Operating Systems
COMS W4118
Reading Notes 2

Alexander Roth

2015 – 01 – 31

Advanced Programming in the Unix Environment Chapter 10: Signals

10.1 Introduction

- Signals are software interrupts.
- Signals provide a way of handling asynchronous events.

10.2 Signal Concepts

- Every signal has a name and all names begin with the three characters `SIG`.
- Signal names are all defined by positive integer constants in the header `<signal.h>`.
- No signal has a signal number of 0.
- It is considered bad form for the kernel to include header files meant for user-level applications, so information is placed in the kernel header and then included by the user-level header. Thus, we have `<sys/signal.h>` for Mac OSX.
- Numerous conditions can generate a signal:
  - Terminal-generated signals occur when users press certain terminal keys.
  - Hardware exceptions generate signals.
  - The `kill()` functions allow us to send signals to other processes.
  - Software conditions can generate signals when a process should be notified by various events.

- The *disposition* or *action* associated with the signal can be one of three things:
    1. Ignore the signal. (Note: `SIGKILL` and `SIGSTOP` can never be ignored.)
    2. Catch the signal. (Note: `SIGKILL` and `SIGSTOP` can never be caught.)
    3. Let the default action apply.

10.3 `signal` function

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```
Returns: previous disposition of signal if OK, `SIG_ERR` on error

- The *signo* argument is just the name of the signal.
- The value of *func* is
    1. the constant `SIG_IGN`
    2. the constant `SIG_DFL`
    3. the address of a function to be called when the signal occurs
- `SIG_IGN` tells the system to ignore the signal.
- `SIG_DFL` tells the system to follow the default action associated with the signal.
- When the address of a function is given, we are arranging a "catch" within the system.
- The function requires two arguments
    1. The first argument, *signo*, is an integer.
    2. The second argument is a pointer to a function that takes a single interger argument and returns nothing.

Program Start-Up

- When a program is executed, the status of all signals is either default or ignore.
- We are not able to determine the current disposition of a signal without changing the disposition of the signal.

Process Creation

- When a process calls `fork`, the child inherits the parent's signal dispositions.

10.4 Unreliable Signals

- In earlier versions of the UNIX system, signals were unreliable; this means that signals could get lost: a signal could occur and the process would never know about it.
- No way to remember if a signal occured without having to do anything about it. Could only ignore signal or act on it.

10.5 Interrupted System Calls

- If a process caught a signal while the process was blocked in a "slow" system call, the acll was interrupted.
- The system returned an error and `errno` was set to `EINTR`.
- The slow system calls are those that can block forever.
  - Reads that can block the caller forever if data isn't present with certain file types.
  - Write that can block the caller forever if the data can't be accepted immediately by these same file types.
  - Opens on certain file types that block the caller until some condition occurs.
  - The `pause` function and the `wait` function.
  - Certain `ioctl` operations
  - Some of the interprocess communication functions
- `wait` and `waitpid` are always interrupted when a signal is caught.

10.6 Reentrant Functions

- When a signal that is being caught is handled by a process, the normal sequence of instructions begin executed by the process is temporarily interrupted by the signal handler.
- If the signal handler returns, then the normal sequence of instructions that the process was executing when the signal was caught continues executing.
- The Single UNIX Specification specifies the functions that are guaranteed to be safe to call from within a signal handler.
- These functions are reetrant and called *async-signal safe*. They block any signals during operation if delivery of a signal might cause inconsistencies.
- As a general rule, when calling fucntions from a signal handler, we should save and restore `errno`.
- If we call a nonreentrant function from a signal handler, the results are unpredictable.

Advanced Programming in the Unix Environment Chapter 3: File I/O

3.1 Introduction

- Most file I/O on a UNIX system can be performed using only five functions:
  1. `open`
  2. `read`
  3. `write`
  4. `lseek`
  5. `close`
- *Unbuffered I/O* means that each `read` or `write` invokes a system call in the kernel.

3.2 File Descriptors

- A file descriptor is a non-negative integer, used to reference a file by the kernel.
- File descriptors 0, 1, and 2 are Standard Input, Standard Output, and Standard Error respectively.

