Locked Data Structures

Prof. Dr.-Ing. Andreas Heil

v1.0.0

Wiederholung: Locks

- Durch das Aufrufen einer `lock`-Routine wird versucht Zugriff auf ein ein Lock, also eine Art Token zu erhalten, die das alleinige Ausführungsrecht eines kritischen Abschnitts sicherstellt
  Ruft jemand anderes nun `lock` auf, kehrt diese Methode solange nicht zurück, wie das Lock nicht freigegeben wurde
  Erst wenn der ursprüngliche Aufrufer durch `unlock` das Lock wieder frei gibt, kehrt eine der bisher aufgerufene `lock`-Routinen zurück und lässt die Ausführung des kritischen Abschnitts zu

Anforderungen an ein Lock

- **Mutual Exclusion:** Locks müssen gegenseitigen Ausschluss ermöglichen
- **Fairness:** Hat jeder Thread eine faire Chance das Lock zu erhalten, sobald es wieder freigegeben wird? Anders ausgedrückt: Es muss vermieden werden, dass ein Thread verhungert (engl. starve), wenn er „ewig“ auf das Lock wartet
- **Overhead:** Hält sich der Aufwand für einen einzelnen Thread in Grenzen, um das Lock zu erhalten und wieder freizugeben? Wie verhält es sich mit der Performance, wenn es viele Threads und ggf. mehrere CPUs gibt?

Ansatz 1: Interrupts deaktivieren

Durch Ausschalten der Interrupts, wird Code in einem kritischen Abschnitt nicht mehr unterbrochen

```
void lock() {
  DisableInterrupts();
}
void unlock() {
  EnableInterrupts();
}
```

Pro und Contra deaktivierte Interrupts

Vorteil:

- Einfach

Nachteil(e):

- Privilegierte Operation könnte ggf. missbraucht werden, indem am Anfang vom ganzen Programm `lock und erst am Ende `` unlock` aufgerufen wird
- Funktioniert nicht auf Multi-CPU-Systemen
- Wird, wenn überhaupt, nur vom Betriebssystem intern genutzt, um auf eigene Datenstrukturen zuzugreifen oder um ungünstige Interrupt-Situationen vorzubeugen

Ansatz 2: Einfache Variable

Weshalb kann die Verwendung einer einfachen Variablen als Lock nicht genügen?

Zur Erinnerung: Im POSIX-System verwenden wir auch eine Variable in Kombination mit `lock`- und `unlock`-Routinen. Wie funktioniert das?

Grundidee:

- Einfache Variable (ein Flag) als Lock nutzen
- Aufruf von `lock` testet ob Flag == 1 ist, falls nicht wird dieses auf 1 gesetzt und Thread hat jetzt das Lock
- Ruft nun ein zweiter Thread `lock` auf, wartet er solange bis das Flag wieder 0 ist

Ansatz 2: Kein Gegenseitiger Ausschluss

Thread 1	Thread 2
call `lock()`	
while (flag==1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag==1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // wird jetzt nochmal gemacht	

Ansatz 2: Probleme

- **Problem 1 (Korrektheit):** Durch ungünstige Context-Switches schaffen es beide Threads nun zu laufen, da beide Threads das Flag auf 1 setzen konnten
- **Problem 2 (Performance):** Während ein Thread auf das Lock wartet, prüft er ständig die Variable und verschwendet Rechenzeit des anderen Prozesses z.B. durch Context-Switch, Speicherzugriff, etc. (eng. spin-waiting)

Ansatz 3: Spin Locks mit Test-and-Set

- Hardware-Unterstützung bereits in den 1960ern, die heute noch zum Einsatz kommt
- Test-and-Set bzw. Atomic Exchange
- Liefert den »alten« Wert an der Adresse von `old_ptr`
- Setzt gleichzeitig den Wert an `*old_ptr` auf den Inhalt von new
- Wichtig: Diese Operationen werden atomar aufgerufen!

```
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr; // fetch old value at old_ptr 3
  *old_ptr = new; // store ’new’ into old_ptr
  return old; // return the old value
}
```

Ansatz 3: Funktioniert weil… Szenario 1

- Annahme, Thread ruft `lock` auf und kein anderer Thread hält das Lock (flag==0)
- `TestAndSet` liefert 0, also wird der aufrufende Thread nicht weite prüfen (spinning)
- Das Flag wird außerdem auf 1 gesetzt (atomar!), d.h. kein anderer Thread kann das Lock erhalten
- Ist der Thread fertig, ruft er `unlock` auf und das Flag wird wieder auf 0 gesetzt

