Today

http://PollEv.com/sarasprenkle

- Interrupt Handling
- Processes
  - State
  - Creation

Project 1 Checkpoint

- Goal:
  - Understanding booting
  - Writing the letter ‘A’
- ~3 who haven’t created their GitHub repository yet
- ~100 Lines of code
  - Just an estimate
  - Mine was around 86, including comments
    - With notes about common problems
  - Yours may be more streamlined

Review

- The OS is interrupt-driven
  - What are examples of when interrupts are triggered?
  - Why are interrupts used?
  - Why is interrupt-driven a good way to design the OS?
  - How are interrupts handled?
    - What are the goals for how interrupts are handled?

Review: Exceptions: trap, fault, interrupt

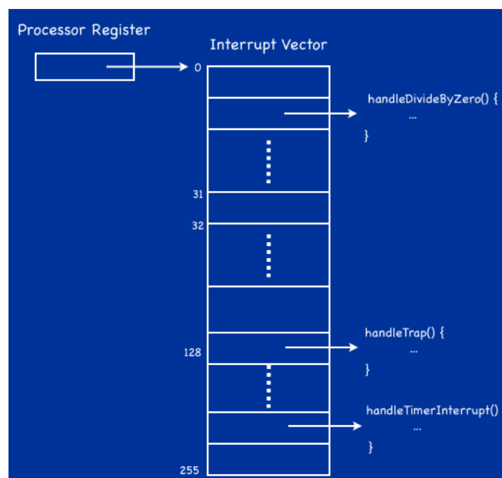
	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** user program requests. Examples: open, close, read, write, fork, exec, exit, wait, kill	**fault/exception** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event (not related to instruction that just executed): I/O op completed, clock tick, power fail, etc.

Review: How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- ***Atomic*** transfer of control
  - A single instruction changes:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Review: User Mode to Kernel Mode: Details

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - system call?
  - interrupt? then which hardware device?
  - which exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



Project 2

Saving the State of the Interrupted Process

- Privileged hw register points to ***Exception*** or ***Interrupt Stack***
  - on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack before handler runs. Why?
  - then handler pushes the rest
  - On return, do the reverse
- Why not use user-level stack?
  - reliability: even if user's stack points to invalid address, handlers continue to work
  - security: kernel state should not be stored in user space
    - could be read/written by user programs
- One interrupt stack per processor/process/thread

The interrupt vector is used to determine the action taken by the OS when:

An exception occurs

An interrupt occurs

A system call is executed

All of the above

None of the above

Poll Everywhere Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app

Question

http://pollev.com/sarasprenkle

The interrupt vector is used to determine the action taken by the OS when:

A. An exception occurs
B. An interrupt occurs
C. A system call is executed
D. **All of the above**
E. None of the above

Switching Back!

- From an interrupt, just reverse all steps!
  - asynchronous, so not related to executing instruction
- From exception and system call, increment PC on return
  - synchronous, so you want to execute the *next* instruction, not the same one again!
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by the hardware

