# Chapter 4 : Processes

- What is a process?
- How and why are processes *scheduled*?
- What kinds of operations can be performed on processes?
- How do processes work together?
- What's the difference between processes and threads?

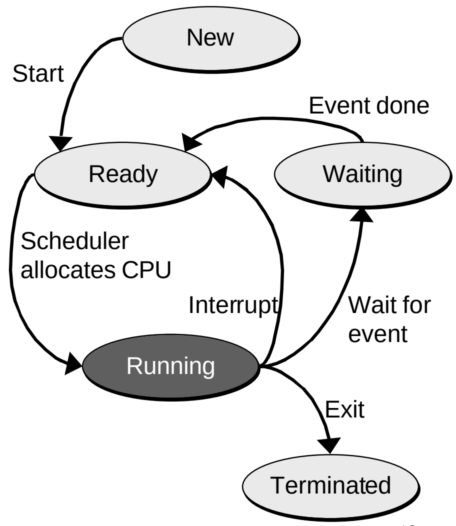



# What is a Process?

- A process is a program / job in execution
  - » Execution proceeds sequentially
  - » Program may be interactive or batch
- Job == process : the two terms are used interchangeably
- What characterizes a process?
  - » Program text (the code that's running)
  - » Current program counter (instruction that's currently being executed)
  - » Values on the stack
  - » Values in data section

# Process State

- Process changes state during execution
  - » New : created, but not yet run
  - » Running: currently using the CPU
  - » Waiting: waiting for some event (external to itself) to occur
  - » Ready: waiting to be assigned to a processor
  - » Terminated: finished execution
- Scheduler switches processes between these states



# Process State Information

- CPU must store information about a process when it’s not running
- Information includes
  - » Program
  - » Stack
  - » Data area
  - » “Miscellaneous”
- Miscellaneous information includes
  - » State of the CPU while running the process
  - » Scheduling information
  - » Memory usage information
  - » Accounting information
  - » I/O status

## Process Control Block (PCB)

- Holds process information not stored elsewhere
  - » Used to reactivate process
  - » Used to store process resource usage information
- Process control block includes
  - » Program counter (when process suspended)
  - » CPU registers (when process suspended)
  - » Information for
    - – CPU scheduling
    - – Memory management
    - – Accounting
    - – I/O status
- PCB is allocated when process created, freed when process exits




## Process Scheduling

- CPU scheduler picks a “ready” process to run
  - » If none available, executes the “idle” program
- Scheduler keeps processes in several queues
  - » Ready queue: list of processes ready to run (ready state)
  - » Device queues: for each device, process waiting for activity from the I/O device
- Processes move between queues
  - » I/O request moves process from ready to device queue
  - » Request completion moves process from device to ready queue
- Movement done by switching pointers in the PCB

## Long-term vs. Short-term Scheduling

- Long-term scheduler: selects process to run next
  - » Common in batch systems
  - » Not so common in interactive systems (user decides which process to run next)
  - » Runs infrequently (seconds or minutes)
  - » Controls number of programs running simultaneously (degree of multiprogramming)
- Short-term scheduler
  - » Picks which ready process to run next
  - » Allocates CPU to that process
  - » Present in interactive & batch systems
  - » Runs frequently (~10 milliseconds or less)




## Context Switch

- CPU is running one process, wants to switch to another process
- Requirements
  - » Save the current process's CPU state in its PCB
  - » Load the new process's CPU state from its PCB
- Context switch time is overhead : no useful work done by CPU during this time
  - » Reduce switch time to as little as possible
  - » Context switch infrequently to avoid wasting time
- Context switch is hardware-dependent
  - » Switch code varies greatly by CPU
  - » Speed of switch depends on how much information has to be swapped

## Creating a Process

- "Root" process created when the OS is first run
- Processes can create other processes
  - » Process's creator is called its parent
  - » Tree of processes, starting at root
  - » Issue: what if a process parent exits before it does?
- Resource sharing between processes
  - » Child can share none, some or all of parent's resources
  - » Who chooses the resources to be shared?
    - – Operating system
    - – Parent or child
- Address space (type of resource)
  - » Child gets a duplicate of parent's
  - » Child has a new program loaded




## Process Creation in UNIX

- How is it done?
  - » `Fork` system call creates a new process
    - – Duplicate of original process
  - » `Exec` system call replaces process memory space with a new program
    - – Usually (but not always) used immediately after `fork`
- Execution
  - » Parent can execute at same time as child
  - » Parent can wait for child to complete
  - » Parent can exit before child, in which case child becomes child of root process

## Exiting a Process

- Process executes last statement and exits
  - » Explicit exit: call to `exit()`
  - » Implicit exit: last statement in `main()` executed
- Parent (or OS) may abort child process
  - » Process is no longer needed
  - » Process is using too many resources
  - » Parent is terminating (OS may allow child to continue)
- Operating system deallocates process resources
  - » PCB returned to pool of PCBs in OS
  - » Memory freed up for reuse

## Two Processes are Better Than One?

- Many problems can be solved better with multiple streams of execution rather than one
  - » Solution may be simpler to write
  - » Task can be sped up
    - – Run on multiple CPUs
    - – One process waits for I/O while another runs
  - » Potentially more robust against process failure (bugs)
- Process must agree to work together
  - » Individual process is, by default independent of others
  - » Process explicitly requests to work with another process

