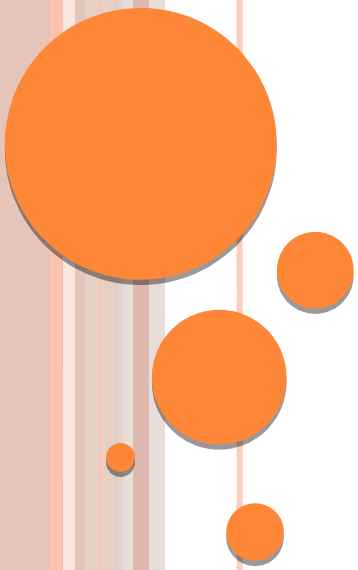


İŞLETİM SİSTEMLERİ

DERS 5 THREAD (İŞ PARÇACIKLARI)

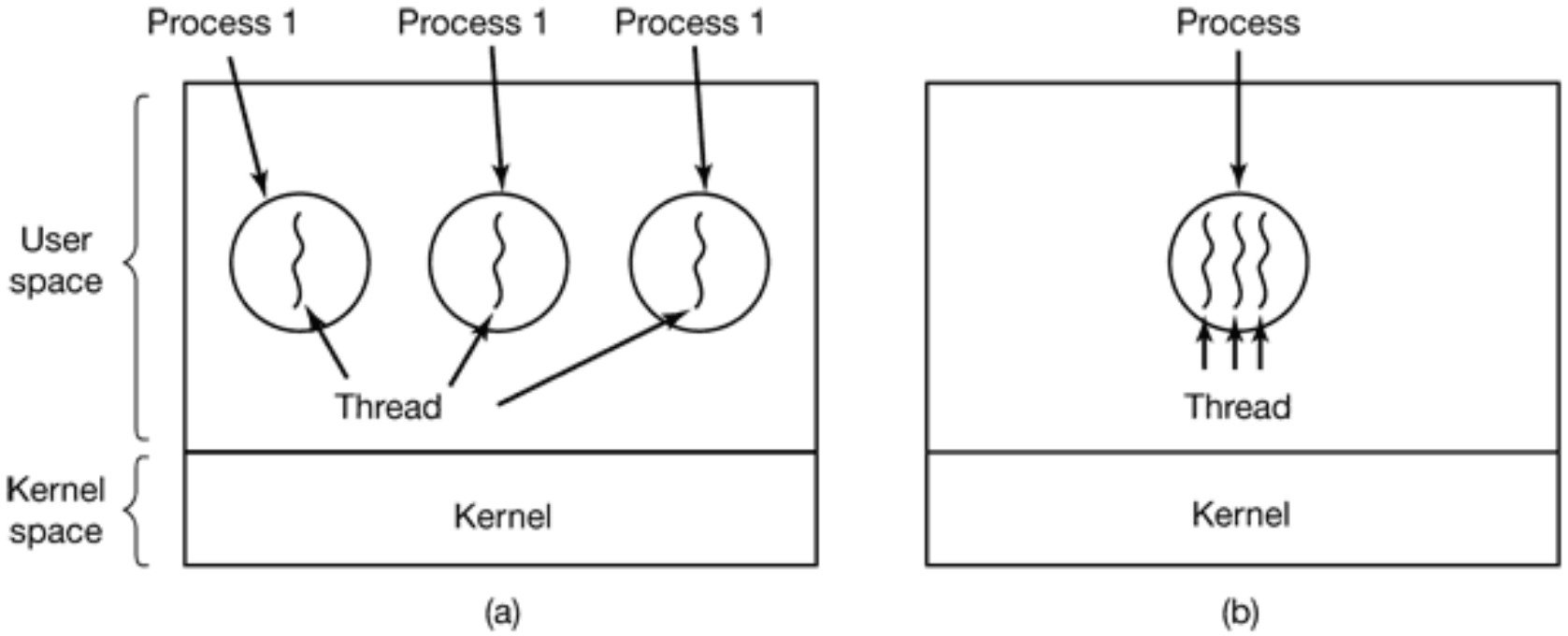
THREAD (İŞ PARÇACIĞI)

- Proses model iki temel kavram üzerine kurulmuştur: Kaynakların gruplandırılması ve programın icrası
- Her bir proses ana bellek üzerinde kendine ait bir adres alanına sahiptir. Bu adres alanında program metni, veriler ve prosesin çalışması için gerekli olan kaynakların bilgileri yer alır. Bu kaynaklar açık olan dosyalar, çocuk prosesler, bekleyen alarmlar, sinyal bilgileri, hesaplama için kullanılan parametreler vb bilgilerdir.
- Proses çalışması için gerekli olan bütün kaynakları proses kendi adres bölgesinde gruplandırırak, daha hızlı çalışan ve daha kolay yönetilebilen bir forma sokulmuş olur.

