,160 /s

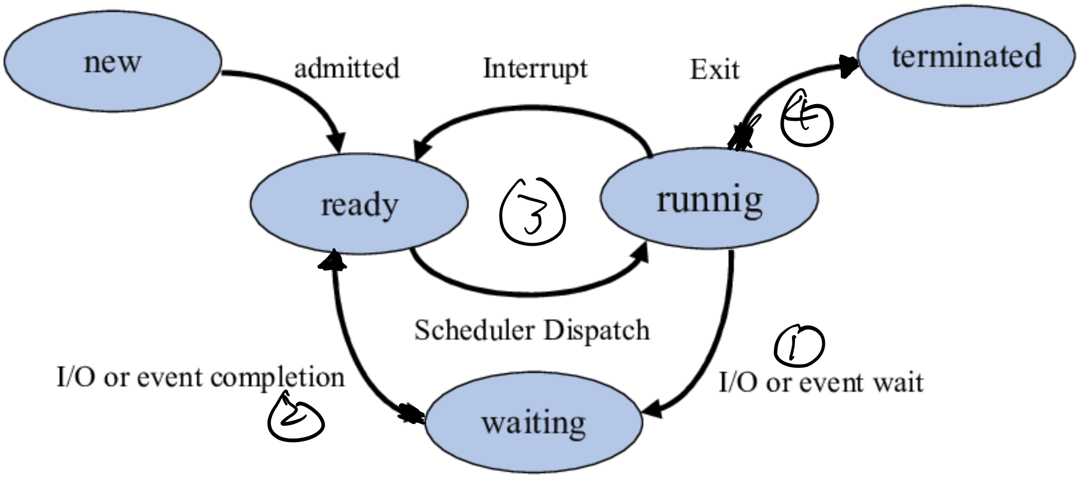
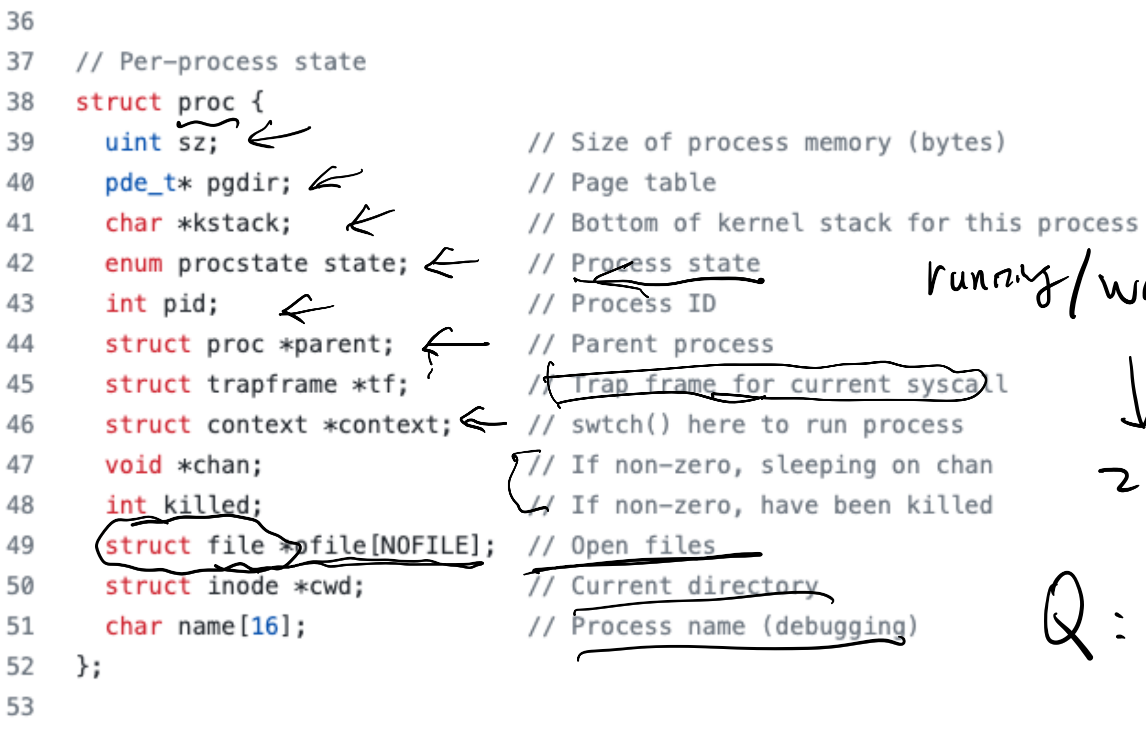


