Assignment 4 - Free Response Questions

Graded

4 Days, 23 Hours Late

Student

Christian Blake

Total Points

44.6 / 60 pts

Question 1

Q1 — **3.6** / 12 pts

+ **12 pts** Excellent

+ **11 pts** Good

+ **10 pts** Mostly Right

✔ + **8.5 pts** Right Track

+ **6 pts** Valiant Effort

+ **0 pts** No Answer

– **4.9 pts** You did not account for the TLB access time with the hit rate and you didn't account for the 2 levels of the page table

Late -10%

Question 2

Q2 — **6** / 12 pts

+ **12 pts** Excellent

+ **11 pts** Good

+ **10 pts** Mostly Right

+ **8.5 pts** Right Track

✔ + **6 pts** Valiant Effort

+ **0 pts** No Answer

See the key on Canvas for the correct answer.

Question 3

Q3 **12** / 12 pts

✔ **+ 12 pts** Excellent

+ 11 pts Good

+ 10 pts Mostly Right

+ 8.5 pts Right Track

+ 6 pts Valiant Effort

+ 0 pts No Answer

Question 4

Q4 **11** / 12 pts

+ 12 pts Excellent

✔ **+ 11 pts** Good

+ 10 pts Mostly Right

+ 8.5 pts Right Track

+ 6 pts Valiant Effort

+ 0 pts No Answer

With this approach, you would need to signal the readvocab semaphore twice as you call down() on it twice

Question 5

Q5 **12** / 12 pts

✔ **+ 12 pts** Excellent

+ 11 pts Good

+ 10 pts Mostly Right

+ 8.5 pts Right Track

+ 6 pts Valiant Effort

+ 0 pts No Answer

Question assigned to the following page: 1

CS 480 OPERATING SYSTEMS

Assignment 04

Part I Free Response Questions (60 points)

Due: Beginning of the Class, **Nov 16th**

Late Due: Beginning of the Class, **Nov 21st**

You must work on this part on your own.

5 Questions – Each has 12 points.

Make sure to include the **single examinee affidavit** below at the beginning of your answer sheet. You must work on your own for this part. It would be a red flag if we find submissions from two students are with same incorrect answers in multiple parts. Also as specifically mentioned in the Syllabus, asking questions, and getting answers from online sources are considered plagiarism.

SINGLE EXAMINEE AFFIDAVIT

"I, the undersigned, promise that this assignment submission is my own work. I recognize that should this not be the case; I will be subject to plagiarism penalties as outlined in the course syllabus."

Student Name: ____Christian Blake_____ (Print your name

here as signature) RED ID: _____824904815_____

Date: _____11/21/2023_____

1. A virtual memory system uses a two-level page table. Suppose each physical memory access takes 12 ns, a TLB access requires 3 ns. To ensure the effective memory access time to be within 17 ns, what would be the minimal TLB hit rate? Assume the multi-level page table walk has negligible overhead other than the memory access times.

17 =(x * 12 ns) + (1- x)(3ns + 24)
17 = 12x + 3 +24 -3x -24x
10 = 15x
X = 10/15 = ⅔ or 66.67%

2. A 1-level page table is used for virtual paging for a 32-bit virtual address space that is divided into 16 pages (i.e., with the page size as 2^28).

Question assigned to the following page: 2

Page # 0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0XB
➔0xC 0xD
0xE
0xF

Frame #	Valid	Referenced	M
0x7	1	0	
0x2	1	1	
	0	0	

	0	0	0	1
0x4	1	0	1	1
	0	0	0	0
0x9	1	0	0	0
	0	0	0	0
0xD	1	1	0	0
0x8	1	1	0	0
0x1	1	1	1	0
	0	0	0	0
0xA	1	1	0	0
0x3	1	0	1	1
	0	0	0	0
	0	0	0	0

Use WSClock page replacement algorithm to select a victim frame and report the results.

Imagine page entries (rows) are arranged in clock positions, with the clock hand starting from **virtual page 0xC**, walking the clock by moving down in the table one page at a time. After reaching the end of the table, the clock hand moves to page 0x0 (the beginning of the table) and moves down from there.
The last virtual time each page is referenced is put in the last column (note this column is not part of the page table). A daemon periodically copies referenced bit, modified bit, and virtual time to a Clock data structure for the WSClock algorithm to use.

