Zawartość:

- wątek

- zasady pracy z wątkami

- metody otwierania wątków

- problemy wynikające z wielowątkowości i metody radzenia sobie z nimi

Wątek -> jednostka wykonująca kod programu

Każdy uruchamiany program dostaje minimum jeden wątek, który wykonuje ten program.

```cpp
#include <cstdio>
#include <windows.h>

int main(){
    unsigned long long counter = 0;
    for(;;){
        printf("Hello World %u\n", ++counter);
        Sleep(2000);
    }
}
```

```
Type "apropos word" to search for commands related to "word" ...
Reading symbols from .\sleep.exe...
(gdb) r
Starting program:
[New Thread 7656.0x30d0]
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
```

Program może stworzyć nową jednostkę wykonania czyli wątek i wskazać jej kod, który ma wykonywać.

Każdy wątek ma ID najczęściej w postaci liczby.

Wątki można podzielić na dwie grupy:

- CPU bound – wątki wykonujące skomplikowane obliczenia, wątki takie wykorzystują cały przydzielony im czas procesora
- IO bound – wątki współpracujące z urządzeniami IO (są aktywne tylko wtedy gdy te urządzenia tego wymagają), wątki te gdy nic nie robią oddają procesor dla innych wątków, do tej grupy zaliczają się również wątki nie korzystające z IO, ale posiadające w kodzie funkcje typu Sleep()

Przełączanie wątków pracujących równolegle zachodzi gdy:

- Wątek wykorzysta cały przydzielony mu czas
- Wątek dobrowolnie zrezygnuje z pozostałego przydzielonego mu czasu (IO bound)

Algorytm przełączania:

1. Zatrzymanie aktualnie pracującego wątku

2. Zapisanie jego kontekstu
3. Wybranie wątku do uruchomienia
4. Wczytanie jego kontekstu
5. Uruchomienie wątku

Narzędzia do programowania wielowątkowego:

- WinAPI (tylko Windows)
- thrd_create (język C)
- std::thread (język C++)
- threading.Thread (Python)
- Thread (Java)
- Task (C#)

Zmienna lokalna wątku – zmienna znajdująca się w przestrzeni globalnej (nazwa jest myląca). Każdy w programie ma swoją kopię tej zmiennej mimo, że każdy odwołuje się do za pomocą tego samego identyfikatora.

Mówiąc prościej tworzymy w globalnej przestrzeni np. zmienna typu integer o nazwie „licznik" i przypisujemy do niej wartość 0. Każdy wątek odwołuje się do niej za pomocą identyfikatora „licznik" jednak każdy czyta i zmienia inne miejsce w pamięci. Jeżeli otworzymy 10 wątków i każdy zrobi na tej zmiennej kod:

licznik++;

to w pamięci będziemy mieli 10 razy wartość 1 (na początku było 0) a nie raz 10 mimo, że w kodzie widzimy tylko jedną definicje zmiennej „licznik".

Zasady pracy z wątkami:

- Każdy wątek podrzędny (otwarty przez programistę w trakcie trwania programu) powinien się zakończyć przed zakończeniem wątków głównego. Jeżeli otworzymy wątek, który wykonuje długo jakieś bardzo skomplikowane obliczenia i nie każemy głównemu wątkowi na niego poczekać to główny wątek skończy się wcześnie a razem z nim cały program (funkcja main dobiegnie końca mimo, że otwarty wątek coś jeszcze robi). W takiej sytuacji system „ubije" wszystkie wątki podrzędne. Należy posłużyć się barierą synchronizującą.
- Jeżeli dzielimy problem na kilka mniejszych problemów tak by każdy uruchomić na osobnym wątku (skracając w ten sposób czas trwania rozwiązywania problemu) należy dokonać podziału na jak najbardziej równe kawałki. W przeciwnym wypadku zysk ze zrównoleglenia nie będzie optymalny.
- Nie każdy problem da się podzielić. Jeżeli wątki muszą korzystać w swoich obliczeniach z wyników obliczeń innych wątków to zrównoleglenie danej części problemu nie jest możliwe.
- Wątki mogą zakończyć się w różnej kolejności. Należy wziąć to pod uwagę sklejając rozwiązania pomniejszych problemów w całość.

