Interprocess Communication (IPC)

CS-502 Operating Systems
Fall 2006

(Slides include materials from *Operating System Concepts*, 7th ed., by Silbershatz, Galvin, & Gagne and from *Modern Operating Systems*, 2nd ed., by Tanenbaum)

Interprocess Communication

- Wide Variety of interprocess communication (IPC) mechanisms – e.g.,
    - Pipes & streams
    - Sockets & Messages
    - Remote Procedure Call
    - Shared memory techniques
  - OS dependent
- Depends on whether the communicating processes share all, part, or none of an address space

Common IPC mechanisms

- *Shared memory* – read/write to shared region
  - E.g., `shmget()`, `shmctl()` in Unix
  - Memory mapped files in WinNT/2000
  - Need critical section management
- *Semaphores* – *post_s*() notifies *wait*ing process
  - Shared memory or not, *but semaphores need to be shared*
- *Software interrupts* - process notified asynchronously
  - *signal* ()
- *Pipes* - unidirectional stream communication
- *Message passing* - processes send and receive messages
  - Across address spaces
- *Remote procedure call* – processes call functions in other address spaces
  - Same or different machines

Shared Memory

- Straightforward if processes already share entire address space
  - E.g., threads of one processes
  - E.g., all processes of some operating systems
    - eCos, Pilot
- Critical section management
  - Semaphores (or equivalent)
  - Monitors (see later)

Shared Memory (continued)

- More difficult if processes inherently have independent address spaces
  - E.g., Unix, Linux, Windows
- Special mechanisms to share a portion of virtual memory
  - E.g., `shmget()`, `shmctl()` in Unix
  - Memory mapped files in Windows XP/2000, Apollo DOMAIN, etc.
- Very, very hard to program!
  - Need critical section management among processes
  - Pointers are an issue

IPC – Software Interrupts

- Similar to hardware interrupt.
  - Processes interrupt each other
  - Non-process activities interrupt processes
- Asynchronous! Stops execution then restarts
  - Keyboard driven – e.g. cntl-C
  - An alarm scheduled by the process expires
    - Unix: SIGALRM from **`alarm()`** or **`settimer()`**
  - resource limit exceeded (disk quota, CPU time...)
  - programming errors: invalid data, divide by zero, etc.

Software Interrupts (continued)

- `SendInterrupt(pid, num)`
  - Send signal type **`num`** to process **`pid`**,
  - `kill()` in Unix
  - (NT doesn't allow signals to processes)
- `HandleInterrupt(num, handler)`
  - type **`num`**, use function **`handler`**
  - `signal()` in Unix
  - Use exception handler in WinNT/2000
- Typical handlers:
  - ignore
  - terminate (maybe w/core dump)
  - user-defined

IPC – Pipes

```
#include <iostream.h>
#include <unistd.h
#include <stdlib.h>
#define BUFFSIZE 1024
char data[ ] = “whatever”
int pipefd[2]; /* file descriptors for pipe ends */
 /* NO ERROR CHECKING, ILLUSTRATION ONLY!!!!! */
main() {
        char sbBuf[BUFFSIZE];
        pipe(pipefd);
        if (fork() > 0 ) { /* parent, read from pipe */
                close(pipefd[1]); /* close write end */
                read(pipefd[0], sbBuf, BUFFSIZE);
                /* do something with the data */
        }
        else {  close(pipefd[0]); /* close read end */
                 /* child, write data to pipe */
                write(pipefd[1], data, sizeof(DATA));
                close(pipefd[1]);
                exit(0);
        }
}
```

IPC – Pipes

```
#include <iostream.h>
#include <unistd.h
#include <stdlib.h>
#define BUFFSIZE 1024
char data[ ] = “whatever”
int pipefd[2]; /* file descriptors for pipe ends */
 /* NO ERROR CHECKING, ILLUSTRATION ONLY!!!!! */
main() {
        char sbBuf[BUFFSIZE];
        pipe(pipefd);
        if (fork() > 0 ) { /* parent, read from pipe */
                close(pipefd[1]); /* close write end */
                read(pipefd[0], sbBuf, BUFFSIZE);
                /* do something with the data */
        }
        else {  close(pipefd[0]); /* close read end */
                 /* child, write data to pipe */
                write(pipefd[1], data, sizeof(DATA));
                close(pipefd[1]);
                exit(0);
        }
}
```

