CPSC 261 Midterm 2
Thursday March 17^{th}, 2016

[9] 1. Multiple choices

[5] (a) Among the following terms, circle all of those that refer to a responsibility of a thread scheduler:

Solution :

- Avoiding deadlocks
- Detecting race conditions
- Fairness
- Providing condition variables
- Starting and stopping threads

[4] (b) Amongst the following properties of a cache, circle all of those can be determined from its memory mountain.

Solution :

- Associativity
- Block size
- Total size
- Write policy (write-through or write-back)

[6] 2. When a memory access to an address a is performed, the bits of a are separated into three groups for the purpose of looking for the contents of memory address a in a cache. What role does each of these groups play in the access to the cache?

[2] a. The tag bits:

Solution : They are used to verify that the data retrieved from the cache is actually from the memory location that the CPU is trying to access.

[2] b. The index bits:

Solution : They select the cache set that contains the data the CPU is trying to access, if it is in the cache.

[2] c. The block offset bits:

Solution : They indicate which part of the data block (which one(s) of the memory addresses whose data it contains) is requested.

[8] 3. Thread synchronization

[2] a. What is the advantage of using condition variables instead of relying only on mutex locks to provide mutual exclusion between threads?

Solution : We avoid busy-waiting, which means the CPU can execute other threads and do useful work instead of looping for many clock cycles, waiting for the lock to become available.

[3] b. Are threads an example of soft or hard modularity? Justify your answer.

Solution : They are an example of soft modularity: threads share global variables and the heap, which means one thread can interfere with another thread's execution.

[3] c. In Dekker's algorithm, does the variable `turn` force the two processes to take turns accessing the critical section? Why or why not? The code for one of the processes is provided below (the other is symmetric).

Solution : No, it doesn't. It ensures the two processes will take turn IF they are both trying to access the critical section simultaneously. If one process is not attempting to access the critical section, then the other process can access it as many times in a row as necessary.

```
p0:
   entrance_intents0 = true;
   while (entrance_intents1) {
    if (turn ≠ 0) {
     entrance_intents0 = false;
     while (turn ≠ 0) {
      // busy wait
     }
     entrance_intents0 = true;
    }
   }
   // insert critical section here
   turn = 1;
   entrance_intents0 = false;
```

[7] 4. Consider the following array declaration

```
long a[10][512]
```

on a machine where `sizeof(long) = 8`. Assume that a CPU uses a cache with 512 **blocks**, and a block size of 32 bytes. For each cache configuration and array element listed, write down the number of the set (cache location) in which the array element would be stored if it were accessed by the program. Assume that in every case $a[0][0]$ ends up at the beginning of a block mapped to set 0. Hint: start by computing the number of sets and the length of each row of the array.

Solution :

Cache configuration	$a[0][8]$	$a[1][0]$	$a[3][0]$	$a[6][0]$
Direct-mapped	2	128	384	256
2-way set associative	2	128	128	0
8-way set associative	2	0	0	0
fully associative	0	0	0	0

[11] 5. Consider an implementation of a double-ended queue (a queue where elements can be inserted and removed from both ends) using a doubly-linked list:

```
typedef struct {
    long value;
    struct s_node *prev, *next;
} t_node;
typedef struct {
    struct s_node *first, *last;
} t_deque;
```

In order to allow this data structure to be accessed from multiple threads, a student decides to use two mutex locks `lockfirst` and `locklast`. He obtains the following code for the `insert_front` and `insert_back` functions:

```
1  void insert_front(t_deque *queue, long value) {
2     pthread_mutex_lock(&lockfirst);
3     t_node *newnode = make_node(NULL, queue->first, value);
4     if (queue->first != NULL) queue->first->prev = newnode;
5     queue->first = newnode;
6     pthread_mutex_unlock(&lockfirst);
7     pthread_mutex_lock(&locklast);
8     if (queue->last == NULL) queue->last = newnode;
9     pthread_mutex_unlock(&locklast);
10 }

11 void insert_back(t_deque *queue, long value) {
12    pthread_mutex_lock(&locklast);
13    t_node *newnode = make_node(queue->last, NULL, value);
```

```
14      if (queue->last != NULL) queue->last->next = newnode;
15      queue->last = newnode;
16      pthread_mutex_unlock(&locklast);
17      pthread_mutex_lock(&lockfirst);
18      if (queue->first == NULL) queue->first = newnode;
19      pthread_mutex_unlock(&lockfirst);
20  }
```

These two functions insert a new element at the front (back) of the queue.

