Chapter 3
Process Description and Control

Part A: Sections **3.1–3.2**

Useful pre-knowledge for this lecture is: “***State Transition Diagrams***”, as already seen in **Chapter 17** of COS**151** Introduction to Computer Sc.

Overview of Ch.3

- **What is a process?**
  - Background
  - Processes and process control blocks
- **Process states**
  - Two-state process model
  - Creation and termination
  - Five-state model
  - Suspended processes
- **Process description**
  - Operating system control structures
  - Process control structures

- **Process control**
  - Modes of execution
  - Process creation
  - Process switching
- **Execution of the operating system**
  - Non-process kernel
  - Execution within user processes
  - Process-based operating system
- UNIX SVR4 **process management**
  - Process states
  - Process description
  - Process control

Process "Elements"

Two essential "elements" of a process are:

Program code (*Algorithm*)

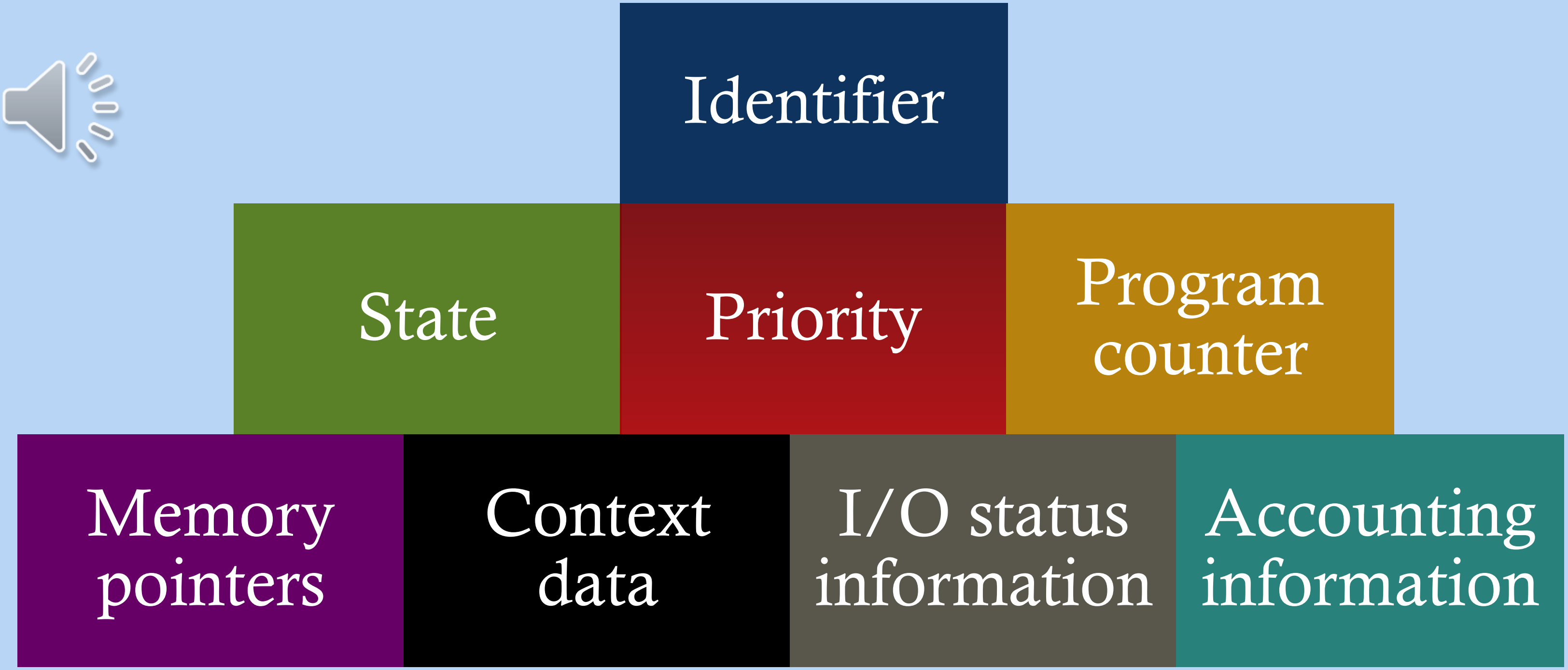
(may be shared with other processes that are "run" the same program, (*for example in parallel recursion*))

A set of data associated with that code

When the processor (CPU) begins to execute the program code, we refer to this "running" entity as *process*. However we can also speak of "sleeping processes" when the CPU is currently busy with another (different) process.

Further Process "Elements"

- While a program is running, its process can be uniquely characterized by a number of **"elements" in the OS's "logbook"**, including:



Identifier

State

Priority

Program counter

Memory pointers

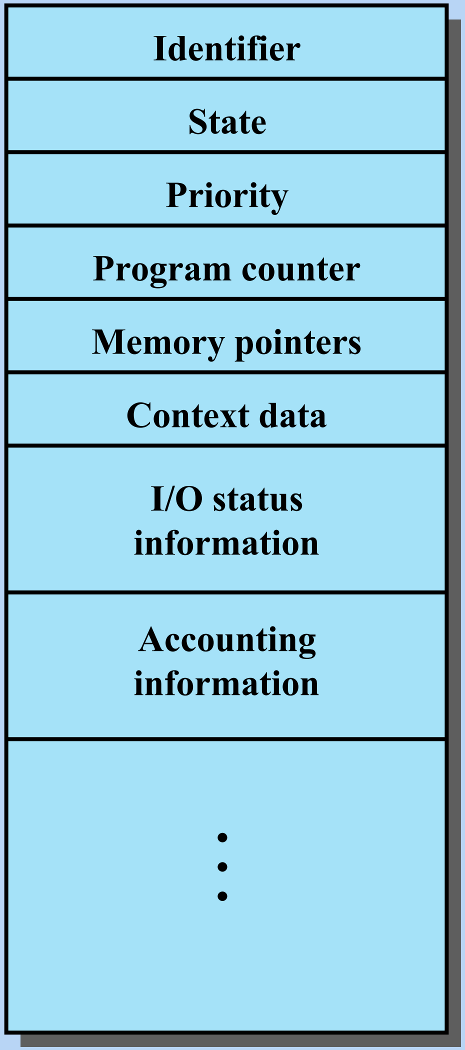
Context data

I/O status information

Accounting information

Process Control Block (PCB)

- **Data Structure in the “logbook”**
- **Contains the information “elements” about a process** which were shown on the previous slide
- Makes it possible to interrupt a running process, and later resume it, as if no interruption had occurred
- Created, stored (in memory) and managed by the operating system: The **“logbook” consists of *many* PCBs**!
- Key means that allows support for multiple processes to be “in the system” (***multi-programming***)



