1.
What is the problem with the following code?

```
struct T {
    int a;
    size_t b;
};

T array[arraySize];

#pragma omp parallel sections num_threads(2)
{
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].a = 1;
   }
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].b = 2;
   }
}
```

The use of "`parallel sections`" means that each subblock specified by "`#pragma omp section`" can be run concurrently, by different threads. As the `size_t i` variable was declared outside of the parallel section, both threads use a shared loop variable when iterating.

The problem can be solved by declaring the loop variable as local.

2.
Use OpenMP to parallelize the following code. What would happen if this was a one-dimensional array, in a single for loop, and the same parallelization was used?

```
int i, j;

#pragma omp parallel for private(i,j) shared(array, n)
for(i=0; i<n; i++)
      for(j=1; j<n; j++) {
            array[i][j]+= array[i][j-1];
      }

1D Case:
for(j=1; j<n; j++) {
            array[j]+= array[j-1];
      }
```

For the 2D case, the workshare works out to where each thread has its own i and then runs their own `array[i][j]+= array[i][j-1]` in sequence. Thus, race conditions can be avoided. In order to share both i and j (which will cause race conditions), a collapse clause is needed. (https://stackoverflow.com/questions/13357065/how-does-openmp-handle-nested-loops)

It is a little more apparent in the following:

```
      int i, j;
      #pragma omp parallel for private(i,j) shared(array, n)
      for(i=0; i<n; i++) {
            array[i][0] = foo();
            for(j=1; j<n; j++) {
                  array[i][j]+= array[i][j-1];
            }
      }
```

Here we can clearly see that only the i variable will be split amongst the threads, not j.

Also, loop variables for an “omp for” are default private. Variables declared outside of the parallel region are default shared. Thus, even #pragma omp parallel for private(j) will suffice, since array and n were declared outside of the parallel region, and i will be default private.

If it was a one dimensional array, in a single for loop, there is a high chance of race conditions. For example, if Thread 1 has indexes 0-4 and Thread 2 has indexes 5-9, Thread 5 can update array[5] with the old value of array[4]. (This is the race condition that was avoided in using nested loops)

3.
Optimize the following code, using OpenMP.

```
void hello(long *old, long *new, int n) {
    int i;
    double sumWeights=0, sum=0;

    sum = n * old[0];

#pragma omp parallel for reduction(+:sumWeights)
    for(i = 0; i < n; i++) {
        new[i] = old[i] * exp(100.0f/old[i]);

        sumWeights += new[i];
    }

    sumWeights /= sum;

#pragma omp parallel for
    for(i = 0; i < n; i++)
        new[i] = new[i]/sumWeights;
}
```

We can combine the first two for loops and use the reduction clause to parallelize them. The division "`sumWeights /= sum`" only needs to happen once.

4.
What are the differences between dynamic and static linking? What are some advantages and disadvantages?

Linking allows for the splitting of code (for one program) across different files. When an executable is generated, different segments of code (libraries) from different files can be combined to make the one program. The main differences between dynamic and static linking stem from when the code is combined:

Static Linking	Dynamic Linking
Can be thought of as “appending” code to a single file	Libraries are loaded into each executable as the need arises (can be load-time or run-time)
Suffers from possibility of duplication of code among executables	Libraries can be shared between executables (eliminates code duplication)
Typically faster than dynamic linking, and allows programs to load in constant time	Might be slower than static linking, since during each library has to be loaded as the need arises, and loads in variable time
if a source program is changed in static linking, everything has to be recompiled and linked	if a source program is changed in static linking, only one module must be recompiled for dynamic linking

5.
What type of exception would each of the following lead to? Are they synchronous or asynchronous exceptions? What is their return behavior?

a. Dividing by 0
   Abort; Synchronous; Does not return to next instruction

b. Tired of waiting for your “optimized” code for the OpenMP lab, you terminate your process by pressing Ctrl-C at the keyboard
   Interrupt; Asynchronous; Does return to next instruction

c. The MMU fetches a PTE from the page table in memory, but the valid bit is zero
   Fault; Synchronous; Re-executes faulting instruction or aborts if unrecoverable

d. You create a file using the open() system call
   Trap; Synchronous; Does return to next instruction

Async exceptions: interrupts (for example, signal from an I/O device)
Sync Exceptions: traps (intentional exceptions), faults (possibly recoverable errors), aborts (nonrecoverable errors)

What's the difference? Asynchronous exceptions are caused due to events in I/O devices, i.e. outside of the CPU. Synchronous exceptions are caused due to executing instructions.

Faults:

- as previously discussed, a fault is a result of an error that may be correctable by the handler (processor transfers control to the fault handler)
- if handler can fix the error condition, the control is returned and the processor re-executes the instruction (if not, then it returns to an abort routine)

what is a type of fault we have discussed in class?
PAGE FAULTS => MMU triggers page faults while trying to translate virtual addresses

suppose we have a normal page fault, i.e. a legal operation performed on a legal virtual address

- what might the handler do?
  - select a victim page
  - swap it out if it is dirty
  - swap in the new page
  - update the page table
- now, when the handler returns, the CPU will re-execute the original instruction, but this time the MMU will translate the address without resulting in a page fault

6. (Textbook 9.3)
Given a 32 bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P:

P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

The number of VPO and PPO bits are the same, and is the log base 2 of the page size. VPN and PPN are the remaining bits from the virtual address size and the physical address size, respectively.

Thus: for 1 KB,

VPO = PPO = $\log_2 1024 = 10$

VPN = 32 - 10 = 22

PPN = 24 - 10 = 14