

Started on Wednesday, 29 November 2023, 20:06

State Finished

Completed on Tuesday, 12 December 2023, 13:08

Time taken 12 days 17 hours

Question 1

Complete

Marked out of 5.00

Suppose, in an otherwise working implementation of **clab**, **queue_new** were defined as follows

```
/* @return The new queue, or NULL if memory allocation failed */
queue_t *queue_new(void) {
    return malloc(sizeof(queue_t));
}
```

This compiles, and satisfies the requirement stated in the comment.

However, this **queue_new** is incorrect. Explain why.

Hint: What did we neglect to do? How is that a problem? How do we fix that?

Hint: Running **qtest** on **trace.cmd** consisting of

```
new
ih hello_world
reverse
rh world
```

produces the following output.

```
$ ./qtest -v 2 -f trace.cmd
cmd> new
cmd> ih hello_world
cmd> reverse
[1] 6015 segmentation fault (core dumped) ./qtest -v 2 -f trace.cmd
```

The code snippet correctly does as the description says, and returns a pointer to the queue or NULL if memory allocation failed. What is missing from this code is the initializing of the pointers that point to list-elements at the head (and possibly tail). These pointers should be initialized to NULL for a new queue.

During the insertion of an item, it is not a problem, as the non-initialized head pointer is simply assigned as 'next' for the new item at the head. The problem appears during reversal, as this will require us to sequentially go through the items in the queue and reverse their pointers. When we try to access the non-initialized pointer saved as 'next' for the element at the head of the queue, we will get a segmentation fault as the empty pointer was only re-assigned but never initialized.

Question 2

Complete

Marked out of 5.00

Describe your implementation of `queue_free`.

Hint: explain each step, and why it is needed.

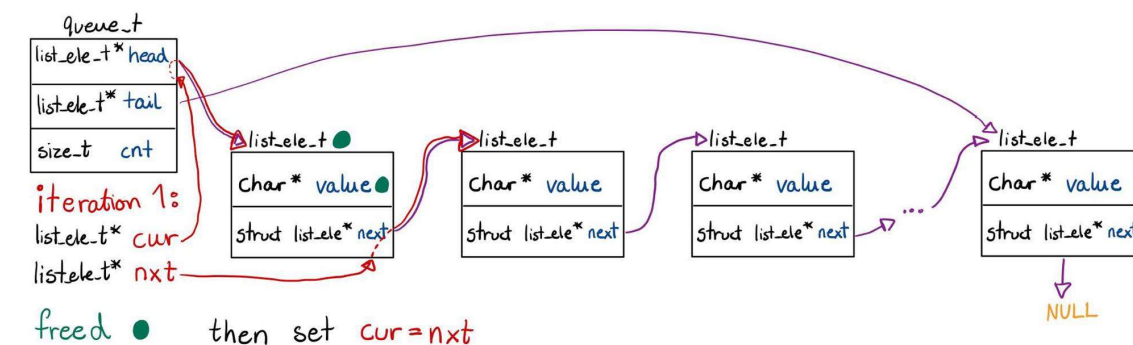
My implementation of `queue_free` contains 3 main steps;

(1) checking whether the passed `queue_t` pointer is a NULL pointer

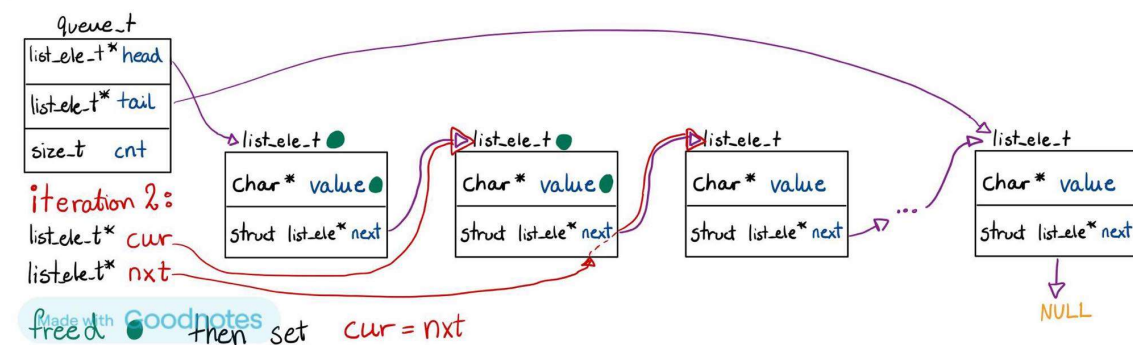
I return NULL when the pointer passed is a NULL-pointer. When the queue-pointer passed is NULL there will be no queue-object to free.

