CS 111 Final exam

Yunjing Zheng

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: Yunjing Zheng

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.

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 <u>secondary storage device</u>, **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.

Flash: Larger, slower to erase
DRAM: Smaller, faster, more failure prone

If you ever wish to relocate one device to another PC and want both to continue working independently of the other, then design two file systems. However, if you are only concerned with maximizing their performance together, design one.

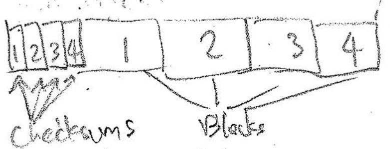
I will have a file system that mostly uses DRAM, since it is faster. It will be an EXT3 based file system with most things on DRAM, uncommonly used block data on flash. A background process will search for groups or large files in DRAM that have not been modified in a set time, say a week (or less time when the file system is getting full) and move data to flash storage. The flash will be written to wherever free, and the flash block bitmap (on DRAM) and inode (on DRAM) will be updated. (Inodes block pointer will have certain values, say range 10,000-20000 for flash.)

<u>File creation</u>: Create in DRAM
<u>Read</u>: find inode in DRAM (directories most/all in DRAM for fast access), get inode #, go through blocks. If block pointer is -1, then go to flash. Read from block corresponding to inode block #.
<u>Write</u>: Write block to DRAM. Just block - don't copy whole file if most of file is in flash. If was in flash, Update special flash block bitmap on DRAM. Update block bitmap and inode blk # on DRAM.

Logging: FS does not need logging, but it should be careful when moving data to RAM. Requires OS help. FS should confirm that info was successfully written to flash before changing inode's block # to flash block one, and updating its flash block bitmap (on DRAM, about flash data) to the block number.

Scrubbing: Every once in a while FS should check its flash block bitmap for unused/ near unused sections, rewrite the few blocks to fuller sections, and ask flash to erase the section.

Reliability: Store a checksum for each data block near the blocks for flash and DRAM blocks. This way the FS can locate corrupted data more easily. Since 1.1 Tbyte is a lot of memory, space should not be an issue. Can consider duplicating info for further reliability, if that's a concern.



Advantages of FS: minimizes flash erasing. utilizes the speed of DRAM and the space of flash.

Concerns: Requires great DRAM and FLASH cooperation, which needs OS support.

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 is a bad idea for a typical personal computer. The bidding process will required a ton of overhead. A typical multilevel priority queue scheduler with many I/O-calling processes will spend a lot of cycles rapidly round robining through those high priority processes at around 1 millisecond per process. If the processes bid for such a small time slice, the CPU will spend too much (likely over 50%) of the time handling the bidding and too little time executing processes. Thus bidding will only work for long time slices with little to no I/O. Furthermore, there are concerns about fairness. A bidding can easily cause certain processes to take up nearly all CPU time and other processes to be starved if credits are not distributed perfectly.

In practice, an extra category can be added to process control block the number of credits the process thread currently has. During the bidding period, to increase efficiency, all ready processes can be randomly placed in a bidding queue. Those who are willing to pay a increment more than current bid will stay in the queue and the CPU can cycle through the queue asking "will you pay this or give up" until only one process is in the queue. The bid price will be subtracted from the process's credits. Of course, CPU should decide on a set time slice, starting price, and increment price. Or, the CPU can have several different sets of processes in different queues competing for different time slices, and cycle through the sets. The CPU should hand out credits to old processes every so often so long running processes don't die.

Each process should specify a bidding policy. The ones without a bidding policy should be given a default policy so that they don't stop functioning.

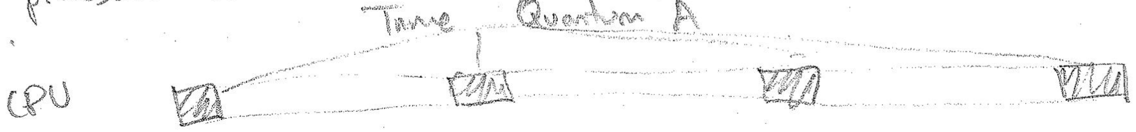
Kernal level threads should have seperate credits. Default distribution can be process total credits / # threads or 1/2 total credits for main thread, $\frac{1}{2 \times \# \text{threads}}$ × total credits for other threads. Libraries should allow the threads to specify a custom distribution policy. Threads should bid seperately since threads who are waiting on a critical resource should not bid while threads who are using critical resources should bid high to allow for optimum performance.

Comparison: This method lets the kernal know how important it is that each thread/process runs at the cost of high overhead. It cannot handle lots of high I/O processes as well as a round robin, "switch upon I/O call" approach. It is not as fair as a multilevel scheduling queue which ensures that each process gets to go at least every once in a while. It is much slower than both methods. The method is best for situations in which most processes require cooperation with other processes/threads and often cannot do anything useful until another process has completed their part. However, the yield command often works for these scenarios and may be less over-head than bidding. Works like an importance rating for soft real time, does not work for hard real time, since in hard real time every process must run on time, so policy isn't really a thing.

Vagueness

Given Credits: OS should decide how many credits its own software starts with and obtain every time slice. Users should be able to modify the start and obtain values of their own processes, within a reasonable range.

Time Quantum: As mentioned before, time slice cannot be very small. To increase efficiency, time slices should be at least 50x the bidding time in order use the CPU efficiently. There can be many different processes (long running no I/O, short lots of I/O) that can go in diffrient queues and bid for different time quantums, so it's not necessary to decide on a single time-slice. To further increase efficiency, processes can bid on a set of time slices rather than single one.



Handle Interrupts:

On multicore CPUs, one core can be in charge of critical tasks and a bit of process execution in spare time, letting other cores focus on process execution. The core that must balance critical tasks and process execution should be able to sell time quantums, push back time quantums in case of interrupt, and refund processes of they were push too far back or dropped.