Week 3 - Friday

COMP 3400

Last time

- What did we talk about last time?
- Files
    - Opening
    - Closing
    - Reading
    - Writing
    - Polling

Questions?

Assignment 2

Project 1

Files

File metadata

- The data in the file is the sequence of bytes it contains
- The **metadata** of a file gives information about the file itself
  - Obscure OS stuff like inode number and hard links to the file
  - User ID of the owner
  - Group ID of the owner
  - Device type
  - File size
- This information can be stored in a **`struct stat`** and retrieved with:
  - **`fstat()`** Gets information from a file descriptor
  - **`stat()`** Gets information from a path

Interpreting metadata

- The following shows some fields in **`struct stat`**
- The **`st_mode`** field is a bitwise OR of permissions and other information from the table on the right

```
struct stat {
  dev_t     st_dev;     // device of inode
  ino_t     st_ino;     // inode number
  mode_t    st_mode;    // protection mode
  nlink_t   st_nlink;   // hard links to file
  uid_t     st_uid;     // user ID of owner
  gid_t     st_gid;     // group ID of owner
  dev_t     st_rdev;    // device type
  off_t     st_size;    // file size in bytes
  // Other fields depending on OS ...
};
```

Name	Description
`S_IFIFO`	Named pipe (IPC)
`S_IFCHR`	Character device (terminal)
`S_IFDIR`	Directory file type
`S_IFBLK`	Block device (disk drive)
`S_IFREG`	Regular file type
`S_IFLNK`	Symbolic link
`S_IFSOCK`	Socket (IPC, networks)

Example getting file metadata

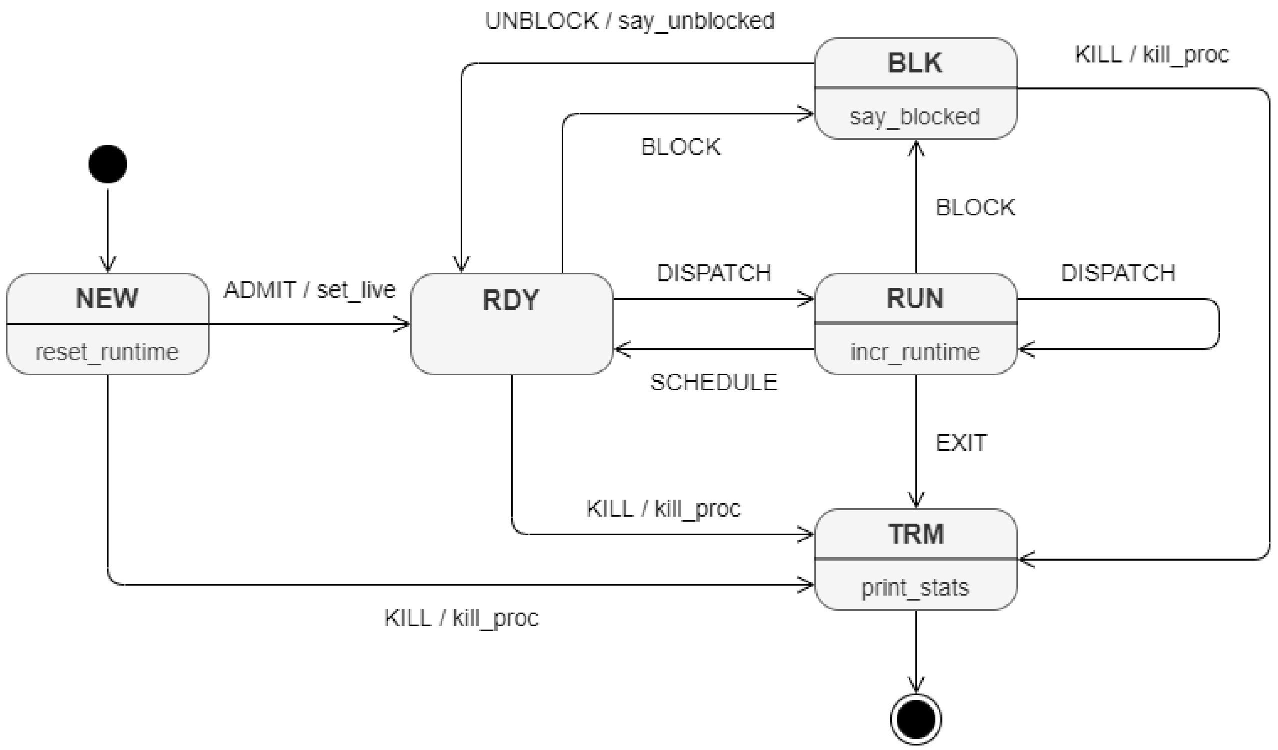
- The following code finds out how big a file (stored with file descriptor **`fd`**) is in bytes:

```c
struct stat metadata;
fstat (fd, &metadata);
printf ("File size: %lld bytes\n",
        (long long)metadata.st_size);
```

Events and Signals

Events

- Processes are created, run, and are eventually destroyed
- As shown in Assignment 2, processes can also:
  - Go into a blocked state, waiting for I/O
  - Be suspended, which means it doesn't get scheduled
- We can cause an event to happen to a process by sending it a **signal**



Command line signals

- You can send signals to processes from the command line
  - **`Ctrl-C`**: **`SIGINT`** (interrupt)
  - **`Ctrl-Z`**: **`SIGTSTP`** (terminal stop, usually suspends)
- Signals often result in the process being killed
- Perhaps for that reason, the **`kill`** command is used to send arbitrary signals (not just killing ones)
  - Flag gives the kind of signal
  - Then specify the PID of the process

```
> kill –KILL 8382
```

Common signals

- When using the kill command, the flag can either be the name of the signal (**`-KILL`**) or its number (**`-9`**)
- Here are some common signals:

Name	Number	Description
`SIGINT`	2	Interrupts the process, generally killing it. Sent with **`Ctrl-C`**.
`SIGKILL`	9	Kills the process. Cannot be ignored or overwritten.
`SIGSEGV`	11	Sent to a process when it has a segmentation fault.
`SIGCHLD`	18	Sent to a parent when a child process finishes. Used by **`wait()`**.
`SIGSTOP`	23	Suspends the process. Cannot be ignored or overwritten.
`SIGTSTP`	24	Suspends the process. Sent with **`Ctrl-Z`**.
`SIGCONT`	25	Resumes a suspended process.

