CMPSCI 677 Operating Systems **Spring 2022**

Lecture 7: February 14

Lecturer: Prashant Shenoy *Scribe: Mrinal Tak, Srividhya Pattabiraman(2023)*

7.1 Overview

In the previous lecture, we learned about the different kinds of threads, specifically user-level threads and kernel-level threads. In this lecture we will cover:

Thread Scheduling

Scheduler Activations

Lightweight process

Multiprocessor scheduling

Distributed scheduling

Case Studies : V-System, Sprite, Volunteer Computing, Condor

Cluster Scheduling

7.2 Thread Scheduling Example

Here, we assume that CPU scheduler uses round-robin time slices. We are given that the program consists of two functions: foo() and bar().

Case 1. No I/O involved: For a single threaded process, sequential execution will first complete the execution of foo() and then after the completion of foo(), the execution of bar() starts. In case of multithreaded process, two separate threads will execute the functions foo() and bar() concurrently and the execution will be interleaved. In terms of completion time, there will be no improvement.

Case 2. I/O involved: Here we will see the advantage of running a multi-threaded process. In case of sequential run, the CPU will be idle and there will be no computation. But in case of multi-threaded process, execution time will shrink, as I/O does not block CPU. If we run the multi-threaded process on multiple cores, the I/O does not help us run extra computation, but the execution time shrinks due to true parallelism.

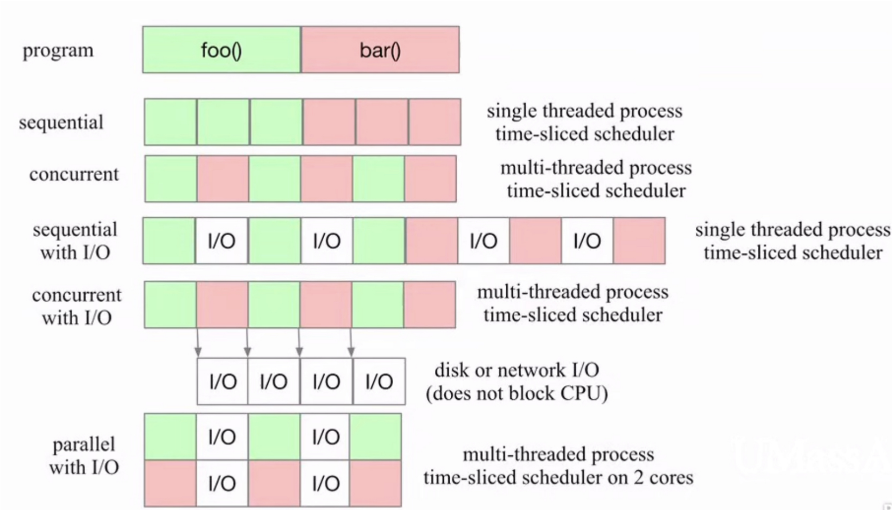

Figure 7.1: Example of thread scheduling.

7.3 Scheduler Activations

Scheduler activations are a threading mechanism which give the performance of user-level threads with the behavior of kernel threads. There is active cooperation and exchange of information between the user-level thread scheduler and the OS to get better scheduling. The same idea is also present in lightweight processes.

7.4 Lightweight Processes

Lightweight processes are a schedulable entity which exist on a layer between kernel-threads and user-threads. They are managed by the kernel. One actual process may use multiple LWPs. Each LWP is bound to a kernel thread. User-level threads can be flexibly mapped to LWP. The mapping is decided by the developer and can be $n : m$, where n threads in user-level are mapped to m threads in kernel-level.

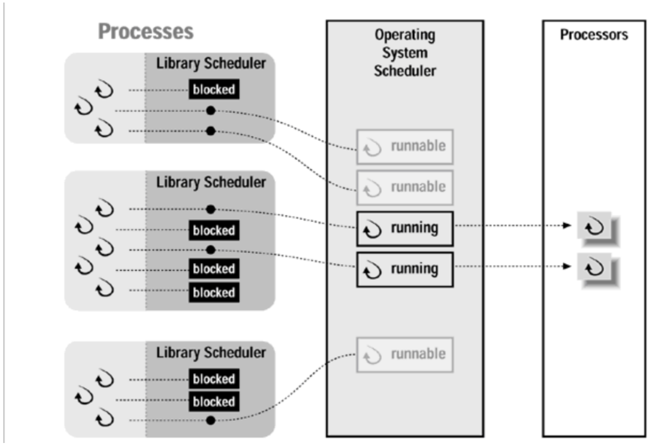

Figure 7.2: Lightweight processes.

