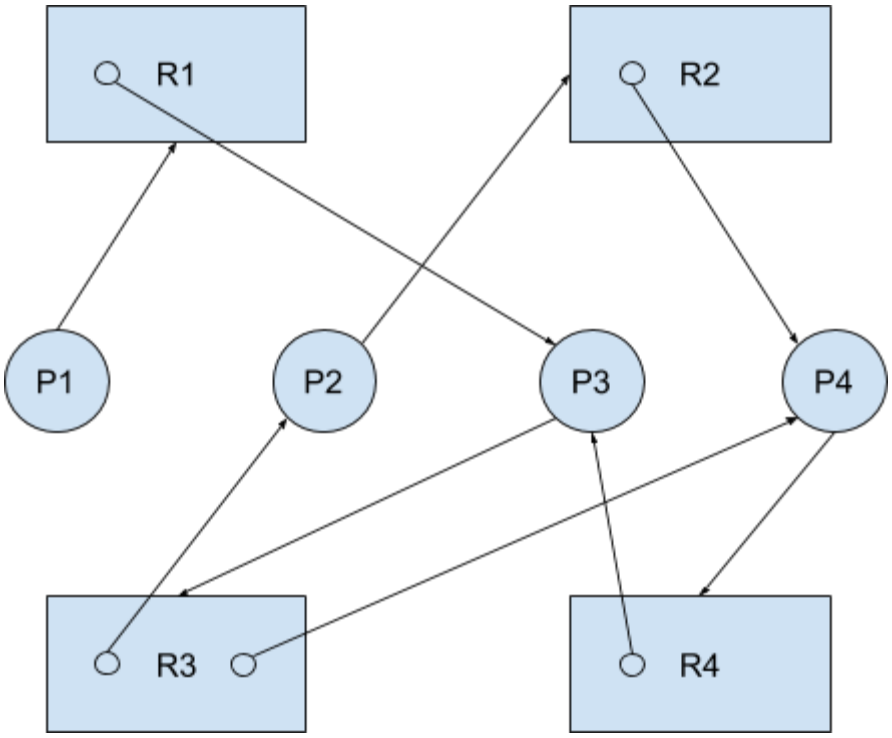
Part A: [True / False]

Mark the following (T-F) questions using T for True and F for False in the table below.
Illegible answers will be considered incorrect.

		T / F	TOPIC
	Direct Memory Access (DMA) bypasses the cache or TLB to read directly from main memory.	**F**	**INTRO**
2.	In the pyramid-shaped memory hierarchy, *registers* appear above *cache.*	**T**	**INTRO**
3.	Multiple threads within the same process share heap memory but not stack memory.	**T**	**PROC**
4.	In UNIX, the *exec*() system call creates a new child process, and the *fork* system call replaces the current process's program with a new program and executes it.	**F**	**PROC**
5.	The *round robin* scheduling algorithm avoids the *convoy effect*.	**T**	**SCHED**
6.	*Priority inversion* is a solution to the *priority inheritance* problem.	**F**	**SCHED**

7.	The following pseudo-code for a process and TestAndSet function, without any other assumptions, solves the critical section problem between N processes:
	
	`function TestAndSet(boolean lock)`
	`{`
	`  boolean initial = lock;`
	`  lock = FALSE;`
	`  return initial;`
	`}`
	
	`Initial value of lock = TRUE;`
	
	`function Process(int N) {`
	`  Repeat`
	`  while(TestAndSet (lock))`
	`no-op;`
	`  critical section`
	`  lock = TRUE;`
	`  remainder section`
	`  until FALSE`
	`}`
		F	**SYNC**

8.	The *Progress* requirement for the critical section problem states that: if no process is currently in the critical section and then one requests to enter the critical section, this requesting process will be granted access to the critical section in a finite period of time.	**F (any proc. Gets to go maybe not this one…)**	**SYNC**
9.	*Paging* breaks up a program's view of memory into logical blocks such as "main", "global variables", and functions/procedures	**F**	**MEM**
10.	*Late binding* of memory addresses enables the compiler to produce more efficient code than *early binding*.	**F**	**MEM**
11.	Consider an OS that employs the *working set model* to track the number of recent page references. To avoid the problem of	**T**	**MEM**

