CSC209: Software Tools and Systems Programming

Week 9: Pipes and Signals [1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

Inter-Process Communication

Exit statuses provide a very limited form of *inter-process communication*, specifically between a child and parent process.

This week we will discuss *pipes*, which are a more useful mechanism for communicating between processes.

Open pipes are also associated with file descriptors, so they can be read/written very similarly to regular files.

Pipes

- A *pipe* is a one-way, first-in first-out (FIFO) communication channel that can be used between two processes
- Pipes are created using the `pipe()` system call
- A pipe has two file descriptors: One for reading and one for writing
- For usage: `man 2 pipe`
- For useful background info: `man 7 pipe`

Using a Pipe in a Single Process

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MSG_SIZE 13
char *msg = "hello, world\n";
int main(void)
{
  char buf[MSG_SIZE];
  int p[2];
  if (pipe(p) == -1) {
    perror("pipe");
    exit(1);
  }
  write(p[1], msg, MSG_SIZE); // No error checking: Bad
  read(p[0], buf, MSG_SIZE); // No error checking: Bad
  write(STDOUT_FILENO, buf, MSG_SIZE);
  return 0;
}
```

Pipe for Inter-Process Communication

- Recall: Upon calling `fork()`, child process gets a copy of the parent process' file descriptor table
- So if a process creates a pipe, and then forks, the child process will have a copy of the file descriptors to the pipe
- This allows the parent and child to communicate across the pipe

Child Communicating to Parent Over a Pipe

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

#define MSG_SIZE 13
char *msg = "hello, world\n";
int main(void)
{
  char buf[MSG_SIZE];
  int p[2];
  if (pipe(p) == -1) { perror("pipe"); exit(1); }
  if (fork() == 0) //Child writes
    write(p[1], msg, MSG_SIZE); // No error checking: Bad
  else { //Parent reads from pipe and prints
    read(p[0], buf, MSG_SIZE); // No error checking: Bad
    write(STDOUT_FILENO, buf, MSG_SIZE); // Bad
  }
  return 0;
```

Closing Unneeded File Descriptors

- Each process should close any file descriptors that it is not using, especially with pipes
- If a process calls `read()` on a pipe, but there is no data ready to be read, it will **stall forever**, unless:
  - New data arrives on the pipe; or
  - **All** file descriptors (across all processes) referring to the write end of the pipe have been closed
- Again: Read and understand `man 7 pipe`

Signals

- Signals are unexpected, asynchronous events that can happen at any time
- Unless you make special arrangements, most signals terminate your process

Signals You Already Know

- Signals are another basic form of Inter-Process Communication
- You have already been sending signals through the shell, e.g., via the `ctrl+c` and `ctrl+z` key combinations

Question

When you hit `ctrl+c` to terminate a program, what really happens?

Answer

The terminal sends the `SIGINT` signal to the process.

By default, `SIGINT` (“Interrupt from keyboard”) terminates the process.

Similarly, `ctrl+z` triggers a `SIGTSTP` signal (“Stop typed at terminal”)

Other Signals

The kernel sends several other signals to terminate processes when a program misbehaves:

- `SIGSEGV` Invalid memory reference
- `SIGFPE` Floating-point exception
- `SIGILL` Illegal instruction

There are also the user-defined signals `SIGUSR1` and `SIGUSR2` that we can use for our own purposes. By default they terminate the process.

Sending Signals From The Shell

To send a signal `SIGNAME` to one or more processes:

```
$ kill -SIGNAME pid
```

For example:

```
$ kill -SIGINT 11248
```

```
$ kill -SIGKILL 11248
```

The `SIGKILL` and `SIGSTOP` signals cannot be caught, blocked, or ignored (see `man 7 signal`).

Signal Handling

There are three options for handling signals:

1. Write a signal handler function, which will be called automatically upon receipt of a signal.
2. Ignore the signal (i.e., the signal does nothing to your process).
3. Use the default action.

Changing the Default Action

Two options for changing the default signal action:

1. The `sigaction()` system call
2. The `signal()` C standard library function

`signal()` is cross-platform but more limited in scope, whereas `sigaction()` is more flexible but found only on POSIX.1-compliant systems.

More in the GNU C library manual

Signal example

```c
#include <stdio.h>
#include <signal.h>

void handle_interrupt(int sig){
    printf("caught signal code %d\n", sig);
}

int main(){

    signal(SIGINT, handle_interrupt);
    // ....

}
```

Sending A Signal From One Process to Another

One process can send a signal to another process using the misleadingly-named `kill()` system call.

Limitations of Signals: Information

- Signals convey no information, aside from what type of signal (e.g., `SIGINT`, `SIGUSR1`) it is
- Generally only used to indicate abnormal conditions: Not for data exchange

Limitations of Signals: Queuing

- Multiple instances of the same signal do not queue
- If signal `X` is sent while a previously-sent signal `X` is pending, then the second `X` is lost
- Example, if your process receives a `SIGCHLD` (Child stopped or terminated), it may be that only one child process has terminated, or that *multiple* child processes have terminated

Writing to a Broken Pipe

- `SIGPIPE` is sent to a process that tries to write to a pipe or to a *socket* that does not have any readers

Async Signal Safety

From `man 7 signal-safety`:

> *Suppose a program is in the middle of a call to a `stdio` function such as `printf` where the buffer and associated variables have been partially updated. If, at that moment, the program is interrupted by a signal handler that also calls `printf`, the second call to `printf` will operate on inconsistent data, with unpredictable results.*

Extra Slides

The sigaction() system call

sigaction is both the name of the system call, and the struct that it takes as the 2nd and 3rd arguments:

```
int sigaction(int sig,
    const struct sigaction *act,
    struct sigaction *oldact);
```

What is a Signal Action?

