TUNIS
SOUTH MEDITERRANEAN
UNIVERSITY

Chapter 2: Processes

Presented by: **Dr. Ikram Saadaoui**

Process Concept

- **Program** is a passive entity that is usually found on hard drives or magnetic disks
- **Process** is an active entity that starts when a program file is loaded into memory.
- One program can start many processes:
  - Consider 10 instances of FireFox process (10 tabs)
  - Consider multiple users executing the same program
- A process itself can be an execution environment for other process. Example : The command java runs the JVM as an ordinary process, which in turn executes the Java program in the virtual machine.
- process execution must progress sequentially. No parallel execution of instructions of a single process

Process in Memory

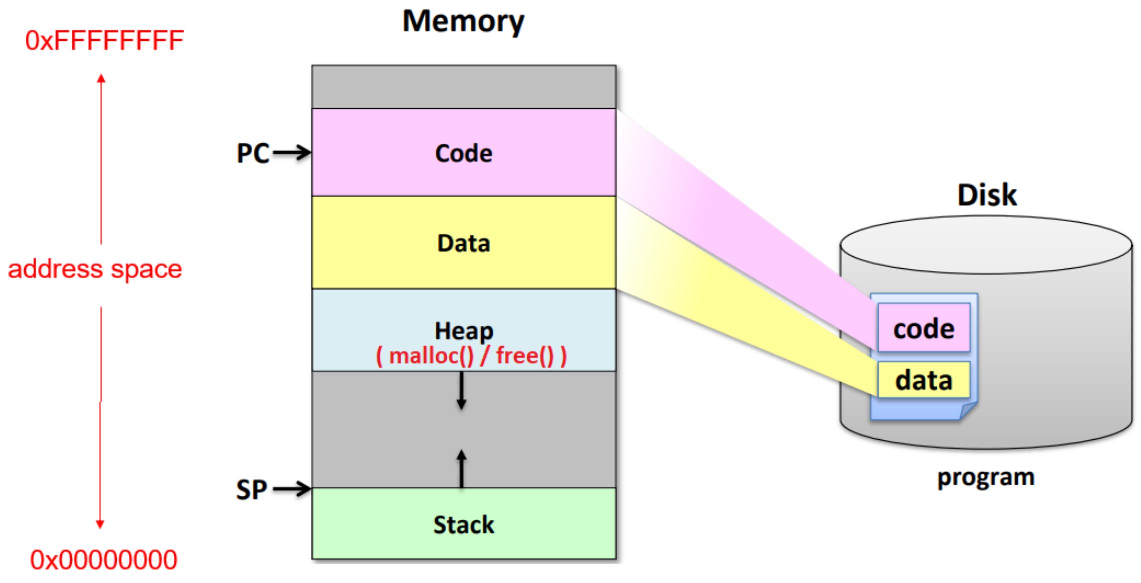
- Code section: machine code of the running program
- Data section: Consists of global and static variables that are initialized by the programmer. It does not change at run-time.
- Stack containing temporary data

