CS 111 Final exam

Jeffrey Hsinping Xu

TOTAL POINTS

100 / 100

QUESTION 1

1 Question 1 50 / 50

✓ **- 0 pts** **Correct**

- 5 pts Need more discussion of reliability issues.

- 3 pts Sequential write not particularly expensive for flash. Cheaper than random writes, since random locations likely to require erase first.

- 3 pts Not necessary/helpful to virtualize BB DRAM, especially if using a log.

- 5 pts How will system/application/user determine which files go where?

- 2 pts How will you handle one device becoming full while the other still has space?

- 5 pts No discussion of metadata issues.

- 25 pts You didn't design a file system for battery backed DRAM. You used it to hold process data, which is NOT what I asked for.

- 15 pts No discussion of how flash file system would work.

- 5 pts With proper implementation, computer crash need not affect battery backed DRAM.

- 5 pts Flash storage is slower than DRAM, battery backed or not.

- 5 pts Using battery backed DRAM purely as a sanity check for flash is a tremendous waste of its possibilities.

- 3 pts Random writes are bad for flash, unless to an unused location.

- 10 pts Which is it, one file system implementation for two separate devices or using the BBDRAM as a cache for the flash (which implies a unified file system)?

- 2 pts Can handle circular log overwriting issue by keeping track of head and performing garbage collection.

- 2 pts Different block sizes will lead to complexities in the block I/O cache.

- 25 pts Incomplete answer.

- 7 pts Inconsistent discussion, contradictory in places. Poor performance choice of what to store where. Why use fast DRAM to store infrequently read files? DRAM is also good for frequently written files, while flash is good for infrequently written files.

- 15 pts How will you make good use of the particular characteristics of each device?

- 5 pts Are the mechanics of using metadata to build and access files going to be the same for both devices? Why or why not?

- 20 pts Lacking details of how the implementation would actually work.

- 7 pts BB DRAM is much quicker than flash and perfectly happy with random writes, which flash isn't. How can those characteristics be profitably used?

- 40 pts DOS FAT is a terrible choice. Not well suited to the characteristics of either device, imposes many artificial and unnecessary limitations, doesn't support file system options and features easily supportable by other approaches.

- 5 pts More details on cache management needed for this option.

- 2 pts Not clear that using 1/11 of your storage capacity for pure caching is a good idea.

- 2 pts Benefit of providing virtual address space for files in DRAM not clear. Adds complexity, for what benefit? Why not just use pages?

- 2 pts Need more details on automated methods of moving files into/out of flash.

- 5 pts DRAM has no seek time, so creating cylinder groups is unnecessary.

- 10 pts Insufficient discussion of how to handle flash memory issues, like single write and erase

cycles.

- 10 pts Flash memory holds data persistently without power. If it didn't, storing checksums would not help one bit, since they would be lost, too.

- 8 pts Insufficient details of how DRAM file system works.

- 3 pts Needs more discussion of how DRAM caching is organized.

- 3 pts Unless you fiddle with hardware architecture, BB DRAM can't be used as regular DRAM. It's a device, not word addressable.

- 8 pts EXt3/4 poorly suited for flash, due to single write/slow erase issues.

- 10 pts Unless file system specially designed to use it as cache, BBDRAM won't help with caching. It's a device, not main memory. You don't discuss how reads and writes would be sent through BBDRAM first.

- 10 pts This design will direct most reads to slower flash.

- 5 pts Desirable to have frequently read data in DRAM. Not clear how your design achieves that (other than metadata and directories).

- 15 pts This use of BBDRAM does not provide much advantage of its good characteristics, such as faster read/write performance and write-in-place.

- 10 pts Not clear how you are leveraging relative advantages of each device.

- 20 pts No discussion of flash implementation.

- 0 pts Unclear on

QUESTION 2

2 Question 2 **50 / 50**

✓ **- 0 pts Correct**

- 5 pts There are differences between scheduling cloud resources and all processes on a multicore system. How do those figure in?

- 10 pts How are bids calculated?

- 5 pts No discussion of kernel thread scheduling.

- 5 pts No discussion of real time scheduling.

- 3 pts No comparison to round robin.

- 3 pts No comparison to priority scheduling.

- 3 pts Just setting the bid to highest number of credits limits the flexibility of the approach, which is its main attraction.

- 2 pts If you only get more credits when lower priority process blocks you, low priority processes could starve. Discussion of this is inconsistent.

- 3 pts Kernel itself is not threaded. Kernel threads are process threads known about and scheduled by the kernel, as opposed to user level threads.

- 2 pts More details on real time comparison.

- 3 pts Can processes ever get more credits? If so, how? If not, what happens when a process uses up all its initial credits?

- 5 pts Why is set of memory pages allocated to a process relevant to bids?

- 5 pts What do you mean by "put in the process queue which can minimize average waiting time and maximize throughput?" Which queue is that? How is it determined?

- 3 pts Kernel threads are not required for this idea. How to support them if you have them?

