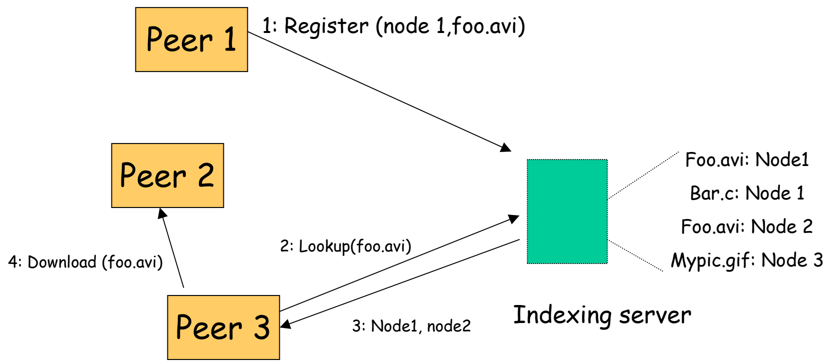
Course Project

- Part 1: Peer-to-peer file sharing with centralized index



Course Project

- Two entities
  - Central indexing server
    - List of all files at peers
  - Peer (both client and server)
    - [client] Search for a file at the indexing server
    - Download file from a peer, update indexing server
    - [server] listen for download requests and service
  - Provide concurrency at the central indexing server and peer
- Feel free to use any prog language and any mechanism (threads, RPC, RMI, sockets, semaphores…)

User-level threads

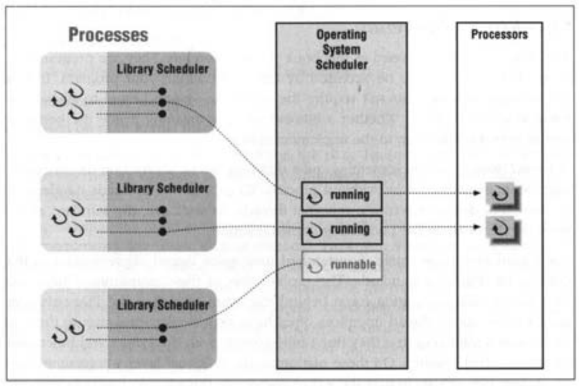


Figure 6-1: User-space thread implementations

Kernel-level threads

- Kernel aware of the presence of threads
  - Better scheduling decisions, more expensive
  - Better for multiprocessors, more overheads for uniprocessors

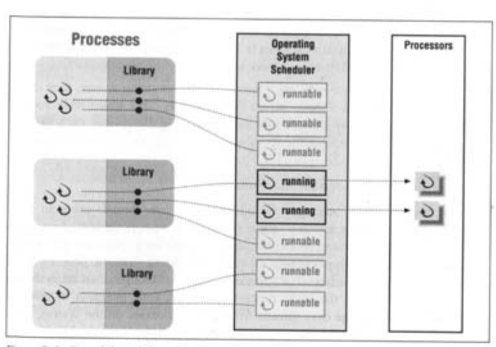


Figure 6-2: Kernel thread-based implementations

Light-weight Processes

- Several LWPs per heavy-weight process
- User-level threads package
  - Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread *[two-level scheduling]*
  - Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

LWP Example

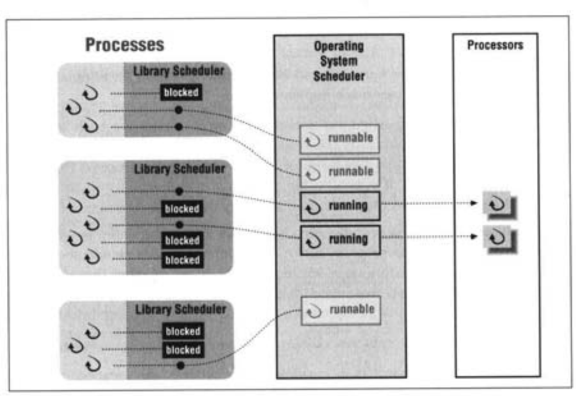


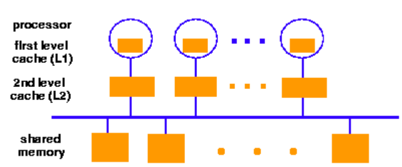
Figure 6-3: Two-level scheduler implementations

Thread Packages

- Posix Threads (pthreads)
  - Widely used threads package
  - Conforms to the Posix standard
  - Sample calls: pthread_create,…
  - Typical used in C/C++ applications
  - Can be implemented as user-level or kernel-level or via LWPs
- Java Threads
  - Native thread support built into the language
  - Threads are scheduled by the JVM