Dual Mode Execution: One Piece of the Protection Pie

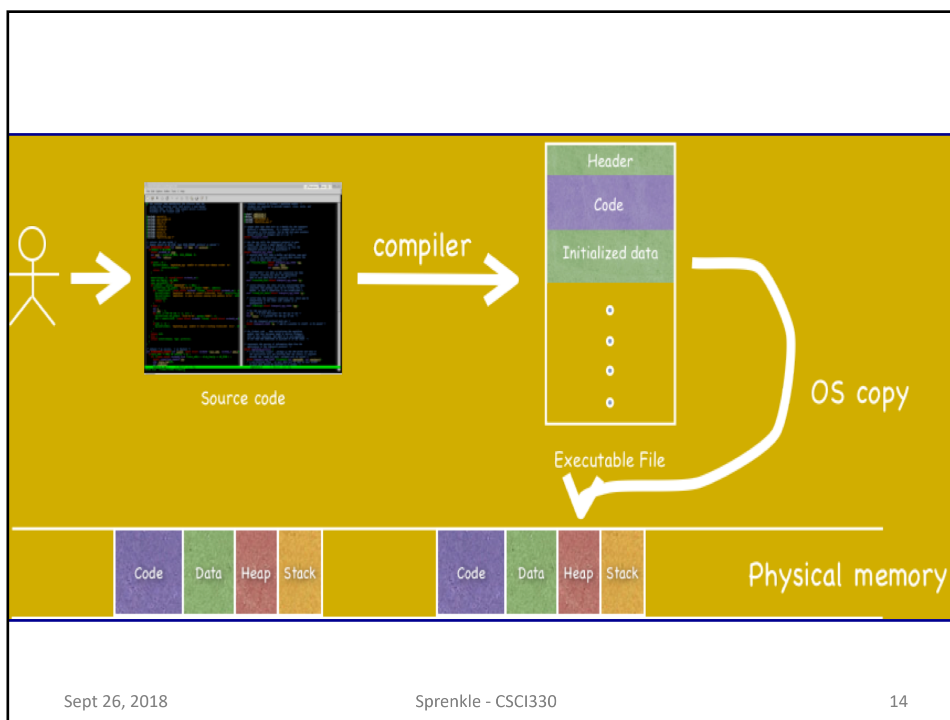
- For efficient protection, the hardware must support at least 3 features:
  - Privileged instructions
  - Timer interrupts
  - Memory protection

Dual Mode Execution: One Piece of the Protection Pie

- Privileged instructions
  - Instructions only available in kernel mode
  - In user mode, no way to execute potentially unsafe instructions
  - Prevents user processes from, for instance, halting the machine
  - Implementation: mode status bit in the process status register

Dual Mode Execution: One Piece of the Protection Pie

- Timer interrupts
  - Kernel must be able to periodically regain control from running process
  - Prevents process from gaining control of the CPU and never releasing it
  - Implementation: hardware timer can be set to expire after a delay and pass control back to the kernel



Dual Mode Execution: One Piece of the Protection Pie

- Memory protection
  - In user mode, memory accesses outside a process' memory region are prohibited
  - Prevents unauthorized access of data
  - Implementation: We'll return to this later in the course

Dual-Mode Summary

- Operating System provides protection through dual-mode execution
  - Mode changes through interrupts (e.g., time slice), exceptions, or system calls.
  - A status bit in a protected processor register indicates the mode
  - Privileged instructions can only be executed in kernel mode

So far: Approximately Chapters 1-2 with a bit of 3
Now onto more 3

Deeper dive

PROCESSES

Review

- What is a process?

Main OS Process-related Goals

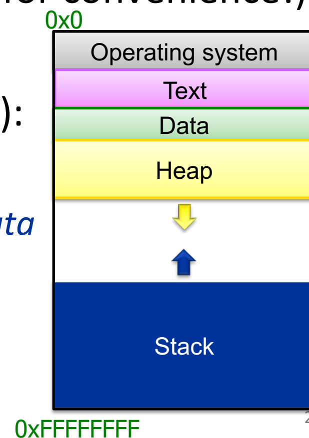
- **Overarching Challenge**: how to implement & ensure efficient use of system resources?

- Interleave the execution of existing processes to maximize processor utilization
- Provide reasonable response times
- Allocate resources to processes
- Support inter-process communication, synchronization, and user creation of processes

Process Resources: Memory

- Abstraction: Virtual Address Space (VAS)
- Give *every process* the illusion of having all of the system's memory. (for convenience!)

- At process startup (fork+exec):
  - Code loaded from disk to *text*
  - Static variables initialized in *data*
  - Stack created in *stack*



Process Resources: I/O

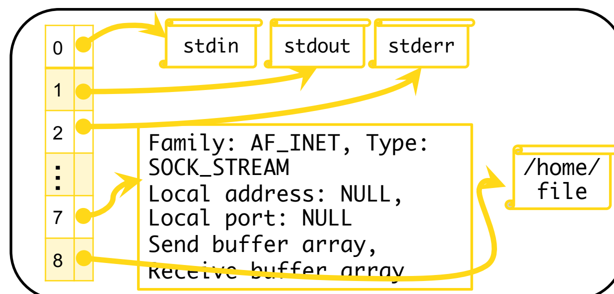
- Abstraction: File
- Old Unix adage: “Everything is a file”, including:
  - files (duh)
  - sockets (abstraction used for network communication)
  - pipes (send the output of one process to the input of another)
  - most I/O devices (e.g., mouse, printer, graphics card)*

*Not the only way to access these devices.

