IPC - cz. 1
Kolejki komunikatów i pamięć wspólna

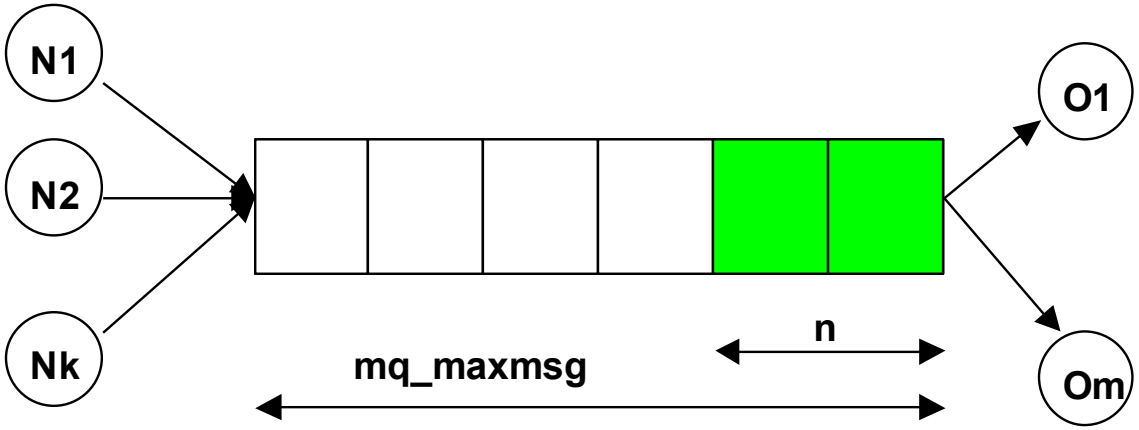
Ostatnia modyfikacja: 03.03.2020

POSIX IPC

	Kolejki komunikatów	Pamięć wspólna	Semafory
Plik nagłówkowy	**<mqueue.h>**	**<sys/mman.h>**	**<semaphore.h>**
Tworzenie/ otwieranie/usuwanie	**mq_open(), mq_close(), mq_unlink()**	**shm_open(), shm_unlink()**	**sem_open() sem_close(), sem_unlink(), sem_init(), sem_destroy()**
Operacje sterujące	**mq_getattr(), mq_setattr()**	**ftruncate(), fstat()**	
Operacje komunikacji	**mq_send() mq_receive(), mq_notify()**	**mmap() munmap()**	**sem_wait(), sem_trywait(), sem_post(), sem_getvalue()**

- Trwałość obiektów POSIX IPC to tzw. **trwałość jądra** za wyjątkiem **semafora w pamięci**, który ma **trwałość procesu** (*process persistence*) – obiekt istnieje tak długo, aż ostatni proces z niego korzystający dokona zamknięcia obiektu.

POSIX – kolejki komunikatów



Schemat komunikacji procesów za pomocą kolejki komunikatów

Podstawowe cechy kolejek komunikatów:

- Istnieje możliwość wskazania tej samej kolejki przez niezwiązane procesy. Kolejka komunikatów może być widoczna w systemie plików (Linux), ale nie musi
- Przekazywanie komunikatów (o długości 0 do **mq_msgsize**) jest niezawodne. Kolejka ma trwałość w ramach systemu, tzn. istnieje do restartu systemu lub do jawnego usunięcia.
- W procesie kolejka jest identyfikowana przez deskryptor kolejki, zmienną typu **mqd_t**.. Deskryptor kolejki może być implementowany jako deskryptor pliku.
- Kolejka ma skończoną pojemność (**mq_maxmsg** komunikatów)
- Dostęp do funkcji realizujących kolejkę wymaga użycia biblioteki **rt**.

POSIX – kolejki komunikatów

- Komunikaty mają długość maksymalną (**mq_msgsize**) określoną w czasie tworzenia kolejki. Operacje odczytu muszą być zawsze przygotowane na odbiór komunikatu o maksymalnej długości.
- Kolejka ma określoną, w czasie tworzenia, długość (**mq_maxmsg**). Gdy zostanie ona przekroczona - proces piszący do kolejki będzie zablokowany (przy pracy w domyślnym trybie: blokującym) - aż będzie dostatecznie dużo miejsca wolnego w kolejce, bądź do przerwania sygnałem.
- Komunikaty odczytywane z kolejki zachowują strukturę, chociaż w kolejce mogą znajdować się komunikaty o różnej długości.
- Komunikatom można nadać priorytet (liczba całkowita bez znaku, mniejsza od stałej **MQ_PRIO_MAX**>=32, do pobrania przez **sysconf**()). Komunikaty o najwyższym priorytecie są umieszczane na początku kolejki (w porządku FIFO).
- Operacja odczytu z pustej kolejki blokuje odbiorcę (wątek), jeśli dostęp jest w trybie z blokowaniem.
- Implementacja definiuje **MQ_OPEN_MAX** (>=8) – maksymalną liczbę kolejek, które w danej chwili mogą być otwarte przez jeden proces.
- Istnieje interfejs plikowy do parametrów kolejki (**man namespaces(7)**): patrz **/proc/sys/fs/mqueue**.

