Operating Systems: Internals and Design Principles

Chapter 3
Process Description and Control

Ninth Edition

By William Stallings

Summary of Earlier Concepts

- A computer platform consists of a collection of hardware resources
- Computer applications are developed to perform some task
- It is inefficient for applications to be written directly for a given hardware platform
- The OS was developed to provide a convenient, feature-rich, secure, and consistent interface for applications to use
- We can think of the OS as providing a uniform, abstract representation of resources that can be requested and accessed by applications

OS Management of Application Execution

- Resources are made available to multiple applications
- The processor is switched among multiple applications so all will appear to be progressing
- The processor and I/O devices can be used efficiently
- The execution of an application corresponds to the existence of one or more processes

Process Elements

Two essential elements of a process are:

- **Program code**, which may be shared with other processes that are executing the same program
- **A set of data** associated with that code

When the processor begins to execute the program code, we refer to this executing entity as a ***process***

In Chapter 2, we also mentioned another component, the **execution context** (or *process state*) of the program

Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:
  - **Identifier**: A unique identifier associated with this process, to distinguish it from all other processes
  - **State**: If the process is currently executing, it is in the running state
  - **Priority**: Priority level relative to other processes
  - **Program counter**: The address of the next instruction in the program to be executed
  - **Memory pointers**: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes

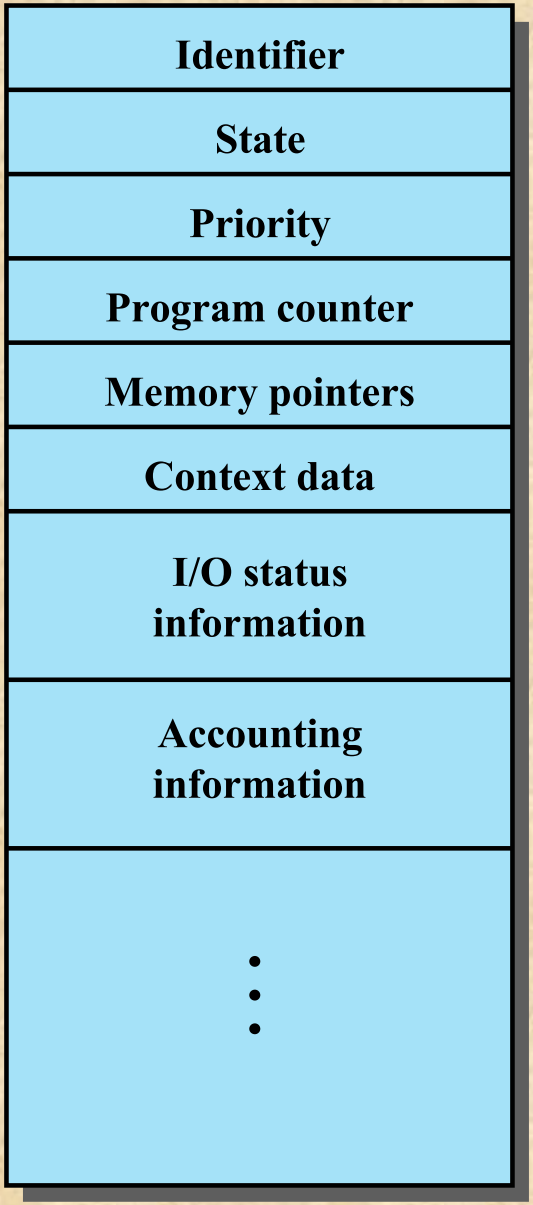
Process Elements

- **Context data**: These are data that are present in registers in the processor while the process is executing
- **I/O status information**: Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on
- **Accounting information**: May include the amount of processor time and clock time used, time limits, account numbers, and so on

- The information is stored in a data structure, typically called a **process control block**

Process Control Block

- Created and managed by the OS
- Contains sufficient information so it is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Key tool that enables the OS to support multiple processes and to provide for multiprocessing



Identifier

State

Priority

Program counter

Memory pointers

Context data

I/O status information

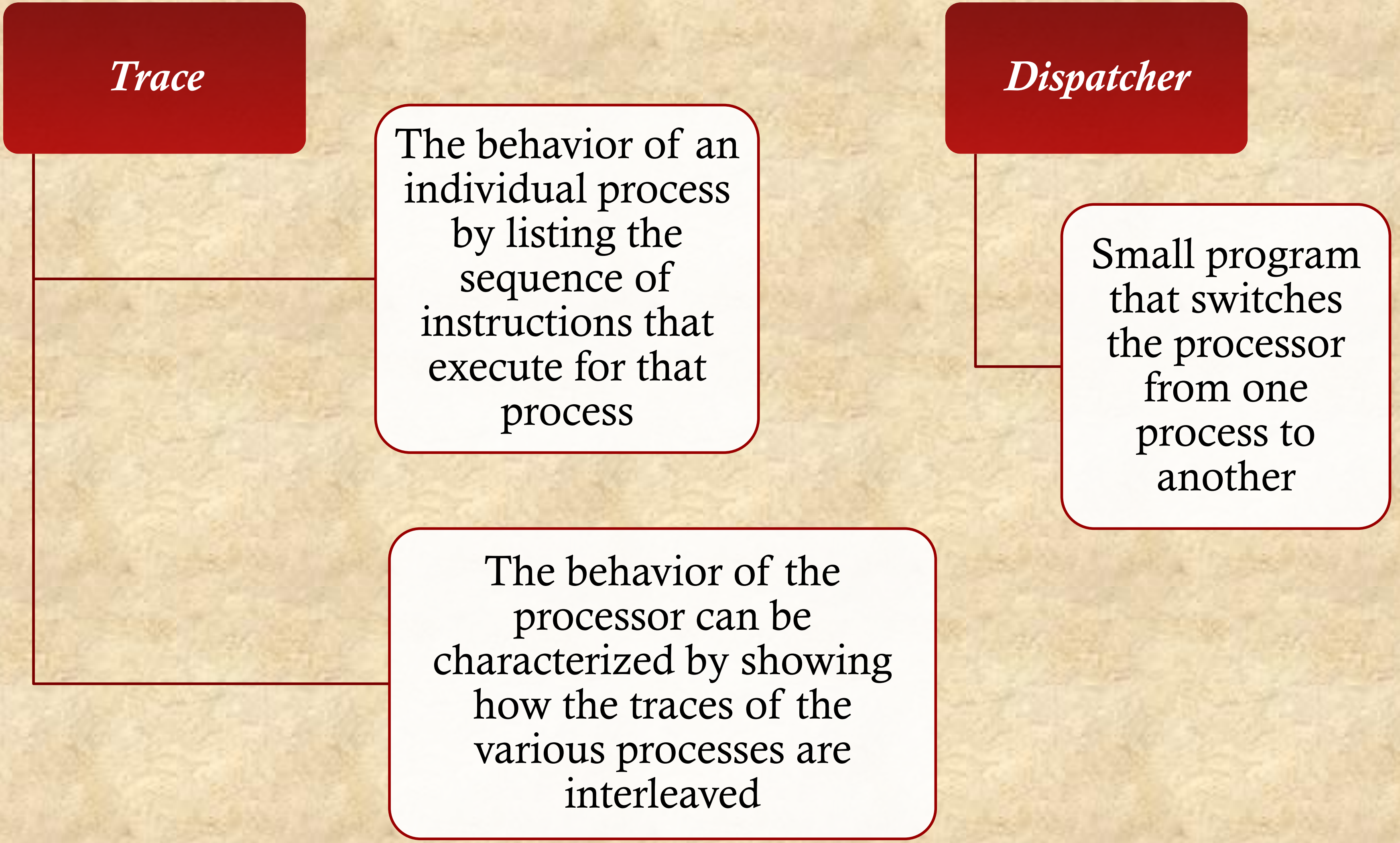
Accounting information

⋮

Figure 3.1 Simplified Process Control Block

Process Control Block

- When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value
- The OS is now free to put some other process in the *running* state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute
- Thus, we can say that a process consists of program code and associated data plus a process control block



Trace

The behavior of an individual process by listing the sequence of instructions that execute for that process

The behavior of the processor can be characterized by showing how the traces of the various processes are interleaved

Dispatcher

Small program that switches the processor from one process to another

Process Execution

Figure 3.2 shows a memory layout of three processes. In addition, there is a small **dispatcher** program that switches the processor from one process to another

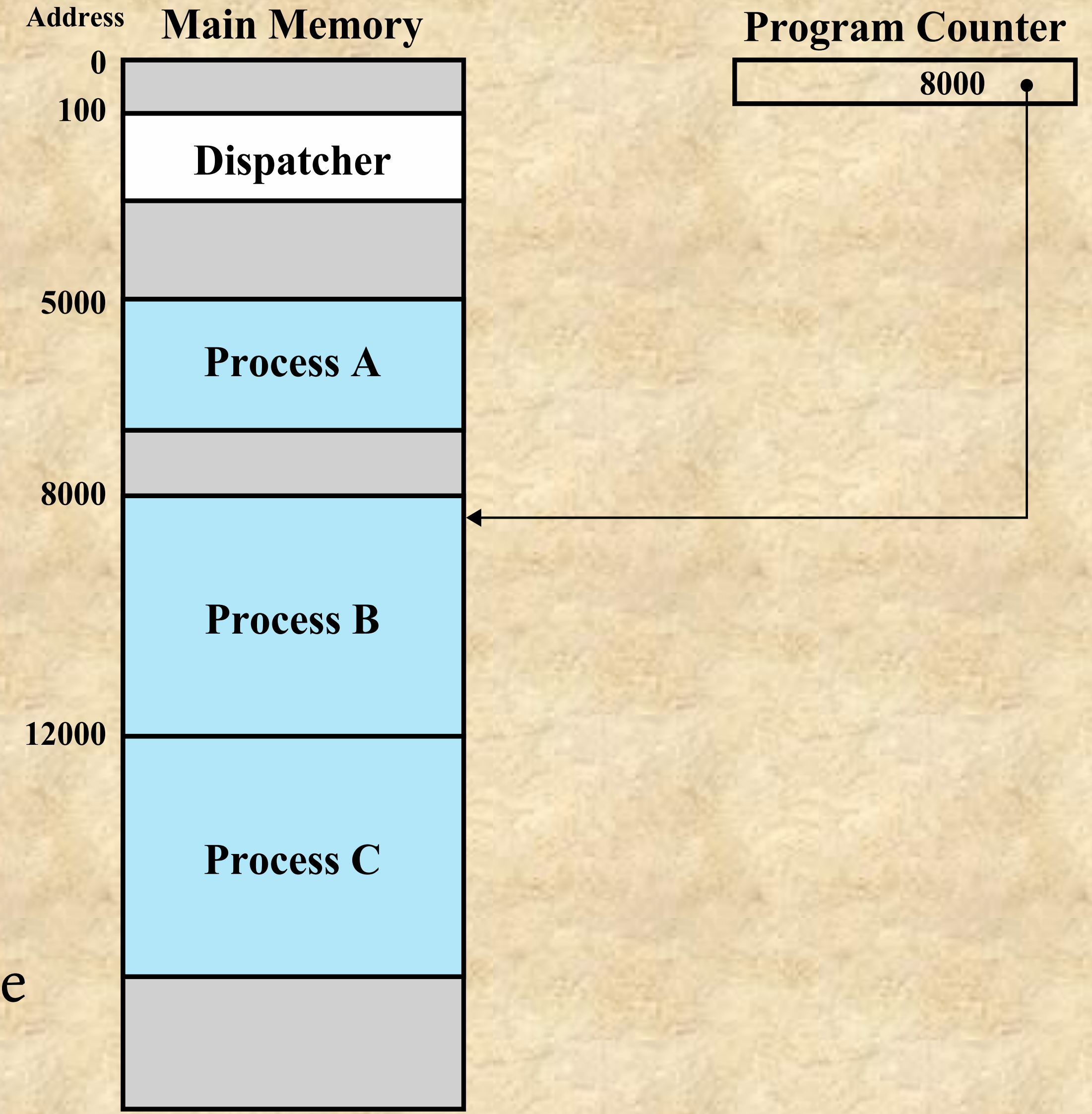


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

Figure 3.3 shows the traces of each of the processes during the early part of their execution. The first 12 instructions executed in processes A and C are shown. Process B executes four instructions, and the fourth instruction invokes an I/O operation for which the process must wait

