Advanced Operating Systems

Chapter 3: **Processes**

Reihaneh khorsand

Contents

- Threads
- Multithreaded Clients
- Multithreaded Servers
- Code Migration

THREADS

A **process** is generally defined as **a program in execution**.

Having a form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance.

Threads *are often provided in the form of a thread package.*

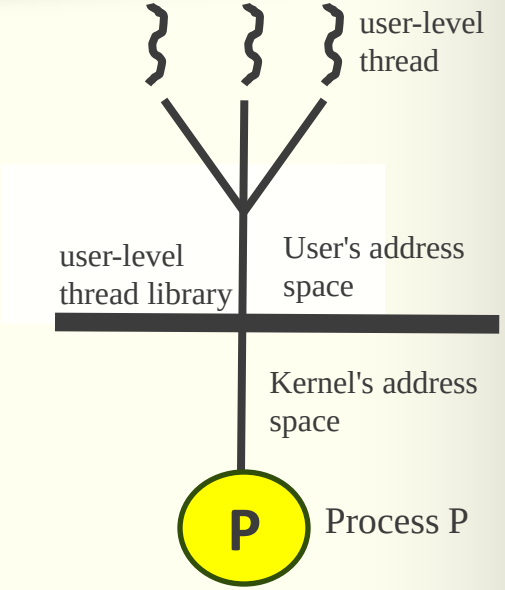
Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as condition variables.

There are basically **two approaches** to **implement a thread package:**

1. The first approach is to construct a thread library that is executed entirely in user mode.
2. The second approach is to have the kernel be aware of threads and schedule them.

ULT s(User Level Threads)

Advantages



✓ It is cheap to create and destroy threads.
Because all thread administration is kept in the user's address space

✓switching thread context can often be done in just a few instructions.

✓Particular scheduling

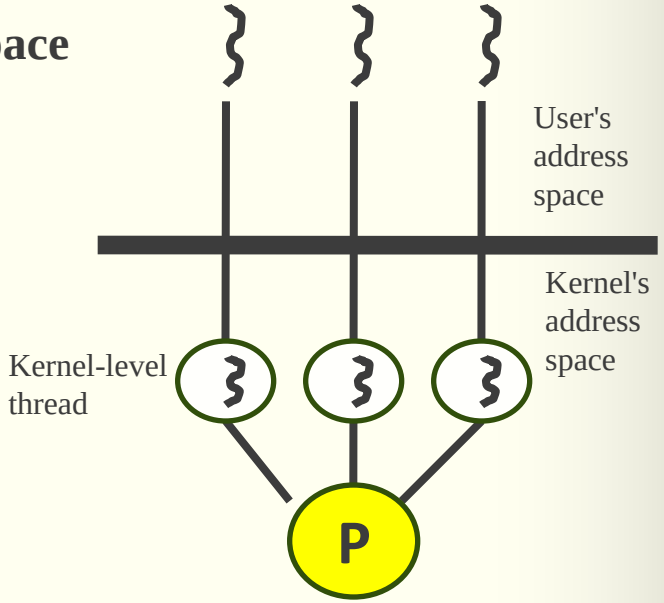
✓User-level threads can execute on every OS

Disadvantages

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.

KLTs (Kernel Level Threads)

All thread administration is kept in the kernel's address space



Advantages

Invocation of a system call is non blocking

Disadvantages

- There is a high price to pay: 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.

A hybrid form of user-level and kernel-level threads: lightweight processes (LWP)

An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.

In addition to having LWPs, a system also offers a user-level thread package.

Offering applications the usual operations for creating and destroying threads.
In addition, the package provides facilities for thread synchronization. such as condition variables.

The **important issue** is that the thread package is implemented entirely in user space.
In other words. all operations on threads are carried out without intervention of the kernel.

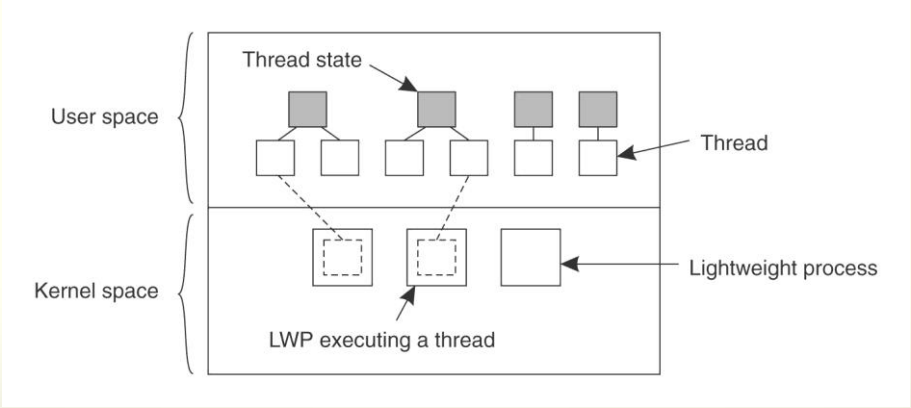


Fig 2: Combining kernel-level lightweight processes and user-level threads

lightweight processes (LWP) (continue)

Each LWP can be running its own (user-level) thread.

Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP.

Assigning a thread to an LWP is normally implicit and hidden from the programmer.

When creating an LWP (which is done by means of a system call), the LWP is given its own stack, and is instructed to execute the scheduling routine in search of a thread to execute.

If there are several LWPs, then each of them executes the scheduler.
The **thread table**, which is used to keep track of the current set of threads, is thus shared by the LWPs.

Protecting this table to guarantee **mutually exclusive** access is done by means of mutexes that are implemented entirely in user space.

lightweight processes (LWP) (continue)

In other words, synchronization between LWPs does not require any kernel support.

When an LWP finds a runnable thread, it switches context to that thread.

Meanwhile, other LWPs may be looking for other runnable threads as well.

If a thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the **scheduling routine**.

'When another runnable thread has been found, a context switch is made to that thread.

The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Several advantages to using LWPs in combination with a user-level thread package:

1. Creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all.
2. Provided that a process has enough LWPs, a blocking system call will not suspend the entire process.
3. There is no need for an application to know about the LWPs. All it sees are user-level threads.
4. LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application.

The **only drawback** of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads.

However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

Multithreaded Clients

Multithreaded Clients → To establish a high degree of distribution transparency

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.**

Developing the **browser** as a **multithreaded client** simplifies matters considerably.

Multithreaded Clients (continue)

As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts.

Each thread sets up a separate connection to the server and pulls in the data.

Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process.

Multithreaded Clients (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

Web servers have been **replicated** across multiple machines, where each server provides exactly the same set of Web documents.

Multithreaded Clients (continue)

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 no replicated server.

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.

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.

Multithreaded Servers

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.

If the thread is suspended, another thread is selected to be executed.

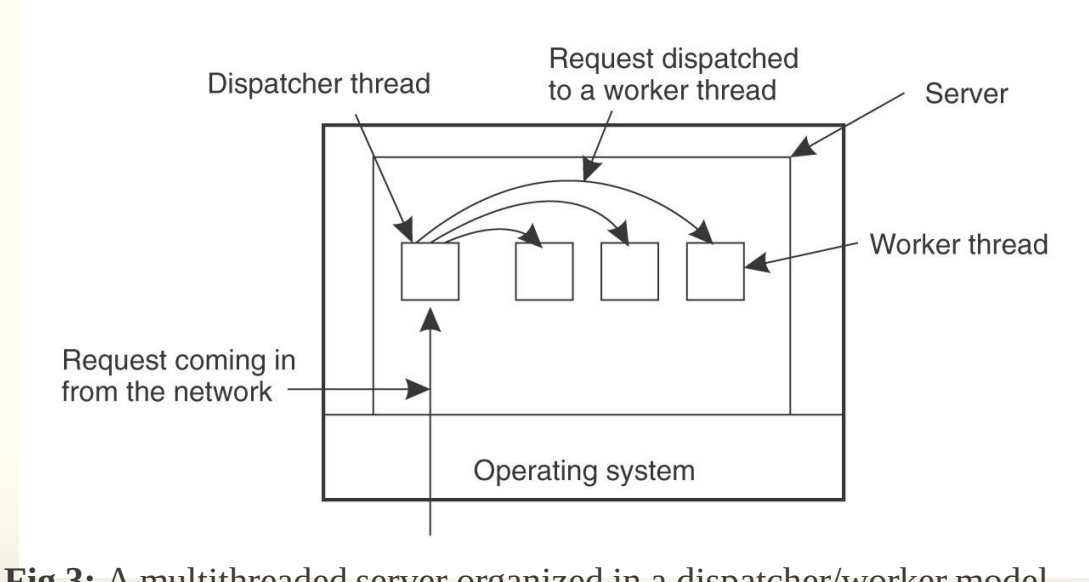


Fig 3: A multithreaded server organized in a dispatcher/worker model.

CODE MIGRATION

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another

Reasons for Migrating Code:

✓Performance

✓Flexibility

Reasons for Migrating Code/Performance

The basic idea is that overall system performance can be improved if processes are moved from **heavily-loaded** to **lightly-loaded** machines.

Load is often expressed in terms of the CPU queue length or CPU utilization.

