CPSC 351 Operating System Concepts

Chapter 3: Processes

Chapter3-A

Chapter 3: Processes

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

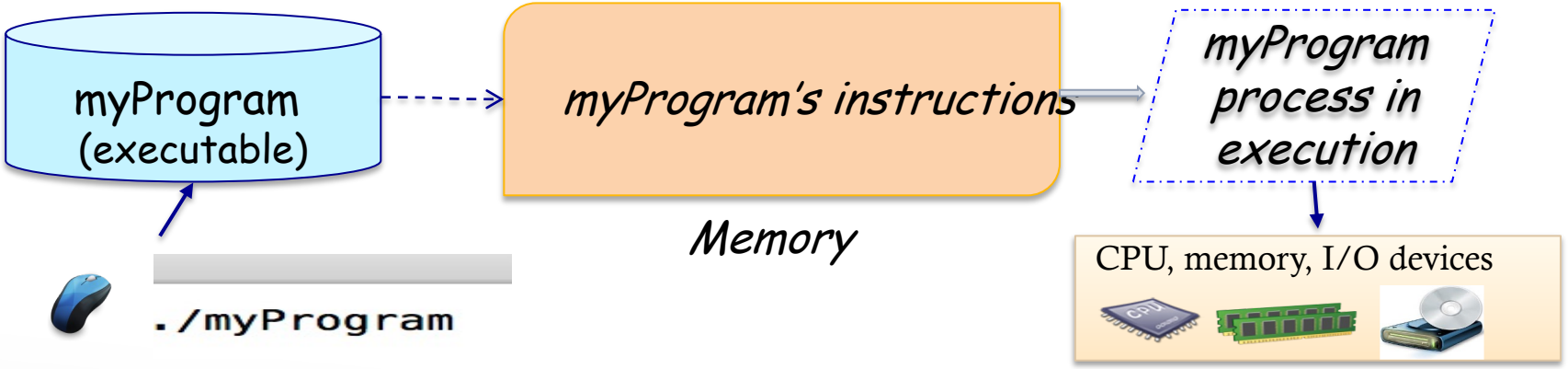
Objectives

- 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 explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

Process Review

A *process* can be defined as:

- A program in execution
- An instance of a running program
- The entity that can be assigned to, and executed on, a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources



Process Concept

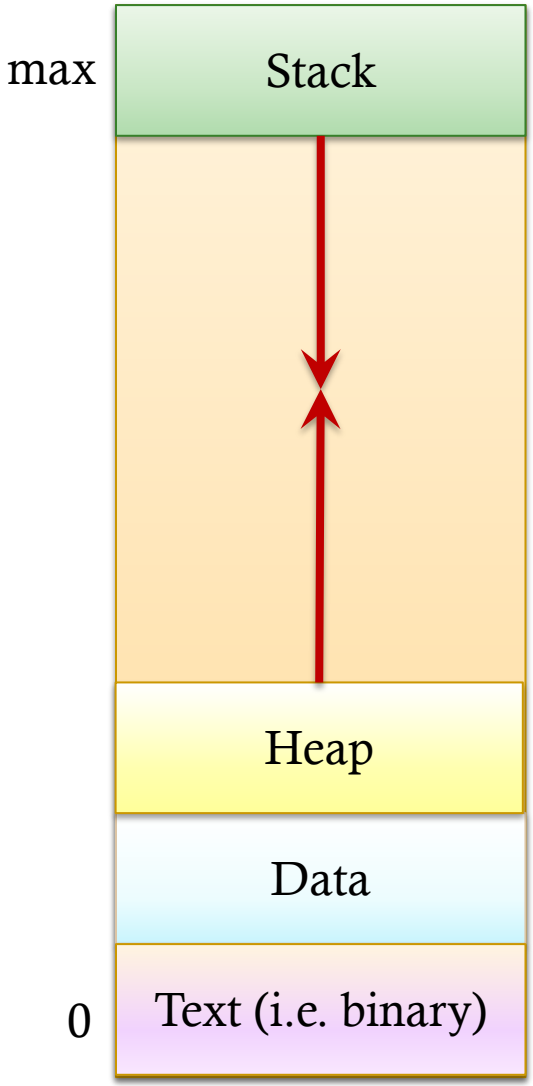
- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (executable file), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory

- A process is more than the program code
- Includes
  - Current activity such as program counter (PC specifying the next instruction to execute) and contents of processor registers
  - Stack: stores function parameters, local variables, and return address
  - Heap: contains dynamic memory allocated during process runtime
  - Data: contains global variables
  - Text: stores the program instructions (i.e., the executable)



Process in Memory Example

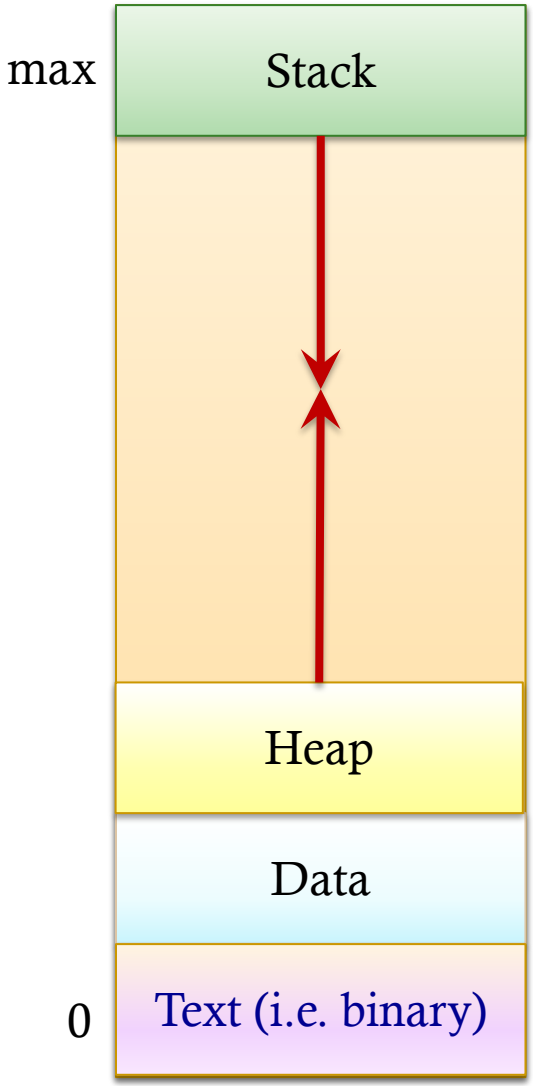
```
int a = 1;
int f(int c) { int b = c+1; return b;}
int main()
{       int d = 2;
        char* arr = new char[100];
        for(int i = 0; i < 100; ++i)
                arr[i] = '\0';
        f(d);
        delete arr;
        return 0;
}
```

