Operating Systems
[3. Process]

Chung-Wei Lin
cwlin@csie.ntu.edu.tw
CSIE Department
National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations
- Describe and contrast interprocess communication using shared memory and message passing
- Design programs that use pipes and POSIX shared memory to perform interprocess communication
- Describe client-server communication using sockets and remote procedure calls

Outline

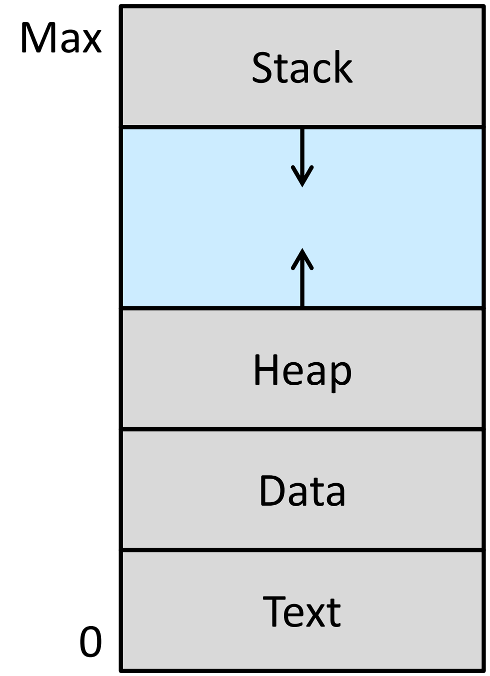
- **Process Concept**
  - The Process, Process State, Process Control Block, Threads
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept

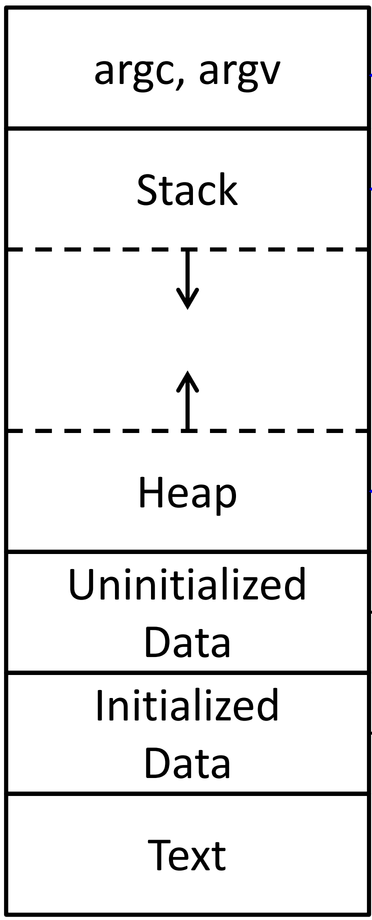
- **Process**: a program in execution
  - A program is a **passive** entity
    - A file (executable file) containing a list of instructions stored on disk
  - A process is an **active** entity with
    - A program counter specifying the next instruction to execute
    - A set of associated resources
  - A program becomes a process when an executable file is loaded into memory
    - Double-click an icon representing the executable file
    - Enter the name of the executable file on the command line
  - Although two processes may be associated with the same program, each of these is a separate process
    - It is also common to have a process that spawns many processes as it runs

Memory Layout of a Process

- Text section
  - The executable code
- Data section
  - Global variables
- Heap section
  - Dynamically allocated during program run time
- Stack section
  - Temporary data storage when invoking functions
  - Examples: function parameters, return addresses, local variables
- The stack and heap sections can shrink and grow dynamically during program execution
  - The operating system must ensure they do not overlap one another



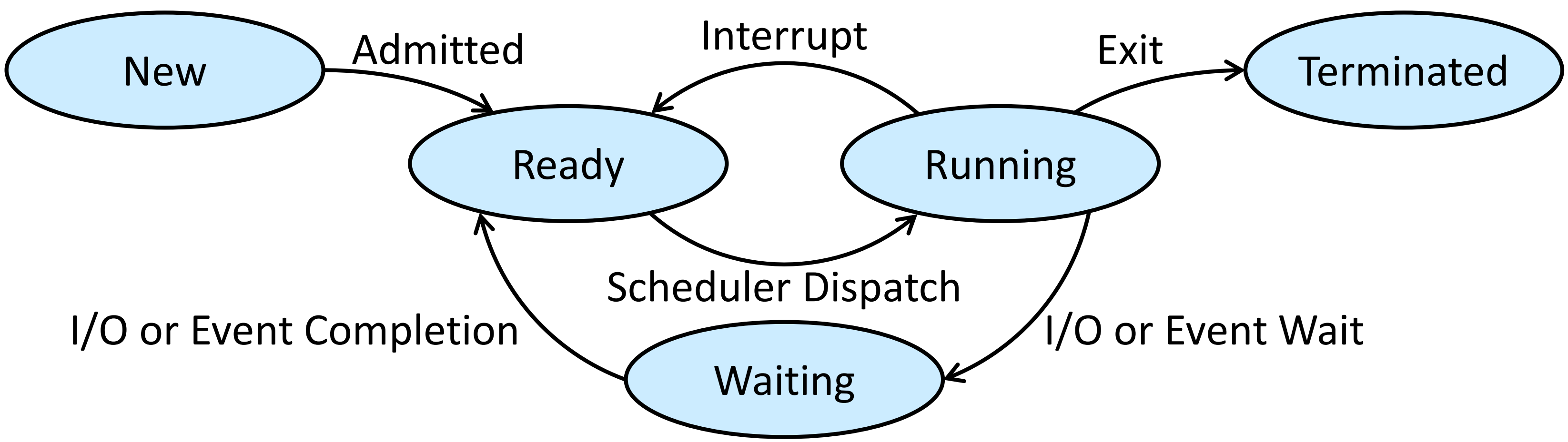
Memory Layout of a C Program




```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Process State

- As a process executes, it changes **state**
  - **New**: the process is being created
  - **Ready**: the process is waiting to be assigned to a processor
  - **Running**: instructions are being executed
    - Only one process can be running on any processor core at any instant
  - **Waiting**: the process is waiting for some event to occur
    - Examples: I/O completion, reception of a signal
  - **Terminated**: the process has finished execution



Process Control Block

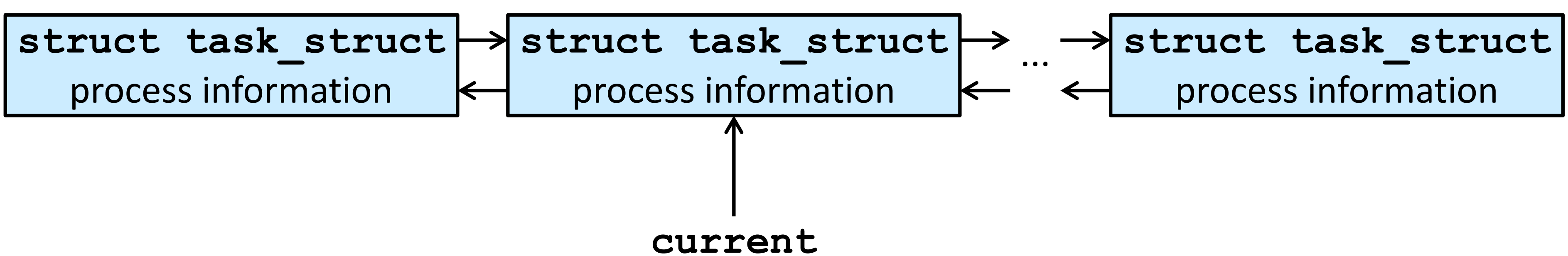
- Each process is represented in the operating system by a **process control block** (PCB), also called a **task control block**
  - Process state
  - Process number
  - Program counter
    - Address of the next instruction to be executed for this process
  - CPU registers
    - Register set where process needs to be stored for execution for running state
  - CPU-scheduling information: priority, queue pointers, etc. [Chapter 5]
  - Memory-management information: memory limits, etc. [Chapter 9]
  - Accounting information
    - Amount of CPU and real time used, time limits, account numbers, etc.
  - I/O status information
    - List of I/O devices allocated to the process, list of open files, etc.

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 */
```

