Zasoby oprogramowania równoległego w architekturze SMP

Platforma Windows NT based. Proces, wątek, synchronizacja.

- **Proces jest obiektem jądra systemu operacyjnego, który reprezentuje egzemplarz programu uruchomionego. Proces się składa z:**
  - **obiektu jądra, który służy dla kierowania procesem przez system operacyjny**
  - **przestrzeń adresowa, która obejmuje kod i dane wszystkich module *.exe, *.dll, a również stos potoków i stos pamięci, alokowanej dynamicznie**

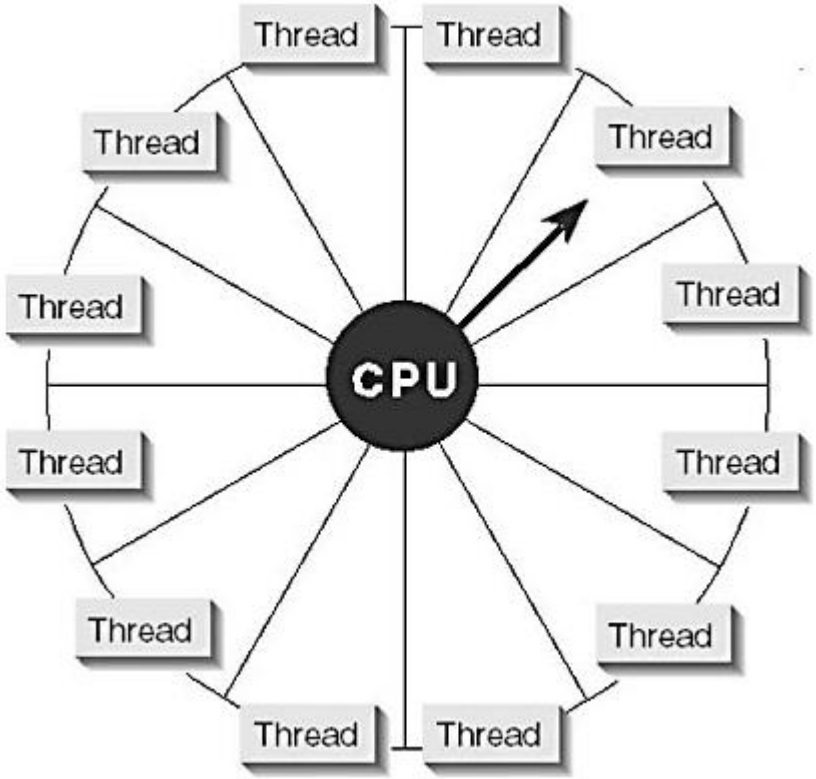
- **Sam proces nic nie działa. Dla tego, żeby proces coś wykonywał, jest koniecznie uruchomienie wątku (wątków). Właśnie wątki niosą odpowiedzialność za wykonanie instrukcji kodu, zawartego w przestrzeni adresowej procesu.**

- **Każdy wątek ma swój własny stos (stack) i w trakcie wykonania instrukcji kodu rozdziela rejestry CPU.**

- **Każdy proces ma przynajmniej jeden wątek, który wykonuje kod w przestrzeni adresowej. Jeśli zamknąć wszystkie wątki procesu, system operacyjny automatycznie zniszczy ten proces, a również i jego przestrzeń adresową.**

- **Dla wszystkich istniejących wątków system operacyjny (OS) planuje pewne odcinki czasu.**

- **CPU udziela każdemu wątkowi kwanty czasu. Dla komputera jednoprocesorowego powstaje szeregowanie cykliczne. Dla komputerów wieloprocesorowych algorytm zrównoważenia obciążenia procesorów jest bardziej skomplikowany.**



- **Kiedy tworzy się proces, system operacyjny tworzy pierwotny wątek. Ten wątek może stworzyć wielu potomnych wątków (child threads), które mogą tworzyć swoi własne potomne wątki.**

