Chapter 3
Process Description and Control

Part C: Sections 3.4–*end*

Modes of Execution

User Mode

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

System Mode

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

➔ Remember **Chapter 2**: ***"Monitor"***

Table 3.7

Typical Functions of an Operating System

Kernel

Process Management (*this* chapter)

- Process creation and termination
- Process scheduling and dispatching
- Process switching
- Process synchronization and support for interprocess communication
- Management of process control blocks

Memory Management (➔ chapters 7–8)

- Allocation of address space to processes
- Swapping
- Page and segment management

I/O Management (➔ chapter 11)

- Buffer management
- Allocation of I/O channels and devices to processes

Support Functions

- Interrupt handling
- Accounting
- Monitoring

Process Creation

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

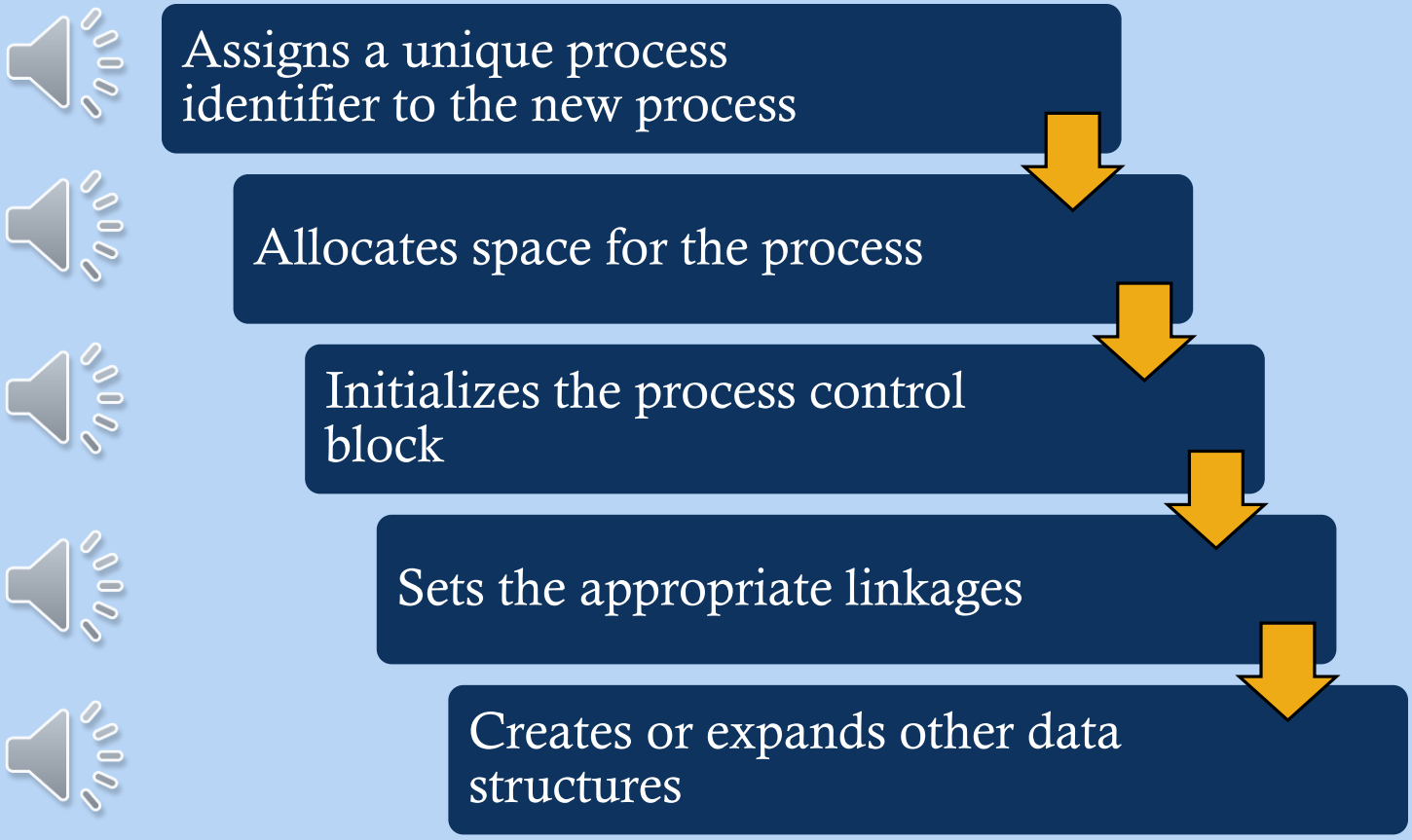


Table 3.8
Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	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

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

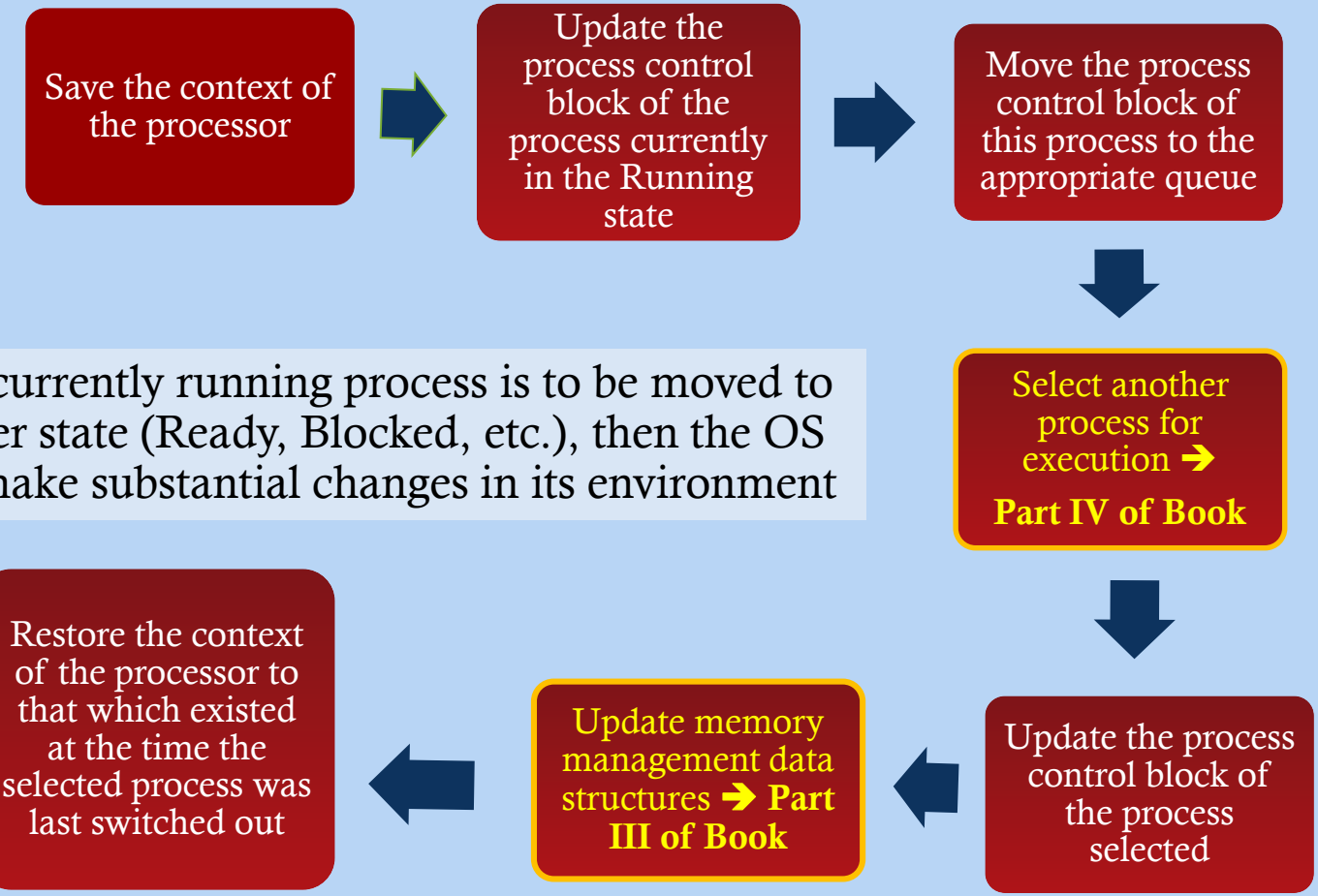
If an interrupt is pending the processor:

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:



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

Execution of the Operating System

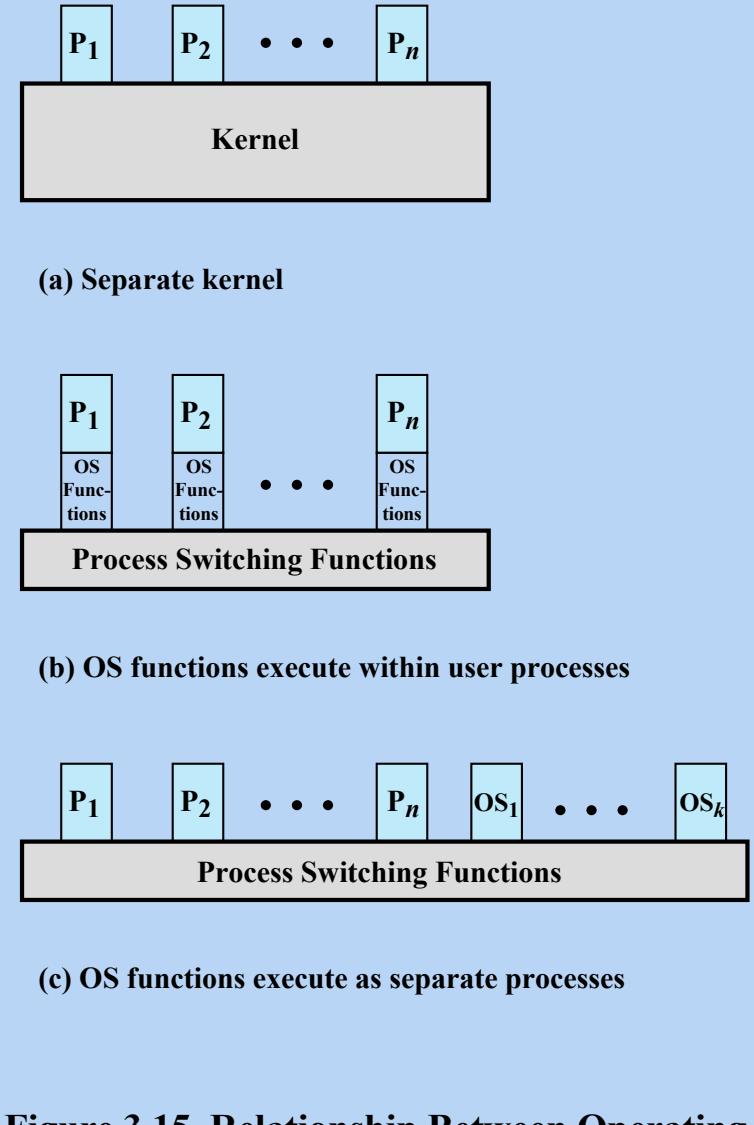


Figure 3.15 Relationship Between Operating System and User Processes

Details to Fig.3.15**b**

Execution Within User Processes

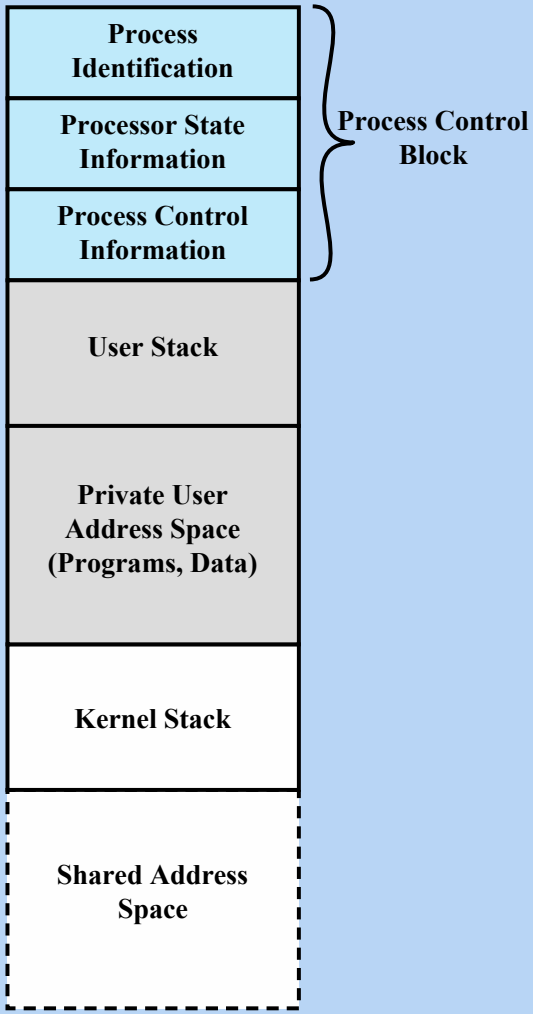


Figure 3.16 Process Image: Operating System Executes Within User Space

Example:

Unix SVR4

- Uses the model where most of the OS executes within the environment of a user process
- System processes run in kernel mode
  - Executes operating system code to perform administrative and housekeeping functions
- User Processes
  - Operate in user mode to execute user programs and utilities
  - Operate in kernel mode to execute instructions that belong to the kernel
  - Enter kernel mode by issuing a system call, when an exception is generated, or when an interrupt occurs