Ansatz 3: Funktioniert weil… Szenario 2

- Annahme, Ein anderer Thread hält das Lock (flag==1)
- `TestAndSet` liefert 1, also wird der aufrufende Thread nicht weite prüfen (spinning)
- Das Flag wird außerdem auf 1 gesetzt (atomar!), das ist aber nicht schlimm
- Solange der andere Thread das Lock besitzt, liefert `TestAndSet` immer 1

Ansatz 3: SPinning Lock

- Wir haben das erste funktionierende Lock gebaut
- Spin Lock: Einfachste Variante, dreht sich (engl. to spin) solang unter der Verwendung von CPU Cycles bis das Lock verfügbar wird
- Benötigt einen Preemtive Scheduler (Sie erinnern sich!?), der Threads via Timer unterbricht, damit ein anderer Thread laufen kann ohne Preemtive Scheduling machen Spin Locks auf Systemen mit nur einer CPU keinen Sinn, da der Thread sonst nie die CPU freigeben würde

In x86 z.B. via `xchg` realisiert (x86 Instruction Set Reference^1):

> „… This instruction is useful for implementing semaphores or similar data structures for process synchronization.”

Spin Locks: Kurze Betrachtung

Korrektheit

- Wenn, wie zuvor umgesetzt, stellen Spin Locks sicher, dass nur ein einziger Thread einen kritischen Abschnitt ausführen kann

Fairness

- Nope, es kann sein, dass der wartende Thread niemals in den kritischen Abschnitt kommt, und können somit zum „verhungern“ eines anderen Threads führen

Performanz

- Jeder wartende Thread nutzt eine Zeitscheibe zum Prüfen, daher auf 1-CPU-Systemen nicht sonderlich performant, auf Multi-CPU Systemen ganz OK, solange Anzahl der Threads ~ Anzahl der CPUs

Ansatz 4: Compare-and-Swap

Wird ebenfalls in gängigen Systemen (x86 via `cmpxchg`) eingesetzt[^2]

- Prüft, ob der Wert an Adresse von `ptr` dem Wert von `expected` entspricht
- Falls ja, wird der Wert von `new` gesetzt
- Dann wird der (unveränderte) Wert von `original` zurück geleifert
- Nutzung analog zu Test-and-Swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
  int original = *ptr;
  if (original == expected)
    *ptr = new;
  return original;
}
```

Weitere Ansätze

- Weitere (Hardware-basierte) Ansätze werden z.B. in OSTEP Kapitel 28 behandelt
  - Load-Linked und Store-Conditional
  - Fetch-and-Add
- Problem bisher: Thread, der auf den Lock wartet, verschwendet u.U. ganze Zeitscheiben zum warten
  Bei $N-1$ Threads werden somit N Zeitscheiben verschwendet
- Wie kann also ein Lock entwickelt werden, dass nicht unnötig CPU-Cycles verschwendet?

Lösungsansatz 1: Yield

- Anstelle zu warten, kann ein Thread sich selbst mittels `yield` »ent-schedulen«, d.h. ein anderer Thread wird anstelle dessen geplant
- Funktioniert gut bei zwei Threads
- Was aber, wenn viele Threads um das Lock konkurrieren?
- Dann werden sehr viele Context Switches ausgeführt, und wie wir wissen, lohnen sich diese erst ab einer gewissen Laufzeit

Lösungsansatz 1: Code-Beispiel

```
void init() {
  flag = 0;
}

void lock() {
  while (TestAndSet(&flag, 1) == 1)
    yield(); // give up the CPU
}

void unlock() {
  flag = 0;
}
```

Lösungsansatz 2: Queues verwenden

- Bisher wird alles mehr oder weniger dem Zufall überlassen.
- Der Scheduler entscheidet darüber, welcher Thread als nächstes ausgeführt wird
- Je nachdem welcher Thread ausgewählt wird, wartet dieser auf das Lock oder entscheduled(mittels `yield`) sich, während ein anderer Thread hätte ausgeführt werden können
- In beiden Fällen ist dies reine Verschwendung von Rechenkapazität

Wie kann diese Situation verbessert werden? Hier hilft uns das Betriebssystem!

Referenzen