Multiprocessor Scheduling
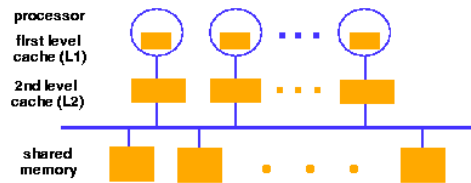
•Will consider only shared memory multiprocessor

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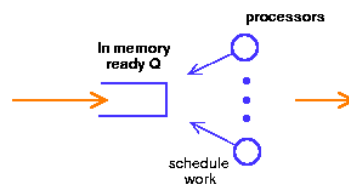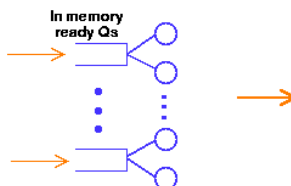
•Multi-core systems: some caches shared (L2,L3); others are not

Multiprocessor Scheduling

•Central queue – queue can be a bottleneck

•Distributed queue – load balancing between queue

Scheduling

- Common mechanisms combine central queue with per processor queue (SGI IRIX)
- Exploit *cache affinity* – try to schedule on the same processor that a process/thread executed last
- Context switch overhead
  - Quantum sizes larger on multiprocessors than uniprocessors

Parallel Applications on SMPs

- 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
- Provide applications with more control over its scheduling
  - Users should not have to check if it is safe to make certain system calls
  - If one thread blocks, others must be able to run

Distributed Scheduling: Motivation

- Distributed system with *N* workstations
  - Model each w/s as identical, independent M/M/1 systems
  - Utilization *u*, P(system idle)=1-*u*
- What is the probability that at least one system is idle and one job is waiting?
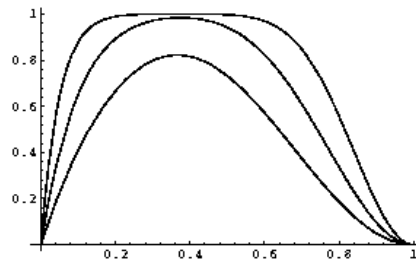
Implications

- Probability high for moderate system utilization
  - Potential for performance improvement via load distribution
- High utilization => little benefit
- Low utilization => rarely job waiting
- Distributed scheduling (aka load balancing) potentially useful
- What is the performance metric?
  - Mean response time
- What is the measure of load?
  - Must be easy to measure
  - Must reflect performance improvement

Design Issues

- Measure of load
  - Queue lengths at CPU, CPU utilization
- Types of policies
  - Static: decisions hardwired into system
  - Dynamic: uses load information
  - Adaptive: policy varies according to load
- Preemptive versus non-preemptive
- Centralized versus decentralized
- 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
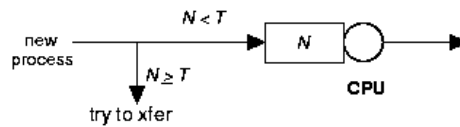
- *Transfer policy*

- *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

```
                              ↑  sender
        _____              initiated
     ---- avg. load ---
     recvr    _____
   initiated  ↓
```
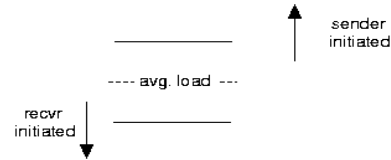
- Improved symmetric policy: exploit polling information
  - Two thresholds: *LT, UT, LT <= UT*
  - Maintain sender, receiver and OK nodes using polling info
  - Sender: poll first node on receiver list …
  - Receiver: poll first node on sender list …

```
         sender
UT    _____
         OK
LT    _____
         recvr
```
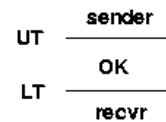
Case Study: V-System (Stanford)

- 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, if still receiver, transfer job, else try another

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
  - 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
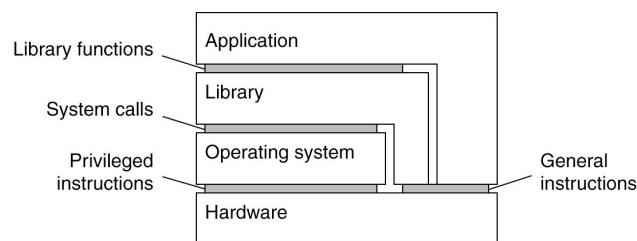   - Redirect communication from home WS to receiver

Virtualization

Program
Interface A
Hardware/software system A

(a)

Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B

(b)

- Virtualization: extend or replace an existing interface to mimic the behavior of another system.
  – Introduced in 1970s: run legacy software on newer mainframe hardware
- Handle platform diversity by running apps in VMs
  – Portability and flexibility

Types of Interfaces

Library functions — | Application |
System calls — | Library |
Privileged instructions — | Operating system | — General instructions
| Hardware |

- Different types of interfaces
  – Assembly instructions
  – System calls
  – APIs
- Depending on what is replaced /mimiced, we obtain different forms of virtualization

Types of Virtualization

- Emulation
  - VM emulates/simulates complete hardware
  - Unmodified guest OS for a different PC can be run
    - Bochs, VirtualPC for Mac, QEMU
- Full/native Virtualization
  - VM simulates "enough" hardware to allow an unmodified guest OS to be run in isolation
    - Same hardware CPU
 - IBM VM family, VMWare Workstation, Parallels,…

Types of virtualization

- Para-virtualization
  - VM does not simulate hardware
  - Use special API that a modified guest OS must use
  - Hypercalls trapped by the Hypervisor and serviced
  - Xen, VMWare ESX Server
- OS-level virtualization
  - OS allows multiple secure virtual servers to be run
  - Guest OS is the same as the host OS, but appears isolated
    - apps see an isolated OS
  - Solaris Containers, BSD Jails, Linux Vserver
- Application level virtualization
  - Application is gives its own copy of components that are not shared
    - (E.g., own registry files, global objects) - VE prevents conflicts
  - JVM

Examples

(a)

(b)
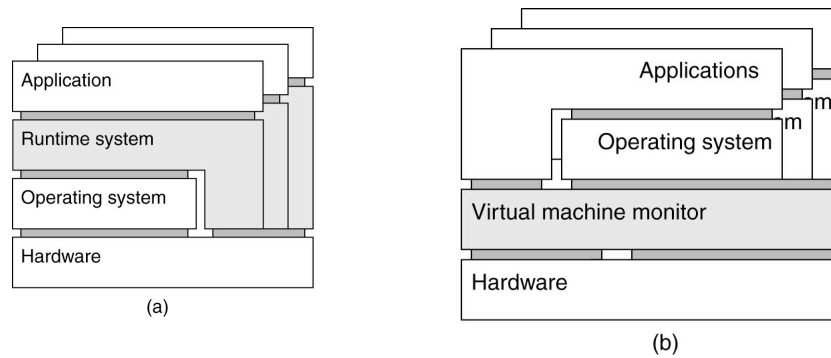
- Application-level virtualization: "process virtual machine"
- VMM /hypervisor