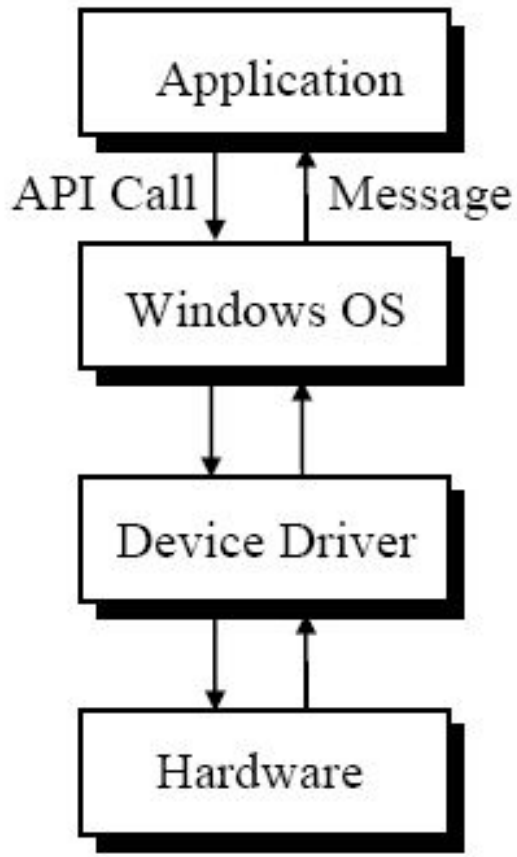
Operating Systems-Module 2

What is an API?

- API, an abbreviation of ***a****pplication* ***p****rogram* ***i****nterface*, is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

- Most operating environments, such as MS-Windows, provide an API so that programmers can write applications consistent with the operating environment. Although APIs are designed for programmers, they are ultimately good for users because they guarantee that all programs using a common API will have similar interfaces. This makes it easier for users to learn new programs.


Application
API Call
Message
Windows OS
Device Driver
Hardware

What is an API?

- An API is an abstraction that describes an interface for the interaction with a set of functions used by components of a software system. The software providing the functions described by an API is said to be an *implementation* of the API.

An API can be:

- general, the full set of an API that is bundled in the libraries of a programming language, e.g. Standard Template Library in C++ or Java API.
- specific, meant to address a specific problem, e.g. Google Maps API or Java API for XML Web Services.
- language-dependent, meaning it is only available by using the syntax and elements of a particular language, which makes the API more convenient to use.
- language-independent, written so that it can be called from several programming languages.

WINDOWS API

- **Purpose**

  The Microsoft Windows application programming interface (API) provides services used by all Windows-based applications.

  You can provide your application with a graphical user interface; access system resources such as memory and devices; display graphics and formatted text; incorporate audio, video, networking, or security.

- **Where Applicable**

  The Windows API can be used in all Windows-based applications. The same functions are generally supported on 32-bit and 64-bit Windows.

- **Developer Audience**

  This API is designed for use by C/C++ programmers. Familiarity with the Windows graphical user interface and message-driven architecture is required.

EXAMPLE WIN32 API

- **Application window is created by calling the API function CreateWindow().**

- **Create Window with the comments identifying the parameter.**

```
hwnd = CreateWindow
 (“classname”,                 // window class name
TEXT ("The First Program"),          // window caption
WS_OVERLAPPEDWINDOW,        // window style
CW_USEDEFAULT,                 // initial x position
CW_USEDEFAULT,                 // initial y position
CW_USEDEFAULT,                 // initial x size
CW_USEDEFAULT,                 // initial y size
NULL,                          // parent window handle
NULL,                          // window menu handle
hInstance,                     // program instance handle
NULL) ;    // creation parameters, may be used to point some data for reference.
```

- **Overlapped window will be created, it includes a title bar, system menu to the left of title bar, a thick window sizing border, minimize, maximize and close button to the right of the title bar.**

- **The window will be placed in default x, y position with default size. It is a top level window without any menu.**
- **The CreateWindow() will returns a handle which is stored in hwnd.**

Characteristics of a Good API

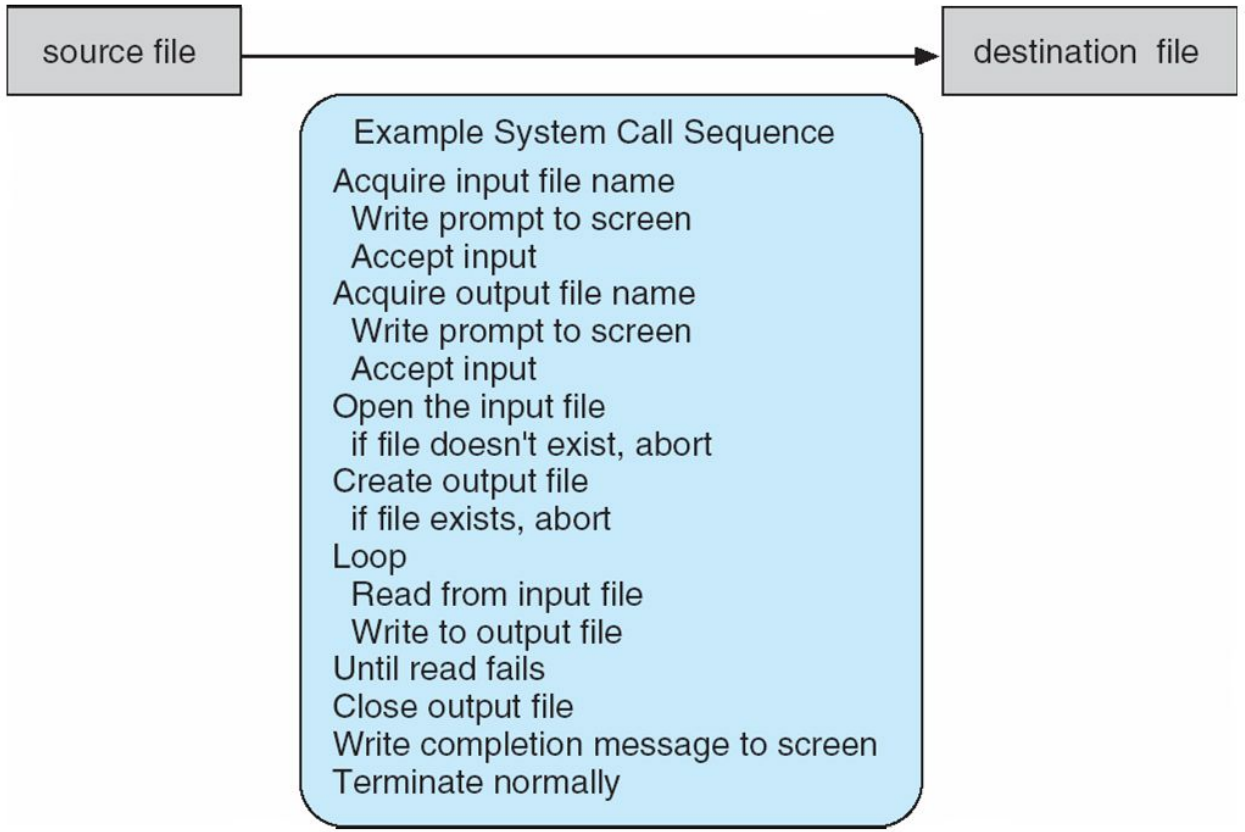
- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

System Calls

- Programming interface to the services provided by the OS
- The kernel is a computer program that is the core of a computer’s operating system, with complete control over everything in the system.
- A system call is how a program requests a service from an operating system's kernel.
- System calls provide an essential interface between a process and the operating system.
- Typically written in a high-level language (C or C++) can be written in assembly language for low level tasks.
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

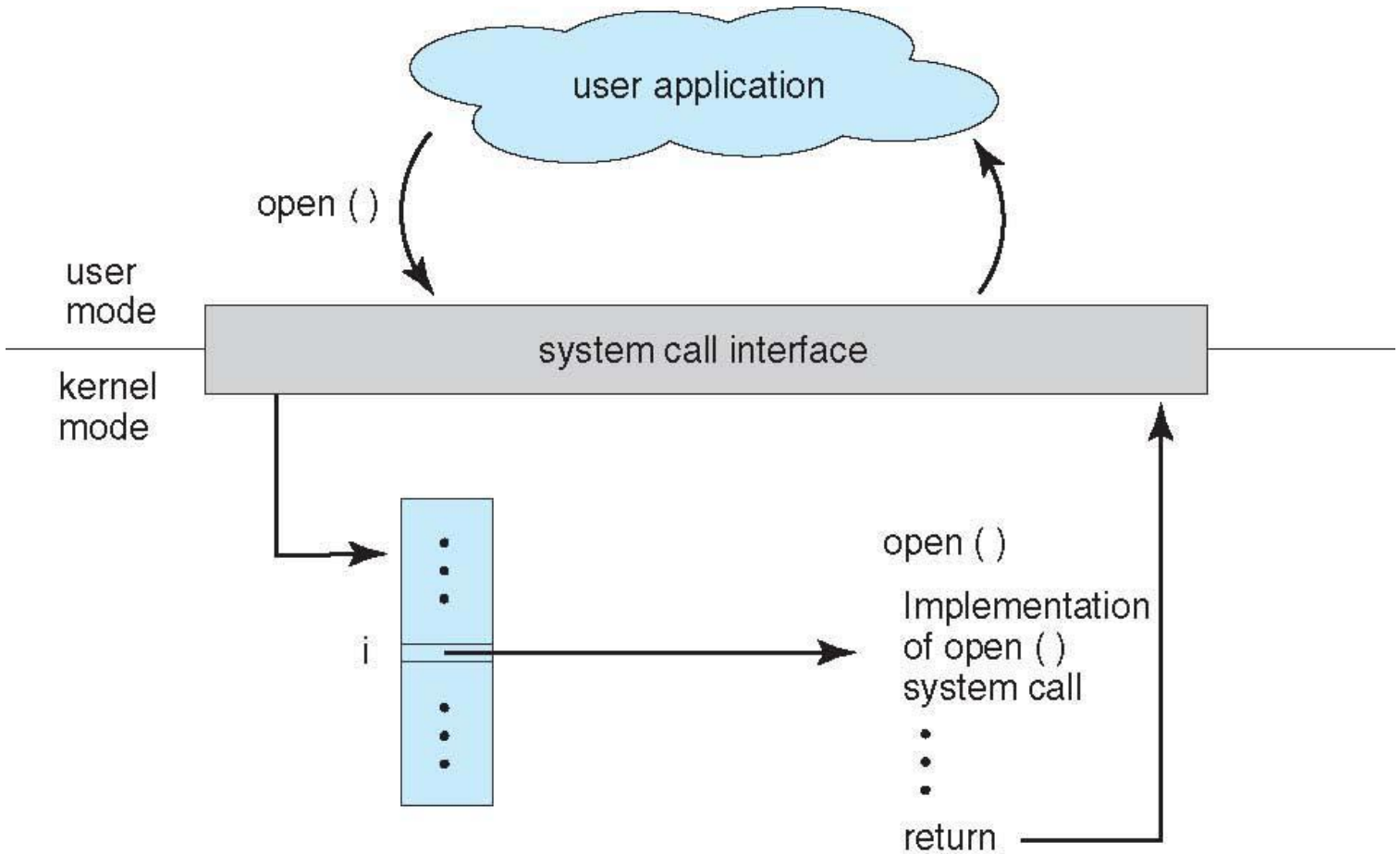
- System call sequence to copy the contents of one file to another file



