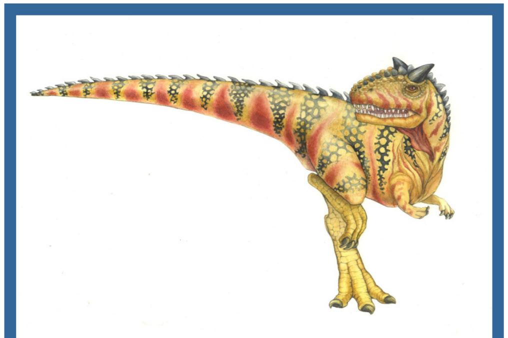
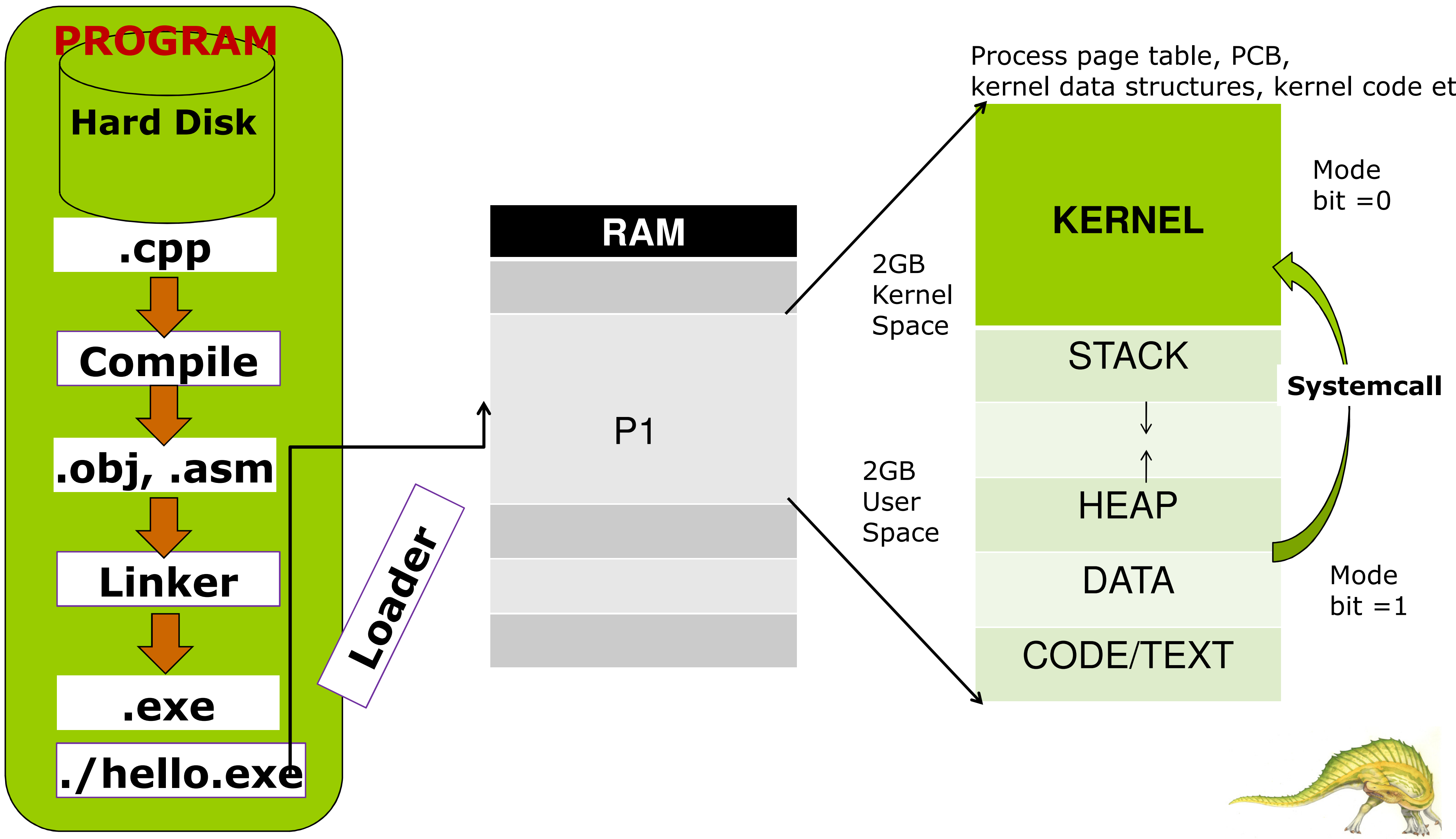
Lecture 4, 5
Process Management



Program to Process



Memory Organization for an Executed Program

- When a program is loaded into memory, user space is organized into four regions of memory, called segments:

  text segment, data segment, stack segment & heap segment

- **Text segment (or code segment)**
  - where the compiled code of the program itself resides.
- **Data segment (Data & BSS)**
  - data area contains
    - global or static or resister variables that are initialized.
  - BSS contains
    - global or static or register variables that are uninitialized.
  - Pointer variable int *arr ; declared in global then in data else in stack
- **Heap segment**
  - dynamically allocated variables are allocated in here.
  - it is managed by malloc and free.

Stack segment

• contains the program stack, a LIFO structure.

•$sp register point to the top of the stack.

•memory is allocated for automatic variables within functions.

