System Calls / Process

Interface to the OS

- Shells allow a user to interface with the operating system.
- A second strategy is through a user friendly graphical user interface (GUI).

Interface to the OS

- Executable will need OS services.
  - System.out.println; printf; scanf.
- This is done through a system call.
  - Program passes relevant information to OS.
  - OS performs the service if
    - The OS is able to do so, and
    - The service is permitted for this program at this time
- System calls are typically written in C and C++.
  - Tasks that require hardware to be accessed directly may be written using assembly language.

Interface to the OS

- It is important to note that as programmers we actually call a function in a library (system function) which actually make the system call.
- We will often use the terms interchangeably.

Interface to the OS

- Let's say that a user program has the following line of code:

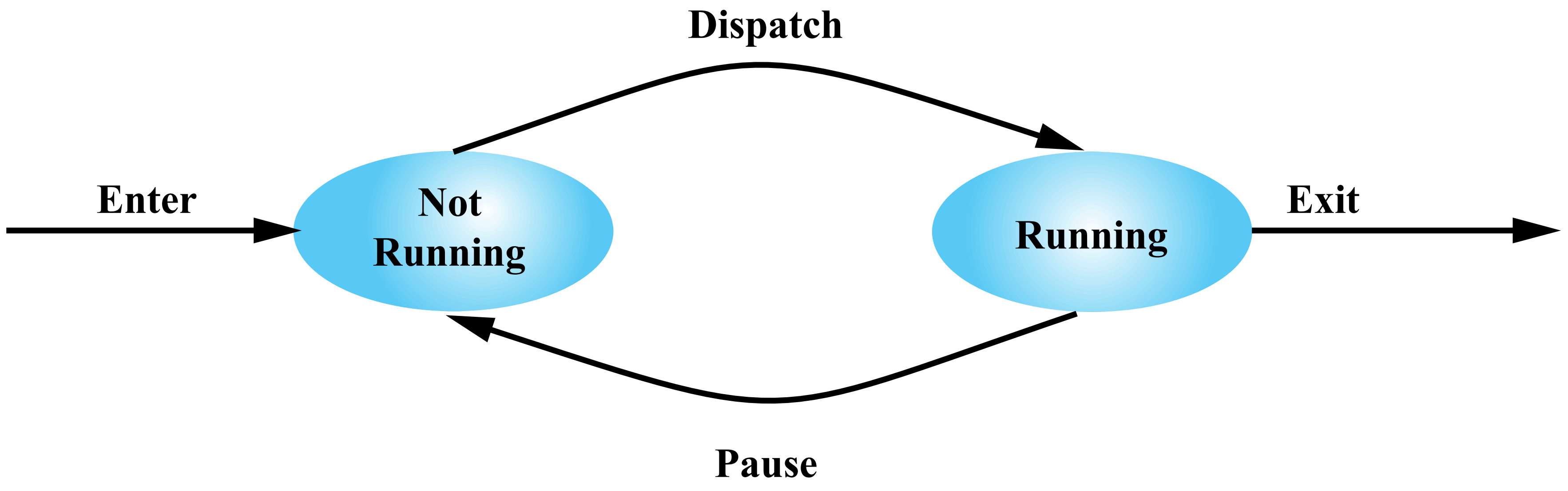
  ```
  count = read(fd, buf, nbytes)
  ```

- Some issues to be addressed:
  - How are parameters passed?
  - How are results provided to the user program?
  - How is control given to the system call and the operating system?
- To understand how these issues are addressed we will briefly focus on processes and CPUs.

What is a Process?

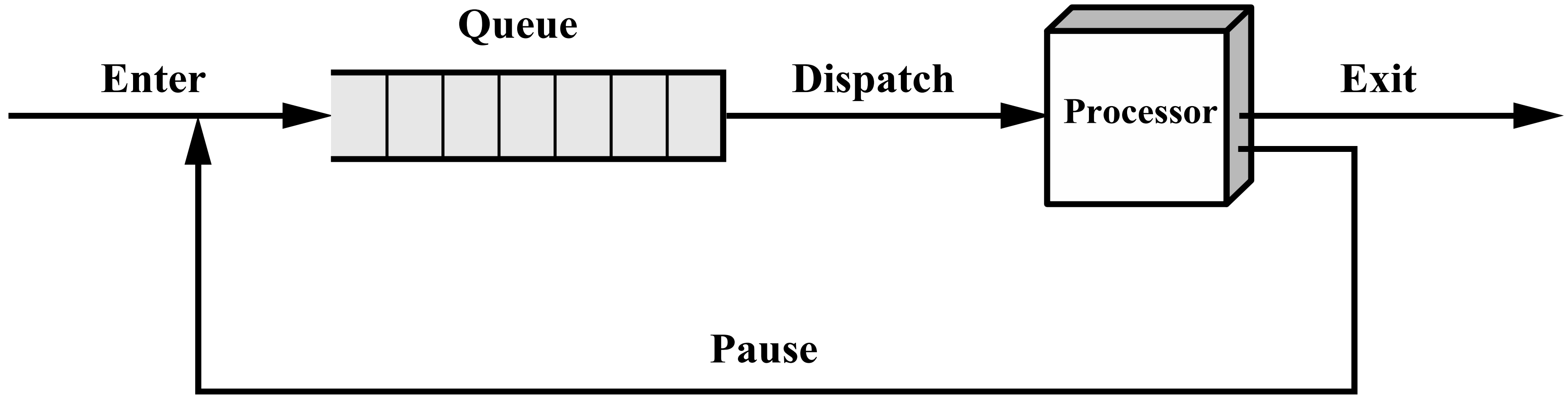
- A process is a program in execution.
  - Program is passive. File containing a list of instructions stored on disk.
  - Process is active. A program counter specifying the next instruction to execute.
  - A program becomes a process when an executable file is loaded into memory.
- Address space of a process is the memory locations which the process can read or write.

Two-State Process Model



(a) State transition diagram

Two-State Process Model (cont.)



