With demand paging, adding memory to a system always improves the cache hit rate.

False, adding memory to a demand paging system using FIFO can decrease hit rate (Belady's Anomaly)

It is not possible to implement user-kernel separation without hardware support for dual mode operation.

False, Software-based techniques can be used in place of hardware support.

The size of a process' Working Set may depend on the number of processes currently running.

True, a process' working set is determined by the process' memory reference pattern. Since virtual time is used to track this behavior, it is unaffected by other processes.

Doubling the block size in the UNIX file system will double the maximum file size.

False, the amount of data in each data block is doubled and the size of each singly, doubly, and triply indirect block is also doubled increasing the total number of data blocks that can be indexed.

Adding a memory cache never hurts performance.

False, adding a cache increases the time to perform a reference in all cases. Caching can hurt performance when the hit rate is low. Additionally, the first access is always slower with a cache.

Address translation (virtual memory) is useful even if the total size of memory as summed over all possible running programs is guaranteed to be smaller than a machine's total physical memory.

True, address translation simplifies the load/linking process and allows programs to be placed anywhere in physical memory. Also, it enforces isolation between programs.

When the total execution times of each of the long-running (no I/O) processes waiting to be scheduled are equal, FIFO is the better choice (in terms of response time) of scheduling algorithm to use instead of round robin.

True, Using FIFO yields a lower average response time because jobs finish sooner than with round robin, where all jobs finish in the last N time slices.

In a multi-level address translation scheme, the amount of memory used by the translation tables for each process' virtual address space is always proportional to the size of the virtual address space.

False, the amount of memory used by the translation tables is proportional to the amount of the virtual address space that is in use by the process.

Monitors are more powerful than Semaphores because Monitors can implement solutions to synchronization problems that Semaphores cannot.

False, Monitors and Semaphores have equivalent expressive power for synchronization problems. Semaphores can be used to implement monitors and vice versa.

A CPU scheduling algorithm cannot provide both fairness and minimum average response time.

True, there is a fundamental tradeoff between fairness and average response time.

A correct application using Banker's algorithm for all requests will never deadlock.

True, Banker's algorithm only allows requests that leave the app in a safe state.

SRTF and SJF are optimal page replacement algorithms that cannot be implemented in practice.

False, SRTF and SJF are scheduling algorithms.

If a set of cooperating threads can correctly share a single core uniprocessor, then they will correctly execute if run on separate processors of a shared-memory multiprocessor.

False, Threads can use disabling of interrupts to cooperate on a uniprocessor, but that may not work correctly on multiprocessors.

Dual-mode operation is required to protect the operating system in a multiprogrammed system.

False, software fault isolation techniques can be used to protect the OS from malicious programs.

The four conditions that must hold in order for deadlock to occur are: Hold-and-wait, circular waiting, starvation, and mutual exclusion.

False, starvation is not a condition for deadlock. It should be preemption.

The main advantage of multilevel page tables is that they use page table memory efficiently.

True, multilevel page tables use memory more efficiently for sparse address spaces than single level tables.

Demand paging requires the programmer to take specific action to force the operating system to load a particular virtual memory page.

False, the OS automatically loads pages from disk when necessary.

Each thread has its own stack.

True, a thread is a unit of execution and uses its stack to store the return addresses and arguments passed to functions/methods.

Starvation implies deadlock.

False, a system may exit from starvation but not from deadlock.

It's generally possible to substitute a semaphore for a condition variable, because sem.V()/sem.P() have similar semantics to cond.signal()/cond.wait().

False, cond.signal() before cond.wait() has no effect, but sem.V() before sem.P() has an effect. Conditional variables are not commutative and a semaphore has no notion of an implicit lock.

SRTF is the optimal scheduling algorithm, but it is generally not implemented directly due to excessive context switching overhead.

False, it is not implemented because it is impossible to predict the future.

Using a smaller page size increases the size of the page table.

True, you need more pages for allocating the same memory which means more entries in the page tables.

Moving from a single level page table to a two-level page table will always decrease the memory footprint used by the page table.

False, If a process uses all addressable memory, you have strictly more memory used by a 2-level page table.

Historically, what has been the difference between a computer virus and a computer worm?

A computer virus requires human intervention to spread (e.g., visiting a website, running a Trojan program, connecting a USB), while a worm spreads on its own.

List two reasons why operating systems disable interrupts when a thread/process sleeps, yields, or switches to a new thread/process.

(1) To prevent context switching from occurring while registers are being saved/restored.
(2) To enable a thread/process to add itself to a queue without allowing another thread to interrupt the action.

Give a definition for a non-blocking write operation.

The caller returns immediately and the caller is informed later whether the write was successful or not.

Explain what causes thrashing in a system that uses virtual memory management.

Thrashing occurs when memory pages are constantly being swapped out of the main memory and to disk at a rate too rapid to accomplish any real work. The system spends all its time swapping pages and not actually doing any computations.

