CMPSCI 677 Operating Systems **Spring 2014**

Lecture 4: January 31

Lecturer: Prashant Shenoy *Scribe: Sneha Shankar Narayan*

4.1 Threads

4.1.1 Thread Packages

- **Posix Threads (pthreads):** Widely used in C/C++ applications, and defines only an interface that needs to be implemented. This can be implemented as user-level, kernel-level, or via LWPs. This is cross-platform.

- **Java Threads:** Threading support is built-into the language and the JVM schedules threads. This is a two-level schedule decision, the kernel schedules the JVM and the JVM schedules threads. If thread support is implemented as kernel-level threads in the JVM, then they may be visible all the way to the kernel.

4.2 Multiprocessor Scheduling

A multicore system consists of cores (or CPUs) and caches (L1, L2, L3). Caches consist of data/instructions fetched from memory, and are used to speed up execution. Some caches (L1) are not shared with other cores, where other caches (L2, L3) are shared. A processor accesses anything in the memory using a shared bus.

In the multi-core/multi-processor setting, each core/processor makes an independent scheduling decision. The kernel runs on a processor, and the kernel scheduler picks a process for execution. Once a process starts to run, it either runs for the entire timeslice, or does I/O and gives up its timeslice. Once this happens, the kernel runs again and picks the next process, and so on. This happens in parallel on all the processors.

Two different ways of implementing a multiprocessor scheduler:

- **Central queue:** In this case, there is a single, global queue. Whenever a scheduler runs, it picks the next task (i.e. thread or process) from the head of the global queue. Given this setup, the global queue is a shared data structure, which must be locked by a scheduler before it picks a task from it. Once the queue is locked, other cores must wait for it to be unlocked before they can pick the next task. As the number of cores grows, the queue becomes a bottleneck. To overcome this problem, the centralized queue must be distributed to form multiple queues.

- **Distributed queue:** In this case, we have more than one queue for a set of processors. In the degenerate case, we have one queue per processor. With multiple queues, the central queue bottleneck issue can be resolved. However, we have to now figure out how to assign tasks to queues so the load is balanced across processors. If a queue has more CPU bound tasks, its corresponding processor will be more heavily loaded, versus one which has a lot of I/O bound tasks. To solve this problem, we need to periodically redistribute tasks so the queues are load-balanced.

Load-balancing the queues alone is not enough to improve performance. *Cache affinity* needs to be considered as well - *cache affinity* means that if a task was previously run on a core, it has an affinity to that core. If a task is assigned to a core, reassigning that task to the same core will enable the use of existing task related data/instructions that are already in the core's cache from the previous processing of that task. So, when multiprocessor schedulers reassign tasks to a set of distributed queues, they keep cache affinity in mind and rerun tasks on cores they ran on before. Central queue does not respect cache affinity as there is no control over which task is picked next - it is just the next task in the queue. However, with a single queue per core, cache affinity can be respected by making the task go to the end of its current queue. If load balancing is done, it may not be possible to respect cache affinity every time. While reassigning tasks to queues, there is a trade-off between load-balancing queues and respecting cache affinity, and every OS makes its own choice.

In multiprocessor settings, where tasks can get reassigned and have to start with cold caches, time slices are somewhat longer to allow the caches to warm up.

4.2.1 Parallel Applications

These are massively threaded applications, where threads are running in parallel to achieve some tasks. For such specialized applications, we need specialized schedulers. So far the schedulers we have seen try to schedule tasks on different processors while trying to be fair across all threads. These specialized schedulers schedule all the threads from an application on all cores, all at once - this is called co-scheduling or gang scheduling. These schedulers are used in special purpose machines that run massively parallel applications. As these threads are scheduled in a coordinated way, if a thread wants to communicate with another thread of the application, it does not have to wait for it to be scheduled at a later time.

When one thread is blocked, preemption can be done on that thread or busy wait can be done and context switching can be avoided.

4.3 Distributed Scheduling

We now move to scenarios where there are 'n' machines in the system. A distributed scheduler assigns tasks to machines in a system of 'n' machines.

4.3.1 Motivation and Implications

If the system is lightly loaded, and a task comes in, then with high probability it will be assigned to a machine that is idle. In such a case, a distributed scheduler is not really needed to move the job elsewhere. On the other extreme, if the system as a whole is heavily loaded, then a task could come in and with high probability it would get assigned to a machine that is loaded. In this case too, there is no need to do distributed scheduling as there is no other better machine to send the job to. Distributed scheduling is beneficial when the system as a whole is moderately loaded. In this case, there is potential for performance improvement via load distribution.

