國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Institute of Artificial Intelligence Innovation
Department of Computer Science

Operating System

Lecture 03: Processes Concept

Shuo-Han Chen 陳碩漢
shch@nycu.edu.tw

Wed. 10:10 - 12:00 EC115 +
Fri. 11:10 – 12:00 Online

Course Schedule

W	Date	Lecture	Online	Homework
1	Sept. 4	Lec00: Couse Overview & Historical Prospective		
2	Sept. 11	Lec01: Introduction	V	
3	Sept. 18	Lec02: OS Structure	V	HW01
4	Sept. 25	Lec03: Processes Concept	X	
5	Oct. 2	Lec07: Memory Management	V	
6	Oct. 9	Lec08: Virtual Memory Management	V	HW02
7	Oct. 16	Lec04: Process Scheduling	V	
8	Oct. 23	School Midterm Exam		
9	Oct. 30	Lec05: Process Synchronization	V	
10	Nov. 6	Lec06: Deadlocks	V	HW03
11	Nov. 13	Lec09: File System Interface	V	
12	Nov. 20	School Event – No class		
13	Nov. 27	Lec10: File System Implementation	V	HW04
14	Dec. 4	Lec11: Mass Storage System	V	
15	Dec. 11	Lec12: IO Systems	V	
16	Dec. 18	School Final Exam		

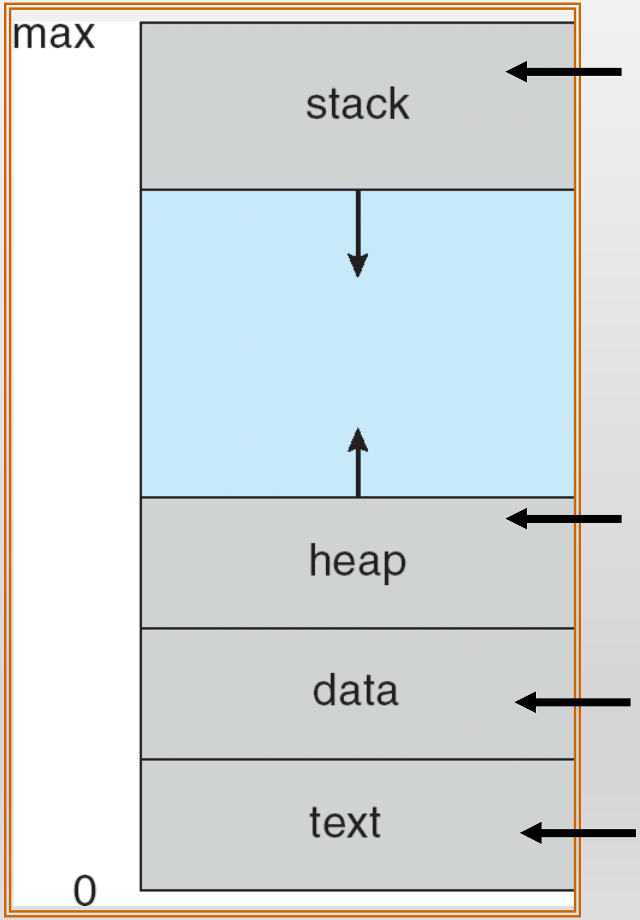
Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Process Concept

- An operating system concurrently executes a variety of programs (e.g Web browser, text editor, etc)
  - Program – passive entity: binary stored in disk
  - Process – active entity: a program in execution in memory
- A process includes:
  - Code segment (text section)
  - Data section– global variables
  - Stack –temporary local variables and functions
  - Heap –dynamic allocated variables or classes
  - Current activity (program counter, register contents)
  - A set of associated resources (e.g. open file handlers)

Process in Memory



temporary data (e.g. function parameters, return addresses, local variables)

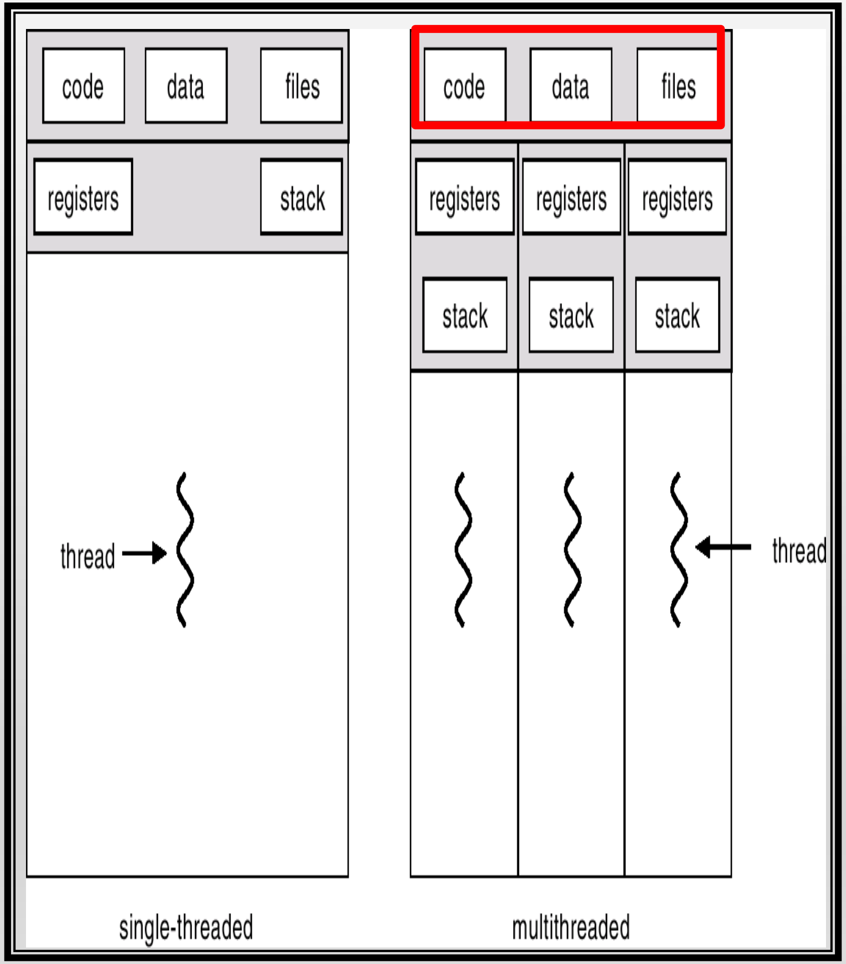
dynamic allocation (e.g. class object, pointer object)

global variables

code

Threads

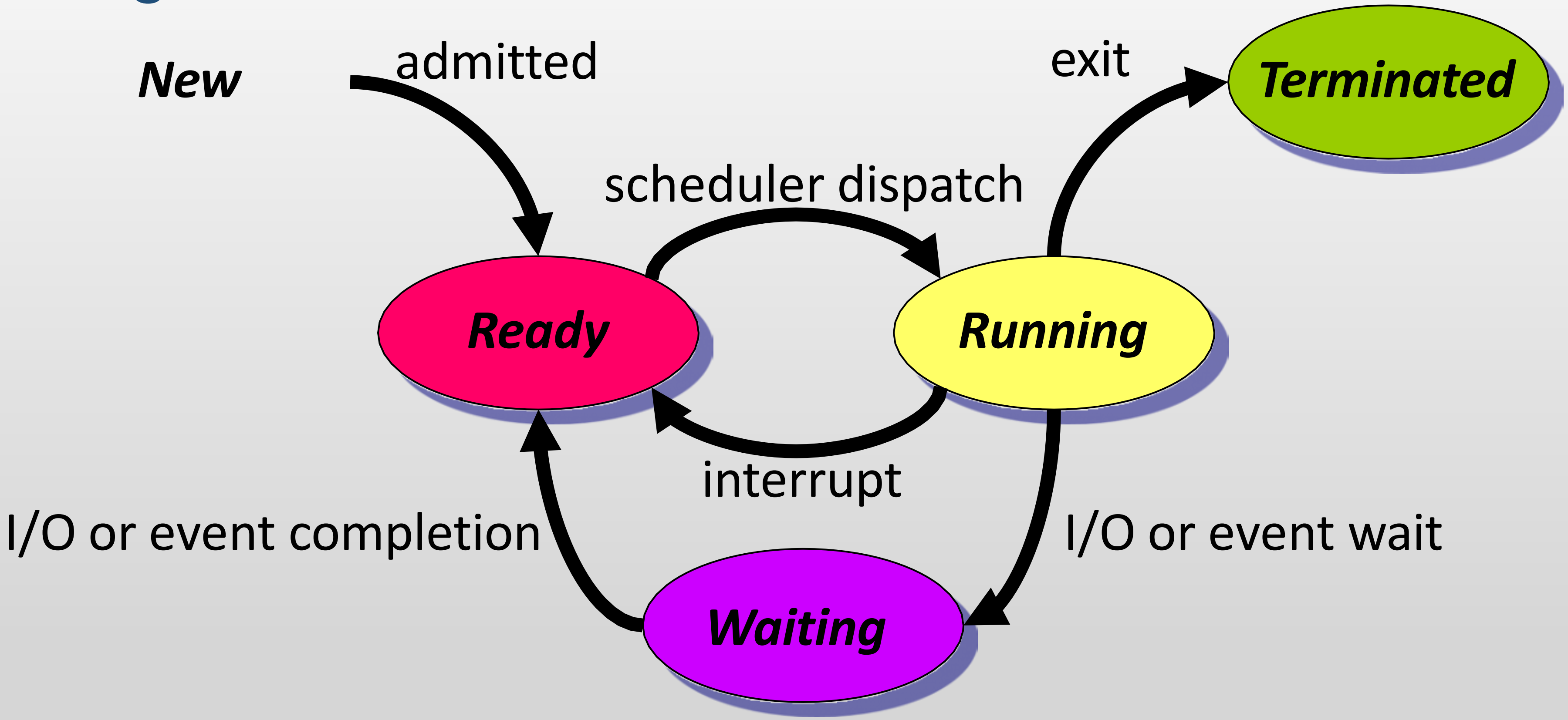
- A.k.a lightweight process: basic unit of CPU utilization
- All threads belonging to the same process share
  - code section, data section, and OS resources (e.g. open files and signals)
- But each thread has its own
  - thread ID, program counter, register set, and a stack