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	**(b) Trace of Process B**	**(c) Trace of Process C**

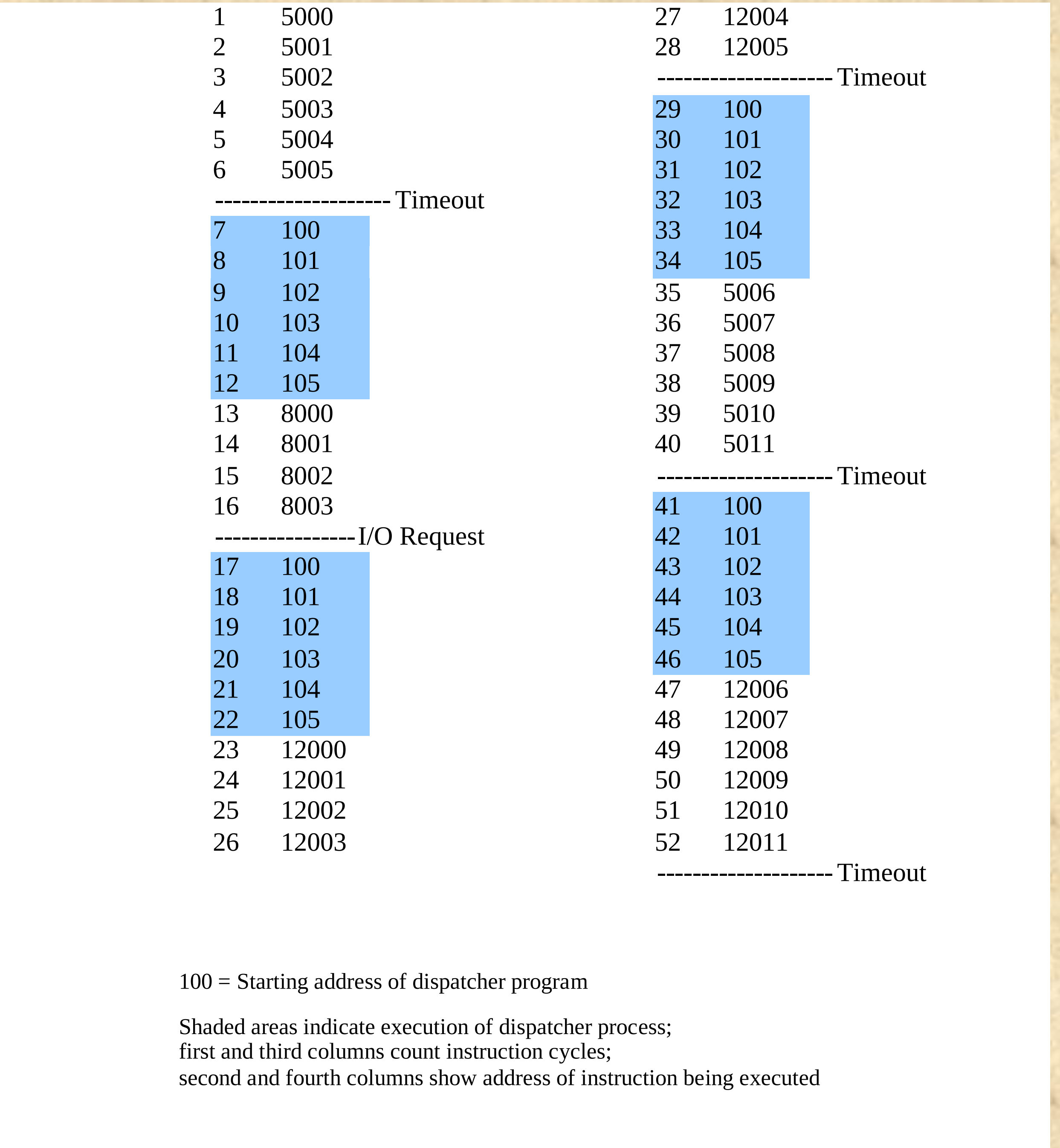
5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Now let us view these traces from the processor's point of view. Figure 3.4 shows the interleaved traces resulting from the first 52 instruction cycles The shaded areas represent code executed by the dispatcher

We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles, after which it is interrupted

What is the sequence of running processes?



