CS422/CS522 Final Exam

May 11th, 2007

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately 150 minutes to complete this exam, plus the traditional extra half hour to complete work already started.

1 Buffer allocation (20 points)

A graphics processor has n frame buffers that can be used to carry out various graphics operations. Each operation requires the exclusive use of either one or two frame buffers for the duration of the operation (which is always finite). Suppose that you have a large number of processes that each want to carry out a sequence of 1-buffer and 2-buffer operations, interspersed with other work for which no buffers are needed.

1. Describe a mechanism for allocating buffers to processes that ensures that no two processes ever use the same buffer at the same time while making efficient use of buffers. You may assume you have available standard synchronization primitives such as mutexes, condition variables, and semaphores.

2. Does your mechanism allow for (a) deadlock, (b) starvation, or (c) processes waiting even though enough buffers are available? If it does, is there a corresponding tradeoff that justifies these properties? If it doesn't, what properties of the underlying synchronization primitives do you need to assume to avoid these problems?

Solution

There are several ways to do this, depending on what properties you want. We'll concentrate on the problem of delaying processes when not enough buffers are available; the problem of actually keeping track of which buffers are allocated to which processes can be handled with the usual data-structure-with-a-mutex approach.

1. Single mutex for all buffers. This is deadlock- and starvation-free, but doesn't use the buffers very efficiently. The only real selling point is simplicity. We can do better.

2. Semaphore. Use a semaphore to allocate buffers, with the value of the semaphore equal to the number of unallocated buffers. There is a minor complication with 2-buffer operations; if two processes P and Q both want 2 buffers we may get at least a temporary deadlock if P and Q each down the semaphore once starting from 2. The solution is to wrap the down operation on the semaphore in a mutex, so that a process can only execute down if it holds the mutex.

   If the lock is fair, this solution is both deadlock- and starvation-free, since any process that is waiting is either holding the lock and waiting for the semaphore, in which case it proceeds as soon as enough operations complete, or is waiting for the lock. Since each lock-holder eventually downs the semaphore enough times and gives up the lock, any process waiting for it eventually gets in. (If the lock allows for starvation then we may get starvation.)

   There is a possibility that this solution allows for 1 buffer to go unused even though there is a 1-buffer operation waiting; this can occur if a 2-buffer process is holding the lock. Merely allowing 1-buffer processes to skip the lock doesn't prevent this problem, since the 2-buffer process might have already downed the semaphore.

3. Multiple queues with condition variables. If we really care about full utilization, we can create separate queues for 2-buffer operations and 1-buffer operations. When process arrives, it locks a global mutex to examine the state of the system; if there are enough free buffers, it grabs as many as it needs (marking them as in use via the state variables) and unlocks the mutex. If there are not enough buffers, it puts itself on the 1-buffer or 2-buffer queue and then waits on the corresponding condition variable.

   When a process releases a buffer (or buffers), it locks the global mutex, updates the state, and then (a) signals the 2-buffer condition variable if some process is waiting for two buffers and at least two buffers are available, or (b) signals the 1-buffer condition variable otherwise.

   This solution is deadlock-free and never leaves a buffer unused while there are processes waiting. But it does allows starvation of 2-buffer operations if there are enough 1-buffer operations to saturate the system. There is an inherent trade-off here: if we want to avoid starvation of 2-buffer operations, we must accept the possibility of holding a buffer idle waiting for a second buffer to become available. (However, we could have a more complicated scheduler with priorities for old

2-buffer operations that allowed tuning parameters to control exactly how much we trade off delayed operations vs. occasional underutilization.)

2 Paging with infinite RAM (20 points)

An unbelievable breakthrough in quantum-mechanical storage technology allows us to build memory chips that store 2^{128} words of data in the superimposed quantum states of a single electron. Because we can only read and write 2^{40} words per second, this means that our memory is effectively unlimited.

A CPU manufacturer proposes including this device on-chip and eliminating all caching and virtual memory, replacing the translation lookaside buffer and MMU with a simple pair of base and bound registers. Describe three common operating system features that would be significantly more expensive, difficult, or impossible to implement if this change were made.

Solution

We are looking for places where the base and bound registers don't provide as much protection as a full paging system or hacks where the kernel uses the VM system to intercept memory operations. Some possibilities:

1. Protection of code segments against writing or data segments against execution.
2. Shared libraries.
3. Shared pages used for IPC.
4. Shared pages with copy-on-write (e.g. for efficient implementation of `fork`).
5. Memory-mapped files.
6. Virtualization for CPUs with sensitive but unprivileged operations, where we use the memory system to catch execution of pages we haven't rewritten or reads of pages we have.
7. Debugging based on trapping memory accesses.

Note that in each case we can use the bound register to mark off the region we want to protect, but it's a crude tool; every memory access to an address higher than the protected region would also produce a trap.

