DCS216 Operating Systems

Lecture 05
Processes (1)

Mar 11th, 2024

Instructor: Xiaoxi Zhang
Sun Yat-sen University

Content

- **Process Concept**
  - What is a Process?
    - (Informally) A process is a program in **execution**, or a **running** program.
    - Program is a **passive** entity, a lifeless thing: it just sits there on the disk.
    - Process is an **active** entity: it consumes CPU and Memory.
    - Program becomes process when it is **loaded** into memory

Process Concept

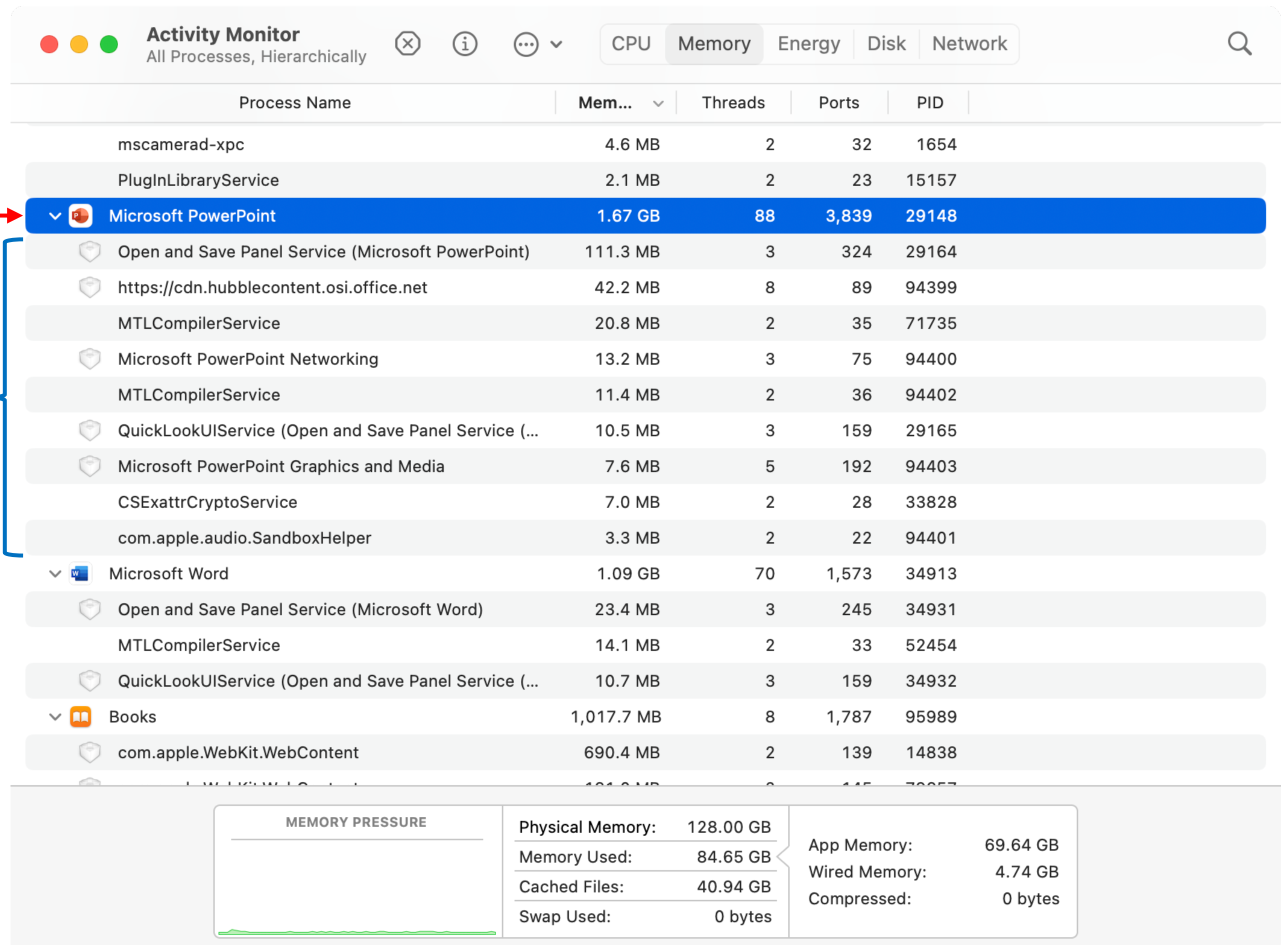
- What is a Process?
  - (Informally) A process is a program in **execution**, or a **running** program.
  - Program is a **passive** entity, a lifeless thing: it just sits there on the disk.
  - Process is an **active** entity: it consumes CPU and Memory.
  - Program becomes process when it is **loaded** into memory
- Definition: **execution environment with restricted rights**
  - One or more **threads** executing in a single **address space**
  - Owns file descriptors, network connections
- Instance of a running program
  - When you run an executable file, it runs in its own process
  - **Application**: one or more **processes** working together
- Processes are protected from each other
- OS protected from processes
- In modern OSes, anything outside of kernel runs in a process

Process Concept

- Instance of a running program
  - When you run an executable file, it runs in its own process
  - **Application**: one or more **processes** working together

Application PowerPoint

Processes that constitute PowerPoint



Activity Monitor — All Processes, Hierarchically — CPU | Memory | Energy | Disk | Network

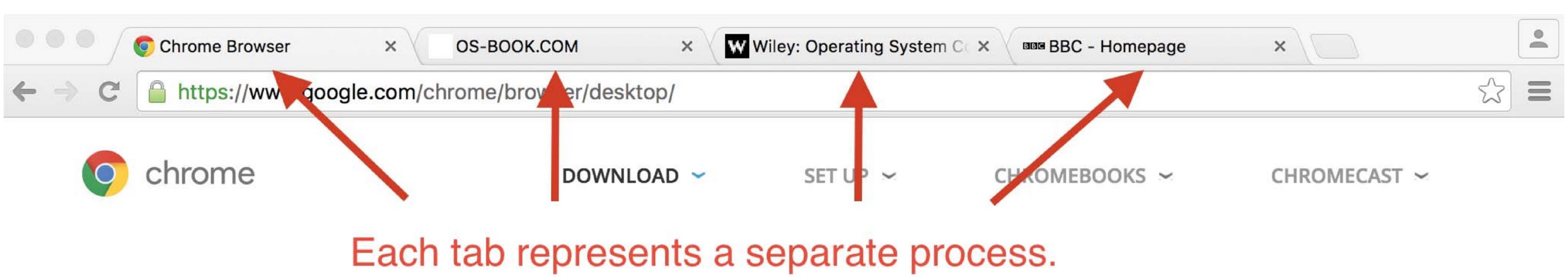
Process Name	Mem...	Threads	Ports	PID
mscamerad-xpc	4.6 MB	2	32	1654
PlugInLibraryService	2.1 MB	2	23	15157
Microsoft PowerPoint	1.67 GB	88	3,839	29148
Open and Save Panel Service (Microsoft PowerPoint)	111.3 MB	3	324	29164
https://cdn.hubblecontent.osi.office.net	42.2 MB	8	89	94399
MTLCompilerService	20.8 MB	2	35	71735
Microsoft PowerPoint Networking	13.2 MB	3	75	94400
MTLCompilerService	11.4 MB	2	36	94402
QuickLookUIService (Open and Save Panel Service (...	10.5 MB	3	159	29165
Microsoft PowerPoint Graphics and Media	7.6 MB	5	192	94403
CSExattrCryptoService	7.0 MB	2	28	33828
com.apple.audio.SandboxHelper	3.3 MB	2	22	94401
Microsoft Word	1.09 GB	70	1,573	34913
Open and Save Panel Service (Microsoft Word)	23.4 MB	3	245	34931
MTLCompilerService	14.1 MB	2	33	52454
QuickLookUIService (Open and Save Panel Service (...	10.7 MB	3	159	34932
Books	1,017.7 MB	8	1,787	95989
com.apple.WebKit.WebContent	690.4 MB	2	139	14838

MEMORY PRESSURE

Physical Memory:	128.00 GB
Memory Used:	84.65 GB
Cached Files:	40.94 GB
Swap Used:	0 bytes
App Memory:	69.64 GB
Wired Memory:	4.74 GB
Compressed:	0 bytes

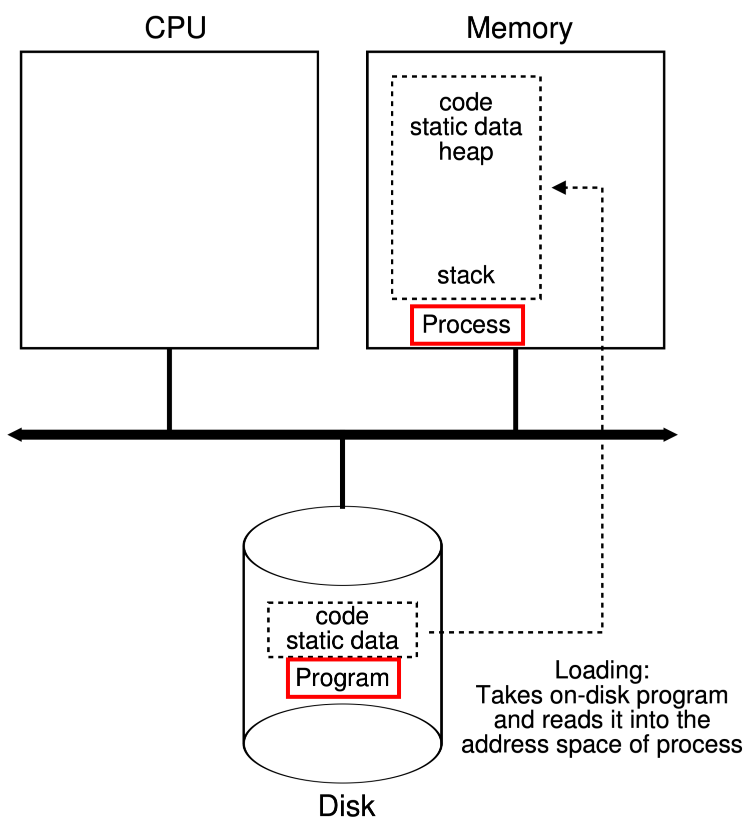
Process Concept

- **Multiprocess Application Example – Google Chrome Browser**
  - Many web browsers ran as single process
    - 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 (渲染) webpages, deals with HTML, JavaScript. A new renderer created for each website opened
      - Runs in sandbox (沙盒) restricting disk and network I/O, minimize security risks
    - **Plug-in** process – for each type of plug-in (插件)



