2018-02-27

# Effiziente Synchronisations- und Sperrverfahren

HARRY FLOHR
Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

# Inhalt

Einleitung und Begriffsklärung ..... 1

Barrier ..... 2

Spinlock ..... 3

Mutex ..... 4

Semaphor ..... 5

Condition variables ..... 7

Effizienzabschätzungen gestallten sich schwierig ..... 8

Fazit ..... 9

# Einleitung und Begriffsklärung

Bevor auf verschiedene Synchronisations- und Sperrverfahren eingegangen wird, soll kurz erläutert werden, warum diese Verfahren benötigt werden. Nebenläufigkeit setzt sich zusammen aus der Aufteilung eines Programmes in Programmabschnitten, welche unabhängig voneinander und somit zeitgleich ausgeführt werden können und der anschließenden Zusammenführung der Abschnitte. Realisiert wird dies über Threads, welche synchronisiert werden müssen. Informationen müssen verteilt und Ergebnisse wieder zusammengeführt werden. Threads können auch bereitstehen, um beim Eintreten einer Bedingung oder Ereignis aktiv zu werden. Somit gibt es verschiedene Gründe, warum Threads auf etwas warten müssen. Sie müssen warten, damit ein Programmabschnitt freigeben wird, eine Bedingung für ihre Fortsetzung eingetreten ist oder alle beteiligten Threads eine verteilte Aufgabe abgeschlossen haben.

Wenn Threads gleichzeitig mit geteilten Ressourcen arbeiten kann es zu Konflikten kommen, welche durch Sperrverfahren vermieden werden sollen. Einen solchen Konflikt beschreibt die Critical section, welche einen Programmabschnitt darstellt, der bei gleichzeitigem Zugriff von mehreren Threads zu unerwarteten oder fehlerhaften Verhalten führen kann. Ein Beispiel ist die Race Condition beim gleichzeitigem Zugriff auf geteilte Ressourcen(http://bigdata-guide.blogspot.de[1]).

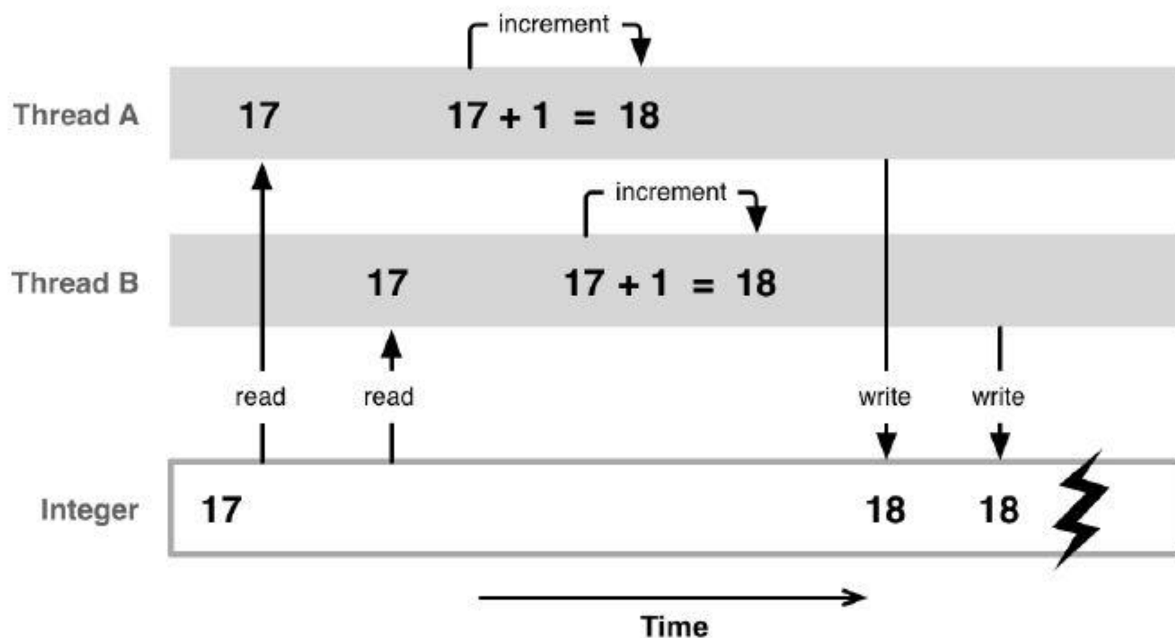


*Abbildung 1: Race Condition*

In Abbildung 1 arbeiten zwei Threads parallel und greifen auf einen gemeinsamen Speicher zu. Beide sollen einen Integer einlesen und je einmal inkrementieren. Wird dieser Programmabschnitt mehrfach aufgerufen, können unterschiedliche Ergebnisse herauskommen. In einigen Fällen hat Thread A seine Inkrementierung abgeschlossen und den Wert 18 in den Speicher geschrieben bevor Thread B den Speicher ausliest. Nachdem Thread B nun auch inkrementiert hat, steht im Speicher das erwartete Ergebnis von 19. Ohne Zugriffssteuerung kann hier jedoch eine Race Condition wie in Abbildung 1 auftreten, indem Thread B aus dem Speicher liest, bevor Thread A das neue Ergebnis ablegen konnte. Nun inkrementiert Thread B ebenfalls nur die 17 zur 18 und schreibt das gleiche Ergebnis wie Thread A in den Speicher. Um diese Race Condition zu vermeiden wird ein Sperrverfahren benötigt, welches Thread B warten lässt, bis Thread A seine Aufgabe mit der geteilten Ressource abgeschlossen hat.

