UNIVERSITY OF BIRMINGHAM

Computer Systems
Process Management

Lecture Objectives

- To introduce the notion of a process - a program in execution, which forms the basis of all computation
- To describe the various operations on processes, including scheduling, creation and termination
- To understand different ways in which an operating system may be executed

Lecture Outline

- Process Concept
- Process States
- Process Scheduling
- Operations on Processes
- Execution of the Operating System

The Process Concept

The concept of process is fundamental to the structure of modern computer operating systems. Its evolution in analyzing problems of synchronization, deadlock, and scheduling in operating systems has been a major intellectual contribution of the computer science.

WHAT CAN BE AUTOMATED? THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT PRESS, 1980

The Process Concept

"A process is a program in execution"

- Program is **passive** entity stored on disk (**executable file**), whereas a process is an **active** entity.
  - A program becomes a process when its executable file is loaded into memory.
- Execution of program may be started via:
  - GUI mouse clicks,
  - command line entry of its name etc.
- One program can be several processes
  - Consider multiple users executing the same program

Process Representation

Text

Three basic components

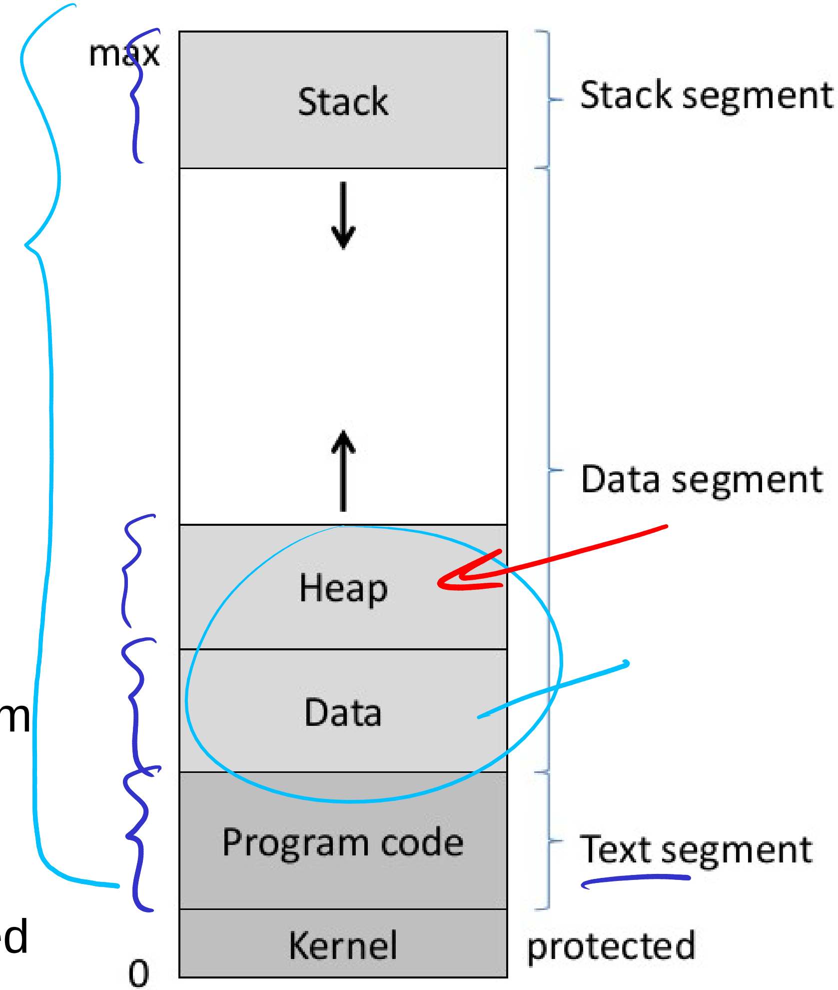
- Executable Program **Code**
- **Data** related to the Program
- Execution **Context**
  - Process ID, Group ID, User ID
  - Stack Pointer, Program Counter, CPU Registers
  - File Descriptors, Locks, Network Sockets
- Execution Context is essential for process switching.

Process Structure

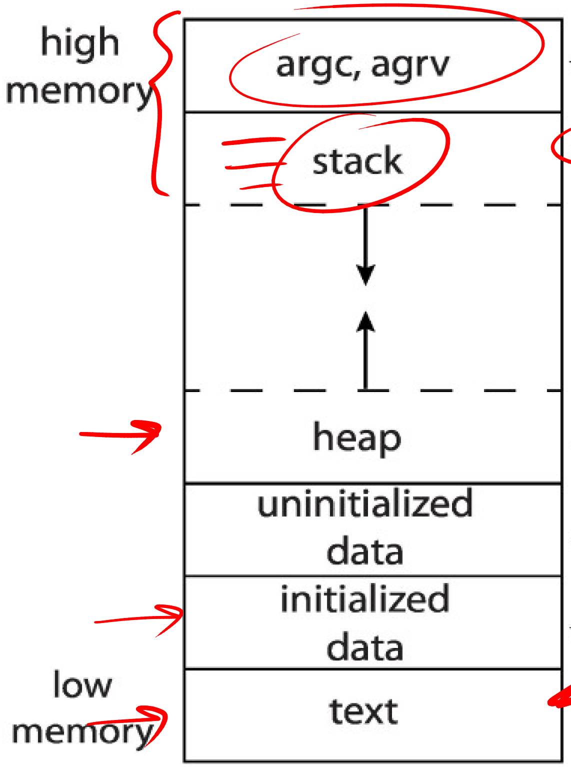
- A process is more than the program code, which is sometimes known as the **text** section.
- It also includes the current activity:
  - The value of the program counter
  - The contents of the processor's registers
- It also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables)
- It also includes the **data** section, which contains global variables.
- It may also include a **heap**, which is memory that is dynamically allocated during process run time.

Process Memory Layout

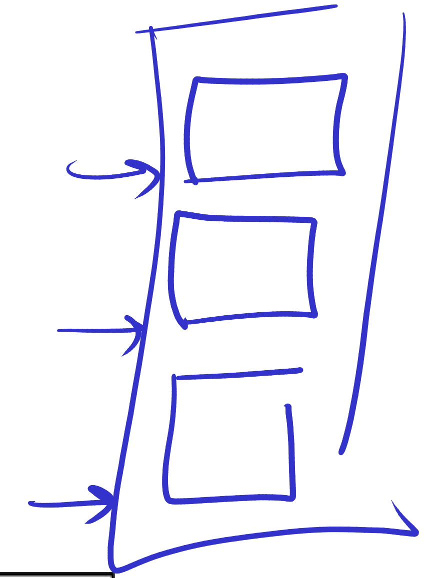
- Process Image
  - Layout of process in memory
- Segments
  - **Stack** Segment
    - Used for function calls
  - **Data** Segment
    - Static Variables, constants
    - Dynamic Allocation of memory from Heap
  - **Text** Segment
    - Contains the program code, shared between processes



