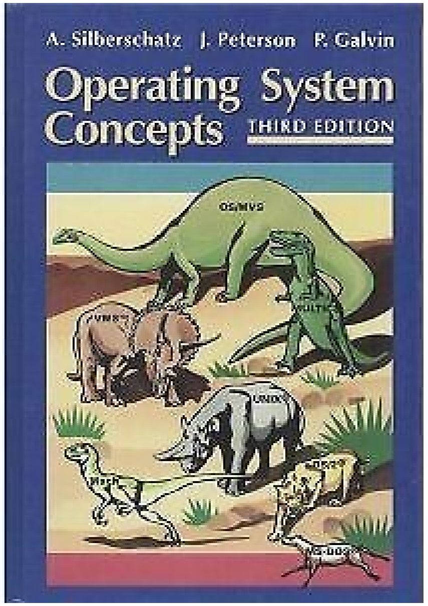
Chapter 3: Process Concept

School of Computing, Gachon Univ.
Joon Yoo



Gachon University
School of Computing

Objectives

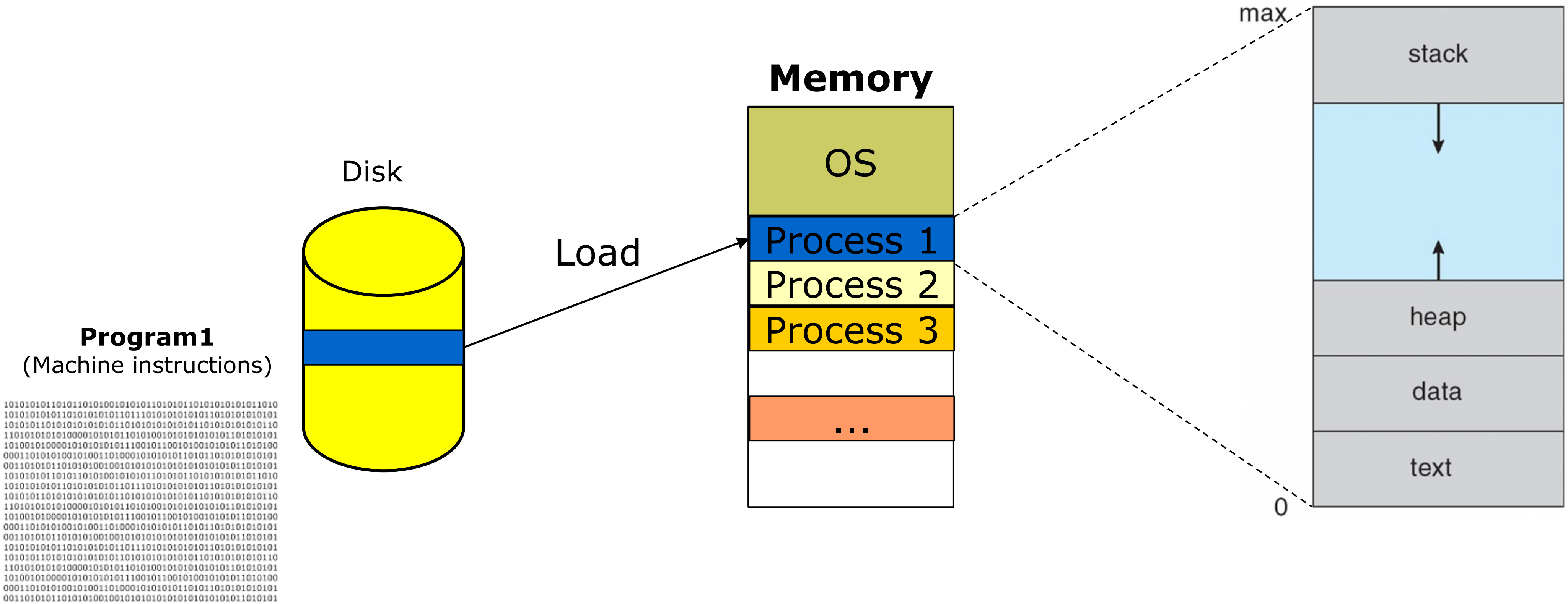
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing

Chapter 3: Process Concept

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

Process Concept

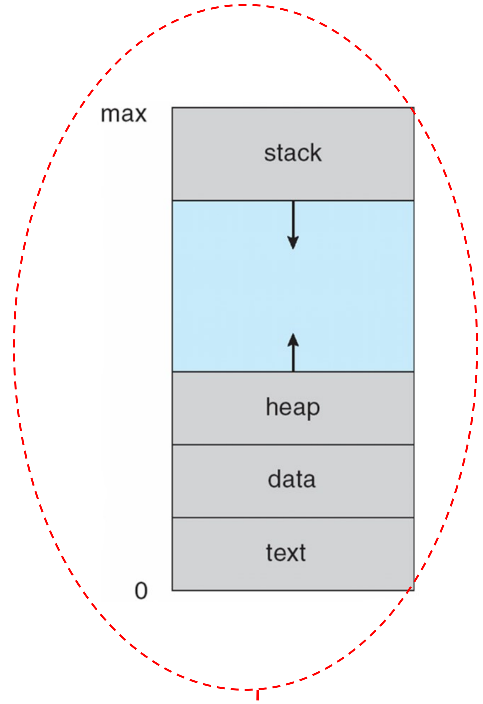
- **Process** (= **job/task, ≠program**)
  - a program *in execution*

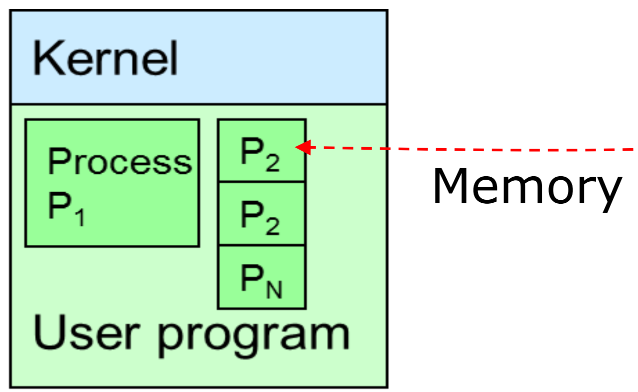


Gachon University
School of Computing

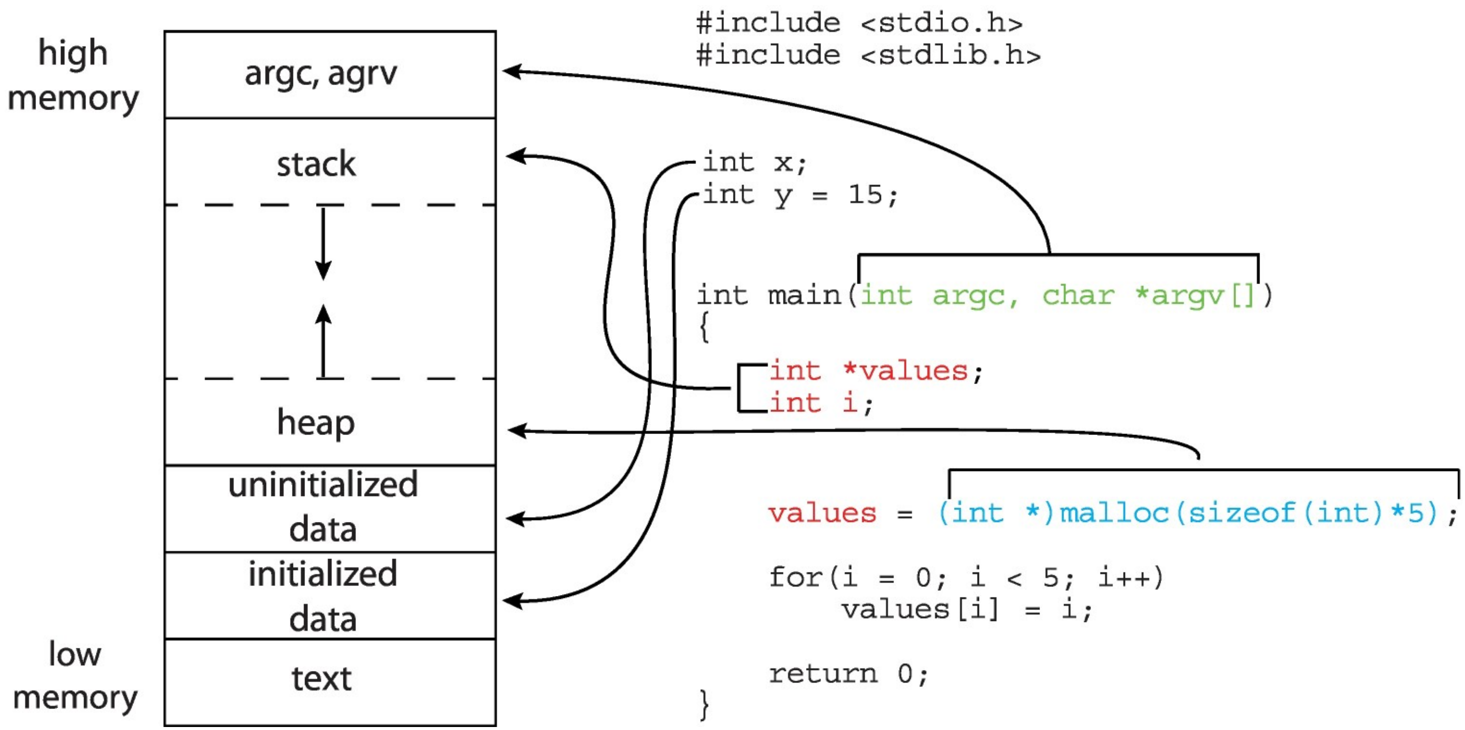
Process in Memory

- Process <u>**in memory**</u>
  - **Text (Code)**: The binary **program instructions**
  - **Data**: Static data. (e.g., global **variables**)
  - **Stack**: temporary local **data**
    - Function parameters, return addresses, local variables
  - **Heap**: memory ***dynamically*** allocated **during run time** (e.g., C malloc(), Java objects)





