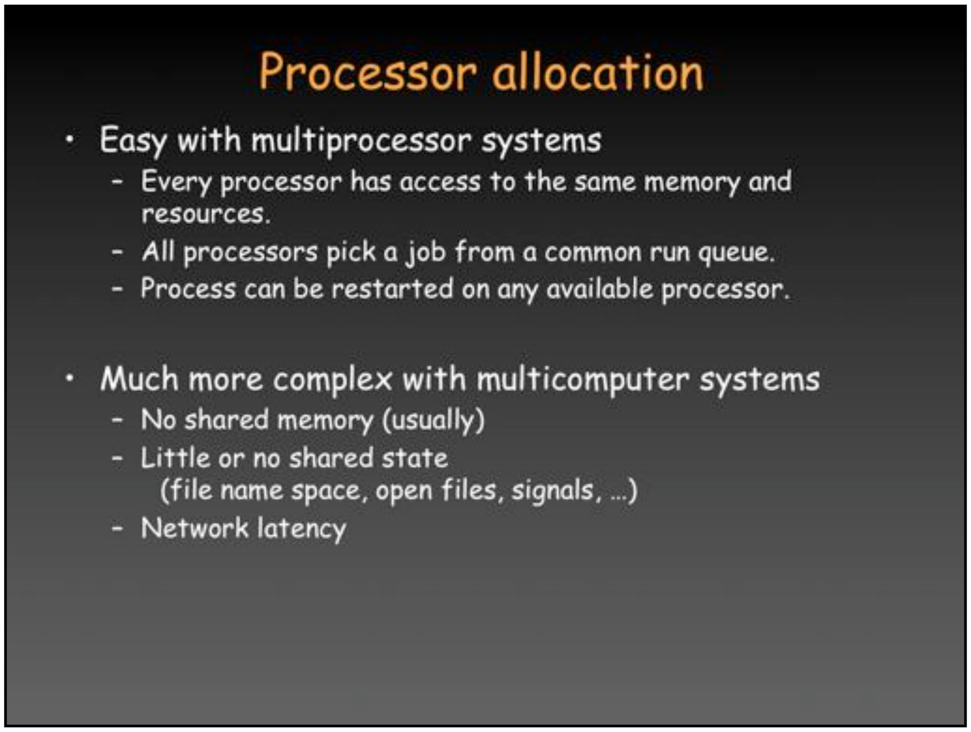
Processor Allocation and Migration

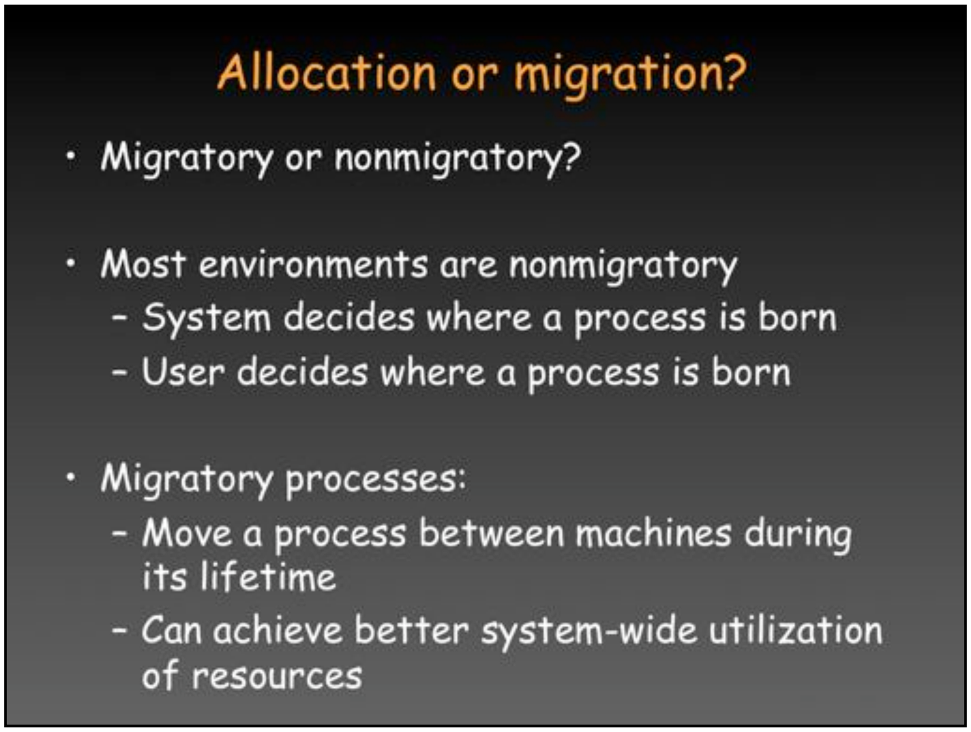


Processor allocation

- Easy with multiprocessor systems
  - Every processor has access to the same memory and resources.
  - All processors pick a job from a common run queue.
  - Process can be restarted on any available processor.
- Much more complex with multicomputer systems
  - No shared memory (usually)
  - Little or no shared state (file name space, open files, signals, …)
  - Network latency

