Today

http://PollEv.com/sprenkle

- Processes
  - Creation
  - Management

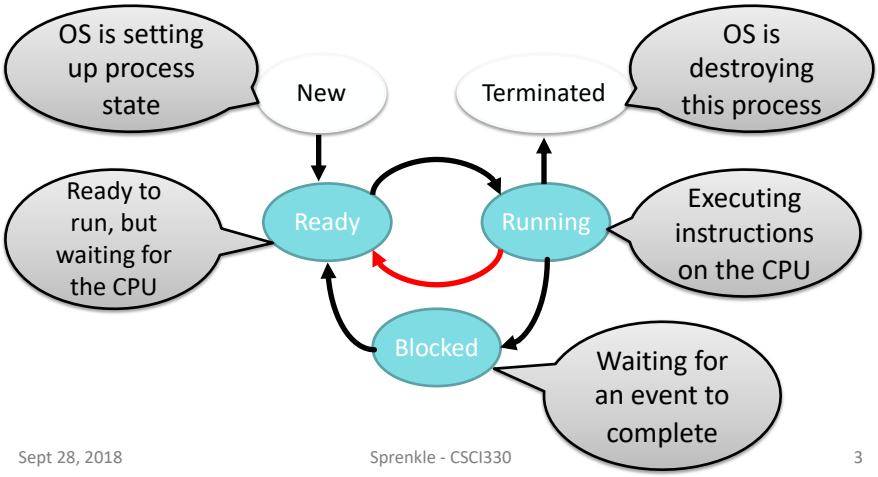
> Operating systems are like underwear —
> nobody really wants to look at them.
> -- Bill Joy
> Co-Founder Sun Microsystems
> Original author of vi

Review

- What are the 3 primary execution states of processes?
- How does the OS keep track of information about processes?
  - What is some of the info it keeps?
- How does a process create another process?
  - What are the characteristics of that newly created process?
  - What do we call a process that creates another process?

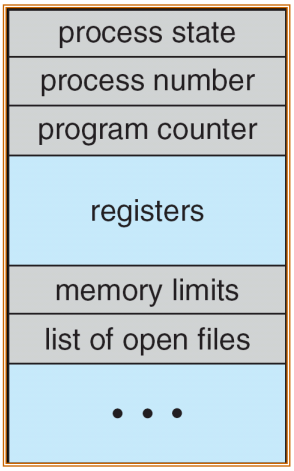
Review: Process Life Cycle

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



Review: 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



Review: 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!");
   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
```

Fork returns TWICE!
Once to child (with 0)
Once to parent (with child id)

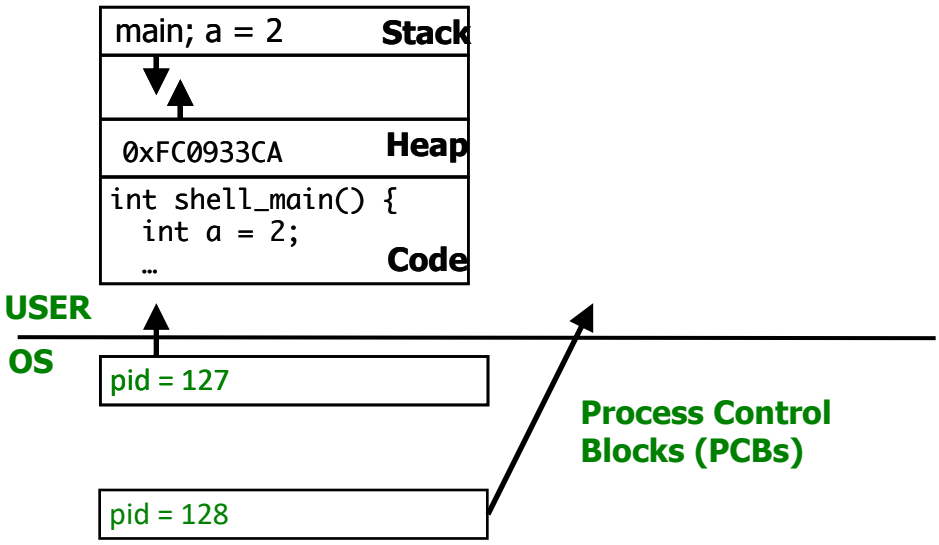
exit and wait

- exit(int rv)
  - Causes the program to exit
  - main function returns the specified return value (rv).
    - e.g. exit(-1);
  - Reaching the end of the main function results in an *implicit* exit(0).

exit and wait

- exit(int rv)
  - Causes the program to exit
  - main function returns 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 (block) until *any* one of its child processes has completed.
    - So that parent can get child's return value
  - 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()`



Fork Problem

What will this program output?

