The following problem deals with a virtual memory system with **a 12 bit address space (from 0 to 4096 (4K) locations).** The system is byte addressable and uses a **256 byte per page** organization. The real memory, therefore, is organized into **16 page frames of 256 bytes** each. Assume the operating system itself occupies the last six pages permanently, and that user programs will page against the **first 10 pages** as they run. Remember, the 12 bit address space will allow a user to have a virtual address space of **4096 bytes** (16 pages) even though only 10 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. All process are full size programs using all 16 virtual pages available. **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 **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 ?

B. To what **REAL physical byte address** will control be transferred when the current JUMP instruction executes ??

SYSTEM PAGE FRAME TABLE AND CURRENT PAGE TABLE FOR RUNNING PROCESS A ARE SHOWN BELOW (THE OS KEEPS THESE IN ITS SPACE)

SYSTEM PAGE FRAME TABLE (PHYSICAL PAGES)

PAGE #	STATUS
0	OWNED BY A
1	OWNED BY B
2	OWNED BY C
3	OWNED BY B
4	OWNED BY A
5	FREE
6	OWNED BY A
7	OWNED BY C
8	OWNED BY C
9	OWNED BY A
10	OP SYS
11	OP SYS
12	OP SYS
13	OP SYS
14	OP SYS
15	OP SYS

PAGE TABLE FOR PROCESS A

FRAME # (BASE 2)	VALID BIT
NONE	0
NONE	0
NONE	0
0000	1
NONE	0
0110	1
NONE	0
1001	1
NONE	0
NONE	0
0100	1
NONE	0
NONE	0
NONE	0
NONE	0
NONE	0

CPU

PC(BASE 2) 101010010110

IR(BASE 2) JUMP 111011011011

1 0 1 0 1 0 0 1 0 1 1 0

VP = 10 offset = 150

PP = 4 ans: <4><150>

1 1 1 0 1 1 0 1 1 0 1 1

VP = 14 offset = 219

PAGE FAULT

ONLY FREE PAGE IS 5

PP = 5 ans: <5><219>

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 a resource is made by a process which is **already holding other resources**, the request may be **denied** based on a system imposed **ordering** required for allocations. For example, in a system imposed ordering it may be required that any process that must hold a tape drive and a printer at the same time must ask for and obtain the tape drive(s) before asking for the print device(s).

- Resources which are **currently in use by other processes** will cause a requesting process to block waiting for their availability in FIFO order.

- Whenever a resource is **freed,** some blocked process needing such resource may secure the resource and **move to the ready state**.

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

Mutex Resources
Hold and Wait
No Pre-emption
Circular Wait

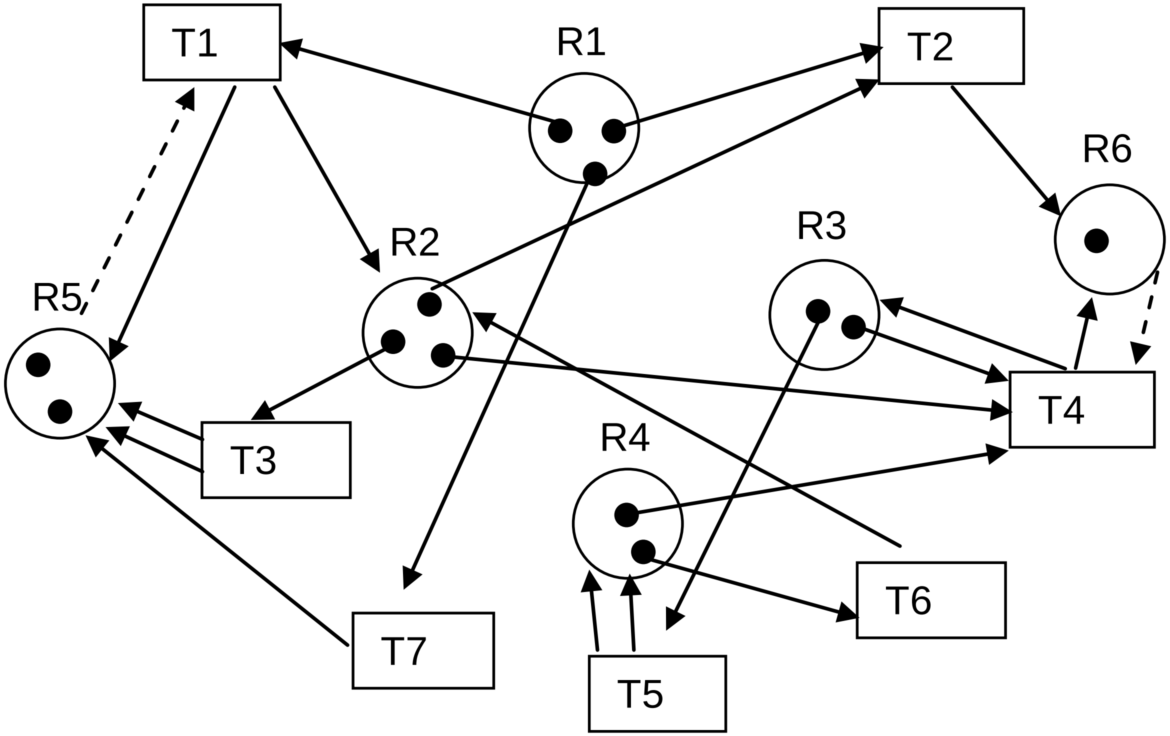
B. Can **deadlock** occur in the system described above ? **If so**, give an example. **If not**, which **necessary condition** cannot occur that would be required for a deadlock ?

