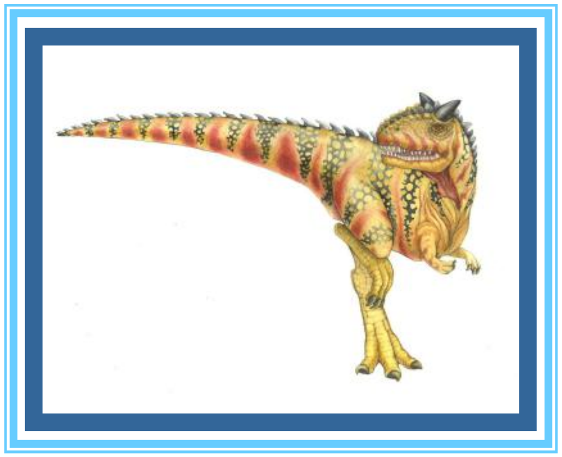
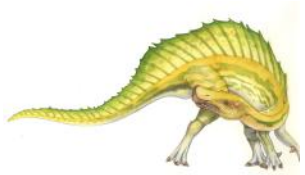
4: Process



4: Process

- Process Concept
- Process Scheduling
- Operations on Processes



Objectives

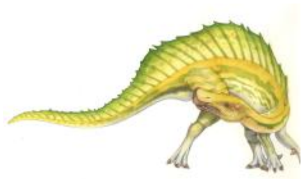
- Identify the separate components of a process and illustrate how they are represented in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution in memory; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers (all current data of the program inside the CPU)
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables and static 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 by the operating system's **loader**.
- 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



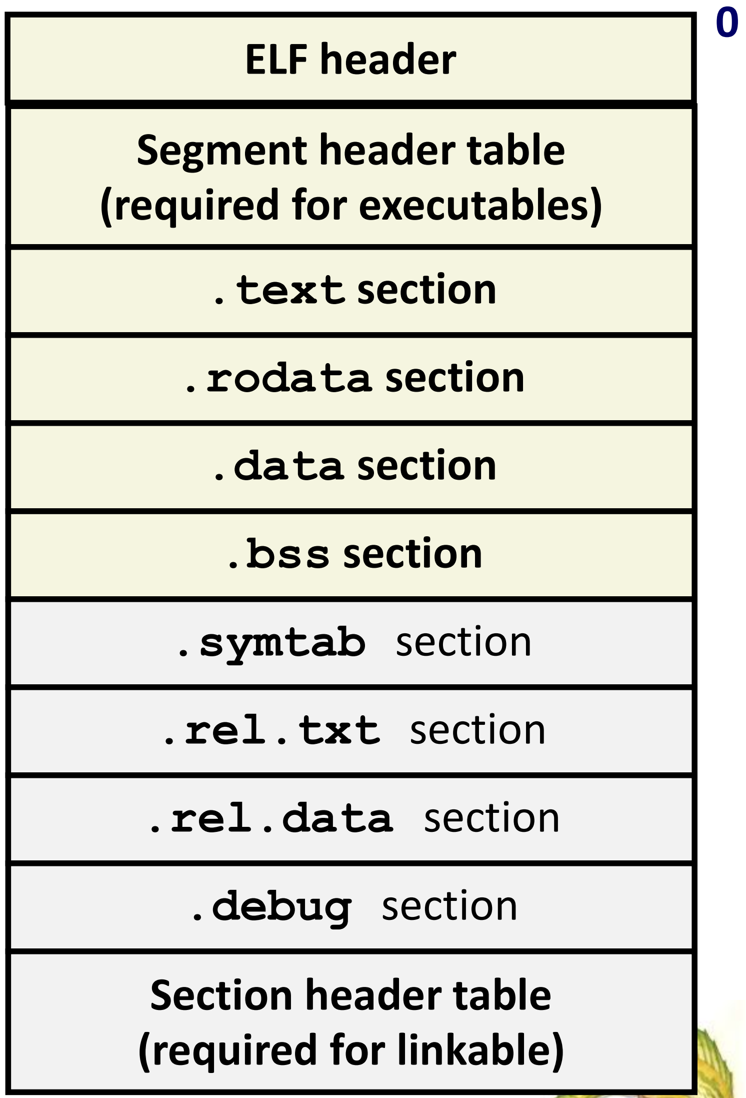
Executable and Linkable Format (ELF)

- Standard binary format for object files
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - **Relocatable** object files (`.o`),
  - **Executable** object files `(a.out or no extension`)
  - **Shared** object files (`.so`)

- Generic name: ELF binaries

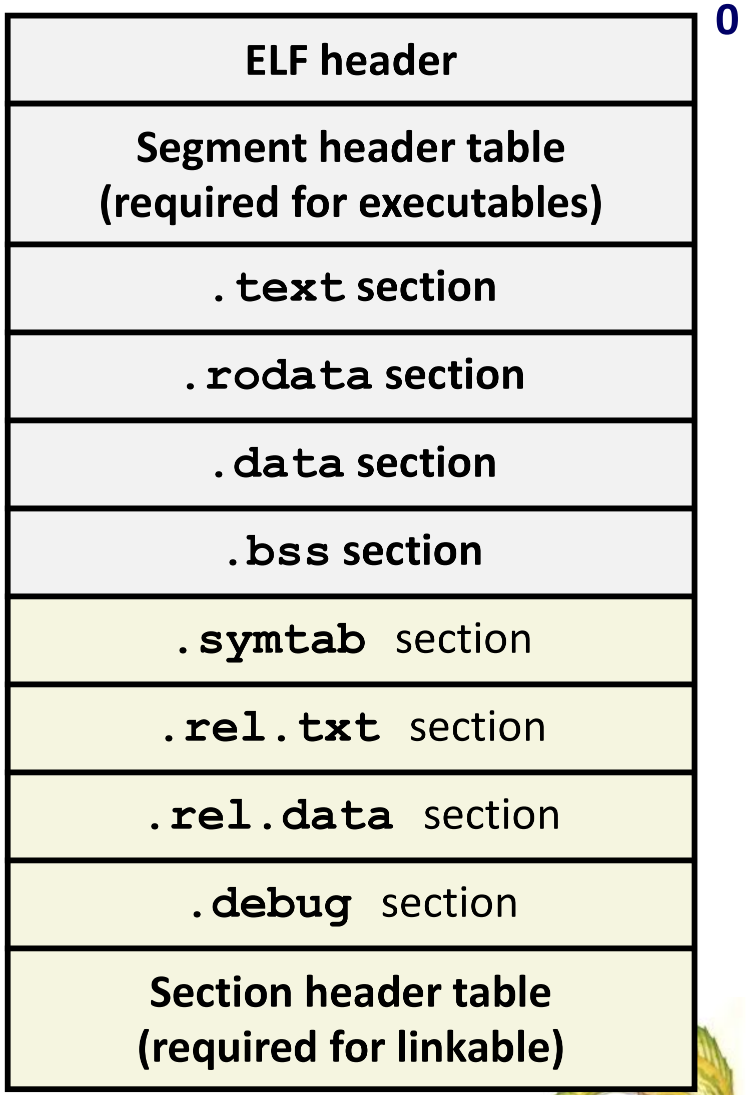
ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: **jump tables**, ..,constant data
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space

