4. Processes and Processors in Distributed Systems

- 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.

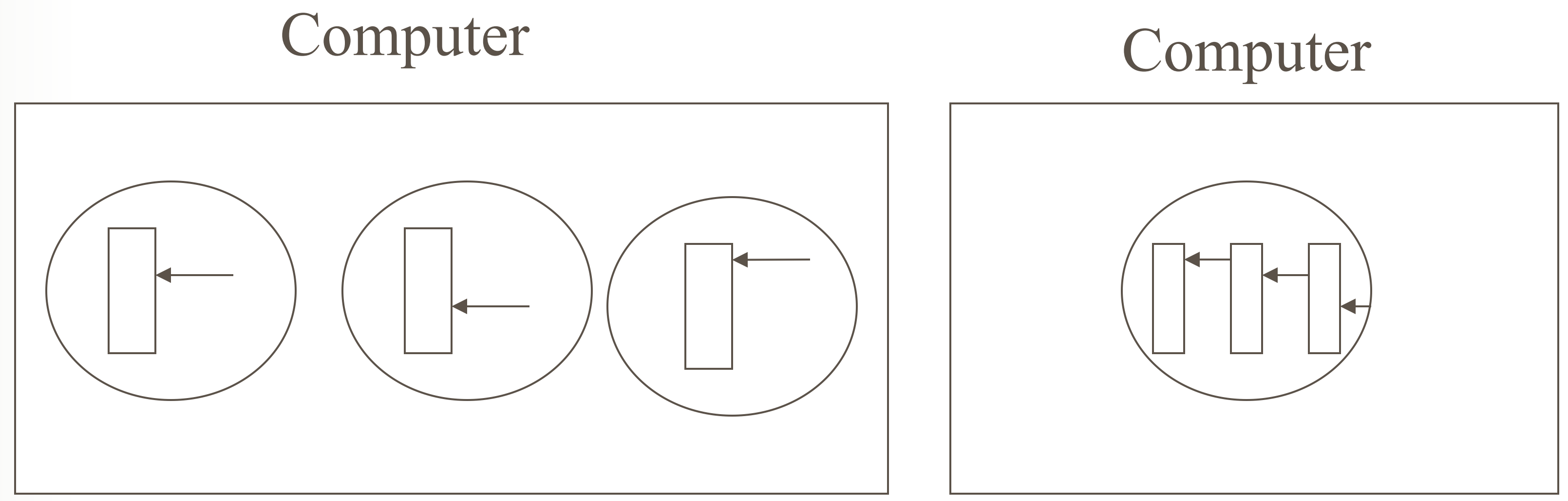
Introduction to threads

- Thread is a lightweighted 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.

- 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.

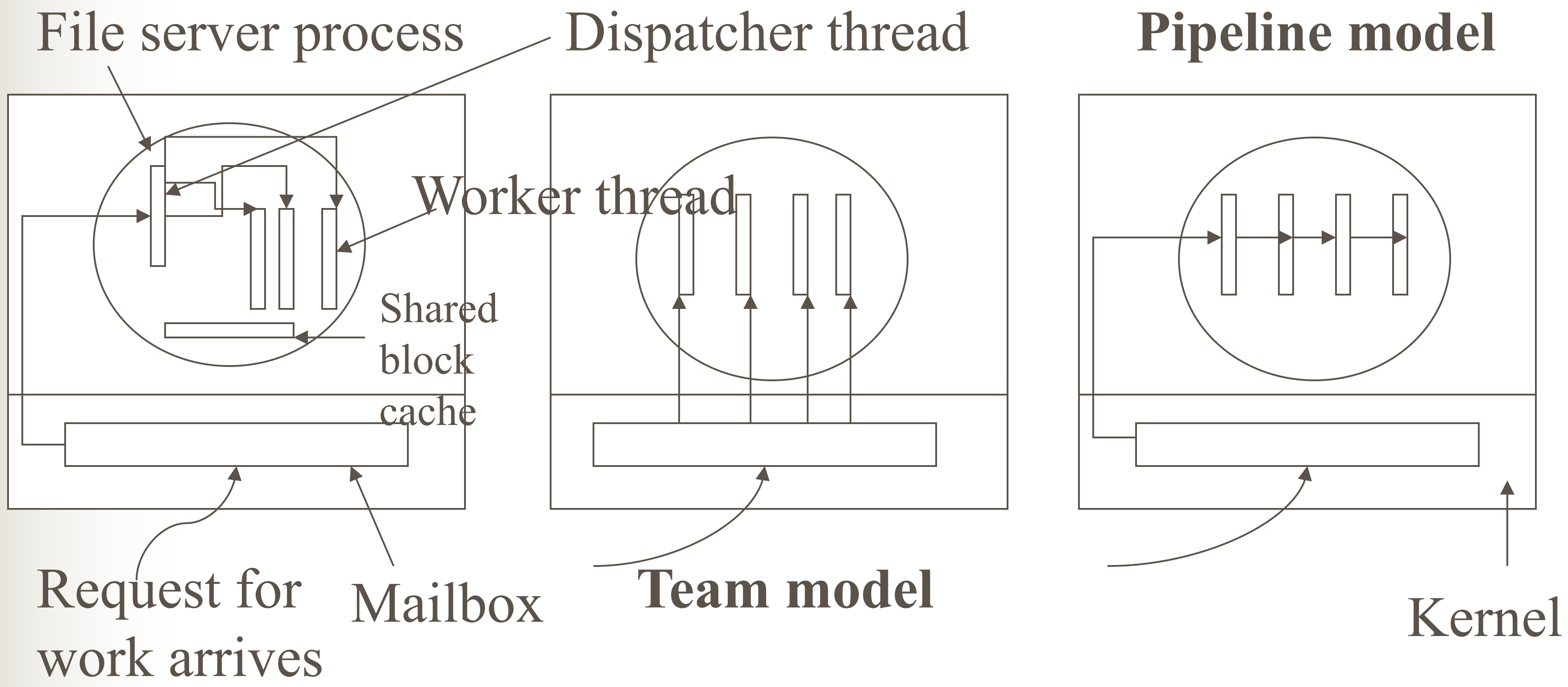
- There is no protection between threads:
- (1) it is not necessary (2) it should not be necessary: a process is always owned by a single user, who has created multiple threads so that they can cooperate, not fight.

Threads



Thread usage

Dispatcher/worker 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.

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.

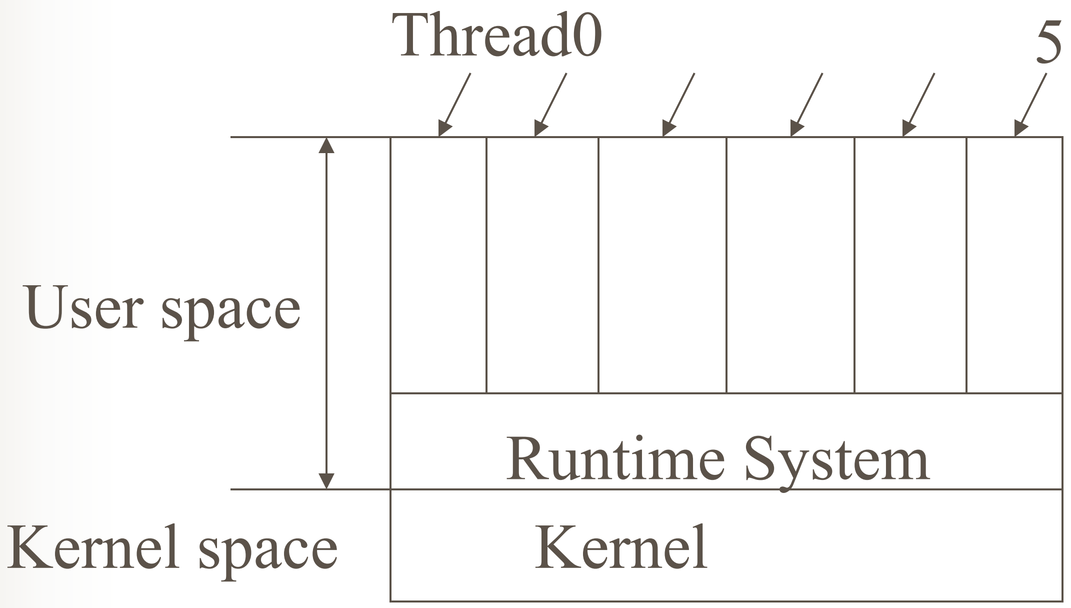
Mutex

- 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.

  Instead, it returns a status code indicating failure.

Implementing a threads package

- Implementing threads in user space



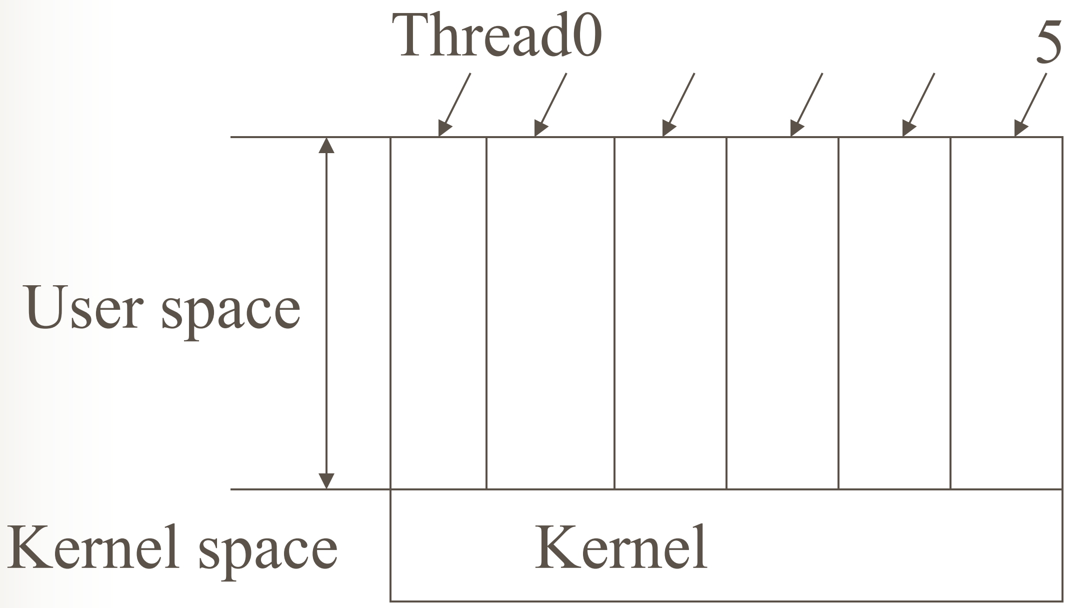
Advantage

- 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 another thread can be used.

