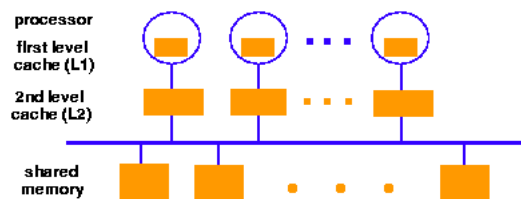


Distributed and Cluster Scheduling

- Part 1: Multiprocessor scheduling
- Part 2: Distributed Scheduling
- Part 3: Cluster Scheduling

Part 1: Multiprocessor Scheduling

- Shared memory symmetric multiprocessor (SMP) or multi-core CPU

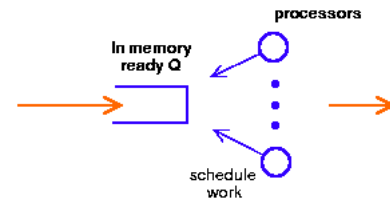


- Salient features: One or more caches: cache affinity is important
 - Semaphores/locks typically implemented as spin-locks: preemption during critical sections
- Multi-core systems: some caches shared (L2,L3); others are not

Multiprocessor Scheduling

- Central queue

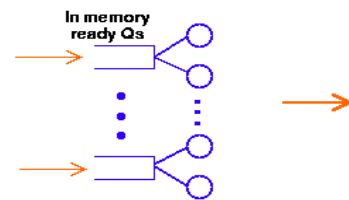
- queue can be a bottleneck;
- utilizes all processors;
- poor cache affinity



- Distributed queue

- imbalance between queues
- load balancing between queue
- good cache affinity

- Exploit *cache affinity* – try to schedule on the same processor that a process/thread executed last



Gang Scheduling

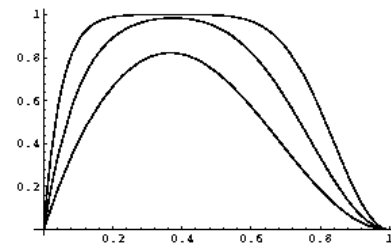
- *Gang scheduling*: schedule **parallel application** at once on all cores/processors
 - Reduces waiting/blocking from message passing/IPC
 - Same idea also applies to a cluster setting
- Effect of spin-locks: what happens if preemption occurs in the middle of a critical section?
 - Preempt entire application (co-scheduling)
 - Raise priority so preemption does not occur (smart scheduling)
 - Both of the above

Part 2: Distributed Scheduling

- Distributed scheduling arose in the workstation era
- Workstation on every desk, many idle
 - **Harness idle cycles on workstations**
 - Scheduling in a *Network of Workstations (NoW)*
 - User submits job to local machine
 - OS schedules locally if load is low
 - OS schedules remotely on an idle machine otherwise
- Distributed system with N workstations
 - To understand benefits of the approach:
 - Model each w/s as identical, independent M/M/1 systems
 - Utilization u , $P(\text{system idle})=1-u$

Harnessing Idle Cycles in NoW

- What is the probability that at least one system is idle and one job is waiting?
- High utilization => little benefit
- Low utilization => rarely job waiting
- Probability high for moderate system utilization
 - Potential for performance improvement
 - Distributed scheduling (aka load balancing) useful
- What is the performance metric?
 - Mean response time
- What is the measure of load?
 - Must be easy to measure and reflect performance improvement
 - Queue lengths at CPU, CPU utilization
- Stability: $\lambda > \mu$ => instability, $\lambda_1 + \lambda_2 < \mu_1 + \mu_2$ => load balance
 - Job floats around and load oscillates

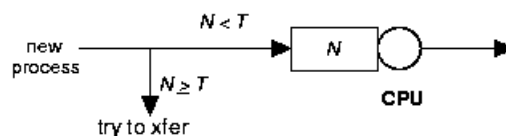


Components

- *Transfer policy*: **when** to transfer a process?
 - Threshold-based policies are common and easy
- *Selection policy*: **which** process to transfer?
 - Prefer new processes
 - Transfer cost should be small compared to execution cost
 - Select processes with long execution times
- *Location policy*: **where** to transfer the process?
 - Polling, random, nearest neighbor
- *Information policy*: when and from where?
 - Demand driven [only if sender/receiver], time-driven [periodic], state-change-driven [send update if load changes]