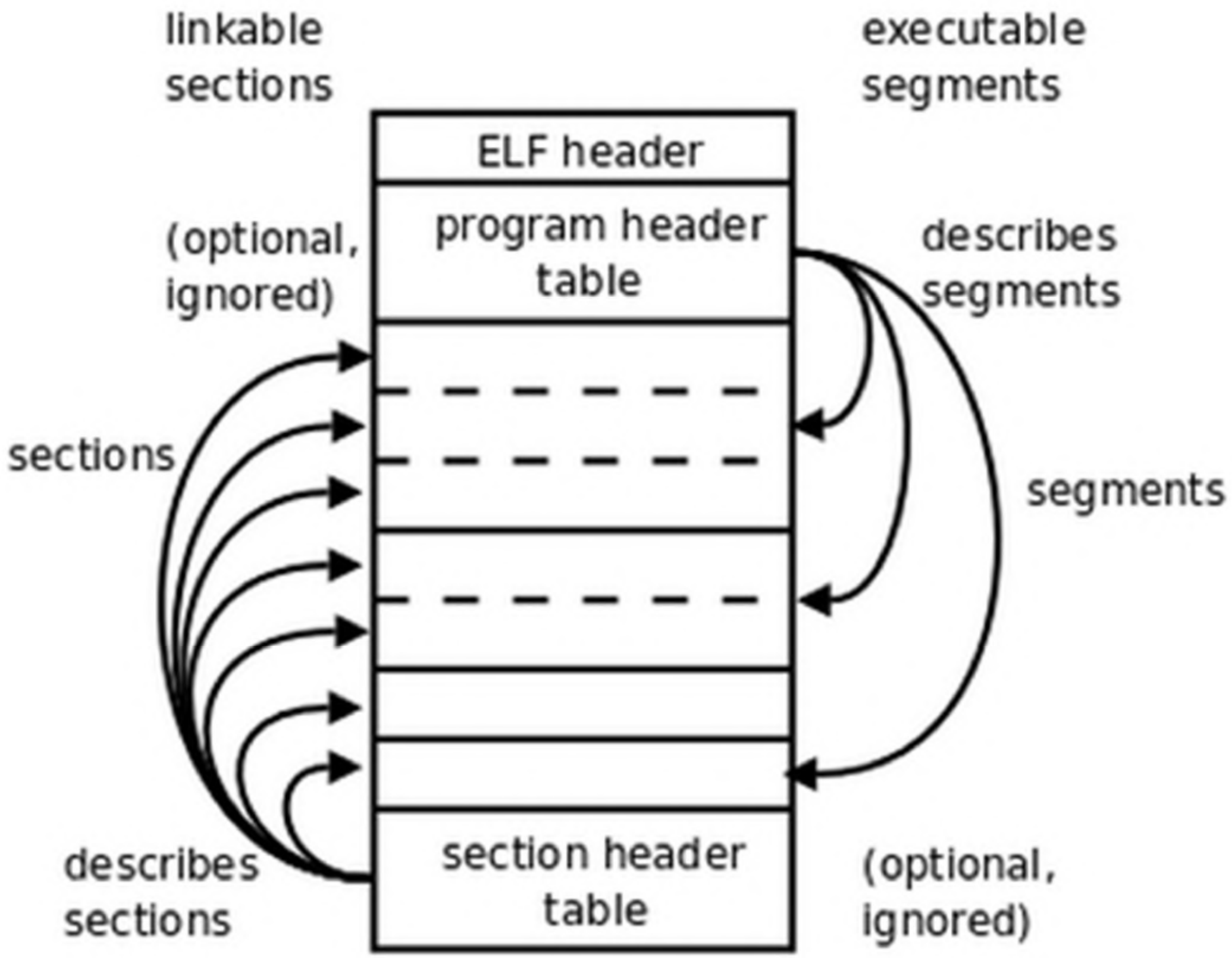


ELF Object File Format (cont.)

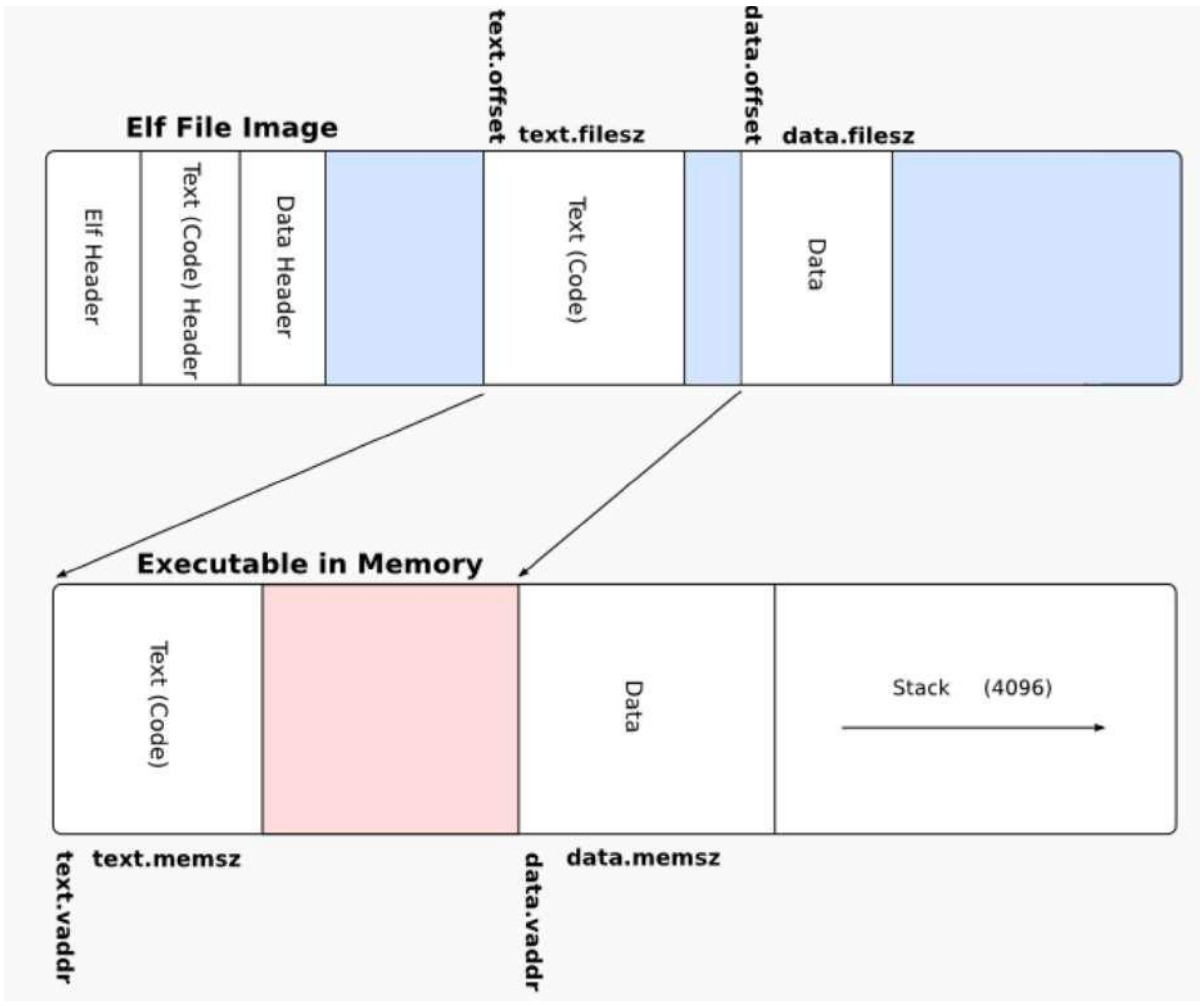
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - **Relocation info** for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- `.rel.data` section
  - **Relocation info** for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (**`gcc -g`**)
- Section header table
  - Offsets and sizes of each section

