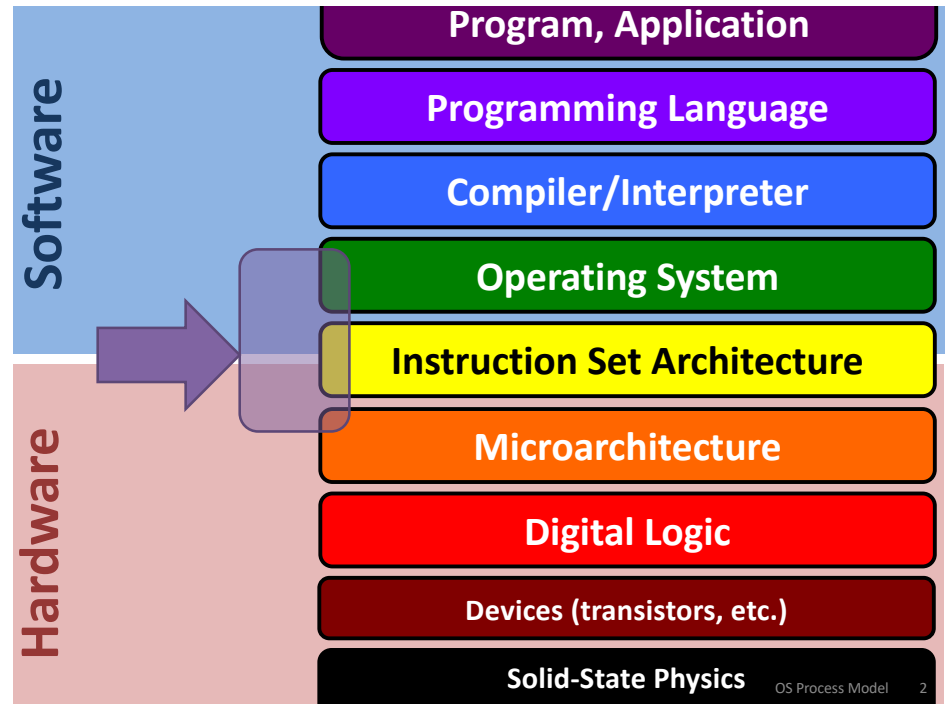




# Operating Systems and the Process Model

Process model  
Process management  
(Unix/Linux/macOS)



## Operating Systems

**Problem: unwieldy hardware resources**  
complex and varied

limited

### Solution: operating system

Manage, abstract, and virtualize hardware resources

**Simpler, common interface** to varied hardware

**Share** limited resources among

**Protect**

## Operating Systems, a 240 view

*barely scraping the surface*

### Key abstractions provided by *kernel*

process

virtual memory

### Virtualization mechanisms and hardware support:

context-switching

exceptional control flow

address translation, paging, TLBs

# 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**

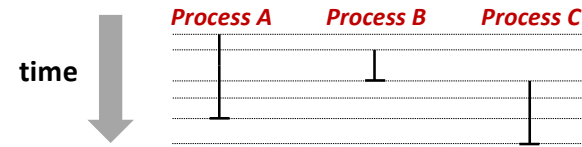
Each process seems to have exclusive use of full memory



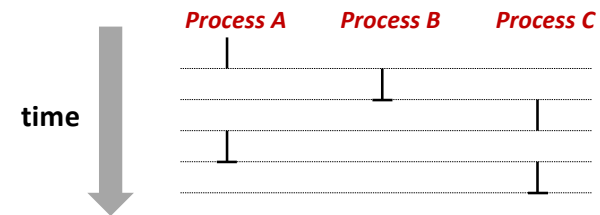
Why? How?

# Implementing logical control flow

**Abstraction:** every process has full control over the CPU



**Implementation:** time-sharing

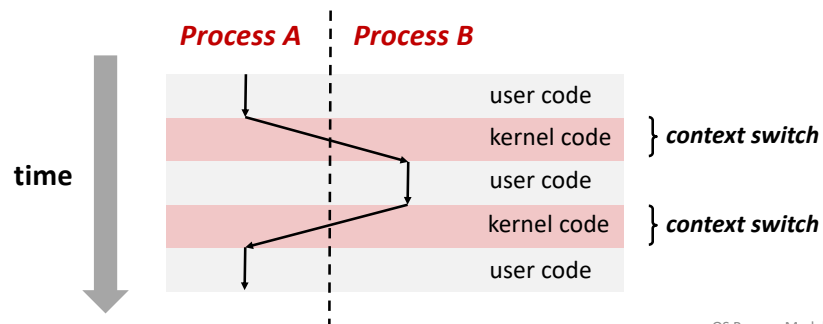


# Context Switching

*Kernel* (shared OS code) switches between processes

Control flow passes between processes via *context switch*.

