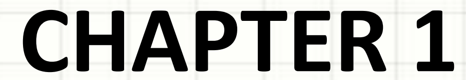


Process Concept

(Parallelism, Cooperation, and Competition)

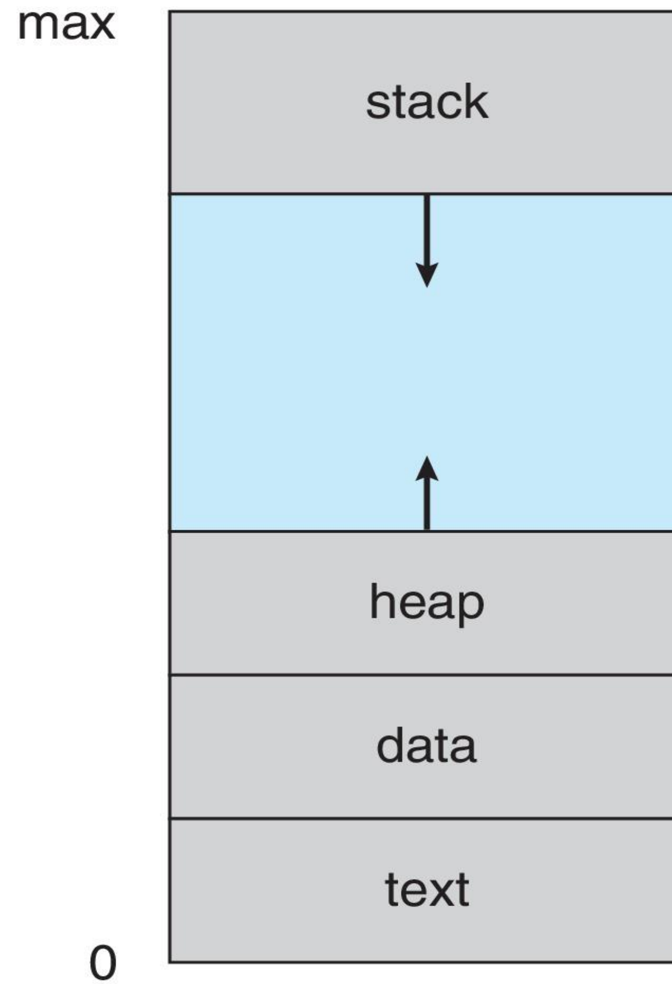
Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion.
- 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