3.3 `open` and `openat` Functions

```
#include <fctnl.h>
int open(const char *path, int oflag, .../* mode_t mode */ );
int openat(int fd, const char *path, int oflag, .../* mode_t mode
*/);
```

Both return: file descriptors if OK, -1 on error

- The *path* parameter is the name of the file to open or create.
- The function has a multitude of options, which are specified by the *oflag* argument.
- The file descriptor returned by `open` and `openat` is guaranteed to be the lowest-numbered unused descriptor.
- The *fd* parameter for `openat` has three use cases:
  1. The *path* parameter specifies an absolute pathname. *fd* is ignored.
  2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated.

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value `AT_FDCWD`.

- `openat` addresses two problems
  1. It gives threads a way to use relative pathnames to open files in directories other than the current working directory.
  2. It provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors
- TOCTTOU errors is the idea that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call.

3.4 `creat` Function

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```
Returns: file descriptor opened for write-only if OK, -1 on error

- `creat` is equivalent to `open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);`
- `creat` only opens a file for writing.

3.5 `close` Function

```
#include <unistd.h>
int close(int fd);
```
Returns: 0 if OK, -1 on error

- When a process terminates, all of its open files are closed automatically by the kernel.

3.6 `lseek` Function

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```
Returns: new file offset if OK, -1 on error

- The interpretation of the *offset* depends on the value of the *whence* argument.
  - If *whence* is `SEEK_SET`, the file's offset is set to *offset* bytes from the begging of the file.
  - If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*.
  - If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*.

- A file's current offset must normally be a non-negative integer. However, certain devices could allow negative offsets.
- Thus, the return value from `lseek` should be tested against -1 if there is an error.
- `lseek` only records the current file offset within the kernel – it does not cause any I/O to take place.
- If the offset is greater than the file's current size, the next `write` call is said to *extend* the file. This is also known as creating a hole in a file and is allowed.
- A hole in a file isn't required to have storage backing it on disk.

3.7 `read` Function

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```
Returns: number of bytes read, 0 if end of file, -1 on error

- There are several cases in which the number of bytes actually read is less than the amount requested:
  - When reading from a regular file, if the end of the file is reached before the requested number of bytes have been read.
  - When reading from a terminal device.
  - When reading from a network.
  - When reading from a pipe or FIFO.
  - When reading from a record-oriented device.
  - When interrupted by a signal and a partial amount of data has already been read.
- The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

3.8 `write` Function

```
#include<unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
```
Returns: number of bytes written if OK, -1 on error

- A common cause for a `write` error is either filling up a disk or exceeding the file size limit for a given process.
- The write operation starts at the file's current offset.

- If `O_APPEND` option was specified when the file was opened, the file's offset is set to the current end of the file before each write operation.
- After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

- When sequential reads are detected, the system tries to read in more data than an application requests, assuming that the application will read it shortly.

3.10 File Sharing

- The UNIX System supports the sharing of open files among different processes.
- The kernel uses three data structures to represent an open file, and the relationships among them determine the effect on process has on another with regard to file sharing.
  1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors (a vector with one entry per descriptor). Associated with each descriptor are
     (a) The file descriptor flags
     (b) A pointer to a file table entry
  2. The kernel maintains a file table for all open files. Each file table entry contains
     (a) The file status flags for the file.
     (b) The current file offset
     (c) A pointer to the v-node table entry for the file.
  3. Each open file has a v-node structure that contains information about the type of file and pointers to functions that operate on the file.
- Each process that opens a file gets its own file table entry, but only a single v-node table entry is required for a given file. This is so each process has its own current offset for the file.

3.11 Atomic Operations

Appending to a File

- Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel might temporarily suspend the process between the two function calls.

`pread` and `pwrite` Functions

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
```
Returns: number of bytes read, 0 if end of file, -1 on error
```
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```
Returns: number of bytes written if OK, -1 on error

- Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.
  - There is no way to interrupt the two operations that occur when we call `pread`.
  - The current file offset is not updated.
- Calling `pwrite` is equivalent to calling `lseek` followed by a call to `write`.

