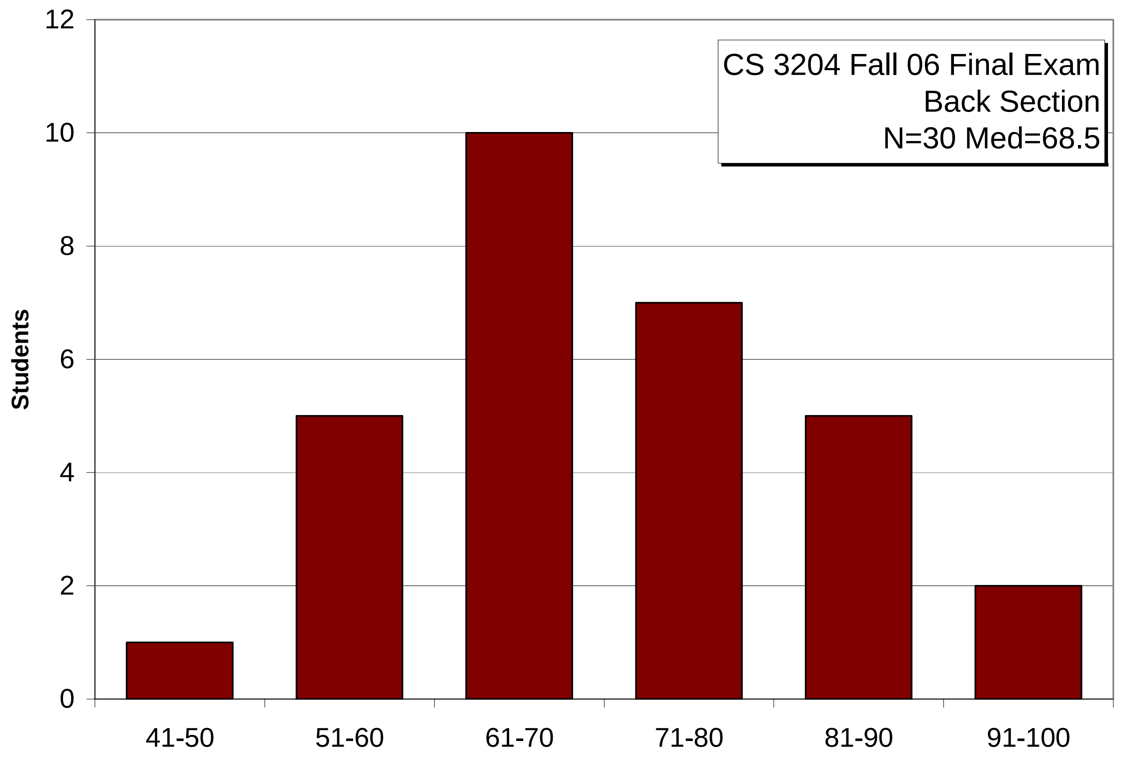
CS 3204 Final Exam Solution

30 Students took the final exam. The table below shows the scores for each problem. I was quite happy about the relatively high median and believe it reflects a real learning result. While grading the exam, I felt that most of you had achieved a basic understanding of the concepts being tested.

Problem	***1***	***2***	***3***	***4***	***5***	***Total***
Max	14	22	28	17	16	94
Min	5	7	9	3	4	48
Med	10	16	22.5	11	12	68.5
Avg	10.3	16.0	22.2	10.8	11.1	70.6
StdDev	2.3	3.0	5.1	3.6	4.2	11.7



Solutions are shown in this style.
Grading Comments are shown in this style.

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.

You should mention that an invalid instruction faults are rare, as such faults usually signal a condition that should not happen, and such rare events are not worth optimizing. Wrong answers included making vague statements such as "they thought faults were always a bad thing." Keep in mind that this meeting took place around 1989, which was more than 20 years after the invention of virtual memory (see the "RCA has it, IBM doesn't" ad I posted from 1971.) Engineers understood the differences between the different types of faults by 1989.

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.

Answering that question was easy if you'd studied the sample midterm, question 2a). Note that the problem states twice that the exception that was frequently dispatched was an invalid (or illegal) instruction exception, not a page fault exception. As such, it has nothing to do with virtual memory or page faults. Page faults are caused when an instruction such as a load attempts to access memory that's not present or to which it doesn't have access privileges, it does not make that instruction invalid. Even if illegal instructions faults had been faulting faster than page faults, it's not clear how they could have used them to implement virtual memory – on the other hand, it's straightforward how to use an illegal instruction to implement a system call trap. I gave 1 pt partial credit for recognizing that a page fault involves a mode switch still.
I also accepted if you speculated that Microsoft may have used illegal instructions to emulate binaries written for a previous architecture generation, although Intel's policy of keeping backward compatibility wouldn't have necessitated such an approach.

