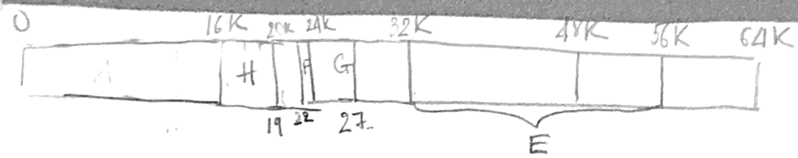


-5

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 allocation 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		13K
2	B		2K
3	C		1K
4	D		2K
5		A	
6	E		24K
7	F		2K
8		C	
9	G		3K
10		D	
11		B	
12	H		3K
13	I		7K

$2^{10} = 1,024$
$2^{11} = 2,048$
$2^{12} = 4,096$
$2^{13} = 8,192$
$2^{14} = 16,384$
$2^{15} = 32,768$

ANSWERS

Request A at 0

Request B at 16k

Request C at 18k

Request D at 20k

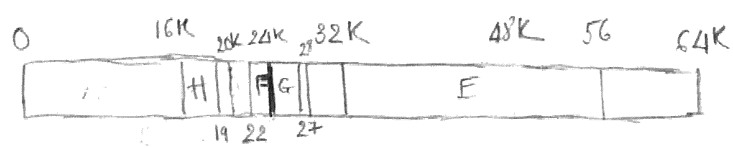
Request E at 32k

Request F at 22k

Request G at ~~Ø~~ 24

Request H at ~~4k~~ 16

Request I at ~~8k~~ Ø



-10

15 POINTS

2. The code shown below **compiles with no errors**, and has **no system call or library call errors**. It is to be **executed** by a process that has just used the execlp() system call to load and run this code from an a.out type file (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 **BLUE** and **signals** the associated condition variable using the **pthread_cond_broadcast()** call to ensure that both threads will 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 ✓

BLUE ?

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 process never finishes because the ~~2~~ created threads will remain locked and will not unlock and the threads will hang in Join.

th 1 will finish

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**.

the problem is from lines 28 to 30
we should set color = BLUE
using cond-wait varible.
so other threads have access to the
Condition variable. unlock mutex

4/4 5/4

Red

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

enum COLOR {RED, GREEN, ORANGE, BLUE} 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 = BLUE;
        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_cond_broadcast(&color_condx);
        return NULL;
}

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

15 POINTS

3. On an x86 **single core Linux** platform, when the **Initial Thread** (IT) of a newly created process (a process just created by a **fork()** call) begins executing in user space for the **very first time**, it always encounters an **immediate TLB fault**, even though it may then get **an L1 hit** after updating its TLB via a table walk.

   A. **Explain why** the thread encounters an **immediate TLB miss.**

   The thread is created due to HW context switch,
   table walk
   TLB in user space is being invalidated.

   B. **Explain** how it's possible for the thread to get **an L1 hit** after filling the missed TLB entry (via the table walk mechanism).

   A new process is created from a fork() call, so the data and code are in cache from the parent.

   C. Is there likely to be a **page fault or context switch** involved in the sequence described above (i.e. the TLB miss, followed by table walk, followed by L1 hit) ?? **Explain.**

   No, if the page is in memory or cache, the TLB table walk cannot page fault or context switch

431 t1
c 33
8 26 R1 OK t1
2
R2 waits t2

431 t3
6 24
8 ~~2~~ 64 R2 OK t3
1

440 t4
c 24
8 44 R3 OK t4
0

440 t5
6 24
8 26
2

431 t6
6 24
8 26
3

431 t7
c 24
8 26 R4 waits t7
9

431 t8
6 24
8 44 R5 OK t8

413 t9
6 24
8 44
0
R4 waits t9

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

3

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 thread needs, 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.

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

ANSWERS:

Request #1 granted at t(1)

Request #2 granted at t(3)

Request #3 granted at t(4)

Request #4 granted at X cannot be granted.

Request #5 granted at t(8)

4 23 21
6 3 3
8 2 6
32

4 31
6 3 3
8 24 64
20
R2 waits t(2)

4 3 1
6 32 34
8 24 64
231
R2 waits t(3)

4 34 10
6 2 4
8 4 4
10

4 3 1
6 2541
8 2 6
30
R4 waits

4 3 1
6 2 4
8 24 64
31

4 1 3
6 2541
8 4 4
30

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 <22> <214>

   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 <5><1107>

Tables on next page →

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

SYSTEM MEMORY FRAME TABLE (MFT)	PG #	PAGE TABLE FOR PROCESS A: VALID BIT	PG 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
FREE	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
OWNED BY C	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

CPU

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

2^4 2^2 2^0 $2^7 2^6$ 2^4 $2^2 2^1$

IR(BASE 2) 21 214

JUMP 0 0 1 1 0 | 0 0 1 0 0 0 1 0 1 0 0 1 1

$2^2 2^1$ 2^0 2^6 2^4 $2^1 2^0$

<~~8~~6> 1107

(a) 10110

2^4 $2^2 2^1$ = 22

<22><214>

(b) page fault

vp 6 is not mapped to pp.