System Call Implementation

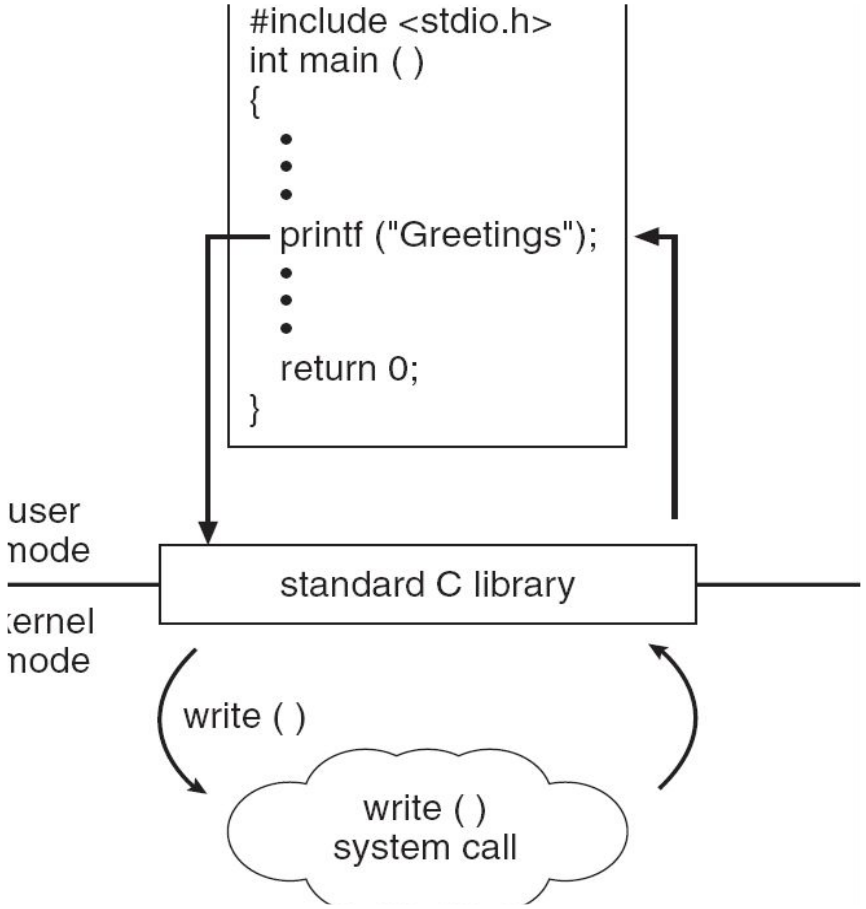
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution;
- The entity that can be assigned to and executed on a processor
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

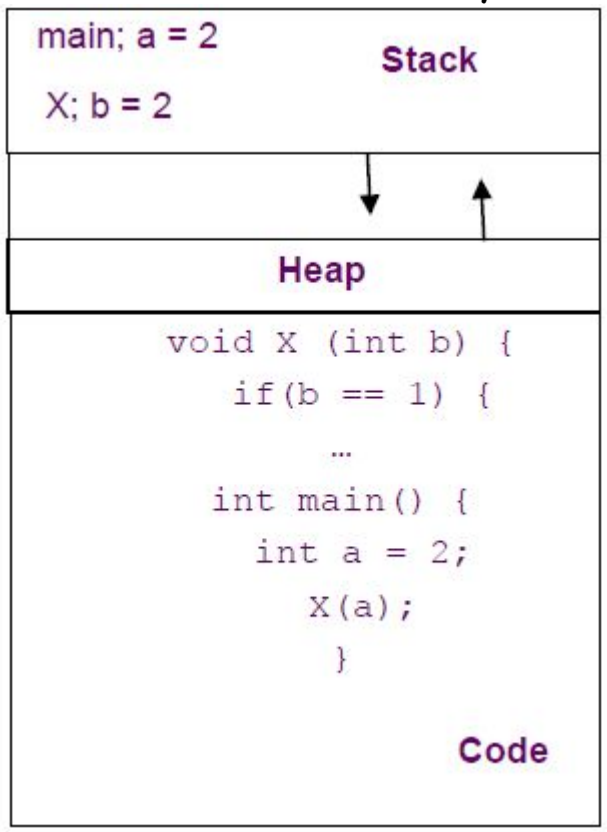
Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

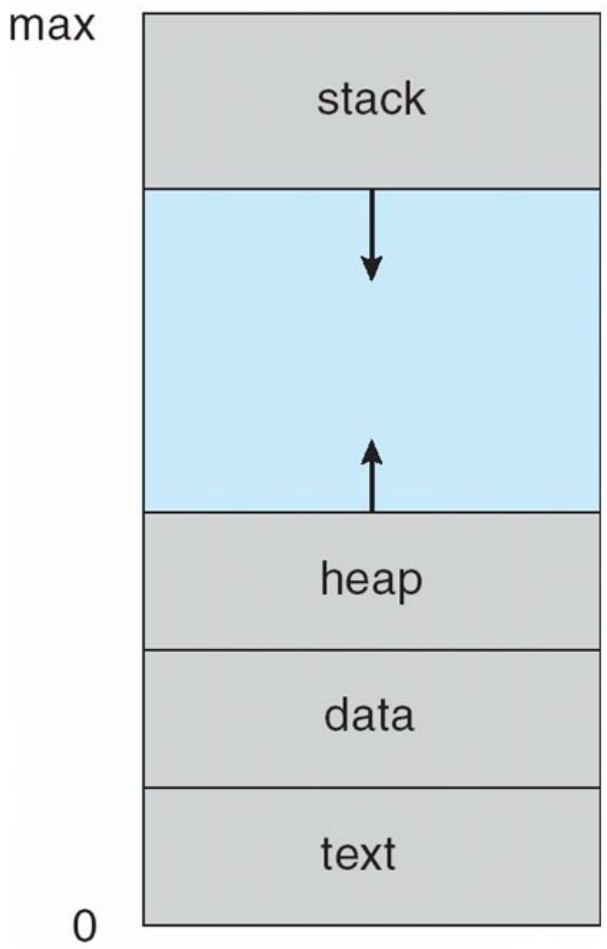
What you wrote

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

What is in memory.

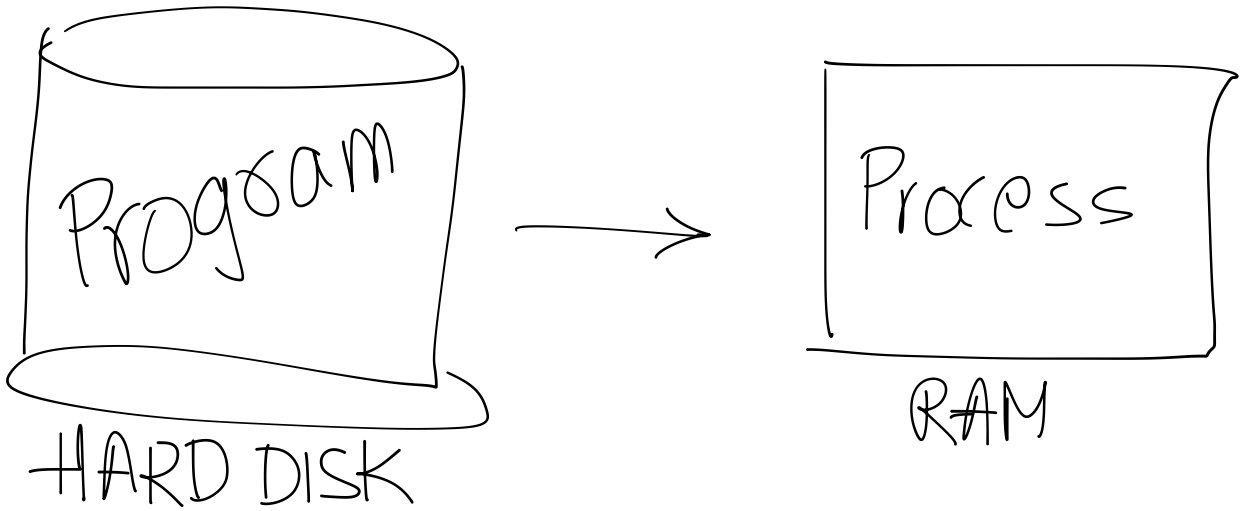


Process in Memory



Process Concept (Cont.)

- Program is ***passive*** entity stored on disk , process is ***active***
  - Program becomes process when file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc



Program, executable and process

In order to execute a program, the operating system must first create a process and make the process execute the program.

