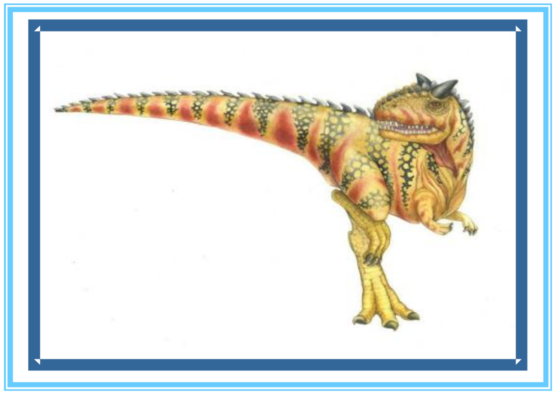
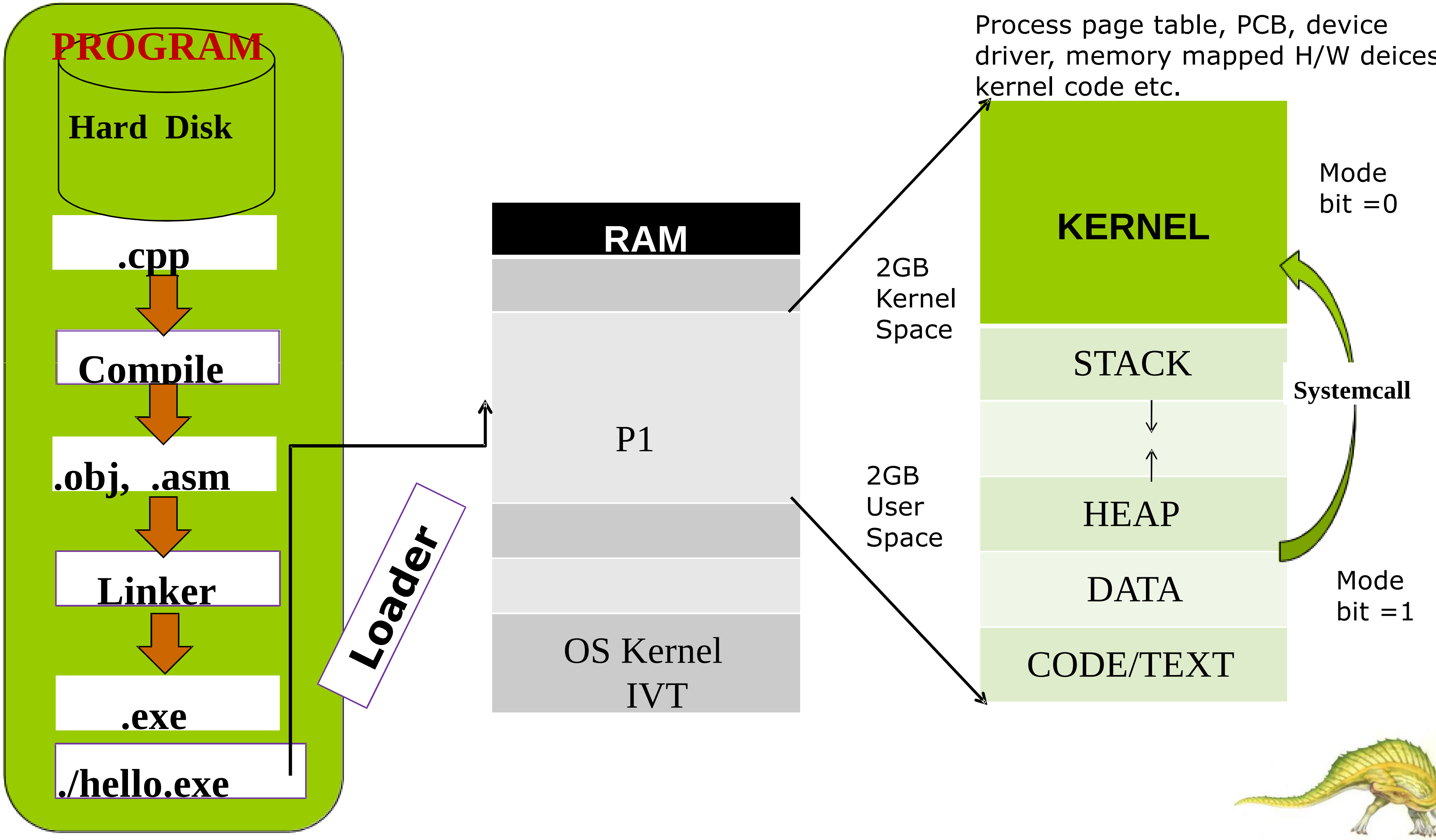
Lecture 4, 5
Process Management



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 external variables that are initialized.

BSS contains

global or static or external 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 (local variables)within functions scope.

