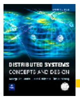
Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.



email: *authors@cdk2.net*

Viewing: These slides must be viewed in slide show mode.

Distributed Systems Course
Operating System Support

Chapter 6:

6.1 Introduction
6.2 The operating system layer
6.4 Processes and threads
6.5 Communication and invocation
6.6 operating system architecture

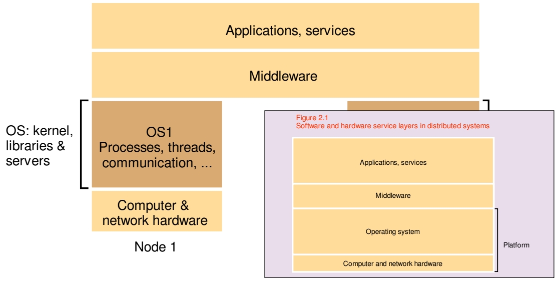
Learning objectives

- Know what a modern operating system does to support distributed applications and middleware
  - Definition of network OS
  - Definition of distributed OS
- Understand the relevant abstractions and techniques, focussing on:
  - processes, threads, ports and support for invocation mechanisms.
- Understand the options for operating system architecture
  - monolithic and micro-kernels

*

System layers

Figure 6.1



*

Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
  - RPC and RMI (Sun RPC, Corba, Java RMI)
  - event distribution and filtering (Corba Event Notification, Elvin)
  - resource discovery for mobile and ubiquitous computing
  - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
  - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)
  - do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

*

Networked OS to Distributed OS

- Distributed OS
  - Presents users (and applications) with an integrated computing platform that hides the individual computers.
  - Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
    - *In a distributed OS, the user doesn't know (or care) where his programs are running.*
  - One OS managing resources on multiple machines
  - Examples:
    - *Cluster computer systems*
    - *Amoeba, V system, Sprite, Globe OS*

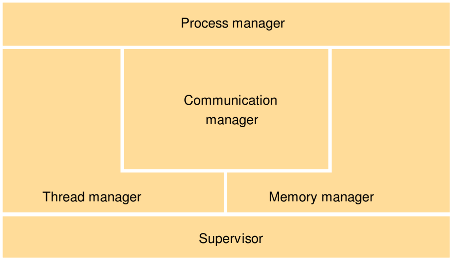
The support required by middleware and distributed applications

- OS manages the basic resources of computer systems
- Tasks:
  - programming interface for these resources:
    - abstractions such as: processes, virtual memory, files, communication channels
    - Protection of the resources used by applications
    - Concurrent processing
  - provide the resources needed for (distributed) services and applications:
    - Communication - network access
    - Processing - processors scheduled at the relevant computers

*

Core OS functionality

Figure 6.2



*

Protection:

- Why does the kernel need to be protected?
- Kernels and protection
  - kernel has all privileges for the physical resources, processor, memory..
- execution mode
  - kernel and user
- address space
  - kernel and user
- user transferred to kernel

  - system call trap
    - try to invoke kernel resources
    - switch to kernel mode
- cost
  - switching overhead to provide protection

Processes and Threads (1)

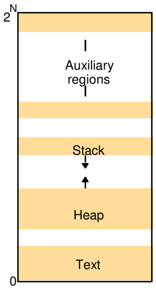
- process has one environment
- thread: activity, "thread" of execution in one environment
- execution environment:
  - an address space
  - synchronization and communication resources
  - i/o resources
- why execution environment?
- threads share one execution environment, why?
- older names: heavyweight and lightweight processes
- Address space

Processes and Threads (2)

- Address space
  - unit of management of a process' virtual memory
- Regions
  - Text, heap, stack
- Each region
  - beginning virtual address and size
  - read/write/exe permissions for the process' threads
  - growth direction
- Why regions:
  - different functionalities, for example:
    - different stack regions for threads
    - memory-mapped file
- Shared memory regions among processes?
  - libraries
  - kernel
  - data sharing and communication

Processes and Threads (3): Process address space

Figure 6.3



*

Processes and Threads (4): process creation

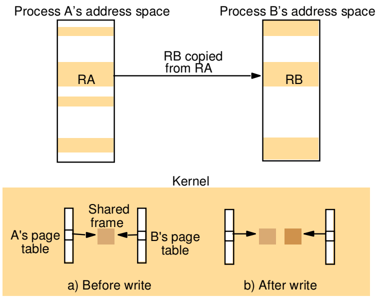
- Distributed OS
  - choice of target host
  - actual creation of execution env
- choice of target host
  - transfer policy: local or remote?
  - location policy: if not local, which host/processor
    - V and Sprite system: user has a command and OS chooses
    - Amoeba: a run server decides, more transparent
- static and adaptive location policies
  - static: predetermined function
  - adaptive: depends on current state of the processing nodes
- load-sharing systems
  - centralized: one load manager
  - hierarchical: tree of managers
  - decentralized: nodes exchange information, local decision
  - sender-initiated
  - receiver-initiated
- process migration
  - moved while it's running
  - what need to be sent?