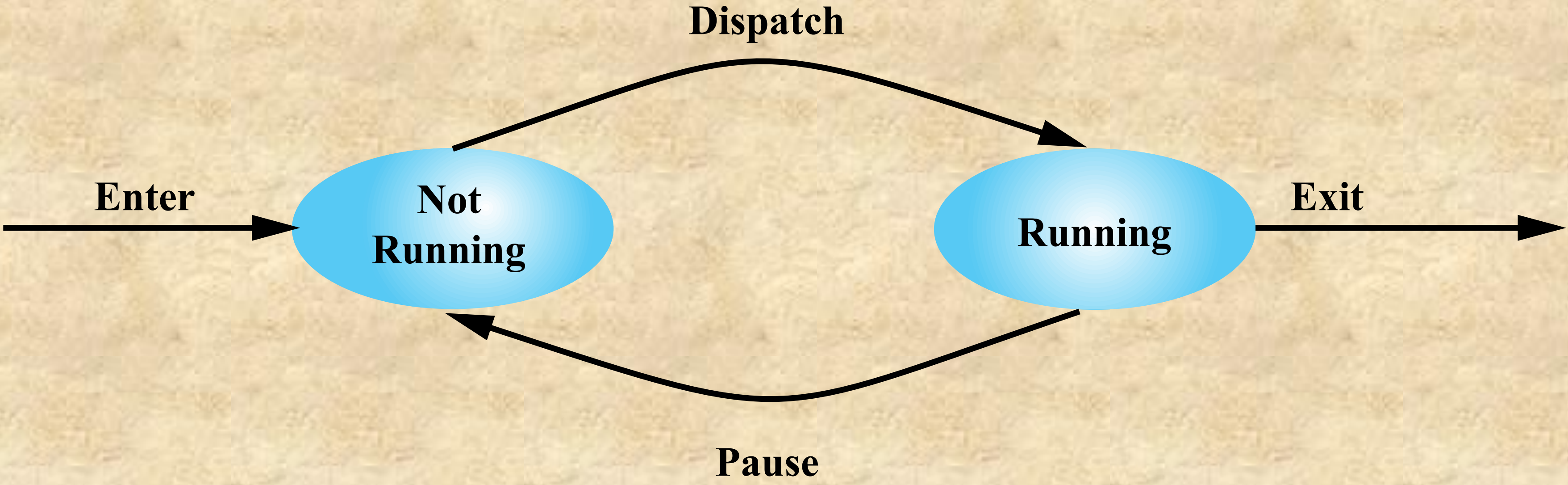
```
1       5000
2       5001
3       5002
4       5003
5       5004
6       5005
-------------------- Timeout
7       100
8       101
9       102
10      103
11      104
12      105
13      8000
14      8001
15      8002
16      8003
----------------I/O Request
17      100
18      101
19      102
20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

100 = Starting address of dispatcher program

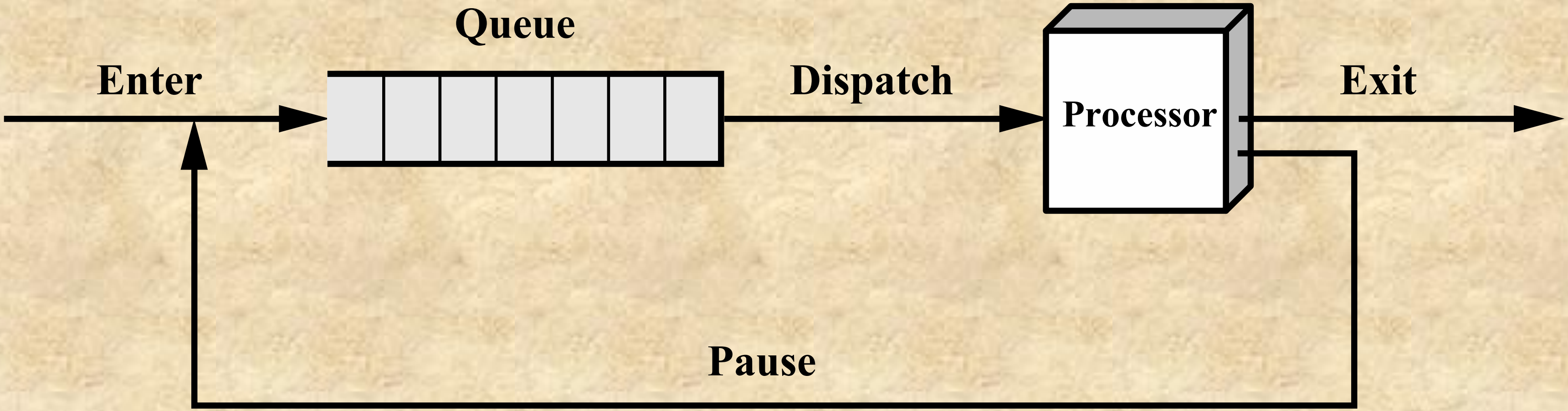
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Two-State Process Model



(a) State transition diagram



(b) Queuing diagram

Figure 3.5 Two-State Process Model

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 Spawning

Process spawning	*Parent process*	*Child process*
• When the OS creates a process at the explicit request of another process	• Is the original, creating, process	• Is the new process

Process Termination

- 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)

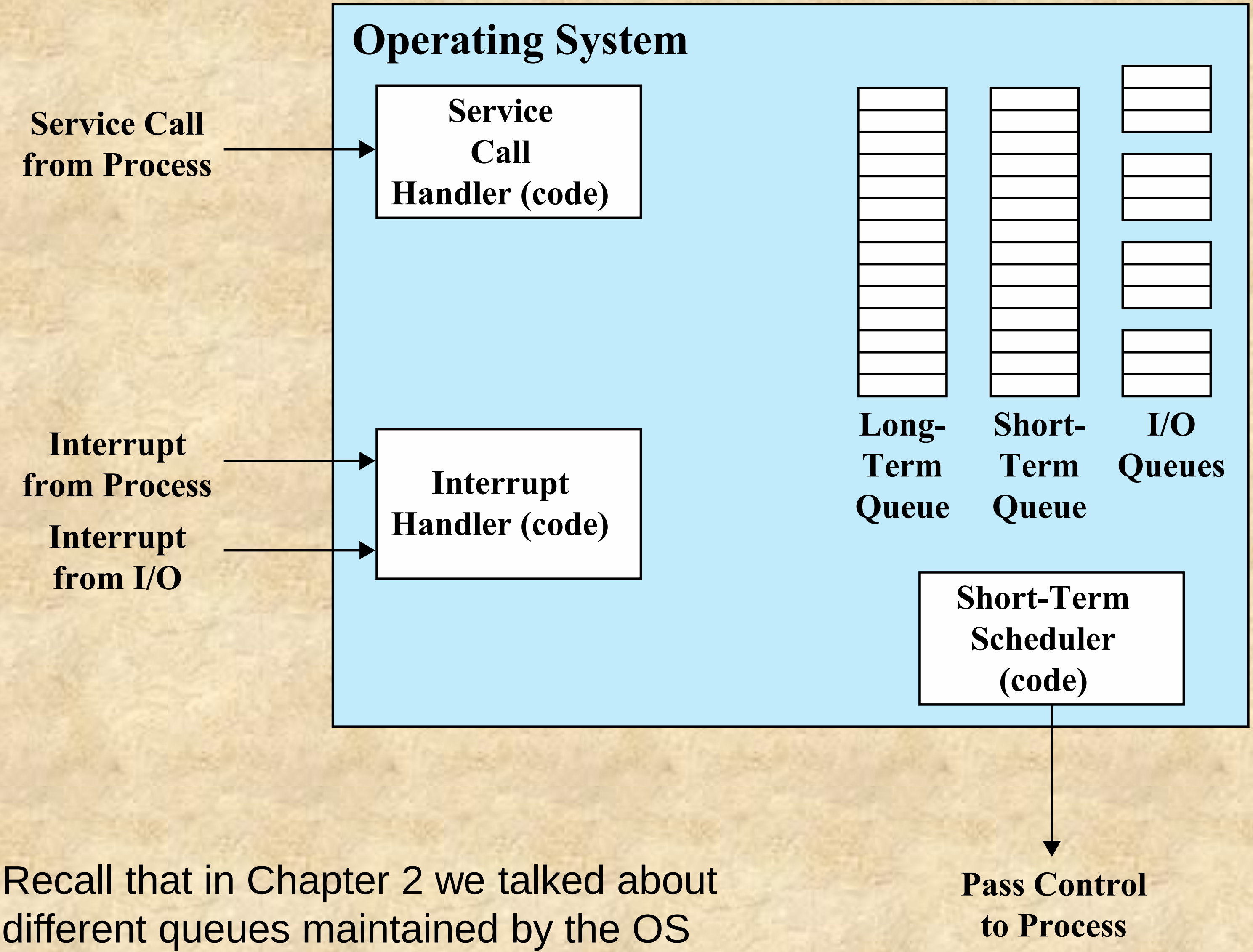
Table 3.2 Reasons for Process Termination

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.

(Table is located on page 111
in the textbook)

Five-State Process Model

- If all processes were always ready to execute, then the queuing discipline suggested by Figure 3.5b would be effective. The queue is a first-in-first-out (FIFO) list and the processor operates in round-robin fashion on the available processes (each process in the queue is given a certain amount of time, in turn, to execute and then returned to the queue, unless blocked)
- However, even with the simple example that we have described, this implementation is inadequate: Some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue. Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest



Recall that in Chapter 2 we talked about different queues maintained by the OS

Figure 2.11 Key Elements of an Operating System for Multiprogramming

Five-State Process Model

- **Running**: The process that is currently being executed. We assume a computer with a single processor, so at most one process at a time can be in this state
- **Ready**: A process that is prepared to execute when given the opportunity
- **Blocked**: A process that cannot execute until some event occurs, such as the completion of an I/O operation. Also called ***waiting***
- **New**: A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created (check the *long-term queue* in Chapter 2)
- **Exit**: A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason

Five-State Process Model

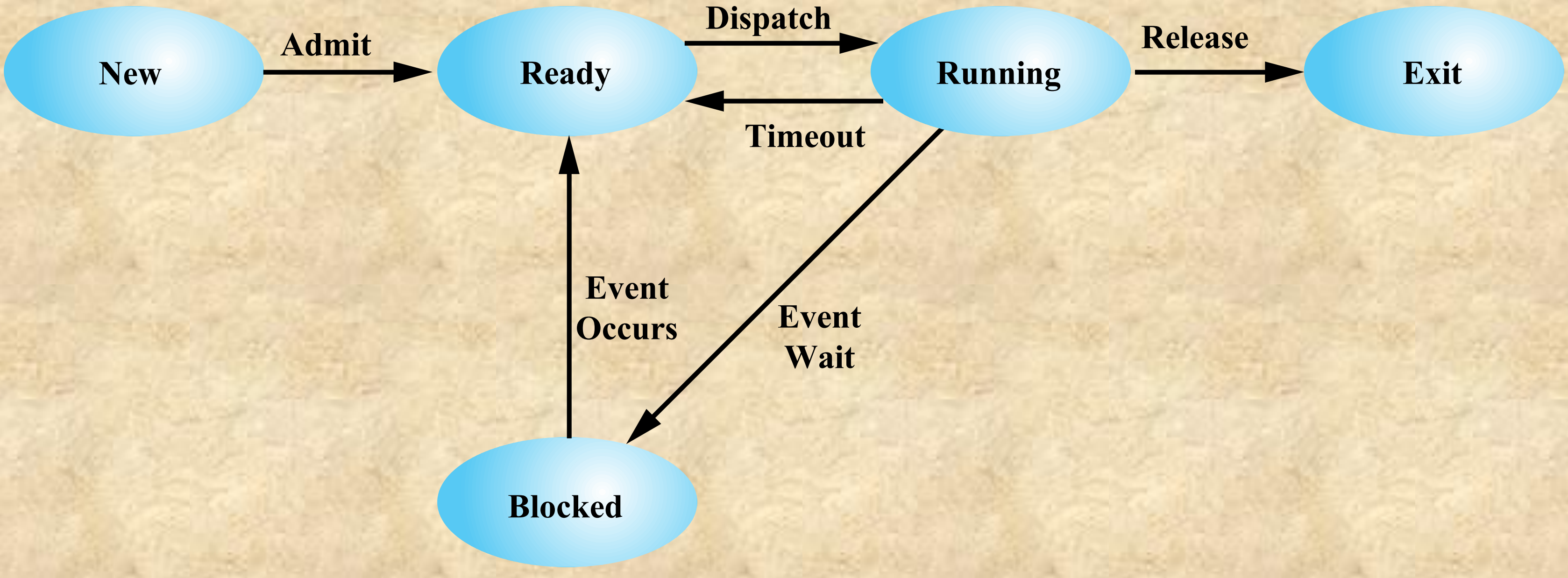


Figure 3.6 Five-State Process Model

You may also have a look at https://en.wikipedia.org/wiki/Process_state

Five-State Process Model

- **Null → New:** A new process is created to execute a program. This event occurs for any of the reasons listed in Table 3.1
- **New → Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on the number of existing processes or the amount of virtual memory committed to existing processes. This limit assures there are not so many active processes as to degrade performance
- **Ready → Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher

Five-State Process Model