List the terms that best describe each of the following:

a. **Operating system code executed when an asynchronous device signals the CPU.**
    i. Interrupt handler
b. **A type of disk arm scheduling policy**
    i. FIFO, Shortest seek first, SCAN, C-SCAN, or elevator

Briefly describe the steps taken to read a block of data from the disk to the memory using DMA controlled I/O.

1. CPU programs the DMA controller by setting its registers so it knows what to transfer where.
2. The DMA controller initiates the transfer by issuing a read request to the disk controller.
3. The disk controller fetches the next word from its internal buffer and writes to the memory.
4. When the write is finished the disk controller sends an acknowledgement to the DMA controller.
5. If there is more data to transfer then the DMA controller repeats steps 2 through 4. If all data has been transferred the DMA controller interrupts the CPU to let it know that the transfer is complete.

Explain what a symbolic link is and list at least two of its drawbacks.

A symbolic link is a way to achieve file sharing. When we want to share a file, we create a new file, or a link. This file contains the path name of the file we want to share. Later when we access this link file, the operating system sees that the file being accessed is a link so it looks up and accesses the referred to file. A symbolic link needs extra processing time because it has to follow the path to the link and requires some disk space fors its i-node data.

What is a the difference between starvation and deadlock?

Starvation implies that a thread cannot make progress because other threads are using resources it needs. Starvation can be recovered from if, for example, the other processes finish. Deadlock is a circular wait without preemption that can never be recovered from.

When using Banker's algorithm for resource allocation, if the system is in an unsafe state, will it always eventually deadlock?

No, because processes may not request their total possible resources, and may release some resources before acquiring others

.**Give a two to three sentence description of a two-level page table.**

A two-level page table uses two levels of page tables where a page table pointer points to the top-level table. Entries in the top-level table point to the lower-level page tables. The lower level tables contain PTEs pointing to the physical locations.

State one advantage and one disadvantage of two-level page tables.

Advantage: Can map sparse addresses more efficiently.

Disadvantage: Require an additional memory reference to translate virtual to physical addresses.

List the four requirements for deadlock.

Mutual exclusion, non-preemptable resources, hold and wait, circular chain of waiting.

5. (12 points total) Demand Paging

For each of the following page replacement policies, list the total number of page faults and fill in the contents of the page frames of memory after each memory reference.

a. (4 points) FIFO page replacement policy:

Reference	A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
Page #1	***A***	*A*	*A*	*A*	*A*	***D***	*D*	*D*	***C***	*C*	*C*	***B***	*B*	*B*	*B*	*B*
Page #2	-	***B***	*B*	*B*	*B*	*B*	***A***	*A*	*A*	***D***	*D*	*D*	*D*	***C***	*C*	*C*
Page #3	-	-	***C***	*C*	*C*	*C*	*C*	***B***	*B*	*B*	***A***	*A*	*A*	*A*	*A*	***D***
Mark X for a fault	*X*	*X*	*X*			*X*	*X*	*X*	*X*	*X*	*X*	*X*		*X*		*X*

Number of FIFO page faults? **12**

b. (4 points) LRU page replacement policy:

Reference	A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
Page #1	***A***	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	***D***	*D*	*D*	*D*	***C***	*B*	***D***
Page #2	-	***B***	*B*	*B*	*B*	*B*	*B*	*B*	*B*	*B*	***A***	*A*	*A*	*A*	*A*	*A*
Page #3	-	-	***C***	*C*	*C*	***D***	*D*	*D*	***C***	*C*	*C*	***B***	*B*	*B*	*C*	*C*
Mark X for a fault	*X*	*X*	*X*			*X*			*X*	*X*	*X*	*X*		*X*		*X*

Number of LRU page faults? **10**

c. (4 points) MIN page replacement policy:

Reference	A	B	C	B	A	D	A	B	C	D	A	B	A	C	B	D
Page #1	***A***	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	*A*	***D***
Page #2	-	***B***	*B*	*B*	*B*	*B*	*B*	*B*	***C***	*C*	*C*	*C*	*C*	*C*	*C*	*C*
Page #3	-	-	***C***	*C*	*C*	***D***	*D*	*D*	*D*	*D*	*D*	***B***	*B*	*B*	*B*	*B*
Mark X for a fault	*X*	*X*	*X*			*X*			*X*			*X*				*X*

Number of page faults? **7** – *Last column may place D anywhere since there are no additional references.*

5. (12 points) Demand paging.

Consider the following page reference string:

A, B, A, C, D, A, D, E, C, D, A, B

We have 3 physical pages, and all are initially all empty. For each of the following page replacement algorithms list whether a page fault occurs on a particular page reference. For grading purposes, the ordering of pages within the 3 available frames does not matter. Mark pages whose reference bit is set using an asterisk (*).

