IPC – part 1
Message queues and shared memory

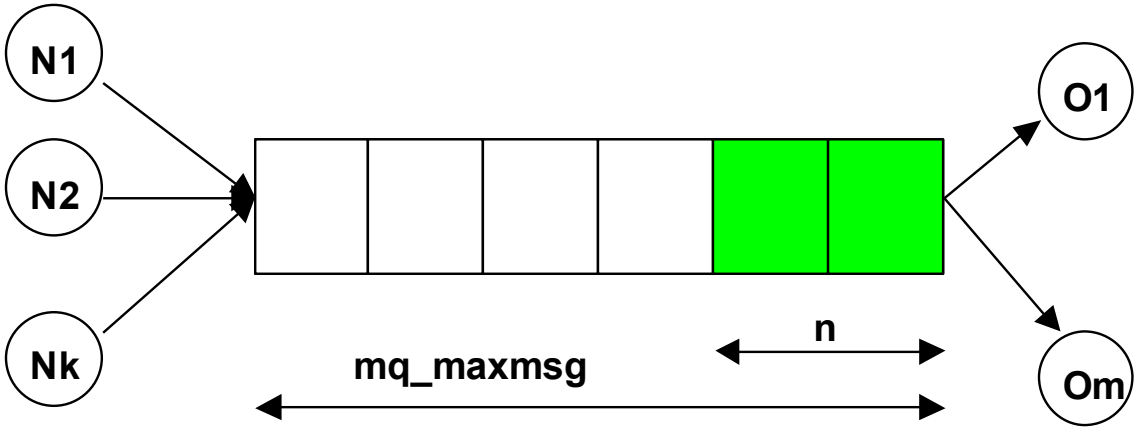
Last modification date: 03.03.2020

POSIX IPC

	Message queues	Shared memory	Semaphores
Header files	**<mqueue.h>**	**<sys/mman.h>**	**<semaphore.h>**
Creation/ opening access/removal	**mq_open(), mq_close(), mq_unlink()**	**shm_open(), shm_unlink()**	**sem_open() sem_close(), sem_unlink(), sem_init(), sem_destroy()**
Control operations	**mq_getattr(), mq_setattr()**	**ftruncate(), fstat()**	
Communication	**mq_send() mq_receive(), mq_notify()**	**mmap() munmap()**	**sem_wait(), sem_trywait(), sem_post(), sem_getvalue()**

- POSIX IPC object have mostly **kernel persistence** (with notable exception of the semaphore in memory, which is **process persistent**, i.e. it exists until the last process which uses it closes connection)

POSIX – message queues



The idea of message-queue based communication

Basic features of the message queues:

- Each queue can be targeted by unrelated processes. The queue can be visible in the system (eq. Linux) or not (depending on implementation).
- Message passing (of length 0 up to **mq_msgsize**) is reliable. The queue is kernel persistent, i.e. it exists until it is explicitly removed or till the system restart.
- The queue is identified by a descriptor of type **mqd_t**.. The queue descriptor can be implemented as the file descriptor.
- Each queue has finite capacity (**mq_maxmsg** messages)
- Message queue interface functions are available via the real-time library: **rt**.

POSIX – message queues

- Maximum message size (**mq_msgsize**) is specified during MQ creation. Message retrieval has to be always prepared to messages of maximum size..
- MQ has a maximum size (**mq_maxmsg**) which is specified during its creation. When a thread attempts to exceed it – will be blocked (when in standard blocking mode) until sufficient space is available or until interrupted by a signal.
- Messages can be of different length, but MQ keeps their integrity..
- Messages are assigned priorities (a nonnegative integer smaller than **MQ_PRIO_MAX**>=32). The highest priority messages are retrieved first.
- Message retrieval from the empty message queue blocks the thread – if the operation is performed in the default blocking mode.
- Maximum number of MQ that can be open in one process (**MQ_OPEN_MAX** >=8) is implementation dependent.
- A file interfaces exists to MQ parameters (**man namespaces(7)**) see: **/proc/sys/fs/mqueue**

The header and MQ attributes

- Basic data structures and functions related to POSIX message queues are defined in the header file: **<mqueue.h>**

- **MQ attributes** are exchanged within the following structure:

```
struct mq_attr {
  long mq_flags;   /* 0 or O_NONBLOCK */
  long mq_maxmsg;  /* Max. nr of message in MQ */
  long mq_msgsize; /* Max. message length (bytes) */
  long mq_curmsgs; /* Actual nr of messages in the queue */
};
```

Message queue creation/opening

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

The function returns MQ descriptor or **(mqd_t)(–1)** on failure – setting error code in **errno** .

