Praktikum MyThreads

"mythreads"

self-made user level threads

Frühlingssemester 2012

M. Thaler, J. Zeman

Überblick

In diesem Praktikum implementieren wir ein eigenes, einfaches User-Level Thread-Paket. User-Level Threads werden innerhalb eines Prozesses verwaltet und sind für das Betriebssystem *unsichtbar*. Unsere User-Level Threads erlauben sowohl preemptives Scheduling als auch non-preemptives Scheduling, im Moment allerdings ohne garantierte Unterstützung von Fliesskomma-Datentypen in Thread-Funktionen (Fliesskomma Register werden nicht gespeichert)

Sie vertiefen in diesem Praktikum die Handhabung von Prozessen und Threads durch das Betriebssystem. Im Vordergrund stehen dabei das Scheduling, die Warteschlangen (queues) und das Task-Switching.

Sie erhalten zusätzlich einen guten Einblick in die Komplexität eines Schedulers für verschiedene Anwendungsgebiete.

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!

Inhalt

1. Aufgabenstellung

1.1 Ziele

Die Ziele dieses Praktikums sind

- Vertiefung Ihrer Kenntnisse zum Prozessmanagement in Multitasking-Betriebsystemen am Beispiel der Implementation von eigenen User-Threads.
- Einsatz Ihres Moduls `mylist` aus Praktikum 1 für die Verwaltung von Threads mit mehreren Warteschlangen: priorisierte Read-Queues und eine Wait-Queue.

Mit Hilfe von Programmvorlagen implementieren resp. 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()` und `sleep()` System Calls
- Task-Switching
  - Thread-Kontext
  - Stack-Frames bei x86-Prozessoren

1.2 Aufgaben

Bitte lesen Sie das Praktikum zu Hause durch! Die theoretischen Grundlagen und hilfreiche Zusatzinformationen finden Sie in Abschnitt 2 dieser Anleitung.

1.2.1 Ein einfacher Scheduler

Um Ihnen einen schnellen Einstieg zu ermöglichen, stellen wir Ihnen einen einfachen Scheduler und die Implementation einer einfachen Ready-Queue zur Verfügung. Beides sind C Module, eine detaillierte Beschreibung des Schedulers finden Sie in Abschnitt 2.3.

1. Die Dateien zu diesem Praktikum finden Sie auf dem Webserver: **`MyThreads.tgz`**.
2. Studieren Sie die Beschreibung der Module `mythread`, `scheduler` und `queues` in Abschnitt 2.1.2 und die entsprechenden Listings zum Praktikum. Dazu gehören auch die Demo-Programme `main1.c` und `main2.c`.
3. Übersetzen Sie die Programme `main1` und `main2` mit `make` (`make all`). Experimentieren Sie mit den beiden Programmen und versuchen Sie folgende Fragen zu beantworten (für preemptives und non-preemptives Schedulin):

   - wie werden die Threads verwaltet?

   - in welcher Reihenfolge werden die Threads aktiviert (Scheduling Verfahren)?

   - welchen Einfluss hat die Reihenfolge der Erzeugung der Threads auf das Scheduling?

   - welchen Einfluss hat die Wahl des Schedulingverfahrens

4. Notieren Sie, wie sich `mythreads` von den `pthreads` unterscheiden, beachten Sie speziell folgende Punkte:
   - Start des Scheduling
   - Kernel- oder User-Level Threads
   - Erzeugung und Initialisierung von mythreads
   - Struktur des Hauptprogramms
   - warten auf die Beendigung der Threads am Ende des Programms
   - was geschieht, wenn ein Thread aus `mythread` die Linux Systemfunktion `sleep()` aufruft?
   - werden Mutexes benötigt?

1.2.2 Ihr eigener Thread Scheduler

Erweitern Sie den Demo-Scheduler zu einem Thread-Paket. Dazu gehört die Implementation folgender Punkte:

- **Verwaltung einer beliebigen Anzahl, dynamisch erzeugter Threads in der Ready-Queue**. Verwenden Sie Ihr Listenpaket aus Praktikum 1 (oder die entsprechende Musterlösung).
- **Priority-Scheduling**: jeder Thread kann drei verschiedenen Prioritätsstufen zugeordnet werden (HIGH, MEDIUM, LOW). Für jede Prioritätsstufe wird eine Ready-Queue benötigt. Das Scheduling berücksichtigt zuerst die Threads in den Warteschlange mit den höchsten Prioritäten.
- **Implementation der Funktion `MyThreadSleep(x)`**, die den Thread für x Millisekunden schlafen legt. Diese Threads werden in einer Wait-Queue nach aufsteigenden Aktivierungszeiten einsortiert und bei Erreichen der Aktivierungszeit in die entsprechende Ready-Queue verschoben. Dazu muss bei jedem Scheduling-Event die Systemzeit mit der Aktivierungszeit der einzelnen Threads in der Wait-Queue verglichen werden.
- **Überprüfen des Stacks**: jeder Thread besitzt einen eigenen Stack fester Länge, der bei der Erzeugung festgelegt wird (Default 4096 Bytes). Die Stackbereiche werden dynamisch alloziert, dürfen also auf keinen Fall überlaufen, sonst werden eventuell Daten von anderen Threads überschrieben. Deshalb muss bei jedem Thread-Switch der Stack des Threads überprüft werden. Beim Unterschreiten einer vorgegebenen Limite muss der Thread terminiert werden.
- **Debugging:** bauen Sie an kritischen Stellen Funktionen ein, die z.B. Informationen zu den Threads in den einzelnen Warteschlangen ausgeben können. Die Debugging-Ausgaben sollen einfach ein- und ausgeschaltet werden können. Ein Beispiel finden Sie im Listing zum Scheduler
- **Fakultativ 1:** Priority Scheduling mit statischen Prioritäten kann zu Starvation (Verhungern) führen. Implementieren Sie eine einfache Scheduling Strategie die Startvation verhindert. Die Priorität muss ähnlich wie bei Unix dynamisch angepasst werden, indem z.B. die Wartezeit oder die mittlere CPU-Nutzung berücksichtigt wird.
- **Fakultativ 2:** Portieren Sie Ihr Thread-Paket auf eine Windows-Umgebung nach Ihrer Wahl.