Podstawowe typy i plik nagłówkowy

- Podstawowe typy danych i prototypy funkcji są w pliku nagłówkowym:
  <mqueue.h>

- **Atrybuty kolejki** przekazywane są w strukturze:

```
struct mq_attr {
  long mq_flags;   /* 0 albo O_NONBLOCK */
  long mq_maxmsg; /* Maks. liczba wiadomości w kolejce*/
  long mq_msgsize; /* Maks. długość wiadomości (B) */
  long mq_curmsgs; /* Aktualna liczba komunikatów
                      w kolejce */
};
```

Tworzenie kolejki/otwieranie dostępu

```
mqd_t   mq_open(const char *name, int oflag
        /* , mode_t mode, struct mq_attr *attr */);
```

Funkcja **mq_open** zwraca identyfikator kolejki, albo **(mqd_t)(–1)** , ustawiając kod błędu w zmiennej globalnej **errno**.

Parametry wywołania:

name – łańcuch identyfikujący kolejkę komunikatów.

oflag - tryb tworzenia kolejki, jak dla plików (**O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NONBLOCK**)

mode - prawa dostępu do tworzonej kolejki (**r** i **w** - jak dla plików)

attr - wsk. do struktury atrybutów kolejki (pola: **mq_maxmsg**, **mq_msgsize**)

Uwagi:

- wykonanie funkcji może zostać przerwane (kod wyjścia –1, **errno==EINTR**), wskutek obsługi sygnału przez proces wywołujący **mq_open**()
- W SO Linux kolejki tworzone są w wirtualnym systemie plików, który można domontować np. do katalogu **/dev/mqueue**. Informacje o kolejkach są dostępne w poddrzewie systemu plików: **/proc/sys/fs/mqueue/**

POSIX MQ – przestrzeń nazw i identyfikatorów

- Parametr **name** wskazuje na napis (*C-string)* będący nazwą kolejki komunikatów POSIX.
  - POSIX nie wymaga, by nazwa była widoczna w systemie plików czy była dostępna dla funkcji systemowych korzystających z nazw ścieżkowych.
  - Parametr **name** musi spełniać wymagania nazwy ścieżkowej (*pathname*).
    - Jeśli ***name*** rozpoczyna znak ‚/', to każdy proces wywołujący ***mq_open( )*** z taką nazwą wskazuje na tą samą kolejkę komunikatów – póki nie zostanie usunięta z systemu.
    - Jeśli **name** nie rozpoczyna znak ‚***/***' – konsekwencje zależą od implementacji.
    - Konsekwencje wielokrotnego wystąpienia w nazwie znaku ‚***/***' zależą od implementacji. W SO Linux ***name*** rozpoczyna znak ‚/'; nie może być więcej takich znaków w ***name***.
- Deskryptor kolejki komunikatów POSIX może być implementowany za pomocą deskryptora plików. Wówczas proces może jednocześnie mieć otwartych {**OPEN_MAX**} plików i kolejek.
- Dokumentacja kolejek w systemie Linux : **mq_overview(7)**

Zamykanie dostępu i kasowanie kolejki

- Gdy proces przestaje korzystać z kolejki powinien ją zamknąć za pomocą

```
int mq_close(mqd_t mq);
```

- Kolejkę kasuje się za pomocą:

```
int mq_unlink(char *name);
```

Funkcja powoduje natychmiastowe usunięcie nazwy wskazanej kolejki z systemu; sama kolejka jest usuwana z systemu wtedy, gdy wszystkie procesy, które otwarły dostęp do tej kolejki zamkną deskryptory kolejki (za pomocą **mq_close**) .

Wysyłanie komunikatów

```
int_t mq_send( mqd_t mqdes, const char *msg_ptr,
               size_t msg_len, unsigned msg_prio );
```

wstawianie komunikatu (msg_ptr[0],…msg_ptr[msg_len-1]) do kolejki **mqdes**

```
int mq_timedsend( mqd_t mqdes, const char *msg_ptr,
                  size_t msg_len, unsigned msg_prio,
                  const struct timespec *abs_timeout );
```

wstawianie komunikatu do kolejki (z ograniczonym czekaniem)

Parametry wywołania:

mqdes - identyfikator kolejki komunikatów
msg_ptr - adres bufora wysyłanego komunikatu
msg_len - długość wysyłanego komunikatu
msg_prio - priorytet komunikatu (od 0 do MQ_PRIORITY_MAX)
abs_timeout - odległość czasowa (od północy 1.I.1970r.) końca okresu czekania na dostęp do kolejki

```
struct timespec {
        time_t tv_sec;      /* sekundy */
        long   tv_nsec;     /* nanosekundy */
    };
```

Funkcje wysyłające zwracają 0 przy pomyślnym wstawieniu komunikatu (**mq_timedsend**() - w zadanym przedziale czasu), albo **–1** przy niepowodzeniu (**errno** zawiera kod błędu).

Jeśli kilka wątków blokuje na **mq_send**()/**mq_timedsend**() z powodu pełnej kolejki, to przy zwolnieniu miejsca odblokowywany jest wątek **o najwyższym priorytecie, czekający najdłużej**.

Uwaga: obydwie funkcje mogą zostać przerwane po obsłużeniu przez proces sygnału.

Odbiór komunikatów

int mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio_p);

Funkcja pobiera z kolejki związanej z deskryptorem **mqdes** do bufora o długości **msg_len**, wskazywanego przez **msg_ptr** najstarszą wiadomość o największym priorytecie. Jeśli **msg_prio_p!=NULL**, to wartość ***msg_prio_p** staje się równa priorytetowi pobranej wiadomości.

