Kapitel 5: Threadwechsel (Thread Switch), Threadzustände (Thread States)

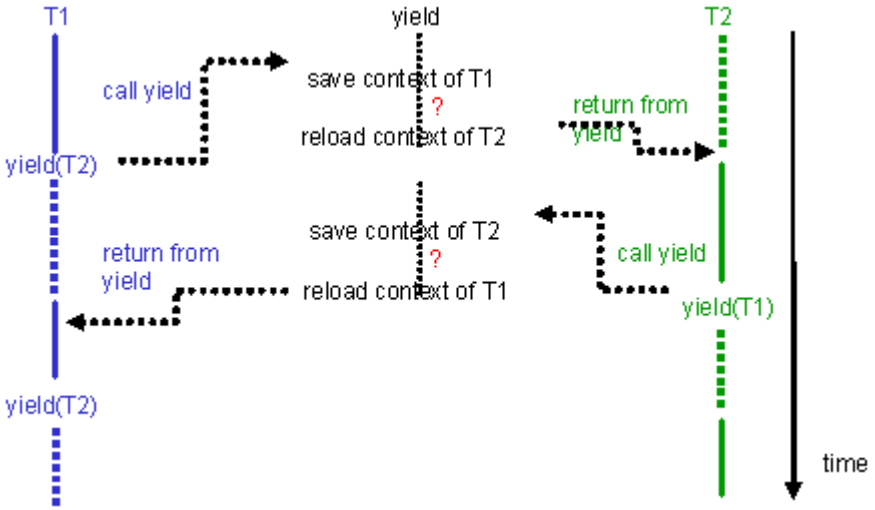
- Motivation und Einführung
- Kooperatives Scheduling
- Realisierung des freiwilligen Umschaltens mittels yield()
- Ereignisse zum Threadwechsel
- Vereinfachter Threadwechsel
- Nutzer- und Kernstapel

5.1 Motivation und Einführung

Wir werden eine ganze Reihe von Ereignissen kennen lernen, weswegen man von dem gerade rechnenden Kernel-Level Thread auf einen anderen Thread umschalten kann oder eventuell sogar sogar muss. Im Falle des so genannten "kooperativen Schedulings" gibt dabei der gerade rechnende Kernel-Level Thread einer Task T_i freiwillig den Prozessor zugunsten eines anderen Kernel-Level Threads der gleichen Task T_i (oder einer anderen Task T_j) auf. Bei User-Level Thread kann es auch dieses kooperative Verhalten geben, da kann ein reiner User-Level Thread (PULT) der Task T_i aber nur einem anderen reinen User-Level Thread der gleichen Task T_i den Prozessor weitergeben.

Andere Ereignisse, die einen Threadwechsel bedingen, werden wir später studieren.

5.1.1 Vereinfachter Threadwechsel von PULTs



Zu klären bleibt nur noch, was genau an den ?-Stellen innerhalb der Prozedur `yield()` passieren muss, wenn `yield()` beispielsweise als Prozedur (oder als Makro) realisiert ist.

```
procedure yield(NT:thread)
begin
...
save context of CT
CT.sp := SP; SP := NT.sp;
load context of NT
...
return to NT
end
```

Analyse: Wenn `yield()` als Prozedur realisiert ist, dann wird stets an der gleichen Programmstelle die Umschaltung vorgenommen, d.h. wir müssen den Befehlszeiger (IP) überhaupt nicht explizit retten und wieder neu laden. Der Call- bzw. Return-Mechanismus der Prozedur sorgt dafür, dass wir nach der Prozedur im neuen Thread weitermachen können. Innerhalb von `yield()` muss allerdings der Stapel(stack) ausgetauscht werden.

Wie funktionsfähig ist ein System, in dem ausschließlich `yield()` zum Threadwechsel angeboten wird?

A) Es gebe nur das reine User-Level Threadmodell, das `yield()` anbietet, dann gilt Folgendes:

- Der Systemkern kennt keine User-Level Threads, somit könnte er nie zwischen Tasks hin- und herschalten (*Kernel is not aware of existence of user level threads*), es muss also zumindest im Kern eine Art Task bzw, Prozesswechsel geben.
- Das Umschalten der User-Level Threads erfordert keinen Wechsel von Anwendermodus in den Kern und wieder zurück.
- Die Schedulingstrategie kann anwenderspezifisch und damit effizient implementiert werden.
- Alle Programmierer einer multi-threaded Anwendung müssen sehr kooperativ sein.

