Operating systems 02

Sigmund Granaas

October 1, 2020

1 Processes and threads

1.1 Explain the difference between a process and a thread

A process and a Thread can technically complete the same tasks. They are however, not interchangeable with each other. A process is a single instance of a running program. The process' memory is isolated and cannot share anything that is stored in memory with another process. Threads are used by a process to execute code concurrently. A single process can execute code concurrently using threads, there is no theoretical limit besides hardware/software which limits the amount of concurrent threads. Because threads are created within the process, all threads can share memory. This enables threads to communicate and exchange information between threads. Sharing memory space can also lead to problems when programming with threads. One thread can corrupt the memory of another thread if not programmed carefully, which can lead to memory leaks and security issues.

In short, a process is a single instance of a running program, whereas a thread is a section(or unit) of this process with the option to execute code in parallel if using multiple threads.

1.2 Describe a scenario where it is desirable to:

1.2.1 Write a program that uses threads for parallel processing

To efficiently utilize the power of a modern multi-core processor, your code needs the ability to be run in parallel. A single-threaded program is limited by the single core it has been allocated to. Most programs cam benefit from running code concurrently, however, trading code simplicity and safety for speed might not always be the right choice.

A scenario where it is absolutely necessary to execute code in parallel would be a web server. A single person with a slow connection could make all other web clients have to wait in line for the first persons' request to finish before the web server can handle the rest of the requests in line. Serving unique web clients can be handled as completely separate tasks, which makes it a perfect candidate for parallel processing.

1.2.2 Write a program that uses processes for parallel processing

Processes has a higher resource cost than threads in addition to increasing latency when you need to communicate between processes. Some scripting languages like Python or JavaScript does not support the use of threads and are single-threaded by nature. Parallel processing in a language like this creates a need for running a new process to run concurrent code on your CPU.

Multi-threaded code can be complex, and is prone to security flaws. Running tasks as a new process will ensure memory safety by process isolation.

1.3 Explain why each thread requires a thread control block(TCB).

The TCB represent a single thread. The TCP is a subset of the PCB(Process Control Block) stored in the kernel. The kernel(It can also be the JVM) decides which threads(assuming it is a pre-emtive thread) should be running at every point in time. Due to the abstraction between the kernel threads and the user level threads, the kernel needs to keep track of the state of all running threads. Threads will be paused and resumed, and can call themselves into a wait state, where they need to be called by another thread to resume processing. Threads cannot be self managed(This

can cause a lot of issues.) which is why the kernel needs the TCP to schedule, interrupt, resume and dispose threads after they are done.

1.4 What is the difference between cooperative (voluntary) threading and pre-emptive (involuntary) thread-ing? Briefly describe the steps necessary for a context switch for each case.

The difference between pre-emtive and cooperative threads comes down to what decides which threads should be running. Cooperative threads will run until completion, or yield() control to another thread, which in turn has control. Pre-emptive threads does not decide amongst themselves which thread should run, but they have assigned priority the each thread. Pre-emtpive threads can also yield control to other threads manually, but you don't have to rely on threads yielding for the kernel to give control to another thread. The pre-emptive model is more popular with modern processors with multiple cpu cores because it truly allows the processor to run concurrent code. Cooperative threads can be more efficient, but the gains are minimal.

A context switch is the act of saving the thread state and loading the context for another thread so it can resume running another thread. All threads that are ready to be run will be in a runnable state.

If the context switch is voluntary, the thread will either call a library function to exit() the thread or yield() to another thread. Additionally a thread can encounter an error and give control to the next thread in line.

Involuntary context switching can be triggered in several ways. Most common is the hardware interrupt, which occurs periodically, saving the state of the thread, and runs the handler code and decides to keep running the last thread or load another thread's state from the TCP and continue running this thread. Hardware events usually triggers interrupt to the running thread(although we can try to run our code on different threads to limit this) because they have a higher priority. An important point is that the state of the thread should be saved in a way that makes it seem to the thread like nothing happened.

When switching between two threads, involuntary context switching is disabled to ensure threads are put into the right state and that the right data has been saved. Involuntary switching to another thread during a context switch can corrupt memory.

2 C program with POSIX threads

```c
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10
pthread_t threads[NTHREADS];
void *go (void *n) {
    printf("Hello from thread %ld\n", (long)n);
    pthread_exit(100 + n);
// REACHED?
}
int main() {
    long i;
    for (i = 0; i < NTHREADS; i++)
        pthread_create(&threads[i], NULL, go, (void*)i);
        for (i = 0; i < NTHREADS; i++) {
            long exitValue;
            pthread_join(threads[i], (void*)&exitValue);

            printf("Thread %ld returned with %ld\n", i, exitValue);
        }
    printf("Main thread done.\n");
    return 0;
}
```

2.1 Which part of the code is executed when a thread runs? Identify the function and describe briefly what it does.

The function passed down to the thread is the go() function. The function prints to the console, before exiting with a return value.

2.2 Why does the order of the "Hello from thread X" Message change each time you run the program?

The for loop creating all of the threads runs in order from 1 to 10, but the execution of the threads is not up to the threads. Each of these threads is a pre-emptive thread, which means that the order of execution is up to the kernel to decide. In general, the order of the messages are right, but sometimes we can see the effect of the seemingly random prioritization of the threads when they are being run concurrently.

2.3 What is the minimum and maximun number of threads that could exist when thread 8 prints "Hello"?

The minimum number of threads that could exist would be 1, which is thread 8 itself. The maximum number of threads would be 10, if thread number 8 had the highest prioritization(after the main process' thread.) on a single core system, you could see thread number 8 being the first thread to print to the console.

2.4 Explain the use of *phthread_join* function call

The order of the seccond system of prints is always in order, this is because of the use of phthread_join. The main program uses a for-loop to print to the console. The function of the join call, is to suspend the current thread until the target thread, in this case defined by i, has exited. The main thread cannot complete logging before thread 9 has exited, but can complete parts of the for-loop according to how many of the threads have finished. What I would have expected to see, if the go function was a computationally heavy function, would be for parts of the remaining for-loop to finish before a majority of the threads has finished.

2.5 What would happen if the function *go* is changed to behave like this:

```
void *go (void *n) {
    printf("Hello from thread %ld\n", (long)n);
    if(n == 5)
        sleep(2); // Pause thread 5 execution for 2 seconds
    pthread_exit(100 + n);
    // REACHED?
}

Putting thread 5 into a blocking state for 2 seconds will trigger a voluntary
    context switch, allowing thread 5 to resume execution in 2 seconds, but the
    wait might be longer if thread 5 is not prioritized. Causing one thread to
    sleep does not halt the other running threads, except for the main thread,
    which in the for-loop has to wait for thread 5 to exit(printing hello executed
    before the sleep function) before it can continue executing the for-loop. What
    you see in practise is a 2 second delay from printing "thread 4 has exited" and
     "thread 5 has exited" before resuming normal execution.
```

2.6 When *phthread_join* return for thread X, in what state is thread X?

In the documentation it states that phthread_join will run after the target thread has terminated. This means that the state of the thread X is... Terminated.