Operating Systems: Practice: Lesson 3

Sevak Amirkhanian

Change your .gitignore

```
CMakeLists.txt.user
CMakeCache.txt
CMakeFiles
CMakeScripts
Testing
Makefile
cmake_install.cmake
install_manifest.txt
compile_commands.json
CTestTestfile.cmake
_deps
build/
```

Useful commands: Part 1

Clean the build
make clean

Rebuild the project
make -B

Find files with the given name
find -type f -name my_file_name

Find directories with the given name
find -type d -name my_dir_name

Find text in the file / files
grep pattern file1 … fileN

Useful Commands: Part 2

Copy the file
cp -f source_path destination_path

Copy the directory
cp -rf source_path destination_path

Remove file
rm -f file_path

Remove directory
rm -rf dir_path

Remove files/directories with the name
find -type f -name file | xargs rm -f
find -type d -name dir | xargs rm -rf

Aliases

Alias is a shell provided feature which helps us to map a command with the given input to another command with much more convenient name. In its essence, alias is a mapping.

For example:
ll is an alias of *“ls -l”*

Useful Aliases

```
alias cpf='cp -f'
alias cpd='cp -rf'
alias rmf='rm -f'
alias rmd='rm -rf'
alias add='git add .'
alias cmm='git commit -m'
alias md='mkdir'
alias mf='touch'
alias h='history'
alias fh='history | grep'
```

How to effectively use command history?

Your shell saves all the history of your command executions. Using this history you can re-run the previously executed command in a much simpler way.

To find the command with the name use:
history | grep name

This will list commands with execution IDs:
ID1: command_1
....
IDN: command_N

To run command_1
!ID1

What is errno?

errno is defined by the ISO C standard to be a modifiable lvalue of type *int*, and must not be explicitly declared; *errno* may be a macro. *errno* is thread-local; setting it in one thread does not affect its value in any other thread.

How is errno changed and how can we use it?

errno is always set by the last system call. It is the error code of this last system call if any error occurred. To access errno-associated message we use perror(..):

```
void perror(const char *s);
```

Sync primitives can be shared.

Synchronization primitives in POSIX like mutexes, semaphores and conditional variables can be shared across different processes.

How to make POSIX mutex shared?

Whether mutex is shared or not, should be determined with the mutex attribute. In order to make mutex shareable, we need to use the following POSIX interface:

```
int pthread_mutexattr_setpshared(
pthread_mutexattr_t *attr,
int pshared);
```

`PTHREAD_PROCESS_SHARED` should be passed as pshared value if we want the mutex to be accessible from other processes.

Semaphores

Semaphore is a another synchronization primitive which have similarities with conditional variables.

Semaphore is basically an integer counter which has 2 operations:
P() // increments the counter
V() // decrements the counter

The value of the semaphore is the number of units of the resource that are currently available

POSIX semaphores

POSIX semaphores have 4 main operations:

sem_init(...)

sem_post(..)

sem_wait(..)

sem_destroy(...)

https://man7.org/linux/man-pages/man7/sem_overview.7.html

Binary semaphores

If the semaphore count is 1, semaphore is essentially a mutex which is why in literature mutex is sometimes called *binary semaphore*.

sem_wait() -> pthread_mutex_lcck()

sem_post() -> pthread_mutex_unlock()

Real-life examples of semaphores

Semaphores may be used in networking operating systems in routers for connection throttling.

Semaphores may also be used database drivers to restrict the amount of parallel connections, hence read-write operations.

Producer-Consumer Problem

The simple of description of the problem is:

We have a storage with limited storage with size N.

We have M producers who periodically produce a new item and insert into the storage if and only if there is a free space in the storage.

We have K consumers who periodically consume a new item from the storage if and only if there the storage is not empty.

Solution with semaphores

We will use 3 semaphores:

1 binary semaphore for lock

1 counting semaphore for full state

1 counting semaphore for empty state

Object Pool Pattern

The object pool pattern is a creational design pattern which uses previously allocated and initialized set of objects instead of initializing them on demand.

Real-life examples:
1. C# runtime called CLR pools all strings
2. ODBC drives pool database connections

Homework 3: Thread Pool

Threads are kernel objects and working with them has obvious performance penalties. Therefore pooling them seems a good idea.

You need to provide an interface executing multiple tasks on a finite set of kernel threads.

The interface can be seen in threadpool.h file.

Extra point:
Increase threadpool size if all threads are busy.

Thank you.