Karadeniz Teknik Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü
Bilgisayar Sistemleri Laboratuarı

WINDOWS'TA İŞLEM TABANLI ÇOKGÖREVLİLİK VE SENKRONİZASYON

Bu deneyde ana hatlarıyla multiprocess-multithread kavramlarına ve thread'ler arası senkronizasyona değinilecektir. Deney sonunda öğrencilerin bu tür programları sorunsuz olarak geliştirebilmesi amaçlanmaktadır. Deneyde geliştirme aracı olarak MS Visual Studio 6.0 kullanılacaktır.

Deneye Hazırlık

1. Deneyde kodlama Windows API fonksiyonları yardımıyla gerçekleştirilecek olduğundan HANDLE, DWORD gibi değişken tiplerinin ne anlama geldiğini öğreniniz.

2. Hyperthreading teknolojisi hakkında araştırma yapınız.

3. Deney sorularının cevaplarını araştırınız.

4. Deney föyünü dikkatlice okuyunuz. Gerekli teoriksel altyapı için gerekirse İşletim Sistemleri ders notlarına başvurunuz.

1. MULTIPROCESSED ve MULTITHREADED PROGRAMLAR

Bütün modern işletim sistemleri aynı anda birden çok işlem yapmaktadırlar. Bir program koşarken diskten veri okunabilir, ya da yazıcıdan çıktı alınabilir. Bu tür sistemlerde, birden çok işlemci (veya çekirdek) olduğunda işler bu işlemciler arasında dağıtılmakta, az sayıda işlemci olduğunda ise anahtarlama ile yalancı paralelleştirme sağlanmaktadır. Bu modelin ana yapısı olan process'ler program counter'ın, kaydedicilerin ve değişkenlerin değerini de içeren koşan bir programdır.

Thread (lightweight process olarak da adlandırılır) ise, bir process'in birbirinden ayrı çalışabilen parçalarına denir. Bir thread koştuğu zaman, programdaki bir fonksiyonu icra eder. Koşan programı ifade eden bir process, programın "main" fonksiyonunu icra eden ana thread olarak adlandırılan bir thread ile başlar. Multithreaded programlamada ana thread, başka fonksiyonları icra eden thread'leri oluşturabilir (bu thread'ler de başkalarını).

Programcı açısından process ve thread'ler nelere sahiptir ve neleri paylaşırlar?

Thread'lerle çalışmanın process'lerle çalışmaya göre avantaj ve dezavantajları nelerdir?

Multitasking ile multithreading arasındaki fark nedir?

1.1 Process Oluşturma ve Sonlandırma

Process'ler çalıştırılacak olan exe dosyasının ismini parametre olarak alan ve ilk threadi oluşturan CreateProcess WinAPI fonksiyonu ile oluşturulurlar. Bu fonksiyon ile istenildiğinde programlar komut satırı argümanları ile çalıştırılabilir, ya da parametre aktarımı gerçekleştirilebilir.

```
STARTUPINFO startin;
PROCESS_INFORMATION pinfo;
startin.cb= sizeof(STARTUPINFO);
memset(&startin, 0, sizeof(startin));
CreateProcess(NULL,  "notepad.exe", NULL, NULL, FALSE, 0, NULL, NULL, &startin, &pinfo);
```

Windows'ta parent-child process kavramı var mıdır? Açıklayınız.

Oluşturulan bir process kendi kodu içerisinden ExitProcess, başka bir process tarafından ise TerminateProcess API fonksiyonları ile silinir. Bir process'in başka bir process tarafından sonlandırılması ancak hiçbir çare kalmadığında uygulanabilecek son yöntem olmalıdır.

```
ExitProcess(GetLastError()); //kendi process'ini sonlandırır
BOOL isTerminated=TerminateProcess(processHandle, 0); //handle'ı ile belirtilen process'i sonlandırır
```

CloseHandle() API fonksiyonu ile process sonlandırılabilir mi?