Memory Layout of a C Program



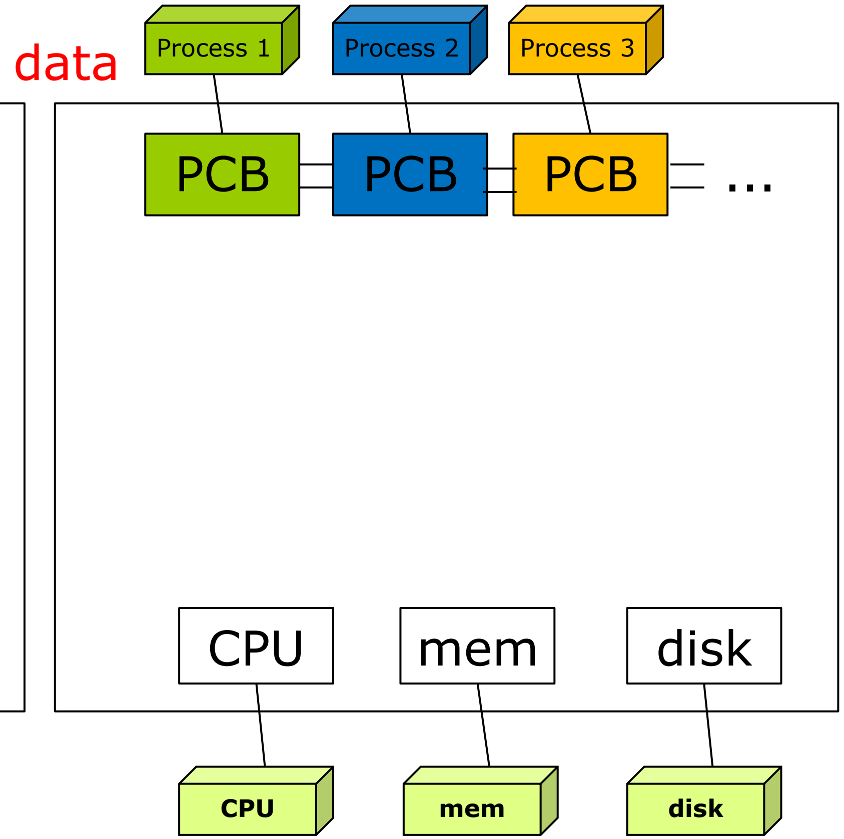
Gachon University
School of Computing

Kernel Memory Space

Kernel code (text)

Kernel code

- System call, Interrupt handling code
- Resource management code
- Others…

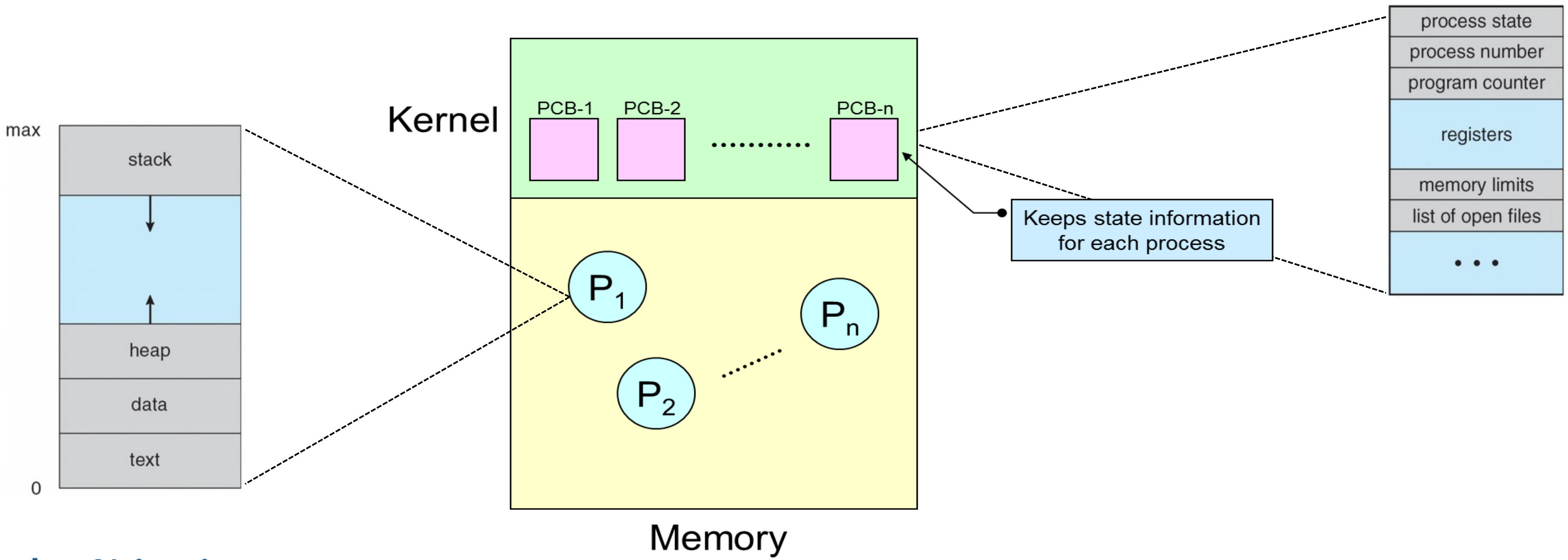


: Table (Data Structure)

: Object (HW or SW)

Process Control Block (PCB)

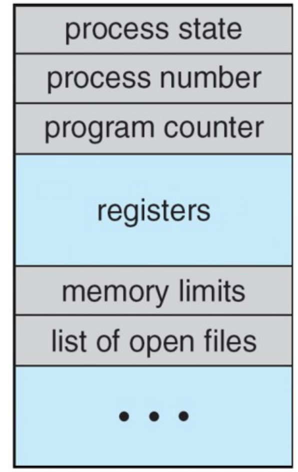
- Each process is registered to and managed by OS
  - manages and take care of all the processes.
  - Therefore, the OS needs to manage the current information (e.g., state) of each process – use a data structure called **PCB (Process Control Block)**



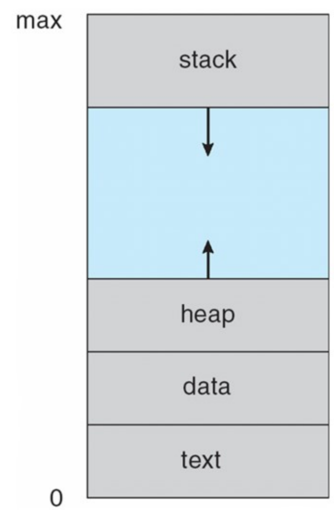
Process Control Block (PCB)

PCB: OS maintains the **information** for *each* process

- Process **state** (next slide)
- Process **number**: process id (pid)
- **Program counter (PC)** – next instruction address
- **CPU registers** – contents of registers (in CPU)
- CPU scheduling information (Ch. 5)
- Memory-management information (Ch. 9-10)
  - Where is the process located in **memory**?
- I/O status information (Ch. 12-15)
  - I/O devices allocated to process, list of open files

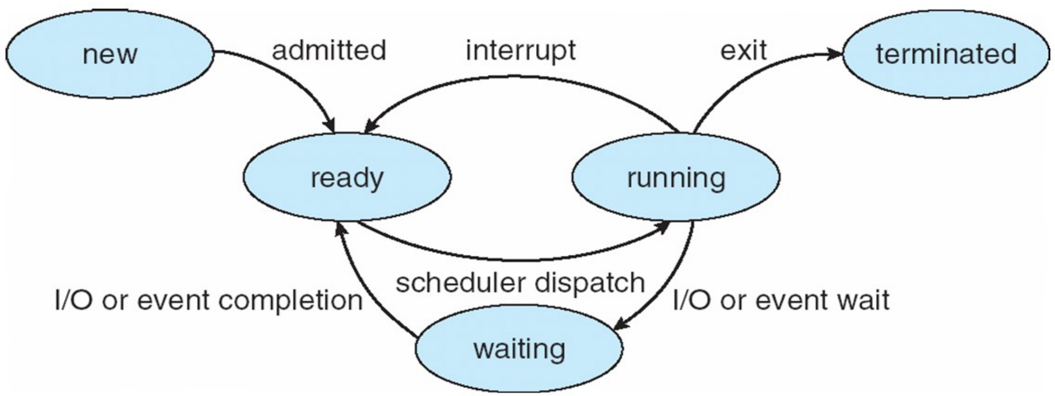


PCB



Process State

- As a process executes, it changes **state**
  - **new**: Before process is created (not a process yet)
  - **ready**: The process (in memory) is ready to be assigned to CPU
  - **running**: Process instructions are being executed by CPU
  - **waiting**: The process is waiting for some event (e.g., I/O operation) to occur
  - **terminated**: The process has *finished* execution (not a process anymore)



Important!

