CS 111

Scribe Notes for 4/13/05

by Jon Binney, Alexandra, Adriana Magana, Kathy Lee

Processes

Process Control Block

For each process, the kernel maintains some state in memory. This state includes the ID of the process, the memory of the process, and a few other things. When a new process is created using fork(), it inherits most of its information. Below is listing of all of the parts of the control block for each process, and whether it's the same, different, or a copy of the process which it was created from.

State info	**(S)hared, (D)ifferent, or (C)opied**
pid (process ID)	D
ppid (parent process ID)	D
registers	C
eip (instruction ptr)	C
VM state	C
signal handlers	C
file descriptor table	C
file descriptors	S
accounting info (identity (userID, groupID), restrictions)	C

Detail Note: The file descriptor *table* is copied, but individual file descriptors are shared. This means, for example, that a file opened in the child is not available in the parent. (Opening a new file adds a new file descriptor to the file descriptor table, and file descriptor tables are not shared between parent and child.) However, if the parent opens a file `f` and then forks, both the parent and child will have access to that open `f` file. Furthermore, the parent and the child truly share `f`: if one of them reads 10 bytes from the file, the other one sees the file pointer jump ahead by 10.

System Calls

- fork()
- exec()
- waitpid() => can only wait for your children
- exit()

fork() creates a new process which is a copy of the parent process. But how will the new process know that it is the child, and not the parent? fork() solves this by returning a value of zero to the child process, and by returning the pid of the child to the parent. (a positive integer). If there is an error and the child process cannot be created, then a value of -1 is returned to the parent.

Consider the following code for creating a new process using fork(). What will happen when the code is run? Carefully consider the order in which commands are executed!

```
1  int num = 0;
2  int main(int argc, char *argv[]) {
3      pid_t p = fork();
4      printf("Pre-fork...\n");
5      if (p > 0) {
6          printf("P, C is %d\n", p);
7      } else if (p == 0) {
8          printf("C %d, P is %d\n", getpid(), getppid());
9      } else
10         abort();
11 }
```

What will be printed by this segment of code? When the instruction pointer reaches the fork() command, an identical child process is created and run. It starts at the first command after the fork(), so the "printf("Pre-fork...\n")" command is run twice! Then the if statement on line 5 causes the parent to print out the child's pid, and the child to print out its own pid, and its parent's. Finally, if the return value of fork() is negative, the program aborts. Assuming a pid of 3 for the parent process, and 42 for the child process, would be:

```
Pre-fork...
P, C is 42
Pre-fork...
C 42, P is 3
```

The order of the statements may vary each time the program is run, because the system is alternating between the two processes. In this case, the parent process was finished outputting before the child started. If the processes were scheduled differently, however, we could see the lines in a different order. Next let's modify the code so that "Pre-fork..." is printed only once, by putting line 4 before the fork().

```
1  int num = 0;
2  int main(int argc, char *argv[]) {
3      printf("Pre-fork...\n");
4      pid_t p = fork();
5      if (p > 0) {
6          printf("P, C is %d\n", p);
7      } else if (p == 0) {
8          printf("C %d, P is %d\n", getpid(), getppid());
9      } else {
10         abort();
11     }
12 }
```

Now the parent process prints out "Pre-fork...", then creates the child process using fork(). They each print out some info, and all is well. Or is it? What if the kernel decided to switch between processes halfway through one of the printf commands? Then the output might be interleaved, resulting in sompething like:

```
Pre-fork...
P, C is C 42, P is 3
42
```

Is this kind of problem possible? Lets look at a couple of possible implementations of the printf() function to find out. The most obvious way to implement printf() is to loop through the given string, calling write() on each character to write it to the screen.

```
1  printf(const char * s) {
2      while(*s != '\0') {
3          write(s, l, f);
4          s++;
5      }
6  }
```

With this implementation of printf(), interleaving of output from multiple processes is a definite problem. There is nothing to prevent the kernel from switching to another process halfway through the string. In order to fix this problem, we need to note that system calls are atomic; meaning that the kernel will not switch to another process halfway through a system call. The solution, then, is to use just one system call instead of many of them in a loop.

```
1  printf(const char *s) {
2      write(s, strlen(s), f);
3  }
```

In this implementation of printf, we make just one call to write(), and tell it to print out the entire string. Since write() is a system call, we can be sure that it will finish before the kernel switches to another process.

When you run the program, the output appears to be correct. Then you try redirecting the output into a file, and notice that the resulting file looks like:

```
Pre-fork...
P, C is 97
Pre-fork...
C is 97, P is 45
```

Why is "Pre-fork..." printing out twice? The answer lies in the way printf is actually implemented by the operating system. When writing to the screen, printf works as we expect; when we call it with some string, it writes the string to the screen and exits. When it is writing to a file, however, printf() attempts to minimize the number of system calls. It stores all of the strings you tell it to write to file, and then periodically writes them all at once. This implementation looks something like (some of this is just pseudocode, but you should get the idea):

```
1  printf(const char * s) { (file f)
2      static char fbuf[8192];
3      static inf fpos = 0;
4      memcpy(fbuf + fpos, s, min(8192 - fpos, strlen(s)))
5      pos_t = min(8192 - fpos, strlen(s))
6      if(full) {
7          write(fbuf, 8192, f);
8          do again w/rest of string
9      }
10 }
```

When we call printf("Pre-fork...\n") in the parent before we fork, it might not actually get written to the file yet. Instead it is put into the buffer that printf() maintains. When we call fork(), the new process is a copy of the old process, including the memory, so it inherits the same buffer. Then when the child calls printf, whatever was in the buffer from before is printed out. To correct this, we must flush the buffer before forking. The corrected code is:

```
1   int num = 0;
2   int main(int argc, char *argv[]) {
```

```
3      int i, status;
4      printf("Pre-fork...\n");
5      fflush(stdout);
6      pid_t p = fork();
7      if (p > 0) {
8         printf("P, C is %d\n", p);
9      } else if (p == 0) {
10        printf("C %d, P is %d\n", getpid(), getppid());
11     } else
12        abort();
13  }
```

Now, since we call fflush(stdout), the buffer is flushed and the problem is fixed.

In our final version for this subsection, let's communicate a bit of information from the child to the parent. Now the parent will not exit until the child completes, and that the parent prints the child's exit status.

```
1   int num = 0;
2   int main(int argc, char *argv[]) {
3      int i, status;
4      printf("Pre-fork...\n");
5      fflush(stdout);
6      pid_t p = fork();
7      if (p > 0) {
8         printf("P, C is %d\n", p);
9         waitpid(p, &status, 0);
10        printf("C status is %d\n", status);
11     } else if (p == 0) {
12        printf("C %d, P is %d\n", getpid(), getppid());
13        for (i = 0; i < 1000000000; i++)
14           /* do nothing */;
15        exit(num);
16     } else
17        abort();
18  }
```

Signals

Well, *one* problem is fixed; but now the child counts up to a billion before exiting! The parent is going to wait for a long time on line 9. What if the parent decides that the child's computation is no longer important? We need some way for one process to interrupt another. That is, we need some form of **asynchronous notification**.

In C and Unix, asynchronous notification is implemented by *signals*. The operating system defines a set of signals, which are given names like `SIGSEGV` and `SIGKILL`. Each process defines an optional *handler* for each signal, which is just a function that will be called when the signal is received. The process control block contains a list of signal handlers. When a process receives a signal, the operating system interrupts the process -- even if it's in the middle of an infinite loop! -- and calls the relevant signal handler. When the signal handler returns, the rest of the program will pick up where it left off (unless, of course, the signal handler made the process exit).

The operating system will automatically generate some signals; for example, it generates a `SIGCHLD` signal when a child process dies, and the `SIGSEGV` ("segmentation violation") signal when the process accesses illegal memory. But other processes can generate signals too, using the `kill` function.

To define a signal handler, the process calls the `signal` function. The `handler` argument is a function pointer.

```
int kill(pid_t process, int signal);

typedef void (*sig_t)(int signal);     // function pointer type
sig_t signal(int signal, sig_t handler);
```

Note that signal handlers have exactly one argument, the signal number. There's no way to pass more data with a signal. If you want more complex inter-procedure communication, you need to send real messages.

Let's use the signal functions to do some asynchronous notification. Our first goal: Make the child process's exit status equal the number of `SIGUSR1` signals it received. Does the following code work?

```
1   int num = 0;
2   void h(int signal) {
3      num++;
4   }
5   int main(int argc, char *argv[]) {
6      int i, status;
7      printf("Pre-fork...\n");
8      fflush(stdout);
9      pid_t p = fork();
10     if (p > 0) {
11        printf("P, C is %d\n", p);
12        kill(p, SIGUSR1);
13        waitpid(p, &status, 0);
14        printf("C status is %d\n", status);
15     } else if (p == 0) {
16        signal(SIGUSR1, h);
17        printf("C %d, P is %d\n", getpid(), getppid());
18        for (i = 0; i < 1000000000; i++)
19           /* do nothing */;
20        exit(num);
21     } else
22        abort();
23  }
```

Close, but no cigar. Remember that after the `fork`, the child and the parent might run in any order. In particular, the parent might reach line 12, and call `kill`, before the child had a chance to install a signal handler! If this happens, the child's *default* handler will be called; and for `SIGUSR1`, the default handler kills the process. No good.

How do we get around this? The simplest way is to set up the signal handler *before* the fork. Signal handlers are copied between forked processes, so the child will have the right handler from the get go. But the *parent* must restore *its* signal handler. So:

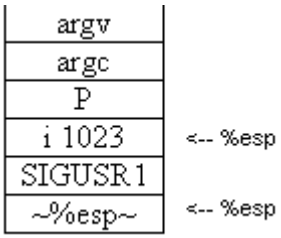
```
1   int num = 0;
2   void h(int signal) {
3      num++;
4   }
5   int main(int argc, char *argv[]) {
6      int i, status;
7      printf("Pre-fork...\n");
8      fflush(stdout);
9      signal(SIGUSR1, h);  // set handler before fork
10     pid_t p = fork();
11     if (p > 0) {
12        printf("P, C is %d\n", p);
13        signal(p, SIG_DFL);  // restore default handler
```

```
14        kill(p, SIGUSR1);
15        waitpid(p, &status, 0);
16        printf("C status is %d\n", status);
17     } else if (p == 0) {
18        printf("C %d, P is %d\n", getpid(), getppid());
19        for (i = 0; i < 1000000000; i++)
20           /* do nothing */;
21        exit(num);
22     } else
23        abort();
24  }
```

As you can tell, signals are somewhat hard to deal with. But it gets worse. Many function calls should not be used from within signal handlers, including old standbys like `malloc`. This is because the signal might have happened at the worst possible time. Furthermore, the old `signal` interface was *unreliable*. If a process was sent many signals at once, the system could lose some of them. Even worse, signal handlers were "one-off": the system would remove every handler H before calling that handler. Later revisions of the signal API have fixed many of these problems, but some of them persist. It is hard to use signals to build inter-procedure communication, but they are still better than anything else at stopping infinite loops and such things.

Child's Stack

This shows a picture of the child process's stack after a signal is delivered. The operating system has decided to deliver the signal *on top of the current stack*. The uppermost stack pointer `%esp` shows where the child was interrupted; the lower stack pointer corresponds to the signal code. Some aspects, such as return addresses, not drawn.



Threads

What are threads?

Threads are basic units of CPU utilization; they are multiple threads of control in a program that run concurrently in a shared memory space.

Threads are similar to processes, in that both represent a single sequence of instructisons executed in parallel with other sequences, either by time slicing or multiprocessing. Threads are a way for a program to split itself into two or more simultaneously running tasks, but unlike processes, threads are not isolated from each other: threads share the same memory space.

Threads are distinguished from traditional multi-tasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, share memory and other resources directly.

More specifically, a thread consists of a program counter (PC), a register set, and a stack space. Each thread

needs to have its own stack, since threads will generally call different procedures and thus have a different execution history. As mentioned previously, unlike processes, threads are not independent from each other: threads share with other threads their code section, data section (global variables – but have own local variables), and OS resources such as open files and signals.

Since the threads of a program share the same address space, one thread can modify data that is used by another thread. This sharing of memory can be both beneficial and harmful. The benefit of memory sharing is that it facilitates easy communication between threads. The downside is that poorly written programs may cause one thread to inadvertently overwrite another data being used by another thread.

Processes vs Threads

- Like processes, only one thread is running at a time and they share CPU
- Like processes, threads excecute sequentially within a process
- Like processes, threads can also create children
- Like processes, a thread can be in the running, blocked, ready, or terminated state.
- Unlike processes, threads aren't isolated from each other
- Unlike processes, threads share the memory addres space
- Unlike processes, threads can assist one another

Why are threads useful?

- Since they share common data, threads do not need to use interprocess communication
- Threads don't use up a lot of OS resources because threads do not need new address space, global data, program code or operating system resources
- Threads makes context switching to be faster because we would only have to save and/or restore PC, SP and registers.
- Responsiveness: allows a program to run even if part of it is blocked (i.e. a web browser allows user interaction in one thread while an image is being loaded in another).

fork vs clone

fork() is a system call that creates a new process as well as a copy of all associated data structures of the parent process -- that is, the VM is copied. Clone() is like fork except that instead of copying memory, the child shares the parent's address space. That is, instead of copying all data structures, the new process points to the data structures (such as the data structures representing the list of open files, the signal-handling information, and virtual memory) of the parent process. That is, a cloned process is effectively just a thread! A new process control block and a new process ID are created with the use of clone. Type man clone in the Linux shell to view API for this system call.

The following code depicts the issues encountered when using clone to create threads:

```
int main(int argc, char *argv[]) {
    pid_t p;
    if ((p = clone()) > 0)
            /* parent */
    else if (p == 0)
            /* child */
    else
            abort();
}
```

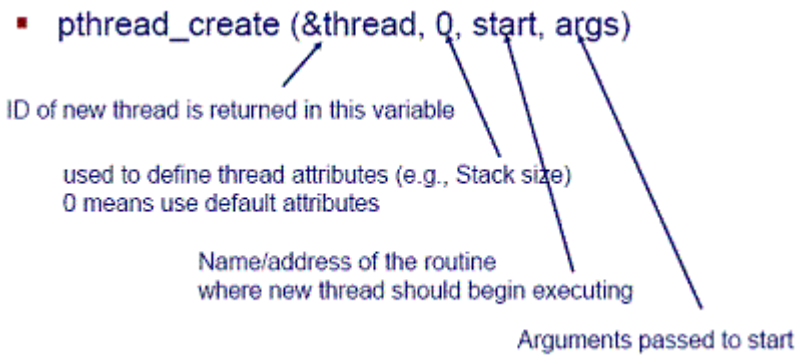
Problem with the above code: return value p is going to be the same in both parent and child because threads share the same memory space (p resides in the same address for both parent and child).

Therefore, because threads share memory space, a new interface to create threads is needed. Pthreads is a POSIX standard API for thread creation and synchronization functions that manipulate threads from C programs. This API contains the function pthread_create (which uses clone) to create threads, and it's precisely the use of this function that fixes the problem in the previous piece of code. The difference lies in that pthread_create takes in an address where the child process id is to be stored in the stack allocated to the child process. So p is no longer overriden. Pthreads also contains the function pthread_join. Termination of a thread unblocks any other thread that's waiting using pthread_join.

pthread_create API:

#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

- When the child process is created, it executes **start_routine(arg)**. (This differs from fork(), where execution continues in the child from the point of the fork() call.)
- When the start_routine(arg) function application returns, the child process terminates.
- Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by **thread**.
- pthread_create() returns zero if it was successful
- **attr** is used to define the stack size of the child process. If attr is NULL (or 0), the default attributes are used.



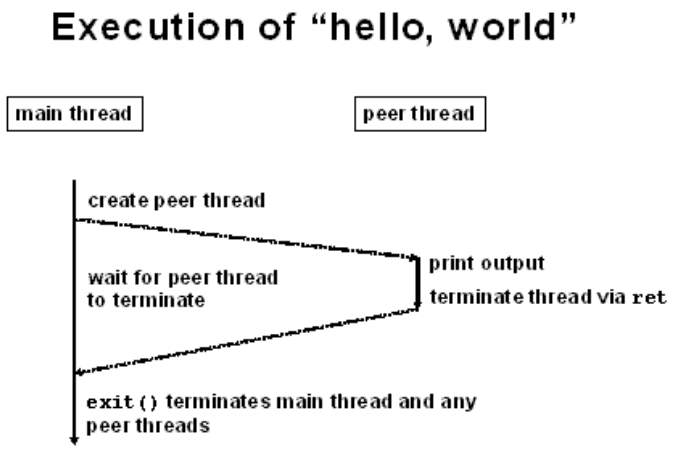
Following is an example program that uses pthread_create. This example was not discussed in class but I found it to be helpful in understanding how pthread_create works.

Example: Hello World

```
void *print_message(void *ptr);

void main (void)
{
  pthread_t thread1, thread2;
  /* create first thread */
  if (pthread_create(&thread1, NULL, print_message, (void*) "Hello")) {
    perror("pthread_create : can't create thread 1");
    exit(-1);
  }
  /* create second thread */
  if (pthread_create(&thread2, print_message, (void*) "World")) {
    perror("pthread_create : can't create thread 2");
    exit(-2);
  }
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  exit(0);
}
```

This is what happens during execution:



figures presented in the thread section can be found at: http://www.cis.upenn.edu/~lee/05cis505/slides_pdf/lec3-processes-v3.pdf

Scheduling

What is scheduling?

When there are more than one process exist, scheduling refers to the task of managing CPU sharing among those processes. Whenever a process want to use the CPU, it make a request to scheduler, and if the CPU is available, the scheduler will allocate this process to CPU. Whenever the running process wait for I/O, scheduler will allocate another process who is waint to CPU. With the scheduling mechanism, the CPU utiliation is optimal.

This mechanism overlapping computation to utilize resources efficiently. The CPU utilization is max and the throughput of total amount of work done is max as well. Since all processes share the CUP, the latency which is delay to completion will occur, and there is waiting time as well. The user latency which is delay for user actions is acknoledged