Multiprocessor Scheduling

•Will consider only shared memory multiprocessor

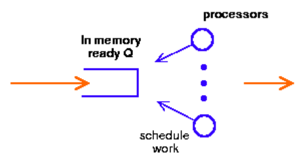


•Salient features:

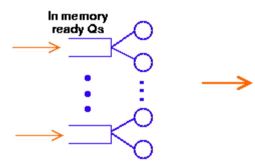
- One or more caches: cache affinity is important
- Semaphores/locks typically implemented as spin-locks: preemption during critical sections

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

Code and Process Migration

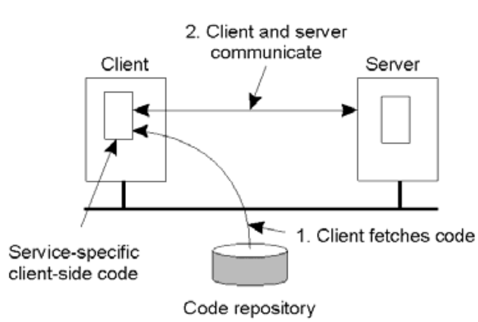
- Motivation
- How does migration occur?
- Resource migration
- Agent-based system
- Details of process migration

Motivation

- Key reasons: performance and flexibility
- Process migration (aka *strong mobility*)
  - Improved system-wide performance – better utilization of system-wide resources
  - Examples: Condor, DQS
- Code migration (aka *weak mobility)*
  - Shipment of server code to client – filling forms (reduce communication, no need to pre-link stubs with client)
  - Ship parts of client application to server instead of data from server to client (e.g., databases)
  - Improve parallelism – agent-based web searches

Motivation

- Flexibility
  - Dynamic configuration of distributed system
  - Clients don't need preinstalled software – download on demand



Migration models

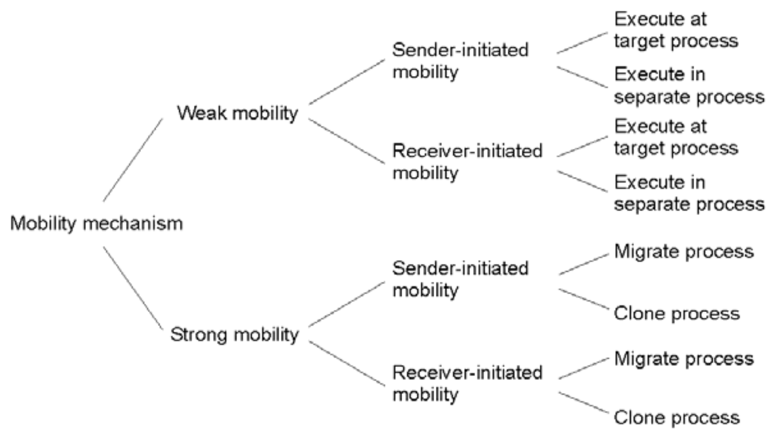
- Process = code seg + resource seg + execution seg
- Weak versus strong mobility
  - Weak => transferred program starts from initial state
- Sender-initiated versus receiver-initiated
- Sender-initiated (code is with sender)
  - Client sending a query to database server
  - Client should be pre-registered
- Receiver-initiated
  - Java applets
  - Receiver can be anonymous

Who executes migrated entity?

- Code migration:
  - Execute in a separate process
  - [Applets] Execute in target process
- Process migration
  - Remote cloning
  - Migrate the process

Models for Code Migration

•



Do Resources Migrate?

- Depends on resource to process binding
  - By identifier: specific web site, ftp server
  - By value: Java libraries
  - By type: printers, local devices
- Depends on type of "attachments"
  - Unattached to any node: data files
  - Fastened resources (can be moved only at high cost)
    - Database, web sites
  - Fixed resources
    - Local devices, communication end points

Resource Migration Actions

Resource-to machine binding

		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

- Actions to be taken with respect to the references to local resources when migrating code to another machine.
- GR: establish global system-wide reference
- MV: move the resources
- CP: copy the resource
- RB: rebind process to locally available resource

Migration in Heterogeneous Systems

- Systems can be heterogeneous (different architecture, OS)
  - Support only weak mobility: recompile code, no run time information
  - Strong mobility: recompile code segment, transfer execution segment [migration stack]
  - Virtual machines - interpret source (scripts) or intermediate code [Java]