(b) Queuing diagram

Figure 3.5 Two-State Process Model

Five-State Process Model

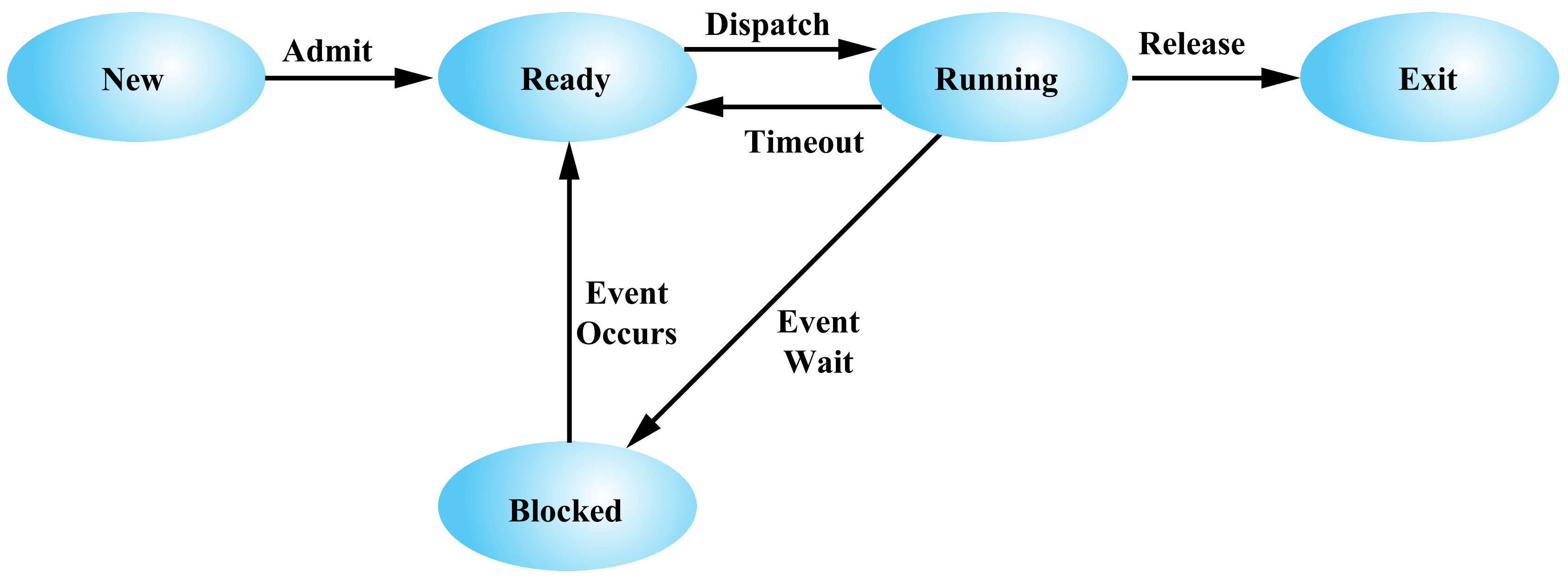


Figure 3.6 Five-State Process Model

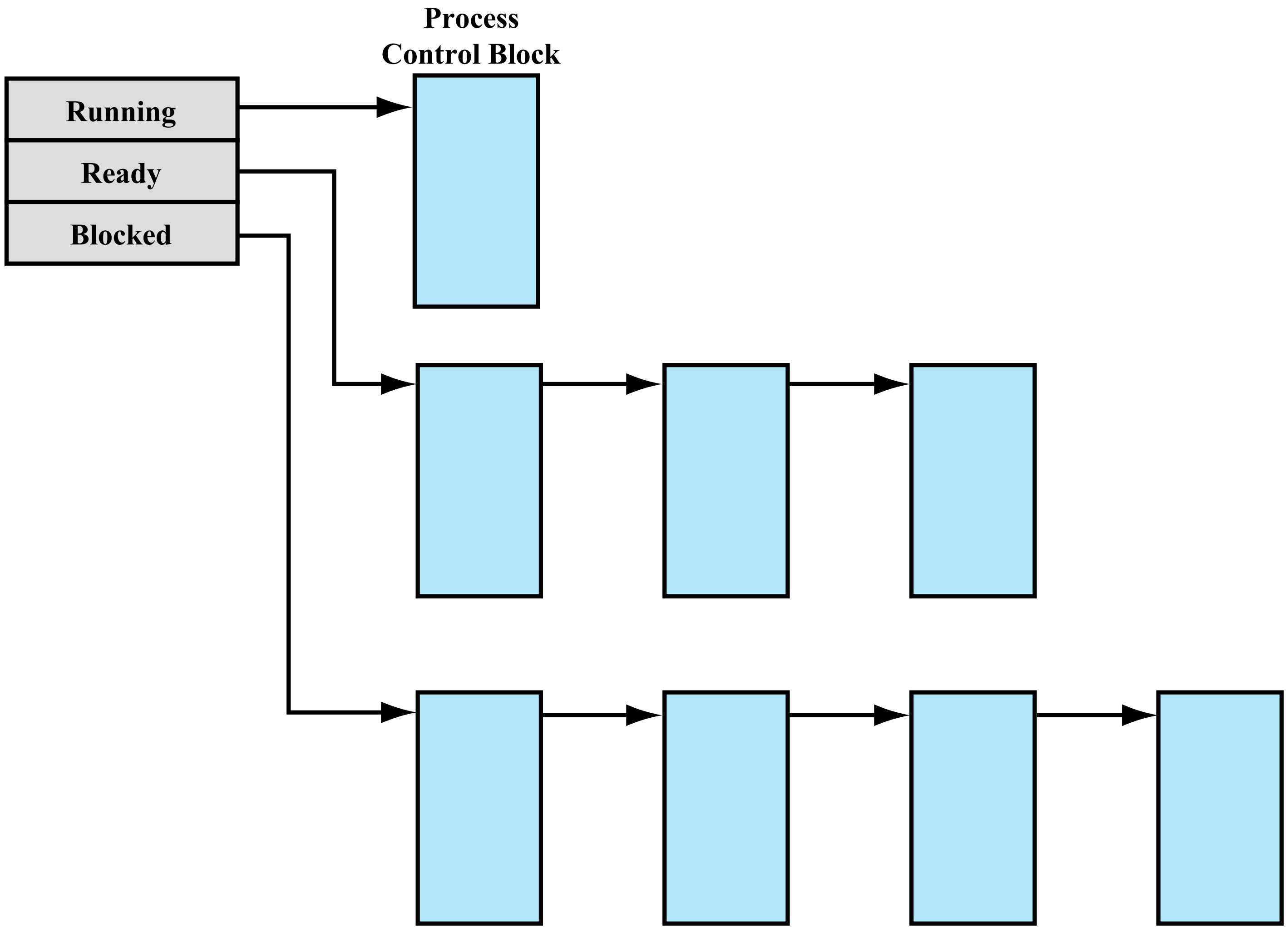


Figure 3.14 Process List Structures

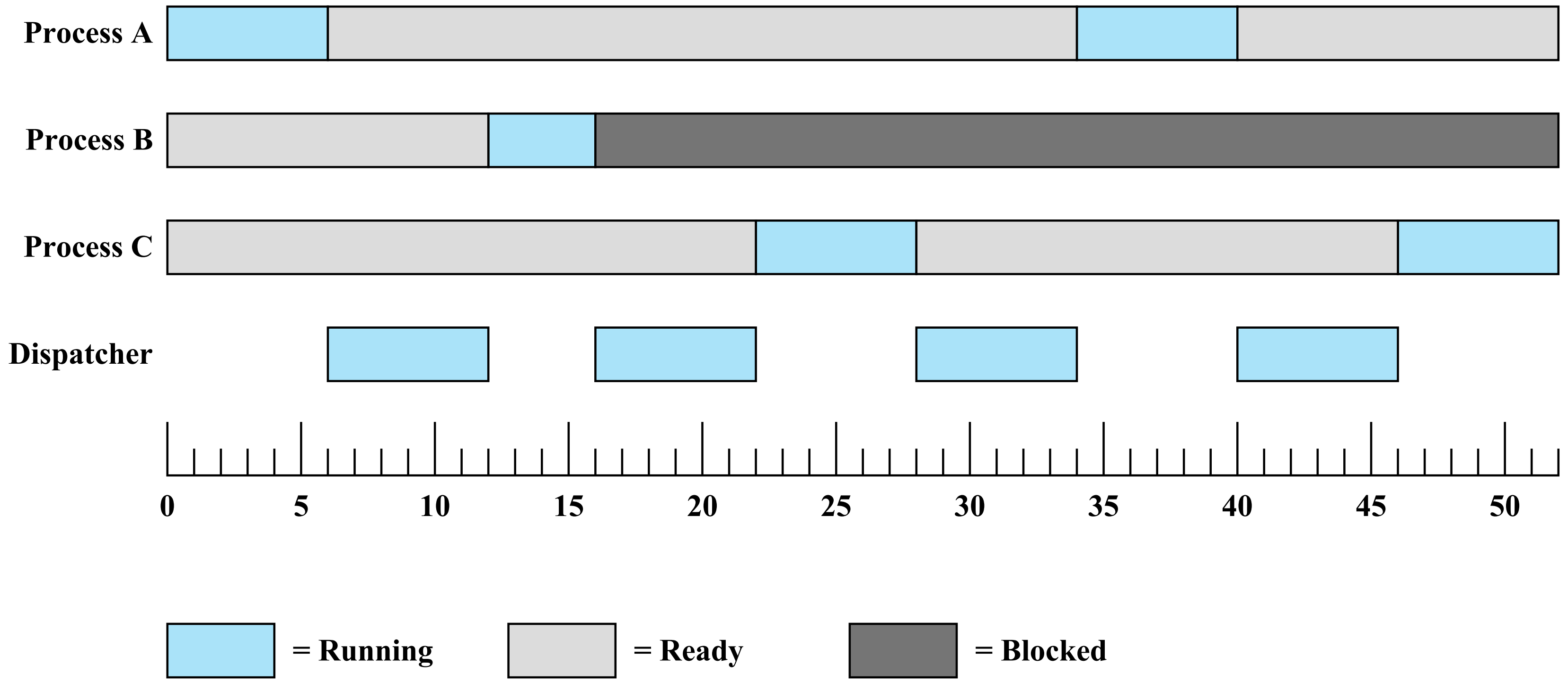
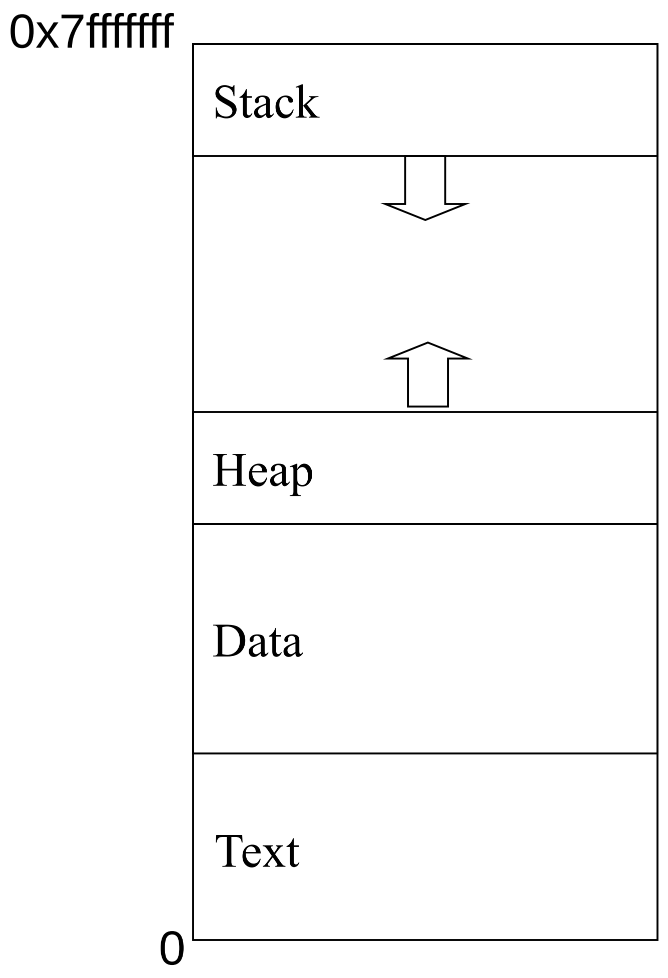