[1] http://bigdata-guide.blogspot.de/2014/01/what-is-race-condition.html (Stand 20.02.2018)

Damit das Sperren oder Freigeben der verschiedenen Verfahren nicht ebenfalls Race Conditions auslösen kann, wird es in einer atomaren Operation gelöst (Love 2005[2]). Eine atomare Operation ist eine Instruktion, welche nicht unterbrochen werden kann. So kann nur ein Thread die Sperrung durchführen und während dieser kann kein anderer Thread eine Sperrung ausführen.

Ein anderer Konflikt bei Sperr- und Synchronisationsverfahren ist das Deadlock (Love 2005[3]). Hier wartet ein oder mehrere Threads auf die Freigabe eines Schlosses, welche aber niemals eintritt. Die häufigste Variante eines Deadlocks ist ein Thread, der eine Ressource gesperrt hat und auf eine andere Ressource wartet, welche ein Thread gesperrt hat, welcher wiederum auf die erste Ressource wartet. So halten beide Threads eine Ressource gesperrt und warten auf den jeweils anderen, geben also die gesperrte Ressource nicht mehr frei. Auch ein einzelner Thread kann in einem Deadlock enden, wenn er versucht eine von ihm gesperrte Ressource erneut zu sperren. Falls dieser Fall nicht von der verwendeten Programmbibliothek abgedeckt ist, wartet nun ein Thread auf die Freigabe einer Ressource, welche er selber gesperrt hält. Zur Vermeidung von Deadlocks sollte die Sperrreihenfolge genau bedacht werden, was in verschachtelten Programmabläufen nicht immer möglich ist. Hier können Funktionen der Programmbibliotheken oder die Vermeidung von Sperren helfen.

Das Warten eines Threads lässt sich über zwei Arten umsetzen. Zum einen kann der Thread aktiv bleiben und in einer Schleife prüfen, ob der Wartegrund noch vorliegt. Diese Warteart wird busy waiting genannt. Dabei gibt der wartende Thread seine Ressourcen nicht frei und muss durch die dauerhafte Prüfung nicht von außerhalb benachrichtigt werden, wenn der Wartegrund wegfällt. Bei der anderen Warteart wird der Thread blockiert und schlafen geschickt(www.linfo.org[4]). Dabei werden alle genutzten Ressourcen des Threads freigegeben und stehen damit anderen Threads zur Verfügung. Bei diesem Context switch muss jedoch der Zustand des blockierenden Threads gespeichert werden, damit dieser beim Aufwecken wieder zu Verfügung steht. Der Zustand setzt sich aus allen für die weitere Ausführung benötigten Informationen zusammen. Das kann z.B. CPU-Register, Stack Pointer und Programm Counter sein. Daraus ergeben sich für beide Wartezeiten ihre Kosten und damit ihre Vor- und Nachteile. Ein busy waiting gibt zwar seine Ressourcen nicht frei, sondern nutzt Systemleistung nur für die wiederholte Prüfung des Wartegrunds, muss dafür aber nicht den teuren Context switch durchführen. Dagegen können die Ressourcen eines blockierten Threads durch den Context switch anderweitig genutzt werden, während der Thread wartet.

## Barrier

Die Barrier ist ein sehr simples aber auch kostspieliges Synchronisationsmittel (Haziza 2009[5]). Wie der Name schon andeutet, definiert die Barrier einen Wartepunkt, an dem Threads auf die Erfüllung der Barrierbedingung warten müssen. Die Barrierbedingung ist eine vorher festgelegte Anzahl an Threads, welche die Barrier erreicht haben müssen, bevor der Programmabschnitt fortgesetzt wird. Erreicht ein Thread die Barrier, wird er blockiert und schlafen geschickt. Durch ihren Aufbau ist die Barrier in den meisten Programmbibliotheken einfach zu nutzen. Es wird ein Aufruf der Barrier am gewünschten

[2] Love 2005, S. 185 ff.
[3] Love 2005, S. 178 ff.
[4] http://www.linfo.org/context_switch.html (Stand 20.02.2018)
[5] Haziza 2009 F. 34 ff

Wartepunkt durch beteiligte Threads benötigt, so wie die einmalige Definition der Anzahl der beteiligten Threads. Das Warten und Wecken löst anschließend die gewählte Programmbibliothek.

