Inter-Process Communications (IPCs)

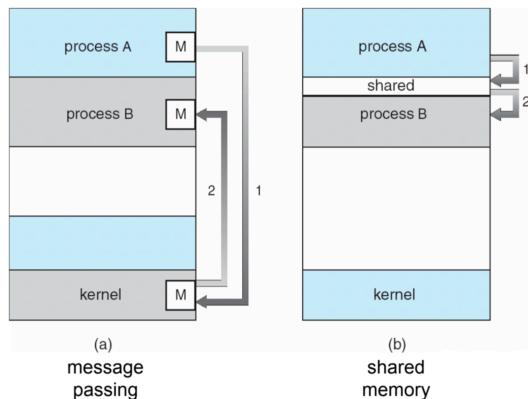
ICS412 - Fall 2009
Operating Systems

Henri Casanova (henric@hawaii.edu)

Communicating Processes

- Processes within a host may be **independent** or **cooperating**
- Reasons for cooperating processes:
  - Information sharing
    - e.g., Coordinated access to a shared file
  - Computation speedup
    - e.g., Each process uses a different core
  - Modularity
    - e.g., Systems designed as sets of processes are modular because one process can be easily replaced by another
  - Convenience
    - Some tasks are expressed naturally as sets of processes
- The means of communication for cooperating processes is called **Interprocess Communication** (**IPC**)
- Two broad models of IPC
  - Shared memory
  - Message passing

Communication Models



message passing

shared memory

Communication Models

- Most OSes implement both models
- Message-passing
  - useful for exchanging small amounts of data
  - simple to implement
- Shared memory
  - faster
    - Message passing: 1 syscall per communication
    - Shared memory: a few syscalls initially, and then none
  - more convenient for the user
  - more difficult to implement

Shared Memory

- Processes need to establish a shared memory region
  - One process creates a shared memory segment
  - Processes can then “attach” it to their address spaces
    - Note that this is really contrary to the memory protection idea central to multi-programming!
- Processes communicate by reading/writing to the shared memory region
  - They are responsible for not stepping on each other’s toes
  - The OS is not involved at all
- The textbook has a producer/consumer example, which you must read
  - Processes read/write data in a shared buffer
  - We’ll talk about producer/consumer again

Message Passing

- With message passing, processes do not share any address space for communicating
- Two fundamental operations:
  - send(message)
  - recv(message)
- If processes P and Q wish to communicate they
  - establish a communication “link” between them
    - This “link” is an abstraction that can be implemented in many ways
      - network link, shared memory
  - place calls to send() and recv()
  - optionally shutdown the communication “link”
- Message passing is key for distributed computing
  - Processes on different hosts cannot share physical memory!
- But it is also very useful for processes within the same host

Message Passing Design Decisions

- There are many possible design decisions
  - How are links established?
  - Fixed- or variable-length messages
    - Fixed is easier to implement in the OS, but more difficult for users to work with
  - Can a link be associated to more than two processes?
  - Can there be more than one link between two processes?
  - Is a link uni- or bi-directional?
  - etc.
- Let's look at 3 questions:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

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
- A variation: (asymmetric)
  - **send** (*P, message*) – send a message to process P
  - **receive**(*&Who, message*) – receive a message from any process, whose identify is stored in variable Who
- Challenge: establish/maintain a process name space?
  - What if processes can change id?

Indirect Communication

- Messages transit through mailboxes (or "ports")
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of the 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
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives:
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A, message*) – receive a message from mailbox A

Indirect Communication

- The mailbox sharing issue:
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Possible 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
    - Perhaps notify the sender of who the receiver was

Synchronous/Asynchronous

- Message passing may be either blocking or non-blocking
- **Blocking**, or **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
  - When both are blocking, the operation is called a rendez-vous communication style
- **Non-blocking**, or **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
    - With the option to check on status later ("was my message received?")
  - **Non-blocking** receive has the receiver receive a valid message or null
    - With the option to block
- The terms blocking/non-blocking and synchronous/asynchronous are typically used interchangeably
  - In some contexts, subtle differences are made, but we can ignore them

Buffering

- While messages are in transit, they reside in some location, called a "message queue" (or simply the "link")
- There are three typical message queue implementations
  - Zero-capacity
    - There can be no waiting message
    - The sender is blocked
    - This enforces a "rendez-vous"
  - Bounded capacity
    - At most n messages can reside in the queue
      - Or n message bytes
    - If the queue is full, then the sender must block
  - Unbounded capacity
    - The sender never blocks
      - There should never be anything truly unbounded though

IPCs in Practice

- Let's now look at a few example IPCs provided by real OSes
  - POSIX Shared Memory
  - Message Passing in Mach
  - Sockets
  - RPCs
  - LPCs in Win XP
  - Pipes
- In a future project you'll implement a small message passing facility as part of a kernel

Example: POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment

```
id = shmget(IPC_PRIVATE, size, IPC_R | IPC_W);
```

  - Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

  - Now the process can write to the shared memory

```
sprintf(shared_memory, "hello");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

  - Complete removal of the shared memory segment is done with

```
shmctl(id, IPC_RMID, NULL);
```

- See posix_shm_example.c

Example: POSIX Shared Memory

- Question: How do processes find out the ID of the shared memory segment?
- In posix_shm_example.c, the id is created before the fork() so that both parent and child know it
  - how convenient
- There is no general solution
  - The id could be passed as a command-line argument
  - The id could be stored in a file
  - Better: one could use message-passing to communicate the id!
- On a system that supports POSIX, you can find out the status of IPCs with the 'ipc -a' command
  - run it as root to be able to see everything
  - you'll see two other forms of ipcs: Message Queues, and Semaphores

Example: Mach Message Passing

- Section 3.5.2 in the textbook goes through a description of mailbox-based message passing in the Mach OS
  - It's not difficult, but make sure you read it
- Extra copies: one big performance hit for message-passing is data copy
  - At a minimum: two copies
    - copy from user space to kernel space
    - copy from kernel space back to user space
  - Mach uses some sort of hidden shared memory implementation of message-passing to avoid the copies!

Example: Sockets

- A socket is a communication endpoint, and with two such endpoints two processes can communicate
  - Socket = ip address + port number
- Sockets are typically used to communicate between two different hosts, but also work within a host
  - Most network communication in user programs is written on top of the socket abstraction
    - e.g., you'd find sockets in the code of a Web browser
- Section 3.6.1 describes Java Sockets
  - You must read it, especially if you have never used Java sockets before
- Let's look at a C socket example
  - We won't talk too much about sockets
  - They belong more in an networking course
  - See socket_server.c and socket_client.c on the Web site for full code (i.e., that checks all error codes)

C Sockets: Server

```
struct sockaddr_in sock_addr;
int sockfd, connectedfd;

sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&sock_addr, 0, sizeof(sock_addr));
sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons(1234);
sock_addr.sin_addr.s_addr = INADDR_ANY;

bind(sockfd,(const void *)&sock_addr, sizeof(sock_addr));
listen(sockfd, 10);
connectedfd = accept(sockfd, NULL, NULL);
char buffer[128];
read(connectedfd, buffer, 128)
fprintf(stdout,"Server received message: '%s'\n",buffer);
sprintf(buffer,"got it");
fprintf(stdout,"Server: writing to the socket\n");
write(connectedfd, buffer, 1+strlen(buffer));
shutdown(connectedfd, SHUT_RDWR);
close(connectedfd);
exit(0);
```

C Sockets: Client

```
struct sockaddr_in sock_addr;
int sockfd;

sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&sock_addr, 0, sizeof(sock_addr));
sock_addr.sin_family = AF_INET;
sock_addr.sin_port = htons(1234);
inet_pton(AF_INET, " localhost", &sock_addr.sin_addr) < 0);
connect(sockfd, (const void *)&sock_addr, sizeof(sock_addr));
char buffer[128];
sprintf(buffer,"hello there");
write(sockfd, buffer, 1+strlen(buffer));
read(sockfd, buffer, 128);
fprintf(stdout,"Client: received reply: '%s'\n",buffer);
shutdown(sockfd, SHUT_RDWR);
close(sockfd);
exit(0);
```

Remote Procedure Calls

- So far, we've seen unstructured message passing
  - A message is just a sequence of bytes
  - It's the application's responsibility to interpret the meaning of those bytes
- RPC provides a procedure invocation abstraction across hosts
  - A "client" invokes a procedure on a "server", just as it invokes a local procedure
- The magic is done by a client stub, which is code that:
  - marshals arguments
    - Structured to unstructured, under the cover
  - sends them over to a server
  - wait for the answer
  - unmarshals the returned values
    - Unstructured to structured, under the cover
- A variety of implementations exists
- See all details in Section 3.6.2 in the textbook
- We'll talk more about RPCs later in the semester

Local Procedure Calls in XP

- Windows XP uses an LPC mechanism for structured message passing between processes on the same host
  - Essentially like RPC, but just happens to be local, and therefore doesn't go out to the network
  - Described in detail in Section 3.5.3
- LPCs are not visible to the application program, but are hidden inside the code of the Win32 library
  - It's something that system developers use, and that Win32 users use without knowing they do
- Like in Mach, a shared-memory trick is used to improve performance for large messages and avoid memory copies
  - The caller can request a shared memory region, in which messages will be stored/retrieved and not copied back and forth from user space to kernel space
    - This is obviously not possible with RPCs

UNIX Pipes

- Pipes are one of the most ancient, yet simple and useful, IPC mechanisms provided by UNIX
  - They've also been available in MS-DOS from the beginning
  - See the textbook for the Win32 pipe example
- In UNIX, a pipe is mono-directional
  - Two pipes must be used for bi-directional communication
- On talks of the write-end and the read-end of a pipe
- The system call to create a pipe is **pipe()**
- pipe() takes as input a blank array of two "file descriptors"
- Upon return, the value of each file descriptor is set
- Let's see briefly what a file descriptor is

File Descriptors

- A process performs I/O operations using a "file descriptor" abstraction
  - A file descriptor is simply an integer
  - A process can have a (bounder) number of open file descriptors
  - Typically user programs use them as "streams" of type FILE *
    - Under the cover, a FILE * contains a file descriptor
    - One can convert from a stream to a file descriptor with the fileno() call, and from a file descriptor to a stream with the fdopen() call
- Despite their names, file descriptors can be used for other purposes than referencing files in the file system
  - e.g., a file descriptor can be associated to a network socket
  - The "file" notion is to be taken in a very broad sense

The pipe() System Call

- Example call to pipe()