7.5 Multiprocessor scheduling

Here, we will talk about single machines with shared memory multiprocessor or multi-core CPU. Looking at the diagram below, the circles at the top are processors. Caches are present at each of the processors which will speed up the performance of anything that is executing on those processors. They keep instructions or data, and are used to speed up the execution of the program. There might be more than two level of caches (L1, L2, L3). Some caches are shared; others are dedicated to processors. Memory, or RAM, are shared across all of the processors using system bus (represented by the blue line in the diagram); a program running on one processor can access any address in memory. Multi-processor scheduling involves scheduling tasks in such environment.

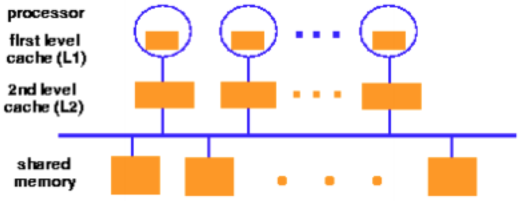

Figure 7.3: Multiprocessor scheduling.

7.5.1 Central queue implementation

In a central queue, all of the processors share a single global queue or run queue where all the threads and processes are present. Execution of a process happens one quantum at a time and whenever the time slice on any processor is going to end, that processor will look at the ready queue, pull a thread or process from that queue, and schedule it. This is essentially what happens in a uni-processor system.

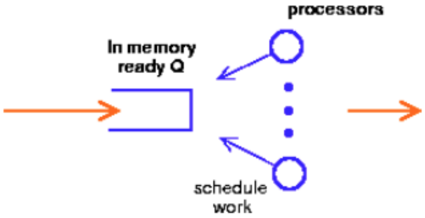

Figure 7.4: A central queue.

7.5.2 Distributed queue implementation

In a distributed queue, there are more than one queue in the system. The processes or threads are going to be part of one of those queue. When a processor becomes idle it is going to look at its local queue, and only take the next job in that queue to schedule for execution one quantum at a time.

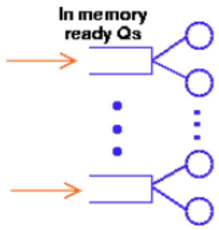

Figure 7.5: A distributed queue.

7.5.3 Pros and cons of centralized queue and distributed queue

The centralized queue is just shared data structure. As the number of processors grows, we face the problem of **synchronization bottleneck**. We have to use some synchronization primitives - the standard one being a lock. When the time slice on one processor ends, that processor has to go to the queue, lock the queue, and pick a job. If another processor becomes idle during this, this processor has to wait until the previous processor has picked the job and released the lock. No matter how efficient synchronization primitives are, there is still going to be synchronization overhead. That overhead will grow as the number of processors increases because there is going to be **lock contention**. There will be a lot of idle time-slices which will be wasted.

Distributed queue experiences less contention. When there is one queue per processor, there is hardly contention at all. The processor just goes to its local queue and pull a thread. Multiprocessor scheduling wouldn't be impacted by the # of processors in a system.

But distributed queue needs to deal with **load imbalance**. Suppose there are n tasks. The n jobs will be split across the distributed queues. How splitting occur matters - if the queues are not equally balanced then some tasks are going to get more CPU's time than others. Tasks in shorter queues get more round-robin timeshare. To deal with this, every once in a while we need to re-balance this queue if there is imbalance in the load by looking at all the queues and their lengths, and equalize them again so that every process gets approximately fair share of CPU time.

Cache affinity is important. Respecting cache affinity can hugely improve time efficiently. A process or thread has affinity to a processor because the cache there holds its data. Let's say processor 1 picks up a job and schedules it for a time slice. By the time that time slice ends, the cache for that processor would be warm, with data and instruction for that job stored in the cache. The process goes to the end of the queue, eventually appears at the front, and get scheduled at another processor. There is no mapping of task to processor in the centralized queue. That processor will start from a cold cache, with no data and instruction for this job. The initial instructions will all be cache misses. Program execution is slowed down to warm up the cache, and then time slice ends. That is why central queue has poor cache affinity.