Fig13 from “Priority based round robin (PBRR) CPU scheduling algorithm”

preemptive : ① ~ ④

non - - - - - : ①, ④



```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

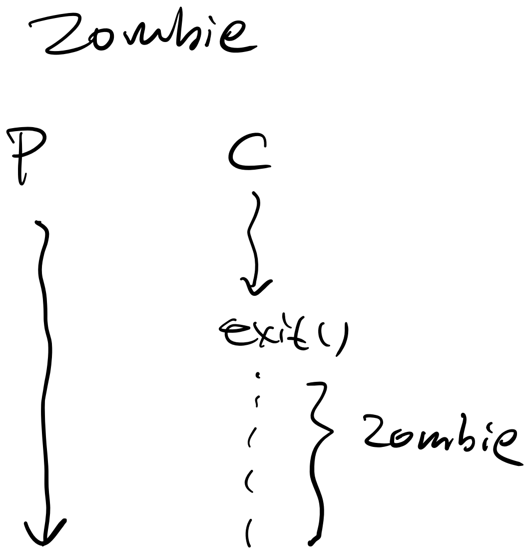
running / waiting / ready

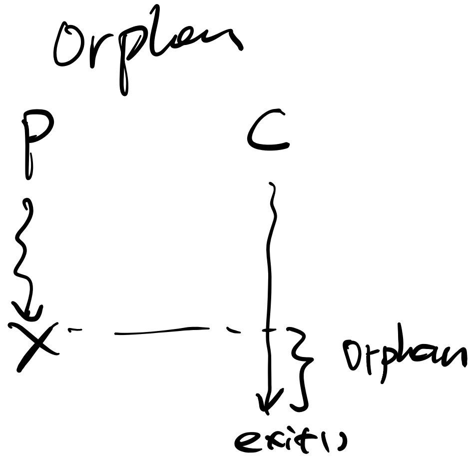
↓

zombie

Q: orphan?

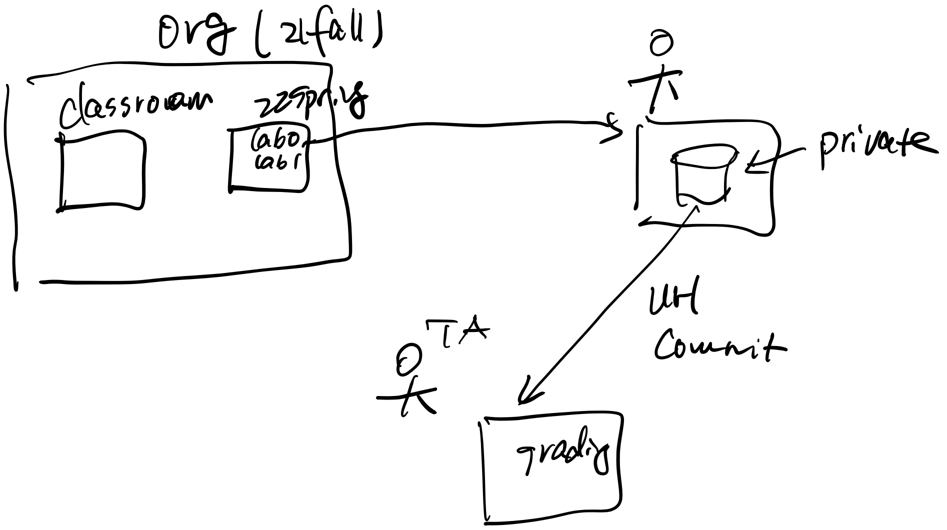
Borrowed from xv6: https://github.com/mit-pdos/xv6-public/blob/eeb7b415dbcb12cc362d0783e41c3d1f44066b17/proc.h





1. Shell internal continued & discussions
2. Implementation of processes
3. Context switch intro
4. Scheduling intro
5. Scheduling disciplines (part I)
   FIFO/SJF

"21fall" ~~"22spring"~~



Lab1

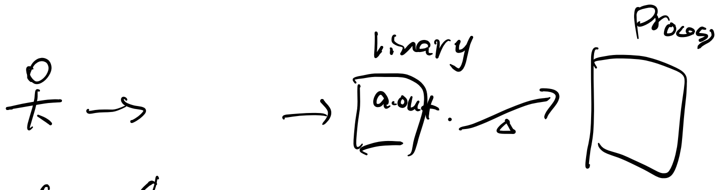
---

fork bomb

fork() / wait() / exec()
execve()

file descriptor

0/1/2



ls > file.txt

cat [illegible].txt | shuf -n 1

CS5600
Handout week03a

The handout is meant to:

--illustrate how the shell itself uses syscalls

--communicate the power of the fork()/exec() separation

--give an example of how small, modular pieces (file descriptors, pipes, fork(), exec()) can be combined to achieve complex behavior far beyond what any single application designer could or would have specified at design time.

1. Pseudocode for a very simple shell

