Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Inter-Process Communication (IPC)

- Processes may communicate via IPC *objects*
  - *message queues* allow processes to send and receive messages
  - *shared memory* allows processes to view a common area of memory where all processes can write/read
  - *semaphores* enable only a few process to access a shared resource or enable synchronization
- IPC objects are implemented by (at least) two standards:
  1. System V, older standard: first released by AT&T in 1983,
  2. POSIX, more recent standard inspired by System V, rapidly spreading and adopted by many

  They are both available in many Unix-systems, such as Linux
- The course will adopt System V standard, in the future we may switch to POSIX
- Some documentation can be found at
  `http://www.tldp.org/LDP/lpg/node7.html`
  (Warning: it is dated 1995!)

IPC objects: persistence

- IPC objects are *persistent*: they survive in the kernel space even after all the processes (creating or accessing the object) have terminated
  - this is good: IPC objects enable the communication between
    1. processes that just know the "name" of the IPC object (such as in FIFOs)
    2. even different invocations of the same executable
  - this is bad: it is worse than forgetting to `free` a dynamically allocated memory (by `malloc`)
    - if not explicitly removed (when needed), they can quickly fill up the memory
- It is possible to create, list or erase current IPC objects at command line
  - the command `ipcs` shows the status of current IPC objects
  - the command `ipcs -l` shows the system limits on the resources
    - also available in the `/proc/sys/kernel/` file system
    - can be modified by `sudo sysctl kernel.msgmax=65536`
  - the command `ipcmk` creates an IPC object (only Linux, not standard)
  - the command `ipcrm` erases the specified IPC object

IPC objects: IDs and keys

- Any System V IPC *object* (message queue, shared memory, or semaphore) is identified by a unique identifier (ID) of type `int`
  - uniqueness is per type: there may be two IPC objects of different type with the same ID
- Processes that are willing to use the same IPC object for communication, must both know its ID
  1. Processes may get the ID from a common *key*
- For each type of IPC object the `???get()` function returns the ID from a key
  1. `int msgget(key_t key, ...)` to get a message queue
  2. `int semget(key_t key, ...)` to get a semaphore
  3. `int shmget(key_t key, ...)` to get a shared memory

key

- How can two processes agree on a key?
  1. the key can be hard-coded in a common `.h` file (via `#define`)
  2. the key can be `IPC_PRIVATE` to create a new object (not really private since it may be shared, unfortunate choice of name)
  3. the key can be `getppid()`, if object shared among siblings

Getting an IPC object from a key

- All three types of IPC objects have a similar method to get the ID
- Any process accessing the object should call the `???get()` functions to get the ID, unless the ID is already known:
  - inherited from parent by `fork()`
  - passed as parameters at invocation time

```
int msgget(key_t key, int flags);
int shmget(key_t key, size_t size, int flags);
int semget(key_t key, int nsems, int flags);
```

- the IPC object identifier associated to `key` is returned. It may be:
  - the ID of a new object just created by calling `???get()`
  - the ID of an existing object previously created by others

  Check next slide for the precise behaviour
- `flags` specifies the read/write permissions of user/group/others in the standard octal form
  - 0400 read to user
  - 0020 write to group
  - 0666 read/write to everybody
  - ...
- Also `flags` may include macros in bitwise OR

Four ways to “get” an IPC object (Example: `msg` queues)

1. Create a **new object** and **communicate the ID to processes**

   `id = msgget(IPC_PRIVATE, 0600); /* 0600 is an example */`

   - then give the ID via command-line (through `execve(...)`) or
   - use the copied variable through `fork()`

2. Create a **new object** with a **given** `key`

   `id = msgget(key, IPC_CREAT | IPC_EXCL | flags);`

   - if IPC object with `key` **exists**, return −1 and `errno=EEXIST`
   - if IPC object with `key` does **not exist**, it is created

3. Use an **existing object** with a **given** `key`

   `id = msgget(key, 0644);            /* 0644 is an example */`

   - the ID of the **existing** object associated to `key` is returned
   - −1 returned and `errno=ENOENT` if no IPC object exists with `key`

4. Use an **existing object** with a **given** `key`, **or create it if not exists**

   `id = msgget(key, IPC_CREAT | 0660);`

   - if IPC object with `key` **exists**, same as `msgget(key, flags)`
   - if IPC object with `key` does **not exist**, same as
     `id = msgget(key, IPC_CREAT | IPC_EXCL | flags);`

Typical issues due to persistence

1. Say that your program creates and uses some IPC object
2. Say that your program crashes or it never ends and you have to stop it by `Ctrl+C`
3. Then you fix it and you launch it again

The IPC object of the second run still has the same content it had after the first run!!!