Program

A set of instructions which is in human readable format. A passive entity stored on secondary storage.

Executable

A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage.

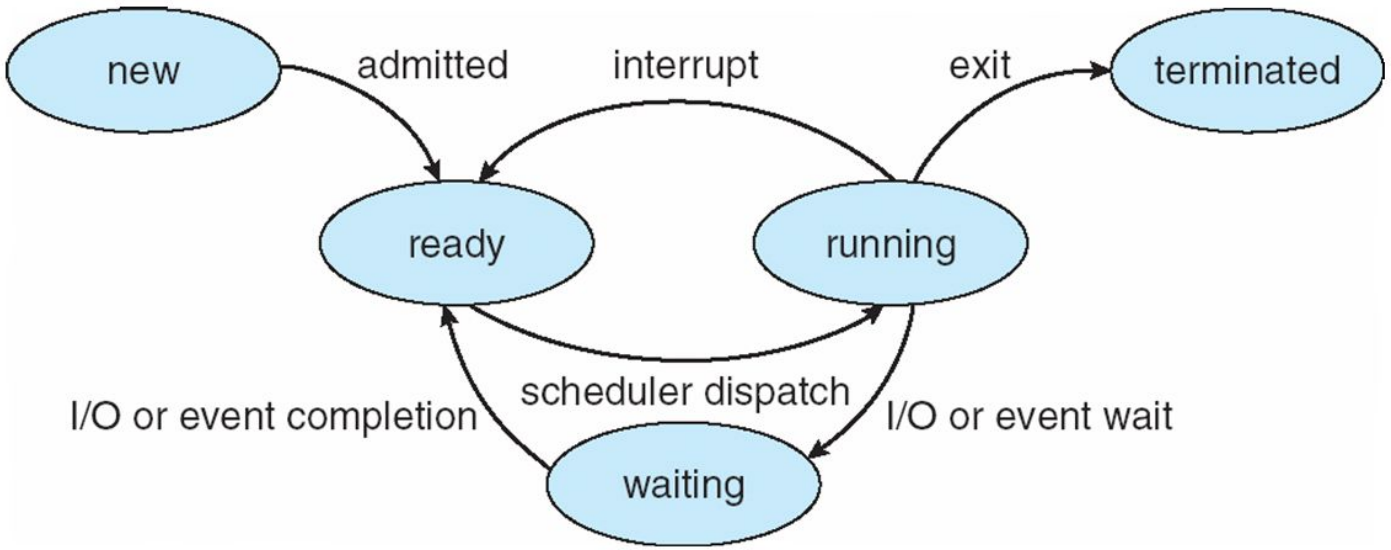
Process

A program loaded into memory and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting). A process is an active entity and needs resources such as CPU time, memory etc to execute.

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

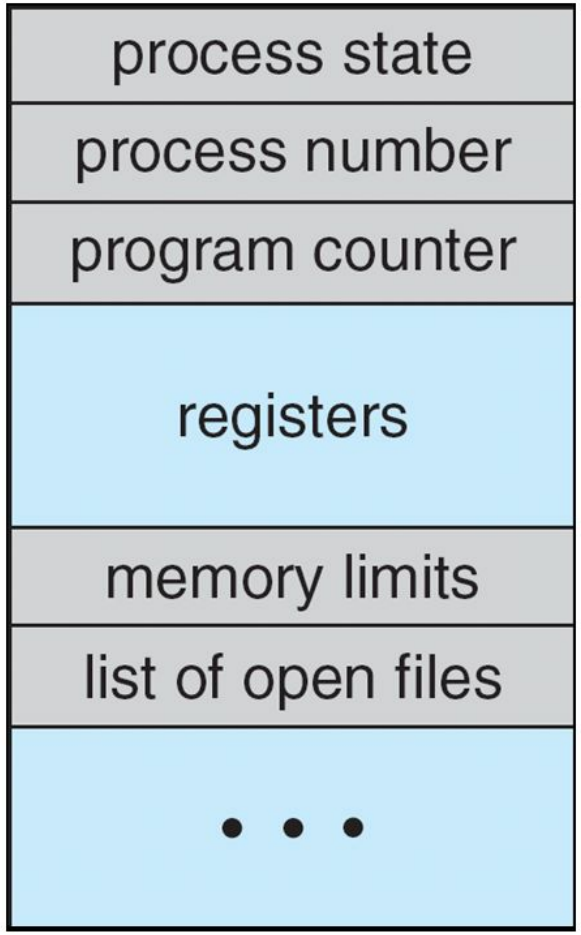


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

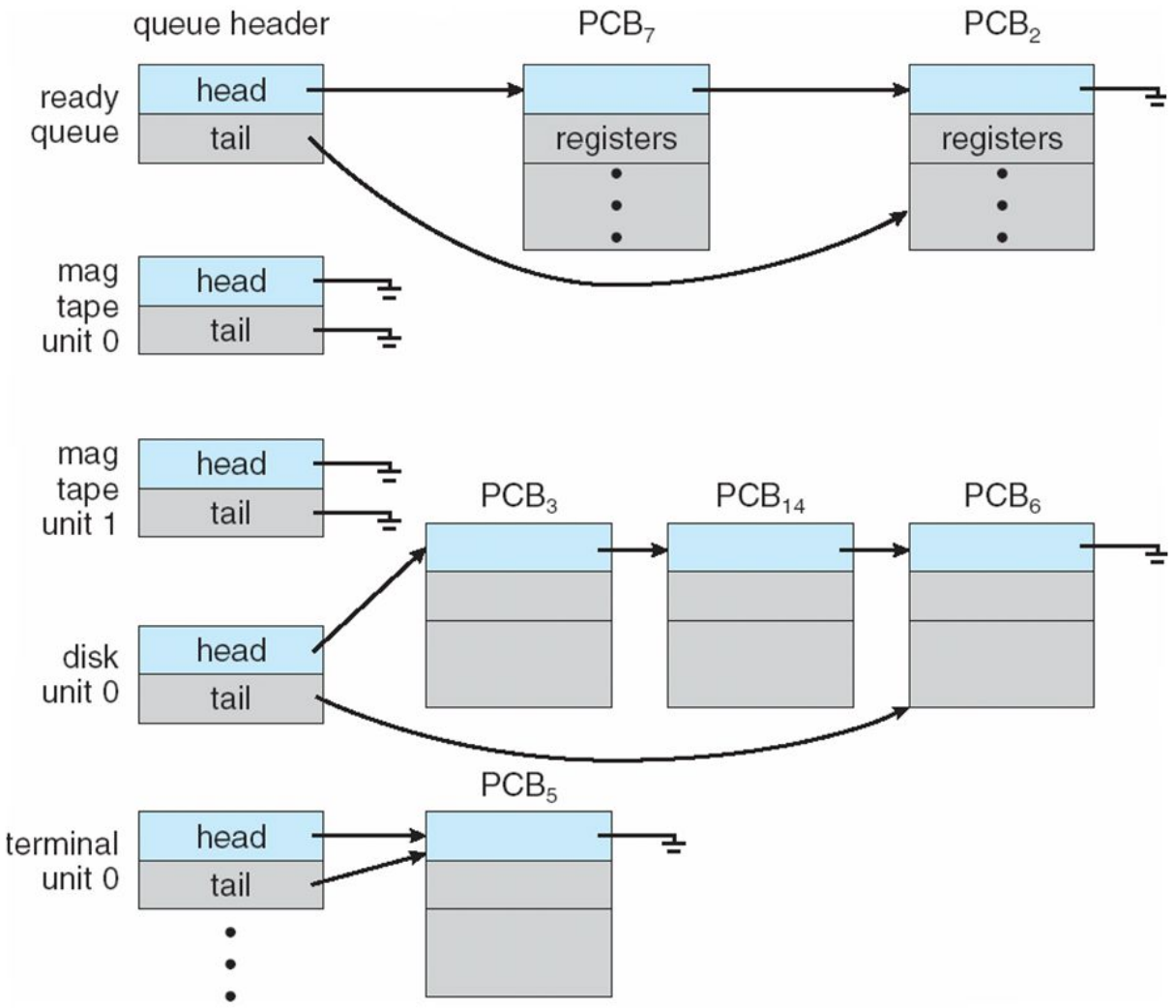
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process Scheduling

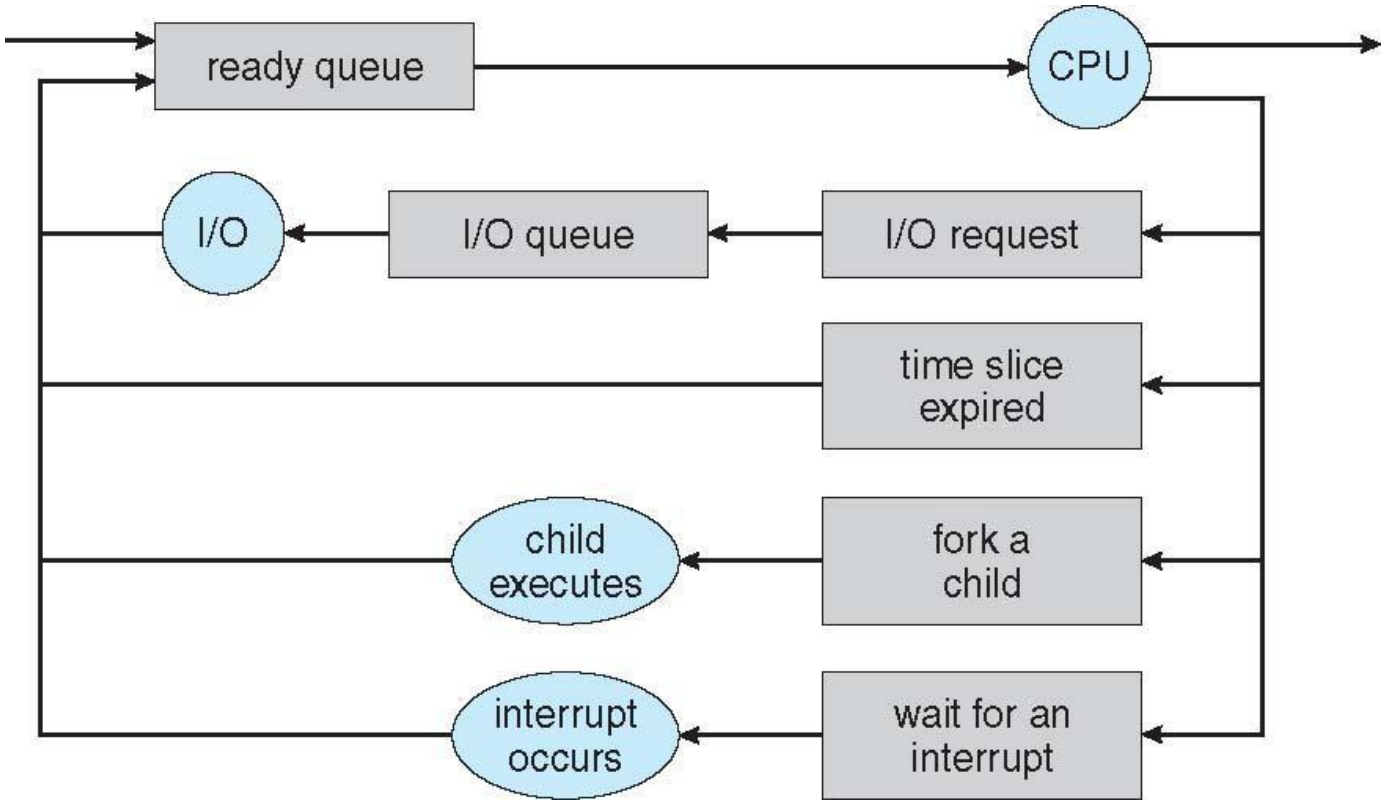
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

