CSE 601: Distributed Systems

Toukir Ahammed

Introduction to threads

- Although processes form a building block in distributed systems, practice indicates that the granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient.
- Instead, it turns out that having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance.

Process

- To execute a program, an operating system creates a number of virtual processors, each one for running a different program.
- To keep track of these virtual processors, the operating system has a **process table**, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc.
- A **process** is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors.
- The fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent.

Threads

- Thread means a segment of a process.
- Like a process, a thread executes its own piece of code, independently from other threads.
- However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation.
- Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads.

Why use threads?

- Process switching is generally (somewhat) more expensive as it involves the OS.
- Thread context switching can be done entirely independent of the operating system.
- Creating and destroying threads is much cheaper than doing so for processes.

Why use threads?

- Avoid needless blocking: a single-threaded process will block when doing I/O; in a multithreaded process, the operating system can switch the CPU to another thread in that process.
- Exploit parallelism: the threads in a multithreaded 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.

Why use threads?

There is also a pure software engineering reason to use threads:

- Many applications are simply easier to structure as a collection of cooperating threads.
- Think of applications that need to perform several (more or less independent) tasks.
- For example, in the case of a word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.

Why use threads?

- Instead of using processes, an application can also be constructed such that different parts are executed by separate threads.
- Communication between those parts is entirely dealt with by using shared data.
- Thread switching can sometimes be done entirely in user space.
- The effect can be a dramatic improvement in performance.

Thread usage in nondistributed systems

- Consider an application such as a spreadsheet program, and assume that a user continuously and interactively wants to change values.
- An important property of a spreadsheet program is that it maintains the functional dependencies between different cells, often from different spreadsheets.
- Therefore, whenever a cell is modified, all dependent cells are automatically updated.
- When a user changes the value in a single cell, such a modification can trigger a large series of computations.

Thread usage in nondistributed systems

- If there is only a single thread of control, computation cannot proceed while the program is waiting for input.
- Likewise, it is not easy to provide input while dependencies are being calculated.
- The easy solution is to have at least two threads of control:
  - one for handling interaction with the user and
  - one for updating the spreadsheet.
- In the mean time, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.

Threads in distributed systems

- An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running.
- This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.

Multithreaded clients

- A typical example where this happens is in Web browsers.
- In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc.
- To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component.
- Setting up a connection as well as reading incoming data are inherently blocking operations.

Multithreaded clients

- When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.
- A Web browser often starts with fetching the HTML page and subsequently displays it.
- To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in.

Multithreaded clients

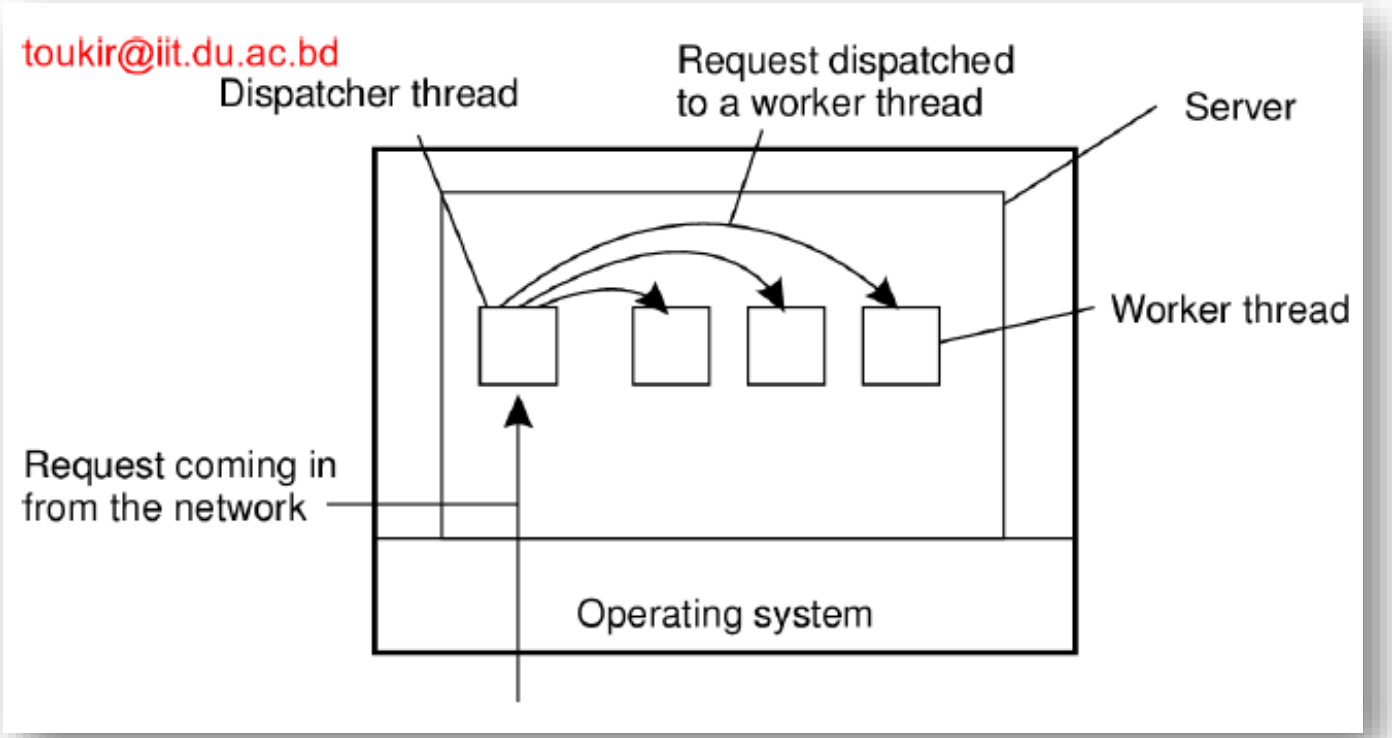
- While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images.
- The latter are displayed as they are brought in.
- The user need thus not wait until all the components of the entire page are fetched before the page is made available.

Multithreaded servers

- Consider the organization of a file server that occasionally has to block waiting for the disk.
- The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply.

Multithreaded servers

- Here one thread, the **dispatcher**, reads incoming requests for a file operation.
- The requests are sent by clients to a well-known end point for this server.
- After examining the request, the server chooses an idle (i.e., blocked) **worker thread** and hands it the request.



Exercise

- In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in a cache in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

Solution

- In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is:

$$2/3 \times 15 + 1/3 \times 90 = 40$$

- Thus, the mean request takes 40 msec and the server can do 1000/40 = 25 request per second.
- For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 1000/15 = 66.67 requests per second.