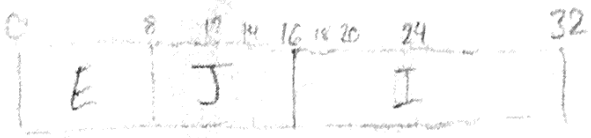


20 POINTS

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^{15} (32K) bytes. (Addresses run from 0 to 32k -1, and can be given in K form, i.e. location 4096 = 4K.) Assume that when memory is allocated from a 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. (Power of 2 values found on the references pages may be helpful here, since request sizes are given in actual bytes, not KBs.)

TIME	JOB REQUESTING	JOB RETURNED	REQUEST SIZE(bytes)
1	A		6,000 8K
2	B		3,300 4K
3	C		1,500 2K
4		A	
5	D		12,000 16K
6	E		8,100 8K
7	F		2,000 2K
8		D	
9	G		1,300 2K
10	H		13,500 16K
11		F	
12		G	
13	I		14,700 16K
14		C	
15		B	
16	J		7,600 8K

ANSWERS

Request A at 0 Request F at 14K

Request B at 8K Request G at 16K

Request C at 12K Request H at NO SPACE

Request D at 16K Request I at 16K

Request E at 0 Request J at 8K

15 POINTS

2. Consider the following resource-allocation policy for a fixed inventory of **serially reusable** resources of three different types (such as tape drives, printers, shared memory, etc.):

   - Requests and releases of resources are allowed at any time.
   - If a request for resources cannot be satisfied because resources are not available, then we check any processes that are blocked, waiting for additional resources:
   - If they have the desired resources, then these resources **are taken away** from those blocked processes and are **given** to the requesting process.
   - The inventory of resources that the requesting process is collecting is **increased** to include the resources that were taken away from the blocked process.
   - If the requesting process is still not able to collect the resources it needs to continue to run, it will be **forced to block and wait for what it needs**. Once it blocks, its current resource inventory will become **vulnerable to other running processes looking for additional resources.**
   - Whenever a resource is freed, blocked process needing such resource are made ready, and when they are dispatched to the run state they will have a chance to again try and secure their necessary inventory of resources so that they can continue to run.

For example, consider a system with **three resource types** and a **total inventory** of (4,2,2). If process **A** asks for (2,2,1) it gets them. If process **B** asks for (1,0,1), it also gets them. Then, if process **A** asks for a further allocation of (0,0,1), it is blocked (resource not available). If process **C** now asks for (2,0,0), it gets the available one from the systems (1,0,0) inventory and the one which was allocated to process **A** (since process **A** is blocked). Process **A's** allocation goes down to (1,2,1) and its need (outstanding request) goes up to (1,0,1).

Problem 2 is continued on next page

2. *Continued:*

A. List the **4 necessary conditions** for a deadlock to occur in a computing system.

Mutually exclusive resources
Hold-and-wait condition
No pre-emption
Circular waiting

B. Can **deadlock** occur in the system described above ? **If you think it can**, give an example. **If you think it cannot**, describe which necessary condition **cannot occur** that would be required for deadlock ?

No because no pre-emption is denied

C. Can **indefinite postponement** occur in the system described ? **Explain.**

Yes, IP can occur when pre-emption is allowed if a process continuously has its resources taken before it can finish its work

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.**

Since the thread is in a different process (HW context switch) a TLB shootdown invalidated the previous entries

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).

Because the forked process is running the same code as its parent, and since the parent was just running its memory would still be in cache

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, because the memory left by the parent is still valid for the child at this point, neither page faults nor context switches happen during the process described

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 ~~4~~ ~~1~~ 2 1	2 ~~0~~ ~~3~~ 2
b	6	~~3~~ ~~6~~ ~~2~~ 3	~~3~~ ~~0~~ ~~4~~ 4
c	8	2 4	6

free
~~4~~ ~~4~~ ~~3~~ ~~4~~ 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 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 ~~wait~~ wait
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 wait
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 never

proc	max	current	diff
a	4	1	3
b	6	2	4
c	8	4	4

free
3

8k =

18 bit

$2^{18} \sim 256$

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 <107><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 <13><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
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
FREE	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

page fault
map to PP 13 (only one free)