Processor allocation was not a serious problem when we examined multiprocessor systems (shared memory). In those systems, all processors had access to the same image of the operating system and grabbed jobs from a common job queue. When a quantum expired or a process blocked, it could be restarted by any available processor.

In multicomputer systems, things get more complex. We may not be able to use shared memory segments or message queues to communicate with other processes. The file system may look different on different machines. The overhead of dispatching a process on another system may be high compared to the run time of the process.



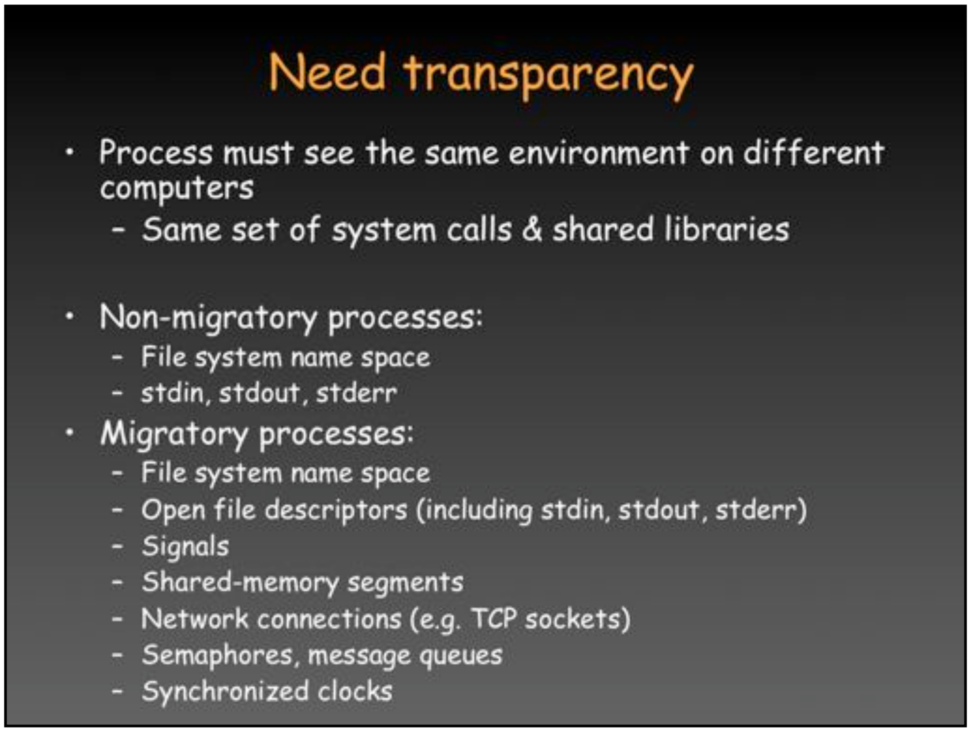
Allocation or migration?

- Migratory or nonmigratory?
- Most environments are nonmigratory
  - System decides where a process is born
  - User decides where a process is born
- Migratory processes:
  - Move a process between machines during its lifetime
  - Can achieve better system-wide utilization of resources

Most of today's environments have a nonmigratory model of processor allocation. A processor is chosen by the user (e.g. by the workstation being used or by an *rsh* command) or else the system makes an initial decision on a system on which the process will execute. Once it starts, it will continue running on that processor.

An alternative is to support **process migration**, where processes can move dynamically during their lifetime. The hope in such a system is that it will allow for better system-wide utilization of resources (e.g. as one computer becomes too heavily loaded, some of the processes can migrate to a less loaded system).

When we discuss implementing processor allocation, we are talking about one of two types of processes: **nonmigratory** processes remain on the processor on which they were created (the decision is *where* to create them); **migratory** processes can be moved after creation, which allows for better load balancing but is more complex.

