Major Exam

● Graded

Student

Saharsh Laud

Total Points

19.5 / 60 pts

Question 1

Q1 — **7** / 10 pts

1.1 A — **5** / 5 pts

✔ **+ 2.5 pts** *sigret* is responsible for invoking the kernel to restore the user thread's context

✔ **+ 2.5 pts** *ret* pops return address from the stack and jumps to it. It does not restore the context, hence if we replace *sigreturn* with *ret*, the execution resumes with corrupted state.

+ 0 pts Incorrect

1.2 B — **2** / 5 pts

+ 3 pts Stack overflow can overwrite return addresses, and when signal handler returns, it can open way for attacks.

+ 2 pts A different stack is used for handling signals to make sure that no program gets illicit access to the kernel as kernel level functions can be compromised.

✔ **+ 1 pt** Partial explanation of stack overflow

✔ **+ 1 pt** Partial explanation why different stack used

+ 0 pts incorrect

Question 2

Q2 **0** / 10 pts

2.1 **A** **0** / 4 pts

+ 1 pt Yes
secret() can be invoked

+ 3 pts Correct Explanation:
buffer[10] overwrites the return address of foo()

+ 0 pts Incorrect Explanation

✔ **+ 0 pts** Incorrect/Blank.

2.2 **B** **0** / 2 pts

+ 2 pts Correct.

+ 0.5 pts No, secret won't be invoked.

+ 1.5 pts Heap grows upwards, while stack grows downwards. There is a huge gap between stack and heap. Stack stores the return address of foo. Buffer overflow in heap won't overwrite the return address of foo in the stack. Thus secret won't be invoked when foo is returns back to main.

✔ **+ 0 pts** Incorrect.

+ 1 pt Exception: There is almost negligible probability secret being invoked when heap overwrite will update the return address of foo with start address of secret. It will happen only if stack and heap have grown very much, and their top are almost touching.

2.3 **C** **0** / 4 pts

+ 1 pt Identified stack overflow attack in the given context.

+ 3 pts Correct: Use of Canary Tokens

A random value stored between local variable (buffer) and the return address. If the buffer overflows, the random value is overwritten & detected later.

✔ **+ 0 pts** Incorrect/Blank

Question 3

Q3 **0** / 10 pts

✔ **+ 0 pts** Incorrect/Unattempted

+ 2 pts Top 32 bits are used to store timestamp

+ 2 pts `update()` along with placing the values also increments the timestamp

+ 4 pts `snapshot()` reads all timestamps twice, if unchanged it returns else it tries again

+ 2 pts Justification of global progress (At least one thread completes)

Question 4

Q4 **7** / 10 pts

4.1 **A** **3** / 3 pts

✔ **+ 1.5 pts** High overheads
Sending an IPI to all cores forces each core to context switch and handle the interrupt, increasing CPU usage and latency

✔ **+ 1.5 pts** Grace period
Determining when grace period ends require acknowledgement from all the other cores, which adds complexity and synchronization overhead, making the process expensive.

+ 0 pts wrong/Not Attempted/Unrelated

4.2 **B** **4** / 7 pts

+ 0 pts Click here to replace this description.

✔ **+ 2 pts** **Design**: Tree RCU organizes per-CPU quiescent state data into a tree structure.
Each leaf node corresponds to a CPU's rcu_data structure.
Internal nodes are rcu_node structures that aggregate quiescent state info from child nodes.

✔ **+ 2 pts** **Benefits:** Parallel Aggregation: Each internal node summarizes the state of its children. Only the root node needs to be checked to determine if all CPUs have entered the quiescent state.

This structure supports parallel updates and hierarchical checking, which avoids bottlenecks and makes the system highly scalable.

+ 5 pts Detection of the end of the Grace Period efficiently:
-> When a thread blocks (you know it is out of the read lock-unlock phase)
->Switches to user-mode execution
->Enters an idle loop
->Switch the context to run a kernel thread

+ 1 pt partial explanation of design

+ 1 pt Partial mention of benefits

+ 2 pts Partial mention of how grace period ends

Question 5

Q5 **3.5** / 10 pts

5.1 **A** **0.5** / 3 pts

+ 0 pts Correct

+ 2.5 pts Formulation correct, but some mistake in explaination, please refer to comment

+ 3 pts exact formulation Virtual_Address = vma.start_address+i<<12, and explained the parameters of the expression

✔ **+ 0.5 pts** If linear search explained

+ 0 pts Incorrect

+ 2.5 pts correct formulation with explaination but multiplying index by page size missing

+ 1.5 pts Partial concept explained, but formulation missing or parameters used in formulation not clear