I/O Resource Accounting

- For each process, OS maintains a file descriptor table.
  - Give integer **f**ile **d**escriptor to process, store details in OS
- By default, all processes get stdin, stdout, stderr
- For anything else, explicitly ask the OS (e.g., open())

`printf` is a `write()` to FD 1 (`stdout`).



Why treat all of these I/O things as files?

It's less error-prone **A**

It provides higher performance **B**

It's simpler to access all of them in the same way **C**

Some other reason **D**

oll Everywhere Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Why treat all of these I/O things as files?

A. It's less error-prone.
  - Restricted interfaces → less opportunities for errors

B. It provides higher performance.
  - Adding abstractions → slows it down but it's worth it

C. It's simpler to access all of them in the same way.
  - Definitely! We saw that with `fprintf`

D. More than one of these.

E. Some other reason(s).

Device Interrupts

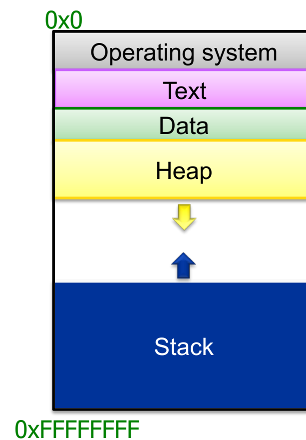
- Kernel needs to communicate with physical devices
- Devices operate ***asynchronously*** from the CPU
  - Polling: Kernel checks some time period to check if I/O is done (less efficient)
  - Interrupts: Kernel can do other work in the meantime

Process State

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

- The code for running the program
- The Program Counter (PC) indicating the next instruction
- An execution stack with the program's call chain (the stack) and the stack pointer (SP)
- The static data for running the program
- Space for dynamic data (the heap), the heap pointer (HP)
- *Values of CPU registers*
- A set of OS resources in use (e.g., open files)
- *Process identifier (PID)*
- *Process execution state*
- *(and more)*



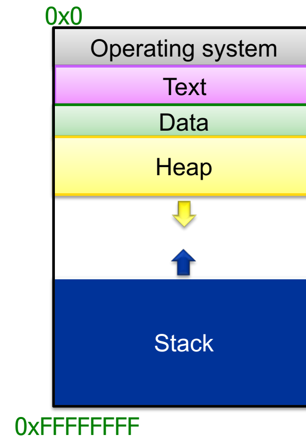
(Italics – state not shown)

Process State

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

- The code for running the program
- The Program Counter (PC) indicating the next instruction
- An execution stack with the program's call chain (the stack) and the stack pointer (SP)
- The static data for running the program
- Space for dynamic data (the heap), the heap pointer (HP)
- *Values of CPU registers*
- A set of OS resources in use (e.g., open files)
- *Process identifier (PID)*
- ***Process execution state***
- *(and more)*



(Italics – state not shown)