Process State

- States
  - New: the process is being created
  - Ready: the process is in the memory waiting to be assigned to a processor
  - Running: instructions are being executed by CPU
  - Waiting: the process is waiting for events to occur
  - Terminated: the process has finished execution

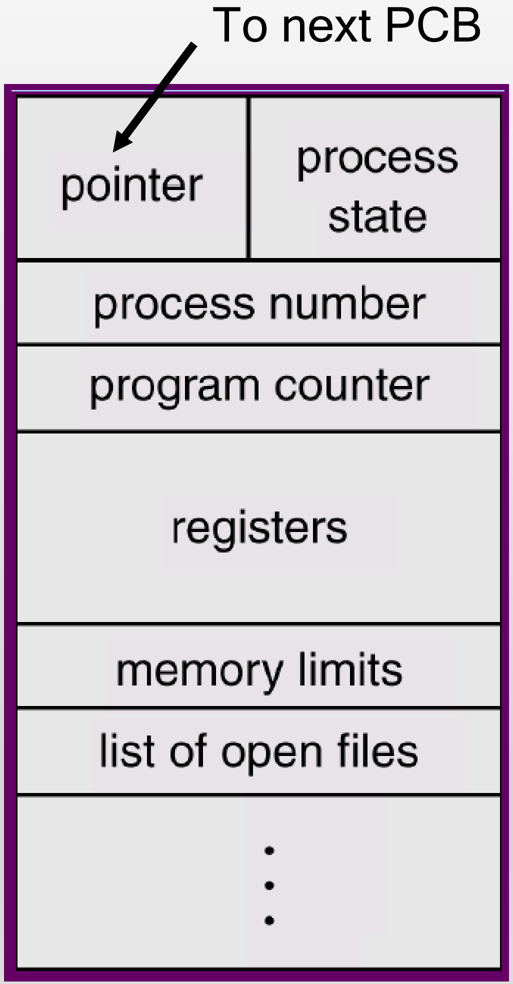
Diagram of Process State



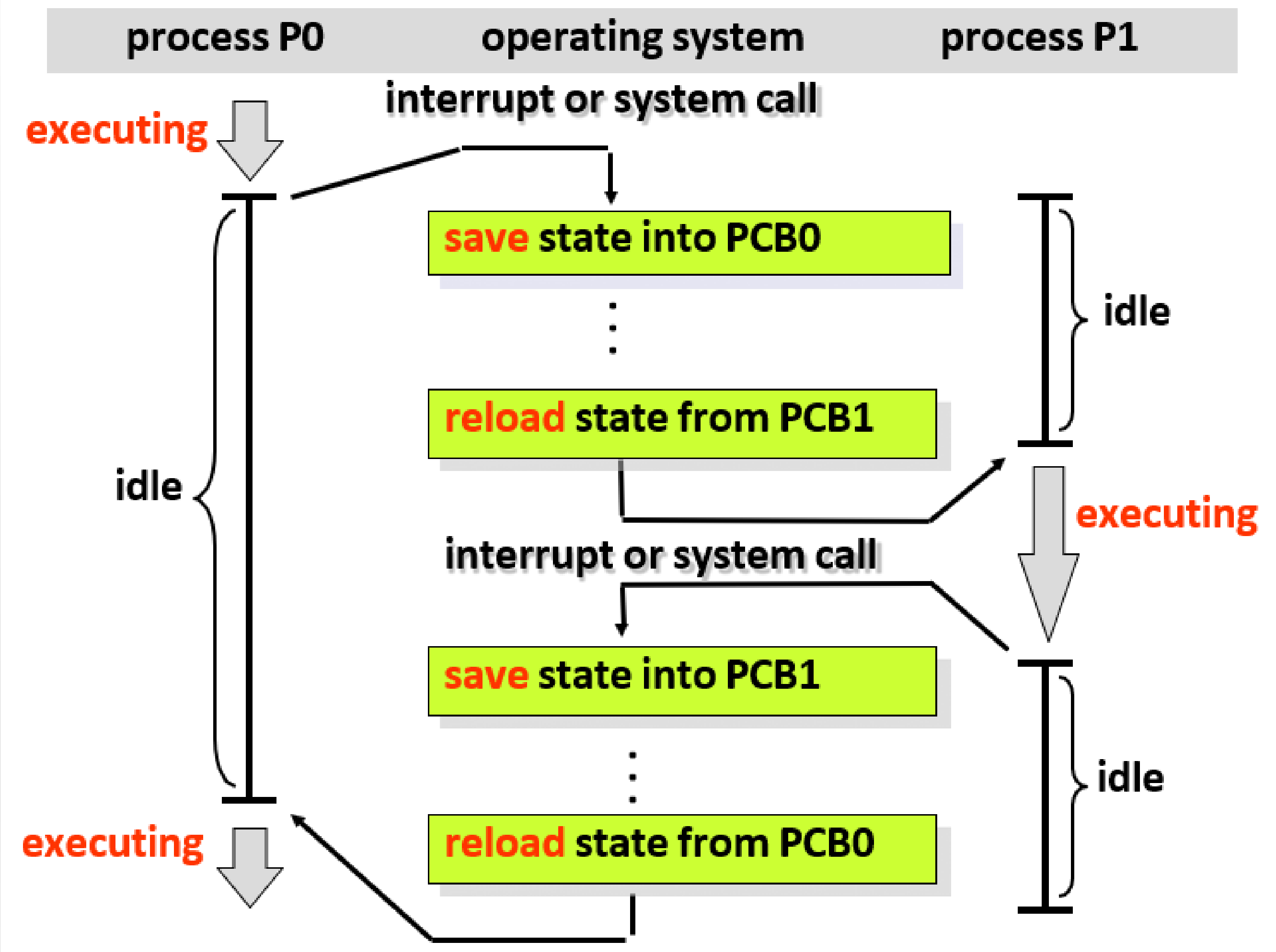
- Only one process is running on any processor at any instant
- However, many processes may be ready or waiting

Process Control Block (PCB)

- Info. associated with each process
- Process state
- Program counter
- CPU registers
- CPU scheduling information (e.g. priority)
- Memory-management information (e.g. base/limit register)
- I/O status information
- Accounting information



Context Switch



Context Switch

- Context Switch: Kernel saves the state of the old process and loads the saved state for the new process
- Context-switch time is purely **overhead**
- Switch time (about 1~1000 ms) depends on
  - memory speed
  - number of registers
  - existence of special instructions
    - a single instruction to save/load all registers
  - hardware support
    - multiple sets of registers (Sun UltraSPARC – a context switch means changing register file pointer)

Review Slides (1)

- What’s the definition of a process?
- What’s the difference between process and thread?
- What’s PCB? its contents?
  - Process state
  - Program counter
  - CPU registers
- The kinds of process state?
  - New, Ready, Running, Waiting, Terminated
- What’s context switch?

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Process Scheduling

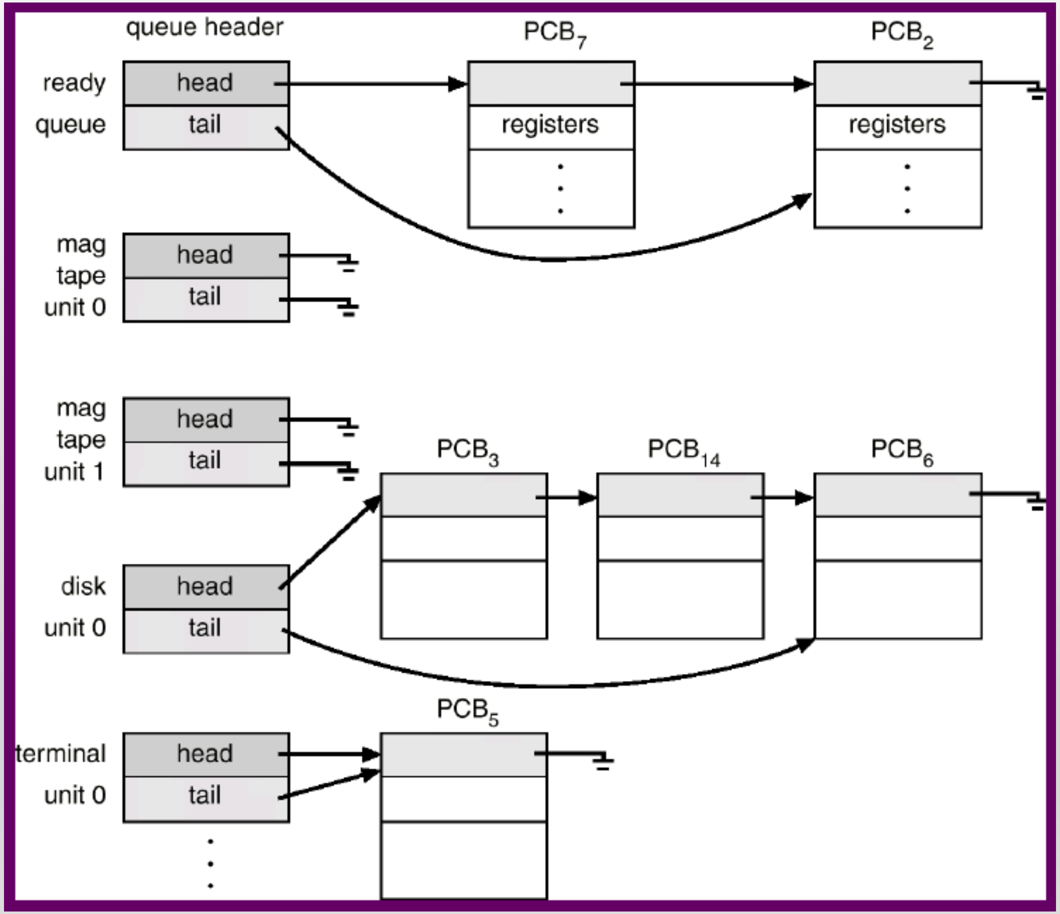
- Multiprogramming: CPU runs process at all times to maximize CPU utilization
- Time sharing: switch CPU frequently such that users can interact with each program while it is running
- Processes will have to wait until the CPU is free and can be re-scheduled

