

Chapter 3: Processes





Chapter 3: Processes

- ❑ Process Concept
- ❑ Process Scheduling
- ❑ Operations on Processes
- ❑ Interprocess Communication
- ❑ IPC in Shared-Memory Systems
- ❑ IPC in Message-Passing Systems
- ❑ Examples of IPC Systems
- ❑ Communication in Client-Server Systems





Objectives

- ❑ Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- ❑ Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- ❑ Describe and contrast interprocess communication using shared memory and message passing.
- ❑ Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- ❑ Describe client-server communication using sockets and remote procedure calls.
- ❑ Design kernel modules that interact with the Linux operating system.





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





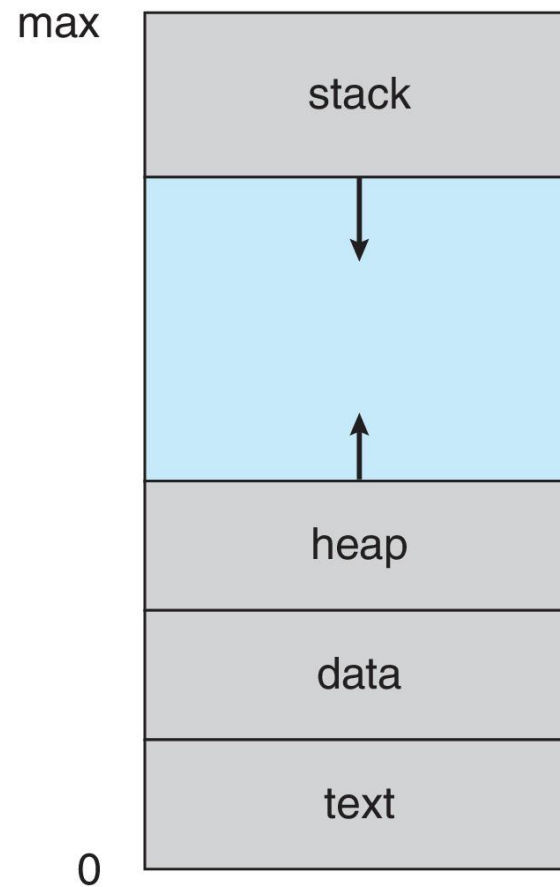
Process Concept (Cont.)