- **Running → Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution. There are several other alternative causes for this transition, which are not implemented in all operating systems.
  - For example, the OS assigns different levels of priority to different processes. Suppose, process A is running at a given priority level, and process B, at a higher priority level, is blocked. If the OS learns that the event upon which process B has been waiting has occurred, this moving B to a ready state, then it can interrupt process A and dispatch process B. We say that the OS has **preempted** process A
  - Finally, a process may voluntarily release control of the processor. An example is a background process that periodically performs some accounting or maintenance function

Five-State Process Model

- **Running → Blocked:** A process is put in the **Blocked state** if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code
  - For example, a process may request a service from the OS that the OS is not prepared to perform immediately. It can request a resource, such as a file or a shared section of virtual memory, that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data, or waiting for a message from another process

Five-State Process Model

- **Blocked → Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs
- **Ready → Exit**: For clarity, this transition is not shown on the state diagram. In some systems, a parent may terminate a child process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated
- **Blocked → Exit:** The comments under the preceding item apply

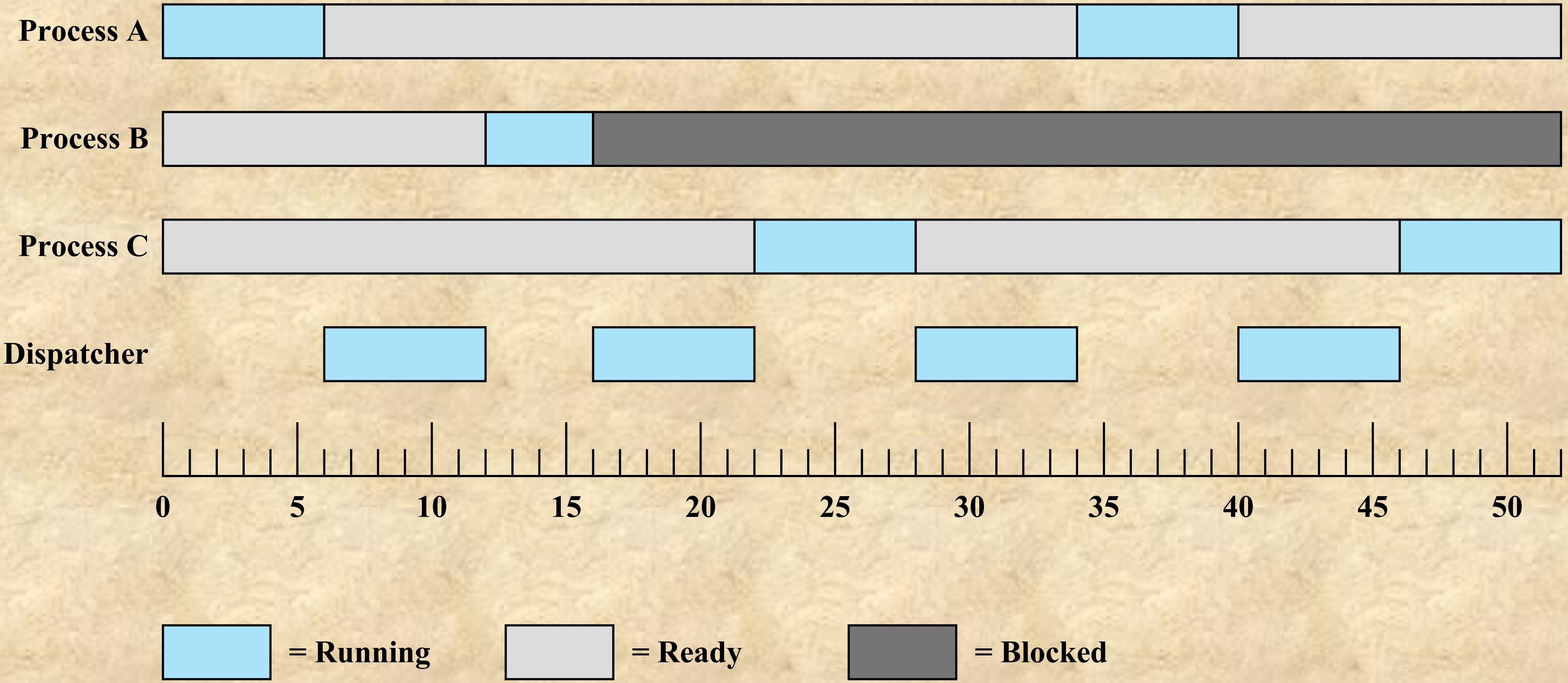


Figure 3.7 Process States for Trace of Figure 3.4

Step-by-step refinements of the queuing model:

- Separate **Ready** and **Blocked** queues
- **Multiple Blocked** queues, one for each event
- If the dispatching of processes is dictated by a priority scheme, then it would be convenient to have **a number of Ready queues, one for each priority level**. The OS could then readily determine which is the highest-priority ready process that has been waiting the longest

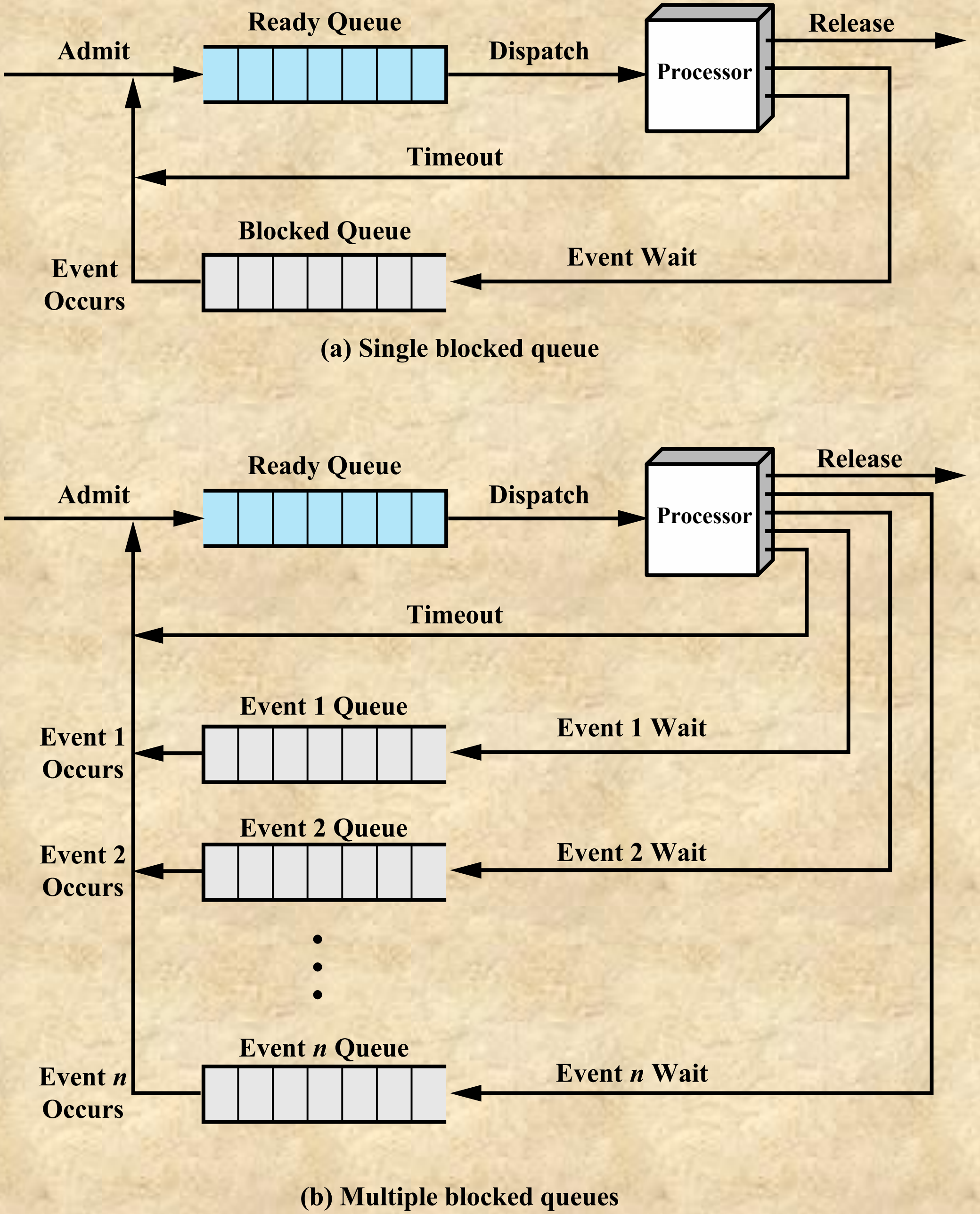
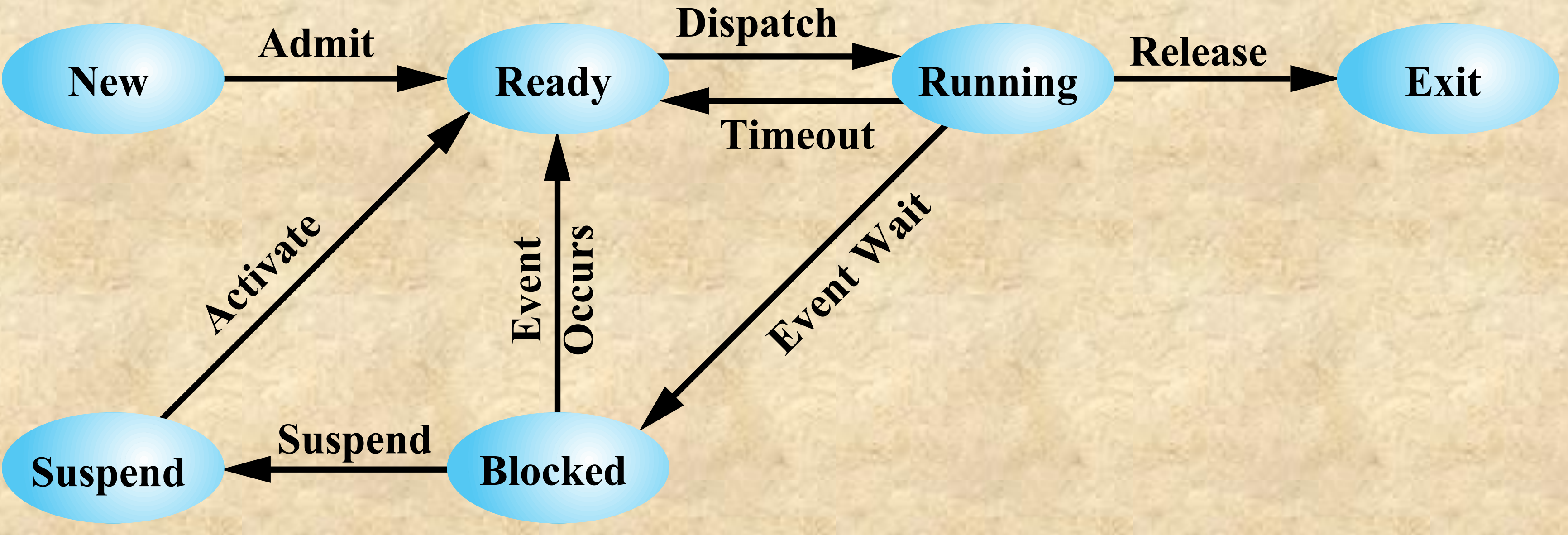


Figure 3.8 Queuing Model for Figure 3.6

Suspended Processes

- Swapping
  - Involves moving part or 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
  - Swapping, however, is an I/O operation and therefore there is the potential for making the problem worse, not better. Because disk I/O is generally the fastest I/O on a system, swapping will usually enhance performance

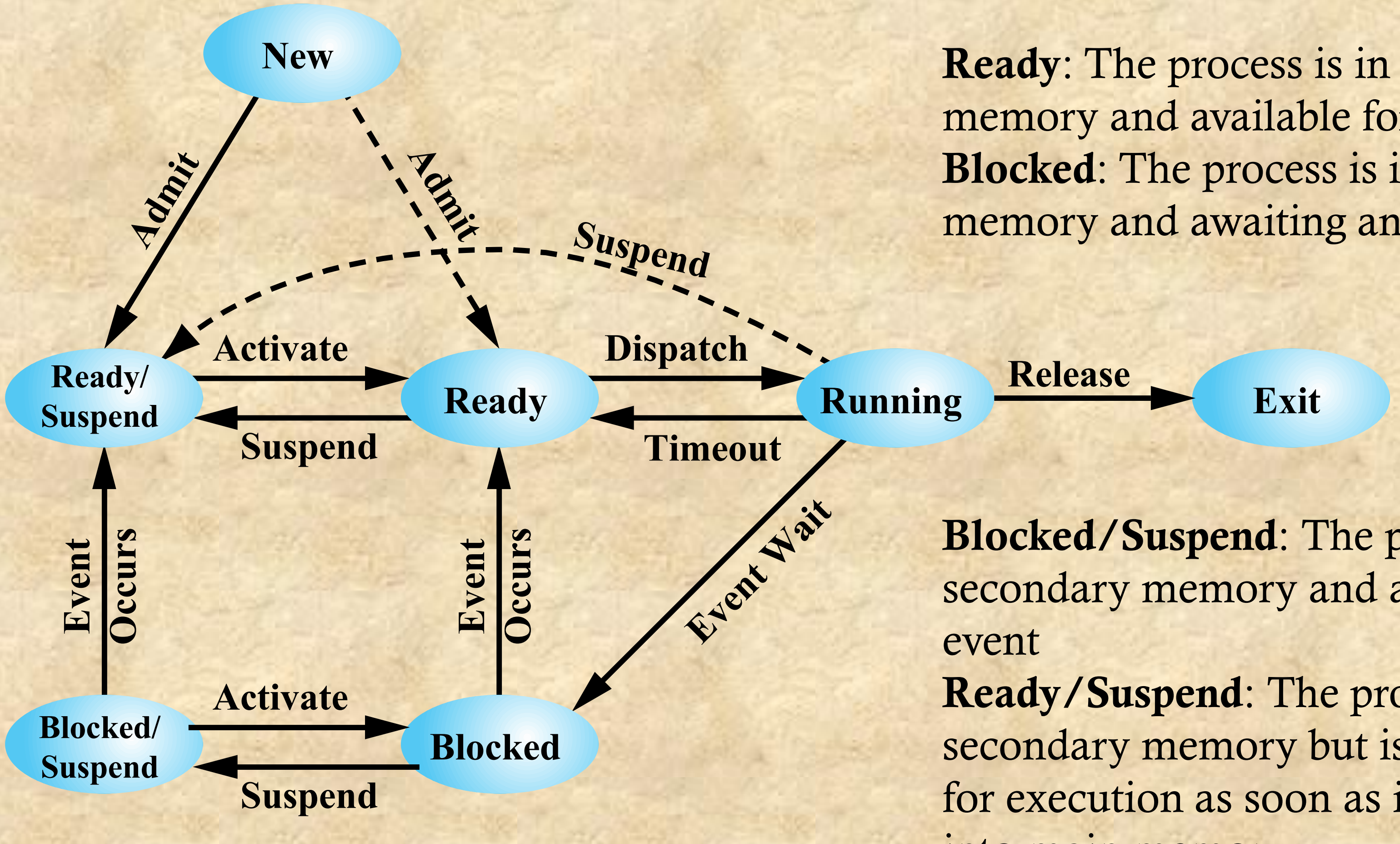


(a) With One Suspend State

Figure 3.9 Process State Transition Diagram with Suspend States

Is one Suspend state enough?

Recall that each process in the Suspend state was originally blocked on a particular event. When that event occurs, the process is not blocked and is potentially available for execution



(b) With Two Suspend States

Ready: The process is in main memory and available for execution

Blocked: The process is in main memory and awaiting an event

Blocked/Suspend: The process is in secondary memory and awaiting an event