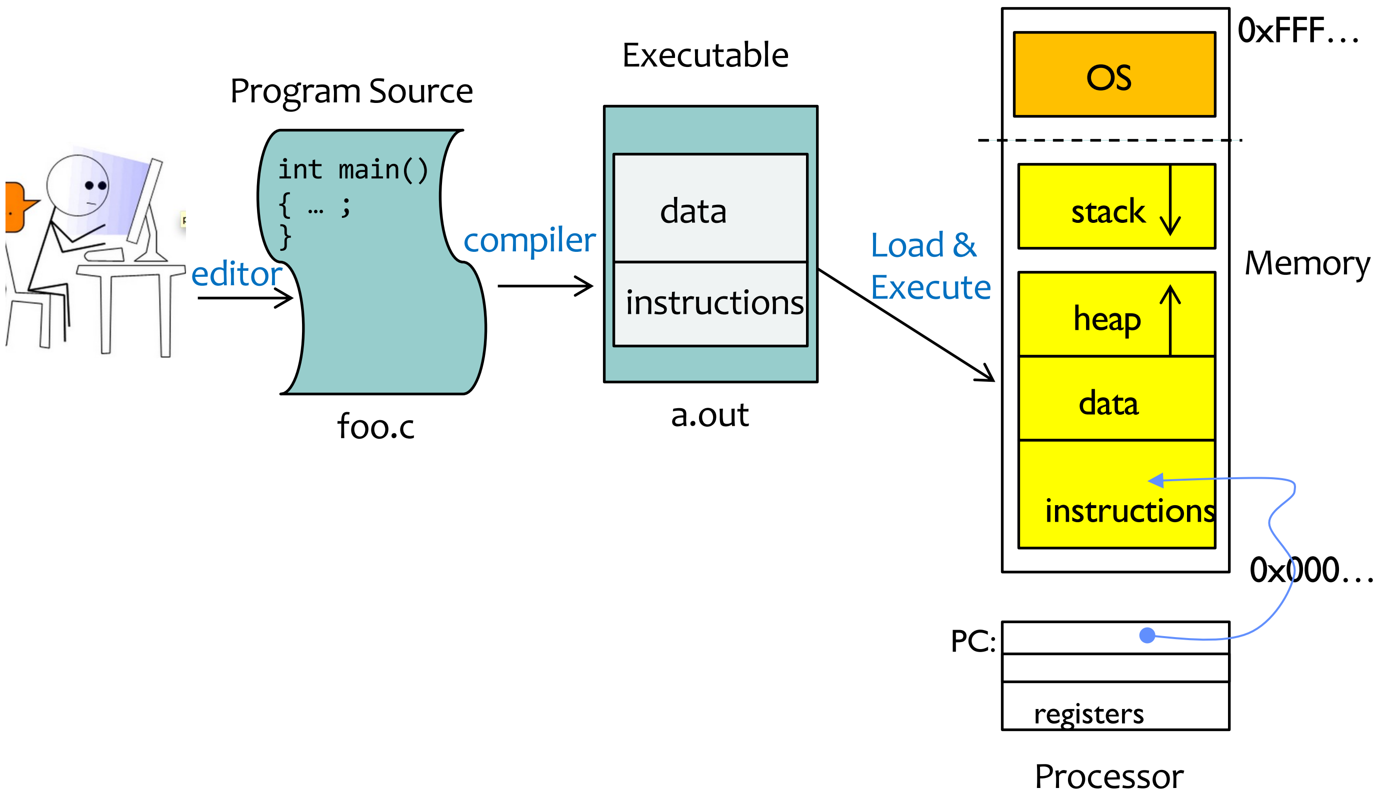
Process Concept

- What is a Process?
  - (Informally) A process is a program in **execution**, or a **running** program.
  - Program is a **passive** entity, a lifeless thing: it just sits there on the disk.
  - Process is an **active** entity: it consumes CPU and Memory.
  - Program becomes process when it is **loaded** into memory



Process Concept

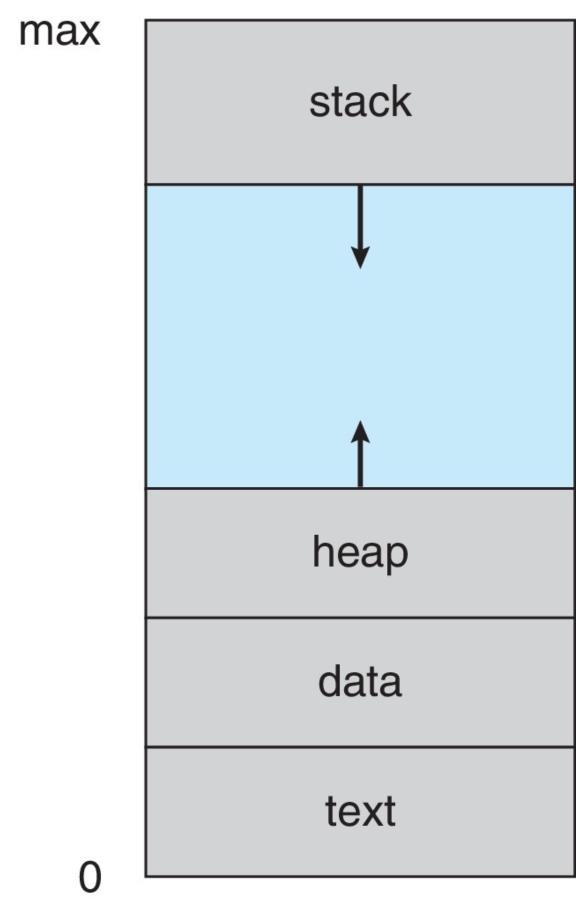
- **Running of a program**
  - Load instruction and data segments of executable file into memory
  - Create stack and heap
  - “Transfer control to program”
  - Provide services to program
  - While protecting OS and program



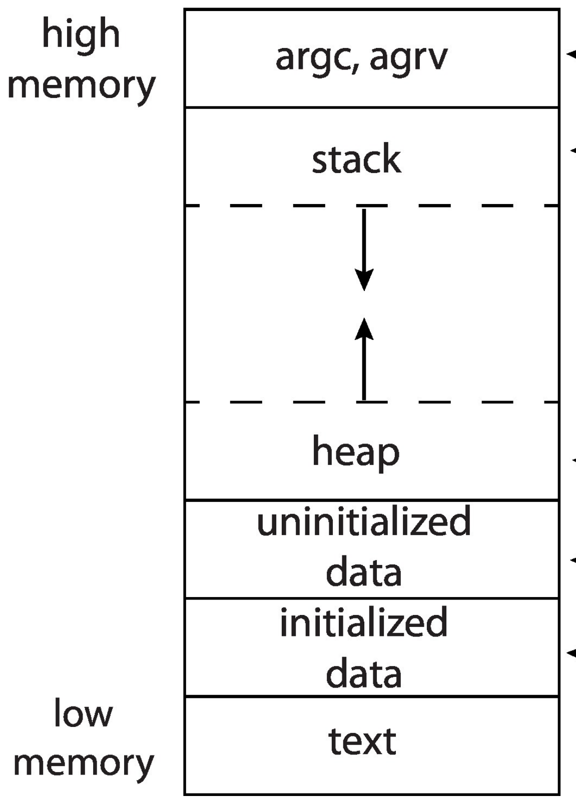
Process Concept

Process Consists of

- **Text** Section: the program code
- **Data** Section: global variables
- **Stack** Section: temporary data storage when invoking functions, e.g., *function parameters*, *return addresses*, *local variables*
- **Heap** Section: dynamically allocated memory during runtime via `malloc()`.
- Current Activity:
  - **Program Counter** (Instruction Pointer)
  - **Processor Registers**