Data

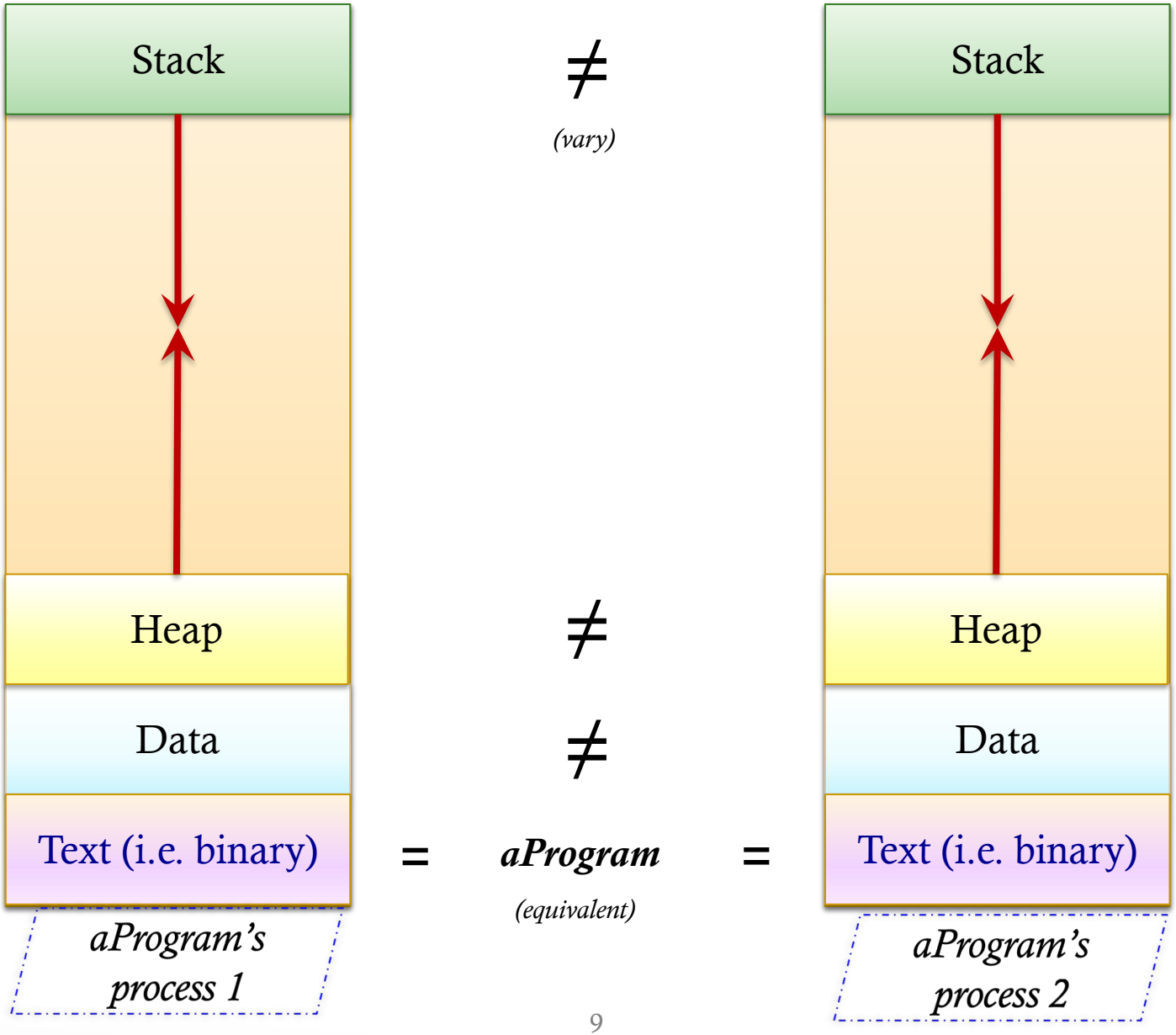
Heap

Stack

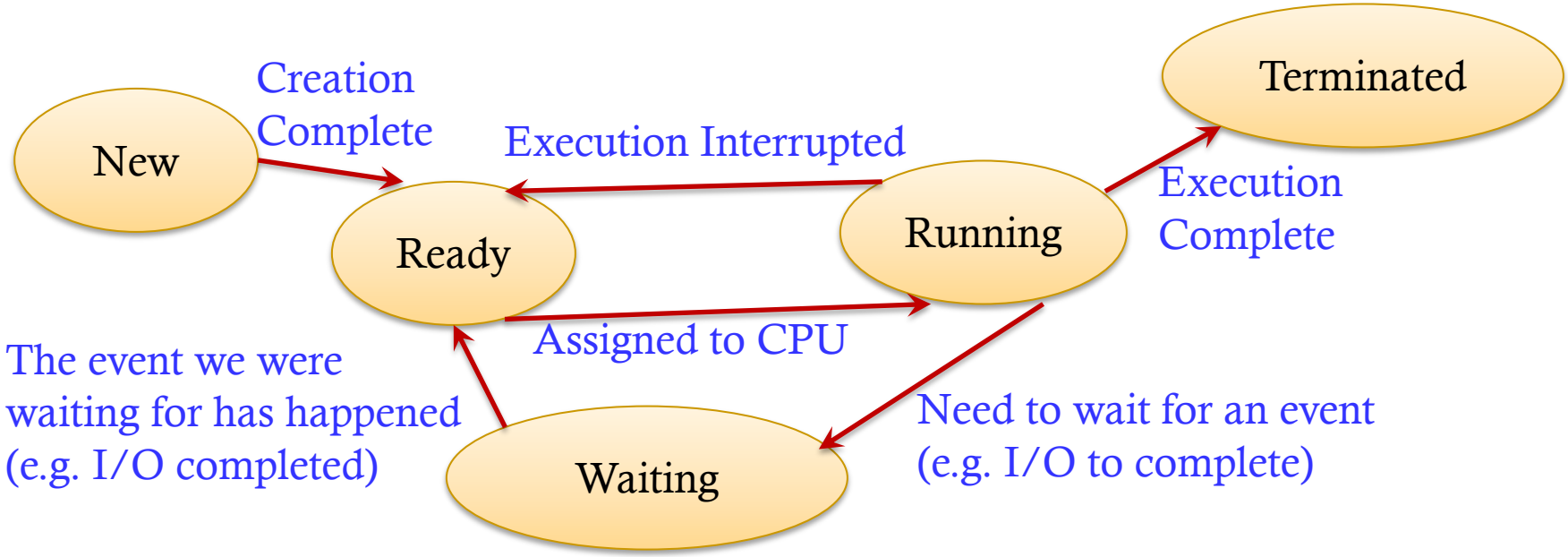
Text (i.e., executable instructions)



One Program Two Processes



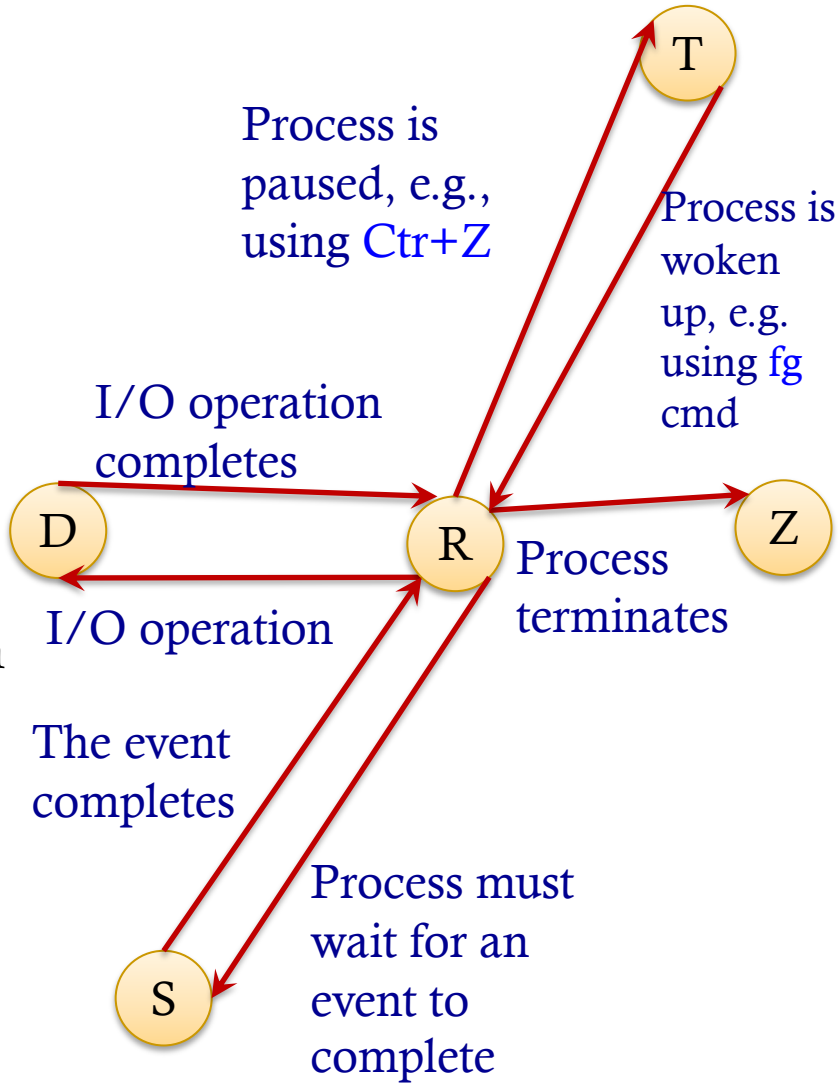
Process State



- The states of a process:
  - New: The process is being created
  - Ready: The process is waiting to be assigned to a processor
  - Running: Instructions are being executed
  - Waiting: The process is waiting for some event (e.g., I/O completion)
  - Terminated: The process has finished execution