- Within the Linux kernel, all active processes are represented using a doubly linked list of **`task_struct`**
  - The kernel maintains a pointer **`current`** to the process currently executing on the system



Threads

- The process model so far has implied that a process is a program that performs a single **<u>thread</u>** of execution
  - The process performs only one task at a time
  - Example: the user cannot simultaneously type in characters and run the spell checker
- Most modern operating systems allow a process to have multiple threads of execution
  - The process performs more than one task at a time
    - Especially beneficial on multicore systems where multiple threads can run in parallel
  - The PCB is expanded to include information for each thread
    - Other changes throughout the system are also needed [Chapter 4]

Outline

- Process Concept
- **Process Scheduling**
  - Scheduling Queues, CPU Scheduling, Context Switch
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Scheduling (1/2)

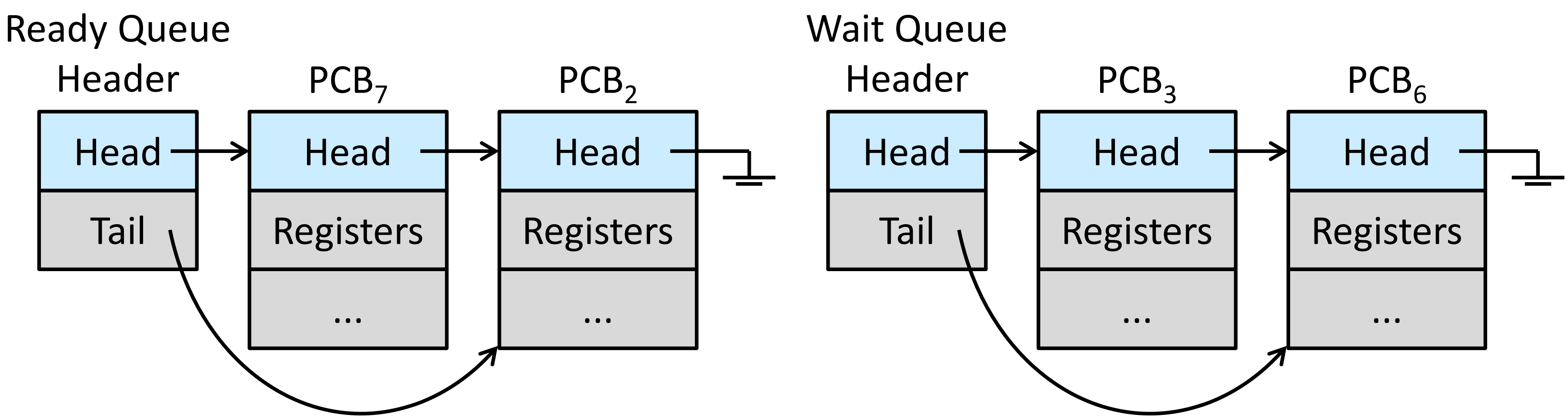
- Objective of multiprogramming
  - To have some process running at all times and maximize CPU utilization
- Objective of time sharing
  - To switch a CPU core among processes so frequently that users can interact with each program while it is running

Process Scheduling (2/2)

- To meet the objectives, the process scheduler selects an available process for program execution on a core
  - Each CPU core can run one process at a time
  - If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled
  - The number of processes currently in memory is known as the **degree of multiprogramming**
- To meet the objectives, it also requires taking the general behavior of a process into account
  - An **I/O-bound process** uses more of its time doing I/O than computations
  - A **CPU-bound process** uses more of its time doing computations than I/O

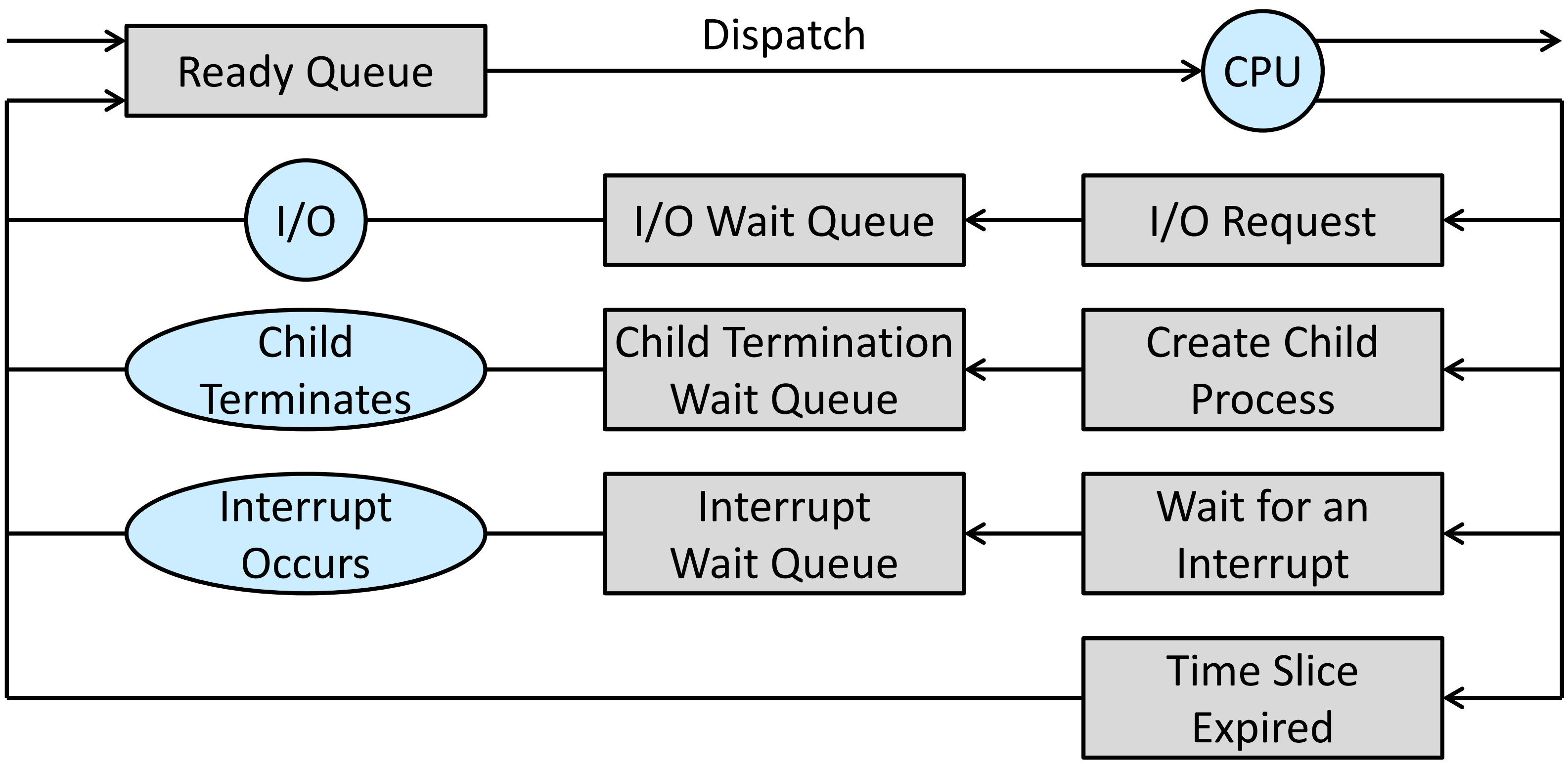
Scheduling Queues

- The system includes some queues
  - Ready queue
    - The set of processes ready and waiting to execute on a CPU's core
  - Wait queues
    - The set of processes waiting for a certain event (e.g., completion of I/O) to occur
  - A queue is generally stored as a linked list