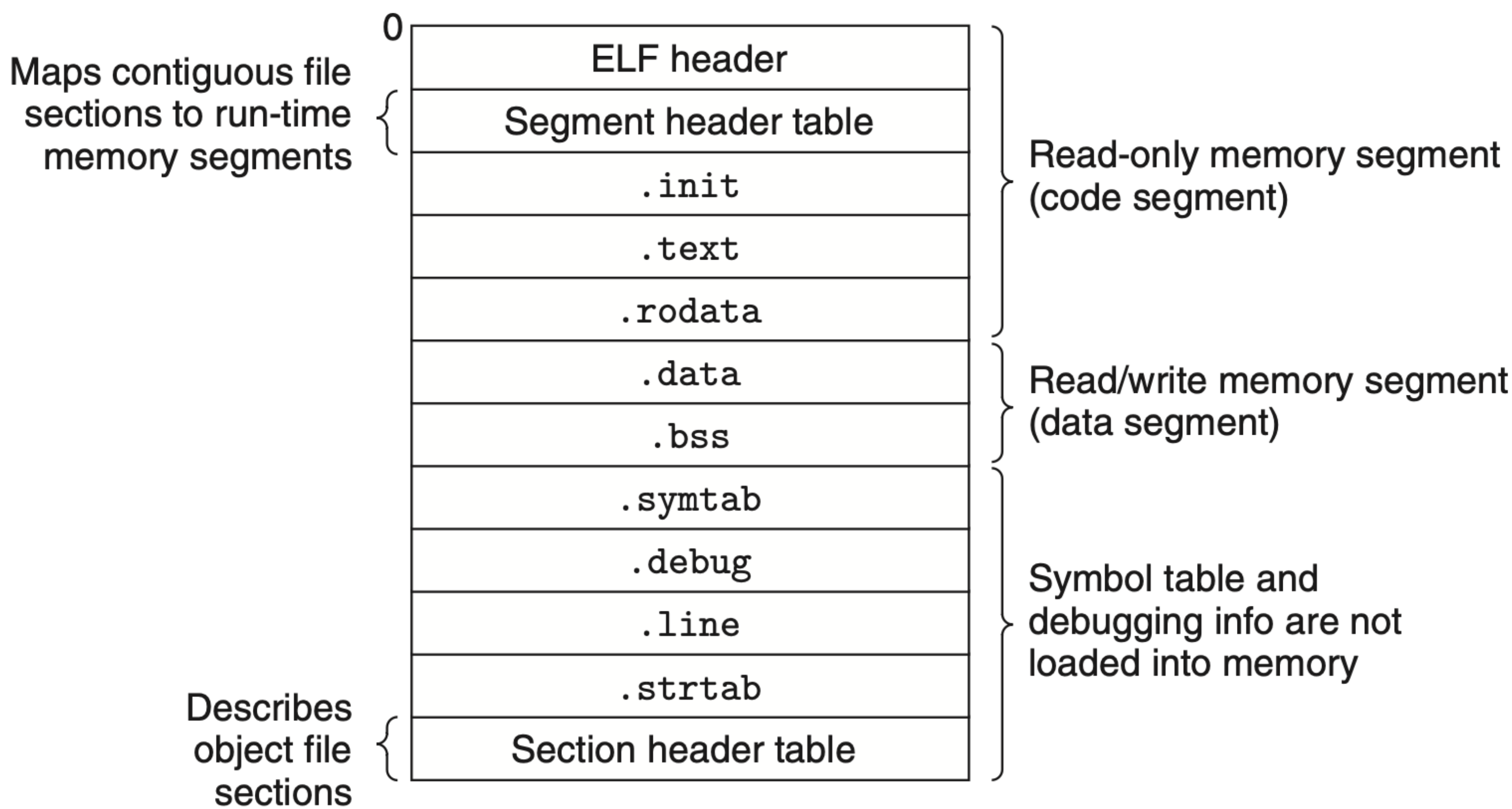


Process Memory Layout



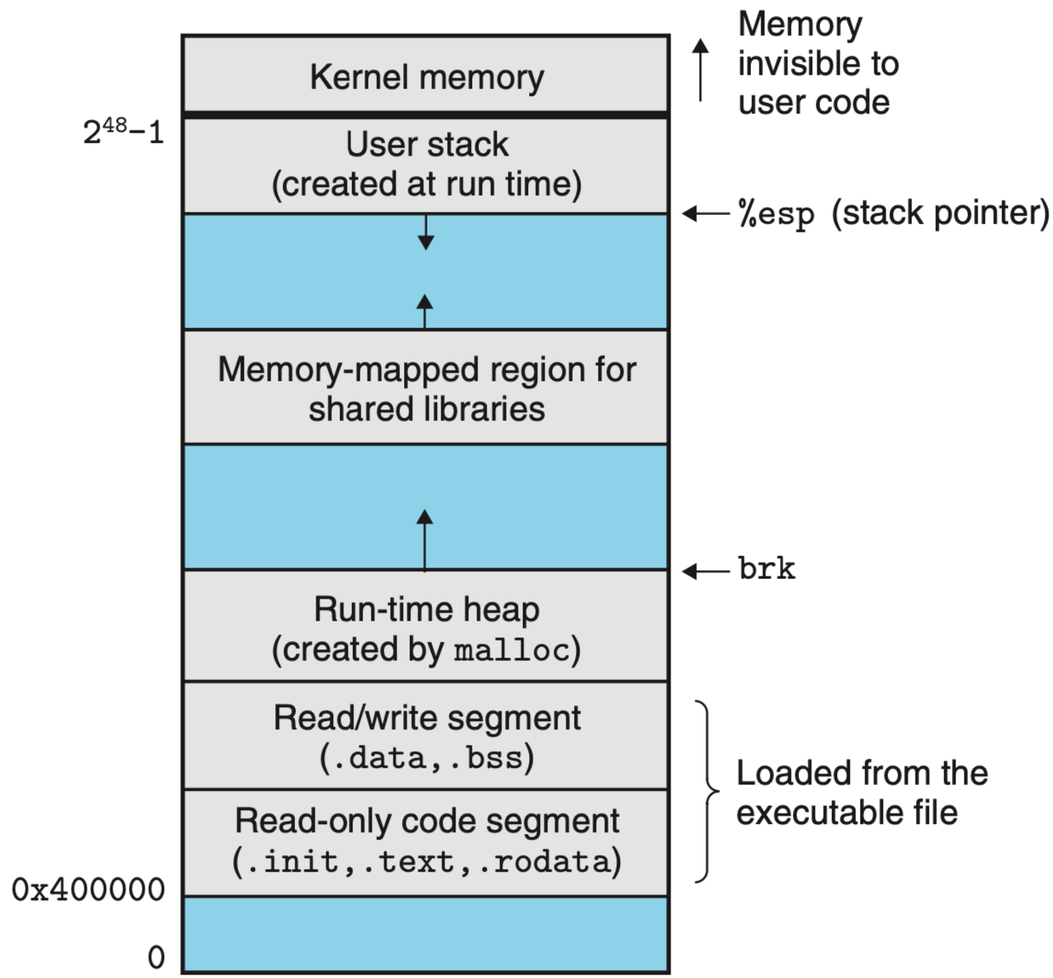

```c
#include <stdio.h>
#include <stdlib.h>

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;
}
```

- **Typical ELF Executable Object File on Linux**
  - Exercise: Use `objdump` to inspect various sections of an executable file (a program)



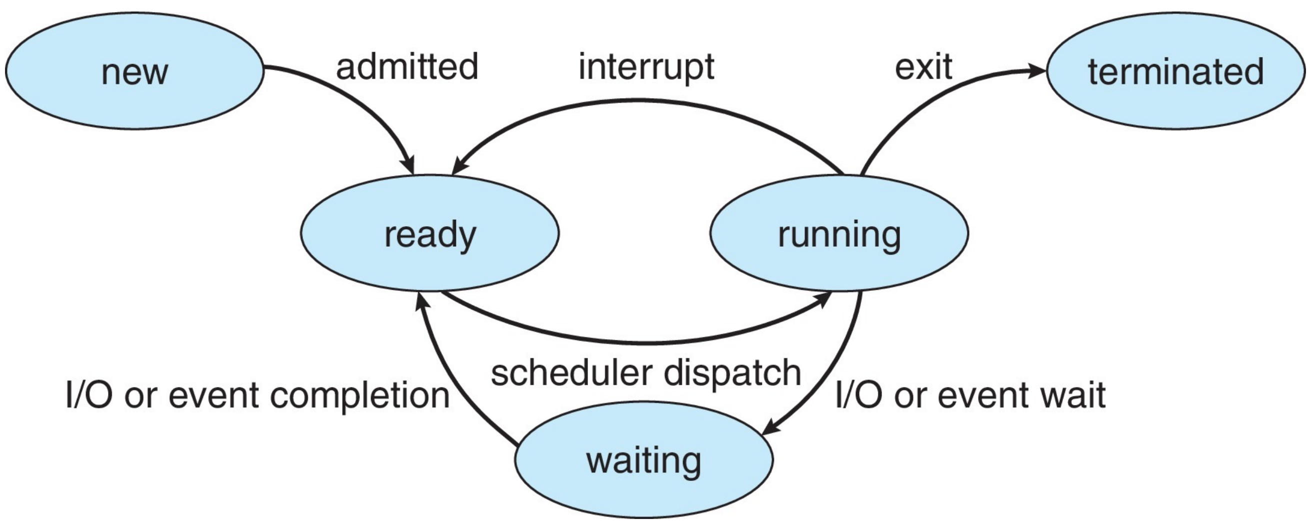
Process Memory Layout on Linux

- Exercise: Use gdb to inspect memory layout of a process



Process States

- As a process executes, it changes state
  - **New:** The process is being created
  - **Running:** Instructions are being executed
  - **Waiting (Blocked):** 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

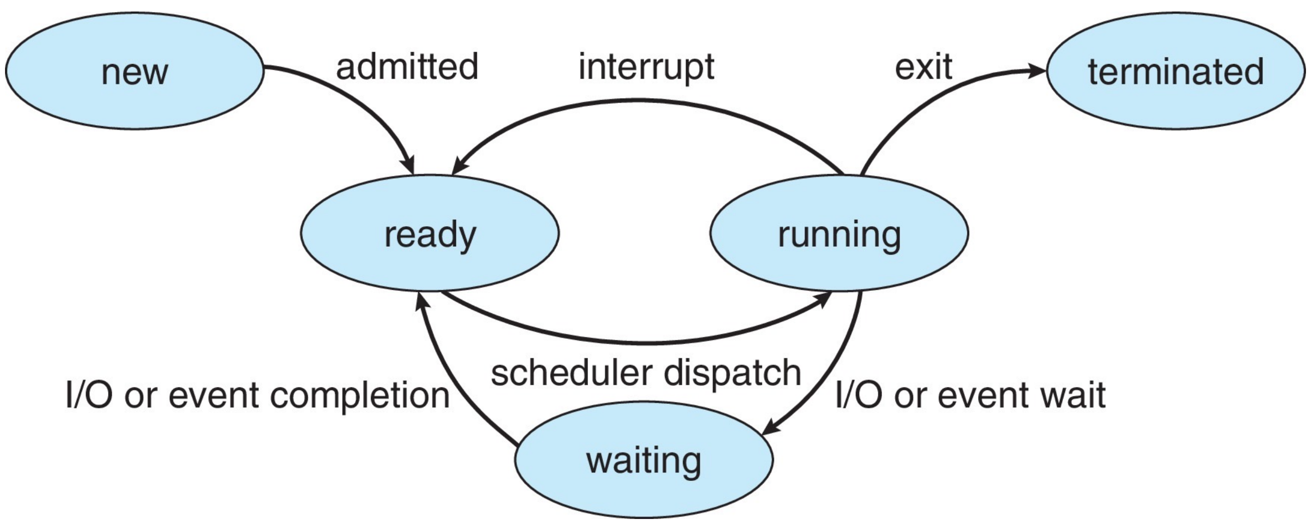


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
```

Using gcc to generate executable file ps_state. The **program** ps_state now sits somewhere on the disk. There is no process of the **program** ps_state running at the moment.

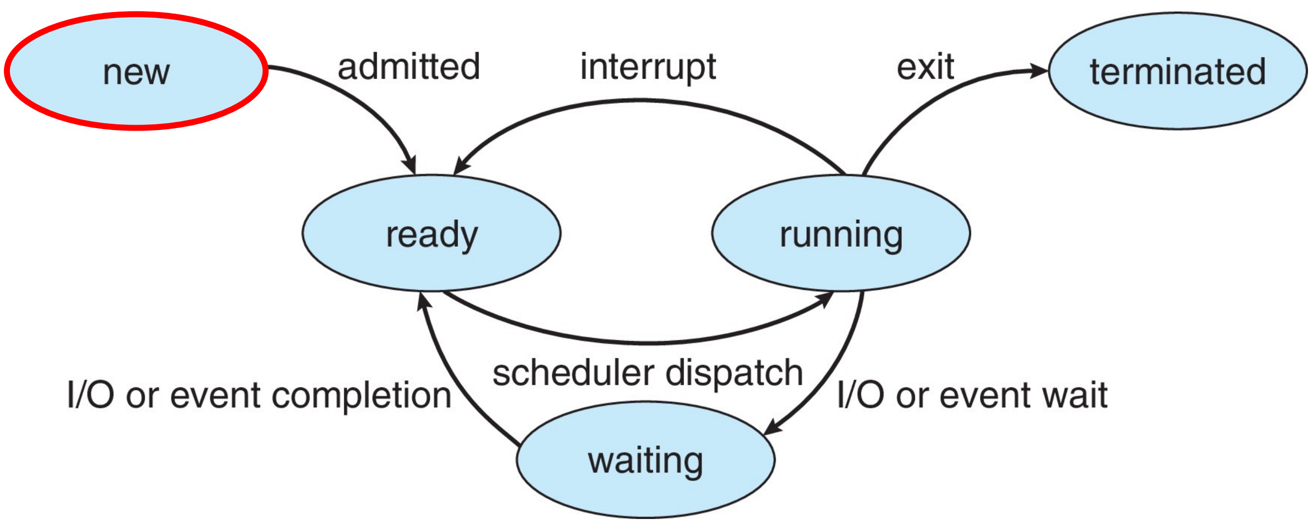


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
```

Execute the program ps_state.
The process is admitted and being created by the OS.