4.3.2 Design Issues

- **Measure of load:** When should the scheduler be invoked? We need to be able to measure the load on systems so we can decide whether to move a job from one system to another or not. Some metrics are queue lengths at CPU and CPU utilization.

- **Types of policies:** We need to define policies based on which the scheduler will decide to move load around. These policies could be static policies that are hardwired into the system. Policies can also be dynamic. Dynamic policies are based on querying the systems for their loads and deciding what to do based on loads. We can also have adaptive policies. Adaptive policies are those in which the policies or algorithms themselves change based on different loads and situations.
- **Preemptive versus non-preemptive:** Preemptive distributed scheduling means that you can move a currently executing process to another machine. Non-preemptive distributed scheduling means that scheduler cannot move a task that is executing. Non-preemptive schedulers are easier to implement. Preemptive schedulers are more flexible, but are more complicated to implement as they involve process migration.
- **Centralized versus decentralized:** In centralized policies there is a coordinator that keeps track of load on machines and decides where to move jobs. In a decentralized world each machine takes its own decision, there is no central coordinator and local decisions are made.
- **Stability:** The design needs to ensure that when you send a job to another machine, that machine does not get overloaded. Suppose a machine becomes idle and all other machines in the system send jobs to it. The machine then becomes overloaded, tries to send jobs elsewhere, and the load keeps oscillating. If policies are not in place to deal with such situations, the system becomes unstable, no real progress is made, and resources are wasted in sending jobs from one machine to another.

4.3.3 Components

A policy can be decomposed into four different components:

- **Transfer Policy:** The transfer policy deals with questions like: when should distributed scheduling be invoked? or when should a process be transferred?
- **Selection Policy:** Which process to send?
- **Location Policy:** Where to transfer the process?
- **Information Policy:** What information needs to be tracked in the system in order to make the above decisions?

4.3.4 Sender-initiated Policy

The sending machine does all the distributed scheduling.

- **Transfer Policy:** A very simple threshold based policy is used. If the load on the system is above a certain threshold, it sends the tasks somewhere else.
- **Selection Policy:** Here a non-preemptive scheduling policy is assumed. Only newly arrived tasks can be shifted, processes that are executing cannot.
- **Location Policy:** A machine can be picked at random. Machines can be polled sequentially or in parallel, and jobs can be sent to the least loaded machine. If multiple machines are searching for machines to send jobs to, many jobs can start arriving at one system, leading to unstability. To avoid unstability, the top k least loaded systems can first be chosen, and then one of those can be randomly picked to send the job to.

4.3.5 Receiver-initiated Policy

The receiving machine makes decisions.

- **Transfer Policy:** If a system's load falls below a threshold, the system becomes a receiver, and looks for jobs.
- **Selection Policy:** The system can either accept newly arrived processes, or if it supports process migration, it can accept partially executed blocks as well. (Sender-initiated policies can also decide to send partially executed tasks if they support process migration)
- **Location Policy:** The system can poll machines and look for the most heavily loaded machine, and relieve its load by asking for jobs. Sequential or parallel polling can be done.

4.3.6 Symmetric Policies

Both sender-initiated and receiver-initiated techniques co-exist in this system. Depending upon your load, you may be a sender waiting to off-load jobs, or a receiver looking for jobs.

- There are two thresholds - a low threshold and a high threshold. If any machine's load goes below the low threshold, it becomes a receiver, and similarly, any machine whose load goes above the high threshold becomes a sender. Machines whose load falls in between the two thresholds are neither senders nor receivers. Such machines just continue to work on the jobs they currently have.
- When you are a sender the sender-initiated policies will apply. If you become a receiver, the receiver-initiated policies will apply.
- In a very large system, having say 10,000 nodes, when senders and receivers actively trying to do send and receive tasks, the chances of finding a match increase. Otherwise, a sender may have to poll several machines before it can find a receiver.

4.4 Systems that implement Distributed Scheduling

Examples of systems that implement distributed scheduling, including two research-based, and two real-world systems, are described next.

4.4.1 Case Study 1: V-System (Stanford)

V-System was a distributed operating system designed in the late nineties. It was one of the two research projects that came up with the notion of distributed scheduling, but did not become a commercial project. It assumed a state-change driven information policy, in which if a system saw any significant changes in load (memory utilization or CPU utilization), it broadcast its load information to all nodes. So every node had an insight into the load of all systems. Nodes sent these broadcast messages asynchronously. 'M' least loaded system were receivers, and all the other nodes were senders.

