NAME Hoang Do 83

EXAM #2 COMP 3080 OPERATING SYSTEMS
April 16, 2019

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.

-4

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		7K
2	B		9K
3	C		13K
4	D		3K
5		A	
6	E		4K
7		B	
8	F		3K
9		D	
10	G		18K
11		E	
12		C	
13	H		24K
14	I		7K

ANSWERS

Request A at 0 K

Request B at 16 K.

Request C at 32K

Request D at 8 K

Request E at 12K.

Request F at 0K

Request G at No space

Request H at ~~4K.~~ 32

Request I at ~~32K.~~ 8

−3

15 POINTS

2. The code shown on the next page **compiles to an executable with no errors**, and has **no system call or library call errors** (the line numbers are included for reference in answering part B below). As the program begins normal execution, it runs the **main()** function in its initial thread (**IT**), where it initializes a global enumeration called **color** to the constant value **RED**. The IT then initializes a mutex, and a condition variable and creates **two new threads**. After creating the threads, the IT **safely changes** the color variable to **ORANGE** and **signals** the associated condition variable using the **pthread_cond_broadcast()** call to ensure that both threads can eventually awake from their condition waits and check their conditions (**pthread_cond_signal()** may only prepare one waiting thread to awake, so the broadcast version is used here instead). The **IT** then moves to a **join** call, waiting for the **two new threads** to finish, so it can print its final message and exit, but the **IT**, and thus the process, **never finishes**.

A. Show what **output is produced** by this process, based on the code provided:

~~RED~~

COLOR INITIALIZED TO RED ✓

orange → green

B. Although **some progress** is made in this process (producing the output you listed above in Part A) the process **never finishes**.

1. **Explain why the process never finishes** (even if some thread(s) do(es)).

The main thread will be hung in Join since it Join the creation sequence but ~~the~~ may finish in any sequence If the thread can't Join, it can unlock the lock for next thread.

Condition var is never signalled by th0, so th1 can not leave its cond wait. and it is stuck in Join

2. Using the line numbers included for reference, show **at what line** and **what specific code** you would add, to fix the logic problem, and allow the process to come to a **normal termination**.

Before return NULL at line 45 and 55 add.

pthread_cond_broadcast(&color_condX) ✓

pthread_cond_signal(&color_condx);

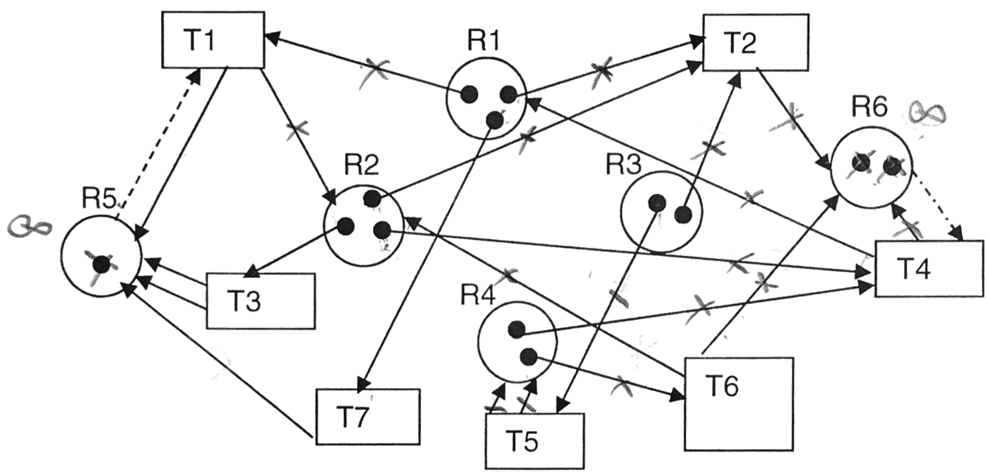
```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

enum COLOR {RED, GREEN, ORANGE} color;

pthread_t           thread_id[2];
pthread_mutex_t     color_lock;
pthread_cond_t      color_condx;

void *th0();
void *th1();

int  main(int argc, char *argv[])
{
        color = RED;
        printf("\nCOLOR INITIALIZED TO RED\n");
        pthread_mutex_init(&color_lock, NULL);
        pthread_cond_init(&color_condx, NULL);
        if(pthread_create(&thread_id[0], NULL, th0, NULL) != 0){
            perror("pthread_create failed ");
            exit(3);
        }
        if(pthread_create(&thread_id[1], NULL, th1, NULL) != 0){
            perror("pthread_create failed ");
            exit(3);
        }
        pthread_mutex_lock(&color_lock);
        color = ORANGE;
        pthread_mutex_unlock(&color_lock);
        pthread_cond_broadcast(&color_condx);
        pthread_join(thread_id[0], NULL);
        pthread_join(thread_id[1], NULL);
        printf("\nPROGAM COMPLETE\n");
        exit(0);
}

void *th0(){
        pthread_mutex_lock(&color_lock);
        while(color != ORANGE)
            pthread_cond_wait(&color_condx, &color_lock);
        color = GREEN;
        printf("\nCOLOR ORANGE CHANGED TO COLOR GREEN\n");
        pthread_mutex_unlock(&color_lock);
        return NULL;
}

void *th1(){
        pthread_mutex_lock(&color_lock);
        while(color != GREEN)
            pthread_cond_wait(&color_condx, &color_lock);
        color = ORANGE;
        printf("\nCOLOR GREEN CHANGED TO COLOR ORANGE\n");
        pthread_mutex_unlock(&color_lock);
        return NULL;
}
```

