Processes

Module 06

What is a process?

A process is a ***running*** program, containing:

- Code (the actual program instructions)
- State
    - Program Counter (which instruction is it executing?)
    - Data
        - Register values
        - Stack
        - Heap

What does it look like in memory?

Process Creation

- When the computer boots, its starts the core OS process.
  - The OS will likely create many others
  - Some foreground (the window manager)
  - Some background (daemons)
- The OS will manage **all** processes
  - Type ps on UNIX, use Task Manager on Windows
- Processes can create new (child) processes

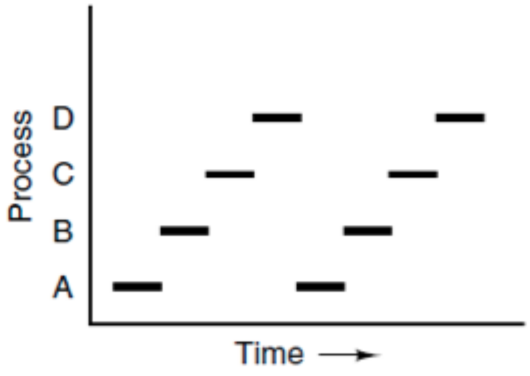
Process Creation

- In POSIX, we have a single system call to create a new process: `fork`
  - Windows has `CreateProcess`

- Calling fork results in a new process being created, with its own **address space**

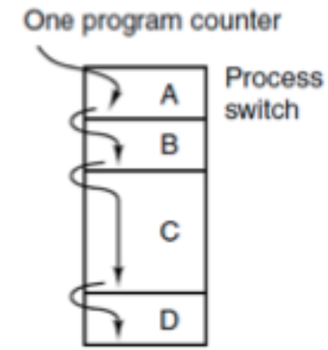
Many Processes

- The Operating System’s **scheduler** makes sure to switch back and forth between processes, to ensure all have their share*

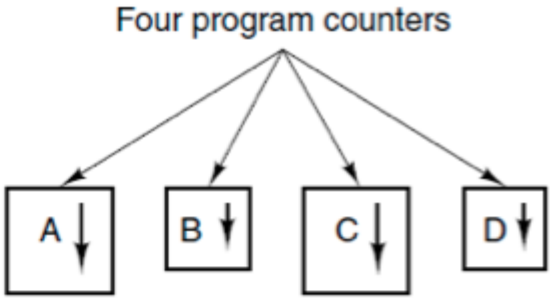


(c)

Real vs. Conceptual



The real CPU has only on PC, we **switch** between processes by setting it into a given process's memory



Conceptually, we can think of each process having its own **logical** PC (and registers)

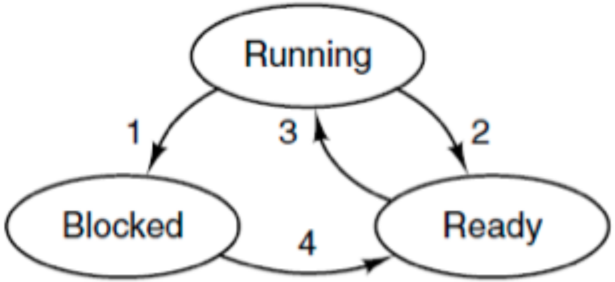
Process Termination

- Processes can voluntary exit
  - POSIX: `exit()`
  - Windows: `ExitProcess()`
- The operating system can *kill* processes
  - If a process executes an illegal instruction, the OS can kill it
  - Another program (if it has authorization) can also kill a process
    - POSIX: `kill(pid)`
    - Windows: `TerminateProcess(pid)`

Process Hierarchies

- Each process has a unique ID
  - POSIX uses integers
  - Windows uses Handles
- When a process creates another process, the new process’s ID (pid) is returned
  - A parent / child relationship is formed
  - Parent can kill off child processes

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

From the OS perspective:

- Only one process “running”
- Many processes can be in the “ready queue”
- Many processes can be blocked, waiting on many things

Process Implementation

- An OS maintains a **Process Table**
  - Often called Process Control Block

- The Process Table must hold enough information to restart a process which has been suspended (ready or blocked)
  - Like what?

Process Table Data

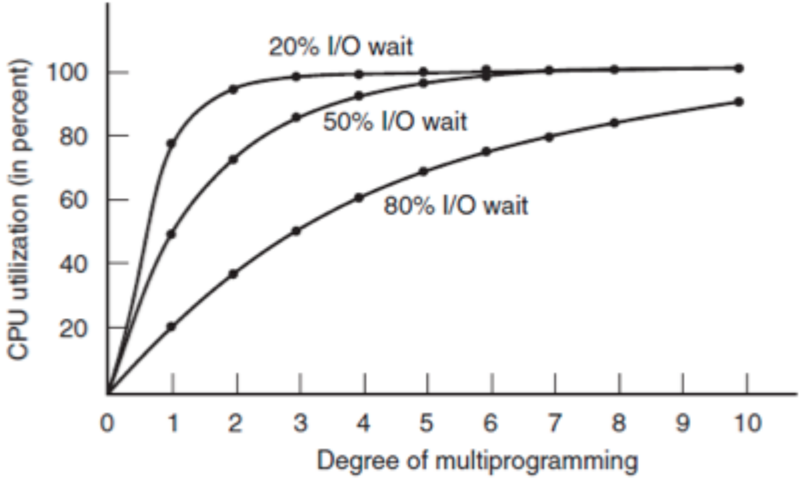
A Process Table will need *at least* the following information about each process

- PC, Stack Pointer, register contents
- Memory allocation / address
- status of open files
- accounting and scheduling information

Multiprogramming

Its critical that an OS can run other processes when one process waits for I/O!

