CS415 Module 2 Part B - Using fork() and exec()

Athens State University

Contents

1 The Process Model

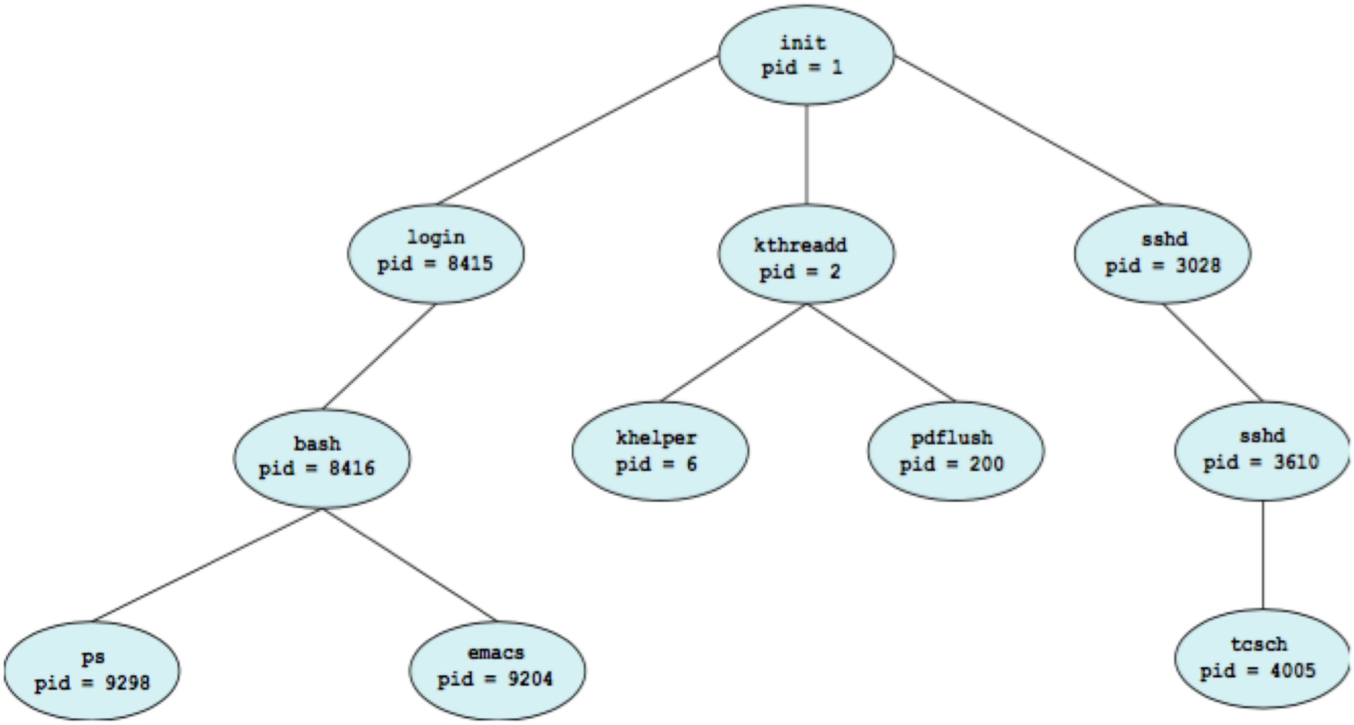
The Unix SVR4 Process Model

- Most of the OS executes within the environment of a user process
- System processes run in kernel mode
  - Perform administrative and housekeeping functions
- User processes
  - Operate in user mode to execute user programs and utilities
  - Switch to kernel mode (if permitted) to execute instructions that belong to the kernel
  - Enter kernel mode by issuing a system call, when an exeception is generated, or when an interrupt occurs

Distinguished Processes

- The first process in a Unix system (Linux, for example) is the `init` process
  - First and only process started by the kernel. Kernel creates the process and goes idle
  - Responsible for creating all other processes
  - Can be configured to start different things

A Process Tree



2 Process creation

Process creation

- Option 1: *cloning* (e.g. POSIX `fork()` and `exec()`)
  - Pause current process and save its state
  - Copy all or parts of current processes PCB
  - Add new PCB to the ready queue
  - Must have way to distinguish between parent and child
- Option 2: `from scratch` (Win32 `CreateProcess()`)
  - Load code and data into memory
  - Create and initialize PCB
  - Add new PCB to ready queu

