香港中文大學

The Chinese University of Hong Kong

二０一九至二０年度下學期科目考試

Course Examination 2nd Term, 2019-20

科目編號及名稱 Course Code & Title :	CSCI 3150 Introduction to Operating Systems				
時間 Time allowed:	1	小時 hours	30	分鐘 minutes	
學號 Student ID			姓名 Student Name		

Instructions:

- Total marks: 100
- Open-book examination.
- Answer all questions.
- There are totally 7 pages (including this page).
- You can **put your answer in any form** (e.g. typing in this file, handwriting with other papers, etc.) but you **must submit your answer with your name and student ID** (in pdf, word, image, etc.) via **the final exam submission link in the Blackboard**.
- Multiple submissions are allowed. The last submission via the final exam submission link in the Blackboard will be used for grading.
- When the exam ends at 11:00, you will have 10 more minutes to submit your answer. **At 11:10, the submission link will disappear** (after that, no more submissions are allowed). Note that we will and can **only grade what you submitted in the Blackboard.**

1. MLFQ (30 marks)

Given the following MLFQ scheduling rules, process information and queue information , fill in the blanks of the scheduling result table.

(1) Rules:

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in a round-robin fashion using the time slice (quantum length) of the given queue.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
- Rule 6: Whenever a job is put into a new queue, which ***includes all cases in Rule 3 (a job enters the system), Rule 4 (a job moves down one queue) and Rule 5 (move all jobs to the topmost queue after time period S)***, sort all the jobs on that queue by pidnum, so the job with the smallest pidnum will be scheduled first.

(2) Process information:

ProcessNum 4
pidnum:123, arrival_time:50, execution_time:90
pidnum:13, arrival_time:70, execution_time:100
pidnum:1023, arrival_time:10, execution_time:160
pidnum:12, arrival_time:80, execution_time:28

(3) Queue information

QueueNum 3
Period_S 300
Time_Slice_Q3 10 Allotmenttime_Q3 30
Time_Slice_Q2 40 Allotmenttime_Q2 80
Time_Slice_Q1 60 Allotmenttime_Q1 120

Scheduling result table:

	Time-slot	**pidnum (Process ID)**	**Arrival Time**	**Remaining Time**
(1)	10 - 20	1023	10	150
(2)	20 - 30	1023	10	140
(3)	30 - 40	1023	10	130
(4)				
(5)				
(6)				

(7)				
(8)				
(9)				
(10)				
(11)				
(12)				
(13)				
(14)				
(15)				
(16)				
(17)				
(18)				
(19)				
(20)				
(21)				
(22)				
(23)				

2. Semaphore (25 marks)

In this question, you are required to **create 10 child threads** and use the **POSIX semaphore API** to achieve the same function with ***pthread_join()*** (i.e., the main thread waits until all child threads exit normally). Note that it is required that we only allow to have **at most 3 child threads to run** at the same time. You need to replace all ***/*YOUR CODE HERE*/*** (totally five places) with your own code.

```
#include <...>
// Hint: int sem_wait(sem_t *sem) /*P operation*/
// Hint: int sem_post(sem_t *sem) /*V operation*/
// Hint: sem_getvalue will get the current value of the semaphore
```

```
sem_t sem;
void thr_exit() {

  /*YOUR CODE HERE*/;

}
void thr_join() {
  int semValue = 0;

  while (/*YOUR CODE HERE*/) {
    sem_getvalue(&sem, &semValue); // Get the current value of sem
  }
}
void *child(void *arg) {
  int thr_id = /*YOUR CODE HERE*/;
  printf("child %d created and exited\n", thr_id);
  thr_exit();
  return NULL;
}
int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  // Hint: int sem_init(sem_t *sem, int pshared, unsigned int value)

  sem_init(/*YOUR CODE HERE*/);

  pthread_t p[10];
  int thr_idx[10];
  int i;
  for(i=0; i<10; i++) {
    thr_idx[i] = i;

    /*YOUR CODE HERE*/;

    // Hint: int pthread_create(pthread_t *p, const pthread_attr_t *attr,
    //             void* (*func)(void*), void *arg)
    pthread_create(/*YOUR CODE HERE*/);

  }

  thr_join();
  printf("parent: end\n");
  return 0;
}
```