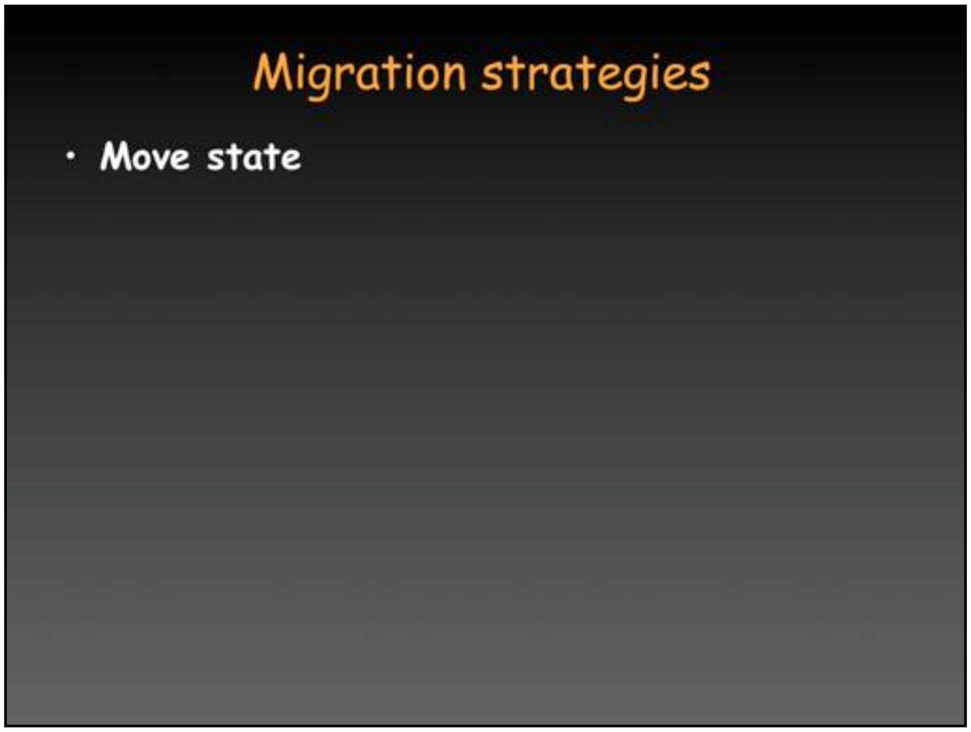


Need transparency

- Process must see the same environment on different computers
  - Same set of system calls & shared libraries
- Non-migratory processes:
  - File system name space
  - stdin, stdout, stderr
- Migratory processes:
  - File system name space
  - Open file descriptors (including stdin, stdout, stderr)
  - Signals
  - Shared-memory segments
  - Network connections (e.g. TCP sockets)
  - Semaphores, message queues
  - Synchronized clocks

If we are to run a process on an *arbitrary* system, it is important that all systems present the same execution environment. Certainly system binaries must be capable of executing on a different machine (unless we use interpreted pseudocode such as Java). Processes typically do not run in a vacuum but read input and write output. Even if a process will never migrate to another machine during execution it should have predictable access to a file system name space (it would be hard to debug a program that opens a different file or fails to open a file depending on what system it was assigned to). To accomplish this, any of the files that a program will read/write should be on a distributed file system that is set up to provide a uniform name space across all participating machines. Moreover, the process may have to forward operations on the standard input and standard output file descriptors to the originating machine. This may be done during the creation of those file descriptors on the remote machine using a mechanism such as sockets (this is what *rsh* does).

With migratory processes, things get more complicated. If a running process is to continue execution on a different system, any existing descriptors to open files must continue to operate on those files (this includes stdin, stdout, stderr as well as other files). If a process expects to catch signals, the signal mask for the process should be migrated. If there are any pending signals for the process, they also must be migrated. Shared memory should continue to work if it was in use (this will most likely necessitate a DSM system). Any existing network connections should also continue to be active. Since a process may rely on a service such as system time (to time latencies, for example), clocks should be synchronized.