Fill out the tables below and specify the number of page faults:

Least Recently Used (6 pts)					**Second Chance (6 pts)**				
Page Refs	Fault? (Y/N)	3 Page Frames			Page Refs	Fault? (Y/N)	3 Page Frames		
A	*Y*	*A*	-	-	**A**	*Y*	*A**	-	-
B	*Y*	*A*	-	-	**B**	*Y*	*A**	*B**	-
A	*N*	*A*	*B*	-	**A**	*N*	*A**	*B**	-
C	*Y*	*A*	*B*	*C*	**C**	*Y*	*A**	*B**	*C**
D	*Y*	*A*	*D*	*C*	**D**	*Y*	*D**	*B*	*C*
A	*N*	*A*	*D*	*C*	**A**	*Y*	*D**	*A**	*C*
D	*N*	*A*	*D*	*C*	**D**	*N*	*D**	*A**	*C*
E	*Y*	*A*	*D*	*E*	**E**	*Y*	*D**	*A**	*E**
C	*Y*	*C*	*D*	*E*	**C**	*Y*	*C**	*A**	*E*
D	*N*	*C*	*D*	*E*	**D**	*Y*	*C**	*D**	*E*
A	*Y*	*C*	*D*	*A*	**A**	*Y*	*C**	*D**	*A**
B	*Y*	*B*	*D*	*A*	**B**	*Y*	*B**	*D**	*A*

Consider the following function that swaps two elements in the same array. The array is indexed from 0. Assume that only the memory loads and stores are atomic, i.e., assume each instruction in `swap_array()` is atomic.

```
void swap_array(int a[], int i, int j) {
      int tmp;
      tmp = a[j];
      a[j] = a[i];
      a[i] = tmp;
}
```

a) (5 points) Assume input array `A[3] = {1,2,3}`, and assume that `swap_array(A,0,1)`, and `swap_array(A,1,2)`, respectively, are called each in a **different** thread. What are the possible output values of `A[]` after the two threads finish the execution?

```
      Thread 1
swap_array(A[], 0, 1) {
   tmp = A[1];
   A[1] = A[0];
   A[0] = tmp;
}
```

```
      Thread 2
swap_array(A[], 1, 2) {
      tmp = A[2];
      A[2] = A[1];
      A[1] = tmp;
}
```

Note that we have race conditions only on A[1] (i.e., both threads write/read A[1]), and that tmp is a local variable allocated on stack, and thus each thread will have its own copy.

```
      Thread 1
   tmp1 = A[1];
   A[1] = 1;
   A[0] = tmp1;
```

```
      Thread 2
   tmp2 = 3;
   A[2] = A[1];
   A[1] = tmp2;
```

Which further can be simplified as:

```
      Thread 1
   tmp1 = A[1];
   A[1] = 1;
   A[0] = tmp1;
```

```
      Thread 2
   A[2] = A[1];
   A[1] = 3;
```

So there are 6 possibilities:

a) A[2] = A[1]; A[1] = 3; tmp1 = A[1]; A[1] = 1; A[0] = tmp1; Output: {3, 1, 2}
b) A[2] = A[1]; tmp1 = A[1]; A[1] = 3; A[1] = 1; A[0] = tmp1; Output: {2, 1, 2}

c) A[2] = A[1]; tmp1 = A[1]; A[1] = 1; A[1] = 3; A[0] = tmp1; Output: {2, 3, 2}
d) tmp1 = A[1]; A[2] = A[1]; A[1] = 3; A[1] = 1; A[0] = tmp1; Output: {2, 1, 2}
e) tmp1 = A[1]; A[2] = A[1]; A[1] = 1; A[1] = 3; A[0] = tmp1; Output: {2, 3, 2}
f) tmp1 = A[1]; A[1] = 1; A[2] = A[1]; A[1] = 3; A[0] = tmp1; Output: {2, 3, 1}

b) (5 points) Assume `swap_array(A,0,1,)` and `swap_array(A,1,2)`, respectively, are called each in a **different** thread and the output is `A[3] = {1,2,3}`. What are the possible input values of `A[]`?

Assume A = {a, b, c}. Then from previous point, we have = {b,c,a}, {c,a,b},{c,a,a},{b,a,b},{b,c,c}. The outoput has 3 distinct values, and thus only the output {b, c, a} and {c,a,b} are the valid one. Using substitution, we have the following inputs: A = {3,1,2}, and A={2,3,1}

c) (5 points) Assume an initial array `A[3] = {1,2,3}`. Add synchronization primitives to the `swap_array()` code to guarantee that after the execution of `swap_array(A,0,1)` and `swap_array(A,1,2)`, respectively, the output is either `A = {2,3,1}` or `A = {3,1,2}`.

Just apply a lock on all instructions in swap_array(), i.e.,

```
void swap_array(in a[], int i, int j) {
     int tmp;
     acquire(&lock);
     …
     release(&lock);
```

8. (6 points) Resource Allocation Tables:

Consider the following snapshot of a system with 5 processes (P1, P2, P3, P4, P5) and 4 resources (R1, R2, R3, R4). There are no outstanding queued unsatisfied requests.

