• Last class:
  – Process Creation
• Today:
  – Process Management
Process Description
Process State

• What do we need to track about a process?
Process Control Block

- State of running process
- Linked list of process control information
Per Process Control Info

- Process state
  - Ready, running, waiting (momentarily)
- Links to other processes
  - Children
- Memory Management
  - Segments and page tables
- Resources
  - Open files
- And Much More…
/proc File System

- Linux and Solaris
  - `ls /proc`
  - A directory for each process
- Various process information
  - `/proc/<pid>/io` -- I/O statistics
  - `/proc/<pid>/environ` -- Environment variables (in binary)
  - `/proc/<pid>/stat` -- process status and info
Context Switch

• OS switches from one execution context to another
  – One process to another process
  – Interrupt handling
  – Process to kernel *(mode transition, not context switch)*

• Current Process to New Process
  – Save the state of the current process
    • *Process control block* which describes the state of the process in the CPU
  – Load the saved context for the new process
    • Load the new process’s *process control block* into OS and registers
  – Start the new process

• Does this differ if we are running an interrupt handler?
Context Switch
Context Switch

• No useful work is being done during a context switch
  – Speed it up

• Hardware support
  – Multiple register sets (Sun UltraSPARC)

• However, hardware optimization may conflict
  – TLB flush is necessary
  – Different virtual to physical mappings on different processes
Process Description Summary

• Serves two purposes
  – Track per process resources
  – Save CPU state on context switch

• Process control block
  – Represents both aspects
  – CPU state
    • Program counter, registers
  – Resources
    • Linked lists of pages, child processes, files, etc.
Process Scheduling
Process Scheduling

• What do we need to know about processes to choose the next one to run?
  – Actual scheduling details/algorithms will be discussed later
Scheduling Processes

• Processes transition among *execution states*
Process States

• Running
  – Running == in processor and in memory with all resources

• Ready
  – Ready == in memory with all resources, waiting for dispatch

• Waiting
  – Waiting == waiting for some event to occur
  • see OSC 7e Fig. 3.2
State Transitions

• New Process $\Rightarrow$ Ready
  – Allocate resources
  – End of process queue

• Ready $\Rightarrow$ Running
  – Head of process queue
  – Scheduled

• Running $\Rightarrow$ Ready
  – Interrupt (Timer)
  – Back to end of process queue
State Transitions: Page Fault Handling

• Running ==> Waiting
  – Page fault exception (similar for syscall or I/O interrupt)
  – Wait for event

• Waiting ==> Ready
  – Event has occurred (page fault serviced)
  – End of process queue (or head?)

• Ready ==> Running
  – As before…
State Transitions: Other Issues

- Priorities
  - Can provide policy indicating which process should run next
    - More when we discuss scheduling…

- Yield
  - System call to give up processor
  - For a specific amount of time (sleep)

- Exit
  - Terminating signal (Ctrl-C)
Interprocess Communication
Process Communication

• Processes need to share information
  – Don’t work in isolation
• Discuss a variety of ways
  – Doesn’t include regular files and signals
IPC Mechanisms

• Two fundamental methods
• Shared memory
  – Pipes, shared buffer
• Message Passing
  – Mailboxes, Sockets
• Which one would you use and why?
Shared Memory

- Two processes share a memory region
  - One writes: *Producer*
  - One reads: *Consumer*
- Producer action
  - While buffer not full
  - Add stuff to buffer
- Consumer actions
  - When stuff in buffer
  - Read it
- Must manage where new stuff is in the buffer…
Shared Memory -- Producer

item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
Shared Memory -- Consumer

item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
Shared Memory

• Communicate by reading/writing from a specific memory location
  – Setup a shared memory region in your process
  – Permit others to attach to the shared memory region

• **shmget** -- create shared memory segment
  – Permissions (read and write)
  – Size
  – Returns an identifier for segment

• **shmat** -- attach to existing shared memory segment
  – Specify identifier
  – Location in local address space
  – Permissions (read and write)

• Also, operations for detach and control
Pipes

• Producer-Consumer mechanism
  – `prog1 | prog2`
  – The output of `prog1` becomes the input to `prog2`
  – More precisely,
    • The standard output of `prog1` is connected to the standard input of `prog2`
• OS sets up a fixed-size buffer
  – System calls: `pipe`, `dup`, `popen`
• Producer
  – Write to buffer, if space available
• Consumer
  – Read from buffer if data available
Pipes

- **Buffer management**
  - A finite region of memory (array or linked-list)
  - Wait to produce if no room
  - Wait to consume if empty
  - Produce and consume complete items

- **Access to buffer**
  - Write adds to buffer (updates end of buffer)
  - Reader removes stuff from buffer (updates start of buffer)
  - Both are updating buffer state

