Veronica Backer-Peral, Talia Bahar, Maya Pegler-Gordon
7 Feb. 2020
CMSI 387: Operating Systems
Dr.Johnson

Homework #1

1. **What is the difference between an operating system and middleware?**

There are multiple similarities between operating systems and middleware that might make them hard to discern. For instance, the textbook explains how they're both "software used to support other software," focused on creating controlled interactions, and used to merge different strings of computation and bring them together.

The primary difference between the two is that whereas an operating system relies on hardware as the basis from which to provide it's API services, middleware uses an operating system. In this sense, middleware is the 'middle' layer between the operating system and some of the programs that we as users interact with.

Essentially, an operating system provides application programmers with an abstract view of the underlying hardware resources, taking care of low-level details so that the applications can be programmed more simply. Meanwhile, middleware acts as the "go-between" for applications and the operating systems itself by using services from an operating system to supply applications with interaction services. Operating systems and middleware both support other software but they rely upon different service providers.

2. **What is the relationship between threads and processes?**

The author prefaces his explanation of threads and processes by explaining that often the two words are used interchangeably. However, for the purpose of his book, he establishes two distinct definitions.

Threads are the "unit of concurrency," or in essence a sequence of computational steps that follow from the instructions laid out in a program. A process is a "container" that holds and protects threads to ensure that they do not overlap or override each other. Therefore, a process can hold one or multiple threads, and those threads do not interact with threads within other processes.

3. **Of all the topics in the textbook, which are you most excited to learn more about?**

We are particularly excited about learning about virtual memory. Although this topic is briefly mentioned in the introduction, there is an entire chapter on the subject to look forward to. We have already touched a bit on memory in class, especially in terms of cache memory, so we know that memory is an essential part to allowing multiple processes to run at the same time. However, it seems really interesting to understand when all of these concepts, which are traditionally located within the computer, began to be transformed into the virtual realm.

4. **Suppose thread A goes through a loop 100 times, each time performing one disk I/O operation, taking 10 milliseconds, and then some computation, taking 1 millisecond. While each 10-millisecond disk operation is in progress, thread A cannot make any use of the processor. Thread B runs for 1 second, purely in the processor, with no I/O. One millisecond of processor time is spent each time the processor switches threads; other than this switching cost, there is no problem with the processor working on thread B during one of thread A/s I/O operations.**
    a. **Suppose the processor and disk work purely on thread A until its completion, and then the processor switches to thread B and runs all of that thread. What will the total elapsed time be?**

       Thread A: 10 milliseconds + 1 millisecond = 11 milliseconds X 100 TIMES
       === 1100 milliseconds = 1.1 seconds
       SwitchThreads (A,B) = 1 millisecond
       Thread B: 1 second
       Total: 1100 milliseconds + 1 millisecond + 1 millisecond = **2101 milliseconds**

    b. **Suppose the processor starts out working on thread A, but every time thread A performs a disk operation, the processor switches to B during the operation and then back to A upon the disk operation's completion. What will the total elapsed time be?**

       Thread A: [in I/O]
       SwitchThreads (A,B) = 1 millisecond
       Thread B = 10 milliseconds
       SwitchThreads (B,A) = 1 millisecond
       Thread A: 1 millisecond

       REPEAT 100 TIMES [time elapsed: 1300 milliseconds]
       Thread B: 100 milliseconds

       Total Time Elapsed: 1,400 milliseconds

c. **In your opinion, which do you think is more efficient and why?**

It appears that method 2 is far more efficient. In this case, the difference between the two methods was only 701 milliseconds. However, if thread B had more loops, the difference between the two methods would have been extremely marked. Without switching between threads, the 10 milliseconds of I/O time that Thread A has to wait are completely lost. However, by switching between one thread and another, we are able to maximize the use of the processor instead of waiting around for Thread A to finish. Even if there is a slight cost in implementing the SwitchThread method, we would assume that this cost is minimal in comparison to the benefits.