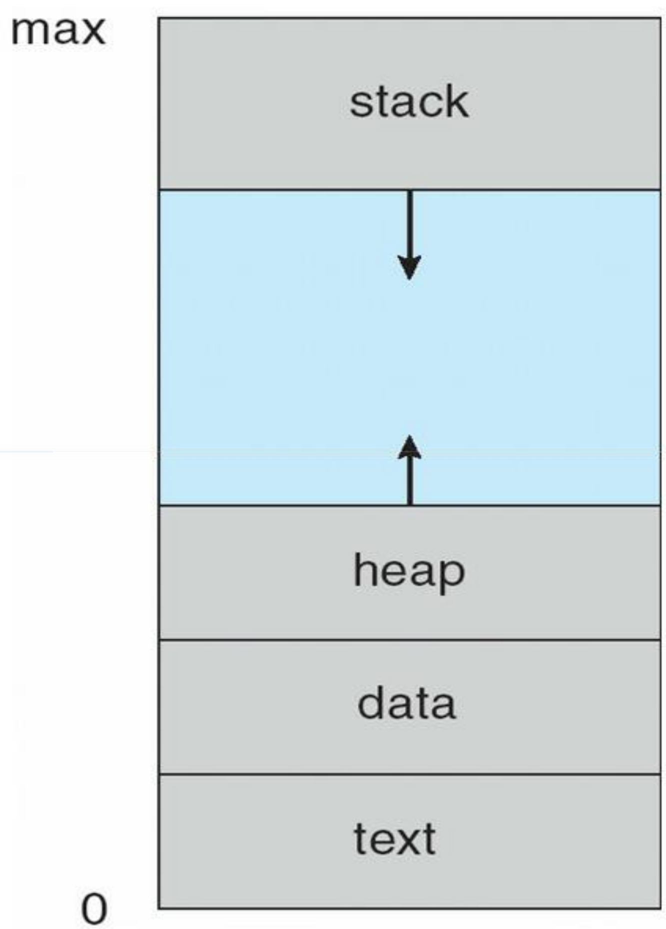
The Process-Executable Program

- 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

Process in Memory

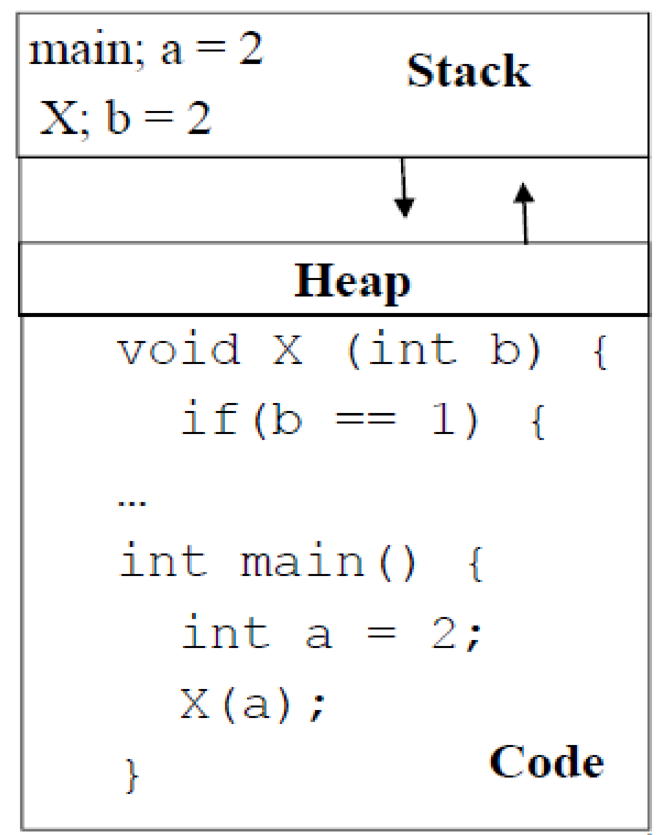


- Program to process.
- What you wrote

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

- What must the OS track for a process?

- What is in memory.



