Hello Dolly

Johan Montelius and Amir H. Payberah

1 Introduction

This is an experiment where we will explore how processes are created and what is shared between them. You should have a basic understanding of processes, but we will not assume you're an expert C programmer.

We will use the library procedure `fork()`, so first, take a look at the manual pages. You will probably not understand everything they talk about, but we get the important information that we need to start experimenting.

```
$ man fork
```

We see that fork requires the library `unistd.h`, so we need to include this in our program. We also read that fork will return a value of type `pid_t`. This type is defined in a header file included by `unistd.h` and is a way of making the code architecture independent. We will ignore this and assume that `pid_t` is a `int`. Further down the man pages we read that fork returns both the process identifier of the child process and zero. This is strange; how can a procedure return two different values? Let's give it a try, create a file called `dolly.c` and write the following:

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

int main() {
  int pid = fork();

  printf("pid = %d\n", pid);
  return 0;
}
```

The above program will call `fork` and then print the returned value. Compile and run this program, what is happening?

2 The parent and the child

So, calling `fork()` somehow creates a duplicate of the executing process, and the execution then continues in both copies. By looking at the returned value, we can determine if we are executing in the *parent process* or if we are in the *child process*. Try this extension to the program:

```
int main() {
  int pid = fork();

  if(pid == 0)
    printf("I'm the child %d\n", getpid());
  else
    printf("My child is called %d\n", pid);

  printf("That's it %d\n", getpid());
  return 0;
}
```

This is not the only way this could have been implemented. One could, for example, have chosen to have a construction where we would provide a function that the child process would call. Different operating systems have chosen different strategies, and Windows, for example, have chosen to provide a procedure that creates a new process independent of the parent process.

2.1 Wait a minute

To terminate the program in a more controlled way, we can have the parent `wait()` for the child process to terminate. Try the following:

```
int main() {
  int pid = fork();

  if(pid == 0) {
    printf("I'm the child %d\n", getpid());
    sleep(1);
  } else {
    printf("My child is called %d\n", pid);
    wait(NULL);
    printf("My child has terminated \n");
  }
  printf("This is the end (%d)\n", getpid());

  return 0;
}
```

The parent process waits for its child process to terminate (actually, it waits for any child it has spawned). Only then will it proceed, print out the last row and terminate.

2.2 Returning a value

A process can produce a value (e.g., an integer) when it terminates, and the parent process can pick up this value. If we change the program so that the child process returns 42 as it exists, the value can be picked up using the `wait()` procedure.

```
  if(pid == 0)
    return 42;
  else {
    int res;
    wait(&res);
    printf("the result was %d\n", WEXITSTATUS(res));
  }
```

2.3 A zombie

A *zombie* is a process that has terminated but whose parent process has not yet been informed. As long as the parent has not issued a call to `wait()`, we need to keep part of the child process. When calling `wait()`, the parent process should be able to pick up the exit status of the child process and possibly a return value. This information is lost if the child process is completely removed from the system.

We can see this in action if we terminate the child process but wait for a while before calling `wait()`. Do the following changes to the program, call it `zombie.c`, compile and run it in the background.

```
  if(pid == 0) {
    printf("check the status\n");
    sleep(10);
    printf("and again\n");
    return 42;
  } else {
    sleep(20);
    int res;
    wait(&res);
    printf("the result was %d\n", WEXITSTATUS(res));
    printf("and again\n");
    sleep(10);
  }
  return 0;
}
```

Check the status of the processes using the `ps` command. Notice how the two processes are created and how the child becomes a zombie and is removed from the system once we receive the return value.

```
$ gcc -o zombie zombie.c
$ ./zombie&
 :
$ ps -ao pid,stat,command
 :
```

2.4 A clone of the process

So we have created a child process that is a *clone* of the parent process. The child is a copy of the parent with an identical memory. We can exemplify this by showing that the child has access to the same data structures but that the structures are just copies of the original data structures. Extend `dolly.c` and try the following:

```
int main() {
  int pid;
  int x = 123;
  pid = fork();

  if(pid == 0) {
    printf("  child:  x is %d\n", x);
    x = 42;
    sleep(1);
    printf("  child:  x is %d\n", x);
  } else {
    printf("  mother: x is %d\n",  x);
    x = 13;
    sleep(1);
    printf("  mother: x is %d\n",  x);
    wait(NULL);
  }
  return 0;
```

As you see, both the parent and the child sees ha variable `x` as `123` but the changes made are only visible by themselves. If you want to see something very strange you can change the printout to also print the memory address of the variable `x`. Do this for both the parent and the child, and you will see that they are actually referring to the same memory locations.