- Nie należy używać wątków do banalnych zagadnień. Otwarcie wątku oraz jego synchronizacja też kosztuje trochę czasu, więc jeżeli będziemy otwierać wątki dla prostych funkcjonalności możemy na tym stracić.

Zasady pracy z wątkami jeżeli operują one na pamięci:

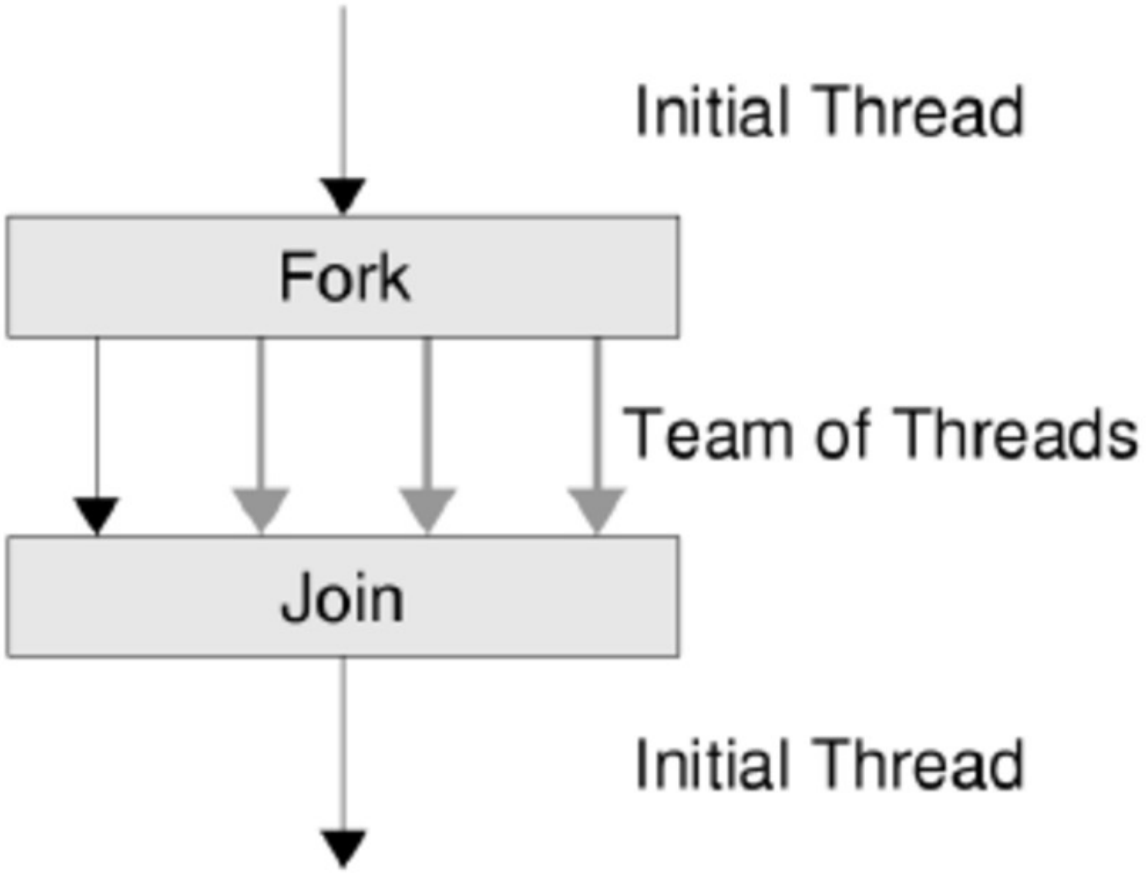
- Nie może dojść do sytuacji, w której więcej niż jeden wątek wykonuje operację na tej samej zmiennej chyba, że jest to operacja atomowa.
- Powyższa zasada nie dotyczy sytuacji gdy każdy wątek operuje na swojej komórce tablicy (każda komórka jest tak naprawdę zmienną)

Synchronizacja – operacja polegająca na poczekaniu, aż konkretny wątek skończy swoje działanie i dopiero ruszeniu do przodu.

