04. Process API

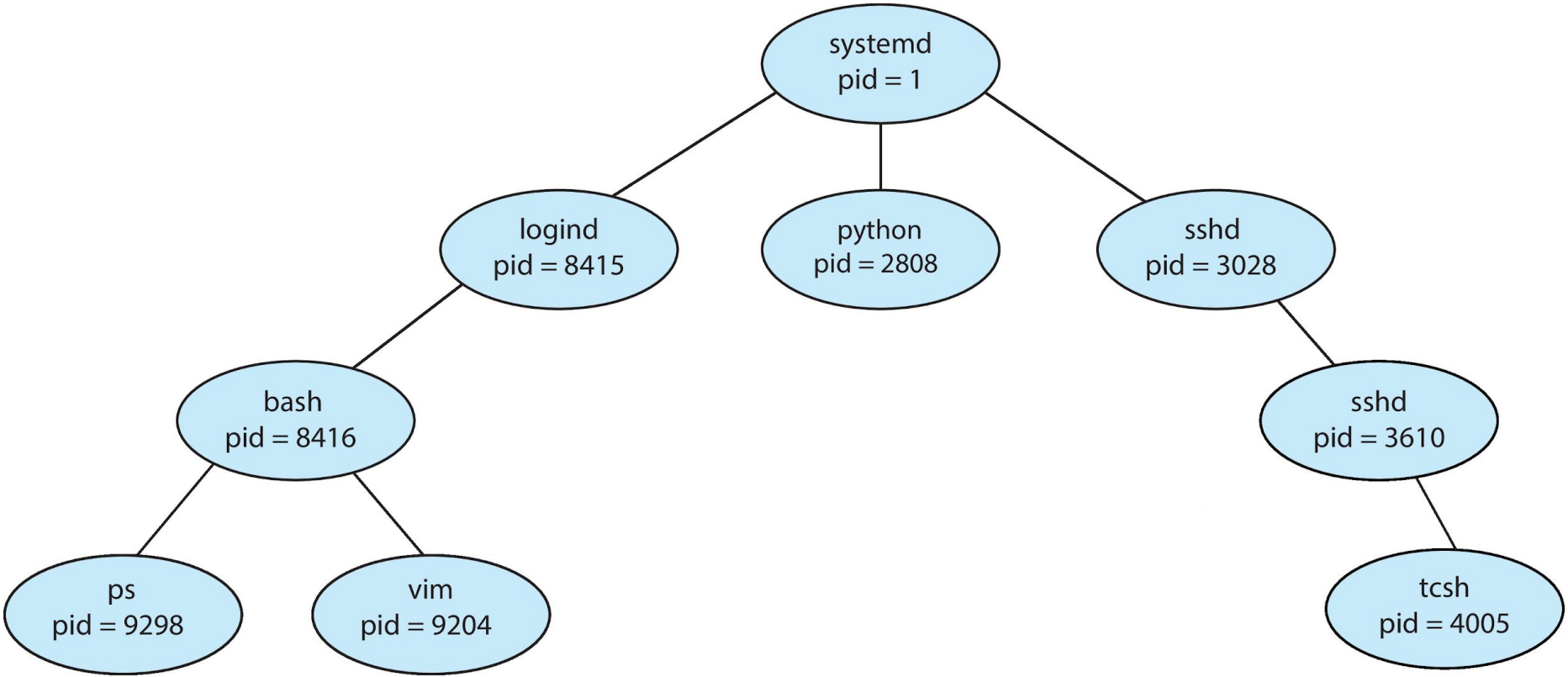
CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Process Creation

- Parent process creates child processes, which, in turn create other processes, forming **a tree of processes**
  - Generally, process identified and managed via a process identifier (**PID**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and children share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux



Obtaining Process IDs

- pid_t getpid(void)
  - Returns PID of current process
- pid_t getppid(void)
  - Returns PID of parent process
- The pid_t data type is a signed integer type which is capable of representing a process ID
  - In the GNU C Library, this is an int

fork() System Call

- In Unix, processes are created using **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
  - Again, remember this! It returns the child's PID to the parent, 0 to the child

