2. (25 points) Synchronization primitives: Consider a machine with hardware support for a single thread synchronization primitive, called Compare-And-Swap (CAS). Compare-and-swap is an atomic operation, provided by the hardware, with the following pseudocode:

```
int compare_and_swap(int *a, int old, int new) {
  if (*a == old) {
     *a = new;
     return 1;
  } else {
     return 0;
  }
}
```

Your first task is to implement the code for a simple spinlock using compare-and-swap. You are not allowed to assume any other hardware or kernel support exists (*e.g.,* disabling interrupts). You may assume your spinlock will be used correctly (*i.e.*, no double release or release by a thread not holding the lock)

a. (3 points) Fill in the code for the `spinlock` data structure.

```
struct spinlock { /* Fill in */

int value = 0;
```

We deducted one point for extraneous statements.

```
}
```

b. (4 points) Fill in the code for the `acquire` data function.

```
void acquire(struct spinlock *lock) { /* Fill in */

while (cas(&lock->value, 0, 1) == 0)
   ;    /* spin */
```

We deducted two points for extraneous statements, two points for a missing while and four points if your solution did not work.

```
}
```

c. (4 points) Fill in the code for the `release` data function.

```
void release(struct spinlock *lock) { /* Fill in */

lock->value = 0;
```

We deducted two points for extraneous statements, one point for an unnecessary Compare-and-Swap (stores of a word are atomic), and four points if your solution did not work.

```
}
```

After completing your implementation, you realize that using a spinlock is inefficient for applications that may hold the lock for a long time. You consider using the following two primitives to implement more efficient locks: `atomic_sleep` and `wake`.

`atomic_sleep` is an atomic operation, provided by the hardware, with the following pseudocode:

```
void atomic_sleep(struct *lock, int *val1, int val2){
      *val1 = val2;  /* set val1 to val2 */
      enqueue(lock);     /* put current thread on a
                            lock's wait queue*/
      sleep();     /* put current thread to sleep */
}
```

`wake` is non-atomic with the following pseudocode:

```
void wake(struct lock *lock){
  dequeue(); /* remove a thread (if any) from lock's
                wait queue and add it to the
                scheduler's ready queue */
}
```

Your second task is to reimplement your lock code more efficiently using `atomic_sleep` and `wake`. You may use Compare-And-Swap if you want. You are not allowed to assume any other hardware or kernel support exists (*e.g.,* disabling interrupts).

d. (4 points) Fill in the code for the **new** `lock` data structure.

```
struct lock { /* Fill in */

int guard = 0;
int value = 0;
Queue queue = NIL;
```

We deducted two points for a missing guard or value and one point for extraneous variables.

```
}
```

e. (5 points) Fill in the code for the **new** `acquire` data function.

```
void acquire(struct lock *lock) { /* Fill in */

   while (1) {
      while (cas(&lock->guard, 0, 1) == 0);
      if (value == 1) {
         atomic_sleep(&lock, &lock->guard, 0);
      } else {
         lock->value = 1;
         lock->guard = 0;
         return;
      }
   }
```

We expected solutions that used a spinlock on a guard for protecting the lock variable, and atomic_sleep for efficient waiting for the lock. We deducted two points for extraneous statements, two points for not setting the lock variable, one point for a missing outer while loop, 2 points for a missing guard, 2 points for not sleeping, one point for misusing the guard, and one point for a dangerous double release of the guard in acquire and release.

```
}
```

f. (5 points) Fill in the code for the **new** `release` data function.

```
void release(struct lock *lock) { /* Fill in */

   while (cas(&lock->guard, 0, 1) == 0);
   lock->value = 0;
   wake(&lock);
   lock->guard = 0;
```

We deducted two points for extraneous statements, two points for no guard, two points for failing to wake a waiting thread, and two points if your solution had other errors, such as not releasing the lock.

```
}
```

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:

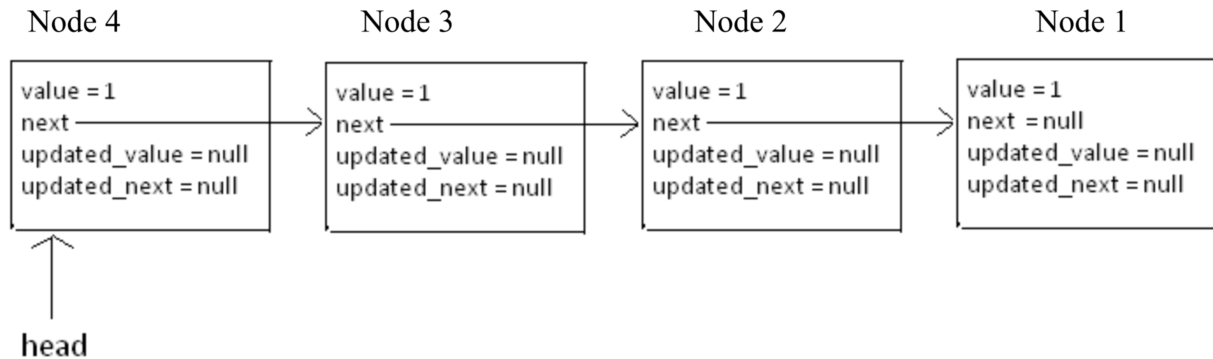
```
     while (true) {
             Compute stuff;
             BARRIER();
             Read other threads results;
   }
```

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

4. (22 points) Deadlock:

A restaurant would like to serve four dinner parties, P1 through P4. The restaurant has a total of 8 plates and 12 bowls. Assume that each group of diners will stop eating and wait for the waiter to bring a requested item (plate or bowl) to the table when it is required. Assume that the diners don't mind waiting. The maximum request and current allocation tables are shown as follows:

Maximum Request	**Plates**	**Bowls**
P1	7	7
P2	6	10
P3	1	2
P4	2	4

Current Allocation	**Plates**	**Bowls**
P1	2	3
P2	3	5
P3	0	1
P4	1	2

a. (4 points) Determine the Need Matrix for plates and bowls.

Need	**Plates**	**Bowls**
P1	*5*	*4*
P2	*3*	*5*
P3	*1*	*1*
P4	*1*	*2*

b. (7 points) Will the restaurant be able to feed all four parties successfully? *Clearly explain your answer – specifically, why no or why/how there is a safe serving order.*

The vector of available resources is A = (2, 1).
First, serve P3, and A will be (2, 2).
Then, serve P4, and A will be (3, 4).
There are not enough resources available to serve P1 or P2. Therefore, the original resource allocation state is ***unsafe****. The restaurant cannot feed all four parties successfully.*

If you did not say that the allocation is unsafe, we deducted one point.
If you said "yes" or safe, we deducted 4 points.
If you said that there was deadlock or starvation, we deducted 3 points (saying you can't finish ever, cost2 points).
If your solution did not include the Banker's Algorithm or you were not specific in your explanation, we deducted 2 points

4. (continued) Deadlock

c. (11 points) Assume a new dinner party, P5, comes to the restaurant at this time. Their maximum needs are 5 plates and 3 bowls. Initially, the waiter brings 2 plates to them. In order to be able to feed all five parties successfully, the restaurant needs more plates.

i. (2 points) Determine the new Need Matrix for plates and bowls.

Need	**Plates**	**Bowls**
P1	*5*	*4*
P2	*3*	*5*
P3	*1*	*1*
P4	*1*	*2*
P5	*3*	*3*

ii. (6 points) At least how many plates would the restaurant need to add?

If add 1 plate, there are 9 plates and 12 bowls totally.

Initially, A = (1, 1).
Serve P3, A = (1, 2).
Serve P4, A = (2, 4).
So there are not enough resources for serving the 5 parties.

