Lecture 15: OS and Processes

CS 105 Spring 2025

Intro to Operating Systems

- the **operating system** is a piece of software that manages a computer's resources for its users and their applications
  - Examples: OSX, Windows, Ubuntu, iOS, Android, Chrome OS

Intro to Operating Systems

Intro to Operating Systems

- resource allocation
- isolation
- communication
- access control

- multiprocessing
- virtual memory
- reliable networking
- virtual machines

- user interface
- file I/O
- device management
- process control

Operating System Modes

Kernel Mode	User Mode

Operating System Modes

Kernel Mode

- unrestricted access to hardware

User Mode

- must ask kernel to access hw (system call)

Operating System Modes

Kernel Mode

- unrestricted access to hardware
- mediates all hardware access (access control)

User Mode

- must ask kernel to access hw (system call)

Operating System Modes

Kernel Mode

- unrestricted access to hardware
- mediates all hardware access (access control)
- can execute privileged instructions

User Mode

- must ask kernel to access hw (system call)
- attempts to execute privileged instructions cause exceptions

Operating System Modes

Kernel Mode

- unrestricted access to hardware
- mediates all hardware access (access control)
- can execute privileged instructions

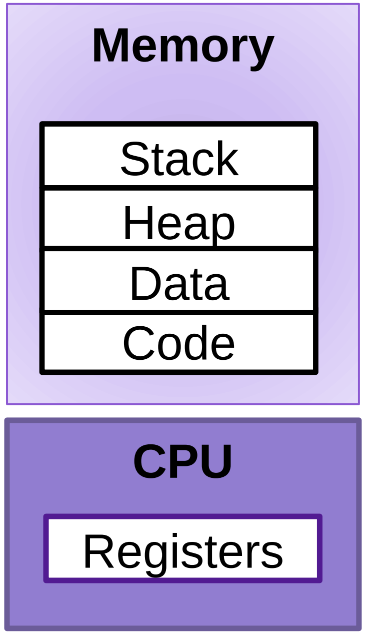
User Mode

- must ask kernel to access hw (system call)
- attempts to execute privileged instructions cause exceptions

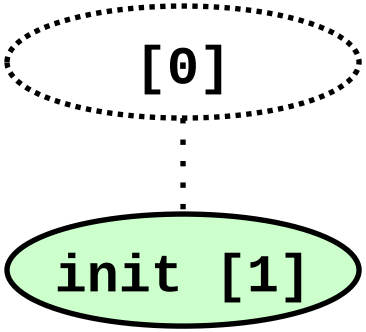
- Operating system mode is set in hardware, can't be changed by user-level code

Processes

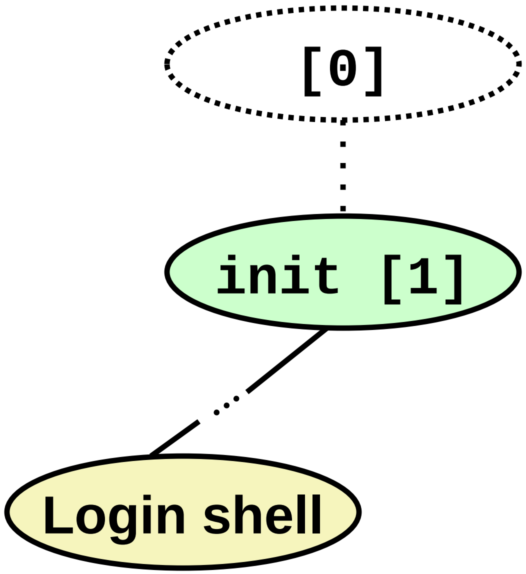
- A ***program*** is a file containing code + data that describes a computation
- A ***process*** is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"



