Question 1: Translation Look-ahead Buffer (8)

Which of the following runs fastest (assume page size is 4096 bytes)?

a) int i, j, npages = 10000;
   char *arr;
   arr = (char*)malloc(4096*(npages+1));

   for (i=0 ; i < npages ; i++) {
      for (j=0 ; j < 4096 ; j++) {
         arr[i+j] += 1;
      }
   }

b) int i, j, npages = 10000;
   char *arr;
   arr = (char*)malloc(4096*(npages+1));

   for (i=0 ; i < npages*4096 ; i++) arr[i] += 1;

c) int i, j, npages = 10000;
   char *arr;
   arr = (char*)malloc(4096*(npages+1));

   for (j=0 ; j < 4096 ; j++) {
      for (i=0 ; i < npages ; i++) {
         arr[i+j] += 1;
      }
   }

(a) (b) (c) (a and b) (a and c) (b and c) (no difference)
Question 2: Memory assignment (5)

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

<table>
<thead>
<tr>
<th>Process:</th>
<th>212KB</th>
<th>417KB</th>
<th>112KB</th>
<th>426KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>First fit:</td>
<td>500KB</td>
<td>600KB</td>
<td>500KB stuck*</td>
<td></td>
</tr>
<tr>
<td>Best fit:</td>
<td>300KB</td>
<td>500KB</td>
<td>200KB</td>
<td>600KB</td>
</tr>
<tr>
<td>Worst fit:</td>
<td>600KB</td>
<td>500KB</td>
<td>300KB stuck*</td>
<td></td>
</tr>
</tbody>
</table>

Best fit wins

* - waits for either the 500KB or 600KB chunk to free up

---

Question 3: Thrashing (5)

Consider a system with a page size of 4KB and suppose the four processes above are active in the system. If working set sizes are 50 percent of total size, how many frames are needed for these processes if thrashing is to be prevented?

total frames: $53 + 105 + 28 + 107 = 293$

working sets: $27 + 53 + 14 + 54 = 148$

Less than 148 frames runs the risk of thrashing
If page size is increased, what benefits and what suffers?

If page size is increased there will be fewer page faults which reduces time overhead spent waiting for I/O to disk. But internal fragmentation will be increased. However, smaller page size improves locality (matches locality of the process) which reduces I/O to disk. A larger page size means smaller page tables.

How does a multi-level page table work and how does it save memory?

The lowest level contains page table entries that translate virtual page numbers to page frame numbers and also contain flags such as dirty, reference, and access bits for control. Space is allocated for page tables in proportion to the amount used by running processes (empty parts of page table disappear - this is how it saves memory). An index into this space for a process is determined by a fixed number, say 10, of bits in the virtual address. Another, say 10, bits of the virtual space indexes into a page directory which contains references to the base addresses of the page tables. The base of the page directory is contained in a special register. For more than two levels the above repeats and the special base register refers to the topmost page directory.

How does an inverted page table work and what is so good about it?

Instead of one page table per process, the inverted table is a single page table containing page table entries for all pages in memory. The table is a hash table where the key is a combination of the process ID and the virtual page number. The table is much smaller than the corresponding normal page tables would be.
Question 5: Demand paging

Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

Let $p$ be the page fault rate (the probability that a memory access results in a page fault). Then $(1 - p)$ is the probability that a memory access costs 100 nsec. The probability that a page fault costs 20 msec is $0.7 \times p$ and the probability that a page fault costs 8 msec is $0.3 \times p$. Since 1 nsec = 1000000 msec,

$$(1 - p) \times 100 + 0.7 \times p \times 20000000 + 0.3 \times p \times 8000000 = 200$$

$$(14000000 + 2400000 - 100)p = 100$$

$$p = 100/(16400100) = 6.1 \times 10^{-6} = .0000061 = .00061\%$$
Question 6: mutex

```c
pthread_mutex_t lock;

void confirm () {
    pthread_mutex_lock(&lock);
    printf("A ");
    pthread_mutex_unlock(&lock);
}

void ping () {
    pthread_mutex_lock(&lock);
    printf("B ");
    confirm();
    printf("C ");
    pthread_mutex_unlock(&lock);
}

void *boy (void *d) {
    ping();
    pthread_exit(NULL);
}

void *girl (void *d) {
    ping();
    pthread_exit(NULL);
}

int main () {
    pthread_t g, b;

    pthread_create(&g, NULL, girl, NULL);
    pthread_create(&g, NULL, boy, NULL);

    pthread_exit(NULL);
}
```

What condition (of four) can be made false to prevent deadlock?

There exists a circular chain of threads such that each thread holds one more lock that is being requested by the next thread in the chain.

State a change to the code that causes that conditions to be false.