Identifier

State

Priority

Program counter

Memory pointers

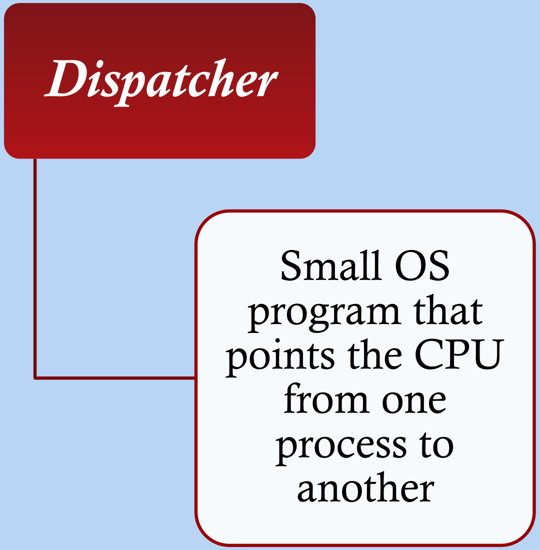
Context data

I/O status information

Accounting information

⋮

Figure 3.1 Simplified Process Control Block



Dispatcher

Small OS program that points the CPU from one process to another

To ***Computer Science*** students **only**:

For illustration see the program counter (**PC**) in **Figure 5.32** of the 5th Chapter of the **COS151 "Introduction" Book**

Process Execution

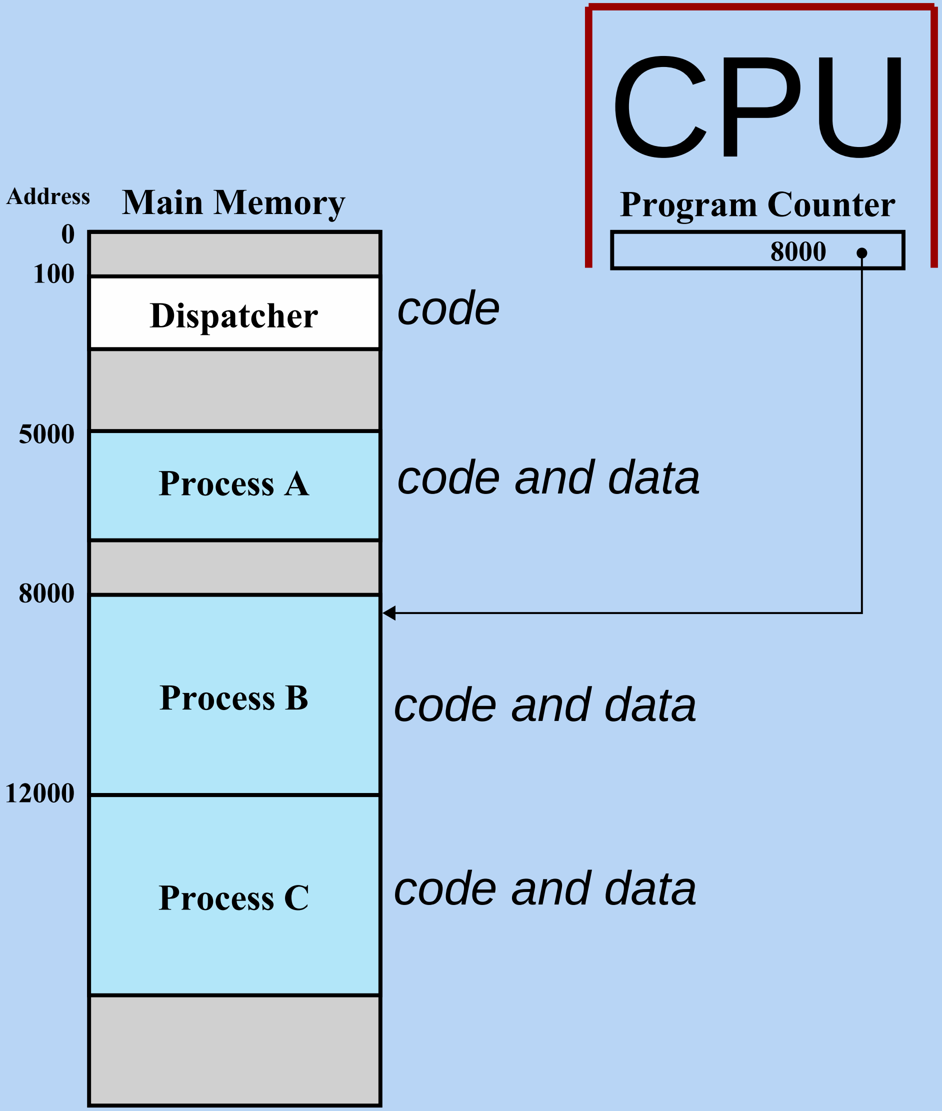
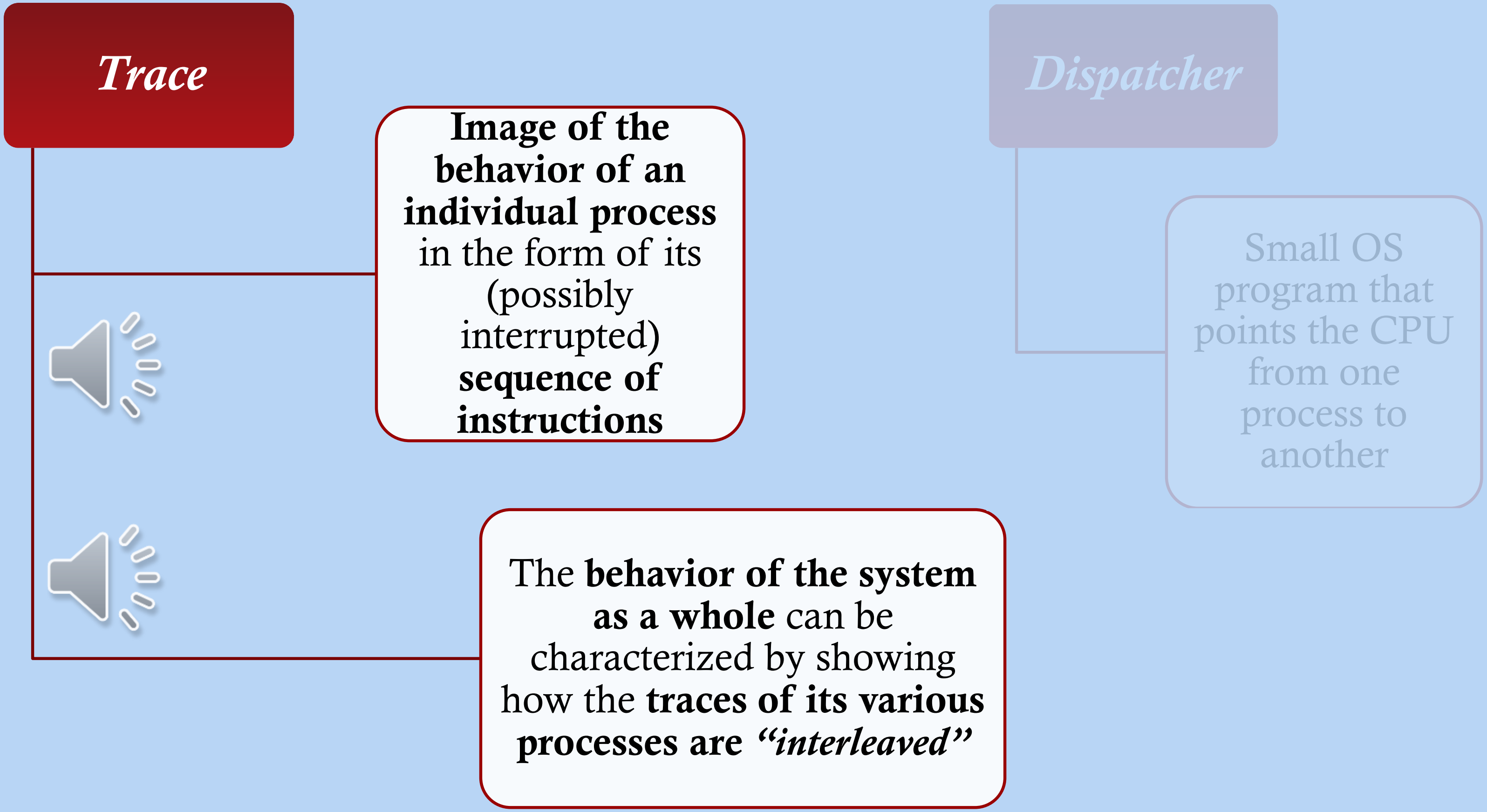


