Mysterious life of a Process

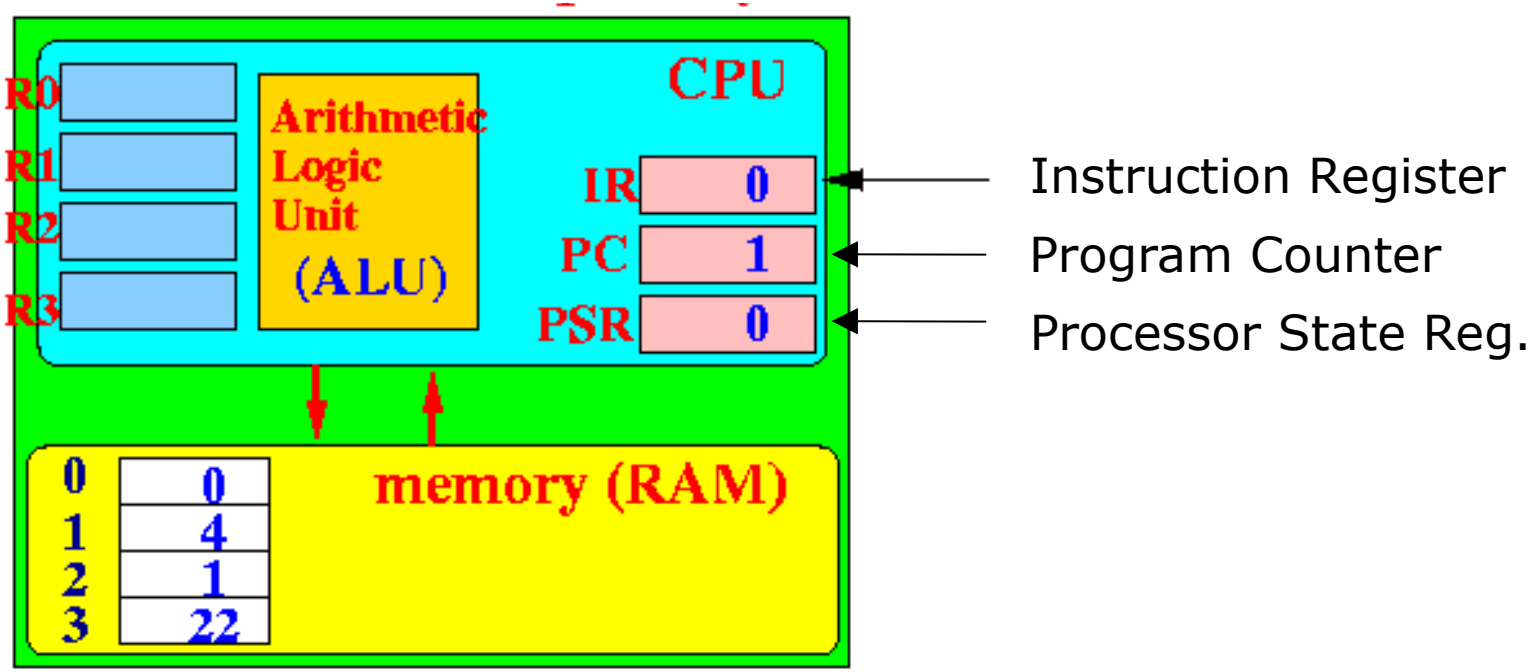
Process Concept

- Program is ***passive*** entity stored on disk (**executable file**),
- Process is ***active*** when executable file loaded into memory
- Execution of program 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 Concept

- Process – a program in execution; process execution must progress in sequential fashion
- The terms **job** and **process** is used almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion

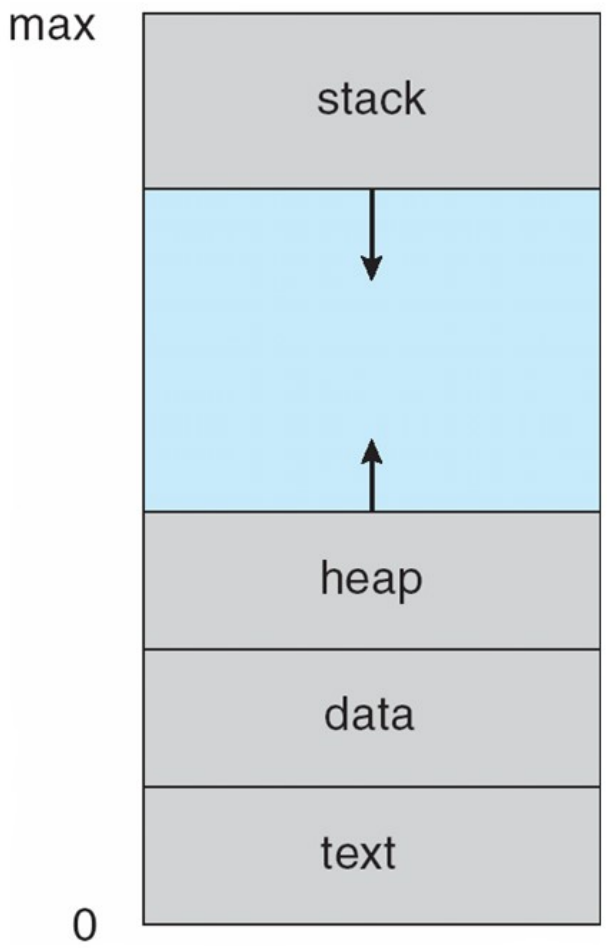
Architecture of CPU



Process Elements

- A process is comprised of:
  - Program code (possibly shared)
  - A set of data
  - A number of attributes describing the state of the process

