Chapter 5:

3. Busy waiting is when the processing is continuously running a wait method in a loop while it's waiting for the resource it needs. Other types of waiting are mutex locks and semaphores. A way to prevent busy waiting is to put the process to sleep for a small set of time and have it check if the resource is free after that.

4. Spinlock isn't appropriate for a single CPU because it will use a lot of resources to context switch out of the running process. It's better to use a time slice.

7. A race condition is possible in this scenario, where the same variable is being changed by both of the processes. If there is a starting balance of $200 in the account, and the husband pulls out $150 and the wife puts in $100, they'll both see $150 in the account, which would be odd for both of them based on the operations they executed. Using a mutex lock to prevent changing of the variables by multiple processes is a way to solve this problem .

10. This would just be a non-preemptive CPU with no way of stopping a process when it's running.

11. In a multiprocess system, it's not a good idea because then any process can interrupt another process that's running causing a large amount of context switching. It's better to just use the multiple cores for concurrent processes. No way to insure mutual exclusion with multiple processors.

12. While waiting for a semaphore, a busy wait would use a large number of resources by the CPU and prevent other processes from running. If the process is able to acquire the semaphore and then enters a spin lock, no other process could access the resource.

16. To prevent busy waiting we would need an FIFO queue to place the processes in, and when the current running process is done, we move the process out of the waiting queue.

18. If the spin lock is greater than T, then we're better off using using a mutex lock with sleeping processes.

19. The atomic process will ensure that each time the process is started, it will complete, but the issue is what happens between when the process is started and when the increment method is called. The mutex lock is the safer option because it ensures that the integer will never be acquired outside of the mutex lock.

20. a. Race condition: number-of-processes - it can be changed at any time by any process even while another process is running
b. The acquire and release functions would go before the if-else statement and before the number-of-processes is decremented in the second method.
c. The atomic integer would not prevent race conditions. The race condition is in the if else loop meaning at 255 processes, it would decrement one, and prevent another from running, even though it was at 254 processes.

25. If we have a semaphore, the semaphore keeps track of the available resources. In a monitor, only one process at a time can be running by default, which prevents race conditions and make sure that processes are synchronized (notify all methods).

26. Slides - Three semaphores. Mutex for the CS, full for the producer and empty for the consumer. These dictate how data is transferred in the buffer.

28. Starvation prevention is key for this problem. We can assign multiple mutex locks in the program where multiple readers can access data, but at most, one writer can be in there.

36. NA

Chapter 6:

3. Gantt Charts (look up)

4. If you know that your jobs will be very large, it will ensure that the smaller processes are competed first and reduce the number of context switches necessary as time progresses.

5. a. When jobs come in with a high priority, SJF will choose the job that is the shortest to run first.
b. ?
c. When new processes come into the queue, if they have a higher priority then they will be executed faster.
d. RR and SJF are related because this will give an even time slice to each process and then the SFJ will ensure that the shortest one always gets the time slice.

10. CPU bound programs will have a higher latency than I/O latency. Think if a user clicks a mouse and has to wait for a response, they may not want to use that operating system.

13. Each process having its own run queue is beneficial because it allows for the processes to be run in multiple cores, but in the case where one processor gets a large number of long processes, the average wait time will go up, as opposed to having two processors both running a more equal workload. However, both processors with a shared queue could be running long processes and therefore creating more context switching and resource sharing.

16. See Homework 4

17. See paper drawing in notebook ******

20. a. In effect, that process will have increased its priority since by getting time more often it is receiving preferential treatment.
b. The advantage is that more important jobs could be given more time, in other words, higher priority in treatment. The consequence, of course, is that shorter jobs will suffer.
c. Allot a longer amount of time to processes deserving higher priority. In other words, have two or more quantums possible in the Round-Robin scheme.

21. The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1 + 10.1 (as each I/O-bound task executes for 1millisecond and then incur the context switch task, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 * 100 = 94%.

22. A multilevel queue can be maximized by using a feedback queue. The first two can be RR with a lower time-slice at the top and a higher time-slice below, and a third level of FCFS. If a process is too long, it will move to a lower priority queue to finish running, while other more important process can continue to run at the top level. If a process still needs more time after the second queue, it can be moved to the lowest priority queue where it's FCFS and will run for as long as it needs.

24. FCFS is more favorable towards short processes because it will be able to run more processes in a given time frame if they're shorter.
RR will incur less context switching is all processes are short enough to be completed in the quantum given to them.
Multilevel Feedback Queues: These are just multiple implementations of given secluding algorithms that favor shorter processes, because longer processes are all just going to be run in a FCFS queue where less of them will be able to run in a given time frame.

31. See drawings in notebook

Chapter 7:

4. This answer is not the best answer because it is too broad. It's better to define a locking policy that has as narrow a scope as possible.

10. No, it's not possible for one process with a single thread to have a deadlock. The four Coffman conditions aren't met. Circular Wait (not possible) Mutual Exclusion (True) Hold-and-Wait(False), Non-preemptive (false).

11. Circular Wait (each car is waiting for another car in the intersection to move)
Mutual Exclusion (When a car is in the intersection, another car cannot be there too)
Hold-and-Wait (Each car in the intersection is waiting for another car to move in front of them)
Non-preemptive (the cars cannot be swapped out for other cars to move through)
b. No car can enter the intersection unless they can pass all the way through.

