CS 219 Spring 2025 Mid-semester exam KEY

(10 questions, 60 marks, 30% weightage)

Name:_____ **Roll number:**_____

Questions 1–10 carry 6 marks each. Write your answer neatly in the space provided.

1. For each of the statements below on process management, indicate if the statement is true or false. There is no need to provide a justification.

   (a) When a hardware device raises an interrupt, the trap instruction (e.g., `int n` in x86) is executed by the kernel device driver. **Ans:** False

   (b) A preemptive CPU scheduler performs only voluntary context switches, i.e., it switches out a process only when it blocks or exits. **Ans:** False

   (c) When a parent process with 2 child processes (both blocked on disk I/O, and in sleeping state) invokes the wait system call (the default variant), the system call blocks the parent process. **Ans:** True

   (d) Every switch from user mode to kernel mode during the execution of a process leads to a context switch. **Ans:** False

   (e) During the execution of the trap instruction (e.g., `int n` in x86), the program counter and stack pointer (EIP, ESP) are saved on the kernel stack by the hardware itself. **Ans:** True

   (f) When a parent process terminates while its child process is still running, the child is also terminated immediately via a signal by the OS. **Ans:** False

2. Answer the following questions about the context switching code in xv6.

   (a) The `swtch` function takes two arguments: the address of the context pointer of the old process, say `struct context **oldc`, and a pointer to the context of the new process, say, `struct context *newc`. After the registers are pushed on the stack of the old process, two steps happen: the old context pointer is updated with the latest value of `esp`, and the stack pointer shifts to the context of the new process. Write down the code for these two steps using C-like syntax. Your answer must have two assignment statements using the variables `oldc` and `newc` declared above, and the stack pointer `esp`.

   Ans: `*oldc = esp; esp = newc;`

   (b) Why does the `swtch` function push and pop only some registers, and not all registers, while saving and restoring context?

   Ans: Only callee-save registers need to be saved inside the function, as caller-save registers would have been saved before entering the function code, by the caller.

   (c) During the `swtch` function, does the `esp` contain addresses from high virtual address space (belonging to the kernel) or from the low address space?

   Ans: high virtual address

(d) A process in xv6 can invoke the `sched` function (which in turn calls `swtch`) from 3 different places. Two of these are from the `sleep` function which blocks a process, and the `exit` system call which terminates the process. What is the third place?

Ans: yield (timer interrupt)

(e) A process in xv6 can return from `swtch` and resume execution in three different places in the kernel code. One of these is the `forkret` function where new processes start. List the other two.

Ans: yield and sleep

3. Consider a process P that forks a child process C in a simple OS. After fork completes, the parent and child processes separately map a shared memory segment of one page into their virtual address space. This shared memory page starts at virtual address S_P in process P, and at S_C in C. The kernel code and data are mapped starting at virtual address K in both processes. Let $A_Q(V)$ denote the physical address corresponding to virtual address V when computed using the page table of process Q. State whether the following statements are true or false. (No justification needed.)

   (a) We will always have $S_P = S_C$. **Ans:** False

   (b) We will always have $A_P(S_P + X) = A_C(S_C + X)$ for every offset X in the shared memory page. **Ans:** True

   (c) We will always have $A_P(V) = A_C(V)$ for all valid virtual addresses V. **Ans:** False

   (d) We will always have $A_P(V) = A_C(V)$ for all valid virtual addresses V in the user part of the address space. **Ans:** False

   (e) We will always have $A_P(K + X) = A_C(K + X)$ for every valid offset X in the kernel part of the address space. **Ans:** True

   (f) We would have had $S_P = S_C$ if the shared memory segment was mapped in the parent process before the fork system call was executed. **Ans:** True

4. For each of the statements below on memory management, indicate if the statement is true or false. There is no need to provide a justification. Ignore the presence of CPU caches in this question.

