

Duration: 1 hour 15 min

Examiner: A. Goel

Please fill your student number and name below and then read the instructions below carefully.

**Student Number:**   

**Last Name:**       

**First Name:**      

### Instructions

**Examination Aids: No examination aids are allowed.**

Do not turn this page until you have received the signal to start.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the last blank page as scratch space. This page will not be marked.

This exam consists of 5 questions on 10 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 36.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

### MARKING GUIDE

Q1: \_\_\_\_\_ (4)

Q2: \_\_\_\_\_ (8)

Q3: \_\_\_\_\_ (10)

Q4: \_\_\_\_\_ (6)

Q5: \_\_\_\_\_ (8)

TOTAL: \_\_\_\_\_ (36)

**Question 1. Concurrency** [4 MARKS]

Suppose two threads execute the following C code concurrently, accessing shared variables *a*, *b*, and *c*:

```
// initialization code
int a = 4;
int b = 0;
int c = 0;

// Thread 1                                // Thread 2

if (a < 0) {                                b = 10;
    c = b - a;                                a = -3;
} else {
    c = b + a;
}
```

What are the possible values for *c* after both threads complete? You can assume that reads and writes of the variables are atomic, and that the order of statements within each thread is preserved in the code generated by the C compiler. Marks will be deducted for wrong or too many answers.

**Question 2. Synchronization** [8 MARKS]

To make laughing gas, two Nitrogen (N) atoms and one Oxygen (O) atom have to be fused. Let us synthesize this reaction by treating the atoms as threads and synchronizing them.

Each N atom invokes a procedure `nReady()` when it is ready to react, and each O atom invokes a procedure `oReady()` when it is ready. The procedures delay until there are at least two N atoms and one O atom present, and then the `oReady()` procedure calls a `Laugh()` procedure. After the `Laugh()` call, the two instances of `nReady()` and one instance of `oReady()` return.

To be fair, the N and the O atoms must fuse in the order they arrive. For example, N1 and N2 should fuse with O1, N3 and N4 should fuse with O2, and so on.

The code below shows a proposed solution for creating laughing gas. The `count` variable tracks the order in which the N atoms arrive. You may assume that the semaphore implementation enforces FIFO order for wakeups, i.e., the thread waiting longest in `P()` is always the next thread woken up by a call to `V()`.

```
Semaphore nwait = 0;
Semaphore owait = 0;
int count = 0;

nReady()
{
    count++;
    if (count % 2 == 1) {
        P(nwait);
    } else {
        V(owait);
        P(nwait);
    }
    return;
}

oReady()
{
    P(owait);
    Laugh();
    V(nwait);
    V(nwait);
    return;
}
```

**Part (a)** [2 MARKS] Unfortunately, this code has a race. Describe a case in which this code will not work.

**Part (b)** [6 MARKS] Below, you will try to fix the race. We have added the `lock()` code below. You need to add `unlock()` in the code below. Write within each box shown below, **one** of the following: 1) **unlock**, if the code needs an unlock there, 2) **deadlock**, if adding an unlock there may cause a deadlock, and 3) **incorrect**, if adding an unlock there may cause any other kind of incorrect synchronization.

```
Semaphore nwait = 0;  
Semaphore owait = 0;  
int count = 0;
```

```
Lock L;    // new code
```

```
nReady()
```

```
{  
    lock(L); // new code  
    count++;
```

```
←
```

```
    if (count % 2 == 1) {
```

```
←
```

```
        P(nwait);
```

```
←
```

```
    } else {
```

```
←
```

```
        V(owait);
```

```
        P(nwait);
```

```
←
```

```
    }
```

```
←
```

```
    return;
```

```
}
```

**Question 3. Scheduling** [10 MARKS]

There are three processes, A, B and C in the system. Process A is CPU-bound and always runnable. Processes B and C are IO-bound. After Process B runs for 1 unit of time, it needs to block for 1 unit of time. After Process C runs for 1 unit of time, it needs to block for 3 units of time.

**Part (a)** [6 MARKS] Show how the three processes are scheduled by a round-robin scheduler. The time slice of the scheduler is 1 unit of time. The first 12 time slices are shown in the three tables below. Show the process that is running in the “Running Process” table. Also, show the runnable processes in the “Ready Queue” table, and the blocked processes in the “Wait Queue” table, at each time slice.

Start by scheduling Processes A, B, and C, as shown below. Assume that a process becomes runnable just before the arrival of a timer interrupt. Also, assume that if multiple processes become runnable at the same time, then they are added to the ready queue in the order A, B, and C.

