NATIONAL UNIVERSITY OF SINGAPORE

Semester 2 AY2018/19

CS5250 – ADVANCED OPERATING SYSTEMS

Time allowed: 2 hours

DRAFT OF ANSWERS

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **FIVE (5)** questions and comprises **SIXTEEN (16)** printed pages including this page.
2. This is an **OPEN BOOK** assessment.
3. Answer *all* questions. Note that the full mark for each question is different.
4. Write your answers on *this* **QUESTION AND ANSWER SCRIPT**. Answer only in the space (the box) given. Any writing outside this space will *not* be considered.
5. Fill in your Student Number with a pen, clearly on the front page and at least a few of the pages (at the top right and on all pages if possible) of this script.
6. You may use pencil to write your answers.
7. At the end of the assessment, please check to ensure that your script has all the pages properly stapled together.
8. Note that when a number is written as "0xNNNN" it means that "NNNN" is in base 16.
9. Approved calculators are allowed for this assessment.

For Examiner's use only

Q1	/15
Q2	/30
Q3	/20
Q4	/20
Q5	/15
TOTAL	/100

QUESTION 1 (15 marks)

(1a) The GNU Hash Bloom filter is uses the following code (as give in class):

- `H1 = dl_new_hash(symbol_name)`
- `H2 = H1 >> shift2`
- `N = ((H1 / C) % maskwords)`
- `BITMASK = (1 << (H1 % C)) | (1 << (H2 % C))`
- `bloom[N] |= BITMASK`
- `Test: (bloom[N] & BITMASK) == BITMASK`

Explain (i) why this is a k=2 Bloom filter, and (ii) how does it compare to the k=3 example given in class (slide 38 of Lecture Note set 4).

(10 marks)

ANSWER:

(i)

The bits at position H1 and H2 are the two hashed bits that are checked.

(ii)

The bit array (bloom[]) is accessed using a hash. So first hash is used to access the array, then the other 2 are used to check. So the hash is used 3 times, like in k=3 bloom filter using a single bit vector,

(1b) Explain the role of the global offset table (GOT) and procedure linkage table (PLT) in dynamic linking, especially at runtime. Why must some procedures be called via the PLT instead of directly in the code? (5 marks)

ANSWER:

The GOT is used for runtime resolution of external symbols, symbols whose final address in the virtual memory space is unknown at compile time. It is a table of addresses that will be fixed up by the runtime loader. The PLT contains code that will either call the dynamic loader (via the GOT) or the external routine directly to the routine (also via the GOT) once it has been resolved. Only local procedures whose addresses in the virtual memory space – either in absolute form or relative to some know location such as the current PC – can be called directly.

QUESTION 2 (30 marks)

(2a) Refer to the Linux load weight table given below:

```
static const int prio_to_weight[40] = {
    /* -20 */     88761,     71755,     56483,     46273,     36291,
    /* -15 */     29154,     23254,     18705,     14949,     11916,
    /* -10 */      9548,      7620,      6100,      4904,      3906,
    /*  -5 */      3121,      2501,      1991,      1586,      1277,
    /*   0 */      1024,       820,       655,       526,       423,
    /*   5 */       335,       272,       215,       172,       137,
    /*  10 */       110,        87,        70,        56,        45,
    /*  15 */        36,        29,        23,        18,        15,
};
```

Suppose there are 5, and only 5, tasks labelled A to E, at nice levels -2, -1, 0, 1, 2, respectively, in the system. Compute their respective share of CPU time. (10 marks)

ANSWER:

Total weight of system = 1586 + 1277 + 1024 + 820 + 655 = 5362.

Share of Task A: 1586 / 5362 = 30%

Share of Task B: 1277 / 5362 = 24.8%

Share of Task C: 1024 / 5362 = 19.1%

Share of Task D: 820 / 5362 = 15.3%

Share of Task E: 655 / 5362 = 12.2%

(2b) Compute the period for each task under Linux CFS under the same assumptions as in **Q2a**. (10 marks)

ANSWER:

Total weight of system = 1586 + 1277 + 1024 + 820 + 655 = 5362.