- Diğer kavram ise, prosesin icra edilen kısmı ki, bu kısma **thread (iş parçacığı)** adı verilmektedir.
- **İş parçacığı,** bir sonraki komutun adresini tutan bir program sayacına, çağrılmış ama daha sonlanmamış her bir prosedür (fonksiyon) için ayrı bir alan tutan yığına ve işlemci içindeki registerlardaki değerleri saklayan register değişkenlerine sahiptir.
- Prosesler kaynakları gruplandırak için kullanılırken, iş parçacıkları işlemci içersine giren ve prosesin işlem kısmını icra eden kod parçalarıdır.

- İş parçacıkları, bir proses içinde birden fazla işlem biriminin çalışmasına olanak sağlayan yapılardır.
- Tek bir proses içerisinde çoklu iş parçacıklarının paralel (multi-threading) çalışması, multiprogramming yapısına sahip bir işletim sisteminde birden fazla prosesin aynı anda çalışmasına benzerdir.
- Bir proses içerisindeki tüm iş parçacıkları, aynı adres uzayını,açık dosyaları, genel değişkenler gibi kaynakları paylaşırlar. Prosesler ise fiziksel belleği, diskleri, yazmaçları ve diğer genel kaynakları paylaşırlar.

- Aşağıdaki Şekil a'da herbiri tek bir iş parçacığı içeren üç farklı proses gösterilmektedir. Bu proseslerin her biri ayrı bir adres alanına sahiptir. Şekil b'de ise üç iş parçacığına sahip bir proses gösterilmiştir. Şekil b'deki iş parçacıkları aynı prosese ait oldukları için aynı adres alanını kullanırlar.

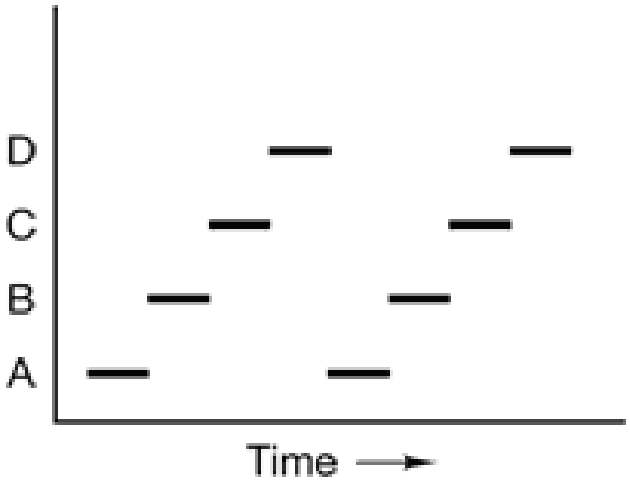


- Bir prosese ait bütün iş parçacıkları aynı adres alanını kullanılırlar. Yani prosese ait global değişkenler bütün iş parçacıkları tarafından paylaşılır. Bir iş parçacığı, ait olduğu prosesin adres alanına erişebildiği için, bu alanı okuma, bu alana yazma yetkisine sahiptir.

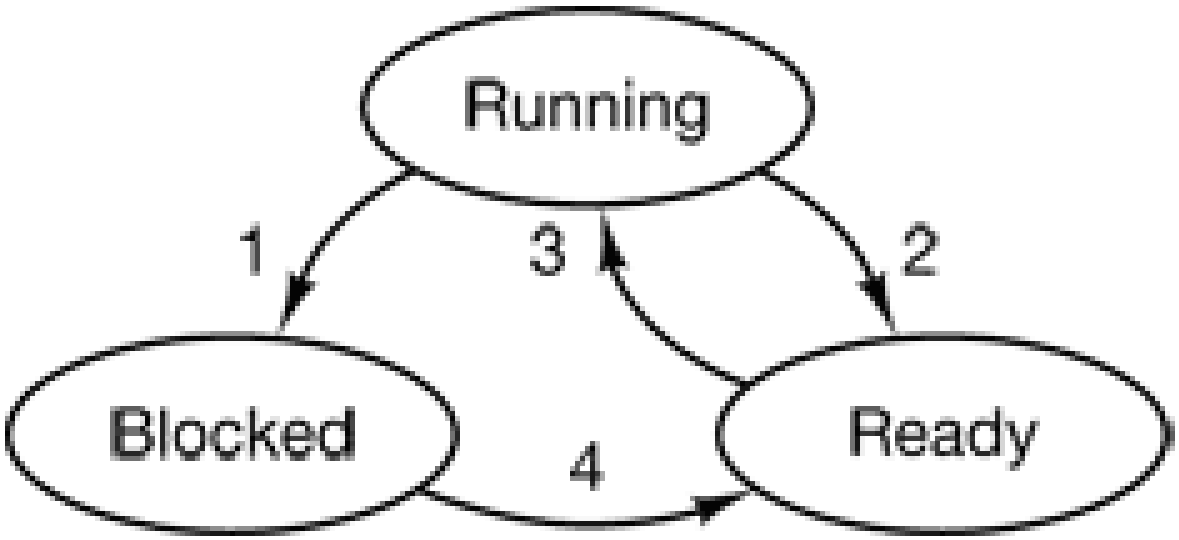
- İş parçacıkları prosesler gibi bir yarış durumunda değil bir dayanışma içindedirler. İş parçacıkların prosesin adres alanını paylaştıkları gibi, açık olan dosyalarını, coçuk proseslerini, alarmlar ve sinyaller gibi başka verilerini de ortak