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int      sa_flags;
    void     (*sa_restorer)(void);
};
```

`sa_handler` can be set to `SIG_IGN` (ignore), `SIG_DFL` (default action), or the address of a handler function (see `man 2 sigaction`).

```c
/* EXAMPLE */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int i = 0;
void handler(int signo) {
  fprintf(stderr, "Sig %d; total %d.\n", signo, ++i);
}

int main(void) {
  struct sigaction newact;
  sigemptyset(&newact.sa_mask);
  newact.sa_flags = 0;
  newact.sa_handler = handler;
  if (sigaction(SIGINT, &newact, NULL) == -1) exit(1);
  if (sigaction(SIGTSTP, &newact, NULL) == -1) exit(1);

  for(;;); //Infinite loop
}
```

Signal Sets

A signal set (`sigset_t`) is a *bit vector* that specifies the set of signals to block; operate on them using the following standard library functions:

1. `int sigemptyset(sigset_t *set);`
2. `int sigfillset(sigset_t *set);`
3. `int sigaddset(sigset_t *set, int signo);`
4. `int sigdelset(sigset_t *set, int signo);`
5. `int sigismember(const sigset_t *set, int signo);`

See `man 3 sigsetops` for usage.

Note: Also recall last week's discussion on bitwise operators.

Masking Signals During Program Execution

- As our program may receive signals spontaneously at any time, we may need to block some signals from being delivered at an inopportune moment (e.g., writing to disk).
- This *temporary* block is different from ignoring a signal entirely.
- Use the `sigprocmask()` system call to examine or change the set of blocked signals, via a *mask* (i.e., a bit vector representing a set of signals)

Types and Portability

What is a `sigset_t` anyway? Good opportunity to demonstrate the primary need for `typedef`: Allowing us to write *portable code*

From `x86_64-linux-gnu/bits/types/sigset_t.h`:

```
typedef __sigset_t sigset_t;
```

From `x86_64-linux-gnu/bits/types/__sigset_t.h`:

```
#define _SIGSET_NWORDS (1024 / (8 * sizeof (unsigned long i
typedef struct
{
    unsigned long int __val[_SIGSET_NWORDS];
} __sigset_t;
```

Masking Signals: Example

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigprocmask(SIG_BLOCK, &set, &oldset);
/*... Critical operation ...*/
sigprocmask(SIG_SETMASK, &oldset, NULL);
```

Masking Signals During Signal Handler Execution

From `man 2 sigaction`:

> *sa_mask specifies a mask of signals which should be blocked (i.e., added to the signal mask of the thread in which the signal handler is invoked) during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER flag is used.*

Pending Signals

- A blocked signal will be delivered if it is later unblocked
- Between the time when it is generated and when it is delivered, a signal is said to be *pending* (see `man 7 signal`)
- The `sigpending` system call allows you to examine the currently-pending signals

Signal Safety

- An *async-signal-safe* function is one that can be safely called from within a signal handler.

None of the `stdio` functions are async-signal-safe.

- Yes: If we give you any examples that use `stdio` functions in a signal handler, the examples may not always run properly
- A safe technique is to set *global variable* flags from the signal handler and call the `stdio` functions from your “main” program code

Efficiently Pausing a Program

Using an empty loop is a very inefficient way to await the arrival of a signal. This is like running in place rather than sitting down. We may instead use the `pause` system call to suspend execution until a handled signal arrives.

Caution: `pause` can result in “missing” a signal.

Efficiently Pausing a Program: Example

In the following example, we write a program that:

1. Blocks `SIGINT` while doing some important work
2. Unblocks `SIGINT`
3. Waits for a `SIGINT` to be sent before terminating

Any `SIGINT` sent during the important work should "count" (i.e., terminate the program after `SIGINT` is unblocked)

Note that signals can still sneak in after the unblock and before the `pause`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>

void handler(int signo) {
  fprintf(stderr, "Caught!\n");
}
```

```c
int main(void) {
  time_t startTime;
  sigset_t mask, oldmask;
  struct sigaction newact;
  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  if (sigprocmask(SIG_BLOCK, &mask, &oldmask) == -1)
    exit(1);
  sigemptyset(&newact.sa_mask); newact.sa_flags = 0;
  newact.sa_handler = handler;
  if (sigaction(SIGINT, &newact, NULL) == -1)
    exit(1);
  fprintf(stderr, "SIGINT blocked. Doing important work\n"
  startTime = time(NULL);
  while (time(NULL) < startTime + 10);

  fprintf(stderr, "Now unblocking SIGINT\n");
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1)
    exit(1);
  pause();
  fprintf(stderr, "Got the SIGINT. Exiting...\n");
  return 0;
}
```

More on Typedefs

The Linux kernel coding style should help further clarify the purpose of `typedefs` such as `sigset_t` that are used in the Linux system call interfaces.

In short: To present an abstraction for types that should only be accessed/manipulated using a specific set of functions (e.g., like how a `sigset_t` should only be modified/accessed via the `sigsetops` functions).