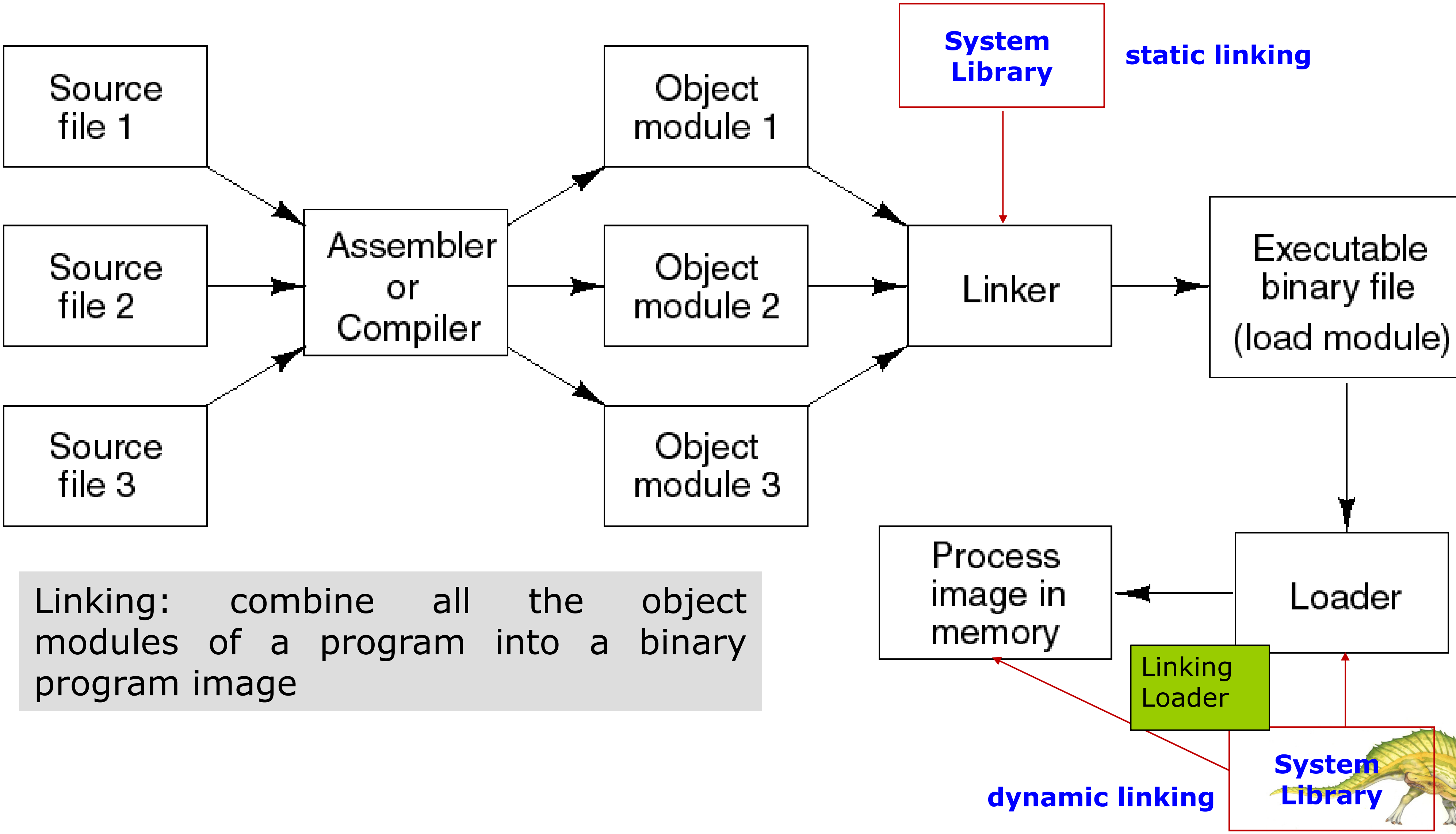
The Process-Executable Program

- We write a program in e.g., Java.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```
void X (int b) {
    if(b == 1) {
...
int main() {
   int a = 2;
   X(a);
}
```

Program counter
next instruction address

Steps for Loading a Process in Memory



Linking: combine all the object modules of a program into a binary program image

Details for running a program

- A program consists of code and data

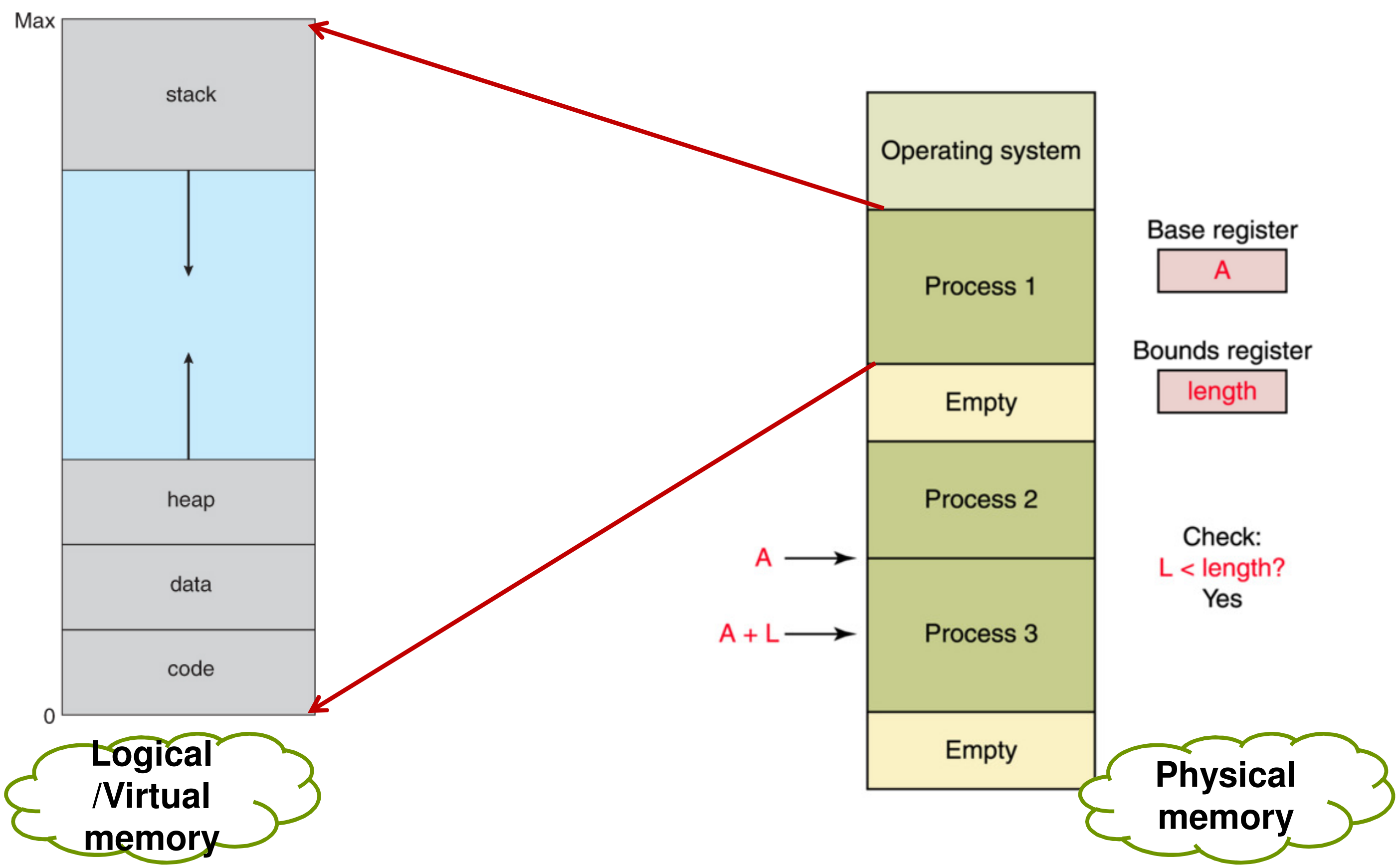
- On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes "argc", "argv" on the stack
  - sets the CPU registers properly & calls "_start()"

- Program starts running at _start()

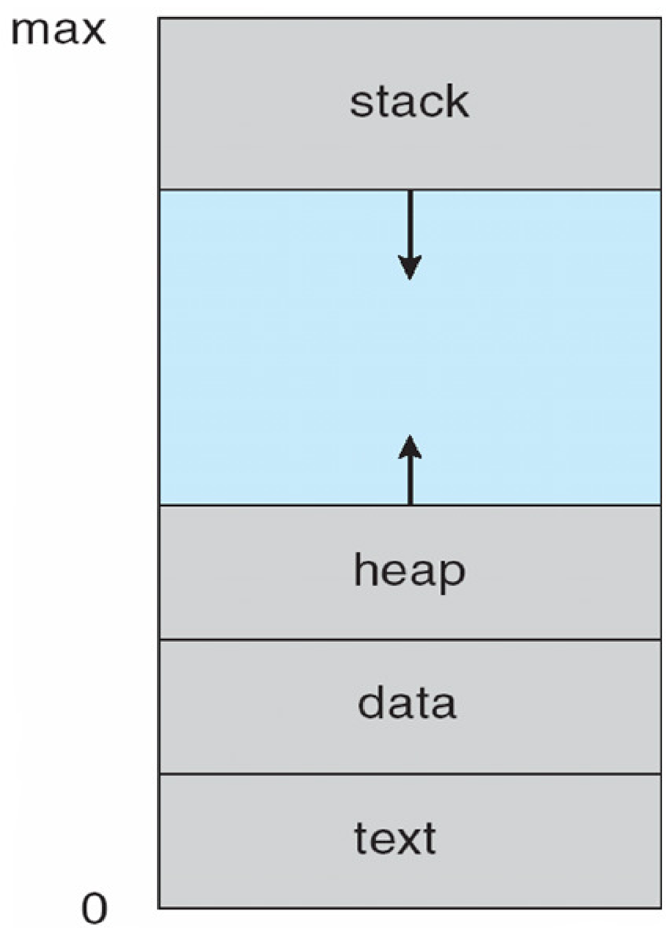
```
_start(args) {
    initialize_java();
    ret = main(args);
    exit(ret)
}
```

  we say "process" is now running, and no longer think of "program"

- When main() returns, OS calls "exit()" which destroys the process and returns all resources


Max
stack
heap
data
code
0
Logical /Virtual memory
Operating system
Process 1
Empty
Process 2
A
A + L
Process 3
Empty
Base register
A
Bounds register
length
Check:
L < length?
Yes
Physical memory

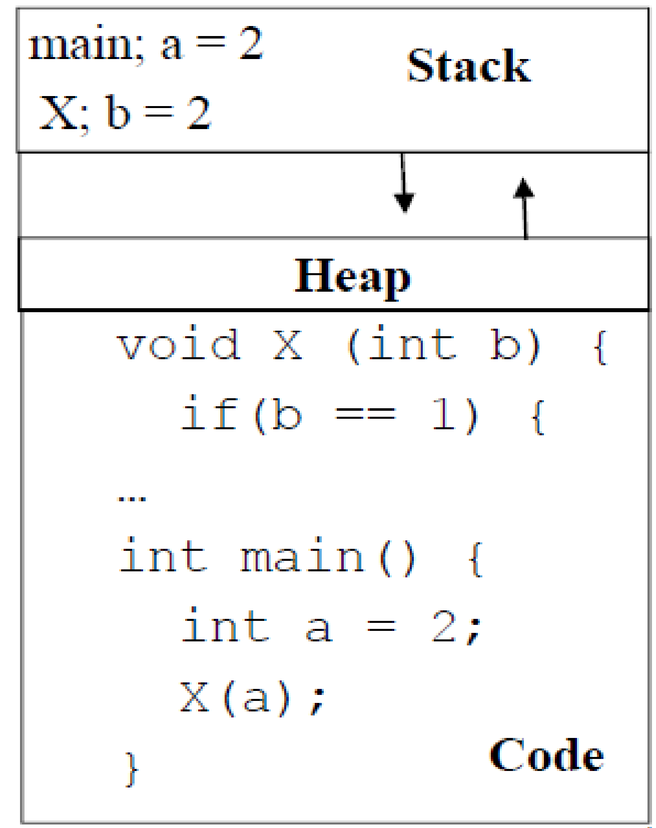
Process in Memory



- Program to process.
  - What you wrote

