

Sztuka Wytwarzania Oprogramowania

Wykład 10 - współbieżność. Współbieżne wzorce projektowe, cz 2

Robert Nowak

25L

Plan wykładu

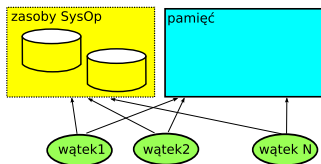
- ▶ procesy, wątki, wyścigi, blokady, skalowalność
- ▶ obsługa wejścia/wyjścia, pętla obsługi zdarzeń, asynchroniczna obsługa wejścia-wyjścia, boost::asio, reaktor, proaktor
- ▶ operacje atomowe, std::atomic, algorytmy bez blokad (lock-free)
- ▶ powtórzenie przed kolokwium 2.

powtórzenie: procesy, wątki, wyścigi, blokady (1)

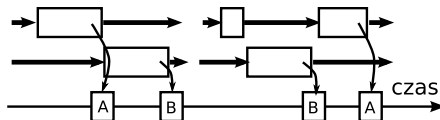
- ▶ Procesor działa znacznie szybciej, niż inne urządzenia
- ▶ współczesne komputery są wieloprocessorowe (wielordzeniowe)

Wątek realizuje niezależne ciągi instrukcji w ramach procesu.

- ▶ wątki współdzielą kod, dane oraz zasoby.
- ▶ mechanizm przełączania nie wprowadza dużych narzutów



W aplikacjach współbieżnych mogą wystąpić wyścigi.

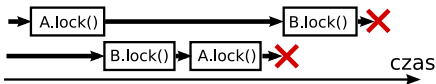


powtórzenie: procesy, wątki, wyścigi, blokady (2)

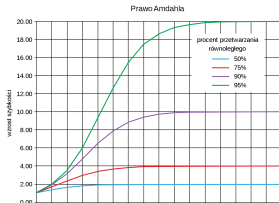
Wyścigom można zapobiegać wykorzystując sekcje krytyczne (blokady).

```
#include <mutex>
mutex.lock();
//Sekcja krytyczna, w danej chwili dostęp ma tylko jeden wątek
mutex.unlock();
```

Niewłaściwe stosowanie blokad może prowadzić do zakleszczeń.



Skalowalność:



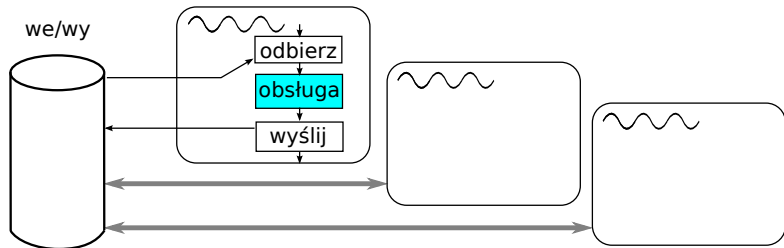
Obsługa urządzeń wejścia wyjścia

System operacyjny zapewnia 3 metody obsługi urządzeń wejścia/wyjścia:

- ▶ synchroniczne blokujące - sterowanie wraca, gdy operacja jest zakończona
- ▶ synchronicznie nieblokujące - sterowanie wraca natychmiast z informacją, czy udało się zrealizować operację (np. z informacją o ilości odczytanych bajtów), użytkownik może operację ponowić
- ▶ asynchroniczne - sterowanie wraca natychmiast, użytkownik dostarcza uchwyt (handler, callback), który będzie wołany, gdy operacja się zakończy

Obsługa urządzeń bazująca na wątkach

Wykorzystuje synchroniczne metody obsługi wejścia/wyjścia

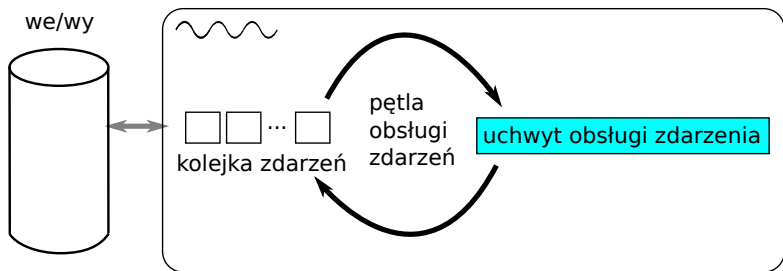


Właściwości:

- ▶ sekwencyjna (prosta) struktura sterowania
- ▶ narzuty: synchronizacja, blokady, mechanizm przełączania wątków

Obsługa urządzeń bazująca na zdarzeniach

Wykorzystuje synchroniczne lub asynchroniczne metody obsługi wejścia/wyjścia



- ▶ odwrócenie sterowania: rejestrujemy uchwyty, które będą wykonywane
- ▶ brak kontroli nad kolejnością obsługi
- ▶ brak narzutów na synchronizację, blokady, wątki

Boost.Asio - synchroniczne/asynchroniczne we/wy w C++

Przenośna obsługa:

- ▶ zegarów (timer)
- ▶ gniazd (socket) UDP, TCP (oraz strumieni z nimi związanych)
- ▶ portów szeregowych
- ▶ sygnałów
- ▶ synchronicznych/asynchronicznych operacjach na uchwytach do plików

Wsparcie dla:

- ▶ Win64, Win32 (np. Windows NT), Windows 95, 98, Me
- ▶ Linux (jądra od 2.4), 32 i 64bit,
- ▶ Mac OS X, Solaris, QNX i inne.

Boost.Asio - zegary

```
//Funkcja wołana, gdy zajdzie odpowiednie zdarzenie
void event(const boost::system::error_code&) {
    cout << "timer 2 event" << endl;
}

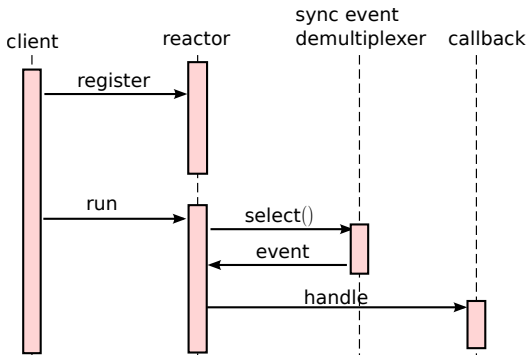
int main() {
    boost::asio::io_service io; //obiekt obsługujący zdarzenia
    boost::asio::deadline_timer t1(io, boost::posix_time::seconds(3));
    t1.wait(); //rejestracja zdarzenia i wykonanie synchroniczne
    cout << "timer 1 event" << endl;

    boost::asio::deadline_timer t2(io, boost::posix_time::seconds(3));
    t2.async_wait(event); //rejestracja zdarzenia i wykonanie synchr.
    io.run(); //obsługa zdarzeń asynchronicznych
    return 0;
}
```

Wzorzec projektowy reaktora (reactor)

Obsługa urządzeń oparta o zdarzenia. Odwrócenie sterowania.

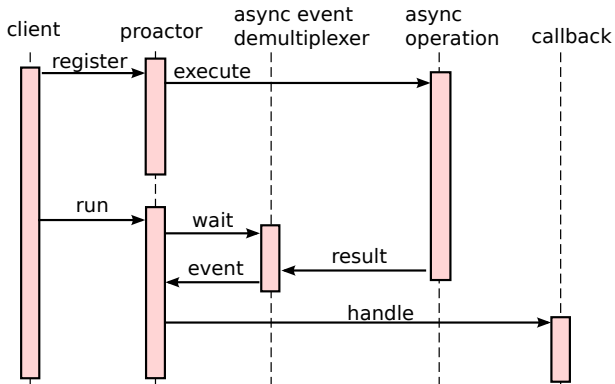
Wykorzystuje nieblokujące, synchroniczne operacje wejścia-wyjścia.



Wzorzec projektowy proaktora (proactor)

Obsługa urządzeń oparta o zdarzenia. Odwrócenie sterowania.

Wykorzystuje asynchroniczne operacje wejścia-wyjścia.



Boost.Asio, porty szeregowo

```
using Time = steady_timer::duration; using Error = const error_code&;
class SerialPort {
public:
    SerialPort(const string& n, Time t):io(), timeout(t), port(io, n), timer(io){}
    ~SerialPort() { port.close(); }
    const vector<char>& read_n(int n) { //petla obsl. zdarzen
        timer.expires_from_now( timeout);
        timer.async_wait( [=](Error e){this->timerEvent(e);});
        readSomeCall(n);
        io.run();
        return buffer;
    }
private:
    io_service io_; Time timeout;
    serial_port port; //port szeregowy
    steady_timer timer; //zegar
    char tab [1]; //bufor transmisji
    vector<char> buffer; //bufor odczytu
    //...
```

