

2026

CS 332

Subject 3

NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details



Module 1



Job Scheduling

Job Scheduler (Long-term scheduling)

It is the procedure that determines which job (program) will be moved from secondary memory (HDD) to main memory for eventual CPU execution.

Job Scheduler deals with Job Scheduling

[Long term scheduling / scheduler]

CPU scheduling:

CPU Dispatcher (Short-term scheduling)

It is the procedure that determines which main memory-available program is to use the CPU next among the processes available in the **Ready Queue/List**.

CPU Dispatcher also known as **Dispatcher** or **CPU Scheduler** deals with CPU scheduling

[Short term scheduling / scheduler]

Process State

A process can be in any one of a number of states: **ready, running, exit, suspend**, etc



Context

All the info needed by OS to start or resume the execution of a program(chapter 2 term) or process (chapter 3 term) after it was stopped or its execution suspended

Context switch

Generic term indicating a switch in context from A to OS or OS to B. It involves a user mode to kernel mode switch as well or the other way around. Activity such as interrupt handling or dealing with multiprogramming or multitasking or timesharing involves context switching.

Swapping

Process that is not using the CPU or is blocked waiting for I/O is 'swapped' and thus moved to 'disk'. A 'swap' can be a full swap or 'partial' swap, if only a portion of the process was in main memory prior to the swap. Can be referred to as a roll-in or roll-out or swap-in (from disk to MM) or swap-out (from MM to disk).

Thrashing

I/O activity where data moves back and forth between the CPU and the disk but the CPU does not seem to perform actual work.
Useless disk-activity a result of bad program locality

*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



Process Structure Overview

1. User program/User process info (process elements)

2. Global structures of the kernel with info about ALL processes (also uses item 3 info)

3. Kernel info about a process

1+3 is known as Process Image



Process Structure Overview

1. User program/User process info (process elements)

2. Global structures of the kernel with info about all processes

3. Kernel info about a process

1+3 is known as Process Image

Process Elements

- Two essential elements of a process are:

Program code (aka text, instructions)

- which may be shared with other processes that are executing the same program

A set of data associated with that code
(init, uninit aka bss, heap, user stack, kernel stack?, shared memory data)

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

Table 3.4

Typical Elements of a Process Image

User Data	init, bss, shmem, heap including	User Stack
The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.		
User Program	text	
The program to be executed.		
Stack		User 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).		



A program (in SM) is associated with the following

- P1.a. text (i.e. instructions or code),
- P1.b. initialized data
- P1.c. uninitialized data (i.e. bss)

and provisions (size to be decided at startup time or later) for

- P1.d. heap data
- P1.e. user stack data
- P1.f. **possibly** shared (memory) data
- P1.g. kernel stack data

Note: if we use generically the term stack we still imply a user stack and a kernel stack

P1. Program (aka Process) elements (in MM)

A program that has now moved into main memory (MM) and is a process has

- P1.a. text (i.e. instructions or code),
- P1.b. initialized data
- P1.c. uninitialized data (i.e. bss)
- P1.d. heap data
- P1.e. user stack data
- P1.f. **sometimes** shared (memory) data
- P1.g. kernel stack data

Note: if we use generically the term stack we still imply a user stack and a kernel stack



Process Structure Overview

1. User program/User process info

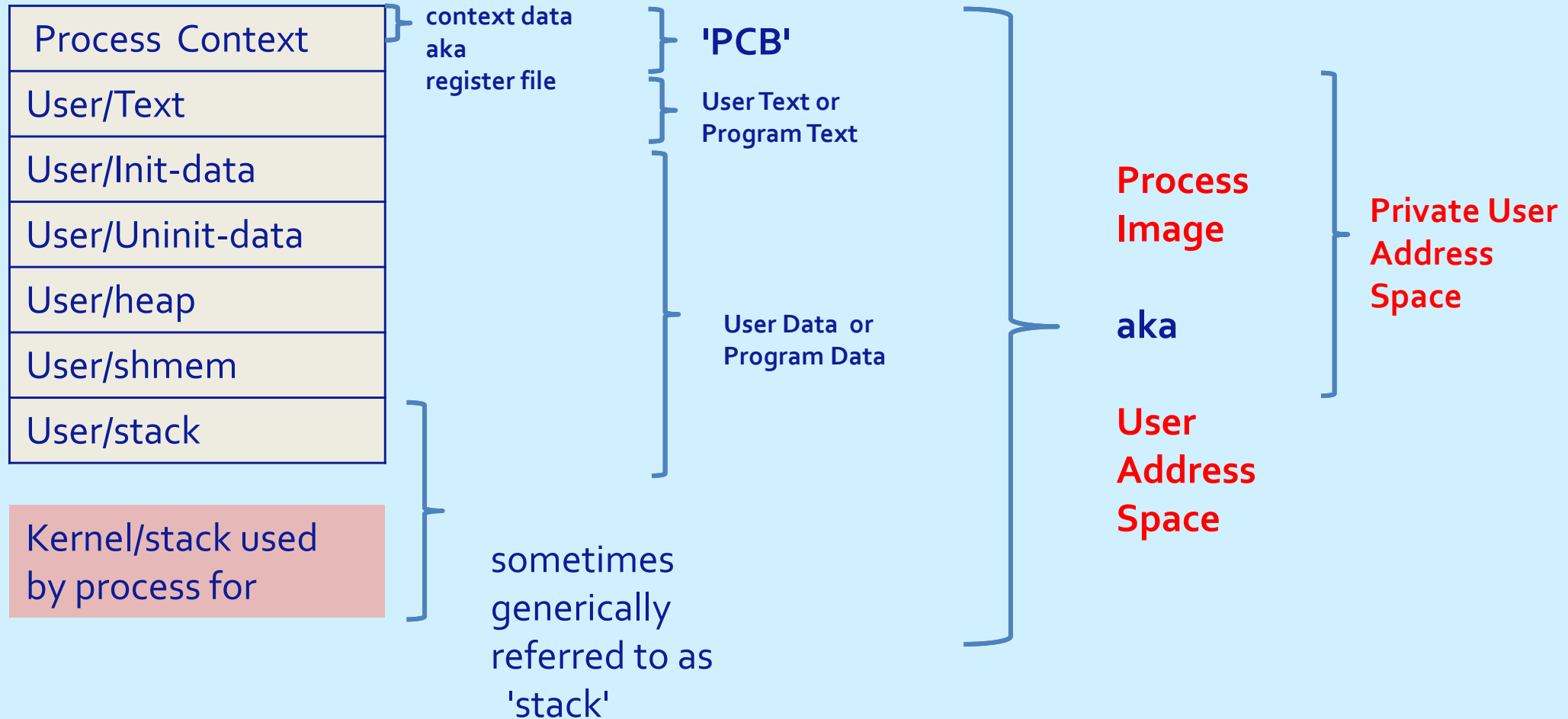
2. Global structures of the kernel with info about all processes