1.2 Thread Oluşturma ve Sonlandırma

Bir thread CreateThread API fonksiyonuyla oluşturulup çalıştırılmaya başlanır.

```
DWORD ThrFunc(LPVOID param); //Thread olarak kullanılacak fonksiyonun prototipi
DWORD ThrID; //Global olarak tanımlanacak olan thread'in ID'si
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThrFunc, (LPVOID) hwnd, 0, &ThrID); //Thread'i
        fonksiyon ismi, parametresi ve thrID değişkenleri ile oluşturur
```

Thread, thread fonksiyonunun icrasının bittiğinde sonlanabileceği gibi şu şekillerde de sonlandırılabilir:

a. Thread akışı içerisinde doğrudan ExitThread API fonksiyonunun çağırılmasıyla (prototipi ExitProcess ile benzerdir). ExitThread başka bir thread tarafından değil, thread içinden çağırılmalıdır.

b. Başka bir thread'in TerminateThread fonksiyonunu çağırması sonucunda (prototipi TerminateProcess ile benzerdir).

c. Process'in ana thread'i sonlandığında. GUI uygulamalarında WinMain, console uygulamalarında main ana thread akışını temsil eder.

1.3 Thread'lerin Durdurulması ve Çalıştırılması

Programcı isterse handle değerini bildiği bir thread'i SuspendThread API fonksiyonuyla bloke edebilir (tarifeleme dışı bırakılır). Tekrar tarifelemeye alınabilmesi için ResumeThread API fonksiyonu kullanılmalıdır.

```
DWORD prevSuspendCounter=SuspendThread(threadHandle); //thread'i durdurur
DWORD prevSuspendCounter=ResumeThread(threadHandle); //thread'in icrasını sürdürür
```

Her thread'in bir durdurma sayacı vardır. Thread oluşturulduğunda bu sayaç birdir. Bu sayaç sıfırdan farklı olduğunda thread tarifeleme dışı bırakılır. Aslında ResumeThread fonksiyonu yalnızca durdurma sayacını bir eksiltir. SuspendThread fonksiyonu ise sayacı bir arttırır. Sayaç sıfır dışı ise thread'i tarifeleme dışı bırakır.

Bir thread'in durumunu (çalışıyor ya da durdurulmuş) kod parçası ile nasıl belirlersiniz?

Bir thread, BLOCKED, RUNNIG, RUNNABLE durumların üçünden birinde bulunabilir. Aşağıda verilen durumlar arası geçişlerinin mümkün olup olmadığını açıklayınız.

1. *BLOCKED durumundan RUNNING durumuna geçiş.*
2. *RUNNING durumundan BLOCKED durumuna geçiş.*
3. *RUNNABLE durumundan BLOCKED durumuna geçiş.*

Bir thread uyku modundayken işlemci zamanı tüketir mi? Açıklayınız.

1.4 Thread'lerin Öncelik Derecelendirilmesi

Win32'de öncelikli döngüsel (priority round robin) çizelgeleme denilen bir yöntem kullanılır. Bu yöntemde her thread'in 0-31 arasında bir öncelik derecesi vardır. En yüksek öncelikli (en yüksek değere sahip) thread grubu diğerlerine bakılmaksızın kendi aralarında çalıştırılır. O grup tamamen bitirilince daha düşük gruptakiler yine kendi aralarında çizelgelenir ve işlemler böyle devam ettirilir.

Düşük öncelikli thread'ler hangi durumlarda çalışma imkânı bulabilir?

