CS 153 Lab4 and 5

Kishore Kumar Pusukuri
Outline

Introduction
Why threads?
POSIX Threads
Synchronization
A “thread” is a straightforward concept: a single sequential flow of control.

In traditional operating systems, each process has an address space and a single thread of control.

Threads run in the same address space.

In a high-level language you normally program a thread using procedures, where the procedure calls follow the traditional stack discipline.

Within a single thread, there is at any instant a single point of execution.

Having “multiple threads” in a program means that at any instant the program has multiple points of execution, one in each of its threads.
Why concurrency?

- In many applications multiple activities are going on at once. Some of these may block from time to time.
- By decomposing such an application into multiple sequential threads that run quasi-parallel, the programming model becomes simple.
- Threads are lighter weight than processes, they are easier (i.e. faster) to create and destroy than processes.
- In many systems, creating a thread goes 10-100 times faster than creating a process.
Introduction

Why threads?
POSIX Threads
Synchronization

word processor
The programmer can mostly view the threads as executing simultaneously.

The programmer is required to decide when and where to create multiple threads.

Additionally, the programmer must occasionally be aware that the computer might not in fact execute all his threads simultaneously.

Having the threads execute within a “single address space” means that the computer’s addressing hardware is configured so as to permit the threads to read and write the same memory locations.
## Processes vs Threads

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
Each thread executes on a separate call stack with its own separate local variables, but the off-stack (global) variables are shared among all the threads of the program.

The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.

What about protection between threads?
To make it possible to write portable threaded programs, IEEE has defined a standard for threads in IEEE standard 1003.1c. This thread package it defines is called Pthreads. The standard defined 60 function calls.

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread_create</td>
<td>Create a new thread</td>
</tr>
<tr>
<td>Pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>Pthread_join</td>
<td>Wait for a specific thread to exit</td>
</tr>
<tr>
<td>Pthread_yield</td>
<td>Release the CPU to let another thread run</td>
</tr>
<tr>
<td>Pthread_attr_init</td>
<td>Create and initialize a thread’s attribute structure</td>
</tr>
<tr>
<td>Pthread_attr_destroy</td>
<td>Remove a thread’s attribute structure</td>
</tr>
</tbody>
</table>
int pthread_create(
    pthread_t * thread,
    const pthread_attr_t * attr,
    void * (*start_routine)(void *),
    void *arg);

returns the thread id.

attr - Set to NULL if default thread attributes are used. Thread attributes: scheduling parameter, stack address etc.

void * (*start_routine) - pointer to the function to be threaded.

*arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

void pthread_exit(void *retval);

retval - Return value of thread.
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d, tid\);  
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d, i\);  
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)&));

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d, status\);  
            exit(-1);
        }
    }  
    exit(NULL);
}
-lpthread

When a thread is created, it prints a one-line message announcing itself, then it exits.

The order in which the various messages are interleaved is nondeterminable, and may vary on consecutive runs of the program.

Study prog1.c and prog2.c
mutex

> Why we need mutual exclusion?
> A mutex, which stands for mutual exclusion is the most basic form of synchronization.
> A mutex is used to protect a critical region, to make certain that only one thread at a time executes the code within the region.
> Since only one thread at a time can lock a given mutex, this guarantees that only one thread at a time can be executing the instructions within a critical region.
lock_the_mutex(...);
critical region
unlock_the_mutex(...);
Study prog3.c and prog4.c
Introduction
Why threads?
POSIX Threads
Synchronization