Kenneth Woodard

CS 361: Operating Systems

Final Exam

December 13, 2022

1.

a. The program would be expected to run faster on the processor with a MMU with a one-level page table. The more levels you use can help organize your memory better; thus, improving the management of the memory, but this increases the complexity. The one-level page table it takes less time to find each table entry because it only has to go to one address in memory. Also, if there is a TLB miss, it will also require grabbing more memory for a two-level system to find the page table entry. This will take more time. This is why it increases the time required to read and write to memory when there are more levels being used.
b. The principle advantage of using a two-level page table is that less memory is required. If you used a single-level page table, a lot of useless space in memory will be used up. The two-level page table facilitates only using a slice of memory you need while running a process. This lowers the amount of RAM required for the program. It does this by organizing the data in a compact way where there is less unused memory separating important data.
c. A switch between user processes causes a temporary lower hit rate for the TLB because one process is most likely going to use different page table entries than another process will. The solution is to flush the TLB. Since the TLB will be filled with instructions/ addresses that aren't relevant with the new program, TLB misses are to be expected temporarily.
d. Temporal locality describes how an instruction that has been used recently is saved for later in the cache. It does this so that it can access these instructions faster (because the cache is faster). Obviously, this increases the efficiency of a running process. This will increase the effectiveness of the translation look-aside buffer because it will decrease the

chances of the TLB missing. This is because temporal locality essentiality improves the location of the memory we are likely to use next. The TLB uses temporal locality to increase performance.

2. Lottery scheduling implements a lottery-type system for choosing which process will run first. Each process that is scheduled to run is assigned a number of tickets. A list is used to keep track of the processes to run and their corresponding tickets. Each ticket is numbered and each process has a range of tickets (example: tickets numbered 0 to 50 for process A, tickets numbered 50-125 for process B). A random number generator is then used to select a random number. If ticket number 27 was chosen (in the previous example), process A would be chosen to run first. Subsequently, if it was ticket no. 1000 it would be neither A nor B. Then it would continue this way. Obviously, if you have more tickets, you have a higher probability of priority. Stride scheduling works in a different fashion. Each process or job to run is assigned a *stride*. This is basically inverse to the number of tickets in the previous example. Each stride has a pass value that basically increments to show the progress of the progress. In stride scheduling, if a job's pass value is the lowest, then that job will be selected next. To start, any of the processes can be chosen because they all will have a pass value of zero. This process will run until its pass value equals its stride value. It will do this for each job first. Then, it will run the process whose pass value is less than the other jobs. It will continue to run processes in this fashion: choosing the job with the lowest pass value and running it; reassess, rinse, repeat. If all the pass values are equal, it will act as it did in the beginning. In terms of fairness, stride scheduling is more fair than lottery scheduling. This is because stride scheduling ensures that all the processes are implemented in a semi-simultaneous fashion. For example, even though process A has a higher stride than process B, process A won't be chosen again second until the pass values of each are equal or process A has a lower pass than A. This forces the jobs to be completed close to the same time. It is as if stride scheduling is predicated on guaranteeing equity among the jobs. This contrasts with lottery scheduling because lottery scheduling is predicated on probability. There is no guarantee which process in the queue will be selected at any given time with lottery scheduling. Therefore, there is no way to guarantee that jobs will be completed at

comparable times. When it comes to complexity, lottery scheduling is far simpler in terms of implementation when compared to stride scheduling. Stride scheduling has to maintain the pass value for each job at each point during execution while the lottery scheduling only has to maintain the tickets for each remaining process.

3.

a. During a system call, a control register will hold a system call number. The trap handler then checks the number. If it is legitimate, then it executes the system call. It does this to prevent anything malicious being passed to the OS while in kernel mode. This indirection provides protection for the operating system. Protecting the operating system in turn protects storage devices, etc that are connected to it. Similarly, the memory management unit needs to ensure each process only has access to the memory that it needs, no more. This also provides protection to the system.
b. When a system call is made, the mode needs to switch from user to kernel mode. When this happens the program control register will cause a software interrupt by executing a special trap instruction. The MMU has saved the address info, data, program counter, etc. before this interrupt. This is so it can be restored and used again later.
c. When operating systems isolate different processes from each other, the program control register makes sure that neither process can access what the other process is using. The control register ensures one program does not modify the other program's code by limiting each process' access privileges. Likewise, the memory management unit assigns each process its own virtual memory space or page to ensure no overlap between them.

