1.5.1) Dual-Mode Operation

Needs to protect the OS from user programs. Also need to protects users from each other.

The approach taken by most computer systems is to provide hardware support that make it possible to differentiate among various modes of execution. At the very least we need two separate modes.

When a user application request a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request.
System and Application Programs

Computer Hardware

Operating System

Services

- program execution
- I/O operations
- communication

helpful for the user

- error detection
- file systems

resource allocation

accounting

ensuring the efficient operation of the system itself.

- protection and security

system calls

user interfaces

- GUI
- batch
- command line

System calls are helpful for the user, ensuring the efficient operation of the system itself.
2.3) The C Standard Library

The C standard library provides a portion of the system-call interface for many versions of UNIX and Linux.

```c
#include<stdio.h>
printf("The meaning of life is %d", 42);
// Will return here after printf()
```

In this example, the printf() statement is intercepted by the standard C library. The standard C library takes care of constructing the string to print and invokes the write system call to actually print the string.
Chapter 3: Processes

An operating system executes a variety of programs:

★ Batch system – jobs
★ Time-shared systems – user programs or tasks

Text book uses the terms job and process almost interchangeably

★ Process – a program in execution; process execution must progress in sequential fashion

A process includes:
★ program counter
★ stack
★ data section
3.1.1) A process in memory

**Stack:** Temporary data such as function parameters, local variables and return addresses.

The stack grows from high addresses towards lower address.

**Heap:** Dynamically allocated (malloc) by the program during runtime.

The heap grows from low addresses towards higher address.

**Data:** Statically (at compile time) global variables and data structures.

**Text:** The program code.
3.1.1) An example of memory usage

The size of the needed storage for the **global** variable $x$ is known at compile time and storage for $x$ is allocated in the **data segment**.

The size of the needed storage for the **local** variables $y$ and $str$ are also known at compile time but these are only needed within main and storage is allocated on the **stack**.

The value of the **parameter** to the malloc function might not be know at compile time. Parameters used in function calls are pushed onto the **stack**.

The **malloc** function **dynamically** allocates memory in the heap.

```
// Example C program
#include <stdlib.h>

int x;

int main(void) {
    int y;
    char* str;
    str = malloc(50);
    return EXIT_SUCCESS;
}
```
3.1.1) An example of memory usage

The size of the needed storage for the **global variable** \( x \) is known at compile time and storage for \( x \) is allocated in the **data segment**.

The size of the needed storage for the **local variables** \( y \) and \( str \) are also known at compile time but these are only needed within main and storage is allocated on the **stack**.

The value of the **parameter** to the `malloc` function might not be know at compile time. Parameters used in function calls are pushed onto the **stack**.

The **malloc** function **dynamically** allocates memory in the heap.

When `malloc` returns, the parameter used to call `malloc` is no longer needed and is popped from the **stack**.

When `main` returns, the local variables \( y \) and \( str \) are no longer needed and are popped from the **stack**.
In a modern languages running on a modern OS, you'll get either a stack overflow (hurray!) or `malloc()` will fail when you try to grow the heap. But not all software is modern, so let's look at the failure modes:

★ If the stack grows into the heap, the typically C compiler will silently start to overwrite the heap's data structures. On a modern OS, there will be one or more virtual memory **guard pages** which prevent the stack from growing indefinitely. As long as the amount of memory in the guard pages is at least as large as the size of the growing procedure's activation record, the OS will guarantee you a segfault. If you're DOS running on a machine with no MMU, you're probably hosed.

★ If the heap grows into the stack, the operating system should always be aware of the situation and some sort of system call will fail. The implementation of `malloc()` almost certainly notices the failure and returns NULL. What happens after that is up to you.

For a discussion of this subject, see http://stackoverflow.com/questions/1334055/what-happens-when-stack-and-heap-collide
A process can be in one of many states:

- New
- Running
- Waiting
- Ready
- Terminated
In brief, the PCB serves as the repository for any information that may vary from process to process.
3.1.3) Context switch from one process to another

### PCB
- state
- process id/number
- program counter
- registers
- CPU scheduling information (chapter 5)
- memory management information (chapter 8)
- accounting information
- memory limits
- I/O status information
When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a *context switch*.

*Context* of a process represented in the PCB.

Context-switch time is *overhead*; the system does no useful work while switching.

Time dependent on *hardware support*.

<table>
<thead>
<tr>
<th>PCB₀</th>
<th>PCB₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>state</td>
</tr>
<tr>
<td>process id/number</td>
<td>process id/number</td>
</tr>
<tr>
<td>program counter</td>
<td>program counter</td>
</tr>
<tr>
<td>registers</td>
<td>registers</td>
</tr>
<tr>
<td>CPU scheduling information (chapter 5)</td>
<td>CPU scheduling information (chapter 5)</td>
</tr>
<tr>
<td>memory management information (chapter 8)</td>
<td>memory management information (chapter 8)</td>
</tr>
<tr>
<td>accounting information</td>
<td>accounting information</td>
</tr>
<tr>
<td>memory limits</td>
<td>memory limits</td>
</tr>
<tr>
<td>I/O status information</td>
<td>I/O status information</td>
</tr>
</tbody>
</table>
Do you remember the difference between *multiprogramming* and *time sharing* (multi tasking)?
1.4) Multiprogramming

The OS keeps several Jobs in memory at the same time.

One Job is chosen for execution and the status is changed to executing.

Eventually, the job may have to wait for some task, such as an I/O operation.

The status of the executing job is changed to waiting.

Instead of being idle waiting for the task to complete, the OS simply switches to another job.

Eventually the task job 1 is waiting for will complete and job 1 will change status from waiting to ready to run.
1.4) Multitasking (timesharing)

A logical extension of multiprogramming

On Job is chosen for execution and the status is changed to executing.

A timer is used to enforce interrupts to be sent to the CPU at regular intervals.

The status of the executing job is changed to Ready to Run.

A new job is selected to utilize the CPU.

The switching between jobs is so fast that the user can interacts with each program as if they were all executing in parallel.

An executing job can still be taken of the processor by other interrupts (for example I/O) and the status changed to waiting.
The objective of multiprogramming is to have some process running, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
The objective of **multiprogramming** is to have some process running at all time, to **maximize CPU utilization**.

The objective of **time sharing** is to switch the CPU among processes so frequently that **users can interact** with each program while it is running.
The objective of multiprogramming is to have some process running at all time, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
We have already talked about the *ready queue*. Maybe we need more queues?

Processes migrate among various queues:

**Job queue:** set of all processes in the system.

**Ready queue:** set of all processes residing in main memory, ready and waiting to execute.

**Device queues:** set of processes waiting for an I/O device.
3.2.1) Ready Queue And Various I/O Device Queues

![Diagram showing Ready Queue and various I/O Device Queues]
3.1.3) Representation of Process Scheduling

**Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue.

**Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could be removed forcibly from the CPU.
- The process creates a new subprocess and wait for the subprocess’s termination.
**Long-term scheduler**
(or job scheduler)

Selects which processes should be brought into the ready queue.

A process may execute for only a few milliseconds before waiting for an I/O request.

Often, the short-term scheduler executes at least once every 100 milliseconds.

Therefore, **the short-term scheduler must be fast.**

---

**Short-term scheduler**
(or CPU scheduler)

Selects which process should be executed next and allocates CPU.

The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).

The long-term scheduler executes much less frequently compared to the short-term scheduler.

Therefore, **can afford to take more time** compared to the short-term scheduler.

---

The primary distinction between these two schedulers lies in frequency of execution.
3.2.2) Addition of Medium Term Scheduling

**Swapping:** Sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. A process can be swapped in and out from memory by the medium-term scheduler.

**I/O-bound process:** spends more time doing I/O than computations, **many short CPU bursts.**

**CPU-bound process:** spends more time doing computations; **few very long CPU bursts.**
3.3.1) Process Creation

**Parent** process create **child** processes, which, in turn create other processes, forming a tree of processes.

Generally, process identified and managed via a **process identifier** (pid).

**Resource sharing models:**
- ★ Parent and children share all resources
- ★ Children share subset of parent’s resources
- ★ Parent and child share no resources

**Execution models:**
- ★ Parent and children execute concurrently
- ★ Parent waits until children terminate

**Address space models:**
- ★ Child duplicate of parent
- ★ Child has a new program (different than parent) loaded into it
Create a new process using `fork()`

Parent Process

- Parent calls `fork()`
- Resources (open files, etc)

Child Process

- TEXT (instructions)
- DATA

OS creates a new process - a child process

The child process is a copy of the parent (copy of TEXT, DATA and Resources)
3.3.1) Process Creation in Unix

Parent uses the **fork()** system call to create a new child process.

Parent may use the **wait()** system call to wait for the child to terminate.

The child may use the **exec()** system call used after a fork to replace the process’s memory space with a new program.

The child uses the **exit()** system call to terminate.
After `fork()`, parent and child will execute the same program (TEXT). How can they be distinguished from each other?
On success, `fork()` returns twice!

`fork()` returns $\text{pid} > 0$ if executing in the parent process.

`fork()` returns 0 if executing in the new child process.
3.3.1) C program example using fork()

Always check for errors!!!

On error, fork() returns -1

fork() returns 0 if executing in the new child process.

Returns pid > 0 if executing in the parent process.

It’s the pid of the child that is returned.

```c
#include <stdio.h>  // printf()
#include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE

int main(void) {
    pid_t pid;
    // Parent uses the fork() system call to create a child process.
    switch(pid=fork()) {
        case -1: // ERROR
            break;
        case 0: // CHILD PROCESS
            // Child process code here
            exit(EXIT_SUCCESS);
        default: // PARENT PROCESS
            // Parent process code here
            wait(NULL);
            exit(EXIT_FAILURE);
    }
}
```
3.3.1) C program example using fork()

```c
#include <stdio.h>  // printf()
#include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE

int main(void) {
    pid_t pid;
    // Parent uses the fork() system call to create a child process.
    switch(pid=fork()) {
        case -1: // ERROR
            perror("Could not create child process.");
            exit(EXIT_FAILURE);
        case 0: // CHILD PROCESS
            printf("Hello! My parent is %ld\n", (long) getpid(), (long) getppid());
            printf("Child says: Godbye!\n", (long) getpid());
            exit(EXIT_SUCCESS);
        default: // PARENT PROCESS
            printf("Parent says: Child got PID %ld\n", (long) getpid(), (long) pid);
            pid = wait(NULL);
            printf("Parent says: Child %ld died:(\n", (long) getpid(), (long) pid);
            printf("Parent says: Godbye!\n", (long) getpid());
            exit(EXIT_SUCCESS);
    }
}
```

Always check for errors!!!

On error, `fork()` returns -1.

On success, `fork()` return twice!

`fork()` returns 0 if executing in the new child process.

Returns pid > 0 if executing in the parent process.

It’s the pid of the child that is returned.

Parent uses `wait()` to suspend execution until one of its child processes terminates.
3.3.1) C program example using fork() - test run

```
karl ~/Documents/Teaching/OS/2011: gcc simple_fork.c
karl ~/Documents/Teaching/OS/2011: ./a.out
<32435> Parent says: Child got PID <32436>
<32436> Child says: Hello! My parent is <32435>
<32436> Child says: Goodbye!
<32435> Parent says: Child <32436> died :( 
<32435> Parent says: Goodbye!
karl ~/Documents/Teaching/OS/2011:
```
3.3.2) Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it using the `exit()` system call.

The parent may use `wait()` to read the status value of a terminated child process.

Using `exit()`, the child may leave a status value (typically an integer) to its parent process to read later.
Processes within a system may be independent or cooperating. Cooperating processes can affect or be affected by other processes, including sharing data.

Reasons for cooperating processes?

- Information sharing
- Computation speedup
- Modularity
- Convenience
3.4) Inter Process Communication (IPC)

Message Passing

Communication take place by means of messages exchanged between the cooperating processes.

Shared Memory

A region of memory that is shared by cooperating process is established. Process can then exchange information by reading and writing data to the shared region.
3.4.1) The Producer-Consumer Problem

Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.

What if the producer produces faster than the consumer consumes or vice versa?

*John relies on the input of Jerry to carry out his job*

What if the producer produces faster than the consumer consumes or vice versa?
3.4.1) The Producer-Consumer Problem

The producer generates data and puts it into a buffer.

An intermediate buffer.

The consumer consumes data from the buffer.

*Unbounded-buffer* places no practical limit on the size of the buffer. *Bounded-buffer* assumes that there is a fixed buffer size.
3.4.1) A N item shared memory FIFO buffer

Let's look at a small example with N = 3

When the buffer is empty: \( \text{in} == \text{out} \)

\[
\text{in}_{\text{after\_write}} = (\text{in}_{\text{before\_write}} + 1) \bmod N
\]

\[
\text{out}_{\text{after\_read}} = (\text{out}_{\text{before\_read}} + 1) \bmod N
\]
3.4.1) A N item shared memory FIFO buffer

A first attempt to implement this in C.

```c
#include <stdio.h>   // printf(), NULL
#include <stdlib.h>  // exit(), EXIT_FAILURE

#define BUFFER_SIZE 3

typedef struct {
   char data;
} item_t;

// The following data must be shared by producer and consumer.

item_t buffer[BUFFER_SIZE];
int in  = 0;
int out = 0;
```

Using a structure, we can easily change the contents of the buffer later.
3.4.1) A N item shared memory FIFO buffer

A first attempt to implement this in C.

```c
item_t consume(void) {
    // Wait for new items in the buffer an consume them.
    while ( /* */ )
        ; // Do nothing -- nothing to consume.

    // Remove an item from the buffer.
    item_t item = buffer[out];
    // Update the out index.
    out = /* */ ;
    return item;
}

void produce(item_t item) {
    // Wait for a free slot in the buffer.
    while (/* */ )
        ; // Do nothing -- no free slots in buffer.

    // Write item to buffer
    buffer[in] = item;
    // Update the in index.
    in = /* */ ;
}
```
A N item shared memory FIFO buffer

A first attempt to implement this in C.

```c
item_t consume(void) {
    // Wait for new items in the buffer and consume them.
    while (in == out)
        ; // Do nothing -- nothing to consume.

    // Remove an item from the buffer.
    item_t item = buffer[out];
    // Update the out index.
    out = (out + 1) % BUFFER_SIZE;
    return item;
}

void produce(item_t item) {
    // Wait for a free slot in the buffer.
    while (( (in + 1) % BUFFER_SIZE) == out)
        ; // Do nothing -- no free slots in buffer.

    // Write item to buffer
    buffer[in] = item;
    // Update the in index.
    in = (in + 1) % BUFFER_SIZE;
}
```
3.4.1) A N item shared memory FIFO buffer

Example step-by-step

```c
void produce(item_t item) {
    // Wait for a free slot in the buffer.
    while (( (in + 1) % BUFFER_SIZE) == out) {
        // Do nothing -- no free slots in buffer.
    }

    // Write item to buffer
    buffer[in] = item;
    // Update the in index.
    in = (in + 1) % BUFFER_SIZE;
}
```

0) We start with empty buffer (in == out == 0).

1) Let’s produce two items A and B.

2) When attempting to write C to slot 2:

```
// (in +1) % BUFFER_SIZE
(2 +1) % 3 == 0 == out   ==> BLOCKED
```

3) Consume A.

4) When attempting to write C to slot 2:

```
// (in +1) % BUFFER_SIZE
(2 +1) % 3 == 0 != out  (out == 1) not blocked anymore.
```

This implementation only allows for at most N-1 items in the buffer at any time.
3.4.1) A N item shared memory FIFO buffer

A first attempt to implement this in C.

```c
void produce(item_t item) {
    // Wait for a free slot in the buffer.
    while ((in + 1) % BUFFER_SIZE == out) {
        // Do nothing -- no free slots in buffer.
    }

    // Write item to buffer
    buffer[in] = item;
    // Update the in index.
    in = (in + 1) % BUFFER_SIZE;
}

item_t consume(void) {
    // Wait for new items in the buffer an consume
    while (in == out) {
        // Do nothing -- nothing to consume.
    }

    // Remove an item from the buffer.
    item_t item = buffer[out];
    // Update the out index.
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

This implementation only allows for at most N-1 items in the buffer at any time. In section 6.1 we'll modify this solution to allow for N items at the same time.

This implementation does not allow for concurrent access by the producer and consumer. For this solution to work, both the produce operation and consume operation must be atomic. In chapter 6 we'll learn how to synchronize access to shared memory.

We yet don't know how to share memory between processes. We'll learn how to do this in section 3.5.1.
The kernel keeps track of I/O objects using descriptors.

By default, a process got three open descriptors.

0 - Standard Input (STDIN)

1 - Standard Output (STDOUT)

2 - Standard Error (STDERR)
Reading files using the `open()` and `read()` system calls

When opening a file for reading, a descriptor number is returned. In this example, a descriptor with value is 3 returned.

Reading from the file is done by using the descriptor as a reference.

```c
int fd;
char buff[BUFF_SIZE];

fd = open("foo.txt", O_RDONLY);
read(fd, buff, BUFF_SIZE);
```
# Duplication of descriptors

```c
#include <unistd.h>  // Header file that provides
                // access to the POSIX
                // operating system API.

int dup2(int fildes, int fildes2);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fildes</td>
<td>The source file descriptor. This remains open after the call to dup2.</td>
</tr>
<tr>
<td>int fildes2</td>
<td>The destination file descriptor. This file descriptor will point to the same file as fildes after this call returns.</td>
</tr>
<tr>
<td>return value</td>
<td>dup2 returns the value of the second parameter (fildes2) upon success. A negative return value means that an error occurred.</td>
</tr>
</tbody>
</table>
Redirection using the dup2() system call

It’s possible to redirect input or output using the `dup2()` system call.

```c
int fd;
char buff[BUFF_SIZE];

fd = open("foo.txt", O_RDONLY);
dup2(fd, 0);
```

Change the descriptor 0 (STDIN) to refer to the same object as the descriptor `fd`.
Redirection using the `dup2()` system call

It’s possible to redirect input or output using the `dup2()` system call.

```c
int fd;
char buff[BUFF_SIZE];

fd = open("foo.txt", O_RDONLY);
dup2(fd, 0);
```

Change the descriptor 0 (STDIN) to refer to the same object as the descriptor `fd`.

- **User Space**
  - `int fd;`
  - `char buff[BUFF_SIZE];`
  - `fd = open("foo.txt", O_RDONLY);`
  - `dup2(fd, 0);`

- **Kernel Space**
  - **Descriptors**
    - 0 stdin
    - 1 stdout
    - 2 stderr
    - 3 read file

- **File**
  - "foo.txt"
3.6.3) Pipes

Pipes where one of the first IPC mechanisms in early UNIX systems.

What do we need to think about when implementing pipes?

★ Does the pipe allow unidirectional or bidirectional communication?

★ If two-way (bidirectional) communication is allowed, is it half duplex (data can travel only one way at time) or full duplex (data can travel in both directions at the same time?)

★ Must a relationship (such as parent-child) exist between communicating processes?

★ Can the pipes communicate over a network, or must the communication process reside on the same machine?
3.6.3.1) Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer consumer fashion:

★ The *producer writes* to one end of the pipe (the write end).

★ The *consumer reads* from one end of the pipe (the read end)

★ As a result, ordinary pipes are *unidirectional*, allowing only one-way communication.

If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
As a result of the `pipe()` system call, a pipe object is created. A pipe is a FIFO buffer that can be used for inter process communication (IPC).

Now a single process can write data to, and later read data from a pipe ...
Using `pipe()` together with `fork()`

After `fork()`, the child process have the same set of descriptors including read and write descriptors to the pipe.

```c
int pfd[2];
pipe(pfd);
// pfd[0] = 3
// pfd[1] = 4
fork();
```
Using pipe(), fork() and close()

**Parent Process**

```c
int pfd[2];
pipe(pfd);
// pfd[0] = 3
// pfd[1] = 4
fork();
close(pfd[0]);
```

The parent can close the read descriptor to the pipe.

Now, the parent can act as a single producer and the child as a single consumer of data through the pipe using the read() and write() system calls - just as if it was a file.

**Child Process**

```c
close(pfd[1]);
```

The child can close the write descriptor to the pipe.

**Descriptors**

<table>
<thead>
<tr>
<th>Parent Process</th>
<th>Child Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 stdin</td>
<td>0 stdin</td>
</tr>
<tr>
<td>1 stdout</td>
<td>1 stdout</td>
</tr>
<tr>
<td>2 stderr</td>
<td>2 stderr</td>
</tr>
<tr>
<td>3 pipe read</td>
<td>3 pipe read</td>
</tr>
<tr>
<td>4 pipe write</td>
<td>4 pipe write</td>
</tr>
</tbody>
</table>

**Kernel Space**
```c
int pfd[N][2];
pid_t pid;
int i, status;

for (i = 0; i < N; i++) {
    pipe(pfd[i]);
}

for (i = 0; i < N; i++) {
    pid = fork();

    if (pid < 0) {
        perror("Child was not created");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // CHILD GOES here...
        exit(EXIT_SUCCESS);
    } else {
        // PARENT GOES here...
    }
}
```

If we need several pipes, it's convenient to use an array where each element is a pair of descriptors. 

**Example: N = 2**

Since the pipes are created by the parent before the children are created, all children will have open descriptors to all pipes.

<table>
<thead>
<tr>
<th>pipe nr</th>
<th>read descriptor</th>
<th>write descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pfd[0][0]</td>
<td>pfd[0][1]</td>
</tr>
<tr>
<td>1</td>
<td>pfd[1][0]</td>
<td>pfd[1][1]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>pfd[N-1][0]</td>
<td>pfd[N-1][1]</td>
</tr>
</tbody>
</table>

**pfd** is a two dimensional array, an array where each element is a two element array.
Signals

A signal is a limited form of **inter-process communication** (IPC).

Essentially it is an **asynchronous** notification sent to a **process** in order to notify it of an event that occurred.

When a signal is sent to a process, the operating system **interrupts** the process's normal **flow of execution**.

If the process has previously registered a **signal handler**, that routine is executed. Otherwise the default signal handler is executed.
Signals

Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals, for example:

★ **Ctrl-C** sends an INT signal (SIGINT); by default, this causes the process to **terminate**.

★ **Ctrl-Z** sends a TSTP signal (SIGTSTP); by default, this causes the process to **suspend** execution.
Signals

The `kill(2)` system call will send the specified signal to the process, if permissions allow. Similarly, the `kill(1)` command allows a user to send signals to processes.

The `raise(3)` library function sends the specified signal to the current process.

Exceptions such as **division by zero** or a **segmentation violation** will generate signals (here, SIGFPE and SIGSEGV respectively, which both by default cause a core dump and a program exit).

The kernel can generate a signal to notify the process of an event. For example, **SIGPIPE** will be generated when a process writes to a pipe which has been closed by the reader; by default, this causes the process to terminate.
Sending yourself a signal

An example of code that causes a process to suspend its own execution by sending itself the STOP signal:

```c
#include <unistd.h>    /* standard unix functions, like getpid() */
#include <sys/types.h> /* various type definitions, like pid_t */
#include <signal.h>    /* signal name macros, and the kill() prototype */

/* first, find my own process ID */
pid_t my_pid = getpid();

/* now that i got my PID, send myself the STOP signal */
kill(my_pid, SIGSTOP);
```
Setting up a signal handler using `signal()`

```c
#include <stdio.h> /* standard I/O functions */
#include <unistd.h> /* getpid(), pause() */
#include <sys/types.h> /* pid_t */
#include <signal.h> /* signal name macros, and the signal() */
/* prototype */

/* A signal handler is an ordinary function. */
void catch_int(int sig_num) {
    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    /* and print the message */
    printf("Don't do that");
    fflush(stdout);
}

/* Set the INT (Ctrl-C) signal handler to 'catch_int' */
signal(SIGINT, catch_int);

/* An infinite loop of doing nothing. */
for ( ;; )
    pause(); /* Suspend until any signal is received. */
```

A code snippet that causes the program to print the string "Don't do that" when a user presses Ctrl-C.
Playing with signals

Write a program that count the number of times the user pressed Ctrl-C.

The 7th time Ctrl-C is pressed, the program should terminate.
The signal() system call is used to set a signal handler for a single signal type.

Where are static variables stored?

Make sure the program will terminate if Ctrl-C is pressed.

The signal() system call is used to set a signal handler for a single signal type.
<table>
<thead>
<tr>
<th>Attempt</th>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Empty pipe, writer attached</td>
<td>Read blocked</td>
</tr>
<tr>
<td>Write</td>
<td>Full pipe, reader attached</td>
<td>Write blocked</td>
</tr>
<tr>
<td>Read</td>
<td>Empty pipe, no writer attached</td>
<td>EOF returned</td>
</tr>
<tr>
<td>Write</td>
<td>No reader</td>
<td></td>
</tr>
</tbody>
</table>

By default, a SIGPIPE causes the process to terminate.

Don’t forget to close unused pipe file descriptors using the close() system call.
It’s a good habit to ALWAYS close any descriptors you do not intend to use.

Example:

Child 0 will only WRITE data to the parent using pipe 0.
Child 1 will only READ data from the parent using pipe 1.
The parent will only read from pipe 0 and write to pipe 1.
As the number of children increases, the number of open descriptors connecting all processes through the pipes quickly become quite large.
3.3.1) Process Creation - one more time

DESCRIPTION

The `exec` family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function `execve(2)`. (See the manual page for `execve(2)` for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.
If `execvp()` successfully manages to replace the image of the process, this statement will never be reached.

The `execvp()` standard C library function

```c
#include <unistd.h>     // execvp()
#include <stdio.h>      // NULL
#include <stdlib.h>     // exit(), EXIT_FAILURE

int main(void) {
    char* argv[] = {"date", "+DATE: %Y-%m-%d\nTIME: %H:%M:%S", NULL};
    execvp(argv[0], argv);
    exit(EXIT_FAILURE);
}
```

argv is an array of strings.

0  The first string is the name of a system program (any program in your PATH)

1  the second string is the first parameter to the program.

2  NULL is used to indicate no more parameters.
The `execvp()` standard C library function

```c
#include <unistd.h> // execvp()
#include <stdio.h> // NULL
#include <stdlib.h> // exit(), EXIT_FAILURE

int main(void) {
    char* argv[] = {"date", "+DATE: %Y-%m-%d\nTIME: %H:%M:%S", NULL};
    execvp(argv[0], argv);
    exit(EXIT_FAILURE);
}
```

Terminal — bash — 57×7

```
karl ~/Documents/Teaching/OS/2011: gcc execvp_test.c
karl ~/Documents/Teaching/OS/2011: ./a.out
DATE: 2011-01-24
TIME: 01:25:40
karl ~/Documents/Teaching/OS/2011:
```
A first attempt at implementing a shell

```c
#include <unistd.h> // fork()
#include <sys/wait.h> // waitpid()
#include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE
#include <stdio.h> // printf(), stdin

#define CMD_LEN 128

int main(int argc, char **argv) {
    char cmd[CMD_LEN];
    char *cmd_args[2], *buff;
    int status, n;
    pid_t child_pid;

    while(1) {
        write(1, ">> ", 3);
        n = read(0, cmd, CMD_LEN);
        if(n == 0) break; /* EOF reached; exit program */
        cmd[n - 1] = '\0'; /* replace 'n' with '\0' */

        /* fork off child to execute command */
        child_pid = fork();
        if(child_pid == 0) {
            cmd_args[0] = cmd;
            cmd_args[1] = NULL;
            execvp(cmd, cmd_args);
            /* If execvp returns, the command is bad. */
            write(1, "Command not found\n", 19);
            exit(EXIT_FAILURE);
        }

        / * wait for child to exit before continuing */
        waitpid(child_pid, &status, 0);
    }
    return 0;
}
```

0. The shell forks a new child for each command read from the user.

1. The child process uses `execvp()` to replace the current process image with a new process image. In this case the process image of the system program to run.

2. The parent waits to the child to finish executing the command.
A first attempt at implementing a shell

This is a very simple shell:
★ we can only run one system program at the time.
★ we cannot give any options to the system programs.
Writing such a program should be pretty easy - don’t you think?