If add 2 plates, there are 10 plates and 12 bowls totally.

Initially, A = (2, 1).
Serve P3, A = (2, 2).
Serve P4, A = (3, 4).
Serve P5, A = (5, 4).
Serve P1, A = (7, 7).
Serve P2, A = (10, 12).

Therefore, at least 2 plates are needed.

We treated parts (ii) and (iii) as a combined 9 points.
If you had an incorrect number of plates, but the correct ordering, we deducted 5 points.
If you had an incorrect number of bowls, but the correct number of plates, and an incorrect ordering, we deducted 4 points.
If your had the correct number of plates, but an incorrect ordering, we deducted 6 points.

iii. (3 points) Show a safe serving sequence.

A safe serving sequence is P3-P4-P5-P1-P2.

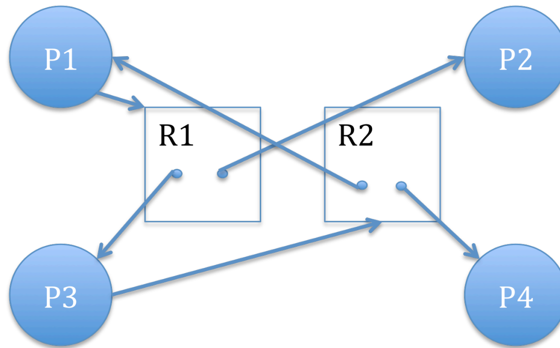
Question 2. *Deadlock (15 points)*

Consider a system with four processes P1, P2, P3, and P4, and two resources, R1, and R2, respectively. Each resource has two instances. Furthermore:

- P1 allocates an instance of R2, and requests an instance of R1;
- P2 allocates an instance of R1, and doesn't need any other resource;
- P3 allocates an instance of R1 and requires an instance of R2;
- P4 allocates an instance of R2, and doesn't need any other resource.

(5 points each question)

(a) Draw the resource allocation graph.



(b) Is there a cycle in the graph? If yes name it.

P2 and P4 are running, P1 is waiting for R1, and P2 is waiting for R2.

(c) Is the system in deadlock? If yes, explain why. If not, give a possible sequence of executions after which every process completes.

There is a cycle, but no deadlock.

- P2 finishes, release R1;
- P4 finishes, release R2;
- P1 acquires R1, finishes and release R1,R2;
- P3 acquires R2, finishes and release R1,R2;

Question 4. *Scheduling (20 points)*

Consider three threads that arrive at the same time and they are enqueued in the ready queue in the order T1, T2, T3.

Thread T1 runs a four-iteration loop, with each iteration taking one time unit. At the end of each iteration, T1 calls yield; as a result, T1 is placed at the end of the ready queue. Threads T2 and T3 both run a two-iteration loop, which each iteration taking three time units. At the end of first iteration, T2 synchronizes with T3, i.e., T2 cannot start the second iteration before T3 finishes the first iteration, and vice versa. While waiting, T2 (T3) is placed in the waiting queue; once T3 (T2) finishes its first iteration, T2 (T3) is placed at the end of the ready queue. Each process exits after finishing its loop.

Assume the system has one CPU. On the timeline below, show how the threads are scheduled using two scheduling policies (FCFS and Round Robin). For each unit of time, indicate the state of the thread by writing "R" if the thread is running, "A" if the thread is in the ready queue, and "W" if the thread is in the waiting queue (e.g., T2 waits for T3 to finish the first iteration, before T2 can run its second iteration).

(a) (6 points) **FCFS (No-preemption)** FCFS always selects the thread at the head of the ready queue. A thread only stops running when it calls yield or waits to synchronize with another thread. What is the average completion time?

T1	R	A	A	A	A	A	A	A	A	A	R	A	A	A	R	R
T2	A	R	R	R	W	W	W	A	A	A	A	R	R	R		
T3	A	A	A	A	R	R	R	R	R	R						

(b) (6 points) **Round Robin (time quantum = 2 units)** When a thread is preempted it is moved at the end of the ready queue. What is average completion time?

