Threads

Prof. Dr.-Ing. Andreas Heil

v1.0.0

Wiederholung

- Bisher gelernt
  - Virtualisierung einer CPU – also mit »Scheduling« den Anschein erwecken, als gäbe es viele CPUs
  - Virtualisierung von Speicher – also mittels »Adressräumen« den Anschein erwecken, als hätte jedes Programm seinen ganz eigenen Speicher
  - Dabei galt implizit immer die Annahme, dass ein Prozess aus einem einzigen Ausführungsstrang besteht

Was sind Threads?

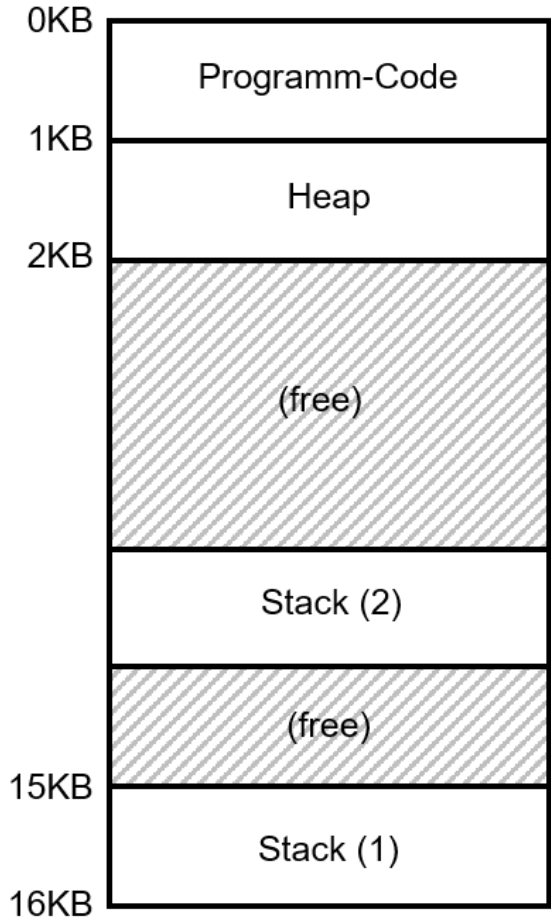
- Anstelle nur eines Ausführungsstrangs besitzen »multi-threaded« Programme mehrere Ausführungsstränge
- Jeder dieser Ausführungsstränge (engl. thread) kann grundsätzlich wie ein Prozess verstanden werden
- Ein wichtiger Unterschied: Während jeder Prozess seinen ganz eigenen Adressraum besitzt, teilen sich Threads einen Adressraum und können auf dieselben Daten zugreifen

Eigenschaften von Threads

- Threads werden grundsätzlich wie Prozesse behandelt
  - Benötigen einen Programm Counter (PC) für die nächste Instruktion
  - Haben eigene Register
  - Laufen zwei Threads auf einer CPU, muss ein Context-Switch stattfinden
  - Für den Context Switch wird ein *Thread Control Block* (TCB) analog zum Process Control Block (PCB) benötigt
  - Wichtiger Unterschied: Der Adressraum bleibt beim Context-Switch von Threads der gleiche

Multi-Thread-Adressräume

Threads laufen unabhängig, rufen eigene Routinen mit eigenen lokalen Variablen und Rücksprungadressen auf und benötigen daher dedizierte Stacks (engl. thread-local storage)



Warum überhaupt Threads verwenden?

Warum überhaupt Threads verwenden?

Offensichtlich: Parallelisierung, sobald mehrere CPUs bereitstehen, können Aufgaben gleichzeitig durchgeführt werden

Etwas subtiler: Würde ein Prozess aufgrund einer I/O-Operation geblockt werden, könnte ein weitere Thread im Prozess weiterlaufen und somit das blockieren des Prozesses vermeiden

Vorteil: Threads greifen auf dieselben Daten innerhalb eines Prozesses zu.

Threads und Determinismus

- Threads laufen nicht deterministisch, da Routinen im Thread unabhängig vom Aufrufer laufen
- Was dann tatsächlich läuft wird durch den Scheduler gesteuert
- Problem verschärft sich noch, da Threads auf dieselben Daten zugreifen

Konsequenz: Nicht-deterministische Programmläufe und Ergebnisse

- OSTEP Kapitel 26.2 zum Erzeugen von Threads (in C)

Scheduling Beispiel

Hier: Erhöhen eines Counters um 1

- Variable für Counter liegt bei Adresse 0x8049a1c
- Wert der Variable wird in Register eax geladen (Zeile 1)
- Wert wird um 1 erhöht (Zeile 2)
- Neuer Wert wird aus Register zurück an Andresse 0x8049a1c geschrieben

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Scheduling Beispiel (Forts.)

Annahme: Zwei Threads, T1 und T2

- T1 führt den Code aus und erhöht den Wert (z.B. 50) um 1
- In Register eax steht nun 51
- Jetzt: Timer Interrupt, Betriebssystem speichert T1
  - Programm Counter
  - Register einschließlich eax
- Nun wird T2 ausgeführt
- T2 lädt den Wert von 0x8049a1c
- In eax steht nun 50 (Jeder Thread hat seine eigenen virtualisierten Register!!!1)
- eax wird erhöht und zurückgeschrieben

Scheduling Beispiel (Forts.)

- Jetzt findet nochmal ein Context-Switch statt und T1 wird wieder ausgeführt
- Alle Register werden geladen
- In eax steht nun 51
- T1 führt nun noch Zeile 3 aus und schreibt 51 an 0x8049a1c
- Was sollte aber eigentlich in unserer Variable stehen?

 OSTEP Kapitel 26.3 für detaillierten Trace des Ablaufs

Race Conditions

- Ergebnis abhängig vom Timing bei der Ausführung von Code: Race Condition
- Führt zu einem nicht-deterministischen Programmfehler, der in der Regel schwer zu finden ist (Ergebnis heißt auf engl. indeterminate)
- Code, der die Race Condition auslösen kann wird kritischer Abschnitt (engl. critical section) genannt
- Greift auf eine gemeinsame Variable (engl. shared variable) bzw. gemeinsame Ressource (engl. shared resource).
- Durch wechselseitigen Ausschluss (engl. mutual exclusion) wird erreicht, dass wenn sich ein Thread in einem kritischen Abschnitt befindet, kein anderer Thread auf diesen Code zugreifen kann…
- Aber wie?

Referenzen