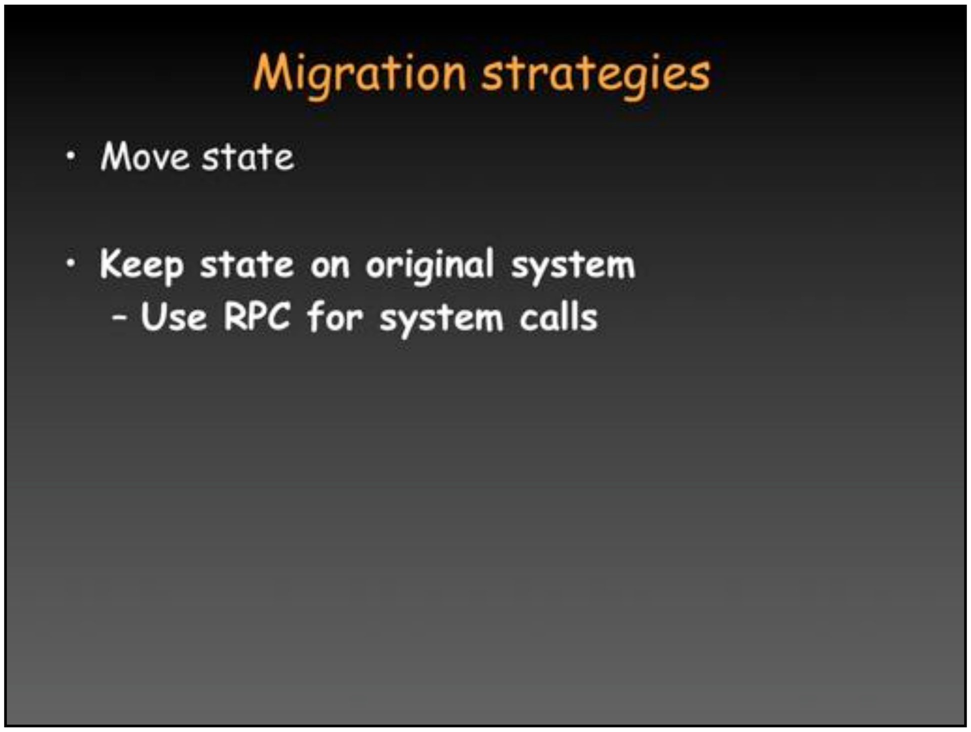


Migration strategies

- Move state

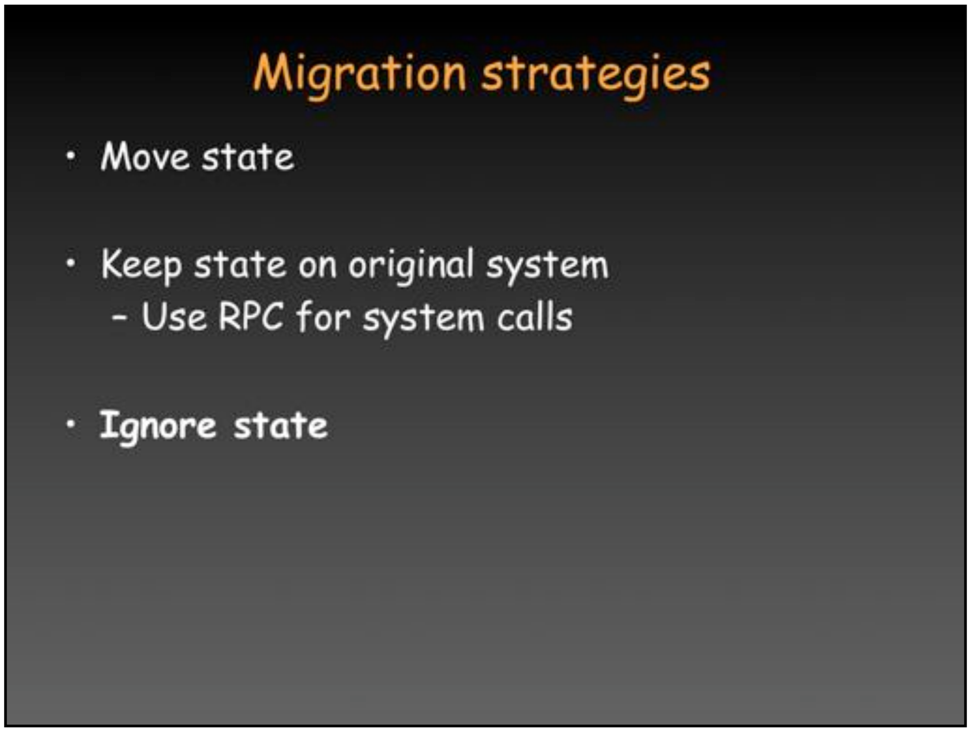
Three strategies for migration can be adopted.

The most thorough, and most complicated, is to move the entire system state. This means that open file descriptors have to be reconstructed on the remote system and the state of kernel objects such as signals, message queues and semaphores has to be propagated. Mechanisms should also exist for shared memory (if the os supports it) and sending signals/messages across different machines. To implement this requires a kernel that is capable of migrating this information as well as a global process ID space.