```
int fd[2];
if (pipe(fd)) {
   perror("pipe()");
   exit(1);
}
// fd[0] is the read-end
// fd[1] is the write-end
```

- A pipe is used to establish communications between a pair of parent-child processes
  - On UNIX and Windows systems
- This all works because the child shares the parent's file descriptor
  - e.g., useful for a server that accepts a socket connection and forks a child process to perform the work needed
- Important: each process closes one end of the pipe
- Textbook example of parent/child communication via pipe:
  - Figures 3.23 and 2.24

Pipe Example from Textbook

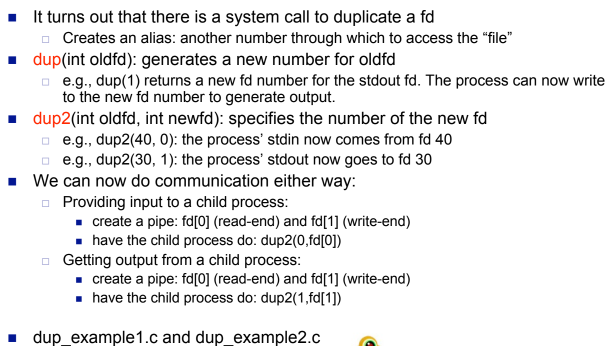
```
int fd[2];
char buffer[64];
pipe(fd);
if (!fork()) { // child
    close(fd[0]);
    write(fd[1], buffer, 64);
    close(fd[1]);
} else {
    close(fd[1]);
    read(fd[0], buffer, 64);
    close(fd[0]);
}
exit(0);
```

Pipes and exec()

- Sometimes, one wants to communicate with a child process that execs some executable
  - e.g., fork of a child that runs “/bin/ls” and retrieve its output
- In this case, one can’t put code in that executable to tell it to write its output to the write-end of a pipe
- Question: how can we establish communication?
- Answer: by overriding some standard file descriptors with the end of a pipe
- A standard way to do I/O for a process is to use stdin and stdout
  - e.g., system programs do this
- The standard C library defines stdin and stdout as FILE* streams
- We’ve seen that streams embed file descriptors
- These file descriptors are set as follows
  - stdin: 0
  - stdout: 1
  - (and stderr: 2)

Pipes and exec()

- It turns out that there is a system call to duplicate a fd
  - Creates an alias: another number through which to access the “file”
- dup(int oldfd): generates a new number for oldfd
  - e.g., dup(1) returns a new fd number for the stdout fd. The process can now write to the new fd number to generate output.
- dup2(int oldfd, int newfd): specifies the number of the new fd
  - e.g., dup2(40, 0): the process’ stdin now comes from fd 40
  - e.g., dup2(30, 1): the process’ stdout now goes to fd 30
- We can now do communication either way:
  - Providing input to a child process:
    - create a pipe: fd[0] (read-end) and fd[1] (write-end)
    - have the child process do: dup2(0,fd[0])
  - Getting output from a child process:
    - create a pipe: fd[0] (read-end) and fd[1] (write-end)
    - have the child process do: dup2(1,fd[1])
- dup_example1.c and dup_example2.c



Command-line Pipes

- The “pipe” command-line feature, |, corresponds to a pipe
- The command “ls | grep foo” creates two processes that communicate via a pipe
  - The ls process writes on the write-end
  - The grep process reads on the read-end
- An arbitrary number of pipes can be created:
  - ls -R / | grep foo | grep -v bar | wc -l

Named Pipes

- The pipes we’ve talk about so far are called anonymous pipes
- Another flavor are named pipes
  - UNIX calls them FIFOs
- These pipes exit in the file system and exit beyond the lifetime of individual processes
- See the short description on Page 139 in the textbook

Conclusion

- Communicating processes are the bases for many programs/services
- OSes provides two main ways for processes to communicate
  - shared memory
  - message-passing
- Each way comes with in many variants and flavors
- Reading: Sections 3.4, 3.5, 3.6
- You’ll use pipes for the last questions of Programming Assignment #2