T1	R	A	A	A	A	R	A	A	A	R	A	A	A	A	R	
T2	A	R	R	A	A	A	R	W	A	A	R	R	A	A	A	R
T3	A	A	A	R	R	A	A	R	R	A	A	A	R	R		

(c) (8 points) Assume there are two processors P1 and P2 in the system. The scheduler follows the policy of FCFS with no preemption. When the scheduler assigns tasks, always assign a task to P1 before assigning to P2. Instead of using "R" to mark running, use "P1" or "P2" to indicate where the task runs. What is the average completion time?

T1	P1	A	A	P1	A	A	A	P1	P1							
T2	P2	P2	P2	W	P2	P2	P2									
T3	A	P1	P1	P1	P1	P1	P1									

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

-4 for using the wrong algorithm (e.g., SRTF instead of SJF)

-1 for those who assumed that the newly arrived processes go to the front of the queue (unlike previous years, the problem expects processes to be added to the end of the queue when they arrive)
-1 for each mistaken sequence (of processes).

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

-2 for using the wrong algorithm (e.g., SRTF instead of SJF)
-1 for those who assumed that the newly arrived processes go to the front of the queue (unlike previous years, the problem expects processes to be added to the end of the queue)
-1 for each mistaken sequence (of processes).

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. (18 points) Paging:

Suppose you have a system with 32-bit pointers and 4 megabytes of physical memory that is partitioned into 8192-byte pages. The system uses an Inverted Page Table (IPT). Assume that there is no page sharing between processes.

a. (8 points) Describe what page table entries should look like. Specifically, how many bits should be in each page table entry, and what are they for? Also, how many page table entries should there be in the page table?

Virtual addresses are 32 bits, and split into two parts. The page number is the first 19 bits, and the offset within the page is the last 13 bits (2^{13} = 8,192).

Virtual Page Number	*Offset*
19 bits	*13 bits*

The inverted page table is a mapping of physical addresses to virtual addresses. Memory is 4 megabytes, partitioned into 512 pages. Therefore the inverted page table will consist of 512 entries. Each of these entries must have:

19 bits for the virtual page number of the physical page.
Some number of bits (16 in Unix) for the processs ID of the process that owns the page.
Protection bits (r/w/x)

We awarded two points each for: the correct number of IPT entries, storing the virtual page number in the IPT, storing the process ID in the IPT, and storing the protection bits(any reasonable set was accepted) in the IPT.
We subtracted one point for each extraneous item that you stored in the IPT, and subtracted one point for storing the physical page number instead of the virtual page number in the IPT.

b. (5 points) Describe how an IPT is used to translate a virtual address into a physical address.

A virtual address is translated to a physical address hashing virtual addresses (worth two points). Thus, in the normal case, the translation may be found in a few memory lookups rather than an entire table traversal. If it is found, and the owning process is equal to the current running process (owning PID check is worth 2 points), then its index in the table is the frame number of the physical page. If it is not found, then a memory fault must occur (not found fault is worth 1 point).
We also accepted a general search of the IPT, as described in the book and midterm review session.

c. (3 points) How can you make an IPT more efficient? *Explain your solution and how it works in detail.*
Add a Translation Lookaside Buffer (TLB), an associative memory that can perform fast lookup on virtual addresses and provide the translation in much less time than it takes to perform a memory access. TLB's are effective, especially when combined with caches, because programs tend to have locality in accessing pages. The use of a TLB was worth one point, the explanation was worth two points.

Alternatively, if you used the book solution for part (b), we accepted a hash-based enhancement for this part.

d. (2 points) What effect, if any, does your solution in part (c) have on what happens on a context switch?
On a context switch, the TLB must be flushed. That's it.