IPC – Message Passing

- Indirect Communication – mailboxes
  - Messages are sent to a named area – *mailbox*
  - Processes read messages from the mailbox
  - Mailbox must be created and managed
  - Sender blocks if mailbox is full
  - Enables many-to-many communication
- Within one machine and among machines
  - MACH (CMU)
  - GEC 4080 (British telephone exchanges)

IPC – Message Passing

- Indirect Communication – mailboxes
  - Messages are sent to a named area – *mailbox*
  - Processes read messages from the mailbox
  - Mailbox must be created and managed
  - Sender blocks if mailbox is full
  - Enables many-to-many communication
- Within one machine and among machines
  - MACH (CMU)
  - GEC 4080 (British telephone exchanges)

IPC – Message Passing

- Indirect Communication – mailboxes
  - Messages are sent to a named area – *mailbox*
  - Processes read messages from the mailbox
  - Mailbox must be created and managed
  - Sender blocks if mailbox is full
  - Enables many-to-many communication
- Within one machine and among machines
  - MACH (CMU)
  - GEC 4080 (British telephone exchanges)

Acknowledgements

- A message back to sender indicating that original message was received correctly
- May be sent piggy-back on another message
  - *Implicit* or *explicit*
- May be *synchronous* or *asynchronous*
- May be *positive* or *negative*

Message Passing issues

- Scrambled messages (checksum)
- Lost messages (acknowledgements)
- Lost acknowledgements (sequence no.)
- Destination unreachable (down, terminates)
  - Mailbox full
- Naming
- Authentication
- Performance (copying, message building)

Beyond Semaphores

- Semaphores can help solve many traditional synchronization problems, BUT:
  - Have no direct relationship to the data being controlled
  - Difficult to use correctly; easily misused
    - Global variables
    - Proper usage requires superhuman attention to detail
- Another approach – use programming language support

Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors

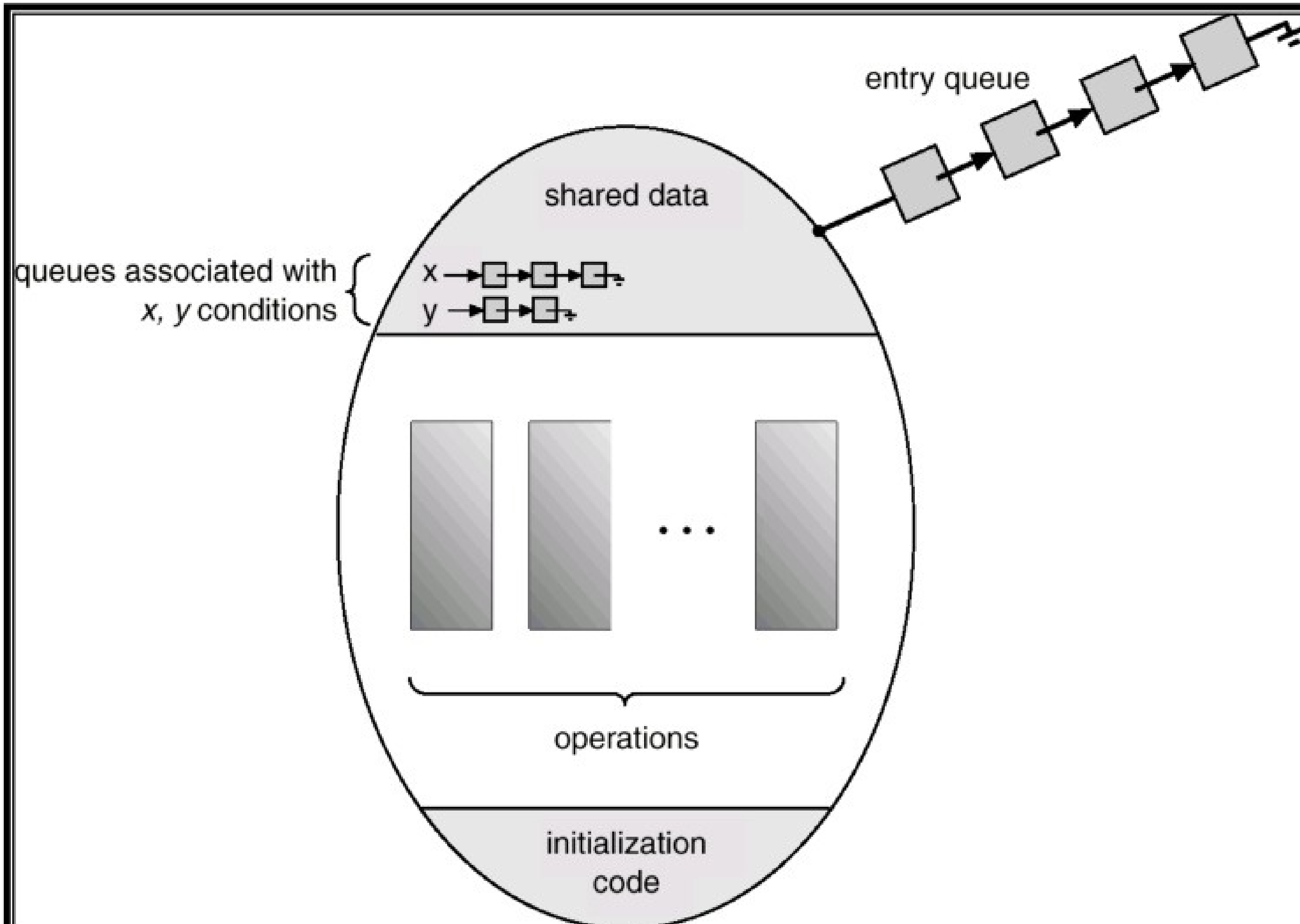
```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

wait and *signal* (continued)

- When process invokes `wait`, it relinquishes the monitor lock to allow other processes in.
- When process invokes `signal`, the resumed process must reacquire monitor lock before it can proceed (inside the monitor)

Monitors – Condition Variables



Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors

```
void insert (item i) {
    if (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}

item remove () {
   if (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Monitors – variations

- Hoare monitors: `signal(c)` means
  - run waiting process immediately (and acquires monitor lock)
  - signaler blocks immediately (and releases lock)
  - condition guaranteed to hold when waiter runs
- Mesa/Pilot monitors: `signal(c)` means
  - Waiting process is made ready, but signaler continues
    - waiter competes for monitor lock when signaler leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
  - being woken up is only a hint that something has changed
    - must recheck conditional case

Monitors (Mesa)

```
void insert (item i) {
    while (count == N) wait(full);
    /* add item i */
    count = count + 1;
    if (count == 1) then signal(empty);
}
item remove () {
   while (count == 0) wait(empty);
   /* remove item into i */
   count = count - 1;
   if (count == N-1) signal(full);
   return i;
}
```

Synchronization

- Semaphores
  - Easy to add, regardless of programming language
  - *Much* harder to use correctly
- Monitors
  - Easier to use and to get it right
  - Must have language support
  - Available in Java
- See
  - Lampson, B.W., and Redell, D. D., "Experience with Processes and Monitors in Mesa," *Communications of ACM,* vol. 23, pp. 105-117, Feb. 1980. (.pdf)
  - Redell, D. D. *et al.* "Pilot: An Operating System for a Personal Computer," *Communications of ACM,* vol. 23, pp. 81-91, Feb. 1980. (.pdf)

Remote Procedure Call

Remote Procedure Call (RPC)

- *The* most common means for communicating among processes of different address spaces
- Used both by operating systems and by applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems
- Fundamental idea: –
  - Server processes export an *interface* of procedures/functions that can be called by client programs
    - similar to library API, class definitions, etc.
- Clients make local procedure/function calls
  - As if directly linked with the server process
  - Under the covers, procedure/function call is converted into a message exchange with remote server process

RPC – Issues

- How to make the "remote" part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

RPC Model

- A server defines the service interface using an *interface definition language* (IDL)
  - the IDL specifies the names, parameters, and types for all client-callable server procedures
    - example: Sun's XDR (external data representation)
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
  - *Server-side* and *client-side*
- Linking:–
  - Server programmer implements the service's functions and links with the *server-side* stubs
  - Client programmer implements the client program and links it with *client-side* stubs
- Operation:–
  - Stubs manage all of the details of remote communication between client and server

RPC Stubs

- A *client-side stub* is a function that looks to the client as if it were a callable server function
  - I.e., same API as the server's implementation of the function
- A *server-side stub* looks like a caller to the server
  - I.e., like a hunk of code invoking the server function
- The client program thinks it's invoking the server
  - but it's calling into the client-side stub
- The server program thinks it's called by the client
  - but it's really called by the server-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

Marshalling Arguments

- *Marshalling* is the packing of function parameters into a message packet
  - the RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

RPC Binding

- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Remote Procedure Call is used …

- Between processes on different machines
  - E.g., client-server model
- Between processes on the same machine
  - More structured than simple message passing
- Between subsystems of an operating system
  - Windows XP (called *Local Procedure Call*)

Questions?

Next Topic