Tworzenie procesu

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,                  // name of exec. module
LPTSTR lpCommandLine,                       // command line string
LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
LPSECURITY_ATTRIBUTES lpThreadAttributes,   // SD
BOOL bInheritHandles,                       // handle inheritance option
DWORD dwCreationFlags,                      // creation flags
LPVOID lpEnvironment,                       // new environment block
LPCTSTR lpCurrentDirectory,                 // current directory name
LPSTARTUPINFO lpStartupInfo,                // startup information
LPPROCESS_INFORMATION lpProcessInformation  // process information
);
```

Wywołanie tej funkcji powoduje:

- **tworzenie obiektu jądra procesu przez system operacyjny i ustawienie licznika obiektu jądra na 1. Jądro procesu – to struktura danych, która służy do kierowania procesem i zawiera informacje statystyczną o procesie.**

- **tworzenie wirtualnej przestrzeni adresowej i załadowanie kodu i danych dla *.exe pliku i wszystkie DLLs w przestrzeń adresową procesu.**

- **OS tworzy obiekt jądra wątku i ustawia licznik obiektu jądra wątku na 1. To będzie pierwotny wątek procesu. Jądro wątku – to struktura danych, która służy do kierowania wątkiem i zawiera informacje statystyczną o wątku.**

- **Pierwotny wątek zaczyna wykonanie kodu. Jeśli przy tworzeniu procesu występuje sukces, funkcja CreateProcess zwraca TRUE.**

Zakończenie procesu

1. Funkcja, z której pierwotny wątek zaczyna wykonanie kodu, dochodzi do instrukcji return. To – najlepszy sposób zakończenia wątku:

- **Każdy obiekt C++, tworzony przez ten wątek , będzie zniszczony poprawnie, używając własny destruktor**
- **OS poprawnie zwolni stos wątku**
- **OS ustawi kod zakończenia procesu na wartość, którą zwraca funkcja wejściowa pierwotnego wątku**
- **Os zmniejszy licznik obiektu jądra procesu o 1**

2. **Jeden z wątków procesu wywołuje funkcje ExitProcess() (unikamy po możliwości tego)**
   - **Część kodu po wywołaniu ExitProcess nigdy nie będzie wykonana.**
   - **Wszystkie wątki tego procesu będą przerwane. Chocza dokumentacje SDK głosi, że wszystkie resursy wątków i procesu (stacks, heap, files) będą zwolnione poprawnie, dla programów C++ może to powodować to, że destruktory klas nie zadziałają:**

```
#include <windows.h>
#include <stdio.h>
class CSomeObj
{
 public:
        CSomeObj()  { printf("Constructor\r\n"); }
        ~CSomeObj() { printf("Destructor\r\n"); }
 };

CSomeObj g_GlobalObj;
void main ()
{
        CSomeObj LocalObj;
        ExitProcess(0);      // This shouldn't be here
                             // At the end of this function, the compiler automatically added
                             // the code necessary to call LocalObj's destructor.
                            // ExitProcess prevents it from executing.
}
```

Wydruk:

```
Constructor
Constructor
```

Jeśli ExitProcess() usunąć z kodu, wydruk będzie taki:

Constructor
Constructor
Destructor
Destructor

- **Funkcja ExitProcess() przeznaczona dla kasowania procesu przez swoje własne wątki.**

3. Jeden z wątków procesu wywołuje funkcje TerminateProcess(….) (unikamy po możliwości tego). Jest przeznaczona dla kasowania procesu dowolnym wątkiem (nie tylko własnym). Służy dla przymusowego (awaryjnego) zakończenia procesu, kiedy proces w skutek jakichś zdarzeń (błędów) gubi sterowanie, nie odpowiada na notyfikacji.

4. Wszyscy wątki procesu pozostałe zakończone w skutek wywołania funkcji ExitThread(..), TerminateThread (…), system operacyjny niszczy proces. To się rzadko kiedy zdarza.

- **Przy zakończeni procesu:**
  - **Każdy wątek procesu będzie zakończony**
  - **Wszystkie obiekty użytkownika, a również GDI, będą zwolnione, a obiekty jądra – zamknięte (zwolnienie pamięci, steku, zamknięcie plików).**
  - **Obiekt jądra „proces” przechodzi w stan „signaled”.**
  - **Licznik obiektu jądra zmniejsza się o 1.**

- **Przykład tworzenia procesu**

```
void CCalcExtern::StartProc()
/*=========================================================================
Starts new process and suspends the given process until new process will
be finished
=========================================================================*/
{
        PROCESS_INFORMATION pi;
        STARTUPINFO StartupInfo;
        ZeroMemory(&StartupInfo, sizeof(StartupInfo));
        StartupInfo.cb = sizeof(StartupInfo);
```

