Optional Project Assignment #1

MUST BE PROGRAMMED ON THE BORLAND C COMPILER. I use command line compiler:
BCC SPACE.C ...

YOU CAN USE EITHER THE SMALL, MEDIUM or LARGE MEMORY MODEL.

In this assignment you are to implement a multi-threaded game program called Space Invaders. You will be required to context switch several types of tasks or threads; Aliens, Tank(s) and Bullets (Up to a maximum of approximately 20 threads for the SMALL memory model). Each of these threads will update a set of parameters and will perform screen I/O. Since the screen is a shared resource, you will have to enforce mutual exclusion (in order to protect it against corruption) using a Semaphore.

I. Implementation concepts

The image of an executable program in memory has three segments: code, data, and stack. In a multitasking system, multiple processes run on the same CPU and share the same hardware support. At the end of a time quantum, the scheduler selects the next process to be run from the ready queue and performs a context switch. A context switch involves saving the state of the currently running process and restoring the state of the selected process. On a PC, the state of a running process is represented by all the registers, its stack and its data segment.

Ideally, a multitasking system needs to support the loading of executable programs from disk to memory and then executing the programs. Unfortunately, the setup for loading is complicated as it involves memory allocation for each segment. We simplify this setup by loading just one executable file that has the code for the main program and the application threads. Thus, all threads share the same code and data segments.

In order to make each of the threads, which are just subroutines in your program, behave like separate processes, a separate stack will have to be created and maintained for each thread. In fact, each of these stacks will be used to save the state of the registers. The data segment, in your implementation, will contain the stacks for each of the threads as well as a set of global variables. Global variables defined in the program will be accessable to all threads (shared memory) so care must be taken so that reads and writes to these variables do not corrupt the values. For this project, most of the global variables shared among the threads can be 'unprotected' since inconsistent updates to them will be harmless. However, this will not be the case for some variables, especially the system data structures you will use to schedule and track threads. Protection can be accomplished using 'asm cli' and 'asm sti'. Details of the thread state, the context switch and semaphores are provided in a separate help sheet.

II. Program behavior and function requirements

1) You are required to implement a multi-threaded Space Invaders game which is composed of three types of threads, Tank(s), Aliens and Bullets. Each of the threads will be represented by a system data structure called a PCB (Process Control Block, sort of a misnomer). The list of PCBs are created at program startup by calling the AllocTaskStructs routine provided. This routine will 'malloc' a list of PCBs as well as a stack and parameter structure for each PCB in the list.

2) Each PCB will contain a 'next' pointer that you will use to put the PCB on one of two Queues, Ready or Blocked. The Ready Queue is a global queue while the Blocked Queue is part of a Semaphore used to provide mutual exclusion for Screen updates. Each thread can be on one and only one Queue. When a thread is NOT waiting on the Screen and has not been terminated, it should be placed on the Ready Queue (unless it is the currently running thread). If the thread wants to update the Screen, it MUST call Wait. If another thread is using the Screen, the current thread is placed on the Blocked queue and another Ready Queue thread is scheduled to run. When the thread that 'owns' the Screen finishes its updates to the Screen, it MUST call Signal. If threads exist on the Blocked queue, the current thread (the one that called Signal) is placed on the Ready Queue and a thread from the Block queue is scheduled to run.

3) A Timer ISR MUST be installed using interrupt number 0x8 (DO NOT USE 0x1c). The Timer ISR will be called 18 times a second and will context switch threads on the Ready Queue in a round robin fashion. This is accomplished by resetting the current values of the Stack offset and segment registers - referenced as _SS and _SP in your C code. You must first save the current values of _SS and _SP in the PCB of the running thread, Enqueue the current thread, Dequeue a new thread from the Ready Queue and finally reset the values of _SS and _SP using the values saved in the PCB of the new thread.

4) You MUST also install the keyboard handler provided using interrupt number 0x9. The keyboard handler will buffer user input and the Tank thread will read this input when it is scheduled to run. I have provided you with the keyboard ISR routine and ring buffer routines you will need to process keystrokes from the user. Since the thread stacks are relatively small, installing the keyboard handler will guarantee that stack overrun will not occur. Calling C library functions such as getc() require unpredictable amounts of stack space since these functions usually contain calls to other library functions.

5) You will have to write the code to implement the three threads as described in the comments in a3_utils.c. Specifically, one Tank thread must be created and put on the Ready Queue at program startup. Several Alien threads must also be added at program startup. Bullet threads will be created dynamically by both the Tank and Aliens.

6) The Tank thread is controlled by the user using the keyboard. Only three keyboard characters are considered valid - the left and right arrow keys and the 'f' key. Pressing the left or right arrow keys should move the Tank one

position left or right, respectively, on the Screen. The keyboard ISR that I have provided will put 'l' and 'r' on the ring buffer when the left and right arrow keys are pressed, respectively. Pressing the 'f' key should cause the Tank thread to create a Bullet thread that is then placed on the Ready Queue (see the comments in a3_utils.c).

7) The Tank thread MUST call the Scheduler directly if no characters exist in the keyboard buffer. In this way, the Tank does not consume busy wait CPU cycles when it has nothing to do (sort of simulates a system call). You will also need to generate Alien threads occasionally from the Tank thread since Aliens will terminate when they are hit by Tank Bullets. The Tank thread should NEVER be terminated.

8) Several Alien threads should be created at program startup and placed on the Ready Queue. Alien threads update their column position (and possibly their row position) using a random number generator 'rand'. rand returns a number between 0 and 2^15-1 every time it is called. Aliens also create Bullet threads either at random or at predetermined intervals. Alien threads MUST terminate themselves (self-terminate) if they are hit by Tank bullets.

9) Bullet threads are created either by the Tank or by the Aliens. In order to distinguish between the two types of Bullet threads, you will have to pass a parameter to the CreateTask routine (which will be called from the code in the Tank procedure and Alien procedure). The parameter value should be saved in the parameter structure created for the Bullet thread. A pointer to this parameter is saved in the PCB of the Bullet threads and is also made accessable by the Bullet thread via its stack (see CreateTask). Bullet threads are short lived in that they travel from the top of the screen to the bottom or vise versa. When they pass off of the screen, they MUST terminate themselves. On their journey across the screen, they MUST register hits on the Tank or Aliens by updating the Tank or Alien parameter structure. The Tank and Alien parameter structures are accessable through their corresponding PCBs. The Tank or Alien thread, when it is run again, will then examine their own parameters and take the appropriate action (terminate for Aliens, update global game variables for both).

10) Other details are discussed in the comments in the code given in the file a3_utils.c.

11) Terminated threads should be removed from both queues and the PCB marked as available for reuse.

12) The game should be terminated and the original Keyboard and Timer ISRs restored when the user presses Ctrl-C. You may also terminate the game when a certain number of tanks and or aliens are destroyed. The screen must be cleared when the game terminates and if you want, you can display game statistics on the cleared screen.

VI. What to turn in

1. The grading sheet with your comments and name (See the grading sheet for details).

2. All your source programs named 'a3*.*', a project file 'a3.ide', and the executable 'a3.com' or 'a3.exe', in the root directory on a clean (without other garbage files) 3.5" diskette. Please put your name on your diskette also.

3. A printout of your entire programs. The printout must be the same version as the files on your diskette.

4. The grading sheet and printout should be stapled together with the grading sheet as the coversheet.