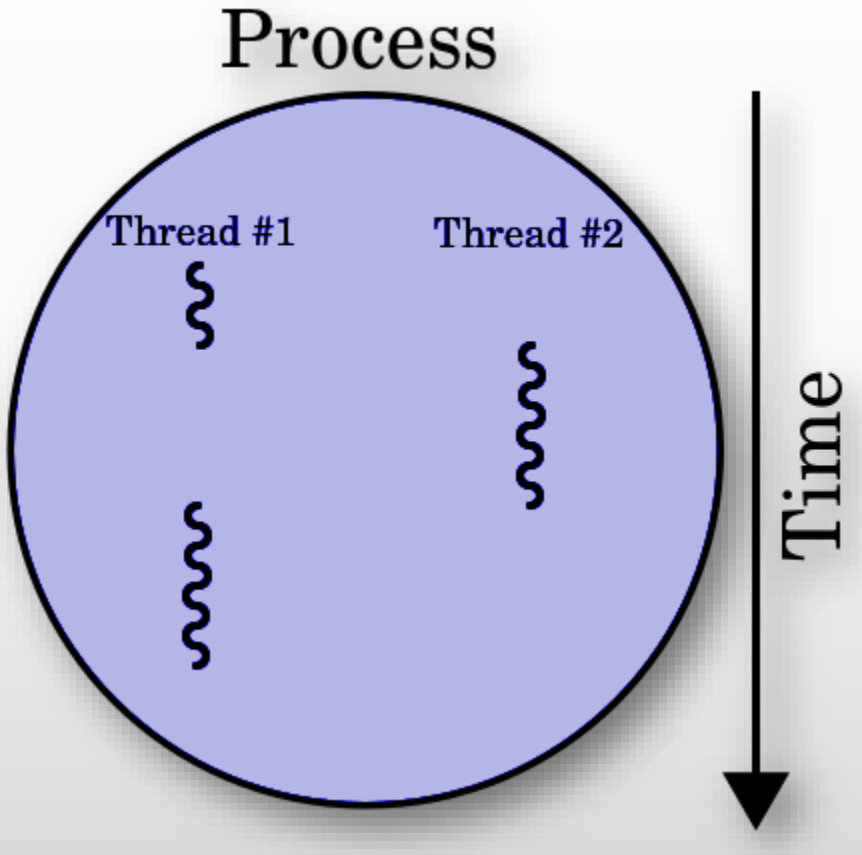
Bölüm 4: İş Parçacıkları

İşletim Sistemleri

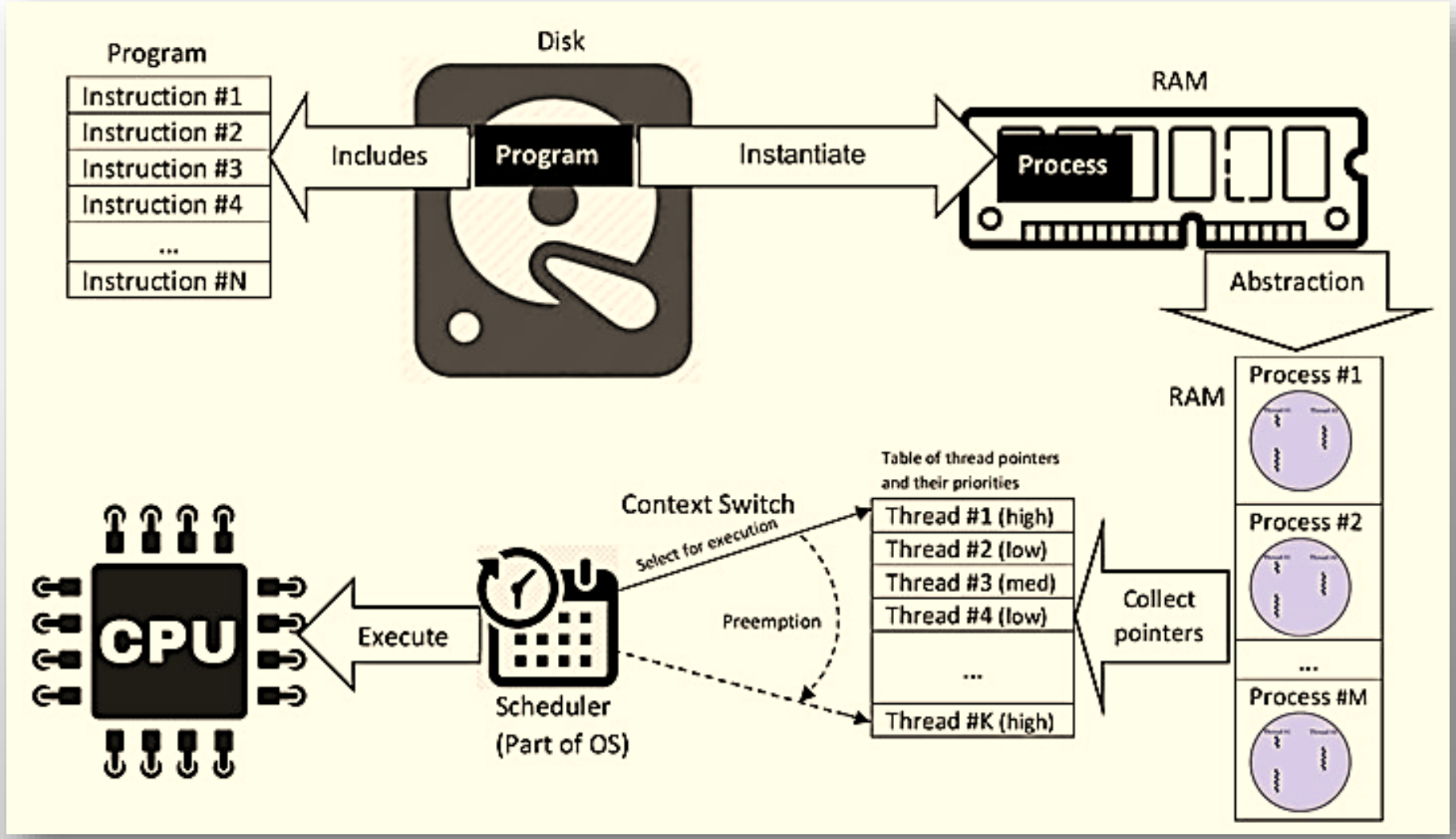
İş Parçacığı (Thread)

- Kendi başına çalışabilen, süreç içindeki en küçük iş birimi.
- Süreç, bir programın yürütülmekte olan bir örneği.
- İş parçacığı ise süreç içindeki birden fazla yürütme birimi.
- İş parçacıkları, süreçlere göre daha hafif ve hızlı bir iletişim sağlar.
- Aynı süreç içindeki iş parçacıkları aynı adres uzayını paylaşır.
- Çoklu iş parçacıklı uygulamalar, işlemci kaynaklarını daha etkin kullanır.