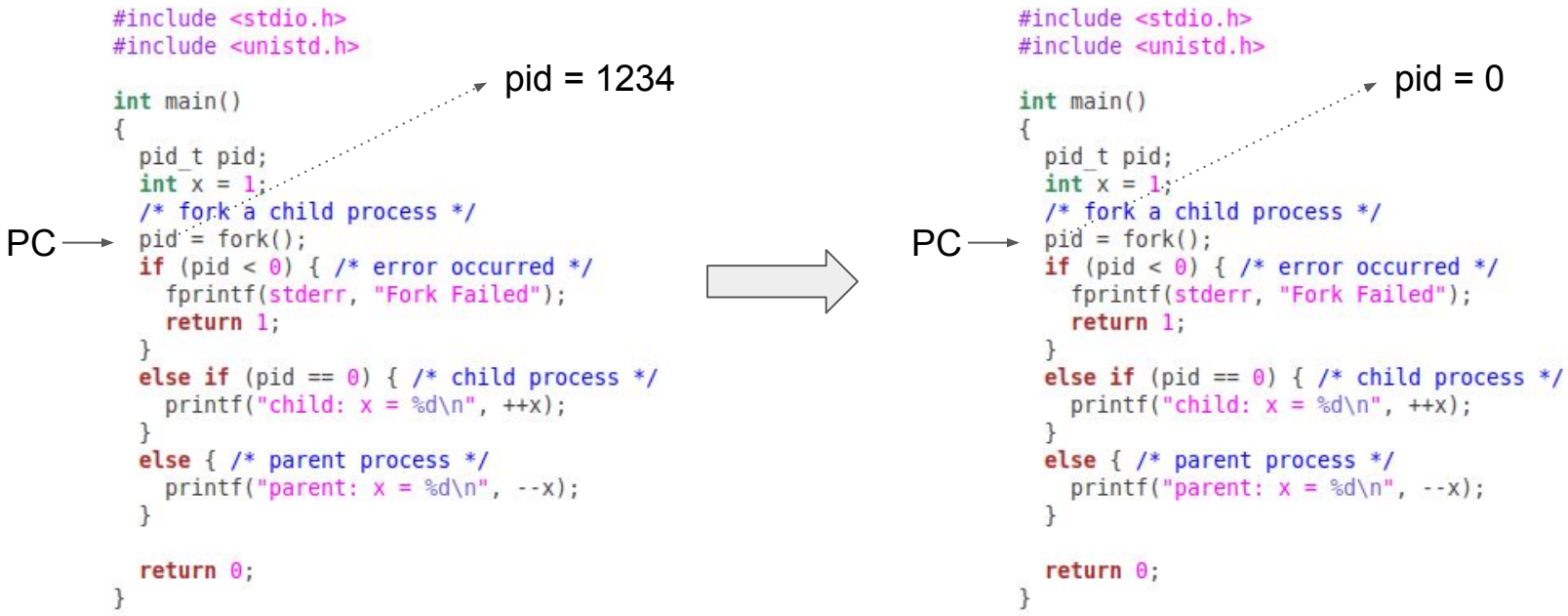
fork() Example

- 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

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
  pid_t pid;
  int x = 1;
  /* fork a child process */
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    printf("child: x = %d\n", ++x);
  }
  else { /* parent process */
    printf("parent: x = %d\n", --x);
  }