- Possible fixes:
  1. "get" the object with `IPC_PRIVATE` key it always returns a new object
     - ★ may run out of memory
  2. install `Ctrl+C` handler that cleans up objects
  3. remove old objects at command line by `ipcrm`

Outline

Lifecycle of a message queue

1. A message queue Q is created by process A
2. Q is opened for being used (send/receive) by processes $P_1, \ldots, P_n$
3. Processes $P_1, \ldots, P_n$ send and receive messages over Q as needed
   - sent messages are enqueued to the tail
   - received messaged are searched from the head (may pick messages other than the first one)
4. Sender processes send messages even if nobody will ever receive them
   - no `SIGPIPE`-like method when all read ends are closed (as in pipes)
   - if queue is full, **a process may be blocked forever**
5. Receiver processes cannot know when senders have finished
   - no `EOF`-like method when all write ends are closed (as in pipes)
   - if nobody is sending, **a process may be blocked forever**
6. Once sender processes have finished sending their messages, the message queue will persist in the kernel and receiver processes remain blocked waiting for a message until the queue exists
7. It is necessary to determine correctly the condition that allows the deallocation of the message queue

Creating a message queue (or getting ID of an existing one)

- The system call

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

  returns the identifier of a message queue associated to `key`
  - `msgflag` is a list of ORed ("|") options including:
    - read/write permissions (least significant 9 bits) in standard octal form (the "execute" (x) permission is ignored)
    - `IPC_CREAT`: if queue exists, its ID is returned; if it doesn't exists, it is created
    - `IPC_EXCL` (used only with `IPC_CREAT`): the call fails (with `errno=EEXIST`) if the queue exists
- Message queues are persistent objects: they will survive to the death of the creator, they must be erased expicitly

Message format

- Messages **must** start with a `long` value: the type of a message
  - the type **must** be strictly positive (not zero): sending a message with non-positive type results in an **error**
  - the type can be used to select messages to be read
- For example, the default message structure

```
struct msgbuf {
  long mtype;              /* type of message */
  /* my personal message goes here */
};
```

- The user can define any message structure as long as:
  1. the first `sizeof(long)` bytes are reserved to the message type
  2. the total length of the message does not exceed the maximum

     ```
     cat /proc/sys/kernel/msgmax
     ```
- Messages of length 0 are also acceptable. If so only `sizeof(long)` bytes are sent
- Do not use pointers in a message: pointers live into the memory of a process. A pointer written by another process does not make sense

Sending a message to a queue

- Messages are sent by the `msgsnd()` system call

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int
    msgflg);
```

- The caller process must have write permissions on the queue to send a message
  - `msqid`, the ID of the message queue where the message is sent
  - `msgp`, pointer to the message structure
  - `msgsz`, size of the message content (excluding `sizeof(long)` bytes of the heading type)
- If queue is full
  - the call `msgsnd()` blocks until some space for the message is made, or
  - if flag `IPC_NOWAIT` is set, it returns `-1` with `errno = EAGAIN`
- After processes have finished sending, they cannot close their "write end" as in pipes
  - Message queues are "closed" (erased) separately once they are no longer needed

Receiving a message

- To receive a message from the queue `msqid` and copy it to the buffer pointed by `msgp` the `msgrcv()` system call is used

