Question 2a)

There are three different copies of x

Parent	Child 1	Child 2	Code line number
x = 100	x = 100		Line 9
	x = 90		Line 12
		x = 100	Line 15
x = 80		x = 80	Line 16
x = 70			Line 18

So, the final answers are x = 70, x = 90 and x = 80.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char** argv[]){
    int child = fork();
    int x = 100;

    if(child == 0){//child
        x-=10;
    }
    else{//parent
        child = fork();
        x-=20;
        if(child){
            x-=10;
        }
    }
    return 0;
}
```

2b)

For this we would look at an example where we have two processes with several threads, in the case of user-level threads a thread would be picked and run until it is finished or becomes blocked, it then looks for another thread in the same process and continues. After the process is finished it moves onto the next process and repeats the process of running the threads as detailed before. The advantage of this is that this can be implemented in an operating system that does not support threads.

Now in the case of kernel-level threads, the first thread is picked and run until it is blocked or complete. The kernel-level thread however has the option to either run a ready thread in the same process or if none are available it goes to the next process and runs a thread that is ready there. This repeats until both processes are finished. The advantage here is that if there is a page fault in a process the kernel can still move to another thread that is ready while the page is brought in from the disk.

2c)

enter_region:

```
TSL REGISTER.LOCK     |copy lock to register and set lock to 1
CMP REGISTER,#0       |was lock zero?
JNE enter_region      |if it was nonzero, lock was set, so loop
RET                   |return to caller; critical region entered
```

leave_region:

```
MOVE LOCK,#0          |store a 0 in lock
RET                   |return to caller
```

So, in this case to achieve mutual exclusion a process before entering its critical region, calls enter-region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave-region, which stores a 0 in lock. The processes must call enter_region and leave-region at the correct times for the method to word, if It doesn't it will not achieve mutual exclusion.

2di)

The problem with Peterson's solution and the above TSL instruction is that they have an issue with requiring busy waiting. Essentially, for these solutions when a process waits to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is allowed to enter the critical region.