cnt;
} bufor_t;
```

```
int main()
{   sem_t *mutex;
   sem_t *empty;
   sem_t *full;
   int shmid;
   void *shared_memory = (void *)0;
   bufor_t *shared_stuff;
   int i=0;
   shmid = shmget((key_t)1234,
                  sizeof(bufor_t), 0666 | IPC_CREAT);
   if (shmid == -1) {
      fprintf(stderr, "shmget failed\n");
      exit(EXIT_FAILURE);
   }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);
   }
   printf("Memory attached at %X\n",
                                   (int)shared_memory);
   shared_stuff = (bufor_t *)shared_memory;
   shared_stuff->head=0;
   shared_stuff->tail=0;
   shared_stuff->cnt=0;
```

Rozwiązanie – pamięć dzielona (2)

```
mutex = sem_open("mutex",O_CREAT,S_IRWXU,1);
empty = sem_open("empty",O_CREAT,S_IRWXU,BSIZE);
full =  sem_open("full",O_CREAT,S_IRWXU,0);
//Producenci:
printf("Tworze producentow...\n");
{
  if(fork()==0)
 { printf("Producent %d uruchominy\n",i);
   sleep(1);
   while(1)
   {static int j=0;
    sem_wait(empty);
    sem_wait(mutex);
    printf("Prod: %d - cnt: %d head: %d tail: %d\n",
    i, shared_stuff-> cnt,shared_stuff->head,
    shared_stuff->tail);
    printf(shared_stuff->buf[shared_stuff->head],
    "Komunikat %d producent %d",j++,i);
    shared_stuff-> cnt ++;
    shared_stuff->head = (shared_stuff->head +1) %
                                                BSIZE;
    sem_post(mutex);
    sem_post(full);
     sleep(1);
    }
    exit(0); } }
```

```
printf("Tworze konsumentów...\n");
  {   if(fork()==0)
     {  printf("Konsument %d uruchominy\n",i);
        sleep(2);
        while(1)
        { sem_wait(full);
          sem_wait(mutex);
          printf("Konsument %d - cnt: %d odebrano %s\n",
           i, shared_stuff->cnt,
           shared_stuff->buf[shared_stuff->tail]);
           shared_stuff-> cnt --;
           shared_stuff->tail = (shared_stuff->tail +1) %
                                                    BSIZE;
           sem_post(mutex);
           sem_post(empty);
           sleep(1);
        } exit(0);
     }
  }
  {   int stat_val;
     pid_t child_pid;
     for(i=0;i<2;i++)
     child_pid = wait(&stat_val);
  }
  return 0;
}
```

Dyskusja

- Medium komunikacyjnym jest kolejka zrealizowana we współdzielonym obszarze pamięci
- Zalety:
  - Mniejsze ograniczenia na rozmiar przesyłanych danych
- Wady:
  - Konieczność „samodzielnego” zarządzania prawidłowym dostępem do danych
  - Bardziej złożony kod
- Takie rozwiązanie zmniejsza ograniczenie ze względu na pojemność bufora do wymiany danych. Warto stosować takie rozwiązanie, gdy pomiędzy producentem, a konsumentem przesyłane są większe porcje danych/komunikaty

Kolejki komunikatów

- Umożliwiają przesyłanie danych pomiędzy procesami w sposób podobny do potoków.
- Pozwalają na przesyłanie komunikatów z priorytetami
- Pozwalają na przesyłanie komunikatów o zmiennej długości, pod warunkiem ich prawidłowego odbioru (IPC-tak, POSIX-oczekują komunikatów o stałym rozmiarze)
- Na długość kolejki i rozmiar komunikatu są nałożone systemowe ograniczenia.

Funkcje IPC obsługujące kolejki komunikatów

```
#include <sys/msg.h>

// Sterowanie bezpośrednie pracą kolejki:
int msgctl(int msqid, int cmd, struct msqid_ds *buf);

// Utworzenie kolejki i uzyskanie do niej dostępu:
int msgget(key_t key, int msgflg);

// Pobranie komunikatu z kolejki:
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

// Dodanie komunikatu do kolejki:
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

msgget()

int msgget(key_t key, int msgflg);

- key – unikalny klucz do dostępu do kolejki
- msgflg – znaczniki zezwoleń
- Funkcja zwraca identyfikator kolejki lub -1 w przypadku błędu

msgsnd()

int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

- msqid – identyfikator kolejki komunikatów
- msg_ptr – wskaźnik do wysyłanego komunikatu
- msg_sz – rozmiar komunikatu bez pola typ komunikatu (patrz niżej)
- msgflg – reakcja na przepełnienie kolejki
  - IPC_NOWAIT – funkcja powróci natychmiast bez wysyłania komunikatu, zwracając wartość -1
  - IPC_WAIT – funkcja zawiesi proces w oczekiwaniu na wolne miejsce w kolejce
- Funkcja zwraca 0 w przypadku pomyślnego zapisu w kolejce lub -1 w przypadku błędu.