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13



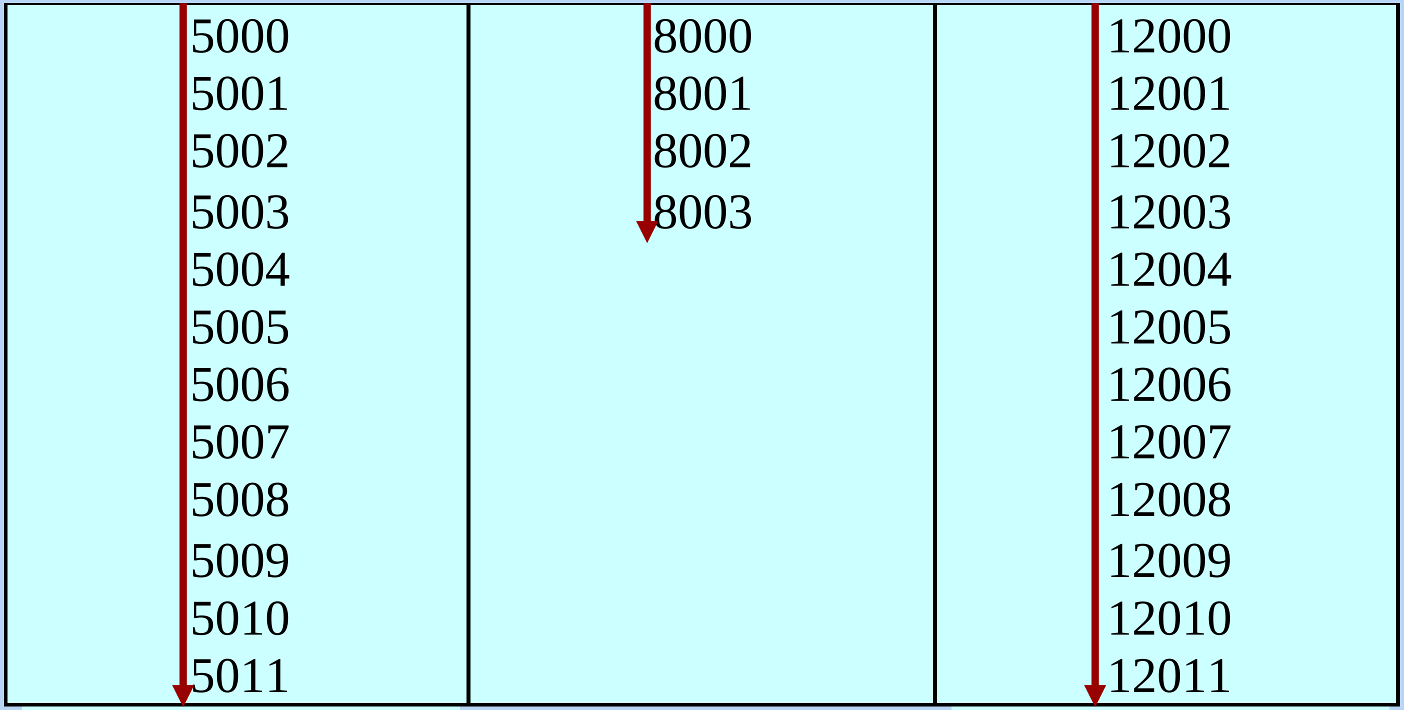
Trace

Image of the behavior of an individual process in the form of its (possibly interrupted) **sequence of instructions**

The **behavior of the system as a whole** can be characterized by showing how the **traces of its various processes are** ***"interleaved"***

Dispatcher

Small OS program that points the CPU from one process to another



(a)	(b)	(c)
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A
(if not interrupted)

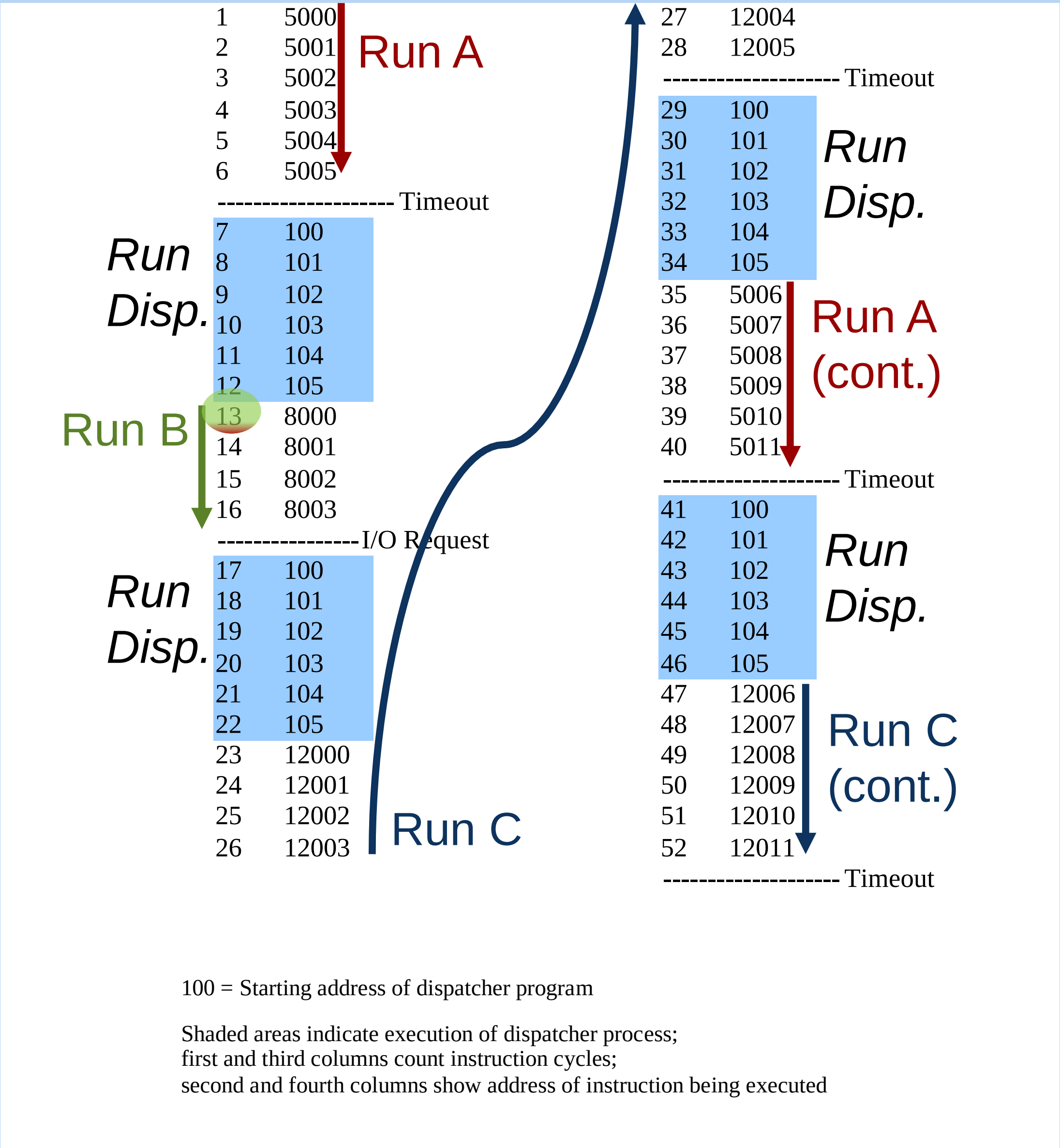
(b) Trace of Process B
(if not interrupted)