	Current Allocation				Max Need				Still Needs			
Process	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	2	0	1	4	5	3	2	4	3	3	1	0
P2	2	0	0	3	2	4	3	3	0	4	3	0
P3	0	0	3	4	2	5	5	6	2	5	2	2
P4	1	0	1	0	2	2	3	2	1	2	2	2
P5	4	1	1	0	7	3	2	2	3	2	1	2

i) Is this system currently deadlocked? Why or why not? If not deadlocked, give an execution order. Consider the currently available resources below:

Currently Available Resources

R1	R2	R3	R4
1	2	2	2

The system will be in an unsafe state according to Banker's Algorithm. P4 will be able to finish, but nothing else will be able to finish, we call this state "deadlock."

-0 Correct. It was okay to say that this was an unsafe state but that it is not necessarily deadlock because threads may not need all of the resources they declare as max needs.

-3 For saying "No there is not deadlock"

ii) If a request from a process P1 arrives for (2, 1, 1, 0), should the request be immediately granted? Why or why not? If yes, show an execution order.

Currently Available Resources

R1	R2	R3	R4
3	4	1	0

P1 will finish first. Then either P4 or P5 will finish. All schedules following are valid except those beginning with P1 P4 P3. The request made here is not in addition to nor does it replace the max needs or still needs columns.

-3 For saying it can be granted but not providing a valid schedule.

-5 For saying that it can't or shouldn't be granted for any reason.

4. (18 points total) Scheduling. Consider the following processes, arrival times, waiting, priority, and CPU processing requirements:

Process Name	Arrival Time	Priority	Waits on Lock Held by	Processing Time
1	0	1	None	4
2	2	20	1	3
3	5	1	None	3
4	6	10	3	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). Notes:

- For RR and Priority, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it.
- If a thread tries to acquire a lock, it will do so at the end of its first time slice. If thread A waits for a lock (formerly) held by thread B and B has already finished, it does not wait and acquires the lock immediately.
- Assume the currently running thread is not in the ready queue while it is running.
- Turnaround time is defined as the time a process takes to complete after it arrives

Time	**FIFO**	**RR**	**Priority**
0	**1**	**1**	**1**
1	*1*	*1*	*1*
2	*1*	*2*	*2*
3	*1*	*1*	*1*
4	*2*	*1*	*1*
5	*2*	*3*	*2*
6	*2*	*4*	*2*
7	*3*	*2*	*4*
8	*3*	*3*	*4*
9	*3*	*2*	*3*
10	*4*	*3*	*3*
11	*4*	*4*	*3*
Average Turnaround Time	*((4-0)+(7-2)+(10-5)+(12-6))/ 4 =* ***5***	*((5-0)+(10-2)+ (11-5)+(12-6))/4 =* ***6.25***	*((5-0)+(7-2)+(12-5)+(9-6))/4=* ***5.0***

5. (15 points total) Scheduling. Consider the following processes, arrival times, and CPU processing requirements:

Process Name	Arrival Time	Processing Time
1	0	4
2	2	3
3	5	3
4	6	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). For RR and SRTF, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. Also, assume that the currently running thread is not in the ready queue while it is running. The turnaround time is defined as the time a process takes to complete after it arrives.

Time	**FIFO**	**RR**	**SRTF**
0	**1**	**1**	**1**
1	*1*	*1*	*1*
2	*1*	*2*	*1*
3	*1*	*1*	*1*
4	*2*	*2*	*2*
5	*2*	*3*	*2*
6	*2*	*4*	*2*
7	*3*	*1*	*4*
8	*3*	*2*	*4*
9	*3*	*3*	*3*
10	*4*	*4*	*3*
11	*4*	*3*	*3*
Average Turnaround Time	*((4-0)+(7-2)+ (10-5)+(12-6))/ 4 =* ***5***	*((8-0)+(9-2)+ (12-5)+(11-6))/4 =* ***6.75***	*((4-0)+(7-2)+ (12-5)+(9-6))/4=* ***4.75***

Each column is worth 5 points: 3 for correctness of the schedule (we deducted 1/2/3 points if you made minor/intermediate/major mistakes), and 2 for the average Turnaround time (1 point was deducted for minor errors).

5. (15 points total) Scheduling.
   a. (15 points) Consider the following processes, arrival times, and CPU processing requirements:

Process Name	Arrival Time	Processing Time
1	0	3
2	1	5
3	3	2
4	9	2

For each scheduling algorithm, fill in the table with the process that is running on the CPU (for timeslice-based algorithms, assume a 1 unit timeslice). For RR and SRTF, assume that an arriving thread is run at the beginning of its arrival time, if the scheduling policy allows it. The turnaround time is defined as the time a process takes to complete after it arrives.

