What is an Operating System?

Middleware between the HW and User/Program:

User
Program
Operating System
Computer Hardware

Manages the underlying HW

- Coordinates shared access to HW
- Efficiently schedules/manages HW resources

Provides easy-to-use interface to the HW

- Example: just type: `./a.out` to run a program

Kernel

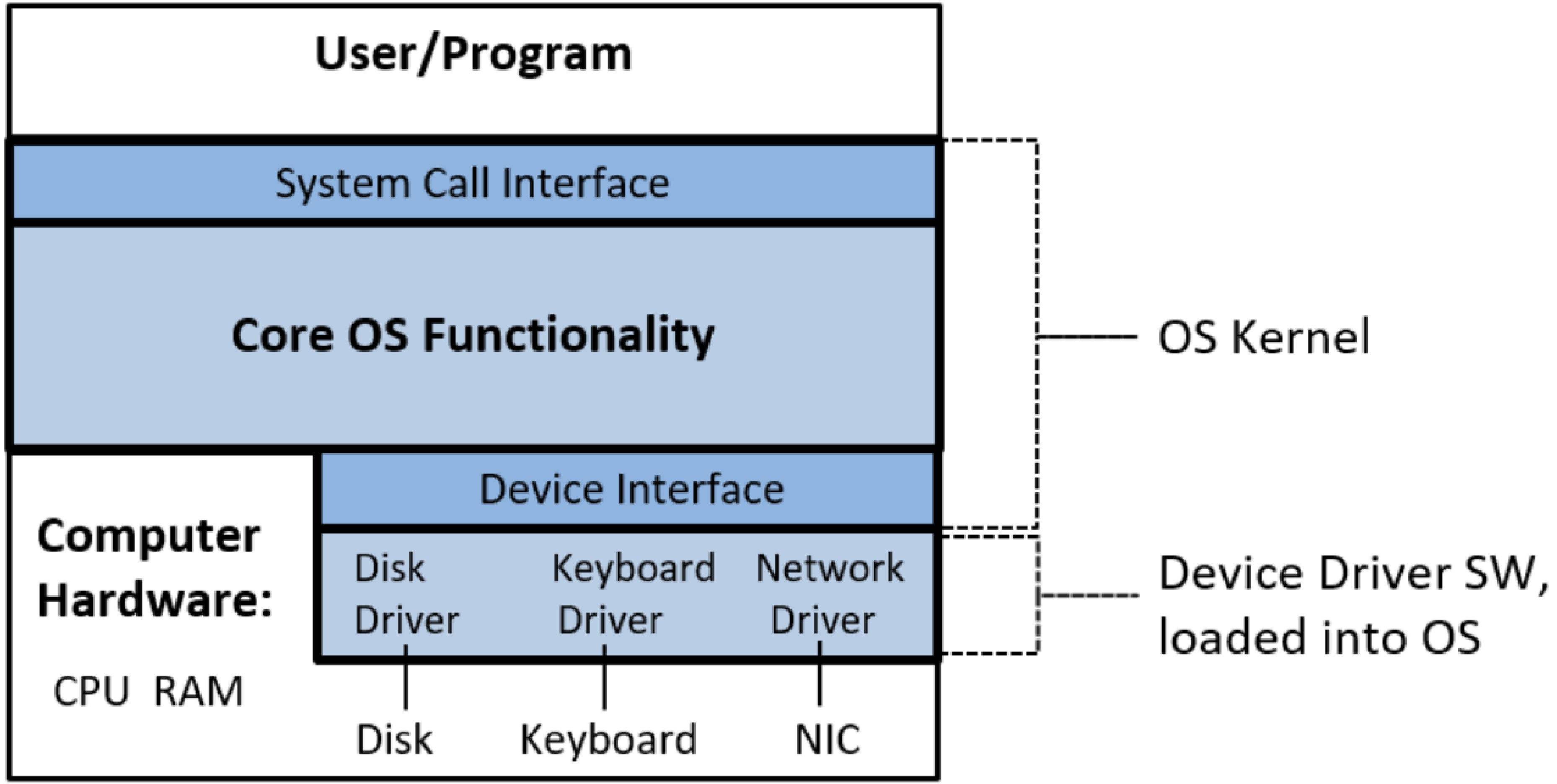
Implements core OS functionality

- Mechanisms for hardware to run programs (e.g. software, applications)
- Policies for efficiently managing and sharing resources

Implements the **system call interface**

- APIs for interacting with the hardware
- Examples: gettimeofday, open, fork

Kernel and Operating System



Operating System: Important Concepts

Process: abstraction that represents a running program

Virtual memory: abstraction that represents memory to a process

The Operating System

- It is software (not HW), but its *special* software

- When you start (e.g. boot up) your computer
  - “boot”, as in the computer “pulls itself up by its own bootstraps”

  - Your computer loads the OS as part of its ***boot sequence***
    - OS is a program, just like .\a.out, usually stored on disk
    - The OS is loaded by ***firmware***, e.g. permanent software located in read-only memory
      - BIOS (Basic Input-Output System): traditional firmware for booting the OS
      - UEFI (Unified Extensible Firmware Interface) : new firmware for booting the OS
    - After the OS is loaded, we can access and control the computer’s hardware
      - via system calls, like printf or open
      - via shell commands, like ls and cd

What happens when you run a program?