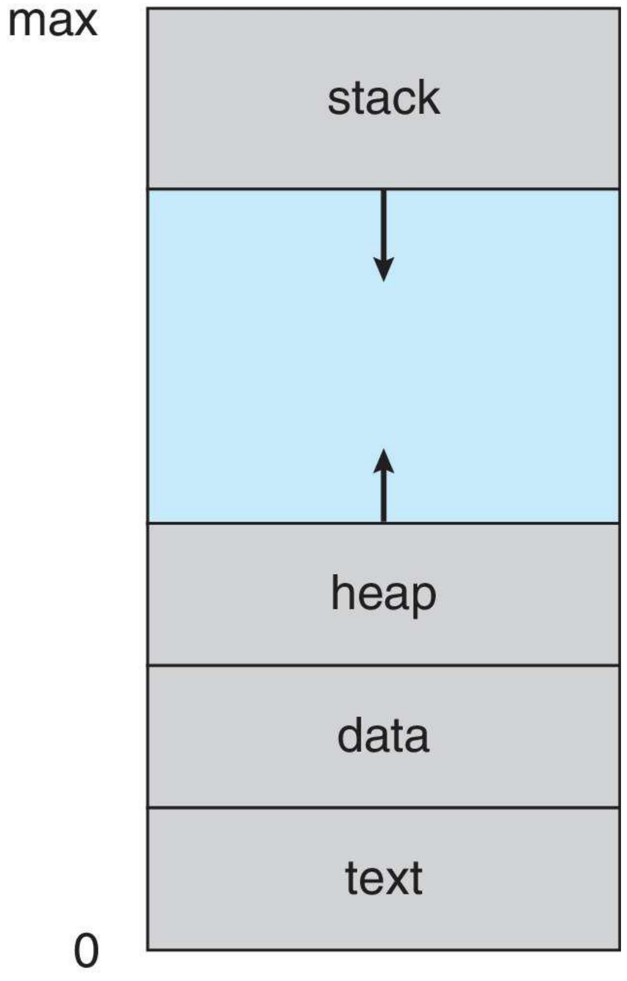


ELF Object File Format (cont.)

