



Kampus
Merdeka
INDONESIA JAYA

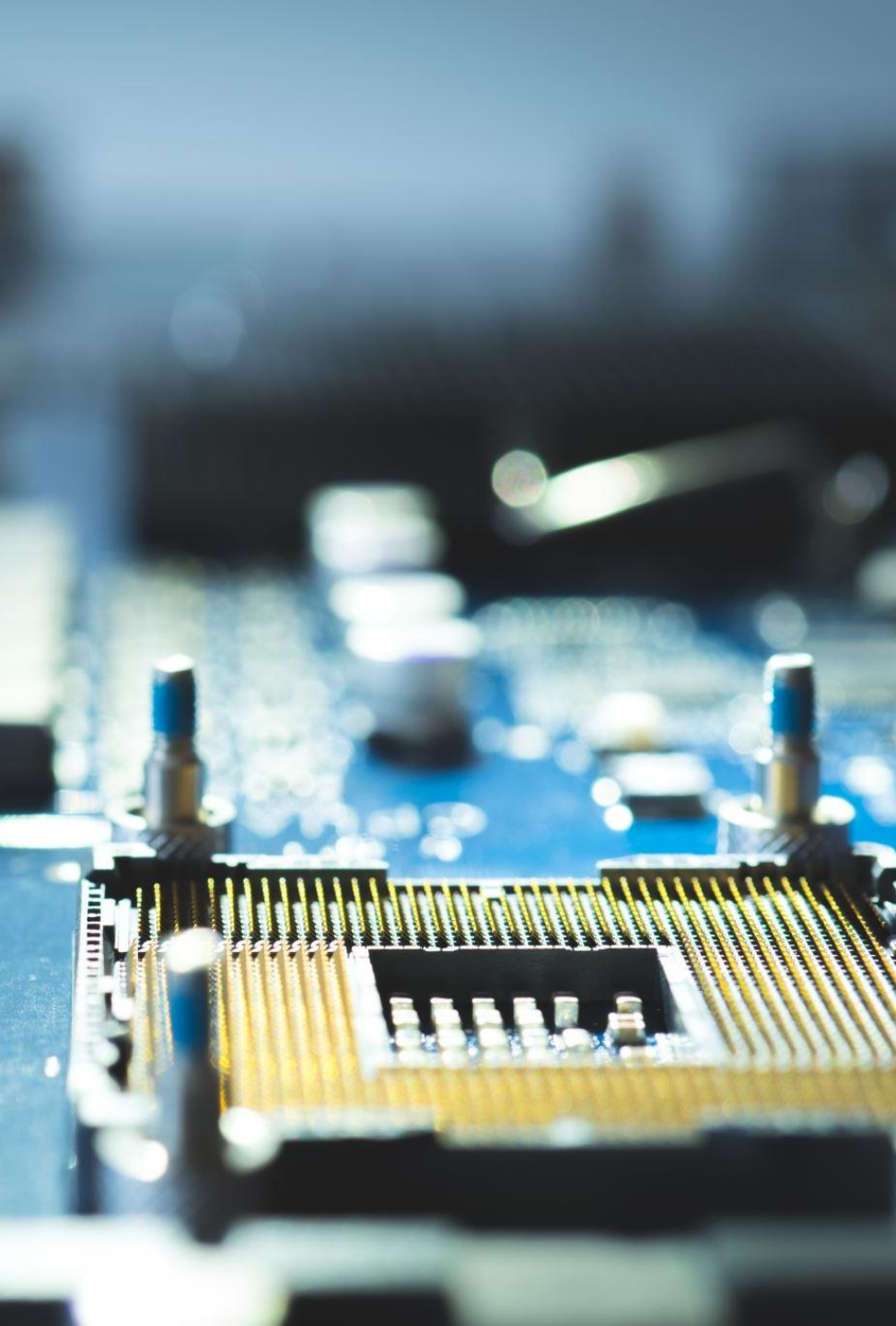


TOP #369
QS
WORLD
UNIVERSITY
RANKINGS



Process - Management Process

BADRUS ZAMAN



Agenda

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Communication in Client-Server Systems

Process Concept

An operating system executes a variety of programs:

- Batch system – **jobs**
- Time-shared systems – **user programs** or **tasks**

Textbook uses the terms job and process almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

Multiple parts

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
- **Function parameters**, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

Process Concept (contd.)

Program is passive entity stored on disk (**executable file**), process is active

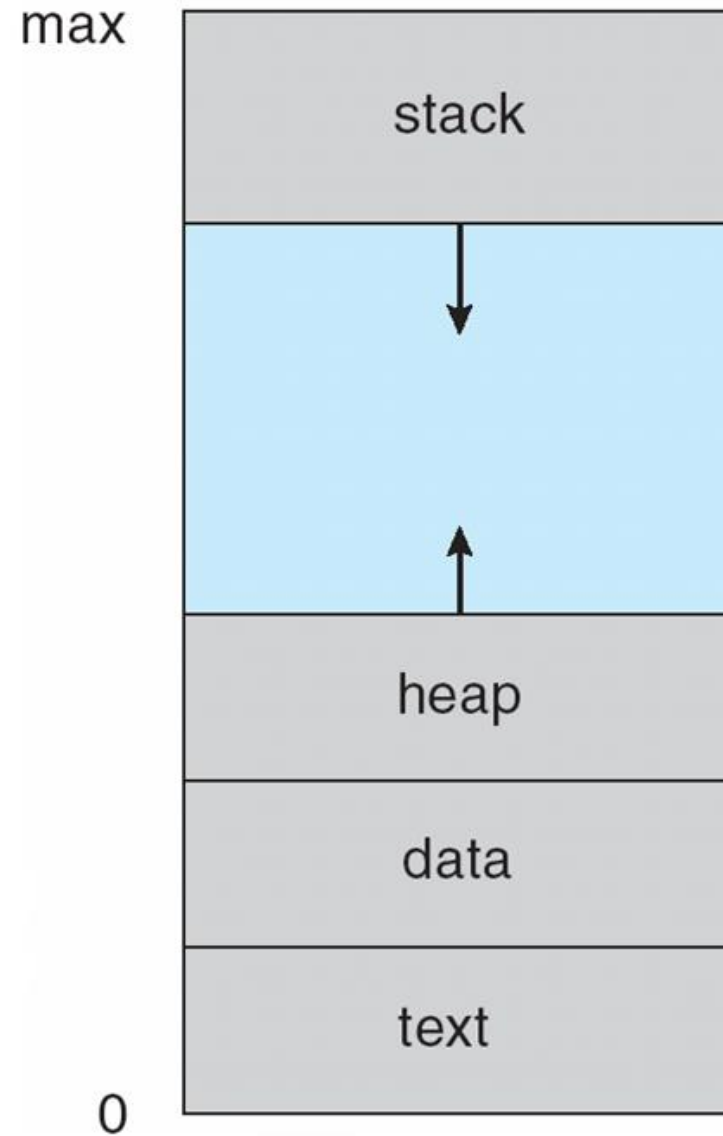
- Program becomes process when executable file loaded into memory

Execution of program started via GUI mouse clicks, command line entry of its name, etc.