Parameters:

name – the message queue name.

oflag - specifies access mode (**O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NONBLOCK**)

mode - access rights (**r** and **w** – as for files)

attr - pointer to the message attributes structure (fields: **mq_maxmsg, mq_msgsize**)

Remarks:

- Function call can be interrupted by signal delivery to the process calling **mq_open**() (the function returns –1, **errno==EINTR**),
- Linux implements message queues in a virtual file system, which can be mounted, e.g. to the folder **/dev/mqueue**. Information on message queues can be found in a subtree of: **/proc/sys/fs/mqueue/**

POSIX MQ – namespace and id-space

- The **name** argument points to a string naming a message queue.
  - It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments.
  - The **name** argument shall conform to the construction rules for a pathname.
    - If **name** begins with the slash character, then processes calling **mq_open( )** with the same value of **name** shall refer to the same message queue object, as long as that name has not been removed.
    - If **name** does not begin with the slash character, the effect is implementation-defined.
    - The interpretation of slash characters other than the leading slash character in **name** is implementation-defined. In Linux slash character has to be the first char of **name**; no other instances of that character in **name** are allowed.
- A message queue descriptor may be implemented using a file descriptor, in which case applications can open up to at least **{OPEN_MAX}** files and message queues.
- For Linux MQ characteristics: see **mq_overview(7)**

Closing access to MQ/ MQ removal

- When a process no longer needs an MQ it should close access to the MQ with:

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

- A message queue with given **name** can be removed with:

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

Note: the function removes the name of the MQ from the system immediately, but the queue is really destroyed when all processes, which opened access to this MQ close the MQ descriptor with **mq_close** call.

Sending messages

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

stores a message (msg_ptr[0],…msg_ptr[msg_len-1]) into MQ **mqdes**

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

same as **mq_send** but with limited wait time in case MQ is full.

Parameters:

mqdes - message queue descriptor
msg_ptr - address of the message buffer
msg_len - message length (in bytes)
msg_prio - message priority (0 up to MQ_PRIORITY_MAX)
abs_timeout - specifies a ceiling on the time for which the call will block. It is an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01, 00:00:00 (UTC)

```
struct timespec {
        time_t tv_sec;      /* seconds */
        long   tv_nsec;     /* nanosecondsy */
      };
```

The above function return 0 on success **–1** on failure (**errno** contains numeric error code).
The functions can be interrupted by signal delivery.

If several threads are blocked in **mq_send**()/**mq_timedsend**() because of full MQ, then upon free space release one thread is unblocked – the one of highest priority and longest waiting time.

Message retrieval

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

The function retrieves a message from the queue identified with descriptor **mqdes** to a buffer of length **msg_len**, pointed at by **msg_ptr**. It is the oldest message of highest priority. If **msg_prio_p!=NULL**, then ***msg_prio_p** is set to the message priority.

Upon success the function returns length of the message. On failure it returns -1 (after setting **errno**), and the queue does not change.

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

The function behaves as **mq_receive**(), but it returns not later than specified with ***abs_timeout** (absolute timeout is based on the **CLOCK_REALTIME** clock).

Remarks:

- Both functions can be interrupted by signal delivery.
- If a process sets up asynchronous notification and yet it blocks at **mq_receive()** call then a new message unblocks **mq_receive()** first.

Testing status of a message queue

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

It copies attribute structure of the MQ specified with descriptor **mqdes** to ***attr**

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

It copies attributes from ***newattr** to the attribute structure of the MQ specified with descriptor **mqdes**; the old attributes are stored in ***oldattr** (**oldattr!=NULL**).

Remark: **mq_setattr()** can only change attribute **mq_flags** (**0** or **O_NONBLOCK**).