Function arguments

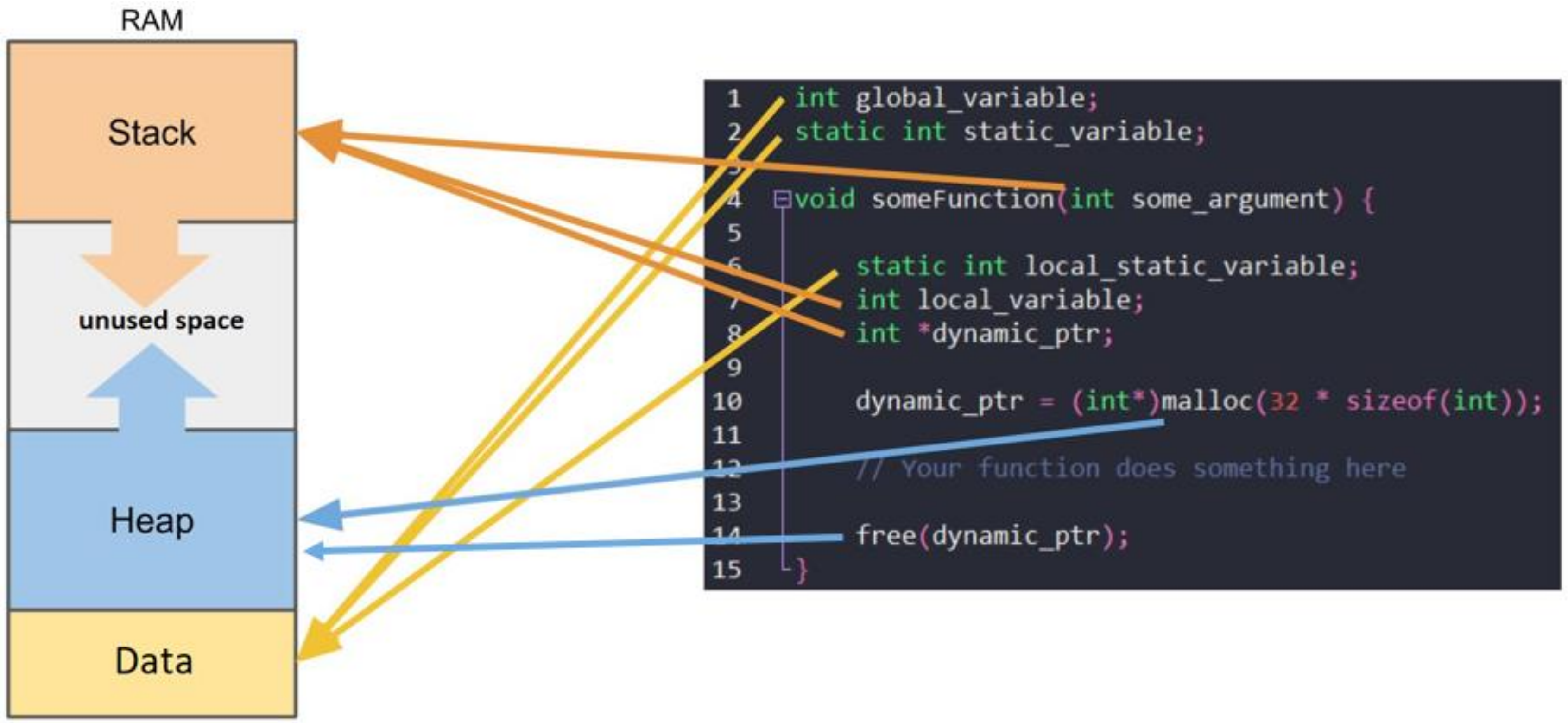
Return values

local variables

- Heap : memory reserved for dynamically allocated data:
  - malloc() allocates chunks of memory in the heap,
  - free() deallocates or frees data



Memory Layout of a C Program

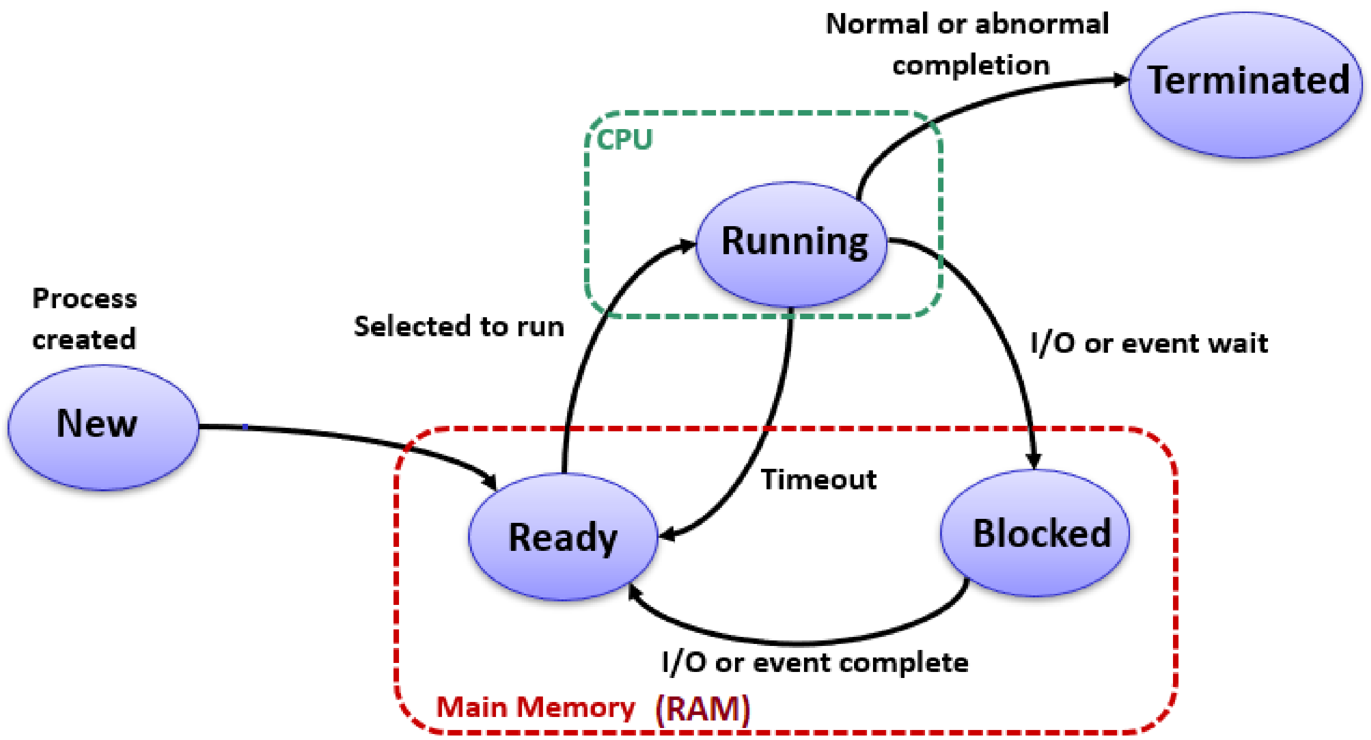


```c
int global_variable;
static int static_variable;

void someFunction(int some_argument) {

    static int local_static_variable;
    int local_variable;
    int *dynamic_ptr;

    dynamic_ptr = (int*)malloc(32 * sizeof(int));

    // Your function does something here

    free(dynamic_ptr);
}
```