3. Kernel info about a process

1+3 is known as Process Image



Process Image



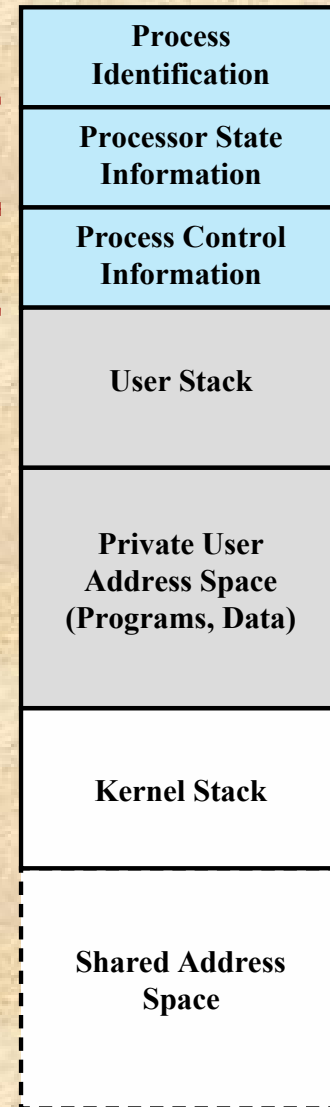
Execution Within User Processes

Process
Image

ProcessID,
UserID, GroupID
etc

Register file

Process state,
Everything
else



Process Control
Block

text,
user data

**Figure 3.16 Process Image: Operating System
Executes Within User Space**

Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

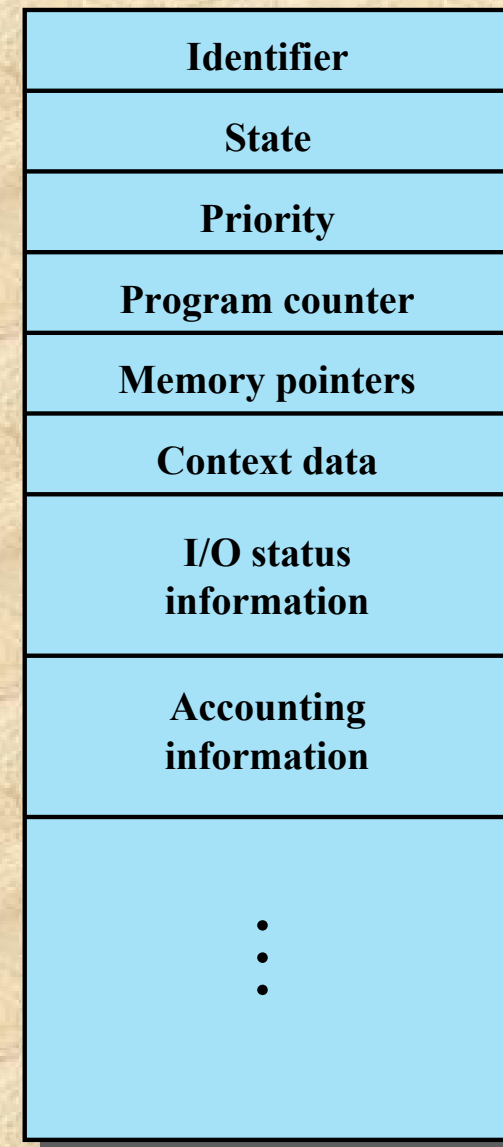


Figure 3.1 Simplified Process Control Block



B1 Process identification

PID (aka identifier), PPID, user-name, group-name

B2. Processor state information

registers, interrupt info, PSW

B3. Process Control Information

process state(running,ready,waiting,etc)

scheduling info (priority, time run, time left, etc)

event waiting, shared memory (IPC),

memory (virtual, pages), privileges, I/O etc,

path to exec on HDD

PID: Process ID , PPID : Parent Process ID



Process Structure Overview

1. User program/process elements(details)

2. Global structures of the kernel with info about all processes

3. Per process structure(s) of the kernel

1+3 is known as Process Image



A. Process List/Table (PL). [Kernel DS] An array of pointers to PCBs
(List of processes in OS that are still around)

[Typical Linux size 32768 x Now:8B, Then: 4B]

B. Process Control Block(PCB) [Kernel DS]. A DS used by the OS to manage a process. PCB is the data structure that contains info about the process (process context) associated with a program.

[Typical Linux size 7-8KiB per PCB as of 2022]

C. Memory Tables (MT) [Kernel DS],

D. File Tables(FT) [Kernel DS],

E. I/O (Device) Tables (DT) [Kernel DS]

D. The 'Four' pointers: PL,MT,FT, and DT. PL points to PCBs

Note: MT, FT, DT contains information extracted from the PCB of ALL processes.

Note:Linux calls PCB a task_struct. In 1999 its precursors was ~1,000B and in 2022 it is ~7,000B.

Process Tables / List

- 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

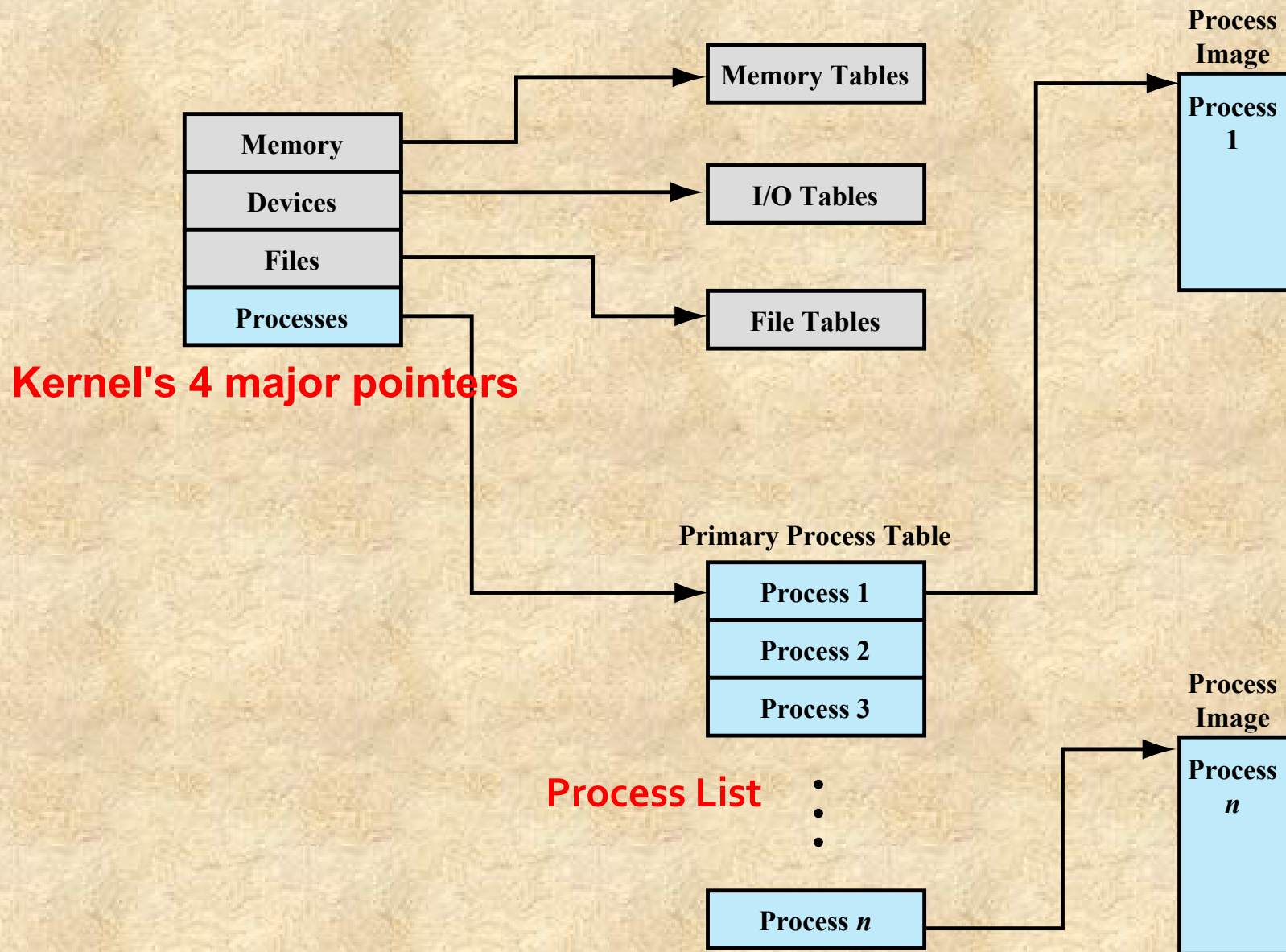


Figure 3.11 General Structure of Operating System Control Tables

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

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
 - In which case the OS has little or no knowledge of files
 - In other operating systems, much of the detail of file management is managed by the OS itself

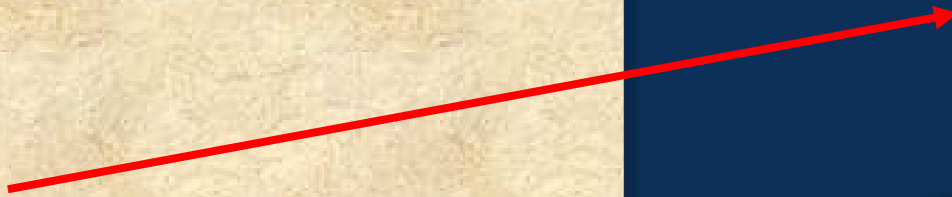
I/O (Device) Tables

- 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

DMA





Process Structure Overview

1. User program/process elements(details)
 2. Global structures of the kernel with info about all processes
 3. Per process structure(s) of the kernel
- 1+3 is known as Process Image**

