CS 3204 Sample Final Exam

Solutions are shown in this style. This final exam was given Fall 2006.

1 Anecdotes and Flame Wars (14 pts)

a) (6 pts) Raymond Chen relates the following anecdote in his blog "The Old New Thing" in this entry from Dec 15, 2004:

> I was reminded of a meeting that took place between Intel and Microsoft over fifteen years ago. (…)
>
> Since Microsoft is one of Intel's biggest customers, their representatives often visit Microsoft to show off what their latest processor can do, lobby the kernel development team to support a new processor feature, and solicit feedback on what sort of features would be most useful to add.
>
> At this meeting, the Intel representatives asked, "So if you could ask for only one thing to be made faster, what would it be?"
>
> Without hesitation, one of the lead kernel developers replied, "Speed up faulting on an invalid instruction."
>
> The Intel half of the room burst out laughing. "Oh, you Microsoft engineers are so funny!" And so the meeting ended with a cute little joke.
>
> After returning to their labs, the Intel engineers ran profiles against the Windows kernel and lo and behold, they discovered that Windows spent a lot of its time dispatching invalid instruction exceptions. How absurd!

i. (2 pts) Why did Intel's engineers think Microsoft's lead developer was joking?

Normally, faulting on an invalid instruction is an uncommon occurrence – as such, it is not something for which architecture designers should optimize.

ii. (4 pts) Give a reasonable explanation for why that version of Windows spent so much time dispatching invalid instruction exceptions!

It turned out that for that processor, it was the fastest way to force a mode switch, so Microsoft used an invalid instruction as a system call trap.

b) (8 pts) Here is an excerpt from the now famous flame war between Linus Torvalds, the inventor of Linux, and Andy Tanenbaum, professor of computer science at Vrije University, which took place in 1992. In this

reply, which is part of a longer exchange, Torvalds criticizes Tanenbaum for the lack of multithreading in Tanenbaum's Minix OS:

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Subject: Re: LINUX is obsolete
Date: 29 Jan 92 23:14:26 GMT

```
If I had made an OS that had problems with a multithreading filesystem, I
wouldn't be so fast to condemn others: in fact, I'd do my damndest to
make others forget about the fiasco.
```

From: ast@cs.vu.nl (Andy Tanenbaum)
Subject: Re: LINUX is obsolete
Date: 30 Jan 92 13:44:34 GMT

```
A multithreaded file system is only a performance hack.  When there is
only one job active, the normal case on a small PC, it buys you nothing
and adds complexity to the code.  On machines fast enough to support
multiple users, you probably have enough buffer cache to insure a high
cache hit rate, in which case multithreading also buys you nothing.  It
is only a win when there are multiple processes actually doing real disk
I/O.  Whether it is worth making the system more complicated for this
case is at least debatable.
```

i. (4 pts) Explain why Tanenbaum claims that a multithreaded file system doesn't buy anything if there's only one job active!

A multithreaded file system allows multiple threads to execute quasi-simultaneously inside the file system code, thereby allowing them to issue disk I/O requests in parallel. Doing so increases the I/O device utilization and allows for better disk scheduling. If there's only one job active, then there can't be multiple threads inside the file system code, for obvious reasons.

ii. (4 pts) Give one reason why modern OS such as Linux or Windows ended up using multithreaded file system implementations nonetheless, even on PCs used primarily by a single user.

It turns out that today's PC did end up having multiple, simultaneously active jobs after all: desktop programs use multiple threads, and several programs can be active at once (say Google desktop indexing your files while your browser is playing a flash animation while mp3's are played.)

2 Virtual Memory (22 pts)

a) (4 pts) Why is exact least-recently used (LRU) not a practicable page replacement strategy for modern virtual memory systems?

An exact implementation of LRU would require updating a queue or stack data structure on every memory access. Since every instruction involves a memory access (since the instruction itself must be fetched from memory), such updates would have to happen at pipeline speed. Especially on modern processors, this is not feasible.

b) (10 pts) Most of you implemented a simple 1-bit clock algorithm in project 3. The clock algorithm is said to approximate an LRU replacement policy. As some of you noticed, for tests such as page-merge-seq, the first page chosen by this algorithm to be evicted to swap turned out to be a stack page.

i. (4 pts) In making this choice, did the clock algorithm make a good decision? Justify your answer!