One program can be several processes

- Consider multiple users executing the same program

Process in Memory



5 Process State

new: The process is being created

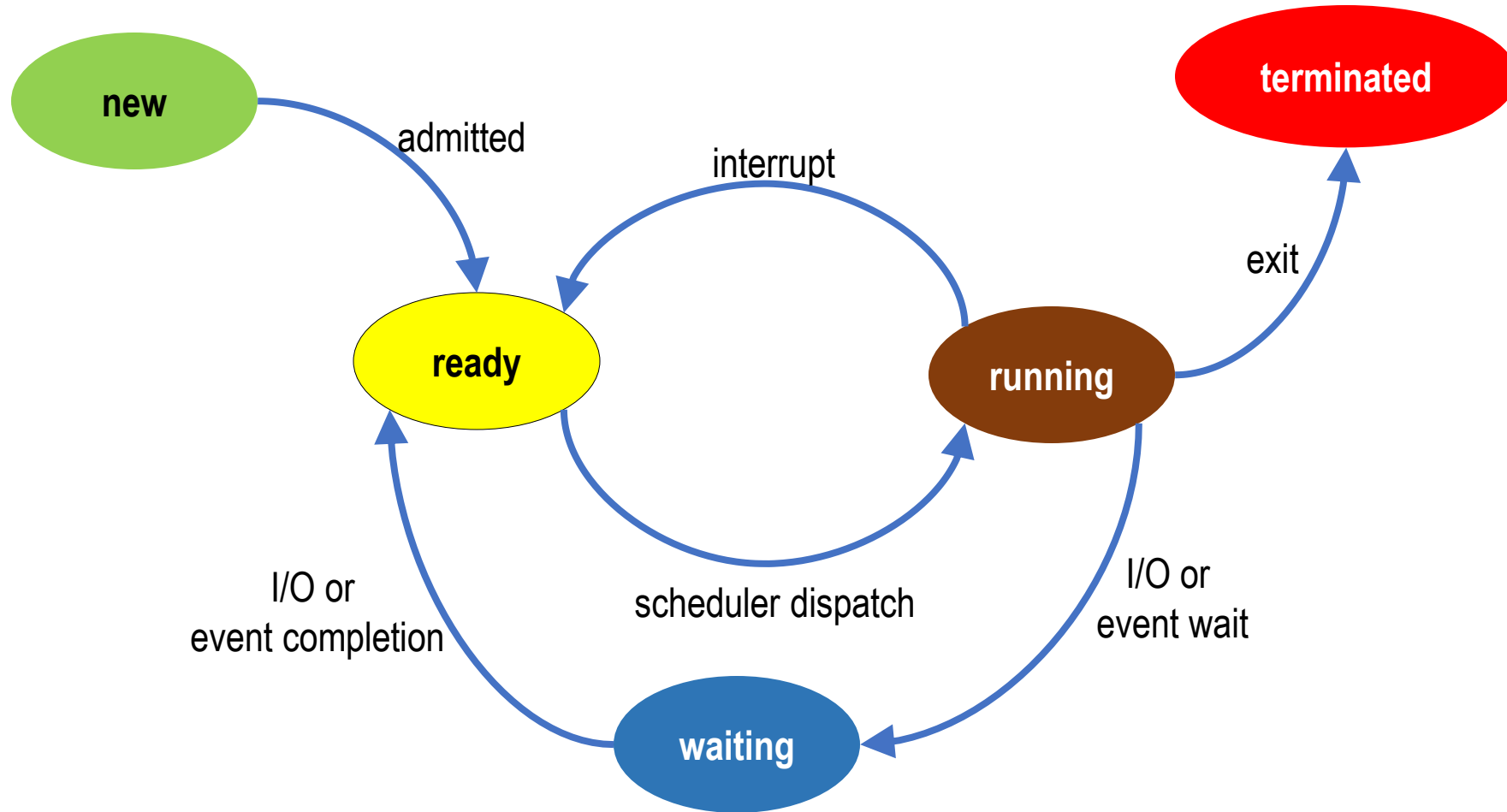
running: Instructions are being executed

