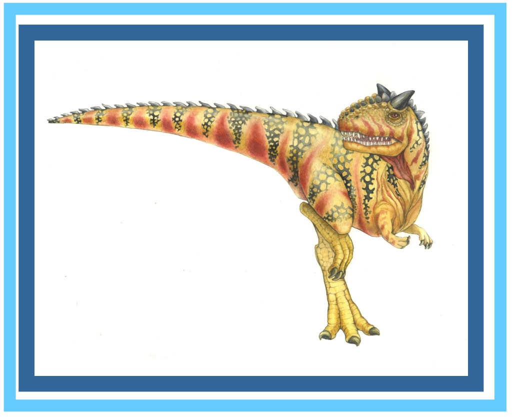
Chapter 3: Processes



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

modified by Stewart Weiss

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To describe communication in client-server systems

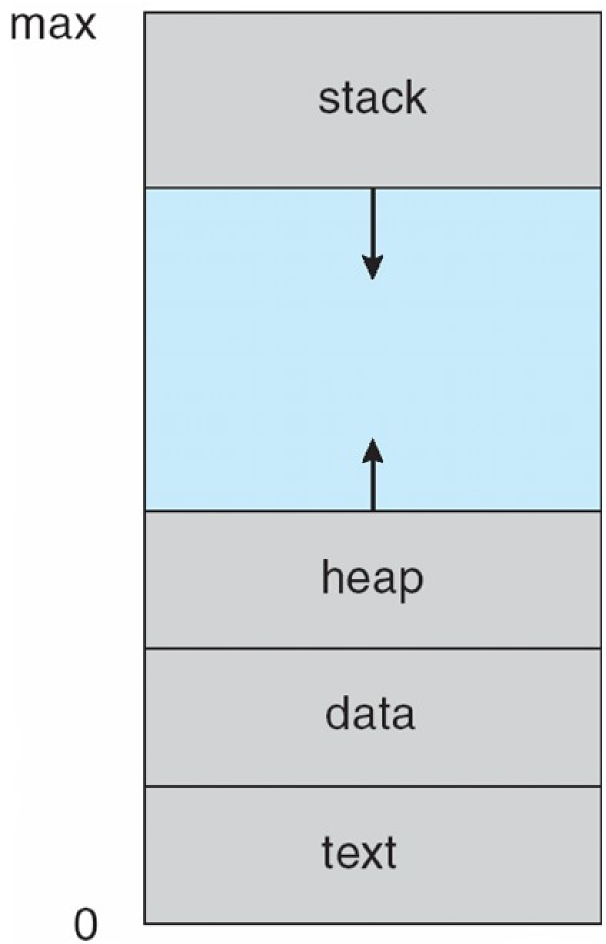
modified by Stewart Weiss

Process Concept

- An operating system executes programs:
  - Batch system – jobs are executed
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- ***Job – same definition as a process, but term is restricted to batch systems***
- A process includes:
  - program code (called the ***text***)
  - program counter
  - runtime stack
  - data section (for globals, both initialized and uninitialized)
  - heap (for dynamically allocated variables
  - other stuff that the operating system uses

