Solutions

Comprehensive Exam: Software Systems (60 points) **Autumn 1991**

1. (20 points total) *Concurrency*

   (a) (7 points) Synchronization can be implemented using Dekker's algorithm. Lamport's algorithm.. etc. assuming that BCCI has designed the machine with atomic read and write operations (so a read operation never returns the result of a partial write). Thus. the machine can still be useful but they should market the machine according to the costs of this approach. Namely. the machine would be best with fewer processors (ideally 2) to simplify the Dekker's implementation. Also. the machine would be best with applications with low-contention locks because of the cost of contention on locks. and even the cost of acquiring the locks without contention. (The extreme is to use the machine just for multi-programming. when only the operating system executes with coupled parallelism.) Of course. all this assumes that the Bulgarians dont surprise us by announcing they provided a "split" compare-and-swap instruction like in the MIPS R4000. For the future. you advise them to provide hardware support for locking to support fine-grain concurrency. but no fancy hardware semaphores. etc. (assuming you think that is a danger). Their omission does affect whether processes do busy-waiting. because typical read-modify-write operations simply efficiently support busy-waiting. Process blocking is implemented by the operating system.

   (b) A read-only process can run into problems in a variety of ways. even if there is only one writer. For example, it might be report generator needs to balance income and expense statements. An update during its scan of the data could produce inconsistent results. On the other hand. Smidley might implement a singly-linked list out of memory records that are either free or in the list. A record can be inserted as a single write (after initialization) and removed with a single write. The updater can also increment a generation number for the data structure so the reader processes can determine if the data structure might have changed during their access. and then simply redo the operation if so. Another good example is having the writer update a shadow copy of a data structure and then replace the real data structure with the shadow copy in one atomic write to a pointer, for instance. Answers to the first part that include faking multiple writers, such as having the readers ask another process to do the writes. are considerable suboptimal because they indicate a lack of appreciation of common readers/single writer conflicts, which are important. Answers to the second part that basically have Smidley implementing synchronization (in the sense of causing readers to wait) are also suboptimal because they clearly do not recognize the wide range of scenarios in which blocking can be avoid altogether.

   (c) Paula, Hoare has the signaler suspend, the signaled acquire the monitor lock immediately. and the signaler then resume as soon at the signaled had blocked again or left the monitor. Thus. the signaled process seems the state of the monitor at the time of the signal. but the signaler has to be careful when it resumes. Brinch-Hansen. in actually trying to implement this nightmare (which Hoare never bothered with) in Concurrent Pascal. required that the signal be the last action taken by the signaler before leaving the monitor. But. Paula. just like there are musicologists. and then there are those that produce hit records. in practice. it is more common to implement monitors so as to reschedule the signaled processes but have them wait for the monitor lock. like all other processes. and in particular wait until the signaler departs the monitor. Consequently. a signaled process has to recheck the condition to ensure it is true. despite the original intentions of Tony Hoare. Thus. the signal is a *hint*. not a guarantee. (This reduces the coupling of the monitor mechanism to the scheduler. and reduces the typical number of context switches. all important things in practice.)

2. (20 points total) **Virtual Memory**

   (a) The CPU time is clearly 15 seconds per pass. With LRU and a sequential scan of the data corresponding to a more or less sequential scan of the pages. one would expect a page fault on each of the 37.500 pages of data. for a cost of 375 seconds. (There is no overlap of CPU and disk transfer time because we assume we are just brainlessly tripping across each missing page.) Thus, a scan in the simple approach takes 390 seconds. A factor that might make this worst is the mapping of the data onto pages. For example. imagine that the data is arranged as two stripes across all the pages such that we cycle through all the pages twice per timestep. (Sounds bad, but some vectoring processing data organizations do in fact lead to such bad locality behavior!) A simple improvement on this approach is to use an "elevator" approach, where one scans back and forward. so on each scan, one only pages in the "last" 50 megabytes of the scan. so the cost is 125 seconds for page-in plus 15 seconds CPU, for 140 seconds per timestep.