Queueing Diagram

- A ready queue and a set of wait queues
- Circles represent the resources that serve the queues
- Arrows indicate the flow of processes in the system



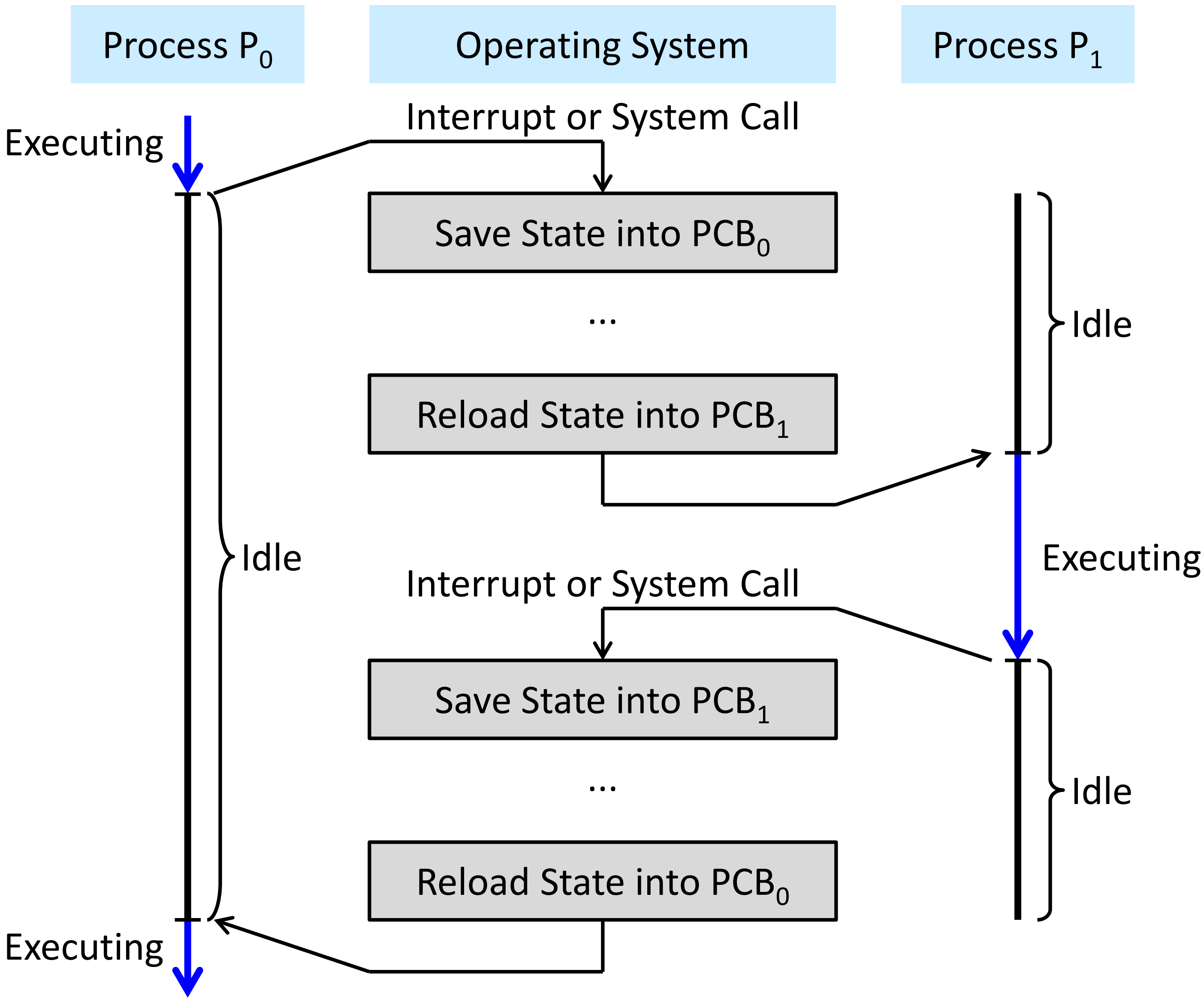
CPU Scheduling

- A process migrates among the ready queue and various wait queues throughout its lifetime
- A **CPU scheduler** executes at least once every 100 milliseconds, although typically much more frequently
  - An I/O-bound process may execute for only a few milliseconds before waiting for an I/O request
  - A CPU-bound process will require a CPU core for longer durations, but the scheduler is unlikely to grant the core to it for an extended period
- **Swapping** [Chapter 9]
  - Remove a process from memory (i.e., active contention for CPU) to disk
  - Reduce the degree of multiprogramming
  - Later, reintroduce the process into memory and continue its execution

Context Switch (1/2)

- Switching the CPU core from one process to another
- View of the current process
  - A **state save** of the current state of the CPU core and then a **state restore** to resume operations
- View of the system
  - **Context switch**: a state save of the current process and then a state restore of a different process
- The context of a process is represented in its PCB
  - The value of the CPU registers, the process state, and memory-management information, etc.

Diagram Showing Context Switch



Context Switch (2/2)

- Context switch time is pure overhead
  - The system does no useful work while switching
- Switching speed (usually several microseconds) depends on
  - Memory speed
  - Number of registers that must be copied
  - Existence of special instructions and hardware support
    - Example: a single instruction to load or store all registers
    - Example: multiple sets of registers, where a context switch here simply requires changing the pointer to the current register set
- The more complex the operating system, the greater the amount of work that must be done

Multitasking in Mobile Systems

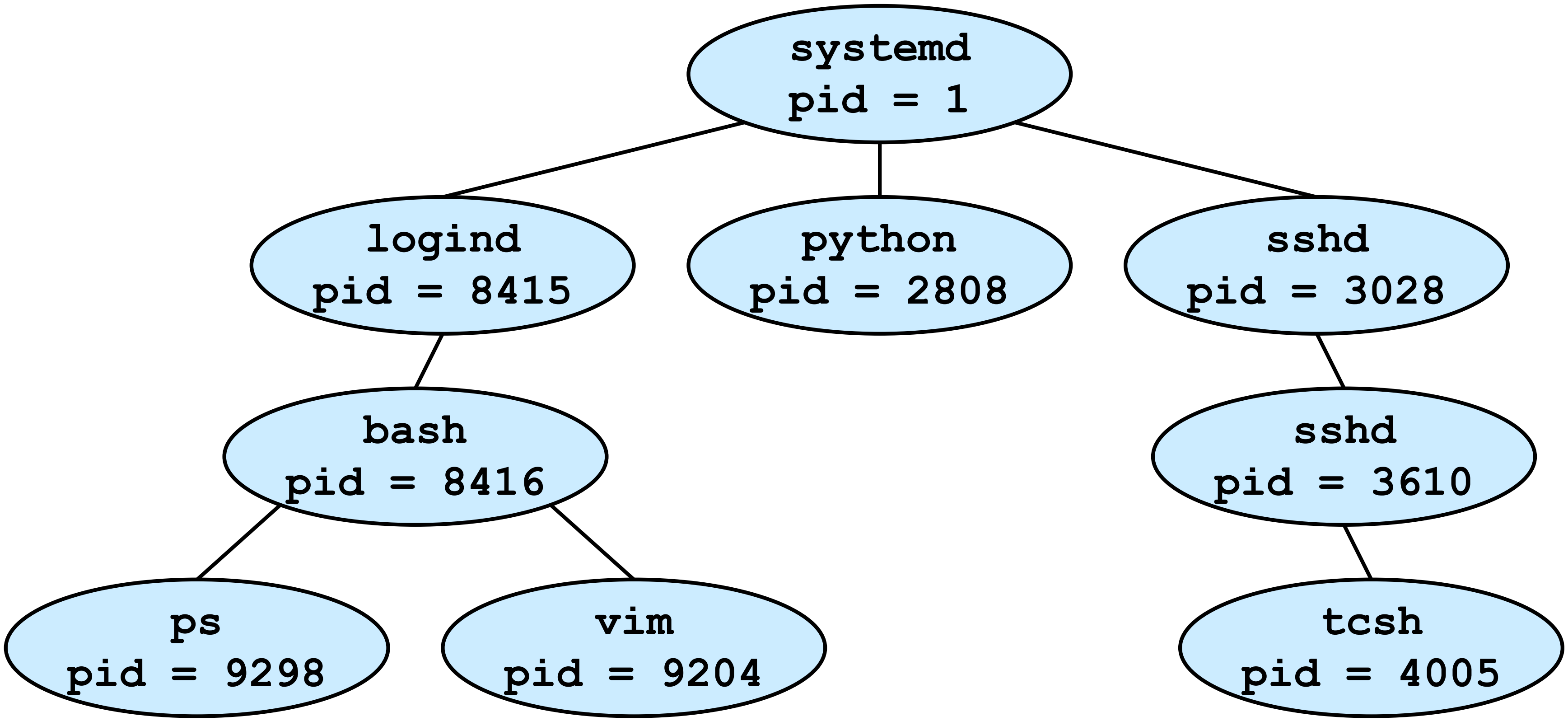
- Beginning with iOS 4, iOS allows a single **foreground** application to run with multiple **background** applications
  - Early versions of iOS does not provide user-application multitasking
    - Only one application runs in the foreground
    - All other user applications are suspended
  - **Split-screen**
    - A larger screen allows running two foreground applications at the same time
- Android runs foreground and background with fewer limits
  - A **service** runs on behalf of the background process
    - The service runs even if the background application is suspended
    - Services do not have a user interface and have a small memory footprint