3 A hash-based filesystem (20 points)

A filesystem implementer proposes replacing the index structures of a traditional filesystem with hashing. In this scheme, every block on the disk is assigned a number in the range 0 to $N-1$. Each stored block contains a header structure that specifies (a) the unique name of the file it belongs to, if any; and (b) the position of the block in the file. To read or write the i-th block of file f, we look at block $h(f,i)$ on disk, where h is a *hash function* that acts like a random function. If block $h(f,i)$ is the wrong block (it belongs to some other file or is in the wrong position for the desired file), we look at blocks $h(f,i)+1$, $h(f,i)+2$, etc. until we find the one we are looking for, wrapping around from $N-1$ to 0. No other structures are stored on the disk except for the actual file contents.

Describe the advantages and disadvantages of this scheme compared to a traditional filesystem. For the disadvantages, explain whether they are fixable with small changes to the system or whether they are inherent in the choice of using hashing.

Solution

Advantages

1. Many file operations require fewer disk operations, especially when the disk is mostly empty. For example, creating a new small file can require as little as one block read (to see if the hashed block is unused) and one block write. Creating a new small file in a traditional filesystem is likely to require updating a free list, a directory, an inode, etc.

2. It's almost impossible to create inconsistencies, since there are no data structures to keep consistent. The one place where we have to be careful is in deleting blocks, where it may be necessary to fill in blocks from the end of the chain to avoid losing blocks should be hashed earlier but were pushed out by the block we are getting rid of. Even in this case we have enough information in the blocks themselves to recover the correct structure at through mostly local scanning.

Disadvantages

1. Space overhead for filenames. For large files, keeping the filename in each block is going to be more expensive than storing it once in a

directory somewhere. This is somewhat fixable by assigning shorter unique IDs (e.g., collision-free hashes of the actual filenames).

2. Space overhead for empty blocks. We need a lot of empty blocks to keep the hash chains from growing too long. No easy fix for this.

3. Internal fragmentation. Every file is scattered across the entire disk, so there is no locality to exploit in reading multiple blocks from the same file. No fix.

4. No directories! How do we find out what files exist? Fixable: Add directories. Cost: Lose some of the performance benefits.

5. No inodes! How do we keep track of what properties a file has? Fixable: Add inodes (possibly as header structure in block 0). Cost: Lose more performance benefits.

6. No block lists! How do we know what blocks a file contains without probing for them? Fixable: Add block lists. Cost: We just finished recreating the traditional filesystem, except without the locality. Alternative fix: Insist that files not contain internal holes, and use binary search (or just an entry in the header structure in block 0) to find the last block. Cost: Have to enforce contiguity.

4 NMS: The Network Mutex System (20 points)

Consider the usual mutex interface, where we have procedures `mutex_lock` and `mutex_unlock` with the usual property that `mutex_lock` will block for all but the first process that calls it until that process calls `mutex_unlock`. Suppose that you want to implement these procedures on a cluster of machines that communicate via UDP packets. How would you do so? What information would you include in each packet and where would it be sent? What information needs to be stored in each process? How would you deal with lost packets? With delayed packets? Does your solution guarantee fairness?

You may assume for this problem that no machines crash and that if some machine A keeps sending packets to another machine B, eventually one of them gets through. You may also assume, if it makes things easier, that each machine has a clock that never runs backward, although it may not show the same time as other machines' clocks.

Solution

Here is a solution that uses the clocks to avoid problems with duplicate packets. We create a lock server process on one of the machines. When a process calls `mutex_lock`, it sends a lock-request packet from a local UDP port containing the current timestamp. It continues to send lock-request packets after a short timeout until it receives a lock-granted message back from the server. When it calls `mutex_unlock`, it sends lock-release messages (from the same port and with the same timestamp as the previous lock-request messags) to the server until it receives a lock-release-acknowledged message.

The server keeps track of (a) which IP address, timestamp, and port it has granted the lock to (if any); and (b) the highest timestamp for each IP address and port for which it has *released* the lock. It sends lock-granted only if (a) is null and (b) does not contain a timestamp greater than or equal to the timestamp in the lock-request message (this prevents regranting a lock in response to a delayed message from a process that has already released it, but does not prevent resending lock-granted to a process that has not done so). If the server receives a lock-release message, it sets (a) to null and updates (b) if the address and timestamp match, and responds with lock-release-acknowledged in all cases.

In order to correctly implement a mutex, it must be the case that no two processes can acquire the lock at the same time. Observe that if the server sends lock-granted to A and then B, it must have received a lock-release message from A before granting the lock to B. This can only occur if A has already called `mutex_unlock`; so even if A has not yet received the acknowledgment, it has already given up the lock.

We also want to avoid deadlock. The bad outcome is if the server has granted a lock but never releases it. But this can only occur if the grantee does not call `mutex_unlock`; otherwise the grantee is either still in `mutex_lock` (in which case it eventually sends enough lock-requests that some response gets through and gets it out), or it has called `mutex_unlock` and will continue to send lock-release messages until the server responds (at which point it must have already unlocked its local copy of the lock).

Note that this solution is not even remotely fair. Nothing prevents the server from handing out the lock to the same process over and over again. A more sophisticated approach that would prevent starvation would be for the server to put unfulfilled lock-request messages into a queue, and respond to them in order as the lock becomes available.