(2) Freeing up space used by list elements

Secondly, I check whether the queue is empty; If it is, I will go directly to step 3. Otherwise, I will free list elements sequentially. To free list elements I keep track of a pointer to the current list element and the next list element. I free the allocated space for current and current's value while iterating over the list. I have drawn up the two first iterations of the loop freeing the list elements;



The while-loop will terminate once the current pointer has become a NULL pointer, indicating having iterated through the entire list.



(3) Freeing up the space allocated for the queue t object.

Lastly, I free the space used by the queue structure itself (i.e. head pointer, count and tail pointer).

Question 3

Complete

Marked out of 5.00

Describe your implementation of `rotate_t`.

Hint: show your implementation, say what each part does, then explain how your implementation came to be: which program optimizations yielded speedups, which did not, and why? how did you split the whole task into subproblems? what do the `pthread` function calls do?

The attached code is my implementation of `rotate_t`.

Variables and Struct

Variables that will be used by all threads are declared as global variables (`src`, `dst`, `dimension`), while the thread-specific input variables are contained in a struct `"thread_rotate_input"` containing integers `begin` & `end`.

`void *thread_rotate(void *arg)`

This is the code executed by each thread, and is passed a `"thread_rotate_input"` object as a void pointer. The input parameters `"begin"` and `"end"` are the delimiters of the outer for-loop iterating over items row-wise for the destination. Having the row-wise iterations in the outer loop optimizes spacial locality for the writes to `dst`.

I Implemented loop unrolling, which reduces the number of times the computer runs the comparisons and jumps used to determine when the loop has executed the desired number of times. I also implemented sharing of common sub-expressions in the inner loop to decrease the number of function calls to `RIDX()` and reduce expensive repeated calculations, as multiplications are performance expensive and function calls are known optimization blockers.

At the end of the function we make sure to run `"pthread_exit(NULL)"` such that the thread will terminate at the end of the function execution.

`void rotate_t(int dim, pixel *src, pixel *dst)`

If the dimension is less than 256, I will run the regular `"rotate"` without threading, as initializing the threads creates an overhead that will only be 'paid-back' in performance increase for larger images.

If the image is bigger than 256x256, we will create 4 threads each getting chunks of the same size. I initialize 2 arrays of size 4; one containing the `"thread_rotate_input"` objects, one containing the `pthread_t` objects and set the global variables of `src`, `dst` and `dim`. Lastly I use two for-loops, one creating the threads and one joining them together.

 [_rotate_t.c](#)

Question 4

Complete

Marked out of 5.00

Would vectorization benefit `rotate_t`? How, or why not?

Would multithreading benefit `blend_v`? How, or why not?

Hint: What is it about the problem that makes this (not) beneficial? For a "why not", give a solid argument. For a "how", demonstrate the benefit.

Vectorization of `rotate_t`

Vectorization is beneficial in cases where we are doing calculations on several numbers that could be executed in parallel with the same operands. When we are rotating an image, we are simply moving a pixel-value from a specific source location to a specific destination location without modifying or doing calculations. We further access the source locations column-wise which is not aligned next to each other in memory.

In summary, vectorization would not benefit `rotate_t` as src accesses are not contiguous in memory, and would only require additional overhead of space in RAM with no speedup to gain; simply overcomplicating the implementation.

Multithreading of `blend_v`

Multithreading is beneficial when we can divide our task into disjoint subparts that can be executed concurrently. In the case of `blend_v`, this would be beneficial for large enough images, as creating and starting up a thread takes some time and would therefore not give a performance increase for small images that are not too calculation-intensive to begin with.

In `blend_v` we are sequentially doing the same calculations to 4 pixels at a time (using vectorization), yet for large enough images we could divide this into sub-images taking a number of rows each. Each thread can concurrently make the calculations to each sub-image without disturbing each other before exiting and returning to main. Finding the threshold for when it would give a performance-increase could be done empirically by examining the performance difference for larger and larger images. The benefit has been demonstrated by the performance increase between `rotate` and `rotate_t` with multithreading.