NO DL, the Circular Wait Condition is Denied by imposing strict allocation ordering

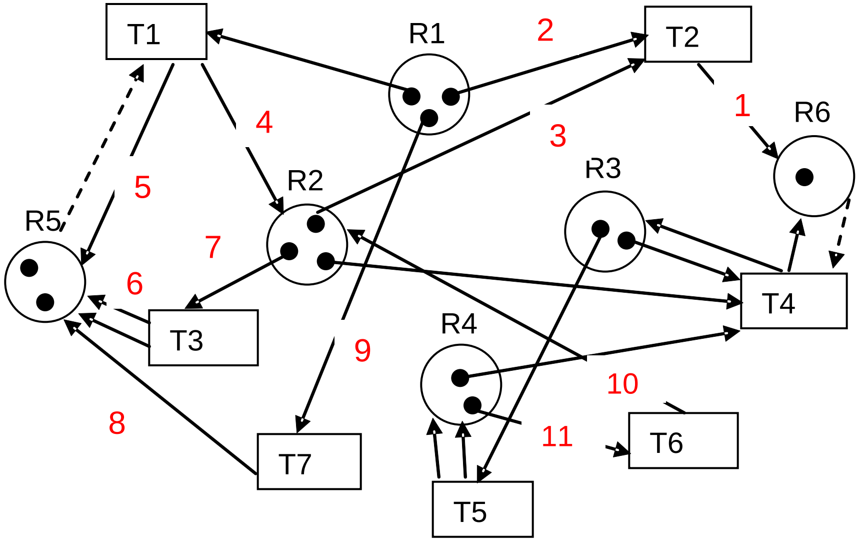
C. Can **indefinite postponement** occur ? **Explain.**

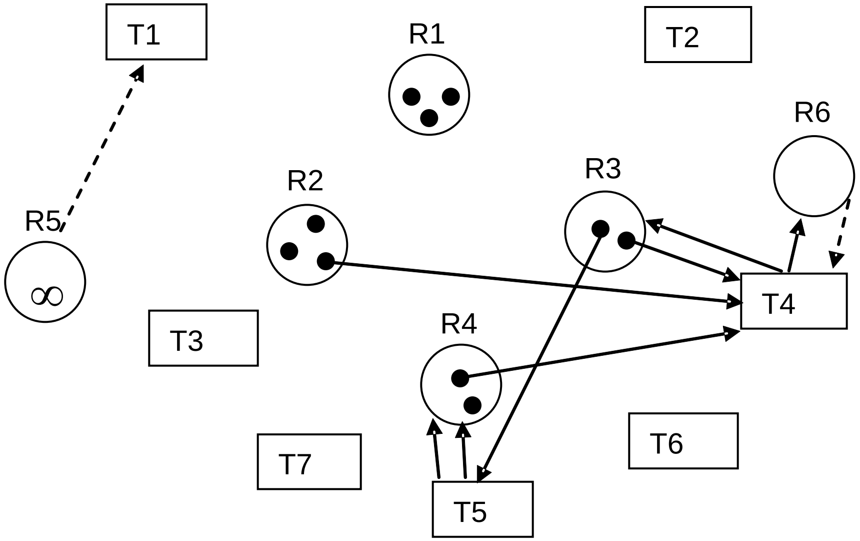
NO IP … IP is a problem when denying No Pre-emption