```
int msgrcv(int msqid, void *msgp,
           int msgsz, long mtype, int msgflg);
```

- Process must have read permissions on the queue to receive a msg
- `msgsz` is the size of the message (without type) copied to the buffer
- The received message is selected as follows:
  - if `(mtype == 0)`, the first message in the queue is selected
  - if `(mtype > 0)`, the first message of type `mtype` is selected
  - if `(mtype > 0)` and `MSG_EXCEPT` flag is set, the first message of type **different than** `mtype` is selected
  - if `(mtype < 0)`, the first message in the queue of the **lowest** type less than or equal to `mtype` is selected (low types have a high priority)
- If no message is selected by the rules above
  - the call `msgrcv()` blocks until a selected message arrives, or
  - if flag `IPC_NOWAIT` is set, it returns `-1` with `errno = ENOMSG`
- the received message is erased from the queue (unless `MSG_COPY` flag)

Errors on sending/receiving messages

- Both `msgsnd()` and `msgrcv()` may fail and return `-1`
- The error code `errno` is as follows:
  - `EACCES`: no permission to operate
    - tried `msgsnd()`, but no write permission
    - tried `msgrcv()`, but no read permission
  - `EIDRM`: the message queue was removed (see later how to remove)
  - `EINTR`: the process caught a signal while waiting on a blocking `msgsnd()`/`msgrcv()` call
    - on full queue for `msgsnd()`, or
    - no selected message available for `msgrcv()`
    - `msgsnd()` and `msgrcv()` **cannot** be restarted after the handler with flag `SA_RESTART`
  - `ENOMEM`, `E2BIG`: system limits reached

Controlling (and deleting) a message queue by `msgctl()`

- The system call `msgctl()` enables several actions to be performed on the message queue

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

  - `msqid`, is the ID of the queue
  - `cmd`, describes the action to be taken over the queue
  - `buf`, is a parameter for the action (see next for details)
- To remove and deallocate the queue `msqid`

```
int msgctl(int msqid, IPC_RMID, NULL);
```

  - after the queue is removed, processes blocked on `msgrcv()` on the queue `msqid` will be unblocked with `errno=EIDRM`

Controlling (and deleting) a message queue by `msgctl()`

- To get the status of the queue

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

it will return the data structure of the queue (`man msgctl`)

```
struct msqid_ds {
  struct ipc_perm msg_perm; /* Owner,permission */
  time_t msg_stime;       /* Time of last msgsnd */
  time_t msg_rtime;       /* Time of last msgrcv */
  time_t msg_ctime;       /* Time of last change */
  msgqnum_t msg_qnum;      /* Cur # msg in queue */
  msglen_t msg_qbytes;/* Max bytes allowed in Q */
  pid_t msg_lspid;         /* PID of last msgsnd */
  pid_t msg_lrpid;         /* PID of last msgrcv */
};
```

Queues vs. pipes

- Message queues offer an IPC facility similar to pipes

	pipes	**message queues**
unit of data	byte	message (any user-defined data structure)
terminology	write, read, file descriptors	send, receive, IDs
lifecycle	closed after all read-/write file descriptors are closed	*persistent*: stay alive even after all processes (creator, senders, receivers) terminates
read blocks	if empty & some write ends are open	always if empty
write blocks	if full	if full
deallocation	implicitly after all fd are closed	must be made **explicitly** by the user
abstraction	low	high

Example of usage of message queues

- Sender process sends a message of type `argv[1]` to a queue. The text of the message is read from `stdin`
  `test-ipc-msg-snd.c`
- Receiver process receives a message of type `argv[1]` and prints its content to `stdout`. If a "special" message of type `MSGTYPE_RM` is received (which is a user-defined macro in *`ipc-msg-common.h`*), then the message queue is erased
  `test-ipc-msg-rcv.c`
- they share a common header file
  `ipc-msg-common.h`

Example 2 of usage of a message queue

- The parent process:
  1. Create a queue
  2. Forks `NUM_PROC` sender child processes
  3. Forks a receiver process
  4. Waits for the sender processes to terminate
  5. Waits for the queue to be empty
  6. Deallocate the queue, waits for the receiver, then exit
- Each sender child process:
  1. Sends `NUM_MSG` to the queue of type from `1` to `NUM_MSG`
- The receiver process:
  1. Receives all messages from the queue and prints them

test-ipc-msg-fork.c

Message queues: POSIX APIs

- For historical reasons, the course follows the System V API
- However, today the POSIX standard is dominant
- Here is a one slide overview
  - `man mq_overview` for an overview of POSIX message queues
  - Messages in POSIX queues have a *priority* (similar to SysV *type*)
  - `mq_open(...)`, `mq_close(...)` to create and close the queue
  - `mq_send(...)`, `mq_receive(...)` to send/receive messages
  - `mq_notify(...)` to enable asynchronous notification of message queue via `struct sigevent`
  - when interrupted by a handled signal, blocking system calls may be restarted if `SA_RESTART` flag is set