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

Execution Within User Processes

Process
Image

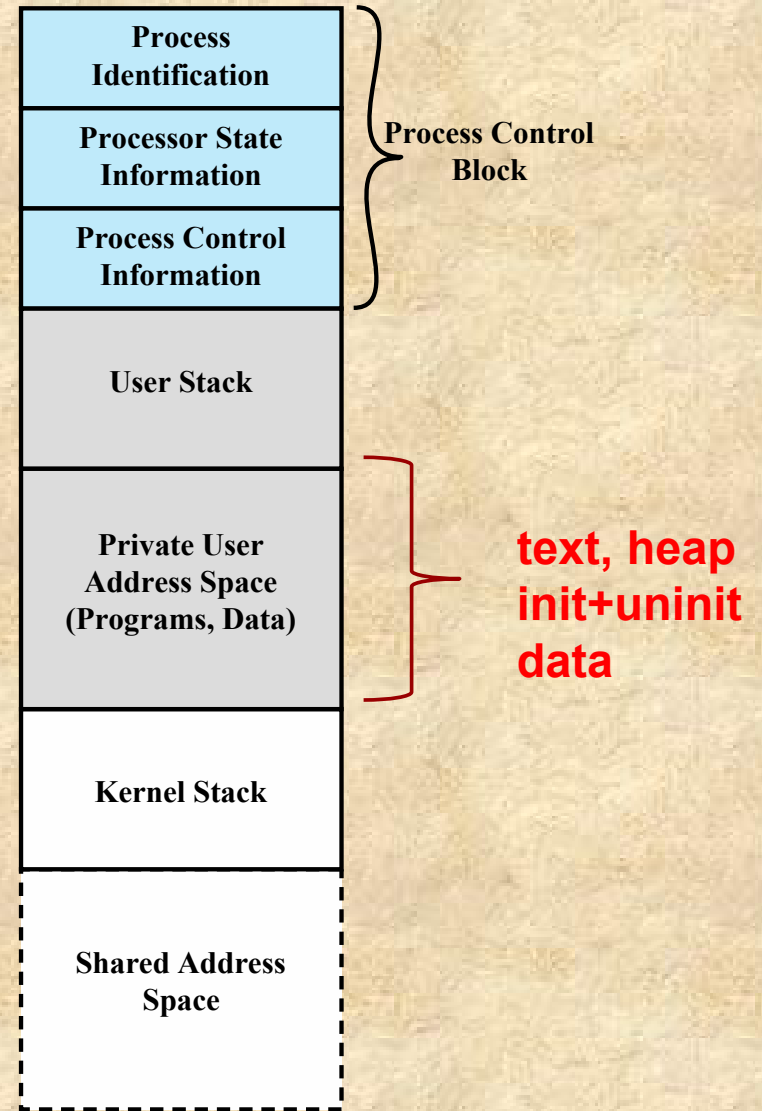


Figure 3.16 Process Image: Operating System Executes Within User Space

B1

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
- 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



B1 Process identification

PID , PPID, user-name, group-name

B2. Processor state information

registers, interrupt info, PSW

B3. Process Control Information

process state(running,ready,waiting,etc)

scheduling info (priority, time run, time left, etc)

event waiting, shared memory (IPC),

memory (virtual, pages), privileges, I/O etc,

path to exec on HDD

B1

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

B2

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.

Process Identification

Processor State Information

Table 3.5 Typical Elements of a Process Control Block

(page 1 of 2)

(Table is located
on page 125 in the
textbook)

B2

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 plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

B2

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	I D	V I P	V I F	A C	V M	R F	0	N T	I O P L	O F	D F	I F	T F	S F	Z F	0	A F	0	P F	1	C F	

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

More detail than
 Chapter 1/Subject 1

Figure 3.12 x86 EFLAGS Register

B3

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
- 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

B3

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 Execution

Process States

CPU Dispatcher,
CPU Scheduler
Short term
scheduler

aka

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

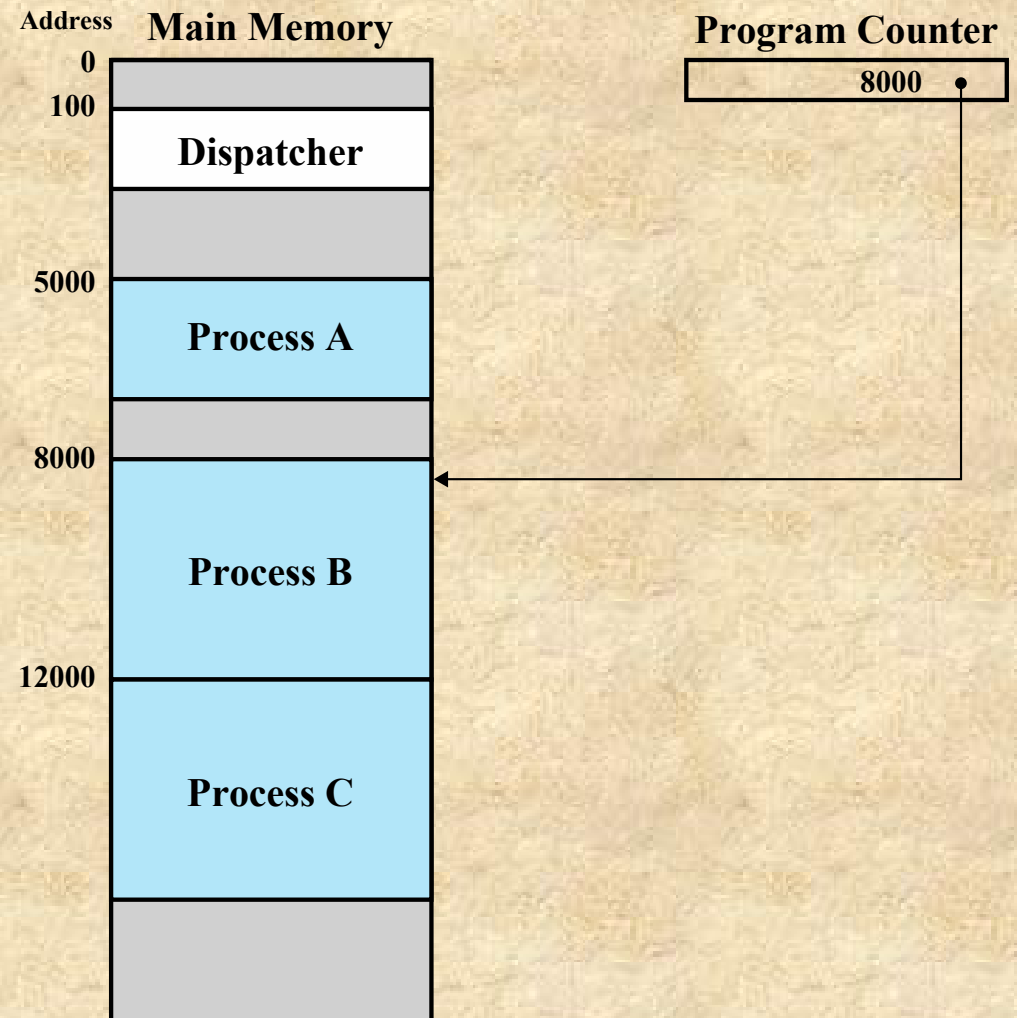


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

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

Typo in the book (page 107)

10]103 instead of 10 103
11]104 instead of 11 104

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

Practice 1



- 1.1. The OS can access information about processes using 4 pointers. Name them and describe them
- 1.2. Who can modify the PCB? User or Kernel?
- 1.3. Why do i use the term kernel in 1.2 (answer two of two)?
- 1.4. Who can modify user data ? User or Kernel?
- 1.5. Based on slides 1-29, how many processes does a Linux system can have? Justify your answer based on the slides (URLs do not count, slide pages do!)



Module 2



Process Creation Process Termination and other Process States... and ... interactions

Role of the Process Control Block

task struct in linux

- The most important data structure in an OS
 - Contains all of 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 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

Modes of Execution

User Mode

- Less-privileged mode
- User programs typically execute in this mode

System Mode aka

Kernel Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

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

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

Process Creation

- Once the OS decides to create a new process it:

Assigns a unique process identifier to the new process



Allocates space for the process



Initializes the process control block



Sets the appropriate linkages



Creates or expands other data structures

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

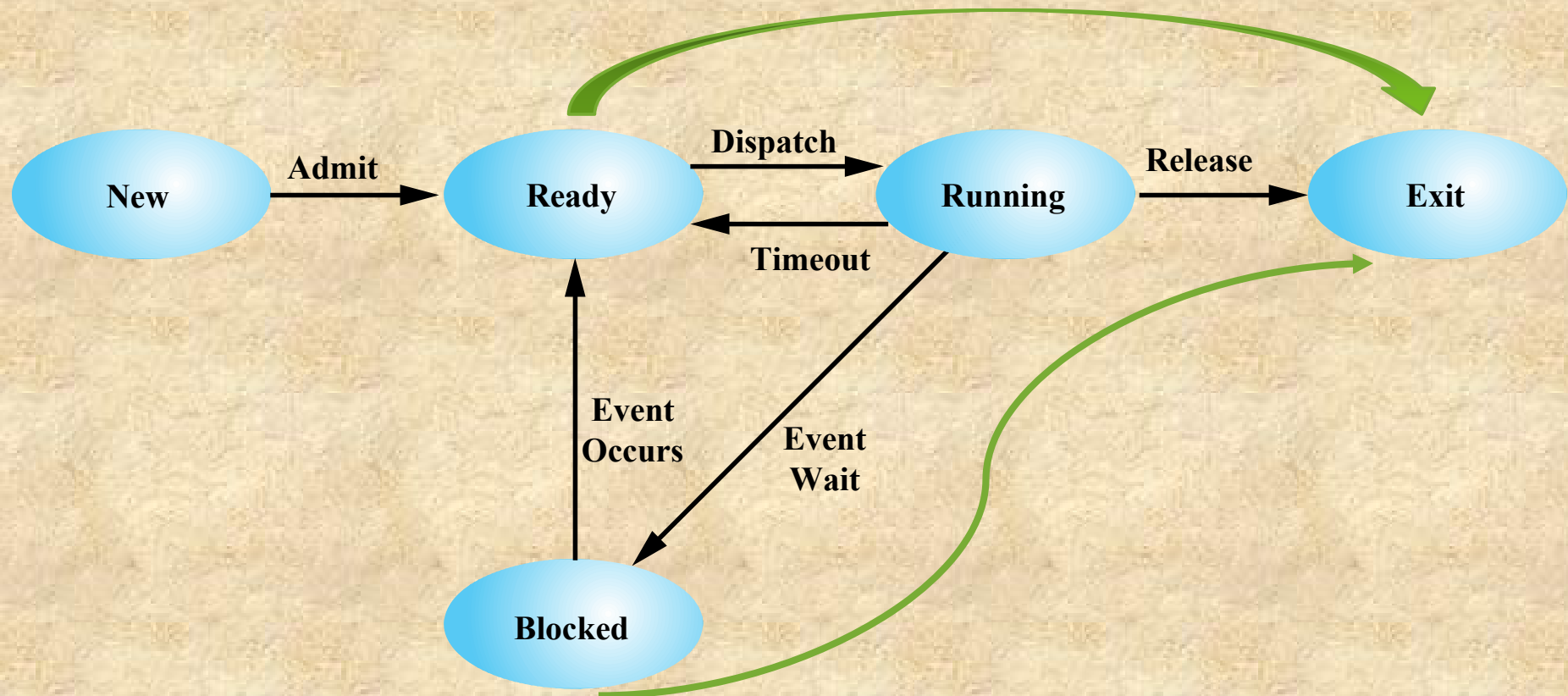
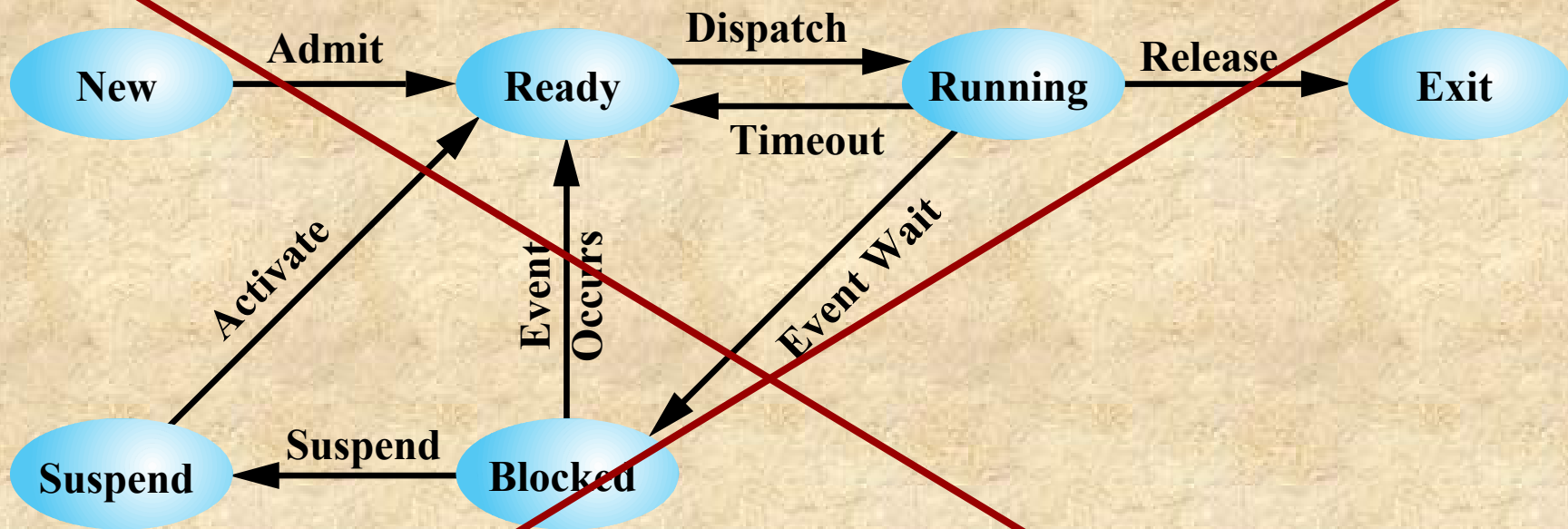


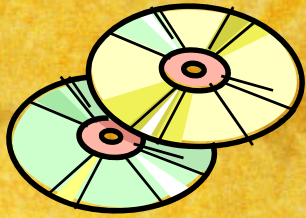
Figure 3.6 Five-State Process Model



(a) With One Suspend State

move on to the next
7state model/diagram

Figure 3.9 Process State Transition Diagram with Suspend States



Suspended Processes

■ Swapping

A blocked process is still using the MM

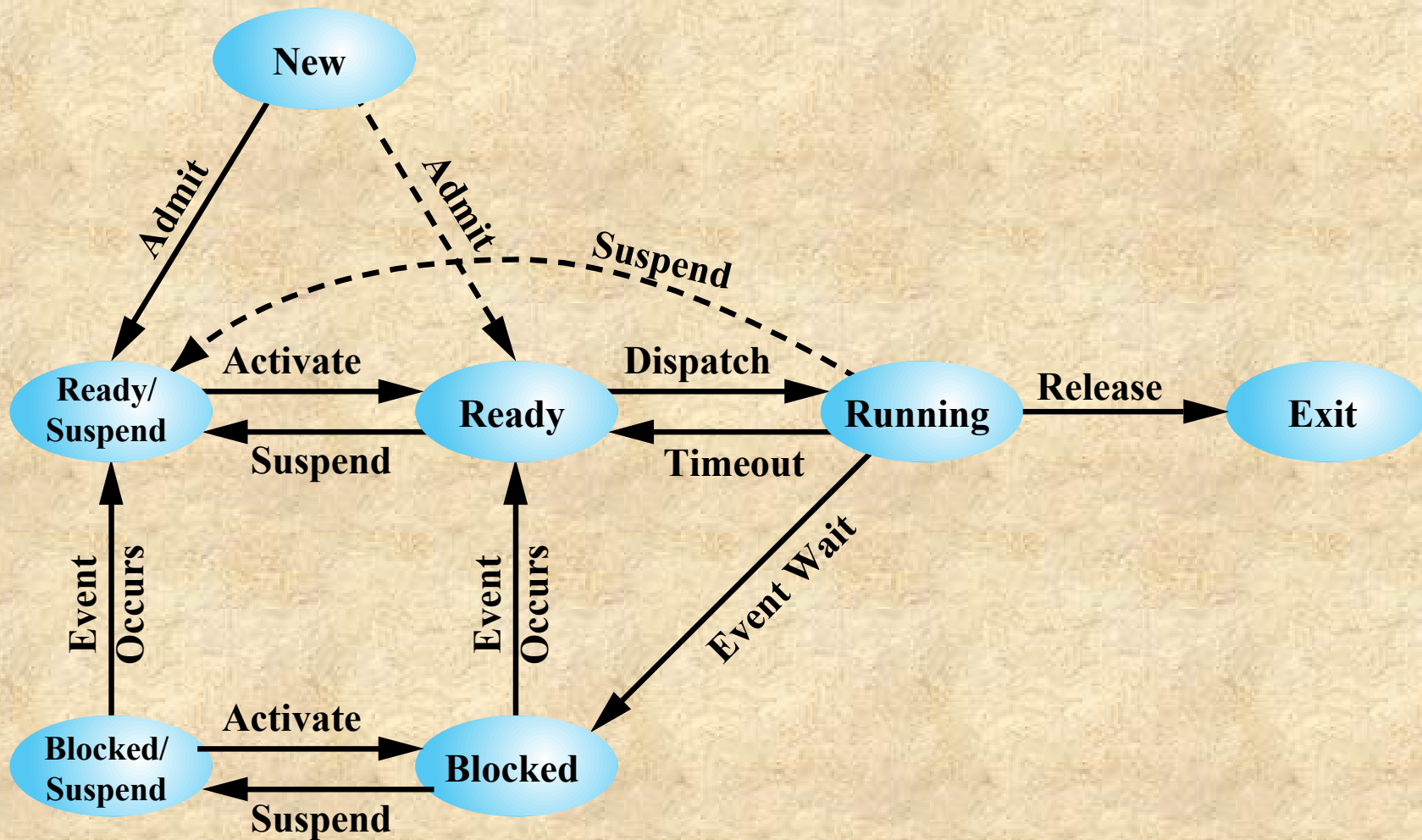
- Involves moving part of all of a process from main memory to disk
- When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue
 - This is a queue of existing processes that have been temporarily kicked out of main memory, 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
- 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

Characteristics of a Suspended Process

- The process is not immediately available for execution
- The process may or may not be waiting on an event
- 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.