   (b) The obvious thing is to prepage and post-flush the pages as we use them. Simplistically, that means we have 2 milliseconds per page-in and out and 37.500 pages. so it takes 75 seconds for CPU to page in pages. and 15 seconds to compute. so 90 seconds. However, the latency for page-in/out in 10 milliseconds it still takes at least 375 seconds to complete page-ins for one scan. or 125 seconds using the elevator trick. (Here, we assume that we can page-in and page-out concurrently. else the times are doubled.) Thus, the page-ahead/behind purely allows the CPU time to be overlapped with the disk activity. but the elapsed time is the maximum of the two, which is the disk in both cases. Clearly. this elapsed time would be reduced if we could prepage multiple pages simultaneously, like if we had four independent disk channels, two simultaneous page-in and two for simultaneous page out. Then. the disk latency is reduced to 75 seconds with the evalator approach. and CPU time dominates, so a timestep is 90 seconds. A similar effect occurs if we can do multi-page transfers, with a single charge of 1 ms for initiating, one charge for rotational latency, and double or whatever the transfer time. (There

4

are generally limits on the number of pages of contiguous transfer without a seek or rotational delay though.) Clearly, that could get the disk I/O fully overlapped with the CPU time as well. There is at least one additional scheme for minimizing paging (without the elevator/reverse scan) but that is beyond what we consider here (Contact David Cheriton if interested.)

(c) A key real world factor not mentioned is seek time, which can be 10's of milliseconds, so significant. Without significant locality, seek time could easily triple the disk access time, and thus triple the elapsed time for the timestep. Another factor is the available I/O bandwidth. As described above, the elapsed time with 4 channels is far less than one channel. The final other factor is contention with other applications and system services running on the machine, which could place additional competing demands on the memory, I/O bandwidth and CPU, further slowing down the execution of the program. There is some effect because of paging of the operating system data structures themselves, such as page tables. However, with any reasonable VM design, the page tables are a small percentage of the size of the virtual memory, like 5 percent or less, so their overall effect should be minimal. Also, the page tables for a read-ahead page are naturally referenced as part of the read-ahead, so one would not expect random, unpredictable page faults from these structures either.

3. (20 points total) **File Systems**

(a) The key functions of a file system include file access (i.e. open, close, read, write) implementation — the basic file abstract data type implementation — file data buffering, disk allocation of space for files, file directory implementation, file protection, and facilities for backup and recovery. As part of the implementation, there are generally heuristics and algorithms that recognize common file properties, such as most files are short, most files are read sequentially in total, and most files are short-lived.

(b) The 64-bit addressing allows one to address all files as part of this large address space in theory (perhaps) but there are problems in practice. First, most files are small but some are very large, and incrementally grow to a large size, like log files. Locating gigabytes of address space per file could use up even a 64-bit address space, but less means that a file might have to change its name as it got bigger. Second, users will want to identify files by character-string names anyway, so there still has to be a file directory system. This directory system or some additional mechanism would still be required to implement file protection and security, which is typically lacking from a virtual memory system (because only one address space can access its segments, in the conventional model). Thirdly, there is an evolution problem in practice. Some machines may have 64-bit addressing now, but when will all have it? One could argue that the existing mechanisms in a typical virtual memory mechanism for buffering (the page pool), disk allocation and disk transfer can subsume that of the file system. but these tend to be the same basic mechanisms used in the file system. so one is considering a reduction on software. not elimination of the techniques I have carefully studied. Moreover. the trend is actually to have file systems mechanisms subsume virtual memory mechanisms (including the unification of the file and page buffer pools) because the file system mechanisms tend to be more powerful. In particular. one would like to retain the file optimizations for read-ahead with sequential access. etc. regards of whether you regard this as virtual memory system or file system. Also. virtual memory systems have not conventionally provided facilities for persistent storage in the area of backup and recovery. so file functionality there would also be required. By the way. 64-bit addressing does not necessarily imply any changes in the amount of data kept in physical memory. just that kept in virtual memory. So. yes I missed some great parties. but I still learned some important stuff!

(c) Himmel. too bad you were watching Sesame Street when the dudes at MIT were developing Multics. which basically did all this stuff 20 years ago. It didnt exactly catch on. did it? More seriously. many applications and programmers seem to just prefer the stream read/write interface rather than a memory-mapped I/O interface to files. For one. it is safer because if your program goes wild. it is less likely to corrupt the file. One might view that it throws file access into the memory allocation problem. and memory corruption bugs are some of the hardest for programmers. Also. lots of I/O refers to things other than disk files. such as terminals. pipes. communication lines. etc These object do not have convenient fixed-size blocks that are compatible with the virtual memory page size (necessarily). so these devices would have to be handled by a separate mechanism. a major step back from the Unix uniform I/O mechanisms. In general. history is on my side: the idea of single-level store has been around for a long time. and while not a failure completely. it has failed to displace the stream model of I/O. and conceptual separation between virtual memory and secondary storage.