Linux Process Creation

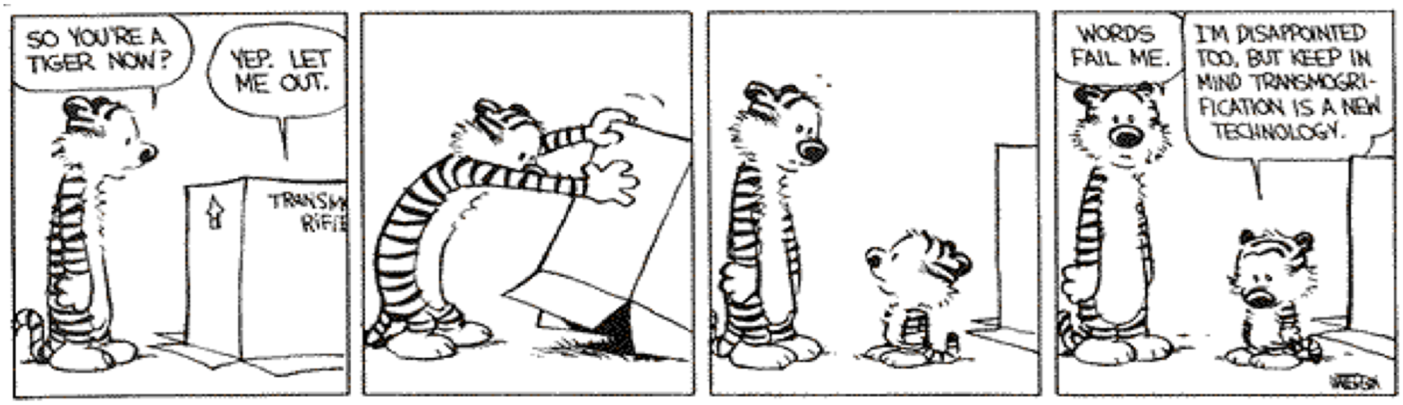
- Process creation is by means of the `fork()` system call
- The causes the OS, in Kernel Mode, to

1. Allocate a slot in process table for the new process
2. Assign a unique process ID to the child process
3. Make a copy of the process image of the parent, excepting any shared memory
4. Increment counters for any files owned by the parent, to reflect that an additional process now also owns those files
5. Assigns the child process to the Ready to Run state
6. Returns the ID number of the child process to the parent process, and a 0 value to the child process

The `fork()` system call

- Child is ALMOST a complete copy of the parent
  - Has its own PID and different PPID
  - Resource limits reset to zero
  - Resets any locks on files
- If the system call succeeds, then the PID of the child process is returned to the parent and 0 is returned to the child process
- If the system calls, then a -1 is returned to the parent

The `exec()` family of system calls


SO YOU'RE A TIGER NOW?
YEP. LET ME OUT.
TRANSM RIFIE
WORDS FAIL ME.
I'M DISAPPOINTED TOO, BUT KEEP IN MIND TRANSMOGRI-FICATION IS A NEW TECHNOLOGY.

If `fork()` is our equivalent to Calvin's duplicator, then `exec()` is equivalent to his transmorgifer

The `execve()` system calls

```
execve(pathname, argv, envp)
```

- Loads a new program (*pathname*) with argument `argv` and environment list *envp* in program memory
- Existing program text is discarded, and new stack, data, and heap segments are created for the program

The *execve()* system call is the lowest level member of the `exec()` family of system calls. We will start with this call and add the others as we go along.

So, process creation in POSIX environments such as Linux is a two step process: (1) call `fork()` to duplicate the parent, and (2) if desired, call one of the `exec()` system calls to copy a new program on top of the program currently running in the child process.

