Parallel & Distributed System

Process, Threads & Virtualization

Md. Biplob Hosen

Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

Contents

- **Threads:**
  - ✓ Introduction to Threads
  - ✓ Thread Usage in non-distributed Systems
  - ✓ Threads in Distributed Systems
- **Virtualization:**
  - ✓ Principle of Virtualization

- Reference Books
  - ✓ Distributed Systems: Principles and Paradigms, 3rd Edition by Andrew S. Tanenbaum & Maarten van Steen, Publisher: Pearson Prentice Hall **[CH-03]**.

Overview

- The concept of a process originates from the field of operating systems where it is generally defined as a program in execution.
- From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with.
- However, when it comes to distributed systems, other issues turn out to be equally or more important.
- 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 get better performance.

Introduction to Threads

- 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.
- Jointly, these entries form a **process context**.
- 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 operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior.
- In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent. Usually, the operating system requires hardware support to enforce this separation.

Continue…

- For example, each time a process is created, the operating system must create a complete independent address space.
- Likewise, switching the CPU between two processes may require some effort as well.
- In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.
- A thread also executes its own piece of code, independently from other threads.
- A thread system generally maintains only the minimum information to allow a CPU to be shared by several threads.
- In particular, a thread context often consists of nothing more than the processor context, along with some other information for thread management.
- For this reason, protecting data against inappropriate access by threads within a single process is left entirely to application developers.

Continue…

- We thus see that a processor context is contained in a thread context, and that, in turn, a thread context is contained in a process context.
- There are two important implications of deploying threads.
- First of all, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading even leads to a performance gain.
- Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort.

Threads in non-distributed Systems

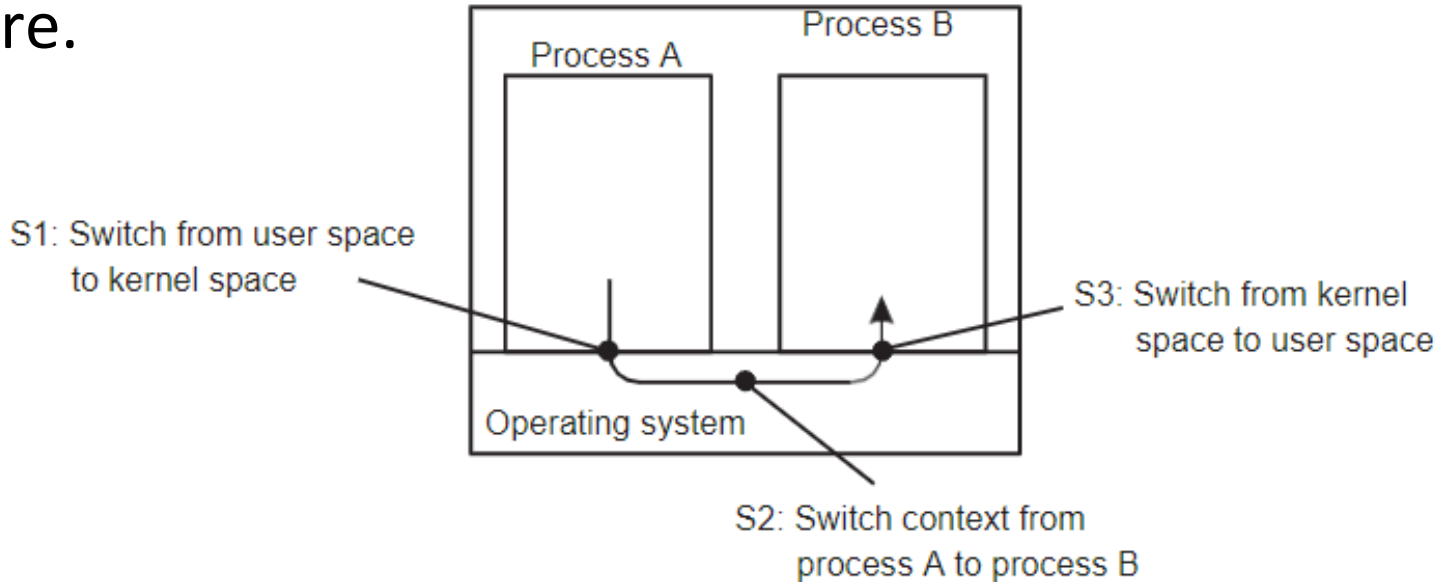
- There are several benefits to multithreaded processes that have increased the popularity of using thread systems.
- The most important benefit comes from the fact that in a single-threaded process, whenever a blocking system call is executed, the process as a whole is blocked.
- To illustrate, consider 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.
- If there is only a single thread of control, computation cannot proceed while the program is waiting for input.