This argues using distributed queue based scheduling. Once process joins a queue it will stay in the same queue. Next time it gets scheduled there will be data and instructions from the previous time slice i.e. start with warm cache. Once in a while the process could be moved to another process for load balancing where it will start from cold cache but it is not a common scenario. Even though you might try to schedule process or thread on the same processor where it was executed last, every time you run again you still might not have all your data and instructions in the cache due to other processes in your system. You want to have larger time slices or quantum durations to account for the time when there is some early fraction of the quantum maybe all caches misses. You want to still get to run on the caches once it's warmed up for some period of time. As a result, time slices in multiprocessor systems tend to be longer than the ones in uniprocessor systems.

While designing CPU scheduler, Cache Affinity plays the most important part. It should be considered over synchronization bottleneck and lock contention. So, there are couple of things to keep in mind while scheduling on multiprocessors: 1. Exploit cache affinity: Try to schedule on the same processor that a process/thread executed last. 2. Pick larger quantum sizes for the time slicing to decrease context switch overhead.

7.5.4 Scheduling parallel applications on SMP using gang scheduling

One last point on multiprocessor scheduling. Until now, we assumed that the processes/threads are independent. We just picked the one at the front of the queue and we did not care about what was running in other processors. If you think of a parallel process with many many threads or a job with many processes that are coordinating some larger activities, you might not want to schedule them independently. For example, let's say there are two cpu's and a process with two threads. Let's assume when these threads are executed they also need to communicate with each other for application needs. When T1 runs on Processor A, T2 may not be running on Processor B i.e. some other job may be running on Processor B. If T1 sends a message to T2, then it will wait for a reply until processor B runs T2 (so T2 can actually receive the message and process it). There will increase the waiting time.

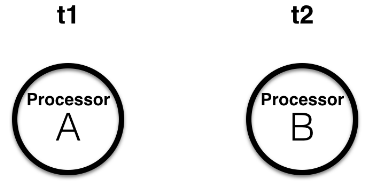

Figure 7.5: Two cpu, and a process with two threads.

Gang scheduling schedule parallel jobs to run on different processors together, as a group in the same time slice. This allows true parallelism, i.e. all the threads of a process can run simultaneously. Gang scheduling is used in special-purpose multi-processors that are designed specifically to execute specialized, massively parallel applications such as scientific jobs.

If one component blocks, the entire applications will be preempted. The remaining $n-1$ timeslices will end and the all the gang components will resume when the other thread gets unblocked. Smart scheduling can take care of this issue too, by adjusting the priority. This will hold off from relinquishing the CPU allowing the other threads to continue to do useful work.

Q: Is there a general purpose scheduling that you would normally run on a parallel machine and switch to gang scheduling only when there is a parallel application?

A: That is technically possible, but the hardware machines that gang schedulers run on are so specialized you would not run general-purpose applications.

Q: What is spin-lock?

A: There are many ways to implement a lock. One form is while the thread is waiting for the lock to get released, it will wait in a loop (”spin) while repeatedly checking if the lock is available. It wastes cycle but it is one way to implement a lock. It is a form of busy waiting. In some parallel application that’s how you implement a lock. Other way is if the lock is not free, the thread blocks the process.

Q: If a machine has four cores and a process runs on two cores, what will happen to the other two?

A: That depends on the scheduling policy of the machine. If it implements gang scheduling, another process that requires two threads and schedule them. If it is general purpose scheduler, you can schedule any other process in the queue.

7.6 Distributed scheduling

7.6.1 Motivation

Consider the scenario in which we have N independent machines connected to each other over a network. When a new application or process arrives at one of the machines, normally the operating system of that machine would execute job locally. In distributed scheduling, you have additional degree of flexibility. Even though the user submit job at machine i, the system may actually decides to execute the job at machine g and bring back the results. Basic reason is the harvest idle cycles on workstations. Referred to as scheduling in a Network of Workstations (NoW). Distributed scheduling - taking jobs that are arriving at one machine and running them somewhere else in the system - does this makes sense (any advantage to this)?

Scenario 1: Lightly loaded system

You should run job locally at the machine where it arrives. No benefit of moving a job because you have enough resources to do it.

Scenario 2: Heavily loaded system

You want to move a job somewhere else but there are no resources available in the system for you to run that job.

