95

NAME Dang Nhi Ngoc Ngo - 01553277

EXAM #2 COMP 3080 OPERATING SYSTEMS
November 20, 2018

Print your name on this exam. Print (or write very clearly) your answers in the spaces provided on the exam....if you need more space use the back of the exam sheet and indicate in the primary answer space that you have used the back.

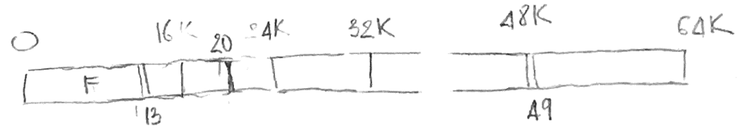
A collection of reference pages are provided as a separate handout. You do not have to pass these pages in UNLESS you have work on the pages that you want me to see, in which case, please reference this work in the primary answer space on the exam.

There are 6 questions with points as shown for a total of 100 points. Please keep your answers <u>BRIEF</u> and to the point.

20 POINTS

1. Using the buddy system of memory allocation, fill in the **starting addresses** for each of the following **memory allocation requests** as they enter an initially empty memory region which has a memory size of 2^{16} (64K) bytes. Addresses run from 0 to 64k -1, and can be given in K form (i.e. location 4096 = 4K.) Assume that when memory is allocated from a given block-size list, the available block of memory closest to address 0 (shallow end of memory) is always given for the request. Give the **address** of each allocation in the space provided below **if the allocation can be made**, or write in **"NO SPACE" if the allocation cannot be made** at the time requested.

TIME	JOB REQUESTING	JOB RETURNED	REQUEST SIZE(BYTES)
1	A		12K
2	B		3K
3	C		17K
4		A	
5	D		5K
6	E		4K
7		B	
8		D	
9	F		13K
10	G		2K
11		E	
12		C	
13		G	
14	H		15K



ANSWERS

Request A at 0

Request B at 16K

Request C at 32K

Request D at 24K

Request E at 20K

Request F at 0

Request G at 16K

Request H at 16K

15 POINTS

2. The following simple program (headers not shown) named `th_run` compiles and links (using –lpthread) with no errors, but on one particular execution on a multi-core Linux machine (like mercury) it produced the following output but **never completed** (i.e. no shell prompt ever prints again until a **ctl C** is typed in):

```
bash-3.00$ ./th_run
THREAD 1 IS RUNNING
THREAD 5 IS RUNNING
```

SOURCE CODE FOR th_run:

```
#define N 5
pthread_mutex_t lock;
void    *th(void *arg){
   pthread_mutex_lock(&lock);
   printf("THREAD %d IS RUNNING\n", *((int *)arg));
   return NULL;
} // end th

int  main(int argc, char *argv[]){
   pthread_t        thread_id[N];
   int              arg[N];
   int              i;
   pthread_mutex_init(&lock, NULL);
   pthread_mutex_lock(&lock);
   for(i=0; i<N; i++){
     arg[i] = i + 1;
     if(pthread_create(&thread_id[i],NULL, th,
                       (void*)(&arg[i]))!=0){
          perror("thread create failed ");
          exit(1);
     }
   }
   for(i=0; i<N; i++){
     pthread_mutex_unlock(&lock);
     pthread_join(thread_id[i], NULL);
   }
   printf("\nProgram with %d threads is done\n", N);
} // end main
```

Problem 2 continued on next page:

Problem 2 continued:

A. Provide a **detailed explanation** of why this program **never finishes**:

The program never finishes, because the thread in main execute is not unlocked, the new created thread also gains locked They remain locked and the threads will hang in join.

Join issue

The main thread is hung in join, since it joins in create sequence, but threads may finish in any sequence (here 1 is followed by 5). If it can't join, it can not unlock the lock for the next thread.

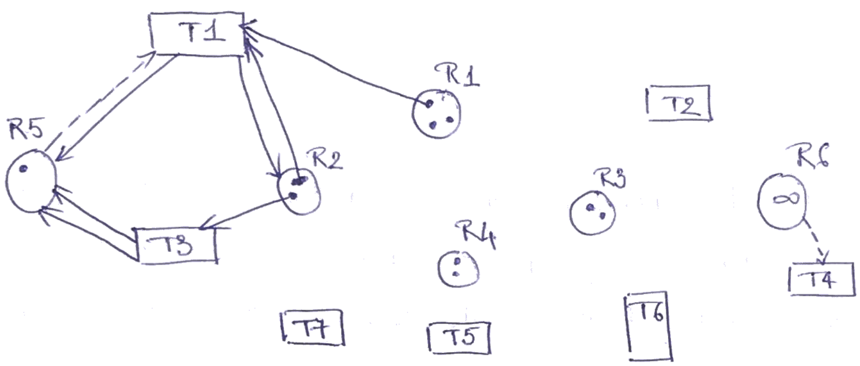
B. If we run this program **repeatedly**, could we ever expect a **particular execution** to complete ? **Explain:**

If we run this program repeatedly and before the new created thread runs, we could expect a particular exection to complete.