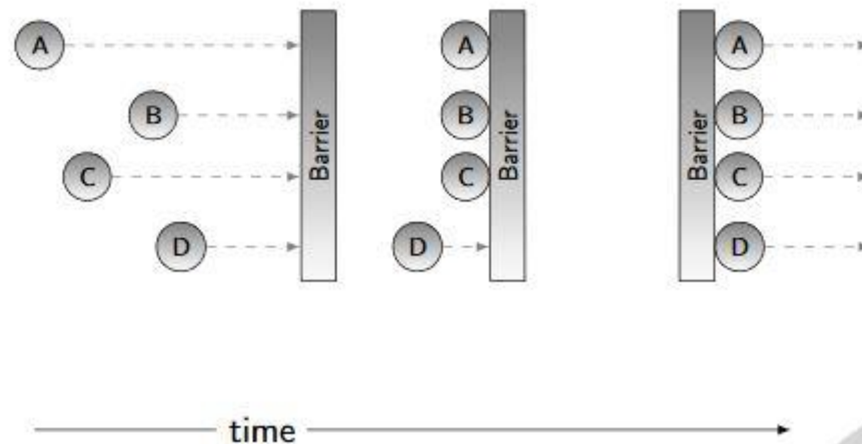


*Abbildung 2: Barrier*

Aufgrund ihrer Funktionsweise kann die Barrier nicht als Sperrverfahren genutzt werden, um den Zugriff auf eine Critical Section zu steuern. Ebenso kann die Barrier nicht als Eventhandler genutzt werden. Sie wird als Synchronisationsverfahren verwendet, um das Ergebnis von auf mehrere Threads verteilte Aufgaben wieder zusammen zu führen oder um den gleichzeitigen Start einer Aufgabe auf verschieden Threads zu garantieren. Die hohen Kosten einer Barrier ergeben sich aus ihrer Funktionsweise. Ein eintreffender Thread wird schlafen geschickt, bis die vorher festgelegte Anzahl an Threads an der Barrier in Warteposition sind. Nun werden alle wartenden Threads aufgeweckt und können ihren Programmabschnitt fortsetzen. Die Wartezeit eines einzelnen Threads wird somit vom langsamsten Thread bestimmt, der die Barrier als letztes erreicht. Außerdem bedeutet das gleichzeitige Aufwecken aller wartenden Threads einen hohen Overhead durch den Context Switch.

## Spinlock

Das Spinlock ist eine Variante des aktiven Wartens (busy waiting) (Love 2005[6]). Ein aktiv wartender Thread wird nicht schlafend geschickt, sondern bleibt in seinem Context. Beim Spinlock wird eine Critical Section mit einem Schloss versehen. Erreicht ein Thread das Spinlock, prüft dieser ob das Schloss gesetzt ist. Ist das Schloss ungesetzt, kann der Thread das Schloss setzen und die Critical Section ausführen. Ist das Schloss schon gesetzt, fängt der Thread wiederholt an zu prüfen ob das Schloss noch gesetzt ist. Verlässt ein Thread die Critical Section, gibt er das Schloss frei und wartende Threads können nun das Schloss setzen.

*Abbildung 3: Spinlock*

[6] Love 2005 S. 192f

Das Spinlock hat durch dieses Funktionsweise zwei größere Nachteile. Zum einen wird durch das aktive Warten Prozessorleistung verschwendet. Das wiederholte Prüfen des Schlosses kostet Leistung und liefert keinen Programmfortschritt. Außerdem sind Spinlocks unfair, d.h. nach Freigabe des Schlosses erhält der erste prüfende Thread Zugang, unabhängig von Wartezeit oder Threadpriorisierung. Diese Nachteile legen das optimale Anwendungsgebiet fest. Verbringen die Threads nur sehr wenig Zeit in der Critical Section oder wird die Section nur selten betreten, ist die erwartete Wartezeit für Threads sehr kurz und ein Spinlock sinnvoll, denn die mit Warten verbrachte Zeit ist kürzer als die benötigte Zeit für einen Context Switch, falls der Thread blockiert wird.

## Mutex

Der Mutex ist, ähnlich wie das Spinlock, ein Sperrverfahren für die Critical Section (Tanenbaum 2009[7]). Der Begriff stammt von mutual exclusion, also dem wechselseitigen Ausschluss. Eine Critical Section erhält ein Schloss und Threads prüfen dieses bei Erreichen der Section. Ist das Schloss ungesetzt, wird es gesetzt und die Section betreten. Ist das Schloss schon gesetzt, wird der Thread schlafen geschickt und wartet auf die Freigabe. Verlässt ein Thread die Critical Section, gibt er das Schloss frei und weckt wartende Threads auf. Welcher Thread aufgeweckt wird, liegt an der verwendeten Programmbibliothek und richtet sich nach Wartezeit oder Threadpriorisierung.