Przy pomyślnym wykonaniu funkcja zwraca długość pobranej wiadomości. W przypadku niepowodzenia funkcja zwraca –1 (ustawiając **errno**), a kolejka komunikatów nie ulega zmianie.

int mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio_p, const struct timespec *abs_timeout);

Funkcja pobiera wiadomość tak jak **mq_receive**(), oczekując najdłużej do momentu określonego w strukturze wskazywanej przez **abs_timeout**. (czas odmierzany przez zegar CLOCK_REALTIME).

Uwagi:

- Obydwie funkcje mogą zostać przerwane po obsłużeniu przez proces sygnału.
- Jeśli proces ustawi powiadomienie asynchroniczne i zablokuje się na **mq_receive()**, to nowa wiadomość odblokuje **mq_receive()** (ma pierwszeństwo).

Testowanie statusu kolejki

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

Pobiera do struktury wskazywanej przez **attr** atrybuty kolejki skojarzonej z deskryptorem **mqdes**

int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
struct mq_attr *oldattr);

Ustawia atrybuty kolejki skojarzonej z deskryptorem **mqdes**.

newattr wskazuje na strukturę z nowymi atrybutami, a **oldattr** (jeśli nie **NULL**) – wskazuje na strukturę, w której funkcja umieszcza poprzednie atrybuty.

Uwaga: funkcja **mq_setattr** może zmienić jedynie atrybut **mq_flags** (**0** albo **O_NONBLOCK**).

Fragmenty przykładu

Nadawca

```
char buf[25], *mq_name=...;
mqd_t mqdes;
struct mq_attr attr;
attr.mq_maxmsg=1;//one msg only
attr.mq_msgsize=sizeof(buf);
mqdes=mq_open(mq_name,
          O_RDWR | O_CREAT, FILE_MODE,
           &attr);
if(mqdes==(mqd_t)-1){/* wyjście z błędem*/ }
while(fgets(buf,sizeof(buf),stdin)){
   char *ptr;
   int pri=msgnr%3;
   ptr=strchr(buf,'\n');
   if(ptr) *ptr='\0';
   else buf[sizeof(buf)-1]='\0';
   if(mq_send(mqdes,buf,strlen(buf)+1,pri)
   <0) break;
   msgnr++;
}
mq_close(mqdes);
```

Odbiorca

```
int main(int argc, char *argv[]){
char buf[25], *mq_name=...;
unsigned int pri, timeout=...;
mqd_t mqdes;
struct mq_attr attr;
If((mqdes=mq_open(mq_name,
   O_RDONLY,NULL))==(mqd_t)-1)
        {/* błąd */}
if(mq_getattr(mqdes,&attr)<0) {/* błąd */}
if(attr.mq_msgsize>sizeof(buf)){ exit(1);  }

while(1) {
      if(mq_receive(mqdes,buf,sizeof(buf),
           &pri)<0) break;
      buf[sizeof(buf)-1]='\0';
      puts(buf);
}
mq_close(mqdes);
```

Asynchroniczne powiadomienie

int mq_notify(mqd_mqdes, struct sigevent *notification);

Funkcja umożliwia dla każdej kolejki POSIX rejestrację **pojedynczego powiadomienia** o asynchronicznym zdarzeniu, polegającym na pojawieniu się komunikatu w pustej kolejce. Dla danej kolejki może być powiadamiany (jednokrotnie) tylko jeden proces. W reakcji na zdarzenie można zamówić:

- **doręczenie wskazanego sygnału** do procesu odbiorcy (SIGEV_SIGNAL)
- **uruchomienie wątku** ze wskazaną funkcją roboczą i argumentem wywołania (SIGEV_THREAD)
- brak powiadamiania (SIGEV_NONE)

Jeżeli notification==NULL →proces **odwołuje powiadomienie** (jeśli je wcześniej zamówił)

```
struct sigevent {
  int    sigev_notify;  /* Metoda: SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD */
  int    sigev_signo;       /* Sygnał powiadomienia (dla SIGEV_SIGNAL) */
  union sigval sigev_value; /* Dane przekazane z powiadomieniem */
  void (*sigev_notify_function) (union sigval);   /* Funkcja robocza wątku (dla metody
                          SIGEV_THREAD) */
  void  *sigev_notify_attributes;  /* Atrybuty funkcji roboczej wątku powiadamiania */
};
union sigval {              /* Dane przekazane z powiadomieniem */
  int     sival_int;
  void   *sival_ptr;
 };
```

Uwaga: po doręczenia powiadomienia do procesu rejestracja jest **usuwana**

Przykład wykorzystania powiadomienia – doręczenie sygnału SIGUSR1

```
void handler(int sig, siginfo_t *s, void *p){/* Uwaga: obsługa sygnałów RT */
  signr=sig;  pid_nr=s->si_pid;          return;
}
static struct sigaction siga;
static struct sigevent not;
...
mqdes =mq_open(mq_name,O_RDONLY,NULL);
if (mqdes==(mqd_t)-1){/* obsługa błędu */}
siga.sa_flags=SA_SIGINFO; siga.sa_sigaction=handler;

if(sigaction(SIGUSR1,&siga,NULL)<0){/* obsługa błędu */}
not.sigev_notify=SIGEV_SIGNAL; /* Powiadomienie sygnałem */
not.sigev_signo=SIGUSR1; /* Wybór numeru sygnału powiadomienia */
if(mq_notify(mqdes, &not)<0){/* błąd */ }/* Rejestracja powiadomienia */
```