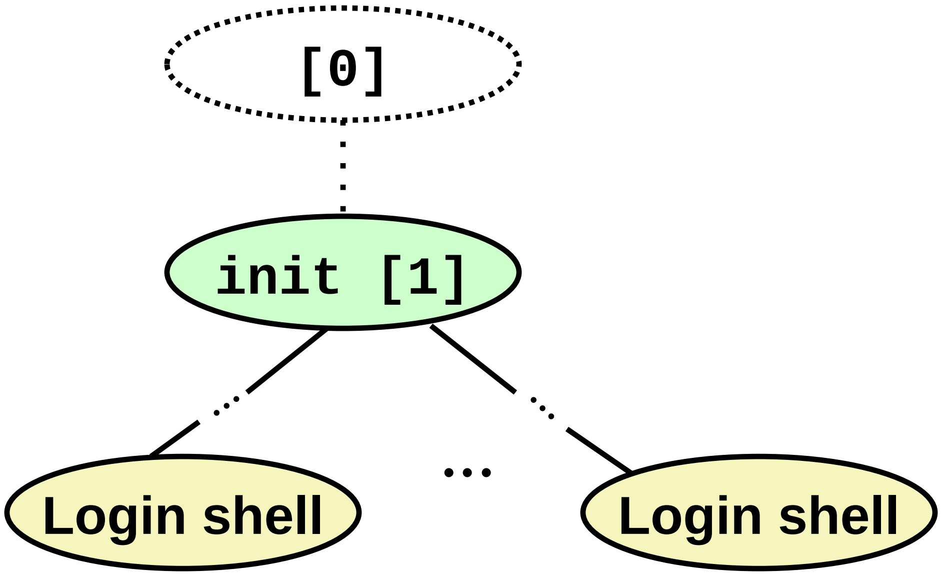
Linux Process Hierarchy



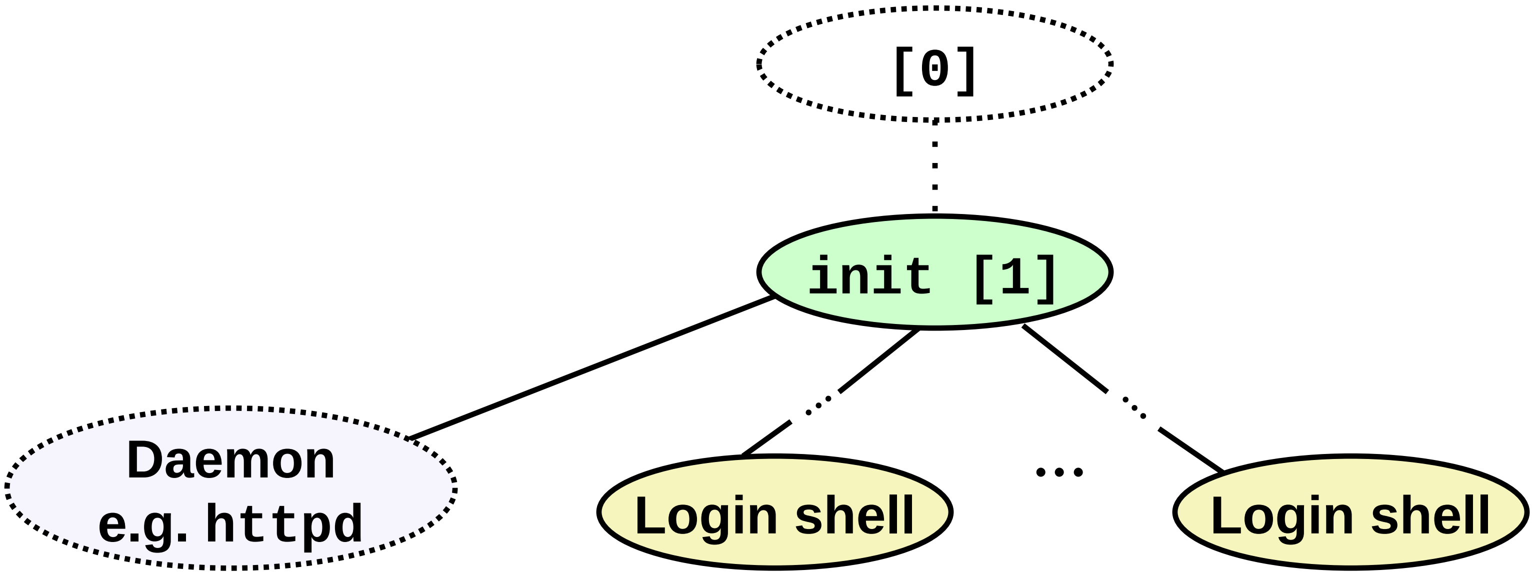
Linux Process Hierarchy



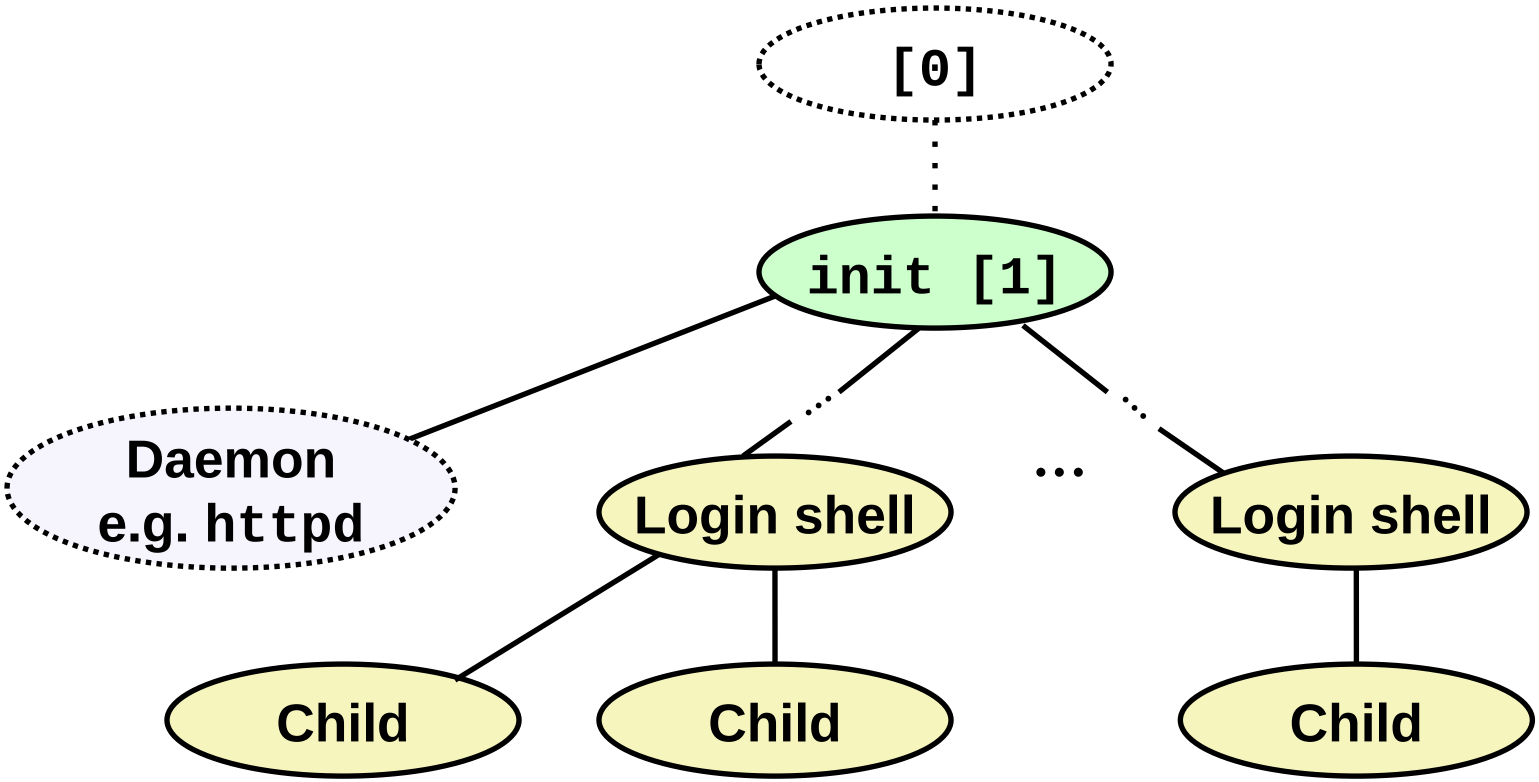
Linux Process Hierarchy



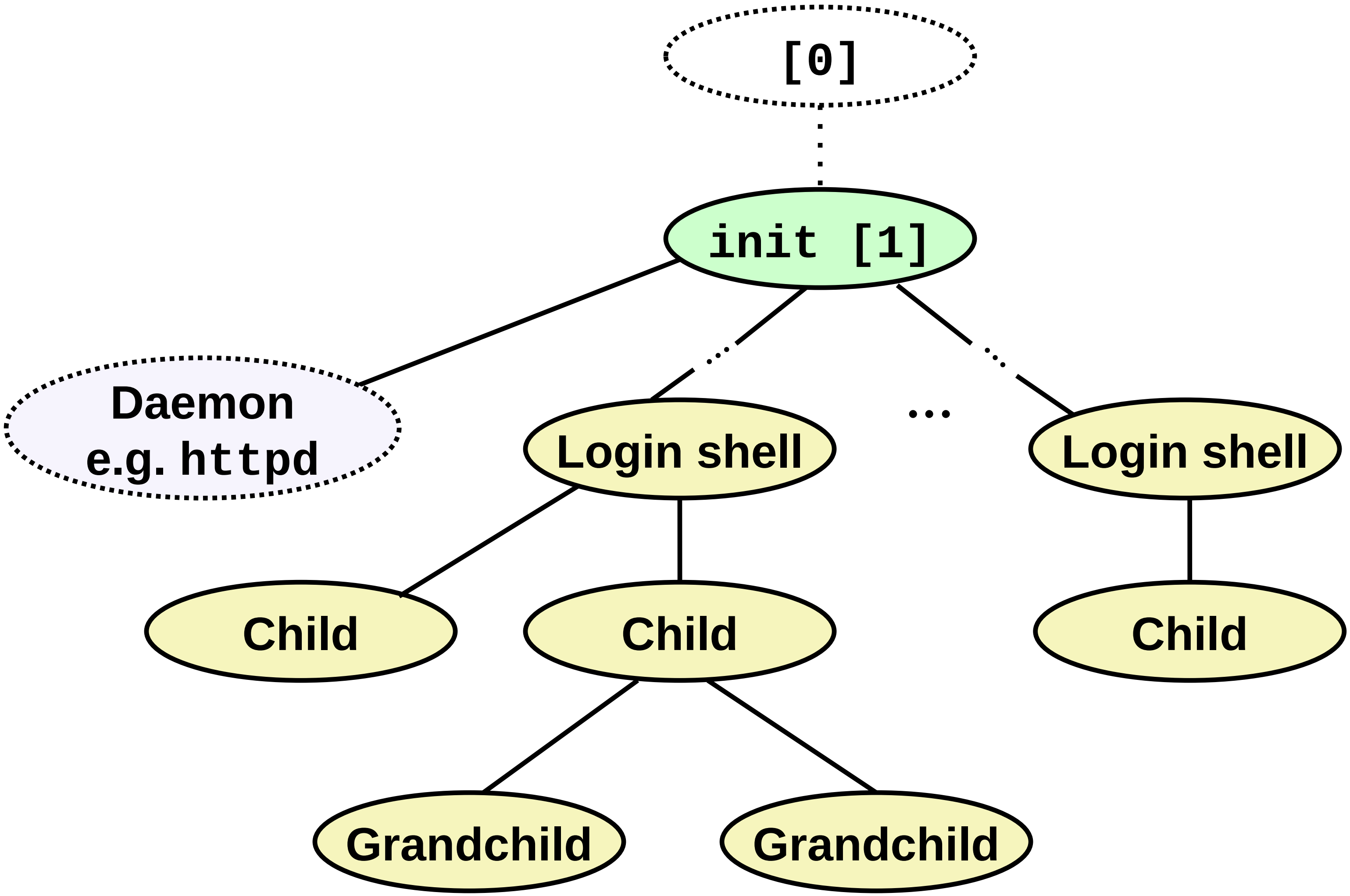
Linux Process Hierarchy



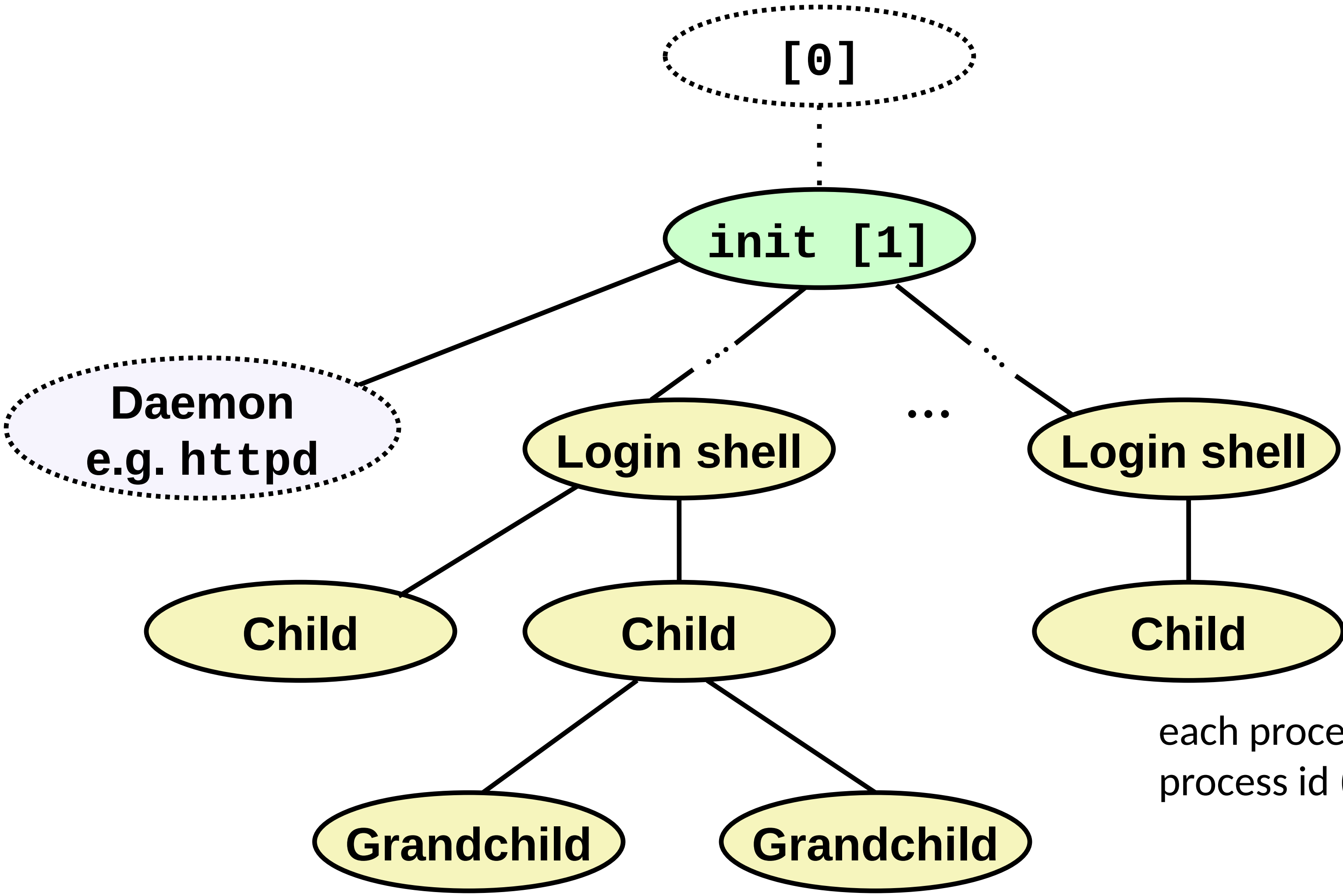
Linux Process Hierarchy



Linux Process Hierarchy

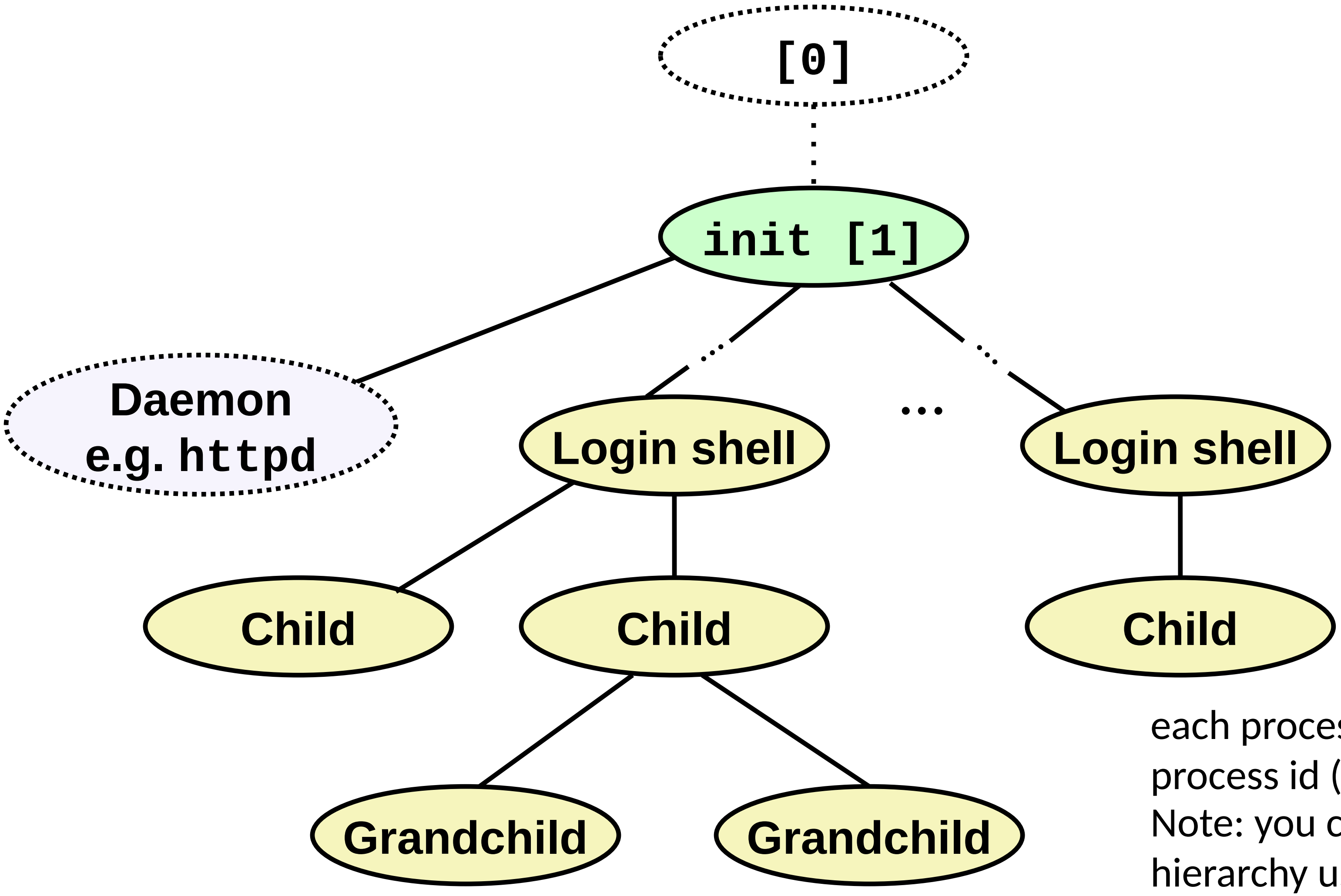


Linux Process Hierarchy



each process has a unique process id (**pid**)

Linux Process Hierarchy



each process has a unique process id (**pid**)
Note: you can view the hierarchy using the Linux `pstree` command

Creating Processes

- *Parent process* creates a new running *child process* by calling **`fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

Creating Processes

- *Parent process* creates a new running *child process* by calling **`fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- **`fork`** is interesting (and often confusing) because it is called ***once*** but returns ***twice***

fork Example

```
int main(){

    pid_t id;
    int x = 1;

    id = fork();
    if (id == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- Call once, return twice

fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent

fork Example

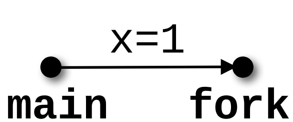
```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
          return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

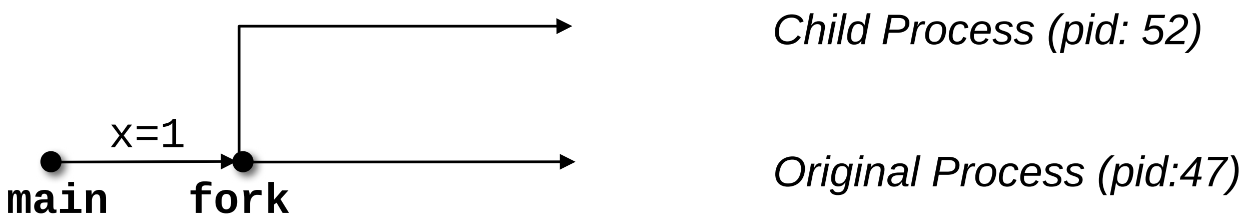


Original Process (pid:47)

fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

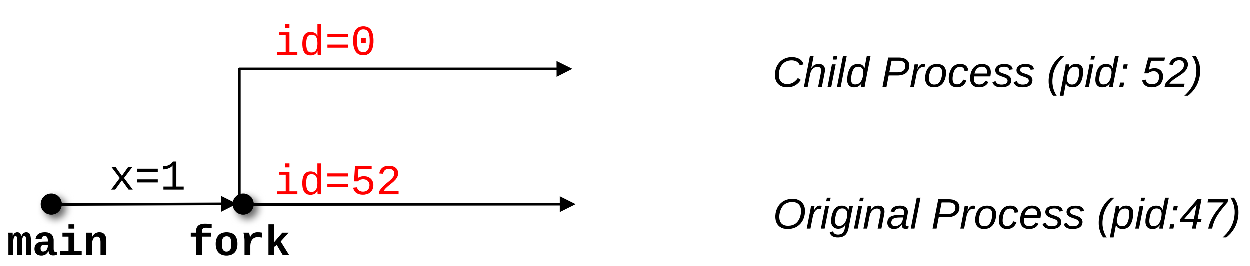
- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child



fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

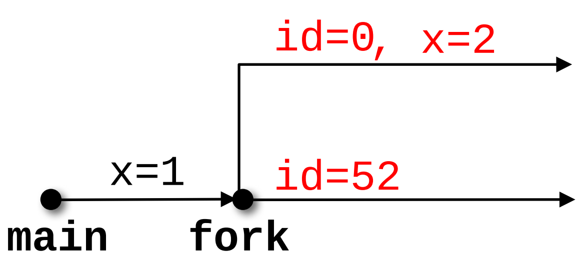
- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child



fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child



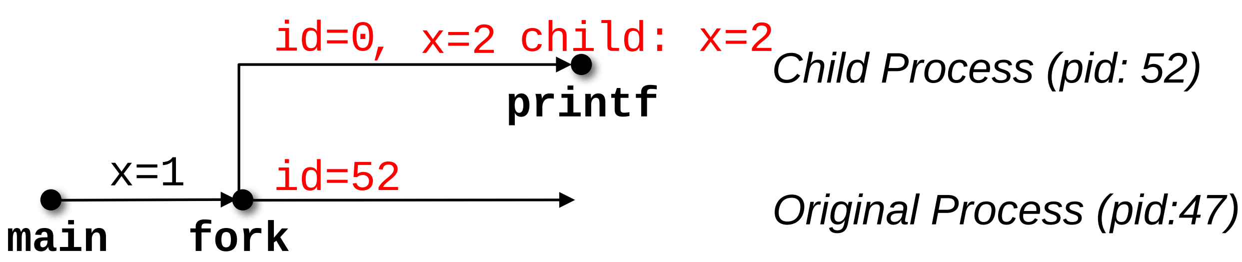
Child Process (pid: 52)

Original Process (pid:47)

fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

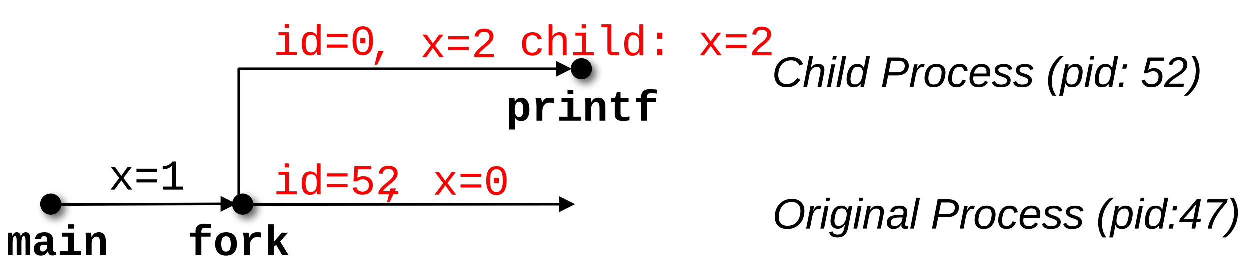
- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - stdout is the same in both parent and child



fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

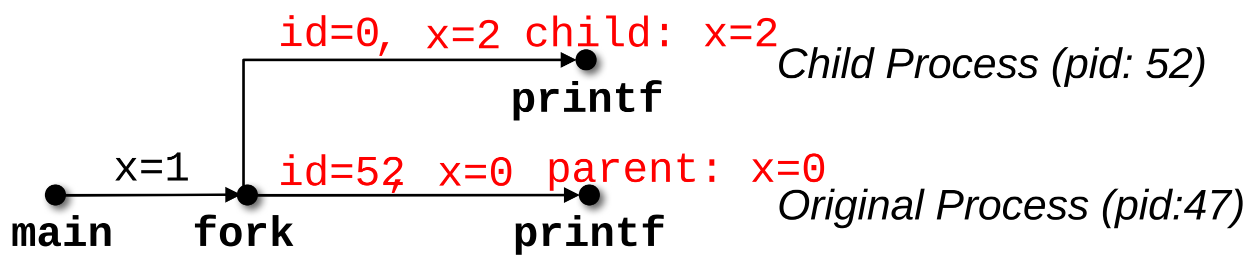
- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - stdout is the same in both parent and child



fork Example

```
int main(){

   pid_t id;
   int x = 1;

   id = fork();
   if (id == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
         return 0;
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - stdout is the same in both parent and child



execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`

execve: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- Loads and runs in the current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`

execve: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- Loads and runs in the current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context

execve: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- Loads and runs in the current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called **once** and **never** returns
  - …except if there is an error

execve Example

