Unit 4 - Distributed Operating Systems

Processes and Processors in Distributed Systems

Processes and processors in distributed systems

- Threads Concepts
- Design Issues of Threads Package
- System model -Work Station Model, Processor Pool & Hybrid Model
- Processor Allocation Algortihms
- Scheduling in distributed systems
- Load balancing and Sharing approach
- Fault tolerance
- Real time distributed systems,

Unit 4 - Distributed Operating Systems

Threads Concepts

- In most traditional OS, each process has an address space and a single thread of control.
- It is desirable to have multiple threads of control sharing one address space but running in quasi-parallel.

Introduction to threads

- Thread is a **light weighted process**.
- The analogy: thread is to process as process is to machine.
- Each thread runs **strictly sequentially** and has its own **program counter** and **stack** to keep track of where it is.
- Threads **share the CPU** just as processes do: first one thread runs, then another does.
- Threads can create **child threads** and can block waiting for system calls to complete.

Threads

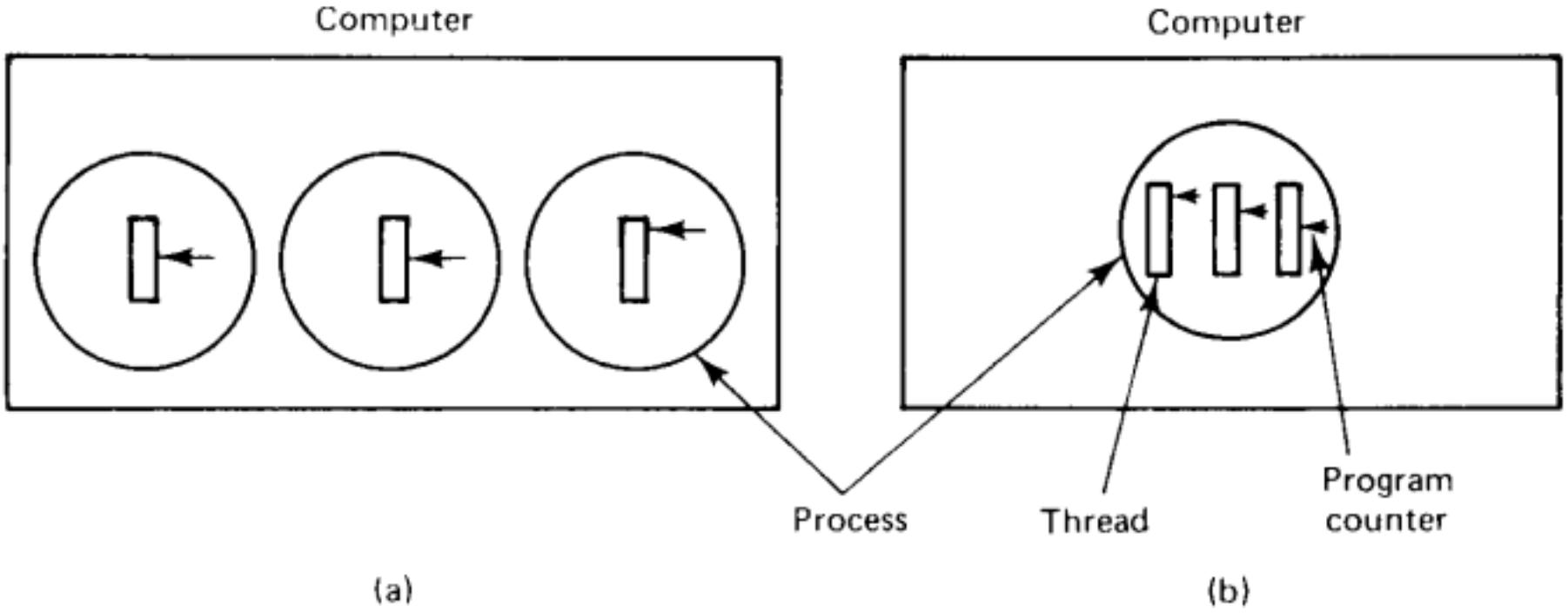


Fig. 4-1. (a) Three processes with one thread each. (b) One process with three threads.

Introduction to threads

- All threads have exactly the **same address space**.
- They share **code section, data section, and OS resources** (open files & signals).
- They share the same **global variables**.
- One thread can read, write, or even completely wipe out another thread's stack.
- Threads can be in any one of several states: **running, blocked, ready, or terminated.**

Introduction to threads

- There is no protection between threads:

(1) it is impossible

(2) it should not be necessary: a process is always owned by a single user, who has created multiple threads so that they can cooperate, not fight.

- In addition to sharing an address space all threads share :
  - Set of open files
  - Child processes
  - Timers and Signals etc.

Introduction to threads – Per Thread and Per Process concepts

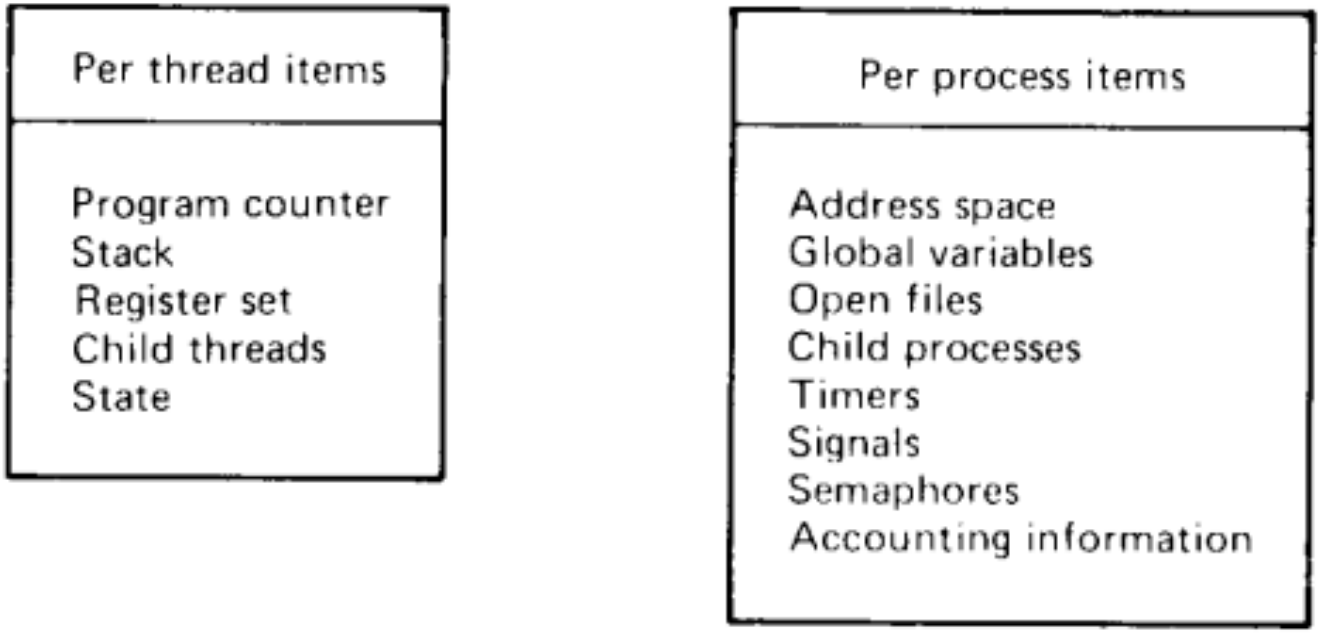


Fig. 4-2. Per thread and per process concepts.

Thread usage

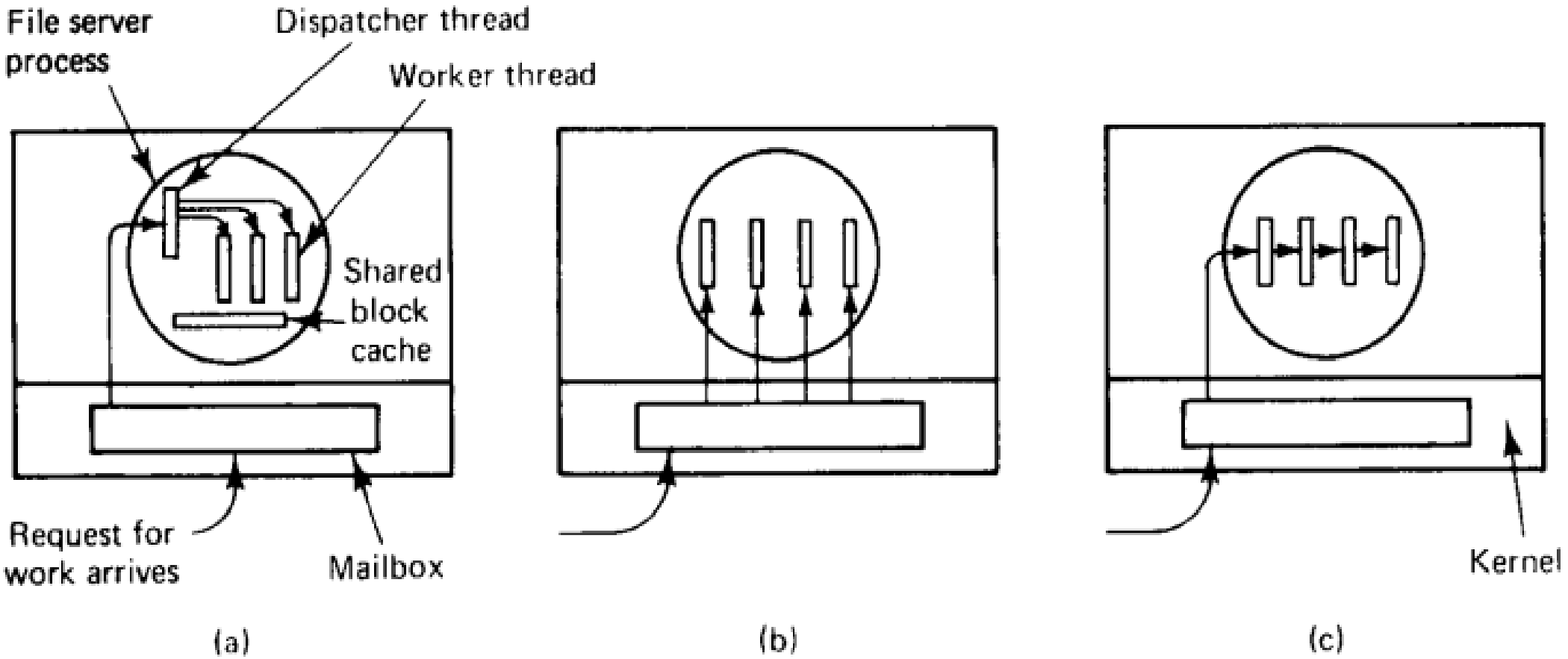


Fig. 4-3. Three organizations of threads in a process. (a) Dispatcher/worker model. (b) Team model. (c) Pipeline model.

Advantages of using threads

1. Useful for clients: if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server.
2. Handle signals, such as interrupts from the keyboard. Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting for signals.
3. Producer-consumer problems are easier to implement using threads because threads can share a common buffer.
4. It is possible for threads in a single address space to run in parallel, on different CPUs.

Unit 4 - Distributed Operating Systems

Design Issues of Threads Package

Design Issues for Threads Packages

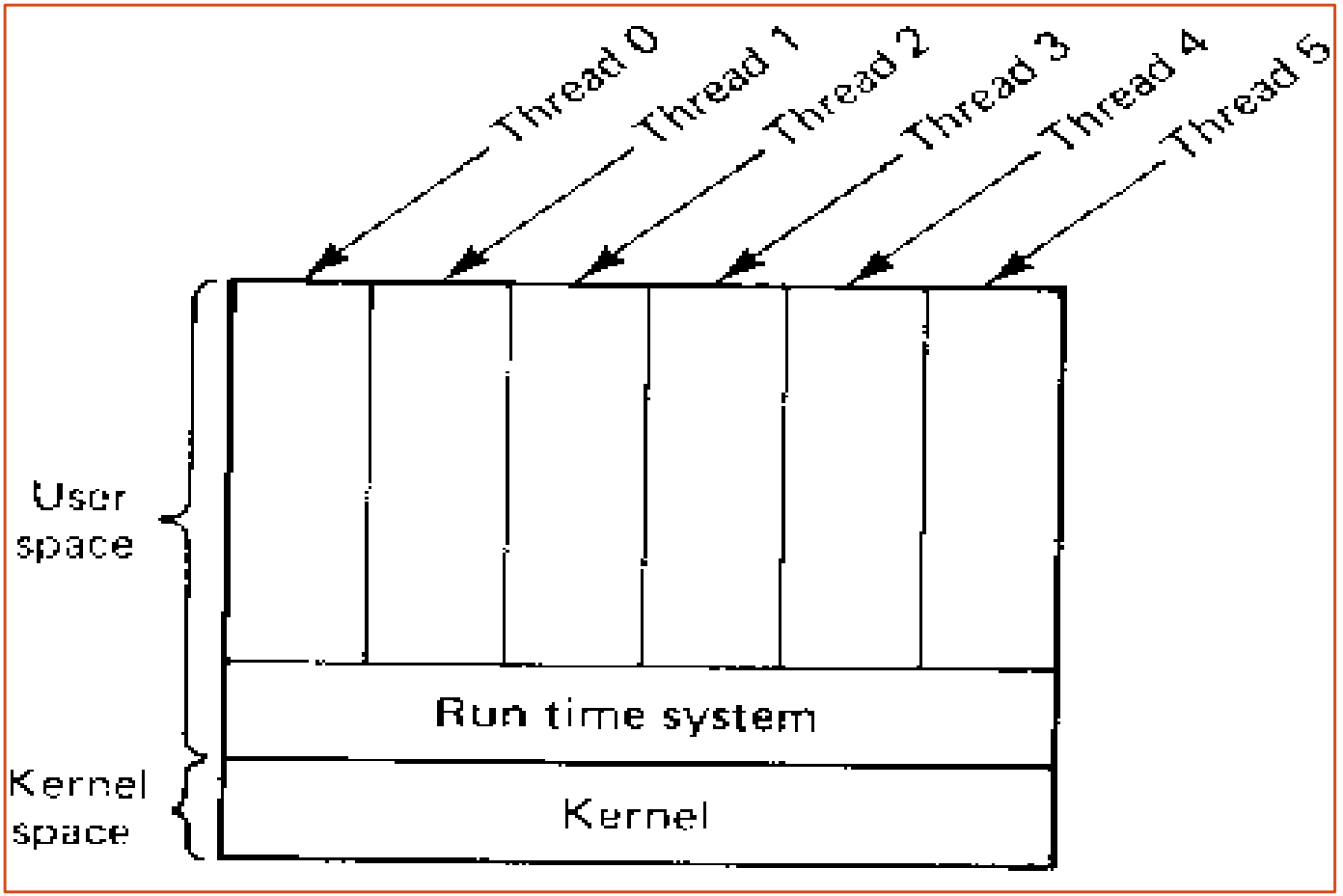
- A set of primitives (e.g. library calls) available to the user relating to threads is called a **thread package**.
- **Static thread**: the choice of how many threads there will be is made when the program is written or when it is compiled. Each thread is allocated a fixed stack. This approach is simple, but inflexible.
- **Dynamic thread**: allow threads to be created and destroyed on-the-fly during execution

Mutex

- If multiple threads want to access the shared buffer, a mutex is used. A mutex can be locked or unlocked.
- Mutexes are like binary semaphores: 0 or 1.
  - Lock: if a mutex is already locked, the thread will be blocked.
  - Unlock: unlocks a mutex. If one or more threads are waiting on the mutex, exactly one of them is released. The rest continue to wait.
  - Trylock: if the mutex is locked, Trylock does not block the thread.

