Niniejszy artykuł został opublikowany jako rozdział książki:
„Materiały konferencyjne IV Ogólnopolskiej Konferencji Inżynierii Gier Komputerowych".
Wydawnictwo Akademii Podlaskiej, Siedlce 2007

W razie jakichkolwiek pytań, uwag proszę pisać pod adres:
mborowiec@mbapp.com

Programowanie wielowątkowych gier w języku C++

Marcin Borowiec

Państwowa Wyższa Szkoła Zawodowa w Tarnowie

Streszczenie

Dzięki popularyzacji komputerów z procesorami wielordzeniowymi i konsol nowej generacji (XBox360, PS3) przed programistami gier komputerowych pojawia się nowe wyzwanie: zaprojektowanie wielowątkowej gry. Niestety rozdzielenie zadań na wątki i późniejsza ich synchronizacja nie jest prosta. Bardzo łatwo jest popełnić błąd, natomiast znaleźć bardzo trudno. Jeszcze gorsze jest to, że bez odpowiednich podstaw teoretycznych łatwo o błąd, który ukaże się dopiero po uruchomieniu gry w specyficznych okolicznościach na danej platformie. Referat zawiera wszystkie podstawowe, teoretyczne informacje o wątkach i ich synchronizacji. Razem z dokumentacją do wybranej przez nas biblioteki wystarczy do napisania poprawnego programu.

1. Wstęp

Do tej pory stosowanie wielowątkowości w grach było rzadko spotykane. Powód jest bardzo prosty. Równoległe przetwarzanie ma sens praktycznie tylko w dwóch przypadkach:

- Wykorzystujemy funkcje blokujące, dobrym przykładem jest tutaj domyślny tryb w jakim działają gniazda sieciowe, gdy wywołujemy funkcje read wątek zostaje zablokowany przez system dopóki nie będzie danych do pobrania lub połączenie zostanie zamknięte.
- Chcemy wykorzystać moc systemów wieloprocesorowych, komputery

kierowane do graczy były tworzone w oparciu o jeden procesor jednordzeniowy. Po co więc tracić czas żeby zaopatrzyć nasz program w coś co i tak nikt nie wykorzysta.

Producenci procesorów kierowanych na rynek domowych komputerów natrafili na problemy z kolejnym przyspieszaniem procesorów poprzez zwiększanie częstotliwości taktowania. Rozwiązaniem okazało się zwiększenie liczby rdzeni. Procesory czterordzeniowe są już w sprzedaży a dwurdzeniowe stają się pomału standardem. Osoba, która kupi teraz nowy komputer będzie chciała żeby gry wykorzystały potencjał jego maszyny. Nie wolno także zapomnieć o najnowszej generacji konsol. PlayStation 3 czy XBox360 mają procesory kilkurdzeniowe. Trudno jednak znaleźć dobre źródło informacji na ten temat, niniejszy referat ma pomóc zapełnić tą lukę.

2. Równoległe przetwarzanie danych w teorii

Równoległe przetwarzanie danych można osiągnąć tworząc nowy proces albo nowy wątek. Jak wiadomo wszystko ma swoje wady i zalety.

Tabela 2.1 Porównanie procesów I wątków

Proces	Wątek
➢ Zawiera co najmniej jeden wątek	➢ Jednostka, dla której system przydziela czas procesora
➢ Nie wykonuje kodu, dostarcza wątkom przestrzeń adresową	➢ Wykonuje kod procesu
➢ Sterta i zasoby systemowe dzielone są pomiędzy wszystkie wątki procesu	➢ Kontekst wątku składa się z licznika rozkazów, stanu rejestrów, stosu
➢ Większe bezpieczeństwo: zamknięcie jednego procesu spowodowane błędem nie powoduje automatycznego zamknięcia innego procesu, lepsza separacja danych zasobów	➢ Łatwiejsze i szybsze metody synchronizacji

W grach liczy się przede wszystkim wydajność, z kolei bezpieczeństwo nie jest aż tak ważne. Jeżeli w którymś z wątków nastąpi błąd krytyczny wystarczy wyświetlić komunikat o błędzie, zapisać logi do pliku i zamknąć grę. Nikt nie straci przez to życia (najwyżej wirtualne ;)). Oczywistym jest że w takim przypadku do uzyskania równoległości obliczeń lepiej jest wykorzystać wątki.

2.1. Metody komunikacji pomiędzy wątkami

Dwa różne wątki muszą się w jakiś sposób ze sobą komunikować (np. trzeba przekazać dane o nowym położeniu gracza do wątku głównego z pomocniczego zajmującego się sztuczną inteligencją). Wymiana danych może się odbyć na dwa sposoby:

- Użycie systemowych funkcji do komunikacji takich jak np. potoki, kolejki, gniazda sieciowe itp. Metody te charakteryzują się tym że nie trzeba ich dodatkowo synchronizować. Niestety mają też dużo wad. Jeden wątek musi w jakiś sposób sformatować te dane (np. ułożyć dane w potoku, dodać odpowiednie nagłówki z informacjami o typie wysyłanych danych) aby inny mógł je odpowiednio zinterpretować. Dodatkowo tracimy czas na systemowych funkcjach, które muszą kopiować dane z jednego miejsca pamięci w drugie. Z tych względów metody te są mało przydatne w programowaniu gier i nie będą poruszane w dalszej części referatu. Zainteresowanych odsyłam do [1] i [2]
- Wykorzystanie faktu że wątki należące do jednego procesu dzielą ze sobą tą samą przestrzeń adresową. Niestety dostęp do wspólnych danych wymaga synchronizacji. To właśnie temu tematowi poświecony jest referat.

