Processes, System Calls and Exception Handling in Unix/Linux

Harry H. Porter III

Portland State University
cs.pdx.edu/~harry

December 6, 2015

Normally the CPU executes one instruction after the other. This is called **sequential execution**. This sequence constitutes the CPU's **flow of control**.

We can describe the sequential execution of a CPU with the following pseudo-code:

```
LOOP FOREVER
        FETCH NEXT INSTRUCTION
        INCREMENT PC
        DECODE INSTRUCTION
        EXECUTE INSTRUCTION
ENDLOOP
```

This simplistic view assumes there is only one CPU (not multiple cores) and it assumes that there is no pipelined execution. Nevertheless, it describes the operation of many practical computers.

Normally, after the execution of one instruction, the CPU will execute the instruction that immediately follows it in memory.

Notice that the "execute" step may involve updating the Program Counter (PC), thus causing a jump, call, or return (i.e., a **transfer of control**) on the next cycle. So exceptions to the sequential execution of instructions can occur whenever the following instructions are executed, but these are effectively under program control. They occur only when execution reaches one of these instructions.

CALL
RETURN
JUMP
CONDITIONAL BRANCHES

Other than that, there seems to be no way to avoid sequential instruction execution. This means that the computer is only capable of reacting to events when the code explicitly checks for these events.

Asynchronous events are events that can come at any time. Here are examples:

The disk completes a read or write operation
A packet arrives on a network connection
An alarm clock or timer goes off
A program tries to do something invalid and cannot continue execution
The user types a keyboard key or presses a mouse button

For asynchronous events like these, there is a mechanism that breaks the sequential execution of instructions. This allows the computer to immediately respond to the event, regardless of what it was busy doing when the event occurred.

We call changes to the normal sequential flow of control "exceptions". The CPU is equipped with hardware that facilitates exception processing. In addition, there are higher-level mechanisms in software that also perform something similar to the hardware exception processing.

At the software level, there are a number of different techniques and mechanisms that break the traditional model of sequential execution in the FETCH-INCREMENT-DECODE-EXECUTE cycle. The main techniques in Unix/Linux are:

- TIME SLICING / PROCESS SWITCHING / MULTITASKING
- SIGNALS
- LONG JUMPS

At the lowest level, exception handling is done in the hardware. Hardware exceptions handled by the CPU are typically very fast, with little overhead. At the higher-level, when the sequential flow of execution is altered via the software techniques listed above, there is more overhead.

We use the term **exceptions** to refer to the exceptional control flow that happens at the low, hardware level. Here is a rough classification of the different kinds of exceptions that can occur:

- **Hardware Exceptions**
  - **Asynchronous Exceptions** (Interrupts)
    - Examples:
      - Timer/Clock interrupts
      - Disk I/O completion
      - Interrupts from other I/O devices
  - **Synchronous Exceptions**
    - **Traps**
      - Examples:
        - System Calls
        - Other traps
    - **Aborts**
      - Examples:
        - Machine check
    - **Faults**
      - Examples:
        - Arithmetic problems (like zero-divide)
        - Unimplemented instruction
        - Page Faults
        - Protection errors

Asynchronous exceptions are also called **asynchronous interrupts** or simply **interrupts**. Such an event can occur at any moment and the occurrence of the event is unrelated to which instructions are being executed. Asynchronous interrupts come from outside the CPU's FETCH-INCREMENT-DECODE-EXECUTE cycle. Various I/O devices will cause asynchronous interrupts to indicate they have completed performing the I/O operation.

The timer can be thought of as an I/O device although it is located inside of the CPU and isn't involved in transferring data from the computer to any other device. Instead, the timer causes **periodic interrupts**.

A **timer interrupt** is something that occurs on a regular basis and is used for **time-slicing**. For example, a timer interrupt might occur every millisecond. The timer interrupt occurs when a user-level application process is running. The sequential execution of the process is interrupted and control is transferred back into the kernel. These timer interrupts allow the kernel to regain control of the CPU. The kernel uses timer interrupts to switch from one process to the next. In this way, all processes can make forward progress, even though there is only one CPU. This is the

essence of **multitasking**. Hardware support for timer interrupts is required in order to allow the OS to implement multitasking.

A **synchronous exception** occurs as the result of executing some particular instruction. An exception that is caused intentionally is called a **trap**. The programmer has included an instruction in the program that, when executed, will cause a trap. The most common example is the **system call** (i.e., **syscall**) instruction. The programmer will include a syscall instruction whenever it is necessary to request a service from the kernel.

The syscall instruction is used to invoke a kernel function. Here are some functions that will execute a syscall instruction whenever they are invoked.

open(), close(), read(), write(), fork(), wait(), exit(), execve(), …

What they have in common is that they all request a service that must be performed by the kernel. These operations cannot be done within user-level code. There is only one syscall instruction, but it can be used to invoke any of the kernel functions.

These functions are called **wrapper functions**. The wrapper functions execute in user address space as **user-level code** running in **user mode**. The wrapper function doesn't do the actual work; instead it makes a system call into the kernel. For each wrapper function there is a corresponding **syscall trap handler function** in the kernel. The handler function is invoked as part of the exception processing. Later, the handler will makes a return to the wrapper function (which is user-level code) using the **RETI** (**Return From Interrupt**) instruction..

There are hundreds of these system calls in Unix/Linux and there is pretty good standardization across difference versions of Unix/Linux. Most system calls respect the specifications documented in the POSIX standard. **POSIX** is an agreed-upon standard specification that covers all Unix/Linux software. Almost all system calls on your Unix/Linux computer are **POSIX compliant**, but a few are OS-specific. (You should avoid anything that is not POSIX compliant.)

There may be other traps. For example, many CPUs have a **breakpoint instruction** to help debuggers like **gdb** do their work.

Many CPUs include circuitry that is always checking quietly to verify that the CPU is operating properly. This circuitry can be ignored by the programmer and the programmer does not need to understand anything at all about it. The circuits are active at all times, constantly watching for anything that goes wrong. As long as the CPU is working properly, nothing happens, but when this **fault detection circuitry** detects a hardware failure, a **machine check exception** occurs.

Some computers may have minimal or no fault detection circuitry, while others will have extensive circuitry. Software that is **mission-critical** must be robust. When a machine check is detected, the overall system must continue to function or at least do its best and fail as gracefully as possible. A machine check exception might be used to implement **graceful failure**, whereby a

failing machine goes down in a planned sequence designed to be safe and achieve the best (or "least-bad") outcome.

In many OSes, the kernel will respond to a machine check exception by (1) terminating the process that was running at the time of the failure, and (2) logging the failure so that a string of frequent failures can be detected. If the machine check occurs while the kernel is executing, the kernel may abandon every running process and simply reboot itself.

The remaining types of exceptions are loosely called **faults**.

Certain conditions can arise during some instructions that make completion of the instruction impossible. These include faults such as **Floating Point Exception (FPE)** or **Arithmetic Exception**. For example, dividing by zero typically causes an exception. Likewise, the floating point standard specifies which operations may raise an exception. For example, subtracting infinity from infinity is an error. In such cases, the instruction stream is interrupted and the process cannot continue. The process will be aborted. (Actually, the kernel will send a signal to the process; signals are discussed later in this document.)

If the CPU fetches an instruction and, when decoding the instruction, determines that it is not a valid, defined instruction that is implemented, then a fault occurs. For some "instructions" (i.e., for some invalid bit sequences that have been fetched), there is no such instruction and the process cannot continue. A **invalid instruction fault** occurs and the process is aborted.

In other cases, the bit pattern may represent a known instruction that implemented on some models of the processor but not other models. If the instruction is not implemented by the version of the CPU in question, the CPU cannot execute it. Instead, the instruction must be **emulated** (i.e., performed by software). In this case of an emulated instruction, the kernel exception handler will perform the instruction's execution using a software algorithm. Then the kernel will return to the process and normal instruction execution will resume with the following instruction.

Simple CPUs sometimes implement their floating point instructions with emulation. The floating point operations are quite complicated and some CPUs are unable to execute these instructions in hardware. So they **emulate the floating point instructions** using exception fault processing.

Another kind of fault is the **page fault**. Page faults are used in conjunction with **virtual memory**. The idea is that a user-level process is executing instructions. Normally those instructions (and their data) are located in physical main memory and so everything works. But sometimes, the instructions or data are not present in main memory, in which case a page fault occurs.

When the kernel handles a page fault, it first determines what happened. There are several possibilities:

(1) The required bytes are currently not in main memory but have been swapped out to disk.
(2) The process is trying to write to bytes that are marked "read-only".
(3) The process is trying to access bytes that are completely out of range and invalid.

In case (1), the process has done nothing wrong. At any one time, some of the pages of the logical (i.e., virtual) address space will be kept in main memory while other less-used pages of the address space are kept out on disk. When the page of memory that is needed is not one of the pages currently in main memory, a page fault is triggered. To deal with this page fault, the kernel will go out to disk, get the required page and move it into main memory, and then restart the process. The process will be restarted on the same instruction that caused the fault, and this instruction will be **re-tried**. Now, since the bytes are in main memory, the faulting instruction will work okay and there will be no page fault. This happens all the time (although we design the OS so that it doesn't happen too much, or the process would spend all its time waiting on the disk!)

In case (2), the process is trying to update a page in its virtual address space that was marked read-only. This is a problem. To guard against viruses and malware, the pages containing code are generally marked read-only. Any program that violates this is either buggy or up to no good. The kernel will terminate the process with a **segmentation fault** (i.e., a **seg-fault** or **SIGSEGV** signal).

In case (3), the process is trying to access a byte within its virtual address space that has not been allocated. The virtual address space is divided into a series of contiguous **pages**. Each page is 4 Kbytes. With a 4 Gbyte virtual address space, there will be 1,048,576 pages, so we can number the pages 0, 1, 2, 3, … 1,048,575. The code (the **.text** segment) goes into some pages. The **.data** segment goes into other pages. The stack and heap are in other pages. But most of the pages are simply unused and these pages are said to be **unallocated**. If the process tries to access a byte in an unallocated page, it indicates the program is buggy. Just as in the previous case, the kernel will terminate the process with a **segmentation fault** (i.e., a **seg-fault** or **SIGSEGV** signal).

The final kind of exception we want to mention is a **protection fault** (also called **protection violation** or **privileged instruction violation**).

At any moment, the CPU is either in **kernel-mode** or is in **user-mode**. Kernel code is always executed in kernel-mode. User-level code is always executed in user-mode, and this is really the definition of exactly what is "kernel code" and what is "user-level code".

In kernel-mode, all instructions may be executed and all security checking is turned off. In user-mode, some instructions are not allowed. Such instructions are said to be privileged instructions. A **privileged instruction** may only be executed in system-mode. If an attempt is made to execute a privileged instruction when the CPU is in user-mode, the CPU will detect this security breach and a protection fault will occur. The kernel will then get control and the offending user-level process will be terminated. This is how the kernel protects itself from mean-spirited user code and enforces a security barrier between itself and user code.

Kernel-mode is sometimes called "supervisor mode", which dates back to the days when early OSes were called "supervisor programs".

How are exceptions processed by the CPU?

The set of hardware exceptions is determined by the designers of the CPU. At the time the CPU is designed, they determine: (1) Which exceptions exist, (2) what exactly causes the exception, and (3) what the CPU does when an exception occurs. Each type of exception is given a number.

Some x86-64 exceptions:

0	divide by zero
6	invalid instruction
13	protection fault
14	page fault
15	floating point exception
18	machine check
32-255	syscall trap and other OS-defined exceptions.

An **exception table** (sometimes called the **interrupt vector**) is an array of pointers. This array has a fixed, predetermined layout and is kept in a specific location in physical main memory. The details of the table format and exactly where in memory it is located are determined by the CPU designers. The authors of the kernel code work with the details they are given. Often the exception table is located at the **bottom of memory**, i.e., at address zero.

The exception table contains one entry for each exception type. Each entry is the address of a kernel function (or at least should be, if the kernel is to work correctly). This is the address of an exception handler function. An **exception handler function** is part of the kernel. A handler will be invoked and executed whenever an exception occurs. There is one exception handler function for each type of exception.

An interesting fact is that once the kernel has finished booting and is up and running user-level processes, the only way the kernel can be entered or get control is by executing a handler as a result of a hardware exception.