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

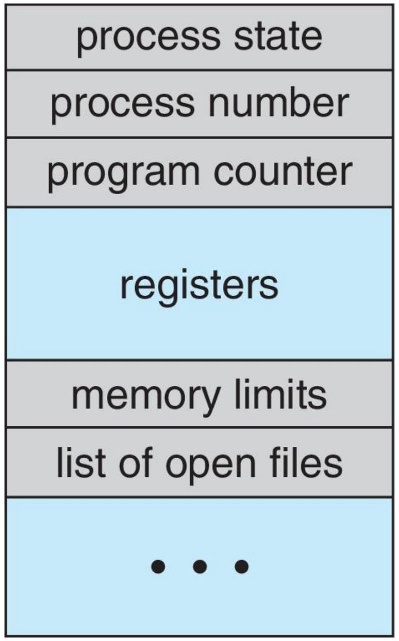
Diagram of Process State



Process Control Block (PCB)

When a process is created, OS allocates a PCB for it. The PCB is a data structure holding information associated with each process :

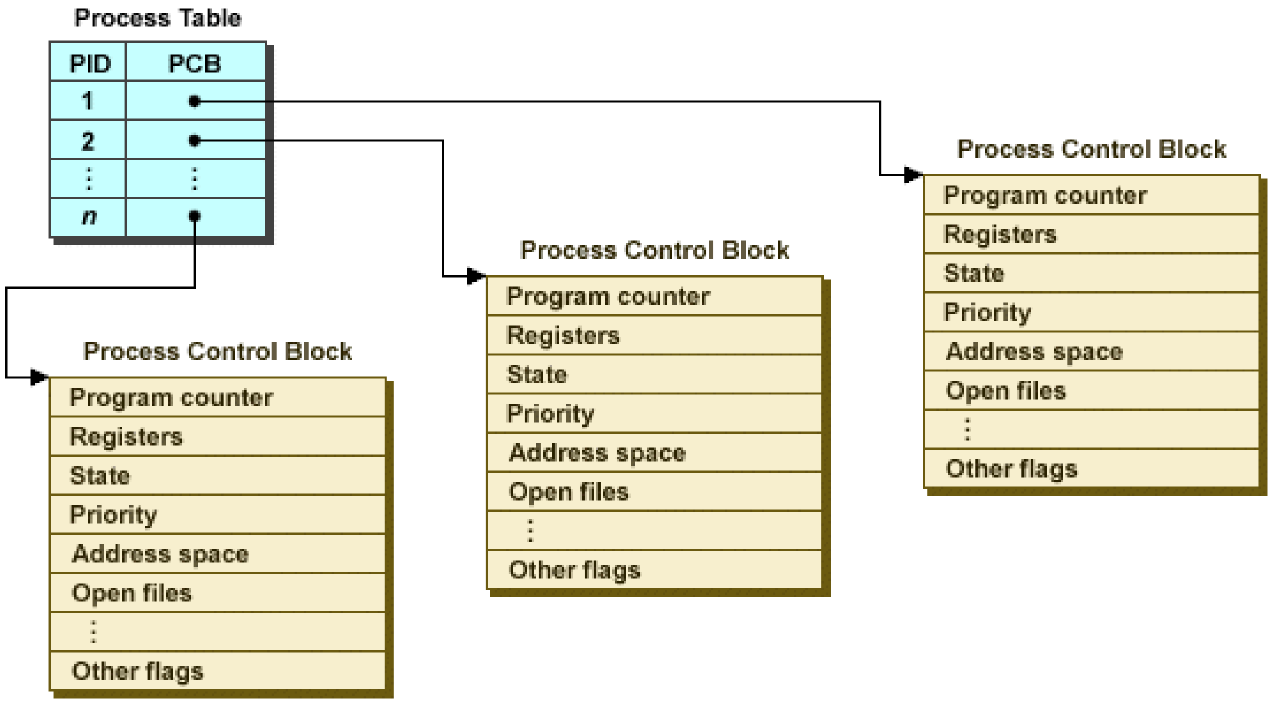
- Process state (ready, waiting, running, . . .)
- Process Number (Process ID)
- CPU registers
  - Program counter (PC), indicating the next instruction to be executed
  - Stack pointer (SP) stores the address at the top of the stack
- CPU scheduling info. (priority, queues)
- Memory-management info. (base, limit)
- I/O status info. (open files tables)



When a process is terminated OS deallocates (deletes) the PCB.

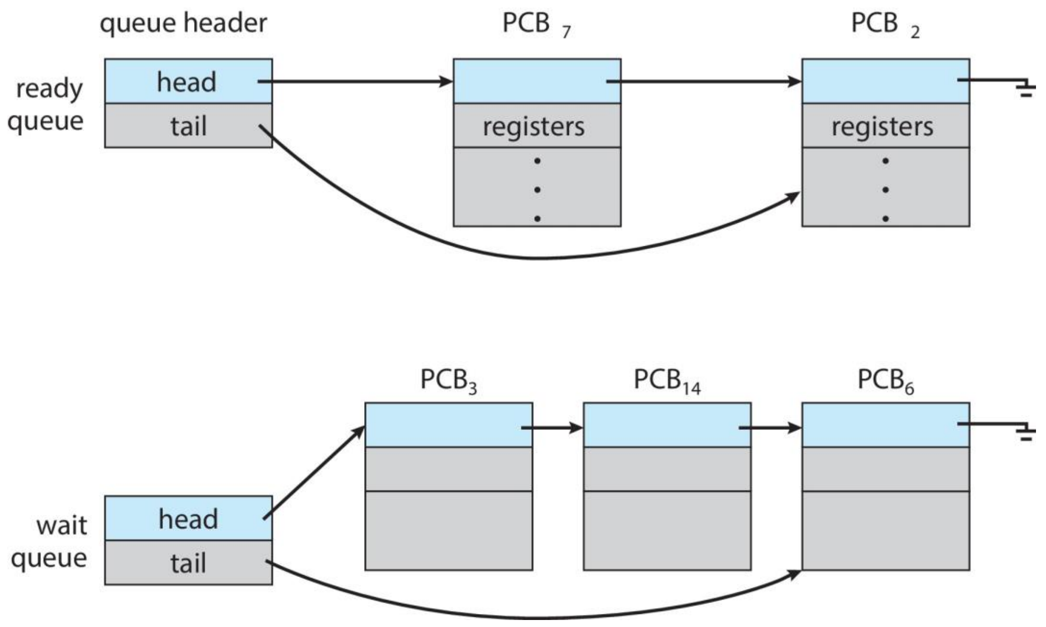
Process Control Block (PCB)

- Operating systems use one main data structure to manage the created processes on a computer, called the process table.
- The process table contains all the information essential to the operating system to ensure consistent management of processes.
- It is stored in the kernel's memory space, which means that processes cannot access it.