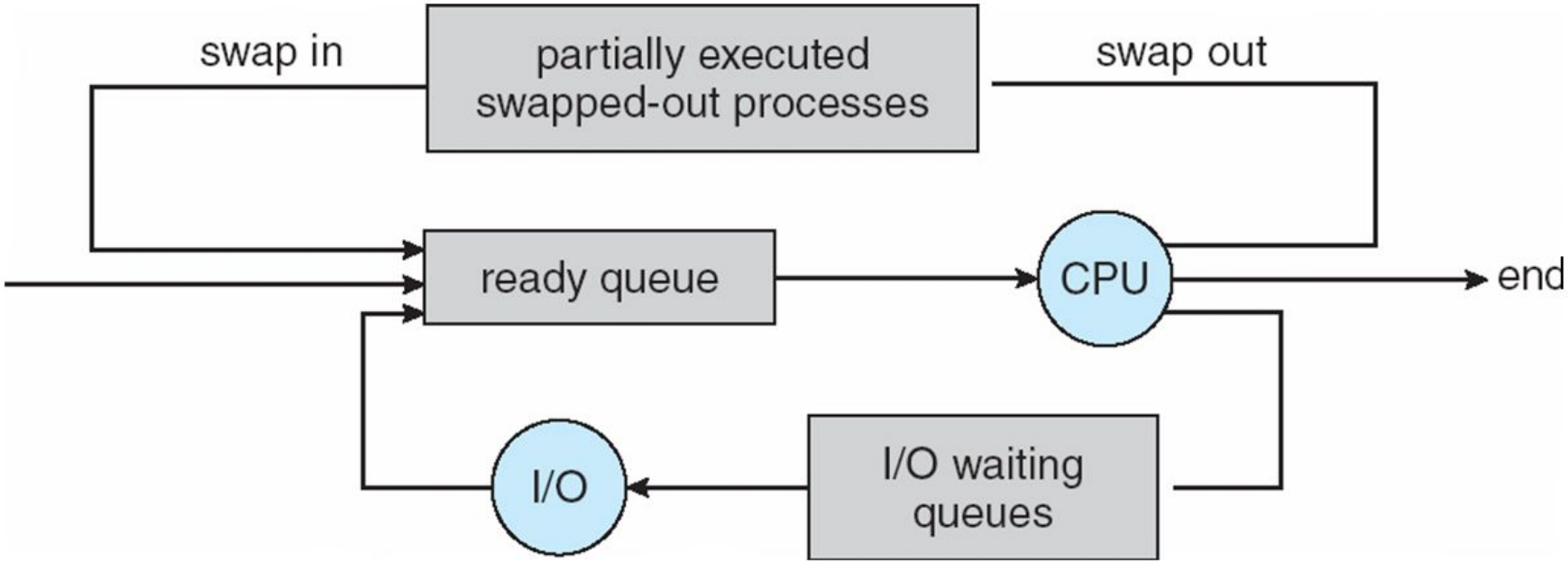


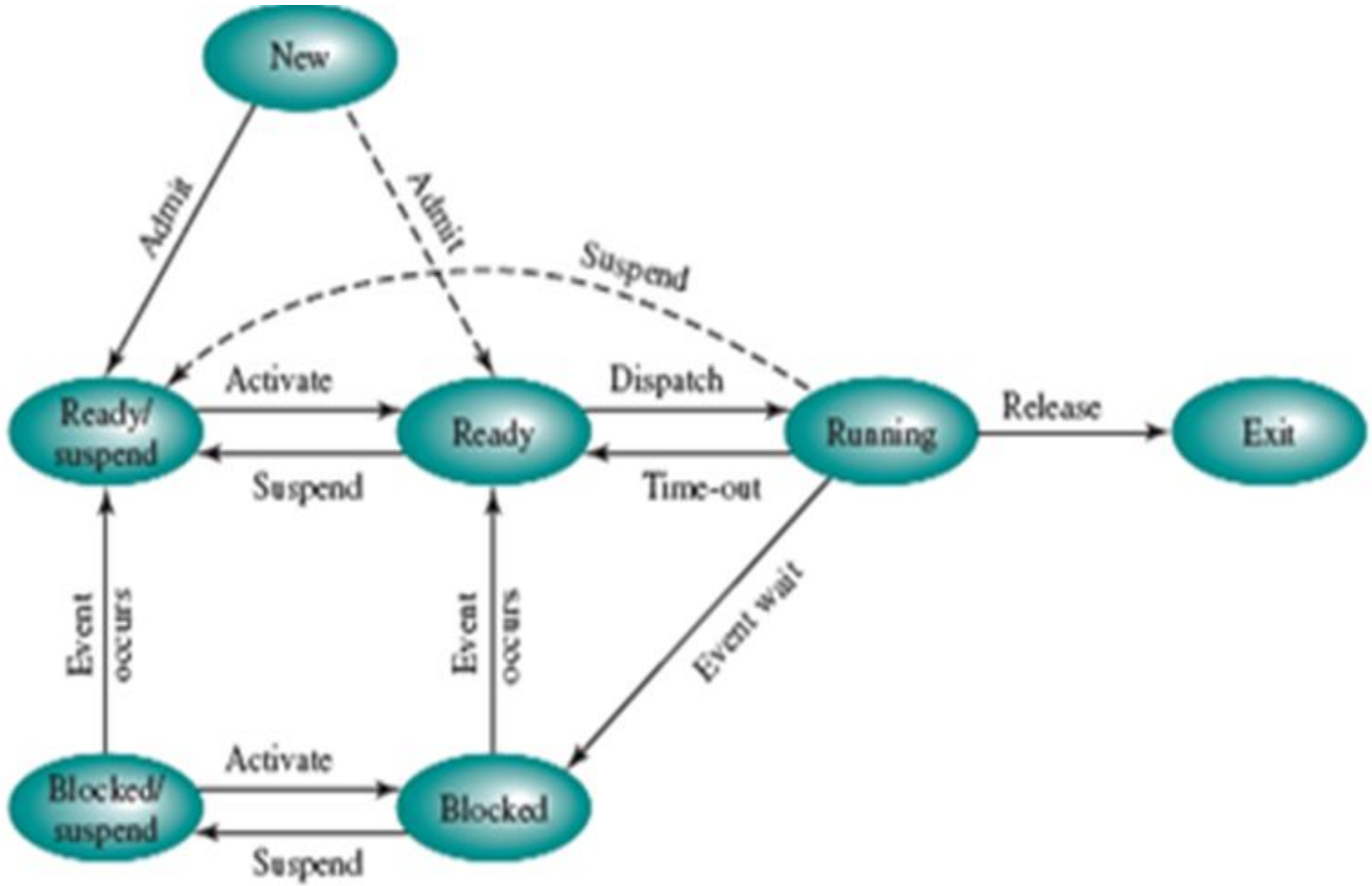
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