Process States in Linux

- Use the ps utility to learn process state in Unix:
  - Example: ps –aux
  - Interpreting the output of ps
    - **R** the process is running or runnable (on run queue)
    - **D** uninterruptible sleep (usually I/O)
    - **S** interruptible sleep (waiting for an event to complete)
    - **Z** defunct/zombie, terminated but not reaped by its parent (discussed later)
    - **T** stopped, either by a job control signal or because it is being traced



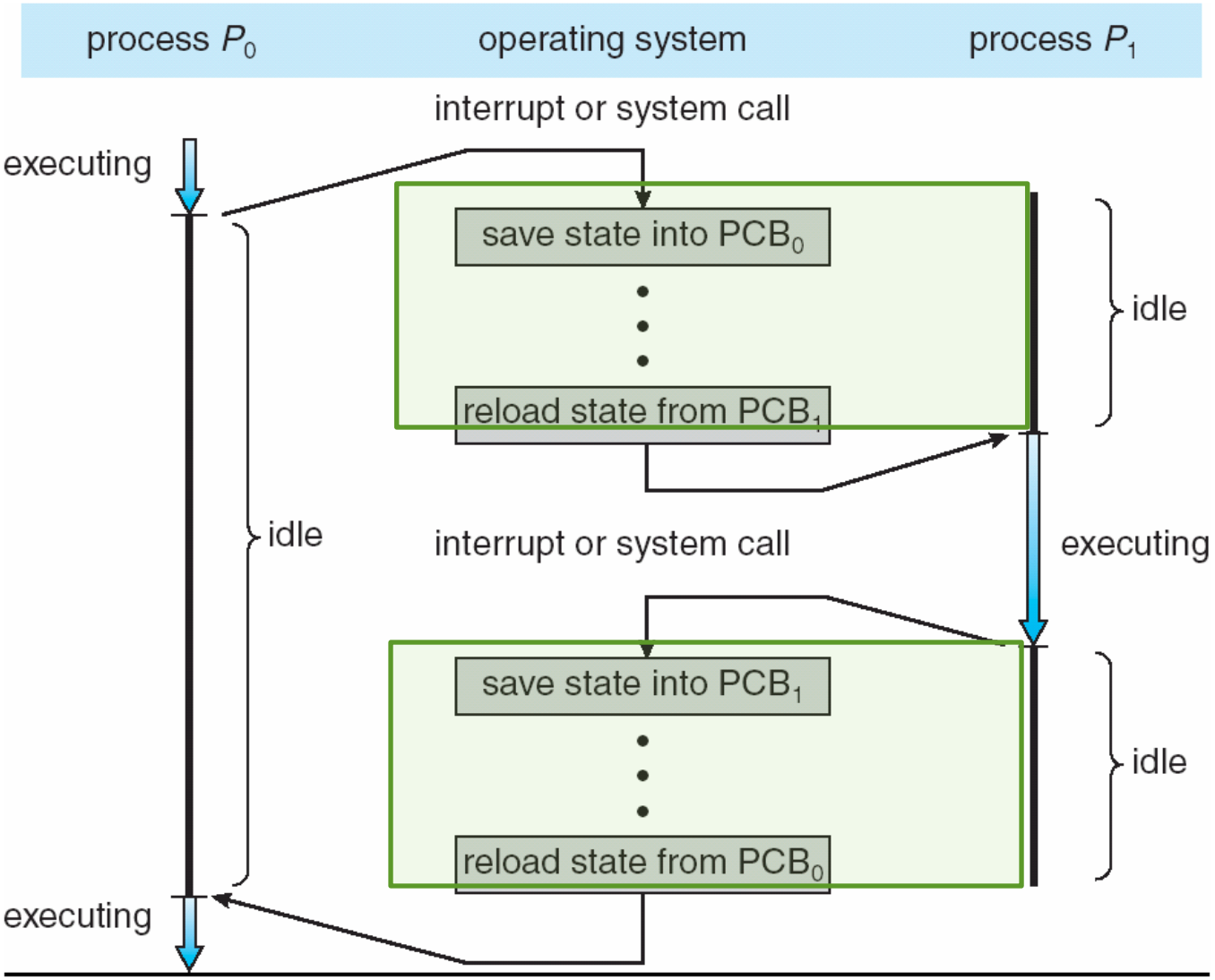
Process Control Block

- OS uses a Process Control Block (PCB) to represent each process
- PCB Components:
  - Process State: the state of the process, e.g., running, waiting, …
  - Process ID: a unique ID associated with the process, e.g., 1234567
  - Program Counter: the address of the next instruction to be executed
  - CPU Registers: the current values of the accumulators, stack pointers, etc…
  - CPU-Scheduling Information: process priority and other info needed for assigning the process to the CPU

Process Control Block (cont)

- OS uses a Process Control Block (PCB) to represent each process
  - Memory Management Information: info about memory allocated to the process
  - Accounting Information: CPU used, clock time elapsed since start (e.g., how long the process was running), time limits, …
  - I/O Status Information: list of I/O devices used by the process, list of open files, etc…

CPU Switch From Process to Process



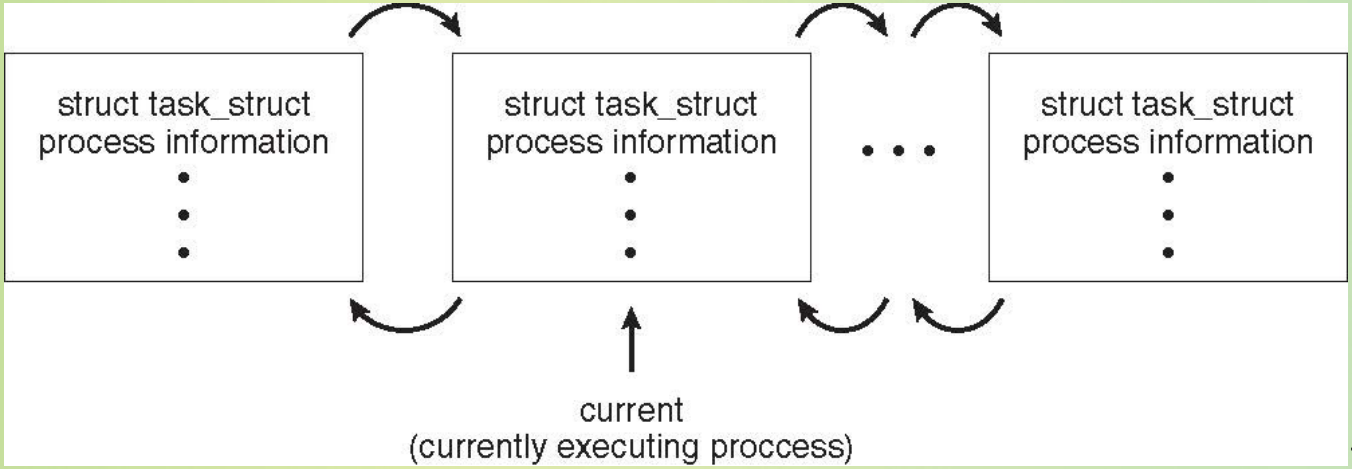
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Representation in Linux

