Bryan Arnold

CSE 4300

3/8/18

Homework 2

1a) A priority has a set order of what types of jobs will be dealt with first. For instance, a job of priority 1 would have higher precedence to be completed than a job with priority of 3. In SJF, the job with the shortest duration to be complete would be done first. This is where the two are similar. A priority exists in SJF based on the duration of jobs, so the shortest job in SJF has the highest priority. This is what the two algorithms share.

b) In Multilevel Feedback Queues, the lowest level of the queues works somewhat like FCFS. Since each queue is based on priorities, the lowest priority queue will have very low priority jobs of the same magnitude. So, to determine where one goes, there is no true way. Whichever job shows up first in this level, is what's done first. This is how MLFQ is similar to FCFS.

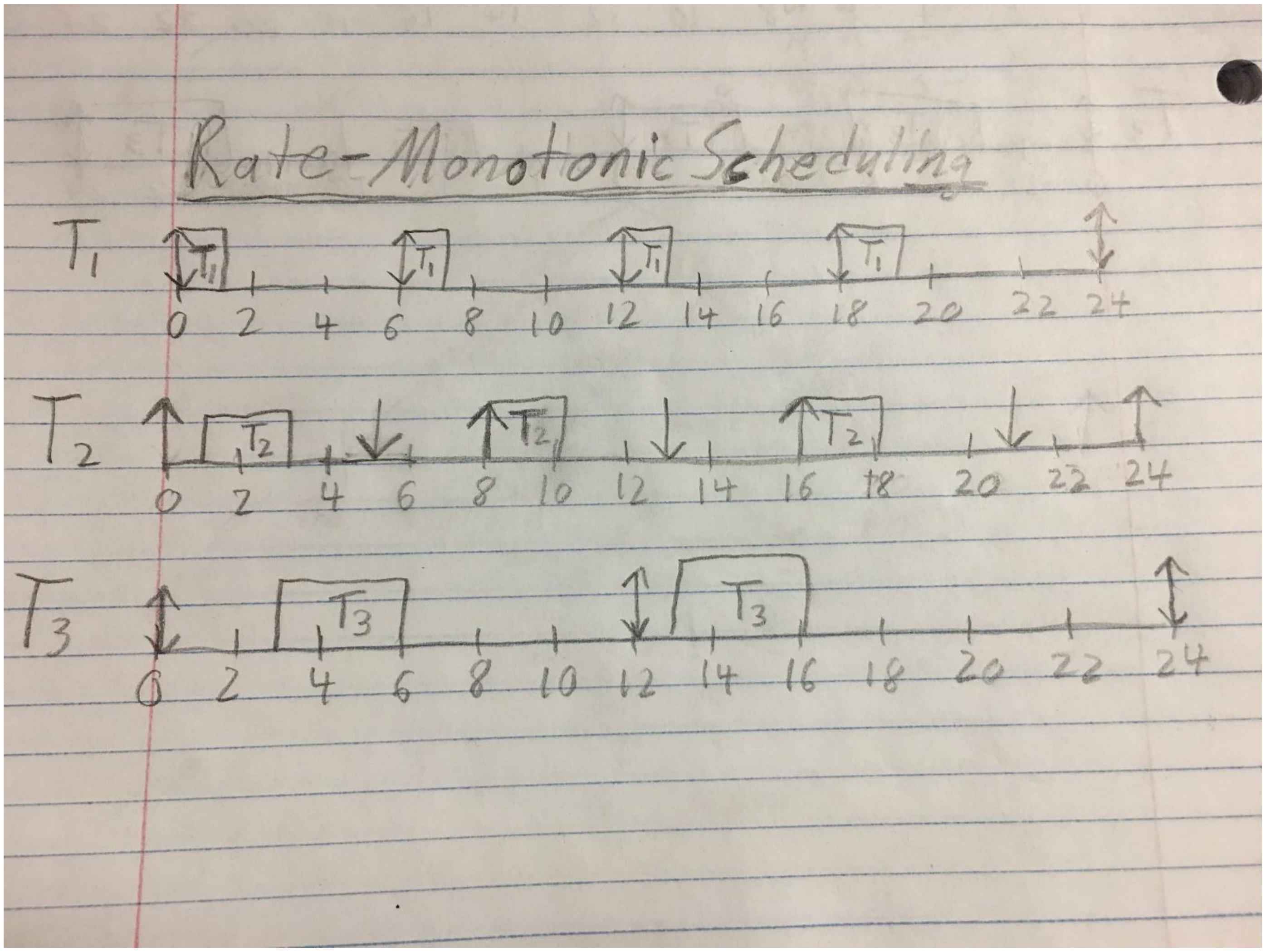
c) In FCFS, there is pseudo priorities put in place. The job that arrives first, and existed the longest, would be done first over any other jobs. This is like a form of priority. Whatever job has been there the longest/came first, has the highest priority to be completed. This is how priority and FCFS share something in their algorithms.