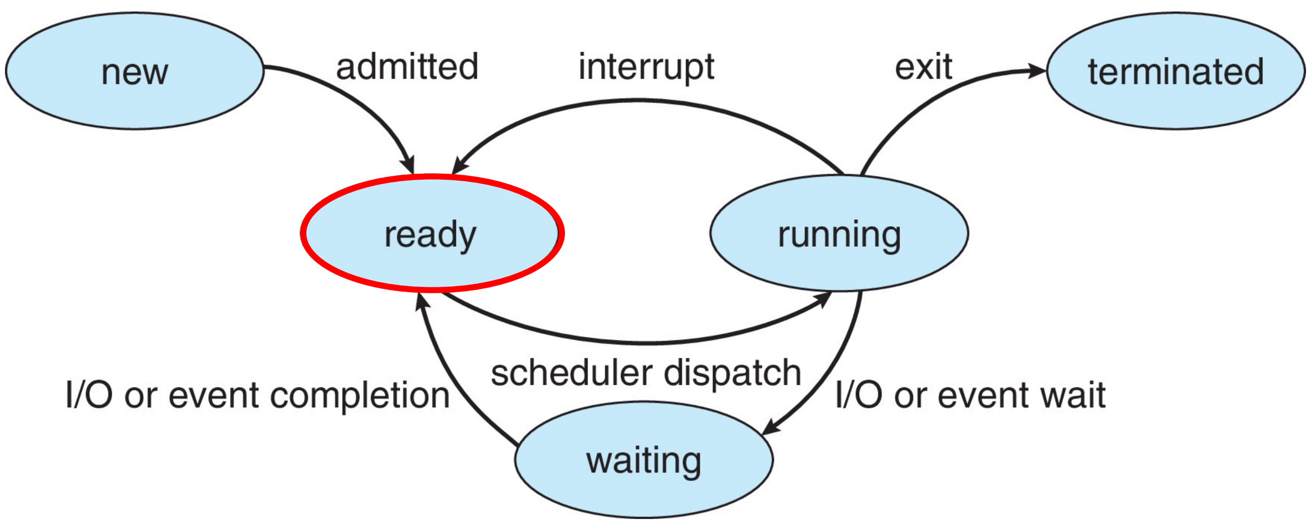


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
```

The process has been created and ready to run.



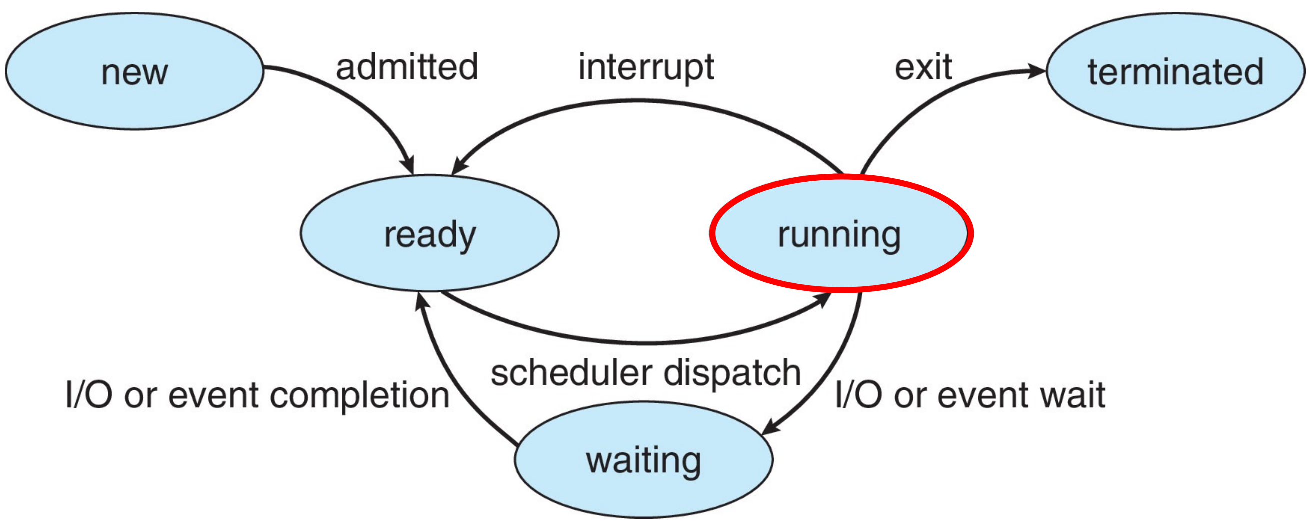
Process Concept

Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
```

The scheduler has chosen to run this process on one of the CPU(s).



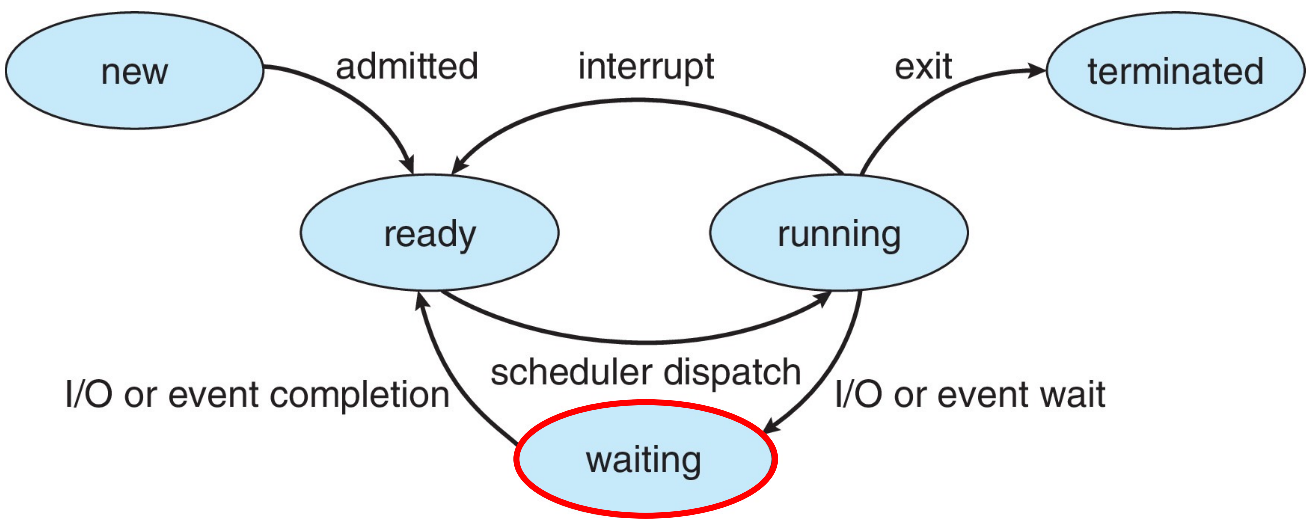
Process Concept

Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
```

The process has requested I/O.
The scheduler moves this process from running to waiting list.
The process is waiting for I/O to complete.

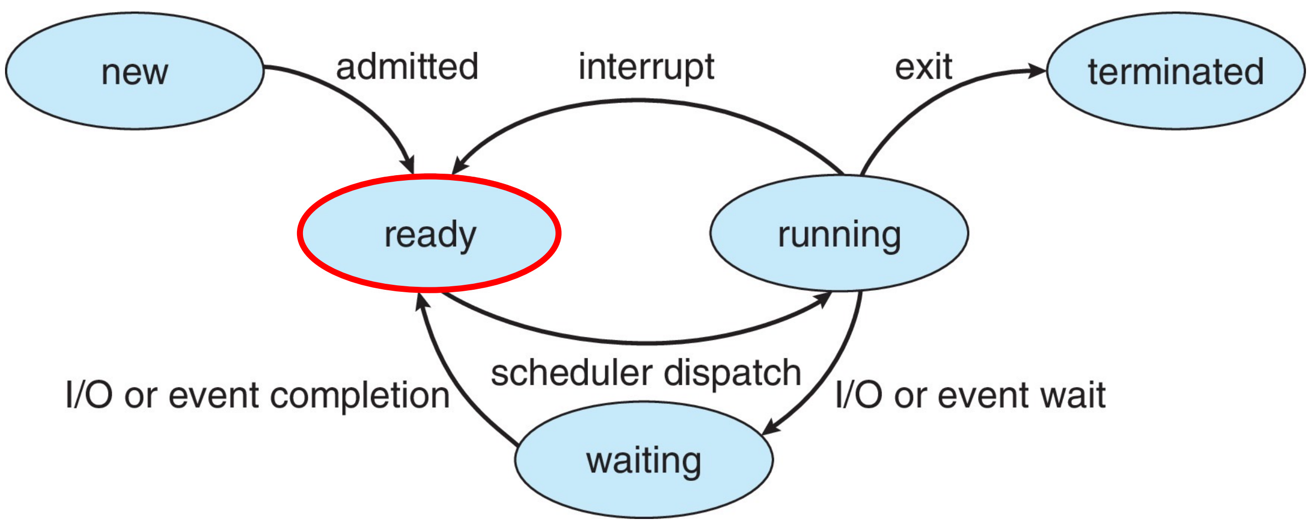


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
```

I/O completed, sends an interrupt to the CPU. The CPU scheduler moves this process to ready list. But this process is not running yet.