modified by Stewart Weiss

Typical Memory Layout of Process

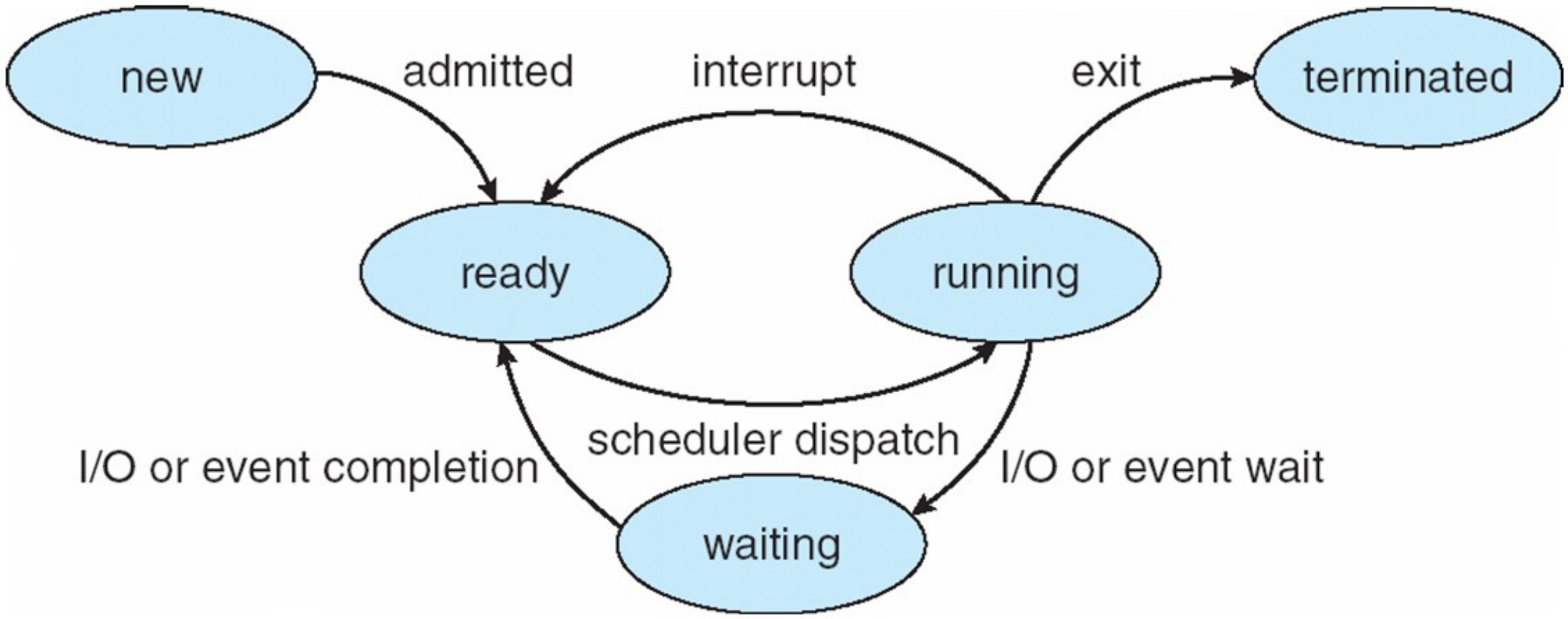


Process State

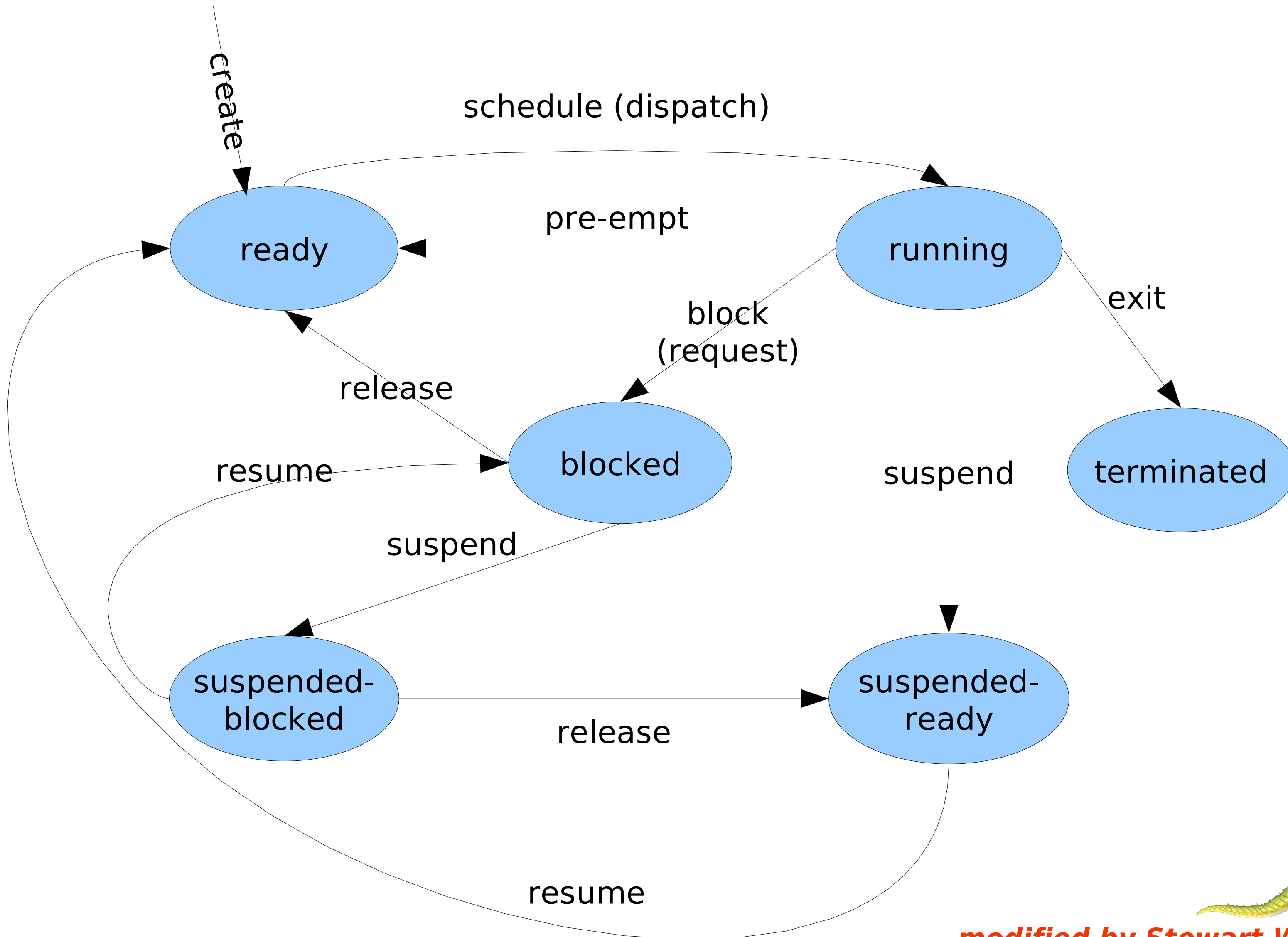
- As a process executes, it changes *state*
  - **new**: The process is being created ***(This is not a real state!)***
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
  - ***suspended sometimes***

modified by Stewart Weiss

Book's Diagram of Process State



My Diagram of Process State



modified by Stewart Weiss

Process Control Block (PCB)

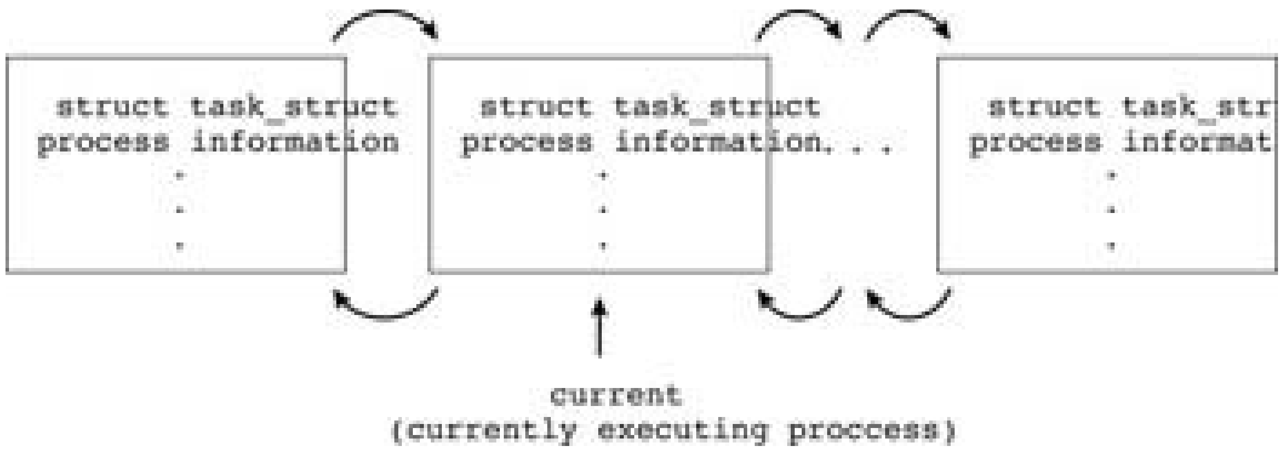
PCB is the data structure containing information associated with each process, needed by OS, including:

- Process state
- Program counter (only when not running!)
- CPU registers (hardware state, only when not running!)
- CPU scheduling information (priority etc)
- Memory-management information (where stuff is)
- Accounting information (e.g. how much resources used so far)
- I/O status information (which files open, what it's waiting for)
- list of child processes, parent process
- owner, group, etc

modified by Stewart Weiss

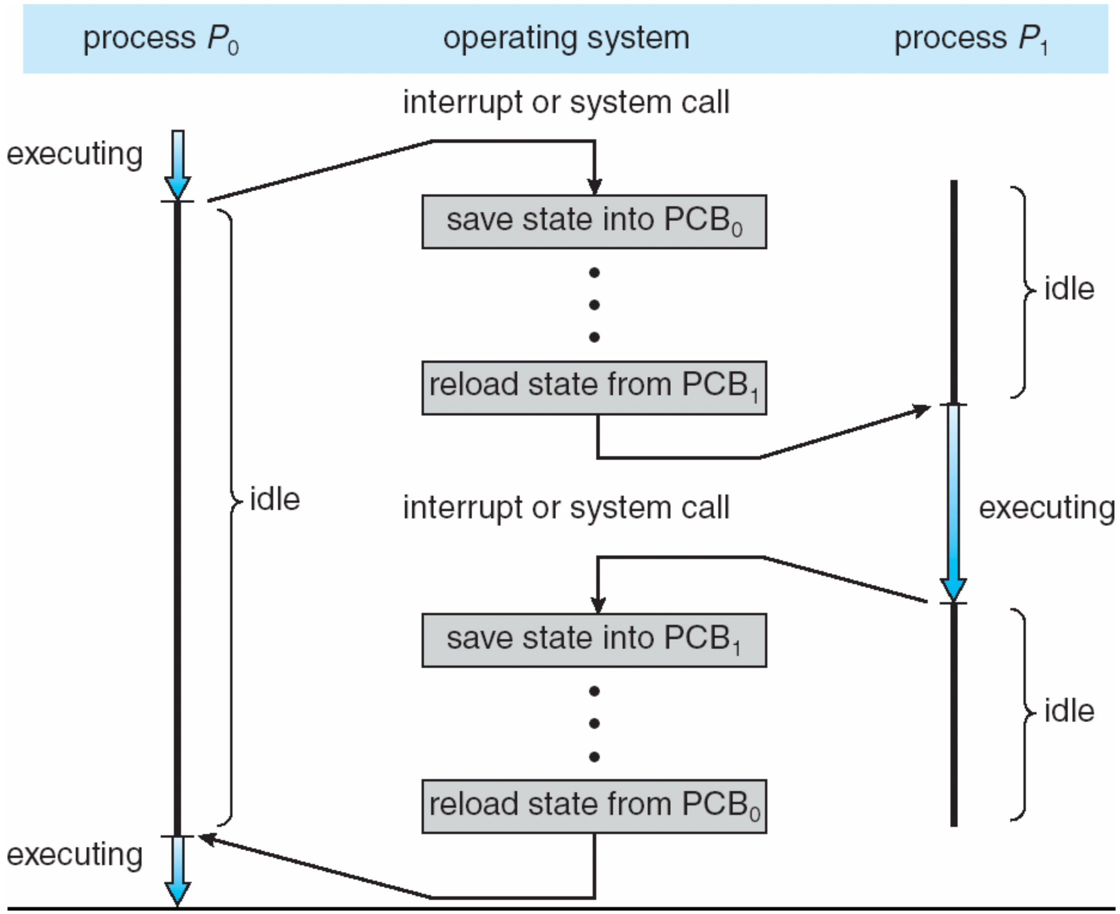
Process Table

- OS can keep PCBs in an array, or in linked lists, or in a mixed method like an array with embedded free list and active list (as was done in BSD UNIX).
- Linux uses linked lists of task_structs:



modified by Stewart Weiss

CPU Switch From Process to Process



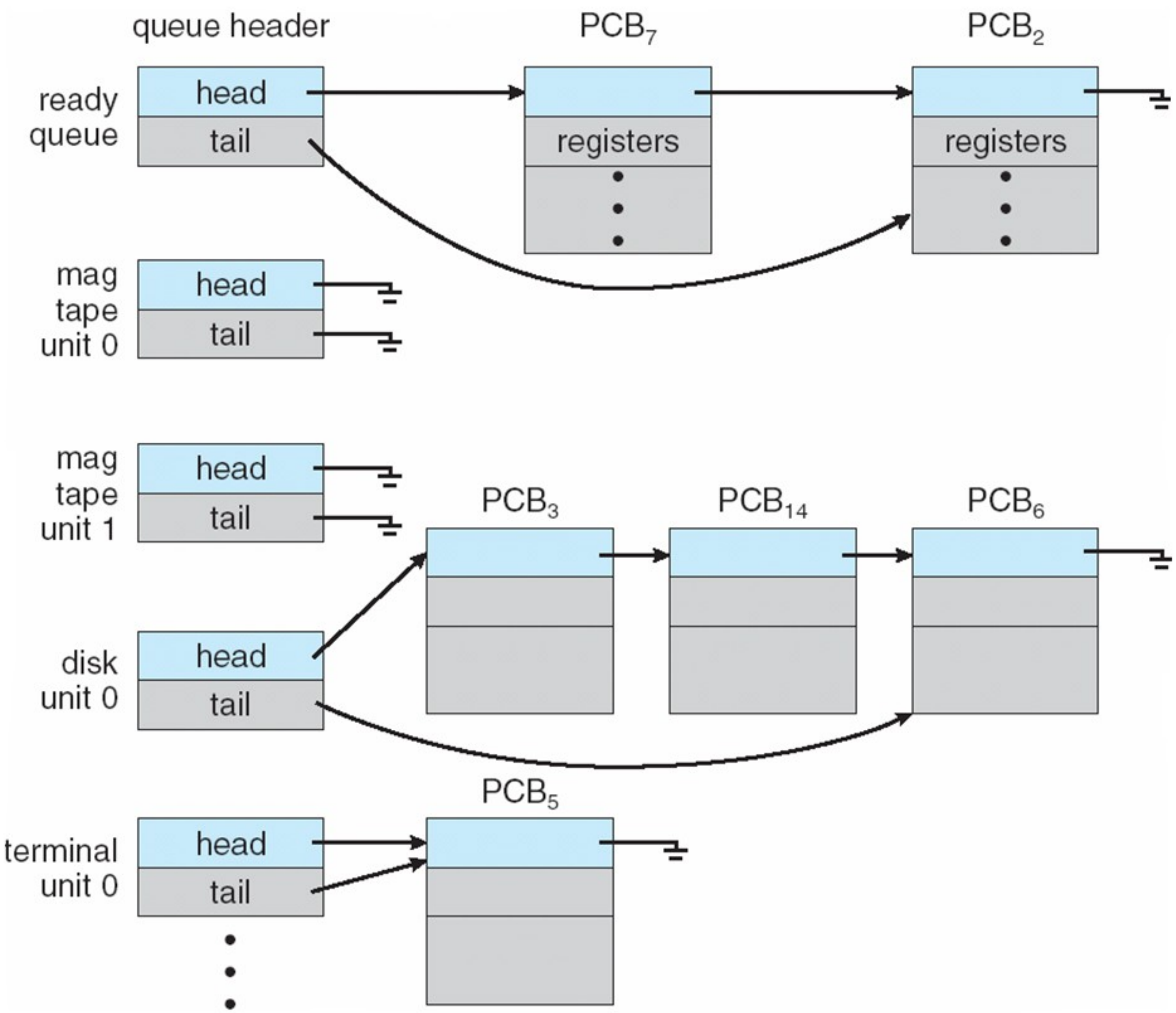
modified by Stewart Weiss

Process Scheduling Queues

- **Job queue** – set of all processes in the system ***(only in batch mode)***
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

modified by Stewart Weiss

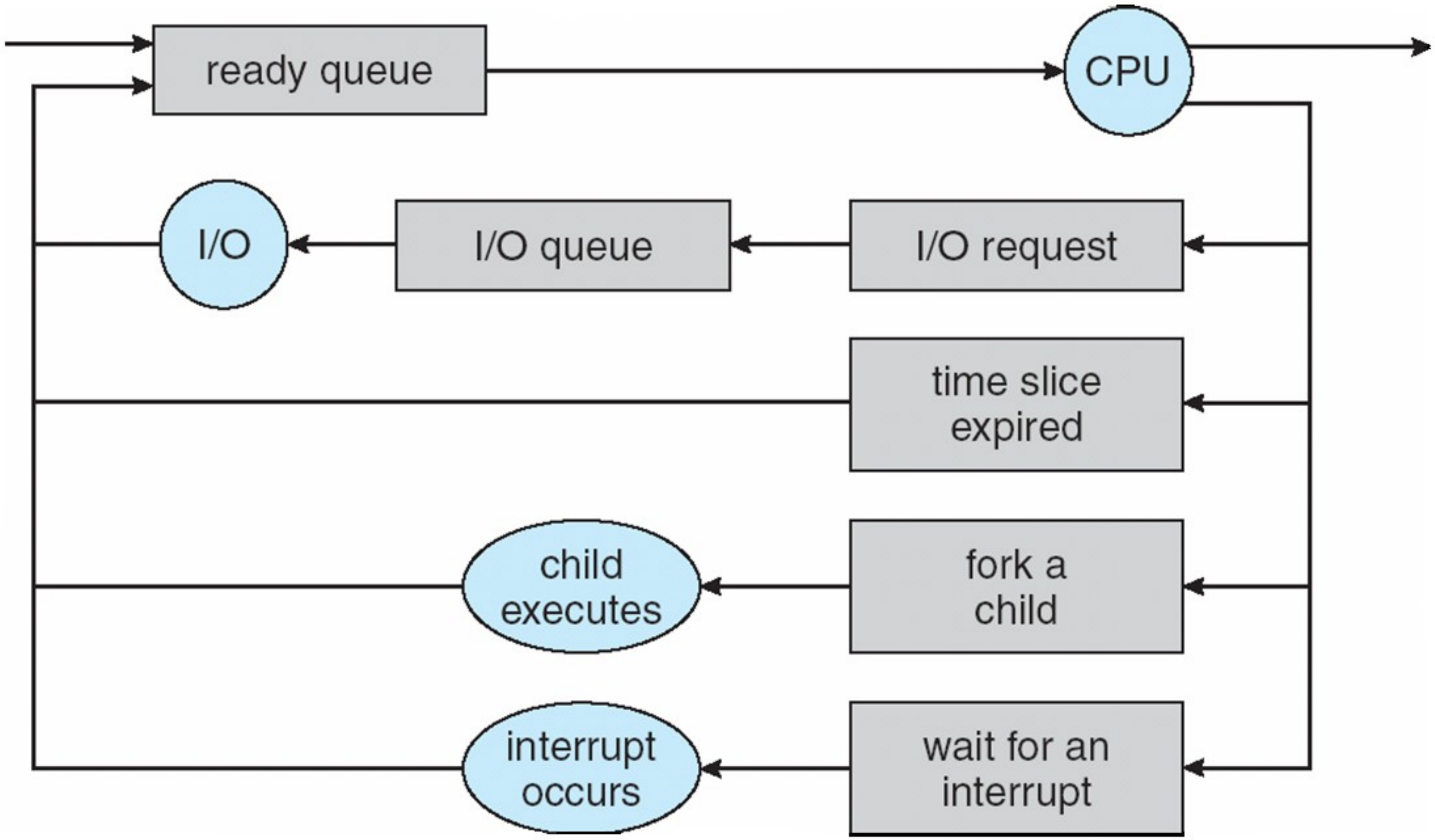
Ready Queue And Various I/O Device Queues



