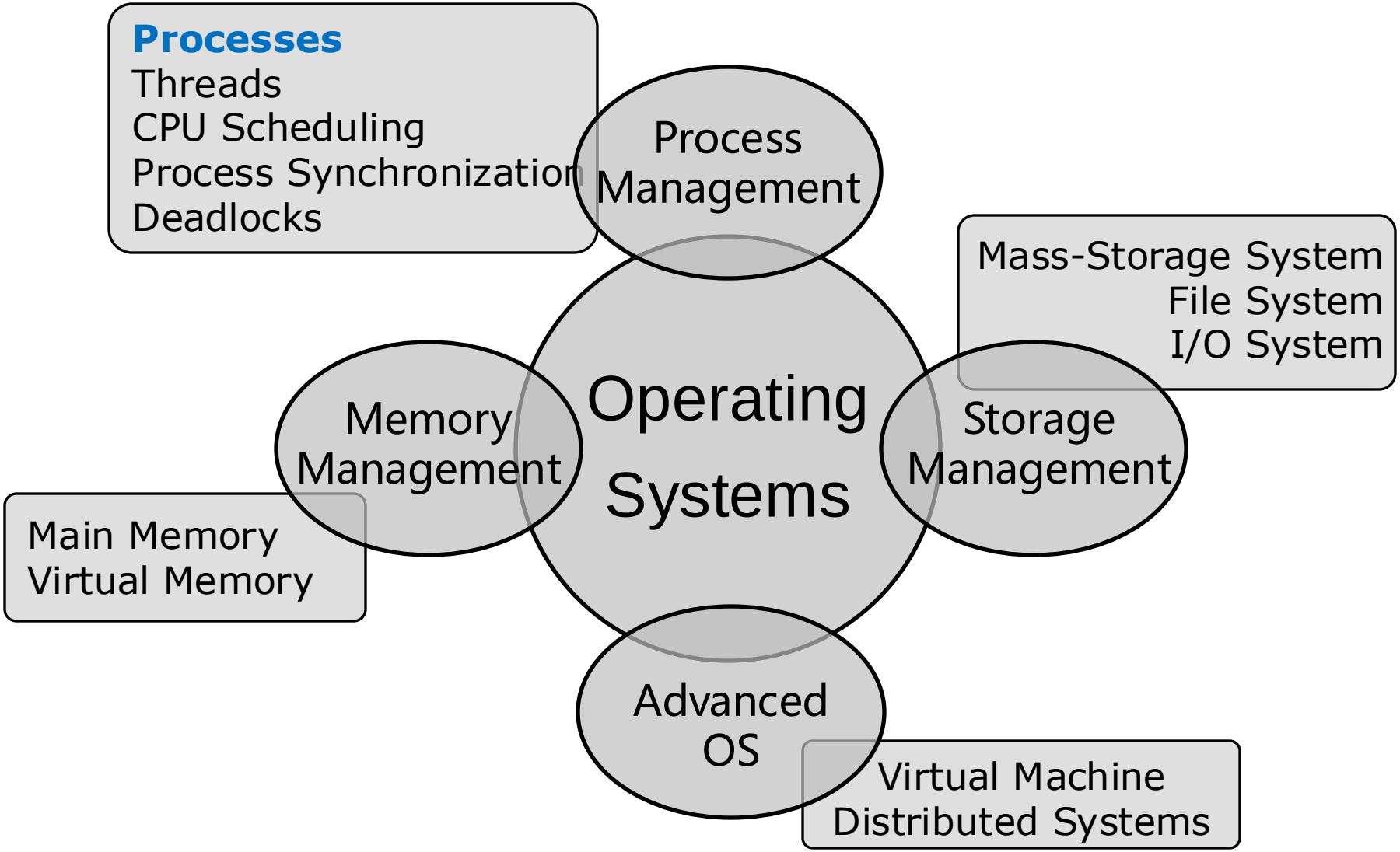
Processes

Shengzhong Liu
Department of Computer Science and Engineering
Shanghai Jiao Tong University

Operating System Topics



Outline

- Process Basics:
  - Concept
  - Scheduling
  - Operations
- Interprocess Communication
  - IPC: Shared Memory
  - IPC: Message Passing
    - Direct Communication
    - Indirect Communication
    - Example: UNIX Pipes
- Communication in Client-Server Systems
  - Sockets
  - Remote Procedure Call (RPC)

Process Concept

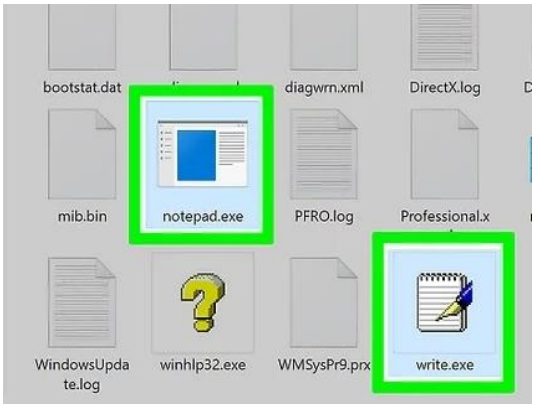
Process Concept

- **Process** is a program in execution
  - Process execution must progress in sequential fashion
  - No parallel execution of instructions of a single process
- **Program** vs. **Process**
  - Program becomes process when an executable file is loaded into memory

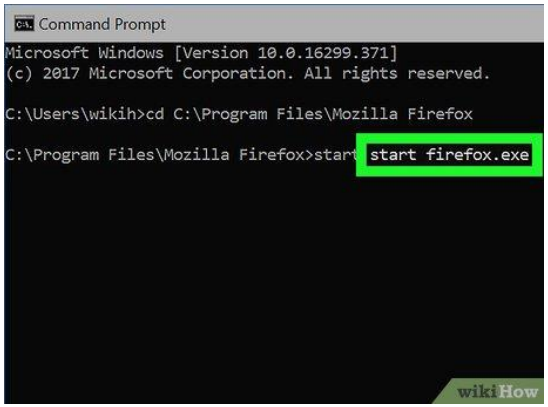
	Program	Process
Definition	Passive entity in secondary storage	Active entity created during program execution
Property	Contains a set of instruction	An instance of executing program loaded into the main memory
Lifetime	Exist for a long time until deleted	Exist for a limited time until termination
Resource Requirement	No resource requirement	Requires CPU, memory, I/O
Control	No control block	Has control block (i.e., PCB)

Process Concept

- Two ways to load executable files:
  - (Double-)Click an icon of the executable program file
  - Enter the executable file on the command line



Double Click in GUI

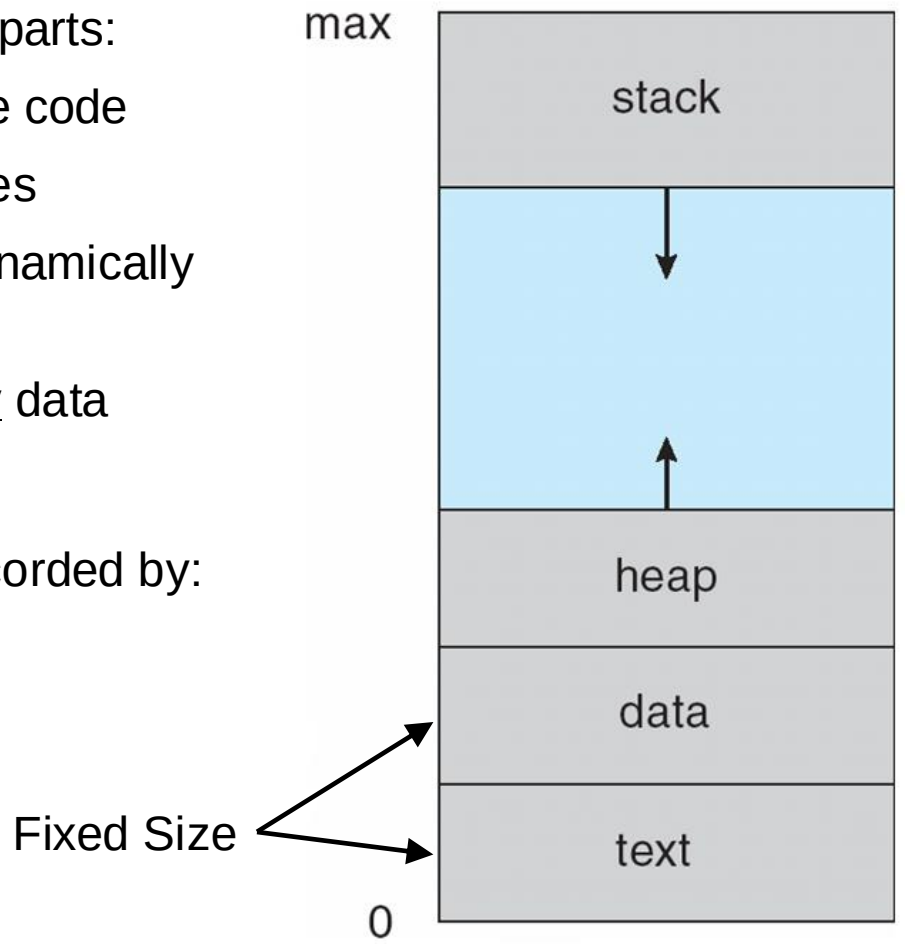


Command Line

- One program can become several processes
  - Multiple users can execute the same program
  - Same user may invoke many copies of the web browser program

Process in Memory (1)

- Process memory is divided into 4 parts:
  - **Text section** – the executable code
  - **Data section** – global variables
  - **Heap** - containing memory dynamically allocated during run time
  - **Stack** - containing **temporary** data when invoking functions
- Current process activity status recorded by:
  - Program counter
  - Processor registers contents



Process memory layout

Heap Memory vs. Heap Data Structure

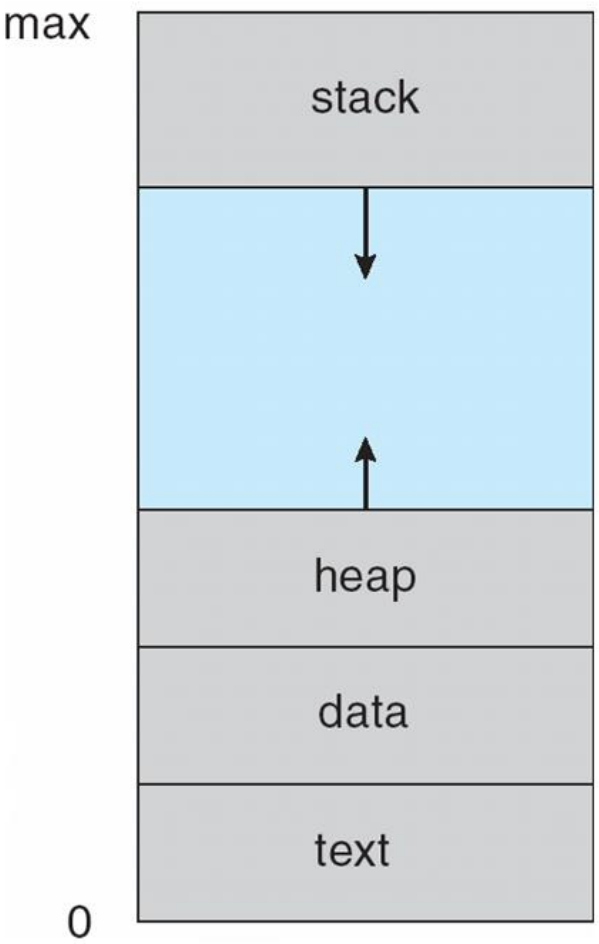
- **Heap memory area**:
  - A region of a process's memory used for dynamic memory allocation
  - "Heap" in memory means a large collection of things without order
- **Heap data structure**:
  - A type of binary tree that satisfies the heap property
  - Commonly used to implement priority queues
- The heap memory area is not related to the heap data structure in terms of implementation