[3] a. What is wrong with this implementation of the two functions `insert_front` and `insert_back`?

Solution : Suppose the queue is currently empty, and that separate threads call `insert_front` and `insert_last` simultaneously. Thread 1 first acquires lock `lockfirst` and creates node `Node1` while thread 2 acquires lock `locklast` and creates node `Node2`. By the time both functions return, `queue->first` and `queue->last` will point to two separate notes (which is fine) but these nodes will not point to each other (their `prev` and `next` fields will be NULL).

[3] b. Here is a version of the function `insert_front` that fixes the problem you explained in part (a). Why might it deadlock?

```
1   void insert_front(t_deque *queue, long value) {
2      pthread_mutex_lock(&lockfirst);
3      t_node *newnode = make_node(NULL, queue->first, value);
4      if (queue->first != NULL) queue->first->prev = newnode;
5      queue->first = newnode;
6      pthread_mutex_lock(&locklast);
7      if (queue->last == NULL) queue->last = newnode;
8      pthread_mutex_unlock(&locklast);
9      pthread_mutex_unlock(&lockfirst);
10  }
```

```
1   void insert_back(t_deque *queue, long value) {
2      pthread_mutex_lock(&locklast);
3      t_node *newnode = make_node(queue->last, NULL, value);
4      if (queue->last != NULL) queue->last->next = newnode;
5      queue->last = newnode;
6      pthread_mutex_lock(&lockfirst);
7      if (queue->first == NULL) queue->first = newnode;
8      pthread_mutex_unlock(&lockfirst);
9      pthread_mutex_unlock(&locklast);
10  }
```

Solution : The two locks `lockfirst` and `locklast` are acquired in opposite order by the two functions. So if function `insert_front` locks `lockfirst`

at the same time `insert_back` locks `locklast`, then each function will try to acquire the lock being held by the other function, and neither will ever be able to proceed.

[5] c. Suggest an improved implementation of the `insert_first` and `insert_last` functions that will not deadlock. Do not write code; simply explain briefly how you would modify the functions from part (b).

Solution : The idea is that both functions need to lock `lockfirst` and `locklast` in the same order, as they start, but release a lock if they detect they won't need to modify the corresponding field of the `t_deque` structure. This will allow insertions at the front and back of the deque to proceed almost completely in parallel except in the case where the deque was initially empty (in which case they will be done sequentially).

[9] 6. A CPU has a 4-way set associative ($E = 4$) cache, with 64-byte block size ($B = 64$), 32 sets ($S = 32$), and a least recently used replacement policy. Assume that `sizeof(long)` is 8 and that we have the following C declaration:

```
long a[16][256];
```

What will be the approximate miss rate for each of the following loops? Justify your answers!

[3] a.
```
for (i = 0; i < 16; i++)
    for (j = 0; j < 256; j++)
        sum += a[i][j];
```

Solution : Elements are accessed in the order `a[0][0]`, `a[0][1]`, `a[0][2]`, ... Since each cache block holds 8 elements, only the first access will be a cache miss. So we get a 12.5% miss rate.

[3] b.
```
for (j = 0; j < 256; j++)
    for (i = 0; i < 8; i++)
        sum += a[i][j];
```

Solution : The elements are accessed in column-major order. Each row of the array takes $8 \times 256 = 2048$ bytes, which corresponds to 32 cache blocks worth of data ($32 \times 64 = 2048$). Thus elements `a[0][0]`, `a[1][0]`, `a[2][0]`... all end up in the same set.

By the time the loop tries to access `a[0][1]`, the block containing `a[0][0]` to `a[0][7]` will already have been replaced in the cache by (that set will contain `a[4][0]` to `a[4][7]`, `a[5][0]` to `a[5][7]`, `a[6][0]` to `a[6][7]` and `a[7][0]` to `a[7][7]`. Hence we get a 100% miss rate.

[3] c.
```
for (j = 0; j < 256; j++)
    for (i = 0; i < 4; i++)
        sum += a[i][j];
```

Solution : The elements are once again accessed in column-major order. However, because we are only accessing four of the eight rows, the block block containing `a[0][0]` to `a[0][7]` will still be in the cache when the algorithm needs to access `a[0][1]` (along with `a[1][0]` to `a[1][7]`, `a[2][0]` to `a[2][7]` and `a[7][0]` to `a[7][7]`. Hence we get a 12.5% miss rate (once a block is loaded into the cache, it will not be evicted).