# Operating System Support

Buch: 10.2.1, Kap. 3

Fragen:

1. **Was ist der Unterschied zwischen Prozess und Thread? Was ist beim Einsatz von Multi-Threading zu beachten? Welche Bedeutung haben Threads in verteilten Systemen, insbesonders in Client/Server-Umgebungen?**

Unter einem *Prozess* versteht man ein Programm, das auf einem der virtuellen Prozessoren des OS ausgeführt wird, welche in einer Prozesstabelle gemanaged werden. Dabei wird unter vergleichsweise hohem Aufwand sichergestellt, dass einzelne Prozesse von anderen Prozessen aufgrund der „geteilten" CPU nicht in ihrer Korrektheit beeinträchtigt werden (*concurrency transparency*). Wird ein neuer Prozess gestartet, muss vom OS ein völlig neuer Adressraum zur Verfügung gestellt werden. Zudem werden beim Wechseln von einem Prozess zum nächsten u.a. der CPU Kontext gespeichert oder Register modifiziert.

Im Unterschied zu Prozessen spielen sich *Threads* auf einer feineren Ebene ab. Ein Prozess kann aus mehreren Threads bestehen, die sich den selben Adressraum teilen. Hierbei muss der Programmierer dafür sorgen, dass ein Thread keinen Schaden/unerwünschte Zustände bei konkurierenden Speicherzugriffen erzeugt. Der Wechsel zwischen Threads ist weniger Aufwändig als bei Prozessen, es muss lediglich der CPU Kontext gewechselt werden. Aus diesem Grund ist die Performance einer Multithread-Applikation kaum schlechter, zumeist sogar besser als bei einem einzigen Thread. Dennoch muss beim Design eines solchen Systems sorgfältig vorgegangen werden, da mehrere Threads nicht automatisch voneinander geschützt sind, wie dies bei Prozessen der Fall ist.

Vorteile von Threads:

- *einfachere Implementierung*: (divide&conquer) - jeder Thread muss nur mehr einen Teil des Gesamtproblems lösen
- *höherer Durchsatz*: blockiert ein Thread, so kann billig aus den nächsten gewechselt werden.
- Es kann *höhere Transparenz* erreicht werden (zB. Verbergen der Latenzzeiten → während auf das Ergebnis des Servers gewartet wird, kann ein anderer Thread weiterarbeiten)

Bei der Verwendung von Threads muss jedoch darauf geachtet werden, dass ein blockierender Thread nicht seinen ganzen Prozess stoppt. Dies kann man mittels *Lightweight Processes (LWP)* erreichen, die nicht wie Threads im User-Space sondern im Kernel Space beheimatet sind und die eine Scheduling Routine für die Threads durchführen.

Für Verteilte Systeme sind Threads von großer Bedeutung, da sie eine Kommunikation zwischen Einheiten durch mehrere gleichzeitige Verbindungen ermöglichen. Ein typisches Beispiel für einen *MT-Client* ist ein Web Browser. Nachdem die HTML-Datei geladen wurde, kümmern sich mehrere Threads um die restlichen zu ladenden Daten wie Grafiken, Applets, usw. und ermöglichen so eine Darstellung am Rechner, obwohl im Hintergrund noch Teile geladen werden. Wenn die Daten der Website auf mehrere Server verteilt (repliziert) sind, ermöglicht dies ein paralleles Laden der Threads, wodurch sich die Ladezeit markant verringert.

Von noch größerer Bedeutung sind Threads auf *Server*-Seite, wo sie einen hohen Grad an Parallelismus ermöglichen. Eine Möglichkeit wäre die Verwendung eines *Dispatcher Threads*, der einkommende Anfragen übernimmt und an einen freien (Worker-) Thread übergibt. Wenn dieser, beispielsweise aufgrund eines System Call wie *blocking read* blockiert, wird vom Dispatcher bei neuen Anfragen einfach ein anderer, Thread ausgewählt. Dies führt im Vergleich zu einer 1-Thread-Lösung zu einer hohen Leistungssteigerung, da mehr Anfragen/Sekunde bearbeitet werden können.

