香 港 中 文 大 學

The Chinese University of Hong Kong

二 0 二 0 至二一年度上學期科目考試

Course Examination 1st Term, 2020-21

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	1155116312		姓名 Student Name	Ng Chi Hun	

Instructions:

- Total marks: 100
- Open-book examination.
- Answer all questions.
- There are totally 12 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**. As a backup, you may send it via Email: cuhk.csci3150@gmail.com (optional).
- 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**. Similarly, for email submission, all emails after the deadline will be directly discarded.

1. (10 marks)

Fill in the missing statements in Area 1 to 8 in the C program shown below so as to implement the following commands:

ps aux | sort | tail -n 3 | grep midterm | wc -l

by dup() and pipe().

Notes:

(1) 0 is the standard input and 1 is the standard output;

(2) With the following statements:

int pfd[2];

pipe(pfd);

pfd[0] and pfd[1] are the read/write ends of the pipe, respectively.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
int main() {
  int fd;
  int p1_fd[2], p2_fd[2], p3_fd[2], p4_fd[2];
  pipe(p1_fd);
  pipe(p2_fd);
  pipe(p3_fd);
  pipe(p4_fd);
  if (fork() > 0){
    // Area 1 starts


    // Area 1 ends
    close(p1_fd[0]);close(p1_fd[1]);close(p2_fd[0]);close(p2_fd[1]);
    close(p3_fd[0]);close(p3_fd[1]);close(p4_fd[0]);close(p4_fd[1]);
    execl("/bin/ps", "ps", "aux", NULL);
  } else {
    if (fork() > 0) {
      /* The pipe between "ps" and "sort"*/
      // Area 2 starts


      // Area 2 ends

      /* The pipe between "sort" and "tail"*/
      // Area 3 starts


      // Area 3 ends
      close(p1_fd[0]);close(p1_fd[1]);close(p2_fd[0]);close(p2_fd[1]);
      close(p3_fd[0]);close(p3_fd[1]);close(p4_fd[0]);close(p4_fd[1]);
```

```
      execl("/bin/sort", "sort", NULL);
    } else {
      if (fork() > 0) {
        /* The pipe between "sort" and "tail"*/
        // Area 4 starts


        // Area 4 ends

        /* The pipe between "tail" and "grep"*/
        // Area 5 starts


        // Area 5 ends
        close(p1_fd[0]);close(p1_fd[1]);close(p2_fd[0]);close(p2_fd[1]);
        close(p3_fd[0]);close(p3_fd[1]);close(p4_fd[0]);close(p4_fd[1]);
        execl("/bin/tail", "tail", "-n", "3", NULL);
      } else {
        if (fork() > 0) {
          /* The pipe between "tail" and "grep"*/
          // Area 6 starts


          // Area 6 ends

          /* The pipe between "grep" and "wc"*/
          // Area 7 starts


          // Area 7 ends
          close(p1_fd[0]);close(p1_fd[1]);close(p2_fd[0]);close(p2_fd[1]);
          close(p3_fd[0]);close(p3_fd[1]);close(p4_fd[0]);close(p4_fd[1]);
          execl("/bin/grep", "grep", "midterm", NULL);
        } else {
          /* The pipe between "grep" and "wc"*/
          // Area 8 starts