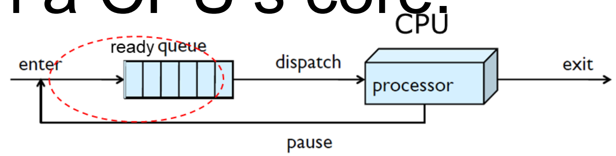
- Only **one** process is **running** on any CPU core at any instant
- All the other processes are waiting in **ready** or **waiting** states

Chapter 3: Process Concept

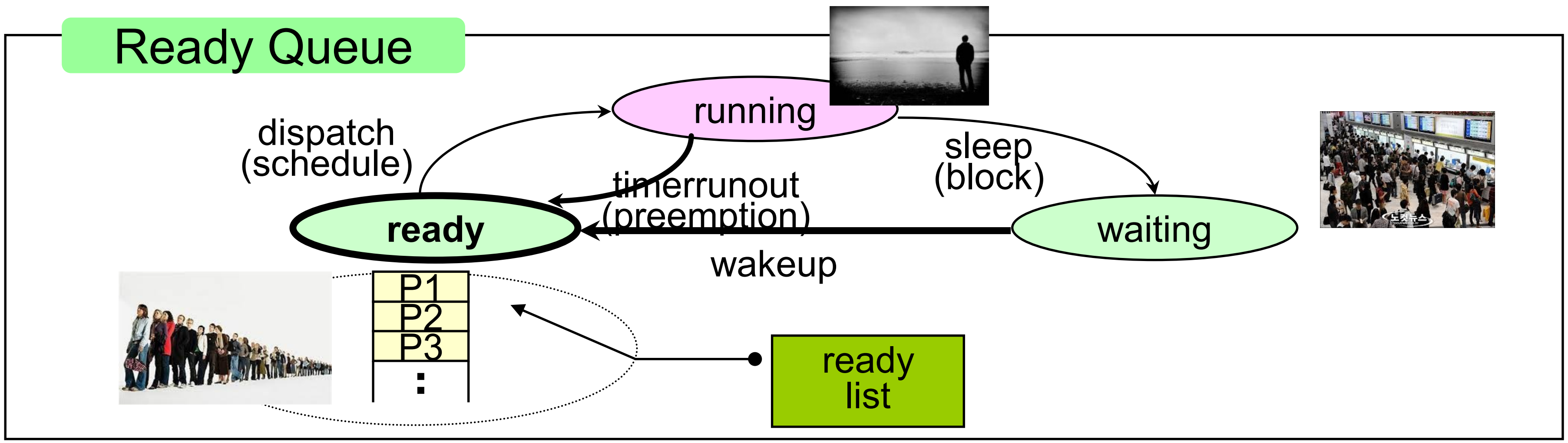
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication

Scheduling Queues

- **Ready queue**
  - The processes that are ready to execute on a CPU's core. (in **Ready state**; waiting for **CPU**)

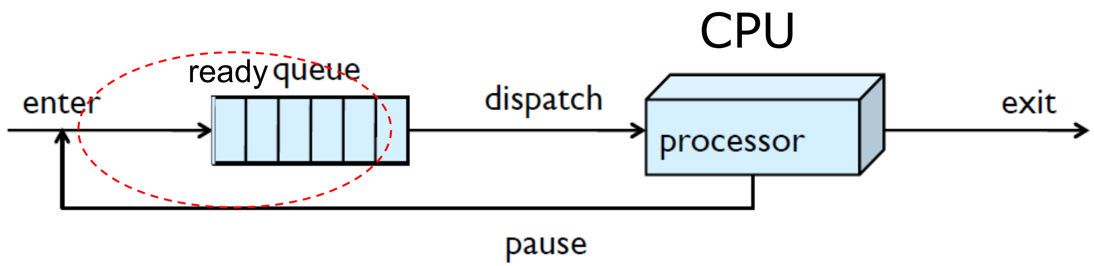


  - Question: How many processes can be **running** at the same time?
  - Question: How many processes can be **ready** at the same time?



CPU Scheduler

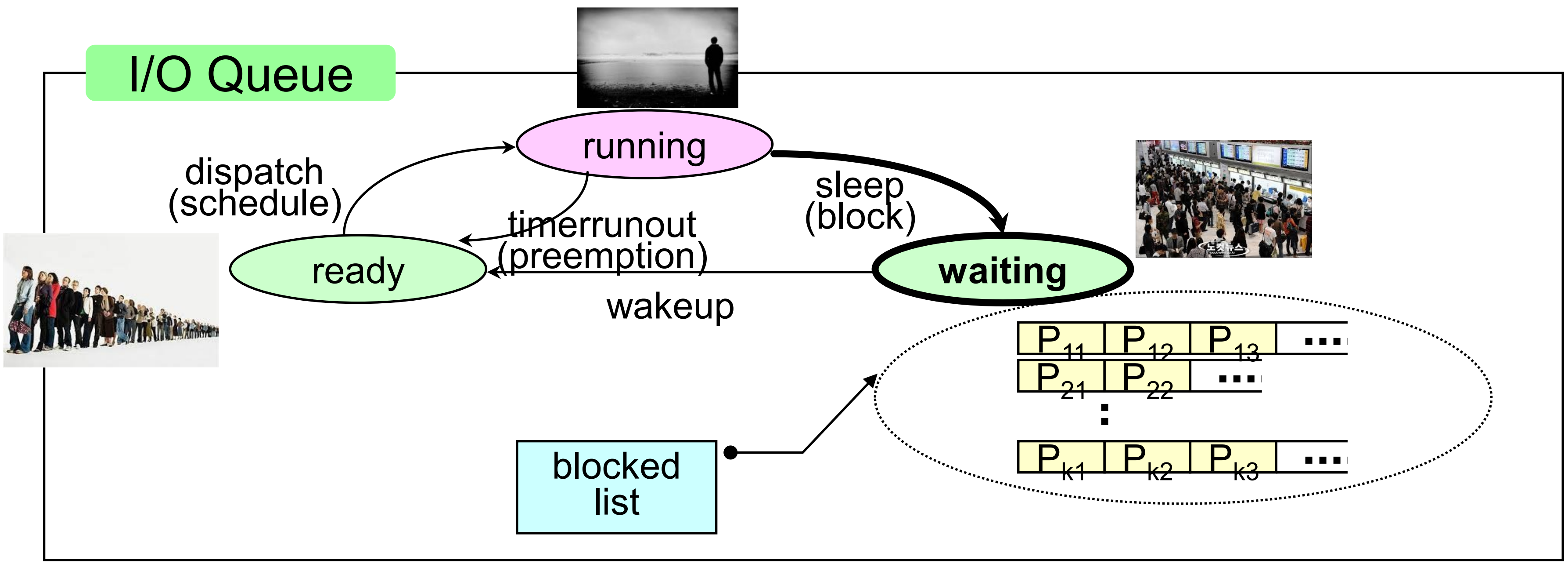
- **CPU Scheduler**
  - Selects from among the processes in ready queue, and allocate the CPU core to one of them.



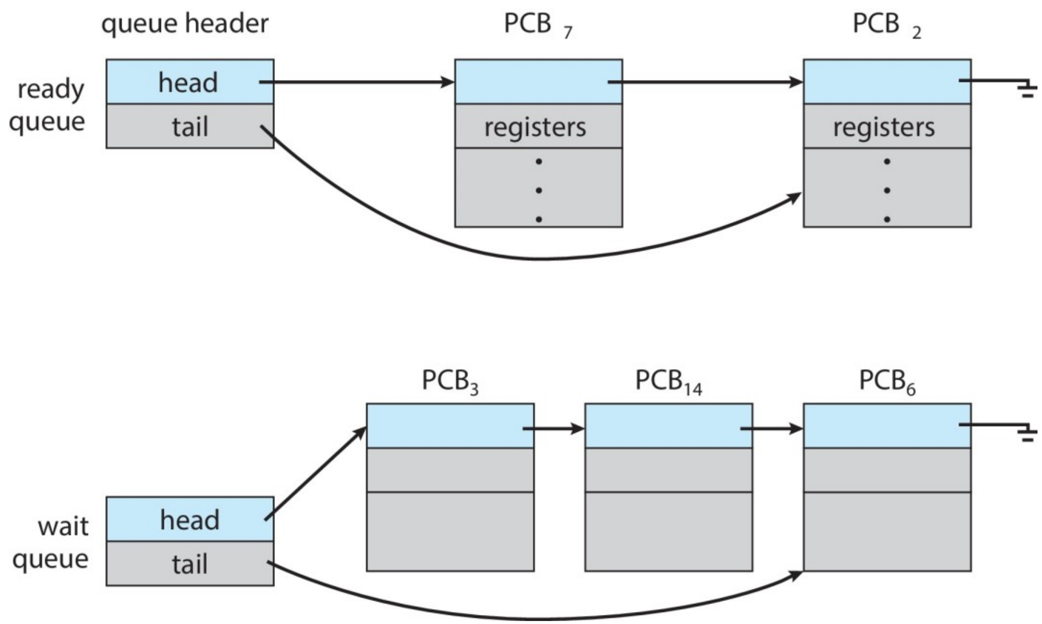
  - Use CPU scheduling algorithms (Ch. 5)

Scheduling Queues

- **Waiting queue**
  - When a process is allocated the CPU, it **executes** for a while and eventually **quits** or **interrupted**
  - or **waits for an event** (e.g., I/O completion)
    - The process has to be **waiting** (or **blocked**) in the **wait queue**
    - Also called Blocked list (block queue, I/O queue)