Details on signals

- Some signals are similar
  - **`SIGINT`** and **`SIGKILL`** both kill the process
  - **`SIGSTOP`** and **`SIGTSTP`** both suspend the process
- Some of these signals can be overridden to do different things (and some can't)
- Have you ever meant to put the **`&`** down when you run **`gedit`** or another GUI program?
  - You can suspend the program by typing **`Ctrl-Z`**, then run **`bg`** to move it to the background
- Normally, **`SIGSEGV`** causes a program to print an error message and die
  - It's possible to make a custom signal handler to do something different
  - Debuggers like **`gdb`** do this

Sending signals in a program

- Just as you can use the kill command from the command line, you can also call the **`kill()`** function to send a signal to another process
- The function takes two parameters:
  - PID of the process to kill
  - **`int`** value giving the signal, usually a named constant

```
kill (pid, SIGSTOP); // Suspends process with pid
```

- You can usually only kill processes that you own
  - Unless you're a superuser (like root)

Example of `kill()` function

- Below, a parent forks a child
- The child goes into an infinite loop
- Then, the parent kills the child

```
pid_t child_pid = fork ();
if (child_pid < 0)
  exit (1); // exit if fork failed

if (child_pid == 0)
  while (1) ; // child loops

sleep (1); // parent sleeps for 1 second
kill (child_pid, SIGKILL); // parent kills the child
```

Custom signal handlers

- Although signals have default actions for processes, *some* signals can be overridden
- A process can define what happens when, for example, it's interrupted
- First, you need a function that will get called when a particular signal happens
  - It must take an **`int`** (the signal) and return **`void`**
- Example that prints "I don't want to die!" and then exits

```
static void
handler(int signal)
{
  write(STDOUT_FILENO, "I don't want to die!\n", 21);
  exit(0);
}
```

Asynchronous signal safe

- Wouldn't it have been easier to call `printf()` in the previous signal handler example?
- Yes, but you should **not**
- Only asynchronous signal safe functions should be called in signal handlers
  - Or else the results are unpredictable!
- Functions that have static buffers inside of them (like `printf()` and `scanf()`) are not asynchronous signal safe
- For more information (and a list of functions In you can call):

```
> man signal-safety
```

- TLDR: Keep signal handlers short, don't call functions unless you're sure they're safe, and `printf()` and `scanf()` are **not** safe

Overriding the signal handler

- Once you've written the custom signal handler, you have to override it with the `sigaction()` function:

```
int sigaction(int signal, const struct sigaction *action,
struct sigaction *old);
```

- The **`action`** parameter is a **`struct sigaction`** with a function pointer to the new handler
- The old parameter is **`NULL`** unless you want to find out what the old signal handler was

Overriding example

- The following code overrides the **`SIGINT`** signal with the handler from a couple of slides back
- Then it goes into an infinite loop until someone interrupts it (like with **`Ctrl-C`**)

```
int
main (int argc, char *argv[])
{
  struct sigaction sa; // Struct we'll add the handler to
  memset(&sa, 0, sizeof(sa)); // Zero out the contents first
  sa.sa_handler = handler;

  // Override SIGINT handler
  if (sigaction (SIGINT, &sa, NULL) == -1)
    printf ("Failed to overwrite SIGINT.\n");

  printf ("Entering loop\n");
  while (1);  // Loop until signal
  return 0;
}
```

Reborn like a phoenix

- It's sort of cool that we can make a handler print something special before crashing the program
- But we can also do some code to handle the signal and then jump back to a safe location
  - Away from blocked I/O or an infinite loop
  - Somewhere that's been marked and is still on the stack
- To do that, we need two functions

```
// Set jump location
int sigsetjmp(sigjmp_buf context, int mask);

// Jump to location
int siglongjmp(sigjmp_buf context, int value);
```

Full example

```
sigjmp_buf context;

static void handler(int signal)
{
  write(STDOUT_FILENO, "I don't want to die!\n", 21);
  siglongjmp (context, 1); // Jumps to marked location with value 1 (insane!)
}

int main (int argc, char *argv[])
{
  struct sigaction sa;
  memset(&sa, 0, sizeof(sa)); sa.sa_handler = handler;

  if (sigaction (SIGINT, &sa, NULL) == -1)
    printf ("Failed to overwrite SIGINT.\n");

  if (sigsetjmp (context, 0))  // Marks location and returns 0 the first time
    printf ("Resuming execution\n");

  printf ("Entering loop\n");
  while (1);  // Loop until signal
  return 0;
}
```

Except, of course, there's more

- Signal handling can be tricky
- What happens when a signal is sent a second time?
- There are masks you can set in the **`struct sigaction`** that determine if your handler is used repeatedly
- The **`sigprocmask()`** function can also be used to change which signals are blocked, inside of your handler

Upcoming

Next time…

- Interprocess communication (IPC)

Reminders

- Finish Assignment 2
  - Due **tonight** by midnight!
- Keep working on Project 1
- Read sections 3.1 and 3.2