Fragment of example code

Sender

```
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){/* error handling */ }
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);
```

Reader

```
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)
       {/* error handling */}
if(mq_getattr(mqdes,&attr)<0) {/* error*/}
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);
```

Asynchronous notification

int mq_notify(mqd_mqdes, struct sigevent *notification);

The function allows the calling process to register or unregister for delivery of one **asynchronous notification** when a new message arrives on the empty message queue referred to by the descriptor **mqdes.** For given MQ only one process can be notified. It is possible to specify the following notifications: :

- **Sending a signal** to the process (SIGEV_SIGNAL)
- **Start a thread** with given working function and starting arg. (SIGEV_THREAD)
- "null" notification (SIGEV_NONE)

If **notification==NULL** →process **unregisters notification**

```
struct sigevent {
  int    sigev_notify;  /* Method: SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD */
  int    sigev_signo;       /* Notification signal (for SIGEV_SIGNAL) */
  union sigval sigev_value; /* Data accompanying the notification */
  void (*sigev_notify_function) (union sigval);   /* Working function (for
            SIGEV_THREAD) */
  void  *sigev_notify_attributes;  /* Attributes of the thread working function */
};
union sigval {              /* Data accompanying notification */
  int     sival_int;
  void   *sival_ptr;
 };
```

Note: When the notification is sent to the registered process, its registration shall be **removed**.

Example – notification with SIGUSR1

```
void handler(int sig, siginfo_t *s, void *p){/* Note: RT
signal handling */
        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){/* błąd */}
 siga.sa_flags=SA_SIGINFO; siga.sa_sigaction=handler;

 if(sigaction(SIGUSR1,&siga,NULL)<0){/* error handling */}
 not.sigev_notify=SIGEV_SIGNAL; /* Notification by signal*/
 not.sigev_signo=SIGUSR1; /* Signal selection */
 if(mq_notify(mqdes, &not)<0)/* Registration of notif. */
        {/* error */}
```

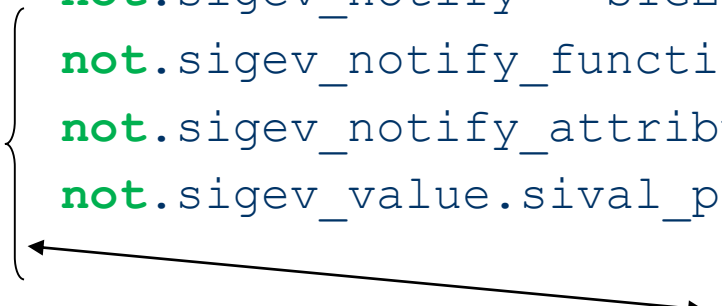
Example – notification with a new thread

```
int main(int argc, char *argv[]){
    mqd_t mqdes;
    struct sigevent not;
    if(argc!= 2){ /* missing MQ name*/}

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

    not.sigev_notify = SIGEV_THREAD;/* Notification by thread*/
    not.sigev_notify_function = tfunc;/* working function */
    not.sigev_notify_attributes = NULL;
    not.sigev_value.sival_ptr = & mqdes;    /* tfunc arguments*/

    if (mq_notify(mqdes, &not) == -1) {/* error handling */}
    pause(); /* Process will terminate in tfunc() */
    return EXIT_SUCCESS;
}
```