- 10 pts How are credits assigned to processes?

- 3 pts Fairness issues?

- 5 pts More details on mechanics of making bids.

- 2 pts Not desirable to gain advantage by adding threads to your process.

- 3 pts More details on assigning credits, such as how to deal with priorities and fairness.

- 5 pts Can't work if processes are given fixed credits at start and never get more.

- 2 pts Do threads get separate allocation of credits or share the owning process' credits?

- 2 pts Your proposed method of adding credits should prevent starvation, though not necessarily guarantee fairness.

- 2 pts Issue of gaming the credit assignment procedure.

- 1 pts More details on priority scheduling comparison.

- 1 pts No comparison to hard real time.

- 2 pts Why should owning process provide bids for its kernel level threads?

- 2 pts Method of assigning/adjusting credits not clearly described.

- 1 pts Probably better to keep credits/bids in PCB rather than stack, since stack can get overwritten in some cases. Also, PCBs more readily accessible than stacks.

- 1 pts No comparison to soft real time.

Final Examination
Summer 2017
CS 111

Name: Jeffrey Xu

This is an open book, open note test. You may use electronic devices to take the test, but may not access the network during the test. You have two hours to complete it. Please remember to put your name on all sheets of your answers.

There are 2 questions on the test, each on a separate page, followed by several blank pages to hold your answers. You must answer both of them. Each problem is worth 50% of the total points on the test.

You must answer every part of each problem. Read each question CAREFULLY, make sure you understand EXACTLY what question is being asked and what type of answer is expected, and make sure that your answer clearly and directly responds to the asked question. If you do not answer part of the question YOU WILL lose points.

I am looking for depth of understanding and the ability to solve real problems. I want to see specific answers. Vague generalities will receive little or no credit (e.g., zero credit for an answer like "no, due to the relocation problem."). Superficial answers will not be sufficient on this exam.

Organize your thoughts before writing out the answer. If the correct part of your answer is buried under a mountain of words, I may have trouble finding it. Write your answers on the front of test pages only. Anything written on the back of pages will not be graded. If the space provided is insufficient for you answer, talk to me.

Jeffrey Xu

1. Consider a system that has a 1 Tbyte flash storage device and 100 Gbytes of battery-backed DRAM. Battery-backed DRAM is exactly like regular DRAM in its performance characteristics, but it is attached to its own battery (plus the general power supply), which guarantees that the memory will retain its state regardless of reboots, power failures, or other events that would ordinarily cause DRAM to lose its state. To be perfectly clear, the battery-backed DRAM is a secondary storage device, **not** the system's normal DRAM used for the ordinary purposes of DRAM. Should you design one file system implementation that is used to support an independent file system on each of these two secondary storage devices, or should you design a one file system implementation that integrates the two devices to support a single file system, or should you design two entirely independent file system implementations, one for each device? Why? Whichever answer you choose, describe the key design characteristics of the file system(s) you would build and explain how those choices fit into your rationale for your choice. Consider all issues that we discussed as important in file system design.

We should probably not use the same file system implementation for both the flash storage and the DRAM because doing so would either ignore the unique behavior of requiring a full-block erase to reprogram a page, causing failed writes to already programmed pages, or apply the same copy-on-write techniques to the DRAM, where they would cause unnecessary write amplification. Thus, option 1 is out. Option 3, on the other hand, would require the user to be aware of both file systems and make wise choices about which file system to put a file in depending on how much I/O is expected to be done on it. This option provides poor encapsulation. Thus, I favor option 2, designing one file system implementation that integrates the two devices to support a single file system because it allows the file system to exploit the different behavior of the two forms of persistent memory while also hiding the decisions of how to do so from the user. Note that this answer assumes the file system is custom-made for this system as if it needs to be reused, it may run into issues on machines without the same setup of flash and DRAM.

(continued on next page)

Jeffrey Xu

In order to minimize the cost of erasing and rewriting blocks in the flash storage, I would use a log-structured file system. In addition to take advantage of the DRAM's speed, I would use it for the more heavily trafficked parts of the file system, as well as caching recently or commonly used files. Because copying-on-write, as required by flash storage, regularly changes the location of data blocks, the inodes will all be stored in the DRAM. As inodes are small, the smaller size of the DRAM should not be an issue, and there is likely to be plenty of space left over to cache writes. The flash storage would hold the large majority of the data, but because the DRAM is not volatile, there is less of a rush to copy new writes from the cache in DRAM to the larger flash storage. In fact, each time data is written to a file, the block intended to be written to should be read from the flash storage (a much faster operation than writing) and copied into DRAM, and the corresponding block pointer in the inode can then be changed to point to the block's location in DRAM. As 32-bit block addresses allow 4 billion blocks, as long as block size is guaranteed to be at or above 1KB, the highest-order bit of the block address can be reserved to indicate whether the block is in DRAM or flash memory. Because writes to DRAM are so fast, this virtually eliminates consistency issues, as long as the data is copied into DRAM before the inode's block pointer is updated. As mentioned before, there is less of a rush to write updates in DRAM back to flash because there is no risk of losing data, so write-back can and likely should be delayed until either the file is closed or