```c
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

```c
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c

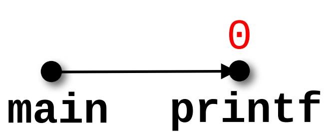
execve Example

```
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

```
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c



Parent (pid = 47)

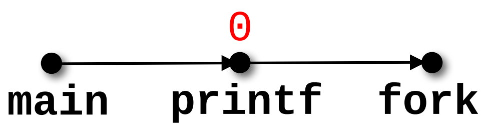
execve Example

```c
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

```c
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c



Parent (pid = 47)

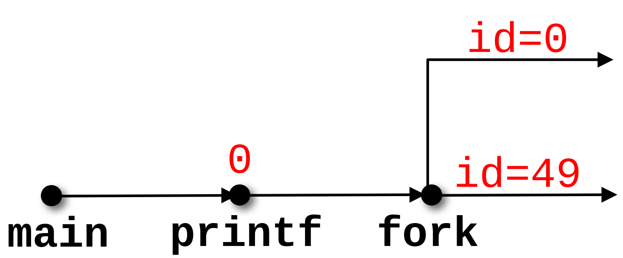
execve Example

```c
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

```c
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c



Child (pid = 49)

Parent (pid = 47)

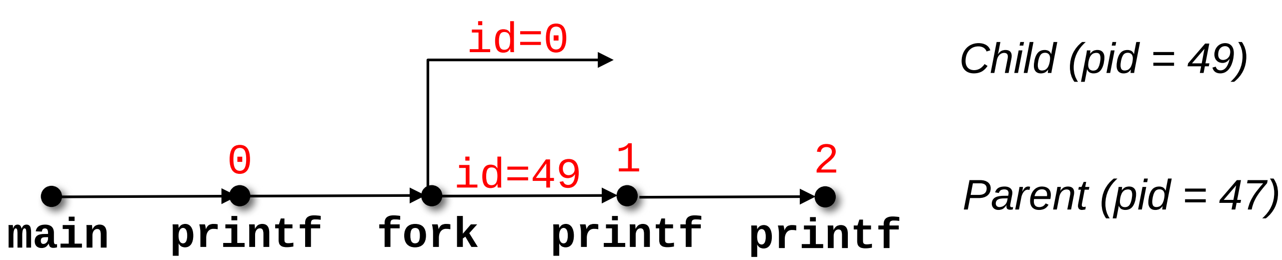
execve Example

```c
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

```c
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c



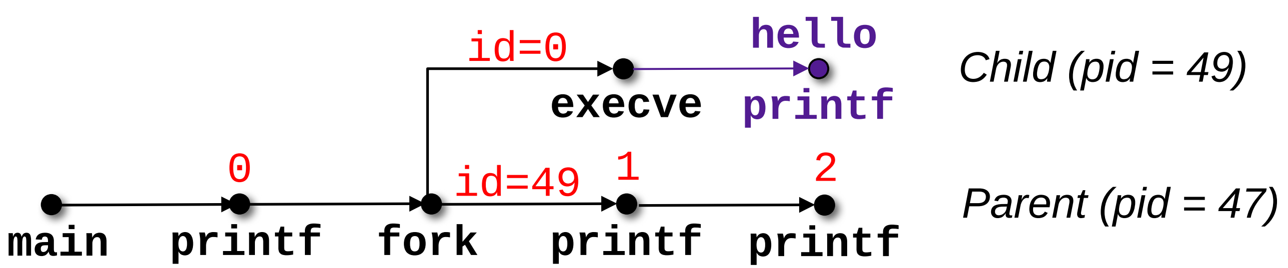
execve Example

```c
int main(int argc, char** argv){

  printf("0\n");
  pid_t id = fork();

  if(id == 0){ // if child
    execve("hello", NULL, NULL);
  } else { // if parent
    printf("1\n");
  }

  printf("2\n");
  return 0;
}
```

exec.c

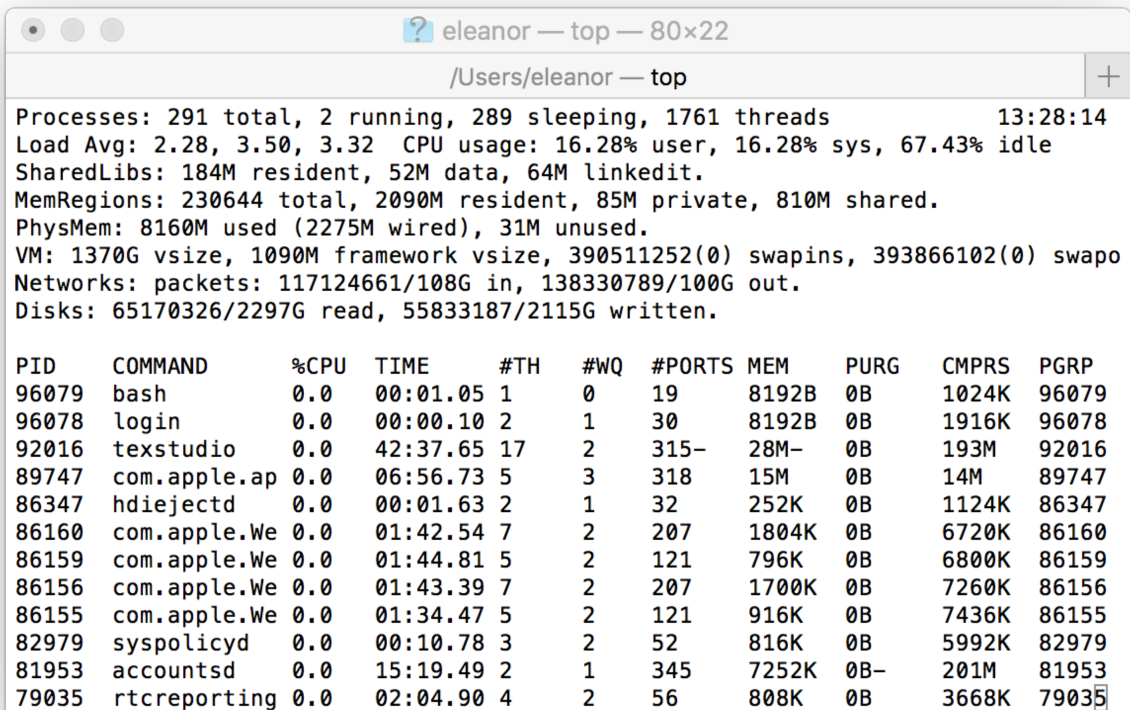
```c
int main(int argc, char** argv){
  printf("Hello!\n");

  return 0;
}
```

hello.c



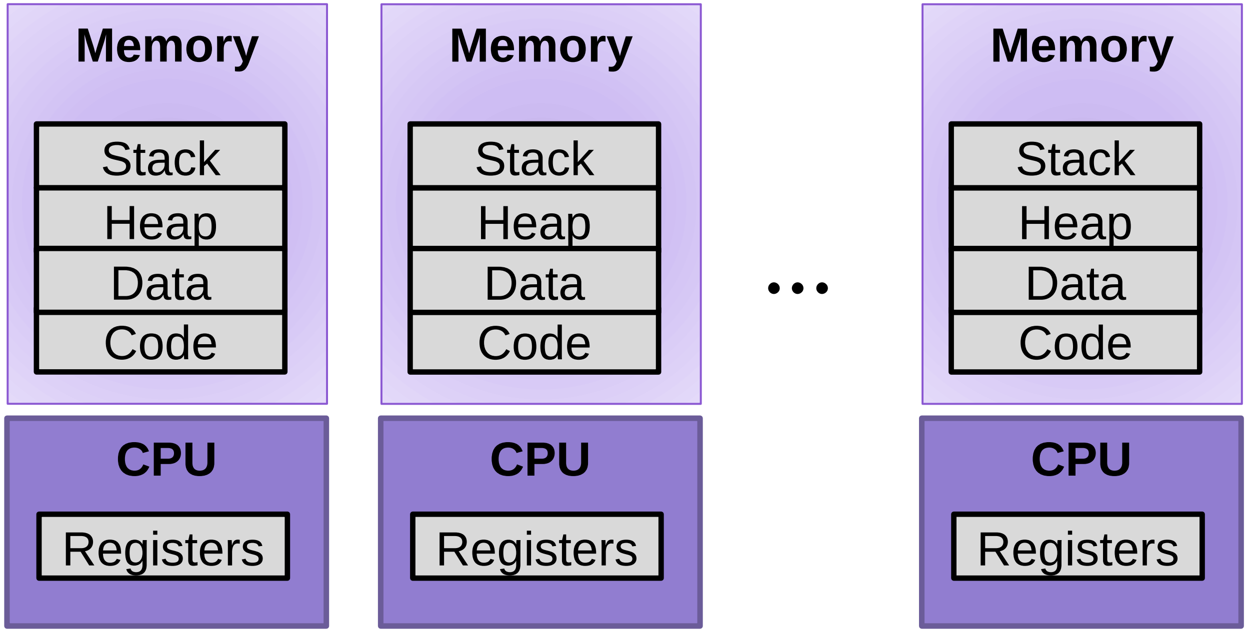
Multiprocessing

- Computer runs many processes simultaneously
- Running program “top” on Mac
  - Identified by Process ID (PID)



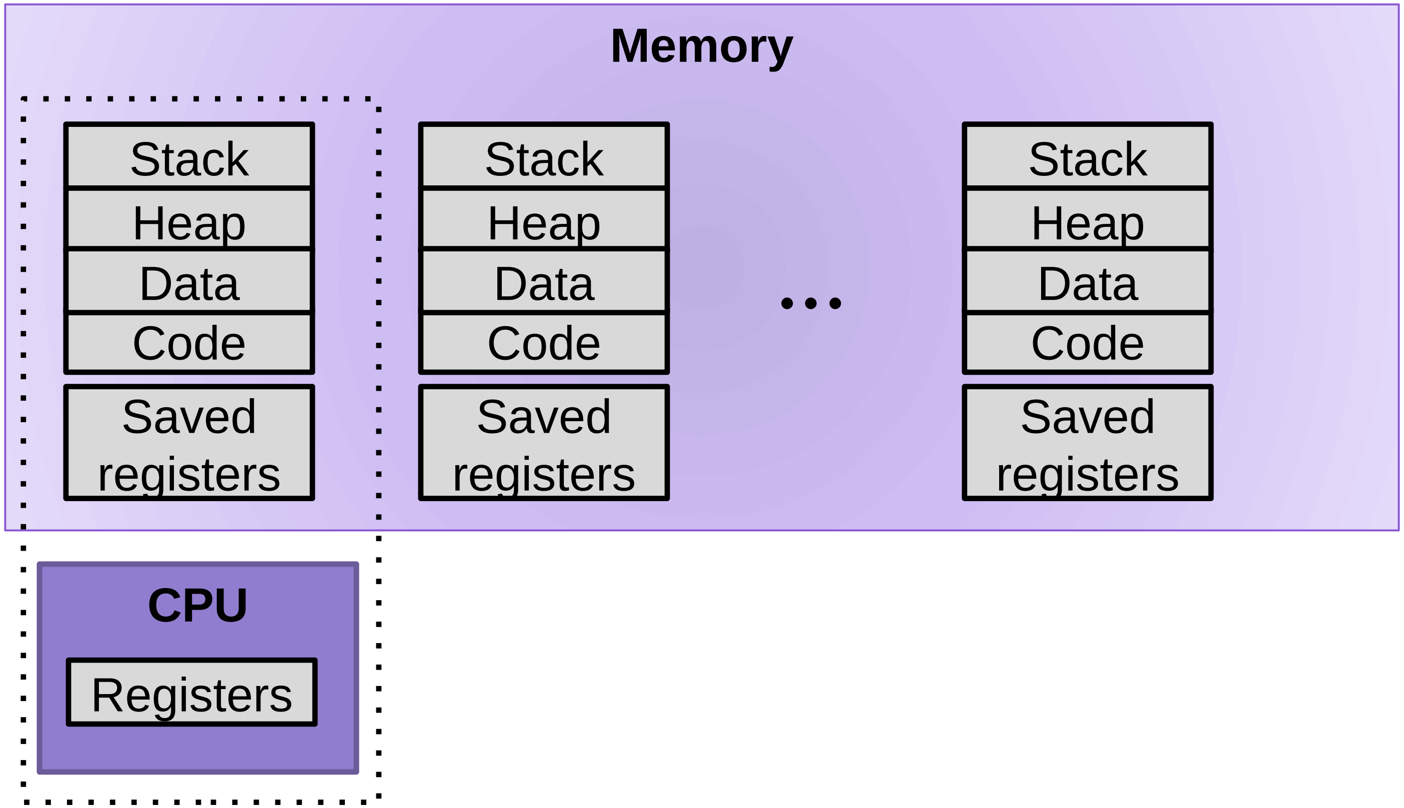