Assuming base if the default of 6ms, we have:

Period for Task A: 6 * 1586 / 5362 = 1.77ms

Period for Task B: 6 * 1277 / 5362 = 1.43ms

Period for Task C: 6 * 1024 / 5362 = 1.15ms

Period for Task D: 6 * 820 / 5362 = 0.92ms

Period for Task E: 6 * 655 / 5362 = 0.73ms

(2c) The actual **`vruntime`** update formula under Linux CFS is:

`vruntime +=` (physical time ran) * (weight of nice 0 task) / (weight of task)

Describe how the priority of a task influences the scheduling decision. (5 marks)

ANSWER:

This formula advances vruntime by a scaled value of the actual time the task ran. Because the weight of the higher priority tasks is larger, the actual increment in vruntime will be smaller for a higher priority task, giving it a higher chance of being to the left of the RB tree of tasks. Conversely, lower priority tasks will have their vruntime increment exaggerated in line with their weights, making them more likely to move to the right.

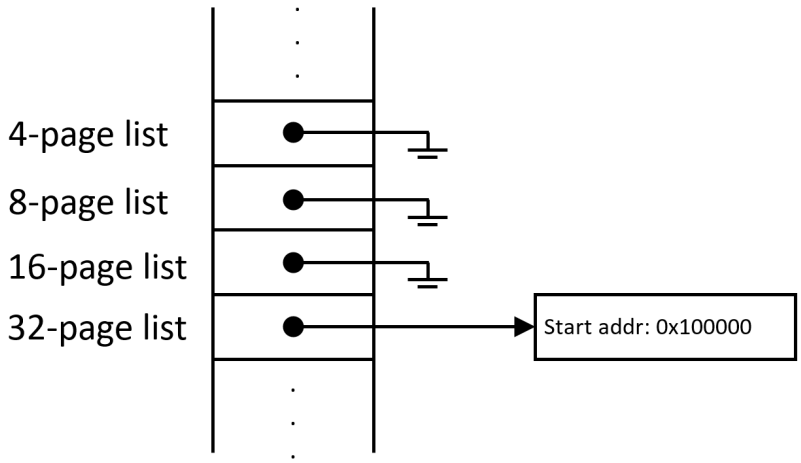
(2d) Suppose there are only two compute-only tasks in a single CPU system that runs indefinitely and never sleeps, one having the highest priority and the other having the lowest. How does CFS ensure that both tasks get a "fair" share of CPU time? In particular, will lower priority task ever to execute? (5 marks)

ANSWER:

The high priority task gets to run most of the time and for a long time. Each time it's current period expires, its vruntime gets updated. Meanwhile the vruntime of the low priority task remains the same. Eventually the vruntime of the high priority task will be greater than that of the low priority task – which gives the low priority task an opportunity to run, albeit for a short period. This way, even the lowest priority task will eventually get some (small) share of the CPU.

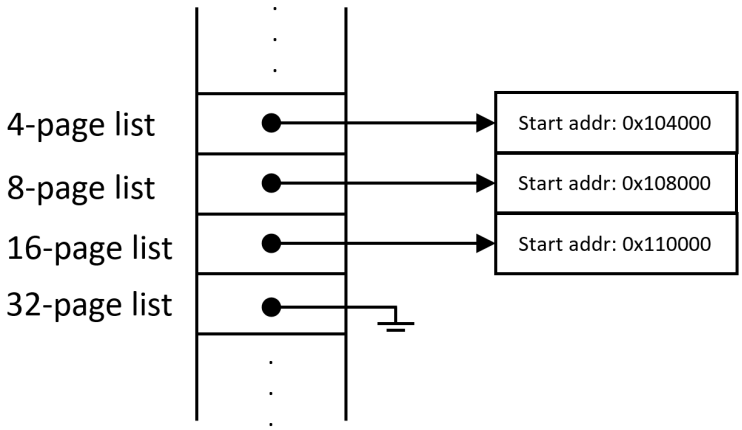
QUESTION 3 (20 marks)

(3a) Suppose the following is the current state of the buddy allocation list:



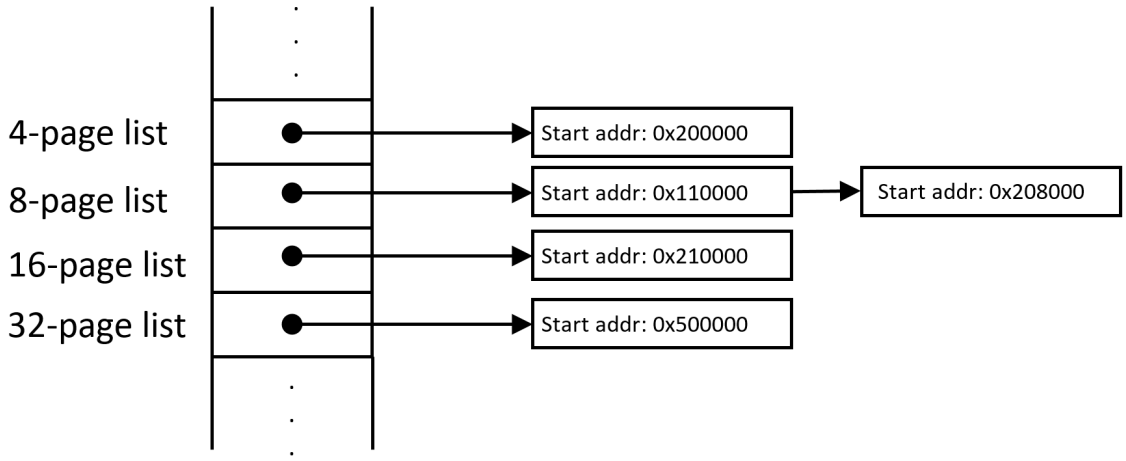
Draw the state of the lists after the allocation request for a block of 4 pages is satisfied. You are required to state the starting address of each block in the lists as well as give the starting address of the block returned. (5 marks)

ANSWER:



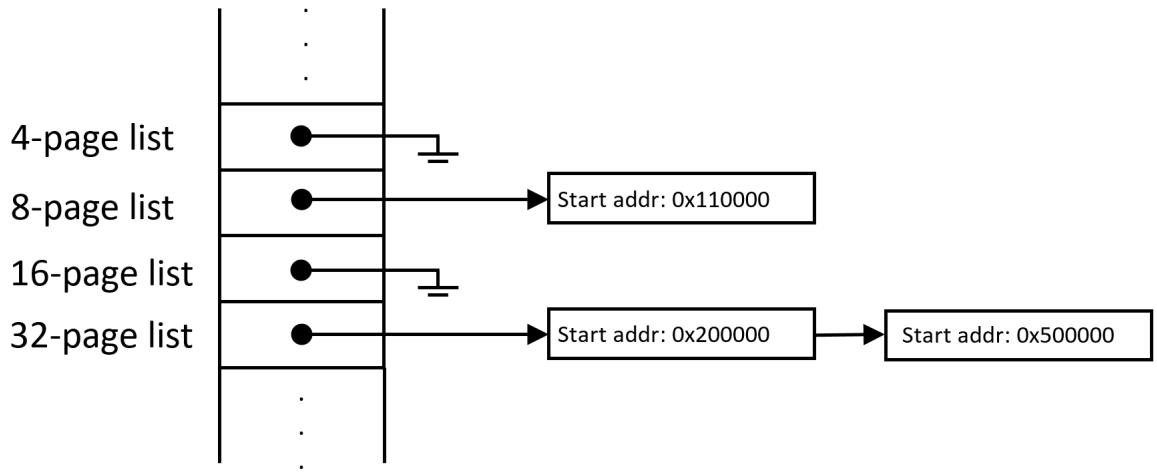
Starting address of returned block: 0x100000 (will also entertain 0x104000, 0x108000, 0x10c000 as correct answers since it is not specified which of the 4-page sub-blocks may be returned.

(3b) Suppose the following is the current state of the buddy allocation list:



Draw the state of the lists after a free request for a block of 4 pages with the starting address of **`0x204000`** is completed. You are required to state the starting address of each block in the lists. (5 marks)

ANSWER:



(3c) Under what circumstances is it (i) possible and (ii) impossible for a virtual address to miss in the TLB but at the same time hit in one of the data caches? (10 marks)

ANSWER:

(i)

If a data cache is large enough and the TLB is too small or busy, then an address could be cached into the data cache but its translation may not be cached in the TLB. What is important is that the translation via the page tables must be a valid one.

(ii)

What would be impossible would be a situation where the data is in the data cache, we have a TLB miss **AND** the translation via the page table is invalid. A page table translation is invalid means that the correspondence page(s) were never properly allocated in the process' virtual address space. If so, how can it be that valid data from a non-existent virtual page is cached?

QUESTION 4 (20 marks)

(4a) The size of largest file in ext2 format is actually limited by the 32-bit **`i_blocks`** field in the inode that represents the number 512-byte sector (confusingly also called "blocks") that a file can hold when in actual fact a data block in ext2 can be 1KB, 2KB, 4KB or 8KB. Now suppose we ignore the **`i_blocks`** field. What is the largest file that an ext2 file system using 4KB blocks can theoretically be, assuming pointers are 32-bits long? Show your working. (10 marks)

ANSWER:

The first 12 pointers point directly to 4KB blocks, i.e., the maximum would be 12 * 4KB = 48KB.

