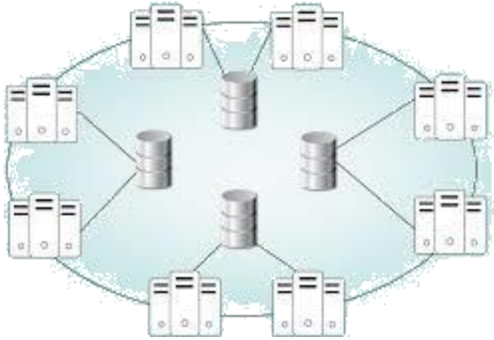
Distributed and Parallel Computer Systems

CSC 423

Spring 2021-2022

Lecture 10



Processes in Distributed Systems

INSTRUCTOR

DR / AYMAN SOLIMAN

➢ Contents

❑ Introduction to Threads

- We build virtual processors in software, on top of physical processors:

- Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

❑ Introduction to Threads

- Thread: A minimal software processor in whose context a series of instructions can be executed.
    - Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

- Process: A software processor in whose context one or more threads may be executed.

- Executing a thread, means executing a series of instructions in the context of that thread.

❑ Context Switching

- Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).

- Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions.

- Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread

❑ Context Switching

- Threads share the same address space.

- Process switching is generally more expensive
  - each Process has its own address space.

- Creating and destroying threads is much cheaper than doing so for processes.

❑ Threads

- In many distributed systems, it is possible to have multiple threads of control within a process.

- multiple threads of control sharing one address space but running in quasi-parallel

❑ Threads

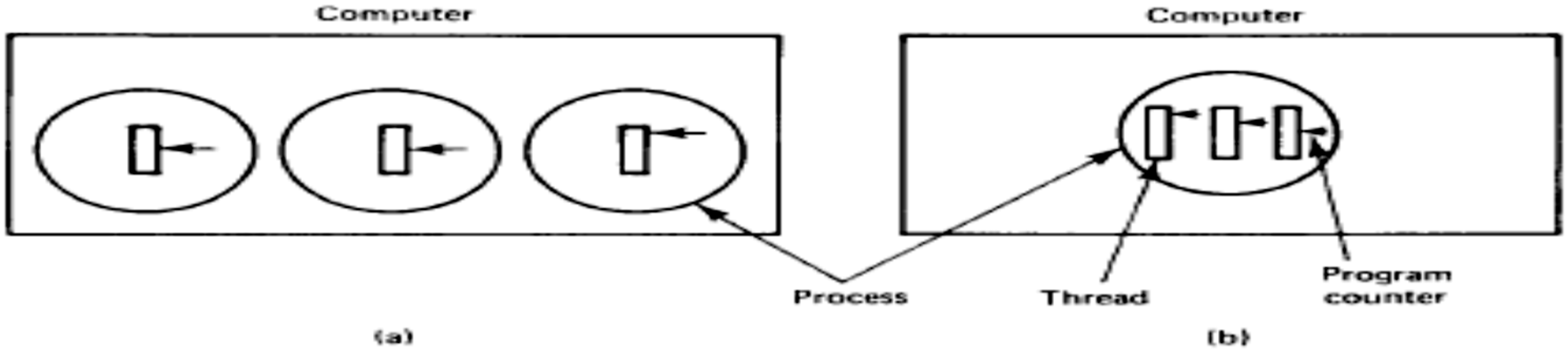
- For example, a file server that occasionally has to block waiting for the disk. If the server had multiple threads of control, a second thread could run while the first one was sleeping.

- It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space.

❑ Threads

- Each process has its own program counter, its own stack, its own register set, and its own address space.

- 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 (timesharing).