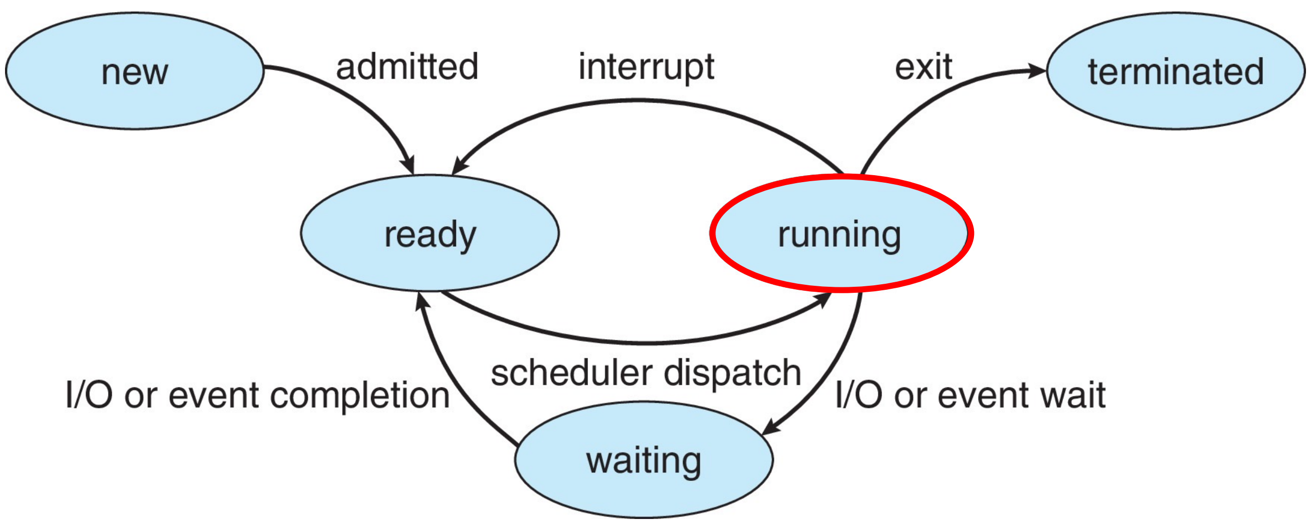


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
```

This process is scheduled to run, performing normal function logic.

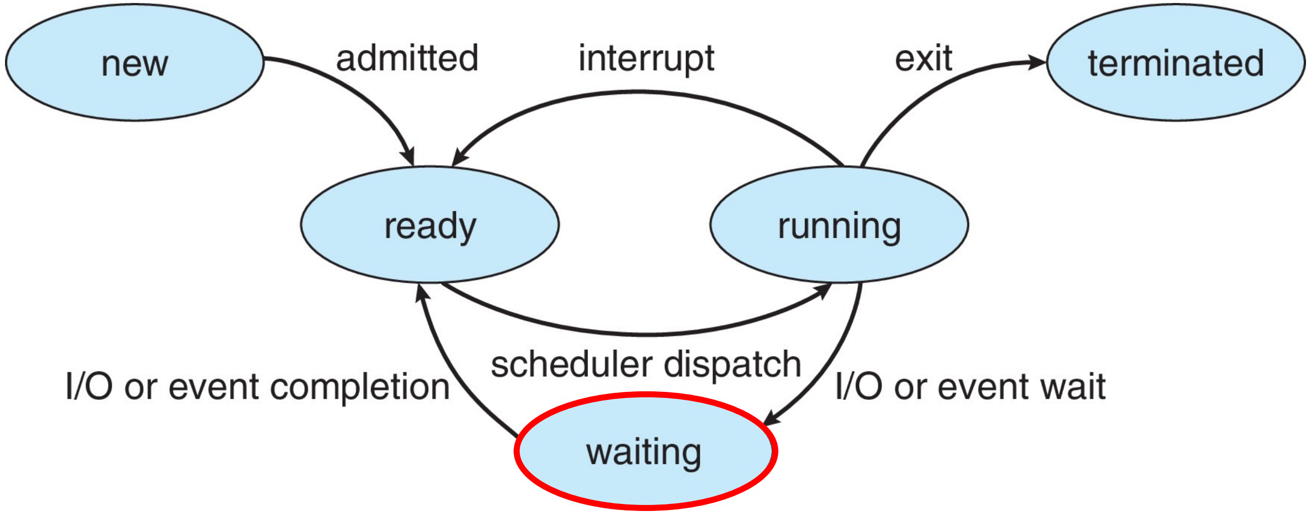


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
```

The process has requested I/O again.
The scheduler moves this process from running to waiting list.
The process is waiting for I/O to complete.

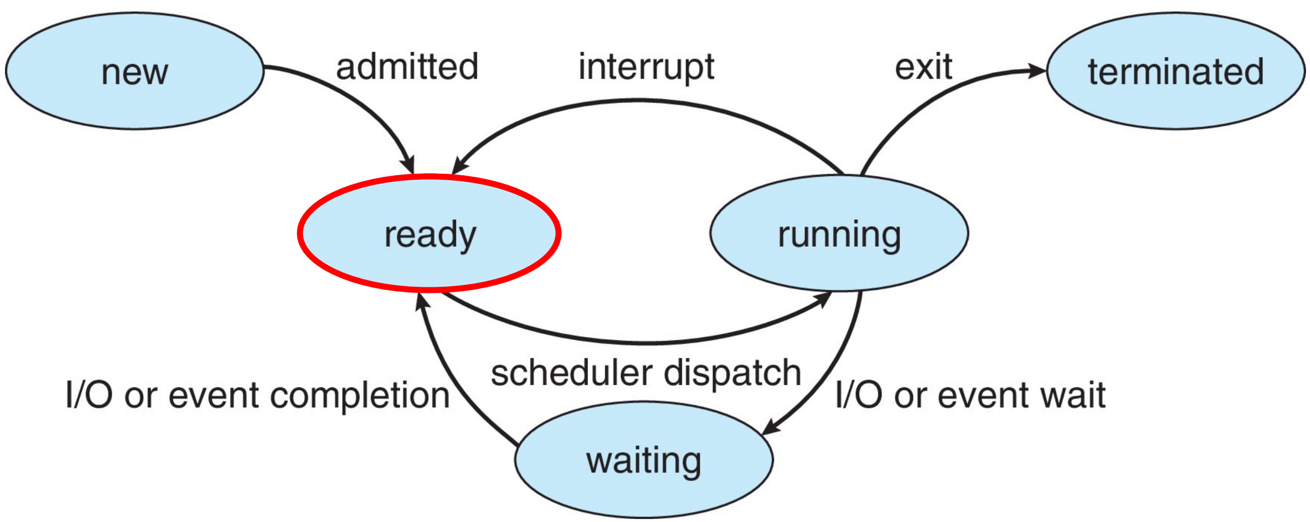


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
n*n: 9
```

I/O completed, sends an interrupt to the CPU. The CPU scheduler moves this process to ready list. But this process is not running yet.

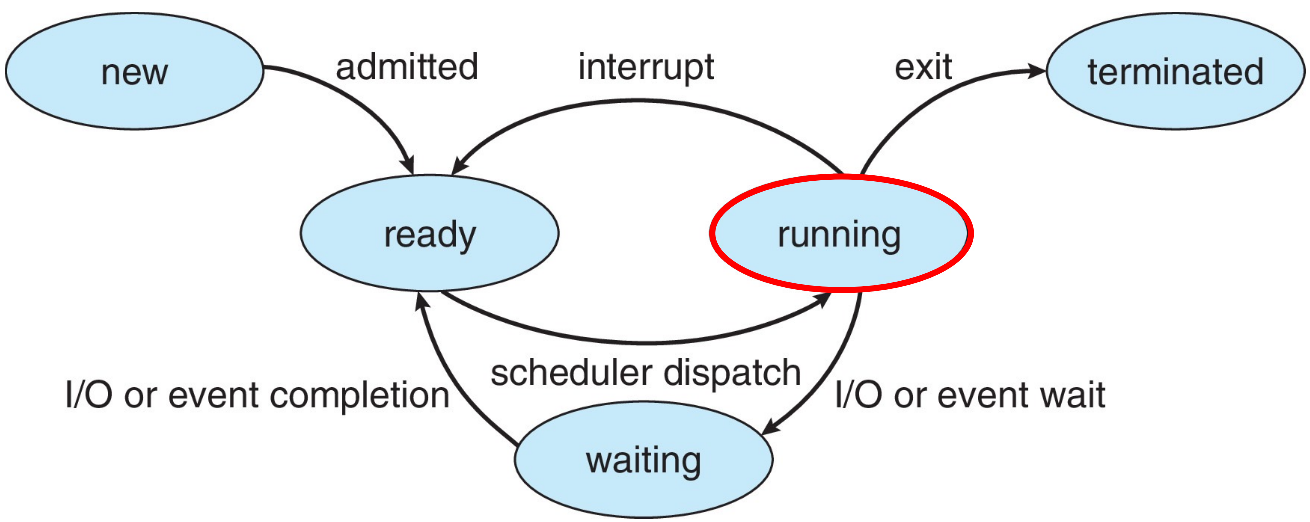


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
n*n: 9
```

This process is scheduled to run, performing normal function logic.

