Chapter 4

Resource and Process Management

Content….

4.1 Desirable Features of global Scheduling algorithm, Task assignment approach, Load balancing approach, load sharing approach

4.2 Introduction to process management, 4.2, Threads, Virtualization, Clients, Servers, Code Migration

Content….

4.2 Introduction to process management,

- Process Concept
- Process Scheduling
- Operations on Processes
- Process migration / Inter process Communication(IPC)

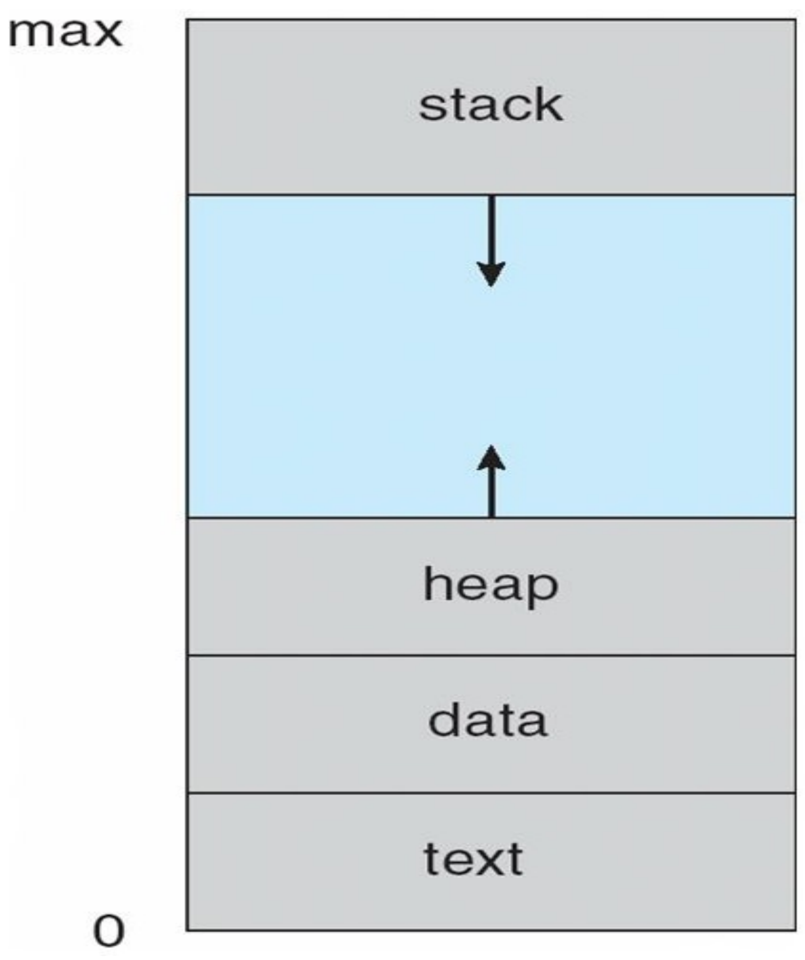
- **Threads,**
- **Virtualization,**
- **Clients, Servers,**
- **Code Migration**

Process Concept

- **A Program is an executable file which contains a certain set of instructions written to complete the specific job or operation on your computer.**
- **A Process is an execution of a specific program. It is an active entity that actions the purpose of the application.**
- **Process** – a program in execution;
- Process contain multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data

  Function parameters, return addresses, local variables

  Eg. Recursion
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time (eg malloc, new..)

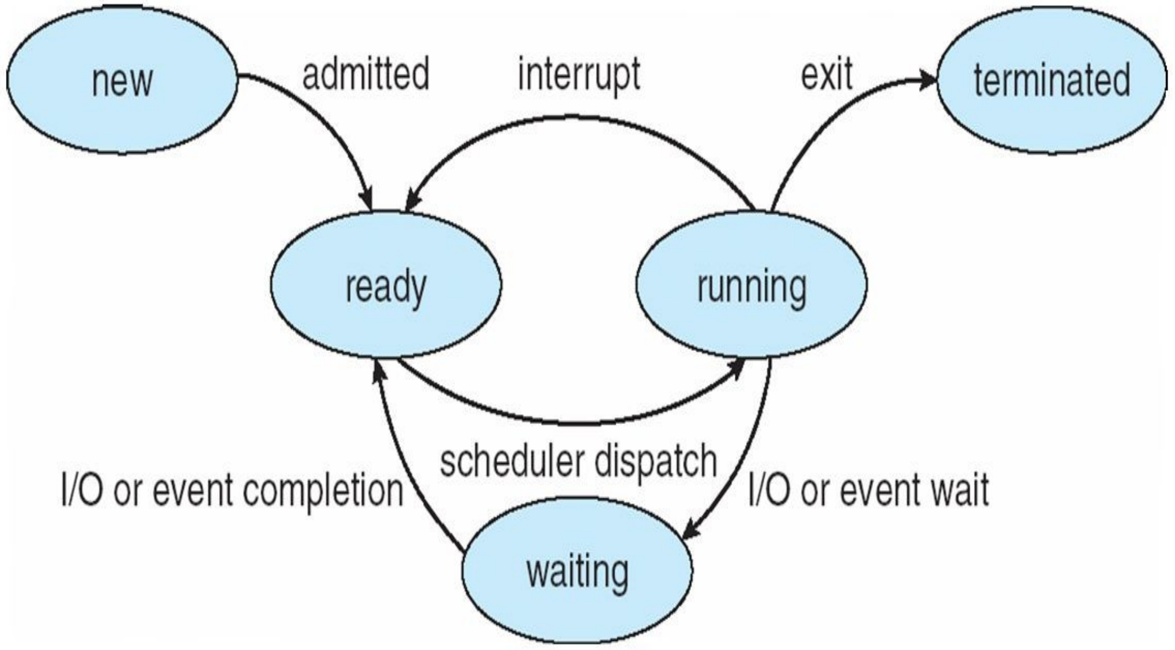


Process Concept

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
- Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process States

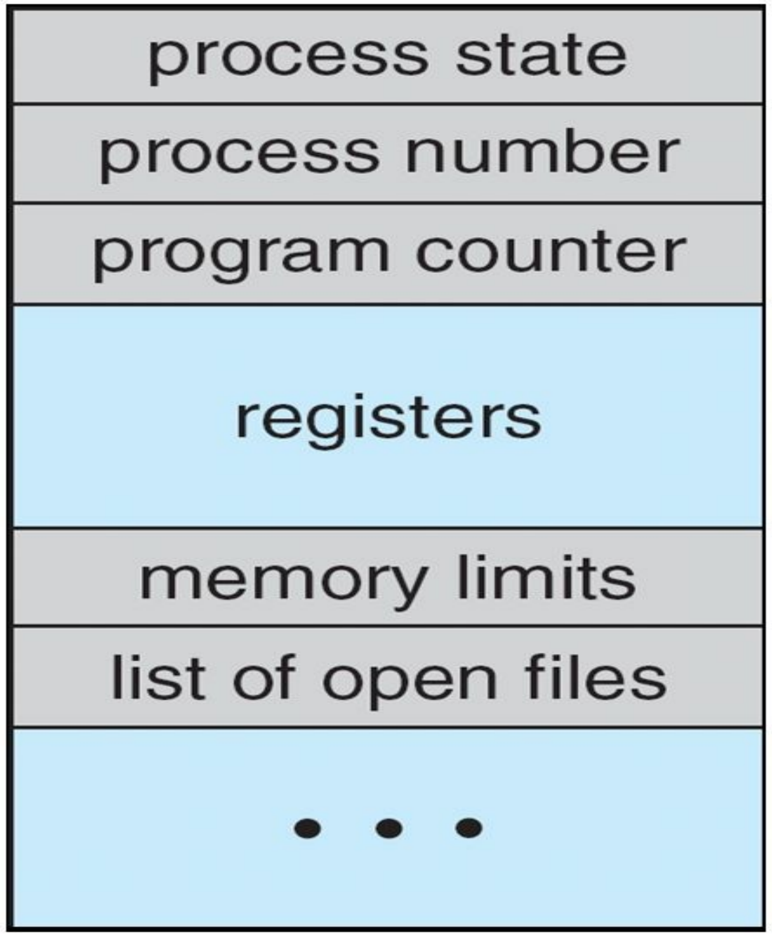
- As a process executes, it changes **state**
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



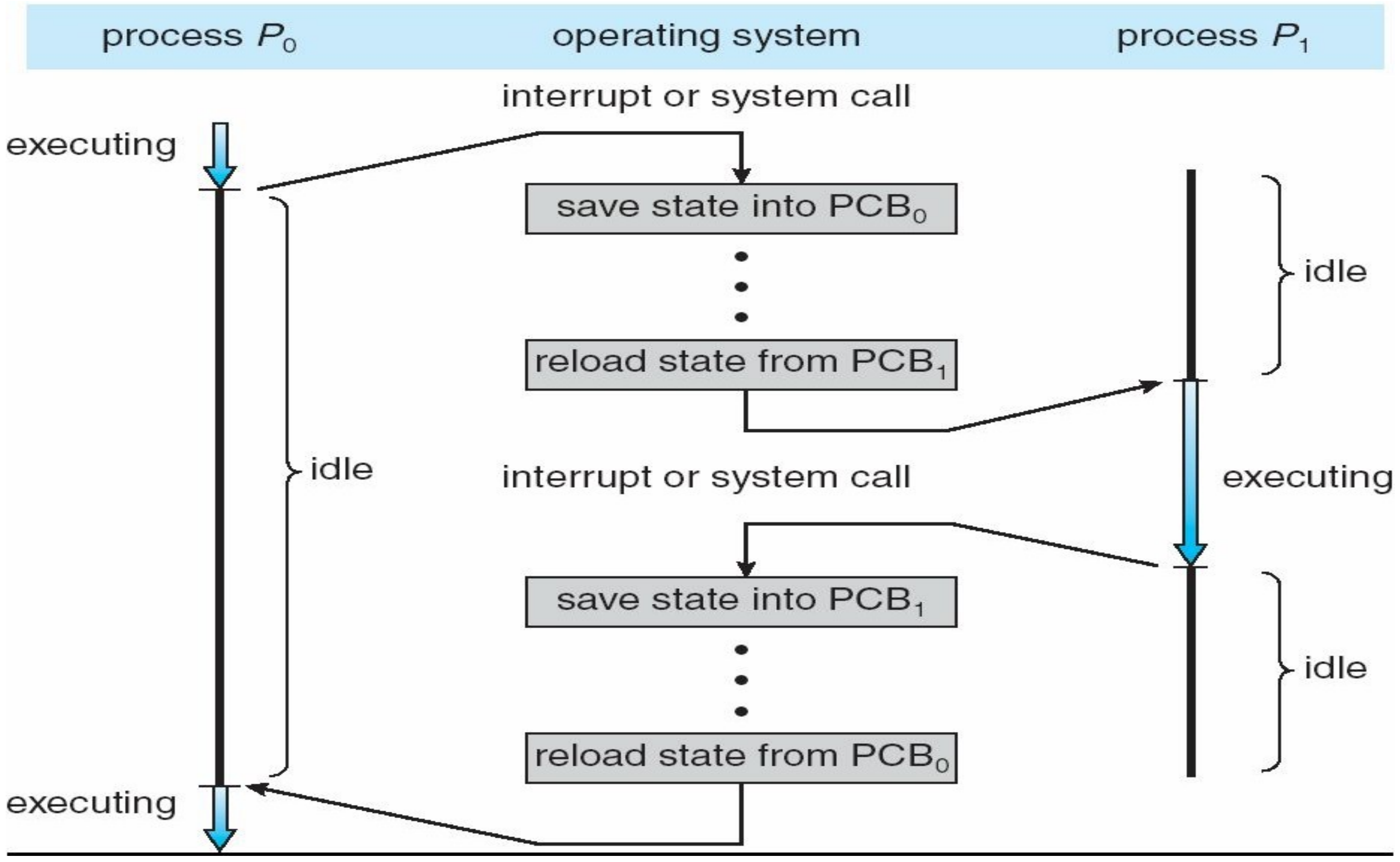
Process Control Block (PCB)

Information / Data Structure associated with each process called PCB (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



