Język, biblioteka lub system	Funkcje, klasy lub typy	Uwagi

C
(od C11)
	`_Atomic` (typ)
	`atomic_compare_exchange_strong`
	`atomic_compare_exchange_weak`
	`atomic_fetch_add`
	`atomic_fetch_add_explicit`
	`...`
		Dostępne w ramach pliku nagłówkowego *stdatomic.h*, tylko w wypadku, gdy makro `__STDC_NO_ATOMICS__` nie jest zdefiniowane.

C++ (od C++11)	`std::atomic<T>` `std::atomic_flag`	Szablon `std::atomic` może wewnętrznie korzystać z dodatkowych blokad (np. muteksów) w niektórych przypadkach. Klasa `std::atomic_flag` jest natomiast zawsze implementowana bez zewnętrznych blokad (tj. korzysta z niskopoziomowych atomowych operacji blokujących).
boost (C++)	`Boost.Atomic<T>`	Implementacja `std::atomic<T>` z C++11 dla platform, które go dostarczają.

WinAPI	`InterlockedIncrement`
	`InterlockedExchange`
	`InterlockedAnd`
	`...`	

Tabela 2. Przykłady funkcji ustanawiających bariery

Język, biblioteka lub system	Funkcje lub klasy

C
(rozszerzenie GCC)
	`__sync_synchronize`
	`__sync_lock_test_and_set`
	`__sync_lock_release`

C, C++
(rozszerzenie Visual C++)
	`_ReadBarrier`
	`_WriteBarrier`
	`_ReadWriteBarrier`

C i C++ (od C11 i C++11)	`atomic_thread_fence`
WinAPI	`MemoryBarrier`

9.2. Spinlocki – wirujące blokady

Najprostszym niskopoziomowym mechanizmem synchronizującym jest tzw. spinlock (dosłownie: wirująca blokada), którego fragmenty mogą być wykorzystane do budowy innych, bardziej złożonych i częściej używanych w praktyce mechanizmów, jak muteksy czy zdarzenia. W najprostszym wydaniu spinlock jest aktywną pętlą (*spinning*), która oczekuje, aż dana

blokada (*lock*) zostanie zniesiona – w praktyce sprowadza się to do ciągłego, atomowego sprawdzania, czy dana zmienna „synchronizująca" (lub „obserwowana") zmieniła wartość. Przykładu spinlocka tego typu zastosowałem w przykładzie w poprzednim podrozdziale:

```
...
static volatile LONG g_poor_mans_lock;
...
  // Próba zsynchronizowania wątków, by wystartowały naraz, co
  // zwiększy ilość obserwowalnych kolizji. W praktyce kilka wątków
  // może być wywłaszczonych w tym momencie, niemniej jednak istnieje
  // istotne prawdopodobieństwo, że przynajmniej dwa wątki uruchomią
  // się w tym samym czasie.
  InterlockedIncrement(&g_poor_mans_lock);
  while (g_poor_mans_lock != THREAD_COUNT);
...
```

W powyższym przypadku wątki testowały zmienną `g_poor_mans_lock` i blokowały kod do momentu, aż ta nie osiągnęła wartości równej `THREAD_COUNT`. Należy zwrócić uwagę na kilka szczegółów:

- Zmiana wartości zmiennej, na której oparta jest blokada, powinna być atomowa (stąd użycie `InterlockedIncrement` do zaznaczenia, iż kolejny wątek jest gotowy).
- Jeśli spinlock jest realizowany w języku, który pozwala na agresywne optymalizacje[140], to zmienna, na której oparta jest blokada, powinna być z nich wykluczona. W przypadku języków C i C++ odpowiedzialne jest za to słowo kluczowe `volatile`, które informuje kompilator, iż wartość danej zmiennej może ulec zmianie w nieprzewidziany sposób w dowolnym momencie wykonania.
- Dopóki warunek blokady nie zostanie spełniony, aktywna pętla tego typu zużywa 100% przydzielonego czasu procesora.

Tworząc przedstawiony kod, brałem pod uwagę to, iż zastosowany spinlock miał na celu jedynie zsynchronizowanie rozpoczęcia pewnych operacji. Jeśli spinlock byłby użyty jako zdarzenie informujące o dostępności nowych danych w innej zmiennej, powinien zawierać on również barierę – o problemach wynikających z braku barier w przypadku spinlocków wspominałem już w ramce „Problemy manualnej synchronizacji [VERBOSE]".

[140] W szczególności moduł optymalizatora mógłby uznać, że nie warto ciągle pobierać wartości zmiennej z pamięci – lepiej pobrać ją raz przed pętlą, a potem operować na rejestrze (optymalizatory niektórych języków mogą, ale nie muszą, ignorować istnienie innych wątków). Co więcej, bazując na fakcie, że pętla jest pusta (a więc zmienna nigdy nie zostanie w niej zmodyfikowana), optymalizator mógłby użyć równoważnego przy obranych założeniach zapisu:

```
if (g_poor_mans_lock != THREAD_COUNT) {
   while(1);
 }
```