Process Memory Layout

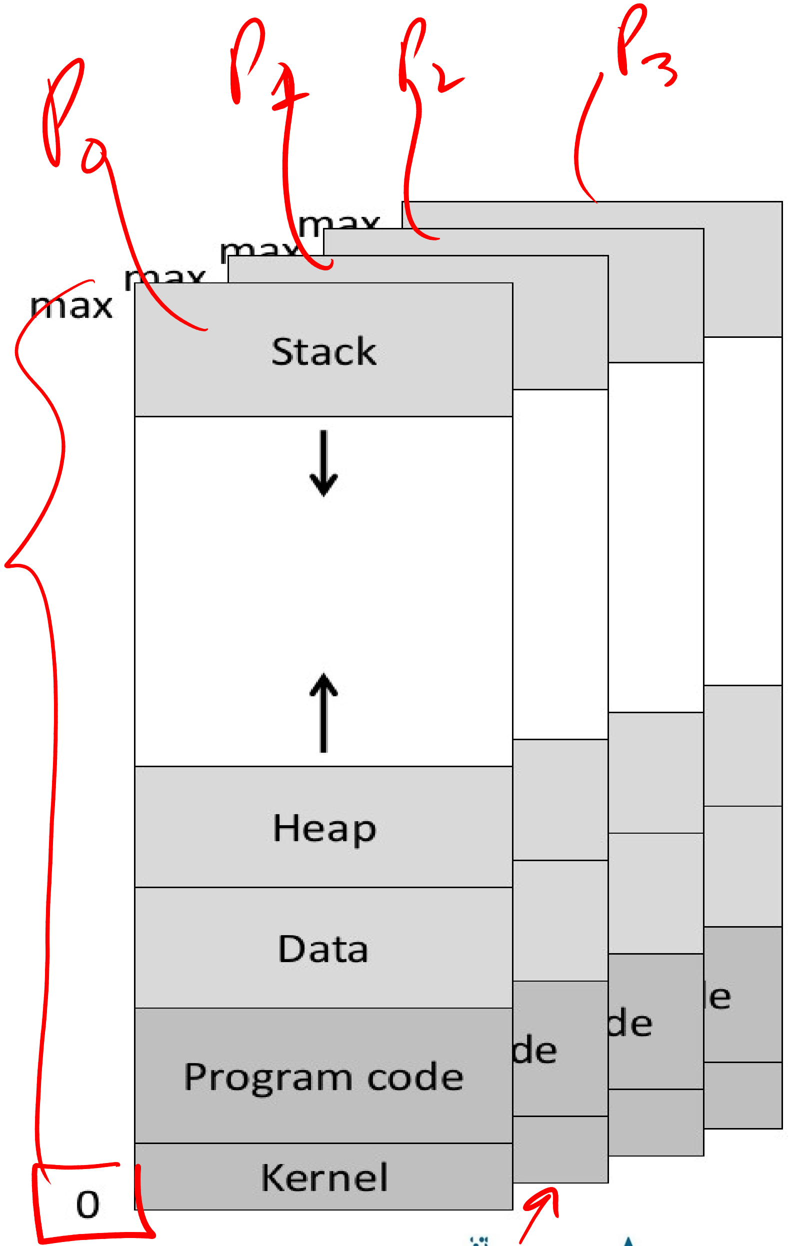



```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```



Virtualization

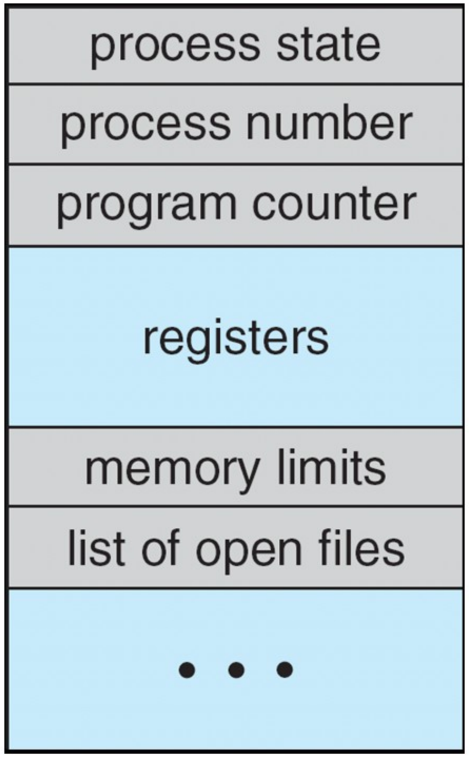
- Each program invoked results in the creation of a separate process, each with its own process image.
- Each process image appears to "own" the complete address space.
- Each process image starts at address 0 – How is this possible ?
- Virtual to Physical address mapping is required!



Process Control Block (PCB)

Information associated with each process (also called Task Control Block (TCB))

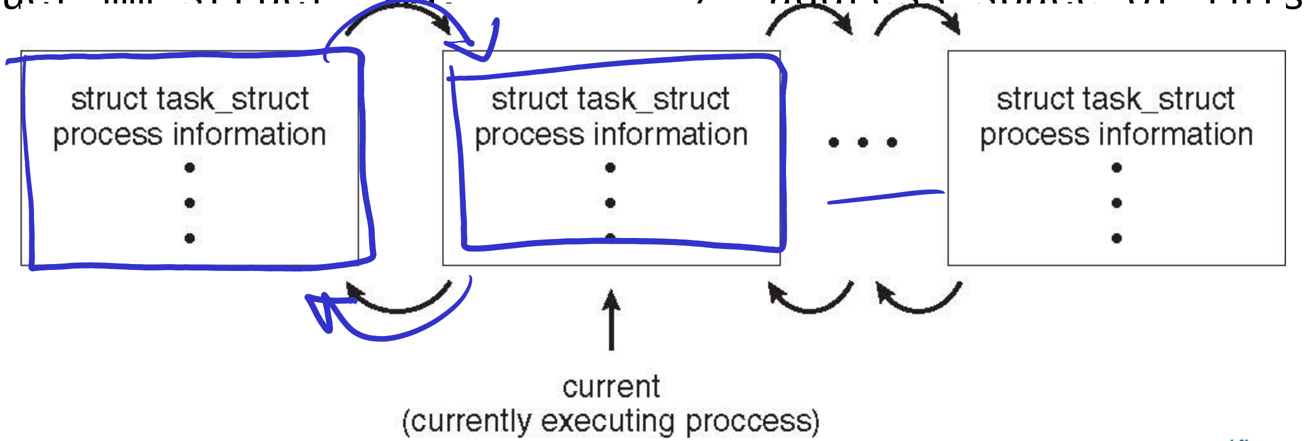
- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information** – priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to 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 Representation in Linux

Represented by the C structure **`task_struct`**