Alternatively, if you used the book solution for part (b) and a hash enhancement for part (c), then we accepted the answer "nothing" for this part.

Problem 4: Virtual Memory [20 pts]

Consider a multi-level memory management scheme with the following format for virtual addresses:

Virtual Page # (10 bits)	Virtual Page # (10 bits)	Offset (12 bits)

Virtual addresses are translated into physical addresses of the following form:

Physical Page # (20 bits)	Offset (12 bits)

Page table entries (PTE) are 32 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory):

Physical Page # (20 bits)	OS Defined (3 bits)	0	Large Page	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid

Here, "Valid" means that a translation is valid, "Writeable" means that the page is writeable, "User" means that the page is accessible by the User (rather than only by the Kernel). *Note: the phrase "page table" in the following questions means the multi-level data structure that maps virtual addresses to physical addresses.*

Problem 4a[2pts]: How big is a page? Explain.

Since the offset is 12 bits, then a page is 2^{12}=4096 bytes. You had to show an actual calculation and mention the offset to get full credit.

Problem 4b[2pts]: Suppose that we want an address space with one physical page at the top of the address space and one physical page at the bottom of the address space. How big would the page table be (in bytes)? Explain.

The page table of interest has two non-null pointers for the first level, pointing at 2 second-level elements of the page table. Each element is a page in size. Thus, there are 3 pages = 3×4096 = 12288

Problem 4c[2pts]: What is the maximum size of a page table (in bytes) for this scheme? Explain.

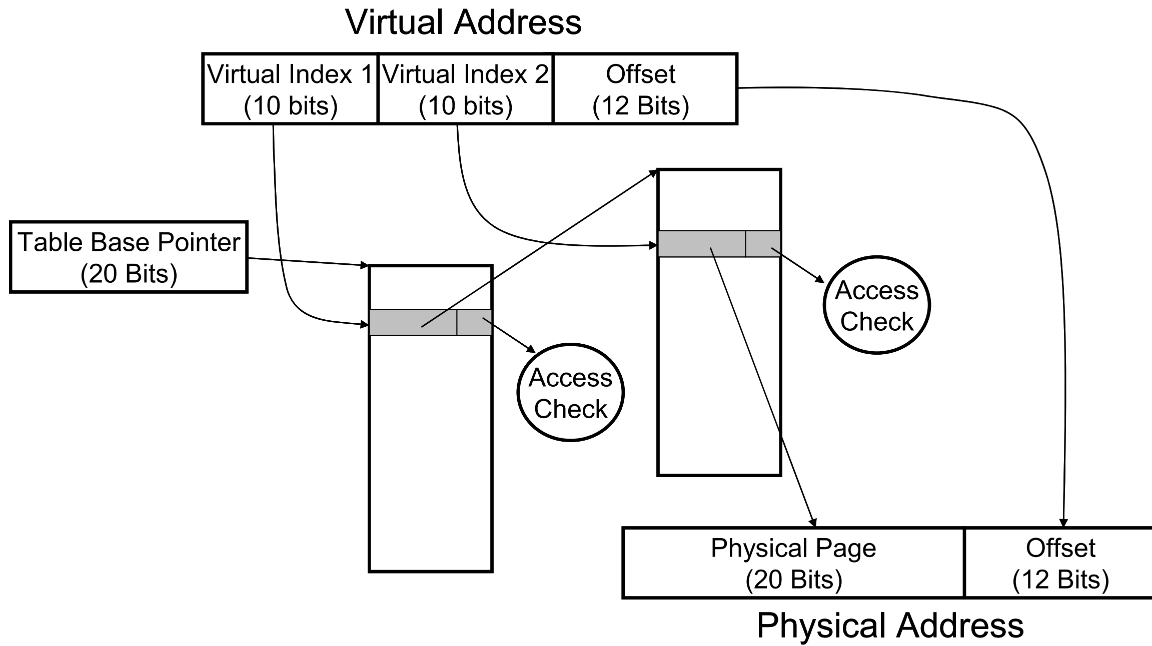
The maximum page table size has an entry for every virtual address. Thus -- all pointers non-null at the top level, each of which points at a second-level element of the page table. Thus, the total size of the page table has 1 + 1024 = 1025 page table elements = 1025×4096 = 4198400 bytes.

