DSC204A: Scaleable Data Systems Winter 2025

Lecture 9: Multiprocessor Scheduling and Memory Virtualization

Instructor: Umesh Bellur *Scribe: Kira Fleischer, Gabrielle Rackner*

1 Recap: Multiprocessor

As a review of the previous lectures, modern machines have multiple processors and multiple cores per processor. We covered several different scheduling policies, including First In First Out (FIFO), Shortest Job First (SJF), Shortest Completion Time First (SCTF), and Round Robin (RR). Of these four strategies, the first two are not preemptive, meaning once a process is running, it will not be interrupted. Alternatively, the latter two strategies are preemptive, meaning the OS can interrupt a running process to run a new (usually higher priority) process. Additionally, we reviewed the fact that for RR scheduling, the completion time does not need to be known beforehand, while for the other three policies, this does need to be known. Lastly, we reviewed the metrics used to evaluate the effectiveness of such policies: average turnaround time and fairness.

2 Multiprocessor Scheduling

Multiprocessor scheduling is complex since load balancing needs to be considered. Load balancing ensures that different cores/processes are kept roughly equally busy, reducing idle times. There are two main multiprocessor scheduling schemes we discuss: single-queue multiprocessor scheduling (SQMS) and multi-queue multiprocessor scheduling (MQMS).

- **SQMS** A first example of SQMS can be seen in Figure 1 below. With SQMS, there is only one queue for processes to join. For example, at the first time step, only processes A-D can run, and process E is not running. Then at the next time step, each job moves to a different core to allow E to begin running. The jobs continue to move core-to-core to ensure all processes get a chance to run. The issue with this policy is caching and context switching; since each job is bouncing between CPUs, context needs to be switched at every time step. The L1 cache, which is core-specific, essentially becomes useless as it needs to be reset every time.

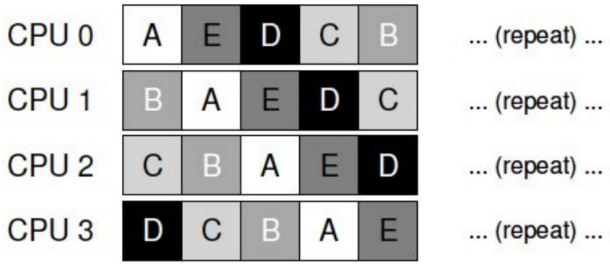

Figure 1: SQMS Example 1

An improvement upon this policy is the second SQMS example seen in Figure 2 below. This policy reduces the amount of context switching because processes A-D remain on the same core, and only E now requires updating the context.

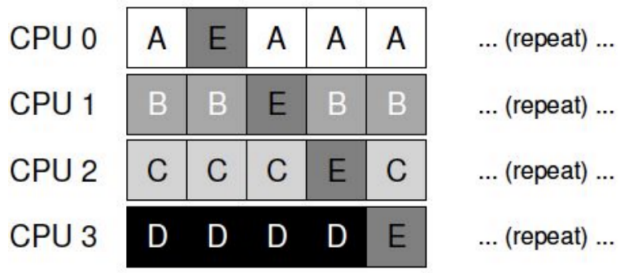

Figure 2: SQMS Example 2

- **MQMS** With MQMS, there are now multiple queues that processes can enter. In the example in Figure 3 below, there are two queues. The policy begins in a Round Robin fashion, then when job C is completed, on the first core now only process A remains running, and on the second core RR continues with process B and D. However once process A is completed, CPU 0 becomes idle as it does not have any more processes scheduled. This highlights the fairness and load-balancing issue that MQMS faces.

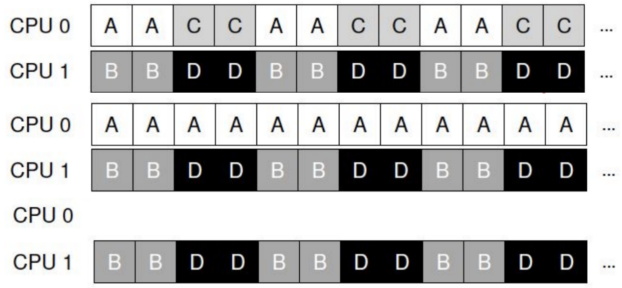

Figure 3: MQMS Example

3 Memory Virtualization

As a reminder, CPU virtualization enables parallelism and gives the illusion that many different processes can all be running at once. Similar to how CPU virtualization shares time, memory virtualization shares space, giving the impression that a task has all the memory it needs. This leads to the following goals of memory virtualization:

- **Transparency** Just like CPU virtualization, memory virtualization should be transparent with the OS handling everything behind the scenes, such as context switching.
- **Efficiency** Memory virtualization should be efficient in both time and space.
- **Protection/Isolation** Each task needs to have its own memory and that task's memory should be protected from other tasks.

