



CS3281 / CS5281

Process Creation and Control

CS3281 / CS5281

Spring 2024

**Some lecture slides borrowed and adapted from CMU's
"Computer Systems: A Programmer's Perspective"*



Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu



Review

- System calls are how user-level processes request services from the kernel
 - Many kinds of system calls: connect to a network host, read bytes, open a file, close a file, etc
- System calls are supported by special machine-code instructions
 - On x86: int 80h or syscall
 - On RISC-V: ecall
 - These “trapping” instructions cause a lot of hidden work to happen
 - Control flow jumps to the OS kernel
 - The kernel handles the request
 - Control returns to the user-space application
- Today we’ll look at the system calls for:
 - Creating a process
 - Running a program
 - Waiting for a process to terminate

System Call Error Handling

- 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 the handful of functions that return void
- Example:

Good practice in Linux,
but not available in xv6

```
if ((pid = fork()) < 0) {  
    fprintf(stderr, "fork error: %s\n", strerror(errno));  
    exit(1);  
}
```

Obtaining Process IDs

Linux

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

xv6

- `int getpid(void)`
 - Returns PID of current process
- ~~`pid_t getppid(void)`~~
 - No such function exists

Note that `pid_t` is just a signed 32 bit integer on most platforms.
Type is defined for portability.

Creating and Terminating Processes

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 (next lecture when we 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 (next lecture)
 - 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: fork()

- *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 gets 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 Example

- 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

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

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

```
linux> ./fork
parent: x=0
child : x=2
```


Modeling fork() with Process Graphs

- A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program:
 - Each vertex is the execution of a statement
 - $a \rightarrow b$ means a happens before b
 - Edges can be labeled with current value of variables
 - `printf` vertices can be labeled with output
 - Each graph begins with a vertex with no incoming edges
- Any topological sort of the graph corresponds to a feasible total ordering (i.e., valid output)
 - Total ordering of vertices where all edges point from left to right

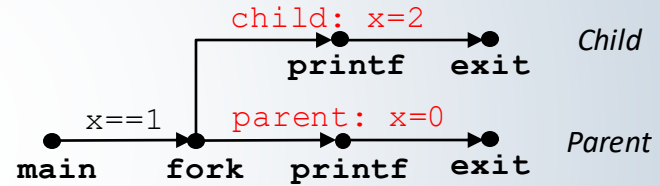
Process Graph Example

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

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

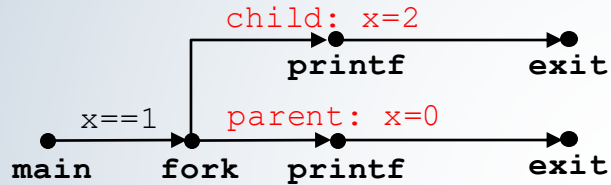
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

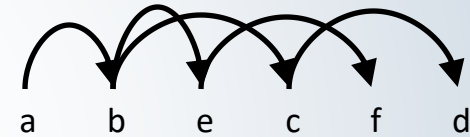


Interpreting Process Graphs

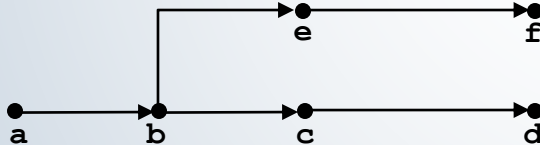
- Original graph:



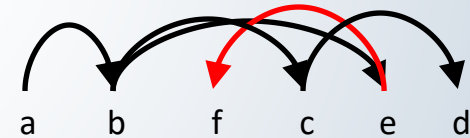
Feasible total ordering:



- Relabeled graph:



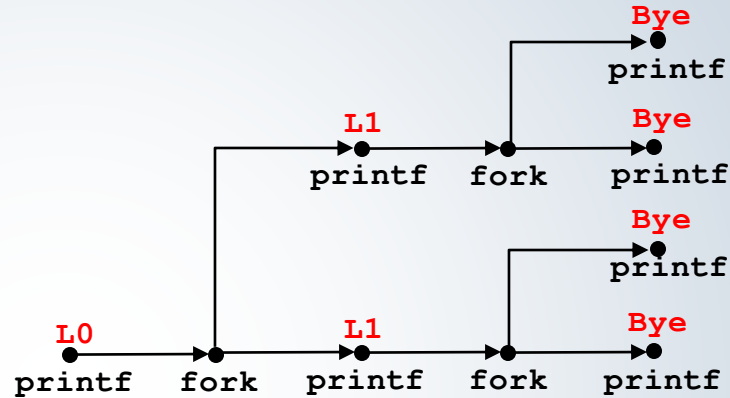
Infeasible total ordering:



fork() Example: Two Consecutive Forks

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

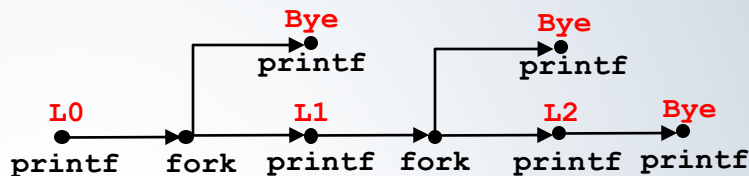
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork() Example: Nested Forks in Parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
L1
Bye
Bye
L2
Bye