Outline

- Process Concept
- Process Scheduling
- **Operations on Processes**
  - **Process Creation**, Process Termination
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Creation (1/2)

- A **parent** process creates **children** processes, which in turn create other processes, forming a tree of processes
  - Generally, processes are identified via a **process identifier** (`pid`)
- A tree of processes on a typical Linux system



Process Creation (2/2)

- Resource (memory, files, etc.) sharing options
  - The child shares all of the parent's resources
  - The child shares a subset of the parent's resources
    - Prevent any process from overloading the system by creating too many children
  - The child shares none of the parent's resources
- Execution options
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated
- Address-space options
  - The child is a duplicate of the parent process
    - It has the same program and data as the parent
  - The child has a new program loaded into it

Process Creation Using UNIX `fork()`

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 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");
    }
    return 0;
}
```

Process Creation Using UNIX **`fork()`**

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    /* fork a child process */
    pid = fork();
```

- **`fork()`** creates a new process
  - The child (new) process consists of a copy of the address space of the parent (original) process
  - Both processes continue execution at the instruction after **`fork()`** with one difference
    - The return code for **`fork()`** is nonzero (**`pid`** of the child) for the parent
    - The return code for **`fork()`** is zero for the child

```
        printf("Child Complete");
    }
    return 0;
}
```

Process Creation Using UNIX `fork()`

- **`exec()`** loads a binary file into memory and starts execution
  - **`execlp()`** in the example code
  - It destroys the original memory image

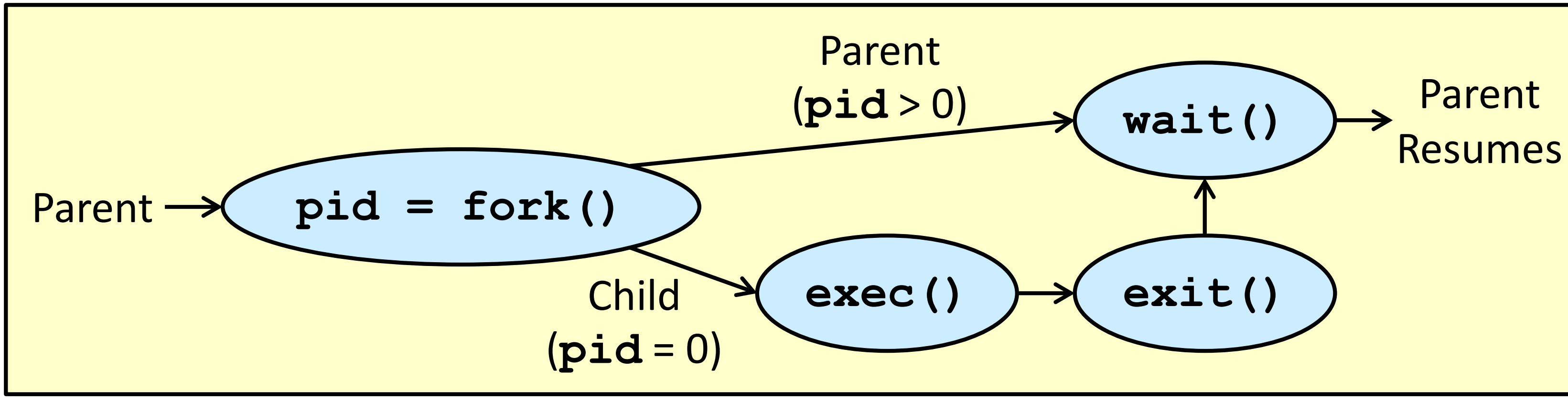
```
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait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

Process Creation Using UNIX `fork()`

- The parent calls **`wait()`** to move itself off the ready queue until the termination of the child
  - The child terminates by either implicitly or explicitly invoking **`exit()`**
    - Implicitly: the C run-time library (added to UNIX executable files) includes **`exit()`** by default
    - Explicitly: directly use **`exit()`**

```
            return 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");
        }
        return 0;
    }
```

Process Creation Using UNIX `fork()`




```
        fprintf(stderr, "Fork Failed");
        return 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");
    }
    return 0;
}
```

Process Creation Using Windows API

- **`CreateProcess()`** requires loading a specified program into the address space of the child process at process creation
  - UNIX example: **`fork()`** has the child process inheriting the address space of its parent
- **`CreateProcess()`** expects no fewer than ten parameters
  - UNIX example: **`fork()`** is passed no parameter
- **`WaitForSingleObject()`** waits for the child process to complete
  - UNIX example: **`wait()`** is equivalent
- Note: address-space options
  - The child is a duplicate of the parent process
    - It has the same program and data as the parent
  - The child has a new program loaded into it

Process Creation Using Windows API

```
int main(VOID){
      STARTUPINFO si;
      PROCESS_INFORMATION pi;
      /* allocate memory */
      ZeroMemory(&si, sizeof(si));
      si.cb = sizeof(si);
      ZeroMemory(&pi, sizeof(pi));
      /* create child process */
      if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si, &pi)){
           fprintf(stderr, "Create Process Failed");
           return -1;
      }
      /* parent will wait for the child to complete */
      WaitForSingleObject(pi.hProcess, INFINITE);
      printf("Child Complete");
      /* close handles */
      CloseHandle(pi.hProcess);
      CloseHandle(pi.hThread);
}
```

Outline

- Process Concept
- Process Scheduling
- **Operations on Processes**
  - Process Creation, **Process Termination**
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Termination (1/3)

- A process finishes executing its final statement and asks the operating system to delete it by using the **`exit()`** system call
  - A status value (typically an integer) is returned to its waiting parent
  - All the resources are deallocated and reclaimed by the operating system
- **`exit()`** provides an exit status