Process Scheduling Queues

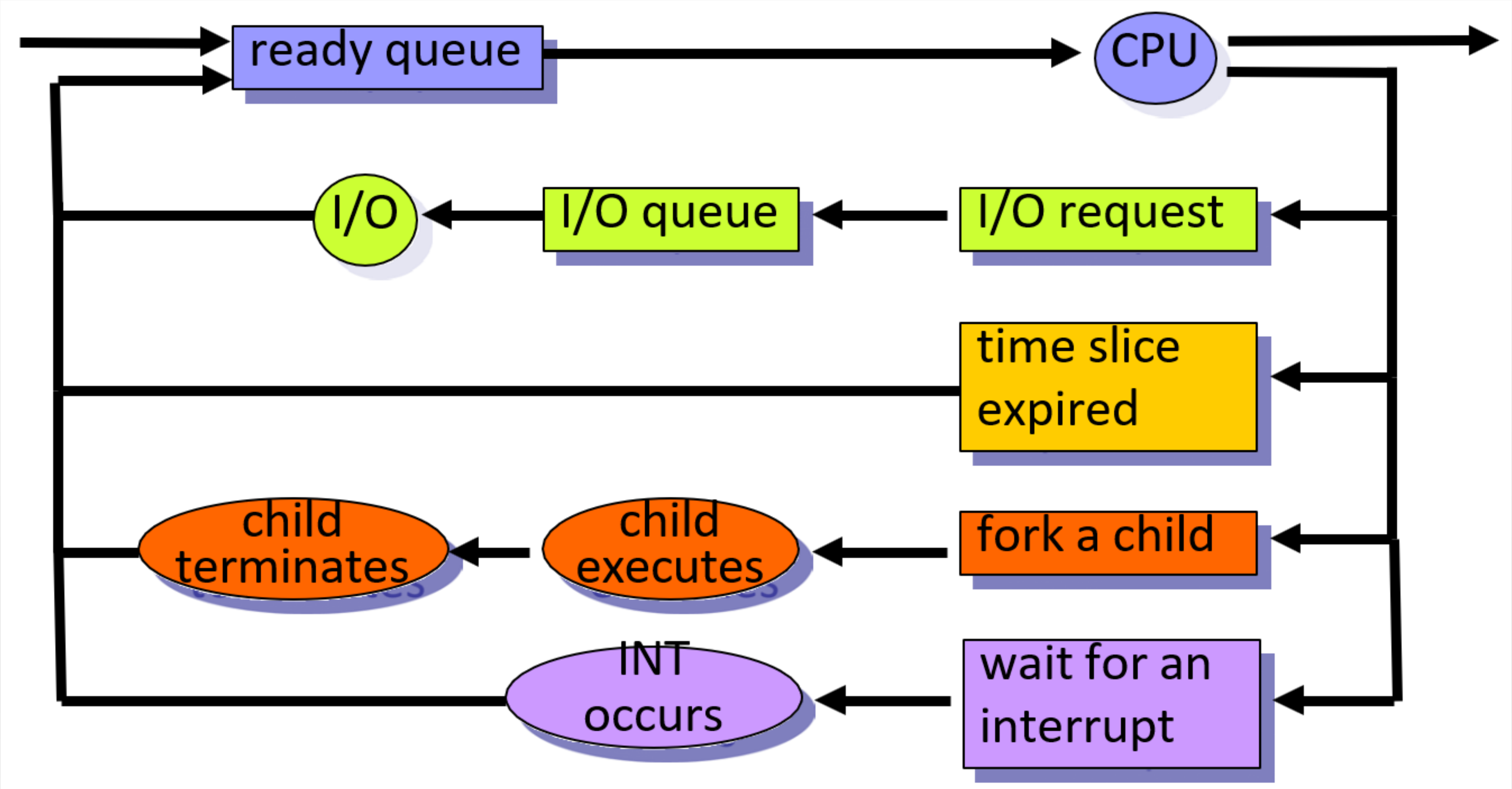
- Processes migrate between the various queues (i.e. switch among states)
- Job queue (New State) - set of all processes in the system
- Ready queue (Ready State) - set of all processes residing in main memory, ready and waiting to execute
- Device queue (Wait State)- set of processes waiting for an I/O device

Process Scheduling Queues

I/O queue

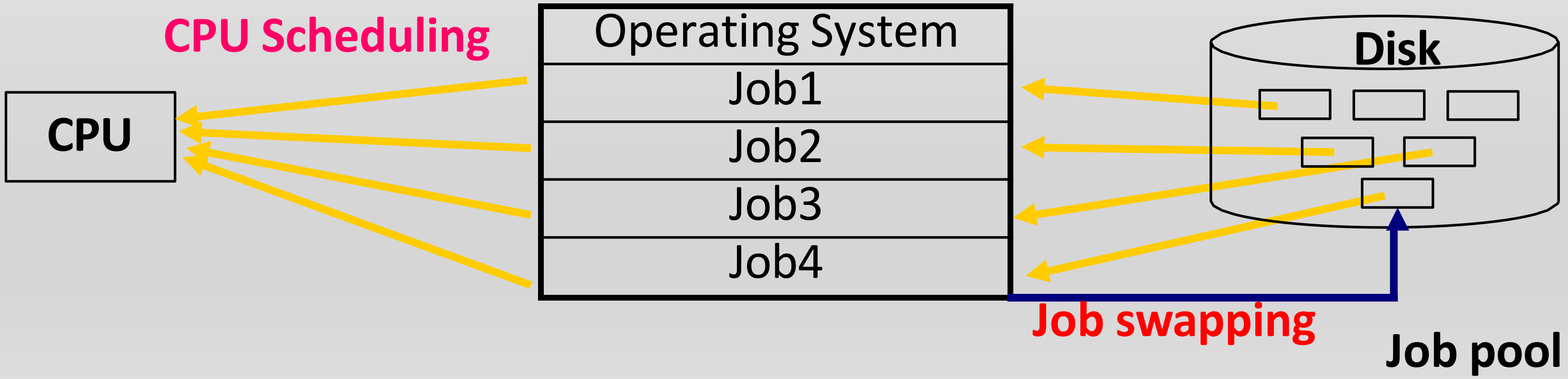


Process Scheduling Diagram



Schedulers

- Short-term scheduler (CPU scheduler)- selects which process should be executed and allocated CPU (Ready state -> Run state)
- Long-term scheduler (job scheduler) - selects which processes should be loaded into memory and brought into the ready queue (New state -> Ready state)
- Medium-term scheduler - selects which processes should be swapped in/out memory (Ready state -> Wait state)

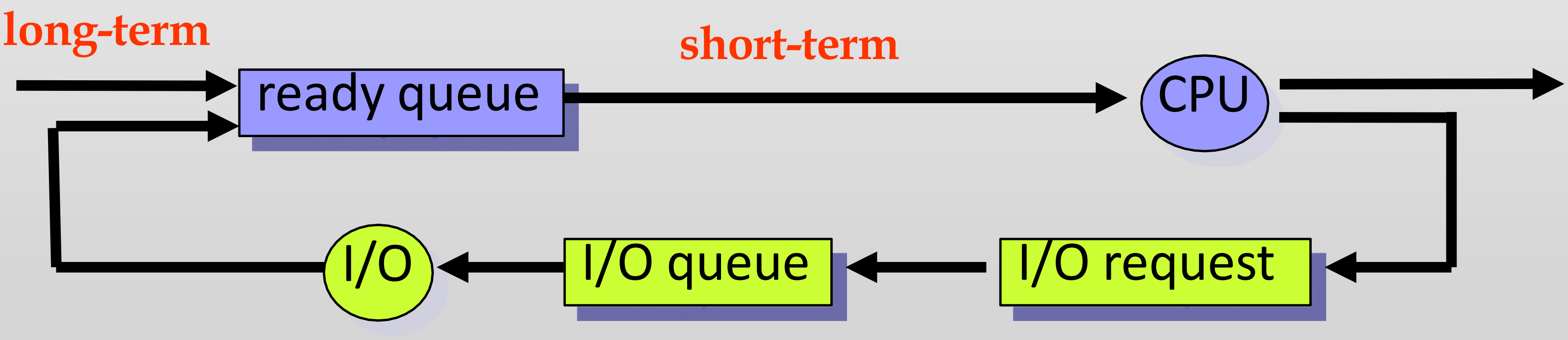


Long-Term Scheduler

- Control **degree of multiprogramming**
- Execute less frequently (e.g. invoked only when a process leaves the system or once several minutes)
- Select a good mix of CPU-bound & I/O-bound processes to increase system overall performance
- UNIX/NT: no long-term scheduler
  - Created process placed in memory for short-term scheduler
  - Multiprogramming degree is bounded by hardware limitation (e.g., # of terminals) or on the self-adjusting nature of users

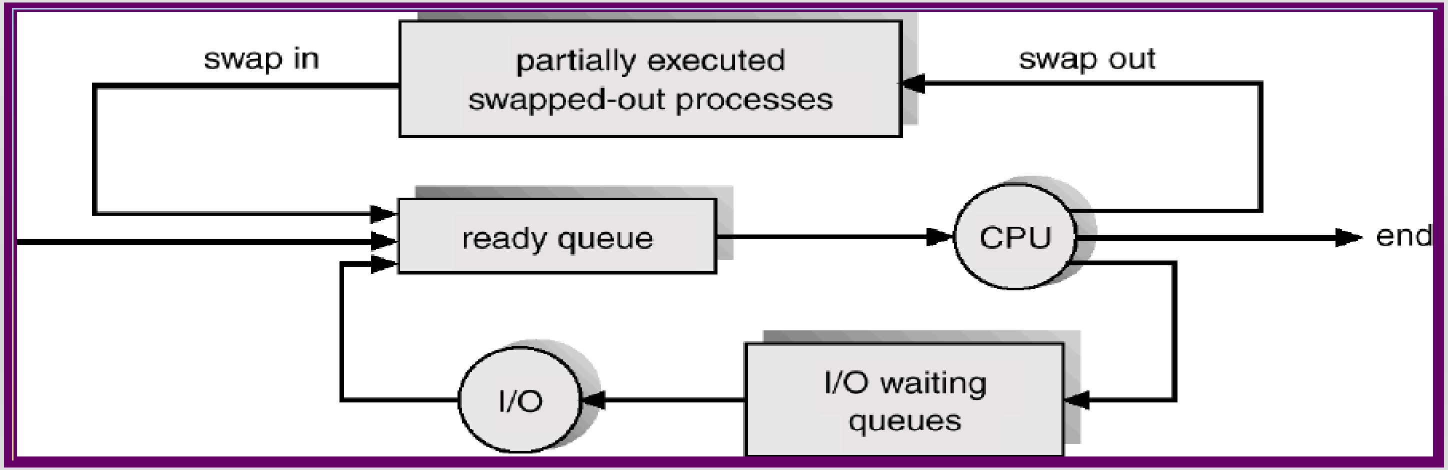
Short-Term Scheduler

- Execute quite frequently (e.g. once per 100ms)
- Must be efficient:
  - if 10 ms for picking a job, 100 ms for such a pick,
  - -> overhead = 10 / 110 = 9%