Bir thread'in önceliği process'in öncelik sınıfının değeriyle thread'in göreceli önceliğinin toplanması biçiminde elde edilir. Process ve thread'lerin öncelik sınıfı şunlardır:

		Win32 process sınıfı öncelikleri					
Win32 Thread Öncelikleri		REALTIME_PRIORITY_CLASS	HIGH-PRIORITY_CLASS	ABOVE_NORMAL_PRIORITY_CLASS	NORMAL_PRIORITY_CLASS	BELOW_NORMAL_PRIORITY_CLASS	IDLE_PRIORITY_CLASS
	THREAD_PRIORITY_TIME_CRITICAL	31	15	15	15	15	15
	THREAD_PRIORITY_HIGHEST	26	15	12	10	9	6
	THREAD_PRIORITY_ABOVE_NORMAL	25	14	11	9	7	5
	THREAD_PRIORITY_NORMAL	24	13	10	8	6	4
	THREAD_PRIORITY_BELOW_NORMAL	23	12	9	7	5	3
	THREAD_PRIORITY_LOWEST	22	11	8	6	4	2
	THREAD_PRIORITY_IDLE	16	1	1	1	1	1

Process'in öncelik sınıfı 2 biçimde belirlenebilir:

a. CreateProcess fonksiyonunun 6. parametresinde yukarıdaki sembolik sabitler kullanılarak.

b. Process'e ilişkin herhangi bir thread içerisinde SetPriorityClass API fonksiyonuyla. Ayrıca GetPriorityClass API fonksiyonuyla da process'in öncelik sınıfının ne olduğu elde edilebilir.

```
BOOL isSet=SetPriorityClass(procHandle, HIGH-PRIORITY_CLASS); //process'in önceliğini değiştirir
```

Bir thread'in göreceli önceliği de iki biçimde belirlenebilir:

a. Thread oluştururken (CreateThread ile) 5. parametrede yukardaki sabitlerden biri girilerek,

b. SetThreadPriority API fonksiyonuyla. Prototipi:

```
BOOL isSet=SetThreadPriority(threadHandle, THREAD_PRIORITY_IDLE); //thread'in önceliğini değiştirir
```

Tabii thread'in göreceli önceliği process'te olduğu gibi GetThreadPriority fonksiyonuyla elde edilir.

2. WINDOWS SENKRONİZASYON NESNELERİ

Multithreaded olarak geliştirilen uygulamalarda thread'ler birçok zaman birbirinden bağımsız olarak çalışmamaktadır. Örneğin, aynı kaynağı birden çok thread'in sorunsuz biçimde kullanması gerekebilir. Ya da fonksiyon tam olarak paralelleştirilemediğinden seri kısımlar mevcut olup bir thread diğer bir thread'den sonuç üretmesini bekleyebilir. Bu tür durumlarda kullanılan en yaygın ve sorunsuz çözüm işletim sisteminin sağladığı, tarifeleyicinin bölmeyeceği bir süreç oluşturan senkronizasyon nesnelerinin kullanımıdır.

Neden processlerin haberleşmesinde mesaj geçme yöntemi yerine paylaşımlı bellek kullanımı tercih edilir?

Race condition ve deadlock terimleri neyi ifade etmektedir?

Mutual exclusion, deadlock oluşumuna sebep olur mu? Açıklayınız.

Windows dört çeşit senkronizasyon nesnesini (klasik semafor, mutex semafor, olay nesnesi, kritik bölge nesnesi) destekler. Bununla birlikte tümü, şu veya bu şekilde, semafor kavramına dayalıdır. Senkronizasyon nesnelerini incelemeden önce senkronizasyon nesnelerinde yardımcı fonksiyon olarak kullanılan WaitForSingleObject ve WaitForMultipleObjects fonksiyonlarını inceleyelim.

2.1 WaitForSingleObject ve WaitForMultipleObjects Fonksiyonları

Bu fonksiyonlar bir olay gerçekleşene kadar bir thread'in ya da process'in bloke edilmesini sağlarlar (bir tür bariyer noktası oluştururlar). Bu olaylar geniş bir yelpazeyi (event, process, semaphore, thread gibi) içine alır.