- **Issues**
  - What happens when end is reached (e.g., in finite array)?
  - What happens if reading and writing are concurrent?
IPC -- Message Passing

- Establish communication link
  - Producer *sends* on link
  - Consumer *receives* on link

- IPC Operations
  - **Y**: Send(X, message)
  - **X**: Receive(Y, message)

- Issues
  - What if X wants to receive from anyone?
  - What if X and Y aren’t ready at same time?
  - What size message can X receive?
  - Can other processes receive the same message from Y?
IPC -- Synchronous Messaging

- Direct communication from one process to another
- Synchronous send
  - \texttt{Send}(X, \text{message})
  - Producer must wait for the consumer to be ready to receive the message
- Synchronous receive
  - \texttt{Receive}(id, \text{message})
  - Id could be X or anyone
  - Wait for someone to deliver a message
  - Allocate enough space to receive message
- Synchronous means that both have to be ready!
IPC -- Asynchronous Messaging

- *Indirect* communication from one process to another
- Asynchronous send
  - `Send(M, message)`
  - Producer sends message to a buffer M (like a mailbox)
  - No waiting (modulo busy mailbox)
- Asynchronous receive
  - `Receive(M, message)`
  - Receive a message from a specific buffer (get your mail)
  - No waiting (modulo busy mailbox)
  - Allocate enough space to receive message
- Asynchronous means that you can send/receive when you’re ready
  - What are some issues with the buffer?
IPC -- Sockets

• Communication end point
  – Connect one socket to another (TCP/IP)
  – Send/receive message to/from another socket (UDP/IP)

• Sockets are named by
  – IP address (roughly, machine)
  – Port number (service: ssh, http, etc.)

• Semantics
  – Bidirectional link between a pair of sockets
  – Messages: unstructured stream of bytes

• Connection between
  – Processes on same machine (UNIX domain sockets)
  – Processes on different machines (TCP or UDP sockets)
  – User process and kernel (netlink sockets)
IPC -- Sockets

host X
(146.86.5.20)

socket
(146.86.5.2/1625)

web server
(161.25.19.8)

socket
(161.25.19.8/80)
IPC -- Sockets

• Issues
• Communication semantics
  • Reliable or not
• Naming
  – How do we know a machine’s IP address? DNS
  – How do we know a service’s port number?
• Protection
  – Which ports can a process use?
  – Who should you receive a message from?
  • Services are often open -- listen for any connection
• Performance
  – How many copies are necessary?
  – Data must be converted between various data types
Remote Procedure Calls

- IPC via a procedure call
  - Looks like a "normal" procedure call
  - However, the called procedure is run by another process
    - Maybe even on another machine

- RPC mechanism
  - Client stub
  - "Marshall" arguments
  - Find destination for RPC
  - Send call and marshalled arguments to destination (e.g., via socket)
  - Server stub
  - Unmarshalls arguments
  - Calls actual procedure on server side
  - Return results (marshall for return)
Remote Procedure Calls

- User calls kernel to send RPC message to procedure X.
- Kernel sends message to matchmaker to find port number.
- Kernel places port P in user RPC message.
- Kernel sends RPC to server.
- Matchmaker receives message, looks up answer.
- Matchmaker replies to client with port P.
- From: server To: client
  Port: kernel
  Re: RPC X
  Port: P
- Daemon listening to port P receives message.
- Daemon processes request and sends output.
- From: RPC To: client
  Port: kernel
  <output>
Remote Procedure Calls

• Supported by systems
  – Java RMI
  – CORBA

• Issues
  – Support to build client/server stubs and marshalling code
  – Layer on existing mechanism (e.g., sockets)
  – Remote party crashes… then what?

• Performance versus abstractions
  – What if the two processes are on the same machine?
Remote Procedure Calls

- Marshalling

```
val = server.someMethod(A,B)

stub

A, B, someMethod

boolean return value

remote object

boolean someMethod (Object x, Object y)
{
    implementation of someMethod
    ...
}
```
IPC Summary

• Lots of mechanisms
  – Pipes
  – Shared memory
  – Sockets
  – RPC

• Trade-offs
  – Ease of use, functionality, flexibility, performance

• Implementation must maximize these
  – Minimize copies (performance)
  – Synchronous vs Asynchronous (ease of use, flexibility)
  – Local vs Remote (functionality)
Summary

• Process
  – Execution state of a program

• Process Creation
  – fork and exec
    – From binary representation

• Process Description
  – Necessary to manage resources and context switch

• Process Scheduling
  – Process states and transitions among them

• Interprocess Communication
  – Ways for processes to interact (other than normal files)
• Next time: Threads