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.

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**. Otherwise, **diskless workstations**.

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 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 network 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

Using Idle Workstation

- The earliest attempt to use idle workstations is command:
- rsh machine command
- The first argument names a machine and the second names a command to run.

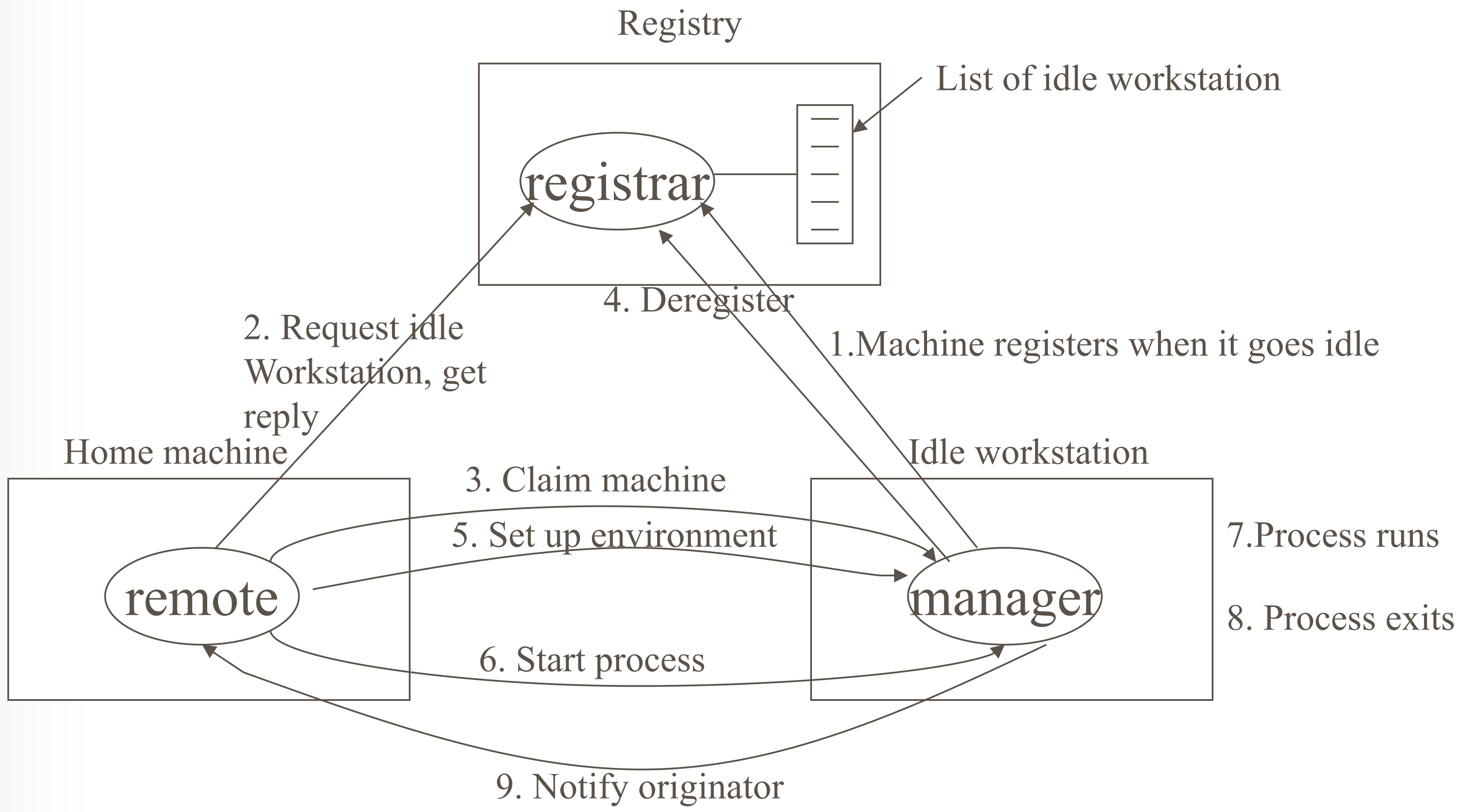
Flaws

- 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.

What is an 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.
- 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. 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 counter).

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 v, process rate u. mean response time T=1/(u-v).
- If there are n processors, each with input rate v and process rate u.
- If we put them together, input rate will be nv and process rate will be nu. Mean response time will be T=1/(nu-nv)=(1/n)T.

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.

The goals of allocation

- 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. E.g. a 1-sec job that takes 5 sec. The ratio is 5/1.

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

How to measure a processor is overloaded or underloaded?

- 1 Count the processes in the machine? Not accurate because even the machine is idle there are some daemons running.
- 2 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.
- 3 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.

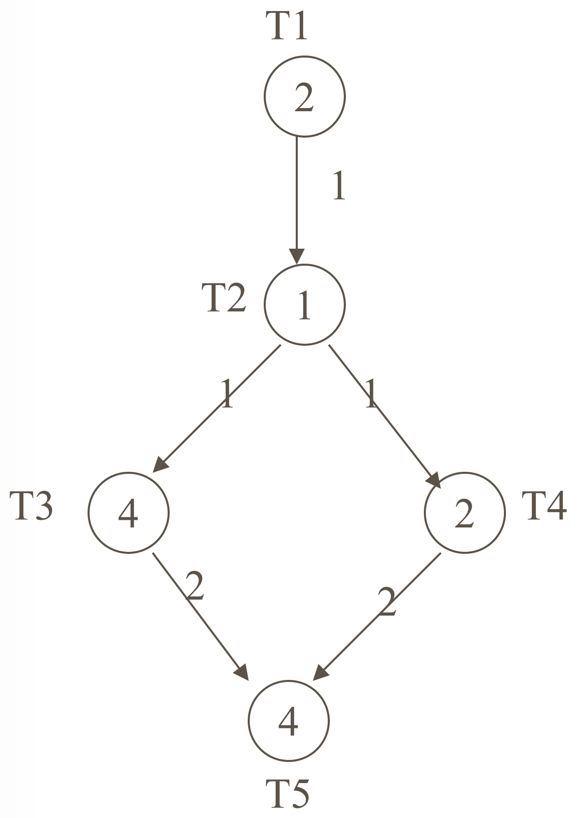
- How to calculate complexity?

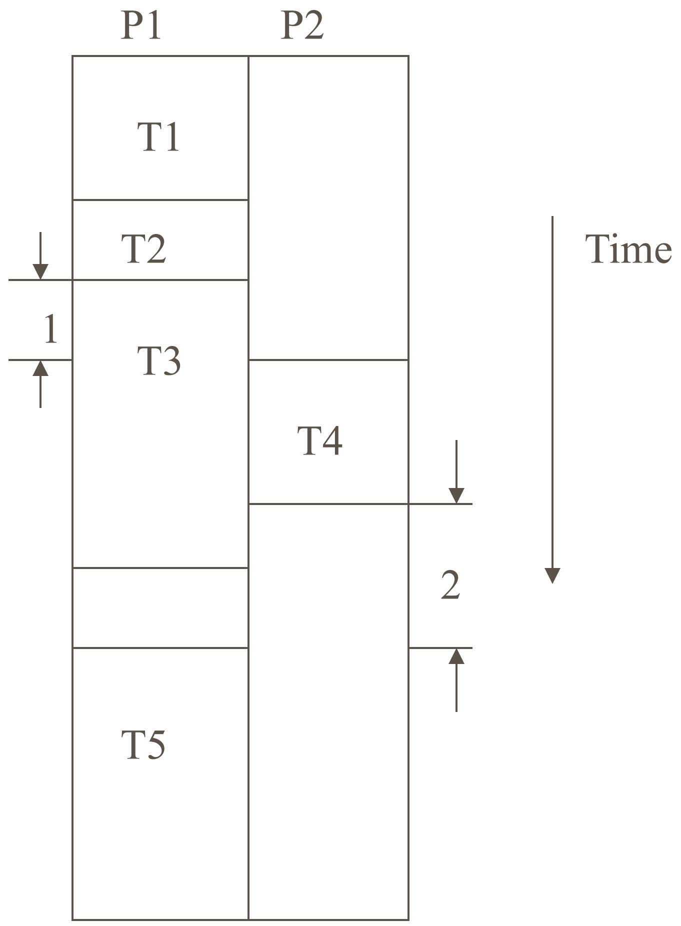
  If an algorithm performs a little better than others but requires much complex implementation, better use the simple one.