Question 5

Complete

Marked out of 5.00

Describe your solution to Part I (binary bomb) Phase 3.

Hint: what input defuses this phase? how did you come by this input: where is it stored? why does it have this form? what do the compare-instructions compare? explain your process (objdump, gdb, etc.)

I worked with the binary bomb using gdb to walk through the assembly step-by-step. The final input that defused phase_3 of the bomb was the following:

3 w 686

When trying to find out the correct input with gdb, I set a breakpoint on phase_3 and disassembled. Examining the assembly, I noted down 11 places where phase_3 has a function call to explode_bomb. Stepping through the code in the function I sequentially build requirements to the input on the basis of whether a compare will result in jumping to an explode_bomb OR not jumping away from an impending explode_bomb.

Requirements:

The first place a compare happens is at <phase_3+51>. This compares the number 2 to the number of space-separated elements in the input (saved in rax after the function call in <phase_3+46>). It is a 'less-than-or-equal' jump that we do not want to happen, meaning we need *3 elements or more*.

The next statement compares the number 7 to the first item in our list. It is an 'above' jump that we do not want to happen, meaning that our *first item needs to be a number less than or equal to 7*. The first number further has an impact on where the notrack jmp at <phase_3+85> jumps to. I chose the number 3, which results in a jump to <phase_3+194>.

The next compare, following the jump, compares the third item in our list to \$0x2ae. If they are exactly equal we will jump, which we want as the following line of assembly is an explode_bomb function call. This means that *the third item must be the hexadecimal 2ae, which is 686 in decimal*.

The last comparison is between the second item in our list and %al which is the low-byte from rax. A few instructions previously, 0x77 was moved into rax. We want these two items to be equal, as the jump will result in an explode_bomb function call if they are not equal. Thus, I enter *'w' as the second item, which has the hex-code under ASCII of 0x77*.

In summary, the requirements to the input were:

1. 3 or more space-separated items.
2. First item less than 7 (chose 3).
3. Third item equal to hexadecimal 0x2ae / decimal 686.
4. Second item equal to hexadecimal 0x77 (ASCII lower-case 'w')

Question 6

Complete

Marked out of 5.00

Describe your attack in Part II (buffer overflow) Phase 4.

Hint: what is your payload? what is its overall structure? what do you wish to accomplish? how does your payload accomplish this?

When doing a buffer overflow attack we are trying to exploit the fact that a non-protected reading mechanism might overwrite return-addresses on the stack when exceeding the size of the buffer.

The goal of phase 4 is to trigger the function Touch2, which requires a cookie as input parameter, using return-oriented programming. Return-oriented programming is a way of crafting the payload such that it utilizes the binary encoding of instructions residing in executable memory as opposed to executing instructions directly from the stack.

My payload for phase 4 of buffer overflow looks like so:

```
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
00 00 00 00 00 00 00 00 /* BUFFER */
1a 27 40 00 00 00 00 00 /* popq %rax gadget address */
a9 29 b7 28 00 00 00 00 /* cookie */
fa 26 40 00 00 00 00 00 /* movq %rax,%rdi gadget address */
42 25 40 00 00 00 00 00 /* touch2 address */
```

The first 56 bytes of my payload fills the buffer with all 0s. I found the buffer size by finding the number of bytes which the stack pointer was decreased by prior to taking input into the buffer. Touch2 needs a cookie as it's input, and for this to happen I will need to find some instructions in the program to move the 8-byte cookie into the %rdi register.

The instructions (gadgets) for the payload can be located anywhere in the executable binary code, but it needs to follow the structure **<instruction> c3**, for which <instruction> is the binary encoding of some instruction sequence. Binary c3 is the return instruction, which goes to the return-address pointed to by the stack. By sequentially stacking addresses of gadgets, one can create a program from instructions located in executable memory.

I find 2 useful gadgets:

- *popq %rax*
- *movq %rax,%rdi*

I use the gadgets, the cookie and the touch2-address in the buffer overflow, resulting in:

1. Instruction pointer goes to return-address which has been overwritten with the address of the gadget *popq %rax*.
2. *popq* pops the cookie, and places it in %rax.
3. Instruction pointer goes to the return-address which points to the gadget *movq %rax,%rdi*.
4. Content of %rax moved into %rdi. Now the cookie is in the correct register.
5. Instruction pointer goes to the return-address which points to Touch2 function.
6. Touch2 is executed.

Question 7

Complete

Marked out of 5.00

Describe your implementation of multithreading.

Hint: how do you ensure that the proxy is always ready to receive requests? which part of the request-processing do you hand off to a thread?

I implement multithreading using the pthread library. I chose to pass on the request handling from right after accepting the client request. This means, that each thread will be responsible for handling connecting to the server and passing on the content to the client. The main changes applied to the initial working proxy code is for the following functions:

handle_connection_request

This is the function that is called in main in the infinite while loop. Given that the client_fd is non-zero, a thread will be created for handling the connection and passing of content between server and client. The thread function receives the client_fd as input through a void pointer.

Once the thread has been created using '*pthread_create*' we detach the thread using '*pthread_detach*'. When we detach a thread, we hand off the workload to a thread without pausing execution of our function while waiting for the thread to finish at the point of '*pthread_join*' in main. This ensures that we are ready to receive the next request while the thread handles the last connection request.

handle_request_thread

This is the code executed by each thread upon creation. Runs the function 'handle_request' which connects to the server and passes the content. At last, it closes the client connection and exits its execution. The 'handle_request' function has not been modified for the task of multithreading.

Question 8

Complete

Marked out of 5.00

Describe your implementation of caching.

Hint: what data structure do you use to represent the cache? how do you prevent race conditions?

We implement caching for our proxy, as fetching content from memory is more efficient than connecting with a server and requesting content to be passed on to the client.

How is the Cache implemented?

The below described struct and helper-functions are used to implement caching in the function 'handle_request'. Before creating a connection to a server, it first checks whether the domain_name exists in the cache and writes to the client from cache if it is found. When an item is found in cache and passed on to the client, the importance score is reset to `__INT_MAX__` and all other cache entry importances are decreased.

Otherwise, the server connection is created and client writes are conducted as usual. After this process, the content and domain_name is written to cache using the `write_cache_item()` function. I chose an eviction policy based on an importance measure in terms of 'last time used', i.e. the cache contains the 10 most recently requested items at any point.

Preventing Race Conditions:

The mutex locks are used to prevent race conditions, by ensuring that we do not have a thread writing to a cache_entry while it is being read, or several threads writing to the same cache_entry.

I chose to implement mutex locks separately for reading and writing and on each individual cache_entry. I chose this approach as it allows for several readers of the same cache_entry at once, whilst blocking for writing modifications. I also chose to create the locks for the individual cache entries, as many concurrent reads / writes from threads would otherwise likely be "in the way" of each other, even if they were interested in different cache entries.

Struct and helper-functions

In order to handle caching, I define a struct that contains variables to store content and domain name for a cache entry. The struct contains the following:

- domain_name: A char-array of length 'MAX_LINE'
- content: A char-array of length 'MAX_OBJECT_SIZE'
- cur_reader_count: the number of threads currently reading the entry.
- is_empty: Indicator for whether the cache entry is empty
- importance_score: An integer score, higher scores indicate recent usage.
- lock_w: locks the cache for writing, ensuring no one can read or write simultaneously.
- lock_r: locks the cache for reading, ensuring no one can write while the cache is being read.

Further I use a number of helper-functions to define behavior for certain operations:

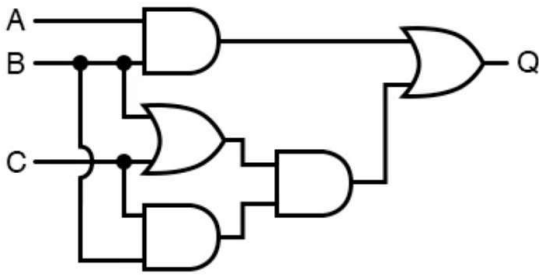
- `init_cache()`: returns a cache that has been initiated with maximum number of items as an array of cache_entry objects. This initializes all cache_entry variables as well.
- `readers_lock(cache_entry*)`: locks for reading, if it is the first reader it will lock for writing as well, then increase the reader count and unlock the read-lock.
- `readers_unlock(cache_entry*)`: locks for reading, decreases the reader count and unlocks the write lock if no more readers are left, then unlocks from reading.
- `writers_lock(cache_entry*)`: locks write-lock
- `writers_unlock(cache_entry*)`: unlocks write-lock
- `search_cache(char*)`: searches through cache for the domain_name sequentially.
- `find_cache_to_write()`: searches the cache and returns either (1) index of an empty cache or otherwise (2) index of the cache with lowest importance score.
- `decrease_importance(int)`: decreases the importance_score by 1 for each cache_entry except the index passed as input.
- `write_cache_item(char*, char*)`: finds a cache to write to and locks it for writing. Uses 'strncpy' to copy domain_name and content into the cache_entry from passed pointers, sets is_empty to 0 and importance_score to `__INT_MAX__`. Unlocks from writing and decreases importance score of all other entries.