   (a) When the CPU accesses a virtual address and the MMU sees a TLB hit, then the physical memory need not be accessed to fetch the actual memory contents. **Ans:** False

   (b) Every TLB miss always results in the MMU walking the page table to fetch the page table entry.**Ans:** True

   (c) Every TLB miss always results in a page fault and a trap to the OS. **Ans:** False

   (d) If the page table entry corresponding to a virtual address has both the valid and present bits set, then accessing this address will always result in a TLB hit. **Ans:** False

   (e) If the page table entry corresponding to a virtual address has both the valid and present bits unset, then accessing this address will always result in a TLB miss. **Ans:** True

   (f) Every transition from user to kernel mode during the execution of a process results in a switch of page tables also. **Ans:** False

5. Consider a simple system with an 8-bit virtual address space, 16 byte pages, and 4 byte page table entries (PTEs), and no demand paging. You are given below the contents of a few pages in the system, which are part of the two-level page table of a process, and hold page table entries.

- The outermost page directory contains the PTEs [X, 0, Y, 0], with pointers to inner page table pages at frame numbers X and Y.
- The inner level page table at frame number X contains the PTEs [7, 0, 3, 0].
- The inner page table at frame Y contains the PTEs [2, 5, 0, 4].

Note that the PTEs in a page are represented as an array, where a positive number is the physical frame number of a valid PTE, and an invalid PTE is indicated by the number 0. What happens when we translate the virtual addresses given below using the page table shown above? Indicate if the translation succeeds or results in a trap. Further, if the address can be translated, show the translated physical address in binary or decimal format.

(a) 12
Ans: VA = 0000 1100, PA = 0111 1100 = 124

(b) 42
Ans: VA = 0010 1010, PA = 0011 1010 = 58

(c) 123
Ans: Trap (maps to second entry in outer page table, which is invalid)

(d) 132
Ans: VA = 1000 0100, PA = 0010 0100 = 36

(e) 156
Ans: VA = 1001 1100, PA = 0101 1100 = 92

(f) 196
Ans: Trap (maps to fourth entry in outer page table, which is invalid)

6. Consider a system with a 42-bit virtual address space, 4KB pages, and 4 byte page table entries (PTEs). The system uses hierarchical multi-level paging for memory management. Recall that 1KB = 2^{10} bytes, 1MB = 2^{20} bytes, 1GB = 2^{30} bytes. Consider a process running in this system.

(a) What is the maximum number of pages in the virtual address space of the process?
Ans: 2^{30}

(b) How many levels does the page table of the process have?
Ans: 3

(c) What is the maximum number of pages in the innermost level of the page table?
Ans: 2^{20}

(d) What is the maximum number of pages required to hold all levels of the page table?
Ans: $2^{20} + 2^{10} + 1$

(e) For the rest of the question, assume that the process has accessed 2^{10} unique pages in its address space. The OS allocates all pages, including pages for page tables, on demand. That is, page table pages are allocated when they contain at least one valid entry. What is minimum possible number of pages required for the page table of the process, across all levels, to hold these 2^{10} valid PTEs?
Ans: 3

(f) For the above question, what is maximum number of pages that may be required for the page table of the process, across all levels, to hold the 2^{10} valid PTEs?
Ans: $2^{10} + 2^{10} + 1$

7. Consider the system described in the previous question. The virtual address space of a process in this system contains 2 MB code (with compile time data), a 2MB gap of invalid address space, 2MB heap, another 12MB gap, and a 2MB stack in the user part of the address space. The kernel code and data are mapped to start in the last 1GB of the address space, and contain 200 MB of code and data. The rest of the virtual address space is not used. Consider the multi-level page table that contains page table mappings for this address space. Assume that all pages of the page table are allocated on demand, i.e., only when they contain at least one valid entry.

   (a) How many pages are required to hold these page table mappings in the innermost level of the page table of the process? That is, how many pages are present in the innermost level of the page table?
   Ans: 53

   (b) Consider the first page in the set of pages allocated for the innermost level of the page table. How many valid PTEs does it have?
   Ans: 512

   (c) How many valid PTEs are present in the innermost level of the page table, across all pages in the innermost level?
   Ans: 3 X 512 (for user) + 50 X 1024 (for kernel) = 52736

   (d) How many pages are needed in each of the remaining levels of the page table of the process (other than the innermost level)? You must list the number of pages needed at each level (other than innermost level) separately.
   Ans: 2 in the mid-level and 1 in the outermost level

   (e) How many valid PTEs are present in the outermost page directory?
   Ans: 2

8. Consider a process that accesses the following pages during its execution (virtual page numbers are provided below in the order of access).
   `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 2, 7`
   You may assume that none of the pages are in memory when the process starts execution, and are fetched into memory only upon a page fault. The OS allocates only 3 physical frames to hold these pages of the process, and evicts one of the pages of this process while handling a page fault.