waiting: The process is waiting for some event to occur

ready: The process is waiting to be assigned to a processor

terminated: The process has finished execution

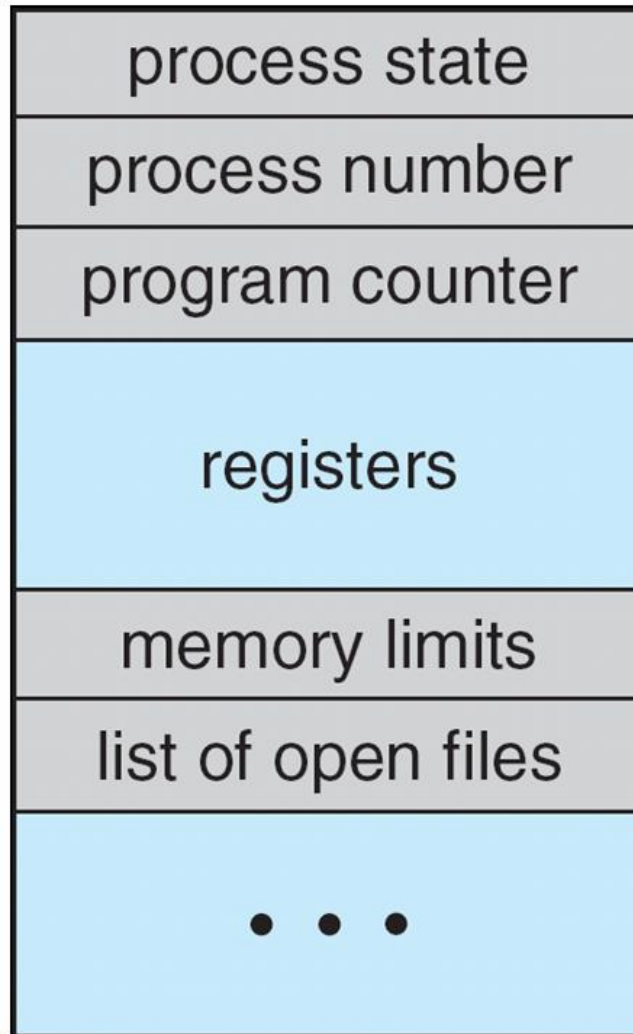
Process State Diagram





Process Information

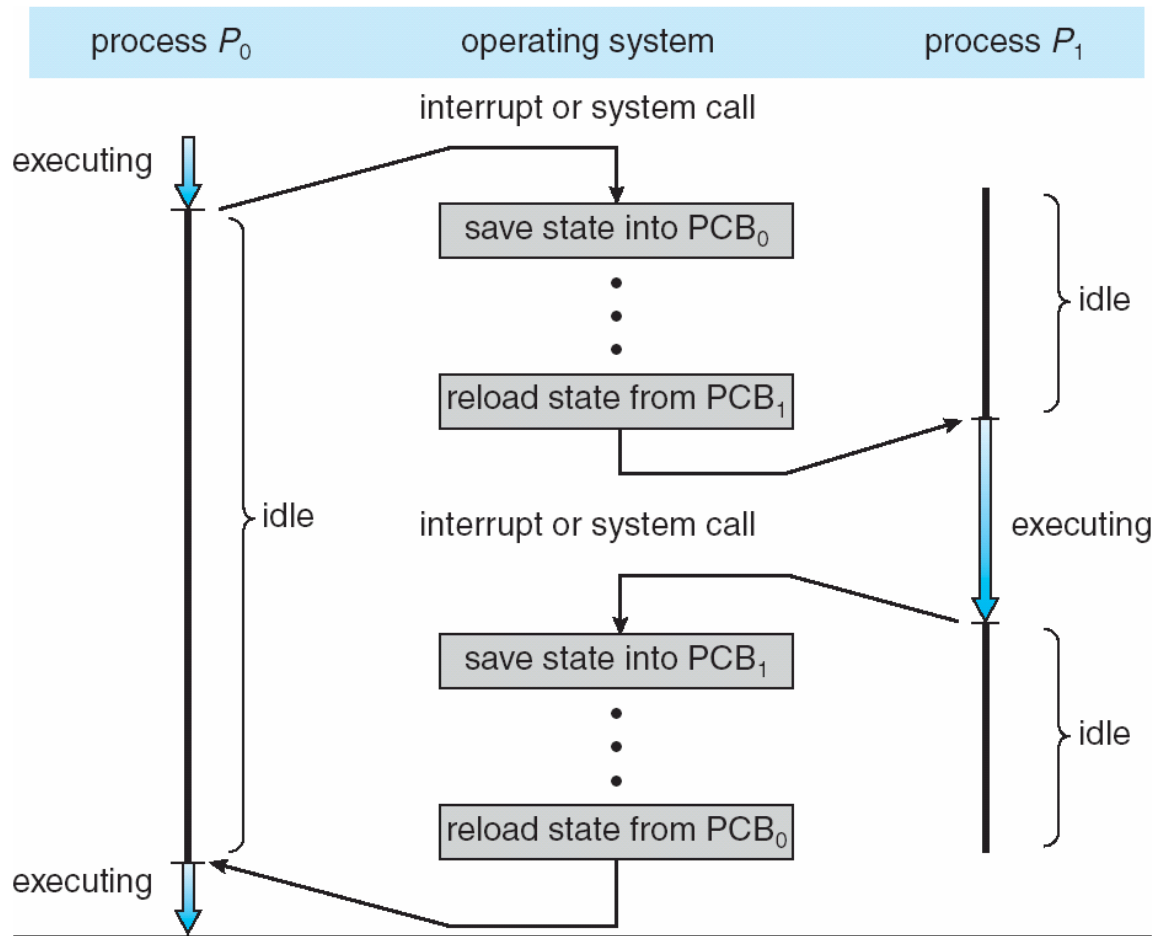
- Dimanakah informasi proses disimpan?
 - Data struktur dari OS dalam bentuk table: Satu entry table/linked list => struktur data untuk menampung informasi satu proses (array of structure).
 - PCB



Process Control Block (PCB) or 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 process, list of open files

CPU Switch From Process to Process



Threads

So far, process has a single thread of execution

Consider having multiple program counters per process

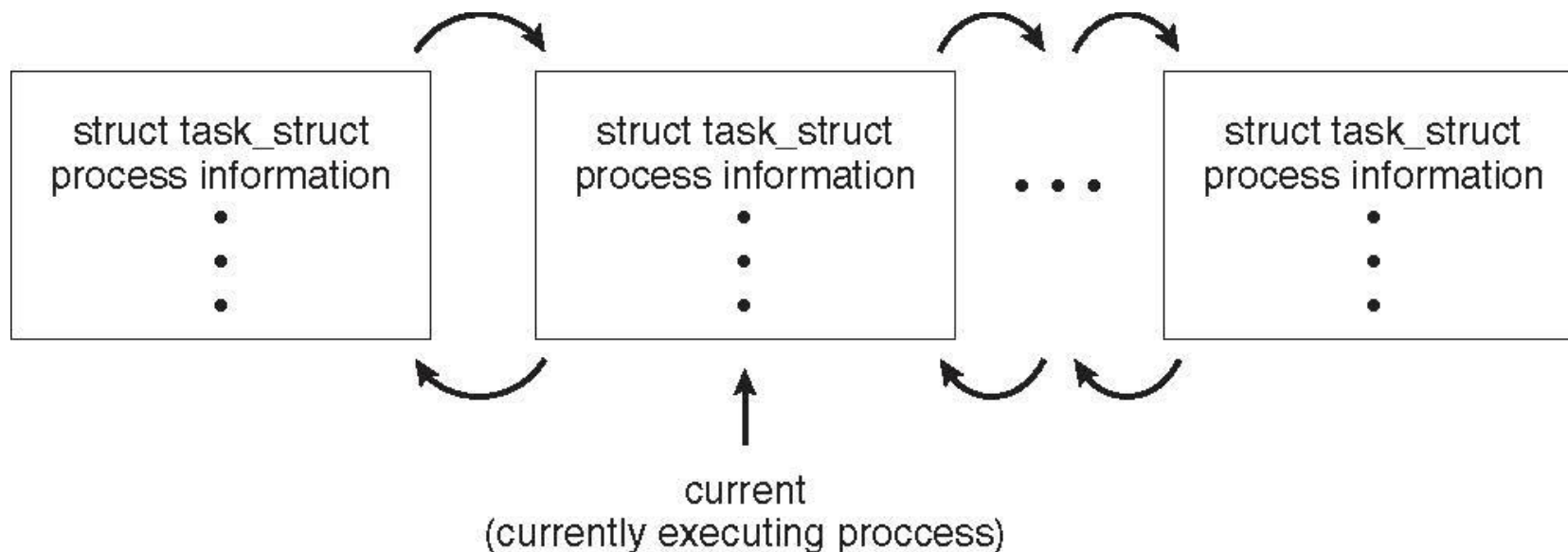
- Multiple locations can execute at once
 - Multiple threads of control -> **threads**