```
/* exit with status 1 */
exit(1);
```

- **`wait()`** returns the process identifier of the terminated child

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

  - What if there are multiple children or there is no child?
    - http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/wait.html

Process Termination (2/3)

- A parent may terminate the execution of one of its children
  - The child has exceeded its usage of some of the resources that it has been allocated
    - The parent must have a mechanism to inspect the state of its children
  - The task assigned to the child is no longer required
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates
    - **Cascading termination**: if a process terminates (either normally or abnormally), then all its children must also be terminated
    - It is normally initiated by the operating system

Process Termination (3/3)

- A **zombie** is a process that has terminated, but whose parent has not yet called **`wait()`**
  - When a process terminates, its resources are deallocated by the OS
  - However, its entry in the process table (containing the exit status) must remain until the parent calls **`wait()`**
  - Once the parent calls **`wait()`**, the zombie's process identifier and process-table entry are released
- An **orphan** is a process that its parent did not invoke **`wait()`** and terminated
  - UNIX assigns the **`systemd`** (or **`init`**) process as the new parent
  - The **`systemd`** (or **`init`**) process periodically invokes **`wait()`**
    - Allow the exit status of any orphan to be collected
    - Release the orphan's process identifier and process-table entry

Android Process Hierarchy

- Importance hierarchy from most to least important
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- If system resources must be reclaimed, Android will first terminate a least-important process

Chrome Browser

- Multiprocess architecture: three different types of processes
  - The **browser** process manages the user interface, disk and network I/O
  - **Renderer** processes handles HTML, Javascript, images, and so forth
    - A new renderer process is created for each website opened in a new tab
  - A **plug-in** process is created for each type of plug-in in use
    - Examples: QuickTime or Flash
- Each tab represents a separate process
  - Better isolation
  - Better security
    - Renderer processes run in a **sandbox** which means that access to disk and network I/O is restricted

Outline

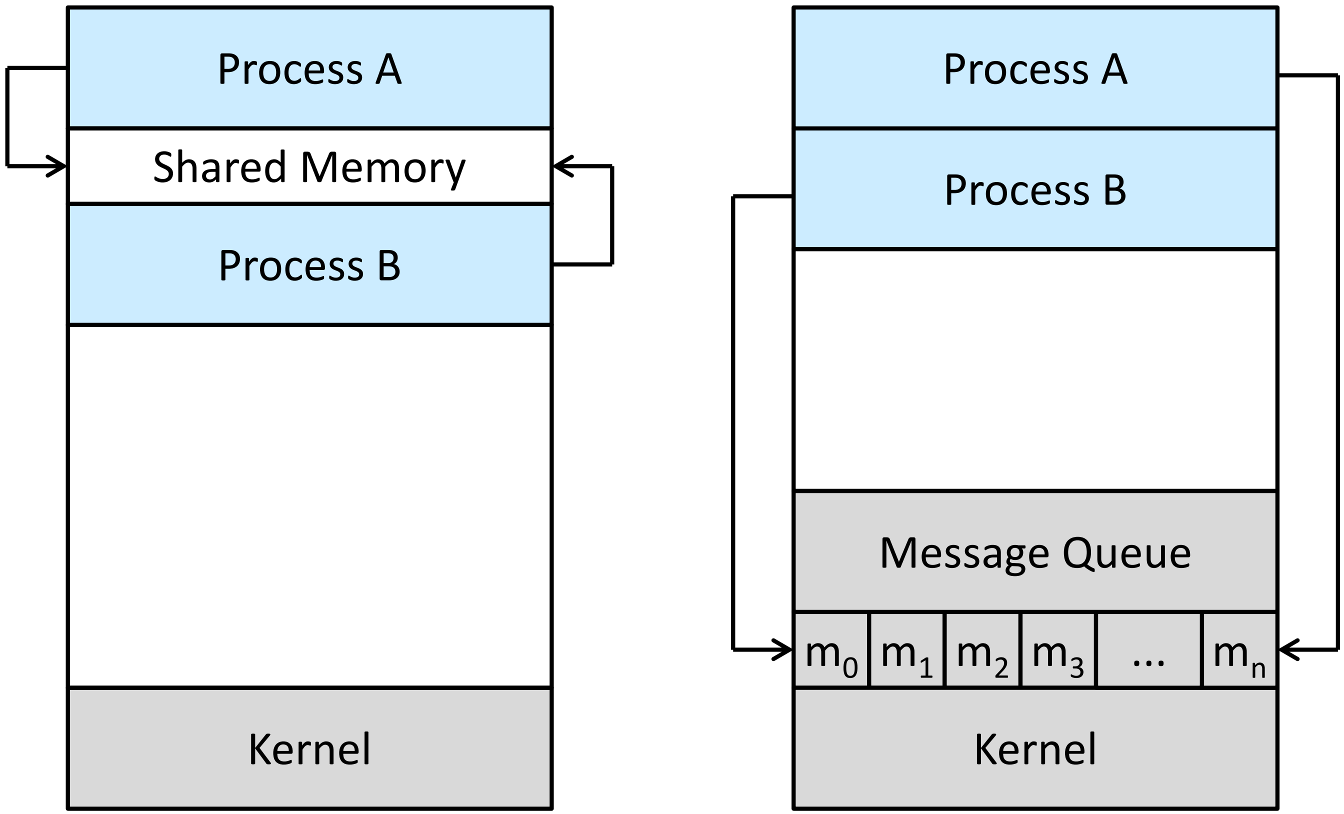
- Process Concept
- Process Scheduling
- Operations on Processes
- **<u>Interprocess Communication</u>**
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Interprocess Communication (1/2)

- A process may be either independent or cooperating
  - A process is **independent** if it does not share data with any other processes executing in the system
  - A process is **cooperating** if it can affect or be affected by the other processes executing in the system
    - Any process that shares data with other processes is a cooperating process
- Reasons for process cooperation
  - Information sharing, computation speedup, modularity
- Cooperating processes require an **interprocess communication** (IPC) mechanism for data exchange
  - Model 1: **shared memory**
    - Generally faster (routine memory access once shared memory is established)
  - Model 2: **message passing**
    - Easier to implement in a distributed system

Interprocess Communication (2/2)

- Cooperating processes require an interprocess communication (IPC) mechanism for data exchange



Shared Memory

Message Passing

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- **IPC in Shared-Memory Systems**
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

IPC in Shared-Memory Systems

- Require communicating processes to establish a region of shared memory
  - Typically, the region resides in the address space of the process creating the shared-memory segment
  - Other processes attach the segment to their address space
- Normally, the operating system tries to prevent one process from accessing another process's memory
  - Shared memory requires that two or more processes agree to remove this restriction
  - The processes are also responsible for
    - The form of the data
    - The location of the data
    - Not writing to the same location simultaneously

Producer-Consumer Problem

- A **producer** process produces information that is consumed by a **consumer** process
  - Examples
    - A compiler may produce assembly code that is consumed by an assembler
    - The assembler may produce object modules that are consumed by the loader
    - A web server produces web content such as HTML files and images, which are consumed by the client web browser requesting the resource
- Shared memory solution
  - Have a buffer of items that can be filled (produced) by the producer and emptied (consumed) by the consumer
    - This buffer resides in the shared memory
  - Synchronize the producer and the consumer [Chapters 6 and 7]
    - The consumer does not try to consume an item that has not yet been produced

Types of Buffers

- The **unbounded buffer** places no practical limit on the size of the buffer
  - The producer can always produce new items
  - The consumer may have to wait for new items
- The **bounded buffer** assumes a fixed buffer size
  - The producer must wait if the buffer is full
  - The consumer must wait if the buffer is empty

Bounded Buffer

- The shared **`buffer`**

```
#define BUFFER_SIZE 10
typedef struct {
   ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

  - The variable **`in`** points to the next free position in the buffer
  - The variable **`out`** points to the first full position in the buffer
  - The buffer is empty when **`in == out`**
  - The buffer is full when **`((in + 1) % BUFFER_SIZE) == out`**
  - This scheme allows at most **`(BUFFER_SIZE – 1)`** items in the buffer

Producer Process Using Shared Memory

- Producer process using shared memory

```
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;
}
```

Consumer Process Using Shared Memory

- Consumer process using shared memory

```
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 */
}
```