We must run it in creation order, and expect the thread is not locked.

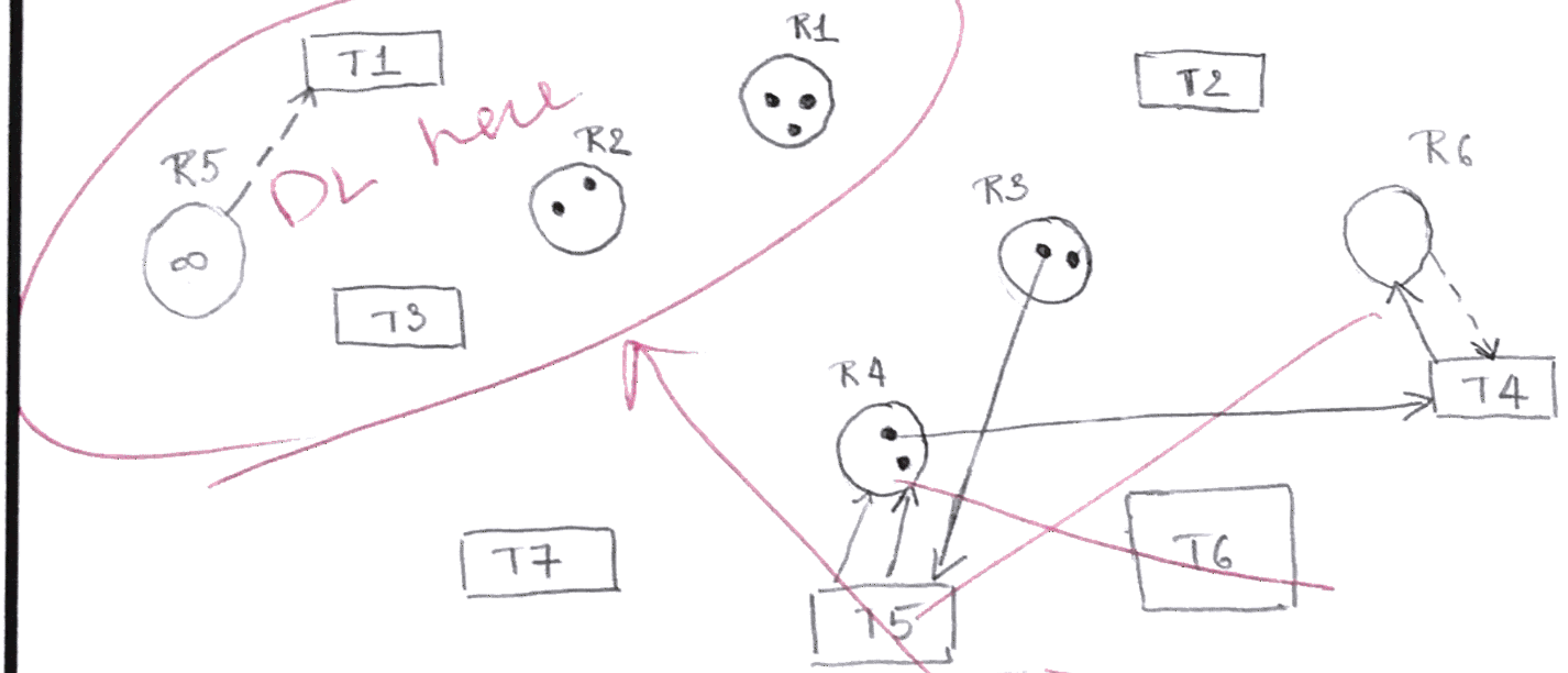
✓

Yes, if in a given run threads could finish in any order, we could expect that they would occasionally complete in create sequence, and satisfy the join loop.



