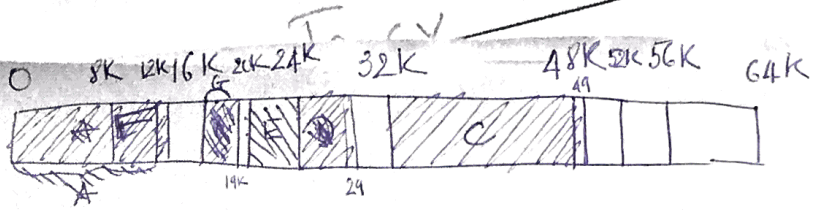


-6

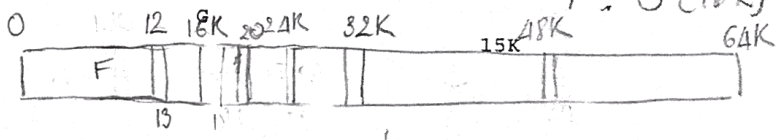
20 POINTS

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		

0 → 12
16 → 19
32 → 49
0 → 16 free

A:
B: 16k (4k)
C:
D: 24k (8k)
E:
F: 0 (16k)



ANSWERS

Request A at 0

Request B at 16k

Request C at 32k

Request D at 24k

Request E at 20k

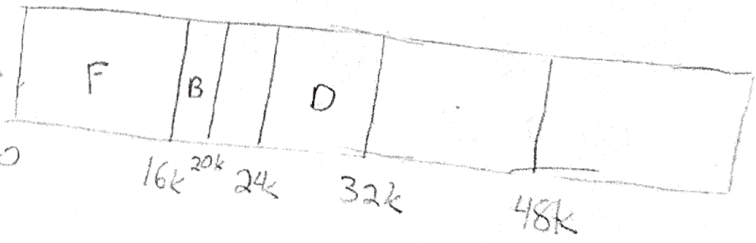
Request F at ~~16k~~ 0

Request G at ~~No Space~~ 16

Request H at ~~32k~~ 16 0

1024(1k)
2048(2k)
4096(4k)
8192(8k)
16384(16k)
32764(32k)
64k

20k
32k 48k



7/4

★ Final

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 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:

17/4

-5

Problem 2 continued:

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

The thread executing main is locked and then creates a thread, which also gains a lock. The thread however is never unlocked so it continues to ~~spin~~ indefinetly hang in join

B. If we run this program **repeatedly**, could we ever expect a **particular execution** to complete ? **Explain:** If we ran this more then once, enough times to unlock the newly created threads the first time it was ran, they would eventually all unlock and one process will finish running ? must run in creation order

T1.1

★ Final

-5

15 POINTS

3. Let ω = **2 3 1 3 2 4 3 2 4 5 1 6 7 5 6 7 4 5 6 7 2 1**, be a **page reference stream** for a given system. You are asked to work with **the Least Frequently Used algorithm** (referred to as the NFU algorithm in our book) below. You must determine the **number of page faults** that will occur for the stream shown above with an **LFU replacement algorithm** for a memory with **3 physical frames** and a memory with **5 physical frames**. (Please use the **grid help sheet** on the next page for this problem.)

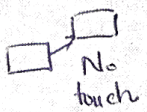
A. Assuming the primary memory is initially empty, how **many page faults** will the given reference stream have using the page replacement algorithm **LFU** for :

1. A memory with **3 physical frames** 17 ✓
2. A memory with **5 physical frames** 15

B. In our discussion of **memory objects**, we described some objects as being **anonymous** (e.g. stack objects) and some as being **file based** (e.g. text objects). **Explain how** the pages of an **anonymous** object are **managed differently** from a **file based** object with respect to **page replacement of dirty pages that must be backed up for possible re-use.**

7
0

Problem #3 continued on the next page →



Add new in stack or → re-organize based on which number will show up first in the future compare 1 pair

Problem #3 continued:

LRU just copy down

Grid Help Sheet

LFU How many times? 2 times 5 5>4
1 time 4

	X									X		X		X	X	X		X	X				
ω	2	3	1	3	2	4	3	2	4	5	1	6	7	5	6	7	4	5	6	7	2	1	
1	2	3	1	3	2	4	3	4	4	5	5	5	7	7	6	6	6	5	5	7	7	7	
2		2	3	1	1	2	2	1	1	4	4	4	4	4	4	4	4	4	4	4	4	4	
3			2	2	3	1	1	3	3	1	1	1	1	1	1	1	1	1	1	1	1	1	
4						3	4	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	
5										2	2	6	6	5	5	7	7	7	6	6	2	2	
6																							
7																							
c1																							
c2				1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	2
c3				1	1	1	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3
c4							1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
c5																							2
c6																							3
c7																							5
x	1	2	3	3	3	4	4	4	4	5	5	6	7	8	9	10	10	11	12	13	14	15	7

max # faults

★ Final

-4

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 happens a short time after the previous event, with the first event occurring at time zero. 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 this 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 request (i.e. 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 thread wants, that waiting thread 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.