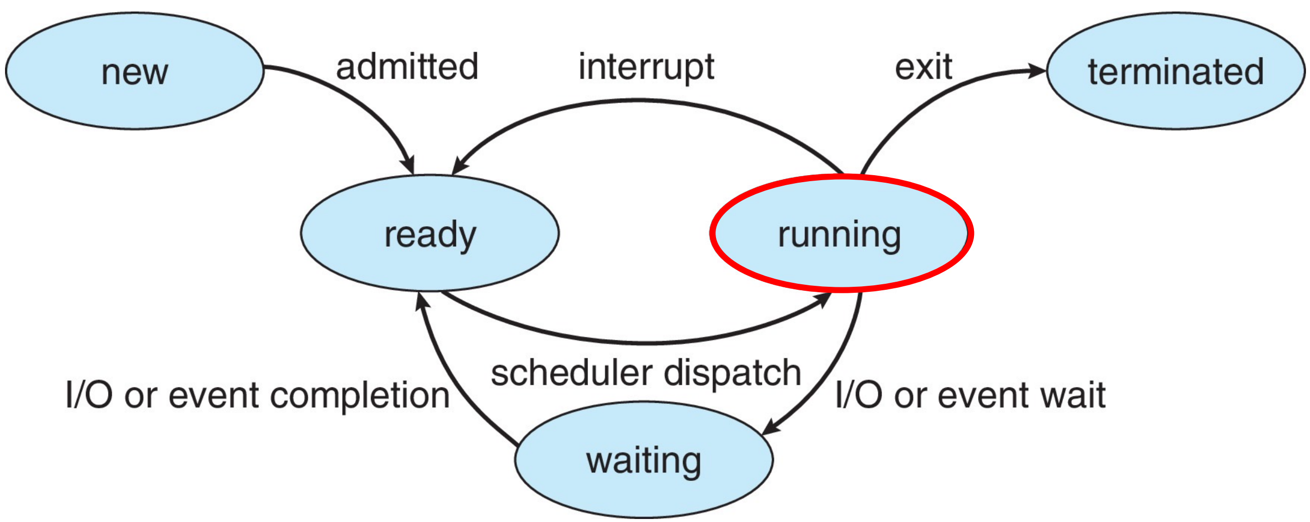


Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
n*n: 9
```

This process has reached the end of program execution, e.g., invoking `exit()` system call to terminate.



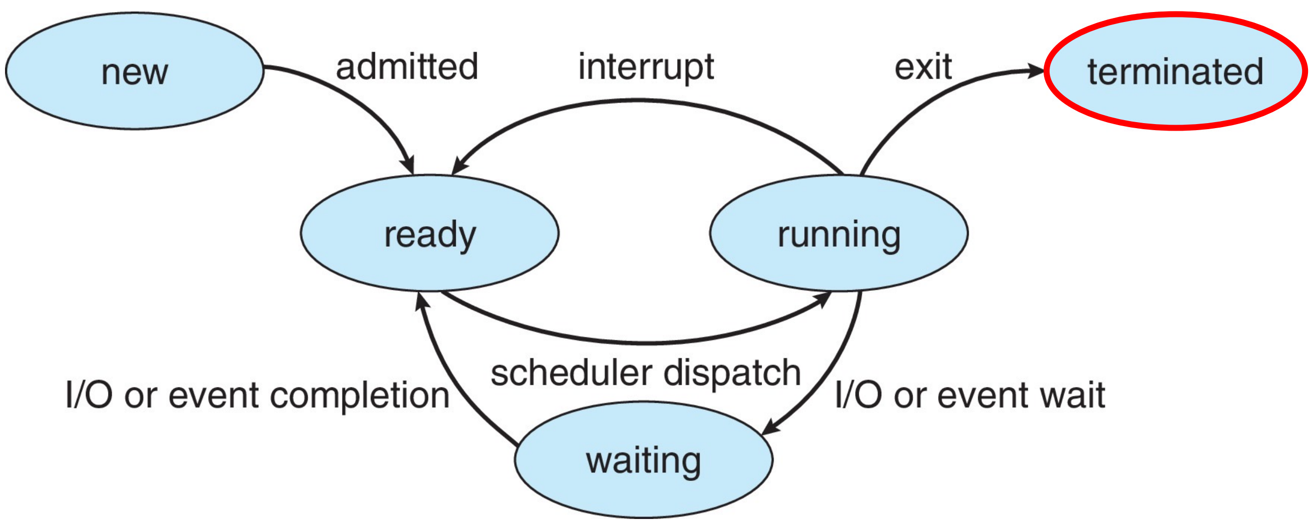
Process Concept

Process States Example

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    n = n * n;
    printf("n*n: %d\n", n);
    return 0;
}
```

```
$ gcc -o ps_states ps_states.c
$ ./ps_states
3
n*n: 9
```

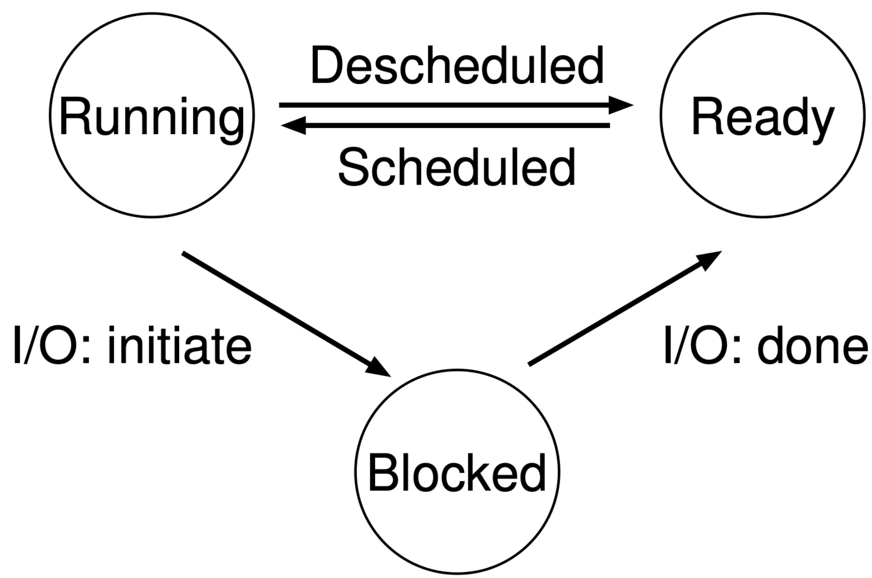
This process is now terminated by the OS.



Process Concept

Process States

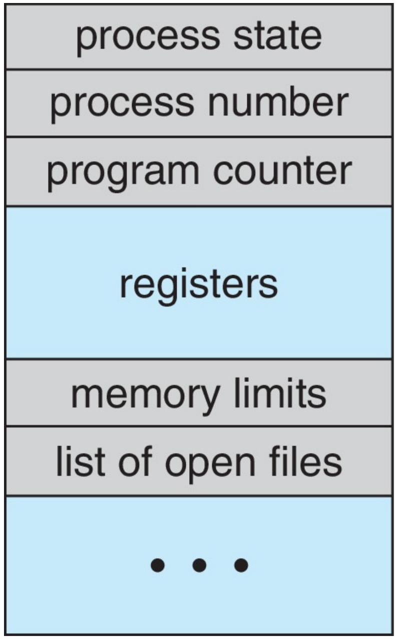
- A Simplified View:
  - **Running:** Instructions are being executed
  - **Waiting (Blocked):** The process is waiting for some event to occur
  - **Ready:** The process is waiting to be assigned to a processor



Process Concept

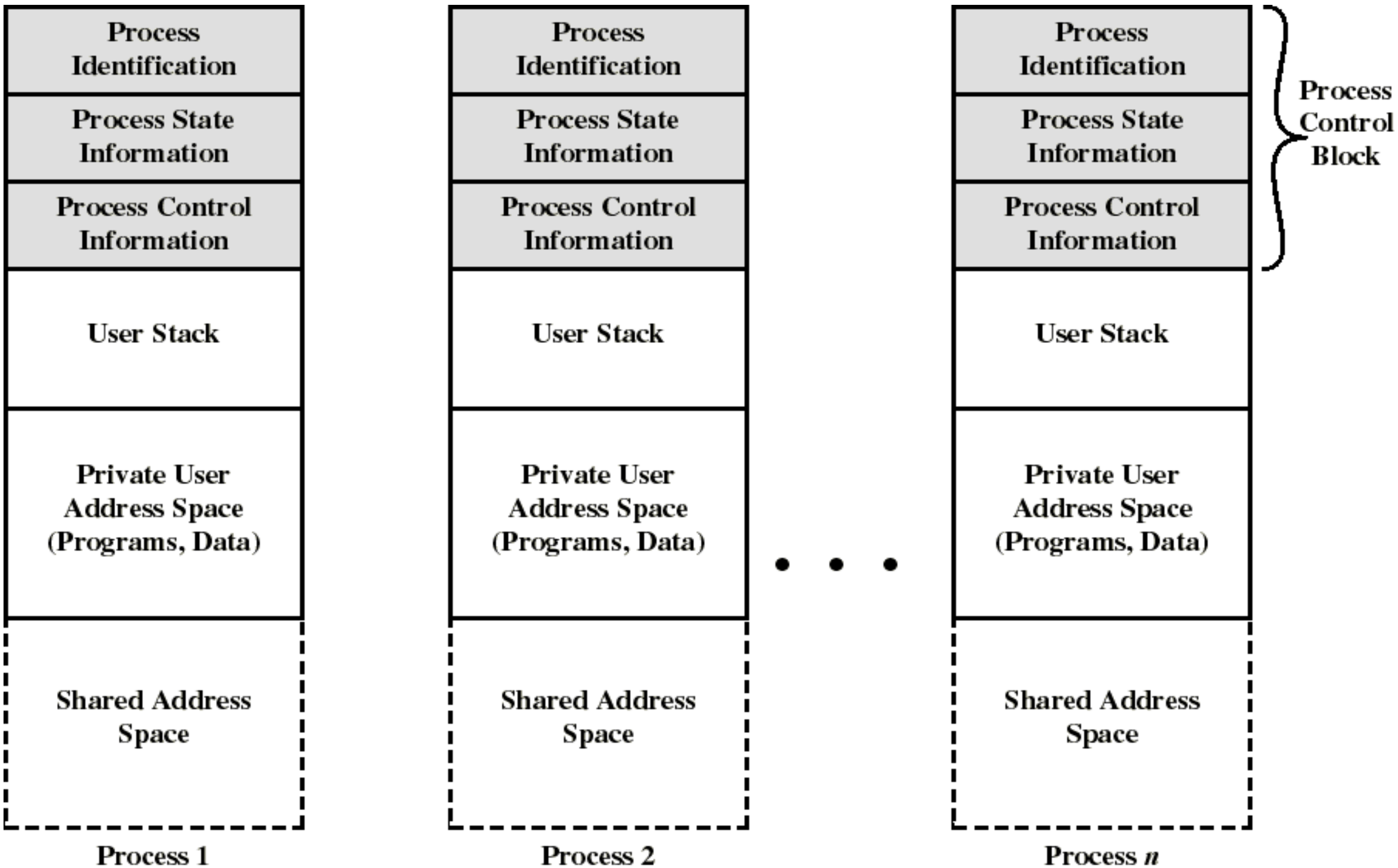
- **Process Control Block (PCB)**

  (Information associated with each process):
  - **Process State**: running, waiting, etc.
  - **Program Counter:** location of instruction to execute next
  - **CPU Registers**: contents of all process-centric registers
  - **CPU Scheduling Info**: priorities, scheduling queues
  - **Memory Management Info**: memory allocated to process
  - **Accounting Info**: CPU used, clock time elapsed, time limits
  - **I/O Info**: I/O devices allocated to process, list of open files



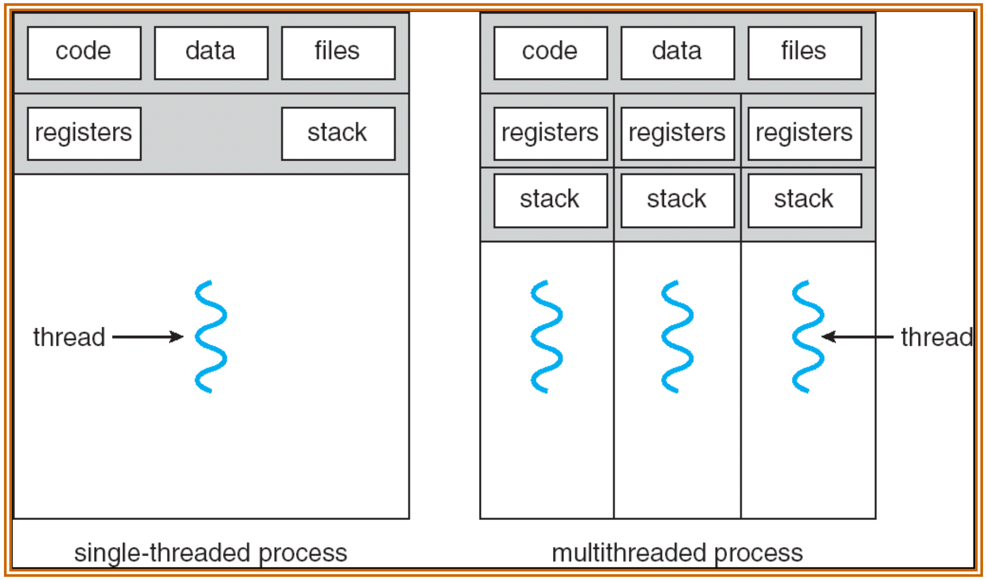
Process Control Block (PCB)

- Where exactly is the PCB?
  - In the kernel's memory space, part of the kernel's data structures.