CPU

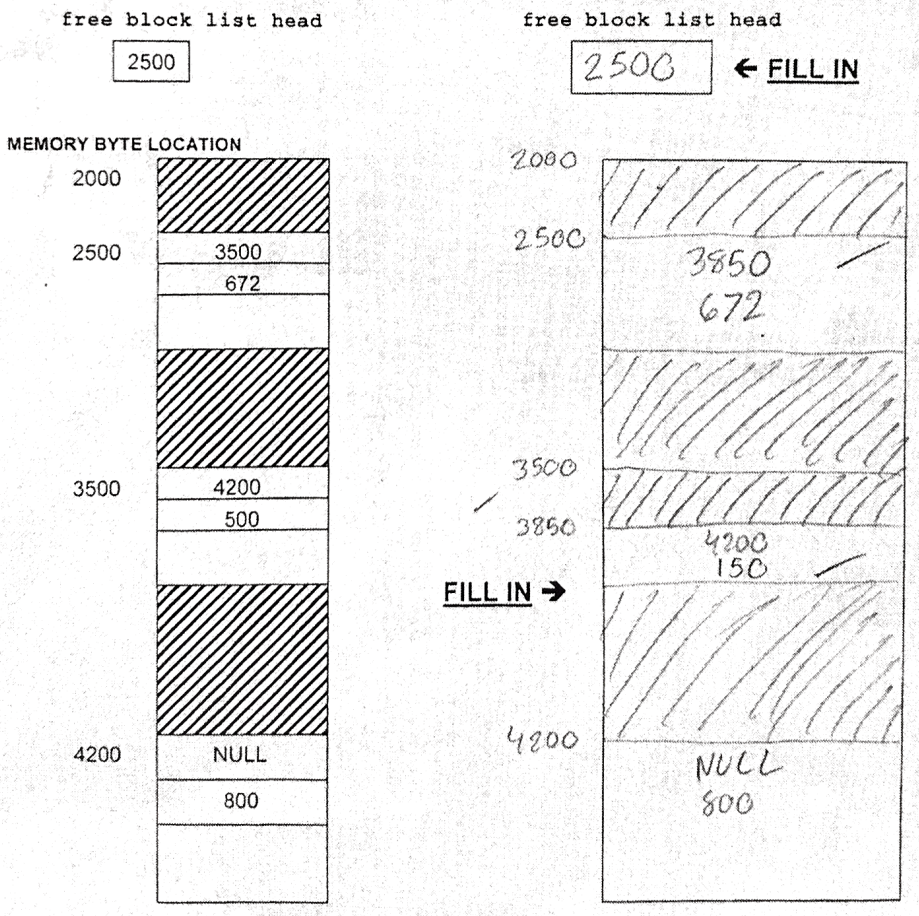
PC(BASE 2) 1 0 0 1 1 | 0 0 0 0 0 1 1 0 1 0 1 1 0

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

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:

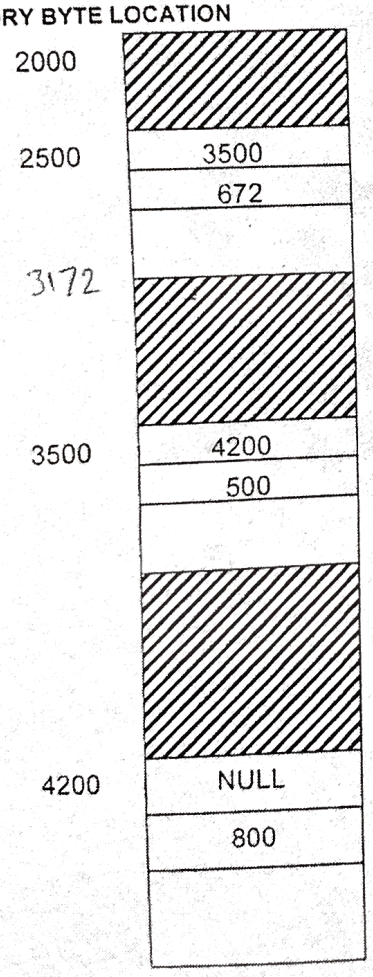
Problem 6 continued:

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 **328 bytes** is made at memory byte location (heap address) **3172**.

free block list head

2500

MEMORY BYTE LOCATION



free block list head

2500 ← FILL IN

FILL IN →