CS330 - End Semester Exam (Part - II)

Total marks: 60 **Duration: 150 mins**

The exam is open books, notes, printed materials. You are allowed to browse the web. Consulting/discussing with anyone which includes asking questions in any forum during the exam is strictly prohibited. *Unless specified explicitly, assume 64-bit X86 architecture with multiple CPUs.* Be precise with your answers. No marks for vague answers. Make reasonable assumptions and state them explicitly.
All the best!

1. Consider the following program in the UNIX/Linux system.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
   int pid;
   int mpid = 0;
   for(int ctr=0; ctr<5; ++ctr){
     pid = getpid();
     if(pid % 2){
         mpid = fork();
     }
     if(mpid % 2 == 0)
         dup(1);
   }
}
```

Assume that all system call invocations are successful. [2 + 4 = 6]

   (a) What is the maximum and minimum number of processes created during the execution of the above program (including the main process)?

   (b) Let us define R to be the number of references to the `STDOUT` file object. Assuming the value of R is 1 at the start of the program (before execution of the `for` loop), what is the maximum and minimum value of R during the execution of the program?

2. In the classical `fork()` implementation, the child inherits the parent address space i.e., both the memory pages and page tables are copied. This requires creating duplicate content in memory at the time of `fork()` which can be inefficient, both in terms of memory and CPU usage, especially when the child process invokes `exec` immediately after returning from `fork`. We want to design a variant of `fork` (say `forkopt`) such that new executables can be launched faster than `fork` in a memory efficient manner. [10]

   While designing `forkopt`, you are not allowed to modify the semantic of fork. However, you may propose some restrictions related to the scheduling order (between the parent child process) and the nature of execution of the child process. Clearly explain the working of `forkopt` along with design and implementation aspects of `forkopt`. Further, if you are imposing any restrictions on the scheduling and/or on the nature of child process execution, describe why the restriction is required and what is your strategy to impose the restriction in the OS.

   Hint: Maximum efficiency is achieved when the page table is shared between parent and child processes (similar to threads) as long as possible.

3. 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 existing OS level functions, [14]

   `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.

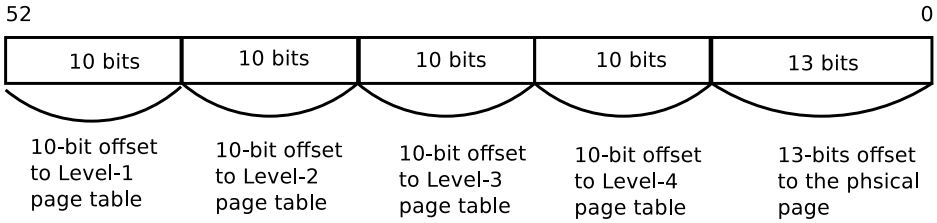
   `decompressPFN(void *inPTR, u32 dstpfn, u32 size)`: Decompresses compressed data of length = `size` stored at `inPTR` and stores the decompressed data in the `dstpfn`. The caller of this function should ensure that the decompressed data is of length 4096 bytes.

   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 explained as part of the answer. You are required to discuss the implementation of the `compress` system call—both its 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? Note that, your implementation should handle both correctness and efficiency aspects.

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 also 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.



4. Consider a 4-level paging system for fifty-three bit (53-bit) virtual address (as shown above) and a page size of eight kilobytes (8KB). For simplicity, assume that, the OS is executing on a separate address space and no OS address is mapped in the user process address space. Answer the following questions, $1+2+2+5=10$

   (a) If there are N processes, what is the maximum address space size of each process?

   (b) What is the maximum amount of memory which can be allocated using a single page table page at every level till Level 3?

   (c) The maximum physical memory supported by the above paging system is given as one petabyte (where 1 petabyte = 2^{50} bytes). If ten bits are used to maintain the access flags and five bits are reserved by the hardware, how many bits remain unused in any page table entry?

   (d) If a process successfully unmaps 4GB memory from an existing mapping using a single `unmap` system call (every virtual address is uniquely mapped to a physical address), what is the maximum and minimum amount of free memory increase in the system? (ignore the memory usage change due to manipulation of VM related data structures like `vm_area` etc.)

5. Consider the following implementation as a solution to critical section problem for two threads T_0 and T_1. 7

```
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.

6. We would like to design a function that atomically dequeues the element E from the head of a queue $Q1$ and enqueues E at the tail of a queue $Q2$. The function `AtomicMove` shows a buggy attempt at designing such a function. Every queue object has a mutex M (initialized as `unlocked`). 5

```
void AtomicMove (Queue *Q1, Queue *Q2)
{
    mutex_lock(Q1->M);
    mutex_lock(Q2->M);
    Item E = Q1->Dequeue();
    Q2->Enqueue(E);
    mutex_unlock(Q2->M);
    mutex_unlock(Q1->M);
}
```

Point out the problem with this implementation and present a correct implementation.

7. 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 four double indirect block addresses. Assume that, the block size and on-disk inode size are four kilobytes (4KB) 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., 4KB. Answer the following questions. $\boxed{1+3+4=8}$

(a) 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?

(b) A user process opens and successfully reads an existing file at `/home/user/courses/os/notes.txt` of size four kilobytes, where `/` is the file system mount point. Assume that, the super block, all the inodes and index meta-data required to access the file are present in memory and no other blocks are cached. What is the minimum and maximum number of disk block accesses required to complete the read operation?

(c) In the above question, assume that the file `notes.txt` is of size 1GB and the user process opens the file (with the path same as previous question), after that it reads one block (4KB) at a random block offset (offset is a multiple of 4KB) 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. No file data is cached. What is the minimum and maximum number of disk block accesses required to complete the read operation?