Threads

Multiple threads of control per process makes it much easier to build distributed applications and to get better performance as opposed to having only one or more processes.

Processes

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, priviledges, etc. Jointly these form a process context.

Process context is similar to **processor context**, which may store program counter, return address, sometimes register values such as the stack pointer.

Process = program in execution on one of the OS's virtual processors.

Multiple processes should not affect the correctness of each other's behaviour. They may share the same CPU and other resources concurrently, this fact is made transparent (requires special hardware support). This transparency comes at a price, for example, each time a process is created, the OS must create a seperate address space. Allocation can mean initialising the memory segments with say `0`, setting up the stack to hold temporary data, copying the program into the static text section. It may need to swap some processes, if there are more processes that can fit into the main memory and yada yada, yada.

Now talking about *threads*, they execute their own piece of code, independently from other threads. However, in contrast to processes, no attempt to achieve high degree of concurrency is made if this would result in degradation of performance. Therefore, a thread system generally contains minimum amount of information to allow a CPU to shared by several threads.

A thread context often contains nothing more than processor context, along with some other information for thread management. For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution. Information that is not strictly necessary to manage multiple threads is ignored. For this reason, protecting data against inappropiate access by other threads is left entirely to application developers.

Two Important Implications:

- Performance of a multithreaded application need hardly ever be worse than its single-threaded counterpart. In fact, in many cases, multithreading may be even lead to performance gain.
- Threads are automatically protected against each other the way processes are, development of multithreaded applications require additional intellectual effort.

Threads in non-distributed 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. If there is only a single thread of control, computation cannot proceed while the program is waiting for the input.

Easy solution: 2 threads, one for taking input and another for updating the UI. In addition, a third one can save the changes to the disk.

Another advantage: It becomes possible to exploit parallelism when executing the problem on a multiprocessor or a multicore system. Each thread can be assigned to a different CPU/core while shared data is stored in the main memory (shared). When properly designed, such parallelism can be transparent: can equally well on a uni-processor system, albeit slower.

Large applications are often developed as a collection of cooperating programs, each to be executed by a seperate process. Cooperation between programs is implemented by means of **interprocess communication (IPC)** mechanisms. For unix systems, these include (named) pipes, message queues, and shared memory segments. **Drawback: Context Switching.** (as shown below)

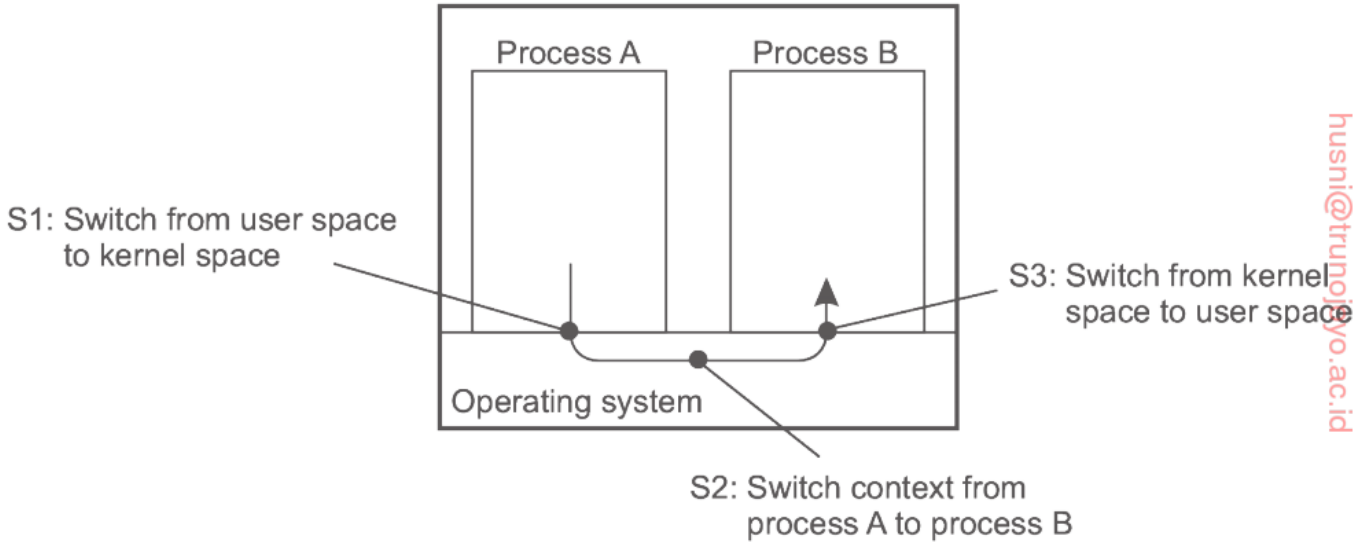

Figure 3.1: Context switching as the result of IPC.

changing from user mode to kernel mode: requires changing the memory map in the MMU, as well as flushing the TLB.

Instead of using processes, the application can be constructed such that different parts are executed by seperate threads. Communication is done entirely by shared data. Thread switching can be done entirely in user space => dramatic improve in performance.

Finally: Many applications are simply easy to structure as a collection of cooperating threads.