CPU Utilization is paramount!



Programming with Processes

Let's look at the details of the fork() system call

- When process A calls fork, a new process is created ***while process A is waiting for fork to return***.
- Process A's entire address space is copied
- The new process begins executing at the same location (in the copy) as process A
- So… both the parent and child are executing **the same code, at the same place**

What?

```
int pid;
printf("I’m here\n”);
pid = fork();

if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

What?

```
int pid;
printf(“I’m here\n”);                Only one process active
pid = fork();

if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

What?

```
int pid;
printf(“I’m here\n”);
pid = fork();         One process calls this
                      Two processes are active when fork returns
if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

What?

```
int pid;
printf(“I’m here\n”);
pid = fork();
```

pid is not the same in both processes!

```
if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

fork returns 0 to the child process

fork returns the pid (not zero) of the child to the parent

What?

```
int pid;
printf("I'm here\n");
pid = fork();

if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

fork returns 0 to the child process

fork returns the pid (not zero) of the child to the parent

What?

```
int pid;
printf("I'm here\n");
pid = fork();

if ( pid == 0 ) {
  // this is the child process
  printf("Child\n")
} else {
  // this is the parent process
  printf("Parent\n")
}

printf("We're here!\n");
```

Both processes get here - prints twice!

Another Example

Lets take a look at `forks.c` to see more clearly how address spaces work.

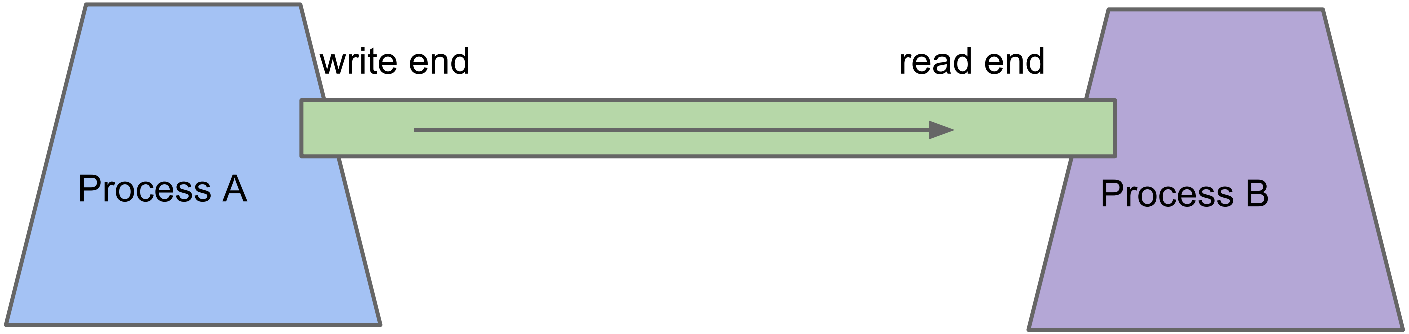
Inter-process Communication (IPC)

We’ve seen that data can flow from parent to child at child creation - but not afterwards

IPC is a big topic, but we’ll now introduce some really basic functionality - **pipes**.

(POSIX) Pipes

Pipes are ONE WAY channels that **any** process can read/write from if the have its *id*.



A pipe represents a FIFO queue of bytes

Working with Pipes

```
int pfd[2];  // file descriptors

int result = pipe(pdf);  // creates a pipe, returns status
```

After pipe call `pfd[0]` contains descriptor for read end of pipe, `pfd[1]` contains the descriptor for write end.

Use read and write system calls to read and write pipe. These calls require an fd, which is the file descriptor (`pfd[0]` for read, `pfd[1]` for write.

```
int read(int fd, char * buffer, int maxlen)
int write (int fd, char * buffer, int numBytes);
```

Pipe Example

Lets take a look at `pipe_example.c` to see how this works in practice.

forks are messy

If you are finding forked code confusing, join the crowd!

- No one finds it pleasant to imaging multiple processes running through the same code
- It leads to ***lots of mistakes*** - usually do to the following pattern:

bad fork code...

```
do {
    cout << “Enter a command:  “;
    cin  >> command;
    if (command == quit )
        done = true;
    pid = fork();
    if (pid == 0 )
        do(command);
    else
        do(something else);
} while (!done);
```

Let’s discuss exactly why this is so horrible...

Using exec

```
int execlp(const char *file, const char *arg, ...);
```

Documentation (http://linux.die.net/man/3/execlp)

The initial argument for these functions is the name of a file that is to be executed.

The const char *arg and subsequent ellipses in the execl(), execlp(), and execle() functions can be thought of as arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. **The first argument, by convention, should point to the filename associated with the file being executed**. The list of arguments must be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast (char *) NULL.

Exec Example

Lets take a look at `prompt.c` and `echo.c` to see how this works in practice.

Where's Windows in all this?

- `CreateProcess` is basically POSIX's `fork` + `exec`, all rolled up into one
- Probably an easier API, although less flexible
- Windows has several types of pipes - the most common is a "named pipe".

Up next...

Please read Chapter 2.2 on Threads