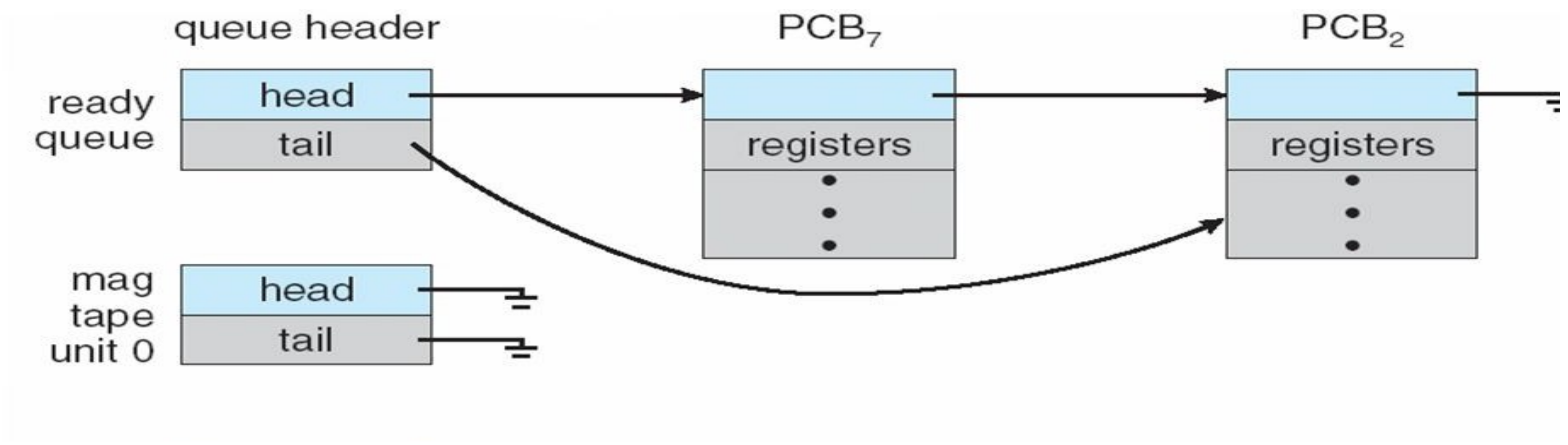
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU multiple contexts loaded at once

Process Scheduling

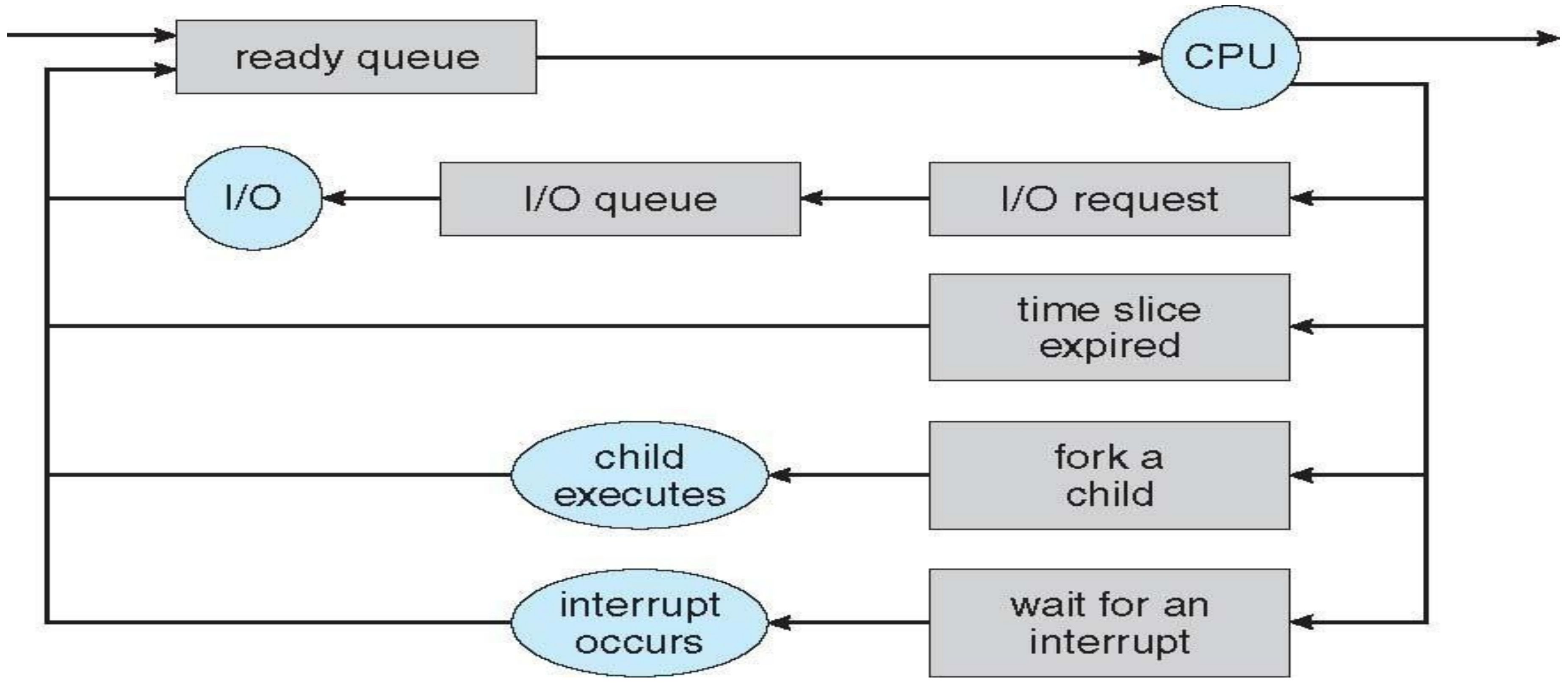
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

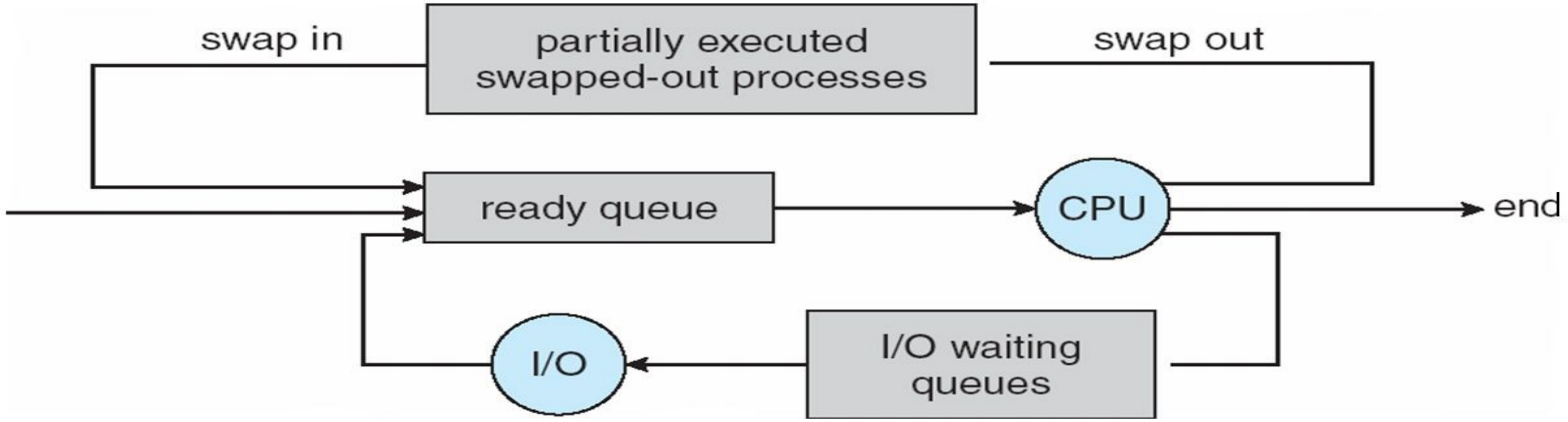


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

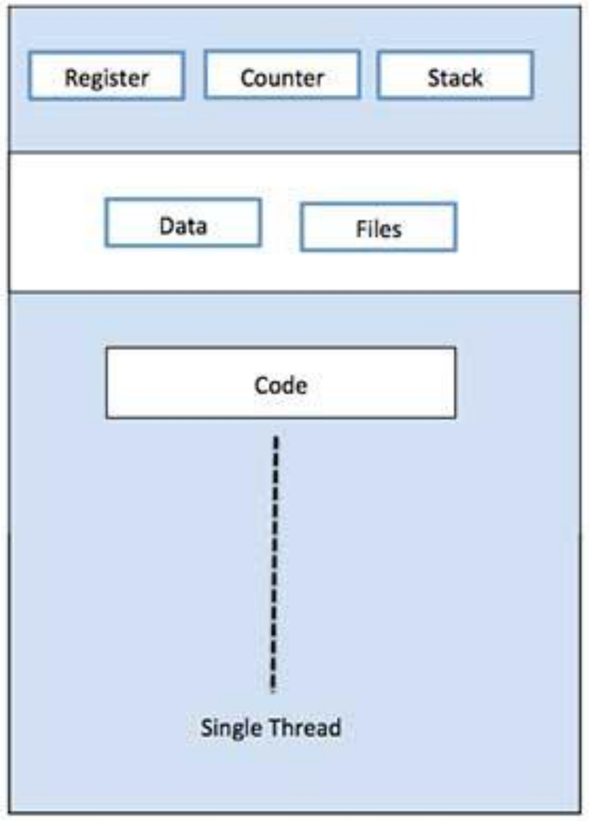
Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

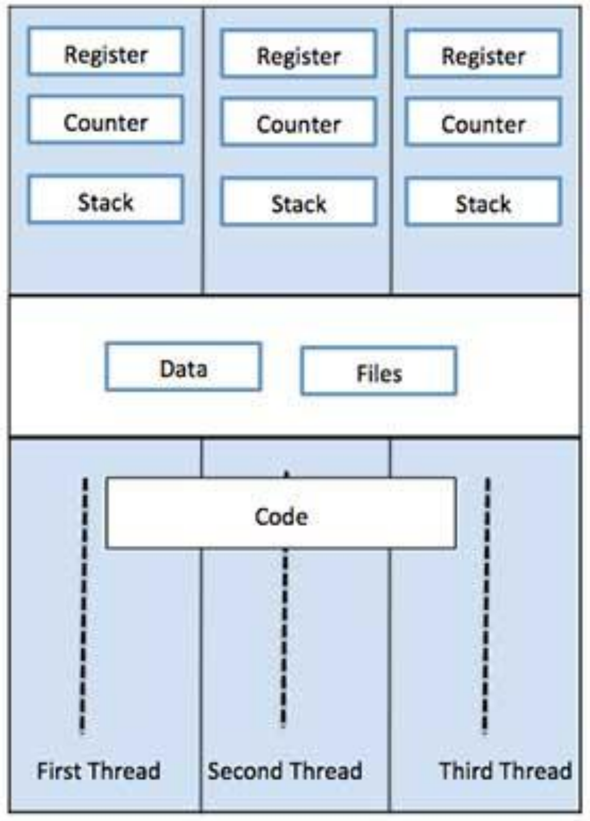


Thread

- Thread is basic unit of CPU utilization, it consist of thread id, program counter, stack , files etc..
- In multithreading one thread share with other thread belongs same process its code , data and files.



Single Process P with single thread

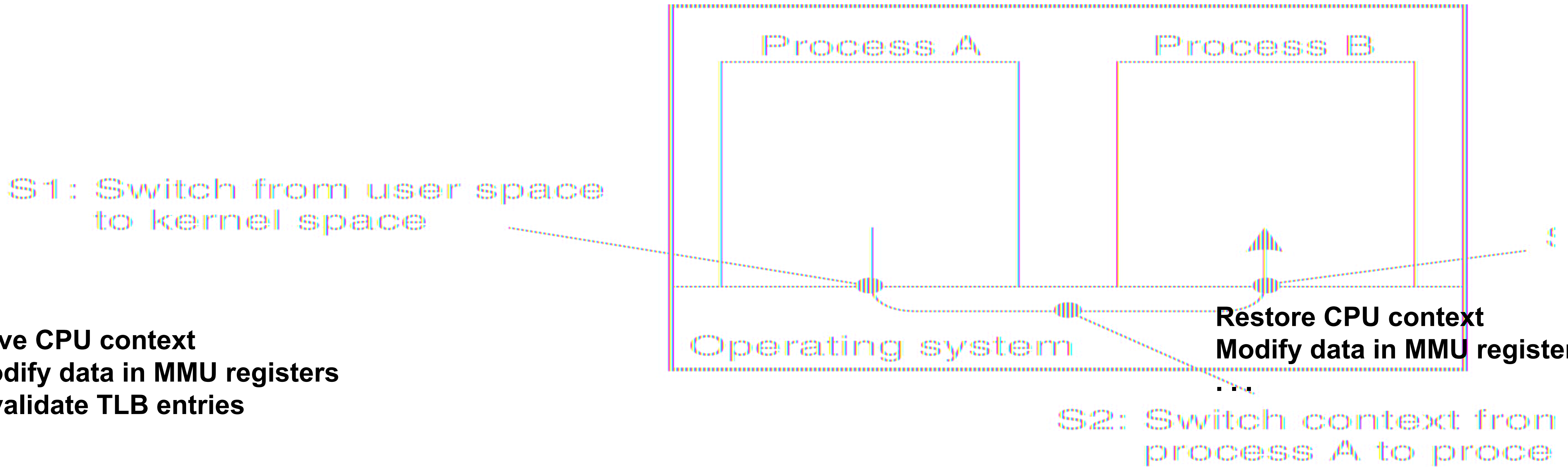


Single Process P with three threads

Benefits of Threads

- **Faster Context Switching**
- **Responsiveness** : e.g Browser
- **Utilization multiprocessors architecture :** the threads in a multi-threaded process can be scheduled to run in parallel on a multiprocessor or multicore processor.
- **Avoid process switching:** structure large applications not as a collection of processes, but through multiple threads.
- **Resource Sharing**

Overhead Due to Process Switching


Process A
Process B
S1 : Switch from user space to kernel space
Operating system
S2: Switch context from process A to proce

Save CPU context
Modify data in MMU registers
Invalidate TLB entries
. . .