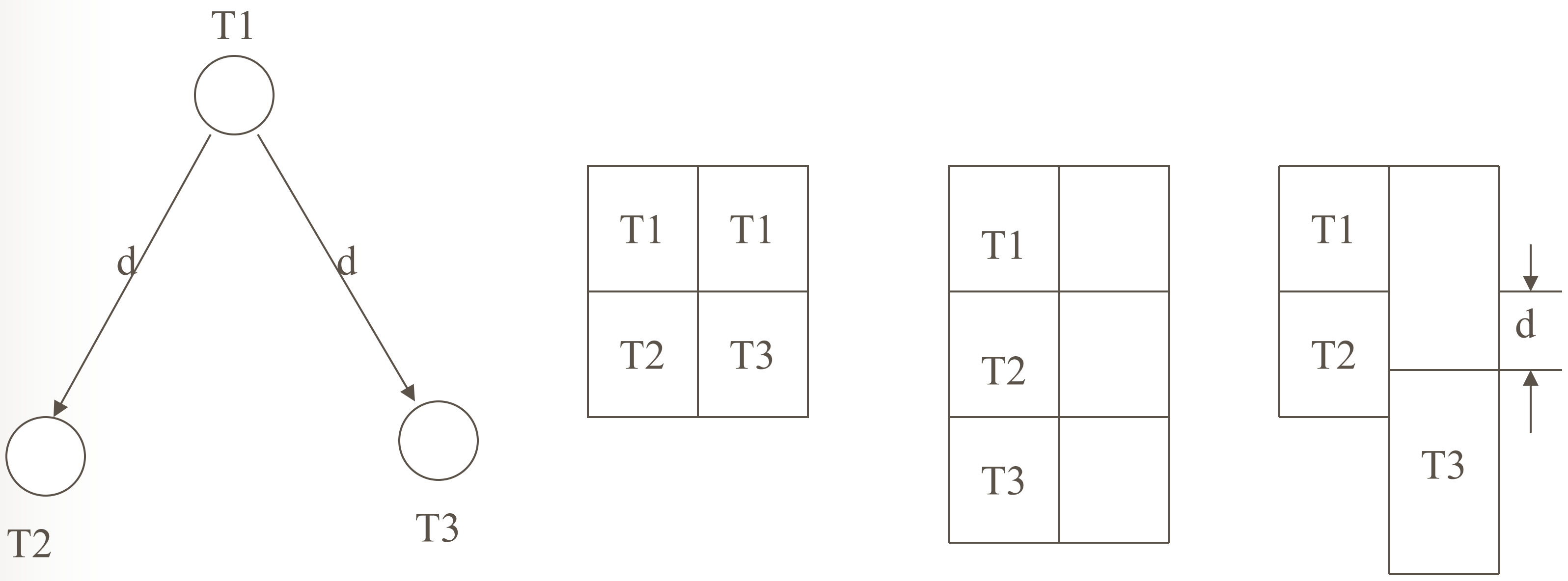
- How to deal with stability?

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

Load distribution based on precedence graph

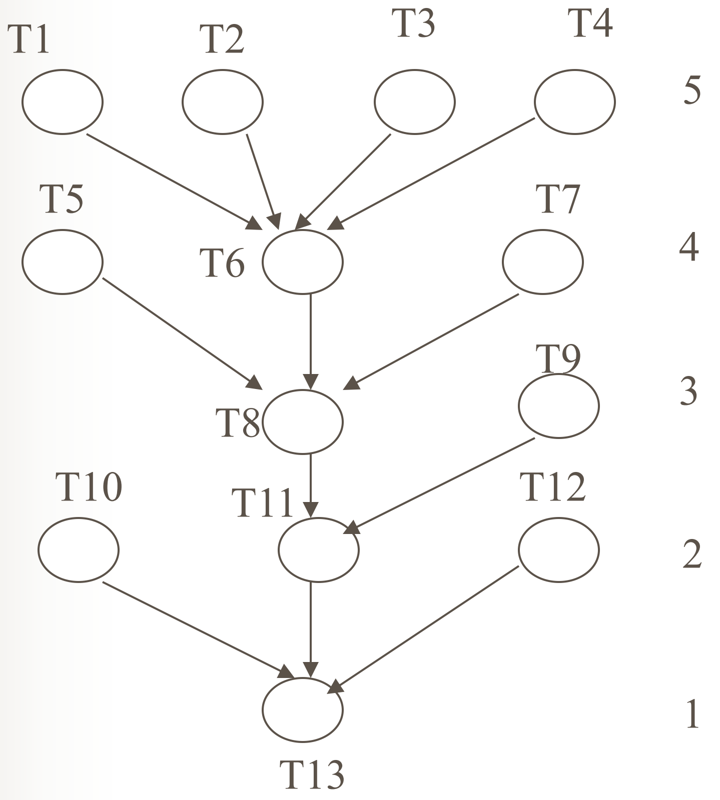





T1
d
d
T2
T3
T1
T1
T2
T3
T1
T2
T3
T1
T2
d
T3

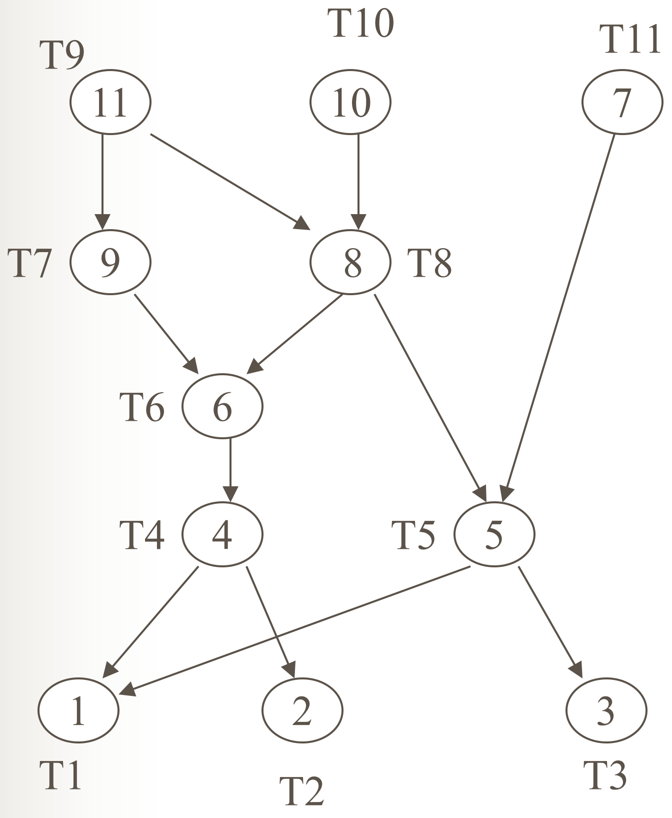
Two Optimal Scheduling Algorithms

- The precedence graph is a tree



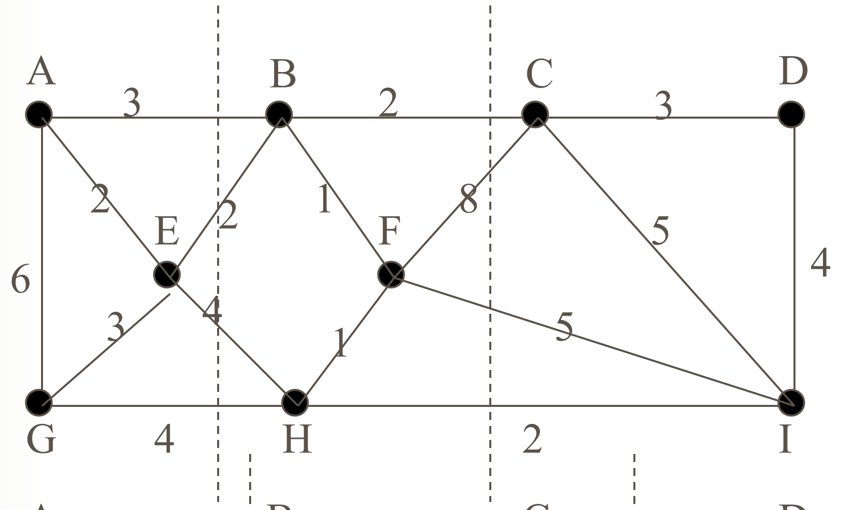
T1	T2	T3
T4	T5	T7
T6	T9	T10
T8	T12	
T11		
T13		

- There are only two processors

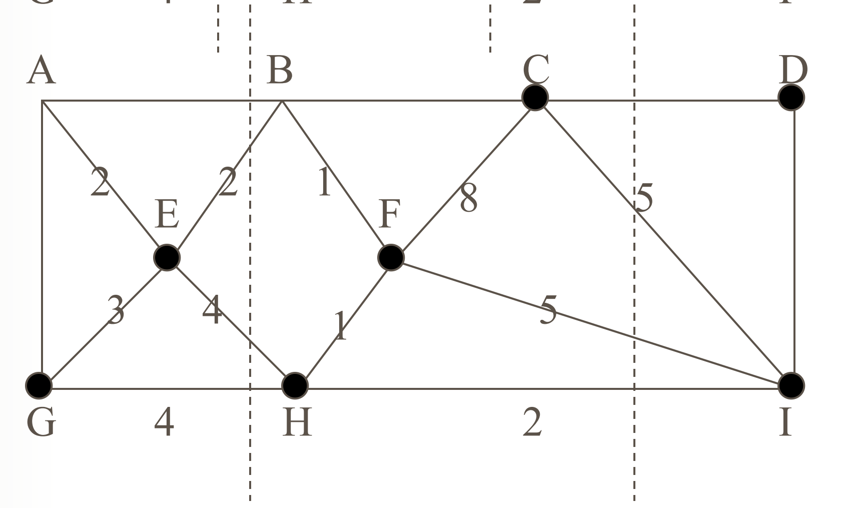


T9	T10
T7	T8
T11	T6
T5	T4
T3	T2
T1	

A graph-theoretic deterministic algorithm



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



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

Dynamic Load Distribution

- **Components of dynamic load distribution**
- Initiation policy
- Transfer policy
- Selection policy
- Profitability policy
- Location policy
- Information policy

Dynamic load distribution algorithms

- Load balancing algorithms can be classified as follows:

- Global vs. Local
- Centralized vs. decentralized
- Noncooperative vs. cooperative
- Adaptive vs. nonadaptive

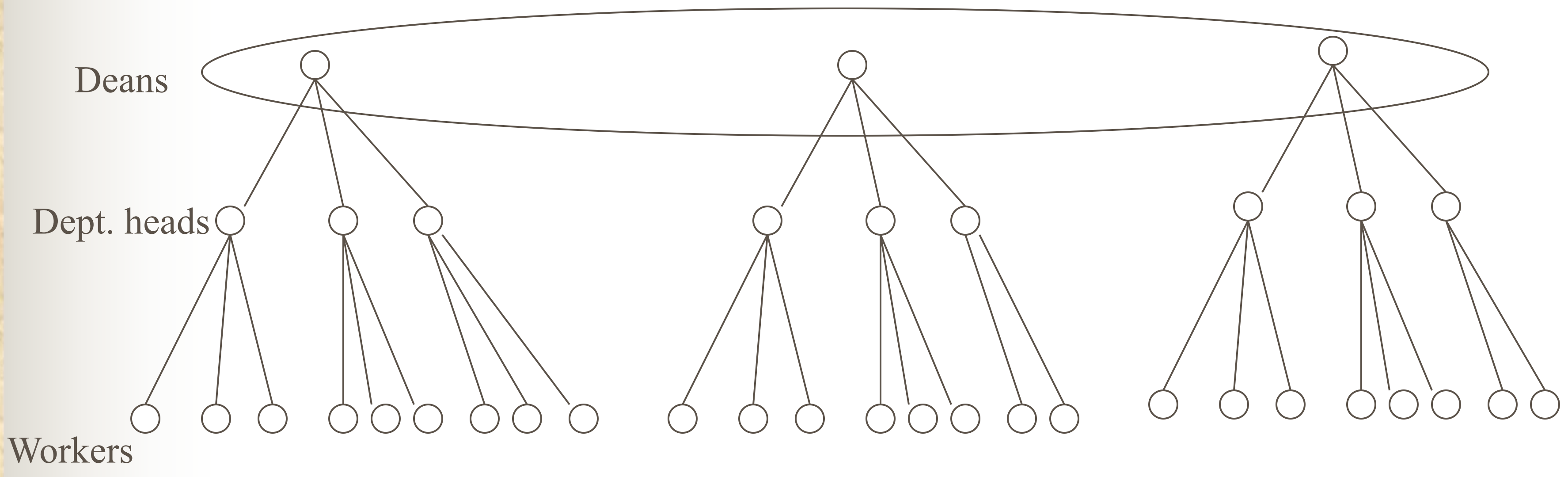
A centralized algorithm

- **Up-down algorithm**: a coordinator maintains a **usage table** with one entry per personal workstation.

