

LECTURE - XVII
INTERPROCESS COMMUNICATION

Tevfik Koşar

Louisiana State University
November 30th, 2010

Interprocess Communication (IPC)

- Threads may want to communicate beyond the process boundaries for:
 - Data Transfer & Sharing
 - Event notification
 - Resource Sharing & Synchronization
 - Process Control
- If threads belong to the same process, they execute in the same address space, i.e. they can access global (static) data or heap directly, without the help of the operating system.
- However, if threads belong to different processes, they cannot access each others address spaces without the help of the operating system.

2

Interprocess Communication (IPC)

- There are two fundamentally different approaches in IPC:
 - processes are residing on the same computer
 - (i.e. a shared memory system)
 - processes are residing on different computers
- The first case is easier to implement because processes can share memory either in the user space or in the system space.
- In the second case the computers do not share physical memory, they are connected via I/O devices (for example serial communication or Ethernet). Therefore the processes residing in different computers can not use memory as a means for communication

3

IPC Approaches

- We have already learned:
 - Shared memory
 - Pipes
 - Sockets
 - Signals
- We will learn:
 - Message Passing
 - FIFO (Named Pipes)

4

IPC: Message Passing

5

Message Passing

- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(message) – message size fixed or variable
 - **receive**(message)
- If *P* and *Q* wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

6

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

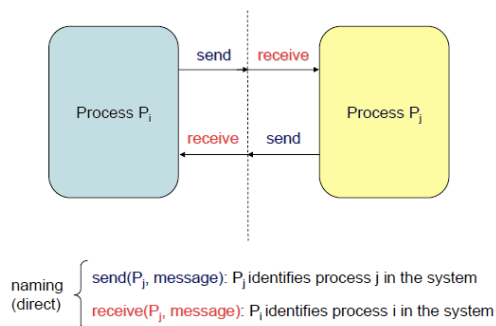
7

Direct Communication

- Processes must name each other explicitly:
 - `send(P, message)` – send a message to process P.
 - `receive(Q, message)` – receive a message from process Q.
- Properties of communication link:
 - Links are established automatically.
 - A link is associated with exactly one pair of communicating processes.
 - Between each pair there exists exactly one link.
 - The link may be unidirectional, but is usually bi-directional.

8

Direct Communication - Naming



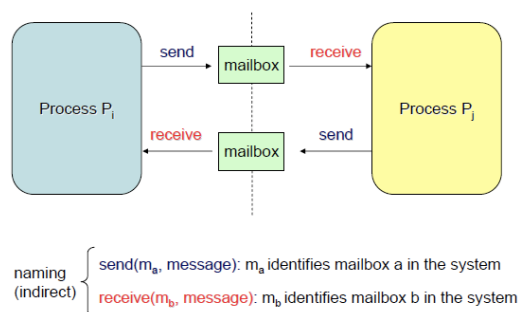
9

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports):
 - Each mailbox has a unique id,
 - Processes can communicate only if they share a mailbox.
- Properties of communication link:
 - Link established only if processes share a common mailbox,
 - A link may be associated with many processes,
 - Each pair of processes may share several communication links,
 - Link may be unidirectional or bi-directional.

10

Indirect Communication - Naming



11

Indirect Communication

- Operations:
 - create a new mailbox,
 - send and receive messages through mailbox,
 - destroy a mailbox.
- Primitives are defined as:
 - `send(A , message)` – send a message to mailbox A,
 - `receive(A , message)` – receive a message from mailbox A.

12

Indirect Communication

- Mailbox sharing:
 - P_1 , P_2 , and P_3 share mailbox A,
 - P_1 sends; P_2 and P_3 receive,
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes.
 - Allow only one process at a time to execute a receive operation.
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

13

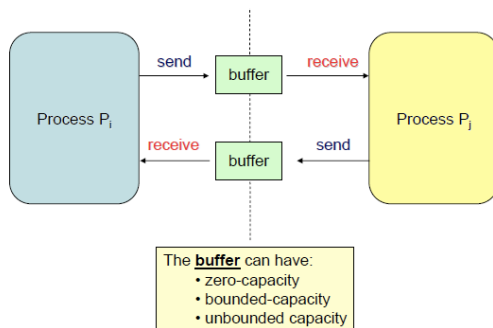
Buffering

Queue of messages attached to the link; implemented in one of three ways:

1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous).
2. Bounded capacity – finite length of n messages. Sender must wait if link full.
3. Unbounded capacity – infinite length.
Sender never waits.