İş Parçacığı



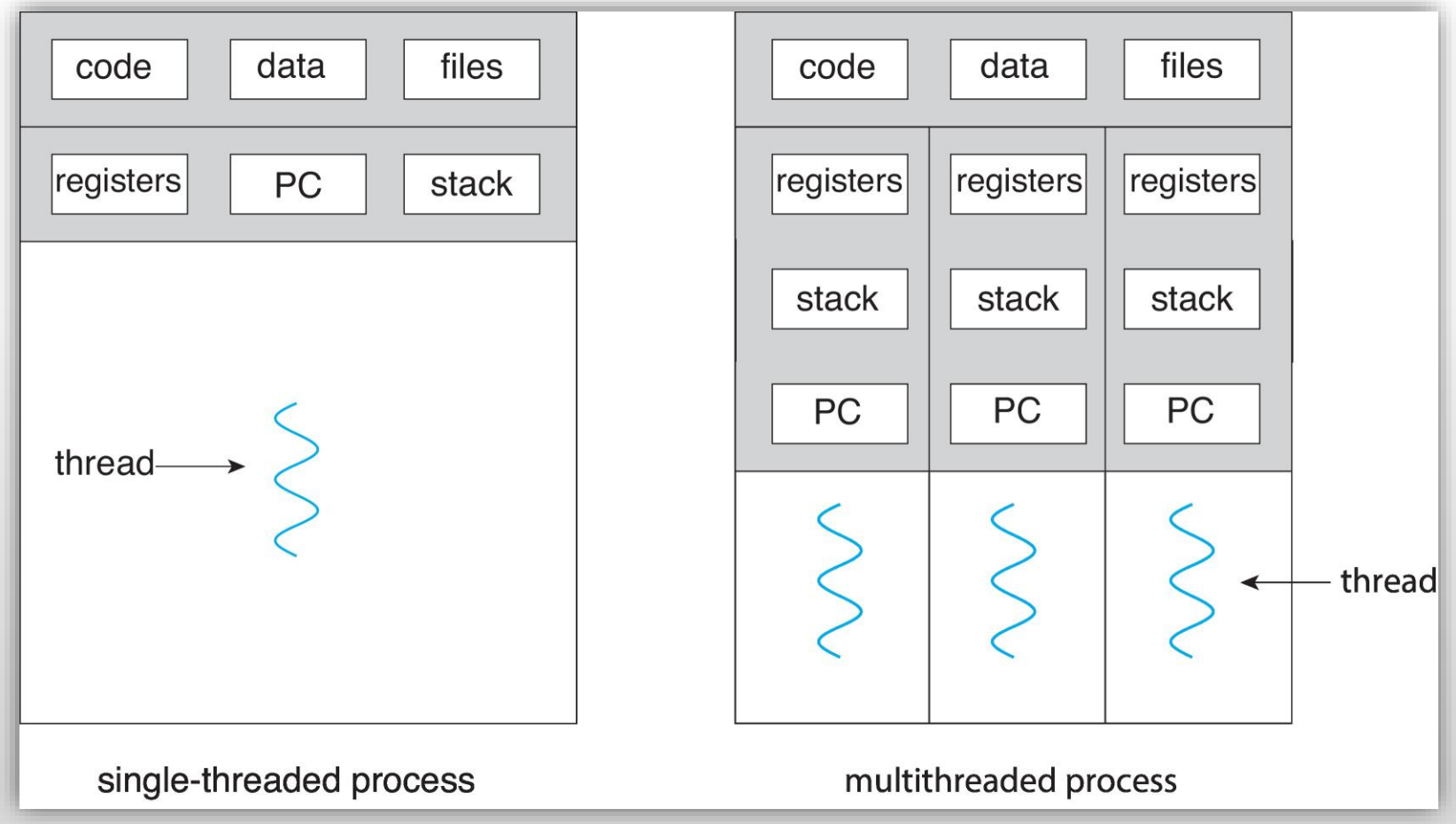
İş Parçacığı



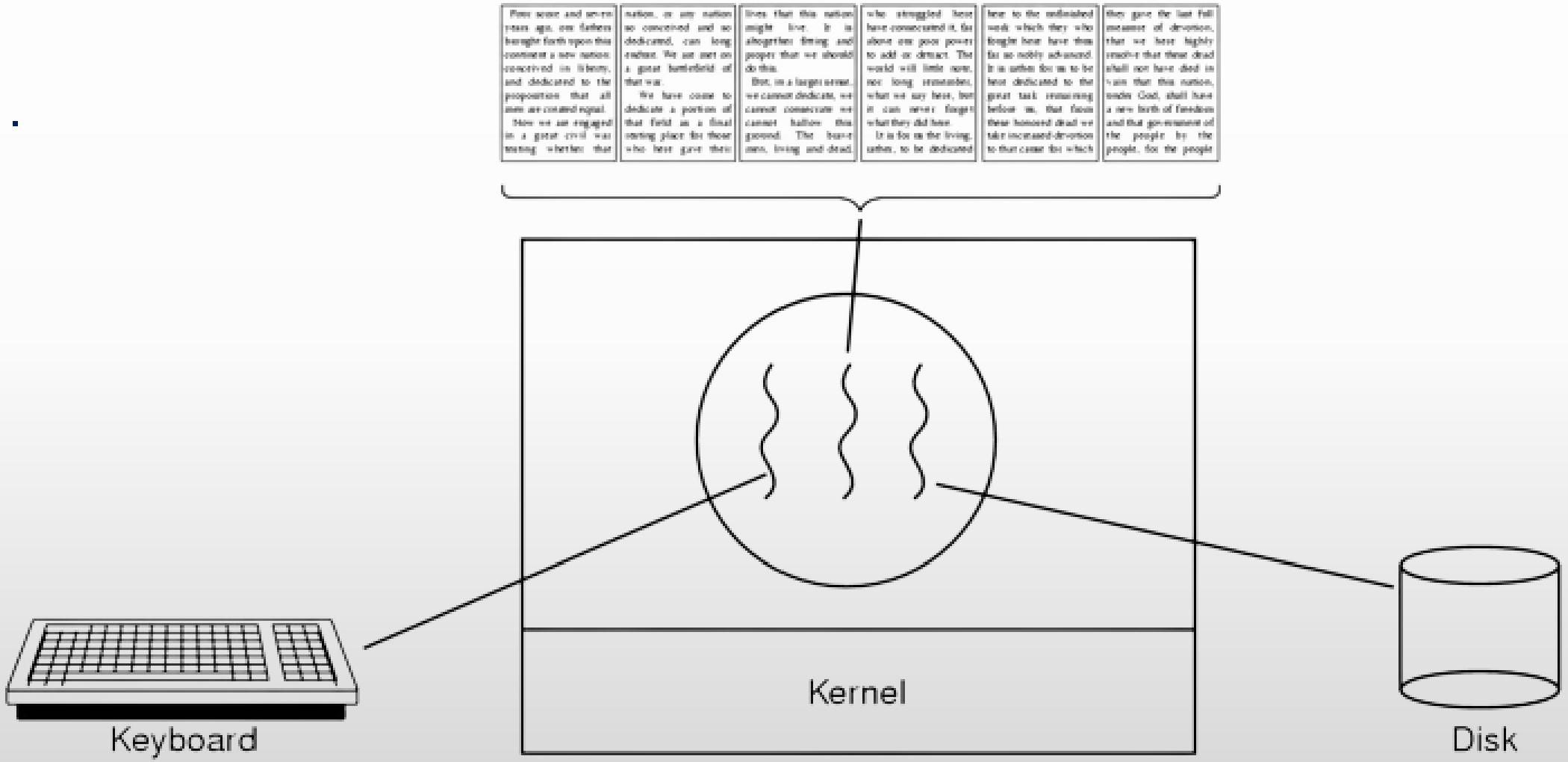
İş Parçacığı

- Modern uygulamalar çoklu iş parçacıklıdır (*multi-threaded*).
- İş parçacıkları uygulama içerisinde çalışır.
- Farklı görevler, ayrı iş parçacıkları tarafından yürütülebilir.
  - Ekranı güncelleme,
  - Veri getirme,
  - Bir ağ isteğini yanıtlama gibi.
- Süreç oluşturma masraflı, iş parçacığı oluşturma az maliyetlidir.
- Kodu basitleştirir, verimliliği artırır.
- İşletim sistemi çekirdeği genellikle çok iş parçacıklıdır.