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

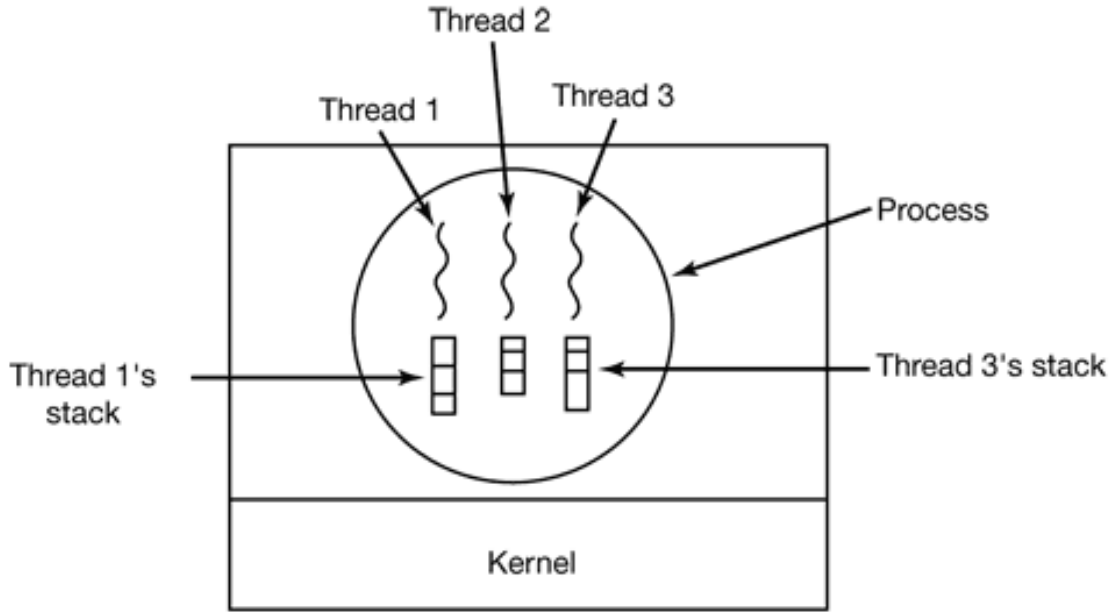
- Multithreading, multiprogramming'e benzer bir çalışma yapısına sahiptir. İşlemci iş parçacıkları arasında hızlı bir şekilde anahtarlama yaparak, sanki aynı anda birden fazla iş parçacığını çalıştırıyormuş izlenimi verir.



- Prosesler gibi iş parçacıklarının da içinde bulunabileceği durumlar: **çalışıyor (running), bloklanmış (blocked), hazır (ready), or sonlandırılmış (terminated)**



- Her bir iş parçacığı kendine ait bir yığın alanına sahiptir. Bu yığın çağrılmış fakat daha sonuç döndürmemiş her bir prosedür için bir alan ayırır. Bu alanda prosedürün yerel değişkenleri ve prosedür görevini tamamladığında hangi adrese dönüleceği bilgisi tutulur. Her bir iş parçacığı kendi bünyesindeki prosedürleri çağırabilir. Dolayısı ile her bir iş parçacığının kendi yığın alanına sahip olması gerekir.