Introduce a second lock, call it `lock1`, and change `&lock` to `&lock1` in `confirm`. 
Question 7: Two-phase locking (10)

Is the following an example of two-phase locking? **No**

**Thread 1** | **Thread 2**
--- | ---
read_lock(Y); | read_lock(X);
read_item(Y); | read_item(X);
unlock(Y); | unlock(X);
write_lock(X); | write_lock(Y);
read_item(X); | read_item(Y);
X:=X+Y; | Y:=X+Y;
write_item(X); | write_item(Y);
unlock(X); | unlock(Y);

Is the following an example of two-phase locking? **Yes**

**Thread 1** | **Thread 2**
--- | ---
read_lock(Y); | read_lock(X);
read_item(Y); | read_item(X);
write_lock(X); | write_lock(Y);
unlock(Y); | unlock(X);
read_item(X); | read_item(Y);
X:=X+Y; | Y:=X+Y;
write_item(X); | write_item(Y);
unlock(X); | unlock(Y);

Describe the property that is guaranteed by two-phase locking

**Serializability**: a transaction schedule is serializable if its resulting memory state is the same as the result of executing the transactions sequentially: that is, without overlapping in time.
Question 8: Device drivers: interrupts (20)

Why are interrupts important with respect to device drivers?
I/O operations are typically slow and cost a lot of time to complete. Rather than have processors tied up waiting for an I/O task to complete, they are scheduled to work on other tasks. When an I/O operation completes, an interrupt alerts the OS and the OS proceeds to handle the result of the operation.

An interrupt is typically handled in two sections, called the “top half” and the “bottom half.” Why is that and what does each “half” accomplish (it is OK to mix up top and bottom - just state the concept)?

The interrupt should be handled with interrupts turned off but doing so may be too difficult on the system so handling is split into a very short, uninterruptible piece which sets up work for a work queue plus a more complex, interruptible piece that actually handles the work generated from the completion of the I/O operation such as transferring data to userspace.

An interrupt may be missed. What protection can be added to facilitate a graceful recovery in that event?

A kernel timer could be started prior to the operation that is designed to generate the interrupt. If the operation completes normally and an interrupt is generated, the timer is canceled. If the timer goes off, the interrupt has been missed and a routine that cleans up the mess is called to gracefully recover.

What can you do if your driver probes for an IRQ line using an OS call and the result returned is ambiguous?

Manually request an IRQ line from the OS. In Linux, it might look like this:

```c
irqreturn_t probeIt(int irq, void *dev_id) {
    if (the_irq == 0) the_irq = irq;  /* found */
    if (the_irq != irq) the_irq = -irq; /* ambiguous */
    return IRQ_HANDLED;
}

void probe (void) {
    int h;
    if ((h = request_irq(3, probeIt, SA_INTERRUPT, "probe", NULL)) < 0) {
        printk(KERN_INFO "Whoops\n");
        return;
    }

    the_irq = 0;  /* possible IRQ line */
    outb_p(0x10,base+2);  /* enable interrupt */
    outb_p(0x00,base);
    outb_p(0xFF,base);  /* toggle the bit */
    outb_p(0x00,base+2);  /* disable interrupt */
    udelay(5);  /* wait a bit */
    free_irq(h, NULL);  /* uninstall the handler */
    if (the_irq < 0) printk(KERN_INFO "Whoops\n");
}
```

Question 9: Device Drivers: barriers (15)

What does barrier do in the following (that is, what could go wrong if barrier were not put into this statement)?

```c
static inline void incr_bp(volatile unsigned long *index, int d) {
    unsigned long new = *index + d;
    barrier();
    *index = (new >= (buffer + PAGE_SIZE)) ? buffer : new;
}
```

The compiler might optimize the two lines into one like this:
```c
*index = (*index + d >= (buffer + PAGE_SIZE)) ? buffer : *index + d;
```
which, if interrupted, might lead to an unpredictable value for index.

Why is rmb put into the following piece of code?

```c
ssize_t readIt (char *buf, size_t count) {
    int retval = count;
    unsigned long port = 0x378;
    unsigned char *kbuf = kmalloc(count, GFP_KERNEL);
    ...
    insb(port, kbuf, count);
    rmb();
    ...
    return retval;
}
```
To make sure the port is fully read before continuing to process the result.

The following snippet is taken from a Mellanox Infiniband HCA low-level driver. The object sq is a work queue whose head is sq.head. The object sq.db is a “doorbell” interrupt that is used to notify the device that some work is to be done. The call to mthca_write64 raises the interrupt. Put wmb or rmb call(s) wherever you think they should go, if at all, and say why.

```c
qp->sq.head += MTHCA_ARBEL_MAX_WQES_PER_SEND_DB;
wmb(); /* set up the doorbell record first */
*qp->sq.db = cpu_to_be32(qp->sq.head & 0xffff);
wmb(); /* translate doorbell record to hw format */
mthca_write64(dbhi, (qp->qpn << 8) | size0, 
    dev->kar + MTHCA_SEND_DOORBELL, 
    MTHCA_GET_DOORBELL_LOCK(&dev->doorbell_lock));
/* finally ring the doorbell */