(c) Trace of Process C
(if not interrupted)

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

In *Time-Sharing Systems* (see Chapter 2):



```
1      5000                          27     12004
2      5001                          28     12005
3      5002                          -------------------- Timeout
4      5003                          29     100
5      5004                          30     101
6      5005                          31     102
-------------------- Timeout         32     103
7      100                           33     104
8      101                           34     105
9      102                           35     5006
10     103                           36     5007
11     104                           37     5008
12     105                           38     5009
13     8000                          39     5010
14     8001                          40     5011
15     8002                          -------------------- Timeout
16     8003                          41     100
----------------I/O Request          42     101
17     100                           43     102
18     101                           44     103
19     102                           45     104
20     103                           46     105
21     104                           47     12006
22     105                           48     12007
23     12000                         49     12008
24     12001                         50     12009
25     12002                         51     12010
26     12003                         52     12011
                                     -------------------- Timeout
```

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2 , **for 52 CPU Cycles**

CPU cycle number **13** refers to the situation of Figure 3.2

If the Dispatcher does not get activated frequently enough, then the many users do not get a “smooth” inter-activity experience, (i.e.: long waiting time for the next user process to get “attention” of the CPU).

If the Dispatcher gets activated too often, then too much valuable CPU-time is consumed by the Dispatcher itself (rather than by the various user processes).

In this example, *the Dispatcher “consumes” 24 out of 52 CPU Cycles*, which implies that only about 50% percent of all CPU-Time is available for the user processes !

Activity Diagram:

Multi-Programming with **1 CPU**

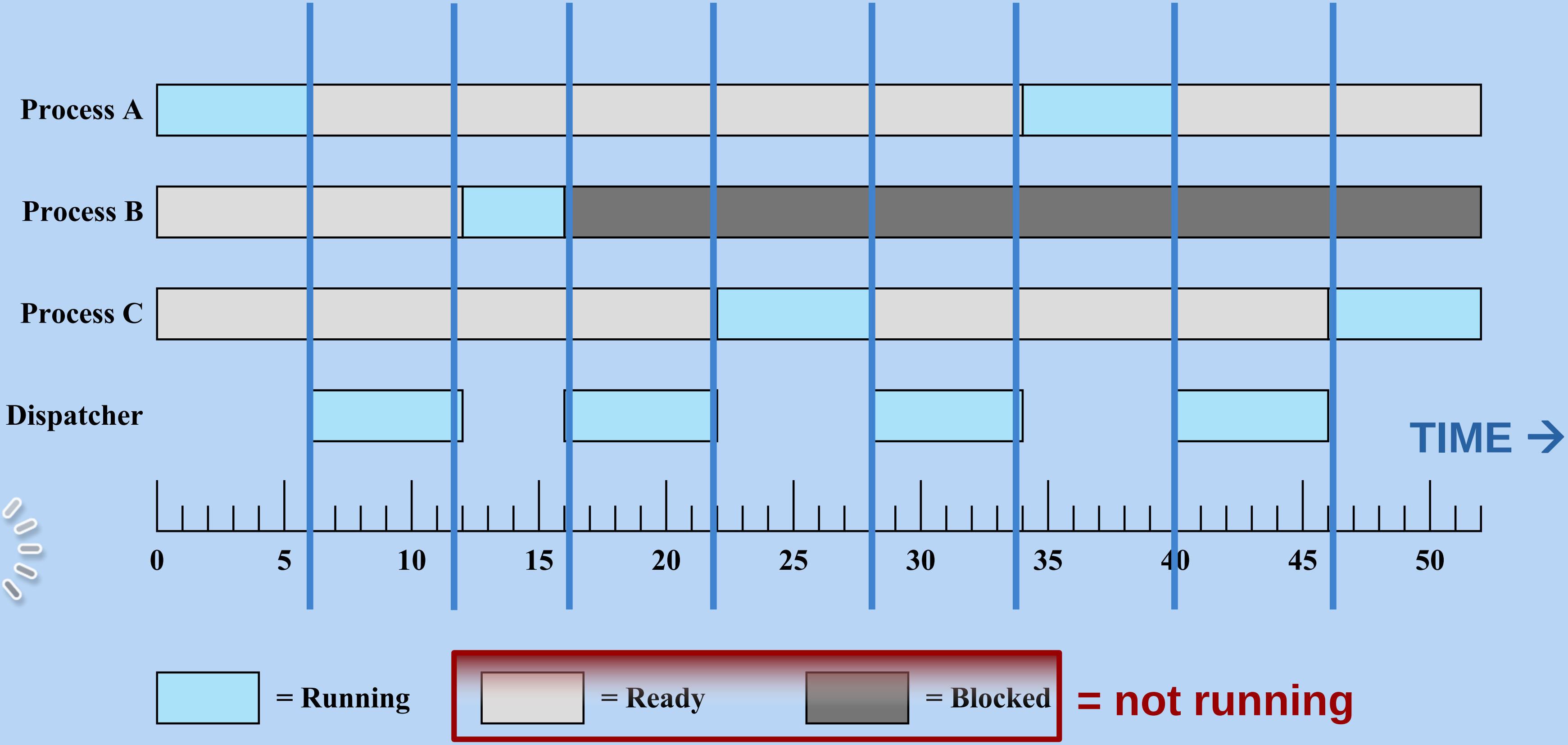
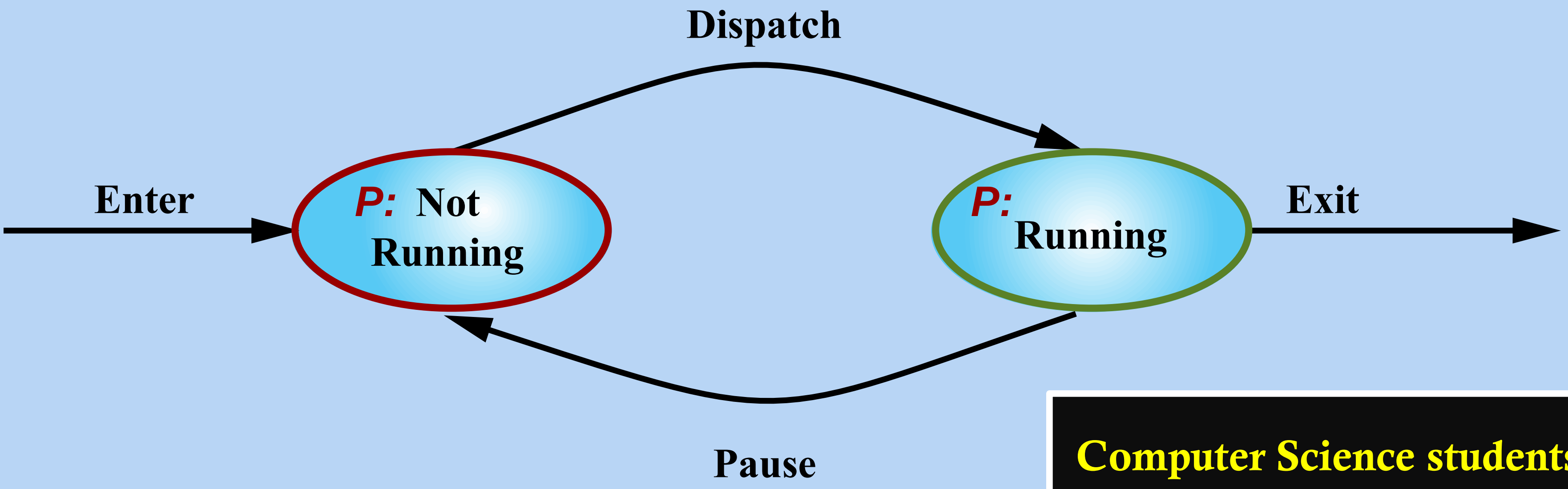