Migration strategies

- Move state
- Keep state on original system
  - Use RPC for system calls

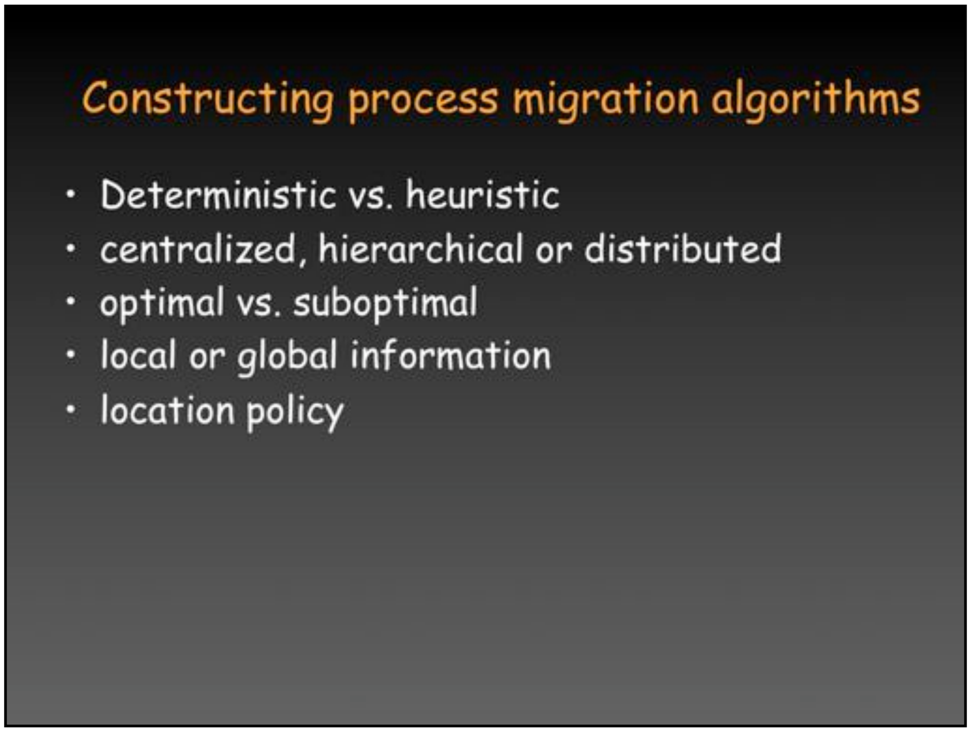
A somewhat easier design, still requiring operating system kernel modifications, is to maintain a concept of a “home” system. This is the approach taken by the Berkeley **Sprite** operating system (which is built from Berkeley Unix). The system on which a process is created is considered its “home”. The operating system supports the invocation of system calls through an operating-system-level remote procedure call mechanism. When a process that has migrated issues a system call (e.g. *read, write, ioctl, get time of day*), the operating system checks whether this machine is the process’ home system or whether it has migrated here. If it’s the home system, the call is processed locally. If the process migrated from another system, any system call that needs kernel state (such as file system operations) is forwarded to the home system (which maintains state on behalf of that process). The system call is processed on the home machine and results are returned to the requestor via the remote procedure call.



Migration strategies

- Move state
- Keep state on original system
  - Use RPC for system calls
- **Ignore state**

Finally, the easiest design is to assume that there is little or no state that deserves to be preserved. This is an approach taken by **Condor**, a software package that provides process migration for Unix systems without kernel changes. The assumption here is that there is no need for any inter-process communication mechanism: processes know they are running on a foreign system.



Constructing process migration algorithms

- Deterministic vs. heuristic
- centralized, hierarchical or distributed
- optimal vs. suboptimal
- local or global information
- location policy

There are a number of different issues in constructing processes migration algorithms:

deterministic vs. heuristic

> if we know all the resource usage up front, we can create a deterministic algorithm. This data is usually unknown and heuristic techniques often have to be employed.

Centralized, hierarchical, or distributed

> a centralized algorithm allows all the information necessary for making scheduling decisions to reside in one place but it can also put a heavy load on the central machine. With a hierarchical system, we can have a number of load managers, organized in a hierarchy. Managers make process allocation decisions as far down the tree as possible, but may transfer processes from one to another via a common ancester.