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

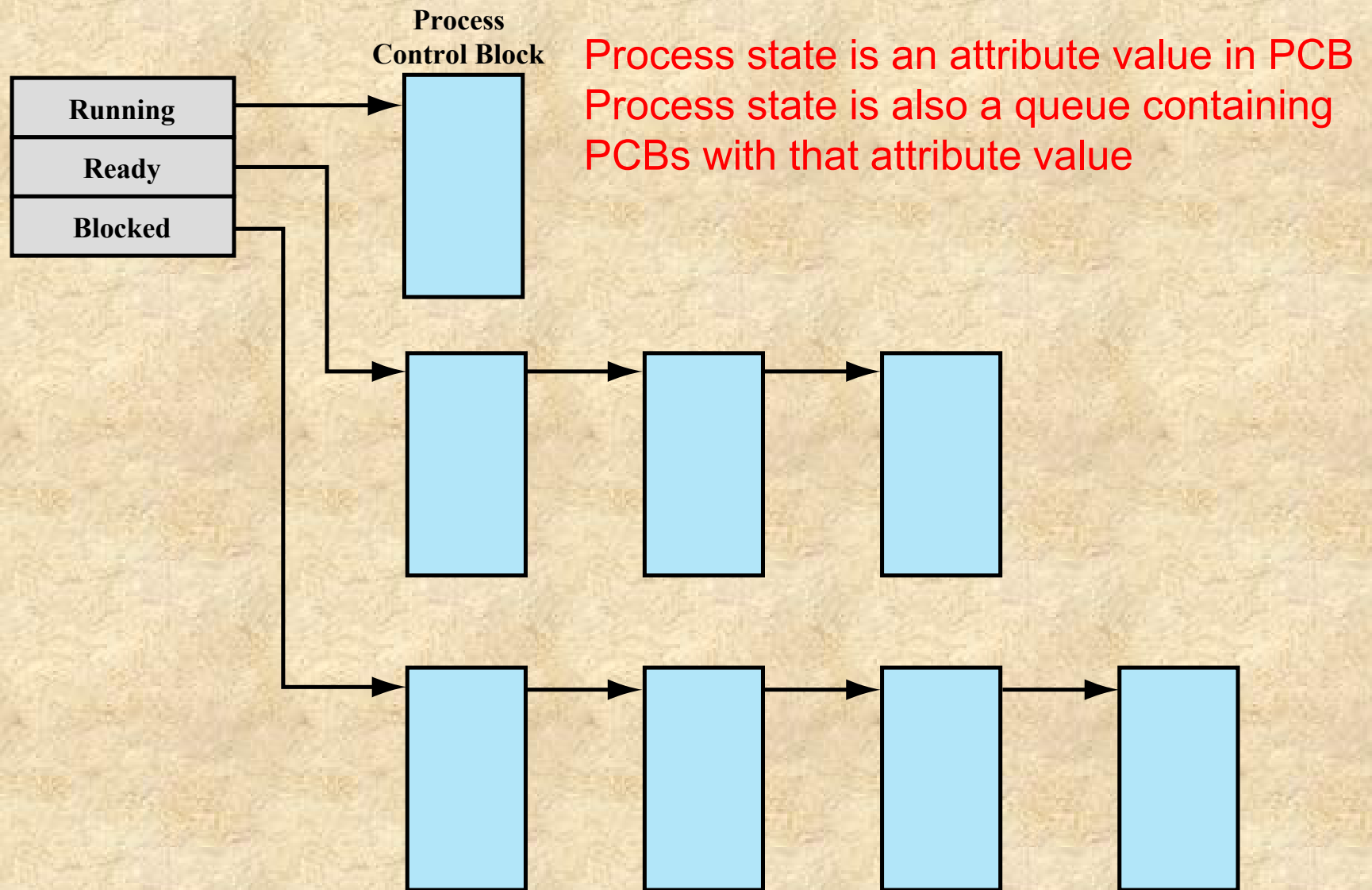


Figure 3.14 Process List Structures



Process Interruption...

Process context switching

A--> OS interrupt handling --> ?

A--> OS --> B

Table 3.8

Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap / Fault	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function



Interrupt External to the execution of current instruction

Asynchronous hardware instigated or
Hardware failure or
Clock or Timer
[Afterwards: **exec next (user) instruction ...**
unless
...
interrupted again]

**Trap or
Fault/Abort** Related to execution of current instruction.

Fault: Soft-based eg division by zero exception
Fault: Page-Fault (Chapter 8) virtual memory
Fault: to handler and then **re-exec CURRENT (user) instruction or abort**
Trap: Control to OS handler(user to kernel mode)
Trap: Then **continue with execing NEXT (user) instruction.**
Abort: Irrecoverable fault
Abort: to (maybe) handler and then **exit**

**Supervisor
call (Trap)** Related to execution of current instruction that explicitly generates it; a trap is subsequently generated

Calling a system function call ; it is a **Trap** (see above)
(eg Subject 2 : fopen)

Note: Sometimes collectively Trap, Fault/Aborts are referred to as exceptions. The term exception might include also interrupts.

System Interrupts

Interrupt

- Due to some sort of event that is external to and independent of the currently running process
 - Clock interrupt
 - I/O interrupt
 - Memory fault
- Time slice
 - The maximum amount of time that a process can execute before being interrupted

Trap

- An error or exception condition generated within the currently running process
- OS determines if the condition is fatal
 - Moved to the Exit state and a process switch occurs
 - Action will depend on the nature of the error the design of the OS

Mode Switching

If no interrupts are pending the processor:



Proceeds to the fetch stage and fetches the next instruction of the current program in the current process

If an interrupt is pending the processor:



Sets the program counter to the starting address of an interrupt handler program



Switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

Change of Process State

- The steps in a full process switch are:

A

Save the context of the processor

OS

Update the process control block of the process currently in the Running state

OS

Move the process control block of this process to the appropriate queue

OS

If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

Select another process for execution

OS

Restore the context of the processor to that which existed at the time the selected process was last switched out

B

OS

Update memory management data structures

OS

Update the process control block of the process selected



- 2.1. Who generates an interrupt?
- 2.2. Who generates a trap (eg a context switch request)?
Ans: A user program requesting say the execution of a system function call (eg printf) asking the kernel to handle it (in kernel mode)
- 2.3. Who generates a signal?
Ans: A user program(?) issuing CTRL-C that will be dealt with by the kernel in ... mode
- 2.4. Hardware or Software based ? Assign a label to Interrupt, Trap, Signal.
- 2.5. Is a swapped process always suspended?
- 2.6. Is a suspended process always swapped?
- 2.7. Why would we suspend a newly created process?
- 2.8. How many context switches on the previous slide? (A to B)



Module 3



fork(): To be discussed

vmfork(): 'optimized fork' , obsolete/not used

clone(): Linux generic call

A linux task is a **process or thread**

A **full clone** is a **fork** (process creation)

A limited clone is used for **thread (lightweight process) creation**

Linux "clone"()