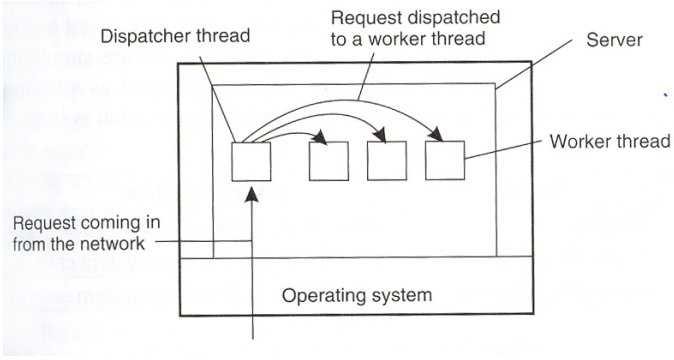
2. **Welche Aspekte Verteilter Systeme sind auf Client-Seite zu berücksichtigen? Wie werden User Interfaces in die Architektur Verteilter Systeme eingebunden? Wie können dabei verschiedene Arten der Transparenz unterstützt werden?**

Clients müssen in verteilten Systemem mit dem User und dem Remote Server parallel kommunizieren. Dies kann durch Multithreading am besten und einfachsten geschehen. Weiters können dadurch zB Latenzzeiten bei der Kommunikation überbrückt werden. Wenn es die Server unterstützen, so kann auch über Load-Balancing nachgedacht werden, so dass sich ein Client von mehreren Server-Replikas die Daten parallel besorgt (siehe auch 1). Auch die lokale Abarbeitung von Daten steigert die Performance, da zB der Netzwerktransfer minimiert werden kann (zB durch Formularkontrolle am Client anstatt am Server).

Neben Benutzeroberfläche und anderer für die Applikation benötigter Software besteht die Client-Software aus Komponenten, mit denen *Verteilungstransparenz* erzielt werden soll. Im Idealfall sollte ein Client nicht erkennen, dass er mit einem entfernten Prozess kommuniziert.
Die *Zugriffstranzperenz* wird häufig realisiert, indem aus der vom Server gebotenen Schnittstellendefinition ein Client-Stub erzeugt wird. Der Stub bietet dieselbe Schnittstelle, die auch auf dem Server zur Verfügung steht, verbirgt jedoch die möglichen Unterschiede in Hinblick auf die Maschinenarchitektur, sowie die eigentliche Kommuniktion.
Es gibt unterschiedliche Möglichkeiten, *Orts-, Migrations- und Relokationstransparenz* zu realisieren. Die Verwendung eines praktischen Namenssystem ist dabei wesentlich. In vielen Fällen ist auch die Zusammenarbeit mit clientseitiger Software wichtig. Hat beispielweise ein Client bereits ein Bind zu einem Server, kann der Client direkt informiert werden wenn sich die Position des Servers ändert. In diesem Fall kann die Middleware des Clients die aktuelle Position des Servers vor dem Benutzer verbergen und die erneute Bindung zu dem Server ggf. transparent vornehmen.
Auf ähnliche Weise, mit Hilfe clientseitiger Lösungen, wird von vielen Verteilten Systemen die *Replikationstransparenz* implementiert.
Die Maskierung von Kommunikationsfehlern mit einem Server erfolgt normalerweise über Client-Middleware. Beispielsweise kann eine Client-Middleware so konfiguriert werden, dass sie wiederholt versucht, eine Verbindung mit einem Server aufzunehmen, oder nach mehreren Versuchen einen anderen Server kontaktiert *(Fehlertransparenz).*

3. **Geben Sie grundlegende Design-Entscheidungen für Server an und bewerten Sie diese. Gehen Sie auf den Unterschied zwischen stateful und stateless Servern genauer ein und geben Sie Beispiele an. Erläutern Sie anhand einer Skizze Architektur und Funktionsweise eines multi-threaded Servers (zB File- oder Web-Server). Was ist beim Einsatz von Multi-Threading vom Entwickler besonders zu beachten?**

Zunächst ist zwischen einem *iterativen* Server, der eine Anfrage selbst behandelt und, wenn nötig eine Antwort an den Client schickt, sowie einem *Concurrent* Server, der die Anfrage an einen Dispatcher Thread weiterleitet (bzw. einen neuen Prozess startet) und auf die nächste Anfrage wartet, zu unterscheiden.