Ready/Suspend: The process is in secondary memory but is available for execution as soon as it is loaded into main memory

Figure 3.9 Process State Transition Diagram with Suspend States

State Transition Model with Suspend States

- **Blocked** → **Blocked/Suspend**: If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked
  - This transition can be made even if there are ready processes available, if the OS determines that the currently running process or a ready process that it would like to dispatch requires more main memory to maintain adequate performance
- **Blocked/Suspend** → **Ready/Suspend:** A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs

State Transition Model with Suspend States

- **Ready/Suspend → Ready:** When there are no ready processes in main memory, the OS will need to bring one in to continue execution
  - In addition, it might be the case that a process in the Ready/Suspend state has higher priority than any of the processes in the Ready state. In that case, the OS designer may dictate that it is more important to get at the higher-priority process than to minimize swapping

State Transition Model with Suspend States

- **Ready → Ready/Suspend:** Normally, the OS would prefer to suspend a blocked process rather than a ready one, because the ready process can now be executed, whereas the blocked process is taking up main memory space and cannot be executed. However, it may be necessary to suspend a ready process if that is the only way to free up a sufficiently large block of main memory. Also, the OS may choose to suspend a lower–priority ready process rather than a higher–priority blocked process if it believes that the blocked process will be ready soon

State Transition Model with Suspend States

- **New → Ready/Suspend and New → Ready:** When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue. In either case, the OS must create a process control block and allocate an address space to the process. It might be preferable for the OS to perform these housekeeping duties at an early time, so that it can maintain a large pool of processes that are not blocked. With this strategy, there would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition

State Transition Model with Suspend States

- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? **But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that the blocking event for that process will occur soon. Under these circumstances, it would seem reasonable to bring a blocked process into main memory in preference to a Ready/Suspend process (ready process here in the textbook, which is not correct)**

State Transition Model with Suspend States

- **Running → Ready/Suspend:** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory

- **Any State → Exit:** Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state

Characteristics of a Suspended Process

We can generalize the concept of a suspended process. Let us define a suspended process as having the following characteristics:

1. The process is not immediately available for execution

2. The process may or may not be waiting on an event. If it is, this blocked condition is independent of the suspend condition, and occurrence of the blocking event does not enable the process to be executed immediately