(continued on next page)

the available space in DRAM falls below a certain threshold. Delaying write-back allows exploitation of locality of reference, allowing subsequent writes to not wait for data to be read from the flash. It also reduces the number of writes to flash, prolonging the lifespan of the flash device. Furthermore, if some file is always very frequently being written to, such as an OS log of some kind, it may never be written back to flash. (I forgot to explain how inodes would be organized, so this part will be off-topic.) At the start of DRAM, there would be bootup data and a superblock indicating things like block size, inode size, and the number of inodes. This would be followed by a series of inode groups consisting of nothing but a fixed-size inode bitmap indicating which inodes are free in the group, and a number of fixed-size inodes equal to the number of bits in the bitmap. This prevents the bitmap from having to be arbitrarily large and allows the inode table to grow. (Back to explaining writes.) When a file is closed or DRAM space is falling low and some blocks have not been written to in a long time (determinable by a clock algorithm similar to that used for paging), then the block is written to a new place in flash, the block in DRAM is marked free in its bitmap, and the block pointer in the inode is updated. Because DRAM is so fast, the last two operations will complete at effectively the same time, but they must be started after the write to flash to prevent pointing to garbage or invalid data. Directories behave like regular files containing directory entries linking names to inodes.

As it is decentralized throughout this answer, here is a summary:

DRAM

not superblock (inode-free-bitmap inodes)×n clock pointer (block-free-bitmap block-dirty-bitmap block)×m

flash

Jeffrey Xu

2. You are part of a team working on a new operating system for multicore processors. One of your colleagues suggests a different model for scheduling, one based on bidding. Processes will be given credits (or perhaps be sold credits, or perhaps somehow earn credits – bright new ideas tend towards vagueness). They can use these credits to bid on processor time at the next scheduling decision. The highest.bidder spends its credits and gets the processor for either the time it bid for or for some standard time quantum (again, vagueness in the idea). Either a process can increase its bid at a later time, possibly pre-empting the previous winning process, or a process with a winning bid holds the core for the period it bid for regardless, leaving aside OS interrupts to handle critical system tasks, like handling interrupts (one more vague point).

   Is this a good idea? How would it work in practice? How would bids be calculated and registered by processes? How should kernel level threads play into this scheduling approach? Do they make their own bids or does the owning process bid for them? How would this scheduling method compare to other alternatives, such as round robin or priority based scheduling? Would it work well for real time (hard or soft)? Discuss the mentioned areas of vagueness in the idea, settling on an approach for each.

This sounds like a more granular version of priority, with a process' priority being replaced by its number of credits. It does have some advantages, though: there no longer needs to be a fixed number of priority levels, and if a process knows it will need a short response time in the future it can effectively lower its current priority by bidding less than it has, saving credits to ensure it wins a later bid. This idea of saving up priority does not exist in a priority scheduler. The mechanisms used for this can be used for round robin scheduling by having each process start with a different number of credits and giving each waiting process a credit at each timer interrupt. Because this system can provide round robin scheduling but can also be used in other ways, it is an upgrade over round robin scheduling.

Adjusting and registering bids would require system calls. There could be a number of different bidding policies, such as how much to bid relative to how many credits are owned. Bidding nothing would be equivalent to sleeping, for example.

(continued on next page)

Jeffrey Xu

8.

I propose that processes that are not currently running earn credits at each timer interrupt. This means that processes that run for a short time gain more credits, over a period of time than longer running (because they miss fewer opportunities to earn credits) ones, which fits the behavior of multi-level feedback queues. This allows us to avoid the question of how long to let a winning bidder run and simply say that when another process is able to bid over a certain threshold, it preempts the running process. To prevent this threshold from increasing without bound, everyone will lose some number of credits at each auction even if they lose.

2

To implement this idea, we could replace the waiting list of processes with a heap sorted by each process' current bid. At each scheduling decision, we can simply take the top of the heap.

5

Threads should make their own scheduling decisions, or else their owning process may bid high for a thread that is waiting for a lock and cannot progress.

This could work well for real time systems by having a process gain an increasing number of credits as its deadline approaches. However, in a hard real time system, a job would need to gain a virtually infinite amount of credits just before it becomes too late to finish the job if it is started now, and if the runtime of each job is known well enough to do so, it would be better to use a non-preemptive scheduler to avoid the overhead of timer interrupts. In a soft real time system, bidding works well to spread work across multiple jobs when there are no near deadlines and switch to focusing on individual jobs as their deadlines draw near.

(continued on next page)

Jeffrey Xu

This scheduler seems like a good idea, as it is capable of simulating simpler schedulers but offers more flexibility in policies. However, using it effectively may require good understanding from programmers, as it is less immediately intuitive than round robin or priority scheduling.