CS342: Operating Systems Lab

Department of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
North Guwahati, Assam 781 039

Training Lesson

OS Lessons: Threads, Thread lifecycle, Scheduling, Run-time stack, User Programs (processes)
Rating: Easy
Last update: 04 June 2018

This document teaches the basic topics to the students of CS342 that are needed by them to begin working on CS342 exercises. These topics will be covered in detail in their CS341 and CS346 classes later. The coverage here is very simple and only enough to understand the exercises.

Introduction

An operating system is a program just like any C program! It has function `main()`. The translated program gets loaded into a computer memory and runs just like a factorial program may run. When we give command `factorial 10` to a Linux shell, the program code is inserted in the computer memory and control passed to its function `main()`. Each function in a program expects its parameters to be available in the topmost frame (activation record) in an agreed run-time stack.

Function `main()` expects no different. For the command to run factorial program discussed above, function `main()` expects the stack frame to have two parameters. One telling the count of the arguments (`int argc`) and other pointing to an array of pointers (`char *argv[]`) to these argument strings.

However, unlike program `factorial` that will be loaded and supported by a sophisticated operating system like Linux, Linux itself may be loaded and given control by a primitive bootstrap code. PintOS is a small but sophisticated operating system. It too gets loaded on its computer; albeit a simulated computer to run. It too gets its initial control through its function `main()`. You can locate this function in file `threads/init.c`

However, PintOS (and Linux) are programs that must aid loading and running of the other programs. The operating systems must aid these programs to run and also for reasons of efficient resource utilization let many active programs run concurrently.

Each activity chain defined by execution of a relevant code in PintOS is a thread. A relevant piece of code may a program or a utility code within PintOS kernel.

All important information for a thread is maintained by PintOS in `struct thread` that can be viewed in file `threads/thread.h`. Since many activities are running concurrently there are many concurrent threads in an operating system. However, there is only one processor. Only one thread can run (progress) at any given point in time.

This brings us to a scheduler. Scheduler is a collection of data-structures and functions (algorithms) within an operating system. Data-structures record all threads in the system. From time to time, scheduling events occur that cause the scheduler code to be run. When the code runs it can select a thread from its data-structures to run next. Thread that was running is inserted into the data-structures to run again at a later time. You will be exploring PintOS code to find these data-structures and functions as you do your exercises under project Threads. And add new features and functionality to them.

As many threads are running concurrently, one can expect them to be using system resources, for example, in memory data-structures and external files and devices at the same time. Inconsistent use of these can cause problems. Thus, you will also find in PintOS code synchronization primitives that prevent multiple threads from entering critical phases simultaneously. Only one thread must perform a task at a time that is prone to problems.

Thread Life-Cycle

A thread (activity) may be divided into 5 easily identifiable states. Two of them are obvious – being created and being removed. To create a thread, the OS kernel must get memory space for `struct thread` to be the prime representative for the thread. This `struct` is initialized and then added into the scheduler's data-structures. Once there, it will get access to the processor time and can do useful activities. The initial activities include location of the code to run and creation of initial stack-frames.

There are three stacks located in `struct thread`! See function `thread_create()`. Unsurprisingly, these have small space allocations and are for special purposes. The stack used will be determined by the nature of the activity (functions comprising the activity). These three stack frames play crucial role in creating a new thread and aiding it to joint the pool of other threads in the system.

We will come back to discuss about these stacks embedded in `struct thread` after we have learned about the stacks and frames (also called activations records for functions) in the context of a C program.

Returning to the issue of thread life cycle, we have already talked of initialization phase and termination phase. In between the thread is performing useful computation and other related activities. These activities only occur when the thread is RUN state. It has the processor to run the instructions in the program code. However, other concurrent threads cannot be ignored for too long. They too must be run. To do this each RUN state comes with a time quantum. On completion of that time the thread is put into READY state and some other thread in ready state is chosen to run. This is done by timer interrupts. At any given time one would likely see many ready threads and one running threads in the system.

However, it may so happen that there is no thread ready to run. Proper functioning of an OS requires some thread to be running otherwise activities of the system will come to a halt; no running thread means no system activity! This is easily taken care of by creating a special thread that is always there and ready. It is called *idle* thread. It has the least priority; it runs only when no one else is around to keep the system going.

There is one more important state of the threads. It is state BLOCKED; also called WAITING. When a thread (program) needs an external event to occur before it can continue further, for example data read, it must wait for that event. Threads in this state are waiting. They need not be scheduled by the scheduler to run. When the waited event occurs, the thread will be transferred back to state ready.

Quick summary: All computational and OS activities occur in the context of a thread. The threads share processor time as disjoint periods. Scheduler ensures that processor time is shared properly by the threads. Switching (replacing) of a running thread by a ready thread can be synchronous (usually synchronous switch are voluntary) activity, or asynchronous (usually involuntary) change. A voluntary switch occurs at the fixed points in the program when thread/program performs an action that needs to wait for an external (outside the program) event. In this case, thread normally goes into a block state.