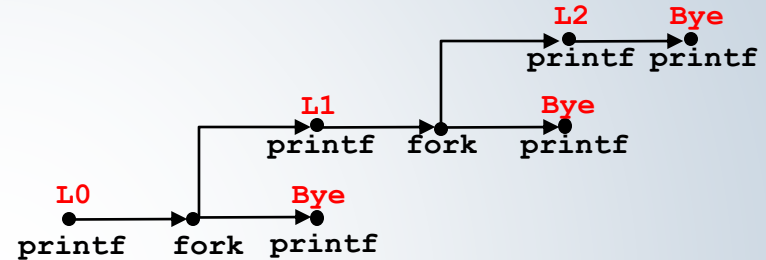
Infeasible output:

L0
Bye
L1
Bye
Bye
L2

fork() Example: Nested Forks in Children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

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

Linux, not xv6



After fork()

- The new process inherits:
 - Process group ID
 - Resource limits
 - Working directory
 - Open file descriptors
 - We will cover these again later -- they help implement pipelines such as:
`find . | grep '\.java' | wc -l` // find all the .java files and tell me how many there are
- But what if we want to execute a different program?

exec(): Loading a New Program

- The `exec()` function loads a new program
 - The existing address space is blown away and loaded with the data and instructions of the new program
 - However, things like the PID and file descriptors remain the same
- `exec()` causes the OS to:
 - Destroy the address space of the calling process
 - Load the new program in memory, creating a new stack and heap
 - Run the new program from its entry point

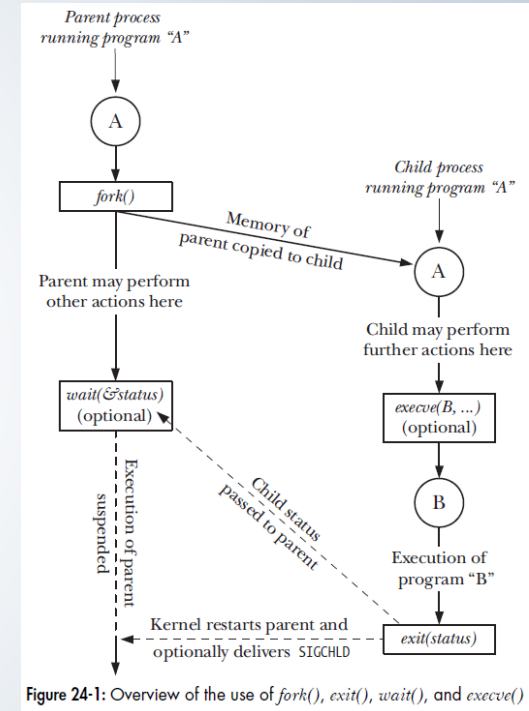


Figure 24-1: Overview of the use of `fork()`, `exit()`, `wait()`, and `execve()`

execve(): Loading and Running Programs

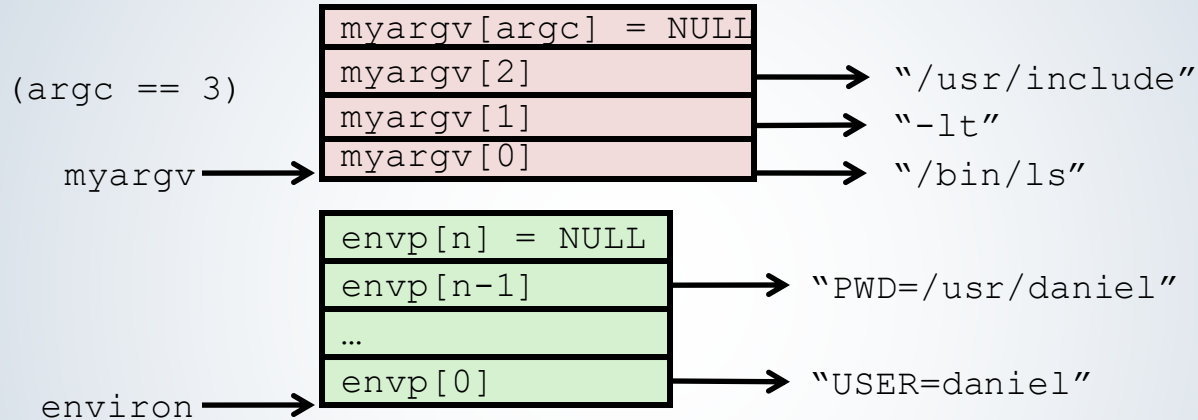
- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process:
 - Executable file filename
 - Can be object file or script file beginning with `#!/interpreter` (e.g., `#!/bin/bash`)
 - ...with argument list `argv`
 - By convention `argv[0] == filename`
 - ...and environment variable list `envp`
 - “name=value” strings (e.g., `USER=droh`)
 - `getenv`, `putenv`, `printenv`
- Overwrites code, data, and stack
 - Retains PID, open files and signal context
- Called once and never returns
 - ...except if there is an error

