Process Management
Chapter 3

Processes

- A process is a program in a state of execution (created but not terminated)
  - Program is a passive entity – one on your disk (survivor.class, kelly.out, …)
  - Process is an active entity – in memory. A unit to which OS allocates resources
- OS interleaves the execution of several processes.
- OS allocates resources to processes
  - For example, memory, CPU time, # of socket connections etc
- OS allows users to create processes and processes to spawn other processes
- OS supports communication among processes
  - Also known as IPC (inter-process communication)

When does a process gets created?

- Submission of a batch job
- User logs on
- Created by OS to provide a service to a user (ex: printing a file)
- Spawned by an existing process
  - a user program can dictate the creation of a number of processes

When does a process gets terminated?

- Batch job issues *Halt* instruction
- User logs off
- Process executes a service request to terminate
- Error and fault conditions

Reasons for Process Termination

- Normal completion
- Time limit exceeded
- Memory unavailable
- Memory bounds violation
- Protection error
  - example: write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event

Reasons for Process Termination

- I/O failure
- Invalid instruction
  - happens when try to execute data
- Privileged instruction
- Operating system intervention
  - such as when deadlock occurs
- Parent request to terminate one offspring
- Parent terminates so child processes terminate

Process States

- Let us start with these states:
  - The **Running** state
    - The process that gets executed (single CPU)
  - The **Ready** state
    - any process that is ready to be executed
  - The **Blocked** state
    - when a process cannot execute until some event occurs (ex: the completion of an I/O)

Other Useful States

- The New state
  - OS has performed the necessary actions to create the process
    - has created a process identifier
    - has created tables needed to manage the process
  - but has not yet committed to execute the process (not yet admitted)
    - because resources are limited

Other Useful States

- The Exit state
  - Termination moves the process to this state
  - It is no longer eligible for execution
  - Tables and other info are temporarily preserved for auxiliary program
    - Ex: accounting program that cumulates resource usage for billing the users
- The process (and its tables) gets deleted when the data is no more needed

