Wątki w Win32API

Plan prezentacji:

1. **Czym jest proces?**
2. **Czym jest wątek?**
   - Plusy i minusy wątków
   - Tworzenie i niszczenie wątków
   - Priorytety wątków
3. **Funkcje do zarządzania wątkami w Win32API.**
4. **Mechanizmy synchronizacji wątków**
   - Zdarzenia
   - Mutexy
   - Semafory
   - Sekcja krytyczna
   - Zegary oczekujące
5. **Przykłady problemów synchronizacji wątków**
   - Problem producent-konsument
   - Problem ucztujących filozofów

Czym jest proces?

- Procesem określamy wykonywany program, w skład którego wchodzą: kod programu, licznik rozkazów, stos procesu i sekcja danych. Jest to też zadanie systemowe (np. przydział pamięci)

  Procesy dzielimy na ciężkie i lekkie.

- Proces ciężki to proces wykonujący się w odrębnej przestrzeni adresowej od procesu, który go stworzył. Koszt jego utworzenia jest na tyle duży, że jeśli tylko można, lepiej korzystać z procesów lekkich (wątków).

Czym jest wątek?

- Proces lekki (wątek) - jest to proces działający w tej samej wirtualnej przestrzeni adresowej co tworzący go ciężki proces. Jest to współbieżnie wykonujący się fragment kodu danego procesu. Wątki ze sobą współpracują. Grupa równouprawnionych wątków dzieli kod, przestrzeń adresową, otwarte pliki, zasoby systemu i należy do tego samego użytkownika. Natomiast posiada on własny licznik rozkazów, zbiór rejestrów procesora i stos.

Czym jest wątek?

- Proces po załadowaniu do systemu nie wykonuje kodu, dostarcza jedynie przestrzeni adresowej wątkom. **To wątki są jednostkami, którym system przydziela czas procesora**. Każdy proces w systemie ma niejawnie utworzony jeden wątek wykonujący kod programu. Każdy następny wątek w obrębie jednego procesu musi być utworzony explicite.

- Tworzenie wielu wątków w obrębie jednego procesu jest czasami bardzo przydatne. Wątki mogą na przykład przejmować na siebie długotrwałe obliczenia nie powodując "zamierania" całego procesu. Ponieważ wątki współdzielą zmienne globalne procesu, możliwa jest jednoczesna praca wielu wątków na jakimś zbiorze danych procesu.

Plusy i minusy wątków

Plusy:

- Dzielenie zasobów sprawia, że przełączanie między wątkami oraz tworzenie wątków jest tanie w porównaniu z procesami ciężkimi.
- Oszczędne wykorzystanie zasobów systemu – dzięki ich współużytkowaniu.
- Współpraca wielu wątków pozwala zwiększyć przepustowość i poprawić wydajność (np. jeśli jeden wątek jest zablokowany, to może działać inny).
- Lepsze wykorzystanie architektury wieloprocesorowej/wielordzeniowej (wątek → procesor/rdzeń) oraz dedykowanej technologii Hyper-Threading.

Plusy i minusy wątków

Minusy:

- Przy jednowątkowym jądrze każde odwołanie wątku poziomu użytkownika do systemu powoduje wstrzymanie całego zadania.
- Nieadekwatny przydział czasu procesora (zadanie wielowątkowe i jednowątkowe mogą dostawać tyle samo kwantów czasu).
- Brak ograniczeń dotyczących wątków może doprowadzić do wyczerpania zasobów systemu, takich jak pamięć lub czas procesora.

Tworzenie i niszczenie wątków

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES
lpThreadAttributes, SIZE_T dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
__drv_aliasesMem LPVOID lpParameter, DWORD
dwCreationFlags, LPDWORD lpThreadId );
```

```
uintptr_t _beginthread( void( __clrcall *start_address )( void * ),
unsigned stack_size, void *arglist );
```

_clrcall *start_address – adres funkcji, która zawiera kod wątku

unsigned stack_size – rozmiar stosu, możemy dać 0

void *arglist – przekazujemy argumenty do wątku

Tworzenie i niszczenie wątków

- **Tworzenie wątków:**

uintptr_t _beginthreadex(void *security, unsigned stack_size, unsigned (__clrcall *start_address)(void *), void *arglist, unsigned initflag, unsigned *thrdaddr);

***security** - wskaźnik do SECURITY_ATTRIBUTES struktury, która określa, czy zwracany uchwyt może być dziedziczony przez procesy podrzędne.

*initflag – flaga jaką otrzymuje wątek, może oznaczać czy ma ruszyć od razu po utworzeniu czy ma być zawieszony

*thraddr – 32 bitowy wskaźnik na identyfikator wątku

Tworzenie i niszczenie wątków

- Niszczenie wątków

Wewnątrz wątku:

```
void ExitThread( DWORD dwExitCode );
```

Poza wątkiem:

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode );
```

Priorytety wątków

BOOL SetThreadPriority(HANDLE hThread, int nPriority);

BOOL SetThreadPriorityBoost(HANDLE hThread, BOOL bDisablePriorityBoost);

BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);