Time	**FIFO**	**RR**	**SRTF**
0	**1**	**1**	**1**
1	*1*	*2*	*1*
2	*1*	*1*	*1*
3	*2*	*3*	*3*
4	*2*	*2*	*3*
5	*2*	*1*	*2*
6	*2*	*3*	*2*
7	*2*	*2*	*2*
8	*3*	*2*	*2*
9	*3*	*4*	*2*
10	*4*	*2*	*4*
11	*4*	*4*	*4*
Average Turnaround Time	*3+7+7+3/ 4 =* ***5***	*6+10+4+3/ =* ***5.75***	*3+2+9+3/4=* ***4.25***

Each column is worth 5 points: 3 for correctness of the schedule (we deducted 1/2/3 points if you made minor/intermediate/major mistakes), and 2 for the average Turnaround time (1 point was deducted for minor errors).

4. (24 points total) **CPU scheduling.** Consider the following **single-threaded** processes, arrival times, and CPU processing requirements:

Process ID (PID)	Arrival Time	Processing Time
1	0	6
2	2	4
3	3	5
4	6	2

a) (12 points): For each scheduling algorithm, fill in the table with the ID of the process that is running on the CPU. Each row corresponds to a time unit.
   - For time slice-based algorithms, assume one unit time slice.
   - When a process arrives it is immediately eligible for scheduling, e.g., process 2 that arrives at time 2 can be scheduled during time unit 2.
   - If a process is preempted, it is added at the tail of the ready queue.

Time	FIFO	RR	SJF
0	1	1	1
1	1	1	1
2	1	1	1
3	1	2	1
4	1	1	1
5	1	3	1
6	2	2	4
7	2	1	4
8	2	3	2
9	2	4	2
10	3	2	2
11	3	1	2
12	3	3	3
13	3	4	3
14	3	2	3
15	4	3	3
16	4	3	3

b) (6 points): Calculate the response times of individual processes for each of the scheduling algorithms. The response time is defined as the time a process takes to complete after it arrives.

	PID 1	PID 2	PID 3	PID 4
FIFO	6	8	12	11
RR	12	13	14	8
SJF	6	10	14	2

-0.5 for each mistake

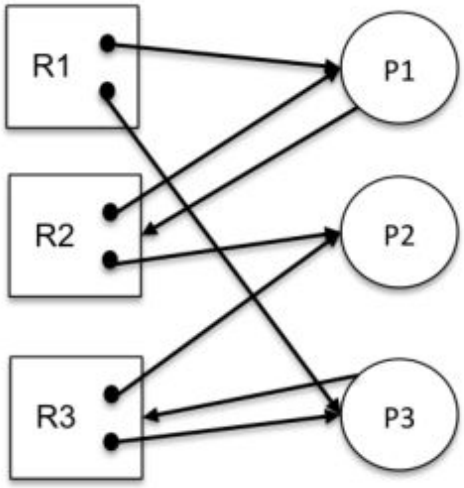
c) (6 points) Consider same processes and arrival times, but assume now a processor with **two** CPUs. Assume CPU 0 is busy for the first two time units. For each scheduling algorithm, fill in the table with the ID of the process that is running on each CPU.

- For any non-time slice-based algorithm, assume that once a process starts running on a CPU, it keeps running on the same CPU till the end.
- If both CPUs are free, assume CPU 0 is allocated first.

Time	CPU #	FIFO	RR	SJF
0	**0**	X	X	X
	1	1	1	1
1	**0**	X	X	X
	1	1	1	1
2	**0**	2	1	2
	1	1	2	1
3	**0**	2	1	2
	1	1	2	1
4	**0**	2	3	2
	1	1	1	1
5	**0**	2	2	2
	1	1	3	1

6	**0**	3	1	4
	1	4	2	3
7	**0**	3	3	4
	1	4	4	3
8	**0**	3	3	
	1		4	3
9	**0**	3	3	
	1			3
10	**0**	3		
	1			3

5. (12 points) **Deadlock**: Consider the following resource allocation graph:



a) (3 points) Does the above allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

No. A possible execution sequence is P2, P1, P3.

1 point for yes/no.
2 points for a possible execution sequence, or an explanation that the graph can not fulfill four deadlock conditions.

b) (3 points) Assume now that P2 also demands resource R1. Does this allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

Yes. If P2 demands R1, none of P1, P2, or P3 will get all resource they need to be processed. Therefore, there is a deadlock.

1 point for yes/no.
2 points for the explanation that all four deadlock conditions are met, or any reasonable explanation about no thread can proceed.

c) (3 points) Assume the allocation graph at point b), and, in addition, assume that R2 has now three instances. Does this allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

No. The only possible execution sequence is P1, P2, P3.

1 point for yes/no.
2 points for a possible execution sequence, or an explanation that the graph can not fulfill four deadlock conditions.

d) (3 points) Add to the original allocation graph an additional process P4 that demands an instance of R1. Does the allocation graph contain a deadlock? Explain your answer using no more than ***two*** sentences.

No. A possible execution sequence is P2, P1, P3, P4.

1 point for yes/no.
2 points for a possible execution sequence. Or any reasonable explanation based on (a).

6. (14 points) **Caching**: Consider a memory consisting of four pages (frames), and consider the following reference stream of virtual pages A, B, C, D, E, C, A, B, C, D, F.

a) (4 points) Consider the LRU page replacement algorithm. Fill in the following table showing all page faults. What is the number of page faults?