Implementing a threads package

Implementing threads in user space



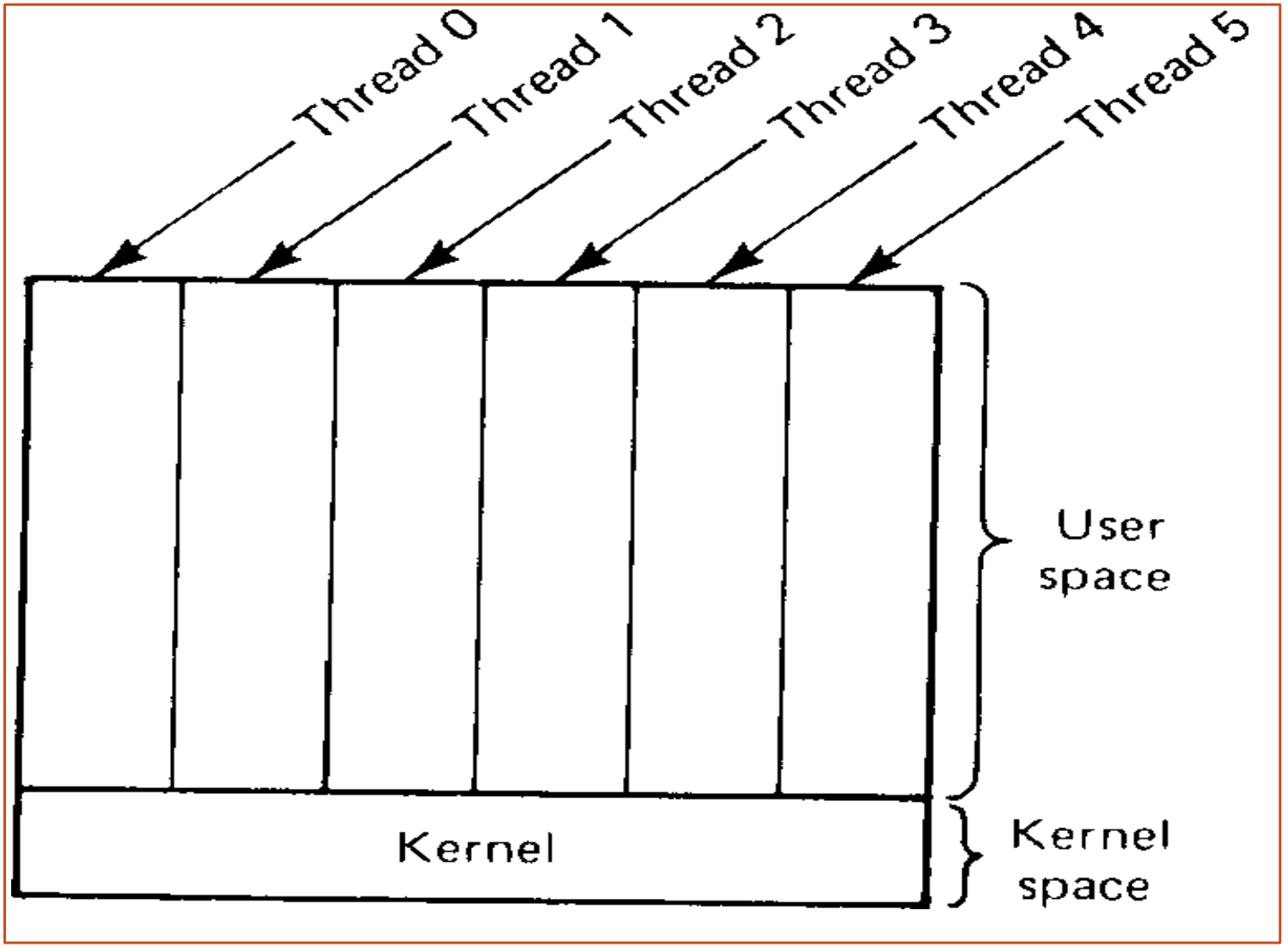
Advantage

- User-level threads package can be implemented on an operating system that does not support threads. For example, the UNIX system.
- The threads run on top of a runtime system, which is a collection of procedures that manage threads. The runtime system does the thread switch. Store the old environment and load the new one. It is much faster than trapping to the kernel.
- User-level threads scale well. Kernel threads require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Disadvantage

- Blocking system calls are difficult to implement. Letting one thread make a system call that will block the thread will stop all the threads.
- Page faults. If a thread causes a page faults, the kernel does not know about the threads. It will block the entire process until the page has been fetched, even though other threads might be runnable.
- If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- For the applications that are essentially CPU bound and rarely block, there is no point of using threads. Because threads are most useful if one thread is blocked, then

Implementing threads in the kernel



- The kernel knows about and manages the threads. No runtime system is needed.
- When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation and destruction.
- To manage all the threads, the kernel has one table per process with one entry per thread.
- When a thread blocks, the kernel can run either another thread from the same process or a thread from a different process.

Scheduler Activations

- Scheduler activations combine the **advantage of user threads (good performance) and kernel threads**.
- The goals of the scheduler activation are to mimic the **functionality of kernel threads**, but with the **better performance and greater flexibility** usually associated with threads packages implemented in **user space**.
- Efficiency is achieved by **avoiding unnecessary transitions** between user and kernel space. If a thread blocks, the user-space runtime system can schedule a new one by itself.

- Disadvantage:
  Upcall from the kernel to the runtime system violates the structure in the layered system.

Unit 4 - Distributed Operating Systems

System model -Work Station Model, Processor Pool & Hybrid Model

System Models

- **The workstation model:**

  the system consists of workstations scattered throughout a building or campus and connected by a high-speed LAN.

  The systems in which workstations have local disks are called **diskful workstations or disky worskstation**. Otherwise, **diskless workstations**.

Fig. 4-10. A network of personal workstations, each with a local file system.

Why diskless workstation?

- If the workstations are diskless, the file system must be implemented by one or more remote file servers. Diskless **workstations are cheaper**.
- **Ease of maintenance**: installing new release of program on several servers than on hundreds of machines. Backup and hardware maintenance is also simpler.
- Diskless **does not have fans and noises**.
- Diskless provides **symmetry and flexibility**. You can use any machine and access your files because all the files are in the server.

Advantage:

- low cost
- easy hardware and software maintenance
- symmetry and flexibility.

Disadvantage:

- heavy networks usage;
- file servers may become bottlenecks.

Diskful workstations

The disks in the diskful workstation are used in one of the four ways:

1. **Paging and temporary files** (temporary files generated by the compiler passes).
   Advantage: reduces network load over diskless case
   Disadvantage: higher cost due to large number of disks needed

2. Paging, temporary files, and system binaries (binary executable programs such as the compilers, text editors, and electronic mail handlers).

Advantage: reduces network load even more

Disadvantage: higher cost; additional complexity of updating the binaries

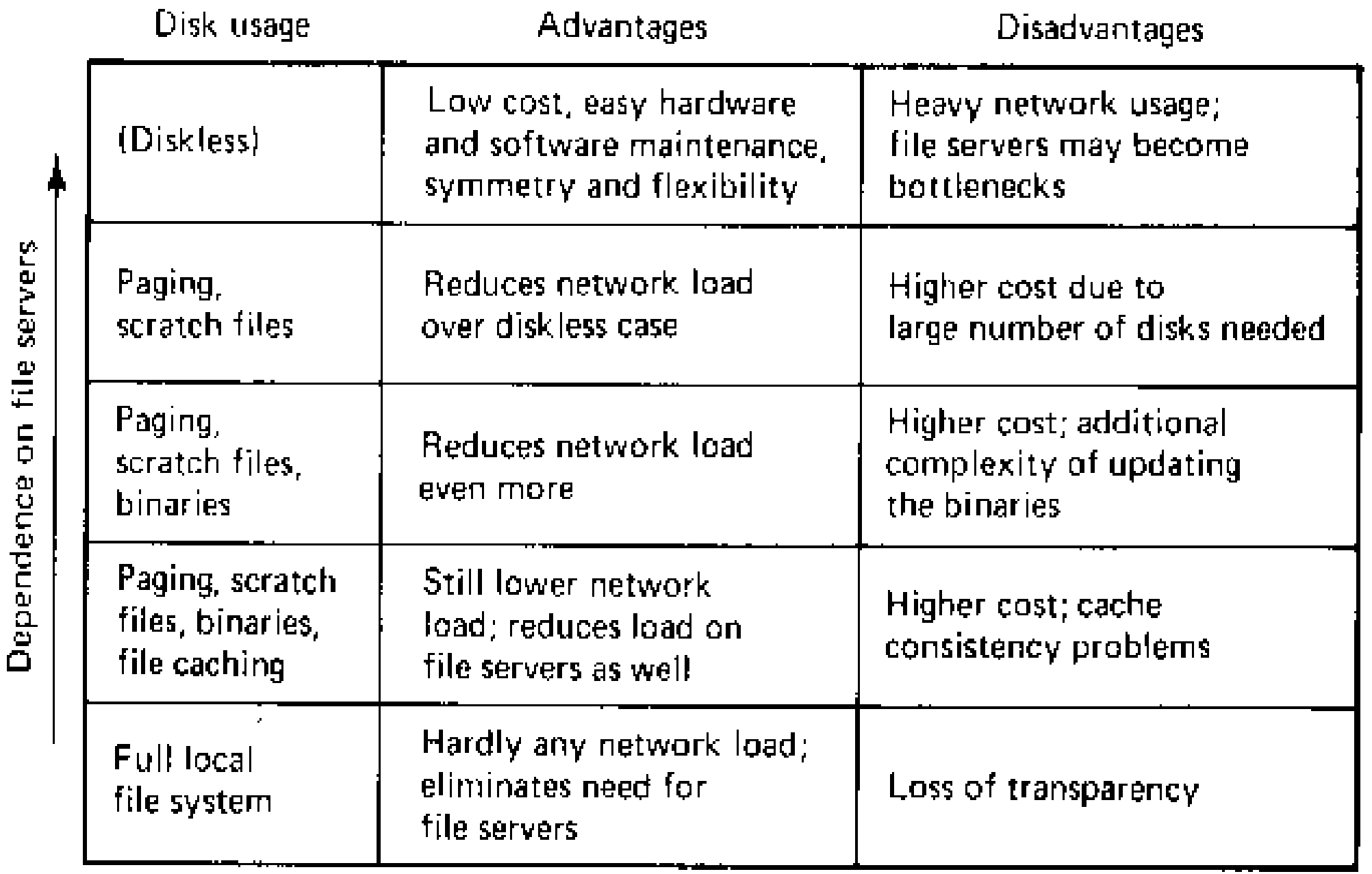
3. **Paging, temporary files, system binaries, and file caching** (download the file from the server and cache it in the local disk. Can make modifications and write back. Problem is cache coherence).

   Advantage: still lower network load; reduces load on file servers as well

   Disadvantage: higher cost; cache consistency problems

4. **Complete local file system** (low network traffic but sharing is difficult).

   Advantage: hardly any network load; eliminates need for file servers Disadvantage: loss of transparency


Disk usage
Advantages
Disadvantages
(Diskless)
Low cost, easy hardware and software maintenance, symmetry and flexibility
Heavy network usage; file servers may become bottlenecks
Paging, scratch files
Reduces network load over diskless case
Higher cost due to large number of disks needed
Paging, scratch files, binaries
Reduces network load even more
Higher cost; additional complexity of updating the binaries
Paging, scratch files, binaries, file caching
Still lower network load; reduces load on file servers as well
Higher cost; cache consistency problems
Full local file system
Hardly any network load; eliminates need for file servers
Loss of transparency
Dependence on file servers

Using Idle workstation

- If no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle.

Using Idle Workstation

- The earliest attempt to allow idle workstations to be utilized is command:

  rsh machine command

- The first argument names a machine and the second names a command to run. rsh run the specified command on specified machine.

Fig. 4-10. A network of personal workstations, each with a local file system.

Flaws

1) User has to tell which machine to use.
2) The program executes in a different environment than the local one.
3) Maybe log in to an idle machine with many processes.

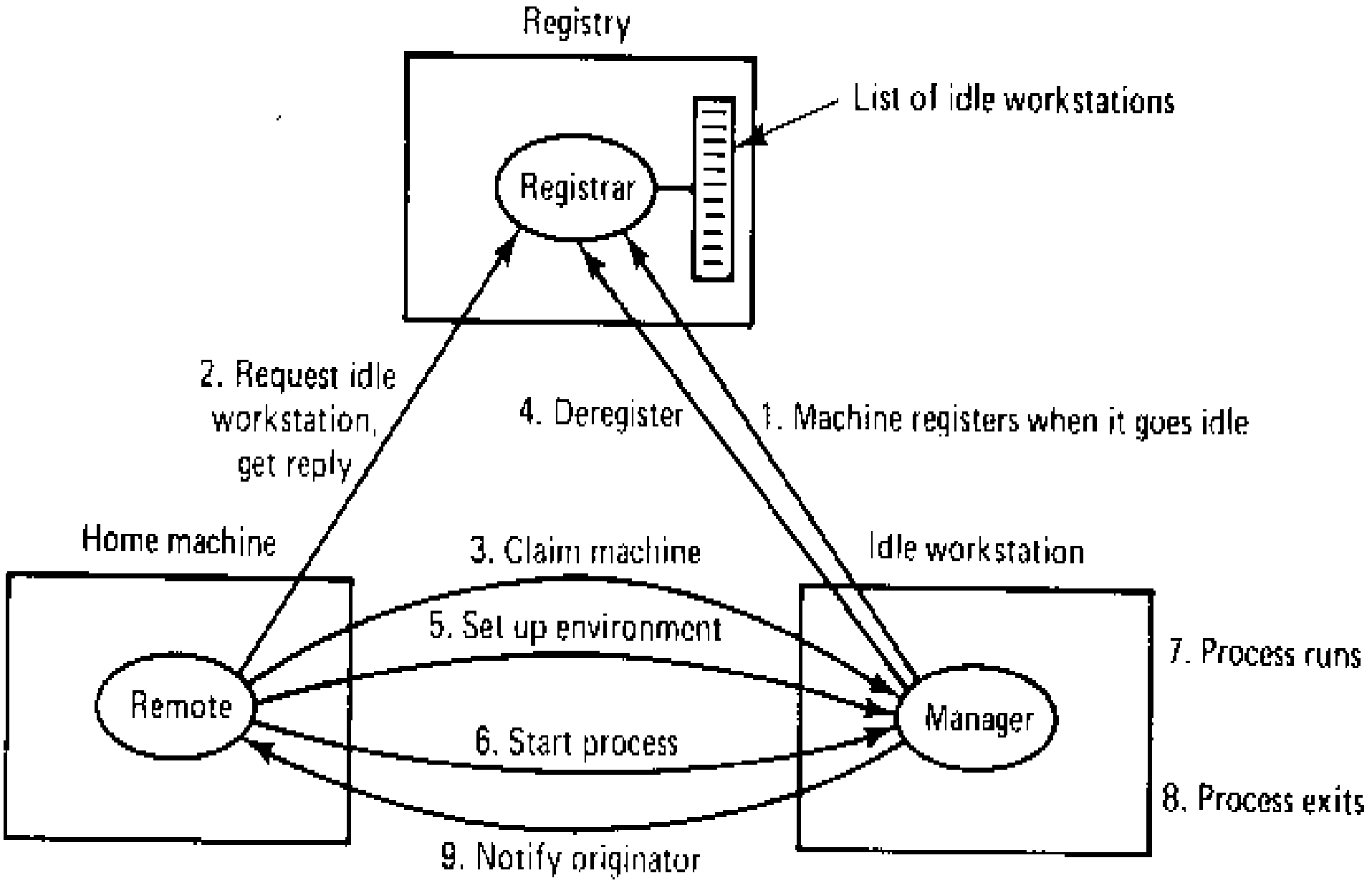
Using Idle workstation

- The algorithms used to locate idle workstations can be divided into two categories:

Server driven--if a server is idle, it registers in registry file or broadcasts to every machine.

Client driven--the client broadcasts a request asking for the specific machine that it needs and wait for the reply.

