Wątki

Wątki w MFC

- Proces - “egzemplarz” wykonywania aplikacji
- Wątek (thread) – ścieżka wykonywania w aplikacji
  - Jest co najmniej jeden – primary thread
  - Można tworzyć dodatkowe
  - Klasą bazową dla wątków jest CWinThread
  - Są dwa rodzaje:
    - user interface
    - worker

Worker thread

- Służą do obsługi zadań “w tle”, tak, aby użytkownik nie musiał czekać – aby nie był blokowany interfejs
- Nie jest konieczne dziedziczenie po CWinThread
- Wątek jest konstruowany w oparciu o “controlling function”:

```
UINT MyControllingFunction( LPVOID pParam );
```

- Do rozpoczęcia wątku służy funkcja AfxBeginThread

AfxBeginThread

- CWinThread* AfxBeginThread(
  AFX_THREADPROC pfnThreadProc,
  LPVOID pParam,
  int nPriority = THREAD_PRIORITY_NORMAL,
  UINT nStackSize = 0,
  DWORD dwCreateFlags = 0,
  LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);

AfxBeginThread

- pfnThreadProc - wskaźnik do funkcji wątku
- pParam – parametr który ma być przekazany do funkcji wątku
- nPriority – priorytet. 0 dla priorytetu jak wątku tworzącego
- nStackSize – rozmiar stosu. 0 dla rozmiaru jak wątku tworzącego
- dwCreateFlags - 0 – watek rusza natychmiast po utworzeniu, CREATE_SUSPENDED – nie rusza aż do wywołania ResumeThread
- Wartość zwracana – wskaźnik do nowego wątku

```
UINT MyThreadProc( LPVOID pParam )
{
    CMyObject* pObject = (CMyObject*)pParam;

    if (pObject == NULL)
       return 1;   // not valid

    // do something with 'pObject'

    return 0;   // thread completed successfully
}


// inside a different function in the program
.
.
.
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
```

CWinThread

- m_hThread – uchwyt wątku
- m_nThreadID – ID wątku
- GetMainWnd – wskaźnik do okienka przypisanego do wątku
- GetThreadPriority, SetThreadPriority – priorytet wątku
- PostThreadMessage – przesłanie wiadomości do wątku
- ResumeThread – uruchomienie wątku
- SuspendThread – zawieszenie wątku

CWinThread

- OnIdle – obsługa “czasu wolnego”
- PreTranslateMessage, ProcessMessageFilter, Run – przetwarzanie wiadomości przez wątek
- operator HANDLE – konwersja na typ HANDLE

Kończenie wątku

- Wątek powinien sie zakończyć kulturalnie
- Możliwe są dwie sytuacje kulturalnego zakończenia:
  - Nie ma nic więcej do zrobienia
  - Otrzymano polecenie zakończenia od innego wątku
- W obu przypadkach można użyć:
  - return
  - AfxEndThread(UINT nExitCode, BOOL bDelete = TRUE);
- Wątek może po sobie posprzątać
- Aby zadziałało, wątek musi być “sprawny”

Kończenie wątku - niekulturalne

- BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
- Umożliwia zakończenie wątku z innego wątku
- Nie pozwala kończącemu się wątkowi na posprzątanie
- Używać tylko w ostateczności!

Komunikacja i synchronizacja

- Wątki muszą się komunikować aby:
  - Wymieniać dane
  - Wydawać polecenia
- W MFC wątek może pracować tylko na tych obiektach MFC, które utworzył
- Komunikować można się np. poprzez:
  - Windows messages (PostMessage, PostThreadMessage)
  - Obiekty nie-MFC

PostMessage, PostThreadMessage

- BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
- BOOL PostThreadMessage(DWORD idThread, UINT Msg, WPARAM wParam, LPARAM lParam);

Obsługiwanie wiadomości

- Wiadomości definiowane przez użytkownika mają kod od WM_USER do 0x7FFF
- Aby je obsłużyć, należy zastosować makro
  - ON_MESSAGE(message, memberFxn)
  - ON_THREAD_MESSAGE(message, memberFxn)
- Funkcje obsługujące wiadomość to odpowiednio
  - afx_msg LRESULT memberFxn(WPARAM, LPARAM)
  - afx_msg void memberFxn(WPARAM, LPARAM)

Why concurrency is a problem?

- Concurrency and asynchronous processing is typical in real world
- It can pose a problem when many entities (people, machines, processing threads) use (share) the same resource (or a limited number of resources)
- A trivial example is that of an elevator – the cabin is single resource that many people want to use. The problem is how to control the elevator to minimise waiting time?

Why concurrency is a problem?

- The elevator '”scheduling” control is not crucial, at worst improper algorithm may result in long waiting times
- There are situations where handling concurrency is crucial for correct behaviour of the system

Why concurrency is a problem?

- Unwanted results of concurrency include:
  - Race conditions
  - Resource starvation
  - Deadlocks

Race conditions

- When race conditions occur, the result of the system behaviour may be unexpected and is dependent on the sequence of other events
- Race conditions were first described in electronics (logic circuits), when parallelism is typical
- An example of race conditions may be poorly implemented ATM

ATM case

- Lets imagine the ATM operation is following
  - Authorise client
  - Get account balance
  - Get client request
  - Update account
  - Dispense cash

ATM case

- This can translate to following situation:
  - Time = t_0 Authorisation OK
  - Time = t_0+2s Balance is 1000
  - Time = t_0+10s Client requested 800, 800<1000
  - Time = t_0+12s Account updated by -800, 200 left
  - Time = t_0+14s Cash dispensed
- Everything worked fine

ATM case

- Now, lets imagine that there are two cards attached to the same account (wife and husband, corporate account etc.)
- Two persons at nearly the same time want to withdraw money. What may happen?

