Kapitel 4: Tasks, Prozesse, Threads

- Motivation und Einführung
- Prozess, Task
- Thread Modelle
- Thread Typen
- Threadkontrolle
- Implementierung von Threads
- Ausführung von Threads

4.1 Motivation und Einführung

Der ursprünglich eingeführte Begriff des Prozesses (vom Lateinischen *procedere* = voranschreiten abgeleitet) beinhaltete in erster Linie nur eine Abstraktion für die Ausführung von Programmen, also beispielsweise eine Anwenderaktivität (*application*), aber auch eine außerhalb des Systemkerns implementierte Systemaktivität wie z.B. den Kommadointerpretierer (*shell*) in Unix ähnlichen Systemen. Solche Aktivitätseinheiten benötigen zu ihrer Ausführung gewisse Systemressourcen, die sie entweder vom Systemkern auf Anfrage (*via system call*) oder von spezifischen Dienstgeben (*servern*) zugeteilt bekommen. Bei genügend großem Ressourcenvorrat kann man solche Aktivitäten nebenläufig oder in SMP-Systemen u.U. sogar auch echt parallel ausführen lassen.

Seit den Tagen des ersten strukturierten Mehrprogrammsystems Multics verbindet man mit einem Prozess jedoch auch noch eine geschützte Programmausführungsumgebung, den so genannten Prozessadressraum. Wenn wir als Analogie aus dem täglichen Leben eine Aktivität mit einer Person gleichsetzen, dann könnte einem Adressraum der Begriff Haushalt oder Wohnung entsprechen, somit stellt also ein Prozess einen Singlehaushalt dar. Wie wir wissen, gibt es zumindest im täglichen Leben aber auch Familienhaushalte oder Wohngemeinhaften, nur welchen Nutzen verspricht man sich davon? Zumindest eine Statistik gibt darauf eine einleuchtende Antwort: Im Mittel wirtschaften Singlehaushalte erheblich ungünstiger.

Im übertragenen Sinne müssen wir also nach den Kosten bei der Abwicklung von Prozessen fragen und sie mit den entsprechenden Kosten einer mehrfädigen (*multi threaded*) Task vergleichen. Entstehen beispielsweise Kosten, wenn man einen neuen Adressraum anlegt? Offensichtlich, denn pro Adressraum müssen wir im Kern einen Verwaltungsblock anlegen, der uns angibt, aus welchen Bestandteilen dieser Adressraum besteht und wie groß die einzelnen Adressraumabschnitte sind. Ferner müssen wir vermerken, an welchen Adressen diese Adressraumabschnitte (*regions*) jeweils beginnen usw. Diese Art "Mietvertrag" müssen wir bei einer mehrfädigen Task jedoch nur einmal ausfüllen, da sich die verschiedenen Taskaktivitäten, eben die Threads ja ihre Wohnung, sprich den Adressraum teilen.

Ferner ist die Interaktionsmöglichkeit innerhalb einer Task erheblich einfacher, da sich diese Threads "mühelos" über gemeinsame Daten verständigen können (vergleichbar mit dem Gemeinschaftsraum in einer WG). Singles hingegen müssen ihre Informationen via Telefon miteinander austauschen oder sich gegenseitig besuchen, der IPC zwischen Adressräumen ist in der Regel wesentlich teurer als der Zugriff auf gemeinsame Daten. Man kann zwar eine parallelisierbare Anwendung auch mit mehreren nebenläufigen Prozessen implementieren, gleichwohl ist deren Implementierung mittels einer mehrfädigen Task nicht nur billiger, sondern in der Regel auch raum- und laufzeiteffizienter. Bei einer IPC zwischen *Prozessen* ist immer ein Prozesswechsel verknüpft, der auf den meisten Systemen um Größenordnungen *langsamer und aufwendiger* als ein Threadwechsel zwischen Threads der gleichen Task ist.

Ein Betriebssystem selbst könnte auch davon profitieren, dass bestimmte Systemkomponenten als getrennte Aktivitäten, aber eben doch im gleichen Adressraum implementiert werden. Der Ruf nach leichter handhabbaren und effizienter zu implementierenden nebenläufigen bzw. parallelen Aktivitätseinheiten kam aber zweifellos von solchen Anwendern, deren zu implementierende Applikationen per se parallelisierbar gestaltet werden konnten. Für solche "leichtgewichtigeren" Aktivitätseinheiten hat sich der Begriff Thread (Faden) eingebürgert. Auf Applikationsebene unterscheidet man somit *single-threaded* (einfädige) Prozesse von *multi-threaded* Tasks, d.h. alle Threads einer Task laufen im gleichen Adressraum.