Assuming the current virtual time is **1700**, and the age threshold in virtual time unit for the WSClock algorithm is **300**.

Simulate the algorithm:

1) Show details of the algorithm walking through entries until selecting the victim frame, you must indicate the action taken for each page entry (**use the following table**) as a result from the execution of the algorithm:
    a. update to the page entry
    b. any page write that was scheduled.
    c. reason why the action was taken.
    d. put no action if no action was taken.

Questions assigned to the following page: 2 and 3

e. **ignore the invalid** page table entries (i.e., entries with valid flag as 0)

2) **Report the victim frame** selected by the algorithm and indicate **why**.

clock hand at a page	Action	
0xC	Referenced bit cleared, age above threshold, page recently accessed 1410>1400	
0xD	Modified, write scheduled. Selected as victim frame	
0x0	Referenced bit cleared, page below threshold	
0x1	Page write scheduled, recently modified schedule write and clear referenced(it's already cleared though)	
0x4	Write scheduled, Not referenced, modified , and the age is older than 1400	
…		
…		

3. Can the following **entry** and **exit** (pseudocode) subroutines be used for implementing a **formal critical section** (i.e., mutual exclusion, progress, bounded waiting properties)? **Justify** your answer.

You can assume the entry(..) and exit(..) would be executed atomically. int *lock is a shared variable that is initialized to zero (unlocked).

```
int *lock = 0; // shared variable among processes or threads

entry(int *lock) {
 locked = test-and-set-lock(lock);
 if (locked) {
     //append the process to the waiting process list
     add(processId, waitingProcesses);
     set state to blocked;

   }

}
```

Questions assigned to the following page: 4 and 3

```
exit(int *lock) {
 *lock = 0;
```



```
  //select a process with the shortest job from the waiting processes list
  nextProcess = selectProcessWithShortestJobFrom(waitingProcesses);
  set state of nextProcess to ready;

}
```

// example code for using the above implementation for a critical section

entry (lock)

access to shared data //critical region

exit(lock)

Answer:
No. The Pseudocode has mutual exclusion and is seen by the test-and-set lock. It's atomic and sets the lock to 1, but if the previous state was 1, the thread goes to the waiting list. Otherwise, it enters the critical section if the lock was 0. This works for mutual exclusion. Next, progress wise, the code doesn't focus on who get's to go next in a clear way. The next process is being picked by the running code not those waiting to execute. Lastly, bounded waiting is not properly implemented, because there is no way to make sure each process gets a turn, some processes could have long wait times, or never be executed.

4. Suppose the main thread starts 3 worker threads at the same time, they are readvocab_thread, readlines_thread, and countvocab_thread. Use **a semaphore** or **semaphores** to change the following **pseudo** code to:
   - ensure the **readlines** thread only starts to execute **after** the **completion** of the **readvocab** thread, and
   - ensure the **countvocab** thread only starts to execute **after** the **completion** of the **readvocab** and **readlines** threads.

Assume semaphores are shared variables among threads and be sure to initialize them to an appropriate value. **Note**:

a. reasonable pseudo code with sufficient **details** and **comments** is good,
b. use a conceptual semaphore with up and down operations as demonstrated in the class.

```
Readvocab semaphore = 0
Readlines semaphore = 0
countvocab_thread() {
      Down (Readvocab);
      Down (readlines);
```

Questions assigned to the following page: 4 and 5

```
   countvocab();
}

readvocab_thread (...) {


   readvocab();
   up(readvocab);
}

readlines_thread (...) {

       down(readvocab);
   readlines();
   up(readlines);
}
```


