4.2
If the main thread runs fast enough, it will terminate before some threads can finish printing.

4.7
Threads have their own stack, calling parameters and local variables are on their own stacks so they are per thread.

4.8
Local variables are located on stack and treads have their own stack so they are per-thread.

5.1
No, because disabling interrupt won't stop other cores from running other threads. This single processor solution will not work in multi-processor condition.

5.2
When A is at: while (notB == 1), noteA = 1;
Case 1: (noteB = 0, milk > 0) or ((noteB = 1, milk > 0)),
    Oh there is the milk!
Case 2: (noteB = 0, milk = 0),
    A will continue to finish buying milk, and B will not pass if(noteA == 0) and return.
Case 3: (noteB = 1, milk = 0)
    A will wait at the while loop, and b will not go to the shop because noteA = 1 and eventually take off the note(= 0), then A will go to the shop to buy the milk, yay! :D

5.3
Three different threads are accessing three different queues, there is no sharing resource problem; however there is a problem with the waiting function in main. It only waits for the first threads to finish., creating a race condition between thread1's removal and thread 2 and 3's insertion.
Case 1:
Threads 2 and 3 finishes first or they all finish at the same time, the output would be as expected.
Case 2:
Thread 1 wins and pushing the main thread to finish thread 1's removal before thread 2 and 3 finishes their insertion, it'll start remove items from the thread 2 and 3, and then there might be a bunch of outputs says "nothing there".

5.6
The problem with Mesa semantic is that between signaling and wait resume, other threads can perform operations to change the state of the queue, then the thread has to, again, wait in the while loop when a resumed from wait. However in Hoare semantics the signal resumes the desired thread directly, so the thread can then be done with this queue and then go on with its life...

5.7

```
.
.
.
while ((nextEmpty - front) == MAX || !itemRemoved.isEmpty()) {
        itemRemoved.wait(&lock);
}
.
.
.
while ((front == nextEmpty) || !itemAdded.isEmpty()) {
        itemAdded.wait(&lock);
}
.
.
.
```

6.1
2 1 3 and 1 3 2

6.3
n + 1 would do, because it only need 1 more chopstick to break circular waiting.

6.6
Don't understand the meaning of the term serialize-ability here. I did read the book, I'm still not getting it...

A) It has bounded resources of A,B,C,D,E
B) It has no preemption, the resource can only be released by the acquiring thread.
C) It can be waiting while holding. One can get A and wait on B,C,D,E.
D) There is no circular waiting. Even in the worst scenario, which nothing is released during the second part, the higher order locks can't obtain any other locks lower order than it, according to the rule.
So the freedom from deadlock is ensured.

6.9
Take one chopstick if available > (3 - what this philosopher have), if not move to the next philosopher.

Take one chopstick if available > (k - what this philosopher have), if not move to the next philosopher.

Q1 preemptive scheduling is a scheduling strategy that prioritize the “better” process.
Q2. This algorithm favors I/O bound programs because I/O bound programs tend to have shorter processor time, that way the scheduler can let them quickly finish their job and give CPU-bound program longer time to deal with their CPU intensive problems.
Q3.
A. To utilize CPU usage scheduler would make as least switch as possible between process, which delays response from other other process.
B. To optimize turnaround time, CPU would spend as much time as possible to run this process rather than putting it into the queue, which means as less wait time as possible.
C. I/O device utilization requires CPU jump between user mode and kernel mode and between processes, which lower the efficiency, which is the CPU utilization rate.