BOOL SetProcessPriorityBoost(HANDLE hProcess, BOOL bDisablePriorityBoost);

Priorytety wątków

- **Dopuszczalne wartości priorytetów wątków:**

W każdej z klas priorytetów wyróżniamy:
bezczynny, najniższy, poniżej normy, normalny, powyżej normy, wysoki, krytyczny
Poniżej odpowiednie wartości liczbowe dla powyższych rodzajów

IDLE_PRIORITY_CLASS
1, 2, 3, 4, 5, 6, 15
BELOW_NORMAL_PRIORITY_CLASS
1, 4, 5, 6, 7, 8, 15
NORMAL_PRIORITY_CLASS
1, 6, 7, 8, 9, 10, 15
ABOVE_NORMAL_PRIORITY_CLASS
1, 8, 9, 10, 11, 12, 15
HIGH_PRIORITY_CLASS
1, 11, 12, 13, 14, 15, 15
REALTIME_PRIORITY_CLASS
16, 22, 23, 24, 25, 26, 31

Funkcje do zarządzania wątkami w Win32API

- **ThreadProc** – funkcja aplikacji, określająca początek wątku,
- **OpenThread** – otwiera istniejący obiekt wątku ,
- **ResumeThread** – zmniejsza liczbę zawieszonych wątków,
- **SuspendThread** – zawiesza wątek,
- **SetThreadAffinityMask** – wybiera procesor na jakim ma wykonać się wątek ,
- **WaitForSingleObject** – czeka aż dany obiekt znajdzie się w stanie sygnalizowanym lub upłynie limit czasu,
- **WaitForMultipleObject** – czeka aż jeden lub wszystkie obiekty znajdą się w stanie sygnalizowanym lub upłynie limit czasu

Mechanizmy synchronizacji wątków

- **Zdarzenia:**

Zdarzenie tworzymy za pomocą funkcji CreateEvent(). Każdy oczekujący wątek widzi zdarzenie w dwóch stanach. Jest zgłoszone lub odwołane. Za pomocą funkcji SetEvent() informujemy system o zaistnieniu zdarzenia. Od tej pory zdarzenie jest zgłoszone i wszystkie wątki oczekujące na jego zgłoszenie (za pomocą funkcji WaitForSingleObject()) mogą wznowić działanie. Funkcja ResetEvent() odwołuje zdarzenie.

Mechanizmy synchronizacji wątków

- **Zdarzenia:**

Funkcja PulseEvent() powoduje zgłoszenie zdarzenia, po czym natychmiastowe jego odwołanie. Jeśli zdarzenie odwoływane jest ręcznie, to funkcja PulseEvent() przepuszcza wszystkie wątki oczekujące w danej chwili na zdarzenie, po czym odwołuje zdarzenie, jeśli zaś zdarzenie odwoływane jest automatycznie, to funkcja PulseEvent() przepuszcza tylko jeden wątek z puli oczekujących w danej chwili wątków, po czym odwołuje zdarzenie.

Mechanizmy synchronizacji wątków

- **Mutexy** (mutually exclusive - wzajemnie wykluczający się).

Mutex jest obiektem służącym do synchronizacji. Jego stan jest ustawiony jako "sygnalizowany", kiedy żaden wątek nie sprawuje nad nim kontroli oraz "niesygnalizowany" kiedy jakiś wątek sprawuje nad nim kontrolę. Synchronizacja za pomocą mutexów realizuje się tak, że każdy wątek czeka na objęcie mutexa w posiadanie, zaś po zakończeniu operacji wymagającej wyłączności, wątek uwalnia mutexa. W celu stworzenia mutexa, wątek woła funkcję CreateMutex().

Mechanizmy synchronizacji wątków

- **Mutexy**

W chwili tworzenia wątek może zażądać natychmiastowego prawa własności do mutexa. Inne wątki (nawet innych procesów) uzyskują uchwyt mutexa za pomocą funkcji OpenMutex().

Następnie czekają na objęcie mutexa w posiadanie (za pomocą WaitForSingleObject()). Do uwalniania mutexów służy funkcja ReleaseMutex().

Jeśli wątek kończy się bez uwalniania mutexów, które posiadał, takie mutexy uważa się za porzucone. Każdy czekający wątek może objąć takie mutexy w posiadanie

Mechanizmy synchronizacji wątków

- **Semafory:**

Semafory mogą być wykorzystywane tam, gdzie zasób dzielony jest na ograniczoną ilość użytkowników. Semafor działa jak furtka kontrolująca ilość wątków wykonujących jakiś fragment kodu. Za pomocą semaforów aplikacja może kontrolować na przykład maksymalną ilość otwartych plików, czy utworzonych okien. Semafory są w działaniu bardzo podobne do mutexów.

Mechanizmy synchronizacji wątków

- **Semafory:**

Nowy semafor tworzony jest w funkcji CreateSemaphore().

Wątek tworzący semafor specyfikuje wartość wstępną i maksymalną licznika. Inne wątki uzyskują dostęp do semafora za pomocą funkcji OpenSemaphore() i czekają na wejście za pomocą funkcji WaitForSingleObject(). Po zakończeniu pracy w sekcji krytycznej wątek uwalnia semafor za pomocą funkcji ReleaseSemaphore().