There is DL between T3 and T1

DRAW THE FINAL, REDUCED RESOURCE GRAPH HERE AND STATE YOUR CONCLUSION ABOUT ANY DEADLOCK ON THE LINE BELOW:



YOUR CONCLUSION: ~~T4, T5~~ in deadlock

20 POINTS

4. The following information depicts a system consisting of 3 processes (**a, b, and c**) and **10 tape drives** which the processes must share. The system is currently in a **"safe"** state with respect to deadlock:

process	max tape demand	current allocation	outstanding claim
a	4	2	2
b	6	3	3
c	8	~~2~~ 4	~~6~~ 4

Following is a sequence of events each of which occurs a short time after the previous event with the first event occurring at time zero. The exact time that each event occurs is not important except that each is later than the last. I have marked the times **t(1), t(2),** etc. for reference. Each event either **requests or releases** some tape drives for one of the processes. If a system must be kept **"safe"** at all times, and if a request can only be met by providing all the requested drives, indicate the time at which each request will be granted using a **first-come-first-served** method for any processes that may have to wait for their request (i.e. request 5 granted at t(9)) or indicate that a request will not be granted any time in the sequential time listed. (Note: if a process releases some drives at time(x) which a waiting process needs, that waiting process will get its drives **at that time(x).** Put your final answers in the space provided below.

TIME	ACTION	
t(1)	request #1	c requests 2 drives
t(2)	request #2	a requests 2 drives
t(3)	release	a releases 3 drives
t(4)	request #3	b requests 3 drives
t(5)	request #4	a requests 1 drive
t(6)	release	b releases 4 drives
t(7)	request #5	b requests 1 drive
t(8)	release	a releases 1 drive

ANSWERS:

Request #1 granted at t(6)

Request #2 granted at t(2)

Request #3 granted at t(4)

Request #4 granted at t(5)

Request #5 granted at X

4 2 2 t1
6 3 3
8 2 6 R1 waits t1
3

4 4 0 t2
6 3 3
8 2 6 R1 waits t2
R2 OK t2
1

4 1 3 t3
6 3 3
8 2 6 R1 waits t3
4

4 1 3 t4
6 6 0
8 2 6 R1 waits t4
R3 OK t4
1

4 2 2 t5
6 6 0 R1 waits t5
8 2 6 R4 OK t5
0

4 2 2 t6
6 2 4
8 ~~2~~ 4 ~~6~~ 4 R1 OK t6
~~4~~ 2

4 2 2 t7
6 2 4
8 4 4 R5 waits t7
2

4 1 3 t8
6 ~~2~~ 3 ~~4~~ 3
8 4 4 R5 waits t8
~~3~~ 2

15. POINTS

5. The following problem deals with a virtual memory system with **an 18 bit address space (from 0 to 262,144 (256K) locations).** The system is byte addressable and uses an **8192 (8k) bytes per page** organization. The virtual memory, therefore, is organized into **32 page frames of 8k bytes** each for each process. For this system, the physical memory is configured with 16 real pages, with the operating system itself occupying the last 2 pages permanently, and all user programs paging against the **first 14 physical pages** as they run. Remember, the 18 bit address spaces will allow each user process to have a virtual address space of **32 pages** (256K bytes) even though only **14 real pages** (112K bytes) will be available for all running users to share during execution. The current status of this system is shown below for a time when 3 processes, **A, B and C**, are active in the system. **A is presently in the running state** while B and C are in the ready state. As you look at the current CPU registers, you can see that the **running thread in process A has just fetched a JUMP instruction** from its code path. The **PROGRAM COUNTER (PC)** value shown is the (binary) **VIRTUAL address** of the JUMP instruction itself, which is now in the INSTRUCTION REGISTER (**IR**), and the JUMP instruction shows a (binary) **VIRTUAL address to jump to** as it executes.

$2^5 = 32$

A. From what **REAL physical byte address** did the current JUMP instruction in the **IR** come from (i.e. what **physical address** does the **PC** point to) ? (You can give a <page, offset> combination or the single number actual address, but **use base 10 numbers** either way)

Give a base 10 answer ⟨10⟩⟨329⟩

B. To what **REAL physical byte address** will control be transferred when the current JUMP instruction executes ?? (Remember, **a page fault can occur** if a process thread references an invalid page, and faults are satisfied by connecting a virtual page to an available free physical page.) (Again, you can give a <page, offset> combination or the single number actual address, but **use base 10 numbers** either way).