Medium-Term Scheduler

- swap out: removing processes from memory to reduce the degree of multiprogramming
- swap in: reintroducing swap-out processes into memory
- Purpose: improve process mix ,_free up memory_
- Most modern OS doesn't have medium-term scheduler because having sufficient physical memory or using virtual memory

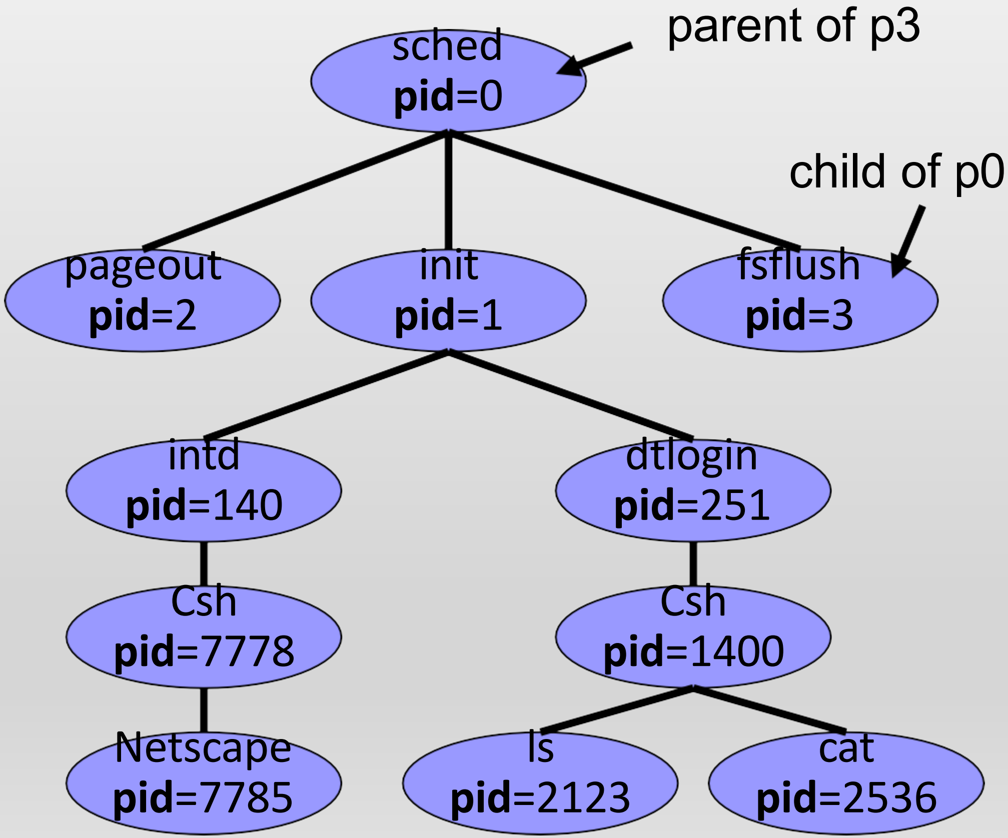


Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Tree of Processes

- Each process is identified by a unique processor identifier (pid)



UNIX: "**ps -ael**" will list complete info of all active processes

Process Creation

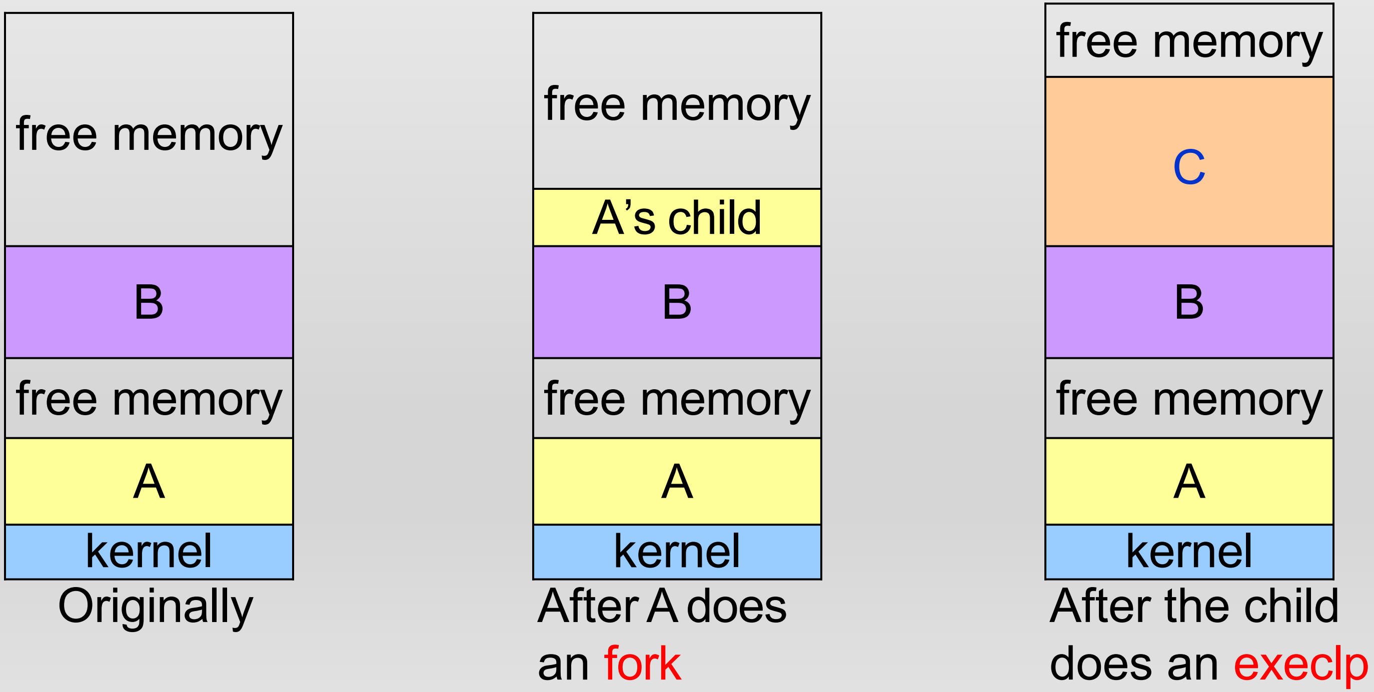
- Resource sharing
  - Parent and child processes share all resources
  - Child process shares subset of parent's resources
  - Parent and child share no resources
- Two possibilities of execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Two possibilities of address space
  - Child duplicate of parent, communication via sharing variables
  - Child has a program loaded into it, communication via message passing

UNIX/Linux Process Creation

- fork system call
  - Create a new (child) process
  - The new process duplicates the address space of its parent
  - Child & Parent execute concurrently after fork
  - Child: return value of fork is 0
  - Parent: return value of fork is PID of the child process
- execlp system call
  - Load a new binary file into memory – destroying the old code
- wait system call
  - The parent waits for **one of its child processes** to complete

UNIX/Linux Process Creation

- Memory space of fork():
  - Old implementation: A’s child is an exact copy of parent
  - Current implementation: use copy-on-write technique to store differences in A’s child address space



UNIX/Linux Example

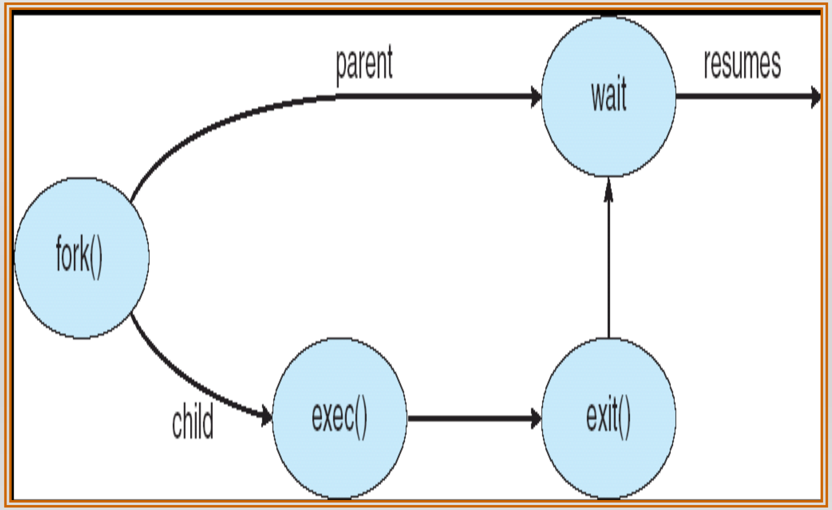
```
#include <stdio.h> void main( )
{
      int A;
      /* fork another process */
      A = fork( );

      if (A == 0) { /* child process */
            printf("this is from child
      process\n");   execlp("/bin/ls", "ls",
      NULL);

      } else { /* parent process */
            printf("this is from parent
            process\n");
            int pid = wait(&status);
            printf("Child %d completes", pid);
      }
```

Output:

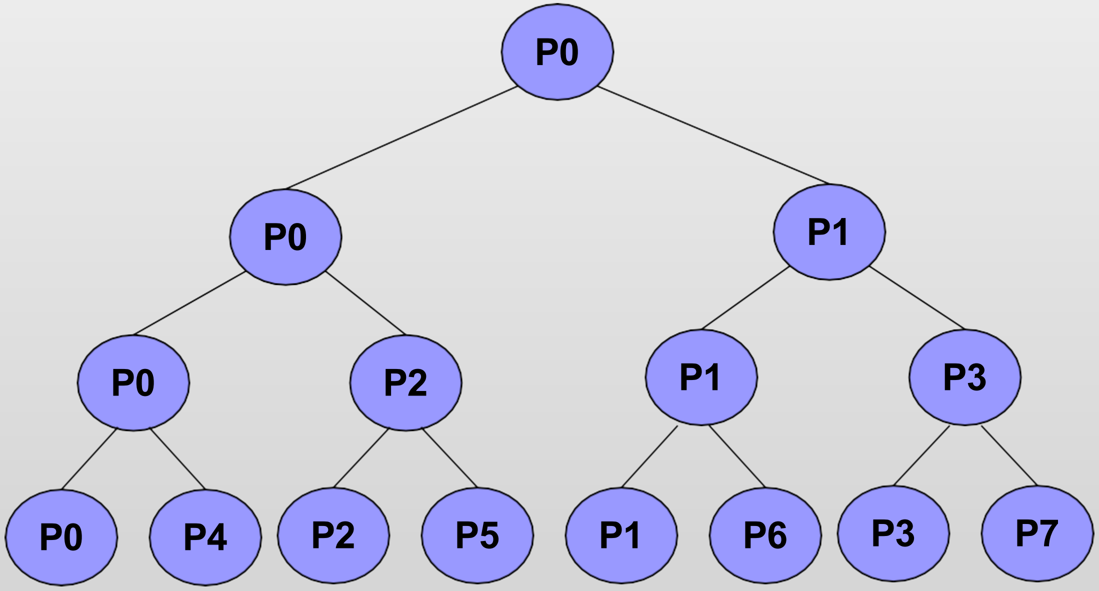
this is from child process
this is from parent process
a.out hello.c readme.txt
Child 32185 completes
process ends 32185



Example Quiz:

- How many processes are created?

```
#include <stdio.h>
#include <unistd.h>
int main() {
	for (int i=0; i<3; i++){
		fork();
	}
	return 0;
}
```



Process Termination

- Terminate when the last statement is executed or exit() is called
  - All resources of the process, including physical & virtual memory, open files, I/O buffers, are deallocated by the OS
- Parent may terminate execution of children processes by specifying its PID (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- Cascading termination:
  - killing (exiting) parent -> killing (exiting) all its children

Review Slides (2)

- What’s long-term scheduler? features?
- What’s short-term scheduler? features?
- What’s medium-term scheduler? features?
- What’s the different between duplicate address space and load program? Their commands?

Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

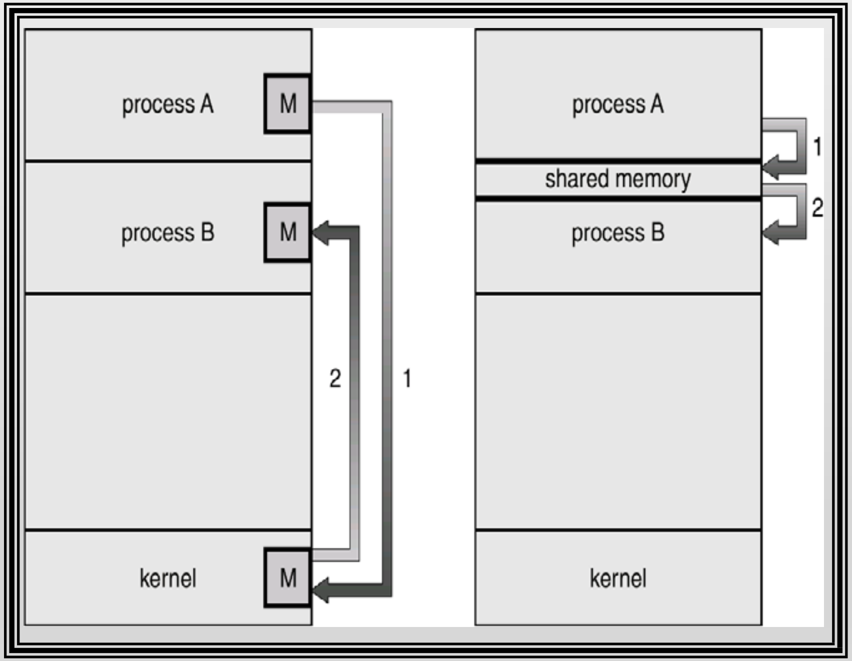
Interprocess Communication

- IPC: a set of methods for the exchange of data among multiple threads in one or more processes
- Independent process: cannot affect or be affected by other processes
- Cooperating process: otherwise
- Purposes
  - information sharing
  - computation speedup (not always true…)
  - convenience (performs several tasks at one time)
  - modularity

Communication Methods

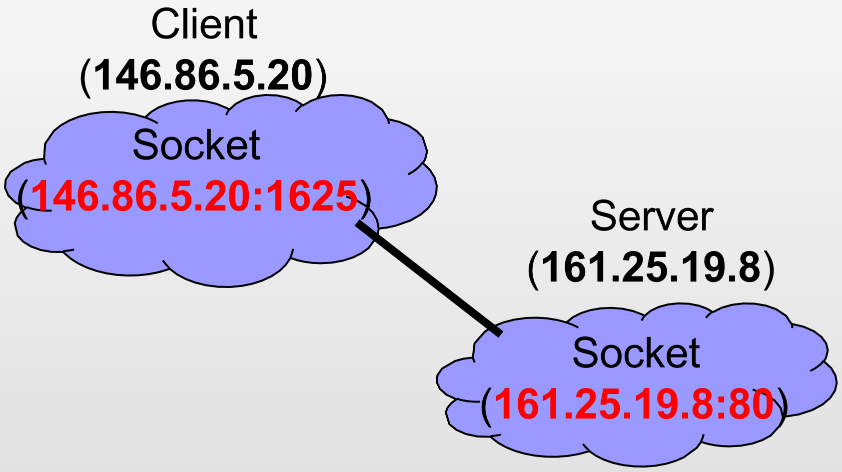
- Shared memory:
  - Require more careful user synchronization
  - Implemented by memory access: faster speed
  - Use memory address to access data
- Message passing:
  - No conflict: more efficient for small data
  - Use send/recv message
  - Implemented by system call: slower speed

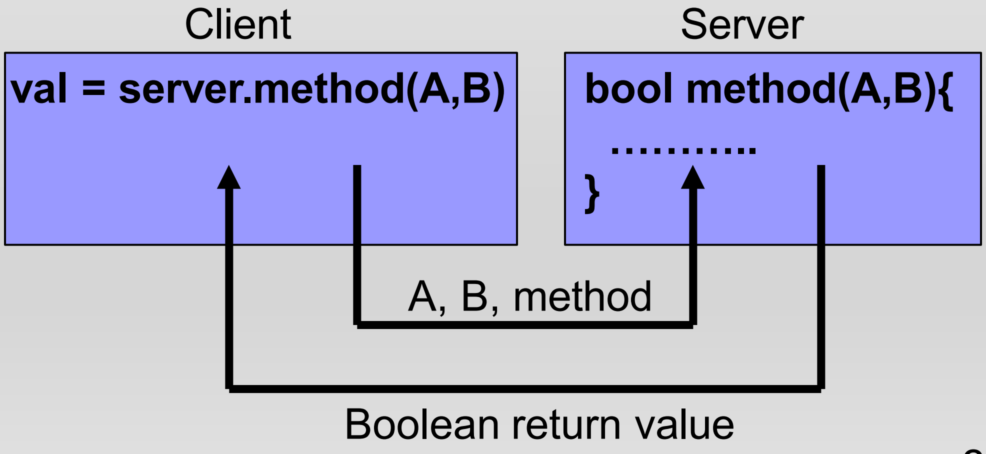
Msg Passing Shared Memory



Communication Methods

- Sockets:
  - A network connection identified by IP & port
  - Exchange unstructured stream of bytes
- Remote Procedure Calls:
  - Cause a procedure to execute in another address space
  - Parameters and return values are passed by message





Interprocess Communication

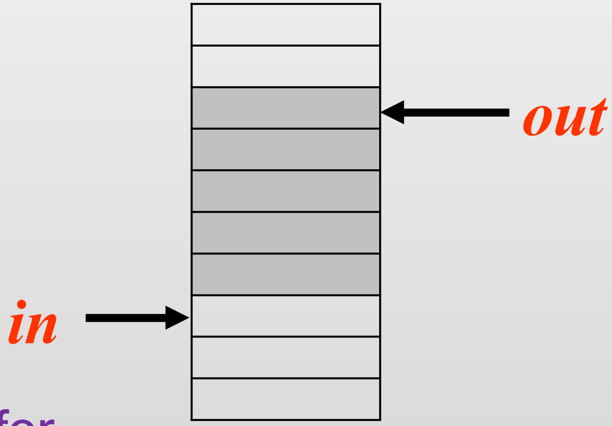
- Shared Memory
- Message Passing
- Socket
- Remote Procedure Calls

Shared Memory

- Processes are responsible for…
  - Establishing a region of shared memory
    - Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment
    - Participating processes must agree to remove memory access constraint from OS
  - Determining the form of the data and the location
  - Ensuring data are not written simultaneously by processes

Consumer & Producer Problem

- Producer process produces information that is consumed by a Consumer process
- Buffer as a circular array with size B
  - next free: in
  - first available: out
  - empty: in = out
  - full: (in+1) % B = out
- The solution allows at most (B-1) item in the buffer
  - Otherwise, cannot tell the buffer is fall or empty



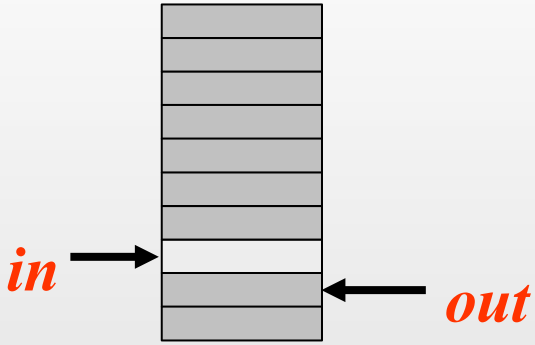
Shared-Memory Solution

```
/*producer*/
while (1) {
   while (((in + 1) % BUFFER_SIZE) == out)
            ; //wait if buffer is full
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

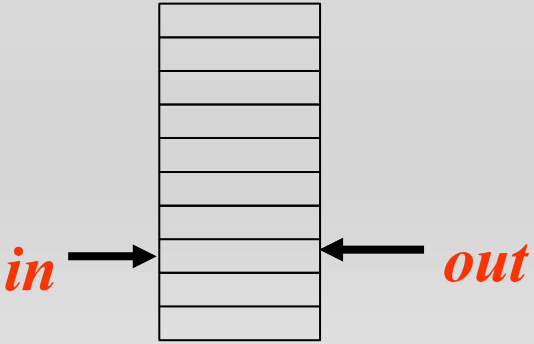
"in" only modified by producer