Problem 4d[2pts]: How big would each entry of a fully-associative TLB be for this management scheme? Explain.

Each entry of a fully-associative cache needs enough storage for (1) the cache tag and (2) the valid bit. The tag for a TLB is the virtual page number, which is 20 bits. A valid bit is 1 bit. Thus, the simplest correct answer for this question is 21 + 32 (size of PTE) = 53 bits. A slightly more sophisticated answer would recognize that there are 4 bits in the PTE that are not needed by the hardware (bits 8-11). Thus, one could say that there are 21 + 28 = 49 bits.

Problem 4e[2pts]: Sketch the format of the page-table for the multi-level virtual memory management scheme of (4a). Illustrate the process of resolving an address as well as possible.

We were looking for something like the following diagram:



Problem 4f[10pts]: Assume the memory translation scheme from (4a). Use the Physical Memory table given on the next page to predict what will happen with the following load/store instructions. Assume that the base table pointer for the current ***user level process*** is `0x00200000`.

Addresses are virtual. The return value for a load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. Possible errors are: **invalid, read-only, kernel-only.**
Hint: Don't forget that Hexidecimal digits contain 4 bits!

Instruction	Result
Load [0x00001047]	0x50
Store [0x00C07665]	ok
Store [0x00C005FF]	ERROR: read-only
Load [0x00003012]	*ANS: 0x84*

Instruction	Result
Store [0x02001345]	*ANS: Ok*
Load [0xFF80078F]	*ANS: ERROR: Invalid*
Load [0xFFFFF005]	*ANS: 0x66*
Test-And-Set [0xFFFFF006]	*ANS: ERROR: Read-only*

Physical Memory [All Values are in Hexidecimal]

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
00000010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
00001010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
00001020	40	03	41	01	30	01	31	03	00	03	00	00	00	00	00	00
00001030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
00001040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
….																
00002030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
00002040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
00002050	10	00	31	01	10	03	31	01	12	03	30	00	10	00	10	01
….																
00004000	30	00	31	01	11	01	33	03	34	01	35	00	43	38	32	79
00004010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
00004020	23	03	20	03	00	01	62	08	99	86	28	03	48	25	34	21
….																
00100000	00	00	10	65	00	00	20	67	00	00	30	00	00	00	40	07
00100010	00	00	50	03	00	00	00	00	00	00	00	00	00	00	00	00
…																
00103000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
00103010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	67
…																
001FE000	04	15	00	00	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
001FE010	10	15	00	67	10	15	10	67	10	15	20	67	10	15	30	67
…																
001FF000	00	00	00	00	00	00	00	65	00	00	10	67	00	00	00	00
001FF010	00	00	20	67	00	00	30	67	00	00	40	65	00	00	50	07
…																
001FFFF0	00	00	00	00	00	00	00	00	10	00	00	67	00	10	30	65
…																
00200000	00	10	00	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	40	07	00	10	50	07	00	10	60	07	00	10	70	07
00200020	00	10	00	07	00	00	00	00	00	00	00	00	00	00	00	00
…																
00200FF0	00	00	00	00	00	00	00	00	00	1F	E0	07	00	1F	F0	07
…																

3. (17 points total) Memory management:

a. (7 points) Consider a memory system with a cache access time of 10ns and a memory access time of 200ns, *including the time to check the cache*. What hit rate *H* would we need in order to achieve an effective access time 10% greater than the cache access time? (Symbolic and/or fractional answers are OK)

Effective Access Time: $T_e = H * T_c + (1 - H) * T_m$,
where $T_c = 10ns$, $T_e = 1.1 * T_c$, *and* $T_m = 200ns$.