	thrashing, it swaps out a process if the total # demand frames exceeds the # available frames.		
12.	*External fragmentation* is free memory between allocated partitions. *Internal fragmentation* is memory that is internal to an allocated partition but is not used.	**T**	**VirMEM**
13.	The dynamic storage allocation strategies of *worst-fit* and *best-fit* are better than *first-fit* in terms of speed and storage utilization.	**F**	**VirMEM**
14.	*Segmentation* leads to problems with *external fragmentation*; using *segmented paged memory* overcomes this (at the cost of some *internal fragmentation*).	**T**	**VirMEM**
15.	The *Banker's Algorithm* is used to avoid deadlocks when considering multiple instances of each resource type.	**T**	**DEADLOCK**
16.	Consider a system having p processes, where each process needs a maximum of m instances of resource type R1. Given that there are r instances of resource type R1 in total, we can ensure that the system is deadlock-free by guaranteeing enough instances of R1, i.e. ensuring that: $r \geq p(m - 1)$	**F** $r \geq p(m\text{-}1) + 1$	**DEADLOCK**
17.	The following resource allocation graph contains a deadlock: 	**T**	**DEADLOCK**
18.	The *Indexed Allocation* scheme increases storage overhead by including an index table, which is a block containing the addresses of each actual file block.	**T**	**FILEsys**

19.	In a filesystem using *tree-structured directories*, the leaf nodes are directories and the interior nodes are files.	**F**	**FILEsys**
20.	The *Linked Allocation* scheme causes external fragmentation.	**F**	**FILEsys**

Part B: [Processes / Threads & CPU Scheduling]

Question B.1: Inter-process communication

Briefly (1-2 sentences) explain the main difference between **shared memory** IPC and **message passing** IPC.

Shared memory maps addresses to common physical memory. Uses read/write into memory to communicate between processes.
Message passing uses send() / receive() methods to communicate between processes, e.g. using mailboxes.

Question B.2: Round Robin and Time Quantum

Consider N processes sharing the CPU in a round-robin fashion (N>=2). Assume that each context switch takes S ms and that each time quantum is Q ms. For simplicity, assume that processes never block on any event and simply switch between the CPU and the ready queue. Also, assume that a process is still in the ready queue while a context switch is happening.

a) What happens if Q is much smaller than S? What happens when $Q \rightarrow \infty$, i.e. is much larger than the maximum turnaround time of all the processes? Be brief (1-2 sentences max) in your answer.

If Q << S, system spends most of the time context switching and performs poorly. If Q is much larger than any process's burst time, we basically have FCFS algorithm.

b) If you use RR for scheduling, which of the three performance metrics (waiting, response, turnaround time) is more likely to be improved? Why (1-2 sentences max)?

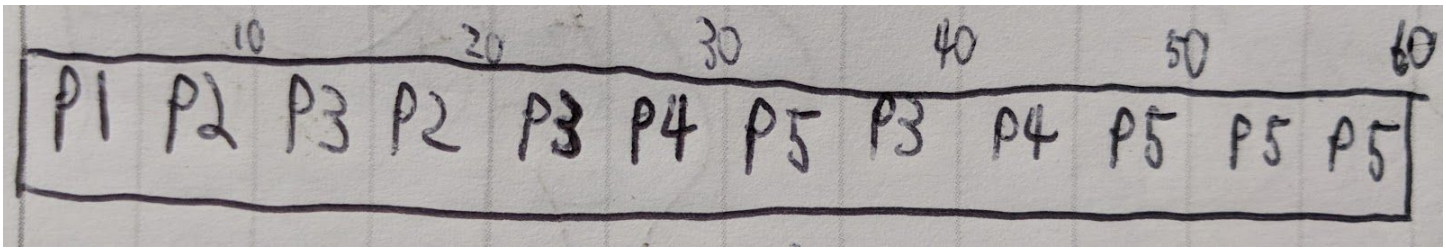
Response time because of how RR fairly switches between each process, thereby more quickly giving a newly-arrived process a turn.

Question B.3: CPU Scheduling Algorithms

Consider the set of process:

Process ID	Arrival Time	Burst Time
P1	0	5
P2	0	10
P3	4	15
P4	18	10
P5	22	20

a) Draw the GANTT chart for the **Round Robin** (time quantum = 5) scheduling algorithm. Use the same implementation you used for the programming assignment i.e. the processes should always run in PID order. Assume there is no context-switch overhead. Show your work for partial credit.



b) Write your answer to the following performance metrics given your above GANTT charts. Show your work for partial credit.

