1. If a user-level program could disable interrupts, then it would never need to stop for the timer interrupt, which means this process use the CPU for as long as it wants without letting other processes run.
2. In a multiprocessor system, disabling interrupts is costly because the message to disable the interrupts needs to be passed to all the processors and this will delay a process from entering the critical section.
3. This implementation would need a waiting queue to determine the order of process that want to acquire the lock. This means you would need a condition variable that can add and remove process from the queue by blocking and running them respectively. In the acquire method, you would need to check if the semaphore was greater than 0. If it is, then you increment the semaphore, you block the current process, and you add it to the queue. In the release method, you would need to decrement the semaphore, remove the next process in the queue, and signal for it to run.
4.

```
int lock = 0;
int semaphore = 0;
wait() {
	while(test_and_set(&lock)
		;
	if (semaphore == 0)
		//add process to waiting queue for the semaphore
	else
		semaphore--;

	lock = 0;
}

signal() {
	while(test_and_set(&lock)
		;
	if (semaphore == 0 && this is a process in waiting queue)
		//run the first process in the waiting queue
	else
		semaphore++;

	lock = 0;
}
```

5.

```
int time = 0;
int process_count = 0;
condition sleeping;

delay(int ticks) {
	int wait_time = time + ticks;
	process_count++;
	while(wait_time < time)
		sleeping.wait();
	process_count--;
```

```
}

update() {
        time++;
        for (int i = 0; i < process_count; i++)
                sleeping.signal();
}
```

6. C code and makefile attached in zip.