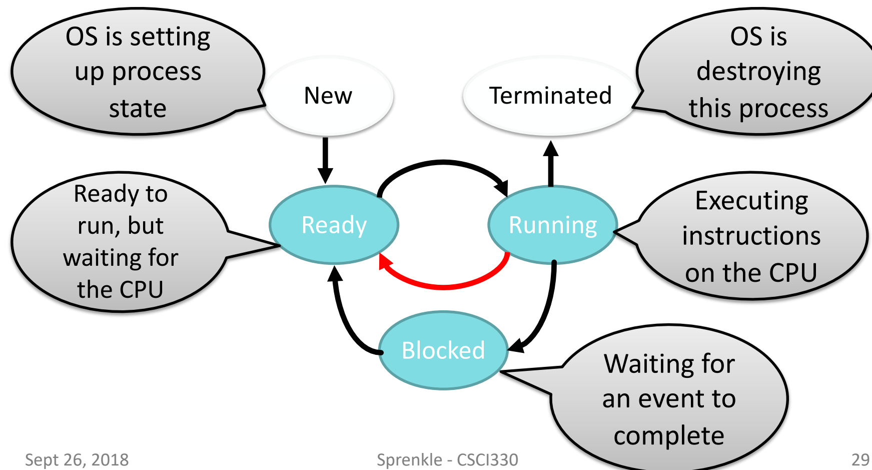
Process Execution State

- "What can this process do right now?"
- **Running**: process is executing on the CPU
- **Ready**: process is ready to execute, but we have to give it the CPU
- **Waiting / blocked**: process is waiting for *something* to happen before it can continue.
  - Does NO GOOD to schedule it.

Examples:
- Waiting for I/O to complete.
- Process needs to wait for exclusive resource (e.g., mutex).
- Process asks to be put to sleep for a while…

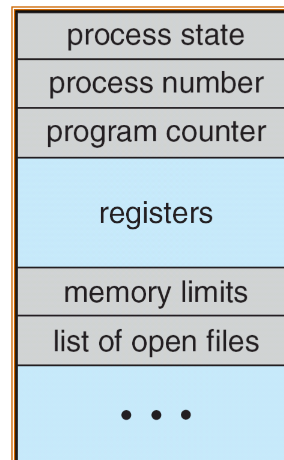
Process Life Cycle

- Processes are always either ***running***, ***ready to run***, or ***blocked waiting for an event*** to occur



Process Control Block (PCB)

- Kernel data structure kept in memory
- Represents the execution state and location of each process when it is not executing
  - Process identification number, program counter, stack pointer, contents of general purpose registers, memory management information (HP, etc), username of owner, list of open files…
  - Basically, any process execution state that is not stored in the address space
- PCBs are initialized when a process is created and deleted when a process terminates



When a process is waiting for I/O, what is its scheduling state?

- Ready
- Running
- Blocked
- Zombie
- Exited

Poll Everywhere — Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app

Question

When a process is waiting for I/O, what is its scheduling state?

A. Ready
B. Running
C. **Blocked**
D. Zombie
E. Exited

It doesn't make sense for a process to go from___. Why not?

running to waiting/blocked

ready to waiting/blocked

ready to running

running to ready

Poll Everywhere Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app

It doesn’t make sense for a process to go from___. Why not?

A. running to waiting/blocked

B. ready to waiting/blocked

C. ready to running

D. running to ready

Creating a Process

- When a program begins running, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes argc and argv on the stack
  - sets the CPU registers properly and calls _start()
- Program starts running at _start()

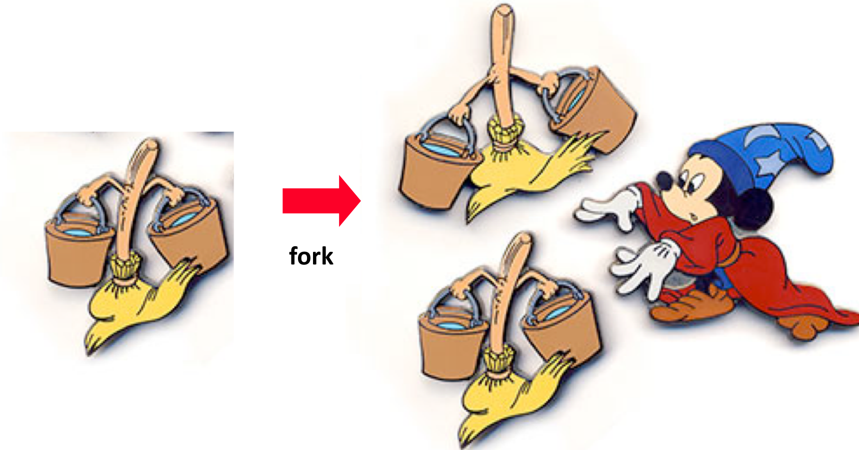
```
_start(args) {
    ret = main(args);
    exit(ret)
}
```

  - we say "process" is now running and no longer think of "program"
- When main() returns, OS calls exit() which destroys the process and returns all resources
  - unless the process calls exit() on its own

How to Create a Process

- One process can create other processes
  - The created processes are the ***child*** processes
  - The creator is the ***parent*** process
- In some systems, the parent defines (or donates) resources and privileges to its children
- The parent can either wait for the child to complete or continue in parallel

The essence of Unix process "fork"



Oh Ghost of Walt, please don't sue me.