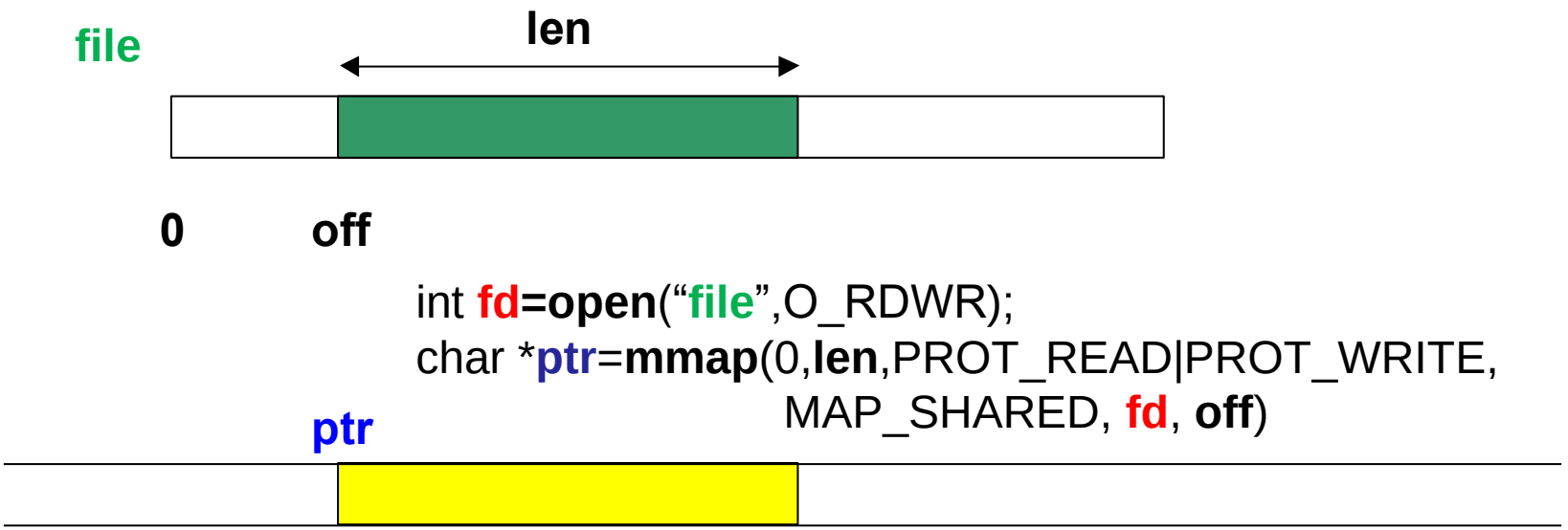


Example - continuation

```
static void  tfunc(union sigval sv){/* thread worker fun */
      struct mq_attr attr;
      ssize_t nr;
      void *buf;
      mqd_t mqdes = *((mqd_t *) sv.sival_ptr); /* MQ descr. */
/* Retrieval of MQ attributes (msg size needed) */
       if (mq_getattr(mqdes, &attr) == -1) {/* błąd */ }
/* Allocation of message buffer*/
       buf = malloc(attr.mq_msgsize);
       if (buf == NULL) {/* error handling … */}
/* Retrieval of the message (first in the queue) */
       nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);
       if (nr == -1) {/* error handling */}
       printf(„%ld B read from MQ\n”,(long) nr);
       free(buf);/* Freeing buffer */
       exit(EXIT_SUCCESS);          /* Process termination */
}
```

Memory mapping of files

Idea of memory mapping of a part of a file into the address space of a process



ptr[0] reference to the byte of the file number **off**

ptr[len-1] a reference to the last byte of the file which was mapped

Memory mapping of files – cont.

void * **mmap** (void ***addr**, size_t **len**, int **protect**, int **flags**, int **fd**, off_t **off**) – creates mapping of **len** bytes of an open file associated with descriptor **fd**, starting from byte nr **off**, to the address space of the calling process which is returned by **mmap.** Parameter **addr,** suggests the address to **mmap** (it is preferable to set **addr=0**, so that **mmap** makes a choice).

Parameter **protect**:

- **MAP_PRIVATE** updates to the mapped memory are not seen by other processes
- **MAP_SHARED** – memory changes are seen by other processes and are carried out to the underlying file

protect contains also protection bits: **PROT_READ, PROT_WRITE, PROT_EXEC**

int **msync** (void ***addr**, size_t **len**, int **flags**) – flushes changes made to the specified in-core copy of the mapped file (**len** bytes, starting from the address **addr)** to the mapped file.