```
int main() {
    int x = 20;
    int pid = fork();
    int status;
    if (pid != 0) {
        printf("Parent's x before wait is %d\n",x);
        x = x + 5;
        wait(&status);
        printf("Parent's x after wait is %d\n",x);
        printf("Parent's child's status is %d\n",
                    status);
    } else {
        printf("Child's x before sleep is %d\n",x);
        sleep(3);
        x = x + 10;
        printf("Child's x after sleep is %d\n",x);
    }
}
```

fork_problem.c

Fork Problem

```
Parent's x before wait is 20
Child's x before sleep is 20
Child's x after sleep is 30
Parent's x after wait is 25
Parent's child's status is 0
```

Top two lines of output could be swapped.

```
int main() {
    int x = 20;
    int pid = fork();
    int status;
    if (pid != 0) {
        printf("Parent's x before wait is %d\n",x);
        x = x + 5;
        wait(&status);
        printf("Parent's x after wait is %d\n",x);
        printf("Parent's child's status is %d\n",
                    status);
    } else {
        printf("Child's x before sleep is %d\n",x);
        sleep(3);
        x = x + 10;
        printf("Child's x after sleep is %d\n",x);
    }
}
```

Another Fork Program

```
int main() {
    int pid= fork();
    int i, status;

    if (pid != 0 ) {
        for(i=0; i<10; i++) {
            printf("Parent process %d running.\n", getpid());
            sleep(1);
        }
        wait(&status);
    }
    else {
        for(i=0; i < 10; i++) {
            printf("Child process %d running.\n", getpid());
            sleep(1);
        }
    }
}
```

getpid syscall: Get processID of current process.

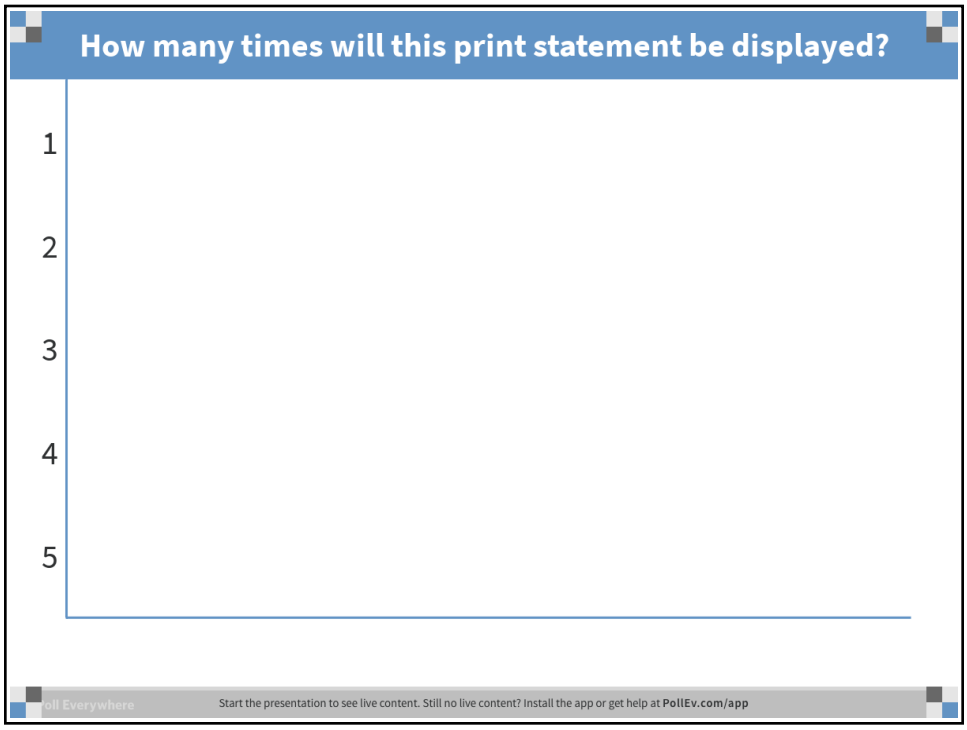
dueling_processes.c

What the fork?

```
int main() {
  fork();
  fork();
  printf("Process %d exiting.\n", getpid());
}
```

How many times will this print statement be displayed?

sixforks.c



What the fork?

