Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Problem Set: Advanced topics (networking, performance engineering)

1. Consider a high performance networking application running on a high end multicore server. It is found that, under high incoming packet load, the system spends a large fraction of its time handling interrupts and context switches, leading to very little productive work at the application layer. Suggest one mechanism by which this problem can be mitigated. (For this question and the next, you are required to provide a description of the mechanism, not just its name.)

   Ans: Polling or batching interrupts

2. Consider a web server that uses non-blocking event-driven I/O for network communication, but uses blocking I/O to access the disk. The web server wishes to run as multiple processes, so that the server can be available even if some subset of the processes block on disk I/O. Further, the web server wishes to receive web requests only on port 80, and not at different ports in the different processes. Suggest one mechanism by which the multiple server processes can handle requests arriving on a single port on the system.

   Ans: The master process can open a socket on port 80, fork multiple processes, and all child processes can accept connections from the same socket on port 80 using locks for mutual exclusion. Or, the master process can alone listen to requests on port 80 and assigns all blocking disk I/O to worker processes via IPC mechanisms.

3. Consider a TCP server socket program written in an event-driven manner. The server receives requests from multiple concurrent clients. The main thread of the server monitors read events on the server listening socket as well as all client sockets using the `select` or `epoll` family of system calls. When not processing any events, the server always blocks in an event-driven wait loop, e.g., `epoll_wait`, waiting for notifications. To process a client request, the server must read some data from the disk and send a reply back to the client. In order to avoid blocking the main event-driven server thread during disk reads, the server uses worker threads to block on disk I/O. After reading a client request, the main server thread spawns a new worker thread and passes the client request to the thread. This worker initiates the disk read and blocks until the disk read completes, while the main server thread continues to process other events on the sockets. Once a worker thread completes the disk read, it places the data read from the disk in a shared datastructure that is accessible by the main server thread (using suitable locking), and terminates. Assume that the main server thread does not automatically get any notification from the OS when the worker thread terminates. Now, we require that the server send a response back to the client once the worker thread has completed the disk read operation. There are multiple mechanisms to accomplish this goal, and the rest of the question lets you explore the various design choices.

   (a) One way in which the server can send responses back to the clients is by monitoring the status of the shared datastructure every time an event occurs, i.e., when the main server thread returns from the `select` or `epoll_wait` system calls. At this time, if the server

finds that some worker has placed a response in the shared buffer, it can fetch this response and send it back to the client on its socket. This solution is almost but not fully correct. Describe one scenario where this solution will fail.

(b) Describe how you can fix the server design to overcome the above failure scenario.

Ans: (a) if only one request and no other traffic comes in, server will never wake up from epoll loop (b) Have a unix socket from worker to main thread. When worker finishes, it will write to the unix socket, which will cause the server to wake up from the event loop.

4. This question will explore the design of a simple multi-process file server. The server is expected to serve multiple concurrent clients. Each client opens a TCP connection to the server, and sends requests for files on that connection. The web server must read the name of the requested file from the socket, read the file from disk, write it back into the socket, and wait for the client to request the next file. The server must serve the client in this manner until the client closes the connection.

   You are given a single-threaded file server running as a single process on a multicore system with a Linux-like OS. Because the server performs multiple blocking operations (accepting new connections, reading from sockets, reading from disk), a single server process can neither effectively serve multiple clients nor efficiently utilize the multiple CPU cores. In order to overcome these problems, you must modify the server to fork some children and delegate work to the child processes. Note that you are constrained to increase the parallelism of the server only by spawning processes and not threads. Further, you may not make any of the blocking operations (e.g., socket reads) non-blocking.

   The following sub-parts of the question will let you describe the design of your multiprocess file server. Note that multiple correct designs are possible; it is sufficient if you describe one of the possible correct designs.

   (a) When (i.e., at what point in the server's code) does the main server process spawn a child process? And when does it reap this process?

   (b) How is the work of handling a client (accepting a connection, reading from socket, reading from disk, writing to socket) divided between the parent and child processes? In other words, who does what part of the work?

   (c) How do the parent and child processes exchange information with each other? Note that the communication can be implicit via variables or file descriptors inherited at the time of fork, or explicit via IPC mechanisms You must describe all such exchange of information between the parent and child processes.

   (d) State any other aspect of your design not covered by the questions above, and any assumptions you have made in your solution.

   Ans: One possible design is spawn a child every time a new client connects and accept returns. The parent process periodically (say before calling accept) reaps any dead children. The child process does the network read, disk read, network write for the client, while the main process does the job of accepting new connections. The child gets the new file descriptor of the client to serve via a variable implicitly from the parent.

5. Consider a web server that receives 1 M requests/second over the network. Some cores in the system are dedicated to running ksoftirq (botton half of the interrupt handler) processing, while

some other cores are dedicated to running a multi-threaded user application that processes these requests. Incoming traffic is distributed equally to the cores running the kernel (ksoftirq) processing first, and then distributed equally to the application cores for further processing. You may ignore the time taken for other tasks like top-half interrupt handling or any other work done on the server.

(a) The ksoftirq thread on each core runs in a tight loop, continuously processing requests without blocking in any way, and needs 20 microseconds to process each request. What is the minimum number of cores that must be running ksoftirq threads in order to process all incoming load in a timely manner?

Ans: 20 cores (each core does 50K req/s)

(b) After kernel processing, the incoming requests are distributed equally and processed by a multithreaded web server program running on multiple cores. Each application thread runs for 10 microseconds on the CPU to handle the request, and spends another 990 microseconds waiting for disk I/O for the request. After spending 1 millisecond processing a request in this way, it moves on to the next request. What is the minimum number of application threads that must be run on each core to fully utilize the CPU cycles of the core? Assume that the disk I/O is not the bottleneck.

