Praktikum MyThreads

"mythreads"

self–made user level threads

Frühlingssemester 2020

M. Thaler, J. Zeman

Überblick

In diesem Praktikum werden wir Teile eines eigenen, einfachen User-Level Thread-Paketes implementieren. User-Level Threads werden innerhalb eines Prozesses verwaltet und sind für das Betriebssystem *unsichtbar*. Unsere User-Level Threads erlauben dennoch preemptives Scheduling, im Moment allerdings ohne Unterstützung von Fliesskomma-Datentypen: Fliesskomma Register werden nicht gesichert (siehe dazu auch den Hinweis in Abschnitt 2.3.4).

Sie vertiefen in diesem Praktikum die Handhabung von Prozessen und Threads durch das Betriebssystem. Im Vordergrund stehen dabei das Dispatching d.h. das Einplanen und Umschalten der Threads sowie die Verwaltung der verschiedenen Warteschlangen (queues). Sie erhalten zusätzlich einen guten Einblick in die Komplexität eines Schedulers.

Dank
Wichtige Grundlagen und Lösungsansätze für dieses Praktikum wurden im Rahmen der Projektarbeit PA99/zem/4ab von Patrick Brunner und Roger Schneider erarbeitet. Herzlichen Dank!

Inhaltsverzeichnis

1 Einführung

1.1 Ziele

Die Ziele dieses Praktikums sind:

- Vertiefung Ihrer Kenntnisse zum Taskmanagement in Multitasking-Betriebssystemen am Beispiel der Implementation von selbst implementierten User-Level Threads.
- Einsatz Ihres Moduls **mlist** aus Praktikum Intro für die Verwaltung von Threads mit mehreren Warteschlangen: priorisierte Ready-Queues und eine Wait-Queue.

Mit Hilfe von Programmvorlagen implementieren bzw. erweitern Sie ein eigenes, einfaches User-Level Thread-Paket mit mehreren Modulen. Sie lernen dabei folgende Komponenten eines Multitasking-Kernels vertieft kennen:

- Scheduling (Dispatcher)
- Thread-Zustände (vereinfachte Prozesszustände)
- Scheduling Strategien für die Auswahl des nächsten Threads
- Task-Queues
- Implementation der `yield()`, `exit()` und `sleep()` System Calls
- Task-Switching
- Thread-Kontext
- Stack-Frames bei x86-Prozessoren

1.2 Durchführung und Leistungsnachweis

Es gelten grundsätzlich die Vorgaben Ihres Dozenten zur Durchführung der Praktika und zu den Leistungsnachweisen im Kurs BSy.
Die Inhalte des Praktikums gehören zum Prüfungsstoff.

1.3 Vorbereitende Aufgaben

Bereiten Sie das Praktikum in folgenden Schritten vor:

1. Die Dateien zu diesem Praktikum finden Sie im File **mythreads.tgz**, entpacken Sie das File, dabei werden vier Verzeichnisse angelegt: **priorityQueue**, **cfsQueue**, **waitQueue** und **dynamicPrioQueue**.
2. Wenn Ihr Listenmodul aus **Intro** korrekt funktioniert, kopieren Sie die Files **mlist.c** und **mlist.h** in alle vier Verzeichnisse. Sie werden für die Implementierung der Queues die Funktionen aus diesem Modul benötigen.
3. Lesen Sie die Aufgabenstellung und anschliessend den Theorieteil um sich einen Überblick zu verschaffen, bevor Sie mit der Implementierung beginnen.

2 Theorie

Im folgenden geben wir zuerst eine Übersicht zu den Modulen des Schedulers und im zweiten Teil eine Einführung zur Funktion des Thread Schedulers.

2.1 Module des Thread Paketes

Das Thread Paket besteht aus folgenden Modulen:

mythreads	Schnittstellenmodul mit einheitlichem Naming, das Header File muss vom Hauptprogramm eingebunden werden
mthread	Definition des Thread-Control-Blocks und Implementation der Initialisierungs- und Zugriffsfunktionen
dispatcher	Implementation der Dispatcherfunktionen
queues	Implementation der Queues und den entsprechenden Zugriffsfunktionen
mlist	Listenpaket für die Realisierung der Queues (siehe Praktikum Intro)

Figure 1 zeigt die Abhängigkeiten zwischen den einzelnen Modulen des Schedulers:

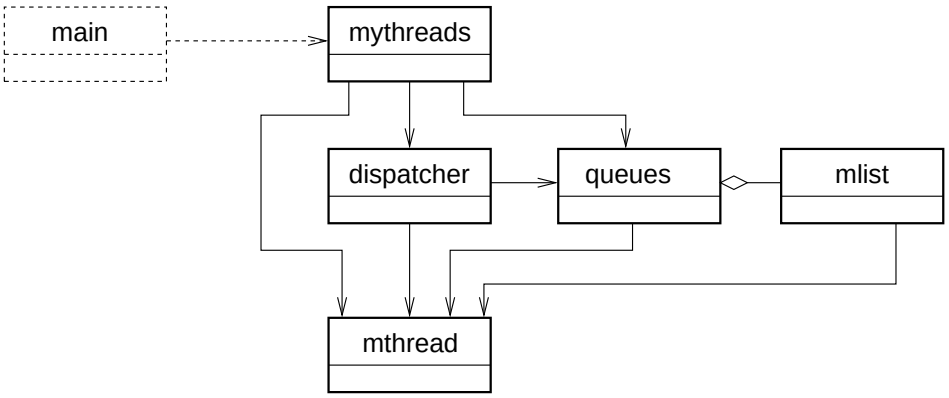

Fig. 1: Module und ihre Abhängigkeiten

In Anhang A finden Sie ein Sequenzdiagramm, das die Initialisierung, sowie das Starten und die Terminierung eines Threads beschreibt.