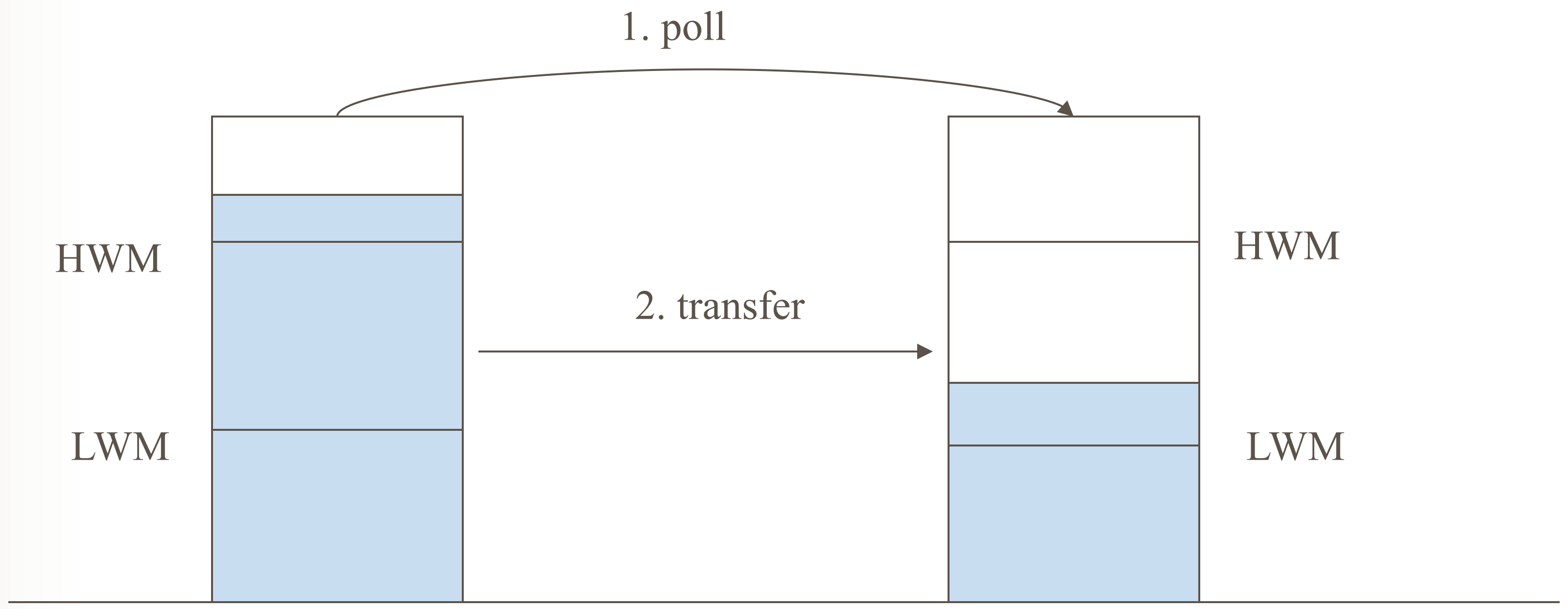
1. When a workstation owner is running processes on other people's machines, it accumulates penalty points, a fixed number per second. These points are added to its usage table entry.
2. When it has unsatisfied requests pending, penalty points are subtracted from its usage table entry.

- 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.
- When a processor becomes free, the pending request whose owner has the lowest score wins.