Figure 3.7 Process States for Trace of Figure 3.4

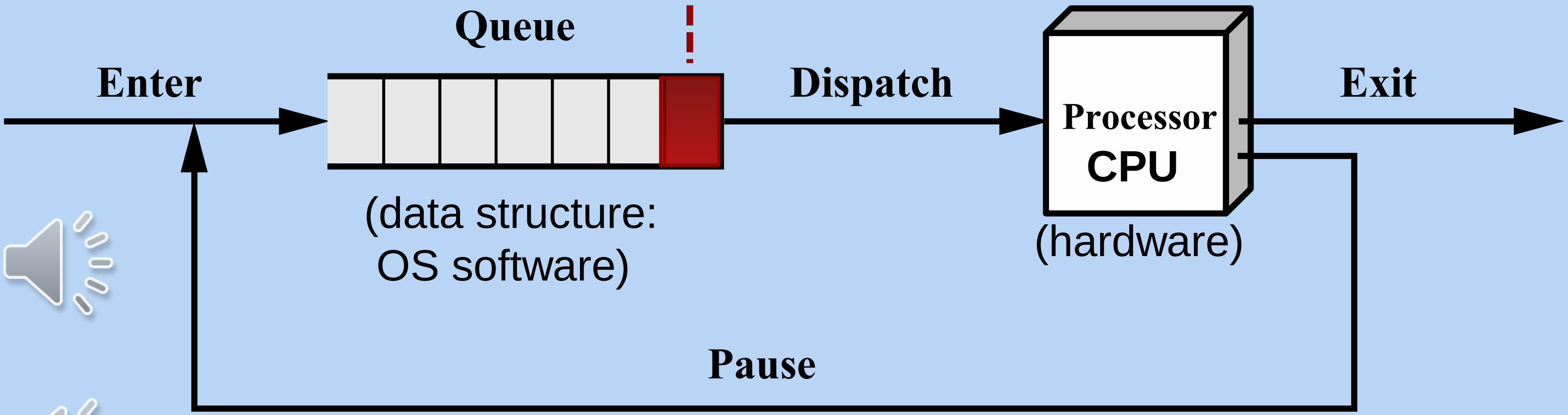
Two-State Process Model for some Process *P*



(a) State transition diagram

Computer Science students: For "***State Transition Diagram***" recall **Fig.17.4** in Chapter **17** of **COS151** Introduction to Comp.Sc.!

PCB is in Queue while its corresponding Process is in State "***not running***"



Queue

Enter

Dispatch

Exit

Processor

CPU

(data structure: OS software)

(hardware)

Pause

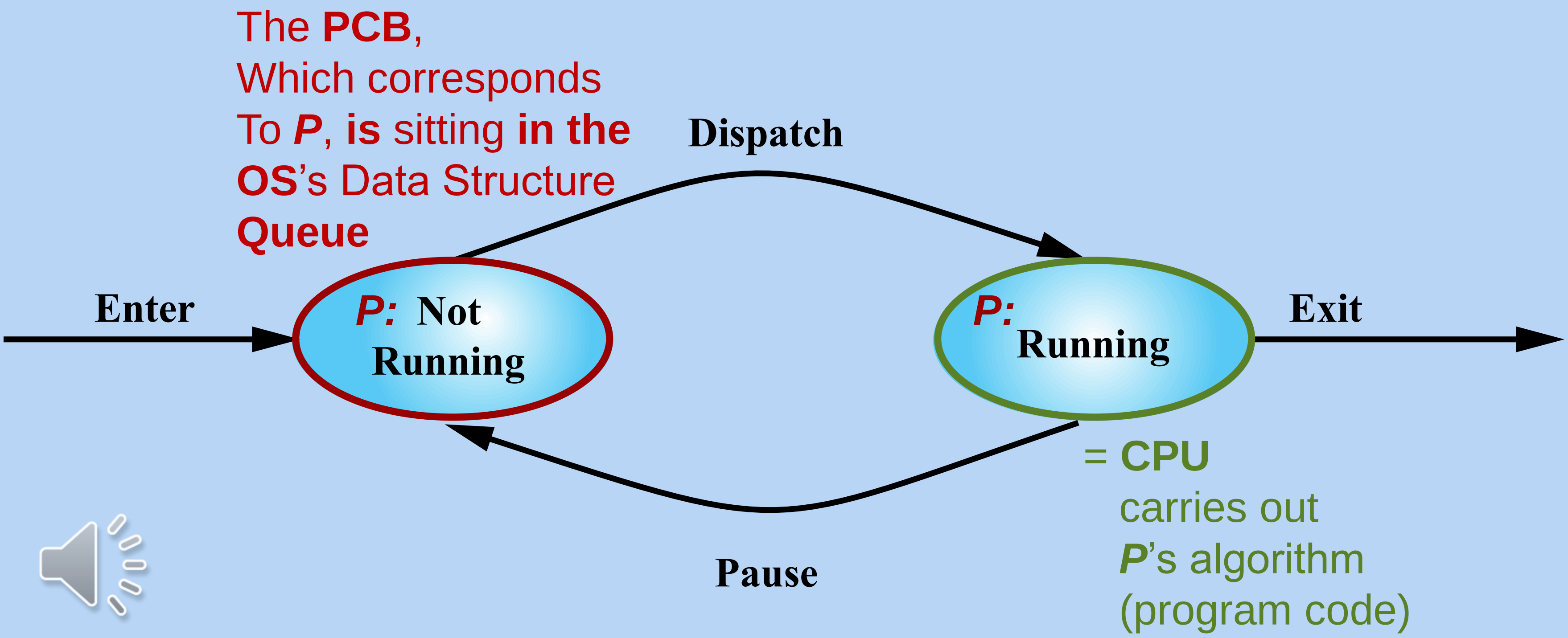
(b) Queuing diagram

Technical *Implementation* of the

Figure 3.5 Two-State Process Model

which *corresponds* to the abstract State Transition Diagram (model) shown on the previous slide