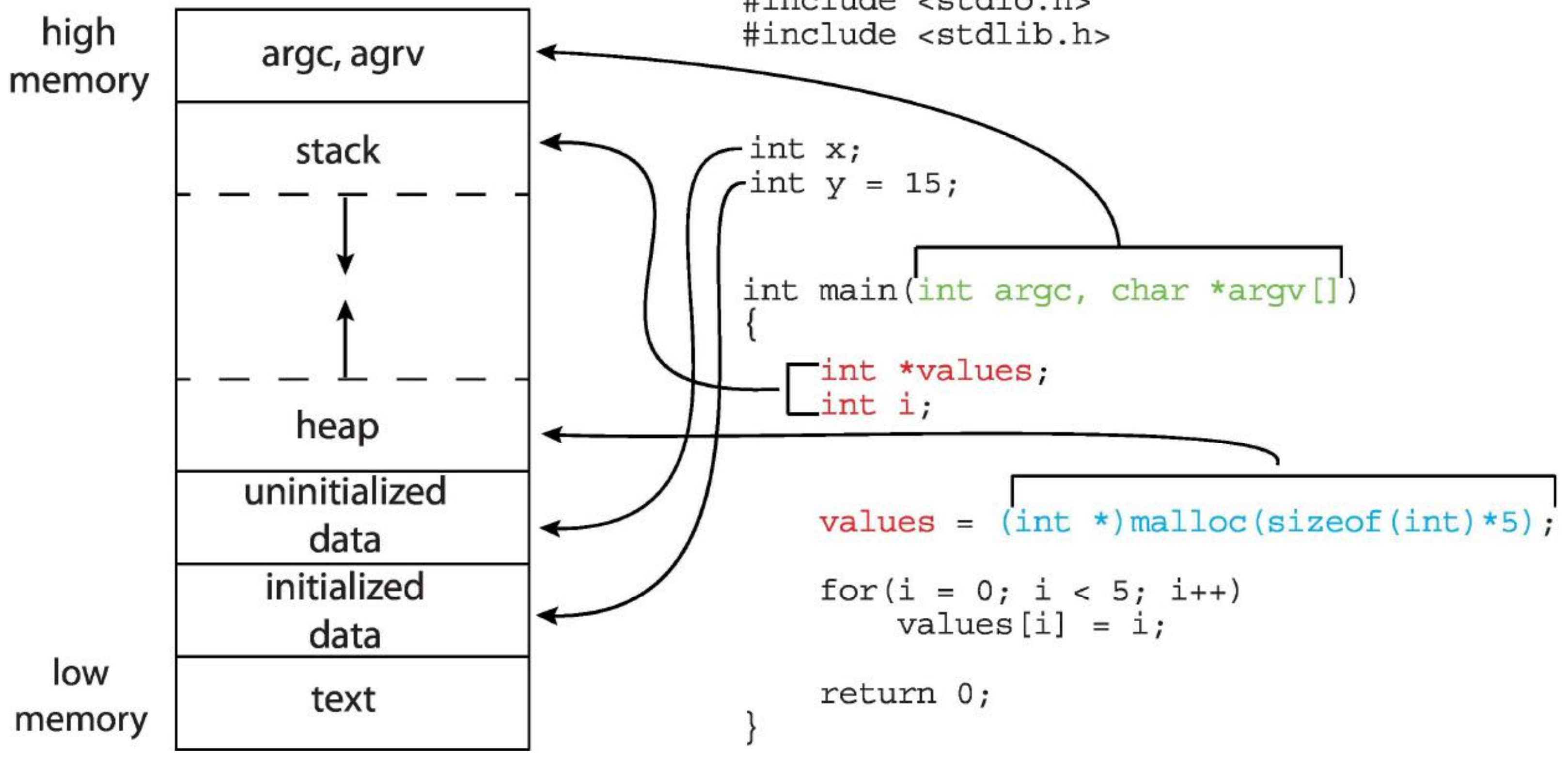


Process in Memory



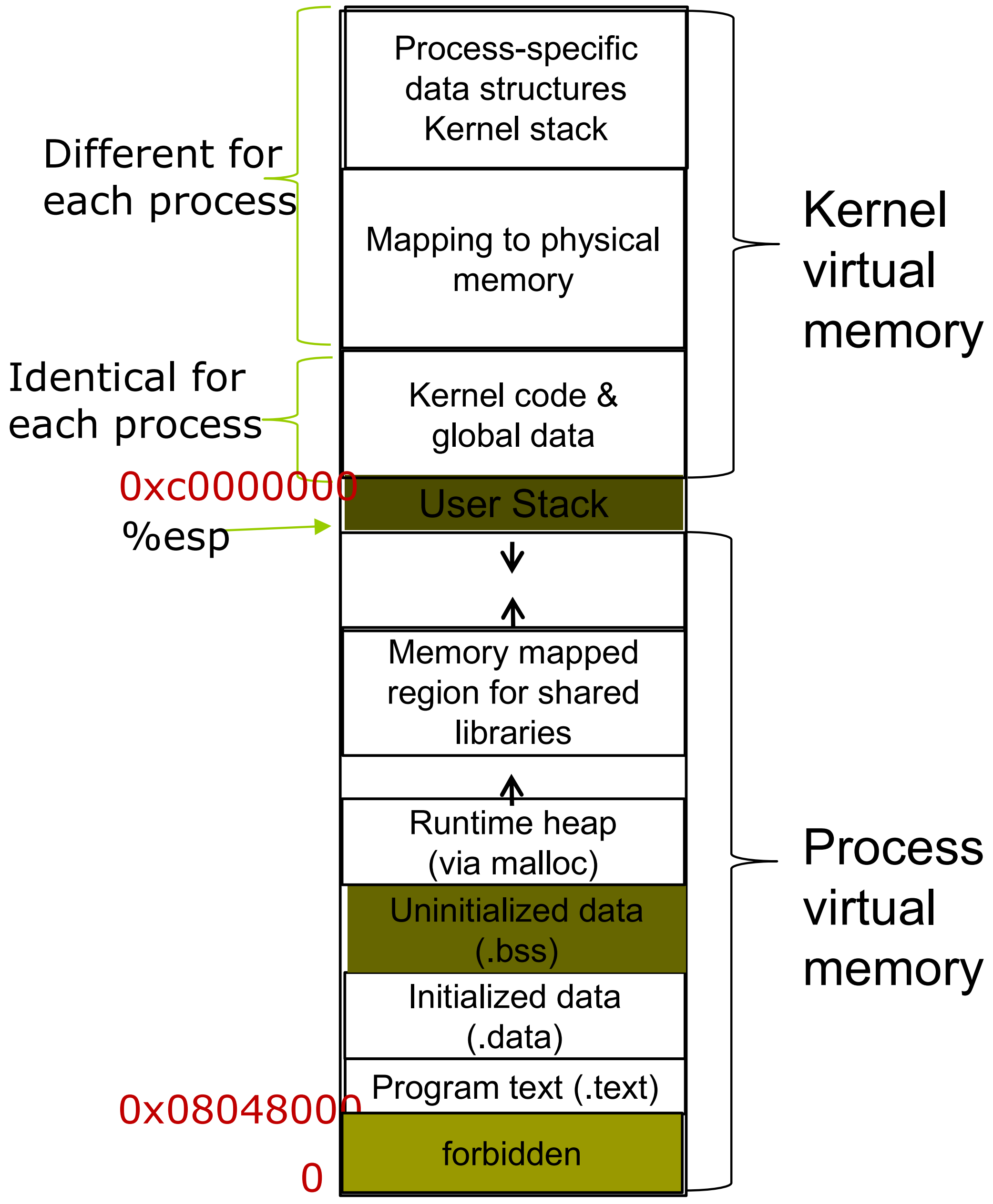


Memory Layout of a C Program



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

      values = (int *)malloc(sizeof(int)*5);

      for(i = 0; i < 5; i++)
          values[i] = i;