Threading Implementation

Often provided in a threads package, consisting of operations to create and destroy threads as well as operations on synchronization variables such as mutex and condition variables.

Two approaches to construct a threads package:

- A thread library that is executed entirely in user space
- Have the kernel be aware of threads and schedule them.

User Level Thread Library

Advantages

- It is cheap to create and destroy threads (the cost is primarily determined by allocating memory to set up a thread stack).
- Switching thread context can often be done in just a few instructions (Only the values of the CPU registers need to stored and subsequently reloaded with previously stored values of the thread being switched).

Disadvantages

- Deploys the **many-to-one threading model**: multiple threads are mapped into a single schedulable entity. As a consequence, a blocking system call will immediately block the entire process.

Kernel Level Thread Library

The above problem is solved by implementing threads in OS's kernel, leading to what is known as **one-to-one threading model** in which every thread is schedulable entity.

Drawback: Every thread operation (creation, deletion, synchronization, etc) will have to carried out by the kernel, requiring a system call => Switching thread contexts will becomes expensive.

Threads vs Processes: Using concurrent processes has the advantage of seperating the data space which can prevent others from interfering with its data through the OS which in the case of threads is all manual work to be done by the application developer.

Multithreaded Clients

Distributed systems that operate in wide-area networks may need to hide long interprocess message propogation times.

Web browsers display the web document that is being fetched as it coming (in parts, meaning it doesn't need to wait for the entire page to be fetched) to do the above, also the user thus not wait for all the components to be fetched before the page is displayed/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, seperate threads can be activated to take care of fetching the other parts (images, videos, fonts, etc).

Each thread => seperate connection to the server => fetch data (standard blocking system calls can be used, provided they do not suspend the entire process)

Code for each thread is the same and simple. Several connections to the same server is made, if the server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling the files that make up the page one after the other.

However, in many cases, the web servers have been replicated across multiple machines, where each server 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 comes in, it is forwarded to one of the servers, often using a round robin strategy or some other load balancing scheme. When using a multithreaded client, connections may set up to different replicas, allowing the data to be travelled in parallel, effectively establishing that the entire document is displayed in a much shorter time than with a non-replicated server provided the client can handle truly parallel streams of incoming data.

Multithreaded Servers

Consider the organisation of a file server that occassionally has to block waiting for the disk.

One possible organisation:

- The **dispatcher**, reads an incoming request for a file operation.
- The requests are sent by clients to a well known end point of the 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 the disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

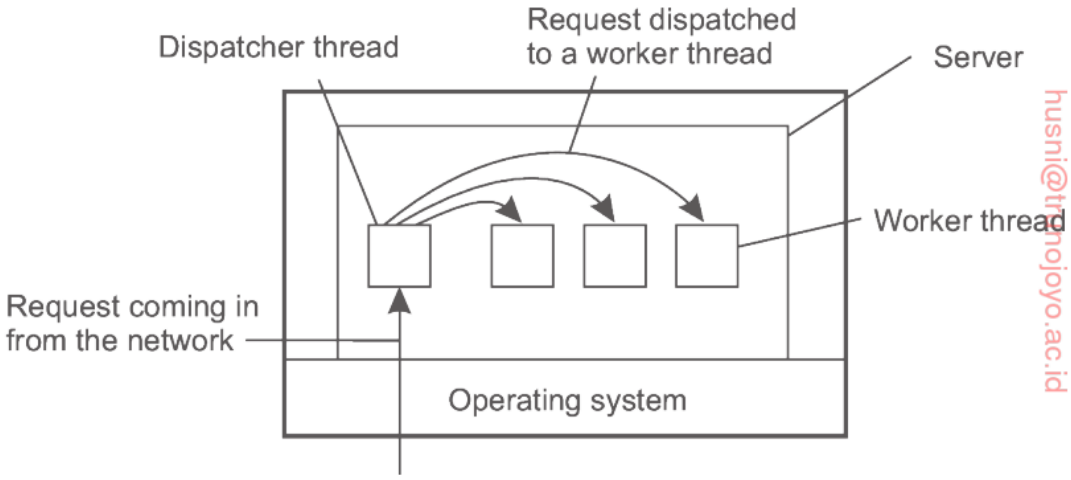

Figure 3.4: A multithreaded server organized in a dispatcher/worker model.

In a single threaded server, the main loop of the file server gets a request, examines it, and carries out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. In addition, the 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 unit time can be processed.

So, possible design choices:

- Multithreaded file server
- Single-threaded file server
- A big single-threaded finite-state server

Single-threaded finite-state server (case of NodeJS, I think!)

When a request comes in, one and only one thread examines it. If it can be satisfied by the in-cache memory, fine, but if not, thread must access the disk. However, instead of issuing a blocking disk operation, the thread schedules an asynchronous disk operation for which it will be later interrupted by the OS. To make this work, the thread will record the status of the rquest (namely, that it has a pending disk operation), and continues to see if there was any other incoming requests that require its attention.

Once the pending disk operation has been completed, the OS 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 client, again using a non-blocking call to send a message over the network.

Three Ways to construct a server

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