  return 0;
}
```

```
zzk@isis:~/courses/CS4352/demo$ ./a.out
parent: x = 0
child: x = 2
```

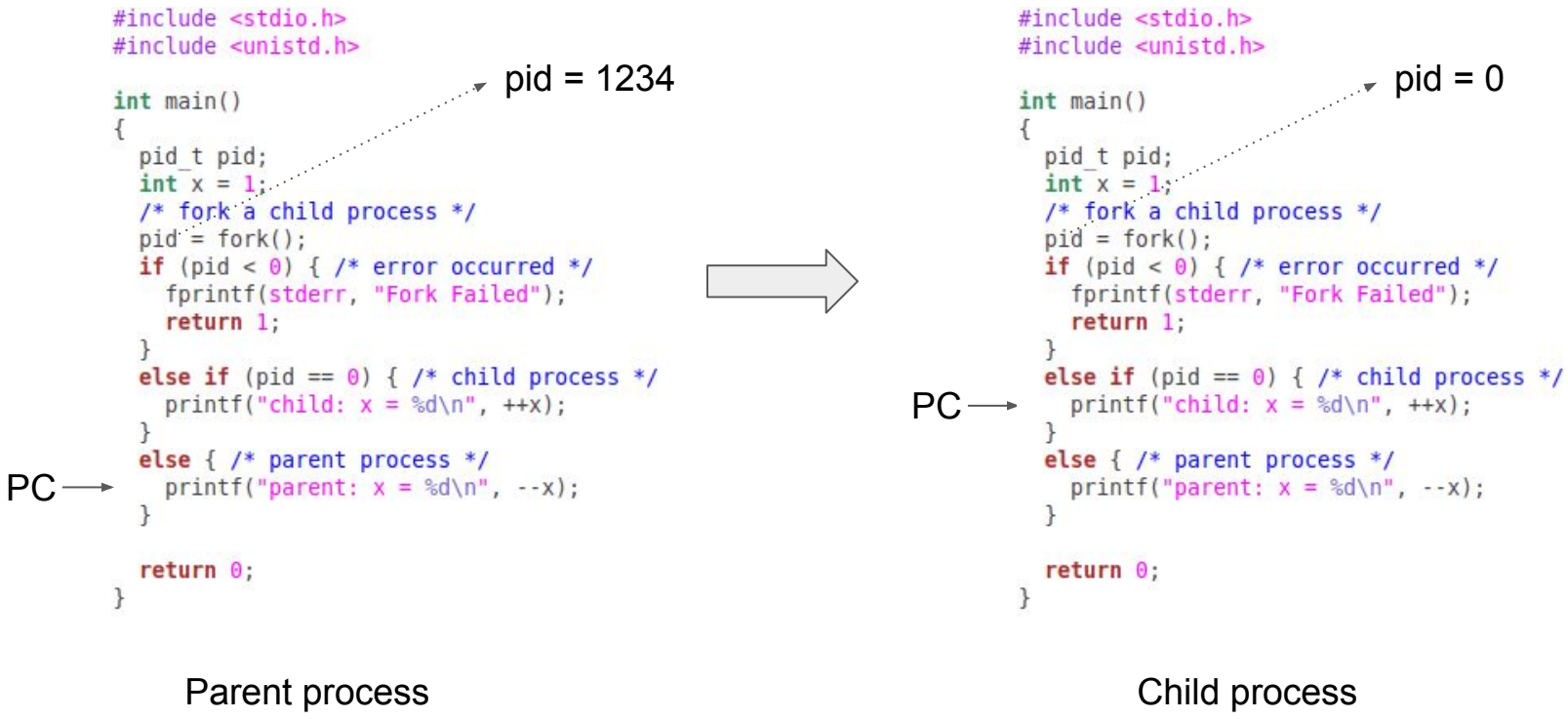
Duplicating Address Spaces



Parent process

Child process

Divergence


#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
int x = 1;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
printf("child: x = %d\n", ++x);
}
else { /* parent process */
printf("parent: x = %d\n", --x);
}
return 0;
}
pid = 1234
PC
Parent process
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
int x = 1;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
printf("child: x = %d\n", ++x);
}
else { /* parent process */
printf("parent: x = %d\n", --x);
}
return 0;
}
pid = 0
PC
Child process

Execution Order

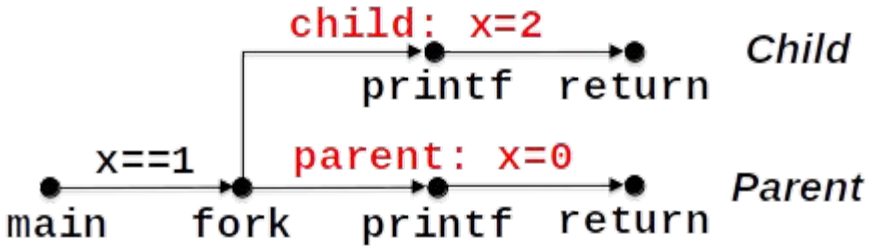
```
zzk@isis:~/courses/CS4352/demo$ ./a.out
parent: x = 0
child: x = 2
```