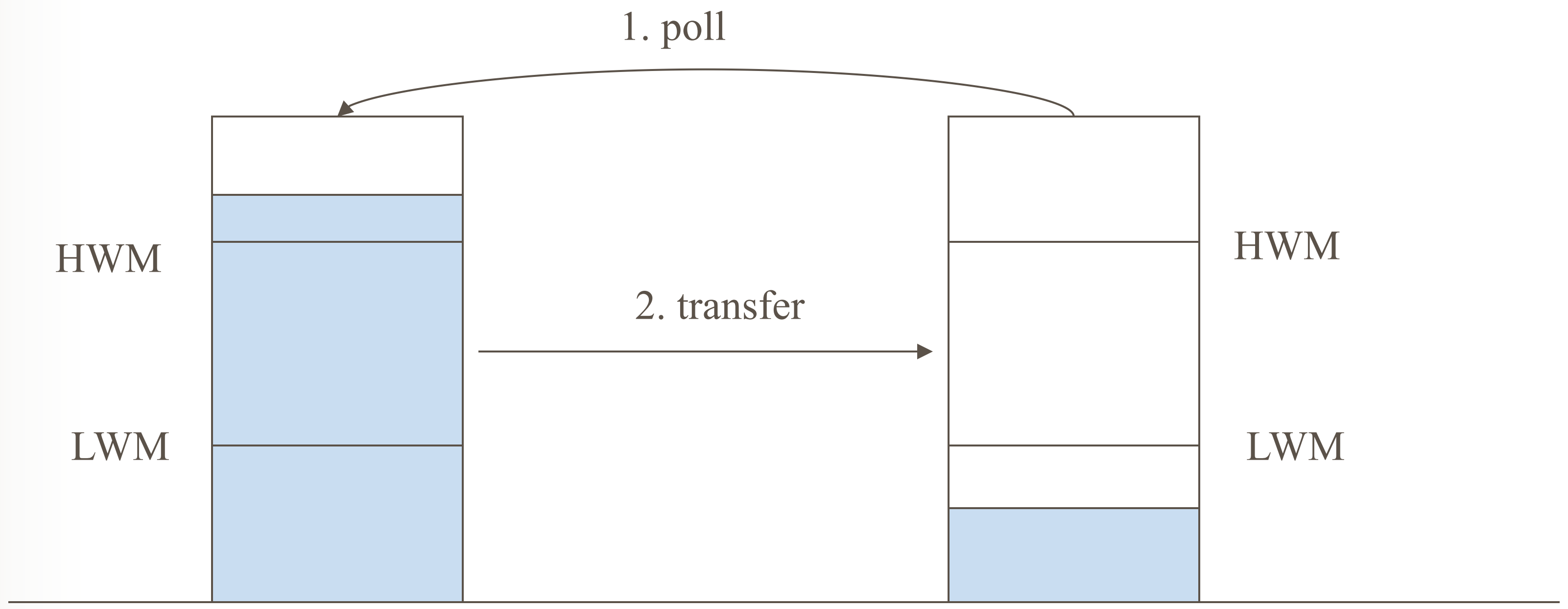
A hierarchical algorithm



A sender-initiated algorithm



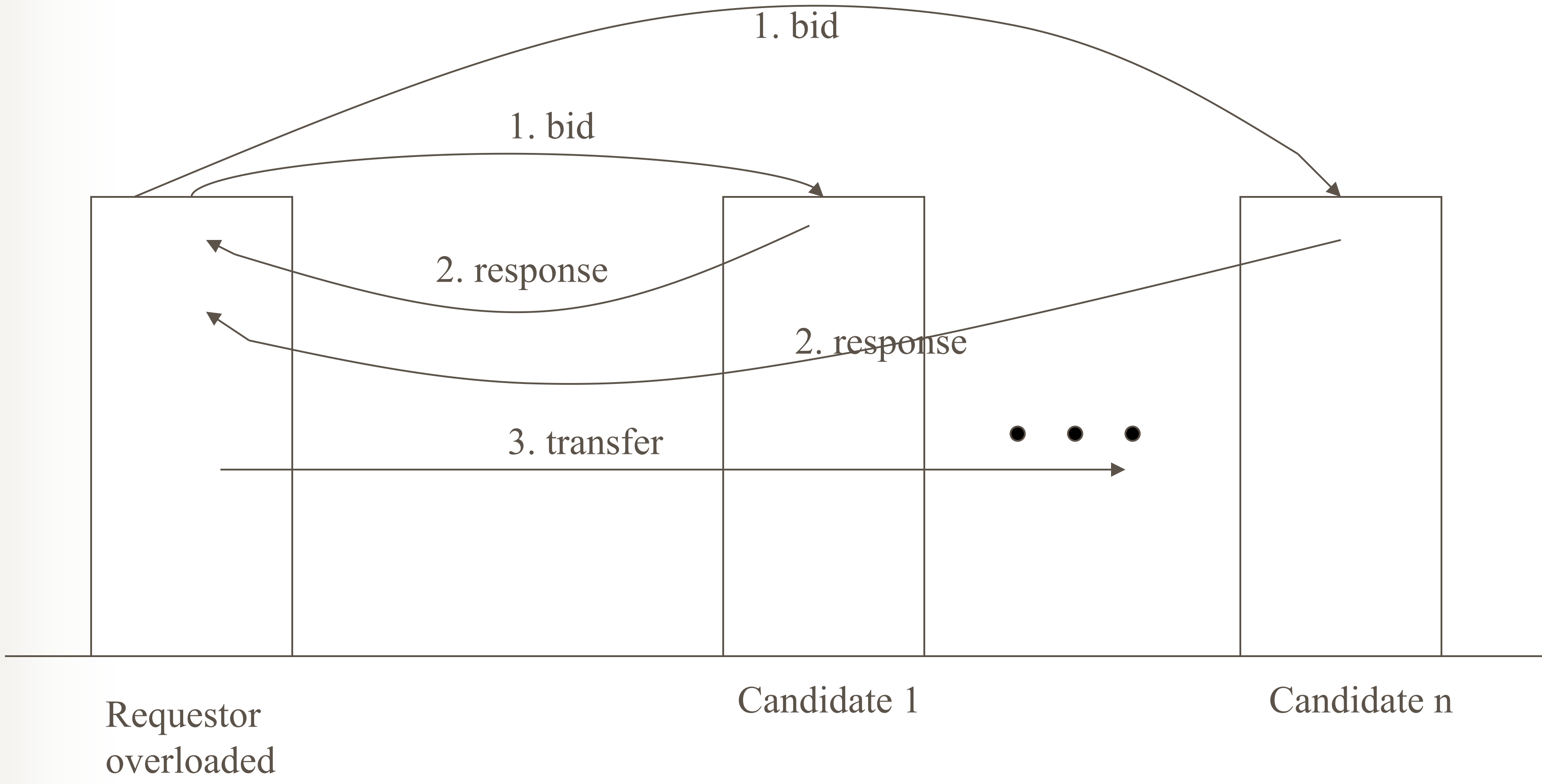
A receiver-initiated algorithm



A bidding algorithm

- This acts like an economy. Processes want CPU time. Processors give the price. Processes pick up the process that can do the work and at a reasonable price and processors pick up the process that gives the highest price.

Bidding algorithm



- *Iterative* (also called *nearest neighbor*) *algorithm*: rely on successive approximation through load exchanging among neighboring nodes to reach a global load distribution.
- *Direct algorithm:* determine senders and receivers first and then load exchanges follow.

Direct algorithm

- the average system load is determined first. Then it is broadcast to all the nodes in the system and each node determines its status: overloaded or underloaded. We can call an overloaded node a *peg* and an underloaded node a *hole*.
- the next step is to fill holes with pegs preferably with minimum data movements.

Nearest neighbor algorithms: diffusion

- $L_u(t+1)=L_u(t)+\Sigma_{v \varepsilon A(u)} (\alpha_{u,v} (L_v(t)-L_u(t))+\Phi_u(t))$
- A(u) is the neighbor set of u.

  $0<=\alpha_{u,v}<=1$ is the diffusion parameter which determines the amount of load exchanged between two neighboring nodes u and v.

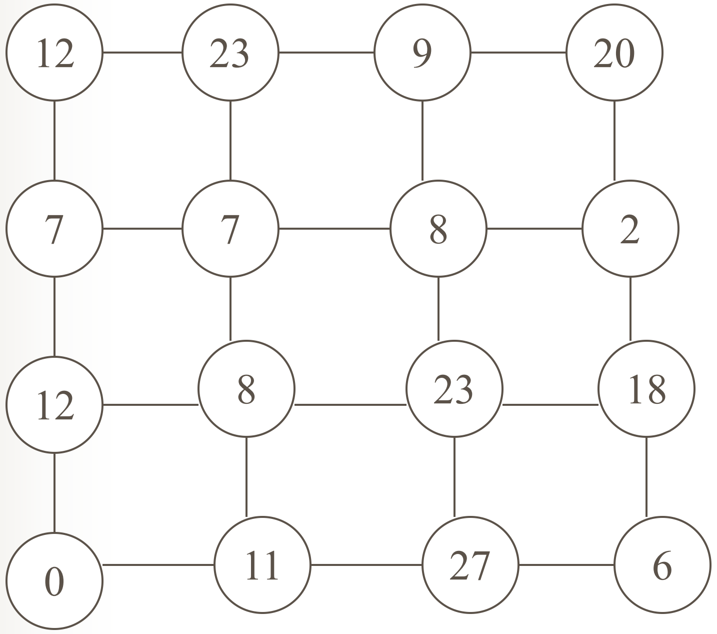
$\Phi_u(t)$) is the new incoming load between t and t+1.

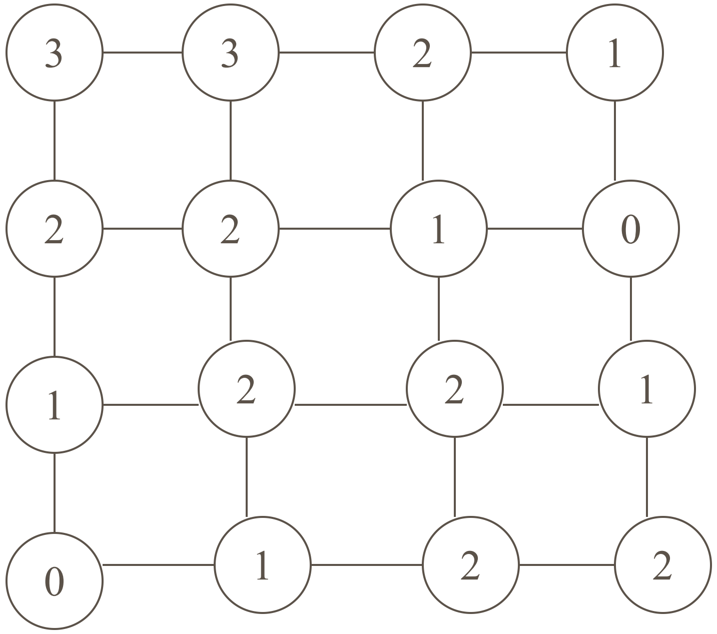
Nearest neighbor algorithm: gradient

- One of the major issues is to define a reasonable contour of gradients. The following is one model. The *propagated pressure* of a processor u, p(u), is defined as

- If u is lightly loaded, p(u) = 0
- Otherwise, p(u) = 1 + min{p(v)|v є A(u)}

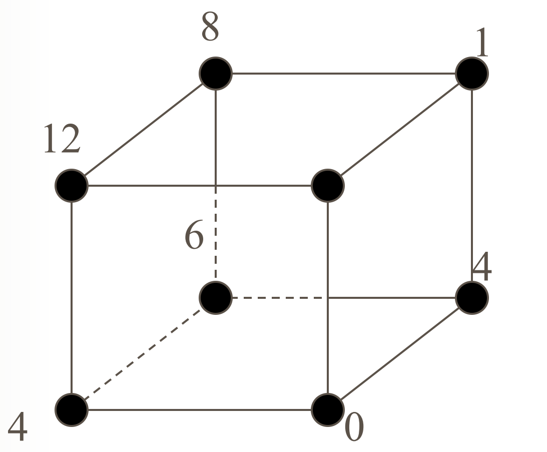
Nearest neighbor algorithm: gradient

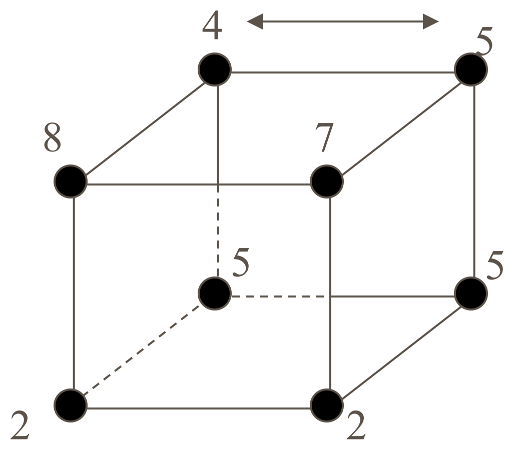


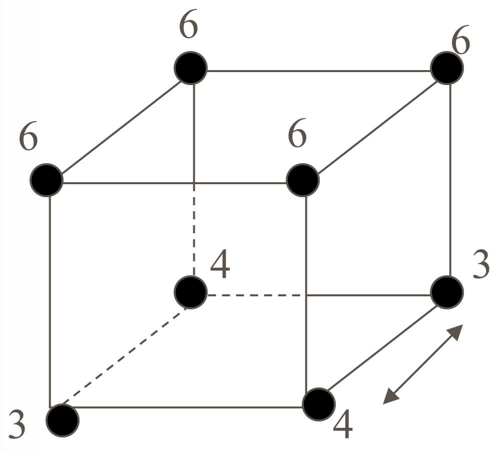


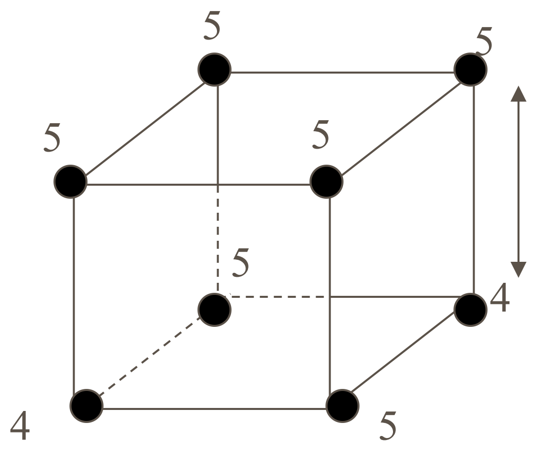
A node is lightly loaded if its load is < 3.