modified by Stewart Weiss

Representation of Process Scheduling

- A queuing theory model can be used to analyze and optimize operating system design.



modified by Stewart Weiss

Process Characterization

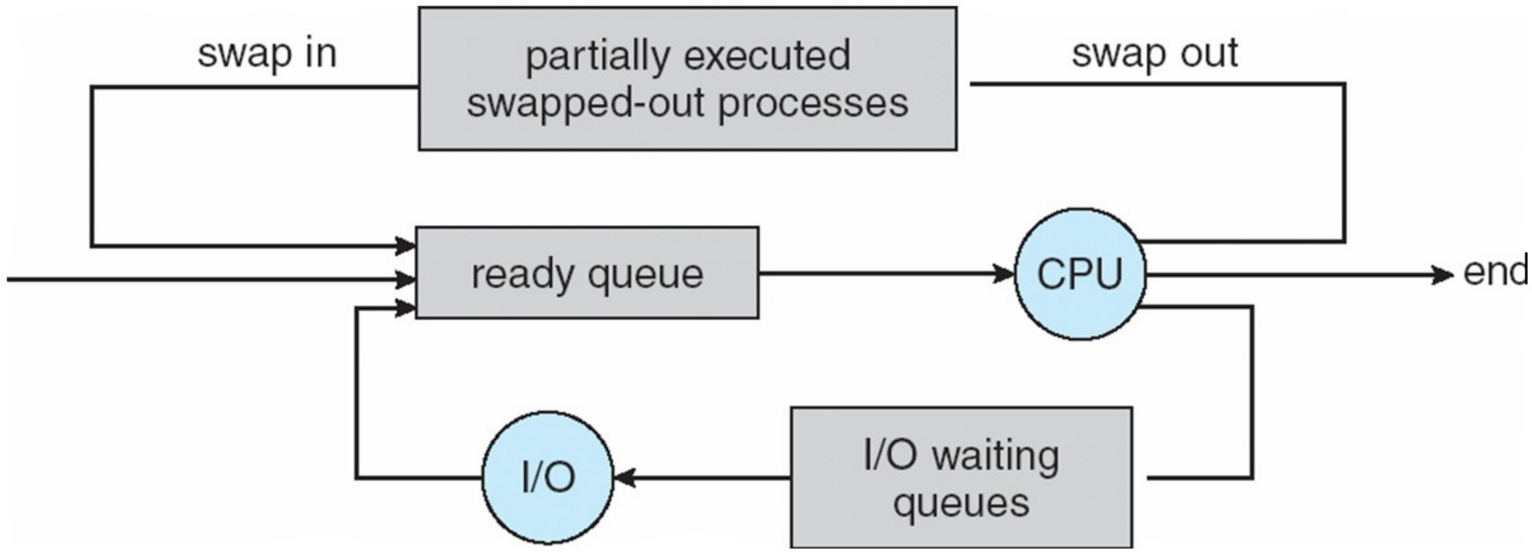
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts – runs on CPU and quickly asks for I/O
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts, so it will not remove itself from CPU very often.

Schedulers

- **Long-term scheduler** (or job scheduler) – in batch systems only; controls process mix (I/O-bound versus CPU-bound) to maximize resource utilization.
- Medium-term scheduler - selects which processes should be brought into the ready queue ***(in UNIX this was the swapper – chose which processes to swap in and out of memory) ; controls degree of multi-programming***
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU; must be very fast because it may run 10 or more times per second

modified by Stewart Weiss

Medium Term Scheduling



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Process Creation

- **Parent** process create **child** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- New process needs resources. Where do they come from?
- Resource sharing choices:
  - Parent shares all of its resources with child
  - Children share subset of parent's resources (UNIX)
  - Parent and child share no resources (Windows)
- Execution choices: after child is created:
  - Parent and children execute concurrently (UNIX)
  - Parent waits until children terminate