Threads

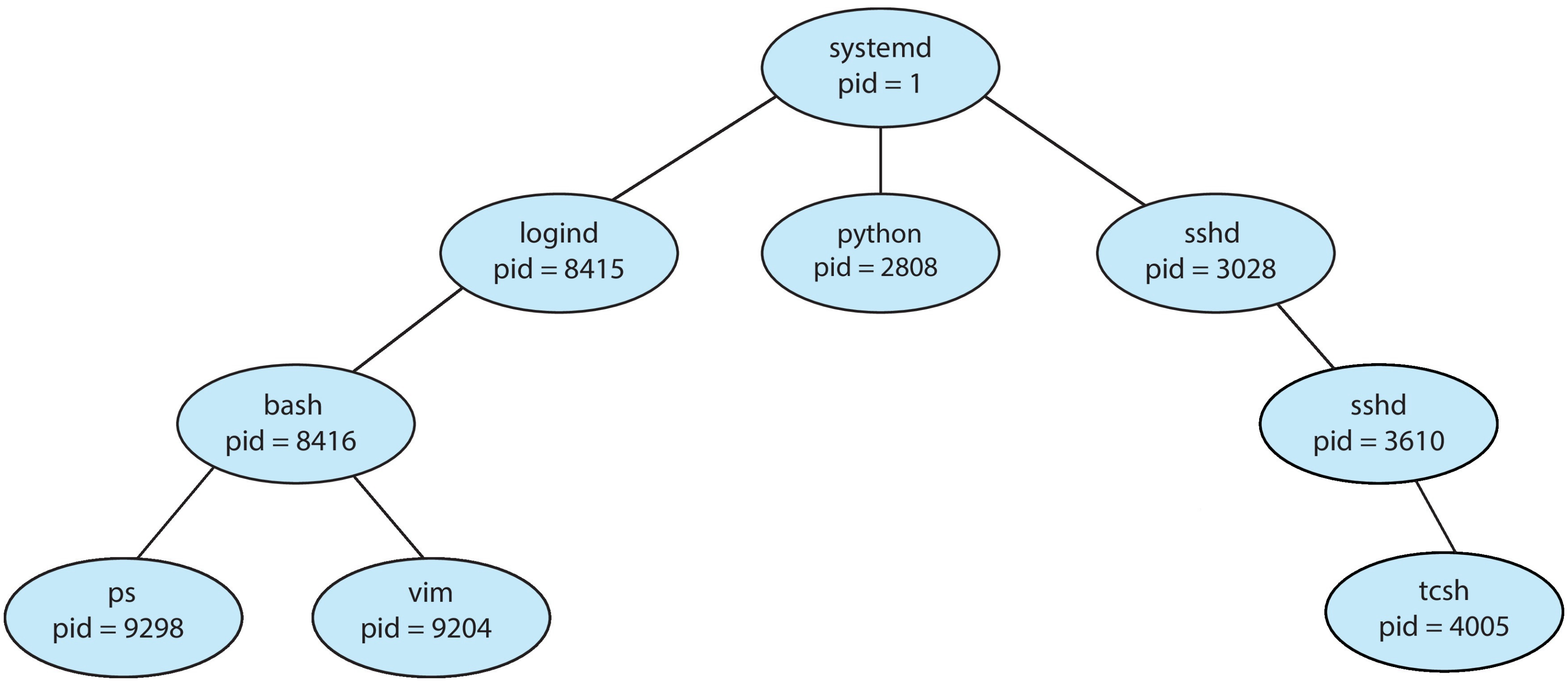
- The process model discussed so far has implied that a process is a running program that performs a single **thread** of execution
- Modern OSes have extended the process concept to allow a process to have multiple **threads** of execution
  - thus enabling a process to perform more than one task at a time
  - multiple threads of control → **threads**
  - PCB extended to contain multiple program counters
- *More on Threads in later chapters when dealing with concurrency*



- **Operations on Processes**
  - Processes in most systems can execute concurrently
  - OS can spawn (衍生) new processes (Of course!)
  - Processes themselves should also be able to spawn new processes
    - via system calls
  - OS 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 process identifier (pid)



Process Creation

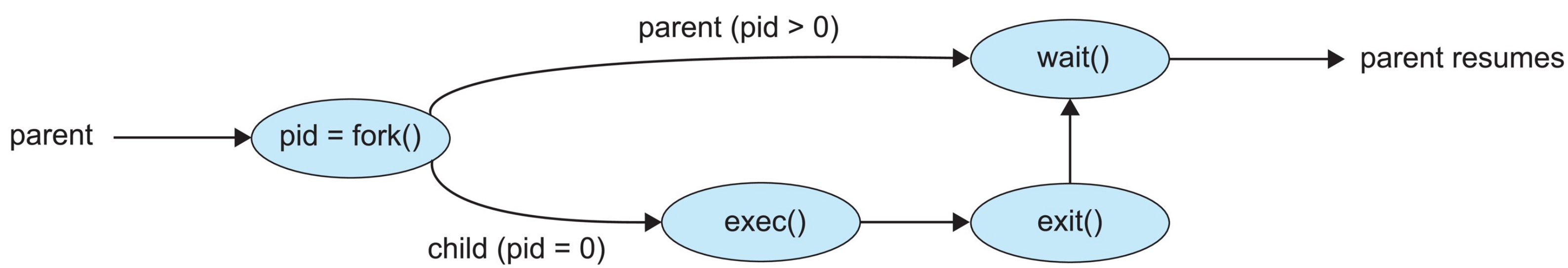
- **Parent** process create **children** processes, which in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via process identifier (pid)

```
$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─agetty
        ├─containerd───12*[{containerd}]
        ├─cron
        ├─dockerd───13*[{dockerd}]
        ├─rsyslogd───3*[{rsyslogd}]
        ├─sshd───sshd───sshd───bash───tmux: client
        ├─systemd─┬─(sd-pam)
        │         ├─dbus-daemon
        │         └─pulseaudio───2*[{pulseaudio}]
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-resolve
        ├─systemd-timesyn───{systemd-timesyn}
        ├─tmux: server─┬─bash───emacs
        │              ├─6*[bash]
        │              ├─bash───vim
        │              └─bash───pstree
        └─upowerd───2*[{upowerd}]
```

- **Process Creation**
  - **Parent** process create **children** processes, which in turn create other processes, forming a **tree** of processes
  - Generally, process identified and managed via process identifier (pid)
  - Possible resource sharing:
    - Parent and children processes share **all** resources
    - Children processes share **subset** of parent's resources
    - Parent and children processes share **no** resources
  - Possible Execution:
    - Parent and children processes execute **concurrently**
    - Parent **waits** until children processes terminate
  - Address space:
    - Child duplicate of parent
    - Child has a program loaded into it (different address space)

Process Creation

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



Process Creation

UNIX fork()

- `pid_t fork(void);` causes creation of a new process.

```c
/* p1_fork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {          /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
    } else {               /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc, getpid());
    }

    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

Process Creation

UNIX fork()

- `pid_t fork(void);` causes creation of a new process.

```c
/* p1_fork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
    } else {                /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc, getpid());
    }

    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

Parent and child run concurrently. Either parent or child could get executed first. This could also happen:

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am child (pid: 96648)
Hi, I am parent of 96648 (pid: 96647)
```

- **UNIX `wait()`**
  - `pid_t wait(int *stat_loc);` suspends execution of current process until its child process has completed.

```
/* p2_fork_wait.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
    } else {                /* Parent (old process) */
        int rc_wait = wait(NULL);
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc_wait, getpid());
    }

    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

```
$ gcc -o p2_fork_wait p2_fork_wait.c
$ ./p2_fork_wait
Hello world (pid: 96874)
Hi, I am child (pid: 96875)
Hi, I am parent of 96875 (pid: 96874)
```

Process Creation

UNIX `wait()`

- `pid_t wait(int *stat_loc);` suspends execution of current process until its child process has completed.

```c
/* p2_fork_wait.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
    } else {               /* Parent (old process) */
        int rc_wait = wait(NULL);
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc_wait, getpid());
    }

    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

```
$ gcc -o p2_fork_wait p2_fork_wait.c
$ ./p2_fork_wait
Hello world (pid: 96874)
Hi, I am child (pid: 96875)
Hi, I am parent of 96875 (pid: 96874)
```

`wait()` system call **blocks** the parent process until the child process finishes execution, thus making sure **child finishes before parent**.

UNIX exec()

- `int exec(...);` transforms the calling process into a new process

```
/* p3_fork_wait_exec.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {          /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
        execl("/bin/date", "date", NULL);
        printf("This line shouldn't print.\n");
    } else {               /* Parent (old process) */
        int rc_wait = wait(NULL);
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc_wait, getpid());
    }
    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

```
$ gcc -o p2_fork_wait p2_fork_wait.c
$ ./p2_fork_wait
Hello world (pid: 96874)
Hi, I am child (pid: 96875)
Hi, I am parent of 96875 (pid: 96874)
```

```
$ gcc -o p3_fork_wait_exec p3_fork_wait_exec.c
$ ./p3_fork_wait_exec
Hello world (pid: 97477)
Hi, I am child (pid: 97478)
Sun Mar 10 06:32:20 CST 2024
Hi, I am parent of 97478 (pid: 97477)
```

Process Creation

UNIX exec()

- `int exec(...);` transforms the calling process into a new process

```c
/* p3_fork_wait_exec.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
        execl("/bin/date", "date", NULL);
        printf("This line shouldn't print.\n");
    } else {                /* Parent (old process) */
        int rc_wait = wait(NULL);
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc_wait, getpid());
    }
    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

```
$ gcc -o p2_fork_wait p2_fork_wait.c
$ ./p2_fork_wait
Hello world (pid: 96874)
Hi, I am child (pid: 96875)
Hi, I am parent of 96875 (pid: 96874)
```

```
$ gcc -o p3_fork_wait_exec p3_fork_wait_exec.c
$ ./p3_fork_wait_exec
Hello world (pid: 97477)
Hi, I am child (pid: 97478)
Sun Mar 10 06:32:20 CST 2024
Hi, I am parent of 97478 (pid: 97477)
```

child now runs a different program

UNIX exec()

- `int exec(...);` transforms the calling process into a new process

```c
/* p3_fork_wait_exec.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    printf("Hello world (pid: %d)\n", getpid());

    pid_t rc = fork();

    if (rc < 0) {          /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("Hi, I am child (pid: %d)\n",
                getpid());
        execl("/bin/date", "date", NULL);
        printf("This line shouldn't print.\n");
    } else {               /* Parent (old process) */
        int rc_wait = wait(NULL);
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc_wait, getpid());
    }
    return 0;
}
```