Running Process

1	2	3	4	5	6	7	8	9	10	11	12
A	B	C									

Ready Queue

head											
tail											

Wait Queue

head											
tail											

If there is a repeating pattern of running processes, circle the pattern in the “Running Process” table.

Roughly, what fraction of the CPU does each process take?

A =

B =

C =

**Part (b)** [4 MARKS] Show how the three processes are scheduled by a fixed priority scheduler. Assume Process C runs with the highest priority, Process B with medium priority, and Process A with the lowest priority. Start by scheduling Process C, as shown below.

Running Process											
1	2	3	4	5	6	7	8	9	10	11	12
C											

Ready Queue											
head											
tail											

Wait Queue											
head											
tail											

If there is a repeating pattern of running processes, circle the pattern in the “Running Process” table.

Roughly, what fraction of the CPU does each process take?

A =

B =

C =

**Question 4. Paging [6 MARKS]**

The figure below shows the address translation mechanism used by a 64-bit processor. Starting with a 48-bit virtual addresses, it uses 4 levels of page table to translate the address to a 52-bit physical address. Each page table entry is 8 bytes long.

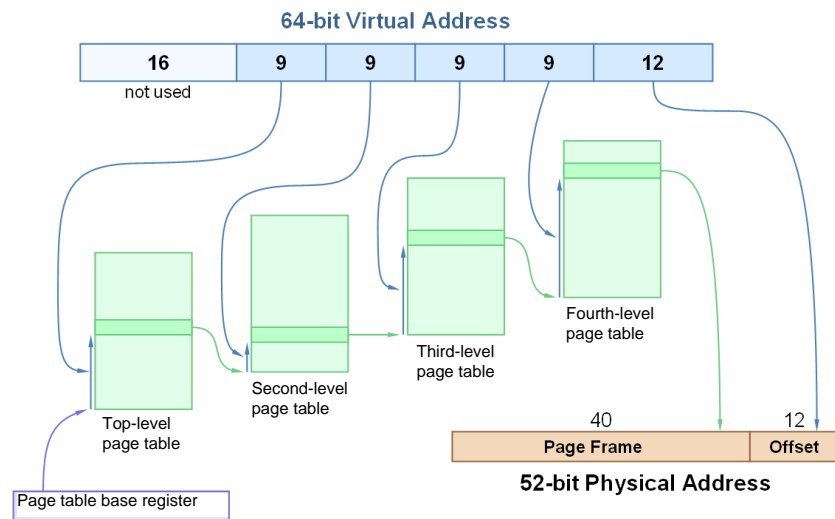


Figure 1: 4-level Page Table

If the page size is increased, the number of levels of page table can be reduced. How large must pages be in order to translate 48-bit virtual addresses with only a 2 level page table? Draw the address translation mechanism, similar to the figure above, corresponding to this page size.

**Question 5. TLB [8 MARKS]**

A 32-bit processor uses a two-level page table, with a page size of 4KB. It has two TLBs, an I-TLB (instruction TLB) that stores mappings for code addresses, and a D-TLB (data TLB) that stores mappings for data addresses. Both TLBs are fully associative (all TLB entries are looked up in parallel), and both have 64 entries. Consider the following code snippet:

```
char array[4096 * 64];

void simple() {
    int i = 0;
    int or = 0;
    int and = 1;

    for (i = 0; i < 4096 * 64; i++) {
        or = or | array[i];
    }
    for (i = 0; i < 4096 * 64; i++) {
        and = and & array[i];
    }
}
```

**Part (a)** [1 MARK] How many I-TLB misses will take place when the `simple()` function is run? Provide a justification for each TLB miss. State any assumptions that you make.

**Part (b)** [3 MARKS] What is the **minimum** number of D-TLB misses that will take place when the `simple()` function is run? Provide a justification for each TLB miss. State any assumptions that you make.



**Part (c)** [2 MARKS] Let us estimate the performance overhead of using the paging system. Assume that we have disabled the processor cache, so that every load and store instruction accesses memory (RAM). Assume also that there is no cost to accessing the TLB, and the compiler has performed no optimizations. Also, we will only consider accesses to `array` in the `simple()` function. How many memory accesses are performed to read `array`? How many memory accesses are performed to read the page table? Now estimate the overhead of the paging system. State your answer as a single number, possibly in terms of a power of 2.

**Part (d)** [2 MARKS] Suggest a way of changing the code in the `simple()` function to reduce the number of D-TLB misses. What is the number of misses with your code change? Provide a justification.

*[Use the space below for rough work.]*