Must then have storage for thread details, multiple program counters in PCB

Process Representation in Linux

Represented by the C structure `task_struct`

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



Process Scheduling

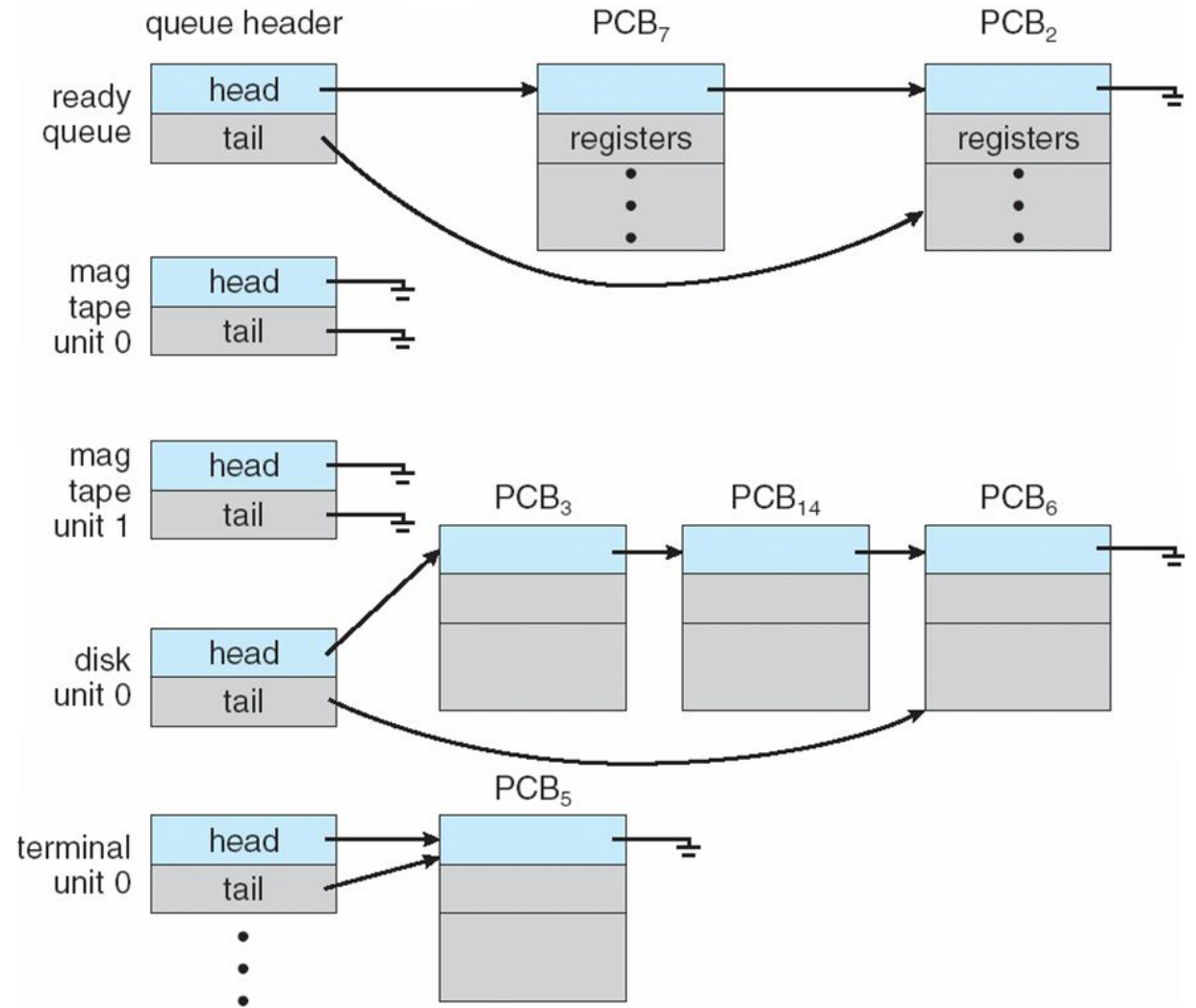
Maximize CPU use, quickly switch processes onto CPU for time sharing

