CS481 Final

David Lochridge

May 20, 2014

1

Managing the Processors

1.1

Having an operating system in this HPC environment is absolutely necessary. With such a large number of nodes and processors available, requires a managed resource system that an operating system provides, not to mention the added benefits of allowing multiple programs to interact with a given node, or scheduling of programs for individual nodes or processors as needed. The programming overhead for managing these nodes individually, especially considering the number of nodes and potential for cross-program interference, is a nightmare. Stated simply, regardless of which precautions you may take in your own program, without an operating system, a malicious user, such as a disgruntled programmer unable to design their own operating system, may decide to alter your program's state and cause any program you run to fail in a variety of maddening and awful ways.

Even assuming that hardware failures do not occur in such a massive system, and that no other users get any time on the system while you have that time, removing the abstraction of hardware access would require far more programming time than would be gained from removing that lowest level of abstraction.

Space-time (Non?)continuum

1.2

Time-shared in this context is context switching among processes on individual cores, as well as spreading access time over individual nodes and processes in the given compute node. While it is outside the scope of the HPCOS, submission of jobs from an external interface is also space-shared.

Space-shared in this context is storage of memory on disk and in memory space, to ensure that none of it interferes or overlaps each other. On the filesystem, users' data and programs are also a form of space sharing.

1.3

Time-sharing on the HPCOS would give the advantage of allowing multiple processes to run on some of the compute nodes, or over multiple compute nodes at once.

1.4

Space-sharing on the HPCOS constitutes sharing physical space on the individual compute nodes, and managing memory being used concurrently by the (potentially) hundreds of processes running on that node.

1.5

Different tasks running on a single compute node should be processes, to prevent interference in each others' memory space, as well as sustain the principle of least privilege. Using threads would require that all processes use the same memory space, part of the reason that an Operating System exists in the first place. In the case of some HPCOSs, only one task is given to a compute node or processor, in which case that process should still be a process for the purpose of preventing a possible system failure. In that event, if the Operating System were running the process as a thread, recovery would be considerably more difficult, and the entire compute node might need to be restarted.

1.6

Tasks running across different nodes are required to be processes. Any communication between these processes would have to be explicit communication, as attempting to maintain some semblance of shared space is far beyond current capabilities, though a number of research projects are working on allowing that communication in an implicit and efficient way. Even considering this abstraction, the processes would only have simulated shared memory, and would continue to remain separate processes.

Managing Memory

4.1

Assuming address space size is limited by the physical memory, and the size of the physical memory will not be expanded in the future:

2 TiB = 2^{40} bytes * 2 / 2^{10} = 2^{41} bytes

To address to any arbitrary byte, we would then need 41 bit long addresses.

4.2

In a non-swapping, non-paging based scheme, all state for a given process is always kept in memory. Context switching between processes is faster, as the

swapping that would slow that process down no longer exists, but each process is allowed a much smaller fraction of the memory than it normally would be. As the TLB and disk would not be used without swapping or page tables to cache, the average memory access latency would be 200 ns, with some number of reads and writes using memory every single time.

4.3

The address space limit for each program in the above scheme becomes the amount of memory available divided by the number of processes. In the HPCOS, this is:
2 TiB / 128 processes = 2^{41} bytes / 2^6 processes = 2^{35} bytes per process

4.4

Assuming that, on a TLB miss, the OS must load the page into the TLB before making another TLB request for the same page, and similarly for any given page.

With a single level page table, the costs of a memory read or write for each operation is as follows:

TLB Hit - 20 ns + 200 ns = 220 ns

TLB Miss, Page hit - 20 ns + 200 ns + 200 ns + 20 ns + 220 ns = 660 ns

TLB Miss, Page miss - 660 ns + 200 ms = 200 ms, 660 ns (200.00066 ms)

Dirty page eviction - 200 ms + 660 ns + 200 ns + 200 ms = 400 ms, 860 ns (400.00086 ms)

Assuming we have 2000 memory operations that fall into this distribution, there are:

1800 TLB hits

190 Page table hits

10 Page table misses

1 dirty page eviction (One of the page table misses)

Our total running time for those 2000 operations would be: 200 ns * 1800 + 660 ns * 190 + (200 ms, 660 ns) * 9 + 400 ms, 860 ns * 1 = 2200 ms, 492200 ns (2200.49286 ms)

Averaged over 2000 operations, we find 1.10024643 ms per operation.

Therefore, the average effective memory access latency would be 1.1 ms, 246.43 ns.

4.5

With an address space of 64 GiB and 16 KiB pages, we can have a total of $64*2^{30}$ / $16*2^{10}$, or 2^{22} pages. Because of this, we require 8 * 2^{22} bytes, or 32 MiB of space for our page table.

4.6

To make each table in a multi-layered page table for the system above into the size of a single page, we reduce the size of page tables to 16 KiB at each chance possible. To do so, we need a total of 14 bits to offset into that space, and determine the next page table to offset into. With our addresses being size 2^{41}, this means our address would become two offsets of 14, followed by one offset of 13.

We would then have a three levelled page table, in which the first two page lookups would return full page tables of size 16 KiB, the last page table lookup would index into a partially filled page table, and the final offset would give us the desired page entry.

Managing Storage

5.1

A RAID 0 scheme in this context would work by distributing data among the external network servers, and striping them as though each fileserver was an individual drive. This scheme would be beneficial in preventing bandwidth loss, though it would cause a huge number of problems in case any system failed.

A wise choice in this plan would be to ask that the NFS as a whole use a RAID 1 setup, for the most reliable, or a RAID 5 setup, for improved general performance, as well as some drive failure resistance.

5.2

For large contiguous reads, a program could attempt to reach multiple fileservers in the NFS, making requests for different section of the data that it had to read, in order to increase the read bandwidth. If the servers were in a RAID 0 scheme, as discussed earlier, the multiple requests would be handled in one attempt, with a request to an arbitrary server from the program, and multiple servers responding with their portions of the requested data.

5.3

For write requests to large, contiguous blocks in memory, the opposite approach should be taken compared to previously. Considering that any given fileserver has the same data as another, the program would send the data to one of the reachable filesystems, which would then write the data to disk, and propagate

that update to further disks, mimicing a RAID 1 without the requirement that all disks complete their write.

In the case that a different fileserver could be reached after the first write, and it hadn't had a previous change propagated to it, some performance would be lost. In that case, a program might want to attempt to access the NFS multiple times, in order to reach fileservers until it found one that had received its previous updates.

5.4

The primary difference between RAID 0 and RAID 5 is the block of parity stored once per stripe on RAID 5. While RAID 0 has a tendency to show better write and read times, RAID 5 makes use of data parity, allowing continued use after a disk failure or removal. In the event that the disks being used had different seek and spin speeds, RAID 5 may be able to outperform RAID 0 by avoiding use of the slower disk, but those returns would not likely make a huge difference.