WaitForSingleObjects fonksiyonu parametre olarak girilen senkronizasyon nesnesi olumlu duruma gelene kadar, ya da belirtilen süre dolana kadar bloklama sağlar. WaitForMultipleObjects ise bu fonksiyonun birden çok olay için genişletilmiş biçimidir. Bu fonksiyonlar diğer nesnelerle beraber kullanıldığında burada örnek koda yer verilmeyecektir.

WaitForMultipleObjects ile yapılan bekleme sonlandığında, sonlanma nedeni nasıl öğrenilebilir?

2.2 Semafor Nesnesi

Semafor, ortak bir kaynağa belirli sayıda süreç ya da thread'in erişebilmesine izin vermek için kullanılır. Semaforlar, değerleri, erişim izni elde edildiğinde azaltılıp, erişildikten sonra artırılan sayaçlar kullanılarak gerçekleştirilir. Bu kontrol ve arttırma işlemi işletim sistemi tarafından kesintisiz bir süreç olarak sunulur. Semafor nesnesi CreateSemaphore fonksiyonu ile oluşturulur. Sayacı ise ReleaseSemephore fonksiyonu ile arttırılır:

```
HANDLE g_Semaphore; //semafor nesnesinin handle'ı global olarak tanımlanmalı
g_Semaphore=CreateSemaphore(NULL, 4, 4, "NesneAdi"); // Semafor nesnesi başlatımlama anında
        başlangıç ve maksimum değerleri girilerek oluşturulmalı

DWORD dwWaitResult=WaitForSingleObject(g_Semaphore, INFINITE);
// Aynı anda erişimin sınırlandırılacağı kritik kod bu kısma yazılmalı
ReleaseSemaphore(g_Semaphore, 1, NULL); //sayacı 1 arttırır

CloseHandle(g_Semaphore); //semaforun kullanımı bittikten sonra silinmeli
```

2.3 Mutex Semafor Nesnesi

Mutex semafor kaynağa aynı anda sadece tek bir erişime izin verir; bu da standart semaforun özel bir şekli olarak görülebilir. Mutex nesnesinin bir özyineleme sayacı vardır. Bu sayaç, her WaitForSingleObject fonksiyonunun çağrımında 1 arttırılır, her ReleaseMutex fonksiyonunda ise 1 azaltılır. Kodlamada, erişim hakkını sağlayabilmek için bu sayaç değerinin 0 olmasına dikkat edilmelidir.

Mutex nesnesini oluşturan CreateMutex ve sayacı azaltan ReleaseMutex fonksiyonlarıdır.

```
HANDLE g_Mutex;
g_Mutex=CreateMutex(NULL, TRUE, "NesneAdi"); //Önce mutex nesnesi oluşturulmalı

DWORD waitResult=WaitForSingleObject(g_Mutex, INFINITE);
// Aynı anda erişimin yasaklanacağı kritik kod bu kısma yazılmalı
ReleaseMutex(g_Mutex);

CloseHandle(g_Mutex);
```

2.4 Olay Nesnesi (Event Object)

Bu nesne, bir başka thread ya da süreç kullanılabileceğini söyleyinceye kadar o kaynağa erişimi bloke eder; olay nesneleri, özel bir olayın gerçekleştiğini (kodda istenen yere gelindiğini) bildirirler.

Olay senkronizasyon nesnesi CreateEvent fonksiyonu ile oluşturulur, setEvent ile setlenir.

```
HANDLE g_Event[2]; //Olmasını istediğimiz iki olay için global değişken dizisi
g_Event[0]=CreateEvent(NULL, FALSE, FALSE, "NesneAdi1");
g_Event[1]=CreateEvent(NULL, FALSE, FALSE, "NesneAdi2"); //İlk olarak olay nesneleri, manuel resetleme ve başlangıç durumu parametreleriyle oluşturulmalı

WaitForMultipleObjects(2, g_Event, WAIT_ALL, INFINITE); //Olaylar gerçekleşene kadar beklemek için

SetEvent(g_Event[0]); //Olayın gerçekleştiğini belirtir
ResetEvent(g_Event[0]); //Olayın gerçekleşmesini tekrar bekleyebilmek için (manuel reset seçilmişse)
```

Olay nesnesi process'ler arası senkronizasyonda nasıl kullanılır?