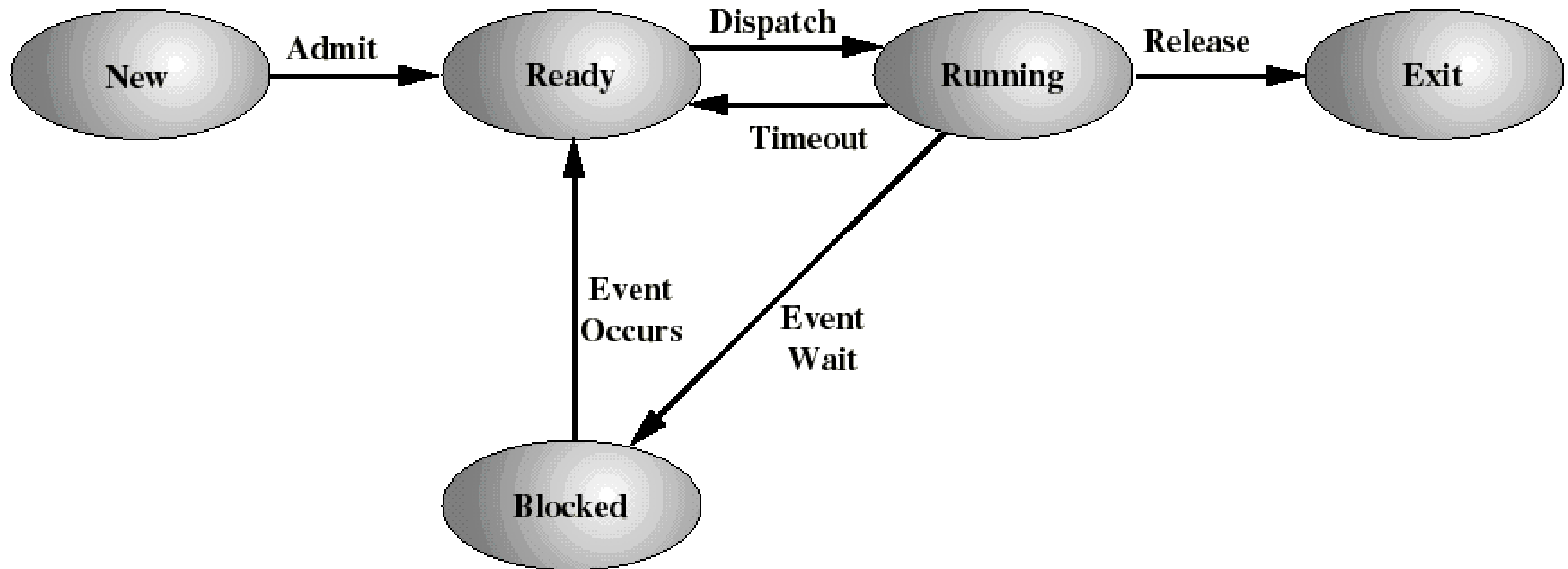
Process Transitions

- Ready --> Running
  - When it is time, the dispatcher selects a new process to run
- Running --> Ready
  - the running process has expired his time slot
  - the running process gets interrupted because a higher priority process is in the ready state

Process Transitions

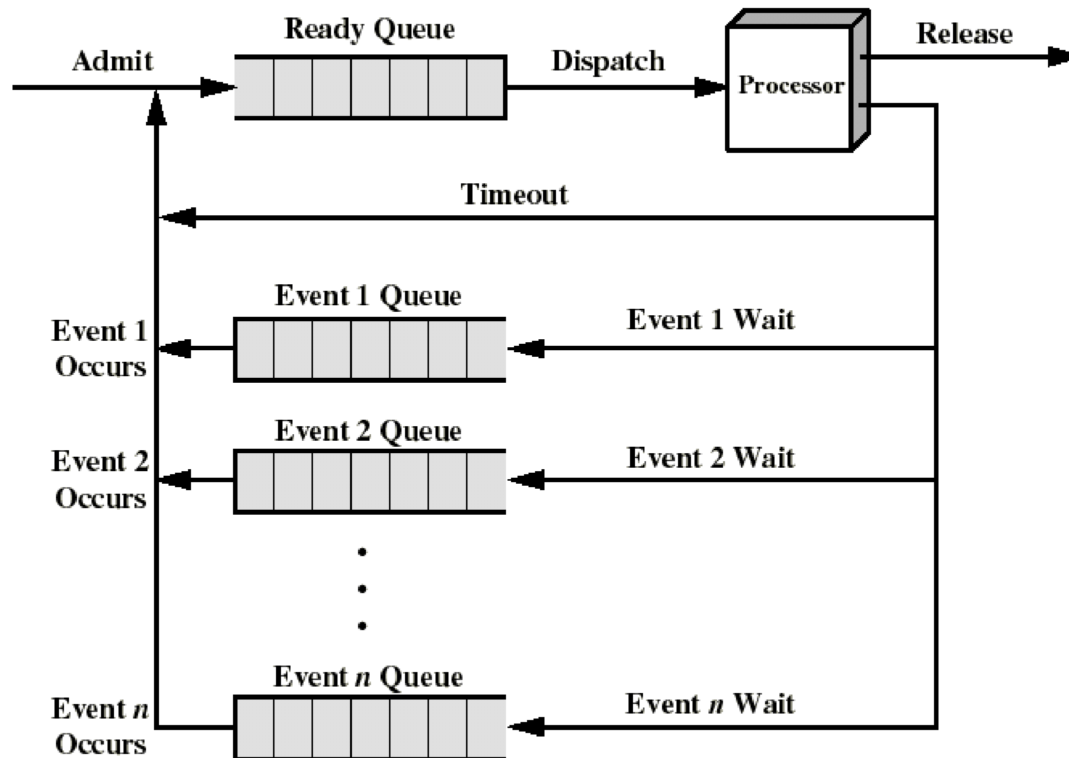
- Running --> Blocked
  - When a process requests something for which it must wait
    - – a service that the OS is not ready to perform
    - – an access to a resource not yet available
    - – initiates I/O and must wait for the result
    - – waiting for a process to provide input (IPC)
- Blocked --> Ready
  - When the event for which it was waiting occurs