Przykład synchronizacji:

Na początku funkcji main otworzyliśmy 3 wątki i one działają sobie w tle. Funkcja main po ich otwarciu biegnie dalej ze swoim kodem i dochodzi do jego końca. Należy przed instrukcja „return 0” wstawić blok, który poczeka na zakończenie otwartych wcześniej wątków tak by nie zaszła sytuacja, że wątek główny kończy się przed wątkami podrzędnymi i te zostają zabite przez system.

Analogicznie należy postąpić jeżeli chcemy na wątku głównym (np. w mainie) skorzystać z wyników działania wcześniej otwartych trzech wątków. Przed kodem wykorzystującym wyniki należy postawić synchronizację tak by mieć pewność, że w momencie użycia danych wyliczonych przez wątki te ich obliczanie dobiegło końca.



Na powyższym schemacie czarna strzałka to wątek główny. Fork to otwarcie trzech dodatkowych wątków. Join to bariera, która ponownie łączy ścieżki w jedną.

Problem nierównego podzielenia zadań.

Jeden z pod wątków dostał do zrobienia wyraźnie więcej niż pozostałe przez co nie korzyść ze zrównoleglenia nie jest maksymalna.

Przykład:

Zadanie wykonuje się 30 sekund na jednym wątku, ale programista podzielił je na trzy części dla trzech wątków. Jeżeli przydzieli każdemu wątkowi część trwającą około 10 sekund całość wykona się w mniej więcej w 10 sekund (część między fork a join). Jeżeli 1 i 2 wątek dostaną części po 5 sekund a 3 wątek dostanie 20 sekundowy kawałek całość potrwa 20 sekund.

Prawdziwy zysk ze zrównoleglenia:

Załóżmy, że otwarcie jednego wątku trwa 1 sekunde. Synchronizacja i zamknięcie wątku trwa 2 sekundy.

Dostajemy zadanie, które trwa 15 sekund i dzielimy je na 3 wątki gdzie każdy dostaje mniej więcej 1/3 do wykonania czyli po 5sekund.

Czas rozwiązania problemu wyniesie 1*3 + 5 + 2*3 czyli 14 sekund. Jak widać zysk jest znikomy.

Co innego gdy mamy zadanie trwające np. 45 sekund i dzielimy je na 3 wątki po 15 sekund.

Czas rozwiązania problemu wyniesie 1*3 + 15 + 2*3 czyli 21 sekund co jest dużym skróceniem czasu wykonania w porównaniu do 45 sekund.

Wiele zadań edukacyjnych na zajęciach łamię tę regułę. Jednak w zadaniach tych ważniejsze jest pokazanie składni oraz co jak działa niż faktyczny zysk.

Odczyt i zapis do tej samej zmiennej.

Problem ten polega na tym, iż wiele wątków próbuje zapisywać (ustawiać wartość, inkrementować, dekrementować itp.) tę samą zmienną.

Sytuacja ta w 99% przypadków skończy się błędnym wynikiem.

Przykład:

Wątek numer 1 inkrementuje (zmienna++) zmienną co 2 sekundy w pętli.

Wątek numer 2 sprawdza czy dana zmienna jest parzysta lub nie i wyświetla odpowiedni komunikat również co 2 sekundy.

Program ten w oczekiwaniu programisty powinien wyświetlać na zmianę informacje o parzystości i nieparzystości. Jednak tak nie będzie.

Problem :

Oczywiście mimo, że ustawiono taki sam delay w postaci 2 sekund nie gwarantuje, że wątki będą lub nie będą próbowały odnieść się do zmiennej w tym samym momencie. Może dojść do sytuacji, że wątek jeden podmienił już dwa dolne bajty z czterech na nową wartość, ale nie zdążył podmienić dwóch pozostałych bo wątek 2 odczytał wartość (zakładamy, że zmienna jest typu int czyli ma 4 bajty). W konsekwencji wątek 2 dostaje dwa nowe bajty i dwa stare, które tworzą błędną liczbę co przekłada się na komunikat inny niż oczekiwano.