2.5 Kritik Bölge Nesnesi

Bir kodun aynı anda diğer süreçler tarafından da kullanılması, bir kritik bölge nesnesi ile, o bölüm kritik bölge yapılarak önlenebilir.

InitializeCriticalSection fonksiyonu işletim sistemi düzeyinde senkronizasyonu oluşturabilmek için çeşitli flag değişkenlerini oluşturur ve çeşitli ilk işlemleri yapar. EnterCriticalSection fonksiyonu ise kritik koda giriş hakkını elde etmeye çalışır. Aynı yapı değişkeninin adresiyle bu fonksiyonu geçmiş olan bir thread varsa bu thread bloke olur, yoksa giriş hakkı elde edilir. LeaveCriticalSection API fonksiyonu da kritik koda giriş hakkı elde etmiş fonksiyonun bu hakkı geri bırakması için kullanılır. Fonksiyonun çağırılmasıyla, bu nedenle bloke olmuş bir thread tekrar tarifelemeye dahil edilecektir.

```
CRITICAL_SECTION m_cs;
InitializeCriticalSection(&m_cs); //başlatımlama işlemleri için

EnterCriticalSection(&m_cs);
// Aynı anda erişimin yasaklanacağı kritik kod bu kısma yazılmalı
LeaveCriticalSection(&m_cs);

DeleteCriticalSectionResetEvent(g_Event); //ilk işlemlerin geri alınması için
```

Kritik bölge nesneleri sadece process içindeki thread'lerin senkronizasyonunda kullanılırken diğer senkronizasyon nesneleri, süreçler arasındaki senkronizasyonda da kullanılabilirler.

3. DENEY TASARIMI VE UYGULANMASI

a. Bir thread tarafından oluşturulan dizi elemanlarının, ana threaddeki ekran çıktısı *0,0,0,5,5,5,10,10,10,15,15,15,...* şeklinde olmaktadır. Bunun için gerekli olan Thread() ve main() fonksiyonlarını yazınız. Senkronizasyon için olay nesnesini kullanınız.

b. 1. soruda yazılması istenen fonksiyonları kritik bölge nesnesi kullanarak gerçekleştiriniz.

c. Aşağıda verilen Yaz() thread'inde üretilen bir veriyi okuyacak olan Oku() threadini ve bu threadlerin çalıştırılmasını sağlayan ana threadi yazınız.

```
void Yaz()
{
   …
     while (true)
      {
      if (WaitForSingleObject(hMutex, INFINITE) == WAIT_FAILED)
            ExitThread(0);

      if (state == 0)
       {
         ReleaseMutex(hMutex);
         WaitForSingleObject(hOkuma, INFINITE);
         continue;
        }
       break;
      }
   …
    prinft("Yazıldı\n");
    state= 0;

       ReleaseMutex(hMutex);
       PulseEvent(hYazma);
  …
}
```

d. Yazdığınız uygulamalarda herhangi bir process ve threadin öncelik sınıfını belirlemeye çalışınız.

4. DENEY RAPORU

a. Deney sorularını kısaca açıklayınız.

b. Bir multithread uygulama örneği vererek, uygulamada tek thread ya da olduğundan daha fazla thread kullanıldığında uygulamanın işlem hızını değerlendiriniz. Multithread olarak programlanması güç olan problemler neler olabilir, kısaca açıklayınız.

c. **synchronized** metodunu kullanmak, deadlock durumunun gerçekleşemeyeceğini garanti eder mi, açıklayınız.

d. Deneyin uygulama aşamasındaki problemlerin çözümlerini belirtiniz.