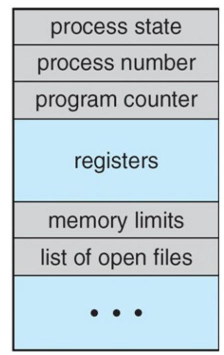


Ready Queue and Wait Queues Linux Data Structure



I/O (device) queue

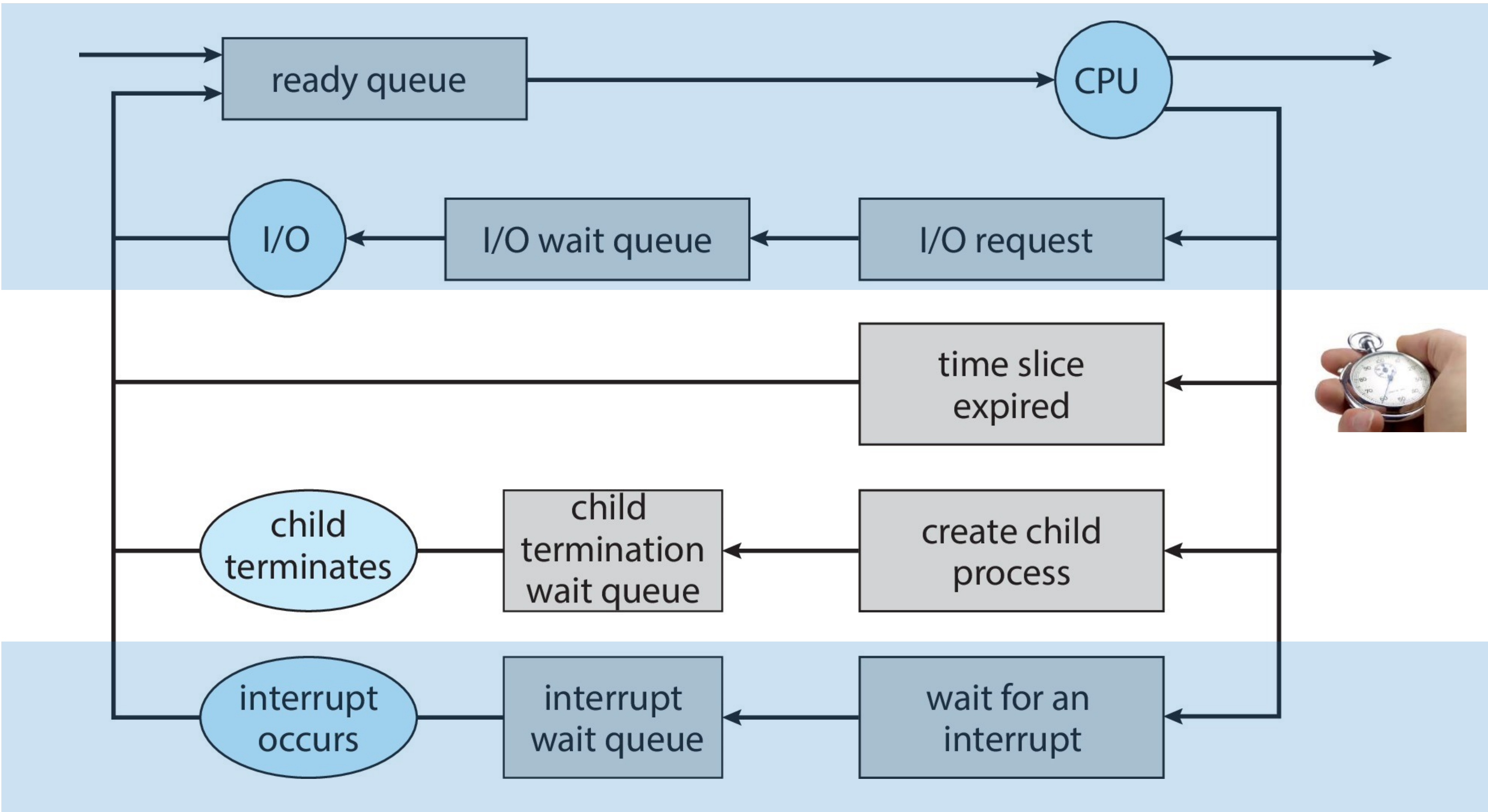
- <include/linux/sched.h>
- Queues are usually implemented with linked lists
  - Queue header → pointing the first and last **PCB** structures
  - Each **PCB** structure has the address of its next **PCB** structure



PCB

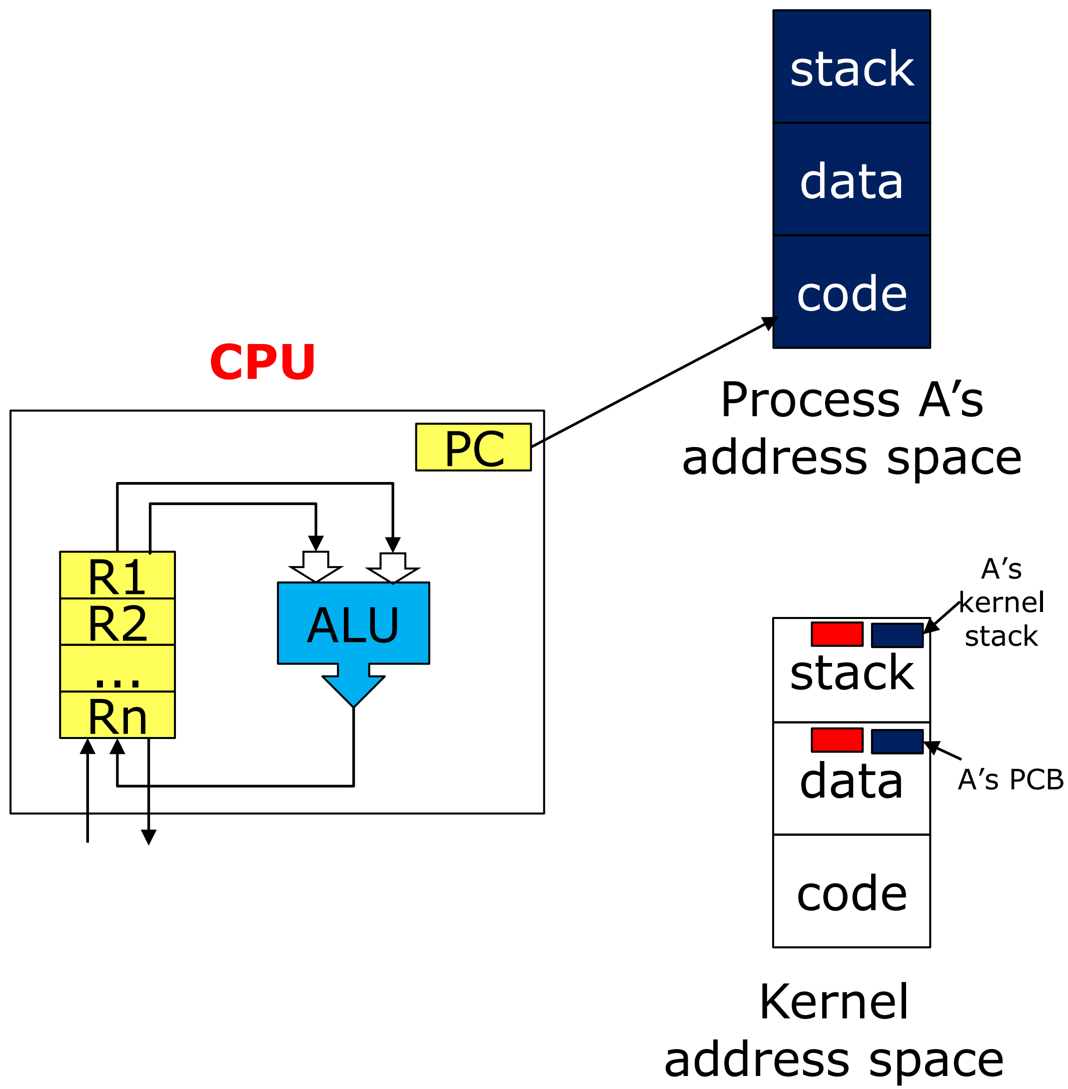
Representation of CPU Scheduling

Queuing diagram



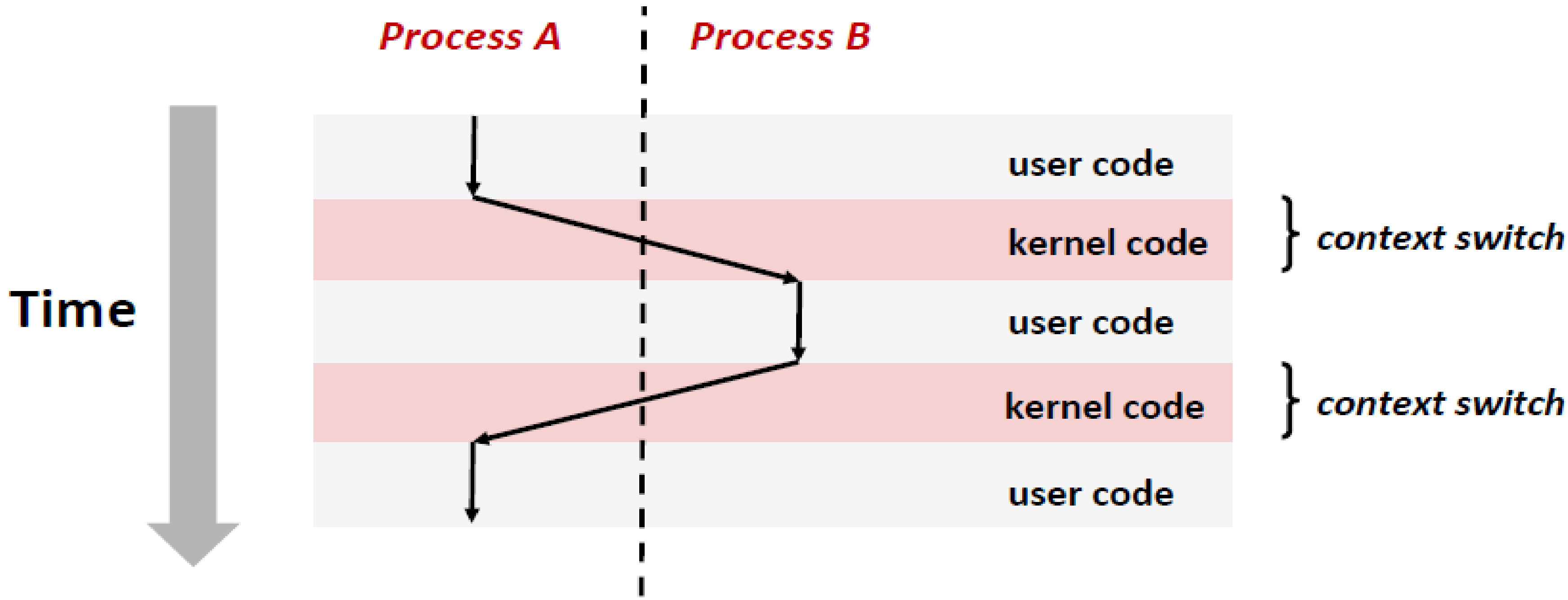
Process Context

- CPU execution context
  - Program Counter (PC)
  - Registers
- Process memory space
  - code, data, stack
- Process management in OS
  - Process Control Block (PCB)
  - Kernel stack



