Lecture 4: Process Creation and Control

CS 3281

Daniel Balasubramanian, Shervin Hajiamini

Lecture slides borrowed and adapted from the CMU course: Computer Systems: A Programmer’s Perspective

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
    - 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:

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

Obtaining Process IDs

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

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 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 Example

```c
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);
}
```

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

- 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

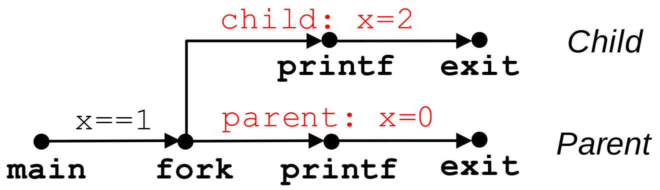
Modeling `fork` with Process Graphs

- 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 -> 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 inedges
- Any topological sort of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

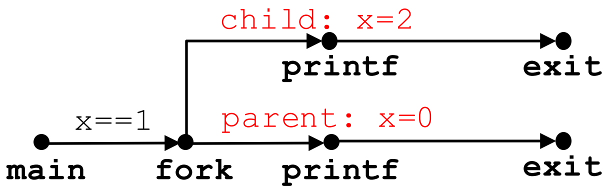
```
int main()
{
    pid_t pid;
    int x = 1;

    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

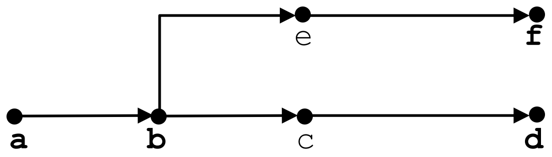


Interpreting Process Graphs