```
int main() {
  fork();
  fork();
  printf("Process %d exiting.\n", getpid());
}
```

How many times will this print statement be displayed? **4**

1. Main/Original process
2. Main's first forked process
3. Main's second forked process
4. Main's first forked process's child forked process

Likely Questions

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

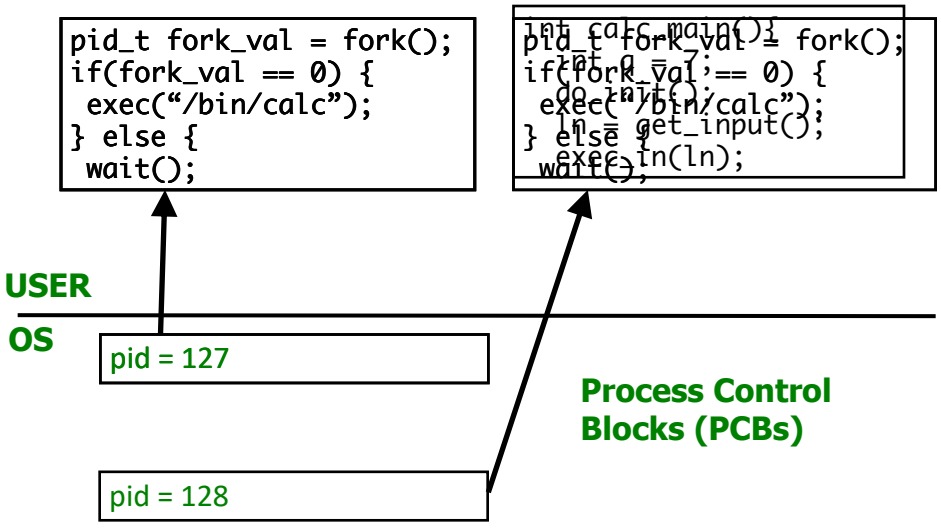
exec()

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

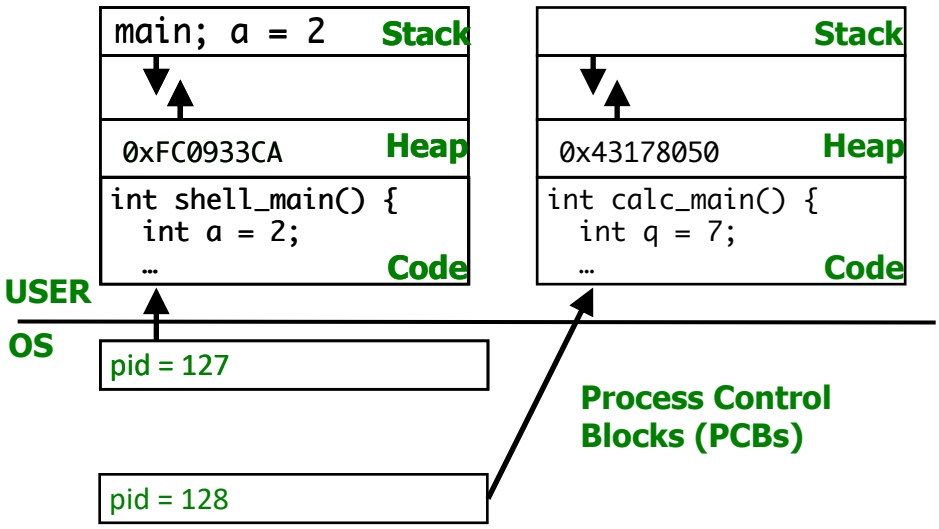
fork() and exec(): Pseudocode