Boost.Asio, porty szeregowo (2)

```
//...
void readSomeCall(int n) {
    port.async_read_some(tab, [=](Error e, int n){this->readEvent(e,n);});
}
void readEvent(Error error, int n) {
    if( error ) return; //jeżeli wystąpił błąd odczytu
    buffer_.push_back(tab[0]);
    if(buffer_.size()>=n){ //koniec czytania
        timer.cancel();
        return;
    }
    readSomeCall(n);
}
void timerEvent(Error error) {
    if( error ) return; //jeżeli zdarzenie wycofane, to nic nie rób
    port.cancel(); //wygeneruje zdarzenie odczytu z błędem
}
};
```

Instrukcje atomowe

- ▶ niektóre operacje są transakcjami (wykonują się w całości, albo wcale)
- ▶ za pomocą takich operacji możemy tworzyć algorytmy współbieżne bez blokad (lock-free)
- ▶ takie algorytmy mogą działać szybciej, niż tworzenie sekcji krytycznych.

Operacje atomowe: gwarantuje sprzęt (procesor), ale nie tylko!

```
int x = 0;
++x; //nie jest atomowa!
//1.odczyt z pamięci do rejestru, 2.wykonanie ++x, 3.zapis do pamięci
```

Przykłady dla C++ `std::atomic`.

- ▶ typy danych, które mogą być atomowe,
- ▶ narzuty przy operacjach na typach atomowych,
- ▶ algorytmy bez blokad.

Instrukcje atomowe - przykład (1)

```
std::atomic<int> x(0);  
++x; //instrukcja atomowa, blokujący dostęp do pewnego obszaru pamięci
```

CPU registers

CPU registers

L1 cache

L1 cache

L2 cache

L2 cache

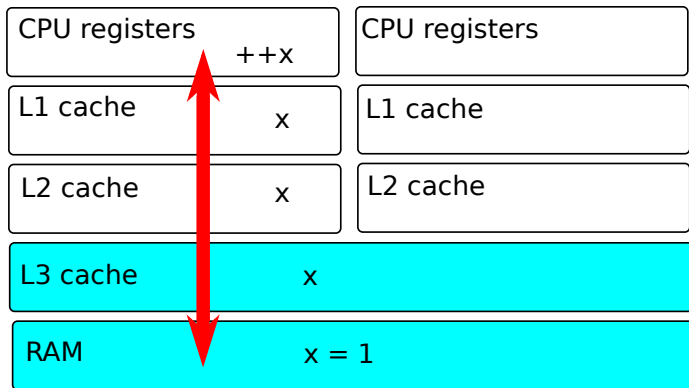
L3 cache

RAM

x = 0

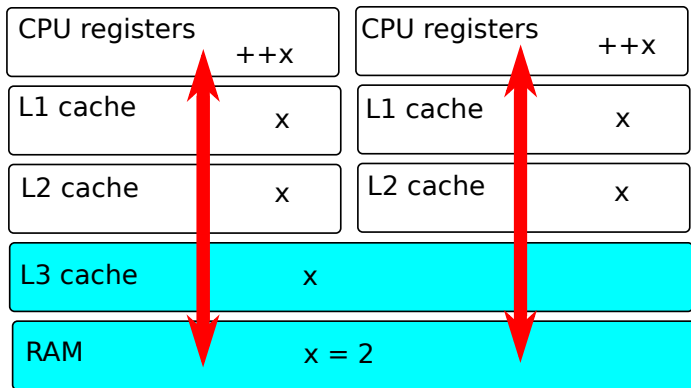
Instrukcje atomowe - przykład (1)

```
std::atomic<int> x(0);  
++x; //instrukcja atomowa, blokujący dostęp do pewnego obszaru pamięci
```



Instrukcje atomowe - przykład (1)

```
std::atomic<int> x(0);  
++x; //instrukcja atomowa, blokujący dostęp do pewnego obszaru pamięci
```



Typy danych, które mogą być atomowe

Atomowe mogą być wszystkie typy, dla których kopiowanie jest trywialne (obiekt to ciągły obszar pamięci, nie ma funkcji wirtualnych)

```
std::atomic<int> i=0; //OK
std::atomic<double> d=0.0; //OK
struct A { long x; long y; };
std::atomic<A> a; //OK, nie zawsze lock-free!
```