Both yes and no were accepted answers, if properly justified.
No: stack pages in general are likely to be reused again in the near future and as such are generally not good candidates for eviction.
Yes: because in this particular test case, this one was a stack page that belonged to the page-merge-seq parent process, which was suspended at the time the child process exhausted physical memory. This process won't be run until the child that caused page eviction had finished, so evicting its pages, even its stack pages, may be a good decision that leaves more physical memory for the child. I also accepted if you implied that a page near the top of the stack would remain unused until the program's execution unwinds to that stack page.

ii. (6 pts) Explain how the clock algorithm arrived at the decision it did!

*In a demand-paged system such as Pintos where pages are only brought in when accessed, *every* frame will have the accessed bit set when the clock algorithm is invoked for the first time. The algorithm iterates through all frames, resets the accessed bits of their associated page table entries, and chooses the first frame it encounters on the second revolution of the clock hand, which is the page frame that was allocated first. In other words, in the absence of access bits that provide an estimate of how recently a page was accessed, the clock algorithm defaults to FIFO! That is why it picked the page whose frame was installed first, which happened to be a stack page because the stack is the very first page a process installs when it starts (in setup_stack()).*

c) (4 pts) Suppose a process is thrashing in a system that uses a local page replacement policy. Would a scheduler such as the BSD4.4 scheduler you implemented in project 1 assign a high or a low dynamic priority to that process? Justify your answer!

It would be given a high priority since it is highly I/O bound. It will not use very many CPU ticks since it spends most of its time blocked waiting for a page fault to be serviced. Despite its name, thrashing is a phenomenon that is characterized by low CPU utilization.

d) (2 + 2 pts) Why does an OS for the x86 such as Pintos need to invalidate the TLB when evicting a page (i.e., in pagedir_clear_page()), but not when installing a page (i.e., in pagedir_set_page()) during a page fault?

When a page is evicted, the TLB may have cached that page's virtual-to-physical mapping. To avoid using a stale mapping, that entry must be invalidated. Unfortunately, on the x86, this implies the entire TLB must be invalidated.
If, on the other hand, a new page is installed during a page fault, there can be no stale mapping in the TLB (after all, if there had been a mapping, no page fault would have occurred!), so there is no need to invalidate anything.

3 File Systems (28 pts)

a) (10 pts) Consider multilevel indices as used in the original Unix file system and in Berkeley's FFS. Many of you chose a variant of them to implement project 4.

i. (1 pts) Consider a multilevel index as an example of an abstract *map* data type that maps keys to values, as discussed in your data structure classes. What are the *keys* in this map?

ii. (1 pts) What are the *values* in this map?

The multi-level indices are used to find which sector on disk contains a particular block of a file, so the keys are the file offsets (counted in sector-sized blocks), and the values are the actual on-disk sector numbers.

iii. (2 + 2 pts) Multilevel indices in this use are much simpler than other implementations of maps, including many types of k-ary search trees with which you are familiar. Give two reasons why.

Answer I accepted included:

- *The height of the tree is fixed, there is no rebalancing.*
- *The fan-out of each node is known a priori.*
- *There's no removal operation that needs to be supported.*
- *The set of keys is fixed from a known, continuous set, so no custom comparator must be supported.*
- *All interior nodes and the root have a similar or identical structure internally.*
- *Small files involve a tree with only 1 node, the root.*

iv. (4 pts) Multilevel index-based file systems do not suffer from external fragmentation. Explain briefly what that means and why that is the case.

It means that a file system using those indices will never have to reject an allocation request while there are still free sectors on disk – any sector can be used to extend a file. This property holds because a multi-level index does not require any contiguous area on disk.

b) (4 pts) Recent file systems such as XFS and ReiserFS use a technique called *delayed allocation*, in which the on-disk position (i.e., the sector number) of a newly written file block is not decided until that block is flushed from the buffer cache to disk.
Why can this approach lead to better performance?

A key performance advantage is to read sequentially from continuous sectors whenever possible. Most current OS use preallocation mechanisms to decide how much continuous space to allocate for a file. These mechanisms may over or underestimate how many continuous sectors are needed. If you delay the allocation decision until flush time, you (usually) have more accurate information about how many continuous sectors you'll need.

c) (4 pts) Consistency. Give one example of an unacceptable inconsistency that FFS's design (or any of the newer journal or write-ordering based file systems) prevents!