Restore CPU context
Modify data in MMU registers
. . .

Figure : Context switching as the result of IPC.

Thread Implementation

- Kernel-level
  - Support multiprocessing
  - Independently schedulable by OS
  - Can continue to run if one thread blocks on a system call.
- User-level
  - Less overhead than k-level; faster execution
- Hybrid Threads
  - Combination of kernel and user level Support multiprocessing

User-level Threads

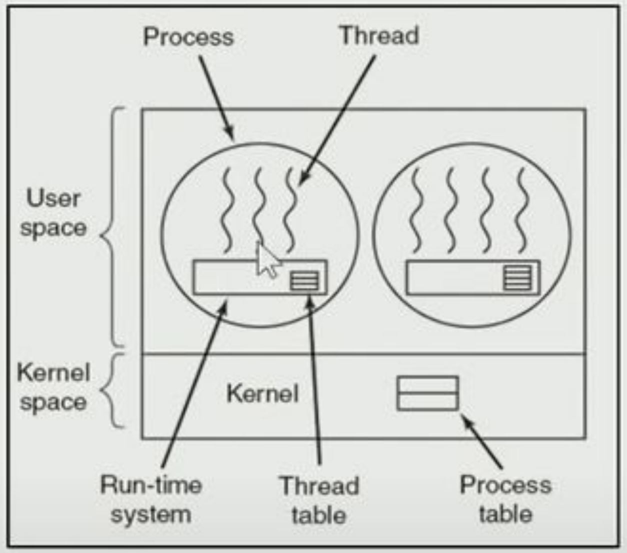
- User-level threads are created by calling functions in a user-level library. (eg Java)
- The process that uses user-level threads appears (to the OS) to be a single threaded process so there is no way to distribute the threads in a multiprocessor or block only part of the process.
- The advantage here is that they are even more efficient than kernel threads – no mode switches are involved in thread creation or switching.

User-level Threads

- Advantages:
  - Fast (really lightweight)

    (no system call to manage threads. The thread library does everything).
  - Can be implemented in an OS that does not support threading.
  - Switching is fast. No switch from user to protected mode.

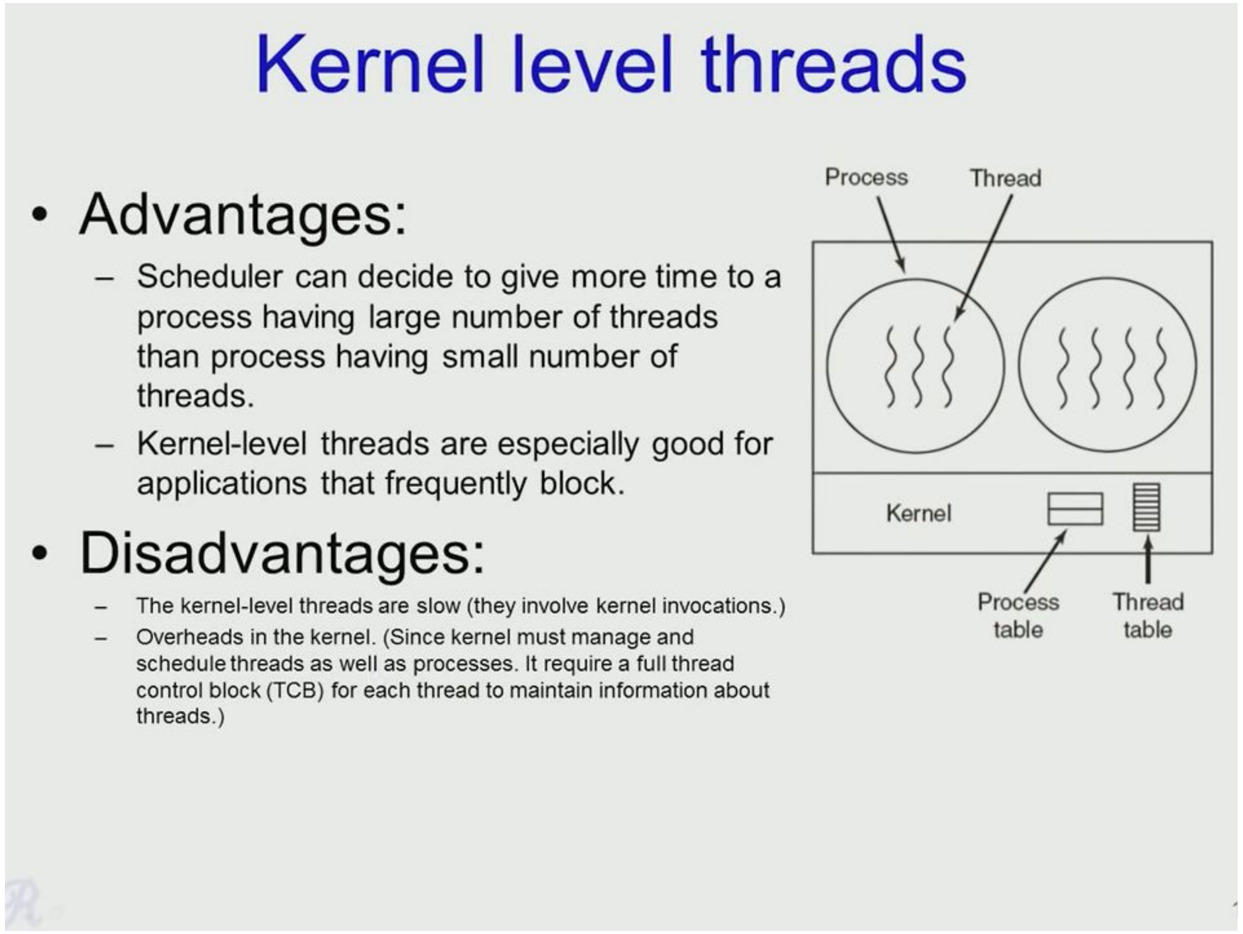
- Disadvantages:
  - Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
  - Lack of coordination between kernel and threads. (A process with 100 threads competes for a timeslice with a process having just 1 thread.)
  - Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)



Kernel-level Threads

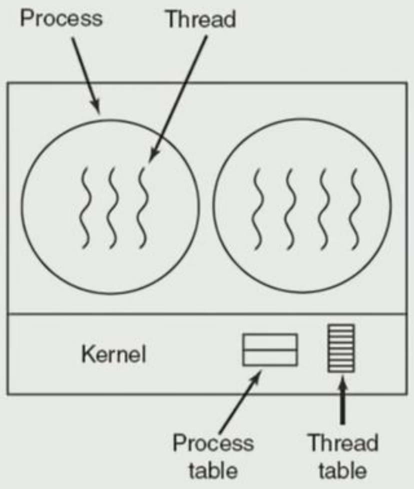
- The kernel is aware of the threads and schedules them independently as if they were processes.
- One thread may block for I/O, or some other event, while other threads in the process continue to run.
- A kernel-level thread in a user process does **not** have kernel privileges.

Kernel-level Threads



Kernel level threads

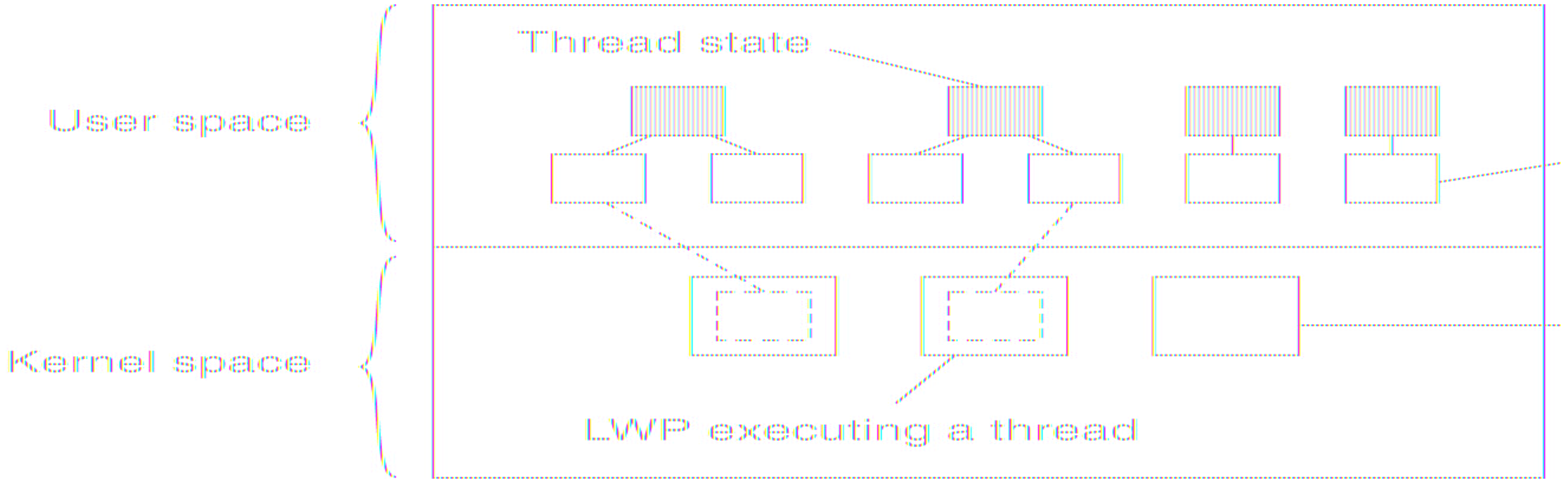
- Advantages:
  - Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.
- Disadvantages:
  - The kernel-level threads are slow (they involve kernel invocations.)
  - Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.)



Hybrid Threads –Lightweight Processes (LWP)

- LWP is similar to a kernel-level thread:
  - It runs in the context of a regular process
  - The process can have several LWPs created by the kernel in response to a system call.
- User level threads are created by calls to the user-level thread package.
- The thread package also has a scheduling algorithm for threads, runnable by LWPs.

Thread Implementation



Combining kernel-level lightweight processes and user-level threads.

Hybrid threads – LWP

- The OS schedules an LWP which uses the thread scheduler to decide which thread to run.
- Thread synchronization and context switching are done at the user level; LWP is not involved and continues to run.
- If a thread makes a blocking system call control passes to the OS (mode switch)
  - The OS can schedule another LWP or let the existing LWP continue to execute, in which case it will look for another thread to run.

Hybrid threads – LWP

- Solaris is a system that uses a variation of the hybrid approach.
  - Solaris implementation has evolved somewhat over time
- Processes have LWP and user-level threads (1-1 correspondence).
- The LWP is bound to a kernel level thread which is schedulable as an independent entity.
- Separates the process of thread creation from thread scheduling.

Hybrid threads – LWP

Advantages of the hybrid approach

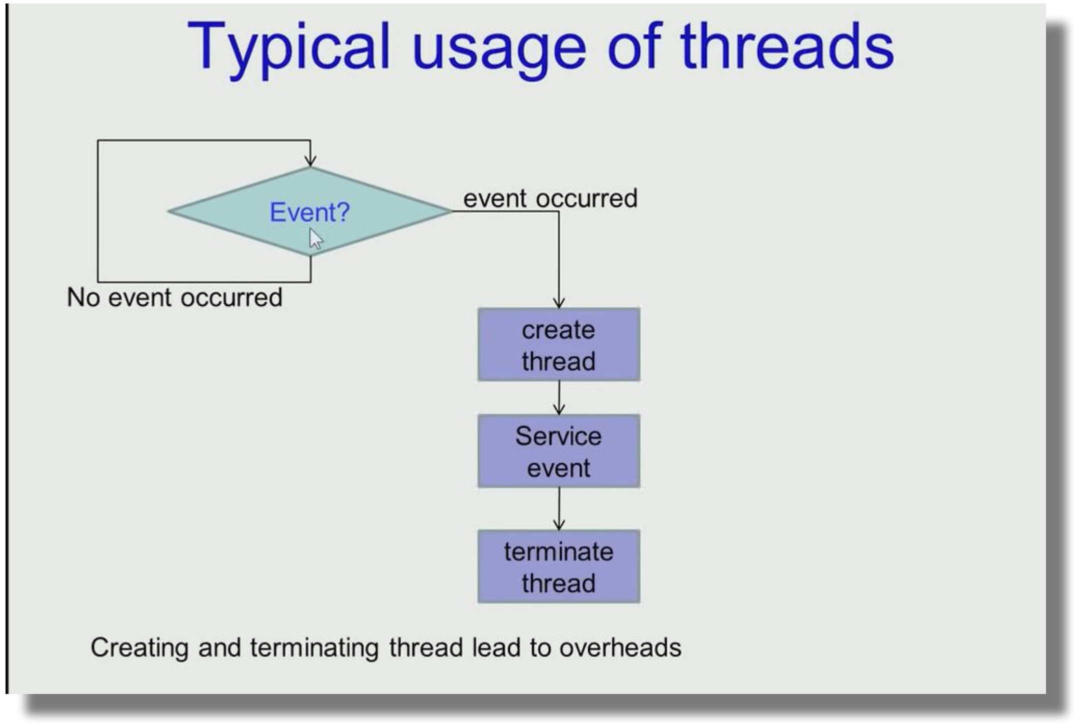
- Most thread operations (create, destroy, synchronize) are done at the user level
- Blocking system calls need not block the whole process
- Applications only deal with user-level threads
- LWPs can be scheduled in parallel on the separate processing elements of a multiprocessor.