Process in Memory

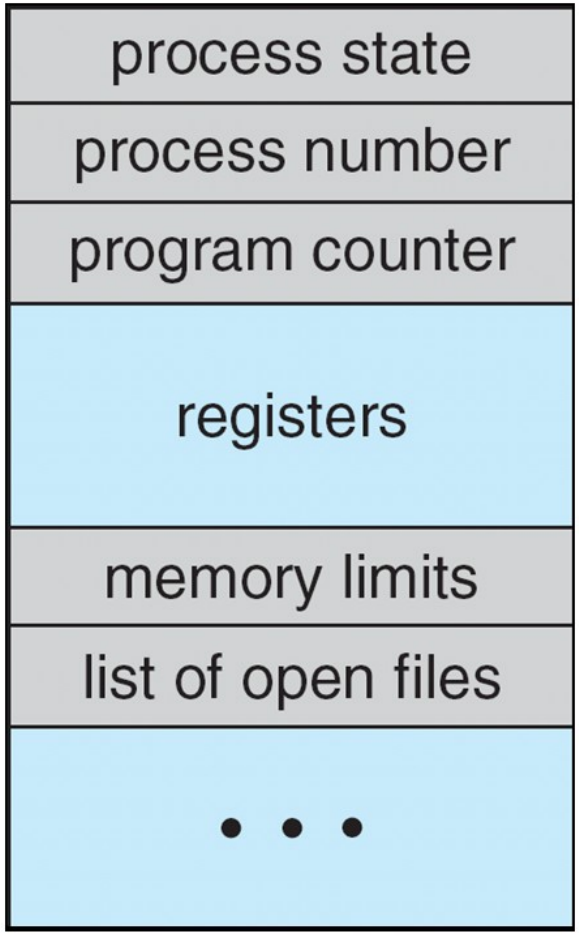


Process Control Block (PCB)

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

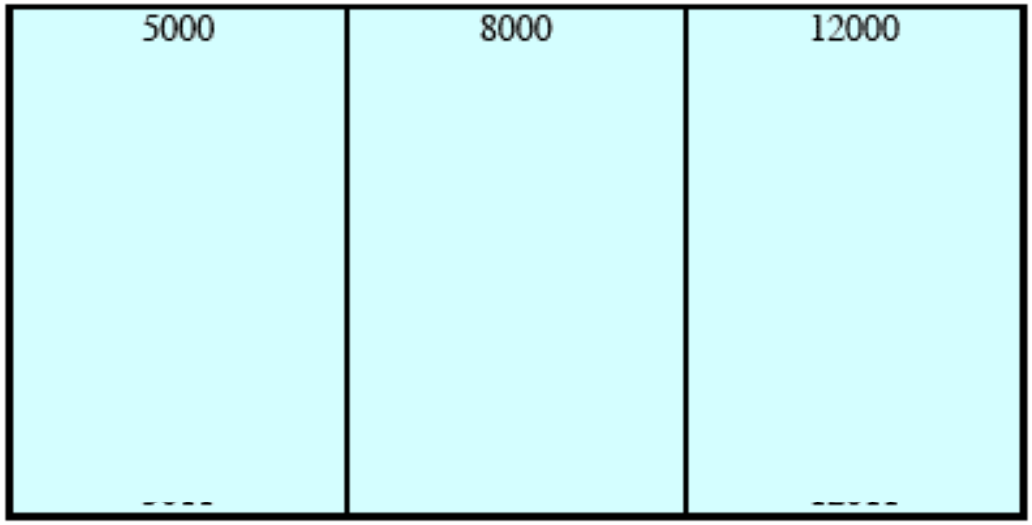


Trace of the Process

- The behavior of an individual process is shown by listing the sequence of instructions that are executed
- This list is called a ***Trace***
- ***Dispatcher*** is a small program which switches the processor from one process to another