A registry-based algorithm for finding & using idle workstation

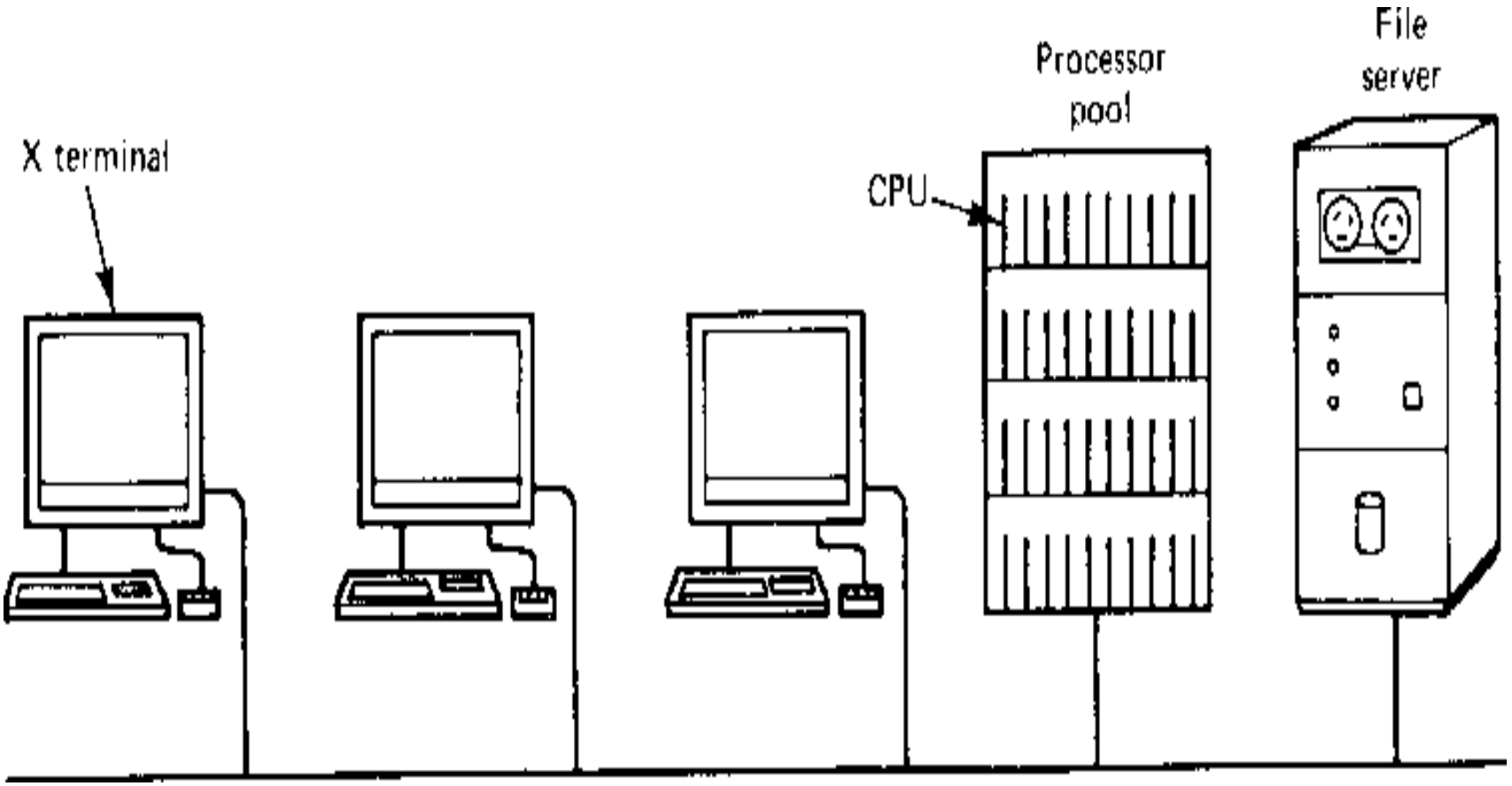


How to run the process remotely?

- To start with, it needs the same view of the file system, the same working directory, and the same environment variables.
- Some system calls can be done remotely but some can not.
  For example, read from keyboard and write to the screen can never be carried out on remote machine. (All system calls that query the state of machine)
- Some must be done remotely, such as the UNIX system calls SBRK (adjust the size of the data segment), NICE (set CPU scheduling priority), and PROFIL (enable profiling of the program

The processor pool model

- A processor pool is **a rack full of CPUs in the machine room**, which can be dynamically allocated to users on demand.

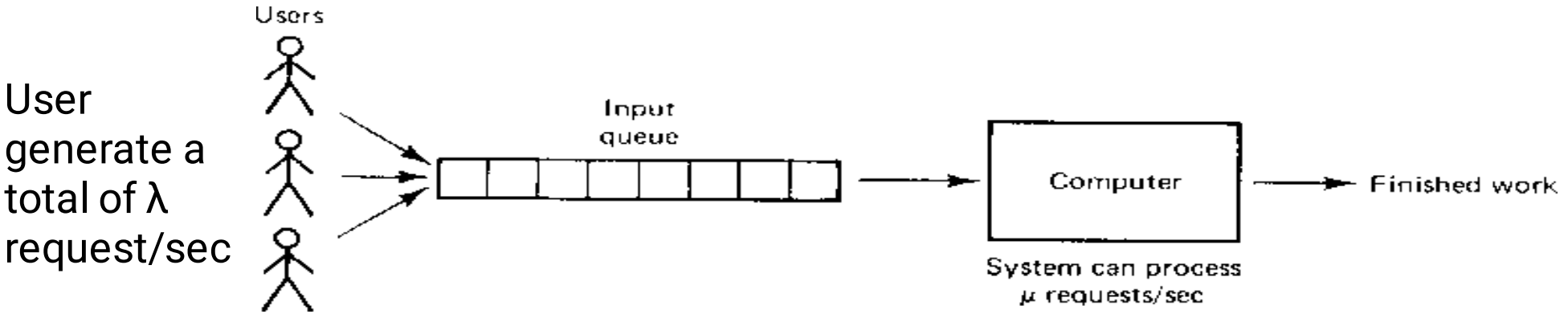


Why processor pool?

- Input rate λ, process rate μ. mean response time **T=1/(μ-λ).**
- If there are n processors, each with input rate λ and process rate μ.
- If we put them together, input rate will be nλ and process rate will be nμ. Mean response time will be

$$T_1 = 1/(n\mu - n\lambda)$$

$$T_1 = T/n$$



A basic queueing system

Example

- Consider a file server that is handling 50 request/sec (μ)
- But get only 40 request/sec(λ)
- The Mean Response time?
- If λ go to 0 (no load)
- What will be response time?

A hybrid model

- A possible compromise is to provide each user with a personal workstation and to have a processor pool in addition.
- For the hybrid model, even if you can not get any processor from the processor pool, at least you have the workstation to do the work.

Processor Allocation

- Determine which process is assigned to which processor. Also called load distribution.
- **Two categories:**

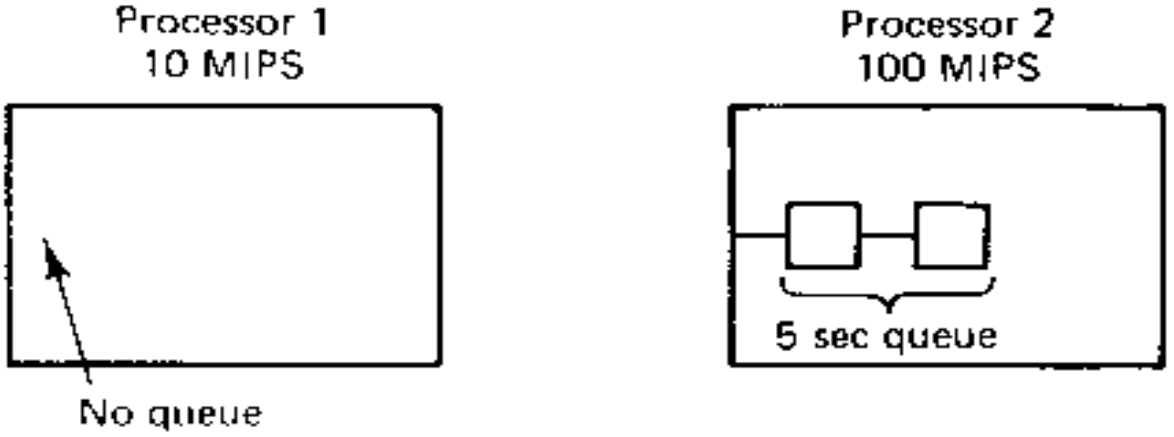
- Static load distribution-**nonmigratory**, once allocated, can not move, no matter how overloaded the machine is.
- Dynamic load distribution-**migratory**, can move even if the execution started. But algorithm is complex.

Allocation Models

1 Maximize CPU utilization

2 Minimize mean response time/ Minimize response ratio

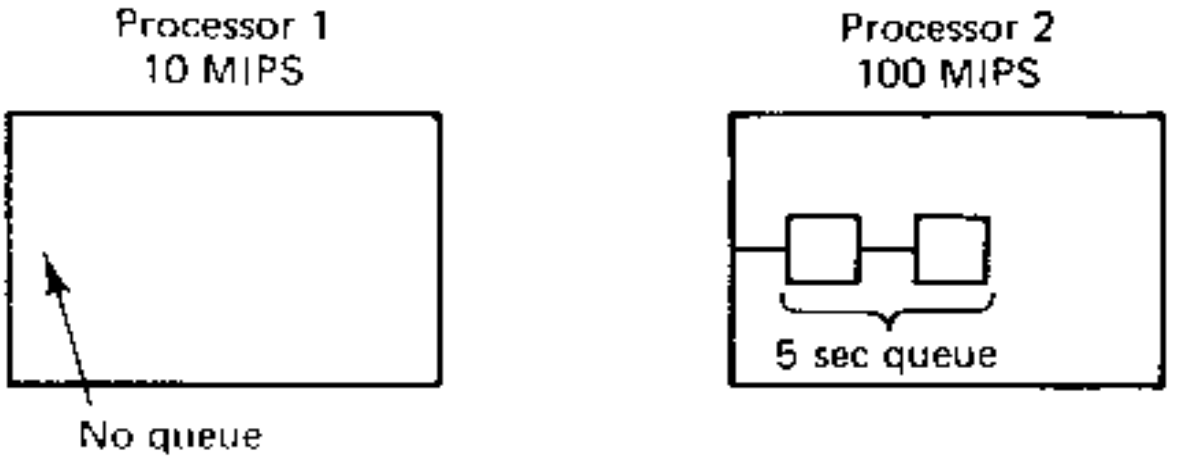
Response ratio-the amount of time it takes to run a process on some machine, divided by how long it would take on some unloaded benchmark processor.



Process		
A (100 million instructions)	10 sec	6 sec
B (300 million instructions)	30 sec	8 sec

Response times of two processes on two processors

Allocation Models



Process		
A (100 million instructions)	10 sec	6 sec
B (300 million instructions)	30 sec	8 sec

Response times of two processes on two processors

Allocation 1: Process A to Processor 1 & Process B to Processor 2

Mean Response time: (10 + 8) / 2 = 9 sec

Allocation 2: Process A to Processor 2 & Process B to Processor 1

Mean Response time: (30 + 6) / 2 = 18 sec

Since Allocation 1 is better assignment for minimizing response time.

Unit 4 - Distributed Operating Systems

Processor Allocation Algorithms

Design issues for processor allocation algorithms

- Deterministic versus heuristic algorithms
- Centralized versus distributed algorithms
- Optimal versus suboptimal algorithms
- Local versus global algorithms
- Sender-initiated versus receiver-initiated algorithms

Design issues for processor allocation algorithms

- **Deterministic algorithms** are appropriate when everything about process behaviour is known in advance.
  - At the other extreme are systems where the load is completely unpredictable.
  - Processor allocation in such systems cannot be done in a deterministic, mathematical way, but of necessity uses ad hoc techniques called heuristics.
- **The second design issue:** is centralized versus distributed.
  - Collecting all the information in one place allows a better decision to be made, but is less robust and can put a heavy load on the central machine. Decentralized algorithms are usually preferable, but some centralized algorithms have been proposed for lack of suitable decentralized alternatives.

Design issues for processor allocation algorithms

- **The third issue:** is related to the first two: Are we trying to find the best allocation, or merely an acceptable one?
  - Involve collecting more information and processing it more thoroughly. In practice, most actual distributed systems settle for heuristic, distributed, suboptimal solutions because it is hard to obtain optimal ones.
- **The fourth issue** relates to what is often called **transfer policy**.
  - When a process is about to be created, a decision has to be made whether or not it can be run on the machine where it is being generated. If that machine is too busy, the new process must be transferred somewhere else.
  - The choice here is whether or not to base the transfer decision entirely on local information. If the machine's load is below some threshold, keep the new process; otherwise, try to get rid of it.

Design issues for processor allocation algorithms

- **The Fifth issue:** Deals with location policy.
  - Once the transfer policy has decided to get rid of a process, the location policy has to figure out where to send it.
  - Location policy cannot be local.
    - Needs information about the load elsewhere to make an intelligent decision.
    - This information can be disseminated in two ways, however. In one method, the senders start the information exchange. In another, it is the receivers that take the initiative.

Design issues for processor allocation algorithms

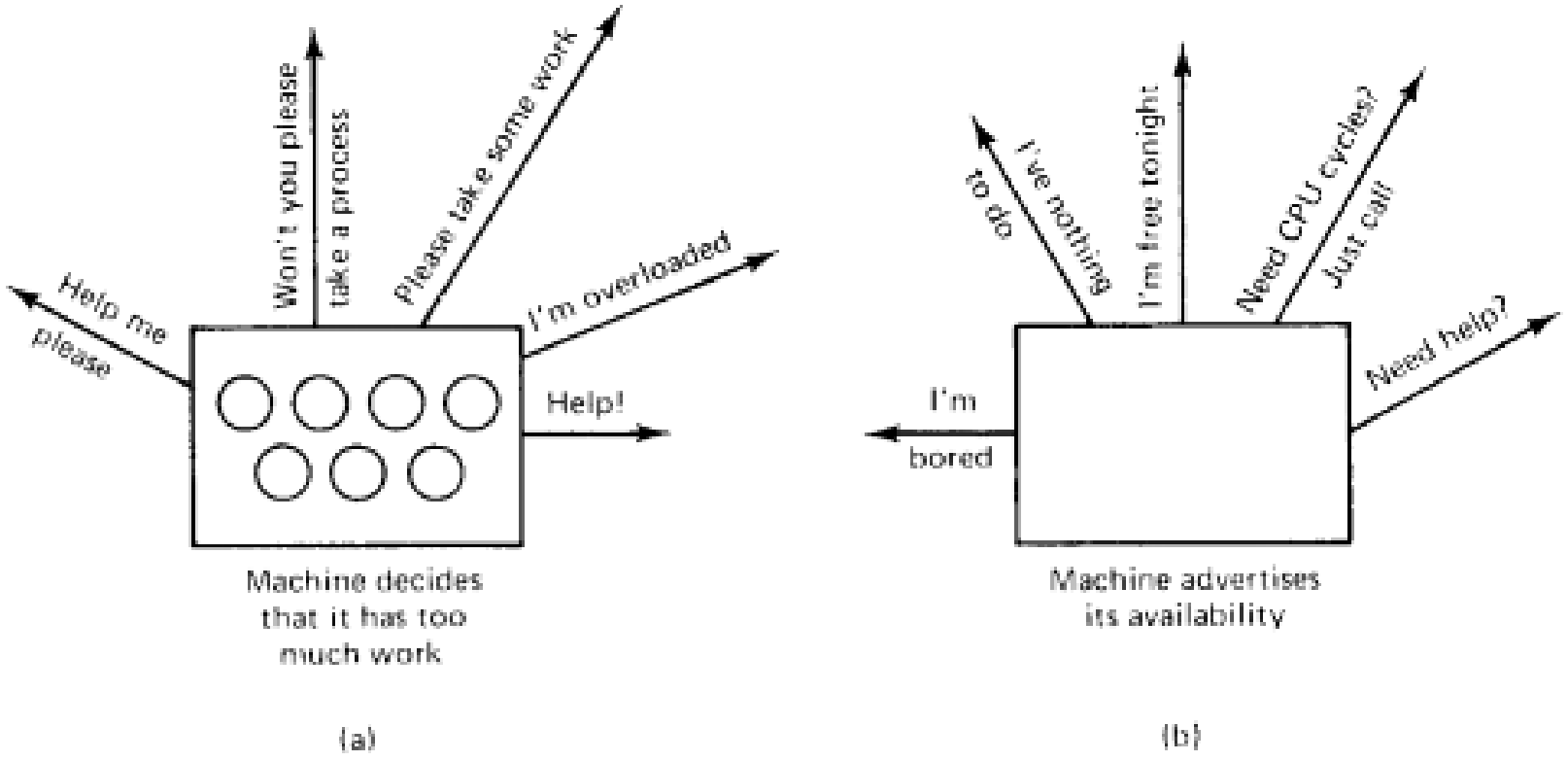


Fig. 4-16. (a) A sender looking for an idle machine. (b) A receiver looking for work to do.