- The issue of synchronization is not addressed here
  - Both of the producer process and the consumer process attempt to access the shared buffer concurrently [Chapters 6 and 7]

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- **IPC in Message-Passing Systems**
  - Naming, Synchronization, Buffering
- Examples of IPC Systems
- Communication in Client-Server Systems

IPC in Message-Passing Systems (1/2)

- A message-passing facility provides at least two operations

```
send(message)
receive(message)
```

- Messages can be either fixed or variable in size
  - The system-level implementation is more straightforward with fixed-sized messages
  - The programming tasks are simpler with variable-sized messages

IPC in Message-Passing Systems (2/2)

- A communication link must exist between processes sending and receiving messages
- This link can be implemented in a variety of ways
  - We are concerned here not with the link's physical implementation
    - Examples: shared memory, hardware bus, or network [Chapter 19]
  - We are concerned with its **logical implementation**
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- **IPC in Message-Passing Systems**
  - **Naming**, Synchronization, Buffering
- Examples of IPC Systems
- Communication in Client-Server Systems

Direct Communication (1/2)

- Each process that wants to communicate must explicitly name the recipient or sender of the communication
  - Symmetry in addressing
    - **`send(P, message)`** : send a message to process **`P`**
    - **`receive(Q, message)`** : receive a message from process **`Q`**
- Properties of communication link
  - A link is established automatically between every pair of processes that want to communicate
    - The processes need to know only each other's identity to communicate
  - A link is associated with exactly two processes
  - Between each pair of processes, there exists exactly one link

Direct Communication (2/2)

- Each process that wants to communicate must explicitly name the recipient or sender of the communication
  - Asymmetry in addressing
    - **`send(P, message)`** : send a message to process **`P`**
    - **`receive(id, message)`** : receive a message from any process, where **`id`** is set to the name of the process with which communication has taken place
- Disadvantage (both symmetric and asymmetric): limited modularity
  - Changing the identifier of a process may necessitate examining all other process definitions
  - All references to the old identifier must be found and modified to the new identifier

Indirect Communication (1/4)

- The messages are sent to and received from mailboxes (or ports)
  - **`send(A, message)`** : send a message to mailbox **`A`**
  - **`receive(A, message)`** : receive a message from mailbox **`A`**
  - A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed
  - Each mailbox has a unique identification
  - A process can communicate with another process via a number of different mailboxes
  - Two processes can communicate only if they have a shared mailbox

Indirect Communication (2/4)

- Properties of indirect communication link
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox
- Properties of direct communication link (for comparison)
  - A link is established automatically between every pair of processes that want to communicate
  - A link is associated with exactly two processes
  - Between each pair of processes, there exists exactly one link

Indirect Communication (3/4)

- Processes P_1, P_2, and P_3 share mailbox **`A`**
  - Process P_1 sends a message to **`A`**
  - Both P_2 and P_3 execute a **`receive()`** from **`A`**
  - Which process will receive the message sent by P_1?
- Some solutions
  - Allow a link to be associated with two processes at most
  - Allow at most one process at a time to execute a **`receive()`** operation
  - Allow the system to select arbitrarily which process will receive the message
    - Define an algorithm for selecting which process will receive the message

Indirect Communication (4/4)

- A mailbox owned by a process (the mailbox is part of the address space of the process)
  - When a process that owns a mailbox terminates, the mailbox disappears
  - Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists
- A mailbox owned by the operating system
  - The operating system must allow a process to
    - Create a new mailbox
    - Send and receive messages through the mailbox
    - Delete a mailbox
  - The process that creates a new mailbox is the mailbox's owner
  - The ownership and receiving privilege may be passed to other processes through appropriate system calls

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- **IPC in Message-Passing Systems**
  - Naming, **Synchronization**, Buffering
- Examples of IPC Systems
- Communication in Client-Server Systems

Synchronization

- Message passing may be either **blocking** (**synchronous**) or **nonblocking** (**asynchronous**)
  - Blocking send
    - The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Nonblocking send
    - The sending process sends the message and resumes operation
  - Blocking receive
    - The receiver blocks until a message is available
  - Nonblocking receive
    - The receiver retrieves either a valid message or a null
- Different combinations of **`send()`** and **`receive()`** are possible
  - When both send() and receive() are blocking, we have a **rendezvous**

Blocking Producer and Consumer

- The solution is trivial with blocking **`send()`** and **`receive()`**
- The producer process

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

- The consumer process

```
message next_consumed;
while (true) {
   receive(next_consumed);
   /* consume the item in next_consumed */
}
```

Outline

- ❑ Process Concept
- ❑ Process Scheduling
- ❑ Operations on Processes
- ❑ Interprocess Communication
- ❑ IPC in Shared-Memory Systems
- ❑ **IPC in Message-Passing Systems**
  - ➢ Naming, Synchronization, **Buffering**
- ❑ Examples of IPC Systems
- ❑ Communication in Client-Server Systems

Buffering

- Messages exchanged by communicating processes reside in a temporary queue
  - No matter communication is direct or indirect
- Basically, such queues can be implemented in three ways
  - Zero capacity
    - The sender must block until the recipient receives the message
  - Bounded capacity
    - If the link is full, the sender must block until space is available in the queue
  - Unbounded capacity
    - The sender never blocks

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- **Examples of IPC Systems**
  - **POSIX Shared Memory**, Mach Message Passing, Windows, Pipes
- Communication in Client-Server Systems

POSIX Shared Memory

- Portable Operating System Interface (POSIX)
  - A family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems [Wikipedia]
    - Variants of Unix and other operating systems
- Several IPC mechanisms are available for POSIX systems
- Here, we explore the POSIX API for shared memory
  - Create a shared-memory object
    - `shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);`
  - Configure the size of the object in bytes
    - `ftruncate(shm_fd, 4096);`
  - Establish a memory-mapped file containing the shared-memory object and return a pointer to the memory-mapped file
    - `mmap()`

https://man7.org/linux/man-pages/man3/shm_open.3.html
https://man7.org/linux/man-pages/man2/mmap.2.html

Producer with POSIX Shared Memory

```
int main() {
     /* the size (in bytes) of shared memory object */
     const int SIZE = 4096;
     /* name of the shared memory object */
     const char *name = "OS";
     /* strings written to shared memory */
     const char *message_0 = "Hello";
     const char *message_1 = "World!";
     /* shared memory file descriptor */
     int fd;
     /* pointer to shared memory obect */
     char *ptr;
     /* create the shared memory object */
     fd = shm_open(name, O_CREAT | O_RDWR, 0666);
     /* configure the size of the shared memory object */
     ftruncate(fd, SIZE);
     /* memory map the shared memory object */
     ptr = (char *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     /* write to the shared memory object */
     sprintf(ptr,"%s",message_0);
     ptr += strlen(message_0);
     sprintf(ptr,"%s",message_1);
     ptr += strlen(message_1);
     return 0;
}
```

Consumer with POSIX Shared Memory

```
int main() {
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory obect */
    char *ptr;
    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = (char *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    /* read from the shared memory object */
    printf("%s", (char *)ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

Outline

- ❑ Process Concept
- ❑ Process Scheduling
- ❑ Operations on Processes
- ❑ Interprocess Communication
- ❑ IPC in Shared-Memory Systems
- ❑ IPC in Message-Passing Systems
- ❑ **Examples of IPC Systems**
  - ➢ POSIX Shared Memory, **Mach Message Passing**, Windows, Pipes
- ❑ Communication in Client-Server Systems

Mach Message Passing (1/3)

- Mach was included in the macOS and iOS operating systems
  - Mach was initially designed for distributed systems
- Most communication in Mach, including all inter-task communication, is carried out by messages
  - Create a new **port** (mailbox) and allocate space for its message queue
    - `mach_port_allocate()`
  - Send and receive messages
    - `mach_msg()`
- Two special ports are created with a task (similar to a process)
  - The kernel has receive rights to the **Task Self** port
    - The task can send messages to the kernel
  - The task has receive rights to the **Notify** port
    - The kernel can send notification of event occurrences to the task

Mach Message Passing (2/3)

- Two message fields
  - A fixed-size message header and a variable-sized body