Represented by the C structure `task_struct`

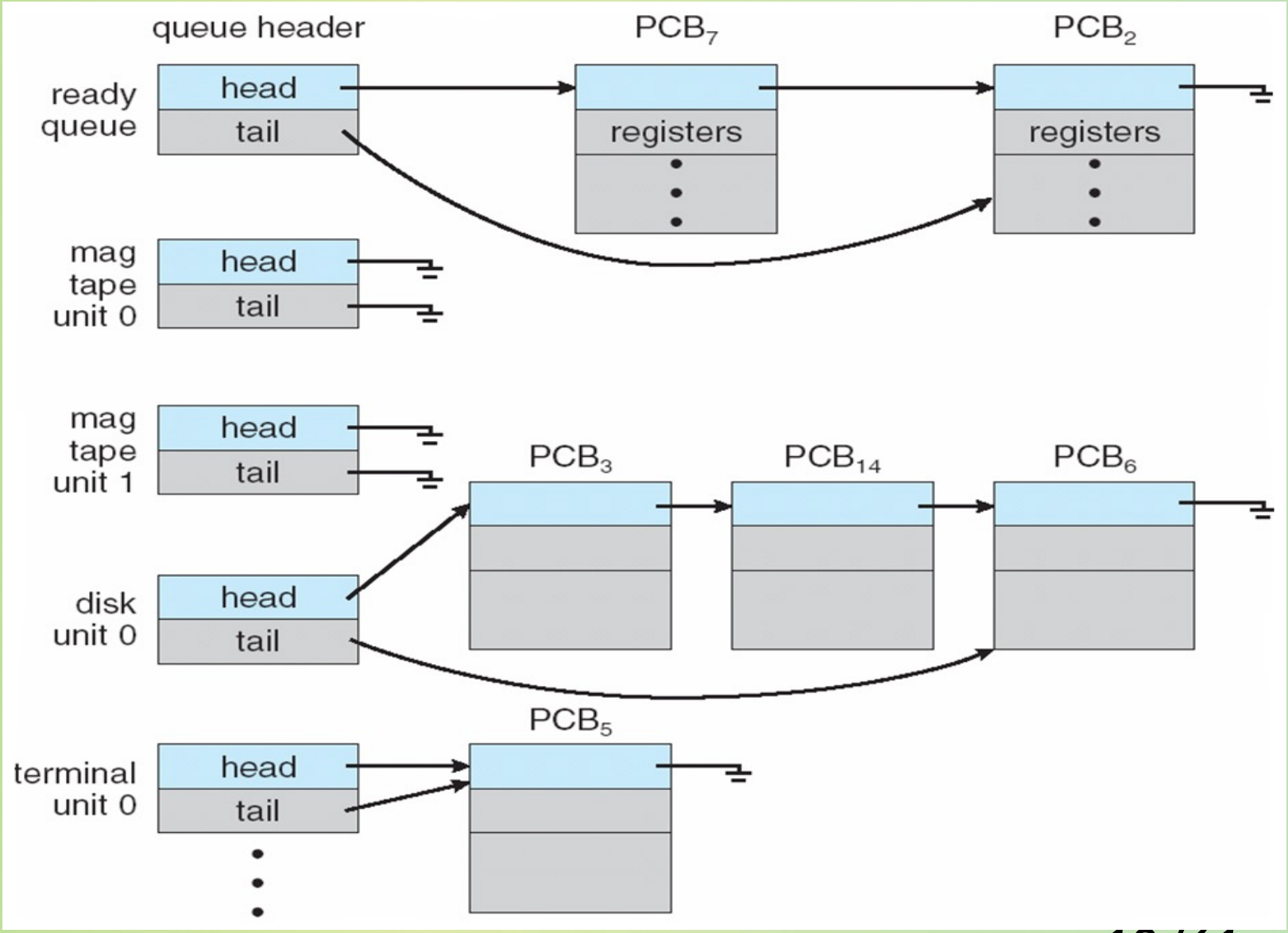
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

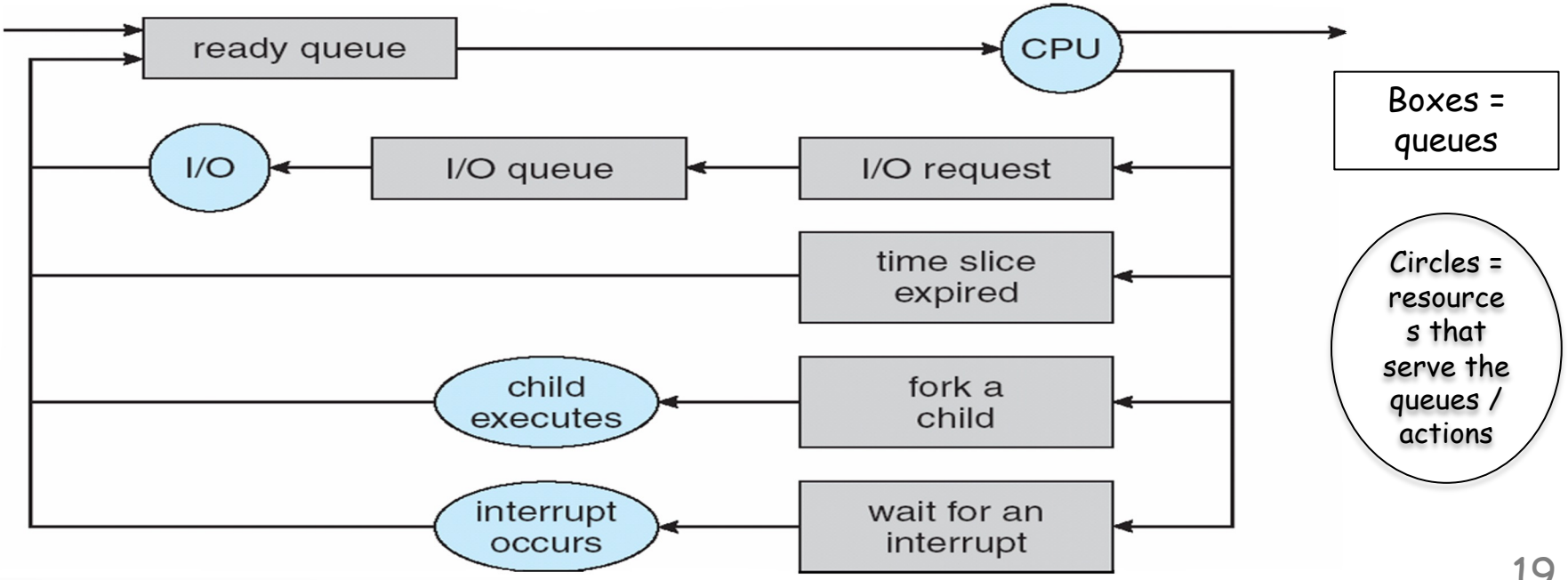
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing 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

Ready Queue And Various I/O Device Queues



Process Scheduling: The Queuing Mechanism in Action

1. New process is placed on a ready queue
2. The process is dispatched, i.e., selected for execution
3. During execution:
   - Process may be queued on the device queue due to an I/O request
   - Process can create a new subprocess and wait for its termination
   - Process can be interrupted and removed from the CPU
   - Process time slice (i.e., time allotted for its CPU use) expires:
     - Process is removed from the CPU
     - is placed on the ready queue until scheduled to run for another time slice



Schedulers

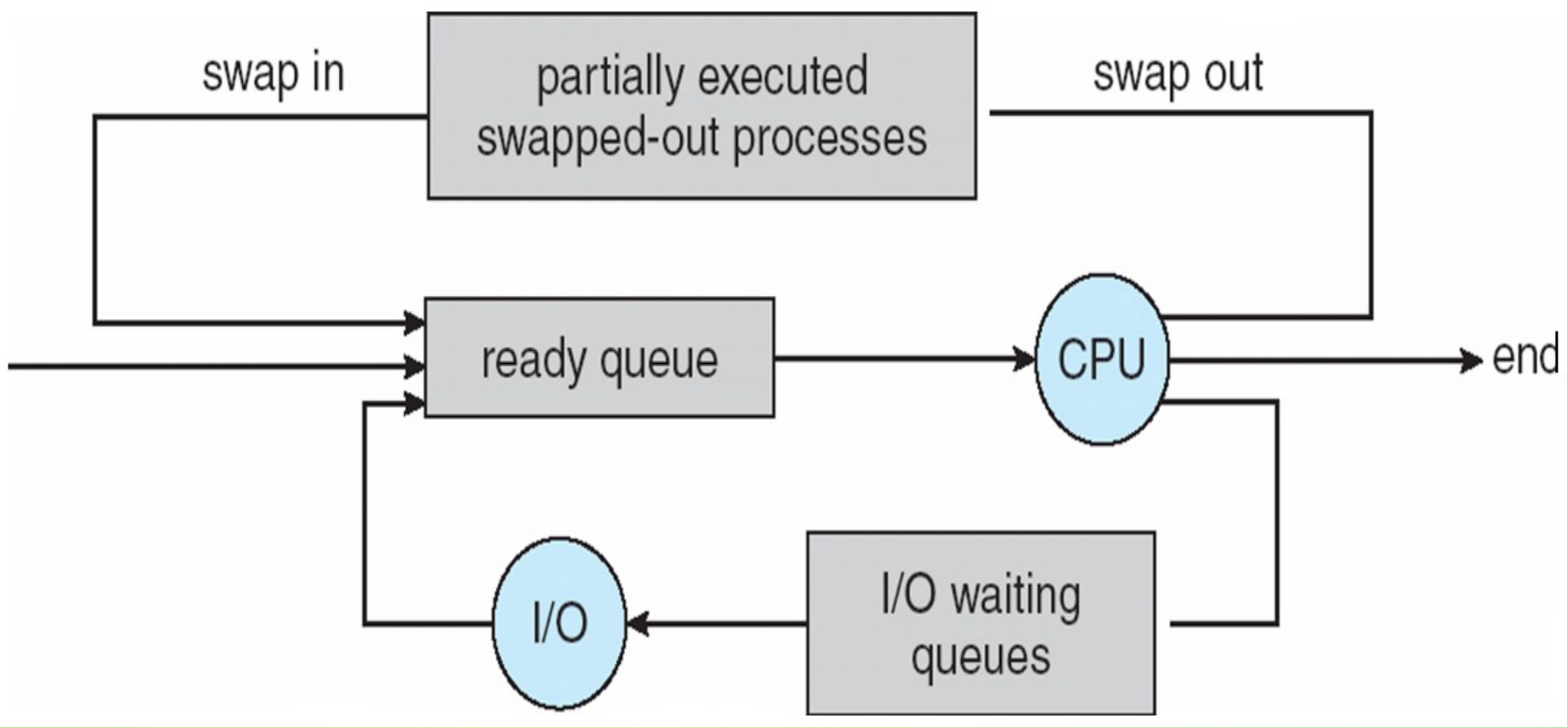
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Process Scheduling: Long-term Schedulers

