Lab: Process scheduling

2 February 2006

This lab will help to illustrate process states, queues, and the process control block—all concepts from chapter 4. To complete this exercise, you will probably want to have a few pages of scrap paper handy.

We will simulate the execution of several processes on a very simple Central Processing Unit (CPU). The CPU has four registers:

`ACC`	the 'accumulator'
`PC`	the program counter
`BAS`	the base register
`LIM`	the limit register

The accumulator is used to hold temporary results during computation. The program counter keeps track of which instruction we are executing in a given program. Finally, the base and limit registers are used for protected memory management, as discussed in class.

Apart from the CPU, we will need to represent a small chunk of memory. Draw the memory as an array `M`, with each index marked, like this:

`M[0]`	
`M[1]`	
`M[2]`	
`M[3]`	
`M[4]`	
`M[5]`	
`M[6]`	
`M[7]`	
`M[8]`	
`M[9]`	

Now I'm going to give you a small program to simulate on the CPU. Start by initializing `M[5]` to `3` (write the number 3 into the slot labeled `M[5]` above.) Next, draw a picture of the CPU and its four registers. Initialize `PC` to 1, `BAS` to 3, and `LIM` to 6. The accumulator `ACC` may remain empty.

We have initialized the CPU so that the memory owned by the current process ranges from location 3 to location 5. (Memory addresses will be offset by the base, and must remain less than the limit.) Now, we will trace through the following program.

```
1. ACC ← M[2]
2. if ACC = 0 then PC ← 5
3. ACC ← ACC - 1
4. PC ← 2
5. M[2] ← ACC
6. exit
```

We begin by looking at the contents of the program counter, `PC`. It contains 1, right? So we will execute instruction number 1. But first, increment `PC` so it is already pointing to the next step. The `PC` should now be 2.

Now we execute instruction 1, which copies the value **from** memory location 2 **into** the accumulator register of the CPU. But what do we mean by memory location 2? We must take the memory protection mechanism into account! When you see `M[2]` in a program, add the contents of the `BAS` register, and check that the result is less than the contents of the `LIM` register.

In this case, we start with `M[2]`. `BAS` contains 3, and 2+3 is 5. Is 5 less than 6 (the limit)? Yes. Now we are ready to do the copy, but we will copy from `M[5]` (the offset memory location) into the accumulator. Your accumulator should now contain the value 3. **Important:** whenever you see `M[`*i*`]` in a program, don't forget to add the base before accessing the memory.

Okay, now you are finished executing instruction 1. What next? To determine what to do next, just look at the `PC`. It contains 2, so we should do instruction number 2 next. But first, increment `PC` so it is already pointing to the next step. The `PC` should now be 3.

Now we execute instruction 2, which tests whether the accumulator is zero. The accumulator contains 3, so the condition is false. That means we just ignore the part after then. We are now finished executing instruction 2.

What next? To determine what to do next, just look at the PC. It contains 3, so we should do instruction number 3 next. But first, increment the PC so it is already pointing to the next step. The PC should now be 4.

Now we execute instruction 3, which decrements the accumulator. Subtract one from the value in the accumulator, and store the result back into the accumulator. In other words, the value 3 in the accumulator now becumes 2. We are finished executing instruction 3.

What next? To determine what to do next, just look at the PC. (Is this getting repetitive yet?) The PC contains 4, so we should do instruction number 4 next. But first, increment the PC so it is already pointing to the next step. The PC should now be 5.

Now we execute instruction 4, which changes the PC! Assigning a new value to the program counter is essentialy a go to statement, because it forces us to jump to a different instruction than we would otherwise. Go ahead and overwrite the value in the PC with the number 2. We are now finished executing instruction 4.

What next? To determine what to do next, just look at the PC. A-ha! The PC contains 2, so we should do instruction number 2 next. But first, increment the PC so it is already pointing to the next step. The PC should now be 3.

Exercise 1. Go ahead and execute instruction 2, and then continue until you reach an instruction which tells you to exit this process. Once you hit the exit, what is the value in the accumulator? What is the value in M[5] (which this process called M[2])?

Exercise 2. What did this simple program do? How many times did it iterate through the loop?

Program: factorial

This program computes the factorial of its parameter. The parameter is whatever value is initially in M[0]. Recall that the factorial of 4 is 4×3×2×1=24. When the program exits, the result will be in M[1].

```
1. M[1] ← 1
2. ACC ← M[0]
3. if ACC = 1 then PC ← 9
4. ACC ← ACC * M[1]
5. M[1] ← ACC
6. ACC ← M[0] - 1
7. M[0] ← ACC
8. PC ← 3
9. exit
```

Exercise 3. Reset the CPU, initializing PC to 1, BAS to 7, and LIM to 9. That is, M[0] in this program will map to M[7], and M[1] will become M[8] (by adding the base). Initialize M[7] to 3 and simulate execution of the program. You should end up with 3 factorial (= 6) in M[8] when you are finished.

Program: Fibonacci numbers

This program computes a sequence of integers called the *Fibonacci numbers*. The first two Fibonacci numbers are both 1. After that, the next number is always the sum of the previous two. So 2 (= 1 + 1) is the third Fibonacci number. The fourth is 3 (= 1 + 2). The fifth is 5 (= 2 + 3). Here is more of the sequence:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...
```

This will compute the n^{th} Fibonacci number, where n is the value in M[0] at the start of the program. When the program exits, the answer will be in M[2].

```
1. M[1] ← 1
2. M[2] ← 1
3. if M[0] < 3 then PC ← 13
4. ACC ← M[0]
5. M[0] ← ACC - 2
6. ACC ← M[1] + M[2]
7. M[1] ← M[2]
8. M[2] ← ACC
9. ACC ← M[0] - 1
10. if ACC = 0 then PC ← 13
11. M[0] ← ACC
12. PC ← 6
13. exit
```

Exercise 4. Once again, reset the CPU. Initialize PC to 1, BAS to 1, and LIM to 4. Initialize the absolute address M[1] (known as M[0] to this process) with the value 6. Simulate the execution of the program. You should end up with the 6th Fibonacci number (= 8) in M[3] (known as M[2] to this process) when you are finished.

Scheduling

Now that you are familiar with how to simulate a CPU (and, more specifically, with the two programs on this page), we will attempt to run *both* of them simultaneously on *one* CPU. The "operating system" (in other words, *you*) will switch between the two different tasks whenever the timer interrupt occurs.

To keep track of what is going on, each process will need its own *process control block* (PCB). It looks like this:

Process ID	
ACC	
PC	
BAS	
LIM	

Initialize both PCBs by setting the process IDs to 1 and 2 (respectively), the PCs to 1, and the base and limit to some *non-overlapping* regions of memory. Each task should have at least 3 slots of memory to call its own.

Next, you need to provide the initial paremeters for both tasks. Initialize M[0] of the factorial process with the number 5, and set M[0] of the Fibonacci process to 8.

Once the PCBs are initialized, both of them go into the *ready queue*, and we do a *dispatch:*

Dispatch

1. Select the process at the front of the ready queue.
2. Copy the contents of its PCB into the corresponding CPU registers.
3. Let it run! You will know which instruction to execute next by looking at the value of the PC.

Timer interrupt

When the timer interrupt occurs, do the following:

1. Finish whatever instruction you are currently executing.
2. Copy the values of all 4 registers into the PCB of the current process.
3. Move the PCB to the end of the ready queue.
4. Do the **Dispatch** procedure, above.