Osłabione modele pamięci, `<atomic>` w C++11

Seminarium ASK 2017
Jakub Piecuch

Czego oczekujemy od sprzętu?

- Programista niskopoziomowy oczekuje łatwości rozumowania na temat programów
  - Łatwo przewidzieć wynik programu patrząc na kod
- Wszyscy inni oczekują, żeby maszyna działała szybko
- Jak mają się do siebie te dwa oczekiwania?

Przewidywalność programów

W przypadku programów jednowątkowych:

- Oczekujemy że wynik będzie **identyczny** z tym otrzymanym przez **sekwencyjne** wykonanie wszystkich instrukcji (jedna po drugiej) **w kolejności programu**, w szczególności:
  - odczyt z pamięci zawsze powinien zwrócić wartość ostatniego (w kolejności programu) zapisu do tego samego miejsca w pamięci

W przypadku programów wielowątkowych na systemie z jednym procesorem:

- Wynik powinien być identyczny z tym otrzymanym przez **sekwencyjne wykonanie dowolnego przeplotu** instrukcji ze wszystkich wątków, gdzie instrukcje pojedynczego wątku **następują po sobie w porządku programu**
  - odczyt z pamięci zawsze powinien zwrócić wartość ostatniego zapisu do tego samego miejsca w pamięci **w kolejności przeplotu**

Czego oczekujemy od systemów wieloprocesorowych?

Jak przyspieszyć wykonanie programów (i przy tym zachować ich poprawność)?

- Wielopoziomowe pamięci cache
  - znacznie zmniejszają opóźnienia dostępów do pamięci
  - nie wpływają na poprawność programów jednowątkowych i wielowątkowych na systemach UP
  - problem w SMP: co jeśli różne procesory mają w cache różne wartości tej samej komórki pamięci?
- Wykonywanie instrukcji (w tym dostępów do pamięci) poza porządkiem programu
  - niezależne od siebie instrukcje można wykonywać w dowolnej kolejności bez naruszania poprawności (w UP)
  - ukrywa opóźnienia zapisów i odczytów pozwalając no rownoległe przetwarzanie wielu operacji naraz i pozwalając odczytom na przeczytanie wartości która jeszcze nie dotarła do pamięci (write buffering)
  - problem w SMP: mogą istnieć zależności pomiędzy instrukcjami z różnych wątków o których procesor nie wie!
- Wniosek: skoro te optymalizacje zachowują poprawność w UP, to w SMP też tak będzie... prawda?

Co może pójść nie tak: sekcja krytyczna

Na początku a = b = 0

```
P1:

a = 1;
if b == 0 {
   // Sekcja krytyczna
   a = 0;
} else { ... }
```

```
P2:

b = 1;
if a == 0 {
      // Sekcja krytyczna
   b = 0;
} else { ... }
```

- Czy możliwa jest sytuacja, w której P1 i P2 będą w sekcji krytycznej w tym samym momencie?
- Z punktu widzenia P1, zapis do `a` i odczyt `b` to niezależne instrukcje!
  - P1 może odczytać b zanim zapisze do a
- U P2 sytuacja jest taka sama!
- Może się zdarzyć że P1 odczyta b = 0 i P2 odczyta a = 0

Co może pójść nie tak: producent-konsument

Na początku wszystkie wskaźniki = NULL

```
P1:

while (<there are more tasks>) {
    task = get_from_free_list();
    task->data = …;
    <insert task into task queue>
}
head = <head of task queue>
```

```
P2:

while (my_task == NULL) {
    <begin critical section>
    if (head != NULL) {
        my_task = head;
        head = head->next;
    }
    <end critical section>
}
... = my_task->data;
```

- Załóżmy że P2 ma kopię zmiennej head w swoim prywatnym cache
- Nawet jeśli P1 zmieni wartość head to nie mamy gwarancji że P2 zobaczy tę zmianę!

Problem z SMP: out-of-order

- Założenie na którym opiera się technika out-of-order (czyli to, że instrukcje pomiędzy którymi nie ma zależności danych lub kontroli mogą zostać wykonane w dowolnej kolejności względem siebie) nie jest prawdziwe w kontekście SMP
  - Wątki mogą używać współdzielonych danych do komunikacji/synchronizacji
  - Nie zmienia to faktu że większość programu wątki nie komunikują się tylko operują na własnych, prywatnych danych
- W systemach UP programista nie ma sposobu na wymuszenie kolejności dostępów do pamięci bo po prostu nie ma takiej potrzeby
  (chyba że mamy MMIO)
  - Jeśli chcemy pozwolić na przestawianie kolejnością dostępów do pamięci to musimy udostępnić mechanizm pozwalający programiście wymusić kolejność tam gdzie jest ona ważna.
  - Programista musi wiedzieć gdzie wymuszenie kolejności jest konieczne

Problem z SMP: out-of-order

Są dwa rozwiązania:

- Wymusić, aby wszystkie dostępy do pamięci były wykonywane w kolejności programu
  - Co znaczy “wykonywane”?
  - To dosyć mocne ograniczenie, może znacznie zmniejszyć korzyści wynikające z wykonywania instrukcji poza porządkiem programu
  - Z drugiej strony, życie programisty jest łatwiejsze
- Wyspecyfikować, w jakich sytuacjach jakie dostępy do pamięci mogą zostać zamienione kolejnością przez procesor i udostępnić mechanizm pozwalający programiście wymusić kolejność dostępów
  - Pozwala na zastosowanie bardziej agresywnych optymalizacji
  - Pisanie poprawnych programów jest dużo trudniejsze

Innymi słowy, zdefiniować **model pamięci**.