          // Area 8 ends
          close(p1_fd[0]);close(p1_fd[1]);close(p2_fd[0]);close(p2_fd[1]);
          close(p3_fd[0]);close(p3_fd[1]);close(p4_fd[0]);close(p4_fd[1]);
          execl("/bin/wc", "wc", "-l", NULL);
        }
      }
    }
  }
}
```

2. (15 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 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). For the jobs arriving at the same time, schedule the job with **largest** pid first.
- 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 and will be at the tail of the target queue, which means it will be scheduled last).
- Rule 5: After some time period S, move all the processes in the system to the topmost queue, and sort all the jobs by pid. The job with the **largest** pid will be scheduled first.

Notes:

- Sorting for all jobs will happen every time it arrives the Period S.
- For sorting in Rule 3 and Rule 5, the job with the largest pid will be scheduled first.

(2) Process information:

ProcessNum 5
pidnum:45, arrival_time:10, execution_time:90
pidnum:323, arrival_time:70, execution_time:100
pidnum:2023, arrival_time:50, execution_time:150
pidnum:122, arrival_time:80, execution_time:28
pidnum:1242, arrival_time:80, execution_time:20

45 10 90
2023 50 150
323 70 100
122 80 28
1242 80 20

(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	45	10	80
(2)	20 - 30	45	10	70
(3)	30 - 40	45	10	60
(4)	40 – 80	45	10	20

1
2
3
1

(5)	80-90	2023	50	140	1
(6)	90-100	323	70	90	
(7)	100-110	122	80	18	
(8)	110-120	1242	80	10	
(9)	120-130	2023	50	130	2
(10)	130-140	323	70	80	
(11)	140-150	122	80	8	
(12)	150-160	1242	80	0	
(13)	160-170	2023	50	120	3
(14)	170-180	323	70	70	3
(15)	180-188	122	80	0	
(16)	188-208	45	10	0	2
(17)	208-248	2023	50	80	1
(18)	248-288	323	70	30	1
(19)	288-300	2023	50	68	
(20)	300-310	2023	50	58	
(21)	310-320	323	70	20	
(22)	320-330	2023	50	48	
(23)	330-340	323	70	10	
(24)	340 - 350	2023	50	38	
(25)	350 - 360	323	70	0	
(26)	360 - 398	2023	50	0	

A B C D

3. (15 marks)

Suppose that we have four processes, namely, A, B, C, and D, and their strides are 150, 100, 25, and 50, respectively. **Fill in the first 10 scheduling steps in the table based on stride scheduling** (With the same pass value, the scheduling priority order is A>B>C>D).

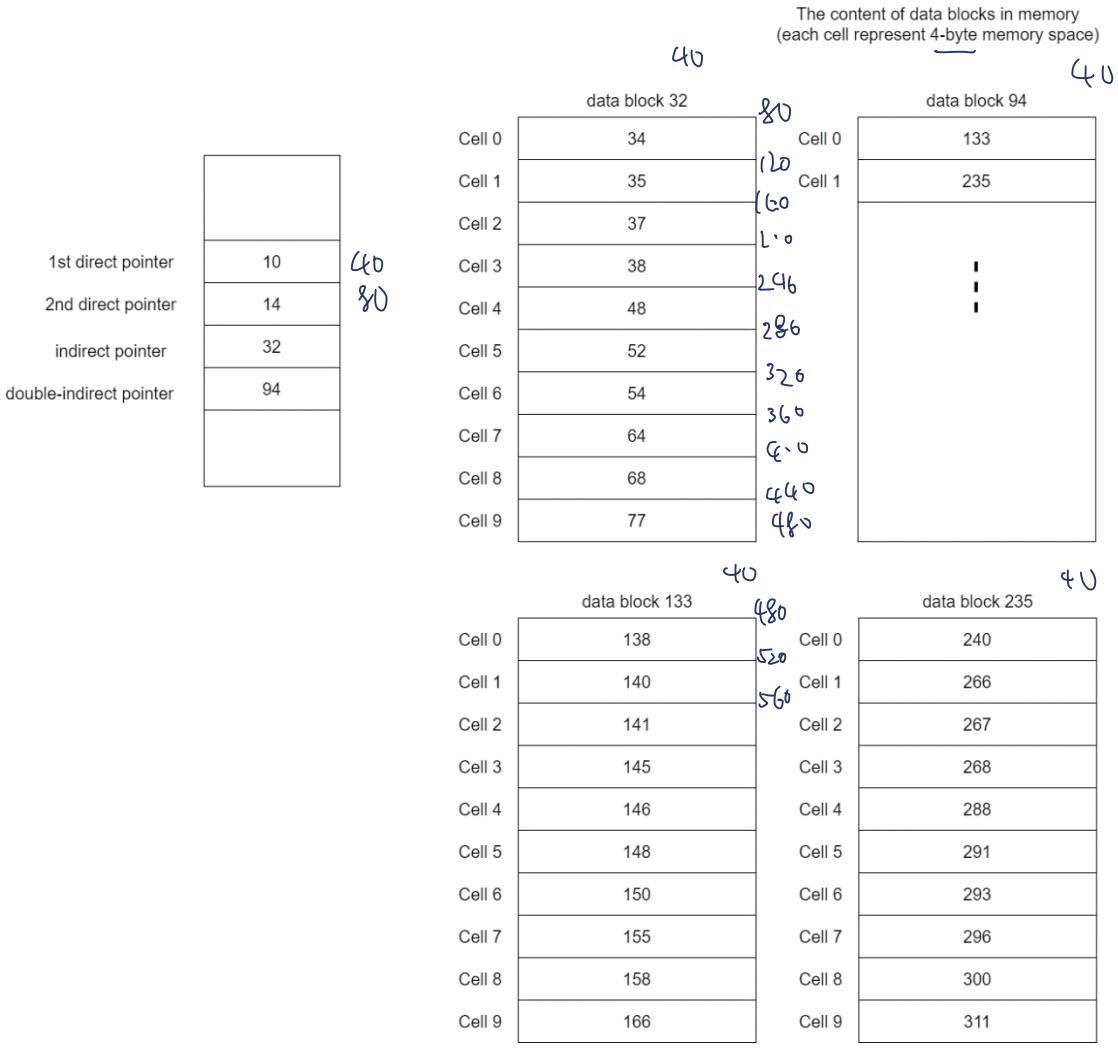