A Five-state Process Model



Ready to exit: A parent may terminate a child process

A Queuing Discipline



- **Ready queue without priorities (ex: FIFO)**
- **When event n occurs, the corresponding queue is moved into the ready queue**

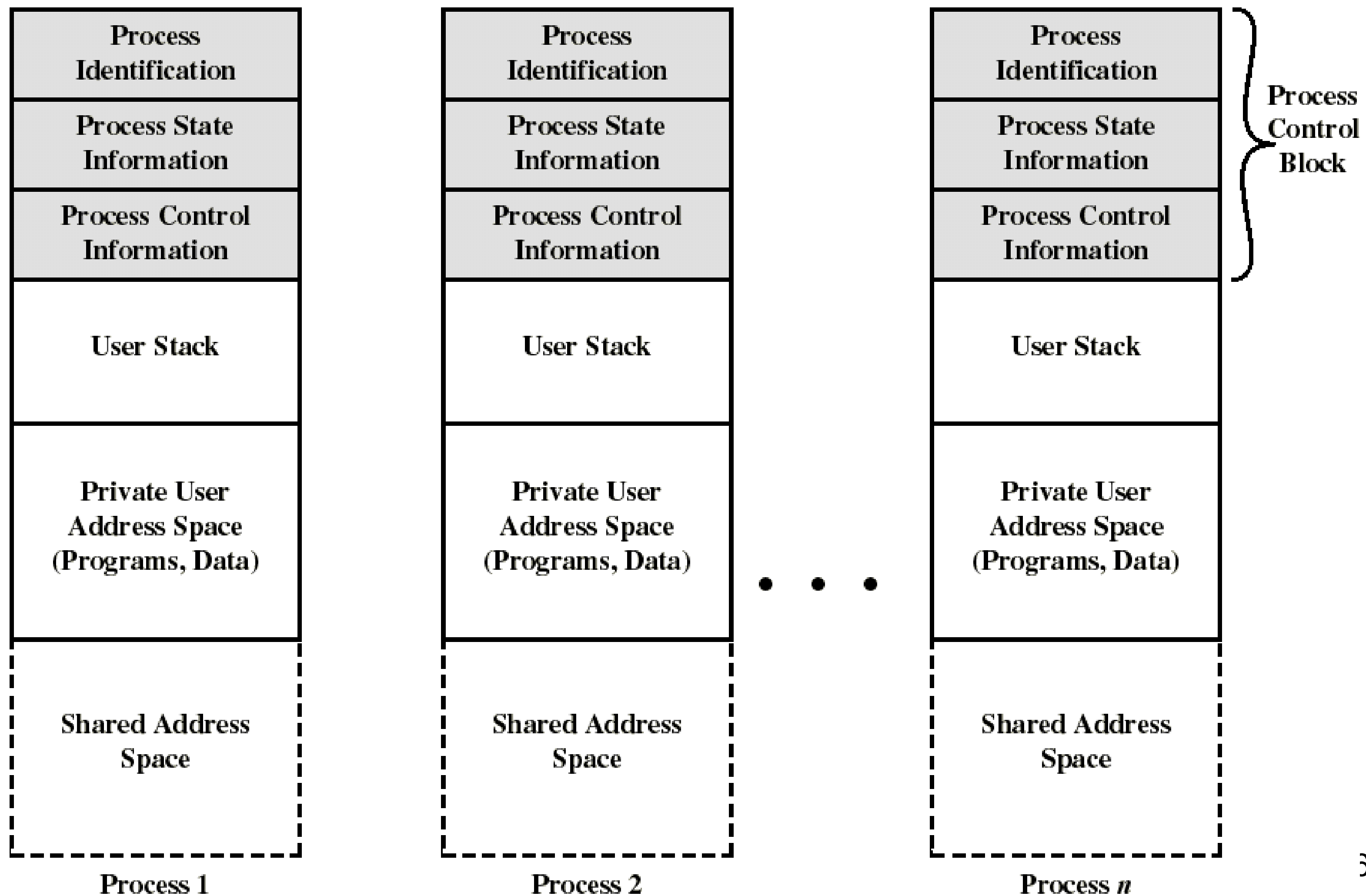
Operating System Control Structures

- An OS maintains the following tables for managing processes and resources:
  - Memory tables (see later)
  - I/O tables (see later)
  - File tables (see later)
  - Process tables (this chapter)

Process Image (process constituents)

- User program
- User data
- Stack(s)
  - for procedure calls and parameter passing
- Process Control Block (execution context)
  - Data needed (process attributes) by the OS to control the process. This includes:
    - Process identification information
    - Processor state information