Ref Page	A	B	C	D	E	C	A	B	C	D	F
1	A				E					D	
2		B					A				F
3			C								
4				D				B			

9 page faults.

-1 for every wrong page fault
-1 for not writing the number of page faults

b) (4 points) Consider now the MIN page replacement algorithm. Assume the reference stream continues with virtual pages A, C, B, F (i.e., the entire reference stream is A, B, C, D, E, C, A, B, C, D, F, *A, C, B, F*). Fill in the following table showing all page faults. How many page faults are there?

Ref Page	A	B	C	D	E	C	A	B	C	D	F
1	A										
2		B									
3			C								
4				D	E					D	F

7 page faults.

-1 for every wrong page fault

c) (3 points) Consider again the LRU replacement policy, and the original reference stream. What is the minimum memory size (in pages) such that the number of faults to be no larger than 6? Explain.

5 pages. Ignoring F that appears last, all pages in the reference stream fit in a 5 page memory, so for these references we only have 5 compulsory misses. Referring F will cause another page fault, thus 6 page faults in total.

1.5 points for correct answer;
1.5 points for explanation.

We gave partial credit (1 point) if you said 6 pages.

d) (3 points) Replace a **single** reference in the original reference stream (e.g., change the third reference from C to A) such that to reduce the number of page faults by **two** when using LRU. Show the resulting reference stream and the corresponding fault in the following table:

Ref Page	A	B	C	D	E	C	A	C	C	D	F
1	A				E						F
2		B					A				
3			C								
4				D							

*We also gave full points if you gave an example reducing the number of page faults by **more** than two (since we made this clarification during the exam).*

-1 point for every mistake in your page.

a. (5 points) Consider a UNIX system with the following components:

- Disk blocks are 1024 bytes. Sectors are 512 bytes long.
- The inumbers are 4-bytes long.
- An inode contains file attributes and a total of 14 block pointers, including twelve direct pointers, one single indirect entry, and one double indirect entry. The total inode size is 256 bytes.
- Both indirect and double indirect blocks take up an entire disk block.

How many disk reads are needed to read the 300th block of file A into memory, assuming initially only A's inode is in memory? Show your reasoning/steps for partial credit.

One index block contains 1KB/4B=256 entries. The direct entries and the single indirect entry can support up to 10 + 256 blocks, which is less than 300 blocks. So we need to use the double indirect entry. Thus, the number of disk reads is 3 – one read for the double indirect block, one read for the single indirect block, and one read for the data block.

b. (6 points) Using the components described above, list the set of disk blocks that must be read into memory in order to read the 12,000-byte long UNIX file `/usr/cs162/final.doc` in its entirety. Assume no inodes or directories are currently in memory, and that the directories in question all fit into a single disk block each. (Note: this is not always true in reality.)

(1) Read in file header for / (root). The root is always at fixed spot on disk.
(2) Read in first data block for / (root).
(3) Read in file header for `usr`.
(4) Read in first data block for `usr`.
(5) Read in file header for `cs162`.
(6) Read in first data block for `cs162`.
(7) Read in file header for `final.doc`.
(8) Read in first data block for `final.doc`.
(9 – 17) Read in second through 12th data blocks for `final.doc`.
(Note: 12th data block will be only partially full.)

4. (20 points) Concurrency control: Consider the following pseudocode that aims to implement a solution for the Dining Philosopher problem. Note that a philosopher can use any chopstick.

```
// assume chopstick[i].status = FREE, for 1 <= i <= N
get_chopstick(boolean hold_one_chopstick) {
  lock.acquire();
  for (i = 1; i <= N; i++) {
    if (chopstick[i].status == FREE) {
      chopstick[i].status = BUSY;
      return i;
    }
  }
  lock.release();
  return -1;
}

release_chopstick(i) {
  if (i == -1) return;
  chopstick[i].status = FREE;
}


philosopher() {
  plate = FULL;
  while (plate == FULL) {
    chopstick1 = get_chopstick(FALSE);
    if (chopstick1 != -1) {
      chopstick2 = get_chopstick(TRUE);
      plate = EMPTY;
      release_chopstick(chopstick2);
    }
    release_chopstick(chopstick1);
  }
}

main() {
  for (i = 1; i <= N; i++) {
    thread_fork(philosopher());
  }
}
```

a. (2 points) Name an error in how synchronization primitives are used in `get_chopstick()`

There is a missing `lock.release()` *before* `return i`.

b. (10 points) After fixing the error in part (a), does the program work correctly? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

Is not guaranteed to work. Every philosopher can get a chopstick, fail to get the second one, release the chopstick they hold, and repeat!