Thus, $(1.1)(10) = 10H + (1 - H)200$

$11 = 10H + 200 - 200H$

$-189 = -190H$

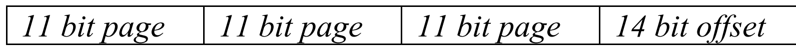
$H = 189/190$

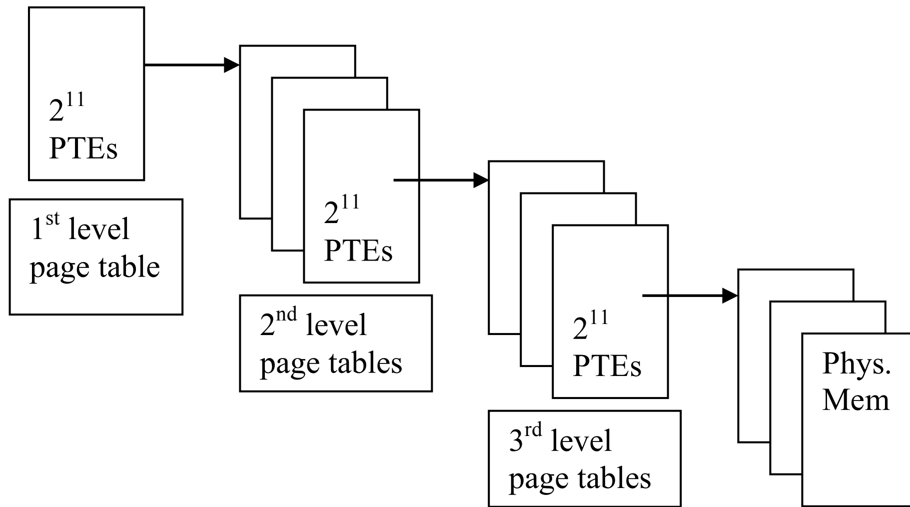
We awarded 4 pts for the correct formula and 3 pts for the correct answer. Many students missed the fact that the miss time includes ***both*** *the memory access time and the cache access time. If the formula was missing the cache access time, we deducted two points – if the answer based upon this incorrect formula was correct, we did not deduct any additional points.*

b. (10 points) Suppose you have a 47-bit virtual address space with a page size of 16 KB and that page table entry takes 8 bytes. How many levels of page tables would be required to map the virtual address space if every page table is required to fit into a single page? Be explicit in your explanation and show how a virtual address is structured.

A 1-page page table contains 2,048 or 2^{11} *PTEs* ($2^3 * 2^{11} = 2^{14}$ *bytes), pointing to* 2^{11} *pages (addressing a total of* $2^{11} * 2^{14} = 2^{25}$ *bytes). Adding a second level yields another* 2^{11} *pages of page tables, addressing* $2^{11} * 2^{11} * 2^{14} = 2^{36}$ *bytes. Adding a third level yields another* 2^{11} *pages of page tables, addressing* $2^{11} * 2^{11} * 2^{11} * 2^{14} = 2^{47}$ *bytes. So, we need* ***3 levels****.*

The correct answer is worth 5 pts. Correct reasoning is worth up to 5 pts (1 pt for identifying that there are 2^{11} *PTEs per page, 2 pts for describing how page tables are nested, and 2 pts based upon the quality of the argument).*


11 bit page
11 bit page
11 bit page
14 bit offset


2^11 PTEs
1st level page table
2^11 PTEs
2nd level page tables
2^11 PTEs
3rd level page tables
Phys. Mem

Question 6. Caches *(20 points)* A tiny system has 1-byte addresses and a 2-way associative cache with four entries. Each block in the cache holds **two** bytes. The cache controller uses the LRU policy for evicting from cache when both rows with the same "index" are full.

(a) () Use the figure below to indicate the number of bits in each field.