Problem z SMP: cache

- Różne procesory mogą w tym samym czasie widzieć różne wartości w tej samej komórce pamięci
- Trzeba w jakiś sposób koordynować dostęp wielu procesorów do tej samej linii cache
  - Informacje o zmianie danej linii muszą w jakiś sposób dotrzeć do wszystkich procesorów przechowujących u siebie tę samą linię
  - Dostępy wszystkich procesorów do jednej komórki pamięci muszą być globalnie uporządkowane

Tą koordynacją zajmuje się **protokół spójności pamięci podręcznej**.

Co to model pamięci?

Model pamięci:

- specyfikuje w jakiej kolejności operacje w programie będą widoczne dla **zewnętrznych obserwatorów** (np. pozostałe procesory, urządzenia wejścia-wyjścia)
- specyfikuje inne nieintuicyjne sytuacje mogące się wydarzyć (np. brak natychmiastowości zapisów)
- zapewnia mechanizmy umożliwiające programiście wymuszenie kolejności, w jakiej instrukcje będą obserwowane tam, gdzie bez nich byłoby to niemożliwe

Model pamięci UP

- Wszystkie dostępy do pamięci są natychmiastowe i wykonywane w kolejności programu
  - To ma jedynie widzieć programista, w rzeczywistości kolejność dostępów może być inna
- Odczyt z pewnego miejsca w pamięci zawsze zwraca wartość ostatniego (w porządku programu) zapisu do tego miejsca
- Łatwy do zrozumienia dla programisty
- Pozwala na wykorzystanie optymalizacji zwiększających wydajność

Jak rozszerzyć tę ideę do systemów SMP?

- Jak zinterpretować “ostatni” zapis do danego miejsca w pamięci?
  - Ostatni pod względem fizycznego czasu wysłania żądania do kontrolera pamięci?
  - Co jeśli komórka jest w cache i w ogóle nie wysyłamy takiego żądania?

Spójność sekwencyjna *(Sequential Consistency)*

- Intuicyjne rozszerzenie modelu UP
- Wynik programu jest taki sam jak wynik otrzymany przez wykonanie w którym operacje wszystkich procesorów wykonane są w jakimś globalnym sekwencyjnym porządku oraz operacje każdego pojedynczego procesora następują po sobie w porządku programu
- To daje nam dwie gwarancje:
  - Efekty operacji na pamięci każdego procesora są obserwowane przez resztę systemu w porządku programu
  - Zapisy do pamięci wydają się natychmiastowe (nie może dojść do sytuacji w której jeden procesor widzi nowo zapisaną wartość a drugi nie)
- W modelu SC program wielowątkowy wykonywany na wielu procesorach zachowuje się tak jakby się wykonywał na maszynie UP
- Łatwo rozumować o działaniu równoległych programów

Model SC: sekcja krytyczna

```
Na początku a = b = 0
```

```
P1:
a = 1;
if b == 0 {
   // Sekcja krytyczna
   a = 0;
} else { ... }
```

```
P2:
b = 1;
if a == 0 {
      // Sekcja krytyczna
   b = 0;
} else { ... }
```

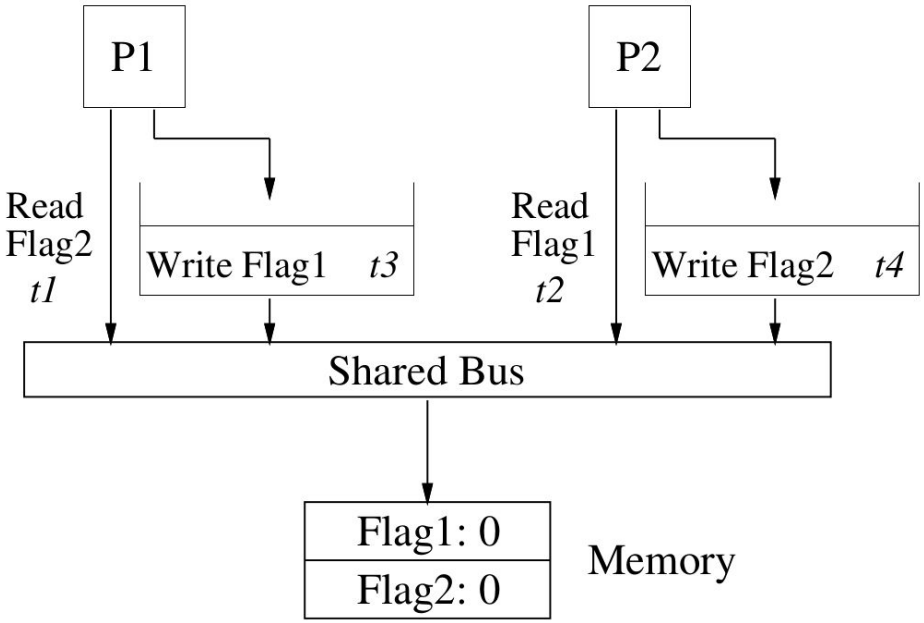
- Mamy gwarancję że najwyżej 1 wątek wejdzie do sekcji krytycznej
- Załóżmy że P1 odczytał b = 0:
  - To oznacza że wykonał się zapis a = 1 i nie wykonał się zapis b = 1
  - Jeśli P2 odczyta a = 0 to będzie to po wyjściu P1 z sekcji krytycznej

Optymalizacje sprzętowe a SC

- SC nie współgra dobrze z technikami ukrywania opóźnień operacji na pamięci
- Dla prostoty załóżmy system SMP bez pamięci podręcznej, w którym dostępy do pamięci wykonywane są w kolejności programu
  - Znowu, co znaczy “wykonywane”?