Process Scheduling

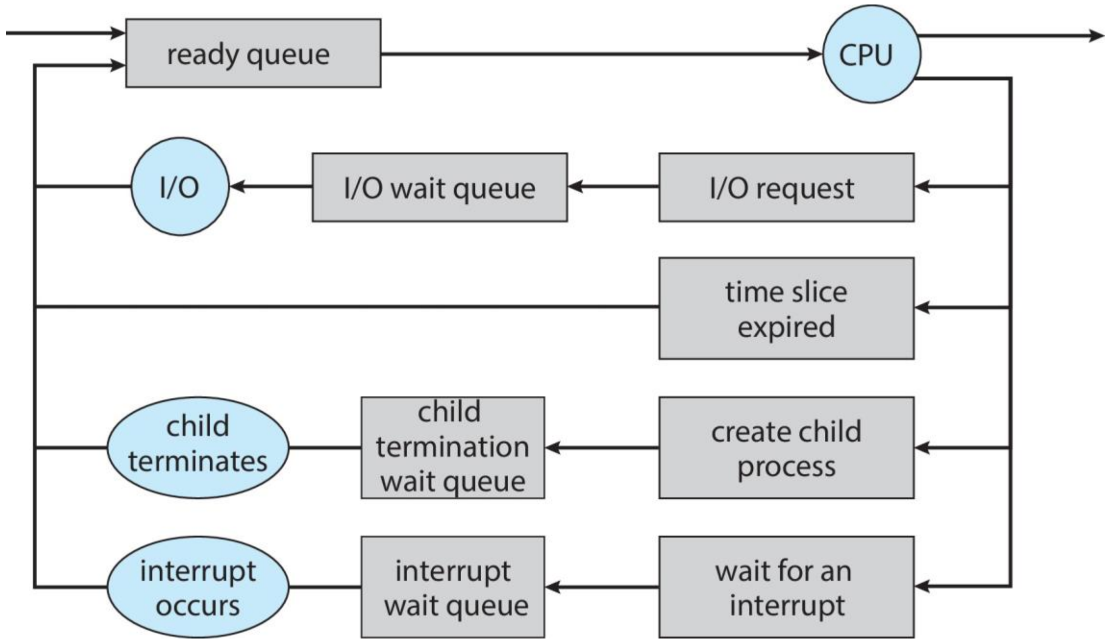
- The OS maintains a collection of queues (implemented with a linked list) that represent the state of all processes in the system:
  - **Ready queue** – a set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – a set of processes waiting for an event (i.e., I/O)