```
struct message {
   mach_msg_header_t header;
   int data;
};
```

- The send and receive operations themselves are flexible
  - Example: when a message is sent to a port, if the port's queue is full, the sender has several options (specified via parameters to `mach_msg()`)
    - Wait indefinitely until there is room in the queue
    - Wait at most n milliseconds
    - Do not wait at all but rather return immediately
    - Temporarily cache a message

Mach Message Passing (3/3)

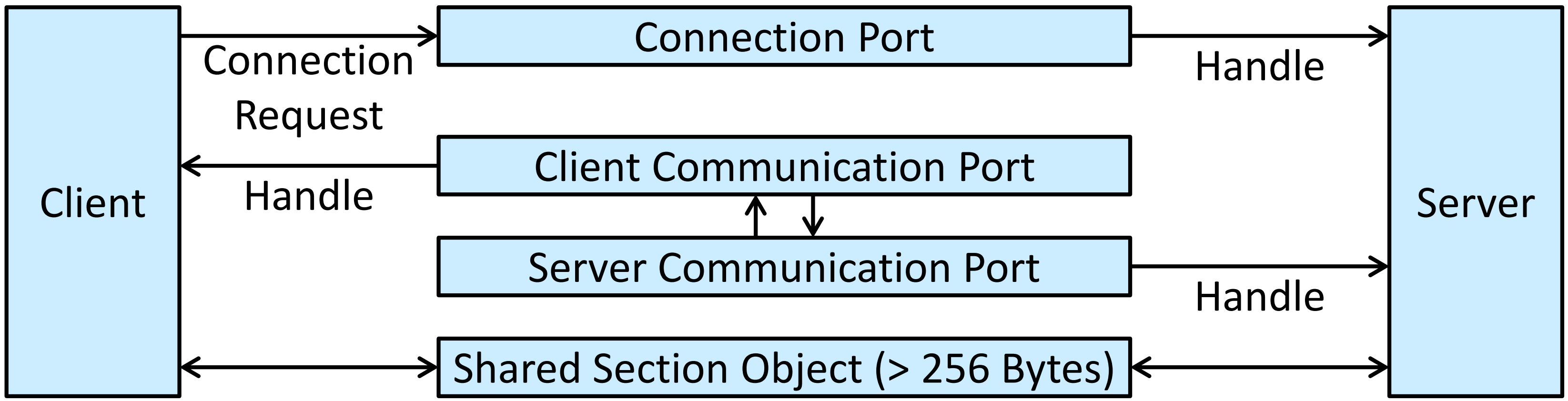
```
mach_port_t client;
mach_port_t server;

/* Client Code */
struct message message;
// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;
// send the message
mach_msg(&message.header, // message header
      MACH_SEND_MSG, // sending a message
      sizeof(message), // size of message sent
      0, // maximum size of received message - unnecessary
      MACH_PORT_NULL, // name of receive port - unnecessary
      MACH_MSG_TIMEOUT_NONE, // no time outs
      MACH_PORT_NULL // no notify port
);

/* Server Code */
struct message message;
// receive the message
mach_msg(&message.header, // message header
      MACH_RCV_MSG, // sending a message --> receiving a message
      0, // size of message sent
      sizeof(message), // maximum size of received message
      server, // name of receive port
      MACH_MSG_TIMEOUT_NONE, // no time outs
      MACH_PORT_NULL // no notify port
);
```

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- **Examples of IPC Systems**
  - POSIX Shared Memory, Mach Message Passing, **Windows**, Pipes
- Communication in Client-Server Systems

Windows

- **Support multiple operating environments (subsystems)**
  - Application programs communicate with these subsystems via a message-passing mechanism
  - Application programs can be considered clients of a subsystem server
- **Advanced local procedure call (ALPC) facility**
  - The client opens a handle (abstract reference) to the server's **connection-port** object and sends a connection request to that port
  - The server creates a channel and returns a handle to the client
    - The channel consists of two private **communication ports** for client-server messages and for server-client messages



Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- **Examples of IPC Systems**
  - POSIX Shared Memory, Mach Message Passing, Windows, **Pipes**
- Communication in Client-Server Systems

Pipes

- A conduit allowing two processes to communicate
  - Pipes were one of the first IPC mechanisms in early UNIX systems
- Simple but with some issues
  - Bidirectional or unidirectional?
  - If bidirectional, half duplex (only one direction at a time) or full duplex (both directions at a time)?
  - Must a relationship (such as parent-child) exist between the communicating processes?
  - Can the pipes communicate over a network, or must the communicating processes reside on the same machine?
- Two common types used on both UNIX and Windows systems
  - Ordinary pipes
  - Named pipes

Ordinary Pipes

- Ordinary pipes allow two processes to communicate in standard producer-consumer fashion
  - The producer writes to one end of the pipe (the **write end**)
  - The consumer reads from the other end (the **read end**)
- An ordinary pipe is unidirectional
- An ordinary pipe cannot be accessed from outside the process that created it
- Ordinary pipes on Windows are termed **anonymous pipes**

Ordinary Pipes in UNIX

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void) {
      char write_msg[BUFFER_SIZE] = "Greetings";
      char read_msg[BUFFER_SIZE];
      int fd[2];
      pid_t pid;
      pipe(fd);                       /* create the pipe */
      /* omit codes if pipe failed */
      pid = fork();                   /* fork a child process */
      /* omit codes if fork failed */
      if (pid > 0) {                  /* parent process */
            close(fd[READ_END]);      /* close the unused end of the pipe */
            write(fd[WRITE_END], write_msg, strlen(write_msg)+1); /* write to the pipe */
            close(fd[WRITE_END]);     /* close the write end of the pipe */
      }
      else {                          /* child process */
            close(fd[WRITE_END]);     /* close the unused end of the pipe */
            read(fd[READ_END], read_msg, BUFFER_SIZE); /* read from the pipe */
            printf("read %s",read_msg);
            close(fd[READ_END]);      /* close the read end of the pipe */
      }
      return 0;
}
```

Named Pipes

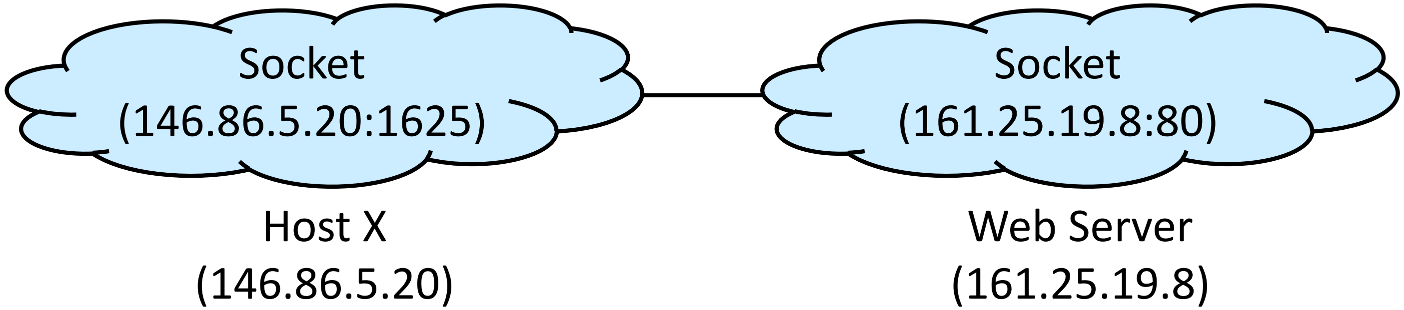
- Named pipes provide a much more powerful communication tool
  - Communication can be bidirectional
  - No parent-child relationship is required
  - Several processes can use a named pipe for communication
    - In a typical scenario, a named pipe has several writers
  - Named pipes continue to exist after communicating processes have finished
- Both UNIX and Windows systems support named pipes
  - Half-duplex in UNIX
  - Full-duplex in Windows