Parameter **flags**:

- **MS_SYNC** – function waits until data are written to the file
- **MS_ASYNC** – the function only initiates the writing

int **munmap** (void *addr, size_t len) – unmaps memory range: **addr** to **addre+len-1**

POSIX IPC – shared memory

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

Creates a new shared memory segment and/or opens connection between the segment and a file descriptor which will be representing the segment. **`name`** is a name of the segment (restrictions as for message queue names).

oflag:

- **0** – opens connection
- **O_CREAT** – creates segment and opens connection
- **O_EXCL| O_CREAT** – creates a new segment and opens connection; otherwise fails

Uwagi:

- Creation of a new segment requires permission bits in: **mode**
- Linux creates POSIX shared memory object in a virtual file system, which can be mounted - typically in the folder **/dev/shm**
- Documentation of POSIX shared mamory: **shm_overview(7)**

POSIX IPC – shared memory use

After opening connection to a shared memory segment:

- it is necessary to set its size:

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

  Note: initial segment size is 0

- it is necessary to map the segment (represented with a file descriptor) into address space of the process using **mmap()** with **MAP_SHARED**
- use memory pointer to access the shared memory segment
- after use the segment should be unmapped with **munmap()**

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

Destroys specified **name** of the POSIX shared memory segment.

POSIX IPC – shared memory, cont.

Some other functions, which are related to POSIX shared memory segment use :

- **close()** – enables closing the file descriptor created with **shm_open()** when no longer needed.
- **fstat()** – can retrieve **struct stat** with such information about memory segments as:
  - **st_size** - size,
  - **st_mode** – permission bits
  - **st_uid, st_gid** –UID and GID of the owner
- **fchown()** – enables change of the owner
- **fchmod()** – enables change of permission bits

Example of shm use

```
#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 */
```

UNIX System V IPC

	Message queues	Shared memory	Semaphores
Header file	**<sys/msg.h>**	**<sys/shm.h>**	**<sys/sem.h>**
Creation/opening	**msgget()**	**shmget()**	**semget()**
Control operations	**msgctl()**	**shmctl()**	**semctl()**
Communication	**msgsnd()** **msgrcv()**	**shmat()** **shmdt()**	**semop()**

- Unix System V IPC objects have **kernel persistence** – they exist until the kernel is re-loaded or they are explicitly deleted.
- **Namespace**
  - IPC objects are global (one namespace, accessible to all processes)
  - A key of type **key_t** (positive integer) identifies an object in the system.. Suggested method of the key generation:

    key_t ftok(const char *pathname, int id);
  - After opening an IPC object is available to the process via IPC System V **object identifier**; the identifier is unique for each IPC object type.

UNIX System V IPC – commands

Properties of the currently available System V IPC objects can be displayed with operating system commands:

ipcs [-asmq] [-clupt] % info about IPC object of specified type

ipcs [-smq] -i id % info about IPC object of given id

One can also use file interface (see **namespaces(7)**):

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

Removal of System V IPC objects with system command **ipcrm** requires the object identifier **id** or **key** :

ipcrm {msg | sem | shm } id % removal of specific type of IPC object

ipcrm [-q | -s | -m] id % the same.

ipcrm [-Q | -S | -M] key % the same but with key not id

Unix System V IPC objects – access rights

For each IPC object the kernel creates and maintains the following structure (see **<sys/ipc.h>, svipc(7)**) with access rights to the object:

```
struct ipc_perm {
  uid_t uid;   /* UID id of the owner */
  gid_t gid;   /* GID id of the owner group */
  uid_t cuid;  /* UID id of the object creator (user id) */
  gid_t cgid;  /* GID id of the object creator group */
  mode_t mode; /* access rights (RWXRWXRWX) */
  ulong_t seq; /* (SVR4) sequence number, incremented after
       each destruction of an object with given key */
  key_t key;   /* key */
};
```

Access rights are checked prior to **each operation** involving IPC object.

Creation and opening access to IPC objects

The following describes generic format of functions which enable IPC object creation and opening access:

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

XXX is substituted by

msg - for message queues

shm - for shared memory (parameter **size_t sz** is needed than)

sem - for semaphores (parameter **int sz** is needed than)

oflag is a bit OR of patterns that determine access rights (RW-RW-RW-) and I**PC_CREAT**, possibly **IPC_EXCL**

The function returns integral object id, which is subsequently used for operations involving the object. The id is unique within a type of IPC objects (msg, shm, sem).