Question 9

Complete

Marked out of 3.00

Here is a logic diagram.



Which of the following Boolean expression are equivalent with Q (the output of the circuit)?

Pick all that apply.

- ☒ $A \& B \mid B \& C \& B \& C$
- ☒ $A \& B \mid B \& C \& (B \mid C)$
- ☒ $B \& (A \mid C)$
- ☐ $A \& B \& C$

Question 10

Complete

Marked out of 3.00

Which of the following are entries in the truth table for this circuit?

Pick all that apply.

- ☐ $A=1, B=1, C=0, Q=0$
- ☒ $A=1, B=0, C=1, Q=0$
- ☒ $A=1, B=1, C=0, Q=1$
- ☒ $A=0, B=1, C=1, Q=1$

Question 11

Complete

Marked out of 4.00

Here is a short program written in Y86-64

```
irmovq $D, %rdi      # instruction nr. 0
irmovq $S, %rsi      # instruction nr. 1
irmovq two, %rax      # instruction nr. 2
mrmovq 0(%rax), %rax  # instruction nr. 3
main:
  addq %rdi, %rax      # instruction nr. 4
  addq %rax, %rax      # instruction nr. 5
  subq %rdi, %rsi      # instruction nr. 6
  je done              # instruction nr. 7
  jmp main              # instruction nr. 8
done:
  irmovq $0, %rdi      # instruction nr. 9
  halt                 # instruction nr. A
two:
  .quad 0x000000000002
```


Indicate, for each of the following values in place of D and S, the value of register `%rax` after program execution. Assume that the program is running on SEQ hardware.

- D = 1, S = 3. After program execution, `%rax` =
- D = 2, S = 6. After program execution, `%rax` =
- D = 2, S = 3. After program execution, `%rax` =

Question 12


























Complete

Marked out of 5.00

Indicate, for each of the following clock cycles during program execution, which program instruction is in which stage of the pipeline (i.e. in which of the F, D, E, M, W pipeline registers). In each field, indicate the instruction number of the instruction there, or  for pipeline bubble.

Assume $D = 1$, $S = 3$, and that the program is running on PIPE hardware (incl. forwarding- and pipeline-control-logic).

Hint: We have pre-filled the stages for the first four cycles.

	F	D	E	M	W
1.	0				
2.	1	0			
3.	2	1	0		
4.	3	2	1	0	
5.		3	2	1	0
6.	4		3	2	1
7.		4		3	2
8.			4		3
9.	5			4	
10.	6	5			4
11.		6	5		
12.			6	5	
13.	7			6	5
14.		7			6
15.			7		