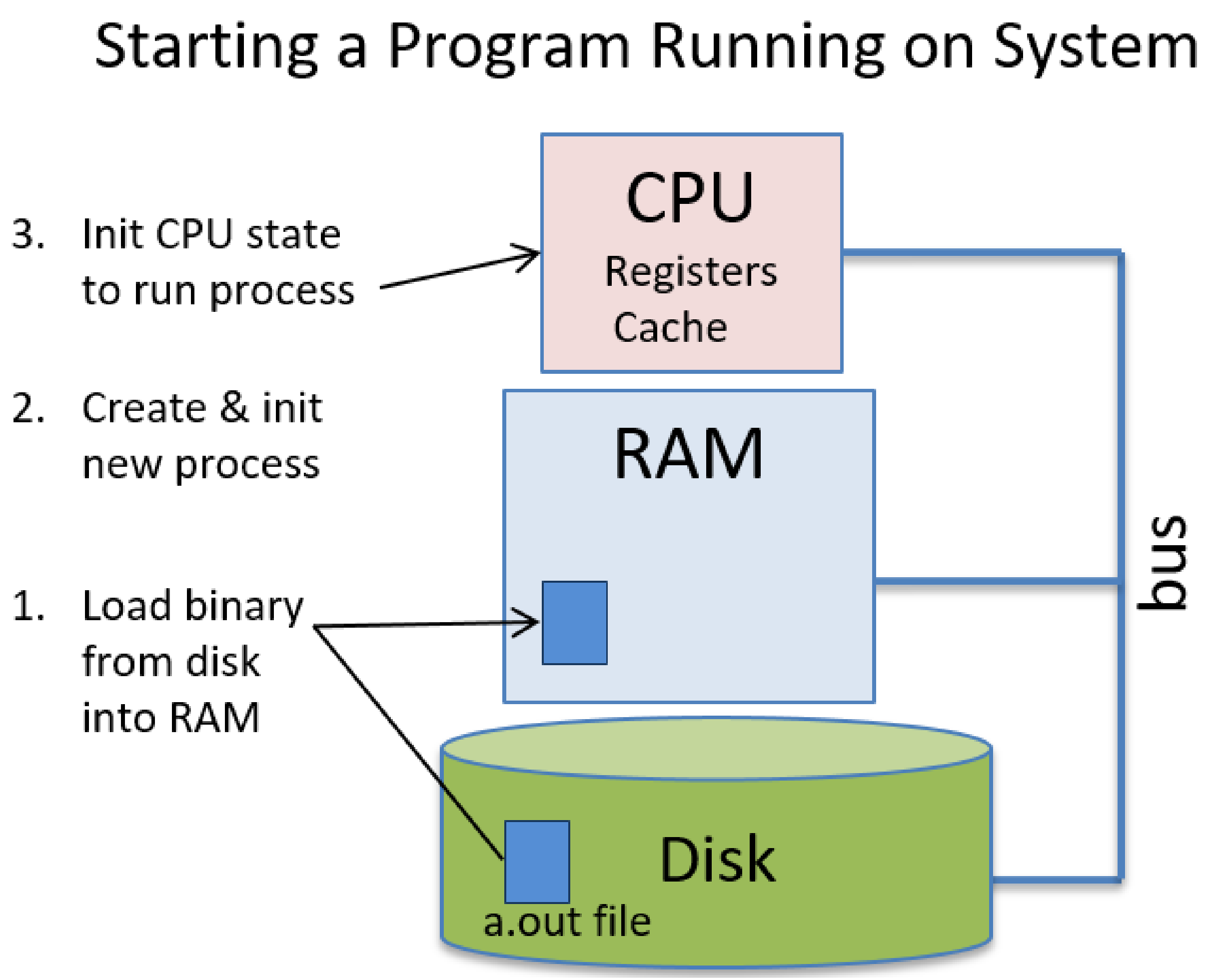


Image by Tia Newhall

Processes

A **process** is an instance of a running program

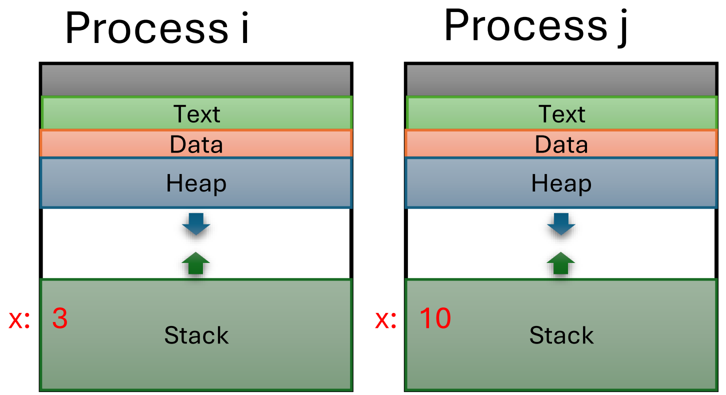
`./a.out` # OS creates a process for this run of a.out

`./a.out&` # OS creates a separate process for this run of a.out

One of the main **abstractions** implemented by OS

Features of a process:

1. Private Virtual Address Space
2. Lone View of use of the HW



Processes and you

You create new processes when

- launch a program at the command line
- double-click on a shortcut on your desktop
- boot your computer

Demo

Job Control: Multiple processes in bash

A **foreground process** receives user input (stdin), e.g. has total access to the terminal (stdin, stdout, control commands like Ctrl-Z, Ctrl-C)

A **background process** does not receive user input

$ command // runs a program in the foreground

$ command & // runs a program in the background

$ jobs // lists all programs

$ kill <jobid> stop a program

$ fg <jobid> switch a background program to the foreground

The ability to run multiple programs from the console is called **job control**

Example: Cntrl-z, bg

```
$  ./inf_loop
^Z
[1]+  Stopped  ./inf_loop
$ ps w
  PID TTY        STAT   TIME COMMAND
28850 pts/2      Ss     0:00 bash
29011 pts/2      T      0:03 ./inf_loop
29021 pts/2      R+     0:00 ps w
$ bg
./inf_loop
$ ps w
  PID TTY        STAT   TIME COMMAND
28850 pts/2      Ss      0:00 bash
29011 pts/2      R      0:03 ./inf_loop
29105 pts/2      R+      0:00 ps w
```

Ctrl-z sends a SIGTSTP signal to every process running in the foreground, process is STOPPED

bg: sends SIGCONT signal and process runs in the background (shell continues)

ps w STAT field values:

First letter:

- S: sleeping
- T: stopped
- R: running

Second letter:

- s: session leader
- +: foreground process

Slide by Tia Newhall

Example: Cntrl-c, fg

```
$  ./inf_loop
^Z
[1]+  Stopped  ./inf_loop
$ ps w
  PID TTY       STAT   TIME COMMAND
28850 pts/2     Ss     0:00 bash
29011 pts/2     T      0:03 ./inf_loop
29021 pts/2     R+     0:00 ps w
$ fg
./inf_loop
^C
$ ps w
  PID TTY       STAT   TIME COMMAND
28850 pts/2     Ss      0:00 bash
29105 pts/2     R+      0:00 ps w
```

fg: sends SIGCONT and moves process into the foreground (shell waits)

Ctrl-c sends a SIGINT signal to every process running in the foreground (process will exit)

<u>`ps w` `STAT` field values:</u>

First letter:

- S: sleeping
- T: stopped
- R: running

Second letter:

- s: session leader
- +: foreground process

Slide by Tia Newhall

Exercise: Processes

- Examine the processes running on your computer
  - `ps –AejH`

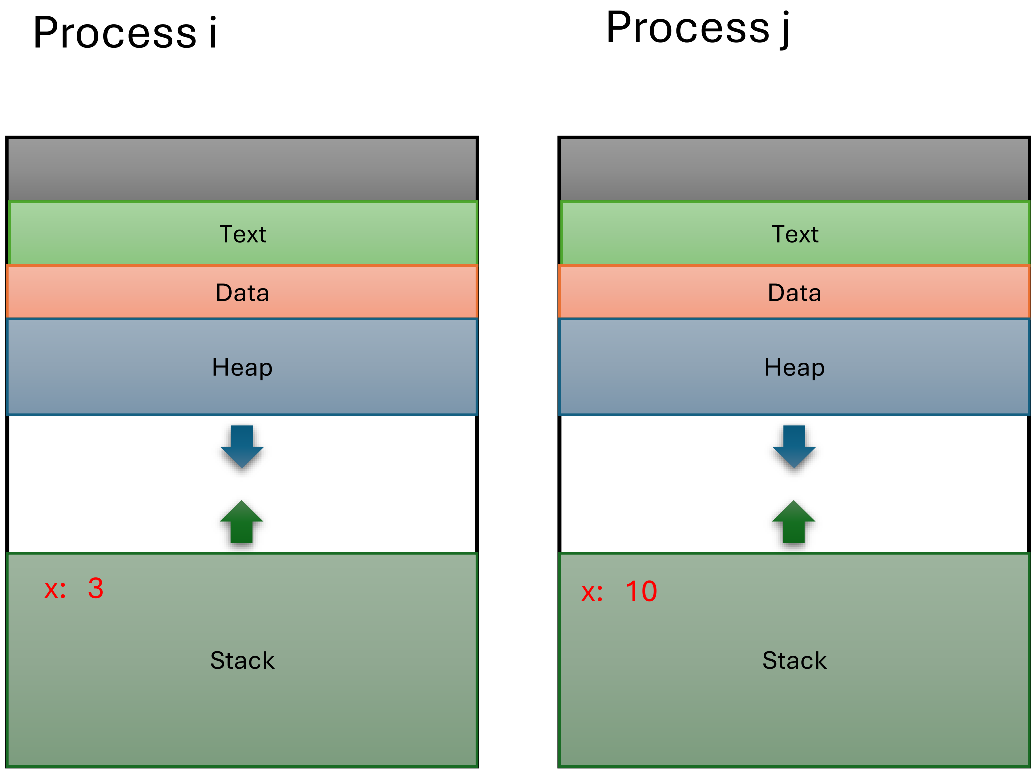
- Try running processes in the foreground and background
  - Suspend a process (Ctrl-Z)
  - Restore the process to foreground (fg)
  - Move a process from the background to the foreground (fg/bg)
  - Kill a process (Ctrl-C)

- Windows/Mac: Task Manager

Processes: the lone view

Each process "thinks" they are the only process running

- they have their own **private address space**
- multiple instances of a program each get their own private variables
- memory "appears" sequential
- sole access to the hardware and operating system



Processes: the lone view

Multiprogramming: allowing more than one process on the computer at a time

+ more efficient use of the HW
    - If one process is using disk, another can use the CPU

- OS needs to implement process abstraction

Timesharing: Each process gets a small time slice on CPU

- Each get a few ms on CPU (1 ms = 10^{-3} seconds)
- Looks like it is the only thing running

How the lone view is maintained by the OS

OS does a Context Switch to swap a different process onto the CPU:

1. Save context of current process running on CPU
   - Its Register Values:

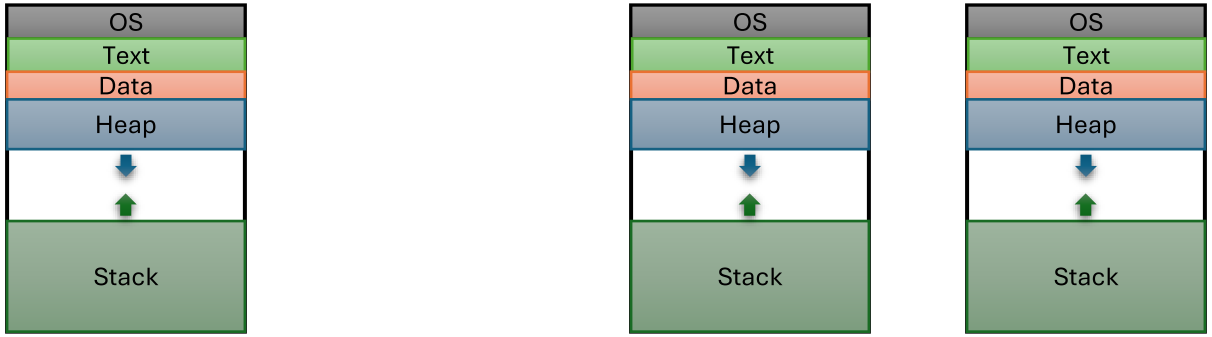
     (PC, stack pointers, general purpose register values …)
   - Its Memory state (heap, stack, …)
   - Other Stuff (open files, …)

2. Restore other process's context on CPU & let it run
   - Continue execution at next instruction where it left off

CPU Scheduling policy decides when and to what process to do a context switch

How does OS run to perform context switch?

- Processes are managed by a shared chunk of OS code called **the kernel**
  - The kernel runs in the context of any process

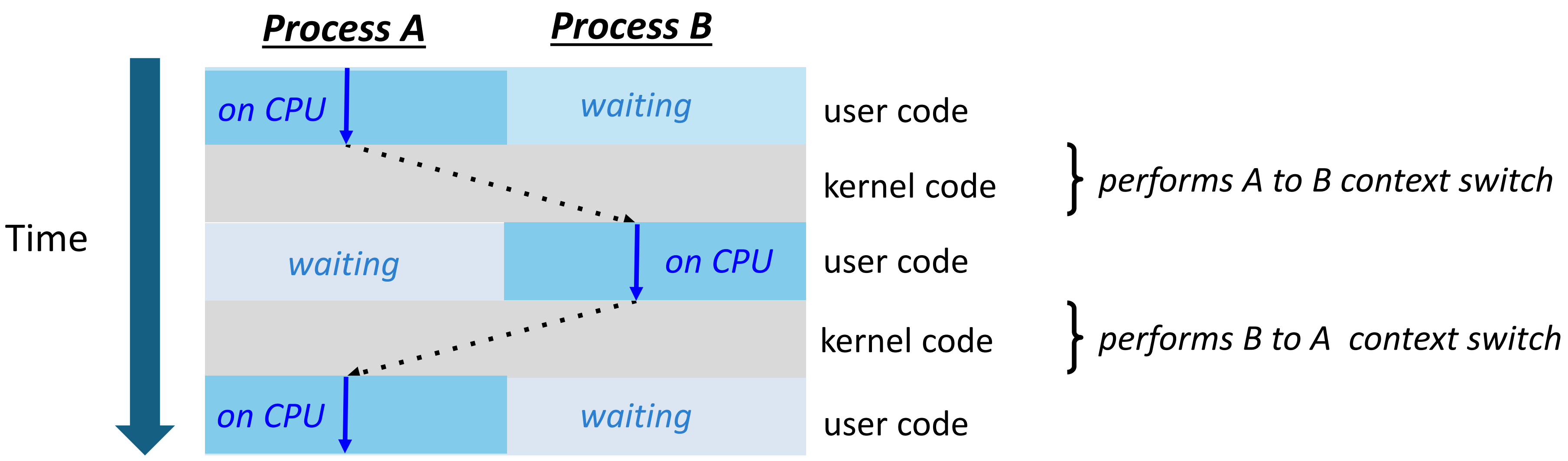


  - A process runs in a special mode to execute kernel code
    - Processes run in either User Mode or Kernel Mode
    - Kernel Mode on HW interrupt or trap instr execution
- Control flow passes from one process to another via executing kernel **context switch** code

Context Switching by OS

Control flow passes from one process to another via executing OS kernel *context switch* code

- The OS kernel runs in the context of any process



A and B are *concurrent* : their execution flows overlap in time

Process properties

OS needs to keep information with each process:

- Process identifier (pid) uniquely identifies all concurrent process in the system
- Process state: **Running, Ready, Blocked, Exited**
- Lots of other stuff (Execution Context, state for CPU scheduling algorithm, ...)