Nearest neighbor algorithm: dimension exchange

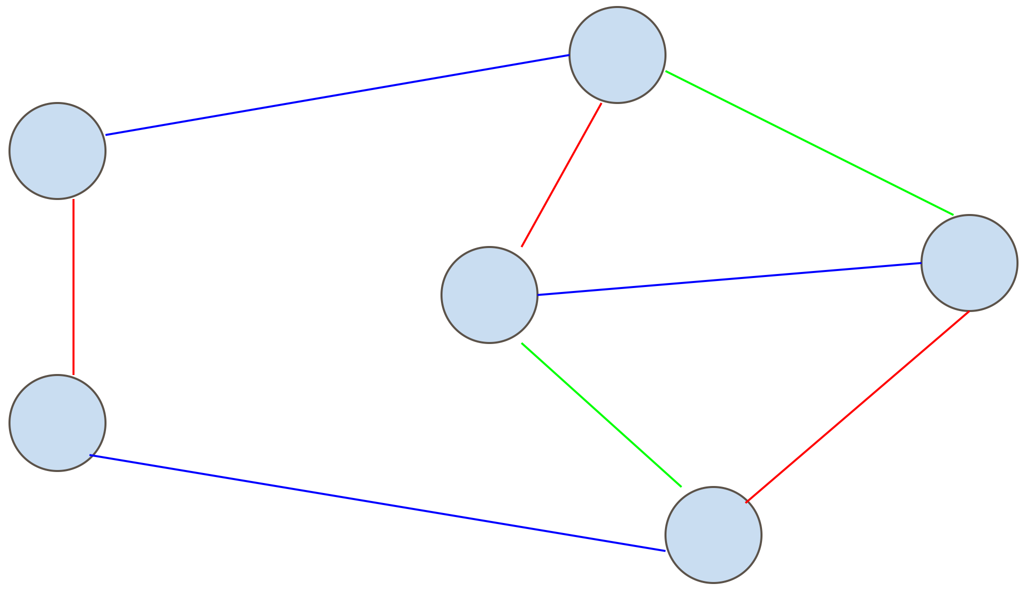








Nearest neighbor algorithm: dimension exchange extension



Fault tolerance

- **component faults**
- • Transient faults: occur once and then disappear. E.g. a bird flying through the beam of a microwave transmitter may cause lost bits on some network. If retry, may work.
- • Intermittent faults: occurs, then vanishes, then reappears, and so on. E.g. A loose contact on a connector.
- • Permanent faults: continue to exist until the fault is repaired. E.g. burnt-out chips, software bugs, and disk head crashes.

System failures

- There are two types of processor faults:
-
- 1 Fail-silent faults: a faulty processor just stops and does not respond
- 2 Byzantine faults: continue to run but give wrong answers

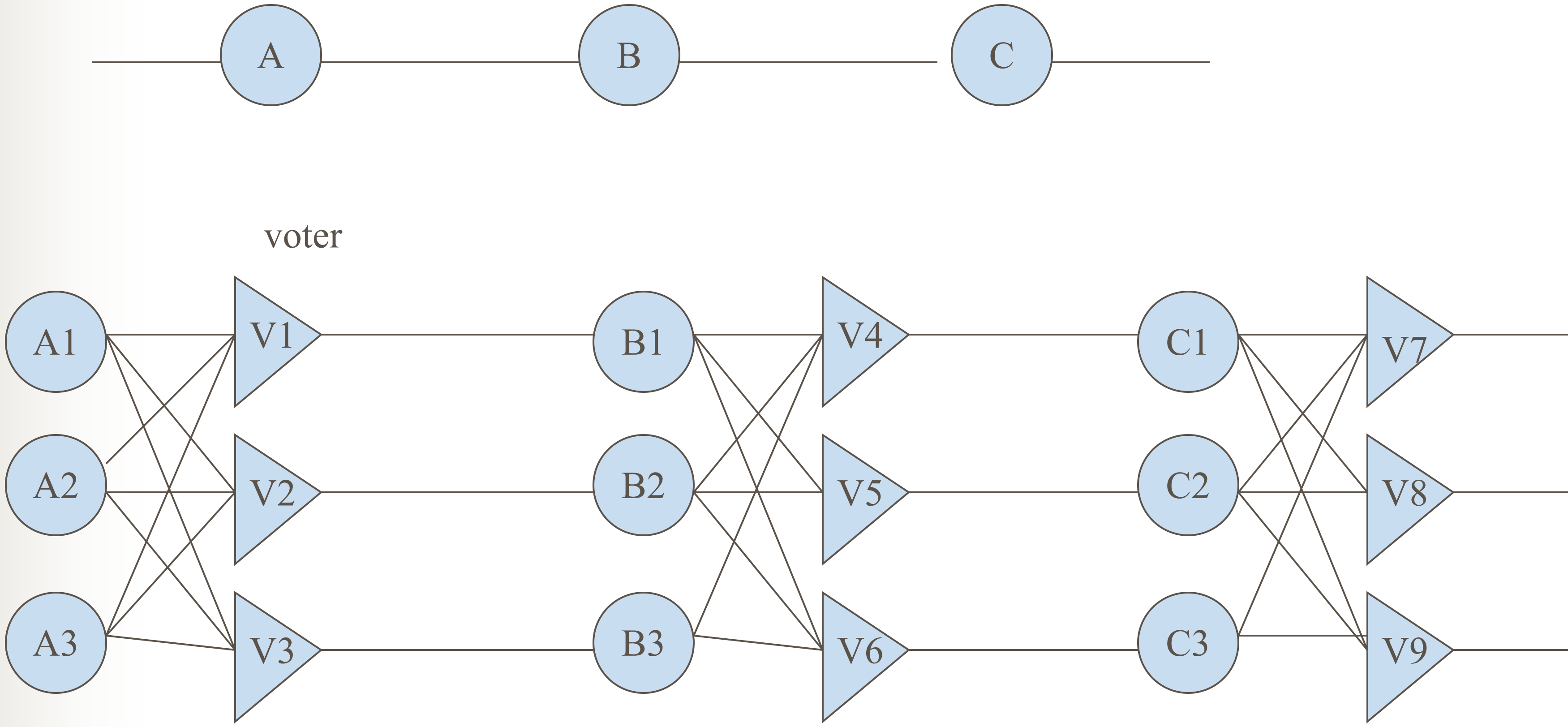
Synchronous versus Asynchronous systems

- Synchronous systems: 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**. Otherwise, it is **asynchronous**.

Use of redundancy

- There are three kinds of fault tolerance approaches:
- 1 Information redundancy: extra bit to recover from garbled bits.
- 2 Time redundancy: do again
- 3 Physical redundancy: add extra components. There are two ways to organize extra physical equipment: **active replication** (use the components at the same time) and **primary backup** (use the backup if one fails).

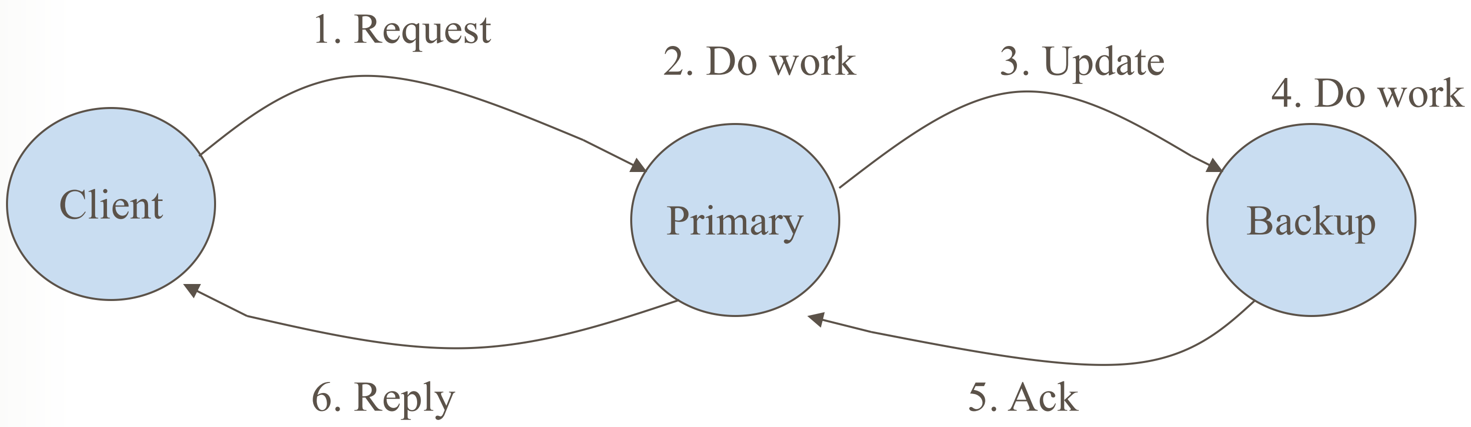
Fault Tolerance using active replication



How much replication is needed?

- A system is said to be **k fault tolerant** if it can survive faults in *k* components and still meet its specifications.
- *K+1* processors can fault tolerant *k* fail-stop faults. If *k* of them fail, the one left can work. But need *2k+1* to tolerate *k* Byzantine faults because if *k* processors send out wrong replies, but there are still *k+1* processors giving the correct answer. By majority vote, a correct answer can still be obtained.