Table 3.9 UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

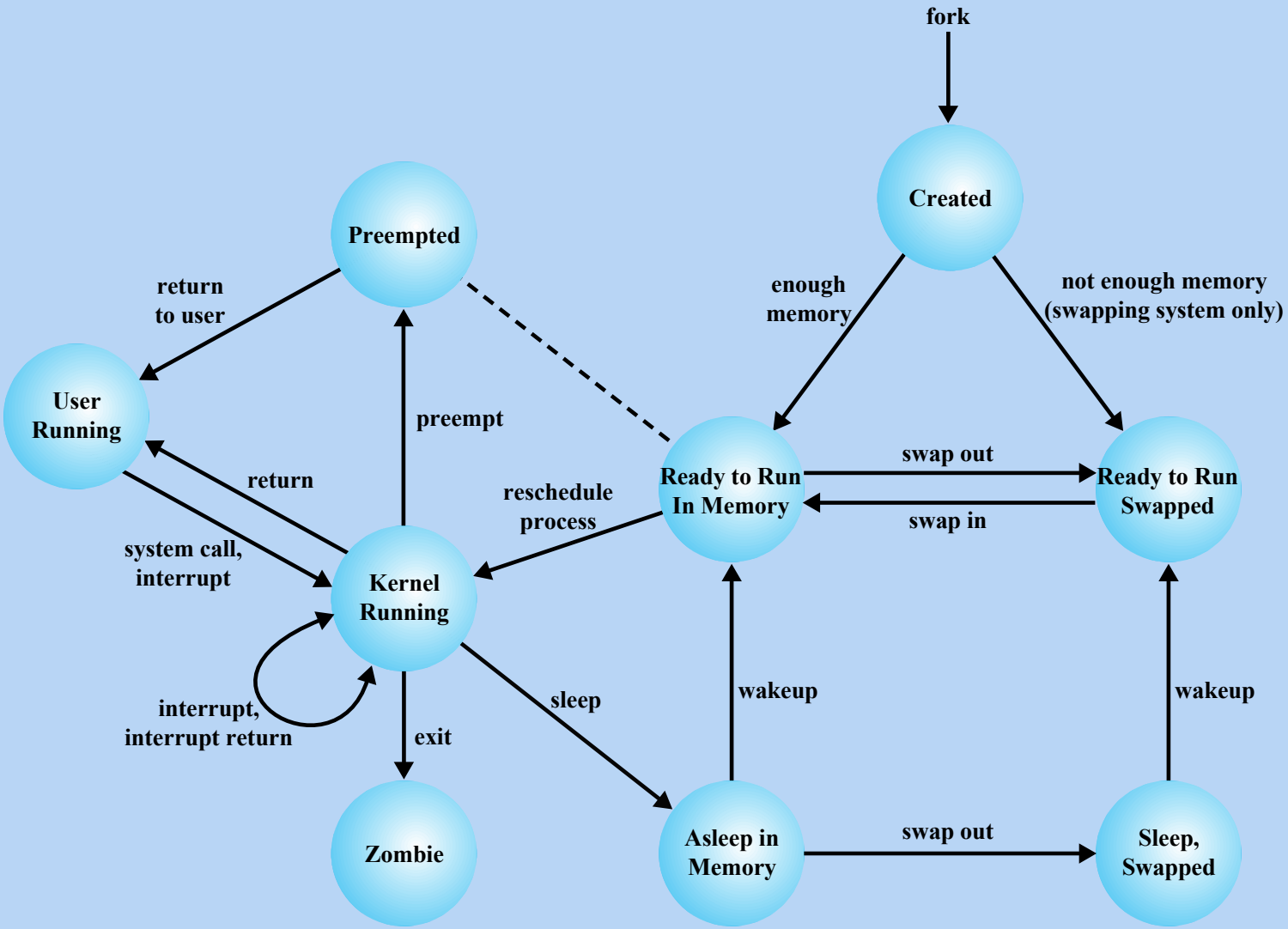


Figure 3.17 UNIX Process State Transition Diagram

Table 3.10 UNIX Process Image

User-Level Context	
Process text	Executable machine instructions of the program
Process data	Data accessible by the program of this process
User stack	Contains the arguments, local variables, and pointers for functions executing in user mode
Shared memory	Memory shared with other processes, used for interprocess communication
Register Context	
Program counter	Address of next instruction to be executed; may be in kernel or user memory space of this process
Processor status register	Contains the hardware status at the time of preemption; contents and format are hardware dependent
Stack pointer	Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption
General-purpose registers	Hardware dependent
System-Level Context	
Process table entry	Defines state of a process; this information is always accessible to the operating system
U (user) area	Process control information that needs to be accessed only in the context of the process
Per process region table	Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute
Kernel stack	Contains the stack frame of kernel procedures as the process executes in kernel mode

Table 3.11 UNIX Process Table Entry

Process status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process size	Enables the operating system to know how much space to allocate the process.
User identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

Table 3.12 UNIX U Area

Process table pointer	Indicates entry that corresponds to the U area.
User identifiers	Real and effective user IDs. Used to determine user privileges.
Timers	Record time that the process (and its descendants) spent executing in user mode and in kernel mode.
Signal-handler array	For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function).
Control terminal	Indicates login terminal for this process, if one exists.
Error field	Records errors encountered during a system call.
Return value	Contains the result of system calls.
I/O parameters	Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O.
File parameters	Current directory and current root describe the file system environment of the process.
User file descriptor table	Records the files the process has opened.
Limit fields	Restrict the size of the process and the size of a file it can write.
Permission modes fields	Mask mode settings on files the process creates.

Example: UNIX

Process Control

- 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

A *true* anecdote ☺

Many many many years ago
in the university's open UNIX lab,
a student launched the progam

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

A few seconds later, the lab's Sys-Admin woke up from his lab-slumber, and started shouting, yelling, cursing and swearing in a loud voice...

Can you explain why?

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.