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 filesystem 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.)

Note that the question asked for a rationale for why OS designers ultimately decided to use multithreaded file systems even in desktop OS run on single-user PCs, despite the complexity pointed out by Tanenbaum.
For both parts i and ii, I was pretty generous in what answers I accepted, as long as they addressed the question and didn't include outright wrong information.

2 Virtual Memory (22 pts)

a) (4 pts) Why is exact least-recently used (LRU) not a practical 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.

This question had a typo: I meant to say "practicable", as in "feasible" or "possible." Many of you read the question as "is strict LRU a beneficial strategy in a recent system with a unified buffer & VM cache?" and gave a counter-example of where blindly applying LRU to a buffer cache does not give optimal replacement, such as large looping accesses or the case of a page that's accessed only once but is kept in the cache in preference to older data that is going to be used. If you correctly described one such scenario and correctly explained why LRU fails, I gave full credit.
Note that a pure sequential access is not a correct example – if access is sequential, it's all misses and the replacement policy doesn't matter. I should point out though that except for MIN, no replacement strategy is always optimal, but that doesn't necessarily make it impractical in the sense of losing any benefit.

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.

For full credit, you had to justify your answer and provide a reason.

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()).*

For full credit, you had to make it clear that you understood that Clock defaults to FIFO unless the accessed bits provide additional information. Note that the problem specifically talked about the "first" page chosen by the algorithm. At that point, the clock hand will not have advanced, so the Clock doesn't yet approximate LRU (at least not in the model you implemented in Pintos, where eviction only happens when a page is needed. If you've implemented a watermark-based trigger instead, where the clock starts running before you actually run out of page frames, you would have to state that in your answer.)
I gave 3 pts partial credit for properly explaining how clock works in general.

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.*

I gave 2 pts partial credit if you at least acknowledged that thrashing means low CPU utilization + high I/O utilization, without mentioning the consequent behavior of the scheduler.

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.

Wrong answers included that when installing a page, an "old mapping" would be overwritten or that the TLB caches only frequently used mappings. For a memory access to succeed, the mapping must be in the TLB (at least on the x86).

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*

These two questions were intended to test whether you understood the purpose of the multilevel indices you had implemented at an abstract level.
For part iii., I was looking for at least two different arguments.

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.

For full credit, you had to discuss what absence of external fragmentation is and you had to specifically mention why a multi-level indexed based structure avoids it.

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 issue is to read sequentially from continuous sector numbers whenever you can. 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.

Alternative answers that were accepted for full credit included arguing that delaying the allocation (and updating the freemap & index blocks) reduces the latency of an individual write as seen by the application, and arguing that delayed allocation would allow an allocation decision that is closer to where the disk head is located, eliminating seek time. The latter, however, is only a research idea at this point for single disks – it is used for filesystems that lie on RAID structures, however, in order to skirt the small-write problem.
I deducted one point if you also claimed that the disk space that is allocated in this delayed fashion could be used by other applications in the interim – this is not true, the system must fail a write if there's no space left. We can't speculate that there will be sufficient space by the time the write is going to disk, we must ensure this, for instance, by keeping a free space counter.

I gave no credit if you suggested that the reason is to save bringing in the metadata (index blocks etc.) – you have to examine the metadata before you can decide whether a new block needs to be allocated or not. It doesn't matter if you do it at eviction time or at write time as far as the number of disk accesses is concerned. (Unless you decided on a strategy, also possible, where each write that overwrites an entire block would lead to a reallocation of that block. You'd have to state it.)

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.

To get credit, your answer had to refer to a fault scenario were it makes sense to distinguish between inconsistencies from which one can recover and those that one must avoid by design.

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.

What I was looking for in these two problems was that you understood what is on disk vs. what is in memory, and when you'd access what. I deducted heavily if I sensed confusion about this issue.
An ACL is not simply Unix-style permissions for owner, group, and world. It's a list of user ids and their associated permissions.

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.

For the permission checks, note that checking in the system call handlers is insufficient – for instance, if a file is mmap'd, no system call would occur. In addition, at least in Pintos, you couldn't check in open() or mmap() because the process doesn't pass intended permissions. As these are Pintos-specific issues, I didn't deduct for them.
Note also that the problem specifically asked to address read, write, and execute, and that it specifically asked to say "Where in the kernel". I deducted if your answer didn't address these points.

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.