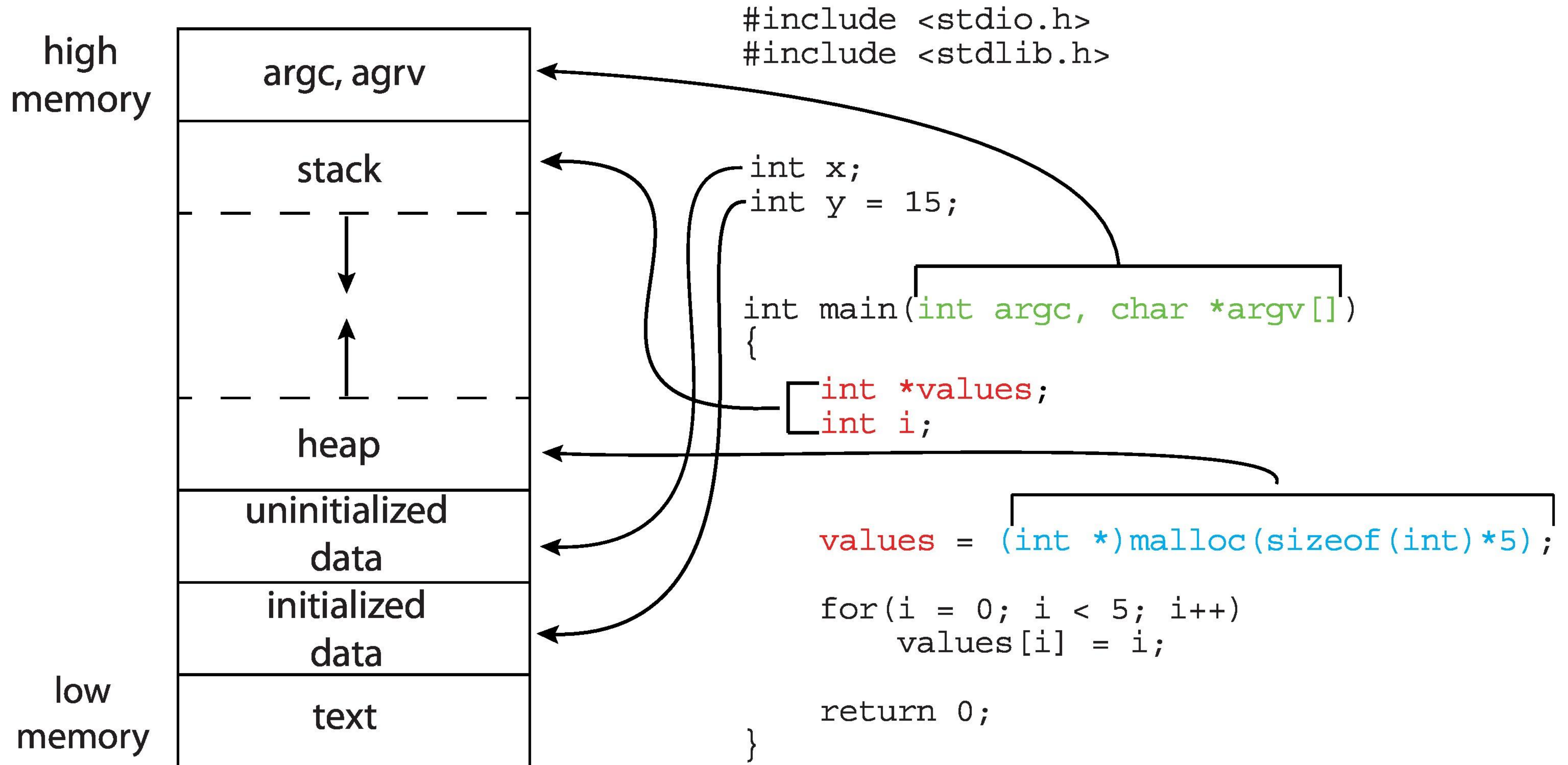
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is 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 in Memory