+ 1 pt Some introductory concept

+ 1 pt Explaination is correct, but mistake in the formulation, please refer to the comment provided for this rubric item

+ 2 pts formulation written but explaination lacking

+ 2 pts explaination correct but mistake in formulation, please refer to comment to know more about the mistake

5.2 **B** **0** / 3 pts

+ 3 pts If forking ,CoW, many to one mapping between vmas to anon_vma and after child page modification, one to many mapping between vma to anon_vmas explained

+ 1.5 pts Partial concept explained

✔ **+ 0 pts** Incorrect

+ 1 pt Some basic concept

5.3 **C** **3** / 4 pts

+ 1 pt Cache-behaviour:
For smaller anon_vma_chain sizes, sorted array would perform better due to better cache locality. However, for larger datasets or ones with freq modifications, the cost of array resizing and shifting of elements negate benefits

+ 1.5 pts Concurrency:
Red black tree more suited for anon_vma_chain operations because they are frequent and occur under contended locks. Tree rotations and node updates are localized allowing concurrent operations on different part of tree with minimum contention

✔ **+ 1.5 pts** Efficiency
Red black tree more efficient for dynamic workloads with frequent insertions and modifications as seen in anon_vma_chain during page faults, fork,mem reclaimation

+ 0 pts Wrong/Unrelated

💬 **+ 1.5 pts** Point adjustment

Question 6

Q6 **2** / 10 pts

6.1 **A** **0** / 2 pts

+ 2 pts When there are no pages of oldest generation left.

✔ **+ 0 pts** Incorrect

6.2 **B** **0** / 2 pts

+ 2 pts When there is no temporal locality

✔ **+ 0 pts** Incorrect

6.3 **C** **2** / 2 pts

✔ **+ 2 pts** when there is access to large file then can evict older file pages

+ 0 pts Incorrect

6.4 **D** **0** / 4 pts

+ 4 pts Prefetching/read-ahead mechanisms: The kernel proactively loads adjacent file data into the page cache, triggering accessed bits even if the application doesn't use them. Correct

+ 2 pts Partially correct

✔ **+ 0 pts** Incorrect

Operating Systems End Term Exam

2024-25 Sem II

Name: SAHARSH LAUD

Entry number: 2024MCS2002

- [ ] COL 331
- [x] COL 633
- [ ] ELL 405
- [ ] ELL 783

Total: 60 marks

- Each question is for 10 marks.
- Prove and justify all your answers.

1. Answer the following questions:
   a. What happens if we replace the *sigreturn* system call with a *ret* instruction? [5]

If we replace the sigreturn system call with ret instruction then instead of the return call to fetch the process happening in user space there would be need for it to occur in kernel space which is less efficient and slower scheme. When a signal handler is invoked the current process is halted and its return

address is fetched back by sigreturn so that its Context can be loaded directly and process can resume from where it left off directly in user space without any need for kernel level interventions.
If ret is used then kernel level thread needs to handle and we need to switch to kernel mode ~~used~~ to switch back to the process & continue its execution.

b. What is the advantage of using a different/alternate signal stack? [5]
[Hint: Look at the next question (Question #2). There are security/correctness issues. Explain with examples and provide all the relevant details.]

The advantage of using alternate signal stack is that its a more secure method. Also we can stack up the signal calls in order of their priority and the highest priority signal can be popped from the top of stack.
We can also store bits for valid/ invalid signal in the stack which would lead to the correctness of the signal and corresponding signal handler. Nested signal handling can be easily implemented through this wherein we push the signal calls as they're called and then can start handling them.

2. Consider the following piece of code:

```
void secret() {
```

```
      ...
}
void foo(){
      int j, buffer[5];
      for (j=0; j < 5; j++) buffer[j] = 14;
      buffer[10] = (int) (&secret);    /* Line BUF */
}


int main(){
      foo();
}
```

a. Can the function *secret* get invoked because of the code in Line BUF? If yes, how? If not, why not? [4]

No, the function secret cannot be invoked because the line BUF is trying to store the address of the pointer to this function at a position that is beyond the allocated space for the buffer array → buffer[5] → ie. It can have 5 elements. But buffer[10] = int(&secret) is trying to store the value of integer of address at buffer[10]. Since its static memory allocation so size of array buffer cannot increase hence the call buffer[10] is invalid and the function secret cannot get invoked because of the code in line BUF.

b. How will the answer to part (a) get modified if *buffer* is stored on the heap? [2]

If stored on heap then Yes buffer[10] line buf can call secret function because heap memory allocation is dynamic so to have buffer[10] the size of the stack allocated memory in heap will be increased dynamically and buffer[10] = int (&secret) will be successful so line buf can access secretfunction.