    - Process control information

Process images in memory



Location of the Process Image

- Each process image is in memory
  - may not occupy a contiguous range of addresses (depends on the memory management scheme used)
  - both a private and shared memory address space is used
- The location if each process image is pointed to by an entry in the Process Table
- For the OS to manage the process, at least part of its image must be brought into main memory

Process Identification (in the PCB)

- A few numeric identifiers may be used
  - Unique process identifier (always)
    - indexes (directly or indirectly) into the primary process table
  - User identifier
    - the user who is responsible for the job
  - Identifier of the process that created this process

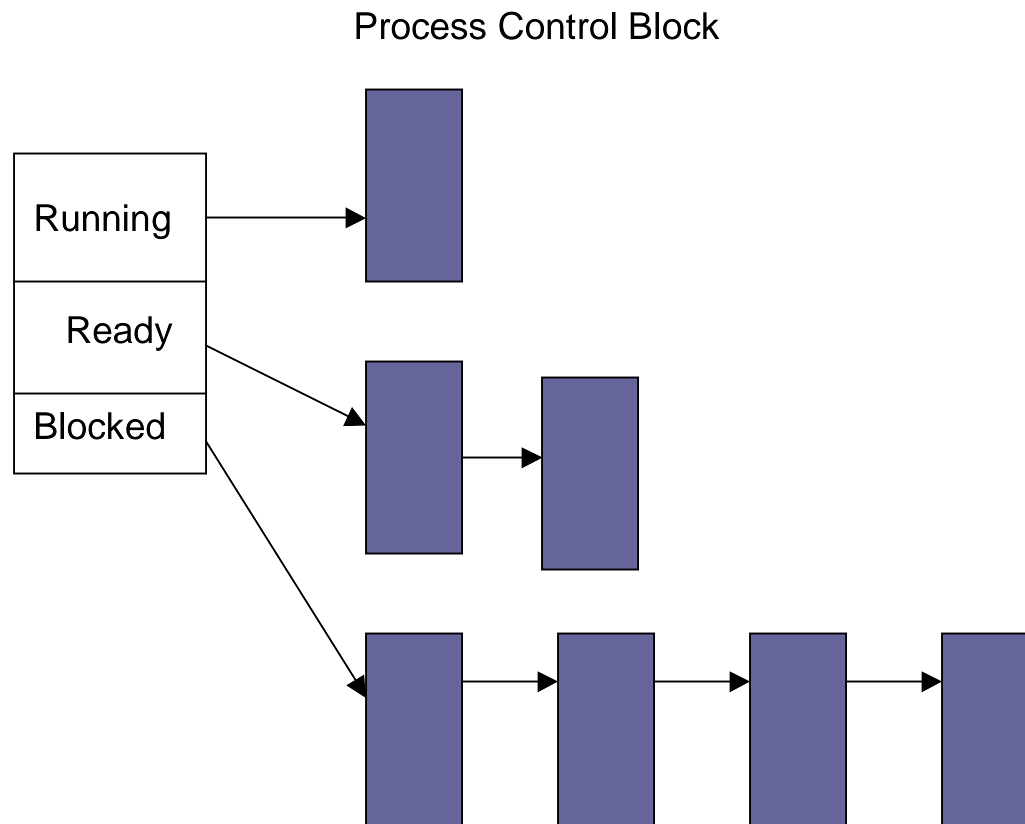
Processor State Information (in PCB)

