CPSC 261 Sample Midterm 2
March 2016

[10] 1. Short answers

[3] (a) Multiplexing and aggregation are two techniques used for virtualization. How do they differ?

Solution : Multiplexing uses a single resource to simulate multiple different ones. Aggregation is the opposite: users believe there is a single resource, but the work is divided between many.

[2] (b) How can the parameters of a cache (block size, set size) be modified without making the cache bigger to take better advantage of temporal locality in a program? Explain your answer.

Solution : We will want to use smaller (and hence more) blocks, as this will allow more separate pieces of information to be kept in the cache simultaneously. Thus we will increase the number of sets. Increasing the associativity would also help, although that's a double-edged sword since it might slow down the process of retrieving data from the cache.

[2] (c) How can the parameters of a cache (block size, set size) be modified without making the cache bigger to take better advantage of spatial locality in a program? Explain your answer.

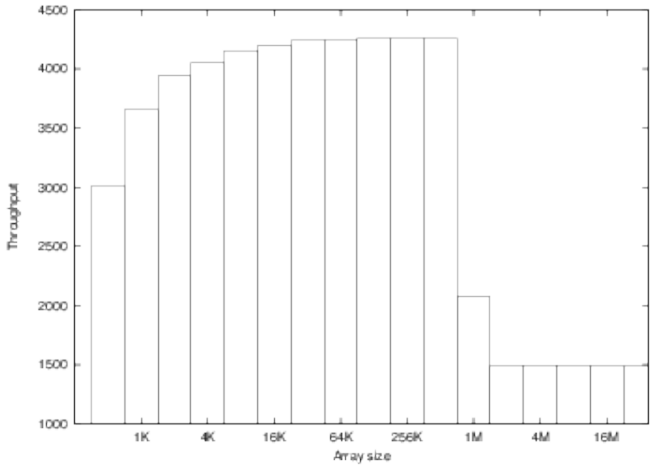
Solution : We will want to use larger (and fewer) blocks since, under the assumption that the program has excellent spatial locality, loading more data each time means fewer accesses to main memory.

[3] (d) What is a deadlock? Explain one situation where it might occur.

Solution : A deadlock happens when two processes are waiting for each other before proceeding. Neither process will be able to continue executing. A deadlock might occur, for instance, if two processes try to lock two resources A, B in opposite order. If one process locks A while the second one locks B, and then both try to obtain the resource the other process has locked, both will end up waiting forever.

[6] 2. A memory mountain was obtained for an unspecified CPU. In this question, you will be asked to identify two features of the CPU, given a "slice" of that memory mountain. You **must** justify your answers.

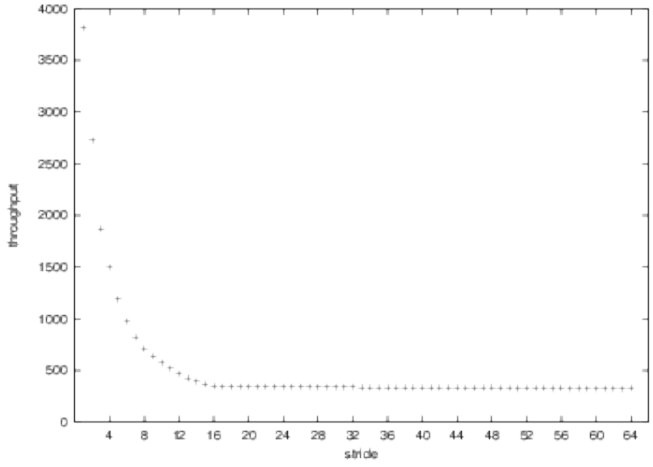
[3] a. The following figure shows throughput as a function of the array size, for a fixed stride.



The largest size shown in the figure is larger than the size of the CPU's largest cache. What is the approximate size of that largest cache?

Solution : The size of the largest cache is either 512KB or 1MB (this is where throughput drops off sharply). The L2 cache on this specific machine is 1MB; the drop in troughput for a 1MB array is most likely due to the fact that the memory mountain program was running on a multiuser system, and hence shared the cache with several other processes.

[3] b. The following figures shows throughput as a function of stride for an array that does **not** fit in the CPU's largest cache.



What is the approximate size of each cache line in bytes, knowing that `sizeof(int) == 4`?

Solution : Each cache line is 64 bytes long: once the stride is 16 or larger, the program uses only one word per cache line, and hence increasing the stride further does not change the throughput.

[10] 3. In this problem, we will consider a cache that is 4-way set associative ($E = 4$), with 4-byte block size ($B = 4$) and 8 sets ($S = 8$). Suppose that the CPU accesses memory 1 byte at a time, uses 17-bit addresses ($m = 17$), and that the cache initially contains the following data (all values are in hexadecimal, B0 through B3 refer to the four bytes in each block):