Note: If **key** takes the special value **IPC_PRIVATE** than a new IPC object is created and such, that no combination of **pathname** and **id** parameters in **ftok()** call is able to create the special value: **IPC_PRIVATE**.

Creation and opening access to IPC objects

Results of **XXXget()** function calls depend on the parameter **oflag** as shown below. Success of the attempt depends also on access rights, obviously.

Parameter **oflag**	IPC object with given key does not exists	IPC object with given key already exists
No **IPC_CREAT** and no **IPC_EXCL** bit patterns	Error, **errno==ENOENT**	OK, the object with given key is subject to opening access operation
IPC_CREAT bit pattern set	OK, new object may be created	as above
(IPC_CREAT | IPC_EXCL) bit pattern set	OK, new object may be created	Error, **errno==EEXIST**

Unix System V IPC – message queues

- Messages are represented with structures of the following form:

```
struct msgbuf {
        long mtype; /* positive (>0) message type */
        char mtext[1];/* message content length (>=0)(here:1)*/
}
```

  Maximum message length (**MSGMAX**) is the operating system configurable value.

- The open message queue is represented by a identifier of **int**.type.
- The message queue has maximum capacity **MSGMNB** that is system configurable. When a process thread attempts to overflow the queue – is blocked (when performs message send in default blocking mode).
- Message are guaranteed to keep integrity, even though the messages can be of different sizes..
- Message retrieval blocks the process thread (if attempt is made in the default blocking mode) when the requested message is not found (e.g. queue is empty).
- The maximum number of queues (**MSGMNI**) is operating system configurable

Note: file interface is available for (**MSGMAX, MSGMNB, MSGMNI**) via **msgmax**, **msgmnb**, **msgmni** entries in the **/proc/kernel** directory

Sending messages

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

The function attempts to send the message of length **len** stored at address **ptr** to the message queue identified with **msqid** id. When successful the function returns 0 (–1 in case of error with global variable **errno** set to an numeric error code).

flag if 0 => the function blocks when there is not enough space for the message in the queue

flag equal to **IPC_NOWAIT** => the function returns error (**errno==EAGAIN**), when there is not enough space for the message in the queue

Remarks:

- **msgsnd** does not interpret the message structure (besides **mtype**)
- **len** specifies size of data part of the struct **msgbuf**, i.e. should be equal to **sizeof(msgbuf) – sizeof(long)** ; **len** can be 0.
- Message type (**mtype**) makes possible to link all messages of the same type in a list (FIFO order); the message queue creates also a list of all messages (FIFO order).

Message retrieval

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

The function retrieves a message of maximum length (of data part) equal to **len** into a buffer pointed at by **ptr** from a message queue with identifier **msqid**. On success the message is removed from the queue and 0 is returned.–1 is returned in case of error with global variable **errno** set to an numeric error code). By default the function blocks if no requested message is found.

type == 0 => the function retrieves the oldest message

type >0 => the function retrieves the oldest message of specified type; if also (**flag & MSG_EXCEPT**) is not zero then the function retrieves the oldest message of type that is different from **type**

type<0 => the function retrieves the oldest message of type <= |**type**|

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

(flag & IPC_NOWAIT)!=0 => the function fails with **errno==ENOMSG** if no requested message is in the queue

(flag & MSG_NOERROR)!=0 => the function truncates too long message (normally failure is accompanied with **errno==E2BIG**.

Control operations

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

The following commands (**cmd**) can be issued for a message queue with given id (**msqid**):

IPC_RMID – the queue is destroyed immediately (the messages are lost)

IPC_SET – sets new parameters of the queue, 4 fields of the structure **msqid_ds** are modified: **msg_perm.uid, msg_perm.gid, msg_perm.mode, msg_qbytes**, also **msg_ctime** is updated. The command can be only performed for processes which EUID is equal: **0**, msg_perm.**cuid** or msg_perm.**uid**