Weiters stellt sich die Frage, wie der Server kontaktiert werden kann. Grundsätzlich handelt es sich bei der *Schnittstelle* um einen Endpunkt (Port) auf der Maschine des Servers. Dieser ist entweder als „well-known“ Port dem Client automatisch bekannt (zB HTML, FTP, etc.), oder muss vom Client, etwa über einen am Server laufenden Daemon (hängt selbst an einem WKP) identifiziert werden. Von Bedeutung im Design ist auch, wie *Interrupts,* etwa im Fall eines Download-Abbruchs seitens des Client von einem FTP-Server, behandelt werden. Dies kann entweder durch clientseitiges Schließen der Anwendung (Server beendet ebenfalls die Verbindung) oder mittles *out-of-band Daten* geschehen, die einen Interrupt beim Server auslösen.

Ein anderes Kriterium ist, ob die Server Informationen über deren Clients persistent speichern. Im Fall von *stateless* Servern (zB Web-Server, der lediglich HTML-Anfragen bearbeitet) ist dies zumeist nicht der Fall, wogegen *stateful* Server wie etwa File-Server mit Schreibrechten für bestimmte User eine User-Tabelle führen müssen. Der Vorteil gegenüber stateless Servern liegt hier in einer höheren Performance, die allerdings mit dem großen Aufwand einer Wiederherstellung im Fall eines Server-Crashes erkauft wird. Manche stateless Server unterstützen einen sogenannten *Soft State*, wo Information für einen vom Client bestimmten Zeitraum gespeichert wird. Dies ist dann der Fall, wenn zB ein Client vom File-Server für kurze Zeit über Updates informiert werden will. Weiters: *session state/permanent state* (siehe Dependability and Fault Tolerance).

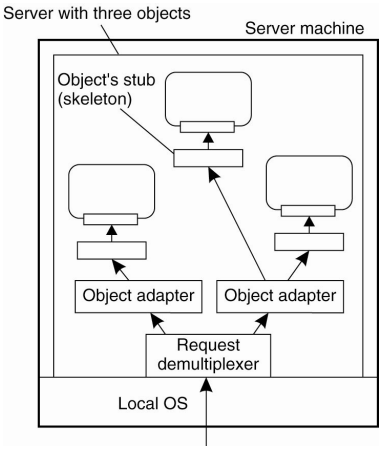
Beim *Einsatz von Multi-Threaded Servern* ist vom Entwickler besonders Wert auf die Nebenläufigkeit zu legen. Problem: Threads greifen alle auf den selben Speicherbereich des Prozesses zu und können so gegenseitig Werte überschreiben, was zu Inkonsistenzen führen kann. Auch muss bei der Verwendung von

Threads darauf geachtet werden, dass ein blockierender Thread nicht seinen ganzen Prozess stoppt. Dies kann man mittels *Lightweight Processes (LWP)* erreichen, die nicht wie Threads im User-Space sondern im Kernel Space beheimatet sind und die eine Scheduling Routine für die Threads durchführen.