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,

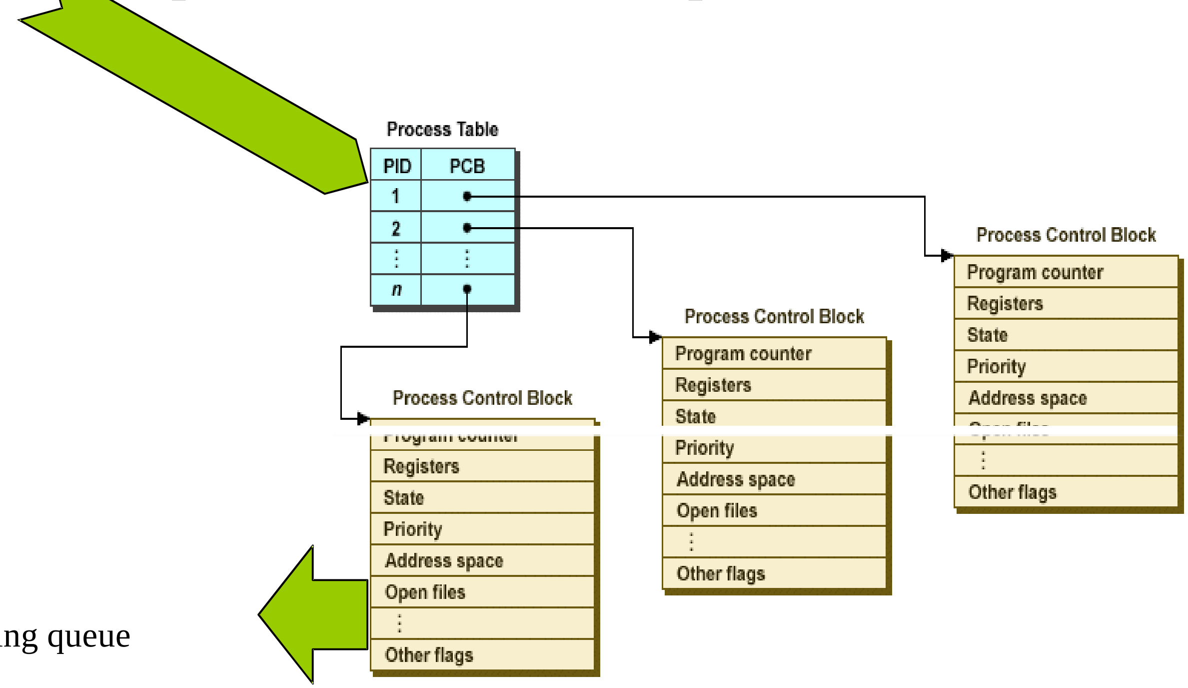
OS must create a complete independent address space (base, limit)
(i.e., processes do not share their heap or stack data)

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

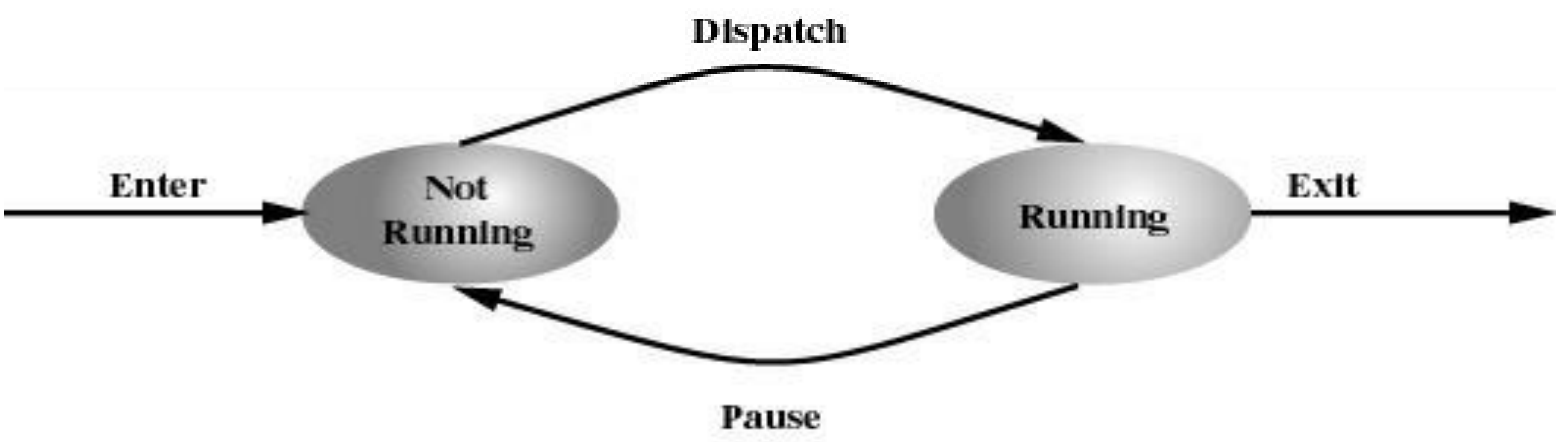
Accounting information, etc.-amount of CPU & real time used, time limits, account numbers, job/process number

Stay in kernel
(Main Memory)