Set	Valid	Tag	B0	B1	B2	B3	Valid	Tag	B0	B1	B2	B3
0	1	5B3	CC	28	41	3D	1	6FA	45	CD	64	6D
	0	CFC	9E	38	19	C9	1	A45	E2	BB	2D	9F
1	1	126	AB	D9	85	CE	1	CC2	3D	84	F1	75
	1	431	7F	D5	A1	C1	0	1DA	26	5F	9B	5B
2	1	769	B6	41	96	67	0	59C	50	39	C8	48
	0	908	70	3E	0A	A4	1	FA9	25	FC	06	34
3	1	7A4	F5	20	7B	B7	0	602	CD	C3	C6	EF
	1	99C	01	C2	DE	8C	1	4F0	39	6F	16	6D
4	0	B29	79	BE	58	3D	1	A4E	65	58	3F	0E
	1	94F	6A	87	68	09	1	FD8	D8	D8	9C	F9
5	1	409	E3	49	28	1A	0	806	54	13	8A	9E
	1	AD9	94	F0	82	EF	1	650	75	CA	28	E3
6	0	F1C	1A	71	40	CD	0	22D	EA	3F	85	18
	1	506	A7	4C	88	C1	1	4BE	9A	17	D7	58
7	1	32D	F5	11	9A	26	1	C3B	3D	F7	20	9E
	1	84A	58	84	EB	46	1	4E4	2E	38	80	33

[2] (a) The following diagram shows the format of an address (one bit per box). Write in each box which field the bit belongs to: CO (the cache block offset), CI (the cache set index) and CT (the cache tag).

Solution :

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

[8] (b) The following table contains, for four **read** operations:

i. The binary representation of the address using 17 bits.
ii. Which set will be searched to find the data.
iii. The tag that will be compared against the cache lines' tags.
iv. Whether it's a cache hit or a cache miss.
v. If it's a cache hit, what value was read from the cache.

Fill in all entries in this table (except for the value read if the access for that row is a cache miss). A binary to hexadecimal conversion table is included on the last page of this test.

Solution :

Address (binary)	Set	Tag	Hit or Miss?	Value read
0 1100 0000 0100 1110	3	602	Miss	
0 1000 0110 0010 0111	1	431	Hit	C1
0 1010 1101 1101 1011	6	56E	Miss	
1 1000 1001 0111 00xx	4	C4B	Miss	

[13] 4. Threads Synchronization

[2] a. Under which conditions does a *race condition* occur? Be as precise as you can.

Solution : Race conditions occur when two or more threads try to access a resource simultaneously, and these accesses interleave. For example: thread 1 reads `a`, thread 2 reads `a`, then thread 1 writes a new value for `a`, and finally thread 2 writes a new value for `a`.

[2] b. In class, we discussed three correct implementations of our bounded buffer example: one that uses locks only, one that uses locks and condition variables, and one that uses locks and semaphores. What advantage do the second and the third implementations have over the first one?

Solution : They don't busy-wait: the threads do not spend a large amount of time looping and waiting for a condition to be satisfied.

[3] c. Consider the first implementation of our bounded buffer example:

```
while (buf->in - buf->out == N) {
    pthread_mutex_unlock(&lock);
    usleep(random() % LITTLESTALL);
    pthread_mutex_lock(&lock);
}
```

Why is the lock release before the call to `usleep` and reacquired afterwards?

Solution : If the lock wasn't released, then no other thread would be allowed to remove items from the buffer, which means this thread would be stuck forever waiting for the buffer not to be full.

[3] d. Consider now the second implementation of our bounded buffer example (this is taken from our `send` function):

```
while (buf->in - buf->out == N) {
    pthread_cond_wait(&forspace, &lock);
}
```

The call to `pthread_cond_wait` will terminate (and return to the `send` function) under two situations. Which ones?

Solution : It will terminate if either (1) no other thread is holding the condition variable or (2) a thread holding the condition variable calls

```
pthread_cond_signal(&forspace)
```

and it is the current thread's turn to acquire it.

[3] e. Why did we not write

```
if (buf->in - buf->out == N) {
    pthread_cond_wait(&forspace, &lock);
}
```

instead?

Solution : We can not assume that the condition `buf->in - buf->out == N` will be false when the call to `pthread_cond_wait` returns. It's possible that between the time the receiver has executed

```
pthread_cond_signal(&fordata);
pthread_mutex_unlock(&lock);
```

and the call to `pthread_cond_wait` returns, another thread has snuck in and filled the buffer again.

[6] 5. A CPU has a 2-way set associative ($E = 2$) cache, with 16-byte block size ($B = 16$), 8 sets ($S = 8$), and a least recently used replacement policy. Assume that `sizeof(int)` is 4 and that we have the following C declaration:

```
int a[4][32];
```

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

[3] a.

```
for (i = 0; i < 32; i++)
    for (j = 0; j < 4; j++)
        sum += a[j][i];
```

Solution : The elements are accessed in column-major order. The cache contains $8 \times 2 \times 16 = 256$ bytes, and each row of the array takes $4 \times 32 = 128$ bytes. By the time the loop will try to access (say) `a[0][1]`, the block containing `a[0][0]` to `a[0][3]` will already have been replaced in the cache by `a[2][0]` to `a[2][3]`, so we will get a 100% miss rate.

[3] b.

```
for (j = 0; j < 4; j++)
    for (i = 0; i < 32; i++)
        sum += a[j][i];
```

Solution : Elements are accessed in the order `a[0][0]`, `a[0][1]`, `a[0][2]`, ... Since each cache block holds 4 elements, only the first access will be a cache miss. So we get a 25% miss rate.