An involuntary switch is primarily due to an external or timer interrupt. The thread typically is placed in a ready queue. The thread will wait for the scheduler to "dispatch" it at a later time to resume computation.

Actions and activities related to thread scheduling and switching are all responsivities of the OS. In PintOS this code is primarily located in directories `threads/` and `devices/` in PintOS. Application programmers do not directly write any code to support these activities. Code that an application programmer writes is all related to the application requirements.

Introduction to Compiler related issues

When a function is called there is a set protocol so that the calling function and the called function can communicate parameters and return values. For this they share a program-wide stack called runtime stack. Caller creates a new activation record for the function to be called. In this record or frame the caller places at set and agreed locations space for return value, function parameters, and the instruction reference at which the caller will resume its activities after the call completes and returns to the caller.

The called function, uses the space above (be warned that the actual direction of stack growth, on your computer, may be from a low address towards a high address, or it may be from a high address towards a low address) for its local variables. The called function may, in turn, call other functions following the exact same protocol.

Another issue of interest to us is linking of the functions in the program into a single executable. We will ignore the dynamic linking and pretend that a monolithic executable is constructed before the program starts running. This monolith has machine code for all functions included in the program. Thus, in the executable each calling function knows where the code for the called function is located. This resolution is done by a linker that is run after compiler has translated C function codes into machine (or some similar low-level) codes.

C also lets us access this location of the function codes as a pointer. You can declare variables of the right type to hold these pointers. Function name is also such a pointer. Please see function `run_actions()` in file `threads/init.c` to see how the idea is used in PintOS. Some more information is easily seen on internet; for example: https://en.wikipedia.org/wiki/Function_pointer#Simple_function_pointers

Test Cases for Exercises in Project Threads

The section tries to explain how command `make check` included in PintOS runs for exercises under project Threads.

For projects after thread command `make check` tests PintOS code augmented by you by creating an executable of a test program and runs the test program on your implementation of PintOS. The test program is not part of the kernel but a separate program that runs on the kernel.

However, the test programs used in project Threads are included into the kernel by the linker that runs before PintOS is created. The following command is copied from the screen output of command `make` under directory `threads/`. Notice that linker `ld` is linking all test cases here into PintOS kernel

```
ld -melf_i386 -T threads/kernel.lds.s -o kernel.o
threads/init.o threads/thread.o threads/switch.o
threads/interrupt.o threads/intr-stubs.o threads/synch.o
threads/palloc.o threads/malloc.o threads/start.o
devices/timer.o devices/kbd.o devices/vga.o devices/serial.o
devices/disk.o devices/input.o
devices/intq.o devices/rtc.o lib/debug.o lib/random.o
lib/stdio.o lib/stdlib.o lib/string.o lib/arithmetic.o
lib/ustar.o lib/kernel/debug.o lib/kernel/list.o
lib/kernel/bitmap.o lib/kernel/hash.o lib/kernel/console.o
tests/threads/tests.o tests/threads/alarm-wait.o
tests/threads/alarm-simultaneous.o
tests/threads/alarm-priority.o tests/threads/alarm-zero.o
tests/threads/alarm-negative.o tests/threads/priority-change.o
tests/threads/priority-donate-one.o
tests/threads/priority-donate-multiple.o
tests/threads/priority-donate-multiple2.o
tests/threads/priority-donate-nest.o
tests/threads/priority-donate-sema.o
tests/threads/priority-donate-lower.o
tests/threads/priority-fifo.o tests/threads/priority-preempt.o
tests/threads/priority-sema.o tests/threads/priority-condvar.o
tests/threads/priority-donate-chain.o
tests/threads/mlfqs-load-1.o tests/threads/mlfqs-load-60.o
tests/threads/mlfqs-load-avg.o tests/threads/mlfqs-recent-1.o
tests/threads/mlfqs-fair.o tests/threads/mlfqs-block.o
```

Thus, all test cases in threads project are set as activities within the kernel. These threads are appropriately called kernel threads – these thread only run the kernel code and runs it in the supervisory (kernel) mode. The specific (test case) function is identified through an assembly language code `run_test()`. See file `tests/threads/tests.h`

The user programs (all tests cases after exercises on threads), however, must run outside the kernel in less privileged user mode. These codes are user programs that an unknown user writes and runs on PintOS kernel. So these are not linked into the kernel code. Instead, they are loaded into memory by PintOS and allowed to run. The difference in these manners of code handling is noticed in function `run_task()` in file `threads/init.c`

The user programs may seek services from the kernel through system calls – syntactically the system calls look similar to the function calls but their implementation details are quite different. The methods and features needed for this are lessons to be learned in the later projects.

Contributors:

Vishv Malhotra, Gautam Barua, Arnab Sarkar