optimal vs. suboptimal

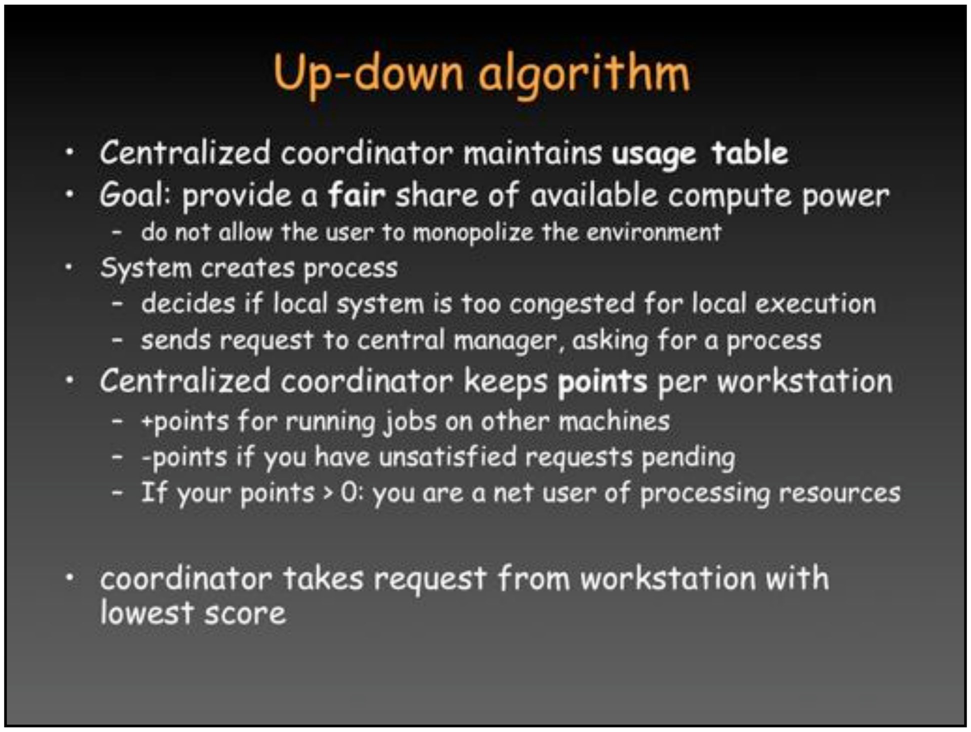
> do we really want the *best* allocation or simply an acceptable one? If we want the best allocation, we'll have to pay a price in the computation and data needed to make that decision. Quite often it's not worth it.

local or global?

> Does a machine decide whether a process stays on the local machine using local information (its system load, for example) or does it rely on global system state information? This is known as the transfer policy.

location policy

> Does the machine send requests asking for help or does it send requests for work to perform?



Up-down algorithm

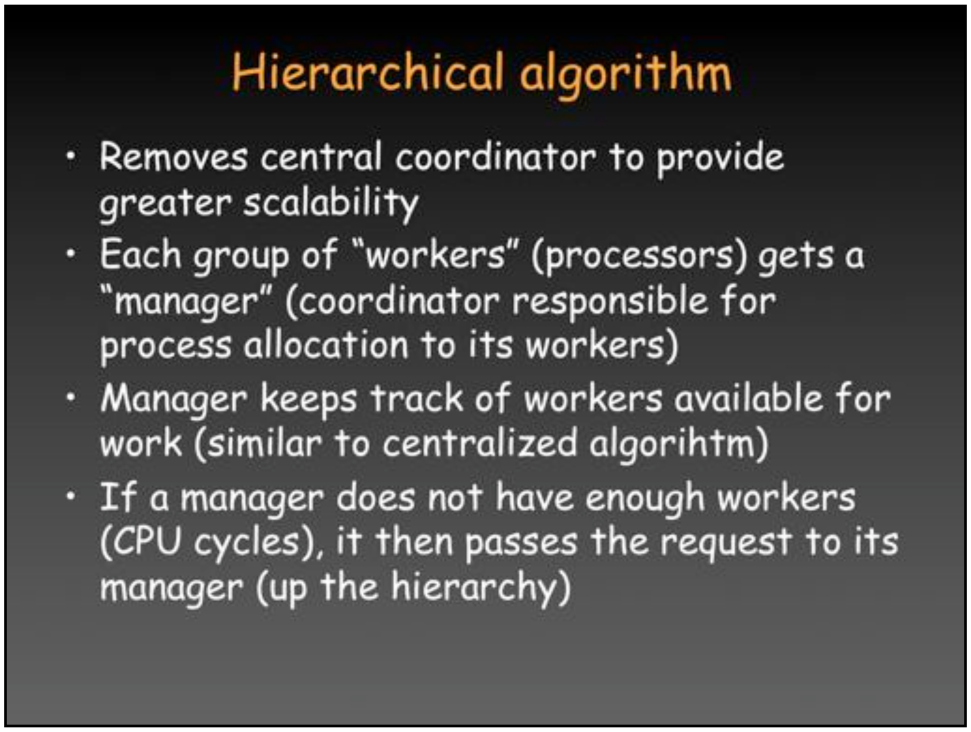
- Centralized coordinator maintains **usage table**
- Goal: provide a **fair** share of available compute power
  - do not allow the user to monopolize the environment
- System creates process
  - decides if local system is too congested for local execution
  - sends request to central manager, asking for a process
- Centralized coordinator keeps **points** per workstation
  - +points for running jobs on other machines
  - -points if you have unsatisfied requests pending
  - If your points > 0: you are a net user of processing resources
- coordinator takes request from workstation with lowest score

The up-down algorithm (Mutka and Livny, 1987) relies on a centralized coordinator which maintains a *usage table*. This table contains one entry per workstation. Workstations send messages containing updates to this coordinator. All allocation decisions are based on the data in this table.