```
while (1) {
  write(1, "$ ", 2);
  readcommand(command, args); // parse input
  if ((pid = fork()) == 0) {  // child?
    execve(command, args, 0);
  } else if (pid > 0) {       // parent?
    wait(0);                  //wait for child
  } else {
    perror("failed to fork");
}}
```

2. Now add two features to this simple shell: output redirection and backgrounding

By output redirection, we mean, for example:

```
$ ls > list.txt
```

By backgrounding, we mean, for example:

```
$ myprog &
$
```

```
while (1) {
  write(1, "$ ", 2);
  readcommand(command, args);   // parse input
  if ((pid = fork()) == 0) {    // child?
    if (output_redirected) {
      close(1);
      open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
    }
    // when command runs, fd 1 will refer to the redirected file
    execve(command, args, 0);
  } else if (pid > 0) {       // parent?
    if (foreground_process) {
      wait(0);                //wait for child
    }
  } else {
    perror("failed to fork");
  }
}
```

fork/exec
separation

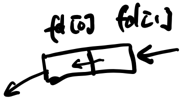
3. Another syscall example: pipe()

The pipe() syscall is used by the shell to implement pipelines, such as

```
$ ls | sort | head -4
```

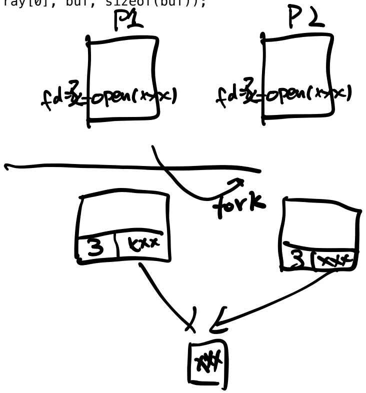
We will see this in a moment; for now, here is an example use of pipes.

```
// C fragment with simple use of pipes

int fdarray[2];
char buf[512];
int n;

pipe(fdarray);
write(fdarray[1], "hello", 5);
n = read(fdarray[0], buf, sizeof(buf));
// buf[] now contains 'h', 'e', 'l', 'l', 'o'
```



4. File descriptors are inherited across fork

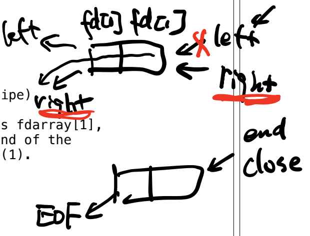
```
// C fragment showing how two processes can communicate over a pipe

int fdarray[2];
char buf[512];
int n, pid;

pipe(fdarray);
pid = fork();
if(pid > 0){     // parent
  write(fdarray[1], "hello", 5);
} else {         // child
  n = read(fdarray[0], buf, sizeof(buf));
}
```

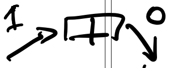


5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe().