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

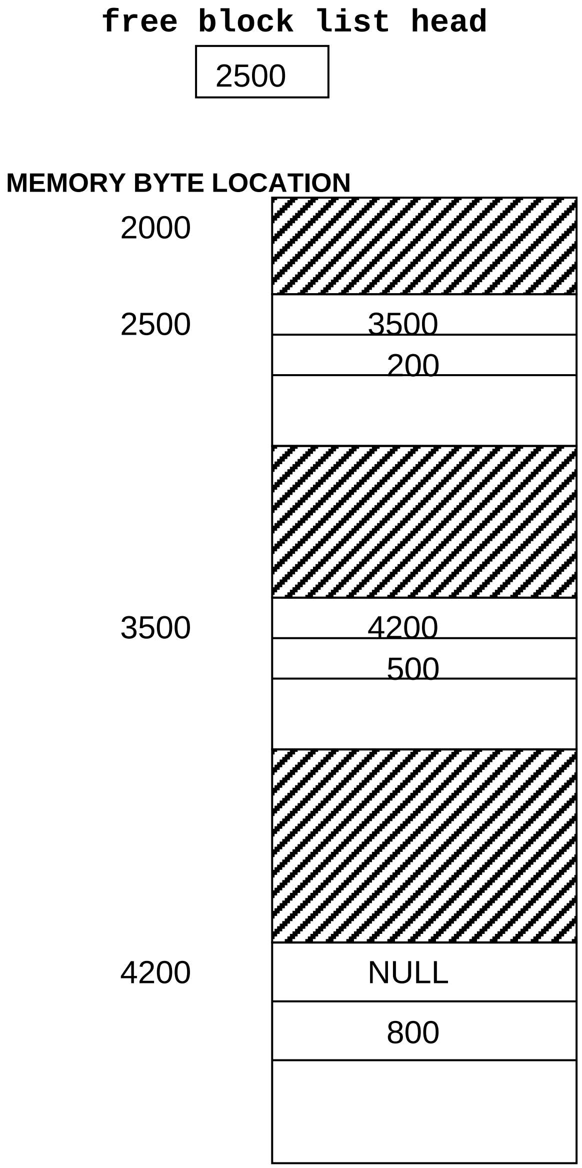


DRAW THE FINAL, REDUCED RESOURCE GRAPH HERE:

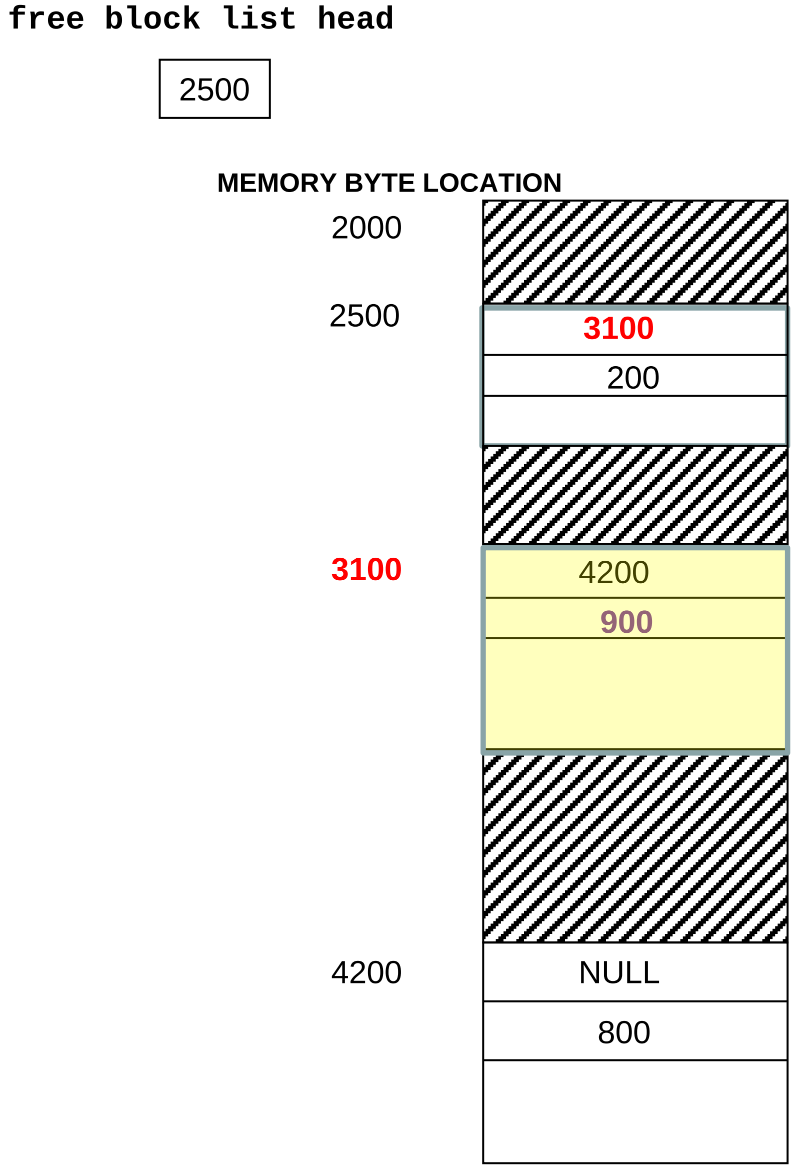

T1
R1
2
T2
1
R6
4
3
R3
5
R2
R5
7
6
T4
T3
R4
9
10
8
11
T6
T7
T5


T1
R1
T2
R6
R5
R2
R3
∞
T4
T3
R4
T6
T7
T5

The figure below depicts the free space at a point in time **just before** a **`free(3100)`** operation is called by some process that had **previously requested** a block of heap space of **size 400 bytes**. You must **fill in the empty rectangle** on the right side of the diagram with the **complete new picture** of the free space after the **`free()`** operation discussed above has completed.



FILL IN ALL DETAILS OF THE NEW FREE LIST ORGANIZATION BELOW



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	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 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 times 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

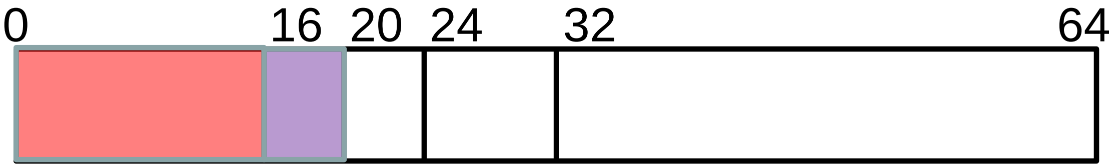
ANSWERS:

Request #1 granted at ____**???**_______

Request #2 granted at ______**@ T2**______

Using the buddy system of memory allocation indicate 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) words. (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 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:

TIME	JOB REQUESTING	JOB RETURNED	REQUEST SIZE(WORDS)
|	A		12K
|	B		3K
v	C		17K
|		A	
|	D		1K
v	E		6K
|		B	
|		D	
v	F		8K
|		E	
|		C	
v	G		15K



ANSWERS

Request A at ____________ Address 0

Request B at ____________ Address 16K

When a **heavy-weight context switch** occurs on a UNIX or Windows system (one process address space leaving the CPU and another coming onto the CPU) we say that a **TLB shootdown** must be done to the processor's memory management **TLB** entries.

A. Draw a diagram and **label and describe** the fields in a **TLB** entry

VALID	**Virtual Addr**	**PhysAddr + PROT (PTE)**

B. Explain what a **TLB shootdown** is and why it is necessary on a **heavy-weight** context switch.

Virtual addresses do NOT map the same in a new address space

C. Is a **TLB shootdown necessary** when a context switch is made between **one thread** of a process and **another thread of the same process** ?? **Explain.**

NO: Same address space is used by ALL threads of a process