Week 2: The Process abstraction

TABLE OF CONTENTS

Weekly Summary and Where are we?

Topics

- **Last Week:** Introduction to systems in general
- **This Week:** Introduction to the Process abstraction and how to use it in C
- **Next Week:** How the OS schedules processes (and other things)

Assignments

- **Lab 0** (setting up your dev environment) should be submitted already!
- **Lab 1** (getting familiar with C, `gdb`, `vim`, `ctags`, etc) is due on Friday
- **Lab 2** (building `ls`) is out; due 30 Jan 2026.
- **HW 1** is due on Friday

Reading Summary

- You should have caught up on Chap 1 and Chap 2 (Introduction to Operating Systems) from OSTEP last week
- Reading for this week was Chap 4 (the Process Abstraction) and Chap 5 (the Process API in C) from OSTEP
- **Chap 4** summarizes the need for a process abstraction
  - A ***process*** is an abstraction for a program that can be run
  - Once a process can be abstracted, the OS can run different processes at different times to most efficiently use the underlying hardware

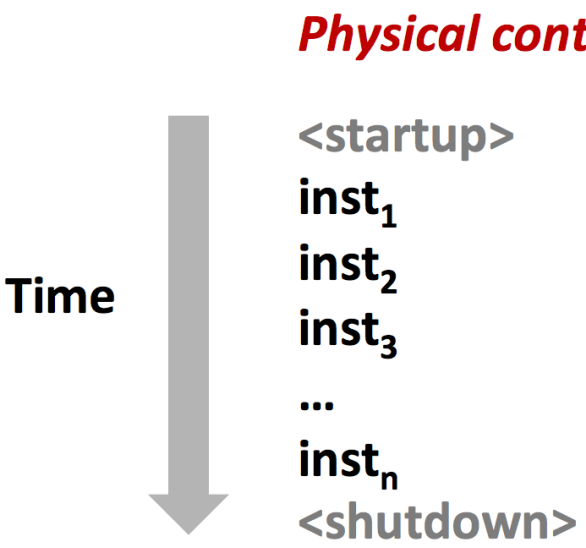
    - This is the first "thing" we're virtualizing
    - A process abstraction contains its state; all the things we need to know about it running
        - What lives in each register
        - process ID
        - parent process
        - What lives in each memory location
        - ...
    - The process abstraction lives as an actual data structure that is provided
        - This is part of the API: Application Programming Interface that allows anyone to work with processes as long as they use the agreed-upon structures and format
- **Chap 5** introduces the Process API for Unix
    - `fork()`: Runs a new instance of the current program
    - `wait()`: Tells this process
    - `exec()`

Processes

Definition: A process is an instance of a running program.

- One of the most profound ideas in computer science
- Not the same as "program" or "processor"

A processor ONLY reads a sequence of instructions from start to finish; this is the ***control flow***.



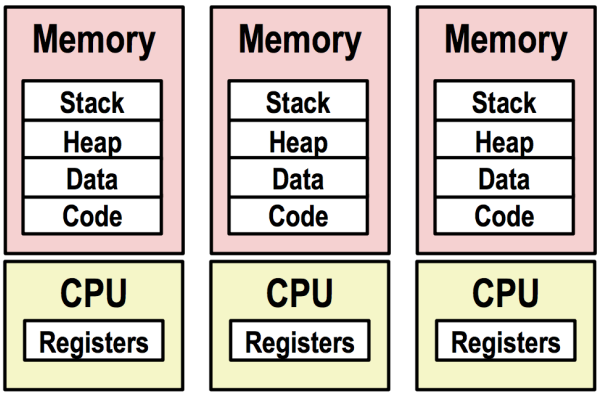
A processor runs a process.

Process provides each program with two key abstractions:

- ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
- ***Private address space***
    - Each program seems to have exclusive use of main memory
    - Provided by kernel mechanism called ***virtual memory***