```
// Pseudocode for a Unix shell that can run processes in the
// background, redirect the output of commands, and implement
// two element pipelines, such as "ls | sort"

void main_loop() {

  while (1) {
    write(1, "$ ", 2);
    readcommand(command, args);     // parse input
    if ((pid = fork()) == 0) {      // child?
      if (pipeline_requested) {
        handle_pipeline(left_command, right_command)
      } else {
        if (output_redirected) {
          close(1);
          open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
        }
        exec(command, args, 0);
      }
    } else if (pid > 0) {           // parent?
      if (foreground_process) {
        wait(0);                    // wait for child
      }
    } else {
      perror("failed to fork");
    }
  }
}

void handle_pipeline(left_command, right_command) {

  int fdarray[2];

  if (pipe(fdarray) < 0) panic ("error");
  if ((pid = fork ()) == 0) {     // child (left end of pipe)

    dup2 (fdarray[1], 1);         // make fd 1 the same as fdarray[1],
                                  // which is the write end of the
                                  // pipe. implies close (1).
    close (fdarray[0]);
    close (fdarray[1]);
    parse(command1, args1, left_command);
    exec (command1, args1, 0);

  } else if (pid > 0) {           // parent (right end of pipe)

    dup2 (fdarray[0], 0);         // make fd 0 the same as fdarray[0],
                                  // which is the read end of the pipe.
                                  // implies close (0).
    close (fdarray[0]);
    close (fdarray[1]);
    parse(command2, args2, right_command);
    exec (command2, args2, 0);

  } else {
    printf ("Unable to fork\n");
}}
```

left | right





Cat student.txt | shuf

---

6. Commentary

Why is this interesting? Because pipelines and output redirection are accomplished by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors. That is, the *identical code* for "ls" can result in printing to the screen ("ls -l"), writing to a file ("ls -l > output.txt"), or getting ls's output formatted by a sorting program ("ls -l | sort").

This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of ls would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

What makes it work is that the author of ls expressed their code in terms of a file descriptor:
write(1, "some output", byte_count);
This author does not, and cannot, know what the file descriptor will represent at runtime. Meanwhile, the shell has the opportunity, *in between fork() and exec()*, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.

- good abstraction
  - file descriptor
  - 0/1/2
  - fork/exec sep
- fork() today

cow

"Fork today is a convenient API for a single-threaded process with a small memory footprint and simple memory layout that requires fine-grained control over the execution environment of its children but does not need to be strongly isolated from them. In other words, a shell."

——"A fork() in the road"
https://www.microsoft.com/en-us/research/uploads/prod/2019/04/fork-hotos19.pdf