```
/*consumer*/
while (1) {
   while (in == out); //wait if buffer is empty
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
}
```

"out" only modified by consumer




```
/* global data structure */
#define BUFSIZE  10
item buffer[BUFSIZE];
int  in = out = 0;
```



Interprocess Communication

- Shared Memory
- Message Passing
- Socket
- Remote Procedure Calls

Message-Passing System

- Mechanism for processes to communicate and synchronize their actions
- IPC facility provides two operations:
  - Send(message) - message size fixed or variable
  - Receive(message)
- Message system – processes communicate without resorting to shared variables
- To communicate, processes need to
  - Establish a communication link
  - Exchange a message via send/receive

Message-Passing System

- Implementation of communication link
  - physical (e.g., shared memory, HW bus, or network)
  - logical (e.g., logical properties)
    - Direct or indirect communication
    - Symmetric or asymmetric communication
    - Blocking or non-blocking
    - Automatic or explicit buffering
    - Send by copy or send by reference
    - Fixed-sized or variable-sized messages

Direct communication

- Processes must name each other explicitly:
  - Send (P, message) - send a message to proc P
  - Receive (Q, message) - receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - One-to-One relationship between links and processes
  - The link may be unidirectional, but is usually bi- directional

Direct communication

- Solution for producer-consumer problem:

```
/*producer*/
while (1) {
        send (consumer, nextProduced);
}
/*consumer*/
while (1) {
        receive (producer, nextConsumed);
}
```

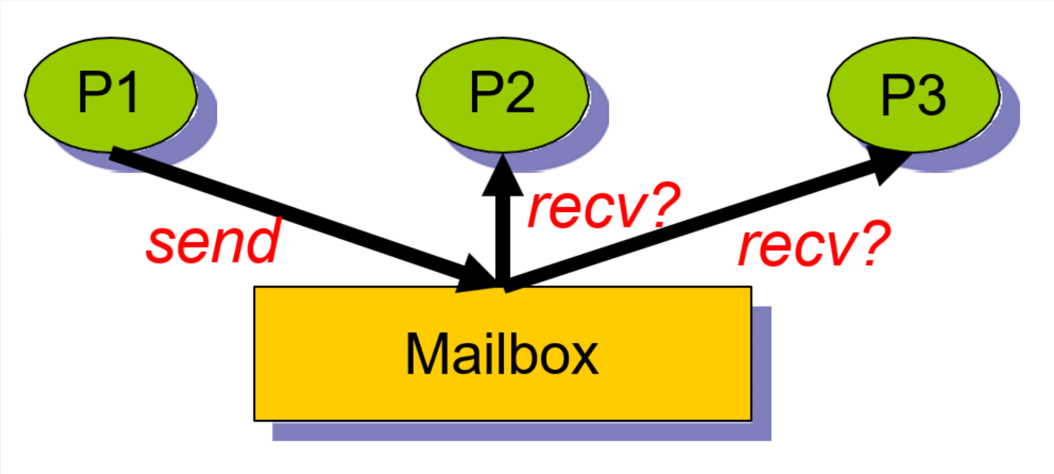
- limited modularity: if the name of a process is changed, all old names should be found

Indirect communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique ID
  - Processes can communicate if they share a mailbox
  - 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
  - Many-to-Many relationship between links and processes
  - Link may be unidirectional or bi-directional
  - Mailbox can be owned either by OS or processes

Indirect Communication

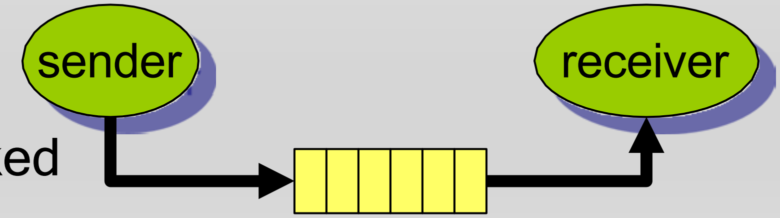
- Mailbox sharing



- 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 a single receiver. Sender is notified who the receiver was

Synchronization

- Message passing may be either blocking (synchronous) or non-blocking (asynchronous)
  - Blocking send: sender is blocked until the message is received by receiver or by the mailbox
  - Nonblocking send: sender sends the message and resumes operation
  - Blocking receive: receiver is blocked until the message is available
  - Nonblocking receive: receiver receives a valid message or a null
- Buffer implementation
  - Zero capacity: blocking send/receive
  - Bounded capacity: if full, sender will be blocked
  - Unbounded capacity: sender never blocks

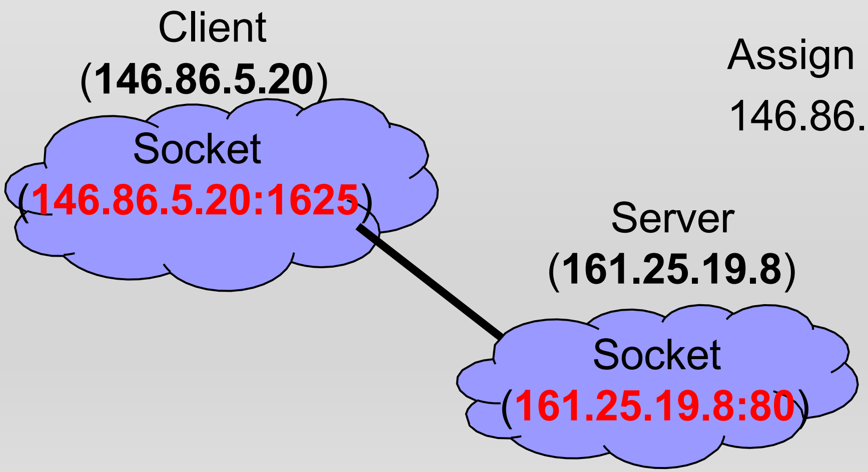


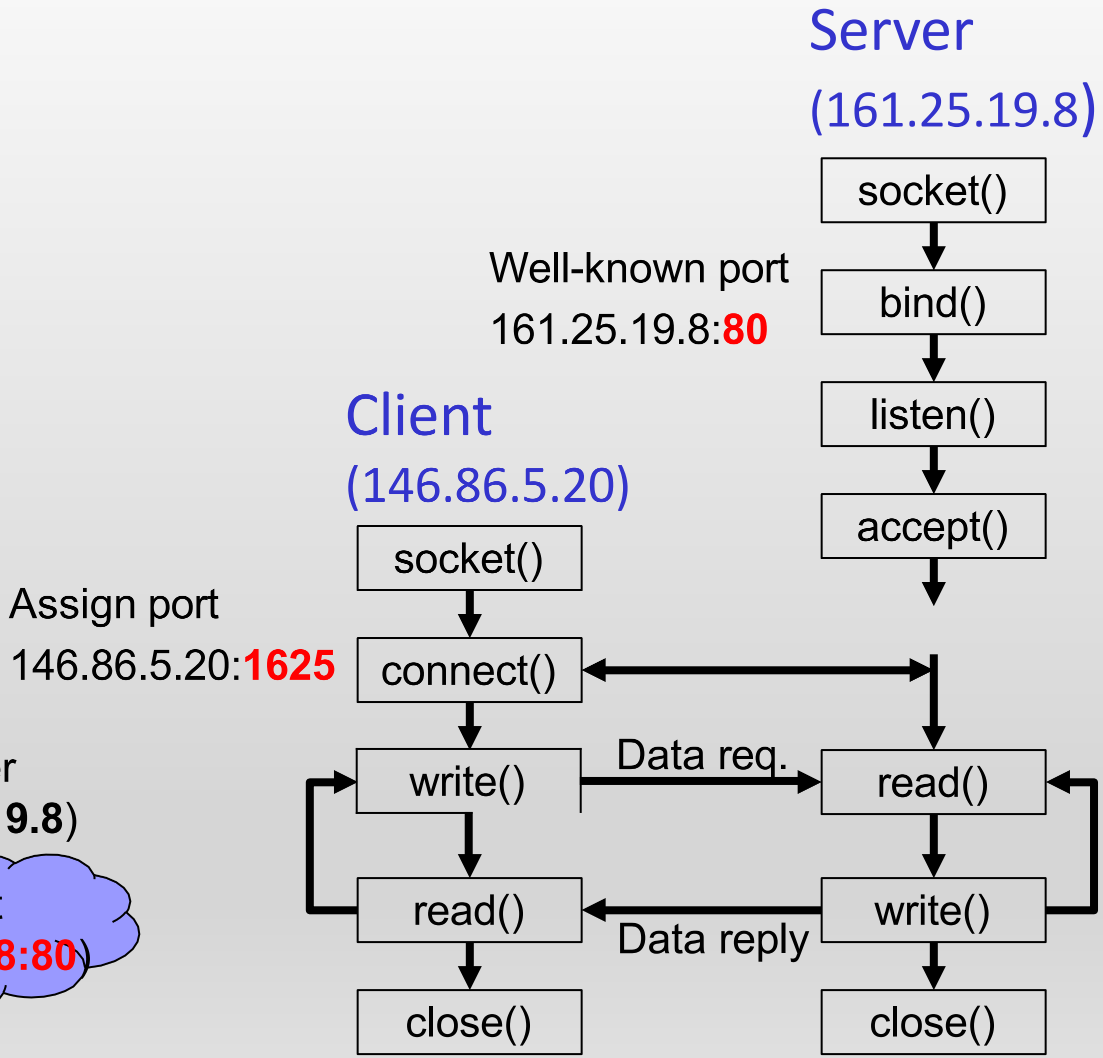
Interprocess Communication

- Shared Memory
- Message Passing
- Socket
- Remote Procedure Calls

Sockets

- A socket is identified by a concatenation of IP address and port number
- Communication consists between a pair of sockets
- Use 127.0.0.1 to refer itself

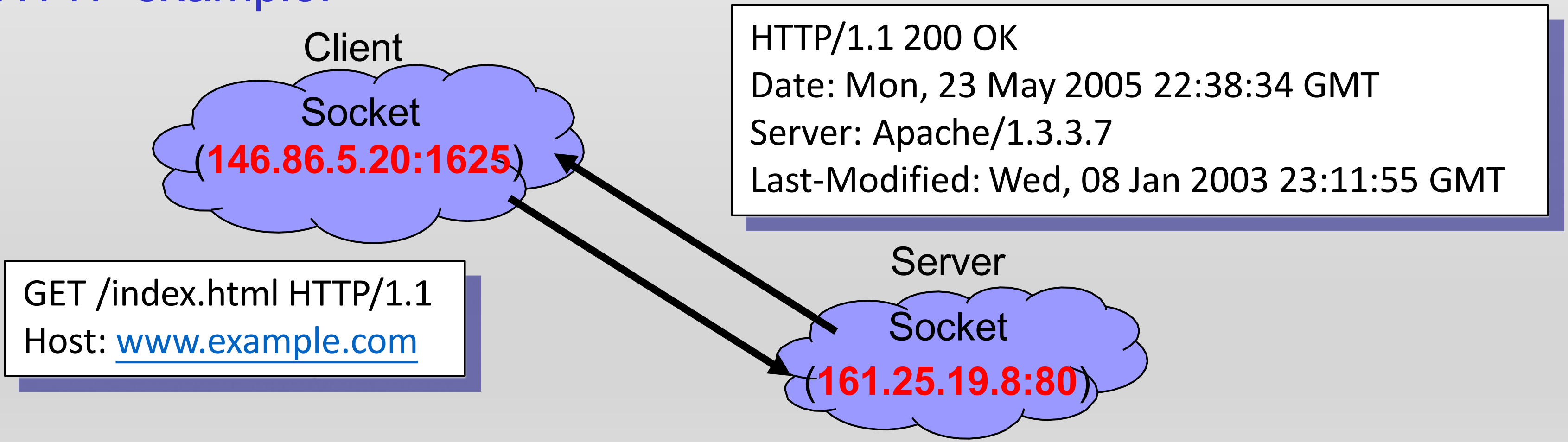




Sockets

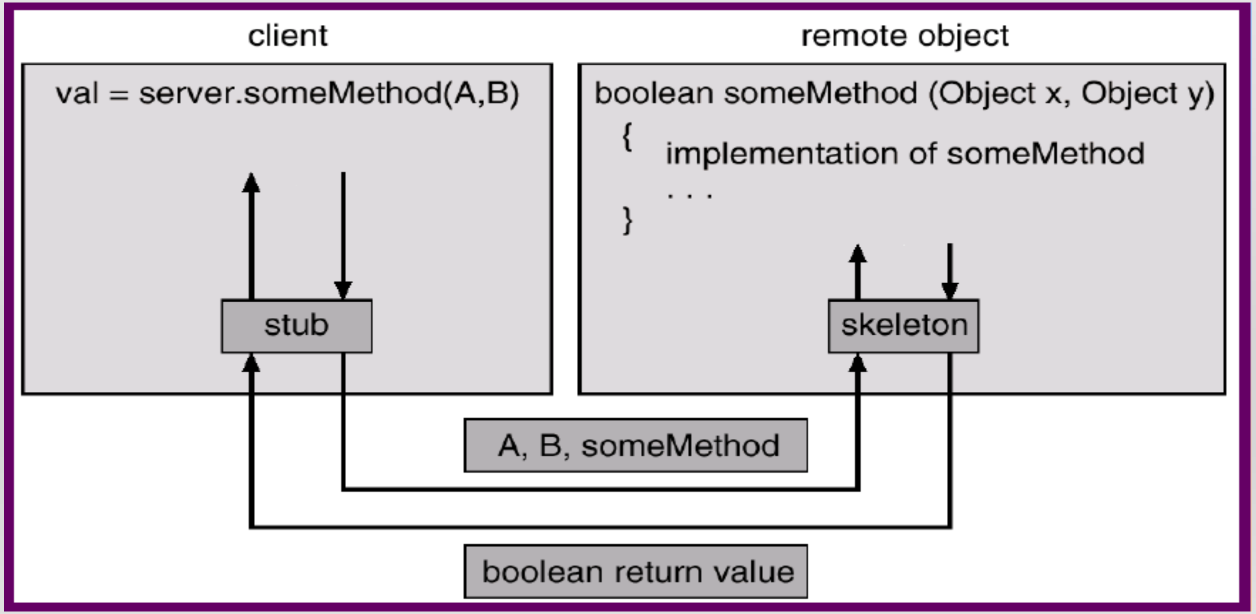
- Considered as a low-level form of communication unstructured stream of bytes to be exchanged
- Data parsing responsibility falls upon the server and the client applications

HTTP example:



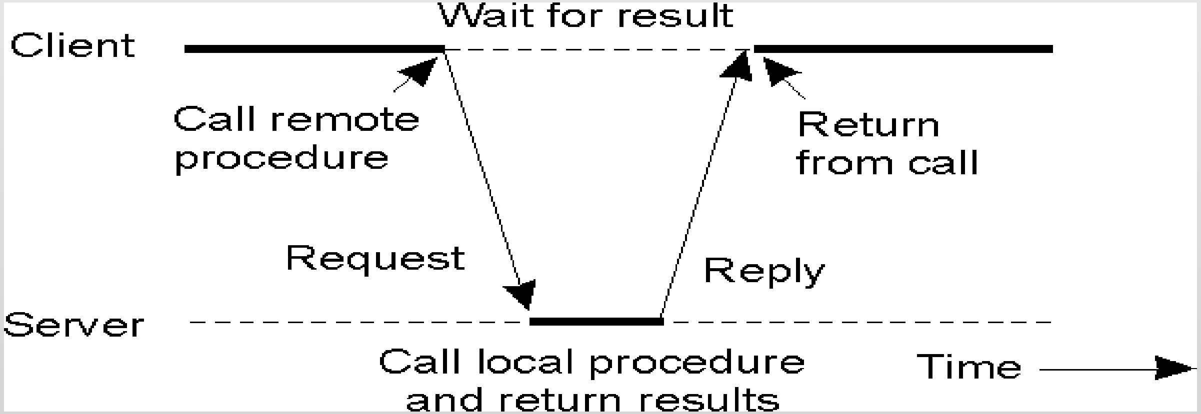
Remote Procedure Calls: RPC

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - allows programs to call procedures located on other machines (and other processes)
- Stubs - client-side proxy for the actual procedure on the server



Client and Server Stubs

- Client stub:
  - Packs parameters into a message (i.e. parameter marshaling)
  - Calls OS to send directly to the server
  - Waits for result-return from the server



Server stub:

- Receives a call from a client
- Unpacks the parameters
- Calls the corresponding procedure
- Returns results to the caller

Review Slides (3)

- Shared memory vs. Message-passing system?
- Direct vs. Indirect message-passing system?
- Blocking vs. Non-Blocking?
- Socket vs. RPC?

Reading Material & HW

- Chap 3
- HW (Problem set)
    - 3.1
    - 3.2
    - 3.5
    - 3.7
    - 3.10

Example: POSIX Shared Memory

size R/W mode