```
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
```


Example: fork() and exec()



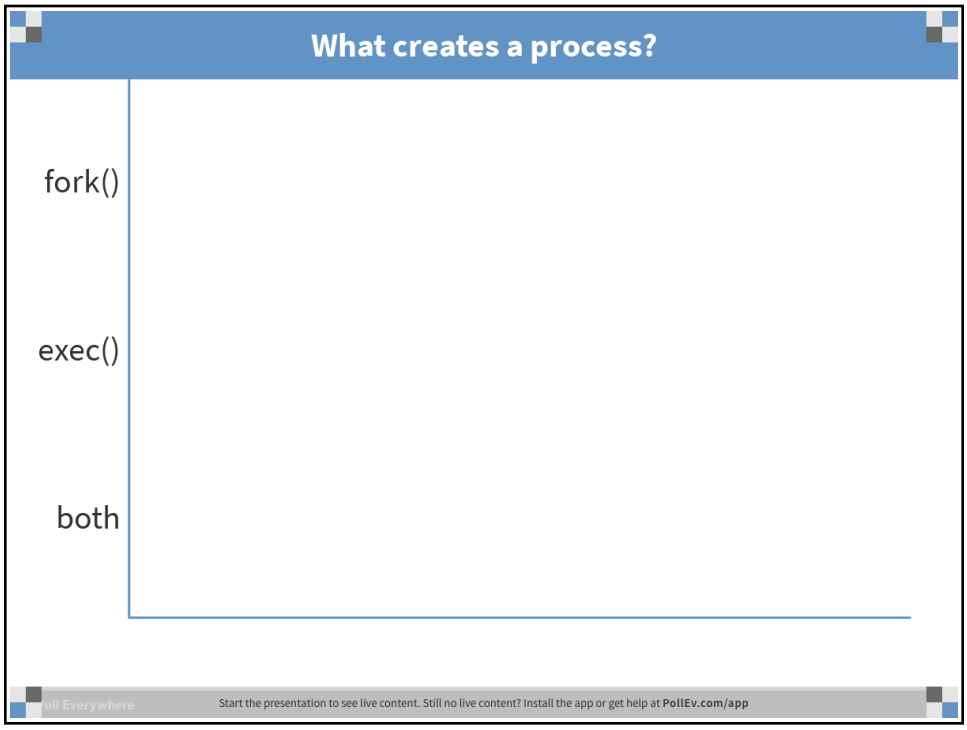
Example: fork() and exec()



Example

```
int main() {
    int pid = fork();
    int status;
    if (pid != 0) {
        wait(&status);
        printf("Parent's child's status is %d\n", status);
    } else {
        printf("Child: My process id is %d\n", getpid());
        printf("Child: My parent process id is %d\n",
                    getppid());
        char *args[]={"ps", NULL};
        execvp(args[0], args);
        printf("Did I get here?\n");
    }
}
```

exec_example.c



Question

What creates a process?

A. fork()
B. exec()
C. both

Exec changes the existing process to run a different program

Zombie (or defunct) Processes

- Process has terminated
  - All of process's resources are cleaned up
  - EXCEPT its PCB
  - Dead, but not gone…
  - All child processes go into this state, though, likely just briefly
- Parent process has not yet collected its status
  - Parent hasn't completed its call to `wait`

Orphaned Processes

- Parent terminates before the child:
  - In some instances, the child becomes an *orphan process*
    - In UNIX, parent automatically becomes the `init` process
  - In other instances, all children are killed (depending on the shell)
    - Bash kills all child processes when it receives a SIGHUP
- Child can orphan itself to keep running in the background
  - nohup command (also prevents it from being killed when SIGHUP is sent)

The Unix Shell

- When you log in to a machine running Unix, the OS creates a shell process for you to use
- Every command you type into the shell is a *child* of your shell process
  - For example, if you type `ls`, the OS forks a new process and then execs `ls`

If you type an & after your command, Unix will run the process in parallel with your shell, otherwise your next shell command must wait until the first one completes.

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:

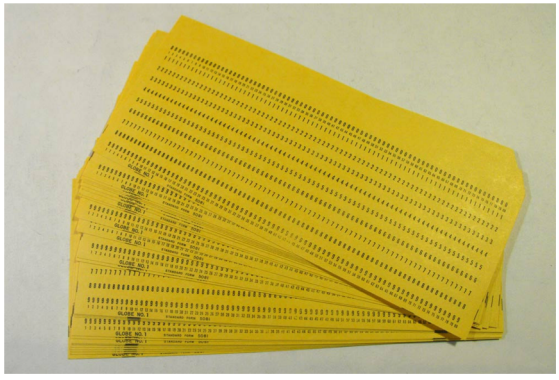
```
kill -9 <PID>
```

Summary of Process Management

- A process is a unit of execution
- Processes are represented as Process Control Blocks in the OS
- At any time, a process is either New, Ready, Blocked, Running, or Terminated
- Processes are created and managed through system calls
  - Fork, exec, wait,

Before Processes (or even OSes)

- Feed in program
- Wait for output
- Feed in next one…



What are the benefits of this **model**?

Run programs ***serially***

Looking Ahead

- Project 1 – Due Tuesday
- Monday: Scheduling Processes
- Project 2 coming out next week