Rozwiązania:

- Jeżeli to możliwe, każdy wątek powinien pisać do swojej zmiennej lokalnej a następnie po zakończeniu działania wątków należy na głównym wątku dokonać złączenia. W powyższym przykładzie zastosowanie tego rozwiązania jest nie możliwe.
- Należy wykorzystać operacje atomowe. Operacje atomowe to operacje, które nie da się rozbić na części. W tym przypadku operacja inkrementacji mogła by być nie podzielna (zmienia wszystkie bajty). To samo tyczy się operacji odczytu. W wyniku zastosowania

operacji atomowych wątki czekają na zakończenie danej operacji atomowej drugiego wątku przed rozpoczęciem swoich działań.

- Należy stosować znane metody blokujące dany fragment pamięci przed użyciem przez inny wątek gdy już jakiś z niej korzysta (np. Muteksy). Przykłady w dalszej części dokumentu.

Bardzo dużym problemem jest fakt, że jeżeli procesor widzi w kodzie dwie niezależne operacje na pamięci wykonywane jedna po drugiej to niekoniecznie musi je wykonać w takiej kolejności jak to napisał programista. Problem ten występuje tak naprawdę na niskim poziomie (asembler), ale może odbić się także na językach wyższego poziomu.

Metody synchronizacji dostępu do pamięci:

Bariery pamięci – instrukcje blokujące wykonanie kolejnych instrukcji operujących na pamięci do momentu aż wszystkie występujące przed blokadą się wykonają. Przykłady: lfence, sfence, mfence. Część instrukcji sama w sobie jest taką barierą.

Operacje atomowe – operacje, których nie da się podzielić na mniejsze części. Efekt tych operacji widziany jest w całości albo w cale (podobny motyw do transakcji z baz danych).

Metody synchronizacji w językach wyższego poziomu

Spinlock – metoda polegająca na utworzeniu globalnej zmiennej będącej flagą. Przykładowa wartość 0 oznacza, że zasób którego flaga „broni" jest wolny zaś wartość 1 oznacza, że jest zajęty.

Wracając do przykładu z parzystością. Wątek inkrementujący przed inkrementacją właściwej zmiennej nadaje fladze (co ważne w sposób atomowy) wartość 1 a po zakończeniu inkrementacji wartość 0 co oznacza odpowiednio założenie blokady i zwolnienie jej. Drugi wątek przed sprawdzeniem parzystości wykonuje taki kod:

```
while (flaga == 1) {}

//test
```

Pętla blokuje wątek na czas gdy wątek modyfikujący założył blokadę. Gdy ten ściągnie blokadę pętla się kończy i wykonywany jest test.

W metodzie tej ważne jest, by operacje na fladze były atomowe. Przykład z inkrementacją nie jest tutaj najlepszy bo sama inkrementacja nie trwa za długo. Jeżeli wątek modyfikujący dokonywał by dłuższych operacji miało by to więcej sensu.

Muteks – najczęściej stosowana metoda synchronizacji. Idea muteksu również bazuje na fladze, jednak jest to mechanizm dostarczony przez bibliotekę/język dzięki czemu programista nie musi się martwić o to czy przy zmianie flagi musi posłużyć się odpowiednią synchronizacją. Dodatkowo muteks pozwala łatwo blokować wątki gdy jest ich więcej niż dwa.

Użycie muteksu polega na stworzeniu jego globalnej instancji a następnie w każdym wątku napisaniu:

```
mutex.lock();

//wrażliwy kod

mutex.unlock();
```

Muteksy pozwalają na prace z większą ilością wątków. Jeżeli w momencie wywołania metody lock() inny wątek już ma zablokowany muteks dochodzi do sporu i wątek czeka aż uda mu się założyć blokadę (uda się gdy inny wątek ją zdejmie).

Muteks można skonfigurować tak by czekał tylko określony czas lub podjął określoną ilość prób zajęcia.