Implementation Issues for Processor Allocation Algorithms

How to measure a processor is overloaded or underloaded?

- Count the processes in the machine? Not accurate because even the machine is idle there are some daemons running.
- Count only the running or ready to run processes? Not accurate because some daemons just wake up and check to see if there is anything to run, after that, they go to sleep. That puts a small load on the system.
- Check the fraction of time the CPU is busy using time interrupts. Not accurate because when CPU is busy it sometimes disable interrupts.

- How to deal with overhead?

  A proper algorithm should take into account the CPU time, memory usage, and network bandwidth consumed by the processor allocation algorithm itself.

- How to calculate complexity?

  If an algorithm performs a little better than others but requires much complex implementation, better use the simple one.

- How to deal with stability?

  Problems can arise if the state of the machine is not stable yet, still in the process of updating.

Example Processor Allocation Algorithms

- A Graph-Theoretic Deterministic Algorithm
- A Centralized Algorithm
- A Hierarchical Algorithm
- A Sender-Initiated Distributed Heuristic Algorithm
- A Receiver-Initiated Distributed Heuristic Algorithm
- A Bidding Algorithm

1) A graph-theoretic deterministic algorithm

- The system can be represented as a weighted graph, with each node being a process and each arc representing the flow of messages between two processes.
- Problem then reduces to finding a way to partition (i.e., cut) the graph into k disjoint subgraphs, subject to certain constraints (e.g., total CPU and memory requirements below some limits for each subgraph).
- For each solution that meets the constraints, arcs that are entirely within a single subgraph represent intramachine communication and can be ignored.
- Arcs that go from one subgraph to another represent network traffic. The goal is then to find the partitioning that minimizes the network traffic while meeting all the constraints.
- Figure 4-17 shows two ways of partitioning the same graph, yielding two different network loads.

A graph-theoretic deterministic algorithm

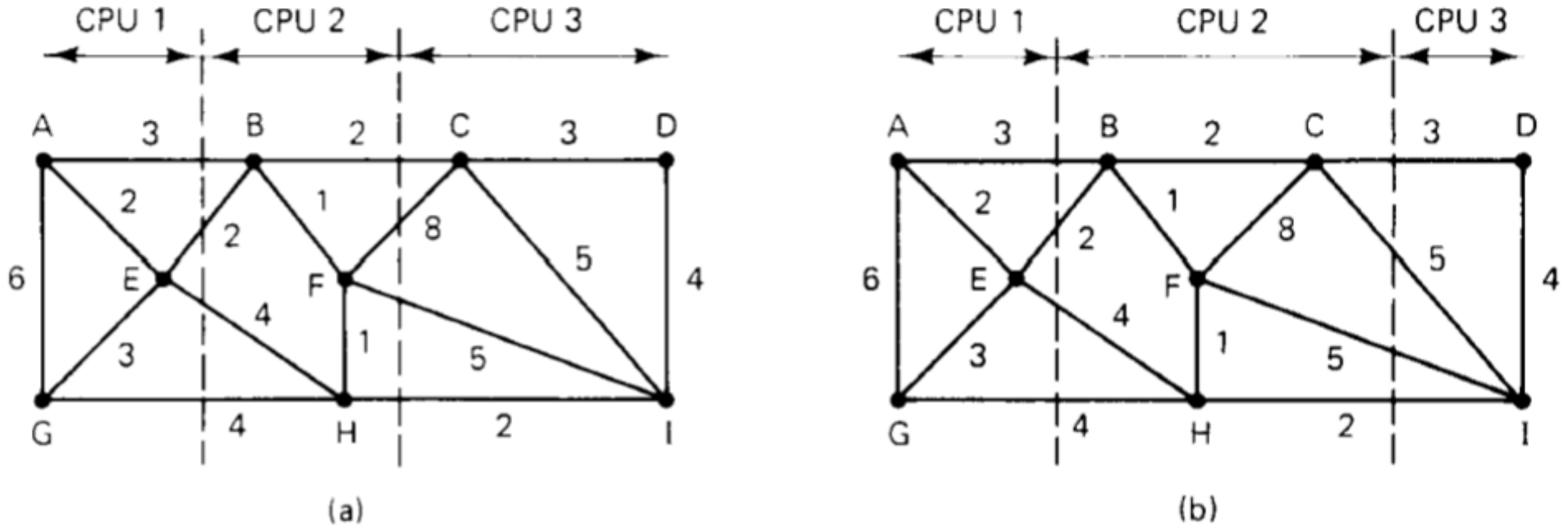
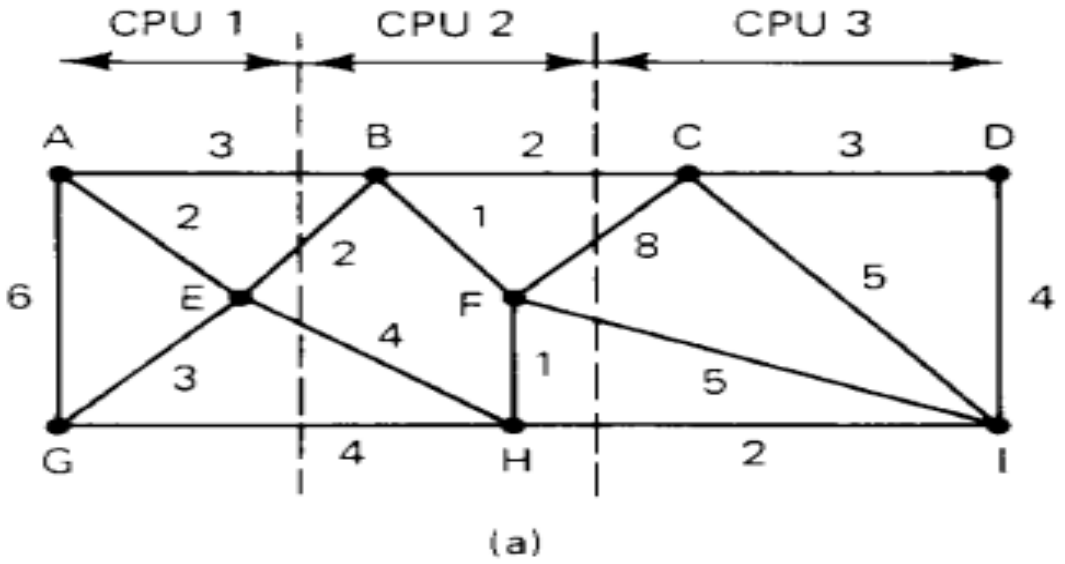


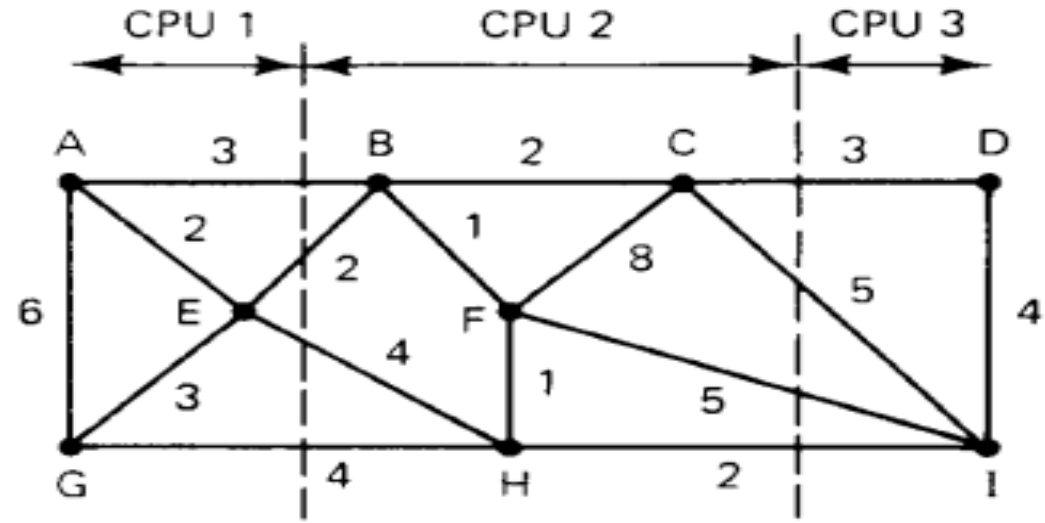
Fig. 4-17. Two ways of allocating nine processes to three processors.

A graph-theoretic deterministic algorithm



(a)

Total network traffic:
3+2+4+4+2+8+5+2=30



(b)

Total network traffic:
3+2+4+4+3+5+5+2=28

A graph-theoretic deterministic algorithm

- In Fig. 4-17(a), we have partitioned the graph with processes A, E, and G on one processor, processes B, F, and H on a second, and processes C, D, and I on the third.
- The total network traffic is the sum of the arcs intersected by the dotted cut lines, or 30 units.
- In Fig. 4-17(b) we have a different partitioning that has only 28 units of network traffic. Assuming that it meets all the memory and CPU constraints, this is a better choice because it requires less communication.

2) A Centralized Algorithm

- Does not require any advance information.
- Is centralized in the sense that a coordinator maintains a usage table with one entry per personal workstation (i.e., per user), initially zero.
- when significant events happen, messages are sent to the coordinator to update the table.
- Allocation decisions are based on the table.
- These decisions are made when scheduling events happen: a processor is being requested, a processor has become free, or the clock has ticked.

A Centralized Algorithm

- When a process is to be created, and the machine it is created on decides that the process should be run elsewhere, it asks the usage table coordinator to allocate it a processor. If there is one available and no one else wants it, the request is granted. If no processors are free, the request is temporarily denied and a note is made of the request.
- When a workstation owner is running processes on other people's machines, it accumulates penalty points, a fixed number per second, as shown in Fig. 4-18.
- When no requests are pending and no processors are being used, the usage table entry is moved a certain number of points closer to zero, until it gets there. In this way, the score goes up and down, hence the name of the algorithm.
- Usage table entries can be positive, zero, or negative. A positive score indicates that the workstation is a net user of system resources, whereas a negative score means that it needs resources. A zero score is neutral.

A Centralized Algorithm

- **The heuristic used for processor allocation**: When a processor becomes free, the pending request whose owner has the lowest score wins.
- As a consequence, a user who is occupying no processors and who has had a request pending for a long time will always beat someone who is using many processors. This property is the intention of the algorithm, to allocate capacity fairly.

Drawbacks

- Do not scale well to large systems.
- The central node soon becomes a bottleneck, not to mention a single point of failure.

A Hierarchical Algorithm

- **One approach** : To organize them in a logical hierarchy independent of the physical structure of the network, as in MICROS
- This approach organizes the machines like people in corporate, military, academic, and other real-world hierarchies. Some of the machines are workers and others are managers.
- For each group of k workers, one manager machine (the " department head") is assigned the task of keeping track of who is busy and who is idle. If the system is large, there will be an unwieldy number of department heads, so some machines will function as "deans," each riding herd on some number of department heads. If there are many deans, they too can be organized hierarchically.
- This hierarchy can be extended with the number of levels needed growing logarithmically with the number of workers. Since each processor need only maintain communication with one superior and a few subordinates, the information stream is

A Hierarchical Algorithm

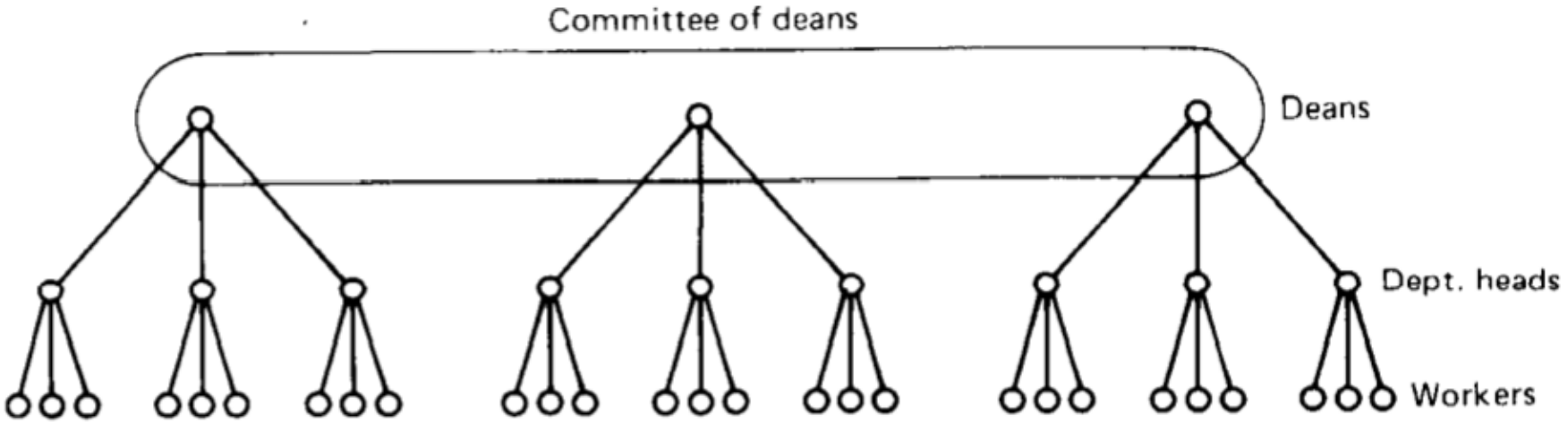


Fig. 4-19. A processor hierarchy can be modeled as an organizational hierarchy.

A Hierarchical Algorithm

- Question is: What happens when a department head stops functioning (crashes)?

Answer:

- To promote one of the direct subordinates of the faulty manager to fill in for the boss. The choice of which can be made by the subordinates themselves, by the deceased's peers, or in a more autocratic system, by the sick manager's boss.
- To avoid having a single (vulnerable) manager at the top of the tree, one can truncate the tree at the top and have a committee as the ultimate authority, as shown in Fig. 4-19. When a member of the ruling committee malfunctions, the remaining members promote someone one level down as replacement.

A Sender-Initiated Distributed Heuristic Algorithm

- The algorithms described above are all centralized or semi centralized. Distributed algorithms also exist.
- when a process is created, the machine on which it originates sends probe messages to a randomly-chosen machine, asking if its load is below some threshold value.
  - If so, the process is sent there.
  - If not, another machine is chosen for probing.
- Probing does not go on forever. If no suitable host is found within N probes, the algorithm terminates and the process runs on the originating machine.

A Sender-Initiated Distributed Heuristic Algorithm

- An analytical queueing model
  - Using this model, it was established that the algorithm behaves well and is stable under a wide range of parameters, including different threshold values, transfer costs, and probe limits.
- Under conditions of heavy load, all machines will constantly send probes to other machines in a futile attempt to find one that is willing to accept more work.
- Few processes will be off-loaded, but considerable overhead may be incurred in the attempt to do so.

A Receiver-Initiated Distributed Heuristic Algorithm

- Whenever a process finishes, the system checks to see if it has enough work.
- If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked.
- If no work is found with N probes, the receiver temporarily stops asking, does any work it has queued up, and tries again when the next process finishes.
- If no work is available, the machine goes idle. After some fixed time interval, it begins probing again.