- What if there are more processes than can fit into memory?
  - Solution: Temporarily store / spool some processes on a mass storage device, e.g., hard drive
- Long-term Scheduler selects spooled processes to load from the mass storage device into main memory
  - Executes less frequently than a short-term scheduler
  - Key idea: maximize resource utilization by selecting a mix of CPU bound and I/O bound processes
    - I/O bound processes are processes that spend more time doing I/O operations than CPU computations
    - CPU bound processes are processes that spend more time doing CPU computations than I/O operations
  - Rational:
    - If all selected processes are I/O-bound, then ready queue will always be empty i.e., nothing to execute on the CPU!
    - If all selected processes are CPU-bound, the device queues will be empty i.e., the devices go unused
  - ❖Not all systems such as UNIX and Windows have long-term scheduler

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



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
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single foreground process- controlled via user interface
  - Multiple background processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

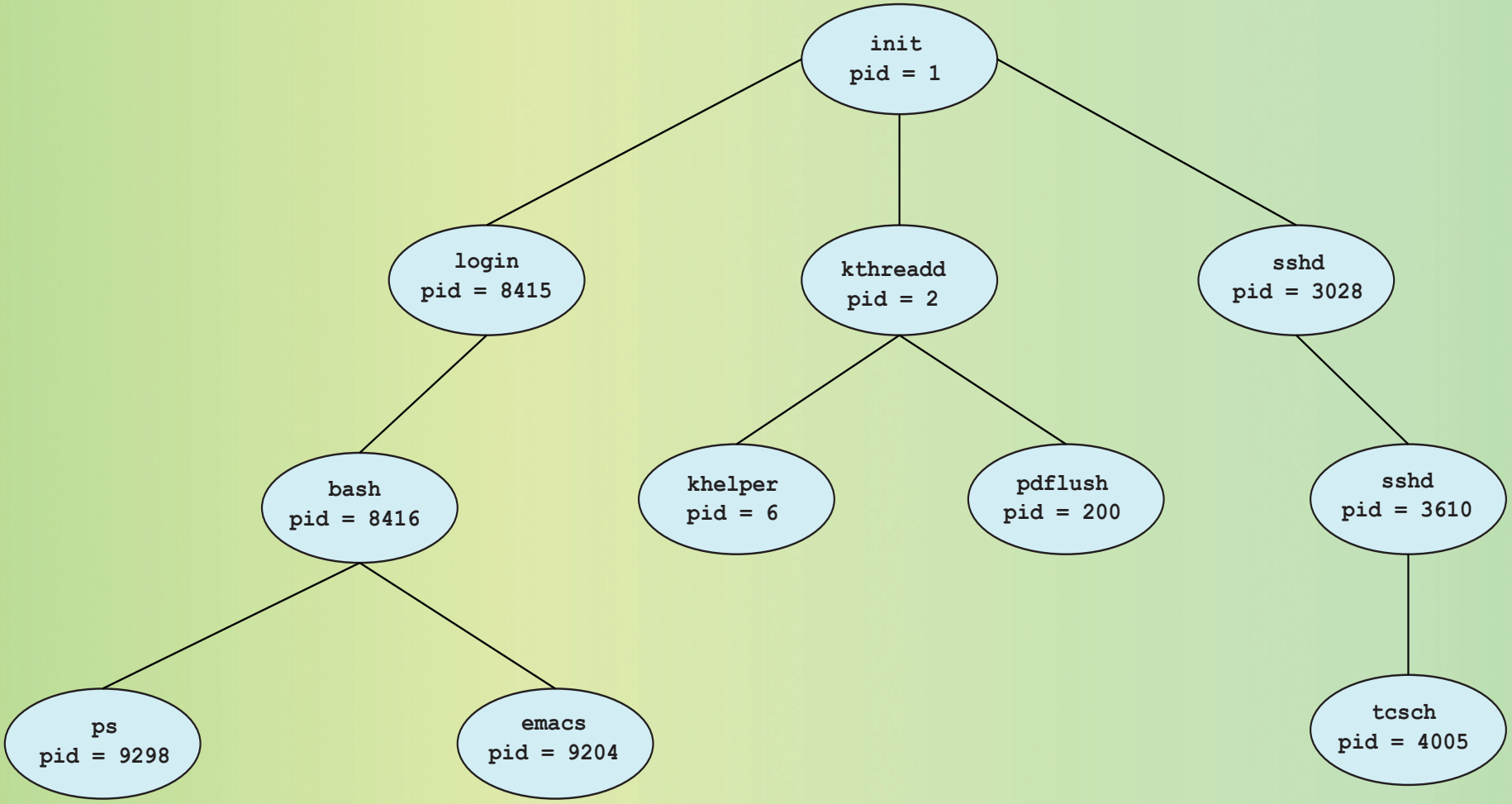
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