Sorcerer's Apprentice Atari Game

fork()

- In Unix, processes are created by `fork()`
- `fork()` copies a process into an (identical) process
  - Copies variable values and program counter from parent to child
  - Returns twice: once to the parent and once to the child
    - In parent, it is child process id
    - In child, it is 0
  - Both processes begin execution from the **same point**
    - Immediately following the call to `fork()`
  - Each process has its own memory and its own copy of each variable
    - Changes to variables in one process are not reflected in the other!

fork()

```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        …..
} else {
        /* child */
        …..
        exit(status);
}
```

The **fork** syscall returns *twice*:

1. It returns a zero in the context of the new child process.
2. It returns the new child process ID (pid) in the context of the parent.

fork() Pseudocode

```
pid_t fork_val = fork();              //create a child
if((fork_val == FORKERR)              //FORKERR is #define-d to -1
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                //fork_val != child's PID
  printf("I am the child!");          //so child continues here
  return EXIT_SUCCESS;
else
  pid_t child_pid = fork_val          //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);      //wait for child to finish
```


exit and wait

- exit(int rv)
  - Causes the program to exit with the main function returning the specified return value (rv).
    - e.g. exit(-1);
  - Reaching the end of the main function results in an *implicit* exit(0).
- wait(int *status)
  - Causes a process to wait until any one of its child processes has completed.
  - The `waitpid` system call can be used to wait for a *specific* child process to complete.
  - status is loaded with the return value from the child's call to exit. Use NULL to discard status.

Example: fork()

```
pid_t fork_val = fork();
if(fork_val == 0) {
 printf("Child!\n");
} else {
 wait();
```

```
pid_t fork_val = fork();
if(fork_val == 0) {
 printf("Child!\n");
} else {
 wait();
```

USER

OS

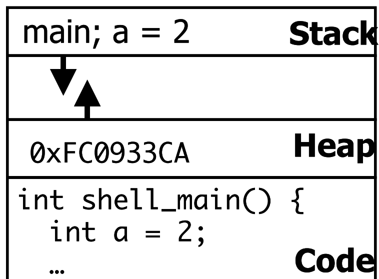
pid = 127

pid = 128

Process Control Blocks (PCBs)

 (stopped here)

Example: fork()



USER

OS

pid = 127

pid = 128

Process Control Blocks (PCBs)

Ummm, okay, but….

Why do I want two copies of the same process? What if I want to start a different process? How do I do that?

exec()

- Overlays a process with a new program
  - PID does not change
  - Arguments to new program may be specified
  - Code, stack, and heap are overwritten
    - Sometimes memory-mapped files are preserved
- Child processes often call `exec()` to start a new and different program
  - New program will begin at `main()`
- If call is successful, it is the *same* process, but it is running a *different* program!

fork() and exec(): Pseudocode

```
pid_t fork_val = fork();                    //create a child
if((fork_val = fork()) == FORKERR)
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                      //child continues here
  exec_status = exec("calc", argc, argv0, argv1, …);
  printf("Why would I execute?");           //should NOT execute
  return EXIT_FAILURE;
else
  pid_t child_pid = fork_val                //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);            //wait for child to finish
```


Practical Usage: ps and kill

If you have a process running that you need to kill:

- From the command line, type:
  `ps –au <login_name>`
- Find the process you would like to terminate (the name is in the CMD column) and then determine its PID. You can do this visually or use `grep`:
  `ps –au <login_name> | grep <program_name>`
- From the command line, type:
  `kill -9 <PID>`

Looking Ahead

- Project 1 due Tuesday night
- Process scheduling