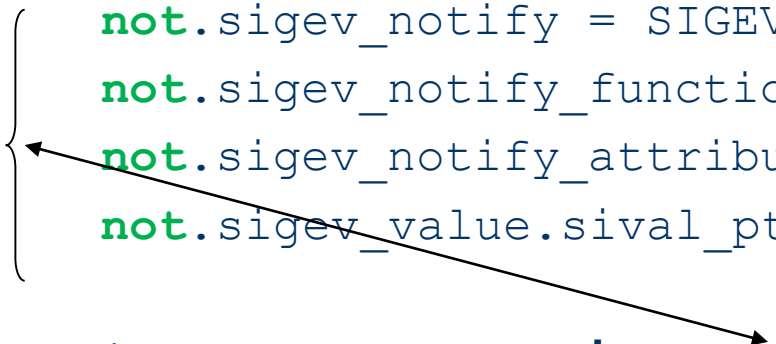
Przykład powiadomienia przez rozpoczęcie nowego wątku

```
int main(int argc, char *argv[]){
    mqd_t mqdes;
    struct sigevent not;
    if(argc!= 2){ /* błąd wywołania, brak nazwy kolejki */}

    mqdes = mq_open(argv[1], O_RDONLY);
    if (mqdes == (mqd_t) -1) {/* błąd */}

    not.sigev_notify = SIGEV_THREAD;/* Powiadamianie wątkiem */
    not.sigev_notify_function = tfunc; /* F. robocza wątku */
    not.sigev_notify_attributes = NULL;
    not.sigev_value.sival_ptr = &mqdes; /* Arg. f. roboczej*/

    if (mq_notify(mqdes, &not) == -1) {/* błąd */}
    pause(); /* Proces będzie zakończony w f. roboczej wątku */
    return EXIT_SUCCESS;
}
```

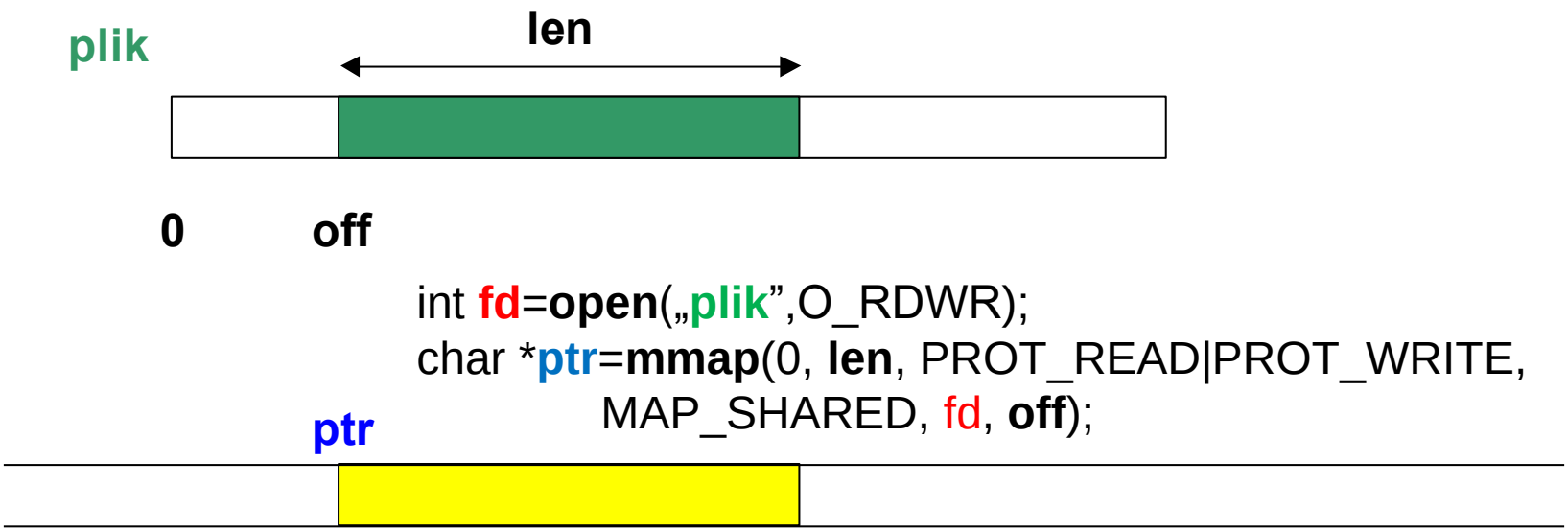


C.d. przykładu

```
static void  tfunc(union sigval sv){/* Funkcja robocza wątku
  powiadamiania */
       struct mq_attr attr;
       ssize_t nr;
       void *buf;
       mqd_t mqdes = *((mqd_t *) sv.sival_ptr); /* Deskryptor
                        kolejki */
/* Pobranie maks. długości wiadomości */
        if (mq_getattr(mqdes, &attr) == -1) {/* błąd */ }
/* Alokacja bufora wiadomości */
        buf = malloc(attr.mq_msgsize);
        if (buf == NULL) {/* błąd */}
/* Wczytanie wiadomości (pierwszej w kolejce) */
        nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
        if (nr == -1) {/* błąd */}
        printf(„Wczytano %ld B z kolejki\n”,(long) nr);
        free(buf);/* Zwolnienie bufora */
        exit(EXIT_SUCCESS);          /* Zakończenie procesu */
}
```

Odwzorowanie plików w pamięci

Idea odwzorowania (części) pliku w przestrzeń adresową procesu



ptr[0] jest odniesieniem do bajtu pliku numer **off**

ptr[len-1] jest odniesieniem do ostatniego odwzorowanego bajtu pliku o numerze **off+len-1**

