CS 537
Lecture 4
Inter-Process Communication

Michael Swift

Questions for this Lecture

- How can multiple processes cooperate?

Interprocess Communication (IPC)

- To cooperate usefully, threads must communicate with each other
- How do processes and threads communicate?
  - Shared Memory
  - Message Passing
  - Signals

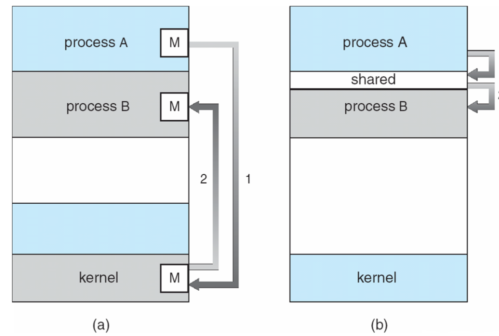
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
  - Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  – Information sharing
  – Computation speed-up
  – Modularity
  – Convenience

Communications Models



IPC: Shared Memory

- Processes
  – Each process has private address space
  – Explicitly set up shared memory segment within each address space
- Threads
  – Always share address space (use heap for shared data)
- Advantages
  – Fast and easy to share data
- Disadvantages
  – Must **synchronize** data accesses; error prone
- Synchronization: Topic for end of semester

IPC: Signals

- Signal
  – Software interrupt that notifies a process of an event
  – Examples: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT
- What happens when a signal is received?
  – Catch: Specify signal handler to be called
  – Ignore: Rely on OS default action
    - Example: Abort, memory dump, suspend or resume process
  – Mask: Block signal so it is not delivered
    - May be temporary (while handling signal of same type)
- Disadvantage
  – Does not specify any data to be exchanged
  – Complex semantics with threads
  – Not implemented in Windows

© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dussea, Michael Swift

IPC: Message Passing

- Message passing most commonly used between processes
  - Explicitly pass data btween **sender** (src) + **receiver** (destination)
  - Example: Unix pipes, Windows LPC
- Advantages:
  - Makes sharing explicit
  - Improves modularity (narrow interface)
  - Does not require trust between sender and receiver
- Disadvantages:
  - Performance overhead to copy messages
- Issues:
  - How to name source and destination?
    - One process, set of processes, or mailbox (port)
  - Does sending process wait (I.e., block) for receiver?
    - Blocking: Slows down sender
    - Non-blocking: Requires buffering between sender and receiver

IPC: Message Passing details

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- 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)

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE count)
             == out)
     ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
while (true) {
     while (in == out)
        ; // do nothing -- nothing to consume
     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
     return item;
  }
```

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

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

Example: Pipes

```
int pipedes[2], pid; // pipedes[0] = read, pipedes[1] = write
pipe(pipedes);
pid = fork();
if (pid == 0) { // child
  write(pipedes[1],"hello", sizeof("hello");
} else { //parent
  read(pipedes[0],buffer,100);
}
```

- Access to pipes is through file system calls; can be used most places a file can (Q: where not?)
- Kernel implements a bounded buffer; reader blocks if full and writer blocks if empty

Redirection

- Processes have a list of open files – a "file descriptor table" as part of the PCB
- File system calls provide an index (a file descriptor) into that table; table records whether descriptor is in use and points to a data structure representing the open file.
- On Unix, fd 0,1,2 are reserved:
  - fd 0 = standard input, can only be read
  - fd 1 = standard output, can only be written
  - fd 2 = standard error, can only be written

Redirection (2)

- By default, stdein, stdout, stderr refer to the console/ termainl
- In the shell, "redirection" comands change where these point:
  - Pipes: command1 | command2 means send stdout of command1 to stdin of command2 using a pipe
  - Redirecting: command1 > file means send stdout of command1 to a file
  - Redirecting: command2 < file means send the contents of a file to stdin

Implementing redirection

- Goal: replace file descriptors 0 and 1 for a new process:

```
outfile = open(outfile,"r");
pid = fork();
if (pid == 0) {
 close(stdout);
 dup2(outfile, stdout);
 exec(command2);
}
```

Redirection with pipes

```
shell> command1 | command2
int pipes[2], pid;
pipe(pipes);

pid = fork();
if (pid == 0) {
  close(stdout); close(pipes[0]);
  dup2(pipes[1],stdout);
  exec(command);
}
pid = fork();
if (pid == 0) {
  close(stdin); close(pipes[1]);
  dup2(pipes[0],stdin);
  exec(command2);
}
```