Swasti Mishra

Dr. Beck

COSC 361

13 December 2022

COSC 361 Final

1. **Consider two processors that are identical except that one has an MMU that uses a single-level page table and the other has an MMU with a two-level page table. The Translation Lookaside Buffer (TLB) works identically in both cases. The same program is run with the same input on each.**

   a. **On which processor would the program be expected to run faster, and why?**

   The program will run faster on the single-level page table MMU than the two-level page table MMU. This is because in the two-level page table MMU, the data the program actually needs to access isn't one level away- the operating system has to ask for the outermost page directory, and then find the page table entry within the next level. Navigating multiple levels will make the program slower. Another way to put it is that a single-level page table simply has to load from memory once. The two-level page table will have to load from memory twice.

   b. What is the principal advantage of using a two-level page table?

   Even though one-level page table MMUs are faster, there are advantages to using a two-level page table. The main advantage is that the two-level page table MMU will save more memory. This is because in the single-level page table, all the data must be loaded and available for access. However, in the two-level page table, the operating system can load

another free page when required to increase the table. Also, with a page directory, the actual pages in the table can be placed anywhere there is free space in the memory. In a single-level page table, all the pages must be contiguous.

c. Explain why a switch between user processes results in a temporarily lowered hit rate in the TLB.

A switch between user processes will result in a lowered hit rate in the TLB because the TLB is flushed between process switches. A flush happens by setting the valid bit of each entry in the buffer to zero. This happens for two reasons: first, the cache's entries are of no use to the next process, and second, if the TLB wasn't flushed, the next process could snoop on whatever entries were accessed by the first process. The reason this results in a lowered hit rate is that the TLB is a cache for storing virtual memory to physical memory. When flushed, it is empty. Because it is empty, there are no matches to recent memory and therefore the next few searches will come up as misses. After this temporarily lowered hit rate, however, the TLB will be repopulated and go back to functioning as before.

d. What is Temporal Locality and how does it impact the effectiveness of the TLB?

Temporal locality is when programs use specific pieces of data and resources again and again within a small span of time. The whole point of a TLB is to be a faster way to access data that has just recently been referenced. This happens by updating the TLB when there is a miss, meaning that the data is in the high-speed cache, easy to access for reuse and ready for a cache hit. A cache hit because of high temporal locality means that the TLB is working effectively, whereas low temporal locality increases misses and means that the TLB is not working effectively.

2. **Compare lottery scheduling and stride scheduling in detail, giving a summary of the implementation of each. Address the different notion of fairness implemented by each and the differences between their implementation with regard to complexity and the amount of state that must be maintained.**

In lottery scheduling (proportional-share scheduling), an operating system assigns each process "tickets" depending on their priority, and then holds a "lottery" during each time slice. It does this by choosing a random number that falls between the first and total number of tickets, and then scheduling and running the process that corresponds to the number it chose.

In stride scheduling (deterministic fair-share scheduling), the operating system assigns each process a "stride". A stride is calculated by dividing a large number that is inversely proportional to the number of tickets by the number of tickets each process possesses. When a process is run, the operating system increments a timer or pass value by the length of the stride. The scheduler then runs the process with the lowest pass value. In this way, no process is run to completion in one go- when one process has a longer stride than another, the next process will be run before swapping back and eventually finishing the first.

This makes stride scheduling fairer than lottery scheduling. With lottery scheduling, the operating system will achieve the ticket proportions (priority) probabilistically over time, but with no guarantee. Especially in cases where the job length is short, the "average fairness" can be low. Longer jobs that require more time slices, however, usually yield a higher average fairness with lottery scheduling. With stride scheduling, there is always fairness built in. Each process is given the proportion of CPU time it requires based on the total stride. The cost for stride scheduling, however, is complexity. While with lottery scheduling, all the operating

system must manage is each processes' tickets and the total tickets, stride scheduling requires tracking virtual time on top of tickets, strides, and pass values.

3. **Explain how the Process Control Register and Memory Management are used in the following operating system functions:**

   a. **Protecting resources such as a storage device or a network interface**

      The process control register (PCR) is used for controlling the behavior of hardware. In a resource like a storage device, the PCR would be used to protect from memory leaks by managing the requests from machines. In a storage device, the control register's ability to enable memory paging is important in protecting against memory leaks. While a small program may run fine, a large program with memory leaks can crash a device. Process control registers have some control in these situations, but effective memory management techniques have more.

   b. **Transition from user to kernel execution mode**

      If a program running in user mode requires kernel mode, the process control register (PCR) is used to enable a process's ability to request interrupts. Essentially, if the process is allowed to request an interrupt, the processor status register (PSR) will swap to 1 through use of the PCR and program will be interrupted. This interruption is important- it allows each of the three system layers to be protected from one another while ensuring that they can still work together. The three layers are 1) the program, running in a protected mode with access to virtual memory and paging, 2) the operating system, running in a privileged mode with access to the memory spaces of all programs, and the hardware which is keeping the code that is running in protected mode from running in privileged mode.

c. **Isolating user processes from one another**

The process control register controls interrupts, and therefore is very important in isolating user processes from each other. User processes cannot access privileged mode without permissions granted through the PCR, which is the only mode with access to every process's memory. This is where memory management, as handled by the operating system comes into play. The operating system also manages memory between isolated process with schedulers and ensures that data is still there when a process is switched back to, and that the memory from one process isn't touching the memory of other processes. This occurs when the PCR checks the data and stores it between processes.