Trace of three multi tasking processes

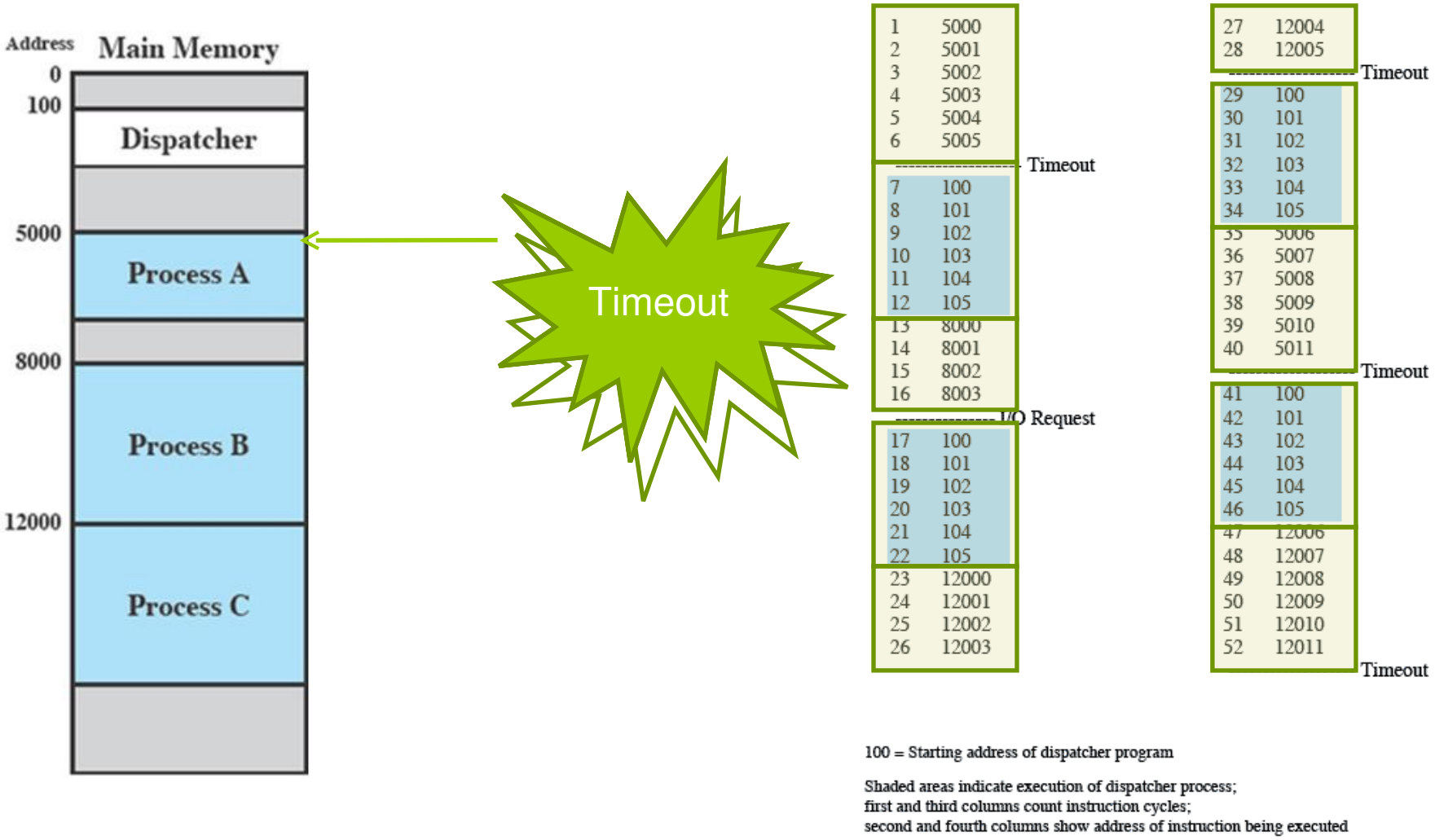
- Each process runs to completion



(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

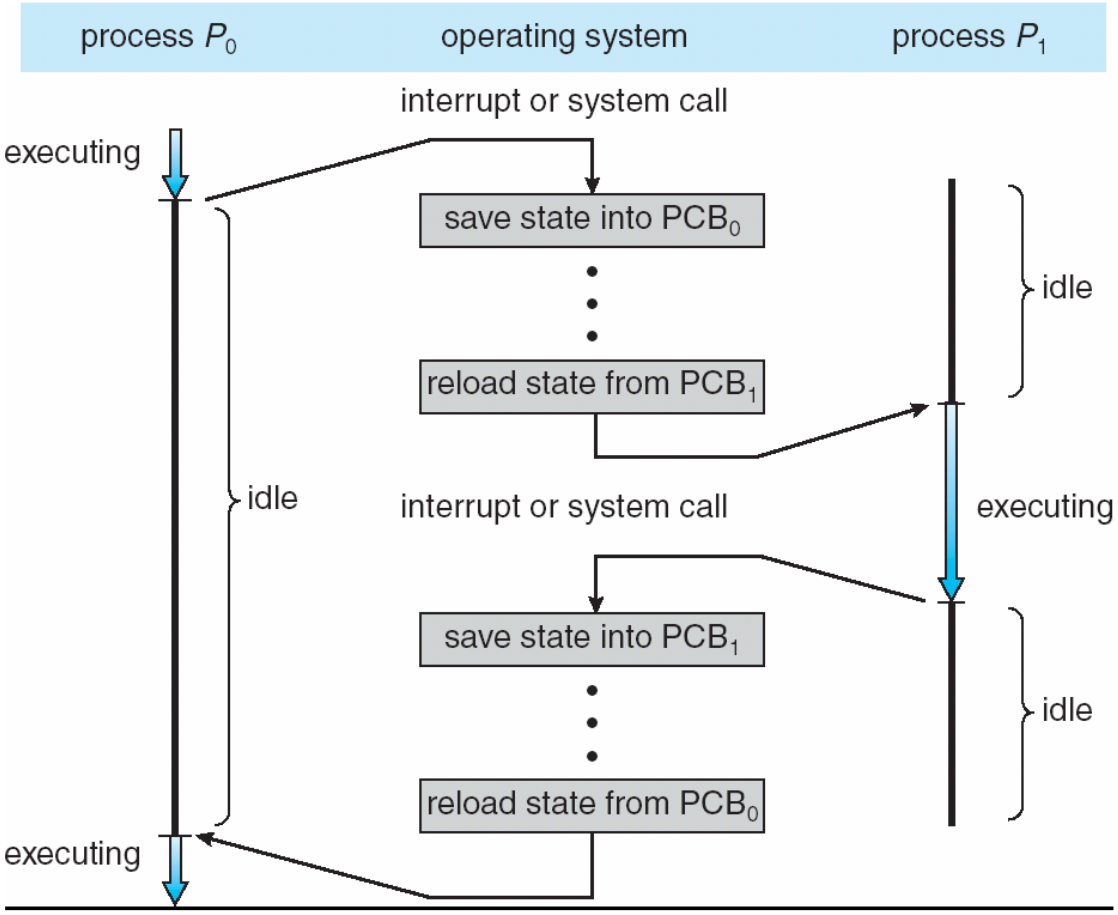
Trace from Processor's perspective



100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

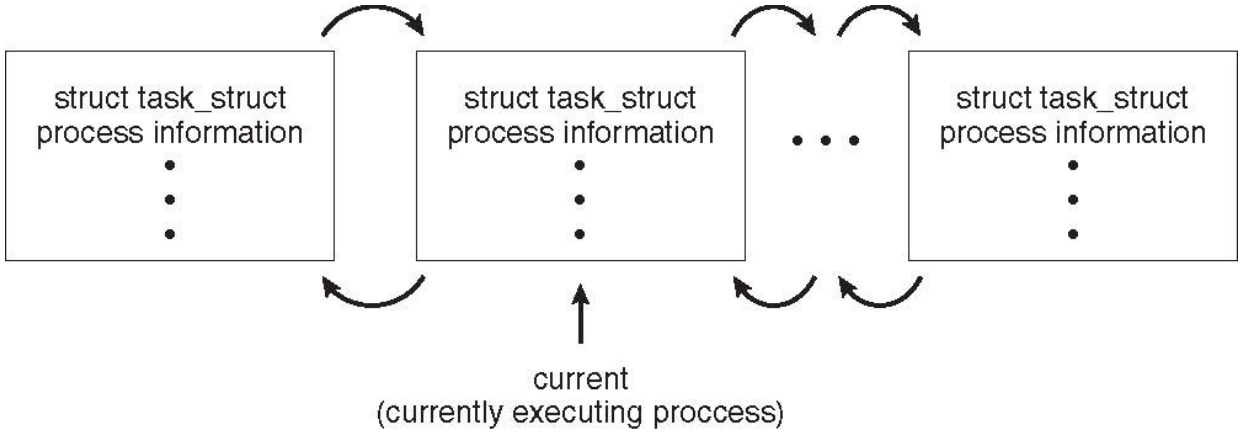
CPU Switch From Process to Process



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 */
```



Process Birth

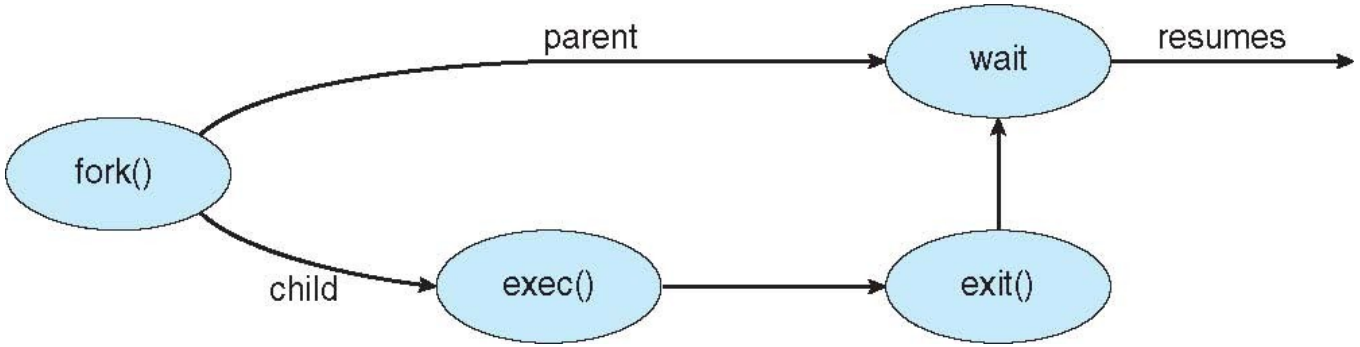
- New batch job
- Interactive Login
- Created by OS to provide a service
- Spawned by existing process