Creating a File

- In general, the term *atomic operation* refers to an operation that might be composed of multiple steps.
- If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure).
- It must not be possible for only a subset of the steps to be performed.

3.12 `dup` and `dup2` Functions

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```
Both return: new file descriptor if OK, -1 on error

- The new file descriptor returned by `dup` is guaranteed to be the lowest-numbered available file descriptor.
- With `dup2`, the value of the new file descriptor is specified with the `dup2` argument.
- The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument.

3.13 `sync`, `fsync`, and `fdatasynch` Functions

- When we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time (*delayed write*).

- The kernel eventually writes all the delayed-write blocks to disk.
- To ensure consistency of the file system with the buffers, we use the `sync`, `fsync`, and `fdatasync` functions.

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
```
Returns: 0 if OK, -1 on error

```
void sync(void);
```

- The function `sync` is called periodically from a system daemon, often called `update`.
- The function `fsync` refers only to a single file, specified by the file descriptor *fd*, and waits for the disk writes to complete before returning.
- The `fdatasync` function is similar to `fsync`, but it affects only the data portions of a file.

Advanced Programming in the Unix Environment Chapter 8: Process Control

8.2 Process Identifiers

- Every process has a unique process ID, a non-negative integer.
- As processes terminate, their IDs become candidates for reuse after some time has passed.
- Process ID 0 is usually the scheduler process and is often known as the *swapper*.
- Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure.
- `init` reads the system-dependent initialization files and brings the system to a certain state.
- `init` never dies and is a normal user process, not a system process like the swapper.
- The following functions return identifiers for processes

```
#include <unistd.h>
pid_t getpid(void);
```
Returns: process ID of calling process

```
pid_t getppid(void);
```
Returns: parent process ID of calling process

```
uid_t getuid(void);
```
Returns: real user ID of calling process

```
uid_t geteuid(void);
```
Returns: effective user ID of calling process

```
gid_t getgid(void);
```
Returns: real group ID of calling process

```
gid_t getegid(void);
```
Returns: effective group ID of calling process

- Note that none of these functions has an error return.

8.3 `fork` Function

- An existing process can create a new one by calling the `fork` function.