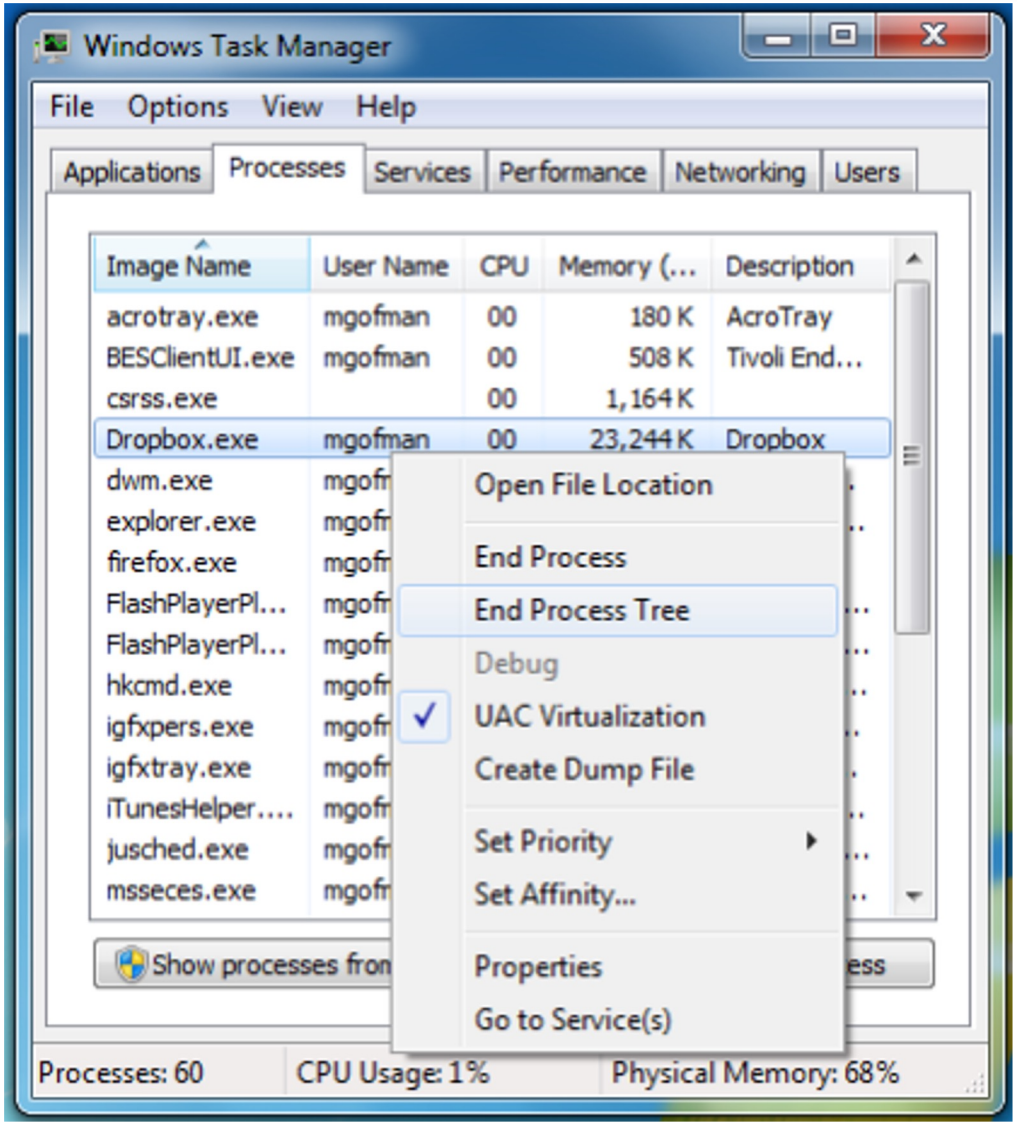
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux



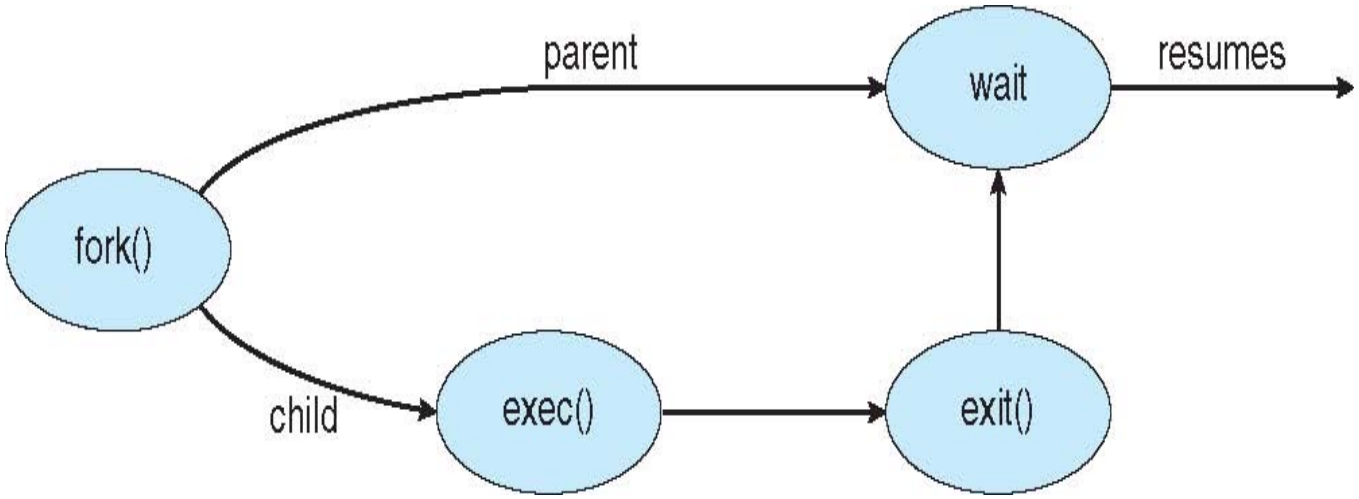
Operations on Processes: Windows Process Tree Example

- Windows: "End Process Tree" terminates the selected process and all its descendants



Unix Process Creation

- fork() system call creates new process
- exec() system call used after a fork() to replace the process' memory space with a new program



Process Creation in Unix/Linux: *fork()*

- *fork()* - a system call is issued by a parent process to create a child process
- Child process is a clone of a parent process
- Both parent and child continue execution at the instruction immediately after fork():
  - In the child fork() returns 0
  - In the parent fork returns process id (pid) of the child
  - fork() returns -1 on failure
- The child process inherits:
  - The set of files opened by the parent process
  - Other resources…

Process Creation in Unix/Linux: exec()/wait()/exit()

- *exec(…)* – replaces the program of the caller process with a new program
- *wait(…)* – waits until the child terminates
- *exit(int exitcode)* – terminates the caller process with the specified exit code

Process Creation in Unix/Linux: exec() variants

- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg,..., char * const envp[]);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);
- int execvpe(const char *file, char *const argv[], char *const envp[]);
- Example: execlp(const char *file, const char *arg, ...);
  - file: the path of the executable image
  - arg0…argn: command line arguments to pass to the process
- All return -1 on failure

Process Creation in Unix/Linux: wait() variants

- wait() and waitpid() return the process id of the child
  - pid_t wait(int *status);
  - pid_t waitpid(pid_t pid, int *status, int options);

- waitid() return 0 on success and -1 on faliure
  - int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

Process Creation in Unix/Linux: Putting it all Together

```
// Parent process
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);
    }
}
```

Start with the parent process

Parent process issues a folk() system call

fork() clones the parent process ➔ Both parent & child continue by executing the next instruction after fork()

```
// Parent process
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);
    }
}
```

```
// Child process
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);
    }
}
```

In parent, fork() returns child's process ID

```
// Parent process
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);
    }
}
```

In child, fork() returns pid of 0

```
// Child process
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);
    }
}
```

Parent issues a wait(…) system call to wait until the child terminates

```
// Parent process
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);
    }
}
```

Child issues a execlp() system call to replace its executable image with that of ls command

```
// Child process
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);
    }
}
```

Parent waits (in wait(…)) for the child process to terminate

```
// Parent process
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);
    }
}
```

ls command executes starting from the first instruction; original child code is destroyed