5. **Find and read the documentation for *pthread_cancel ()*. Then, using your C programming environment, use the information and the model provided in Figure 2.4 on page 26 of the textbook to write a program in which the initial (main) thread creates a second thread. The main thread should sit on a read call of some kind, waiting to read input from the keyboard, waiting until the user presses the Enter key. At that point, it should kill off the second thread and print out a message reporting that it has done so. Meanwhile, the second thread should be in an infinite loop, each time around sleeping five seconds and then printing out a message. Try running your program. Can the sleeping thread print its periodic messages while the main thread is waiting for keyboard input? Can the main thread read input, kill the sleeping thread, and print a message while the sleeping thread is in the early part of one of its five-second sleeps?**

```
/** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *  File name     :  pthread.c
 *  Purpose       :  Runs a thread that sleeps then prints a message until the enter key is pressed.
 *  Author        :  Talia Bahar, Maya Pegler-Gordon, Veronica Backer Peral
 *  Date          :  2020-02-4
 *  Description   :  This program creates a second thread in the initial (main) thread. THe main thread sits
 *                   on a read call waiting until the user presses the Enter key. At that point, it should
 *                   kill off the second thread and print out a message reporting that it has done so.
 *                   Meanwhile, the second thread should be in an infinite loop, each time around
 *                   sleeping five seconds and then printing out "Hey what's up?!".
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
```