Two-State Process Model for some Process *P*



The **PCB**,
Which corresponds
To ***P***, **is** sitting **in the**
OS's Data Structure
Queue

= **CPU**
carries out
P's algorithm
(program code)

(a) State transition diagram

Conceptual Relationship

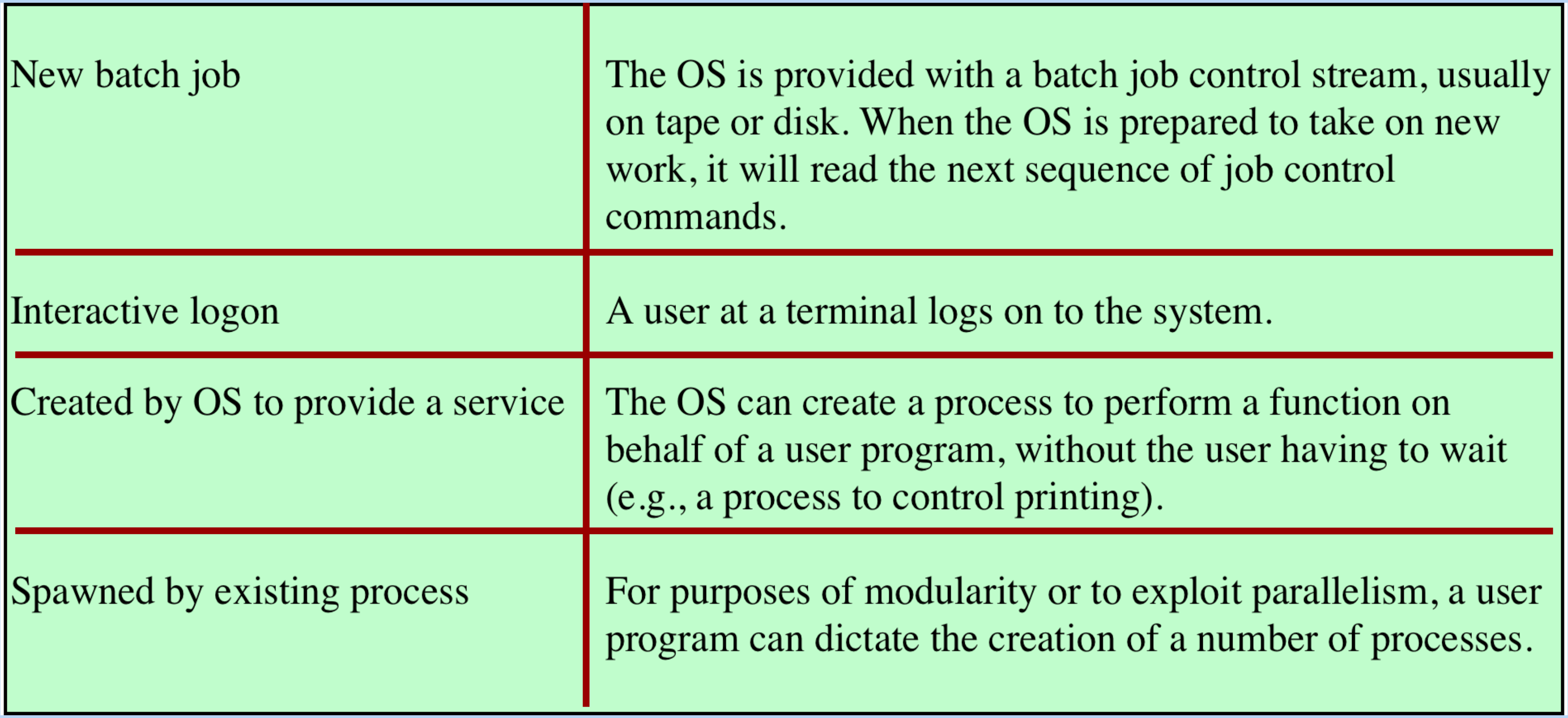
State Transition Diagram

- **Abstract Model** which depicts a **general principle** of operation
- Serves as a **Specification** of **what** shall be implemented

Queuing Diagram

- Shows, in **more technical detail**, **how** the general principle has been **implemented**
- *Typically there are more than 1 possible implementations for a given abstract specification.*

Table 3.1 Reasons for Process Creation



New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Creation by *Spawning*

Process spawning

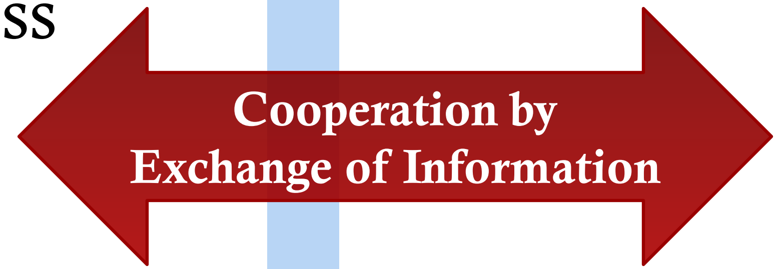
- When the OS creates a process at the explicit request of another process

Parent process

- Is the original, creating, process

Child process

- Is the new process



Cooperation by Exchange of Information

Further Details → **Chapter 5** of **this book**, and **2nd-Year** Course on *Concurrent Programming*

Process Termination: **When a user-algorithm has been carried out completely, then how does the OS know that the corresponding PCB may now be deleted entirely from the "logbook" of all PCBs ?**

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Table 3.2

Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

An *undesirable* “zombie” scenario

- Parent process **P** spawns a Child process **C**
- **P** instructs **C** to collect some data which **P** needs
- In the meantime, **P** is doing some other work …
- For some reason, **P** “dies”

- **C** now exists
- **C** begins to collect data …
- **C** completes the data collection for its parent **P**
- **C** keeps trying and trying and trying (in vain) to communicate the collected data to **P** … … …

➔ The Operating System’s *Process Termination Regulations* must be implemented in such a manner that such a scenario *cannot* occur !

"Zombie"

- An existing process (with PCB) which only "colonizes" resources (e.g.: CPU-time and/or memory-space) **without being able to do any useful work**.

- They can *typically* arise:
  - From "hard" system-crashes and subsequent re-booting *without* proper recovery procedures (➔ Chapter 12)
  - From older versions of user application software not properly un-installed when a newer version of same software got installed

Five-State Process Model for some Process *P* :

indicates, in more details, the reasons WHY a process *P* is "not running"

Note:
"ready" does NOT mean "done"!
"ready" means: "prepared to do something"!