UWAGA: Każdy komunikat musi się zaczynać od liczby typu long int, wskazującej typ komunikatu; liczbę warto ustawić na ustaloną wartość.

Struktura komunikatu ma zwykle postać:

```
struct my_message {
long int message_type;
/* The data you wish to transfer */};
```

msgrcv()

int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

- msqid – identyfikator kolejki komunikatów
- msg_ptr – wskaźnik do odbieranego komunikatu
- msg_sz – rozmiar komunikatu bez pola typ komunikatu
- msgtype – typ komunikatu:
  - 0 – z kolejki pobierany jest pierwszy komunikat
  - >0 – pobierany jest pierwszy dostępny komunikat tego samego typu
  - <0 – pobierany jest pierwszy dostępny komunikat, którego typ ma wartość najniższą spośród wartości takich samych lub mniejszych niż absolutna wartość msgtype
- Funkcja zwraca liczbę bajtów umieszczonych w buforze odbiorczym, komunikat jest kopiowany do bufora wskazywanego przez msg_ptr, a dane są usuwane z kolejki komunikatów. W przypadku błędu funkcja zwraca -1.

msgctl()

int msgctl(int msqid, int command, struct msqid_ds *buf);

- Struktura msqid_ds ma przynajmniej następujące parametry:

```
struct msqid_ds {
uid_t msg_perm.uid;
uid_t msg_perm.gid
mode_t msg_perm.mode;
}
```

- msqid – identyfikator kolejki
- command – czynność, jaką należy podjąć:
  - IPC_STAT – ustawia dane w strukturze msqid_ds aby odzwierciedlały wartości związane z kolejką komunikatów
  - IPC_SET – ustawia wartości związane z kolejką komunikatów na dane, określone w strukturze msqid_ds, jeśli proces ma odpowiednie zezwolenia
  - IPC_RMID – usuwa kolejkę komunikatów

Kolejka komunikatów – przykład – program odbiorczy

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msg_st {
   long int my_msg_type;
   char some_text[BUFSIZ];
};
int main()
{  int running = 1;
   int msgid;
   struct my_msg_st some_data;
   long int msg_to_receive = 0;
   msgid = msgget((key_t)1234, 0666 |
                                           IPC_CREAT);
   if (msgid == -1) {
      fprintf(stderr, "msgget failed with error:
                %d\n", errno);  exit(EXIT_FAILURE);
   }
```

```
   while(running) {
      if (msgrcv(msgid, (void *)&some_data,
                BUFSIZ, msg_to_receive, 0) == -1) {
         fprintf(stderr, "msgrcv failed with error:
                                        %d\n", errno);
         exit(EXIT_FAILURE);
      }
 printf("You wrote: %s", some_data.some_text);
 if (strncmp(some_data.some_text, "end", 3) == 0)
{          running = 0;
}
   } //while

   if (msgctl(msgid, IPC_RMID, 0) == -1) {
      fprintf(stderr, "msgctl(IPC_RMID) failed\n");
      exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}
```

Kolejka komunikatów – przykład – program nadawczy

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_TEXT 512
struct my_msg_st {
   long int my_msg_type;
   char some_text[MAX_TEXT];
};
int main()
{   int running = 1;
   struct my_msg_st some_data;
   int msgid;
   char buffer[BUFSIZ];
   msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
   if (msgid == -1) {
      fprintf(stderr, "msgget failed with error:
%d\n", errno);
      exit(EXIT_FAILURE);
   }
```

```
   while(running) {
      printf("Enter some text: ");
      fgets(buffer, BUFSIZ, stdin);
      some_data.my_msg_type = 1;
      strcpy(some_data.some_text, buffer);

      if (msgsnd(msgid, (void *)&some_data,
MAX_TEXT, 0) == -1) {
         fprintf(stderr, "msgsnd failed\n");
         exit(EXIT_FAILURE);
      }
      if (strncmp(buffer, "end", 3) == 0) {
         running = 0;
      }
   }

   exit(EXIT_SUCCESS);
}
```

Uruchomienie i rezultat działania programów:

```
$ ./msg2
Enter some text: hello
Enter some text: How are you today?
Enter some text: end
$ ./msg1
You wrote: hello
You wrote: How are you today?
You wrote: end
$
```

Kolejki komunikatów- uwagi

- Kolejki komunikatów przypominają w swojej zasadzie działania potoki, choć nie wymagają pseudo-operacji plikowych do ich otwarcia i zamykania
- Pozwalają na przesyłanie (ograniczonych) bloków danych z jednego procesu do drugiego.
- Każdy blok posiada pewien typ, który może zostać zastosowany do „rozpoznawania” komunikatów oraz do ich priorytetowania
- Istnieje systemowe ograniczenie w systemie LINUX na rozmiar pojedynczej wiadomości (MSGMAX = 4096) i rozmiar całej kolejki (MSGMNB = 16384)
- Domyślnie kolejki blokują procesy w oczekiwaniu na wartość do odebrania lub w oczekiwaniu na wolne miejsce w kolejce. Istnieje możliwość wywołania funkcji nadającej i odbierającej z flagą IPC_NOWAIT. Wtedy procesy nie są blokowane, tylko funkcje zwracają odpowiednie wartości błędu.