- As a process changes state, the OS moves its PCB from queue to queue

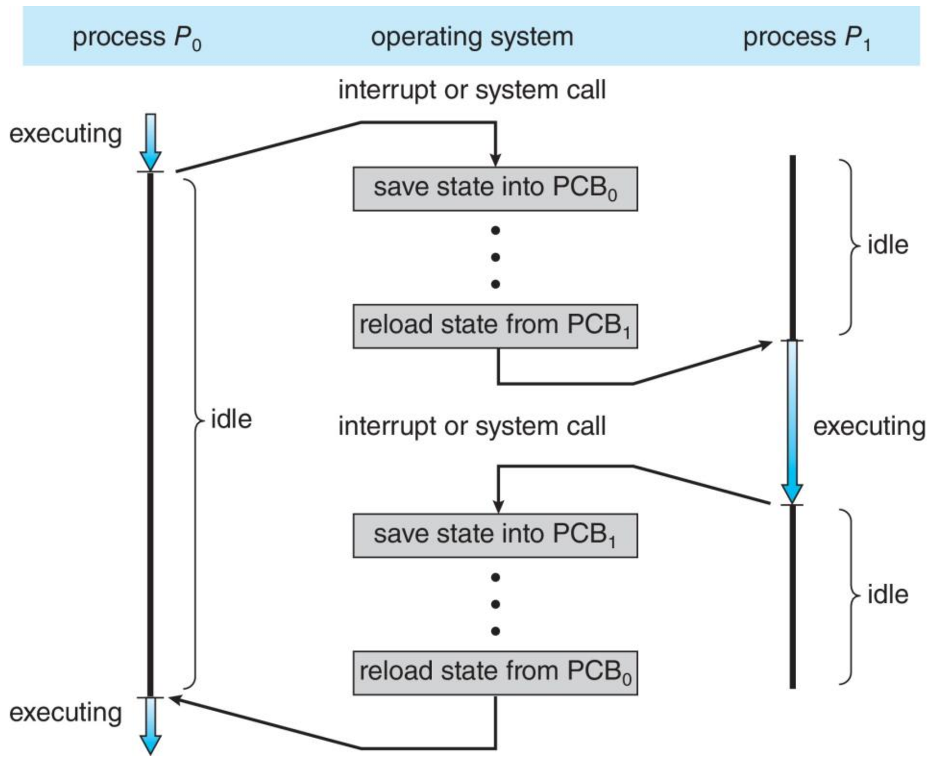
Representation of Process Scheduling

- **Process scheduler** selects a process among available processes for the next execution on the CPU core
  - Goal -- Maximize CPU use, quickly switch processes onto CPU core



Context Switch

- **Context** of a process represented in the PCB
- A **context switch** occurs when the CPU switches from one process to another.
  - When the CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process.



Context Switch

- 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
- The switch time is dependent on the hardware support
  - Some hardware provides multiple sets of registers per CPU

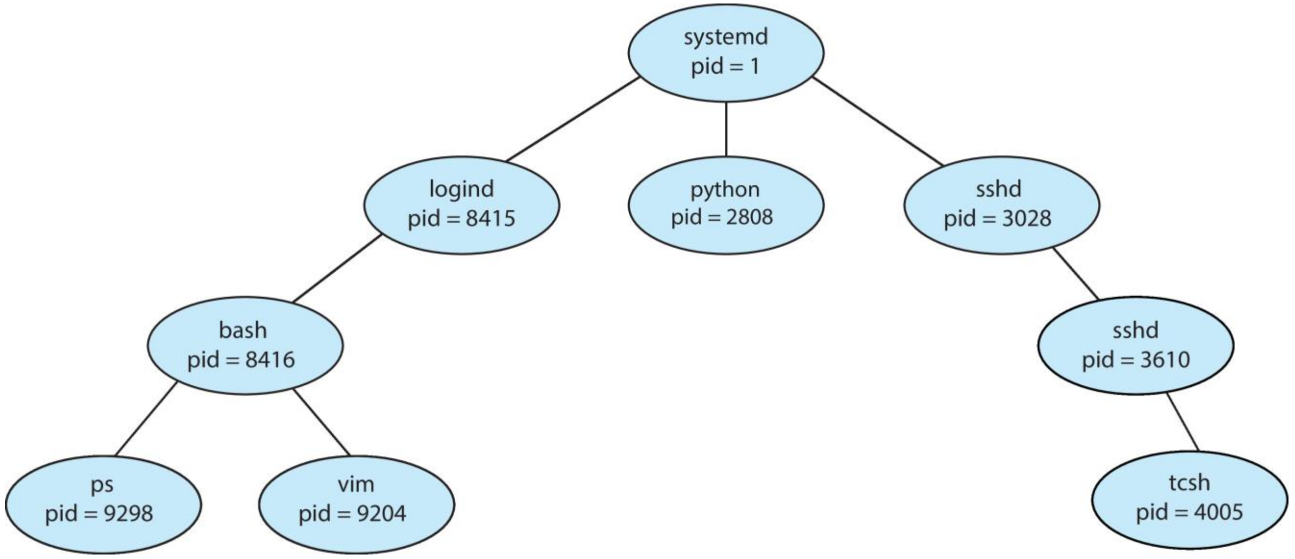
  ➔ multiple contexts loaded at once

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



Process Creation (Cont.)

```
pid_t fork(void);
```

- **Fork** system call is used to create a new (child) process by duplicating the calling (parent) process.