Nota bene: Dieser feine Unterschied zwischen Prozessen und Tasks ist KA spezifisch.

Fragen:

1. Mit welchen typischen Anwenderprozessen bzw. -tasks haben Sie schon gearbeitet?
2. Welche Ihrer bisherigen Programme haben Sie als Tasks, welche als Prozesse konzipiert?
3. Kann man in Java nur Tasks oder auch Prozesse implementieren?
4. Wie lange dauert ein Prozesswechsel auf ihrem Linux- bzw. Windowsrechner und wie lange dauert ein Threadwechsel zwischen Threads der gleichen Task? Wie könnte man dies messen?

4.2 Entwurfsparameter für Aktivitäten

Ein Systemarchitekt kann sich an folgenden Entwurfsparametern orientieren, wenn er Anwender- oder Systemprogrammierern Konzepte für die Implementierung ihrer Anwender- oder Usermode-System-aktivitäten anbieten möchte:

Anzahl von Aktivitäten

- Statisch (in manchen eingebetteten Systemen ist schon zum Entwurfs-, spätestens aber zum Generierungszeitpunkt bekannt, wie viele und welche Aktivitäten im System als Prozesse oder Tasks ausgeführt werden)
- Dynamisch (in diesen Systemen werden zur Laufzeit -meist zu nicht vorhersehbaren Zeiten- neue Aktivitäten erzeugt und u.U. auch bisherige Aktivitäten wieder entfernt.

Frage: In vielen dynamischen Systemen sehen die Systemarchitekten eine Oberzahl an gleichzeitig im System erzeugten Aktivitäten vor. Welche Gründe können dafür sprechen?

Aktivitätstypen

- Vordergrundaktivität: Diese Begrifflichkeit bzw. der konträre Begriff Hintergrundprozess tauchte zum ersten Mal im Zusammenhang mit einem der ersten Timesharing-Systeme CTSS (am MIT entwickelt und implementiert) auf. Einen bequemen Umgang mit Vordergrundprozessen kann man sich mittels Fenstersystemen verschaffen, der aktuelle Vordergrundprozess kann im aktuellen Dialogfenster beobachtet und gegebenenfalls mittels der GUI oder eines Kommandointerpretierers direkt beeinflusst, z.B. abgebrochen werden.
- Hintergrundaktivität: Man kann hierbei zwei verschiedene Klassen von Hintergrundprozessen unterscheiden. Zum einen Anwenderaufgaben, die der Benutzer zwar im aktuellen Dialogfenster startet, diese aber in den Hintergrund verweist, wo sie i.d.R. nur dann aktiviert werden, wenn keine sonstigen Vordergrundaktivitäten durchzuführen sind. Beispielsweise könnte man bei einem interaktiven Computerspiel während der Denkpausen des Spielers langfristige, numerische Kalkulationen im Hintergrund, quasi als Lückenfüller ablaufen lassen. Die andere Klasse von Hintergrundprozessen bilden die so genannten Dämonenprozesse (*daemons*). Ein Dämonprozess wird i.d.R. zum Urladezeitpunkt eingerichtet und steht unter keiner direkten Kontrolle durch einen Benutzer. In Unix herrschen Dämonen vor, deren Namen auf d enden, wie z.B. `syslogd`, der die Systemlogdatei verwaltet. (Im übrigen waren Dämonen in der griechischen Mythologie Wesen, die Dinge bewirken konnten, die außerhalb der Fähigkeit der Götter lagen.)

Frage: Welche Linux- bzw. Unixdämonen kennen Sie bereits?
Frage: Welches Prozessattribut ist in allen Unixdämonen gleich?

Dringlichkeit von Aktivitäten: Die Dringlichkeit einer Anwender- oder einer Systemaktivität hängt wesentlich davon ab, was diese Aktivität nach außen oder innen bewirken soll. Nicht alle Systemaktivitäten sind per se dringlicher als jede Anwenderaktivität, man denke beispielsweise an die Dämonenprozesse von Unix. Innerhalb der Anwenderaktivitäten wird häufig folgende grobe Einteilung nach Dringlichkeitsklassen vorgenommen (mit fallender Dringlichkeit angeordnet):

- Echtzeit (*real time*)
- Interaktiv
- Stapelverarbeitung

Benötigte physische Betriebsmittel pro Aktivität

- CPU
- Hauptspeicher
- Periphere Ein-/Ausgabegeräte

Frage: Welche logischen Betriebsmittel (logical resources) kennen Sie?