Continue…

- 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 meantime, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.
- Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system.
- In that case, each thread is assigned to a different CPU or core while shared data are stored in shared main memory.
- When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, albeit slower.
- Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor and multicore computers.
- Such computer systems are typically used for running servers in client-server applications, but are by now also used in devices such as smartphones.

Continue…

- Multithreading is also useful in the context of large applications.
- Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process.
- This approach is typical for a Unix environment. Cooperation between programs is implemented by means of interprocess communication (IPC) mechanisms.
- The major drawback of all IPC mechanisms is that communication often requires relatively extensive context switching, shown at three different points in the following figure.



Continue…

- Because IPC requires kernel intervention, a process will generally first have to switch from user mode to kernel mode, shown as S1. This requires changing the memory map in the MMU.
- Within the kernel, a process context switch takes place (S2 in the figure), after which the other party can be activated by switching from kernel mode to user mode again (S3).
- 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, although in other implementations, the kernel is aware of threads and schedules them.
- The effect can be a dramatic improvement in performance.

Thread implementation

- Threads are often provided in the form of a thread package.
- Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes.
- There are basically two approaches to implement a thread package.
- The first approach is to construct a thread library that is executed entirely in user space.
- The second approach is to have the kernel be aware of threads and schedule them.
- A user-level thread library has a number of advantages. First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.
- A second advantage of user-level threads is that switching thread context can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched.

Continue…

- A major drawback of user-level threads comes from deploying the many to-one threading model: multiple threads are mapped to a single schedulable entity.
- As a consequence, the invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.
- As we explained, threads are particularly useful to structure large applications into parts that could be logically executed at the same time.
- In that case, blocking on I/O should not prevent other parts to be executed in the meantime.
- For such applications, user-level threads are of no help.
- These problems can be mostly circumvented by implementing threads in the operating system’s kernel, leading to what is known as the one-to-one threading model in which every thread is a schedulable entity.

Continue…

- The price to pay is that every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel, requiring a system call.
- Switching thread contexts may now become as expensive as switching process contexts.
- However, in light of the fact that the performance of context switching is general dictated by ineffective use of memory caches, and not by the distinction between the many-to-one or one-to-one threading model, many operating systems now offer the latter model, if only for its simplicity.

Continue…

- Sometimes, using processes instead of threads has the important advantage of separating the data space: each process works on its own part of data and is protected from interference from others through the operating system.
- The advantage of this separation should not be underestimated: thread programming is considered to be notoriously difficult because the developer is fully responsible for managing concurrent access to shared data.
- Using processes, data spaces, in the end, are protected by hardware support.
- If a process attempts to access data outside its allocated memory, the hardware will raise an exception, which is then further processed by the operating system. No such support is available for threads concurrently operating within the same process.
- A good example of this approach is the organization of the Apache Web server, which, by default, starts with a handful of processes for handling incoming requests. Each process forms a single-threaded instantiation of the server, yet is capable of communicating with other instances through fairly standard means.

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.
- We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.

Multithreaded Clients

- To establish a high degree of **distribution transpa**rency, distributed systems that operate in wide-area networks may need to conceal long interprocess message propagation times.
- The round-trip delay in a wide-area network can be in the order of hundreds of milliseconds, or sometimes even seconds.
- The usual way to hide communication latencies is to initiate communication and immediately proceed with something else.
- 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.

Continue…

- 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.
- 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.
- In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably.
- As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts.

Continue…

- There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server.
- If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed.
- However, in many cases, Web servers have been replicated across multiple machines, where each provides exactly the same set of Web documents.
- The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique.
- When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a non-replicated server.