ATM case

	Client 1	Client 2
Time = t_0	Authorisation OK	
Time = t_0+1s		Authorisation OK
Time = t_0+2s	Balance is 1000	
Time = t_0+3s		Balance is 1000
Time = t_0+5s		Client requested 800, 800<1000
Time = t_0+7s		Account updated by -800, 200 left
Time = t_0+9s		Cash dispensed
Time = t_0+10s	Client requested 800, 800<1000	
Time = t_0+12s	Account updated by -800, **-600 left**	
Time = t_0+14s	Cash dispensed	

Race conditions

- Race conditions may be resolved by resource locking

	Client 1	Client 2
Time = t_0+2s	Balance is 1000, lock account	
Time = t_0+3s		account locked - cannot proceed
Time = t_0+10s	Client requested 800, 800<1000	
Time = t_0+12s	Account updated by -800, 200 left, unlock account	
Time = t_0+14s	Cash dispensed	

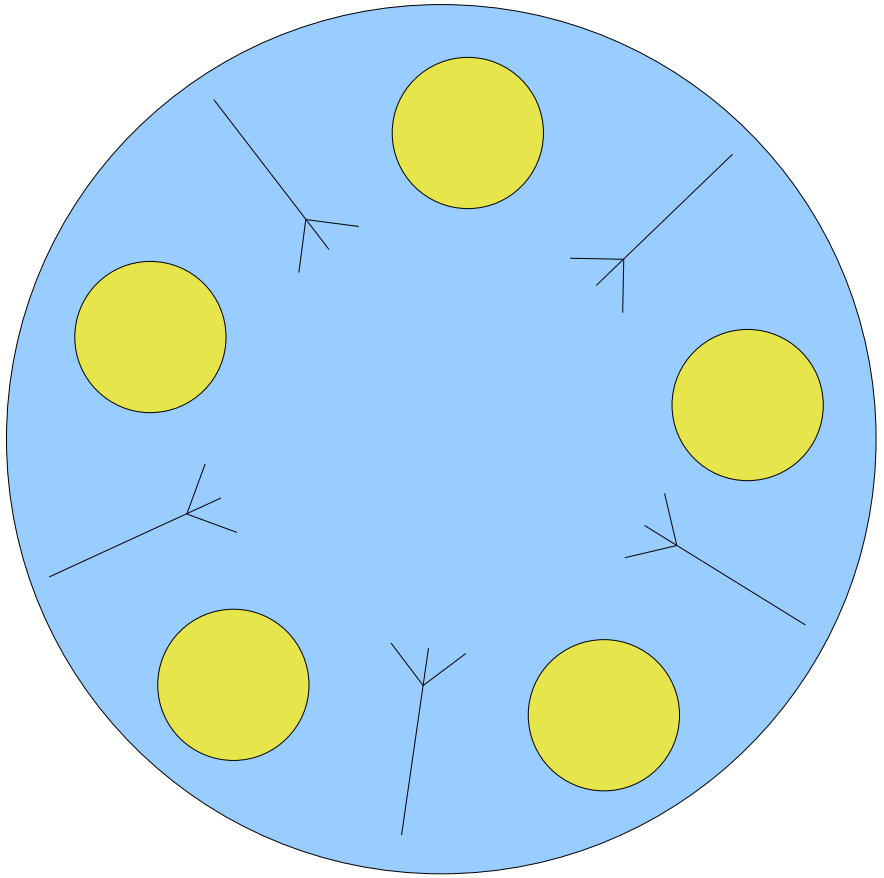
Resource starvation

- The process (person, system) is denied the resource it needs, because other process is not freeing them

Deadlock

- Deadlock occurs when a number (at least 2) processes are waiting for the other to finish (or release resources) and therefore none progresses
- This can be illustrated by dining philosophers problem (invented by Edsger Dijkstra)

Dining philosophers problem



- The philosopher is either thinking, or eating
- The philosophers do not talk (communicate)
- The philosopher can pick a fork to her/his right or left, one at a time
- The philosopher needs both forks to eat

Zasady dostępu do obiektów

- Dostęp do zmiennych o długości odpowiadającej słowu procesora jest “atomowy”
- Klasy MFC nie są “thread-safe” na poziomie obiektów (są na poziomie klas)
- Do synchronizacji (kontroli dostępu) można używać klas CSemaphore, CMutex, CCriticalSection, CEvent
- Klasy CMultiLock i CSingleLock służą do “otrymania” dostępu

Wybór klasy synchronizującej

- Czy aplikacja musi czekać na zdarzenie przed dostępem do zasobu (np. na otrzymanie danych z portu)? Jeśli tak, należy użyć CEvent
- Czy więcej niż jeden wątek aplikacji może w danej chwili korzystać z zasobu? Jeśli tak, należy użyć CSemaphore
- Czy zasobu używa więcej niż jedna aplikacja? Jeśli tak, należy użyć CMutex, jeśli nie - CCriticalSection

CCriticalSection

- Szybkie
- Dostęp tylko jednego wątku w danej chwili
- Brak wsparcia dla wielu procesów
- Użycie:
  - Zrobić CCriticalSection
  - Zrobić CSingleLock na CCriticalSection
  - Sprawdzić czy sekcja jest dostępna przez IsLocked i/lub przejąć sekcję przez Lock
  - Po skończeniu pracy z sekcją użyć Unlock lub zniszczyć (lub pozwolić aby się zniszczył) CSingleLock

CSemaphore

- Dostęp kilku wątków jednocześnie
- Brak wsparcia dla wielu procesów
- Użycie – jak CCriticalSection