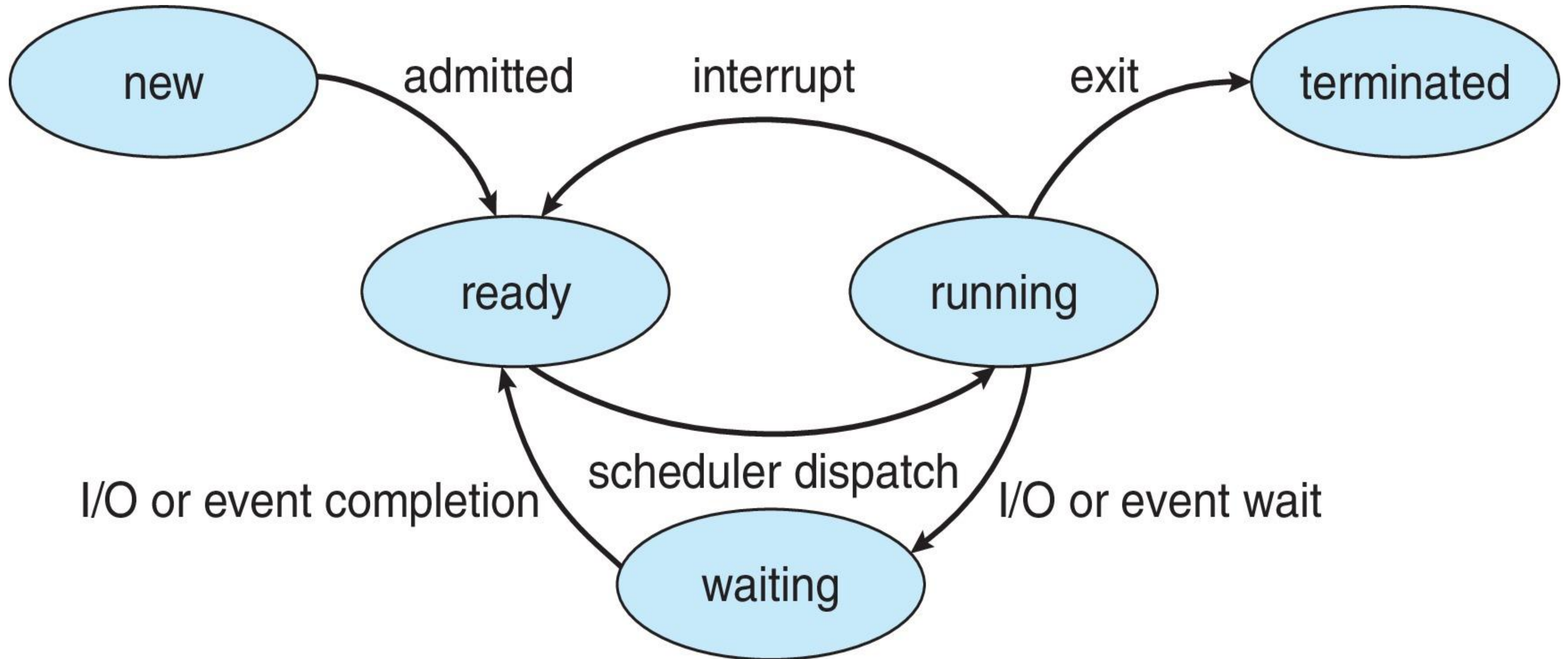
Memory Layout of a C Program



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

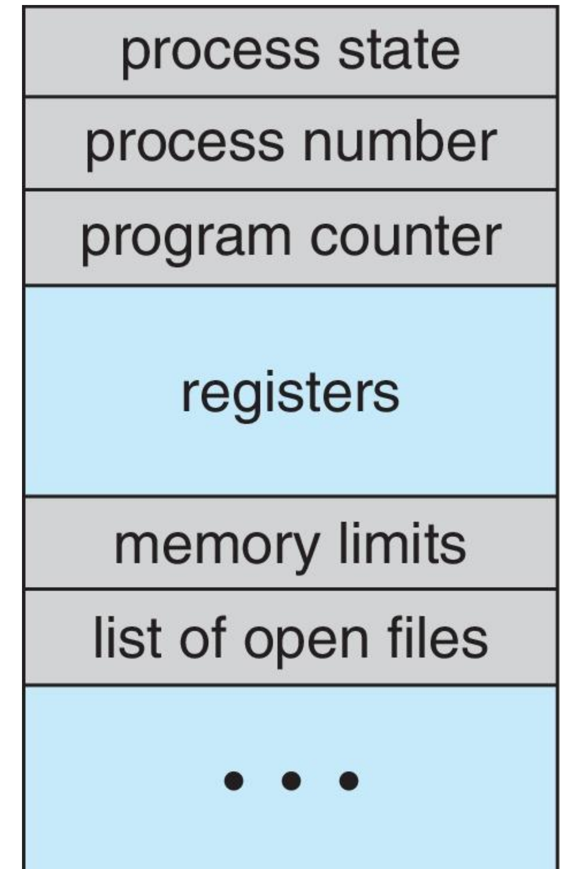
Diagram of Process State



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



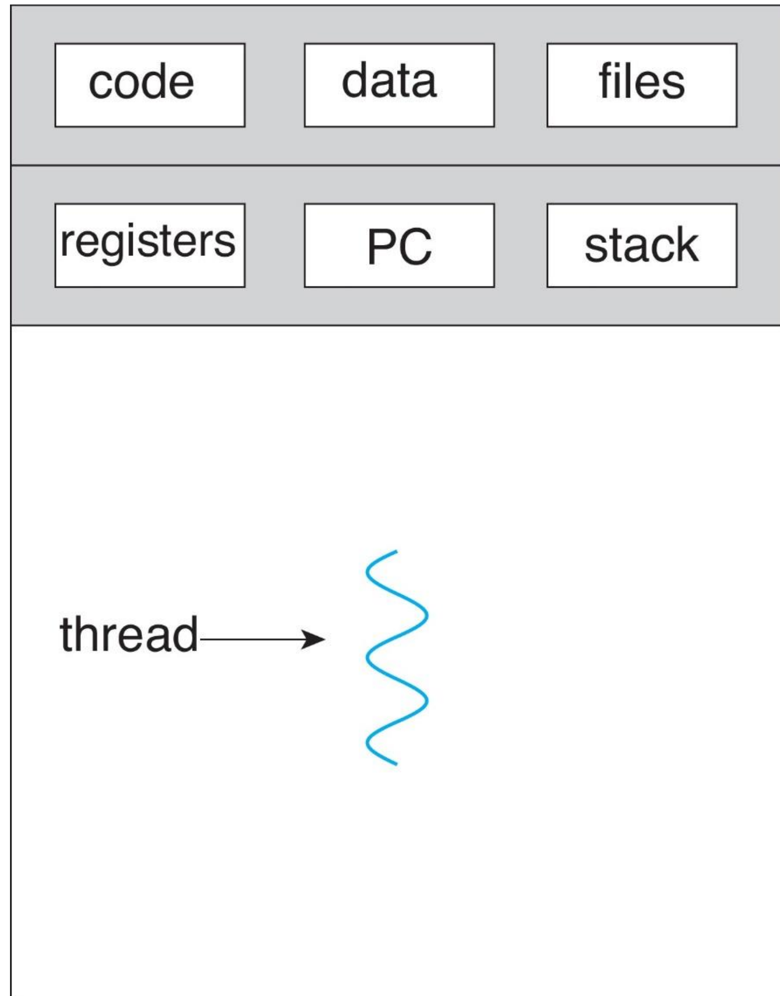
Process table

- The set of process control blocks forms a table called the process table.
- The process table is used by the operating system to manage processes and ensure that they run efficiently and without interfering with each other. For example, the operating system uses the process table to schedule processes, allocate resources to them, and terminate them when they are finished.