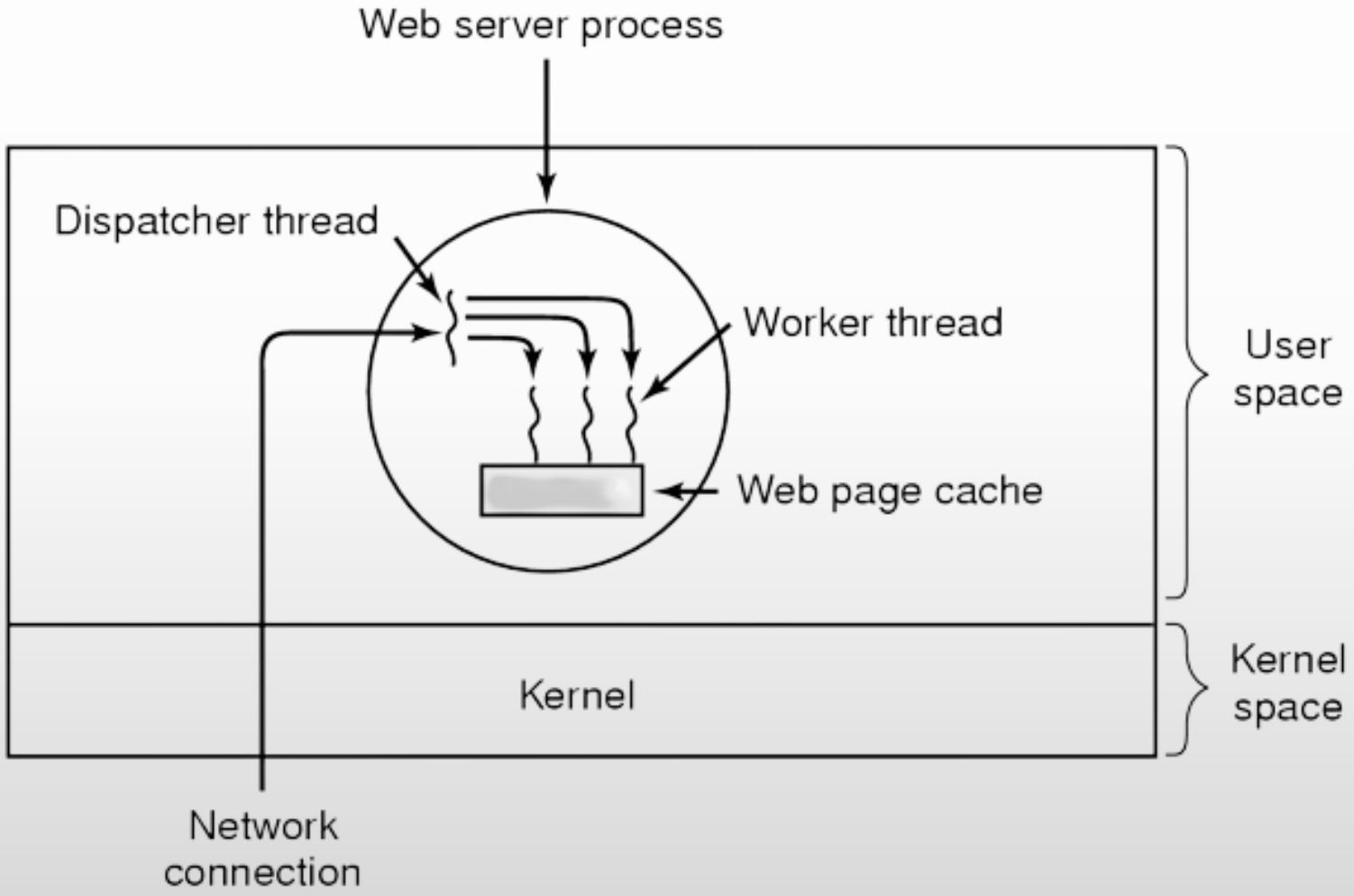
Tekli ve Çoklu İş Parçacığı



Üç Tane İş Parçacığına Sahip Uygulama

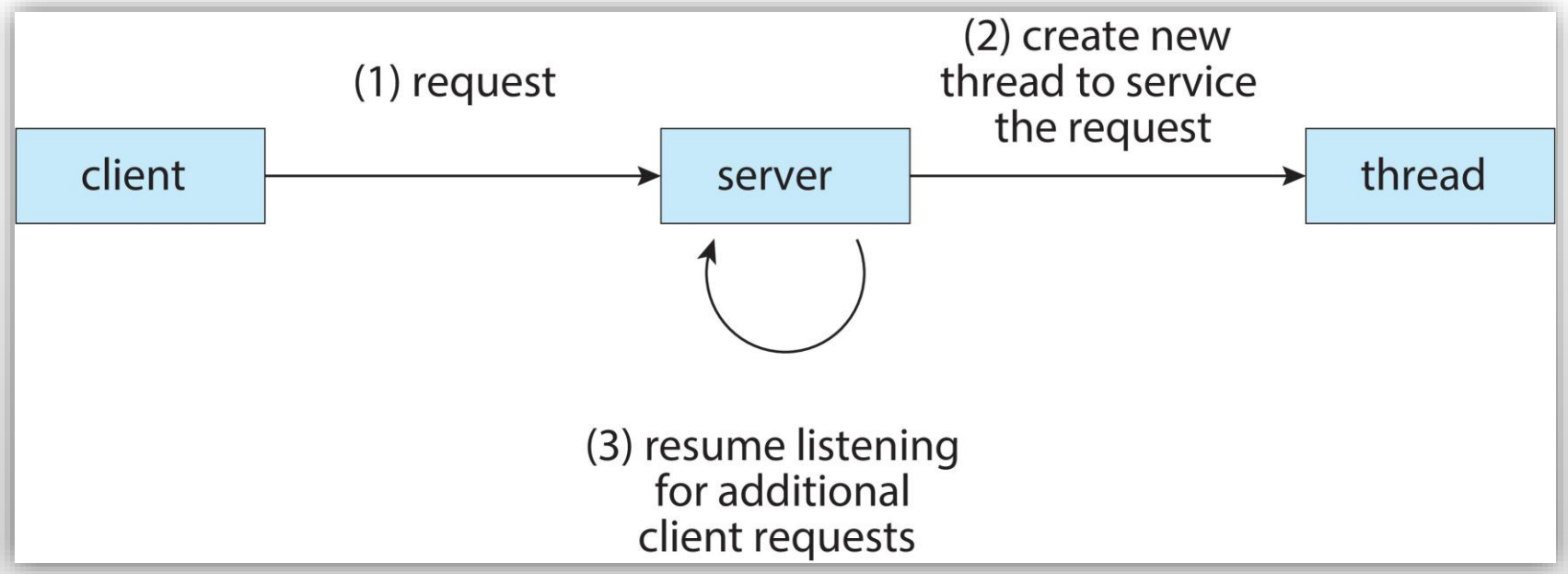


Çok İş Parçacıklı Web Sunucusu



Web Sunucusu

- .



İş Parçacığı Kullanımı

(a) İşlemci görev dağıtıcı (*dispatcher*) iş parçacığı

(b) İşçi (*worker*) iş parçacığı

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

Web Sunucusu

- Çoklu iş parçacığı kullanılmadığında,
  - Eğer sayfa önbellekte değilse, süreç bloke olur.
  - Sayfanın hazır olması beklenirken, işlemci hiçbir şey yapamaz.
- Çoklu iş parçacığı kullanıldığında ise,
  - Sunucu sayfanın hazırlanmasını bir iş parçacığına aktarır.
  - Çalışmaya devam eder.

Web Sunucusu Geliştirmek için Üç Farklı Yol

- Çoklu iş parçacıklı süreç (*multi-threaded process*)
  - Paralellik var, sistem çağrıları bloke olur.
- Tek iş parçacıklı süreç (*single threaded process*)
  - Paralellik yok, sistem çağrıları bloke olur.
- Sonlu durum makinesi (*finite state machine*)
  - Paralellik var, sistem çağrıları bloke olmaz, kesmeler.