Multithreaded Servers

- Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is at the server side.
- Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems.
- However, with modern multicore processors, multithreading for parallelism is an obvious path to follow.
- To understand the benefits of threads for writing server code, 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.
- One possible, and particularly popular organization is shown in the figure.

Continue…

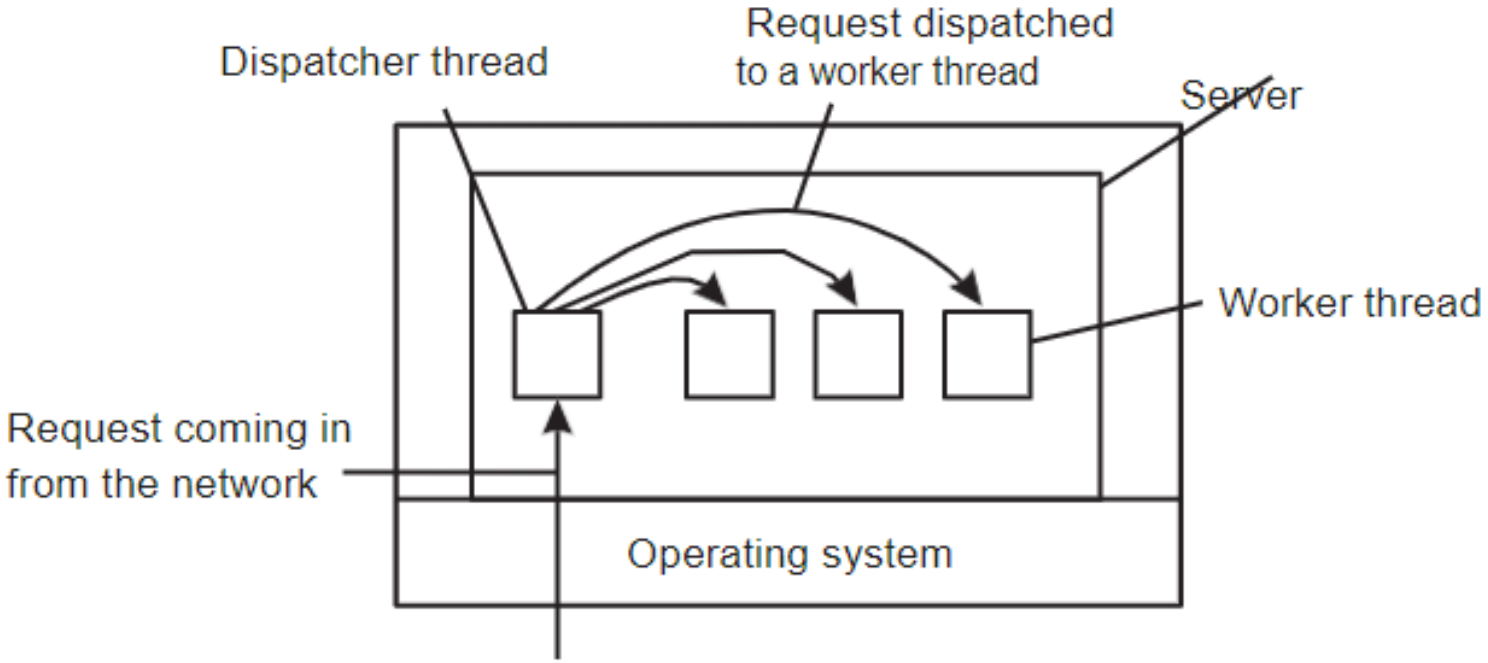


Figure: A multithreaded server organized in a dispatcher/worker model.

- 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.
- The worker proceeds by performing a blocking read on the local file system, which may cause the thread to be suspended until the data are fetched from disk.

Continue…

- If the thread is suspended, another thread is selected to be executed.
- For example, the dispatcher may be selected to acquire more work.
- Or, another worker thread can be selected that is ready to run.
- Now consider how the file server might have been written in the absence of threads.
- One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one.
- While waiting for the disk, the server is idle and does not process any other requests.
- Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk.
- The net result is that many fewer requests per time unit can be processed.
- Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.
- So far we have seen two possible designs: **a multithreaded file server and a single-threaded file server**.