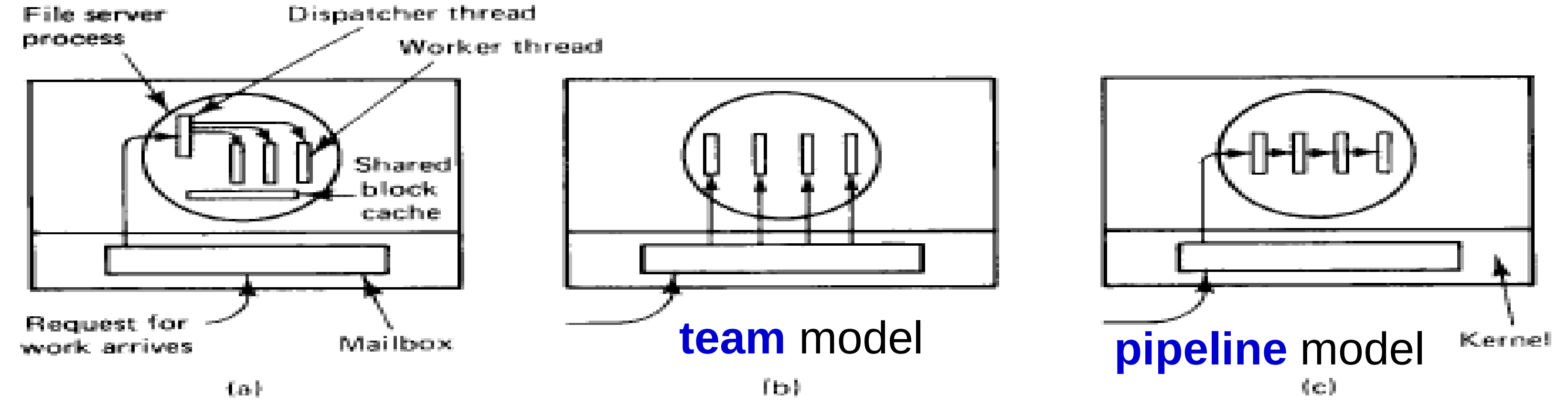


❑ Threads

- Threads can be in any one of several states: running, blocked, ready, or terminated.

❑ Thread Usage

- Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls.
- Consider our file server example again. One possible organization is shown in Fig. Here one thread, the dispatcher, reads incoming requests for work from the system mailbox.
- After examining the request, it chooses an idle worker thread.
- The dispatcher then wakes up the sleeping worker.

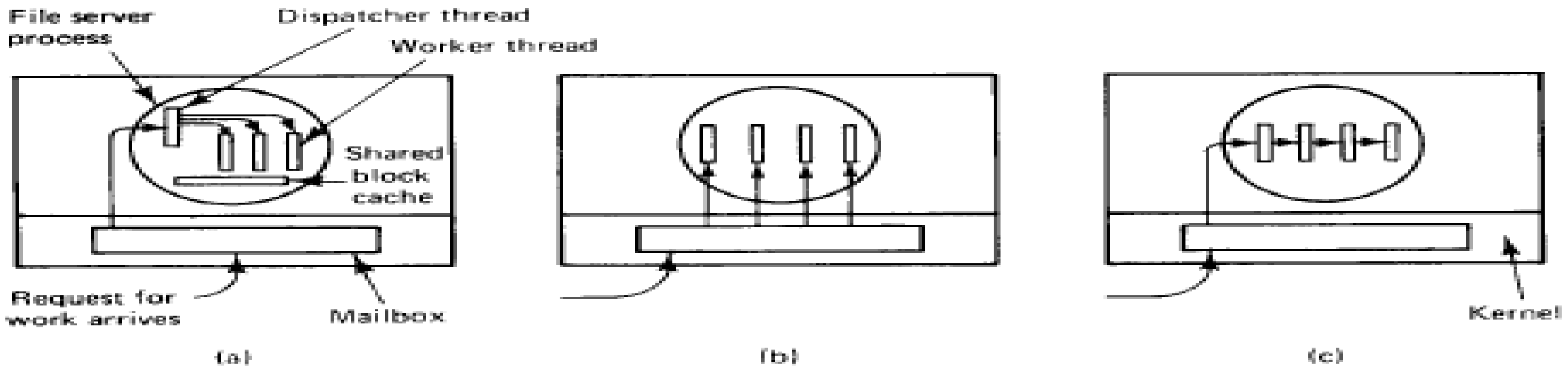


team model

pipeline model

❑ Thread Usage

- The team model is also a possibility. Here all the threads are equals, and each gets and processes its own requests. There is no dispatcher.
- Threads can also be organized in the pipeline model of previous In this model, the first thread generates some data and passes them on to the next thread for processing. The data continue from thread to thread, with processing going on at each step.



❑ Design Issues for Threads Packages

- 1.Thread management
  - Two alternatives are possible here, static threads and dynamic threads.
  - With a static design, 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.
  - Threads can be terminated in one of two ways. A thread can exit voluntarily when it finishes its job, or it can be killed from outside.

❑ Design Issues for Threads Packages

- 2.Access to shared data is usually programmed using critical regions, to prevent multiple threads from trying to access the same data at the same time.

  - One technique that is commonly used in threads packages is the mutex.
  - A mutex is always in one of two states, unlocked or locked. Two operations are defined on mutexes. The first one, LOCK, attempts to lock the mutex. If the mutex is unlocked, the LOCK succeeds and the mutex becomes locked in a single atomic action.
  - A thread that attempts to lock a mutex that is already locked is blocked.