Process scheduler selects among available processes for next execution on CPU

Maintains **scheduling queues** of processes

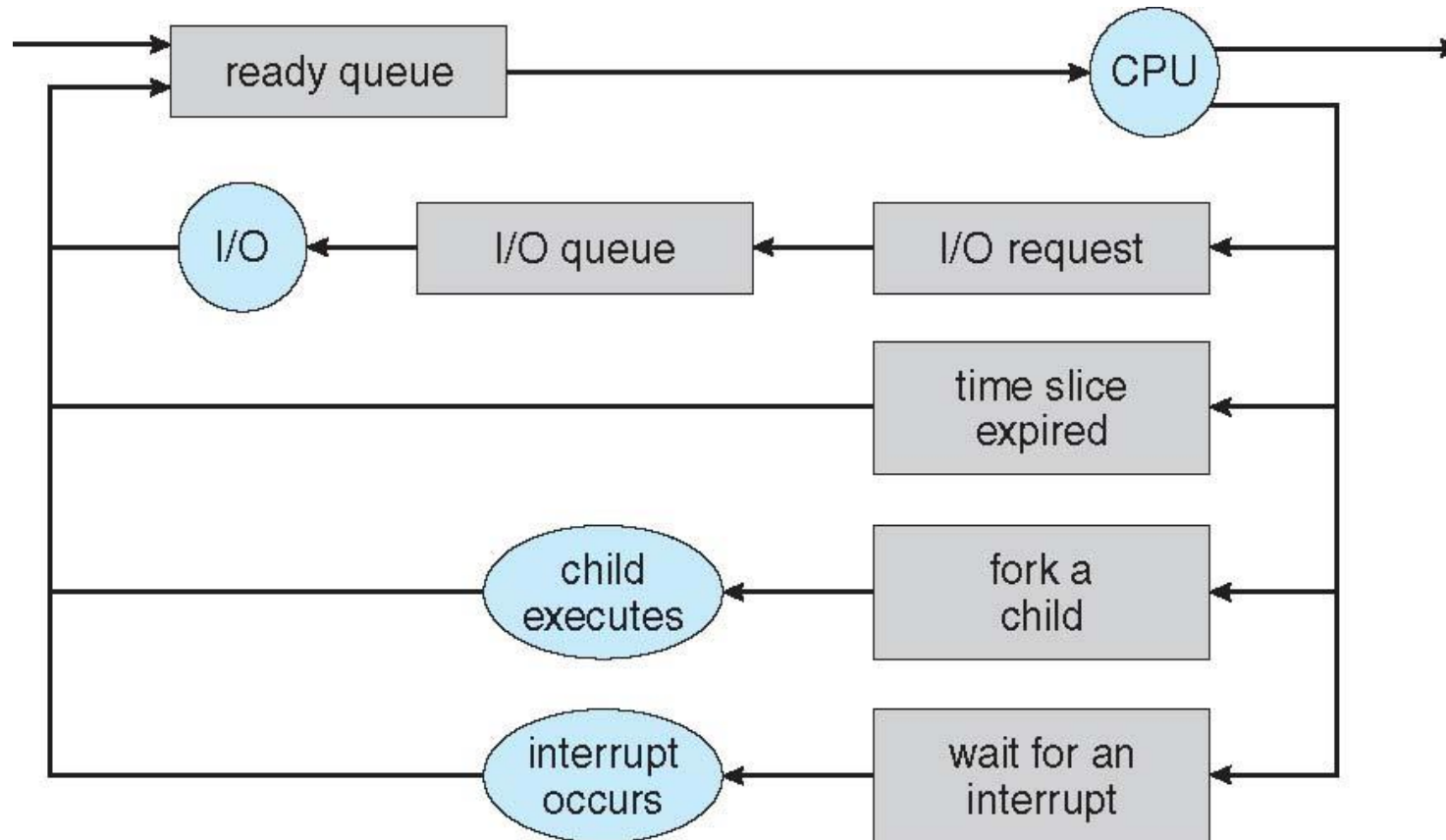
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



Scheduler

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

- Sometimes the only scheduler in a system
- Short-term scheduler is invoked frequently (milliseconds) \Rightarrow (must be fast)

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes) \Rightarrow (may be slow)
- The long-term scheduler controls the **degree of multiprogramming**

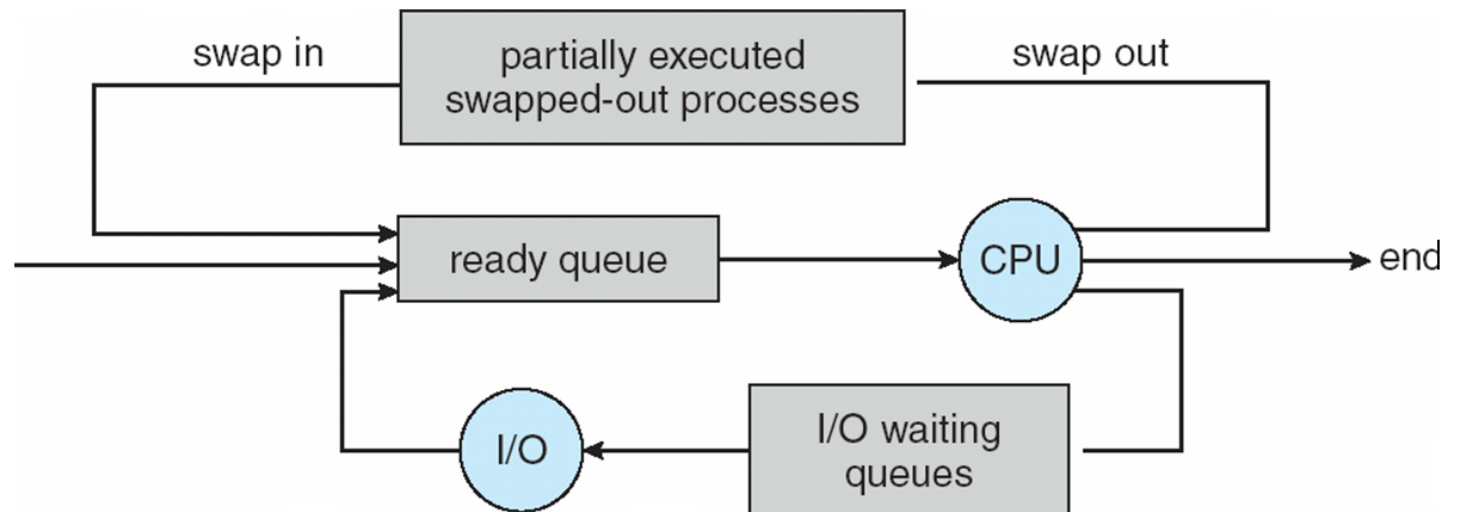
Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Long-term scheduler strives for good *process mix*

Addition of Medium- Term Scheduling

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



Multitasking in Mobile Systems

Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended

Due to screen real estate, user interface limits iOS provides for a

- Single **foreground** process- controlled via user interface
- Multiple **background** processes– in memory, running, but not on the display, and with limits
- Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Android runs foreground and background, with fewer limits

- Background process uses a **service** to perform tasks
- Service can keep running even if background process is suspended
- Service has no user interface, small memory use

Context Switch

When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**

Context of a process represented in the PCB

Context-switch time is overhead; the system does no useful work while switching

- The more complex the OS and the PCB → the longer the context switch

Time dependent on hardware support

- Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Process Creation

Parent process create **children** processes, which, in turn create other processes, forming a **tree** of processes