```
zzk@isis:~/courses/CS4352/demo$ ./a.out
child: x = 2
parent: x = 0
```

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

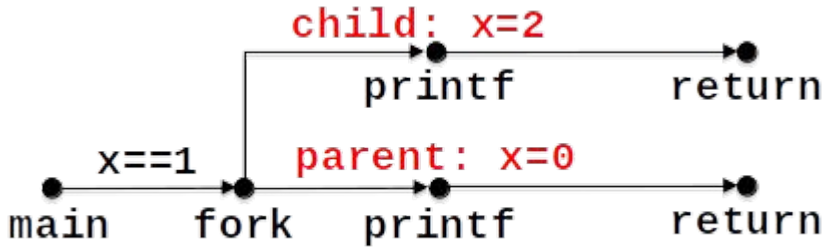
Process Graph Example



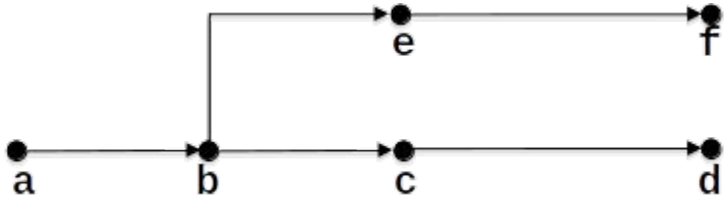

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
  pid_t pid;
  int x = 1;
  /* fork a child process */
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    printf("child: x = %d\n", ++x);
  }
  else { /* parent process */
    printf("parent: x = %d\n", --x);
  }

  return 0;
}
```

Interpreting Process Graphs

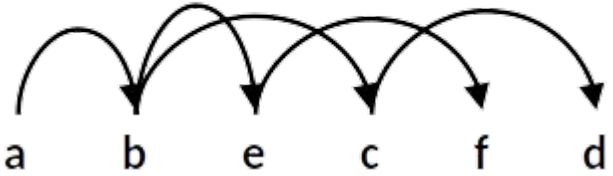
- Original graph:



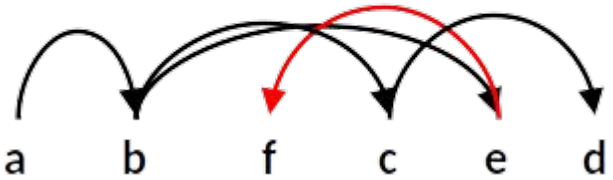
- Relabled graph:



Feasible total ordering:

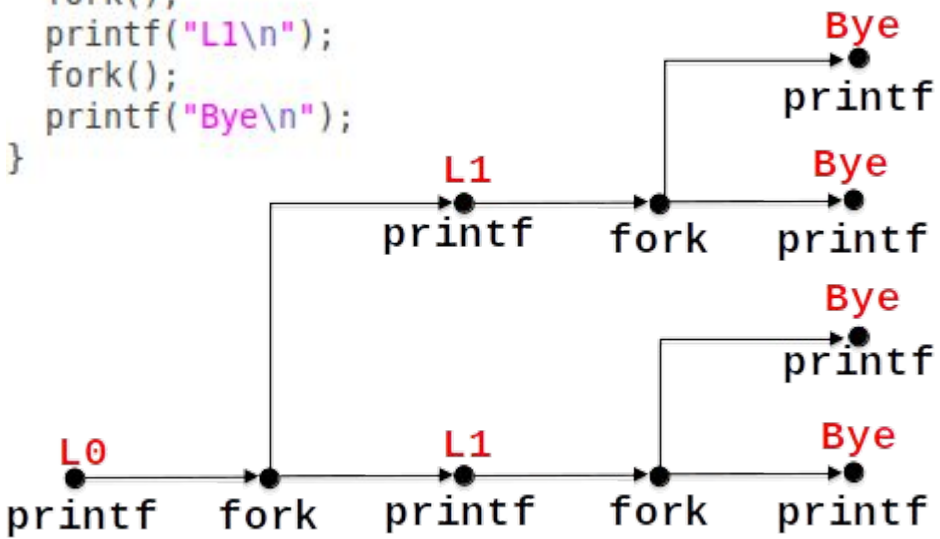


Infeasible total ordering:



fork() Example: Two Consecutive forks

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



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork() Example: Nested forks in Parent

