1 We Interrupt This Program (10 pts.)

1.1 5 pts

Add disable_interrupts before line 29 and enable_interrupts after line 29. Count is the only variable that is modified by both sets of code, therefore we need to prevent readchar's decrement from being interrupted by the keyboard handler.

```
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

Remove lines 9 and 24, as the keyboard handler cannot be interrupted except by another interrupt (like timer). Unless the timer handler calls readchar, the following code is interrupt safe without disabling interrupts.

```
#define BUFSIZE 128

static int buf[BUFSIZE];
static int ins = 0, rem = 0;

void keyboard_handler() {
    int sc, inp;

    disable_interrupts();

    sc = inb(KEYBOARD_PORT);

    inp = ins;
    ins++;

    if (ins == BUFSIZE) {
        if (rem != 0) ins = 0;
        else ins = BUFSIZE - 1;
    }

    if (ins == rem) ins = rem - 1;
    else buf[inp] = sc;

    enable_interrupts();
    outb(INT_CTL_REG, INT_CTL_DONE);
}

char readchar() {
    int sc;

    if (ins == rem) return -1;

    sc = buf[rem];
    rem++;
    if (rem == BUFSIZE) rem = 0;

    return (char) process_scancode(sc);
}
```

2 Building a Question (10 pts.)

```
#define HARDHAT 0
#define HAMMER 1
#define BOTH 2

mutex_t flag;
cond_t cv[3];
int waiting[3];
int resource[2];

/*
 * 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) {
     int i;
     resource[HARDHAT] = hard_hats;
     resource[HAMMER] = hammers;

     mutext_init(&flag);

     for (i = 0; i < 3; i++) {
          waiting[i] = 0;
          cond_init(cv + i);
     }
}

/*
 * 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) {
     /* if resources availible then grant them */
     mutex_lock(flag);
     if ((resource[HARDHAT] - hh >= 0) &&
         (resource[HAMMER] - hammer >= 0) {
          resource[HARDHAT] -= hh;
          resource[HAMMER] -= hammer;
          mutex_unlock(flag);
          return;
     }
```

```
     /* else indicate waiting on correct set
        of resources and sleep */
     if (hh > 0) {
          if (hammer > 0) {
               waiting[BOTH]++;
               cond_wait(cv[BOTH], flag);
          }
          else {
               waiting[HARDHAT]++;
               cond_wait(cv[HARDHAT], flag);
          }
     }
     else {
          waiting[HAMMER]++;
          cond_wait(cv[HAMMER], flag);
     }
     mutex_unlock(flag);
}

/*
 * 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) {
     /* increment resources, then check ‘‘queues */
     mutex_lock(flag);

     resource[HARDHAT]+= hh;
     resource[HAMMER] += hammer;

     /* by checking for the resources and removing them now,
      * the signaled process can just wake up and leave
      * rather than looping to confirm that there are availible
      * resources.
      */
     if (resource[HARDHAT]) {
          if (resource[HAMMER] && waiting[BOTH]) {
               resource[HARDHAT]--;
               resource[HAMMER]--;
               cond_signal(cv[BOTH]);
          }
          else (waiting[HARDHAT]) {
               resource[HARDHAT]--;
               cond_signal(cv[HARDHAT]);
```

```
          }
     }

     if (resource[HAMMER] && waiting[HAMMER]) {
          resource[HAMMER]--;
          cond_signal(cv[HAMMER]);
     }