Fault Tolerance using primary backup

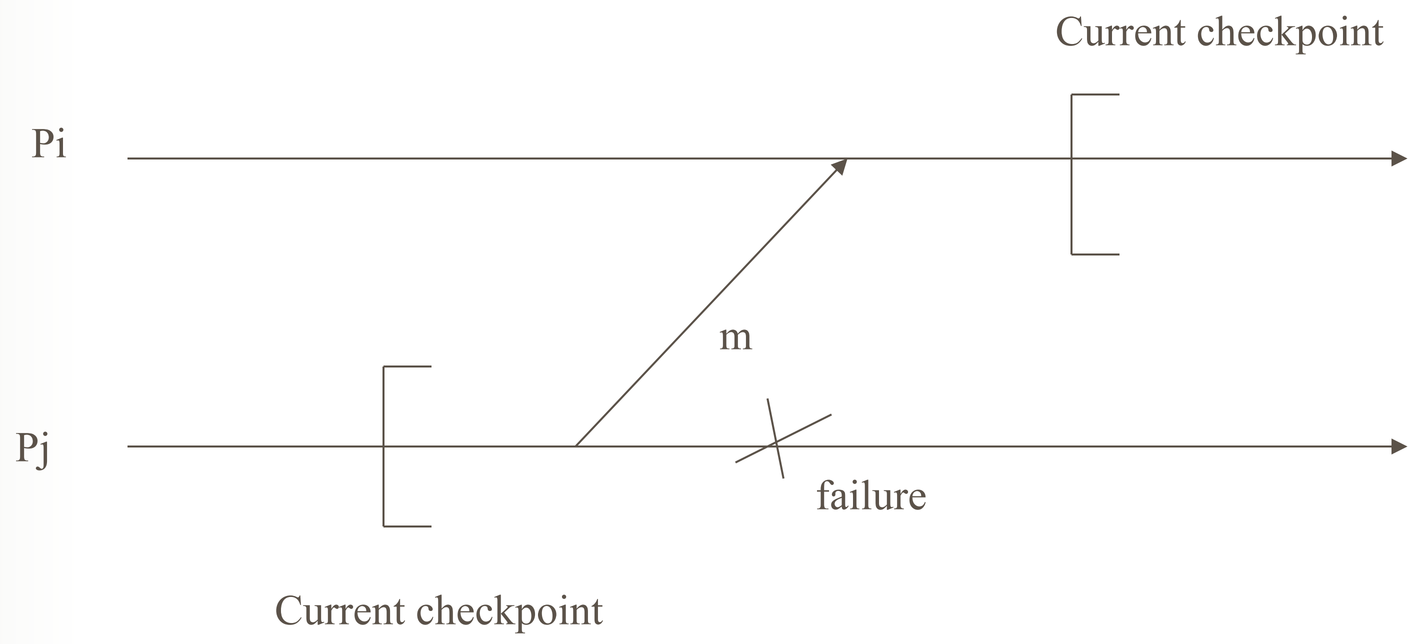


Handling of Processor Faults

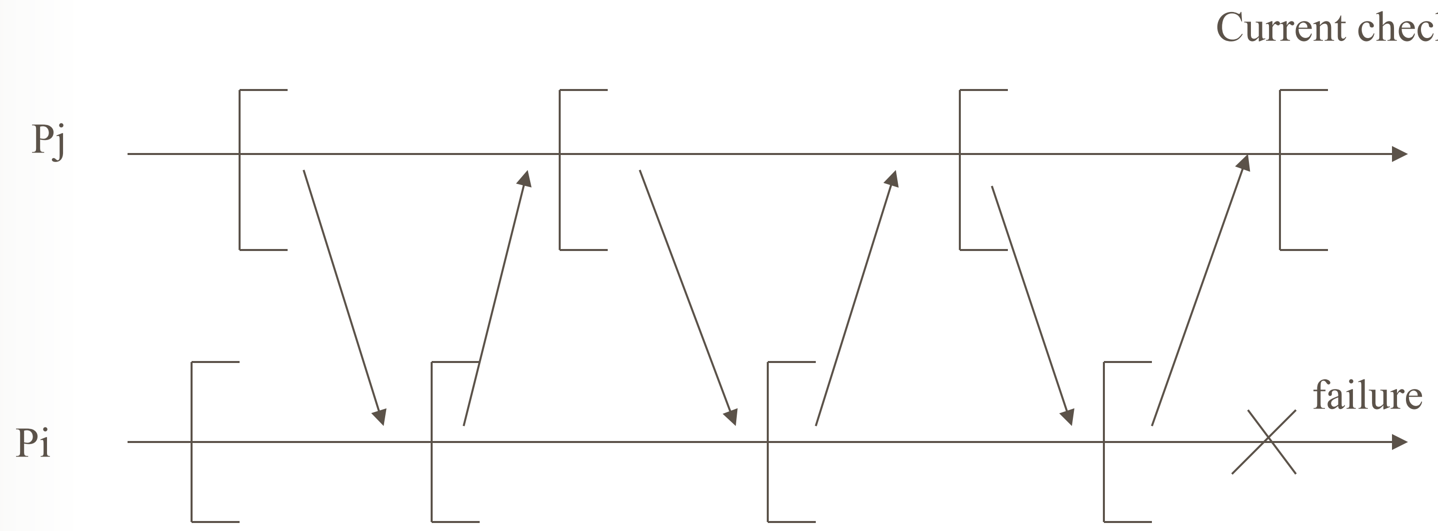
- **Backward recovery** – checkpoints.
- In the checkpointing method, two undesirable situations can occur:
- *Lost message*. The state of process Pi indicates that it has sent a message *m* to process Pj. Pj has no record of receiving this message.
- *Orphan message*. The state of process Pj is such that it has received a message *m* from the process Pi but the state of the process Pi is such that it has never sent the message *m* to Pj.

- A **strongly consistent set** of checkpoints consist of a set of local checkpoints such that there is no orphan or lost message.
- A **consistent set** of checkpoints consists of a set of local checkpoints such that there is no orphan message.

Orphan message



Domino effect



Synchronous checkpointing

- a processor Pi needs to take a checkpoint only if there is another process Pj that has taken a checkpoint that includes the receipt of a message from Pi and Pi has not recorded the sending of this message.
- In this way no orphan message will be generated.

Asynchronous checkpointing

- Each process takes its checkpoints independently without any coordination.

Hybrid checkpointing

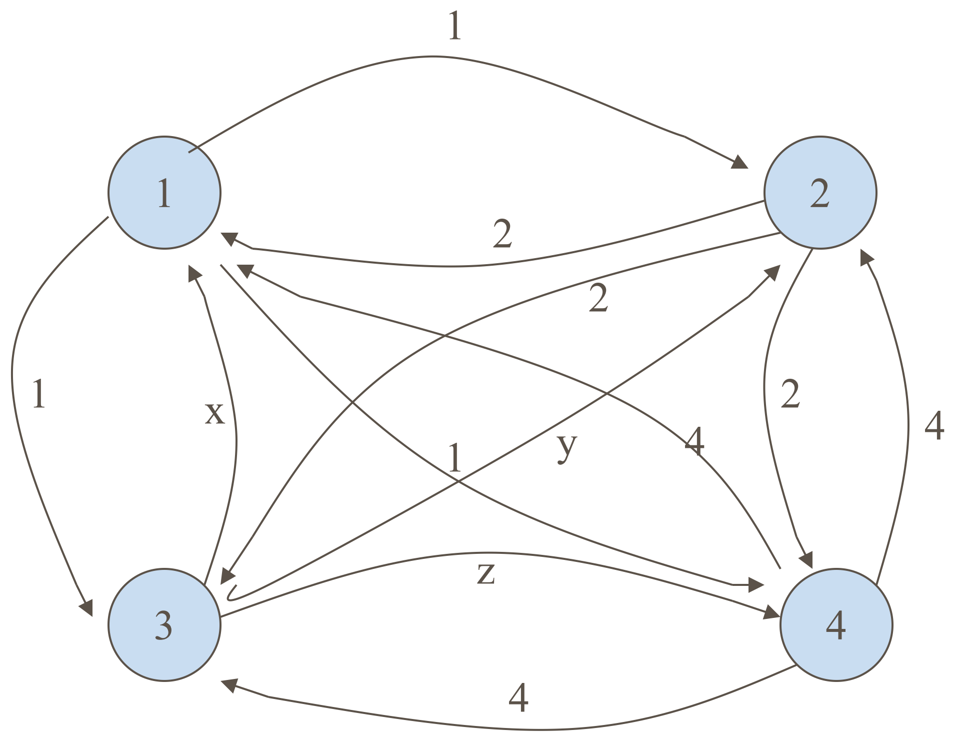
- Synchronous checkpoints are established in a longer period while asynchronous checkpoints are used in a shorter period. That is, within a synchronous period there are several asynchronous periods.

Agreement in Faulty Systems

- **two-army problem**
- Two blue armies must reach agreement to attack a red army. If one blue army attacks by itself it will be slaughtered. They can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.
- They can never reach an agreement on attacking.

- Now assume the communication is perfect but the processors are not. The classical problem is called the **Byzantine generals problem.** **N** generals and **M** of them are traitors. Can they reach an agreement?

Lamport’s algorithm



N = 4
M = 1

- After the first round

1 got (1,2,x,4); 2 got (1,2,y,4);

3 got (1,2,3,4); 4 got (1,2,z,4)

- After the second round

1 got (1,2,y,4), (a,b,c,d), (1,2,z,4)

2 got (1,2,x,4), (e,f,g,h), (1,2,z,4)

4 got (1,2,x,4), (1,2,y,4), (i,j,k,l)

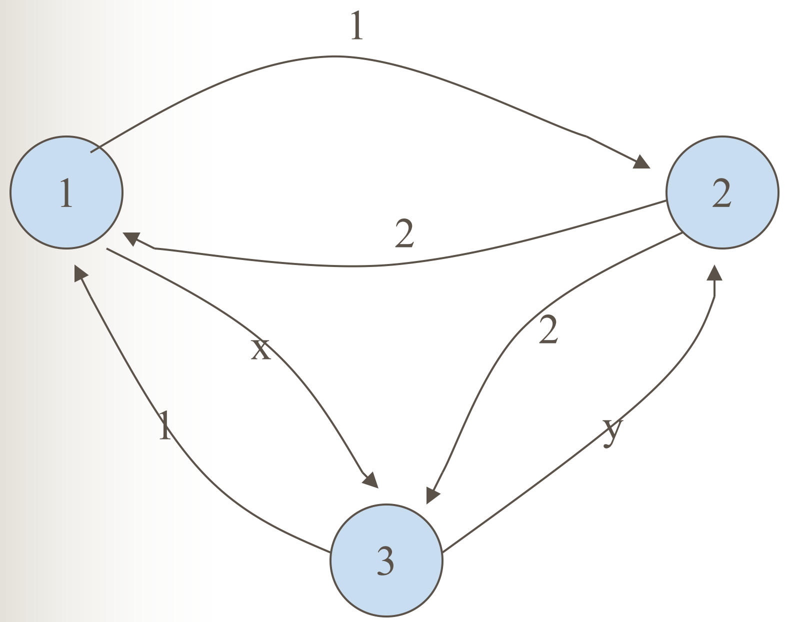
- Majority

1 got (1,2,_,4); 2 got (1,2,_,4); 4 got (1,2,_,4)

- So all the good generals know that 3 is the bad guy.

Result

- If there are m faulty processors, agreement can be achieved only if 2m+1 correct processors are present, for a total of 3m+1.
- Suppose n=3, m=1. Agreement cannot be reached.



After the first round
1 got (1,2,x); 2 got (1,2,y); 3 got (1,2,3)

After the second round
1 got (1,2,y), (a,b,c)
2 got (1,2,x), (d,e,f)

No majority. Cannot reach an agreement.