- Contents of processor registers
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program status word (PSW)
  - contains status information
  - Example: the EFLAGS register on Pentium machines

Process Control Information (in PCB)

- scheduling and state information
  - Process state (ie: running, ready, blocked...)
  - Priority of the process
  - Event for which the process is waiting (if blocked)
- data structuring information
  - may hold pointers to other PCBs for process queues, parent-child relationships and other structures

Queues as linked lists of PCBs



Process Control Information (in PCB)

- Inter-process communication
  - may hold flags and signals for IPC
- process privileges
  - Ex: access to certain memory locations...
- memory management
  - pointers to segment/page tables assigned to this process
- resource ownership and utilization
  - resource in use: open files, I/O devices...
  - history of usage (of CPU time, I/O...)

Context Switching

- Kernel switches among processes
- When a process finishes its time slice or blocks or terminates
- Kernel schedules the next ready process
- This activity is called context switching
- Each process has state (or context) that needs to be saved before switching to another process
- -- Why

Steps in Process (Context) Switching

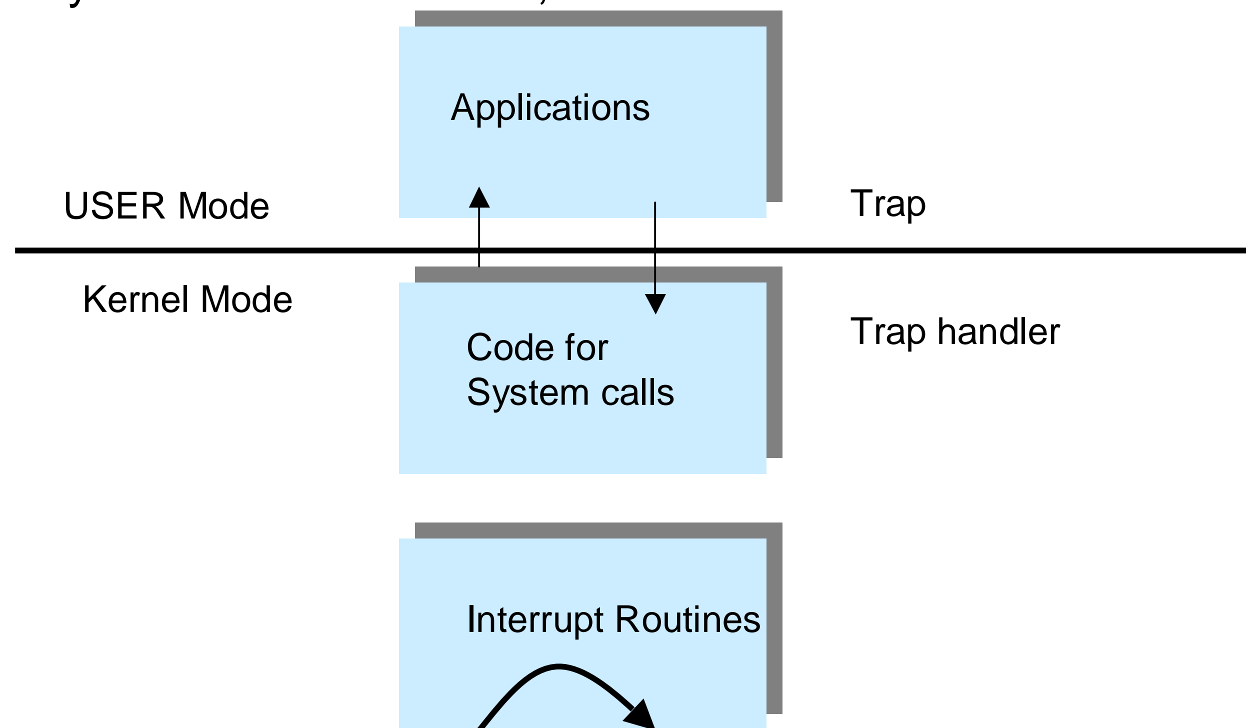
- Save context of processor including program counter and other registers
- Update the PCB of the running process with its new state and other associate info
- Move PCB to appropriate queue - ready, blocked
- Select another process for execution
- Update PCB of the selected process
- Restore CPU context from that of the selected process