```
{
    /* allocate a R/W shared memory segment */
    char* segment_id = shmget(IPC_PRIVATE, 4096, S_IRUSR | S_IWUSR);
```

Mem.location R/W mode

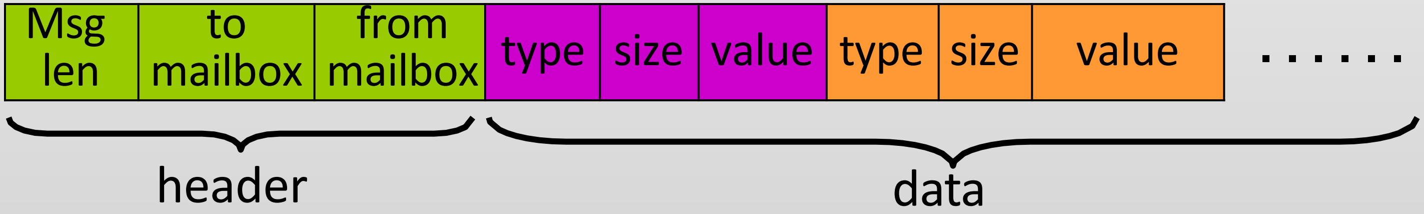
```
    /* attach the shared memory segment */
    char* shared_memory = (char*) shmat(segment_id, NULL, 0);
    /* write a message to the shared memory segment */
    sprintf(shared_memory, “Write to shared memory”);
    /* print out the string from the shared memory segment */
    printf(“%s\n”, shared_memory);
    /* detach the shared memory segment */
    shmdt(shared_memory);
    /* remove the shared memory segment */
    shmctl(shared_memory, IPC_RMID, NULL);
}
```

Example: Mach Message Passing

- Mach operating system
  - developed at CMU
  - microkernel design
  - most communications are carried out by messages and mailboxes (aka ports)
  - Problem: performance (data coping)
- When each task (process) is created
  - kernel & notify mailboxes also created
  - kernel mailbox: channel between OS & task
  - notify mailbox: OS sends event notification to

Mach Mailbox

- port-allocate: system call to create a mailbox
  - default buffer size: 8 messages
  - FIFO queueing
  - a message: one fixed-size header + variable-length data portion
  - implementing both blocking- & non-blocking send/receive



RPC Problems

- Data representations -> integer, floating?
- Different address spaces -> pointer?
- Communication error -> duplicate or missing calls

RPC Problems: Data Representation Issue

- Problem
  - IBM mainframes use EBCDIC char code and IBM PC uses ASCII code
  - Integer: one’s complement and 2’s complement
  - Floating-point numbers
  - Little endian and big endian
- Solution
  - External data representation (XDR)

RPC Problems: Address Space Issue

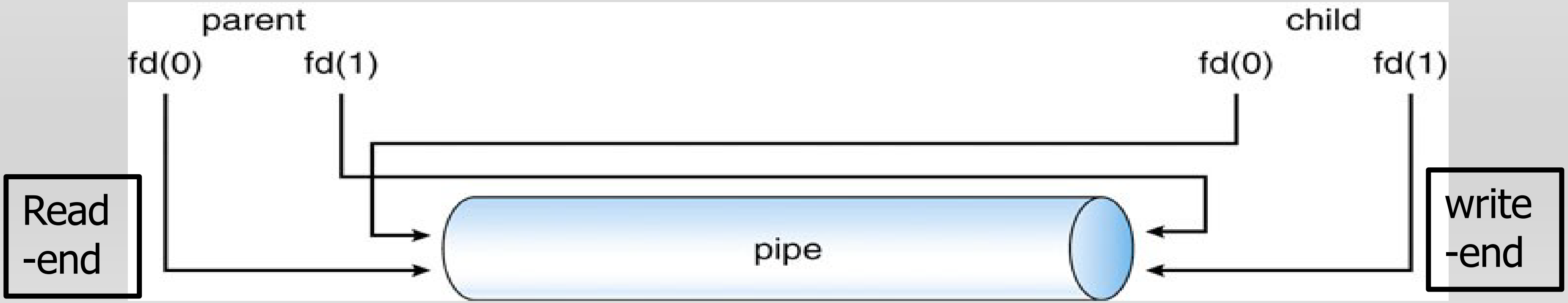
- A pointer is only meaningful in its address space
- Solutions
  - No pointer usage in RPC calls
  - Copy the entire pointed area (such as arrays or strings)
    - Only suitable for bounded and known areas

RPC Problems: Communication Issue

- RPCs may fail, or be duplicated and execute more than once, as a result of common network errors
- at most once: prevent duplicate calls
  - Implemented by attaching a timestamp to each message
  - The server must keep a history large enough to ensure that repeated messages are detected
- exact once: prevent missing calls
  - The server must acknowledge to the client that the RPC call was received and executed
  - The client must resend each RPC call periodically until the server receives the ACK

Pipes

- One of the 1st IPC mechanism in early UNIX systems
- Pipe is a special type of file
- Issues in implementing
  - uni- or bi-directional?
  - half or full duplex? (travel in both directions simultaneously)
  - Must a relationship (parent - child) exist?
  - Over a network, or reside on the same machine?



Ordinary Pipes

- Also called anonymous pipes in Windows
- Requires a parent-child relationship between the communicating processes
  - Implemented as a special file on Unix (via fork(), a child process inherits open files from its parent)
  - Can only be used between processes on the same machine
- Unidirectional: two pipes must be used for two-way communication

UNIX:

```
int fd[2];
pipe(fd);
```

Windows:

```
CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)
```

Named Pipes

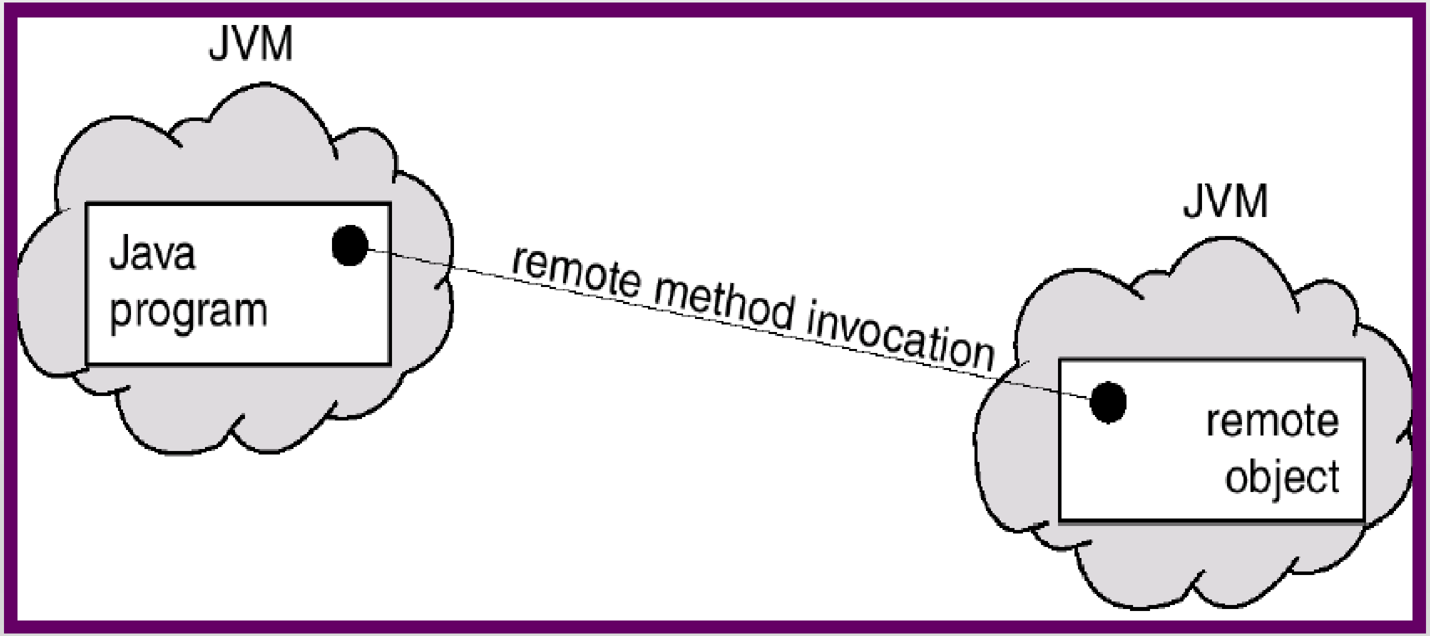
- No parent-child relationship is required
- Several processes can use it for communications
  - It may have several writers
- Continue to exist after communicating processes exit
- In Unix:
  - Also called FIFO
  - Communicating processes have to be on the same machine
- In Windows:
  - bi-directional
  - Communicating processes can be on different machine

UNIX/Linux: Fork

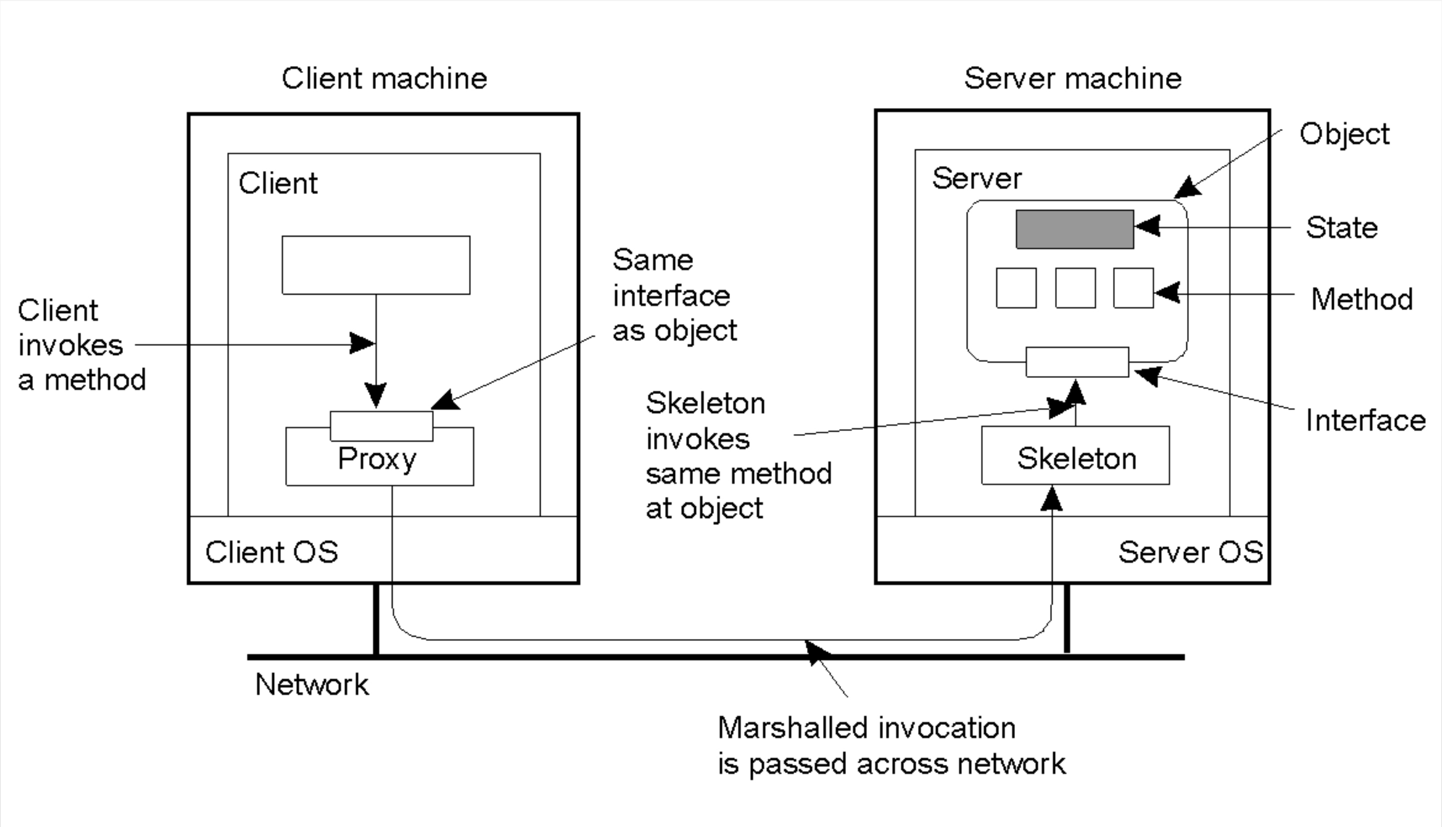
- Inherited from the parent:
  - process credentials
  - environment
  - stack
  - memory
  - open file descriptors
  - signal handling settings
  - scheduler class
  - process group ID
  - session ID
  - current working directory
  - root directory
  - file mode creation mask (umask)
  - resource limits
  - controlling terminal
- Unique to the child:
  - process ID
  - different parent process ID
  - Own copy of file descriptors and directory streams.
  - process, text, data and other memory locks are NOT inherited.
  - process times, in the tms struct
  - resource utilizations are set to 0
  - pending signals initialized to the empty set
  - timers created by timer_create not inherited
  - asynchronous input or output operations not inherited

Remote Method Invocation

- RMI is a Java mechanism similar to RPC
- RMI allows a Java program on one machine to invoke a method on a remote object instead of a function



Distributed Objects



A remote object with client-side proxy

Static & Dynamic RMI

- RMI = Remote Method Invocation
    - Invoke an object's method through proxy
- Static invocation
    - objectname.methodname(para)
    - If interfaces change, apps must be recompiled
- Dynamic invocation
    - invoke(object, method, inpars, outpars)

Q & A

Thank you for your attention