Why might a process be blocked? (T/F)

It’s waiting for another process to do something. T F

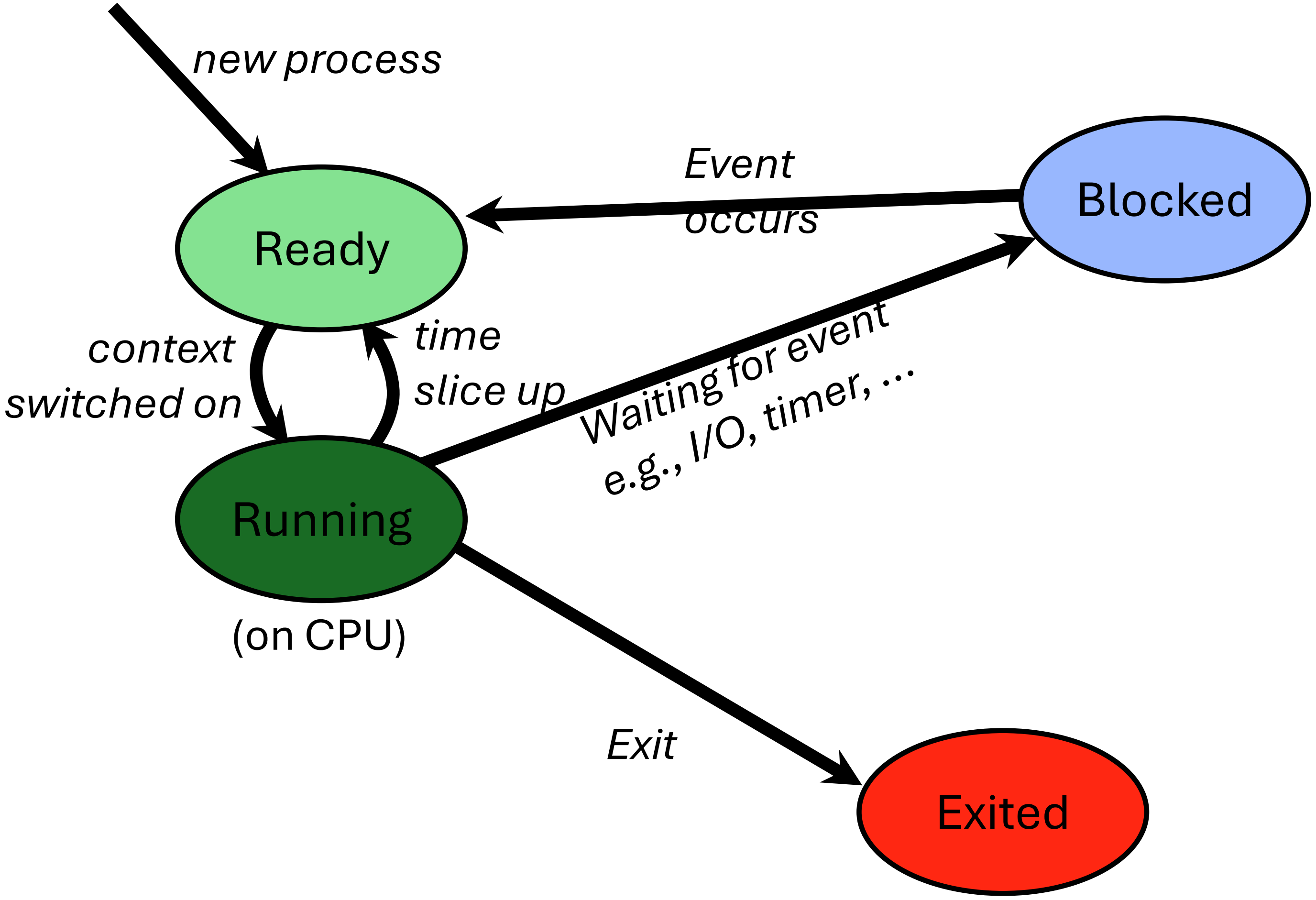
It’s waiting for memory to find and return a value. T F

It’s waiting for an I/O device to do something. T F

It’s in an infinite loop. T F

Process State

Process can be in different states during its lifetime:

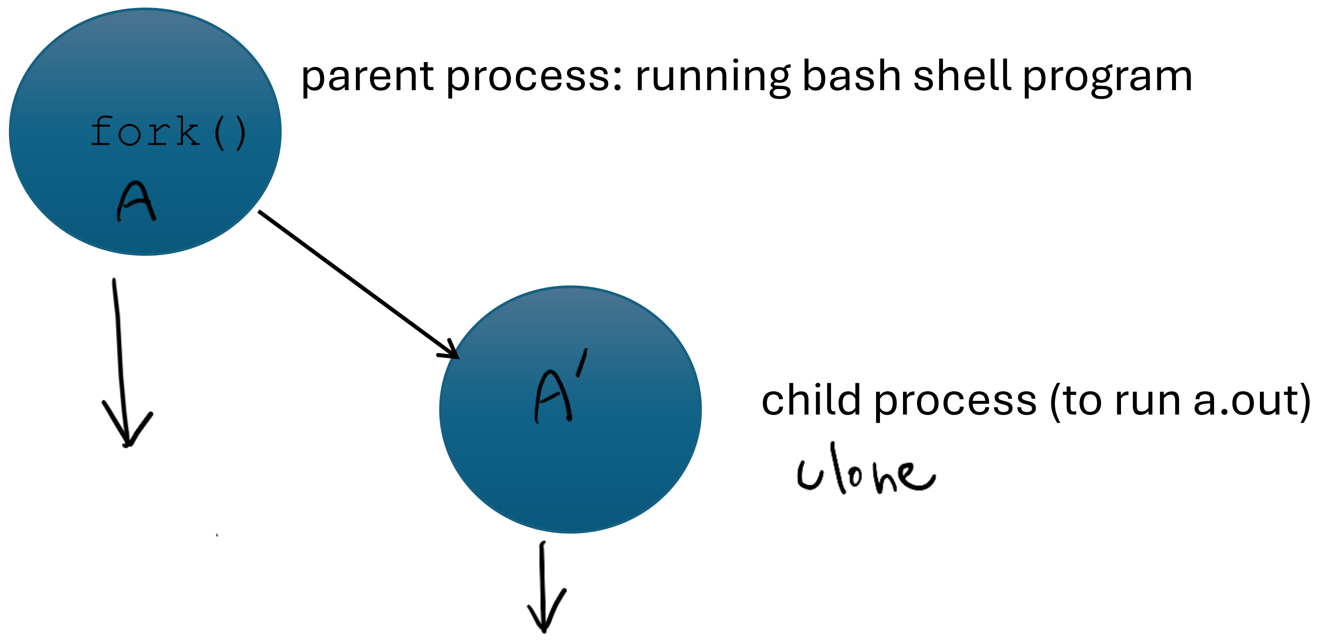


Spawning processes

`fork()`: interrupt OS and create a new process

An existing process (the parent) calls fork to create a new process (the child)