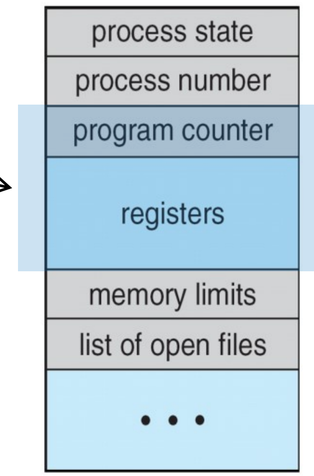
Context Switch

- Process switching from one process to another by OS !
  - called **Context switch**

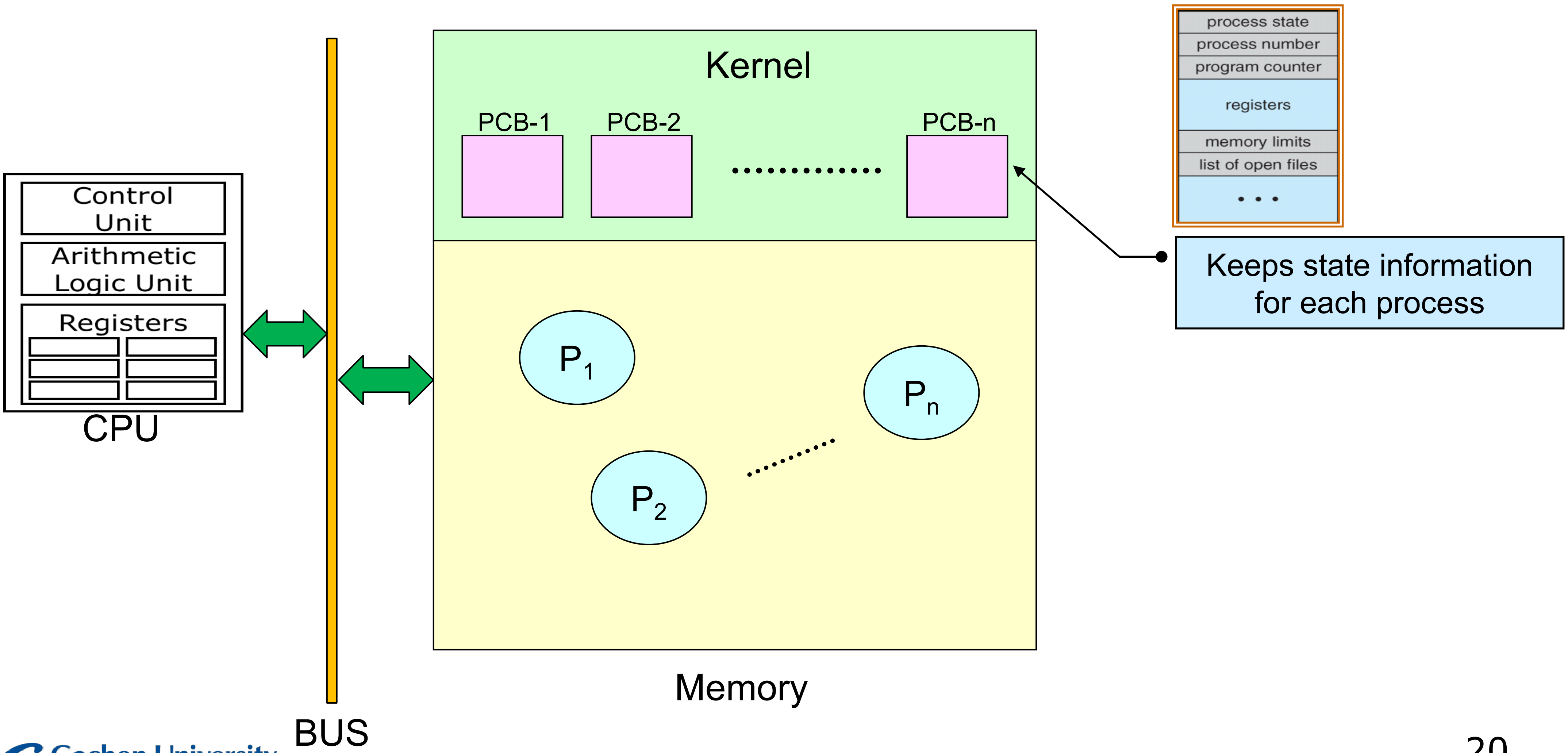


Context Switch

- When CPU switches to ***run*** another process
  - the system must **save the state (context)** of the old process (to resume where we stopped) and
  - load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB

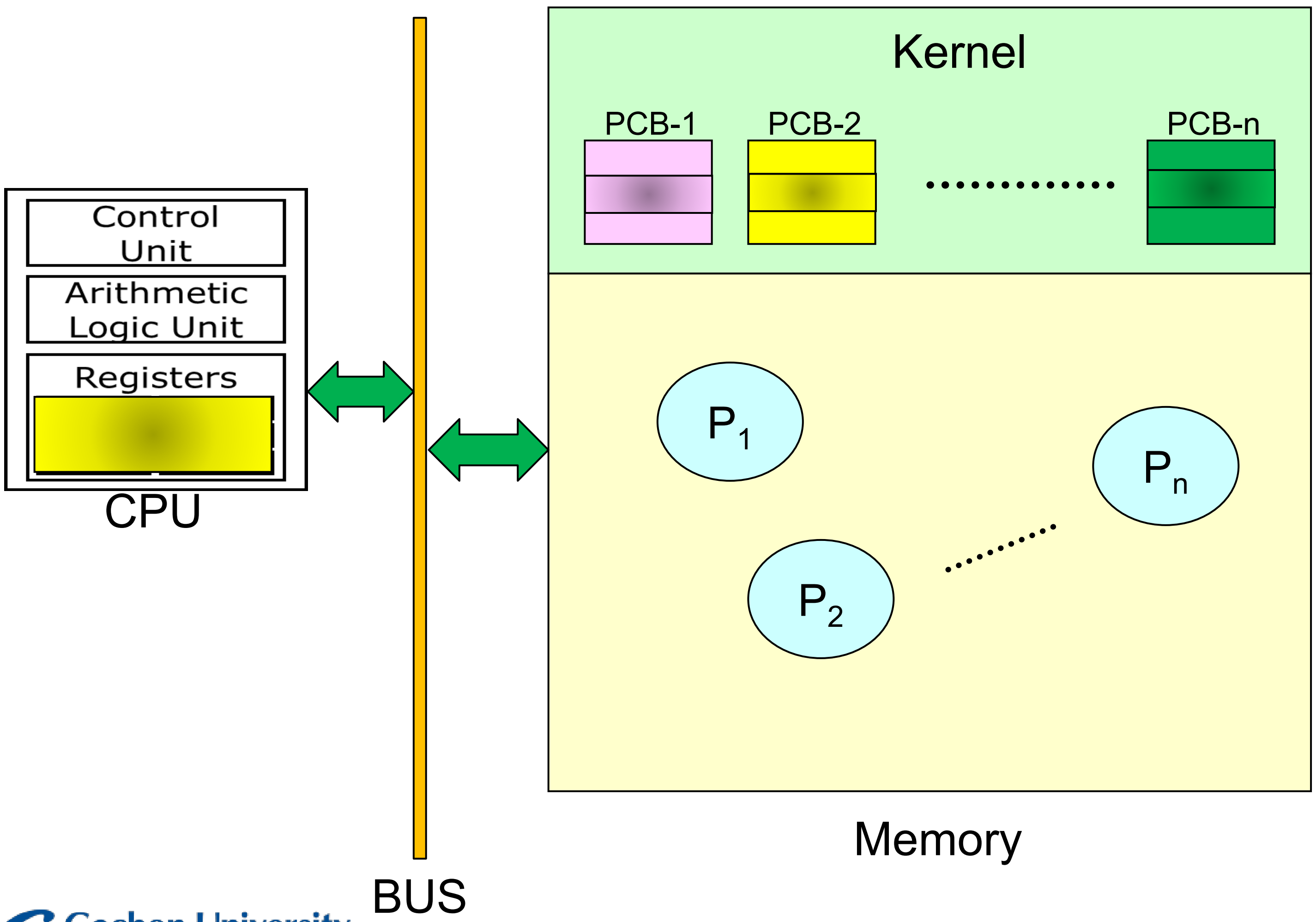


(Cont.)