10
(3/10)

TIME	ACTION	
t(1)	request #1 Y	a requests 1 drive (2/10)
t(2)	request #2 N	c requests 2 drives (2/10)
t(3)	release	b releases 1 drive (3/10)
t(4)	request #3 Y	a requests 1 drive (0/10)
t(5)	release	c releases 2 drives (2/10)
t(6)	release	a releases 1 drive (3/10)
t(7)	request #4 N	b requests 3 drives (3/10)
t(8)	request #5	c requests 2 drive (1/10)
t(9)	release	a releases 2 drives (3/10)

ANSWERS:

Request #1 granted at t(1)

Request #2 granted at ~~t(4)~~ 3

Request #3 granted at t(4)

Request #4 granted at X

Request #5 granted at t(8)

th	max	current	balance
a	4	1	3
b	6	2	4
c	8	4	4

4 3 1 +1
6 3 3
8 2 6 R1 OK +1
2
R2 waits +2
4 3 1 +3
6 2 4
8 4 2 8 4 R2 ok +3
1 3

4 4 0 +4
6 2 4
8 4 4 R3 OK +4
0
4 4 0 +5
6 2 4
8 2 6
2
4 3 1 +6
6 2 4
8 2 6
3

R4 waits +7
4 3 1 +8
6 2 4
8 4 4 R5 OK +8
1
4 1 3 +9
6 5 2 4 1
8 4 4 R4
0 3
waits +9

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 32 real pages, with the operating system itself occupying the last 6 pages permanently, and all user programs paging against the **first 26 physical pages** as they run. Remember, the 18 bit address spaces will allow each user process to have a virtual address space of **256K bytes** (32 pages) even though only 26 real pages 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 IP/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 < 20, 342 >

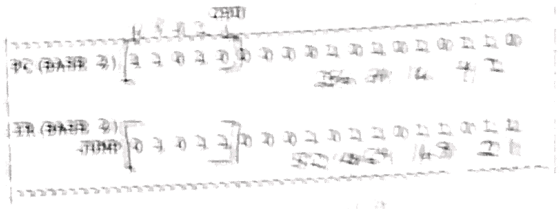
   B. To what **REAL physical byte address** will control be transferred when the current JUMP instruction executes ?? (Remember, **a page fault can occur** if a process thread references an invalid page, and faults are satisfied by connecting a virtual page to an available free physical page.) (Again, 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 < 19, 731 >

Tables on next page →

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

SYSTEM MEMORY FRAME TABLE (MFT)	PAGE #	PAGE TABLE FOR PROCESS A: VALID BIT	PAGE TABLE FOR PROCESS A: FRAME # (BASE 2)
OWNED BY A	0	0	NONE
OWNED BY B	1	1	10000
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	1	11001
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
OWNED BY C	13	0	NONE
OWNED BY C	14	0	NONE
OWNED BY C	15	0	NONE
OWNED BY A	16	1	00110
OWNED BY B	17	0	NONE
OWNED BY C	18	0	NONE
FREE	19	1	01010
OWNED BY A	20	0	NONE
OWNED BY C	21	1	10110
OWNED BY A	22	0	NONE
OWNED BY B	23	1	00111
OWNED BY C	24	0	NONE
OWNED BY A	25	0	NONE
OP SYS	26	1	10100
OP SYS	27	0	NONE
OP SYS	28	0	NONE
OP SYS	29	0	NONE
OP SYS	30	0	NONE
OP SYS	31	0	NONE