Write buffer

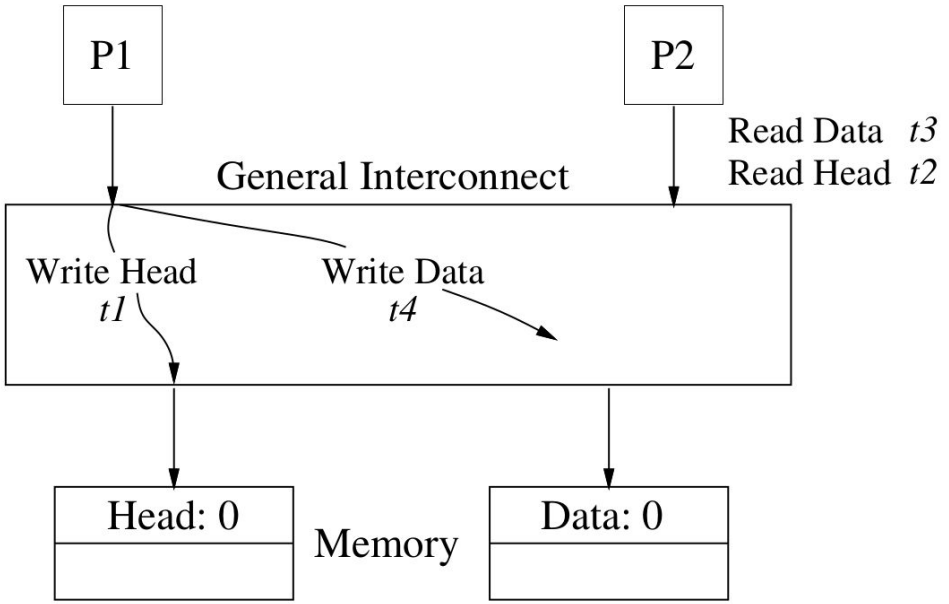
- Przy zapisie do pamięci wpisujemy wartość do bufora i pozwalamy wykonywać się następnym instrukcjom
- Bufor jest lokalny dla procesora, inni go nie widzą
- Odczyty obsługujemy patrząc w pierwszej kolejności do bufora, potem do pamięci
- Konsekwencja: złamana kolejność W → R, psuje SC :(



P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Zachodzące na siebie zapisy *(Overlapping Writes)*

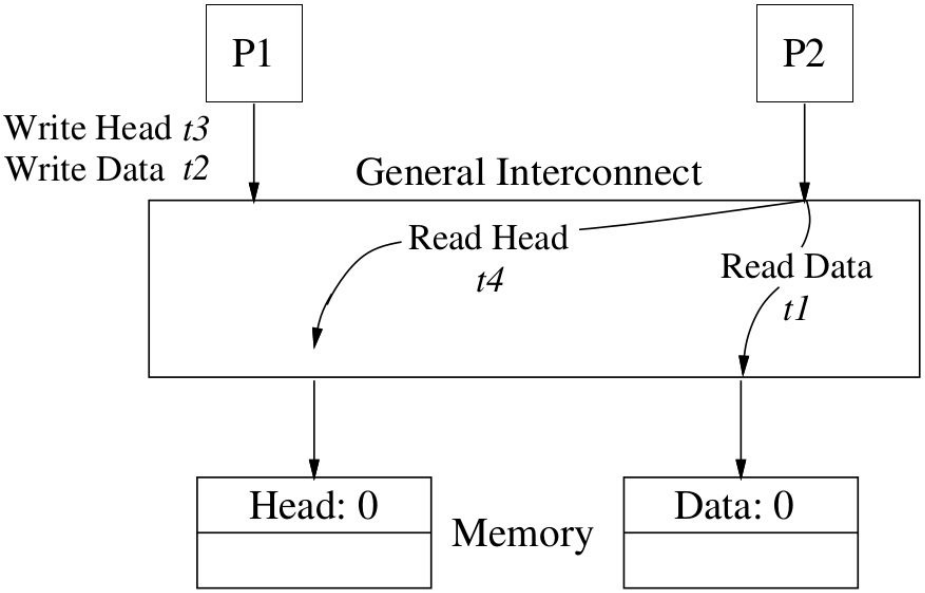
- Nie czekamy na zakończenie się zapisu do pamięci przed rozpoczęciem kolejnego
- Może się np. zdarzyć że poprzednie żądanie będzie opóźnione przez sieć
- Zapisy mogą się wykonać poza porządkiem programu
- Łamiemy kolejność W→W, a tym samym SC :(



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Zachodzące na siebie odczyty *(Overlapping Reads)*

- Przy odczycie z pamięci nie czekamy na wartość i kontynuujemy wykonywać niezależne instrukcje
- Łamiemy kolejność R→W i R→R



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Problemy z SC

- Praktycznie niemożliwa jest efektywna implementacja modelu SC
- Czy na pewno potrzebujemy tak mocnych gwarancji?
- Pokusa optymalizacji sprzętowych jest wielka, zwłaszcza jeśli produkt ma się sprzedać (szybszy = lepszy)
- Może programiści zgodziliby się rozluźnić ograniczenia w zamian za szybsze maszyny?
    - Ale w sumie i tak bardziej liczy się performance

Osłabione modele pamięci

SC charakteryzują 2 warunki:

- Dostępy do pamięci z każdego procesora są wykonywane w kolejności programu
  - “są wykonywane”, czyli obsługiwane przez kontroler pamięci
- Operacje na pamięci są postrzegane przez wszystkie procesory w tej samej kolejności (natychmiastowość operacji względem innych)

Osłabione modele pamięci osłabiają (mniej lub bardziej) przynajmniej 1 z tych warunków.