The goal of the up-down algorithm is to give each workstation owner a fair share of the available compute power (and not allow the user to monopolize the environment).

When a system has to create a process, it first decides whether it should run it locally or seek help. This is generally done in most migration algorithms as an optimization (why seek help when you don't need it?). If it decides to ask for help, it sends a message to the coordinator asking for a processor.

The coordinator's table keeps *points* per workstation. If you run a process on another machine, you get penalty points which are added (*n*/second) to your entry in the usage table. If you have unsatisfied requests pending, then points are subtracted from your entry. If no requests are pending and no processors are used, your entry gradually erodes to zero. Looking at the points for a given workstation, a positive amount indicates that the workstation is a net user of resources and a negative amount indicates that the workstation needs resources. The coordinator simply chooses to process the request from the workstation with the lowest score.



Hierarchical algorithm

- Removes central coordinator to provide greater scalability
- Each group of "workers" (processors) gets a "manager" (coordinator responsible for process allocation to its workers)
- Manager keeps track of workers available for work (similar to centralized algorihtm)
- If a manager does not have enough workers (CPU cycles), it then passes the request to its manager (up the hierarchy)

The centralized algorithm has the same pitfall that all centralized algorithms share: scalability. A hierarchical processor allocation algorithm attempts to overcome scalability while still maintaining efficiency.

In this algorithm, every group of *k* workers gets a "manager" - a coordinator responsible for processor allocation to machines within its group.

Each manager keeps track of the approximate number of workers below it that are available for work.

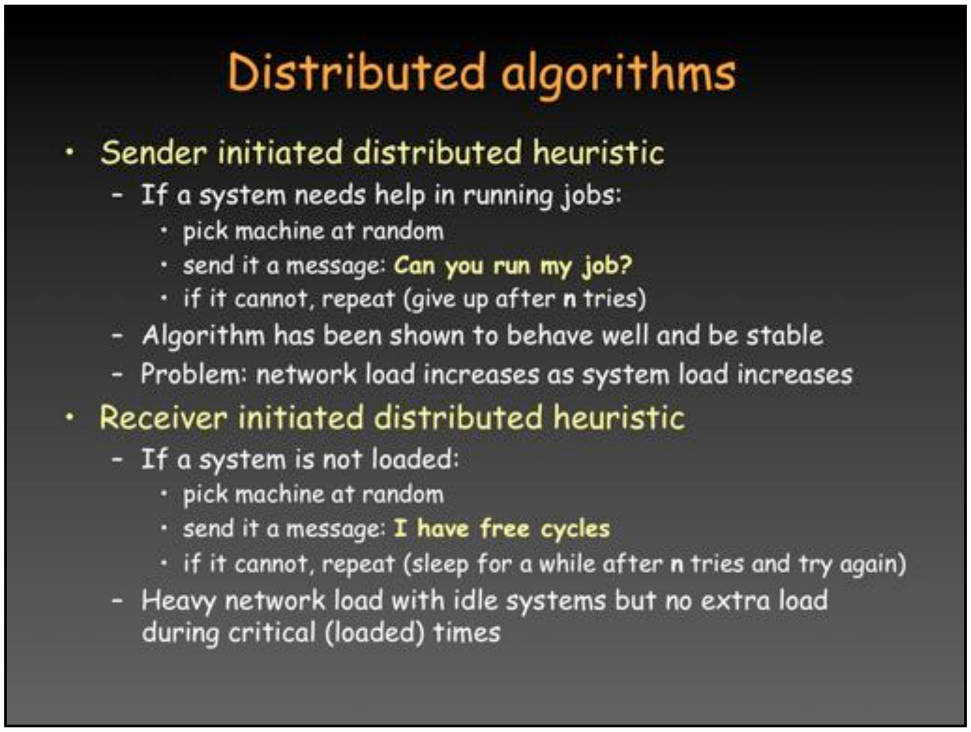
In this case, it behaves like a centralized algorithm.

If, for some job, the manager does not have enough workers (worker CPU cycles), it then passes the request to its manager (up the hierarchy).