Average Response time:

(0+5+6+7+8)/5 = **5.2**

Average Waiting time:

(0+10+21+17+18)/5 = **13.2**

Average Turnaround time:

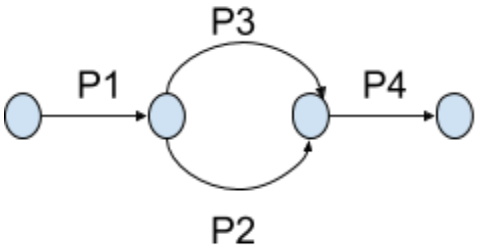
(0+10+21+17+18+60)/5 = **25.2**

((5-0)+(20-0)+(40-4)+(45-18)+(60-22))/5 = **25.2**

Part C: [Synchronization & Deadlocks]

Question C.1: Process Synchronization

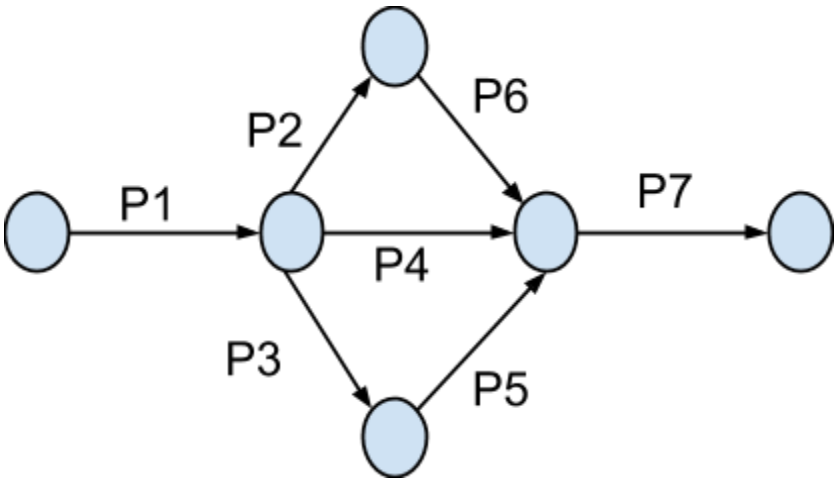
Recall the following process execution diagram:



Recall how we use the following semaphores to enforce the execution order above:
s1=0; s2=0; s3=0;
P1: body; V(s1); V(s1);
P2: P(s1); body; V(s2);
P3: P(s1); body; V(s3);
P4: P(s2); P(s3); body;
Where the semaphores **s1**, **s2**, and **s3** are created with an initial value of 0.

(continued on next page)

Use the same semaphore notation shown above to describe how we can ensure the execution order of the following process execution graph:



Use **all of** the following semaphores in your answer:

s1=0; s2=0; s3=0; **s4=0**; **s5=0**; **s6=0**;

(write your answer below here)

Pr1: body; **V(s1); V(s1); V(s1);**
Pr2: P(s1); body; V(s2);
Pr3: P(s1); body; V(s3);
Pr4: P(s1); body; V(s4);
Pr5: P(s3); body; V(s5);
Pr6: P(s2); body; V(s6);
Pr7: P(s4); P(s5); P(s6); body;

Question C.2: Readers / Writers

Recall the Readers / Writers problem from lecture. Fill in the blanks below to properly complete the following pseudocode implementation:

Shared Data:

```
Semaphore wrt initialized to 1
Semaphore mutex initialized to 1
Integer read_count initialized to 0
```

Writer Process:

```
do {
    wait(wrt);
    ...
    /* writing is performed */
    ...
    signal(wrt);
} while (true);
```

Reader Process:

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
         wait(wrt);
    signal(mutex);
    ...
     /* reading is performed */
    ...
    wait(mutex);
     read_count--;
     if (read_count == 0)
         signal(wrt);
    signal(mutex);
} while (true);
```

Question C.3: Deadlocks & safe state

Consider the following snapshot of a system:

Process	***Allocation***				***Max***				***Available***			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	2	1	4	2	4	2	1	0	0	0
P1	0	1	0	1	0	2	2	2				
P2	1	2	0	0	3	2	1	0				
P3	0	1	1	2	1	1	1	2				
P4	0	0	1	1	1	0	2	1				

a) What is the content of the matrix ***Need***?

Process	***Need***			
	A	B	C	D
P0	1	2	2	1
P1	0	1	2	1
P2	2	0	1	0
P3	1	0	0	0
P4	1	0	1	0

b) Is the system in a safe state? If yes, give a safe sequence of processes. If not, explain why the system is not in a safe state.

Yes, it is in a safe state. The ONLY safe sequence is <P3, P4, P1, P0, P2>

Which will produce Avail. Vectors: <1,1,1,2>; <1,1,2,3>; <1,2,2,4>; <4,2,4,5>...

c) If a request from process P4 arrives for (1,0,0,0), can the request be granted immediately? Please state the reason.

No, because it would leave 0 for all resources

Part D: [Virtual Memory Management]

Question D.1: Effective Access Time

A computer keeps its page tables in memory. Memory access time is 100 nanoseconds (ns). Answer the following questions about the performance of this setup. Show your work.

a) What is the effective access time (i.e. reading a word in memory) with no caching and a two-level page table?

300 ns: one for outer page table lookup, one for inner, and one for actual read

b) Consider the above scenario but with a TLB having a cache hit rate of 98%. If the TLB takes 20 ns to access, what is the effective access time of this setup when considering this TLB?

0.98*120 + 0.02 * 320 = 124 ns

Question D.2: Bits needed for addressing

Consider a system that uses a fixed-partition scheme, with equal partitions of size 2^8 bytes, and the main memory has 2^{16} bytes. A process table is maintained with a pointer to the resident partition for each resident process. How many **bits** are required for the pointer in the process table? Show all your steps.

8 bits:

partitions = (main memory size) / (size of each partition) = 2^16 / 2^8 = 2^8.

Question D.3: Segmentation

Given the following segment table:

Segment	Base (original)	Length	Base (after compaction)
0	100	300	0
1	1400	600	400
2	450	100	300
3	3200	80	1500
4	2200	500	1000
5	3300	33	1580

1. Given the **original** base addresses, what are the physical addresses for the following logical addresses? If it's an invalid address, just write "invalid". Note that (X, Y) => segment X, offset Y

a) (0, 350)
= INVALID

b) (1, 599)
= 1999

c) (2, 50)
= 500

d) (3, 81)
= INVALID

e) (4, 300)
= 2500

f) (5, 0)
= 3300

g) (5, 34)
= INVALID

2. If the above segments are the only segments in the system, i.e. there is only 1 process, and all segments are full, fill the right-most column of the base address table above after completing a disk compaction.

Question D.4: FIFO Page Replacement

Consider the following page reference string:

e, c, b, e, a, g, d, c, e, g, d, a

Considering 4 frames, fill in the following table and then answer how many page faults would occur with the FIFO page replacement algorithm.

RS: reference string; F0: frame 0, F1: frame 1, etc.

Hint: all frames are initially empty, so your first unique pages will all cost one fault each.

a)

Time	1	2	3	4	5	6	7	8	9	10	11	12
RS	e	c	b	e	a	g	d	c	e	g	d	a
F0	e	e	e	e	e	g	g	g	g	g	g	g
F1		c	c	c	c	c	d	d	d	d	d	d
F2			b	b	b	b	b	c	c	c	c	a
F3					a	a	a	a	e	e	e	e
Page fault?	y	y	y	n	y	y	y	y	y	n	n	y

b) Total # page faults:

9

c) Briefly (1-2 sentences) explain *Belady's Anomaly* that can occur in FIFO Page Replacement.

more available frames does not mean less page faults

Part E: [Filesystems]

Question E.1: Linked Allocation File System Operation

Consider a file currently consisting of 10 blocks. Assume that the file control block and the new block information to be added are already in memory. Calculate how many disk I/O operations are required for the linked allocation strategy, if, for one block, the following conditions hold:

HINTS: 1) ignore disk I/O associated with the file control block. 2) each read and each write is an explicit disk I/O. 3) assume a pointer to the end of the list for linked allocation

a. The block is added at the beginning.
1

b. The block is added in the middle.
7 (sequentially read 5, write the new block, and update block 5's pointer)

c. The block is added at the end.
3 (read last block, write new block, update last block's pointer)

d. The block is removed from the beginning.
1

e. The block is removed from the middle.
7

f. The block is removed from the end.
10

Question E.2: Indexed Allocation File System Operation

Consider a *two-level indexed allocation* scheme that uses only a single top-level *index table*. If each block is *B* bytes and each index entry is *P* bytes, what is the maximum file length in bytes?

B^3/P^2 = (B/P)*(B/P)*B

B/P second-level index blocks, where each contains pointers to B/P file blocks, which are each B bytes long