```
void X (int b) {
  if(b == 1) {
...
int main() {
  int a = 2;
  X(a);
}
```

- What is in memory.



- What must the OS track for a process?

Process Concept @ OS

- Textbook uses the terms job and process almost interchangeably.

A process is - Execution of an individual program.

Each time a process is created,

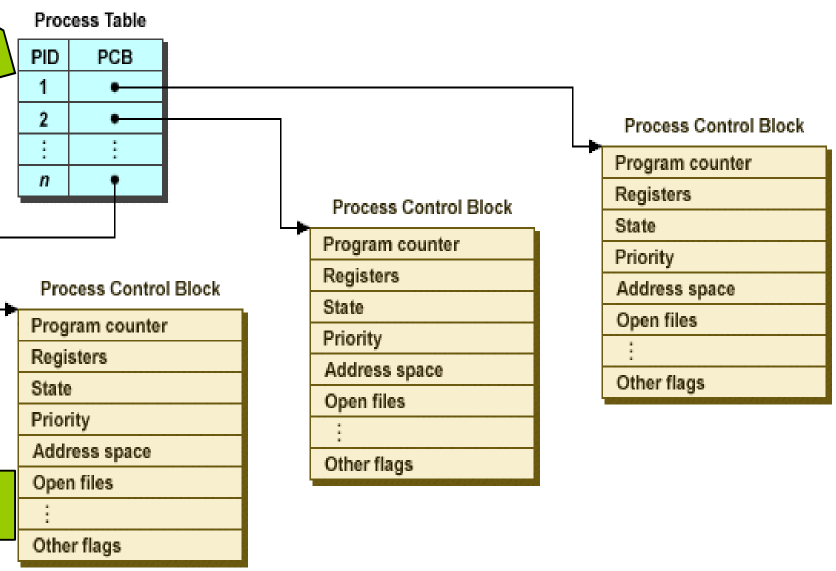
OS must create a complete independent address space (base, limit)
(i.e., processes do not share their heap or stack data)

RAM
P1
P2
P3

Represents by a Data Structure to OS
Called Process Control Block- PCB

Process Control Block (PCB)

- OS maintains a **process table** to keep track of the active processes



- Information for each process:

- Program counter
- Program id, user id, group id
- Program status word
- CPU register values
- CPU Scheduling-process priority, pointer to scheduling queue
- Memory maps-base/limit register, page table, segment table
- Stack pointer
- I/O status Information-allocated I/O devices, list of Open files