Example: Running `$ ./a.out` from the command line

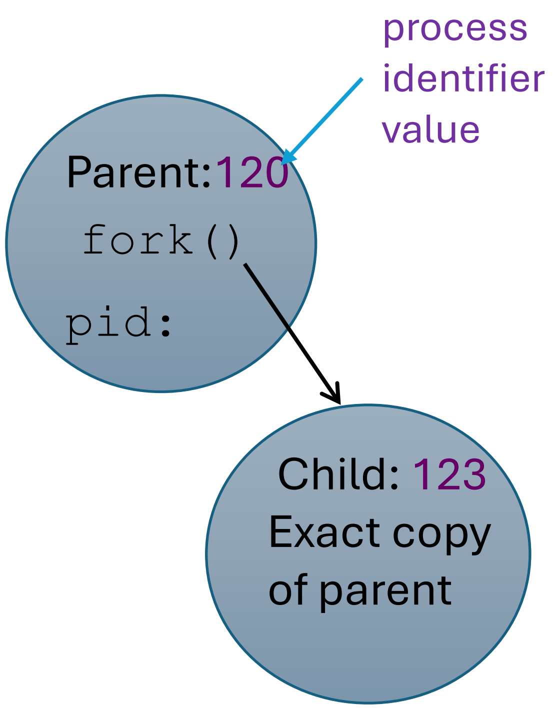


```
pid_t ret;
ret = fork();
```

Creating a new process with fork()

Creates new process (child) that is **identical copy** of the calling process (parent):

- Analogy: An exact clone who shares your memories
- Child receives a copy of parent’s
  - address space, heap, text, data, registers, etc
  - system resources, such as open files
- But each get their own **process identifier value**



Q: when child process is 1st scheduled to run, where is its execution point?

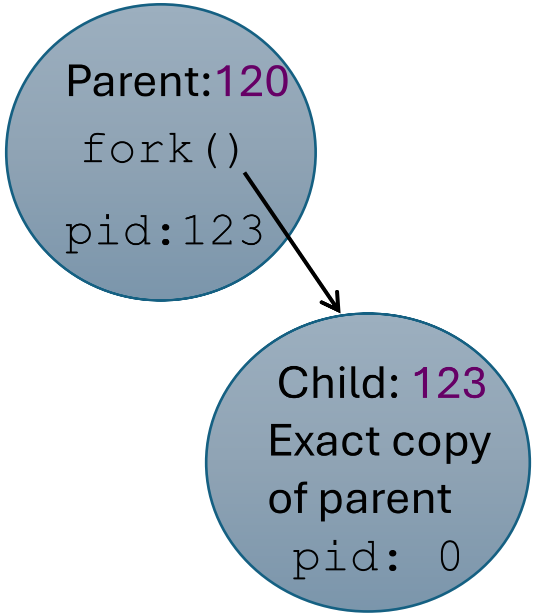
```
pid_t ret;
ret = fork();
```

Creating a new process with fork()

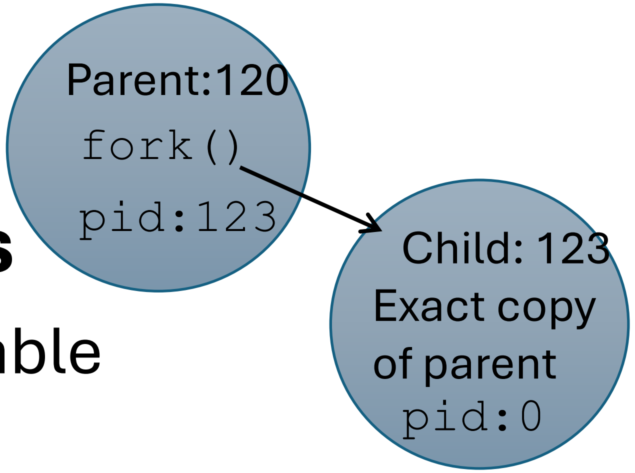
```
ret = fork();
// child and parent continue execution here
```

- `fork()` returns **0** to the child process
- `fork()` returns **child's pid** to parent process

Fork is called ***once*** (by parent) but returns ***twice*** (once in parent process & once in child process)



What Happens after a fork?



Parent & Child become **concurrent processes**

- Both assign return value to their copy of `pid` variable
- Who executes the `printf` statement first?
  - Depends on which gets scheduled on CPU first
  - Can vary every execution: no ordering of concurrent Pi's actions

```
ret = fork(); // both continue after call
if (ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Demo: fork (what is the output of this program?)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  pid_t ret;
  ret = fork();
  printf("pid = %d\n", ret);
  sleep(10);
}
```

fork example

Both Parent and Child process can continue forking

```
void forky()
{
   printf("L0 \n");

   fork();            // parent & child cont.
   printf("L1 \n");   // both print

   fork();            // both fork new child
   printf("Bye\n");// all 4 processes print
}
```

time

Exercise

```
void forky_fork() {
   pid_t ret;

   printf("L0\n");
   ret = fork();

   if(ret == 0) {
      printf("L1\n");
      ret = fork();

      if(ret == 0){
         printf("L2\n");
         fork();
      }
   }
   printf("Bye\n");
}
```

- Draw Process timeline.
- How may processes?

Parent: L0

time

Exercise

```
void fork_cntr() {
   pid_t ret;
   int cntr = 10;

   ret = fork();

   if(ret) { // pid!=0
      cntr++;
   }
   else {
      cntr--;
   }
   printf("%d\n", cntr);
}
```

- Draw Process timeline.
- How may processes?
- What is the value(s) of cntr that is printed?

Parent:

time

Exercise

```c
void fork_cntr() {
   pid_t ret;
   int cntr = 10;

   ret = fork();

   if(ret) { // ret!=0
      cntr++;
      ret = fork();
      if(ret){ // ret !=0
          cntr++;
      }
      else {
         cntr--;
      }
   } else {
      cntr--;
   }
   printf("%d\n", cntr);
}
```

- Draw Process timeline.
- How may processes?
- What is the value(s) of cntr that is printed?