Two-State Process Model

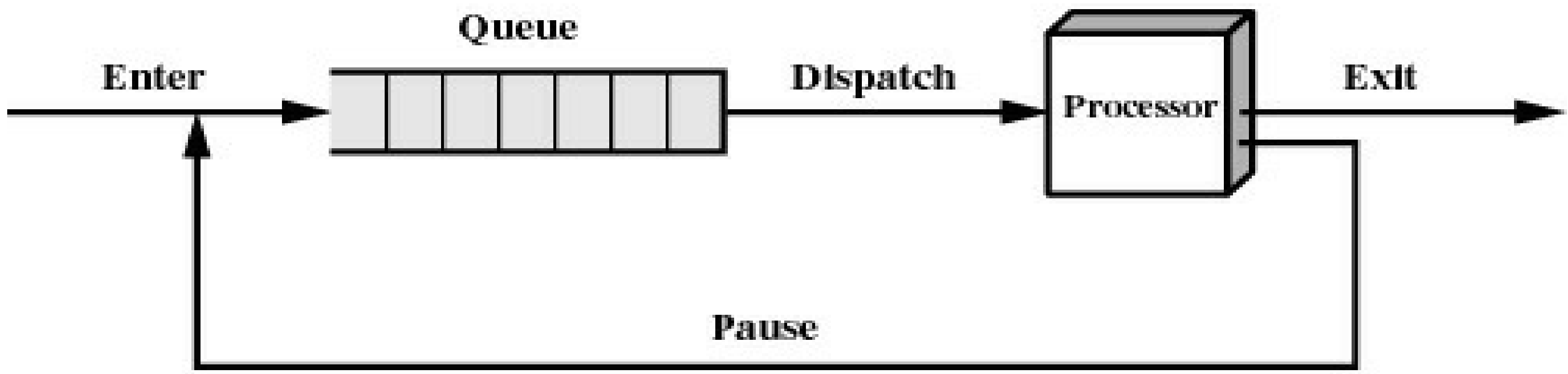
Process may be in one of two states

- Running
- Not-running



(a) State transition diagram

Not-Running Process in a Queue

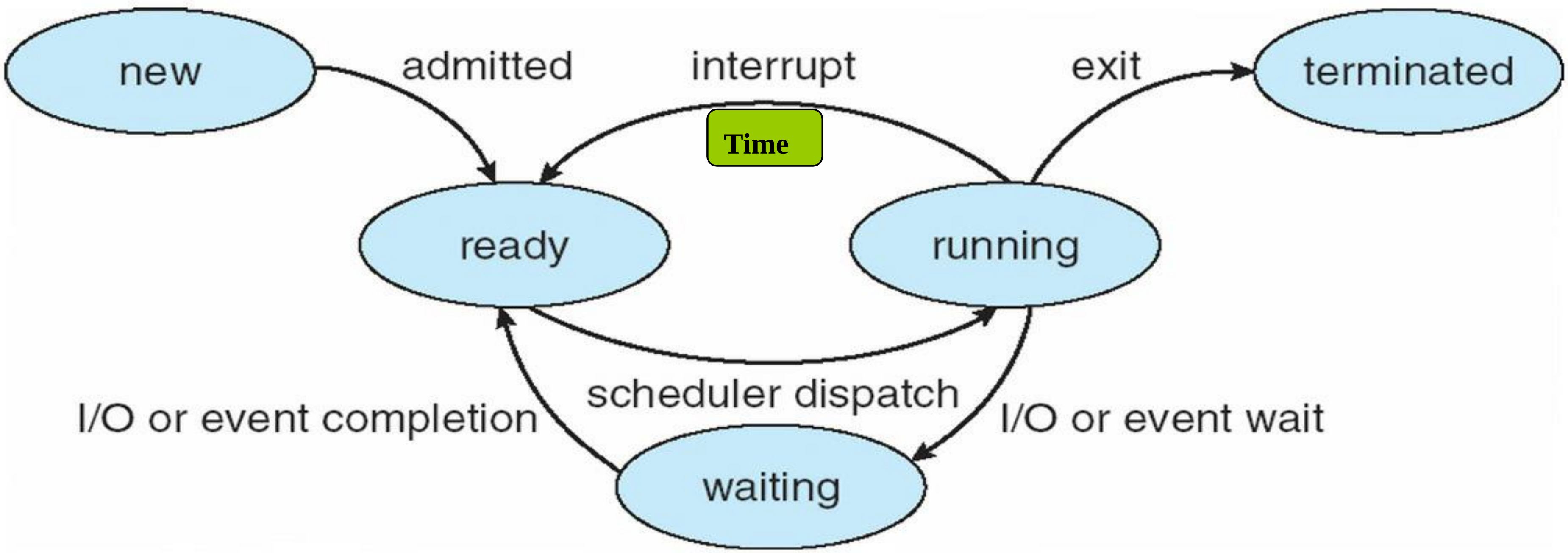


