Systems Notes

Computer Science

Alex Norton

ajn123@vt.edu

May 21, 2014

Contents

Chapter 8: Shell

1 Process

The first thing to understand is that a **computer program** is a passive collection of instructions; a **process** is the actual execution of those instructions.

A **process** in user mode is not allowed to execute privileged instructions. The only way for the process to change from user mode to kernel mode is via an exception such as an **interrupt**, a **fault** or a **trapping system call**.
Processes can have three states:

1. **Running** The process is currently running on a CPU.
2. **Ready** Process could make progress if CPU were available
3. **Blocked** Process is waiting for something (memory, signal, time, etc)

2 Context switching

In computing, a **context switch** is the process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system. What constitutes the context is determined by the processor and the operating system.

A context switch follows these 3 steps:

1. saves the contents of the current process.
2. restores the saved context of some previously preempted process
3. passes control to this newly restored process

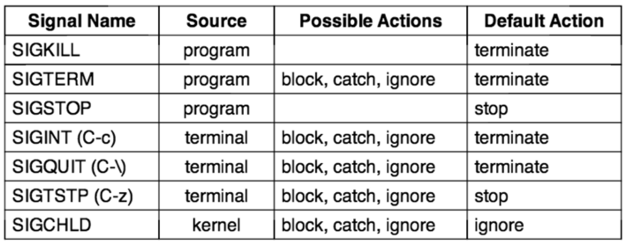
3 File Descriptor

A file descriptor is an indicator for a way of accessing a file. In simple words, when you open a file, the operating system creates an entry to represent that file and store the information about that opened file. Depending on the options you give, you can read and write to this file descriptor.

4 Signal

A **signal** is a message that notifies a process that an event of some type has occured in the system (just like when you press a button on your phone, a message is sent to the operating system). Each signal corresponds to some kind of system event. For example, a signal can be used to cancel background jobs.

A pending signal is a signal that has been sent but not recieved. A process can also block a CERTAIN signal. In this case, blocked sigals can be sent but will not be received until the signal is unblocked.



Signal Name	Source	Possible Actions	Default Action
SIGKILL	program		terminate
SIGTERM	program	block, catch, ignore	terminate
SIGSTOP	program		stop
SIGINT (C-c)	terminal	block, catch, ignore	terminate
SIGQUIT (C-\)	terminal	block, catch, ignore	terminate
SIGTSTP (C-z)	terminal	block, catch, ignore	stop
SIGCHLD	kernel	block, catch, ignore	ignore

Figure 1: Table of possible signals

5 Pipelining

Pipelining works by setting the standard output(1) of the first command to the standard input(0) of the second command in the pipeline. here are a couple of system calls that you may be interested to understand what is happening in more detail, in particular, fork(2), execve(2), pipe(2), dup2(2), read(2) and write(2).

- The dup2 function takes two parameters $dup2(old, new)$ The pointer of old will replace the pointer of new when the function is called. (code example below)

Integer Value	**Name**
0	Standard Input (Stdin)
1	Standard Output (Stdout)
2	Standard Error (Stderr)

Table 1: Integer Values and their File descriptors

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

#define IN 0
#define OUT 1

int main(void)
{
  char string[] = "Hello, world!\n";
  char readbuffer[80];

  // Creates the pipe using the integer array of size two.
  int fd[2];
  pipe(fd);

  // Fork new process
  if(fork() == 0)
  {
    // Copy the write end of pipe to standard out.
    dup2( fd[OUT], OUT );
    // Close read and write end of pipe.
    close( fd[IN] );
    close( fd[OUT] );
    // Child Process: execute new process
    char *cmd[] = {"ls", "-la", (char *) 0};
    execvp("ls", cmd);
  }
  else
  {
    // Clise write end of pipe.
    close( fd[OUT] );
    // Parent Process: Read string from the read side of pipe.
    read( fd[IN], readbuffer, sizeof(readbuffer) );
    printf("Received string: %s", readbuffer);
  }
  return(0);
}
```

6 Fork

This is an example of the fork method in C. fork() clones a process from a process. This new process can be used to execute another process or do other things. NOTE: Once you clone a process you have no idea which order your clones will run in, your code should not depend on the order. Another thing to remember is that fork returns twice, once in the parent and once in the child (the new process you just created). The cloned process is exactly the same except for the return value of fork(). In the child process fork will always return 0 and in the parent it will return the process ID of the child to the parent process. Your code can just check for the child process with == 0 and the parent with an else.

Fork returns the PROCESS ID OF ITS CHILD to the parent process, so that the parent knows its PID of the child to keep track of it. Fork() returns 0 to the child, you don't need it except to check that you are processing the child.

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

int main(){
	int x = 1;

	int pid = fork();// pid contains the childs pid for the
	   parent process.
	if (pid== 0)
	{// only child executes this
		printf("Child, x = %d\n", ++x);
	}
	else {
	// only parent executes this
		printf("Parent, x = %d\n", --x);
	}// parent and child execute this
	printf("Exiting with x = %d\n", x);
	return 0;
}
```