Figure 3.7 Process States for Trace of Figure 3.4

Memory Image of a Unix Process



- Processes have different types of memory segments:
  - Text: program code.
  - Data:
    - Statically declared variables.
  - Heap
    - Areas allocated by malloc() or new (heap).
  - Stack
    - Automatic variables.
    - Function and system calls.
- Invoking the same program multiple times results in the creation of multiple distinct address spaces.

Memory Image of a Unix Process

- Text segment: the machine instructions that the CPU executes.
- The text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells.
- The text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Memory Image of a Unix Process

- Data segment: contains variables that are specifically initialized in the program.
  - Example: the C declaration (appearing outside any function).

```
int maxcount = 99;
```

Memory Image of a Unix Process

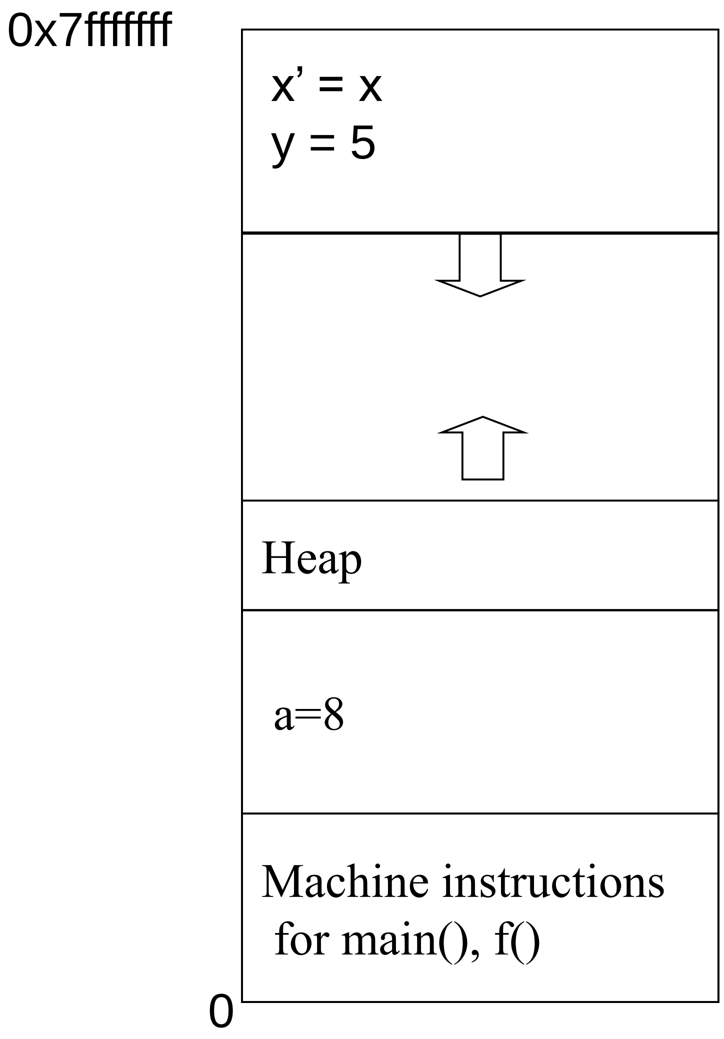
- Stack:
  - Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack.
  - The newly called function then allocates room on the stack for its variables.
  - Question: What if you have recursive functions?

Memory Image of a Unix Process

- Heap
  - For dynamic memory allocation
  - Example: malloc
    - Allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
  - Historically, the heap has been located between the uninitialized data and the stack.

Example

```
int a = 8;
int f (int x)
{
    int y = 5;
    x = x + 1;
    return x;
}
int main()
{
    int x, y;
    x=5;
    y=x+f(a);
}
```



Memory Image of a Unix Process

- The segments are not necessarily in a contiguous area of memory.
- Operating systems must manage the memory so that it appears to be contiguous.

CPU

- A CPU's instruction set defines the instructions that it can execute..
  - This is different for different CPU architectures.
  - All have load and store instructions for moving items between memory and registers.
  - Many instructions for comparing and combining values in registers and putting result into a register.

Parts of a CPU

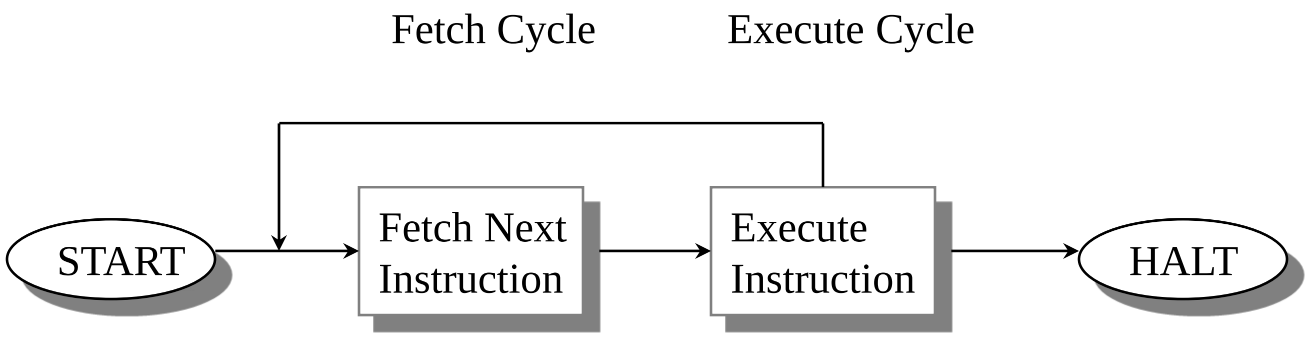
General registers hold key variables and temporary results.

Special registers visible to the programmer e.g.,

- Program Counter (PC) holds the memory address of the next instruction,
- Instruction Register (IR) holds the instruction currently being executed,
- Processor Status Word (PSW) contains control bits including the mode bit.

Basic Instruction Execution

OS loads program (executable) into memory.
OS loads the memory address of the program's starting instruction into the PC register.
A CPU fetches, decodes and executes instructions.
Fetches next instruction pointed to by PC.



