Week 7: Processes Part II

CS211: Programming and Operating Systems

Wednesday and Thursday, 26+27 Feb 2020

In Week 7 of CS211, we'll study

1. Recall...
   - ... The Process (again)
2. wait()
3. Overlaying a process's memory
4. Process Termination
5. Signals
6. Interprocess comms
   - producer-consumer model
   - IPC
7. Example 1: Pipes
8. Exercises

In this week of "Programming and Operating Systems", we continue our study of the theory and programming of ***processes***.

Recall... ... The Process (again)

In Week 6, we started our move towards the "Operating System" part of the course, the need to learn some classical OS Theory. Material from that class, and this one, are based on Chapters 4 and 5 of Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau:

Processes: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf`

Process API: `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`

In Week 6 we learned that

> ***"A Process... is a running programme"*** (OSTEP, p25)

Perhaps the most important task that an operating system must complete is to set a programming running. We learned last week what that meant in theory, but also how we could write a program that makes a **system call** to the operating system to create a process, using the `fork()` function.

This function is defined in the *`unistd.h`* header file.

All processes have a unique ***Process identification number***: **PID**.

Recall...

We examine the creation of processes by calls (from a C program) to the `fork()` function.
`fork()` is define in the the `unistd.h` header file.
IMPORTANT: *`unistd.h`* is not included in the installation of `code::blocks` on campus. Try

- `https://www.onlinegdb.com/online_c_compiler`
- `https://www.jdoodle.com/c-online-compiler`
- `https://paiza.io/projects/`
- `https://rextester.com/l/c_online_compiler_gcc`
- But not `https://www.tutorialspoint.com/compile_c_online.php` or `http://www.compileonline.com/` or `https://www.codechef.com/`.
  Also problematic: `https://repl.it/languages/c` and `https://ideone.co`

Recall...

- The process that calls `fork` is called the **parent**;
- the process that is created is called the **child**;

The prototype for `fork()` is

```
pid_t fork(void);
```

That is,

- it takes no arguments;
- the return type is really an `int`
- the return value is -1 if the `fork()` failed
- otherwise, it returns the PID of the newly made child to the parent.
- the return value is 0 to the newly made child.

- the **child** is distinct from the parent: it gets its own copy of the parent's memory space;
- both parent and child run **concurrently**;
- starting from the `fork()`, the parent and child execute the same instruction set: that is, from the point of the `fork()` onward, the two processes have the same tasks to preform.

Two other important functions:

```
pid_t getpid(void);
pid_t getppid(void);
```

getpid() returns the value of the processes' own PID

getppid() returns the value of the processes' parent's PID

Since the **parent** and **child** have copies of the same memory space and instruction set, `getpid()` and `getppid()` are very useful for working out which is which.

Recall...

`04Fork.c` from Week 6

```
1  // An example of forking a process
   #include <unistd.h>
3  #include <stdio.h>

5  int main(void )
   {
7    int pid1, mypid;

9    pid1 = fork();
     mypid = getpid();

     printf("I am %d\t", mypid);
13   printf("Fork returned %d\n", pid1);
     return(0);
15 }
```

Typical output:

```
I am 7791. Fork returned 0
I am 7790. Fork returned 7791
```

Recall...

01ParentsPID.c

```
int main(void )
{
  int pid1;
  pid1 = fork();
  printf("I am %d\t", getpid());
  printf("fork returned %5d\t", pid1);
  printf("My parent is %d\n", getppid());
  return(0);
}
```

OUTPUT:

I'm proc 7825. fork() returned 0. My partent is 7824

I'm proc 7824. fork() returned 7825. My partent is 5394

So, processes can use `getpid()` and `getppid()` to tell themselves apart. Of course, they could also do this by checking the return value of `fork()`.

02WhoAmI.c

```
int main(void ) {
  int pid1, mypid;
  pid1 = fork();
  mypid = getpid();
  if (pid1 != 0)
    printf("I'm the parent. My PID's %d, my child's is %d\n",
           mypid, pid1);
  else
    printf("I'm the child. My PID's %d, my parent's is %d\n",
           mypid, getppid());
  return(0);
```

Typical Output:

```
I am the parent. My PID is 10770, the child's is 10771
I am the child. My PID is 10771, my parent's is 10770
```

wait()

Often we don't want the parent to continue running while the child process is executing, particularly as the results may be *non-deterministic*: we can't predict with certainly what will happen in what order.
Here is an example of a program for which we cannot predict the order of the output:

wait()

03CountTo10.c

```
int main(void )
{
  int i;
  fork();
  printf("Watch me (%d) count to 10: ", getpid());
  for (i=1; i<=10; i++)
  {
    sleep(2*rand()%2); // sleep for 0 or 2 seconds
    printf("%3d...", i);
    fflush(stdout);
  }
  printf("\n");
  return(0);
}
```

Typical Output:

```
Watch me (11695) count to 10:   1...  2...  3...Watch me
(11696) count to 10:   1...  4...  2...  5...  6...  3...
7...  4...  8...  5...  9...  6... 10...
  7...  8...  9... 10...
```

wait()

A call to the `wait()` function suspends the execution of the parent process until such time as the **child** completes (or, at least, signals to the parent – more about that later).

04WaitAndCount.c

```
 6 #include <sys/wait.h>
   int main(void )
 8 {
     int pid1, i;
10   pid1=fork();
     srand(getpid());
12   if (pid1 != 0) // Parent follows this path
       wait(NULL);

     printf("Watch me (%d) count to 10: ", getpid());
16   for (i=1; i<=10; i++)
     {
18     sleep(rand()%2); // sleep for 0 or 1 seconds
       printf("%3d...", i);
20     fflush(stdout);
     }
22   printf("\n");
```

Overlaying a process's memory

Recall that usually a sub-proc will share the parent's memory only in the sense that it receives a copy:

1. The child can then mimic the parents execution, as in the Examples above
   or
2. its memory space may be ***over-layed*** with another program/set of instructions.

Often, when a subprocess is created, it is over-layed with another program. In C this can be done with the `execlp()` function.
In the following example the sub-procs memory space is over-layed the program text of the `ls` command.
Again we use the `wait()` function.

Overlaying a process's memory

05Overlay.c

```
    int pid1 = fork();

    if (pid1 == 0) // this is the child
12  {
      printf("This is process %d\t", getpid());
14    printf("Here is a directory listing:\n");
      execlp("ls", "ls", NULL);
16  }
    else // parent
18  {
       wait(NULL);
20     printf("This is process %d\t", getpid());
       printf("Subprocess %5d has completed\n", pid1);
22  }
    return(0);
```

Process Termination

The OS is responsible for de-allocating the resources of a process.
It may also be responsible for killing the proc.
Process Termination may occur when:

1. the proc executes its last instruction and asks the operating system to delete it (`exit()`). At that time it will usually:
   - Output data from child to parent (via `wait()`).
   - Have its resources de-allocated by operating system.
2. Parent terminates execution of children processes (`kill()`) because
   - the child/sub-proc has exceeded allocated resources.
   - the task assigned to child is no longer required.
3. the parent is exiting and OS does not allow child to continue if its parent terminates.

On Unix systems, when a process terminates, its children are "re-parented" (adopted) by the `init` process.

Signals

The `kill()` system call mentioned above is an example of a **signal** – a form a communication from one process to another. These provide a facility for ***asynchronous event handling*** (more of this later)

You may have used a `kill()` system call from the task-manager of an OS you use. To use `kill` function from within a C program:

```
1   #include <sys/types.h>
    #include <signal.h>
3   .
    .
5   int kill(pid_t pid, int sig);
```

Signals

06ChildKillsParent.c

```
 8 #include <signal.h>

10 int main(void )
   {
12   printf("Parent process has pid=%d\n", getpid());
     int child_pid = fork();
14   if (child_pid == 0) // child path
     {
16     sleep(3);
       printf("%d telling parent process (%d) to terminate\n",
18             getpid(), getppid() );
       kill(getppid(), SIGKILL);
20   }
     else // parent path
22   {
       sleep(20); // sleep for 20 seconds
24     printf("**%d: never gets to this line, do we?\n",
               getpid() );
26   }
```

Signals

Just terminating a process is not very subtle; instead a process can nominate a function that will be called if, say, the SIGUSR1 signal his sent to it.

07SIGUSR1.c

```
10 void timed_out(int sig)
   {
12   printf("Proc %d called timed_out with signal %d\n",
            getpid(), sig);
14 }

16 void DoSomeThingComplicated(void)
   {
18   sleep(10);
   }
```

Signals

07SIGUSR1.c

```
   int main(void )
22 {
     int pid=fork(); // New process will "watch over old one" and send
24                   // signal if timeout expires.

26   if (pid != 0 ) // I'm not the watcher
     {
28     signal(SIGUSR1, timed_out); // What to do if I get signaled
       DoSomeThingComplicated(); // Something that'll be interrupted

       printf("%d got woken \n", getpid());

       // Finished something complicated. Don't need watcher any more
34     kill(pid, SIGKILL);     // tell watcher to terminate.
     }
```

Signals

07SIGUSR1.c

```
36    else   // I am the watcher
      {
38      printf("I am the watcher (%d)\n", getpid());
        printf("Should %d signal parent (%d)? ('y'/'n')\n",
40              getpid(), getppid());
        if (getchar() == 'y')
42      {
          printf("Sending siguser 1 to parent\n");
44        kill(getppid() , SIGUSR1);
        }
46    }
      return(0);
```

Signals

What the process does ***after*** it has finished calling the signal handler function depends on the situation. In the above examples, it stopped `sleep`ing.
More typically, however, if it had been making a blocking call – such as taking input from the keyboard, or reading from a pipe – it returns to doing that.

Interprocess comms

The `kill()` system call can be considered as a form of ***communucation*** between process (though, arguably, not a very subtle one). Now we'll consider more general *interprocess communication*.

Two processes that are executing on the same computer may be either

Independent: they cannot affect or be affected by the execution of the other.

Cooperating: they can affect or be affected by the execution of each other.

Interprocess comms

It can be advantageous to allow processes communication in order to facilitate

(i) **Information sharing** – e.g., two procs might require access to the same file.

(ii) **Modularity** – different procs might be dedicated to different system functions.

(iii) **Convenience** – e.g., a user might be running an editor, spell-checker and printer, all for the same file.

(iv) **Computational Speed** – on a multiprocessing system, tasks are sub-divided and executed concurrently on different processors.

Basically, cooperating processes require communication in order to share data and to synchronize their actions.

Communication between cooperative processes may be described in terms of the ***producer-consumer model***. One process, e.g., an editor, has information it wants to produce for consumption by another process, e.g., a printer-driver. This may be done using a mutual buffer – the producer writes to the buffer and the consumer reads from it.

The shared buffer may be ***un-bounded*** or ***bounded***:

- ***un-bounded***: no size limit is placed on the buffer. The producer may continue to produce and write to the buffer as long as it wants to,
- ***bounded***: There is a strict limit on the size of the buffer. If it is full the producer must wait until the consumer removes some data.

The buffer may be implemented either by

(i) ***Physically*** by a set a shared variables (i.e., shared memory addresses). Implementation of this approach is the responsibility of the programmer.

(ii) ***Logically*** by *Message Passing* using an **InterProcess Communication Facility**. Such a system is implemented by the OS.

Interprocess Communication (IPC) provides a *logical* communication link via a *message passing facility* with two fundamental operations:

- send message *and*
- receive message.

Two standard forms of IPC are: ***Direct*** and ***Indirect***.
Furthermore, IPC may be: ***Symmetric*** or ***Asymmetric***

Direct Communication:

Each process must nominate explicitly the process with which they want to communicate.
Pairs of communicating procs must know each others ID's in order to establish a ***link***
With this system: a link is established by the two processes automatically, there are exactly two procs associated with a link, and there can be at most one link between two procs.
This is an example of ***Symmetric addressing***.
In the ***Asymmetric*** version, only the sender needs to know the recipient's address – the recipient listens out for any messages addressed to it (this is like making a phone call to someone).
Disadvantage: procs must have details of each other before they start to communicate. Only one link between proc. Only 2 procs can communicate at one time.

Indirect Communication:

The alternative is to have a number of ***ports*** (Mailboxes) in the system. Each has a unique ID.

- For procs to communicate they just need to know the name of the shared mailbox.
  More that two procs can share a mailbox.
  Two procs may share more than one mailbox.

Disadvantage: If one proc writes some data to a mailbox, and two others try to read from it, which get the data? Possible solutions:

- The Mailbox is “owned” by only one proc. Then several procs can write to it, but only one can read.
- Mailboxes are owned only by Operating System. Permissions then granted to process to create and delete mailboxes, and to send or receive messages

Example 1: Pipes

- using the `pipe(name)` function, one process creates a pipe. Here *name* is a `int` array with two elements.
- To put a message into the pipe, write to `name[1]`
- To read a message from the pipe, read from `name[0]`

08Pipes.c

```
/* Which: 08Pipes.c
   What:  a parent and child communicate via a pipe
   Who:   Niall Madden (Niall.Madden@NUIGalway.ie)
   When:  Week 7, 1920-CS211
   Why:   an example of using (unix) pipes */
#include <unistd.h>
#include <stdio.h>

int main(void )
{
int child_pid, parent_pid, pipename[2];
pipe(pipename); // make the pipe
child_pid = fork();
```

Example 1: Pipes

The parent will now send a message (its own pid) to the child.

08Pipes.c

```
   if (child_pid != 0) // This is the parent
16 {
     parent_pid = getpid();
18   printf("I am the parent (%d); My child is %d\n",
        getpid(), child_pid);
20   write(pipename[1], &parent_pid, 4);
   }
```

And the child will read it from the pipe.

08Pipes.c

```
22 else    // The child
   {
24   read(pipename[0], &parent_pid, 4);
     printf("I'm the child (%d); what I read from pipe: %d\n",
26      getpid(), parent_pid);
   }
```

Exercises

Exercise (7.1)

According to the manual, a call to `fflush(stdio)` "forces a write of all user-space buffered data for the given output or update stream via the stream's underlying write function". Explain what this means. Compare the output of `03CountTo10.c` with and without that call. What is the difference? How can it be explained?

Exercise (7.2)

In the example in `06ChildKillsParent.c`, the child process sends a `kill` signal to the parent. Write a version of this when the parent sends the kill signal to the child. How does the output change?

Exercise (7.3)

In the example in `06ChildKillsParent.c`, the child process sends a `kill` signal to the parent, the child should be "re-parented". Verify this happens by checking that `getppid()` changes.