```
#include <unistd.h>
pid_t fork(void);
```
Returns: 0 in child, process ID of child in parent

- The new process created by `fork` is called the *child process*.
- The function is called once but returns twice. The return value in the child is 0, but the return value in the parent is the process ID of the new child.
- A process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- `fork` returns 0 to the child because a process can only have a single parent.
- The child can always call `getppid` to obtain the process ID of its parent.
- The child is a copy of the parent.
- In general, we never know whether the child starts executing before the parent or vice versa (this is determined by the scheduling algorithm used in the kernel).

File Sharing

- One characteristic of `fork` is that al file descriptors that are open in the parent are duplicated in the child.
- It is important that the parent and the child share the same file offset.
- If both parent and child write to the same descriptor, without any form of synchronization, their output will be intermixed.
- Besides the open files, numerous other properties of the parent are inherited by the child:
  - Real user ID, real group ID, effective user ID, and effective group ID
  - Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - The set-user-ID and set-group-ID flags
  - Current working directory
  - Root directory
  - File mode creation mask
  - Signal mask and dispositions
  - The close-on-exec flag for any open file descriptors
  - Environment
  - Attached shared memory segments
  - Memory mappings
  - Resource limits
- The differences between the parent and child are
  - The return values from `fork` are different
  - The process IDs are different
  - The two processes have different parent process IDs
  - The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0.
  - File locks set by the parent are not inherited by the child.
  - Pending alarms are cleared for the child.
  - The set of pending signals for the child is set to the empty set.
- The two main reasons for `fork` to fail are
  1. if too many processes are already in the system

2. if the total number of processes for this real user ID exceeds the system's limit

- When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time.
- When a process wants to execute a different program.
- A `fork` followed by an `exec` is called a *spawn*.

8.4 `vfork` Function

- The `vfork` function creates the new process without copying the address space of the parent into the child. The child runs in the address space of the parent until it calls either `exec` or `exit`.
- `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`.

8.5 `exit` Functions

- A process can terminate normally in five ways:
  1. Executing a `return` from the `main` function.
  2. Calling the `exit` function.
  3. Calling the `_exit` or `_Exit` function.
  4. Executing a `return` from the start routine of the last thread in the process.
  5. Calling the `pthread_exit` function from the last thread in the process.
- The three forms of abnormal termination are as follows:
  1. Calling `abort`
  2. When the process receives certain signals.
  3. The last thread responds to a cancellation request.
- When a parent terminates before the child process terminates, the `init` process becomes the parent process. In such a case, we say that the process has been inherited by `init`.
- This guarantees that every process has a parent.
- The kernel keeps a small amount of information for every terminating proces so that the information is available when the parent of the terminating processs calls `wait` or `waitpid`.

- This information consists of the process ID, the terminal status of the process, and the amount of CPU time taken by the process.
- A process that has terminated, but whose parent has not yet waited for it, it called a *zombie*.

8.6 `wait` and `waitpid` Functions

- When a process terminates, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent.
- The parent can ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```
Both return: process ID if OK, 0, or -1 on error.

- The `wait` function can block the caller until a child process terminates; `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.
- If the caller blocks and has multiple children, `wait` returns when one terminates.
- The argument *statlo* is a pointer to an integer.
- The `waitpid` function provides three features that aren't provided by the `wait` function:
  1. The `waitpid` function lets us wait for one particular process.
  2. The `waitpid` function provides a nonblocking version of `wait`.
  3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

8.7 `waitid` Function

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```
Returns: 0 if OK, -1 on error

- `waitid` allows a process to specify which children to wait for.
- The *id* parameter is interpreted based on teh value of *idtype*.
- The *options* argument is bitwise OR of the flags that are used to indicate which state changes the caller is interested in.

- The *infop* argument is a pointer to a `siginfo` structure, which contains detailed information about the signal generated that caused the state change in the child process.

8.8 `wait3` and `wait4` Functions

```
#include <sys/types.h> #include <sys/wait.h> #include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or -1 on error.

- The only feature provided by these two functions that isn't provided by the `wait`, `waitid`, and `waitpid` functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.
- Resource information includes such statistics as the amount of user CPU time, amount of system CPU time, number of page faults, number of signals received, etc.

8.9 Race Conditions

- A race condition occurs when multiple processes are trying to do somethign with shared data and the final outcome depends on the order in which the processes run.
- *Polling* wastes CPU time because the caller is awakened every second to test the conditions.
- It is necessary to have processes wait for other processes to complete and to have processes who complete to alert waiting processes.

8.10 `exec` Functions

- When a process calls one of the `exec` functions, that process is completely replaced by the new program, and the new program starts executing at its `main` function.
- When a *filename* argument is specified,
  - If *filename* contains a slash, it is taken as a pathname.
  - Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.
- The `PATH` variable contains a list of directories, called path prefixes, that are separated by colons.

- If either `execlp` or `execvp` finds an executable file using a path prefix, but the file isn't machine executable, the function assumes that the file is a shell script and tries to invoke `/bin/bsh` with the *filename* as input to the shell.
- `l` stands for list and `v` stands for vector. Certain `exec` functions require each of the command-line arguments to the new program to be specified as separate arguments. Others require that we build an array of pointers to the arguments, and the address of this array is the argument to these functions.
- The list `exec` functions require the arguments given to end with a null pointer.
- The letter `p` means that the function takes a *filename* argument and uses the `PATH` environment variable to find the executable file.
- `e` means that the function takes an *envp[]* array instead of using the current environment.
- THe new program after an `exec` inherits additional properties from the calling process:
  - Process ID and parent process ID
  - Real user ID and real group ID
  - Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - Time left until alarm clock
  - Current working directory
  - Root directory
  - File mode creation mask
  - File locks
  - Process signal mask
  - Pending signals
  - Resource limits
  - Nice value
  - Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`