      return 0;
}
```

Memory Layout

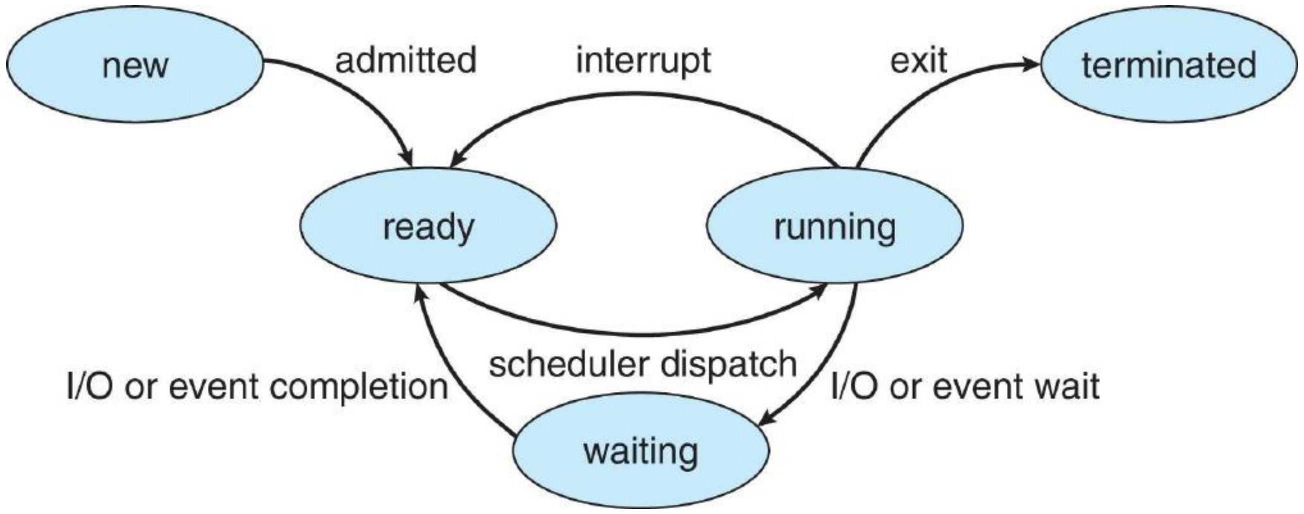


Process-specific data structures Kernel stack
Mapping to physical memory
Kernel code & global data
User Stack
Memory mapped region for shared libraries
Runtime heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
forbidden
Different for each process
Identical for each process
0xc0000000
%esp
0x08048000
0
Kernel virtual memory
Process virtual memory

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed by CPU
  - **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

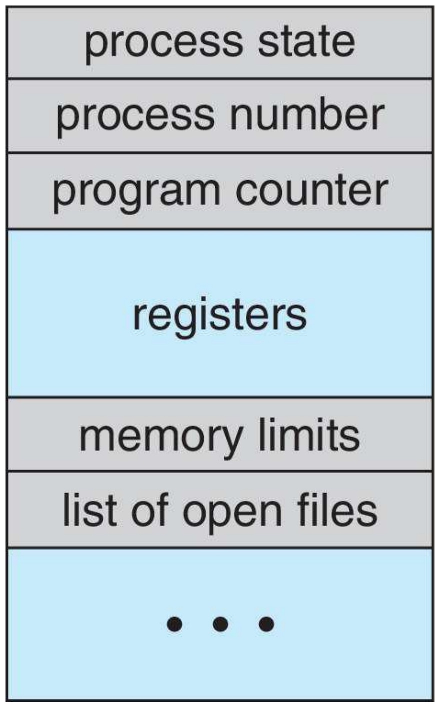


Process Control Block (PCB)

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

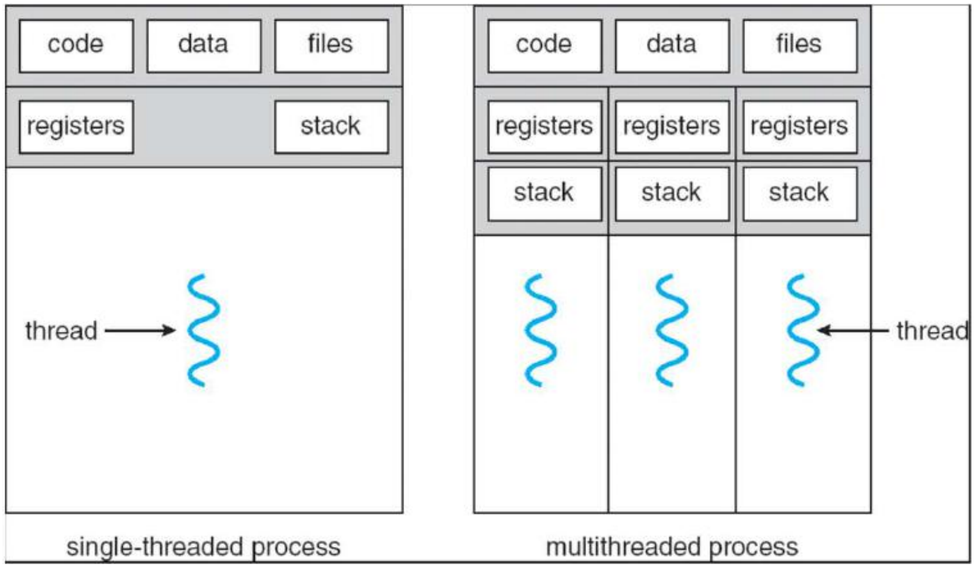


Process Control Block (PCB)

- Each process in memory has a corresponding unique PCB in the kernel.
  - When a new process is created, the kernel creates a new PCB for it.
  - When a process dies, the kernel deletes the process's PCB.
- A PCB is a kernel data structure, not a user process data structure, so:
  - the PCB for a process is stored in the memory reserved for the kernel, not in the memory reserved for the process;
  - the PCB for a process is invisible to the process itself;
  - the PCB information is changed only by the kernel.
- All the PCBs together is how the kernel keeps track of which processes exist in memory, where they are in memory, what they are currently doing (executing, waiting, ...), etc.

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**
- Explored in detail later

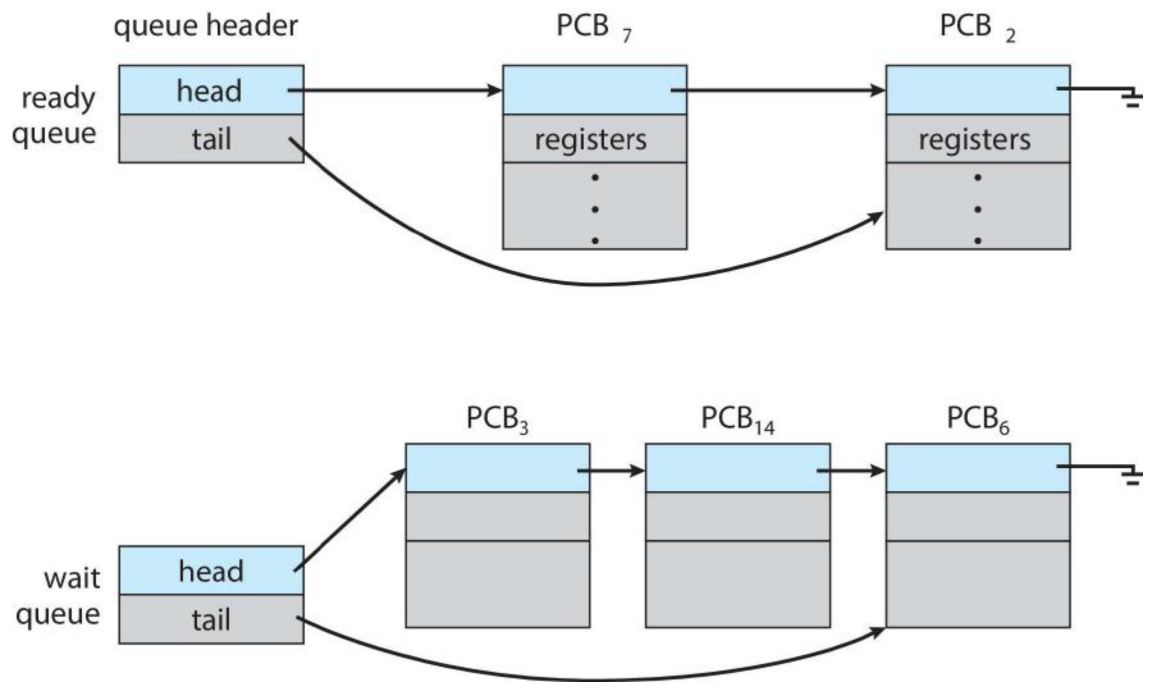


[1] ***A thread is a single sequence stream within in a process***. Because threads have some of the properties of processes, they are sometimes called ***lightweight processes***.

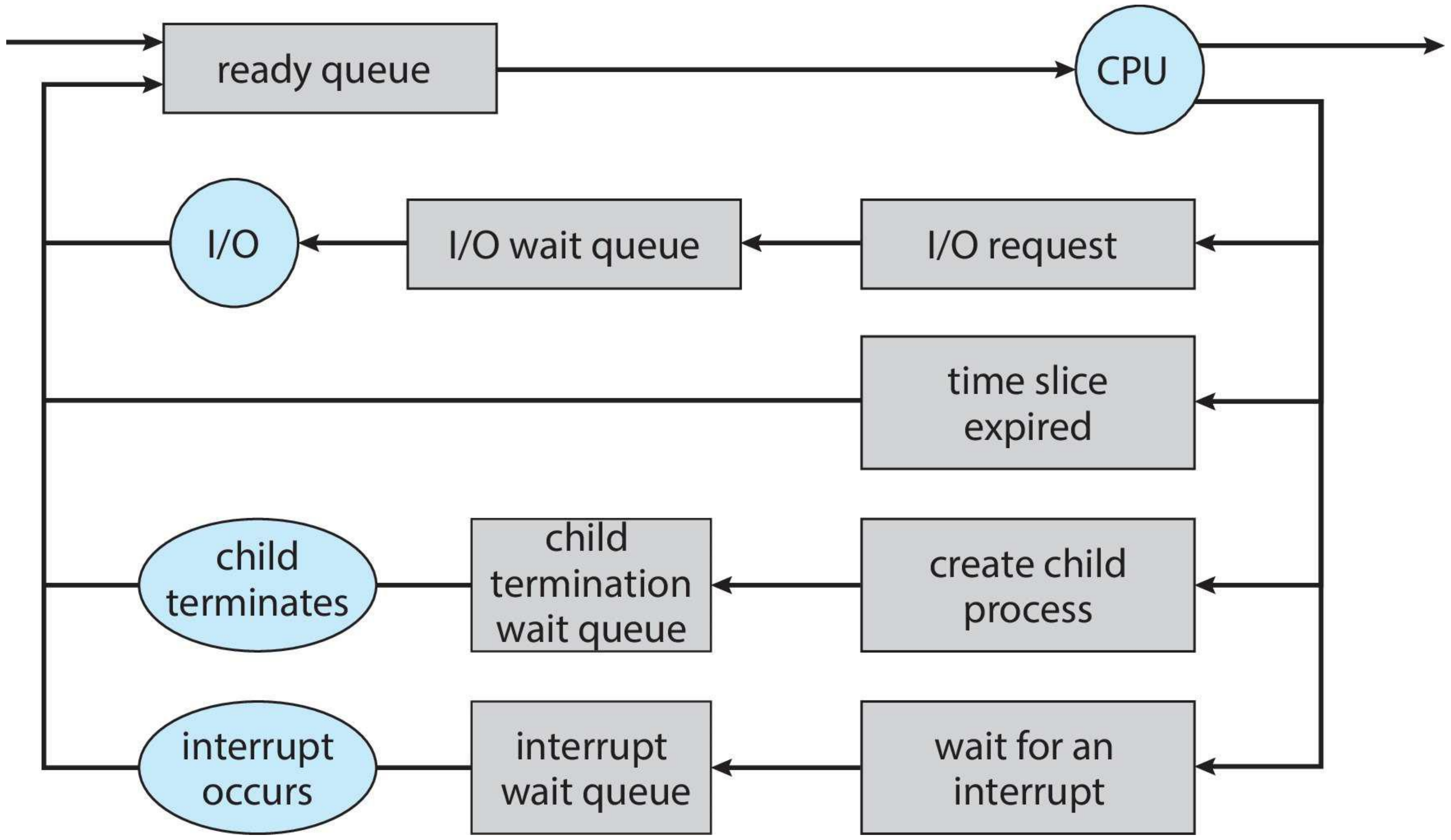
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU core
- **Process scheduler** (algorithm inside the kernel) 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

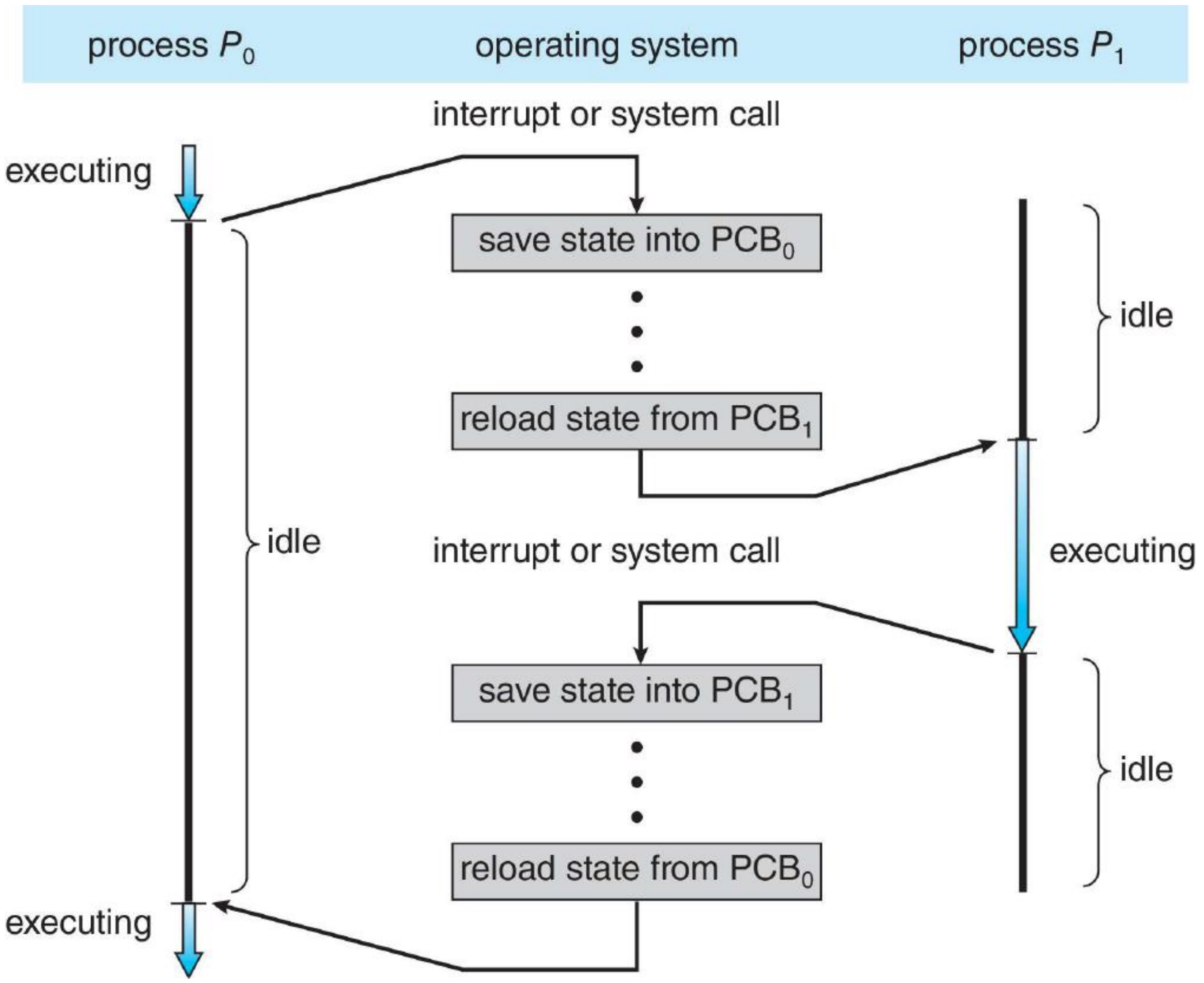


Representation of Process Scheduling



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** (CPU registers, program counter) 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 (iPhone can support multitasking since iOS 4 or iPhone 4)
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface (process with active window focus / input)
  - 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

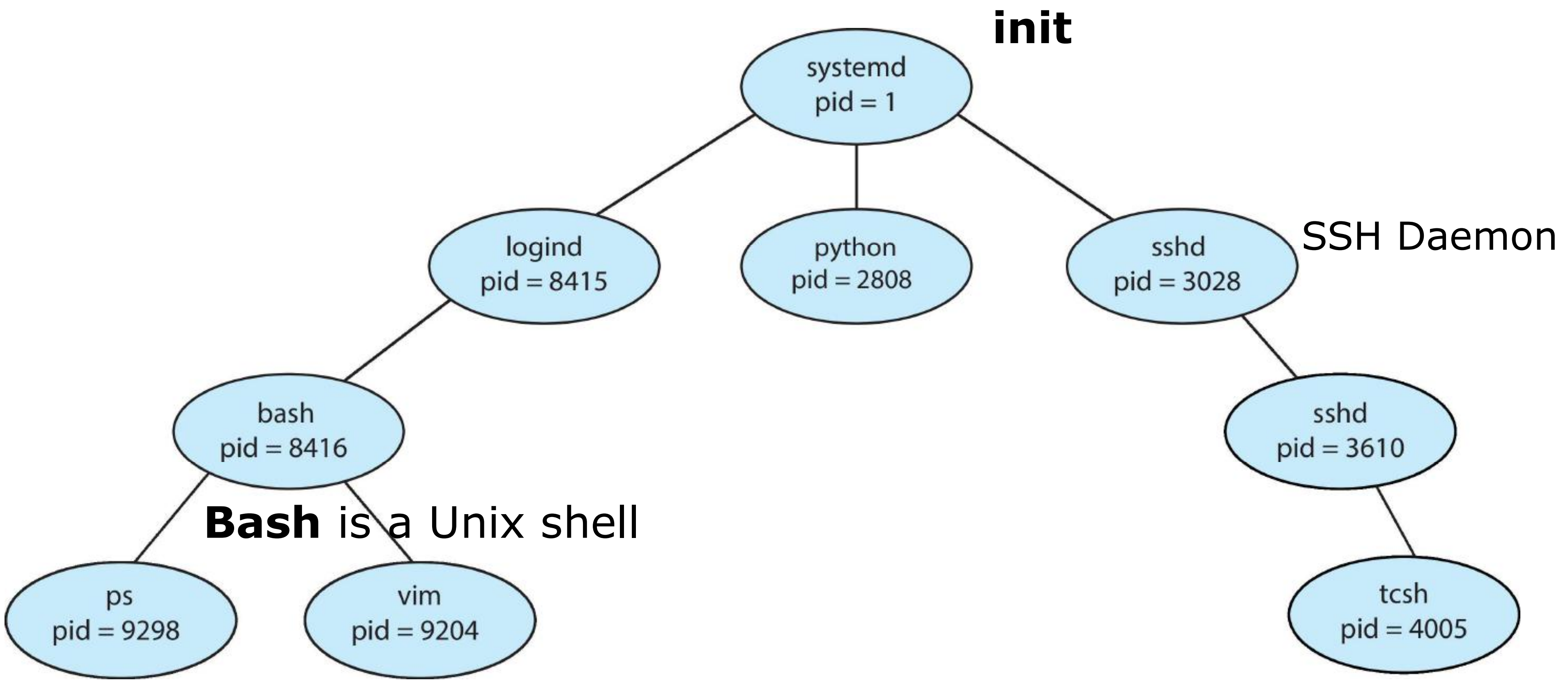
Operations on Processes

- 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
  1. Parent and children **share all** resources
  2. Children **share subset** of parent's resources
  3. Parent and child **share no** resources
- Execution options
  1. Parent and children execute **concurrently**
  2. Parent **waits until** children terminate

A Tree of Processes in Linux



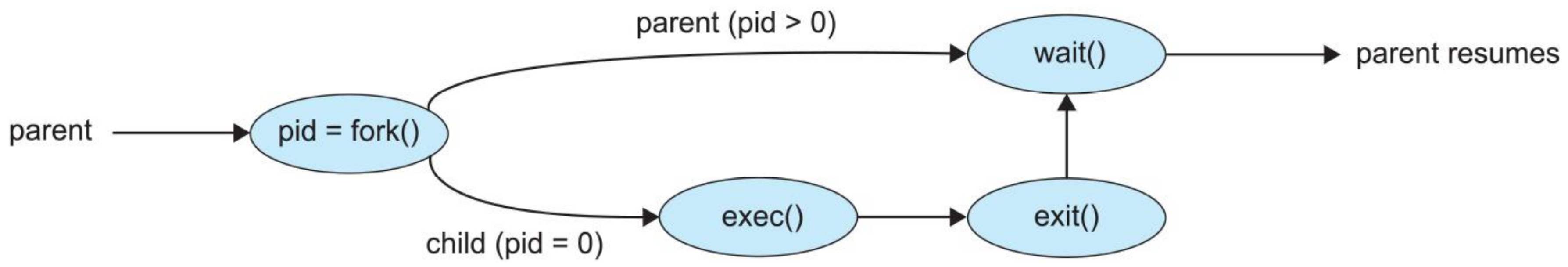
text editor

ps displays information about a selection of the active processes.

tcsh: a Unix shell based on and compatible with the C shell (csh).

Process Creation (Cont.)

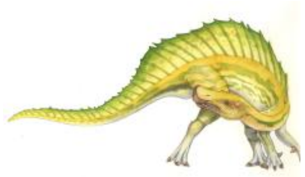
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX/Linux examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** for the child to terminate