B) Neben dem `yield()` der Threadbibliothek gäbe es auch an der Kernschnittstelle einen `yield()` Systemaufruf mit dessen Hilfe ein KLT bzw. ein Prozess den Prozessor zugunsten anderer im Kern bekannter Aktivitäten abgeben kann.

- Was soll ein Systemaufruf `yield()`, aufgerufen von einem reinen User-Level Thread einer multi-threaded Anwendung bewirken? Macht dies Sinn? Es kann sein, dass neben der multi-threaded Anwendung weitere Anwendungen zusammenarbeiten sollen, so dass es u.U. in diesen Fällen Sinn macht, die aktuelle multi-threaded Task als Ganzes zugunsten einer anderen Task oder eines anderen Prozesses zurückzustellen.

Zusammenfassung: `yield()` wird vorzugsweise innerhalb einer multi-threaded Anwendung eingesetzt, reicht somit per se in allgemeinen Systemen nicht aus, alle Umschaltprobleme zu lösen.

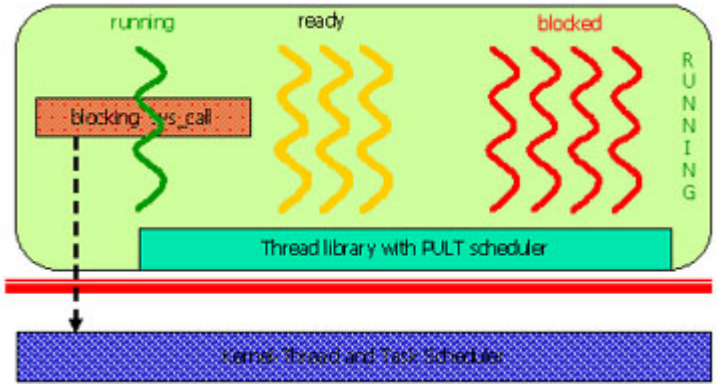
5.1.2 Typische Threadbibliotheksfunktionen

- creating and destroying threads
- passing messages between threads
- scheduling thread execution
- synchronizing each other
- saving and restoring thread context

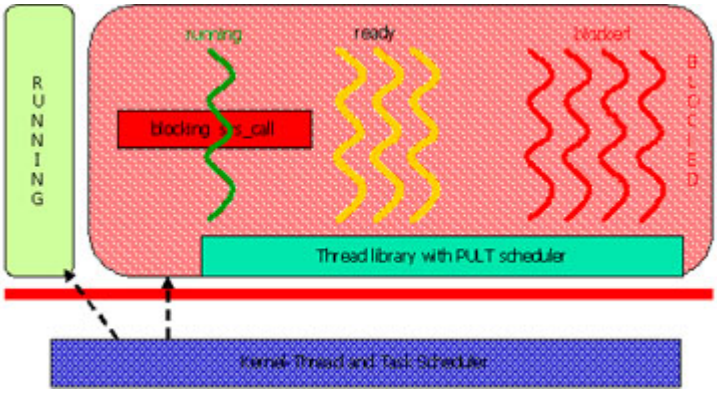
5.1.3 Kernaktivitäten im Umgang mit reinen User-Level Threads

Sobald ein reiner User-Level Thread einen blockierenden Systemaufruf (*system call*) durchführt, wobei dann der "PULT" auch tatsächlich blockiert werden müsste, muss der Kern die ganze Task blockieren, da er ja keinen einzigen reinen User-Level Thread kennt, wohl aber die ihn umgebende Task. Diese Tatsache hat natürlich weitreichende Konsequenzen, so kann es z.B. auch niemals vorkommen, dass gleichzeitig zwei reine User-Level Threads der gleichen Task auf verschiedenen Prozessoren rechnen können.

Was passiert aber nun mit einem User-Level Thread, wenn er einen blockierenden Systemaufruf, z.B. "disk output" initiiert? Die folgenden Bilder geben einen Einblick in diesen Sachverhalt, wobei anzumerken ist, dass man die Threadzustände der reinen User-Level Threads völlig unabhängig von den Zuständen der umgebenden Task zu betrachten hat.