The `wait()` system call

- Typically want the parent process to wait until the child does something or terminates
- When called in the parent process, `wait()` suspends the parent until all child processes terminate
- If the calling process has no children, then the system call returns immediately with a value of -1

Summary of Steps to Create a New Process

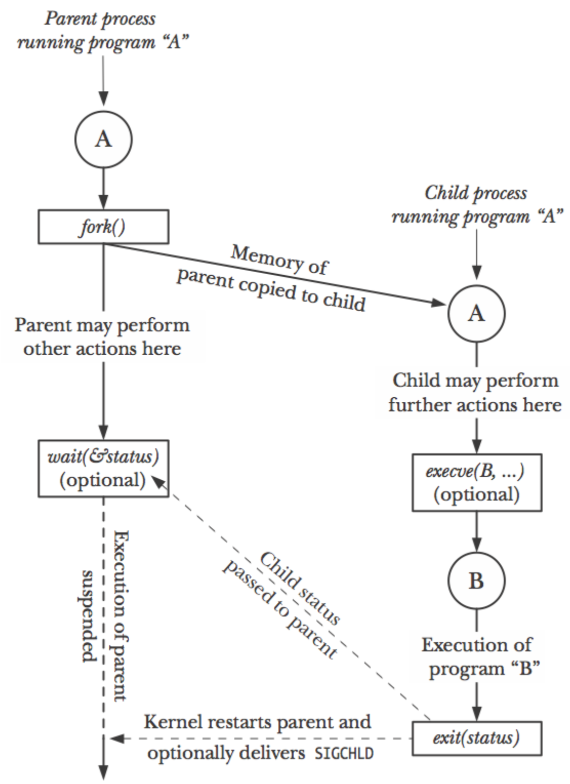

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
Execution of parent suspended
Child status passed to parent
B
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

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

3 A few examples

Basic use of `fork()`

```
#include <cstdlib>
#include <iostream>
#include <string>
// Required by fork routine
#include <sys/types.h>
#include <unistd.h>
using namespace std;
int globalVariable = 2;
int main(int argc, char **argv) {
   string sIdentifier;
   int    iStackVariable = 20;
   pid_t pID = fork();
   if (pID == 0) {                // child
      // Code only executed by child process
      sIdentifier = "Child Process: ";
      globalVariable++;
      iStackVariable++;
    }
    else if (pID < 0) {            // failed to fork

        cerr << "Failed to fork" << endl;
        exit(1);
        // Throw exception
    }
    else {                                   // parent
      // Code only executed by parent process
      sIdentifier = "Parent Process:";
    }
    // Code executed by both parent and child.
    cout << sIdentifier;
    cout << " Global variable: " << globalVariable;
    cout << " Stack variable: "  << iStackVariable << endl;
}
```

Waiting on a process to terminate

```
#include <cstdlib>
#include <unistd.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
int main() {
  pid_t pid;
  int status, died;
     switch(pid=fork()) {
     case -1:
       cout << "can't fork\n";
       exit(-1); // why is there no break here?
     case 0 :
       sleep(2); // this is the code the child runs
       exit(3);  // why is there no break here?
     default:
       died= wait(&status); // this is the code the parent runs
     }
}
```

Waiting on more than one thing

```
#include <cstdlib>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <iostream>
using namespace std;

int main() {
   pid_t pid;
   int status, died;
   switch(pid=fork()) {
   case -1: cout << "can't fork\n";
            exit(-1);
   case 0 : cout << "   I'm the child of PID " << getppid() << ".\n";
            cout << "   My PID is " <<  getpid() << endl;
      sleep(2);
            exit(3);
   default: cout << "I'm the parent.\n";
            cout << "My PID is " <<  getpid() << endl;
            // kill the child in 50% of runs
            if (pid & 1)
               kill(pid,SIGKILL);
            died= wait(&status);
            if(WIFEXITED(status))
               cout << "The child, pid=" << pid << ", has returned "
                    << WEXITSTATUS(status) << endl;
            else
         cout << "The child process was sent a "
                    << WTERMSIG(status) << " signal\n";
  }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char *argv[], char *env[] ) {
   pid_t my_pid, parent_pid, child_pid;
   int status;
   /* get and print my pid and my parent's pid. */
   my_pid = getpid();    parent_pid = getppid();
   printf("\n Parent: my pid is %d\n\n", my_pid);
   printf("Parent: my parent's pid is %d\n\n", parent_pid);
   /* print error message if fork() fails */
   if((child_pid = fork()) < 0 ) {
      perror("fork failure");
      exit(1);
   }
   /* fork() == 0 for child process */
   if(child_pid == 0) {
      printf("\nChild: I am a new-born process!\n\n");
      my_pid = getpid();    parent_pid = getppid();
      printf("Child: my pid is: %d\n\n", my_pid);
      printf("Child: my parent's pid is: %d\n\n", parent_pid);
      printf("Child: I will sleep 3 seconds and then execute - date - command \n\n");
      sleep(3);
      printf("Child: Now, I woke up and am executing date command \n\n");
      execl("/bin/date", "date", (char *)0, (char *)0);
      perror("execl() failure!\n\n");
      printf("This print is after execl() and should not");
      printf(" have been executed if execl were successful! \n\n");

      _exit(1);
   }
```