❑ Design Issues for Threads Packages

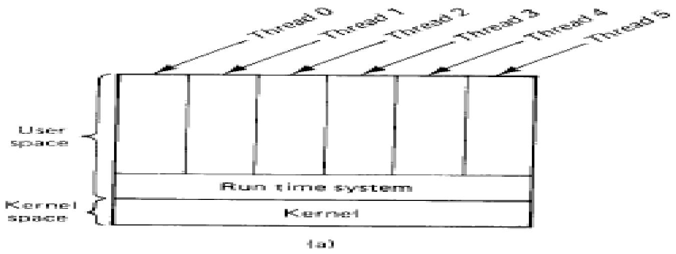
- Another operation that is sometimes provided is TRYLOCK, which attempts to lock a mutex.
- If the mutex is unlocked, TRYLOCK returns a status code indicating success.

- If the mutex is locked, TRYLOCK does not block the thread. Instead, it returns a status code indicating failure.

  - when the thread holding the resource frees it. it calls wakeup, which is defined to wakeup either exactly one thread or all the threads waiting on the specified condition variable.

❑ Implementing a Threads Package

- There are two main ways to implement a threads package: in user space and in the kernel.



- Implementing Threads in User Space
  - The kernel knows nothing about them.
  - The threads run on top of a runtime system, which is a collection of procedures that manage threads.
  - When a thread executes a system call, goes to sleep.
  - User-level threads allow each process to have its own customized scheduling algorithm.
- user-level threads packages have some major problems.
  - First among these is the problem of how blocking system calls are implemented.

❑ Implementing a Threads Package

- when a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction.

- To manage all the threads, the kernel has one table per process with one entry per thread. Each entry holds the thread's registers, state, priority, and other information.

- In addition, if one thread in a process causes a page fault, the kernel can easily run another thread while waiting for the required page to be brought in from the disk (or network).

❑ Scheduler Activations

- Scheduler activations combine the advantage of user threads (good performance) with the advantage of kernel threads (not having to use a lot of tricks to make things work).

- The goals of the scheduler activation work are
  - to mimic the functionality of kernel threads,
  - with the better performance and greater flexibility.

System Models

- In a distributed system, with multiple processors, The processors in a distributed system can be organized in several ways that the workstation model and the processor pool model, and a hybrid form encompassing features of each one.

- The Workstation Model
    - The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN,

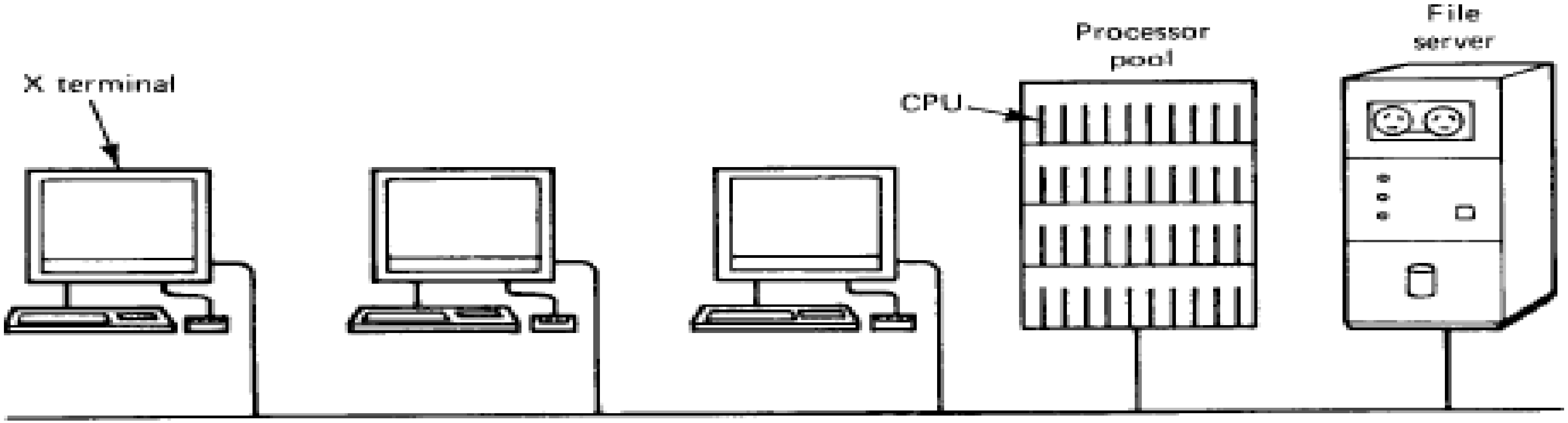
❑ Workstation Model

- In some systems the workstations have local disks and in others they do not.