Give a base 10 answer ⟨13⟩⟨658⟩

Tables on next page →

PHYSICAL MEMORY FRAME TABLE AND CURRENT PAGE TABLE FOR RUNNING PROCESS A

PHYSICAL MEM FRAME TABLE (MFT)	FRAME INDEX	PROCESS A PAGE TABLE VALID BIT	FRAME # (BASE 2)
OWNED BY A	0	0	NONE
OWNED BY B	1	0	NONE
OWNED BY C	2	0	NONE
OWNED BY C	3	1	00000
OWNED BY A	4	0	NONE
OWNED BY C	5	1	00100
OWNED BY A	6	0	NONE
OWNED BY A	7	0	NONE
OWNED BY C	8	0	NONE
OWNED BY A	9	0	NONE
OWNED BY A	10	1	01001
OWNED BY B	11	0	NONE
OWNED BY A	12	1	01100
FREE	13	0	NONE
OP SYS	14	0	NONE
OP SYS	15	0	NONE
	16	1	00110
	17	0	NONE
	18	0	NONE
	19	1	01010
	20	0	NONE
	21	0	NONE
	22	0	NONE
	23	1	00111
	24	0	NONE
	25	0	NONE
	26	0	NONE
	27	0	NONE
	28	0	NONE
	29	0	NONE
	30	0	NONE
	31	0	NONE

CPU

16 8 4 2 1 | 9 8 7 6 5 4 3 2 1 0

PC(BASE 2) 1 0 0 1 1 | 0 0 0 0 1 0 1 0 0 1 0 0 1

IR(BASE 2)
JUMP 0 1 1 1 0 | 0 0 0 1 0 1 0 0 1 0 0 1 0

9 7 4 1

a/ VP = 19 offset = 329

01010

PP = 10 offset = 329

$\langle 10 \rangle$ $\langle 329 \rangle$

b/ VP = 14 offset = 658

Page fault

Only free PP = 13

$\langle 13 \rangle$ $\langle 658 \rangle$

15. POINTS

6. This problem depicts a **memory allocation mechanism** that uses an embedded linked-list to manage an available heap space, just as you must implement for part of assignment #5. The **free block list head** contains the **byte location** (address) **of** the first available free block in the heap. Free block elements include an **embedded header** that consists of a **next** pointer field to point to the next free block, and a **byte size** field that defines the entire size of this free block (including the header fields). **Part A** and **Part B** both assume the **same initial state** of this space and are independent of each other (i.e., however you modify the list after completing Part A, you must assume that the list is back to the initial state shown before you do Part B).

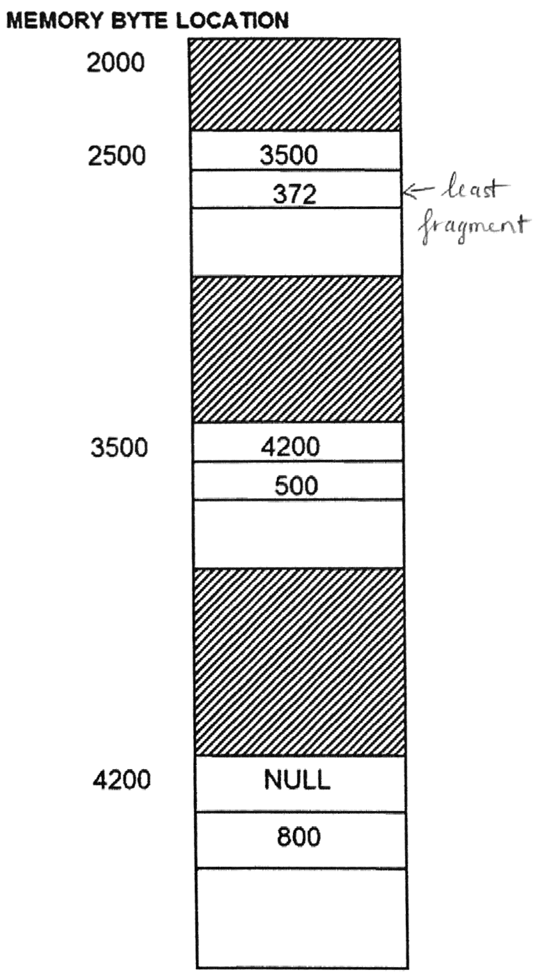
A. Given the initial state of the heap space shown, **fill in** the appropriate **free block list head** value, and **redraw** the organization of this space in the box provided, **after** an **allocation** of **350 bytes** has been made using the **BEST FIT** allocation algorithm.