IPC_STAT – copies to the structure pointed at by **buf** current parameters of the queue

```
struct msqid_ds  contains the following fields (among others):
  struct ipc_perm msg_perm; /* access rights */
  struct msg    *msg_first, *msg_last; /* ptrs to msg lists */
  msglen_t      msg_cbytes; /* current nr of bytes used */
  msgqnum_t     msg_qnum; /* current nr of messages*/
  msglen_t      msg_qbytes; /* max. nr of messages in the queue */
  pid_t msg_lspid, msg_lrpid; /* PID of the most recent process that
                        called msgsnd(),msgrcv() */
  time_t        msg_stime, msg_rtime, msg_ctime; /* time-stamp of the
                the most recent call of msgsnd(), msgrcv(), msgctl() */
```

Example fragments (msend1/mrecv1)

Sender (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 */
   };
```

Reader (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;
```

Unix System V IPC – shared memory

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

It creates new or opens existing shared memory segment of size **len** and key **key**. **oflag** contains bit OR-ed value of access rights and possibly **IPC_CREAT, IPC_EXCL.** The segment is **zeroed**. When successful the function returns **shared memory identifier** and **–1**on failure.

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

It attaches the shared memory segment specified with the identifier **shmid** to the address space of the calling process. Ptr to the beginning of the memory segment is returned on success; otherwise -1.

addr == 0 => segment location is selected by the kernel

addr != 0 => suggested address of the segment in the process space; if also **(flag & SHM_RND) !=0** => address is rounded to a multiple of the virtual memory page size.

Segment is attached in read-only mode when **(flag&SHM_RDONLY)!=0**, otherwise the access mode is read-write..

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

Detaches the segment located at address **shmaddr** from address space of the process. Note that the segment is not removed from the system – unless it has been marked for removal and there is no process currently attached to the segment.

Control operations

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

Available commands (**cmd**) for a shared memory segment with given id (**shmid**):

IPC_RMID – marks the segment for removal (it will be removed when no process attaches it or the system is closed). A segment can be marked by a process with **EUID==0**, the created or the owner – as determined by **shm_perm.cuid** and **shm_perm.uid** fields in the info structure **shmid_ds** of the segment.

IPC_SET – replaces values of **shm_perm.uid, shm_perm.gid, shm_perm.mode** fields of the info structure **shmid_ds** with values from the structure pointed at by **buf**

These operations can be performed by a process with **EUID** equal to **0**, shm_perm.**cuid** or shm_perm.**uid** (see struct shmid_ds)

IPC_STAT – stores the content of the info structure **shmid_ds** of the specified segment to the structure pointed at by **buf**

The info structure of shared memory segments

```
struct shmid_ds. {
  struct _ipc_perm shm_perm; /* access rights structure */
  size_t shm_segsz;  /* segment size in bytes */
  pid_tshm_lpid;     /* PID of last shmat()/smdt() */
  pid_tshm_cpid;     /* PID of creator */
  shmat_t shm_nattch; /* nr of current attaches */
  time_t shm_atime;  /* last attach time */
  time_t shm_dtime;  /* last detach time */
  time_t      shm_ctime;    /* last change time */
...
};
```

Example of shared memory segment use

```
key_t key; int   shmid;  char  *segptr; /* variables declared */
key = ftok(".", 'A'); /* key generation */
if((shmid = shmget(key , SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1){
   if(errno==EEXIST){
        printf("Shared memory segment exists\n");
        if((shmid = shmget(key , SEGSIZE, 0)) == -1){/* error */ ...}
   } else {/* error*/ ... }
} else {
  /* here some initialization of the segment might be done */
}
/*  Attachment of the segment to the process address space */
if((segptr = (char *)shmat(shmid , 0, 0)) == (char *)-1) {
        /* attach error*/ ...
} else printf("shared memory segment has been attached\n");
/* Here segment use might occur, in the form of references to
 segptr [0],... segptr [SEGSIZE-1]
*/
....
shmdt(segptr ); /* Segment detach after use */
```

Effects of some system calls on IPC objects

Object type	fork()	exec()	_exit()
Sys.V msg	Does not matter	Does not matter	Does not matter
POSIX MQ	Child inherits open descriptors	Descriptors are closed	Descriptors are closed
Sys V shm	Segments are attached to the child address space	Segments are detached	Segments are detached
POSIX shm	Child keeps memory mapping	Mapping removed	Mapping removed
Memory mapping with mmap()	Child keeps memory mapping	Mapping removed	Mapping removed