End-Semester Examination
CS330: Operating Systems
Department of Computer Science and Engineering

**Total marks: 70** **Duration: 150 minute**

**Roll No:** 210355 **Signature:** Divyansh

The exam is open books, notes, printed materials. No electronic device should be used during the examination. Consulting/discussing with anyone during the exam is strictly prohibited. *Unless specified explicitly, assume 64-bit X86 architecture with multiple CPUs and an OS with no bugs*
*Best of luck!*

*DO NOT WRITE ANYTHING IN THIS AREA*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Q1 | 6 | Q2 | 6.5 | Q3 | 0 | Q4 | 0 |
| Q5 | 6 | Q6 | 1 | Q7 | 5 | TOTAL | 24.5 |

1. Select the correct answer(s) for the following questions. Note that, you have to *mark all correct choices* to get the credits (no partial marks). No explanation required. $3 \times 5 =$

(a) Which of the following file related system call(s) *can not always be* served without performing any disk I/O before returning to user space after their *successful* execution? Assume a file system that caches *only* the disk blocks corresponding to the inode bitmap, block usage bitmap, inodes and the super block.

A. `lseek`
B. open a file
C. open with `O_CREAT` (i.e., create a file)
D. `read` 8192 bytes from the beginning of a file of size 1MB
E. `write` 8192 bytes at the end of the file of size 1MB

0

(b) Execution of `fsck` utility for a given file system before mounting revealed a mismatch between the number of data blocks calculated from the data block bitmap and all inode data pointers. Assume that, the file system was consistent and fully persisted to disk just before the last on-going operation during which the system crashed. The last on-going operation can be

A. creation of an empty file or directory
B. truncation of a file (deleting file content)
C. write to an existing block of a file (update operation)
D. write to the end of the file (append operation)
E. deletion of a file or directory

0

(c) Consider the following pseudocode.

```
sem_t S1, S2, S3; /* All initialized to zero */
string(){                          number(){
    wait(S1);                          printf("1");
    printf("A");                       post(S1);
    wait(S2);                          printf("2");
    printf("B");                       post(S2);
    wait(S3);                          printf("3");
    printf("C");                       post(S3);
}                                  }
```

Which of the following is/are *possible* output(s) if one thread executes `string()` and the other thread executes `number()` concurrently?

A. 1A2B3C
B. 1AB23C
C. 123ABC
D. 12ABC3
E. 12A3BC

3

(d) Consider two threads $T_1$ and $T_2$ of a process $P$ created using the pthread API in the Linux system. Assuming X86 system with split-mode addressing, which of the following statement(s) is/are *false*.

A. If $T_1$ is in kernel mode, $T_2$ *is not* allowed to enter the kernel mode

B. The kernel stack used by $T_1$ and $T_2$ are different

C. The OS page fault handler *must* handle the page faults caused by $T_1$ and $T_2$ in a serial manner.

D. When the OS scheduler performs a context switch between $T_1$ and $T_2$, the page table base pointer (CR3) register is not changed

E. When the OS scheduler performs a context switch between $T_1$ and $P$, the page table base pointer (CR3) register is changed

3

(e) Consider that there is a page fault for virtual address $V$ of a multi-threaded process on a single CPU system. When the OS page fault handler inspects the page table mappings, it finds that the page table entries are correct for all the levels before the leaf level. Note that, the leaf level is the level-4 PTE in a 4-level page table where the page table base register (e.g., CR3 in X86) points to the level-1 PTE. In the leaf-level page table entry (PTE),

A. the present bit can be zero

B. the present bit can be one

C. the write bit can be one (both read-write allowed)

D. the content can be correct in all aspects if interrupts are disabled during page fault handling

E. the content can be correct in all aspects if interrupts are enabled during page fault handling

0

2. True or False with justification. Justification should be a precise argument supporting your verdict. No marks without proper reasoning. $2 \times 5 = 10$

(a) Physical memory usage of a system will decrease on successful execution of the exec system call.

1.5 False, there can be two cases, one can be the case of Lazy allocation where actual physical mem was not allocated before calling the exec system call. The second scenario can be when the new binary loaded after exec is more physical memory mapped. ?

(b) If a file system always performs non-cached I/O, there will be no consistency issues.

0.5 False; Since all the operations are directly performed on disk and something unavoidable occurs in between and inode mappings couldn't be corrected ?

(c) Any physical memory frame containing the OS code can be swapped out.

0.5 True, Since we are maintaining the previlege bit in the mapping of swapped memory as well, OS code will not worry u

(d) For any memory access sequence, LRU and Belady's optimal (MIN) page replacement will never result in more number of page faults than the FIFO page replacement.

2 False, for # frames =3 and sequence 1, 2, 3, 1, 4, 2, 3 ⇒ LRU will give 6 page faults while FIFO will give 4. Contradiction!

(e) A full TLB flush is necessary during a process context switch.