```
pid t_pid;                                   /* process identifier */
long state;                                  /* state of the process */
unsigned int time_slice                      /* scheduling information */
struct task_struct *parent;    /* this process's parent */
struct list_head children;     /* this process's children */
struct files_struct *files;    /* list of open files */
struct mm_struct *mm;          /* address space of this process */
```



Processes for Processes

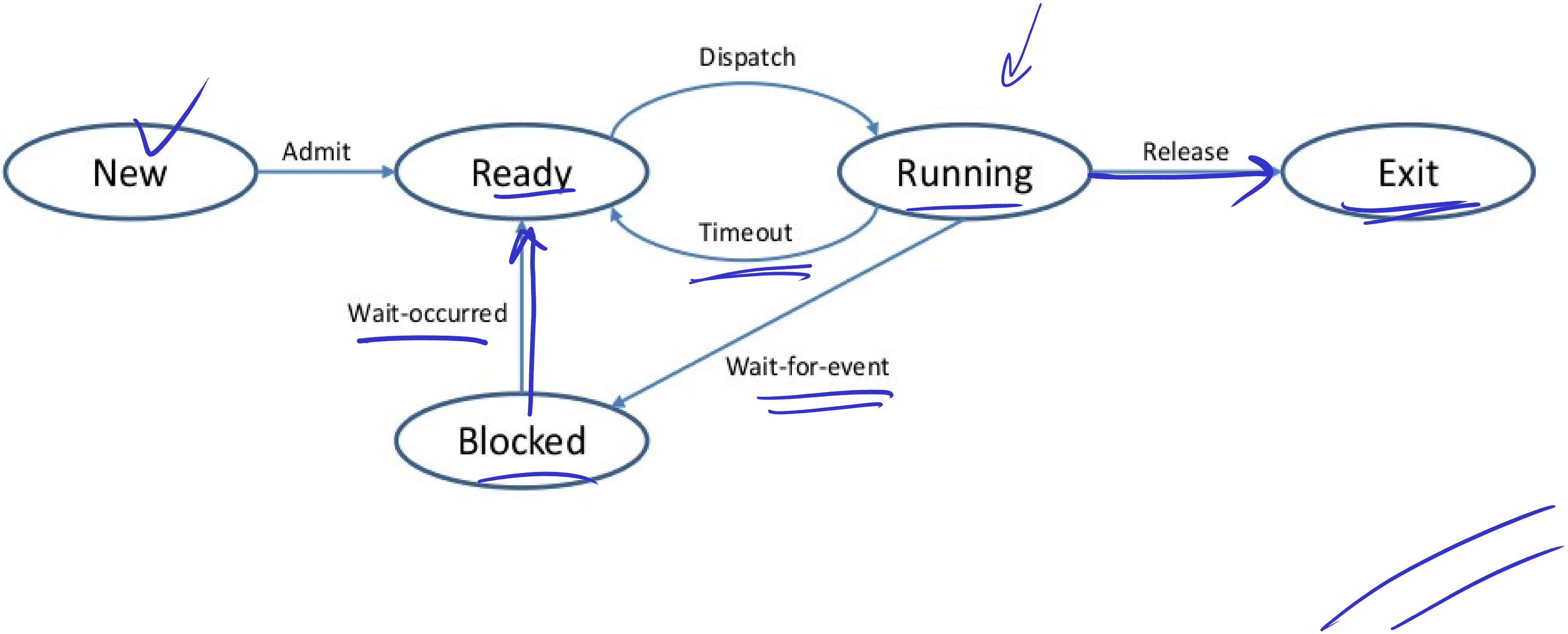
- Process is the execution environment for other code
- Executable Java program is executed within the Java virtual machine (JVM)
- JVM executes as a process that interprets the loaded Java code, and takes actions (via native machine instructions) on behalf of that code

Process States

As a process executes, it changes state

- **new:** The process is being created
- **ready:** The process is waiting to be assigned to a processor
- **running:** Instructions are being executed
- **waiting (blocked):** The process is waiting for some event to occur
- **terminated (exited):** The process has finished execution

Five-State Process Model



Threads

- So far, process has a single thread of execution
- Consider having multiple **program counters** per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Need storage for thread details, multiple program counters in PCB
- More on Threads (Later)

Process Scheduling

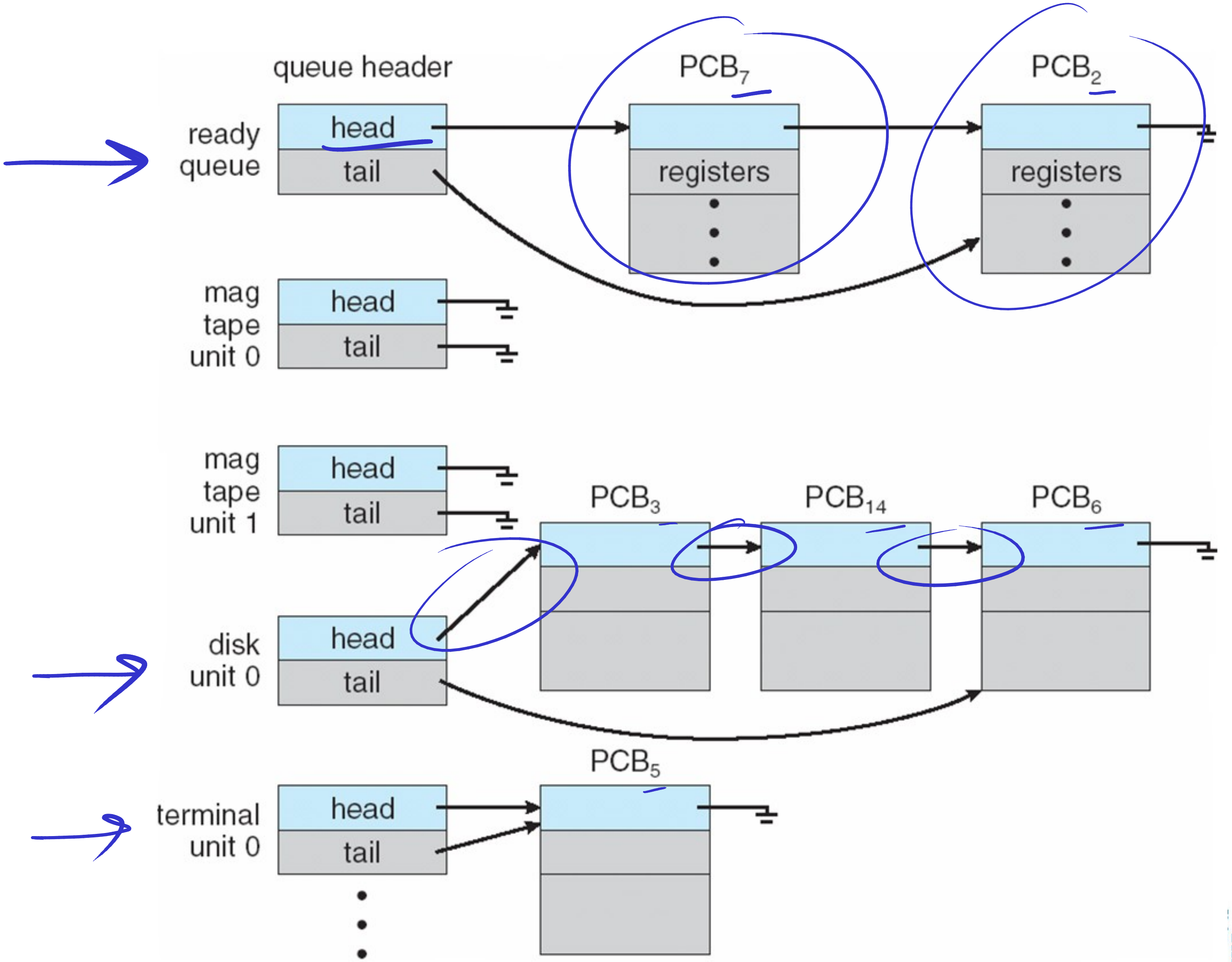
- Maximize CPU use
  - Quickly switch processes onto CPU for time sharing
- Process "gives" up then CPU under two conditions:
  - I/O request