only free page is 5

<5><1107>

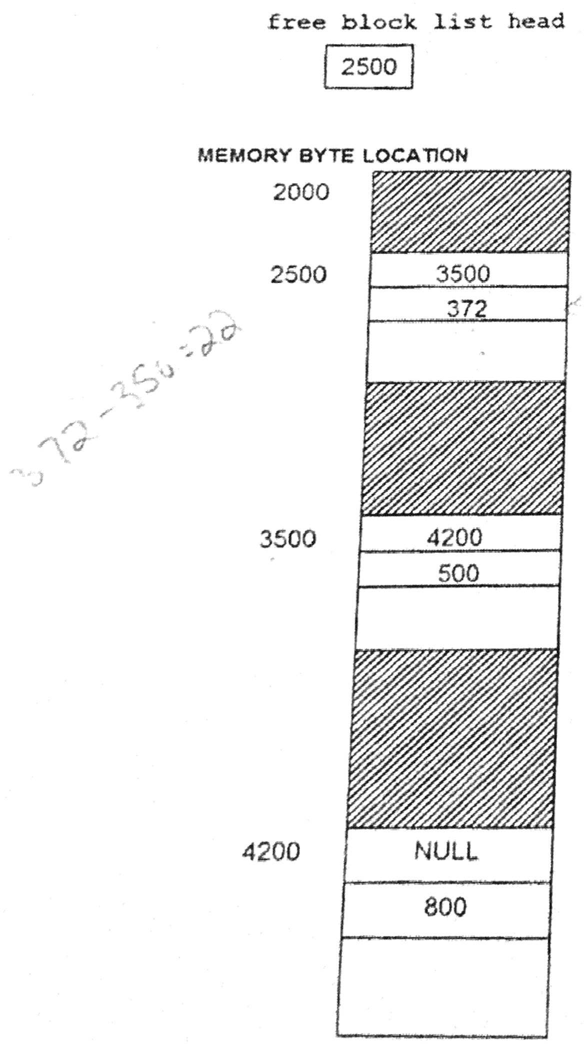
/2018

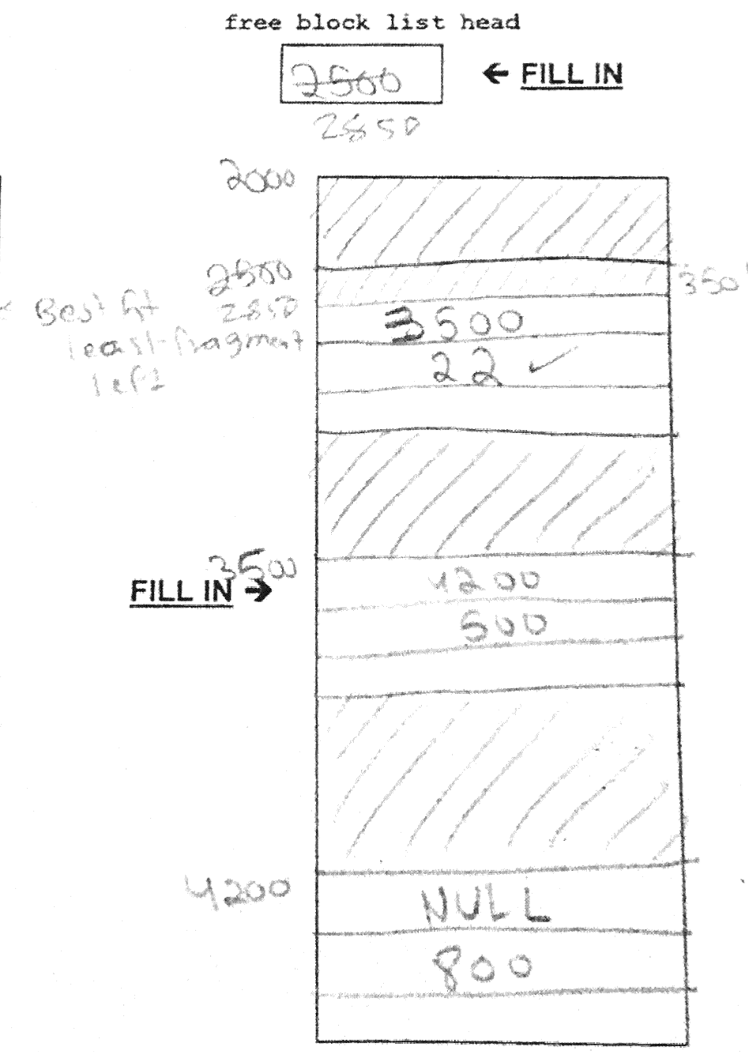
-3

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.