Parent:

time

Terminating a Process

To see program's exit value:

```
./a.out
echo $?
```

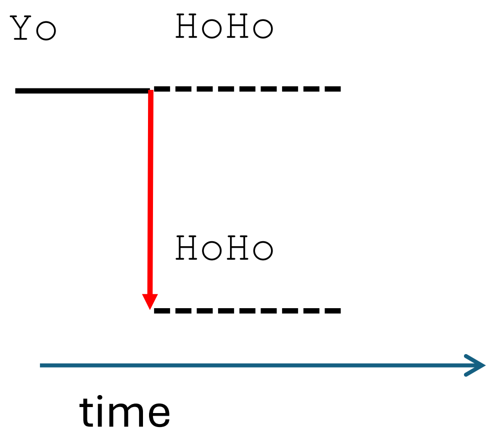
```
void exit(int status);
```

- A process calls `exit` to terminate:

  exit(0);   // 0 means: exit without an error

  exit(1); // non-zero means: an error exit

```
fork_pirates() {
   printf("Yo\n");
   fork();
   printf("HoHo\n");
   exit(0);
}
```



Exiting a program

Method 1: Explicit to the C programmer:

- include a call to `exit` in the program code

Method 2: “Implicit“ hidden from C programmer:

- return from main (code runs after main returns and it calls `exit`)

Method 3: In signal handler code (later)

- **kill signal:** another process tells your program to exit (CNTL-C)
- the process does something irreversibly bad (SEGFAULT)

What Happens when a child process exits?

It becomes a zombie process until its parent reaps it

Zombie process:

- exited, mostly dead, not runnable anymore

  “unlike real zombies they won’t try to eat other processes” – Tia Newhall

- waiting for parent to completely remove all of its state from the system

Demo: Zombies

```
void zombie(){
   if (fork() == 0) {/*child */
         printf("Child, PID = %d\n",
               getpid());
         exit(0);
   } else {/*parent */
         printf("Parent, PID = %d\n",
               getpid());
         while(1) {
          /* Infinite loop */
     }
   }
}
```

```
$ ./a.out &
Parent, PID = 6639
Child, PID = 6640

$ ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 bash
 6639 ttyp9    00:00:03 ./a.out
 6640 ttyp9    00:00:00 ./a.out <defunct>
 6641 ttyp9    00:00:00 ps

$ kill -9 6639
$ ps
```

- Above, the parent doesn't exit because of an infinite loop
- `ps` lists processes started by shell

  `<defunct>`: zombie child process
- `kill -9 6639` kills parent process, which will reap its zombie children

How does parent reap zombie child?

Parent waits for child to exit by calling a `wait` system call:

1. Blocks the parent until the child exits
2. reaps the exited child and returns

Parent receives a SIGCHILD signal when child exits, and its signal handler code calls `wait` to reap the child:

1. Reaps the exited child and returns

wait

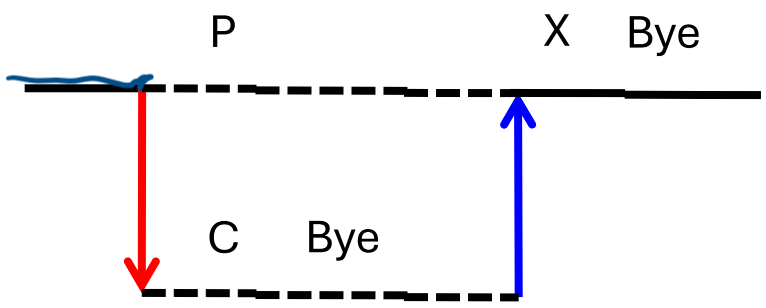
Remove all remaining parts of exited child process from the system

```
// blocks caller (parent) until child process exits,
// returns pid of child process that terminated
pid_t wait(int *child_status);
```

```
// more configurable: wait for specific child, or any, or just
// check and see if a child exited (don't block or reap if not), or …
pid_t waitpid(pid_t pid, int *status,
```

Wait Example

```
void fork_and_wait() {

   int child_status;
   pid_t pid;

   if (fork() == 0) {
      printf("C\n");
            sleep(5);
   }
   else {
      printf("P\n");
      pid = wait(&child_status);
      printf("X\n");
   }
   printf("Bye\n");
   exit();
}
```



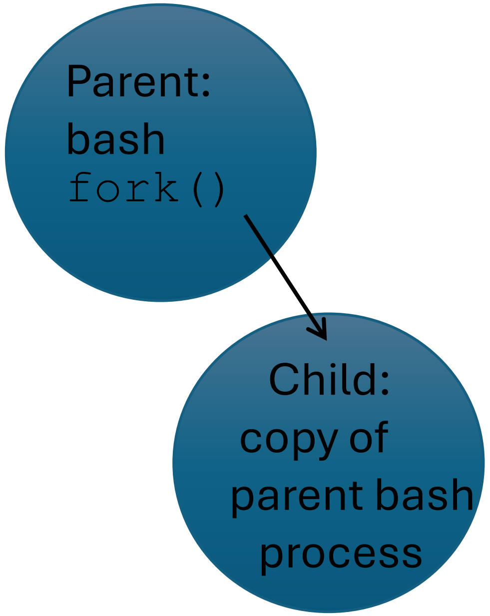
Running a program: forking is not enough

- `fork`: child gets exact copy of parent's address space and point in execution (register values, stack)

Ex: bash shell forks new child when enter command: `./a.out`

Child process is now copy of Parent bash process

***But:** What if we want to launch a new program?*



exec

(ex) `int execvp(char *filename,char *argv[]);`

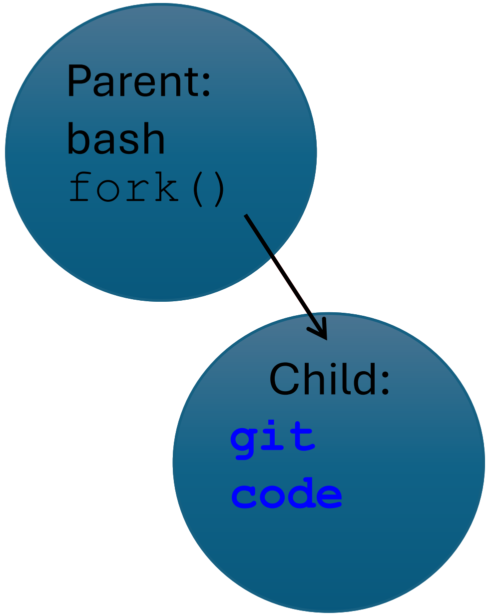
(there are different versions of exec, diff names and diff args)

1. Overlays the executable code from `filename` on the calling process's address space
2. Initializes other parts of memory space: stack, heap, data, …
3. Sets up process to execute the first instruction in the `filename` binary (changes child's `%rip` value)
4. Passes in `argv` as command line arguments

`exec` system call only returns if it fails with an error.