Priorytetowanie komunikatów w kolejce IPC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/msg.h>
#include <sys/ipc.h>
struct message { long int mtype;
                 char mtext[128];};
int send_msg(int qid, int mtype, const char text[])
{ struct message msg = {.mtype = mtype };
  strncpy(msg.mtext, text, sizeof(msg.mtext));
  int r = msgsnd(qid, &msg, sizeof(msg), 0);
  if (r == -1) { perror("msgsnd"); }
  return r; }
int recv_msg(int qid, int mtype, struct message *msg)
{ int r = msgrcv(qid, msg, sizeof(struct message),
mtype, 0);
switch (r) { case sizeof(struct message): /* okay */
                                   break;
          case -1: perror("msgrcv");  break;
          default: printf("only received %d bytes\n",
r);
        }
return r;}
```

```
void producer(int mqid)
{ send_msg(mqid, 1, "type 1 - first");
  send_msg(mqid, 2, "type 2 - second");
  send_msg(mqid, 1, "type 1 - third");

void consumer(int qid)
{ struct message msg;
int r; int i;
for (i = 0; i < 3; i++) {
 r = msgrcv(qid, &msg, sizeof(struct message), -2, 0);
 printf("'%s'\n", msg.mtext); }
}
int main(int argc, char *argv[])
{
int mqid;
mqid = msgget(IPC_PRIVATE, S_IREAD | S_IWRITE);
if (mqid == -1) { perror("msgget"); exit(1); }
pid_t pid = fork();
if (pid == 0) { consumer(mqid);
                exit(0); }
else {   int status; producer(mqid);
         wait(&status); }
int r = msgctl(mqid, IPC_RMID, 0);
if (r)
perror("msgctl");
return 0;}
```

Uruchomienie i rezultat działania programów:

```
$ ./sysv-msgq-example
'type 1 - first'
'type 1 - third'
'type 2 – second'
```

Funkcje POSIX obsługujące kolejki komunikatów

```
#include <mqueue.h>

// Tworzenie kolejki:
mqd_t mq_open(const char *name, int oflag, ...);

// Zamknięcie dostępu do kolejki i zwolnienie deskryptora:
int mq_close(mqd_t mqdes);

// Trwałe usunięcie kolejki:
int mq_unlink(const char *name);

// Wysłanie komunikatu do kolejki:
int mq_send(mqd_t mqdes, char *ptr, size_t len, unsigned prio);

// Odebranie komunikatu od kolejki:
ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned *prio);
```

Priorytetowanie komunikatów w kolejce POSIX

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>
struct message { char mtext[128]; };

 int send_msg(int qid, int pri, const char text[])
 {
 int r = mq_send(qid, text, strlen(text) + 1, pri);
 if (r == -1) { perror("mq_send");}
 return r; }

 void producer(mqd_t qid)
 { send_msg(qid, 1, "This is my first message.");
   send_msg(qid, 1, "This is my second message.");
   send_msg(qid, 3, "No more messages.");}

 void consumer(mqd_t qid)
 { struct mq_attr mattr;
  do { u_int pri; struct message msg; ssize_t len;
 len = mq_receive(qid, (char *) &msg,
                  sizeof(msg), &pri);
 if (len == -1) { perror("mq_receive"); break;}
 printf("got pri %d '%s' len=%d\n", pri, msg.mtext,
len);
int r = mq_getattr(qid, &mattr);
if (r == -1) { perror("mq_getattr"); break; }
} while (mattr.mq_curmsgs); }

int main(int argc, char *argv[])
{  struct mq_attr mattr = {
   .mq_maxmsg = 10,
   .mq_msgsize = sizeof(struct message) };

mqd_t mqid = mq_open("/myq", O_CREAT | O_RDWR,
                           S_IREAD | S_IWRITE, &mattr);
if (mqid == (mqd_t) -1) { perror("mq_open"); exit(1);}

pid_t pid = fork();
if (pid == 0) {   producer(mqid);
                  mq_close(mqid); exit(0); }
else {         int status;  wait(&status);
               consumer(mqid);
               mq_close(mqid);}
mq_unlink("/myq");
return 0;}
```

Wynik przykładowej sesji:

```
$ ./posix-msgq-ex
got pri 3 'No more messages.' len=18
got pri 1 'This is my first message.' len=26
got pri 1 'This is my second message.' len=27
```