  - After N units of time have elapsed (need a timer)
- Once a process gives up the CPU it is added to the "ready queue"
- **Process scheduler** selects among available processes in the ready queue for next execution on CPU

Scheduling Queues

OS Maintains **scheduling queues** of processes

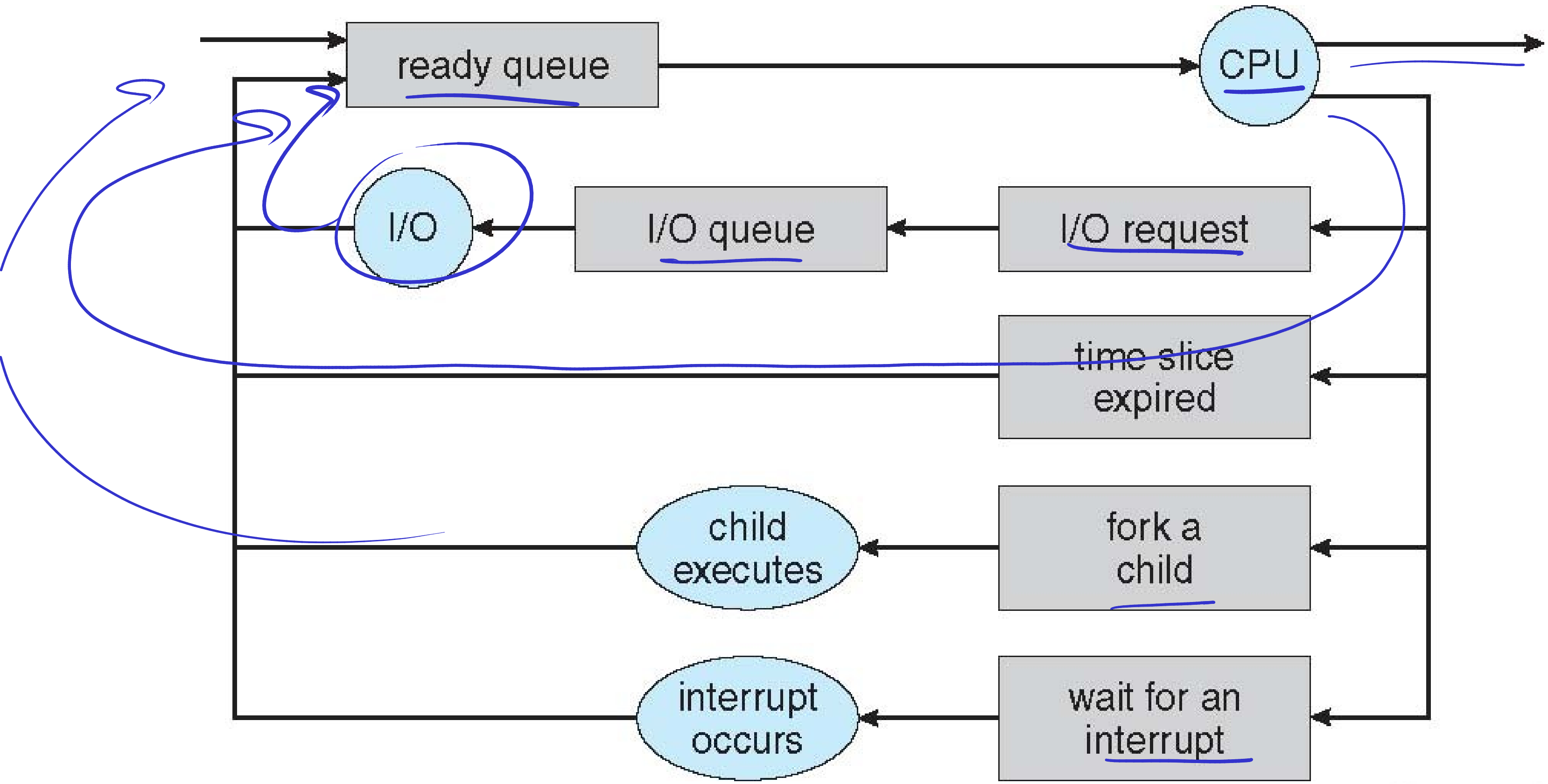
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Scheduling Queues & I/O Device Queues



Representation of Process Scheduling

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



Types of Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates a 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

A : 1000x 1000

B : 1000 x 1000

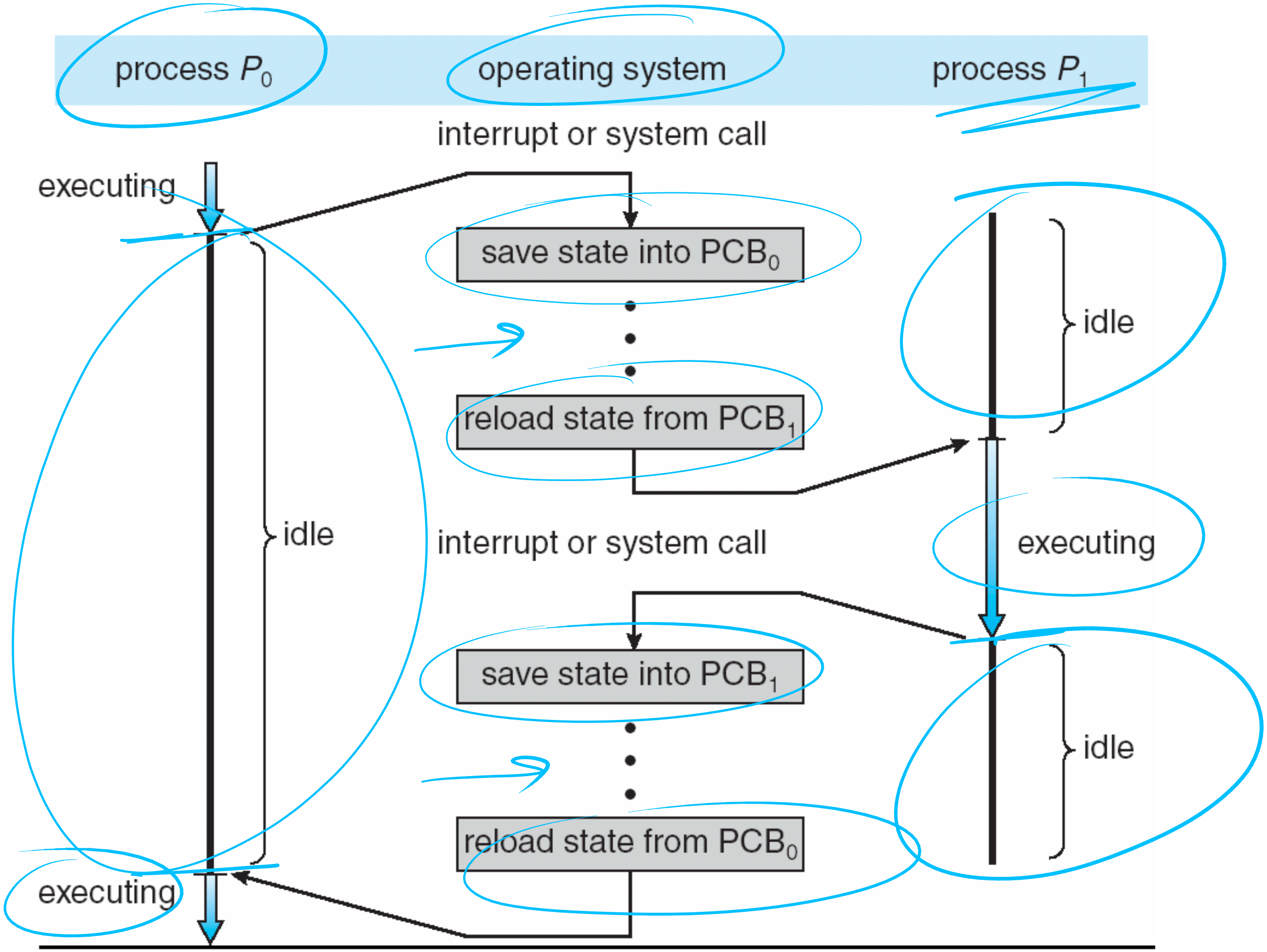
Types of Processes

- 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 a good **process mix**

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 pure 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
  - Some hardware provides multiple sets of registers per CPU => multiple contexts loaded at once

CPU Switch from Process to Process



Dispatch Events

- When does a context switch occur?
- **Clock Interrupt**, occurs after a specified time interval, usually 3-10ms
  - Execution of processes interrupted, control goes back to OS
  - The current process is added to the ready queue
  - Frequency of such an interrupt important system parameter
    - Balance overhead of context switch vs. responsiveness

Dispatch Events (Contd.)

◆ I/O Interrupt

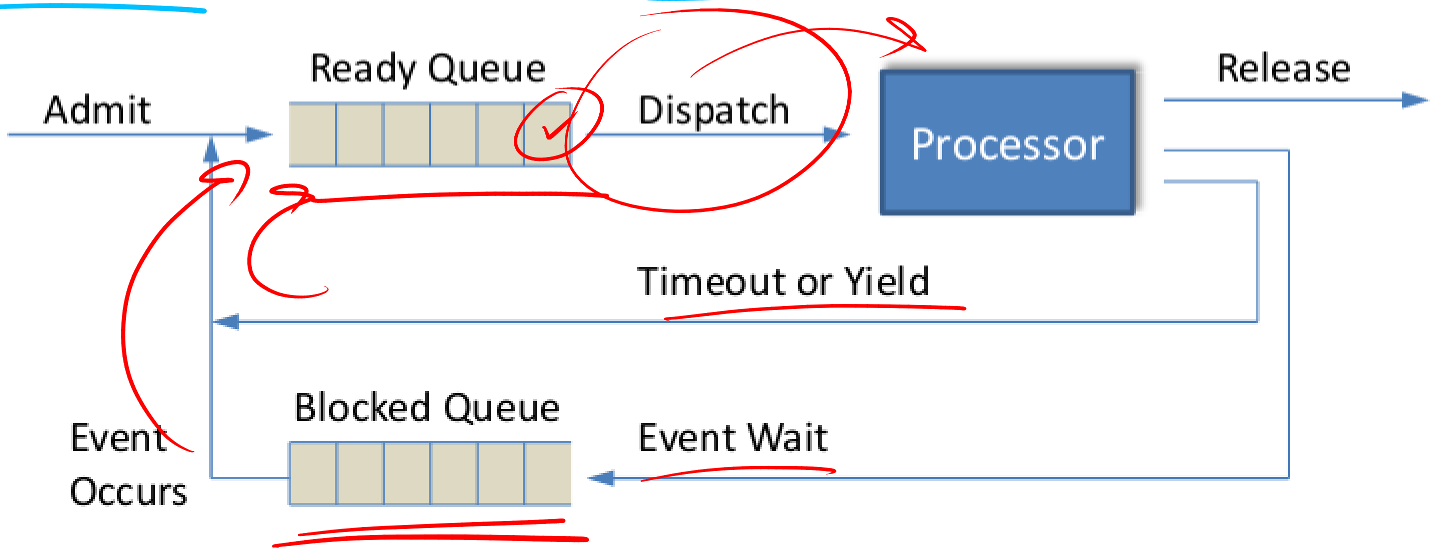
- When I/O action has occurred and data is loaded into memory
- Currently executing process is interrupted