```
    printf("  child:  x is %d and the address is 0x%p\n", x, &x);
```

The explanation is that processes use *virtual addresses* and are identical, but they are mapped to different *real memory addresses*. How this is achieved is nothing that we should explore now, but it is fun to see that it is working.

2.5 What we do share

Since the child process is a clone of the parent process, we do share some parts. One thing that we do share is references to open files. When a process opens a file, the operating system creates a *file table entry*. The process is given a reference to this entry, which is stored in a *file descriptor table* owned by the process. When the process is cloned, this table is copied, and all the references are pointing to the same entries in the *file table*. The standard output is nothing more than an entry in the file descriptor table, so this is why both processes can write to the standard output. We also read from the same standard input, so we also have a race condition.

If you look at man pages, you will see a whole range of structures that the processes share or not share, but most of those are not very interesting to us in this set of experiments.hose are not very interesting to us in this set of experiments.

3 Groups, orphans, sessions, and daemons

The parent process has a special relationship to the child process. The parent process has to keep track of its child, and a child always knows the process identifier of its parent.

```
int main() {
  int pid = fork();

  if(pid == 0) {
    printf("I'm the child %d with parent %d\n", getpid(), getppid());
  } else {
    printf("I'm the parent %d with parent %d\n", getpid(), getppid());
    wait(NULL);
  }
  return 0;
}
```

Compile and run this in a terminal. Who is the parent of the parent process? The following commands might give you a clue.

```
 $ ps a
  :
 $ echo $$
  :
```

We could find more information about the processes using some flags to `ps`. Try `ps -fp $$` to see more details about the shell you are using (`$$` will expand to the process identifier of the shell). The `PPID` field is the parent process identifier. Who is the parent of the shell? Where does it all stop?

3.1 The group

The fate of a parent and its child is not directly linked to each other, but they belong to the same *process group*. Each process group has a process leader, and in our simple examples, the parent process has been the group's leader. The group identifier/leader is retrieved using the system call `getpgid()`.

```c
int main() {
  int pid = fork();

  if(pid == 0) {
    int child = getpid();
    printf("I'm the child %d in group %d\n", child, getpgid(child));
  } else {
    int parent = getpid();
    printf("I'm the parent %d in group %d\n", parent, getpgid(parent));
    wait(NULL);
  }
  return 0;
}
```

A group is treated as a unit by the shell. It can set a whole group to suspend, resume or run in the background (allowing the shell to use the standard input for interaction). However, we will not go into how the shell works, so let's just accept that processes belong to a process group.

3.2 Ophans

As a change, we can try to crash the parent process and see what happens with the child process.

```c
int main() {
  int pid = fork();

  if(pid == 0) {
    int child = getpid();
    printf("child: parent %d, group %d\n", getppid(), getpgid(child));
    sleep(4);
    printf("child: parent %d, group %d\n", getppid(), getpgid(child));
    sleep(4);
    printf("child: parent %d, group %d\n", getppid(), getpgid(child));
  } else {
    int parent = getpid();
    printf("parent: parent %d, group %d\n", getppid(), getpgid(parent));
    sleep(2);
    int zero = 0;
    int i = 3 / zero;
  }
  return 0;
}
```

Save the program in a file called `orphan.c`. Compile and execute the program, and notice how the parent identifier of the child process changes. The process has turned into an *orphan* and adopted by the `upstart` process (or `init` or `systemd` depending on which system you using). Note the new process identifier and then check its state using the `ps` command:

```
$ ps <whatever the process id was>
 :
```

To see something fun, you can take a look at the `process tree` of the process:

```
$ pstree <whatever the process id was>
 :
```

3.3 Sessions and daemons

The origin of the notion of a session is a user attaching and logging in to the system. A session consists of a set of groups and a *session leader*. As with groups, the sessions have identifiers equal to the leader's process identifier. Compile and run the program below. Which process is the session leader of our processes?

```
int main() {
  int pid = fork();

  if(pid == 0) {
    int child = getpid();
    printf("child:  session %d\n", getsid(child));
  } else {
    int parent = getpid();
    printf("parent: session %d\n", getsid(parent));
  }
  return 0;
}
```

When you start a new terminal, a new session is created. The operating system keeps track of sessions and will terminate all groups in a session if the session leader terminates. This means that if you log in to a system and start to run processes in the background, they still belong to the same session as your login shell and will be terminated if the session terminates.

If one wants to create a process that should survive the session, it must form its own session. It becomes a *daemon*, a process running in the background detached from any *controlling terminal*. Daemons perform many of the tasks performed by the operating system. They keep track of network interfaces, USB devices and schedule tasks that should run periodically. Your system will probably have fifty daemons running in the background, but they consume very few resources.

4 Starting a program

So far, we have seen how a process can be created and how the child process is related to its parent process. To understand how an operating system works, there is one more very important functionality that we will take a look at: how we create a process that will execute another program.

When you use the command shell, this happens (almost) every time you enter a command. The shell interprets some commands and will do something for us, but most "commands" are programs the shell will start for us. How is a program actually started?

4.1 transforming a process

In Unix systems, the execution of a program is done by transforming an existing process to run the code of the given program. As you will see, starting a program is done in two steps: (1) creating the new process and (2) transforming the process into executing the program. The mechanism that makes this possible is the family of `exec()` system calls. Look-up the man pages of `exec`, we will use the one called `execlp()`.

