Operating Systems
COMS W4118
Reading Notes 3

Alexander Roth

2015 – 02 – 04

Advanced Programming in the Unix Environment Chapter 7: Process Environment

7.6 Memory Lay out of a C Program

- A C program has been composed of the following pieces:
  - Text segment, consisting of the machine instructions that the CPU executes.
  - Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program.
  - Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol."
  - Stack, where automatic variables are stored, along with information that is saved each time a function is called.
  - Heap, where dynamic memory allocation usually takes place.
- The only portions of the program that need to be saved in the program file are the text segment and the initialized data.

7.7 Shared Libraries

- Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference.
- Shared libraries reduce the size of each executable file, but add some run-time overhead.
- Library functions can be replaced with new versions without having to relink edit every program.

7.8 Memory Allocation

- There are three functions for memory allocation:
  - `malloc`, which allocates a specified number of bytes of memory.
  - `calloc`, which allocates space for a specified number of objects of a specified size.
  - `realloc`, which increases or decreases the size of a previously allocated area.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```
All three return: non-null pointer if OK, `NULL` on error

```
void free(void *ptr);
```

- The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object.
- The function `free` causes the space pointed to by *ptr* to be deallocated.
- The allocation routines are usually implemented with the `sbrk(2)` system call. This system call expands (or contracts) the heap of the process.
- Most versions of `malloc` and `free` never decrease their memory size. Freed space is kept in the `malloc` pool.
- Most implementations allocate more space than requested and use the additional space for record keeping – the size of the block, a pointer to the next allocated block, and the like.

Advanced Programming in the Unix Environment Chapter 14: Advacned I/O

14.8 Memory-Mapped I/O

- Memory-mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t
off);
```
Returns: starting address of mapped region if OK, `MAP_FAILED` on error.

- The *addr* arguments lets us specify the address where we want the mapped region to start.
- The *fd* argument is the file descriptor specifying the file that is to be mapped.
- The *len* argument is the number of bytes to map.
- The *off* is the starting offset in the file of the bytes to map.
- The *prot* argument specifies the protection of the mapped region.
- Each implementation has additional `MAP_xxx` flag values, which are specific to that implementation.
- The `SIGSEGV` signal is normally used to indicate that we tried to access memory that is not available to us, such as read-only memory.
- The `SIGBUS` signal can be generated if we access a portion of the mapped region that does not make sense at the time of the access.
- A memory-mapped region is inherited by a chile across a `fork` but it is not inherited by the new program across an `exec`.

Advanced Programming in the Unix Environment Chapter 15: Interprocess Communication

15.2 Pipes

- Pipes have two limitations:
  1. Historically, they have been half duplex
  2. Pipes can be used only between processes that have a common ancestor.
- The shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

```
#include <unistd.h>
int pipe(int fd[2 ]);
```
Returns: 0 if OK, -1 on error

- Two file descriptors are returned through the *fd* argument:

  fd[0] opened for reading

  fd[1] opened for writing.
- When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all teh data has been read.
2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated.

- When we're writing to a pipe, the constant `PIPE_BUF` specifies the kernel's pipe buffer size.

15.10 POSIX Semaphores

- Semaphores are useful tool in the prevention of race conditions.
- Semaphores are records of how many units of a particular resource are available, coupled with operations to *safely* adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available.

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ...  /* mode_t mode,
unsigned int value */);
```
Returns: Pointer to semaphore if OK, `SEM_FAILED` on error

- When using an existing named semaphore, we specify only two arguments: the name of the semaphore and a zero value for the *oflag* argument.
- When we specify the `O_CREAT` flag, we need to provide two additional arguments. The *mode* argument specifies who can access the semaphore. The *value* argument is used to specify the initial value for the semaphore when we create it.
- To promote portability, we must follow certain conventions when selecting a semaphore name.
  - The first character in the name should be a slash (`/`).
  - The name should contain no other slashes to avoid implementation-defined behavior.
  - The maximum length of the semaphore name is implementation defined.
- The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we operate on the semaphore.

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```
Returns: 0 if OK, -1 on error

- If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically.

```
#include <semaphore.h>
int sem_unlink(const char *name);
```
Returns: 0 if OK, -1 on error

- The `sem_unlink` function removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed.

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```
Both return: 0 if OK, -1 on error

- The `sem_wait` function will block if the semaphore count is 0.

```
#include <semaphore.h> #include <time.h>
int sem_timedwait(sem_t *restrict sem, const struct timespec *restrict
tsptr);
```
Returns: 0 if OK, -1 on error

- The *tsptr* arugment specifies the absolute time when we want to give up waiting for the semaphore.

- To increment the value of a semaphore, we call the `sem_post` function.

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```
Returns: 0 if OK, -1 on error

- To create an unnamed semaphore, we call the `sem_init` function.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, usigned int value);
```
Returns: 0 if OK, -1 on error

- The *pshared* argument indicates if we plan to use the semaphore with multiple processes.

- The *value* argument specifies the initial value of the semaphore.

- When we are done using the unnamed semaphore, we can discard it by calling the `sem_destory` function.

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```
Returns: 0 if OK, -1 on error

- We call the `sem_getvalue` function to retrieve the value of a semaphore.

```
#include <semaphore.h>
int sem_getvalue(sem_t *restrict sem, int *restrict valp);
```
Returns: 0 if OK, -1 on error

- On success, the integer pointed to by the *valp* argument will contain the value of the semaphore.