Odwzorowanie plików w pamięci – c.d.

void * **mmap** (void ***addr**, size_t **len**, int **protect**, int **flags**, int **fd**, off_t **off**) – funkcja tworzy odwzorowanie **len** bajtów pliku związanego z deskryptorem **fd**, począwszy od bajtu nr **off**, z obszarem pamięci, do którego wskaźnik zwraca **mmap.** Parametr **addr,** podpowiadający **mmap** lokalizację użytego fragmentu przestrzeni adresowej, zwykle przybiera wartość 0 (tzn. wybiera **mmap**)

Znaczenie parametru **protect**:

- **MAP_PRIVATE** – modyfikacja w pamięci nie jest zapisywana do pliku
- **MAP_SHARED** – zapis do odwzorowanego obszaru w pamięci spowoduje zapis do odwzorowanego pliku

protect zawiera też bity praw dostępu: **PROT_READ, PROT_WRITE, PROT_EXEC**

int **msync** (void ***addr**, size_t **len**, int **flags**) – wymusza zapis zawartości obszaru pamięci (**len** bajtów, począwszy od addresu **addr)** odwzorowanego w trybie **MAP_SHARED** do odwzorowanego pliku. Znaczenie parametru **flags**:

- **MS_SYNC** – funkcja czeka, aż dane zostaną zapisane
- **MS_ASYNC** – funkcja inicjuje zapis, ale nie czeka na zakończenie

int **munmap** (void *addr, size_t len) –usuwa wszystkie odwzorowania adresów pamięci, od **addr** do **addre+len-1**

Pamięć wspólna POSIX IPC

```
int shm_open(const char *name, int oflag,
      mode_t mode);
```

Tworzy nowy segment pamięci i/lub ustanawia połączenie między segmentem a deskryptorem pliku; zwraca deskryptor reprezentujący otwarty segment (-1 przy niepowodzeniu). Wymagania na nazwę (**name**) – jak dla kolejki komunikatów.

Flagi **oflag**:

- **0** - ustanawia połączenie
- **O_CREAT** - tworzy nowy segment i ustanawia połączenie
- **O_EXCL| O_CREAT** - tworzy nowy segment i ustanawia połączenie lub zwraca błąd, jeśli segment istnieje

Uwagi:

- Tworzenie nowego segmentu pamięci wymaga podania bitów ochrony pliku: **mode**
- W SO Linux obiekty pamięci wspólnej POSIX tworzone są w wirtualnym systemie plików, który można domontować np. do katalogu **/dev/shm**
- Dokumentacja segmentu pamięci wspólnej: **shm_overview(7)**

Pamięć wspólna POSIX IPC

Po otwarciu segmentu należy:

- określić rozmiar segmentu (tak jak rozmiar pliku) za pomocą funkcji:

```
int ftruncate(int fildes, off_t length);
```

Uwaga: bezpośrednio po utworzeniu segment ma rozmiar 0

- odwzorować segment na przestrzeń adresową procesu/wątku (tak jak zwykły plik) za pomocą funkcji **mmap()**z flagą **MAP_SHARED**
- korzystać z odwzorowania tak, jak w przypadku zwykłego adresu
- po zakończeniu usunąć odwzorowanie za pomocą funkcji **munmap()**
- funkcja

```
int shm_unlink(const char *name);
```

Usuwa wskazaną nazwę (**name**) segmentu pamięci dzielonej

Pamięć wspólna POSIX IPC – c.d.

Inne funkcje systemowe, które dotyczą pamięci wspólnej POSIX IPC:

- **close()** – umożliwia zamknięcie deskryptora utworzonego przez shm_open(), gdy nie jest już potrzebny.
- **fstat()** – wypełnia strukturę typu stat informacjami o pamięci wspólnej, w tym:
  - **st_size** - rozmiar,
  - **st_mode** – prawa dostępu
  - **st_uid, st_gid** –UID i GID właściciela
- **fchown()** – umożliwia zmianę właściciela
- **fchmod()** – umożliwia zmianę praw dostępu

Przykład użycia pamięci wspólnej

```
#define SHM_NAME " /shm_tool"      // segment name
#define SHM_LEN 100                          // segment size
...
int   shm_fd;  /* shm id */
char  *segptr;   /* mapped adres of the start of shm segment */
if((shm_fd = shm_open(SHM_NAME,O_CREAT|O_EXCL|O_RDWR,0666)) == -1){
        if(errno!=EEXIST){/* error handling */ ...}
        else {
            printf("Shared memory segment exists\n");
            if((shm_fd = shm_open(SHM_NAME, O_RDWR, 0666)) == -1){
                        /* error handling */ ...
            }
        }
} else {
        printf(„New shared memory segment created\n");
        if(ftruncate(shm_fd,SHM_LEN)==-1){ /* error handling */ ...}
}
if((segptr = (char *)mmap(NULL, SHM_LEN,PROT_READ|PROT_WRITE,
                          MAP_SHARED, shm_fd,0)) == (char *)-1){
                            /* error handling */ ...
}
/* segptr[0],..., segptr[SHM_LEN-1] can be used to access shm segment
 * as if it was a memory buffer of length SHM_LEN
 */
munmap(segptr , SHM_LEN ); /* invalidate shm mapping when not needed */
```