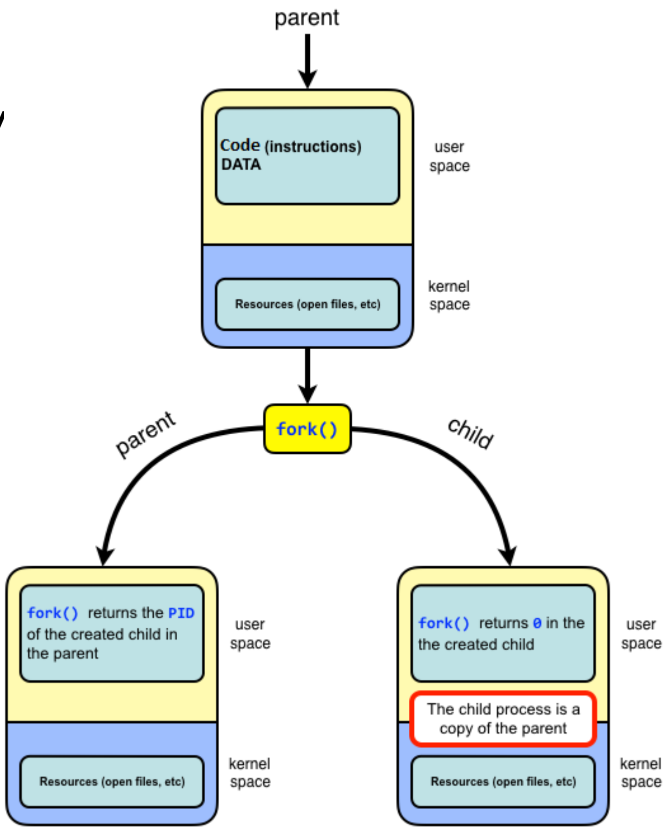
- The new child process is an exact copy of the parent process, except that it has its address space.
  - The child process has the same source code
  - the program counters have the same value, which means they refer to the same source code line

- The child process is independent of the parent process (both have their own PCBs and PIDs)
  - fork() creates and initializes a new PCB for the child process and places it on the ready queue

- The fork() function returns the child's PID in the parent process, and zero in the child process

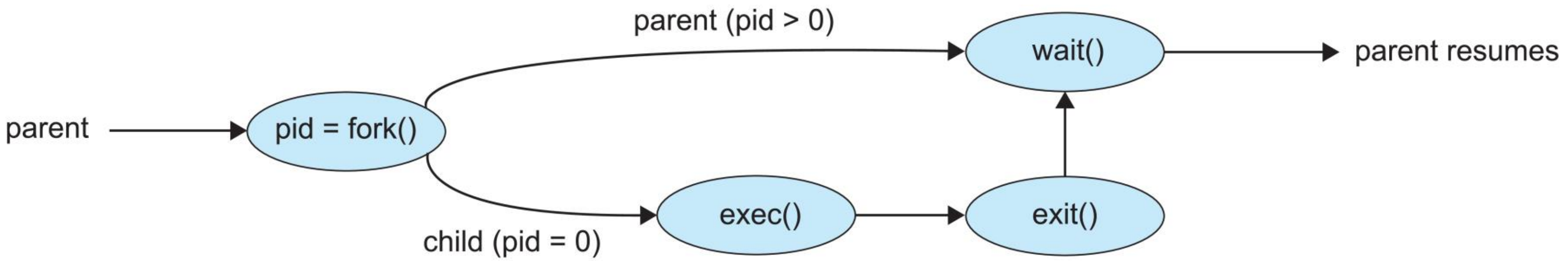
Process Creation (Cont.)

- **Resource sharing options:**
  - Parent and children share all resources
  - Children share a subset of parent's resources
  - Parent and child share no resources
- **Execution options**
  - Parents and children execute concurrently
  - Parent waits until children terminate
- **Address space**
  - The child is a duplicate of the parent
  - The child has a program loaded into it through exec() system call.


parent
Code (instructions)
DATA
user space
Resources (open files, etc)
kernel space
fork()
parent
child
fork() returns the PID of the created child in the parent
user space
Resources (open files, etc)
kernel space
fork() returns 0 in the the created child
user space
The child process is a copy of the parent
Resources (open files, etc)
kernel space

Process Creation (Cont.)