- Accounting information, etc.-amount of CPU & real time used, time limits, account numbers, job/process number

Stay in kernel (Main Memory)

System Call requires to upgrade info???

Structure of Process Images in Virtual Memory

William Stallings, Operating systems, 7th Edition, **Chapter 3, Table 3.5**

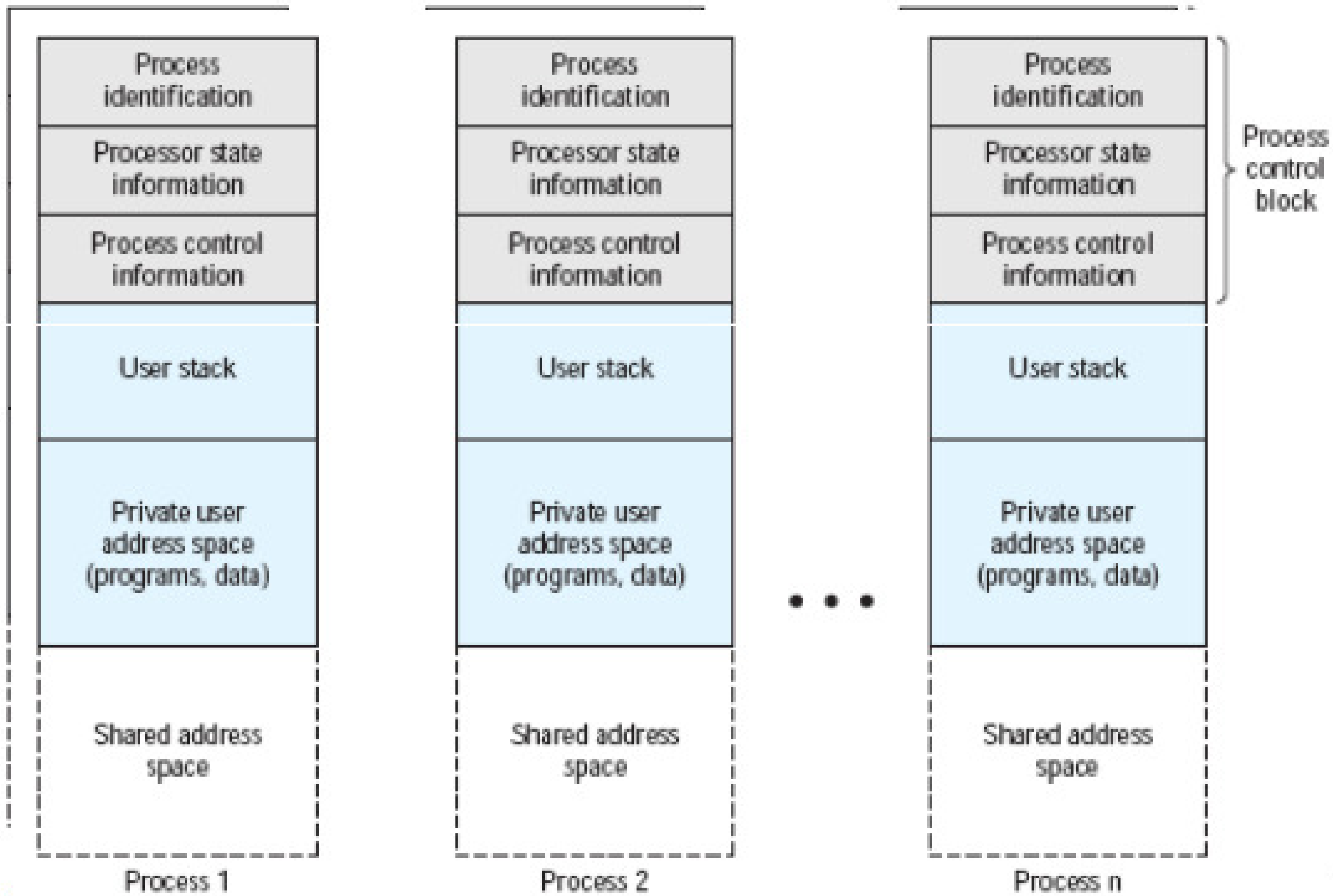


Figure 3.13 User Processes in Virtual Memory

PCB Information [Stalling Book Table 3.6,3.7]

PCB

Process Identification

-Process id
- Parent id
- User id

User visible resisters
8 to 32

Process State Information

Contents of process registers

PSW-
Control & status registers (PC, Arithmetic Logic result, sign, zero, overflow)

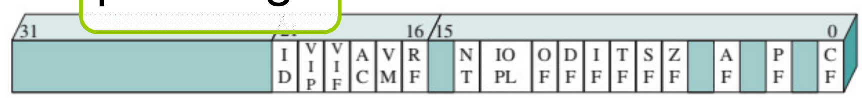
Stack pointer (sp)

Process Control

Scheduling and state information

Data structure

Process priviledge



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag

DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Process Control Information

Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring

A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent–child (creator–created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management

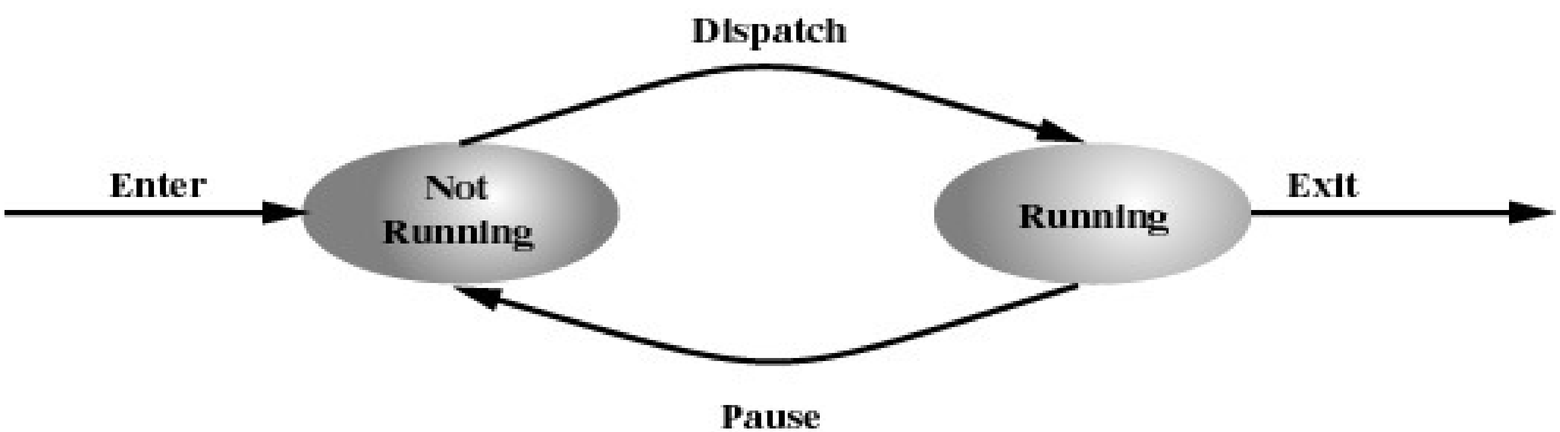
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization

Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

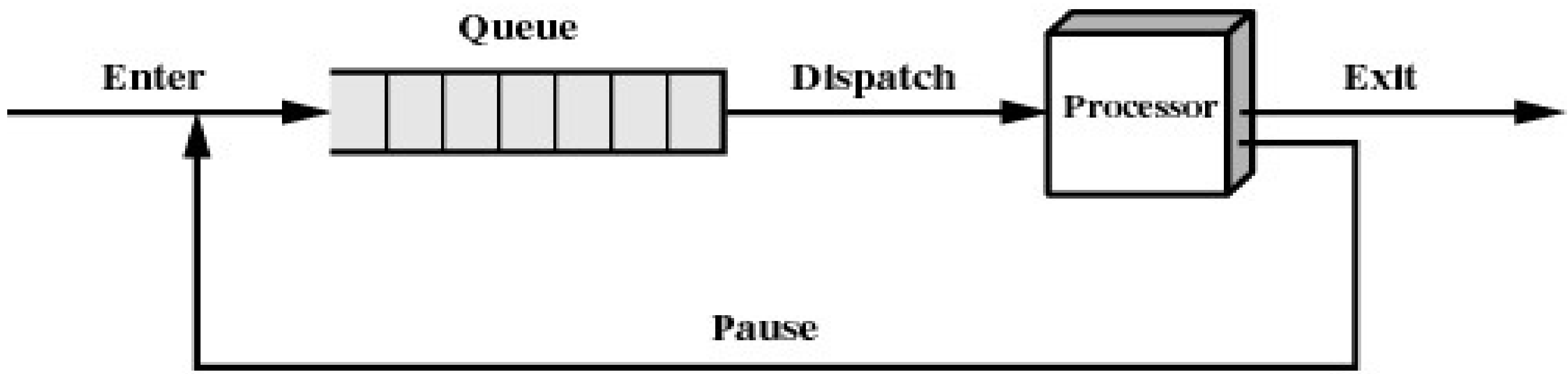
Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

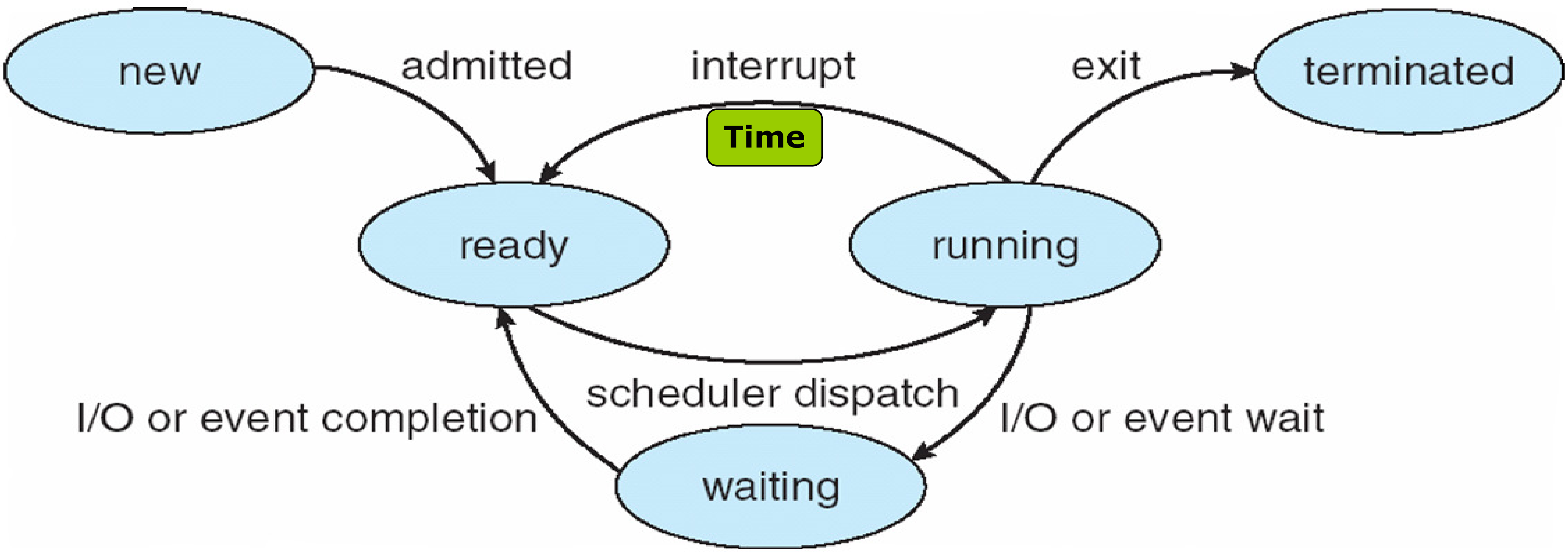
Not-Running Process in a Queue



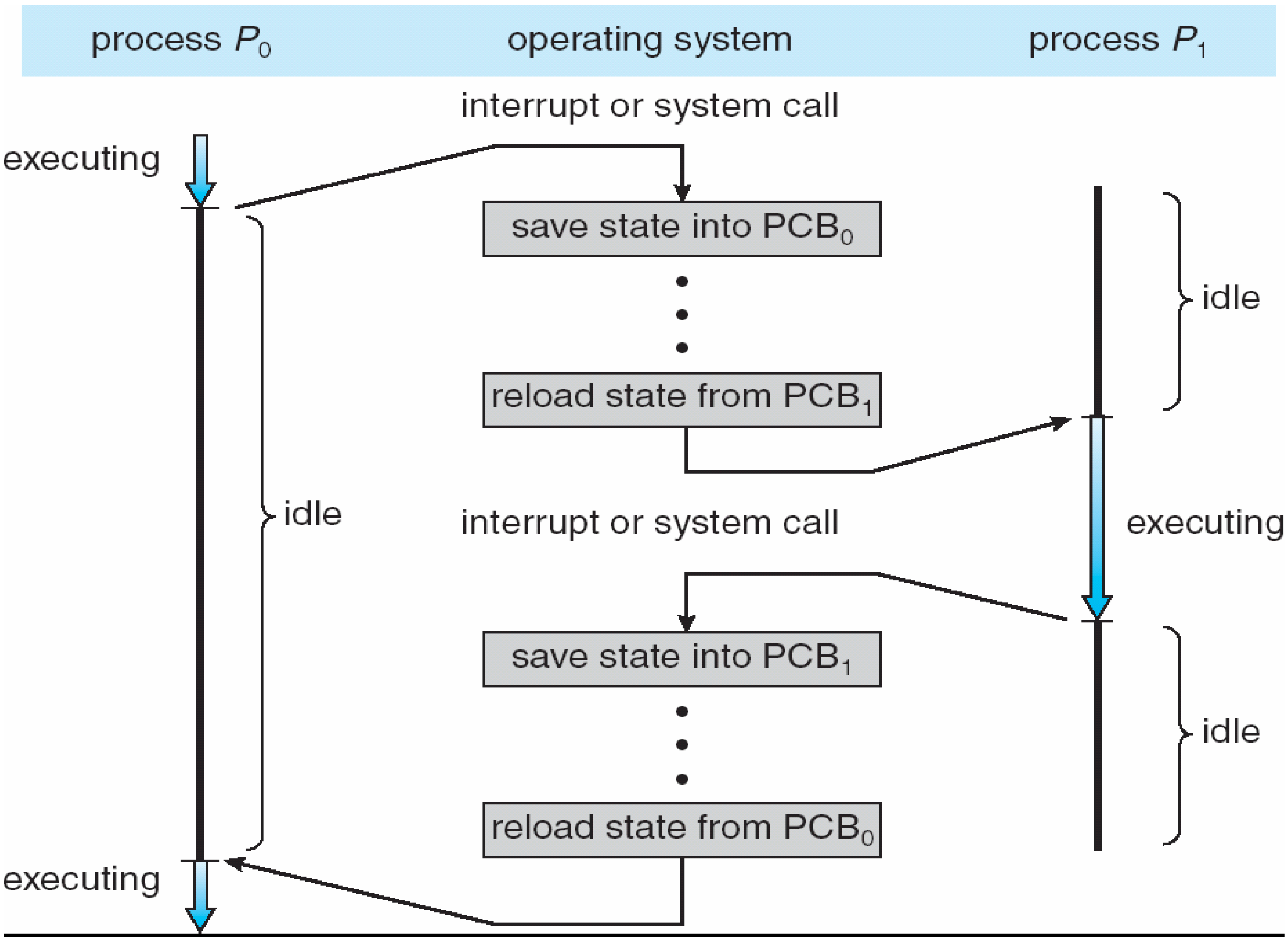
(b) Queuing diagram

Five States Process Model

- As a process executes, it changes *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

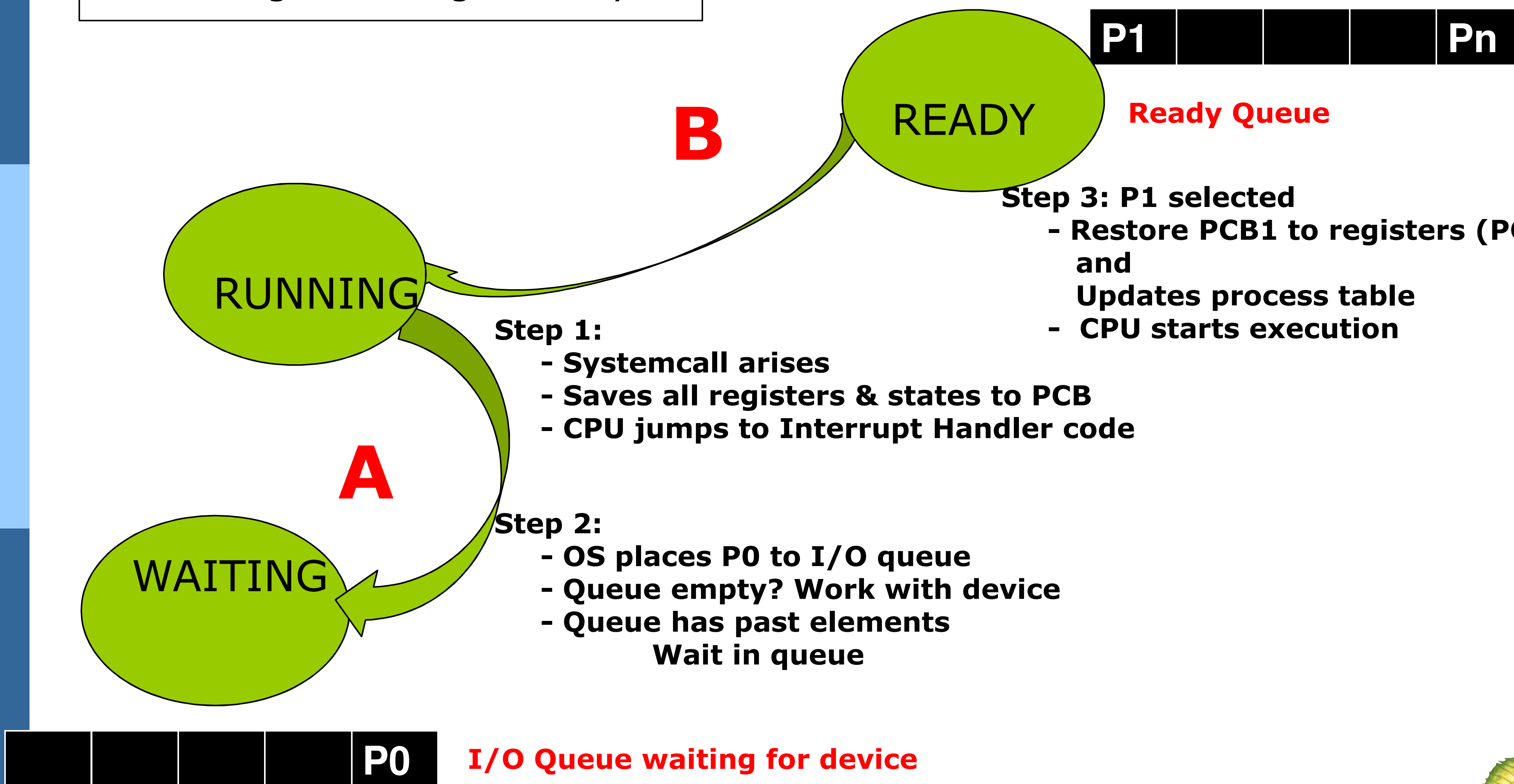


CPU Switch From Process to Process



Context Switch

1. Running->Waiting->Ready



Step 1:
- Systemcall arises
- Saves all registers & states to PCB
- CPU jumps to Interrupt Handler code

Step 2:
- OS places P0 to I/O queue
- Queue empty? Work with device
- Queue has past elements
  Wait in queue

Step 3: P1 selected