Wątki nie wchodzą w posiadanie semaforów!

Mechanizmy synchronizacji wątków

- **Zegary oczekujące:**

Zegar oczekujący przechodzi w stan sygnalizowany po upływie zadanego okresu czasu lub w określonych odstępach czasu z automatycznym powrotem do stanu niesygnalizowanego. Zegary umożliwiają np. regularne wywoływanie określonych wątków.

Tworzymy zegar oczekujący za pomocą funkcji CreateWaitableTimer(). Ustawiamy go za pomocą funkcji SetWaitableTimer().

Mechanizmy synchronizacji wątków

- **Sekcje krytyczne:**

Część programu, w której proces korzysta ze współdzielonej pamięci, nazywa się regionem krytycznym lub sekcją krytyczną. Mechanizm sekcji krytycznej możliwy jest do wykorzystania tylko w obrębie jednego procesu (do synchronizacji wątków), jednak jest to metoda najszybsza i najwydajniejsza.

Mechanizmy synchronizacji wątków

- **Sekcje krytyczne:**

Warunki:

- Żadne procesy nie mogą jednocześnie przebywać wewnątrz swoich sekcji krytycznych.
- Nie można przyjmować żadnych założeń dotyczących szybkości lub liczby procesorów.
- Proces działający wewnątrz swojego regionu krytycznego nie może blokować innych procesów.
- Żaden proces nie powinien oczekiwać w nieskończoność na dostęp do swojego regionu krytycznego.

Mechanizmy synchronizacji wątków

- **Sekcje krytyczne:**

Interfejs programowania **Win32API** udostępnia typ danych CRITICAL SECTION, który wraz z odpowiednim zestawem funkcji może być wykorzystany do implementacji sekcji krytycznej.

CRITICAL_SECTION cs; - dzielona na wszystkie wątki

InitializeCriticalSection(&cs)
DeleteCriticalSection(&cs)
EnterCriticalSection(&cs)
LeaveCriticalSection(&cs)

Przykład problemów synchronizacji wątków

- Problem producent-konsument:

to klasyczny informatyczny problem synchronizacji. W problemie występują dwa rodzaje procesów: producent i konsument, którzy dzielą wspólny zasób - bufor dla produkowanych (konsumowanych) jednostek. Zadaniem producenta jest wytworzenie produktu, umieszczenie go w buforze i rozpoczęcie pracy od nowa. W tym samym czasie konsument ma pobrać produkt z bufora. Problemem jest taka synchronizacja procesów, żeby producent nie dodawał nowych jednostek gdy bufor jest pełny, a konsument nie pobierał gdy bufor jest pusty.

Przykład problemów synchronizacji wątków

- Problem producent-konsument:

Rozwiązaniem dla producenta jest uśpienie procesu w momencie gdy bufor jest pełny. Pierwszy konsument, który pobierze element z bufora budzi proces producenta, który uzupełnia bufor. W analogiczny sposób usypiany jest konsument próbujący pobrać z pustego bufora. Pierwszy producent, po dodaniu nowego produktu umożliwi dalsze działanie konsumentowi. Rozwiązanie wykorzystuje komunikację międzyprocesową z użyciem semaforów. Nieprawidłowe rozwiązanie może skutkować zakleszczeniem.

Przykład problemów synchronizacji wątków

- **Problem ucztujących filozofów**:

Pięciu filozofów siedzi przy stole i każdy wykonuje jedną z dwóch czynności – albo je, albo rozmyśla. Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każda osoba ma przy sobie dwie sztuki – po swojej lewej i prawej stronie. Ponieważ jedzenie potrawy jest trudne przy użyciu jednego widelca, zakłada się, że każdy filozof korzysta z dwóch. Dodatkowo nie ma możliwości skorzystania z widelca, który nie znajduje się bezpośrednio przed daną osobą. Problem ucztujących filozofów jest czasami przedstawiany przy użyciu ryżu, który musi być jedzony dwiema pałeczkami, co lepiej obrazuje sytuację. Filozofowie nigdy nie rozmawiają ze sobą, co stwarza zagrożenie zakleszczenia w sytuacji, gdy każdy z nich zabierze lewy widelec i będzie czekał na prawy (lub na odwrót).

Przykład problemów synchronizacji wątków

- **Problem ucztujących filozofów:**

Relatywnie prostym rozwiązaniem jest wprowadzenie kelnera. Filozofowie będą pytać go o pozwolenie przed wzięciem widelca. Ponieważ kelner jest świadomy, które widelce są aktualnie w użyciu, może nimi rozporządzać zapobiegając zakleszczeniom.

Inne proste rozwiązanie jest osiągalne poprzez częściowe uporządkowanie lub ustalenie hierarchii dla zasobów (w tym wypadku widelców) i wprowadzenie zasady, że kolejność dostępu do zasobów jest ustalona przez ów porządek, a ich zwalnianie następuje w odwrotnej kolejności, oraz że dwa zasoby niepowiązane relacją porządku nie mogą zostać użyte przez jedną jednostkę w tym samym czasie.

Przygotował

Kacper Lamparski