12. Reader - writer locks: unlimited amounts of readers, but only one writer.
YES. (1) Mutual exclusion is maintained, as they cannot be shared if there is a writer. (2) Hold-and-wait is possible, as a thread can hold one reader —writer lock while waiting to acquire another. (3) You cannot take a lock away, so no preemeption is upheld. (4) A circular wait among all threads is possible.

13. The scheduler leads to a deadlock in this situation because it allows for the two programs to run simultaneously. When one process runs do_work_one, it gets the first mutex, but is unable to get the second mutex lock. So the two processes are deadlocked waiting for each other to finish. If the thread can get both mutex before another thread, then there is no deadlock.

14. If we add one more mutex for a person, then it's best if we have a third lock that has to be acquired before the person is able to do any sort of transaction (like a master lock).

```
void transaction(Transaction person, Account from, Account to, double amount)
{
mutex lock0, lock1, lock2;
        lock0 = get lock(person);
        lock1 = get lock(from);
        lock2 = get lock(to);
acquire(lock1);
  acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
  release(lock2);
release(lock1);
}
```

19. Check if the number of chopsticks is even after one is picked up. If it is not, then you put the chopstick back down.
Option 2 – Acquire a mutex lock to access the chopsticks. If there are greater than or equal to 2, you acquire the chopsticks one at a time, then release the mutex lock for another philosopher to get chopsticks.

24. The optimistic assumption is that there will not be any form of circular wait in terms of resources allocated and processes making requests for them. This assumption could be violated if a circular wait does indeed occur in practice.

25. Have a mutex lock that is acquired by each side (not the farmer) and when a farmer shows up it lets them go through the bridge. If there are no more farmers on that side, release the lock and let the other side acquire the lock and let their farmers through.

26. Use a waiting queue for each side.

Chapter 8:

5. By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. “Copying” large amounts of memory could be effected by having different page tables point to the same memory location. However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user’s “copy.

11. See Notebook

12. Dynamic Memory Allocation = Relocation possibility
a. Contiguous memory allocation: Might require relocation if there is not enough space to allow the program to grow its memory space.
b. Pure Segmentation: Same as contiguous. If the program cannot get more memory, it will require a relocation.
c. Pure Paging: This will not require a relocation, the program can continue adding more memory into available pages.

13. Contiguous: external fragmentation - not possible to share code
Segmentation: external fragmentation - sharing code is possible
Paging: Internal fragmentation - sharing code is possible, they need access to page table and frames

15. Mobile operating systems don’t support swapping because they have very limited amounts of flash storage. Unnecessary read and write operations would case the storage system to break down and take up too much valuable space the user will need.

20: See Notebook

25: A paging reference would take 100ns. One for the reading the page table, and one for getting the frame number.
b. Adding a TLB with 75% of the page table references would make the effective memory reference time 75% * 2 + 25% * 50 = 14ns

28: See Notebook

Chapter 9:

2. Lower Bound: n / Upper Bound: p

8. Only did two and four frames for practice

12. a. Thrashing is occurring.
b. CPU utilization is sufficiently high to leave things alone, and increase degree of multiprogramming.
c. Increase the degree of multiprogramming.

14. TLB miss with no page fault -> The frame is loaded from the page table and is already in memory, even if it's not in the TLB
TLB miss and page fault -> The frame isn't in the TLB or the page table
TLB hit and no page fault -> This is just a normal execution
TLB hit and page fault -> Not possible

15. a. A TLB miss with a page fault will change from running to blocked
b. A TLB miss with that is resolved in the page table will not cause a change in execution.
c. No state change

17. The Copy on write feature is used when a processes if forked. The parent and child share the same page table until one of them modifies the page table, upon which the page is copied. This is helpful because it reduces the amount of pages that are being copied and allows for more efficient process creation. Must be on the memory level.

21. See homework 5 (Gant charts and demand paging)

22. See notebook (just kidding, I'm not doing conversions)

24. LFU will be more efficient if there is a few resources that are used many times that will result in less page faults. For LRU, it depends on the order they come in, and could replace resources that will appear again but not in the lifetime of the resources's frames. In the instance where a resource will be used a few times in a row, but then not again for a long time, LRU is more efficient because when the resource is no longer needed it will swap it out, but LFU will keep it in because of the amount of times that the resource was used.

27. Which of the following will improve the systems CPU utilization.
a. **Install a faster CPU:** No effect
b. **Install a bigger paging disk**: Will help to store more items in the paging disk, allowing for larger programs to access more space in the paging disk.
c. **Increase the degree of multiprogramming**: No, this will only make the thrashing worse and increase the number of page faults.
d. **Decrease the degree of multiprogramming**: Yes, this will cut down on the amount of thrashing that is occurring.
e. **Install more main memory**: This will help load more things into main memory
f. **Install a faster hard disk or multiple controllers with multiple hard disks**: No effect
g. **Add preparing to page-fetch algorithms**: Sounds good to me?
h. **Increase Page Size**: This will help because less processes will be running and it will cut down on the amount of thrashing.

30. a. i. The value of the counters will be set to 0,
ii. The counters are increased when a frame is loaded in
iii. The counters are decremented when a page fault happens
iv. The page to be replaced will be selected by whichever has the smallest counter
b/c. See Notebook

32. Thrashing is when a processes doesn't have enough pages to load the entire process into memory and begins doing large amounts of page faults. What it needs is for there to be less multiprocessing for the larger process to be able to load itself into the memory and complete its execution.

38. The advantages of allocating pages of different sizes to its processes will reduce the likelihood of thrashing and will also help to reduce the number of page faults because it won't need to swap in and out of the page table.