modified by Stewart Weiss

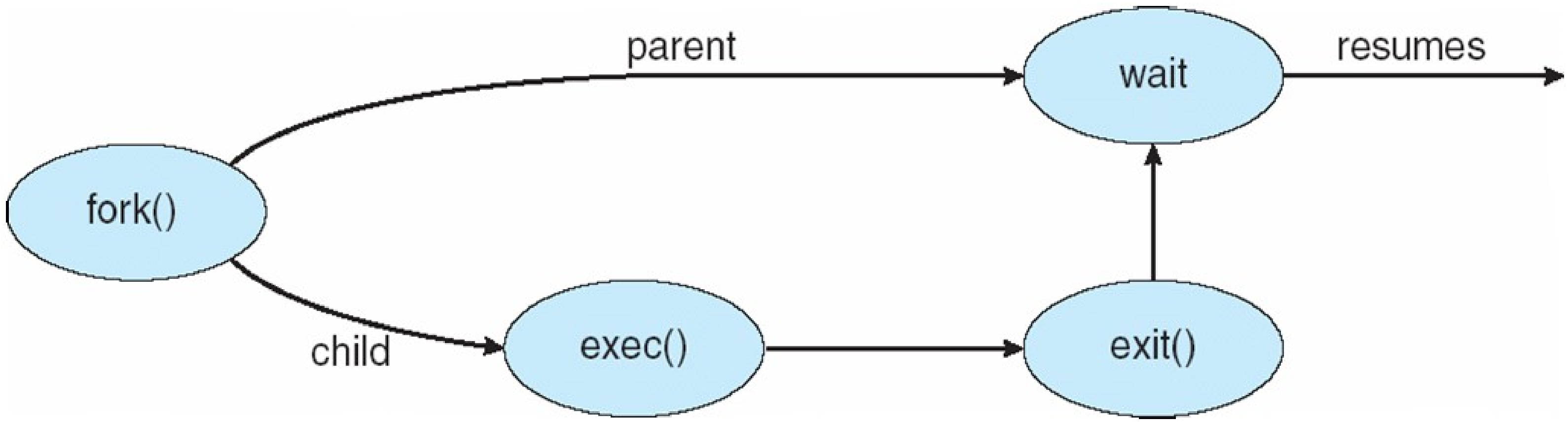
Process Creation (Cont)

- Address space
  - Child gets duplicate of parent (UNIX)
  - Child has a program loaded into it (Windows)
- UNIX examples
  - **fork** system call creates new process (example coming)
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

modified by Stewart Weiss

Process Creation in UNIX

- Typical parent-child execution (like shell in UNIX – parent creates child then blocks itself until child calls exit(). Child replces its program with a new one, runs and then calls exit().



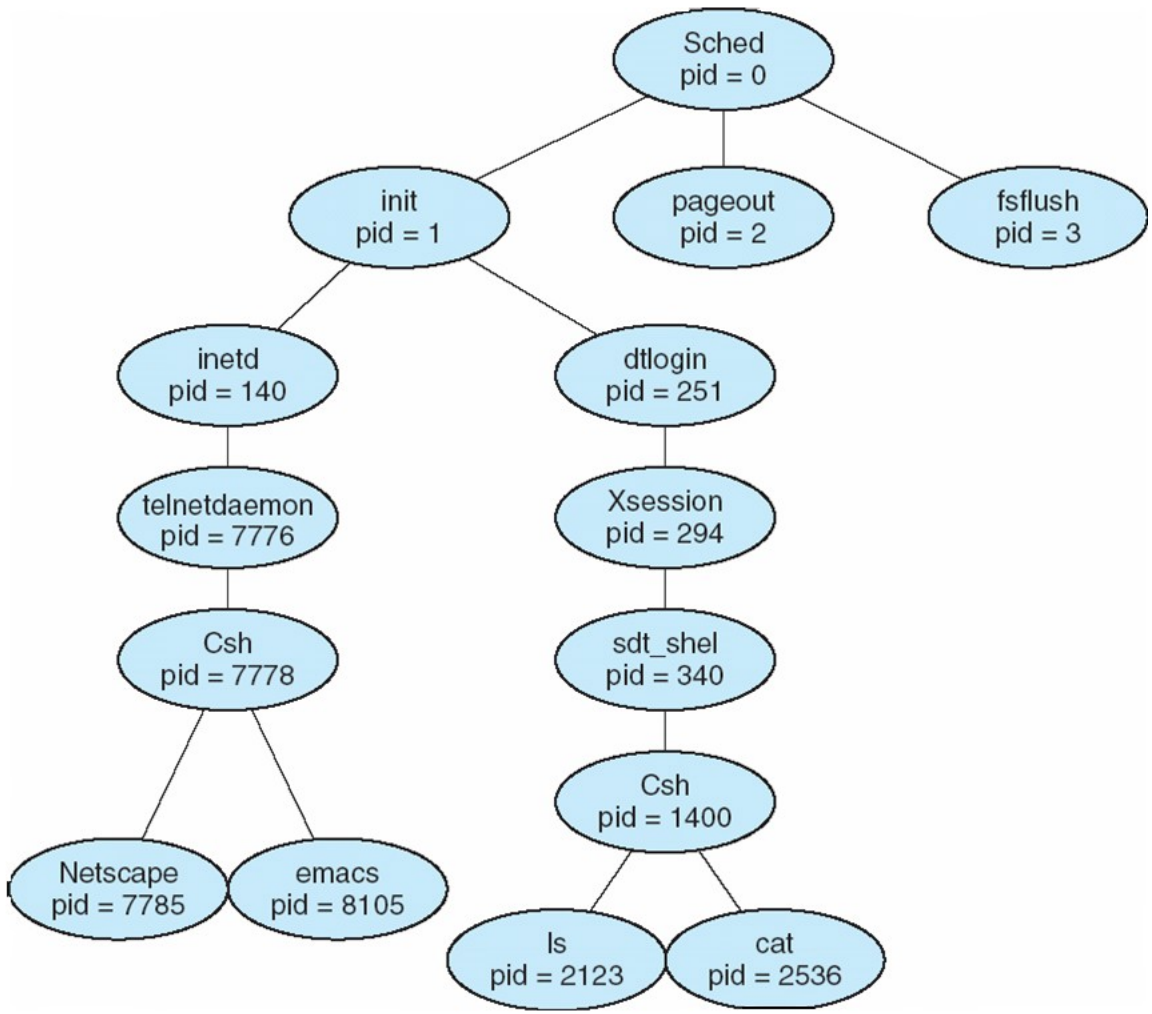
C Program Forking Separate Process

- Code that does what previous slide depicted:

```
int main()
{
pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
              fprintf(stderr, "Fork Failed");
              exit(-1);
     }
     else if (pid == 0) { /* child process */
              execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
              /* parent will wait for the child to complete */
              wait (NULL);
              printf ("Child Complete");
              exit(0);
     }
}
```

modified by Stewart Weiss

A tree of processes on a typical Solaris



modified by Stewart Weiss

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Deliver some data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes if (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates ***(UNIX does – children continue to run, are adopted by init)***
      - All children terminated - **cascading termination**

modified by Stewart Weiss

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- **Definition:** A process is **Independent** if it cannot affect or be affected by the execution of another process
- **Definition.** A process is **Cooperating** if it can affect or be affected by the execution of another process
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