Process Creation

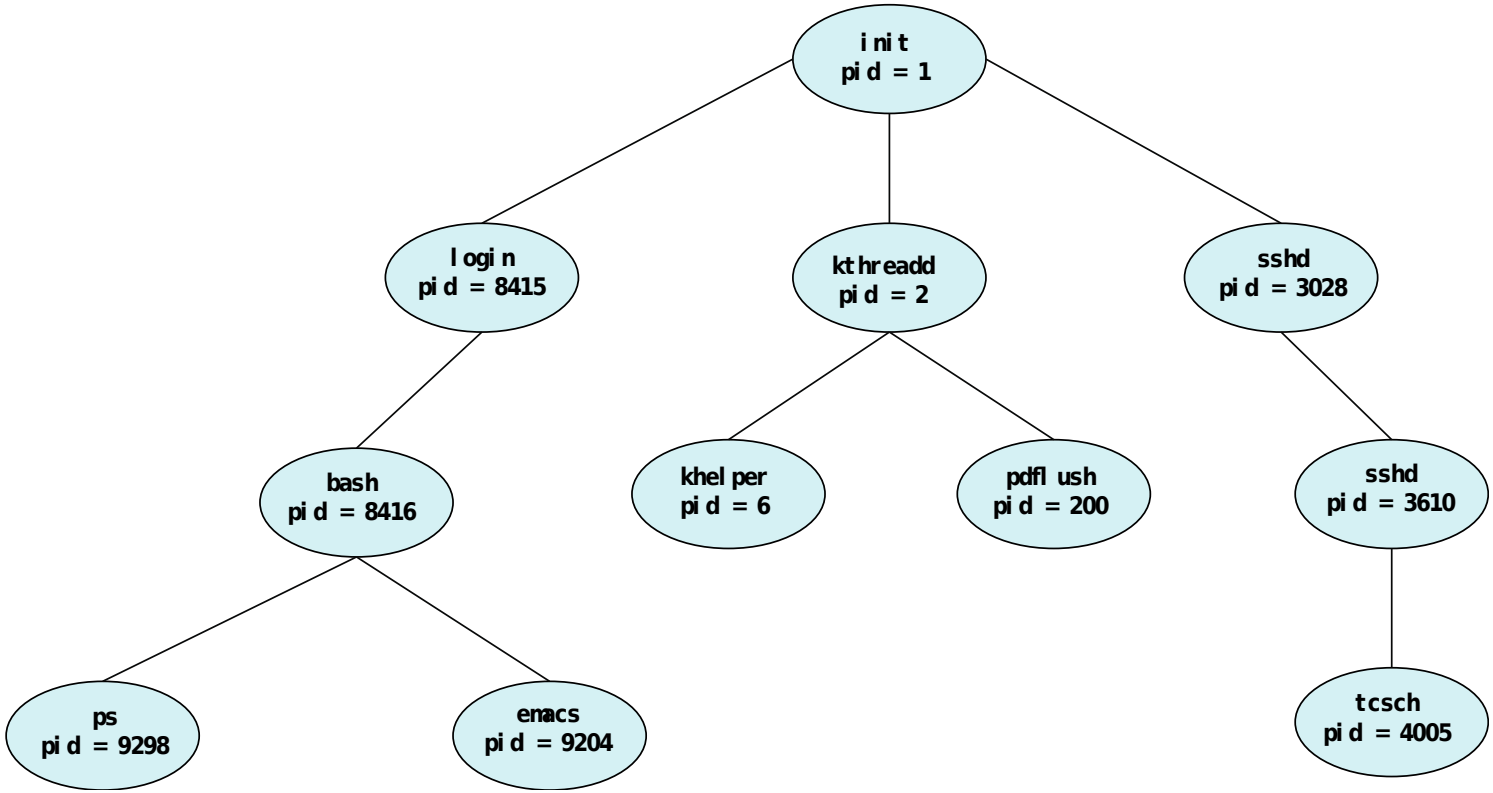
- Once the OS decides to create a new process it:
  - Assigns a unique process identifier
  - Allocates space for the process
  - Initializes process control block
  - Sets up appropriate linkages
  - Creates or expand other data structures

Process Creation in Linux

- A parent process creates the child process
- Each process has a **PID**
- 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



A Tree of Processes in Linux



Parent-child relationship

- 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