Zusammenfassung: In den vielen heute gängigen Systemen werden Betriebsmittel an einen Prozess bzw. an eine Task vergeben, im letzteren Fall zur gemeinsamen Nutzung aller Threads, die zu dieser Task gehören. Aber wir werden auch Beispiele kennen lernen, in denen Betriebsmittel individuell an Threads vergeben werden.

4.3 Prozesszustände

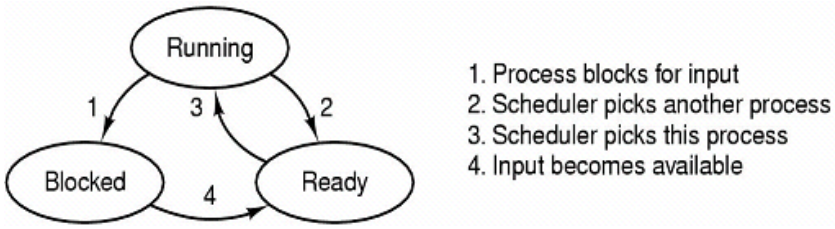
Prozesse können im Verlaufe ihrer Abarbeitung in verschiedene Zustände geraten, aus denen sie nur durch ein Eingreifen des Betriebssystems (meistens des Systemkerns) befreit werden können. In vielen Systemen wird insbesondere das unten abgebildete Dreizustandmodell mit den angedeuteten Zustandübergängen anzutreffen sein.

Im **rechnenden** Zustand (*running*) befindet sich ein Prozess genau dann, wenn ein Prozessor (CPU) ihn gerade ausführt, präziser gesagt, wenn der Inhalt des Befehlszeigerregisters (IP) einer Codeadresse des Prozesses entspricht. (Wir werden später allerdings auch sehen, wenn wir etwas detaillierter auf den Prozesswechsel schauen werden, dass ein Prozess auch dann noch im rechnenden Zustand sein kann, wenn das IP-Register auf einen Befehl im Systemkerncode zeigt.)

Im Zustand **bereit** (*ready*) wird sich ein Prozess genau dann befinden, wenn dem Prozess zu seiner Ausführung auf dem Zentralprozessor eben nur noch dieses Betriebsmittel Zentralprozessor fehlt.

Im Zustand **blockiert** (*waiting* oder *blocked*) hingegen macht es keinen Sinn, den Prozess weiterhin auf den Prozessor zu belassen bzw. ihn am Wettbewerb um den Prozessor teilnehmen zu lassen, da dem Prozess entweder ein sonstiges Hardware- oder Softwarebetriebsmittel fehlt, z.B. eine Information, die gerade von einem Ein-/Ausgabegerät ins System eingeschleust bzw. aus dem System nach draußen transferiert wird.

Als Analogie aus dem täglichen Leben könnte uns ein Meisterkoch dienen, dem gerade zur Zubereitung eines scharf anzubratenden *T-Bone-Steak à l'Anglais* ein wichtiges Gewürz fehlt. Er wird seinen Küchenjungen losschicken und solange das T-Bone-Steak beiseite legen, bis er Zugriff auf das bislang fehlende Gewürz hat, ansonsten würde der Gast nur ein Stück Holzkohle serviert bekommen.



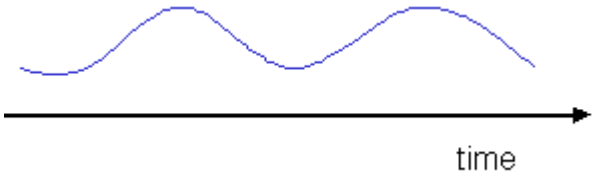
Obwohl obige Abbildung so oder so ähnlich in vielen Textbüchern enthalten ist, kann sie mit den angegebenen Zustandsübergangsfunktionen leicht fehl interpretiert werden. Beim Übergang von rechnend nach blockiert bzw. rechnend nach bereit, sorgt demzufolge der bislang rechnende Prozess durch Aufruf eines direkten bzw. indirekten Systemaufrufs dafür, dass der Kern im Anschluss an die jeweilige Übergangsfunktion einen Prozesswechsel durchführt, so dass somit im Anschluss daran aus der Menge der bereiten

Prozesse ein geeigneter Nachfolgerprozess des zuvor rechnenden Prozesses gefunden wird, der dann der neue rechnende Prozess genannt wird. In obiger Abbildung wird jedoch suggeriert, dass der Ablaufplaner (*scheduler*) eingreift, um so den Übergang des gerade rechnenden Prozesses zurück in die Bereitwarteschlange zu veranlassen. Dieser Vorgang wird allgemein als *Verdrängung* (*preemption*) genannt. In denen von uns betrachteten Ein- und symmetrischen Mehrprozessorsystemen kann keine Planungsinstanz von sich aus tätig werden, sondern es bedarf beispielsweise einer *Zeitscheibenunterbrechung* (*time slice interrupt*), mit deren Hilfe der bislang rechnende Prozess unterbrochen wird, damit dann der Scheduler im Systemkern seine Aktivität im obigen Sinne entfalten kann.

