CMPSCI 677 Operating Systems **Spring 2016**

Lecture 4: February 1

Lecturer: Prashant Shenoy *Scribe: Apoorva Rao Balevalachilu*

4.1 Uniprocessor Scheduling and Threads Contd.

In the previous lecture, we learned about User Level Threads and Kernel Level Threads. We begin this one by introducing Lightweight Processes.

4.1.1 Lightweight Processes

Lightweight processes (LWPs) were first introduced in the Solaris OS. LWPs bridge the user level and the kernel level. Many LWPs can be created for every heavy-weight process. A User-level threads package can create/destroy threads and synchronization primitives. Multithreaded applications can create multiple threads and assign these to LWPs. This is a hybrid model in which the application programmer has more flexibility and control over the mapping between User Level Threads and Kernel Level Threads. Each LWP, when scheduled, searches for a ready thread. The scheduling is carried out on two levels. When a LWP thread blocks on a system call, we switch to kernel mode and OS context switches to another LWP.

4.1.2 Scheduler Activation

Scheduler Activation is a mechanism that allows information to flow back and forth between the multithreading application libraries and the kernel.

- Up-Call from Kernel - Information about CPU availability, I/O done status, etc. is conveyed to the user level libraries.
- Library Informs Kernel - User level libraries inform the kernel about thread creation/deletion.

4.1.3 Thread Packages

- C/C++ use POSIX threads or pthreads that are supported on Windows, Mac OS X, and Linux. POSIX defines only an interface that needs to be implemented. This can be implemented as user-level, kernel-level, or via LWPs.

- Java provides language level support for threads. Like Java, the threads can run on any platform. 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 (For Eg. 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, they are distinct to each processor. Other caches (L2, L3) are shared.

In the multicore/multiprocessor 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 time-slice, or does I/O and gives up its time-slice. Once this happens, the kernel runs again and picks the next process, and so on. This happens in parallel on all the processors.

Approaches for implementing multiprocessor scheduling:

- **Central queue:** In this case, the queue is global and is a shared data structure.
  - Whenever a scheduler runs, it picks the next task (i.e. thread or process) from the head of the global queue. Thus, the global queue 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 and this creates a contention for the lock. To overcome this problem, the centralized queue can be distributed to form multiple queues.
- **Distributed queue:** In this case, we have more than one queue for a set of processors. 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 the one which has a lot of I/O bound tasks. To solve this problem, we need to periodically redistribute tasks to ensure load-balancing in distributed queues.
  - 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 effective use of existing task related data/instructions that are already in the core's cache from the previous processing of that task. In a distributed queue approach, if cache affinity is not respected and the task may be assigned to different queue each time. Then we end up having to start with a cold cache.
- **Per CPU Run queue:** In this case, each scheduler has a queue associated with it
  - In case of per CPU Run queue, the time taken to schedule a process for each scheduler will be constant (irrespective of the number of tasks) since the scheduler does not have to acquire a lock to schedule a new process.
  - Cache affinity is also respected in a per CPU Run queue by making the same processor execute the task every time.

Also note that time slices are somewhat longer in multiprocessors as compared to uniprocessors.

4.2.1 Parallel Applications on SMPs

- These are applications in which threads run perfectly 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.
- *Co-scheduling* (by allowing all or none to run) and *smart scheduling* (by raising priority) are two options available if preemption occurs in the middle of a critical section.

4.3 Distributed Scheduling

A distributed scheduler assigns tasks to machines in a system of N machines.

4.3.1 Motivation and Implications

Consider two scenarios:

- *Lightly loaded system*
  - If a new job arrives, then with high probability it will be assigned to a machine that is idle in a lightly loaded system.
  - The probability that the job needs to wait is low. The product of these two probabilities is low.
  - In such a case, a distributed scheduler does not need to move the job elsewhere.
- *Heavily loaded system*
  - If a new job arrives, the probability that at least one machine is idle is low in a heavily loaded system.
  - The probability that the job needs to wait is high. The product of these two probabilities is low.
  - In this case too, there is no need to perform distributed scheduling because there is no other free machine to send the job to.
- Distributed scheduling is not useful in lightly loaded and heavily loaded systems. 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 and CPU utilization.
- **Types of policies:** We need to define policies based on which the scheduler will decide to move load around. Policies could be one of the following:
  - *Static policies* that are hardwired into the system.

  - *Dynamic policies* are based on querying the systems for their loads and deciding what to do based on loads.
  - *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. That 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? When should a process be transferred?
  - Transfer policies are threshold-based policies which are common and easy to implement.