Process Creation (Cont.)

Example:

- Process 1 contains the code of Program 1.
- The code of Program 1 inside Process 1 calls the fork system call:
  - the CPU switches to kernel mode and executes the kernel code for the fork system call;
  - that kernel code creates a new Process 2, which is a new child process of Process 1 (the parent);
  - that kernel code also copies the whole content of Process 1’s memory into the memory of Process 2;
    - Process 2 is now an exact copy in memory of Process 1, so Process 2 contains the code of Program 1 too!



Process Creation (Cont.)

- When the fork system call is over, the CPU returns to user mode and continues executing the code of Program 1 in both Process 1 and Process 2 (doing context switches between the two processes).
  - When the CPU executes Program 1 again inside Process 1, the fork system call returns with the PID of Process 2 (the child of Process 1) as result.
  - When the CPU executes Program 1 inside Process 2, the fork system call returns with the PID 0 as result.
  - The result of the fork system call is the only way that Process 1 and Process 2 have to know whether they are the parent process or the child process after the fork!
- Note that the fork system call is called only once by Process 1, but returns twice, **once in Process 1 and once in Process 2**, because the new Process 2 is an exact copy of Process 1.
  - After the fork system call is over, Process 1 and Process 2 are executing the exact same code of Program 1 at the exact same place in that code.
  - Again: the only difference between the two processes is the value returned as result by the fork system call.

Process Creation (Cont.)

- After the fork system call returns, the code of Program 1 tests the result returned by fork:
  - if the current process is the parent process (Process 1) then the code of Program 1 calls the wait system call, which moves Process 1 to the wait queue, to wait for the end of the child process;
  - if the current process is the child process (Process 2) then the code of Program 1 calls the exec system call, which completely replaces the code of Program 1 inside Process 2 (and its corresponding data section and heap and stack; everything) with the code of a new Program 2.
    - The CPU then starts executing Program 2 inside Process 2 from the beginning of the main function of the code of Program 2.
- Note that the exec system call is called once by Process 2, but never returns because the code of Program 1 inside Process 2 (the code that does the exec system call) is replaced with the code of Program 2 when the exec system call is done.
  - The only exception to this is if the exec system call fails (usually because Program 2 does not exist). In that case the exec system call returns and Process 2 continues executing Program 1.

Process Creation (Cont.)

- After some time, Program 2 in Process 2 ends. There are two ways for that to happen.
  1. Program 2 directly calls the exit system call. For example: **`exit(23);`**
  2. The main function of Program 2 returns. For example: **`return 23;`**
     - In that case, the exit system call still happens, but instead of being done by the code of Program 2 itself, it is done by the special loader code inside Process 2 which started the main function!