d) RR functions to share everything equally, while SJF isn't fair. SJF gives priority to one job to be completed first, while RR gives every job the same amount of quantum to do some computation. These two algorithms are opposites almost in how they treat jobs in their scheduling, so they do no share anything.

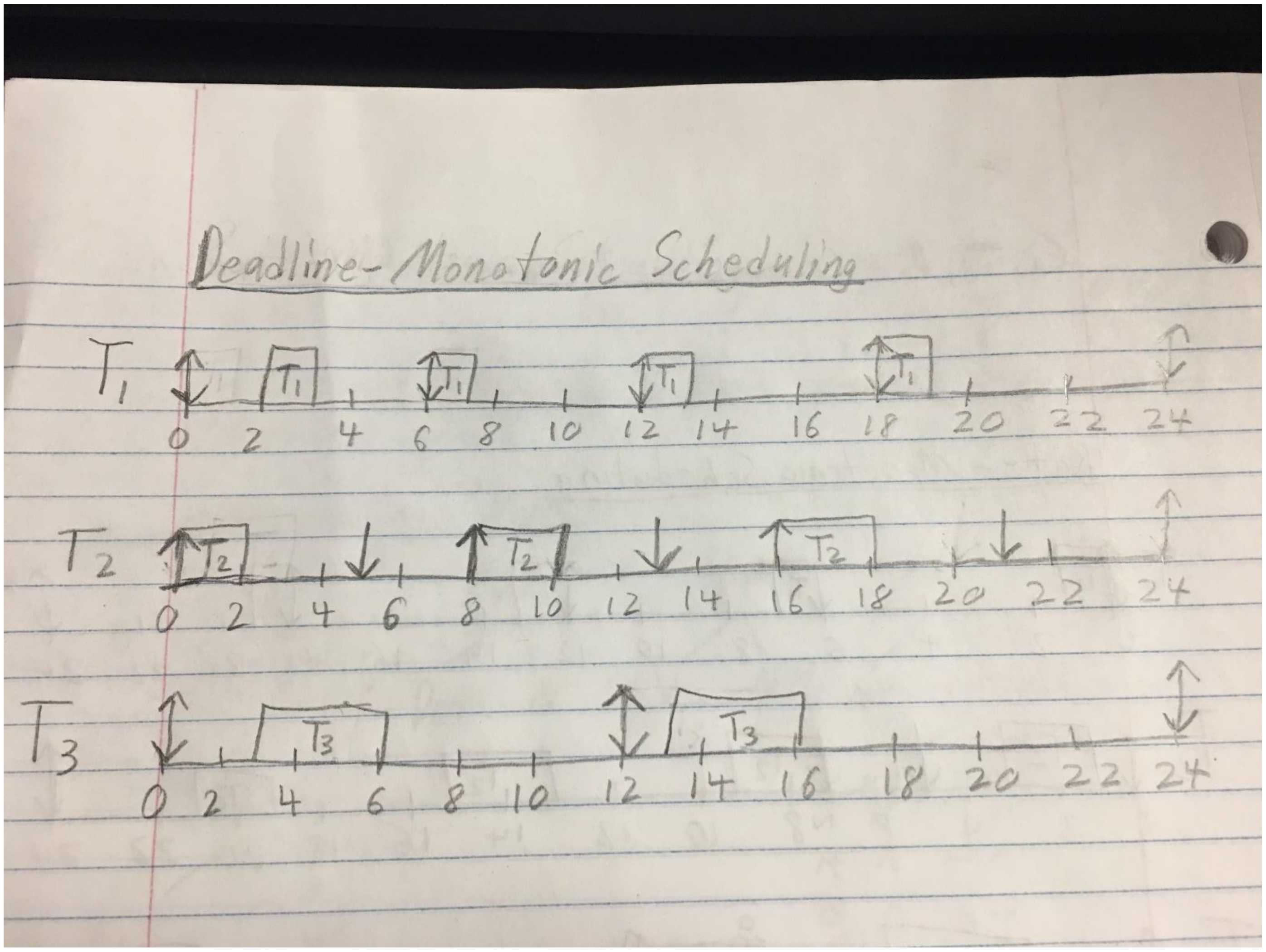
2.1) A periodic task model functions by releasing a job exactly and periodically for some period T. A periodic task also must keep track of a phase, or when the very first job was released. The specification for periodic tasks is denoted by (Phase_i, C_i, T_i, D_i), where D_i is the relative deadline for each job, C_i is the worst-case execution time, and T_i is the period T. For sporadic tasks, the T_i is different than periodic tasks. It is instead the minimal time between any two consecutive job releases. Every other variable is the same in the specification, but Phase_i isn't needed (C_i, T_i, D_i). A sporadic task must also have hard deadline and the time T_i between two jobs cannot be 0. Lastly, aperiodic tasks are like periodic tasks, but the job releasing isn't the same. They are released arbitrarily, if they are identical jobs. Aperiodic tasks also have soft deadlines or no deadline at all, opposite of sporadic tasks, and can have time intervals between jobs be 0.