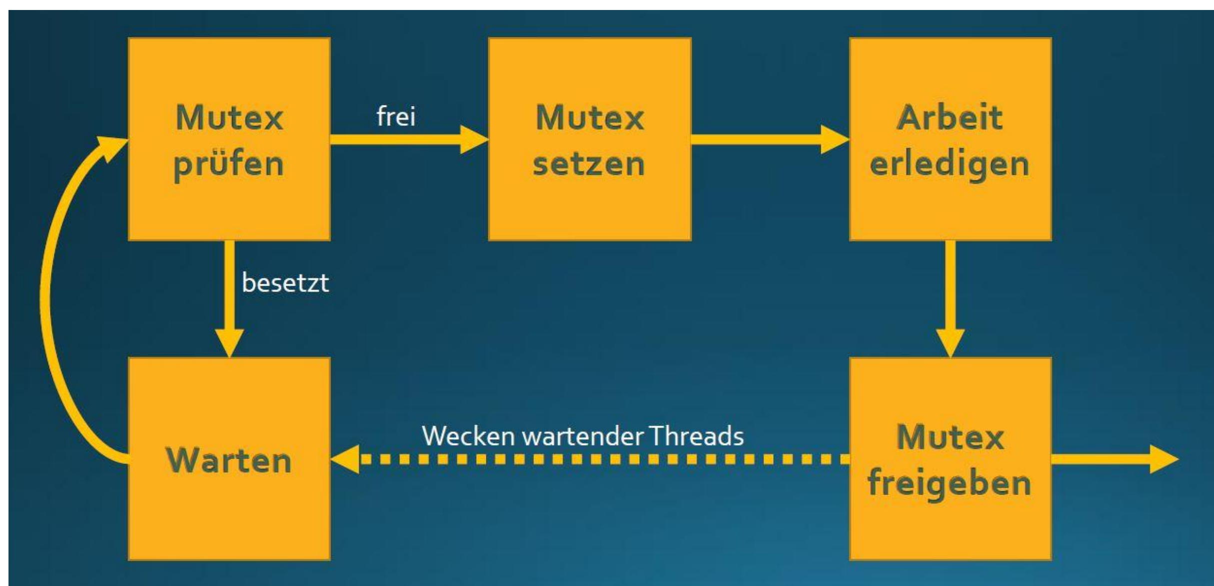


*Abbildung 4: Mutex*

Die effiziente Nutzung von Mutex ist eine Abwägung der Größe des vom Mutex geschützten Bereiches sowie die Häufigkeit der Mutexverwendung. Viele kleine Critical Sections erlauben eine hohe Nebenläufigkeit, aber bringen auch einen erhöhten Overhead durch häufige Context Switch der Threads. Ein Mutex arbeitet mit dem Prinzip eines Schlossbesitzers. Der Thread, der das Schloss gesetzt hat, soll dieses auch wieder freigeben. Dadurch soll ein verfrühtes Freigeben des Schlosses durch andere verhindert werden. Auch soll ein Deadlock vermieden werden, falls der Besitzer erneut versucht das schon gesetzte Schloss zu setzen. Dies würde den Thread schlafen schicken und er könnte das Schloss nicht wieder freigeben, auf dessen Freigabe er nun wartet. Verschiedene Programmbibliotheken bieten unterschiedliche Mutextypen, daher werden hier die vier gängigsten Varianten vorgestellt.

---

[7] Tannenbaum 2009 S. 176f

Der **Fast Mutex** stellt Geschwindigkeit über Genauigkeit indem auf Gültigkeitsprüfungen verzichtet wird. Dadurch können fremde Threads den Mutex freigeben, obwohl sie nicht Besitzer sind. Auch kann der Besitzer versuchen den Mutex erneut zu setzen. Dies liefert einige Fehlerquellen bei unklaren Programmabläufen.

Beim **Errorcheck Mutex** wird Genauigkeit über Geschwindigkeit gestellt und die Gültigkeit jeder Operation geprüft. Dieser Mutex ist besonders für das Debugging nützlich, da spezifische Fehlermeldungen auf falsche Operationen hinweisen. Dies erweitert der **Recursive Mutex**, welcher die gleiche Funktionalität bietet, aber ein mehrfaches Setzen des Mutexes durch den Besitzer zulässt. Dabei wird die Anzahl an Setzungen mitgezählt und zur Freigabe wird die gleiche Anzahl an Freigaben benötigt. Dies ist besonders bei geschalteten Rekursionen hilfreich, bei denen der Programmdurchlauf unklar ist.

Der **Adaptive Mutex** kombiniert den Mutex mit einem Spinlock. Der Mutex hat die gleiche Funktionalität wie ein Fast Mutex, jedoch blockiert ein Thread nicht gleich, wenn er ein gesetztes Schloss erreicht, sondern verbringt einige Zeit in einem Spinlock. Diese Wartezeit ist in Posix keine Konstante, sondern eine immer weiter verfeinerte Abschätzung der erwarteten Wartezeit. Durch das Spinlock ist der Adaptive Mutex günstiger als ein normaler Mutex, wenn er nur kurz geschlossen wird, jedoch erhält er die Unfairness des Spinlocks als Nachteil, da später eintreffende Threads im Spinlock Zugriff erhalten können, obwohl noch schlafende Threads warten.