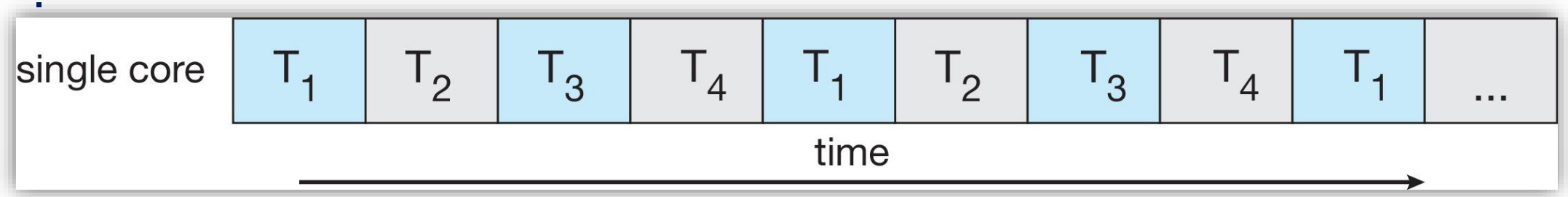
Çoklu İş Parçacıklı Süreç Faydaları

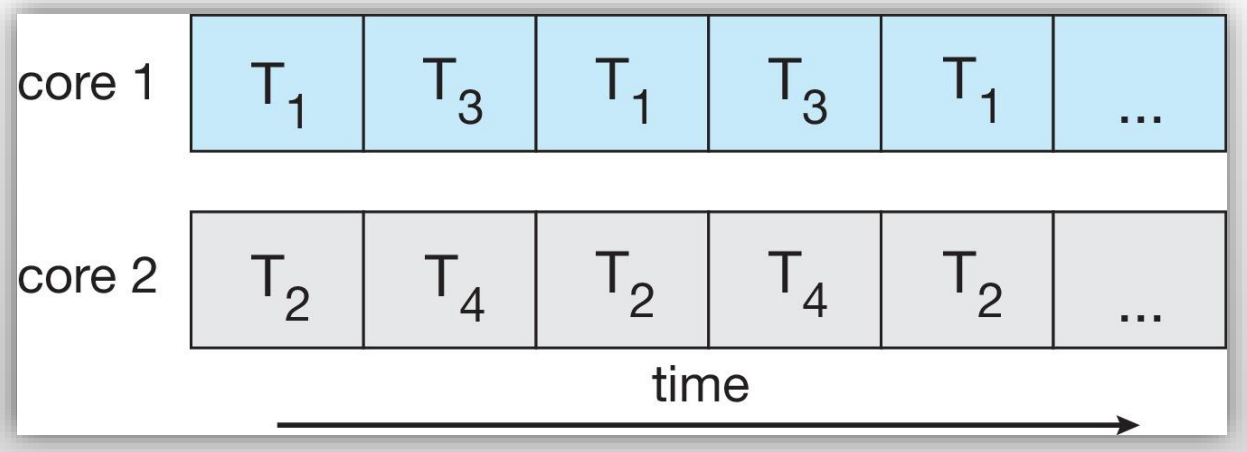
- **Duyarlılık**: özellikle kullanıcı arayüzleri için önemli, sürecin bir kısmı bloke olsa da yürütme devam eder.
- **Kaynak Paylaşımı:** iş parçacıkları, ait oldukları sürecin kaynaklarını paylaşırlar, paylaşımlı bellek veya mesaj iletmeye göre daha kolaydır.
- **Maliyet**: süreç oluşturmaya kıyasla daha az maliyetli, iş parçacığı değiştirme, bağlam anahtarlamaya göre sisteme daha az ek yük getirir
- **Ölçeklenebilirlik**: süreç, çok çekirdekli mimarilerden yararlanabilir.

Çok Çekirdekli Programlama

- **Paralellik (*parallelism*)**, sistem aynı anda birden fazla süreci yürütebilir.
- **Eşzamanlılık (*concurrency*)**, tüm süreçler zamanla ilerleme kaydeder.
- **Getirdiği zorluklar:**
  - Etkinlikleri bölme (*dividing activities*),
  - Denge (*balance*),
  - Veri bölme (*data splitting*),
  - Veri bağımlılığı (*data dependency*),
  - Test etme ve hata ayıklama (*test and debug*).