Heap

Stack

Process in Memory (2)

- The sizes of the text and data sections are fixed.
- The **stack** and **heap** sections can shrink and grow dynamically during execution
  - Stack changes caused by:
    - **Call function**: its activation record pushed onto the stack
    - **Return from the function**: its activation record popped from the stack
  - Heap changes caused by:
    - **Grows** as dynamically allocated memory
    - **Shrinks** when memory is returned to the system

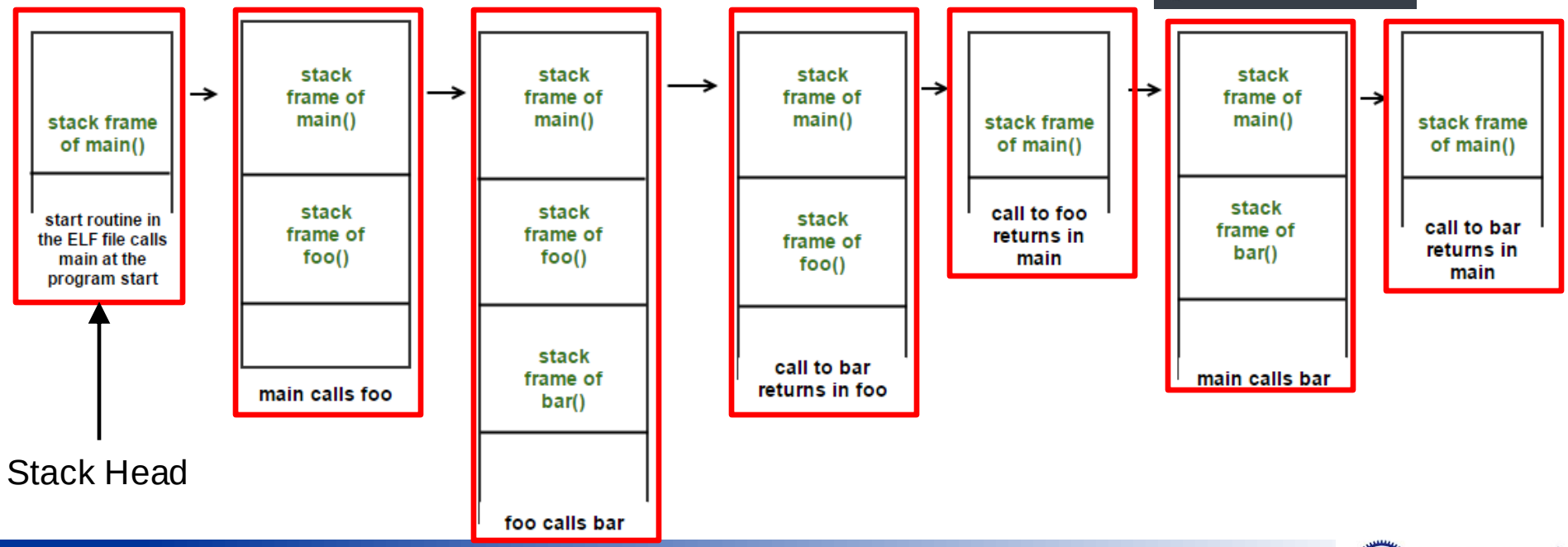


Process memory layout

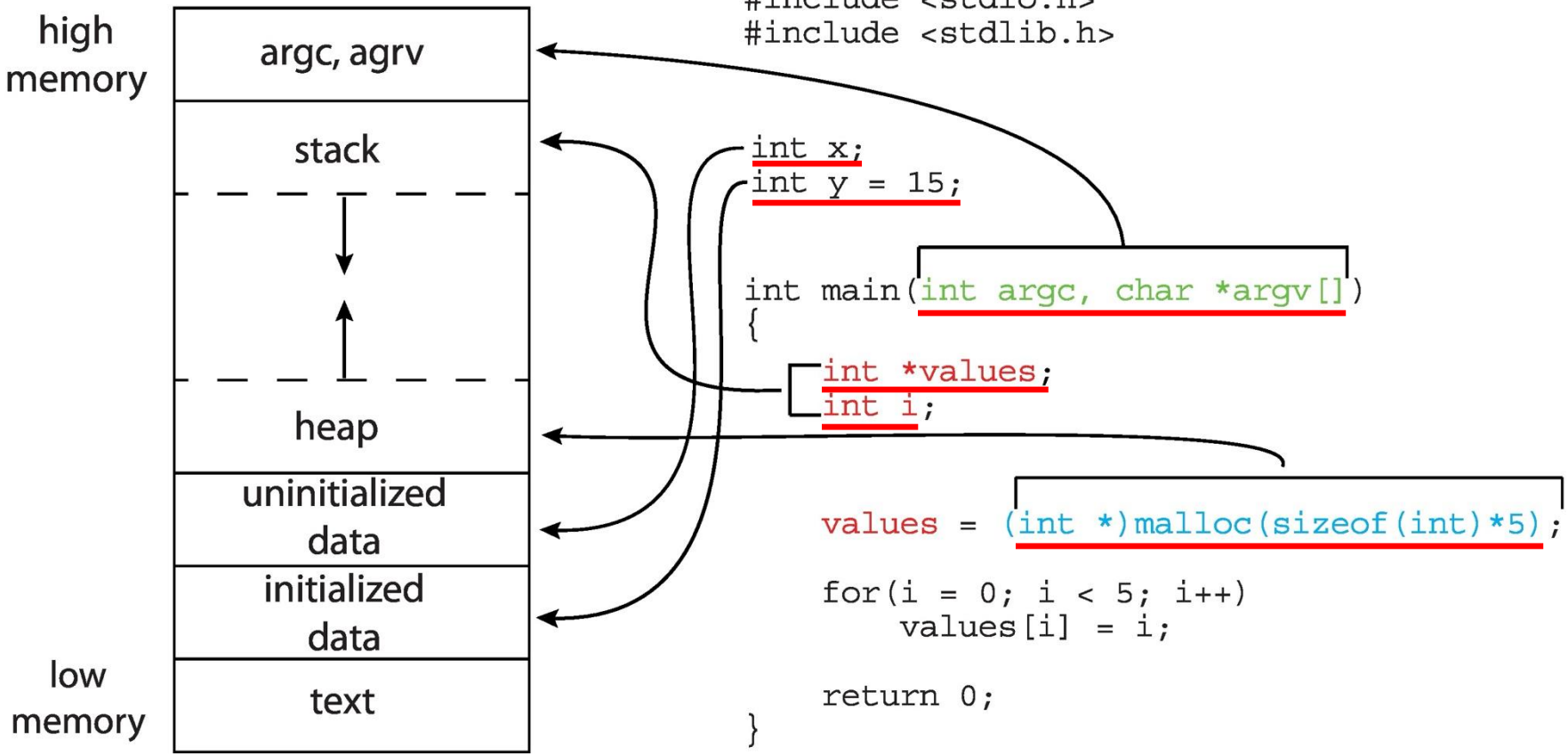
Stack for Saving Function Call Status

- Stacks changes of the right code example.
- Stack frame stores function status:
  - Arguments passed by the caller
  - Local variables
  - Return address to the caller
  - Exception handling information (if any)

```
int main()
{
  foo();
  bar();
}
void foo()
{
  // do something
  bar();
}
void bar()
{
  // do something
}
```

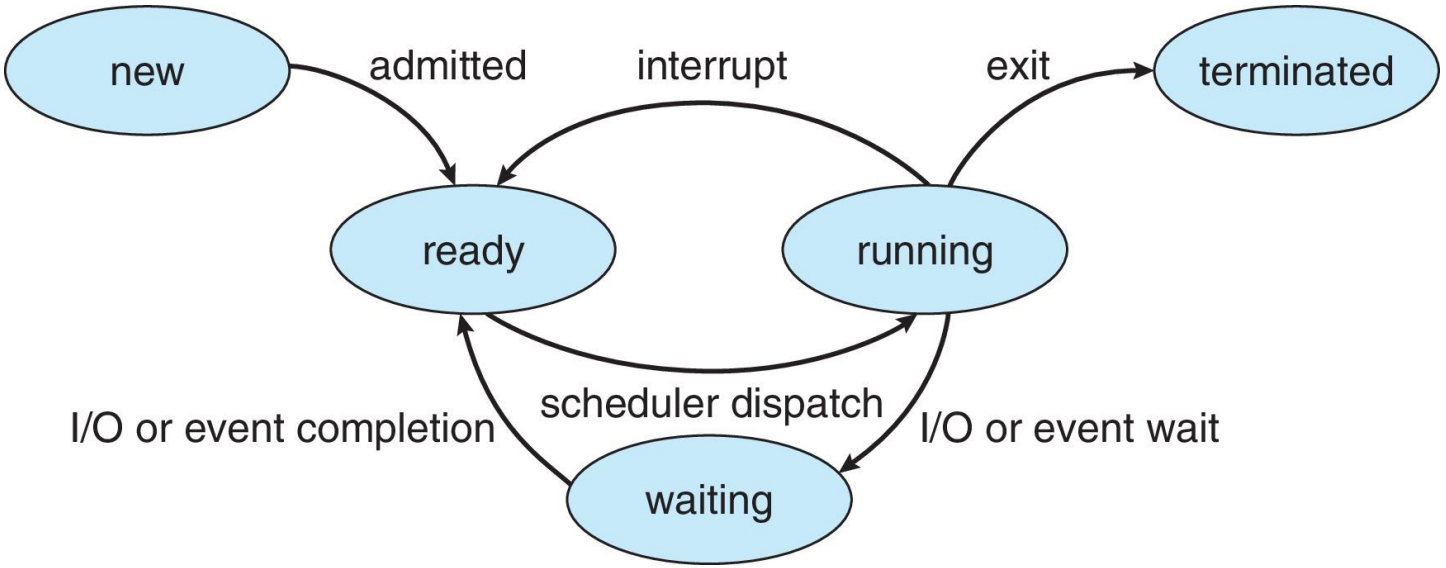


Memory Layout of a C Program



Process State

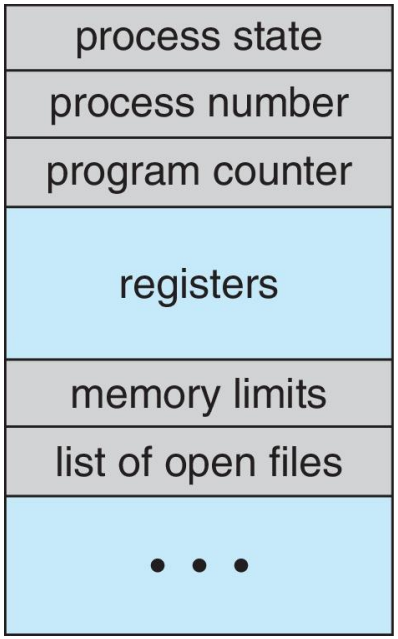
- As a process executes, it changes state
  - **New**: The process is being created
  - **Ready**: The process is waiting to be assigned to a processor
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Terminated**: The process has finished execution