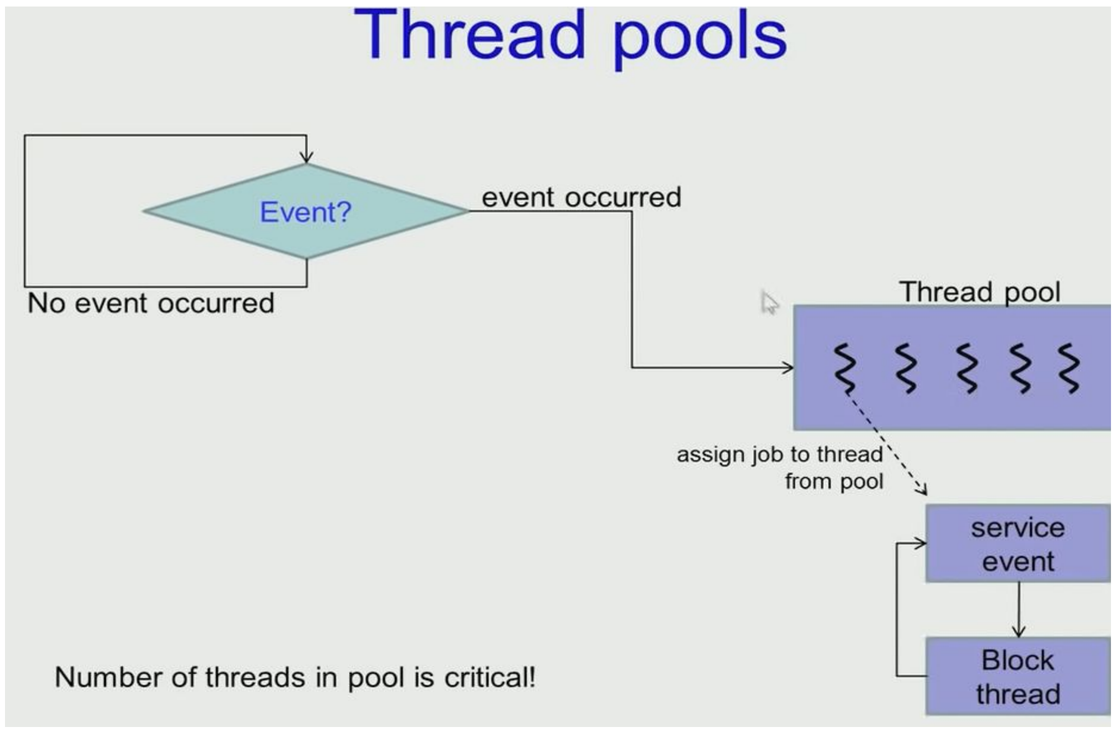
Threads in Distributed Systems

- Threads gain much of their power by sharing an address space

- However, multithreading can be used to improve the performance of individual nodes in a distributed system.
  - A process, running on a single machine; *e.g.*, a client or a server, can be multithreaded to improve performance

Threads in Distributed Systems





Threads in Distributed Systems

Types of threads in Distributed Systems

1. Multithreaded Clients

2. Multithreaded Server

Multithreaded Clients

Multithreaded web client

- Hiding network latencies:
  - Web browser scans an incoming HTML page, and finds that more files need to be fetched.
  - Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
  - As files come in, the browser displays them
- Multiple request-response calls to other machines (RPC):
  - A client does several calls at the same time, each one by a different thread.
  - It then waits until all results have been returned.
- Hide latency by starting several threads
  - One to download text (display as it arrives)
  - Others to download photographs, figures, etc.

Multithreaded Clients

- Even better: if servers are replicated, the multiple threads may be sent to separate sites.
- Result: data can be downloaded in several parallel streams, improving performance even more.
- Designate a thread in the client to handle and display each incoming data stream.

Multithreaded Servers

- Improve performance, provide better structuring
  - Consider what a file server does:
    - Wait for a request
    - Execute request (may require blocking I/O)
    - Send reply to client
- Several models for programming the server
  - Single threaded
  - Multi-threaded
  - Finite-state machine

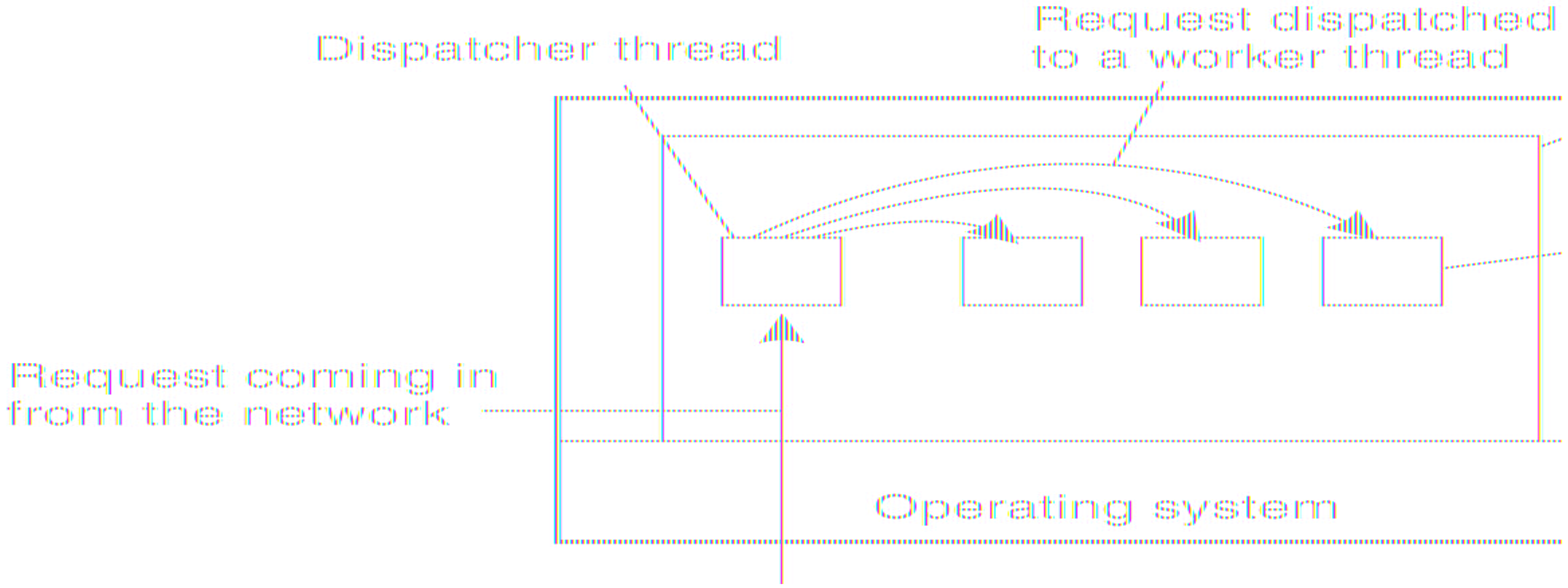
Multithreaded Servers

Single threaded Server

- A single-threaded (iterative) server processes one request at a time – other requests must wait.
- Possible solution: create (fork) a new server process for a new request.
- This approach creates performance problems
- Creating a new server thread is much more efficient.
- Processing is overlapped and shared data structures can be accessed without extra context switches.

Multithreaded Servers

Multithreaded Servers


Dispatcher thread
Request dispatched to a worker thread
Request coming in from the network
Operating system

A multithreaded server organized in a dispatcher/worker model.

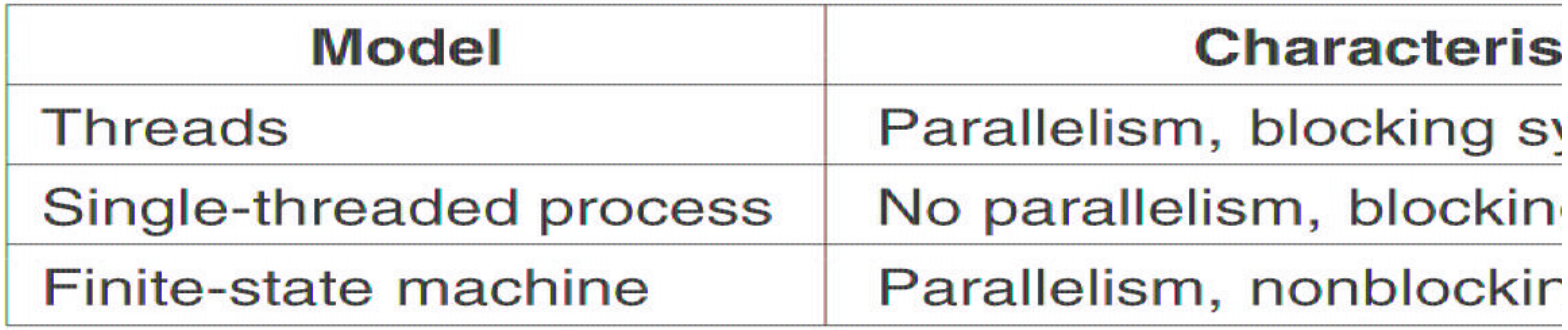
Multithreaded Servers

Finite-state machine

- The file server is single threaded but block for I/O operations
- Instead, save state of current request, switch to a new task – client request or disk reply.
- Outline of operation:
  - Get request, process until blocking I/O is needed
  - Save state of current request, start I/O, get next task
  - If task = completed I/O, resume process waiting on that I/O using saved state, else service a new request if there is one.

Multithreaded Servers

Three ways to construct a server.



Model	Characteris
Threads	Parallelism, blocking sy
Single-threaded process	No parallelism, blockin
Finite-state machine	Parallelism, nonblockir

Virtualization

- Virtual machine technology creates separate virtual machines, capable of supporting multiple instances of different operating systems.
- Use software to make it look like concurrent processes are executing simultaneously (eg **Vmware)**

Benefits

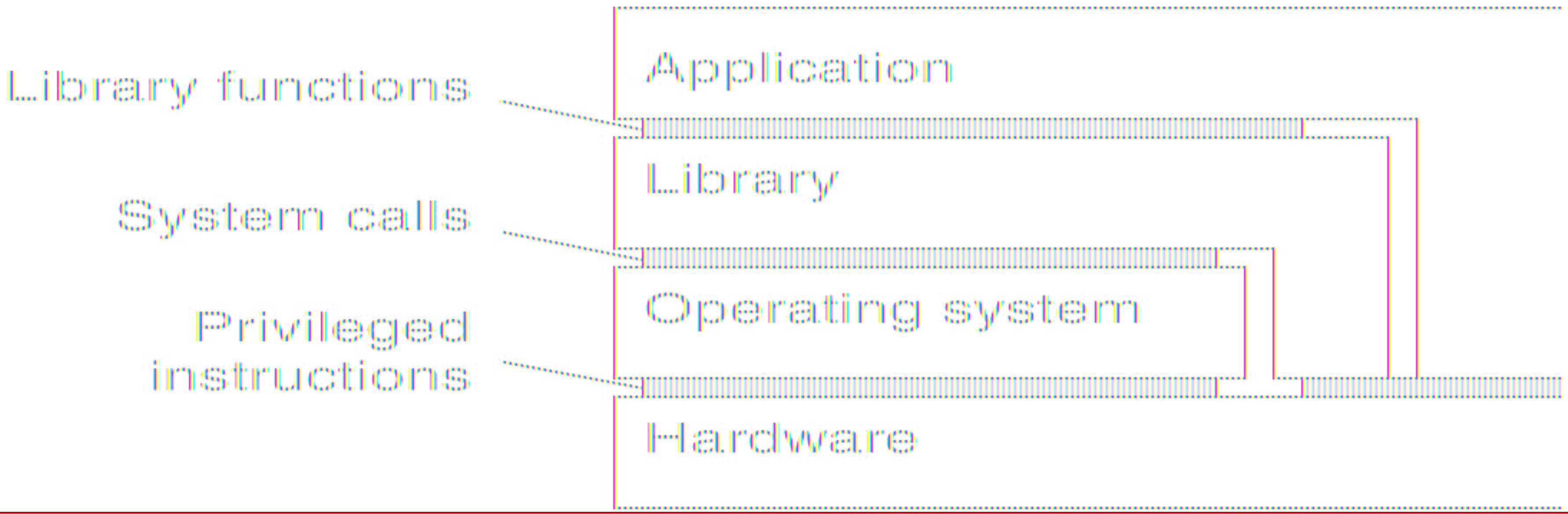
- Hardware changes faster than software
  - Suppose you want to run an existing application and the OS that supports it on a new computer: the VMM layer makes it possible to do so.
- Compromised systems (internal failure or external attack) are isolated.
- Run multiple different operating systems at the same time

