1. You are given a (tiny) memory system with the following parameters:

- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB total memory
- Physical memory consists of 128 pages, or 4 KB total memory

The system uses a multi-level page table. Each page directory entry (PDE) is 1 byte long. The uppermost bit is a valid bit, and the lower 7 bits point to the physical page of the page table entry.

A page table entry (PTE) follows a similar format. The upper bit is a valid bit, and the lower 7 bits refer to the physical page for that virtual page (if the valid bit is 1).

You are given the value of the Page Directory Base Register (PDBR) which tells you where the page directory is located.

You are also given a complete physical memory dump of all 128 pages.

You are then given a list of virtual addresses to translate.

Use the PDBR to find the relevant page table entries for this virtual page. Then find if it is valid. If so, use the translation to form a final physical address. Using this address, you can find the VALUE that the memory reference is looking for. Of course, the virtual address may not be valid and thus generate a fault.

PDBR: 108 (decimal) [This means the page directory is held in this page]

Virtual Address 611c: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 611c: Translates To What Physical Address (And Fetches what Value)? Or Fault?

Virtual Addresses are 15 bits.

Translate address to binary: 110 0001 0001 1100

VPN: 10 bits: 1100001000
Offset: 5 bits: 11100

PDE: 5 bits: 11000 (upper bits)
PTE: 5 bits: 01000 (lower bits)

Directory is at page 108 (as stated in the problem). We use the upper 5 bits of the VPN to get the directory entry. It is the 11000 (0x18) or 24th byte (counting from 0) of the page

page 108: 83 fe e0 da 7f d4 7f eb be 9e d5 ad e4 90 d6 92 d8 c1 f8 9f e1 ed e9 a1 e8 c7 c2 a9 d1 db ff

a1 = 1 0100001 == valid and the page of PTE is (0x21) 33.

Now that we have the page table entry, we use the lower 5 bits of the VPN to get the offset into the PTE. It is the 01000 or (0x8) 8th entry (starting from 0) of page 33.

page 33: 7f 7f 7f 7f 7f 7f 7f 7f b5 7f 9d 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f

b5 = 1 0110101 == valid and the PFN = (0x35) = page 53

We now have the correct PFN (53). To get the physical address we combine the PFN (0x35) 011 0101 with the offset 11100 => 0110 1011 1100 => \textbf{0x6bc}

The value is the 1 1100 (0x1c) 28th byte (starting from 0) of the 53rd page:

page 53: 0f 0c 18 09 0e 12 1c 0f 08 17 13 07 1c 1e 19 1b 09 16 1b 15 0e 03 0d 12 1c 1d 0e 1a \textbf{08} 18 11 00

So the byte at that address is \textbf{0x8}. 


Assuming the same PDBR: 108 (decimal) resolve the following addresses:

Virtual Address 3da8:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
  --> pte index:0xd [decimal 13] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 17f5:
  --> pde index:0x5 [decimal 5] pde contents:0xd4 (valid 1, pfn 0x54 [decimal 84])
  --> pte index:0x1f [decimal 31] pte contents:0xce (valid 1, pfn 0x4e [decimal 78])
  --> Translates to Physical Address 0x9d5 --> Value: 1c

Virtual Address 7f6c:
  --> pde index:0x1f [decimal 31] pde contents:0xff (valid 1, pfn 0x7f [decimal 127])
  --> pte index:0x1b [decimal 27] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 0bad:
  --> pde index:0x2 [decimal 2] pde contents:0xe0 (valid 1, pfn 0x60 [decimal 96])
  --> pte index:0x1d [decimal 29] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 6d60:
  --> pde index:0x1b [decimal 27] pde contents:0xc2 (valid 1, pfn 0x42 [decimal 66])
  --> pte index:0xb [decimal 11] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 2a5b:
  --> pde index:0xa [decimal 10] pde contents:0xd5 (valid 1, pfn 0x55 [decimal 85])
  --> pte index:0x12 [decimal 18] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 4c5e:
  --> pde index:0x13 [decimal 19] pde contents:0xf8 (valid 1, pfn 0x78 [decimal 120])
  --> pte index:0x2 [decimal 2] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
  --> Fault (page table entry not valid)

Virtual Address 2592:
  --> pde index:0x9 [decimal 9] pde contents:0x9e (valid 1, pfn 0x1e [decimal 30])
  --> pte index:0xc [decimal 12] pte contents:0xbd (valid 1, pfn 0x3d [decimal 61])
  --> Translates to Physical Address 0x7b2 --> Value: 1b

Virtual Address 3e99:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
  --> pte index:0x14 [decimal 20] pte contents:0xca (valid 1, pfn 0x4a [decimal 74])
  --> Translates to Physical Address 0x959 --> Value: 1e
2. You have been hired by the VFFA (Vassar Fictitious Food Association) to help the Retreat with their very popular Chili Wednesdays (ed. note: rest in peace chilli wednesdays). To cut down on lines, they are thinking of moving to an automated ticketing system and a "Now Serving Number XX" display like you would see in a supermarket deli. You job is to implement the software in C to simulate this system to see if it will be effective.