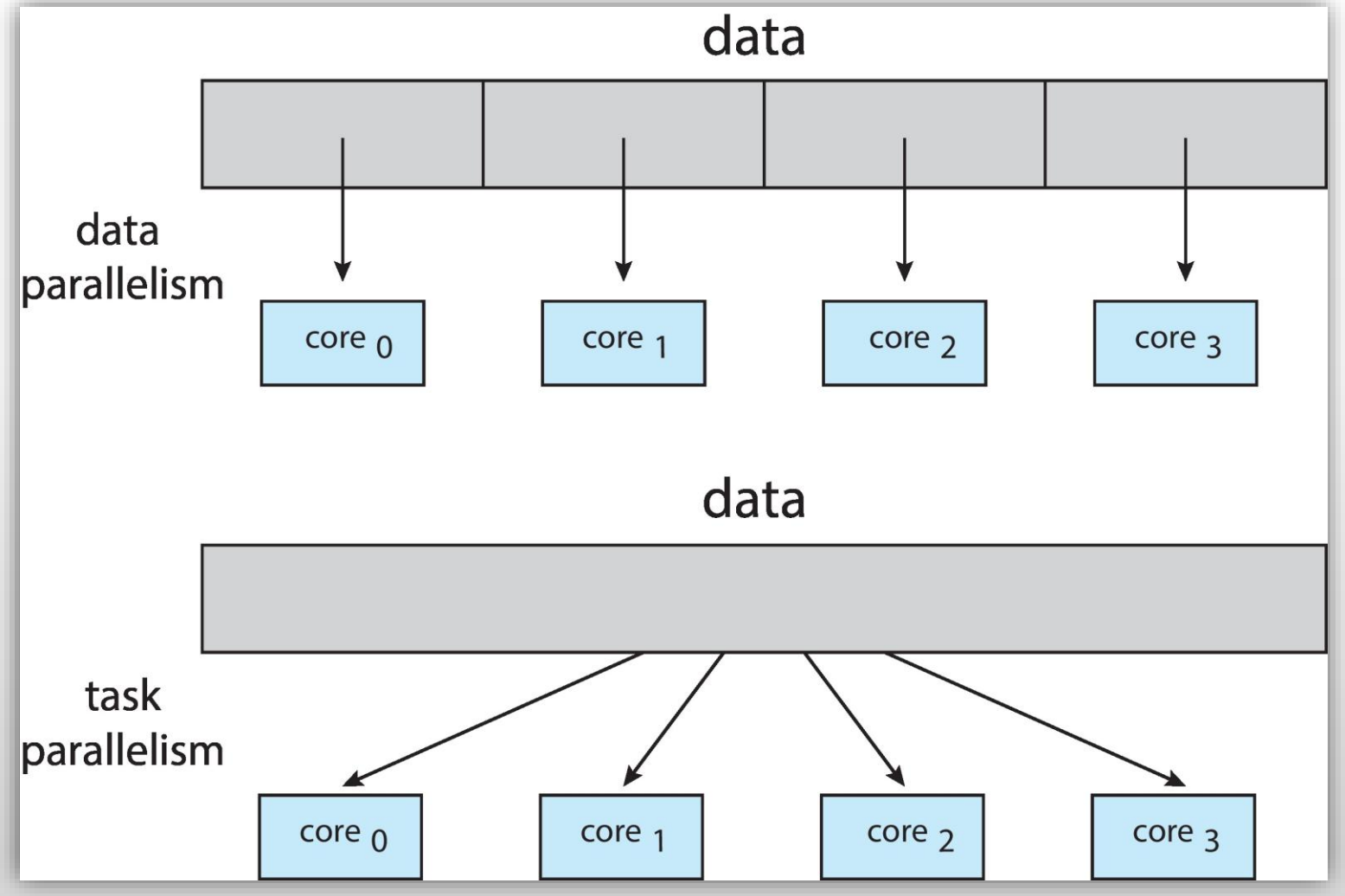
Parallellik ve Eşzamanlılık





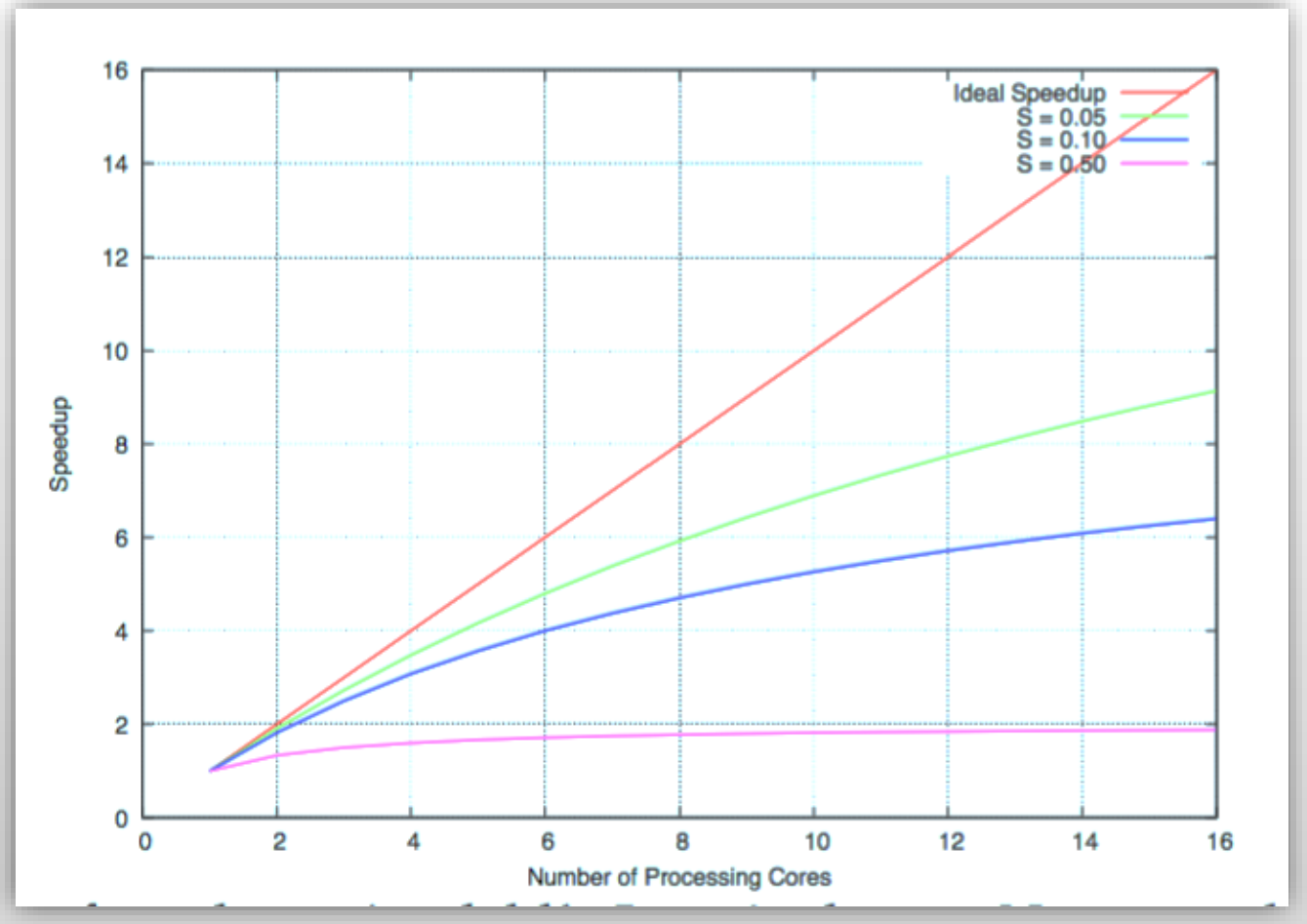
Veri ve Görev Bölme

- .



Amdahl Kuralı

- Programın %95'i paralel hale getirilebilirse, paralel hesaplama kullanan teorik maksimum hızlanma 20 kat olacaktır.



İş Parçacığı Modeli

- Sürece ait olan tüm iş parçacıkları ile paylaşılan veriler
  - Bellek adres uzayı (*address space*),
  - Global değişkenler (*variables*),
  - Açık dosyalar (*open files*),
  - Çocuk süreçler (*child processes*),
  - Bekleyen alarmlar (*waiting alarms*),
  - Sinyali ele alacak süreçler (*signal handlers*)

İş Parçacığı Modeli

- Her bir iş parçacığına özel veriler
  - Program sayacı (*counter*),
  - Yazmaçlar (*register*),
  - Yığın (*stack*),
  - Durum (*state*)

İş Parçacığı

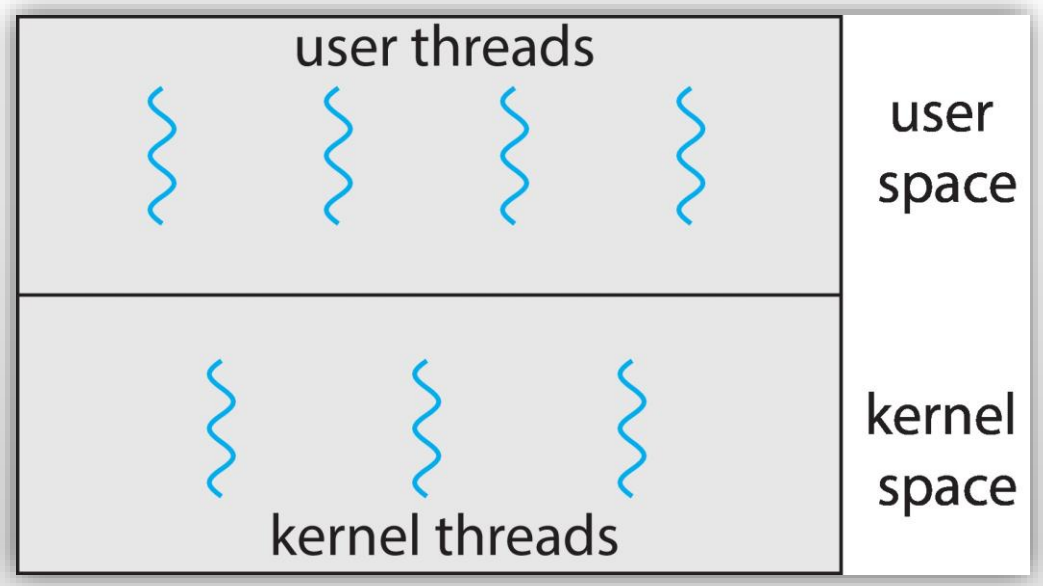
- Kendi program sayacı, yazmaç kümesi ve yığını var.
- Kod (*text*), global veri ve açık dosyaları, aynı süreç içerisinde paralel çalıştığı iş parçacıkları ile paylaşır.
- Kendi süreç kontrol bloğuna (*PCB*) sahip olabilir. (*İşletim sistemine bağlı*)
- Bağlam, iş parçacığı kimliği, program sayacı, yazmaçlar, ve yığın işaretçisini içerir.
- Aynı süreç içerisindeki iş parçacıklarıyla bellek adres uzayı ve bellek yönetimi bilgileri paylaşılır.

İş Parçacıkları Nasıl Çalışır

- Sürecin kendisi de bir iş parçacığı olarak başlar.
- İş parçacığı içeriği (kimlik, yazmaçlar, nitelikler).
- Yeni iş parçacıkları oluşturmak ve kullanmak için kütüphane çağrıları,
  - `thread_create`
    - parametre olarak aldığı prosedürü iş parçacığı olarak başlatır.
  - `thread_exit`
    - iş parçacığını sonlandırır.
  - `thread_join`
    - başka bir iş parçacığının sonlanmasını bekler.
  - `thread_yield`
    - CPU'yu bırakarak, diğer iş parçacıklarına çalışma şansı verir.