Deneyde Kullanılan Fonksiyonların Prototipleri

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation );
```

Fonksiyonun ilk parametresi çalıştırılacak exe dosyanın ismidir. İkinci parametre main ya da WinMain fonksiyonlarına geçirilecek komut satırı argümanlarını belirtir. Komut satırı argümanı boşluklarla ayrılmış olarak girilebilir, ancak ilk boşluksuz string'in çalıştırılacak programı belirtmesi gerekir. Fonksiyonun ilk parametresi NULL ile geçilirse ikincisi; ikincisi NULL ile geçilirse ilki kesinlikle girilmiş olması gerekir. Fonksiyonun üçüncü ve dördüncü parametreleri güvenlik bilgilerini, beşinci parametresi oluşturulan process'in oluşturan process'in handle bilgilerini kullanıp kullanamayacağını, altıncı parametresi yeni process'in oluşturulma özellikleriyle ilgili bayrakları, yedinci parametresi kullanılacak çevre değişkenlerini belirten yazının adresini, sekizinci parametresi programın default dizinini belirtir. Default değerlerin alınması için parametreler NULL olarak girilmelidir. Fonksiyonun dokuzuncu parametresi STARTUPINFO türünden bir yapının adresini alır. Bu yapının içi CreateProcess tarafından doğru bilgilerle doldurulmaktadır. Fonsiyonun onuncu parametresi PROCESS_INFORMATION isimli bir yapının adresini alır. Bu yapının iki önemli elemanı oluşturulan process'e ilişkin ID ve handle değeridir.

```
-------------------------------------
VOID ExitProcess(UINT exitCode);
-------------------------------------
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Burada hProcess process'in handle değeri, uExitCode ise exit code'udur. TerminateProcess fonksiyonu process nesnesinin silinmesini garanti etmez. Bir process'in bu biçimde sonlandırılması ancak hiçbir çare kalmadığında uygulanabilecek son yöntem olmalıdır.

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

Fonksiyonun birinci parametresi güvenlik bilgileriyle ilgilidir. Bu parametre NULL biçiminde geçilebilir. Fonksiyonun ikinci parametresi thread stack alanının uzunluğudur. Fonksiyonun üçüncü parametresi thread akışının başlangıcını belirten fonksiyon işaretçisidir. Bu parametreye thread fonksiyonunun adresi geçirilir. Fonksiyonun dördüncü parametresi thread fonksiyonuna geçirilecek olan parametredir. Fonksiyonun beşinci parametresi thread'in oluşturulma anındaki durumunu belirtir. 0 girilirse thread fonksiyonu thread oluşturulur oluşturulmaz çalıştırılır, CREATE_SUSPENDED olarak girilirse threadi çalıştırmak için ResumeThread fonksiyonu uygulanmalıdır. Fonksiyonun son parametresi thread'in ID değerini almakta kullanılır.

```
-------------------------------------
void ExitThread(DWORD dwExitCode);
-------------------------------------
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
-------------------------------------
DWORD SuspendThread( HANDLE hThread );
```

```
-------------------------------------
DWORD ResumeThread( HANDLE hThread );
-------------------------------------
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
-------------------------------------
DWORD GetPriorityClass(HANDLE hProcess);
-------------------------------------
BOOL SetThreadPriority(HANDLE hThread, int nPriority);
-------------------------------------
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMiliseconds);
```

Fonksiyonun birinci parametresi beklenecek senkronizasyon nesnesinin handle değeridir. İkinci parametre ms cinsinden bloke işleminin en fazla ne kadar süreceğini belirtir. Eğer INFINITE girilirse birinci parametrede belirtilen senkronizasyon nesnesi olumlu duruma gelene kadar bekleme yapılır.

```
-------------------------------------
DWORD WaitForMultipleObjects(
  DWORD dwCount,
  CONST HANDLE *pbHandles,
  BOOL fWaitAll,
  DWORD dwMiliseconds);
```