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 Death

- Normal Completion
- Memory unavailable
- Protection error
- Operator or OS Intervention

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 exists 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

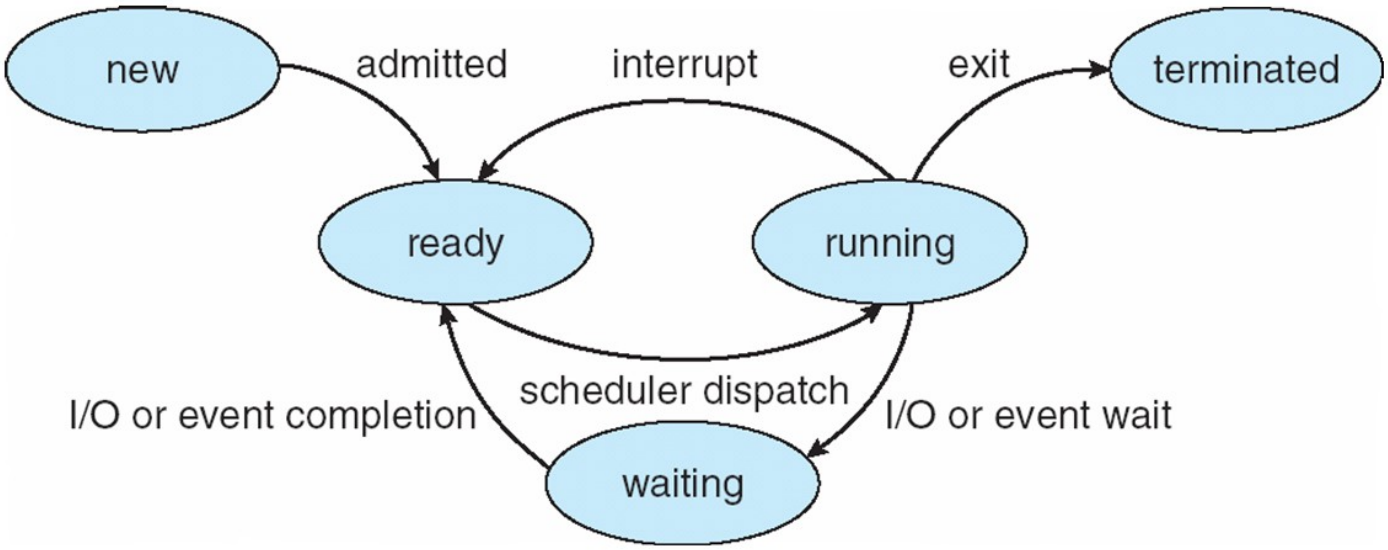
```
pid = wait(&status);
```

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

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

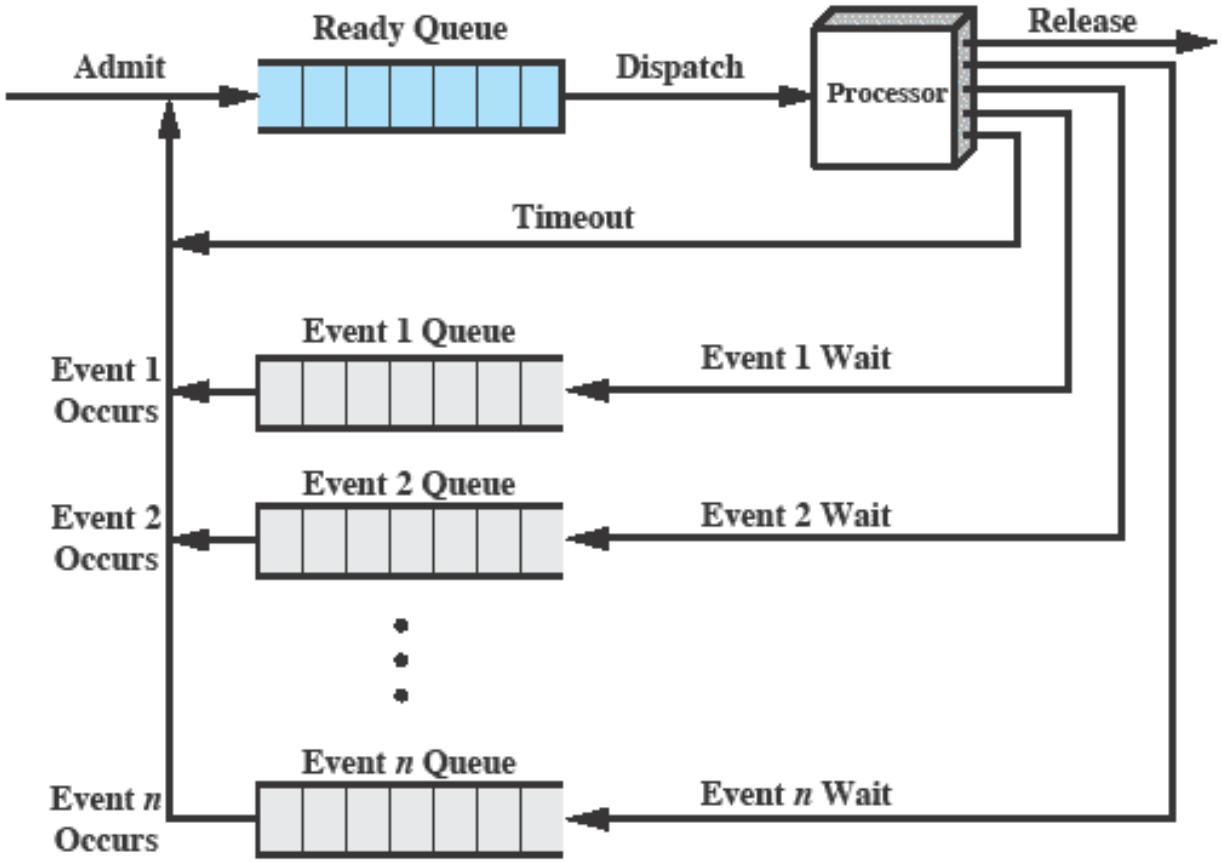
Diagram of Process State



Process Scheduling

- **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