Wichtige Hinweise:

- Alle Funktionen, die mit Warteschlangen (queues) zu tun haben, können Sie mit Hilfe des Moduls `queues` implementieren.
- Erweitern Sie die Funktion `getNextThread()` so, dass zuerst die Wait-Queue nach aktivierbaren Threads abgesucht wird und diese in die Ready-Queue verschoben werden. Dann wird der Thread mit der höchsten Priorität aus der entsprechenden Queue zurückgegeben.
  Überlegen und diskutieren Sie, ob ein soeben *aufgewachter* Thread in die Ready Queue mit höchster Priorität eingefügt werden soll oder in die der Priorität entsprechenden Queue. Soll der Thread an das Ende der Liste oder den Anfang eingefügt werden? Welchen Einfluss hat die Art der Anwendung (Multitasking, Echzeit) auf die Beantwortung dieser Fragen?

- Integrieren Sie Ihr Listenpaket in den Demo-Scheduler. Beginnen Sie mit einer einzigen Ready-Queue, erweitern Sie diese auf drei Prioritäts-Queues und implementieren Sie erst anschliessend die Wait-Queue. Testen Sie jede Änderung ausgiebig.
- Für die Implementation der Wait Queue muss `taskSwitch()` im Module `scheduler` entsprechend angepasst werden: es gibt keinen *idle* Thread, diese Funktion muss von der Funktion `taskSwitch()` übernommen werden.

1.2.3 Testprogramme

Ihr Thread-Paket muss gründlich getestet werden. Schreiben Sie dazu einfache Testprogramme, die einzeln überprüfen, ob:

- mindestens 20 Threads erzeugt werden können (die Threads können auch innerhalb von Threads erzeugt werden)
- das Scheduling, die Thread-Prioritäten und die *Schlafzeiten* richtig berücksichtigt werden
- die Threads in der richtigen Reihefolge scheduliert werden
- ein Stacküberlauf verhindert wird
- die Threads auch mit rekursiven Aufrufen richtig arbeiten
- der allozierte Speicher auch bei Abbruch mit CTRL-C freigegeben wird

1.3 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.

2. Theorie

2.1 Das Thread-Paket für den Demo-Scheduler

In den folgenden Abschnitten beschreiben und diskutieren wir die Komponenten des Thread-Paketes. Wir stellen Ihnen dazu ein Thread-Paket zur Verfügung, das die allernotwendigsten Grundfunktionen realisiert. Dieses Paket beinhaltet eine einzige Ready-Queue (Ready-Queue mit Array realisiert).

2.1.1 Module des Thread-Paketes

2.1.1.2 mythread

Ein Modul zur Verwaltung von Threads, realisiert den Thread-Control-Block. Enthält den Kontext, d.h. sämtliche Informationen, die der Scheduler für die Verwaltung des Threads sowie für Speicherung und Wiederherstellung seines Zustandes benötigt. Die Daten und Funktionen sind in `mythread.h` definiert (für den Zugriff auf die Daten des Thread-Control-Blocks stehen Funktionen zur Verfügung, wir haben einige dieser Funktionen als Macros defniert, aber nicht verwendet).

Dazu gehören:

- 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, nach Aufruf der `myThreadSleep()` Funktion
- 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 resp. bei Unterbruch des Threads gesetzt
- ein Flag mit dem angegeben wird, ob der Thread zum ersten mal aktiviert wird
- sämtliche relevanten Registerinhalte, wird in unserem Fall auf dem Stack abgelegt und nicht explizit im Thread-Control-Block

2.1.1.3 Scheduler

Ein Modul zur Verwaltung von Threads: ist verantwortlich für das Umschalten zwischen den Threads (Thread-Switch). Das Modul ist im File `scheduler.h` definiert und implementiert:

- das Umschalten zwischen Threads, d.h. sichern und speichern des Thread-Kontextes, Auswahl des nächsten Threads, der aktiviert werden soll und das Erzeugen eines Threads sowohl für preemptives Scheduling als auch non-preemptives Scheduling
- die Behandlung von Stack-Überläufen
- das Abfangen von CTRL-C

2.1.1.4 Queues

Ein Modul zur Verwaltung von Queues (Warteschlangen). Implementiert die Funktionen für den Zugriff auf die Ready- und Wait-Queue. Das Modul ist im File `queues.h` definiert. Die wichtigsten Funktionen sind:

- `addToReadyQueue()`: hängt den Thread an die Ready-Queue
- `addToWaitQueue()`: hängt den Thread sortiert nach Aktivierungszeit in die Wait-Queue, wenn er mit `myThreadSleep(x)` die Kontrolle abgegeben hat (die Aktivierungszeit berechnet sich aus der aktuellen Systemzeit + der Schlafzeit des Threads (Zeitangaben in Millisekunden))

- `getNextThread()`: liefert, falls vorhanden, den nächsten Thread in der Ready-Queue (sonst NULL)
- `checkWaitQueue()`: testet, ob in der Wait-Queue eine oder mehrere Threads auf die Aktivierung warten, aber noch nicht lauffähig sind

2.1.1.5 mylist

Ein Modul zur Verwaltung von Listen. Wird für die Realisierung der einzelnen Queues benötigt und speichert pro Element den Kontext (Thread-Control-Block) eines Threads, d.h. eine Struktur des Typs **ThreadCB**. Mylist haben Sie einem früheren Praktikum implementiert und können es vollumfänglich übernehmen.

2.2 Threadverwaltung

Das Verhalten von Threads kann ähnlich wie bei Prozessen mit einem Zustandsdiagramm und einem Queuing-Diagramm beschrieben werden. Fig. 1 zeigt die beiden Diagramme für unseren Thread-Scheduler[1], die Ready-Queue ist für 3 Prioritäten ausgelegt.

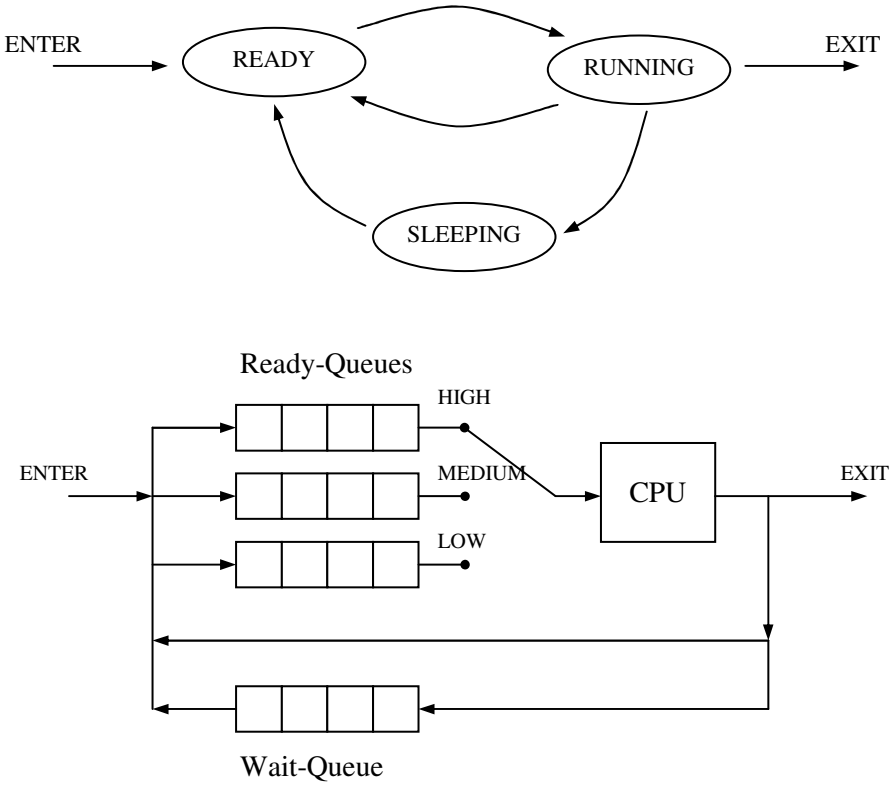


Fig. 1: Zustands- und Queueing Diagramm des Schedulers

[1] Die Thread-Zustände werden in unserer Implementation nicht explizit im Thread-Control-Block gespeichert, weil diese Information implizit durch die Queue gegeben ist

2.2.1 Der erweiterte Scheduling-Algorithmus

Der erweiterte Scheduler soll, drei verschiedene Thread-Prioritäten berücksichtigen sowie schlafende Threads verwalten (siehe Fig. 1).

Die Queues sollen nach folgenden Kriterien verwaltet werden:

- Ein soeben unterbrochener Thread wird aufgrund seiner Priorität an die entsprechende Ready-Queue gehängt.
- Ein neuer Thread wird wie folgt ausgewählt:
  - die Wait-List wird nach aktivierbaren Threads abgesucht (bei diesen Threads ist die Ready-Time kleiner oder gleich der aktuellen Systemzeit)
  - aktivierbare Threads werden aufgrund Ihrer Priorität ans Ende der entsprechenden Ready-Queue gehängt, damit lässt sich verhindern, dass bereits in der Queue vorhandene Threads verhungern
  - ausgehend von der Queue mit der höchsten Priorität, werden nun Threads gesucht, die gestartet werden können und der erste Thread an `taskSwitch()` übergeben
  - sind alle Ready-Queues leer, muss die Wait-Queue nach wartenden Threads abgesucht werden, gibt es keine solchen Threads, kann der Scheduler terminiert werden

Wichtiger Hinweis:

Verwenden Sie für die Realisierung der Wait-Queue die vorgegebene Timer-Funktion `getTime()`, die die Zeit in Millisekunden seit dem Programmstart liefert.