Fonksiyonun ilk parametresi kaç tane senkronizasyon nesnesi için kontrol yapılacağını belirtir. İkinci parametre senkronizasyon nesnelerinin handle'larının bulunduğu dizinin başlangıç adresidir. Üçüncü parametre TRUE ise bütün senkronizasyon nesneleri açık duruma gelene kadar thread bloke edilir. Bu parametre FALSE ise senkronizasyon nesnelerinin herhangi birisi açık duruma geldiğinde thread blokeden kurtulur. Dördüncü parametresi ise maksimum beklenecek zaman aralığıdır.

```
-------------------------------------
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName);
```

Fonksiyonun birinci parametresi güvenlik bilgilerine ilişkindir, NULL geçilebilir. İkinci parametresi semafor sayacının başlangıç adresidir. Üçüncü parametre semafor sayacının maximum değeridir. Yani semafor sayacı ancak burada belirtilen değere erişebilir. Fonksiyonun son parametresi semafor nesnesinin ismidir. Fonksiyon başarılıysa semafor nesnesinin handle değerine başarısızsa NULL değerine geri döner. Bu semafor ismi başka bir process tarafından kullanılmışsa fonksiyon o process'le oluşturulan semafor'un handle değerine geri döner.

```
-------------------------------------
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  PLONG lpPreviousCount);
```

Fonksiyonun ilk parametresi semafor nesnesinin handle değeridir. İkinci parametre semafor sayacının kaç arttırılacağını anlatır (genelde 1). Üçüncü parametre arttırımdan önceki sayaç değerini vermektedir ve NULL olarak alınabilir. Fonksiyonun geri dönüş değeri başarısı hakkında bilgi verir.

```
-------------------------------------
HANDLE CreateMutex(
  PSECURITY_ATTRIBUTES lpSecurityAttributes,
  BOOL bInitilialOwner,
  PCTSTR pszName);
```

Fonksiyonun birinci parametresi güvenlik bilgilerine ilişkindir, NULL geçirilebilir. İkinci parametre TRUE girilirse mutex nesnesi başlangıçta bu fonksiyonu çağıran thread'e giriş hakkını vermiştir (sayaç 1 olmuştur). Fonksiyonun üçüncü parametresi process'ler arasındaki senkronizasyon için gereken isimdir.

```
-------------------------------------
BOOL ReleaseMutex( HANDLE hMutex);
```

Bu fonksiyon mutex nesnesinin erişim hakkını geri bırakır (sayacı 1 azaltır). Fonksiyonun parametresi mutex nesnesinin handle değeridir.

```
-------------------------------------
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCSTR lpName);
```

Fonksiyonun birinci parametresi güvenlik bilgilerini içerir ve NULL girilebilir. İkinci parametresi resetleme işleminin manual mi yoksa otomatik mi yapılacağını belirtir. TRUE ise manual, FALSE ise otomatiktir. Üçüncü parametre olay nesnesinin başlangıçtaki durumunu belirtir. Son parametre programcının vereceği herhangi bir isimdir. Olay nesnesi farklı process'lerin thread'lerini senkronize etmek amacıyla da kullanılabilir.

```
-------------------------------------
BOOL SetEvent(HANDLE hEvent);
```

Olay nesnesinin durumunu açık yapan bu fonksiyonda hEvent, nesnenin handle'ını belirtmektedir.

```
-------------------------------------
void InitializeCriticalSection( LPCRITICAL_SECTION lpCriticalSection);
-------------------------------------
void EnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection);
```

Fonksiyon CRITICAL_SECTION isimli bir yapı değişkeninin adresini parametre olarak alır.

```
-------------------------------------
void LeaveCriticalSection( LPCRITICAL_SECTION lpCriticalSection);
-------------------------------------
```