Problem 1 -- Signal Numbers and Properties

In the parts below, answer both in terms of a symbolic signal number (e.g. SIGxxx) and a specific signal number of your system (be sure to mention whether you are on Linux, BSD/OSX, or something else, along with the processor architecture and kernel version)

1A) If the terminal interrupt character (by default Control-C on most systems) is received on the terminal, which signal is delivered to the foreground process?

1B) Which signal is caused by the following C expression: `*(int *)0 = 1;`

1C) Which signal or signals is/are un-blockable and un-IGNorable?

1D) Which signal does the bash shell send when you use the `fg` built-in command to bring a background process, which had previously tried to read or write the terminal while in the background, to the foreground?

Problem 2 -- Signal Handlers

```
#include <sys/signal.h>
#include <stdio.h>

void handler(int sig)
{
 static int i;
          ++i;
          fprintf(stderr,"In handler instance %d\n",i);
}

main()
{
 struct sigaction sa;
          sa.sa_handler=handler;
          sa.sa_flags=SA_NOMASK;
          (void)sigaction(SIGUSR1,&sa,NULL);
          for(;;)
                    ;
}
```

Analyze the program above and answer the following questions:

2A) How can I send the signal SIGUSR1 to this process, which stuck in the endless loop, from another terminal/shell?

2B) While we are inside the handler function, what can we say about the status of the sigprocmask? Why? Is there anything that you'd suggest changing in this program?

Experiments with Pipes and Signals

In problems 3 and 4, you will write test programs to explore certain properties of pipes and signals. Just as with lab courses in the physical sciences, the purpose of this project is to perform experiments to find answers that we already know, in order to gain proficiency with constructing and conducting such experiments. The code you will write for the 2 problems below will be an "experiment" to test for certain properties of pipes and signals. The answer is not the main point -- you already know the answer. Your code must be able to determine the answer, without manual intervention. This is what we call a "test case program" and we will use this concept again during the course.

For both programs, your submission must include a screen shot or text session capture demonstrating that they performed the experiments correctly.

Problem 3 -- Pipe capacity

We know that a pipe is a FIFO of finite capacity. When the writer(s) is faster than the reader, the pipe fills up. When it reaches capacity, the writer is blocked to enforce flow control (rate limiting). Experimentally determine the pipe capacity on your system, to the nearest 256 bytes. A suggested approach is to set the `O_NONBLOCK` flag (using the `fcntl` system call) on the write side of a pipe that you've created. Write to it using a small write size but don't ever read from it. Report how many bytes you can write into the pipe before the write system call fails. Make sure the failure is `EAGAIN` and not some other error.

Problem 4 -- Signal

If multiple instances of the identical signal are posted to a process before the signal can be delivered, behavior depends on the signal number. For so-called "real time" signals (sig# >=32), as many signal deliveries will happen as there were signals generated. But for conventional signals (sig# < 32), only one instance of the signal will be delivered.

Develop a test program to prove this for one example of a real-time and one example of a non-real-time signal. Establish a signal handler which counts the number of times that the signal is delivered and compare that to the number of times it is generated by several other processes that you fork off. It is important that you spawn a few child processes to bombard the receiver process with signals, otherwise you may never be able to generate the condition where a signal is being sent to the receiver process while its signal handler is still executing.

It is suggested that the number of "killer" children and the number of times they each send a signal be a command-line argument to your test program.

To make this test function automatically, you'll need to have your main process `wait` for all of the senders to complete, then report the signal receipt count. Since signals will be received and handled while the `wait` system call is blocking, you will see `EINTR` errors. These are not fatal errors! Make sure that for the non-real time signal, you can demonstrate that signals "do not count" and conversely for the real-time signal.