Mechanizmy IPC Systemu V

	Kolejki komunikatów	Pamięć wspólna	Semafory
Plik nagłówkowy	**<sys/msg.h>**	**<sys/shm.h>**	**<sys/sem.h>**
Tworzenie/ otwieranie	**msgget()**	**shmget()**	**semget()**
Operacje sterujące	**msgctl()**	**shmctl()**	**semctl()**
Operacje komunikacji	**msgsnd()** **msgrcv()**	**shmat()** **shmdt()**	**semop()**

- Trwałość obiektów IPC Systemu V to tzw. **trwałość jądra** (*kernel persistence*) – obiekty istnieją do **przeładowania systemu** lub do **jawnego usunięcia**
- **Przestrzeń nazw**:
  - Obiekty są globalne (jedna przestrzeń nazw dla wszystkich procesów)
  - Klucz typu **key_t** (liczba całkowita dodatnia) identyfikuje obiekt w systemie. Zalecany sposób generacji:

    key_t ftok(const char *pathname, int id);
  - Po otwarciu obiekt jest dostępny przez **identyfikator obiektu** IPC Systemu V; identyfikator jest unikalny w ramach jednego mechanizmu IPC

IPC Systemu V – polecenia systemowe

Wyświetlanie własności obiektów IPC Systemu V aktualnie dostępnych

ipcs [-asmq] [-clupt] % informacja o obiektach wskazanego typu
ipcs [-smq] -i id % informacja o obiekcie o wskazanym identyfikatorze

Można również używać interfejsu systemu plików (**man namespaces(7)**):

/proc/sysvipc/msg, /proc/sysvipc/sem, /proc/sysvipc/shm

Usuwanie obiektów IPC Systemu V aktualnie dostępnych (w wypadku segmentów pamięci usuwanie jest odroczone – do czasu odłączenia wszystkich procesów-użytkowników) wymaga podania identyfikator **id** albo klucza **key**:

ipcrm {msg | sem | shm } id	% usuwanie obiektu zadanego typu
ipcrm [-q | -s | -m] id	% j.w.
ipcrm [-Q | -S | -M] key	% j.w.

Prawa dostępu do obiektów IPC

Dla każdego obiektu IPC jądro systemu przechowuje strukturę (patrz **<sys/ipc.h>, svipc(7)**) opisującą prawa dostępu:

```
struct ipc_perm {
  uid_t uid;   /* UID użytkownika – właściciela */
  gid_t gid;   /* GID użytkownika – właściciela */
  uid_t cuid;  /* UID użytkownika – twórcy  obiektu */
  gid_t cgid;  /* GID użytkownika – twórcy obiektu */
  mode_t mode; /* tryby dostępu (RWXRWXRWX) */
  ulong_t seq; /* (SVR4) numer kolejny, zwiększany o 1
                  przy każdym usunięciu obiektu
                  o danym kluczu */
  key_t key;   /* klucz */
};
```

Sprawdzenia praw dostępu dokonuje się **przy każdej** operacji na obiekcie IPC.

Tworzenie i otwieranie obiektów IPC

Do tworzenia i otwierania dostępu do obiektów IPC Systemu V służy wywołanie o postaci:

```
int XXXget(key_t key, /* sz, */ int oflag)
```

XXX jest zastępowane przez

msg - dla kolejki komunikatów

shm - dla pamięci wspólnej (wówczas potrzebny jest parametr **size_t sz**)

sem - dla semaforów (wówczas potrzebny jest parametr **int sz**)

oflag jest kombinacją wartości określających prawa dostępu (RW-RW-RW-) oraz **IPC_CREAT** i ew. **IPC_EXCL**

Funkcja zwraca **całkowitoliczbowy identyfikator obiektu**, który jest wykorzystywany przez proces do realizacji operacji na obiekcie. Identyfikator jest unikalny w ramach każdego typu obiektu IPC.

Uwaga: podanie w argumencie **key** stałej **IPC_PRIVATE** daje gwarancję, że jest tworzony nowy, unikatowy obiekt IPC. Nie istnieje żadna kombinacja **pathname** i **id** w wywołaniu **ftok()**, która tworzy klucz o wartości **IPC_PRIVATE**

Tworzenie i otwieranie obiektów IPC – c.d

Sposoby tworzenia/otwierania dostępu do obiektów IPC Systemu V.

Argument **oflag**	Obiekt o podanym kluczu nie istnieje	Obiekt o podanym kluczu istnieje
Brak sygnalizatorów **IPC_CREAT** **IPC_EXCL**	Błąd, **errno==ENOENT**	w porządku, wskazanie istniejącego obiektu
IPC_CREAT	w porządku, utworzenie nowego wpisu	w porządku, wskazanie istniejącego obiektu
IPC_CREAT | **IPC_EXCL**	w porządku, utworzenie nowego wpisu	błąd, **errno==EEXIST**

Kolejki komunikatów IPC Systemu V

- Komunikaty mają postać struktury:

```
struct msgbuf {
        long mtype; /* typ komunikatu, musi być >0 */
        char mtext[1]; /* dane komunikatu, długość >=0 (tu:1)*/
}
```

  Maksymalna długość komunikatu (**MSGMAX**) i maksymalna liczba kolejek komunikatów w systemie (**MSGMNI**) są konfigurowalna na poziomie systemu.