Możliwe osłabienia modelu SC:

Osłabienie warunku porządku programu:

- W→R (może wykonać odczyt przed poprzedzającym zapisem)
- W→W (może wykonać zapis przed poprzedzającym zapisem)
- R→RW (może wykonać zapis/odczyt przed poprzedzającym odczytem)

Osłabienie warunku natychmiastowości operacji:

- Procesor może odczytać wartość zapisaną przez inny procesor zanim wszystkie pozostałe procesory unieważniły starą wartość

Osłabienie obu warunków:

- Procesor może odczytać wartość własnego zapisu zanim pozostałe procesory unieważniły starą wartość

Porównanie osłabionych modeli pamięci (Gharachorloo & Adve, 1995)

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Wymuszanie porządku na dostępach do pamięci

- Każda architektura implementująca osłabiony model pamięci musi udostępnić mechanizm pozwalający programiście na wymuszenie porządku programu na dostępach do pamięci
- Większość robi to poprzez specjalne instrukcje zwane barierami pamięciowymi *(memory barriers/fences)*
- Np. x86-64: LFENCE, SFENCE, MFENCE
  - Zachowany jest porządek R→LFENCE, LFENCE→RW, W→SFENCE, SFENCE→RW, RW→MFENCE, MFENCE→RW
- Analogiczne funkcje w jądrze Linuxa: `rmb()`, `wmb()`, `mb()`
  - Zachowany jest porządek R→`rmb()`, `rmb()`→R, W→`wmb()`, `wmb()`→W, RW→`mb()`, `mb()`→RW

Osłabienie kolejności W→R

- Pozwala na korzystanie z write buffera

Mamy 3 warianty w zależności od tego jak bardzo osłabiamy pozostałe warunki SC:

- IBM 370
  - Odczyt nie może zwrócić wartości zbuforowanego zapisu dopóki ten zapis nie jest widoczny dla wszystkich procesorów
- TSO (Total Store Ordering)
  - Odczyt może zwrócić wartość zbuforowanego zapisu zanim ten zapis stanie się widoczny dla innych procesorów
  - Zaimplementowany np. w x86-64
- PC (Processor Consistency)
  - Odczyt może zwrócić wartość zapisu dowolnego procesora zanim ten zapis stanie się widoczny dla wszystkich procesorów

Różnice pomiędzy IBM 370, TSO i PC

```
Na początku a = flag1 = flag2 = 0
```

```
P1:                          P2:

flag1 = 1;                   flag2 = 1;
a = 1;                       a = 2;
register1 = a;               register3 = a;
register2 = flag2;           register4 = flag2;
```

Wynik: `register1 = 1, register2 = 0, register3 = 2, register4 = 0`

- Niemożliwy na IBM 370, możliwy w modelach TSO i PC

Różnice pomiędzy IBM 370, TSO i PC

Na początku a = b = 0

```
P1:                 P2:                 P3:

a = 1;              if (a == 1)
                       b = 1;
                                        if (b == 1)
                                           register1 = a;
```

Wynik: `b = 1, register1 = 0`

- Niemożliwy na IBM 370 i w modelu TSO, możliwy w modelu PC

Przykład użycia barier: sekcja krytyczna

Na początku a = b = 0

P1:

```
a = 1;
mb();
if b == 0 {
   // Sekcja krytyczna
   a = 0;
} else { ... }
```

P2:

```
b = 1;
mb();
if a == 0 {
      // Sekcja krytyczna
   b = 0;
} else { ... }
```

Osłabienie kolejności W→R i W→W

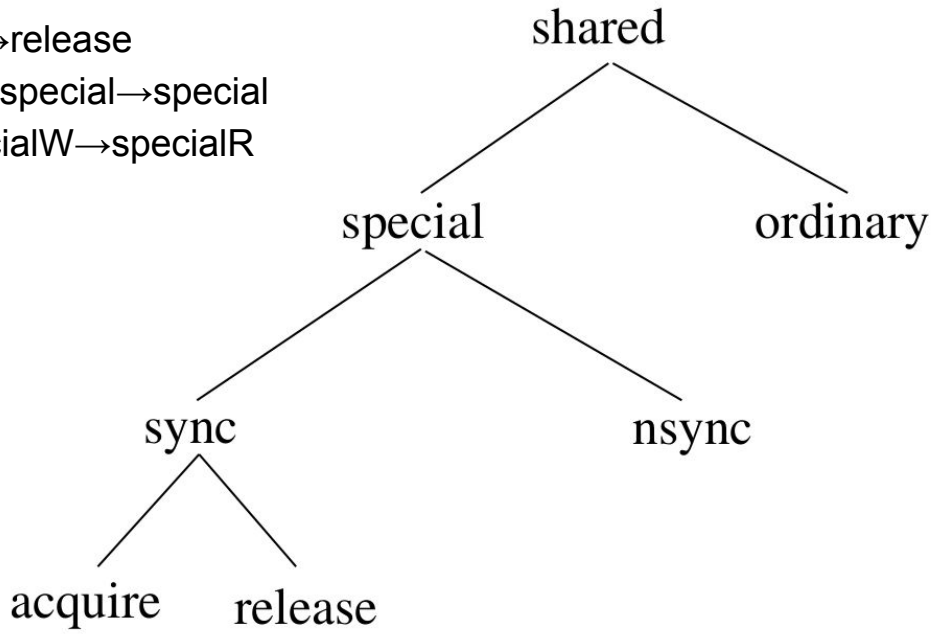
- PSO (Partial Store Ordering)
  - Może zamienić zapis z poprzedzającym zapisem, poza tym identyczny z TSO
  - Zaimplementowany w SPARC V8
- Przykładowy program wraz z koniecznymi barierami:

```
Na początku ready = data = 0
```

```
P1:                     P2:

data = 1;               while (ready == 0);
wmb();                  if (data == 0)
ready = 1;                  launch_missiles();
```