# Producer - Consumer Problem

- Standard example of cooperating processes
  - » One or more *producer* processes: create information
  - » One or more *consumer* processes: consume (use) information generated by the producer(s)
- Producers place items into a buffer, and consumers pull items from the buffer
- Issue: how big is the buffer between producers & consumers?
  - » Unbounded-buffer: buffer can grow to essentially infinite size
  - » Bounded-buffer: buffer is a fixed (limited) size
  - » Different behaviors and implementations for each case




# Bounded-Buffer: Shared Memory

Variables

```
const int n;
typedef … Item;
Item buffer[n];
int in, out;
```

• Correct, but can only fill up n-1 slots

Producer

```
Item pitm;
while (1) {
  …
  produce an item into pitm
  …
  while ((in+1) % n == out)
    ;
  buffer[in] = pitm;
  in = (in+1) % n;
}
```

Consumer

```
Item citm;
while (1) {
  while (in == out)
    ;
  citm = buffer[out];
  out = (out+1) % n;
  …
  consume the item in citm
  …
}
```

## Threads: Sharing Even More

- What is a thread (lightweight process)?
  - » Program counter
  - » Registers
  - » Stack space
- What must threads share with peer threads?
  - » Code (text) section
  - » Data
  - » Operating system resources (usually)
- How are threads and processes related?
  - » Multiple threads operating together are called a task
  - » Traditional (heavyweight) process is a task with exactly one thread




## More on Threads

- Threads can be non-blocking
  - » One thread waits for I/O while another in the same task continues to run
  - » Provides higher throughput and improved performance
  - » Programming can be easier & more efficient
- Threads can be implemented in several places
  - » Kernel-level threads: kernel schedules threads
  - » User-level threads
    - – Implemented as a software library
    - – Kernel doesn't know about threading in application
  - » Both kernel-level and user level: most systems that support kernel-level threads can also supply user level threads

## Interprocess Communication (IPC)

- Processes must be able to communicate with other processes
- IPC is a message system that removes the need for shared variables (visible to the process)
- IPC facility must provide two basic operations:
  - » Send a message
  - » Receive a message
- If two processes want to communicate, they must
  - » Set up a communication link (using IPC facility calls)
  - » Exchange messages with send and receive
- Communication link may be
  - » Physical (shared memory, network)
  - » Logical (OS kernel creates “shared” variables)

## Issues for Implementation

- How are links established?
  - » How does one process find another?
  - » How do processes allow or disallow link formation?
- Can a link include more than one process?
  - » Are messages broadcast?
  - » Is a message delivered when all or one receives it?
- What is the capacity & speed of a link?
  - » Does it provide buffering?
- Are messages fixed or variable size?
  - » Library can be used to implement variable size messages (from the process’ point of view) using fixed size IPC
- Can information be sent two ways on a single link?

## Direct Communication

- Processes must explicitly send and receive messages
  - » Send (P, *message*): send *message* to process P
  - » Receive (Q, *message*): receive *message* from process Q
- Link established automatically when needed
- Exactly one link between any pair of processes
- Each link associated with exactly one pair of processes
- Still unresolved:
  - » How does a process find names of other processes?
  - » Does the receiver need to know who a message is from?




## Indirect Communication: Mailboxes

- Messages sent to or from mailboxes (ports)
  - » Mailbox is equivalent to buffer in producer-consumer
  - » Each mailbox has a unique identifier
  - » Processes can communicate through shared mailboxes
- Communication is done by:
  - » Send (M, *message*): send *message* to mailbox M
  - » Receive (M, *message*): receive *message* from mailbox M
- Communication link has these properties:
  - » Link may be associated with more than 2 processes
  - » Possibly links between any pair of processes
- Who gets messages sent to mailbox M?
  - » Arbitrary (OS decides randomly)
  - » At most one process may receive from mailbox M

## Link Capacity

- Zero capacity
  - » Sender must wait until receiver gets the message
  - » Implicit synchronization between two processes
- Bounded capacity
  - » Sender must wait if more than n messages in the link
  - » Sender can continue otherwise
- Unbounded capacity
  - » Sender never has to wait
  - » Always *some* limit - there’s not infinite capacity in the computer...

## IPC Errors

- Process termination: either sender or receiver terminates before message is delivered
  - » Delete message
  - » Allow message to be delivered
- Message lost
  - » OS resends message until it arrives
  - » Process resends message until it arrives
  - » OS notifies process of lost message
- Message scrambled
  - » Use checksums to detect error
  - » Resend (or drop) scrambled messages

## Example: UNIX

- UNIX supports send & receive and mailboxes (ports)
- Same method for local and remote communication
  - » OS handles network issues if necessary
- Finding a process:
  - » Unique port name (number) advertised publicly
  - » Process wanting to establish communication uses that port
  - » At most one process listens to a particular port
- Communicating
  - » Public port used to decide on private link (new port number)
  - » Send & receive done on the private link
- Cleaning up
  - » Port is released after communication is done
  - » Other processes may reuse the port



## Example: UNIX

- UNIX allows processes to share regions of memory
  - » Syntax varies depending on the kind of Unix
  - » Memory need not have the same address in all processes
- Memory may be shared between more than two processes
  - » Each process must explicitly ask to share the memory
  - » Memory sharing controlled: not just any process can do it
- Processes must manage synchronization themselves (more on that in a bit…)
  - » Operations on shared memory have to be ordered correctly
  - » Some primitives to coordinate between processes