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

static void *child(void *ignored) {
   while (1) {
       sleep(5);
```

```
        printf("Hey what's up?!\n");
    }
}

int main(int argc, char *argv[]) {
    pthread_t child_thread;
    int code;
    code = pthread_create(&child_thread, NULL, child, NULL);
    if (code) {
        fprintf(stderr, "pthread_create failed with code %d\n", code);
    }
    char n;
    int result = scanf("%c", &n);
    if( result != 1 || n == '\n') {
        pthread_cancel(child_thread);
        printf("killed child\n");
    }
    return 0;
}
```

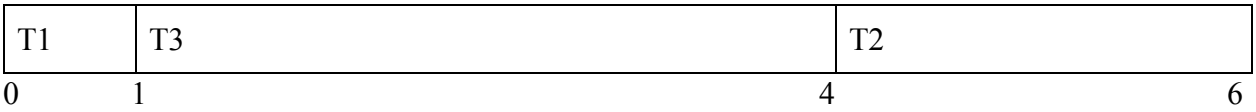
6. **Suppose a system has three threads (T1, T2, and T3) that are all available to run at time 0 and need one, two, and three seconds of processing, respectively. Suppose each thread is run to completion before starting another. Draw six different Gantt charts, one for each possible order the threads can be run in. For each chart, compute the turnaround time of each thread; that is, the time elapsed from when it was ready (time 0) until it is complete. Also, compute the average turnaround time for each order. Which order has the shortest average turnaround time? What is the name of the scheduling policy that produces this order?**

Gannt Chart — a chart that can be used "to see whether a rate-monotonic fixed-priority schedule will work for a given set of threads."
**add waiting time + running for total thread turnaround time

T1	T2	T3
0	1	3	6

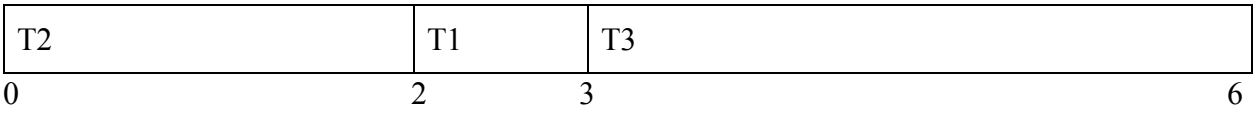
T1: 1
T2: 3 (1 second of waiting for T1 to finish, then 2 seconds of running T2)
T3: 6

Average: 10/3 = 3.3333333 seconds



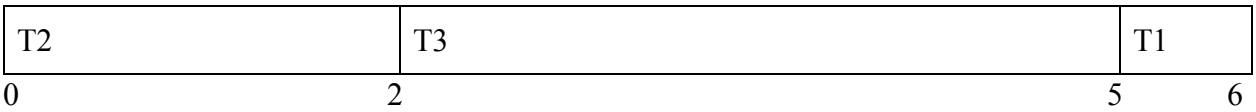
T1: 1
T2: 6
T3: 4

Average: 11/3 = 3.6666667 seconds



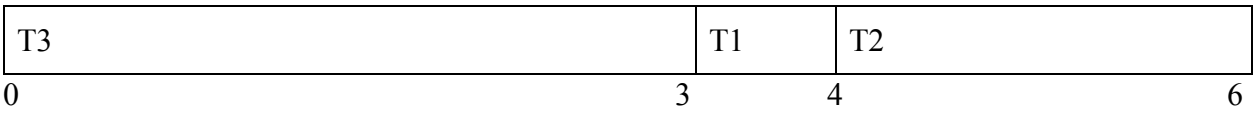
T1: 3
T2: 2
T3: 6

Average: 11/3 = 3.66666667 seconds



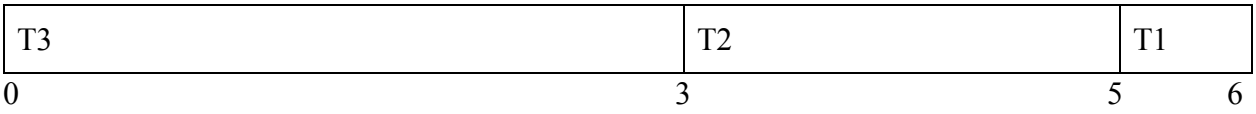
T1: 6
T2: 2
T3: 5

Average: 13/3 = 4.333333333 seconds



T1: 4
T2: 6
T3: 3

Average: 13/3 = 4.33333333 seconds



T1: 6
T2: 5
T3: 3

Average: 14/3 = 4.66666667 seconds

FASTEST ORDER: T1, T2, T3

Scheduling policy: Part 1 of the Liu Layland theorem (also known as Shortest Job First [SJF] Scheduling according to class)

Under a fixed-priority scheduling system, the Liu and Layland theorem states that "if threads will meet their deadlines under any fixed-priority assignment, then they will do so under an assignment that prioritizes threads with shorter periods over those with longer periods." In this case, since there was no given deadline, it is most reasonable to assign the thread with the shortest period to go first and the thread with the longest period to go last, since this will maximize efficiency and create the shortest average turnaround time.

7. **Google the C standard library API and find out how to get information from the command line by using a printf() call to display a prompt, then another call [which you will look up] to get the user input. Write a program in C to prompt the user demographic information including name, age, class year, and any three other data times you wish. Structure the program as a call-and-response program such that each data item is a single question with a single answer. When all data has been obtained, display the data on the console. Each data item must be on a separate line,**

and it must be appropriately labeled. The output must be done using a single printf() statement.

```
/** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *  File name     :  demographics.c
 *  Purpose       :  Provides a simple method to retrieve and print out user given data.
 *  Author        :  Talia Bahar, Maya Pegler-Gordon, Veronica Backer Peral
 *  Date          :  2020-02-4
 *  Description   :  This program prompts the user for their name, age, class year, major, minor,
 *                   and fun fact one at a time using the printf() and gets() functions and then
 *                   prints the results on a new line.
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
```

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {

   char name[25];

   char age[3];

   char classYear[10];

   char major[25];

   char minor[25];

   char funFact[25];

   printf("What is your name?: ");

   gets(name);

   printf("How old are you?: ");

   gets(age);

   printf("What year are you?: ");

   gets(classYear);

   printf("What is your major?: ");

   gets(major);

   printf("What is your minor?: ");

   gets(minor);

   printf("What is a fun fact about you?: ");

   gets(funFact);


   printf("\n\n Name: %s\n Age: %s\n Class Year: %s\n Major: %s\n Minor: %s\n Fun
Fact: %s\n", name, age, classYear, major, minor, funFact);

   exit(0);

}
```