(b) Queuing diagram

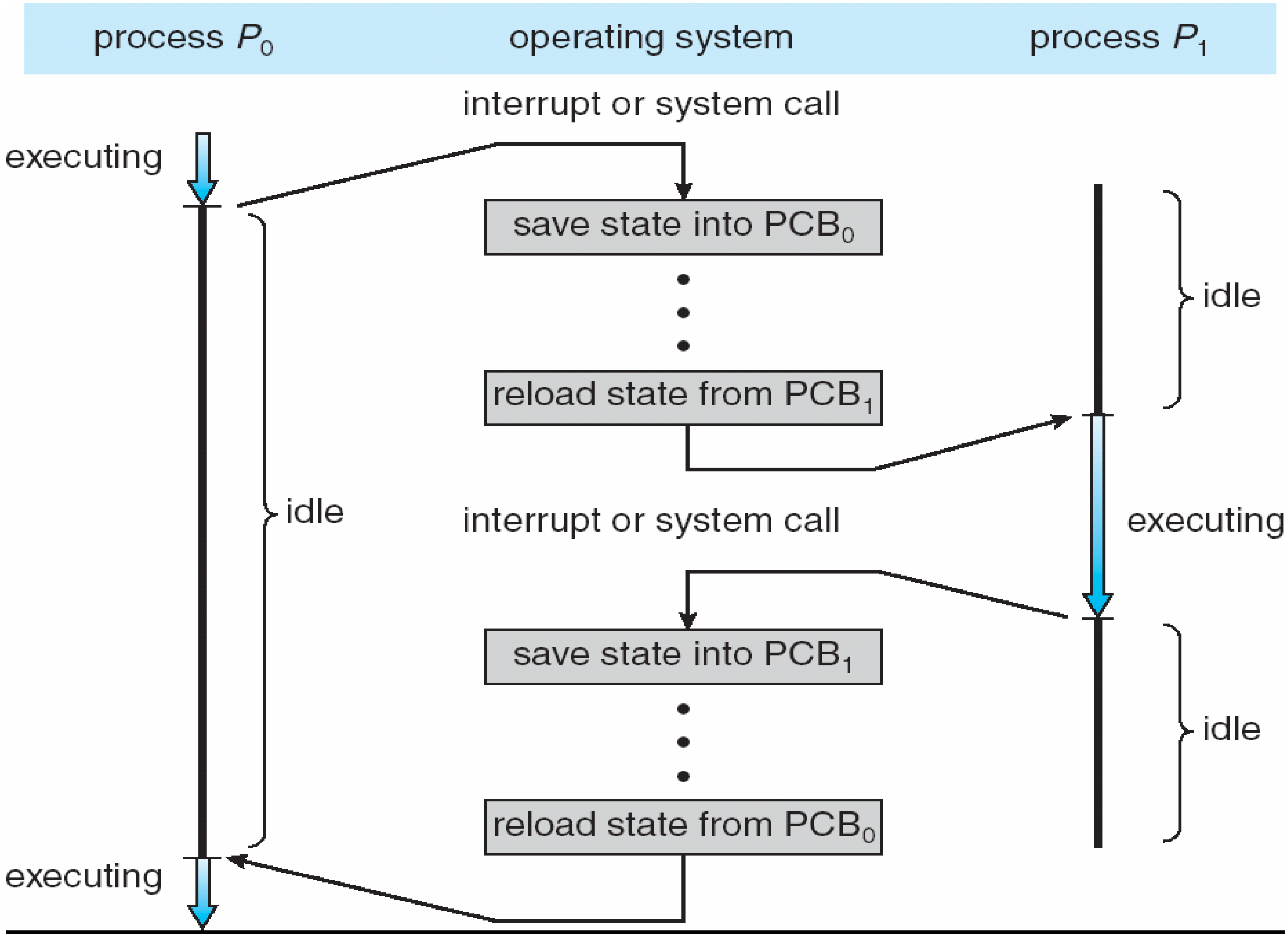
Five States Process Model

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

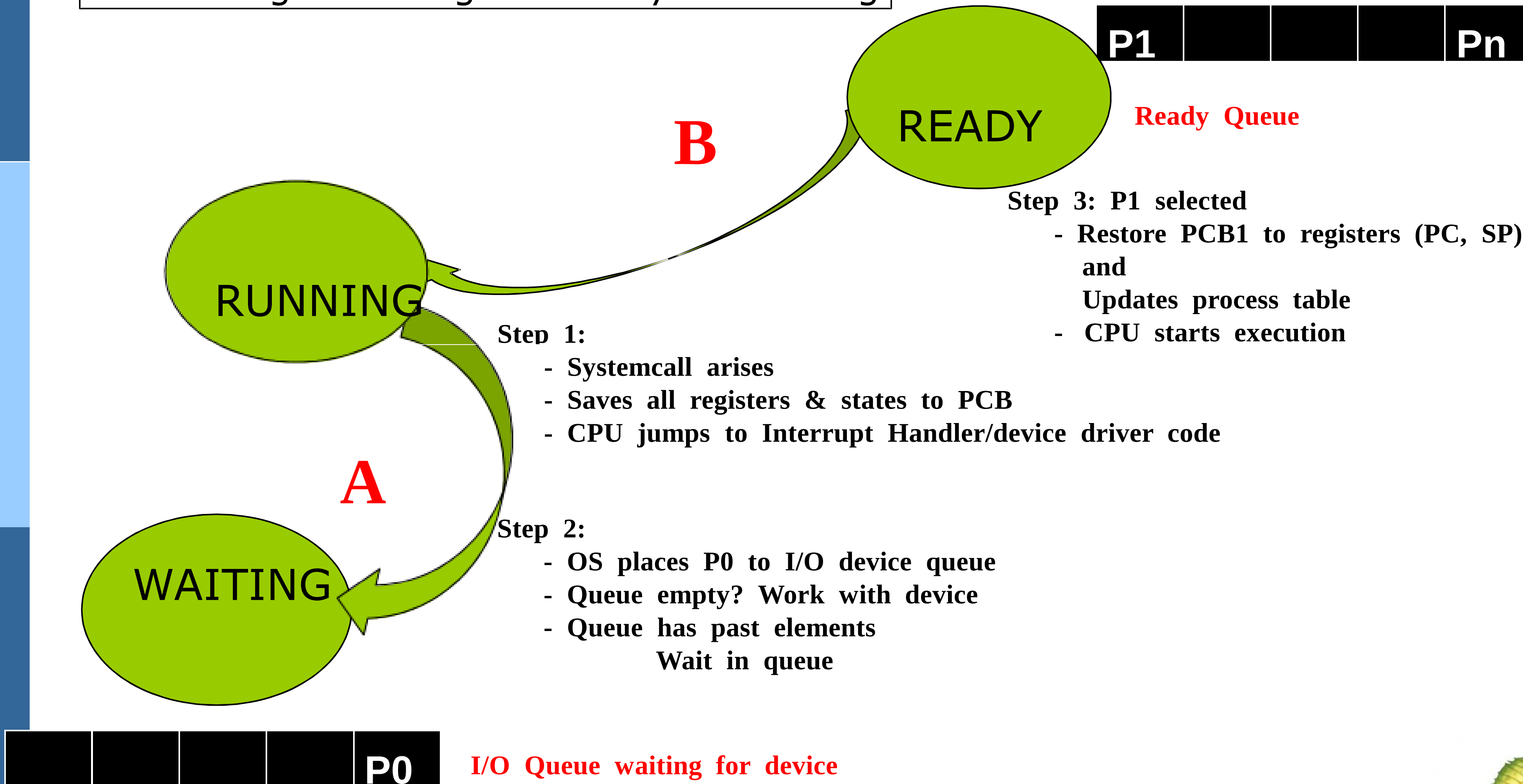


CPU Switch From Process to Process