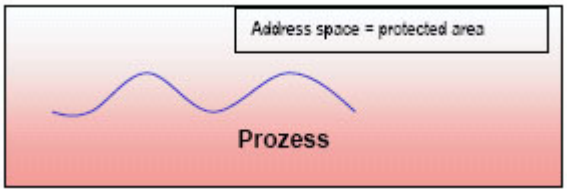
4.4 Thread, Prozess, Task

Da ja ein Thread nicht identisch mit einem Prozess oder einer Task ist, wird er auch durch erheblich weniger Attribute gekennzeichnet sein, siehe später *thread control block* (TCB). Aus der Sicht des Betriebssystems bzw. des Laufzeitsystems ist ein Thread aber zumindest eine

- Aktivitätseinheit und somit ein
- Objekt der Aktivitätsumschaltung (*im konkreten Fall des thread switching*)

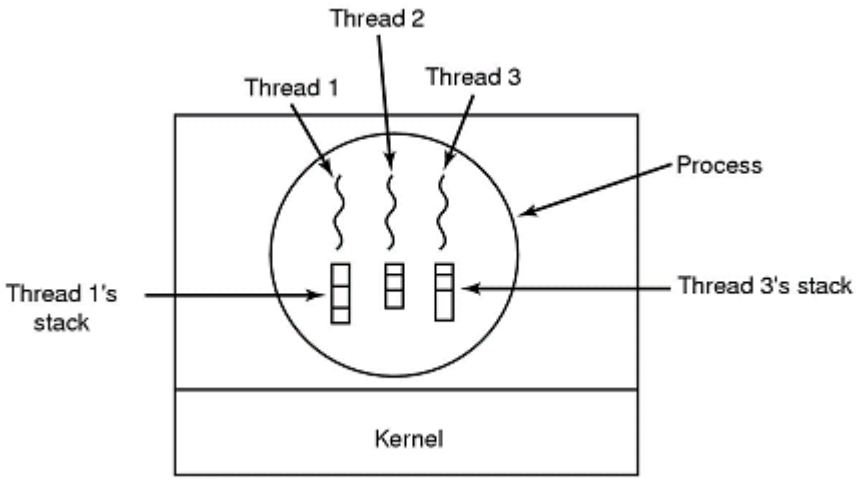


Ein Prozess hingegen ist eine single-threaded Task mit einem Adressraum und diversen zusätzlich benötigten Ressourcen oder Betriebsmitteln. In einer Task können im Gegensatz zum Prozess mehrere Threads definiert und nebenläufig oder im Falle eines SMPs auch parallel ausgeführt werden.



4.4.1 Threadmodelle

Es ist entscheidend für die Abwicklung einer Anwendung, nach welchem Threadmodell deren Threads implementiert worden sind.



In allen Threadmodellen gehen wir davon aus, dass jeder Thread seinen eigenen *Stapel* (*stack*) besitzt, auf dem er seine lokalen Daten ablegen kann. Jeder Anwenderthread benötigt einen ihn umgebenden Adressraum, das kann entweder ein Task- oder Prozessadressraum im Anwendungsbereich sein. (Sollte auch der Kern selbst das Threadmodell für sich anwenden, dann könnten nebenläufige Kernfunktionen auch im Kernadressraum implementiert sein. Im letzteren Fall handelt es sich dann um einen der Systemkernthreads oder kurz *Kernelthreads*.

Mögliche Vorteile von Anwendungen, die als multihreaded Tasks implementiert sind, können sein:

- Verbessertes Antwortzeitverhalten (*responsiveness*)
- Möglichkeit, Ressourcen gemeinsam zu nutzen (*resource sharing*)
- Wirtschaftlichkeit (*economy*)
- Ausnutzen von Mehrprozessorarchitekturen (*utilization of SMPs*)

Bei aller Euphorie über Threads darf man jedoch nicht vergessen, dass es genügend Beispiele für streng sequentielle Anwendungen gibt, in denen ohne weiters auf Threads verzichtet werden kann.