Continue…

- A third alternative is to run the server as a big single-threaded finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the in-memory cache, fine, but if not, the thread must access the disk.
- However, instead of issuing a blocking disk operation, the thread schedules an asynchronous (i.e., nonblocking) disk operation for which it will be later interrupted by the operating system.
- To make this work, the thread will record the status of the request (namely, that it has a pending disk operation), and continues to see if there were any other incoming requests that require its attention.
- Once a pending disk operation has been completed, the operating system will notify the thread, who will then, in due time, look up the status of the associated request and continue processing it.
- Eventually, a response will be sent to the originating client, again using a nonblocking call to send a message over the network.

Continue…

- It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls and still achieve parallelism.
- Blocking system calls make programming easier as they appear as just normal procedure calls.
- In addition, multiple threads allow for parallelism and thus performance improvement.
- The single-threaded server retains the ease and simplicity of blocking system calls, but may severely hinder performance in terms of number of requests that can be handled per time unit.
- The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls, which is generally hard to program and thus to maintain.

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

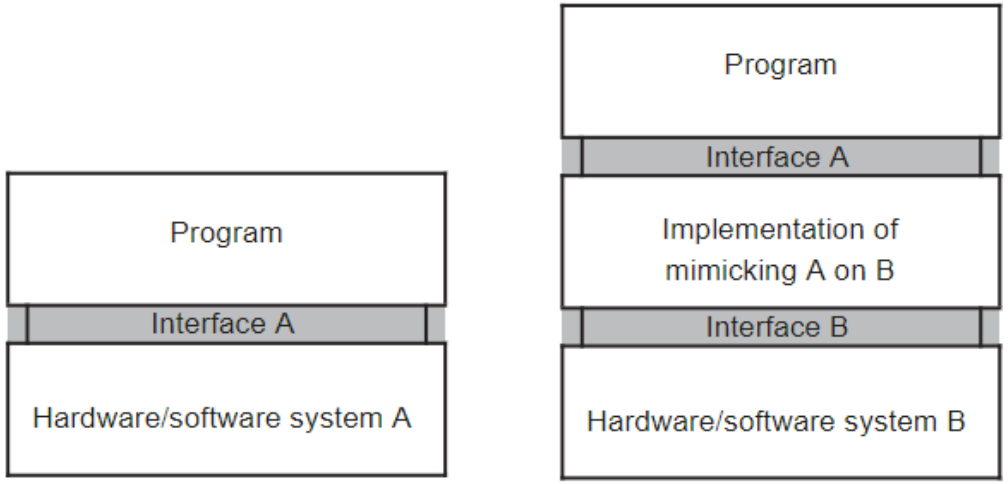
Three ways to construct a server

Virtualization

- Threads and processes are ways to do more things at the same time.
- In effect, they allow us to build programs that appear to be executed simultaneously.
- On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time.
- By rapidly switching between threads and processes, the illusion of parallelism is created.
- This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as **resource virtualization**.

Principle of Virtualization

- In practice, every (distributed) computer system offers a programming interface to higher-level software, as shown in the figure-a.
- There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems.
- In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in figure-b.



(a) General organization between a program, interface, and system
(b) General organization of virtualizing system A on top of B

Virtualization and Distributed Systems

- One of the most important reasons for introducing virtualization, was to allow legacy software to run on expensive mainframe hardware.
- The software not only included various applications, but in fact also the operating systems they were developed for.
- As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reducing, virtualization became less of an issue.
- However, matters have changed again since the late 1990s. First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction (e.g., middleware and applications), are often much more stable.
- In other words, we are facing the situation that legacy software cannot be maintained in the same pace as the platforms it relies on.
- Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Continue…

- Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network.
- This connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients.
- At the same time the various resources should be easily accessible to these applications.
- Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries and operating system, which, in turn, run on a common platform.
- This last type of virtualization provides a high degree of portability and flexibility.

Continue…

- For example, in order to realize content delivery networks that can easily support replication of dynamic content, management becomes much easier if edge servers would support virtualization, allowing a complete site, including its environment to be dynamically copied.
- These arguments are still valid, and indeed, portability is perhaps the most important reason why virtualization plays such a key role in many distributed systems.

Thank You ☺