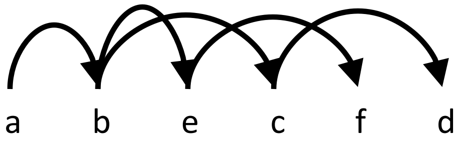
- Original graph:



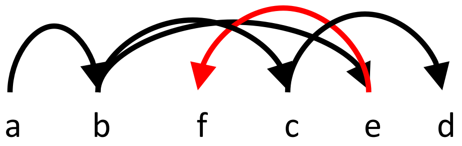
- Relabled graph:



Feasible total ordering:



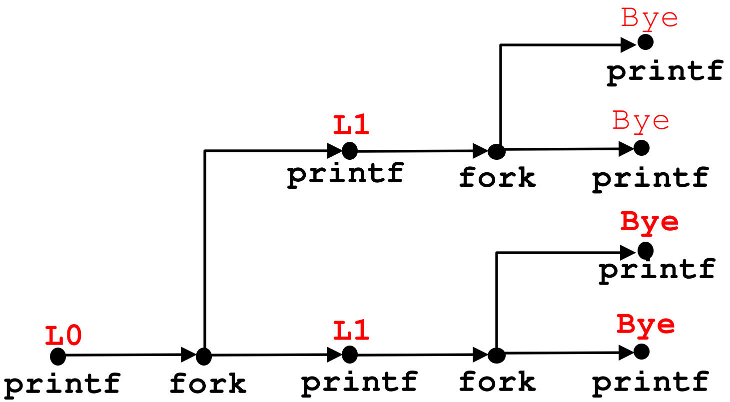
Infeasible total ordering:



`fork` Example: Two consecutive `fork`s

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

forks.c



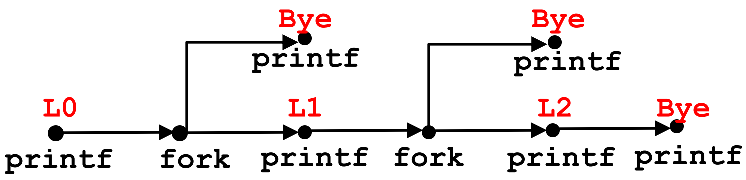
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

`fork` Example: Nested `fork`s in parent

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

forks.c



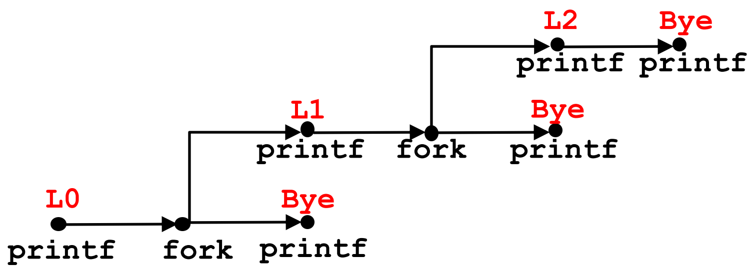
Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

`fork` Example: Nested `fork`s 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

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

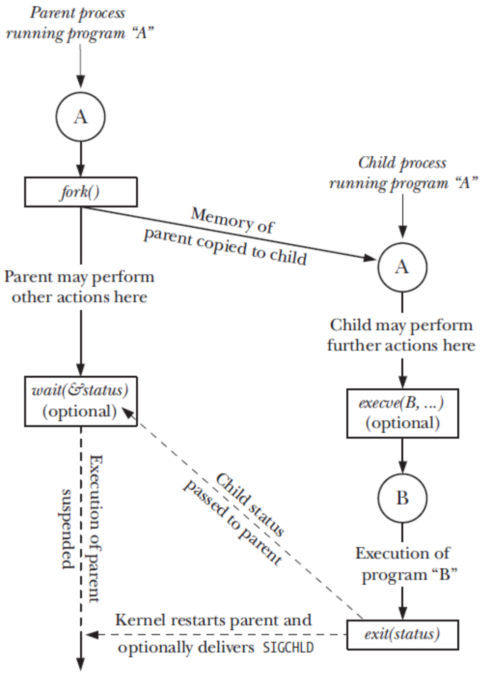


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

`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

Why are fork() and exec() separate?

- Why are fork() and exec() separated into two calls?
  - The separation allows the parent process to "fix-up" file descriptors after fork() but before exec()
  - We'll cover this in detail later; they allow the parent to redirect the input and output of the new process