```
	DWORD dwExitCode;
	LPTSTR lpCommandLine = "Q:\\win32app\\Disp\\Debug\\Disp.exe";

	BOOL fSuccess = CreateProcess( NULL, lpCommandLine, NULL, NULL,
			FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &StartupInfo, &pi);

	if(!fSuccess)
	{
			AfxMessageBox("StartProc fault");
			throw EXCEPT_CALC;
	}

	//close thread handle
	CloseHandle(pi.hThread);

	//delay the proceeding of code until the new process will be finished
	WaitForSingleObject(pi.hProcess, INFINITE);

	GetExitCodeProcess(pi.hProcess, &dwExitCode);

	CloseHandle(pi.hProcess);

}//StartProc
```

Wątki

- **Aplikacji, które mają tylko jeden wątek, są aplikacjami jednowątkowymi. Na przykład, program Mulm – to typowa aplikacja jednowątkowa. Po naciśnięciu na przycisk „Start" wszystkie kontrolki dialogu są zablokowane: dopóki obliczenia się nie skończą – żaden przycisk nie da się nacisnąć. Niema możliwości przerwać wykonanie zadania (tylko przez menedżer zadań).**

- **A to – przykład aplikacji wielowątkowej (SOLVER, SCAD). Tu jeden wątek z bardziej wysokim priorytetem obsługuje dialog, a drugi wątek z bardziej niskim priorytetem – obliczenia. Takie podejście nadaje możliwość nie tylko informować użytkownika o etapach wykonania zadania, a i przerwać lub tymczasowo zawiesić jego wykonanie.**

- **Wielowątkowość – to jest niezbędna składowa obliczeń równoległych dla architektury SMP (Symmetric MultiProcessor systems).**

- **Wątek składa się z:**
  - **obiektu jądra (struktury danych), posługującej do kierowania wątkiem i przechowywaniem danych statystycznych**
  - **stosu wątku, który wspiera parametry wszystkich funkcji, zmienne lokalne wątku i kod zwracany wątkiem.**

- **Wątek wyznacza kolejność wykonania instrukcji kodu. Wątek wykonuje instrukcji kodu w obszarze przestrzeni adresowej procesu i operuje z danymi, umieszczonymi w przestrzeni adresowej procesu.**

- **Jeśli mamy kilka wątków tego samego procesu, to oznacza, że te wątki dzielą pomiędzy sobą przestrzeń adresową procesu. Dwa albo więcej wątków mogą wykonywać ten sam kod i operować tymi samymi danymi. Wątki mogą rozdzielać HANDLEs jądra, dla tego że tabele HANDLEs istnieje dla każdego procesu, przecież nie dla każdego wątku.**

- **Tworzenie wątku:**

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,      //SD
SIZE_T dwStackSize,                            // initial stack size
LPTHREAD_START_ROUTINE lpStartAddress,         // thread function
LPVOID lpParameter,                            // thread argument
DWORD dwCreationFlags,                         // creation option
LPDWORD lpThreadId                             // thread identifier
 );
```

- **OS tworzy obiekt jądra – wątek**
- **OS wydziela pamięć dla steku wątku z przestrzeni adresowej procesu**
- **Nowy wątek ma dostęp do wszystkich HANDLEs obiektu jądra, pamięci**

procesu, steków innych wątków tego samego procesu. Dla tego w zadaniach wielowątkowych komunikacja pomiędzy wątkami jednego procesu jest prosta.

- **<u>Funkcja wątku</u>. Każdy wątek zaczyna wykonanie kodu z funkcji wątku:**

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
        DWORD rtwResult = 0;
        //Coś wykonuje

