CS376 W14 Assignment

Dalton Rothenberger

Directions: Edit this document to answer the following questions. If you need to draw a diagram or handwrite something, be sure to include it in this document also.

1. Consider a CPU scheduling algorithm that gives higher priority to processes that have used the least processor time in the recent past. Why does this algorithm favor processes that need to block on I/O frequently? Will CPU intensive processes starve? Explain your answers.

It will favor IO processes because they have a relatively short CPU burst. The CPU intensive processes will not starve because the IO processes will release the CPU when they go to do their IO so then these other processes can access it

2. In multicore systems, processes that are ready to run wait on a run queue. Consider a multicore system. Is it better for each core to maintain its run queue, or would it be better for a single run queue to be shared by all the cores? Give the benefits and drawbacks of each

   It is better for each core to maintain its run queue and then look at the other cores to see if they can assist the other cores when they are available to do so or if they are overloaded they can give work to other cores. The cores can do this via push and pull migrations where in a push migration they give work to another core and a pull migration they take work from other cores. This allows for the cores to be flexible and load balance but can lead to other problems. One problem is whether the moved process should be copied over to the other cores cache or not but usually it is not copied cause the slower access time is okay because the moved process may be returned to the core it came from. The single run queue is good because it avoids this problem of processor affinity but is less efficient because it is not able to load balance as easily as the other method. If a core gets overloaded or under utilized it cannot be assisted by the other cores in this schema.

3. Consider a round-robin type scheduling scheme where instead of defining a time quantum for the entire system, each process gets assigned a quantum. The system keeps track of who uses their entire quantum and who doesn't. If a process blocks on I/O before it uses its entire quantum executing in the CPU, its quantum is reduced.   If a process uses its entire quantum, its quantum is increased. Assume a minimum quantum of 50 and a maximum quantum of 500. Also assume the increment/decrement is 5.   Which type of process does this favor? A process that uses more CPU (aka CPU bound) or a process that uses more I/O (I/O bound)

The scheduler would favor CPU bound processes as they are rewarded with a longer time quantum when they use the entire quantum. IO processes will have their quantum's reduced due to blocking for IO before using up the entire quantum.

4. Using your book or the internet, research what the "nice" command does for Unix systems. Describe it in your own words. Explain why some systems may allow any user to assign a

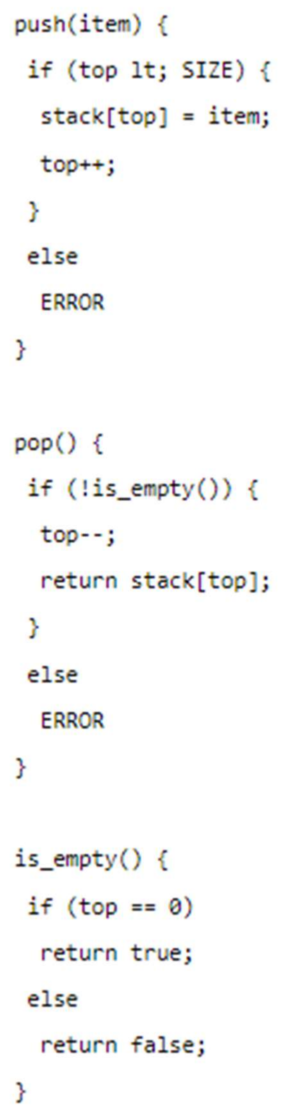
process a nice value >=0 but only allow the root (aka the administrator all powerful) user to assign nice values < 0.

The nice command allows for the adjustment of the nice value for a given process. By changing the nice value of a process allows for the user to set an advised CPU priority that can be used by the scheduler to determine if a process will get more or less CPU time. The values range from -20 to 19 where -20 is the highest priority so more CPU time and 19 has the lowest priority so less CPU time. Some systems allow for a nice value >=0 by people that are not the root because while its priority may be lowered it will still get CPU time. With < 0 the user could cause a process to have higher priority over a "life sustaining" process for the OS so this should only be allowed by the root because they are supposed to be the powerful user so one they should know how to use the power to not mess up the OS and they can make a process have a priority over other user's processes nice value.

5. Recall our real-time scheduling schemes of rate-monotonic and earliest deadline first. Why is rate monotonic typically preferable? Give a specific circumstance where EDF is superior to rate monotonic.

Rate-monotonic is considered optimal because if a set of processes cannot be scheduled by RMS then it cannot be scheduled by any other scheduling algorithm that has constant priorities. Also, rate-monotonic is typically easier to understand and implement than EDF. EDF much more difficult to understand and implement than RMS because it uses dynamic priorities. EDF is superior to RMS because it can have 100% CPU utilization theoretically, but it cannot ever achieve this due to context switching and interrupt handling. There are times when RMS will not be able to schedule processes so that they meet their deadlines but EDF will be able to because it will change the priorities of the processes. The periodic nature of RMS would cause a process to be interrupted when the period of another process comes up.