Sender-initiated Policy

- *Transfer policy*



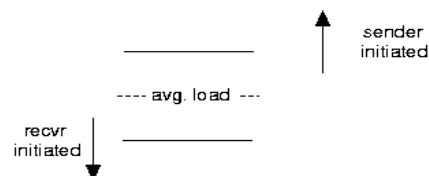
- *Selection policy*: newly arrived process
- *Location policy*: three variations
 - *Random*: may generate lots of transfers => limit max transfers
 - *Threshold*: probe n nodes sequentially
 - Transfer to first node below threshold, if none, keep job
 - *Shortest*: poll N_p nodes in parallel
 - Choose least loaded node below T

Receiver-initiated Policy

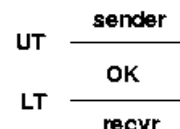
- Transfer policy: If departing process causes load $< T$, find a process from elsewhere
- Selection policy: newly arrived or partially executed process
- Location policy:
 - Threshold: probe up to N_p other nodes sequentially
 - Transfer from first one above threshold, if none, do nothing
 - Shortest: poll n nodes in parallel, choose node with heaviest load above T

Symmetric Policies

- Nodes act as both senders and receivers: combine previous two policies without change
 - Use average load as threshold



- Improved symmetric policy: exploit polling information
 - Two thresholds: LT , UT , $LT \leq UT$
 - Maintain sender, receiver and OK nodes using polling info
 - Sender: poll first node on receiver list ...
 - Receiver: poll first node on sender list ...



Case Study 1: V-System (Stanford)

- State-change driven information policy
 - Significant change in CPU/memory utilization is broadcast to all other nodes
- M least loaded nodes are receivers, others are senders
- Sender-initiated with new job selection policy
- Location policy: probe random receiver from M , if still receiver, transfer job, else try another

Case study 2: Sprite (Berkeley)

- Workstation environment => owner is king!
- Centralized information policy: coordinator keeps info
 - State-change driven information policy
 - Receiver: workstation with no keyboard/mouse activity for 30 seconds *and* # active processes < number of processors
- Selection policy: manually done by user => workstation becomes sender
- Location policy: sender queries coordinator
- WS with foreign process becomes sender if user becomes active: selection policy=> home workstation

Sprite (contd)

- Sprite process migration is a building block for scheduling on to remote machines
 - Facilitated by the Sprite file system
 - State transfer
 - Swap everything out
 - Send page tables and file descriptors to receiver
 - Demand page process in
 - Only dependencies are communication-related
 - Redirect communication from home WS to receiver

Case study 3: Condor

- Condor: use idle cycles on workstations in a LAN
 - Active project at U. Wisconsin, can use even today
- Used to run large batch jobs, long simulations
- Idle machines contact condor for work
- Condor assigns a waiting job
- User returns to workstation => suspend job, migrate
 - supports process migration
- Flexible job scheduling policies

Case Study 4: Volunteer Computing

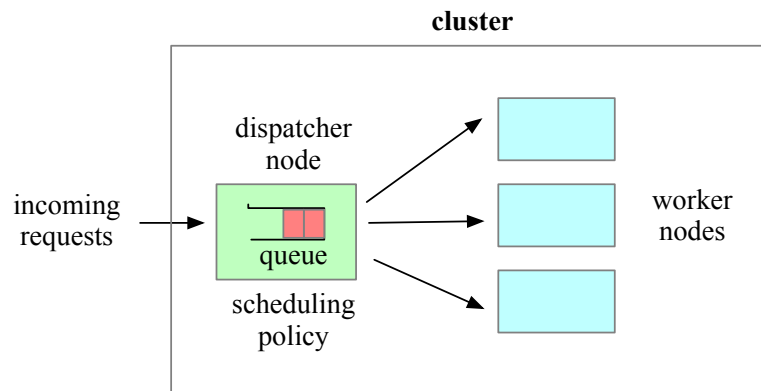
- Modern way to harness idle cycles in PCs over WAN
 - Harness compute cycles of thousands of PCs on the Internet
- Volunteer Computing
 - PCs owned by different individuals
 - Donate CPU cycles/storage when not in use (pool resources)
 - Idling machine contacts coordinator for work
 - Coordinator: partition large parallel app into small tasks
 - Assign compute/storage tasks to PCs
- Examples: [Seti@home](#), BOINC, P2P backups
 - Volunteer computing

Part 3: Cluster Scheduling

- Scheduling tasks on to a cluster of servers
 - Machines are cheap, no need to rely on idle PCs anymore
 - Use a cluster of powerful servers to run tasks
 - User requests sent to the cluster (rather than a idle PC)
- **Interactive** applications
 - Web servers use a cluster of servers
 - “Job” is a single HTTP request; optimize for response time
- **Batch** applications
 - Job is a long running computation; optimize for throughput