Generally, process identified and managed via a **process identifier (pid)**

Resource sharing options

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

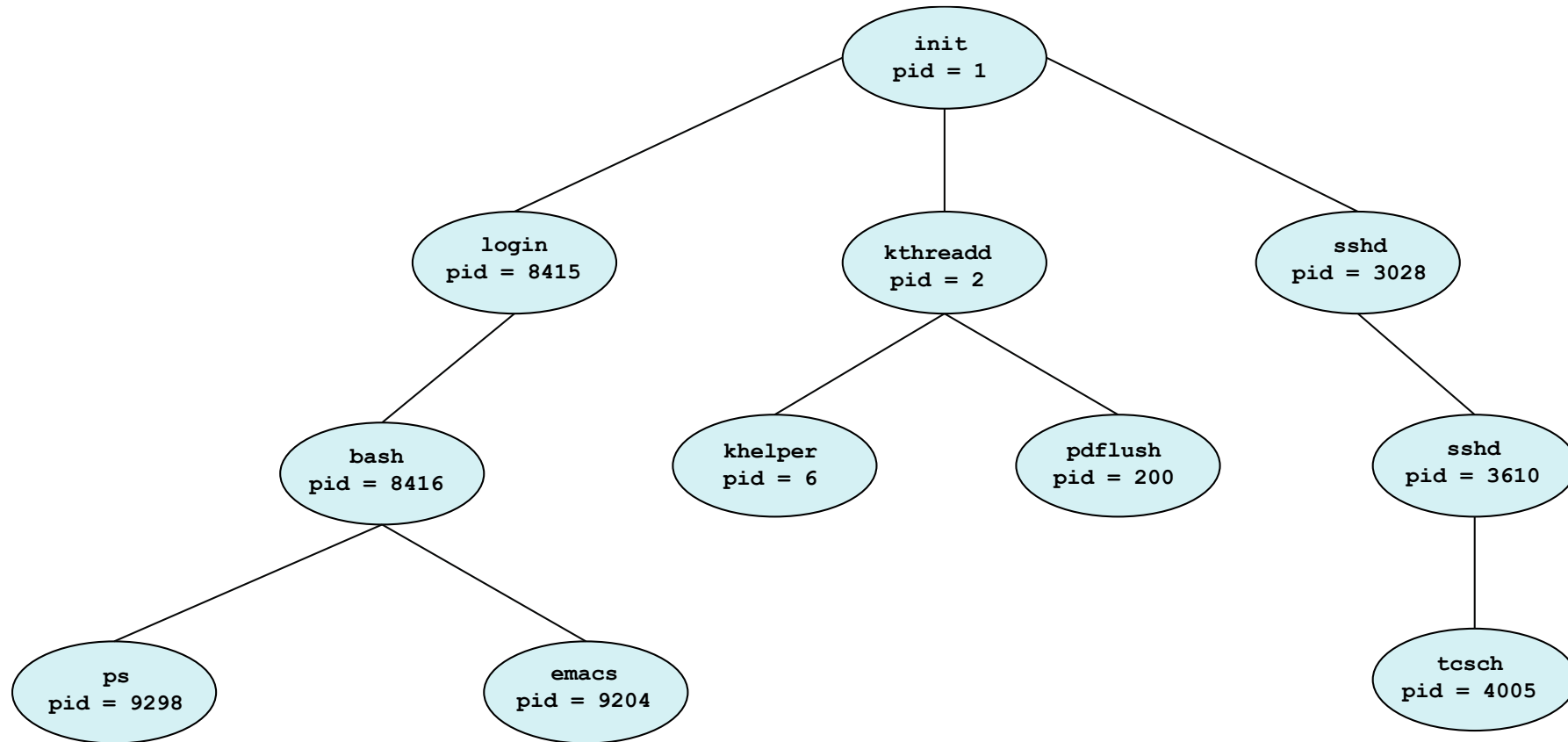
Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

Address space

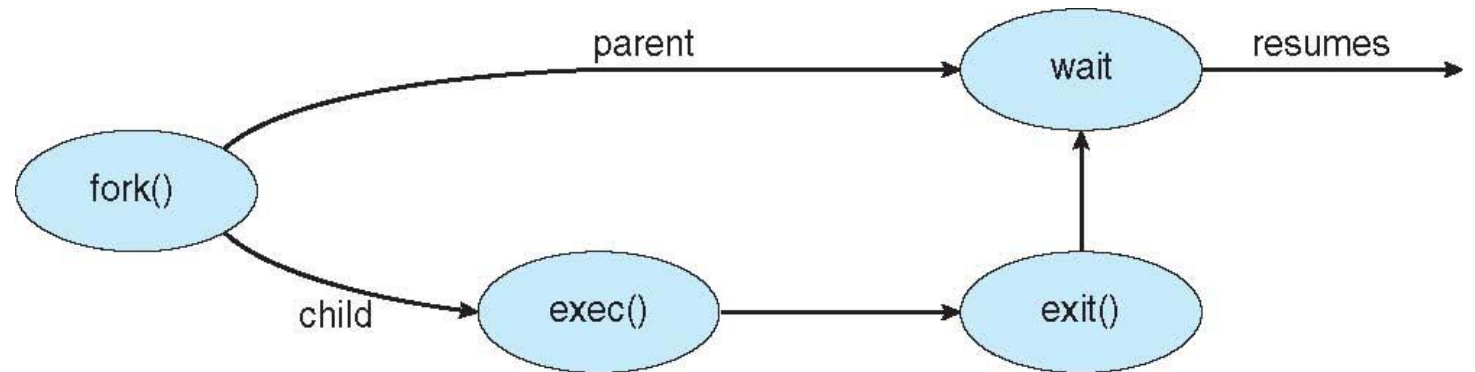
- Child duplicate of parent
- Child has a program loaded into it

A Tree of Processes in Linux



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



C Program Forking Separate Process

```
#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;
}
```


Creating a Separate Process via Windows API

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

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");

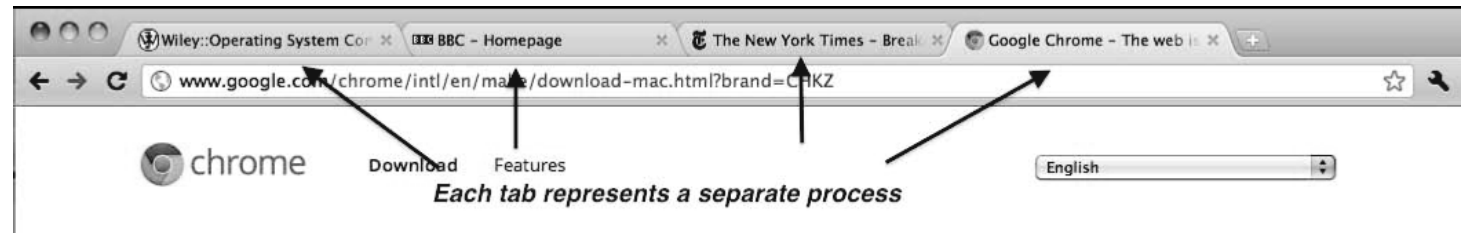
    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the OS does not allow a child to continue if its parent terminates
- Some OS do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.

