



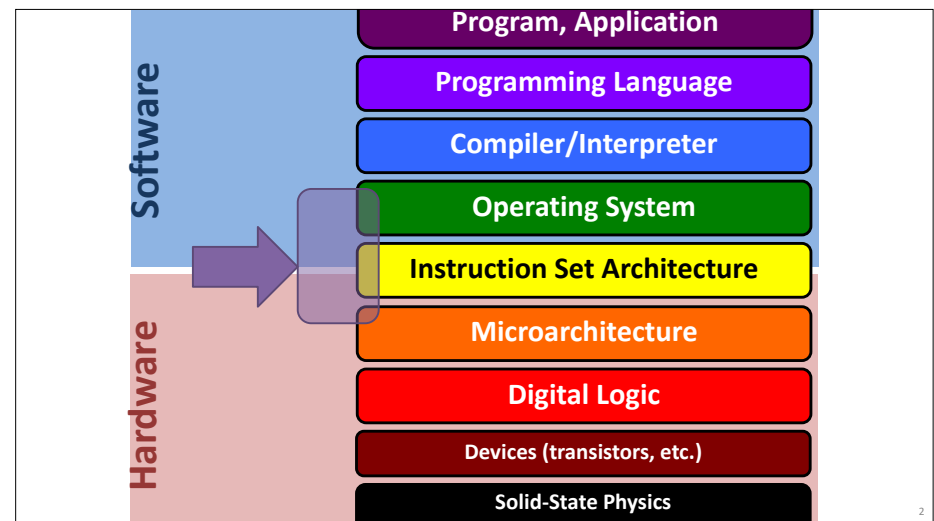
Operating Systems and the Process Model

Process model
Process management

Big picture: how does code run on
real, multiprocessor systems?

<https://cs.wellesley.edu/~cs240/>

1



2

Motivation

Why doesn't this program disable my laptop entirely?

```
int main() {  
    while (1) {  
    }  
}
```

3

Operating Systems

Problems:

- The overall system shouldn't go down for one bad program
- One set of resources, many different software programs!
- The hardware itself varies across computers

Solution: operating system

Manage, abstract, and virtualize hardware resources

Share limited resources among varied software programs

Protect (from both accidental and malicious damage)

Simpler, common interface to varied hardware

4

Operating Systems, a 240 view *barely scraping the surface!*



Key abstractions provided by *kernel*

processes
virtual memory



Virtualization mechanisms and hardware support:

context-switching
exceptional control flow
memory isolation, address translation, paging



stock.adobe.com 5

Processes

Program = code (*static*)

Process = a running program instance (*dynamic*)

code + state (contents of registers, memory, other resources)

Key illusions:

Logical control flow

Each process seems to have exclusive use of the CPU

Private address space

Each process seems to have exclusive use of full memory

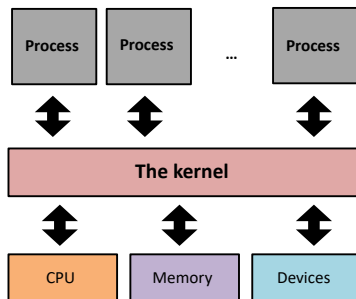
Why? How?

This unit (parts)

Not in detail this semester
But read optional slides & CSAPP!

6

The kernel manages processes



The kernel:

Runs with full machine privilege

On x86: special `%cs` register

Can interrupt processes

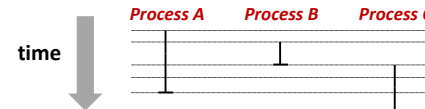
Manages sharing of resources

Is a program (almost*) like any other!

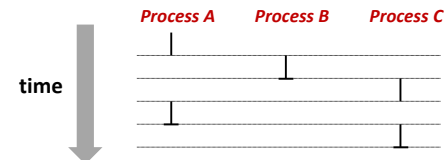
7

Implementing logical control flow

Abstraction: every process has full control over the CPU



Implementation: time-sharing



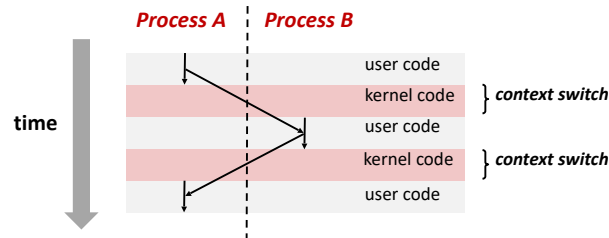
8

Context Switching

Kernel (shared OS code) switches between processes

Control flow passes between processes via **context switch**.