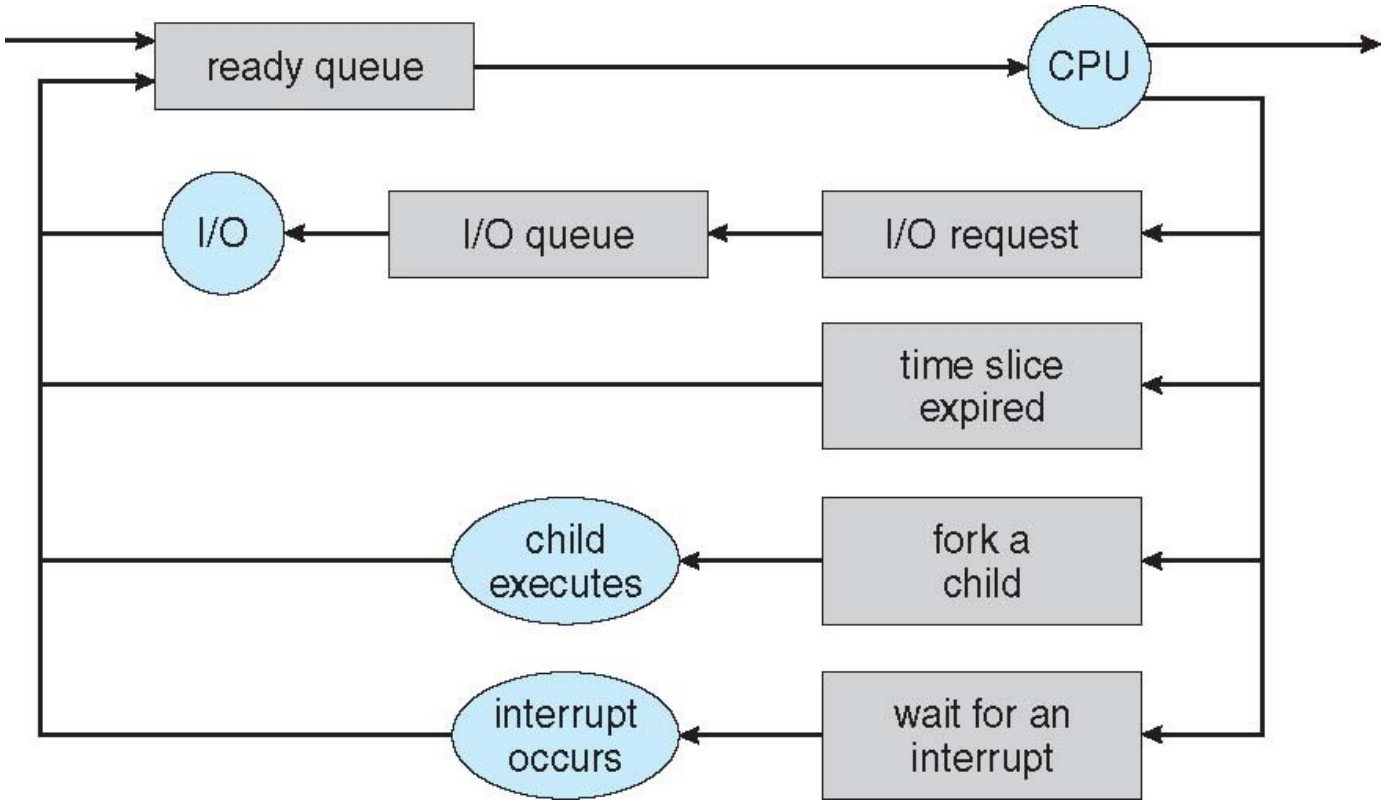
Multiple Blocked Queues



(b) Multiple blocked queues

Representation of Process Scheduling

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



Switching Processes

- Several design issues are raised regarding process switching
  - What events trigger a process switch?
  - We must distinguish between mode switching and process switching.
  - What must the OS do to the various data structures under its control to achieve a process switch?

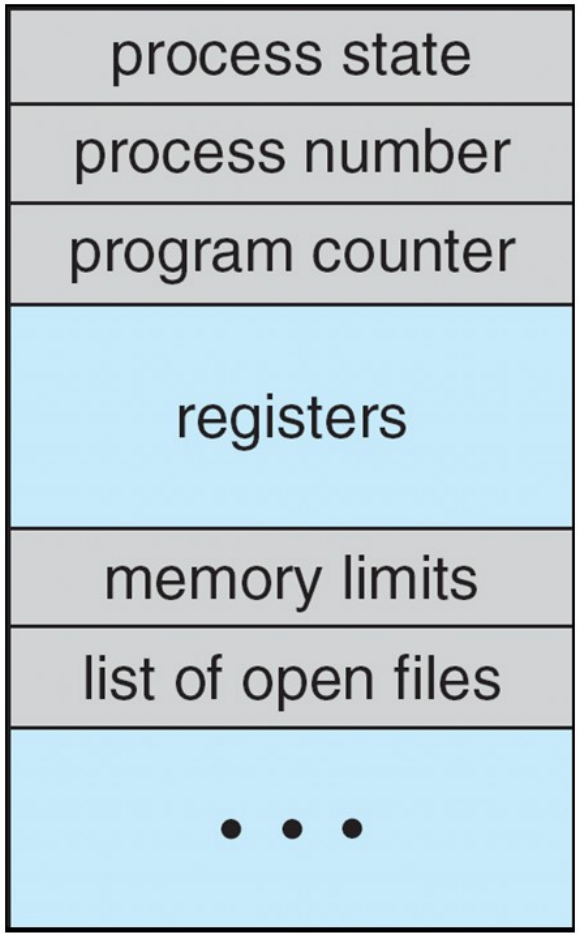
When to switch processes

A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