Problem 6 continued next page:

Problem 6 continued:

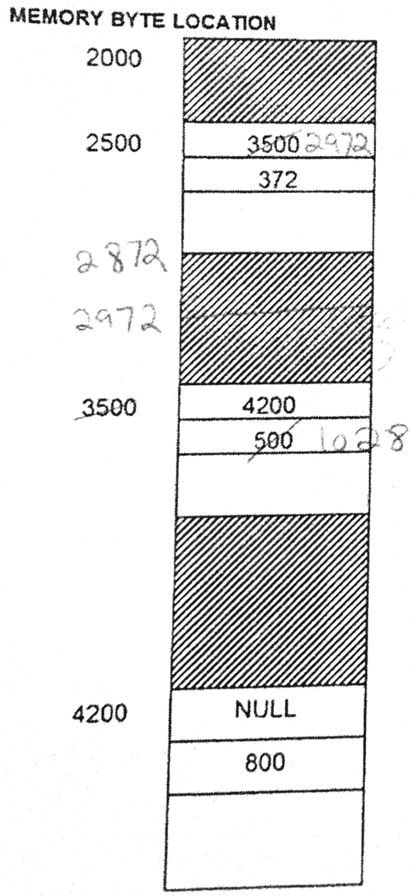
B. Given the initial state of the heap space shown, **fill in** the appropriate **free block list head** value, and **redraw** the organization of this space in the box provided, **after** a **free** operation of a previously allocated block of **528 bytes** is made at memory byte location (heap address)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

free block list head

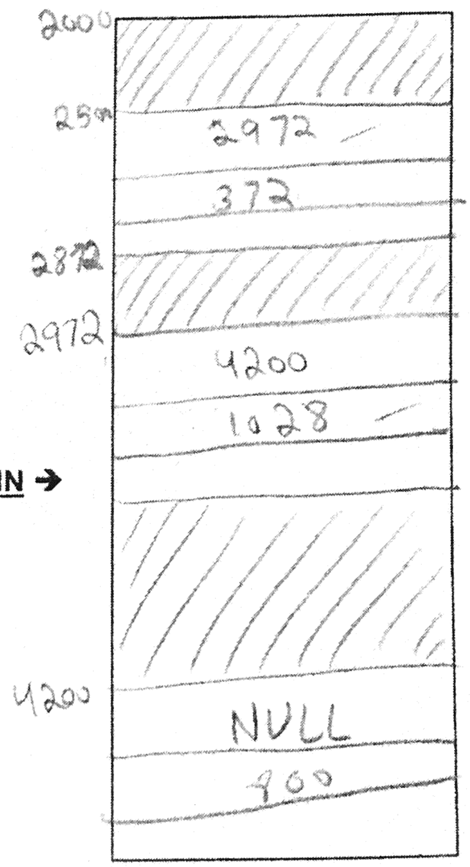
2500

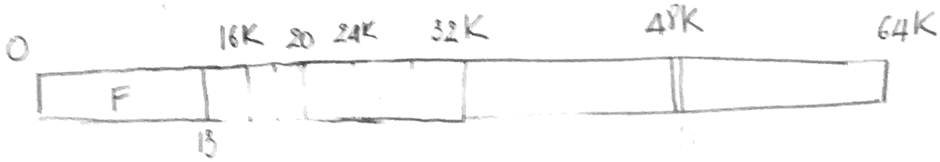
free block list head

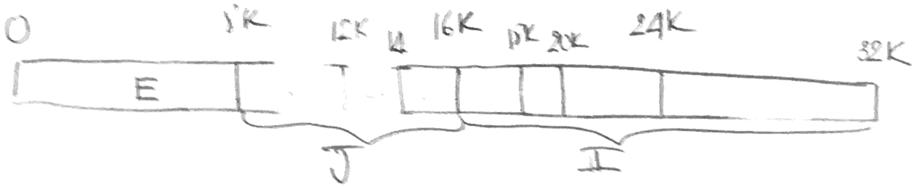
2500 ← **FILL IN**

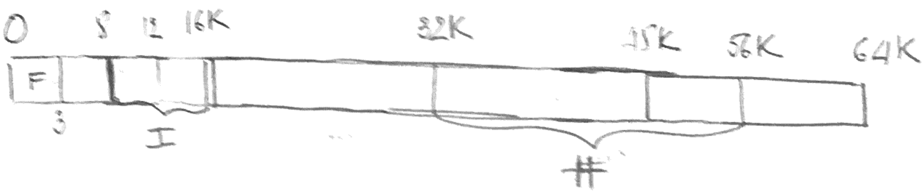


FILL IN →




0
16K
20
24K
32K
48K
64K
F
13


0
8K
12K
14
16K
18K
20K
24K
32K
E
J
I


0
8
12
16K
32K
48K
56K
64K
F
3
I
H