        return(dwResult);
}
```

- **Kiedy skończy się wykonanie instrukcji funkcji wątku, wątek skończy swoją działalność.**

- **Nazwa funkcji wątku może być dowolną. Jeśli program ma kilka różnych funkcji wątku, nazwy tych funkcji powinni być różne**

- **Funkcja wątku przyjmuje tylko jeden parametr – wskaźnik typu void do danych (najczęściej – do struktury danych), które trzeba przekazać wątkowi.**

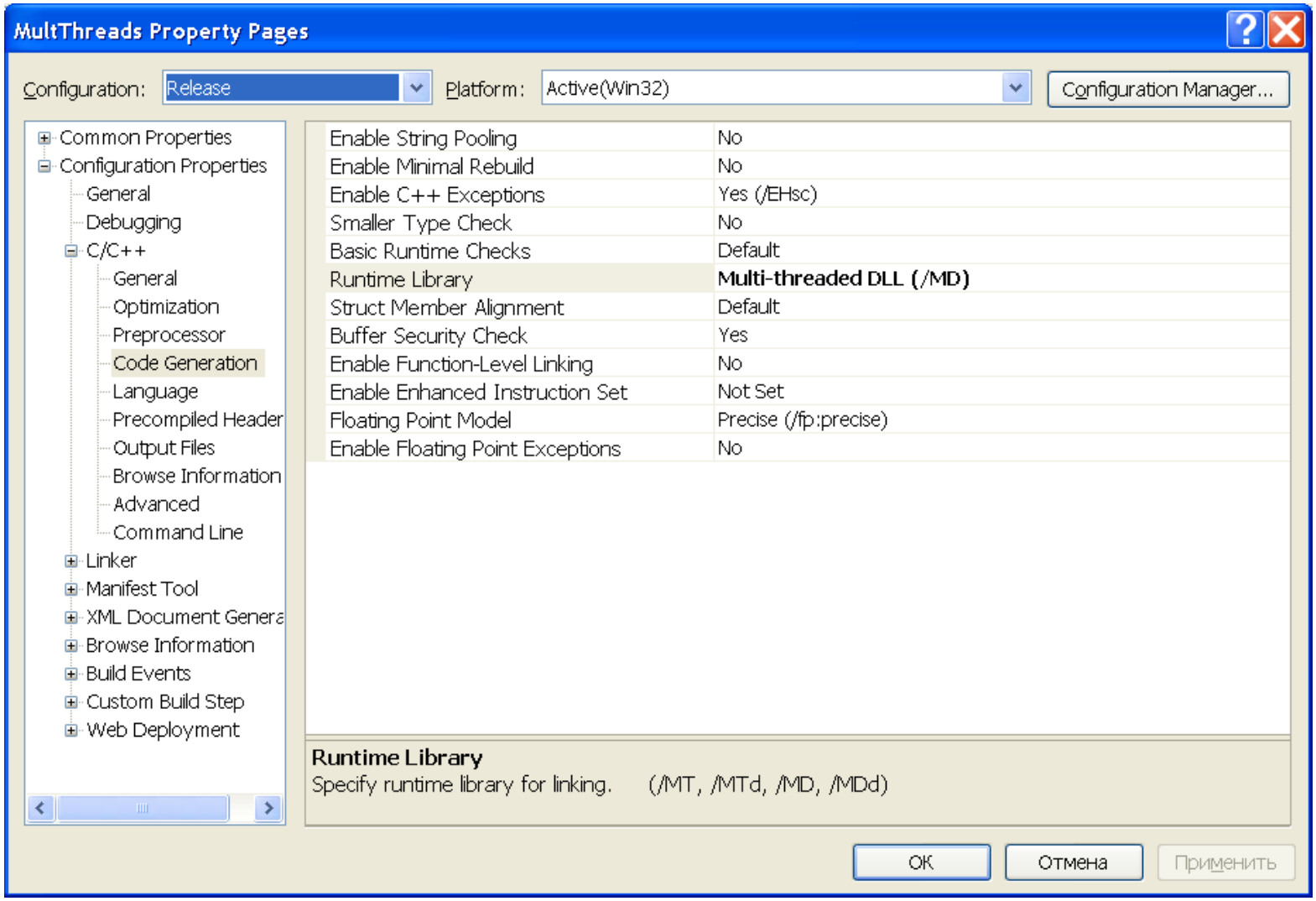
- **Funkcja wątku zwraca kod zakończenia (podobnie funkcją main, WinMain)**

- **Funkcja wątku jak najbardziej powinna używać swoje własne parametry i zmienne lokalne. Do zmiennych statycznych i globalnych mogą odwoływać odrazu wiele wątków – dla zabezpieczenia stanu tej zmiennej trzeba używać synchronizacje – komplikacje kodu, spad wydajności obliczeń, komplikowanie programu.**

- **Zmienne lokalne i parametry wątku pozostają umieszczone w steku wątku – lokalne dane różnych wątków są umieszczone w różnych adresach pamięci - dostęp do tych danych z innego wątku (tego samego procesu) jest możliwy tylko przez programistę przy napisani odpowiedniego kodu, OS dba automatycznie o wyrównani obszarów steków wątków na granice cache-linii procesorów (rozwiązanie cache-konfliktów).**

- **Pamięć dynamiczna jest resursem wspólnym, który się rozdziela pomiędzy wątkami. Dla uniknięcia cache-konfliktów, powodowanych rozdzieleniem pamięci dynamiczną, można użyć TLS (Thread Local Storage) technikę.**

➢ **Przy tworzeniu projektu aplikacji wielowątkowych trzeba ustawić C/C++ Run Time Library jako Multithreaded:**



- **Przykład tworzenia wątku: MultThreads**

- **Po tworzeniu wątku potomnego zmienna HANDLE hThread jest własnością wątku macierzystego. Wątek macierzysty może sterować wątkiem potomnym: zawieszać jego, budzić, zmieniać priorytet, skasować przymusowo.**

- **<u>Zakończenie wątku</u>.**

  1. **Funkcja wątku skończy się – wątek samo likwiduje się (OK – najbardziej pożądany sposób).**
     - **Wszystkie obiekty C++ będą poprawnie zniszczone przez wywołanie destruktorów**
     - **OS poprawnie zwalnia pamięć, zajmowaną stekiem wątku**
     - **OS ustawia kod zakończenia wątku, który zwraca funkcje wątku**
     - **OS zmniejsza licznik wątku o 1.**

  2. **Wątek sam siebie zniszczą, wywołują ExitThread(…) funkcje. Wszystkie resursy wątku będą poprawnie zwolnione, przecież destruktory klas C++ nie będą wywołane. Dla tego lepiej zwracać sterowanie z funkcji wątku (return var), a nie wywoływać jawnie ExitThread(…).**