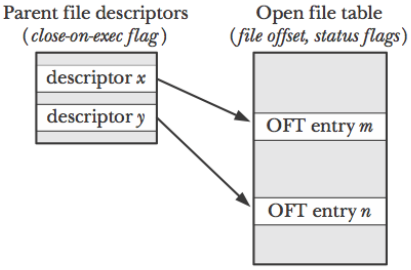
```
    /* parent process  */
    else {
       printf("\nParent: I created a child process.\n\n");
       printf("Parent: my child's pid is: %d\n\n", child_pid);
       system("ps -acefl | grep ercal");   printf("\n \n");
       wait(&status); /* can use waitq(NULL) since exit status
                         from child is not used. */
       printf("\n Parent: my child is dead. I am going to leave.\n \n ");
    }
    return 0;
}
```

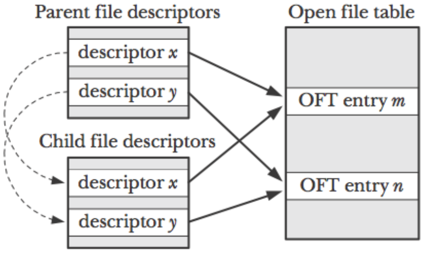
3.1 A short digression

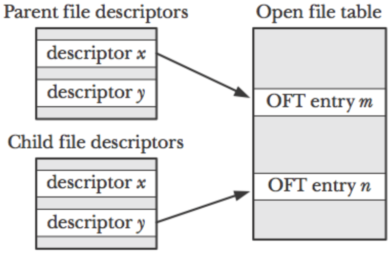
A short digression: files

- We are taught to use the stream classes to access files in C++
- Classical C had the `fopen`, `fread`, `fwrite`, and `fclose` functions
- In both cases, those mechanisms will eventually call the `open`, `read`, `write`, and `close` system calls

So, what happens when you `fork()`?







- Recall that the process is Calvin's duplicator: it's an exact copy
- So, which process owns the metadata for files that accesses up to the point in the program where the `fork()` occurs?
- We need some mechanism that will duplicate this information into the child process

The dup() and dup2() system calls

- Both these calls will create a copy of a given file descriptor
- The dup() call returns the next available file descriptor
- The dup2() call takes two parameters: the old and new file descriptors
  - The system call closes the old descriptor
  - And opens the new descriptor to the same location as the old descriptor
- Note that both system calls work more like an alias than a copy

3.2 And now back to our regularly scheduled program

Using dup() with fork() and exec()

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <cstdio>
#define OUTPATH "output"
#define MESSAGE "Behold, Standard Out is now a file!"
int main(int argc, char **argv) {
 //First, we open a file for writing only
 int outputfd = -1;
 outputfd = open(OUTPATH, O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRGRP | S_IROTH);
 //If we have an error, we exit. N.B. file descriptors less than one are invalid
 if (outputfd < 0) {
     perror("open(2) file: " OUTPATH);
     exit(EXIT_FAILURE);
 }
 //Next, we close Standard Out
 // The lowest file descriptor will now be STDOUT_FILENO
 if (close(STDOUT_FILENO) < 0) {
     perror("close(2) file: STDOUT_FILENO");
     close(outputfd);
     exit(EXIT_FAILURE);
  }
  //Afterwards, we duplicate outputfd onto STDOUT_FILENO,
  //exiting if the descriptor isn't equal to STDOUT_FILENO*
  if (dup(outputfd) != STDOUT_FILENO) {
     perror("dup(2)");
     close(outputfd); // N.B. Remember to close your files!*/
     exit(EXIT_FAILURE);
   }
  close(outputfd); //If everything succeeds, we may close the original file
  puts(MESSAGE);   //and then write our message
  return EXIT_SUCCESS;
}
```