     mutex_unlock(flag);
}
```

3 Are we dead yet? (10 pts.)

3.1 3 pts

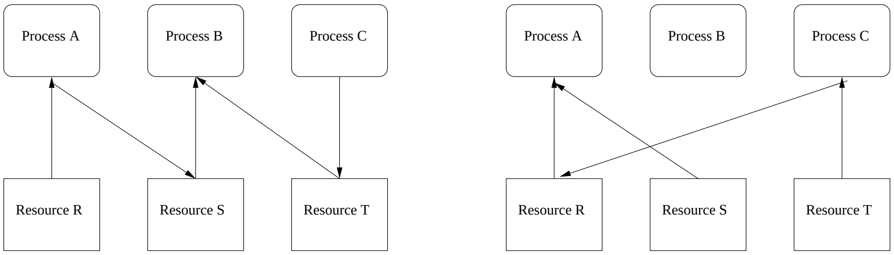
If, after the execution trace shown above, the next event that happens is process A: request(S), will the system be safe, unsafe or deadlocked? Why?

Unsafe, if the following op is process B: request(T) then the execution is safe (see 3.2 for explanation) and if its process C: request(T) then the execution will deadlock (see 3.3).

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?

Safe, B can finish execution releasing its resources. As A already has the resource in contention between it and C, it will then acquire its other resource, complete execution and exit. Leaving C to acquire resources and complete.



(a) After process B: request(T) (b) After process B: release(S)

Figure 1: 3.2 Resource Diagrams

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?

Deadlocked, eventually A, B, and C will make their second requests all of which are for resources which have already been granted thereby creating circular wait, ie deadlock.

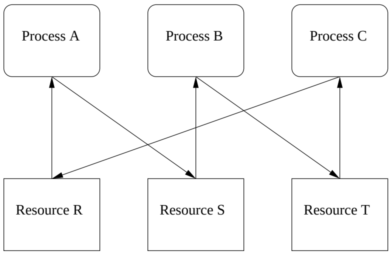


Figure 2: Following process C: request(T), the eventual state

4 Paging Algorithms (10 pts.)

4.1 2 pts

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

```
1e 08 0e 0b 0f 1e 08 0b 1b 1e 0b 0e 1e 1b 0f
```

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	1e	08	*	*	X	1e	08	*	*	X	1e	08	*	*	X
3	1e	08	0e	*	X	1e	08	0e	*	X	1e	08	0e	*	X
4	1e	08	0e	0b	X	1e	08	0e	0b	X	1e	08	0e	0b	X
5	0f	08	0e	0b	X	0f	08	0e	0b	X	1e	08	0f	0b	X
6	0f	1e	0e	0b	X	0f	1e	0e	0b	X	1e	08	0f	0b	
7	0f	1e	08	0b	X	0f	1e	08	0b	X	1e	08	0f	0b	
8	0f	1e	08	0b		0f	1e	08	0b		1e	08	0f	0b	
9	0f	1e	08	1b	X	1b	1e	08	0b	X	1e	1b	0f	0b	X
10	0f	1e	08	1b		1b	1e	08	0b		1e	1b	0f	0b	
11	0b	1e	08	1b	X	1b	1e	08	0b		1e	1b	0f	0b	
12	0b	0e	08	1b	X	1b	1e	0e	0b	X	1e	1b	0f	0e	X
13	0b	0e	1e	1b	X	1b	1e	0e	0b		1e	1b	0f	0e	
14	0b	0e	1e	1b		1b	1e	0e	0b		1e	1b	0f	0e	
15	0b	0e	1e	0f	X	1b	1e	0e	0f	X	1e	1b	0f	0e	

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 one way) Assume the hardware only has a valid and a dirty bit, how could you modify this algorithm to implement LRU? Why is this not done in practice?

Create a table to store the counters of when the page was last accessed, this also serves to confirm that a page is valid. Now mark all pages as invalid. The following is a modification to a generic page fault handler to implement LRU.

```
OnPageFault(addr)
     /* if the page is actually valid but marked
        otherwise so we can update counters to
        get LRU */
     if PageShouldBeValid(addr)
```

```
        /* update the counter for that page */
        UpdateCounter(addr)
        /* mark invalid the page we were accessing. */
        MarkInvalid(LastAccessed)
        LastAccessed = addr
        /* make the page we are currently accessing
          valid so we don’t immediately fault */
        MarkValid(LastAccessed)
    else
        /* Do Normal Page Fault Behaviour */
    return
```

This isn’t done in practice, because it would cause a page fault on each new page access, which is far too many page faults to be practical.