Process Control Block (PCB)

Information associated with each process (also called **task control block**)

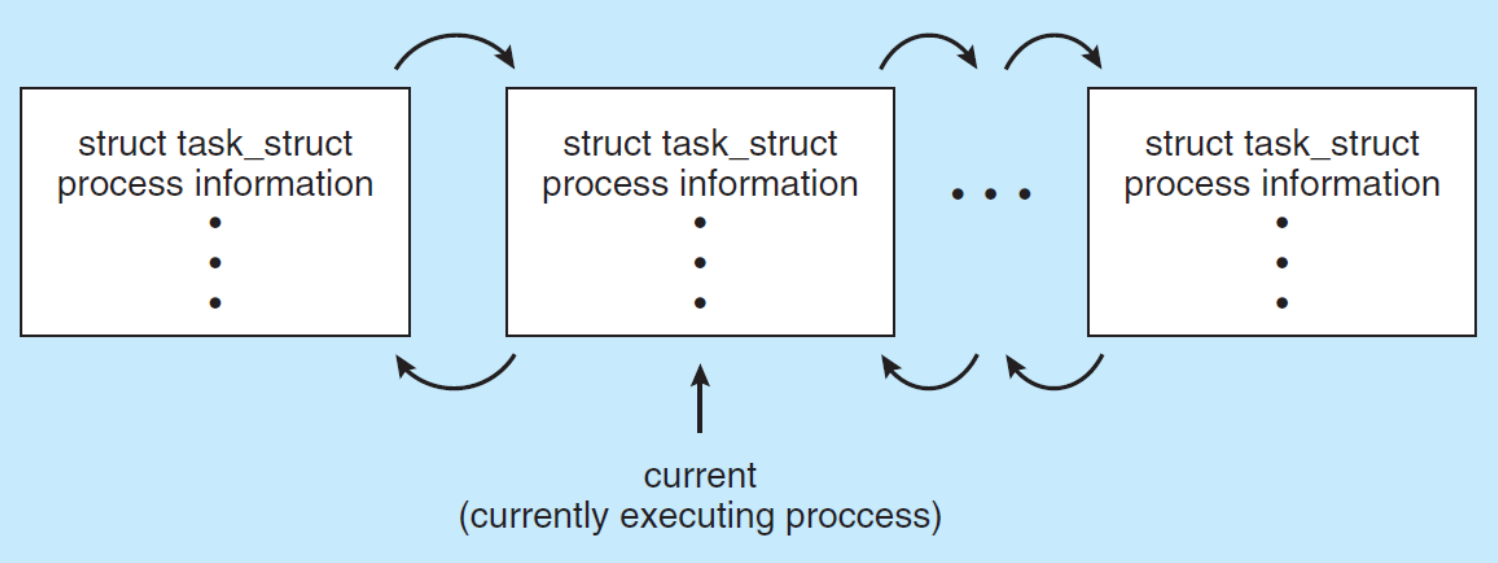
- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information** - priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to the process, list of open files