c. C does not perform *"array bounds checking"*; however, other languages such as Java do it. The reason is that every array access should not be burdened with an additional check operation. What is an efficient way of performing this check without burdening every access with additional instructions? The issue is that the array indexes are out of range. Consider an array of size 10. The user may try to access entries that are in the range 11-15, which is erroneous. Assume that such bugs are limited to accessing array indexes that are *slightly* out of range. They are not *severely* out of range. Propose a solution to detect such out-of-range accesses. They need not be detected immediately. They can be detected slightly later also. [4]

so what we can do is donot do any checks during array access just set a flag to a value and then later check that flags value and if its not as per expectation then we detect that its an out of bound/range access.
Since in given example size of array is 10. We will need to use signed integers and do signed integer arithmetic.
we can do modulo size we can do without mod as well.

→ Flag = (array_size - index) % array_size,
= (10 - i) % 10
→ array [i] → Access value
→ if (Flag < 0) → out of range error
if array [11] → Flag = (10 - 11) % 10 = -1
Similarly for others flag will be negative if out of range & since slightly out of range so no worry about same flag value repeating. This way we can detect out of range access with slight delay.

3. Consider an N-element array. We wish to support two concurrent operations:
   a. Any thread can update any entry (one at a time). Consider the i^{th} entry. Example: *A[i] = 3*
   b. Collect an atomic snapshot of all the entries. This means that we need to collect the instantaneous state of the entire array that existed at a certain point in time.

   Assume that the array starts out to be {1,2}. A process *P* is trying to collect a snapshot. It reads the first element as 1 and then waits for some time. The array is then modified to {2,3}. P will then read the second element as 3, and will return {1,3} as the snapshot. The fact is that {1,3} was never the state of the array. It was either {1,2}, {2,2} or {2,3}.

   How do we implement the *update* and *snapshot* operations using a lock-free program? This means that the entire system should make progress, i.e, one of these operations should always complete for some threads. Do not propose trivial or incorrect solutions.

   Assume that every entry is a 64-bit integer: lower 32 bits are used to store the value, and the remaining 32 bits can be used for any other purpose. Thoroughly justify your answer. [10]

→ To implement such a system we can use the Compare and swap and atomic-read operations to design a lock-free system. We can maintain a queue of updates and snapshots using ticket system.

1) For update add it to queue Then do a ~~Compare and swap.~~

If old A[i] = the value we currently have then no changes have been made and we can update the value at A[i] ie. swap old value with new value.

2) For snapshot add to queue and when chance comes ie. its our ticket number we perform AtomicRead ie. Read it entirely instantaneously so no changes in between can affect the output.

3) To store ticket number we can use upper 32 bits and Compare them with ticket number+1, if its equal then its our turn next.

4) Each operation call is added to queue so one of these operations always completes for one of the threads.

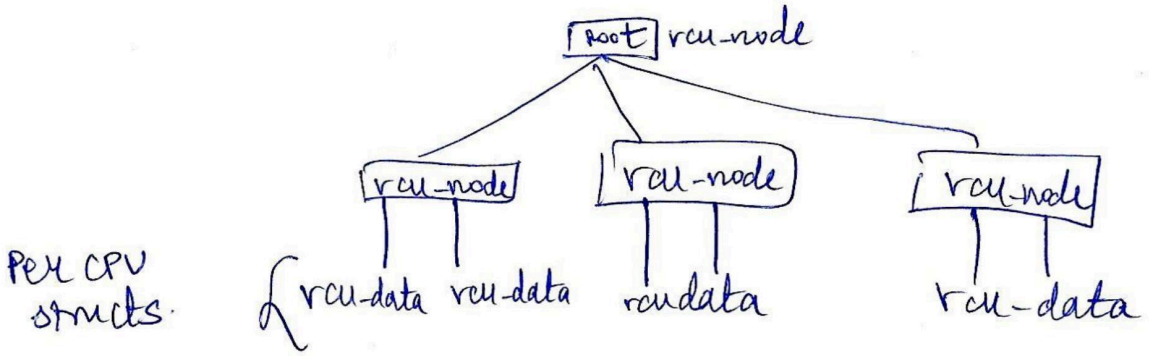
Since we have used CAS, Atomic_read, queues and ticketing this is a lock-free program to update and take screenshot.