Step	Pass (A)	Pass(B)	Pass(C)	Pass(D)	The process to be scheduled
1	0	0	0	0	A
2	150	0	0	0	B
3	150	100	0	0	C
4	150	100	25	0	D
5	150	100	25	50	C
6	150	100	50	50	C
7	150	100	75	50	D
8	150	100	75	100	C
9	150	100	100	100	B
10	150	200	100	100	C

A
AB
ABC
ABCD
ABCDC
ABCDCD
ABCDCDC
ABCDCDCC
ABCDCDCCB

4. (20 marks)

In Small Simple File System (SSFS), we define the **BLOCK_SIZE** as **40 bytes**. Each inode in SSFS now contains two direct pointers, one indirect pointer and one ***double-indirect pointer.***

Suppose there is a file in SSFS and after we have read the contents of its inode and related data blocks into 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.

(a) What is the biggest size we can have for a file with SSFS?

$(2+10+(2\cdot 10))\cdot 40 = 1280B$

(b) Provide 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, and buf1 is a pointer that points to a user-defined buffer.

	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, 3, buf1, 23);	**10**
Example 2	read_t(inum, 13, buf1, 44);	**10,14**
(i)	read_t(inum, 22, buf1, 111);	10, 14, 34, 35
(ii)	read_t(inum, 47, buf1, 214);	14, 34, 35, 37, 38, 48
(iii)	read_t(inum, 237, buf1, 176);	38, 48, 52, 54, 64, 68
(iv)	read_t(inum, 414, buf1, 148);	68, 77, 138, 140, 141
(v)	read_t(inum, 535, buf1, 238);	140, 141, 145, 146, 148, 150, 155
(vi)	read_t(inum, 750, buf1, 116);	150, 155, 158, 166, 240
(vii)	read_t(inum, 817, buf1, 384);	158, 166, 240, 266, 267, 268, 288, 291, 293, 296, 300
(viii)	read_t(inum, 356, buf1, 635);	54, 64, 68, 77, 138, 140, 141, 145, 146, 148, 150, 155, 158, 166, 240, 267
(ix)	read_t(inum, 7, buf1, 510);	10, 14, 34, 35, 37, 38, 48, 52, 54, 64, 68, 77, 138, 140, 141

5. (20 marks)

Suppose the page size is 8 bytes, the first-level page table has 4 entries, the second-level page table has 8 entries and the third-level page table has 8 entries.

(1) 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, the third-level page table index and the offset respectively?

First-level index (2 bits)	Second-level index (3 bits)	Third-level index (2 bits)	Offset (3 bits)

(2) For the virtual address space defined by the page tables shown in the figure, which of the following virtual addresses are mapped? If the virtual address is mapped, what is the corresponding physical address? (The numbers in entries are physical page frame numbers. Entries in gray color are not mapped.)