     - The value 23 returned by the main function as result is then automatically given as argument to the exit system call done by the **special loader code**.
- Either way, directly or indirectly, Process 2 ends up calling the exit system call with the value 23 (as an example) as argument.
  - This value given as argument to the exit system call is called **the “exit status” of Process 2.**

Process Creation (Cont.)

- When Process 2 calls the exit system call:
  1. the kernel destroys Process 2 and deletes its PCB in the kernel's memory;
  2. the kernel wakes up Process 1, which was waiting for its child process to end;
  3. the exit status 23 which was given to the kernel by Process 2 (the child process) as argument to the exit system call is then given by the kernel to Process 1 (the parent process) through the pointer given as argument to the wait system call done by Process 1.
- This allows the parent process to:
  1. **detect when the child process ends**;
  2. **learn why the child process ended**.
     1. An exit status of 0 means the child process terminated normally.
        - This is why in C the main function usually ends with: **`return 0;`**
     2. An exit status different from 0 means the child terminated with an error and the exit code received by the parent process from the child process then gives some information to the parent about why the child died.

Fork ( ) to create a child process

- Fork creates a copy of process (child process)
- Return value from fork (): integer
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Perhaps exceeds resource constraints. sets errno(a global variable in errno.h)
    - Running in original process
- All of the state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors, etc…

C Program Forking Separate Process

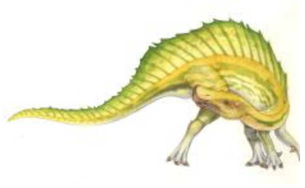
```
int execlp(const char *file, const char* arg, …, NULL);
int execvp(const char *file, char* const argv[]);
```

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid;  // PID of child. pid_t is the same as int
    int status; // Exit status of child.
    printf("Parent: calling fork\n");
    pid = fork(); // New child process starts executing code from here.
    if(pid < 0) { // No child process created.
        printf("Parent: fork failed\n");
        return 1; // Parent process dies.
    }
    else if(pid == 0) {
        // Code only executed by the new child process.
        printf("Child: now running the same program as parent, doing exec\n");
        execlp("xcalc", "xcalc", NULL);
        // If exec succeeded, the new child now starts executing the code of
        // the main function of the xcalc program.  The new child normally
        // never executes the code below, unless exec failed.
        printf("Child: exec failed, die\n");
        return 1; // Child process dies after failed exec.
    } else {
        // Code only executed by the parent process.
        printf("Parent: now sleeping and waiting for child %d to end\n", pid);
        wait(&status);
        printf("Parent: finished waiting for child, child is dead\n");
        printf("Parent: child exit status is %d\n", status);
        return 0; // Parent process dies.
    }
}
```