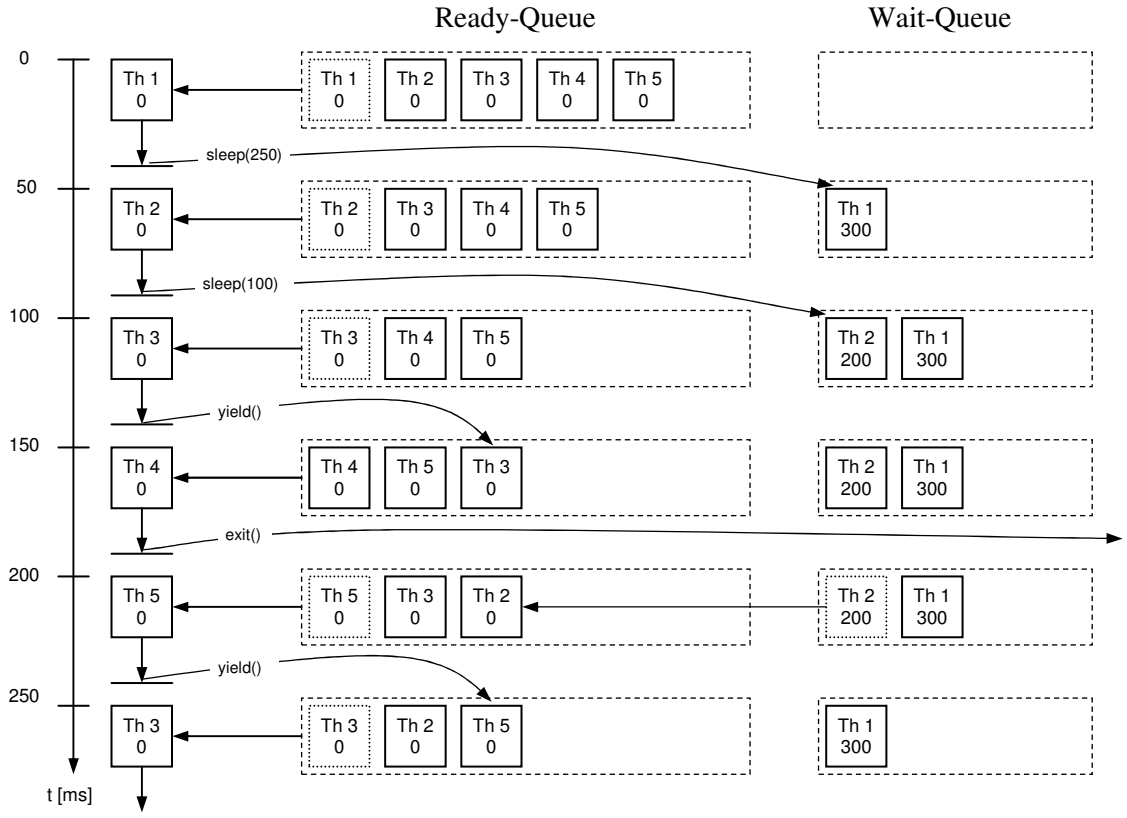


Fig. 2: Ablauf Scheduling mit einer Ready-Queue und einer Wait-Queue

2.3 Der Scheduler

Der Scheduler bildet den Kern des Thread-Paktes und unterstützt `preemptive` und `nonpreemtive` Scheduling. Er ist für das Umschalten zwischen den Threads und für die Auswahl des nächsten Threads verantwortlich. Die Threads sind User-Threads und werden deshalb vollständig innerhalb eines Prozesses verwaltet. Daraus ergeben sich folgende Eigenschaften:

- User-Threads sind einfacher zu implementieren als Kernel-Threads
  - alles spielt sich innerhalb von Benutzerprozessen ab läuft also nur auf einem Core
  - der Kernel muss nicht modifiziert werden
  - bei Fehlern stürzt bestenfalls der Benutzerprozess ab
  - der Kernel weiss nichts von den Threads, was zur Folge hat, dass ein blockierender Thread auch die anderen Threads blockiert
- preemtives Scheduling
  - Threading wird mit `myThreadStrt()` gestartet werden
  - setzen des Preemption Interval beim Startenl: `myThreaStartInterval(x)`, x in µs
  - Preemption wird über UNIX Signale und einen Alarm-Timer gesteuert
- nicht-preemptives Scheduling → nur kooperatives Multithreading möglich
  - Threading wird mit: `myThreaStartFIFO()` gestartet
- folgende Funktionen stehen zudem für kooperatives zur Verfügung :
  - `myThreaYield()`: der Thread unterbricht sich freiwillig
  - `myThreaSleep(x)`: der Thread legt sich für x Millisekunden schlafen
  - `myThreaExit()`: der Thread terminiert
- das Hauptprogramm selbst ist kein Thread (im Gegensatz zu den pthreads)
  - mindestens ein Thread muss im Hauptprogramm mit `myThreadCreate()` erzeugt und in die Ready-Queue eingereiht werden, weitere Threads können anschliessend von diesem Thread erzeugt werden.
  - der Scheduler muss explizit gestartet werden, er terminiert, wenn sich alle Threads beendet haben und kehrt dann ins Hauptprogramm zurück.
  - wenn die Ready-Queue leer ist und in der Wait-Queue keine aktivierbaren Threads vorhanden sind, wird der Scheduler zum *idle*-Thread und wartet, bis ein Thread aktiviert werden kann.