2.1.1 Das Modul `mythreads`

Das Modul `mythreads` definiert folgende Schnitstellenfunktionen als Makros:

`mthreadCreate(T,F,A,P,S)`	`mqInit(); dispatchInit(0)`
	`T = mtNewThread(F,A,P,S); mqAddToQueue(T,0)`
`mthreadJoin()`	`dispatchTask(JOIN)` oder `dispatchTask(START)`
`mthreadYield()`	`dispatchTask(YIELD)`
`mthreadExit()`	`dispatchTask(EXIT)`
`mthreadSleep(X)`	`dispatchTask(X)`

Beachten Sie, dass das Markro `mthreadCreate(T,F,A,P,S)` vier Funktion aufruft: die beiden Initalisierungsfunktionen `mqInit()` und `dispatchInit(0)` sind als Singleton implementiert und können deshalb mit jedem `mthreadCreate()` aufgerufen werden. `DispatchInit(0)` setzt das Scheduling Intervall auf den Defaultwert $1ms$. Das Makro `mthreadJoin()` startet die Ausführung der Threads und kehrt erst zurück, wenn alle Threads terminiert haben.

Das Makro `mthreadCreate()` erzeugt einen Thread-Control-Block und fügt ihn in die entsprechende Queue ein. Folgende Parameter müssen übergeben werden:

Parameter/Variable	Typ	Beschreibung
T	mthread_t*	ein Zeiger auf den Thread-Control-Block
F	tfunc_t	die Startfunktion des Threads (siehe commondefs.h)
A	void*	Zeiger auf das Argument der Thread Funktion
P	tprio_t	Priorität des Threads: HIGH MEDIUM LOW (siehe commondefs.h
S	unsigned	Stackgrösse, muss mindestens STACK_SIZE gross sein (wird automatisch gesetzt, falls zu klein)

Weitere Details entnehmen Sie bitte den entsprechenden Modulen.

2.1.2 Das Modul mthread

Das Modul mthread verwaltet die Threads und realisiert den Thread-Control-Block. Dieser speichert den Kontext der Threads, d.h. sämtliche Informationen, die die Queues und der Dispatcher für die Verwaltung eines Threads sowie für Speicherung und Wiederherstellung seines Zustandes benötigen. Die Daten und Funktionen sind in mthread.h definiert, für den Zugriff auf die Daten des Thread-Control-Blocks stehen teilweise Getter- und Setter Funktionen zur Verfügung.

Die wichtigsten Komponenten des Thread-Control-Blocks sind:

- die Thread-ID: eine eindeutige Kennung (hier ein Zahl)
- die Basis-Priorität: der statische Anteil der Priorität (HIGH, MEDIUM, LOW)
- die Ready-Time: die Aktivierungszeit, nachdem sich ein Thread schlafen gelegt hat
- die Start Funktion: Funktion, die beim Starten des Threads ausgeführt wird
- Zeiger auf Argumente, die der Start Funktion übergeben werden können
- ein Stackbereich: ein Speicherbereich für den Stack
- der aktuelle Stack-Pointer: wird beim Start bzw. bei Unterbruch des Threads gesetzt
- ein Flag mit dem angegeben wird, ob der Thread zum ersten mal aktiviert wird

2.1.3 Das Modul dispatcher

Der Dispatcher Modul ist verantwortlich für das Umschalten zwischen Threads (Thread-Switch), dazu gehört das Speichern des Kontextes, Auswahl (Aufruf von **mqGetNetThread()** im Modul queues) und Starten des nächsten Threads, das Verhindern von Stack-Überläufen und die korrekte Handhabung von CTRL-C (Programm Abbruch).

Gründe für das Umschalten bzw. den Aufruf des Dispatchers können sein:

- das Zeitinterval (Time Slice) ist abgelaufen (preempt, entspricht yield)
- der Thread gibt die CPU freiwillig ab (yield)
- der Thread legt sich schlafen (sleep)
- der Thread terminiert (exit)

Folgende Funktionen stehen zur Verfügung:

dispatchInit()	Singleton, initialisiert den Scheduler, setzt den Time Slice auf 1ms
dispatchTask(arg)	startet oder schaltet Threads um, mögliche Argumente sind: START, EXIT, YIELD bzw. arg $\geq$ 0 (sleep for arg ms)

Hinweis: das Zeitintervall für das preemptive Scheduling wird beim Einplanen eines Threads neu gestartet, d.h. jedem Thread steht jederzeit ein ganzer Time Slice zur Verfügung.

2.1.4 Das Modul queues

Das Modul `queues` verwaltet die Queues (Warteschlangen) und implementiert die Funktionen für den Zugriff auf die Ready und Wait-Queues.

Folgende Funktionen stehen zur Verfügung:

`mqInit()`	Singleton, alloziert und initialisiert die benötigten Queues
`mqDelete()`	entfernt die allozierten Queues
`mqGetNextThread()`	gibt nächsten lauffähigen Thread an den Dispatcher zurück, gibt es keine keine Threads mehr, muss **`NULL`** zurückgegeben werden
`mqAddToQueue(th,arg)`	reiht Thread **`th`** für `arg` $\leq$ `0` in die entsprechende Ready-Queue ein reiht Thread **`th`** für `arg` $>$ `0` in die Wait-Queue ein

Für das Lesen der Zeit steht die lokale Funktion **`mqGetTime()`** zur Verfügung. Die Funktion merkt sich beim ersten Aufruf die aktuelle Zeit (sollte deshalb von `mqInit` aufgerufen werden) und gibt bei allen weiteren Aufrufen die Zeit in Millisekunden seit dem ersten Aufruf zurück. Die `readyTime` eines Threads lässt sich so als Summe aus Sleep Time und aktueller Zeit berechnen.

2.1.5 Das Modul `mlist`

Das Modul `mlist` verwaltet die Listen mit denen die Queues implementiert werden. Jedes Listenelement speichert dabei einen Thread-Control-Block. Dieses Modul wurde bereits in Praktikum Intro realisiert.

2.2 Threadverwaltung und Scheduling

Das Threadverhalten kann ähnlich wie bei Prozessen mit einem Zustandsdiagramm und einem Queueing-Diagramm beschrieben werden. Figur 2 zeigt für Priority-Scheduling das Zustandsdiagramm zusammen mit drei Ready-Queues und einer Wait-Queue.

