OBL2-OS

August 12, 2020

This is a mandatory assignment. Use resources from the course to answer the following questions. **Take care to follow the numbering structure of the assignment in your submission**. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine, for example running natively or virtually on your own PC, or by connecting to `gremlin.stud.iie.ntnu.no` over SSH (Secure Shell). Working in groups is **permitted**, but submissions must be **individual**.

1 Processes and threads

1. Explain the difference between a process and a thread.

   Solution: A process is started and administered by the operating system, while a thread is started and administered by a user (or kernel) process. Threads are light-weight-processes, which are easier to set up, do not require their own memory address space (except for a stack), and generally permit programs to do multiple tasks simultaneously.

2. Describe a scenario where it is desirable to:

   - Write a program that uses threads for parallel processing
   - Write a program that uses processes for parallel processing

   Solution: Some examples:

   - Threads: a web server where the overhead of starting a process introduces too much latency in handling requests. Each time a request arrives, a new thread is started to serve the request. This can be made even faster by starting the threads ahead of time and keeping them in a pool. When a request arrives, just pick a thread from the pool to serve the request.
   - Threads: A program with a GUI that stays responsive while doing some task in the background. Here we would use two threads, one for keeping the GUI up-to-date and the other for the background task.
   - Processes: a web browser, where security is a concern and each browser window or tab can be isolated using the operating system's built-in process abstraction (see `https://blog.chromium.org/2008/09/multi-process-architecture.html`, note, since 2008 most browsers have adopted this model).

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

Solution: For a thread to be scheduled by the processor, we need to remember the state it was in when it was pre-empted. Therefore, we need to save its stack and CPU registers. In addition, we may also want to save some metadata about the thread such as priority, thread ID, status (e.g., ready to be scheduled or waiting for some event).

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

Solution: With voluntary threading, each thread must explicitly *yield* for a context switch to occur. The call to yield (for example, `pthread_yield`), causes the scheduler (via a software interrupt or system call) to put that thread on the run queue and schedule the next thread in the queue. The calling thread's registers are copied to the TCB and stack, which permit it to be scheduled to run again later. In a fully voluntary scenario, threads must periodically yield so that scheduling can occur. Misbehaving threads that do not yield can cause the program to crash or freeze. With involuntary threads, the context switch is initiated by a trap (system call) or an interrupt (for example a timer interrupt). The kernel interrupt handler saves the currently running thread's registers to switch into kernel mode. The kernel handler is then run to handle the interrupt, which in turn invokes the scheduler to choose and start the next thread in the queue.

2 C program with POSIX threads

*nix operating systems use POSIX threads, which are provided by the `pthread` library. Consider the following adapted code from the textbook (the code has been modified slightly to use `pthread`, while the book assumes its own thread implementation).

```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;
}
```

We can compile the code and tell the compiler to link the `pthread` library:

```
$ gcc -o threadHello threadHello.c -lpthread
```

At the command prompt, run the program using `./threadHello`. The program gives output similar to the following:

```
Hello from thread 0
Hello from thread 3
Hello from thread 5
Hello from thread 1
Hello from thread 4
Thread 0 returned with 100
Thread 1 returned with 101
Hello from thread 9
Hello from thread 8
Hello from thread 2
Hello from thread 7
Hello from thread 6
Thread 2 returned with 102
Thread 3 returned with 103
Thread 4 returned with 104
Thread 5 returned with 105
Thread 6 returned with 106
Thread 7 returned with 107
Thread 8 returned with 108
Thread 9 returned with 109
Main thread done.
```

Study the code and the output. Run the code several times. Answer the following questions.

1. Which part of the code (e.g., the task) is executed when a thread runs? Identify the function and describe briefly what it does.

   Solution: The function `go` is the task executed for each thread. It only prints a line "Hello from thread X" to the standard output and then exits with a return value of $100 + n$ where n is the thread id assigned as a parameter when the thread is started using `pthread_create`.

2. Why does the order of the "Hello from thread X" messages change each time you run the program?

   Solution: Creating and scheduling threads are separate operations. Normally, threads are scheduled in the order they are created, but this is not guaranteed. Furthermore, even if the threads are executed in order, they may be pre-empted before they are able to print the message.

3. What is the *minimum* and *maximum* number of threads that could exist when thread 8 prints "Hello"?

   Solution: When the program starts, we count that code running as a thread. The main thread. This thread starts `NTHREADS` (10) threads, all of which could run and complete before thread 8 prints "Hello". So the minimum is **2** threads, and the maximum is **11** threads (the main thread + the 10 that are started).

4. Explain the use of `pthread_join` function call.

> **Solution:** `pthread_join` tells the main thread to wait for the thread identified by the identifier provided as the parameter `threads[i]`. We iterate from $i = 0 < NTHREADS$ and call this function in the order that the threads have been created. This means we will wait for each thread in order.

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?
}
```

> **Solution:** Because `pthread_join` is called in order, there will be a 2 second delay while waiting for thread 5 to finish, and this will also block the main thread from joining on threads 6-10. When the join for thread 5 finally returns (after the 2 second delay), then the joins for threads 6-10 can proceed.

6. When `pthread_join` returns for thread X, in what state is thread X?

> **Solution:** When the join returns, thread X has finished running (`pthread_exit` was called by the thread), and it's exit value is stored in the TCB. The TCB is also moved to the finished list. Thread X is therefore in the *finished* state.