14

Buffering



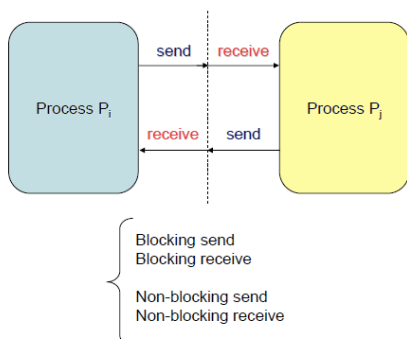
15

Synchronization

- *Message passing* may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**:
 - **Blocking send** has the sender block until the message is received.
 - **Blocking receive** has the receiver block until a message is available.
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** has the sender send the message and continue.
 - **Non-blocking receive** has the receiver receive a valid message or null.

16

Synchronization



17

Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long *message type*
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

18

Message Structure

- Each message structure must start with a long message type:

```
struct mymsg {  
    long msg_type;  
    char mytext[512]; /* rest of message */  
    int somethingelse;  
    ....  
};
```

19

Message Queue Limits

- Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain
- New messages cannot be added if *either* limit is hit (new writes will normally block)
- On linux, these limits are defined as (in /usr/include/linux/msg.h):
 - MSGMAX 8192 /*total number of messages */
 - MSBMNB 16384 /* max bytes in a queue */

20

Creating a Message Queue

- #include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
- The key parameter is either a non-zero identifier for the queue to be created or the value IPC_PRIVATE, which guarantees that a new queue is created.
- The msgflg parameter is the read-write permissions for the queue OR'ed with one of two flags:
 - IPC_CREAT will create a new queue or return an existing one
 - IPC_EXCL added will force the creation of a new queue, or return an error

21

Writing to a Message Queue

- int msgsnd (int msqid, const void * msg_ptr, size_t msg_size, int msgflags);
- msqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:
 - IPC_NOWAIT (non-blocking, return -1 immediately if queue is full)

22

Reading from a Message Queue

- int msgrcv(int msqid, const void * msg_ptr, size_t msg_size, long msgtype, int msgflags);
- msqid is the id returned from the msgget call
- msg_ptr is a pointer to the message structure
- msg_size is the size of that structure
- msgtype is set to:
 - = 0 first message available in FIFO stack
 - > 0 first message on queue whose type equals type
 - < 0 first message on queue whose type is the lowest value less than or equal to the absolute value of msgtype
- msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:
 - IPC_NOWAIT (non-blocking, return -1 immediately if queue is empty)

23

IPC: FIFO (Names Pipes)

24

Pipes are limited

Pipes depend on *shared file descriptors*, shared from a parent processes forking a child process, which *inherits* the open file descriptors as part of the parent's environment for the pipe.

- Question: How do two entirely *unrelated* processes communicate via a pipe?

25

FIFOs: Named Pipes

- FIFOs are “named” in the sense that they have a name in the filesystem (like a file!)
- This common name is used by two separate processes to communicate over a pipe
- The command `mknod` can be used to create a FIFO:

```
mknod MYFIFO (or “mknod MYFIFO p”)
ls -l
echo “hello world” >MYFIFO &
ls -l
cat <MYFIFO
```

26

Creating FIFOs in Code

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
Returns: 0 if OK, -1 otherwise
```

- **path** is the pathname to the FIFO to be created on the filesystem
- **mode** is a bitmask of permissions for the file, modified by the default umask
- **mkfifo** returns 0 on success, -1 on failure and sets `errno` (`perror()`)
- e.g. `mkfifo(“MYFIFO”, 0666);`

27

Example

```
int main(void)
{
    int    fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");

    if ( (fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ( (fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");

    clr_fl(fdread, O_NONBLOCK);

    exit(0);
}
```

28

FIFO vs Pipe

- Pipes do not create files, FIFOs do.
- Unrelated processes can communicate through FIFOs but not through Pipes.

29

FIFO vs File

- A file will keep all the data until deleted/overwritten while FIFO will dump the data after it is read.
- A write to a FIFO will block if there is no corresponding process reading from the pipe, usually blocking the whole process until there's a reader.
- One can only read or write from and to the FIFO, the pointer of the current position can not be moved (lseek is unacceptable)

30

Summary

- Interprocess Communication
 - Message Passing
 - FIFOs

- Next Lecture: Final Review

- Read Ch. 14 from Stevens
- Project-2 is due December 3rd



31

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J. Kim (KAIST), S. Guattery (Bucknell) and J. Schaumann (SIT).

32