Frage: Machen Sie sich an einfachen Bearbeitungsmodellen klar, warum und unter welchen Randbedingungen es zu den oben genannten vorteilhaften Eigenschaften kommen kann, wenn eine Anwendung als Task und nicht als eine Sammlung von Prozessen realisiert wird.

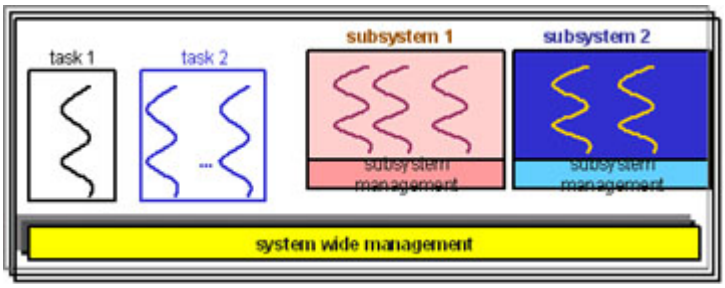
4.4.2 Charakteristische Eigenschaften: Task versus Thread

Welche Attribute (und Ressourcen) sind somit einem Thread bzw. der ihn umgebenden Task (bzw. Prozess) zuzuordnen?

- Objekte bzw. Einheiten, die i.d.R. von allen Threads einer Task gemeinsam genutzt werden:
  - Adressraum (per Defintionem)
  - Globale Variablen (in vielen Threadmodellen)
  - Geöffnete Dateien
  - Kindprozesse
  - Signale und Signalbehandlungen
  - Abrechnungsinformation
  - Taskzustand

- Objekte bzw. Einheiten, die ausschließlich zu einem Thread gehören:
  - Befehlszeiger (*instruction pointer*)
  - Register, flags etc. -> Kontext eines Threads
  - Stapel (*stack*)
  - Threadzustand (im allgemeinen orthogonal zum Taskzustand)

4.4.3 Gültigkeitsbereiche von Threads



Nicht jeder Thread muss im ganzen System bekannt sein, manchmal ist es günstiger, bestimmte Threads nur innerhalb eines Teilsystem bekannt zu geben und deren Kontrolle durch ein lokales Management durch-

zuführen, z.B. werden Threads einer Java-Applikation zumindest teilweise durch die so genannte "Virtuelle Java-Maschine" kontrolliert.

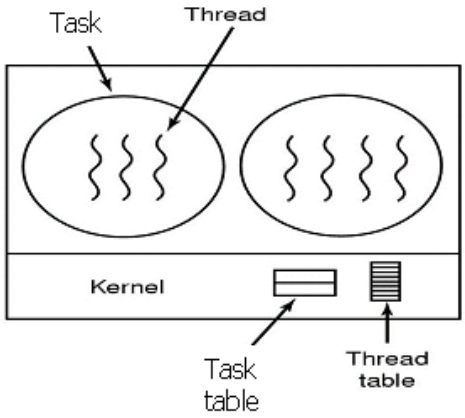
4.4.4 Threadmodelle

Prinzipiell werden drei verschiedene Threadmodelle unterschieden:

- Kernel-Level Threads (KLT)
- Reine User-Level Threads (PULT) und
- Hybride Threads

4.4.4.1 Kernel-Level Threads

Diese Art von Threads ist im gesamten System bekannt, d.h. wenn man dessen Threadnamen bzw. interne Threadkennung (TID = thread identifier) kennt, dann kann man gezielt mit diesem Thread in Verbindung treten, sofern dies aus Schutz- und Sicherheitsgründen erlaubt sein soll. Zu diesem Zweck werden nicht nur die entsprechenden Taskrepräsentanten (TaskCB) einer multi-thraedaed Applikation oder eines multi-threaded Systemserver sondern auch die entsprechenden Threadrepräsentanten, d.h. die TCBs = thread control block im Systemkern angelegt, verwaltet und kontrolliert.



Vorteile von KLTs:

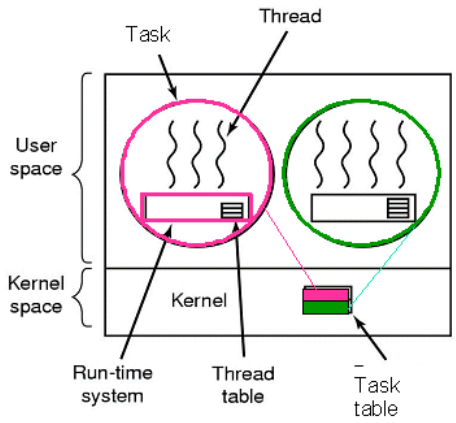
1. Kernel-Level Threads der gleichen Task können echt parallel auf unterschiedlichen Prozessoren eines SMP-System gleichzeitig rechnen.
2. Bei einem blockierenden Systemaufruf (*system call*) wird nur der aufrufende KLT blockiert, andere bereite KLTs der gleichen Task könnten somit auf den freiwerdenden Prozessor zugeordnet werden.