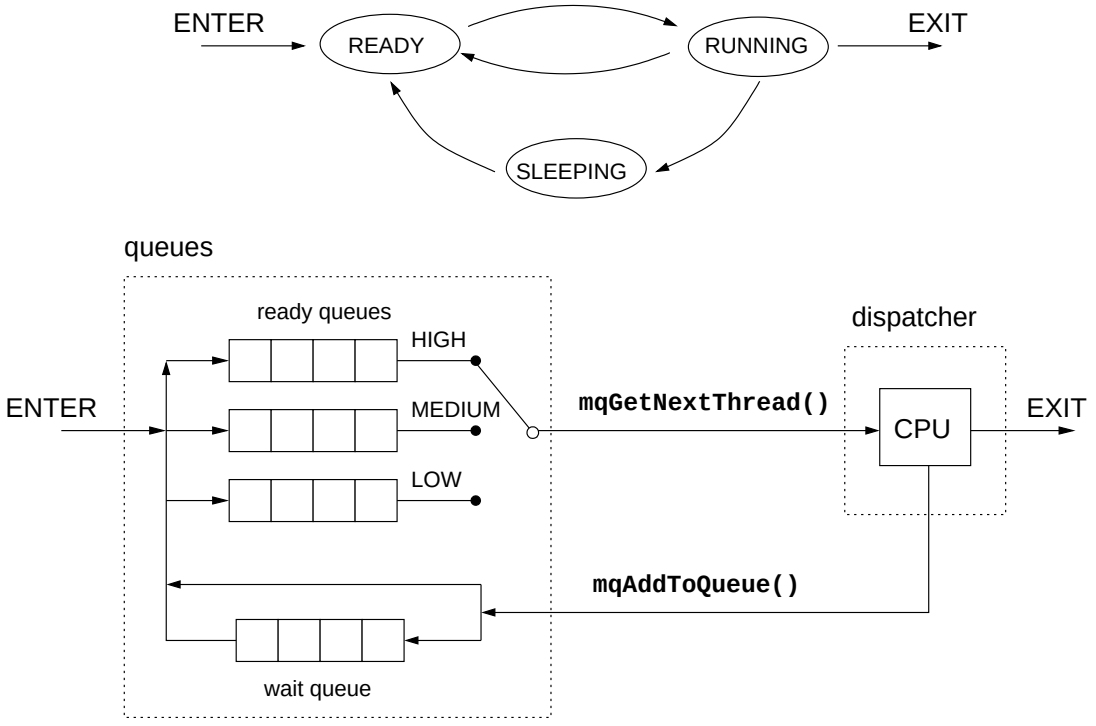

Fig. 2: Zustands- und Queueing Diagramm für Threads

2.2.1 Priority Scheduling und Sleeping Threads

Der voll ausgebaute Scheduler muss drei verschiedene Thread-Prioritäten berücksichtigen und schlafende Threads verwalten (siehe Fig. 2).

Die Queues sollen nach folgenden Kriterien verwaltet werden:

- ein soeben unterbrochener Thread wird aufgrund seiner Priorität an die entsprechende Ready-Queue gehängt oder in die Wait-Queue, wenn er sich schlafen legt
- ein neuer Thread wird wie folgt ausgewählt:
  - die Wait-Queue wird nach aktivierbaren Threads abgesucht, das sind Threads bei denen die `readyTime` kleiner ist als die aktuelle Zeit
  - aktivierbare Threads werden aufgrund Ihrer Priorität ans Ende der entsprechenden Ready Queue gehängt: Threads in der Queue können damit nicht verhungern
  - ausgehend von der Queue mit der höchsten Priorität, wird nun ein Thread gesucht, der gestartet werden kann
  - sind alle Ready-Queues leer, muss die Wait-Queue nach schlafenden Threads abgesucht werden: gibt es keine solchen Threads, kann der Scheduler beendet werden, andernfalls muss solange gewartet werden bis ein Thread aktiviert werden kann

Figur 3 zeigt einen Scheduling-Ablauf mit 5 Threads, einer Ready-Queue und einer Wait-Queue:

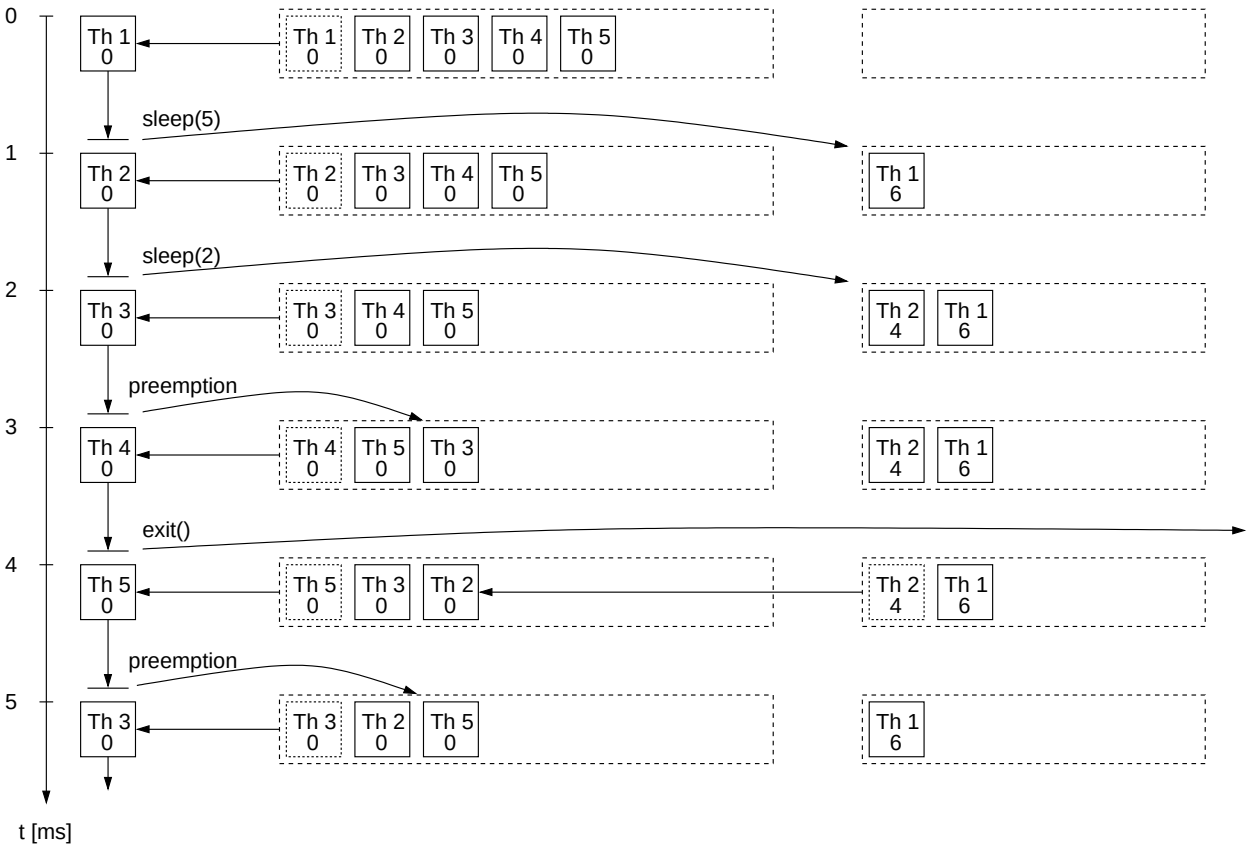

Fig. 3: Scheduling Ablauf für eine Ready und eine Wait Queue

2.2.2 CF-Scheduling: Completely Fair Scheduling

CF-Scheduling in Linux versucht die CPU-Zeit fair auf die lauffähigen Threads zu verteilen und zwar proportional zu ihren Prioritäten. Hier soll eine vereinfachte Variante mit einer Run- und einer Wait-Queue implementiert werden. Figur 4 zeigt das entsprechende Queuing Diagramm:

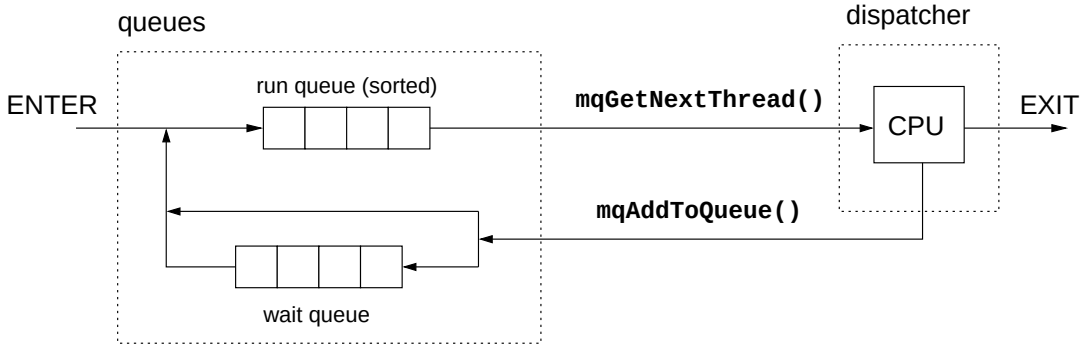

Fig. 4: CF-Scheduling mit einer Wait-Queue

CF-Scheduling sortiert die Threads aufsteigend nach der sogenannten Virtual-Runtime `vRuntime` und wählt jeweils den Thread mit der kleinsten `vRuntime` zum Einplanen auf der CPU (siehe Vorlesung zu Scheduling). Die `vRuntime` wird in unserem Fall wie folgt berechnet:

- Während dem Laufen des Threads erhöht sich die `vRuntime` um die *konsumierte* Laufzeit multipliziert mit der Thread-Priorität, die ≥ 1 sein muss.

  Die `vRuntime` wird damit am Ende des Time Slots (beim Zurückspeichern des Threads in die Ready Queue) wie folgt audatiert: **`vRuntime += mqGetRuntime() * prio`**.

  Da die Laufzeit in unserem Fall nur sehr ungenau gemessen werden kann, gibt `mqGetRuntime()` einen festen Wert zurück: `VR_DEFAULT` (definiert in `commondefs.h`).

- Während der Wartezeit wird pro Time Slot die `vRuntime` um `VR_DEFAULT/(N-1)` erniedrigt, wobei `N` die Anzahl Threads in der Run-Queue ist, d.h. es gilt: `vRuntime -= VR_DEFAULT/(N-1)`.

- Ist beim Einplanen eines Threads die `vRuntime` $! = 0$, muss dieser Offset gespeichert werden und vor dem nächsten Sortieren der Run-Queue bei allen Threads von der `vRuntime` abgezogen werden. Damit bleibt die `vRuntime` stabil bleibt und läuft nicht gegen $\pm\infty$.

2.3 Der Dispatcher

Der Dispatcher bildet den Kern des Thread Paktes und erlaubt preemptive Scheduling. Er ist für das Umschalten zwischen den Threads verantwortlich. Jeder Unterbruch eines Threads ruft den Dispatcher auf: dieser gibt den Thread an das Queue Modul zurück und fordert einen neuen Thread an (siehe auch Fig. 2).

Die Threads sind User Level Threads und werden deshalb vollständig innerhalb eines Prozesses verwaltet. Daraus ergeben sich folgende Eigenschaften:

- User Level Threads sind einfacher zu implementieren als Kernel Level Threads
- die Threads laufen innerhalb eines Benutzerprozesses und deshalb nur auf einem Core
- das Hauptprogramm selbst ist kein Thread (im Gegensatz zu `pthreads`) und das Scheduling muss explizit gestartet werden
- der Kernel weiss nichts von den Threads, damit blockiert ein blockierender Thread auch alle anderen Threads und bei Fehlern stürzt schlimmstenfalls der Benutzerprozess ab
- der Kernel selbst muss nicht modifiziert werden

2.3.1 Preemptive Scheduling

Preemption wird über einen internen Timer (POSIX: `setitimer()`) gesteuert, der in unserem Fall nach Ablauf das Unix-Signal `SIGVTALRM` auslöst. Der dazugehörige Signal-Handler macht nichts anderes als `dispatchTask(YIELD)` aufzurufen. Details zur Implementation sind in `dispatcher.c` und den entsprechenden Manual Pages zu finden.

2.3.2 Thread Umschaltung

Wie kann zwischen verschiedenen Threads umgeschaltet werden?

Die Thread-Umschaltung ist eigentlich eine Umschaltung zwischen Funktionen. Allerdings besitzt jeder Thread einen eigenen Stackbereich, im Gegensatz zu Funktionen, d.h. ein Thread Umschaltung ist eine Umschaltung zwischen Funktionen mit Austausch des Stackbereichs.