Executing OS code

- Application programs cannot directly access I/O devices, other resources etc
- Still, applications must request OS to access them
- How does the user/application execute instructions
  - Application uses system calls
  - System calls are traps (special instructions)
  - After a trap, CPU executes in a special mode called kernel mode
  - OS Code can only be executed when CPU is in kernel mode

Crossing user mode to kernel mode

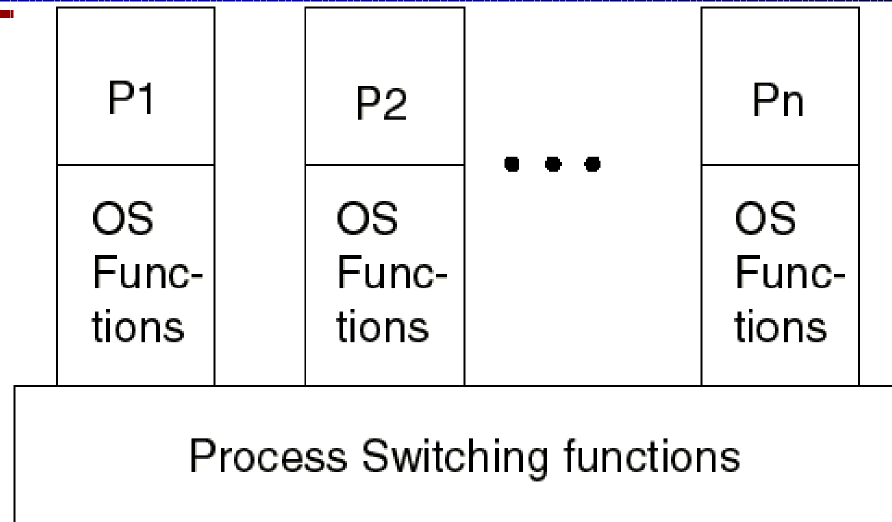
- User-mode program calls kernel procedure using system call
- System call causes CPU to switch to kernel mode
- Once the system call is executed, the CPU reverts back to user mode



Modes of Execution

- To provide protection to PCBs (and other OS data) most processors support at least 2 execution modes:
  - Privileged mode (a.k.a. system mode, kernel mode, supervisor mode, control mode)
    - manipulating control registers, primitive I/O instructions, memory management...
  - User mode
- For this the CPU provides a (or a few) mode bit which may only be set by an interrupt or trap or OS call