—5

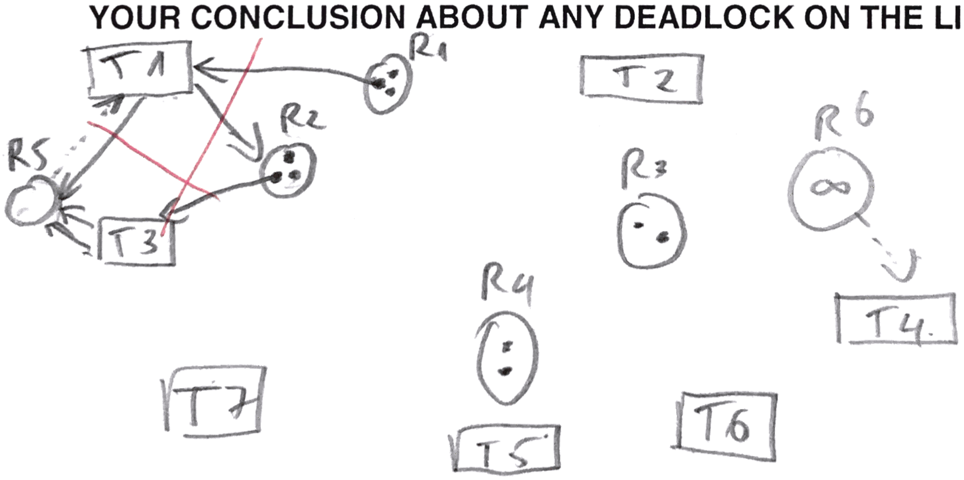
15 POINTS

3. The following resource allocation graph shows the state of a **7 thread** system using **6 types of resources** at a particular instant.

A. Using graph reduction, determine whether any deadlock exists, and if there **is deadlock** indicate the **process(es) and resources involved.** You must draw the **final reduced graph** whether or not there is a deadlock.



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



NO DL

YOUR CONCLUSION: Deadlock at T1 and T3.

B. What is the **complexity** of the graph reduction algorithm for the general system described above with **M** resources and **N** threads **(i.e. O(?))**. Is such an algorithm **more complex**, **less complex** or of the **same complexity** as an algorithm for a system of **re-usable resources only** ? **Explain.**

Complexity ???
General system: O(MN!) ; Graph reduction: O(MN)
There is more complex algorithm for a system of reusable resource only. because it reduces all ~~un~~ unnecessary resource already

20 POINTS

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

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

Following is a sequence of events, each of which occurs a short time after the previous event, with the first event occurring at time one (t(1)). The exact time that each event occurs is not important except that each is later than the previous. 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 threads. 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 threads that may have to wait for their requests (e.g. request 5 granted at t(x)), or indicate that a request will not be granted any time in the sequential time listed. (Note: if a thread releases one or more drives at time(x) that a waiting process needs, that waiting process will get its drives **at that time(x)** provided the system remains in a safe state. Put your final answers in the space provided below.

time	action	
t(1)	request #1	c requests 4 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 6 drives
t(7)	request #5	b requests 1 drive
t(8)	release	c releases 2 drives

4 2 2
6 3 3 t1 wait
8 2 6
3

4 4 0 t1 wait
6 3 3 t2 ok.
8 2 6
1

4 1 3
6 3 3 t1 wait
8 2 6
4

4 1 3 t1 wai
6 6 0 t3 ok
8 2 6
1

4 2 2 t1 wai
6 6 0 t5 ok
8 2 6
0

4 2 2
6 0 6 t1 ok
8 6 2
2

4 2 2
6 1 5 t5 wai
8 6 2
1

4 2 2
6 1 5
8 4 4
3

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 t(8)

15. POINTS

5. The following problem deals with a virtual memory system with **an 18 bit address space (from 0 to 262,143 (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.