Osłabienie kolejności W→RW i R→RW

- Dowolne dwa dostępy do pamięci mogą zostać zamienione kolejnością
- Weak Ordering (WO):
  - Operacje dzielą się na operacje na danych (data) i operacje synchronizacji (synch)
  - Zachowana jest kolejność data→synch, synch→data, synch→synch
- Release Consistency (RCsc/RCpc)
  - Wprowadza inny podział operacji:
  - Zachowuje kolejność acquire→*, *→release
  - Wariant RCsc zachowuje kolejność special→special
  - Wariant RCpc też, z wyjątkiem specialW→specialR
- Inne przykłady
  - ARM
  - PowerPC
  - **Alpha**



Model pamięci DEC Alpha

- Wszystkie chwyty dozwolone
  - W→RW, R→RW
- Udostępnia instrukcje MB, WMB
- Najsłabszy model ze wszystkich architektur wspieranych przez Linuxa

Przykład:

```
Na początku a = 0, b = 1, p = &a

P1:                 P2:

b = 2;
mb();
p = &b;             q = p;
                    d = *q;
```

Czy możliwe jest `q = &b, d = 1`?

- Nie, na każdej architekturze poza Alpha

Model pamięci x86-64

Praktycznie identyczny z TSO, ale Intel nie mówi tego wprost.

Z dokumentacji (*Intel® 64 and IA-32 Architectures Software Developer's Manual,* tom 3, sekcja 8.2.2):

- "Reads are not reordered with other reads"
  - zachowana kolejność R→R
- "Writes are not reordered with older reads"
  - zachowana kolejność R→W
- "Writes to memory are not reordered with other writes"
  - zachowana kolejność W→W
  - istnieje kilka wyjątków
- "Reads may be reordered with older writes to different locations but not with older writes to the same location"
  - może wykonać odczyt przed poprzedzającym zapisem do innego miejsca w pamięci

Model pamięci x86-64

- “Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.”
  - Niektóre instrukcje mają wbudowane bariery
- “Reads cannot pass earlier LFENCE and MFENCE instruction”
  - Zachowana kolejność LFENCE→R, MFENCE→R
- “LFENCE instructions cannot pass earlier reads”
  - Zachowana kolejność R→LFENCE
- “SFENCE instructions cannot pass earlier writes”
  - Zachowana kolejność W→SFENCE
- “MFENCE instructions cannot pass earlier reads or writes”
  - Zachowana kolejność W→MFENCE i R→MFENCE
- “Writes by a single processor are observed in the same order by all processors”
- “Writes from an individual processor are NOT ordered with respect to the writes from other processor”
  - Nie ma gwarancji co do przeplotu dostępów do pamięci z różnych procesorów

Model pamięci x86-64

- Memory ordering obeys causality (memory ordering respects transitive visibility).
  - Nie ma sytuacji w której procesor widzi zapis innego procesora przed innymi procesorami (natychmiastowe zapisy)
- Przykład z dokumentacji:

```
                Na początku x = y = 0
P1:             P2:                 P3:

x = 1;          register1 = x;      register2 = y;
                y = 1;              register3 = x;
```

- Niedozwolony wynik: `register1 = 1, register2 = 1, register3 = 0`

Model pamięci x86-64

- “Any two stores are seen in a consistent order by processors **other than those performing the store**”
  - Odczyt może zwrócić wartość poprzedzającego zapisu tego samego procesora zanim ten zapis stanie się widoczny dla pozostałych procesorów
- Przykład z dokumentacji:

```
                Na początku x = y = 0

P1:                         P2:

x = 1;                      y = 1;
register1 = x;              register3 = y;
register2 = y;              register4 = x;
```

- Dozwolony wynik: register2 = 0, register4 = 0

Model pamięci C++11, std::atomic

Standard C++11 definiuje wyścig następująco:

- *A **memory location** is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width.*
- *Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.*
- *The execution of a program contains a **data race** if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither **happens before** the other, [. . .]. Any such data race results in **undefined behavior.***

Przykład wyścigu według C++11

```
std::vector<int> data;
bool data_ready(false); // Nieatomiczny typ danych

void writer_thread()
{
    data.push_back(42);
    data_ready = true;   // Jeden wątek pisze
}

void reader_thread()
{
    while(!data_ready)   // A drugi współbieżnie czyta
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::cout << ”The answer=” << data[0] << ”\n”;
}
```

Jak unikać wyścigów?

Z pomocą przychodzi standard i relacja *happens before*:

- *An evaluation A **happens before** an evaluation B*
  (or, equivalently, B happens after A) if:
  - *A is **sequenced before** B, or*
  - *A **inter-thread happens before** B.*
- *A is **sequenced before** B* ≡ A poprzedza B w porządku programu
- *An evaluation A **inter-thread happens before** an evaluation B if:*
  - *A **synchronizes with** B, or*
  - *A is **dependency-ordered before** B, or*
  - *for some evaluation X*
    - *A synchronizes with X and X is sequenced before B, or*
    - *A is sequenced before X and X inter-thread happens before B, or*
    - *A inter-thread happens before X and X inter-thread happens before B.*

Jak unikać wyścigów?

