XSI (system V) IPC

System V IPC

- UNIX specification has undergone many stages
- Today, the Open Group extends some more UNIX definitions, and these are in the SUSv4 (Single Unix Specifications version 4) which is the current “UNIX” standard.
- Still, the following IPC objects were first implemented in System V unix, and are still called that way

identifiers and keys

- Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer identifier
- Unlike file descriptors, IPC identifiers are not small integers. Their numbers are not re-used if deleted
- Keys are used by processes to rendezvous with IPC objects. Users can use meaningfull names with keys.

keys basics

- Whenever we create a System V object (msgget, shmget, semget) we have to specify the key.
- The key is of type key_t (generally defined as a long int)
- The key is used to create the identifier.
- If we set key to IPC_PRIVATE, then the creator can create a new IPC object, and can share it with other processes related by fork() (children, grandchildren..)or by storing the identifier in a common file.

using keys(1) : IPC_PRIVATE

- If we set key to IPC_PRIVATE, then the caller will create a new IPC object
- The caller can store the identifier in a place where some other process can find it (like a file)
- If we have a parent-child situation (via fork()), then the child has an access to the identifier after the fork.

using keys(2) : agree on a value

- Both sides can agree on a value
- This can be achieved (for example) by using a common header file
- The problem is that the object can already exist
- In this case, the xget function fails (with error EEXIST)
- The creator can try to delete it, and re-create it again

using keys(3) : using ftok

- Both sides must agree on a **path** and **project id**
- path must be a real accessible file
- Id is an int, but only the lower 8 bits are used
- Note:
  ftok can still create keys that were already used in some situations.
  Always be ready to deal with errors.

flags: IPC_CREATE & IPC_EXCL

- All 3 get functions (msgget, shmget, semget) have a flags field, controlling what is done.
- If we use a “new” key, then we must specify IPC_CREATE to create a a new object.
- If we refer to a used key, we should get an access to an already created object
- If we specify both IPC_CREATE and IPC_EXCL
- If we use IPC_PRIVATE as a key, we will never get access to an old object, we’ll allways create a new one.

flags: permissions

- flags field (badly documented) is also used for permissions:

```
S_IRUSR    -    read permission, owner
S_IWUSR    -    write permission, owner

S_IRGRP    -    read permission, group
S_IWGRP    -    write permission, group

S_IROTH    -    read permission, others
S_IWOTH    -    write permission, others
```

message queue

System V Message queue

- A message queue is a linked list of messages stored within the kernel
- identified by a message queue identifier
- New messages are added to the end of a queue

  by msgsnd
- Every message has:
  - a positive long integer type field
  - a non-negative length
  - the actual data bytes

msqid_ds

- Each message queue has a data structure containing data about the current status of the queue.
- We can get a copy of it using msgctl

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  :
};
```

msgget

- **msgget** is used to create a new message queue
- New messages are added to the end of a queue by **msgsnd**
- Every message has a positive long integer **type** field
- Messages are fetched from a queue by **msgrcv**

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

int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, –1 on error

msgctl

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

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

Returns: 0 if OK, –1 on error

- Set cmd to:
  - IPC_STAT get a copy of msqid_ds
  - IPC_SET set fields like msg_perm.uid or mode
  - IPC_RMID Remove the message queue

msgsnd(cont.)

- msgsnd should block if the queue is full (no more empty slots, or no more available bytes, both are system limits)
- If we specify IPC_NOWAIT, then instead of blocking, the call returns immediately with error EAGAIN (that means: try AGAIN later)
- If we are blocked, then signal will interrupt the blocking (returning EINTR)
- If we are blocked and the queue is removed, then the block is interrupted (returning EIDRM)

receive order

- We don’t have to fetch the messages in a first-in, first-out order
- Instead, we can fetch messages based on their **type** field

msgrcv(1)

- Messages are retrieved from a queue by msgrcv.

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Returns: size of data portion of message if OK, –1 on error

- ptr should point to the same structure as msgsnd !!! (return message will be stored there)
- nbytes has the length of the data we are willing to receive (only the buffer)

msgrcv(2)

- If message is too large, then:

  - setting MSG_NOERROR will truncate the message (but we do not know that it was truncated)

  - if not set, we receive an error (E2BIG), and the message stays in the queue

msgrcv(3)

- The type argument lets us specify which message we want:

  - type == 0 The first message on the queue
  - type > 0 specify exactly the type we look for
  - type < 0 specify a number,
    reading types smaller than this number
    (starting with the lowest)

msgrcv(4)

- We can specify a flag value of IPC_NOWAIT
- ENOMSG will say that message (such as we soecified) could not be found
- The same behaviour as that of msgsnd regarding removal of the queue or signal

MQ exercises

CW: ping pong - step 1 (learn the api)

- Create 2 programs: Ping and Pong.
- Ping is run first (with -c)
- ping sends a series of messages to pong
- Pong responds to each one

  (How many queues do we need ?)
- Pong does not know how many messages will be sent !!!
- Both sides must exit gracefully, but the queue may stay in place

CW: ping pong - step 1 (cont.)

- Ping parameters:

-c	create the queue
-v	verbose
-e	implement EOF as a type (no value)
-d	delete queue
-f	queue name (otional)
-n	number of messages
-s	sleep time (msec)

CW: ping pong - step 1 (cont.)

- Pong parameters:

-v	verbose
-e	implement EOF as a type (no value)
-f	queue name (otional)
-s	sleep time (msec)

HW: ping pong - step 2

- In step 2 we may run several Ping programs, and several Pong programs, each in a terminal.
- Each Ping may receive a different number of messages (-n parameter).
- At least one Ping should run first.
- ONE QUEUE !!!
- Gracefull exit !!!
- Same parameters

System-V semaphores

semaphores

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- It could be used to control access to a shared resource
- This is based on an “atomic” operation, containing 2 functions:
  - first test if a condition is set (e.g: sqmaphore >0)
  - then set the condition (semaphore --)
- No other process (or thread) can go between these operations creating a race.

semaphores roles

- The functionality of a semaphore can be tought of serving 3 separate roles:

- notification: go to sleep until semaphore is set
- counting: “safe” counting
- lock: make sure only one process has access to a resource

system V Semaphores

- Are created in sets (more than a single semaphore at once)
- Are created (semget), then initialized (semctl)
  This can be a problem
- We have to remove semaphores after using them

setget

- creates a set of semaphores:

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

int semget(key_t key, int nsems, int flag);

                              Returns: semaphore ID if OK, –1 on error
```

- If we specify a new set, we have to set nsems.
  If referencing an existing set, we can leave it 0.
- The identifier retreived is an identifier for a set of semaphores (set of values for the semaphore)

setctl

- Use semctl for many operations for the semaphore set, including (among other):

    - IPC_RMID , remove the set
    - GETVAL get the value for a single semaphore in the set
    - SETVAL set a value
    - GETNCNT, # of proccesses waiting that semval>=0
    - GETZCNT   # of proccesses waiting that semval==0

semop

- Used to do the atomic operations on a set of values.

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, –1 on error

one look like this:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

sem_op field

- sem_op > 0, adding this to the value of the semaphore (returning resources) not blocking
- sem_op < 0, we want this number of units from the semaphore.

    - if semval - sem_op >= 0 then we do not block, and substruct this number

    - if semval - sem_op < 0, then we

        * block if IPC_NOWAIT is not set

        * return with error if it is

waiting to zero

- sem_op == 0 means that we want to wait (i.e. block) until semval will come to zero.
- This is unique, and can be usefull sometimes.

semop is atomic

- semop is atomic !!!
- That means it will do it’s operations on all values, or on none of them !!!