```
print("hello world");
fork();
print("\n");
```

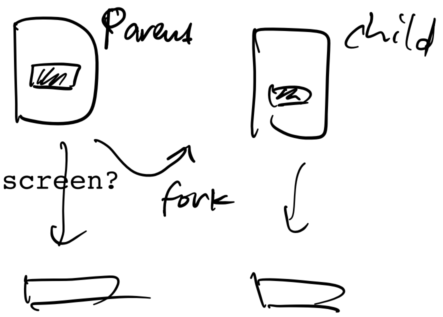
QUESTION: what do you expect to see on screen?

A:
hello world\n
hello world\n

2

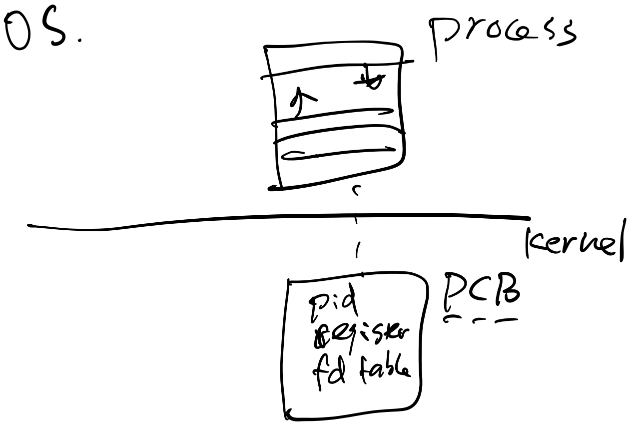
B:
hello world\n
\n

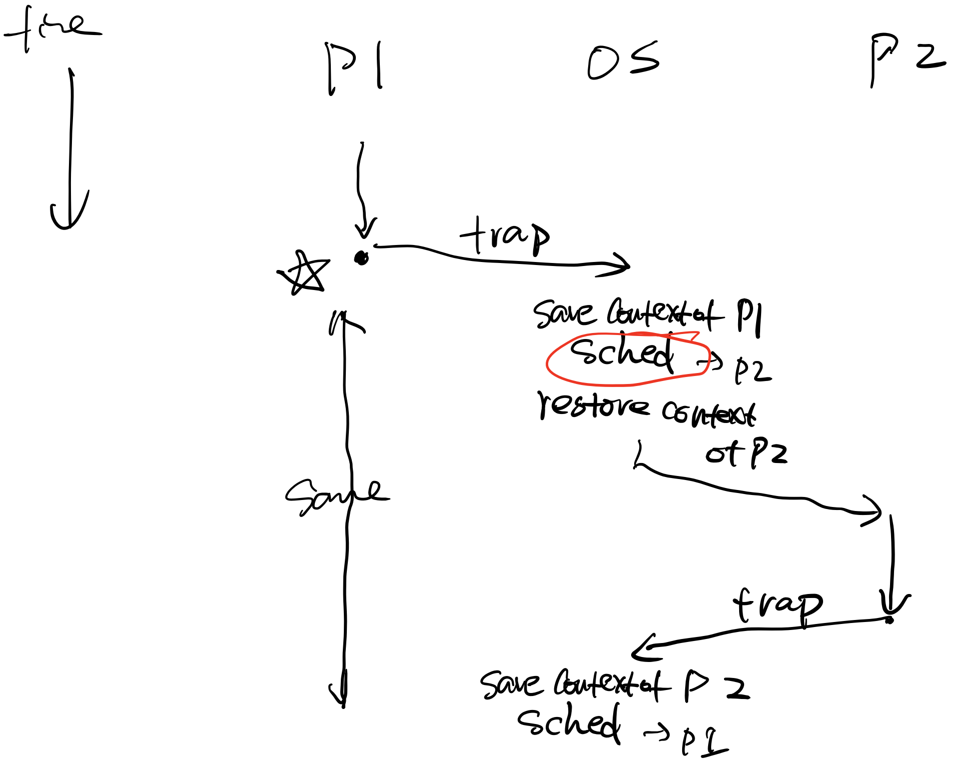
1



onlinegdb.com

OS.

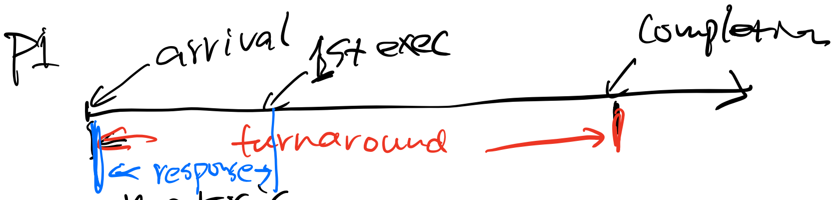




OS
Scheduling

Scheduling game.

- 1 CPU
- multiple process: P1 P2 ... Pn
  - arrival time
  - run to completion time



- metric
  - turnaround time: (c − a)
  - response time: (1st − a)
  - throughput:
  - fairness: no starvation
- Assumptions:
  - I/O
  - ignore context switch cost