Outline

- ❑ Process Concept
- ❑ Process Scheduling
- ❑ Operations on Processes
- ❑ Interprocess Communication
- ❑ IPC in Shared-Memory Systems
- ❑ IPC in Message-Passing Systems
- ❑ Examples of IPC Systems
- ❑ **Communication in Client-Server Systems**
  - ➢ **Sockets**, Remote Procedure Calls

Sockets

- A **socket** is defined as an endpoint for communication
  - A socket is identified by an IP address concatenated with a port number
- Sockets use a client-server architecture
  - A server waits for client requests by listening to a specified port
  - Once a request is received, the server accepts a connection from the client socket to complete the connection
- All ports below 1024 are considered **well-known**
  - SSH: port 22; FTP: port 21; HTTP: port 80
- Communication using a pair of sockets
  - (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (TCP) sockets
    - **`Socket`** class
  - **Connectionless** (UDP) sockets
    - **`DatagramSocket`** class
  - Multicast sockets, allowing data to be sent to multiple recipients
    - **`MulticastSocket`** class, a subclass of the **`DatagramSocket`** class

Date Server in Java

```
import java.net.*;
import java.io.*;
public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();
                PrintWriter pout = new
                  PrintWriter(client.getOutputStream(), true);
                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());
                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Date Server in Java

- The server listens to the port with the **`accept()`** method
  - The server blocks on the **`accept()`** method waiting for a client to request a connection
  - When a connection request is received, **`accept()`** returns a socket that the server can use to communicate with the client

```
                        while (true) {
                                Socket client = sock.accept();
                                PrintWriter pout = new
                                  PrintWriter(client.getOutputStream(), true);
                                /* write the Date to the socket */
                                pout.println(new java.util.Date().toString());
                                /* close the socket and resume */
                                /* listening for connections */
                                client.close();
                        }
                }
                catch (IOException ioe) {
                        System.err.println(ioe);
                }
        }
}
```

Date Client in Java

```
import java.net.*;
import java.io.*;
public class DateClient
{
    public static void main(String[] args) {
        try {
            /* make connection to server socket */
            Socket sock = new Socket("127.0.0.1",6013);
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
              BufferedReader(new InputStreamReader(in));
            /* read the date from the socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);
            /* close the socket connection*/
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- **Communication in Client-Server Systems**
  - Sockets, **Remote Procedure Calls**

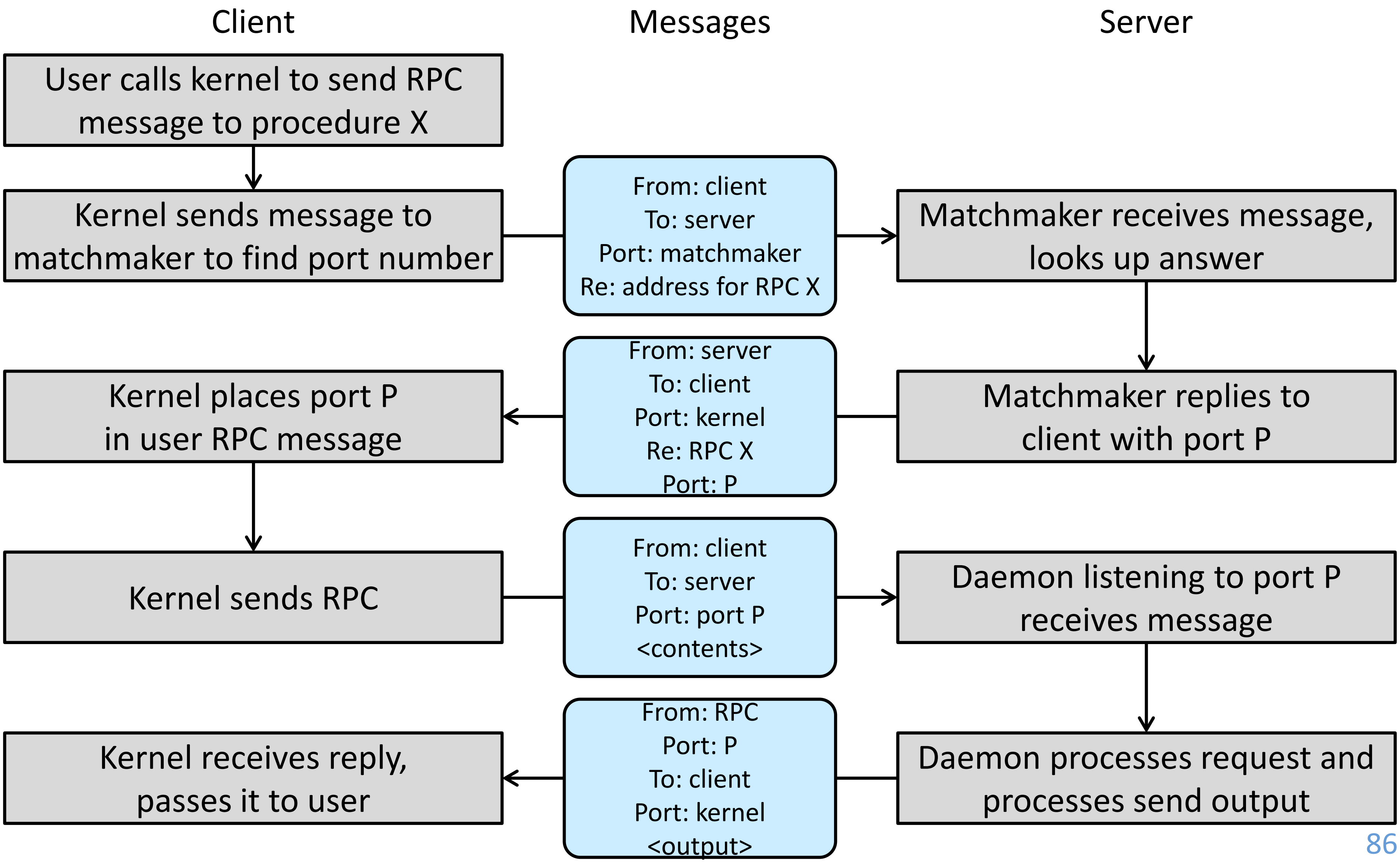
Remote Procedure Calls (1/2)

- Remote procedure call (RPC) **abstracts** the procedure-call mechanism between systems with network connections
  - Allow a client to invoke a procedure on a remote host as it would invoke a procedure locally
- The RPC system provides a **stub** on the client side
  - When the client invokes a remote procedure, the RPC system calls the appropriate stub (with parameters)
  - This stub locates the port on the server and **marshals** the parameters
  - The stub transmits a message to the server using message passing
  - A similar stub on the server side receives this message and invokes the procedure on the server
    - If necessary, return values are passed back to the client using the same technique
- Microsoft Interface Definition Language (MIDL) on windows

Remote Procedure Calls (2/2)

- Many RPC systems define a machine-independent representation of data
  - Example: external data representation (XDR)
    - On the client side, parameter marshaling involves converting the machine-dependent data into XDR
    - On the server side, the XDR data are unmarshaled and converted to the machine-dependent representation
- Remote communication has more failure scenarios
  - Messages can be acted on **exactly once**, rather than **at most once**
- Binding
  - Predetermined
  - Dynamic
    - An operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port

Dynamic Binding and Execution of RPC



Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations
- Describe and contrast interprocess communication using shared memory and message passing
- Design programs that use pipes and POSIX shared memory to perform interprocess communication
- Describe client-server communication using sockets and remote procedure calls

Q&A