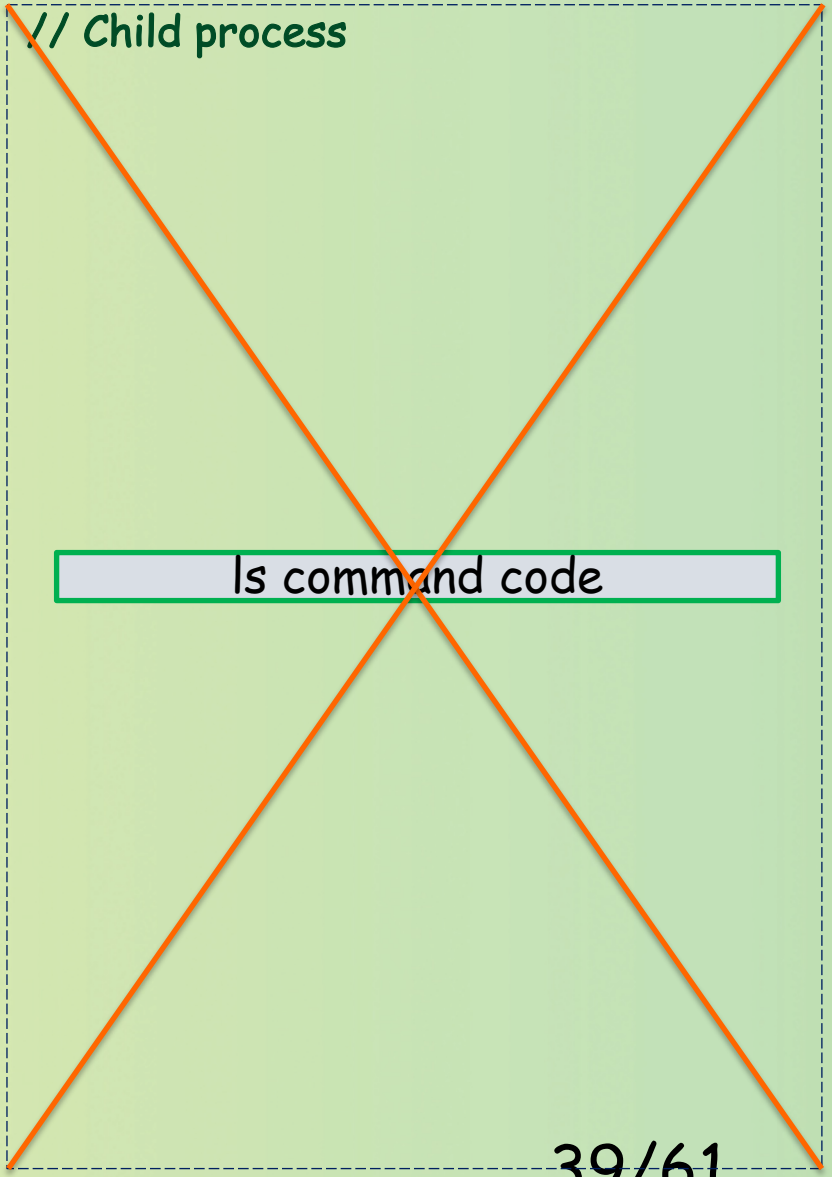
```
// Child process
```

ls command code

wait(…) returns, and parent process executes the next instruction

```
// Parent process
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);
    }
}
```

ls command finishes execution and terminates



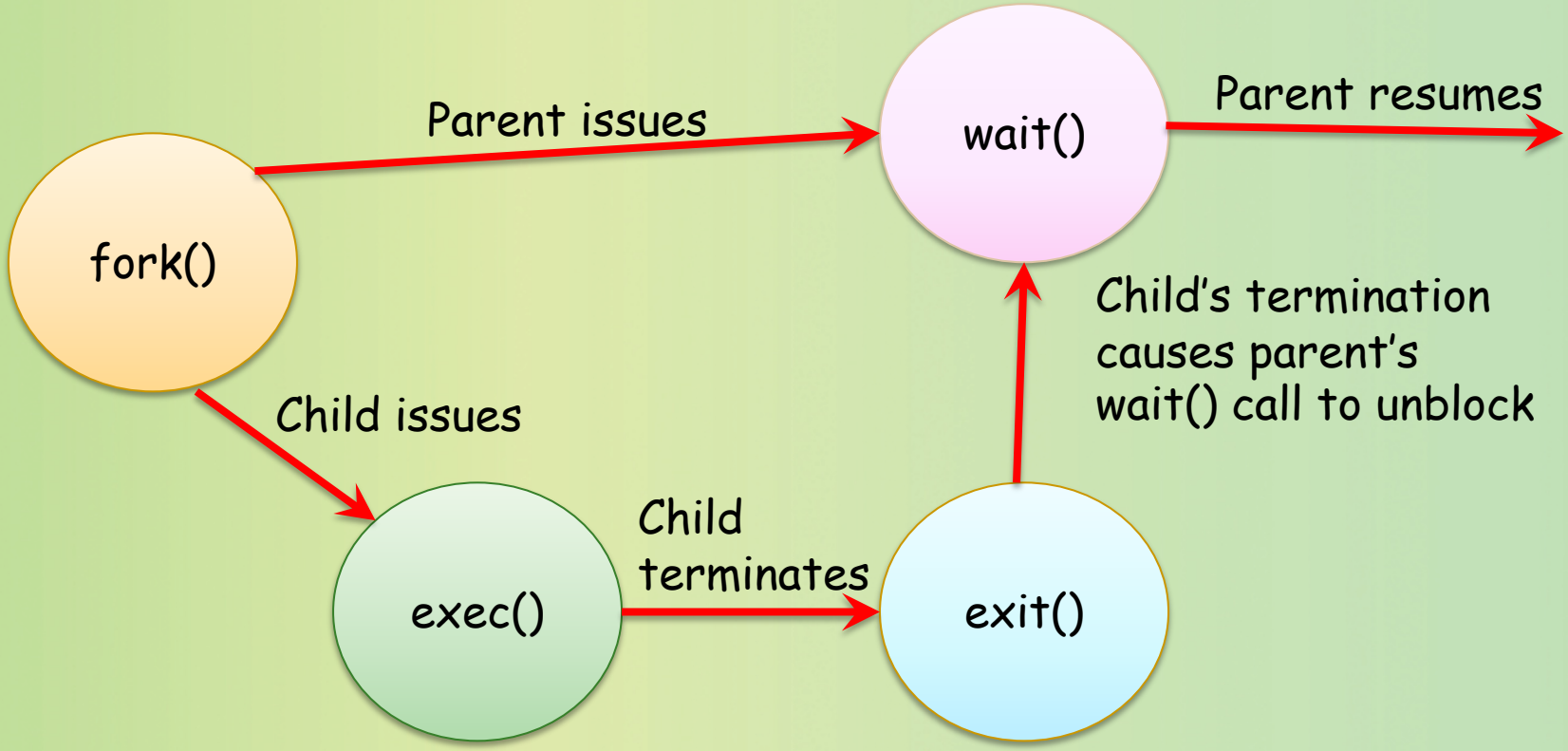
Parent process issues an exit() system call in order to self-terminate

```
/* parent process
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);
    }
}
```

Parent process terminates

```
// Parent process
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);
    }
}
```

Process Creation in Unix/Linux: Summary of fork()/exec()/wait()



Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke `wait()`) process is a zombie
- If parent terminated without invoking `wait`, process is an orphan

Zombie Process

- If a parent forks a child, but does not issue a wait() after the child terminates, the terminated child becomes a zombie process
- Zombie process – a terminated process whose PCB was not deallocated - i.e., PCB contains child's exit code such as the code returned by int main()
  - The child will remain a zombie until the parent calls wait()
    - Child's pid and PCB are released
- Child's exit code may be useful to the parent – e.g., to see whether the child has exited with an error

Orphan Process

- What if the parent process terminates and did not call *wait()* on the child?
  - The child becomes an orphan process
  - init process becomes the new parent of the orphaned children
  - init periodically calls wait() to collect the return statuses of orphans and release the orphan's process pid and PCB

Multiprocess Architecture – Chrome Browser

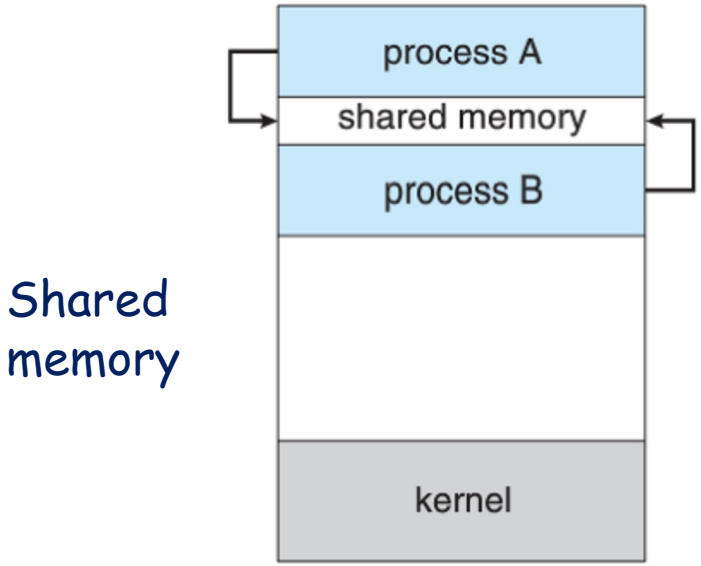
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in process for each type of plug-in