Nachteile von KLTs:

1. Jede Threadoperation ist ein Systemaufruf, erfordert somit zusätzlich einen u.U. aufwendigen Kerneintritt und Kernaustritt.
2. Das Erzeugen eines KLTs erfordert einen zusätzlichen Initialisierungsaufwand (Anlegen eines TCB im Systemkern, allerdings immer noch erheblich weniger als beim Anlegen eines PCBs).
3. Die Threads einer Task werden ausschließlich gemäß der zentralen Planungsstrategie (*global scheduling policy*) vorangetrieben, ob diese Strategie nun gut zur Abwicklungsstrategie der Task passt oder nicht.

4.4.4.2 Reine User-Level Threads (PULTs)

Die reinen User-Level Threads werden häufig im Rahmen eines Laufzeitsystems (Java's Virtual Machine) mittels einer Threadbibliothek verwaltet, d.h. deren Repräsentanten sind nur im entsprechenden Laufzeitsystem als UTCB (*user level thread control blocks*) verankert.



Vorteile von PULTs:

1. Alle Threadoperationen sind auf Benutzerebene angesiedelt, sind somit erheblich schneller. Ferner könnte das Userlevelscheduling an die Bedürfnisse der Taskabwicklung angepasst werden.
2. Das PULT-Modell kann auch auf Betriebssystemen ohne KLT-Threadangebot ausgeführt werden.

Nachteile von PULTs:

1. Gleichzeitig kann jeweils nur ein einziger User-Level Thread der gleichen Task rechnen, da die Prozessorzuteilung Aufgabe des zentralen im Kern realisierten Schedulers ist, der kennt aber nur Tasks, nicht aber deren User-Level Threads.
2. Wenn ein PULT einen blockierenden Systemaufruf durchführt, wird hierdurch die ganze Task blockiert.
3. Der Kernscheduler kann u.U. aus Unkenntnis auch mal eine bereite Task auf den Prozessor zuordnen, obwohl darin gerade nur der Leerlaufthread rechnen könnte, also kein sinnvoller Fortschritt möglich wäre.
4. Der Kernscheduler könnte eine Task verdrängen, von der gerade ein PULT einen u.U. systemweit wirkenden Lock hält, wodurch der gefürchtete Konvoieffekt entsteht.

Im Systemkern ist nur das umgebende Taskobjekt bekannt, also kann auch der zentrale Scheduler im Kern nur die Task als Ganzes umschalten, d.h. es kann somit durchaus zu einer Situation kommen, in der der Scheduler auf Userebene einen bestimmten User-Level Thread gerade rechnend gemacht hat, dass völlig unabhängig dazu aber der zentrale Scheduler eingeschaltet wird und die gesamte Task in den bereiten Zustand zurückversetzt, also den Prozessor einer anderen Task oder einem anderen Prozess weiter gibt. (Warum dies sinnvoll sein kann, werden wir dann im Schedulingkapitel genauer studieren.)

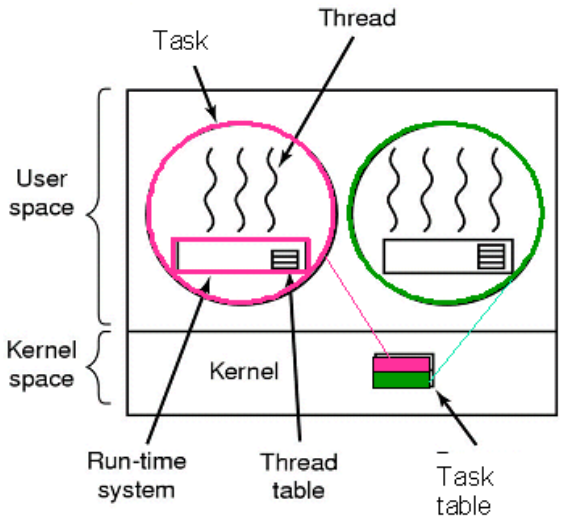
Es gibt nun eine Reihe von günstigen und weniger günstigen Eigenschaften, die mit jedem der beiden bisherigen Threadmodellen verknüpft sind.