5. In a4 programming assignment, instead of using monitors for accomplishing the synchronizations between producers and consumers, **use semaphores** to **write pseudocode for** implementing the same synchronizations, **only** worry about:
   - the **consumer thread function and the producer thread function**. •

   note:
   - make sure the necessary semaphore details are specified:
     a. the initial values - initialize semaphores as shared variables outside the producer and consumer functions.
     b. use a conceptual semaphore with up and down operations as demonstrated in the class.
   - assume:
     a. request type for producing is from a shared variable requestType (either Bitcoin or Ethereum).
     b. use produce(requestType) for producing a crypto request and consume(requestType) for consuming a crypto request.
     c. for mutex access, use a semaphore to implement it.
     d. a broker queue (brokerQueue as the bounded buffer) is created and shared.
     e. **no** need to worry about production limit, synchronization for main thread, or printing.
   - reasonable pseudocode with sufficient **details** and **comments** is good: a. refer to the **semaphore** skeleton code for solving the **producer/consumer with bounded buffer** problem in TEXT and lecture slides.

Question assigned to the following page: 5

I accidentally created the code with semaphores already, here is completed code for producer and consumer that works with semaphores, thought this would be a good use to a dumb mistake:

```
while (true) {
        sem_wait(&shared->queueEmptySemaphore);
        //lock shared mutex before accessing
        pthread_mutex_lock(&shared->queueMutex);

        // Check if production is complete and the queue is
empty
        if (shared->productionComplete &&
shared->requestQueue.empty()) {
            //unlock before break
            pthread_mutex_unlock(&shared->queueMutex);
            break; // Exit the loop and end the thread
        }
        // Pop the request form queue
        TradeRequest request = shared->requestQueue.front();
        shared->requestQueue.pop();

        // Decrement Bitcoin request count if the consumed
request is a Bitcoin request
        if (request.type == Bitcoin) {
            sem_post(&shared->bitcoinRequestSemaphore); //
Signal space for Bitcoin request
            int sval;
            sem_getvalue(&shared->bitcoinRequestSemaphore,
&sval);
            std::cout << "Current semaphore value3: " << sval
<< std::endl;
        }

        pthread_mutex_unlock(&shared->queueMutex);
        sem_post(&shared->queueFullSemaphore); // Signal that
queue has space

        // Update consumed count for the type of request
consumed
        RequestType requestType = request.type;
        consumed[requestType]++;
        consumerArgs->consumed[requestType]++;
```

Question assigned to the following page: 5

```
        //Record request removal
        unsigned int inQueue = shared->requestQueue.size();
        report_request_removed(type, requestType, consumed,
&inQueue);


std::this_thread::sleep_for(std::chrono::milliseconds(sleepT
ime));
    }
```

```
while (true) {
        // For Bitcoin, wait on bitcoinRequestSemaphore
        if (type == Bitcoin) {
            sem_wait(&shared->bitcoinRequestSemaphore);
            int sval;
            sem_getvalue(&shared->bitcoinRequestSemaphore,
&sval);
            std::cout << "Current semaphore value1: " << sval
<< std::endl;
            sem_wait(&shared->queueFullSemaphore);
        }
        else{
            sem_wait(&shared->queueFullSemaphore);
        }

        pthread_mutex_lock(&shared->queueMutex);

        // Check if queue is full or if production limit
reached
        if (shared->totalRequestsProduced >=
producerArgs->productionLimit ||
            shared->requestQueue.size() >=
shared->maxQueueSize) {
            pthread_mutex_unlock(&shared->queueMutex);
            // For Bitcoin, release the
bitcoinRequestSemaphore as well
            if (type == Bitcoin) {
                sem_post(&shared->bitcoinRequestSemaphore);
                int sval;

sem_getvalue(&shared->bitcoinRequestSemaphore, &sval);
                std::cout << "Current semaphore value2: " <<
sval << std::endl;
```

Question assigned to the following page: 5

```
            }
            break;
        }
        // Create and add a new TradeRequest of given type
        TradeRequest request(type);
        shared->requestQueue.push(request);
        shared->totalRequestsProduced++;
        producerArgs->produced[type]++;

        sem_post(&shared->queueEmptySemaphore); // Signal
that queue is not empty

        pthread_mutex_unlock(&shared->queueMutex);
        // Sleep to simulate production time

std::this_thread::sleep_for(std::chrono::milliseconds(sleepT
ime));
    }
```