CS3281 / CS5281

Process Creation and Control

CS3281 / CS5281

Spring 2026

**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

VANDERBILT UNIVERSITY

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 0x80 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, Libc 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. E.g., exit().
- Example:

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

Obtaining Process IDs

<u>Linux</u>

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

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

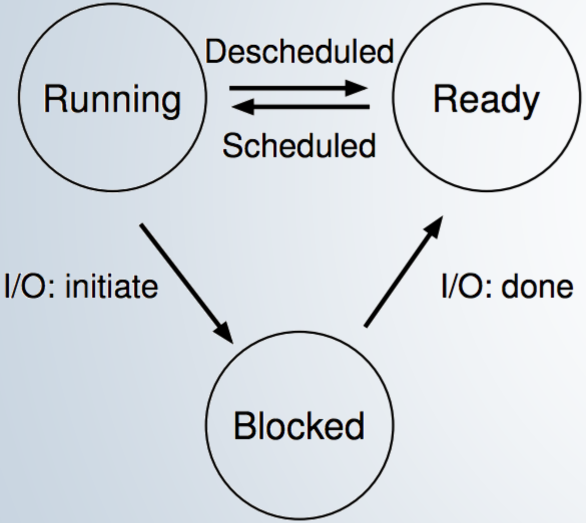
Creating and Terminating Processes

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

- Running
  - Process is either executing on CPU
- Ready
  - Process waits to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
  - i.e. process enters the ready state because of the context switch
- Stopped (blocked)
  - Process execution is *suspended* and will not be scheduled until further notice
  - i.e. process is blocked because it requests a resource that is not available now
- Terminated
  - Process is stopped permanently

Processes don't run all the time

The three basic process states:



- OS ***schedules*** processes
  - Decides which of many competing processes to run.
- A ***blocked*** process is not ready to run.
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke, updating display.
  - While waiting for results, the process often cannot do anything, so it **blocks**, and the OS schedules a different process to run.

Process State – Linux

- The kernel has a *process descriptor* (also called process control block or PCB) 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
  - Saved registers
  - etc.

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

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

```
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);
}
```

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 <u>independent</u>
- Shared open files

```
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
```

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

Question

- What does the following code do?

```
#include <stdio.h>
#include <sys/types.h>

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

Question

- What does the following code do?

```
#include <stdio.h>
#include <sys/types.h>

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

- Creates a new process
  - Then each process creates a new process
  - Then each of those creates a new process…
- Known as a Fork bomb!
  - Machine eventually runs out of memory and processing power and will stop working
- Defense: limit number of processes per user

Modeling fork() with Process Graphs

- A process graph is a useful tool for capturing the partial ordering of program statements:
  - 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 incoming edges
- Any topological sort of the graph corresponds to a feasible total ordering (i.e., valid output)
  - A permutation of vertices where all edges point from left to right

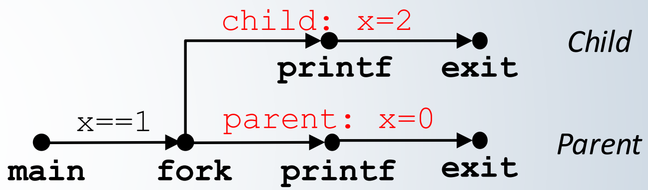
Process Graph Example

```
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);
}
```

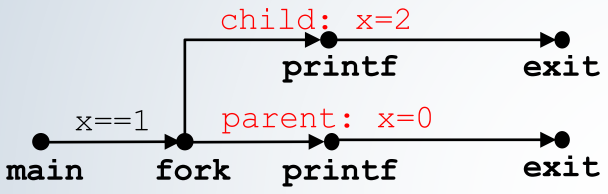
fork.c



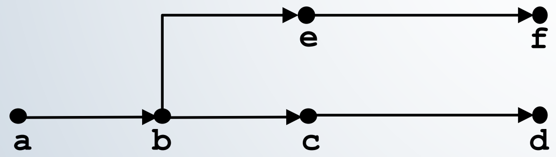
Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