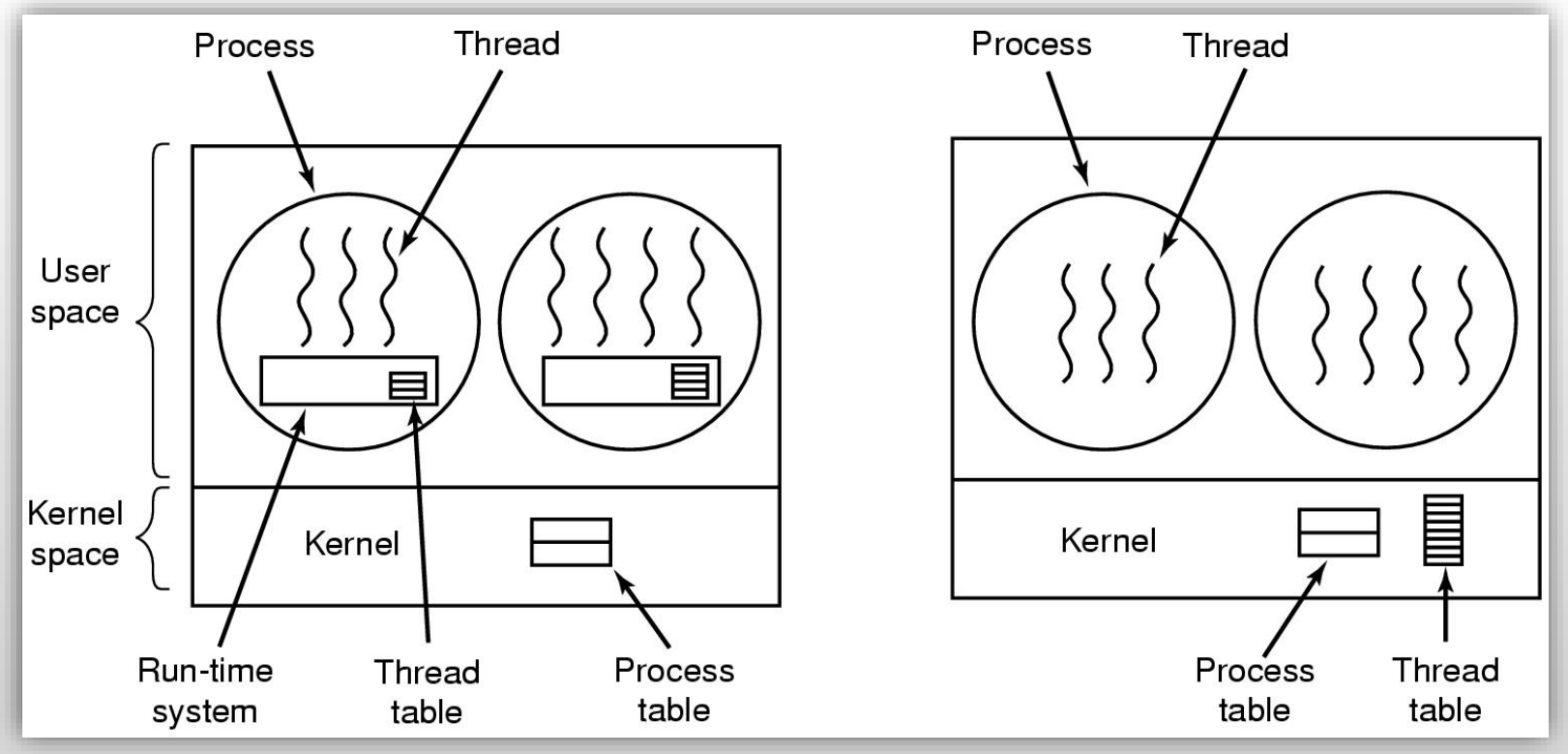
Kullanıcı ve Çekirdek Modu İş Parçacıkları

- .



Kullanıcı ve Çekirdek Modu İş Parçacıkları

(a) Kullanıcı düzeyinde iş parçacığı yönetimi. (b) Çekirdek tarafından yönetilen iş parçacıkları.



Çoklu İş Parçacığı Modelleri

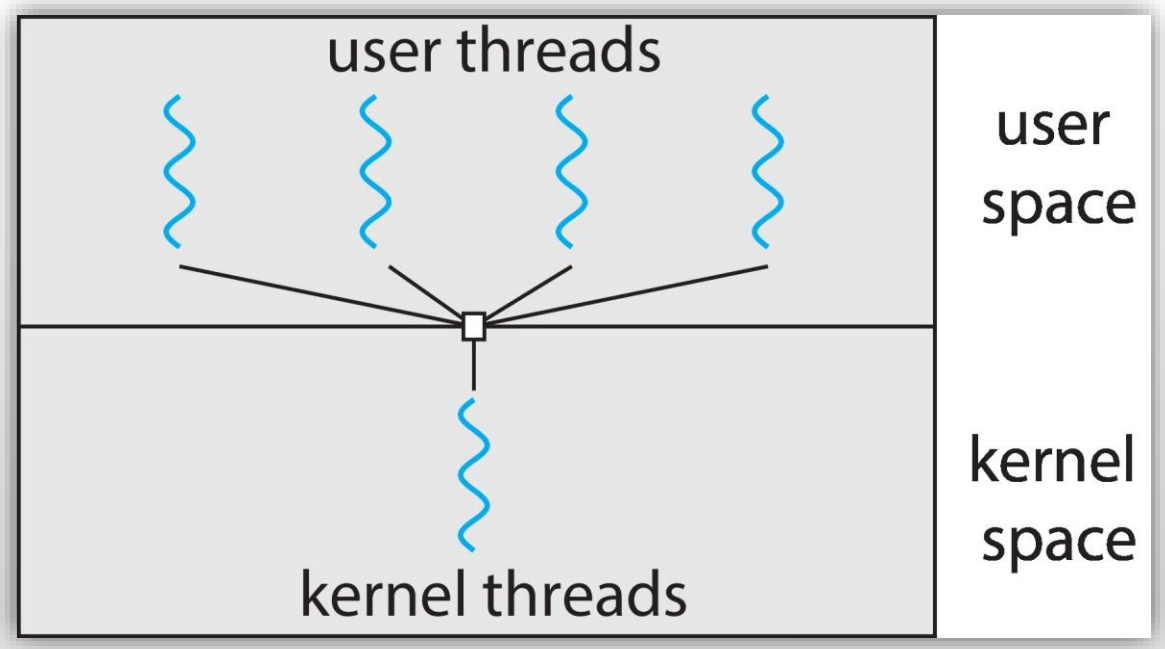
- Çoktan bire (*many-to-one*)
- Bire bir (*one-to-one*)
- Çoktan çoğa (*many-to-many*)

Çoktan Bire Yaklaşım

- Tek çekirdek iş parçacığına eşlenen birçok kullanıcı düzeyi iş parçacığı.
- Bir iş parçacığının bloke olması, tümünün bloke olmasına neden olur.
- İş parçacıkları, çok çekirdekli sistemde paralel olarak çalışmaz.
  - Çünkü; aynı anda yalnızca bir tanesi çekirdekte olabilir.
- Çok az sistem bu modeli kullanıyor.
  - *Solaris Green Threads, GNU Portable Threads*

Çoktan Bire Yaklaşım

- .