```
Processes: 291 total, 2 running, 289 sleeping, 1761 threads            13:28:14
Load Avg: 2.28, 3.50, 3.32  CPU usage: 16.28% user, 16.28% sys, 67.43% idle
SharedLibs: 184M resident, 52M data, 64M linkedit.
MemRegions: 230644 total, 2090M resident, 85M private, 810M shared.
PhysMem: 8160M used (2275M wired), 31M unused.
VM: 1370G vsize, 1090M framework vsize, 390511252(0) swapins, 393866102(0) swapo
Networks: packets: 117124661/108G in, 138330789/100G out.
Disks: 65170326/2297G read, 55833187/2115G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORTS MEM    PURG   CMPRS  PGRP
96079  bash         0.0  00:01.05 1     0    19     8192B  0B     1024K  96079
96078  login        0.0  00:00.10 2     1    30     8192B  0B     1916K  96078
92016  texstudio    0.0  42:37.65 17    2    315-   28M-   0B     193M   92016
89747  com.apple.ap 0.0  06:56.73 5     3    318    15M    0B     14M    89747
86347  hdiejectd    0.0  00:01.63 2     1    32     252K   0B     1124K  86347
86160  com.apple.We 0.0  01:42.54 7     2    207    1804K  0B     6720K  86160
86159  com.apple.We 0.0  01:44.81 5     2    121    796K   0B     6800K  86159
86156  com.apple.We 0.0  01:43.39 7     2    207    1700K  0B     7260K  86156
86155  com.apple.We 0.0  01:34.47 5     2    121    916K   0B     7436K  86155
82979  syspolicyd   0.0  00:10.78 3     2    52     816K   0B     5992K  82979
81953  accountsd    0.0  15:19.49 2     1    345    7252K  0B-    201M   81953
79035  rtcreporting 0.0  02:04.90 4     2    56     808K   0B     3668K  79035
```

Multiprocessing: The Illusion



- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called ***context switching***
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called ***virtual memory***

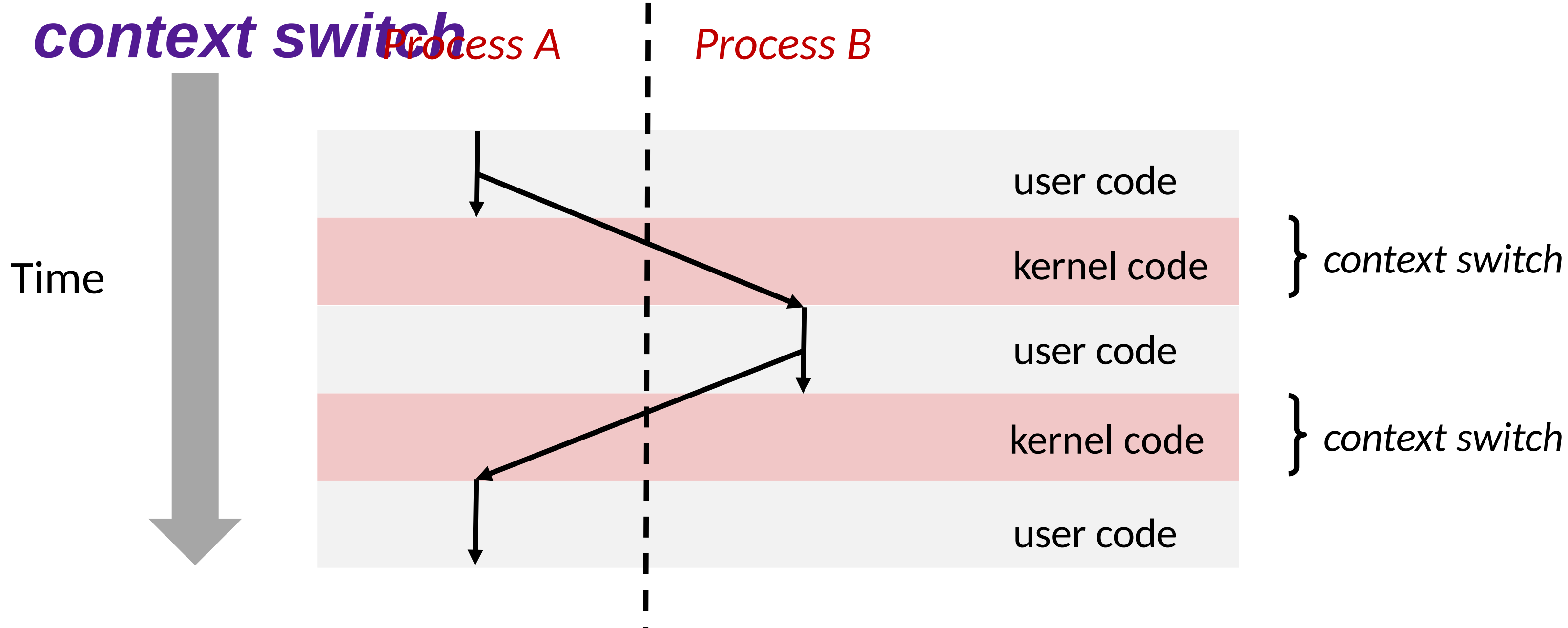
Multiprocessing: The (Traditional) Reality



- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Register values for nonexecuting processes saved in memory
  - Address spaces managed by virtual memory system

Context Switching

- Processes are managed by a shared chunk of memory-resident kernel code
  - Important: the kernel code is not a separate process, but rather code and data structures that the OS uses to manage all processes