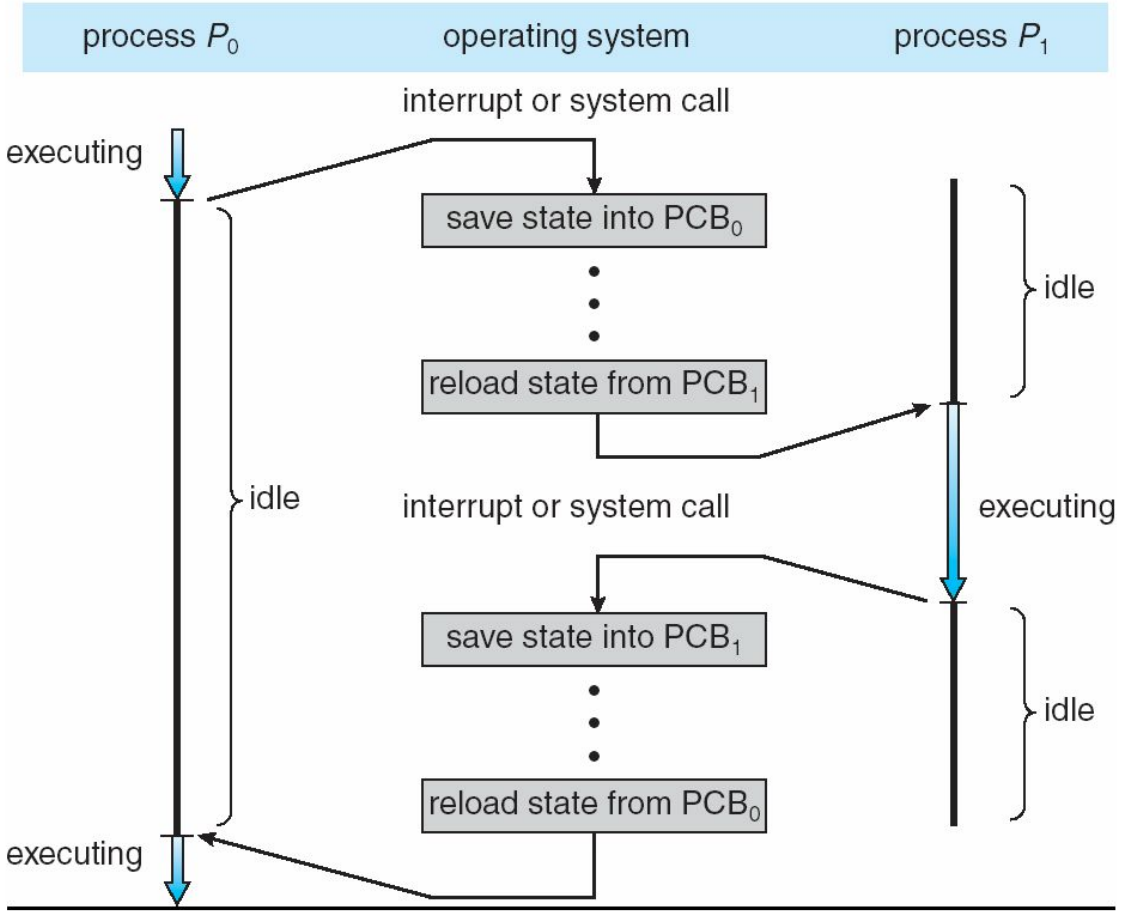



New
Admit
Admit
Suspend
Ready/
suspend
Activate
Ready
Dispatch
Running
Release
Exit
Suspend
Time-out
Event
occurs
Event
occurs
Event wait
Blocked/
suspend
Activate
Blocked
Suspend

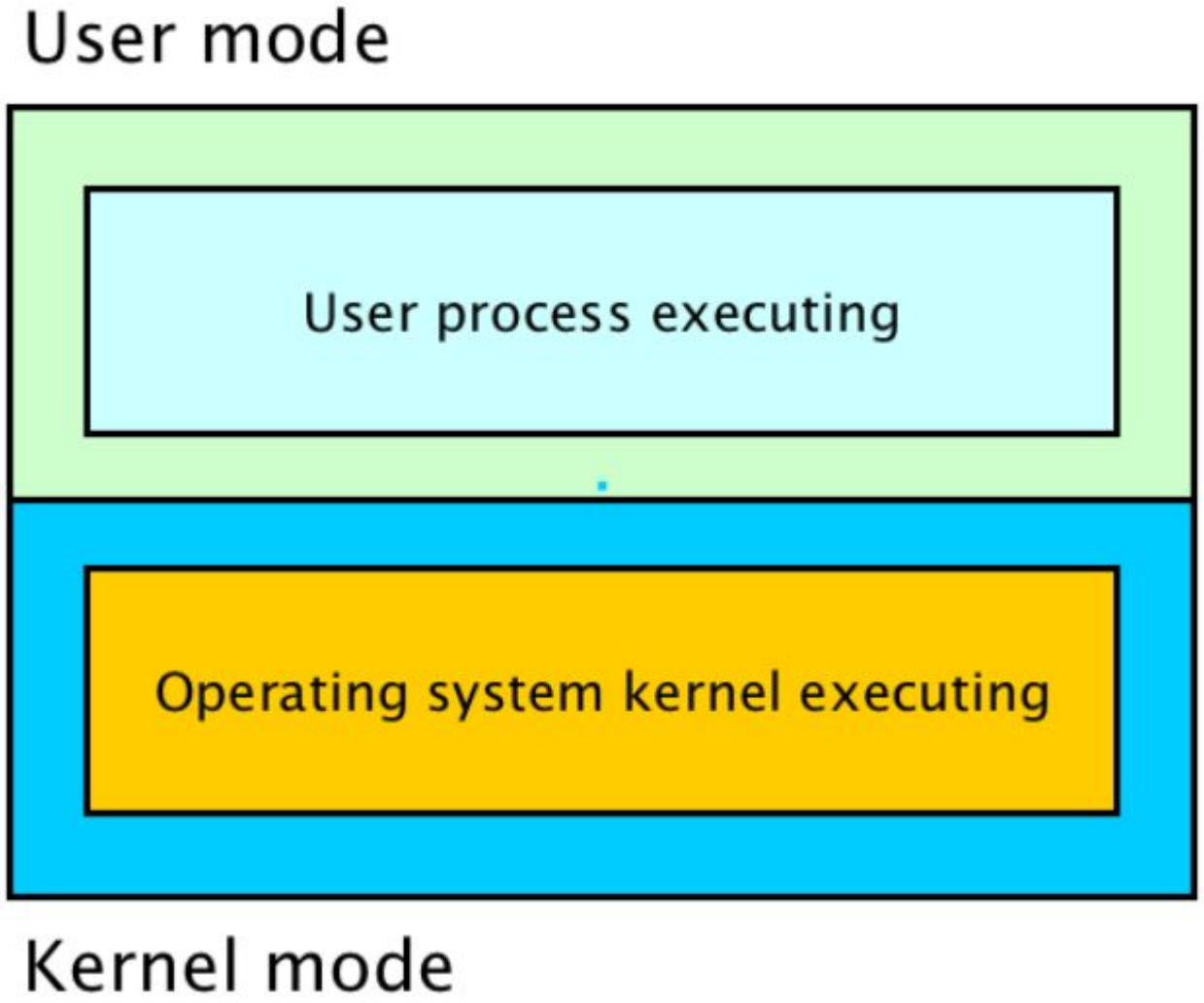
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB □ the longer the context switch
- Time dependent on hardware support