Virtual addresses: 450, 671, 708, 840

First-level Page Table

0	101
	102
	103
3	

Second-level Page Tables

	PFN=101	PFN=102	PFN=103
0			107
		106	
	104		
	105		108
			109
7			

512

Third-level Page Tables

	PFN=104	PFN=105	PFN=106	PFN=107	PFN=108	PFN=109
0		62	23	80	49	43
	3	19	37			32
				48		64
3					77	

64

(1) 3rd level 4 entries (Page Size is 8 byte)
3rd level page map to 8.4 = 32 byte
2nd level 8 entries.
2nd level page table map to 8.32 = 512 byte
1st level page table map to 4.512 = 256 byte
∴ 1st level page is to level, virtual address space has 1024 byte
∵ we have 1024 VA = 2^{10} VA
∴ we need 10 bit for VA

(ii) Page Size is 8 byte
3rd level map 32 byte 2nd level map 256 byte
for VA = 450
VPN1 = 450 / 256 = 1 ∴ VA 7 map to PTOPFN=102,
offset within PT is 450 % 256 = 194, floor(194 / 64 = 3)
∴ VA=450 map to entry 3 which is not mapped

For VA = 671

VPN 1 = 671/256 = 2 which map to PTBPFN=103 (entry 2)

offset = 671%256 = 159, VPN2 = floor(159/32) = 4

∴ map to entry 4 which is PTBPFN=108

159%32 = 31 31/8 = 3 ∴ VA=671 map to PT[3] in PTBPFN=108

∴ Physical address is 77

VA = 708

VPN 1 = 2, → Mapped to PTBPFN=103

offset = 196 VPN2 = 6 → mapp to PTBPFN = 109

offset = 4 floor(4/8) = 0 ∴ VA = 708 map to PT[0] in PTBPFN = 109

Physical address is 43

VA = 840

VPN1 = 3 which is not map in Page Directory

Thus VA = 840 is not mapped

(3) For the following segments, fill in the third-level page tables to complete the page mapping.

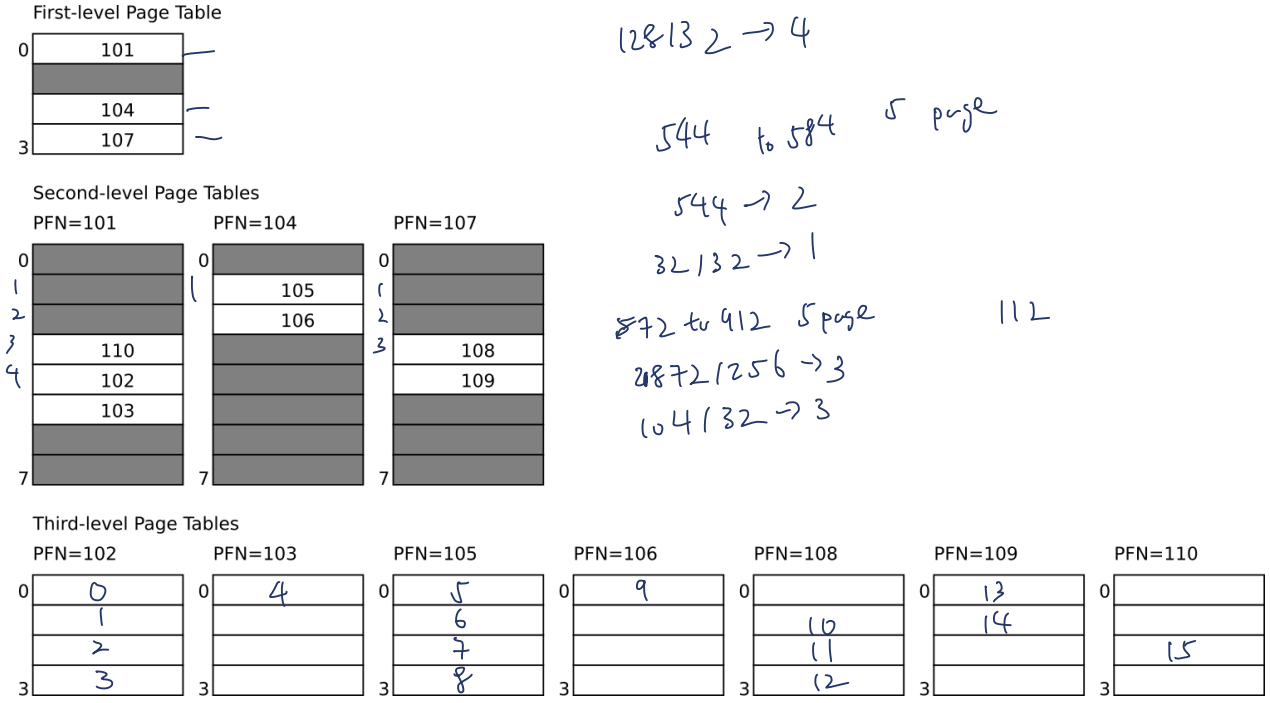
Segments:
Virtual [128, 168) to physical [0, 40)
Virtual [544, 584) to physical [40, 80)
Virtual [872, 912) to physical [80, 120)
Virtual [112, 120) to physical [120, 128)

128 to 168 = 5 page
128/256 → 0
128/32 → 4

544 to 584 5 page
544 → 2
32/32 → 1

872 to 912 5 page
872/256 → 3
104/32 → 3

112



6. (20 marks)

In this question, you are required to **implement read-write lock based on condition variable** in which multiple threads can acquire a read lock at the same time but write lock is exclusive. When a writer has acquired a write lock, all other writers or readers will be blocked until the writer finishes writing. Suppose that we only allow **at most 3 concurrent readers** at the same time. You need to fill in Area 1 to 5 in the following C program with your own code.