- All blocked processes waiting for this I/O action to be completed are moved into the Ready queue
- Dispatcher must decide whether to continue execution of the process currently in Running state (has been interrupted)

Dispatch Events (Contd.)

- **Memory fault / Page fault**
  - Executing process refers to a virtual memory address that is not allocated a physical memory location (data still on disk)
  - Currently executing process (Running state) is interrupted
  - I/O request for bringing in data from disk is issued
  - Currently executing process is switched to blocked state
  - Switch to another process from the Ready queue
  - When I/O completed, blocked process moves back to Ready queue

Dispatcher

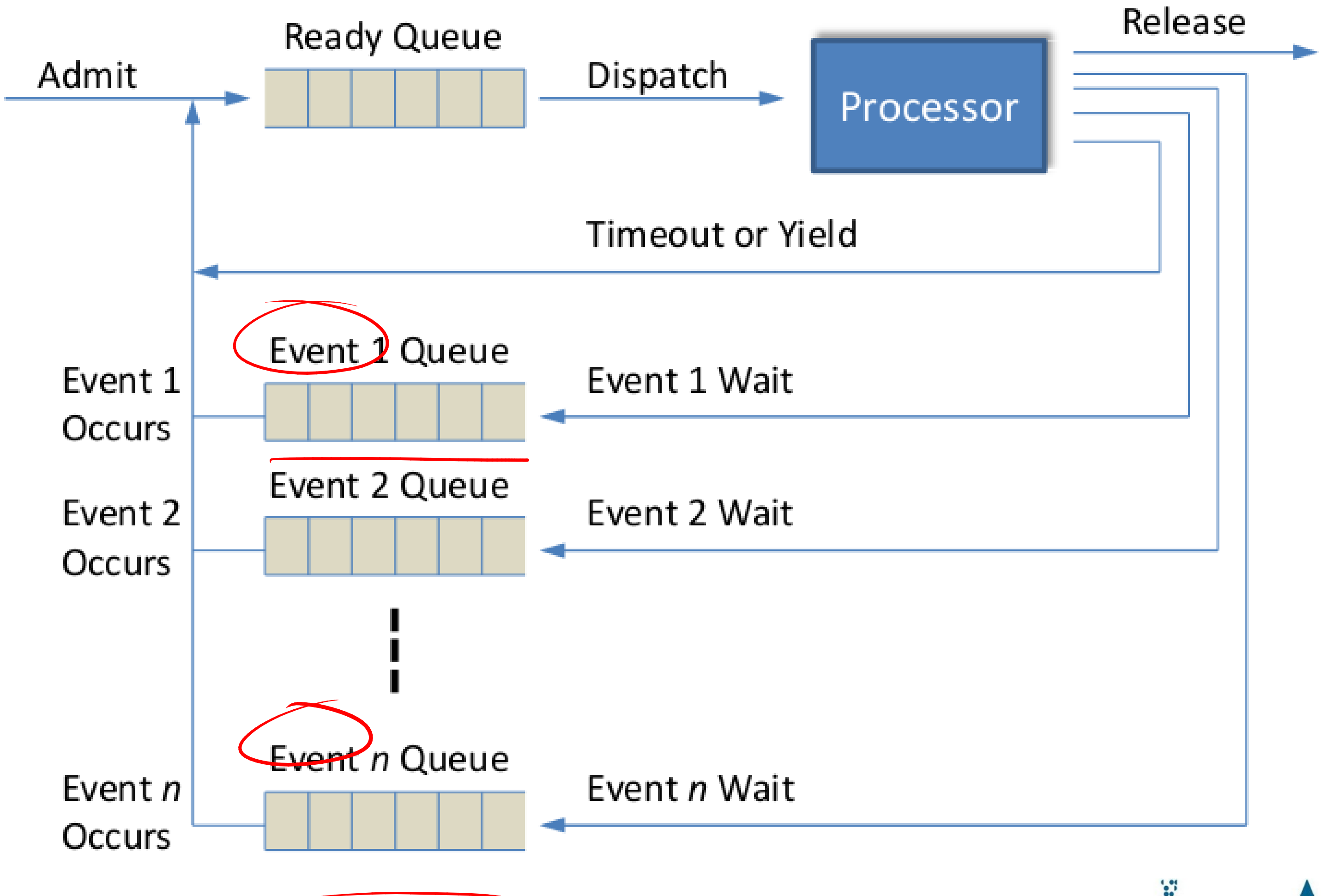
- A Dispatcher is an OS function that allocates CPU to processes, switches CPU from one process to the next.



Weakness:

When a particular event occurs, ALL processes waiting for this event need to be transferred from the “Blocked” queue to the “Ready” queue, but this can only be done by scanning the entire “Blocked” queue, where we may have hundreds or even thousands of processes waiting on different events!

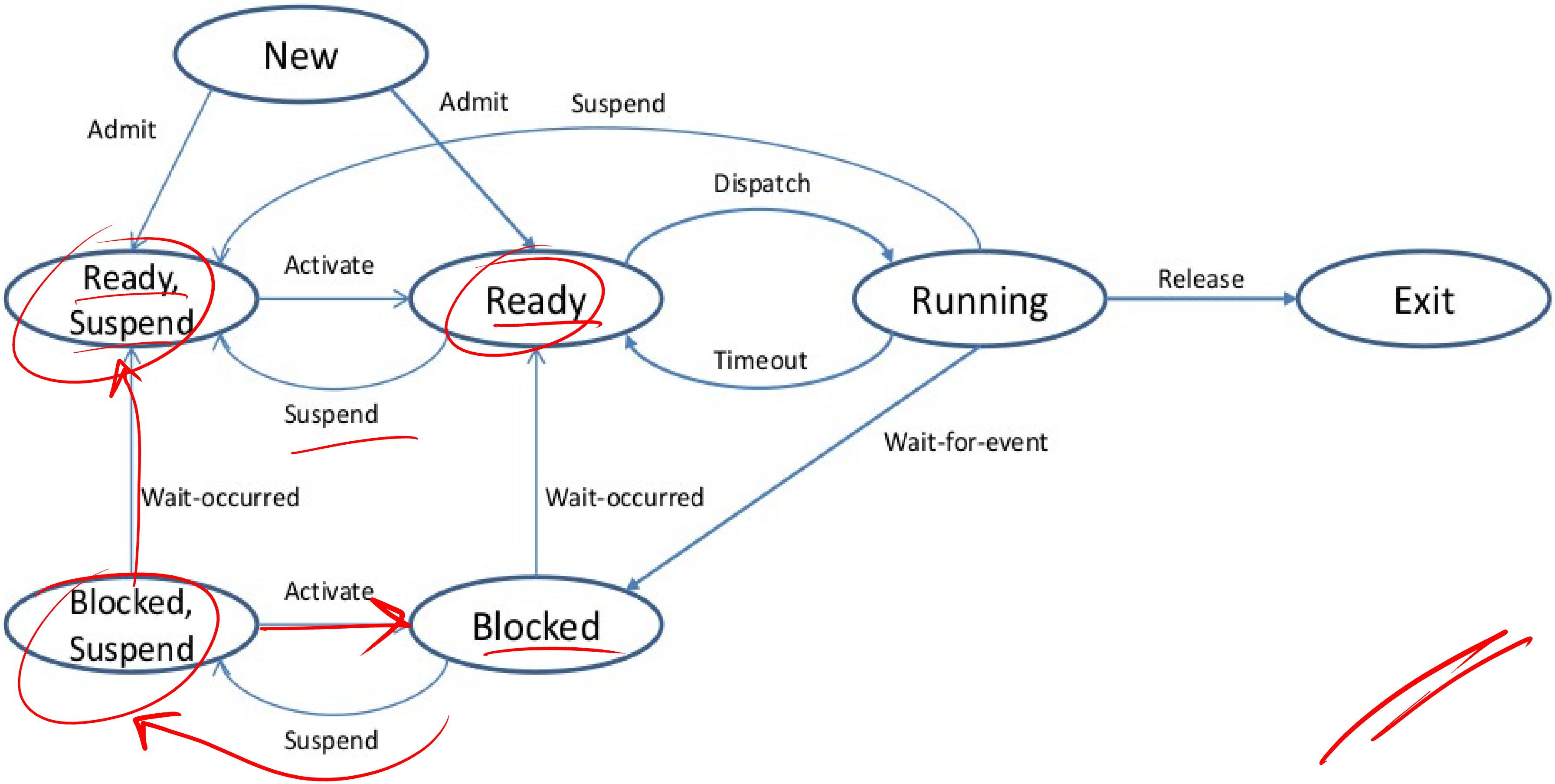
Dispatcher with Multiple Event Queues



Swapping

- **Virtualization:** Computer system appears to allow an "unlimited" number of processes to execute concurrently
- Not all of them may fit into physical memory
  - It may be useful to remove a process from memory and reduce the degree of multiprogramming => **medium-term scheduling**
- **Swapping:** Move a process to secondary memory
- We need two extra process states
  - "ready-suspended"
  - "blocked-suspended"