CPU Switch From Process to Process



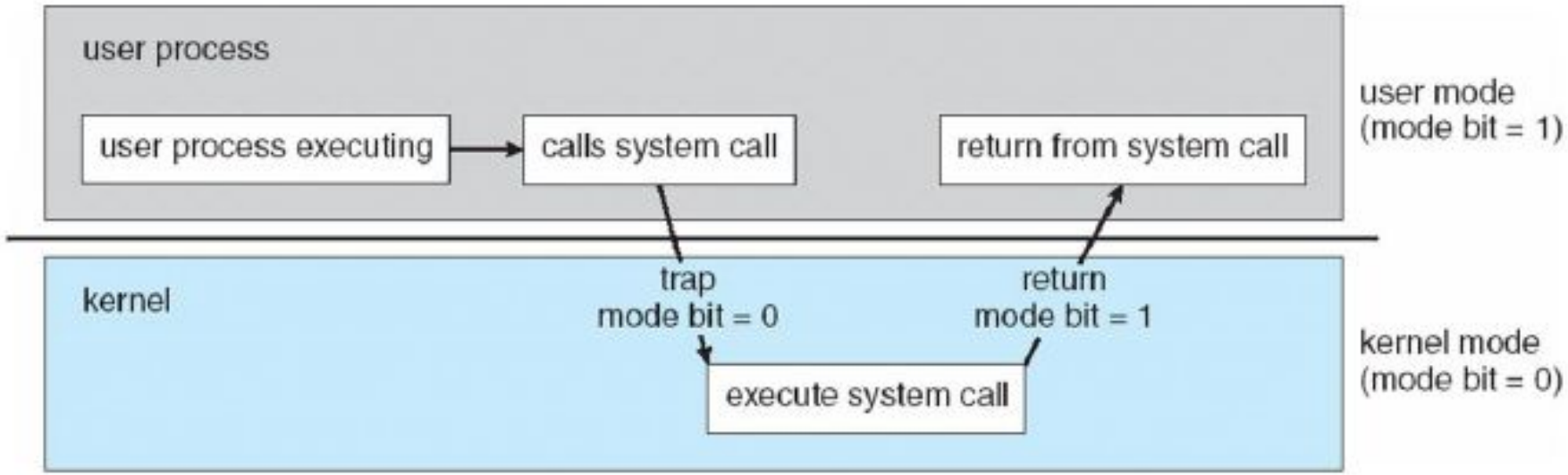
Operating system operations



Operating system operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

Transition from user to kernel mode



Exception and interrupts

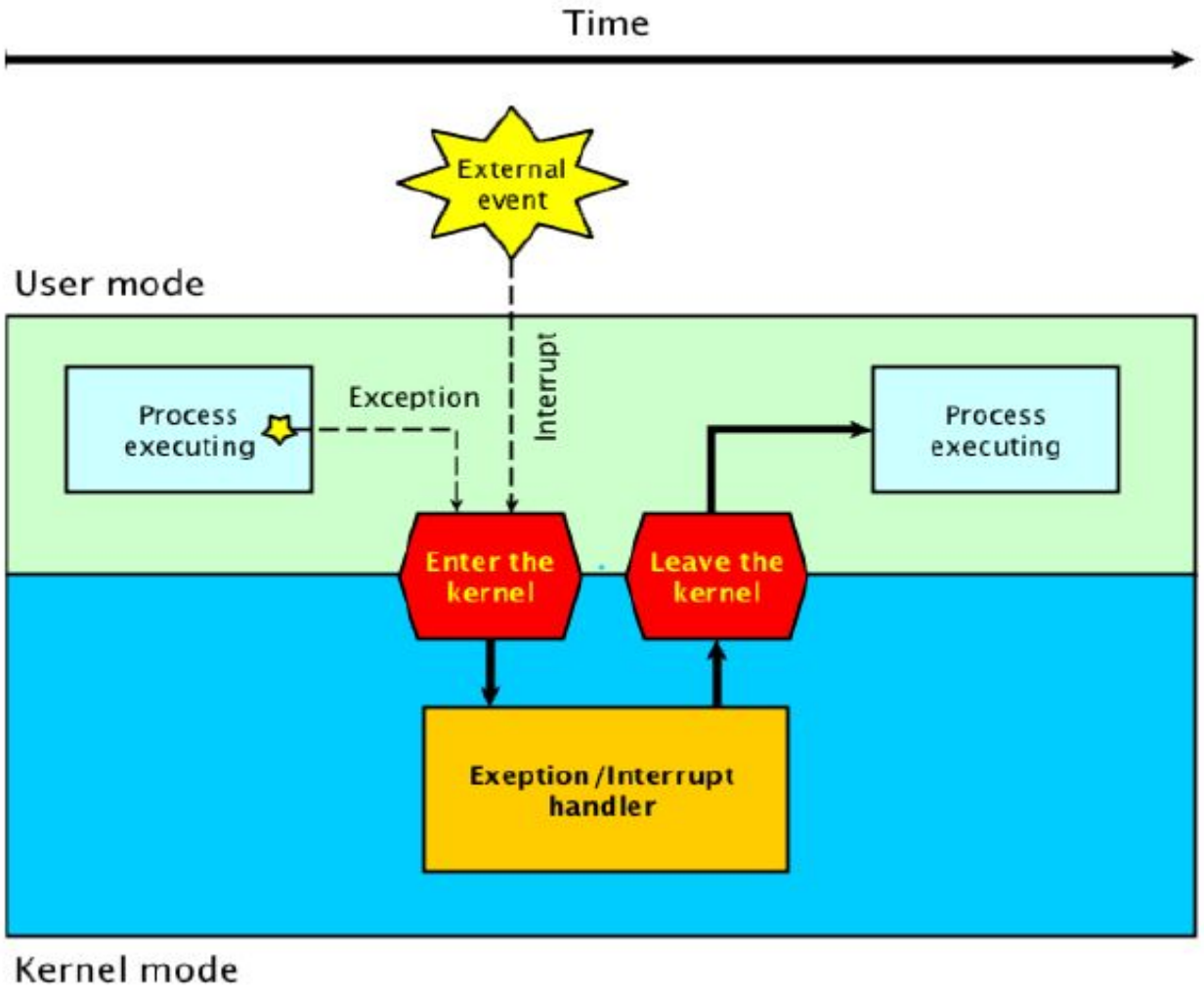
Exceptions are internal and synchronous

- Exceptions are used to handle **internal program errors**.
- Overflow, division by zero and bad data address are examples of internal errors in a program.
- Another name for exception is trap. **A trap (or exception) is a software generated interrupt**.
- Exceptions are produced by the CPU control unit while executing instructions and are considered to be synchronous because the control unit issues them only after terminating the execution of an instruction.

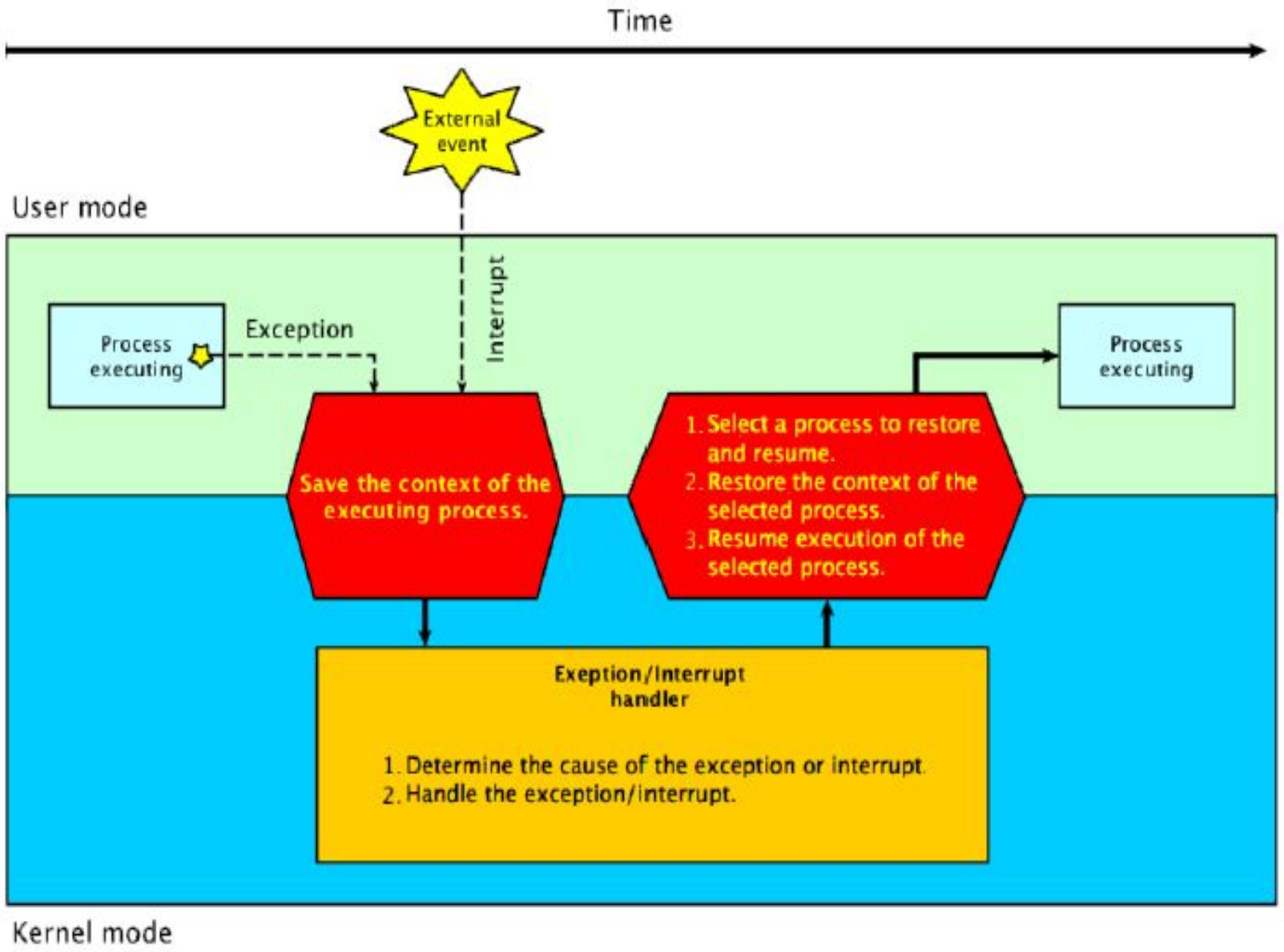
Interrupts are external and asynchronous

- Interrupts are used **to notify the CPU of external events**.
- Interrupts are **generated by hardware devices** outside the CPU at arbitrary times with respect to the CPU clock signals and are therefore considered to be asynchronous.
- Read and write requests to disk is similar to key presses.