- Kolejka jest identyfikowana przez *identyfikator kolejki*, zmienną typu **int**.
- Kolejka ma określoną, w czasie tworzenia, długość (maksymalna wartość **MSGMNB** konfigurowalna na poziomie systemu). Gdy zostanie ona przekroczona, proces piszący do kolejki będzie zablokowany (przy pracy w domyślnym trybie: blokującym).
- Komunikaty odczytywane z kolejki zachowują strukturę, chociaż w kolejce mogą znajdować się komunikaty o różnej długości.
- Operacja odczytu z pustej kolejki blokuje odbiorcę (wątek), jeśli dostęp jest w trybie z blokowaniem.

Uwaga: Wartości (**MSGMAX, MSGMNI, MSGMNB**) są dostępne przez interfejs systemu plików (**msgmax**, **msgmnb**, **msgmni** w **/proc/kernel**)

Wysyłanie komunikatów

```
int msgsnd( int msqid, struct msgbuf  *ptr,
            size_t len, int flag);
```

Funkcja wstawia komunikat o długości **len** wskazywany przez **ptr** do kolejki o identyfikatorze **msqid**, zwracając normalnie 0 (bądź –1 w przypadku niepowodzenia, **errno** określa przyczynę).

flag o wartości 0 => funkcja blokuje, jeśli brak miejsca na komunikat

flag o wartości **IPC_NOWAIT** => funkcja powraca z błędem (**errno==EAGAIN**), jeśli brakuje miejsca na komunikat w kolejce.

Uwagi:

- funkcja **msgsnd** nie interpretuje pola **mtext** struktury **msgbuf**
- **len** określa rozmiar danych, t.j. **sizeof(msgbuf)=sizeof(long); len** może być równe 0 (struktura komunikatu zawiera tylko typ).
- pole **mtype** struktury **msgbuf** pozwala wiązać wiadomości z tą samą wartością **mtype** w listę (porządek FIFO); system układa też inną listę: wszystkich komunikatów danej kolejki, uporządkowaną wg kolejności wstawienia do kolejki

Odbiór komunikatów

```
int msgrcv(int msqid, struct msgbuf  *ptr,
      size_t len, long type, int flag);
```

Funkcja odczytuje komunikat o maksymalnej długości **len** do struktury wskazywanej przez **ptr** z kolejki o identyfikatorze **msqid**, zwracając normalnie 0 (bądź –1 w przypadku niepowodzenia, **errno** określa przyczynę). Domyślnie funkcja blokuje, jeśli nie ma żądanej wiadomości.

type o wartości 0 => funkcja odczytuje najstarszy komunikat

type >0 => funkcja odczytuje najstarszy komunikat podanego typu (usuwając go z kolejki), jeśli jednocześnie (**flag & MSG_EXCEPT)** to pobierany jest najstarszy komunikat typu różnego od **type**

type<0 => funkcja odczytuje najstarszy komunikat typu <= **|type|**

flag o wartości 0 => funkcja blokuje, jeśli brak żądanego komunikatu

(flag & IPC_NOWAIT)!=0 => funkcja powraca z błędem (**ENOMSG**), jeśli aktualnie nie ma odpowiedniego komunikatu w kolejce

(flag & MSG_NOERROR)!=0 => funkcja obcina komunikat, który jest za długi (domyślnie zwracany jest błąd **E2BIG**).

Operacje sterujące kolejką komunikatów

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Dostępne polecenia (**cmd**) dla kolejki o danym identyfikatorze (**msqid**):

IPC_RMID – usuwa z systemu kolejkę komunikatów (z wiadomościami)

IPC_SET – ustawia nowe parametry kolejki, korzystając z czterech pól struktury **msqid_ds: msg_perm.uid, msg_perm.gid, msg_perm.mode, msg_qbytes**, aktualizując też automatycznie pole **msg_ctime**

Polecenia te może wykonać tylko proces z EUID równym **0**, msg_perm.**cuid**,msg_perm.**uid**

IPC_STAT – przekazuje do struktury wskazanej przez **buf** aktualne wartości parametrów kolejki

```
struct msqid_ds  zawiera m.in. następujące pola:
  struct ipc_perm msg_perm; /* prawa dostępu */
  struct msg    *msg_first, *msg_last; /* wskaźniki listy komunikatów */
  msglen_t      msg_cbytes; /* bieżąca liczba bajtów w kolejce */
  msgqnum_t     msg_qnum; /* bieżąca liczba komunikatów w kolejce */
  msglen_t      msg_qbytes; /* maks. dozwolona liczba bajtów w kolejce */
  pid_t msg_lspid, msg_lrpid; /* PID procesu ostatnio wywołującego
                                 msgsnd(),msgrcv() */
  time_t        msg_stime, msg_rtime, msg_ctime; /* czas ostatniego
                        wywołania msgsnd(), msgrcv(), msgctl() */
```

Fragmenty przykładu (msend1/mrecv1)

Nadawca (msend1.c)

```
int queue, i; packet p1;
   if((queue = msgget(QUEKEY,
      IPC_CREAT| S_IRUSR| S_IWUSR|
      S_IRGRP|S_IWGRP))<0){
             /* error */
   }
   p1.mtype=1;
   for(i = 0 ; i < 10; i++){
     snprintf(p1.mtext,TXTSZ,
             "Packet  %d\n",i)
      if(TEMP_FAILURE_RETRY(
         msgsnd(queue ,&p1, TXTSZ,
                0))<0){
                 /* error */
      }
      sleep(1);
   }/* for() */
   sleep(5);/* wait for reader */
   if(msgctl(queue ,IPC_RMID,NULL)<0){
      /* error handling */
   };
```