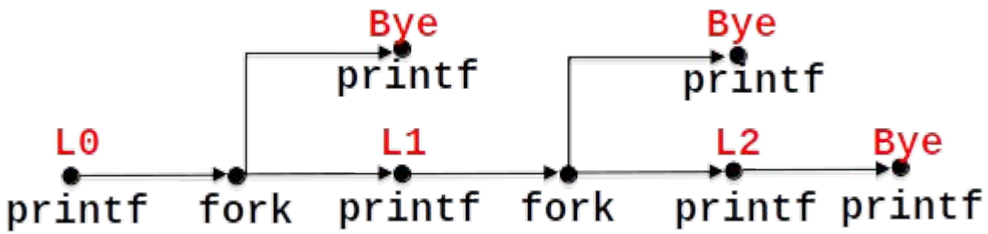
```c
void fork4()
{
  printf("L0\n");
  if (fork() != 0) {
    printf("L1\n");
    if (fork() != 0) {
      printf("L2\n");
    }
  }
  printf("Bye\n");
}
```

Feasible output:

L0
L1
Bye
Bye
L2
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2



exec() System Call

- Wait a second. How do we actually start a new program?
  - We need to use the exec() system call!
- The exec() system call loads a new program
  - The existing address space is blown away and loaded with the data and instructions of the new program
  - The 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
  - However, things like the PID and file descriptors remain the same

Different Wrapper Flavors

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *envp[]);
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);
```

- The base of each is exec (execute), followed by one or more letters:
  - e – an array of pointers to environment variables is explicitly passed to the new process image
  - l – command-line arguments are passed individually (a list) to the function.
  - p – uses the PATH environment variable to find the file named in the file argument to be executed.
  - v – command-line arguments are passed to the function as an array (vector) of pointers.

execve(): Loading and Running Programs

- int execve(char const *path, char const *argv[], char const *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)
  - Argument list argv
    - By convention argv[0] == filename
  - Environment variable list envp
    - “name=value” strings (e.g., USER=zzk)
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

execve() Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  pid_t pid;
  char *myargv[] = {"/bin/ls", "-al", "/", NULL};
  char *environ[] = {NULL};
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
    }
  }
  printf("how many times am I printed?\n");
}
```

Why Are fork() and exec() Separate?

- Most calls to fork() followed by exec()
  - Why not combine them together?
  - Not exactly the same as the combination, but CreateProcess() on Windows creates a new process and loads a program
- This separation is useful when the child …
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task
- It has been thought as one part of Unix philosophy

Process Termination

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate
  - 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
  - The wrapper exit() function runs the functions registered by atexit()
    - atexit() function appends a list of routines that should be called when the application terminates normally
    - The _exit() after main() actually returns control to the operating system
- exit() is called **once** but **never** returns

Reaping Child Processes

- When a process terminates for any reason, the kernel does not remove it from the system immediately
  - Instead, the process is kept around in a terminated state until it is reaped by its parent
- When the parent reaps the terminated child, the kernel passes the child's exit status to the parent, and then discards the terminated process, at which point it ceases to exist
  - A terminated process that has not yet been reaped is called a **zombie**
    - Living corpse, half alive and half dead

How to reap the terminated child processes?

wait(): Synchronizing with Children

- Parent can reap 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:
    - The <sys/wait.h> include file defines several macros for interpreting the status argument
      - E.g., WIFEXITED(status): returns true if the child terminated normally via a call to exit() or a return
      - WEXITSTATUS(status): returns the exit status of a normally terminated child
        - This status is only defined if WIFEXITED returned true

waitpid(): Waiting for a Specific Process

- pid_t waitpid(pid_t pid, int &status, int options)
  - Suspends current process until specific process terminates
  - Many options that can be found by typing “man waitpid”

```
for (i = 0; i < N; ++i) {
  if ((pid[i] = fork()) == 0)
    exit(100 + i); /* Child */
}

for (i = 0; i < N; ++i) {
  pid_t wpid = waitpid(pid[i], &child_status, 0);
  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);
}
```

What If ...