Processor Modes

- CPUs have a mode bit in PSW that defines execution capability of a program.
  - Supervisor mode (mode bit set).
    - Executes every instruction.
    - Operating systems run in this mode.
    - Sometimes called kernel mode.
  - User mode (mode bit cleared).
    - Executes a subset of instructions.
    - User applications run in this mode.

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 kernel mode.
- Kernel of the operating system.

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.

Exceptions

- Exception
  - An abrupt change in control flow in response to a change in processor state.
- Examples:
  - Application program:
    - Requests I/O.
    - Requests more heap memory.
    - Attempts integer division by 0.
    - Attempts to access privileged memory.
    - Accesses variable that is not in real memory (see upcoming "Virtual Memory" lecture).

    Synchronous
  - User presses key on keyboard.
  - Disk controller finishes reading data.

    Asynchronous

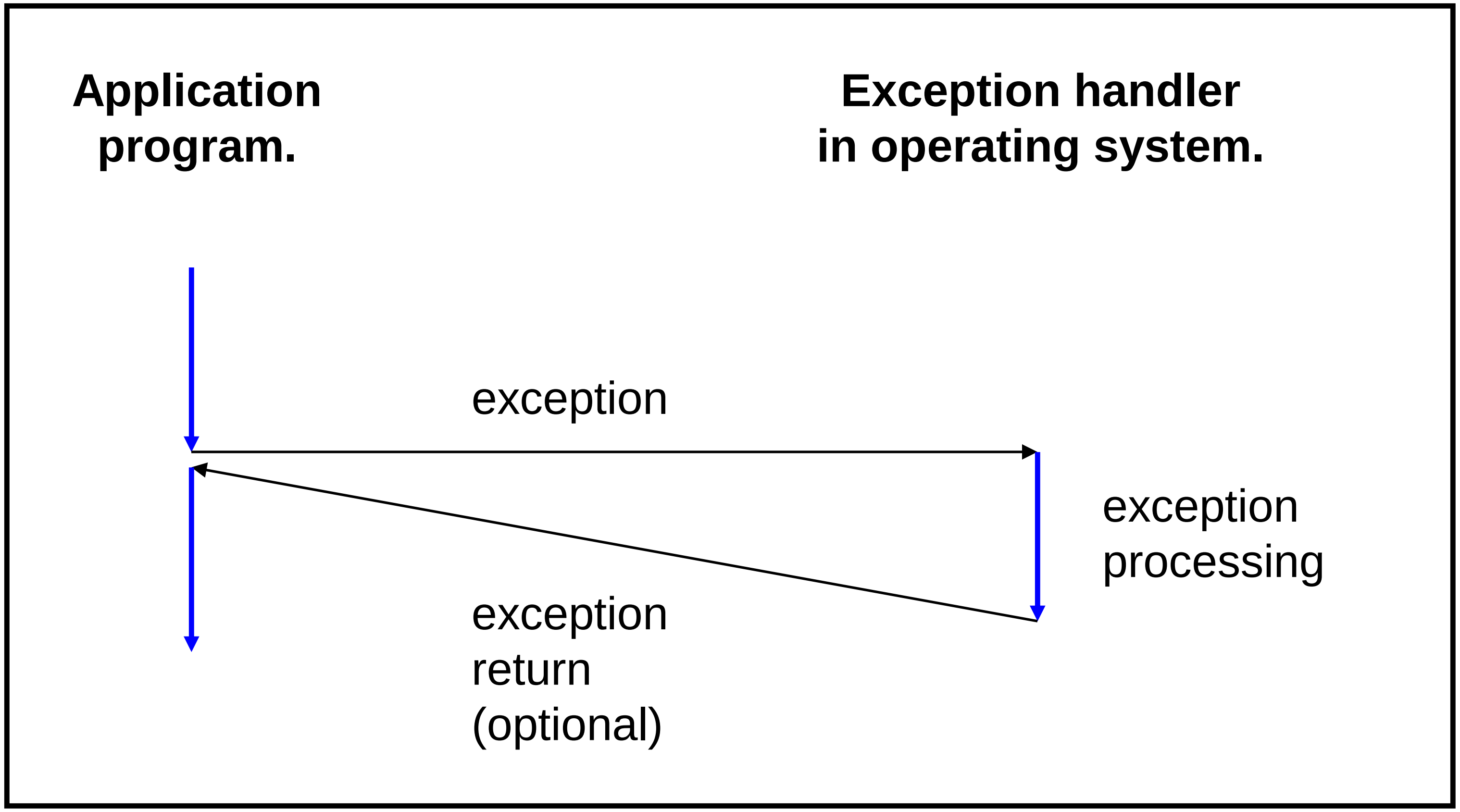
Exceptions Note

- Note:

  Exceptions in OS ≠ exceptions in Java.

Implemented using
`try/catch`
and **`throw`** statements.

Exceptional Control Flow


Application program.
Exception handler in operating system.
exception
exception processing
exception return (optional)