Processes and Threads (5)

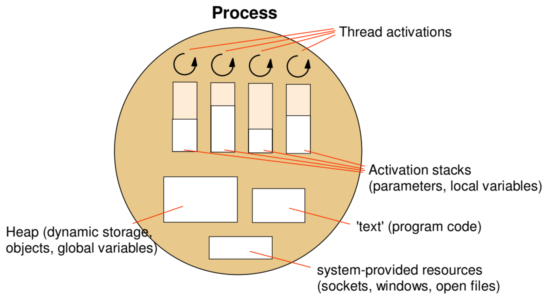
- creating an execution env
  - address space with initial contents
  - initialization
    - *statically defined format from a list of regions*
    - *derived from an existing exe env*
      - fork in unix
      - derived from the parent
      - shares the text region
      - a copy of the heap and stack
      - copy-on-write

Processes and Threads (6): Copy-on-write

Figure 6.4

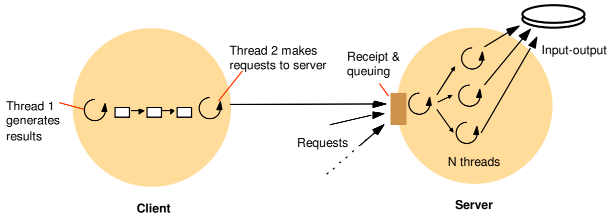


Processes and Threads (7): Thread memory regions



Processes and Threads (8): Client and server

Figure 6.5



The 'worker pool' architecture

See Figure 4.6 for an example of this architecture programmed in Java.

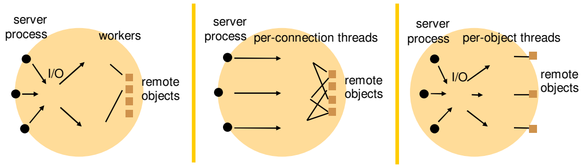
Processes and Threads (9)

- average interval of successive job completions
  - one request: 2 milliseconds of processing and 8 for i/o delay
  - one thread: 2+8 = 10 milliseconds, 100 requests/second

  - two threads: 125 requests/second, serial i/o, why?
  - two threads: 200 requests/second, concurrent i/o, why?
  - two threads with cache (75% hit):
    - *2 milliseconds (.75*0 + .25*8), 500 requests/sec*
  - cpu overhead of caching: 2.5 milliseconds, 400 requests/sec

Processes and Threads (10): server threading architectures

Figure 6.6



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

– Implemented by the server-side ORB in CORBA

(a) would be useful for UDP-based service, e.g. NTP (network time protocol)

(b) is the most commonly used - matches the TCP connection model

(c) is used where the service is encapsulated as an object. E.g. could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.

Processes and Threads (11): Threads vs processes

- Creating a thread is (much) cheaper than a process (~10-20 times)
- Switching to a different thread in same process is (much) cheaper (5-50 times)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

Figure 6.7 State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

*

Processes and Threads (12): Concurrency

- Issues in concurrency:
  - Race condition
  - Deadlock
  - Starvation
- Programming support
  - library (POSIX pthreads)
  - language support (Ada95, Modula-3, Java)

Processes and Threads (13)

- thread (process) execution
  - create/fork
  - exit
  - join/wait
  - yield

Processes and Threads (14)

- Synchronization
  - coordinate current tasks and prevent race conditions on shared objects
  - Critical region: only one thread/process at a time is allowed
  - Why critical regions should be as small as possible?
- Programming support
  - Mutual exclusion
  - Condition Variables
  - Semaphores

Processes and Threads (15): Mutual Exclusion

- Mutual exclusion (mutex)
  - critical region/section
  - before entering critical region, try to lock
  - mutex_lock(l):
    - *if try to lock is successful*
      - lock and continue
    - *else*
      - blocked
  - mutex_unlock(l): release the lock

Processes and Threads (16)

- One producer, one consumer, producer can produce many items (1P1CinfD)
- How about nPnCinfD ?

```
Producer              Consumer
mutex_lock(D)         mutex_lock(D)
                      while (no data) // count <= 0
                        mutex_unlock(D)
                        sleep (how long?  spin poll)
                        mutex_lock(D)
produce one item      consume one item
mutex_unlock(D)       mutex_unlock(D)
```

Processes and Threads (17): Condition Variables

- Condition variable
  - wait for an event (condition) before proceeding
  - Assoicated mutex with the condition
- Waiting for an event

```
1. lock associated mutex m
2. while (predicate is not true)     // "if" could work, but less safe
3.         cv_wait( c, m )
4. do work
5. unlock associated mutex m
```

- Signaling an event

```
1. lock associated mutex m
2. set predicate to true
3. cv_signal( c )          // signal condition variable (wake-up one or all)
4. unlock associated mutex m
```

Processes and Threads (18)

- cv_wait(c, m):
  1. unlock associated mutex m
  2. block thread
  3. put it on the queue to wait for a signal
  4. lock associated mutex m // why?
- cv_signal(c):
  - wake up a thread waiting on the condition