Ans: 100 (computed as 1 millisecond / 10 microseconds)

(c) What is the minimum number of cores that must be running application threads in order to process all incoming load? Assume multiple threads will run on each core, as computed in the previous part, to fully saturate a core.

Ans: 10. Each request needs 10 microseconds, so each core can process 100K req/s (assuming enough threads as available), so we need 10 cores to handle 1M req/s.

(d) Assume that the average response time of each request is 2 milliseconds, which includes all user and kernel processing and wait times in various queues in the system. With an incoming load of 1M requests/second, approximately how many requests are being served by the system at any point in time?

Ans: By little's law, 2000

6. Consider a system that has a 16MB cache in the CPU (assume a simplified model of only a single level of cache), 64 byte cache lines, 4KB page size, 8GB main memory, and a TLB that can hold 1024 entries. The single CPU is running a program that accesses a large array of 4M (2^{22}) integers repeatedly. Assume the integer datatype needs 4 bytes of storage. The array is accessed sequentially from beginning to end repeatedly, and we are interested in computing the long term averaged CPU cache and TLB miss rates in steady state. Assume the CPU cache and TLB use a LRU eviction policy. Ignore CPU / cache / memory / TLB usage due to any other processes, or the kernel. Ignore prefetching or any other optimizations in hardware.

(a) What is the cache miss rate?

Ans: In the long term, the entire array of 16MB fits in cache, so the miss rate after many accesses is close to 0%. However, if one assumes other variables etc that may eat into the cache, and the working set is larger than cache size, then access to every cache line will lead to a miss, and the miss rate would be 1/16.

(b) What is the TLB miss rate?

Ans: 1/1024, or one miss for every 1024 array elements, which happens every time we cross a page boundary. The array spans 4K pages, so all entries will not fit in TLB. In the long term, across repeated accesses, every new page access will lead to a TLB miss, since an entry would have been evicted by the time we loop over it again.

(c) Now, suppose we turn on the huge page optimization, and all page sizes are 4MB. Recompute the CPU cache miss rate in this scenario.

Ans: Same as earlier, 0%

(d) Recompute the TLB miss rate with the huge page optimization of the previous part.

Ans: 0% now because there are only 4 pages now, so only 4 entries are needed in TLB

7. Consider a multi-threaded server that has a master-worker thread pool architecture. The server receives requests to process from clients over the network. Every request is first received by the master thread, which takes 10 microseconds to process the request, before placing it in a request buffer shared with the worker threads. A pool of 500 worker threads fetch requests from this buffer one at a time and process them. The master thread runs on one dedicated CPU, while the worker threads run concurrently and share the 3 other cores available in the system. A worker threads spends 50 microseconds executing on the CPU and 5 milliseconds waiting for disk I/O when processing each request. After completing the request processing in this manner, it goes back to fetch another request from the shared buffer (if available). Assume that the performance of the system is limited by CPU processing at the master or worker cores, and not by disk I/O or the shared buffer size or by any other resource.

   (a) What is capacity of this system in req/s? That is, how many requests per second can this system handle from the clients? Also, mention the bottleneck (e.g., master CPU or worker CPUs) that is fully utilized at saturation.

   Ans: 60K req/s with bottleneck at worker CPU

   The master core can handle 100K req/s. Each worker core can handle 20K req/s, so 60K req/s across all 3 cores. Because the master-worker forms a pipeline, the capacity of the system is limited by the worker threads at 60K req/s. Note that we need about 100 threads per core to fully saturate the worker CPUs, and we have enough, so the number of threads is not the bottleneck.

   (b) Now, suppose we optimize the worker thread code to spend only 20 microseconds on CPU and 2 milliseconds waiting for I/O. Repeat the previous question with this optimization.

   Ans: 100K req/s with bottleneck at master CPU

   Now each worker core can do 50K req/s, so all worker cores can together handle 150 req/s. So the master CPU now becomes the bottleneck and the capacity is now 100K req/s.

8. Consider a multi-threaded web server that receives a large number of requests from clients across the Internet. The users request web pages that are stored on disk at the server. In order to avoid going to disk repeatedly for every request, the server maintains an in-memory cache of popular files, as a linked list of web pages. The traditonal disk buffer cache used in the file system is disabled, and this web page cache is used instead. So, when the server receives a request for a web page over a socket connection, it first looks for the file in the in-memory cache, and reads it from disk only if it cannot find the web page in the linked list. A performance engineer who is given the

task of optimizing the performance of the system can do one of the following things to improve system performance:

- Optimize searching for files in the in-memory cache (say, by using a hash table instead of a linked list)
- Increase size of in-memory cache
- Decrease size of in-memory cache
- Align in-memory cache structures to 64 byte boundaries

For each of the performance problems given below, suggest which one of the above ideas is the best solution to the problem at hand. You must pick only one of the ideas for each problem, and you must briefly justify your choice.

(a) The system has a very high CPU usage, and a profiling tool shows that searching for items in the cache linked list is taking a very long time.

Ans: Optimize search.

(b) The system has a very high TLB miss rate.

Ans: Decrease the size of the cache, in order to reduce the working set size, and hence TLB usage.

(c) The main memory is almost full, leading to lot of swapping and page faults.

Ans: Decrease the size of the cache, in order to reduce the working set size.

(d) The main memory has enough free space, but there is a lot of disk I/O, and the disk capacity is fully utilized.

Ans: Increase the size of the cache, in order to reduce disk I/O