Linux Process Representation

Represented by the C structure **task_struct**

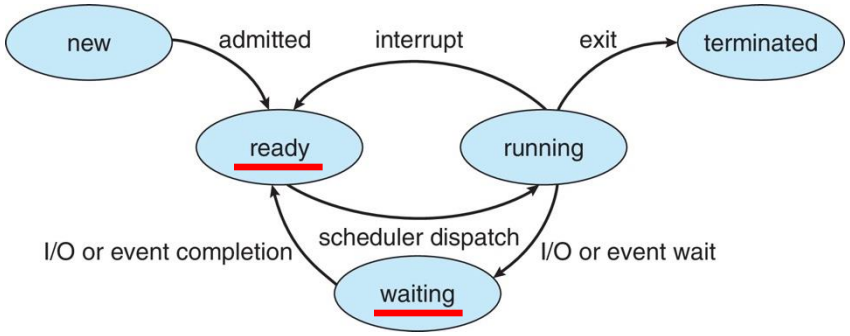
```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */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

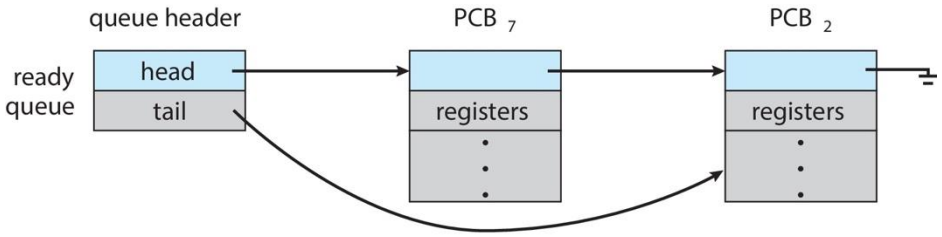


Process Scheduling

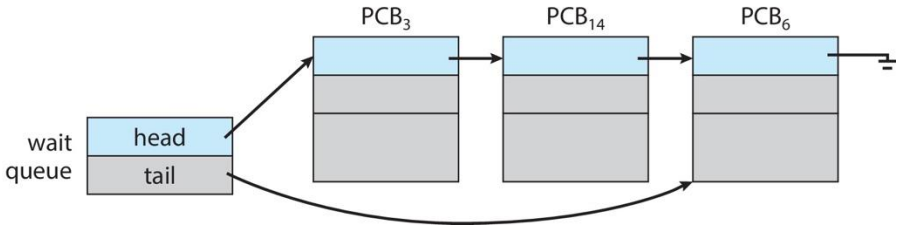
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- **Goal**: Maximize CPU use, quickly switch processes onto CPU core
- Maintains scheduling queues of processes
  - **Ready queue** – set of processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate between different queues



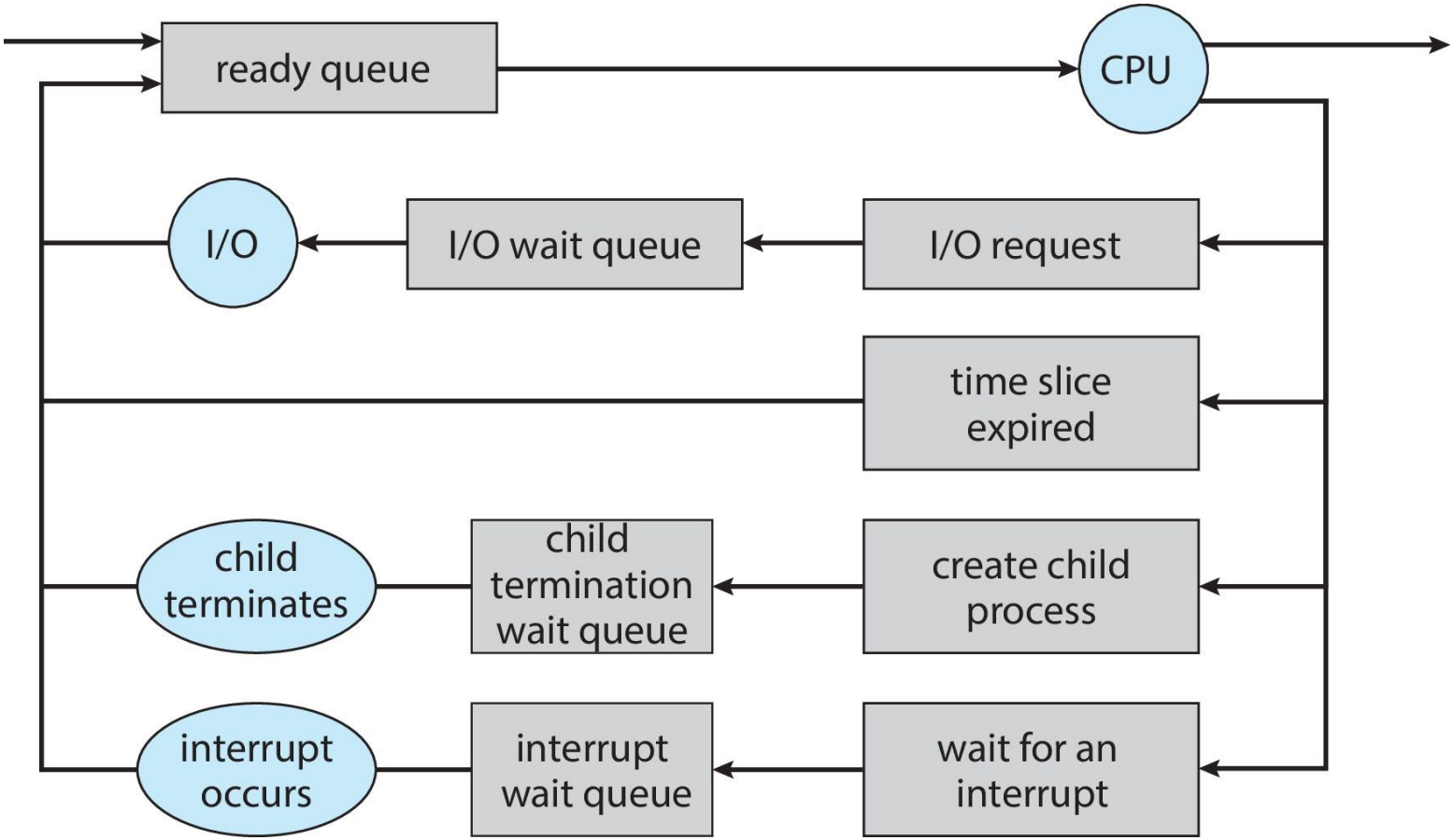


Ready Queue



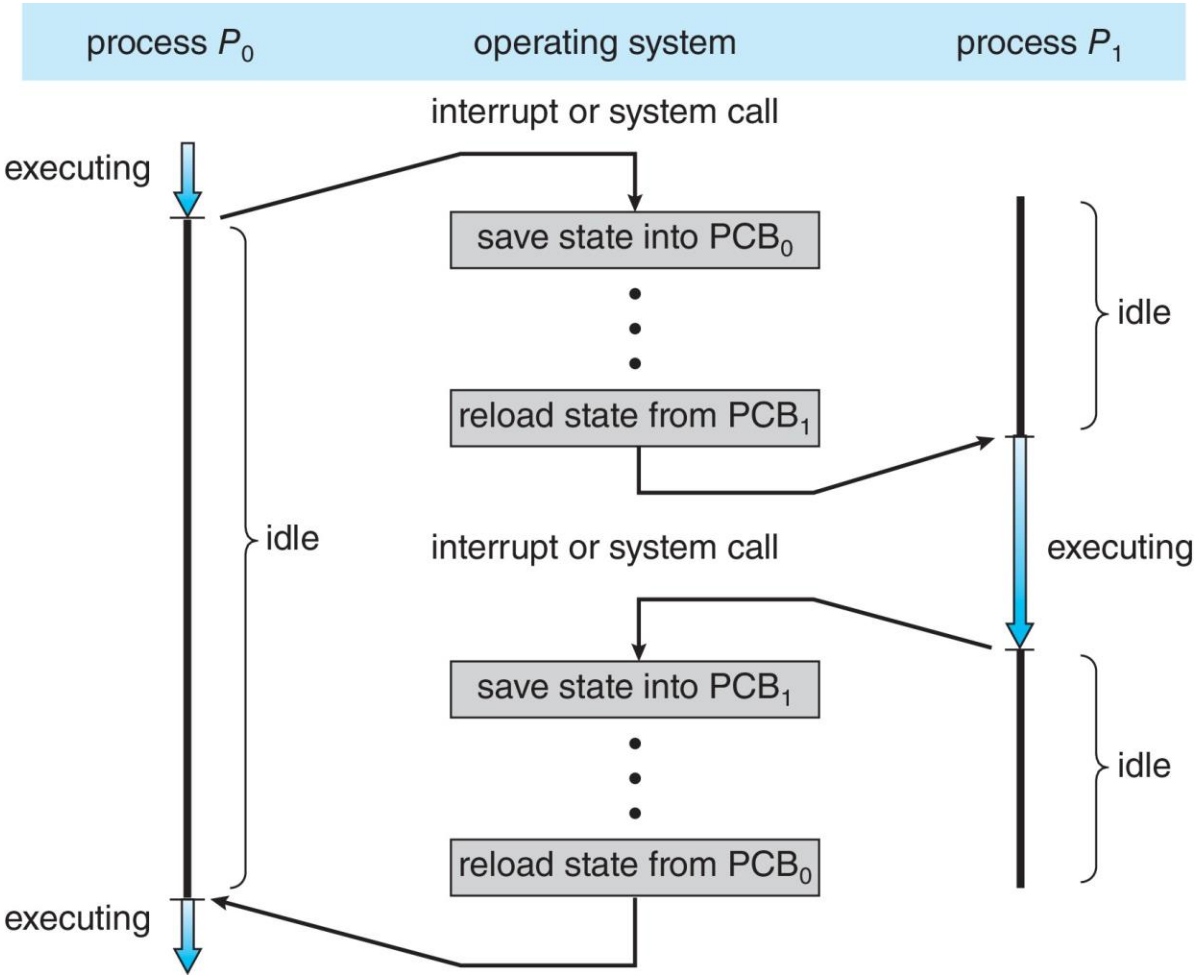
Wait Queue

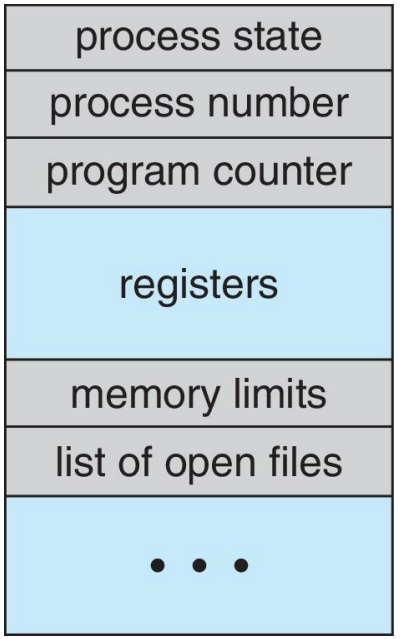
Representation of Process Scheduling



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

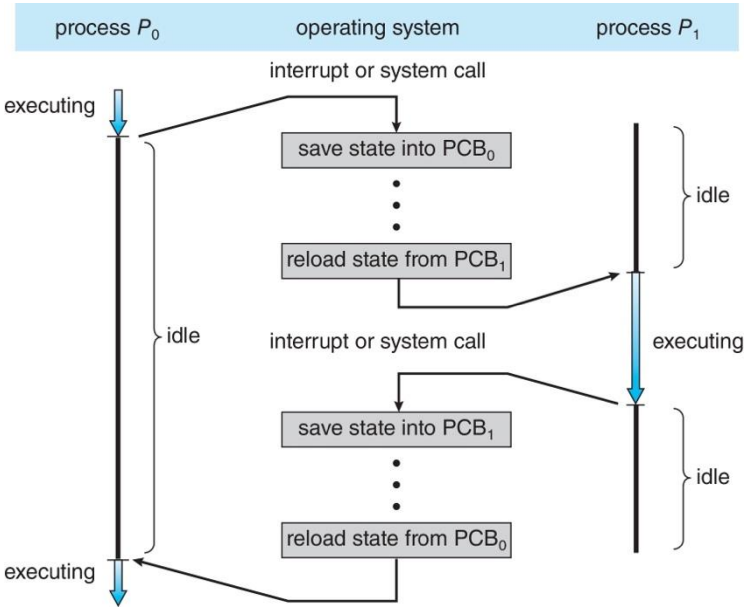




PCB

Context Switch

- When CPU switches to another process, the system must
  - **save the state** of the old process
  - **load the saved state** for the new process via a **context switch**
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch time
- Time-dependent on hardware support
  - Some hardware provides multiple register sets per CPU ➔ multiple contexts loaded at once
  - Reduce context switch overhead



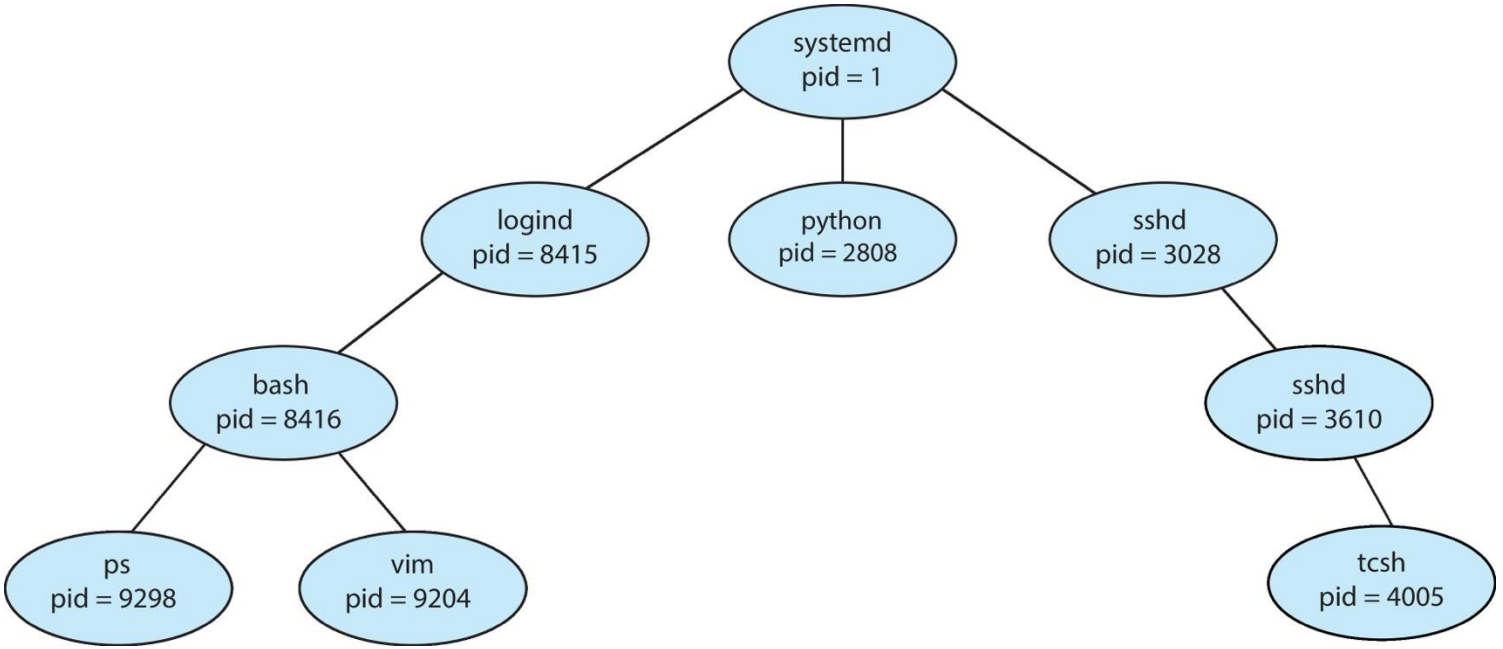
Schedulers

- Two levels of schedulers:
  - **Long-term scheduler** (or job scheduler)
    - Selects which processes should be brought into the ready queue
    - Invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - **Short-term scheduler** (or CPU scheduler)
    - Selects which process should be executed next and allocates CPU
    - Sometimes the only scheduler in a system
    - Invoked very frequently (milliseconds) ⇒ (must be fast)
- Two types of processes:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; some long CPU bursts

Operations on Processes

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**)



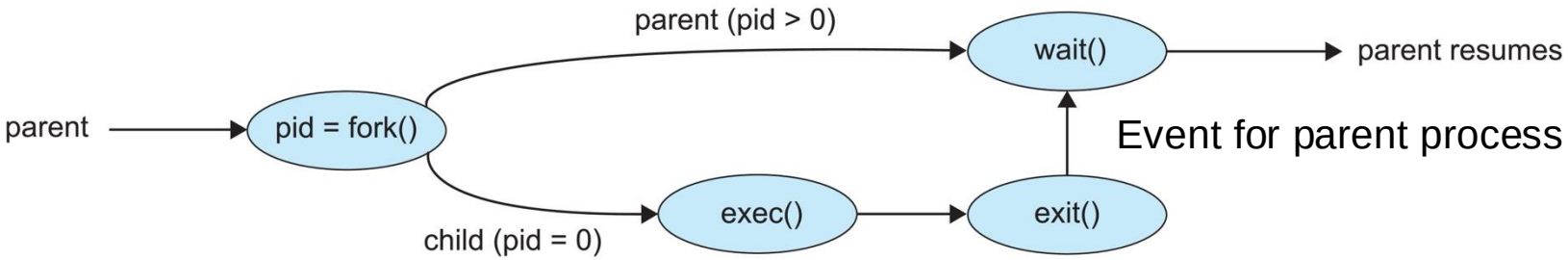
Tree of Process Relations

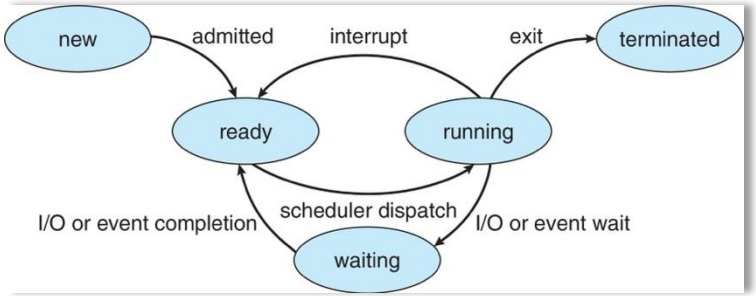
Parent and Child Process Relations

- 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
- Address space options
  - Child duplicate the address space of parent
  - Child has a program loaded into it

Process Creation in UNIX

- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program
  - Parent process calls `wait()` to wait for the child to terminate





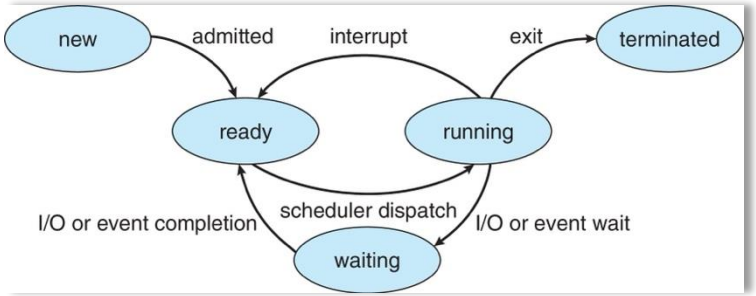
Reference: Process State Transitions

C Program Forks Child Process

- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i = 1;
    pid_t  pid;
    /* fork another 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 */
          wait (NULL);
          printf ("Child Complete.");
    }
    return 0;
}
```