False, we can maintain a ASID to differentiate between different processes while using the same TLB.

2

3. Consider the following code fragment in a system with *five* levels of page table and a page size of *four* kilobytes.

```
int d_calc(int a, int b){
    register int ctr = 0;
    while(ctr < a){
        b += a;
        ctr++;
    }
    return b;
}
```

```
//Disassembly (R1, R2, R3 and A are registers)

.function d_calc                   //function  d_calc, argument 'a' --> R1 and 'b' --> R2
0x7FF001080000: mov $0, R3         //Load R3 with zero ('ctr' --> R3)
0x7FF001080004: cmp R3, R1         //Compare 'ctr' with 'a'
0x7FF001080008: jge (+0xC)         //Jump to 0x7FF0010800018 if 'ctr' >= 'a'
0x7FF00108000C: add R2, R1         //Add R2 = R2 + R1 ('b' += 'a')
0x7FF001080010: inc R3             //Increment counter ('ctr'++)
0x7FF001080014: jmp (-0x14)        //Jump to 0x7FF001080004
0x7FF001080018: mov R2, A          // mov R2 to the register A (return value)
0x7FF00108001C: ret                // Return
```

Assume that there are no page faults during the execution and only the above function is executed in one scheduling interval without any context switches. Answer the following questions. $\boxed{2+3+2+2+2=1}$

(Briefly mention the calculation steps along with the final answer)

(a) In a MMU with no TLB support, what will be the number of memory accesses to perform *address translation* if d_calc(0, 100) is executed?

# fetch = 2 (line *04 and line *18)

# mem_access = 2×5 = 10

0

V = 0x7FF0010800

(b) In a MMU with no TLB support, what will be the number of memory accesses to perform *address translation* if d_calc(5, 100) is executed?

#loops = 5, # fetch in 1 loop = 3, +1 fetch while comparing in last.
+1 acc at *18

# total fetch = 5×3 +2 = 17

# mem access = 17×5 = 85

0

02

(c) Write an expression for the number of memory accesses to perform *address translation* in terms of $a$ and $b$ for a MMU with no TLB (no explanation required). [2]

$3 \times a + 2$

0

(d) What will be the *minimum* number of memory accesses required to perform *address translation* if `d_calc(5, 100)` is executed for a MMU with a TLB (assuming an empty TLB at the start of the function)? [2]

| loop-1 | loop(2-5) | out of loop | |
|---|---|---|---|
| $(1+2) \times 5 + 1$ | 3 | +2 | = 16 |

0

(e) Write an expression for the *minimum* number of memory accesses required to perform *address translation* in terms of $a$ and $b$ for a MMU with a TLB (assuming an empty TLB at the start of the function, no explanation required). [2]

$11 + 3(a-1) + 2 = 3a + 10$

0

4. Consider the following implementation as a solution to critical section problem for two threads $T_0$ and $T_1$.

```
int flag[2] = {0,0};
int turn = 0;
void lock (int id) /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ^ 1] && turn == (id ^ 1)); /* ^ is the bit-wise XOR*/
}
void unlock (int id)
{
    turn = id ^ 1;   // (S1)
    flag[id] = 0;    // (S2)
}
```

Does the above implementation guarantee mutual exclusion? If yes, provide a formal argument for correctness. If not, show a concrete example when the mutual exclusion is violated.

No it does not.

A. The above implementation guarantees mutual exclusion.

Arguments: In the lock function, the condition inside the the while loop demands that the flag of other is 0 or the turn is same as the current id. The violation of any one condition will lead to assigning lock to other

0

The important point to notice here is that both the condition can independently terminate the while loop while on the other hand. demands both to be correct in order to loop.

Robustness to context switch: If context switch happens after assigning of flag in $T_0$, even then "turn" condition will be violated (in the lock). If context switch occurs after assigning turn (in unlock) then the flag condition will be violated.

5. Consider the following pseudocode which is executed by three threads $T_1$, $T_2$ and $T_3$ concurrently. Assume that, the pointer assignment at line #5 is atomic.

```
1. int *ptr = NULL; //Global, shared across threads
2. sem_init(S, 0);  //Initialize the semaphore S with value 0
3. void process(){
4.     if(!ptr){
5.         ptr = malloc(sizeof(int)); ← atomic.
6.         *ptr = 0;
7.         sem_post(S);
8.     }
9.     sem_wait(S);
10.    sem_wait(S);
11.    atomic_inc(ptr); // Atomically increment (*ptr) ← atomic.
12.    sem_post(S);
13.    sem_post(S);
14. }
```

$\boxed{6 + 1.5 + 1.5 =}$

Answer the following questions.

(a) List down all unique deadlock scenarios along with the line numbers for the deadlocked threads. The uniqueness of the deadlock scenario is based on the interleaving (i.e., threads deadlocked at different line numbers, not permutations of threads participating in the same interleaving).