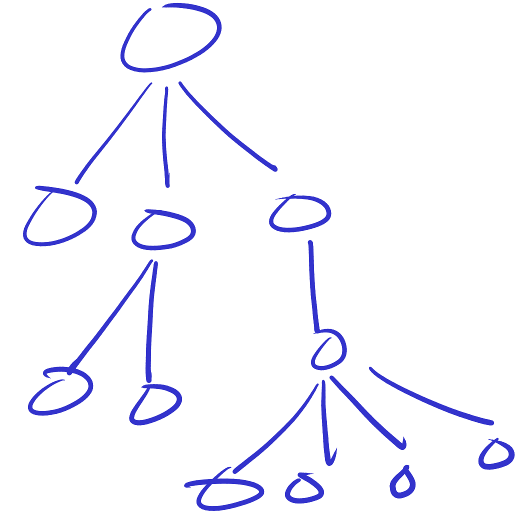
Process States with Suspend



Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation



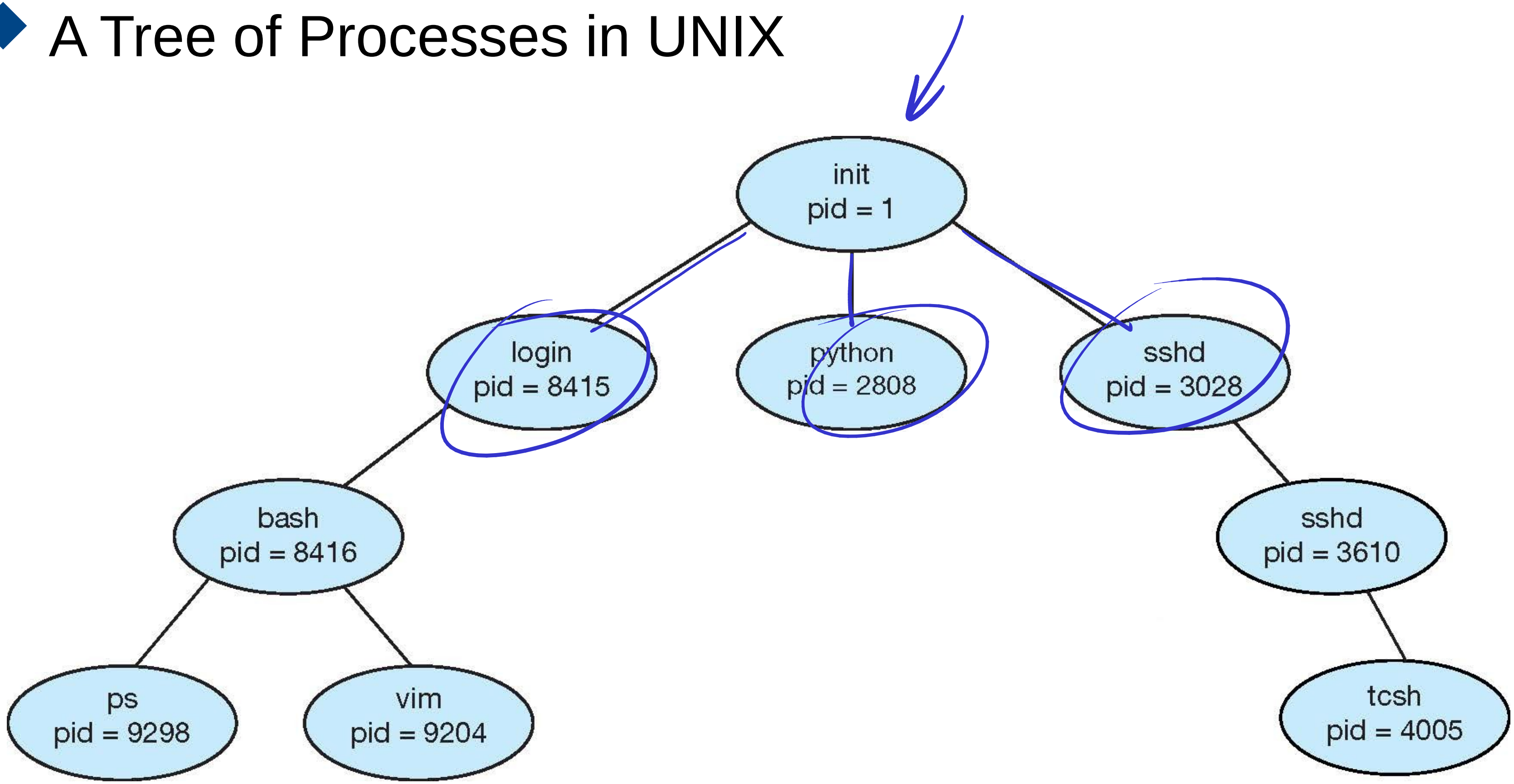
- A **process** may create other processes.
- When a process is created?
  - System boot
  - An existing process spawns a child process
  - User request to create a process
  - A batch system takes on the next job in line
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, a process is identified and managed via a **process identifier (pid)**

Process Creation – Steps Overview

- **Assign unique** process identifier to the new process.
- **Allocate space** for the process
  - Allocate memory for process image (program, data, stack)
- **Initialize** the Process Control Block
- **Add** process to the Ready queue

Process Creation

- A Tree of Processes in UNIX



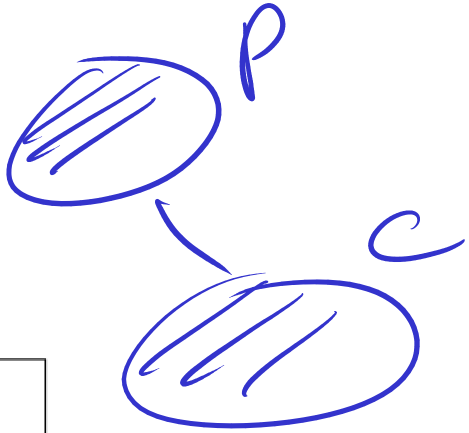
Process Creation (Cont.)

- **Resource sharing** among parents and children 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
- **Address space**
  - A child is a **duplicate** of the parent address space.
  - A child loads a **new program** into the address space.

Process Creation under UNIX

- A process creates new processes with the kernel system calls fork() and exec()
  - A new slot in the process table is allocated
  - A unique process ID is assigned to the child process
  - The process image of the parent process is copied, except the shared memory areas
  - Child process now also owns the same open files as parent
  - Child process is added to the Ready queue and will start execution

Process Creation under UNIX



- System call fork()