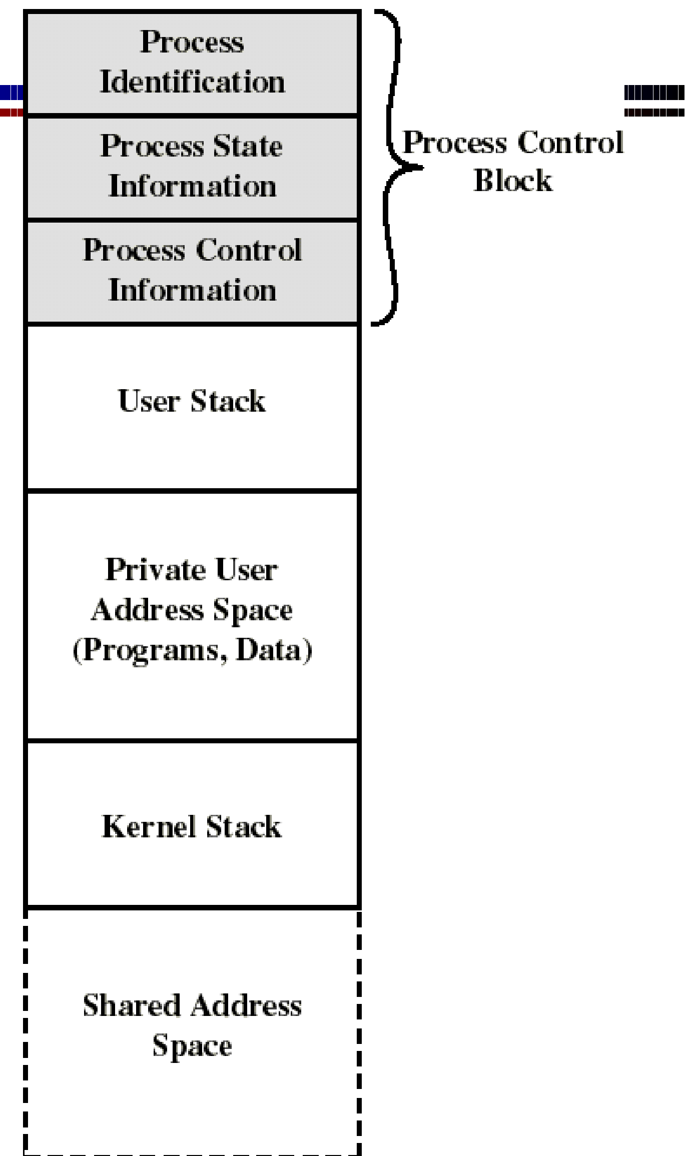
Execution within User Processes



- Virtually all OS code gets executed within the context of a user process
- On Interrupts, Traps, System calls: the CPU switch to kernel mode to execute OS routine within the context of user process (mode switch)
- Control passes to process switching **functions (outside processes) only when needed**

Execution within User Processes

- OS code and data are in the shared address space and are shared by all user processes
- Separate kernel stack for calls/returns when the process is in kernel mode
- Within a user process, both user and OS programs may execute (more than once)



UNIX Process Creation

- Every process, except process 0, is created by the fork() system call
  - fork() allocates entry in process table and assigns a unique PID to the child process
  - child gets a copy of process image of parent: both child and parent are executing the same code following fork()
  - but fork() returns the PID of the child to the parent process and returns 0 to the child process

UNIX System Processes

- Process 0 is created at boot time and becomes the "swapper" after forking process 1 (the INIT process)
- When a user logs in: process 1 creates a process for that user

Examples of fork

- Example1
  - Child process Prints a statement
- Example 2
  - Parent process prints a statement
- Example 3
  - Parent and child print statements
- Example 4
  - Parent and child prints with wait system call

Wait /exit system call

- *int wait(int *statusp)*
  - Wait() system call delays its caller until one of its child terminates
  - Return –1 if no child process exists
- exit system call
  - *exit(int status)*
  - This system call terminates the calling process

Semantics of exec system call

- *exec* system call replaces the current image
- Process ID remains the same
- *execlp(file,arg0,arg1,…argv)*

UNIX Process Image

- User-level context
  - Process Text (ie: code: read-only)
  - Process Data
  - User Stack (calls/returns in user mode)
- Register context

UNIX Process Image

- System-level context
  - Process table entry
    - – the actual entry concerning this process in the Process Table maintained by OS
      - Process state, UID, PID, priority, event awaiting, signals sent, pointers to memory holding text, data...
  - U (user) area
    - – additional process info needed by the kernel when executing in the context of this process
      - effective UID, timers, limit fields, files in use ...
  - Kernel stack (calls/returns in kernel mode)
  - Per Process Region Table (used by memory manager)