Operacje atomowe na typach atomowych:

is_lock_free	bada, czy typ nie używa blokad
(constructor)	tworzy obiekt
operator=	
store	zast. wartość atomową, argument nieatomowy
load	zwraca wartość atomową jako obiekt nieatomowy
compare_exchange_weak	podstawowa operacja dla algorytmów lock-free
compare_exchange_strong	podstawowa operacja dla algorytmów lock-free

Specjalne typy atomowe

<code>atomic<Integral></code> (bool, char, short, int, long, ... unsigned int, ...)	<code>operator++</code> , <code>operator--</code> , <code>operator+=</code> , <code>operator-=</code> , <code>operator&=</code> , <code>operator =</code> , <code>operator^=</code> <code>fetch_add</code> , <code>fetch_sub</code> , <code>fetch_and</code> , <code>fetch_or</code> , <code>fetch_xor</code>
<code>atomic_signed_lock_free</code> , <code>atomic_unsigned_lock_free</code>	wydajne
<code>atomic<float></code> , <code>atomic<double></code> , <code>atomic<long double></code>	<code>fetch_add</code> , <code>fetch_sub</code>
<code>atomic<U*></code> , <code>atomic<shared_ptr></code> , <code>atomic<weak_ptr></code>	wskaźnik i sprytne wskaźniki, operacje na licznikach będą atomowe

Struktury danych bez blokad (lock-free)

```
bool compare_exchange(atomic<T>* obj, T* exp, T val)
    if(obj==exp) { obj = val; return true;}
    else { exp = obj; return false; }
```

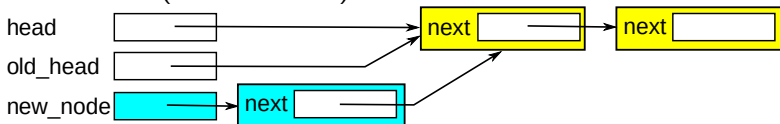
```
struct Node { int value; Node* next; };
std::atomic<Node*> head(nullptr);
```

```
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}
```

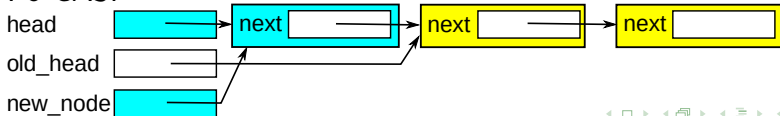
Struktury danych bez blokad (lock-free) (2)

```
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}
```

Przed CAS (bez konfliktu):



Po CAS:



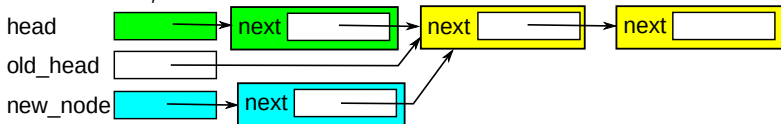
Struktury danych bez blokad (lock-free) (3)

```

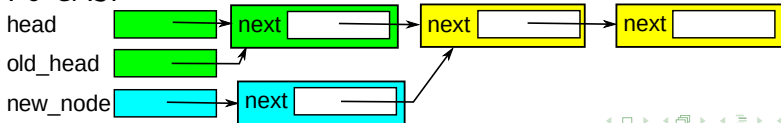
void push_front(int val) { //może być wołane w różnych wątkach
    Node* old_head = head.load();
    Node* new_node = new Node {val,old_head};
    //założenie - konflikty występują rzadko
    while (! head.compare_exchange_weak(old_head,new_node))
        new_node->next = old_head;
}

```

Przed CAS, z konfliktem:



Po CAS:



Instrukcje atomowe - podsumowanie

- ▶ Operacje atomowe są znacznie wolniejsze niż te same operacje nieatomowe, nawet dla typów takich jak 'int' (czekają na dostęp do linii cache),
- ▶ warto je stosować do tworzenia współbieżnych struktur danych (lock-free),
- ▶ algorytmy lock-free są trudne do implementacji
- ▶ algorytmy lock-free są także trudne do zrozumienia.

Operacje atomowe mogą być szybsze lub wolniejsze niż sekcje krytyczne.