4. **Was sind die Besonderheiten von Objekt-Servern? Welche Arten gibt es dabei für die "Invocation", also den Aufruf (evtl. auch die Aktivierung/Activation) eines Objektes auf Server-Seite (Policies hinsichtlich thread, code sharing, und object creation)? Was ist in diesem Zusammenhang ein Objekt-Adapter?**

Aufgabe der Objekt-Server ist das Hosten von verteilten Objekten. Im Unterschied zu gewöhnlichen Servern bieten sie keine eigentlichen Services an, sondern dienen als Schnittstelle für den Aufruf von lokalen Objekten. Services können somit einfach durch den Austausch von Objekten geändert werden. Der Aufruf einzelner Objekte wird durch verschiedener *Policies* (Richtlinien) geregelt, die sich u.a. darauf beziehen, ob der Server lediglich einen einzigen *Thread* (nicht sofort bearbeitbare Anfragen kommen in eine Warteschlange), oder einen für jedes Objekt führt, wobei hier, wie in 3) besprochen, auf Concurrency geachtet werden muss. Auch die Speicherverwaltung ist Thema der Policies, d.h. ob jedes Objekt ein eigenes Speichersegment erhält und somit weder Code noch Daten mit anderen Objekten teilt, oder Codesharing betrieben wird. Im Fall einer Datenbank könnte so ein einziges Objekt von allen Anfragen verwendet werden, um auf die Daten zuzugreifen. Zuguterletzt stehen verschiedene Arten der Aktivierung von Objekten zur Verfügung: Sie können beim Start des Servers (alle zugleich) erstellt werden, oder beim ersten Aufruf (bleiben danach bis zur Beendigung des Servers bestehen bzw. werden gleich nach Ende der Anfrage zerstört).

Ein *Objekt Adapter* ist eine Software, die Mechanismen anbietet, um Objekte aufgrund der zuvor genannten Activation Policies zu gruppieren. Am Server können nun je nach Zahl der Richtlinien mehrere Objekt Adapter bereit stehen, wobei eine hereinkommende Anfrage dem passenden Adapter zugewiesen wird. Dabei sind ihnen die Interfaces der Objekte, die sie kontrollieren, nicht bekannt – sie sind somit generisch und müssen lediglich für eine gewünschte Policy (zur Laufzeit) konfiguriert werden. Der eigentliche Aufruf der Objekte vom Adapter erfolgt schließlich über deren Server Stubs (Skeletons).




5. **Erläutern Sie die wichtigsten Aspekte der Code Migration. Erklären Sie "strong mobility" und "weak mobility" und geben Sie für "weak mobility" ein Beispiel an.**

Code Migration, sei sie *sender- oder receiver-initiated (Java)*, ermöglicht eine Weitergabe von Programmteilen auch im laufenden Zustand. Zwei wichtige Gründe dafür sind eine verbesserte *Performance* sowie erhöhte *Flexibilität*. Im Fall von „teurer“ Kommunikation ist es günstig, einige Berechnungen vom Server zum Client zu verlagern und lediglich das Ergebnis zu übertragen (zB Formular-Eingaben). Generell ist es günstiger, Berechnungen in der Nähe der Daten durchzuführen. Flexibilität wird erhöht, wenn etwa ein Client für die Kommunikation mit einem speziellen Server eigene Software benötigt. Im Fall von Code Migration kann er diese dynamisch, also bei Bedarf, direkt vom Server beziehen und muss sie folglich nicht vorinstalliert haben. Allerdings hat Code Migration auch nachteilige Auswirkungen v.a. auf die Sicherheit (siehe Security).

Hinsichtlich des Umfangs der Datenmigration wird zwischen strong mobility und weak mobility unterschieden. *Weak mobility* ermöglicht den Transfer des Code Segments mit den zur Ausführung benötigten Programmteilen sowie von Daten zur Initialisierung. Es ist hier nur möglich das Programm an einer vordefinierten Position zu starten, wie zB bei *Java Applets* (Programm wird im Adressraum des Browsers ausgeführt) die immer beim Beginn gestartet werden. Der Vorteil dieser Methode ist die

Einfachheit, die einzige Anforderung ist, dass der Code vom Client ausgeführt werden kann. Strong mobility erlaubt dagegen auch den Austausch des Ausführungs-Segments, in welchem der aktuelle Programmstatus (private Daten, Stack, Program-Counter) abgespeichert wird. Charakteristisch ist, dass der laufende Prozess angehalten, auf eine andere Maschine übertragen, und dort fortgesetzt werden kann. Nachteilig ist hier die schwierige Implementierung.