3. **Wątek pozostaje zniszczony innym wątkiem tego samego lub innego procesu przez wywołanie funkcji TerminateThread(…) (ten sposób jest używany , kiedy wątek nie odpowiada na notyfikacje – zawis). Wywołanie TerminateThread nie notyfikuje wątek o to, że on pozostanie zniszczony, więc wątek sam nie jest w stanie zwolnić resursy. TerminateThread jest używamy również, jeśli na żądanie użytkownika działanie wątku roboczego (obliczeniowego) trzeba przerwać z wątku interfejsowego. Wtedy dla poprawnego zamykania każdego obiektu pamięć, alokowana dynamicznie, mysi być zwolniona przez wywołanie funkcji specjalnej – deallokatora. Tak samo i s zamykaniem plików.**

4. **Proces, który jest właścicielem wątku, skończy się (unikamy tego – normalnie, jeśli wszystkie wątki procesu będą zakończone poprawnie).**

- **<u>Przy zakończeniu wątku:</u>**

  - **Wszystkie HANDLEs obiektów użytkownika, który należą wątkowi, pozostają zwolnione przez OS. Większość obiektów należy procesowi i będą zwolnione przy zakończeniu procesu. Wątkowi należą tylko HANDLE okna i hooks.**

- **Kod zakończenia wątku pozostaje zmieniony z STILL_ACTIVE na kod, przekazany w funkcje ExitThread lub TerminateThread.**

- **Obiekt jądra „wątek” przechodzi w stan wolny - wątek podaje sygnał o zakończeniu**

- **Licznik obiektu jądra „ wątek” zmniejsza się o 1.**

➢ **Każdy wątek ma swój własny zestaw rejestrów procesora – Kontekst wątku. Kontekst odbija stan rejestrów procesora w momencie ostatniego wykonania wątku i zapisuje jego w strukturę CONTEXT, która znajduje się w obiekcie jądra „ wątek”. Najważniejsza informacje w kontekście wątku – to stan wskaźnika rozkazów IP i wskaźnika steku SP, które wskazują na odpowiednie adresy pamięci przestrzeni adresowej procesu.**

Szeregowanie wątków, priorytet, przyklejanie wątków do procesorów

- **OS wielozadaniowy z wywłaszczaniem – wątki z większym priorytetem wywłaszczają (odprawiają w stan idle - jałowy) wątki z mniejszym priorytetem.**

Szeregowanie wątków

- **Przez pewny okres czasu (~20 ms) OS przegląda wszystkie obiekty jądra „ wątek” i kontroluje takie, które mogą dostać czas CPU. Dalej OS wybiera jeden z takich obiektów i ładuje w rejestry procesora wartości jego kontekstu – context switching. Dla każdego wątku OS rejestruje, ile razy on był podłączony do procesora.**

- **Wątek wykonuje kod i przetwarza dane swojego procesu. Za kolejne ~20 ms OS zapisze stan rejestrów procesora w kontekst wątku, odłączy wątek od procesora. Dalej OS przeglądnie inne obiekty jądra „ wątek”, gotowe do wykonania, i powtórzy te działania z innym wątkiem.**

- **Ten cykl operacji - wybór wątku, ładowanie jego kontekstu w rejestry procesora, wykonanie, odświeżenie kontekstu, odłączenie od procesora trwa od momentu uruchomienia OS.**