- *We gave 5 points for an example.*

Add code to `get_chopstick()` *to not give the last chopstick to a philosopher if that philosopher doesn't already own a chopstick. For example:*

```
cnt = 0;
for (i = 1; i < N; i++) {
  if (chopstick[i].status == FREE) {
    cnt++;
  }
}
if (cnt == 1 && hold_one_chopstick == FALSE) {
  return -1;
}
```

- *We subtracted 3 points if you did not give the above solution.*
- *We subtracted 1 point if you said it in words, but not give the code.*
- *We subtracted 1 point if your solution was to use mutex around the entire body of* `philosopher()`*, as we considered this solution to do "excessive locking".*

In addition, you need to check for `chopstick2` *in* `philosopher()`*, i.e.,*

```
if (chopstick2 != -1) { plate = EMPTY; }
```

- *We subtracted 2 point if you did not provide the above fix.*

(We also need to protect the body of `release_chopstick()`*, by* `lock.acquire()` *and* `lock.release()`*. However, we did not subtract any points for this.)*

c. (8 points) Assume `main()` launches N+1 philosopher threads, instead of N. Will the program work correctly given the changes you made for parts (a) and (b)? If it does not, give a simple example to show how the program fails, and provide a fix. If it does, use no more than three sentences to argue why it works.

No change needed, as the modified code in (b) will guarantee that the last chopstick will always be picked by a philosopher that already has another chopstick.

3. (12 points) **Synchronization:** A common parallel programming pattern is to perform processing in a sequence of parallel stages: all threads work independently during each stage, but they must synchronize at the end of each stage at a synchronization point called a ***barrier***. If a thread reaches the barrier before all other threads have arrived, it waits. When all threads reach the barrier, they are notified and can begin the execution on the next phase of the computation.

Example:

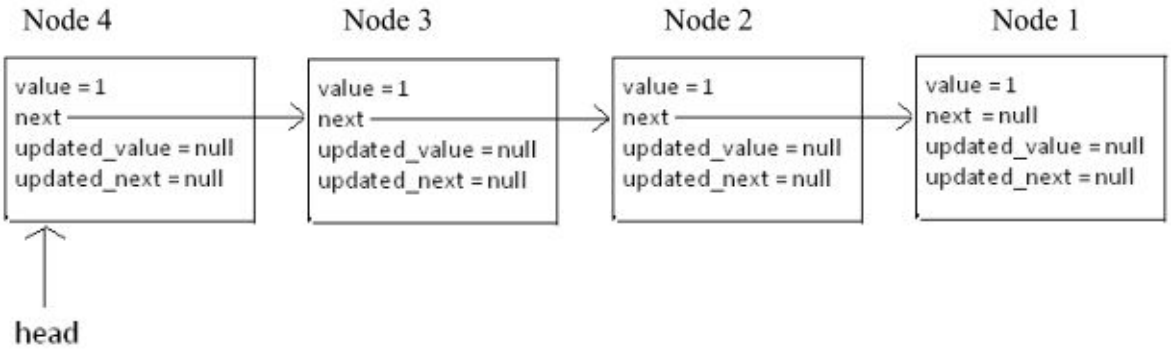
```
while (true) {
        Compute stuff;
        BARRIER();
        Read other threads results;
}
```

a) (4 points) The following implementation of Barrier is incomplete and has two lines missing. Fill in the missing lines so that the Barrier works according to the prior specifications.

```
class Barrier() {
    int numWaiting = 0;        // Initially, no one at barrier
    int numExpected = 0;       // Initially, no one expected
    Lock L = new Lock();
    ConditionVar CV = new ConditionVar();

    void threadCreated() {
        L.acquire();
        numExpected++;
        L.release();
    }
    void enterBarrier() {
        L.acquire();
        numWaiting++;
        if (numExpected == numWaiting) { // If we are the last
            numWaiting = 0;      // Reset barrier and wake threads

            CV.broadcast();      // Fill me in
        } else {     // Else, put me to sleep

            CV.wait(L);          // Fill me in
        }
        L.release() ;
    }
}
```

2 points for CV.broadcast() or CV.wakeAll()
1 point for CV.signal() or CV.wake()

b) (5 points) Now, let us use ***Barrier*** in a parallel algorithm. Consider the linked list below:



In our parallel algorithm, there are four threads (Thread 1, Thread 2, Thread 3, Thread 4). Each thread has its own instance variable ***node***, and all threads share the class variable ***barrier***. Initially, Thread 1's ***node*** references Node 1, Thread 2's ***node*** references Node 2, Thread 3's ***node*** references Node 3, and Thread 4's ***node*** references Node 4.

In the initialization steps, **barrier.threadCreated()** is called once for each thread created, so we have **barrier.numExpected == 4** as a starting condition.

Once all four threads are initialized, each thread calls its **run()** method. The **run()** method is identical for all threads:

```
void run() {
     boolean should_print = true;
     while (true) {
          if (node.next != null) {
               node.updated_value = node.value +
                                    node.next.value;
               node.updated_next = node.next.next;
          } else if (should_print) {
                System.out.println(node.value);
                should_print = false;
          }
          barrier.enterBarrier();
          node.value = node.updated_value;
          node.next = node.updated_next;
          barrier.enterBarrier();
     }
}
```

List all the values that are printed to stdout along with the thread that prints each value. For example, "thread 1 prints 777".