```
int main( ... )
{
    int pid = fork() ;
    if ( pid == 0 )
    {
        // child process
        // program code for child
    }
    else if ( pid > 0 )
    {
        // parent process
        // program code for parent
    }

    return 0 ;
}
```

Process Creation under UNIX

Parent starts

```
int main()
{
    // printf(“Parent\n”);
    . . .
    . . .
    int ret = fork() ;
    . . .
    . . .
    . . .
    . . .
    . . .
}
```

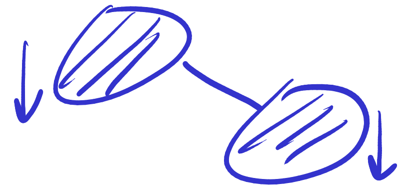
Process Creation under UNIX

Kernel Mode

Parent starts

```
int main()
{
    // printf(“Parent\n”);
    . . .
    . . .
    int ret = fork() ;
    . . .
    . . .
    . . .
    . . .
    . . .
}
```

Process Creation under UNIX



Kernel Mode

Parent starts

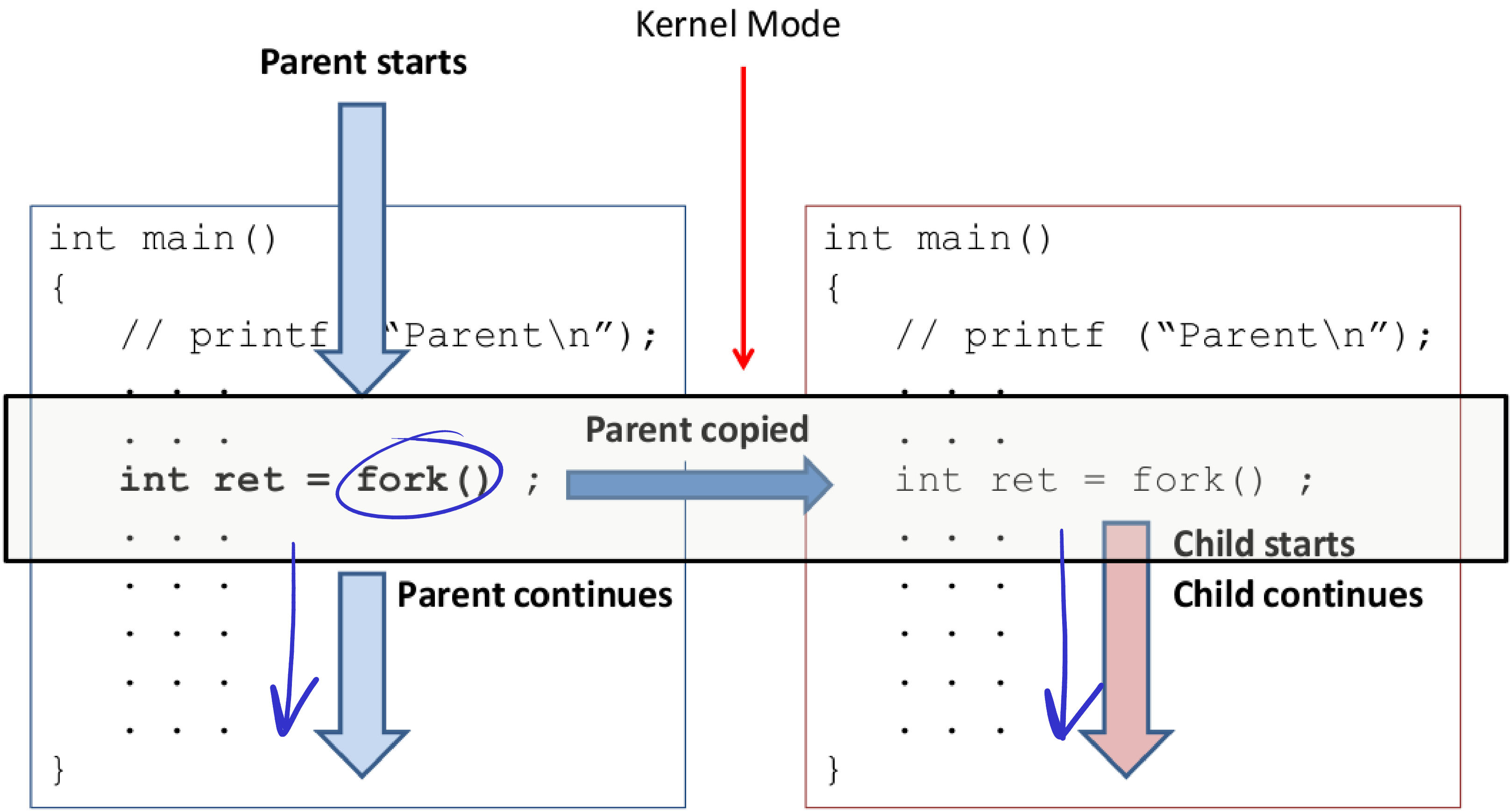
child

```
int main()
{
   // printf ("Parent\n");
   . . .
   . . .
   int ret = fork() ;
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

Parent copied

```
int main()
{
   // printf ("Parent\n");
   . . .
   . . .
   int ret = fork() ;
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

Process Creation under UNIX



Process Creation under UNIX

A paradoxical situation occurs:

- When fork() is called by parent process, there is only one process
- When fork() is finished and returns a return value, there are two processes
  - Child process is clone of parent process, with that, also the call of fork() has been cloned
  - Parent process continues execution at the return from calling fork()
  - Child process begins executing at the same point in the code as parent – at the return from calling fork()

Process Creation under UNIX

- Distinction between parent and child process
  - System call fork() has different return value in parent and child processes
- Return value of fork()
  - Parent process: it returns the process ID of the child process
  - Child process: fork() return 0

Process Creation under UNIX

- Our program has to serve the parent as well as the child process

```
int main( ... )
{
    int pid = fork() ;
    if ( pid == 0 )
    {
        // child process
        // program code for child
    }
    else if ( pid > 0 )
    {
        // parent process
        // program code for parent
    }

    return 0 ;
}
```

Process Creation under UNIX

- Run a new program in child process image
  - Use of exec() system call

```
int main()
{
   // printf ("Parent\n");
   . . .
   . . .
   int ret = fork() ;
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

Parent continues

Parent copied

```
int main()
{
   // printf ("Parent\n");
   . . .
   . . .
   if ( fork()==0 )
   {
   exec ( "myprogram");
   . . .
   . . .
   . . .
}
```

New program

Process Creation under UNIX

- System call exec() replaces content of cloned image with the new program

```
int main()
{
   // printf ("Parent\n");
   . . .
   . . .
   int ret = fork() ;
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

Parent continues

New program starts executing

```
int main()
{
   // printf ("myprogram\n");
   . . .
   . . .
   . . .
   . . .
   . . .
}
```

System Call exec()

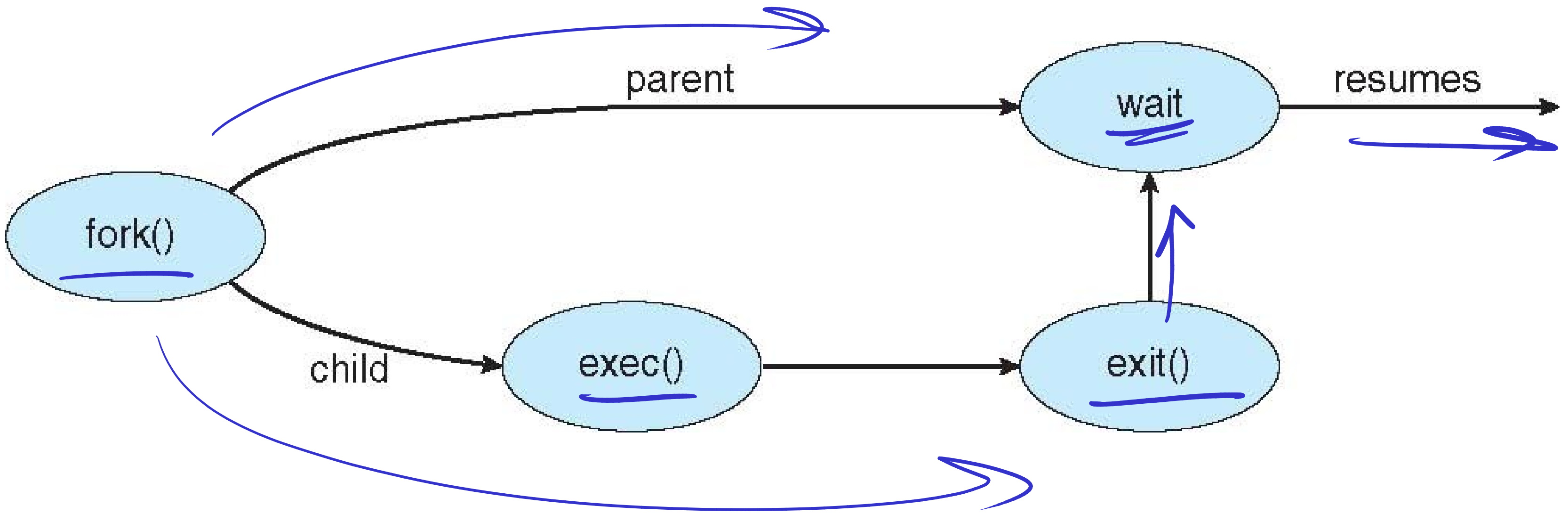
- exec() comes in different flavours
  - execl(), execle(), execlp()
  - execv(), execve(), execvp()
- Meaning of the name annotations
  - "e": an array of pointers to env variables is explicitly passed to the new process image
  - "l": commandline arguments are passed individually
  - "p": PATH environment variable is used to find file name of program to be executed
  - "v": commandline arguments are passed as an array of pointers

C program to create a separate process in UNIX

```
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 Wait

- Parent process may call system function **waitpid()** to wait for the exit of child process



Events Leading to Process Termination

- Normal termination
  - Program ends itself
- Abnormal termination
  - OS intervenes, user sends kill signal (CTRLC)
  - Access to memory locations that are forbidden
  - Time out
  - I/O errors
  - Not enough memory, stack overflow
  - Parent process terminated
  - etc.