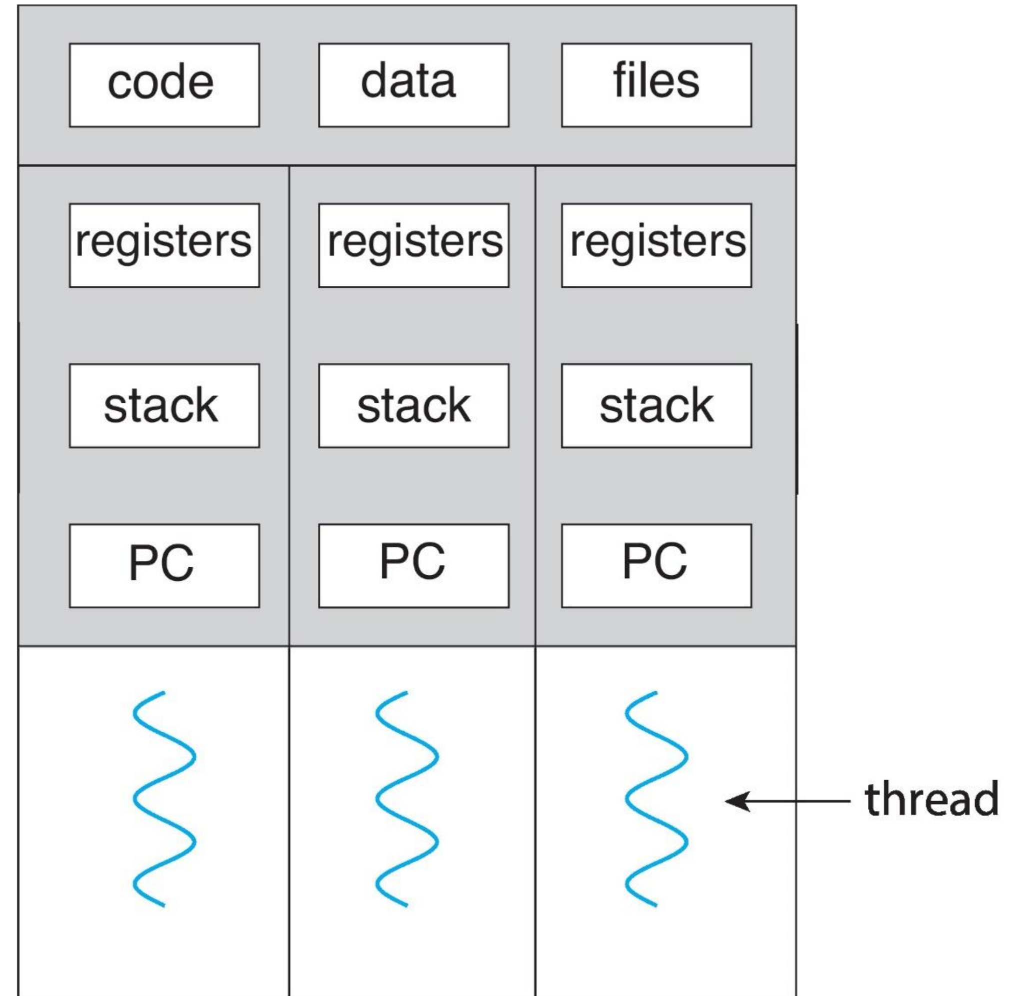
Threads

- 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**
- Must then have storage for thread details, multiple program counters in PCB
- ..

Single and Multithreaded Processes



single-threaded process



multithreaded process

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Operations on Processes

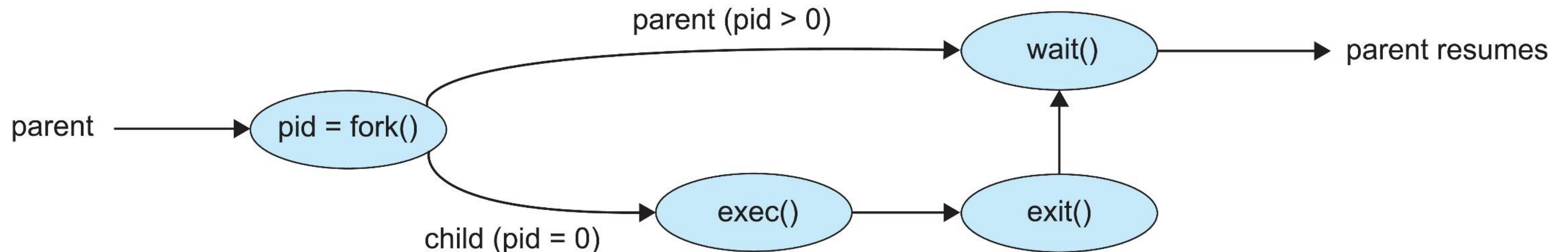
- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