Fragen:

1. Was verstehen Sie unter einem Konvoieffekt?
2. Können Sie weitere Unterschiede zwischen den beiden reinen Threadmodellen eruieren?
3. Was passiert eigentlich, wenn in einer multi-threaded Task ein `fork()` gemacht wird? Wird dann die ganze Task repliziert oder nur der aufrufende Thread? In wie weit ist z.B. die Linux-Lösung hier orthogonal?

4.4.4.3 Hybride Threadmodelle

Mit hybriden Threadmodellen versucht man die Vorteile der beiden reinen Threadmodelle zu verbinden, d.h. die unmittelbare Kontrollierbarkeit von speziellen Methoden der Threadbibliothek und die mögliche Eigenständigkeit und Parallelität der Kernel-Level Threads.



4.5 Konkrete Threadimplementierungen

4.5.1 Posix Threads

Studieren Sie die POSIX-Schnittstelle sehr sorgfältig.

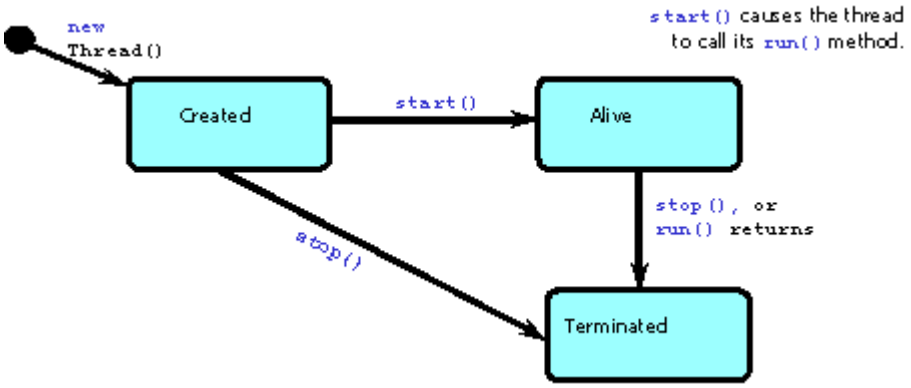
Welches Threadmodell wird angeboten?

Welche Schnittstellenfunktionen werden zu welchem Zweck angeboten?

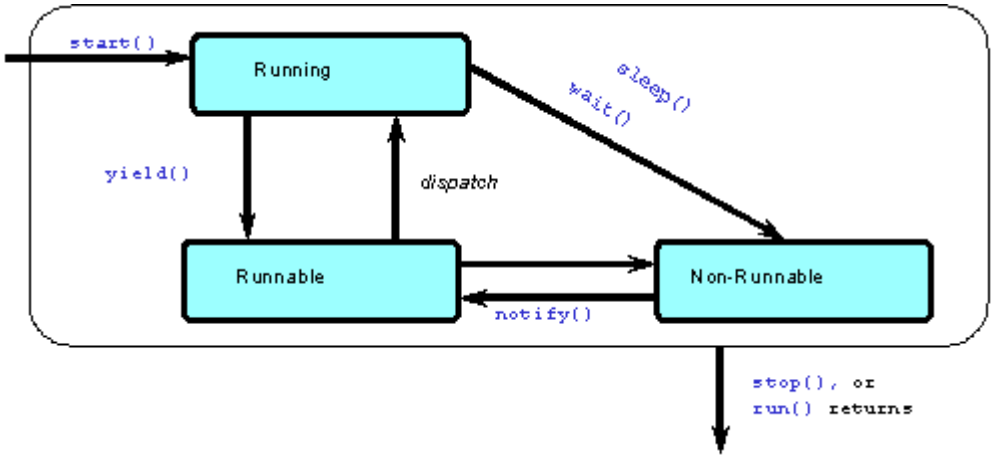
Der Umgang mit Parallelität bzw. Nebenläufigkeit ist zentraler Bestandteil der Systemarchitektur und stellt eine Herausforderung für jeden erfolgreichen Systemarchitekten dar. Dass diese Herausforderung ernst genommen werden muss, können Sie daran studieren, welche katastrophalen Auswirkungen der sorglose bis schlampige Umgang mit der Parallelität in der Vergangenheit gehabt hat. Im Web gibt es mehrere Seiten voll mit Katastrophenmeldungen über eklatante Softwareentwurfs- und/oder -implementierungsfehler allzu sorgloser "Systemlebauer" (siehe http://www.cs.tau.ac.il/~nachumd/horror.html).