Die Wahl zwischen Mutex und Spinlock als Sperrverfahren ist abhängig vom System und Programmabschnitt. Die generelle Vereinfachung, das Spinlocks für sehr kurze Sperrungen und Mutex für längere Sperrungen genutzt werden sollten, gestaltet sich schwierig, da in komplexen Programmen die verbrachte Zeit in einer Critical Section nicht immer abschätzbar ist. Aus diesem Grund werden Hybridlösungen wie der Adaptive Mutex angeboten.

## Semaphor

Der Semaphor (Signalträger) ermöglicht durch seinen Aufbau eine Nutzung sowohl als Sperrverfahren als auch zur Eventverarbeitung (Tanenbaum 2009[8]). Dabei besteht der Semaphor aus einem Zähler, welcher beim Erreichen eines Codeabschnitts geprüft wird. Ist der Zähler größer als Null, gilt der nachfolgende Bereich als betretbar. Der Zähler wird dekrementiert und der Codeabschnitt betreten. Ist der Zähler gleich Null, wird der Thread blockiert und auf eine Erhöhung des Semaphors gewartet. Verlässt ein Thread den vom Semaphor geschützten Abschnitt, inkrementiert er den Zähler und weckt wartende Threads. Die Reihenfolge, in der Threads geweckt werden, bestimmt die verwendete Programmbibliothek.

[8] Tannenbaum 2009 S. 173-175

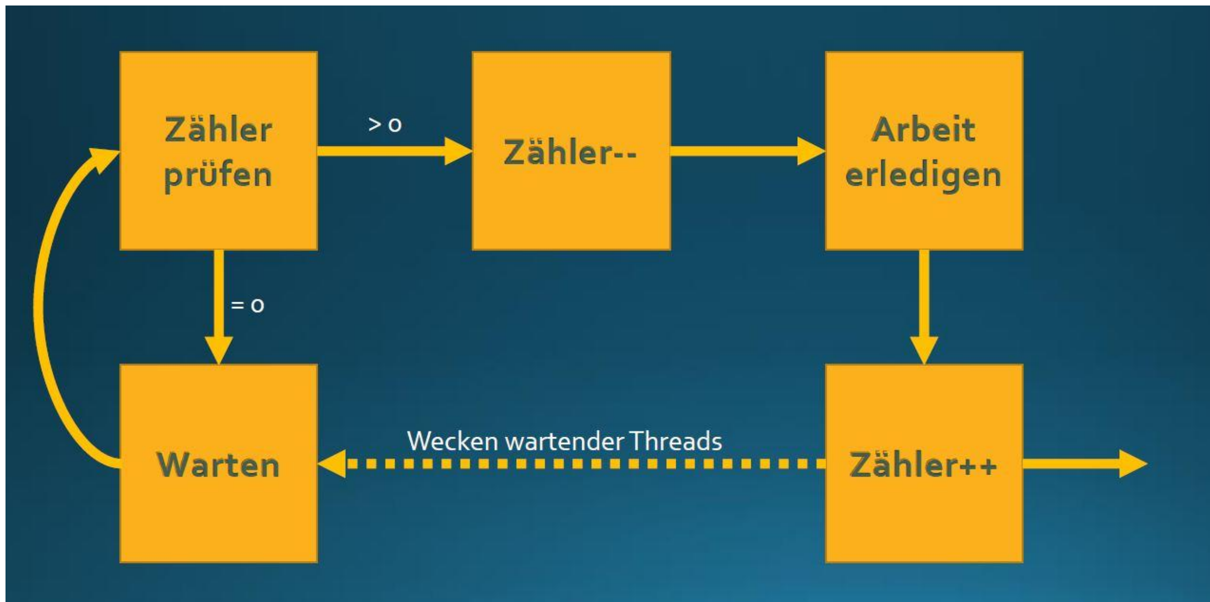


*Abbildung 5: Semaphor*

Die Verwendungsart des Semaphors wird durch den Wert bestimmt, mit dem der Zähler initialisiert wird. Als Sperrverfahren wird der Semaphor mit einem Wert größer Null initialisiert und bestimmt wie viele Threads den nachfolgenden Bereich betreten dürfen. Eine typische Verwendung ist die Zugriffssteuerung auf limitierte Ressourcen, wobei der Zähler mit der Anzahl an gleichzeitig verwendbaren Ressourcen initialisiert wird. Die einfachste Variante ist dabei ein binärer Semaphor, also ein Semaphor bei dem der Zähler nur die Werte Eins und Null annehmen kann. Eins bedeutet die Ressource ist frei und Null bedeutet der Bereich ist gesperrt.

Zur Eventverarbeitung wird der Zähler mit Null initialisiert, der Thread gestartet, der das Event verarbeiten soll und durch den Aufruf des Semaphors blockiert. Nun wartet dieser Thread, bis ein anderer Thread das Event auslöst und dadurch den Semaphor inkrementiert. Dies weckt den Eventverarbeiter, welcher nun den Semaphor dekrementiert und nach der Verarbeitung des Events erneut den Semaphor prüft, um wieder in Warteposition für den Eventeintritt zu gelangen.