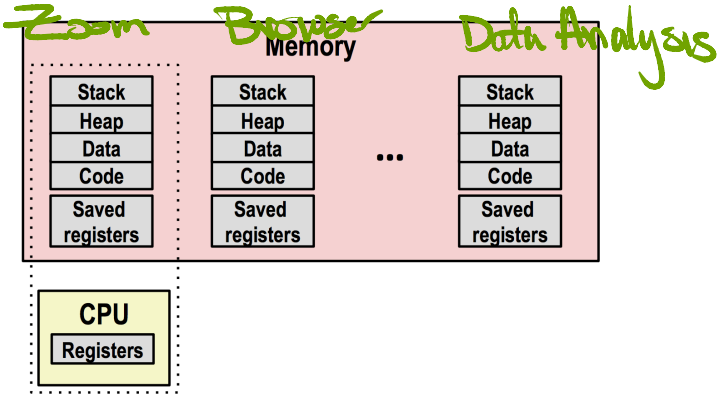
The illusion of *Multiprocessing*

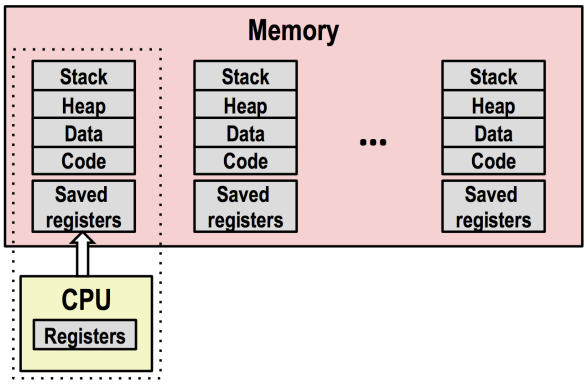
- See `top`

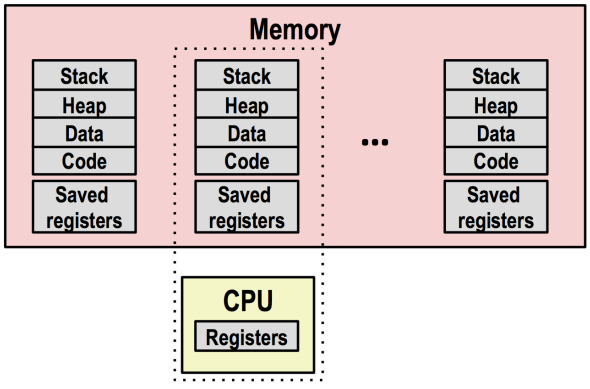


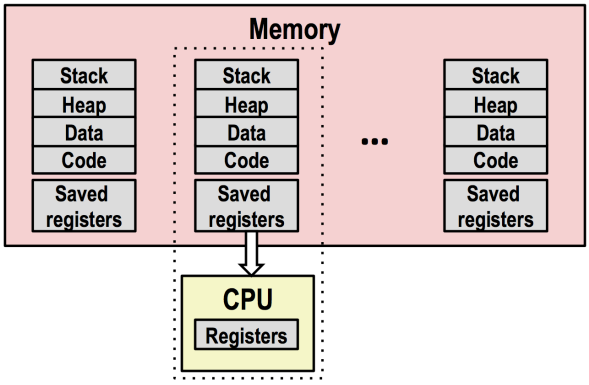
The reality, traditionally:

- Computers have one processor that gives the illusion of being able to run multiple programs/processes at once
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory





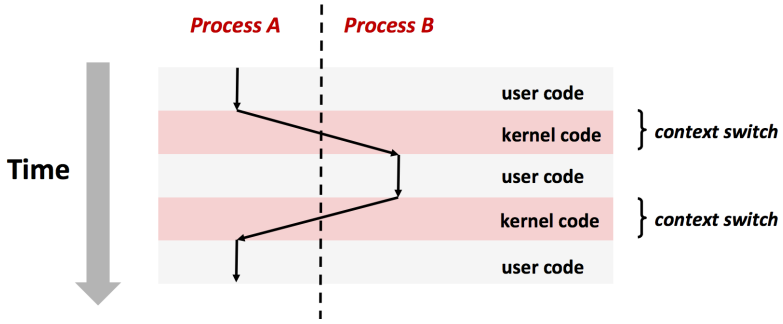




Context switching

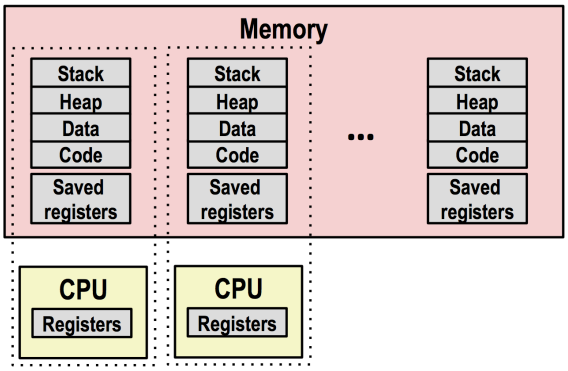
Processes are managed by a shared chunk of memory resident OS code called the kernel

- Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a ***context switch***



Modern systems are *multicore*

- Machines have multiple processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by ***kernel***



Error Handling in Unix/Linux

On error, Linux system-level functions typically return `-1` and set global variable `errno` to indicate cause.