① $T_1$ allocated memory to ptr and came to line Line #10 here $S=0$ and ptr is allocated ~~and T~~ so $T_1$ is waiting while $T_2$ and $T_3$ will directly reach (consider this scenario) to line #9 now since ptr is allocated and hence they are at dead lock on line #9.

1.5

② consider this scenario where $T_1$ executed till 4 then context switch happened and $T_2$ executed till 10 and allocated ptr ( Now again context switch happened to $T_3$ and it reach line 10 and again switched to $T_1$ and it executed till 10, so all 3 are ~~at~~ in a deadlock at same line → #10.

③

1.5

(b) If all threads complete without any deadlock, what will be the minimum number of malloc calls (at line #5) and why? 1.5

Minimum number of malloc calls will be 2 since to cross line 9 and 10 we need S≥2 and sem-post are only occuring in if blocks (before line 9 and 10) which containe the malloc call operation (before it).

1.5

(c) If all threads complete without any deadlock and print the value of (*ptr) at the end of the function, what will be the minimum value of (*ptr) (across all possible execution orders) and why? 1.5

1

The minimum value of (*ptr) will be since for any function to reach the end it will have to pass through 11,12,13 line but if if context switch occurs. and control is returned inside if ~~state~~ block then (*ptr) will be reallocated as 0. but the last printing thread can't avoid increment hence 1.

1.5

6. Consider an indexed file system found in the UNIX operating system. The inodes have an indexed allocation scheme as follows: eight direct block address, four single-indirect block address and two double indirect block addresses. Assume that, the block size and on-disk inode size are one kilobyte (1KB) and 256 bytes, respectively. Block address in the system is 4 bytes long. Further, assume that every directory maintains an array of flat directory entries having the following structure,

```
struct dentry{
        u32 inode-number;
        char name[252];
}
```

Assume that the disk I/O operations are performed in units of block size i.e., 1KB. Answer the following questions (no explanation required). $\boxed{1+1+3+3=8}$

(a) What is the maximum disk size supported by the file system?

$2^{42}$ bytes

1

(b) You have written a program to create as many files as possible under an initially empty directory. How many files your program can successfully create?

4

0

★ (c) A user process opens and successfully reads an existing file at /home/user/courses/os/notes.txt of size one kilobyte, where / is the file system mount point. Assume that, the super block, all the inodes and index meta-data required to access the file and directory are present in memory. What is the minimum and maximum number of disk block accesses required to complete the read operation?

max = 17, min = 17

0

(d) In the above question, assume that the file notes.txt is of size 1MB and the user process opens the file (with the path same as previous question), after that it reads one block (1KB) at a random block offset (offset is a multiple of 1KB) successfully. Further assume that, the super block, all inodes are always cached. However, the file indexes and directory content may or may not be cached. Data blocks of file ~~and directory~~ are not cached. What is the minimum and maximum number of disk block accesses required to complete the read operation?

max = 17, min = 17

0

7. Memory is a precious resource and you want to augment the virtual memory subsystem of the OS to compress the used physical memory to increase the degree of multiprogramming without using swapping. Assume that there is an existing compression logic in the OS which provides functionalities like compression and decompression. Specifically, you can assume the following two OS level functions *are already available*,

`int compressPFN(u32 srcpfn, void *outPTR)`: Compresses 4096 bytes of data in the page frame `srcpfn` and stores the compressed stream into `outPTR`. The length of compressed data is returned by this function.

`void decompressPFN(void *inPTR, u32 dstpfn, u32 size)`: Decompresses compressed data of length = `size` stored at `inPTR` and stores the decompressed data in the `dstpfn`. The caller of this function should ensure that the decompressed data is of length 4096 bytes.

The solution framework is as follows: There is a background user space process (a daemon, say `cmpserv`) which invokes a new system call i.e., `compress` in a periodic manner. The design of `compress` system call (parameters, return value and implementation) is to be discussed as part of the answer. You are required to discuss the implementation of the `compress` system call—both the mechanism and policy.

*Mechanism:* How exactly the compress system call is implemented? What are the side effects on other OS subsystems (like virtual memory etc.) and your proposed methods of handling them. Your implementation should handle both correctness and efficiency.

*Policy:* In this part, you are required to discuss answers to questions like which memory pages to compress, how many memory pages to compress at what time etc. Note that, you should discuss the resource tradeoff implications of your policy e.g., CPU usage vs. memory savings.

Hint: you may not want to compress all the memory pages because of performance reasons. Assume page size of 4KB and X86 like paging support. [1

Parameters of compress:- (addr: virtual addr of user address space, int flag: hinting for the need for compression or decompressing)

5

Mechanism :-

Note: There will be two parts to this implementation, when the routine call "cmpserv" is called from user space. 2nd scenario will be when the contents from the current memory is being accessed, in this case a page-fault error will raised ~~by~~ and OS will invoke the compress call. The identification of this type will be done by the flag.

compression part. The virtual address passed to

flag=0 :-