A Receiver-Initiated Distributed Heuristic Algorithm

- Advantage:
  - Does not put extra load on the system at critical times.
  - When the system is heavily loaded, the chance of a machine having insufficient work is small, but when this does happen, it will be easy to find work to take over.
  - creates considerable probe traffic as all the unemployed machines desperately hunt for work.

A Bidding Algorithm

- Another class of algorithms tries to turn the computer system into a miniature economy, with buyers and sellers of services and prices set by supply and demand (Ferguson et al., 1988).
- The key players in the economy are the processes, which must buy CPU time to get their work done, and processors, which auction their cycles off to the highest bidder.
- Each processor advertises its approximate price by putting it in a publicly readable file.
- This price is not guaranteed, but gives an indication of what the service is worth (actually, it is the price that the last customer paid). Different processors may have different prices, depending on their speed, memory size, presence of floating-point hardware, and other features.

A Bidding Algorithm

- When a process wants to start up a child process, it goes around and checks out who is currently offering the service that it needs.
- It then determines the set of processors whose services it can afford. From this set, it computes the best candidate, where "best" may mean cheapest, fastest, or best price/performance, depending on the application.
- It then generates a bid and sends the bid to its first choice. The bid may be higher or lower than the advertised price.
- Processors collect all the bids sent to them, and make a choice, presumably by picking the highest one. The winners and losers are informed, and the winning process is executed.
- The published price of the server is then updated to

Unit 4 - Distributed Operating Systems

SCHEDULING IN DISTRIBUTED SYSTEMS

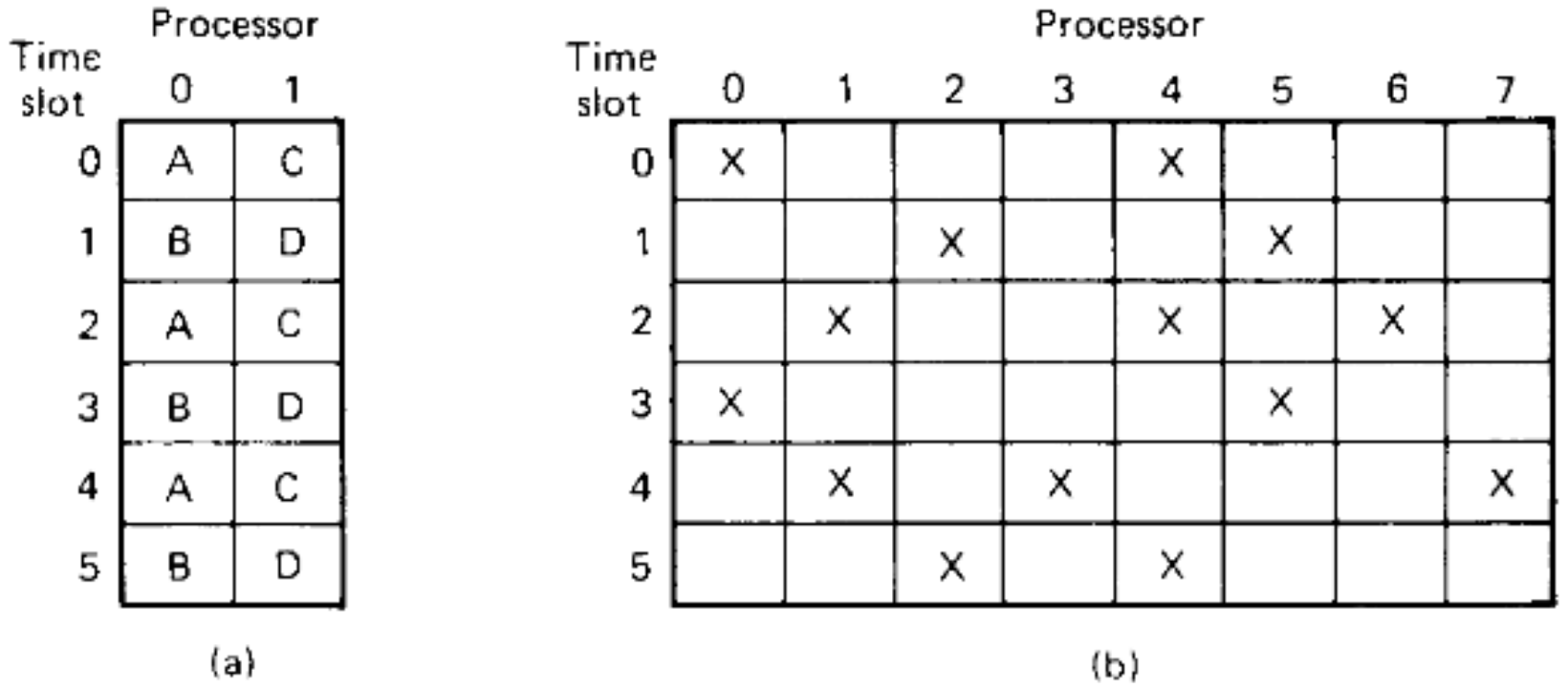
Scheduling in Distributed Systems

Scheduling is essentially **a decision-making process that enables resource sharing among a number of activities by determining their execution order on the set of available resources**.

Scheduling in Distributed Systems

Normally, each processor does its own local scheduling (assuming that it has multiple processes running on it), without regard to what the other processors are doing. Usually, this approach works fine. However, when a group of related, heavily interacting processes are all running on different processors, independent scheduling is not always the most efficient way.

Scheduling in Distributed Systems



Time slot	Processor 0	Processor 1
0	A	C
1	B	D
2	A	C
3	B	D
4	A	C
5	B	D

(a)

Time slot	Processor 0	1	2	3	4	5	6	7
0	X				X			
1			X			X		
2		X			X		X	
3	X					X		
4		X		X				X
5			X		X			

(b)

Fig. 4-20. (a) Two jobs running out of phase with each other. (b) Scheduling matrix for eight processors, each with six time slots. The *X*s indicated allocated slots.

Scheduling in Distributed Systems

Example: Processes A and B run on one processor and processes C and D run on another. Each processor is timeshared in, say, 100-msec time slices, with A and C running in the even slices and B and D running in the odd ones, as shown in Fig. 4-20(a).

Suppose that A sends many messages or makes many remote procedure calls to D.

Scheduling in Distributed Systems

- During time slice 0, A starts up and immediately calls D, which unfortunate is not running because it is now C's turn.
- After 100 msec, process switching takes place, and D gets A's message, carries out the work, and quickly replies. Because B is now running, it will be another 100 msec before A gets the reply and can proceed.
- The net result is one message exchange every 200 msec. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

Co-scheduling

- Takes interprocess communication patterns into account while scheduling to ensure that all members of a group run at the same time.
- First algorithm: uses a **conceptual matrix** in which each column is the process table for one processor, as shown in Fig. 4-20(b).
- Thus, column 4 consists of all the processes that run on processor 4. Row 3 is the collection of all processes that are in slot 3 of some processor, starting with the process in slot 3 of processor 0, then the process in slot 3 of processor 1, and so on.

Co-scheduling

- The gist of his idea is to have each processor use a round-robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, and so on.
- A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized.

Co-scheduling

- By putting all the members of a process group in the same slot number, but on different processors, one has the advantage of N –fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication throughput.
- Thus in Fig. 4-20(b), four processes that must communicate should be put into slot 3, on processors 1, 2, 3, and 4 for optimum performance.
- This scheduling technique can be combined with the hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals.

LOAD BALANCNG AND LOAD SHARING ALGORITHMS

Introduction

- Distributed systems contain a set of resources interconnected by a network
- Processes are migrated to fulfill their resource requirements
- Resource manager are to controlthe assignment of resources to processes
- Resources can be logical (shared file) or physical (CPU)
- We consider a resource to be a processor

Desirable features of a scheduling algorithm

- Stability
  - Unstable when all processes are migrating without accomplishing any useful work
  - It occurs when the nodes turn from lightly-loaded to heavily-loaded state and vice versa
- Scalability
  - A scheduling algorithm should be capable of handling small as well as large networks
- Fault tolerance
  - Should be capable of working after the crash of one or more nodes of the system
- Fairness of Service
  - More users initiating equivalent processes expect to receive the same quality of service

Types of process scheduling techniques

- Task assignment approach
  - User processes are collections of related tasks
  - Tasks are scheduled to improve performance
- Load-balancing approach
  - Tasks are distributed among nodes so as to equalize the workload of nodes of the system
- Load-sharing approach
  - Simply attempts to avoid idle nodes while processes wait for being processed

Task assignment approach

- Main assumptions
  - Processes have been split into tasks
  - Computation requirement of tasks and speed of processors are known
  - Cost of processing tasks on nodes are known
  - Communication cost between every pair of tasks are known
  - Resource requirements and available resources on node are known
  - Reassignment of tasks are not possible

Task assignment approach

- Basic idea: Finding an optimal assignment to achieve goals such as the following:
  - Minimization of IPC costs
  - Quick turnaround time of process
  - High degree of parallelism
  - Efficient utilization of resources

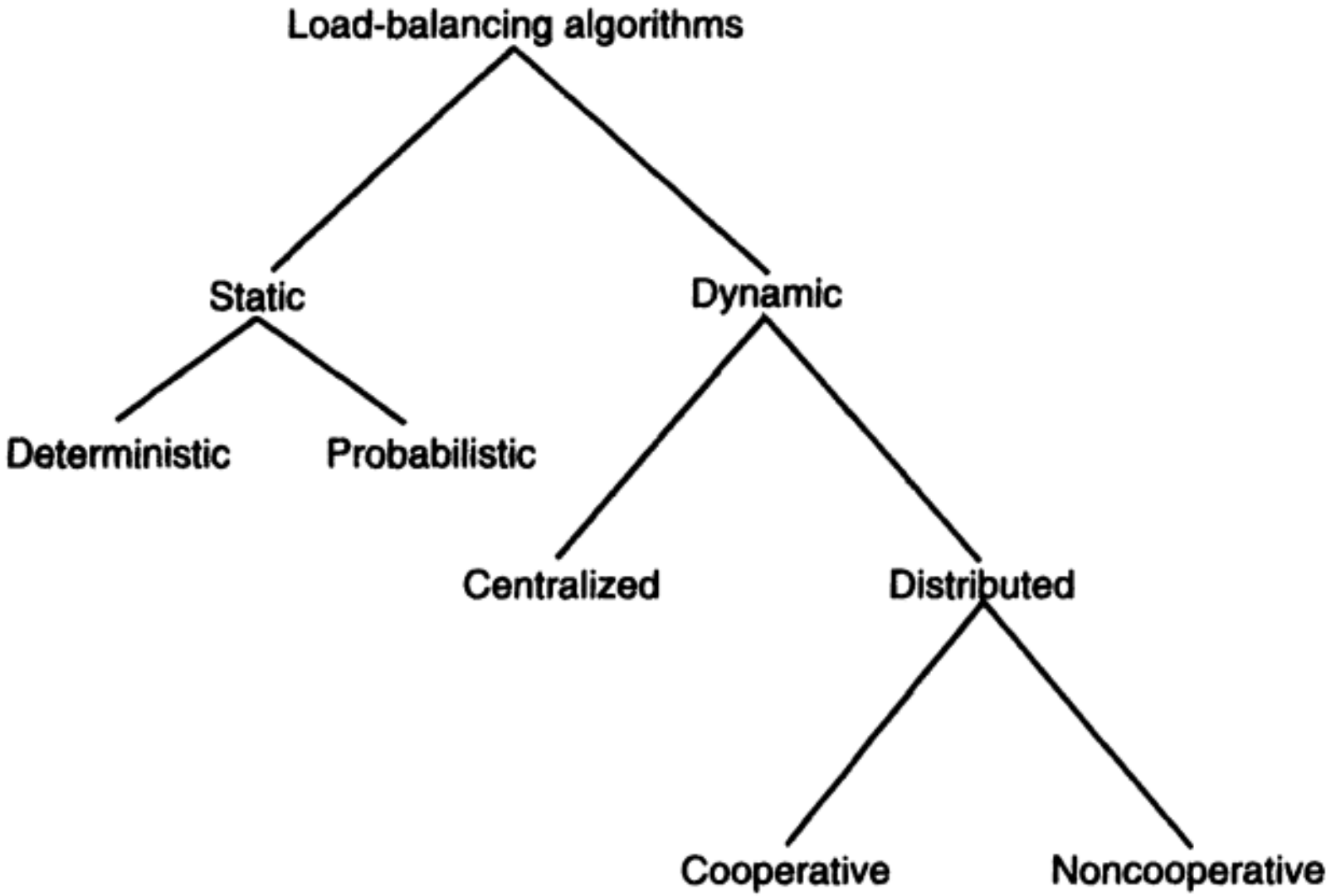


Fig. 7.3 A taxonomy of load-balancing algorithms.

Load-balancing algorithms

Static versus Dynamic

- Static algorithms use only information about the average behavior of the system
- Static algorithms ignore the current state or load of the nodes in the system
- Dynamic algorithms collect state information and react to system state if it changed
- Static algorithms are much more simpler
- Dynamic algorithms are able to give significantly better performance

Load-balancing algorithms

Type of static load-balancing algorithms

Deterministic versus Probabilistic

- Deterministic algorithms use the information about the **properties of the nodes** and the **characteristic of processes** to be scheduled
- Probabilistic algorithms **use information of static attributes of the system** (e.g. number of nodes, processing capability, topology) to formulate simple process placement rules
- Deterministic approach is difficult to optimize
- Probabilistic approach has poor performance

Load-balancing algorithms

Type of dynamic load-balancing algorithms

Centralized versus Distributed

- Centralized approach collects information to server node and makes assignment decision
- Distributed approach contains entities to make decisions on a predefined set of nodes
- Centralized algorithms can make efficient decisions, have lower fault-tolerance
- Distributed algorithms avoid the bottleneck of collecting state information and react faster

Load-balancing algorithms

Type of dynamic load-balancing algorithms

Cooperative versus Noncooperative

- In Noncooperative algorithms entities act as autonomous ones and make scheduling decisions independently from other entities
- In Cooperative algorithms distributed entities cooperatewith each other
- Cooperative algorithms are more complex and involve larger overhead
- Stability of Cooperative algorithms are better

Issues in designing Load-balancing algorithms

- Load estimation policy
  - determines how to estimate the workload of a node
- Process transfer policy
  - determines whether to execute a process locally or remote
- State information exchange policy
  - determines how to exchange load information among nodes
- Location policy
  - determines to which node the transferable process should be sent
- Priority assignment policy
  - determines the priority of execution of local and remote processes
- Migration limiting policy
  - determines the total number of times a process can migrate

Load estimation policy I

for Load-balancing algorithms

- To balance the workload on all the nodes of the system, it is necessary to decide how to measure the workload of a particular node.
  - Some measurable parameters (with time and node dependent factor) can be the following:
    - Total number of processes on the node
    - Resource demands of these processes
    - Instruction mixes of these processes
    - Architecture and speed of the node's processor
- Several load-balancing algorithms use the total number of processes to achieve big efficiency

Load estimation policy II

for Load-balancing algorithms

- In some cases the true load could vary widely depending on the remaining service time, which can be measured in several way:
  - *Memoryless method* assumes that all processes have the same expected remaining service time, independent of the time used so far
  - *Pastrepeats* assumes that the remaining service time is
    equal to the time used so far
  - *Distribution method* states that if the distribution service times is known, the associated process's remaining service time is the expected remaining time conditioned by the time already used

Load estimation policy III.

for Load-balancing algorithms

- None of the previous methods can be used in modern systems because of periodically running processes and daemons
- An acceptable method for use as the load estimation policy in these systems would be to measure the CPU utilization of the nodes
- Central Processing Unit utilization is defined as the number of CPU cycles actually executed per unit of real time
- It can be measured by setting up a timer to periodically check the CPU state (idle/busy)