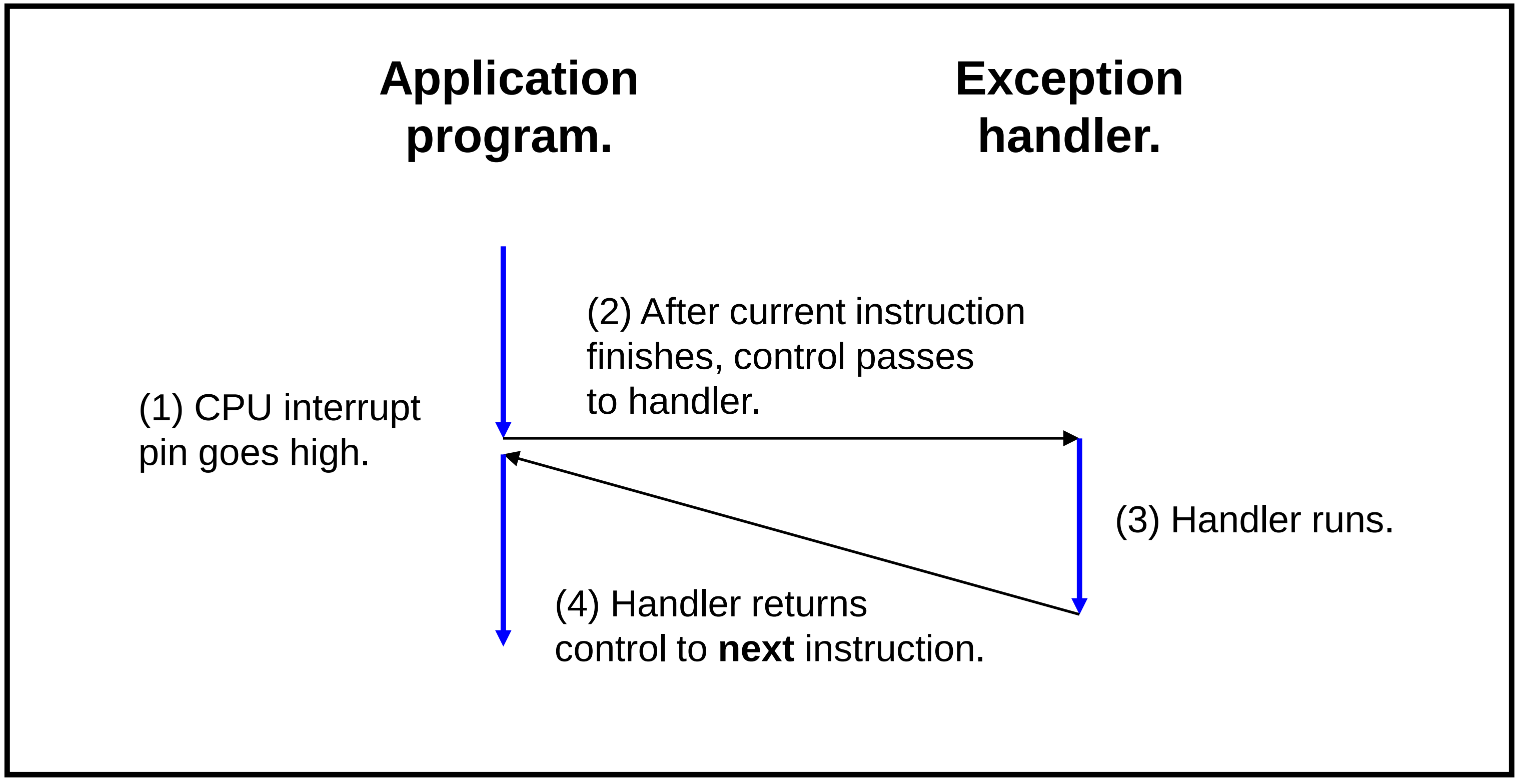
Exceptions vs. Function Calls

- Exceptions are **similar to** function calls.
  - Control transfers from original code to other code.
  - Other code executes.
  - Control returns to original code.
- Exceptions are **different from** function calls.
  - Processor pushes **additional state** onto stack.
    - E.g. values of *all* registers (including FLAGS).
  - Processor pushes data onto **OS's stack**, not application's stack.
  - Handler runs in **privileged mode**, not in **user mode**.
    - Handler can execute all instructions and access all memory.
  - Control **might return** to next instruction.
    - Control sometimes returns to **current** instruction.
    - Control sometimes does not return at all!

Classes of Exceptions

- There are four classes of exceptions.
  - Interrupts
  - Traps
  - Faults
  - Aborts

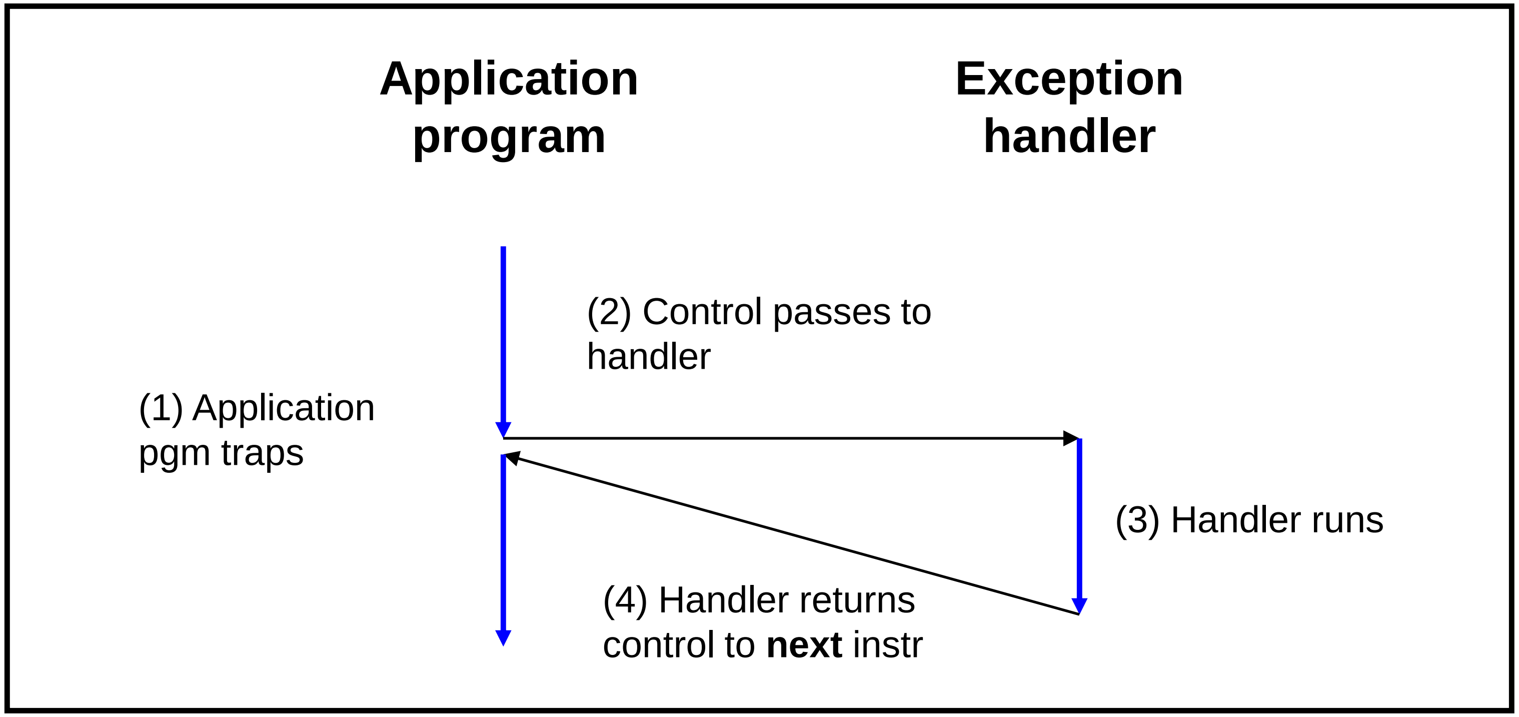
(1) Interrupts


Application program.
Exception handler.
(1) CPU interrupt pin goes high.
(2) After current instruction finishes, control passes to handler.
(3) Handler runs.
(4) Handler returns control to **next** instruction.

Cause: Signal from I/O device.
Examples:
User presses key.
Disk controller finishes reading/writing data.
Timer to trigger another application to run.

An alternative to wasteful polling!

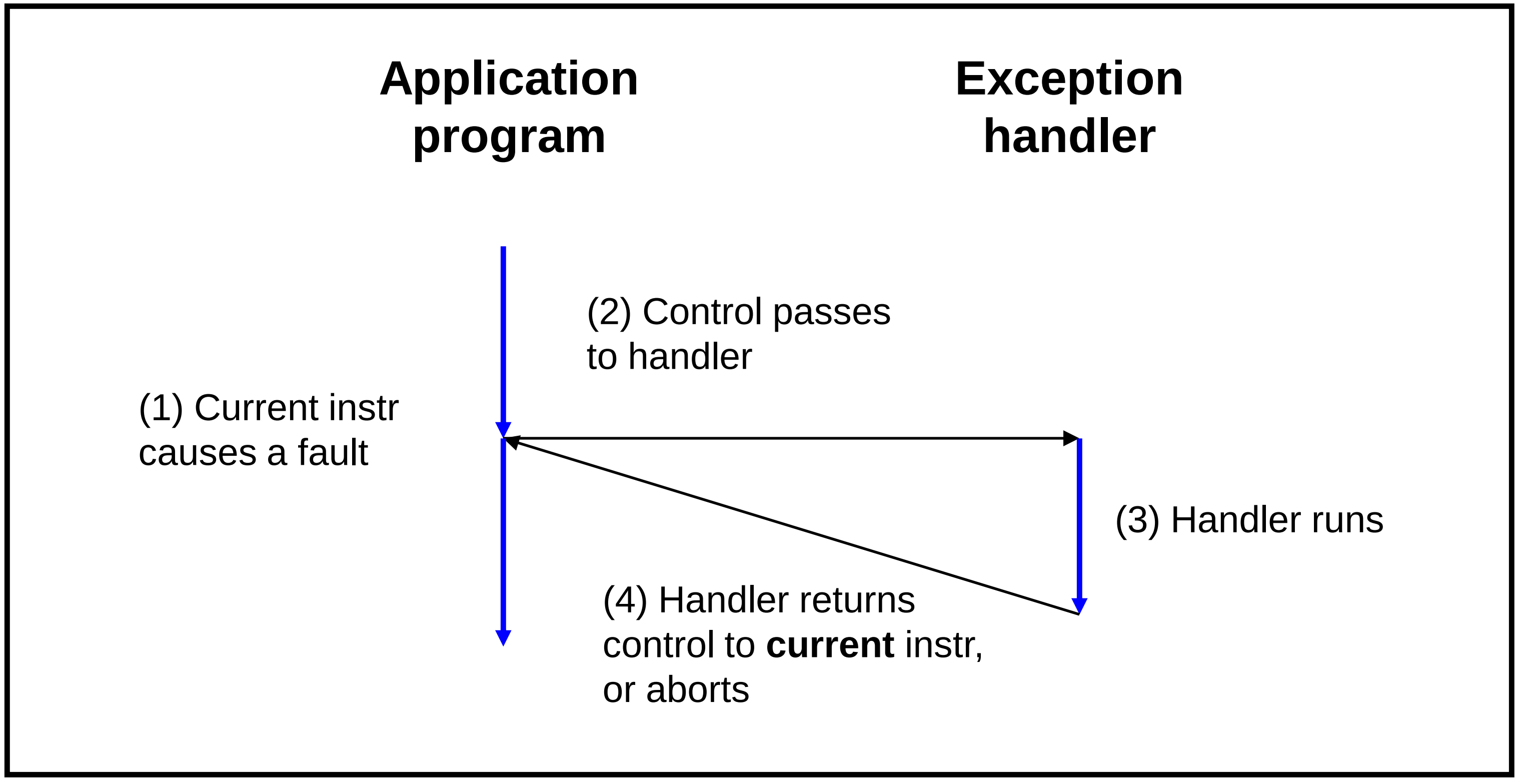
(2) Traps


Application program
Exception handler
(2) Control passes to handler
(1) Application pgm traps
(3) Handler runs
(4) Handler returns control to **next** instr