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



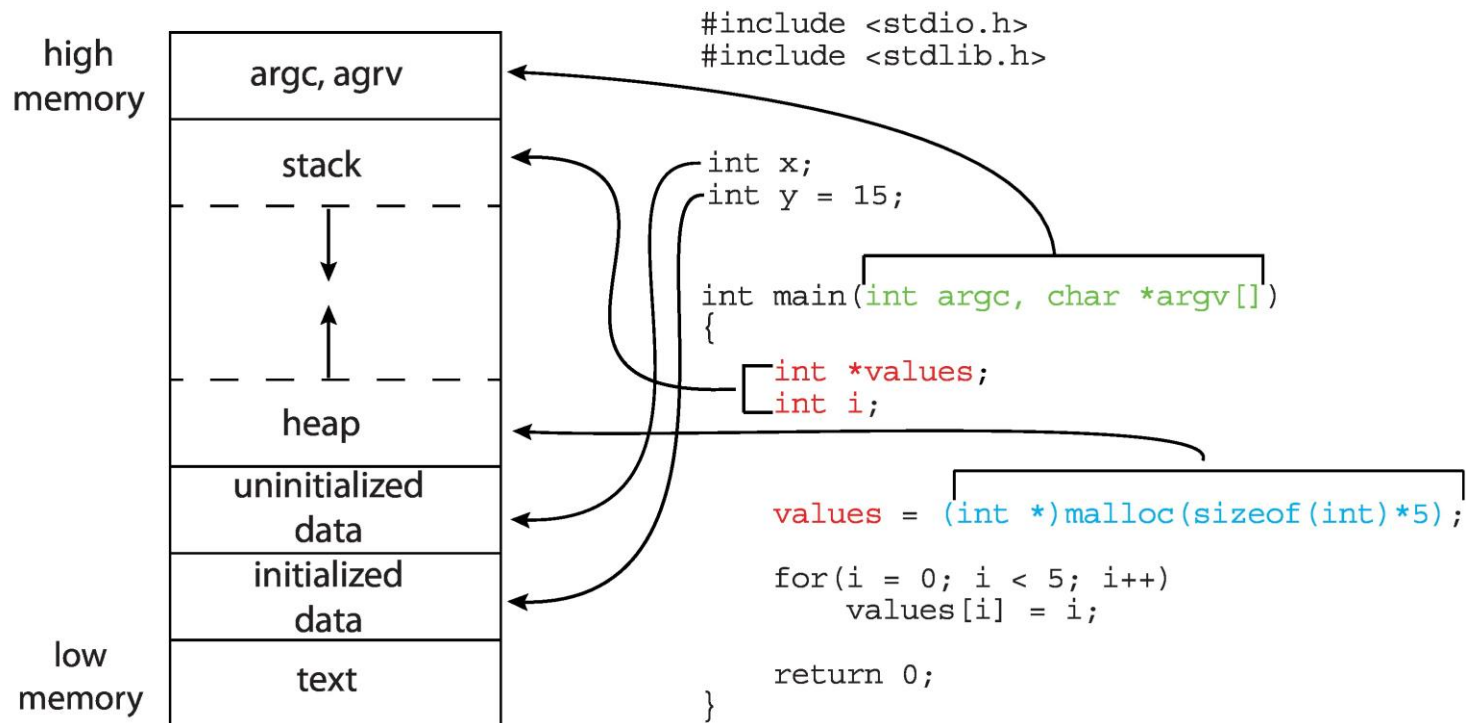


Process in Memory





Memory Layout of a C Program





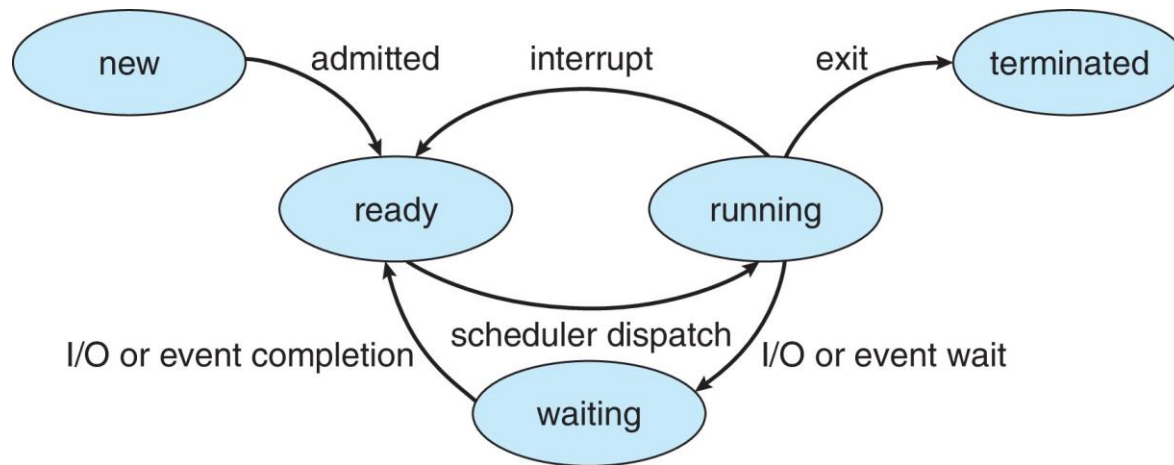
Process State

- 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 state
process number
program counter
registers
memory limits
list of open files
...





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
- ❑ Explore in detail in Chapter 4