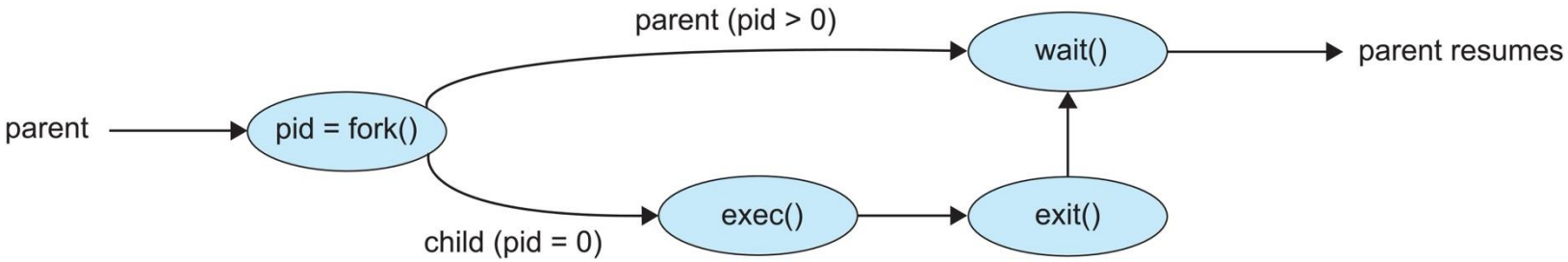
Different executions by processes



Reference: Process State Transitions

Process Termination

- Process executes last instruction 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 the OS
- Parent may terminate the execution of its child 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 OS does not allow a child to continue if its parent terminates
    - **Cascading termination:** All children, grandchildren, etc., are terminated.

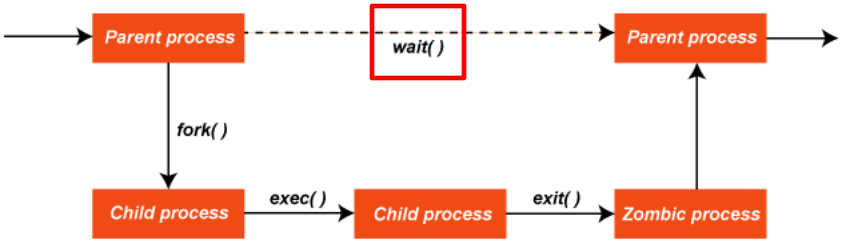


Process Termination

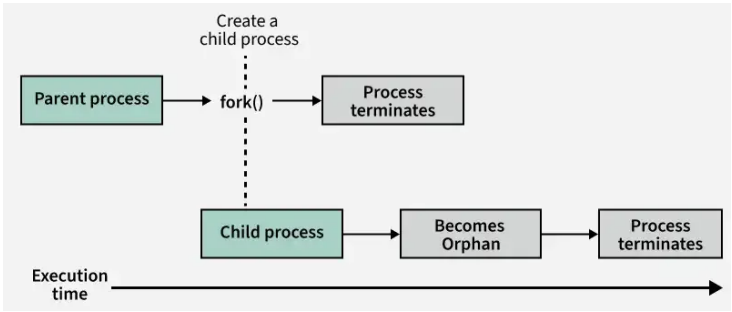
- The parent process may wait for termination of a child process by using the `wait()` system call.
  - The call returns status information and the pid of the terminated process

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

- If a child process terminated, but its parent process does not know its exit status, the child is a **zombie process (僵尸进程)**
  - Zombie = child finished execution + parent still exists but has not waited
- If parent terminated without `wait()`, child process is an **orphan (孤儿进程)**
  - Orphan = child still running + parent is gone



Zombie Process

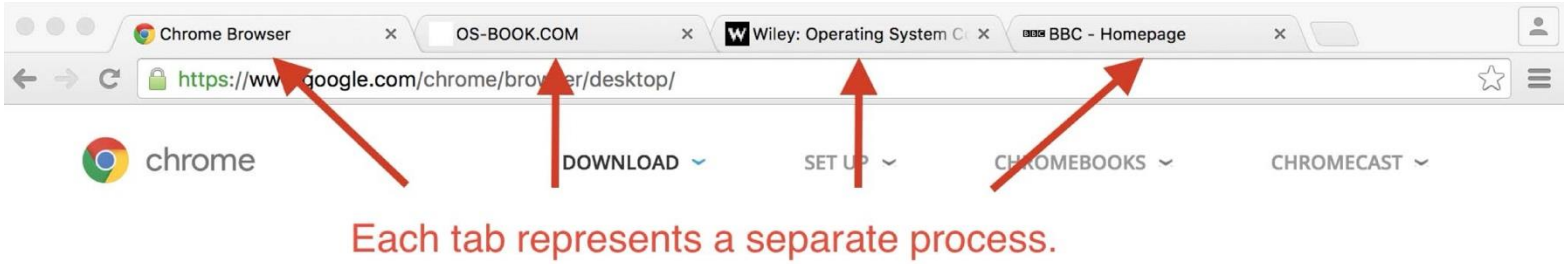


Orphan Process

Interprocess Communication

Motivation: Multiprocess Chrome Browser

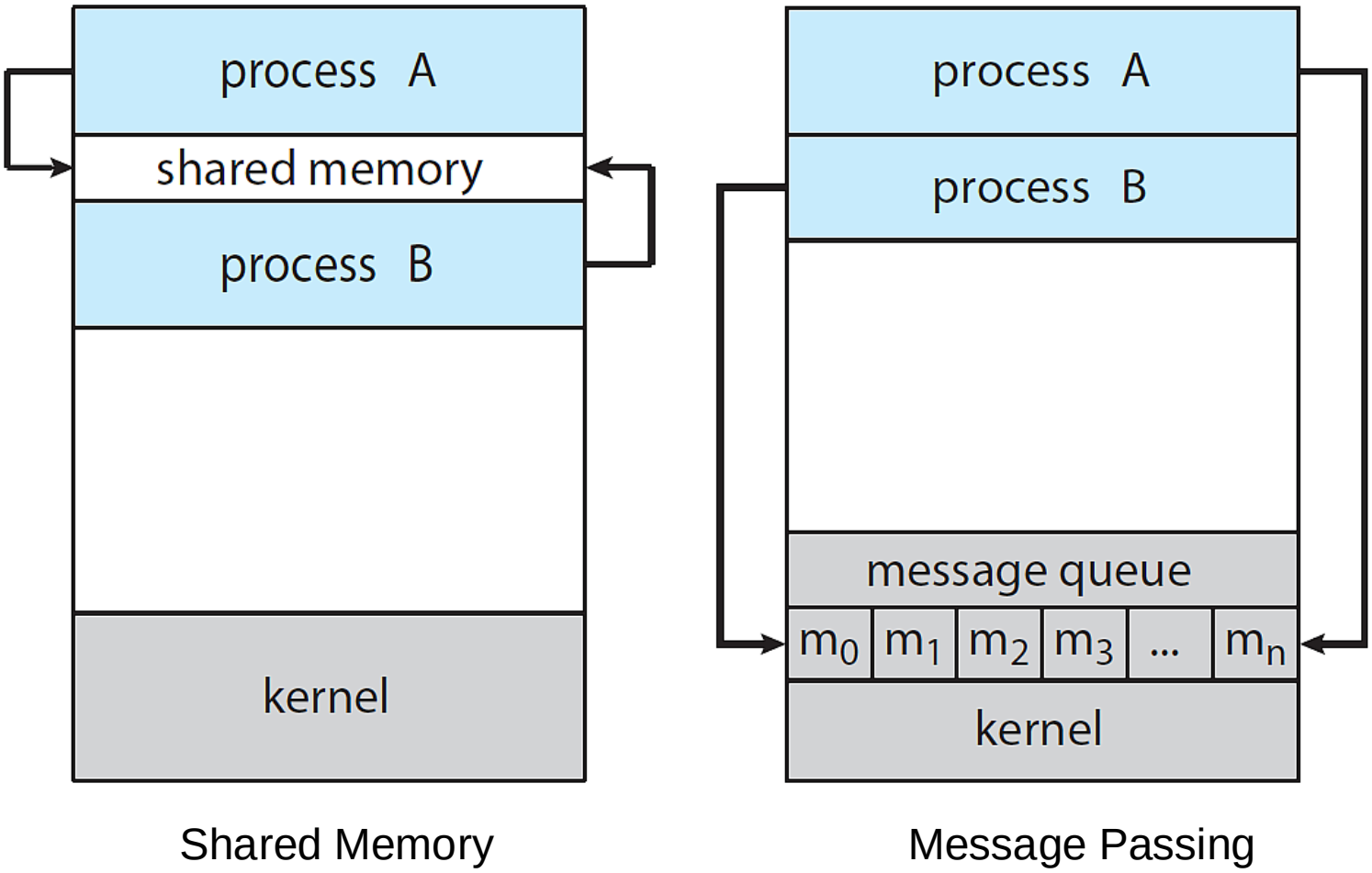
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process (渲染进程) renders web pages, deals with HTML, Javascript.
    - A new renderer created for each website opened
  - **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（共享内存）**
    - **Message passing（消息传递）**