Cause: Intentional (application program requests OS service).
Examples:
Application program requests more heap memory.
Application program requests I/O.

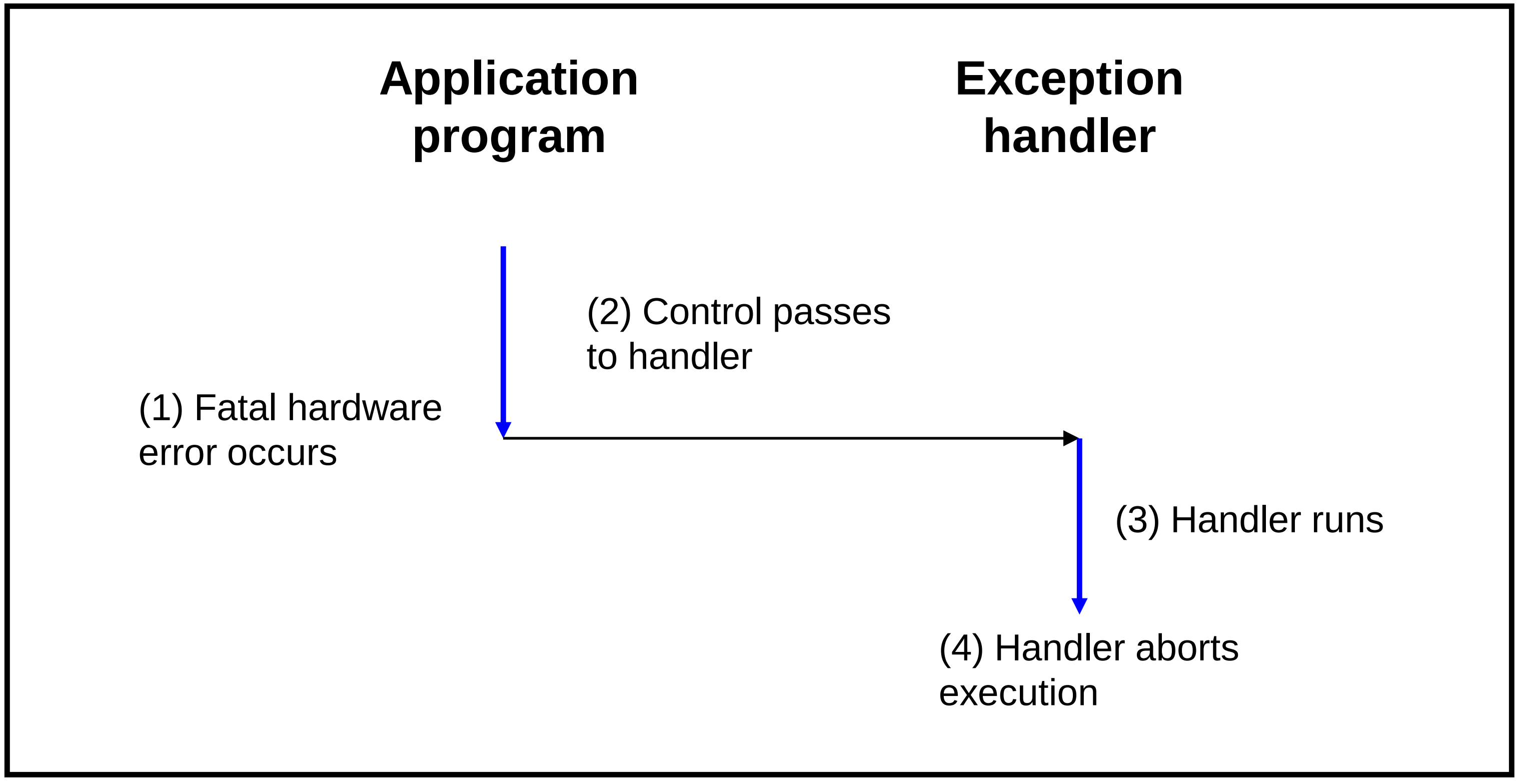
Traps provide a function-call-like interface between application and OS.

(3) Faults


Application program
Exception handler
(2) Control passes to handler
(1) Current instr causes a fault
(3) Handler runs
(4) Handler returns control to **current** instr, or aborts

Cause: Application program causes (possibly) recoverable error.
Examples:
  Application program accesses privileged memory (segmentation fault).
  Application program accesses data that is not in real memory (page fault).

(4) Aborts


Application program
Exception handler
(2) Control passes to handler
(1) Fatal hardware error occurs
(3) Handler runs
(4) Handler aborts execution

Cause: Non-recoverable error.
Example:
Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.).

Summary of Exception Classes

Class	Cause	Asynch/Synch	Return Behavior
Interrupt	Signal from I/O device	Asynch	Return to next instr
Trap	Intentional	Sync	Return to next instr
Fault	(Maybe) recoverable error	Sync	(Maybe) return to current instr
Abort	Non-recoverable error	Sync	Do not return

Exceptions in Intel Processors

Each exception has a number.
Some exceptions in Intel processors:

Exception #	Exception
0	Fault: Divide error
13	Fault: Segmentation fault
14	Fault: Page fault
18	Abort: Machine check
32-127	Interrupt or trap (OS-defined)
128	**Trap**
129-255	Interrupt or trap (OS-defined)

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:

↓

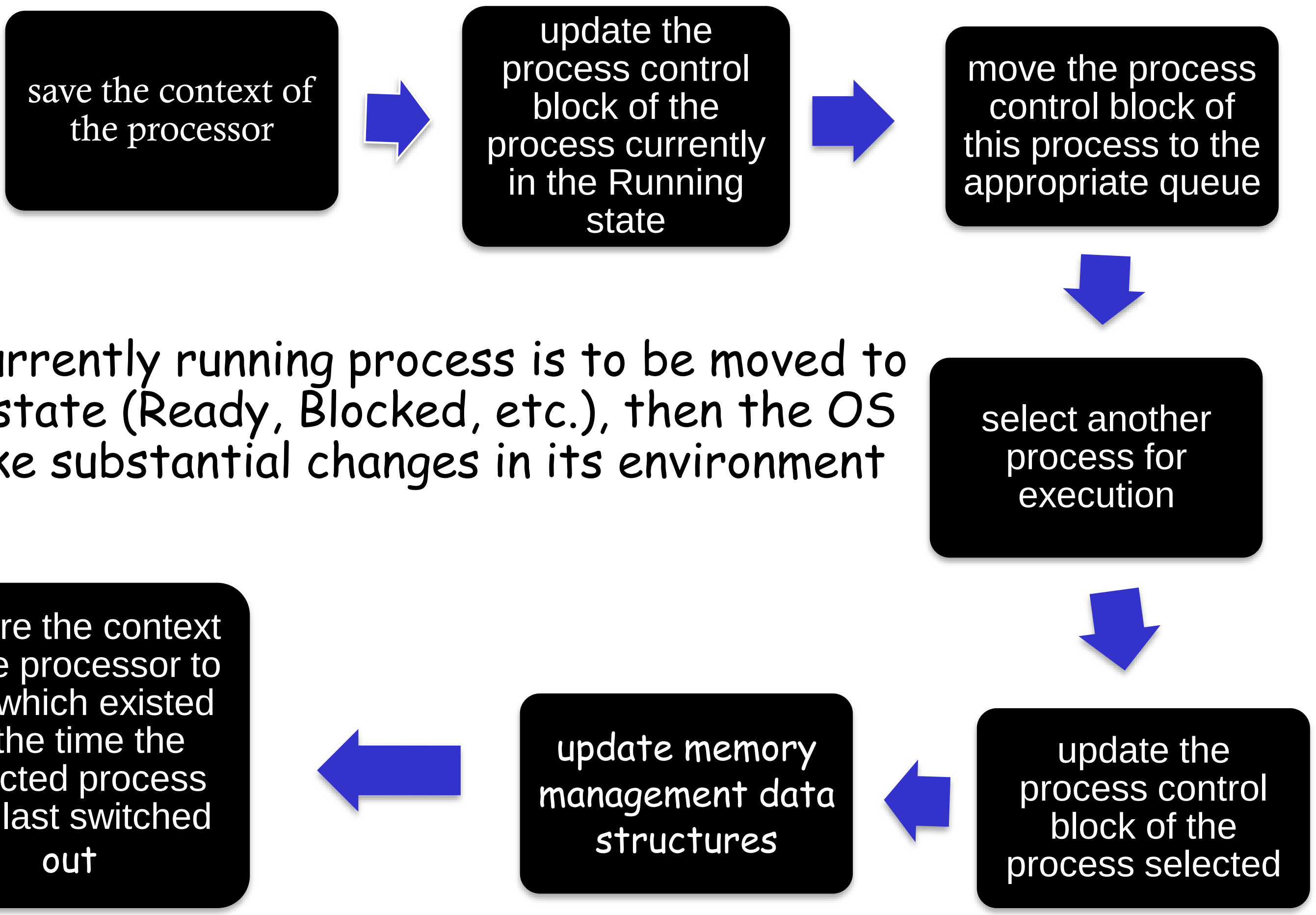
sets the program counter to the starting address of an interrupt handler program

↓

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:


save the context of the processor
update the process control block of the process currently in the Running state
move the process control block of this process to the appropriate queue
select another process for execution
update the process control block of the process selected
update memory management data structures
restore the context of the processor to that which existed at the time the selected process was last switched out

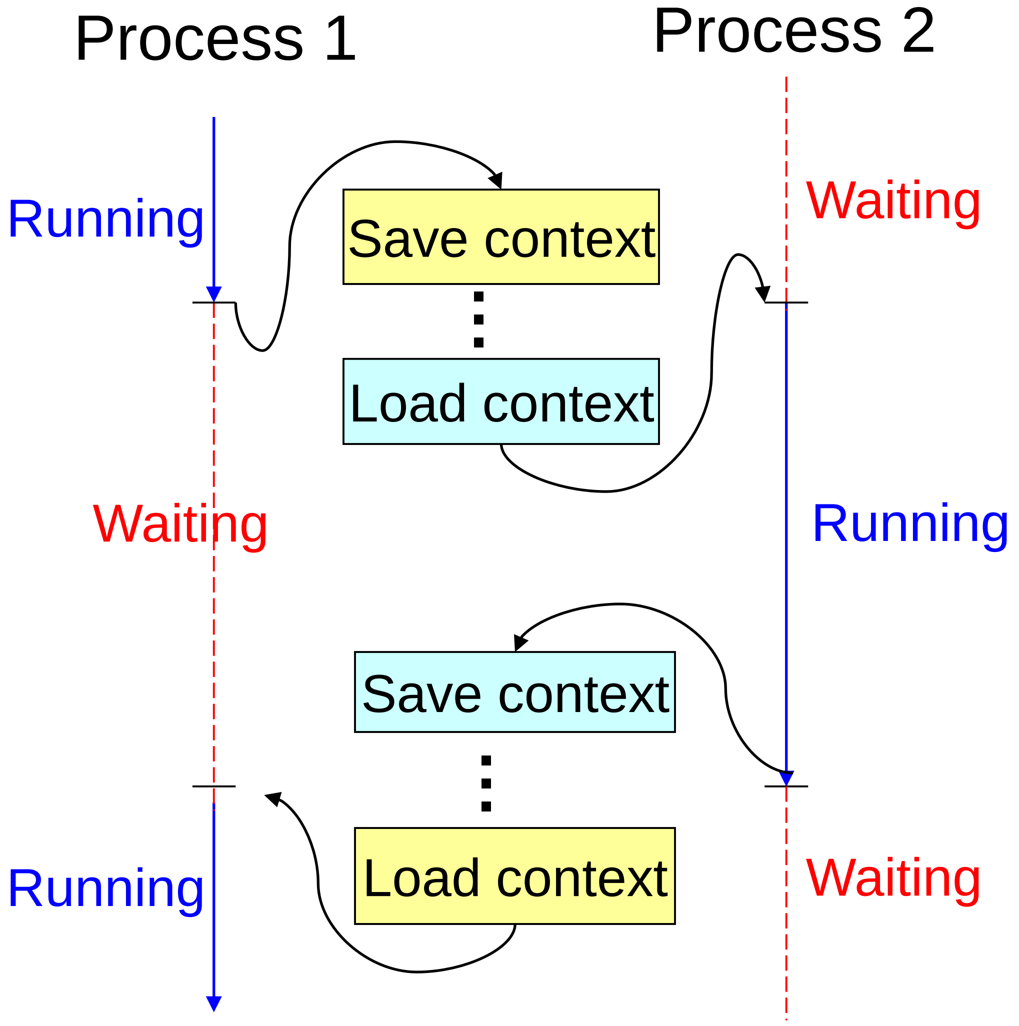
If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

Processor Modes

- Instructions that execute only in supervisor mode are called Privileged Instructions.
- In general, these Privileged Instructions affect entire machine.
- Which of these instructions should run in supervisor mode?
  - Read time-of-day instruction.
  - Set time-of-day clock.
  - Directly access the printer.
  - Open a file for read.

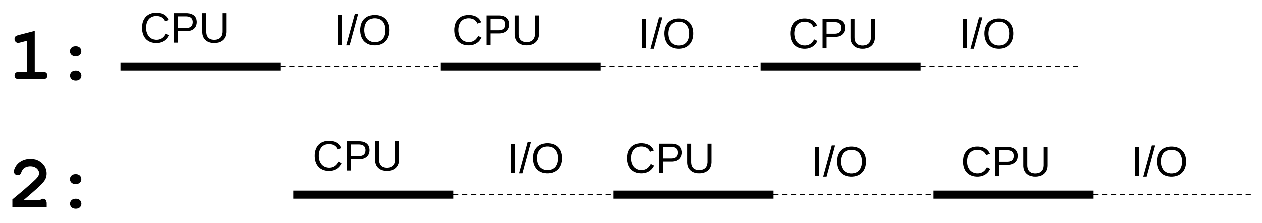
Context Switch Details

- Context
  - State the OS needs to restart a preempted process.
- Context switch
  - Save the context of current process.
  - Restore the saved context of some previously preempted process.
  - Pass control to this newly restored process.



When Should OS do Context Switch?

- When a process is stalled waiting for I/O.
  - Better utilize the CPU, e.g., while waiting for disk access.



- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine.
  - Trade-off efficiency for a finer granularity of fairness.

OS Runs in Supervisor Mode

- When applications need to run privileged instructions, they must call the OS.
  - This is what you are doing with the shell program.
- A user program cannot run privileged instructions since user programs execute only in user mode.
- There is an implicit assumption that the OS is trusted but applications are not.