- Kluczem jest wprowadzenie relacji ***inter-thread happens before*** pomiędzy dostępami do współdzielonych danych.
  - Można to zrobić oddzielając dostępy **instrukcjami synchronizacji,** które wchodzą ze sobą w relację ***synchronizes with*** (synchronizuje się z)
  - W niektórych przypadkach wystarczy relacja *is dependency-ordered before*
- *Certain library calls **synchronize with** other library calls performed by another thread. For example, an atomic **store-release** synchronizes with a **load-acquire** that takes its value from the store*
  - Inny przykład: mutex.unlock() synchronizuje się z mutex.lock()
  - Ważna uwaga: istnienie relacji *synchronizes with* pomiędzy dwoma instrukcjami **nie jest znane w czasie kompilacji**, do synchronizacji dochodzi tylko wtedy, kiedy *load-acquire* na danym obiekcie przeczyta wartość zapisaną tam przez *store-release*, **w czasie wykonania**

- Ok, co to *load-acquire* i *store-release*?

Semantyka acquire-release

- Operacje odczytu na atomicznych obiektach (`std::atomic<>`) mogą (ale nie muszą) dodatkowo **uzyskać** (*acquire*) dany obiekt.
- Operacje zapisu na atomicznych obiektach mogą (ale nie muszą) dodatkowo **zwolnić** (*release*) dany obiekt
- Atomiczne operacje *read-modify-write* (np. *compare-exchange*) mogą zwolnić, uzyskać, lub zwolnić i uzyskać obiekt.
- Jeśli odczyt uzyskujący dany obiekt w wątku A odczytał wartość zapisu zwalniającego ten obiekt w wątku B, to wszystkie zapisy do dowolnych obiektów które poprzedzają zwalniający zapis w wątku B są widoczne dla odczytów następujących po uzyskującym zapisie w wątku A.
- Wszystkie odczyty obiektów atomicznych w C++ domyślnie uzyskują ten obiekt i wszystkie zapisy go zwalniają

Brak wyścigu dzięki acquire-release

```
std::vector<int> data;
std::atomic<bool> data_ready(false); // Atomiczny typ danych

void writer_thread()
{
    data.push_back(42);
    data_ready.store(true);// Zapis domyślnie zwalnia obiekt
}

void reader_thread()
{
    while(!data_ready.load())   // Odczyt uzyskuje obiekt
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    // Mamy gwarancję że odczyt data[0] zwróci 42
    std::cout << ”The answer=” << data[0] << ”\n”;
}
```

Domyślne gwarancje w C++11

- Poza acquire-release, wszystkie operacje na wszystkich atomicznych obiektach mają wspólny globalny porządek w którym operacje pojedynczego wątku występują zgodnie z porządkiem programu.
  - Sequential Consistency! (przynajmniej na zmiennych atomicznych)
- Zachowanie SC na platformach ze słabymi modelami pamięci (ARM, Power) jest kosztowne
- Często zdarza się że nie potrzebujemy SC do tego żeby program działał poprawnie
- Czy możemy nieco osłabić gwarancje w zamian za lepszy performance?

`std::memory_ordering_*`

- W C++11 możemy wyspecyfikować model, którego mają się trzymać operacje na zmiennych atomicznych.
- Sequential Consistency:
  - Domyślny
  - Najbardziej kosztowny, najmniej bólu głowy
  - `memory_ordering_seq_cst`
- Acquire-Release ordering:
  - SC bez globalnego porządku między operacjami na różnych obiektach
  - `memory_order_acquire` (load, RMW), `memory_order_release` (store, RMW), `memory_order_acq_rel` (RMW), `memory_order_consume` (store, RMW)
- Relaxed ordering:
  - Operacje atomiczne nie biorą udziału w synchronizacji
  - Gwarantuje jedynie atomiczność operacji i globalny porządek zmian pojedynczych obiektów
  - `memory_order_relaxed`

Brak wyścigu dzięki acquire-release (bez SC)