- Multithreading özelliği olan sistemlerde her bir proses normal olarak tek bir iş parçacığı olarak çalışmaya başlar. Bu iş parçacığı "*thread_create*" prosedürü ile yeni bir iş parçacığı oluşturabilir. "*thread_create*" prosedürüne çalıştıracağı fonksiyon bir parametre olarak atanır.
- Bir iş parçacığı görevini tamamladığında "*thread_exit*" prosedürünü çağırarak görevini sonlandırır ve kendisine ayrılan bellek alanını boşaltmış olur.
- Diğer yaygın kullanılan bir iş parçacığı çağrısı ise "*thread_yield*" prosedürüdür. Bu çağrı ile iş parçacığı gönüllü olarak işlemci zamanını kardeş iş parçacığına verir.

thread_create.c

```c
#include <pthread.h>
#include <stdio.h>
//threadin calistiracagi fonksiyon bu sekilde tanimlanmalidir.
//void* F(void* data)
void* print_xs(void* unused){
      while(1){
       fputc('c',stderr);
      }
      return NULL;
}

int main(){
    pthread_t thread_id; //threadi tutan degisken, thread numarasi

    //yeni thread olusturur,
    //ilk parametre thread_noyu tutacak degisken
    //ikinci parametre threadin özellikleri (thread_attribute nesnesi)
    //üçüncü değişken threadin çalıştıracagi fonksiyon
    //dördüncü değişken threadin fonksiyonuna gönderilecek değişken
    pthread_create(&thread_id,NULL,&print_xs,NULL);

    while(1){
       fputc('o',stderr);
    }
    return 0;
}
```

thread_create2.c

```c
#include <pthread.h>
#include <stdio.h>

struct karakter_cikti_parametreleri{
        char karakter; //bu karakter
        int kackez;    //kac kez ekrana yazilacak
};
void* karakter_yaz(void* parametreler){
      struct karakter_cikti_parametreleri* p =
              (struct karakter_cikti_parametreleri*) parametreler;
      int i;
      for (i=0;i<p->kackez;i++){
          fputc(p->karakter,stderr);
      }
      return NULL;
}
int main(){
    pthread_t thread_1;
    pthread_t thread_2;
    struct karakter_cikti_parametreleri p1={'c',3000};
    struct karakter_cikti_parametreleri p2={'o',1000};

    pthread_create(&thread_1,NULL,&karakter_yaz,&p1);

    pthread_create(&thread_2,NULL,&karakter_yaz,&p2);

    return 0;
}
```

İŞ PARÇACİĞİ (THREAD) KULLANİMİ

- İş parçacığı kullanımındaki en temel avantaj, bir prosesa ait bir çok işlevin eş zamanlı olarak yapılabilmesidir. Prosesler bazı durumlarda bloklanabilir ama proseslerin iş parçacıklarına ayrılması ile bu prosese ait bir iş parçacığı bloklandığında diğerleri işleme devam edebilir. Böylece tam olmasada yarı-paralel bir çalışma özelliğini ortaya çıkmaktadır.

- İş parçacıklarının gerçek manada performans artışı sağladığı sistemler çok işlemcili sistemlerdir. Çünkü iş parçacıkları bu sistemlerde gerçek anlamda paralel çalışabilme imkanına sahiptirler.

IŞ PARÇACIĞI KULLANİM NEDENLERI

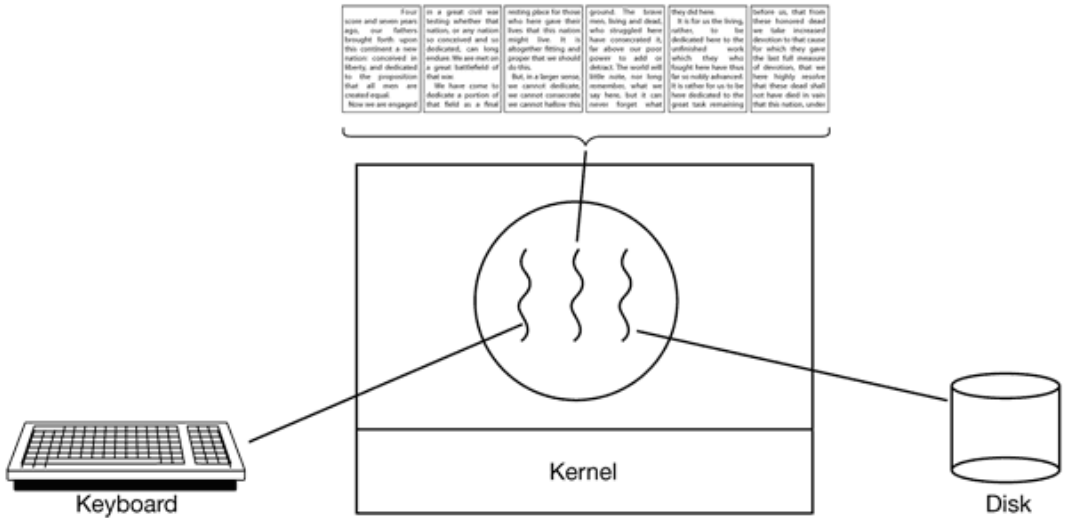
1. Bazı parelel çalışan işlevler ortak bir adres bölgesi kullanmak isterler. Bu adres bölgesinin ortak kullanımı ancak iş parçacığı kullanımı ile mümkün olur.
2. İş parçacıklarının kendine ait verileri çok fazla olmadığı için proseslere oranla daha kolay oluşturulup, yok edilebilirler. Bir çok sistemde iş parçacığı oluşturma işlemi, proses oluşturmaktan 100 kat daha hızlı gerçekleştirilir.
3. İş parçacıklarının hepsi CPU-bağımlı ise performans kazancı sağlamak çok mümkün olmamaktadır. Fakat iş parçacıklarının bir kısmı CPU-bağımlı, bir kısmı ise I/O bağımlı ise, bu iş parçacıklarının eş zamanlı çalışması, dolayısı