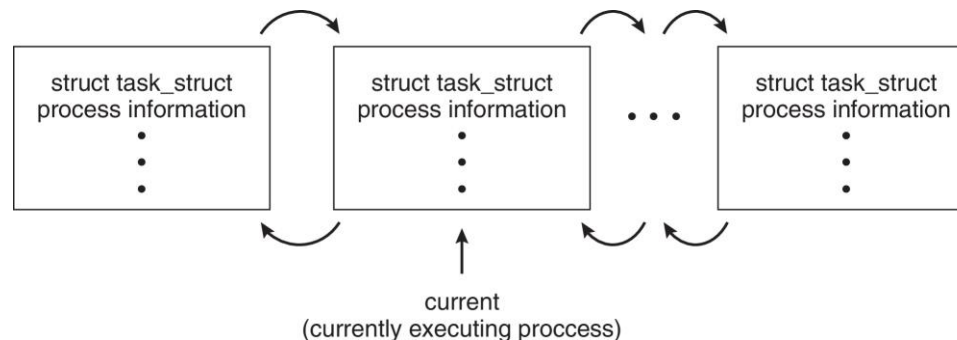




Process Representation in Linux

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 */
```





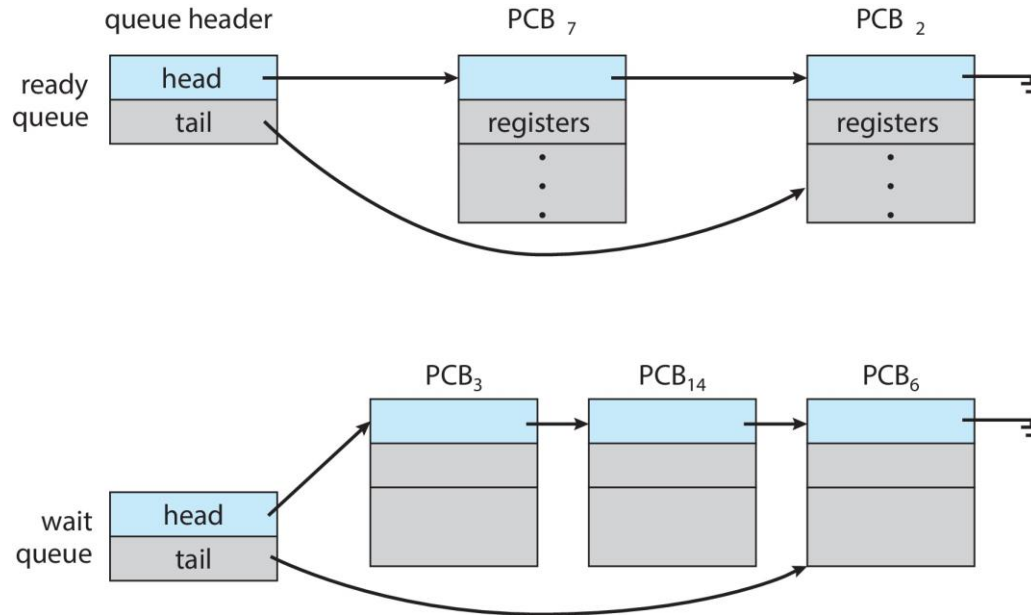
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core
- **Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
 - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
 - **Wait queues** – set of processes waiting for an event (i.e. I/O)
 - Processes migrate among the various queues



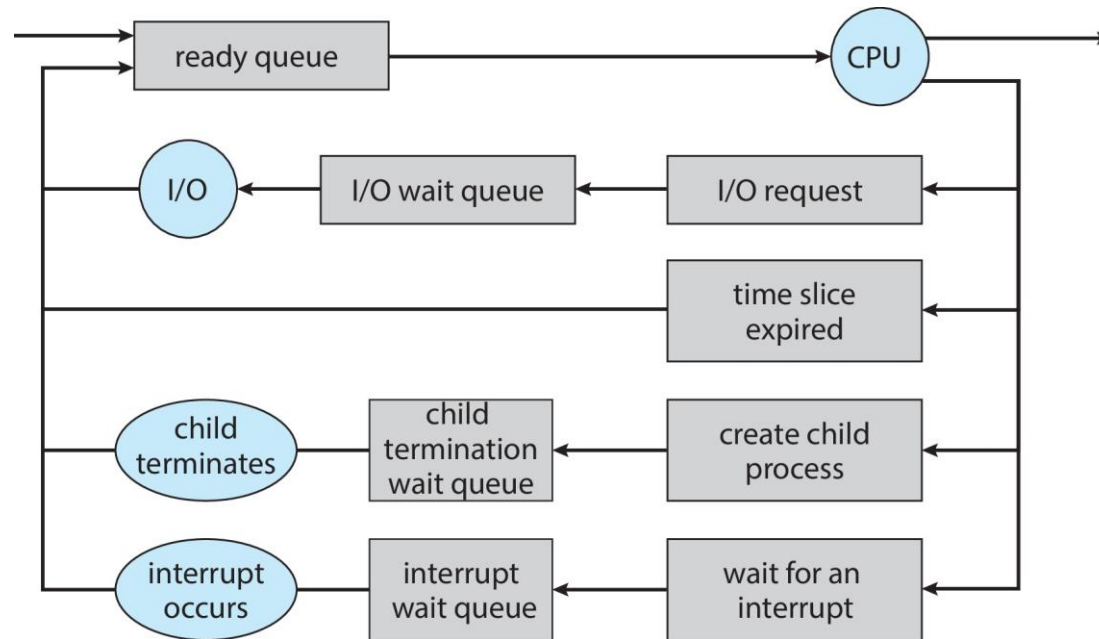


Ready and Wait Queues





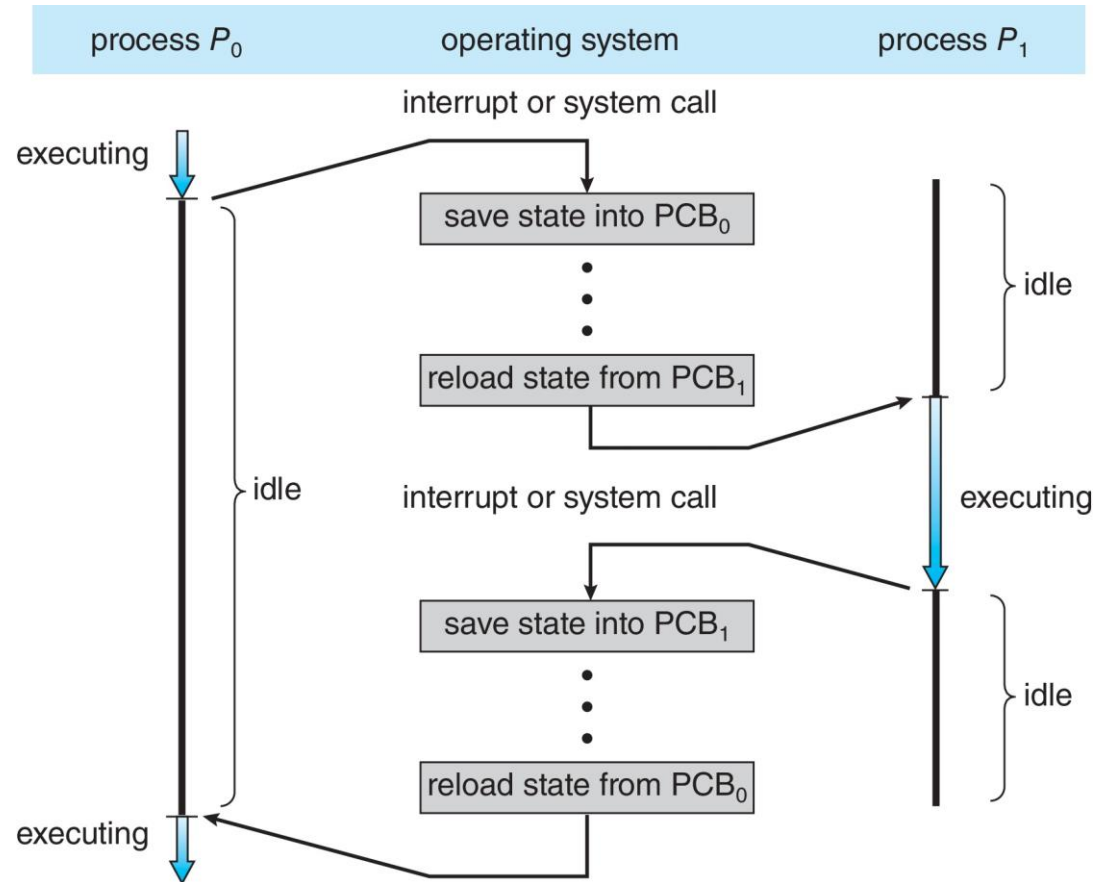
Representation of Process Scheduling





CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.





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