2.3.1 Die Scheduler-Schnittstelle

Der Scheduler muss folgende Funktionen (definiert als Macros) zur Verfügung stellen, über die er aufgerufen werden kann, diese sind in `mythread.h` wie folgt definiert:

```
#define myThreadStart()              taskSwitch(INIT)

#define myThreadStartInterval(x)     initScheduler(PREEMPT, x); \
                                     taskSwitch(INIT)

#define myThreadStartFIFO()          initScheduler(FIFO, 0); \
                                     taskSwitch(INIT)

#define myThreadCreate(t,f,a,p,s)    t = mtNewThread(f,a,p,s); \
                                     initQueues(); \
                                     addToReadyQueue(t)
```

Für die Thread-Kontrolle sind folgende Funktionen definiert:

```
#define myThreadYield()          taskSwitch(YIELD)
#define myThreadExit()           taskSwitch(ZOMBIE)
#define myThreadSleep(x)         taskSwitch(x)
```

Der Scheduler selbst benötigt für die Verwaltung der Threads Funktionen aus dem Modul `queues`:

```
addToReadyQueue(th)          // append thread to ready-queue
addToWaitQueue(th,st)        // sort thread into Wait-Queue (sleeptime)
getNextThread()              // get next thread from queues
checkWaitQueue()             // return number of threads in wait-queue
printQueueStatus()           // print parameters of the ready-queue
printWaitQueue()             // print information of threads in wait-queue
```

Die zwei letzten Funktionen werden nur für Debuggingzwecke benötigt.

2.3.2 Die Thread-Umschaltung

Wie kann zwischen den Threads umgeschaltet werden?

Die Thread Umschaltung zwischen verschiedenen Threads ist im Wesentlichen eine Umschaltung zwischen Funktionen. Allerdings hat jeder Thread einen eigenen Stack, im Gegensatz zu üblichen Funktionen.

Unsere Thread-Umschaltung basiert darauf, wie Funktions- resp. Methodenaufrufe durch den C/C++ Compiler für die x86-Architekteur implementiert werden. Eine wichtige Roller bei diesen Funktionsaufrufen spielen die sogenannten Stack-Frames: Sie werden benutzt, um für jeden Funktionsaufruf die Parameter, die Rücksprungadresse und die lokalen Variablen bereitzustellen. Wir möchten daran erinnern, dass sowohl Parameter als auch lokale Variablen nur innerhalb von Funktionen sichtbar sind, genau das wird durch das Konzept des Stack-Frames garantiert.

In unserem Thread-Paket wird der Stack zudem für jeden einzelnen Thread im Datenbereich über einen genügend grossen Array alloziert, was allerdings zur Folge hat, dass Stack-Overflows abgefangen werden müssen.

Hinweis: Pointer müssen bei 64-Bit Prozessoren in Variablen vom Typ `long unsigend` abgespeichert werden, der Typ `unsigned` ist auf 64-Bit Maschinen nur 32-Bit breit.

Wieso sind Stack-Frames in diesem Zusammenhang wichtig?

Bei der x86-Architektur kann nicht direkt auf den Programmzähler (PC) zugegriffen werden, d.h. es ist nicht möglich, den Programmzähler des unterbrochenen Thteads abzuspeichern und ihn für den nächsten Thread neu zu setzen. Na ja, zumindest nicht direkt: bei einem Funktionsaufruf wird, wie schon erwähnt, die Rücksprungadresse auf dem Stack (im Stack-Frame) abgelegt. Diese Eigenschaft nutzen wir für die Realisierung der Thread-Umschaltung, indem wir zwischen den Stacks der einzelnen Threads umschalten.

Im folgenden werden wir zuerst an einem Beispiel den Aufbau von Stack-Frames repetieren, anschliessend werden wir erklären, wie die Thread-Umschaltung in unserem einfachen Scheduler realisiert ist.

2.3.3 Stack-Frames für die x86 Architektur

Wir gehen von untenstehendem C-Programm aus. Beachten sie, dass die Funktion **`main`** beim Aufruf wie eine normale Funktion mit den Parametern `argc` und `char* argv[]` behandelt wird.

Beim Aufbau der Stack-Frames spielen zwei Register eine wichtige Rolle:

- **der Stack-Pointer**: Daten werden mit PUSH auf dem Stack abgelegt und mit POP vom Stack gelesen, dabei zeigt er bei der x86-Architektur immer auf den zuletzt geschriebenen Eintrag[1] auf dem Stack und läuft von hohen zu tiefen Adressen
- **der Base-Pointer**: direkter Zugriff auf Daten im Stack ist nur über den Base-Pointer möglich, d.h. der Base-Pointer muss für jeden Stack-Frame nachgeführt werden, auf die Daten kann dann relativ zum Base-Pointer zugegriffen werden (*N*: Wortgrösse in Anzahl Bytes), z.B. `local int j` → `MOV AX, [BP+N]`

Beispiel:

```
int main(int argc,char* argv[]) {
    int i,j;
    i = 5;
    j = 7;
    i = add(i, j);
    ...
}

int add(int s1, int s2) {
    int z;
    z = s1 + s2;
    return(z);
}
```