- Control flow passes from one process to another via a ***context switch***

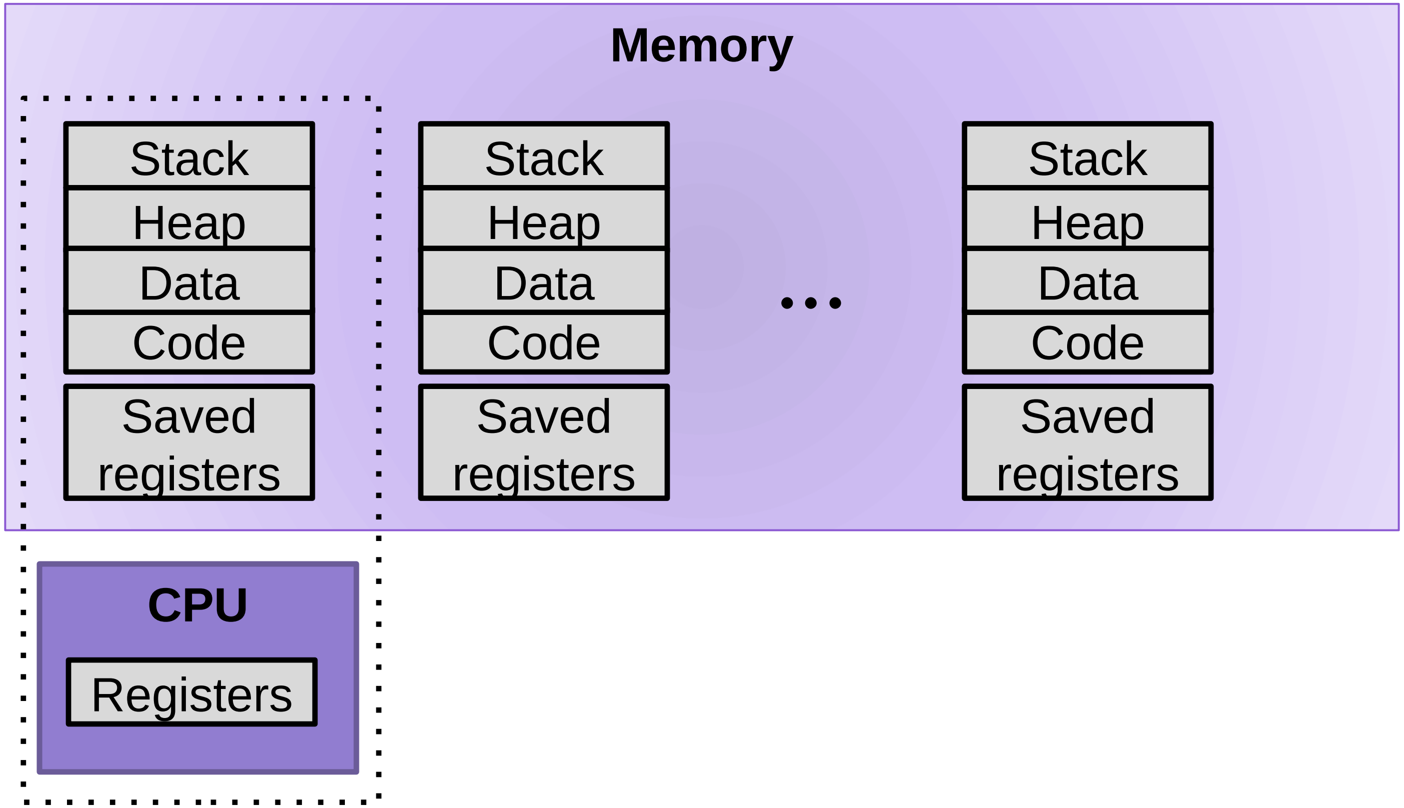


Process Control Block (PCB)

- To implement a context switch, OS maintains a PCB for each process containing:
  - process table, which contains information about the process (id, user, privilege level, arguments, status)
  - location of executable on disk
  - file table
  - register values (general-purpose registers, float registers, pc, eflags…)
  - memory state
  - scheduling information

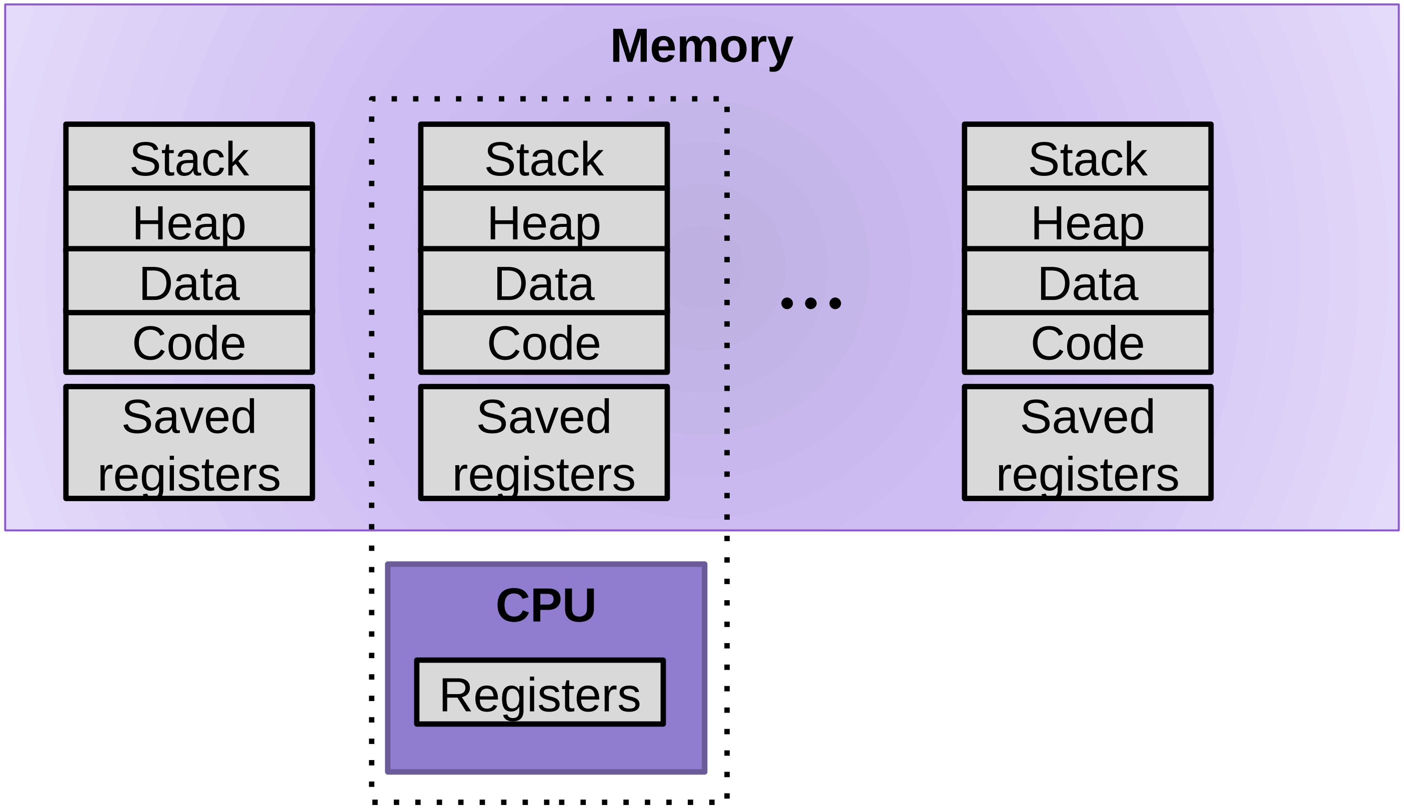
... and more!

Multiprocessing: The (Traditional) Reality



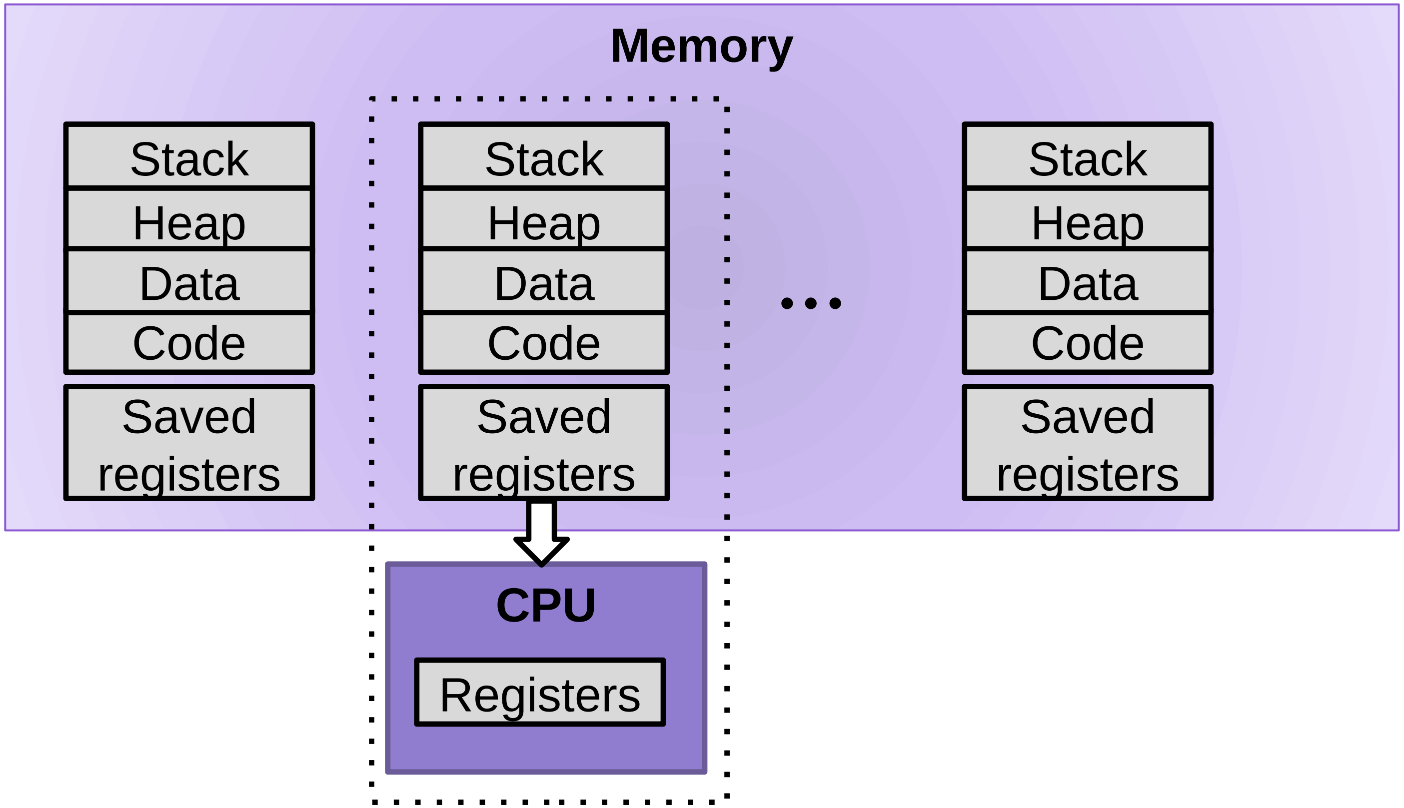
1. Save current registers to memory (in PCB)

Multiprocessing: The (Traditional) Reality



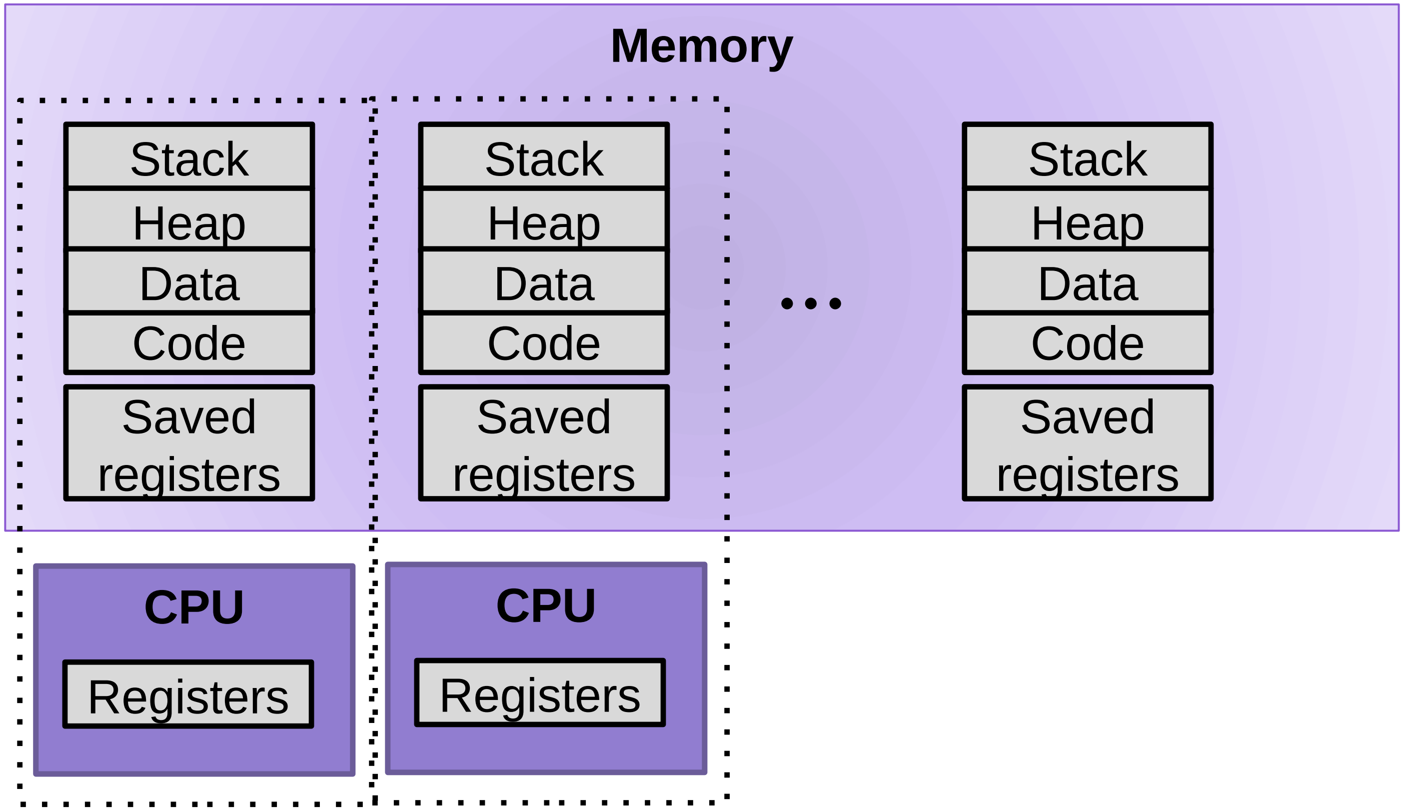
1. Save current registers to memory (in PCB)
2. Schedule next process for execution

Multiprocessing: The (Traditional) Reality



1. Save current registers to memory (in PCB)
2. Schedule next process for execution
3. Load saved registers and switch address space

Multiprocessing: The (Modern) Reality



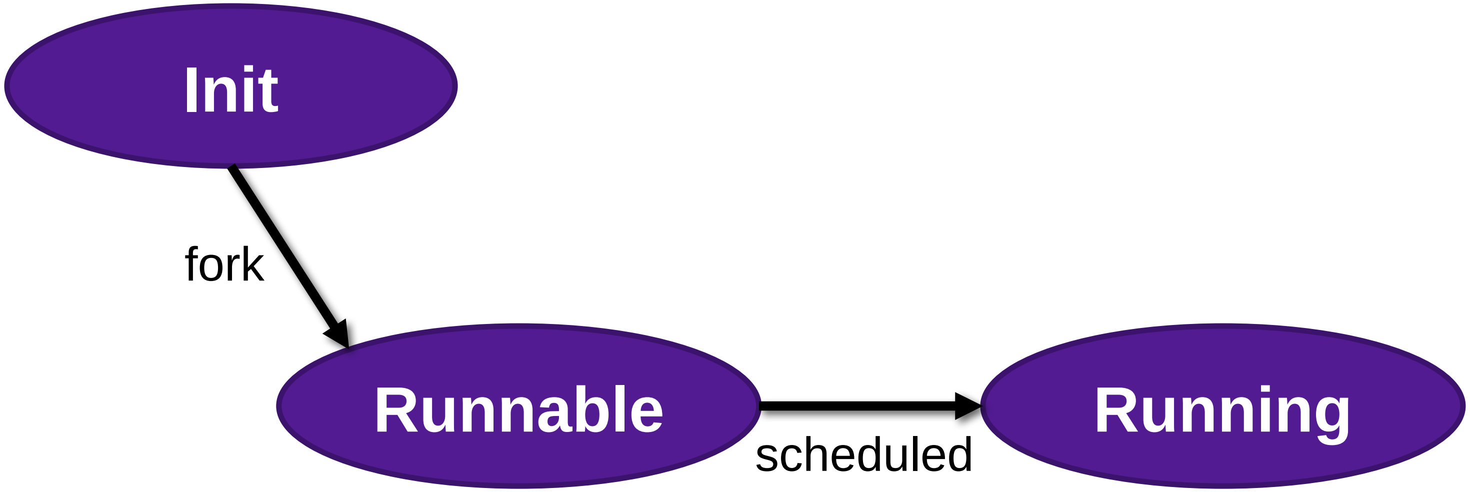
- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Exercise: Context Switching

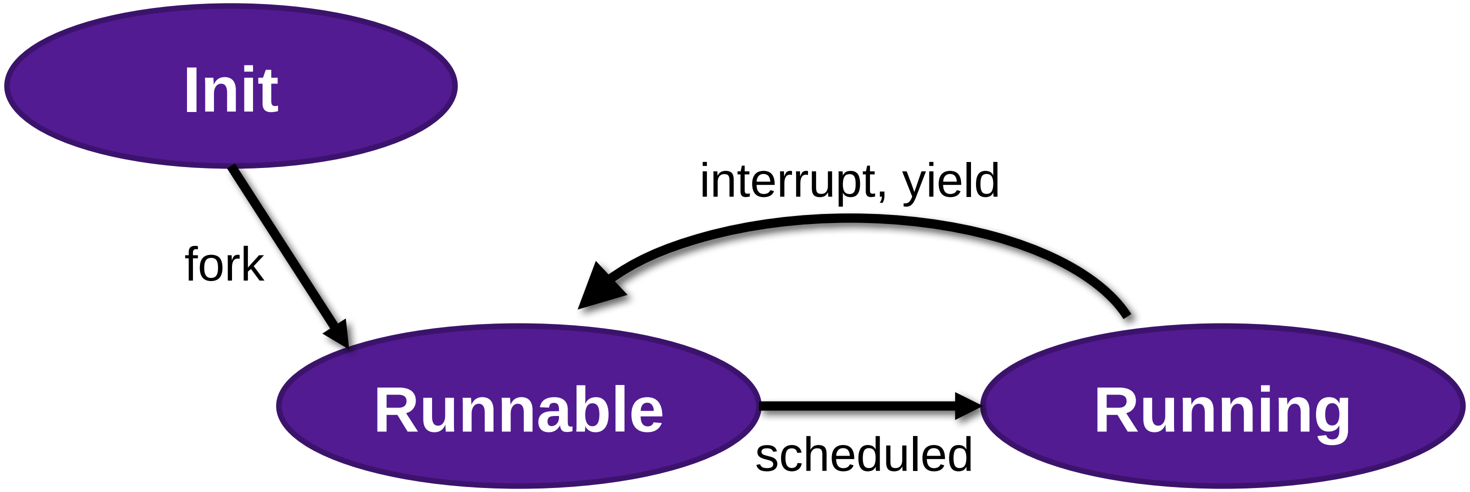
A hardware designer argues that there are now enough on-chip transistors to build a CPU with 1024 integer registers and 512 floating point registers. As a result, the compiler should almost never need to store anything on the stack. As a new operating systems expert, would you recommend building this new design.

Process Life Cycle

Process Life Cycle

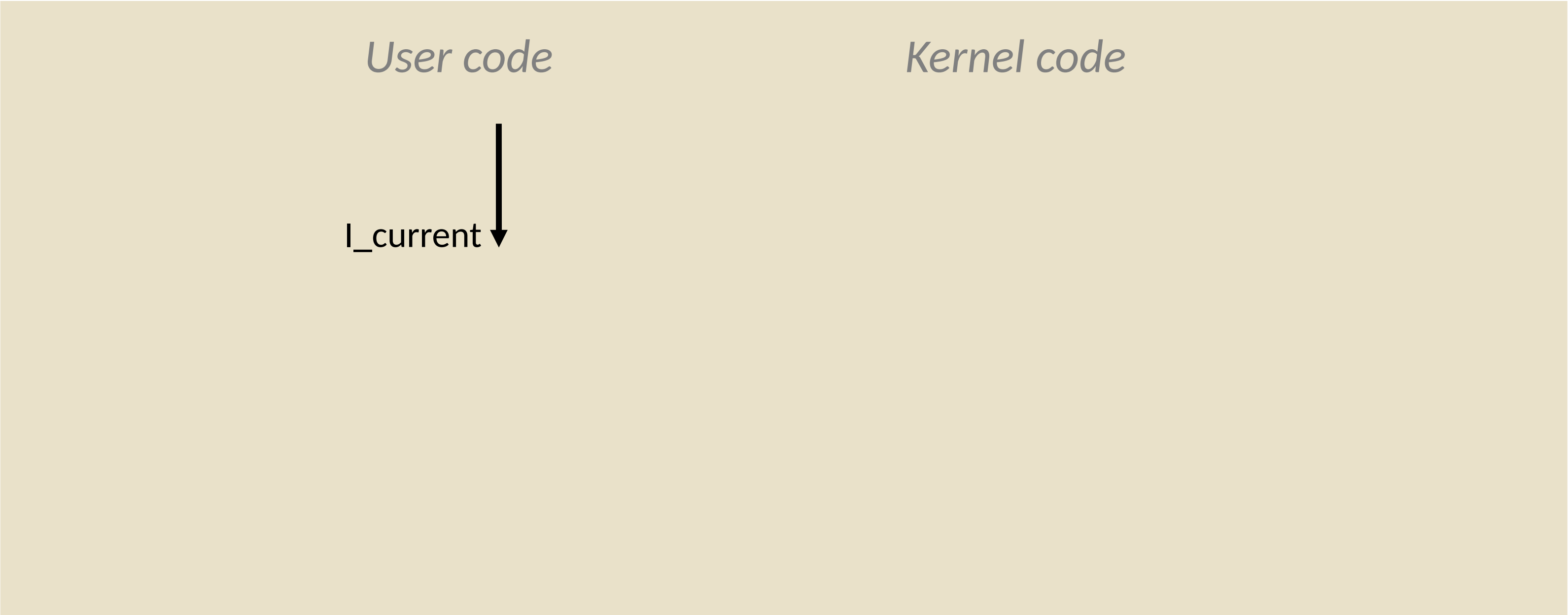


Process Life Cycle



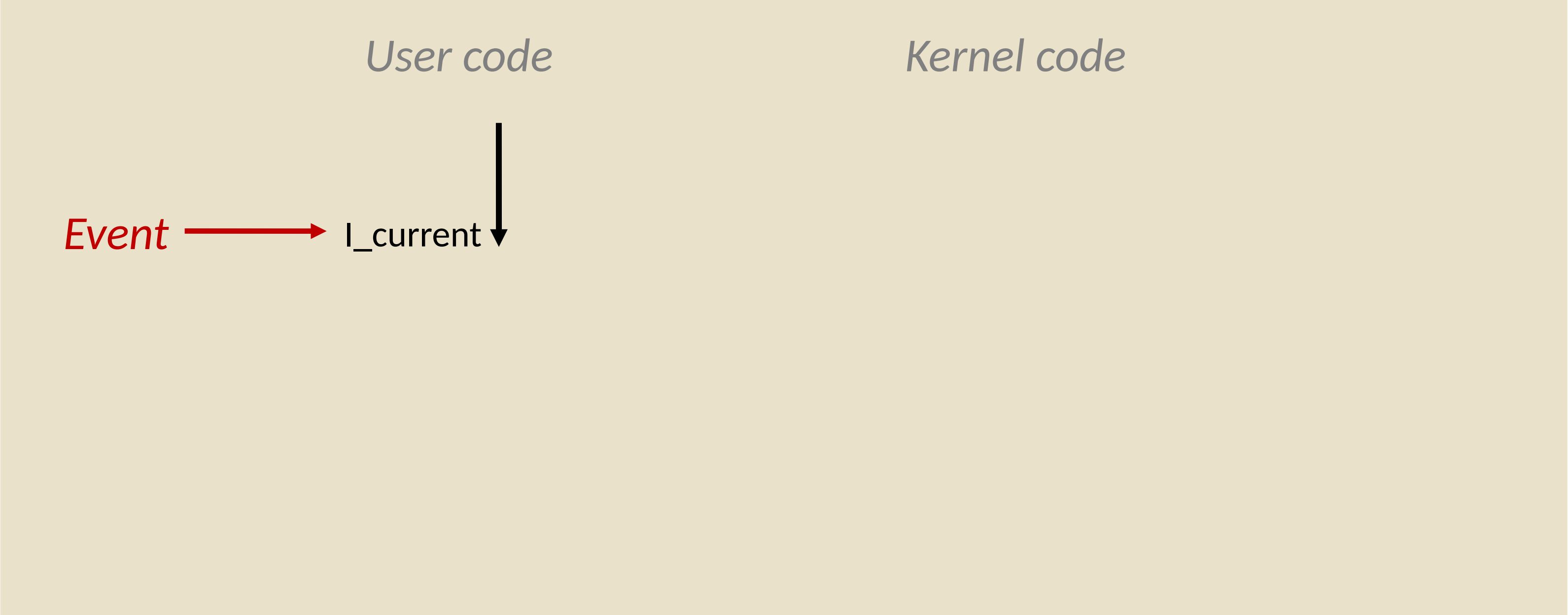
Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)



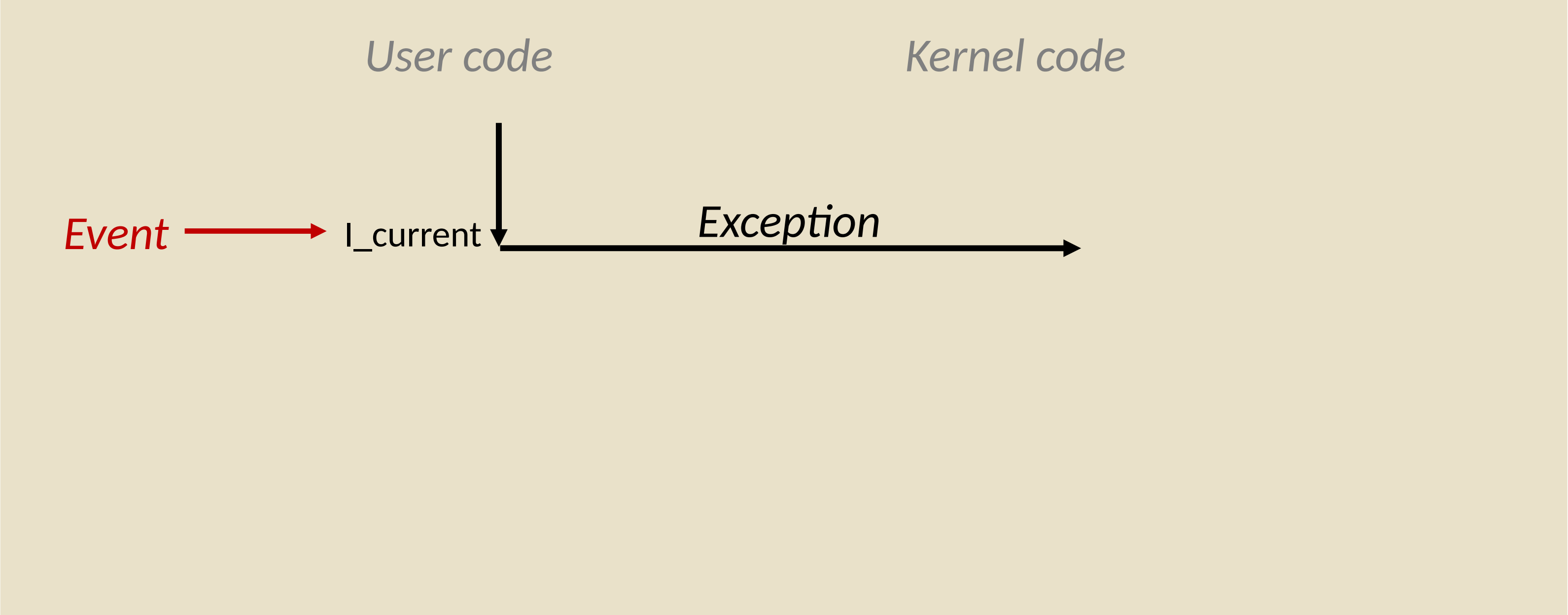
Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)



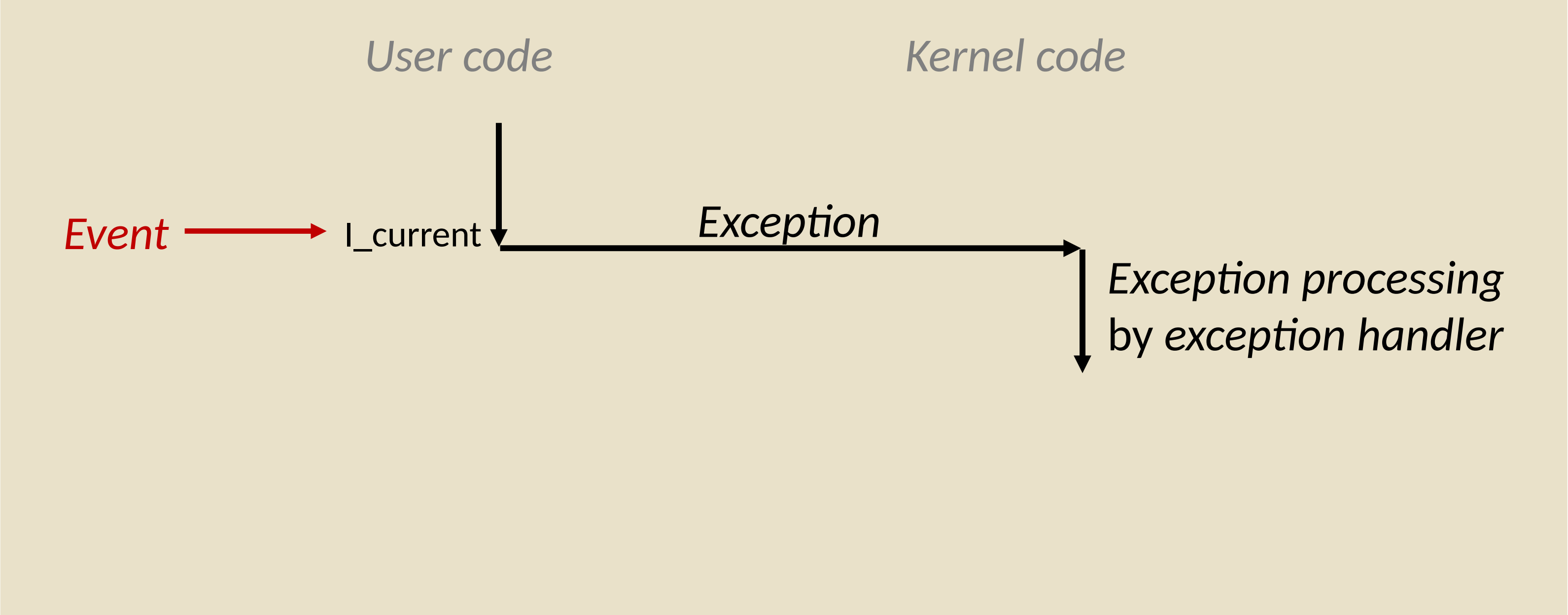
Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)



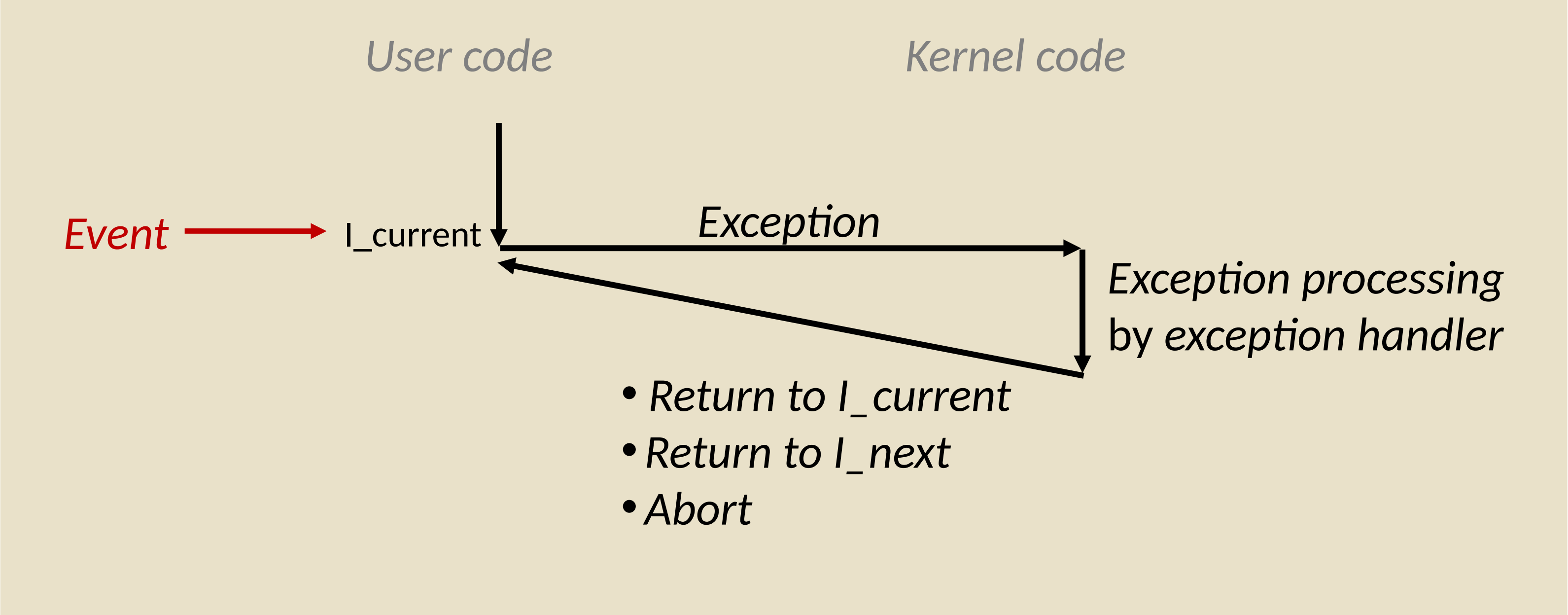
Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)



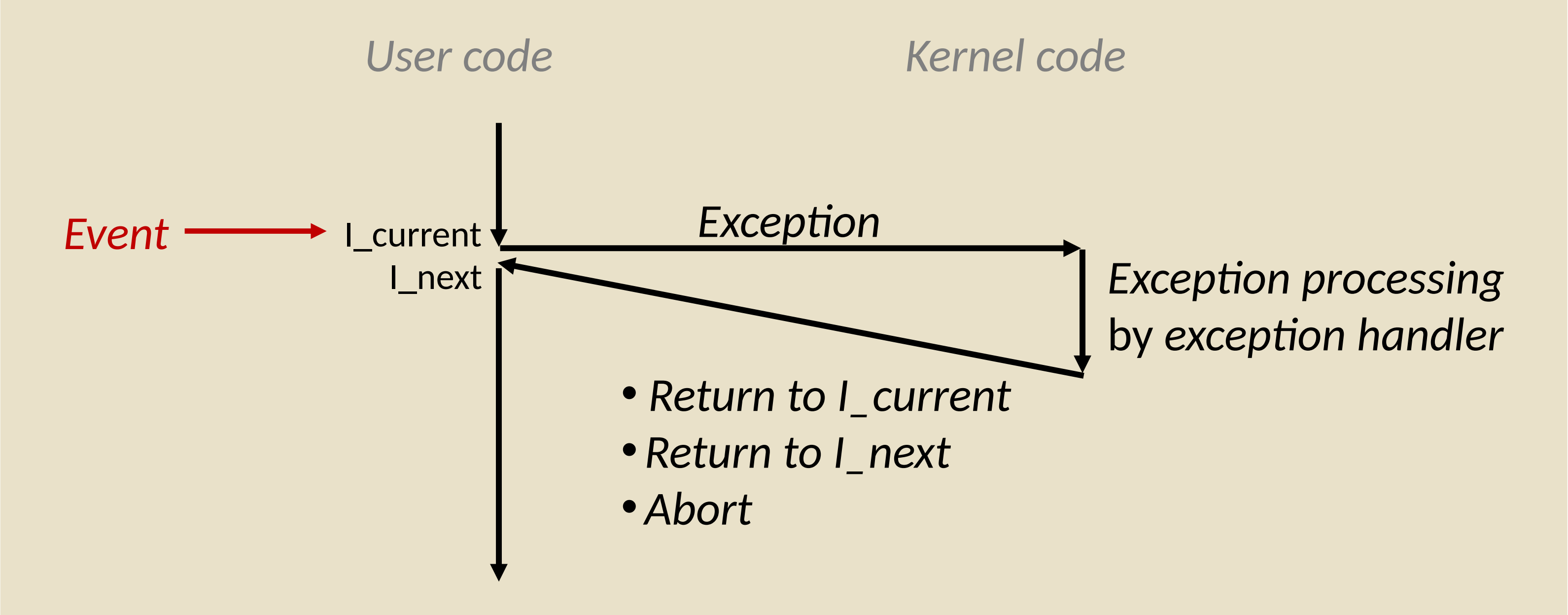
Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)