When an exception occurs, the CPU will finish up the execution of the current instruction (or instructions, in the case of pipelined execution) that was being executed. Then the CPU will effectively insert a "call" to the exception handler function. The CPU hardware consults the exception table to find the address of the right handler function to invoke. Each exception has a different number (see the list above) and that number will determine which entry in the array of addresses is used. The CPU will also switch into system-mode.

The kernel handler function will then run and will take the appropriate action. When it completes, the normal thing is to return to the interrupted instruction sequence. There is a special instruction called **RETURN-FROM-INTERRUPT (RETI)** that is the last thing the kernel exception handler function does. The details are complex, but this is the general idea of what happens when an exception occurs.

Let's see what happens when some user code invokes a Unix/Linux system function such as:

open (filename, options)

This function does a little work, but then calls a wrapper function named **__open()** to make the system call. Here is the relevant part of __open():

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```

[1]

The code moves the number 2 into register %eax and then executes the syscall instruction. At that point, the kernel gets control. The CPU will switch to kernel-mode and the syscall exception handler function will be invoked. This function will look at %eax to determine which system call is desired. The handler will then do the work of preparing the file for I/O and return a result value in %rax.

Note that the term **system call** can mean a Unix/Linux system function such as open(), write(), fork(), etc. Also **system call** is the name of an instruction. There is only one instruction and its opcode name is **SYSCALL**.

In Unix/Linux, each system call has a number. Here are some of the code numbers used in one particular version of Linux. (These numbers are not dictated by POSIX; different versions of Unix/Linux may use different numbers.)

0	read
1	write
2	open
3	close
4	stat
…	
57	fork
59	execve
60	_exit
62	kill
…	

Upon completion of the syscall handler in the kernel, return will be made to the user-level code (using the RETI instruction) and the instruction following the SYSCALL instruction will be executed. This is a compare (CMP) instruction which will test the returned value.

[1] Some examples in this document come from the Bryant and O'Hallaron textbook.

After a page fault occurs and is handled, the return will be to the instruction that caused the exception. Note the difference: After a syscall trap, the return will be to the *following* instruction, but after the page fault, the return will be to the *same* instruction. In the case of a page fault, the instruction will be re-tried a second time.

This makes sense. The page fault may have occurred because the page of memory that was needed was not resident in physical memory. The kernel will not return from the handler until after the page has been read in from disk to memory. So the instruction should be re-tried and will now be able to execute without causing an exception.

It is possible that the page fault occurred because the program trying to do something it should not do. Perhaps the program was trying to access a byte in an unallocated page or perhaps the program was trying to modify a byte in a page that is marked read-only. In such cases, the exception fault handler will not return to the interrupted code. Instead, the kernel will terminate the process altogether. (In some cases, the kernel will send a signal to the process instead of terminating it, but either way, there is never a return to the instruction that caused the exception.)

A **process** is an instance of a program that is running. A program is a static thing. A program may be sitting quietly on disk. The program might not even be on any computer; perhaps the program is written on a napkin, but it is still a program.

Once the program is loaded into main memory and the CPU begins executing the instructions, you have a **process**. A process requires two things:

- An address space (i.e., memory)
- A CPU to execute instructions

In Unix/Linux the program is loaded into a **virtual address space** (sometimes called a **logical address space**). A virtual address space is similar to the physical main memory of a computer. It is a sequence of bytes, and each byte has an address. A virtual address space is different from physical memory in the following ways:

The virtual address space is divided up into pages of size 4Kbytes
Some pages may be marked read-only
Some pages in the address space may be unallocated
The pages of the address space may be stored in physical memory (RAM) or on disk

A process also needs a **thread of control**. This is provided by a CPU which is executing instructions. Since the computer may have only one CPU, the kernel is responsible for sharing that CPU between several processes. (If there are multiple cores, the kernel tries to share them all between the processes, which is a little trickier, but basically the same idea.)

A single program can be running simultaneously within several different processes. For example, in a multi-user system, two individual users might be executing the same program. (Perhaps they are both using the "vi" text editor at the same time.) There is only one "vi" program, but this program is running in two processes concurrently. Normally, these processes would be completely independent and neither would even know about the other.

When the kernel switches the CPU from executing one process to executing another process, it must save the entire state of the first process before the switch.

Switching from one process to the next is called a **context switch**. The kernel performs context switches very often. For example, there may be a context switch every millisecond. This implies that each process gets 1/1000 of a second of CPU execution time before the CPU is switched to another process. It also implies there are 1000 context switches per second.

When a context switch occurs, we can talk about the "previous process" and the "next process". The kernel must save the state of the previous process. The **state of a process** is all the information that is necessary to restart that process at some later time. Mostly, the state consists of the registers, so the kernel must save the values of the registers when the previous process is stopped.

Next, the kernel will load the state of the next process into the registers. Once this happens, the next process can run.

In addition to registers, the state of a process also includes information about the virtual address space. The kernel maintains information about where the pages of the virtual address space are actually stored. A page of virtual memory may be loaded into physical memory or it may be out on disk. The kernel keeps track of a bunch of virtual address spaces and, for each address space, it keeps track of where the pages are, which pages are allocated, which pages are read-only, and so on.

When a context switch occurs, this aspect of the state must also be changed. The information about the previous process's virtual address space must be stored and the information about the next process's virtual address space must be loaded. Then, when the next process runs, all instructions will be executed inside the correct virtual address space.

The kernel is running many processes at any one time. A typical Unix/Linux computer may have over 100 processes that are currently running. Of course only one process is actually executing instructions on the CPU (assuming a single-core processor). The other processes are waiting for their turns. (In the case of a processor with 4 cores, 4 processes would be running concurrently.)

At each context switch, the kernel must choose which process to run next. This part of the kernel is called the **scheduler** and there are a number of different **scheduling algorithms**. The simplest scheduling algorithm is called **round-robin**. With round-robin scheduling, the processes are

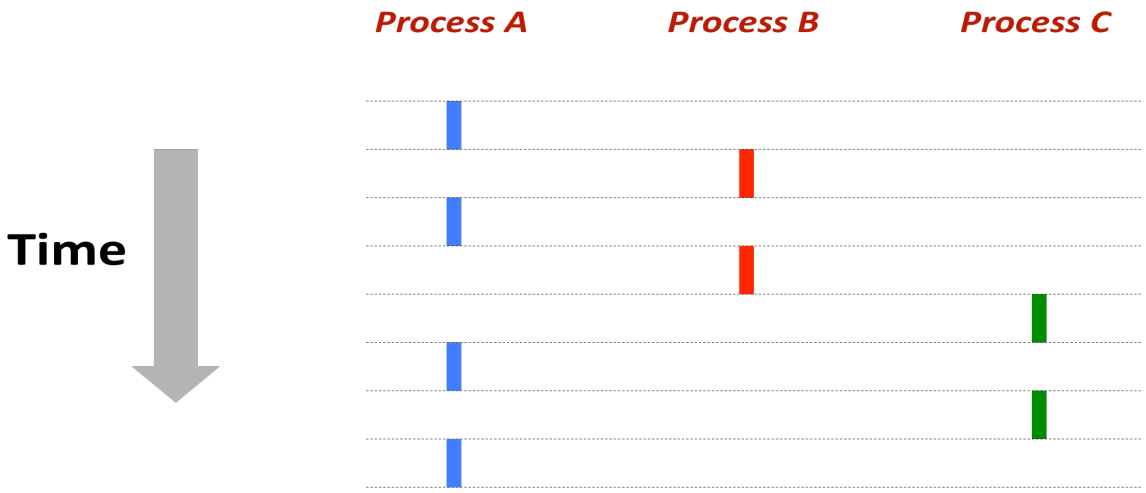
executed in order, one-after-the-other. There is really no scheduling decision at all; the kernel just maintains a queue. At a context switch, the previous process is placed at the tail of the queue and the process at the head of the queue is removed and scheduled next.

These fascinating topics are covered more fully in the Operating Systems class.

In a typical system, there are many processes running at any one time. One process is probably your **shell process**. If you are running a command (such as the "vi" editor or the "gcc" compiler or some other command), then there will be a process created just for that command. In addition, you'll see the original process , which is called **init**. This is a sort of "master process" which is created at boot time. (It's call **launchd** in the Mac OS). In addition, there will be a number of processes concerned with the window interface, the network interface, and the kernel itself. If you have other windows open (such as a web browser) there will be processes for that. The email system involves several processes. And so on!

When several processes are running, we say they are **concurrent**. Only one can be executing instructions at a time (at least on a single core computer), and the others are sitting in the **scheduling queue** waiting for their turns. The context switches occur very frequently (e.g., every millisecond), so to us humans, it seems as if they are all running at the same time, and all making continuous progress.

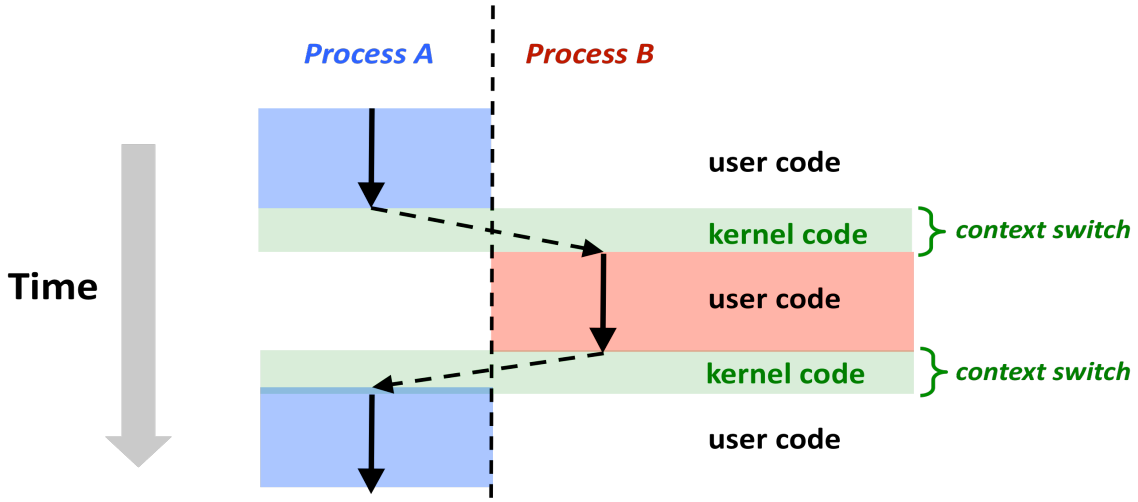
Here is a picture showing 3 processes. Each horizontal dashed line indicates a context switch.



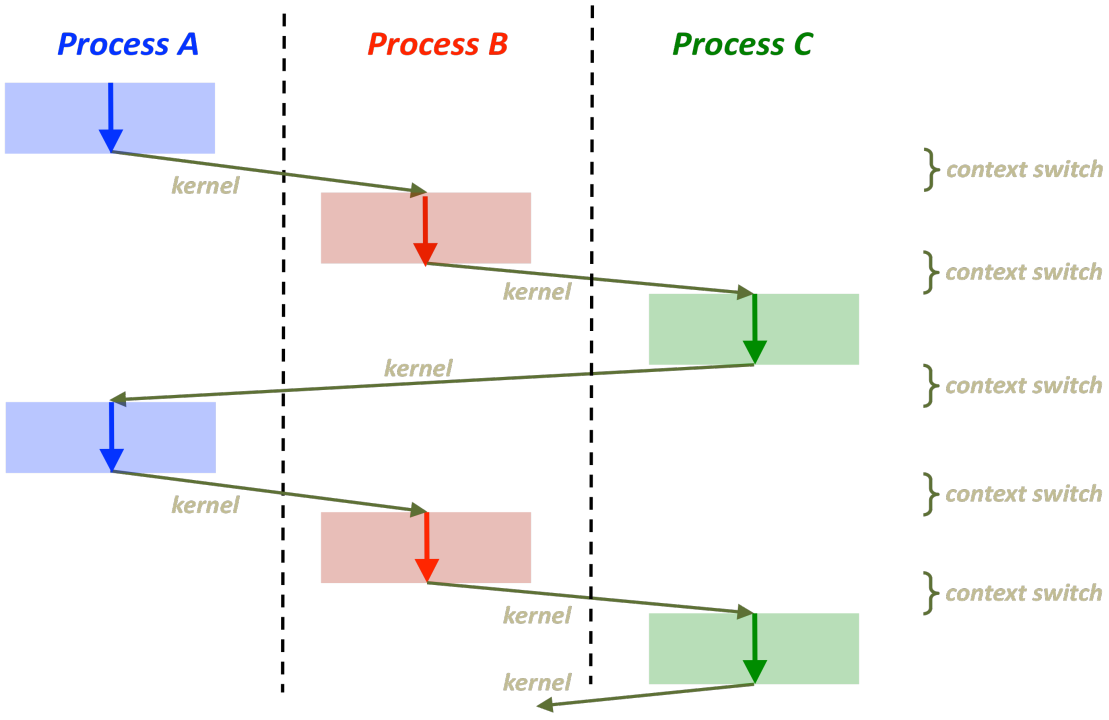
Looking more closely at the context switches in the next picture, we can see that the kernel gets involved each time.