- *Hard and fast rule:*
  - You must check the return status of every system-level function
  - Only exception is for the few functions that return void
  - Example:

```
if ((pid = fork()) < 0) {
            fprintf(stderr, "fork error: %s\n", strerror(errno));
            exit(0);
}
```

Can utilize an error-reporting function:

```
void unix_error(char *msg) /* Unix-style error */
{
            fprintf(stderr, "%s: %s\n", msg, strerror(errno));
            exit(0);
}

if ((pid = fork()) < 0)
            unix_error("fork error");
```

Can simplify more by wrapping system calls with an error-handling function:

```
pid_t Fork(void)
{
            pid_t pid;
            if ((pid = fork()) < 0)
            unix_error("Fork error");
            return pid;
}

pid = Fork();
```

Working with Processes

Things we want to do with processes

- Get process IDs
- Create and terminate processes
- Reap child processes
- Kill the Zombies!
- Synchronizing with child processes

Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process
- `pid_t getppid(void)`
  - Returns PID of parent process

From a programmer's perspective, we can think of a process as being in one of three states:

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
- **Stopped**

    - Process execution is suspended and will not be scheduled until further notice (future class we'll study signals)
- **Terminated**
    - Process is stopped permanently

Terminating processes

- Process becomes terminated for one of three reasons:
    - Receiving a signal whose default action is to terminate (future class)
    - Returning from the `main` routine
    - Calling the `exit` function
- `void exit(int status)`
    - Terminates with an exit status of `status`
    - Convention: normal return status is 0, nonzero on error
    - Another way to explicitly set the exit status is to return an integer value from the main routine
- `exit` is called once but never returns.

Creating processes

- Parent process creates a new running child process by calling `fork`
- `int fork(void)`
    - Returns 0 to the child process, child's PID to parent process
    - Child is *almost* identical to parent:
        - Child get an identical (but separate) copy of the parent's virtual address space.
        - Child gets identical copies of the parent's open file descriptors
        - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called once but returns twice

`fork` examples

```
int main()
{
            pid_t pid;
            int x = 1;

            pid = fork(); // Consider using our error-catching Fork() from above
            if (pid == 0) { /* Child */
                        printf("child : x=%d\n", ++x);
                        exit(0);
            }
            /* Parent */
            printf("parent: x=%d\n", --x);
            exit(0)
}
```

- Call once, return twice
- Concurrent execution
    - Can't predict execution order of parent and child
- Duplicate but separate address space
    - x has a value of 1 when fork returns in parent and child
    - Subsequent changes to x are independent
- Shared open files
    - `stdout` is the same in both parent and child

Reaping child processes

Idea:

- When process terminates, it still consumes system resources
  - Examples: Exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid == 1)
- So, only need explicit reaping in long-running processes
  - e.g., shells and servers

Zombie Example

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by init

Non-Terminating Child Example

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by init

`wait` : Synchronizing with Children

- Parent reaps a child by calling the wait function
- `int wait(int *child status)`
  - Suspends current process until one of its children terminates
  - Return value is the pid of the child process that terminated
  - If child status != NULL , then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in wait.h
    - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
    - Details... look them up if you're interested :)

Synchronizing with children

```
void fork() {
            int child_status;
            if (fork() == 0) {
                        printf("HC: hello from child\n");
                        exit(0);
            } else {
                        printf("HP: hello from parent\n");
                        wait(&child_status);
                        printf("CT: child has terminated\n");
            }
            printf("Bye\n");
}
```

Feasible output: `HC -> HP -> CT -> Bye`

Infeasible output: `HP -> CT -> Bye -> HC`

Another `wait` example

```
void fork_wait() {
            pid_t pid[N];
```

```
        int i, child_status;
        for (i = 0; i < N; i++)
                if ((pid[i] = fork()) == 0) {
                exit(100+i); /* Child */
        }


        for (i = 0; i < N; i++) { /* Parent */
                pid_t wpid = wait(&child_status);
                if (WIFEXITED(child_status))
                        printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
                else
                        printf("Child %d terminate abnormally\n", wpid);
        }
}
```