The 13th block points to a 4KB block of addresses. Since addresses are 32-bits or 4 bytes, there are 1024 pointers. Each of these points to a 4KB block. Hence we have 4096KB, or 4MB.

The 14th block has two levels of indirections. So there are 1024 * 1024 pointers to 4KB blocks, i.e., the maximum would be 4GB of storage.

The 15th block has three levels of indirections. So the maximum is $(1024)^3$ pointers to 4KB blocks, i.e., 4096GB of storage.

In total, we get (4096GB + 4GB + 4MB + 48KB) or well over 4TB for the maximum size of a file.

(4b) In a journaling file system, the journal is also written to the disk. So what happens if in the middle of writing the journal entry to disk, there is a failure? Explain how it is still possible for a journaling file system to recover itself. (5 marks)

ANSWER:

The key is that it must be possible to derive a consistent state of the filesystem during recovery. The operations to be executed on the file blocks are first recorded in the journal. Only after that is done successfully will the file blocks be changed. Suppose failure happens in the first step. Then the data in the file has not been changed and remains consistent. Suppose failure happens in the second step, then the journal is intact and the operations can be replayed to get the file blocks into a consistent state. If failure happens during the recovery process, the entire recovery process can simply be repeated.

(4c) Hard disks and SSDs guarantee that the write of a 512-byte sector is atomic. But a journal entry itself may be longer than 512-bytes. How can the atomic nature of the sector write be used to guarantee the integrity of journal entries? (5 marks)

ANSWER:

A journal entry will likely exceed 512-bytes. In order to guarantee the integrity of the entries, a special marker can be used to mark the end of an entry. If during the recovery process, an journal entry that has its end marker intact must have been completely written to the disk.

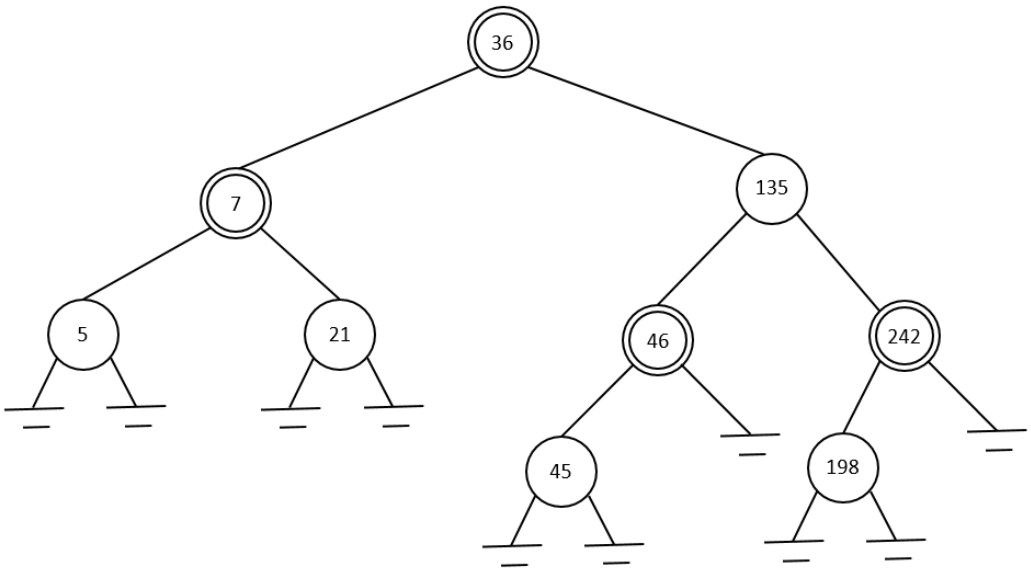
QUESTION 5 (15 marks)

(5a) Draw the final red-black tree after the following insertions are completed. Use a single circle to represent a "red" node and a double concentric circle to represent a "black" node. If you use other notations, make sure you give a legend.

- Insert 5
- Insert 135
- Insert 36
- Insert 46
- Insert 21
- Insert 242
- Insert 7
- Insert 198
- Insert 45

(10 marks)

ANSWER:



(5b) In Linux, exceptions are executed using the kernel stack in the user *process* context but interrupts are executed using the kernel's hard IRQ stack with a special context that is associated with a CPU rather than a process. What do you think are the design considerations for doing this?

(5 marks)

ANSWER:

Exceptions are errors generated usually from executing user code. As such, it makes sense to handle the exception quickly within the user context and send a signal to the same process.

=== END OF PAPER ===