Interprocess Communication

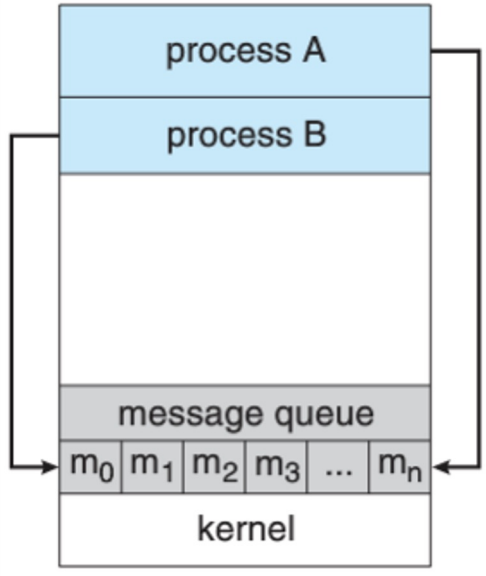
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need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

IPC Mechanisms for Cooperating Processes

- Shared memory: cooperating processes exchange information by reading/writing data from/to a region of shared memory
- Message passing: cooperating processes exchange messages



Shared memory



Message Passing

IPC: Shared Memory vs Message Passing

- Shared memory
  - Faster than message passing: only requires intervention from the OS to establish a shared memory region
  - Good for large transfers of information
  - Disadvantage: requires process synchronization to ensure that no two processes write the same memory location at the same time
- Message passing
  - No need for synchronization
  - Good for small information transfers
  - Easier to implement than shared memory
  - Disadvantage: usually requires OS intervention on every message transfer
    - Can be slower than shared memory

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;
```

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
    while (in == out)
    ; /* do nothing */
    next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   /* consume the item in next consumed
   */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

- Implementation issues:
  - 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?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **`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

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

Indirect Communication

- Operations
  - create a new mailbox (port)
  - 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*

Example: 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.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
   /* produce an item in next produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
  receive(next_consumed);

  /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways

1. Zero capacity – no messages are queued on a link.
   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

Address Space

- Process must be able to reference every resource in its abstract machine
- Assign each unit of resource an address
  - Most addresses are for memory locations
  - Abstract device registers
  - Mechanisms to manipulate resources
- Addresses used by one process are inaccessible to other processes
- Say that each process has its own *address space*

Shared Address Space

- Classic processes sharing program ⇒ shared address space support
- Thread model simplifies the problem
  - All threads in a process implicitly use that process's address space , but no "unrelated threads" have access to the address space
  - Now trivial for threads to share a program and data
    - If you want sharing, encode your work as threads in a process
    - If you do not want sharing, place threads in separate processes

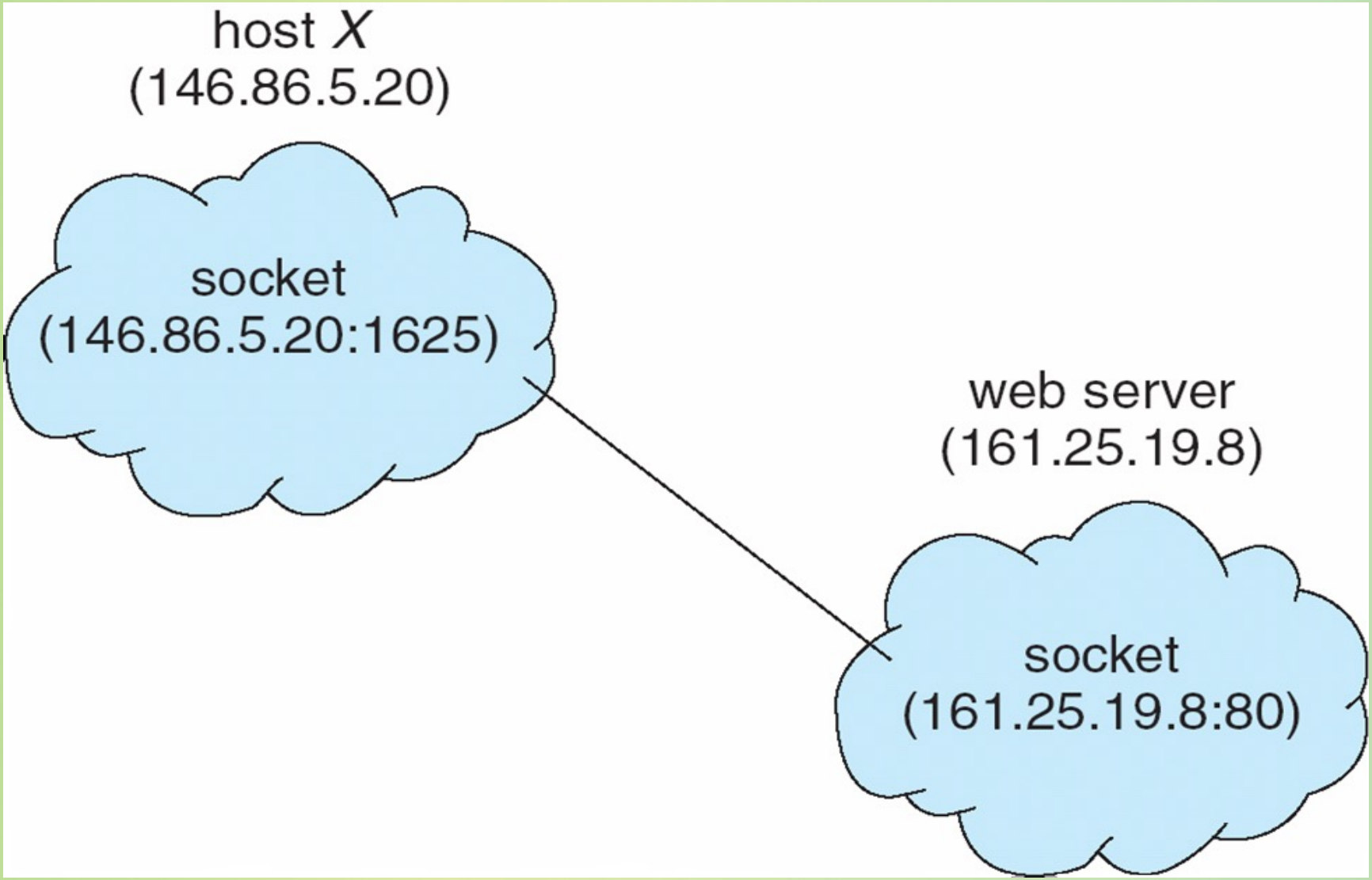
Communications in Client-Server Systems

- Sockets—details are covered in the class of *Computer Communications*
- Remote Procedure Calls
- Pipes

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

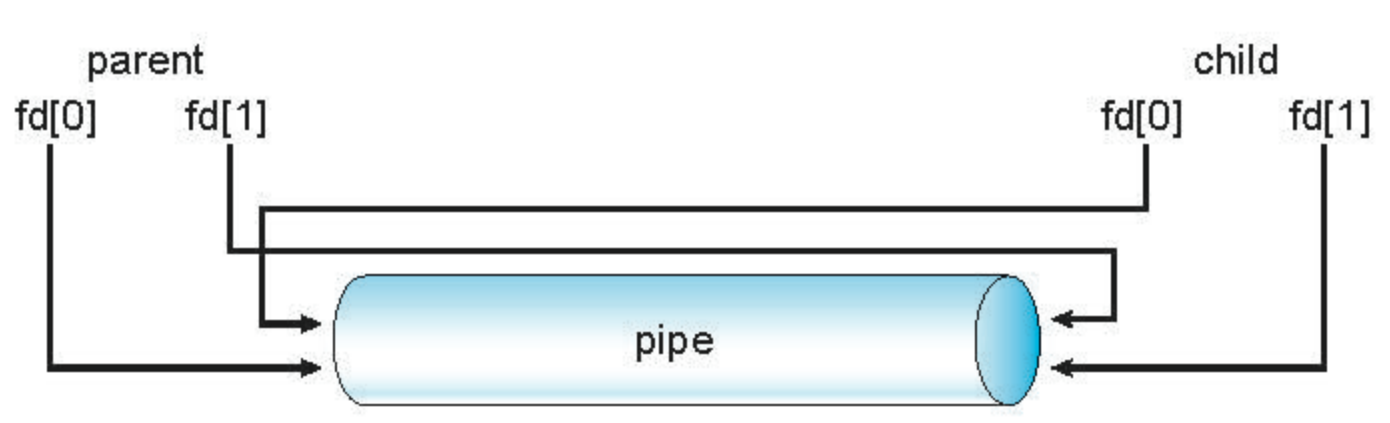
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., *parent-child*) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems