Threads vs. Processes
Remember, there are advantages to both threads and processes. Sometimes the best solution is to use a thread, other times it is to use a process. Some of the key differences that we've studied already in CS 241:
- Threads within a single process share the same address space (eg: they use the same page table)... separate processes have separate address spaces.
- Threads provide very little isolation. A single seg fault in any one thread will bring down the entire process (no matter the number of threads). However, processes provide a lot of isolation.
- Threads are faster to create and faster to switch between; processes are slower.
- Threads can communicate via their shared memory space quite easily. Communicating between two processes is much more difficult and requires IPC.

IPC
IPC stands for "Inter-Process Communications" and is simply a set of methods that allow for the exchange of data among multiple processes.

You've already seen IPC at work without thinking about it:
- The Internet (specifically, "network sockets")
- Command Line Pipes (eg: `cat myfile.txt | grep whatver`)
- Signals (eg: Ctrl+C to kill a program, or a seg fault)

We will discuss six different forms of IPC during the next two weeks!

Pipe
Our first form of IPC is a pipe. A pipe is a one-way, stream-based IPC mechanism that is often synchronized via blocking read() calls. You've seen a pipe used on a command line:

```
program1 | program2
```

The system call to create a pipe is quite simple:

```
int pipe(int pipefd[2]
```

The `pipe()` system call initialize two file descriptors after it returns:
- pipefd[0]: the "read file descriptor", used for reading the contents of the pipe
- pipefd[1]: the "write file descriptor", used for writing the contents of the pipe

A pipe differs from a file as a pipe does not "store" the contents of what was in the pipe. Once data has been read form the pipe, the dead is gone and the pipe is empty until something else is written.

Coding Example
The following code shows data being read in from one process and piped to another process to be outputted to the screen:

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

void main()
{
        int fds[2];

        int read_fd = fds[0];
        int write_fd = fds[1];

        pid_t pid = fork();

        if (pid == 0)  // child
        {
                // Read what the parent is sending
                while (1) {
                        char c;
                        read(read_fd, &c, 1);
                        fprintf(stderr, "%c\n", c);
                }
        }
        else if (pid > 0) // parent
        {
                char *line = malloc(100);
                size_t line_len = 100;

                while (1) {
                        getline(&line, &line_len, stdin);
                        write(write_fd, line, strlen(line));
                }
        }
}
```

We can also create our own program to do pipe the output of one process to the input of another process, just like the command line pipe. The code shown in lecture to do this:

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

int main(int argc, char **argv)
{
        char *first = malloc(100), *second = malloc(100);
        size_t first_len = 100, second_len = 100;

        printf("First command: ");
        fflush(stdout);
        getline(&first, &first_len, stdin);
        first[strlen(first) - 1] = '\0';
        printf("Second command: ");
        fflush(stdout);
        getline(&second, &second_len, stdin);
        second[strlen(second) - 1] = '\0';


        int fds[2];
        pipe( fds );

        int read_fd = fds[0];
        int write_fd = fds[1];

        pid_t pid = fork();
        if (pid > 0) // parent
        {
                /* Parse the input */
                char **av = malloc(100 * sizeof(char *));
                int av_ct = 0;

                av[av_ct] = strtok(first, " ");
                while (av[av_ct] != NULL)
                        av[++av_ct] = strtok(NULL, " ");
                /* End input parse */


                dup2(write_fd, 1); // replace stdout
                execvp(*av, av);
                exit(1);
        }
        else if (pid == 0) // child
        {
                /* Parse the input */
                char **av = malloc(100 * sizeof(char *));
                int av_ct = 0;

                av[av_ct] = strtok(second, " ");
                while (av[av_ct] != NULL)
                        av[++av_ct] = strtok(NULL, " ");
                /* End input parse */


                dup2(read_fd, 0); // replace stdin
                execvp(*av, av);
                exit(1);
        }

        printf("Error. :(\n");
        return 1;
}
```

Use of pipes
In the previous lecture, we saw a basic pattern of code used several times:

```
int fds[2];
pipe(fds);
int read_fd = fds[0], write_fd = fds[1];
pid_t pid = fork();
if (pid == 0) { /* child will use one end of the pipe */ }
else if (pid > 0) { /* parent will use the other end of the pipe */ }
```

This pattern requires any use of a pipe to have a parent-child relationship, a child-child relationship, or some relationship that shares some common parent at some point. This works well for programs that create children processes to do work, but does not work for programs that are run independently and still need to work together.

However, linux has a solution to this! It's called a fifo!

fifo
A fifo is a special file on a system's file system that corresponds to a pipe. Otherwise, it works just like a pipe! Because of this, a fifo is sometimes called a "named pipe". (Similarly, sometimes a "pipe" is sometimes called an "unnamed pipe" to make it clear it's not a fifo. However, an "unnamed fifo" doesn't make sense. In CS 241, we will use the terms "pipe" and "fifo".)

To create this special file, you run the command:

```
mkfifo name
```

For example:

```
$ mkfifo tomato
```

Creating a program to write to a fifo
We can create a program that now writes to this fifo, which is quite a bit different than writing to a pipe even though they both use a pipe to transmit the data.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
        /* Open the fifo for writing */
        FILE *fifo = fopen("tomato", "w");
        printf("fifo is open for writing\n");

        /* Read line by line from the user using getline() */
        size_t line_len = 100;
        char *line = malloc(line_len);

        while (1) {
                getline(&line, &line_len, stdin);

                /* Write the lines to the fifo */
                fprintf(fifo, "%s", line);
                fflush(fifo);
        }
}
```