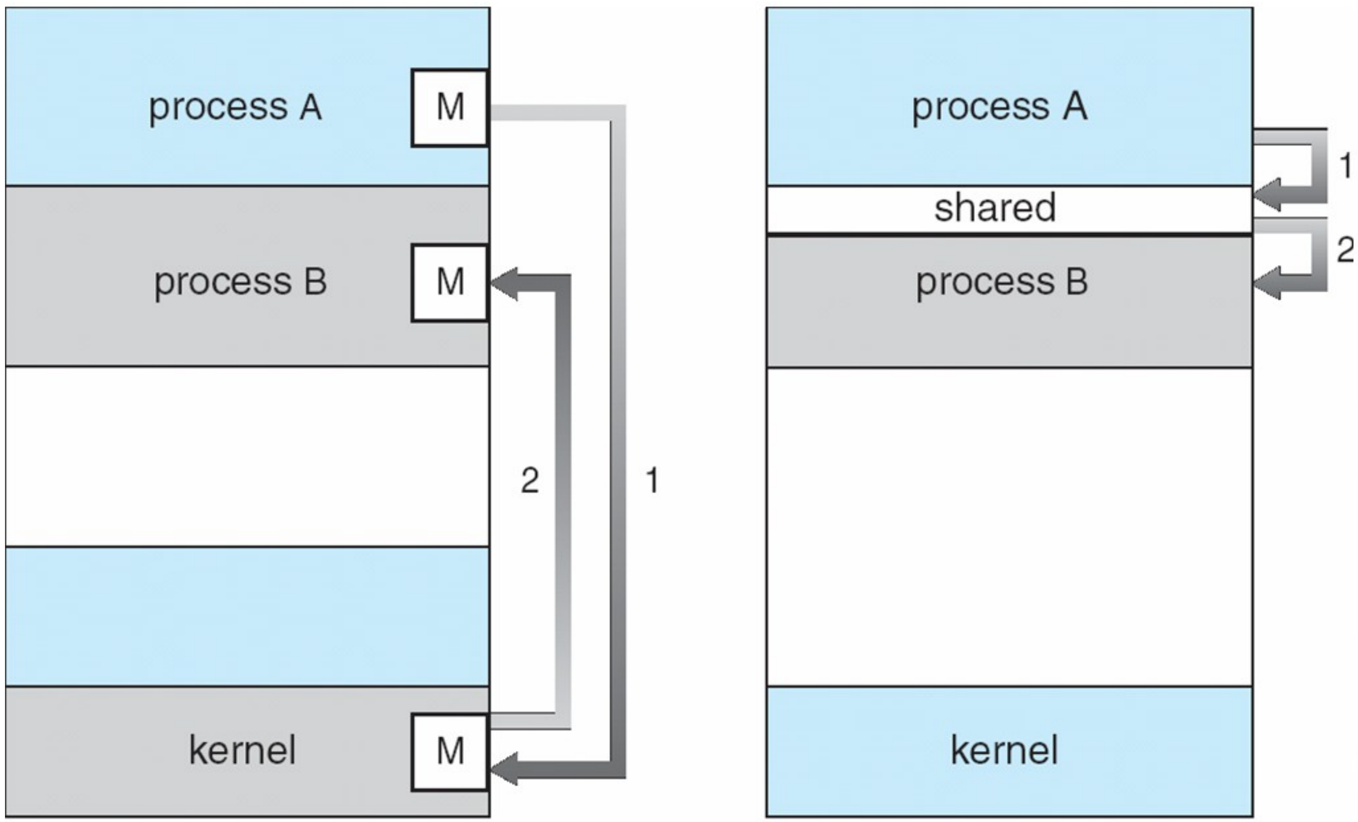
modified by Stewart Weiss

Interprocess Communication

- If processes cooperate they need a method of **interprocess communication** (**IPC**) and the ability to **synchronize** (chapter 6)
- Two models of IPC
  - Shared memory – e.g., common variables or files (think buffer)
  - Message passing – messages sent between processes

modified by Stewart Weiss

Communications Models



(a)
Message Passing

(b)
Shared Memory

modified by Stewart Weiss

Producer-Consumer Problem

- A classical example of a cooperating process problem
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process – use a **buffer** for transfer of data
- Producer puts item into buffer; consumer removes item when it is ready.
  - what if no buffer?
  - what if small buffer?
  - what if unbounded buffer?

modified by Stewart Weiss

Producer-Consumer Problem

- Examples
  - producer -- printing program, consumer – print driver
  - producer – compiler, consumer – assembler
  - UNIX pipe : last | sort
- Two versions:
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

modified by Stewart Weiss

Bounded-Buffer – Shared-Memory Solution

- Shared data, initialized as follows

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;      // index to put next item
int out = 0;     // index from which to get next item
```

- empty when in ==out, full when out == (in+1)% BUFFER_SIZE
  - can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

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

Bounded Buffer – Consumer

```
while (true) {
       while (in == out)
               ; // do nothing -- nothing to   consume

