CSE 121: Operating Systems - Architecture and Implementation
Homework 2
Winter 2007

Due: Tuesday, February 20th, in class

1 Rio File System Cache

a) Soft Update's performance degrades markedly when the file cache fills up. In particular, its meta-data updates cannot always be performed asynchronously. Given an infinite-sized file cache (i.e., all files could fit in memory), where meta-data updates will always be guaranteed to performed asynchronously, would there be any motivation for using a scheme like the Rio File Cache? Explain why or why not.

Yes. File caches can't prevent all write traffic, regardless of their size. Dirty data blocks need to be written to disk eventually for protection against memory corruption or system crashes. To ensure reliability, applications often synchronously flush data to disk (over-riding Soft Updates' attempts to avoid writing to disk), limiting the performance potential of a large cache.

b) While Rio is easy to explain, demonstrating its worth turned out to be quite a challenge. The authors conducted a substantial evaluation consisting of many thousands of bug injections in order to create realistic crashes. The key issue was which of these crashes lead to disk corruption. What was the difference between direct and indirect corruption? Why were both checksums and memtest needed? In other words, why not use just one method to catch both types of corruption?

Direct corruption occurred due to wild writes, when a bug in the operating system corrupted the disk buffer cache by modifying its contents. Indirect corruption, in contrast, invoked proper disk-access functions with invalid parameters, causing eventual corruption of the buffer cache. To test direct corruption, the authors computed a checksum of every buffer cache entry when it was created, so they could tell if the contents of the cache after a crash was the same as the original contents. Indirect corruption, however, would not be detected as the checksum would be computed over the already corrupted contents. As an alternative, the memtest program wrote deterministic contents to desk, so the authors could tell if what was on disk was different than what should have been on disk. While memtest could not directly identify whether corruption was direct or indirect, combining the results of the memtest (all corruption) with the checksum tests (only direct) allows the computation of the amount of indirect corruption.

2 Global Memory Management

a) What is the purpose of the *MinAge* parameter in *GMS*? What could happen if it was calculated incorrectly?

To control when nodes forward pages to a remote node. If a local page is older than MinAge, it is simply evicted without forwarding. If the value was calculated incorrectly, a node might incorrectly evict too much local memory (when it should have forwarded those pages for caching elsewhere in the system), or might cause undue overhead by forwarding local pages that would eventually be evicted during that epoch anyway.

b) Microprocessors have reference bits to help Operating Systems approximate LRU. Can this information provide enough information to help decide a global LRU? If yes, explain the how (i.e. how the *initiator node* uses this information) or why it cannot.

Reference bits are used by the Operating System to decide which page to evict, often through the Clock algorithm. Any idea of age*, however, is relative to pages within its local storage. This isn't sufficient to derive a global LRU since the initiator node will not be able to tell who has the oldest pages in the system. The only way to order pages across machines is to keep timestamps on each page, which would be both expensive and difficult due to clock synchronization issues.*

3 Application-controlled File Caching and Prefetching

a) The Application-controlled File Caching and Prefetching paper describes four necessary conditions for optimal prefetching. Show, by constructing an example of file access stream, that an algorithm is not optimal if it does not satisfy rule 3. (State your assumptions clearly about cache size and the amount of time it takes to read a block)

There were many possible examples here. In general, I was looking for you to show me that if there was some (supposedly) optimal algorithm A that prefetched a page B before A, (where B was subsequently accessed before A), that you could create a new algorithm A', identical to A in every way except it prefetched A instead of B, and performed better.

b) This question will run through an example of the LRU-SP policy high- lighted in the Caching, Prefetching and Disk Scheduling paper. For this question assume that the size of cache is 5 blocks and there are 3 processes running—A, B and C. For readability, a read request is given as a process/block pair. For example, the read request $A12$ means that process A is trying to read block 12. Finally, assume a basic priority scheme where each process gives a higher priority to a higher numbered block. For example, if the least recently accessed block is $A15$, and A has another, more recently accessed block in cache, $A10$, A will choose to have $A10$ evicted instead of $A15$.

Consider the following string of read requests (order is from left to right i.e., $B25$ is first request): $B25, C21, C05, B05, B13, C07, B05, A21, C05, A13, C11, C31$

Assuming an initially empty cache, for EACH read request, give the the contents of cache as well as any place-holders (and specify the cache blocks each place-holder points to). Start showing cache contents after the first 5 requests (since the first 5 trivially fill up cache) that occurs as a result of servicing that request. For example, the following table gives the cache contents after the 5th read request:

Request	Cache4	Cache3	Cache2	Cache1	Cache0
B13	B25	C21	C05	B05	B13
C07	C21	C05	B25(05)	B13	C07
B05	C21	C05	B13	C07	B05
A21	C21(05)	B13	C07	B05	A21
C05	B13	C07	B05	A21	C05
A13	C07	B13(05)	A21	C05	A13
C11	B13(05)	A21	C07(05)	A13	C11
C31	A21	C07(05)	A13	C11	C31

4 Lottery Scheduling

a) This question will run through a basic example of a lottery scheduler. Assume we start with two processes—A and B—and a base currency of 1000 tickets. 500 Tickets are assigned to process A, and 500 tickets are assigned to process B. A and B both have their own currency of 100 tickets each (which is backed by 500 tickets of base currency for both processes). Assume A has two threads associated with it: $T1$ and $T2$. $T1$ is assigned 25 tickets of A's currency and $T2$ is assigned 75 tickets of A's currency. Process B also has two threads associated with it: $T3$ and $T4$, where $T3$ has 20 of B's tickets and $T4$ is assigned 80 tickets of B's currency. Assume that we are starting a new time quantum τ_i and at the beginning of this time quantum thread $T1$ is current blocked awaiting I/O.

   (a) For time quantum τ_i, for each thread $T_j \in T1, T2, T3, T4$, what is the probability that T_j will be chosen to run?

$$\Pr[T1] = 0. \Pr[T2] = .5, \Pr[T3] = .1, \Pr[T4] = .4$$

   (b) Assume that for time quantum τ_i, $T3$ was chosen to run, but $T3$ only ran for half of the quantum. For time quantum τ_{i+1}, $T1$ has become unblocked; for each thread $T_j \in T1, T2, T3, T4$ what is the probability that T_j will be chosen to run?

$$\Pr[T1] = .114. \Pr[T2] = .341, \Pr[T3] = .182, \Pr[T4] = .364$$

b) Describe why the lottery scheduler is a well-suited scheduler for the Monte-Carlo algorithm. For contrast, how would one go about mimicking this behavior using a traditional priority-based scheduler?

Monte Carlo algorithms generate results whose accuracy improves with the square root of the number of iterations, so the goal for each individual computation is to give it high priority early, and decrease its priority later. The challenge is that these priorities need to be relative to the Monte Carlo simulations, but the total priority for all the simulations should remain constant with respect to other processes on the system. The lottery scheduling algorithm allowed calculations to move tickets between themselves in proportion to their current accuracy. In a traditional scheduler, each thread would need to continually update its priority with respect to all the others, but then the priorities for all threads would need to be renormalized to ensure the total priority remained constant. With lottery tickets, the normalization step comes for free.