Question 13

Complete

Marked out of 3.00

Consider naive_rotate from prflab:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
void naive_rotate(int dim, pixel *src, pixel *dst)
{
    int i, j;
    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
}
```

What kind of locality of data does this program exhibit whilst referencing following?

- elements of `src`
- elements of `dst`
- `i`

Question 14

Complete

Marked out of 3.00

What kind of locality of instructions does this program exhibit whilst referencing the following?

- `i=0` instruction
- `j++` instruction
- inner loop body

Question 15

Complete

Marked out of 3.00

Why does thrashing happen? Pick all that apply.

- ☒ High miss rate
- ☒ Capacity miss
- ☒ Cache miss
- ☐ Cache incoherence

Question 16

Complete

Marked out of 3.00

A cache coherence protocol ensures that...

Pick one that applies.

- ☐ ... data in the cache can be accessed in a coherent way.
- ☒ ... contents of the cache is consistent with what is contained in memory.
- ☐ ... a read from an address gives you its most up-to-date value.

Question 17

Complete

Marked out of 3.00

What is a segmentation fault?

Pick one that applies.

- ☐ The data structures in the heap of the process are no longer aligned in memory.
- ☐ The kernel failed to segment memory into pages in response to a request made by the process.
- ☒ The process attempted to access a region of memory which it is not privileged to access.

Question 18

Complete

Marked out of 3.00

What do we gain by having a virtual memory mapped region for shared libraries?

Pick all that apply.

- ☐ Backwards compatibility (it's a legacy feature)
- ☒ Programs will run faster.
- ☐ Simplifies access to shared libraries.
- ☒ Programs will consume less memory.

Question 19

Complete

Marked out of 3.00

When a program reads from a file on disk, which of the following events occurs as a result?

Pick all that apply.

- ☒ The disk interrupts the CPU when the transfer is complete.
- ☐ The disk transfers data to a buffer in the CPU.
- ☐ The program executes the IN instruction (to input from I/O port).
- ☐ The program executes the INT (aka. SYSCALL) instruction.

Question 20

Complete

Marked out of 3.00

Why is it important to close file descriptors once you are done using them?

Pick one that applies.

- ☐ The entity at the other end will not know that the session is complete otherwise.
- ☒ Not doing so introduces a memory leak.
- ☐ Otherwise data may still reside in buffers which haven't been flushed.

Question 21

Complete

Marked out of 3.00

Consider the following C source file `t.c`.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int n = 3;
int k = 0;
int s = 0;

void *incr(void* arg) {
    int tk = ++k;
    int ts = s;
    int i;
    for (i = 0; i < n; i++) {
        ts += tk;
    }
    usleep( ( (float)rand() / RAND_MAX ) * 10 );
    s = ts;
}

int main() {
    pthread_t* tids = malloc( n * sizeof(pthread_t) );
    int i;
    srand(time(NULL));
    for (i = 0; i < n; i++) {
        pthread_create( &(tids[i]), NULL, incr, (void*)NULL );
    }
    for (i = 0; i < n; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("%d\n", s);
}
```

It compiles without error with the following command.

```
gcc -pthread -o t t.c
```

Upon executing this program (`./t`), which of the following are possible outputs of the program?

Pick all that apply.

- ☐ 30
- ☒ 9
- ☒ 3
- ☒ 15
- ☒ 12

Question 22

Complete

Marked out of 3.00

Which of the following variables would need to be protected (e.g. by a mutex) to prevent race conditions?

Pick all that apply.

- ☒ s
- ☐ n
- ☒ k
- ☐ the mutex itself
- ☐ tk

Question 23

Complete

Marked out of 3.00

Why does an optimizing C compiler not do code motion on function calls?

Pick one that applies.

- ☐ Statements in a function body cannot substitute a function call expression.
- ☒ Not all functions are referentially transparent.
- ☐ It is too difficult to detect function calls in source code.

Question 24

Complete

Marked out of 3.00

Suppose process P1 allocates a region on the heap using `malloc`, and gets back an address `0x00000020`. Process P1 then sets the contents of the region to all 1s. Suppose that process P2 then attempts to read from `0x00000020`. What happens?

Pick all plausible outcomes.

- ☒ P2 reads in the contents of the physical page that `0x00000020` maps to in its own address space.
- ☐ P2 reads in the value `0x00000020` had before P1 overwrote it with 1s (P2 does not have access to P1's address space).
- ☒ P2 experiences a segmentation fault.
- ☐ P2 reads in the the all 1s that P1 wrote to `0x00000020`.

Question 25

Complete

Marked out of 3.00

On mainstream computer systems, when one process is currently running, how do other processes get to run?

- ☐ The kernel process suspends the running process, and decides which process goes next.
- ☒ An interrupt forces the processor to context-switch to the kernel at regular intervals.
- ☐ The running process relinquishes control of the processor core once it is ready to do so.

Question 26

Complete

Marked out of 3.00

How can a program, running on a core, ensure that an otherwise non-atomic instruction gets executed atomically?

Pick one that applies.

- ☐ By pausing execution of all other cores.
- ☐ By asking the kernel to perform the instruction on its behalf.
- ☒ By utilizing a special instruction modifier that locks the memory bus.

Question 27

Complete

Marked out of 3.00

What is the benefit of the "Everything is a file" idea implemented in Unix systems?

Pick one that applies.

- ☐ Facilitates adding, modifying, and deleting, devices on your system
- ☒ A unified API for doing I/O on all kinds of resources.
- ☐ Can easily back up and restore all kinds of data.