Wo und wie wird die Information zu einer Funktion gespeichert?

Unsere Thread-Umschaltung basiert darauf, dass in C/C++ bei jedem Funktionsaufruf ein Stack Frame aufgebaut wird. Im einfachsten Fall enthält der Stack Frame nur die Rücksprungadresse, in komplexeren Fällen die Argumente für die Funktion, die Rücksprungadresse, den *alten* Base Pointer und die lokalen Variablen.
Für uns von Bedeutung ist jedoch lediglich die ***Position*** des Stack Pointers, der bei Intel Architekturen immer auf das untere Ende (tiefste Adresse) eines Stack Frames zeigt.

```
int foo(int a, b) {
   int z;
   z = a + b + bar(b);
   return(z);
}

int bar(int x) {
   int v1, v2,
   v1 = x * x;
   v2 = v1 % 3 + x;
   return(v2)
}
```

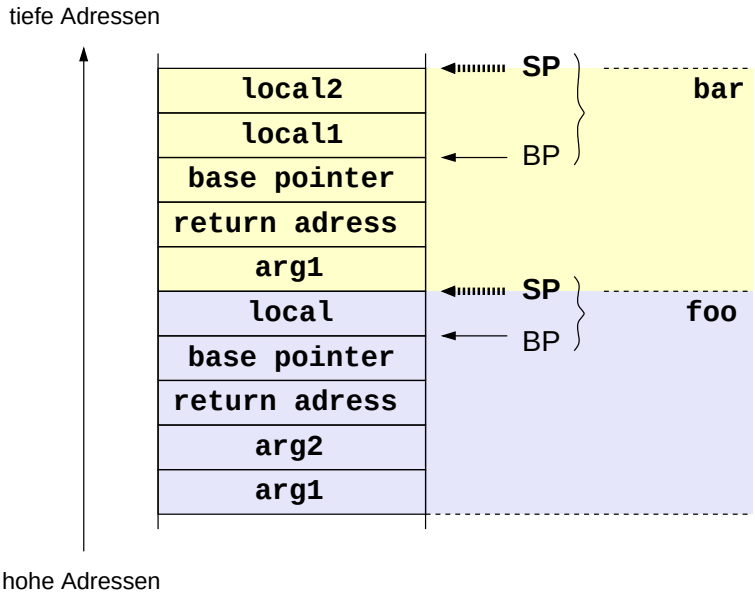

Fig. 5: Beispiel für Stack Frames

Befindet sich ein Programm während der Ausführung innerhalb einer Funktion, stehen im dazugehörigen Stack Frame alle Informationen, um in die *aufrufende* Funktion zurückzukehren, d.h. solange wir auf den Stack Frame Zugriff haben, ist eine Rückkehr ins laufende Programm möglich.

Fazit

Da der Stack Frame im Wesentlichen durch den *Stack Pointer* bestimmt wird, müssen wir bei einer Thread-Umschaltung ***nur*** die Stack Pointer austauschen: damit wird Information zur aktuellen Funktion (aktueller Programmstand) sowie der Stackbereich ausgetauscht.

2.3.3 Das Konzept der Thread-Umschaltung

In Anschnitt 2.3.2 haben wir festgestellt, dass Threads umgeschaltet werden können, indem die Stacks ausgetauscht werden. Die *Umschalt-Funktion* muss demzufolge aber auch als Funktion aufgerufen werden. Das ist gegeben, denn eine Umschaltung aus einer Thread Funktion in den Dispatcher erfolgt ausschlieslich durch den Aufruf **`dispachTask(arg)`**.
Das gilt auch für einen Unterbruch (Preemption) durch das Alarm Signal des Timers: unter Linux und Unix werden Signal Handler wie normale Funktionen auf dem aktuellen Stack des Programms gehandhabt und der Signal Handler ruft zudem `dispachTask(YIELD)` auf.

Bei Unterbruch eines Threads muss sein gesamter Zustand, der sogenannte Thread-Kontext gespeichert werden, um später an der gleichen Stelle im Programm weiterfahren zu können. Zum Thread-Kontext gehören: der Inhalt sämtlicher Register (inkl. Stack- und Base-Pointer), die Flags und der Programmzähler. Der Programmzähler ist implizit auf dem Stack als Rücksprungadresse gespeichert. Die Register und Flags lassen sich jedoch leicht auf den Stack speichern.

Damit ergibt sich folgender Ablauf für das Umschalten zwischen zwei Threads, wobei die farbigen Flächen angeben, in welchem *Stack Kontext* die perationen ausgeführt werden:

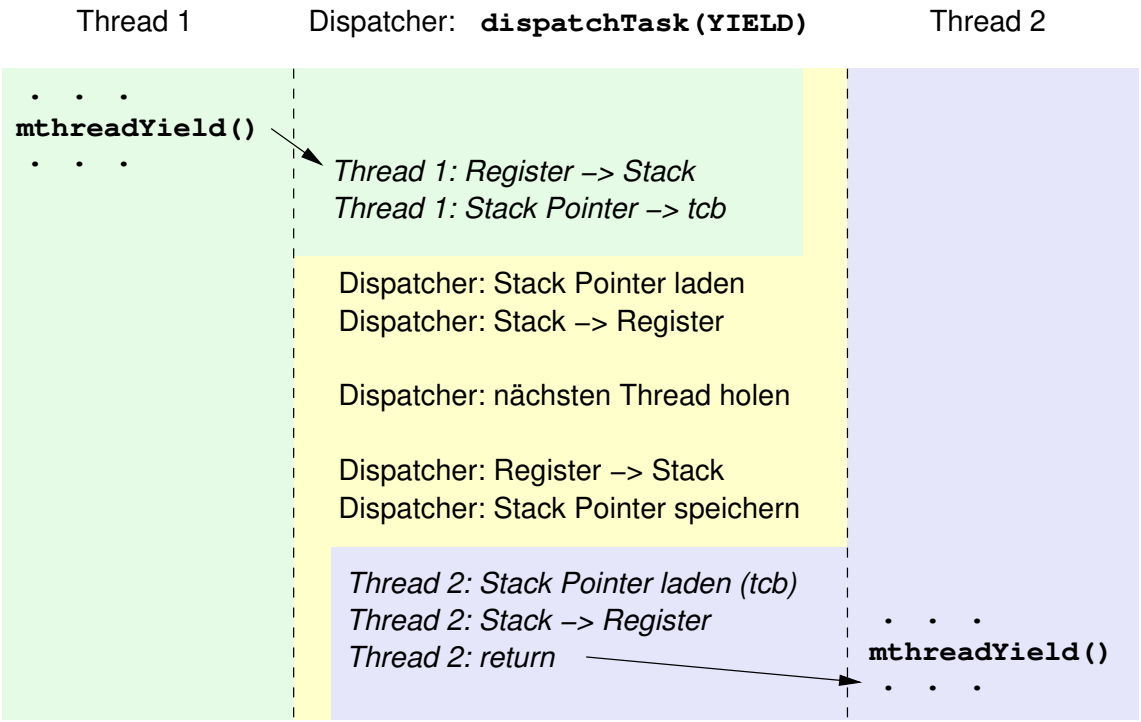

Fig. 6: Thread-Umschaltung

2.3.4 Wichtige Implementationsdetails

Den Code zur Dispatcherfunktion `dispatchTask()` finden Sie in Anhang B. Hier noch einige Erklärungen und Hinweise:

- Der Dispatcher muss unterscheiden, ob ein Thread zum erste Mal läuft oder nicht. Beim ersten Mal muss zuerst der Stack gesetzt werden und anschliessend die Startfunktion mit ihrem Argument aufgerufen werden. Bei den nächsten Aufrufen reicht es den Stack neu zu setzen und mit einem **`return`** in die Thread Funktion zurückzukehren.
- Das Speichern und Wiederherstellen der Register auf dem Stack der Threads sowie des aktuellen Stackpointers sind mit Makros definiert und im Files `asm.h` implementiert. Der Assembler-Code wird automatisch für 32-Bit und 64-Bit CPUs übersetzt.

- Ein Thread der terminiert, kann einfach gelöscht werden. Als *aktiver* Thread ist er ja in keiner Queue abgelegt. Stellen Sie aber sicher, dass alle dynamische allozierten Speicherbereiche korrekt freigegeben werden.

- Der hier implementierte Thread-Scheduler unterstützt keine Fliesskomma-Datentypen. Dazu müsste der Zustand der FPU, MMX und SSE Register gesichert und wieder hergestellt werden. Für das Sichern dieser Register wird pro Thread ein 512-Byte grosser Speicherbereich benötigt. Das Sichern und Wiederherstellen der Resgister geschieht dann mit den Assemblerinstruktionen `FXSAVE m512` und `FXRSTOR m512`, die speziell für Kontextumschaltungen gedacht sind.

  Die dazu notwendigen Inline-Assembler Funktionen haben wir für Sie in `asm.h` vorbereitet: **`SAVE_FX(void *mem)`** sichert die Fliesskomma-Register, wobei **`mem`** ein Zeiger auf den 512-Byte grossen Speicherbereich ist, **`RESTORE_FX(void *mem)`** stellt die Registerinhalte wieder her.

  Für die Implementation muss der Thread-Control-Block um einen 512-Byte grossen Eintrag erweitert werden und der Dispatcher um die Assembler Funktionen `SAVE_FX()` und `RESTORE_FX()`.

3 Aufgaben

3.1 Aufgabe 1

In einem ersten Schritt soll ein einfaches, prioritätstbasiertes Round-Robin Scheduling implementiert werden. Im Verzeichnis **`prioQueues`** finden Sie alle notwendigen Files bzw. Module für die Implementierung, bis auf das Modul `mlist`, das Sie im Praktikum Intro implementiert haben (Files `mlist.c`, `mlist.h`).

Was fehlt, sind die Funktionen **`mqGetNextThread()`** und **`mqAddToQueue()`** im Modul **`queues`**. Sie werden vom Dispatcher aufgerufen und fordern einen neuen Thread zur Einplanung an bzw. geben einen unterbrochenen Thread in die Queue zurück. Die Funktionen `mqInit()` und `mqDelete()` haben wir schon implementiert, wie auch die Definition der Queues als globaler Array mit der notwendigen Anzahl Einträge: `list_t* readyQueue[NUM_PRIO_QUEUES];`.

Hinweis: die Prioritäten sind sind so gewählt, dass sie als Index für den `readyQueue`-Array verwendet werden können (`HIGH` = 0, etc.).

Gehen Sie in folgenden Schritten vor:

1. Studieren Sie die Funktion `mqInit()` und `mqDelete()`, Sie werden die beiden Funktionen in den nachfolgenden Aufgaben anpassen müssen. Beachten Sie, dass `mqInit()` wie ein Singleton arbeitet und dass die Queue in einer globalen Variablen abgelegt wird.

2. Implementieren Sie die Funktion **`mqGetNextThread()`**, die den ersten Thread aus der Queue mit der höchst möglichen Priorität zurück gibt: das ist ein Zeiger auf den Thread-Control-Block vom Typ `mthread_t` bzw. `NULL`, falls in keiner Queue ein Thread gefunden wurde.

3. Implementieren Sie nun die Funktion **`mqAddToQueue()`**, die einen Thread-Control-Block wieder in die Queue mit der entsprechenden Priorität einreiht, den Parameter `sleepTime` können Sie vorerst ignorieren. Verwenden Sie die Funktion **`mlEnqueue()`** aus Ihrem Modul `mlist`.

Für Testzwecke stehen die Programme `main1.c` und `main2.c` zur Verfügung (passen Sie die Programme falls notwendig Ihren Bedürfnissen an).

3.2 Aufgabe 2

Wir wollen zuerst den CF-Scheduler ohne Wait-Queue implementieren. Dazu haben wir im Verzeichnis **cfsQueue** die Datei `queues.c` entsprechend vorbereitet. Anstelle der `prioQueue` haben wir die zwei Queues **runQueue** und **tmpQueue** definiert und die Funktionen `mqInit()` und `mqDelete()` entsprechend angepasst.

Gehen Sie für die Implementation des CF-Schedulers wiederum schrittweise vor:

1. Implementieren Sie die Funktion **mqGetNextThread()** für CF-Scheduling, wie in Abschnitt 2.2.2 beschrieben. Die Virtual-Runtime ist im Thread-Control-Block als **float vRuntime** definiert, zusätzlich müssen Sie in `mqGetNextThread()` eine statische Variable mit Namen `offset` zum Speichern des Offsets definieren.