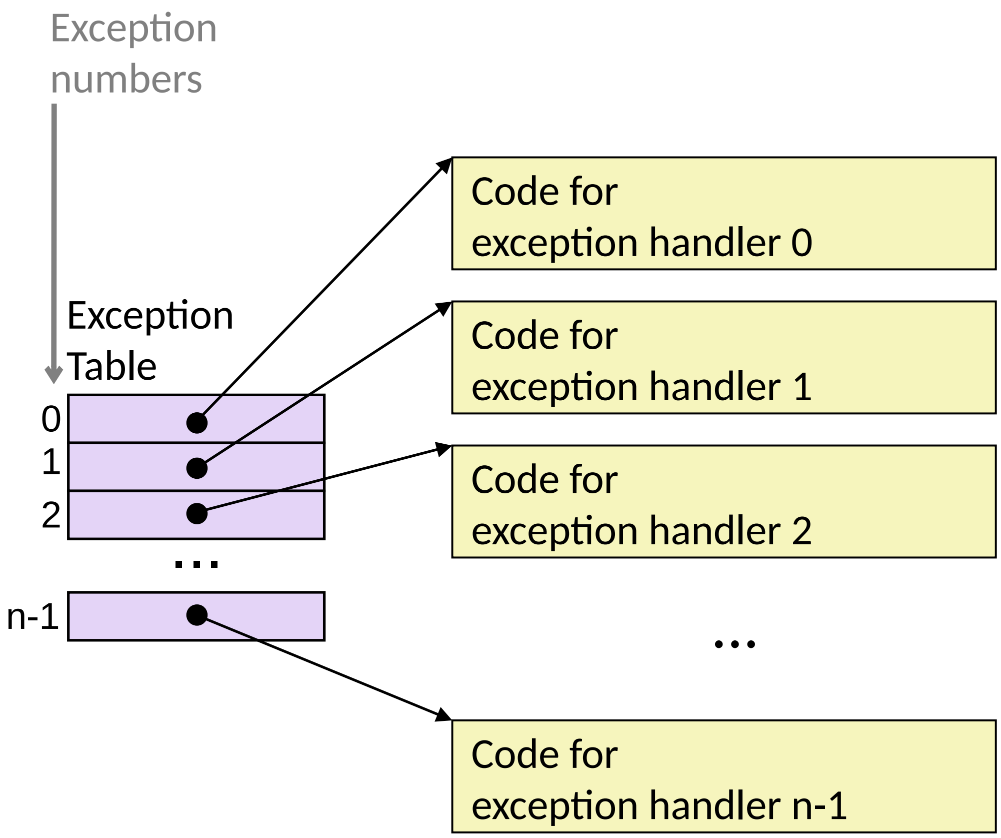


Exceptions

- An **exception** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

- ***Traps***
  - Intentional
  - Examples: ***system calls***, breakpoint traps, special instructions
  - Returns control to “next” instruction

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

- ***Traps***
  - Intentional
  - Examples: ***system calls***, breakpoint traps, special instructions
  - Returns control to “next” instruction
- ***Faults***
  - Unintentional but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
  - Either re-executes faulting (“current”) instruction or aborts

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:

- ***Traps***
  - Intentional
  - Examples: ***system calls***, breakpoint traps, special instructions
  - Returns control to "next" instruction
- ***Faults***
  - Unintentional but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
  - Either re-executes faulting ("current") instruction or aborts
- ***Aborts***
  - Unintentional and unrecoverable
  - Examples: illegal instruction, divide-by-zero, parity error, machine check
  - Aborts current program

Interrupts (Asynchronous Exceptions)

Caused by events external to the process

- Indicated by setting the processor's *interrupt pin*
- Handler returns to "next" instruction

Interrupts (Asynchronous Exceptions)

Caused by events external to the process

- Indicated by setting the processor’s *interrupt pin*
- Handler returns to “next” instruction

Examples:

- Timer interrupt
  - Every few ms, an external timer chip triggers an interrupt
  - Used by the kernel to take back control from user programs

Interrupts (Asynchronous Exceptions)

Caused by events external to the process

- Indicated by setting the processor's *interrupt pin*
- Handler returns to "next" instruction

Examples:

- Timer interrupt
  - Every few ms, an external timer chip triggers an interrupt
  - Used by the kernel to take back control from user programs
- I/O interrupt from external device
  - Hitting Ctrl-C at the keyboard
  - Arrival of a packet from a network
  - Arrival of data from a disk

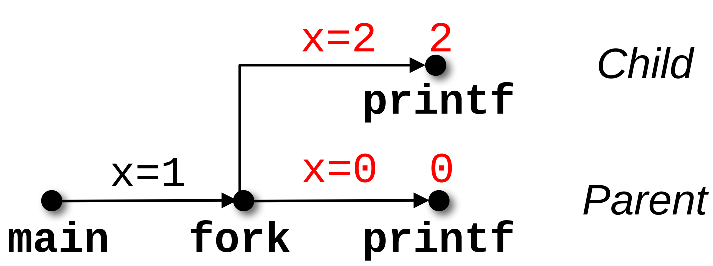
fork Example