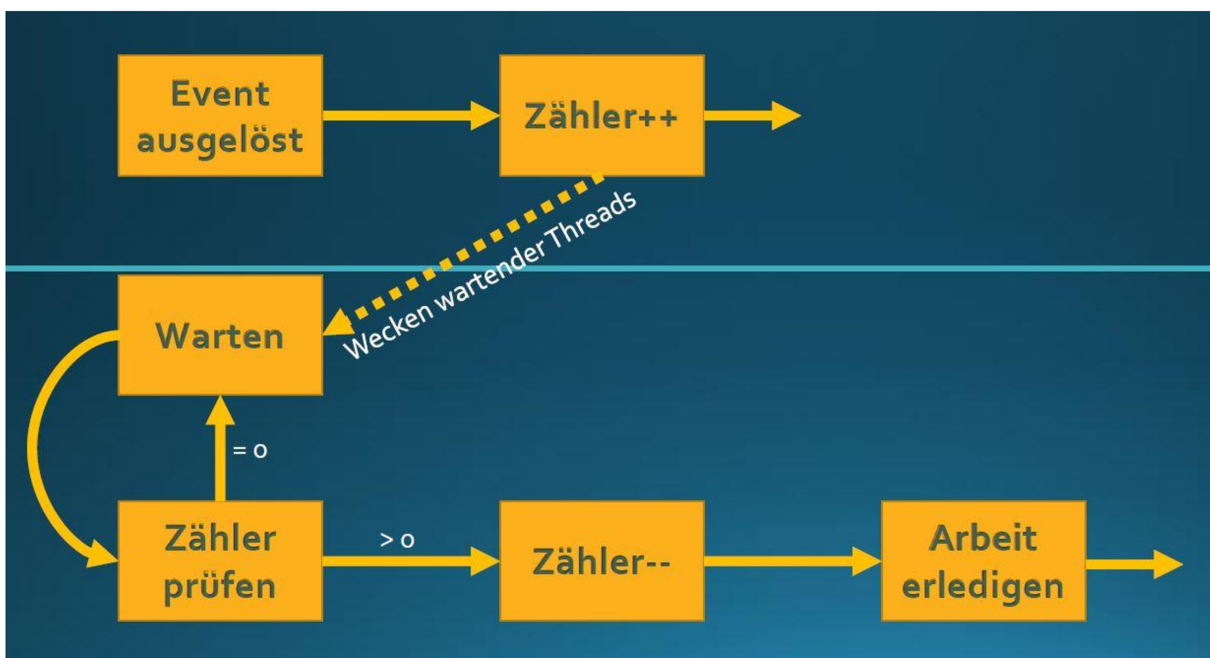


*Abbildung 6: Semaphor als Eventhandler*

Da der Semaphor, wie der Mutex, mit Context Switch arbeitet, gelten für ihn die gleichen Abwägungen für eine effiziente Nutzung. Da der Zähler des Semaphors jedoch besitzlos ist, kann er von jedem Thread inkrementiert oder dekrementiert werden, was eine korrekte Nutzung erschwert. Rekursionen und unklare Programmabläufe können schnell zu einem Deadlock oder der unerwünschten Freigabe des Semaphors führen.

In der Theorie des Besitzers liegt die Unterscheidung zwischen Mutex und Semaphor. Obwohl beide ein ähnliches Verhalten besitzen, liegen ihnen unterschiedliche Verwendungsziele zugrunde. Die Ähnlichkeit wird besonders deutlich, wenn der Fast Mutex mit dem binären Semaphor verglichen wird. Der binäre Zähler des Semaphors kennt hier, wie der Mutex, nur die Zustände „verfügbar" und „besetzt". Ebenfalls fällt im Fast Mutex die Prüfung auf den Besitz des Mutex weg und es kann, wie beim Semaphor, jeder Thread den Mutex freigeben oder sperren. Die Verwendungsziele bestimmen den Unterschied. Der Mutex ist ein Sperrmechanismus zum Schutz einer geteilten Ressource. Hierbei hat der Mutex einen Besitzer, welcher den geschützten Bereich immer in der Reihenfolge Mutex setzen, Bereich bearbeiten, Bereich verlassen und Mutex freigeben durchläuft. Der Semaphor ist ein Signalmechanismus. Er kann entweder den Zugriff auf eine endliche Anzahl an Ressourcen steuern oder zur Eventverarbeitung genutzt werden. Da der Semaphor besitzlos ist, kann ihn jeder Thread erhöhen.