Context =



9

fork

`pid_t fork()`

1. Clone current **parent** process to **create** identical* **child** process, including all state (memory, registers, **program counter**, ...).
2. Continue executing both copies with **one difference**:
 - returns 0 to the **child process**
 - returns **child's process ID (pid)** to the **parent process**

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

fork is unique: called **in one process**, returns **in two processes!**

(once in parent, once in child)

*almost. See man 3 fork for exceptions.

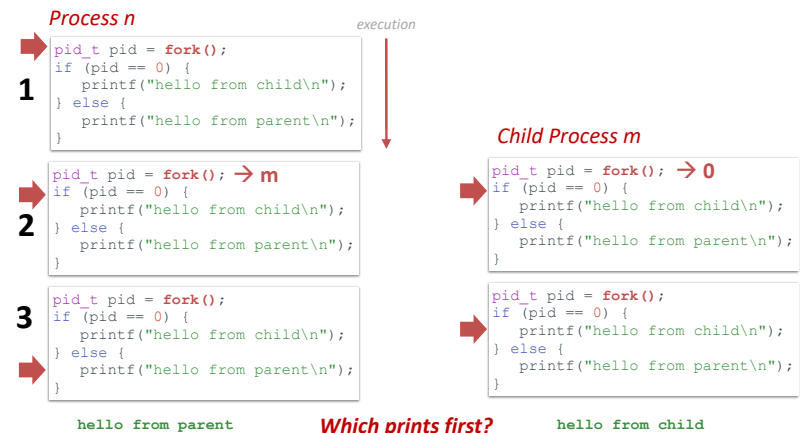
10

Which full line of code is executed twice, once in the parent and once in the child?

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app

Creating a new process with fork



12

Which line prints first?

"hello from parent"

"hello from child"

it depends

they print at the exact same time

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Which line prints first?

"hello from parent"

0%

"hello from child"

0%

it depends

0%

they print at the exact same time

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Recall: what is different about how a call to "fork" returns for the parent vs the child?

The child returns immediately, the parent waits.

The parent returns immediately, the child waits.

The child gets process ID 0, the parent gets non-zero.

The parent gets process ID 0, the child gets non-zero.

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Recall: what is different about how a call to "fork" returns for the parent vs the child?

The child returns immediately, the parent waits.

0%

The parent returns immediately, the child waits.

0%

The child gets process ID 0, the parent gets non-zero.

0%

The parent gets process ID 0, the child gets non-zero.

0%

None of the above

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

fork and private copies

Parent and child continue from **private copies** of same state.

Memory contents (**code**, globals, **heap**, **stack**, etc.),
Register contents, **program counter**, file descriptors...

Only difference: return value from `fork()`

Relative execution order of parent/child after `fork()` undefined

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

17

fork-exec

`fork()` clone current process

`execv()` replace process code and context (registers,
memory)

with a fresh program.

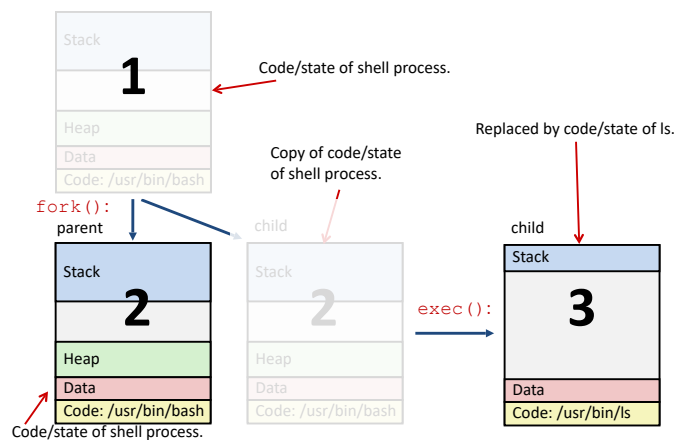
See **man 3 execv**, **man 2 execve**

```
// Example arguments: path="/usr/bin/ls",
// argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char* path, char* argv[]) {
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: exec-ing new program now\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

18

Executing a new program

Running the command `ls` in a shell:



19

execv: load/start a program

