

Week 7: Processes Communication

CS211: Programming and Operating Systems

Wednesday and Thursday, 24+25 March 2021



Usual reminders...

	Mon	Tue	Wed	Thu	Fri
09:00					
10:00					
11:00					
12:00					
13:00				<i>Recorded</i>	
14:00					
15:00	LAB		<i>Recorded</i>		
16:00	LAB				

- 1 This week, we have live-but-recorded classes on Wednesday at 3, and Thursday at 1.

In Week 7 of CS211, we'll study

- 1 Part 1: Recap on Week 6
 - ... The Process (again)
 - `fork()`
- 2 Part 2: More Process Programming
 - `wait()`
 - Overlaying a process
 - Process Termination
- 3 Part 3: Signals
 - `kill()`
 - `signal()`
- 4 Part 4: Interprocess communication
 - producer-consumer model
 - IPC
- 5 Part 5: `pipe()`
- 6 Exercises

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

CS211
Week 7: Processes Communication

Start of ...

PART 1: Recap on Week 6

In Week 6, we started our move towards the “Operating System” part of the course.

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.***

We examine the creation of processes by calls (from a C program) to the `fork()` function.

`fork()` is defined in the the `unistd.h` header file.

IMPORTANT: `fork()` is not included in the `mingw` compiler that comes with `code::blocks` on Windows. Try

- https://www.onlinegdb.com/online_c_compiler
- <https://www.programiz.com/c-programming/online-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>

- 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 perform.

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.

02Fork.c from Week 6

```
1 // An example of forking a process
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main(void )
6 {
7     int pid1, mypid;
8
9     pid1 = fork();
10    mypid = getpid();
11
12    printf("I am %d\t", mypid);
13    printf("Fork returned %d\n", pid1);
14    return(0);
15 }
```

Typical output:

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

01ParentsPID.c

```
6  int main(void )
   {
8    int pid1;
    pid1 = fork();
10   printf("I am %d\t", getpid());
    printf("fork returned %5d\t", pid1);
12   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

```
6  int main(void) {
   int pid1, mypid;
   pid1 = fork();
   mypid = getpid();
   if (pid1 != 0)
10  printf("I'm the parent. My PID's %d, my child's is %d\n",
        mypid, pid1);
12  else
   printf("I'm the child. My PID's %d, my parent's is %d\n",
        mypid, getppid());
14  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

CS211
Week 7: Processes Communication

END OF PART 1

CS211
Week 7: Processes Communication

Start of ...

PART 2: More Process Programming

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 certainty what will happen in what order.

Here is an example of a program for which we cannot predict the order of the output:

03CountTo10.c

```
8  int main(void) {
    int i;
    fork();
10  printf("Watch me (%d) count to 10: ", getpid());
    for (i=1; i<=10; i++)
12  {
        sleep(2*rand()%2); // sleep for 0 or 2 seconds
14  printf("%3d...", i);
        fflush(stdout);
16  }
    printf("\n");
18  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...
```

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");
  return(0);
24 }
```


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.

050verlay.c

```
int pid1 = fork();

12  if (pid1 == 0) // this is the child
    {
14      printf("This is process %d\t", getpid());
      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);
```

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.

CS211
Week 7: Processes Communication

END OF PART 2

CS211
Week 7: Processes Communication

Start of ...

PART 3: Signals

A first look at communicating between processes

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  // more standard stuff goes here
   .
7  .
   int kill(pid_t pid, int sig);
```

06ChildKillsParent.c

```
8 #include <signal.h>
10 int main(void )
11 {
12     printf("Parent process has pid=%d\n", getpid());
13     int child_pid = fork();
14     if (child_pid == 0) // child path
15     {
16         sleep(3);
17         printf("%d telling parent process (%d) to terminate\n",
18               getpid(), getppid() );
19         kill(getppid(), SIGKILL);
20     }
21     else // parent path
22     {
23         sleep(20); // sleep for 20 seconds
24         printf("**%d: never gets to this line, do we?\n",
25               getpid() );
26     }
```

Note - when the child sends the `kill` signal to the child, the child process also terminates.

The `kill()` function can send other signals, but most (such as `SIGABRT`, `SIGILL`, `SIGQUIT`, `SIGTERM`) are just variants on `SIGKILL`).