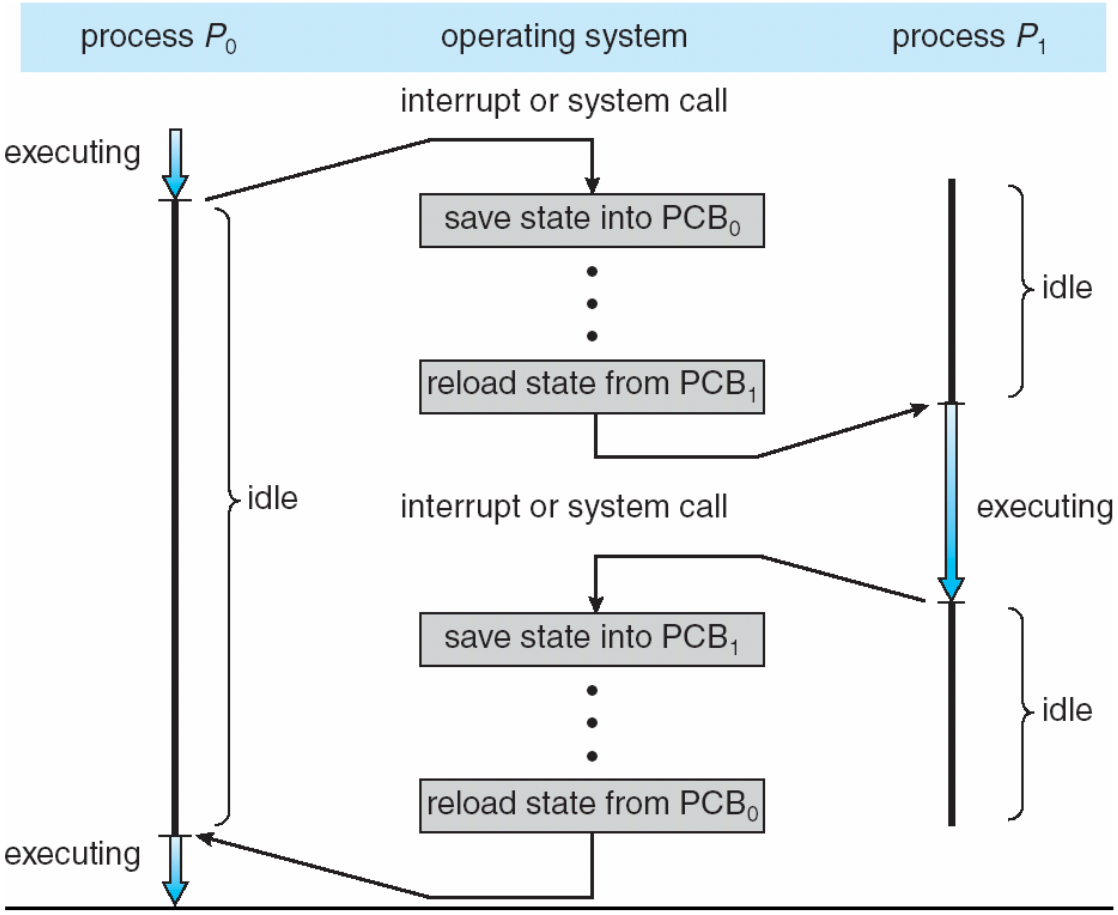
Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

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
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once



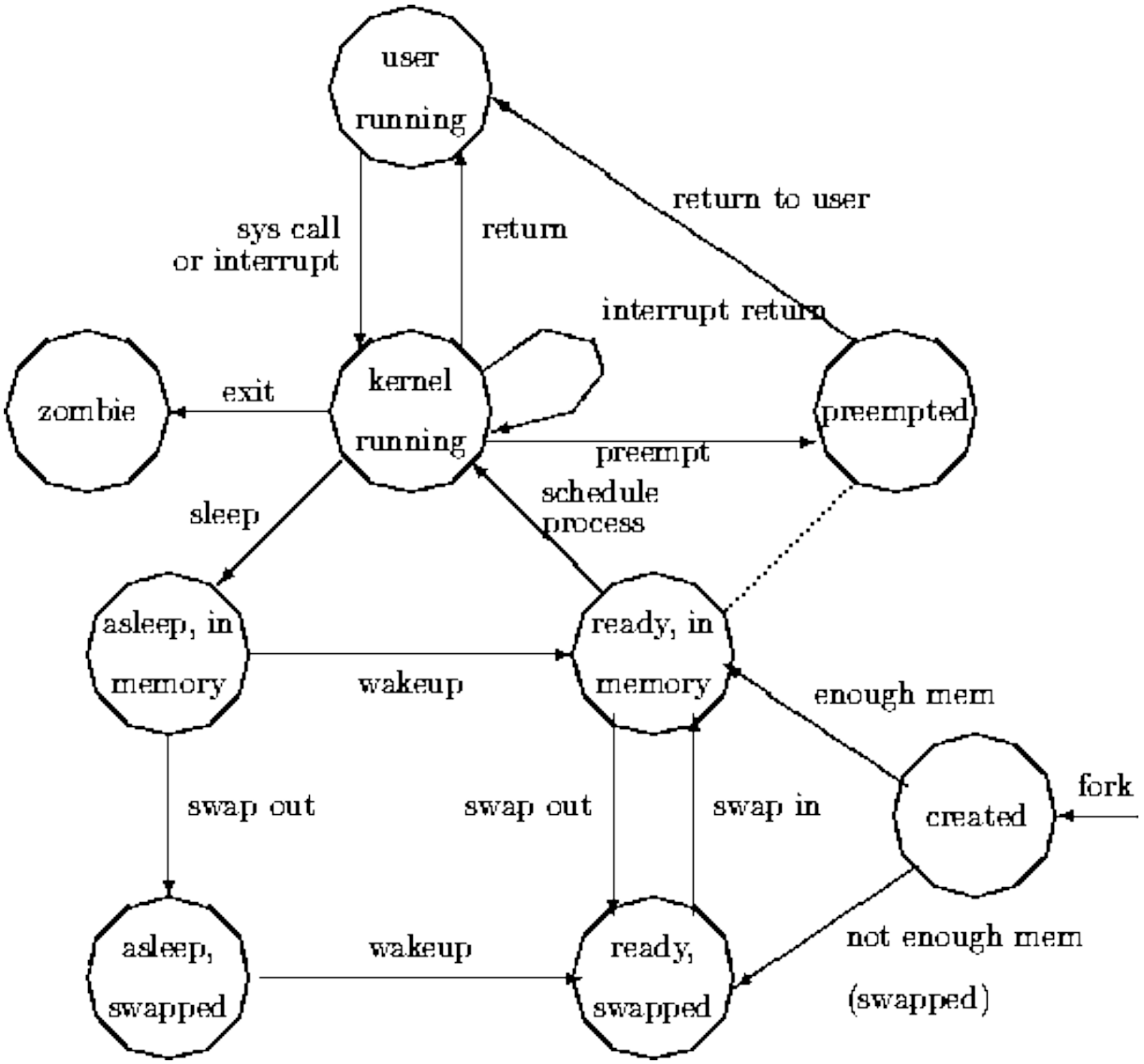
Context Switch From Process to Process



Steps for Switching Processes

1. Save context of processor including program counter and other registers
2. Update the process control block of the process that is currently in the Running state
3. Move process control block to appropriate queue – ready; blocked; ready/suspend
4. Select another process for execution