```
int main(){

    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

fork Example

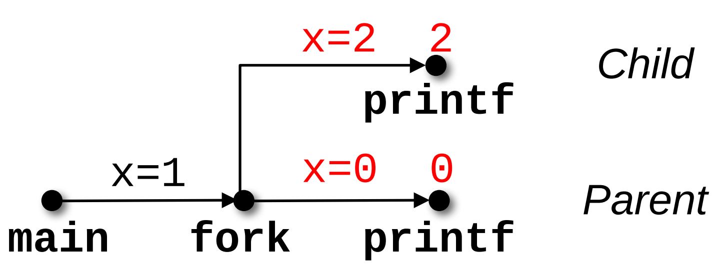
```
int main(){

    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - stdout is the same in both parent and child

fork Example

```
int main(){

    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

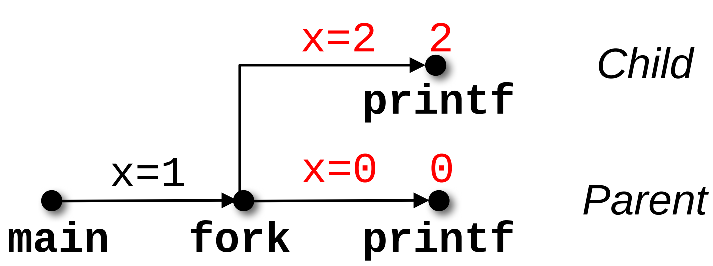


- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - `stdout` is the same in both parent and child
- **Concurrent execution**
  - Can't predict execution order of parent and child

fork Example

```
int main(){

    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

- **Call once, return twice**
- **Duplicate but separate address space**
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- **Shared open files**
  - stdout is the same in both parent and child
- **Concurrent execution**
  - Can't predict execution order of parent and child



Exercise: What are all the possible outputs of this program?

Modeling `fork` with Process Graphs

- A **process graph** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - a -> b means a happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

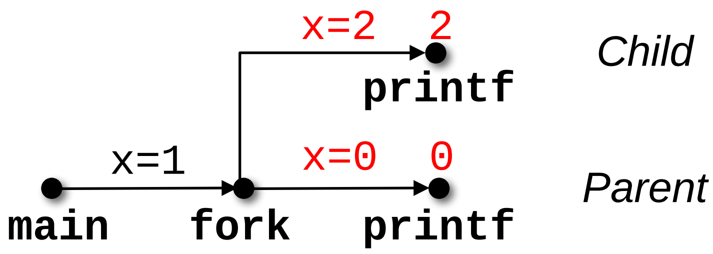
Modeling `fork` with Process Graphs

- A **process graph** is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - a -> b means a happens before b
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

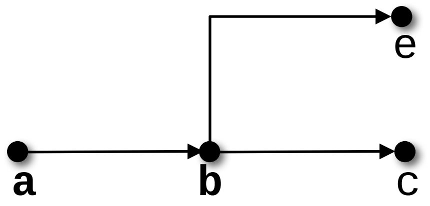
- Any topological sort of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right

Interpreting Process Graphs

- Original graph:

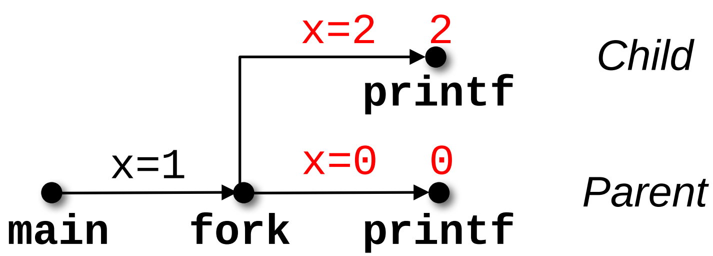


- Relabeled graph:

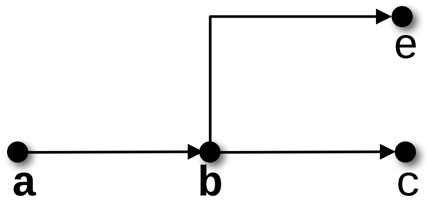


Interpreting Process Graphs

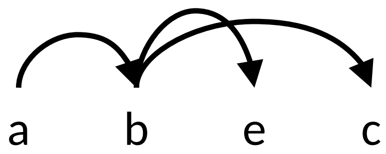
- Original graph:



- Relabeled graph:

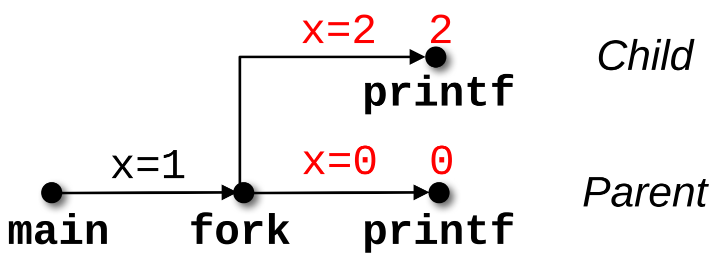


Feasible total ordering:

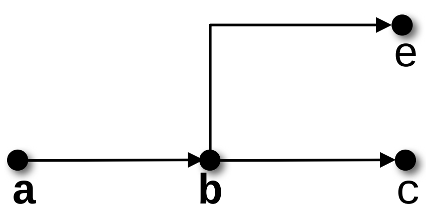


Interpreting Process Graphs

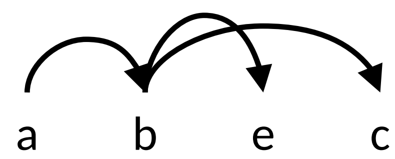
- Original graph:



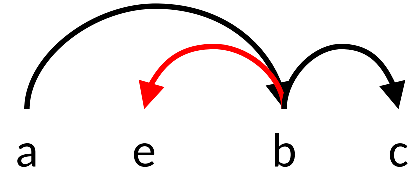
- Relabeled graph:



Feasible total ordering:



Infeasible total ordering:



fork Example: Two consecutive forks

```
void fork1()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

`fork` Example: Two consecutive `fork`s

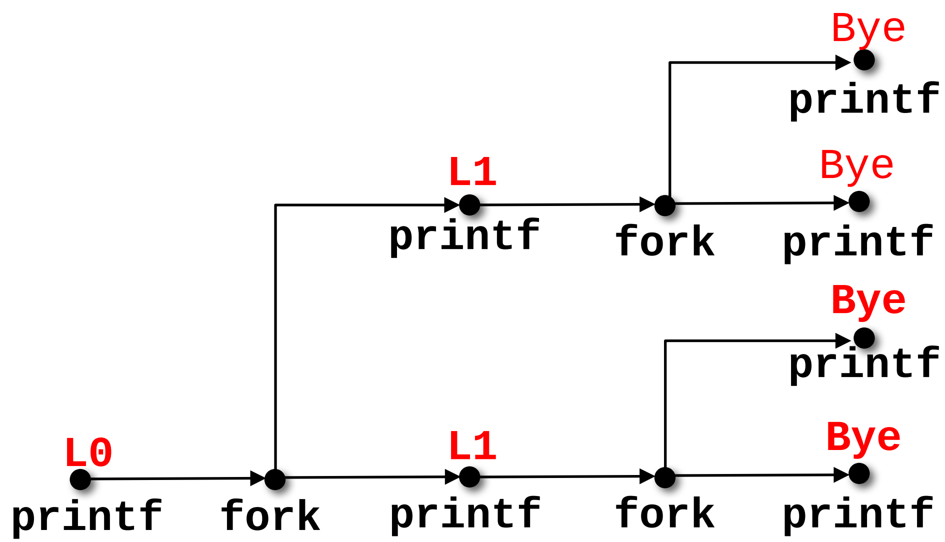
```
void fork1()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Which of these outputs are feasible?

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

`fork` Example: Two consecutive `forks`

```
void fork1()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Which of these outputs are feasible?

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

Exercise: Forks and Feasible Schedules

- For each of the following programs, draw the process graph and then determine which of the possible outputs are feasible

Exercise: Forks and Feasible Schedules

- For each of the following programs, draw the process graph and then determine which of the possible outputs are feasible

```
void fork2(){
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

Exercise: Forks and Feasible Schedules

- For each of the following programs, draw the process graph and then determine which of the possible outputs are feasible

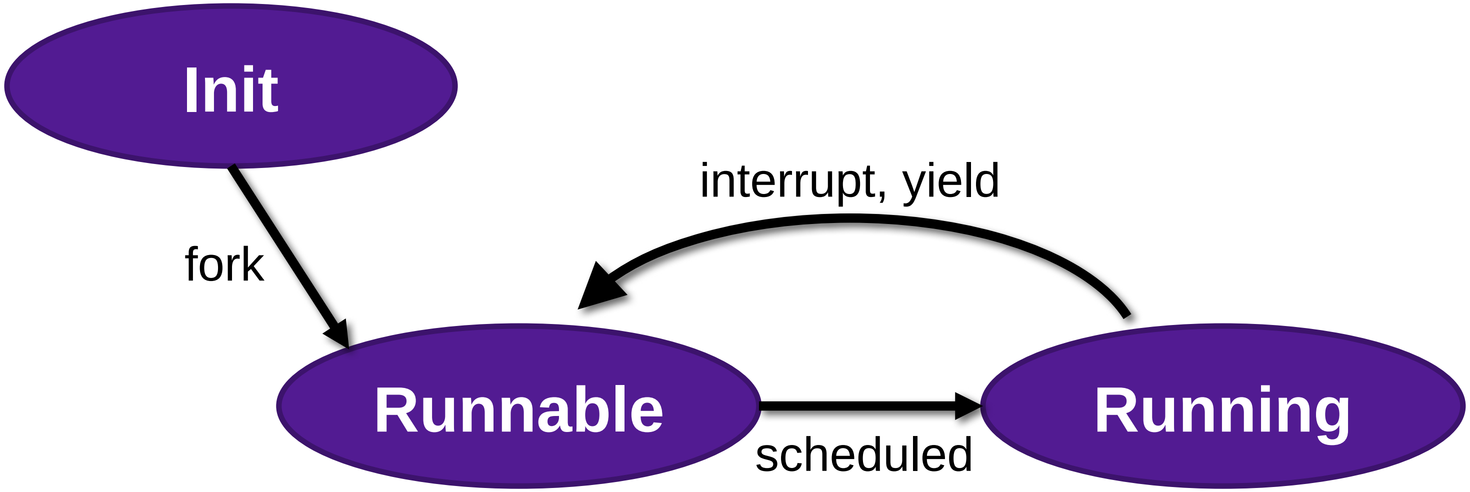
```
void fork2(){
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