   - Sortieren Sie die **runQueue** durch Umkopieren der Threads aus der `runQueue` in die `tmpQueue` mit `mlDequeue()` und `mlSortIn()` sowie anschliessendem Vertauschen der Zeiger auf die Queues → die **runQueue** ist sortiert. Vor dem sortierten Einfügen, muss die `vRuntime` aufdatiert werden:

     ```
     vRuntime = vRuntime - VR_DEFAULT/(N-1) - offset
     ```

     Vergessen Sie nicht für das Sortieren eine entsprechende Callback-Funktion zu implementieren.
   - Mit `mlDequeue()` wird nun der erste Thread aus der sortierten Run-Queue entnommen. Die `vRuntime` dieses Threads muss in der Variablen `offset` gespeichert werden, dann kann der Thread an den Dispatcher zur Einplanung zurück gegeben werden.

2. Implementieren Sie die Funktion **mqAddToQueue()**, die den vom Dispatcher übergebenen Thread in die Run-Queue einreiht und dabei die `vRuntime` wie folgt aufdatiert:

   ```
   vRuntime += mqGetRuntime() * prio
   ```

   Beachten Sie, dass die Prioritäten `HIGH, MEDIUM, LOW` den Werten `0, 1, 2` entsprechen, d.h. `prio` muss wie folgt gewählt werden: `prio = tcb->tPrio + 1`.

Für Testzwecke steht das Testprogramm `main3` zur Verfügung. Das Programm misst im Wesentlichen welchen Anteil an Rechenzeit ein Thread während einem Zeitintervall proportional zu seiner Priorität erhält und vergleicht die Messung mit dem theoretischen Wert (Toleranz $\pm 4\%$).

- `main3.e` ohne Parameter startet 3 Threads mit den Prioritätswerten `HIGH`, `MEDIUM` und `LOW`, wobei jeder Thread nach jeder Iteration `mthreadYield()` aufruft
- `main3.e` mit einem Parameter startet wiederum 3 Threads mit den Prioritätswerten `HIGH`, `MEDIUM` und `LOW`, aber ohne `mthreadYield()` aufzurufen, z.B. `main3.e 1`
- `main3.c` mit 3 Parametern startet die 3 Threads mit entsprechenden Prioritäten, z.B. `main3.e 0 0 8`, mit einem beliebigen 4. Parameter kann `mthreadYield()` ausgeschaltet werden.
- Vergleichen und diskutieren Sie die Schedules von `main2.e` aus Aufgabe 1 und von `main3.e 1` (ein Parameter) aus Aufgabe 2. Was fällt speziell auf?
  Hinweis: dass das Testprogramm `main2.e` mit CFS nicht korrekt funktioniert ist in Ordnung: wieso ist das so? Hinweis: vergleichen Sie die Threadfundktionen.

3.3 Aufgabe 3

Bis jetzt werden Threads nach jedem Unterbruch wieder in die Run-Queue eingereiht: neu soll ein Thread auch für eine gewisse Zeit suspendiert werden können. Dazu muss dieser Threads die Funktion **mthreadSleep(msec)** bzw. `dispatchTask(msec)` aufrufen, wobei das Argument **msec** angibt, wie viele Millisekunden der Thread suspendiert bleibt bzw. in der Wait-Queue warten soll.

Wird der Dispatcher mit einem Argument > 0 aufgerufen, interpretiert er dies als *Schlafdauer* und übergibt den Wert als Parameter **sleepTime** zusammen mit dem Thread-Control-Block an die Funktion **mqAddToQueue()**. Dabei wird der Thread sortiert nach **readyTime** in die Wait-Queue eingereiht. Die `readyTime` kann mit Hilfe der Funktion `mqGetTime()` und der Schlafdauer bestimmt werden:

```
readyTime = mqGetTime() + sleepTime
```

Zusätzlich muss die Funktion `mqGetNextThread()` angepasst werden: bei jedem Aufruf muss überprüft werden, ob wartende Threads lauffähig geworden sind, d.h. ob `mqGetTime()` eine spätere Zeit als die jeweilige Ready-Time zurückgibt.

Kopieren Sie die Files `queues.c` und `dispatcher.c` aus Aufgabe 2 ins Verzeichnis **waitQueue** und gehen Sie wie folgt vor:

1. Implementieren Sie dazu eine zusätzliche Wait-Queue sowie das Handling der Ready-Time in der Funktion **mqAddToQueue()**.
2. Implementieren Sie das Abfragen der Wait-Queue in **mqGetNextThread()**. Gerade aufgewachte Threads werden mit `vRruntime` = 0 in die Run-Queue eingereicht, bevor diese sortiert wird: damit rücken diese Threads an den Anfang der Run-Queue.
3. Testen Sie das Programm, verwenden Sie dazu das Programm `main4.c`.

3.4 Aufgabe 4 (optional)

Priority Scheduling mit statischen Prioritäten kann zu Starvation (Verhungern) führen (siehe Aufgabe 1) im Gegensatz zum CF-Scheduling. Implementieren Sie eine einfache Scheduling Strategie für Priority Schedling aus Aufgabe 1, die Startvation verhindert. Dabei soll die Priorität dynamisch in Funktion der Wartezeit angepasst werden. Überlegen Sie sich dazu eine möglichst einfache Strategie.

A Sequenzdiagramm

Das unten stehendes Sequenzdiagramm zeigt die Initialisierung der Queues und des Dispatchers, sowie die Ausführung einer Thread Funktion. Während der rot markierten Phasen wird Code des Threads mit eigenem Stack ausgeführt. Kursive Namen entsprechen Variablen bzw. Rückgabewerte von Funktionen.