Not in xv6



execve() Example

- **Executes** `"/bin/ls -lt /usr/include"` in child process using current environment:



```
if ((pid = fork()) == 0) { /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Process State – Linux

- The kernel has a *process descriptor* of type struct task_struct for each process
 - Defined in <linux/sched.h>
- Process descriptor contains all the information about a process
- The kernel stores the list of processes in a circular doubly linked list called the task list
- What does the state of a process include?
 - State: running, ready, terminated, waiting
 - Priority
 - Parent
 - PID (process identifier)
 - Address space
 - Pending signals
 - Open files
 - etc.

Process State – xv6

- Also has a data structure for storing process state
 - Found in kernel/proc.h (struct proc)
- Processes are stored in a simple statically allocated array
 - Defined in kernel/proc.c (struct proc proc[NPROC];)
- struct proc still maintains process state, but there is a lot less of it than in linux (26 lines vs 811)!

```
// Per-process state
struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state; // Process state
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    int xstate; // Exit status to be returned to parent's wait
    int pid; // Process ID

    // wait_lock must be held when using this:
    struct proc *parent; // Parent process

    // these are private to the process, so p->lock need not be held.
    uint64 kstack; // Virtual address of kernel stack
    uint64 sz; // Size of process memory (bytes)
    pagetable_t pagetable; // User page table
    struct trapframe *trapframe; // data page for trampoline.S
    struct context context; // switch() here to run process
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
    uint64 prio; // Current scheduling priority
};
```

proc.h

Zombie Example

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6639 ttyp9        00:00:03 forks
 6640 ttyp9        00:00:00 forks <defunct>
 6641 ttyp9        00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6642 ttyp9        00:00:00 ps
```

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

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

Non-Terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
            getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
            getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9        00:00:00 tcsh
 6676 tttyp9        00:00:06 forks
 6677 tttyp9        00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9        00:00:00 tcsh
 6678 tttyp9        00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

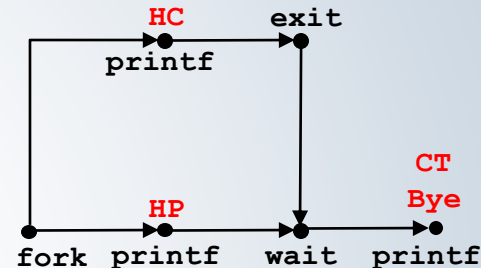
wait(): Synchronizing with Children

- 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
 - `child_status` variable is used to communicate information about the child
 - In xv6, it just returns the exit status of the child process
 - In Linux, macros defined in `wait.h` (e.g., `WIFEXITED`, `WIFSIGNALED`, etc.) can be used to determine how the process was terminated
 - See textbook for details if interested

wait(): Synchronizing with Children

```
void fork9() {  
    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");  
}
```

forks.c



Feasible output:

HC
HP
CT
Bye

Infeasible output:

HP
CT
Bye
HC

Exercise

```
void fork_exercise() {  
    int pid;  
    int status;  
  
    for(i = 0; i < 3; i++){  
        if(i % 2 == 0){  
            pid = fork();  
        }  
        printf("i = %d\n", i);  
    }  
    if(pid != 0){  
        wait(&status);  
    }  
    printf("Bye\n");  
}
```

Draw the process graph

How many times is "Bye" printed?

Summary

- Spawning processes
 - Call `fork()`
 - One call, two returns
- Process completion
 - Call `exit()`
 - One call, no return
- Reaping and waiting for processes
 - Call `wait()` or `waitpid()`
- Loading and running programs
 - Call `execve()` (or variant)
 - One call, no return unless error