Events Leading to Process Termination

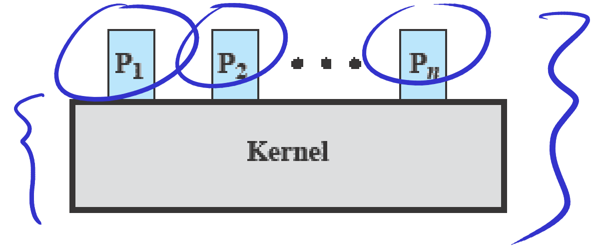
- **Normal exit (voluntary) / Error exit (voluntary)**
  - Regular completion of a process, with or without error code
  - Process voluntarily executes the exit(errNo) system call to indicate to the operating system that it has finished.
- **Fatal error (involuntary)**
  - Uncatchable or uncaught error
  - Service errors, such as: no memory left for allocation, I/O error, etc
  - Total time limit exceeded
  - Arithmetic error, out-of-bounds memory access, etc.
- **Killed by another process via the kernel (involuntary)**
  - The process receives a SIGKILL signal
  - In some systems, the parent takes down all its children with it

Process Termination (Contd.)

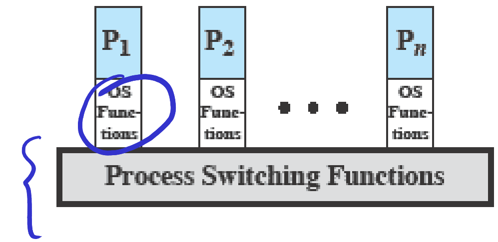
- Some operating systems do not allow a child process to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination**. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- What is a zombie process?

Execution of the Operating System

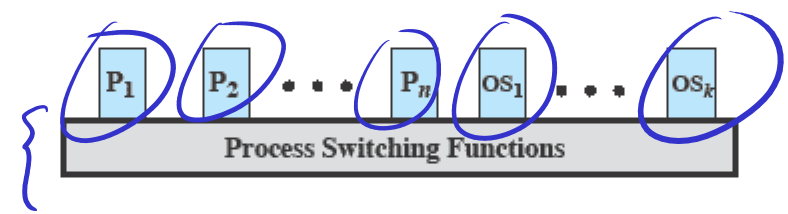
- Is the Operating System itself a process?
- Is it executing
  a) Separately, only when processes are interrupted?
  b) As part of the user process images?
  c) As a set of processes (micro kernel)?



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

Execution of the Operating System

- Is the Operating System itself running as a process? Or as a collection of processes?
- Various design options

a) **Non-process Kernel:**

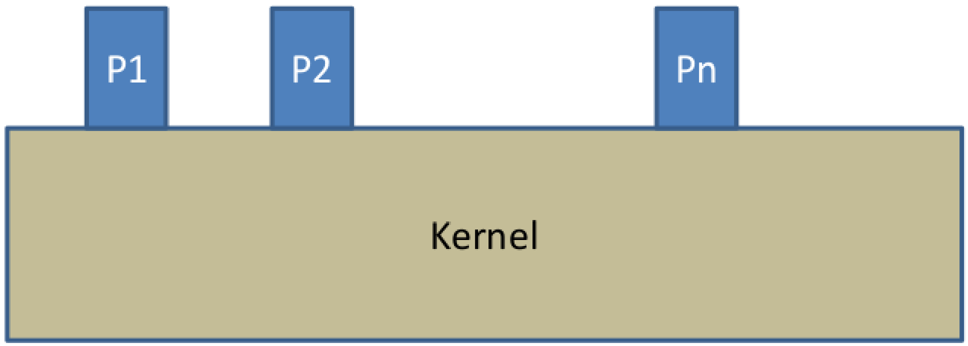
  - Traditional approach, many older operating systems.
  - Kernel only executes when user processes interrupted.

b) Execution of all OS functions **in the context of a user process**, only mode switch is required.

c) **Process based operating systems**, both mode switch and context switch are required.

Non-process Kernel

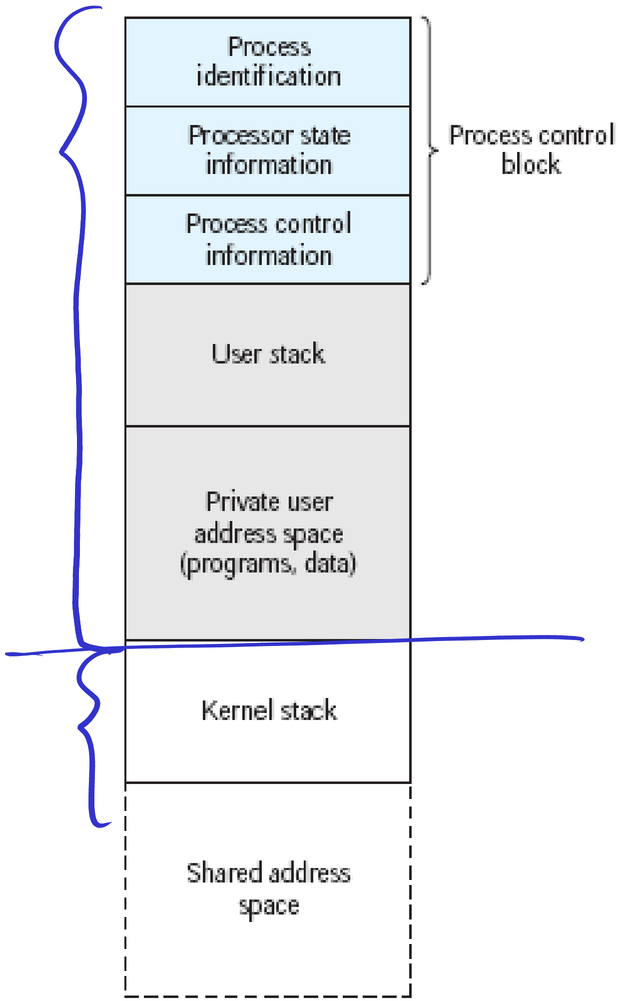
- Kernel acts as a monitor for user processes, only user processes regarded as processes.
  - Kernel operations separate from user operations, kernel is a supervisor
  - No concurrent execution of user processes and kernel
  - Processes are interrupted and control is switched back to kernel
  - Kernel executes only during these interruptions



- OS code is placed in a reserved memory region and executes in privileged mode. It has its own system stack.

Execution within a User Process

- User address space includes kernel functions
  - User address space "covers" kernel, can call system functions from within process
  - Mode switch necessary to execute these functions
  - No context switch, as we are still in the same process
- Dispatcher (process switching) executes outside of user programs.



Process-based Operating Systems

- Kernel functions run as separate processes, run in kernel mode. Dispatcher (process switching) is a small separate part
- Concurrency within kernel, kernel processes scheduled together with user processes
- Useful in multiprocessor environments, as kernel functionality may be distributed to other CPU cores

Summary

- The concept of a process, its structure and representation in memory.
- Different process states and the notion of process scheduling and different types of queues.
- Operations on processes including creation, switching and termination.
- Execution of the Operating System as a non-process kernel, within user process or as a set of separate processes.

References / Links

- Chapter # 3: **Processes,** Operating System Concepts (9th edition) by Silberschatz, Galvin & Gagne
- Chapter #3: **Process Description and Control,** Operating Systems: Internals and Design Principles (7th edition) by William Stallings