2.2. Synchronizacja wątków

Celem synchronizacji jest zapobieganie sytuacji wyścigów (ang. race condition). Problem polega na tym, że wynik działania naszego programu może zależeć od różnych przypadkowych zdarzeń. Jako przykład można wziąć następującą sytuację: Program składa się z dwóch wątków, jeden z nich kasuje elementy wektora (bo np. dane są już nie aktualne), inny pobiera po jednym obiekcie jeżeli taki znajduje się w wektorze. Przebieg zdarzeń może wyglądać tak:

Tabela 2.2 Przebieg programu, przypadek a)

Krok	Wątek 1	Wątek 2
1	Skasuj wszystkie elementy wektora	
2		Czy wektor zawiera jakiś element?
3		Jeżeli tak, pobierz jeden obiekt

Tabela 2.3 Przebieg programu, przypadek b)

Krok	Wątek 1	Wątek 2

1		Czy wektor zawiera jakiś element?
2		Jeżeli tak, pobierz go
3	Skasuj wszystkie elementy wektora	

Tabela 2.4 Przebieg programu, przypadek c)

Krok	Wątek 1	Wątek 2
1		Czy wektor zawiera jakiś element?
2	Skasuj wszystkie elementy wektora	
3		Jeżeli tak, pobierz jeden obiekt

W każdym z powyższych przypadków program zachowa się inaczej. Np. w przypadku b) obiekt znajdujący się w wektorze zostanie pobrany i użyty do dalszych działań, natomiast w przypadku a) zanim sprawdzimy czy jest jakiś obiekt do pobrania, wszystkie elementy wektora są usuwane. Oba przypadki można uznać za poprawne, skoro obiekty zostały usunięte tzn że są już nie potrzebne i nie należy ich przetwarzać dalej. Niestety sytuacja opisana w pkt c) ewidentnie wskazuje na błąd w programie, następuje próba pobrania obiektu, który został przed momentem skasowany.

2.3. Operacje atomowe

Operacje atomowe to takie operacje, które nie mogą zostać przerwane, a po wykonaniu instrukcji wynik działania „widoczny jest od razu dla wszystkich wątków programu". Ich zaletą jest to że nie trzeba blokować dostępu np. za pomocą muteksów (więcej informacji w 2.4). Niestety ilość operacji atomowych na danej platformie jest bardzo mała i przeważnie ogranicza się do prostych operacji na liczbach całkowitych. Wszystko zależy od specyfikacji języka i procesora w komputerze, na którym będzie uruchamiany program.

2.4. Dostęp blokowany, polityki dostępu

W przypadku gdy platforma nie udostępnia mechanizmów do atomowego wykonywania danych instrukcji, należy użyć dodatkowych obiektów, których stan określa czy w danym momencie można wykonać określoną operacje (np. odczyt/zapis jakiegoś zasobu). Obiekty te nazywane są **muteksami** (ang. Mutual Exclusion, wzajemne wykluczanie) a

blokowane instrukcje **sekcjami krytycznymi.** Praktycznie każdy system operacyjny udostępnia funkcje do obsługi muteksów. Możemy je także zaimplementować sami, należy jednak pamiętać że do zmiany stanu należy użyć operacji atomowych z wywołaniem pełnej synchronizacji pamięci.

Jeżeli kilka wątków chce wejść do tej samej sekcji krytycznej, to każdy z nich prędzej czy później musi uzyskać do niej dostęp. W przeciwnym wypadku program zostałby zawieszony. Oczywiście wątki nie mogą wejść do sekcji krytycznej na raz, tylko według ustalonej kolejności. Przydzielanie dostępu może odbywać się w sposób losowy, wątki co jakiś czas sprawdzają czy zasób jest wolny i jeżeli tak to blokują go. Mamy wtedy gwarancje że dany wątek uzyska dostęp najpóźniej gdy wszystkie inne wątki wyjdą z sekcji krytycznej, lub np. za pomocą kolejki FIFO, dostęp przydzielany jest na podstawie kolejności zgłoszeń.

Można także wyróżnić różne polityki dostępu do zasobu:

- dostęp na wyłączność – jeżeli wątek wykonuje dowolną operacje na obiekcie każdy inny wątek zostaje zablokowany gdy próbuje odczytać/zapisać wspólny zasób.
- dostęp dzielony – wątki dzielone są na „pisarzy i czytelników" w zależności od typu operacji (odczyt/zapis) jakie chcą wykonać w danej chwili na obiekcie. Ustalane są następujące zasady: tylko jeden wątek może w danym momencie zapisywać, ale wiele wątków może uzyskać dostęp wtedy gdy chcą tylko odczytywać. Należy zwrócić uwagę że ta polityka mocno faworyzuje czytelników. Pisarze muszą czekać aż każdy z czytelników zwolni dostęp do zasobu.