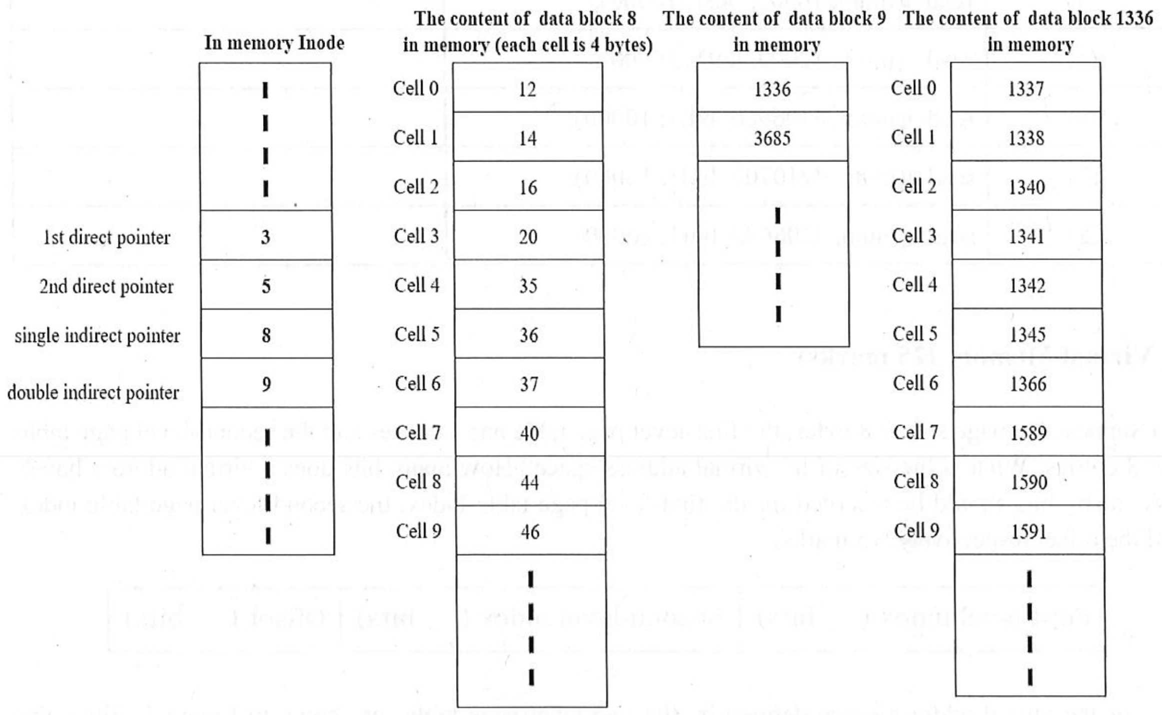
3. File System (20 marks)

In a simple file system called SFS, an inode is defined based on the following structure:

```
struct inode /* The structure of inode, each file has only one inode */
{
  int i_number; /* The inode number */
  time_t i_mtime; /* Creation time of inode*/
  int i_type; /* Regular file for 0, directory file for 1 */
  int i_size; /* The size of file */
  int i_blocks; /* The total numbers of data blocks */
  int direct_blk[2]; /*Two direct data block pointers */
  int indirect_blk; /*One indirect data block pointer */
  int double_indirect_blk; /*One double indirect data block pointer */
  int file_num; /* The number of file in directory, it is 0 if it is file*/
};
```

As shown above, an inode contains two direct data block pointers, one single indirect data block pointer and one double indirect data block pointer. Assume that each pointer is 4 bytes and the size of a data block is 4K bytes.

Suppose that there is a file with SFS and we have read the contents of its inode and related data blocks into the memory as shown below.



Here, each cell represents a 4-byte memory space and the decimal number inside is the unsigned integer stored correspondingly.

Answer the following questions by providing data block numbers in sequence that will be read from the disk (only data blocks that contain file data) when read_t (inum, offset, buf1, count) is called in a user program, where inum is the corresponding inode number for the above inode, buf1 is a pointer to a user-defined buffer, and the description of read_t() is listed below:

*int read_t(int inode_number, int offset, void *buf, int count);*

Description: read_t() attempts to read up to count bytes from the file starting at offset (with the inode number inode_number) into the buffer starting at buf. It commences at the file offset specified by offset. If offset is at or past the end of the file, no bytes are read, and read_t() returns zero. On success, the number of bytes read is returned (zero indicates the end of file), and on error, -1 is returned.

	read_t (inum, offset, buf1, count)	The data block numbers in sequence that will be read from (only list the data blocks that contain file data)
Example 1	read_t(inum, 100, buf1, 200);	3
Example 2	read_t(inum, 100, buf1, 5000);	3, 5
(1)	read_t(inum, 200, buf1, 1200);	
(2)	read_t(inum, 5000, buf1, 5000);	
(3)	read_t(inum, 6000, buf1, 10000);	
(4)	read_t(inum, 10000, buf1, 10000);	
(5)	read_t(inum, 12000, buf1, 20000);	
(6)	read_t(inum, 4206660, buf1, 10000);	
(7)	read_t(inum, 4210700, buf1, 10000);	
(8)	read_t(inum, 4206680, buf1, 20000);	

4. Virtual Memory (25 marks)

(1) Suppose the page size is 8 bytes, the first-level page table has 4 entries and the second-level page table has 8 entries. What is the size of the virtual address space? How many bits does a virtual address have? How many bits should be reserved for the first-level page table index, the second-level page table index and the offset respectively? (5 marks)

First-level index (__ bits)	Second-level index (__ bits)	Offset (__ bits)

(2) For the virtual address space defined by the two-level page tables as shown in Figure 1, given the following virtual addresses: **7, 68, 140, 231,**

which virtual addresses are mapped and if a virtual address is mapped, what is its corresponding physical address? (20 marks)

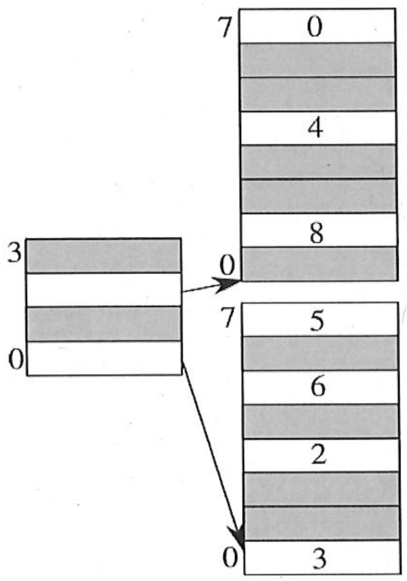


Figure 1. The two-level page mapping table, in which the numbers in entries are physical page frame numbers. Entries in gray color are not mapped.

*****The END****