```
int main() {
  int pid = fork();

  if(pid == 0) {
    execlp("ls", "ls", NULL);
    printf("this will only happen if exec fails\n");
  } else {
    wait(NULL);
    printf("we're done\n");
  }
  return 0;
}
```

The call to `execlp()` will find the program `ls` and then replace the code and data segments of the process with the code and data found in the executable binary. The stack and heap areas are reset, so the program starts the execution from scratch.

4.2 Redirection

Even if the memory segments of the process are cleared, the process keeps the file descriptor table. By changing the table entries, we can make the program read from a standard input of our choice and redirect the standard output. This allows us to control the I/O operations of the program without changing the program in any way. We can create a small program that does nothing but writes to standard output to see how this works. Let's call this program `boba.c`.

```
int main() {
  printf("Don't get in my way.\n");
  return 0;
}
```

Now, if we compile and run this program, we will see the quote printed in the terminal.

```
 $gcc -o boba boba.c
  :
 $./boba
  :
```

Note that we have to write `./boba` and not simply `boba` if you have not set up your `PATH` variable also to include the current directory; more on this later. Now, if we want to redirect the output to a file called `quotes.txt` we could do this from the shell directly.

```
 $ ./boba > quotes.txt
  :
```

To understand how the shell achieves this, we could try to write a program `jango.c`, that clones itself, redirects the standard output, and then transforms the clone into `boba`. Let's go:

```
int main() {
  int pid = fork();

  if(pid == 0) {
    int fd = open("quotes.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    dup2(fd, 1);
    close(fd);
    execl("boba", "boba", NULL);
    printf("this only happens on failure\n");
  } else {
    wait(NULL);
  }
  return 0;
}
```

In the `jango.c` program we open a file `quotes.txt` (providing flags to open it in read-write mode and create it if does not exist). The operating system will grate a new file table entry and add a reference to it in our file descriptor table. The table entry will be the first free entry in the table (3). We then use the system call `dup2()` to copy the entry to position 1 (the location of stdout). We then close the `fd` entry since we will not use this entry anymore. When we now call `execl()`, the process will turn into `boba`. The `boba` program knows nothing about what happened but will direct all output to file descriptor 1 as usual. Try it, and you will see that the file is created and that we will receive the output as expected.

4.3 Pipes

The full beauty of how standard input and output can be redirected is shown when we introduce the concept of *pipes*. A pipe is a FIFO buffer of characters, and when created, we will allocate two file descriptor entries. One entry is for reading and the other for writing. Since we are in full control over the descriptor table before we start executing a program, we can make one program send all the output to another program's input. This is a very powerful tool from the shell to combine sequences of commands.

Assume we have the commands (or rather programs) `ps axo sid` that will print the session identifier of every process in the system, `sort -u` that will sort lines and output only unique and `wc -l` that will count the number of lines. How do we combine these to find the number of sessions in the system? Using the shell, this is done in one line:

```
$ ps xao sid | sort -u | wc -l
  :
```

This is achieved using pipes, and we can set it up ourselves in a program. There are, however, a lot of details to get it right, and we will explore this later in the course. For now, you should explore using the pipes from the command line.

5 Summary

Processes are created by cloning an existing process, the execution continues in the two duplicates, and the only way to tell which copy we are executing is to look at the value returned from `fork()`. A parent and child process are in the same *process group*. If the group leader terminates, all processes in the group will be sent a signal that will likely cause them to terminate. Several groups belong to a *session* with a *controlling terminal*. If the session leader terminates or the controlling terminal closes, the whole session will be terminated. A session that has been detached from any controlling terminal is called a *daemon*. Daemons handle many of the tasks that constitute an operating system. Two copies have identical copies of *file descriptor tables* referring to the same *file table entries*. By changing the descriptor tables, the input and output of a process can be redirected. Two processes can use this to set up a *pipe* between them that acts as a buffered FIFO channel. A process can be transformed to run another program using the system call `exec()`. This will reset all memory segments, but the transformed process keeps the file descriptor table.