7 reentrant

A computer program in **reentrant** if it can be interrupted in the middle of its execution and then safely called again. An interuption can come from a signal or a jump. Once the reentered invocation completes, the previous invocations will resume correct execution.

- Printf() is a NON reentrant function, meaning that it is not safe to interrupt.

8 Signal Handlers

Signal handlers are used to deal with the different types of signals on a USER LEVEL.

When a handler function is invoked on a signal, that signal is automatically blocked (in addition to any other signals that are already in the process's signal mask) during the time the handler is running.

9 PID-process identifier

A PID is a number used to temporarily uniquely identify a process.
One may use the command ”ps j” to see PPID (parent process ID), PID (process ID), PGID (process group ID) and SID (session ID) of processes.

Every process can have a PID, parent ID, and a group ID(gid). A group ID means that all of the processes share the terminal together and you do not need to transfer control for these processes. When you send a signal to a terminal, everything in the terminal control group (same gid) will receive that signal. When you send a signal via kill(), it depends on how you structure the arguments.

10 Resources

Below are other resources I would recommend looking at for further readings.

1. http://stackoverflow.com/

2. ftp://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_toc.html#TOC572, GNU documentation on job control.

3. passes control to this newly restored process

Thread Pool

1 Threads

A **Thread** is a logical flow that runs in the context of the program. Threads share all the same date structures (virtual address space), processes DO NOT. It is also important to note that each thread has its own thread context including a unique integer thread ID (TID), stack,stack pointer, program counter, general purpose registers, and condition codes.

Threads are scheduled automatically by the kernel and are known by an integer ID.
Each process begins life as a single thread called the main thread. At some point the main thread creates a peer thread, and from this point in time the two threads run concurrently. Eventually, control passes to the peer thread via a context switch, because the main thread is doing something slow (read, sleep, or disk access).

Threads are organized in a pool of peers. The main thread is different in that it is always the first thread to run in the process. Each peer can read and write the same shared data.

Variables in threaded C programs are mapped to virtual memory according to their storage classes:

- Global Variables (Shared): A global variable is any variable declared outside of a function. At run, the read/write area of virtual memory contains exactly one instance of each global variable that can be referenced by any thread.

- Local Automatic Variables (Not Shared): This is a variable declared inside a function WITHOUT the static attribute. At run time, each thread's stack contains its own instances of any local automatic variables.

- Local Static Variables (Shared): This is a variable declared inside a function with the static attribute. As with global variables , the read/write area of virtual memory contains exactly one instance of each local static variable in our program.

2 Mutexes

As shown above, threads share global and static variables of a process. This leads to some problems. What if one thread accesses a global variable, changes it and then another thread looks at the just updated same global variable. Remember that your code should never be dependent on what order threads execute in. That update result could change the flow of your program and visa versa if that same thread looks at a global variable before another threads updates its value. This is where critical sections become important.

Critical sections prevent data accesses that could cause problems such as the above example. One way to solve this problem is through the use of mutexes (Great stackoverflow explanation here).