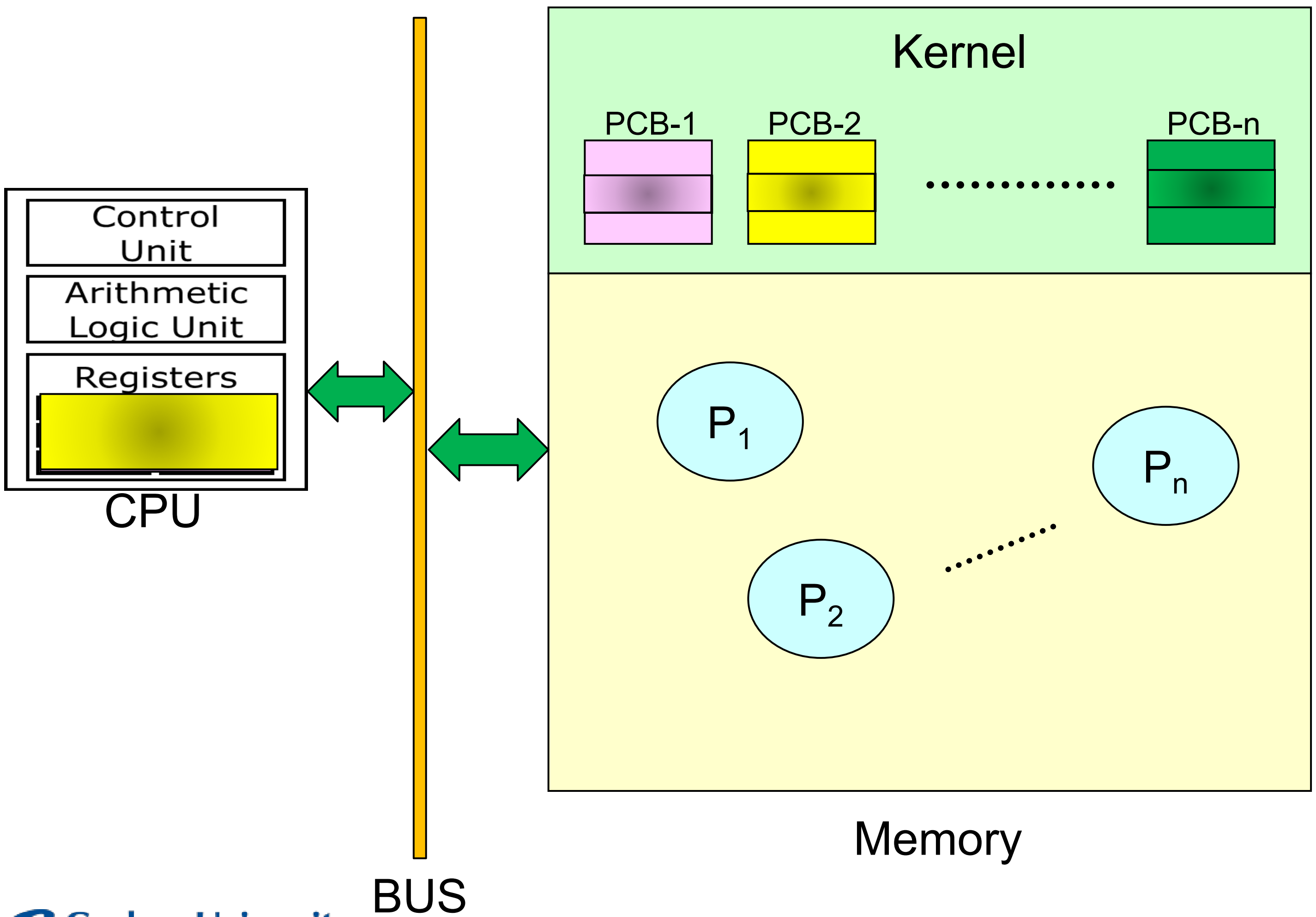


Kernel
PCB-1
PCB-2
PCB-n
process state
process number
program counter
registers
memory limits
list of open files
• • •
Keeps state information for each process
Control Unit
Arithmetic Logic Unit
Registers
CPU
P1
P2
Pn
Memory
BUS

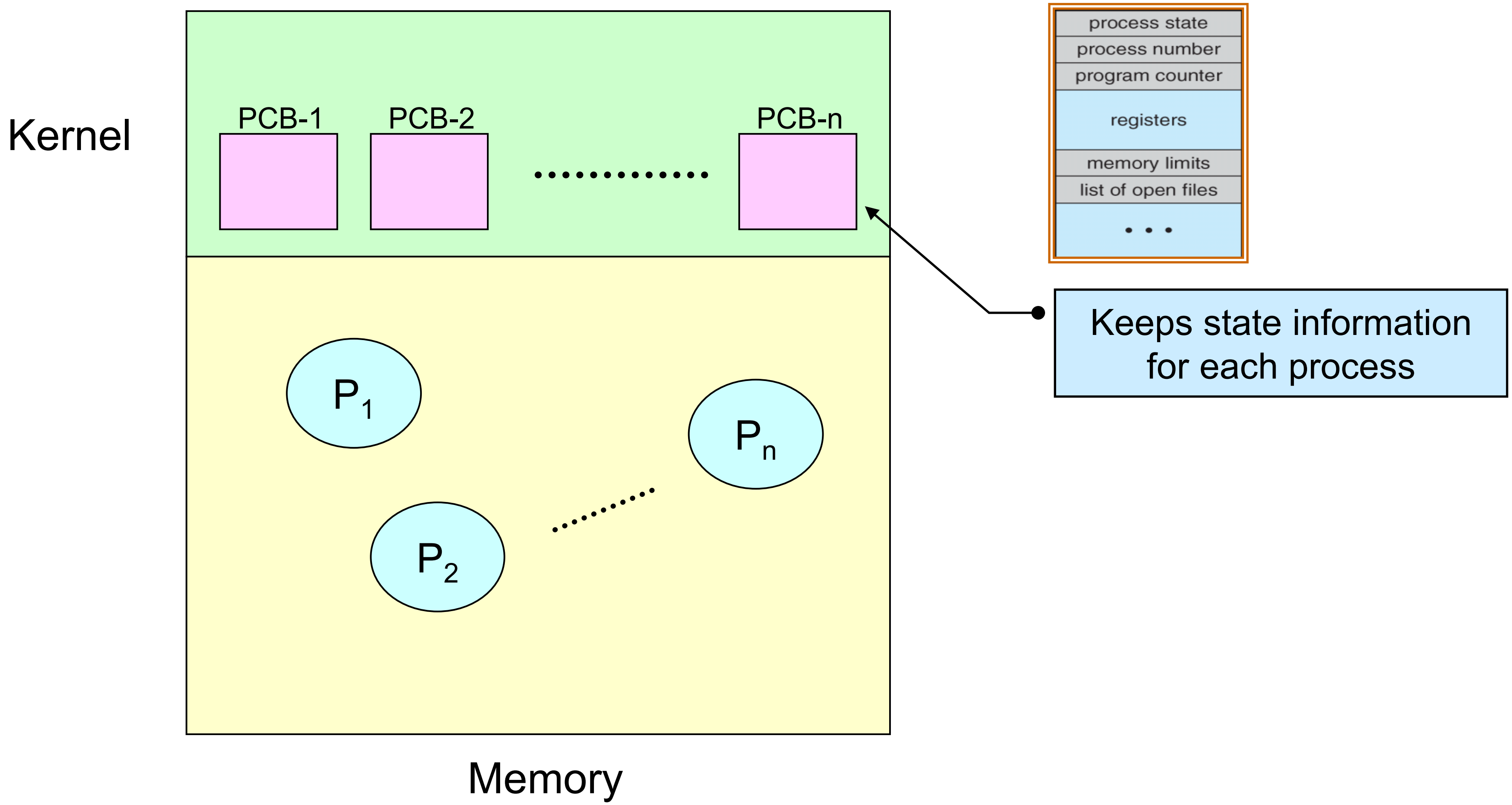
(Cont.)


Kernel
PCB-1
PCB-2
PCB-n
Control Unit
Arithmetic Logic Unit
Registers
CPU
P_1
P_n
P_2
Memory
BUS

(Cont.)


Control Unit
Arithmetic Logic Unit
Registers
CPU
BUS
Kernel
PCB-1
PCB-2
PCB-n
P_1
P_2
P_n
Memory

(Cont.)



Kernel
PCB-1
PCB-2
PCB-n
P1
P2
Pn
Memory
process state
process number
program counter
registers
memory limits
list of open files
• • •
Keeps state information for each process

Context Switch Overhead

- Context-switch time is **overhead**; the system (or CPU) cannot do any other work while switching
  - context switch time depends on memory speed, number of registers that need to be copied, ...
  - typically takes several microseconds

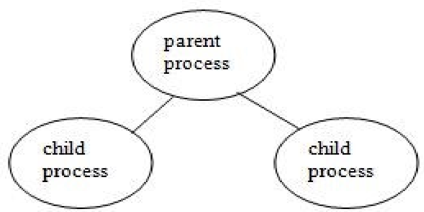
- Context-switch overhead is an important factor in CPU scheduling

Chapter 3: Process Concept

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

Operations on Processes

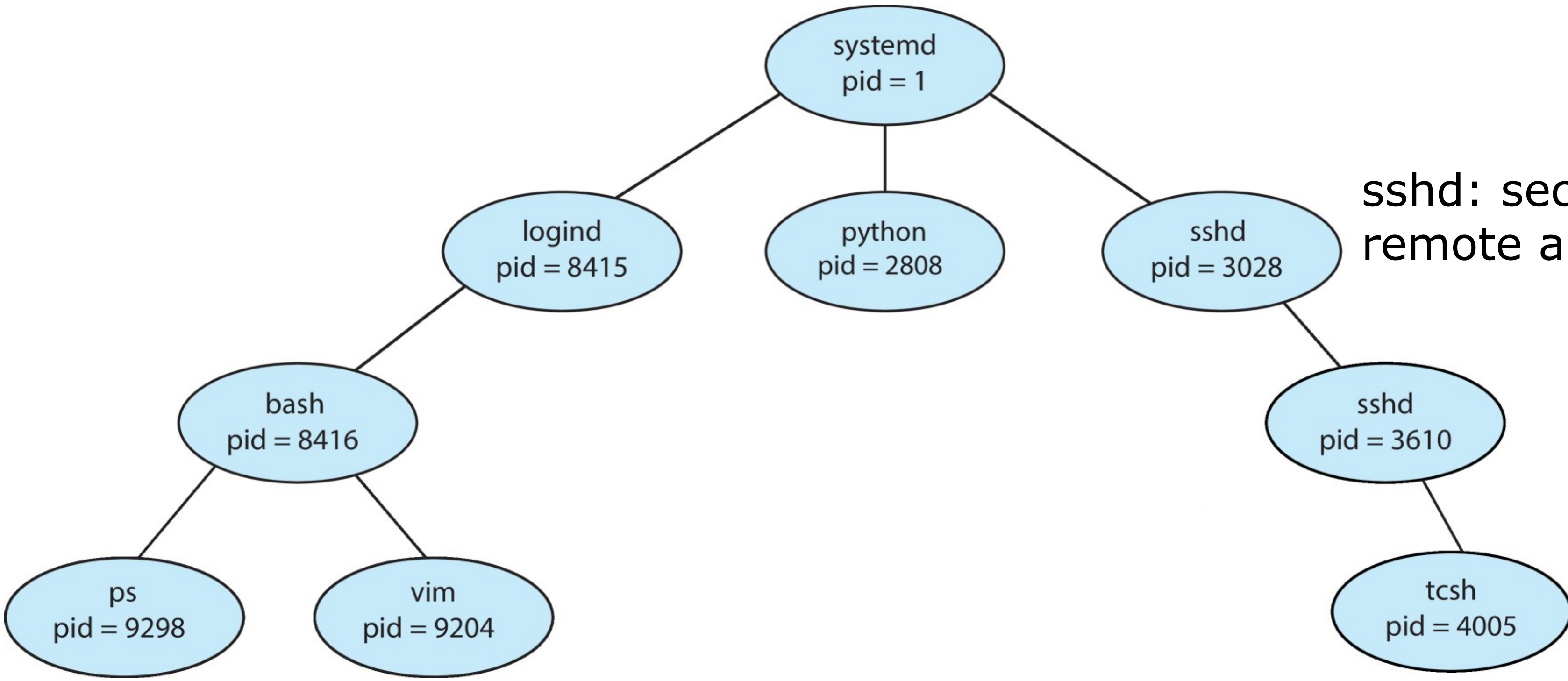
- Processes can execute concurrently, thus they may be **created** and **deleted** dynamically.
  - Operating System must provide mechanisms for process creation, termination, and so on.
  - Generally, process identified and managed via a unique **process identifier** (**pid**), typically an integer number
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes



  - So who created this universe (i.e., the first process)?