I gave 1 pts each simply for mentioning the conditions. Easy points if you studied. I gave an additional point for discussing whether this condition applies. Note that the problems asked specifically "Discuss each of the four conditions." I even indicated that the score would be 4x2.
For the circular wait, I was picky and gave the second point only if your discussion specifically referred to "your implementation of indexed and extensible files."
A common mistake was about the meaning of preemption. In this context, "preemption" would mean that access to a buffer cache block could be forcefully removed. It is unrelated to whether the CPU can be preempted.
A second mistake was regarding the meaning of "hold-and-wait." Here, it would mean to hold one resource (buffer cache block) and wait for another resource (e.g., block). It doesn't mean to hold a lock and then sleep for an indefinite amount of time. The latter would be a starvation condition, not a deadlock one.

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.

Some of you suggested it could be used as a kind of nullification mechanism by which a process's writes can be undone. I gave only partial credit for that, because even though that would indeed be the effect, such a mechanism would not be reliable – the block could have already been evicted to disk. I only accepted this for full credit if you made it clear that it would not be mechanism to be relied upon, but a mechanism that could limit damage that had already occurred. Also, several of you suggested that blocks written by a killed process don't have to be written to disk. That's generally not true.

c) (8 pts) One of the crucial properties one must ensure when implementing a buffer cache in a multi-threaded file system is that the I/O involved when evicting a single buffer cache block does not prevent other, independent operations from proceeding in parallel. Some groups used a “two-level locking” approach for this purpose, which relies on a global lock to protect the buffer cache array or list, and a per-block lock that protects an individual block’s sector number and internal variables. The per-block lock is held when a block is evicted, but the global lock is not. The eviction part of cache_get_block then looks like this:

```
Acquire global lock
Find block to be evicted
Acquire per-block lock
Remember previous sector number, set new sector number in block
Release global lock
Perform actual I/O that writes dirty data to previous sector
Release per-block lock
```

Note that the block must be rededicated to the new sector before releasing the global lock such that another thread that is seeking to access the same new sector will have to wait for the lock on the same to-be-evicted block (which is necessary to ensure that there is at most one block reserved for each sector.)

This approach is correct and thread-safe. However, it does not guarantee that *independent* operations on the buffer cache can proceed in parallel. Construct a scenario that demonstrates this! (Note: two threads seeking to access the same sector are not independent operations, so this scenario would not count.)

It turns out this question also had a bug, although none of you noticed it. It’s actually not correct to reset the sector number of the to-be-evicted block before evicting it, because then a thread trying to access the to-be-evicted block will not find it and mistaken believe it is on disk! However, the rest of the problem still holds.
*The scenario in which a thread has to wait for the I/O of an unrelated operation to complete is when one thread is trying to access a block that is being evicted. The first thread evicts the block and is blocked on I/O, while the second blocks on the per-block lock *while holding the global lock*. Now, any third thread arriving at that time will find the global lock held, preventing access to the buffer cache altogether. Although the operations of the first and second thread are not independent, the operation of the third thread is.*

Many of you answered that the problem is that the global lock is held while it examines the block list, which causes serialization even if two threads are attempting to access different blocks. This serialization, however, is unavoidable and required for correctness. It’s also not an issue on a uniprocessor since threads will not give up the CPU while examining the list. That is, the lock is not held while waiting for I/O, which is the key issue as the problem states in the first sentence. I gave 4 pts partial credit still.

Some of you replied that the thread trying to evict a block may be racing with a thread attempting to access the evicted block. This is true, but those are not independent operations and the interaction here is unavoidable. I gave 4 pts partial credit here, provided you talked about the thread trying to evict and the thread trying to access the evicted block. (The scenario were two threads are going for the same new block was already ruled out explicitly in the question.)

Another answer was that you said that just finding a block to evict involves locking every single block to examine its use status. This, however, is something you should have easily avoided by using lock_try_acquire, and in any event, it's not what the problem asks. I also gave 4 pts partial credit.

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.

Note that it's the stack location that's randomized, not the code location. However, I accepted code location as well because that is theoretically possible as well (for instance, if all executables were linked as position-independent code) and because my discussion in class may not have been clear on this issue.

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.

Some of you said "to avoid I/O overload", but this is only partially reasonable as having a file descriptor open is not the same as actually using it. I gave partial credit for that. I gave no credit for answers that suggested that file descriptors were kept track of in user memory, on the user stack, or similar. If that were the case, the OS doesn't need to limit them specifically.

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.)

“inexpensive” isn’t really a benefit anymore – these days, the I in RAID means independent. Today, multiple single disks are usually cheaper than a RAID array, because RAID arrays are composed of single disks + RAID controller.

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.

Although “user-level threading” technically doesn’t imply that calls are blocking (many ULT libraries provide an I/O veneer to avoid such blocking), I accepted it for full credit.
I gave partial credit for mentioning the danger of deadlock because of the lack of preemption. However, preemption here means preempting the CPU, not preempting a process’s access to a resource (see also problem 4a). An infinite loop is not typically deadlock, and accesses to such resources as locks and semaphores aren’t preemptible even in systems that support preempting the CPU.