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

#define NUMTHRDS 2

pthread_t t [ NUMTHRDS];
int coin_flip;

pthread_mutex_t flip_done;
static void *thread2(void *_){
       pthread_mutex_lock(&flip_done);
       printf("Thread 2: flipped coin %d\n", coin_flip);
}


static void * thread1(void * _)
{
       coin_flip = 1;
       pthread_mutex_unlock(&flip_done);
       printf("Thread 1: flipped coin %d\n", coin_flip);
}
```

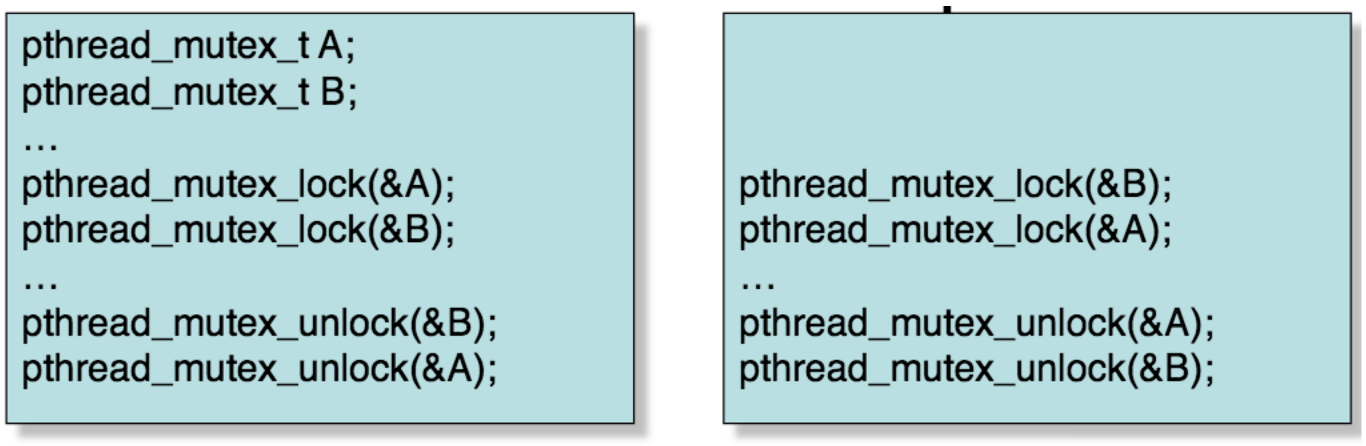
```
int main()
{
       pthread_mutex_init(&flip_done, NULL);
       pthread_mutex_lock(&flip_done);
       pthread_create(&t[1], NULL, thread2, NULL);
       pthread_create(&t[0], NULL, thread1, NULL);
        pthread_mutex_destroy(&flip_done);
       //must have this as main will block until all the
          supported threads are done
       pthread_exit(NULL);
       return 1;
}
```

3 Race Conditions

One thing to watch out for in threads is a race condition. A race condition occurs when two or more threads read and write a shared variable, and the final result depends on the execution of those threads.

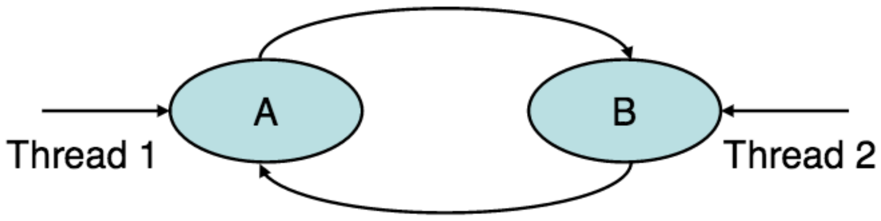
4 Deadlock and Livelock

Deadlocks can occur when two or more threads are each waiting for the other to finish, and thus neither gets done. The picture below illustrates this point. Imagine that one thread executes the code on the left (we will that thread X) and another thread executes the code on the right (Thread Y). Now thread X executes and grabs the mutex A, while thread Y executes and grabs mutex B. Thread X understands that it must wait for mutex B to become available. However thread Y is also waiting to obtain mutex A. Both threads are waiting for another thread to give something up that it will not and we have a deadlock. To always avoid a deadlock, you must always acquire locks in the same order. In this case, decide that A will always be acquired before B on all threads.



```
pthread_mutex_t A;
pthread_mutex_t B;
…
pthread_mutex_lock(&A);
pthread_mutex_lock(&B);
…
pthread_mutex_unlock(&B);
pthread_mutex_unlock(&A);
```

```
pthread_mutex_lock(&B);
pthread_mutex_lock(&A);
…
pthread_mutex_unlock(&A);
pthread_mutex_unlock(&B);
```



A livelock is another kind of failure in which a non-blocked thread cannot make any progress because it keeps retrying an operation it will always fail. Click here for a nice description with pictures.

5 Semaphores

A semaphore is a variable or abstract data type that is used for controlling access to multiple threads or processes
Semaphores provide a great way to ensure mutuaally exclusive access to shared variables. A semaphore that is used to protect shared variables is called a binary semaphore becuase its value is always 0 or 1.

Another important use of semaphores ,besides providing mutual exclusion, is to schedule accesses to shared rescources.

Think of semaphores as bouncers at a nightclub. There are a dedicated number of people that are allowed in the club at once. If the club is full no one is allowed to enter, but as soon as one person leaves another person might enter.

Memory Allocation

1 Start

Web Server

Here are some definitions to start:

- **URI (Uniform Recsource Identifier**- The suffix of the corresponding URS that includes the filename and optional arguments.

- **Network Port**- A number signaling what different type of traffic is sent over (email, internet, texts). Ports talk to other devices as well as other networks through these ports.

- **Sockets**- Let the client and server talk over ports by providing an API that allows communication between the server and the client. Sockets talk to applications.

1 Protocol Layering

A **protocol** is a set of layers used to communicate with eachother. Each instance of a protocol talks virtually to its peer using the protocol. Each instance of a protocol uses only the servicese of a lower layer. Protocols must provide two things:

- Provides a naming scheme for host addresses, a unique address to identify itself.

- Provide a delivery mechanism called a packet which consists of a header and a payload. Header contains size, source, and destination. Payload contains bits sent from source.

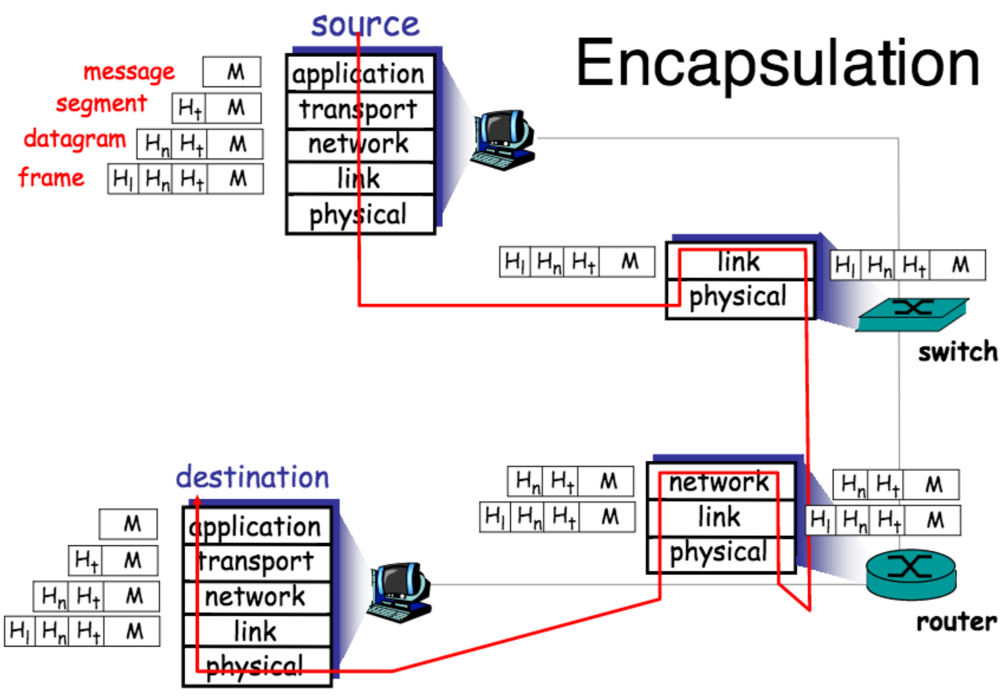


Figure 1: How protocol layering works

Groups	**Treatment X**	**Treatment Y**
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

Table 1: The effects of treatments X and Y on the four groups studied.

Table 1 shows that groups 1-3 reacted similarly to the two treatments but group 4 showed a reversed reaction.

Saturday, 27 March 2010

1 Bulleted list example

This is a bulleted list:

- Item 1
- Item 2
- . . . and so on

2 This is an example experiment

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

3 This is another example experiment

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Formulae and Media Recipes

Media

Media 1

Compound	**1L**	**0.5L**
Compound 1	10g	5g
Compound 2	20g	10g

Table 1: Ingredients in Media 1.

Formulae

Formula 1 - Pythagorean theorem

$a^2 + b^2 = c^2$