## Condition variables

Bei der Condition variable warten Threads auf das Eintreten einer Bedingung (Tanenbaum 2009[9]). Diese Bedingung kann ein bestimmter Zustand gemeinsam genutzter Ressourcen oder das Eintreten eines Events sein. Häufig warten Threads auf die beendete Arbeit anderer Threads. Um nun nicht durch dauerhaftes Prüfen der Bedingung in ein busy waiting zu verfallen, setzt sich der wartende Thread auf eine Queue und blockiert. Erfüllt ein Thread die Bedingung, so meldet er dies den wartenden Threads. Hierbei kann je nach Programmbibliothek gewählt werden zwischen dem Wecken eines einzelnen Threads oder aller wartenden Threads. Die geweckten Threads können nun ihre von der Bedingung abhängige Arbeit fortsetzen. Um Race Conditions zu vermeiden, wird eine Condition variable durch einen Mutex geschützt. So kann sich die Bedingung nicht ändern, während ein Thread sie gerade überprüft. Auch sollen geweckte Threads zuerst die Bedingung prüfen, bevor sie ihre Arbeit fortsetzen, um eine gültige Bedingung zu garantieren. Dies dient besonders dem Schutz vor „spurious wakeups", welche wartende Threads wecken können, obwohl kein Wecksignal geschickt wurde.

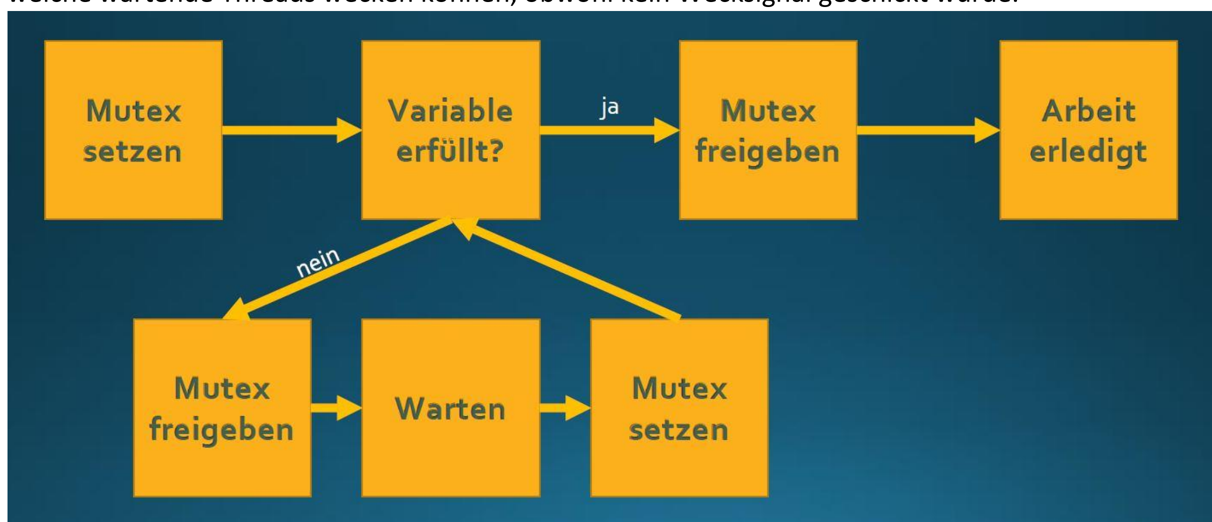


*Abbildung 7: Condition variables*

[9] Tanenbaum 2009 S. 181-187

Da neben dem Overhead der wartenden und geweckten Threads noch der Mutex zum Schutz der Bedingung anfällt, ist die Condition variable ein teures Verfahren. Jedoch liefert der Mutex und das erneute Prüfen der Bedingung eine hohe Robustheit. Durch die Möglichkeit einzelne oder alle wartenden Threads zu wecken ist die Condition variable in ihrer Verwendung flexibel.

## Effizienzabschätzungen gestalten sich schwierig

Eine generelle Abschätzung der Effizienz der verschiedenen Verfahren kann nicht getroffen werden. Ob in einer Situation ein Mutex oder ein Spinlock das effektivere Sperrverfahren ist, ist abhängig vom Programmcode, der genutzten Programmbibliothek, dem Betriebssystem und der Hardware. Ein Beispiel ist eine Versuchsreihe der ArangoDB auf C++11 Basis[10]. Hier wurde auf fünf verschiedenen Systemen gleichzeitiger Datenbankzugriff von 49 lesenden Threads und einem schreibenden Thread simuliert. Getestet wurde das Sperrverfahren Mutex, sowie verschiedene Lockkombinationen und rein atomare Zugriffsfunktionen.

| TESTCASE / OS | WINDOWS 8.1 | GRML LINUX X64 | MAC OS X | LINUX ARM | LINUX X64 |
|---|---|---|---|---|---|
| #0 | 0.033 | 0.0300119 | 0.02503 | 0.18299 | 0.02895 |
| #1 | 479.878 | 5.7003600 | 118.47100 | 21.29970 | 4.47721 |
| #2 | 1.45997 | 4.5296900 | 142.61700 | 23.29240 | 4.72051 |
| #3 | 4.70791 | 6.3525200 | 7.65026 | 23.82040 | 7.87677 |
| #4 | 0.056999 | 0.0454769 | 0.03771 | 0.94990 | 0.035302 |
| #5 | 0.033999 | 0.0263720 | 0.02774 | 0.58076 | 0.017803 |
| #6 | 0.032999 | 0.0286388 | 0.02785 | 0.58125 | 0.017604 |
| #7 | 0.060998 | 0.0528450 | 0.03783 | 0.57926 | 0.033422 |
| #8 | 0.060999 | 0.0536420 | 0.03807 | 0.57851 | 0.033546 |
| #9 | 0.032999 | 0.0299869 | 0.02606 | 0.55443 | 0.017258 |
| #10 | 0.033999 | 0.0235679 | 0.02593 | 0.54985 | 0.017839 |
| #11 | 0.058999 | 0.0534279 | 0.06688 | 0.56929 | 0.030724 |
| #12 | 0.059998 | 0.0676858 | 0.07563 | 0.57466 | 0.036788 |