- If multiple children completed, will take in arbitrary order
- Can use macros `WIFEXITED` and `WEXITSTATUS` to get information about exit status

Summary

- Processes are the abstraction that allows operating systems to utilize hardware resources efficiently
- Processes have a context, which is basically all their data and instructions, and where those items live in memory/hardware
- Before a context switch, the kernel will save a process context for the process currently being executed.
  - After the current process is stored, the next process is restored from it's stored context
  - The new process continues its execution
- As programmers, we can make use of processes by using `fork()` and `exec()` in C.
- Once we start creating new processes, we need to worry about whether our instructions are being executed concurrently or sequentially

Building a Shell

When we speak about "the command line", we're actually referring to ***the shell***. The *shell* is a program that takes keyboard commands and passes them to the operating system to execute. `bash` (Bourne Again Shell) is a common shell included with most Linux distributions; `zsh` (Z Shell) is a default shell available on Macs.

Now that we know how to work with processes via `fork()` and `exec()`, we can actually build our own shell at this point.

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
```

wait_status

ls <e>

```
        {
                perror("Failed to fork");
        }
}
```

Adding output redirection and backgrounding.

Output redirection is something like `ls > list.txt`, where we redirect the output of the `ls` program to the file `list.txt`. Backgrounding means something like `myprog &`, which allows the program to run in the background, not taking up the shell's prompt.

```
while(1)
{
        write(1, "$ ", 2);
        readcommand(command, args); // parse input
        if ((pid = fork()) == 0)
        {
                // child
                if (output_redirected)
                {
                        close(1);
                        open(redirect_file, 0_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                // When the 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");
        }
}
```

Adding piping

Another example: the `pipe()` syscall. This allows something like `$ ls | sort | head -4`.

```
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'
```

File descriptors are inherited across `fork()`.

```c
// 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
{
        n = read(fdarray[0], buf, sizeof(buf));
}
```

* Does Read block ?

Putting it all together

Putting it all together: implement a shell using `fork()`, `exec()` and `pipe()`.

```c
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, 0_CREAT | O_TRUNC | O_WRONLY, 0666);
                                }
                                exec(command, args, 0);
                        }
                } else if (pid > 0)
                {
                        // parent
                        if (foreground_process)
                        {
                                wait(0);
                        }
                } else {
                        perror("Failed to fork");
                }
        }
}
```

ls | sort

```
void handle_pipeline(left_command, right_command)
{
            int fdarray[2];
            if (pipe(fdarray) < 0) panic ("error");

            if ((pid = fork()) == 0)
            {
                        // child; left command
                        dup2(fdarray[1], 1);// make fd 1 the same as fdarray[1], which is the write end of the pip
                                    // implies close(1)

                        close(fdarray[0]);
                        close(fdarray[1]);
                        parse(command1, args1, left_command);
                        exec(command1, args1, 0);
            }
            else
            {
                        if (pid > 0)
                        {
                                    // parent; right command
                                    dup2(fdarray[0]); // make fd 0 the same as fdarray[0], which is the read end of
                                    // implies close(0)

                                    close(fdarray[0]);
                                    close(fdarray[1]);
                                    parse(command2, args2, right_command);
                                    exec(command2, args2, 0);
                        }
                        else
                        {
                                    printf("Unable to fork\n");
                        }
            }
}
```

Why is this interesting?

- Pipelines and output redirection aredone by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors.
- The same code for `ls` can result in:
  - printing to the screen (`ls -l`)
  - writing to a file (`ls -l > output.txt`)
  - getting the output from `ls` formatted by another program (`ls -l | sort`)
- Without redirection, the `ls` implementation would have to anticipate every possible output mode and would build in an interface so the user (client) could specify how the output is treated
- Rather, the author of `ls` wrote it in terms of a file descriptor: `write(1, "some output", byte_count)`.
- Through the use of `fork()` and `exec()`, the shell can make that file descriptor be a pipe, a file, the console, etc.

Summary

- * *

- Process Abstraction
- fork(), waiting for child processes to terminate.
- Write your own exceptions
- Redirecting and piping
- Macro for exit status