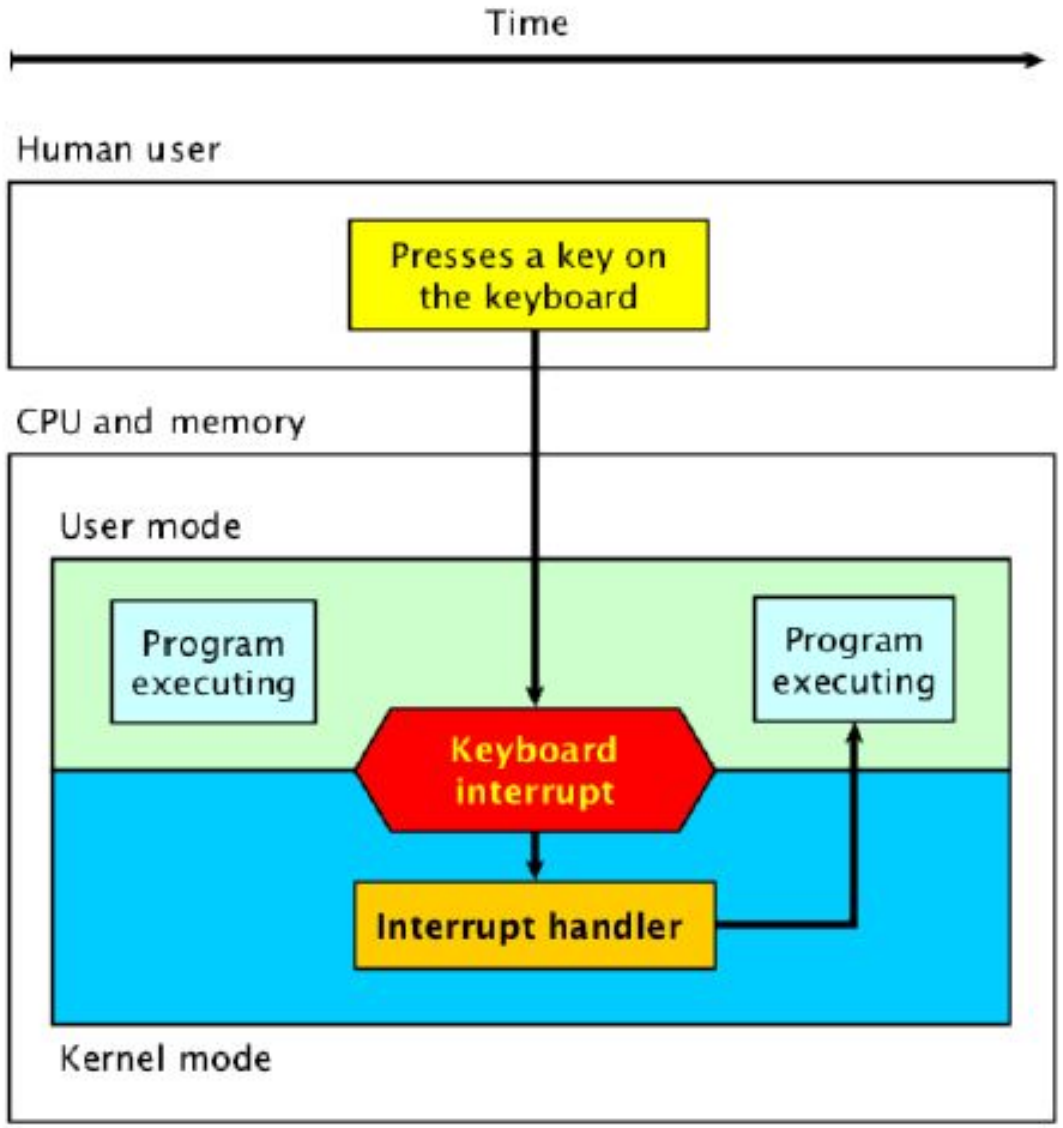
Exception and interrupt handler

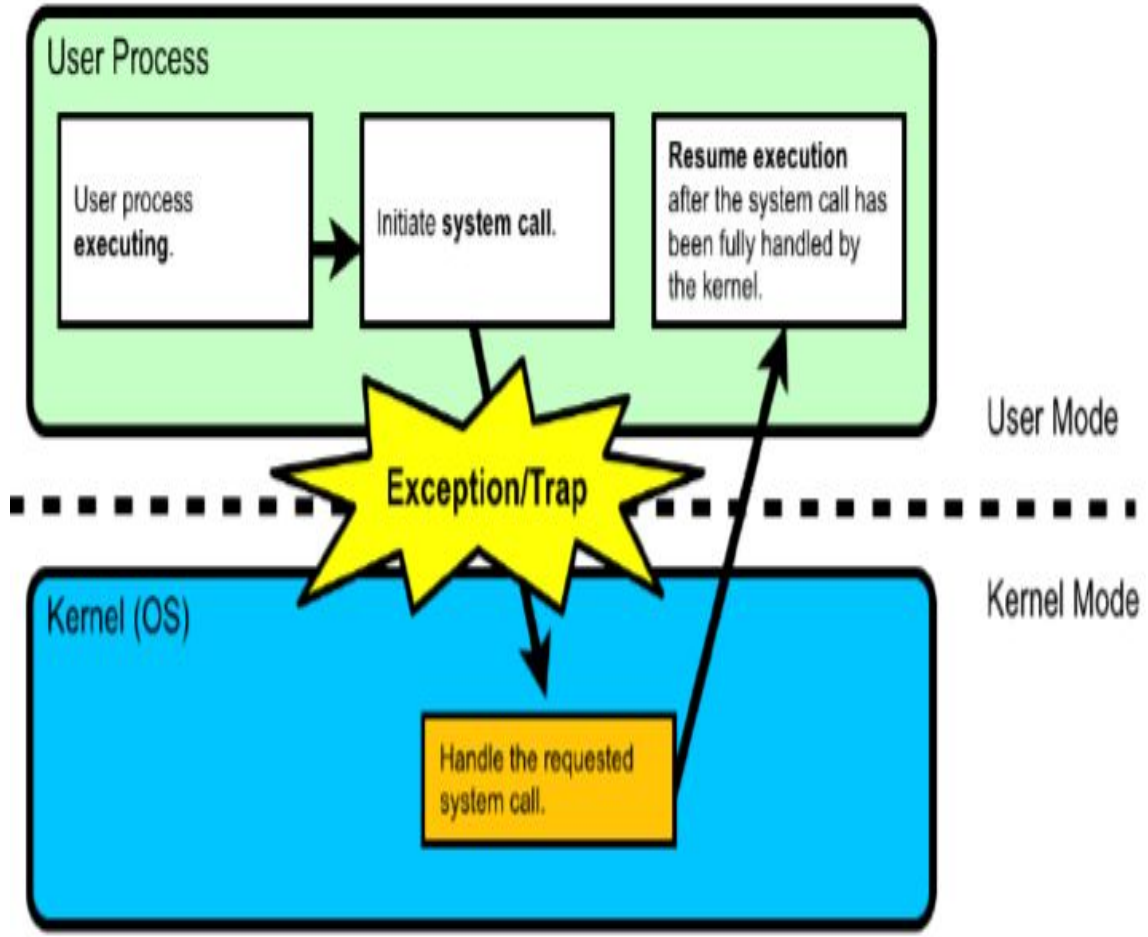


- When an exception or interrupt occurs, execution transition from user mode to kernel mode where the exception or interrupt is handled.
- When the exception or interrupt has been handled execution resumes in user space.



- When an exception or interrupt occurs, execution transition from user mode to kernel mode where the exception or interrupt is handled.
- While entering the kernel, **the context (values of all CPU registers)** of the currently executing process must first be saved to memory.
  - The kernel is now ready to handle the exception/interrupt.
  - Determine the cause of the exception/interrupt.
  - Handle the exception/interrupt.
- When the exception/interrupt have been handled the kernel performs the following steps:
  - Select a process to restore and resume.
  - Restore the context of the selected process.
  - Resume execution of the selected process.


Time
Human user
Presses a key on the keyboard
CPU and memory
User mode
Program executing
Program executing
Keyboard interrupt
Interrupt handler
Kernel mode


User Process
User process executing.
Initiate system call.
Resume execution after the system call has been fully handled by the kernel.
User Mode
Exception/Trap
Kernel Mode
Kernel (OS)
Handle the requested system call.

Operations on Processes

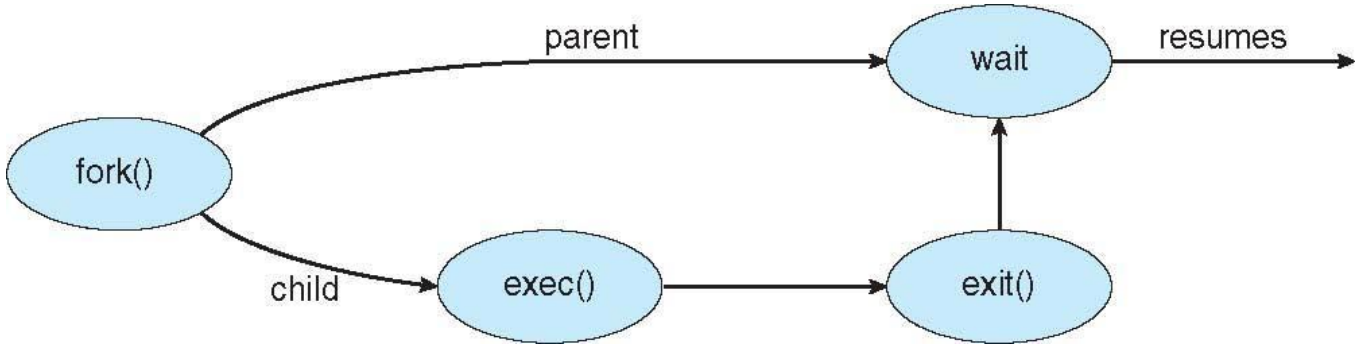
- System must provide mechanisms for:
  - process creation,
  - process termination

Process Creation

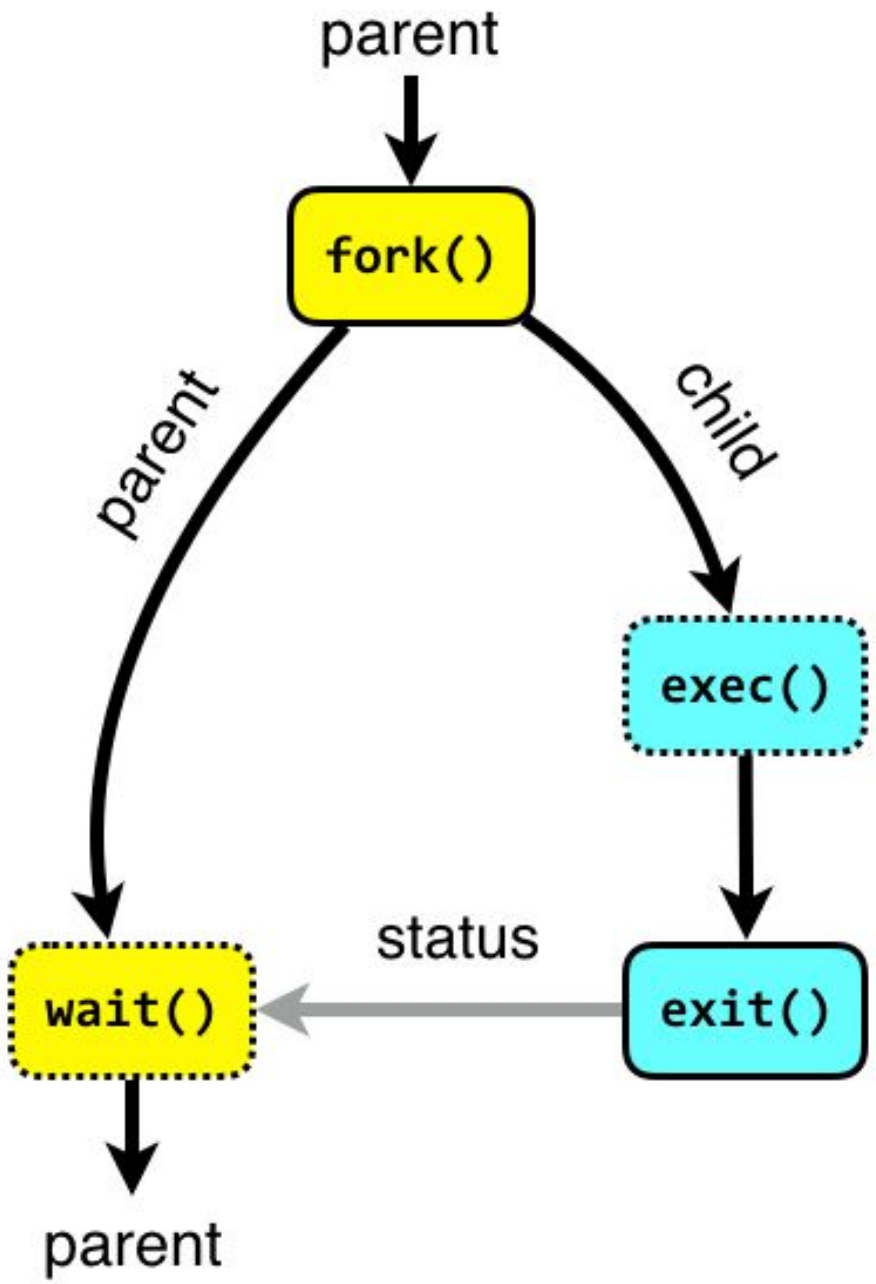
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Process Management - System calls



The following system calls are used for basic process management.

fork

- A parent process uses fork to create a new child process. The child process is a copy of the parent. After fork, both parent and child executes the same program but in separate processes.

exec

- Replaces the program executed by a process. The child may use exec after a fork to replace the process' memory space with a new program executable making the child execute a different program than the parent.