Da die Threadbibliothek nicht zuständig ist für die Bearbeitung des Systemaufrufs, wird der Kern eingeschaltet. Der Kern wiederum blockiert nun die ganze Task, wobei aus der "eingeschränkten" Sicht des PULT-Schedulers der User-Level Threadzustand des PULT nach wie vor rechnend ist, denn dieser PULT-Scheduler ist ja überhaupt nicht in die Bearbeitung des Systemaufrufs involviert. Somit ergibt sich folgender Systemzwischenzustand:



Irgendwann wird das Ereignis eintreffen, weswegen es zu einer Blockierung der obigen multi-threaded Task gekommen ist. Es bleibt dann dem Kernscheduler überlassen, welche PULT-Task bzw. welcher Kernel-Level Thread als nächstes weiterrechnen soll.

In folgender Tabelle sind noch einmal die Vor- und Nachteile von reinen User-Level Threads (PULTs) zusammengefasst:

Vorteile:

- Ein Thread_switch kostet weniger Aufwand, da er ohne *Moduswechsel* stattfinden kann.
- Man kann eine geeignete Schedulingstrategie für Threadwechsel innerhalb einer Anwendung implementieren, sofern dies die Threadbibliothek anbietet.
- PULTs können auf jedem Betriebssystem mit entsprechender Bibliothek laufen.

Nachteile:

- Beim Aufruf eines blockierenden Systemaurfufs durch einen PULT wird *jedes Mal* die ganze Task blockiert.
- 2 PULTs der gleichen Task können *niemals* gleichzeitig auf zwei Prozessoren laufen..

Abschlussbemerkung zu yield():

yield() ist keine brauchbare, alleinige Lösung für das Umschaltproblem, da es zum einen kooperative Benutzer benötigt (selten anzufinden) und da es zum zweiten sowieso andere Umschaltereignisse gibt.

5.2 Umschaltereignisse

- Der gerade rechnende Thread (CT = current Thread) *terminiert*.
- CT initiiert eine synchrone Ein-/Ausgabeoperation, auf deren Ergebnis CT warten muss.
- CT *wartet auf eine Nachricht* von einem anderen Thread.
- CT ist *kooperativ*, gibt die CPU freiwillig an einen anderen Thread ab.
- CT *verbraucht seine Zeitscheibe*.
- CT hat eine *geringere Priorität* als ein anderer bereiter Thread T_j, weil
    - CT von einem Gerät unterbrochen wurde und im Rahmen der Unterbrechungsbehandlung T_j aufgeweckt worden ist.
    - Die Schläferzeit von T_j mittlerweile abgelaufen ist.
    - CT *einen neuen dringlicheren Thread erzeugt hat*.
    - CT *seine bisherige Priorität reduziert hat*.

Umschaltereignisse sind häufig die Folge von synchronen Ausnahmen (*exceptions*) bzw. asynchronen Unterbrechungen (*interrupts*):

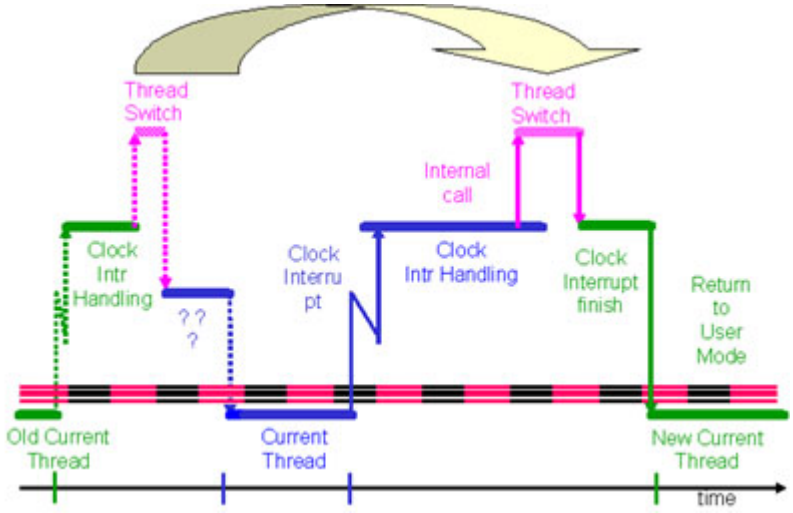
Synchrone Umschaltungen wegen Ausnahmesituationen:

- Fehlerfall (oft reproduzierbar)
    - Division durch Null
    - Adressierungsfehler
- Unvorhersehbares Ereignis
    - Seitenfehler (beim Virtuellen Speicher)
- Breakpunkt
    - Daten
    - Code
- Systemaufruf
    - Trap

Asynchron initiierte Umschaltungen wegen Unterbrechungen seitens der Peripherie:

- Uhrbaustein
    - Ende der Zeitscheibe
    - Weckersignal
- Drucker
    - Papierstau
    - Fehlendes Papier
- Netzwerkkarte
    - Angekommenes Nachrichtenpaket
- Andere CPU
    - Interprozessorsignal
    - Software-Interrupt

5.3 Allgemeines Umschaltereignis: Ende der Zeitscheibe



Wiederum wird der Trick angewendet, dass man innerhalb der thread_switch Prozedur nur den Stack austauscht, um so vom rechnenden Thread zum neuen Thread umzuschalten. Dies geht allerdings nur dann, wenn es sich bei thread_switch um eine Prozedur handelt und nicht um ein Makro, wie dies beispielsweise im L4-Mikrokern gemacht wird.

Frage: Welche Begründung könnte ein Systemarchitekt dafür angeben, dass er ein Makro für den Thread_Switch verwendet?

5.4 Noch offene Fragen

Wie behandelt man die Sonderfälle, dass ein Thread zum ersten Mal bzw. zum letzten Mal den Prozessor erhält? Wie behandelt man den Fall, dass überhaupt kein bereiter KLT-Thread mehr vorhanden ist?

Für all diese Sonderfälle, die zudem in einem extrem kritischen Bereich stattfinden, verwendet man ein sehr einfaches Systemarchitekturprinzip:

Führe Sonderfälle auf den Regelfall zurück!!!

Man installiert hierzu beispielsweise einen Leerlauf-KLT, der immer bereit ist, es sei denn, er muss gerade als Lückenbüßer rechnen. Dabei kann man aber durchaus im Leerlauf-KLT sinnvolle Arbeit verrichten lassen, man muss nur darauf achten, dass er niemals blockieren kann.

Frage: Welche Eigenschaften muss ein Leerlauf-KLT besitzen?
Welche nützlichen Aufgaben könnte man von einem Leerlauf-KLT durchführen lassen?

5.5 Threadzustände, Implementierungen

- Motivation und Einführung
- Notwendige Threadzustände
- Zusätzlich nützliche Threadzustände

5.5.1 Motivation und Einführung

Jedes Mal, wenn es zu einem Threadwechsel -ob PULT oder KLT- kommt, erhebt sich die Frage, auf welchen Thread umgeschaltet werden soll. Häufig genug wird ein anonymer thread_switch durchgeführt, d.h. man weiß zwar sehr genau, welcher Thread nicht mehr weiterlaufen soll, aber der relevante Scheduler

(Ablaufplaner) muss erst noch entscheiden, an welchem der im System zur Verfügung stehenden Threads weitergearbeitet werden sollte.

Wollte man beispielsweise jedes mal die Menge aller im System definierten KLT-Threads abklappern, bis man den Thread gefunden hat, den der Prozessor gemäß eines zu optimierenden Leistungsmaßes weiter ausführen soll, kann dies in großen Systemen mit sehr vielen gleichzeitig im System befindlichen KLTs zu einem erheblichen Leistungseinbruch führen, mit anderen Worten diese Vorgehensweise beim Systementwurf ist schlecht oder überhaupt nicht skalierbar. Somit wäre dies ein fundamentaler, weil dramatischer Systemarchitekturfehler, weil schlichtweg CPU-Power für die Verwaltung, statt für die Produktion ausgegeben wird. (Bis einschließlich Linux Version 2.4 war im Übrigen ein ähnlicher Lapsus implementiert, mit 2.6. hausierte man dann mit dem Schlagwort, dass von nun ab ein O(1) an Stelle des bislang realisierten O(n)-Schedulers implementiert sei.)