3. 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

4. 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.

Fundamentally, we can think of the OS as that entity that manages the use of system resources by processes

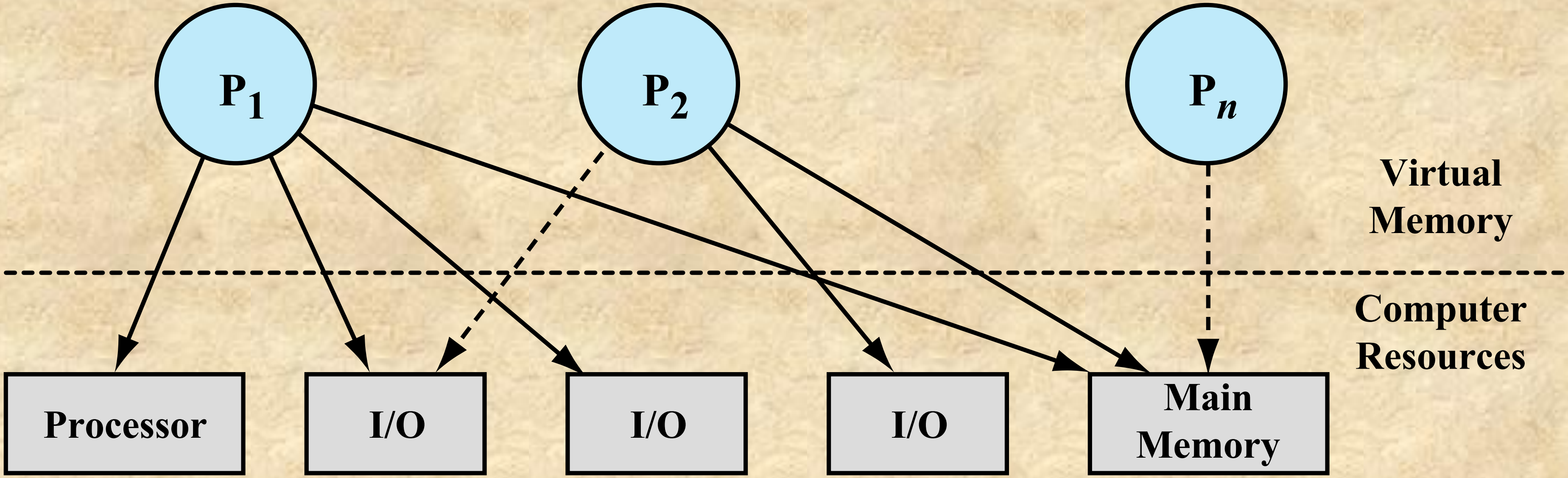
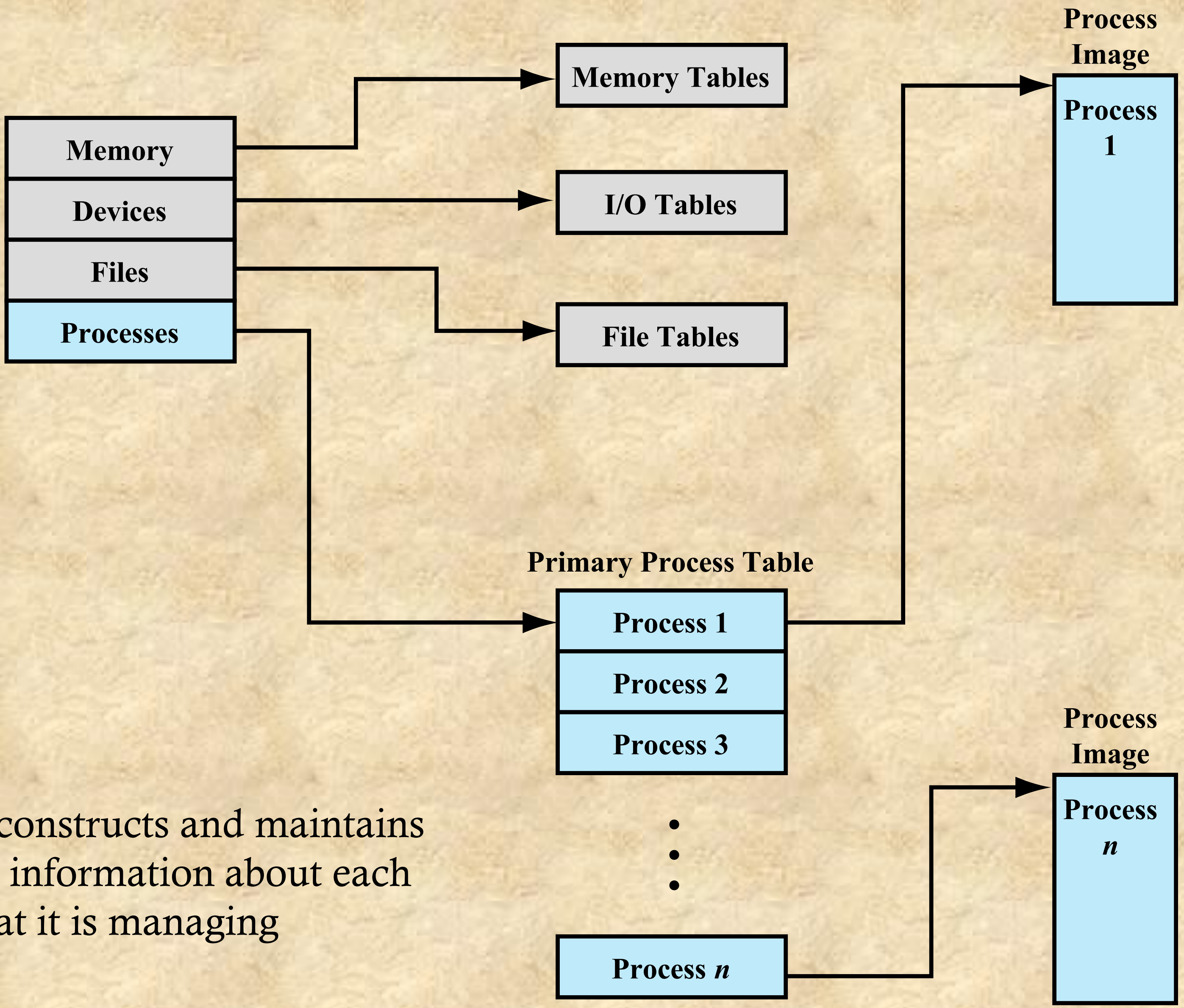


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)



The OS constructs and maintains tables of information about each entity that it is managing

Figure 3.11 General Structure of Operating System Control Tables

Memory Tables

- Used to keep track of both main (real) and secondary (virtual) memory
- Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism

Must include:

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes of blocks of main or virtual memory
- Information needed to manage virtual memory

I/O Tables

- Used by the OS to manage the I/O devices and channels of the computer system
- At any given time, an I/O device may be available or assigned to a particular process

If an I/O operation is in progress, the OS needs to know:

- The status of the I/O operation
- The location in main memory being used as the source or destination of the I/O transfer

File Tables

These tables provide information about:

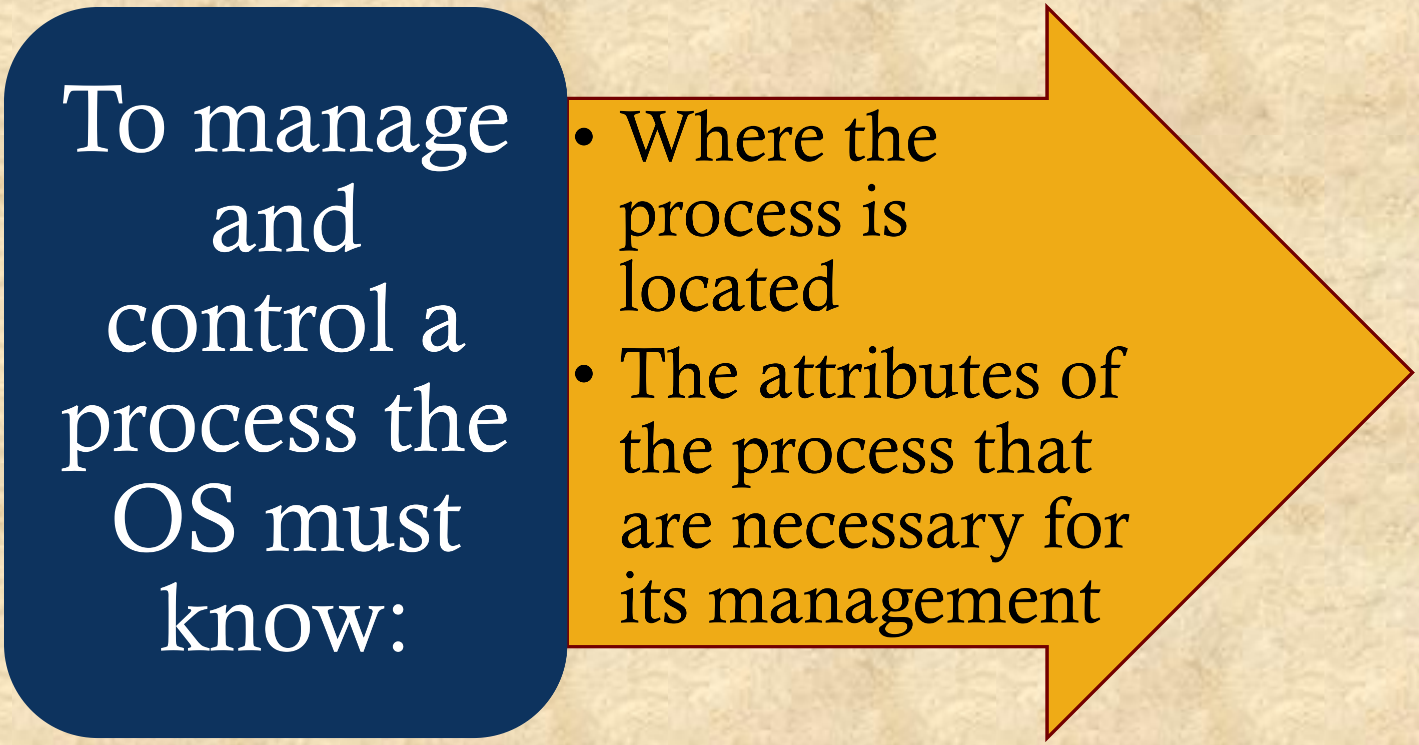
- Existence of files
- Location on secondary memory
- Current status
- Other attributes

Information may be maintained and used by a file management system

Process Tables

- Must be maintained to manage processes
- There must be some reference to memory, I/O, and files, directly or indirectly
- The tables themselves must be accessible by the OS and therefore are subject to memory management

Process Control Structures



To manage and control a process the OS must know:

- Where the process is located
- The attributes of the process that are necessary for its management

Process Control Structures

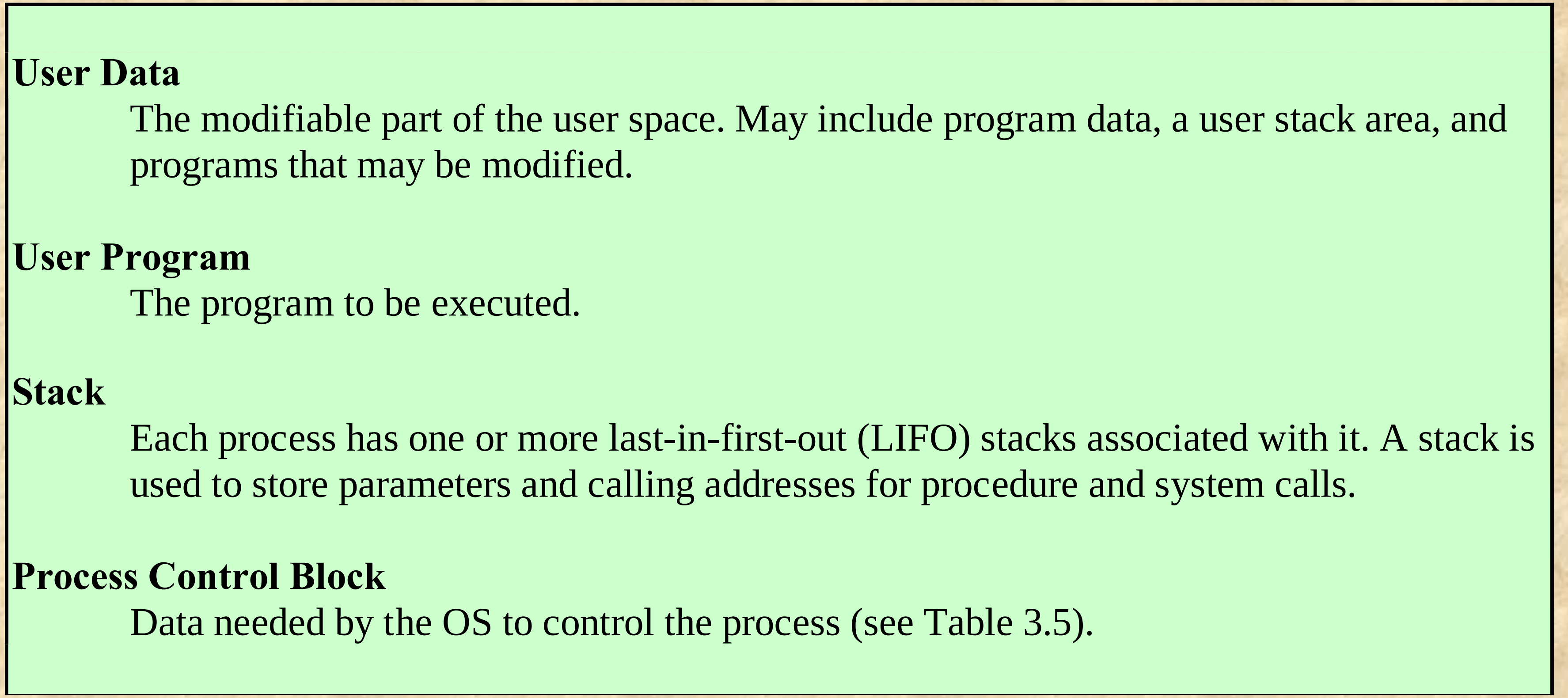
Process Location

- A process must include a program or set of programs to be executed
- A process will consist of at least sufficient memory to hold the programs and data of that process
- The execution of a program typically involves a stack that is used to keep track of procedure calls and parameter passing between procedures

Process Attributes

- Each process has associated with it a number of attributes that are used by the OS for process control (**process control block**)
- The collection of program, data, stack, and attributes is referred to as the **process image**
- Process image location will depend on the memory management scheme being used

Table 3.4
Typical Elements of a Process Image



User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

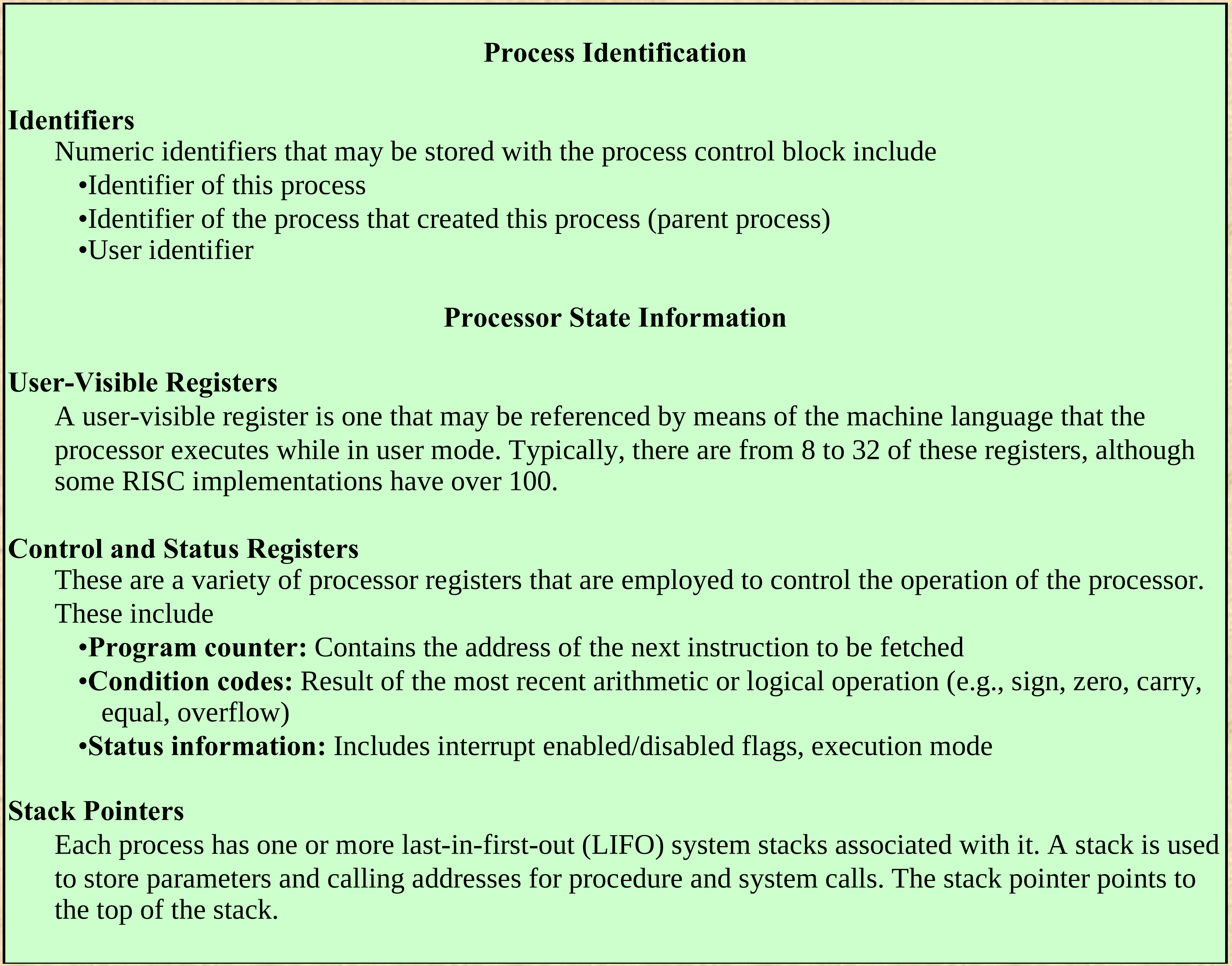
The program to be executed.

Stack

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block

Data needed by the OS to control the process (see Table 3.5).



Process Identification

Identifiers

Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include

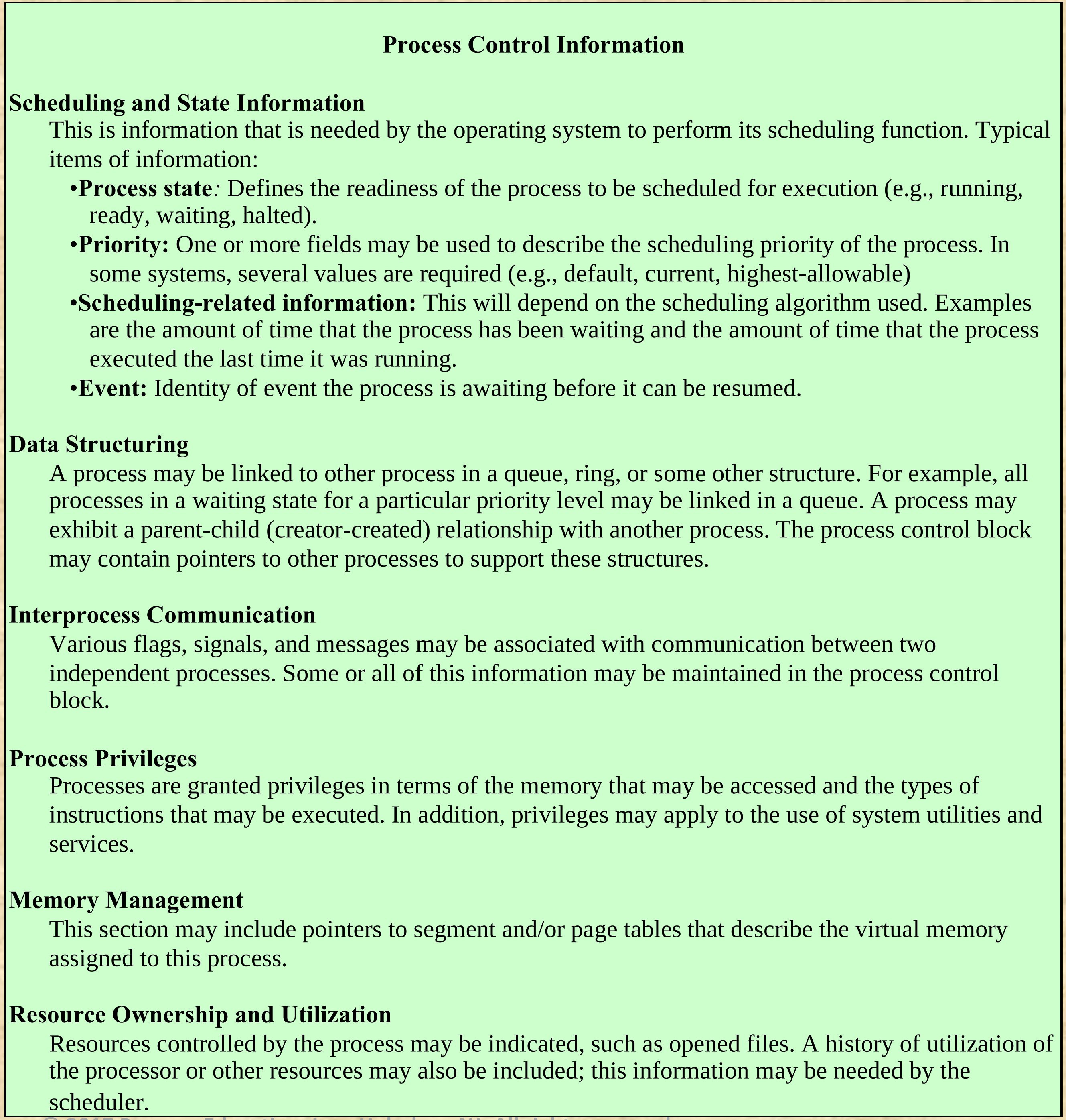
- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers

Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Table 3.5 Typical Elements of a Process Control Block (page 1 of 2)

(Table is located on page 125 in the textbook)



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.

Table 3.5 Typical Elements of a Process Control Block (page 2 of 2)

(Table is located on page 125 in the textbook)

Process Identification

- Each process is assigned a unique numeric identifier
  - Otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier
  - A process may be assigned a user identifier that indicates the user responsible for the job.
- Many of the tables controlled by the OS may use process identifiers to cross-reference process tables
- Memory tables may be organized to provide a map of main memory with an indication of which process is assigned to each region
  - Similar references will appear in I/O and file tables
- When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication
- When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process

Processor State Information

- **Processor state information** consists of the contents of processor registers. While a process is running, the information is in the registers. When a process is interrupted, all of this register information must be saved so it can be restored when the process resumes execution. The nature and number of registers involved depend on the design of the processor. Typically, the register set will include user-visible registers, control and status registers, and stack pointers (Check Table 3.5 for details)
    - Of particular note, all processor designs include a register or set of registers, often known as the program status word (PSW), that contains status information

Processor State Information

Consists of the contents of processor registers

- User-visible registers
- Control and status registers
- Stack pointers

Program status word (PSW)

- Contains condition codes (also referred to as **flags**) plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

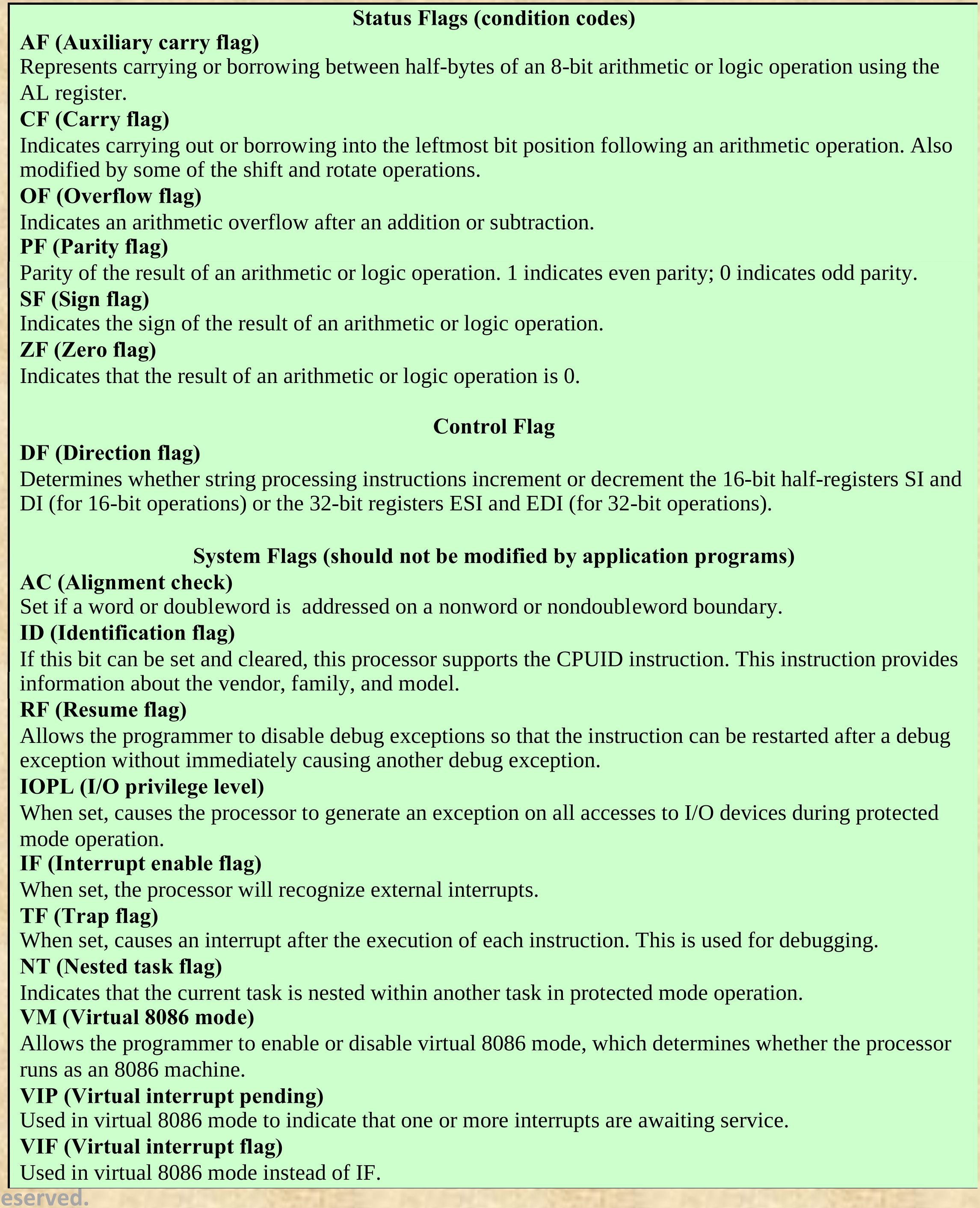
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X	ID	=	Identification flag
X	VIP	=	Virtual interrupt pending
X	VIF	=	Virtual interrupt flag
X	AC	=	Alignment check
X	VM	=	Virtual 8086 mode
X	RF	=	Resume flag
X	NT	=	Nested task flag
X	IOPL	=	I/O privilege level
S	OF	=	Overflow flag
C	DF	=	Direction flag
X	IF	=	Interrupt enable flag
X	TF	=	Trap flag
S	SF	=	Sign flag
S	ZF	=	Zero flag
S	AF	=	Auxiliary carry flag
S	PF	=	Parity flag
S	CF	=	Carry flag

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Shaded bits are reserved

Figure 3.12 x86 EFLAGS Register

Table 3.6 x86 EFLAGS Register Bits



Status Flags (condition codes)

AF (Auxiliary carry flag)
Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.

CF (Carry flag)
Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.

OF (Overflow flag)
Indicates an arithmetic overflow after an addition or subtraction.

PF (Parity flag)
Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.

SF (Sign flag)
Indicates the sign of the result of an arithmetic or logic operation.

ZF (Zero flag)
Indicates that the result of an arithmetic or logic operation is 0.

Control Flag

DF (Direction flag)
Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).

System Flags (should not be modified by application programs)

AC (Alignment check)
Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.

ID (Identification flag)
If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.

RF (Resume flag)
Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.

IOPL (I/O privilege level)
When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.

IF (Interrupt enable flag)
When set, the processor will recognize external interrupts.

TF (Trap flag)
When set, causes an interrupt after the execution of each instruction. This is used for debugging.

NT (Nested task flag)
Indicates that the current task is nested within another task in protected mode operation.

VM (Virtual 8086 mode)
Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.

VIP (Virtual interrupt pending)
Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.

VIF (Virtual interrupt flag)
Used in virtual 8086 mode instead of IF.

(Table is located on page 127 in the textbook)

Process Control Information

- The additional information needed by the OS to control and coordinate the various active processes

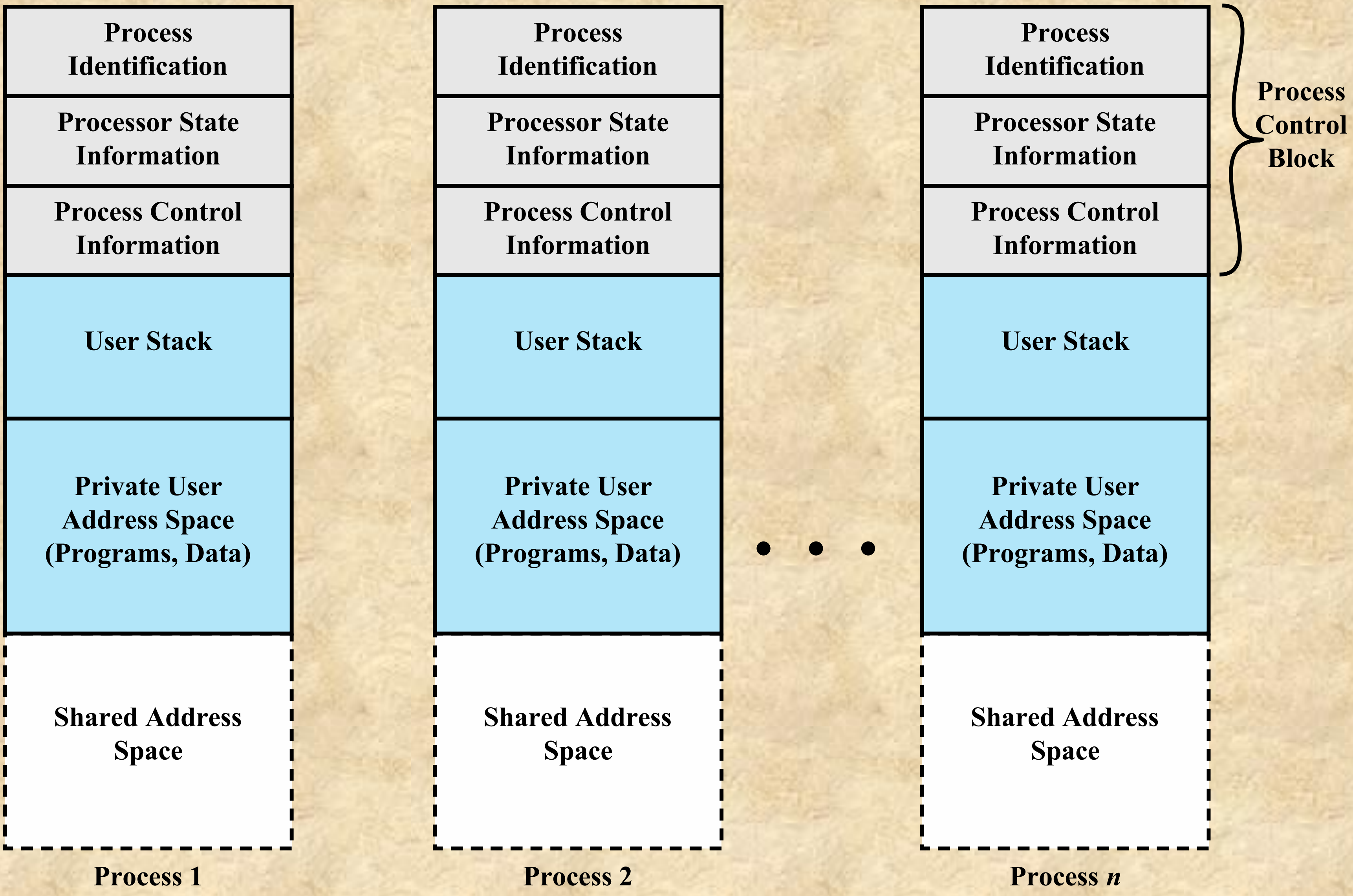

Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process 1
Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process 2
• • •
Process Identification
Processor State Information
Process Control Information
User Stack
Private User Address Space (Programs, Data)
Shared Address Space
Process *n*
Process Control Block

Figure 3.13 User Processes in Virtual Memory

The process control block may contain structuring information, including pointers that allow the linking of process control blocks. Thus, the queues that were described in the preceding section could be implemented as linked lists of process control blocks. For example, the queuing structure of Figure 3.8a could be implemented as suggested in Figure 3.14

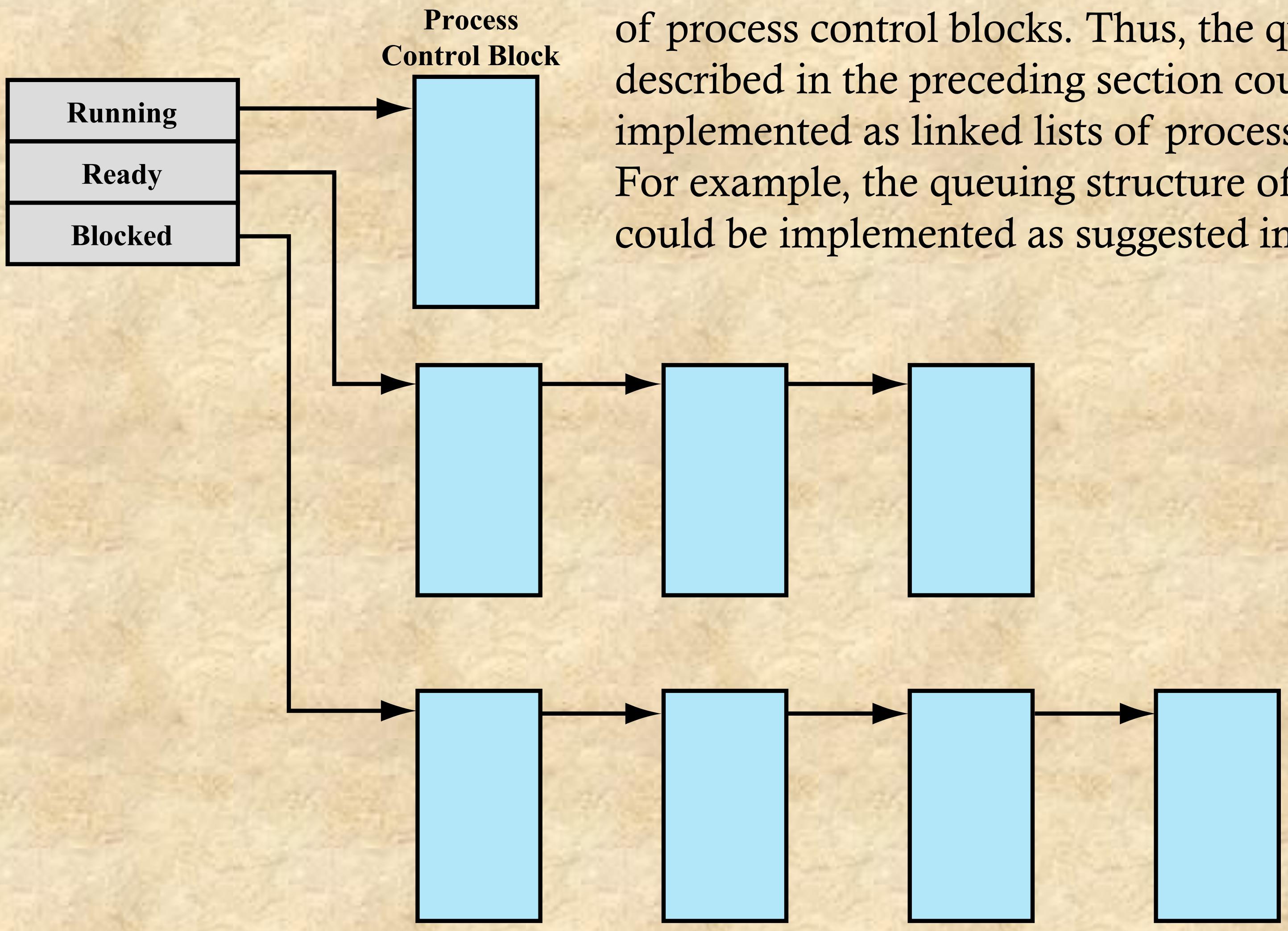


Figure 3.14 Process List Structures

Role of the Process Control Block

- The most important data structure in an OS
  - Contains all the information about a process that is needed by the OS
  - Blocks are read and/or modified by virtually every module in the OS
  - Defines the state of the OS
- Difficulty is not access, but protection
  - A bug in a single routine, such as an interrupt handler, could damage process control blocks, which could destroy the system's ability to manage the affected processes
  - A design change in the structure or semantics of the process control block could affect a number of modules in the OS

Midterm Exam
CSIS 2260-010/011/012

- Up to 50 min, in class from 10:30 AM, Feb 26 (Pacific Time)
- Blackboard online test: MCQ, true/false, filling in the blank, and short questions
- Don't be late, otherwise you may not be allowed to finish/attend
- **You must have ZOOM installed and a working camera. No camera, no exam and a ZERO mark. You will receive the ZOOM link via Email and Blackboard before the exam**
- Coverage: Introduction to Hardware, Chapters 1-3 of the textbook (till what is covered today), and labs 1-5, Open book
- 30% toward the total evaluation
- No labs in the midterm week