Interprocess Communication Models



Shared Memory

Message Passing

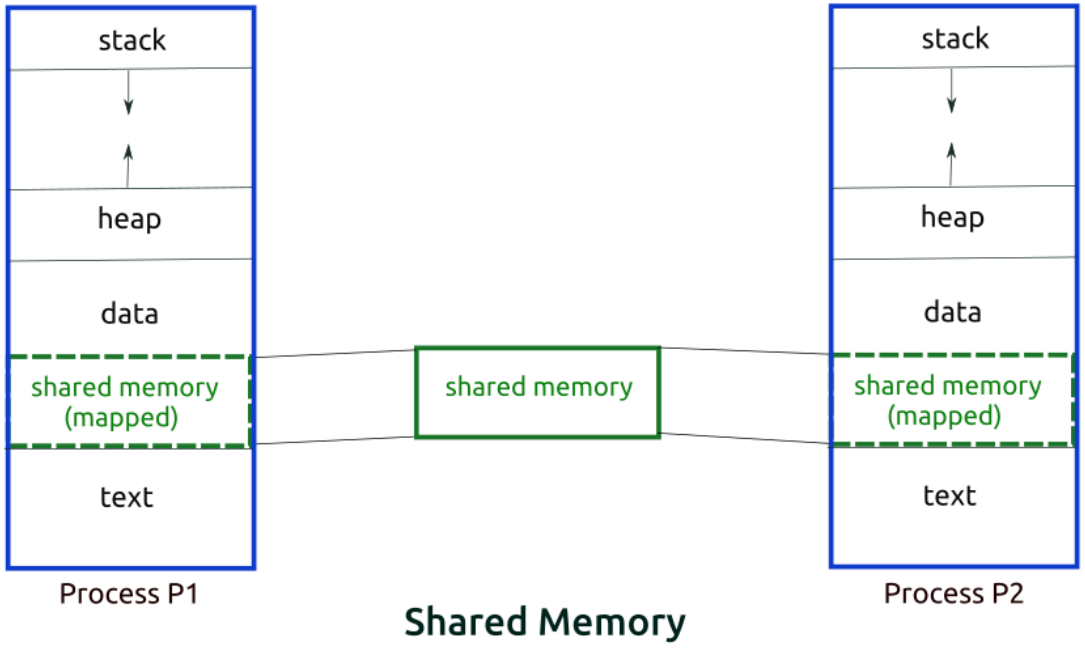
IPC Model Comparison

	Shared Memory	Message Passing
Concept	Allows multiple processes to access common data structures by mapping the same portion of physical memory into their address spaces.	Processes communicate by sending and receiving messages via a communication channel managed by the operating system.
Speed	**Faster**	Slower
Complexity	More complex in managing concurrent access to shared data and ensure data consistency	**Simpler to implement** as the message system abstracts away low-level details
Synchronization	Requires explicity sync mechanisms like semaphores, mutexes, or atomic operations to avoid conflicts.	**Implicit sync** as messaging system ensures msgs sent and received in controlled order
Scalability	Less scalable	**More scalable**, especially in distributed systems
Examples	“mmap()” in UNIX	Sockets, pipes, and message queues.

Interprocess Communication: Shared Memory

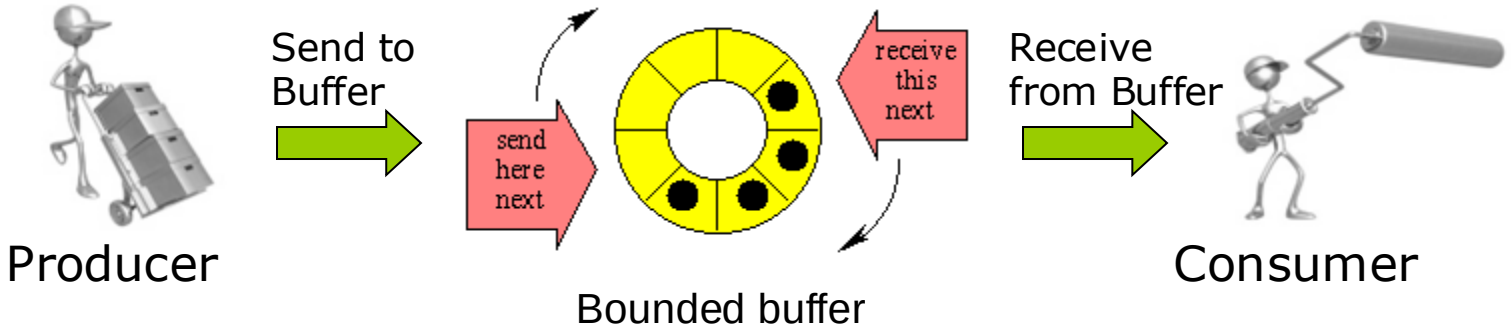
IPC – Shared Memory

- 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 issue is to provide mechanism allowing the user processes to **synchronize the actions** when they access shared memory.



Producer-Consumer Problem

- Paradigm for cooperating processes:
  - Producer process produces information that is consumed by a consumer process
- Two variations:
  - **Unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **Bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume



Bounded-Buffer – Shared-Memory Solution

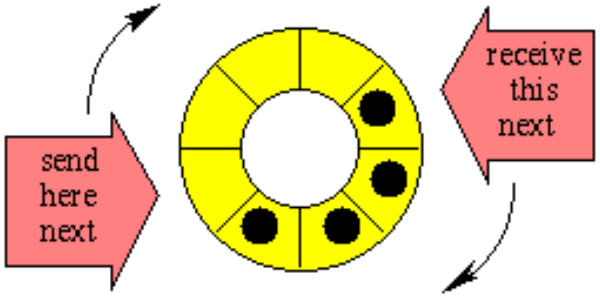
- Shared data

```
#define BUFFER_SIZE 10

typedef struct {
        . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Shared-Memory Solution

Producer Process

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



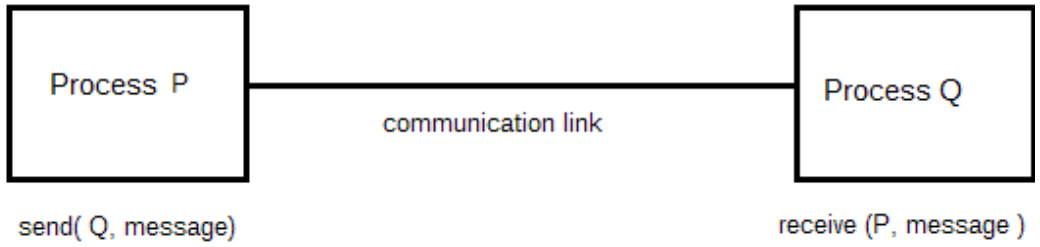
Consumer Process

```
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: Message Passing

Message Passing

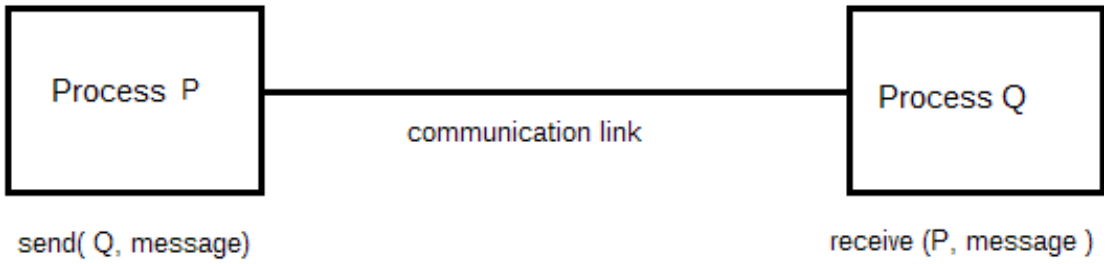
- In message passing systems, processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Two types of message passing:
  - Direct communication
  - Indirect communication



Message Passing: 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
  - The processes need to know only each other's identity to communicate
  - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - There exists exactly one link between each process pair
  - The link may be unidirectional, but is usually bi-directional

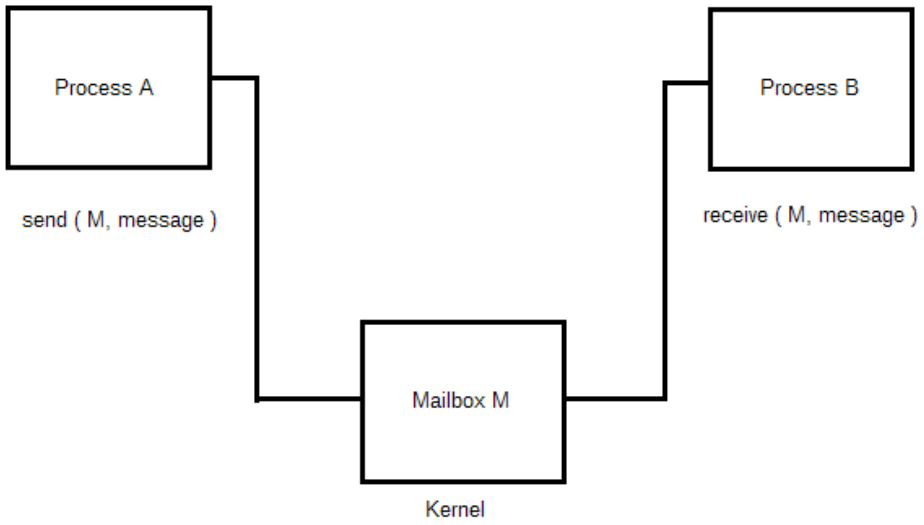


Message Passing: Indirect Communication

- Messages are directed to and received from **mailboxes** (also referred to as **ports**)
  - Each mailbox has a unique ID
  - Processes can communicate only if they share a mailbox
- Primitives are defined as:

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

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



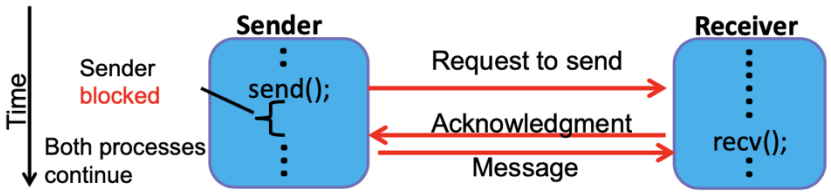
Message Passing: Indirect Communication

- Mailbox sharing has confusions
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message? Not known.