   (a) Assume the OS uses a FIFO page replacement policy. Compute how many page accesses amongst those shown above will result in a page fault.
   Ans: 11

   (b) For the previous question, state which pages will finally be present in memory at the end of all the accesses.
   Ans: 3, 0, 7

   (c) Compute the number of page faults if the OS uses an exact version of the LRU page replacement policy.
   Ans: 10

(d) For the previous question, state which pages will finally be present in memory at the end of all the accesses.

Ans: 0, 2, 7

(e) Now assume that we are using a hypothetical optimal page page replacement policy, that looks into future accesses and evicts the page that won't be used farthest into the future. Compute the number of page faults in this case.

Ans: 8

9. Consider a 32-bit system, with 4KB pages, and 4 byte PTEs. The OS uses a two level page table, like xv6, and does not use any demand paging. Consider the following snippet of OS code (shown in pseudocode, do not worry about syntax errors) to count the number of valid PTEs across both levels of the page table. Fill in the missing blanks so that the code computes the number of valid PTEs correctly.

   A brief explanation of the code is given below. The code iterates over each entry in the outer page table and checks if it is valid. For each valid PTE in the outer page table, we locate the corresponding page of the inner page table using the physical address in the outer PTE, and iterate over all entries in this inner page table page once again. The address `PTBR` stores the physical address of the outer page table, and will be used in the completed code below. The function `getPTE(X)` returns the PTE starting at physical address `X`. Recall that 4-byte PTEs are stored as an array inside each page of the page table, and you must use this information to correctly compute the address of each PTE. The function `isValid(pte)` returns true if `pte` is valid, false otherwise. The function `getPA(pte)` returns the starting physical address of the frame number contained in `pte`. The variable `count` should finally have the count of valid PTEs across both levels.

```
for(i = 0; i < ________ ; i++) { //fill number
  opte = getPTE(_____________) // fill address
  if(isValid(opte)) {
    count++
    inner_pa = getPA(_________) //fill argument
    for(j = 0; j < _______ ; j++) { //fill number
      ipte = getPTE(_____________) // fill address
      if( isValid(___________) ) //fill argument
        count++
    }
  }
}
```

Ans:

```
1024
PTBR + 4*i
opte
1024
```

```
inner_pa + 4*j
ipte
```

10. Consider an implementation of a heap to manage variable sized allocations as studied in class. The free list is maintained as a linked list of `struct freenode` elements that are embedded in each free chunk itself. Every `struct freenode` stores the starting virtual address and size of the free chunk, along with a pointer to the next free chunk. This free list is maintained in sorted order, and is traversed when allocating and freeing chunks from the heap.

```
struct freenode {
  int start;
  int size;
  struct freenode *next;
};
```

    (a) During insertion of freed up chunks into the free list, we need to check if a free chunk lies between two other free chunks in the memory layout, so that we can insert it in the correct sorted position. Write down the condition to check if a given `struct freenode *n` lies between two other free chunks `struct freenode *left` and `struct freenode *right`. To be clear, you must write the condition that must hold on the start addresses and sizes of these free nodes when `n` comes after `left` and before `right` in sorted order, when the free chunks are ordered in ascending order of virtual memory addresses.

    Ans: `left->start < n->start < right->start`

    Assuming chunks are non overlapping, it suffices to just check for start values. We will also consider answers that use size also, and for answers that allow for size of headers.

    (b) The heap must periodically check adjacent free chunks in the sorted list to see if they can be merged into one bigger free chunk. Write down the condition that must hold on `struct freenode *left` and `struct freenode *right` if they can be merged with each other.

    Ans: `left->start + left->size = right->start`
    and optionally `left->next = right`

    (c) Continuing on the previous question, assuming we decide to merge the left and right free chunks, describe how you would update the start address, size, and next pointer of the left free chunk `struct freenode *left` such that it now becomes the merged free chunk (and the pointer to the right free chunk need not exist in the linked list anymore).

    Ans: `left->size = left->size + right->size`
    and `left->next = right->next`