Role of Virtualization in Distributed Systems

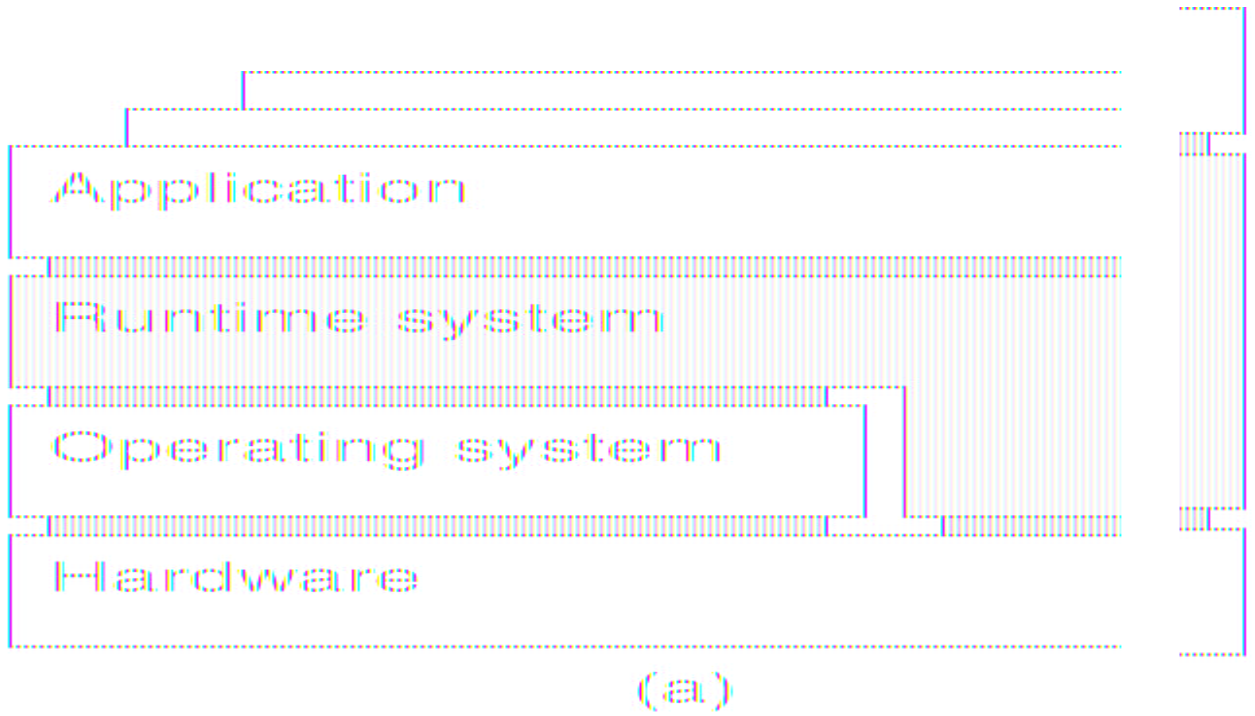
- Portability of virtual machines supports moving (or copying) servers to new computers
- Multiple servers can safely share a single computer
- Portability and security (isolation) are the critical characteristics.

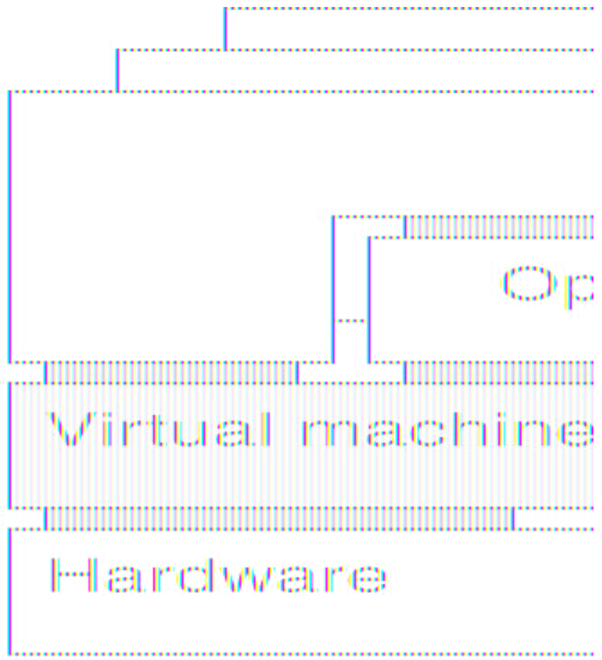
Interfaces Offered by Computer Systems

- Unprivileged machine instructions: available to any program
- Privileged instructions: hardware interface for the OS/other privileged software
- System calls: interface to the operating system for applications & library functions
- API: An OS interface through library function calls from applications.



Two Ways to Virtualize*





<u>Process Virtual Machine</u>: program is compiled to intermediate code, executed by a runtime system (eg JVM, .NET framework)

<u>Virtual Machine Monitor:</u> software layer mimics the instruction set; supports an OS and its applications

Migration models

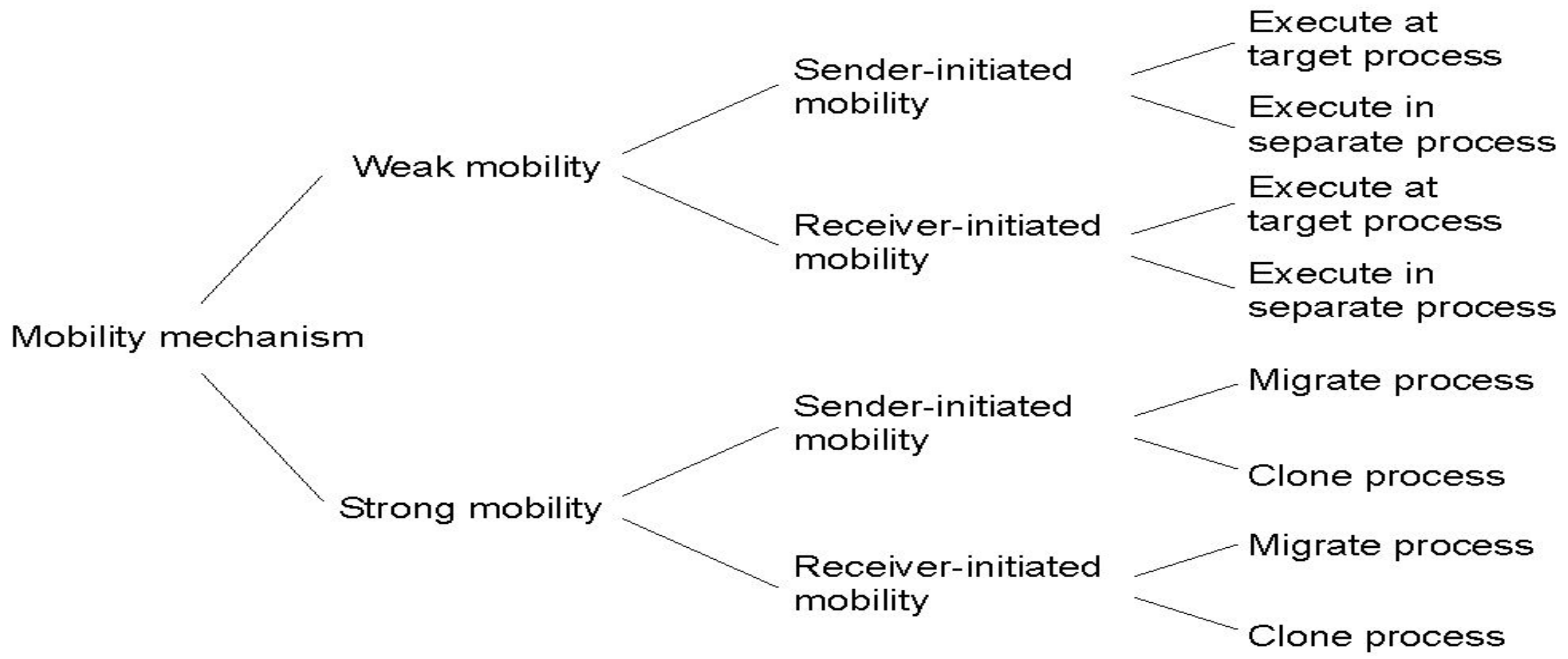
- Process = code seg + resource seg + execution seg
- Weak versus strong mobility
  - Weak => only code transferred
  - Strong => Code and execute segment can transferred
- Sender-initiated versus receiver-initiated
  - Sender-initiated (code is with sender)
    - Client sending a query to database server
    - Client should be pre-registered
  - Receiver-initiated
    - Downloading code from server by a client
    - Java applets

Code Segment

Resource Segment

Execution Segment

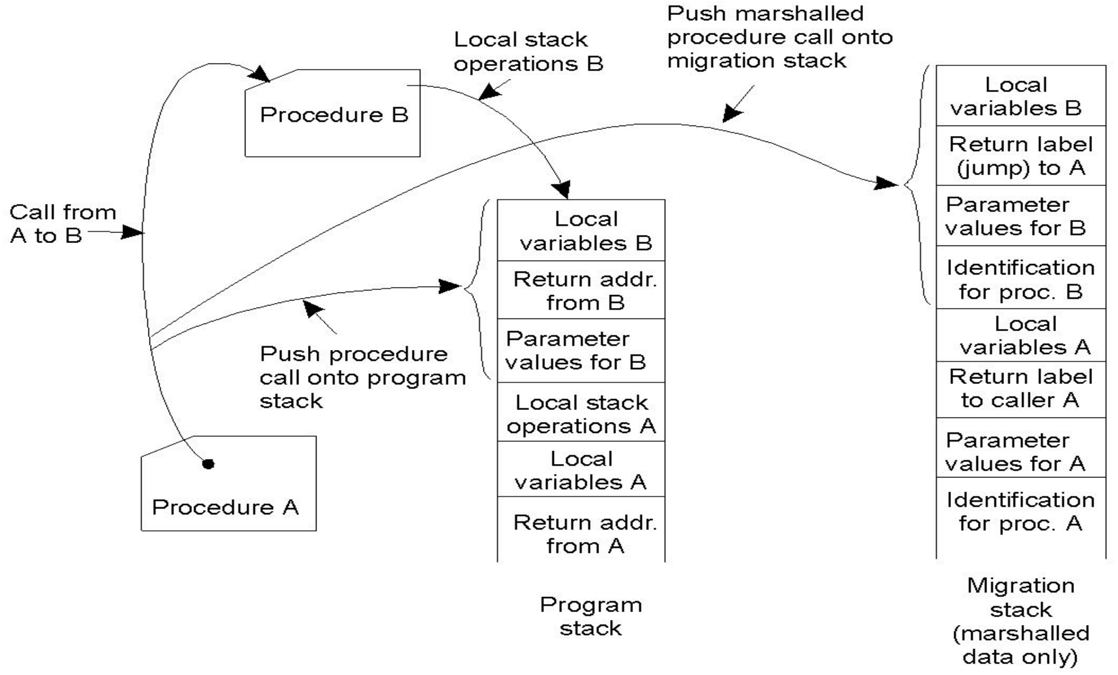
Models for Code Migration


Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

Migration in Heterogeneous Systems

Systems can be heterogeneous (different architecture, OS)