To implement this system you must create a `struct chili` structure and the four functions described below.

```c
void chili_init( struct chili *chili) -- This function is called exactly once at the start of the simulation before there are any students or chili servers in the Retreat.

int chili_get_ticket(struct chili *chili) -- This function is called when a new student enters the Retreat and wants to order chilli. This function returns the student's ticket number, which is the smallest integer that has not been returned to another student. In other words, the first student to call this function gets the number 1, the second student who calls this function gets the number 2, and so on. No two students must ever receive the same ticket number. Once a student receives a ticket they are free to get other items in the Retreat (you do not need to implement this mechanism).

void chili_wait_turn(struct chili *chili, int number) -- This function is called when a student is ready to order their chilli. Number is the value returned to the student from the `chili_get_ticket()` function. This function must not return until there is an available chili server and the "Now Serving Number XX" display shows a number at least as high as number. Once this function returns, the student will go to the counter and order their chilli (you do not need to implement this mechanism).

void chili_wait_customer(struct chili *chili) -- This function is called when a new server arrives or when an existing server is ready to serve a new student. This function must increment the value displayed in the "Now Serving Number XX" display. This function must not return until there is a student ready for service. Once this function returns the server will serve the chilli to the student (you do not need to implement this mechanism).
```
Implementation details:

- You must write your solution in C using the following functions for locks and condition variables:
  - `lock_init(struct lock *lock)`
  - `lock_acquire(struct lock *lock)`
  - `lock_release(struct lock *lock)`
  - `cond_init(struct condition *cond)`
  - `cond_wait(struct condition *cond, struct lock *lock)`
  - `cond_signal(struct condition *cond, struct lock *lock)`
  - `cond_broadcast(struct condition *cond, struct lock *lock)`
- Use only these functions (i.e., no semaphores or other synchronization primitives).
- You may not use more than one lock inside your `struct chili` definition.
- You may assume that someone else has written a function `set_now_serving(int number)`, which will cause `number` to be displayed on the "Now Serving Number XX" display. Call this function as needed in your code.
- Your solution must not use any busy-waiting.
- You do not need to worry about ticket numbers overflowing.
- You can assume that customers are always available when their number comes up (i.e., you do not need to implement a mechanism to skip over non-responsive customers).
- You just have to implement the four functions described above. Assume the simulation software calls your functions when appropriate. The only functions your code should be calling is the lock and condition variable functions shown above and the `set_now_serving()` function.
Use the space below to write a declaration for `struct chili` and the four functions described above. You do not need to write any other code for the customers or servers.

```c
struct chili {
    int next_ticket;
    int now_serving;
    struct lock lock;
    struct condition server_available;
    struct condition customer_available;
};

void chili_init(struct chili *chile) {
    chile->next_ticket = 1;
    chile->now_serving = 0;
    display_now_serving(0);
    lock_init(&chile->lock);
    cond_init(&chile->server_available);
    cond_init(&chile->customer_available);
}

int chili_get_ticket(struct chili *chili) {
    int result;
    lock_acquire(&chili->lock);
    result = chili->next_ticket;
    chili->next_ticket++;
    lock_release(&chili->lock);
    return result;
}

int chili_wait_turn(struct chili *chili, int number) {
    lock_acquire(&chili->lock);
    while (chili->now_serving < number) {
        cond_wait(&chili->server_available, &chili->lock);
    }
    cond_signal(&chili->customer_available, &chili->lock);
    lock_release(&chili->lock);
}

int chili_wait_for_customer(struct chili *chili) {
    lock_acquire(&chili->lock);
    chili->now_serving++;
    display_now_serving(chili->now_serving);
    cond_broadcast(&chili->server_available, &chili->lock);
    cond_wait(&chili->customer_available, &chili->lock);
    lock_release(&chili->lock);
}
3. Threads

Suppose two threads execute the following C code concurrently, accessing shared variables a, b, and c:

Initialization
int a = 4;
int b = 0;
int c = 0;

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; 0 ) {</td>
<td>b = 10;</td>
</tr>
<tr>
<td>c = b - a;</td>
<td>a = -3;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>else {</td>
<td></td>
</tr>
<tr>
<td>c = b + a;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

What are the possible values for c after both threads complete? You can assume that reads and writes of the variables are atomic, and that the order of statements within each thread is preserved in the code generated by the C compiler.

Answer: 4, 7, 13, 14, -3
4. Synchronization

What is the major problem with the following program skeleton? (Ignore the fact that it does not check return codes from the pthread functions.) Suggest a way to fix this code.

```c
pthread_mutex_t mutexA, mutexB; /* mutexes protecting 2 resources */

void *func1(void *) {
    pthread_mutex_lock(&mutexA); /* req resource A */
    /* do something */
    pthread_mutex_lock(&mutexB); /* req resource B */
    /* critical section code */
    pthread_mutex_unlock(&mutexB);
    /* do something */
    pthread_mutex_unlock(&mutexA);
}

void *func2(void *) {
    pthread_mutex_lock(&mutexB); /* req resource B */
    /* do something */
    pthread_mutex_lock(&mutexA); /* req resource A */
    /* critical section code */
    pthread_mutex_unlock(&mutexA);
    /* do something */
    pthread_mutex_unlock(&mutexB);
}

int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&mutexA, NULL);
    pthread_mutex_init(&mutexB, NULL);
    pthread_create(&thread1, NULL, func1, NULL);
    pthread_create(&thread2, NULL, func2, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread1, NULL);
}

Problem with code (no more than one or two sentences):

**Answer:** Deadlock.

Solution (no more than one or two sentences):

**Answer:** Swap the ordering of the locks in func2(). First lock mutexA, then mutexB, then release mutexB, then release mutexB (in the same order as func1()).
5. Synchronization

You’re given the following code for three threads that will be run simultaneously. To make the code compute the correct answer, the programmer has added checks to make sure that the first, next, third, and last calls happen in that order. However, the spinning solution given is extremely inefficient. Use (only) the mutexes and condition variables provided to ensure that the proper ordering is enforced, and that the threads sleep when they can't make progress. Assume that the locks and CVs are properly initialized, and feel free to use abbreviations like lock(), signal(), etc instead of the full pthread function names.

```c
int first_done = 0;
int second_done = 0;
int third_done = 0;
pthread_mutex_t m1;
pthread_cond_t cv1;
pthread_mutex_t m2;
pthread_cond_t cv2;
pthread_mutex_t m3;
pthread_cond_t cv3;

void thread_1() {
    do_first_thing();
    lock(&m1);
    first_done = 1;
    signal(&cv1);
    unlock(&m1);
    lock(&m3);
    while (!third_done) {
        wait(&cv3, &m3);
    }
    unlock(&m3);
    do_last_thing();
}

void thread_2() {
    lock(&m1);
    while (!first_done) {
        wait(&cv1, &m1);
    }
    unlock(&m1);
    do_next_thing();
    lock(&m2);
    second_done = 1;
    signal(&cv2);
    unlock(&m2);
}

void thread_3() {
    lock(&m2);
    while (!second_done) {
        wait(&cv2, &m2);
    }
    unlock(&m2);
    do_third_thing();
    lock(&m3);
    third_done = 1;
    signal(&cv3);
    unlock(&m3);
}

Note: You can use the same lock and CV for all three threads as long as you use broadcast() instead of signal().
```
6. Deadlock

Consider the following snapshot of a system:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Max Allocation</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C D</td>
<td>A B C D</td>
<td>A B C D</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0 1</td>
<td>4 2 1 2</td>
<td>3 3 2 1</td>
</tr>
<tr>
<td>P2</td>
<td>3 1 2 1</td>
<td>5 2 5 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 0 3</td>
<td>2 3 1 6</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>1 3 1 2</td>
<td>1 4 2 4</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>1 4 3 2</td>
<td>3 6 6 5</td>
<td></td>
</tr>
</tbody>
</table>

Part A: Show that the system is in a safe state by demonstrating an order in which the processes may complete.

**One possible order: P1, P4, P2, P3, P5**

Part B: If a request from process P1 arrives for (1, 1, 0, 0) can the request be granted immediately? If it can, so the system is in a safe state by demonstrating an order in which the processes my complete.

**Yes. One possible order is P1, P4, P2, P3, P5**

Part C: If a request from process P2 arrives for (0, 0, 2, 0) can the request be granted immediately? If it can, so the system is in a safe state by demonstrating an order in which the processes my complete.

**No. The request doesn't leave any of resource C available, and all processes need at least one more C to complete.**

Part D: What is the optimistic assumption made in the Banker’s Algorithm? How can this assumption be violated.

**That the process will not ever need more that MAX resources to complete in a reasonable time. The process could need more resources and block on them, possibly causing deadlock.**