1. If full_clone

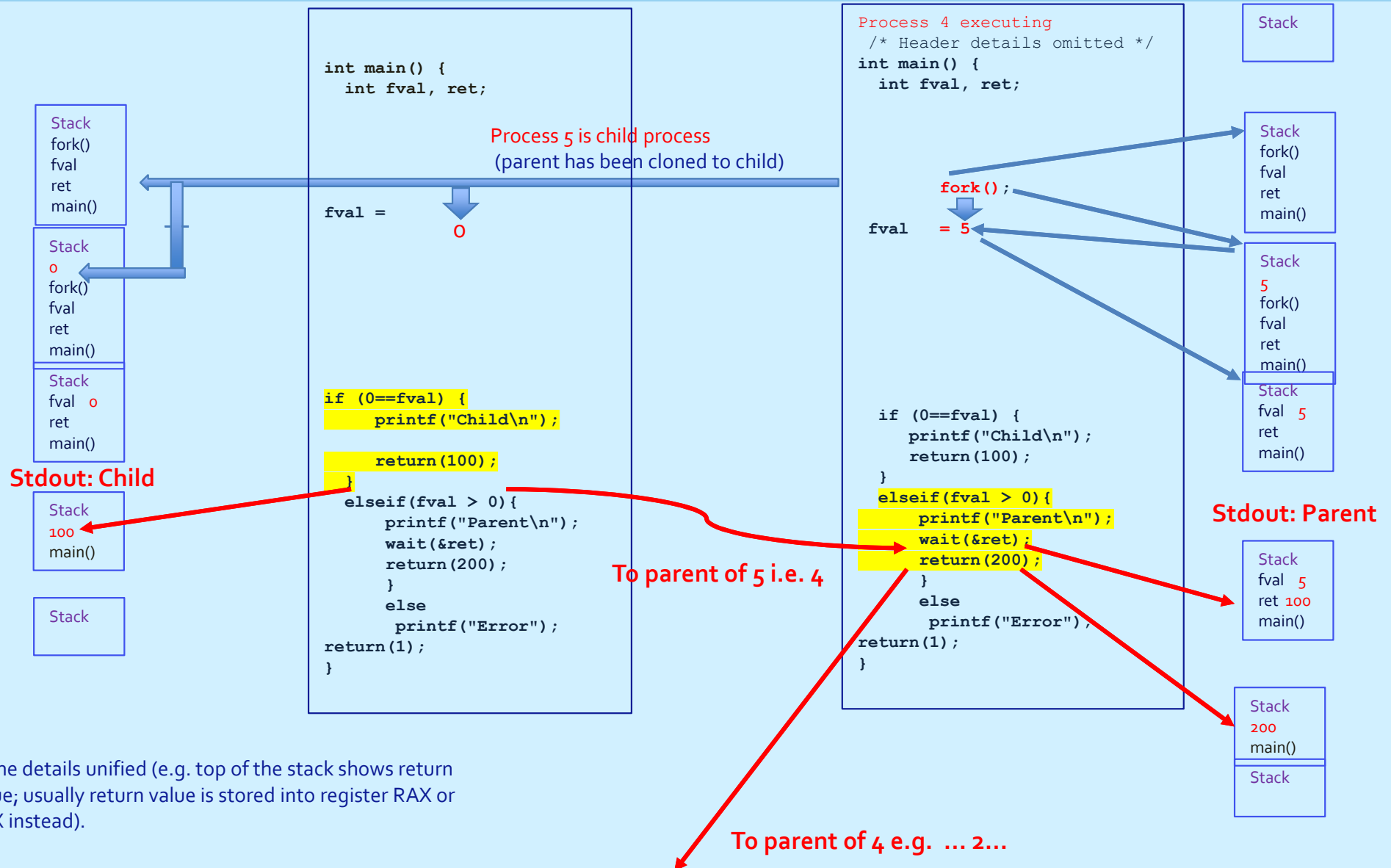
2. /* Do a full clone i.e. a fork to create a process */
 /* All of ~7000B of task_struct updateable */

3. else if limited_clone

4. /* Do a limited clone to create a thread */
 /* A small subset (may be 256-1024B) of task_struct updateable */



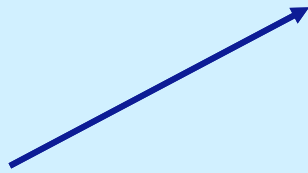
Exec a fork : summary



Execing a fork: more detail



PL
0: not NULL
1: not NULL
2: not NULL
3: not NULL
4: not NULL
5: NULL
6: not NULL
7: etc



Process Image

PCB

PID: 4
PPID: 2
USER: Tom

state: RUNNING
PC: fval=
'Stack'
fork()
fval: -
Ret: -
main()

Process 4 executing

```
/* Header details omitted */  
int main() {  
    int fval, ret;  
    fval = fork();  
    if (0==fval) {  
        printf("Child\n");  
        return(100);  
    }  
    else if (fval > 0){  
        printf("Parent\n");  
        wait(&ret);  
        return(200);  
    }  
    else  
        printf("Error");  
    return(1);  
}
```

Executing a fork: start

```
PL
0: not NULL
1: not NULL
2: not NULL
3: not NULL
4: not NULL
5: NULL
6: not NULL
7: etc
```

Note:
 In Linux the return value
 is returned through
 register RAX (or EAX ...)
 Rather than the 'top' of
 the stack. We economize
 Space in showing it this
 way

Process Image
 PCB
 PID: 4
 PPID: 2
 User: Tom
 STATE: RUNNING
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 2000
 fval:
 ret:
 main() 
 Kernel Stack 3000
 5? (in fact RAX)
 fork()
 Text & Data 11000

Process 4 executing
 /* Header details omitted */
int main() {
 int fval, ret;
 fval = **fork**();
 if (0==fval) {
 printf("Child\n");
 return(100);
 }
 else if (fval > 0){
 printf("Parent\n");
 wait(&ret);
 return(200);
 }
 else
 printf("Error");
return(1);
}

Executing a fork: cloning

```
PL
0: not NULL
1: not NULL
2: not NULL
3: not NULL
4: not NULL
5: NULL
6: not NULL
7: etc
```

Process Image

```
PCB
PID: 4
PPID:2
User:Tom
STATE: RUNNING
PC: fval=
file: stdin
file: stdout
file: stderr
User Stack 2000
fval:
ret:
main()
Kernel Stack 3000
5?
fork()
Text & Data 11000
```

...

Process Image

```
PCB
PID: 4
PPID:2
User:Tom
STATE: RUNNING
PC: fval=
file: stdin
file: stdout
file: stderr
User Stack 2000
fval:
ret:
main()
Kernel Stack 3000
5?
fork()
Text & Data 11000
... ..
```

clone first PCB of
process 4 (red
lines)

Process 4 executing

```
/* Header details omitted */
int main() {
    int fval, ret;
    fval = fork();
    if (0==fval) {
        printf("Child\n");
        return(100);
    }
    else if (fval > 0){
        printf("Parent\n");
        wait(&ret);
        return(200);
    }
    else
        printf("Error");
return(1);
}
```

Executing a fork: NEW

PL
 0: not NULL
 1: not NULL
 2: not NULL
 3: not NULL
 4: not NULL
 5: not NULL
 6: not NULL
 7: etc

Process Image

PCB
 PID: 5
 PPID:4
 User:Tom
 STATE: NEW
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 92000
fval:
ret:
main()
 Kernel Stack 83000
 ?
fork()
 Text & Data 121000

...

Process Image

PCB
 PID: 4
 PPID:2
 User:Tom
 STATE: RUNNING
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 2000
 fval:
 ret:
 main()
 Kernel Stack 3000
 ?
 fork()
 Text & Data 11000

FT as in FILE TABLE

filename	count	processes
stdin	2	2, 4
stdout	2	2, 4
stderr	2	2, 4

then adjust cloned copy and original
 and allocate space for process image
 (lines in italics)

Executing a fork: populate child



PL
 0: not NULL
 1: not NULL
 2: not NULL
 3: not NULL
 4: not NULL
 5: not NULL
 6: not NULL
 7: etc

Process Image

PCB

PID: **5**
 PPID: **4**
 STATE: **NEW**
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 92000
 fval:
 ret:
 main()
 Kernel Stack 83000
 0
 fork()
 Text & Data 121000

Process Image

PCB

PID: 4
 PPID: 2
 STATE: RUNNING
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 2000
 fval:
 ret:
 main()
 Kernel Stack 3000
 5
 fork()
 Text & Data 11000

update PCB of 5

FT as in FILE TABLE

filename	count	processes
stdin	2	2, 4
stdout	2	2, 4
stderr	2	2, 4

Not everything is ready yet ...



Executing a fork: PL, FT, ... adjust

PL
 0: not NULL
 1: not NULL
 2: not NULL
 3: not NULL
 4: not NULL
 5: not NULL
 6: not NULL
 7: etc

Process Image

PCB

PID: 4
 PPID: 2
 STATE: RUNNING
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 2000
 5

fval:
 ret:
 main()

Text&Data 11000

Process Image

PCB

PID: 5
 PPID: 4
 STATE: READY
 PC: fval=
 file: stdin
 file: stdout
 file: stderr
 User Stack 92000
 0

fval:
 ret:
 main()

Text & Data 121000

FT as in FILE TABLE

filename	count	processes
stdin	3	2,4,5
stdout	3	2,4,5
stderr	3	2,4,5

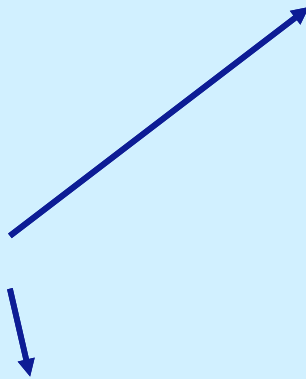
and also global
 file tables eg PL,
 FT, ...
 when done PID:5
 is READY!