Context =



# 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.

# Creating a new process with `fork`

Process *n*

```

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

```

```

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

```

```

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

```

Child Process *m*

```

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

```

```

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

```

hello from parent      *Which prints first?*      hello from child

# 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);
}

```

# 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`

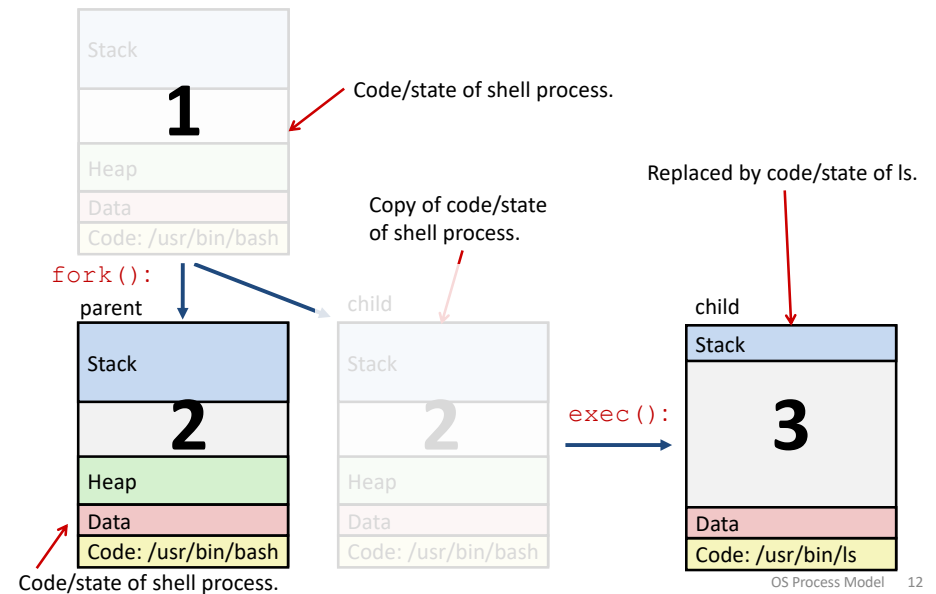
```

// 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");
}

```

Running the command `ls` in a shell:

# Executing a new program



## execv: load/start a program

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

loads/starts program in current process:

Executable `filename`

With argument list `argv`

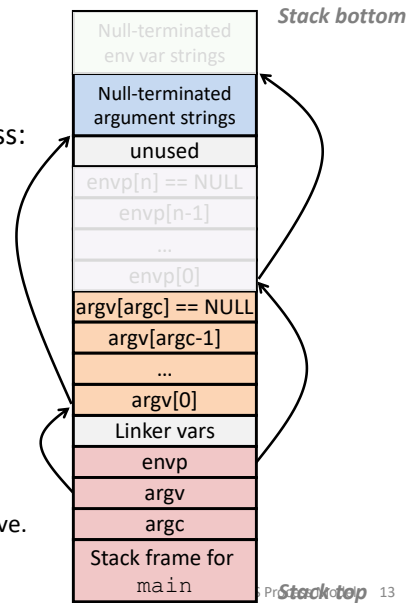
overwrites code, data, and stack

Keeps pid, open files, a few other items

*does not return*

unless error

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



Stack top 13

## exit: end a process

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

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

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

OS Process Model 14

## 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.

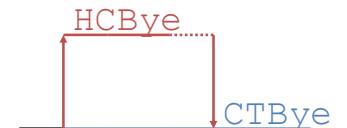
Reap child.

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

See also: `man 3 waitpid`

OS Process Model 15

## waitpid example



```
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);  
}
```

OS Process Model 16

## 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 `init` process (`pid == 1`)

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

## 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.>`"

## Examining processes on Linux (demo)

`ps`

`pstree`

`top`

`/proc`