Processes and Threads (19)

```
Producer              Consumer
mutex_lock(D)         mutex_lock(D)
                      while (no data)
                         cv_wait(yesD, D)
produce one item      consume one item
cv_signal(yesD)
mutex_unlock(D)       mutex_unlock(D)
```

Processes and Threads (20): Semaphores

- binary semaphores = mutex
- counting/integer semaphores
  - P(s) [prolagen -- decrease (Dutch)]
    - if s > 0
      - decrement s
    - else
      - blocked
  - V(s) [verhogen -- increase]
    - increment s

Processes and Threads (21): first try

```
Producer                 Consumer
D=1  //initialization
P(D)                     P(D)
                         while (no data)
                            V(D)
                            sleep
                            P(D)
produce one item         consume one item
V(D)                     V(D)
```

- Does this work?
- What are basically P and V used for?

Processes and Threads (22): 1P1CinfD (nPnCinfD)

```
Producer                 Consumer
D=1   // critical region
Count=0   // # of items
```

```
                         P(Count)
P(D)                     P(D)                 // mutex; binary semaphore
produce one item         consume one item
V(D)                     V(D)
V(Count)
```

```
                         P(Count)
P(D)                     P(D)                 // mutex; binary semaphore
produce one item         consume one item
V(Count)
V(D)                     V(D)
```

```
P(D)                     P(D)                 // mutex; binary semaphore
                         P(Count)
produce one item         consume one item
V(Count)
V(D)                     V(D)
```

Processes and Threads (23)

- Versions 1 and 2 work
  - Version 1 has more concurrency
- Versions 3 doesn’t work
- Exercises
  - One producer, one consumer, one data item (1P1C1D) [lock steps, PCPC...]
  - One producer, one consumer, up to n data items (1P1CnD) same for nPnCnD

Processes and Threads (24)

- What could happen here?

```
mutex_lock(B)        mutex_lock(A)
mutex_lock(A)        mutex_lock(B)
do work              do work
mutex_unlock(B)      mutex_unlock(A)
mutex_unlock(A)      mutex_unlock(B)
```

- How to prevent the problem?

Processes and Threads (25): Scheduling

- Preemptive
  - a thread can be suspended at any point for another thread to run
- Non-preemptive
  - a thread can only be suspended when it de-schedules itself (e.g. blocked by I/O, sync...) [critical region between calls that de-schedule]

Processes and Threads (26): Thread Implementation

- Kernel-level
  - Win NT, Solaris, Mach, Chorus
  - Kernel schedules threads
- User-level
  - library based (pthreads, or in the language like java)
  - run-time system in user space manages threads
  - Kernel schedules processes
- Disadvantages of user-level threads
  - can't take advantage of multiprocessors
  - one thread is blocked because of a page fault, the process is blocked, all the others threads in the same process are blocked
  - threads in different processes aren't scheduled in the same environment
- Advantages of user-level threads
  - less costly to manage
  - scheduling can be customized
  - more user-level threads can be supported

Processes and Threads (27)

- Mixed
  - Mach:
    - user-level code to provide scheduling hints to the kernel
  - Solaris:
    - assign each user-level thread to a kernel-level thread (multiple user threads can be in one kernel thread)
    - creation/switching at the user level
    - scheduling at the kernel level

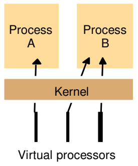
Processes and Threads (28)

- FastThread package
  - hierarchical, event-based scheduling
  - each process has a user-level thread scheduler
  - virtual processors are allocated to processes
    - the # of virtual processors depends on a process's needs
    - physical processors are assigned to virtual processors
    - virtual processors can be dynamically allocated and deallocated to a process according to its needs.
  - Scheduler Activation (SA)
    - event/call from kernel to user-level scheduler
    - represents a **time slice** on a virtual processor (# of SA's < # of virtual processors)
    - user-level scheduler can assign threads to SA's (time slices).

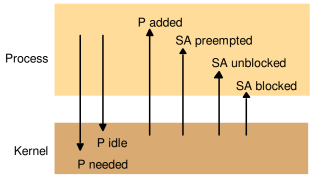
Processes and Threads (29)

- Events from user-level scheduler to kernel
  - *P idle*: virtual processor is idle
  - *P needed*: new virtual processor needed
- Events from kernel to user-level scheduler
  - *virtual processor allocated (P added)*:
    - User-level: scheduler can choose a ready thread to run
  - *SA blocked* (in kernel):
    - Kernel: sends a new SA
    - User-level: a ready thread is assigned to the new SA to run
  - *SA unblocked* (in kernel):
    - user-level: thread is back on the ready queue
    - kernel: allocate new virtual processor to process or preempt another SA
  - *SA preempted* (in kernel):
    - user-level: puts the thread back to the ready queue

Processes and Threads (30): Scheduler activations



A. Assignment of virtual processors to processes



B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

Skip Sections 6.5 and 6.6