Interpreting Process Graphs

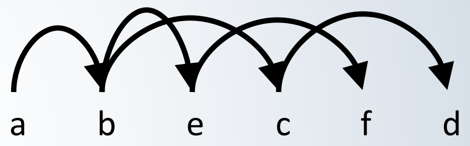
- Original graph:



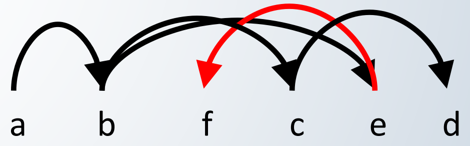
- Relabeled graph:



Feasible total ordering:



Infeasible total ordering:

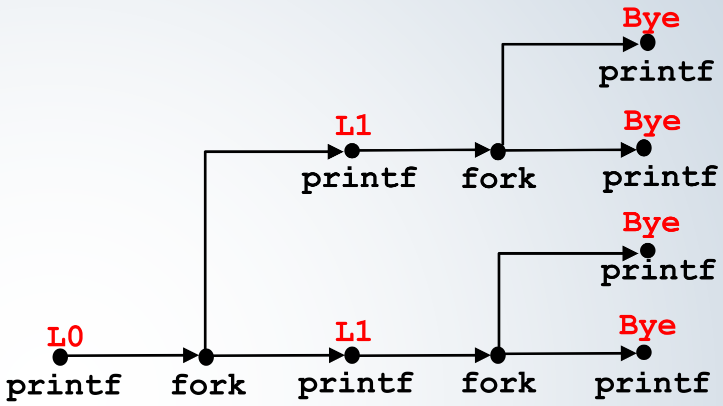


Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

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

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

VANDERBILT UNIVERSITY

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:

Infeasible output:

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

fork() Example: Nested Forks in Parent

```c
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

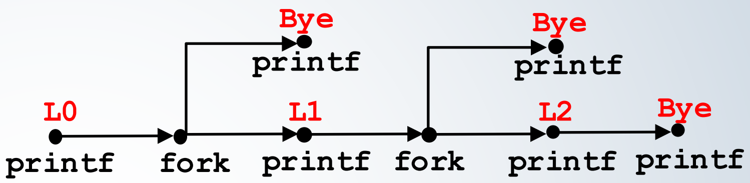
Infeasible output:
L0
Bye
L1
Bye
Bye
L2

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

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

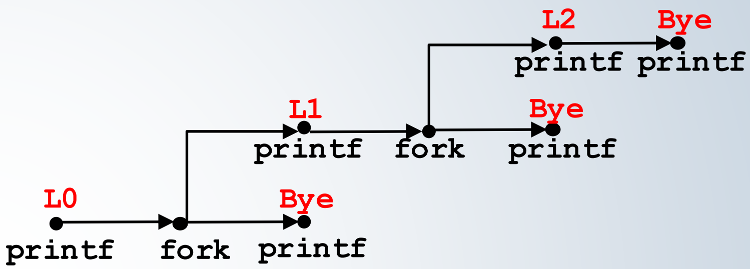
Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

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

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

Reaping Child Processes

- Idea
  - When process terminates, it still consumes system resources
    - Example: Exit status
  - 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. The child can create a new group and become a leader using setpgid(0, 0)
  - Resource limits
  - Working directory. E.g., filedesc = open("testfile.txt",some options…)
  - Open file descriptors

- But what if we want to execute a different program via fork()?

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

exec(): Loading a New Program

- The exec() is a family of functions that load 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

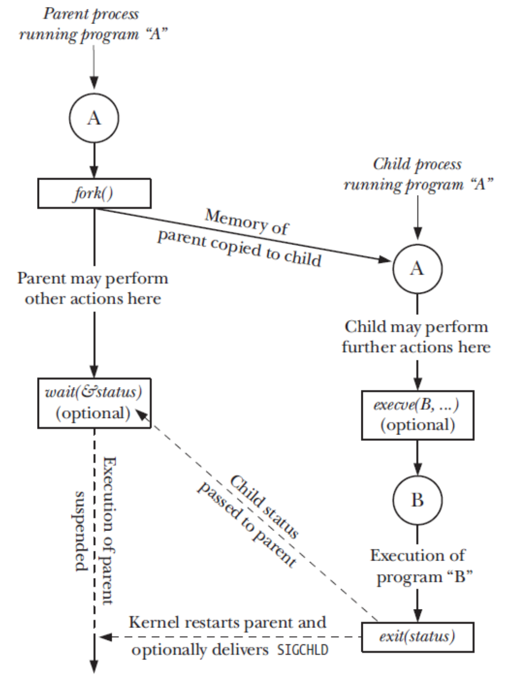


Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

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

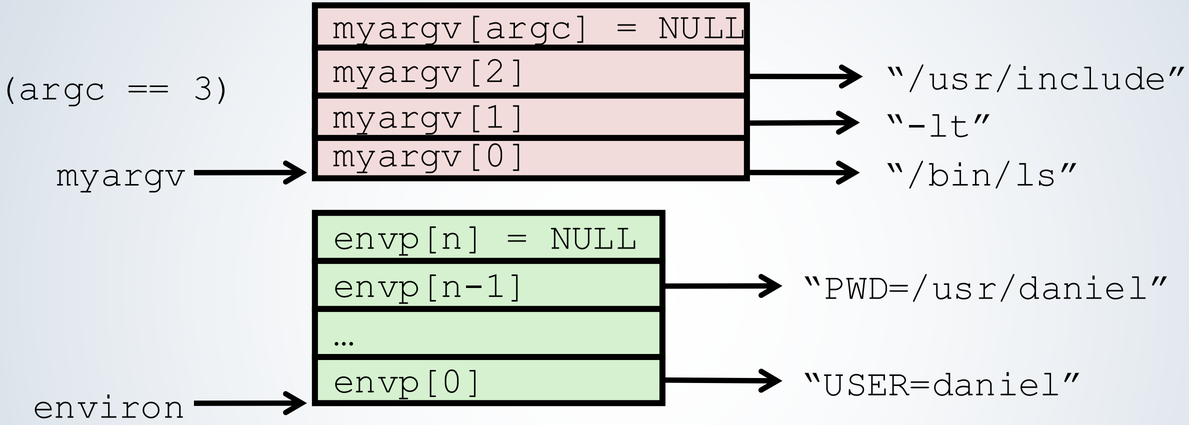
execve(): Loading and Running Programs

- 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 <u>never returns</u> if it succeeds because exec() overwrites the address space of the process
  - …except if there is an error

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

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);
   }
}
```

Zombie Example

```
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 */
   }
}
```

```
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
```

- `ps` shows child process as “defunct” (i.e., a zombie)
- Killing parent allows child to be reaped by `init`. It is known as *cascading* termination. init periodically calls wait() to identify and terminate the orphaned processes

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

Non-Terminating Child Example

```c
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
  – child_status variable is used to communicate information about the child

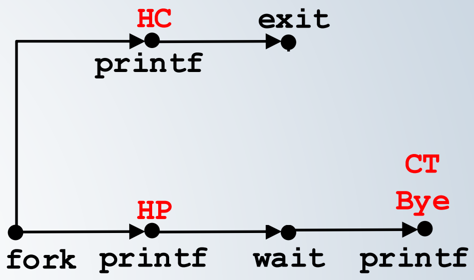
    - In Linux, macros defined in wait.h (e.g., WIFEXITED, WIFSIGNALED, etc.) can be used to determine how the process was terminated
      – See textbook for details if interested

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:

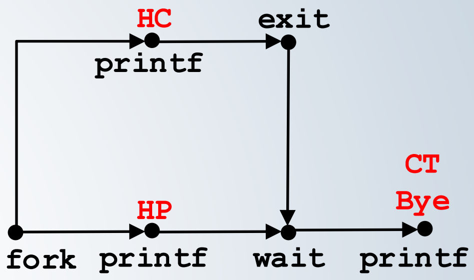
Infeasible output:

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

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

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?

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

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