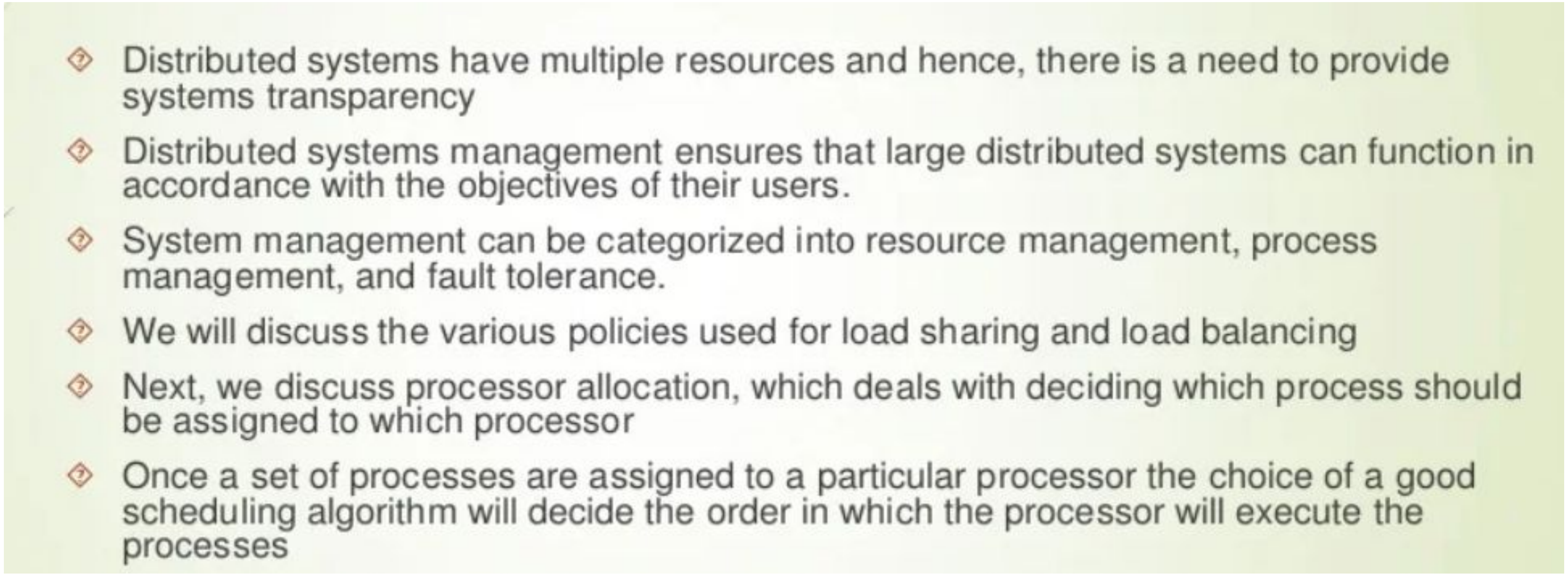
- Weak mobility: recompile code, no run time information
- Strong mobility: recompile code segment, transfer execution segment [migration stack]



Resource and Process Management

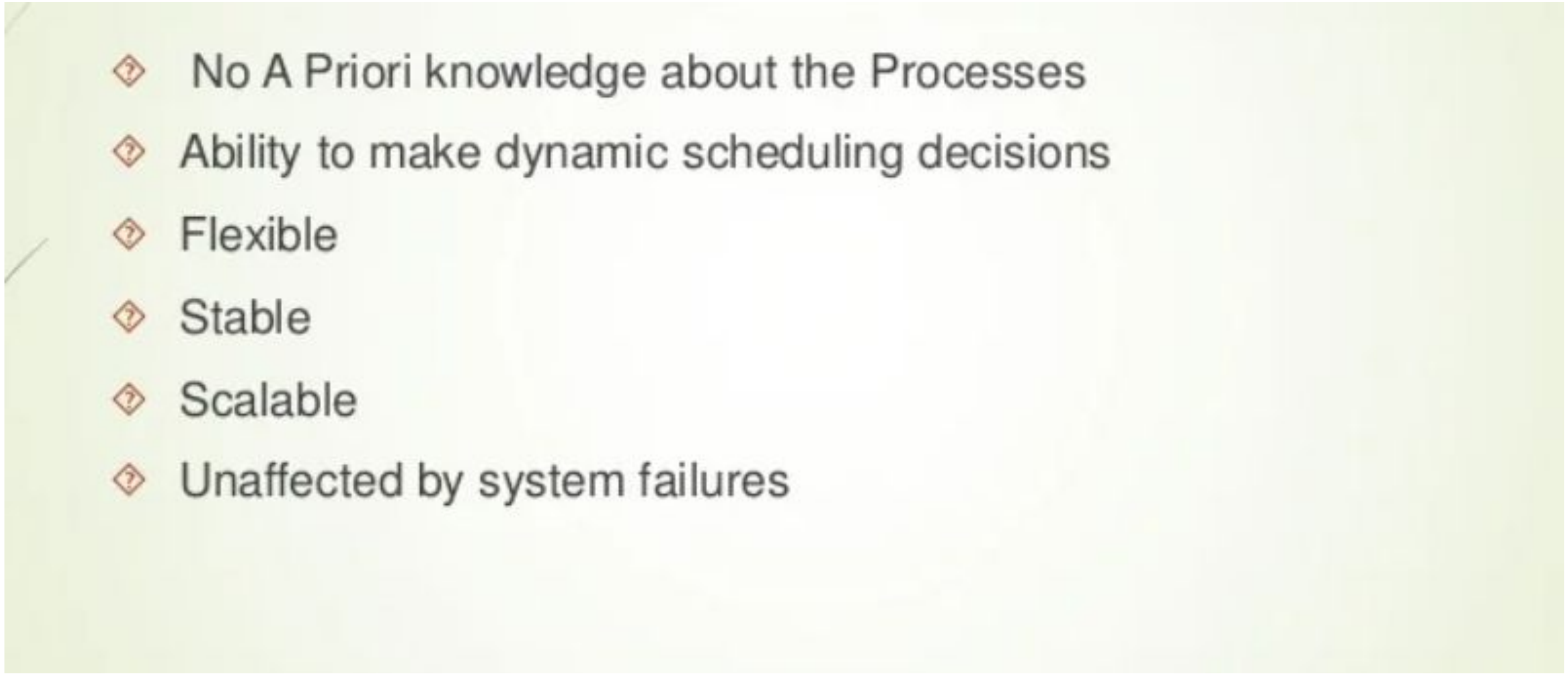
4.1 Desirable Features of global Scheduling algorithm,
Task assignment approach,
Load balancing approach,
Load sharing approach

Introduction



- Distributed systems have multiple resources and hence, there is a need to provide systems transparency
- Distributed systems management ensures that large distributed systems can function in accordance with the objectives of their users.
- System management can be categorized into resource management, process management, and fault tolerance.
- We will discuss the various policies used for load sharing and load balancing
- Next, we discuss processor allocation, which deals with deciding which process should be assigned to which processor
- Once a set of processes are assigned to a particular processor the choice of a good scheduling algorithm will decide the order in which the processor will execute the processes

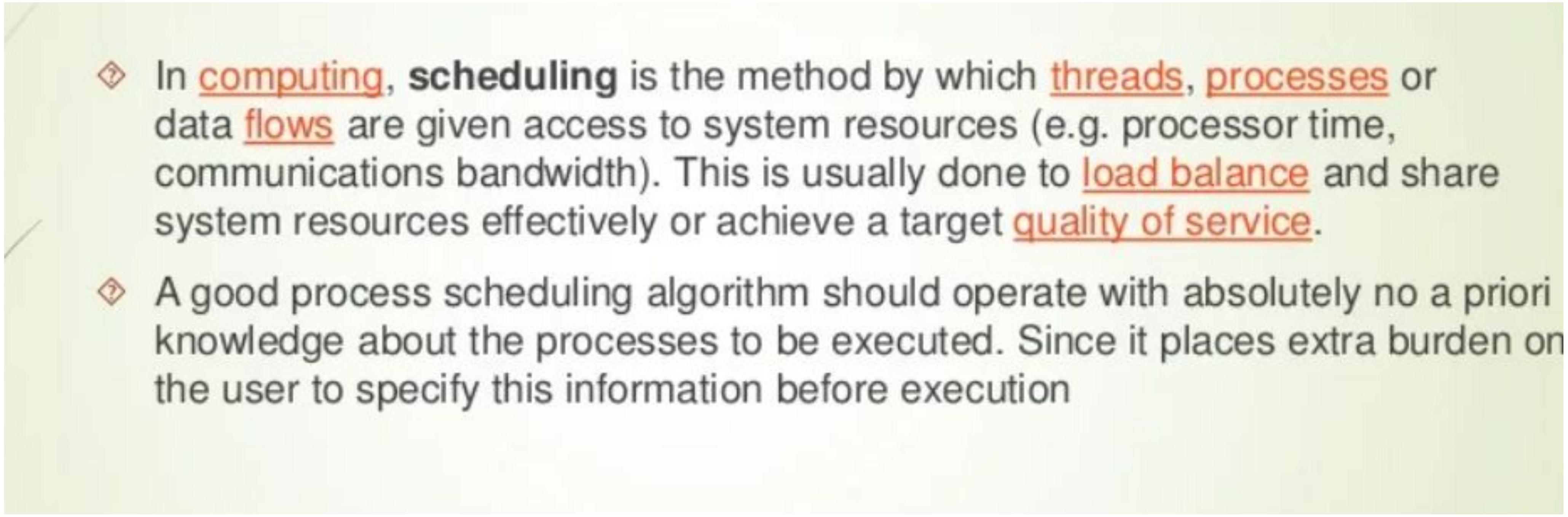
Desirable Features of global Scheduling algorithm



- No A Priori knowledge about the Processes
- Ability to make dynamic scheduling decisions
- Flexible
- Stable
- Scalable
- Unaffected by system failures

Desirable Features of global Scheduling algorithm

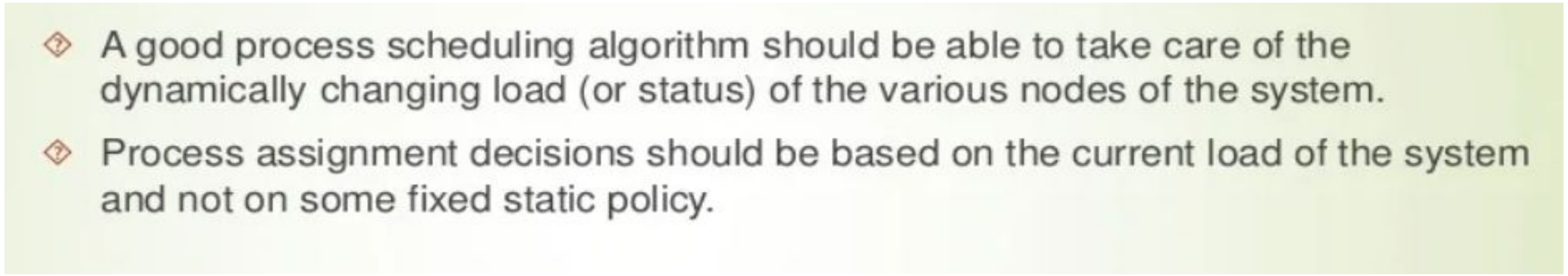
1. No Prior knowledge about the processes



- In computing, **scheduling** is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance and share system resources effectively or achieve a target quality of service.
- A good process scheduling algorithm should operate with absolutely no a priori knowledge about the processes to be executed. Since it places extra burden on the user to specify this information before execution

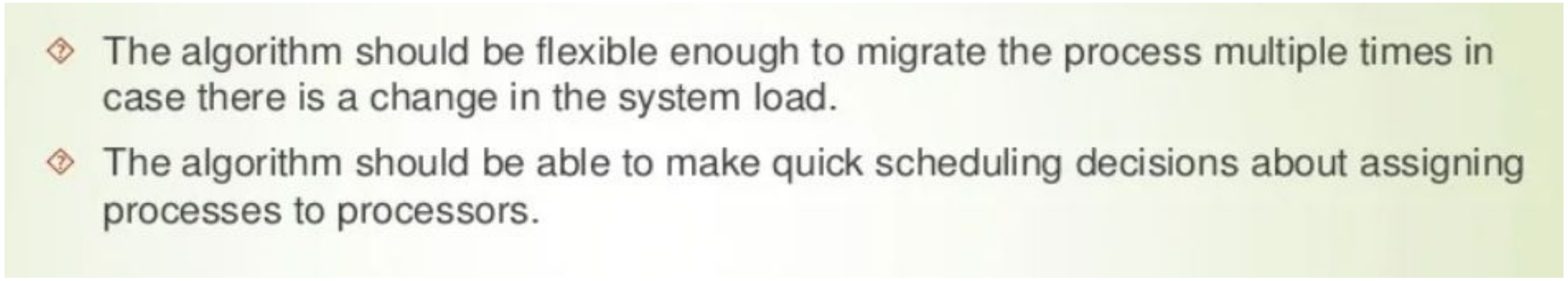
Desirable Features of global Scheduling algorithm

2. Ability to dynamic decision



- A good process scheduling algorithm should be able to take care of the dynamically changing load (or status) of the various nodes of the system.
- Process assignment decisions should be based on the current load of the system and not on some fixed static policy.