2.5. Zakleszczenia (ang. Deadlock)

Zakleszczenie jest jedną z bardzo niepożądanych rzeczy w programowaniu aplikacji wielowątkowych gdyż doprowadza do zawieszania naszego programu. Występuje w przypadku nieprawidłowego blokowania wielu zasobów w jednym momencie. Poniższy przykład ilustruje w jaki sposób może powstać zakleszczenie:

Tabela 2.5 Przykład powstawania zakleszczenia

Nr	Wątek 1	Wątek 2
1	Zablokowanie dostępu do zasobu A	
2		Zablokowanie dostępu do zasobu B
3	Zablokowanie dostępu do zasobu B	
4		Zablokowanie dostępu do zasobu A
5	Operacje na zasobach A	

	i B	
6		Operacje na zasobach A i B
7	Zwolnienie dostępu do zasobu B	
8		Zwolnienie dostępu do zasobu A
9	Zwolnienie dostępu do zasobu A	
10		Zwolnienie dostępu do zasobu B

Wykonanie pierwszej instrukcji w obydwu wątkach jest jak najbardziej prawidłowe. Pierwszy wątek blokuje dostępu do zasobu A drugi do zasobu B. Trzeci krok jest już niebezpieczny. Wątek 1 dodatkowo próbuje uzyskać dostęp do zasobu B. Niestety jest on już zablokowany i trzeba poczekać aż drugi wątek zwolni B. Krok 4 to sytuacja identyczna tylko z wątkiem 2 i zasobem A. Okazuje się że w tym momencie wątki czekają na siebie nawzajem. Pozostaje tylko restart całego procesu albo przynajmniej tych dwóch wątków.

Co zrobić żeby uniknąć takich sytuacji? Jednym z prostszych i skutecznych rozwiązań jest ustalenie kolejności blokowania obiektów. Gdyby w powyższym przykładzie drugi wątek blokował wątki w tej samej kolejności co pierwszy do zakleszczenia by nie doszło. Jeżeli obiekty znajdują się w tablicy możemy np. blokować najpierw te o niższych indeksach lub jeżeli np. programujemy w C++ te o niższym adresie w pamięci.

3. Wielowątkowość w języku C++

C++ jest ciągle najpopularniejszym językiem programowania stosowanym w programowaniu gier komputerowych. Niestety pod względem pisania wielowątkowych programów C++ zostaje w tyle. W najnowszej wersji standardu [3] nie ma nawet nigdzie słowa wątek. Program zdefiniowany jest jako ciąg instrukcji wykonywanych w określonej kolejności. W przetwarzaniu równoległym ta kolejność już nie jest znana. Dlatego teoretycznie każdy wielowątkowy program w C++ możemy uznać za niezgodny ze standardem. Nie oznacza to jednak że nie jest możliwe napisanie w 100% poprawnego programu w języku C++. Podpierając się dokumentacją do kompilatora, którego używamy, funkcjami API systemu operacyjnego czy zewnętrznymi bibliotekami możemy zdziałać bardzo dużo. Dodatkowo na rok 2009 zapowiedziana jest następna wersja standardu języka C++, która ma w końcu rozwiązać problemy z wątkami.

3.1. Model pamięci (ang. Memory model)

Przy omawianiu programowania pod daną platformę najpierw należy zaznajomić się z modelem pamięci, który jest używany. Określa on, które operacje są atomowe i w jaki sposób inne wątki widzą obszar pamięci modyfikowany przez dane instrukcje. Jak już zostało wspomniane aktualna wersja standardu C++ nie definiuje modelu pamięci, należy więc odwołać się do specyfikacji sprzętu na którym uruchomimy nasz program.

- architektura **x86** (komputery PC z procesorami Intel Pentium, Intel Core, AMD Athlon itp) – w przypadku maszyny wieloprocesorowej nie ma żadnej gwarancji, że wszystkie wątki zobaczą zmianę wartości danej komórki w tym samym momencie przy użyciu zwykłych instrukcji zapisu do pamięci. Procesory dobrze wykorzystują tą właściwość optymalizując dostęp do danych. Uzyskanie pełnej spójności danych pomiędzy procesorami wymaga użycia memory barrier (rodział 3.2) [4]
- architektura **POWER** (G4,G5 – PowerMac, Xenon – XBox360, Cell – PlayStation3) – tutaj sytuacja jest jeszcze ciekawsza. Inne wątki mogą zobaczyć że dane zostały zapisane w innej kolejności niż wynika to z kolejności instrukcji programu. Dostajemy za to kilka różnych instrukcji do synchronizacji. [5]
-

Podsumowując: Dopóki różne wątki ograniczają się do odczytu danych z jednego obszaru pamięci to nie jest wymagana synchronizacja. Do utrzymania spójności danych podczas zapisu służą memory barriers.

3.2. Memory barriers

Memory barriers [6] są to specjalnie instrukcje wymuszające na procesorze wykonanie operacji na pamięci w określonej kolejności w zależności od typu operacji i jej położenia (przed albo po barierze). Pełna synchronizacja powoduje że w momencie napotkania bariery wszystkie wątki widzą te same dane w całej pamięci którą dzielą między sobą. Nie zawsze jest to jednak potrzebne dlatego niektóre architektury oferują różne instrukcje, które nie wymuszają pełnej synchronizacji.