- **Selection Policy:**
  - The transfer policy deals with questions like: Which process to send?
  - Its easier to transfer a new process rather than an old one.
  - Processes with longer remaining execution time should be selected for transfer over ones with shorter execution time because the transfer cost should be less than the execution cost.

- **Location Policy:** Where to send the process? This can have three variations:
  - Polling : All machines can be polled and jobs can be sent to the least loaded machine.
  - Random : A machine can be picked at random.
  - Nearest neighbor: The nearest free machine can be picked.

- **Information Policy:** What information needs to be tracked in the system in order to make the above decisions? This can be demand-driven, state-change driven or time-driven.

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:** These can have three variations
  - Random: A Machine can be picked at random.
  - Threshold : N machines are chosen sequentially and we transfer the job to the first node which has a load below the threshold. If none have jobs below the threshold, the job is not scheduled to a different machine.
  - Polling: 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 instability. To avoid instability, 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.
- **Location Policy:** Based on two variations:
  - Shortest or Polling: The system can poll N machines and look for the machine with the heaviest load above threshold T and relieve its load. Sequential or parallel polling can be done.
  - Threshold: N machines are chosen sequentially. Transfer the job from the first node which has a load above the threshold. If none have jobs above the threshold, do nothing.

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.
  - Any machine whose load goes below the low threshold becomes a receiver.
  - 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 policy will apply. If you become a receiver, the receiver-initiated policy will apply.
- If senders and receivers in a very large system actively try to 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 Case studies for multiprocessor and distributed scheduling

In a multiprocessor environment, handling multiple jobs in a efficient manner becomes absolutely necessary. Load balancing when the machines are either completely idle or overloaded can help achieve maximum resource utilization.

4.4.1 V-System(Stanford)

V-System is a distributed operating system developed at Stanford about 25 years ago. It is a simple instantiation of the sender-initiated distributed scheduling policy for new jobs. It uses a state-change driven information policy, i.e. the machine monitors its load from time to time. If there is a significant change in the CPU/memory utilization, the change is broadcast to all other nodes. From among the M least load nodes acting as receivers, a receiver is chosen randomly. It is probed again to check if it is still a receiver. If it is, the workload is transferred to the receiver else other receivers are probed.

4.4.2 Sprite(Berkeley)

Sprite is a distributed operating system developed at Berkeley. By the late 1990's, due to the low cost workstations, almost every user had one. The rationale is that if a machine is idle its CPU cycles can be utilized by other users to run their jobs. At the same time, if the user returns to his or her machine to execute a job, he or she should be able to reclaim ownership of the machine and get the highest priority. It is an instantiation of a receiver-initiated distributed scheduling policy with a centralized coordinator keeping track of state-change information. The operating system monitors the activity on the machine:

- During the period of inactivity (no keyboard or mouse input for 30sec).
- When the number of active processes on the machine is less than the number of processors (or threshold).

When these criteria are satisfied, the machine assumes the role of a receiver. Due to the presence of a centralized coordinator, there is no need for the receiver to probe other machines to find jobs to be executed. The coordinator assigns jobs to receivers. When the user returns to his/her workstation, the foreign process running on the machine is stopped and migrated to another machine. In Sprite even partially run processes can be migrated. Process migration in Sprite is facilitated by the Sprite File System (because Sprite has shared disks).

In order to transfer the sender's state to the receiver,

1. All process information is swapped out to the disks
2. Page tables and file descriptors are sent to the receiver.

3. Pages are swapped in from the disks to the receiver.

The only dependencies are communication related. Communication from the home workstation to the receiver has to be redirected.

Sprite does not support live migration. So the process has to follow a stop - migrate - resume execution cycle.

4.4.3 Volunteer Computing

Volunteer computing is a type of distributed computing in which computer owners donate their computing resources (such as processing power and storage) to one or more internet scale projects.

It all started with the project called SETI@home whose purpose is to analyze radio signals and search for signs of extra terrestrial intelligence. SETI@home first introduced the concept of the internet scale operating system, in which the computing power of thousands of PCs owned by individuals could be harnessed.

In Volunteer Computing, a parallel application is partitioned into small tasks and each of these tasks is assigned to different PCs redundantly by a centralized coordinator.

SETI@home implemented a screen saver mechanism that detected period of inactivity on the machine and contacted the coordinator for work. Once the user logged back on, the screen saver deactivated itself and the user regained complete control of the machine. Other examples are BOINC, P2P backup, etc.