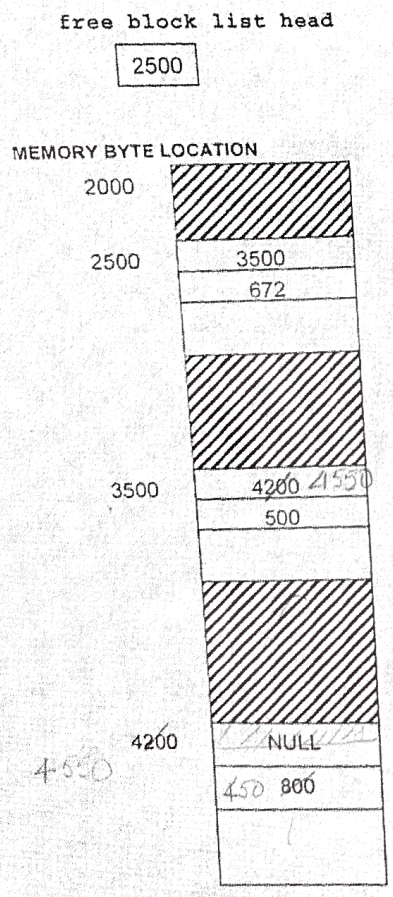


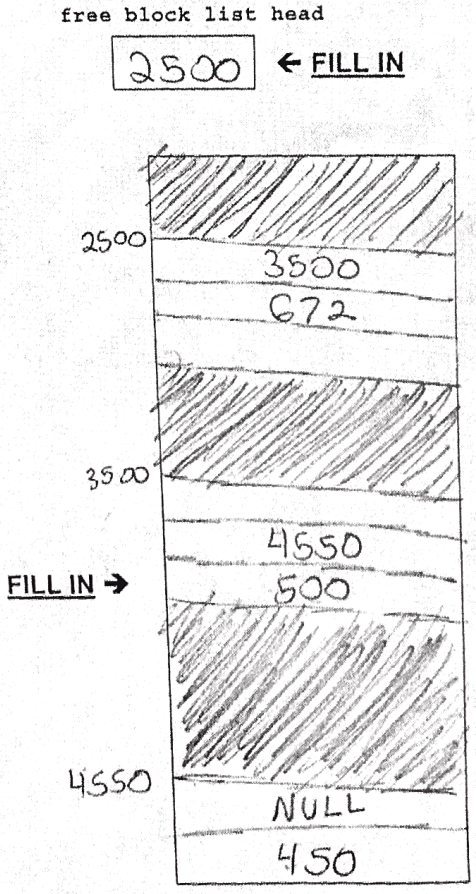
NAME

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.





Problem 6 continued next page:

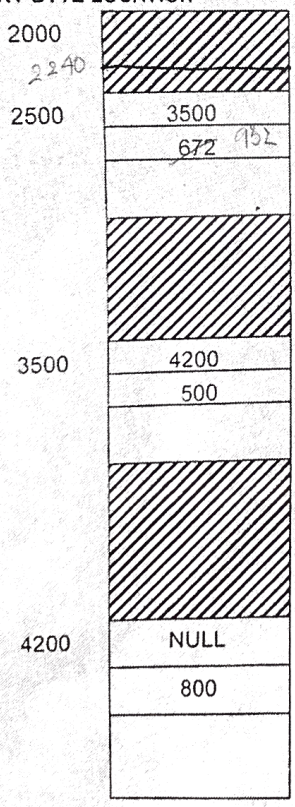
Problem 6 continued:

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 **260 bytes** is made at memory byte location (heap address) **2240**.

free block list head

2500

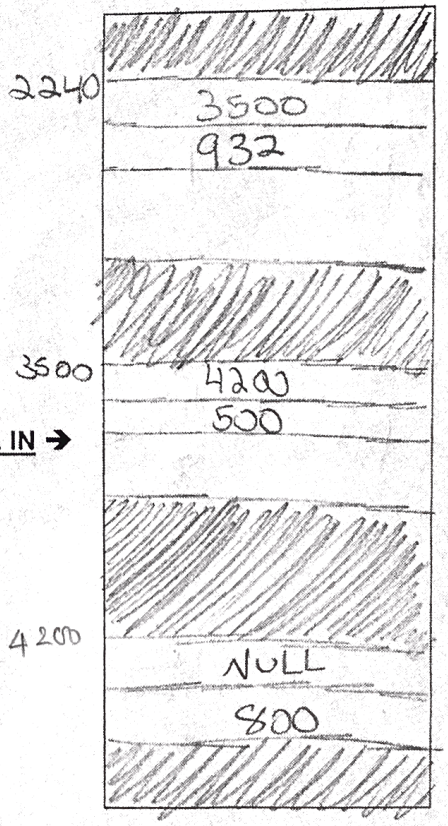
MEMORY BYTE LOCATION

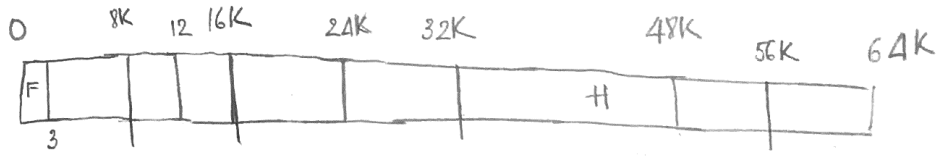


672
260
932

free block list head

2240 ← FILL IN





A 0 → 7 X
B 16 → 25 X
C 32 → 45 X
D 8 → 11 X
E 12 → 16 X
F 0 → 3
G No space
H 32 → 56
I 8

---

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

3

4 4 0 t2
6 3 3 R1 wait t2
8 2 6 R2 OK t2

1

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

4

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

1

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

0

4 2 2 t6
6 0 6
8 6 2 R1 OK t6

2

4 2 2 t7
6 0 6
8 6 2 R5 wait t7

2

4 2 2 t8
6 10 5
8 4 4 R5 OK t8

3