```
int execv(char* filename, char* argv[])
```

Loads/starts program in current process:

Executable **filename**

With argument list **argv**

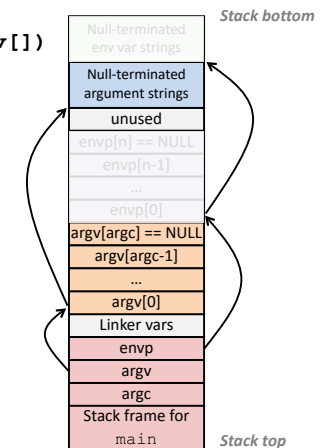
Overwrites code, data, and stack

Keeps pid, open files, a few other items

Does not return

unless error

Also sets up *environment*. See also: `execve`.



20

exit: end a process



```
void exit(int status)
```

End process with status: 0 = normal, nonzero = error.

`atexit()` registers functions to be executed upon exit

21

wait for child processes to terminate



```
pid_t waitpid(pid_t pid, int* stat, int ops)
```

Suspend current process (i.e. parent) until child with `pid` ends.

On success:

Return `pid` when child terminates.

Reap child.

If `stat != NULL`, `waitpid` saves termination reason where it points.

See also: *man 3 waitpid*

22

waitpid example

What is printed, in what order?

ex

```
void fork_wait() {
    int child_status;
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("HC: hello from child\n");
    } else {
        if (-1 == waitpid(child_pid, &child_status, 0)) {
            perror("waitpid");
            exit(1);
        }
        printf("CT: child %d has terminated\n", child_pid);
    }
    printf("Bye\n");
    exit(0);
}
```

23

waitpid example



ex

```
void fork_wait() {
    int child_status;
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("HC: hello from child\n");
    } else {
        if (-1 == waitpid(child_pid, &child_status, 0)) {
            perror("waitpid");
            exit(1);
        }
        printf("CT: child %d has terminated\n", child_pid);
    }
    printf("Bye\n");
    exit(0);
}
```

Printed:

HC: hello from child

Bye

CT: child 1 has terminated

Bye

24

Zombies!



Terminated process still consumes system resources

Reaping with `wait/waitpid`

What if parent doesn't reap?

If any parent terminates without reaping a child, then child will be reaped by `systemd/init` process (`pid == 1`)

What if parent runs a long time? *e.g.*, shells and servers

25

Error-checking

Check return results of system calls for errors! (No exceptions.)

Read documentation for return values.

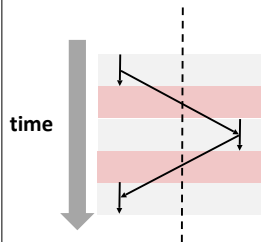
Use `perror` to report error, then exit.

```
void perror(char* message)
```

Print "<message>: <reason that last system call failed.>"

26

Summary



Processes

System has multiple active processes

Each process:

Appears to have total control of the processor

Has isolated access to its own data (usually)

OS periodically "context switches" between active processes

Process management

`fork`, `execv`, `waitpid`

27

Exercise: fork + waitpid

ex

1. Implement the following function using `fork` and `wait`:

```
pid_t fork()
pid_t waitpid(pid_t pid, int* stat, int ops) // Hint: pass 0 for ops

/*
 * Write a C function that creates a child fork that creates a
 * grandchild fork. Make the program print "Hello from grandchild"
 * from the grandchild, then "Hello from child" from the child,
 * making sure these statements happen in this order.
 */
void wait_for_grandchild() {

}
```

28

ex

```
void wait_for_grandchild() {
    int status;
    // Fork once to create child
    pid_t child_pid = fork();
    // Only fork again if in the child thread
    if (child_pid == 0) {
        // Fork again to create grandchild
        pid_t grand_child_pid = fork();
        if (grand_child_pid == 0) {
            // Print from inside the grandchild
            printf("Hello from grandchild\n");
        } else {
            // In the child, wait until the grandchild has printed
            if (-1 == waitpid(grand_child_pid, &status, 0)) {
                perror("waitpid");
                exit(1);
            }
            printf("Hello from child\n");
        }
    } else {
        if (-1 == waitpid(child_pid, &status, 0)) { ... final error check }
    }
}
```