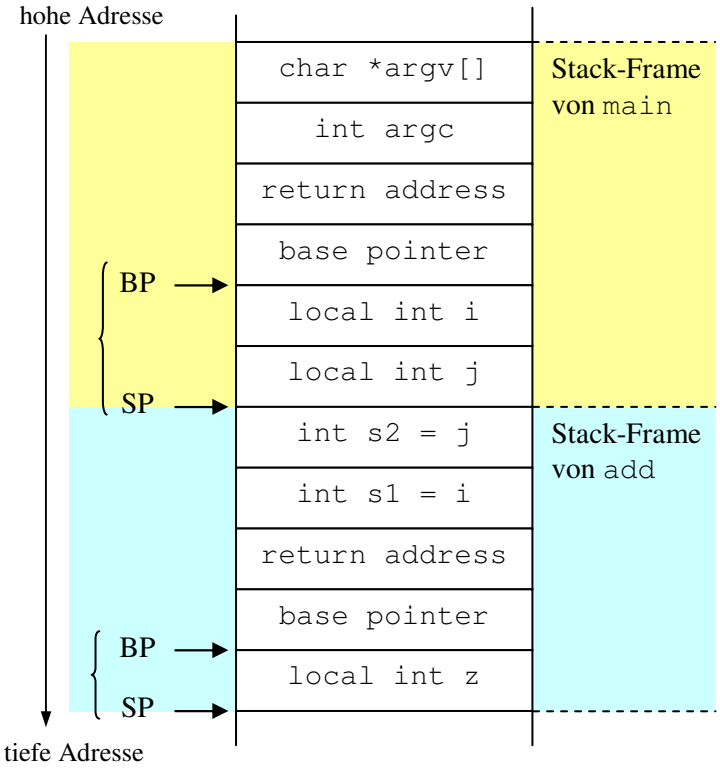


Fig. 3: Beispiel für Stackframes

Beim Aufruf einer Funktion (hier `main`) werden zuerst die Parameter von hinten nach vorne auf den Stack *gepushed* (`char *argv[], int argc`), dann erfolgt der Funktionsaufruf (Assemblerbefehl `CALL`), der automatisch die Rücksprungadresse auf dem Stack ablegt. Anschliessend wird der alte Base-Pointer (Base-Pointer vom vorherigen Stack-Frame) abgelegt und der aktuelle Base-Pointer hinter diesen Eintrag gesetzt, zuletzt wird Platz für die lokalen Variablen reserviert, indem der Stack-Pointer um eine entsprechende Anzahl Einträge nach unten versetzt wird. Beim Aufruf der Funktion `add()` wird analog vorgegangen.

Beim Verlassen einer Funktion wird zuerst der Stack-Pointer an die Position des Base-Pointers gesetzt, der Base-Pointer auf den *alten* Wert gesetzt und anschliessend ins aufrufende Programm zurückgesprungen (Assemblerbefehl `RET`).

Fazit: Ein Stack-Frame ist jeweils vollständig durch das Registerpaar Stack-Pointer , Base-Pointer definiert (siehe Klammern in Fig. 3).

[1] Wir sprechen hier der Einfachheit halber von *Einträgen*: ein Eintrag enthält immer einen Daten- oder Adresswert, unabhängig davon, wie viele Bytes belegt werden

2.3.4 Das Konzept der Thread-Umschaltung

Bei Unterbruch eines Thread muss der gesamte Zustand des Threads, der sogenannte Thread-Kontext abgespeichert werden, um später wieder an der gleichen Stelle fortfahren zu können. Wie schon erwähnt, unterbricht sich ein Thread selber, indem er den Scheduler über eine der Funktion `MyThreadYield()` oder `myThreadSleep()` aufruft. Beim preemtiven Scheduling löst der Alarm-Timer ein Signal aus, das im Signal-Handler die Funktion `myThreadYield()` aufruft.

Zum Thread-Kontext gehören der Inhalt sämtlicher Register (inklusive Stack- und Base-Pointer), die Flags und der Programmzähler. Auf den Programzähler können wir bei der x86-Architektur nicht zugreifen, er ist aber implizit im Stack-Frame gespeichert. Die Register und Flags lassen sich hingegen einfach auf dem Stack ablegen. Der Stack-Frame von `taskSwitch(argument)` hat, nachdem auch noch die Register und Flags auf den Stack gespeichert wurde, folgende Form:

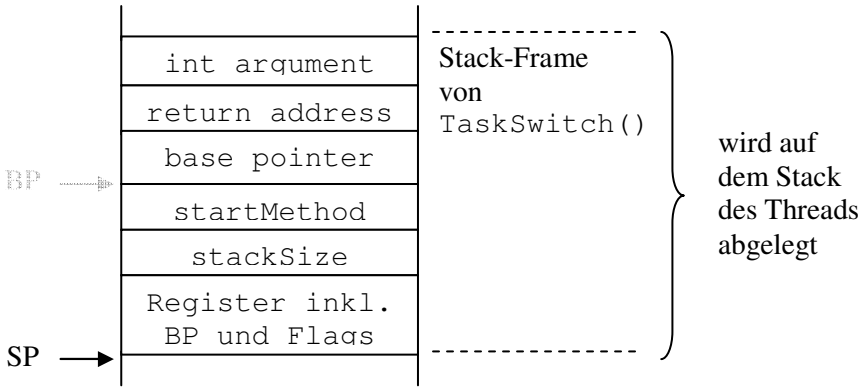


Fig. 4: Stack nach Aufruf von `TaskSwitch()` mit Speicherung der Register und Flags

Da auch der Base-Pointer mit den Registern gespeichert wird, ist der Stack-Frame nun allein durch den Stack-Pointer bestimmt, d.h. beim Umschalten zwischen Threads muss nur der Stack-Pointer abgespeichert werden. Ein Aufruf von `taskSwitch(YIELD)` läuft gemäss Fig. 5 ab:

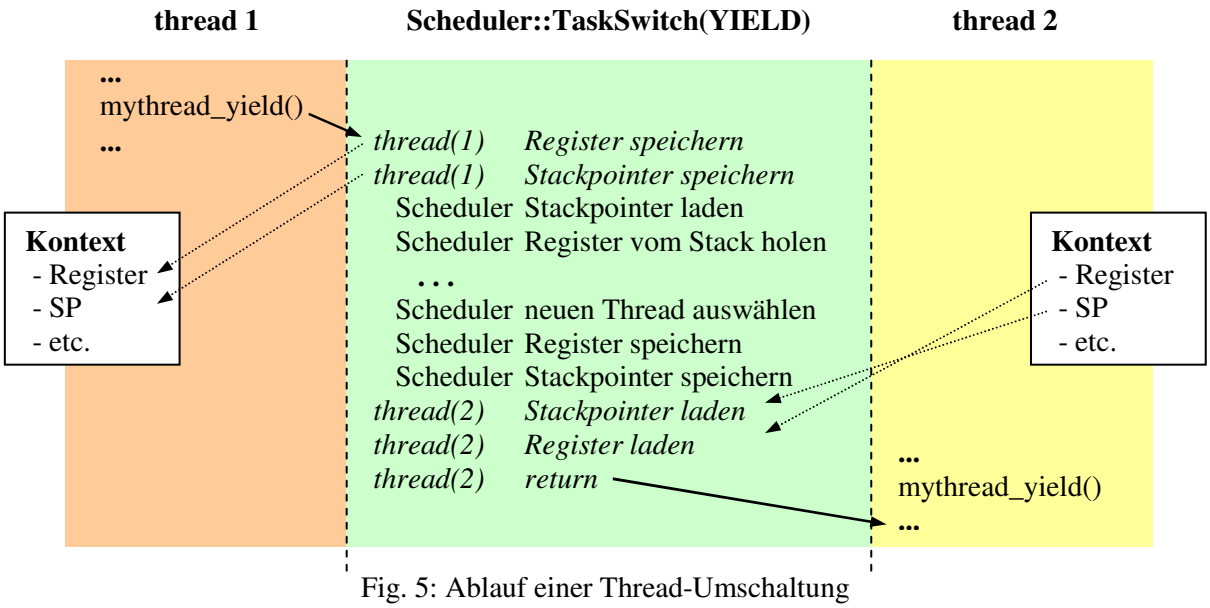


Fig. 5: Ablauf einer Thread-Umschaltung

Hinweis: Die kursiven Zeilen werden auf dem Stack des jeweiligen Threads ausgeführt, die restlichen Zeilen im Stack-Frame des Schedulers.

Beachten sie, dass bei der Thread-Umschaltung der Scheduler für seine Verwaltungsaufgaben einen eigenen Stack-Frame benutzt: das ist übrigens bei jedem Betriebssystem so, der Kernel hat seinen eigenen Stack. Es ist derjenige Stack-Frame, der beim Starten des Scheduler mit dem Aufruf von `taskSwitch(INIT)` erzeugt wurde.

Anmerkungen:

- Der Scheduler wird von allen Threads immer über die Funktion `taskSwitch()` aufgerufen und bei der nächsten Aktiverung des Threads auch wieder mit `return()` verlassen: das läuft auf dem Stack des Threads ab. Der Stack-Frame des Schedulers selbst bleibt immer der gleiche, er wird erst abgebaut, wenn alle Threads abgearbeitet sind.
- Weil der Aufruf von `taskSwitch(argument)` noch im Stack-Frame des Threads geschieht, muss das übergebene Argument als erstes in eine statische Variable kopiert werden, damit nach dem Umschalten auf den Scheduler-Stack der Wert noch verfügbar ist.

2.3.5 Die Implementation des Schedulers

Den (leicht angepasst) Programmcode zum Scheduler finden Sie im Anhang. Hier noch einige Erklärungen und Hinweise:

- der Code in Abschnitt 2.3.6 ist Vergleich zum Code in `MyThreads.tgz` aus Gründen der Leserlichkeit leicht angepasst, die Funktionen für das preemtive scheduling sind weggelassen
- der benötigte Assembler-Code wird automatisch für 32-Bit und 64-Bit CPUs kompiliert, falls ein GNU-Compiler auf einem Linux-System eingesetzt wird, bei anderen Systemen muss unter Umständen der nicht benötigte Code und die bedingte Kompilation auskommentiert werden
- Die Funktion `taskSwitch(argument)` unterscheidet 4 verschiedene Werte resp. Wertebereiche für `argument`:

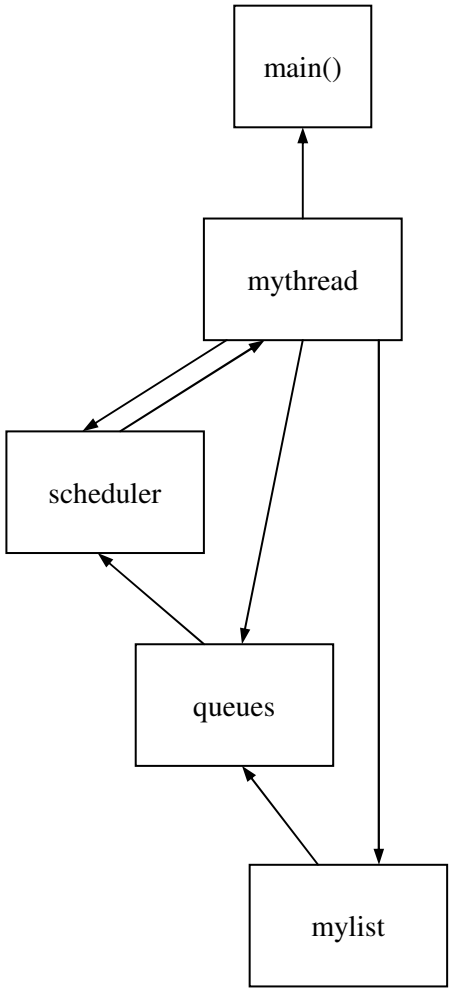
`argument = -2`	`taskSwitch(INIT)`, der Scheduler wird das erste Mal aufgerufen
`argument = -1`	der Thread hat sich beendet mit `myThreadExit()`
`argument = 0`	der gibt freiwillig die Kontrolle ab mit `myThreadYield()`
`argument > 0`	der Thread legt sich für mindestens x Millisekunden schlafen

- Es muss unterschieden werden, ob der Thread das erste Mal läuft oder nicht. Beim ersten Mal muss die entsprechende Benutzerfunktion aufgerufen werden, nachher muss nur der Stack entsprechend umgehängt werden.
- Ein terminierter Thread wird einfach gelöscht. Als *aktiver* Thread ist er ja nicht mehr in der Ready-Queue abgelegt. Stellen Sie sicher, dass bei Ihrem Listenpaket die Destruktoren die dynamisch allozierten Speicherbereiche auch korrekt freigeben.
- Das Speichern und Wiederherstellen der Register, Flags und des Stack-Pointers muss mit Assemblerbefehlen realisiert werden. Diese Assemblerbefehle sind als Macros definiert. Beachten sie, dass der GNU-C/C++ Compiler den AT&T Assembler benutzt, bei dem, im Gegensatz zum Intel-Assembler, die Operanden von links nach rechts kopiert werden:

AT & T	Intel
`MOVL TMP_SP, %esp`	`MOVL esp, TMP_SP`

2.4 Abhängigkeiten zwischen den Programmmodulen

Das Thread Paket ist so strukturiert, dass das Hauptprogramm nur das Header-File `mythreads.h` einbinden muss. Deshalb sind die Funktionen `myThreadYield()`, `myThreadSleep()`, etc., die die Funktion `taskSwitch()` aufrufen in `mythreads.h` definiert. Damit ergibt sich folgende Abhängigkeitsdiagramm:



2.5 Anhang: scheduler: Programmcode: nur Funktion taskSwitch()

Hinweis: der Scheduler benötigt Assemblerbefehle für das Umschalten des Stacks. Der zur Verfügung gestellte Quellcode funktioniert unter Linux und mit dem GNU-C/C++ Compiler für 32-Bit- und 64-Bit-Architekturen (bedingte Compilation). Bei anderen Betriebssystemen und Compilern muss ev. der entsprechende Assemblercode direkt ausgewählt werden. Der Assemblercode für die 64-Bit Architektur ist in einem File abgelegt (zuviel Code), implementiert aber die gleichen Makros wie der 32-Bit Code.

```
//---------------------------------------------------------------------------
// Function TaskSwitch
// executes the thread switching
//
// Arguments
//
//     INIT:   first call of scheduler
//     ZOMBIE: active thread terminates
//     YIELD:  active thread yields
//     n > 0:  active thread waits for n clocks

void taskSwitch(int argument)
{
    TMethod   startMethod;              // start method of a thread
    int       stacksize;                // current stack size

    if (schedType == PREEMPT) {         // stop preemting
        stopPreempting();
        stopTimer();
    }

    schedArg = argument;            // save argument to static variable

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

    switch (schedArg)    {
        case INIT:
            if (schedulerState == 0)
                initScheduler(PREEMPT, DEF_TIMER_INT); // init with defaults
            break;
        case ZOMBIE :                       // 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)
                    addToReadyQueue(activeThread);
            else
                    addToWaitQueue(activeThread, schedArg);
            break;
    }


    /* --- debug ----------------------------------------------------------- */
    if (DEBUG_OUTPUT) {printf("\n\t-> Ready queue\n"); printReadyQueueStatus();}
```

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

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

    if ((activeThread == NULL) && (checkWaitQueue())) {
        while ((activeThread = getNextThread()) == NULL) {
            usleep(1000);
            if (shutdown)
                break;
        }

    }

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

        /* --- debug ------------------------------------------------------------*/
        if (DEBUG_OUTPUT) { printf("\n\t-> Scheduler starting thread: \n\t");
                            mtPrintTCB(activeThread); }

        /* --- 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
            }
            startMethod = mtGetStartMethod(activeThread);   // get start method
            startPreempting();
            startTimer();
            startMethod(mtGetArgPointer(activeThread));     // call start method
        }
        else {
            SAVE_REGS();                            // scheduler: save regs
            SAVE_SP(scheduSP);                      // scheduler: save stackptr
            RESTORE_SP(threadSP);                   // thread:    set stackptr
            RESTORE_REGS();                         // thread:    restore regs
            startPreempting();                      // enable preempting
            startTimer();                           // start timer
        }
    }

    return;
}
```