- Restore PCB1 to registers (PC, SP) and
  Updates process table
- CPU starts execution

Context Switch

2. Waiting->Ready-> Running

RUNNING

B

Step 3:
- **CPU saves running process Registers to PCB**
- **New process is selected from Ready Queue**
- **New process PCB is transformed to register**
- **CPU starts execution**

Ready Queue

P1				Pn

READY

Step 2:
- **Move the waiting process into ready queue (P0)**

A

WAITING

Step 1:
- **Device finishes the job (P0)**
- **Create interrupt to CPU, OS executes interrupt handler**
- **Device is assigned next job**

				P0

I/O Queue

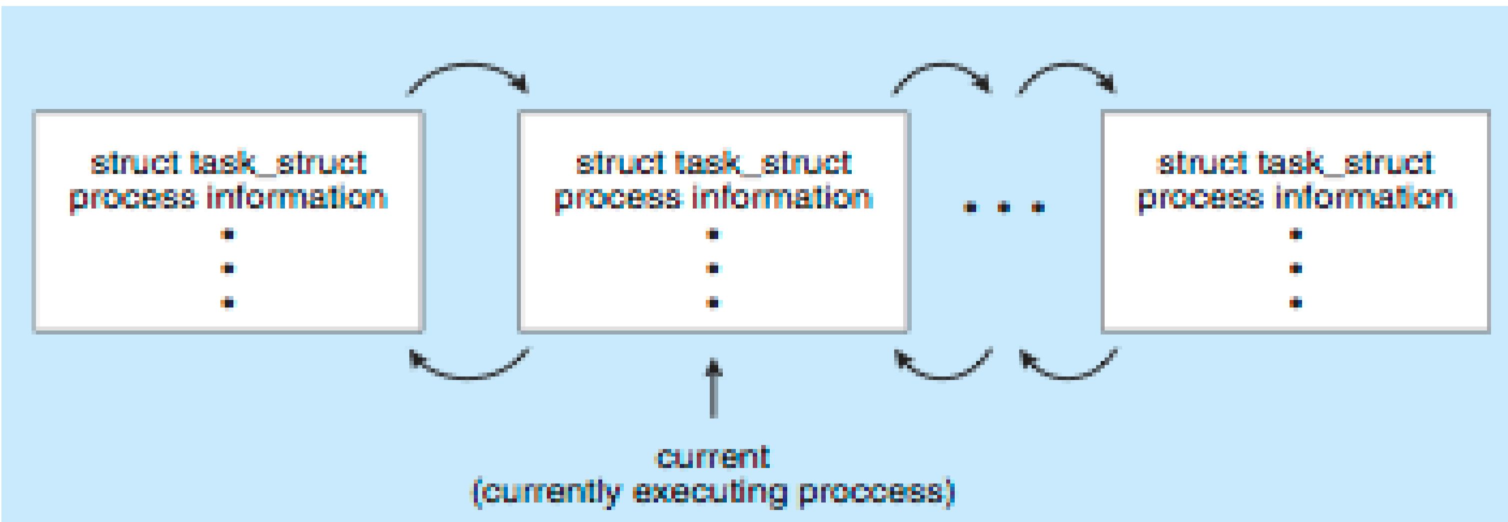
Process Scheduling

- AIM: Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects process among available for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system (HD-Pool)
  - **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

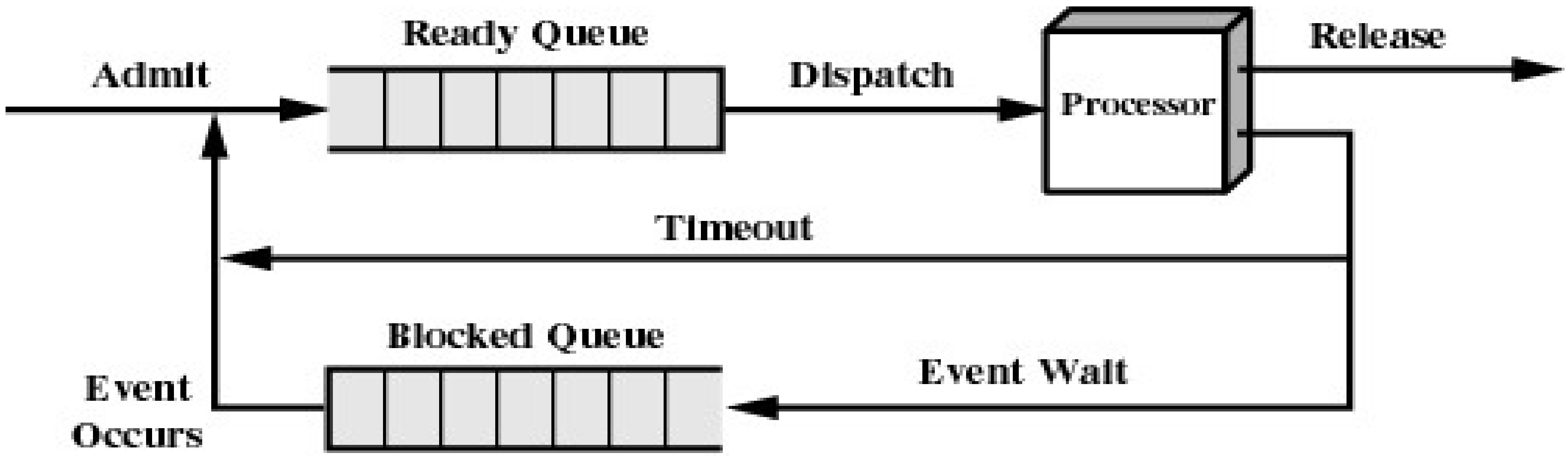
Process Representation in Linux

- 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 pro */
  ```