- What happens if a parent process terminates before its child?
  - Sadly :-(it becomes an orphan
  - An orphan process is adopted by the init process once its parent process dies
- The init process has a PID of 1 and is created by the kernel during system initialization
  - The init process will periodically execute wait() to reap processes
- Long-running programs such as shells or servers should always reap their zombie children
  - Again, even though zombies are not running, they still consume system memory resources

Zombie Example

```c
int main()
{
  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 */
  }
}
```

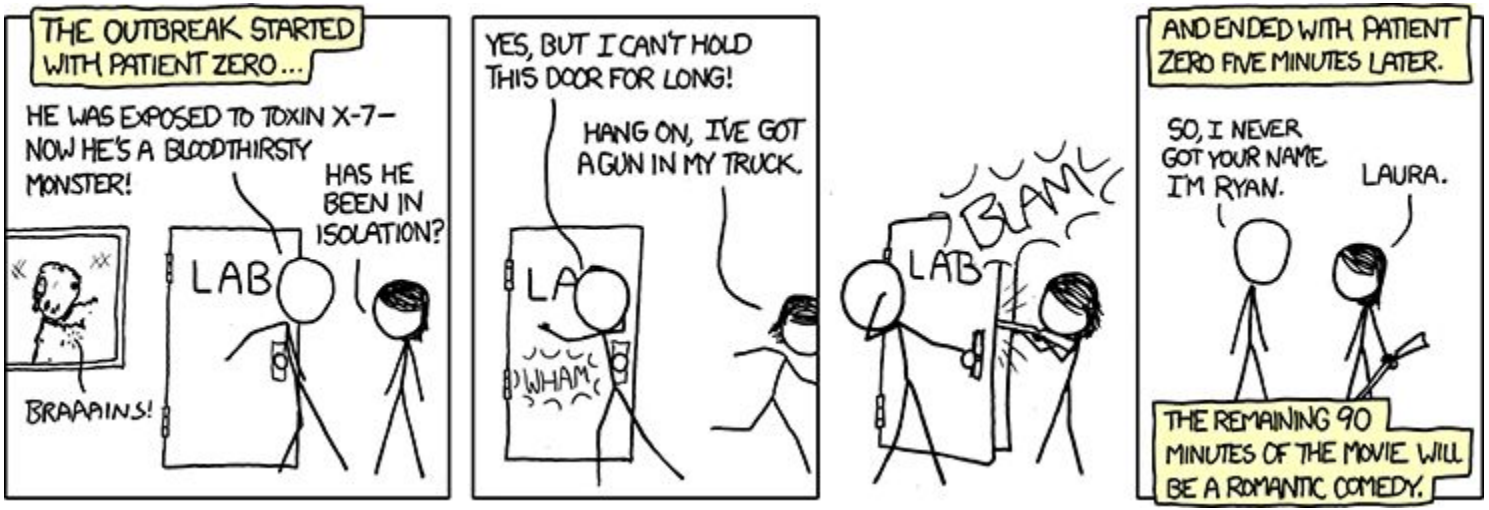
```
zzk@isis:~/courses/CS4352/demo$ ./a.out
Running Parent, PID = 18053
Terminating Child, PID = 18054
^Z
[1]+  Stopped                 ./a.out
zzk@isis:~/courses/CS4352/demo$ ps
  PID TTY          TIME CMD
18028 pts/0    00:00:00 bash
18053 pts/0    00:00:31 a.out
18054 pts/0    00:00:00 a.out <defunct>
18148 pts/0    00:00:00 ps
zzk@isis:~/courses/CS4352/demo$ fg
./a.out
^C
zzk@isis:~/courses/CS4352/demo$ ps
  PID TTY          TIME CMD
18028 pts/0    00:00:00 bash
18250 pts/0    00:00:00 ps
zzk@isis:~/courses/CS4352/demo$
```

Homework

- Start reading Chapter 5

Next Lecture

- We learn process scheduling



Credit: https://xkcd.com/734/