Agreement under different models

- Turek and Shasha considered the following parameters for the agreement problem.

1. The system can be synchronous (A=1) or asynchronous (A=0).
2. Communication delay can be either bounded (B=1) or unbounded (B=0).
3. Messages can be either ordered (C=1) or unordered (C=0).
4. The transmission mechanism can be either point-to-point (D=0) or broadcast (D=1).

A Karnaugh map for the agreement problem

AB \ CD	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	1

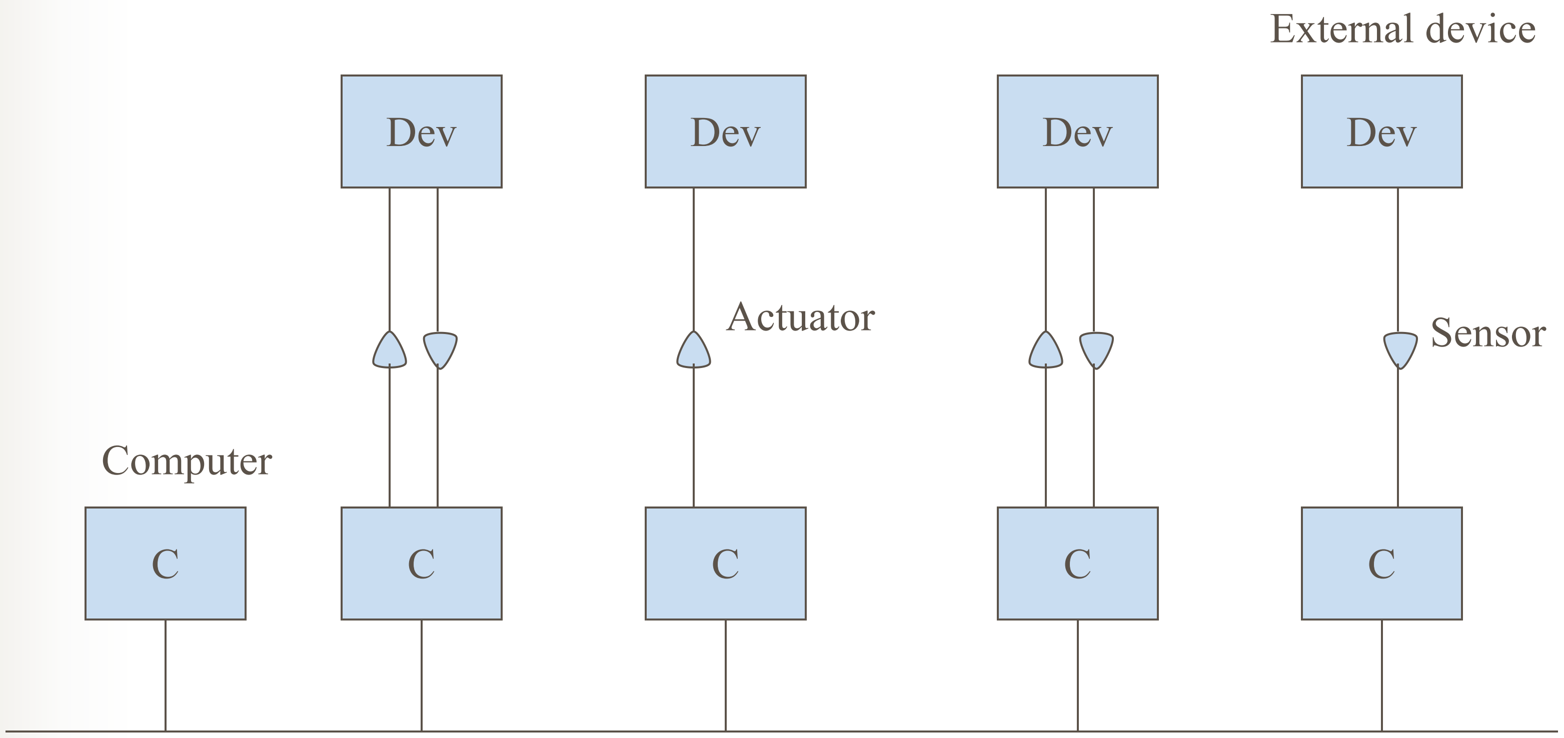
- Minimizing the Boolean function we have the following expression for the conditions under which consensus is possible:

  AB+AC+CD = True
- (AB=1): Processors are synchronous and communication delay is bounded.
- (AC=1): Processors are synchronous and messages are ordered.
- (CD=1): Messages are ordered and the transmission mechanism is broadcast.

REAL-TIME DISTRIBUTED SYSTEMS

- **What is a real-time system?**
- **Real-time programs** interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. E.g. automated factories, telephone switches, robots, automatic stock trading system.

Distributed real-time systems structure



Stimulus

- An external device generates a stimulus for the computer, which must perform certain actions before a deadline.

1. Periodic: a stimulus occurring regularly every T seconds, such as a computer in a TV set or VCR getting a new frame every 1/60 of a second.
2. Aperiodic: stimulus that are recurrent, but not regular, as in the arrival of an aircraft in an air traffic controller's air space.
3. Sporadic: stimulus that are unexpected, such as a device overheating.

Two types of RTS

- Soft real-time systems: missing an occasional deadline is all right.
- Hard real-time systems: even a single missed deadline in a hard real-time system is unacceptable, as this might lead to loss of life or an environmental catastrophe.

Design issues

- **Clock Synchronization** - Keep the clocks in synchrony is a key issue.
- **Event-Triggered versus Time-Triggered Systems**
- **Predictability**
- **Fault Tolerance**
- **Language Support**

Event-triggered real-time system

- when a significant event in the outside world happens, it is detected by some sensor, which then causes the attached CPU to get an interrupt. Event-triggered systems are thus interrupt driven. Most real-time systems work this way.
- Disadvantage: they can fail under conditions of heavy load, that is, when many events are happening at once. This **event shower** may overwhelm the computing system and bring it down, potentially causing problems seriously.

Time-triggered real-time system

- in this kind of system, a clock interrupt occurs every T milliseconds. At each clock tick sensors are sampled and actuators are driven. No interrupts occur other than clock ticks.
- T must be chosen carefully. If it too small, too many clock interrupts. If it is too large, serious events may not be noticed until it is too late.

An example to show the difference between the two

- Consider an elevator controller in a 100-story building. Suppose that the elevator is sitting on the 60th floor. If someone pushes the call button on the first floor, and then someone else pushes the call button on the 100th floor. In an event-triggered system, the elevator will go down to first floor and then to 100th floor. But in a time-triggered system, if both calls fall within one sampling period, the controller will have to make a decision whether to go up or go down, for example, using the nearest-customer-first rule.

- In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. Time-trigger designs have the opposite properties and are furthermore only suitable in a relatively static environment in which a great deal is known about system behavior in advance.

Predictability

- One of the most important properties of any real-time system is that its behavior be predictable. Ideally, it should be clear at design time that the system can meet all of its deadlines, even at peak load. It is known when event E is detected, the order of processes running and the worst-case behavior of these processes.

Fault Tolerance

- Many real-time systems control safety-critical devices in vehicles, hospitals, and power plants, so fault tolerance is frequently an issue.
- Primary-backup schemes are less popular because deadlines may be missed during cutover after the primary fails.
- In a safety-critical system, it is especially important that the system be able to handle the worst-case scenario. It is not enough to say that the probability of three components failing at once is so low that it can be ignored. Fault-tolerant real-time systems must be able to cope with the maximum number of faults and the maximum load at the same time.

Language Support

- In such a language, it should be easy to express the work as a collection of short tasks that can be scheduled independently.
- The language should be designed so that the maximum execution time of every task can be computed at compile time. This requirement means that the language cannot support general **while** loops and recursions.
- The language needs a way to deal with time itself.
- The language should have a way to express minimum and maximum delays.
- There should be a way to express what to do if an expected event does not occur within a certain interval.
- Because periodic events play an important role, it would be useful to have a statement of the form: every (25 msec){…} that causes the statements within the curly brackets to be executed every 25 msec.

Real-Time Communication

- Cannot use Ethernet because it is not predictable.
- Token ring LAN is predictable. Bounded by kn byte times. K is the machine number. N is a n-byte message .
- An alternative to a token ring is the TDMA (Time Division Multiple Access) protocol. Here traffic is organized in fixed-size frames, each of which contains n slots. Each slot is assigned to one processor, which may use it to transmit a packet when its time comes. In this way collisions are avoided, the delay is bounded, and each processor gets a guaranteed fraction of the bandwidth.

Real-Time Scheduling

- Hard real time versus soft real time
- Preemptive versus nonpreemptive scheduling
- Dynamic versus static
- Centralized versus decentralized

Dynamic Scheduling

- **1. Rate monotonic algorithm:**
- It works like this: in advance, each task is assigned a priority equal to its execution frequency. For example, a task runs every 20 msec is assigned priority 50 and a task run every 100 msec is assigned priority 10. At run time, the scheduler always selects the highest priority task to run, preempting the current task if need be.

- **2.Earliest deadline first algorithm:**
- Whenever an event is detected, the scheduler adds it to the list of waiting tasks. This list is always keep sorted by deadline, closest deadline first.

- **3.Least laxity algorithm:**
- this algorithm first computes for each task the amount of time it has to spare, called the laxity. For a task that must finish in 200 msec but has another 150 msec to run, the laxity is 50 msec. This algorithm chooses the task with the least laxity, that is, the one with the least breathing room.

Static Scheduling

- The goal is to find an assignment of tasks to processors and for each processor, a static schedule giving the order in which the tasks are to be run.

A comparison of Dynamic versus Static Scheduling

- Static is good for time-triggered design.
- 1. It must be carefully planned in advance, with considerable effort going into choosing the various parameters.
- 2. In a hard real-time system, wasting resources is often the price that must be paid to guarantee that all deadlines will be met.
- 3. An optimal or nearly optimal schedule can be derived in advance.

- Dynamic is good for event-triggered design.
- 1. It does not require as much advance work, since scheduling decisions are made on-the-fly, during execution.
- 2. It can make better use of resources than static scheduling.
- 3. No time to find the best schedule.