Any of the inconsistencies discussed in class were accepted here. For instance, FFS ensure that a user won't see another user's data after a crash, because it first persistently nullifies pointers to data before reusing the data. Another example includes that if a crash occurs while a file is renamed, the file won't be lost – in the worst case, the old and the new name will both appear for the same file.

d) (10 pts) Suppose you added ACLs to Pintos to allow or restrict read, write, or execute permissions for ordinary files (ignore directories). Assume that users would authenticate using a user id, and that each process carries the user id under which it executes in its PCB.

i. (4 pts) Which on-disk data structures would you change and how?

An ACL in this case is a list or array of (user id, rwx) pairs were rwx would be a 3-bit mask. A small number of such entries could be stored in the on-disk inode; for larger ACLs, you could store a sector number to additional ACL blocks, or even an inode number if you stored the ACLs in a file. Combinations are possible. It's also possible to simply store a reference to an entry in a global ACL file.

ii. (3 pts) Which in-memory data structures in your file system code would change, if any? If none changed, say why not.

Acceptable answers are either none, or the in-memory inode structure. The trade-off is one of memory vs. access speed vs. consistency. If you replicate the ACL in the in-memory inode, you may avoid disk accesses to retrieve it, but you need more memory and have the added task of writing changes back to disk. If you don't, you save the memory, and the consistency comes for free if you use

your buffer cache's dirty mechanism to update them. In addition, for frequently accessed ACLs, it's likely that they would stay cached in the buffer cache.

iii. (3 pts) Where in the kernel would you need to include permission checks to ensure that access permissions are enforced? Consider read, write, and execute.

Read should be enforced in inode_read_at, write in inode_write_at. Execute must be enforced in process_load(), which may require an additional interface to be exported by the inode layer.

4 Buffer Cache (20 pts)

a) (4 x 2 pts) Consider how your implementation of indexed and extensible files in Pintos uses the buffer cache. Assume that the buffer cache is correctly implemented and provides, for each block, the ability to gain shared or exclusive access and to release such access.
Can the way in which your file system issues requests for blocks lead to deadlock? Discuss each of the four necessary conditions for deadlock and state if they apply or not. State your assumptions if necessary!

The four necessary conditions for deadlock are:

Mutual Exclusion: since buffer cache blocks can be locked in "exclusive" mode, this condition applies. (It would not apply if all accesses were in "shared" mode.)

No Preemption: a buffer cache block that is locked cannot be forcefully unlocked and evicted, so this condition applies as well.

Hold and Wait: this condition (likely!) applies as well, because there are situations where you hold exclusive access to more than one buffer cache block, which you must have acquired in some sequence. For instance, when extending a file, you probably want the new block in exclusive access while also holding exclusive access to the index block into which you install a pointer to the new block.

Circular Wait: this condition would indicate a bug in your code. In a straightforward implementation of multilevel indices, there would be a natural locking order, which follows the index blocks down the tree.

b) (4 pts) Suppose you added to your buffer cache API a "cache_purge(disk_sector_t s)" operation. cache_purge(s) would ensure that sector s is no longer in the cache. If sector s is not in the cache at the time of invocation, it does nothing. If it was in the cache, then the block containing it is simply discarded (even if it was dirty) and subsequently

marked as unreserved.
If your buffer cache had such an operation, when would you use it?

You'd use it when a block is deallocated, such as when a file is removed. Doing so will avoid the eviction of other data that was less recently used, but which is still alive and could be accessed again in the future.

c) (8 pts) [Question deleted.]

5 Short Questions (16 pts)

d) (4 pts) What is address space randomization and what is its intended purpose? (Note that this is a two-part question!)

Address space randomization is a technique by which the virtual addresses of a user program's stack is randomly chosen from within a certain range at load time. Its intended purpose is to make stack overflow attacks more difficult, because those attacks are vastly simplified if the actual value of a user process's stack pointer is known.

e) (4 pts) Why do OSes usually impose a limit on the number of file descriptors a process can have open?

The primary reason is that file descriptors take up kernel memory, which is a shared resource that is not subject to per-process limits.

f) (4 pts) Aside from increased fault tolerance, name one other benefit of RAID configurations!

In addition to increased fault tolerance, RAID can improve latency and bandwidth of large reads and writes (RAID-0, RAID-4, RAID-5), and also the latency of small reads (RAID-1.)

g) (4 pts) Suppose you are hired as an embedded systems engineer and your team lead tells you that their OS provides pure user-level threading without preemption. What do you need to be careful of in the application code you write for this system?

You need to avoid long (or infinite) loops to avoid starving other threads, and you have to provide a means to cooperate when being asked to terminate. Depending on the system, you may also have to avoid making any blocking calls or risk blocking the entire process.