However, there are signals that perform other tasks:

- `SIGSTOP`: stop (temporarily; i.e., pause);
- `STGCONT` resume (if stopped).
- `SIGUSR1` a user-defined signal; (see next section)
- `SIGUSR2` another user-defined signal;

Exercise (7.1: `SIGSTOP`, `SIGCONT`)

Write a C program that works as follows:

- The parent process *forks* a child;
- The child process outputs `1, 2, ..., 9, 10`, but *sleeps* for 1 second between each number.
- The parent
 - *sleeps* for 3 seconds, then outputs a message, sends `SIGSTOP` to the child.
 - *sleeps* for 3 seconds, then outputs another message, sends `SIGCONT`
 - *sleeps* for 4 seconds, then outputs a final message,

With the `signal()` function, we can send a signal that tells the process to preform a specific action when it receives a `SIGUSR1` or `SIGUSR2` signal.

07SIGUSR1.c

```
6 #include <stdio.h>
7 #include <unistd.h> // need this for fork
8 #include <signal.h> // defines signal()
10 void signal_handler1(int sig);
12 int main(void )
13 {
14     int child_pid=fork(); // New process will send signal to parent
16     if (child_pid != 0 ) // I'm the parent
17     {
18         signal(SIGUSR1, signal_handler1); // What to do if I get signaled
19         sleep(100); // Wait potentially a long time
20         printf("%d got woken \n", getpid());
21         kill(child_pid, SIGKILL); // tell watcher to terminate.
22     }
```

07SIGUSR1.c

```
24  else // I am the child
    {
    26      printf("I am the child (%d)\n", getpid());
    26      printf("Should %d signal parent (%d)? ('y'/'n')\n",
              getpid(), getppid());
    28      if (getchar() == 'y')
        {
    30          printf("Sending siguser 1 to parent\n");
          kill(getppid() , SIGUSR1);
    32      }
    }
    34  return(0);
}

38  void signal_handler1(int sig) {
    printf("Proc %d called signal_handler1() with signal %d\n",
          getpid(), sig);
40  }
```

What the process does **after** it has finished calling the signal handler function depends on the situation. In the above example, it stopped **sleeping**.

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.

Exercise (7.2: SIGUSR1, SIGUSR2)

Write a C program like `07SIGUSR1` but that sets up signal handlers for both `SIGUSR1` and `SIGUSR2`. The child process should prompt the user to input `1` (for `SIGUSR1`) or `2` (for `SIGUSR2`), and then send that signal to the parent.

CS211
Week 7: Processes Communication

END OF PART 3

CS211
Week 7: Processes Communication

Start of ...

PART 4: Interprocess communication

(IPC)

Part 4: Interprocess communication

The `kill()` system call can be considered as a form of **communucation** between process (though 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.

Part 4: Interprocess communication

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 of 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

CS211
Week 7: Processes Communication

END OF PART 4

CS211
Week 7: Processes Communication

Start of ...

PART 5: pipe()

Part 5: pipe()

A `pipe` is a means of communication, based around the `pipe()`, `write()`, `read()` and functions that allows one process to send data to another. Typically, the data can be anything, but we will use examples of sending `integers`. It is a form of *symmetric direct* communication.

- 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]`

In this example a process will create a pipe, then fork a subprocess, and send its PID to this new subprocess.

Part 5: pipe()

08Pipes.c

```
2  /* Which: 08Pipes.c
   2  What:   a parent and child communicate via a pipe
   4  Who:   Niall Madden (Niall.Madden@NUIGalway.ie)
   4  When:  Week 7, 2021–CS211
   4  Why:   an example of using (unix) pipes */
6  #include <unistd.h>
   6  #include <stdio.h>

   8
  10  int main(void )
  10  {
  12  int child_pid, parent_pid, pipename [2];
  12  pipe(pipename); // make the pipe
   12  child_pid = fork();
```


Part 5: pipe()

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

08Pipes.c

```
16 if (child_pid != 0) // This is the parent
17 {
18     parent_pid = getpid();
19     printf("I am the parent (%d); My child is %d\n",
20           getpid(), child_pid);
21     write(pipe_name[1], &parent_pid, 4);
22 }
```

And the child will read it from the pipe.

08Pipes.c

```
23 else // The child
24 {
25     read(pipe_name[0], &parent_pid, 4);
26     printf("I'm the child (%d); what I read from pipe: %d\n",
27           getpid(), parent_pid);
28 }
```

Exercises

Exercise (7.3)

According to the manual, a call to `fflush(stdout)` “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 (7.4)

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 (7.5)

In the example in `06ChildKillsParent.c`, the child process sends a `kill` signal to the parent, the child should be “re-parented”, but this depends on the Operating System. Check what happens on the system you use by checking if `getppid()` changes.