Here's what happens: Process A is executing. Then a timer interrupt occurs. (For example, such an interrupt will occur every millisecond.) Then the kernel exception handler gets control. The kernel

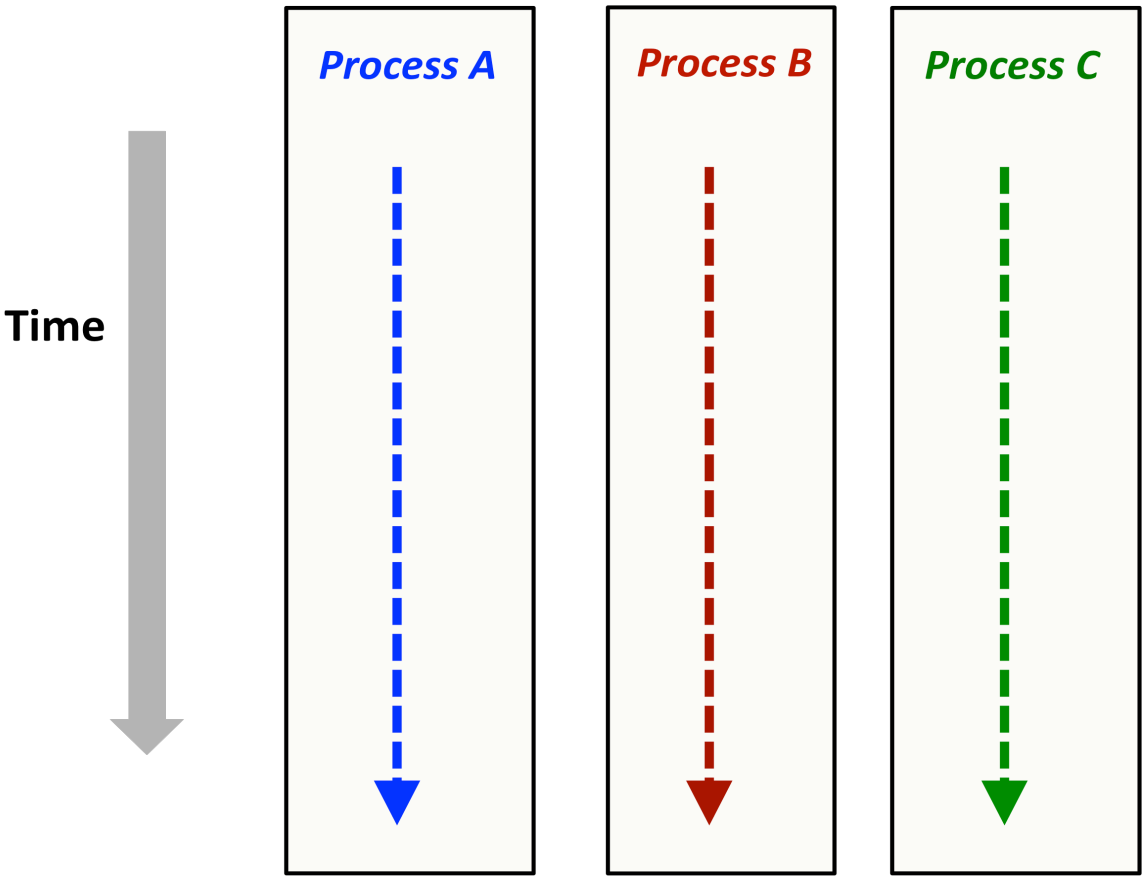
saves the state of process A (the "previous" process) and loads the state of process B (the "next" process). Then process B executes for a millisecond, until the next timer interrupt causes another context switch. Then the kernel gets involved again, and so on, a thousand times per second.



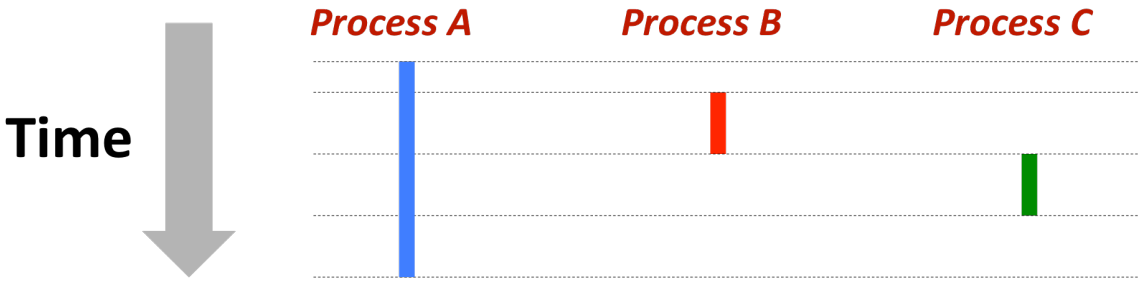
In the next picture we move out a little and see **round-robin scheduling** between 3 processes. (There would actually be many more processes in a typical system, but this shows the idea of round-robin scheduling.)



In the next picture, we move further out and look at an even bigger picture. Now we are unable to see the individual context switches, but we still have the idea that each process is a collection of time-slices. (Each segment of the dashed line represent one time slice.)



Finally, we move out even further. At this point, the time-slices are no longer visible and we can see the entire process execution, from process creation to process termination. In this view, it appears that all processes are executing with true concurrency, although you now know this is an illusion.



In this example we say that processes B and C are **sequential** since one of them (B) terminates before the other process (C) begins. We say that A and B are **concurrent** since their overall

execution overlaps (although we know that on a single core computer, they are time-sliced and never truly execute instructions at the exact same time).

A and B – concurrent execution
A and C – concurrent execution
B and C – sequential execution

If concurrent processes interact in any way, the programmer must be very careful to control their interaction. This is called **concurrency control**. The kernel's scheduler makes no guarantees about the relative speeds of execution or when exactly the context switches occur. If the two processes share data, then they must be careful about the order of reads and writes.

Consider what would happen if process A reads some data, computes a new modified value, and then writes the data back out. Now imagine that process B also does the same sequence: read-compute-write back. Imagine that they both read the same data value at more-or-less the same time. Then they both compute at more-or-less the same time. Assume that process A writes the data back out first, and then process B writes the data back out. Notice that the value that A wrote back out is subsequently overwritten by process B! So the results of process A's computation are lost! Not good!

Notice that this program may work correctly some times and incorrectly other times. This program has a **race condition** (sometimes called a **race bug**). The behavior of the program depends on the actual timing of the processes. This is unpredictable and can vary from run to run. Race bugs may be **intermittent** and **not reliably reproducible**. This makes them very, very difficult to find and correct! Normal debugging procedures (whereby we run the program to see if it works properly) are not adequate for concurrent programs that interact.

Unfortunately, we cannot go into the solutions to concurrency control problems here, but there are many interesting techniques.

Instead, we turn to the question of how processes are created and how programs can be loaded into virtual address spaces and run.

Whenever you invoke a system function (such as open(), read(), write(), fork(), etc.) you should always check the return value to see if an error has occurred. Some functions return "void"—which means nothing is returned—and you don't need to check these of course, but for others, **never ignore error checking**.

For example, the fork() system call should return a non-negative value. (We'll discuss this function a bit later.) If there is a problem, the function will return a negative number. Many system functions also set a global variable called **errno**. They will store into errno an integer value that indicates the specific error.

There are some functions such as **strerror()** and **perror()** that can be used to translate the integer code number in errno into a meaningful string. Here is some typical code which calls fork(). If there is an error, it prints the error message and terminates the process.

```
if ((pid = fork()) < 0) {
        fprintf (stderr, "fork error: %s\n", strerror(errno));
        exit(0);
}
```

Here is similar code that does the same thing. This way of doing it is very common.

```
if ((pid = fork()) < 0) {
        perror ("fork error");
        exit(0);
}
```

If we call a function from many places, it may be burdensome to always check for errors. So we can create a wrapper function to make our life easier, as shown next. Notice that we have given the wrapper function a name beginning with an uppercase "F", in contrast to the system call which begins with a lowercase "f".

```
pid_t Fork(void) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        perror ("fork error");
        exit(0);
    }
    return pid;
}
```

Now we have a way to safely call fork() without cluttering up our code with a bunch of error checking code. We can just write this:

```
pid = Fork();
```

Each process has an id, called the **process ID (PID)**. This is a small number assigned by the kernel when the process is created.

Processes are related in a **parent-child relationship**. Whenever a process is created, it becomes a child of the process that created it. Likewise, the creator is the parent of the new process.

Here are a couple of functions to retrieve the ID number of the current process and the ID of the process's parent. By **current process**, we simply mean the process that is calling the function, i.e., the currently executing process. So with **getpid()**, a process can learn its own PID.

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

These functions have no error conditions.

Every process is in one of the following states:

Running
Stopped
Terminated

A **running** process is either having instructions actively executed by the CPU or is waiting its turn. If a running process is not currently executing instructions, then it is waiting to be **scheduled**. There is nothing preventing the process from making progress except that the CPU is a limited resource that can only execute one process at a time and the CPU is busy executing another process at the moment.

A **stopped** process is said to be **blocked**. It cannot be executed for some reason, but it will go back to running later. A stopped process is waiting for some event. When the event occurs, the process will once again be running. For example, a process could be stopped and waiting on the completion of an I/O operation such as a file I/O. When a page fault occurs, the process must be stopped until the required page can be read in from the disk.

A **terminated** process has completed its execution. It will never run again. There are several ways to terminate a process. If a process invokes the **exit()** system call, it will be terminated. If a process returns from main() it will be terminated. Certain kinds of exceptions (like **segmentation fault** or **protection violation**) will terminate a process. A process can also be terminated by another process. If a process is ever sent the **SIGKILL** signal (which would be sent by another process), it will be terminated.

The **exit()** function can be called to terminate a process.

```
void exit(int status)
```

There is never a return from this function, and there are no error conditions associated with the function itself. The argument is a number and this will become the **exit status** of the process. Other processes may check a process's exit status to see whether the program "worked" or not.

Typically a program that encounters some error condition will terminate with a non-zero exit status. By convention, an exit status of zero means "no problems, all okay", while a non-zero exit status indicates that something went wrong in the program. The exit status should be in the range 0-255. Note that the exit status is non-negative and limited to an 8-bit number.

Often, a value of 1 is used to indicate that an error occurred and all further information about the error is printed to the **stderr** output.

When a program's **main()** function returns, it returns a value. The main() function is actually invoked from a special function that is used to get the process going in the first place. After main() returns, that function will call exit() using the value returned by main(). It doesn't matter how you terminate the process—either returning from main() or directly invoking exit()—they do the same thing.

Notice that exit() is somewhat unusual. You can call it, but it never returns!

The **fork() system call** is used to create a new process.

```
pid_t fork (void)
```

The symbol **pid_t** is a **typedef**. You can often spot typedefs because many end with "_t". The type pid_t is the type of process IDs. Since process IDs are just small integers, I feel the following prototype for fork() is clearer and easier to understand, even though there might be some minor differences between the two for some platforms or implementations of Unix/Linux.

```
int fork (void)
```

The fork() function is invoked by a process (the "parent") and it will create a new process (the "child").

The child will be a **clone** of the parent, in the sense that they are almost exactly identical. The child will have a newly created virtual address space, which will contain exactly the same bytes as the parent's address space. This new address space is a copy, so when either parent or child modifies its address space, the change only affects that address space. The other process will not "see" the change.

Files that are open in the parent will also be open in the child. Typically, stdout is attached to the terminal display. The child will inherit stdout, as well as stdin and stderr, from the parent. After the fork, if either process writes to stdout, the output will appear on the terminal display. If the parent had stdin open before the fork, it will also be open in the child after the fork. If the user types a key, then that character can go to either process; it just depends on which process executes a read() first.

One difference is that the child will be assigned a process ID that is different from the parent's PID.

During the invocation of the fork() system call, the child is created. From that instant onward, they behave the same way. In particular, both processes are executing the same instructions in identical address spaces. The parent (which called fork()) will return from the system call.

Likewise, the child will also return from the call to fork(). After all, they are clones so they behave the same way and therefore do the same thing.

Thus, for fork(), there is one call and two returns!

(For every process, there is a fork() when the process is created and an exit() when the process terminates, so in some sense, the "calls" and the "returns" do end up balancing and being equal in number.)

Another difference between parent and child concerns the value returned from the fork() system call. In the parent, the return value will be the process ID (PID) of the newly created child. In the child, the return value will be zero. Normally, the code immediately following the call to fork() will check the return value.

If the value is zero, then the process is the child process. To put it another way, if a process checks the return value from a call to fork() and finds that it is zero, it will know that it is the child process.

On the other hand, if the value is non-zero, the process must be the parent. So within the parent process, the fork() system call returns the process ID of the child that was just created.

Normally, programs check the return value from fork() and do something different, depending on the PID returned from fork(). In other words, the parent and the child processes can behave differently after the fork(). Once a process checks the return value from fork(), it knows whether it is the parent or the child. At that point their behaviors usually diverge wildly, with the parent and child doing very different things.

Here is an example using fork(). Note that we call our wrapper function Fork() to handle any errors that may arise.

```
pid_t pid = Fork();
if (pid == 0) {
   printf("Hello from child\n");
} else {
   printf("Hello from parent\n");
}
```

This program begins by calling fork(). Then the process (perhaps we should say "both processes") checks the return value to see whether it is the parent or the child. The parent prints one message and the child prints a different message.

What is the output produced by this program? It could be either this:

```
Hello from child
Hello from parent
```

or this:

```
Hello from parent
Hello from child
```

It only depends on which process happens to be scheduled first.

Here is another example. In this example, we see a variable "x". In the child, the variable is incremented before being printed. In the parent, the variable is decremented before it is printed.

Notice that before the call to fork(), there is only one copy of the address space. The variable "x" has only one value, namely 1. After the call, there are two processes with identical address spaces. So each process has its own private copy of variable "x", as well as everything else in the address space. From then on, different things happen in the two processes. The "x" in one address space is incremented to 2, while the "x" in the other address space is decremented to 0.

```
int main() {
    pid_t pid;
    int x = 1;
    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

The output could be this:

```
parent: x=0
child : x=2
```

or this:

```
child : x=2
parent: x=0
```

It just depends on which process is scheduled first.

Here is a program that calls fork() twice. After the first fork, there are two processes. Both proceed and call fork a second time. This results in a total of 4 processes.

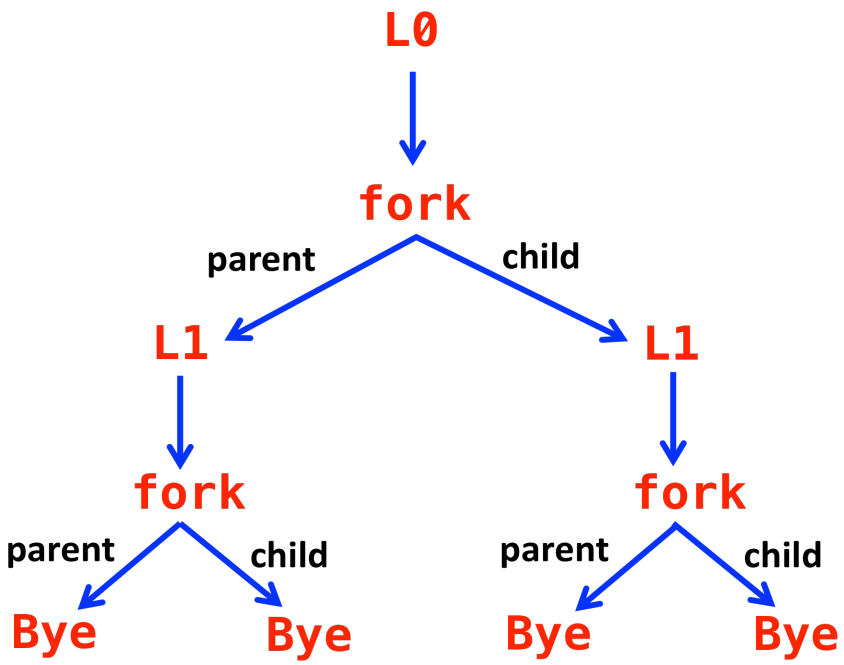
```
void example () {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Here is are all possible outputs:

```
L0      L0      L0
L1      L1      L1
Bye     Bye     L1
Bye     L1      Bye
L1      Bye     Bye
Bye     Bye     Bye
Bye     Bye     Bye
```

One way to understand this is to draw a **process graph**. The process graph for this program is shown below.

In a process graph each node is marked with an action, such as “print L1” or “fork”. The nodes are drawn from top to bottom in the order the actions occur. All actions are done in a linear sequence, one after the other, except “fork” and “wait”. A “fork” node will cause a splitting of the paths, while a “wait” node will be connect two paths.

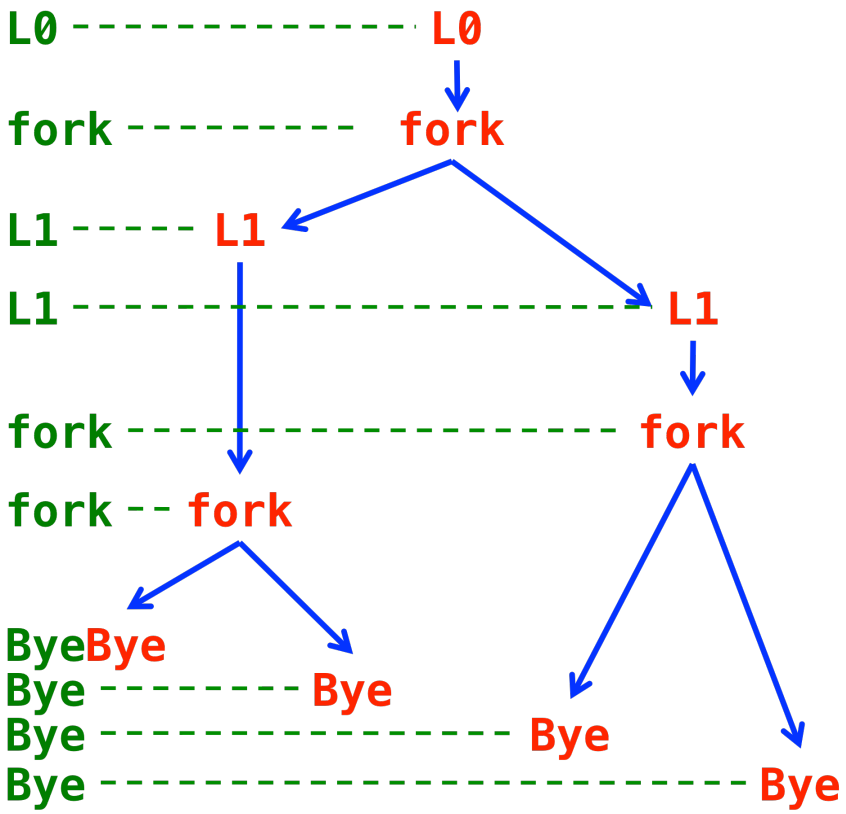


A **linear ordering** is a fully ordered sequence of nodes such that the nodes occur in the order given by the graph.

One such linear order is:

```
L0
fork
L1
L1
fork
fork
Bye
Bye
Bye
Bye
```

This makes sense if we display the process graph like this.



This linear order corresponds to one of the possible output sequences we listed earlier. The other possible outputs each correspond to different linear orders.

Here is another program. Draw the process graph and show all possible outputs.

```
void example () {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Here is another program. Try this one, too.

```
void example () {
    printf("L0\n");
    if (fork() == 0) {
```

```
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

When a process terminates, it provides an **exit status**. This exit status will be consumed by some other process which can then choose to act upon it in some way. Typically the exit status will be consumed by the process's parent. After a process terminates, the kernel will retain the exit status until its parent requests it.

The parent can retrieve a child's exit status by invoking the **wait()** system call:

```
int wait(int *child_status)
```

A parent can create multiple child processes. When the parent invokes wait(), the parent will be suspended until one of its children terminates. The return value from wait() tells which child terminated. (The return type is actually pid_t, but int seems clearer to me.)

The parameter child_status is a pointer to an integer variable. The kernel will store information about the child into this variable. This information includes the exit status, along with other information, which is why the exit status is limited to 0-255 (8 bits in size).

The child_status word can be checked and decoded by several macros, such as **WEXITSTATUS** and **WIFEXITED**.

Consider the following code. The program begins by calling fork(). After that, the child process prints "Hello from child" and then terminates. The parent prints "Hello from parent" and then waits for the child to terminate.

```
void example() {
    int child_status;
    if (Fork() == 0) {
        printf("Hello from child\n");
        exit(0);
    } else {
        printf("Hello from parent\n");
        wait(&child_status);
        printf("Child has terminated\n");
    }
    printf("Bye\n");
}
```

It is impossible to know whether the child will terminate before or after the parent invokes wait(), but it doesn't really matter. In any case, after the child terminates, the parent will continue and print "Bye".

Here are the possible outputs from this program:

```
Hello from child
Hello from parent
Child has terminated
Bye
```

and

```
Hello from parent
Hello from child
Child has terminated
Bye
```

There are no other possible outputs.

If the parent invokes wait() before the child calls exit(), then the kernel will suspend the parent process. The parent process will do nothing more for a while and will wait for the child process to terminate. Hence the name "wait".

If the child calls exit() before the parent calls wait(), then the child process will terminate and go into a post-termination state fondly called being a **zombie**. In a sense, the process will be suspended and be made to wait for the parent. A zombie is a terminated process that is waiting to be fully "dead-and-buried". Once the parent **consumes the exit status**, the termination of the zombie child process will be completed.

Below is another example.

In the first for-loop, fork() is called repeatedly. For example, if N is 5, then fork() will be called 5 times. Each time fork() is called, a child process will be created.

What happens immediately after fork() is called? First, the return value is saved in an array called pid[]. Then the return value is tested to see if it is zero (the child) or not (the parent).

In the child process, the PID will be zero. The process then executes the "then-statement", which causes the child process to terminate immediately. So the child process does not continue to iterate through the loop. The child exits without creating additional processes. The child adds 100 to i, so the 5 children exit with statuses of 100, 101, 102, 103, and 104.

In the parent process, the PID will be non-zero, so the parent will skip the "then-statement". Instead, it will continue iterating through the for-loop to create the rest of the children. After creating the 5 children, the parent has also saved their process IDs in the array pid[].

```
void example () {
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = Fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

In the second for-loop, the parent process will call wait() 5 times. Each of the children will terminate, and the parent will consume all 5 exit statuses. Whenever a child terminates, the wait() system call will return. Then the parent will print out the process ID of the child that terminated, along with the exit status from that child.

When a process terminates, it becomes a **zombie** until its exit status is consumed by another process. The kernel cannot delete all information associated with the process and it must keep information about the process in its tables until the exit status is requested by the parent process.

Once the exit status is consumed by another process, the kernel will finish the termination and delete all information about the process.

Therefore, a parent must always ask for the exit status, or else the zombie process will remain in existence and will take up kernel resources.

When a parent asks for the exit status of a child, we call this **reaping** the process. In the parent fails to reap the child, kernel resources are consumed. If this happens a lot of times, then the kernel can run out of memory and/or space in its internal tables. This can cause a kernel failure, so software must be careful to reap any children it creates.

What if a parent dies without waiting for its children? If a child process is a zombie (i.e., the child has already terminated), then the kernel reaps the child automatically and there is no problem. At the time any process terminates, the kernel will take care of any zombie children that are waiting. But what about a child process which is still active and running at the time its parent terminates?

If a parent terminates without reaping one of its children, and the child is still running, then that child becomes an **orphan** process. The kernel will automatically **re-parent** orphans. A child who has lost a parent will be given a new parent.

Orphan processes are re-parented to the **init** process. After the parent terminates, its children (if any are running) will have their parent process changed to the init process. A process can learn the identity of its parent with the **getppid()** system call.

The init process is always running and mostly waits for things. One of the events it waits for is the termination of any of its children. This includes "adopted orphan" children, who lost their parents and were re-parented to the init process. If one of these adopted child processes terminates, the init process will reap it by simply called wait() to retrieve the child's exit status. (Init ignores the exit status.)

So there is no problem if the parent terminates before its children.

Here is a program that creates a child but fails to reap it. The program creates a child and then each process prints a message which includes the processes' PIDs to make things easier to understand.

The child immediately terminates, but the parent goes into an infinite loop and never invokes the wait() system call. Therefore the child becomes a zombie and, since the parent never terminates, the zombie will remain in existence forever, or at least until we do something about it.

```
void example () {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

If we run this program and then do a **ps** command (to show **process status**), we will see the parent process (which is running) and the child process (which is now a zombie).

```
linux> example &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 example              ← parent process
 6640 ttyp9    00:00:00 example <defunct>    ← child process (zombie)
 6641 ttyp9    00:00:00 ps
```

Next, we use the **kill** command to terminate the parent. This causes the child process to be reaped by the kernel. We can use the **ps** command to verify that both parent and child are now gone.

```
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

In the next example, we have a program that creates a child. This time, it is the child which does not terminate. The child executes an infinite loop and never calls **wait()**. The parent immediately terminates without waiting for its child. Once the parent terminates, the kernel will re-parent the child to the init process.

```
void example () {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

When we run this program, we see what happens. The **ps** command shows us that the parent process is no longer running. We can also see the child process which is stuck in an infinite loop.

```
linux> example
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
```

```
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 example          ← child process
 6677 ttyp9    00:00:00 ps
```

Our solution is to kill the child process manually, as follows:

```
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

What about long running processes that create many children?

As we've seen, if the parent keeps running and does not terminate, then there could be a problem if the parent fails to reap the children it creates. This is actually an issue with **shell** programs and with **server** programs. Shells and servers tend to create a child process to handle each command or incoming request and then forget about these children.

We've seen that a parent can wait() to collect the exit status from its children. There is also a similar system function named **waitpid()** which can be used to wait on a specific process.

```
pid_t waitpid (pid_t pid, int *status, int options);
```

But there is a problem: These system calls will suspend any process that invokes them. As the name "wait" implies, the parent process will be forced to wait until the child terminates. This could be a very long time. But a program such as a shell or server needs to keep going and process more commands. It should never freeze up waiting for child processes.

So we need to allow a process to create a child and then go on about its business, doing other things. The child process will eventually terminate at some point in time, but since the parent process is busy with other activities, the child process becomes a zombie, taking up resources. Since the parent is still active, the process is not re-parented to init. So how can the process be ever reaped?

The solution is to use the **signal** mechanism in Unix/Linux. The short answer is: whenever a child process terminates, the **SIGCHLD signal** is sent to its parent. The parent is **interrupted** and it can then stop doing whatever it is doing and reap the child. The parent can then go back to doing whatever it was doing before it was interrupted.

Signals are discussed in depth later in this document.

Another important system call is **execve()**, which has this prototype:

```
int execve( char *filename,
            char *const argv[],
            char *const envp[] )
```

The filename argument points to a string of characters which is the name of an executable program. The filename is a pathname, which is used to locate a file in the file system. Here are some example pathnames:

```
./a.out
myprog
/bin/kill
```

The execve() system call will transfer control to the named program. There will be no return from the system call. The main() function in the new executable file will be invoked. And the kernel will do all of this within the current process.

More precisely, the current process will be changed and re-used to run a new program. To put it another way: a new process will be created but the process ID of the new process will be the same as the process ID of the old process. The new process will inherit some things from the old process, but some things will be changed.

The virtual address space of the old process will be emptied, re-initialized and loaded with the **.text** and **.data segments** from the executable file. All data from the old address space will be lost.

I suppose we could ask whether it is the *same* address space or a *new* address space, but this is a question of semantics and we'll leave it to philosophers. For convenience here, we'll use the terminology of "old" and "new" processes.

From the prototype, we can see that execve() has a return value (of type integer). The returned value is only relevant if problems occur. The primary problems that can arise are:

- file not found
- file is not an executable
- you don't have permission to execute this file

If there is a problem, a return from execve() is made in the old process, which can then print an error message or something.

If the call is successful, the new process will have the same process ID (PID). Furthermore, all files that were open in the old process at the time of the execve() will remain open in the new process.

It is normally the case that the old process will have **stdin**, **stdout**, and **stderr** open. These file descriptors are normally attached to a particular terminal display. After the execve(), the files are still open, which means that the new program can read and write to the same terminal display.

The shell uses the execve() system call to run commands. When the user types in a command, that command is nothing more that the name of an executable file. (Well, almost always. A few commands are built-in shell commands, which are executed immediately by the shell itself. But most commands are simply the names of executable programs.)

First, the shell will create a new child process. Basically, the only thing the child process does is perform an execve(). The result of invoking execve() is that command is run. Since it happens in a child process, each command will be run in its own separate process. This is a good thing, since the shell can continue running, even if the command does not work correctly and, for example, gets stuck in an infinite loop.

When we ran the ps command in the examples above, we saw that there were separate processes for both the shell (tcsh) and the ps command itself, in addition to the processes we were discussing in those examples.

When the shell creates a child, the same files (including stdin, stdout, and stderr) will remain open. In other words, every file that is open in the parent will be open in the child. Then, when the child process invokes execve(), those same files will remain open, even though the process is now running a new executable program. The effect of this is that when some shell command does I/O, that I/O will go to the same terminal display that the shell was using.

Whenever a program is first started, its main() function is passed some information. Here is the prototype for **main()**:

```
int main (int argc, char * argv[], char * envp [])
```

With main() it is possible to leave out some or all of the parameters.

Normally the compiler will complain about missing parameters, but the C language has a special syntax which allows a variable number of parameters. Another function that has a variable number of parameters is printf().

It is very typical for the programmer to leave out the envp parameter. The compiler will not complain if the programmer writes something like this:

```
int main (int argc, char * argv[]) {
   …
}
```

Sometimes the programmer will leave out all three parameters, but it's a good practice to check to make sure that argc==0 if no arguments are expected.

How are the argv and envp arguments passed from the old process to the new process by the kernel during the execve() system call?

The **envp** array has the same form a the **argv** array: It is an array of pointers and each pointer points to a null-terminated string of characters. Furthermore, the final entry in the array is a null pointer, so it is safe to run through the array looking at each string, while watching for the end by testing the string pointer against null. The **argc** argument to main() gives the number of non-null pointers in the argv array. The argc argument is redundant; it is not really needed and there is no analog for the envp array.

By convention, the form of the strings in envp is

KEY=value

Equivalently, we can think of them as:

ENVIRONEMENT_VARIABLE=string

The KEY is normally capitalized and there are no spaces around the "=" sign. The value can be any string of ASCII characters. For example:

```
MAIL=/var/mail/harry
```

The envp array is used to transfer **environment information** from the shell to a running program giving context information to the program, so the program can understand where it is being executed. This includes information such as

- Which machine it is running on
- Which OS it is running under
- What language the user speaks (English, French, …)
- What sort of terminal display it is running on

You can think of the environment information as being held in a number of **environment variables**. The KEY is the name of an environment variable and the value is its value.

The actual names and meanings of the environment variables is somewhat system dependent and somewhat a matter of history and convention. Furthermore, different shells (like **sh**, **csh**, **tcsh**, and **bash**) differ in details about the environment variables.

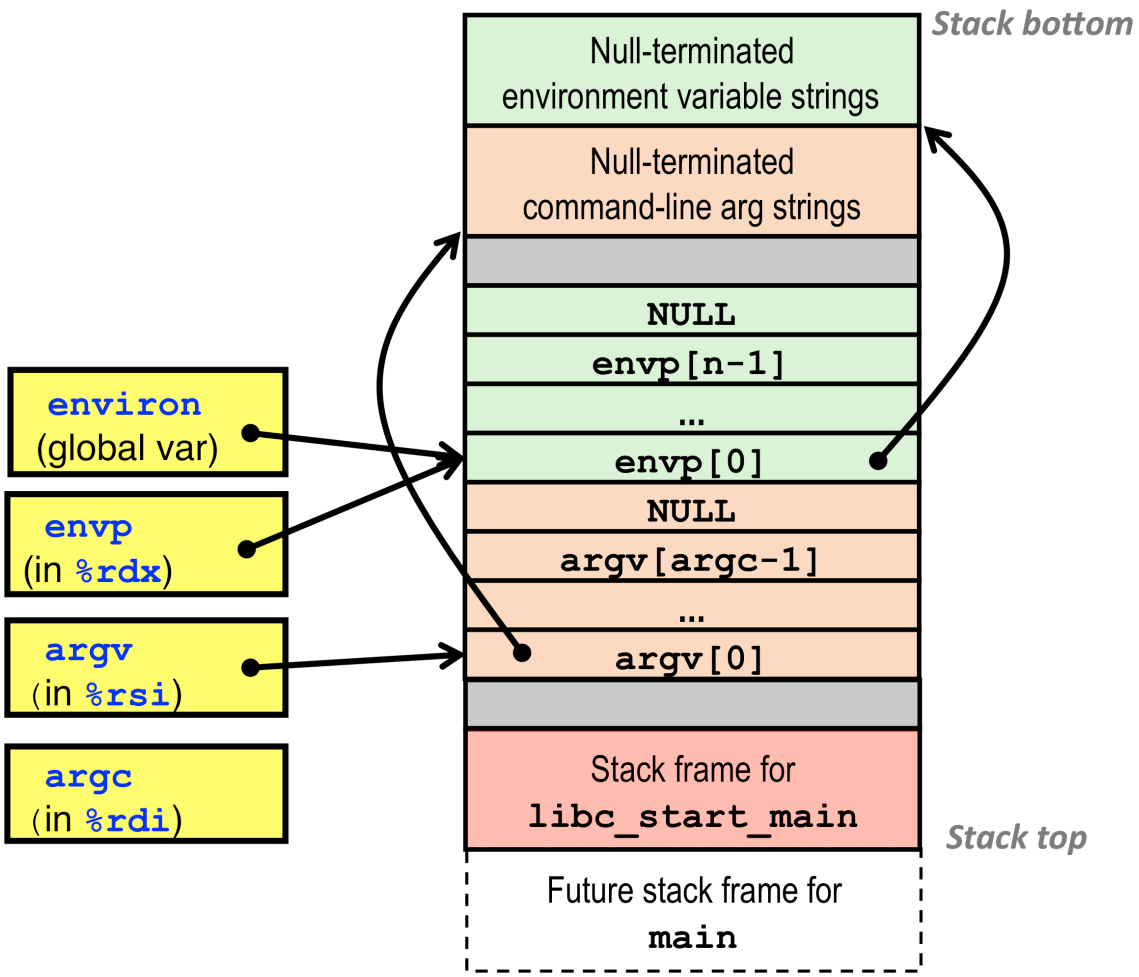
You can use the **printenv** shell commands to see the current environment. Here is an example environment from a shell I was running:

```
linux> printenv
LANG=en_US.UTF-8
USER=harry
LOGNAME=harry
HOME=/u/harry
PATH=.:/u/harry/Desktop/Blitz/BlitzTools:/u/harry/bin:/usr/local/sb
        in:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:
        /usr/local/games:/usr/local/bin
MAIL=/var/mail/harry
SHELL=/bin/csh
SSH_CLIENT=172.30.5.96 53480 22
SSH_CONNECTION=172.30.5.96 53480 131.252.208.85 22
SSH_TTY=/dev/pts/9TERM=vt102
XDG_SESSION_COOKIE=46af7143704ca67c395cc5955416dd69-
        1432846022.328694-390702012
XDG_SESSION_ID=129
XDG_RUNTIME_DIR=/run/user/5031
ICEAUTHORITY=/tmp/.harry_ICEauthority
HOSTTYPE=x86_64-linux
VENDOR=unknown
OSTYPE=linux
MACHTYPE=x86_64SHLVL=1PWD=/u/harry
GROUP=faculty
HOST=ruby
REMOTEHOST=172.30.5.96
UPKGCFG=/u/harry/.pkglist
```

The environment variables can be set using shell commands, such as **setenv**. (These commands are built-in shell commands. If the environment variable were changed within a child process, it would only affect that process, but we need the changes to affect the shell itself.)

When a process invokes the execve() system call, the kernel will initialize the virtual address space of the new process. Primarily, the kernel will create the stack and initialize the "bottom" of the stack. Then the kernel will invoke a special function, **libc_start_main()**. This function will then call main(). If main() does not invoke exit() but returns to libc_start_main() instead, then libc_start_main() will invoke exit() itself. So either way, the exit() system call will be invoked.

Here is how the stack is initialized and what it looks like right before main() is invoked.



The actual startup process is a more complex than this, but this is the general idea. In reality, there is additional code that gets executed before main() is invoked. The code for libc_start_main() and anything else that runs will be included in the executable by the linker.

There is one more additional thing about the execve() system call that is interesting and very useful.

The execve() system call is provided with a filename and this file is normally the executable program which is loaded and run. The executable file should be in the proper format and, for most Unix/Linux systems, this means it must be a legal **ELF file**. If not, it is an error and execve() will return to the old process.

However, if the first 2 bytes of the executable file are the ASCII characters

```
#!
```

then the kernel will look at the first line of the file and get a filename there, immediately following these characters. This is taken to be the name of a **script interpreter** and this is the mechanism used to run **script programs**.

Example scripting languages that are executed by interpreters this way are

Perl
Python
Ruby
Shell scripting languages (e.g., sh, csh, tcsh, bash, …)

A **script** is a program that is written in human-readable format. Script programs are not compiled. Instead, the source code is executed directly as-is; in a sense the source code file *is* the executable. To execute a script program, an interpreter must be used. The interpreter is a normal compiled executable program (probably written in C). The interpreter will read the script file and the interpreter will execute it according to the rules of the language.

Consider what happens when execve() is invoked and the kernel tries to run an executable file that contains the following characters in the first line.

```
#!/usr/bin/python
```

The kernel will go back to the file system, look for, fetch, and execute the program

```
/usr/bin/python
```

From the kernel's point of view, this is just another executable program. But we know it's the Python interpreter. This "program" will then read the rest of the script file, ignoring the first line. The rest of the file is (presumably) a legal Python program which the interpreter will execute.

The Python language and the details of how an interpreter works is another story, for another day. Other scripting languages work the same way.

Even the **shell program** itself can be used to execute programs. Normally, we type in one command after another to the shell program and this input comes from **stdin**. But what if the input comes from a file instead? The shell will do the same thing.

This is useful. You can type a bunch of shell commands into a text file (for example, the file might be called "cmds". Then you can type the following command, which will execute all the commands in the file, one-after-the-other by the **Bourne shell** (which is named **sh**). This is useful for automating tasks.

```
sh < cmds
```

But we can also modify the file to add this line before all the others:

```
#!/bin/sh
```

Then we can simply type:

```
cmds
```

to run all the commands in the file. (You'll also need to change the file to set its permissions to "executable".) It turns out that the common shells (sh, csh, tcsh, bash, …) are full-blown interpreters. Shells can execute loops and if-statements, making them scripted (i.e., interpreted) programming languages. The details of shell programming are beyond the scope of this document.

Let's return to processes and discuss how the shell and kernel work together to create and run processes.

A computer runs many processes at once.

A **process** is a program in execution. To execute a program you need (1) memory and (2) a CPU.

At any one moment, the state of a process consists of the contents of memory and the contents of the CPU registers (including the **PC,** the **program counter**).

To suspend the execution of a process, the **state** (memory and registers) must be saved. To resume execution, the saved state must be reloaded into the CPU. Only then can the process continue executing.

Processes are **suspended** regularly. With multiple processes, the CPU is **switched** from one process (which is suspended) to another process. When a process requests something that may take a while (such as disk I/O), the process will be suspended.

The CPU is switched from one process to another very often, for example, every millisecond. This creates the illusion that all processes are making progress and that all processes are executing simultaneously, i.e., concurrently. This is called **time-sharing** and also **multitasking**.

With 100 processes and only one CPU, each process will run slower than with only 1 process on the CPU.

If the CPU switches from one process to another every millisecond and there are 100 processes, then each process will get to execute about 10 times every second.

Each chunk of time (such as a millisecond of execution) is called a **time-slice**.

One millisecond is 1/1000 of a second. If a process gets 10 time slices every second, then it will get 10/1000 of a second of CPU time every second. It makes sense that if there are 100 processes, each will get 1/100 of the total CPU execution time.

In the old days, a processor chip contained a single CPU. Now a processor chip contains several cores. A **core** is a CPU. In addition, the processor may contain cache, main memory, and I/O devices.

When the computer chip has several cores, time-slicing gets more complicated. With 4 cores on the chip and (say) 100 processes, each process will now get (on average) 4 times as much execution time per second compared to a single core chip. So each process will get 4/100 of a CPU. [Think of it like this: We could put 25 processes on each core, giving each process 1/25 = 4/100 of the total CPU execution time available on a single core.]

There are several system calls in Unix/Linux to control processes.

fork()	used to create a new process
exit()	used to terminate a process
wait(), **waitpid()**	used to coordinate with and to synchronize with processes
execve()	used to run a new program in an existing process

Programs that call fork() in an infinite loop will try to create an infinite number of processes. Such a program is called a **fork bomb** and is a bad thing.

The **initial process** created when Unix/Linux boots is called **init**. The process ID of init is 1.

Each process has a **process ID**, which is a simple integer assigned by the kernel when the process is created. In addition, each process belongs to a **process group**.

Processes are related in a **parent-child** relationship. When a process is created, the new process is a child of the process that created it. Likewise, the creator of the process is the parent.

If the parent process terminates, the child is assigned to a new parent. Every process has a parent, except the init process.

One special kind of process is a **shell process**. This is a process that is executing a shell program (such as **bash**). Normally processes are attached to stdin, stdout, and stderr so they can interact with a human through a terminal interface of some sort.

Some processes are not descendants of shell programs. Such programs are called **daemons**. They are not attached to terminal interfaces so they run quietly in the background. An example would

be a web server process (such as **httpd**), which is a daemon. It listens for page requests coming in from the network and sends out pages in response.

A **shell** is an application program, which means it is just another user-level process running in user-mode. To the kernel, a shell is just another process and is not treated differently from other user-level processes.

The shell allows the user to type in commands and then it executes these commands. Some "built-in" commands are executed immediately but for most commands, the shell creates a new process and runs the command (which is itself a program) in that process.

Whenever the shell reads a line of input, it examines to find the name of a program. The program name and the command name are the same. In other words, each command names a program and when the command is typed, the corresponding program is executed to "do" the command.

To run a command, the shell creates a child process and executes the command program in the child process. Meanwhile, the parent process (the shell) waits for its child to terminate. When the command completes and terminates, the parent wakes back up and (in a continuous loop) reads in the next command line.

Here is an approximation to the shell program.

```
int main () {
    char cmdline[MAXLINE]; /* command line */

    while (1) {
        /* Print a prompt and read a command line */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate the command */
        eval(cmdline);
    }
}

void eval (char *cmdline) {
    char *argv[MAXARGS]; /* Argument list from command line */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */
```

```
    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        pid = Fork();
        if (pid == 0) {
            /* Child: Runs user program */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* Parent: Wait for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else {   /* or print PID of background job */
            printf("%d %s", pid, cmdline);
        }
    }
    return;
}
```

Previously, we've seen the wait() system call, which waits for any of its child processes to terminate. This code uses the **waitpid()** system call which is similar but waits for a specific child to terminate.

When the shell runs a program, it is either as a **foreground** job or as a **background** job. The term **job** is historical and means "a process". A foreground job causes the shell to be suspended. The shell will wait until the foreground job is complete before it prints out the next prompt and reads the next command line.

The following shows the shell syntax to create a foreground job.

```
linux> sleep 7200; rm /tmp/junk
```

The sleep command takes the number of seconds. The semicolon executes one command and, after it completes, executes the second command. This command line will cause the shell to wait for 7200 seconds (2 hours) plus the millisecond it takes to remove the file.

It is better to make this a background job. Use the following syntax (with **&** at the end of the command line) to create a background job. The shell will not wait and will print the next prompt almost immediately.

```
linux> (sleep 7200 ; rm /tmp/junk) &
[1] 907
linux>
```

The shell program above has a problem. For a foreground job, the shell waits for the child to terminate. But for a background job, the shell will continue without waiting. Thus, a job run in as a background process will never get reaped.

Processes that complete must be **reaped**. This means that another process (normally the parent) will wait for the exit status of the child. This is done with the call to **waitpid().**

When a process terminates execution, the OS will clean up after it. The OS can reuse the memory, close open files, etc. However, the OS must remember about the process. In particular, the OS must keep the **exit status** around. Why? Because some other process (normally the parent) may need to see the exit status.

When a process terminates, it becomes a "**zombie**". The process is done executing instructions and is pretty much dead-and-gone, but the kernel cannot get rid of the job entirely. Once the parent process asks for the exit status (with a call to **wait** or **waitpid**), then kernel can then get rid of the job. At this point, we say the zombie is **reaped**. After that, nothing is left of the process.

But how can the shell program reap background jobs? We don't want the shell to wait for the background job; we want the shell to go on with life, prompting and executing other commands.

But if the shell fails to reap child processes, then the kernel can accumulate a bunch of unreaped zombie processes. This could result in failure to release kernel resources and result in kernel failure.

If there are too many processes (including zombies), then a user may exceed his/her quota. You can see what your process limit is with commands like:

```
linux> limit maxproc      # for csh
maxproc      31818
linux> ulimit -u          # for bash
31818
```

A child process can terminate at any time. How can a process be notified about an event that could occur asynchronously? **Asynchronous** means "at any time". Answer: Use a signal.

A **signal** is a sort of message that is **sent** from one process to another process. However, it is a message with no content. The only information is that the signal happened. Signals are used to notify a process that some event has occurred.

Signals are a bit like exceptions and interrupts. A hardware interrupt (such as the disk controller interrupting the CPU to tell it that an I/O operation has completed) is implemented in hardware. A signal is a software analog of a hardware interrupt. Signal transmission is implemented within the kernel. The kernel will notify the process that a signal has been sent to it.

There is a small number of different types of signals. There are only about 30 different signal types and each signal type is given a number.

Each signal is also given a name. The signal name is simply shorthand for an integer in the range 1 to 30.

Here are some of the common signals:

2	**SIGINT**	Interrupt (e.g., control-c from keyboard)
8	**SIGFPE**	Floating point exception
9	**SIGKILL**	Kill program
11	**SIGSEGV**	Segmentation violation
14	**SIGALRM**	Timer signal
17	**SIGCHLD**	Child stopped or terminated
18	**SIGTSTP**	Suspend process
19	**SIGCONT**	Resume a stopped process

A signal is sent to a destination process. Normally another process will send the signal by invoking a system call to send the signal. However, some signals can be sent for other reasons, such as something the process itself did (such as floating-point-exception, seg-fault, etc.)

It is up to the kernel to take it from there and deliver the signal to the process.

The system call to send a signal is:

```
kill (pid, signal)
```

There will be some time between the moment the signal is sent and the signal is acted upon. During this interval, the signal is **pending**. Once the signal is **received**, it is no longer pending.

When a signal is received, one of these actions will occur:

(1) The signal is **ignored**.
(2) The process is **terminated**.
(3) A **signal handler** is executed. In this case, we say the signal is **caught**.

A **signal handler** is a function that exists within the process receiving the signal. The signal handler is run as a normal user-level function. It is called and, when complete, it returns.

A signal handler is similar to an exception handler in the kernel. The similarities are: Each is a function; each is executed in response to some outside event. The differences are: A signal handler is invoked in response to a signal; an exception handler is invoked in response to a hardware exception; a signal handler is invoked by the kernel; an exception handler is invoked by the CPU hardware.

From the point of view of user-level program code, here is what happens: instructions are being executed one-after-the-other. Suddenly after the completion of one instruction, the signal handler is called (i.e., invoked). The handler executes to completion. When the signal handler function executes a return instruction, control returns to the instruction stream that was interrupted.

A signal that has been sent but not received is pending. There can be at most one pending signal of each type. If a second signal is sent before the signal is received, then nothing happens; it is dropped and ignored, as if only one signal was ever sent.

In other words, signals are not queued.

A process may block signals. If a signal is sent, but that signal type is blocked, then the signal remains pending. If and when the process ever unblocks that signal, the signal will be received. If the same type of signal is sent several times while that signal type is blocked, then all but the first are ignored, since signals are not queued or counted. When unblocked, the signal will only be acted upon one time, even though it was sent multiple times.

The kernel implements signals by maintaining two **bit vectors** for each process. Each bit vector is a 32-bit word and each bit position in the word corresponds to one type of signal. For example, bit 2 is for SIGINT and bit 9 is for SIGKILL. This is why the number of signal types is small: there are only 32 bits available in each bit vector.

One bit vector is called “PENDING”. If a bit is 1 it means a signal for that type has been sent but not received. The other bit vector is called “BLOCKED”. If the bit is 1 it means that the process has blocked that type of signal.

When a signal is sent the kernel sets a bit in PENDING to 1. When the signal is received, the kernel clears the bit to 0 .

A process can block and unblock signals by invoking the system call **sigprocmask**. This system call just sets or clears bits in BLOCKED.

Every process has a **process ID (called a PID)** which is a unique integer that identifies the process. The PID is assigned by the kernel when the process is created.

Each process belongs to a **process group**. The process group is identified by an integer called the **process group ID (PGID)**. Process group IDs are just the IDs of some process, which we can think of as the head process of the group. Typically, a parent process will be the leader of a process group containing its child processes.

A process may change its own PGID and the PGID of its children.

When the shell creates a new process, it puts that process in a new process group. The PID of the newly created process will become the PGID for this new process group.

When a new process is created with **fork()**, it will belong to the same process group as its parent.

The shell can be used to create a **foreground job**. You can use the shell to create some **background jobs** as well.

A job is normally a single process running a command. But some processes will fork children. All the processes in a job (there may be more than one) will belong to the same process group. Thus a shell foreground or background job is actually a process group. Often a shell job will consist of a single process, but some processes create children so there can be several processes in the job.

When a signal is sent to a process, it is also sent to all the processes in that group. This is handy. When you type **control-c** to send **SIGINT** to a program, the signal will be sent to that process and all the other processes in that group. This allows you to interrupt a process and all the child processes it has created with a single control-c. Typing control-c will affect an entire shell job, even if the job contains several processes.

The system call **getpgrp()** is used to obtain the PGID of the current process. The system call **setpgid()** is used to change the group of some process.

The shell command **kill** is used to send a signal to a process or a process group. The following command will send SIGINT (control-c) to the process with PID 12345.

```
kill -2 12345
```

You might need to type:

```
/bin/kill -2 12345
```

If a process is not responding to SIGINT (control-c) it could be a problem. Perhaps the program has decided to install a signal handler for SIGINT and the handler has a bug. How can you terminate this program? You must send it SIGKILL.

```
kill -9 12345
```

SIGKILL cannot be caught, so this signal will always terminate the process.

This is not a very good way to terminate processes, but is occasionally necessary when debugging programs that catch SIGINT. I do not recommend terminating processes that you do not understand. Many processes communicate with each other and if one of them suddenly disappears, it may confuse the other processes and result in unpredictable behavior for your computer.

By placing a hyphen in front of the PID, it will cause the kill command to send the signal to a process group, rather than a single process.

```
kill -9 -12345
```

Typing **control-c** will send **SIGINT** to all processes in the foreground process group. The default action is to terminate the process(es).

Typing **control-z** will send **SIGTSTP** to all processes in the foreground process group. The default action is to stop (i.e., suspend) the process(es). This is useful if you wish to move the job to the background so you can keep using the shell to do other things.

The shell command **bg** will put a suspended job in the **background**. The job will resume execution, but the shell will now be free to prompt for commands. Anything you type will go to the shell and not the background job. The shell command **fg** will move a job to the **foreground**. After this command, all typed input will go to the foreground process group and not to the shell. When the foreground job terminates, the shell will once again prompt for the next command line.

You can use the **jobs** command to see what jobs the shell is managing. If there are several suspended jobs, you can specify which job you mean by providing a number to the **bg** or **fg** command.

The **ps** (process status) command will print out which processes are running.

`ps` ← shell and shell jobs; simple display
`ps axl` ← everything on the computer, lots of info

With ps, you can see the status of various processes:

S = Sleeping
T = Stopped
R = Running
+ = Foreground

You can also see process IDs as well as other info.

The **top** command will print out the active processes, in order of which are most active. This is an interactive display and will change continuously. Hit control-c to terminate this command.

Below is some example code.

The first for-loop creates a bunch of children. Let's say N=5 so it creates 5 children. It stores the PID of each child in an array called pid[]. Each child is in an infinite loop.

The second for-loop sends the SIGINT signal to each child process. It goes through the pid[] array and invokes the system call kill() once for each of the 5 processes.

The final for-loop waits for each child process. It calls the wait() system call N=5 times.

The wait() system call is used like this:

```
pid_t pid;
int child_status;
pid = wait (&child_status);
```

The type pid_t is essentially an int, so **wait()** returns the process ID (an integer) of the process that has terminated. There are 5 children, so it could be any one of them. The children will not necessarily terminate in any particular order. Due to the unpredictable scheduling of the kernel, they may terminate in a different order than they were created.

The **wait()** function is also passed the address of another integer. It will store the exit status of the process in this integer. Whenever a process terminates, it will provide an exit status. The convention is that zero means "all okay; no errors" and anything else means "error". This value is provided in calls to **exit()** or as the return value from **main()**. It is restricted to 0-255.

This code uses two macros to interrogate the exit_status value. In particular, the integer will contain more info than just the exit status from the process. So the **WEXITSTATUS()** macro is used to extract the exit status.

```
void example ()
```

```
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1) ;
        }

    /* Parent: Terminate all child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent: Reap terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

The kernel switches from process to process in intervals. Assuming there is only one CPU (i.e., one core), then only one process can execute at a time. The others are said to be **ready** (also called **waiting**); they are not **blocked**. Therefore a running process is either actually executing on the CPU or is waiting for its turn.

The kernel will execute a process for one **time-slice**. Then the kernel will perform a **context switch** to another process. When a process is executing on the CPU, the kernel itself is waiting. The kernel can get control several ways:

(1) A timer interrupt occurs. This event causes a hardware asynchronous interrupt, which is a type of exception. The kernel exception handler will get control and, as a result, the kernel can do a context switch.

(2) The process makes a system call. This event is also a type of exception and the kernel will get control.

(3) Some other I/O device causes an asynchronous hardware interrupt. Perhaps the disk controller is raising the exception to indicate that a disk read or write has completed. Perhaps the user has hit a key and the keyboard/mouse interface is raising the exception. Or the network interface has received a data packet.

Regardless of how the kernel gets control, the kernel will take care of whatever caused the exception. Then it will choose which process to execute next. It may return to the same interrupted process or it may choose a different process. This is the problem of process scheduling.

Before the kernel begins executing a new time slice for a process, it will check to see if there are any signals that are pending for that process. If so, the kernel will try to deliver them.

Recall the PENDING and the BLOCKED bit vectors that exist for each process… The kernel computes

PENDING & ~BLOCKED

A bit vector is one good way to represent a set, as long as the "universe" of possible set elements is small. In our case, there are only 30 different signals. The set called PENDING is represented by a bit vector. A "1" means that element is in the set. A "0" means that element is not in the set. The bit vector PENDING is the set of signals that have been sent but not received.

The bit vector BLOCKED represents the set of signals that are currently blocked by the process. These are the signals that the process just doesn't want to hear about.

The computation

PENDING & ~BLOCKED

uses bit vector operations **&** for logical **AND**, as well as **~** for logical **NEGATION** (or "bitwise complement"). Bitwise Boolean operations can be viewed as set operations:

&	performs set intersection
|	performs set union
~	performs set complement

So the kernel computes the set of signals that are "pending and not blocked". If there are any such signals, then the kernel will act on them, one by one. Typically this means that the interrupt handler is invoked.

So, if there is a signal that has been sent (and is therefore pending) which has not been blocked by the process, the kernel will execute the corresponding interrupt handler before the kernel returns control to where the process last left off execution. Effectively, the kernel "inserts" a call right into the code of the process.

When delivering a signal, the kernel will also mark that signal as no longer pending, which it does by clearing the corresponding bit in PENDING to 0. After the handler returns, the kernel will check to see if there are any other PENDING but not BLOCKED signals. If so, it will execute their handlers.

Finally, when no more signals need to be handled, the kernel will return to the process where it left off.

Each signal type has a default action. The possibilities are:

TERMINATE the process
TERMINATE the process (and "dump core")
IGNORE the signal
SUSPEND the process

A "core dump" means that the logical address space is copied to a file. This is useful for debugging. If something goes wrong with a process, the "core dump" can be examined (with a tool like gdb) to assist in the debugging to try to figure out what happened.

(Early computers used **iron ferrite cores** to store each bit in memory. Back then, memories were small and a "core dump" meant printing the entire contents of memory on paper. This was in the days before tools like gdb existed and debugging often meant analyzing core dumps by hand.)

A process can install a signal handler with the system call **signal()**.

```
handler_t *signal (int signum, handler_t *handler)
```

The behavior for the signal **signum** is changed from whatever the default or previous action was. This system call is used to **install** a signal handler with the kernel. Later, when that signal is sent, the kernel will know which handler to invoke.

The handler argument is a function pointer: it points to the function that is to be used when that signal is received. This argument can also be **SIG_IGN**, which means "ignore the signal" or **SIG_DFL**, which means "revert to the default behavior."

The following code shows an example using this system call.

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
```

```
    exit(0);
}

void example () {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            while(1); /* Child: infinite loop */
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

The code begins with a signal handler function called int_handler() which will print a message and then terminate the process.

The first thing this program does is to install this signal handler using **signal()**. Whenever the process receives SIGINT (control-c) it will catch the signal, print the message, and then call exit().

The first for-loop creates a bunch of child processes, as before. Each child process simply executes an infinite loop. Note that each child is an identical copy of the parent. This means that each child will have a handler installed for SIGINT.

The parent continues execution after creating its children. The parent sends a SIGINT signal to each of its children in the second for-loop.

Finally, the parent waits N times, and collects the exit status for each of its children.

This is good behavior of the parent. It is **reaping** its children, so they are not left as zombies, although the parent itself terminates quickly anyway, so they would not remain zombies for long. When a parent terminates, its child processes are re-parented to the **init process**. The init process is continually reaping, so these zombie children would get reaped almost immediately after the parent terminates, if the parent had failed to do it.

Here is output from this program. Note that the signaling is done in order, but the handlers are not executed in the same order. Finally, the reaping occurs in a slightly different order from the handlers.

```
linux> ./example
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

Each child in the above example has its own virtual address space. Therefore, each has its own copy of the code (the .text segment), its data (the .data segment) and its own heap and stack. Each process has its own (copy of the) handler and each handler is executed in a separate address space.

But every signal handler lives within the address space of some process. It is not a separate process.

Signal handling in Unix/Linux is tricky and there are many opportunities for confusion and bugs.

The different Unix/Linux operating systems have slightly different semantics and this makes code portability tricky when signals are involved. Be careful.

What happens if a process has invoked a system call and is waiting on its return when a signal is received? The handler may be executed, but what happens to the system call? It may return with an error code (see the **EINTER** value of errno) or the system call may just get restarted and return normally. Different OSes do different things. Ugh!

If you must write a signal handler, here are some guidelines.

(1) Keep the handler simple. For example, set a global variable and return.

(2) Be careful when you invoke system calls within a handler. Some system calls are "**async-signal-safe**" and others are not. Only invoke system calls that are async-signal-safe. Unfortunately, common system call like these are not safe:

printf(), malloc(), exit()

(3) Do not modify the global variable **errno**. If you make system calls within a handler, **errno** will be changed. So save **errno** first and restore **errno** later.

(4) Be careful with shared data structures. If a signal handler might run and might access some data structure, be aware that this could happen at any time! If this could happen, you need to block signals whenever your other code accesses the shared data, to make sure a signal handler doesn't "sneak in" and modify the data while your code is using it.

(5) Declare shared variables using the C keyword **volatile**. Compilers try to put variables in registers and leave them there as long as possible. But if the variable is in a register, the other code cannot access it. The volatile keyword forces the compiler to keep the variable in registers for the minimal amount of time and to return it to memory as soon as possible.

In a single-threaded process, there is only one thread of execution. In the traditional Unix/Linux, each process had only one thread. But modern systems allow multiple threads in a single address space. This is good because these threads can share variables and communicate very efficiently.

However with multiple concurrent threads executing in a single address space, care must be taken to make sure the threads synchronize with each other properly. This is a fascinating topic, explored in great length in the Operating Systems class.

Normally, a function is called and returns and you can think of a series of calls and returns like a set of nested parentheses. Each open parenthesis "(" is a "call" and each close parenthesis ")" is a return.

(((() (()())) ()) (()))

Of course you can have recursion:

(((((((((( )))))))))))

With **recursion**, you can have a single function that is active many times at one moment. However, each invocation (except one) is momentarily suspended while the functions it called are executing. So there is a simple concurrency control mechanism: there is not really any concurrency at all. The **caller** is suspended while the **callee** executes.

With a single thread of execution, a simple stack will suffice to keep functions straight. At any one moment, only one invocation of a function is actively executing. All others are frozen until the currently executing function completes and returns.

However, with multiple threads, a single function can be active in different threads. One function can be running *concurrently with itself*. With no further action on the part of the programmer, there is no concurrency control. If the functions reads and modifies a non-local variable, then there could easily be a race condition.

Some functions will work properly in a multi-threaded process and others will not. A function that will work properly in a multi-threaded environment is said to be **reentrant**.

Reentrant functions generally only use local variables and avoid the use of global variables. If the function does access shared, global variables, it must be done carefully, since other copies of the same function will access the same variables concurrently. So all accesses to non-local variables must be carefully managed.

A function is async-signal-safe if it is reentrant or non-interruptible. Some system calls are asynch-signal-safe, and some are not.

Async-signal-safe:

```
_exit, write, wait, waitpid, sleep, kill
```

Not async-signal-safe:

```
printf,  sprintf, malloc, exit
```

The Bryant and O'Hallaron textbook authors provide a library of functions you can use within signal handlers. They call their library of functions "Sio" (Safe I/O Library).

```
ssize_t Sio_puts(char s[]) /* Put string */
ssize_t Sio_putl(long v)   /* Put long */
void Sio_error(char s[])   /* Put message & exit */
```

Here is an example handler using these functions.

```
/* Safe SIGINT handler */
void sigint_handler(int sig) {
    Sio_puts("You hit ctrl-c!\n");
    sleep(2);
    Sio_puts("Let me think...");
    sleep(1);
    Sio_puts("Good bye!\n");
    _exit(0);
}
```

Here is the output from this code:

```
linux> ./sigintsafe
<ctrl-c>
You hit ctrl-c!
Let me think...Good bye!
linux>
```

Here is a program that contains a subtle bug.

The main function begins by installing a signal handler for SIGCHLD. This signal is sent every time a child terminates. Then it has a for-loop that creates a bunch of child processes. Each child waits for one second and then terminates.

Each time a child terminates, a SIGCHLD signal is sent to the parent.

In response, the handler in the parent invokes wait() to reap the child and get the child's exit status. The program also maintains a count "ccount" to keep track of how many unreaped children are left. When the handler reaps a child, it decrements this count.

After creating the child processes, the parent goes into a spin-loop.

A **spin-loop** is a way to wait for an event. The idea is that the spin-loop simply tests the condition and then repeats. A spin-loop may waste some CPU time because the loop might execute hundreds or millions of times, continually testing for the condition. Nevertheless, a spin-loop will work and in some situations is actually the best design choice.

Eventually the count should be decremented to zero and then the parent should exit. But, as we said, there is a bug.

```
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

```
void example () {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
        ;
}
```

Here is the output:

```
linux> ./example
Handler reaped child 23240
Handler reaped child 23241
```
… program hangs here!

Only two children are reaped (we assume N is larger than 2), and the program hangs.

The problem is that *signals are not queued*. Notice that there is a call to **sleep()** in the handler.

You should always be able to insert a call to sleep() anywhere in your code without causing it to fail. A call to sleep() should only slow a program down, never change its output!

But this code has a race condition. A **race condition** is a type of bug. The incorrect behavior is timing dependent. Sometimes the program might work okay, while other times the program might fail. With a race-condition, the behavior of the program is dependent on the vagaries of the kernel's process scheduler. Race-conditions are almost always very subtle, hard to find, and hard to understand. They are probably more common that we appreciate, since the program works most of the time and when it fails, the failure is not always repeatable.

The authors have inserted a call to sleep() in order to increase the odds that the race condition bug will manifest itself and incorrect behavior will be observed.

During the sleep() waiting, several children may terminate. Several SIGCHLD signals will be sent, but they are not queued. Only one signal will be received, regardless of how many are sent. So while processing the first child's termination (including the long sleep), the remaining children all terminate. Many signals are sent during this time, but only one will be received.

So after the handler completes, the kernel will check to see if there are any new signals that are PENDING but not BLOCKED. The answer is "yes", the SIGCHLD signal was sent many times. So the signal handler will be invoked a second time, but only once no matter how many times the signal was sent. The count variable will not be decremented correctly so the spin-loop never terminates.

Here is the fix that the authors propose:

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

What they suggest is to try to reap as many children as possible in the handler.

However their code seems to be flawed.

Imagine that the while-loop picks up a bunch of terminated children and then, when there are no more terminated children, the loop terminates. But next, notice that there is a moment in time directly before the code restores errno. Imagine that in this moment, several children all terminate. Multiple signals will be sent and some will be lost.

This shows how tricky writing concurrent code is. This example should be a lesson on the extreme skill required in writing concurrent code and the ease with which race conditions can be overlooked.

The C language provides two functions called **setjmp()** and **longjmp()** that have unusual properties. These functions are not system calls.

Together, setjmp() and longjmp() provide a way to jump from one place in a program to another distant location. The two locations do not even need to be in the same function!

The typical use is to deal with errors. Imagine that you have a program main() that calls some function foo1() which calls some other function foo2() and so on. At some point, the call stack may be quite deep:

main → foo1 → foo2 → foo3

While executing in foo3, imagine that an catastrophic error occurs. Typical programs print an error message and call exit(), but imagine that this is not acceptable. Instead, the program needs to return to some previous function, such as foo1() and keep going.

You can use setjmp() and longjmp() to achieve this functionality.

Languages like Java and KPL provide a **try-throw-catch** mechanism. This is a much better approach, but it was not invented until after the C language was well established. The try-throw-catch mechanism can be viewed as a cleaned-up, safe way to do this sort of thing. The setjmp() and longjmp() mechanism is unsafe, in the sense that if you make an error, you can crash your program.

Here's how to do use these functions. First, there is a variable called the "jump buffer" which you must include in your program. There is a type called **jmp_buf** for variables like this. We'll call our variable **jb**. Basically, **jb** is just an array of bytes and the exact nature is processor / system dependent.

```
jmp_buf jb;
```

In our example, we want to jump from foo3 back into foo1. So when we are in foo1, before we call foo2, we will call setjmp:

```
i = setjmp (jb);
```

The call to setjmp() will return 0. What setjmp() does is save the current state of all the CPU registers, including the PC and the stack pointer in the array of bytes we named jb.

Then foo1() calls foo2() which calls foo3(). Now assume the catastrophic error condition (whatever it is) is detected and we are ready to completely abandon foo3() and foo2() and jump back into foo1(). So within foo3() we invoke longjmp():

```
longjmp (jb, 123);
```

We provide the longjmp() function with jb, which contains all the info needed to return. We also provide an integer, such as 123.

What happens next is that there is no return from the longjmp() function. Instead there is a return from the original setjmp! Yes, we already returned from that function way back when we

originally called it. Remember? It returned a 0. But now it returns again, only this time it returns whatever integer we supplied to longjmp(). In this example, setjmp() will return 123.

These functions work as follows. Setjmp when first called will save all the registers as they are at that moment. Then later, when longjmp is called, it will take those saved register values and load them into the CPU's registers. When longjmp reloads the stack pointer, it will effective trim the stack back to where it was when setjmp was originally called. When longjmp reloads the PC, that will effectively cause a jump back to the instructions within setjmp.

Below is an example demonstrating setjmp() and longjmp().

In this example, main()will call foo() which will call bar(). We might detect a "catastrophic error" in different places. The error might occur in foo() or it might occur in bar(). Regardless of where the error occurs, we want to return to main() and deal with the error there.

The main function will first call setjmp to capture the state. Setjmp returns 0 so the switch statement will select "case 0" and the function foo() will be called. If there are no errors, then foo() will return. We then "break" out of this case and the switch statement is finished and main() then returns.

The function foo() will do some stuff including checking for error conditions and calling bar(). The function bar() will also do some stuff and check for error conditions.

There are a couple of error conditions that might invoke the jump back to main. If some condition called "errorXXX" occurs or if "errorYYY" occurs, we want to break off execution and immediately return to main().

This code uses the integer to communicate which error caused the jump. If foo() detects "errorXXX" , then number 1 is used. If bar() detects "errorYYY", then 2 is used.

For example, if bar() detects that errorYYY has occurred, it invokes longjmp() with 2. In this case, the next thing that happens is that we return a second time from the call to setjmp(). So now we are back in main() and setjmp() returns, but this time it returns the number 2.

As we saw, after invoking setjmp(), the switch statement tests the return value from setjmp(). This is convenient. If the value is 2, the switch will select "case 2" and do whatever is appropriate to deal with an occurrence of errorYYY.

```
jmp_buf jbuf;

int main()
{
    switch(setjmp(jbuf)) {
    case 0:
```

```
            foo();
            break;
        case 1:
            printf("Detected an errorXXX condition in foo\n");
            break;
        case 2:
            printf("Detected an errorYYY condition in bar\n");
            break;
        default:
            printf("Unknown error condition\n");
        }
    }

    void foo(void) {
        …
        if (errorXXX)
         longjmp(jbuf, 1);
        …
        bar();
    }

    void bar(void) {
        …
        if (errorYYY)
            longjmp(jbuf, 2);
        …
    }
```

The long jump mechanism can be misused by the programmer. The requirement is that we must never jump back into a function that has already returned. Consider this code:

```
jmp_buf jbuf;

main() {
  foo();
  bar();
}

foo() {
   if (setjmp(jbuf)) {
    …
  }
}

bar() {
  longjmp(jbuf, 1);
```

```
}
```

The main() function calls foo() and after foo() returns, main() calls bar(). Within foo(), there is a call to setjmp() which captures the state of the process at that moment. But then foo() returns. The stack frame for foo() is popped off the stack and the stack frame for bar() is pushed onto the stack. Then, within bar(), there is a call to longjmp(). This will restore the registers and we will try to make a jump back to the moment when setjmp() was called.

Suddenly we are back executing code in foo(). But we can't just return to foo(): the stack is all messed up! The frame for foo() is gone and has been overwritten with the frame for bar(). The local variables for foo() – if any – have been overwritten. And when foo() tries to return it is a problem, since foo()'s return address has most likely been overwritten, too. So this program can easily crash or behave in unpredictable ways.

Here is an example program combining signal handling with long jumps. Since we are jumping in/out of signal handlers, we use a variation called **sigsetjmp()** and **siglongjmp()**.

```
#include "csapp.h"

sigjmp_buf jbuf;

void handler(int sig) {
    siglongjmp(jbuf, 1);
}

int main() {
    if (!sigsetjmp(jbuf, 1)) {
        Signal(SIGINT, handler);
        Sio_puts(”STARTING\n");
    } else {
        Sio_puts(”RESTART\n");
    }

    while(1) {
        Sleep(1);
        Sio_puts("processing...\n");
    }
    exit(0); /* Control never reaches here */
}
```

Here is some output:

```
linux> ./restart
STARTING
processing...
```

```
processing...
processing...
<control-c typed here>
RESTART
processing...
processing...
<control-c typed here>
RESTART
processing...
processing...
processing...
```

The main program begins by calling sigsetjmp() as its first action. Then it prints "STARTING" and enters an infinite loop, causing it to print "processing…" every second. When the user types control-c, the handler is invoked. All the handler does is invoke siglongjmp(). This causes a return from sigsetjmp(), which effectively restarts the main() function from the beginning. Only the second time, the call to sigsetjmp() returns the number 1. This causes main() to print "RESTART" and then restart the infinite loop.

For this program, the user will need to use the **kill** command.

In my experience, I have never had reason to use setjmp() and longjmp() in a program. However, I can imagine a situation in which they could be useful and in which nothing else would work quite as nicely.