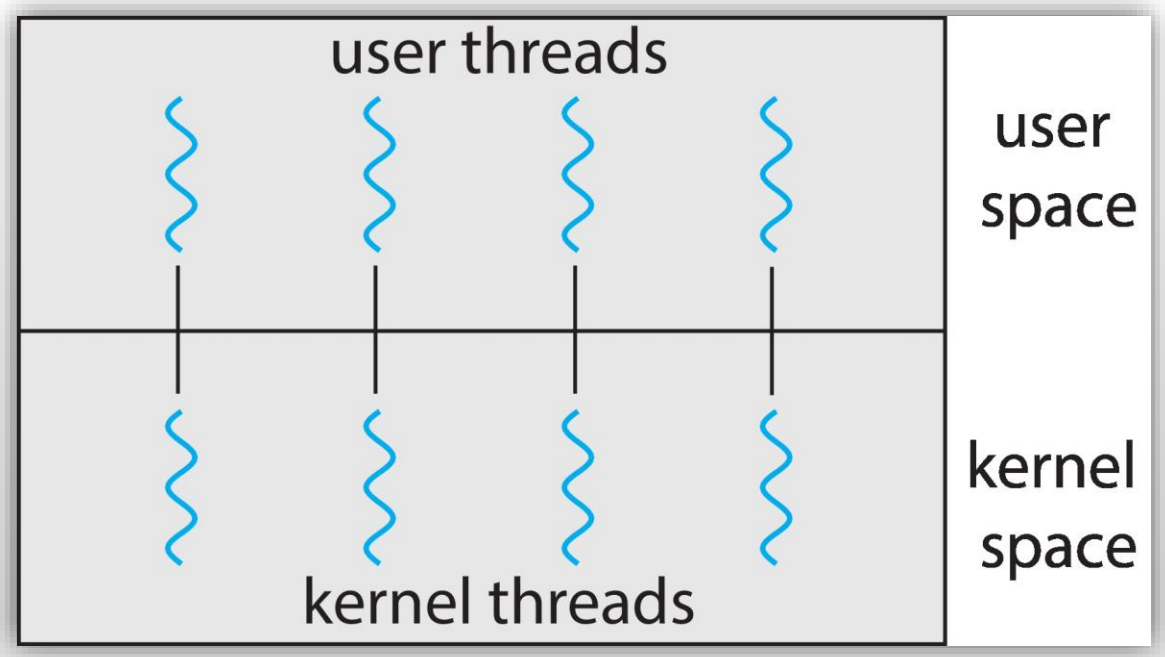


Bire Bir Yaklaşım

- Kullanıcı düzeyi her iş parçacığı, bir çekirdek iş parçacığına eşlenir.
- Kullanıcı düzeyi her bir iş parçacığına, bir çekirdek iş parçacığı oluşturulur.
- Çoktan bire yaklaşımına göre daha fazla eşzamanlılık sağlar.
- Süreç başına iş parçacığı sayısı bazen ek yük nedeniyle kısıtlanır.
- Örnekler;
  - *Windows, Linux*

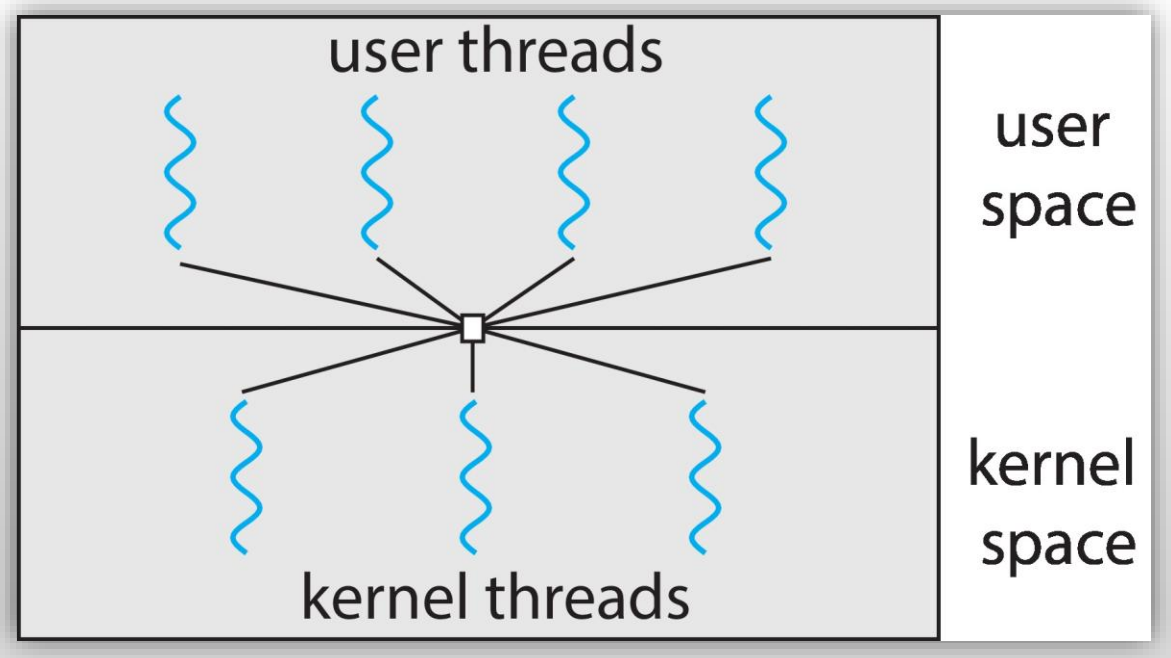
Bire Bir Yaklaşım

- .



Çoktan Çoğa Yaklaşım

- Birçok kullanıcı düzeyi iş parçacığı, birçok çekirdek iş parçacığına eşlenir.
- İşletim sistemi yeterli sayıda çekirdek iş parçacığı oluşturabilir.
- Yaygın değil.



Kullanıcı Modu İş Parçacıkları Avantajlar

- İş parçacığı tablosu, iş parçacığı hakkında gerekli bilgileri içerir,
  - böylece çalışma zamanı sistemi (*RTS*) tarafından yönetilebilir.
- İş parçacığı bloke olursa, çalışma zamanı sistemi (*run time system*),
  - iş parçacığı bilgilerini tabloda günceller.
  - çalıştırılacak yeni bir iş parçacığı bulur.
- Durum kaydetme ve çizelgeleme, çekirdek modunda daha hızlı çağrılır.
  - tuzak kapı yok (*no trap*),
  - önbellek temizleme yok (*no cache flush*).

Kullanıcı Modu İş Parçacıkları Dezavantajlar

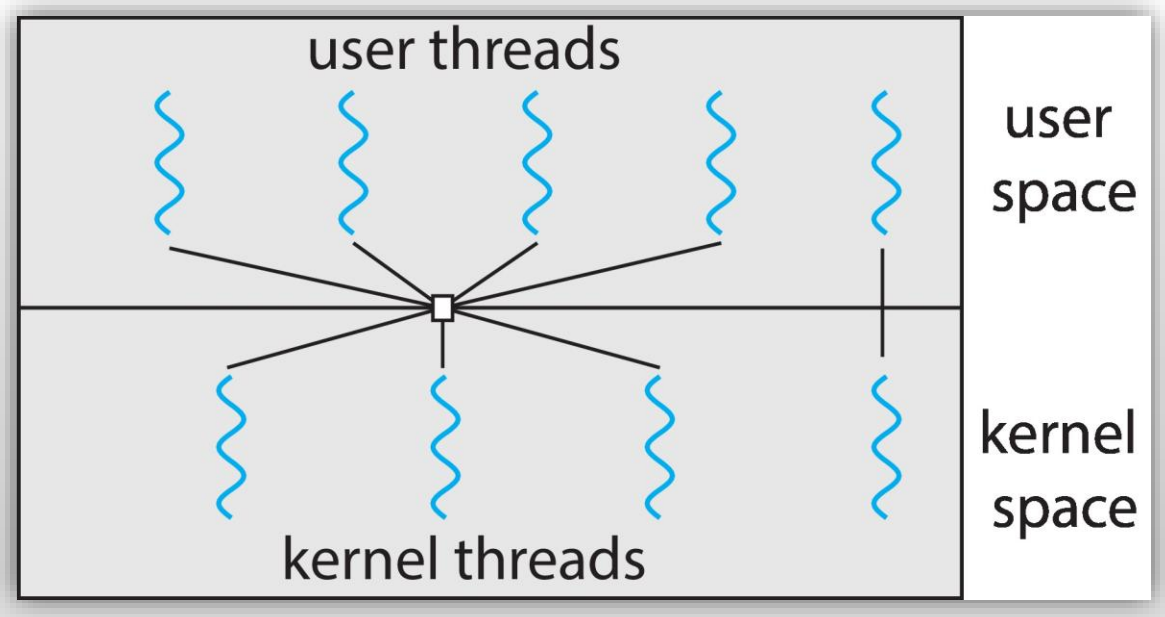
- İş parçacığının sistem çağrısı yürütmesine izin verilmez,
  - Çünkü diğer tüm iş parçacıklarını bloke eder.
- Zarif bir çözüm yok,
  - Sistem çağrılarına izin vermek için sistem kütüphanesi kırılabilir (*hack*).
  - Unix'in bazı sürümlerinde aynı işi yapan fonksiyonlar kullanılabilir.
- İş parçacıkları gönüllü olarak işlemciyi bırakmaz.
  - Kontrolü sisteme geri vermek için periyodik olarak kesmeye uğrar.
  - Maliyetli bir çözüm.