```
$ gcc -o p1_fork p1_fork.c
$ ./p1_fork
Hello world (pid: 96647)
Hi, I am parent of 96648 (pid: 96647)
Hi, I am child (pid: 96648)
```

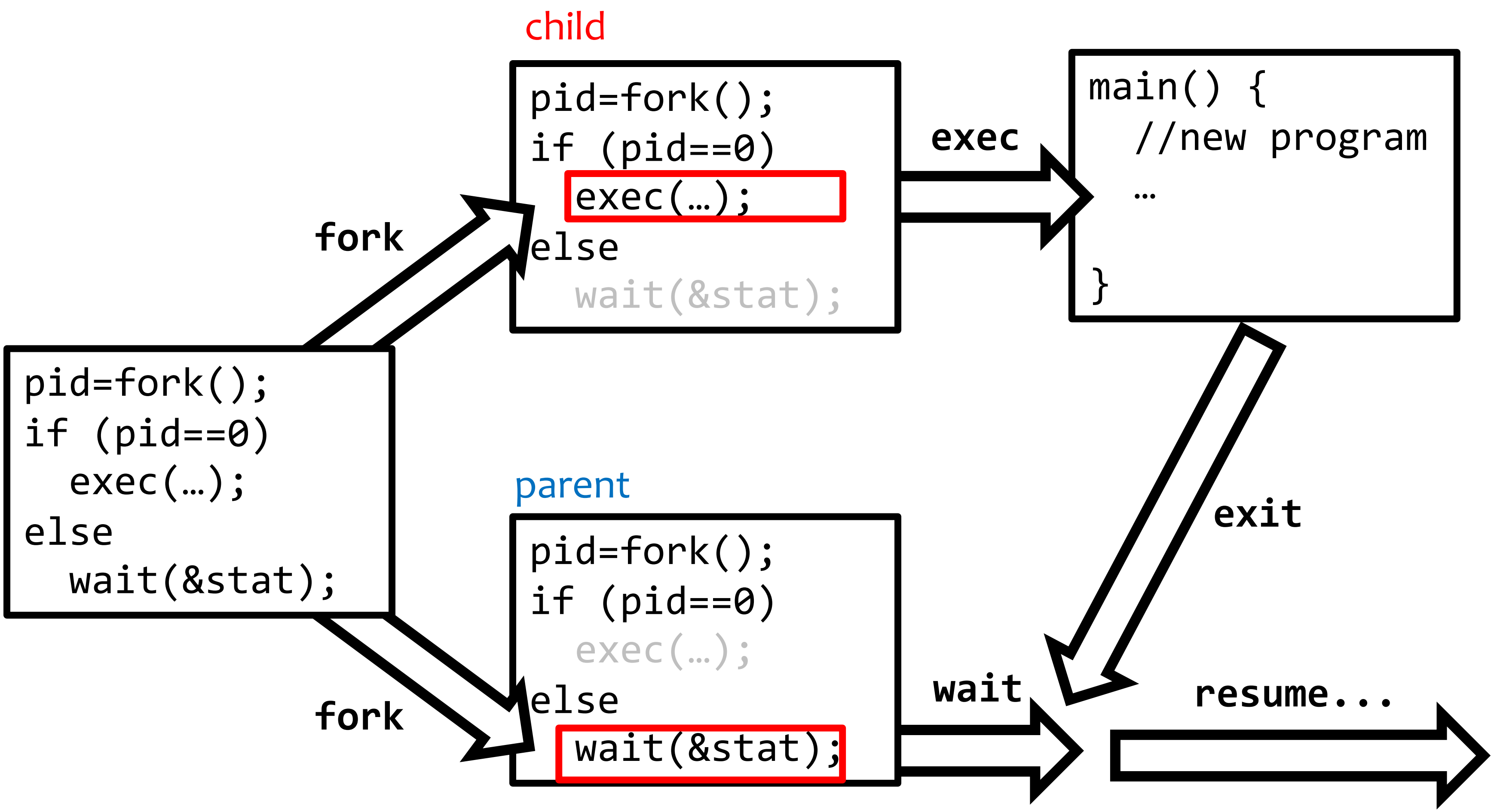
```
$ gcc -o p2_fork_wait p2_fork_wait.c
$ ./p2_fork_wait
Hello world (pid: 96874)
Hi, I am child (pid: 96875)
Hi, I am parent of 96875 (pid: 96874)
```

```
$ gcc -o p3_fork_wait_exec p3_fork_wait_exec.c
$ ./p3_fork_wait_exec
Hello world (pid: 97477)
Hi, I am child (pid: 97478)
Sun Mar 10 06:32:20 CST 2024
Hi, I am parent of 97478 (pid: 97477)
```

program code overwritten, old program instructions discarded.

Process Creation

- UNIX fork(), exec(), wait(), exit()



Process Creation

- **UNIX `exec()`**
  - `int exec(...);` transforms the calling process into a new process
  - `exec()` is actually a family of functions (`` `man 3 exec` ``):
    - `execl("/bin/ls", "ls", NULL);`
    - `execlp("ls", "ls", NULL);`
    - `execle("/bin/ls", "ls", NULL, NULL);`
    - `execv("/bin/ls", param_list);`
    - `execvp("ls", param_list);`
    - `execvpe("ls", param_list, envp_lsit);`
  - `execve()` is the actual system call (`` `man 2 execve` ``) for UNIX process creation. The family of `exec()` functions are C Library Functions that call `execve()`.
  - But anyway, we still call `exec()` a system call for historical reasons and by convention.

Process Creation

- UNIX API
  - `fork()` system call creates new process
  - `exec()` system call used after `fork()` to replace the process' memory space with new program
  - Parent process calls `wait()` waiting for the child to terminate
- Windows API
  - A single `CreateProcess()` system call
    - Complex (ugly) interface

Process Creation

- UNIX API
  - `fork()` system call creates new process
  - `exec()` system call used after `fork()` to replace the process' memory space with new program
  - Parent process calls `wait()` waiting for the child to terminate
- Windows API
  - A single `CreateProcess()` system call
    - Complex (ugly) interface (10 parameters)

```
// Start the child process.
if (!CreateProcess( NULL,   // No module name (use command line)
    argv[1],         // 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
)
```

Process Termination

- Process executes last statement and then asks the OS to delete using the `exit()` system call.
  - returns status data from child to parent (via `wait()`)
  - process' resources are deallocated by OS
- Parent may terminate the execution of children processes
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting
    - Some OSes do not allow a child to continue if its parent terminates
    - Cascading Termination – all children terminated

Process Termination

- Some OSes 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()`), then the terminated child process is a zombie
- If parent terminated without invoking `wait()`, then the (**still running**) child process is an orphan

Zombie (僵尸进程)

- A zombie is a process that has **completed execution** but still has an entry in the process table. It often occurs when the parent process **didn't call** `wait()` to read the exit status of the child process.

```c
/* p4_zombie.c */
int main(int argc, char *argv[]) {
    pid_t rc = fork();

    if (rc < 0) {          /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("    Hi, I am child (pid: %d)\n",
               getpid());
        printf("    Child process exiting...\n");
        exit(0);
    } else {               /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n",
               rc, getpid());
        printf("Parent sleeping...\n");
        sleep(60);
        exit(0);
    }
    return 0;
}
```

```
$ gcc -o p4_zombie p4_zombie.c
$ ./p4_zombie
Hi, I am parent of 4120 (pid: 4118)
Parent sleeping...
    Hi, I am child (pid: 4120)
    Child process exiting...
```

```
$ ps
  PID TTY           TIME CMD
 3855 ttys010    0:00.38 -zsh
 4118 ttys011    0:00.00 ./p4_zombie
 4120 ttys011    0:00.00 <defunct>
```

Child process is a zombie

Process Termination

Zombie (僵尸进程)

- The `fork()` system call creates a new process (child)
- The child process **exits** immediately after printing its message
- The parent process, instead of calling `wait()`, goes to sleep for 60s.
- During the parent's sleep, the child becomes a **zombie**

```c
/* p4_zombie.c */
int main(int argc, char *argv[]) {
    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("    Hi, I am child (pid: %d)\n",
                getpid());
        printf("    Child process exiting...\n");
        exit(0);
    } else {                /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n",
                rc, getpid());
        printf("Parent sleeping...\n");
        sleep(60);
        exit(0);
    }
    return 0;
}
```

```
$ gcc -o p4_zombie p4_zombie.c
$ ./p4_zombie
Hi, I am parent of 4120 (pid: 4118)
Parent sleeping...
    Hi, I am child (pid: 4120)
    Child process exiting...
```

```
$ ps
  PID TTY           TIME CMD
 3855 ttys010    0:00.38 -zsh
 4118 ttys011    0:00.00 ./p4_zombie
 4120 ttys011    0:00.00 <defunct>
```

Child process is a zombie

Orphan (孤儿进程)

- An orphan process occurs when a child process is still running after its parent process has finished. In most OSes, when a parent process terminates, any running child processes are adopted by a special process called `init` or `systemd`, with the pid of 1.

```c
/* p5_orphan.c */
int main(int argc, char *argv[]) {
    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("    Hi, I am child (pid: %d, ppid: %d)\n", getpid(), getppid());
        sleep(10);
        printf("    Hi, I am child (pid: %d, ppid: %d)\n", getpid(), getppid());
    } else {                /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n", rc, getpid());
        sleep(1);
        printf("Parent exiting...\n");
        exit(0);
    }
    return 0;
}
```

```
$ gcc -o p5_orphan p5_orphan.c
$ ./p5_orphan
Hi, I am parent of 4600 (pid: 4599)
    Hi, I am child (pid: 4600, ppid: 4599)
Parent exiting...
    Hi, I am child (pid: 4600, ppid: 1)
```

After parent process terminates, the child process is adopted (领养) by systemd (pid:1)

Orphan (孤儿进程)

- The fork() system call creates a new process (child)
- The parent process prints its message, sleep(1), and then exits.
- The child process prints its message and sleeps for 10s. During child's sleep, the parent process already terminates, leaving the child an orphan.

```c
/* p5_orphan.c */
int main(int argc, char *argv[]) {
    pid_t rc = fork();

    if (rc < 0) {           /* Fork failed */
        fprintf(stderr, "Fork Failed\n");
        exit(1);
    } else if (rc == 0) { /* Child (new process) */
        printf("    Hi, I am child (pid: %d, ppid: %d)\n", getpid(), getppid());
        sleep(10);
        printf("    Hi, I am child (pid: %d, ppid: %d)\n", getpid(), getppid());
    } else {               /* Parent (old process) */
        printf("Hi, I am parent of %d (pid: %d)\n", rc, getpid());
        sleep(1);
        printf("Parent exiting...\n");
        exit(0);
    }
    return 0;
}
```

```
$ gcc -o p5_orphan p5_orphan.c
$ ./p5_orphan
Hi, I am parent of 4600 (pid: 4599)
    Hi, I am child (pid: 4600, ppid: 4599)
Parent exiting...
    Hi, I am child (pid: 4600, ppid: 1)
```

After parent process terminates, the child process is adopted (领养) by systemd (pid:1)

Thank you!