3.1 Abstractions in Memory Virtualization

The **address space** is the abstraction in memory virtualization. The address space is decided using the width of the memory itself. For example, if the machine is 64 bit then the address space is 2^{64} (not physical memory). The Operating System and the current program must be able to fit within the address space. The address space has to be able to map the physical memory which is handled by the Operating System and revealed using **address translation**.

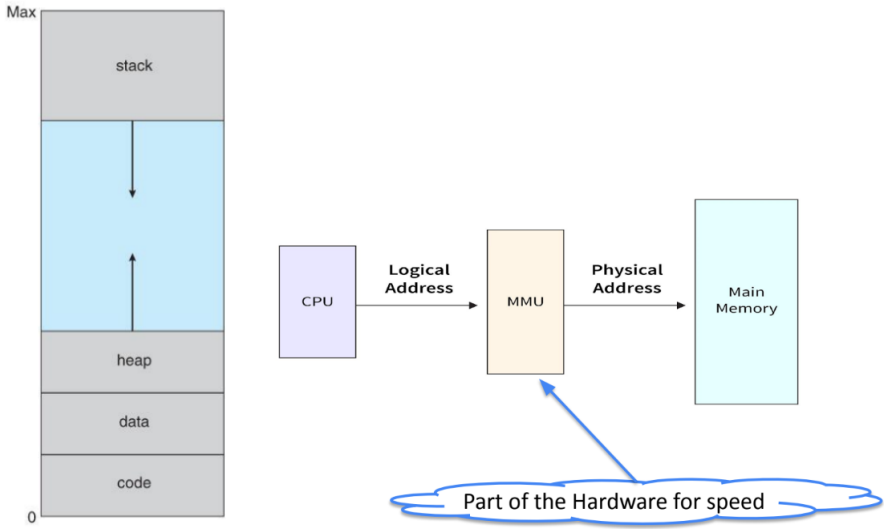

Figure 4: Memory Virtualization Abstraction

Address Translation allows for the transition from virtual to physical memory as CPU only deals with virtual addresses.

3.2 Fragmentation

Processes can only fit into contiguous memory. For example, if a 8 GB process comes in but there isn't 8 GB of contiguous memory available then the process is forced to wait. The process can only be accommodated if there is enough contiguous memory. This is referred to as **external fragmentation**. External fragmentation will waste time as processes have to wait for enough contiguous memory and memory will be underutilized. **Memory Efficiency** can be measured as ratio of memory used to total memory.

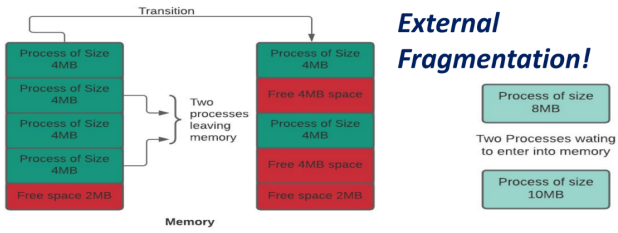

Figure 5: External Fragmentation

Internal fragmentation occurs when the operating system assigns more memory to a process than is used. This results in underutilized memory and can occur if the operating system predicts that the process may grow compared to initial memory estimate.

The solution to both internal and external fragmentation is **paging**. Paging involves dividing processes into small-sized chunks in order to fit within memory. A virtual memory address using paging will have a page number and an offset in order to keep track of the pages. This makes address translation fairly quick. If the page is not in memory then it will be in disk. Pages are swapped out during processing.

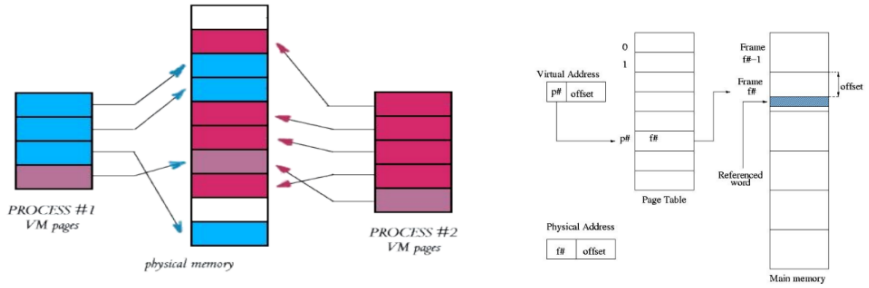

Figure 6: Paging in an OS