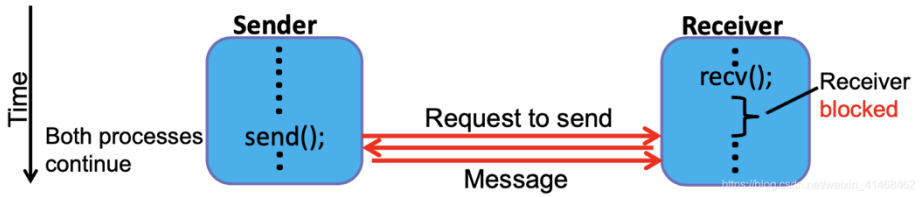
- Solutions to avoid confusion:
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.
    - Sender is notified who the receiver was

Message Passing: 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



Sender-side Blocking



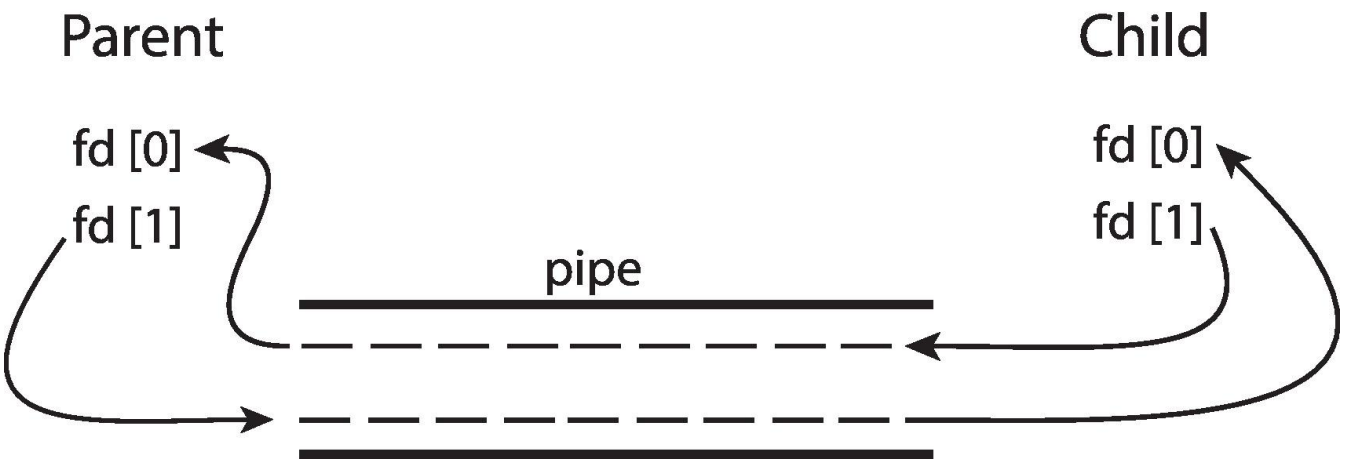
Receiver-side Blocking

Message Passing Example: UNIX Pipes

- Acts as a conduit (管道) allowing two processes to communicate
- Issues to consider:
  - 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?
- Two types:
  - **Ordinary pipes**
    - Cannot be accessed from outside the process that created it.
    - A parent process creates a pipe and uses it to communicate with its child.
  - **Named pipes**
    - Can be accessed without a parent-child relationship.

Ordinary Pipes

- **Ordinary Pipes (普通管道)** 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 unidirectional
- Require parent-child relationship between communicating processes



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

Using Pipe – Part 1

mypipe[0]	read-end
mypipe[1]	write-end

1. Create a pipe and check for errors

```
int mypipe[2];
if (pipe(mypipe)) {
    fprintf (stderr, "Pipe failed.\n");
    return -1;
}
```

2. Fork your threads

3. Close the pipes you don't need in that thread
   - reader should `close(mypipe[1])`
   - writer should `close(mypipe[0])`

Using Pipe – Part 2

4. The writer should write the data to the pipe

```
write(mypipe[1], &c, 1)
```

5. The reader reads from the data from the pipe:

```
while (read(mypipe[0],&c,1)>0) {
    //do something, loop will exit when WRITER closes pipe
}
```

6. When writer is done with the pipe, close it

```
close(mypipe[1]); // EOF is sent to reader
```

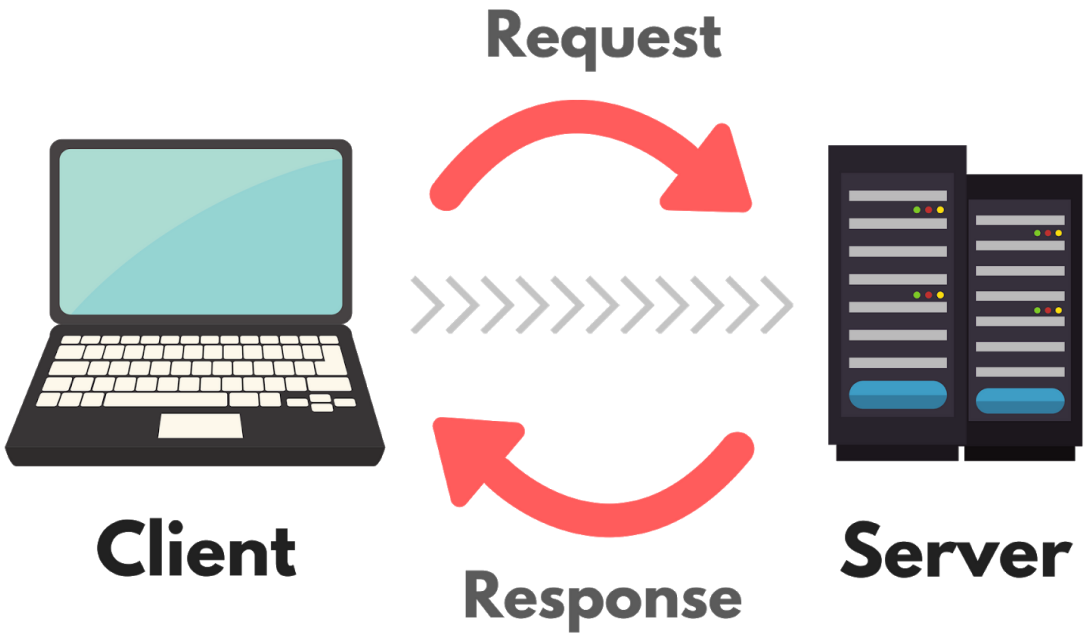
7. When reader receives EOF from closed pipe, close the pipe and exit the polling loop

```
close(mypipe[0]); // All pipes should be closed now
```

Communication in Client-Server Systems

Communication in Client-Server Systems

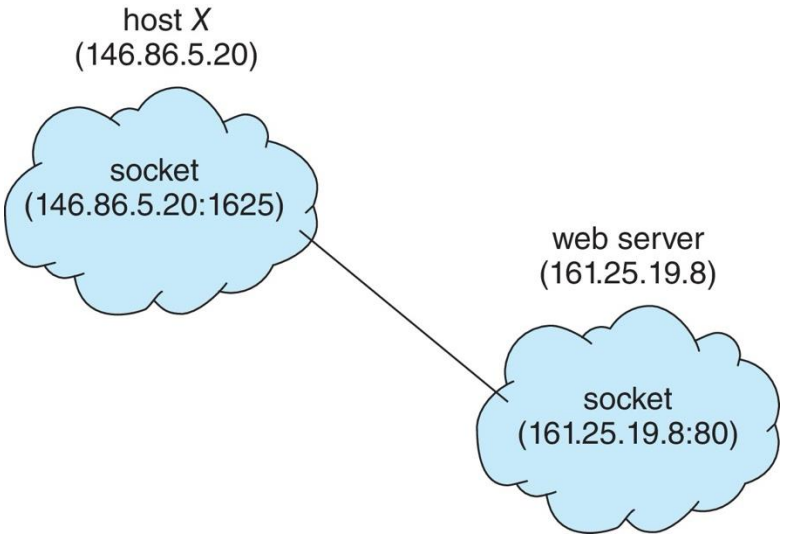
- The previous **shared memory** and **message-passing** approaches work in client-server systems.
- Here we introduce another two communication strategies:
  - Sockets
  - Remote procedure call (RPC)



Client-Server Communication: Sockets

- A **socket** is defined as an endpoint for communication
  - Communication consists of a pair of sockets
  - They communicate over a network between distributed devices.

- 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**
  - 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

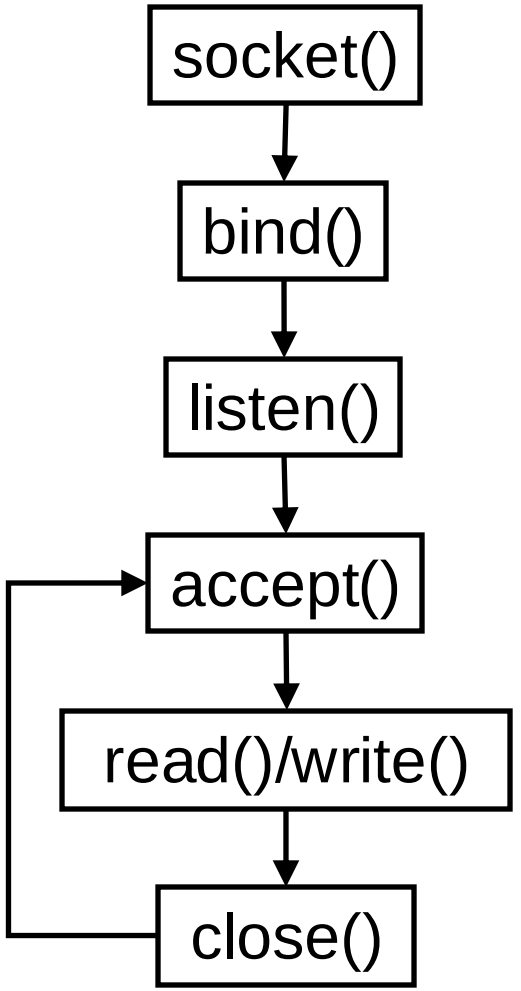


Standard Port Numbers

Port Number	Service name	Transport protocol	Description
7	Echo	TCP, UDP	Echo service
20	FTP-data	TCP, SCTP	File Transfer Protocol data transfer
21	FTP	TCP, UDP, SCTP	File Transfer Protocol (FTP) control connection
22	SSH-SCP	TCP, UDP, SCTP	Secure Shell, secure logins, file transfers (scp, sftp), and port forwarding
23	Telnet	TCP	Telnet protocol—unencrypted text communications
25	SMTP	TCP	Simple Mail Transfer Protocol, used for email routing between mail servers
53	DNS	TCP, UDP	Domain Name System name resolver
69	TFTP	UDP	Trivial File Transfer Protocol
80	HTTP	TCP, UDP, SCTP	Hypertext Transfer Protocol (HTTP) uses TCP in versions 1.x and 2. HTTP/3 uses QUIC, a transport protocol on top of UDP