Process transfer policy I

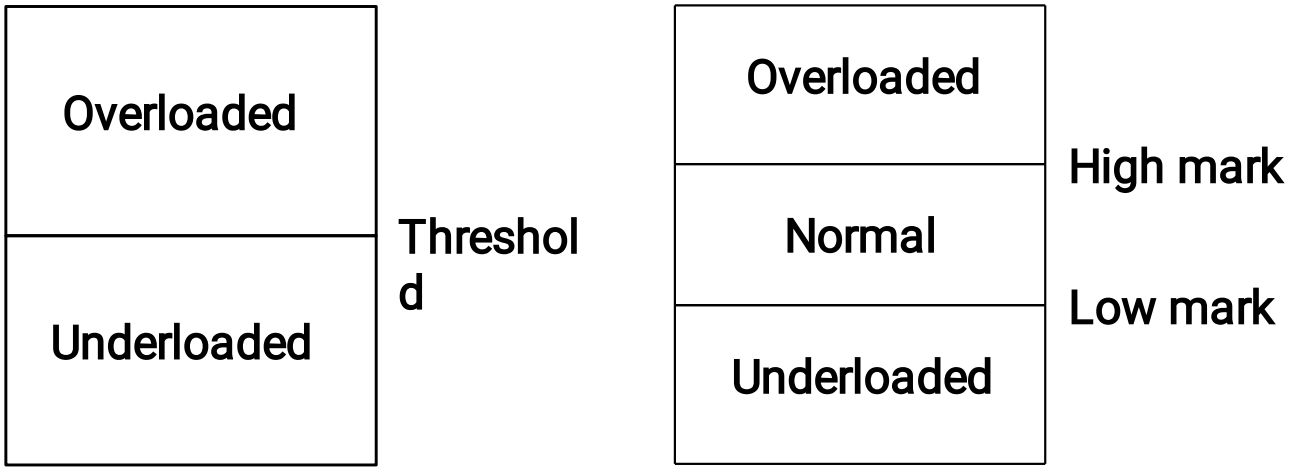
for Load-balancing algorithms

- Most of the algorithms use the *threshold policy* to decide on whether the node is lightly-loaded or heavily-loaded
- Threshold value is a limiting value of the workload of node which can be determined by
  - Static policy: predefined threshold value for each node depending on processing capability
  - Dynamic policy: threshold value is calculated from average workload and a predefined constant
- Below threshold value node accepts processes to execute, above threshold value node tries to transfer processes to a lightly-loaded node

Process transfer policy II

for Load-balancing algorithms

- Single-threshold policy may lead to unstable algorithm because underloaded node could turn to be overloaded right after a process migration
- To reduce instability double-threshold policy has been proposed which is also known as high-low policy



Single-threshold policy

Double-threshold policy

Process transfer policy III

for Load-balancing algorithms

- Double threshold policy
  - When node is in overloaded region new local processes are sent to run remotely, requests to accept remote processes are rejected
  - When node is in normal region new local processes run locally, requests to accept remote processes are rejected
  - When node is in underloaded region new local processes run locally, requests to accept remote processes are accepted

Process transfer policy III.

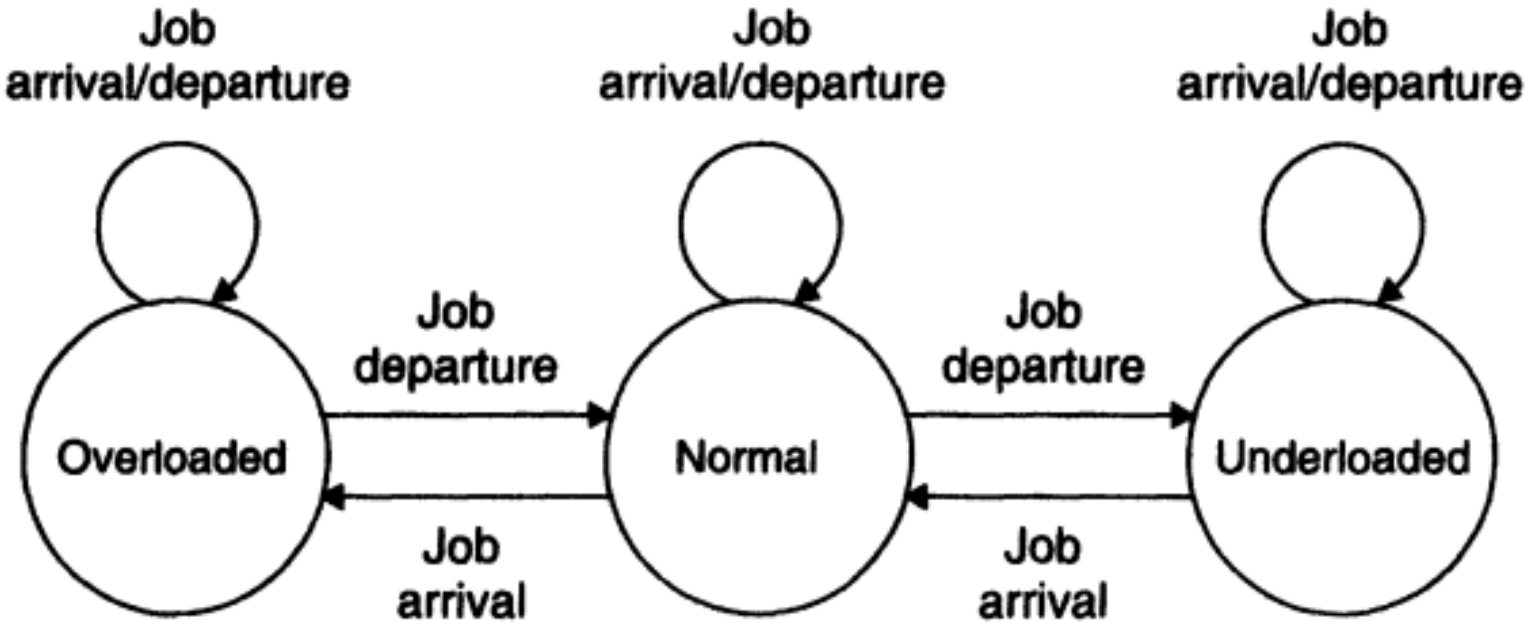


Fig. 7.5 State transition diagram of the load of a node in case of double-threshold policy.

Location policy I

for Load-balancing algorithms

- Threshold method
  - Policy selects a random node, checks whether the node is able to receive the process, then transfers the process. If node rejects, another node is selected randomly. This continues until probe limit is reached.
- Shortest method
  - L_p distinct nodes are chosen at random, each is polled to determine its load. The process is transferred to the node having the minimum value unless its workload value prohibits to accept the process.
  - Simple improvement is to discontinue probing whenever a node with zero load is encountered.

Location policy II

for Load-balancing algorithms

- Bidding method
  - Nodes contain managers (to send processes) and contractors (to receive processes)
  - Managers broadcast a request for bid, contractors respond with bids (prices based on capacity of the contractor node) and manager selects the best offer
  - Winning contractor is notified and asked whether it accepts the process for execution or not
  - Full autonomy for the nodes regarding scheduling
  - Big communication overhead
  - Difficult to decide a good pricing policy

Location policy III.

for Load-balancing algorithms

- Pairing
  - Contrary to the former methods the pairing policy is to reduce the variance of load only between pairs
  - Each node asks some randomlychosennode to form a pair with it
  - If it receives a rejection it randomly selects another node and tries to pair again
  - Two nodes that differ greatly inload are temporarily paired with each other and migration starts
  - The pair is broken as soon as the migration is over
  - A node only tries to find a partner if it hasat least two processes

State information exchange policy I

for Load-balancing algorithms

- Dynamic policies require frequent exchange of state information, but these extra messages arise two opposite impacts:
  - Increasing the number of messages gives more accurate scheduling decision
  - Increasing the number of messages raises the queuing time of messages
- State information policies can be the following:
  - Periodic broadcast
  - Broadcast when state changes
  - On-demand exchange
  - Exchange by polling

State information exchange policy II

for Load balancing algorithms

- Periodic broadcast
  - Each node broadcasts its state information after the elapse of every *T* units of time
  - Problem: heavy traffic, fruitless messages, poor scalability since information exchange is too large for networks having many nodes
- Broadcast when state changes
  - Avoids fruitless messages by broadcasting the state only when a process arrives or departures
  - Further improvement is to broadcast only when state switches to another region (double-threshold policy)

State information exchange policy III

for Load-balancing algorithms

- On-demand exchange
  - In this method a node broadcast a State-Information-Request message when its state switches from normal to either underloaded or overloaded region.
  - On receiving this message other nodes reply with their own state information to the requesting node
  - Further improvement can be that only those nodes reply which are useful to the requesting node
- Exchange by polling
  - To avoid poor scalability (coming from broadcast messages) the partner node is searched by polling the other nodes on by one, until poll limit is reached

Priority assignment policy

for Load-balancing algorithms

- Selfish
  - Local processes are given higher priority than remote processes. Worst response time performance of the three policies.
- Altruistic
  - Remote processes are given higher priority than local processes. Best response time performance of the three policies.
- Intermediate
  - When the number of local processes is greater or equal to the number of remote processes, local processes are given higher priority than remote processes. Otherwise, remote processes are given higher priority than local processes.

Migration limiting policy

for Load-balancing algorithms

- This policy determines the total number of times a process can migrate
  - Uncontrolled
    - A remote process arriving at a node is treated just as a process originating at a node, so a process may be migrated any number of times
  - Controlled
    - Avoids the instability of the uncontrolled policy
    - Use a *migration count* parameter to fix a limit on the number of time a process can migrate
    - Irrevocable migration policy: *migration count* is fixed to 1
    - For long execution processes *migration count* must be greater than 1 to adapt for dynamically changing states

Load-sharing approach

- Drawbacks of Load-balancing approach
  - Load balancing technique with attempting equalizing the workload on all the nodes is not an appropriate object since big overhead is generated by gathering exact state information
  - Load balancing is not achievable since number of processes in a node is always fluctuating and temporal unbalance among the nodes exists every moment
- Basic ideas for Load-sharing approach
  - It is necessary and sufficient to prevent nodes from being idle while some other nodes have more than two processes
  - Load-sharing is much simpler than load-balancing since it only attempts to ensure that no node is idle when heavily node exists
  - Priority assignment policy and migration limiting policy are the same as that
    for the load-balancing algorithms

Load estimation policies

for Load-sharing algorithms

- Since load-sharing algorithms simply attempt to avoid idle nodes, it is sufficient to know whether a node is **busy or idle**
- Thus these algorithms normally employ the simplest load estimation policy of counting the total number of processes
- In modern systems where permanent existence of several processes on an idle node is possible, algorithms measure CPU utilization to estimate the load of a node

Process transfer policies

for Load-sharing algorithms

- Algorithms normally use all-or-nothing strategy
- This strategy uses the threshold value of all the nodes fixed to 1
- Nodes become receiver node when it has no process, and become sender node when it has more than 1 process
- To avoid processing power on nodes having zero process load-sharing algorithms use a threshold value of 2 instead of 1
- When CPU utilization is used as the load estimation policy, the double-threshold policy should be used as the process transfer policy

Location policies

for Load-sharing algorithms

- Location policy decides whether the sender node or the receiver node of the process takes the initiative to search for suitable node in the system, and this policy can be the following:
  - Sender-initiated location policy
    - Sender node decides where to send the process
    - Heavily loaded nodes search for lightly loaded nodes
  - Receiver-initiated location policy
    - Receiver node decides from where to get the process
    - Lightly loaded nodes search for heavily loaded nodes

Location policies II

for Load-sharing algorithms

- Sender-initiated location policy
  - Node becomes overloaded, it either broadcasts or randomly probes the other
    nodes one by one to find a node that is able to receive remote processes
  - When broadcasting, suitable node is known as soon as reply arrives
- Receiver-initiated location policy
  - Nodes becomes underloaded, it either broadcast or randomly probes the other nodes one by one to indicate its willingness to receive remote processes
- Receiver-initiated policy require preemptive process migration facility since scheduling decisions are usually made at process departure epochs

Location policies III.

for Load-sharing algorithms

- Experiences with location policies
  - Both policies gives substantial performance advantages
    over the situation in which no load-sharing is attempted
  - Sender-initiated policy is preferable at light to moderate system loads
  - Receiver-initiated policy is preferable at high system loads
  - Sender-initiated policy provide better performance for the case when process transfer cost significantly more at receiver-initiated than at sender-initiated policy due to the preemptive transfer of processes

State information exchange policies

for Load-sharing algorithms

- In load-sharing algorithms it is not necessary for the nodes to periodically exchange state information, but needs to know the state of other nodes when it is either underloaded or overloaded
- Broadcast when state changes
  - In sender-initiated/receiver-initiated location policy a node broadcasts State Information Request when it becomes overloaded/underloaded
  - It is called broadcast-when-idle policy when receiver-initiated policy is used
    with fixed threshold value value of 1
- Poll when state changes
  - In large networks polling mechanism is used
  - Polling mechanism randomly asks different nodes for state information until find an appropriate one or probe limit is reached
  - It is called poll-when-idle policy when receiver-initiated policy is used with fixed threshold value value of 1

FAULT TOLERANCE

- A system is said to fail when it does not meet its specification.
- Eg: In some cases, such as a supermarket's distributed ordering system, a failure may result in some store running out of canned beans.
- As computers and distributed systems become widely used in safety-critical missions, the need to prevent failures becomes correspondingly greater.

Component Faults

- Computer systems can fail due to a fault in some component, such as a processor, memory, I/O device, cable, or software.
- A fault is a malfunction, possibly caused by a design error, a manufacturing error, a programming error, physical damage, deterioration in the course of time, harsh environmental conditions (it snowed on the computer), unexpected inputs, operator error, rodents eating part of it, and many other causes.

Fault Classification

- Faults are generally classified as
  - Transient
  - Intermittent
  - Permanent.
- Transient faults occur once and then disappear. if the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

Fault Classification

- An intermittent fault occurs, then vanishes of its own accord, then reappears, and so on.
  - A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, whenever the fault doctor shows up, the system works perfectly.
- A permanent fault is one that continues to exist until the faulty component is repaired.
  - Burnt-out chips, software bugs, and disk head crashes often cause permanent faults.

Fault Classification

- Faults and failures can occur at all levels: transistors, chips, boards, processors, operating systems, user programs, and so on.
- if some component has a probability p of malfunctioning in a given second of time, the probability of it not failing for k consecutive seconds and then failing is p(1–p)k. The expected time to failure is then given by the formula

$$\text{mean time to failure} = \sum_{k=1}^{\infty} kp(1-p)^{k-1}$$

Fault Classification

- Using the well-known equation for an infinite sum starting at k-1: k=/(1–), substituting =1–p, differentiating both sides of the resulting equation with respect to p, and multiplying through by –p we see that

$$\text{mean time to failure} = 1/p$$

- For exar ash is 10-6 per second, the mean time to failure is 106 sec or about 11.6 days.

System Failures - Processor faults

- System reliability is especially important in a distributed system due to the large number of components present, hence the greater chance of one of them being faulty.
- Two types of processor faults can be distinguished:
  - 1. Fail-silent faults.
  - 2. Byzantine faults.

Processor faults

- With fail-silent faults, a faulty processor just stops and does not respond to subsequent input or produce further output, except perhaps to announce that it is no longer functioning. These are also called fail-stop faults.
- With Byzantine faults, a faulty processor continues to run, issuing wrong answers to questions, and possibly working together maliciously with other faulty processors to give the impression that they are all working correctly when they are not.

Synchronous versus Asynchronous System

- Deals with performance in a certain abstract sense.

Eg: Suppose that we have a system in which if one processor sends a message to another, it is guaranteed to get a reply within a time T known in advance. Failure to get a reply means that the receiving system has crashed. The time T includes sufficient time to deal with lost messages (by sending them up to n times).

Synchronous versus Asynchronous System

- SYNCHRONOUS :
  - In the context of research on fault tolerance, a system that has the property of always responding to a message within a known finite bound if it is working is said to be synchronous.
- ASYNCHRONOUS:
  - A system not having this property is said to be asynchronous.
- Asynchronous systems are going to be harder to deal with than synchronous ones.