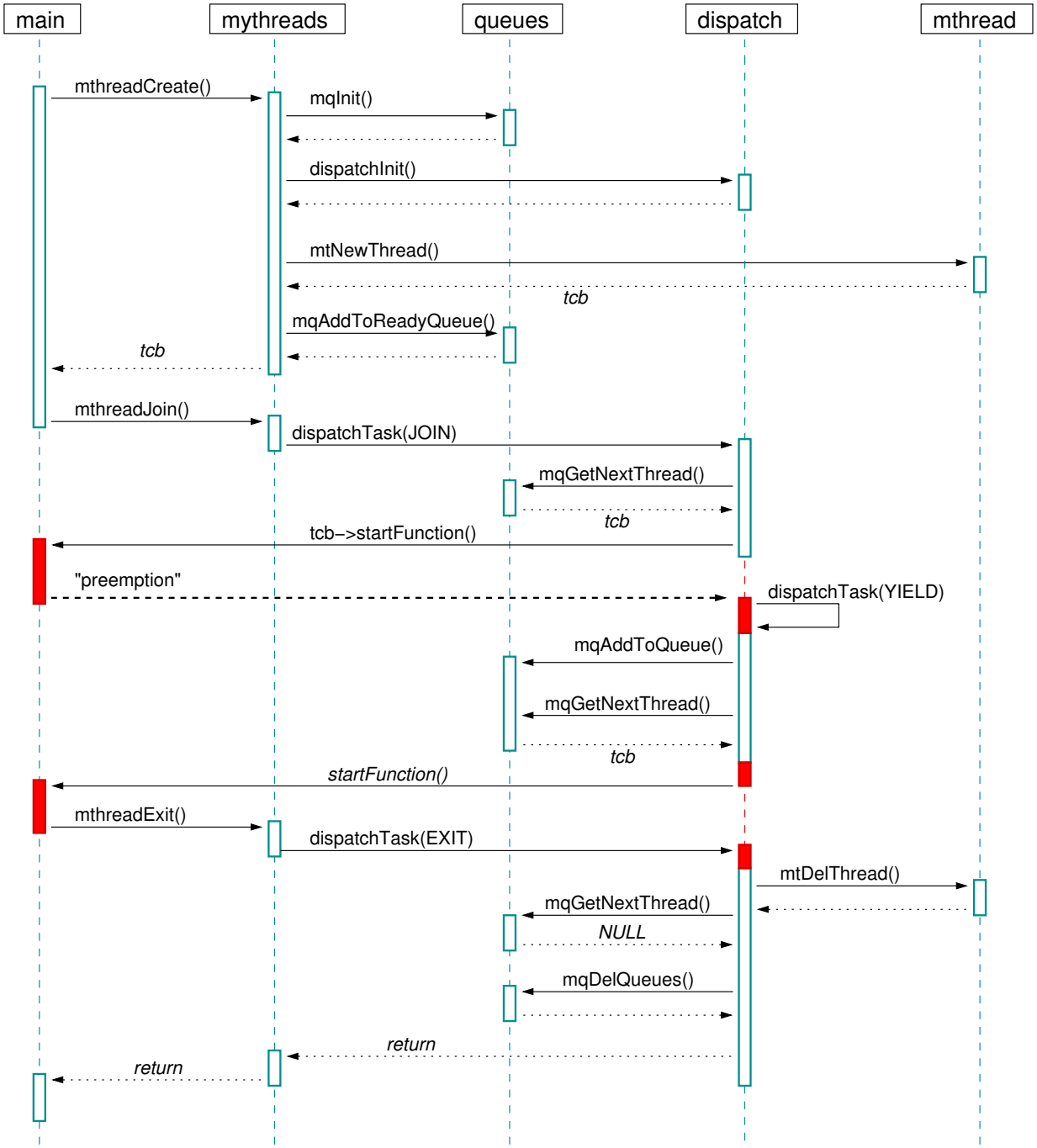

Fig. 7: Sequenzdigramm mit einem Thread

B Programmcode: Dispatcher

```
void dispatchTask(int argument)
{
    tfunc_t   startFunction;              // start function of a thread
    int       stacksize;                  // current stack size
    finishTimeSlice();                    // finish time slice (block signal)
    schedArg = argument;                  // save argument to static variable

    /* --- check how we were called ----------------------------------------- */

    switch (schedArg)    {
        case START:
        case JOIN:
            dispatchInit(DEF_TIME_SLICE);   // init with defaults
            break;
        case EXIT :                         // thread exits with mythread_exit()
            {
            RESTORE_SP(scheduSP);           // scheduler: restore stackpointer
            RESTORE_REGS();                 // scheduler: restore regs and flags
            }
            mtDelThread(activeThread);      // thread:    is deleted
        break;
        default :                           // thread yields or waits
            {
            SAVE_REGS();                    // thread:    save regs and flags
            SAVE_SP(threadSP);              // thread:    save stackpointer
            RESTORE_SP(scheduSP);           // scheduler: restore stackpointer
            RESTORE_REGS();                 // scheduler: restore regs and flags
            }

            mtSaveSP(activeThread, threadSP); // thread:  store stackpointer

            if (schedArg == YIELD)
                mqAddToQueue(activeThread, 0);
            else
                mqAddToQueue(activeThread, schedArg);
            break;
    }
```

siehe nächste Seite für zweiten Teil!

```
    /* --- now we have to look for the next thread to be scheduled ---------- */

    activeThread = mqGetNextThread();               // get next ready thread

    if  ((shutdown) || (activeThread == NULL)) {
        if (shutdown)
            printf("\n*** got CTRL_C\n");
        else {
            printf("\n*** no more threads to schedule ***\n");
        }
        mqDelete();
    }
    else {

        /* --- now start or restart thread ----------------------------------*/

        threadSP = mtGetSP(activeThread);           // thread:    get stackptr

        if (mtIsFirstCall(activeThread)) {          // if thread never run
            mtClearFirstCall(activeThread);         // thread: mark as started
            {
              SAVE_REGS();                          // scheduler: save regs
              SAVE_SP(scheduSP);                    // scheduler: save SP
              RESTORE_SP(threadSP);                 // thread:    set SP
            }
            startFunction = mtGetStartFunction(activeThread);// get start func
            startTimeSlice();                               // start time slice
            startFunction(mtGetArgPointer(activeThread));    // call start func
        }
        else {
            SAVE_REGS();                            // scheduler: save regs
            SAVE_SP(scheduSP);                      // scheduler: save stackptr
            RESTORE_SP(threadSP);                   // thread:    set stackptr
            RESTORE_REGS();                         // thread:    restore regs
            startTimeSlice();                       // start time slice
        }
    }

    return;
}
```