1996 Comprehensive Exam: Software Systems (60 points total)

1. (15 points) Solaris 2 uses "adaptive mutexes" to protect access to every critical data item. An adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked (already in use), the adaptive mutex does one of two things. If the lock is held by a thread that is currently running, the thread waits for the lock to become available. If the thread holding the lock is not currently in the run state, the thread blocks, going to sleep until it is awakened by the lock being released.

   a. Why are both options (spinning and sleeping) provided when the data are already locked, and why does the decision depend on whether the thread holding the lock is running?

Spinning makes sense if the lock will soon be released, since you avoid the overhead of a context switch. Sleeping makes sense if the lock will be held for a while, since you don't want to consume CPU resources needlessly for a long time. The state of the thread holding the lock is a hint about how long the lock may be held. If the thread is not running, it might be a while before it releases the lock. The fact that it's not running could indicate that it's waiting for I/O to complete, or some other such long operation.

   b. For what sort of application workload is this feature beneficial?

One example is that a real-time multi-processor workload might benefit from this approach if locks are only held for a short time. Threads are likely to get the locks they need very quickly if the threads holding them are still running. Workloads that demand efficient use of a multiprocessor also benefit, since if a lock is held for a long time, threads waiting for it will get out of the way and make the processor available for another thread to run.

   c. On a uniprocessor system, how do adaptive mutexes behave?

On a uniprocessor, only one thread can run at a time. Thus, the thread already holding the lock can never be running. This means the thread requesting the lock will always sleep, giving up the processor. This makes it easier for the thread holding the lock to be rescheduled and (one hopes) give up the lock soon.

2. (10 points) Page replacement in a virtual memory system can be either global or local. In global page replacement a page fault is satisfied by selecting a page from the set of all physical pages. In local page replacement, the page fault can only be satisfied from the faulting process' own set of physical pages. The number of physical pages allocated to a process in the local scheme does not change over its lifetime.

   a. What are the potential disadvantages of each scheme?

Disadvantages of global scheme: A process cannot control its own paging behavior, since its pages can be taken away by other processes. One heavily-faulting process could hurt the performance of other small, well-behaved processes.

Disadvantages of local scheme: A process that needs more pages cannot get the benefit of free memory that is under-utilized by some other process. The allocation of pages per process is restrictive. A process that changes behavior over its lifetime can't increase the number of pages allotted to it or give up pages if it no longer needs so much memory.

   b. Why is global replacement more commonly found in existing systems?

A process that needs more memory can take advantage of free pages in the global pool, so memory is more efficiently used overall. Thus system throughput is generally higher in the

global scheme.

3. (10 points) Monitors are a popular mechanism for ensuring synchronization of processes that access system data structures and resources. However, monitors do not guard against starvation.

   a. Briefly describe why monitors do not guard against starvation.

A process waiting on the monitor lock may not be the first one to run when the lock is released. If a new process happens to execute the monitor before the other process is woken up, the new process will get the monitor lock first. If new or lucky processes keep testing the monitor lock at just the right time, then the poor waiting process may wait forever. Note that there are also some definitions of monitor semantics where a process waiting on a condition variable may be "signalled" or "notified," only to find by the time it's scheduled to execute that the condition for which it is waiting is no longer true because another process has run just before it and has changed something. The unlucky process will then have to wait on that condition again. This could go on forever.

   b. Given this problem, why is it practical to use monitors in real-world systems? (How is the starvation problem handled in practice?)

In practice we rely on at least several things to make this scenario unlikely. First, a nicely behaving scheduler can help. Second, we try not to run systems with resources so scarce that there are likely to be so many processes wanting them simultaneously. Third, careful design of the system is important to make sure that monitor locks are not held for a long time, and that particular monitors do not become bottlenecks for resources under a lot of demand. For example, we can balance locking around whole sub-systems with finer-grained locking for individual items. One lock around the whole file system could be a bad idea. A lock per file-data-structure may make more sense.

4. (20 points) File cache update policies can be categorized by the amount of delay they allow before file updates must be written to disk. Compare the following policies based on reliability, process latencies, and disk throughput. For each policy, also describe a workload that would benefit from the specific policy.

   •Write-through immediately (no delayed writes)

The highest reliability, since the process doesn't continue until each write is safe on disk. The highest latencies: a process waits each time it writes for the data to be sent to disk. If there's much I/O, processing speed will be tied to the speed of the disk, rather than the speed of the CPU. The worst throughput: all data modified are written to disk, even if later they are overwritten. There's also no time to order the various writes for more efficient disk access. Applications that demand high reliability may want to be sure their data are written to disk. Transactional workloads would be an example.

   •Write-through on close (write modified data to disk when the file is closed. The close operation doesn't return until the write finishes. If a process with open files exits, the system calls the close operation on these files before the exit completes.)

Reliability depends on how long the files are open and how many writes are done to them before they are closed. If files are open for only a short time, reliability may be pretty good. Latencies also depend on how long files are open. If files are open only a short time, then this is almost as bad as "write-through immediately." Disk throughput also depends on how long files are open and often previously-written data is overwritten. If, for example, some section of data is rewritten 10 times before the file is closed, only the last write needs to be

performed. This leaves the disk available for other operations. Workloads with lots of rewriting and files that are open for a long time might benefit. (Unfortunately, even though this policy is used in some well-known systems, UNIX-like workloads often tend to leave files open for only a short time, so this policy doesn't help performance much in that environment.)

- Delay for 30 seconds (write out modified data from the cache when it's 30 seconds old)

This provides only limited reliability, since data that an application believes are already written could be lost for up to 30 seconds. This improves process latencies, since the process never needs to wait synchronously for the data to reach disk. Throughput is improved since the disk writes can be ordered for efficient disk access. In a typical UNIX workload with lots of temporary files, some files and data will have been deleted before the 30 seconds is over. Any modifications to them no longer need to be written through to disk, which improves disk throughput.

- Delay for 5 minutes (write out modified data from the cache when it's 5 minutes old)

The same as just above, only worse for reliability and better for performance.

5. (5 points) In UNIX, there are separate system calls to start a new process in the image of its running parent (`fork`) and to load into a process's address space fresh code and data from an executable that it should use for execution (`exec`). Why doesn't UNIX use just one system call that both creates a new process and loads the desired image (code and data)?

The UNIX scheme is simpler and more flexible. Many system daemons `fork` a copy of themselves, and the child process shares various resources with its parent and can communicate and cooperate with the parent through these shared resources (e.g. file descriptors). If the call always overwrote the process image and created new resources, then this wouldn't be possible. Instead, `exec` can be called when desired by the child process after it has been forked.

(51)