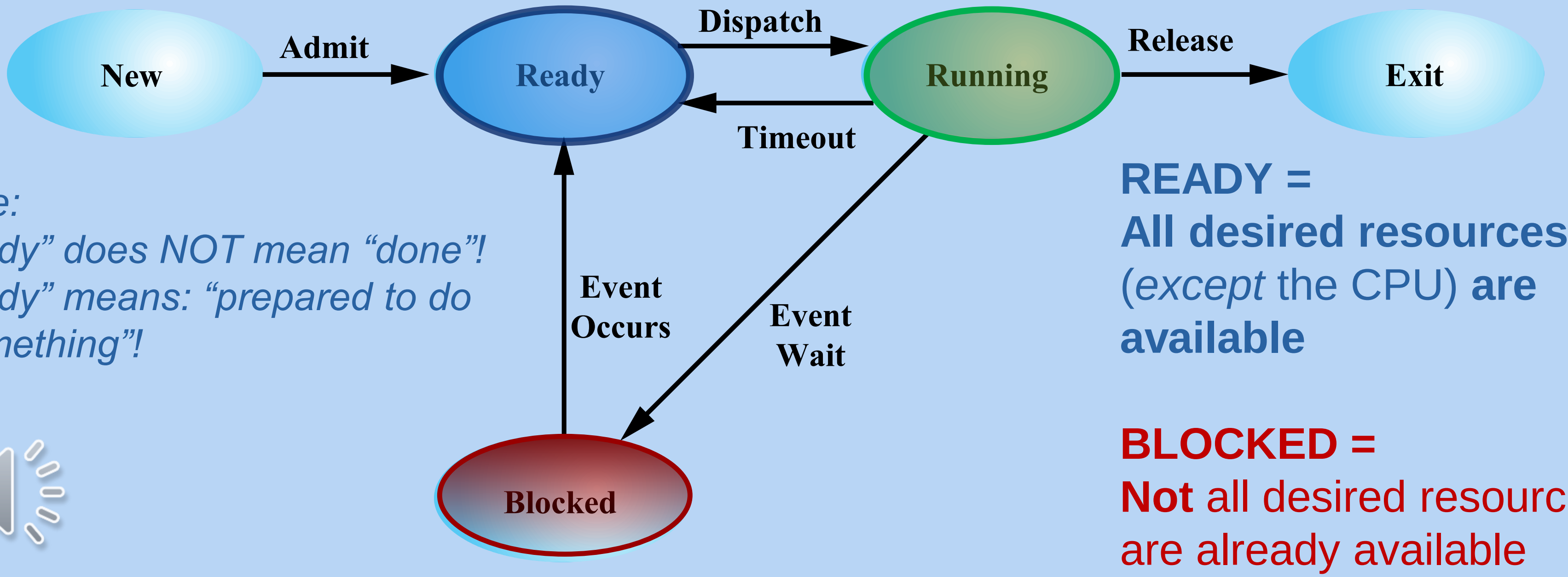
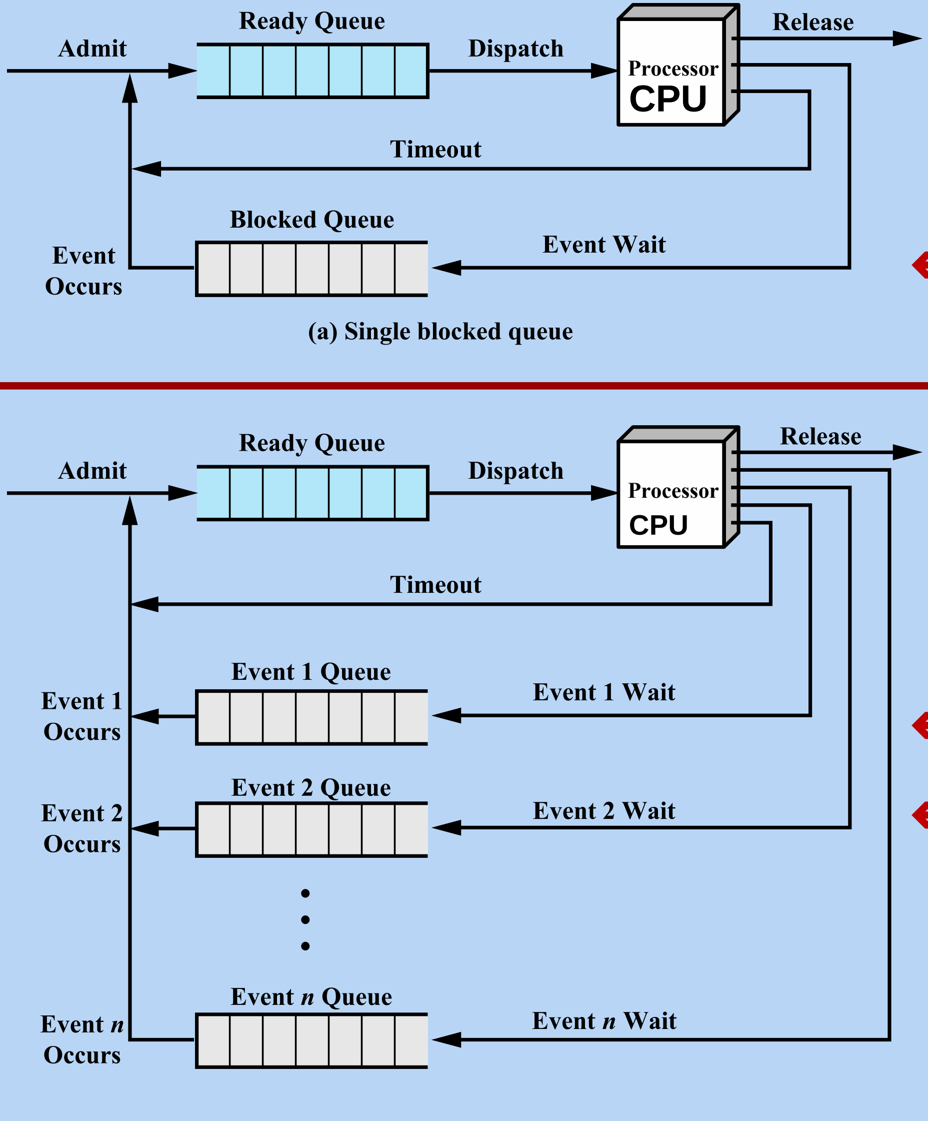


Figure 3.6 Five-State Process Model

READY =
All desired resources (*except* the CPU) **are available**

BLOCKED =
Not all desired resources are already available (e.g. some data, which ***P*** needs, have not yet arrived via I/O)



Ready Queue

Admit

Dispatch

Processor

CPU

Release

Timeout

Blocked Queue

Event Wait

Event Occurs

(a) Single blocked queue

← “Need *something*”

Ready Queue

Admit

Dispatch

Processor

CPU

Release

Timeout

Event 1 Queue

Event 1 Wait

Event 1 Occurs

Event 2 Queue

Event 2 Wait

Event 2 Occurs

Event *n* Queue

Event *n* Wait

Event *n* Occurs

(b) Multiple blocked queues

For example:

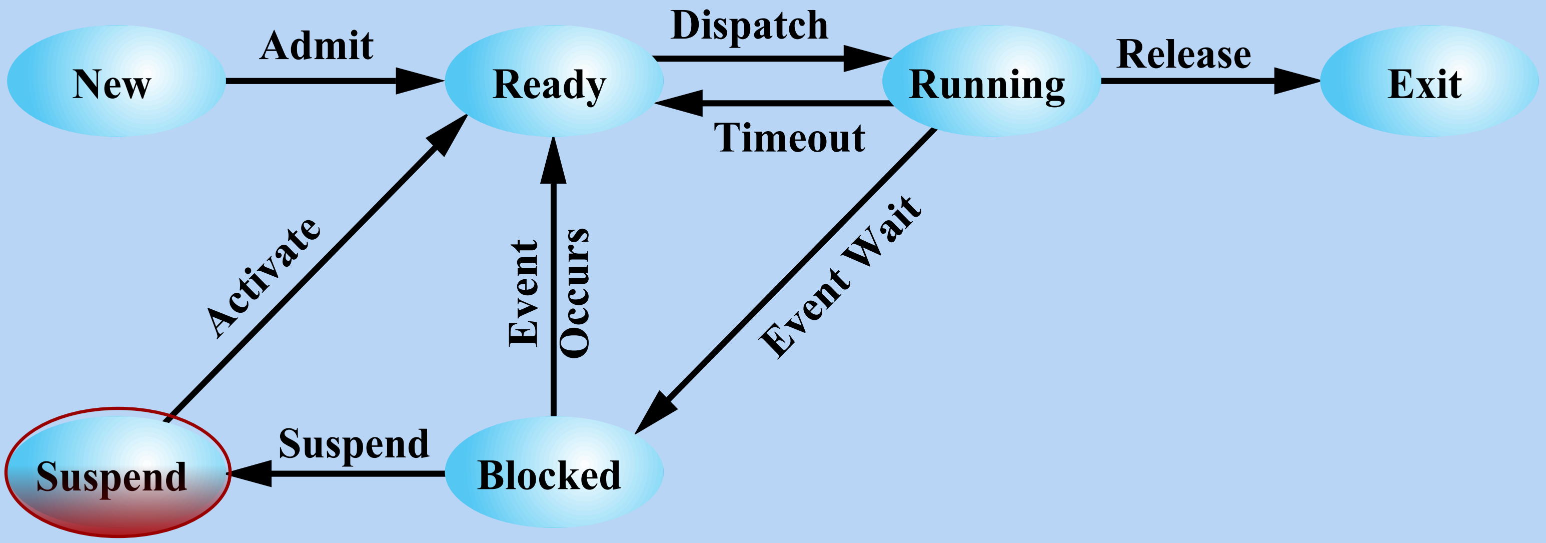
← “Need Printer”

← “Need Hard-Disk”

Figure 3.8 Queuing Model for Figure 3.6

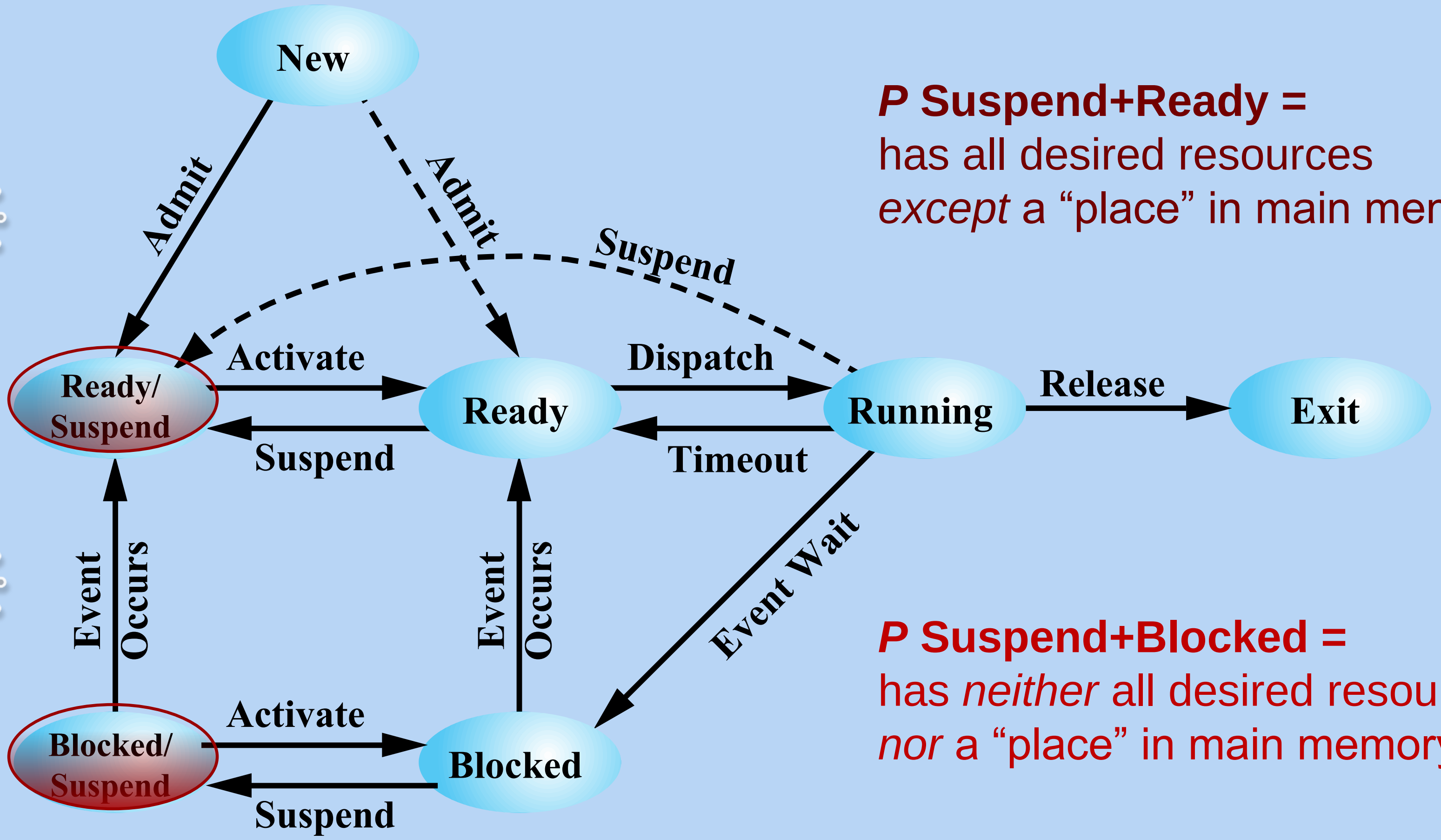
"Suspended" Processes: *"severely" blocked!*

- **Swapping**
  - Involves **moving** part of all of a process **from main memory to disk**
  - When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue
    - This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended
    - The OS then brings in another process from the suspend queue or it honors a new-process request
    - Execution then continues with the newly arrived process
  - ➔ See **Chapters 7** (*Memory Management*) and **11** (*Disk Management*)



(a) With One Suspend State

Figure 3.9 Process State Transition Diagram with Suspend States



***P* Suspend+Ready =**
has all desired resources
except a “place” in main memory

***P* Suspend+Blocked =**
has *neither* all desired resources
nor a “place” in main memory

(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Characteristics of a Suspended Process

- The process is not immediately available for execution: **code not in Main Memory! →** ***CPU’s PC cannot point to it!***
- The process may or may not be waiting on an event or resource
- The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution
- The process may not be removed from this state until the agent explicitly orders the removal

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.