HOMEWORK 2

Name: Vairavan Sivaraman **UNid: u0942570**

6.10 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

1. I/O bound programs perform few computations(less CPU time) before I/O operations.
2. CPU bound programs use their entire time in CPU for performing computation.
3. It is important for scheduler to schedule I/O bound programs before the CPU-bound programs so that we use the system resources effectively.

6.13 In Chapter 5, we discussed possible race conditions on various kernel data structures. Most scheduling algorithms maintain a run queue, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

Each Processor has its own Run queue	A Single Run queue
Scheduling: (Advantage) When scheduler is running in more than one processor and there is only one run queue, there is no contention. The scheduler doesn't have the need to access run queue other that its own run queue.	**Scheduling: (Dis advantage)** When there is more than one process and there is only one queue, there will be contention when two processors scheduler access the single run queue at the same time.To prevent the race condition we have to provide acquiring a lock and reeasing a lock before using the single run queue.
Load Balancing: (Disadvantage) Since there are many run queue, there can be case where one processor is heavily loaded and another processor has less load. There has to be some mechanism to provide load balancing	**Load Balancing: (Advantage)** Since there is only one run queue in this model, load balancing will not be an issue

6.22 Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

The user can use the most of the time quantum allocated for the user process to complete most of the process and voluntarily return the user's process to the CPU before the end of time quantum. By this way the process can maximize the time allocated for the process.

5.20 Consider the code example for allocating and releasing processes shown in

```
#define MAX PROCESSES 255

int number of processes = 0

/* the implementation of fork() calls this function */

int allocate process() {

int new pid;

if (number of processes == MAX PROCESSES)

return -1;

else {

 /* allocate necessary process resources */

++number of processes;

return new pid;

}

}

/* the implementation of exit() calls this function */

void release process() {

        /* release process resources */

--number of processes;

}
```

Figure 5.23.

.

a) Identify the race condition(s).

++ number of process;

--number of process;

b) Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

```
Acquire();

if (number of processes == MAX PROCESSES) {

        release();

        return -1;

}

else {

        /* allocate necessary process resources */

        ++number of processes;

        release();

        return new pid;

        }

Acquire();

-- number of process;

release();
```

c) **Could we replace the integer variable**

```
int number of processes = 0
```

with the atomic integer

```
atomic_t number of processes = 0
```

to prevent the race condition(s)?

Yes, Atomic_t makes the variable synchronized. Hence we don't need the mutex variable to prevent the race condition because mutex variable is also used for synchronization.