Child Process exec's

```
pid_t ret;
ret = fork();
if (ret == 0) {
  if (execvp("/usr/bin/git", argv) < 0) {
    printf("%s: Command not found.\n",
      argv[0]);
    exit(0);
  }
}
```



execvp: child runs "git" code from its start
rather than its copy of parent's code from after fork
(which has been completely overwritten w/a.out by exec)

Sharing resources: fork vs exec

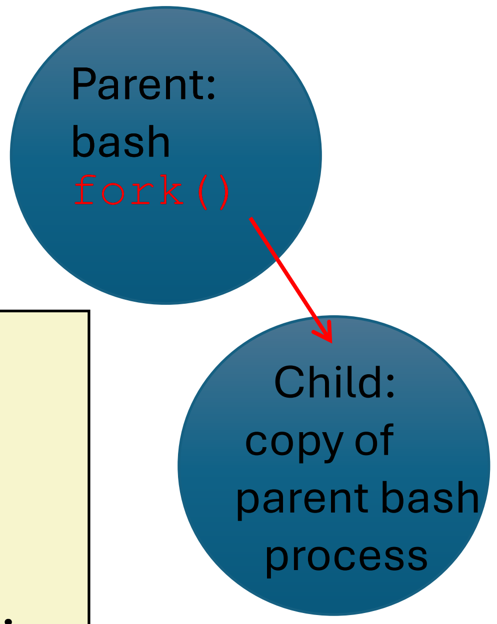
```
int main() {
  char* buffer = malloc(sizeof(char)*10);

  pid_t ret = fork();
  if (ret == 0)
  {
    printf("I am the child! %d\n", getpid());
    exit(0);
  }
  else
  {
    printf("I am the parent! %d\n", getpid());
    free(buffer);
  }
  return 0;
}
```

```
int main(int argc, char* argv[]) {
  char* buffer = malloc(sizeof(char)*10);
  pid_t pid = fork();
  if (pid == 0)
  {
    if (execvp("/usr/bin/git", argv) < 0)
    {
      printf("%s: Command not found.\n", argv[0]);
      exit(0);
    }
  }
  else
  {
    free(buffer);
  }
  return 0;
}
```

Fork-Exec-Wait Example:

```
# (ex) bash shell program: run a.out
$ ./a.out
```

```
// part of bash program to run a.out:
ret = fork(); // create new process
if (ret == 0) { /* child */
  if (execvp("/usr/bin/git",argv)<0) {
    printf("%s:Command not found.\n",argv[0]);
    exit(1);
  }
} else { /* parent */
  // wait for child to exit:
  waitpid(pid,&status,0);
}
```



```
void fork() {
   pid_t ret;
   int status;
   printf(”A\n");
   ret = fork();
   if(ret == 0) {
      printf(”B\n");
      ret = fork();
      printf(”C\n”);
      if(ret == 0) {
        printf(“D\n”);
        exit(0);
      } else {
        wait(&status);
        printf(“E\n”);
      }
   }
   else {
      wait(&status);
      printf(“F\n”);
   }
   printf(”G\n");
   exit(0);
}
```

Exercise

1. Draw Process Timeline
   1. concurrently executing dashed line
   2. no concurrent execution solid line

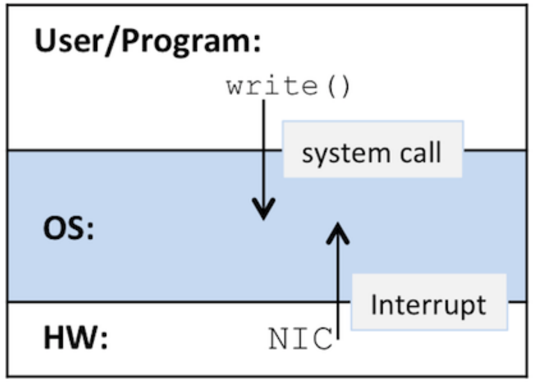
time

2. List all possible output orderings (printf ouput)

Learning when a child exited (and what happened to it)

```
waitpid(pid,&status,0);
if (WIFSIGNALED(status)) {
    int signal = WTERMSIG(status);
    printf("Segfault!! %s\n", strsignal(signal));
    if (WCOREDUMP(status))
    {
        printf("core dumped\n");
    }
}
```

The OS is an Interrupt driven system



The OS waits for requests

1. Interrupt: Stop to process a hardware event (e.g. key press)
2. Trap: Stop to process a software request (e.g. system call such as opening a file)

After handling the interrupt, the OS resumes its previous task

System Calls

Programs use **system calls** to asks the OS to perform an action

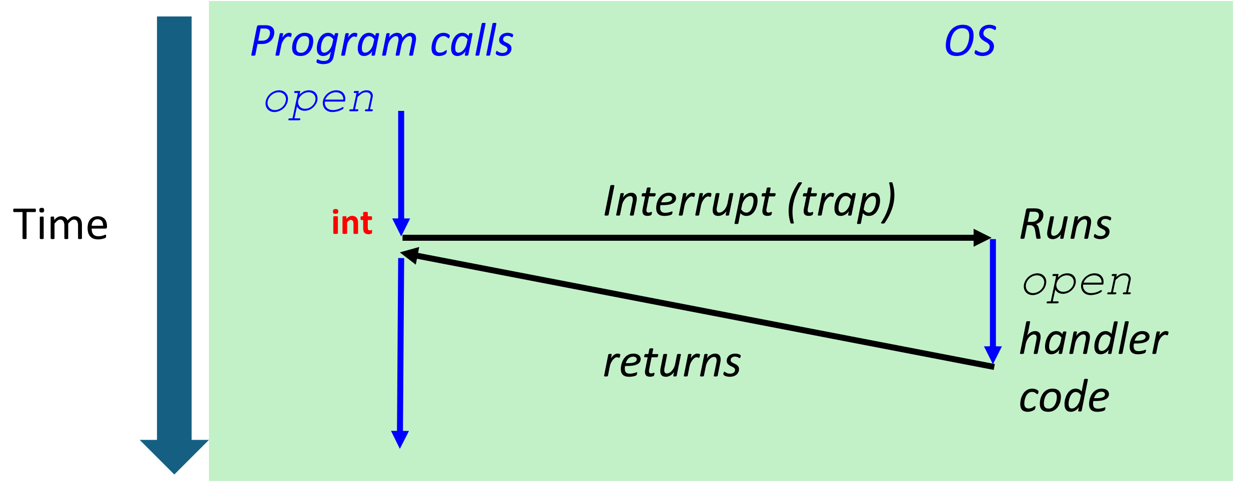
- Examples: open(), close(), fork(), wait(), exec(), etc
- Corresponds to a function call that executes in kernel mode
- Triggers a trap in the kernel