6. **Erläutern Sie das Konzept der Virtualisierung und in weiterer Folge deren Bedeutung für die Code Migration in heterogenen Umgebungen. Beschreiben Sie die zwei verschiedenen Arten von Architekturen von "virtual machines".**

Threads und Prozesse können als eine Möglichkeit gesehen werden, mehrere Dinge zur selben Zeit zu erledigen. Auf einem Computer mit einem einzigen Prozessor ist diese Gleichzeitigkeit natürlich eine Illusion, die durch schnelles Umschalten zwischen beiden Prozessen erreicht wird (*resource virtualization*).

Im Fall von heterogenen Netzwerken, wo eine große Anzahl von Servern den Clients unterschiedliche Applikationen anbietet, kann Virtualisierung helfen, die Vielzahl an unterschiedlichen Plattformen zu reduzieren. Dazu lässt man alle Anwendungen auf eigenen virtuellen Maschinen laufen – u.U. Inklusive eigenem Betriebssystem und Bibliotheken – die ihrerseits auf einer einzigen Plattform zusammengefasst werden können. Auf ähnliche Weise ist es mittels Virtualisierung möglich, *Legacy Systeme* auf neuen Plattformen zu betreiben. Ein weiteres Argument für Virtualisierung liegt in der *Portabilität*. So können etwa Content Delivery Networks ganze Seiten inklusive ihrer Umgebungen replizieren.

Ziel der Virtualisierung ist es, die unterschiedlichen Interfaces eines Computer Systems (jeweils zwischen Anwendung, Bibliotheken, OS und Hardware) nachzuahmen. Eine Möglichkeit ist, mittels einer *Process Virtual Machine* einen einzigen Prozess zu unterstützen. Dabei stellt ein eigenes Runtime System ein Set an Instruktionen zur Verfügung, womit Anwendungen ausgeführt werden. Eine andere Variante wäre, ein System einzuführen, das als eine Schicht zwischen OS und Hardware implementiert wird. Der *Virtual Machine Monitor (VMM)* kann mehrere Programme bzw. verschiedene Betriebssysteme zur gleichen Zeit bedienen und sorgt durch seine Integration als Schicht über der Hardware dafür, dass Software unabhängig von der Hardware betrieben und auf diese Weise eine komplette Umgebung von einer Maschine auf die andere portiert werden kann.

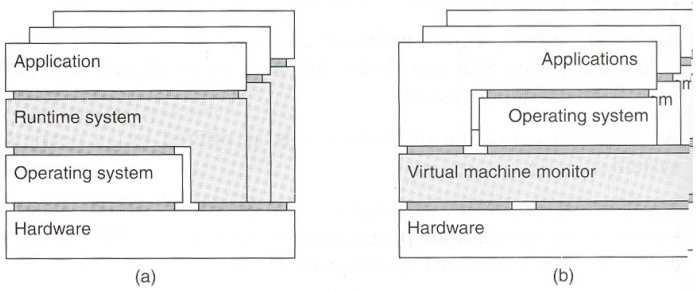



**Figure 3-7.** (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor, with multiple in-

Durch Virtualisierung bekommt man zusätzlich jene Probleme mit Heterogenität in Griff, die in *Zusammenhang mit Code Migration* zwischen unterschiedlichen Plattformen auftauchen. Eine Lösung ist der bereits angesprochene VMM, der ein Verschieben von Prozessen, zusammen mit deren OS, erlaubt. Die alternative Variante wird von Java gewählt, indem statt Maschinencode ein plattform-unabhängiger Code erzeugt wird, welcher wiederum von einer plattform-abhängigen Java Virtual Machine (JVM) interpretiert wird.