MT, DT also
 updated
 accordingly
 re PID:5



Executing a fork: PL, FT, ... adjust

PL
 0: not NULL
 1: not NULL
 2: not NULL
 3: not NULL
 4: not NULL
 5: not NULL
 6: not NULL
 7: etc



Process Image
 PCB
 PID: 4
 PPID: 2
 STATE: RUNNING
 PC: **if**
 file: stdin
 file: stdout
 file: stderr
 User Stack 2000
 fval: 5
 ret:
 main()
 KernelStack 3000
 Text&Data 11000

FT as in FILE TABLE

filename	count	processes
stdin	3	2, 4, 5
stdout	3	2, 4, 5
stderr	3	2, 4, 5

Process Image
 PCB
 PID: **5**
 PPID: **4**
 STATE: **READY**
 PC: **fval=**
 file: stdin
 file: stdout
 file: stderr
 User Stack 92000
 0
 fval:
 ret:
 main()
 Text & Data 121000

Observe that Process 4 has executed the fval= instruction and PC points to the next instruction (the if statement). Process 4 is RUNNING (i.e. in RUNNING state), whereas Process 5 is still waiting and in READY state (the return value of the fork() is still at the top of the User stack and not assigned yet to be the value of variable fval).

Process Control (fork)

- Process creation is by means of the kernel system call, **fork()**

- When a process issues a fork request, the OS performs the following functions:

1

- **Allocates a slot in the process table for the new process**

2

- Assigns a unique process ID to the child process

3

- Makes a copy of the process image of the parent, with the exception of any shared memory

4

- Increments counters for any files owned by the parent, to reflect that an additional process now also owns those files

5

- Assigns the child process to the Ready to Run state

6

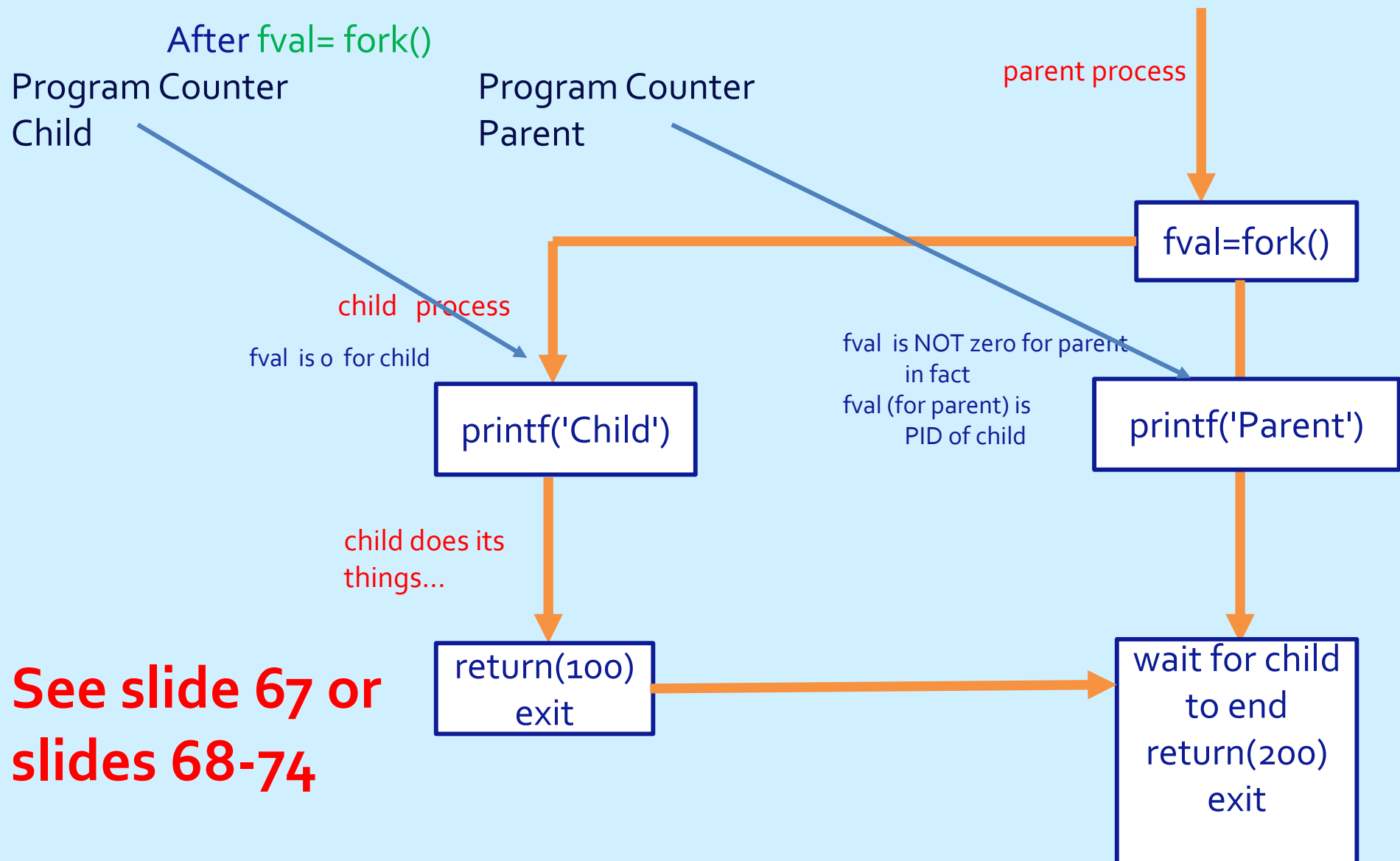
- **Returns the ID number of the child to the parent process, and a 0 value to the child process**

After Creation

- After creating the process the Kernel can do one of the following, as part of the dispatcher routine:
 - Stay in the parent process. Control returns to user mode at the point of the fork call of the parent.
 - Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
 - Transfer control to another process. Both parent and child are left in the Ready to Run state.



Fork mechanism



**See slide 67 or
slides 68-74**



fork1.c and fork2.c

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    fork();
    printf("Hello world!\n");
    return(0); /* all is good*/
}
```

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    int fval;
    fval = fork();
    if ( 0 == fval ) {
        printf("Child\n");
        return(100);
    }
    else {
        printf("Parent\n");
        return(200);
    }
    return(1); /* error? */
}
```

What is the return
value of
fork2.c

(a) for parent
process?

(b) for child process?

In unix/linux

fork() **returns 0** to child process and captured in fval of fork2.c

fork() **returns child's PID** to parent process and captured in fval

fork() **returns a negative number** for error and not captured

Question: Is fork1 robust? Is fork2 robust?

fork3.c



```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
    int fval, stat, rpic;
    fval = fork();
    if ( fval < 0 ) {                                /* Error captured */
        printf("Fork failed on error %d\n", fval);
        return(fval);
    }
    else {
        if ( 0 == fval ) {                            /* Child */
            printf("Child: (fval,pid,ppid)=(%5d,%5d,%5d)\n", fval, getpid(), getppid());
            return(100);
        }
        else {                                        /* Parent */
            printf("Parent: (fval,pid,ppid)=(%5d,%5d,%5d)\n", fval, getpid(), getppid());
            rpid=wait( &stat );
            printf("Child %d returns %6d captured by  stat at the parent\n",
                    rpid, WEXITSTATUS(stat));
            return(200);
        }
    }
    return(0); /* Is it needed ? */
}
```

fork3.c running



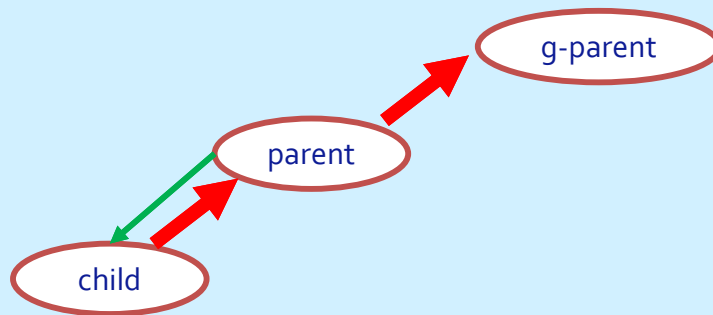
```
% gcc fork3.c -o fork3
```

```
% ./fork3
```

```
Parent: (fval,pid,ppid)=(24974,24973,29022)
```

```
Child: (fval,pid,ppid)=(0,24974,24973)
```

```
Child 24974 returns 200 captured by stat at  
the parent
```



pid : process ID

ppid : parent process's ID

fork4.c



```

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
    /* Error checking for fval removed */
int  forkf(void) {
    int fval, stat, rpid;
    fval = fork();
    if ( 0 == fval ) {
        printf("Child: (fval,pid,ppid)=(%5d,%5d,%5d)\n",fval,getpid(),getppid());
        return(100);
    }
    else {
        printf("Parent: (fval,pid,ppid)=(%5d,%5d,%5d)\n",fval,getpid(),getppid());
        rpid=wait( &stat );
        printf("Child %d returns %6d captured by  stat at the parent\n",
                rpid, WEXITSTATUS(stat));
        return(200);
    }
    return(0);
}

int main(void) {
    int mval;
    mval=forkf();
    printf("Main: (mval,pid)=(%6d,%6d)\n",mval,getpid());
    return(mval+1);
}