Context Switch

1. Running->Waiting && Ready-> Running



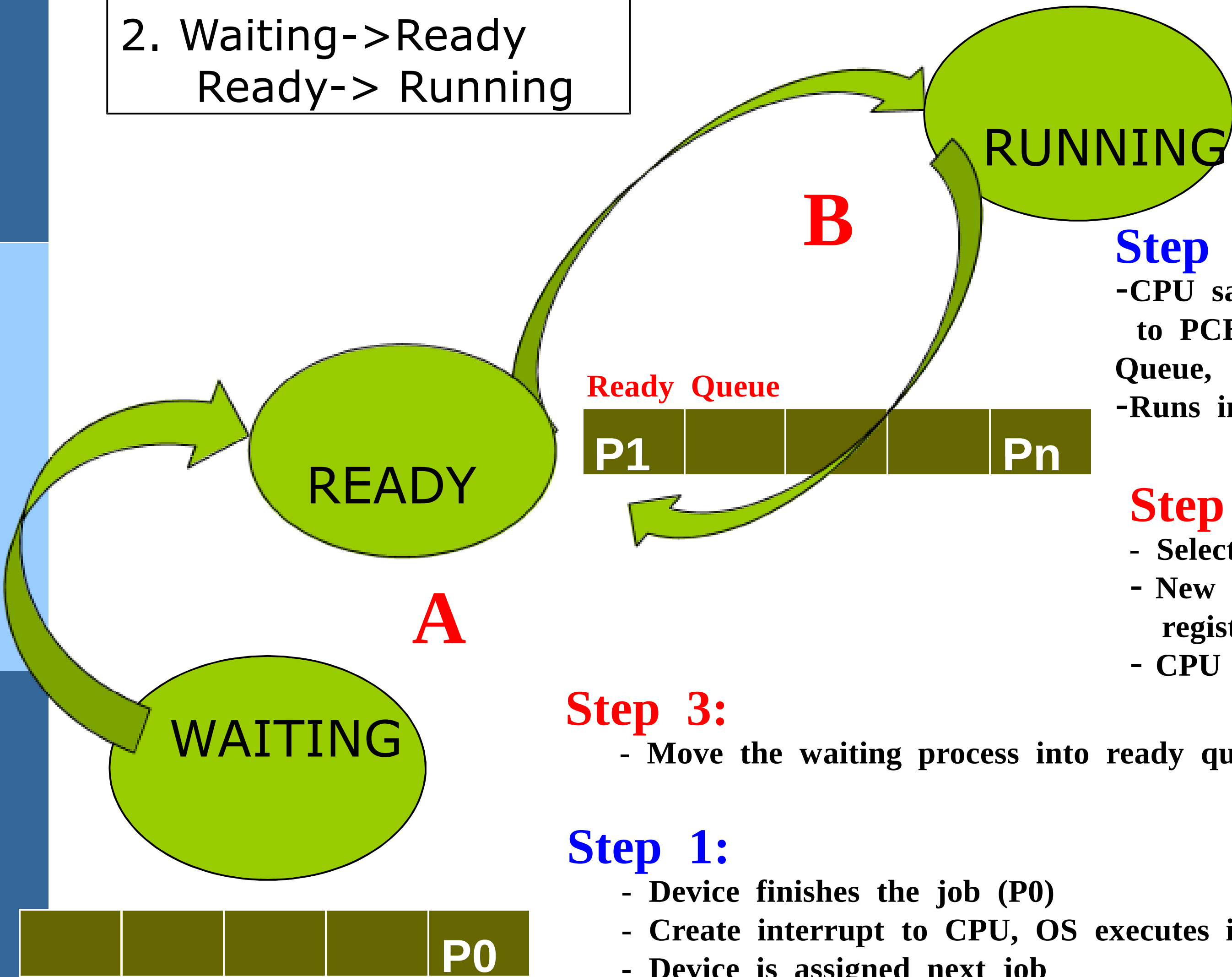
Step 3: P1 selected
- Restore PCB1 to registers (PC, SP) and
  Updates process table
- CPU starts execution