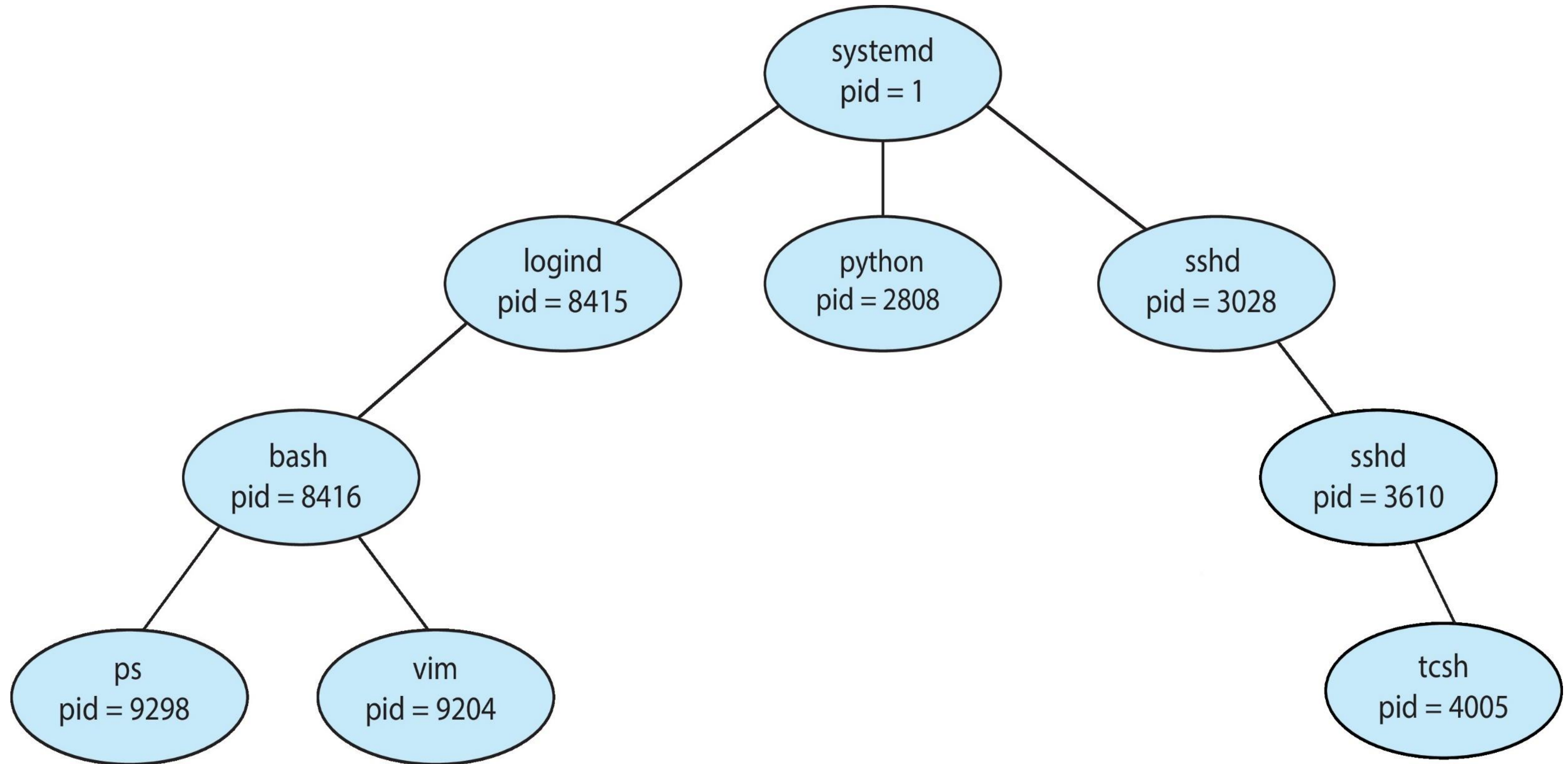
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset 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
  - Parent process calls `wait()` waiting for the child to terminate



A Tree of Processes in Linux



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 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**

The Role of Competition and Cooperation in Process Management

- In a computer system, processes are independent units of execution that can run concurrently to execute programs. During their execution, processes may interact with each other in a variety of ways.

- Interactions between processes can be classified into two broad categories: **cooperation and competition**.

The Role of Competition and Cooperation in Process Management

- **Cooperation:** occurs when two or more processes work together to achieve a common goal. This type of interaction can be classified into two categories: ***direct cooperation*** and ***indirect cooperation***.
  - ***Example***. *Salary calculation and printing of paystubs.*
- **Direct cooperation:** occurs when processes share data or resources. In this case, processes are aware of each other and can directly access each other's data or resources.

- **Indirect cooperation:** occurs when processes exchange messages. In this case, processes do not need to be aware of each other to communicate.

The Role of Competition and Cooperation in Process Management

- **Examples:**
- **Direct cooperation by sharing variables:** Two processes can share a common block of memory to exchange data. For example, two processes might share a variable to keep track of the current state of a game.

- **Indirect cooperation by exchanging messages:** Two processes can send messages to each other to coordinate their actions. For example, a process might send a message to another process to request data or to signal that it is finished with a task.

The Role of Competition and Cooperation in Process Management

- **Competition**: If processes are not aware of each other, they ignore each other's existence but may enter into conflicts for the use of critical resources (shared/common) (memory, processor, peripheral, file, etc.).

- To resolve conflicts between processes, resource usage rules are used. These rules aim to serialize the use of a given resource by processes over time.
  - ***Example***. *Access to a bank account*

CONCEPT OF CRITICAL SECTIONS

- The code executed by a process can be grouped into sections. Some of them need to access critical resources, while others do not. The former are called critical sections. To avoid access conflicts, a mechanism is essential to properly synchronize the execution within critical sections.
  - **Example**: Consider a bank account that is shared by two processes: P1 and P2. Both processes need to be able to access the account balance and update it.