      // remove an item from the buffer
      itemToConsume = buffer[out];
      out = (out + 1) % BUFFER SIZE;
 process itemToConsume;
 }
```

modified by Stewart Weiss

Producer Consumer -- Comments

- The buffer is a circular queue; in and out both wrap around,
- One cell is always empty
- Only producer changes in
- Only consumer changes out
- Is it correct? Can producer and consumer ever try to work on same cell – i.e. as producer is filling cell, consumer is trying to empty it?
- What if there are multiple consumers?

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions without resorting to shared variables
- Necessary when processes cannot access the same memory – e.g. distributed environments
- Message passing facility provides two primitives:
  - **send**(*message*) – message size can be fixed or variable
  - **receive**(*message*)
- 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)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link? I.e., how big a buffer?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
- Direct or indirect naming?
- Symmetric or asymmetric communication?
- Automatic or explicit buffering?

modified by Stewart Weiss

Direct Communication

- Processes name each other explicitly; names are bound at compile time
  - **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
- Because process must be identified at compile time , not very useful

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of 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

modified by Stewart Weiss

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- 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. Sender is notified who the receiver was.
- Linux implements with **`msgsnd()`** and **`msgrcv()`**, which are part of System V

modified by Stewart Weiss

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

modified by Stewart Weiss

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 queue is full; receiver waits if queue is empty
  3. Unbounded capacity – infinite length
     Sender never waits; receiver still waits if queue is empty

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR | S
   IWUSR);
```

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

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

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

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

```
shmdt(shared memory);
```

modified by Stewart Weiss

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

`msg_send(), msg_receive(), msg_rpc()`

  - Mailboxes needed for commuication, created via

`port_allocate()`

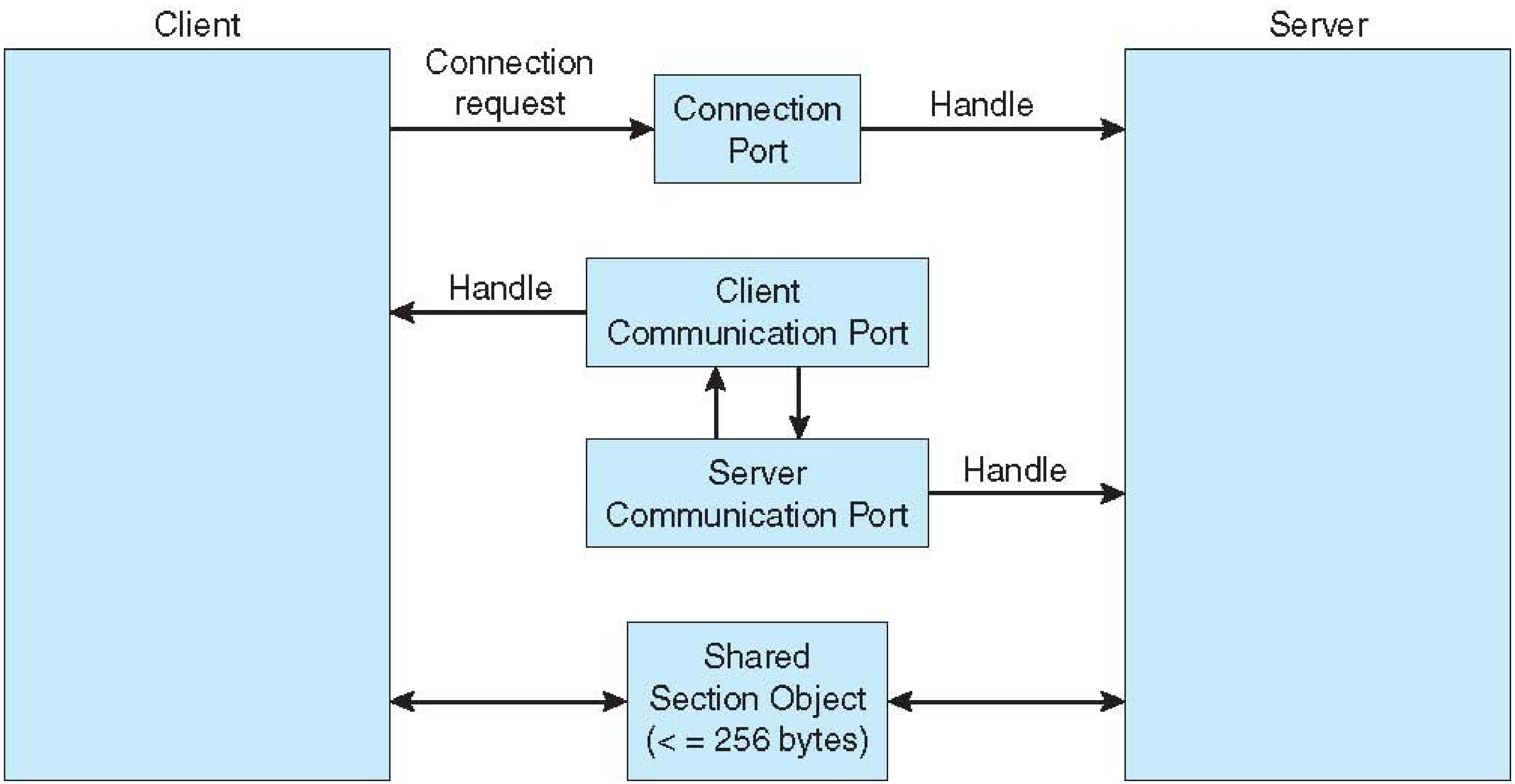
modified by Stewart Weiss

Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's connection port object
    - The client sends a connection request
    - The server creates two private communication ports and returns the handle to one of them to the client
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

modified by Stewart Weiss

Local Procedure Calls in Windows XP



modified by Stewart Weiss