THREAD KULLANİM ÖRNEKLERİ

- İlk örneğimiz, bir kelime işlemcisi olsun. Bir çok kelime işlemcisi dökümanı sanki kağıt üzerindeymiş gibi ekrana getirir.
- Şimdi 800 sayfalık bir dökümanın ilk sayfasındaki bir paragrafın silindiğini düşünelim. Ardından kullanıcı 600. sayfaya gitmek istesin. Kelime işlemcisi olan bu program 1. sayfadan 600. sayfaya kadar bütün dökümanı yeniden formatlayıp, o sayfayı görüntüleyebilir. Çünkü 600. sayfadaki ilk paragrafın ne olacağı belli değildir. Bu durumda 600. sayfa görüntülenmeden ciddi bir geçikme söz konusu olacaktır.

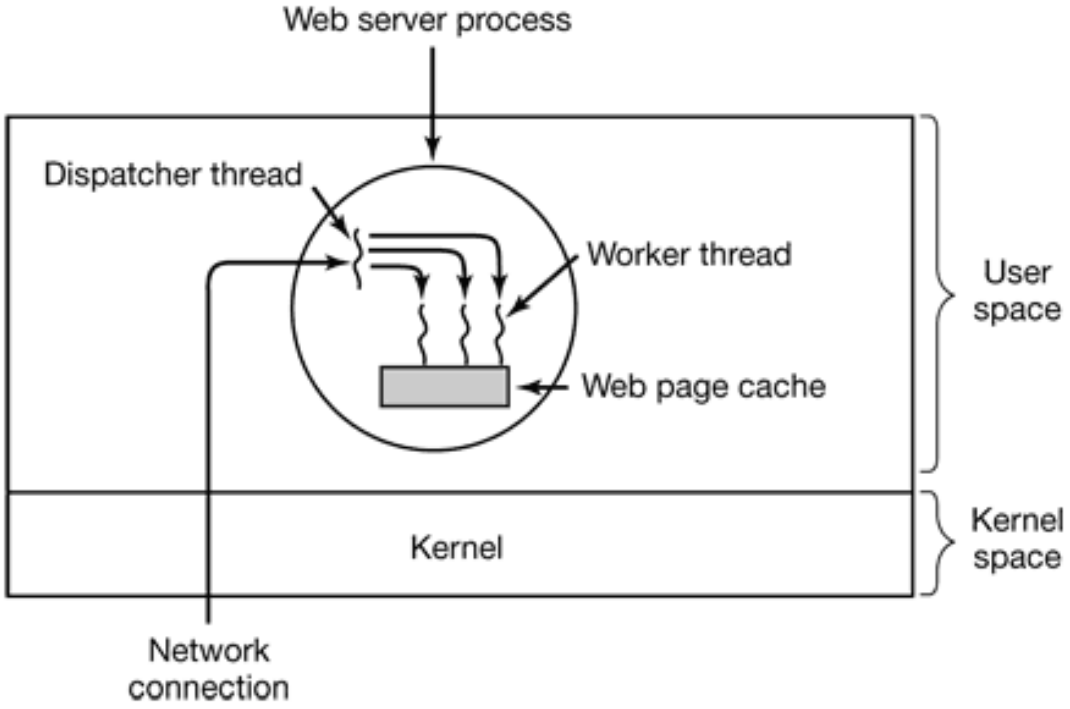
- Bu problemin çözümünde iş parçacıları kullanımı gerekecektir. Kelime işlemcisi iki iş parçacığından oluşan bir program olarak kodlanırsa, birinci iş parçacığı kullanıcı ile etkileşimi sağlarken, diğeri dokümanın yeniden formatlanması için çalışacaktır.
- Birinci sayfadan bir paragraf silindiğinde, etkileşimci iş parçacığı, formatçı iş parçacığına bütün dokümanı formatlamasını söyleyecektir. Bir tarafta etkileşimci iş parçacığı kullanıcının klavyesinden ve faresinden gelen komutları dinleyip, birinci sayfa üzerindeki işlemleri icra ederken, arka planda formatçı iş parçacığı deli gibi çalışarak bütün dokümanı yeniden formatlayacaktır.