5. Update the process control block of the process selected
6. Update memory-management data structures
7. Restore context of the selected process

Linux process states



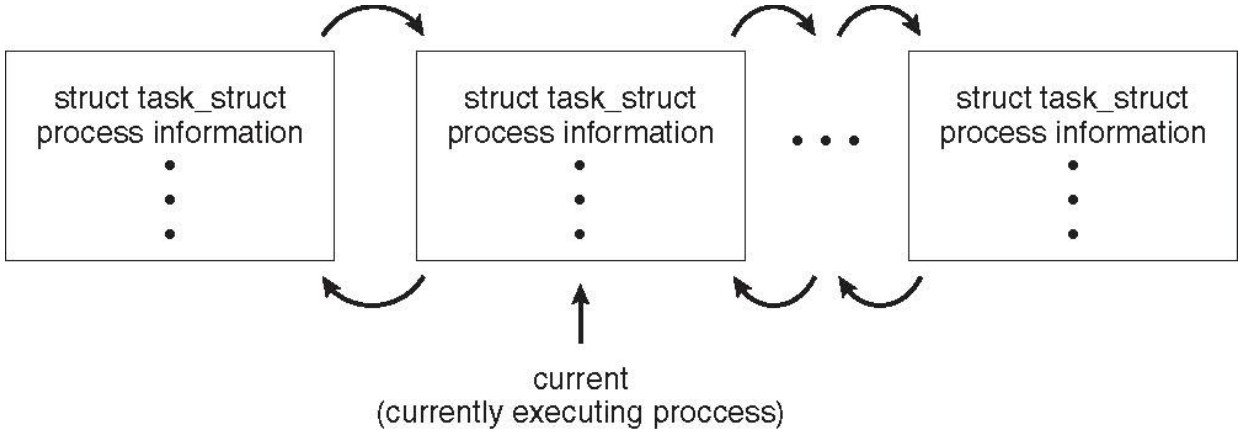
A Linux Process

- A process in Linux is a set of data structures that provide the OS with all of the information necessary to manage and dispatch processes.
- Three elements of the information:
  - user-level context,
  - register context, and
  - system-level context.

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 */
```



Ready Queue And Various I/O Device Queues