2.2)

Rate-Monotonic Scheduling

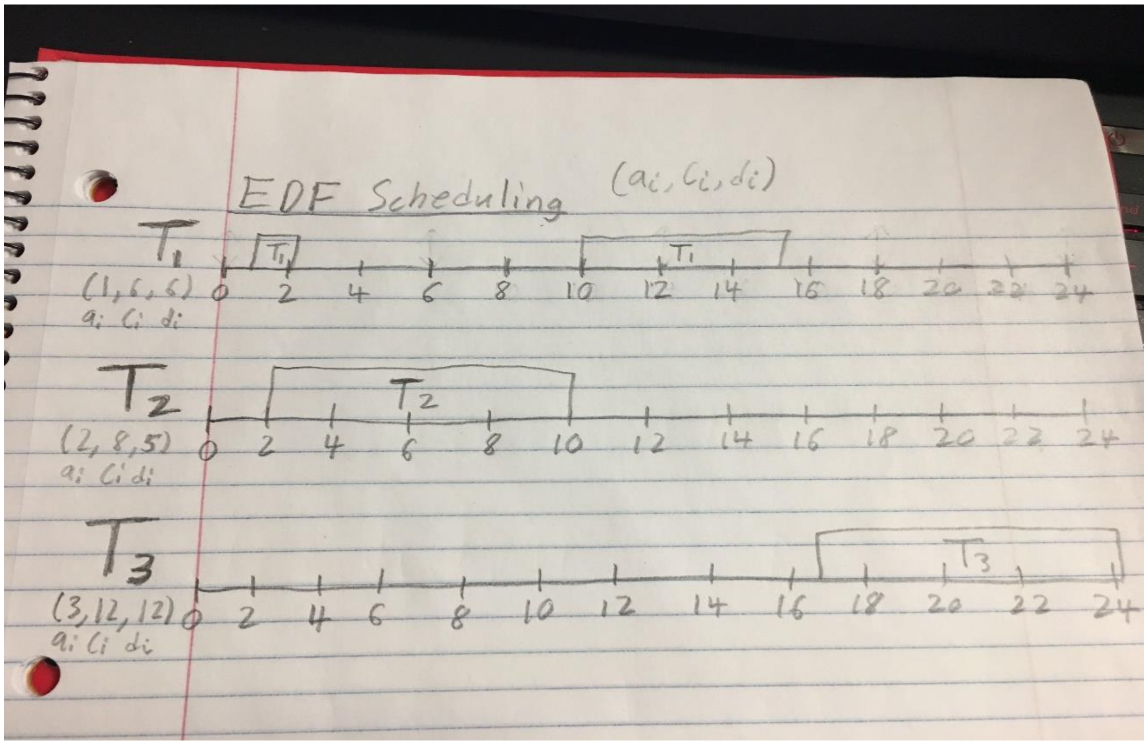


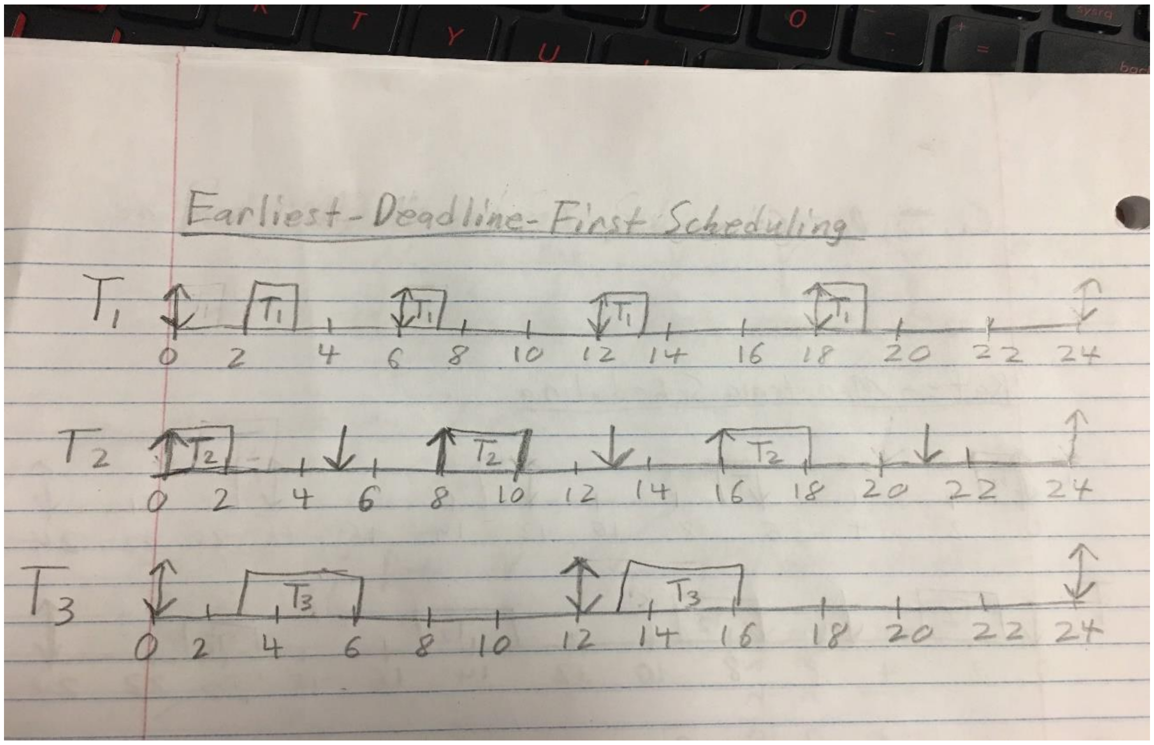
Deadline-Monotonic Scheduling



EDF Scheduling

I wasn't sure which version was wanted, the one similar to the two-previous scheduling, or like the example in the slides. I did both:





2.3) RM average response time = ((1 – 0) + (3 – 1) + (6 – 3)) / 3 = 2

DM average response time = ((2 – 0) + (3 – 2) + (6 – 3)) / 3 = 2

EDF average response time = ((15 – 1) + (10 – 2) + (29 – 17)) / 3 = 34/3

3) Interrupts can be a useful or troublesome thing to have in operating systems. In a single processor system, one processor would be handling everything that computer is doing, meaning any interrupts on this processor entirely impact what happens. By making interrupts user-level program accessible though synchronization primitives, you'd be allowing programs on the user level capabilities to disable interrupts. If a user-level program disables the timer interrupts, this prevents context switching from taking place. This program could then utilize the processor in any way it wants without allowing any other processes to execute. Essentially, a user-level program who could disable interrupts on a single processor machine would be able to take sole control of the processer and no other processes would ever execute if it wanted it to be so.