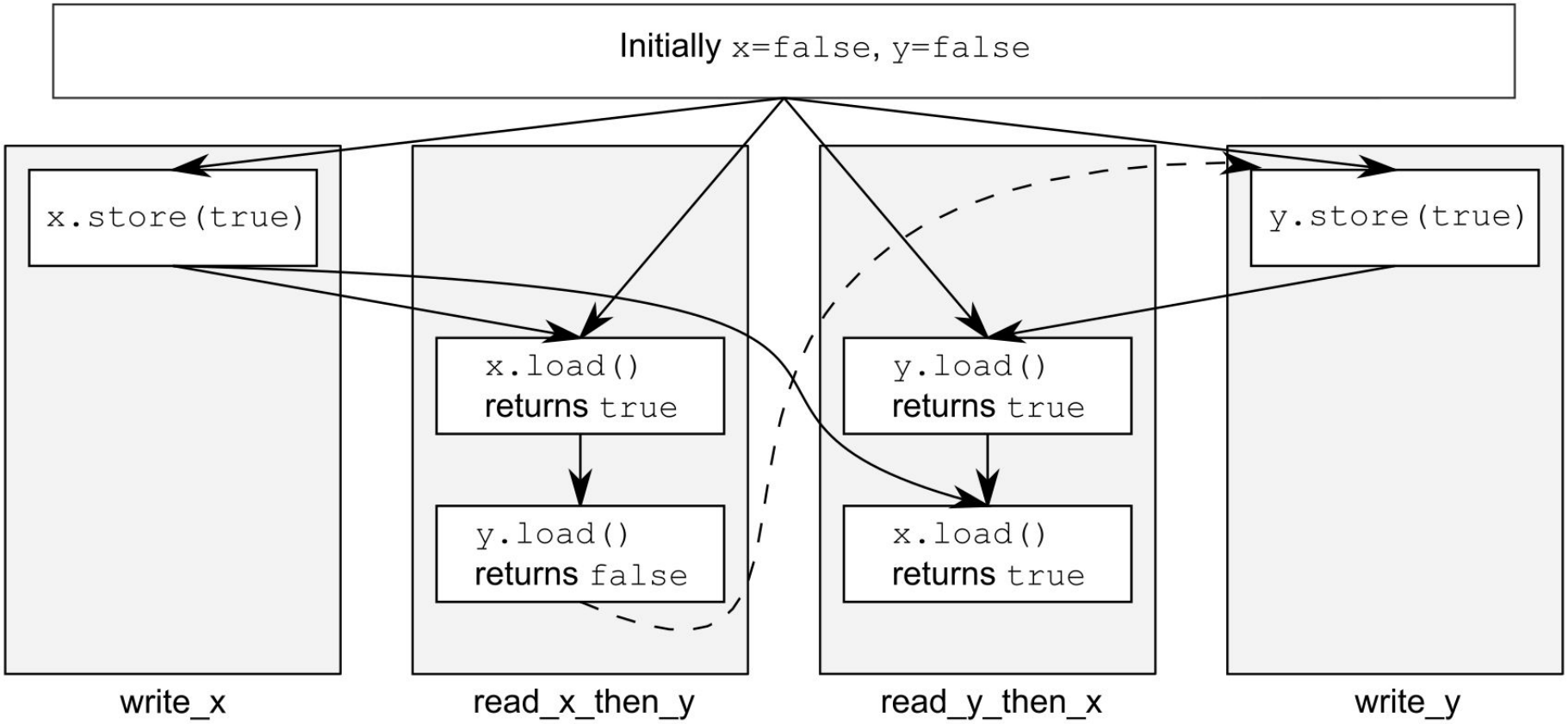
```
std::vector<int> data;
std::atomic<bool> data_ready(false); // Atomiczny typ danych

void writer_thread()
{
    data.push_back(42);
    data_ready.store(true, std::memory_order_release);
}

void reader_thread()
{
    while(!data_ready.load(std::memory_order_acquire))
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    // Mamy gwarancję że odczyt data[0] zwróci 42
    std::cout << ”The answer=” << data[0] << ”\n”;
}
```

Globalny porządek operacji w SC

```
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x()
{
    x.store(true,std::memory_order_seq_cst);
}

void write_y()
{
    y.store(true,std::memory_order_seq_cst);
}

void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))
        ++z;
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))
        ++z;
}
```

Mamy gwarancję że po wykonaniu wszystkich wątków `z > 0`

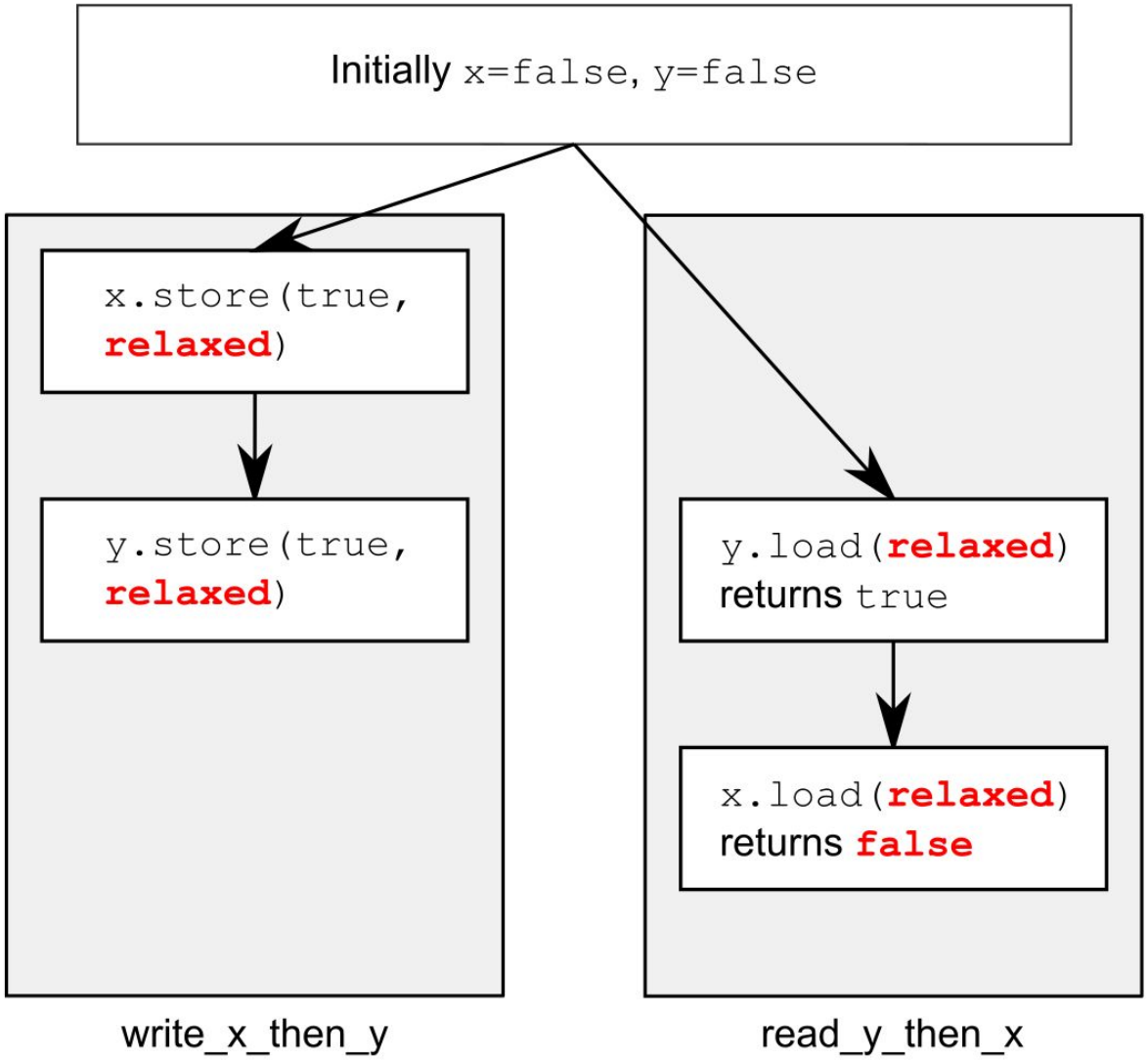
Globalny porządek operacji w SC - *happens before*