- Birçok kelime işlemcisinde otomatik olarak belirli aralıklarla dokumanı kaydetme özelliği vardır. Yani prosesin bellekteki verileri, elektrik kesintisi, bilgisayar kapanması gibi tehlikelere karşı diske kaydedilir. Üçüncü bir iş parçacığı ile periyodik aralıklar ile veriler diske kaydedilir.



- Eğer bu program üç iş parçacığından değil de tek iş parçacığından oluşsaydı ne olurdu?

- Şimdi bir web sunucusunu düşünelim. Bu sunucuya istemciden çeşitli sayfa istekleri geliyor ve sunucu bu isteklere cevap veriyor.
- Bu sunucuda, dağıtıcı (**dispatcher)** bir iş parçacığı internet üzerinden gelen talepleri değerlendirir ve boşta beklemekte olan işçi (worker) iş parçacıklarından birini görevlendirir.



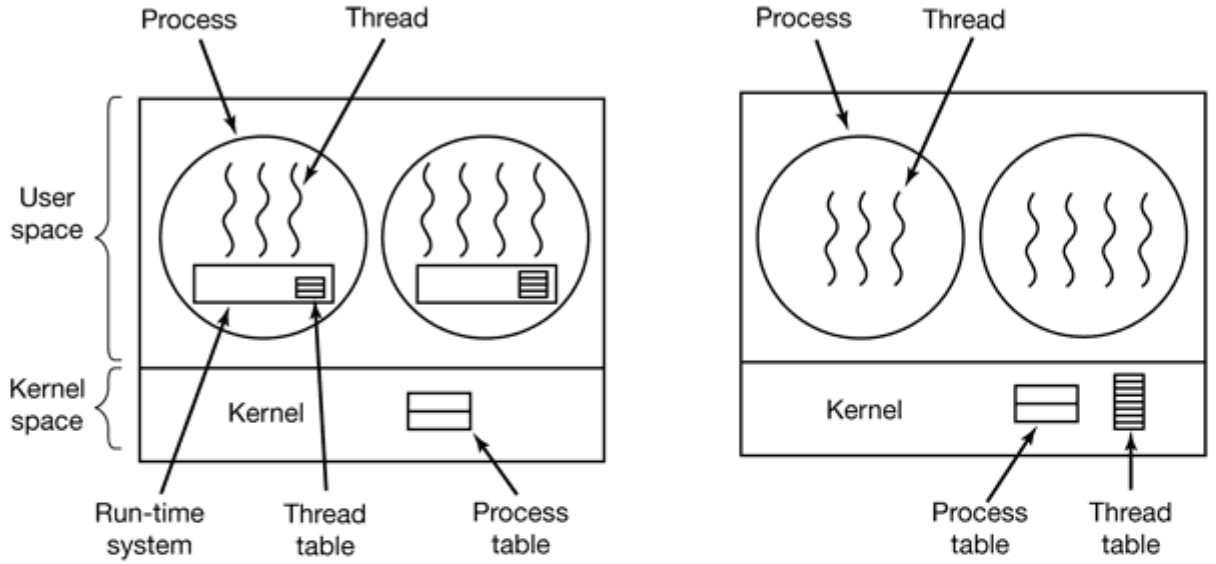
- Üçüncü örnek ise büyük miktarda verinin işlendiği bir proses olsun. Normalde data blok halinde okunur, daha sonra blok halinde işlenir ve son olarak blok halinde diske gönderilir.
- Bu proses üç iş parçacığından oluşacak şekilde yeniden tasarlanabilir. İş parçacığının biri gelen veriyi alıp tampon bellekte biriktirir. İkincisi tampondaki biriken veriyi işler ve sonuçları ayrı bir tampona yazar. Üçüncü iş parçacığı ise bu sonuç tamponundaki verileri diske kaydeder. Böylece bu üç iş parçacığının paralel olarak eş zamanlı çalışması mümkün hale gelmiş olur.