C Program Forking Separate Process

int execlp(const char *file, const char* arg, …, NULL);
int execvp(const char *file, char* const argv[]);

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid;  // PID of child. pid_t is the same as int
    int status; // Exit status of child.
    char *argv[] = {"xcalc", NULL};
    printf("Parent: calling fork\n");
    pid = fork(); // New child process starts executing code from here.
    if(pid < 0) { // No child process created.
        printf("Parent: fork failed\n");
        return 1; // Parent process dies.
    }
    else if(pid == 0) {
        // Code only executed by the new child process.
        printf("Child: now running the same program as parent, doing exec\n");
        execvp(argv[0], argv);
        // If exec succeeded, the new child now starts executing the code of
        // the main function of the xcalc program.  The new child normally
        // never executes the code below, unless exec failed.
        printf("Child: exec failed, die\n");
        return 1; // Child process dies after failed exec.
    } else {
        // Code only executed by the parent process.
        printf("Parent: now sleeping and waiting for child %d to end\n", pid);
        wait(&status);
        printf("Parent: finished waiting for child, child is dead\n");
        printf("Parent: child exit status is %d\n", status);
        return 0; // Parent process dies.
    }
}
```



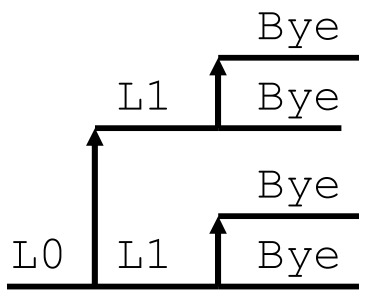
Creating a Separate Process via Windows API

```c
#include <windows.h>
#include <stdio.h>
int main( VOID ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD status;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line).
        "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi )             // Pointer to PROCESS_INFORMATION structure.
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return -1;
    }
    // Wait until child process exits.
    status = WaitForSingleObject( pi.hProcess, INFINITE );
    printf("Child exit status is %d\n", status);
    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
    return 0;
}
```

Fork Examples 1/2

- Both parent and child can continue forking

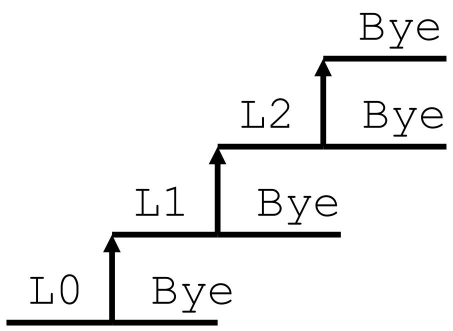
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()  {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
    return 0;
}
```



Fork Examples 2/2

- Both parent and child can continue forking

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (){
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
    return 0;
}
```



Fork Bomb

- Unix: `while(1) fork();`
  - The parent process creates an infinite number of child processes.
  - Each child process executes the same program and creates an infinite number of grandchild processes.
  - Each grandchild process executes the same program and creates an infinite number of grandgrandchild processes.
  - Etc.
  - Result: **an exponential number** of processes are created, the kernel is using all the CPU both creating processes and trying to do context switches between them, the memory becomes full, the whole computer system **slows down** and becomes **unusable**.
    - Therefore modern Unix systems limit the number of processes that a user is allowed to create.

wait() and waitpid()

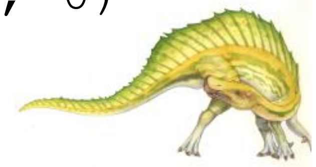
wait() – Syschronizing with the children

- **int** wait(int *child_status)
  - **suspends current process until one of its children terminates**
  - return value is the **pid** of the child process that terminated
  - if **child_status != NULL**, then the object it points to will be set to a status indicating why the child process terminated.

waitpid() – Waiting for specific child process

- waitpid(pid, &status, **options**)
  - **suspends current process until specific process terminates**
  - various options

```
int child_status;
wait(&child_status) => wait_pid(-1, &child_status, 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

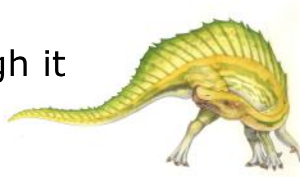
Process Termination

- **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()`**), child process is a zombie[1]
- If parent terminated without invoking **`wait()`** , child process is an orphan[2]

[1]A **zombie process** is a process that has completed execution (via the exit system call) but still has an entry in **the process table**: it is a process in the "Terminated state". This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "**reaped**".
[2]An **orphan process** is a process whose parent process has finished or terminated, though it remains running itself.



Zombies

- Idea
  - When process terminates, it still consumes system resources
    - Various tables maintained by OS
  - Called a “**zombie**”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information (by OS)
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by **init** or **systemd** process
  - So, **only need explicit reaping in long-running processes**
    - e.g., **shells and servers**

Zombie and Orphan Examples

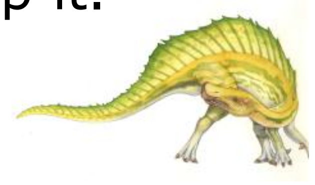
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
    pid_t pid = fork();
    if (pid == 0) {
      /* Child */
      printf("Terminating Child, PID = %d\n",
                 getpid());
      exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
                 getpid());
        while (1)
          ; /* Infinite loop */
    }
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
 pid_t pid = fork();
 if (pid == 0) {
    /* Child */
    printf("Running Child, PID = %d\n",
                 getpid());
    while (1)
      ; /* Infinite loop */
 } else {
    printf("Terminating Parent, PID = %d\n",
                 getpid());
    exit(0);
 }
 return 0;
}
```

- `ps` shows child process as “defunct” (parent didn’t call wait())
- Killing parent allows child to be reaped by *`init/systemd`*

Child process still active even though parent has terminated

init/systemd process will reap it if no other parent reap it.

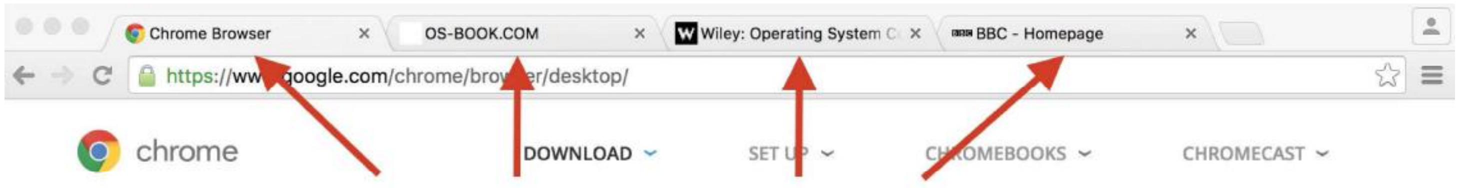


Android Process Importance Hierarchy

- 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 (and MS Edge) 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.

End of Chapter 4