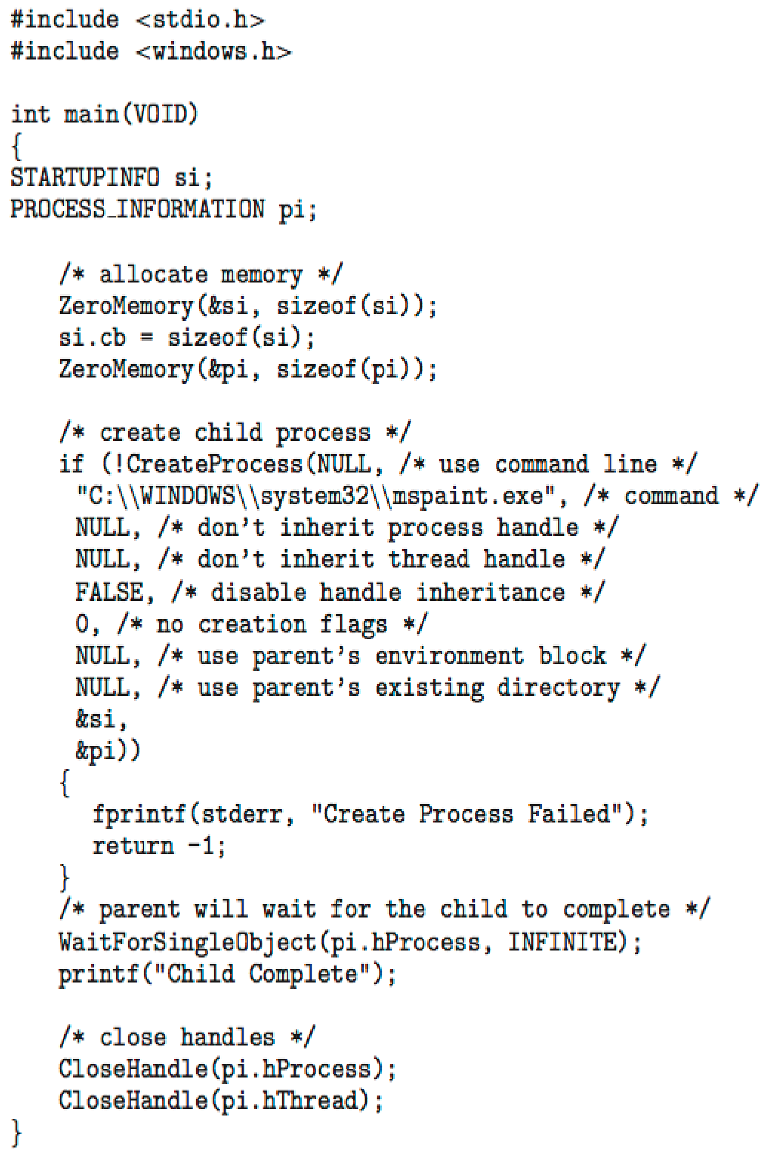
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



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;
}
```

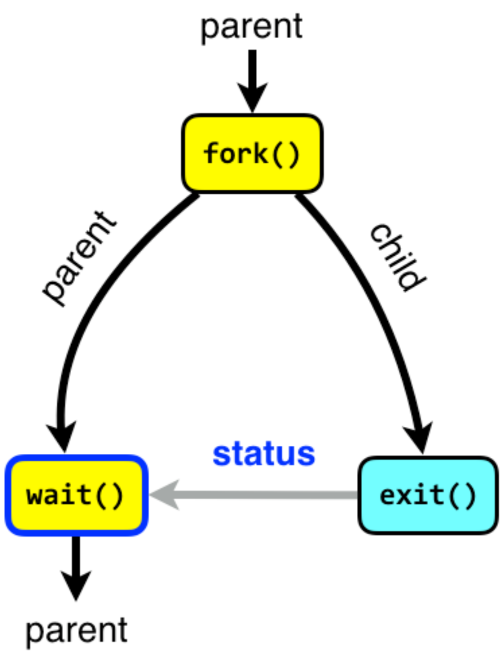
Creating a Separate Process via Windows API



```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination (exit)

- Process executes the last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - delivers **status** value from child to parent (via **`wait()`**)
  - Process resources are deallocated by the operating system



```
#include <stdlib.h>
void exit(int status);
```

Process Termination (wait)

- The parent process may wait for the termination of a child process by using the `wait()` system call.
- The call returns status value and the PID of the terminated process

```
#include <sys/types.h>
#include <sys/wait.h>
pid = wait(&status);
```

- If parent terminated without invoking `wait()`, child process is an **orphan**
- A **zombie** process is a process that has terminated but the entry for the process is still present in the process table, this happens because the parent process has not yet read the child's exit status using the wait() system call.
  - This is typically, a sign of a bug or programming error.

Process Termination (abort)

- The Parent may terminate the execution of the children's processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to a child is no longer required
  - The parent has exited, and the operating system does not allow a child to continue if its parent terminates.
- Some operating systems do not allow the child 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.