KULLANICI UZAYINDA ÇALIŞAN IŞ PARÇACIKLARI

- İş parçacıklarının kullanımı konusunda iki farklı yaklaşım vardır.
  - Kullanıcı uzayında çalışan iş parçacıkları
  - Kernelde çalışan iş parçacıkları
  - Birde hibrit sistemler in kullanımı vardır.
- Kullanıcı uzayında çalışan iş parçacıklarını işletim sistemi desteklemez. Çekirdek, çalışan bir proses içerisindeki iş parçacıklarından habersizdir ve bu prosese tek iş parçacığından oluşuyormuş gibi davranır.
- Bu yöntemin en açık ve en önemli avantajı, işletim sisteminin kendisi iş parçacıklarını desteklemese dahi sistemde iş parçacıklarının kullanılması mümkün hale getirilmiş olur.

- İş parçacıkları bir Run-Time sisteme bağlı olarak çalışırlar. Bu Run-Time sistem iş parçacıklarının yönetiminde kullanılan birçok fonksiyonlara sahip bir kütüphanedir. Bu fonksiyonlardan bazıları: *thread_create*, *thread_exit*, *thread_wait*, and *thread_yield.*
- İş parçacığı yönetimi kullanıcı uzayında yapıldığı için her bir proses kendi **thread tablosunu** tutar**.** Bu thread tablosunda proses tablosu gibi çok veri içermez. Sadece her bir iş parçacığına ait program sayacı, yıgın göstergeci, durum ve yazmaç değerlerini tutulur. Bir iş parçacığı bloklanmış ya da hazır moda geçip tekrar işlemciye girdiğinde aynen kaldığı yerden işlemine devam etmemesi gerekir.

- Bir iş parçacığı kendini bloklayacak bir işlem gerçekleştirdiğinde Run-Time sistem prosedürlerini çağırır. Bu prosedür gerçekten bloklanmayı gerektiren bir durum olup olmadığını kontrol eder. Eğer gerekiyorsa, bloklanan iş parçacığına ait yazmaç değerlerini thread tablosuna yazar, yeni iş parçacığına ait değerleri thread tablosundan işlemcide yazmaçlara yükler. İş parçacıklarının bu şekilde anahtarlanması, kernel tarafından anahtarlamaya göre çok daha hızlı bir şekilde gerçekleşir.

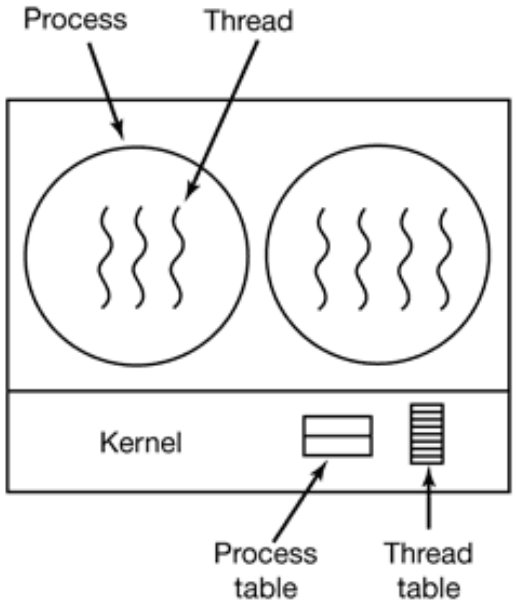


(a) A user-level threads package. (b) A threads package managed by the kernel.

- Bir iş parçacığı çalışmasına ara verince, işlemcideki yazmaç değerlerinin iş parçacığı tablosuna kaydedilmesi, ve iş parçacığı anahtarlaması Run-Time sisteme ait prosedürler tarafından gerçekleştirilir. Bu prosedürleri çağırmak kernel'ı çağırmaya göre çok daha hızlı bir şekilde gerçekleşir. Çünkü ne bir trap'a, ne proses anahtarlamaya ne de ön bellekin (cache) yenilemesine gerek yoktur. Bu da iş parçacığı anahtarlamasının çok hızlı bir şekilde gerçekleştirilmesini sağlar.