4.

a. The optimal page replacement algorithm is a strategy used to decide which page needs to be replaced when there is a need for new one. In this algorithm, it will always choose to replace the page that will be needed last (chronologically) out of all of the pages. This makes sense on paper but does not work in practice. It can only work if the operating system knows ahead of time which pages of memory will be needed and in what order. Therefore, it will never be optimal. It also needs the memory accessing to be consistent

during each time it runs, so that the pages can be swapped in the correct fashion. This is also not practical; and therefore, why it is not used in practice.

b. Spatial locality refers to the proximity of data to the most recently used data in memory. It means that the neighboring addresses have a high chance of being called upon. Temporal locality means that recently used instructions/data are more likely to be reused. These are saved for later in quick-to-access storage (cache), so it can be called again quickly. Temporal locality improves the performance of the LRU algorithm by keeping track of what was recently used. The LRU would obviously want to avoid removing recently used pages. The LRU would also want to avoid replacing pages that are spatially located, or close to recently used data. If the program were to enter a loop, it would inefficient to swap out memory that is about to be used in a loop. Using temporal and spatial locality would improve the performance by making sure the LRU algorithm didn't just blindly choose the least recently used page.

c. The dirty bit basically just tells you whether or not a page of memory has been modified. If the bit = 0, then the page is untouched. If the bit = 1, it has been modified. This can help the Least Recently Used algorithm decide which page to replace. If the dirty bit is 0, then the algorithm should choose that page to replace over another comparable page that has been modified. This is because replacing a page that has been modified requires that the page be wrote back to the disk for later use (and system calls are slow). This means that replacing a clean page is faster than replacing a modified page. Therefore, using a dirty bit with the LRU algorithm increases its performance.

5.

```
typedef struct _rwlock_t {
        sem_t grouplock; // allow either readers or writers
        int writers; // writers in critical section
        int readers; // readers “      “      “
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
```

```
        rw->writers = 0;
        rw->readers = 0;
        sem_init(&rw->grouplock, 0, 1);
}

void rwlock_acquire_lock_reader(rwlock_t *rw) {
        if (rw->writers == 0)                    // make sure no writers have group lock
                rw->readers++;                   // add to number of readers
                sem_wait(&rw->grouplock);  // acquire group lock for reader
}

void rwlock_release_lock_reader(rwlock_t *rw) {
        rw->readers--;  // decrement count of readers
        sem_post(&rw->lock); // release group lock
}

void rwlock_acquire_lock_writer(rwlock_t *rw) {
        if (rw->readers == 0)     //  make sure the readers don't have the mutex
                rw->writers++;   // increment writer number
                sem_wait(&rw->grouplock);  // acquire group lock for writer
}

void rwlock_release_lock_writer(rwlock_t *rw) {
        rw->writers--;
        sem_post(&rw->grouplock); // let go of mutex
}
```

6.

a. Inodes blocks are basically chunks of data that describe a file: the type (file or directory), size, address on disk, time information, and permission information. Notice how

pathname is not one of the pieces of data found in the inode block. This is how you can have multiple files that actually point to the same inode block (links). Hence, they are the same file.

b. All inode blocks are the same size. So, if the file is under a certain size, the metadata for a file can be stored in just one block. This means that opening a 10 byte file should take roughly the same amount of time as opening a 100 byte file. If a file is too big for one inode block, then the first block will be used to point to a second, indirect block (which will then point to a third block and so on, if necessary). This means that the inode tree system is actually geared toward handling small files primarily (because they are more common). And thus, it adjusts accordingly for larger files.

c. A file system can use read ahead to speed up the sequential reading of files by first determining if the file can be read sequentially. If so, then it will retrieve the first block for reading and request the next block to be read next. It does this over and over again until there is no more reading to be done. The reason this is faster is because by the time the reading of a certain block is finished, the next one is ready to read immediately. The subsequent data is requested before the operating system has even finished reading the block before it. This is known as “prefetching.”