Odbiorca (mrecv1.c)

```
int queue; packet p1;
   if((queue = msgget(QUEKEY,
      IPC_CREAT|S_IRUSR|S_IWUSR|
      S_IRGRP|S_IWGRP))<0){
         /* error */
   }
   for(;;){
      if(TEMP_FAILURE_RETRY(
         msgrcv(queue ,& p1, TXTSZ,
                1,0))<0)
         break;
      printf("%s", p1.mtext);
   }
   if(errno) perror("mrecv1 error");
==========================
#define QUEKEY   0x00FF00
#define TXTSZ        80
typedef struct {
long mtype;
char mtext[TXTSZ];
} packet;
```

Pamięć wspólna IPC Systemu V

```
int shmget(key_t key, size_t len, int oflag);
```

Tworzy nowy/otwiera istniejący segment pamięci wspólnej o rozmiarze **len** i kluczu **key**. **oflag** jest alternatywą bitową praw dostępu i ew. **IPC_CREAT, IPC_EXCL.** Tworzony segment jest wypełniany zerami. W przypadku powodzenia funkcja zwraca **identyfikator segmentu**; przy niepowodzeniu **–1**.

```
void * shmat(int shmid, const void *addr, int flag);
```

Dołącza segment pamięci wspólnej o identyfikatorze **shmid** , zwracając wskaźnik do początku segmentu albo -1 – przy niepowodzeniu.

addr == 0 => adres segmentu wybierany jest przez jądro

addr != 0 => adres pod którym system ma dołączyć segment; jeśli przy tym **(flag & SHM_RND) !=0** => adres jest zaokrąglany do wielokrotności rozmiaru strony pamięci wirtualnej.

Segment jest dołączany w trybie tylko do odczytu, gdy **(flag&SHM_RDONLY)!=0**, domyślnie – w trybie odczytu i zapisu.

```
int shmdt(const void *shmaddr);
```

Odłącza segment wskazywany przez **shmaddr.** Segment nie jest usuwany, chyba że zaznaczono segment do usunięcia (**shmctl**) i nie ma więcej dołączeń do segmentu w systemie).

Operacje na pamięci wspólnej

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Dostępne polecenia (**cmd**) dla segmentu o danym identyfikatorze (**shmid**):

IPC_RMID – zaznacza segment do usunięcia (usunięcie odbędzie się po odłączeniu wszystkich procesów od segmentu, bądź przy zamknięciu systemu). Segment może usunąć proces o **EUID==0**, przez **twórcę** albo **właściciela** – określonych przez pola **shm_perm.cuid** i **shm_perm.uid** w strukturze informacyjnej **shmid_ds** segmentu.

IPC_SET – ustawia w strukturze informacyjnej **shmid_ds** segmentu wartości pól **shm_perm.uid, shm_perm.gid, shm_perm.mode** na wartości pobrane z bufora wskazywanego przez **buf**

Polecenia te może wykonać tylko proces z EUID równym **0**, shm_perm.**cuid**, shm_perm.**uid** (patrz struct shmid_ds)

IPC_STAT – zapisuje zawartość struktury **shmid_ds** segmentu do bufora wskazywanego przez **buf**

Struktura opisująca segment pamięci

```
struct shmid_ds. {
  struct _ipc_perm shm_perm; /* struct praw dostępu */
  size_t shm_segsz;  /* rozmiar segmentu w bajtach */
  pid_tshm_lpid;     /* PID procesu ostatniej operacji */
  pid_tshm_cpid;     /* PID procesu twórcy */
  shmat_t shm_nattch; /* aktualna liczba dołączeń */
  time_t shm_atime;  /* czas ostatniego dołączenia */
  time_t shm_dtime;  /* czas ostatniego odłączenia */
  time_t      shm_ctime;    /* czas ostatniej zmiany shmid_ds
                               funkcją shmctl */
...
};
```

Pamięć wspólna – prosty schemat użycia

```
key_t key; int   shmid;  char  *segptr; /* deklaracja zmiennych */
key = ftok(".", 'A'); /* pozyskanie klucza */
if((shmid = shmget(key , SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1){
   if(errno==EEXIST){
        printf("Segment pamięci istnieje\n");
        if((shmid = shmget(key , SEGSIZE, 0)) == -1){/* błąd */ ...}
   } else {/* błąd */ ... }
} else {
/* Tu można wykonać inicjalizację segmentu (domyślnie zerowany) */
}
/*  Dołączenie segmentu pamięci wspólnej do procesu */
if((segptr = (char *)shmat(shmid , 0, 0)) == (char *)-1) {
        /* Błąd dołączenia segmentu */ ...
} else printf("Segment dołączony\n");
/* Użytkowanie segmentu, t.j. odniesienia do
 segptr [0],... segptr [SEGSIZE-1]
*/
shmdt(segptr ); /* Odłączenie segmentu po wykorzystaniu */
```

Efekty wywołania funkcji systemowych

Typ obiektu	fork()	exec()	_exit()
Sys.V msg	bez wpływu	bez wpływu	bez wpływu
POSIX MQ	potomek dziedziczy kopie otwartych deskr.	deskryptory są zamykane	deskryptory sa zamykane
Sys V shm	przydzielone segmenty shm są dołączane do procesu potomnego	segmenty są odłączane	segmenty są odłączane
POSIX shm	potomek zachowuje odwzorowanie w pamięci	odwzorowanie jest usuwane	odwzorowanie jest usuwane
Odwz. Pamięci przez mmap()	Potomek zachowuje odwzorowanie w pamięci	odwzorowanie jest usuwane	odwzorowanie jest usuwane