```
#include <pthread.h>

struct rwlock {
    pthread_mutex_t mutex;
    pthread_cond_t reader_cond;
    pthread_cond_t writer_cond;
    int readers, waiting_readers;
    int writers, waiting_writers;
};

void rwlock_init(struct rwlock* rwl) {
    pthread_mutex_init(&rwl->mutex, NULL);
    pthread_cond_init(&rwl->reader_cond, NULL);
    pthread_cond_init(&rwl->writer_cond, NULL);

    rwl->readers = rwl->waiting_readers = 0;
    rwl->writers = rwl->waiting_writers = 0;
}

void rwlock_read_lock(struct rwlock* rwl) {
    pthread_mutex_lock(&rwl->mutex);
    rwl->waiting_readers++;

    /* AREA 1 BEGIN */


    /* AREA 1 END */

    rwl->waiting_readers--;
    rwl->readers++;

    pthread_mutex_unlock(&rwl->mutex);
}

void rwlock_write_lock(struct rwlock* rwl) {
    pthread_mutex_lock(&rwl->mutex);
    rwl->waiting_writers++;

    /* AREA 2 BEGIN */


    /* AREA 2 END */

    rwl->waiting_writers--;
    rwl->writers++;
```

```
    pthread_mutex_unlock(&rwl->mutex);
}

void rwlock_read_unlock(struct rwlock* rwl) {
    pthread_mutex_lock(&rwl->mutex);
    rwl->readers--;

    if (/* AREA 3 */)
        pthread_cond_signal(&rwl->reader_cond);

    if (/* AREA 4 */)
        pthread_cond_signal(&rwl->writer_cond);

    pthread_mutex_unlock(&rwl->mutex);
}

void rwlock_write_unlock(struct rwlock* rwl) {
    pthread_mutex_lock(&rwl->mutex);
    rwl->writers--;

    /* AREA 5 BEGIN */


    /* AREA 5 END */

    pthread_cond_broadcast(&rwl->reader_cond);

    pthread_mutex_unlock(&rwl->mutex);
}
```

*******The END******