Scenario 3: Moderately loaded system

If a job arrives at machine i, and machine i happens to be slightly less loaded than other machines, then run it locally. If machine i happens to be more heavily loaded than other machines, then find another machine to run that job. This scenario would actually benefit from distributed scheduling.

A more technical phrasing of the question would be: *what's the probability that at least one system is idle and one job is waiting?* System idle means that exists some machine that has more resources to actually run more jobs than it is currently running. Job waiting means there is a job that arrived at heavily loaded machine and waiting to run. We benefit from distributed scheduling when both of these cases are true.

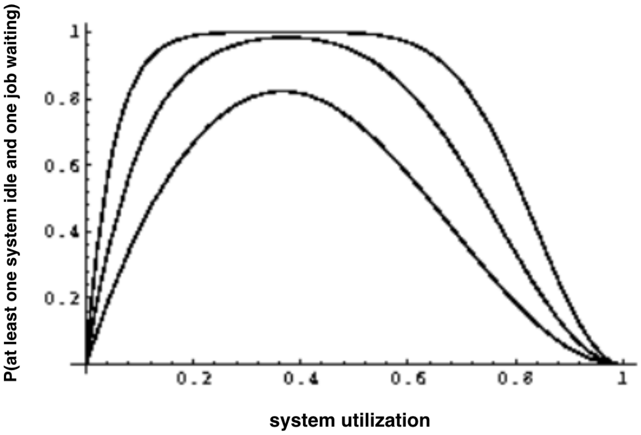

Figure 7.6: The relationship between system utilization and resource under utilization.

- Lightly loaded system:
  P(at least one system idle) high
  P(one job waiting) low
  P(at least one system idle and one job waiting) low
- Heavily loaded system:
  P(one job waiting) high
  P(at least one machine idle) almost zero
  P(at least one system idle and one job waiting) low
- Medium loaded system is the area of high probability under the curve

Note: P(at least one system idle and one job waiting) = P(one job waiting) * P(at least one system idle)

The three lines are constructed by different system parameter settings but the overall curve is the point.

7.6.2 Design issues

Performance metric: mean response time.

Load: CPU queue length and CPU utilization. Easy to measure and reflect performance improvement.

Types of policies:

- In static policy, decisions are hard-wired into scheduling algorithm using prior knowledge of system.
- In dynamic policy, the current state of the load information is used to dynamically make decisions.
- In adaptive policy, parameters of the scheduling algorithm change according to load. Can pick one of static or dynamic policy.

Preemptive vs. Nonpreemptive:

- Preemptive – once job has started executing, scheduler can still preempt it and move somewhere else. Transfer of a partially executed task is expensive due to collection of task's state.
- Nonpreemptive – only transfer tasks that have not begun execution.
- Preemptive schedulers are more flexible but complicated.

Centralized vs. Decentralized:

- Centralized – queue makes decision to send a job globally.
- Decentralized – queue makes decision locally.

Stability: In queuing theory, the arrival rate should be less than system capacity or else the system will become unstable. When arrival rate is reaching system capacity, machines don't want incoming jobs and sends jobs off to other machines which also doesn't want them. Lots of processes are being shuffled around and nothing gets done. Queue starts building up, job floats around, system gets heavily loaded.

7.6.3 Components of scheduler

Distributed scheduling policy has 4 components to it.

i. **Transfer policy** determines **when** to transfer a process to some remote machine. Threshold-based transfer policies are commonly used to classify nodes as senders, receivers, or OK.

ii. **Selection policy** determines **which** task should be transferred from a node. The simplest is to select a newly arrived task that have caused the node to become a sender. Moving processes that have already started executing is more complicated and expensive. The task selected for transfer should be such so that the overhead from task transfer is compensated by a reduction in task's response time.

iii. **Location policy** determines **where** to transfer the selected task. This is done by polling (serially or in parallel). A node can be selected for polling randomly, based on information from previous polls, or based on nearest-neighbor manner.

iv. **Information policy** determines when and from should above information of other nodes be collected - demand-driven, periodic, or state-change-driven - so that the scheduler can make all of its decisions in the right way.