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

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

Context Switch

2. Waiting->Ready
   Ready-> Running



I/O Queue @ Device status table

Step 2:
-CPU saves running process Registers to PCB(and put the process back to Ready Queue, if required)
-Runs interrupt handler routine

Step 4
- Select the new process from Ready Queue
- New process PCB is transformed to register
- CPU starts execution

Step 3:
- Move the waiting process into ready queue (P0)

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

Process Scheduling

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

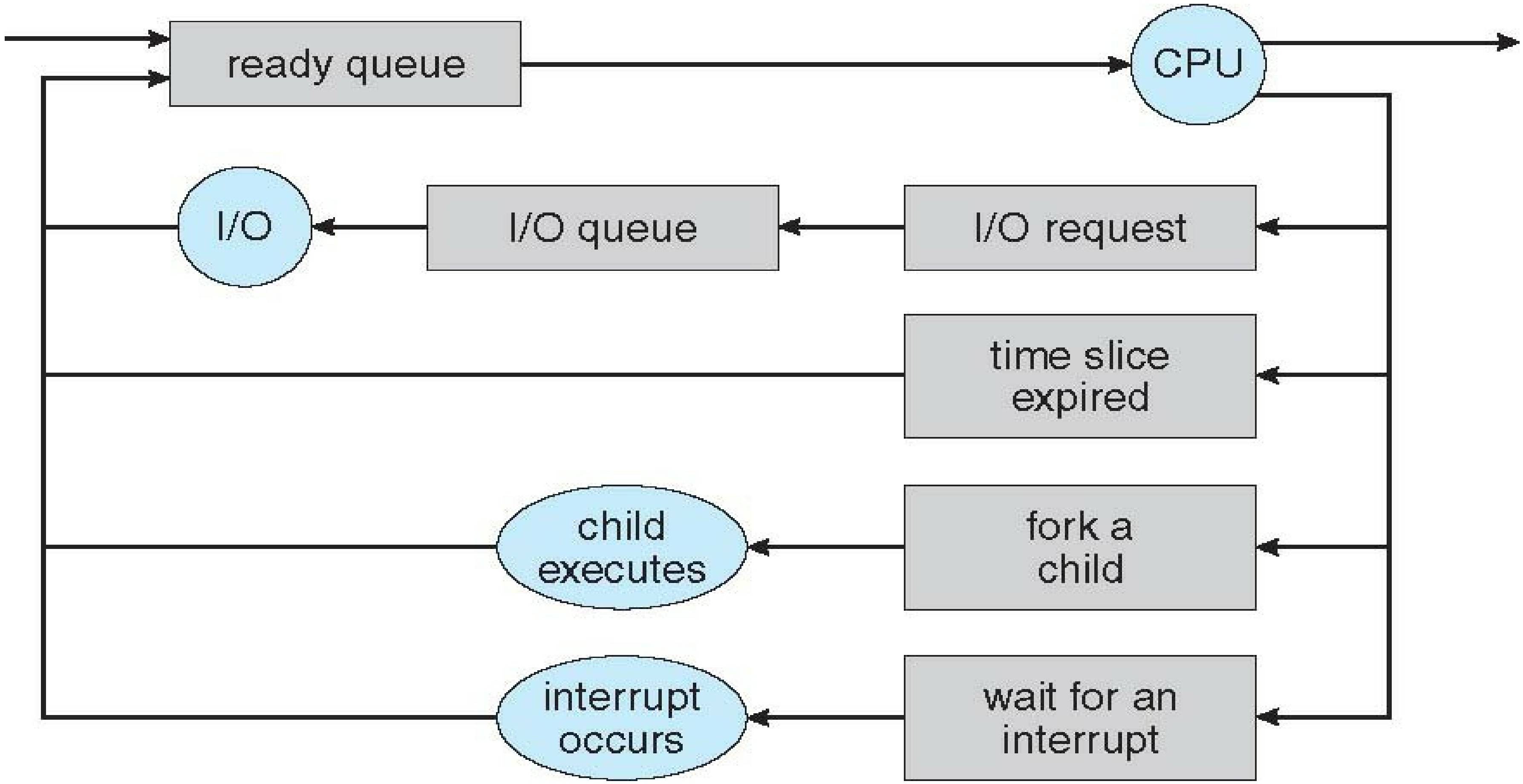
Process scheduler (is a process) selects process among available for next execution on CPU