Powtórzenie

Zakres kolokwium nr 2

- ▶ Metryki
 - ▶ kodu
 - ▶ procesu
- ▶ Pokrycie kodu
- ▶ Jakość testów
- ▶ Analiza oprogramowania
 - ▶ analiza dynamiczna
 - ▶ analiza statyczna
- ▶ Refaktoring
 - ▶ zapachy kodu
 - ▶ jak poprawiać kod
- ▶ Współbieżność
 - ▶ wątki, wyścigi, skalowalność
 - ▶ blokady, zakleszczenia
 - ▶ operacje atomowe (np. `std::atomic`)
 - ▶ obsługa we/wy
- ▶ Wzorce projektowe
 - ▶ RAII,
 - ▶ blokada z podwójnym sprawdz.
 - ▶ pasywny obiekt
 - ▶ reaktor, proaktor

Wyścig?

```
using Counter = int;
struct MTCounter { //wsk. na Counter z mutexem
    MTCounter() : counter(new Counter(0)) {}
    void inc() { lock_guard lock(m_); *counter += 1; }
    int get() { lock_guard lock(m_); return *counter; }
    shared_ptr<Counter> counter; //wskaźnik
    mutex m_;
};
struct Thread {
    Thread(MTCounter counter) : c(counter) {}
    void operator()() { for(int i=0;i<1000000;++i) c.inc(); }
    MTCounter c;
};
int main() {
    MTCounter counter; Thread t1(counter), t2(counter);
    thread thrd1(ref(t1)), thrd2(ref(t2));
    thrd1.join(); thrd2.join();
    return 0;
}
```

Rozwiązanie

```
//Rozwiązanie - sekcja krytyczna
struct CounterSync {
    CounterSync() : value(0) {}
    int value; mutex m;
};
struct MTCounterSync {
    MTCounterSync() : counter(new CounterSync) {}
    void inc() { lock_buard lock(counter->m); counter->value += 1; }
    int get(){ lock_guard(counter->m); return counter->value; }
    shared_ptr<CounterSync> counter; //współdzielony licznik
};
//Rozwiązanie - licznik wykorzystuje operacje atomowe
using CounterAtomic = std::atomic<int>;
struct MTCounterAtomic {
    MTCounterSync() : counter(new CounterAtomic(0)) {}
    void inc() { *counter += 1; }
    int get(){ return *counter; }
    shared_ptr<CounterAtomic> counter; //współdzielony licznik
};
```

Zadanie 2: popraw wydajność. Funkcja main

```
class Data { /* składowe nieistotne */}; using PData = shared_ptr<Data>;
struct Out {
    Out(int size) : size_(size) {}
    int size_;
    std::vector<PData> v_;
    mutex m_;
};
int main() {
    Out out(10000);
    boost::asio::io_service io;
    boost::asio::deadline_timer t01(milliseconds(8) );
    //...
    boost::asio::deadline_timer t99(milliseconds(8) );
    t01.async_wait([&](const error_code& e){ serve_event(out,t01,e);});
    //...
    t99.async_wait([&](const error_code& e){ serve_event(out,t99,e);});
    io.run();
    return 0;
}
```

Zadanie 2: popraw wydajność. Obsługa urządzenia

```
struct Out { //powtórzona z poprzedniego slajdu
    Out(int size) : size_(size) {}
    int size_;
    std::vector<PData> v_;
    mutex m_;
};

void serve_event( Out& out, deadline_timer& t, const error_code& error) {
    if( error ) return;
    //obliczenia lokalne, tworzy dane dla nowej paczki
    lock_guard<mutex> guard(out.m_);
    PData data = PData(new Data()); //tworzy nową paczkę, kopiuje dane
    out.v_.push_back(data);
    if(out.v_.size() < out.size_) {
        t.async_wait([&](const error_code& error){ serve_event(out,t,error);});
    }
}
```

Zadanie 2: popraw wydajność. Rozwiązanie

```
struct Out { //powtórzona z poprzedniego slajdu
    Out(int size) : size_(size) {}
    int size_;
    std::vector<PData> v_;
    mutex m_;
};

//pętla zdarzeń, nie ma konieczności stosowania sekcji krytycznej!
void serve_event( Out& out, boost::asio::deadline_timer& t, const error_code& error ) {
    if( error ) return;
    //obliczenia lokalne, tworzy dane dla nowej paczki
    PData data = PData(new Data()); //tworzy nową paczkę, kopiuje dane
    out.v_.push_back(data);
    if(out.v_.size() < out.size_) {
        t.async_wait([&](const error_code& error){ serve_event(out,t,error);});
    }
}
```

Dziękuję

robert.nowak@pw.edu.pl