7.6.4 Sender-initiated distributed scheduler policy

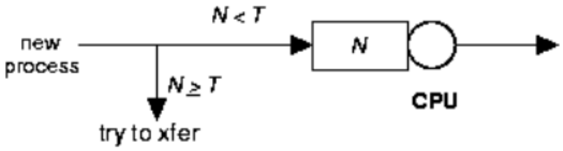

Figure 7.7: A sender-initiated distributed scheduler policy.

Overloaded node attempts to send tasks to lightly loaded node.

- **Transfer policy**: CPU queue threshold, T, for all nodes. Initiated when a queue length exceeds T.
- **Selection policy**: newly arrived tasks.
- **Location policy**:
  - Random: select a random node to transfer the task. The selected node may be overloaded and need to transfer the newly arriving task out again. Is effective under light load conditions.
  - Threshold: poll nodes sequentially until a receiver is found or poll limit has been reached. Transfer task to the first node below threshold. If no receiver is found, then the sender keeps the task.
  - Shortest: poll N_P nodes in parallel and choose least loaded node below T. Marginal improvement.

  There are more sophisticated approaches of finding the node, as the above trivial approaches are not scalable for a high number of nodes. One such approach is, every machine keeps a table which stores the load of other machines. Whenever load on a machine changes, it updates the tables stored in other nodes. In this case the location policy is just a table lookup. The other way to achieve it is to elect a coordinator which maintains the loads of all other nodes, so now there is a central table which maintains the load of other nodes.
- **Information policy**: demand-driven, initiated by the sender.

- **Stability**: Unstable at high-loads

Q: Is the location implemented at a machine level where each machine makes a local decision or can it be implemented globally at a coordinator node or a load balancer which keeps track of every machines' load information and makes decision?

A: Both policies exists. What is shown here is a local machine that is querying but the other policy also exists.

7.6.5 Receiver-initiated distributed scheduler policy

Underloaded node attempts to receive tasks from heavily loaded node.

- **Transfer policy**: when departing process causes load to be less than threshold T, node goes find process from elsewhere to take on.
- **Selection policy**: newly arrived or partially executed process.
- **Location policy**:
  - Random: randomly poll nodes until a sender is found, and transfer a task from it. If no sender is found, wait for a period or until a task completes, and repeat.
  - Threshold: poll nodes sequentially until a sender is found or poll limit is reached. Transfer the first node above threshold. If none, then keep job.
  - Shortest: poll n nodes in parallel and choose heaviest loaded node above T.
- **Information policy**: demand-driven, initiated by the receiver.
- **Stability**: At high loads, a receiver will find a sender with a small number of polls with high-probability. At low-loads, most polls will fail, but this is not a problem, since CPU cycles are available.

7.6.6 Symmetrically-initiated distributed scheduler policy

Sender-initiated and receiver-initiated components are combined to get a hybrid algorithm with the advantages of both. Nodes act as both senders and receiver. Average load is used as threshold. If load is below the low load, node acts as receiver. If load is above high threshold, node acts as sender.

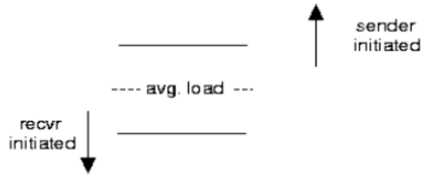

Figure 7.8: A symmetric policy.

Improved versions exploit polling information to maintain sender, receiver, and OK nodes. Sender polls node on receiver list while receiver polls node on sender list. The nodes with load above Upper Threshold(UT) becomes a sender nodes, whereas nodes with load below Lower Threshold (LT) become the receiver nodes.

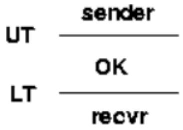

Figure 7.9: An improved symmetric policy.

Q: Isn't there communication overhead?

A: Some; can broadcast your node and keep a table every minute or every time load changes.

Q: How to pick threshold?

A: That's configurable parameter; policy doesn't care.

7.7 Case Study

7.7.1 V-System (Stanford)

Following a state-change driven information policy, the V-System broadcasts information when there is significant change in CPU/memory utilization. Nodes would listen to the broadcast and keep a load table.

The V-System also implements a sender-initiated algorithm which maintains a list of M least loaded nodes. While off-loading, it probes a possible receiver randomly from M and transfers job only if it is still a receiver. When locating a node to receive job transferal, the system needs to double check whether the node on the list of M least loaded nodes is still a receiver because the load table is typically not updated super frequently unless policy did on-demand polling. One job is off-loaded at a time. V-system doesn't implement a symmetrically initiated distributed scheduling policy. The machine could either be a sender or a receiver. The middle ground doesn't exist.

