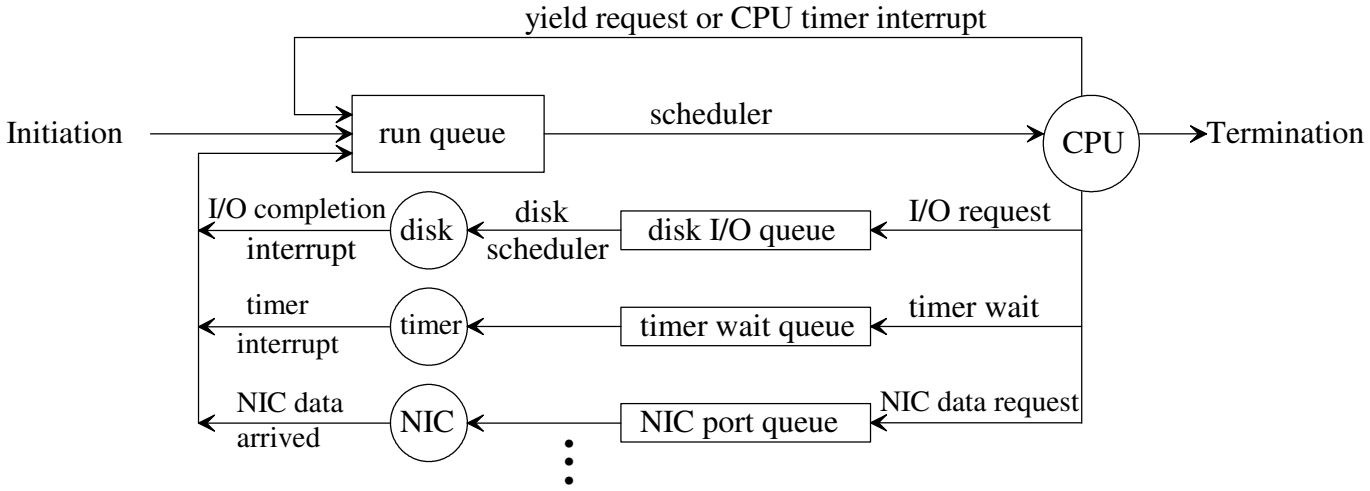
1) "Recent" versions of Linux use the *Completely Fair Scheduler* (*CFS*) to schedule on a processor (with an independent load-balancing mechanism to distribute threads across processors). Characteristics of CFS are:

- each thread has a niceness level (-20 [highest priority] to 19 [lowest priority]). Each niceness level has a weight associated with it (e.g., niceness 0 →1024, 5 →335, etc. via some geometric progression).
- the scheduler round-robins through the threads trying to complete a "pass" in a target time (e.g. 6 ms default on uniprocessor)
- each thread is given a time slice proportional to its weight/(total weights of all runnable threads) (a minimum time slice default is used if proportional time slice is less)

Scheduling mechanism:

- Basically keeps an ordered list of threads by their *virtual run-times* (actually it uses a red-black tree). When a thread runs, its actual run-time is scaled up proportionally to its weight (see Figures 3.10 and 3.11)
- Scheduler is run when:
  - running thread's time slice expires or it yields the processor, or
  - thread (re)enters the run queue

a) Where could a thread (re)enter the run queue from?



b) Why do we not want to start a newly created (initiated) thread with a virtual run-time of zero?

c) What would be a "fair" placement of a newly created thread in the run queue?

run queue: (sorted by virtual run-time)

thread with min. virtual run-time		thread with max. virtual run-time

d) Where would be a "fair" placement of a thread reentering after not being in the run queue for a while?

e) Where would be a "fair" placement of a thread reentering after not being in the run queue for a very short time?

2. What security issues can you think of relating to the scheduler?

3. Consider the following multithreaded C code and its output:

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

int total;   /* global variable */

static void *child(void *ignored) {
  int count;
  printf("Child process started\n");
  for (count=0; count < 1000000; count++) {
    total = total + 1;
  } /* end for */
  printf("Child is done\n");
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t child_thread;
  int code, i;
  char buffer[100];

  total = 0;
  code = pthread_create(&child_thread, NULL, child, NULL);
  if(code){
    fprintf(stderr, "pthread_create failed with code %d\n", code);
    return 0;
  }
  for (i=0; i < 1000000; i++) {
    total = total + 1;
  } /* end for */

  printf("Hit <Enter> after child finishs\n");
  scanf("%c",&buffer);
  printf("total = %d\n",total);
  return 0;
}
```

```
Script started on Wed Jan 25 18:46:27 2012
/opt/pvm3/lib/pvmgetarch: Command not found.
<33 student:~/cs143/cs3430s12/homeworks >./a.out

Child process started
Hit <Enter> after child finishs
Child is done

total = 1105195
<34 student:~/cs143/cs3430s12/homeworks >./a.out

Child process started
Child is done
Hit <Enter> after child finishs

total = 1474658
<35 student:~/cs143/cs3430s12/homeworks >^D

exit

Script done on Wed Jan 25 18:47:23 2012
```

a) What would be the expected `total` value?

b) Why is the `total` value different?

4. Programmers must explicitly (e.g., include code) to control the order of events between threads, called *synchronization*. Synchronization is typically performed by causing one thread to wait for another. Suppose thread A is waiting on thread B to do something, thread C is waiting on A to do something, and B is waiting on C to do something. Complete the following diagram to determine, how long each thread will wait?