KERNEL'DA ÇALIŞAN İŞ PARÇACIKLARI

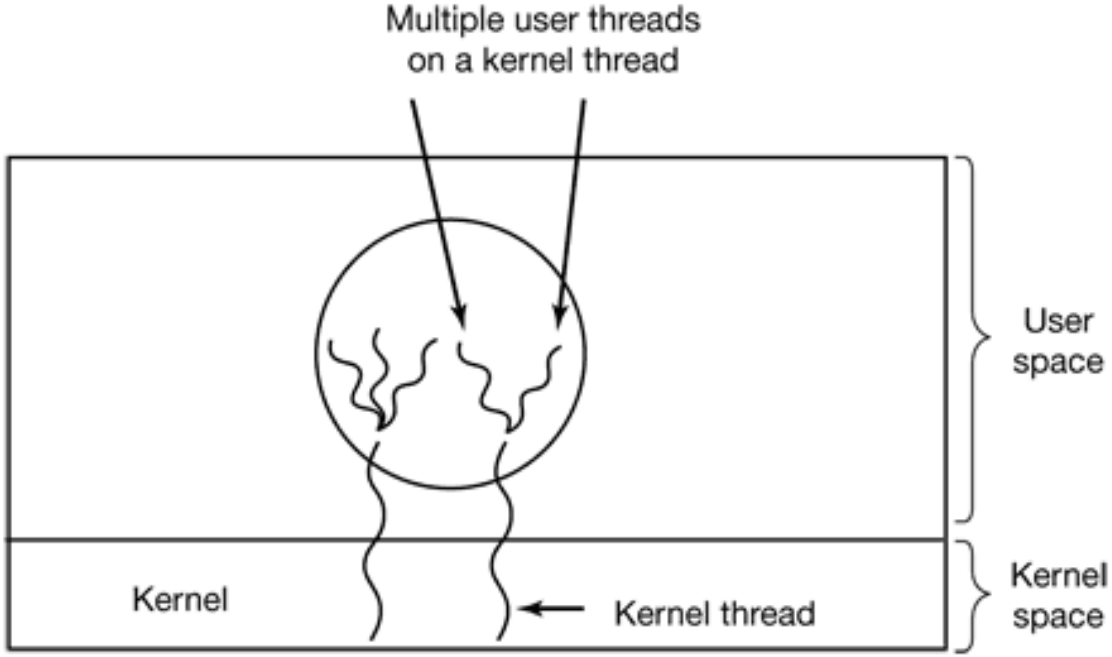
- Bu tip sistemlerde, kernel iş parçacıklarını tanır ve yönetir. Run-Time sisteme ihtiyaç duyulmaz. Proseslerin iş parçacığı tablosu tutmalarına da gerek kalmaz. Thread tablosu bizzat işletim sistemi tarafından tutulur. Bir iş parçacığı yeni bir iş parçacığı oluşturmak istediğinde ya da yok etmek istediğinde bir sistem çağrısı yaparak kernel'daki thread tablosu üzerinde işlem yapılır. Ayrıca kernel prosesler için bir de proses tablosu tutar



- İş parçacıklarının bloklanması sistem çağrıları aracılığı ile yapılır. Sistem çağrıları Run-Time sistem prosedürlerine göre çok daha maliyetli bir şekilde gerçekleşir. Bir iş parçacığı bloklandığında, kernel iş parçacığı tablosundan başka bir iş parçacığı anatharlar. Bu iş parçacığı çalışmakta olan prosese ait diğer bir iş parçacığı olabileceği gibi, farklı bir prosese ait bir iş parçacığı de olabilir. Kullanıcı uzayındaki iş parçacığılerde, Run-Time sistem, iş parçacığı anahtarlamayı hep aynı prosese ait iş parçacığıler üzerinden yapıyordu.
- Bu sistemin en büyük dezavantajı, iş parçacığı işlemleri (oluşturma, yok etme vb.) sistem çağrıları aracılığı ile yapılır ve bu da sisteme ağır bir yük getirir.

HİBRİD İŞ PARÇACIĞI UYGULAMALARI

- Kullanıcı uzayında çalışan iş parçacıkları ile kernel uzayında çalışan çalışan iş parçacıklarının her ikisinin avantajları birleştirilerek hibrit bir kullanım sağlanmıştır. Bu sistemde kullanıcı uzayında çalışan her bir prosese ait iş parçacığı grubuna kernel tarafından bir kernel iş parçacığı tahsis edilmi



- Bu tasarımda, kernel sadece kernel iş parçacıklarını tanır ve sadece onları anahtarlar.
- Kullanıcı uzayında çalışan iş parçacıkları, kernel'daki iş parçacıkları üzerinden işlemciye anahtarlanmaktadır.
- Kullanıcı uzayında iş parçacıkları ile ilgili işlemler yine işletim sisteminde habersiz bir şekilde Run-Time sistem üzerinde gerçekleştirilir.

SORULAR ??