- **OS planuje wykonanie tylko tych wątków, które mogą dostać czas CPU. Większość wątków znajdują się w stanie zawieszenia. Znaczna ilość wątków nie jest w stanie zawieszenia, przecież znajduje się w stanie oczekiwania (idle) – czeka jakiegoś zdarzenia, żeby się obudzić.**

<u>Sterowanie wątkami</u>

- **Zawieszenie i wznowienie wątków:**

  - DWORD SuspendThread(HANDLE hThread);
  - DWORD ResumeThread(HANDLE hThread);

  - **Wątek może zawiesić siebie sam, przecież ponowić jego może tylko inny wątek.**
  - **SuspendThread może zawiesić wykonanie programu. Na przykład, wątek zaczął alokować pamięć dynamicznie, zablokował dostęp do stosu, i w tym momencie pozostał zawieszony. Heap zablokowany – żaden inny wątek nie jest w stanie wydzielić pamięć dynamicznie, dopóki nie będzie ponowiony wątek, który „utrzymuje" heap.**

- **Funkcja :** VOID Sleep(DWORD dwMilliseconds); **Przeprowadzi wątek w stan snu na dwMilliseconds – OS w ciągu tego odcinku nie wydziela wątkowi kwantów czasu. Fragment kodu:**

Sleep(0);

oznacza, że wątek rezygnuje z reszty czasu kwantowego i nadaje możliwość OS podłączyć inny wątek.

- **Czas wykonania wątku–**

BOOL GetThreadTimes(HANDLE hThread, PFILETIME pftCreationTime, PFILETIMt pftExitTime, PFILETIME pftKernelTime, PFILETIME pftUserTime);

- **Os wspiera priorytety wątków (-7, 6):**

THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

- **Przykład DoesNotDoIt:**

BOOL SetThreadPriority(HANDLE *hThread*, int *nPriority*);
int GetThreadPriority(HANDLE *hThread*);

- **Wątek z większym priorytetem wywłaszcza wątki z mniejszym priorytetem: dopóki wątek z większym priorytetem nie jest w stanie oczekiwania, CPU nie będzie przydzielał czas wątkom z mniejszym priorytetem. A to oznacza, że kod, przedstawiony w przykładzie, może nigdy nie skończyć się. Wątki OS mają największy priorytet.**

Synchronizacja wątków

- **Oddziaływanie wątków odbywa się;**
  - **przy wykorzystaniu wspólnych resursów (heap, porty, pliki, okna, …)**
  - **w trakcie poinformowania innych wątków o zakończeni pewnych działań.**

- **Synchronizacja:**
  - **Sekcji krytyczne**
  - **Mutexes**
  - **Semafory**
  - **Zdarzenia**

- **Wątek synchronizuje się z innymi wątkami w taki sposób: jeśli potrzebny resurs jest zajęty, wątek przechodzi w stan snu – Os nie wydziela wątkowi czasu CPU, a wątek nie przeszkodzą działalności innych wątków . Przed tym, jak usnuć, wątek informuje OS, jakie zdarzenie powinno się odbyć, żeby odnowić jego wykonanie. Jak tylko to zdarzenie odbędzie się, wątek dostaje prawo na przydzielenie czasu CPU.**
- **Nigdy nie wolno robić coś takiego:**