The distributed scheduling policy was sender-initiated. Given that the load information may be stale, the location policy involved polling a random receiver from the list of receivers, and confirming whether it was still a receiver. Load information could have been stale because a node that was a receiver in the past may have received jobs and become much more loaded. V-Systems had no process migration, so only new tasks were send to other systems.

4.4.2 Case Study 2: Sprite (Berkeley)

Sprite was also a distributed operating system. It was developed at Berkeley, around the same time as V-System, and it also had distributed scheduling mechanisms built into it. It was built for an environment where every user had a powerful workstation. It was based on the assumption that the load on these workstations was variable - some were heavily loaded, while others were idle. The idea was to take advantage of systems that are lightly loaded or idle. The system determined which workstations were idle by tracking keyboard/mouse activity. If there was no activity for a certain time period (in this case 30 secs), and the load on the machine was below a certain threshold, the machine was considered idle. Sprite tried to ensure that the owner of the workstation was not impacted by the foreign jobs that were moved to their idle system. So, if a user came back to his/her idle system, it was important that their system not be unresponsive. This was achieved by stopping all background jobs and sending them elsewhere as soon as the workstation became active.

Sprite implemented a centralized policy, in which there was a coordinator that kept track of which systems were idle and which were loaded. The coordinator would allow senders to send jobs to idle machines. Sprite was state-change driven, so as soon as a workstation became idle, it sent a message to the coordinator that it was willing to accept jobs. The location policy required the sender to query the coordinator, and the coordinator would tell the sender where to send jobs.

Process migration was a new feature that Sprite added. If the user of a workstation became active, then the foreign jobs executing on that workstation could be migrated to another workstation. In case of Sprite, the file system was distributed and available to all workstations. In order to migrate a process, the state of a suspended process was saved to disk, a new process was created on the receiving machine, the state of the suspended process was loaded from disk to memory on the receiver, and the execution of the migrated process was resumed from where it was suspended. References to open files could be handled fairly easily, given all files were stored on a network server or a distributed file system. The new workstation where the process was migrated had a new IP address, so issues like open sockets were harder to deal with. There were one of two ways this could be handled. A restriction could be put saying process migration would only work for processes that were not doing network I/O, or tunneling could be used to forward packets to the new IP address associated with the migrated process. For now, the assumption is that all the workstations were homogeneous systems.

4.4.3 Case Study 3: Volunteer Computing

Volunteer computing is in use today and is also called Internet scale operating system (ISOS). Users that have idle machines can volunteer their machine's cycles for a good cause. In volunteer computing, when a user as a very large computationally intensive task that requires a large cluster that they do not have, they ask users to donate their systems free cycles. These tasks are generally based on a good cause, which provides an incentive for other users to donate free cycles. Users that donate free cycles download software that runs as a screen-saver on their system. When the system becomes idle, and the screen-saver is activated, it contacts a central server asking for a job. When the user comes back and starts using the system, it deactivates the screen-saver and the task stops running. The user does not have to do anything special to make this happen.

Volunteer computing can be used for embarrassingly parallel tasks like search for extra terrestrial intelligence (SETI@Home) where signals from outer space are scanned for patterns. SETI was the first to do this but there are other systems like BOINC etc. that also use volunteer computing. Application designers looking for free cycles can contribute tasks, contact the coordinator of BOINC, and get free cycles. These application designers could for example be working on drugs for cancer. Users interested in helping these designers in their cause, can volunteer their cycles. Loose form of distributed scheduling where there is a coordinator assigning tasks, which are scheduled on different machines, results are sent back, and machines come and

go. Google SETI@Home and BOINC to learn more.

4.4.4 Case Study 4: Condor

Condor is designed to make use of idle cycles on workstations, or to schedule jobs on a cluster of servers. Condor takes ideas from Sprite implements in LAN environment. An idle machine contacts the Condor coordinator and requests for a job, the coordinator sends a job back. The system executes the assigned job, and if the owner of the machine comes back, the job is suspended. It has support job migration or restarting the job depending on which policy is in place. Condor has a flexible scheduling policy. It can run in a workstation environment or on system of servers. In case of a system of servers, jobs are submitted to a centralized queue, the Condor coordinator takes a job assigns it to a system in the pool. The Sun Grid engine has similar features, and is used to assign jobs on clusters. It is used on the 'Swarm' cluster to assign jobs. Users submit jobs on Swarm, and the distributed scheduler running on the Sun Grid engine decides which of the 100 cores will run a job.