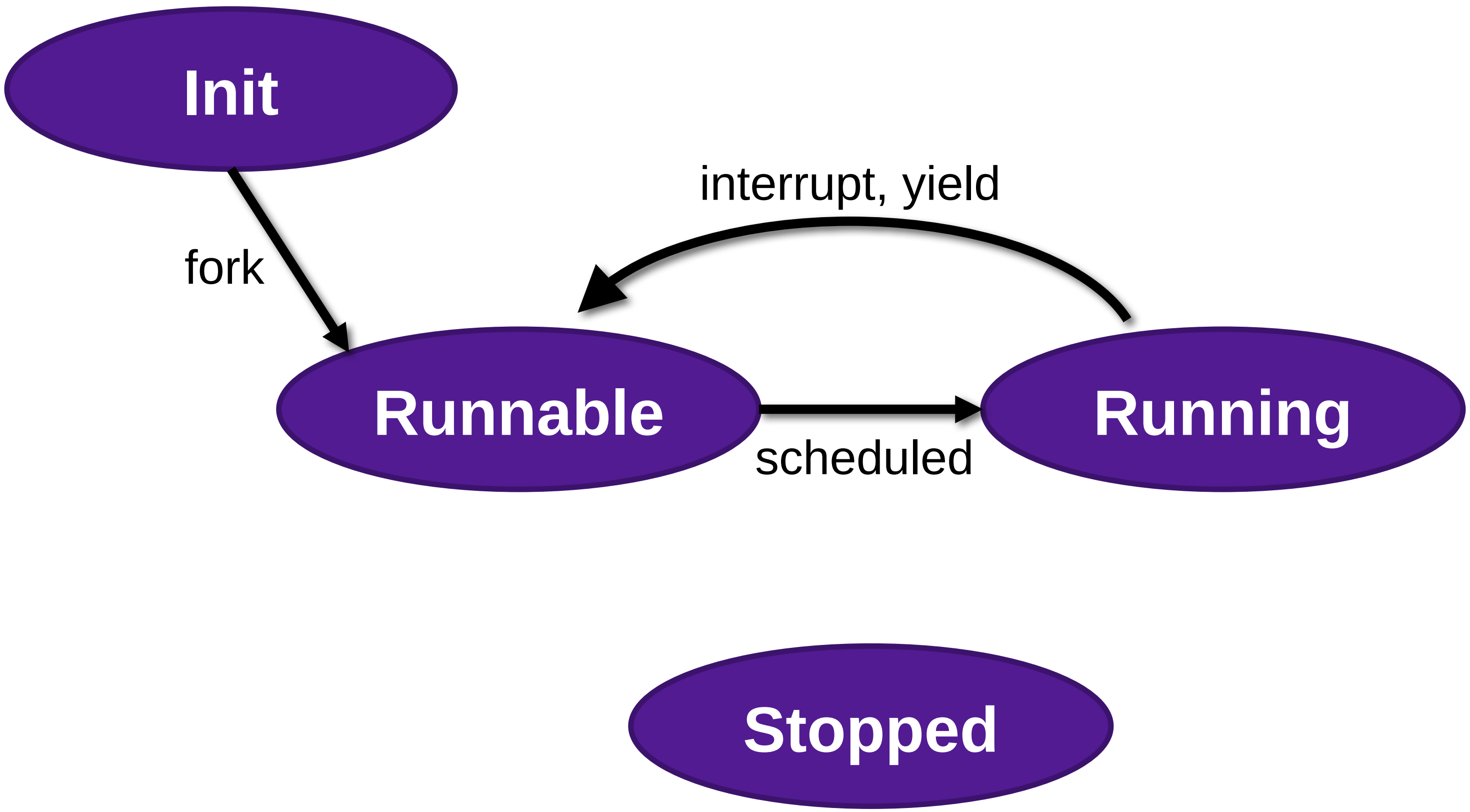
```
void fork3(){
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2

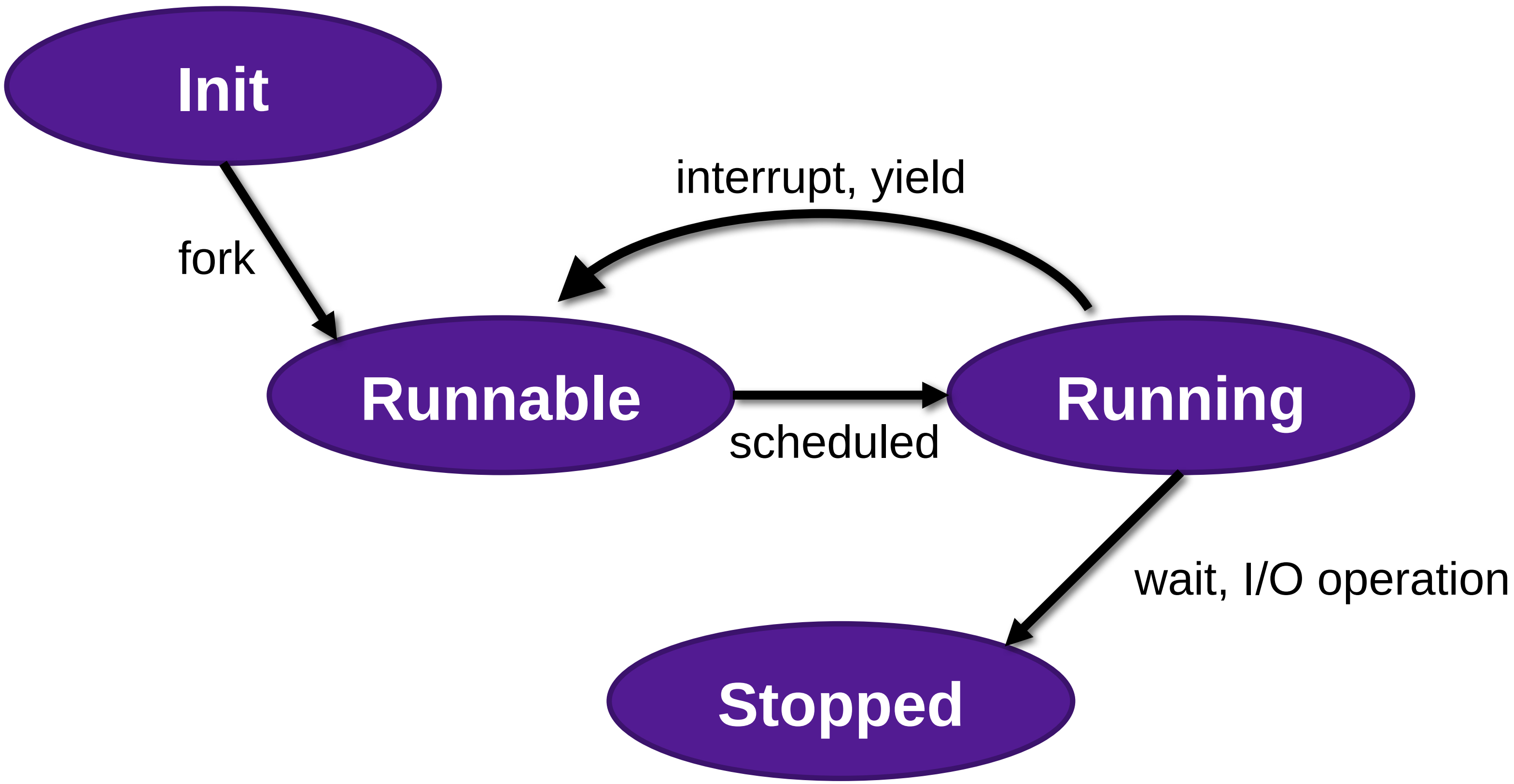
Process Life Cycle



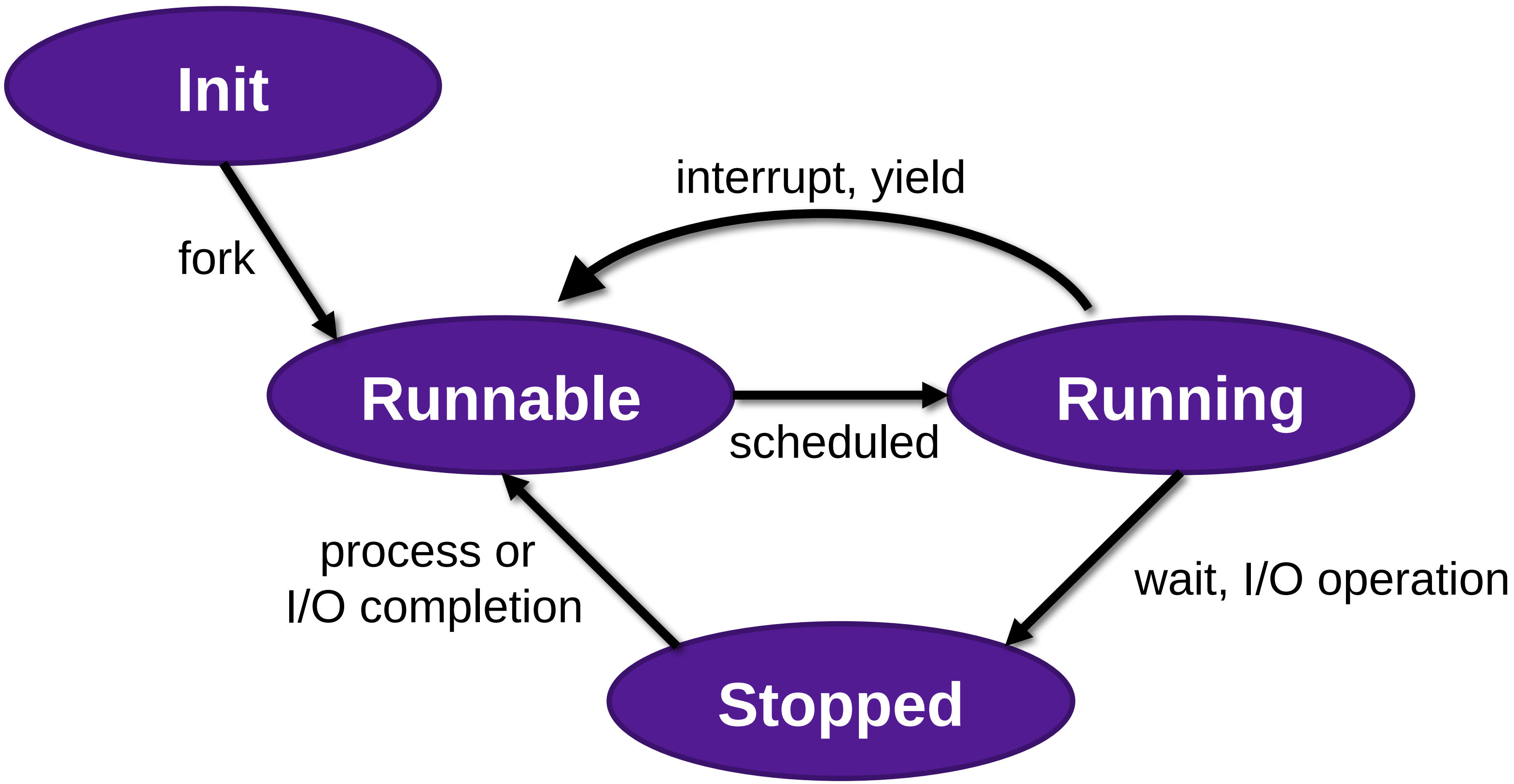
Process Life Cycle



Process Life Cycle



Process Life Cycle



Reaping Children

- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

- **`int wait(int* child_status)`**
  - Suspends current process until any one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status

Reaping Children

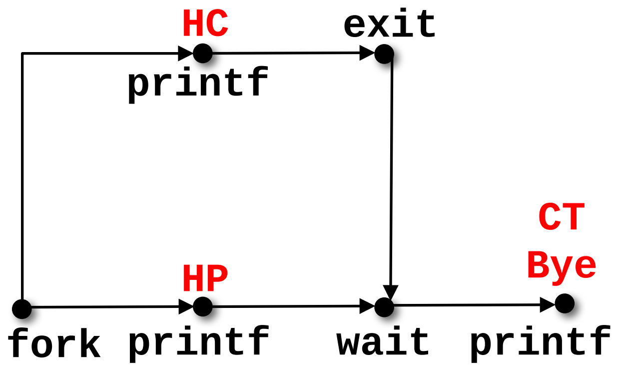
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

- **`int wait(int* child_status)`**
  - Suspends current process until any one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status

- **`int waitpid(pid_t pid, int* child_status, int opt)`**
  - Suspends current process child with pid terminates

wait Example

```
void fork6() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
                exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```



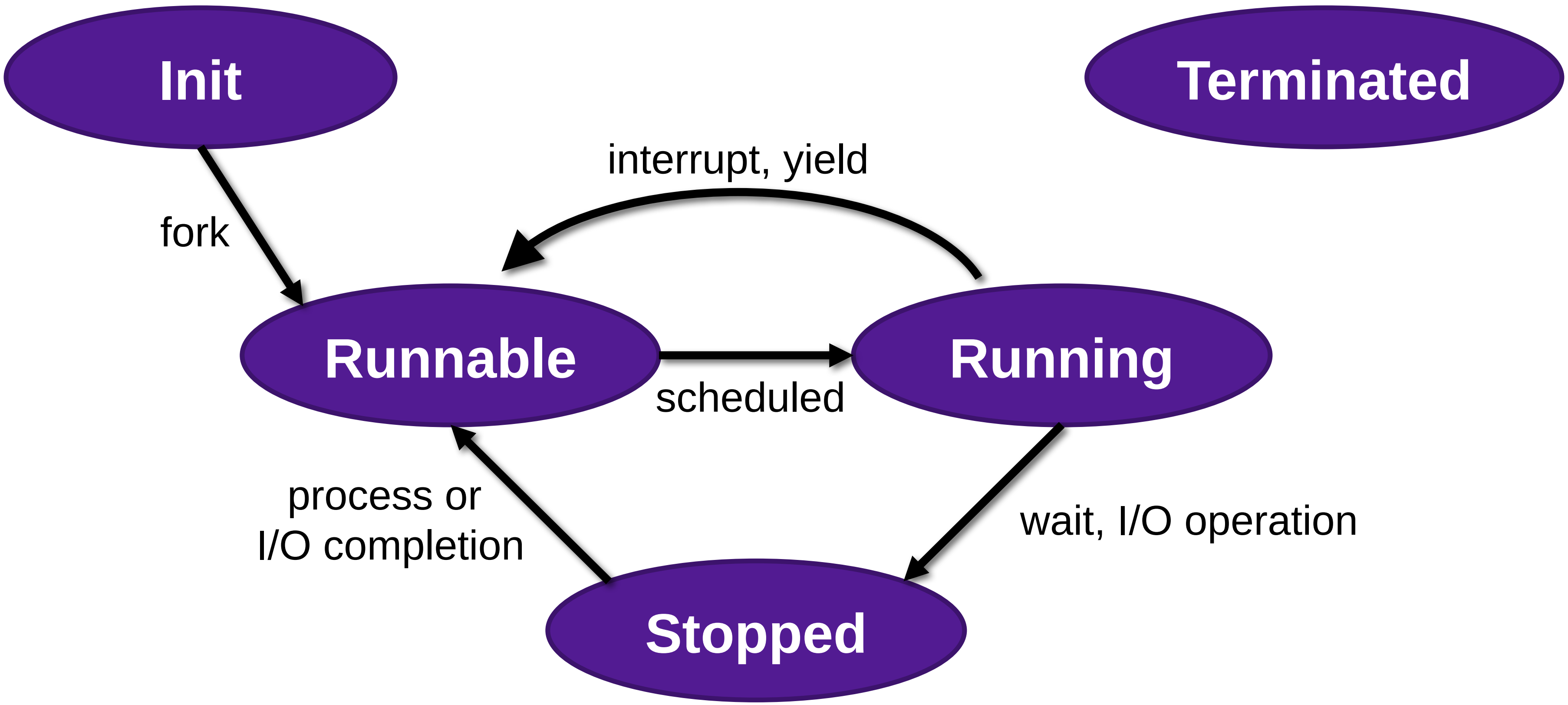
Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

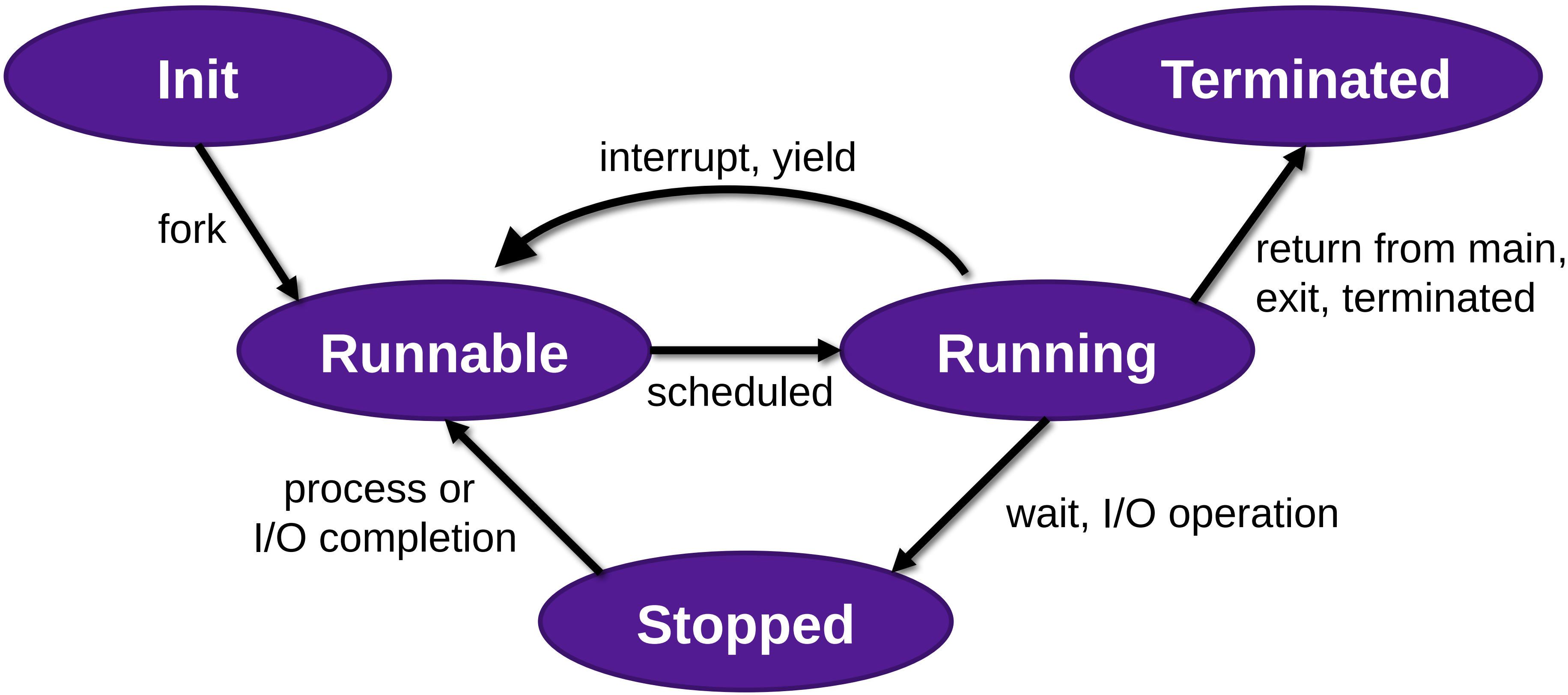
Reaping Children

- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Process Life Cycle



Process Life Cycle



Terminating Processes

- Process becomes terminated for one of three reasons:
  - Returning from the `main` routine
  - Calling the `exit` function
  - Receiving a signal whose default action is to terminate

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Returning from the `main` routine
  - Calling the `exit` function
  - Receiving a signal whose default action is to terminate

- `void exit(int status)`
  - Terminates with an **exit status** of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` is called **once** but **never** returns.