Multiprocess Architecture – Chrome Browser

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



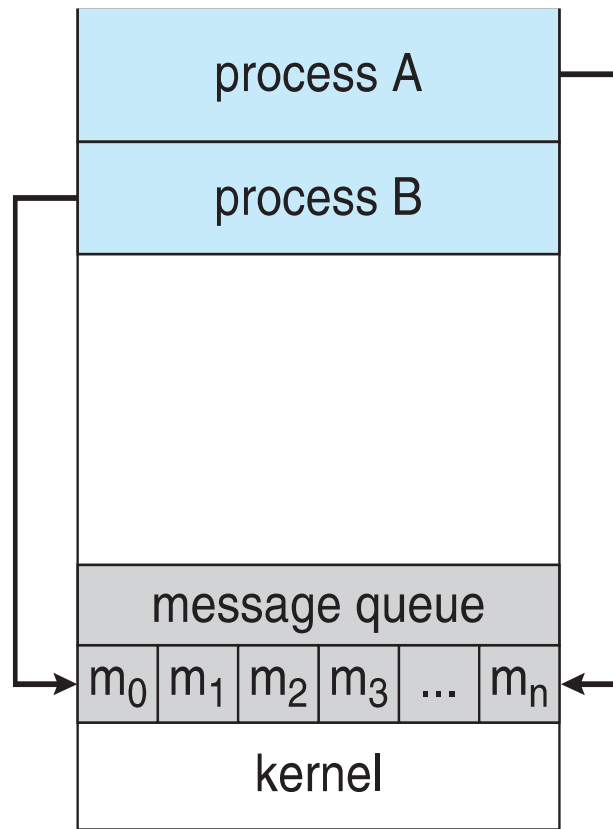


Interprocess Communication

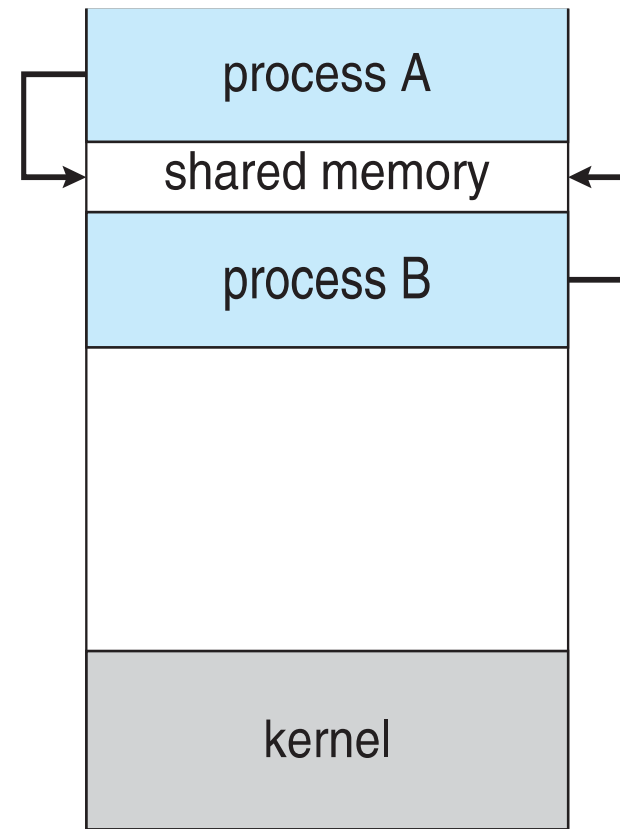
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing
 - Computation speedup
 - Modularity
 - Convenience
- Cooperating processes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

Communications Models

(a) Message passing. (b) shared memory.



(a)



(b)

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
 - **unbounded-buffer** places no practical limit on the size of the buffer
 - **bounded-buffer** assumes that there is a fixed buffer size

Interprocess Communication – Shared Memory

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

Interprocess Communication – Message Passing

Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

- **send**(*message*)
- **receive**(*message*)

The *message* size is either fixed or variable

If processes *P* and *Q* wish to communicate, they need to:

- Establish a **communication link** between them
- Exchange messages via send/receive

Interprocess Communication – Message Passing (contd.)

- Implementation issues:
 - How are links established?
 - Can a link be associated with more than two processes?
 - How many links can there be between every pair of communicating processes?
 - What is the capacity of a link?
 - Is the size of a message that the link can accommodate fixed or variable?
 - Is a link unidirectional or bi-directional?
- Implementation of communication link
 - Physical:
 - Shared memory
 - Hardware bus
 - Network
 - Logical:
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
 - **send** (*P, message*) – send a message to process P
 - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Indirect Communication (contd.)

- Operations
 - create a new mailbox (port)
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - **send**(*A, message*) – send a message to mailbox A
 - **receive**(*A, message*) – receive a message from mailbox A
- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