Q: Is M constant? Does it change?

A: M is decided based on number of nodes in the environment. M is proportional to the size of the network. Optimum value is picked by the system.

7.7.2 Sprite (Berkeley)

Sprite assumes a workstation environment in which ownership is king. When users are on their desktop they own it, and no other tasks will run on it. Other tasks can only run on desktops when there is no user using it. Like V-System, Sprite also uses state-change driven information policy and implements a sender-initiated scheduling algorithm.

There is a **centralized coordinator** which keeps the status of load on all the machines. They made an assumption that if there is no mouse or keyboard activity for 30 seconds and the number of active processes is less than a threshold meaning there is at least one processor idle the node becomes a receiver. Foreign processes get terminated when the user returns.

Sprite implemented **process migration** so that the process can suspend running job on a node and continue running from its last state at some other node, instead of killing and restarting the job. Process migration first stops a running job, writes out all registers and memory contents to disk, and transfers the entire memory contents as well as kernel state to receiver machine. Process migration is fairly complicated and comes with many problems e.g. I/O, network communication. Sprite comes at restriction and can only migrate certain types of process.

How does process migration works? It says process is a program executing in memory. To migrate the process, take the memory contents and move it to another machine. Since there is no shared memory, we use the distributed file system as intermediary. This will not be possible if a process is doing network communication because the IP address is tied to the machine and you can't move the process in the middle to another machine because the new machine will have a different IP address and all the socket connections will be broken.

Steps:

1. Suspend the execution of the process, so that its memory contents no longer change.

2. Copy the contents from memory to disk.

3. On the other machine, start paging in. Since the file system is shared and not the memory, we can load the process into receiver's memory.

Q: Are the jobs exclusively CPU, or they can do I/O too?

A: They can do I/O. There is filesystem shared across all machines in some centralized server.

Q: What if everybody goes to lunch (leave the machine idle) and comes back to work (use machine) at the same time?

A: There can be a queue of jobs awaiting execution, so they can be run on idle nodes when users are away.

Q: Do processes declare whether they can be migrated?

A: On some degree it depends on what the user wants to do. If the user knows if the process does not require network communication, then yes it can be declared.

Q: Does process migration between two machines require same OS?

A: For Sprite, yes.

7.7.3 Condor (U. of Wisonsin)

The distributed system that still exists today. Condor makes use of idle cycles on workstations in a LAN. It can be used to **run large batch jobs** and long simulations. It has a central job management system, called the **condor master**, which idle machines contact to get assigned waiting jobs. It supports process migration and flexible job scheduling policies. If Condor runs on OS that does not support process migration, then we need to configure condor to not terminate the process when user comes back or terminate the job. The SLURM scheduler on UMass Swarm cluster is an example of this paradigm.

Condor is not an OS, it is a software framework which runs over OS.

7.7.4 Volunteer Computing

It is a distributed scheduling on a very large scale over WAN. Volunteer computing is based on the idea that PCs on the internet can volunteer to donate CPU cycles/storage when not in use and pool resources together to form an internet scale operating system. A coordinator partitions a large job into small tasks and sends them to the volunteer nodes.

Reliabilty is an issue here. So instead of relying on a single machine, the same subtask in run on k different machines and it is made sure that the answer matches. Seti@home, BOINC and P2P backups are examples for this paradigm.

Q: Is latency a factor about which machines are near and which machines are far?

A: The latency is not as big a problem because let's say you do 30 second or one minute computation typical latency anywhere on the network is going to be 100 some millisecond so you can pay hundreds of millisecond cost and run for of 10 seconds.

7.8 Cluster Scheduling

Cluster Scheduling is scheduling tasks on a pool of servers. This is the distributed scheduling in today's world. This differs from distributed scheduling on workstations. We assume:

- Machines are cheap enough that looking for free CPU cycles on an ideal workstation is not required. There is a dedicated pool of servers to run the tasks.
- Servers are more computationally more powerful than workstations.
- The users explicitly submit the jobs on the cluster.