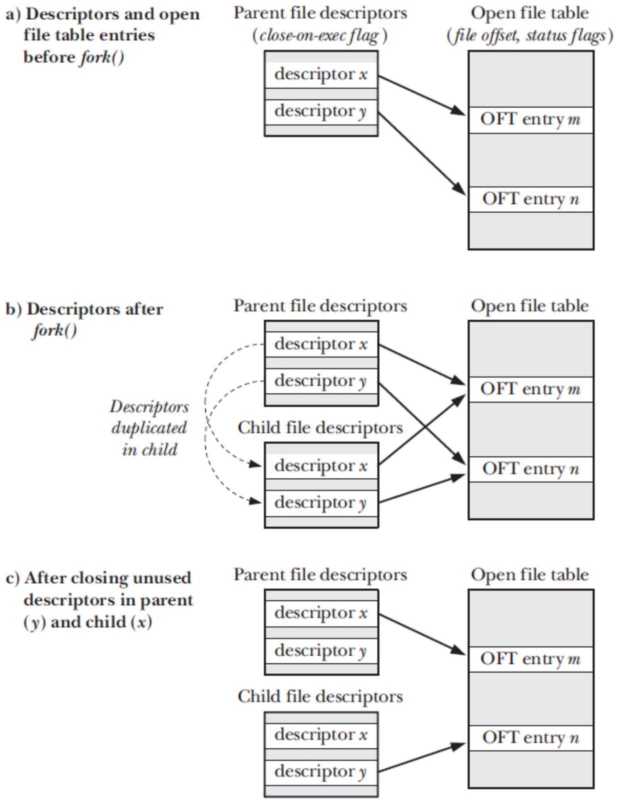
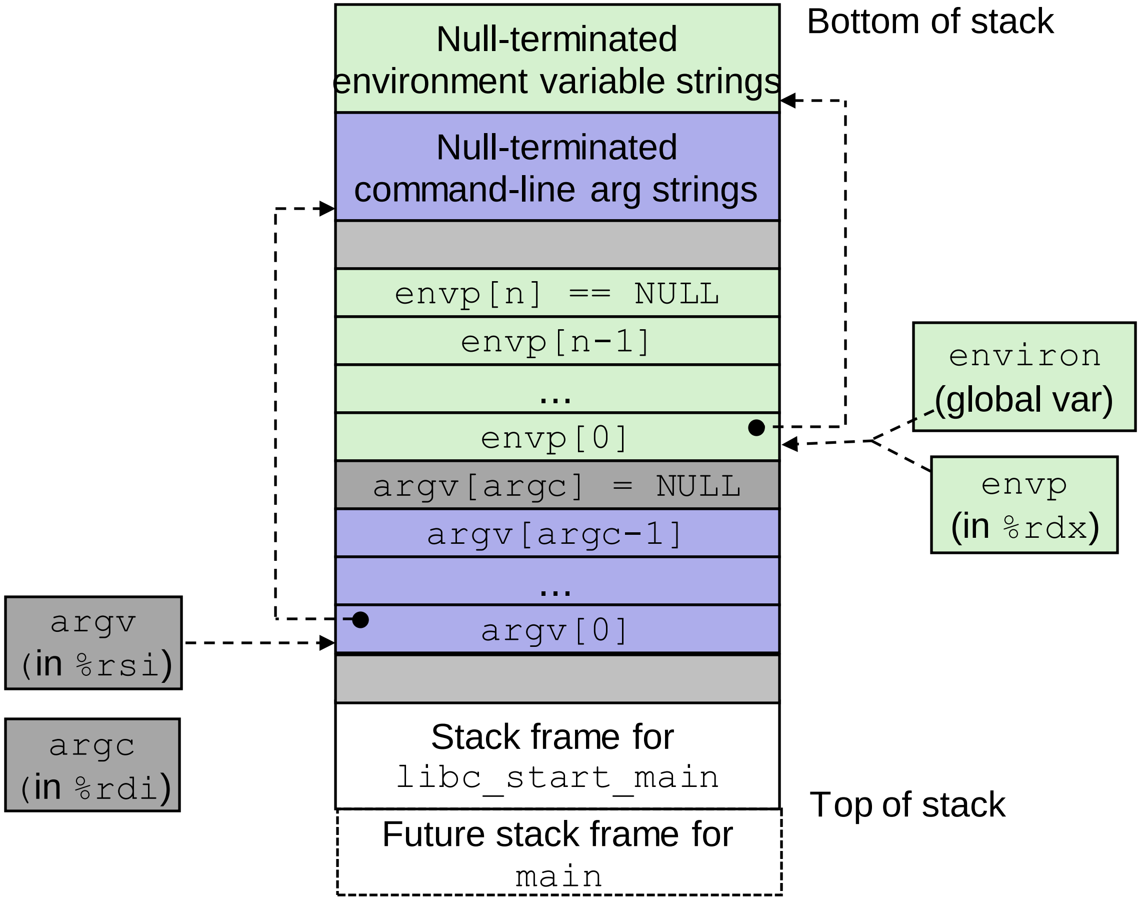


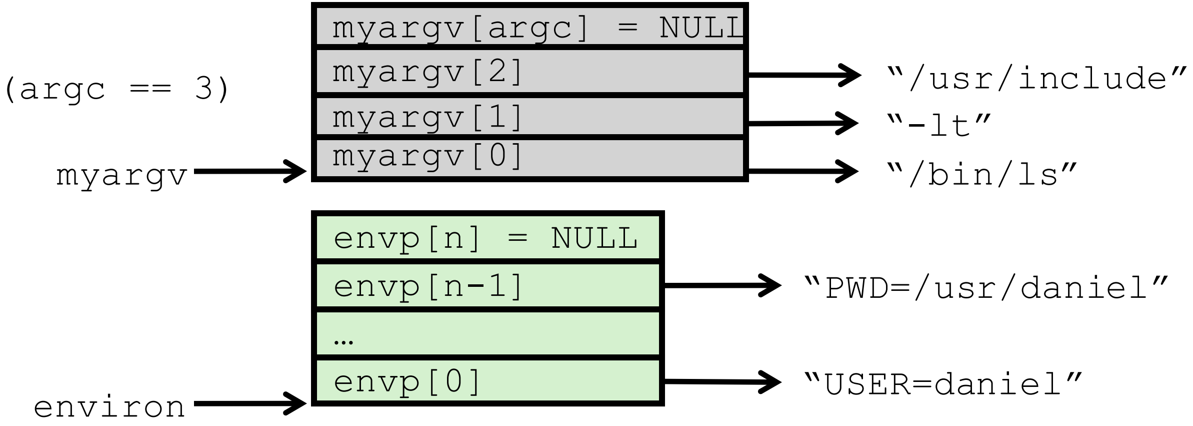
Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

Structure of the stack when a new program starts



`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

- 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

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 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

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

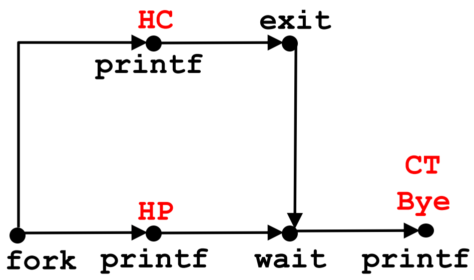
`wait`: Synchronizing with Children

- 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
  - 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
      - See textbook for details

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

Another wait() example

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

```
void fork10() {
   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);
   }
}
```

forks.c

Summary

- Exceptions
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)
- Processes
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space

Summary (cont.)

- 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, (normally) no return