A Tree of Processes in Linux

Root process for all user processes

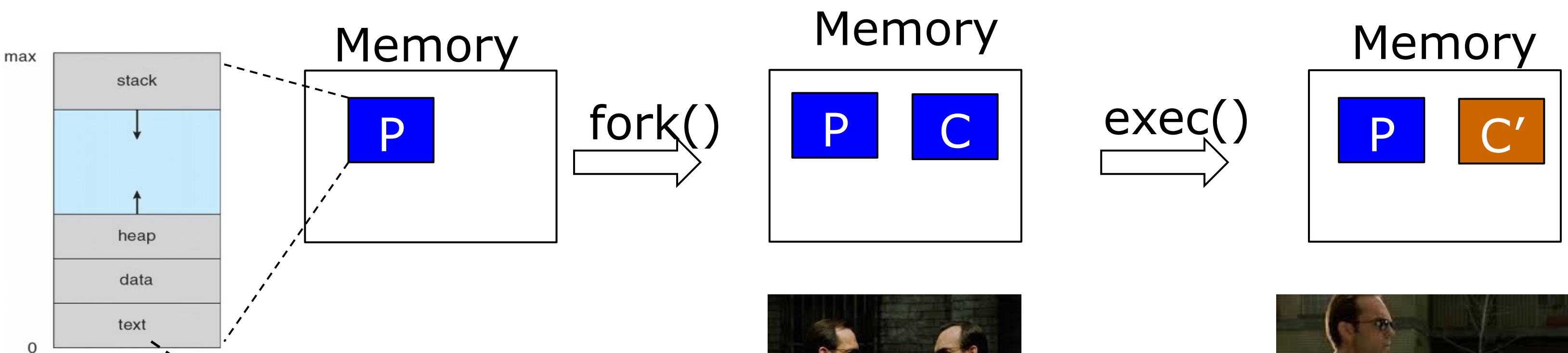


sshd: secure shell remote access

type ps –el in Linux shell

UNIX examples: Process Creation

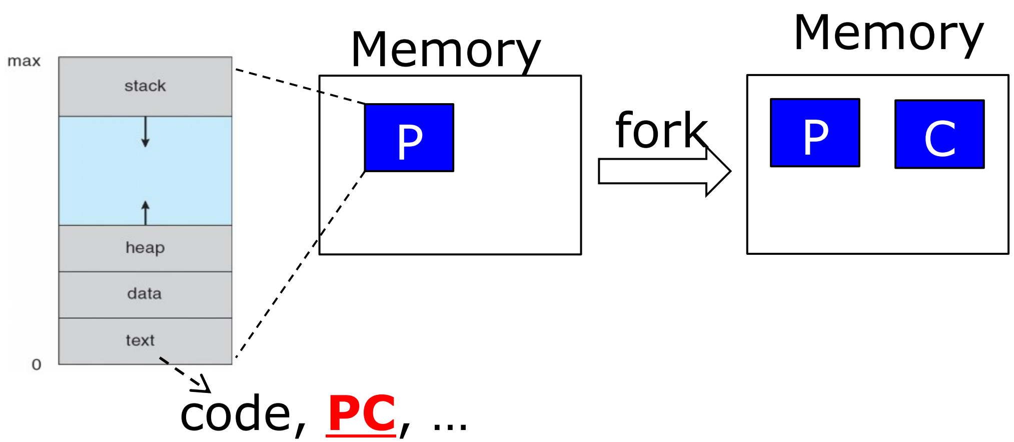
- fork()
  - `fork()` system call creates a new process
  - New process consists of a copy of parent's memory space
  - Both processes continue execution
- exec()
  - `exec()` system call used after a `fork()`, to *replace* the process' memory space with a new program
  - loads a new binary file into memory (deletes original memory – copy of parent)



UNIX examples: Process Creation

- UNIX examples
  - **`fork()`** system call creates a new process
- Both processes (parent and child)
  - continue execution after **`fork()`**,
  - with one difference: **<u>`fork()` return value</u>**:
    - Child process: 0
    - Parent process: **`pid`** of child process (>0)

      process identifier (pid)

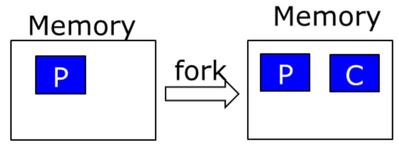



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```

Parent

Child

Fork Timeline Example



Parent (100)

```
#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;
}
```

Parent

Child (150)

```
#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;
}
```

Child

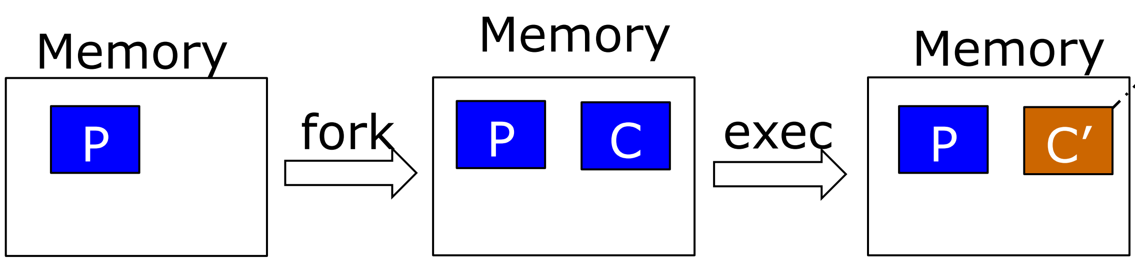
Process Creation

- Memory address space
  - Child duplicate of parent: fork()
  - Child has a program loaded into it: exec() system call
    - **exec()** system call used after a **fork()** to *replace* the process' memory space with a new program

Parent

```c
#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;
}
```

Child

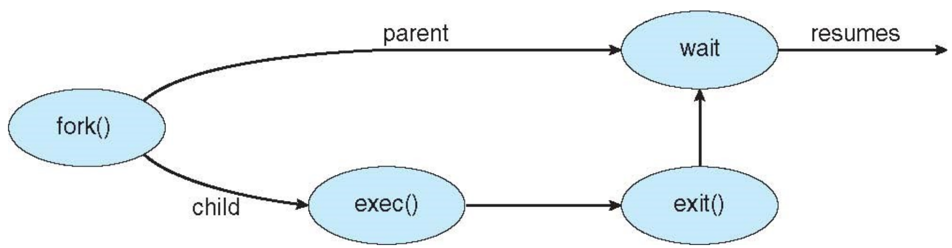


Process Creation

- Execution options
  - Parent and children: Which executes first?
  - Parent can
    - execute concurrently with child or
    - wait until children terminate: `wait()` system call returning the pid:

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

```
#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 Termination

- Process asks the operating system to delete itself: **`exit()`** system call
  - Return status value (integer) to parent process (via **`wait()`**)

- Parent may **terminate** execution of children processes: **`abort()`** system call
  - E.g., child has exceeded allocated resources, task assigned to child is no longer required
  - If parent is exiting, some operating systems do not allow child to continue if its parent terminates
    - All children terminated - **`cascading termination`**

Chapter 3: Process Concept

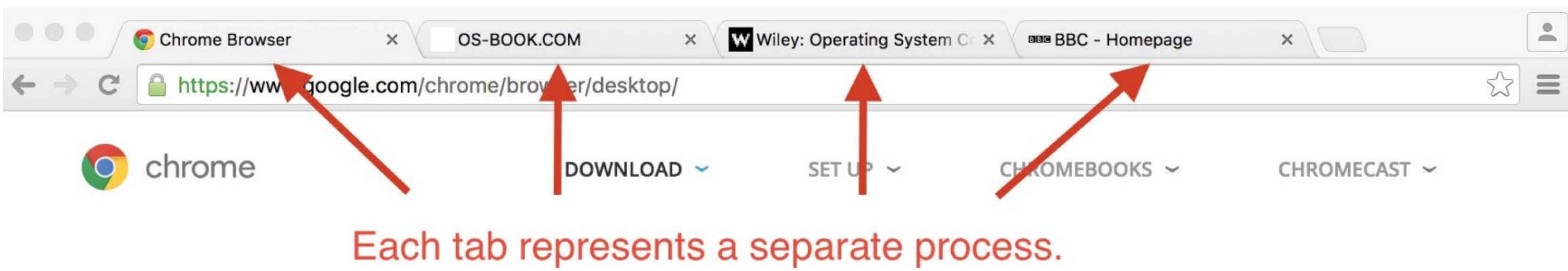
- Process Concept
- Process scheduling
- Operations on Processes
- Interprocess Communication

Interprocess Communication

- In many cases, processes can ***cooperate*** with each others
  - Reasons for cooperating processes:
    - Information sharing – e.g., shared file
    - Computation speedup – e.g., parallel computing in multi-core environment
- Why can't processes just communicate with each other?
  - A process cannot directly access other process's **memory** – why not?
    - A: For system _______________
- Cooperating processes need **interprocess communication (IPC)** provided by OS via system call

Example: Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble (e.g., JavaScript, Flash, HTML5), entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 categories
  - **Browser** process manages user interface, disk and network I/O (1 process created)
  - **Renderer** process renders web pages (deals with HTML, Javascript), <u>new process</u> for each website opened in a <u>new tab</u>
  - **Plug-in** process for each type of plug-in (e.g. Flash, QuickTime)



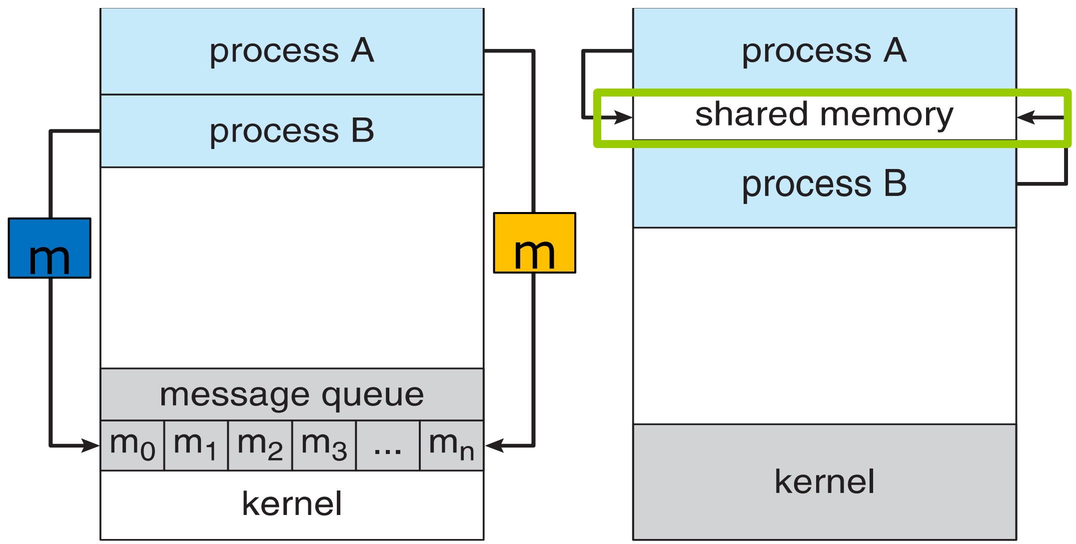
Communications Models

Mechanism

- for processes to **communicate** & to **synchronize** their actions

Two fundamental models of IPC

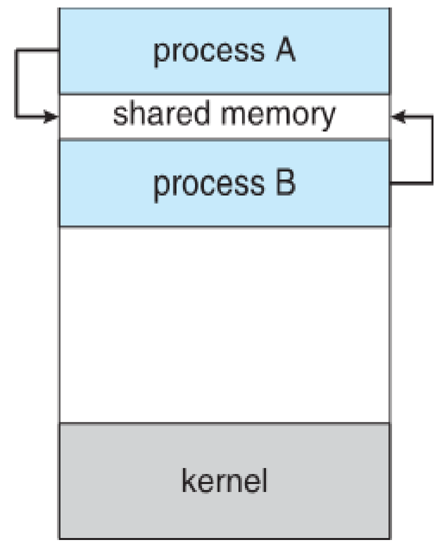
- **Shared memory**
- **Message passing**



(a) Message passing (b): Shared memory

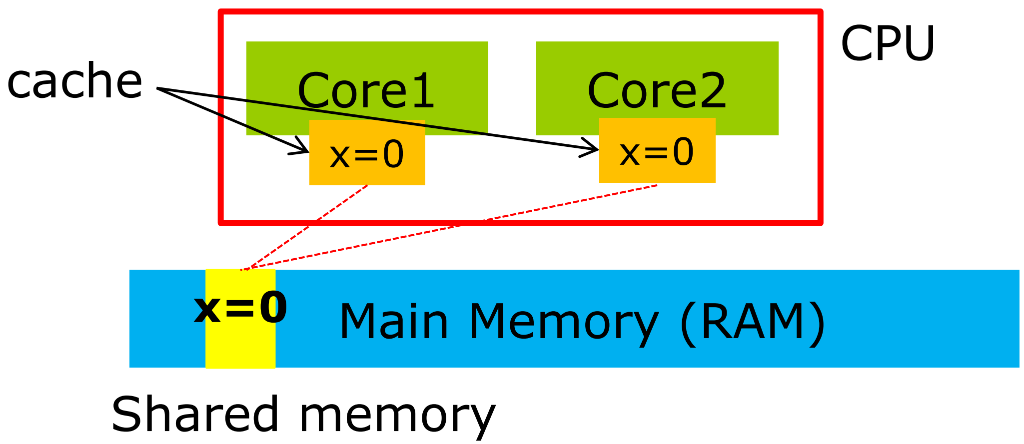
Shared-Memory Systems

- OS prevents one process from accessing another process's memory – Protection
  - **Shared memory**: Two or more processes agree to remove this restriction
- A region of memory that is shared is created by system call
  - Processes exchange information by reading and writing data on shared area
  - All access to shared memory are treated as routine memory access – no need for system call



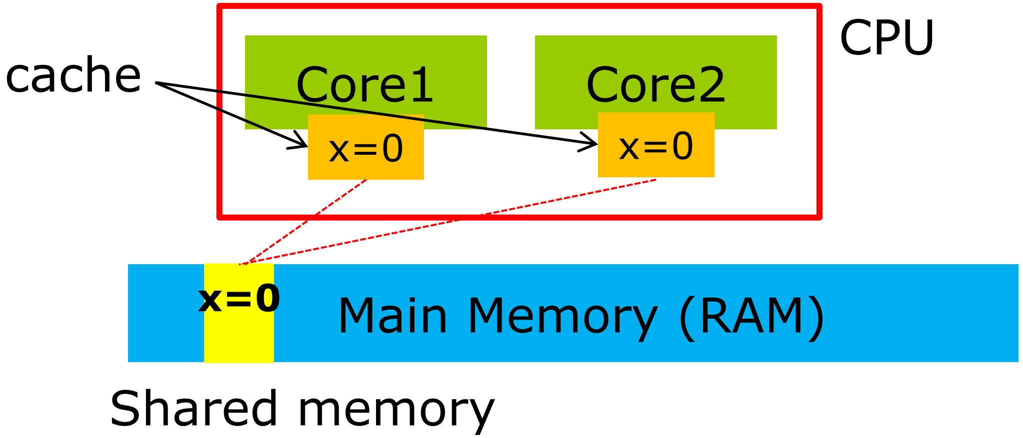
Shared Memory Systems

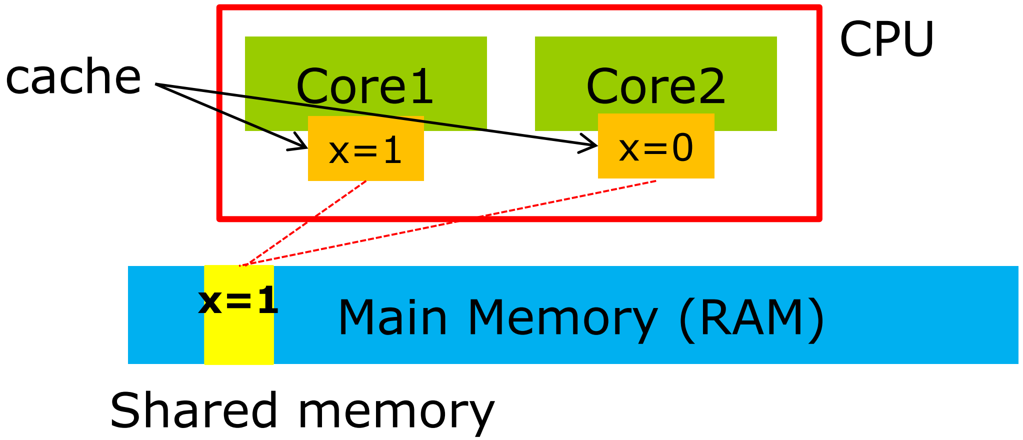
- **Bigger problem:** What happens if two processes attempt to access the shared memory concurrently?
  - Process synchronization (Ch. 6)
- **Another problem:** Multicore processors
  - Each core have separate cache – cache coherence problem



Multicore processor: Cache coherence problem

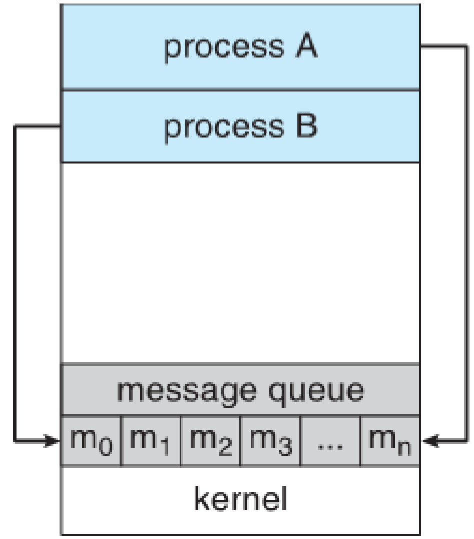
1. process1 in core1 reads "x=0" from shared memory – stored in core1's cache
2. process2 in core2 reads same "x=0" from shared memory – stored in core2's cache
3. process1 in core1 changes data to "x=1"
   - updated core1's cache and shared memory to "x=1"
   - But core2's cache still has "x=0"
   - **Cache coherence problem!**
     - Usually solved by CPU cache hardware





Interprocess Communication – Message Passing

- Message system – processes communicate with each other by **messages** without resorting to shared variables
- IPC facility provides two operations via system call:
  - **`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 system call
    - need ***system call*** *for every message*
    - More time-consuming compared to shared memory
  - e.g., **Microkernel structure**
  - e.g., **Sockets** (networking), Remote Procedure Call (RPC: cloud computing)



https://www.proactiveinvestors.co.uk/