4. Answer the following questions regarding RCU:
   a. Why is it very expensive to send IPIs to all cores and then figure out when the grace period has ended? [3]

sending IPIs to all cores requires switching to kernel level then sending an interrupt and then waiting for the return to check if that particular core has ended its threads grace period or are there still readers left. All this is a costly and time consuming since it requires kernel thread level operations and inter core communication. We need to check for each core separately which is expensive.

b. What is the more efficient mechanism that tree-RCU uses? How and why does it work? [7]

Tree RCU uses augmented Tree structure.
Every ~~task~~ CPU has its struct rcu_data structure
This indicates whether it has entered the quiescent state or not ie. all readers have finished reading after the grace period.
We can register some callback functions here as well.
Then these rcu_data are child of struct rcu_node
If all child ie. rcu_data have entered quiescent state ie. end of grace period.
Then rcu_node denotes this by setting its bit.



since its augmented tree so we can only check the root node to find if all the cores have ended their grace period and entered the quiscent phase. This is because each node in augmented tree maintains information about its child nodes. Also which CPU is still in grace period (bit = 0) can be found in O(log N) time. If all of ended their grace period or not is just O(1) operation.

5. Explain the following regarding reverse mapping:
   a. Given a *struct page* and a *vma*, how do we find the virtual address in the *vma* that points to the page? [3]

We need to Traverse the vma and struct page through the 5 layers
Page Global Descriptor PGD
P4D → Page 4 Level
PUD → Page Upper Level
PMD → Page Middle Level
PTE → Page Table entry.
We traverse the vma through sequence numbers to find the page table entry that contains the virtual address in vma that points to the page.

   b. Justify the many-to-many mapping between vmas and anon_vmas? [3]

One
~~Many~~ pages can have many vmas. To
provide an abstraction in between
anon_vmas are used. one page can
map to many anon_vmas and those
map to one vma. Since multiple
anon_vmas from different pages can map
to different vmas Hence there can be
a many to many mapping between vmas & anon_vmas

c. What will happen if the red-black tree is replaced with a sorted array? Consider cache behavior, concurrency and efficiency. [4]

Cache access will take ~~less~~ more time
since we will need to ~~scan~~ fetch the
first value from the array which is sorted.
or scan through the entire array.
Concurrency control will become more
difficult & complex since Red/black
trees have these labels which can be
used to identify concurrent access.
Overall efficiency will decrease because
in worst case we will need to access
the last entry which takes O(N)
instead of O(log N) in Red black tree.

6. Answer the following questions in the context of the MGLRU algorithm.
   a. When should *min_seq* be incremented? [2]

when a page is aged and added to old generation then min_seq should be incremented.

b. When is it a good idea to evict ANON pages? Explain with examples. [2]

Anonymous pages are not file backed so its good idea to evict them if Random pages are accessed in longer duration and Random page access is a less costly operation.

c. When is it a good idea to evict FILE pages? Explain with examples. [2]

File pages are those which are file backed so its good idea to evict if file pages are being accessed in longer durations and the cost for accessing file pages is lessere example when we have loaded certain code modules / libraries

d. Why could file-backed pages show artificially high refault rates, even if they have not been accessed? [Hint: Think about how the library and kernel code try to optimize file accesses.] [4]

The library and kernel code try to keep related code and files in contiguous groups and load these as a whole so that the cost of accessing files that are needed more frequently is less. Due to this file backed pages show artificially high Default rates because they are loaded in groups (Folios) so even though they might not be accessed the Default rate is high since fetching of these file backed pages is a cheap operation. So we tend to evict them more often.

---------- Rough sheet ----------

update, ⟶ CAS

Atomic SS ⟶ Atomic_read

Array size = 10

11-15 out of bounds

arr[0]

arr [index % size]

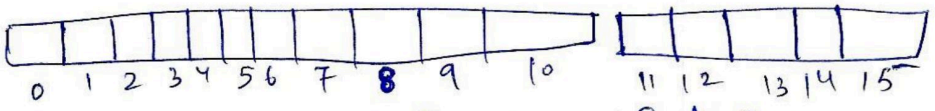
11 % 10 = 1

(10 - 0) % 10

Flag = (10 - 0) % 10

= 0 % 10 = 0

1 → 1 0 → 1 0 → 0



(10 - 11) % 10 → 0

During Array Indexing → Flag = Set

= -1 % 10

= -1

Flag = Set → out of bound error

8 4 2 1

1 1 1 1

1 0 1 0

	8	4	2	1				
0	0	0	0	0				
1	0	0	0	1				
2	0	0	1	0				
3	0	0	1	1				
4	0	1	0	0				
5	0	1	0	1				
6	0	1	1	0				
7	0	1	1	1				
8	1	0	0	0				
9	1	0	0	1				
10	1	0	1	0				
11	1	0	1	1				
12	1	1	0	0				
13	1	1	0	1				
14	1	1	1	0				
15	1	1	1	1				