Frage: Was könnte in diesem Zusammenhang der Begriff O(n)-Scheduler bedeuten?

Es ist also auf den ersten Blick eine vernünftige Vorgehensweise zur Unterstützung des Schedulingverfahrens, wenn man bei der Suche nach einem geeigneten Kandidaten statt in der Menge aller Threads nur in der Teilmenge der Threads sucht, die zum Entscheidungszeitpunkt für die Abarbeitung auf dem Prozessor bereit sind. Beim Threadwechsel haben wir bereits einige Ereignisse festgestellt, in denen ein Thread momentan nichts mit der CPU anfangen kann, da er beispielsweise auf Eingabedaten von einem peripheren Gerät (z.B. von der Tastatur) warten muss.

5.5.2 Implementierung von Threadzuständen

In jedem Fall könnte man den entsprechenden Threadzustand als ein weiteres Attribut des Threads ansehen, womit eine Ablage im Threadkontrollblock gerechtfertigt wäre. Daneben mag es aber aus Effizienzgründen auch geboten sein, alle TCBs des gleichen Threadzustands in einer Datenstruktur zusammenzufassen. Insbesondere trifft dies auf alle Threads zu, die nur noch darauf warten, eine CPU zu bekommen. Solche Threads nennt man auch bereit (ready). Wenn man die Bereitwarteschlange (*ready queue*) so anlegt, dass der als nächster zu rechnende Thread immer am Kopf der Datenstruktur untergebracht wird, spricht man gelegentlich von einem O(1)-Scheduler.

Weitere Threadzustandskandidaten, für deren Implementierung eine gesonderte Datenstruktur vorgesehen werden könnte, sind die diversen Warteschlangen für blockierte Threads. Alle am Kern angebotenen Synchronisationsobjekte, aber auch alle dort angebotenen Ressourcenverwalter sind typischerweise mit individuellen Warteschlangen ausgestattet, an denen sich Threads in den Wartezustand begeben können, bis die Ressource wieder freigegeben wird oder bis eine Nachricht oder ein Signal angekommen ist.

Das obige Threadzustandsdiagramm ist geradezu klassisch und in dieser oder ähnlicher Form in den meisten Betriebssystemen auch realisiert. In dynamischen Systemen muss man auch Zustände einführen, die einen neuen Thread bzw. einen gerade beendeten Thread aufnehmen, manchmal ist es auch nützlich, Threadzustände einzuführen, mit denen man ausdrücken kann, dass ein Thread oder die gesamte Task aktuell im Hintergrundspeicher (z.B. auf Platte) im so genannten *swap device* ausgelagert ist.

Fest zu halten ist jedoch auch, dass jeder zusätzliche Threadzustand auch Aufwand bedeutet, zum einen räumlich, es muss ein zusätzlicher TCB-Eintrag vorgesehen werden oder gar eine separate Datenstruktur, aber auch laufzeitmäßig müssen die Zustandsübergange jeweils berücksichtigt werden.

5.5.3 Unabhängigkeit der Thread- und Taskzustände

Abhängig vom verwendeten Threadmodell sind die Threadzustände abhängig bzw. unabhängig vom jeweiligen Taskzustand.

Beim KLT-Modell bestimmt der höherwertige Threadzustand den umgebenden Taskzustand, d.h. sobald eine Task zumindest einen rechnenden KLT hat, nennt man die Task rechnend. Sobald wenigsten ein KLT bereit ist, aber sonst kein anderer KLT der gleichen Task rechnet, nennt man die umgebende Task auch bereit. In allen anderen Fällen ist die Task blockiert, wenn man nur das Dreizustandsmodell berücksichtigt.

Beim PULT-Modell kann eine Task rechnend sein, obwohl keiner ihrer Arbeiterthreads (*worker threads*) gerade rechnet. Ein Pult kann noch rechnend sein, obwohl die umgebende Task nur bereit oder gar blockiert sein kann.

Bei hybriden Threadmodellen wird die Grauzone an wechselseitigen Abhängigkeiten einfach größer, so dass im konkreten Fall geprüft werden muss, welche Zustandskombinationen möglich sind.