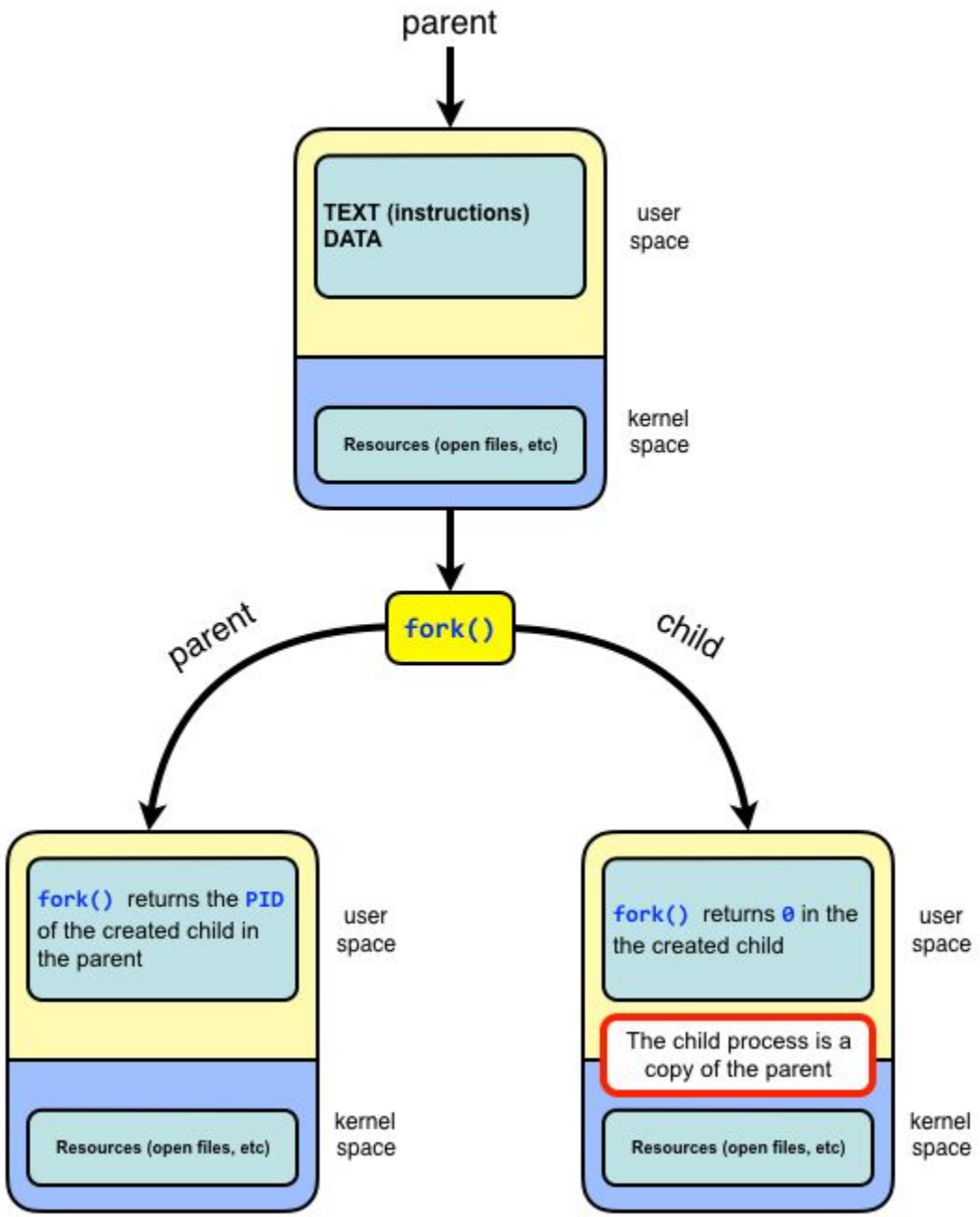
exit

- Terminates the process with an exit status.

wait

- The parent may use wait to suspend execution until a child terminates. Using wait the parent can obtain the exit status of a terminated child

Process Mgt. - System Calls



#include <unistd.h>

pid_t fork(void);

- On success, the PID of the child process is returned in the parent, and 0 is returned in the child.
- On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Fork returns twice on success

- On success fork returns twice: once in the parent and once in the child.
- After calling fork, the program can use the fork return value to tell whether executing in the parent or child.
  - If the return value is 0 the program executes in the new child process.
  - If the return value is greater than zero, the program executes in the parent process and the return value is the process ID (PID) of the created child process.
  - On failure fork returns -1.

fork() - example

```
int main(void) {
  pid_t pid;

  switch (pid = fork()) {
    case -1:
      // On error fork() returns -1.
      perror("fork failed");
      exit(EXIT_FAILURE);
    case 0:
      // On success fork() returns 0 in the child.
      child();
    default:
      // On success fork() returns the pid of the child to the
parent.
      parent(pid);
  }
}
```

```
void child()
{
printf(" CHILD <%ld> I'm alive! My PID is <%ld> and my parent got  PID
<%ld>.\n", (long) getpid(), (long) getpid(), (long)    getppid());
printf(" CHILD <%ld> Goodbye!\n", (long)      getpid());
exit(EXIT_SUCCESS);
}
```

```
void parent(pid_t pid)
{
printf("PARENT <%ld> My PID is <%ld> and I spawned a child with   PID
<%ld>.\n", (long) getpid(), (long) getpid(), (long) pid); printf("PARENT
<%ld> Goodbye!\n", (long) getpid()); exit(EXIT_SUCCESS);
}
```

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

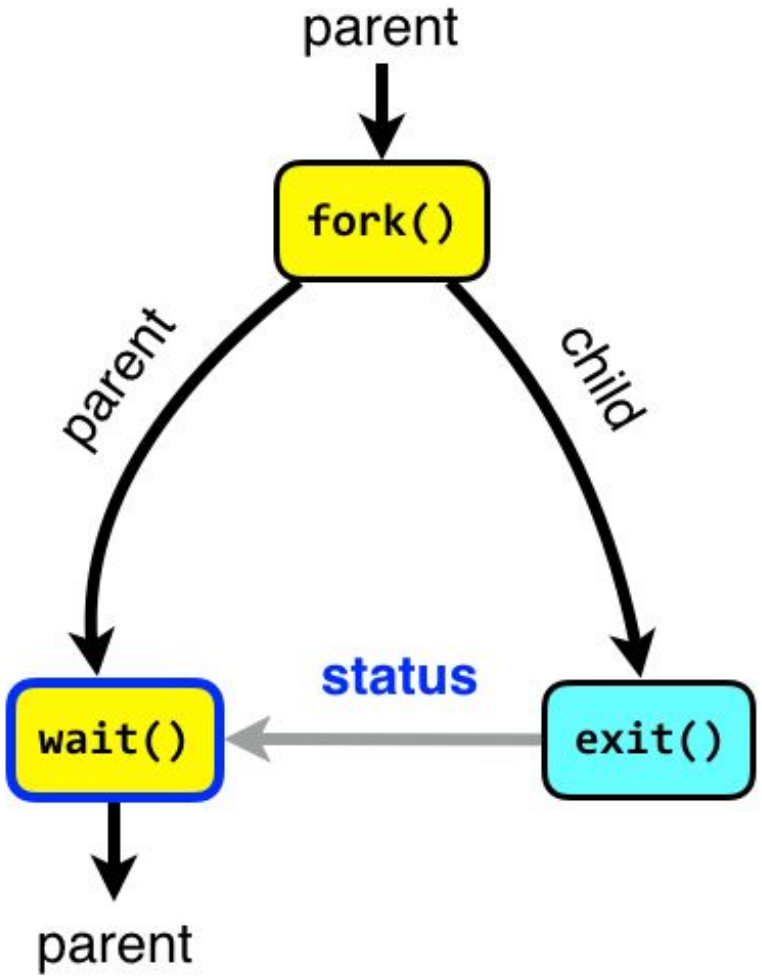
Orphans

- An orphan process is a process **whose parent process has terminated**, though it remains running itself.
- Any orphaned process will be **immediately adopted by the special init system process** with PID 1.
- Processes execute **concurrently**
- Both the parent process and the child process competes for the CPU with all other processes in the system.
- The operating systems decides which process to execute when and for how long. The process in the system execute concurrently.
- In our example program:
  - most often the parent terminates before the child and the child becomes an orphan process adopted by init (PID = 1) and therefore reports PPID = 1
  - sometimes the child process terminates before its parent and then the child is able to report PPID equal to the PID of the parent.

Wait

- The wait system call blocks the caller until one of its child process terminates.
- If the caller doesn't have any child processes, wait returns immediately without blocking the caller.
- Using wait the parent can obtain the exit status of the terminated child.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```



On success, wait returns the PID of the terminated child.
On failure (no child), wait returns -1.

Zombies

- A terminated process is said to be a zombie or defunct until the parent does wait on the child.
- When a process terminates all of the memory and resources associated with it are deallocated so they can be used by other processes.
- However, the exit status is maintained in the PCB until the parent picks up the exit status using wait and deletes the PCB.
- A child process always first becomes a zombie.
- In most cases, under normal system operation zombies are immediately waited on by their parent.
- Processes that stay zombies for a long time are generally an error and cause a resource leak.