Typical Cluster Scheduler

- Dispatcher node assigns queued requests to worker nodes as per a scheduling policy



Scheduling in Clustered Web Servers

- Distributed scheduling in large web servers
 - N nodes, one node acts as load balancer/dispatcher
 - other nodes are replica worker nodes (“server pool”)
- Requests arrive into queue at load balancer node
 - Dispatcher schedules request onto an worker node
- How to decide which node to choose?
 - Scheduling policies: least loaded, round robin
 - Weighted round robin when servers are heterogeneous
- Session-level versus request-level load balancing
 - Web server maintain session state for client (e.g., shopping cart)
 - Perform load balancing at session granularity
 - All requests from client session sent to same worker

Scheduling Batch Jobs

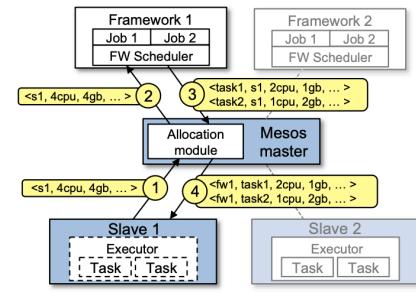
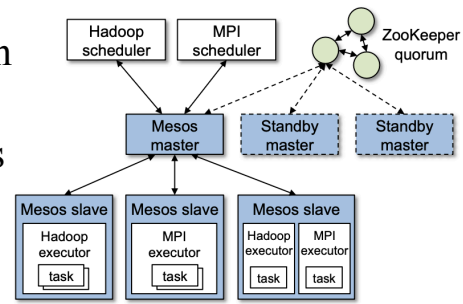
- Batch jobs are non-interactive tasks
 - ML training, data processing tasks, simulations
- Batch scheduling in a server cluster
 - Users submit job to a queue, dispatcher schedules jobs
- SLURM: Simple Linux Utility for Resource Management
 - Linux batch scheduler; runs on > 50% supercomputers
 - Nodes partitioned into groups; each group has job queue
 - Specify size, time limits, user groups for each queue
 - Example: short queue, long queue
 - Many policies: FCFS, priority, gang scheduling
 - Exclusive or shared access to nodes (e.g., MPI jobs)
- Others: SunGridEngine, DQS, Load Leveler, IBM LSF

Mesos Scheduler

- Mesos: Cluster manager and scheduler for multiple frameworks
 - Cluster typically runs multiple frameworks: batch, Spark, ...
 - Statically partition cluster, each managed by a scheduler
 - Mesos: fine-grain server sharing between frameworks
- Two-level approach: allocate resources to frameworks, framework allocates resources to tasks
- **Resource Offers**: bundle of resources offered to framework
 - Framework can accept or reject offer
 - Higher-level policy (e.g., fair share) governs allocation; resource offers used to offer resources
 - Framework-specific scheduling policy allocates to tasks
 - Framework can not ask for resources; only accept/reject resource offers (Paper shows this is sufficient).

Mesos Scheduler

- Four components: **coordinator**, Mesos **worker**, framework **scheduler**, **executor** on server nodes
- Step 1: worker node (6 core, 6GB) becomes idle, reports to coordinator
- Step 2: Coordinator invokes policy, decides to allocate to Framework 1. Sends resource offer
- Step 3: Framework accepts, scheduler assigns task 1 (2C, 2GB) and task 2 (2C, 3GB)
- Step 4: Coordinator sends tasks to executor on node
- Unused resources (2C, 1GB): new offer



Borg Scheduler

- Google's cluster scheduler: scheduling at very large scales
 - run hundreds of thousands of concurrent jobs onto tens of thousands of server
 - Borg's ideas later influenced *kubernetes*
- Design Goals:
 - hide details of resource management and failures from apps
 - Operate with high reliability (manages gmail, web search, ..)
 - Scale to very large clusters
- Designed to run two classes: interactive and batch
 - Long running interactive jobs (prod job) given priority
 - Batch jobs (non-prod jobs) given lower priority
 - % of interactive and batch jobs will vary over time

Borg Scheduler

- Cell: group of machines in a cluster (~10K servers)
- Borg: matches jobs to cells
 - jobs specify resource needs
 - Borg finds a cell/machine to run a job
 - job needs can change (e.g., ask for more)
- Use resource reservations (“alloc”)
 - alloc set: reservations across machines
 - Schedule job onto alloc set
- Preemption: higher priority job can preempt a lower priority job if there are insufficient resources
- Borg Master coordinator: replicated 5 times, uses paxos to
- Priority queue to schedule jobs: uses best-fit, worst-fit