```

Value returned by child is

WEXITSTATUS(stat)

not

stat

fork4.c running



```
% gcc fork4.c -o fork4
% ./fork4
Parent: (fval,pid,ppid)=(27740,27739,29022)
Child:  (fval,pid,ppid)=(    0,27740,27739)
Main: (mval,pid)=(    100, 27740)
Child 27740 returns    101 captured by  stat at the parent
Main: (mval,pid)=(    200, 27739)
```

Questions:

1. Who finishes first?
2. What does it return?
3. Who does it return to?
4. What is the return value of the child proc.?
5. What does parent receive from child on wait?
6. What is the return value of the parent proc.?
7. Who owns (aka is guardian of) the child?
Hint: the one who waits for it!



More examples

Fork 6 through Fork 8

THIS SLIDE DOES NOT EXIST
(DO NOT TRY IT ON AFS)

```
while (1) { fork(); }
```



aka

'Blame the parent'

ORPHAN process is a process for which the parent has already terminated **without bothering to wait** for its child. The OS inherits such a process (and thus the PPID of the orphan process becomes 1), and eventually terminates (with a new 'guardian' parent).

```
% gcc orphan.c -o orphan
% ./orphan
Parent:(fval,pid,ppid)=(29038,29037,29022)
Child: (fval,pid,ppid)=( 0,29038,29037)
Main:(mval,pid)=( 200, 29037)
```

```
completiontime(orphan) >
completiontime(parent)
```

i.e. parent terminates first.

after 13 seconds

```
Child: (fval,pid,ppid)=( 0,29038, 1 )
Main:(mval,pid)=( 100, 29038)
```



aka

'Blame you know who'

ZOMBIE process is/describes a child process between the child's termination and the time the parent reaches the wait statement that processes the return value of the child (at that point the process ceases to be a zombie; if the parent's code does not include a wait statement the zombie process stays zombie for ever...i.e. it still occupies a PCB)

```
completiontime(zombie) <
completiontime(parent)
```

i.e zombie finishes first, return value is still to be 'reaped'/processed. Eventually parent may reap it (and PCB is released) or if not, PID is still being used and ZOMBIE lingers

```
% ps -ef
```

```
alexg  29022 28989 0 11:56 pts/3    00:00:00 -bash
alexg  29912 29022 0 16:54 pts/3    00:00:00 ./zombie
alexg  29913 29912 0 16:54 pts/3    00:00:00 [zombie] <defunct>
```

```
% gcc zombie.c -o zombie
```

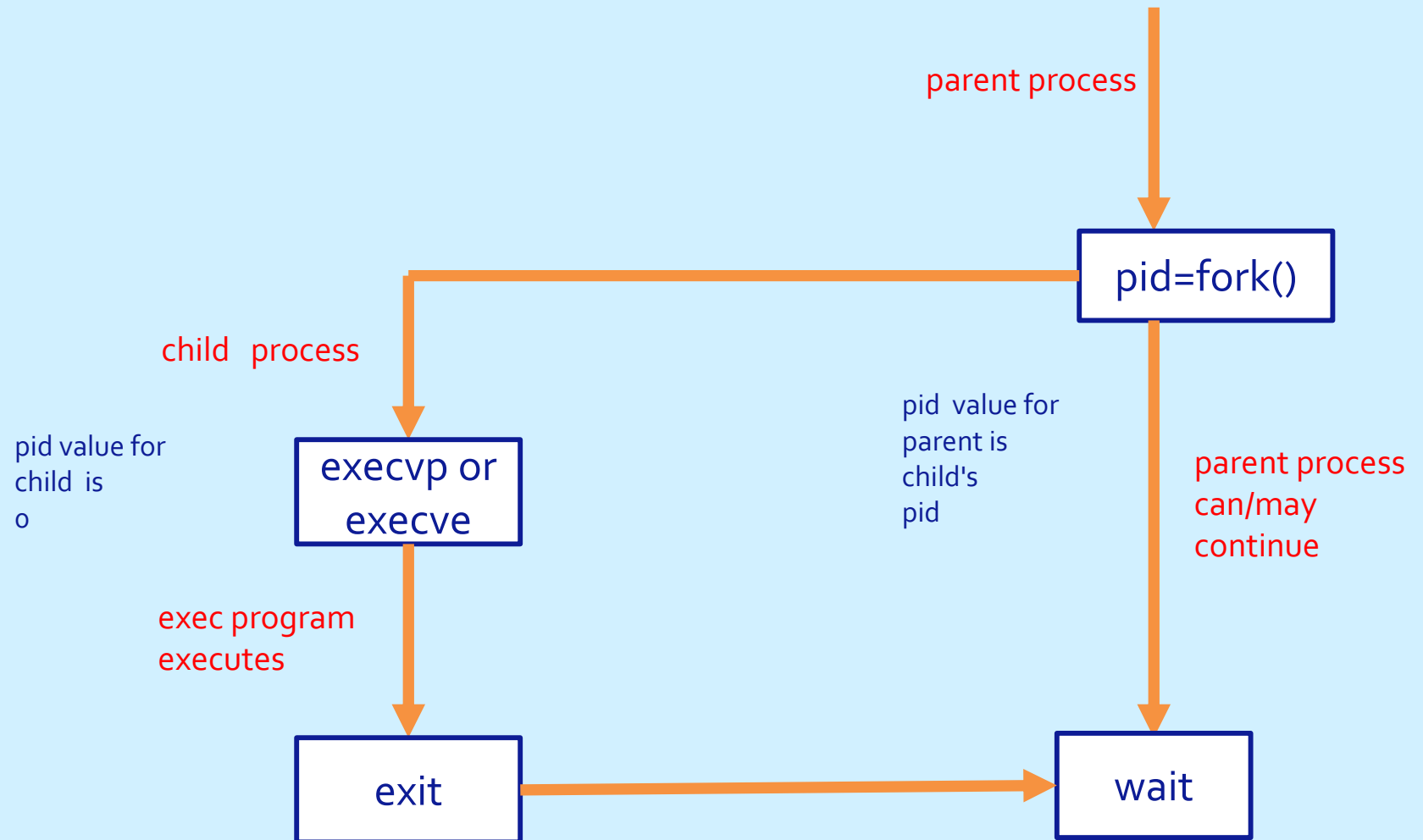
```
% ./zombie
```

```
Parent:(fval,pid,ppid)=(29913,29912,29022)
```

```
Child: (fval,pid,ppid)=( 0,29913,29912)
```

```
Main:(mval,pid)=( 100, 29913)
```

Proper fork usage





In past times,
a `fork()` followed by `exec()` used to waste
time in creating an image of the parent
that was to be replaced with the `exec()`

Thus `fork()` + `exec()` had an optimized
implementation through a
`vmfork()` + `exec()`

In essence a `vmfork()`'s child was utilizing a shared address space with
the parent



- 3.1. How can a system protect itself by the abuse of slide 81?
- 3.2. A program (process) 25 contains three consecutive fork statements
- ```
int main(void) { fork();
 fork();
 fork();
 }
```

No other such statement exists. How many processes will be created? Of those processes, how many will have 25 as parent? Explain.

- 3.3. Did you untar the tar file containing all C source code files, and did you experiment with them on an AFS machine? Practice make perfect!

# Summary

- 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
  - Nonprocess kernel
  - Execution within user processes
  - Process-based operating system
- UNIX SVR4 process management
  - Process states
  - Process description
  - Process control

# NJIT

A white curved line that starts under the 'N', sweeps under the 'J' and 'I', and ends under the 'T', acting as a stylized underline or swoosh.

New Jersey's Science &  
Technology University

*THE EDGE IN KNOWLEDGE*

**Alex. Gerbessiotis**

**CS Department**