4) A race condition can occur very easily in this context. Let's set up a scenario. Say the initial value in the account for the couple is 400 dollars. The wife calls the withdraw() function for 100 dollars, withdraw(100), and the husband calls the deposit() function for 200 dollars, deposit(200). When these operations happen concurrently as described, a race condition can occur. The local value for the wife would be 300 dollars, but before the transaction is committed the deposit(200) function would take place that the husband called. This would make the value 500, but when looking back at the wife's view of the balance, the balance would be set to 300, which is not correct. A simple solution would be a way to serialize the two events. So, as a deposit or withdraw is being made, make any other operations next wait for the operation to finish. Once the initial job is finished, allow the next one in line to go. This is a lock or semaphore implementation that would solve this problem.

5) Test and Set Implementation

```
typedef struct {
    int available = 0;
} lock;

int testset(lock *mutex){
    int status = mutex->available;
```

```
        mutex->available = 1;
        return status;


}
void acquire(lock *mutex) {
        while(1){
                if(testset(&mutex)){
                /* Put thread to sleep on the queue */
                } else {
                        break;
                }
        }
}

void release(lock *mutex){
        mutex->available = 0;
        /* Wake up threads */
}
```

For this implementation, the struct needs to initialize the available int in the lock to 0, meaning the lock is initially available always. Next, the testset function is very basic, it gets the value of the lock's available member, changes it to 1 (meaning it is now unavailable), and returns the initial value of the available. If a thread uses this method and it is available, it will change it to unavailable for the next threads until changed. For the acquire method, each thread infinitely checks whether the lock is available or not by constantly calling testset on the lock. Each thread is put on a queue and sleeps, but if the lock even becomes available, break out of the loop. Lastly, for the release function, simply take the available member of the lock and change it to 0 and wake up all the threads. The threads will then go back to testset checking in acquire until it is their turn to execute.

Compare and Swap Implementation

```
typedef struct {
        int available = 0;
} lock;

int compareAndSwap(lock *mutex){
        int a = mutex->available;
        if(mutex->available == 0){
                mutex->available = 1;
        }
        return a;
}

void acquire(lock *mutex) {
        while(1){
                if(compareAndSwap(&mutex)){
                /* Put thread to sleep on the queue */
                } else {
                        break;
                }
        }
}

void release(lock *mutex){
        mutex->available = 0;
        /* Wake up threads */
}
```

For this implementation, the lock struct is the same, as well as the acquire and release functions. The compareAndSwap method looks at the value of the available member in the lock struct. If it is 0, change it to 1 (meaning a thread has acquired the lock), and return the initial availability value. If a thread comes to this method next, the value will be 1 of the available member, so 1 will be returned. This will cause the thread to be put to sleep in a queue in the acquire function. This thread sleeps until woken up by a thread in the release method, where this process infinitely repeats until terminated. For the release method, it functions exactly the same as in testset implementation.

6) The readers-writers problem has a difficult tradeoff between fairness and throughput to decide. On one hand, the throughput of operations can be drastically increased depending on how many readers can share the values being read. Having multiple readers opposed to just one writer at a time allowed to write on the shared values allows for greater throughput. If just one reader were allowed, the throughput would be abysmal. The tradeoff of fairness arises with the problem of starvation. If too many readers are requesting access to the shared value in the queue with very few writers in the queue, the readers would access it much more often than writers. The writers would be starved at writing to the values with the overcrowded queue or readers or potentially limitless number of readers on the values (if the implementation allowed for it). One way to solve this problem would be timestamps. By keeping timestamps on processes that has been waiting the longest, a finished writer could wake up the process waiting the longest. When a reader would arrive with other readers on the shared values, it would wait if writers were waiting ahead of it in the queue. This would guarantee fairness. Another method would be putting a set amount of readers allowed to read the values. For instance, a maximum number of 5 readers could read at once, then if a writer is waiting, no other readers can enter to read. The writer would enter, write, then exit. If a writer is not waiting the readers get free reign of the data 5 at a time, until a writer wants to access the values, in which case the previous happens.