Multitasking in Mobile Systems

- ❑ Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- ❑ Due to screen real estate, user interface limits iOS provides for a
 - ❑ Single **foreground** process- controlled via user interface
 - ❑ Multiple **background** processes– in memory, running, but not on the display, and with limits
 - ❑ Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- ❑ Android runs foreground and background, with fewer limits
 - ❑ Background process uses a **service** to perform tasks
 - ❑ Service can keep running even if background process is suspended
 - ❑ Service has no user interface, small memory use





Operations on Processes

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





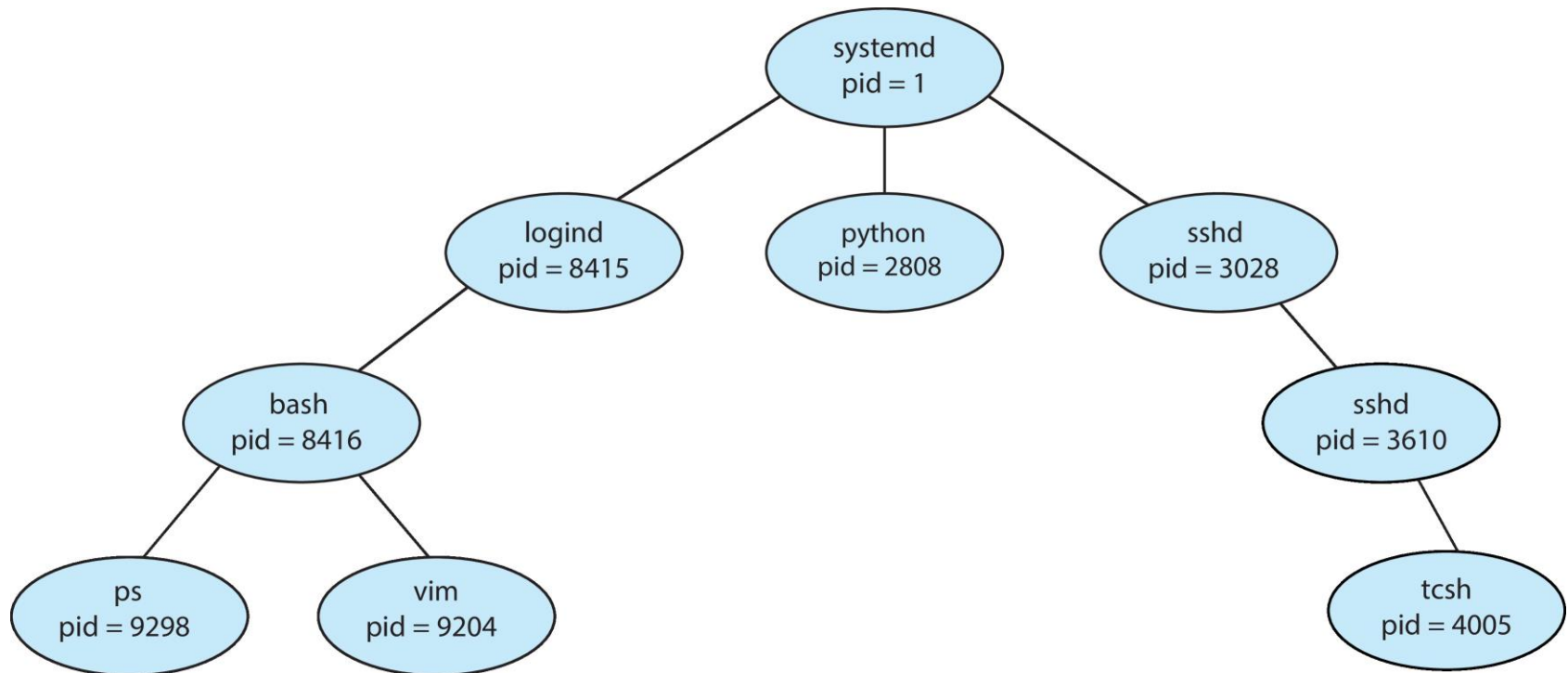
Process Creation

- ❑ **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





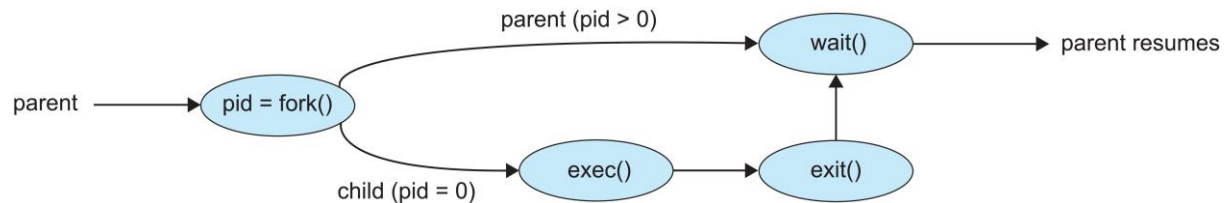
A Tree of Processes in Linux





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()** 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

- 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 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.
 - 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**





# Android Process Importance Hierarchy

---

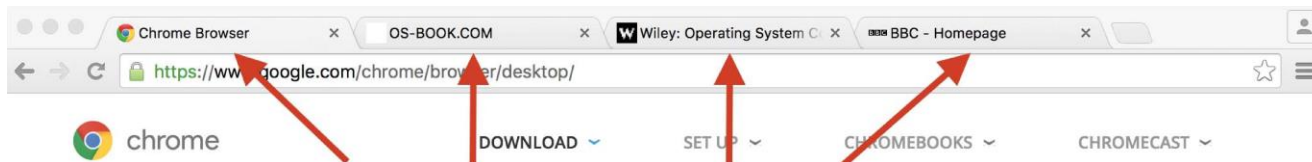
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.





# Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



Each tab represents a separate process.