Answer:
Thread 1 prints 1

Thread 2 prints 2
Thread 3 prints 3
Thread 4 prints 4
Note: This is a parallel list ranking algorithm where the final value of each node is its position in the linked list.

1 point for "Thread 1 prints 1"
1 point for "Thread 2 prints 2"
1 point for "Thread 3 prints 3" or "Thread 3 prints 2 + null" or any other answer with an explicit explanation of what happens when you add a number with null
1 point for "Thread 4 prints 4

c) (3 points) In an attempt to speed-up the parallel algorithm from the previous part (2c), you notice that the line **barrier.enterBarrier()** occurs twice in **run()**'s while loop. Can one of these two calls to **barrier.enterBarrier()** be removed while guaranteeing that the output of the previous part (2c) remains unchanged? If your answer is "yes", specify whether you would remove the first or second occurrence of **barrier.enterBarrier()**.

Answer: no

6. (12 points total) Circular queue and concurrency.
Circular queue is a data structure often used to implement various functionalities in the OS. For example, the sending and receiving buffers in TCP are implemented using circular queues. Consider a circular queue of size N. The figure below shows one instance of a circular queue for N=8, where the `head` points to entry 1, and the `tail` to entry 4. The `head` points to the first element in the queue, while the `tail` to the last element in the queue. If the queue is empty, both `head` and `tail` are initialized to -1. Note that both the `head` and `tail` are incremented using modulo N operator. The pseudocode below implements the `enqueue()` operation. We assume that both these functions are synchronized, i.e., they lock the queue object when executing.

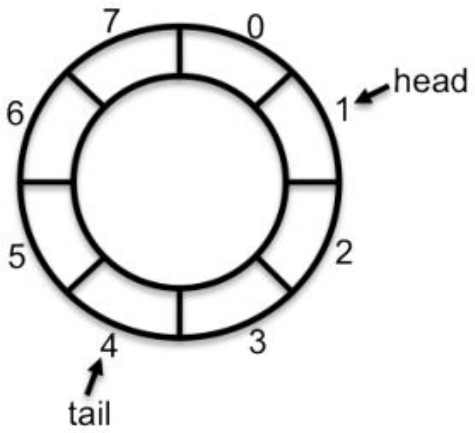


Figure: A circular queue of size N=8 storing 4 entries at positions 1, 2, 3, and 4, respectively.

```
// enqueue new item "data"; return false if queue full
synchronized bool enqueue(item data) {
      if (head == -1) { // queue is empty
            head = tail = 0;
            queue[tail] = data;
      } else { // there is at least one item in the queue
            if ((tail + 1) modulo N != head) {
                  tail = (tail + 1) modulo N;
                  queue[tail] = data;
            } else { // queue full
                  return false;
            }
      }
      return true;
}
```

a) (4 points) Write the pseudocode for the `dequeue()` operation. `dequeue()` should return the item at the head of the queue if queue not empty, and `null`, if queue is empty.

```
synchronized item denque() {
      if (head == -1) {
            Return null;
      } else {
            item = dequeue[head];
            if (head == tail) {
                  head = tail = -1; // empty queue
            } else {
                  head = (head + 1) modulo N;
            }
            return item;
      }
}
```

b) (2 points) Assume N = 8, and assume there were 100 `enqueue()` operations, out of which 5 have failed, and 98 `dequeue()` operations, out of which 7 have failed. Assume initially the queue is empty, and it never gets empty again after the first item is inserted. What are the values of `head` and `tail` after all `enqueue()` and `dequeue()` operations take place?.

The tail is incremented for every successful eqnueue() operation, while head is incremented for every successful dequeue() operations. Since there are 95 successful enqueue() operations, tail = (95-1) modulo 8 = 6. Similarly, since there are 91 successful dequeue() operations, head = 91 modulo 8 = 3.

c) (6 points) Assume circular queue of integers with N = 3, and assume each of the following three operations is called by a different threads. In particular, thread T1 calls `enqueue(10)`, thread T2 calls `enqueue(12)`, while thread T3 calls `dequeue()`. Initially, the circular queue is empty. Please specify what are the possible entries in the circular queue after all three operations are performed, and what is the value returned by the `dequeue()` operation. Use the following table to show your answer. (If a queue entry is outside [head:tail], then use "-" to represent its content.)

thread execution order	head	tail	queue[0]	queue[1]	queue[2]	dequeue value
T1, T2, T3	*1*	*1*	-	*12*	-	*10*
T1, T3, T2	*0*	*0*	*12*	-	-	*10*
T3, T1, T2	*0*	*1*	*10*	*12*	-	*null*
T2, T1, T3	*1*	*1*	-	*10*	-	*12*
T2, T3, T1	*0*	*0*	*10*	-	-	*12*
T3, T2, T1	*0*	*1*	*12*	*10*	-	*null*