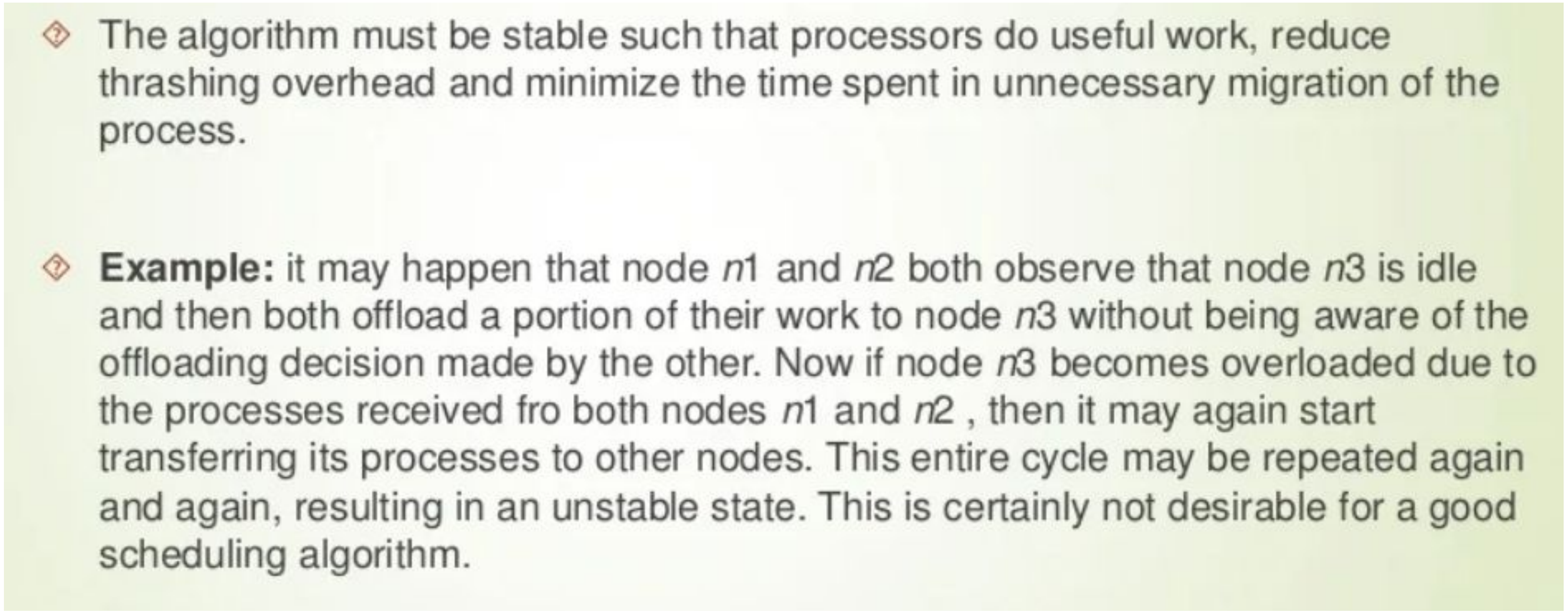
3. Flexible



- The algorithm should be flexible enough to migrate the process multiple times in case there is a change in the system load.
- The algorithm should be able to make quick scheduling decisions about assigning processes to processors.

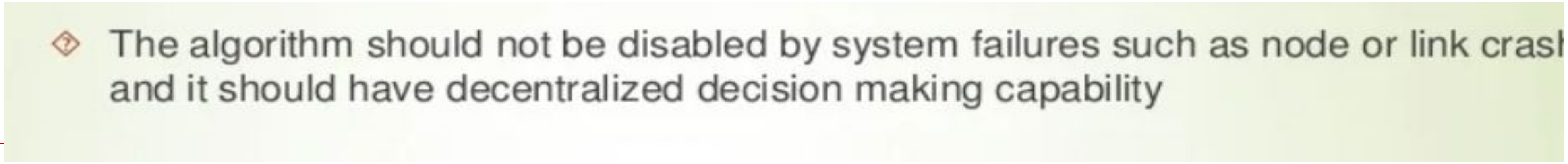
Desirable Features of global Scheduling algorithm

4. Stable



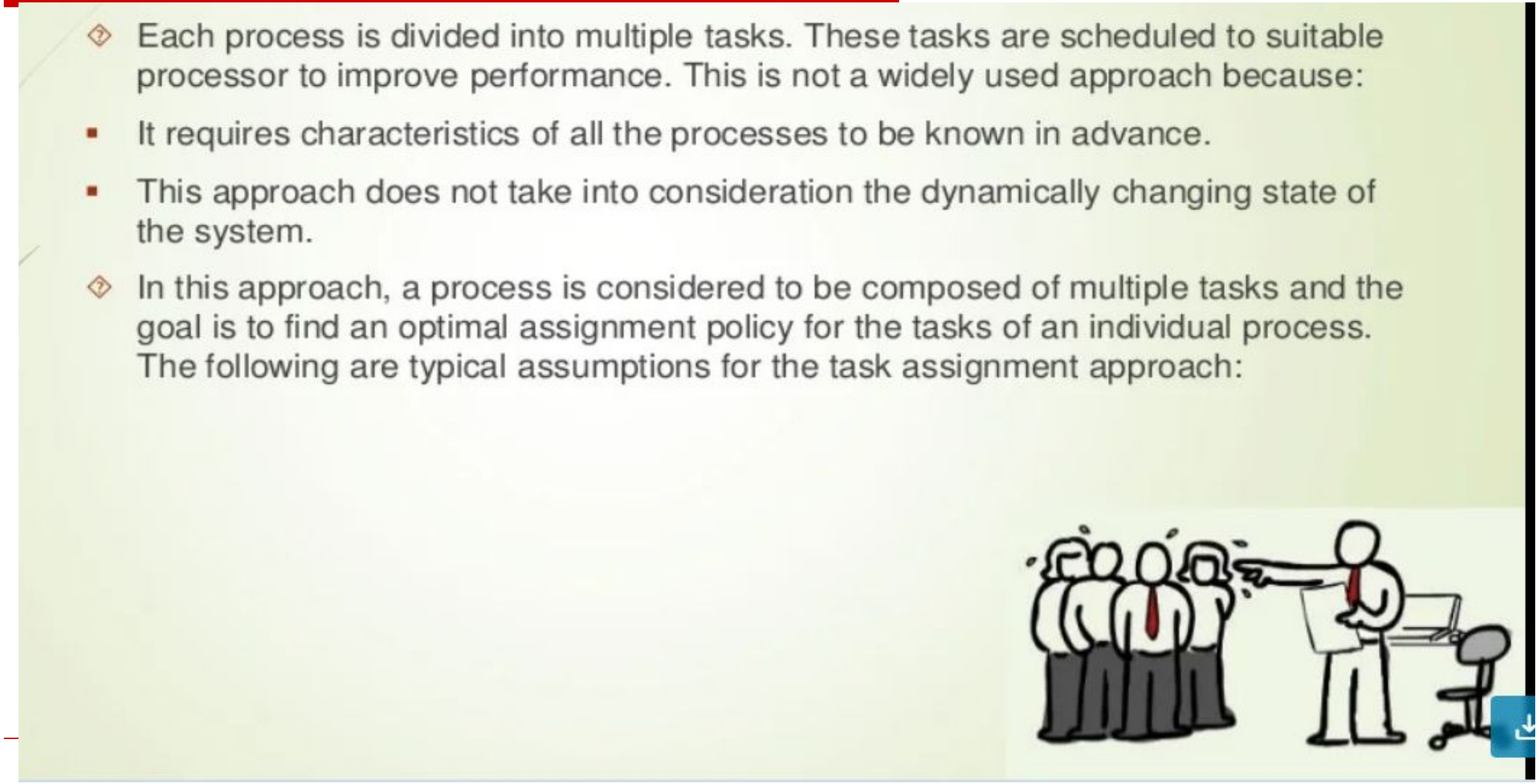
- The algorithm must be stable such that processors do useful work, reduce thrashing overhead and minimize the time spent in unnecessary migration of the process.
- **Example:** it may happen that node *n*1 and *n*2 both observe that node *n*3 is idle and then both offload a portion of their work to node *n*3 without being aware of the offloading decision made by the other. Now if node *n*3 becomes overloaded due to the processes received fro both nodes *n*1 and *n*2 , then it may again start transferring its processes to other nodes. This entire cycle may be repeated again and again, resulting in an unstable state. This is certainly not desirable for a good scheduling algorithm.

5. Unaffected by system failure



- The algorithm should not be disabled by system failures such as node or link cras and it should have decentralized decision making capability

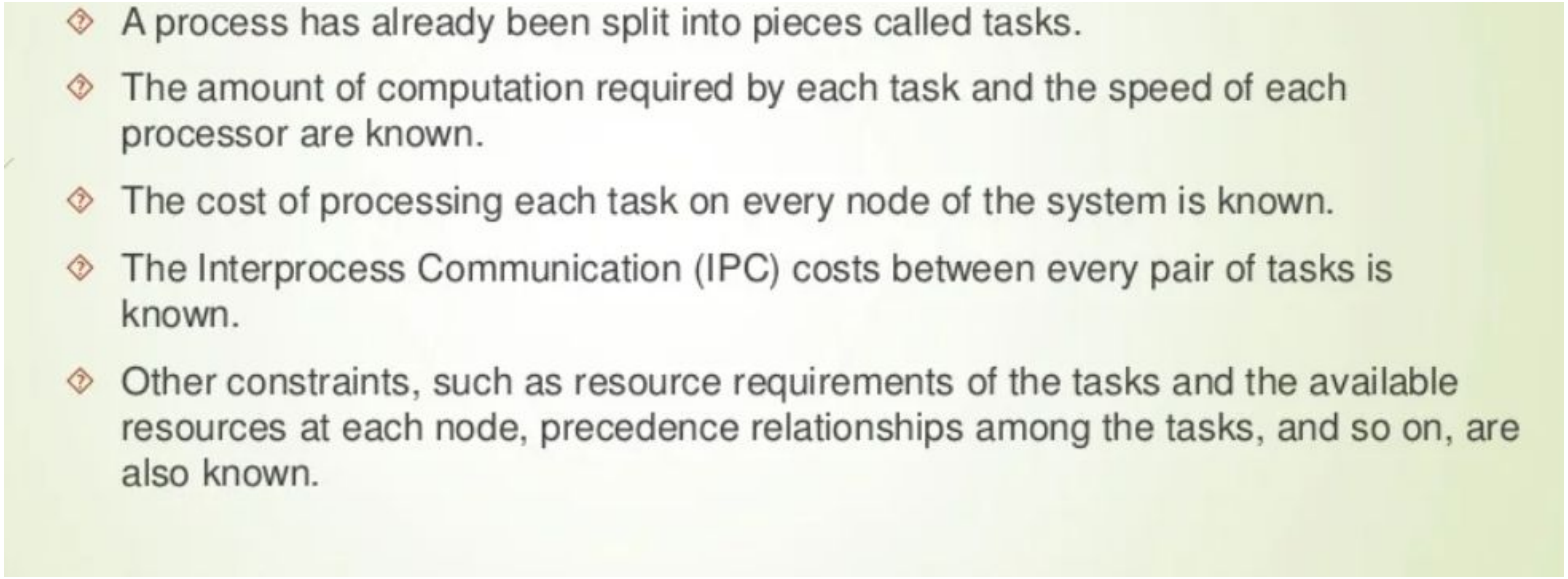
Task Assignment Approach



- Each process is divided into multiple tasks. These tasks are scheduled to suitable processor to improve performance. This is not a widely used approach because:
  - It requires characteristics of all the processes to be known in advance.
  - This approach does not take into consideration the dynamically changing state of the system.
- In this approach, a process is considered to be composed of multiple tasks and the goal is to find an optimal assignment policy for the tasks of an individual process. The following are typical assumptions for the task assignment approach:

Task Assignment Approach

Assumptions Task Assignment Approach



- A process has already been split into pieces called tasks.
- The amount of computation required by each task and the speed of each processor are known.
- The cost of processing each task on every node of the system is known.
- The Interprocess Communication (IPC) costs between every pair of tasks is known.
- Other constraints, such as resource requirements of the tasks and the available resources at each node, precedence relationships among the tasks, and so on, are also known.

Task Assignment Approach

- Goal is to assign the tasks of a process to the nodes of a distributed system in such a manner as to achieve goals such as the following goals:
  - Minimization of IPC costs
  - Quick turnaround time for the complete process
  - A high degree of parallelism
  - Efficient utilization of system resources in general

Task Assignment Approach

- These goals often conflict. E.g., while minimizing IPC costs tends to assign all tasks of a process to a single node, efficient utilization of system resources tries to distribute the tasks evenly among the nodes. So also, quick turnaround time and a high degree of parallelism encourage parallel execution of the tasks, the precedence relationship among the tasks limits their parallel execution.

- Also note that in case of m tasks and q nodes, there are m^q possible assignments of tasks to nodes . In practice, however, the actual number of possible assignments of tasks to nodes may be less than m^q due to the restriction that certain tasks cannot be assigned to certain nodes due to their specific requirements (e.g. need a certain amount of memory or a certain data file).