Brak synchronizacji w Relaxed Ordering

```
std::atomic<bool> x,y;
std::atomic<int> z;

void write_x_then_y()
{
    // Zapis do x poprzedza zapis do y w porządku programu
    x.store(true,std::memory_order_relaxed);
    y.store(true,std::memory_order_relaxed);
}

void read_y_then_x()
{
    // Ale to nie znaczy że y == true ⇒ x == true
    while(!y.load(std::memory_order_relaxed));
    if(x.load(std::memory_order_relaxed))
        ++z; // Może się nie wykonać!
}
```

Relaxed Ordering - *happens before*



Acquire-Release: acquire vs consume

```
struct X {
    int i;
    std::string s;
};
std::atomic<X*> p;
std::atomic<int> a;

void create_x() {
    X* x=new X;
    x->i=42;
    x->s="hello";
    a.store(99,std::memory_order_relaxed);
    p.store(x,std::memory_order_release);
}

void use_x() {
    X* x;
    while(!(x=p.load(std::memory_order_consume)))
        std::this_thread::sleep(std::chrono::microseconds(1));
    assert(x->i==42);
    assert(x->s=="hello");
    assert(a.load(std::memory_order_relaxed)==99); // Może zawieść
}
```

- Consume jest słabszy od acquire
- Zapewnia porządek między operacjami poprzedzającymi store-release i operacjami następującymi po load-consume **zależnymi od przeczytanej wartości**
- W praktyce rzadko stosowany, implementacje standardu nie potrafią przyspieszyć load-consume w stosunku do load-acquire

Bariery (*fences*)

- Pozwalają na synchronizację pomiędzy operacjami atomicznymi z porządkiem `memory_order_relaxed`
- Dwa rodzaje: acquire fence, release fence
- **Release fence** synchronizuje się z **atomiczną operacją load-acquire** (np. `foo.load(std::memory_order_acquire)` jeśli `foo.load` przeczytał wartość **atomicznego zapisu** następującego w porządku programu **po** barierze (np. `foo.store(42, std::memory_order_relaxed)`)
- **Atomiczna operacja store-release** synchronizuje się z **acquire fence** jeśli **atomiczny odczyt poprzedzający barierę** w porządku programu przeczytał wartość tego zapisu.
- Możliwa jest też synchronizacja pomiędzy dwoma barierami - wówczas mamy **atomiczny zapis po release fence** w jednym wątku i **atomiczny odczyt przed acquire fence w drugim**.

Zapobieganie wyścigom przy użyciu barier

```
std::vector<int> data;
std::atomic<bool> data_ready(false); // Atomiczny typ danych

void writer_thread() {
    data.push_back(42);
    std::atomic_thread_fence(std::memory_order_release);
    data_ready.store(true, std::memory_order_relaxed);
}

void reader_thread() {

    while(!data_ready.load(std::memory_order_relaxed))  {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::atomic_thread_fence(std::memory_order_acquire);
    // Mamy gwarancję że odczyt data[0] zwróci 42
    std::cout << ”The answer=” << data[0] << ”\n”;
}
```

Praktyczne zastosowanie barier: skrzynka pocztowa

```
const int num_mailboxes = 32;
std::atomic<int> mailbox_receiver[num_mailboxes];
std::string mailbox_data[num_mailboxes];

void writer_thread() {
    mailbox_data[i] = ...;
    std::atomic_store_explicit(&mailbox_receiver[i],receiver_id,
        std::memory_order_release);
}
// Czytelnik musi sprawdzić wszystkie miejsca w mailbox_data, ale
zsynchronizować
// się musi z tylko jednym wątkiem
void reader_thread() {
    for (int i = 0; i < num_mailboxes; ++i) {
        if (std::atomic_load_explicit(&mailbox_receiver[i],
                                      std::memory_order_relaxed) == my_id) {
            std::atomic_thread_fence(std::memory_order_acquire);
            do_work(mailbox_data[i]);
        }
    }
}
```

Literatura

1. Sarita V. Adve, Kourosh Gharachorloo, *Shared Memory Consistency Models: A Tutorial*, IEEE Computer, Vol. 29 No. 12, pp. 66-76, Dec. 1996
2. Daniel J. Sorin, Mark D. Hill, David A. Wood, *A Primer on Memory Consistency and Cache Coherence,* Morgan & Claypool Publishers, 2011
3. Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3
4. Anthony Williams, *C++ Concurrency in Action: Practical Multithreading*, Manning Publications Co., 2012