A client-server system in which the server manages a huge database.
If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network.

In this case, code migration is based on the assumption that it generally makes sense **to process data close to where those data reside**.

Reasons for Migrating Code/Performance

This same reason can be used for migrating parts of the server to the client.

In many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations.

Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network.

The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Reasons for Migrating Code/Performance

Support for code migration can also help improve performance **by exploiting parallelism**, but without the usual intricacies related to parallel programming.

A typical example is searching for information in the Web.

It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site.

By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speedup compared to using just a single program instance.

Reasons for Migrating Code/Flexibility

If code can move between different machines, it becomes possible to dynamically configure distributed systems.

For example;

Suppose a server implements a standardized interface to a file system.
To allow remote clients to access the file system, the server makes use of a proprietary protocol.

Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application.

This approach requires that the software be readily available to the client at the time the client application is being developed.

Reasons for Migrating Code/Flexibility

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server.

At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server.

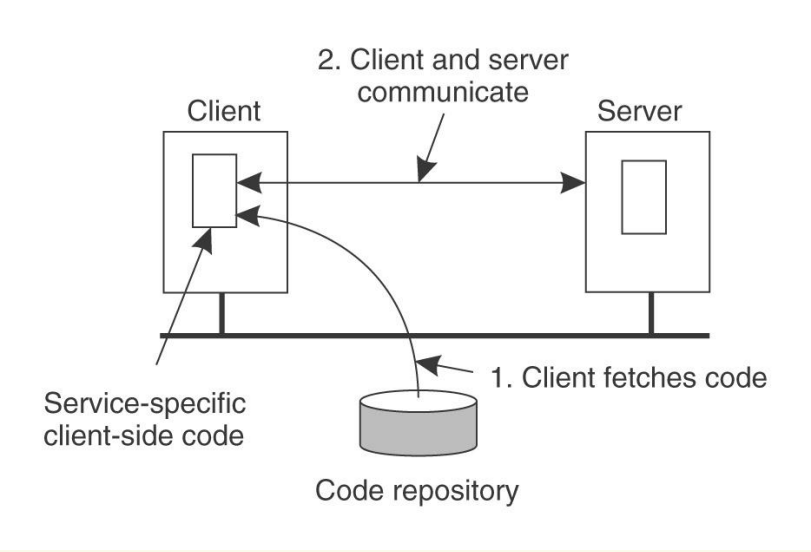


Fig 17: The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Reasons for Migrating Code/Flexibility

This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing **code** is standardized.

Clients need not have all the software preinstalled to talk to servers.

Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed.

Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like.

Changes will not affect existing client applications that rely on the server.

Disadvantages: Low security

Code migration types

Exchanging **data** between processes

Moving **programs** between machines, with the intention to have those programs be executed at the target

In **process** migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well

A process consists of three segments.

Code segment :

The *code segment is the part that contains the set of instructions* that make up the program that is being executed.

Resource segment :

The *resource segment* contains references to external resources needed. by the process, such as files, printers, devices, other processes, and so on.

Execution segment:

an *execution segment is used* to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

Models for Code Migration

Weak mobility:

In this model, it is possible to transfer only the code segment, along with perhaps some initialization data.

A characteristic feature of weak mobility is that a transferred program is always started from one of several predefined starting positions.

This is what happens, for example, with Java applets, which always start execution from the beginning.

The benefit of this approach is its **simplicity**.

Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable.

Models for Code Migration (continue)

Strong mobility:

in systems that support strong mobility the execution segment can be transferred as well.

The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off.

Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Migration and Local Resources

What often makes code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed.

Three types of process-to-resource bindings:

Binding by identifier:

The **strongest binding** is when a process refers to a resource by its identifier.

In that case, the process requires precisely the referenced resource, and nothing else.

An **example** of such a binding by identifier is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address.

Three types of process-to-resource bindings:

Binding by value:

A weaker form of process-to-resource binding is when only the value of a resource is needed.

In that case, the execution of the process would not be affected if another resource would provide that same value.

A typical **example** of binding by value is when a program relies on standard libraries, such as those for programming in C or Java.

Such libraries should always be locally available, but their exact location in the local file system may differ between sites.

Not the specific files, but their content is important for the proper execution of the process.

Three types of process-to-resource bindings:

Binding by types:

The **weakest form** of binding is when a process indicates it needs only a resource of a specific type.

This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

Three types of resource-to-machine bindings

Unattached resources:

They can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated

Fastened resources:

Moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites.

Fixed resources:

Fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices.
Another example of a fixed resource is a local communication end point

Self study

- Migration in Heterogeneous Systems

End of Chapter 3