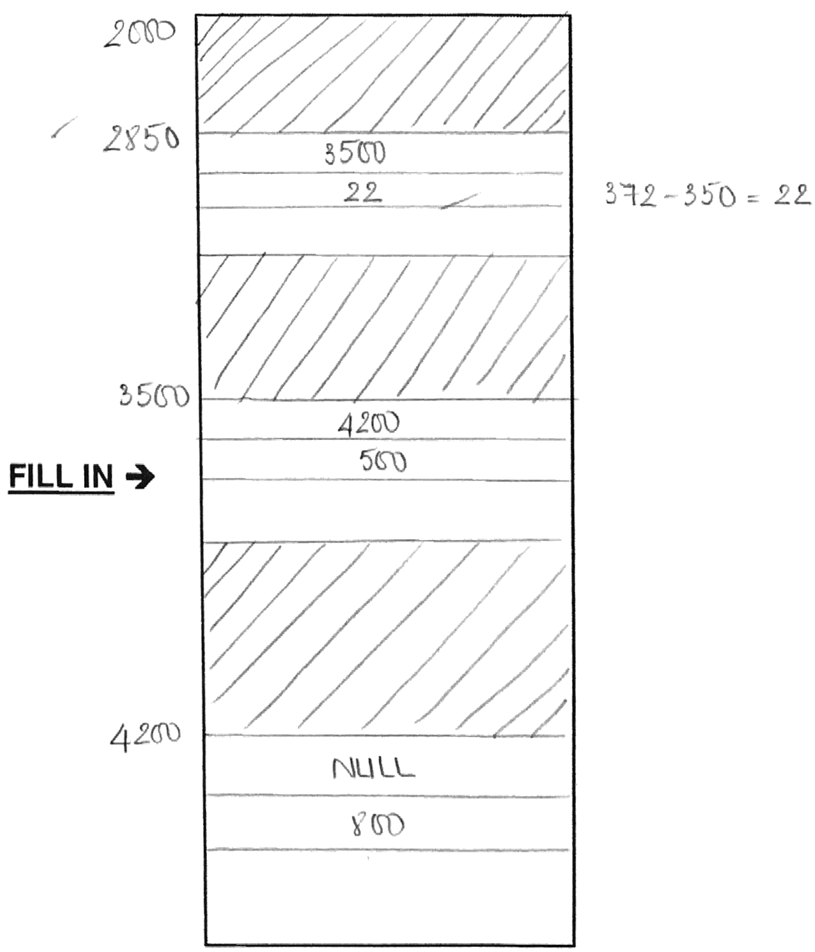
free block list head

2500

free block list head

2850 ← **FILL IN**





Problem 6 continued next page:

Problem 6 continued:

B. Given the initial state of the heap space shown, **fill in** the appropriate **free block list head** value, and **redraw** the organization of this space in the box provided, **after** a **free** operation of a previously allocated block of **528 bytes** is made at memory byte location (heap address) **2972**.

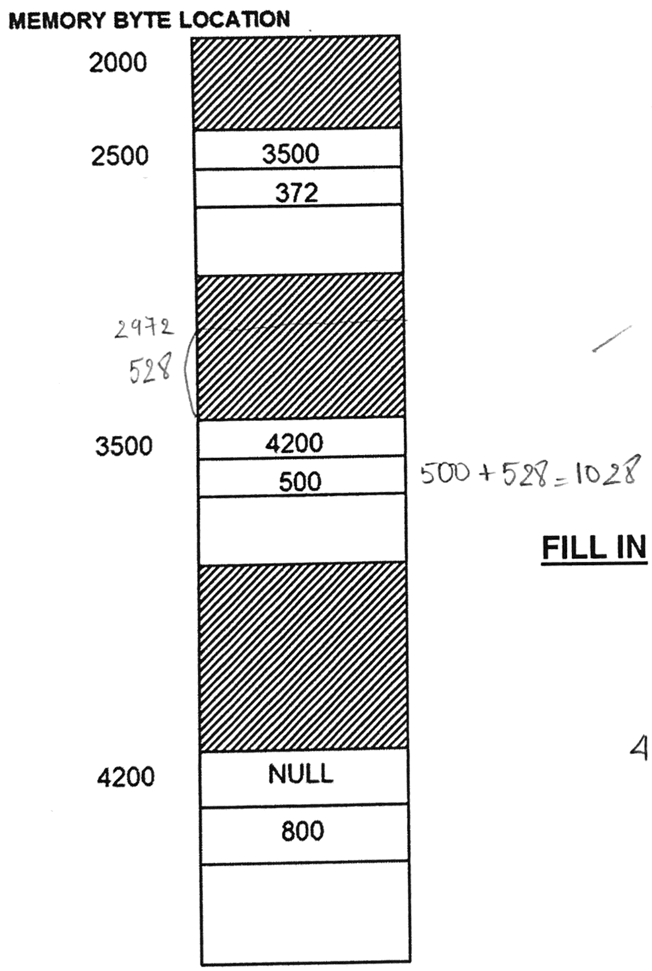
free block list head

2500

free block list head

2500

← **FILL IN**



FILL IN →