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

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

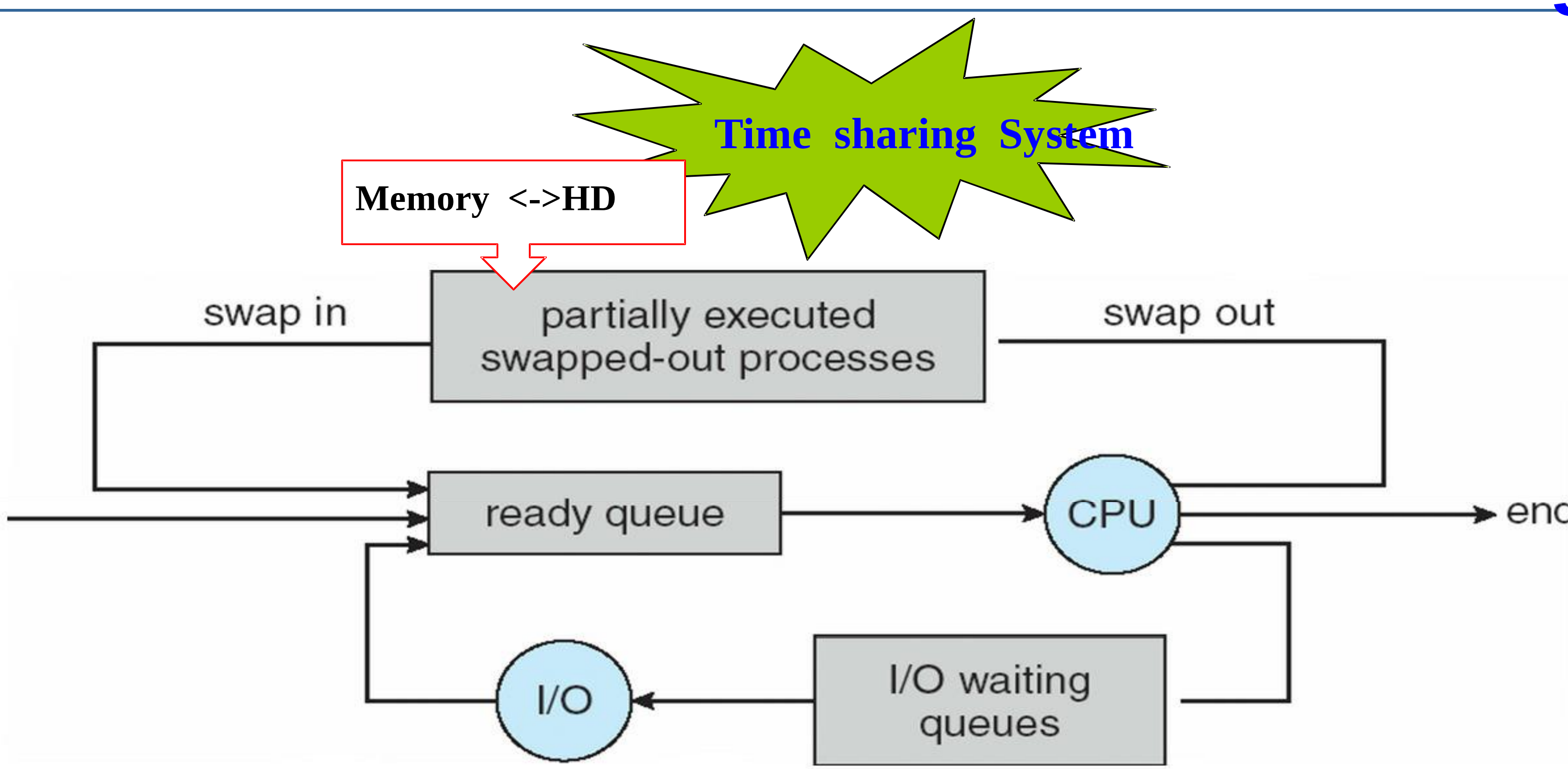
- which process should be executed next and allocates CPU
- invoked very frequently (milliseconds) ⇒ (must be fast)

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

Lecture Materials

Galvin 4.1-4.3

Galvin 13.4.1,13.4.2,13.4.3,13.4.4,13.5

End of Lecture 4, 5