- The last-mentioned are universally called diskless workstations, but the former are variously known as diskful workstations, or disky workstations, or even stranger names.

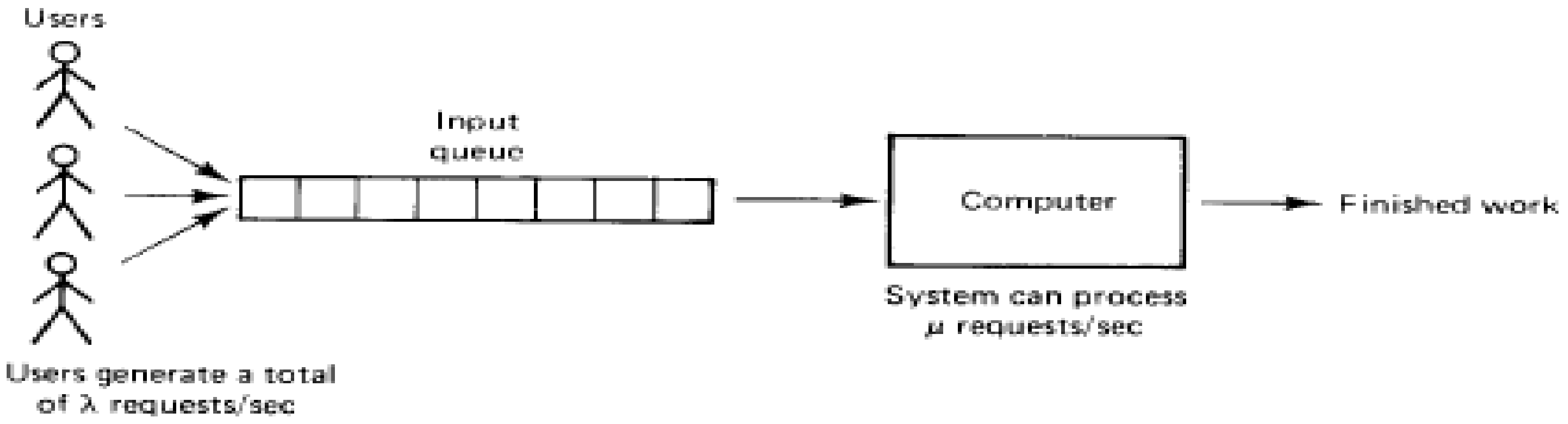
❑ The Processor Pool Model

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



❑ Queueing system

- A queueing system is a situation in which users generate random requests for work from a server.
- When the server is busy, the user's queue for service and are processed in turn.
- Common examples of queueing systems are bakeries and airport check-in counters.



❑ Queueing system

- Queueing systems are useful because it is possible to model them analytically.

- Let us call the total input rate λ requests per second, from all the users combined. Let us call the rate at which the server can process requests μ. For stable operation, we must have $\mu > \lambda$.

- It can be proven that the mean time between issuing a request and getting a complete response, T, is related to μ and λ by the formula

$$T = \frac{1}{\mu - \lambda}$$

❑ Processor Allocation

- In all cases, an algorithm is needed for deciding which process should be run on which machine.
- For the workstation model, the question is when to run a process locally and when to look for an idle workstation.
- For the processor pool model, a decision must be made for every new process.

❑ Allocation Models

- All published models assume that the system is fully interconnected, that is, every processor can communicate with every other processor.

❑ Processor Allocation

- Processor allocation strategies can be divided into two broad classes.

- In the first, nonmigratory, when a process is created, a decision is made about where to put it. Once placed on a machine, the process stays there until it terminates. It may not move.

- In contrast, with migratory allocation algorithms, a process can be moved even if it has already started execution. While migratory strategies allow better load balancing, they are more complex and have a major impact on system design.

❑ Processor Allocation

- Another worthy objective is minimizing mean response time.
- For example, the two processors and two processes Processor 1 runs at 10 MIPS; processor 2 runs at 100 MIPS but has a waiting list of backlogged processes that will take 5 sec to finish off.
- Process A has 100 million instructions and process B has 300 million.
- The response times for each process on each processor (including the wait time) are shown in the figure.

❑ Design Issues for Processor Allocation Algorithms

- The major decisions the designers must make can be summed up in five issues:

1. Deterministic versus heuristic algorithms.
2. Centralized versus distributed algorithms.
3. Optimal versus suboptimal algorithms.
4. Local versus global algorithms.
5. Sender-initiated versus receiver-initiated algorithms.


Thank you