6. Consider the following code which implements a stack using an array. If this code is used in a multiprocessing environment with a pre-emptive scheduling scheme, what data is in danger of race condition? Explain why and how to fix it. Be detailed.



```
push(item) {
 if (top lt; SIZE) {
  stack[top] = item;
  top++;
 }
 else
  ERROR
}

pop() {
 if (!is_empty()) {
  top--;
  return stack[top];
 }
 else
  ERROR
}

is_empty() {
 if (top == 0)
  return true;
 else
  return false;
}
```

Top, stack[top], and the return of is_empty() are in danger of being in a race condition. Top is in danger of a race condition because if pop or push interrupt each other or themselves. This is because the statement top++ and top—require multiple actions. They assign the value of top to a register then increment/decrement the register and then assign it back to top. So if for instance when top++ was running from push and got interrupted from top—from pop during it the following could happen. Top++ could work all the way up to incrementing the register but not assigning the value back to top. Then top—comes along and uses the old value for top and decrements it. Then top++ interrupts top—and assigns back the incremented value to top but then top—finishes after and assigns the decremented value of the old top to top. (ie: old top = 1, top++ = 2, top-- = 0, end up with top=0). stack[top] is also in danger. For example, if push interrupted itself after the if statement, we could have a situation where two items are added but there was only space for 1 in the array. This would put the stack[top] in a race condition because depending on the order of execution which item gets in the array before the other would get to stay whereas the other would error. The return of is_empty() is in danger because if two pops get called in a row and top=1 and the second pop interrupts the first pop before it reaches top—then is_empty() would not return true even though it should have because there is only 1 element but two pops. To fix this issue I would use a solution like the one described in question 7. I would make a Boolean that is shared between all the methods and add loops to all the methods so that whenever any of these 3 methods is running the others gets locked into a loop if they try to interrupt it. The Boolean gets toggled false when a method starts running and does not get turned back true until it is complete. The loop would run infinitely while the Boolean is false so it would not start looping unless the Boolean had been turned false which would only happen when a method is currently being executed. The Boolean would get flagged true once that method is finished allowing the other method to then execute.

7. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
  if (amount gt; highestBid)
     highestBid = amount;
}
```

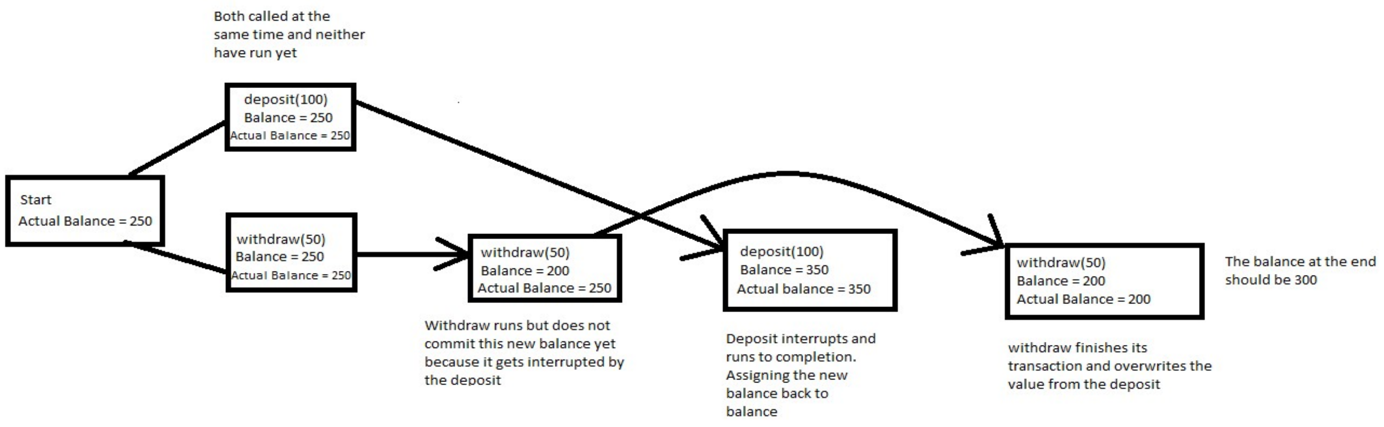
Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

Let the current amount be 100 and then two bids 101 and 102 come in. A race condition could occur where the thread with 101 gets passed the “if” because the current amount is 100. Then it gets interrupted by the 102 thread and it writes its value to amount. Then the thread with 101 finishes by setting amount = 101. Then we end up with a finally bid of 101 even though 102 is the higher bid.

To stop this a Boolean could be added and a loop to the method. The Boolean acts as an indicator of the method already being in use. The Boolean starts as true and gets turned false when it gets passed the loop. The loop only runs while the Boolean is false. The loop keeps other threads of the method from running until the Boolean is switched back to true which would occur at the end of each method. So, for example, the 101 thread is running and sets the Boolean false and gets through the if statement. The 102 thread tries to interrupt but gets caught in the loop. The 101 thread would eventually get back the processor and finish execution and at the end turn the Boolean true. This would then allow the 102 thread to get through the loop and process. This would lead to a final value of 102.

8. Consider a banking system that maintains an account balance with two
functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function, and the wife calls deposit(). Give

an explicit scenario (with drawings/diagrams to illustrate) of when a race condition may occur and a specific solution.


Both called at the same time and neither have run yet
deposit(100) Balance = 250 Actual Balance = 250
Start Actual Balance = 250
withdraw(50) Balance = 250 Actual Balance = 250
withdraw(50) Balance = 200 Actual Balance = 250
Withdraw runs but does not commit this new balance yet because it gets interrupted by the deposit
deposit(100) Balance = 350 Actual balance = 350
Deposit interrupts and runs to completion. Assigning the new balance back to balance
withdraw(50) Balance = 200 Actual Balance = 200
withdraw finishes its transaction and overwrites the value from the deposit
The balance at the end should be 300

Again, like before a solution to this problem would be to add a Boolean that acts as a lock on actual balance as well as a loop to accomplish this locking. When withdrawal/deposit is running it would lock the actual balance by toggling the Boolean so that other method calls would get stuck in a loop and not be able to proceed forward until the Boolean is toggled again. Once the transaction is complete it would toggle to Boolean again which would then allow the other process to run because the loop would finish. This would allow the transactions to run without risking the two interrupting before a transaction can be completed and the balance can be reassigned.