The inner workings of a system call

User-level code

```
usercode
 {
  ...
  read (file, buffer, n);
  ...
 }
```

Library code

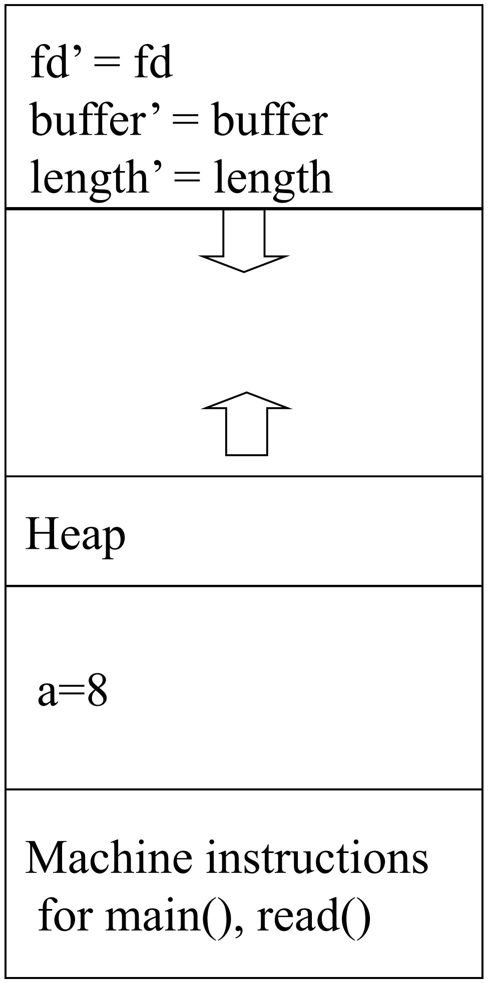
```
Procedure read(file, buff, n)
 {
  ...
  read(file, buff, n)
  ...
 }

_read:
 LOAD r1, @SP+2
 LOAD r2, @SP+4
 LOAD r3, @SP+6
 TRAP Read_Call                    //System Call
```

System Calls and Traps

- TRAP instruction switches CPU to supervisor mode.
    - Executes predetermined OS instruction.
    - The state of the user process is saved so that the OS instructions needed can be executed (system handler).
    - When the system handler finishes execution then the user code can execute.

Making a System Call

- System call: read(fd,buffer,length)
- Step 1:
  - The calling program pushes the parameters onto the stack.
  - This is done for preparation for calling the read library procedure which actually makes the system call.



Making a System Call

- System call:
  read(fd,buffer,length)
- Step 2: The input parameters are passed in registers.
- Step 3: The trap code for read is put into a register.
  - The code tells the OS what system call handler (kernel code) to execute.
- Step 4: TRAP instruction is executed.
  - This causes a switch from the user mode to the kernel mode.

Making a System Call

- System call:
  read(fd,buffer,length)
- Step 5:
  - Read the register.
  - The OS is able to map register contents to the location of the system call handler.
- Step 6: System call handler code is executed.
- Step 7: After execution control returns to the library procedure.

System Call

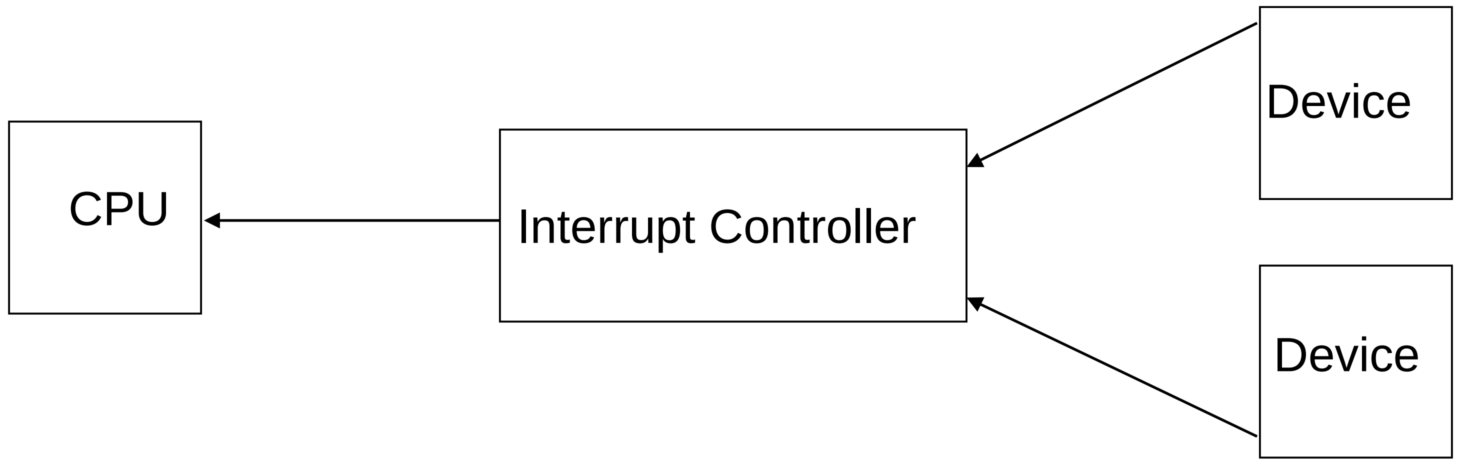
- The system call handler will have to actually wait for data from the disk.
- Reading data from disk is much slower then memory.
- We do not want the CPU to be idle while waiting for data from the disk.
  - Most operating systems allow for another executing program to use the CPU.
  - This is called multiprogramming.
- How does a process find out about reading being completed?

Polling

- What if we have the CPU periodically check the disk controller to see if it is ready to put read data into memory?
- What if the device is not in use or used infrequently?
- Polling is like picking up your phone every few minutes to see if you have a call.
- What if instead you wait to be signaled i.e., wait for an interrupt?

Interrupts

- Give each device a wire (interrupt line) that is used to signal the processor when data transfer is complete.
  - When interrupt is signaled, processor executes a routine called the interrupt handler to deal with the interrupt.
  - No overhead when there is no device ready to be serviced.



Interrupts and Interrupt Handlers

- Interrupt signal is sent over the bus.
- CPU has a table with the entries for each device type which is the address of the first instruction of the code to be executed to handle the interrupt.
- Instruction starts execution of an "interrupt handler" procedure in OS.

Interrupts and Interrupt Handlers

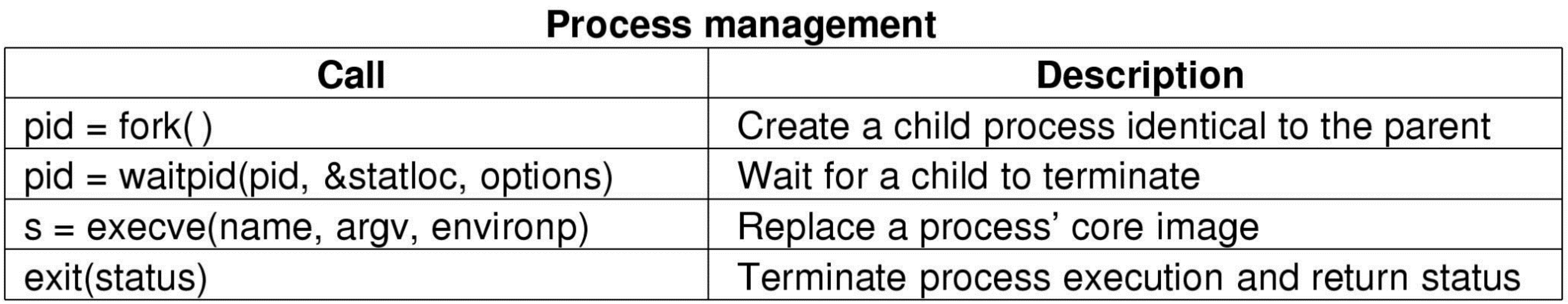
- Interrupt handler.
  - Saves processor state: CPU registers of interrupted process are saved into a data structure in memory.
  - Runs code to handle the I/O.
  - Restores processor state: CPU registers values stored in memory are reloaded.
  - PC is set back to PC of interrupted process.

On all current computers, at least part of the interrupt handlers are written in assembly language. Why?

System Calls and Library Procedures

- The next three slides list some of the most heavily used POSIX (Portable Operating System Interface) procedure calls.
- POSIX is a standard for procedure calls.
  - Specifies an interface.
  - Interface implemented by many different operating systems.

Some System Calls For Process Management



Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some System Calls For File Management

File management

Call	**Description**
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some System Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Summary

- We discussed the implementation of system calls.
- We introduced the concept of process.