CMPSCI 677 Operating Systems **Spring 2015**

Lecture 4: February 4

Lecturer: Prashant Shenoy *Scribe: Seema Guggari*

4.1 Threads

4.1.1 Why Threads?

Threads allow you to have concurrency within a process. Multiple entities called threads execute a process and share the same address space. However, each of the threads have their own stack, program counter, registers etc.

- **Single threaded process:** Traditional processes are single threaded process. If we do any blocking system calls, if suppose an I/O event, then the process blocks. No parallelism is achieved through a single threaded process.
- **Finite-state machine [event based]:** Processes can be event driven systems which have non blocking mode of writing processes . The thread waits until it receives an event, calls the relevant event handler and executes in conjunction with the the thread which receives other events. These are more efficient than single threaded processes achieving parallelism, but difficult to program.
- **Multi-threaded process:** Multiple threads are running in a single process. If one thread blocks, the other thread can proceed . Hence, the notion of parallelism can be achieved on a multiple processors through a multi-threaded process. Thus, threads retain the idea of sequential processes with blocking system calls and yet achieve parallelism

4.1.2 Multi-threaded Server Example

- Threading is also used in Servers and "Multi-threaded Server " is an example of the same.
- Example : **Apache Web Server** It has a pool of pre-spawned worker threads The way to construct a multi-threaded server is by constructing a master thread called as dispatcher thread that listens to requests from each of the clients over a network. Whenever the dispatcher thread receives a request, the request is dispatched to a worker thread if there is a worker thread which is idle. Otherwise the requests are queued by the dispatcher thread. The worker thread processes the request. Things like making a HTTP request, sending back a HTML response is done by a worker thread.
- Concurrency is achieved through this, since multiple requests arriving concurrently are dispatched to worker threads which are handled in parallel.
- The pool of threads can be spawned either using
  - **Static pool management** It consists of pre-spawned set of worker threads.
  - **Dynamic pool management** Server can spawn threads as and when requests come in
    - If suppose there are too many threads in the queue, the dispatcher thread may decide to spawn additional worker threads.

    * And if there are few threads in queue, with many worker threads idle, the dispatcher may decide to terminate some of the threads.

Issues that come up in the context of thread management

- Creation and Deletion of Threads
- Achieving synchronization among multiple threads
- Handling critical section regions of threads
- Whether to create User or Kernel-level threads

4.1.3 User Level Threads

User level threads are managed by the user level library. Operating System is not aware of threads and does not deal with threads. Each process is viewed by the OS as a single threaded process, which is managing the notion of threads in a process. Here threads are implemented in the user space without any support from the Operating System kernel

- **Advantages**
  - It is more efficient, since all the calls to threads are function calls(which are more efficient) to library which are associated to a process.
  - One can achieve flexibility in scheduling the threads in an application , since corresponding scheduling algorithm can be used which is appropriate for the application.
- **Disadvantages**
  - If any thread of a process blocks, the entire process blocks since Operating System is not aware of threads executing in the process.
  - One can achieve concurrency , but not parallelism i.e. threads of the same process cannot run on multiple processors
  - Need to avoid blocking system calls since if one threads blocks, all the threads of a process blocks.

4.1.4 Kernel Level Threads

Threading support is built into the Operating System. OS is aware of the presence of threads inside a process. Here the scheduler schedules all the threads of a process.

- **Advantages**
  - One can achieve true parallelism through kernel level threads. If there are multiple threads and multiple processors, all of them can be executed in parallel.
  - A process can make progress even if one thread of a process blocks, the remaining threads of a process can be executed , since OS is aware of the threads in a process.
- **Disadvantages**
  - Here the threads are bound by CPU scheduling algorithms since they are implemented by the kernel.
  - Thread management is heavy weight and thus turns out to be expensive for uni processors

4.1.5 Light Weight Processes

- Light Weight processes tries to balance the pros and cons of user and kernel level threads. It takes good ideas from both user and kernel level threads and merges both of them.
- Every application process gets mapped onto a light weight process inside a kernel. For example
- Mapping is under the control of the application programmer (which can be static or dynamic). For example, if there are three threads, all the three threads get mapped onto a light weight process or each thread can be mapped onto a light weight process.
- Mapping can be done based on how much concurrency or parallelism , one wants to achieve. . Example: one-to-one, one-to-many etc. Full parallelism is achieved through one-to-one mapping.
- The reason its called a light weight process , since the kernel does not have to allocate or create a new data structure . It is shared with the application process.
- Scheduling Mechanism in light weight process is two level. In the first level, the application programmer schedules each of the threads to a light weight process. Each light weight process is further scheduled onto a processor by the scheduler.

4.1.6 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, they are distinct to each processor. 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.

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 must 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 that load-balancing in distributed queue.
  - 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. In a distributed queue approach, cache affinity is not respected since the task may be assigned to different queue each time and have to start with cold cache. Time slices are somewhat longer to allow caches to warm up.
- **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 lock to schedule a new process.
  - Cache affinity is also respected in a per CPU Run queue by making the task to be executed by the same processor each time.

4.2.1 Parallel Applications on SMP's

- 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

Consider two scenarios:

- *Lightly loaded system*
  - If a job comes in, then with high probability it will be assigned to a machine that is idle.
  - Also, the probability that the job is idle is low.
  - In such a case, a distributed scheduler is not really needed to move the job elsewhere.
- *Heavily loaded system*
  - If a job comes in, the probability that the machine is idle is low.
  - And thus, the probability that the job is idle is high
  - 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 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 at CPU 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. 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?
  - 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 easy to transfer new process over an old process
  - Also, the process with longer execution time should be preferred since the transfer cost should be less than the execution cost
- **Location Policy:** Where to transfer the process? . These can have three variations:
  - Polling : Machines can be polled and jobs can be sent to the least loaded machine
  - Random : A machine can be picked at random
  - Nearest neighbor: A machine can be picked which is closest to it
- **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:** These can have three variations
  - Random: A Machine can be picked at random.
  - 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 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.
  - Threshold : n machines are chosen sequentially. Transfer the job to the first node which has jobs below the threshold. If none have jobs below the threshold, the job is not scheduled to a different machine.

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:** Based on two variations:
  - Shortest or Polling: The system can poll n machines and look for the heaviest load above threshold T, and relieve its load by asking for jobs. Sequential or parallel polling can be done.
  - Threshold: n machines are chosen sequentially. Transfer the job from the first node which has jobs 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. 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.