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 <6><334>

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 <11><731>

Tables on next page →

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

PHYSICAL MEM FRAME TABLE (MFT)	PAGE #	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
FREE	11	0	NONE
OWNED BY A	12	1	01100
OWNED BY B	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	1	00110
	26	0	NONE
	27	0	NONE
	28	0	NONE
	29	0	NONE
	30	0	NONE
	31	0	NONE

$2^4\ 2^3\ 2^0$

```
                         CPU          8   6     3 2 1
PC(BASE 2) 1 1 0 0 1|0 0 0 0 1 0 1 0 0 1 1 1 0

IR(BASE 2)  4 3 2 1        9  7 6   4 3   1 0
      JUMP 1 1 1 1 0|0 0 0 1 0 1 1 0 1 1 0 1 1
```

a) ⟨6⟩⟨334⟩

b) Page fault

Free 11

⟨11⟩⟨731⟩

−5

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 **WORST FIT** allocation algorithm.

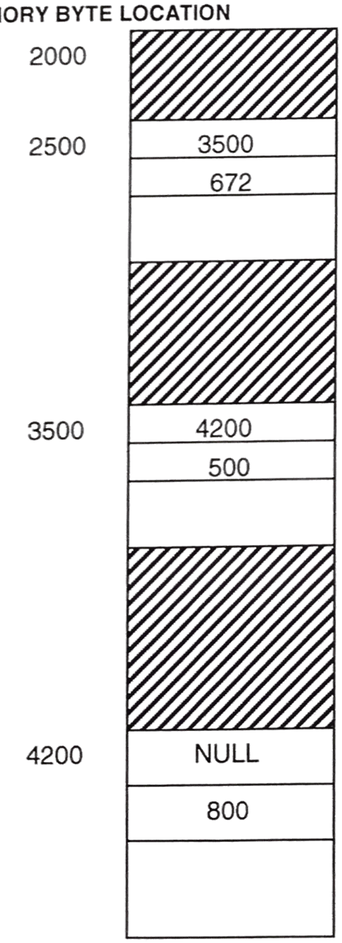
free block list head

2500

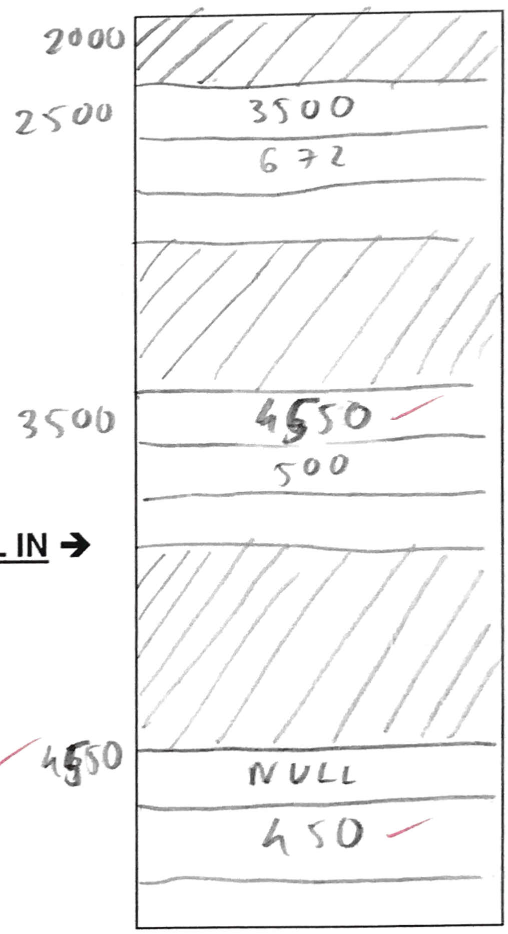
free block list head

2500 ← **FILL IN**

MEMORY BYTE LOCATION



FILL IN →



Problem 6 continued next page:

Problem 6 continued:

B. Given the initial state of the heap space shown, **fill in** the appropriate **<u>free block list head</u>** value, and **redraw** the organization of this space in the box provided, **<u>after</u>** a **<u>free</u>** operation of a previously allocated block of **<u>328 bytes</u>** is made at memory byte location (heap address) **<u>3172</u>**.

`free block list head`

2500

`free block list head`

2500 ← **<u>FILL IN</u>**