The upper manager checks with its subordinates (the pool of up to *k* managers under it) for available workers.

If the request can be satisfied, it is parceled among the managers and, ultimitely, among the workers.

If it cannot be satisfied, the second-level manager may contact a third-level manager. The hierarchy can be extended ad infinitum.



Distributed algorithms

- Sender initiated distributed heuristic
  - If a system needs help in running jobs:
    - pick machine at random
    - send it a message: **Can you run my job?**
    - if it cannot, repeat (give up after **n** tries)
  - Algorithm has been shown to behave well and be stable
  - Problem: network load increases as system load increases
- Receiver initiated distributed heuristic
  - If a system is not loaded:
    - pick machine at random
    - send it a message: **I have free cycles**
    - if it cannot, repeat (sleep for a while after **n** tries and try again)
  - Heavy network load with idle systems but no extra load during critical (loaded) times

Sender initiated distributed heuristic

This algorithm requires no coordinator whatsoever. If a machine decides that it should not run its job locally, it picks a machine at random and sends it a probe message (*"can you run my job?"*).

If the randomly selected machine cannot run the job, another machine is picked at random and a probe sent to it.

The process is repeated until a willing machine is located or after *n* tries.

This algorithm has been shown to behave well and is stable. Its failing is when the overall system load gets heavy. At those times, many machines in the network are looping *n* times, sending requests to machines too busy to service them.

Receiver initiated distributed heuristic

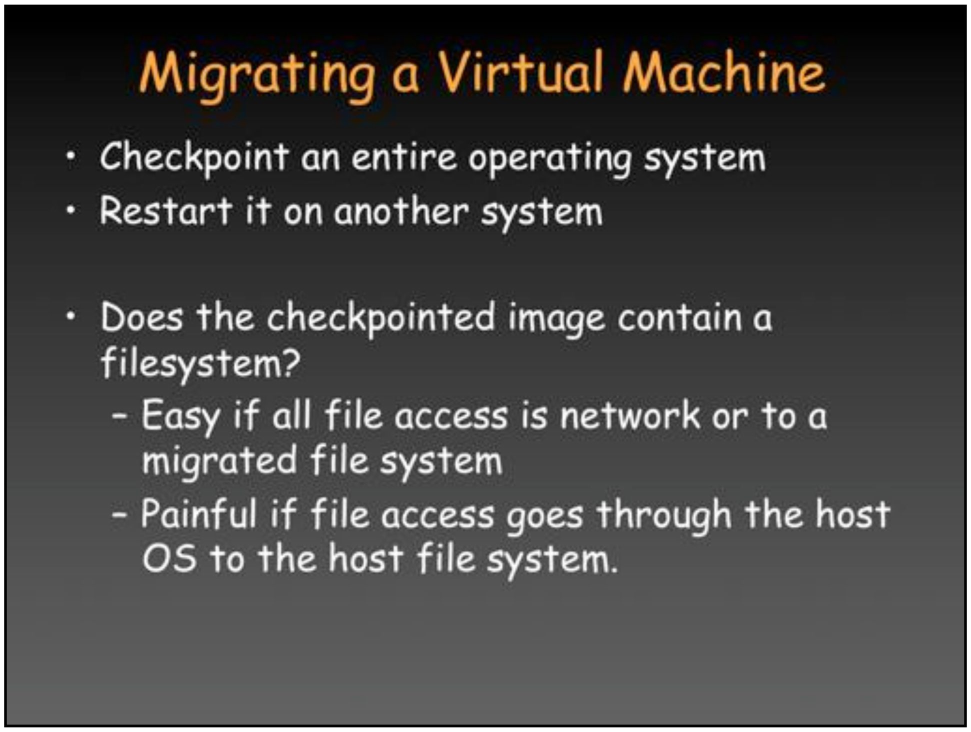
To overcome the problem of traffic in loaded systems, we can do the opposite of a sender initiated algorithm and have machines advertise themselves as being available for work.

In this algorithm, when a processor is done with a process, it picks some random machine and sends it a message: *"do you have any work for me?"*

If the machine responds in the affirmative, the sender gets a job.

If the machine has no work, the sender picks another machine and tries again, doing this *n* times. Eventually, the sender will go to sleep and then start the whole process again until it gets work.

While this creates a lot of messages, there is no extra load on the system during critical times.



Migrating a Virtual Machine

- Checkpoint an entire operating system
- Restart it on another system

- Does the checkpointed image contain a filesystem?
  - Easy if all file access is network or to a migrated file system
  - Painful if file access goes through the host OS to the host file system.

The End.