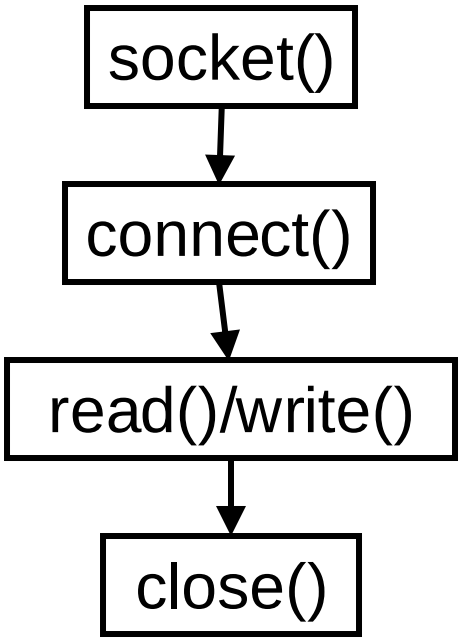
Creation Steps: Server Side

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call (This call typically blocks until a client connects with the server)
5. Send and receive data with read() and write() system calls
6. Close connection with close() system call

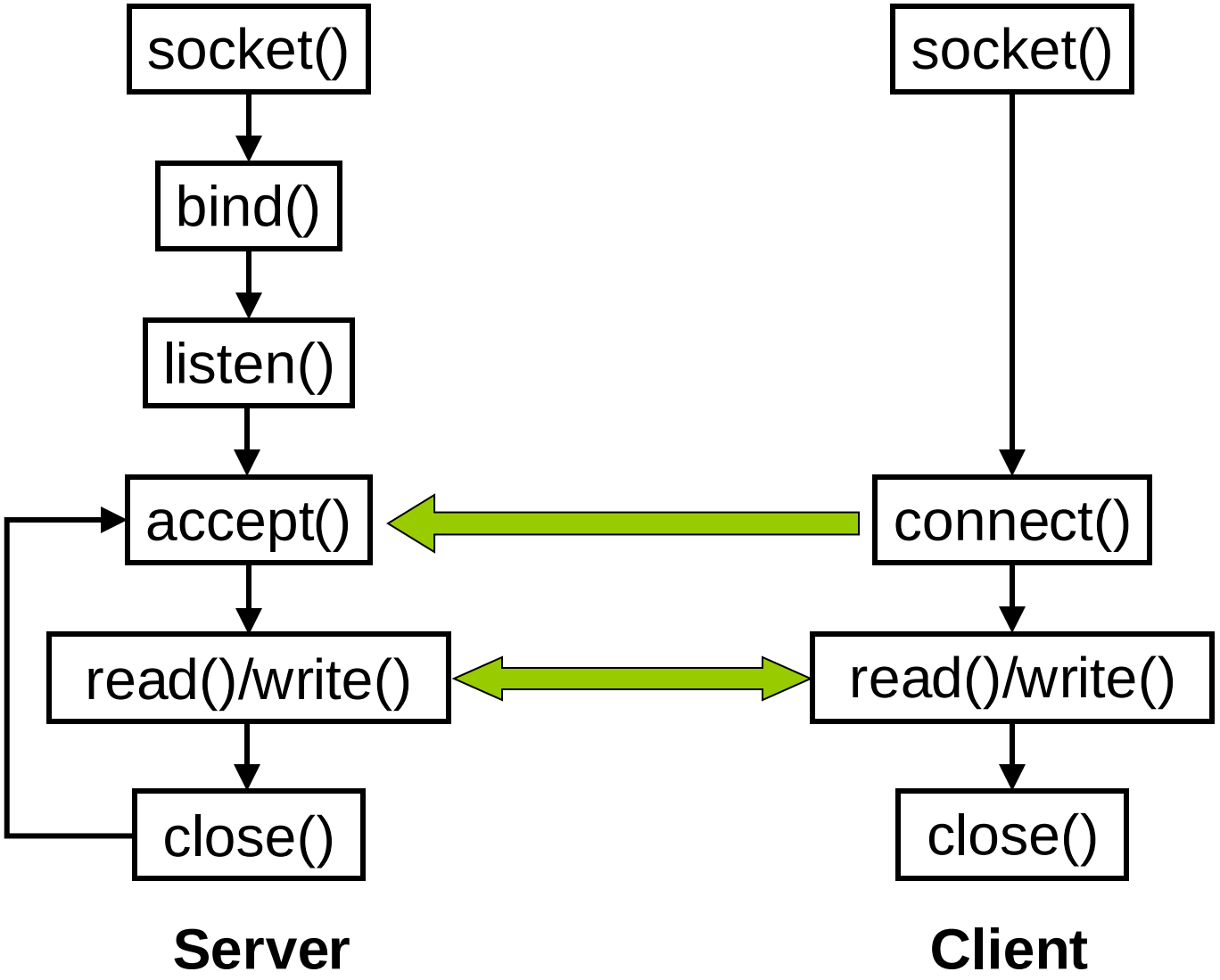


Creation Steps: Client Side

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data with read() and write() system calls.
4. Close the socket with close() system call

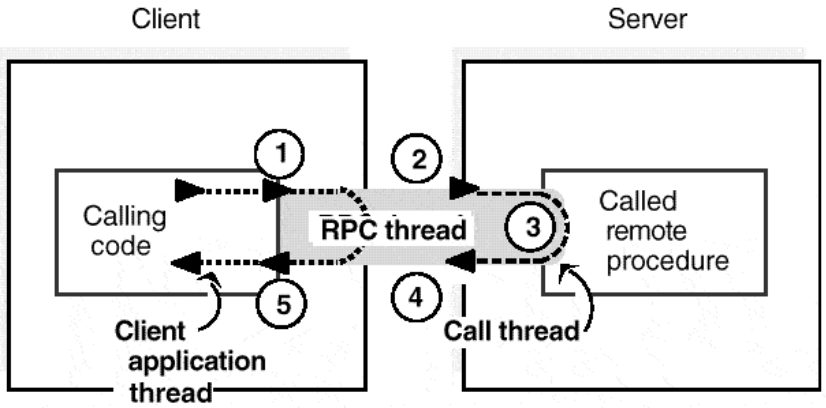


Interaction Between Client and Server

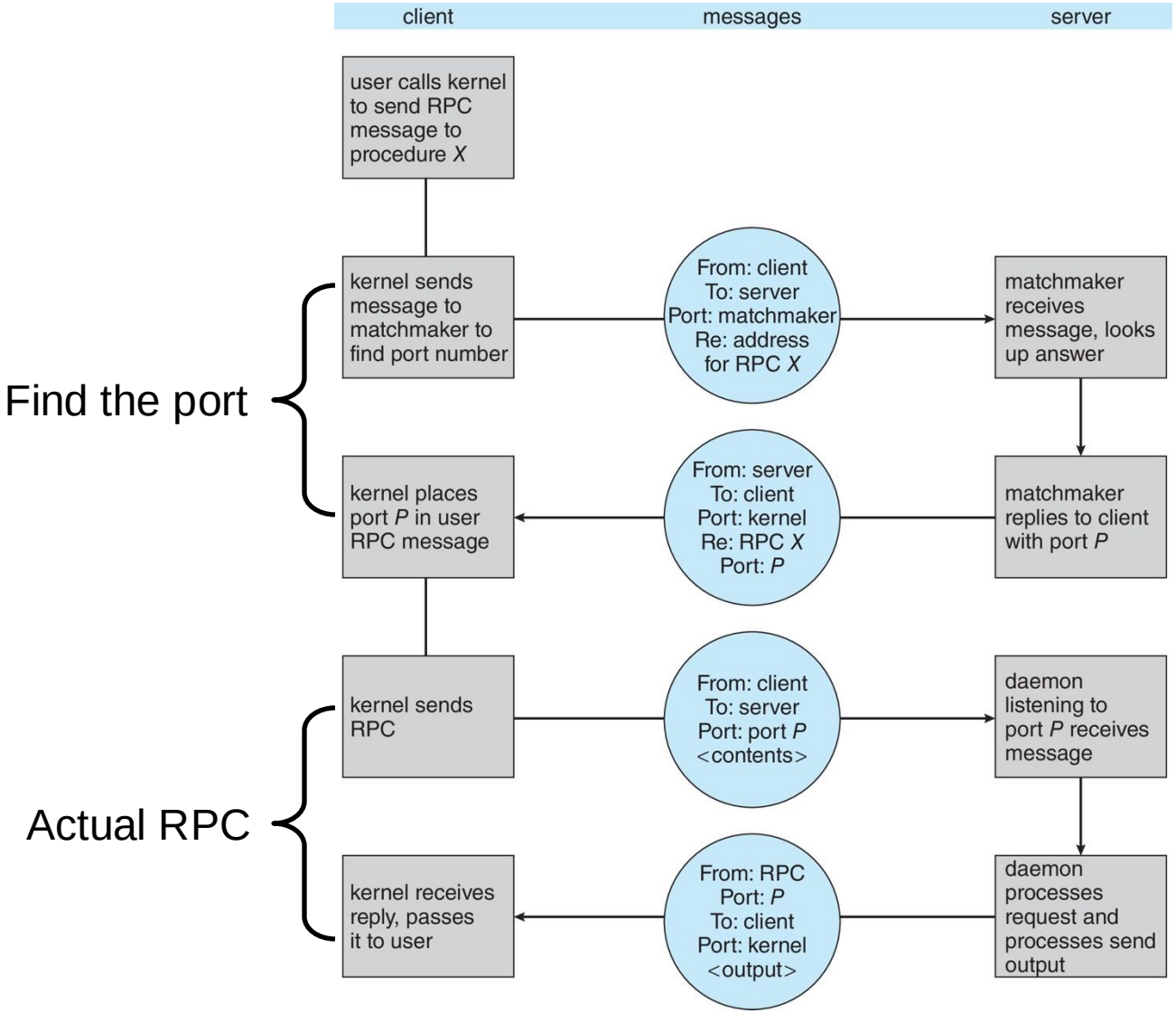


Client-Server Communication: RPC

- **Remote procedure call (RPC):** A function can be invoked on another process that may reside on a separate computer.
  - They also use ports for service differentiation
  - Client calls a procedure on server
- Stubs（存根）:
  - Client-side proxy for the actual procedure on the server
  - The client-side stub locates the server and collects the parameters
  - The server-side stub receives this message, unpacks the collected parameters, and performs the procedure on the server
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***



Execution of RPC



Summary

- Process is a program in execution (active entity vs. passive entity)
- A process memory is represented by 4 sections:

  (1) text, (2) data, (3) heap, and (4) stack
- Use process control block (PCB) as the kernel structure to represent a process
- OS performs context switch when changing from one process to another
- Two interprocess communication mechanisms:
  - Shared memory
  - Message passing
- Two client-server communication mechanisms:
  - Sockets
  - Remote procedure calls (RPC)

Homework

- Reading
  - Chapter 3