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

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits

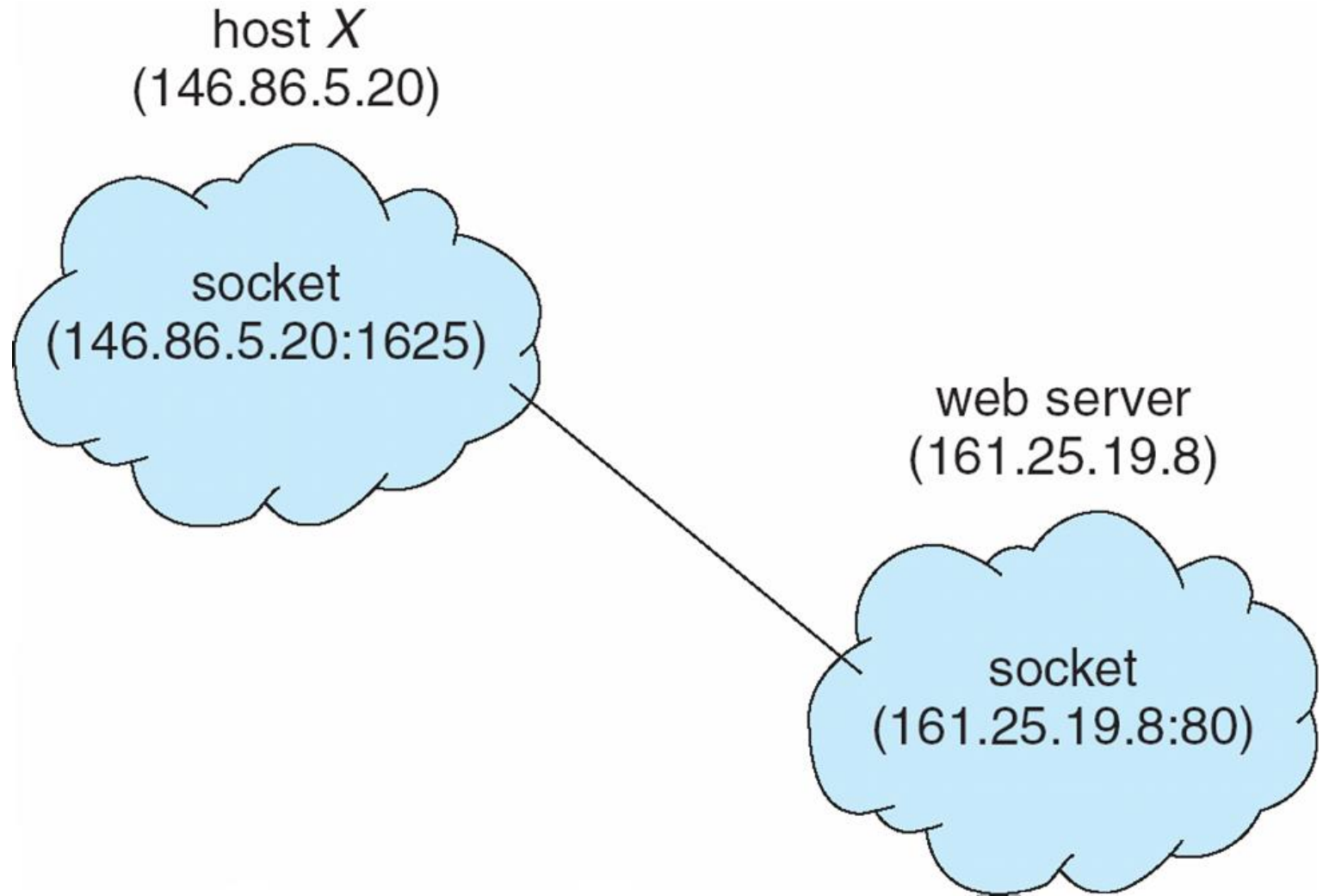
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes

Sockets

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

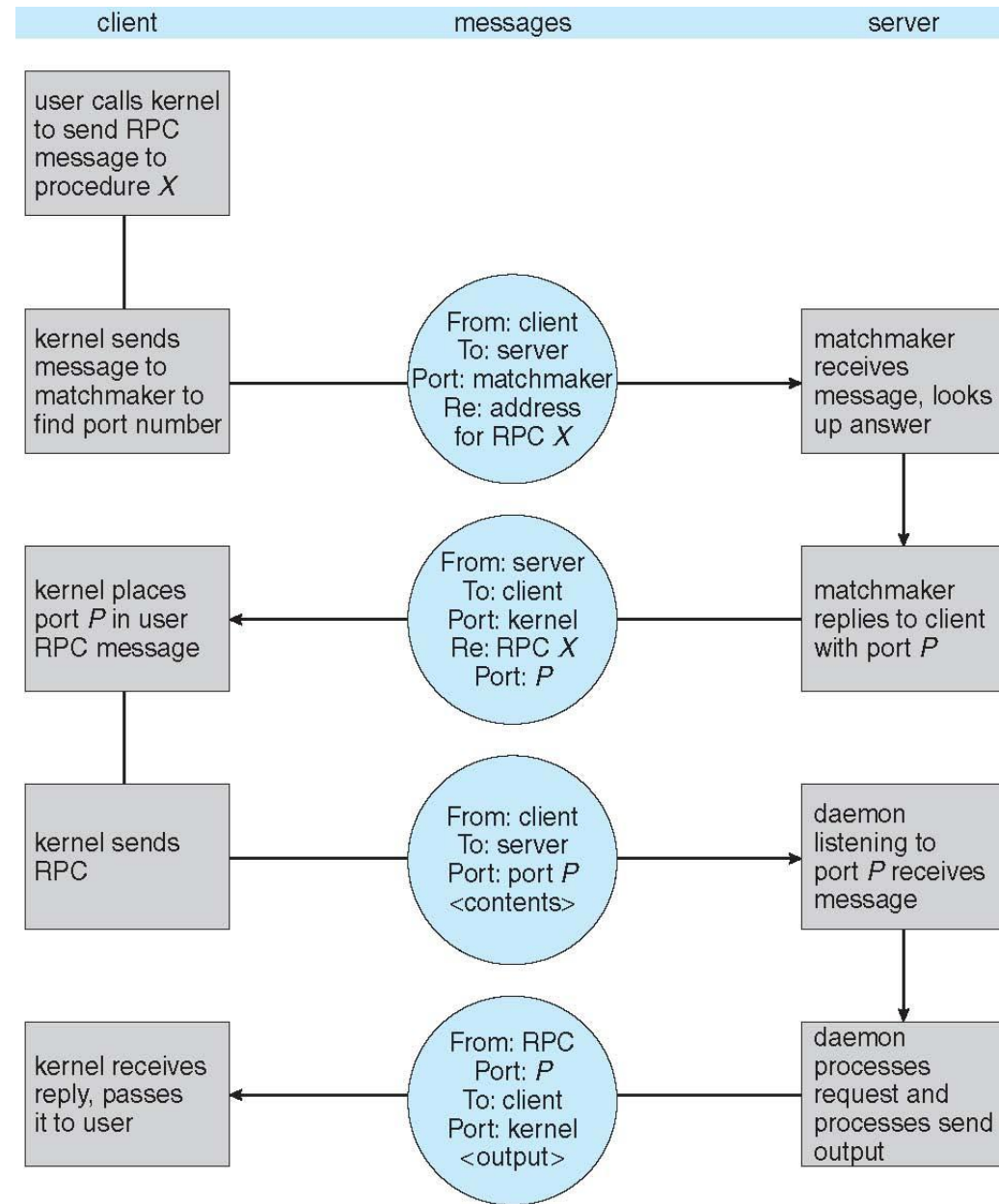
Socket Communication



Remote Procedure Calls

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

Execution of RPC



Pipes

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

Referensi

- Operating System Concepts Ninth Edition ([Avi Silberschatz](#), [Peter Baer Galvin](#), [Greg Gagne](#))

Terima Kasih