Pisząc program w C++ wystarczy zapamiętać dwie rzeczy:

- kompilator nie wstawia nigdzie memory barrier
- jeżeli wykorzystujemy biblioteki do synchronizacji (np. pthread) to częścią operacji takich jak tworzenie, dołączanie się do wątku czy blokowanie, zwalnianie muteksu jest właśnie wywołanie memory barrier

3.3. Dlaczego nie volatile?

Przeszukując informacje w Internecie o programowaniu wielowątkowym można napotkać na dużo nieprawdziwych informacji. Najwięcej związanych

jest ze słowem kluczowym ***volatile***. Sprawia to że wiele osób (głównie początkujących programistów) tworzy nowe wątki, np. za pomocą funkcji *_beginthread* czy *CreateThread* a później synchronizuje je przez zmienne zadeklarowane z kwalifikatorem ***volatile*** bo tak jest najprościej.

Czemu niektórzy uważają że volatile wystarcza do synchronizacji wątków? W standardzie języka C++ [3] w pkt 7.1.5.1 możemy przeczytać: „`volatile` *is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.„* Co oznacza mniej więcej: "volatile jest sugestią dla danej implementacji kompilatora aby nie używał agresywnych optymalizacji na danym obiekcie, ponieważ jego wartość może być zmieniona niezauważalnie dla programu" . Przez co? Np. Przez system operacyjny, który uaktualnia co jakiś czas daną komórkę pamięci używaną przez nasz program. Niestety brak optymalizacji nie wystarczy żeby zagwarantować poprawność synchronizacji. Jako przykład weźmy krótki kod w języku C++ (*a* jest zadalekrowane jak ***volatile int****=1;*):

Tabela 2.6 Przypadek z atomowymi instrukcjami C++

Nr1.	Wątek 1	Wątek 2
1	a+=3;	
2		a+=2;

Wartość zmiennej *a* powinna wynosić 6. Niestety nie ma gwarancji że kompilator każdą z tych operacji zapisze jako jedną instrukcje procesora. W rzeczywistości może to wyglądać tak[1] (opis w pseudojęzyku, przyjęto że rejestry R1 i R2 na starcie programu mają wartość 0):

Tabela 2.7 Przypadek z wydzielonymi wczytaniem/zapisem zmiennej

Nr1.	Wątek 1	Wątek 2	R1	R2	a
1	pobierz a do R1		1	0	1
2		pobierz a do R2	1	1	1
3	dodaj 3 do R1		4	1	1
4		dodaj 2 do R2	4	3	1
5	zapisz wartość R1 do a		4	3	4
6		zapisz wartość R2 do a	4	3	3

Jak widać powyżej, końcowa wartość zmiennej *a* to 3 czyli nie to co oczekiwaliśmy. Dodatkowo kompilator nie wstawia memory barrier przez co

1 Visual C++ 2005 w trybie Release z wyłączonymi optymalizacjami rozdzielił intrukcje *a+=2*; na pobranie, dodawanie i zapis niezależnie od tego czy *a* było zadeklarowane jako *volatile* czy nie

jest pewne że kod używający tylko ***volatile*** zawiedzie w prawdziwym wieloprocesorowym systemie.

Z powyższych przykładów jasno wynika że ***volatile*** nie wystarcza do komunikacji pomiędzy wątkami. Następne pytanie, które należy sobie zadać, czy w ogóle istnieje potrzeba używania tego kwalifikatora. Okazuje się że nie. Modyfikacja zmiennych w programie powinna wyglądać tak:

```
InstrukcjaAtomowa(&obiekt);
```

lub

```
Zablokuj(muteks);
a+=2;
// inne instrukcje na wspólnych zmiennych
Zwolnij(muteks);
```

Kompilator nie zna ciała powyższych funkcji, przez co nie może zoptymalizować kodu przez zostawienie wartości w rejestrze na czas ich wywołania. Jest pewne że wartość zmiennej *a* powiększonej o 2 zostanie zapisana do pamięci przed wywołaniem *Zwolnij*. Zastosowanie ***volatile*** może jedynie spowodować wymuszenie nie przeprowadzania optymalizacji tam gdzie one nie przeszkadzają czyli jedynie niepotrzebnie zwolnić program. Podsumowując: jeżeli w programie zadeklarowano zmienne z kwalifikatorem ***volatile*** to oznacza że jest duża szansa na błąd w tym programie.

3.4. Funkcje do obsługi wątków

Pisząc program wielowątkowy pod Windows można wykorzystać WinAPI, które dostarcza wiele funkcji do tworzenia/ synchronizacji wątków. Niestety jest to biblioteka pisana z myślą o języku C i dość ciężko się ją obsługuje. Pod linuksem jest biblioteka pthread (POSIX Thread) (także tworzona z myślą o języku C), jest jednak bardziej przenośna, oprócz *nixów istnieje implementacja pod Windows opakowująca WinAPI. Osoby zainteresowane mogą znaleźć informacje w [1] [2]. Lepszym rozwiązaniem jest użycie biblioteki, która wykorzystuje właściwości języka C++ np. Boost::Thread [7]. Przykładowy kod znajduje się poniżej:

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/shared_ptr.hpp>

class A
```

```
{
public:
 A(int a):m_member(a)
 {
  boost::mutex::scoped_lock(m_mutex,true);
  //zablokowanie mutekstu - memory barrier dla m_member
  m_count=0;
  m_sum=0;
 } //obiekt scoped_lock jest niszczony co powoduje
zwolnienie mutekstu - memory barrier dla m_count i m_sum
 void fun(int a)
 {
  //funkcja wylicza wartosc na podstawie a i dodaje do
  //m_sum;
  int sum=0;
  for (int i=0; i<a; ++i)
   sum+=m_member; //odczyt m_member nie wymaga
   //synchronizacji
  {
   boost::mutex::scoped_lock(m_mutex,true);
   ++m_count;
   m_sum+=sum;
  } //memory barrier dla m_count, m_sum
 }
 int get_sum()
 {
  int temp;
  //uzycie zmiennej tymczasowej jest konieczne,
  //kopiowanie zwracanej wartosci
  //odbywa sie po usunieciu wszystkich zmiennych
  //lokalnych
  {
   boost::mutex::scoped_lock(m_mutex,true);
   temp=m_sum;
  }
  return temp;
 }
private:
 const int m_member;
 int m_count;
 int m_sum;
 boost::mutex m_mutex;
};
```

```
int main()
{
 const int THREADS_NUM=4;
 //liczba watkow do utworzenia

 typedef boost::shared_ptr<boost::thread> thread_ref;
 //boost::thread nie mozna kopiowac
 thread_ref threads[THREADS_NUM];
 A a(3);
 for (int i=0; i<THREADS_NUM; ++i)
 //start nowego watku, jako parametr podajemy funkcje
 //ktora ma byc wykonana
 //dzieki boost::bind mozemy przekazac metode z
 //okreslonymi parametrami
 threads[i]=thread_ref(new
  boost::thread(boost::bind(&A::fun,&a,i*10000000)));
 for (int i=0; i<THREADS_NUM; ++i)
 {
  //czekanie na zakonczenie pracy watku
  threads[i]->join();
 }
 std::cout<<a.get_sum()<<std::endl;
 std::cin.get();
}
```

Wynik działania programu:

```
180000000
```

Powyższy program pokazuje jak używać bibliotekę Boost::Thread. Przedstawia w jaki sposób wywołać wątek przekazując do jego głównej procedury parametry. Widać też jak możemy blokować dostęp do zasobów za pomocą mutekstów.

Boost::Thread niestety nie jest biblioteką idealną. Brakuje jej nawet kilku podstawowych rzeczy takich jak:

- join nie zwraca żadnej wartości, jeżeli chcemy przekazać wartość z funkcji musimy użyć wspólnej zmiennej i mutekstu do jej synchronizacji
- typów do wykonywania podstawowych atomowych operacji na liczbach całkowitych
- puli wątków – bardzo użyteczna koncepcja rozdzielania zadań, jeżeli utworzymy mniej wątków niż system ma procesorów nie wykorzystamy w pełni jego mocy, jeżeli utworzymy za dużo wątków zajmujemy więcej zasobów (każdą wątek ma m.in. własny stos). Dużo lepszym

rozwiązaniem jest stworzenie grupy wątków, które będą na bierząco wykonywać zadania dodane bez kolejki. Nie tracimy wtedy czasu na stworzenie, usunięcie wątku.

UWAGA:
Niektóre biblioteki oferują funkcje do bezwarunkowego usunięcia wątku. W W WinAPI jest to TerminateThread. Funkcja jest bardzo niebezpieczna w skutkach, przed usunięciem wątku nie są kasowane obiekty, które należą do niego. W efekcie wykonanie programie może okazać się nieprzewidywalne. Jeżeli chcemy zakończyć określony wątek powinnyśmy mu przekazać tą informacje za pomocą dowolnej metody komunikacji, wtedy wątek skończy swoją funkcje główną co prowadzi do prawidłowego zwolnienia wszystkich zasobów.

3.5. Biblioteka standardowa

Zanim zaczniemy korzystać z dobrodziejstw biblioteki standardowej (np. std::string, std::vector) należy upewnić się czy implementacja z której korzystamy jest „thread-safe" tzn czy można ją bezpiecznie używać w programach wielowątkowych i na jakim modelu pamięci jest oparta. W opisach do Visual C++ [8] i STL od SGI [9] można przeczytać że dostęp do instancji klas wchodzących w std powinien się odbywać podobnie jak do zwykłych zmiennych. Tzn: dwa różne wątki mogą czytać ten sam obiekt, ale tylko jeden wątek może zapisywać i to programista musi zadbać o poprawną synchronizację. Twórcy bibliotek zadbali o to aby różne instancje tej samej klasy nie używały zmiennych statycznych oraz dostarczyli bezpieczną wersje alokatora. Dodatkowo implementacja iostreams w Visual C++ 2005 umożliwia zapis na obiektach (np. cout) bez dodatkowej synchronizacji.

3.6. Wątki a wyjątki

Język C++ nie posiada mechanizmu propagacja wyjątków na inne wątki. Żaden z producentów kompilatorów nie dodał też żadnego rozszerzenia, które rozwiązało by ten problem. Oznacza to że główna funkcja wątku (ta, której adres przekazywany jest do funkcji tworzącej wątek), powinna złapać wszystkie wyjątki, a ewentualne informacje o błędach zapisać w zmiennej, którą przeczyta główny program.

3.7. Wątki w C++0x

Przyszła wersja standardu C++ [10] (aktualnie określana jako C++0x) zostanie rozszerzona m.in. o wsparcie dla wątków. Do wydania finalnej wersji pozostało jeszcze ok. 2lata (według planów), ale już teraz warto przyjrzeć się zastosowanym rozwiązaniom. Należy jednak pamiętać że

poniższe informacje oparte są na wersji roboczej standardu (draft) i mogą się jeszcze zmienić. Jedno jest pewne, nie będzie żadnej rewolucji, większość zmian polega na dodaniu nowych klas do biblioteki standardowej. W samym języku dostajemy tylko (a może aż?) zdefiniowany model pamięci [11] [12] i nowe słowo kluczowe „__thread" [13] pozwalające na tworzenie prywatnych zmiennych wątku. Żadna operacja nie jest domyślnie atomowa a synchronizacja powinna być przeprowadzana tak jak przedstawiono w tym rozdziale albo za pomocą nowych funkcji std. Standardowa biblioteka udostępnia nam API [14] do obsługi wątków, włącznie z klasami do operacji atomowych na liczbach całkowitych i wskaźnikach [15]. Udostępnione są dwa szablony, native_atomic i atomic. Ten pierwszy wykonuje tylko takie operacje, które mogą być wykonane natywnie za pomocą instrukcji procesora a drugi emuluje brakujące operacje za pomocą muteksów. Przy wywołaniu danej funkcji musimy jawnie podać jaki typ bariery ma zostać użyty. Istnieje jeszcze propozycja dodania propagacji wyjątków podczas operacji join [16] jednak nie wiadomo czy pojawi się ostatecznie w standardzie.

4. OpenMP

OpenMP API [17] to rozszerzenie dla języków C/C++ i Fortran definiujące model programowania wielowątkowego. Większość producentów dodała już obsługę OpenMP do swoich kompilatorów:

Tabela 4.1 Obsługa OpenMP w popularnych kompilatorach

Nazwa i wersja kompilatora	Wersja OpenMP	Opcja kompilatora włączająca OpenMP
MS Visual C++ 2005	2.0	/openmp
gcc 4.2	2.5	-fopenmp
ICC 9.1	2.5	-openmp

W C++ OpenMP zostało dodane jako zestaw dyrektyw preprocesora (#pragma omp). Oprócz tego mamy kilkanaście funkcji (do sprawdzania ilości procesorów, ilości uruchomionych wątków itp.) i kilka makr preprocesora dostępnych po dołączeniu nagłówka <omp.h>

4.1. Model pamięci

Każdy z wątków może posiadać własny tymczasowy widok na wspólny obszar pamięci[2]. Wszystkie zmienne używane w bloku ***parallel*** można

2 Powodem istnienia tymczasowego prywatnego widoku pamięci są struktury pośrednie pomiędzy wątkiem a pamięcią takie jak rejestry, pamięć cache, lokalna pamięć. Dobrym przykładem jest tutaj architektura procesora Cell gdyż zawiera wszystkie wyżej wymienione

podzielić na prywatne (***private***) i wspólne(***shared***). Dostęp do zmiennych dzielonych powinien być synchronizowany za pomocą instrukcji ***flush***. W pewnych przypadkach operacja ta jest wykonywana niejawnie. Dokładny opis znajduje się w dokumentacji OpenMP.

4.2. Hello world w OpenMP

Przykład prostego programu napisanego w C++ z OpenMP:

```
#include <iostream>

#ifdef _OPENMP
 #include <omp.h>
#endif

int main()
{
 #ifdef _OPENMP
  std::cout<<"OpenMP dostepne w wersji:"<<
   _OPENMP<<std::endl;
  std::cout<<"Ilosc procesorow w systemie: "<<
   omp_get_num_procs()<<std::endl;
  #pragma omp parallel
  {
   #pragma omp master
   {
     std::cout<<"Utworzono grupe "<<
      omp_get_num_threads()<<" watkow"<<std::endl;
   }
   #pragma omp barrier
   #pragma omp critical
   {
     std::cout<<"Pozdrowienia z watku: "<<
     omp_get_thread_num()<<std::endl;
   }
  }
 #else
  std::cout<<"Kompilator nie obsluguje OpenMP"<<
   std::endl;
 #endif
 std::cin.get();
}
```

argumenty [18].

Po skompilowaniu w Visual C++ 2005 i uruchomieniu na komputerze z procesorem Athlon64 X2 otrzymałem taki wynik:

```
OpenMP dostepne w wersji: 200203
Ilosc procesorow w systemie: 2
Utworzono grupe 2 watkow
Pozdrowienia z watku: 0
Pozdrowienia z watku: 1
```

200203 oznacza w tym wypadku datę opublikowania wersji 2.0 OpenMP (dla wersji 2.5 będzie to 200505). Ilość procesorów to dokładnie łączna ilość wątków wykonywanych sprzętowo na danej platformie. Athlon64 X2 ma dwa rdzenie z czego każdy z nich może wykonać tylko jeden wątek w danym momencie. Do utworzenia grupy wątków służy polecenie ***#pragma omp parallel***, instrukcje wewnątrz bloku kodu znajdującego się po tej dyrektywie wykonywane są przez wszystkie wątki. Ilość tworzonych wątków zależy od wielu rzeczy, w tym wypadku utworzono dodatkowo jeden wątek[3] tak aby w sumie liczba wątków w danej grupie wynosiła 2. Wątek główny jest określany jako „master" i otrzymuje numer 0, każdy kolejny utworzony wątek otrzymuje numer o jeden większy. Powyższy program zawiera także inne dyrektywy: ***#pragma omp master*** – kod wykonywany jest tylko przez mastera, ***#pragma omp barrier*** – wykonywanie dalszych instrukcji jest możliwe dopiero wtedy gdy każdy wątek grupy dotrze do bariery, **#pragma omp critical** – sekcja krytyczna, tylko jeden wątek może wykonać zaznaczony kod w danym momencie.

4.3. Rozdzielanie pracy pomiędzy wątki

Po utworzeniu grupy wątków dyrektywą ***parallel*** nasz kod zostanie wykonany przez każdy z wątków. W większości przypadków to nie jest to co chcemy uzyskać. Do przydzielenia danej operacji do wykonania tylko jednemu wątkowi służy ***#pragma omp single***. Na końcu bloku kodu znajduje się bariera. Jeżeli chcemy aby inne wątki nie czekały na wykonanie tych instrukcji musimy dodać parametr nowait[4]. W przypadku gdy mam kilka zadań do wykonania w jednym czasie najlepiej jest użyć ***#pragma omp sections***. Przykład:

```
#pragma omp sections
{
```

3 Przy domyślnych ustawieniach liczba wątków określana jest statycznie przy uruchomieniu programu i równa się ilość wątków jakie platforma może wykonać sprzętowo.

4 Parametr ***nowait*** można użyć do dowolnej konstrukcji, która domyślnie kończy się barierą.

```
 #pragma omp section
 {
   Funkcja1();
 }
 #pragma omp section
 {
   Funkcja2();
 }
 #pragma omp section
 {
   Funkcja3();
 }
}
```

Na końcu ***sections*** znajduje się bariera. Wewnątrz bloku mamy wydzielone zadania do wykonania niezależnie przez kolejne z wątki. W efekcie grupa (pula) wątków pobiera kolejno zadania z kolejki i wykonuje je.

Ciekawą konstrukcją którą można zastosować do pętli jest ***#pragma omp for***. Przykład:

```
#pragma omp for
for (int i=0; i<n; ++i)
 Funkcja(tab[i]);
```

Wykonanie pętli dzielone jest pomiędzy wątki. Nie ma żadnej gwarancji co do kolejności wykonania iteracji (np. Funkcja dla i=4 może być wcześniej wykonana niż dla i=3). Sama konstrukcja ma też dużo ograniczeń co do typu zmiennej u nas nazwanej ***i*** i operacji porównania będącej warunkiem wyjścia z pętli.

5. Struktura wielowątkowej gry

Pisząc dowolny program należy pamiętać że używanie większej ilości wątków niż jest to potrzebne to nie najlepsze rozwiązanie. Najoptymalniej jest utworzyć tyle wątków ile sprzętowo wykonuje dana platforma + wątki, które większość czasu czekają na zakończenie blokowalnych funkcji systemu operacyjnego. Program należy tak zaprojektować aby zminimalizować ilość koniecznych synchronizacji wątków. Memory barriers mają swój narzut czasowy, jednak co najważniejsze większa separacja to mniej potencjalnych błędów.

5.1. Platformy sprzętowe

Tabela 5.1. Platformy sprzętowe dla gier

Platforma	R	W/R	W	Uwagi
Pentium 4, Athlon, Athlon64	1	1	1	
Pentium 4 HT	1	2	2	
Pentium D	2	1	2	FSB do komunikacji
Athlon64 X2	2	1	2	
Core 2 Duo	2	1	2	Shared cache L2
Pentium D Extreme	2	2	4	
Core 2 Quad	4	1	4	Shared cache L2, FSB do komunikacji
AMD 4x4	4	1	4	Dwa fizyczne procesory
AMD „Barcelona”	4	1	4	Shared cache L3 (premiera Q3 2007)
AMD 4x4 z AMD “Barcelona”	8	1	8	Dwa fizyczne procesory, Shared cache L3, (premiera Q3 2007)
Intel „Nehalem”	4	2	8	Shared cache L2, (premiera 2008), nie potwierdzone oficjalnie przez Intela
Xenon (XBox 360)	3	2	6	
Cell (PS3)	1PPE	2	8	
	6SPE[5]	1		

Gdzie: R – liczba rdzeni, W/R – liczba wątków sprzętowo wykonywanych przez rdzeń (HyperThreading), W – łączna liczba wątków wykonywanych sprzętowo przez system.

HyperThreading – w celu lepszego wykorzystania jednostek wykonawczych danego rdzenia dodano możliwość sprzętowego wykonania dwóch wątków. Łącznie oba wątki mogą wykonać więcej instrukcji jednak w porównaniu z dwoma oddzielnymi rdzenia, wydajność jest znacznie gorsza, należy o tym pamiętać jeżeli ustawiamy ręcznie koligację wątków

Shared cache – pamięć cache jest wspólna dla kilku rdzeni, unika się w ten sposób dublowania tych samych, umożliwia dynamiczny podział ilości pamięci cache potrzebny dla danego wątku w zależności od aktualnych potrzeb

FSB używane do komunikacji pomiędzy rdzeniami niektórych procesorów Intela oznacza dłuższy czas synchronizacji danych

Cell – nowa koncepcja zastosowana przy projektowaniu tego procesora wymaga dodatkowych słów wyjaśnień. Procesor posiada jedną jednostkę PPE, jest to rdzeń oparty na PowerPC, dodatkowo otrzymujemy jednostki

5 Cell produkowany jest z 8 jednostkami SPE, jednak jedna jest domyślnie blokowana a druga zajęta jest dla systemu operacyjnego

SPE które mają własny zestaw instrukcji (inny niż PPE) i są zoptymalizowane do wykonania SIMD

5.2. DirectX i OpenGL

Jednym z bardzo ważnych elementów większości gier jest wyświetlanie nieraz bardzo skomplikowanych scen 3D. Do tego celu wykorzystywane są DirectX albo OpenGL. Pierwsze pytanie jakie się pojawia to czy rysowanie grafiki można zoptymalizować rozdzielając je na wątki. Okazuje się że nie ma to dużego sensu jak i możliwości. Główną jednostką obliczeniową jest tutaj GPU i to właśnie operacje na GPU zajmują większość czasu. Dodatkowo same biblioteki nie są przygotowane do wykonywania poleceń z różnych wątków.

W przypadku korzystania z implementacji OpenGL pod Windows podczas inicjalizacji programu jedną z rzeczy, które robimy jest wywołanie funkcji wglMakeCurrent [19]. Jej zadaniem jest wybranie aktualnego kontekstu OpenGL ale tylko dla wątku wywołującego tą funkcje. Co więcej kontekst nie może być dzielony pomiędzy różne wątki.

DirectX udostępnia specjalną flagę *D3DCREATE_MULTITHREADED*, którą możemy przekazać jako parametr funkcji CreateDevice. Wtedy odwołania do Direct3D są „thread-safe". Niestety okazuje się że aktualna implementacja do synchronizacji wykorzystuje globalną sekcje krytyczną co może spowodować nawet spadek wydajności.

5.3. QueryPerformanceCounter

Przy pisaniu gier bardzo często istnieje potrzeba dokładnego odmierzenia czasu np. pomiędzy dwoma generowanymi klatkami animacji. W Windows najlepszymi funkcjami do tego są ***QueryPerformanceCounter*** i ***QueryPerformanceFrequency***. Jednak aby te funkcje działały zgodnie z naszym oczekiwaniem musimy pamiętać o dwóch rzeczach [20]: powinny być wywołane z tego samego wątku, a wątek powinien być przypisany do konkretnego procesora. Można to zrobić wykorzystują funkcję Windows API:

```
SetThreadAffinityMask(GetCurrentThread(), 1);
```

5.4. Podział zadań na wątki

Wiadomo już że najlepszym rozwiązaniem jest przydzielenie głównemu wątkowi zadanie wyświetlania grafiki. Na własne wątki zasługuje również dźwięk, operacje na socketach, operacje odczytu/zapisu na dysku. Natomiast funkcje do kompresji/dekompresji danych, AI i fizyki dobrze jest zrobić na tyle uniwersalnie aby można było je podzielić na kilka wątków. Pozwoli to na lepsze wykorzystanie całej mocy procesora.

6. Zakończenie

W referacie przedstawiono głównie teorie dotyczącą programowania wielowątkowych programów ze szczególnym uwzględnieniem gier. Właśnie takie informacje najtrudniej znaleźć. Do bibliotek takich jak Boost::Thread czy OpenMP istnieje świetna dokumentacja, jednak żeby ją w pełni zrozumieć potrzebna jest teoria.

Bibliografia

[1] Al. Williams „Programowanie Windows 2000 czarna księga" wydawnictwo Helion
[2] Neil Matthew, Richard Stones „Beginning Linux Programming" Wiley Publishing, Inc.
[3] International Standard ISO/IEC 14882:2003 „Programming languages — C++"
[4] AMD64 Architecture Programmer's Manual Volume 1: Application Programming http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf
[5] Specyfikacja PowerISA 2.03 http://www.power.org/resources/downloads/PowerISA_203.Public.pdf
[6] Memory barriers, http://en.wikipedia.org/wiki/Memory_barrier
[7] Boost::Threads http://www.boost.org/doc/html/threads.html
[8] Thread Safety in the Standard C++ Library, http://msdn2.microsoft.com/en-us/library/c9ceah3b(VS.80).aspx
[9] Thread-safety for SGI STL, http://www.sgi.com/tech/stl/thread_safety.html
[10] State of C++ Evolution, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2142.html
[11] Sequencing and the concurrency memory model for C++0x, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2052.htm
[12] A Less Formal Explanation of the Proposed C++ Concurrency Memory Model, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2138.html
[13] Thread-local storage http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2147.html
[14] Multithreading API for C++0X - A Layered Approach, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html
[15] An Atomic Operations Library for C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2047.html
[16] Exception Propagation across Threads http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2107.html
[17] OpenMP 2.5 specification, http://www.openmp.org/drupal/mp-documents/spec25.pdf
[18] Cell Architecture Explained http://www.blachford.info/computer/Cell/Cell0_v2.html
[19] wglMakeCurrent, http://msdn2.microsoft.com/en-US/library/ms537558.aspx
[20] Game Timing and Multicore Processors, http://msdn2.microsoft.com/en-us/library/bb173458.aspx

Programming multi-threaded games in C++

Due to popularization of computers with multi-core processors and new generation consoles (XBox360, PS3), a new challenge for game developers appears: the designing of multi-threaded games. Unfortunately the task of separating and the synchronization of threads is not so simple. It is easy to make a mistake and finding those mistakes can be the biggest challenge. The worst thing is that without basic knowledge you can make mistakes, of which the effects, you can see only in certain specific situations. This paper contains all of the basic information about threads and synchronization. With additional library documentation you can correctly write a multi-threaded program.