Cluster scheduling can be analyzed with respect to two applications:

- **Interactive Applications**
  These are applications where user has sent a request and is actively looking for a response. In this case response time is an important factor in designing the scheduler, e.g., for web services we can run a web server on a set of machines. When a web request comes in it can be assigned to one of the machines.
- **Batch Applications**
  These are long running computations which require powerful machine. In this case the throughput is optimized, e.g., for running simulations.

7.8.1 Typical Cluster Scheduler

Typically, there are two kinds of nodes in a cluster - dispatcher and worker. A request (or task) follows the following process:

1. The user submits a request to the dispatcher node.
2. The dispatcher node places the task on a queue.
3. When a worker node is available, the dispatcher removes the task from the queue and assigns it to a worker node.
4. The worker node runs the task.

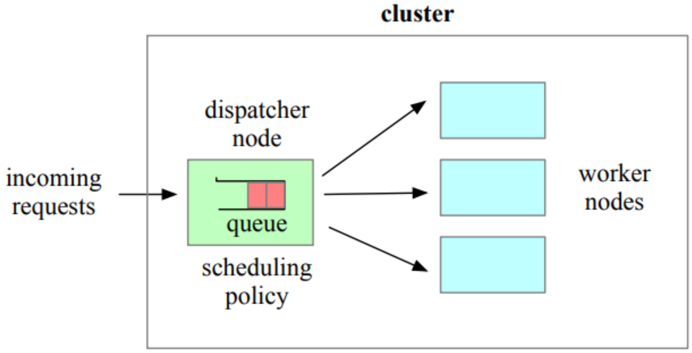

Figure 7.1: A typical cluster scheduler

This design is similar to a thread pool that has listener and worker threads. The scheduling policy depends on the kind of application that has to be run on the cluster.

Question: Would the dispatcher act as single point of failure?
Answer: That would always be a case. To make the system failure tolerant it has to be ensured that in case the dispatcher fails another machine can take over. [1]

7.8.2 Scheduling in Clustered Web Servers

Interactive applications like websites that receive large number of requests are hosted on a cluster. Consider a cluster of N nodes. Here one node acts as the dispatcher, called the *load balancer*. The $(N-1)$ worker nodes are running a replica of the web server and act as the server pool. Any of the worker nodes can service the user requests. The dispatcher receives the request from the user and directs it to one of the worker nodes. The scheduling of the user requests can be done on the basis of:

- The dispatcher can pick up the **least loaded** server.
- The dispatcher can schedule the job in a **round robin** manner. In this case it would not need to get periodic load information updates from the worker nodes.
- If the worker nodes are heterogeneous, **weighted round robin** can be used where you weight a node by the amount of resources and allocate more requests to machines with more computation power.

In case of stateful applications, for eg. an online shopping store the state of the shopping cart is maintained by the machine which served the user request. In this case the request level scheduling won't work as the worker node maintains the session for the user. In these cases **session-level load balancing** is used. When a new web browser starts a session round robin is used to allocate the request to worker node. The server is then mapped to the user and all the subsequent requests by the user are sent to the same machine. One way around this is sending the state back to the client as a cookie which the client has to send it back to the server with every request. This way the state is maintained on the client and not the server. This can allow request level load balancing but would be more expensive as the state updates are sent back to the user.

Question: If you have state, can you keep it in shared memory or database?

[1] A good case study on how this can be achieved is Apache Kafka.

Answer: Shared memory is not a typical OS abstraction. There is no such thing as distributed shared memory that most operating systems support. There can certainly be distributed storage or shared database that can be used to maintain the state. To do so, you'd have to pay the price of I/O to retrieve the state for every request. [2]

7.8.3 Scheduling Batch Applications

This is used for larger jobs like ML training, data processing, simulation. These are long running jobs that can take seconds to as long as hours to finish running. All these jobs come to the dispatcher queue and have to be assigned to a worker node. For this we need a batch scheduler that decides which node to run the job.

One popular batch scheduler available on linux muchines is **SLURM** (Simple Linux Utility for Resource Management). Fundamentally it has a queue and it has a scheduling policy. But it has many other feature:

- It divides the cluster into subgroups. Each subgroup has its own queue.
- This allows user groups with separate scheduling policies for jobs.
- The various queues can put constraints based on runtime of the job. For eg. a queue for short jobs would terminate the job if exceeds the time limit.

[2] For example, Redis can be used as a distributed in-memory cache because of