Software Interrupt (trap)

Traps are implemented as interrupts to the OS that trigger an OS **trap handler** to run

- Example: `int fd = open(filename,options)`

```
0804d070 <__libc_open>:
 . . .
0804d082: int  $0x80   # interrupt 0x80 is trap
```



Hardware Interrupt

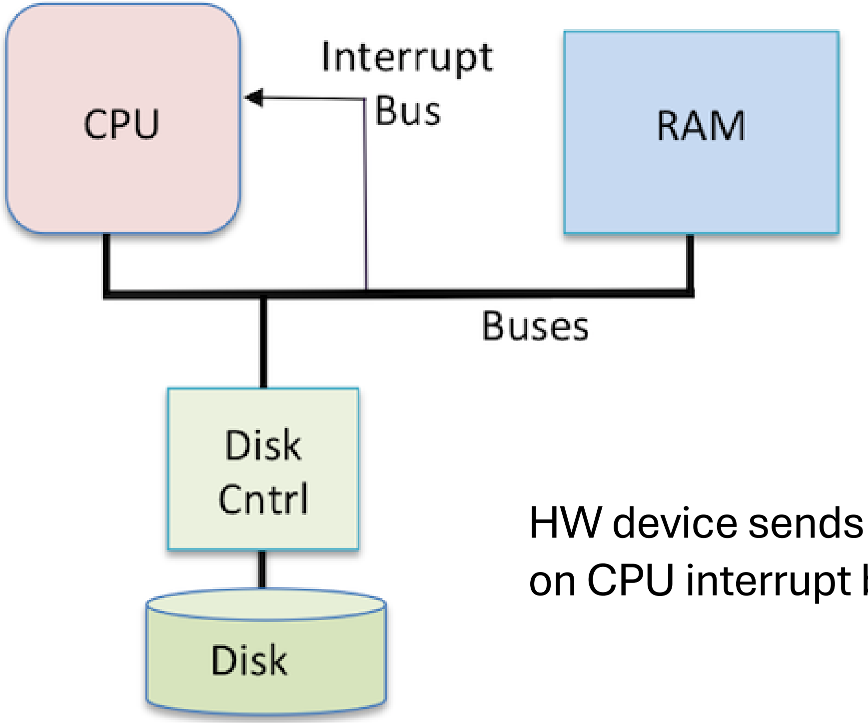
Interrupts are implemented as **signals** on the **interrupt bus**

The CPU responds to the interrupt by running an **interrupt handler**, e.g. code configured to execute in response

Examples:

The keyboard *interrupts* the OS when someone presses a key

The disk interrupts the OS when data is ready to read



HW device sends signal on CPU interrupt bus

Signals

Signals are a type of software interrupt. A small message to tell a process that some event has happened

1. OS sends a signal to a process
   - On behalf of another process that called the `kill` syscall
   - As the result of some event (NULL pointer dereference)

2. A process receives a signal

   Asynchronous: signalee doesn't know when it will get one

   Signals are pending before a process recieves it

3. A signal interrupts the receiving process, which then runs signal handler code
   - default handlers for each signal type in OS
   - programmer can also add signal handler code

Signals

OS identifies specific signal by its number, examples:

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Invalid memory reference (e.g. NULL ptr)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

segfault!

Sending Signals:

Unix command:

`$ kill -9 1234` # send `SIGKILL` signal to process 1234

System call:

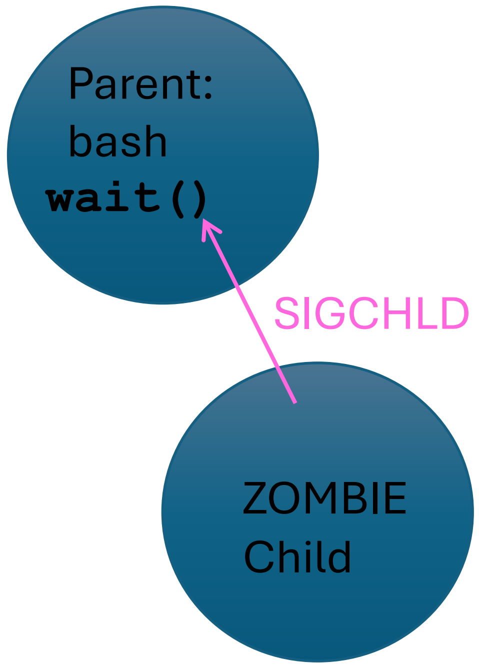
`kill(1234, SIGKILL);` // send `SIGKILL` to process 1234

Implicitly sent: side-effect of program doing something (NULL ptr dereference causes `SIGSEGV`)

Revisiting: How does parent know when child process has exited?

`exit`: is a system call

- OS kernel code runs to perform exiting on behalf of the calling process
- Part of the exiting in the OS involves sending a `SIGCHLD` signal to the exiting process' parent process, notifying it that its child has exited
- The parent's call to `wait` will return after it receives the `SIGCHLD` and has reaped its zombie child
  (parent process blocks on call to `wait` until it receives a `SIGCHLD`)



Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal

- Three possible ways to react:
  - ***Ignore*** the signal (do nothing)
    not all signals can be ignored (e.g. SIGKILL)
  - ***Terminate*** the process on receipt of signal
  - ***Catch*** the signal by executing a user-level function called signal handler

Example: Killing a process

The SIGINT signal terminates a process

We send this signal when we press Ctrl-C

Use strace –e ‘trace=!all’ <cmd> to investigate

```
$ strace -e 'trace=!all' ./infinite
--- SIGINT {si_signo=SIGINT, si_code=SI_USER, si_pid=76,
si_uid=1000} ---
+++ killed by SIGINT +++
```

```
$ kill -2 <pid>
```

Writing your own signal handlers

```
signal(int signum, handler_t *handler);
```

- Modifies the default action associated with the receipt of a particular signal

- `handler` is a ***signal handler*** function
    - When program receives signal, it jumps to start executing the `handler` function.
    - When the `handler` done executing, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Demo: Simple signal handler

```
// signal handler function: called when process receives SIGINT
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/wait.h>

void int_handler(int sig) {
  printf("Proc %d received signal %d\n",getpid(), sig);
  exit(0);
}

void main() {
  signal(SIGINT, int_handler);
  while(1);
}
```

Aside: function pointers

- Functions can be treated like data
  - They have a type determined by their parameters and return types
  - They can be stored in variables

- Applications: responding to asynchronous events (the software doesn't know when they will happen!)
  - user input
  - signals
  - network messages
  - etc

Defining a function pointer in C

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

Example: function pointer

```
#include <stdio.h>

typedef int (*functionType)(int a, int b);

int example(int a, int b) {
  printf("This is a function stored as data! %d\n", a);
  return 3;
}

int main() {
  functionType myFunction = example;
  int val = (*myFunction)(10, 3);
  printf("%d\n", val);
}
```