```
volatile BOOL bFinishCalcul = FALSE;

void WINAPI WinMain(.....)  {

        //.....................................
        CreateThread(....., CalcFun, ....);

        //do something
        //.............

        while(!bFinishCalcul)
        {

        }

        //do something
        //.............
}


DWORD WINAPI CalcFun(LPVOID lpThreadParam)  {

        //do some calculations

        bFinishCalcul = TRUE;

        return (0);
}
```

- **Przykład DoesNotDoIt – zmienna** bFinishCalcul **powinna mieć kwalifikator volatile – nadaje to możliwość równoległym wątkom zmieniać tą zmienne. Inaczej możliwe zapętlanie, osobliwo w wersji realise.**

- **Wadę takiego podejścia:**
  - **Pierwotny wątek nigdy nie przechodzi w stan snu – przez cały czas odbiera resursy CPU**
  - **Zmienna** bFinishCalcul **może nigdy nie dostać wartości TRUE**

Sekcje krytyczne

- **To jest niewielka cząstka kodu, która potrzebuje dostępu monopolowego do jakichś danych. Nadaję możliwość zrobić tak, żeby jednocześnie tylko jeden wątek miał dostęp do wspólnego resursu, który rozdziela się pomiędzy wątkami. Jeśli jeden wątek (nazwiemy jego wątkiem #) wykonuje instrukcji kodu w obrębie sekcji krytycznej, innym wątkom dostęp do resursów, który się znajdują w tym obszarze kodu, jest wzbroniony.**

- **OS może odłączyć na jakiś czas ten wątek od CPU i podłączyć inne wątki, przecież żaden z nich, któremu jest potrzebny ten wspólny resurs, nie dostanie czasu CPU, dopóki wątek # nie wyjdzie poza granice sekcji krytycznej.**

- **Sekcji krytyczne nadają możliwość wykonywać synchronizacje w trybie użytkownika (user mode), nie używając obiektów jądra. To powoduje wysoką prędkość takiej synchronizacji, ale ograniczeniem jest to, że sekcji krytyczne są użyteczne tylko dla synchronizacji wątków w obszarze tego samego procesu.**

- **Przykład MultThreads, NotSynch**

- **Porady do użycia sekcji krytycznych:**

  - **Zastosowanie osobnej sekcji krytycznej dla każdego wspólnego resursu**
  - **Uwaga! Jeśli kod ma kilka różnych obiektów sekcji krytycznych, jest niezbędne dotrzymanie dokładnie tej samej kolejności blokowania w częściach kodu, które są wykonane przez różne wątki. Inaczej – dead lock. (Przykład Dlock)**
  - **Sekcje krytyczne trzeba jak najszybszej zwalniać. Długie zatrzymanie powoduje istotny spad wydajności kodu.**

- **Dla synchronizacji wątków, które należą do różnych procesów, używają obiekty jądra: Mutexes, Semafory, Sygnały. Obiekty te można, oczywiście, używać i do synchronizacji wątków tego samego procesu, przecież oni są wolniejsze od sekcji krytycznych.**

- **Tworzenie obiektu mutex:**

```
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes,      // SD
BOOL bInitialOwner,                           // initial owner
LPCTSTR lpName                                // object name
);
```

bInitialOwner = TRUE - wątek, który stworzył mutex, jest jego właścicielem (mutex jest zajęty).
bInitialOwner = FALSE – mutex jest wolny. Pierwszy z wątków, który czeka na mutex, może zająć jego i przedłużyć swoje wykonanie.
lpName – imie mutex'u lub NULL. Jest używane, jeśli synchronizują się wątki różnych procesów.

- **Otrzymanie HANDLE wątkiem innego procesu:**

```
HANDLE OpenMutex(
DWORD dwDesiredAccess,      // access
BOOL bInheritHandle,        // inheritance option
LPCTSTR lpName              // object name – imie mutex'u, który już jest stworzony
);
```

- **Włączenie mutex'u:**

```
DWORD WaitForSingleObject(
HANDLE hHandle,             // handle do objectu mutex
DWORD dwMilliseconds        // time-out interval in milliseconds or INFINITE
);
```

- **Zwolnienie mutex'u:**

```
BOOL ReleaseMutex( HANDLE hMutex // handle to mutex );
```

- **Przykład Mutx**