Task Assignment Approach

Task assignment example

- There are two nodes, {n1, n2} and six tasks {t1, t2, t3, t4, t5, t6}. There are two task assignment parameters – the task execution cost (x_{ab} the cost of executing task a on node b) and the inter-task communication cost (c_{ij} the inter-task communication cost between tasks *i* and *j*).

<u>Inter-task communication cost</u>

	t1	t2	t3	t4	t5	t6
t1	0	6	4	0	0	12
t2	6	0	8	12	3	0
t3	4	8	0	0	11	0
t4	0	12	0	0	5	0
t5	0	3	11	5	0	0
t6	12	0	0	0	0	0

<u>Execution costs</u>

<u>Nodes</u>

	n1	n2
t1	5	10
t2	2	∞
t3	4	4
t4	6	3
t5	5	2
t6	∞	4

Task t6 cannot be executed on node n1 and task t2 cannot be executed on node n2 since the resources they need are not available on these nodes.

Task Assignment Approach

Task assignment example

1) **Serial assignment**, where tasks t1, t2, t3 are assigned to node n1 and tasks t4, t5, t6 are assigned to node n2:

Execution cost, $x = x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62} = 5 + 2 + 4 + 3 + 2 + 4 = 20$

Communication cost, $c = c_{14} + c_{15} + c_{16} + c_{24} + c_{25} + c_{26} + c_{34} + c_{35} + c_{36} = 0 + 0 + 12 + 12 + 3 + 0 + 0 + 11 + 0 = 38$.

Hence total cost = 58.

2) **Optimal assignment**, where tasks t1, t2, t3, t4, t5 are assigned to node n1 and task t6 is assigned to node n2.

Execution cost, $x = x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{62}$

$= 5 + 2 + 4 + 6 + 5 + 4 = 26$

Communication cost, $c = c_{16} + c_{26} + c_{36} + c_{46} + c_{56}$

$= 12 + 0 + 0 + 0 + 0 = 12$

Total cost = 38

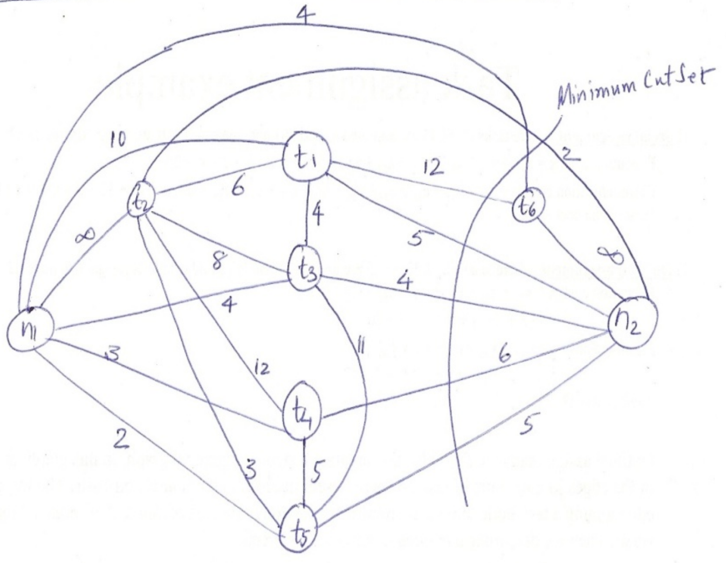
Task Assignment Approach

Assignment Graph

Optimal assignments are found by first creating a static assignment graph. In this graph, the weights of the edges joining pairs of task nodes represent inter-task communication costs. The weight on the edge joining a task node to node n1 represents *the execution cost of that task on node n2 and vice-versa*. Then we determine a *minimum cutset* in this graph.

A ***cutset*** is defined to be a set of edges such that when these edges are removed, the nodes of the graph are partitioned into two disjoint subsets such that nodes in one subset are reachable from n1 and the nodes in the other are reachable from n2. Each task node is reachable from either n1 or n2. The weight of a cutset is the sum of the weights of the edges in the cutset. This sums up the execution and communication costs for that assignment. An optimal assignment is found by finding a minimum cutset.

Assignment Graph with Minimum Cost Cut

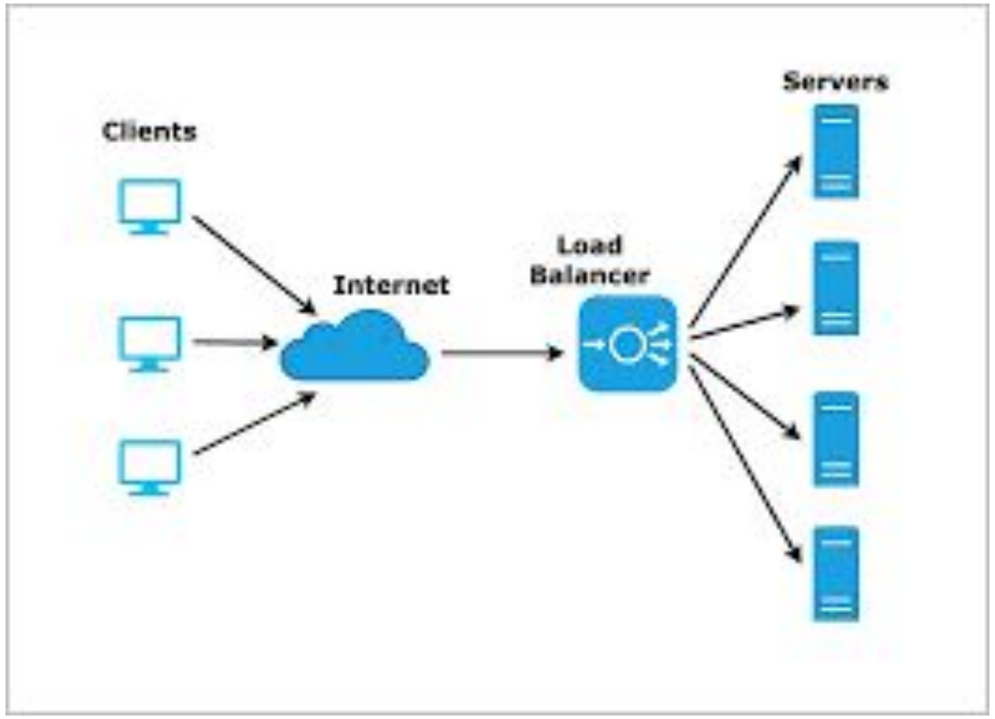


Min Cutset = 2 + 4 + 12 + 5 + 4 + 6 + 5
= 38

Load balancing Approach

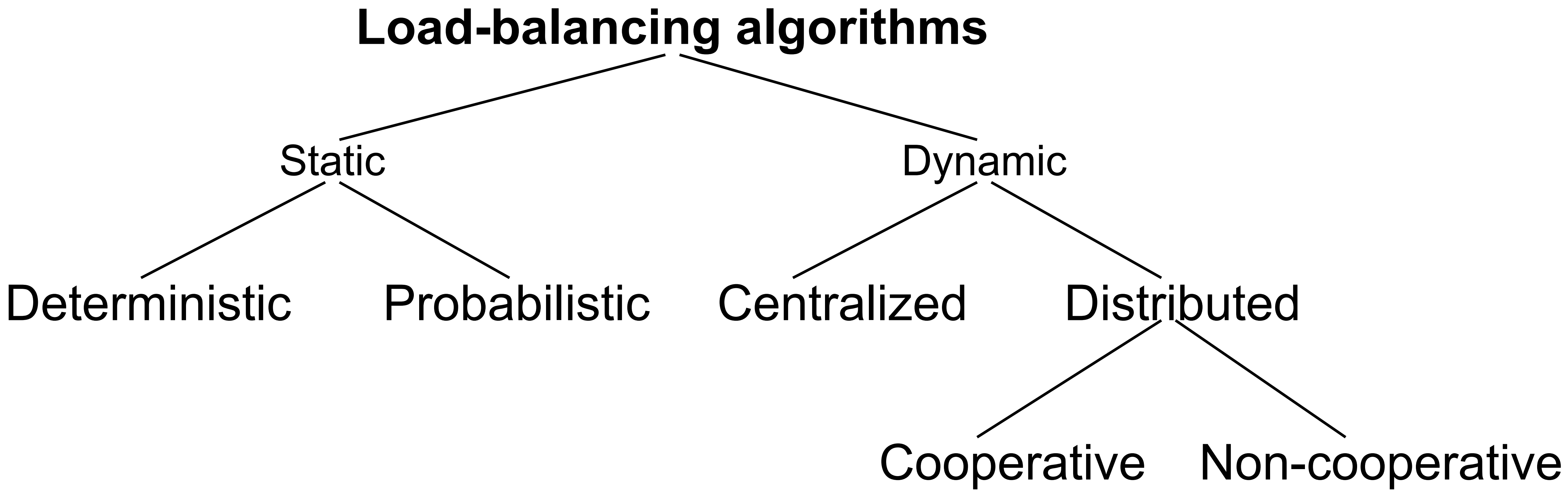
Load balancing

- Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.
- The main goal of load balancing is to maximize the use of resources, reduce the response time, and to reduce the overload of a single resource.
- Load balancing can improve the reliability and availability if multiple components are used in the process instead of a single one.



Load-balancing approach

Type of load-balancing algorithms



A Taxonomy of Load-balancing Algorithms

Load-balancing approach

Type of 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 approach

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 approach

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 approach

Type of distributed 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 <u>total number of processes</u> 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

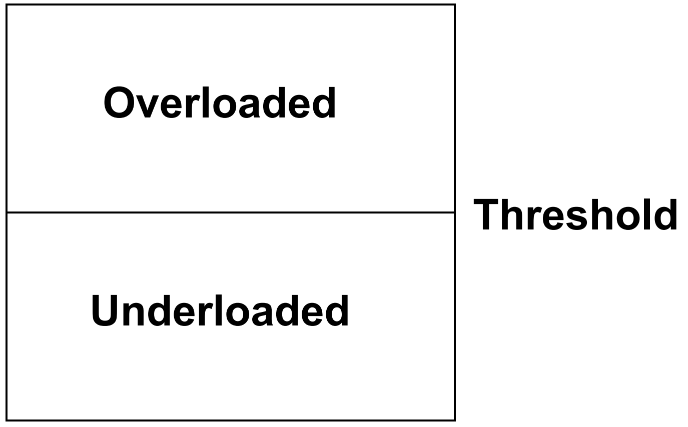
- None of the previous methods can be used in modern systems because of periodically running processes such as mail, OS manger, news daemons and so on exits permanently
- An acceptable method for use as the load estimation policy in these systems would be to measure the CPU utilization of the nodes
- 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

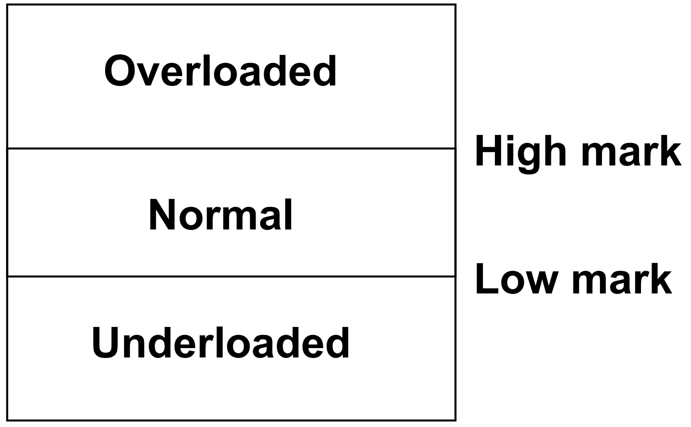
- Most of the algorithms use the *threshold policy* to decide on whether the node is lightly-loaded or heavily-loaded
- Threshold value is a limiting value of the workload of node which can be determined by
  - Static policy: predefined threshold value for each node depending on processing capability
  - 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



Single-threshold policy



Double-threshold policy

- To reduce instability double-threshold policy has been proposed which is also known as high-low 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

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

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 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 randomly chosen node 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 in load 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 has at least two processes

Priority assignment policy

for Load-balancing algorithms

- Selfish
  - Local processes are given higher priority than remote processes.
- Altruistic
  - Remote processes are given higher priority than local processes.
- 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

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 I.
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

SUMMARY

- Resource manager of a distributed system schedules the processes to optimize combination of resources usage, response time, network congestion, scheduling overhead
- Three different approaches has been discussed
    - Task assignment approach deals with the assignment of task in order to minimize inter process communication costs and improve turnaround time for the complete process, by taking some constraints into account
    - In load-balancing approach the process assignment decisions attempt to equalize the avarage workload on all the nodes of the system
    - In load-sharing approach the process assignment decisions attempt to keep all the nodes busy if there are sufficient processes in the system for all the nodes

References

- Pradeep K. Sinha , “Distributed Operating System” PHI Publication 20008.
- George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design", 4th Edition, Pearson Education, 2005.

Thank you !!!