1 We Interrupt This Program (10 pts.)

There has been some confusion surrounding interrupts and handlers. We are giving two implementations of the keyboard handler and readchar(). For each implementation, state which lines of the following code are problematic and why.

Assume that we have included all necessary files.

1.1 5 pts

```
1  #define BUFSIZE 128
2
3  static int buf[BUFSIZE];
4  static int *head = buf, *tail = buf;
5  static int count = 0;
6
7  void keyboard_handler() {
8      int sc;
9
10     sc = inb(KEYBOARD_PORT);
11
12     if (count < BUFSIZE) {
13         *tail = sc;
14         count++;
15         tail++;
16         if (tail == buf + BUFSIZE) tail = buf;
17     }
18
19     outb(INT_CTL_REG, INT_CTL_DONE);
20 }
21
22 char readchar() {
23     int sc;
24
25     if (count > 0) {
26         sc = *head;
27         head++;
28         if (head == buf + BUFSIZE) head = buf;
29         count--;
30     }
31     else return -1;
32
33     return (char) process_scancode(sc);
34 }
```

1.2 5 pts

```
1  #define BUFSIZE 128
2
3  static int buf[BUFSIZE];
4  static int ins = 0, rem = 0;
5
6  void keyboard_handler() {
7      int sc, inp;
8
9      disable_interrupts();
10
11     sc = inb(KEYBOARD_PORT);
12
13     inp = ins;
14     ins++;
15
16     if (ins == BUFSIZE) {
17         if (rem != 0) ins = 0;
18         else ins = BUFSIZE - 1;
19     }
20
21     if (ins == rem) ins = rem - 1;
22     else buf[inp] = sc;
23
24     enable_interrupts();
25     outb(INT_CTL_REG, INT_CTL_DONE);
26 }
27
28 char readchar() {
29     int sc;
30
31     if (ins == rem) return -1;
32
33     sc = buf[rem];
34     rem++;
35     if (rem == BUFSIZE) rem = 0;
36
37     return (char) process_scancode(sc);
38 }
```

2 Building a Question (10 pts.)

At construction sites, there are many workers doing various jobs to complete the building. A construction worker needs a hammer, a hard hat, and nails. At this particular site, the foreman is rather relaxed. The hours they work are up to each worker, so they arrive and depart at whatever time suits them. In addition, the construction workers often forget to bring all of their equipment with them so the foreman provides it to them when they arrive, but he wants the job done quickly so wants to satisfy as many workers as possible even if some never start working. When workers depart work, they will leave some of their equipment behind for other workers to use. The foreman would like you to implement the following code using mutexes and condition variables to allow him to handle his equipment policy. Nails are not part of the equipment list, due to their plentiful nature. You do not need to worry about exact C syntax or invalid parameters.

```
/*
 * This function will be called before any worker arrives.
 * The parameters represent how much equipment the foreman
 * has at the start of the day.
 */
void init(int hard_hats, int hammers);

/*
 * Workers will call this function when they arrive.
 * The parameters are set to 0 if the worker has this and
 * 1 if the worker needs this
 * This should return when the worker has all the equipment
 * he needs.
 */
void arrive(int hh, int hammer);

/*
 * Workers will call this function when they depart.
 * The parameters are what the worker is leaving with
 * the foreman.
 */
void depart(int hh, int hammer);
```

3 Are we dead yet? (10 pts.)

Deadlocks are rather bad to have and can occur in even simple situations. Learning to characterize a particular situation and recognize what future patterns will cause is therefore incredibly important. Given the following situation of three processes (A B C) and three resources (R S T), for the following situations characterize them as safe (having no execution path which leads to deadlock), unsafe (having some execution paths which lead to deadlock), and will deadlock (all execution paths lead to deadlock). To answer the "Why?" question in each part, you may need to draw resource allocation graphs, present safe sequences, and / or make brief proof-like arguments. Please make sure you are concise yet convincing. The resource manager grants all requests that don't violate mutual exclusion.

Process Traces

Process A	Process B	Process C
request(R)	request(S)	request(T)
request(S)	request(T)	request(R)
release(S)	release(T)	release(R)
release(R)	release(S)	release(T)

Execution Trace:

Process A	Process B	Process C
request(R)		
	request(S)	

3.1 3 pts

If, after the execution trace shown above, the next event that happens is process A: request(S), will the system be safe, unsafe or deadlocked? Why?

3.2 3 pts

Assume that, instead of process A: request(S), the third event is process B: request(T), will this system be safe, unsafe or deadlocked? Why?

3.3 3 pts

Finally assume that, instead of process B: request(T), the third event is process C: request(T), will this system be safe, unsafe, or deadlocked? Why?

4 Paging Algorithms (10 pts.)

Given the following memory trace (which is in row–major order: the first reference is 0x07976e and the second reference is 0x02000d), put together a list of which pages will be in memory after each access under the following algorithms: optimal, LRU, and FIFO. Main memory can hold 4 pages and is initially empty. Addresses are 24 bit and pages can hold 16k bytes.

```
0x07976e 0x02000d 0x039004 0x02c432 0x03f00d
0x07be07 0x023223 0x02f430 0x06f402 0x079e04
0x02f10b 0x038dd0 0x078263 0x06c409 0x03e942
```

4.1 2 pts

Process the list of memory addresses to produce a list of page numbers.

4.2 5 pts

Fill in the following table of main memory contents for the three different algorithms using the list from part 1. List the number of the virtual page or a * for unused. Mark an X when the reference causes a page fault.

	FIFO					LRU					Optimal				
	Frames				PF	Frames				PF	Frames				PF
1	1e	*	*	*	X	1e	*	*	*	X	1e	*	*	*	X
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															

For this trace: the number of accesses that don't fault in FIFO + accesses that don't fault in LRU = number of accesses that don't fault in optimal.

4.3 3 pts

LRU is a difficult algorithm to implement in practice, but there are ways to approximate its behavior (see section 10.4.5 of the book for a one way). Assume the hardware has only valid and dirty bits (i.e., no reference bits). How could you modify this algorithm to implement LRU? Why is this not done in practice?