4. **Explain…**

a. **What the optimal page replacement algorithm (OPT) is and why it is not used in practice.**

The optimal page replacement algorithm is a theoretical model for how a perfect operating system could work. In this algorithm, the page that will be used farthest in the future is swapped out for whatever the newest entry is. The reason this algorithm is not used is because it requires that an operating system is clairvoyant, which is obviously impossible.

b. **What temporal and spatial locality are and why and how they improve the performance obtained using Least Recently Used (LRU) page replacement algorithm.**

Temporal locality is when programs use specific pieces of data and resources again and again within a small span of time. Spatial locality is when a program reuses data or resources that are stored close together on the hardware. The Least Recently Used (LRU) page replacement algorithm takes advantage of temporal locality by design- if a page has

not been used recently, it is replaced from the cache. Further, by storing recently used pieces of data close together in the hardware (i.e., spatial locality), LRU can be further optimized because the operating system has to "travel" less far to access relevant information.

c. What the dirty bit is and how it can be used to improve the performance of LRU approximation in page replacement.

The "dirty bit" is a bit that tells the operating system where a block of memory has been modified. This bit is updated when the operating system writes to a piece of memory, which tells the operating system that the changes must be saved to storage. Further, when the operating system is considering replacing a piece of memory, it checks the dirty bit first to see if it needs to save the changes before removing it. In terms of improving the performance of LRU by using dirty bits, we can first examine a page for potential replacement, check if its dirty bit is set, and then store the page before doing anything else with it. If the bit is clean, we can go ahead and not write it to memory before replacing it. If we didn't have dirty bits, we would have to write every page into memory before replacing them, which is less efficient.

5. In a reader-writer lock a group of readers can share the lock, but a writer cannot share the lock with another writer or with a reader. A group lock is a variant of read-writer locks in which a group of writers can share the lock with each other, but not with a reader. Modify the reader-writer lock code in Figure 31.13 of Operating Systems in Three Easy Pieces to implement a group lock.

```
typedef struct _rwlock_t {
  sem_t lock; // binary semaphore (basic lock)
  int readers; // #readers in critical section
```

```
   int writers ;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
   rw -> writers = 0 ;
   rw->readers = 0;
   sem_init(&rw->writelock, 0, 1);
}
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->writers == 1){
    sem_wait(&rw->writelock);
   }
  sem_post(&rw->lock);
}
rw->readers++;
  sem_post(&rw->lock) ;
}
void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  if (rw->writers = 1){
    sem_post(&rw->writelock);
     return();
   }
   rw->readers--;
   sem_post(&rw->writelock);
}
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
   if(rw->readers>=1){
    sem_post(&rw->writelock);
    return();
  }
   rw->writers++;
   sem_post(&rw->writelock);
}
void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
   if(rw->readers >= 1){
    sem_post(&rw->writelock) ;
    return();
  }
  rw->writers-- ;
  sem_post(&rw->writelock);
}
```

6. **File systems**

   a. **Explain the role of inode blocks in the implementation of a file.**

   Inode blocks contain the metadata for a file. This can include a lot of different kinds of information, including the type of file, the size, its permissions, when it was last accessed, and other variables. When files are put on the disk, the inode blocks tell the operating system where the actual data for the file is using indirect and double indirect block pointers. Depending on the size of a file, it may need multiple data blocks, which are all kept track of with the inode and block pointers. In this way, files are saved as trees rather than one contiguous piece of memory.

   b. **How does the structure of an inode tree enable small files small files to be implemented with low overhead but also allow files to grow very large?**

   Inode trees utilize a dynamic tree structure and pointers, meaning that they manage small and large files similarly. For files that start (or are) small, the corresponding data blocks point to the root inode block, which has a small overhead because they only require direct pointers. When the file gets larger, indirect and double indirect blocks are used to store data and are then pointed back to the root node in a system called the multi-level index. All of these indirect pointers have a larger overhead, but still allow files the capacity to grow without getting lost in memory or causing storage problems.

   c. **Explain how a file system can use read-ahead to speed up the sequential reading of files.**

   Read-ahead strategy is the concept that the next bytes will most likely be read after the current bytes, or at least soon. Using a buffer cache, a file system can manage data blocks with the assumption that the next block in the file will be read after the current block. This

speeds up the sequential reading of files because the file system can “read” the next data block while it is still processing the current data block. The operating system knows what the next data block is thanks to inodes. Read-ahead strategy also becomes useful for data file situations where memory pages and file system pages have to be merged into one page cache. By caching all of these pages together, system calls for each page can be skipped, expediting the opening, reading, and closing process.