Use of Redundancy

- The general approach to fault tolerance is to use redundancy.
- Three kinds :
  - Information redundancy,
  - Time redundancy
  - Physical redundancy.
- With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

Use of Redundancy

- With time redundancy, an action is performed, and then, if need be, it is performed again. If a transaction aborts, it can be redone with no harm.
  - Time redundancy is especially helpful when the faults are transient or intermittent.
- With physical redundancy, extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components.
  - For example, extra processors can be added to the system so that if a few of them crash, the system can still function correctly.

Use of Redundancy

- Two ways to organize these extra processors: **Active replication and primary backup.**
- Consider the case of a server.
  - When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely.
  - In contrast, the primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.
- For both of them, the issues are: 1. The degree of replication required. 2. The average and worst-case performance in the absence of faults. 3. The average and worst-case performance when a fault occurs.

Fault Tolerance Using Active Replication

- Active replication is a well-known technique for providing fault tolerance using physical redundancy.
- It has also been used for fault tolerance in electronic circuits for years. Consider, for example, the circuit of Fig. 4-21(a).

Triple Modular Redundancy

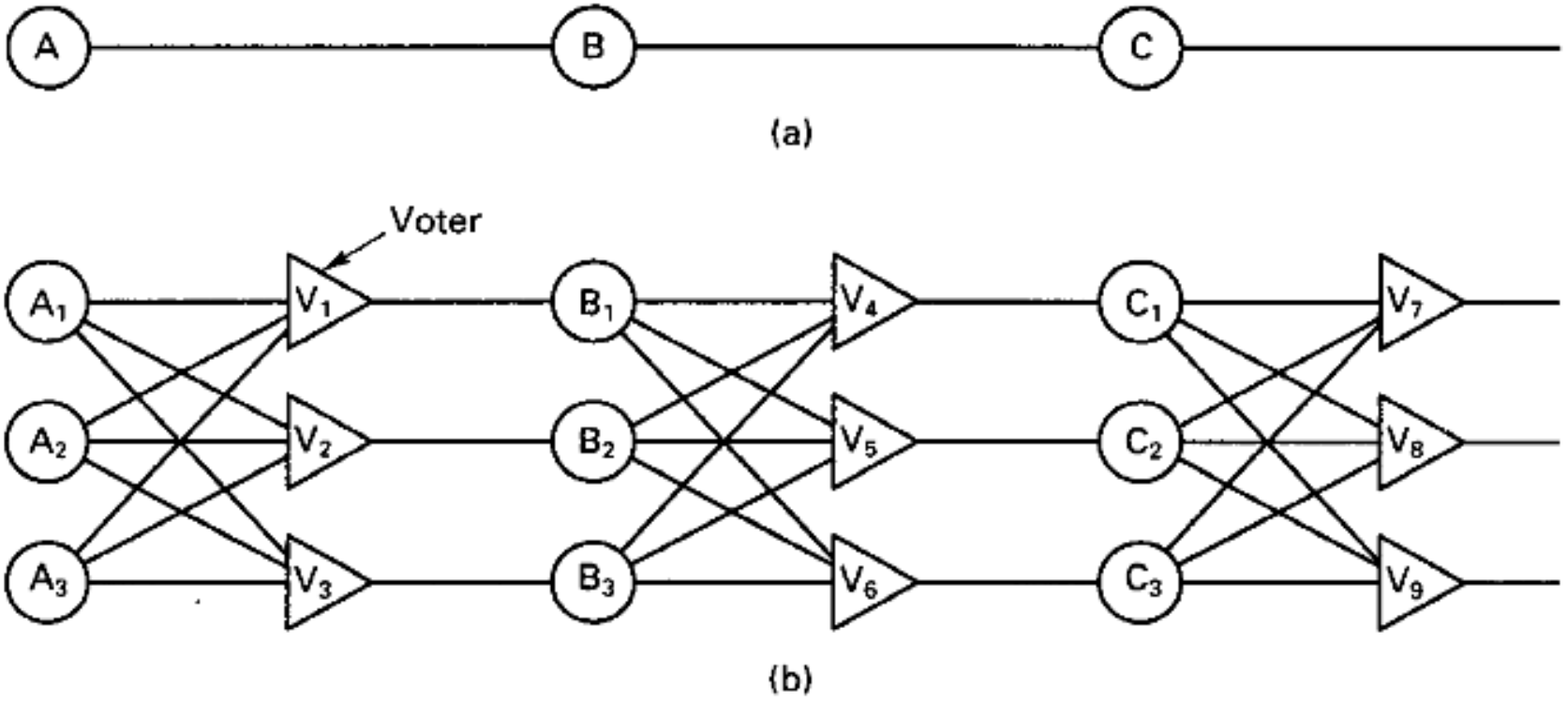


Fig. 4-21. Triple modular redundancy.

Triple Modular Redundancy

- Here signals pass through devices A, B, and C, in sequence. If one of them is faulty, the final result will probably be wrong.
- In Fig. 4-21(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input.
- If all three inputs are different, the output is undefined. This kind of design is known as TMR (Triple Modular Redundancy).

Fault Tolerance Using Primary Backup

- Idea : At any one instant, one server is the primary and does all the work. If the primary fails, the backup takes over.
- Cutover should take place in a clean way and be noticed only by the client operating system, not by the application programs.
- Examples: Government (the Vice President), aviation (co-pilots), automobiles (spare tires), and diesel-powered electrical generators in hospital operating rooms.

Advantage

- Primary-backup fault tolerance has two major advantages over active replication.
- First, it is simpler during normal operation since messages go to just one server (the primary) and not to a whole group. The problems associated with ordering these messages also disappear.
- Second, in practice it requires fewer machines, because at any instant one primary and one backup is needed (although when a backup is put into service as a primary, a new backup is needed instantly).

Example

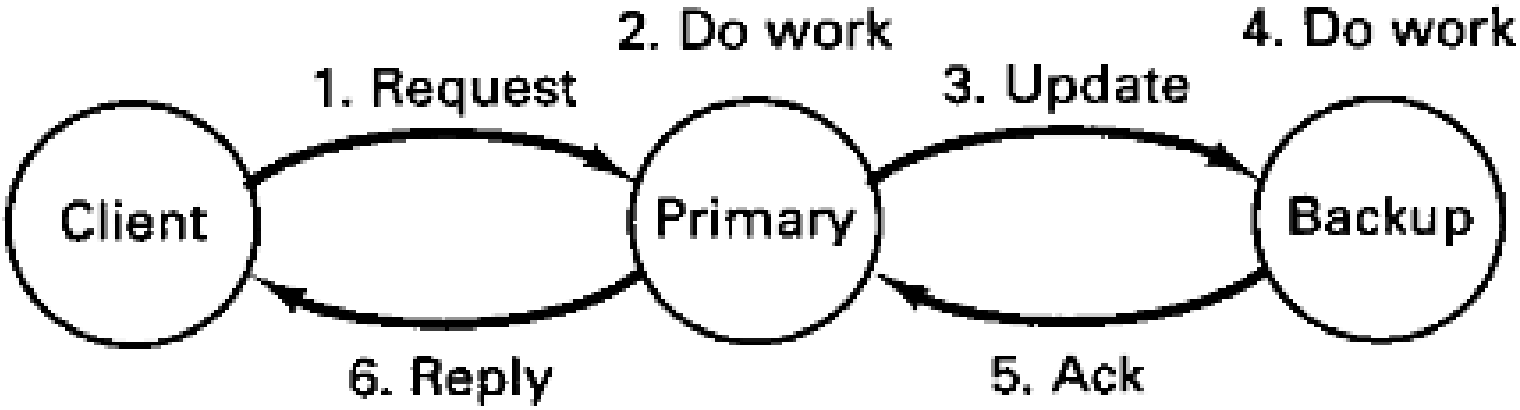


Fig. 4-22. A simple primary-backup protocol on a write operation.

A write operation is depicted. The client sends a message to the primary, which does the work and then sends an update message to the backup. When the backup gets the message, it does the work and then sends an acknowledgement back to the primary. When the acknowledgement arrives, the primary sends the reply to the client.

REAL-TIME DISTRIBUTED SYSTEMS

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

What is a Real-Time System?

- A real-time system interacts with the external world in a way that involves time. [1]
- Such a system is required to complete its work and deliver its services on a timely basis. [2]
- In most cases, a late answer, even if correct, is considered a system failure. [1]

What is a Real-Time System?

Examples:

- Air traffic control system
- Automobile brake system
- Aircraft autopilot system
- Laser eye surgery system
- Tsunami early detection system

What is a Real-Time System?

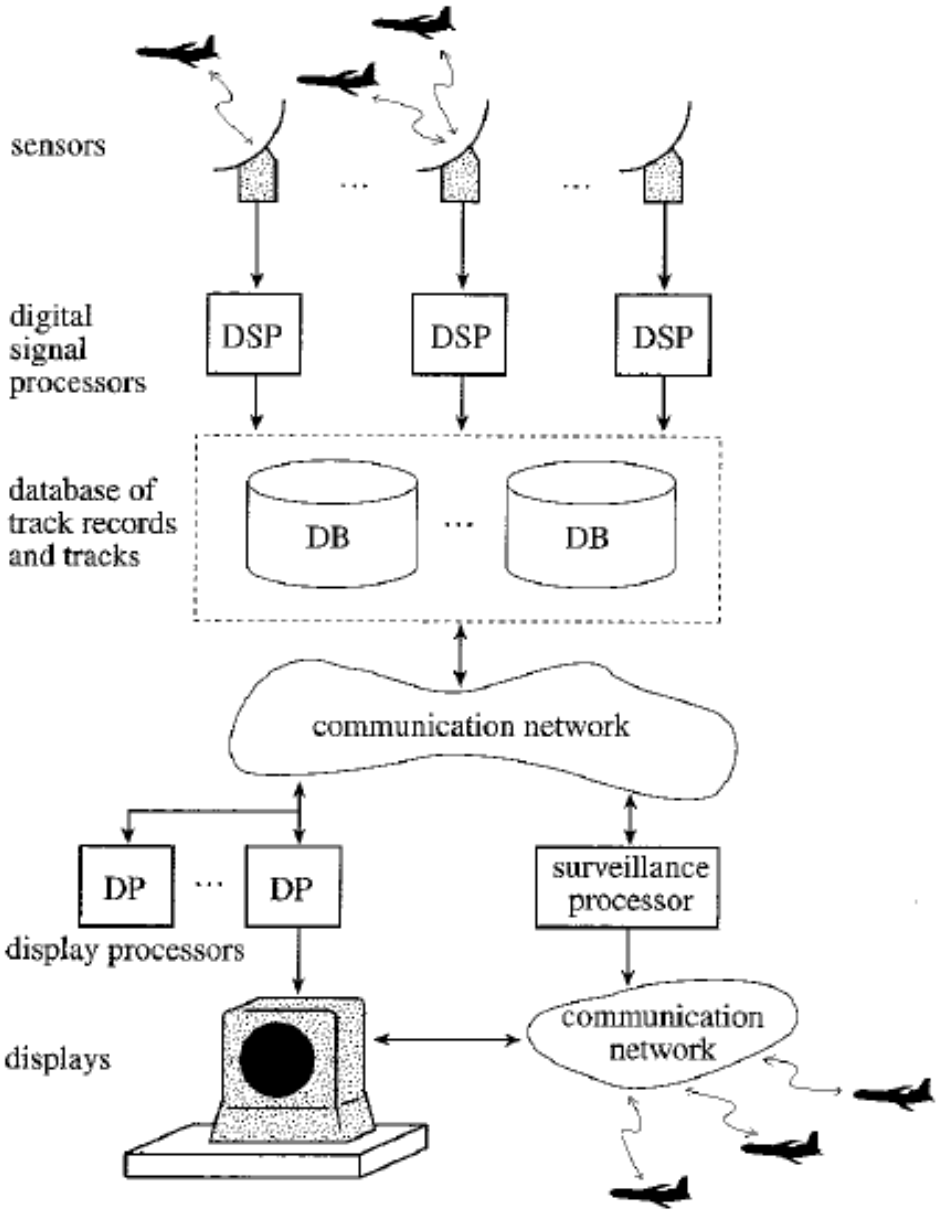


FIGURE 1–5 An architecture of air traffic control system.

[2]

What is a Real-Time System?

Hard vs. Soft real-time systems

- A missed deadline in a hard real-time system is unacceptable. [1]
- An occasional missed deadline in a soft real-time system is okay. [1]
- Hard real-time systems require validation to prove that the system meets timing constraints. [2]

What is a Real-Time System?

Centralized vs. Distributed RTS:

- A single CPU must be able to handle multiple event streams. [1]
- Events may occur periodically, aperiodically, or sporadically. [1]
- As real-time systems become more complex, more processors can be added to distribute the workload. [1]
- However, distributed CPUs introduce problems with task assignment and interprocessor synchronization. [2]

What is a Real-Time System?

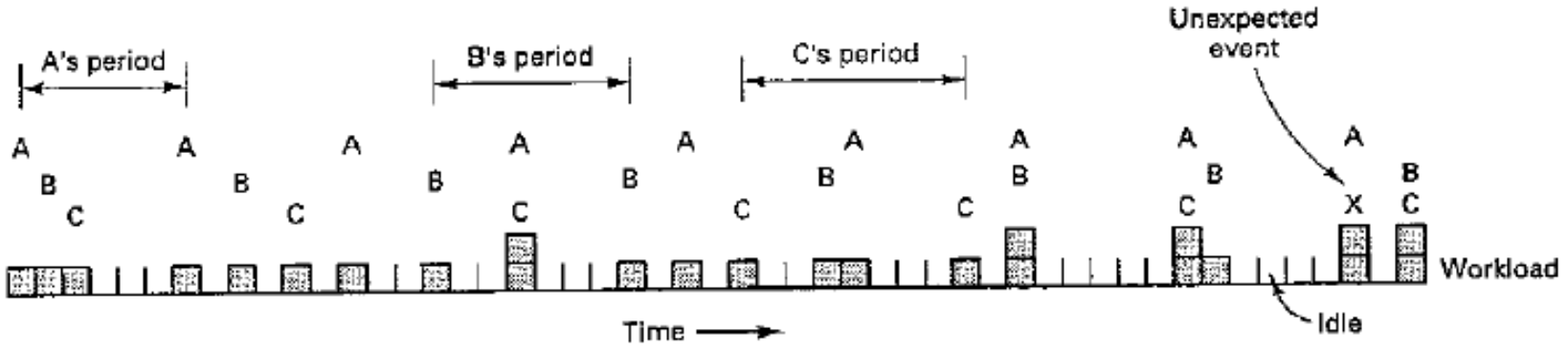


Fig. 4-25. Superposition of three event streams plus one sporadic event.

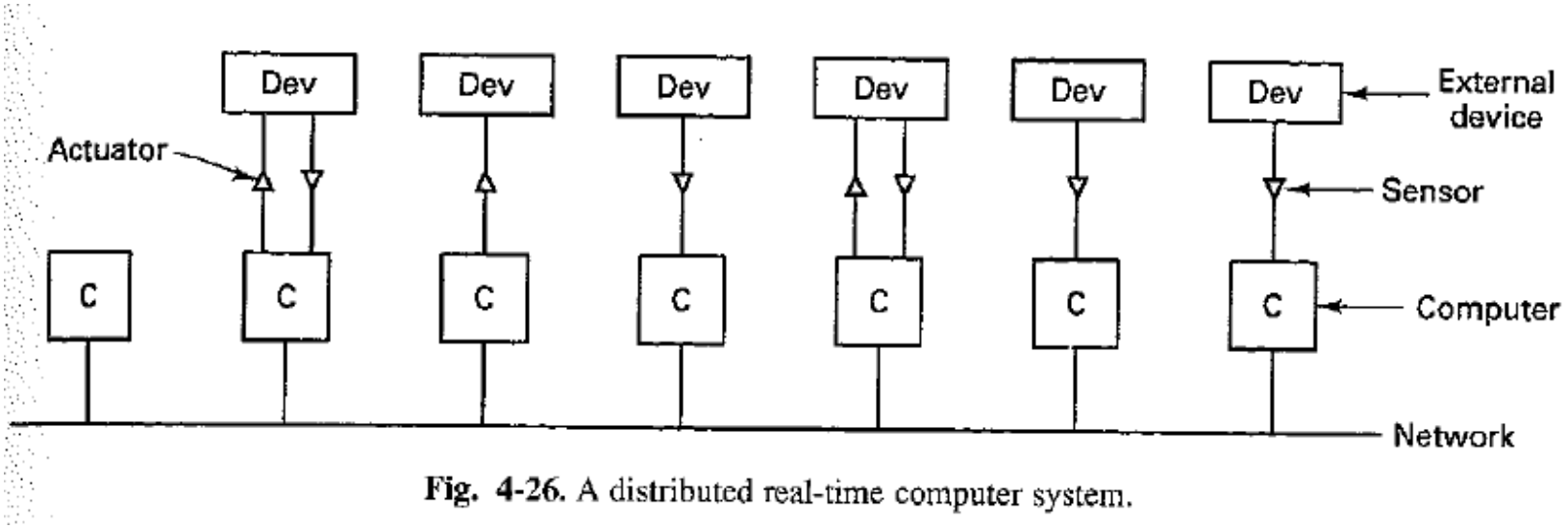


Fig. 4-26. A distributed real-time computer system.