*Abbildung 8: Effizienzabschätzung*

Hier soll nun nicht die gesamte Testreihe eingegangen werden, sondern lediglich Testcase #1 analysiert werden. In diesem Testcase wurde Mutex als Sperrverfahren genutzt und es ist auffällig wie stark sich die Zeiten zwischen den Betriebssystemen unterscheiden. Besonders für Windows fällt auf wie langsam das System mit dem genutzten Mutex ist. Dies bedeutet aber nun nicht das Mutex generell auf Windows eine schlechte Wahl ist. Die genutzte std::mutex Methode von Microsoft Visual C++

[10] https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/ (Stand 21.02.2018)

generiert ungewöhnlich hohen Overhead unter Windows, da neben dem eigentlichen Sperren und Freigeben noch Sicherheitschecks durchgeführt werden, um die Funktionalität auf älteren Windowsversionen abzusichern (stoyannk.wordpress.com[11]).

## Fazit

Das gewählte Beispiel der Effizienzabschätzung zeigt, dass man nicht ein Verfahren als effizienteste Lösung aufstellen kann, sondern nur am Verwendungszweck eine Abschätzung zur richtigen Auswahl treffen kann. Ob nun eine Sperre nur kurzfristig mit einem Spinlock gesichert werden muss oder aufgrund längerer Wartezeiten ein Mutex von Vorteil ist, ist von Anwendung zu Anwendung unterschiedlich. Da mit steigender Komplexität der Anwendungen eine Einschätzung immer schwerer wird, werden Hybridlösungen wie der adaptive Mutex immer häufiger. Für Wahl des effektivsten Verfahren benötigt man Erfahrung mit der genutzten Programmbibliothek, dem Betriebssystem und der Hardware.

[11] https://stoyannk.wordpress.com/2016/04/30/msvc-mutex-is-slower-than-you-might-expect/ (Stand 21.02.2018)

# Literaturverzeichnis

http://bigdata-guide.blogspot.de/2014/01/what-is-race-condition.html (Stand 20.02.2018)

Love 2005, Linux-Kernel-Handbuch, Leidfaden zu Design und Implementierung von Kernel 2.6, Robert Love, Pearson Deutschland (München 2005)

http://www.linfo.org/context_switch.html (Stand 20.02.2018)

Haziza 2009, Locks and Barriers, Frédéric Haziza, Department of Computer Systems, Uppsala University (Uppsala 2009) https://www.it.uu.se/edu/course/homepage/os2/st09/handout-04.pdf

Tannenbaum 2009, Moderne Betriebssysteme, 3. Auflage, Andrew S. Tanenbaum, Pearson Deutschland (München 2009)

https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/ (Stand 21.02.2018)

https://stoyannk.wordpress.com/2016/04/30/msvc-mutex-is-slower-than-you-might-expect/ (Stand 21.02.2018)

# Abbildungsverzeichnis

Abbildung 1: Race Condition

http://bigdata-guide.blogspot.de/2014/01/what-is-race-condition.html

Abbildung 2: Barrier

https://www.it.uu.se/edu/course/homepage/os2/st09/handout-04.pdf

Abbildung 3: Spinlock

Präsentation „Effiziente Synchronisations- und Sperrverfahren“, Harry Flohr, Arbeitsbereich Wissenschaftliches Rechnen, Universität Hamburg 2017

Abbildung 4: Mutex

Präsentation „Effiziente Synchronisations- und Sperrverfahren“, Harry Flohr, Arbeitsbereich Wissenschaftliches Rechnen, Universität Hamburg 2017

Abbildung 5: Semaphor

Präsentation „Effiziente Synchronisations- und Sperrverfahren“, Harry Flohr, Arbeitsbereich Wissenschaftliches Rechnen, Universität Hamburg 2017

Abbildung 6: Semaphor als Eventhandler

Präsentation „Effiziente Synchronisations- und Sperrverfahren“, Harry Flohr, Arbeitsbereich Wissenschaftliches Rechnen, Universität Hamburg 2017

Abbildung 7: Condition Variables

Präsentation „Effiziente Synchronisations- und Sperrverfahren“, Harry Flohr, Arbeitsbereich Wissenschaftliches Rechnen, Universität Hamburg 2017

Abbildung 8: Effizienzabschätzung

https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/