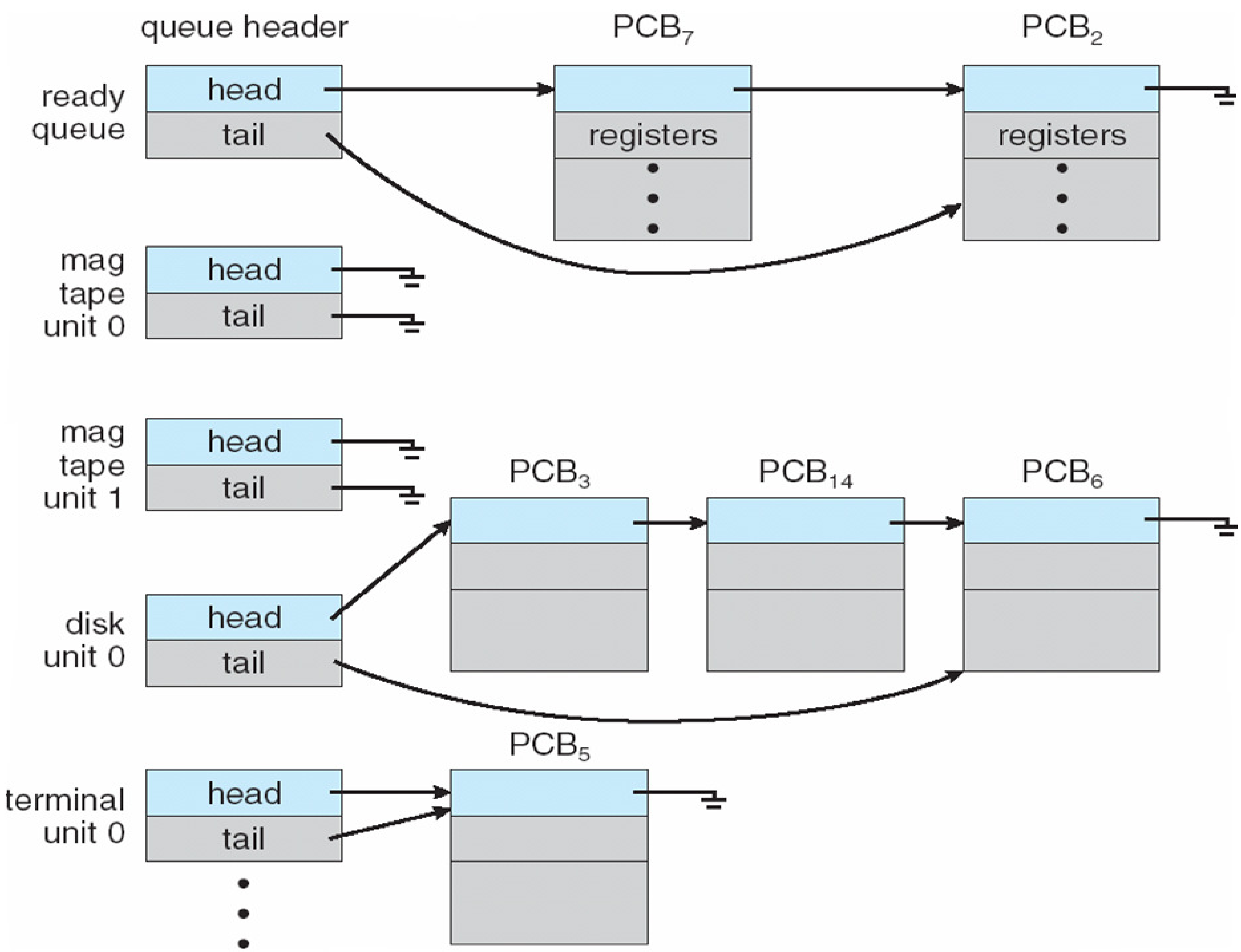

  ```

Using Two Queues

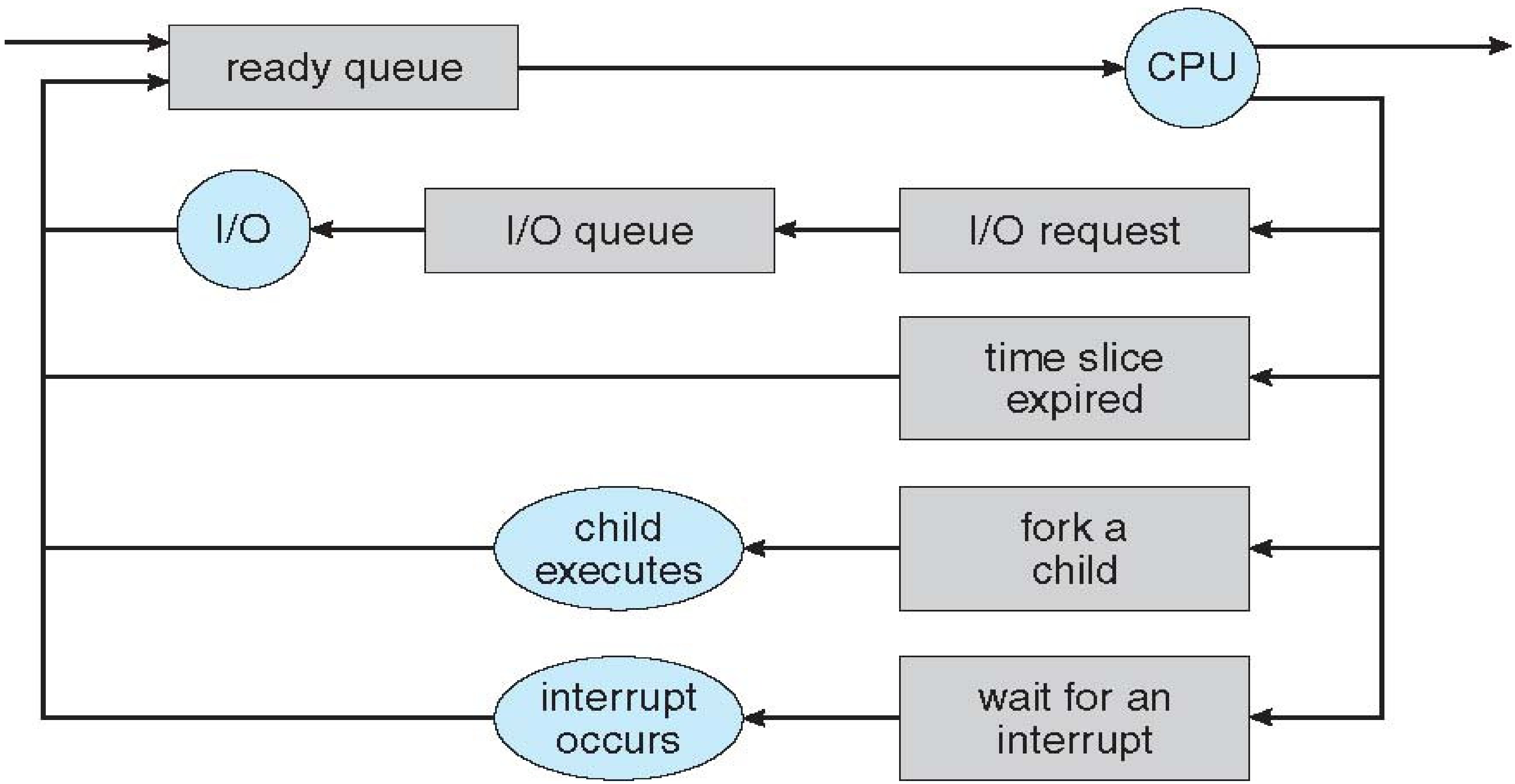


(a) Single blocked queue

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

Long-term scheduler/Job scheduler

- which processes should be brought into the ready queue
- invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- controls the *degree of multiprogramming*

Short-term scheduler/CPU scheduler

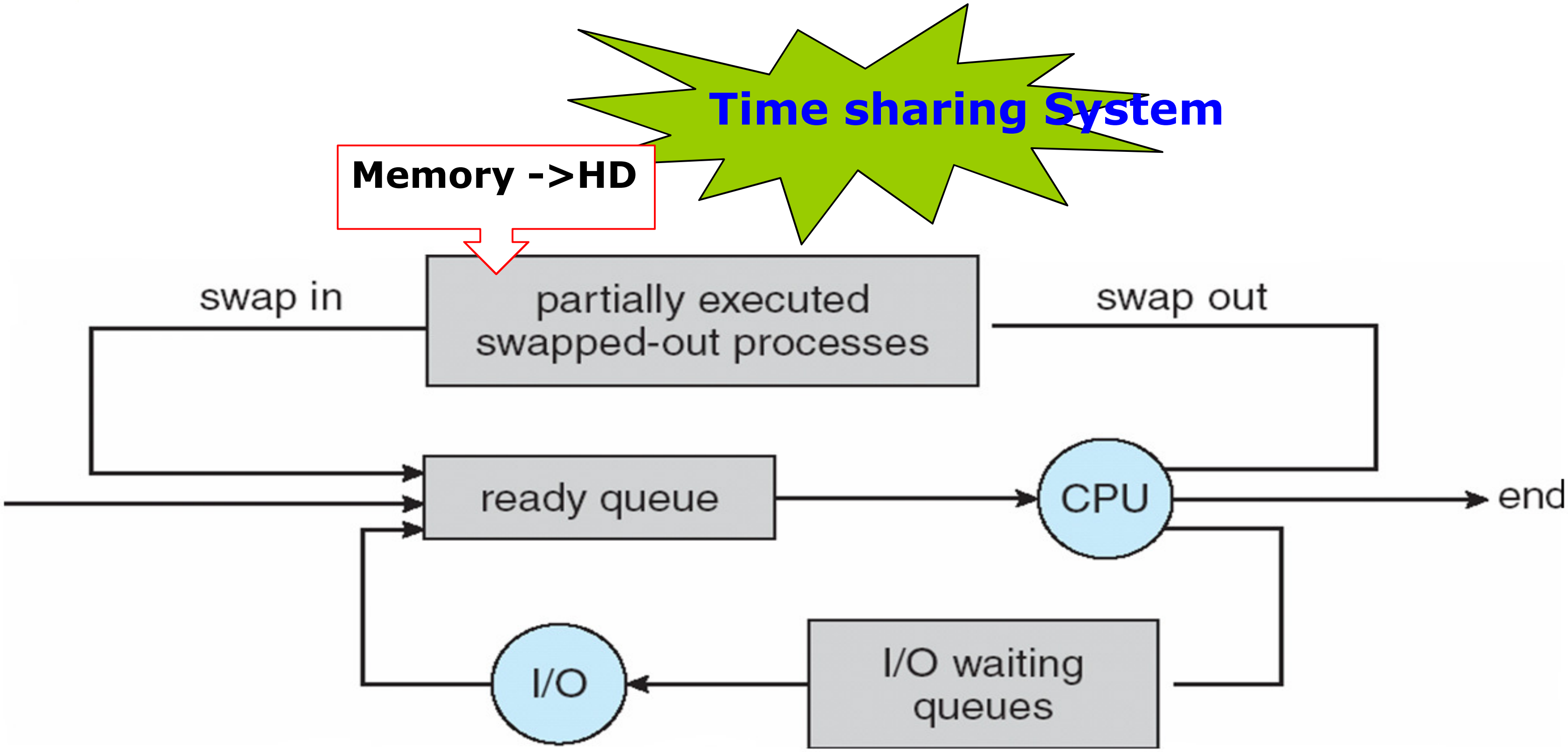
- which process should be executed next and allocates CPU
- invoked very frequently (milliseconds) ⇒ (must be fast)

Balanced

I/O-bound process

CPU-bound process

Addition of Medium Term Scheduling



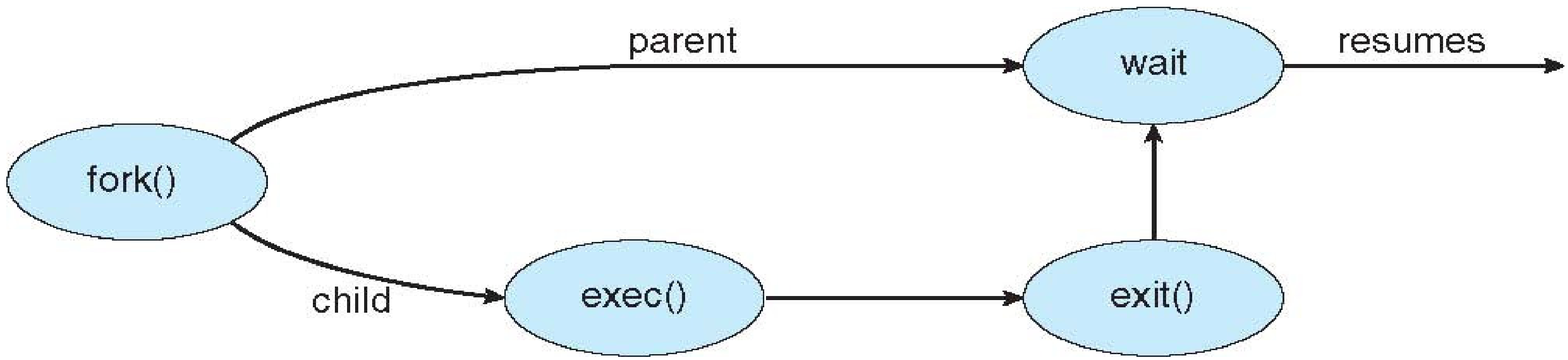
Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a tree of processes.
- **Process identifier** (**pid**): process identified and managed.
- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- **Address space**
  - Child duplicate of parent
  - Child has a program loaded into it
- **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
    - As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.

Process Creation



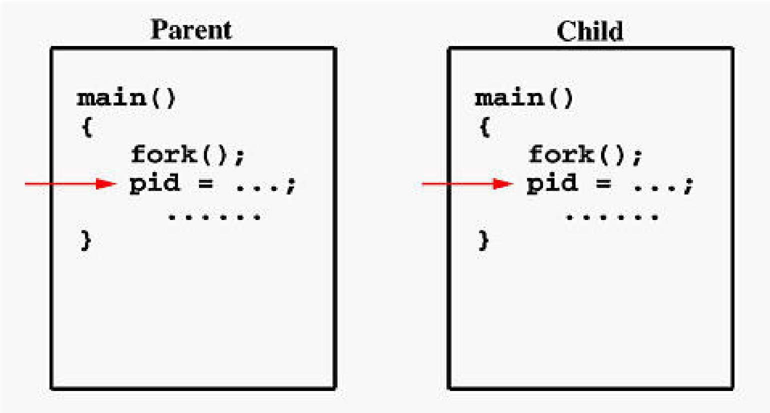
The FORK() System Call

Processes in UNIX are created with the fork() system call.

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

void main(void)
{
    int pid;
    pid = fork();
    if (pid == -1) {
        printf("error in process creation\n")
        exit(1);
    }
    else if (pid == 0) child_code();
    else parent_code();
}
```

Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.



The FORK() System Call

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

#define PROCESS 10

void main(void)
{
    int pid, j;

    for (j=0; j < PROCCESS; j++) {
        pid = fork();
        if (pid == -1) {
            printf("error in creation of process %d\n", j);
            exit(1);
        }
        else if (pid == 0)  child_code(j);
    }
    for (j = 0; j < PROCESS; j++) wait(0);
}
```

```
void child_code(int id)
{
        pid_t myid, pid;
        myid = getpid();
        pid = getppid();

        printf("My pid is %d and my parent's id is %d",
                myid, pid);

        printf("My virtual id is %d\n", id);
        exit(0);
}
```

If we are interested to wait for a particular child, we can use instead of wait(), **waitpid**() (see man pages for more information).

c

C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
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;
}
```

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit system call**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system

No Wait: Zombie process
Specially handle by OS

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is terminated
    - some operating systems do not allow child to continue if its parent terminates
      - All children then also terminated – called **cascading termination**

- Lecture Materials
  - Galvin 4.1-4.4

End of Lecture 4, 5