[1]

What is a Real-Time System?

Myths [1]:

1. Real-time systems are about writing device drivers in assembly code.
2. Real-time computing is fast computing.
3. Fast computers will make real-time systems obsolete.

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

Design Issues

Event-triggered vs. Time-triggered systems

- Event-triggered systems are interrupt driven. Widely used design. [1]
- However, event shower may cause massive interrupts and system failure. [1]
- Time-triggered systems use periodic polling during clock ticks to process events. [1]
- However, some latency may be added in processing each event. [1]

Design Issues

Predictability

- Worst-case completion time of every job is known at design time. [2]
- Easy to validate in a priority-driven, static, uniprocessor system, when jobs are independent and preemptable. [2]
- Distributed systems are generally more dynamic and less predictable, but completion times can be bounded if jobs are preemptable and migratable. [2]

Design Issues

Fault Tolerance

- Many real-time systems control safety-critical devices. [1]
- Active replication (e.g. triple replication and voting) can be used if overhead for synchronization is not too great. [1]
- Primary backups may cause missed deadlines during cutover. [1]
- A fail-safe system can halt operation without danger. [1]

Design Issues

Language Support

- Allow programmer to define periodic tasks or threads. [1]
- Allow programmer to define priority of tasks. [1]
- Supply access to one or more high-resolution timers [2].
- No unbounded operations, like *while* loops or garbage collection.
- Examples: Ada 95, Real-time Java, and Real-time C/POSIX

Design Issues

Commercial RTOSes

- Provide a good collection of features and tools. [3]
- Advantages [2]:
  - Standardized
  - Modular and Scalable
  - Fast and Efficient
  - Support priority scheduling, priority inversion control, high-resolution timers, interrupt handling, memory management, and networking.
- Examples: VRTX, VxWorks, pSOS, LynxOS, QNX, Neutrino. [2, 3]

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

Real-Time Communication

- Must be predictable and deterministic. [1]
- Ethernet is unbounded. [1]
- Token ring can guarantee worst-case delay of *kn* byte times (i.e. *k* machines with *n*-byte packet). [1]
- Time Division Multiple Access (TDMA) provides fixed-size frames with *n* slots. Each processor can be assigned one or more dedicated slots. [1]

Real-Time Communication

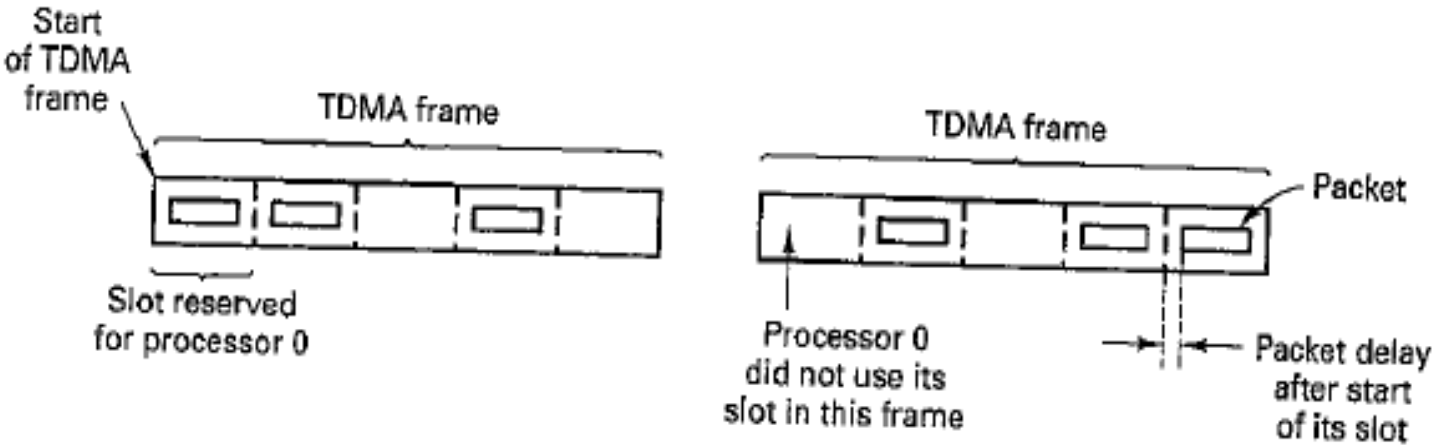


Fig. 4-28. TDMA (Time Division Multiple Access) frames.

[1]

Real-Time Communication

- Real-time connections need to be established between distant machines to provide predictability. [1]
- QoS negotiated in advance to guarantee maximum delay, jitter, and minimum bandwidth. [1]
- Additional resources needed: memory buffers, table entries, CPU cycles, and link capacity. [1]
- Fault tolerance: Duplicate packets over duplicate links (information and physical redundancy). [1]

Real-Time Communication

- Time-Triggered Protocol (TTP) used in the MARS real-time distributed system. [1]
- Redundancy [1] :
  - Each node is comprised of one or more CPUs.
  - Nodes connected by dual TDMA networks.
  - Every node maintains the global state of the system: mode, time, and membership.
- Clock synchronization is critical - each node uses packet skew to adjust clocks [1].
- Node gets rejected by all receiving nodes if [1] :
  - Message is missing from expected slot.
  - Expected acknowledgement is missing.
  - Expected global state doesn't match (CRC).
- Periodic 'initialization' packet broadcast with global state. [1]

Real-Time Communication

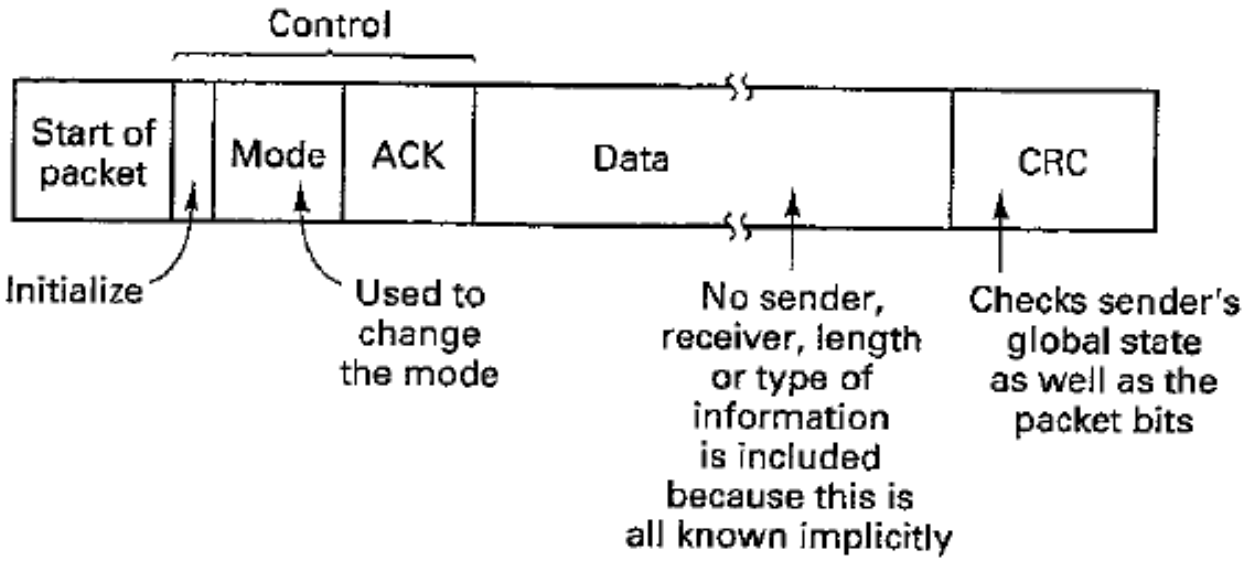


Fig. 4-29. A typical TTP packet.

[1]

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

Real-Time Scheduling

- Scheduler keeps track of state of each task (Running, Ready, or Blocked) and decides which task should be running. [3]
- Scheduling algorithm choices [1]:
  - Hard vs. Soft real-time
  - Preemptive vs. Nonpreemptive
  - Dynamic vs. Static
  - Centralized vs. Decentralized
- A set of *m* periodic tasks is schedulable if the total workload can be handled by *N* processors. [1]

Real-Time Scheduling

- Dynamic Scheduling - scheduling decisions are made during run-time. [1]
- Algorithms [1]:
  - Rate Monotonic Algorithm - task priority is based on execution frequency.
  - Earliest Deadline First - scheduler keeps tasks queued in closest-deadline-first order.
  - Least Laxity - scheduler prioritizes tasks based on least time until deadline.
- None of these algorithms are provably optimal in a distributed system [1].

Real-Time Scheduling

- Static Scheduling - scheduling decisions are made in advance. [1]
- Tasks are partitioned and statically bound to processors. [2]
- The scheduling algorithm is executed before run-time and uses heuristics to find a good schedule [1].
- Each processor’s scheduler simply follows the order of execution stored in a table. [1]

Real-Time Scheduling

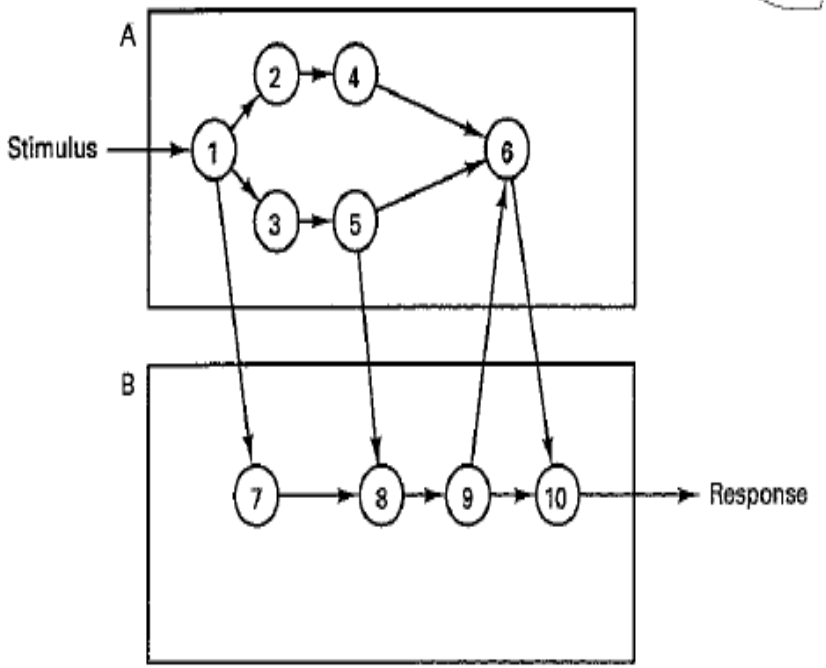


Fig. 4-30. Ten real-time tasks to be executed on two processors.

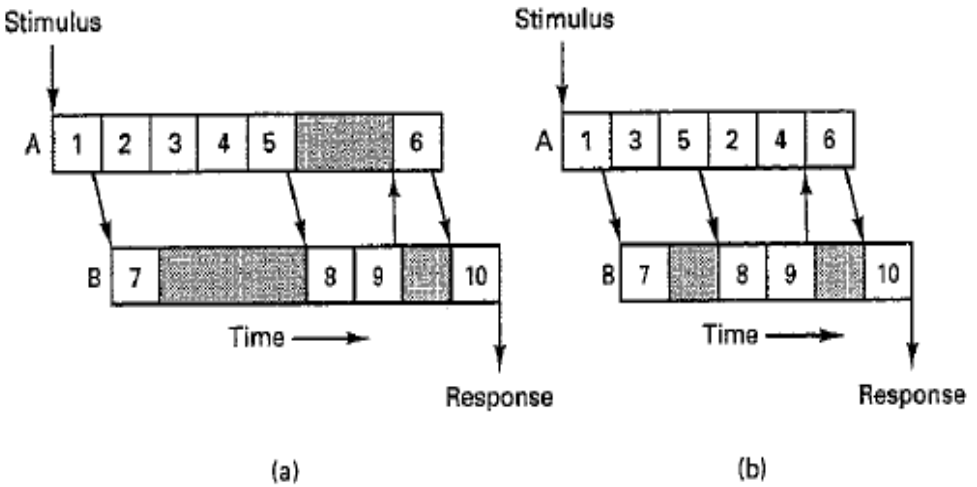


Fig. 4-31. Two possible schedules for the tasks of Fig. 4-30.

[1]

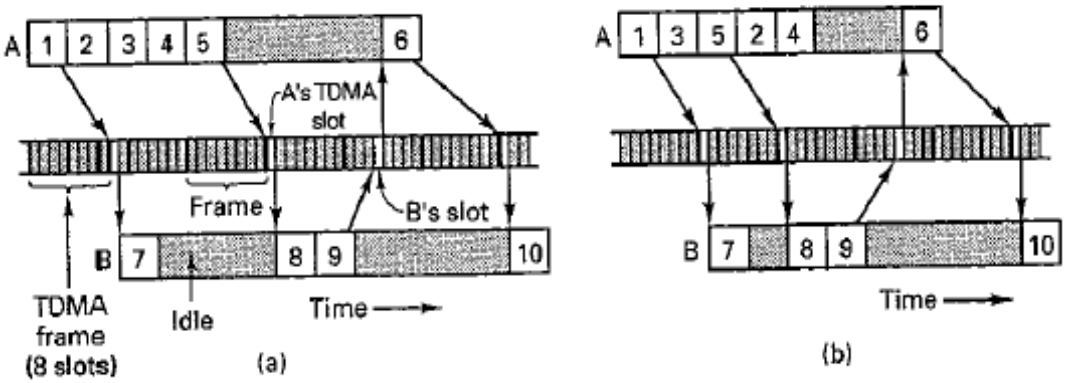


Fig. 4-32. Two schedules, including processing and communication.

REAL-TIME DISTRIBUTED SYSTEMS

- What is a Real-Time System?
- Design Issues
- Real-Time Communication
- Real-Time Scheduling
- History of Real-Time Systems
- References

History of Real-Time Systems

- 1958 - Whirlwind/SAGE (Semi-Automatic Ground Environment) computer built by MIT and IBM for NORAD is first system to operate in real time. Distributed network of radars for tracking bombers.
- 1960 - SABER (Semi-Automatic Business Environment Research) , now SABRE, computer reservations system developed by IBM for American Airlines.
- 1972 - RDOS (Real-time Disk Operating System) released by Data General for their Nova and Eclipse minicomputers.
- 1980s - Commercial RTOSes developed: pSOS (Software Components Group), VRTX (Ready Systems), VxWorks (Wind River Systems), QNX (Quantum Software), LynxOS (LynuxWorks) .

History of Real-Time Systems

- Recent Research:
  - 1996 – BAG: Real-Time Distributed OS [4]
  - 1998 – SROS: Dynamically Scalable Distributed RTOS [5]
  - 2002 – TMO-Linux: Linux-based RTOS supporting execution of Time-triggered Message-triggered Objects [6]
  - 2004 – Autonomic Distributed Real-Time Systems [7]