6 bits	1 bits	1 bits
cache tag	index	byte select

(b) Assume the following access sequence to the memory: 0xff, 0x22, 0x27, 0x24, 0x27, 0xff, 0xf0, 0x24, 0x27, 0x22. Fill in the following table with the addresses whose content is in the cache. Initially assume the cache is empty. The first entry (i.e., the one corresponding to address 0xff) is filled for you.

		0xff	0x22	0x27	0x24	0x27	0xff	0xf0	0x24	0x27	0x22
Set 0	Index: 0				0x24, 0x25				0x24, 0x25		
	Index: 0							0xf0, 0xf1			
Set 1	Index: 1	0xfe, 0xff		0x26, 0x27		0x26, 0x27				0x26, 0x27	
	Index: 1		0x22, 0x23				0xfe, 0xff				0x22, 0x23

Note: each time a byte is accessed, the entire block to which that block belongs is loaded in memory. For example when byte at address 0xff is read, both that byte and the byte at the address 0xfe are read.

(c) How many cache misses did the access sequence at point (b) cause? What is the hit rate?

7 misses, hit rate = 3/10 = 30%

(d) How many compulsory misses (i.e., misses which could never be avoided) did the access pattern at point (b) cause?

5 (0xff, 0x22, 0x27, 0x24, 0xf0)

(e) Assuming the cache access time is 10ns, and that the miss time is 100ns, what is the average access time assuming the access pattern at point (b)?

10ns * 3/10 + 100ns * 7/10 = 73ns

6. (10 points total) Caching: Assume a computer system employing a cache, where the access time to the main memory is 100 ns, and the access time to the cache is 20ns.

a. (2 points) Assume the cache hit rate is 95%. What is the average access time?

*Average Access Time = Hit*cache_access_time + (1-Hit)*memory_access_time*
*= 0.95*20 ns + 0.05*100 ns = 24 ns*
*Alternatively, we accepted solutions that included the cache time in the memory access time: AAT = 0.95 * 20 ns + 0.05 * (20 ns + 100 ns) = 25 ns.*
We subtracted one point for minor errors.

b. (2 points) Assume the system implements virtual memory using a two-level page table with no TLB, and assume the CPU loads a word X from main memory. Assume the cache hit rate for the page entries as well as for the data in memory is 95%. What is the average time it takes to load X?

*The Average Memory Access Time for X (AMAT) requires three memory accesses, two for each page entry, and one for reading X: 3*24 = 72 ns. The alternate solution from (a) yields 3*25 = 75 ns. We only accepted the alternate solution for (b) if you derived the same value for (a).*

c. (3 points) Assume the same setting as in point (b), but now assume that page translation is cached in the TLB (the TLB hit rate is 98%), and the access time to the TLB is 16 ns. What is the average access time to X?

*AAT_X is TLB_hit *(TLB_access_time + AAT) + (1-TLB_hit) * (3 * AAT):*
0.98(16ns + 24ns) + 0.02*(72ns) = 0.98*40ns + 1.44ns = 40.64ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns) = 41.68ns*

It was acceptable to include the TLB time in the TLB_miss calculation:
*TLB_hit *(TLB _time + AMAT) + (1-TLB_hit) * (3 * AMAT +TLB_time).*
0.98(16ns + 24ns) + 0.02*(72ns + 16ns) = 0.98*40ns + 0.02*88ns = 40.96ns*
Alternate AAT from (a): 0.98(16ns + 25ns) + 0.02*(75ns +16ns) = 42ns*
We subtracted one point for each minor error.

d. (3 points) Assume we increase the cache size. Is it possible that this increase to lead to a decrease in the cache hit rate? Use no more than three sentences to explain your answer.

Yes, using a FIFO replacement scheme could result in Belady's anomaly. Also, using the same hash function while increasing the cache size could cause more collisions and reduce the hit rate. The correct answer was worth one point and the justification was worth two points.