Ein persönlicher Hinweis: Programmierer, Informatiker im allgemeinen und Systemarchitekten im besonderen sollten sich gelegentlich auch durchaus mal mit der realen Welt und nicht nur mit virtuellen Welten im Rechner auseinandersetzen, nicht dass noch einmal der aufgehende Mond als eine Schar anfliegender Interkontinentalraketenköpfe interpretiert wird, oder dass das automatische Abschalten eines Düsenaggregats einer Verkehrsmaschine nicht notwendigerweise bedeuten muss, dass der Flieger gerade am Terminal angekommen ist. Wenn dies beispielsweise in 11 000 Metern Höhe passiert, weil ein Aggregat Feuer gefangen hat, dann könnte der arme Pilot und mit ihm alle Passagiere erhebliche Probleme bekommen.

4.5.2 Java Threads



Es gibt ein Zustandsdiagramm für die äußeren Threadzustände und ein detaillierteres für den inneren Zustand "ALIVE", s.u.:



4.6 Threadkontrolle

Nach welchem Threadmodell auch immer nebenläufige oder parallele Aktivitäten implementiert worden sind, benötigen wir in einem System eine Reihe von Kontrollmöglichkeiten, da entweder fälschlicherweise oder sogar mit böswilliger Absicht Anwenderaktivitäten das System stark beeinträchtigen und bei fehlender Systemrobustheit sogar zum Absturz zwingen können:

Auf Grund von Programmierfehlern kann es vorkommen, dass eine Anwendung in eine nutzlose Dauerschleife gerät, in der CPU-Kapazität sinnlos vergeudet wird. Oder ein böswilliger Programmierer versucht mit ständigem Starten von neuen Aktivitäten, das System ganz lahm zu legen, indem er beispielsweise den Vorrat an Threadkontrollblöcken (TCBs) aufbraucht. (Dies war u.a. eine ganz beliebte Dienstgüteattacke (*denial of service attack*) auf ältere Linux-Systeme, die unwiderruflich jedes Mal zum totalen Systemabsturz führte.)

Zusätzlich benötigen wir noch Kontrollen, wenn Aktivitäten an Ressourcen oder an gemeinsamen Daten (oder Dateien) zusammenarbeiten. Darüber hinaus müssen wir einen besonders robusten Mechanismus bereit stellen, wie wir auf einem Einprozessorsystem mit hunderten von potentiell parallelen Aktivitäten umgehen wollen, d.h. wir müssen dafür sorgen, dass zwischen Threads, Prozessen und Tasks folgerichtig, d.h. nach einer möglichst effektiven Schedulingstrategie möglichst effizient umgeschaltet werden kann.

4.7 Thread Repräsentation

Im Threadkontrollblock (TCB) werden alle wesentlichen Attribute zusammengefasst, die den Thread charakterisieren und dessen Kontrolle ermöglichen und damit dessen Abwicklung erleichtern. Umfang und Struktur der TCBs werden von System zu System verschieden sein, da u.U. ganz unterschiedliche Kontrollmöglichkeiten angeboten werden müssen. In einem Echtzeitsystem muss beispielsweise zusätzlich das Threadattribut "Sollzeitpunkt" (*deadline*) im TCB verankert sein, ansonsten könnte der Real-Time-Scheduler keine korrekten Entscheidungen treffen. In jedem System muss ein TCB zumindest aus folgenden Attributen bestehen:

Thread Identifier (TID)
Instruction Pointer (SP)
Stack Pointer (SP)
Status Flags (SF)

4.8 Implementierung der Menge der TCBs

In der Regel werden TCBs von Kernel-Level-Threads im Kern bzw. im Mikrokern implementiert, während die UTCBs in der Regel im entsprechenden Subsystem implementiert werden.

Will man Zugriff auf die Menge aller TCBs gewinnen, um sich z.B. einen Überblick über die aktuellen Kernel-Level Threads zu verschaffen, dann kann man diese TCBs entweder in einem Vektor (array) zusammenhängend abspeichern oder mittels einer geeigneten Datenstruktur miteinander verknüpfen, die erste Realisierungsvariante ist eher in statischen Systemen, die zweite Variante eher bei den dynamischen Systemen anzutreffen. Viele Threadoperationen arbeiten mit der TID als Parameter, wenn hierdurch ein ganz bestimmter Thread (z.B. bei einer IPC) angesprochen werden soll. Andererseits benötigt man für viele Threadoperationen auch Attribute der TCBs, so dass es offensichtlich eine Abbildung von TIDs auf TCB-Anfangsadressen geben muss.

Frage: Welche Datenstruktur ist für diese Abbildung von TID auf TCB-Anfangsadressen geeignet?