Çekirdek Modu İş Parçacıkları

- Çekirdek, kullanıcı modu ile aynı iş parçacığı tablosunu tutar.
- İş parçacığı bloke olursa, çekirdek başka bir tanesini seçer.
  - Aynı süreçten olması gerekmez!
- Çekirdekte iş parçacıklarını yönetmek çok maliyetli.
- Değerli olan çekirdek alanında yer kaplar.

Hibrit Yaklaşım

- Kullanıcı düzeyi iş parçacıkları, çekirdek düzeyi iş parçacıklarına eşlenir.

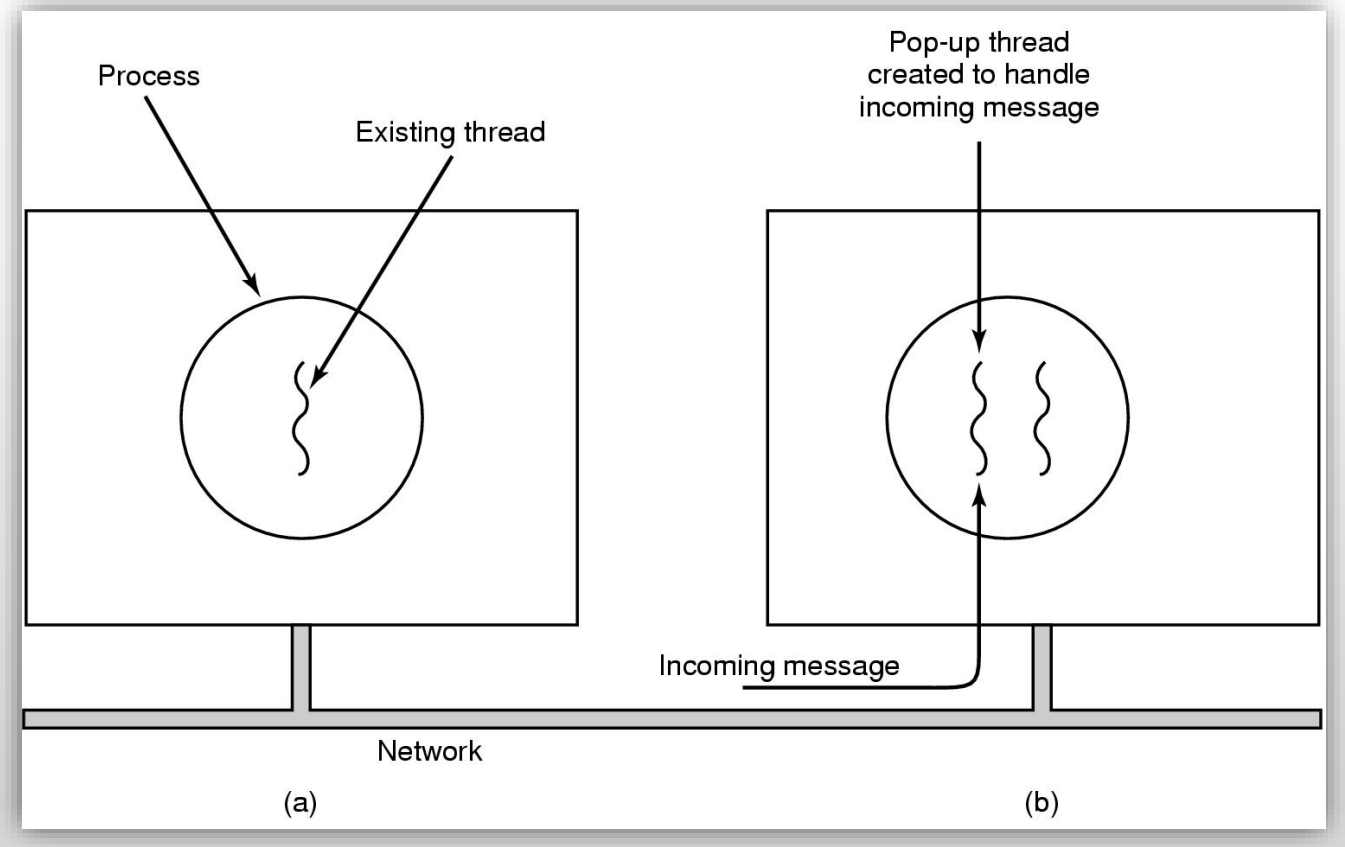


Hibrit Yaklaşım

- Çekirdek, sadece çekirdek iş parçacıklarından haberdardır.
- Kullanıcı düzeyinde iş parçacıkları, çekirdekten bağımsız olarak,
  - oluşturulur, çizelgelenir, ve sonlandırılır.
- Programcı,
  - kaç tane kullanıcı düzeyi ve
  - kaç tane çekirdek düzeyi iş parçacığı kullanacağını belirler.

Açılır (Pop-up) İş Parçacıkları

- Bir mesaj geldiğinde yeni bir iş parçacığı yaratılır.



Açılır İş Parçacıkları (Thread Spawn)

- Sistem, mesaj alma çağrısına bloke olmuş ve
  - mesaj geldikçe mesajı işleyen bir iş parçacığı kullanabilir.
- Her mesaj geldiğinde iş parçacığının geçmişinin geri yüklenmesi gerekir.
- Açılır iş parçacıkları yenidir ve geri yüklenecek verisi yoktur.
- Bu nedenle daha hızlı.

```
// Açılır iş parçacığı tarafından çalıştırılacak fonksiyon
void *threadFunction(void *arg) {
    printf("Merhaba, ben bir açılır iş parçacığıyım!\n");
    return NULL;
}
```

Ana İş Parçacığı Kütüphaneleri

- **POSIX Pthreads:**
  - Unix ve benzeri işletim sistemleri için standart iş parçacığı kütüphanesi.
  - İş parçacığı oluşturma, yönetme ve senkronize etme için API'ler sağlar.
- **Win32:**
  - Windows işletim sistemi için iş parçacığı kütüphanesi.
  - Hafif ve hızlı iş parçacıkları oluşturmayı sağlar.
- **Java:**
  - Thread sınıfı ve Runnable arayüzü ile iş parçacıkları oluşturulur.
  - *Java Virtual Machine (JVM)* tarafından yönetilir.

Diğer İş Parçacığı Kütüphaneleri

- C++ için *Boost.Thread*
- Python için *threading*
- Go için *go-routine*

POSIX kütüphanesi (pthreads)

- Hem kullanıcı düzeyi hem çekirdek düzeyi iş parçacıklarını destekler.
- İş parçacığı oluşturma ve senkronizasyon için bir standart (*IEEE 1003.1c*)
- Tanımlama (*specification*) ve uygulama (*implementation*) değil, bir API.
- İş parçacığı kütüphanesinin nasıl davranması gerektiğini belirtir.
- Uygulama, kütüphanenin geliştirilmesine bağlıdır.
- UNIX işletim sistemlerinde yaygın (*Linux ve Mac OS X*).

POSIX kütüphanesi (pthreads)

- IEEE Unix standart kütüphane çağrıları

İşlev çağrısı	Açıklama
pthread_create	Yeni bir iş parçacığı oluşturur.
pthread_exit	Çağıran iş parçacığını sonlandırır.
pthread_join	Belirli bir iş parçacığının sonlanmasını bekler.
pthread_yield	Başka bir iş parçacığının çalışması için CPU'yu serbest bırakır.
pthread_attr_init	Bir iş parçacığının öznitelik yapısını oluştur ve ilklendirir.
pthread_attr_destroy	Bir iş parçacığının öznitelik yapısını kaldırır.

Üstü Örtülü İş Parçacığı Oluşturma

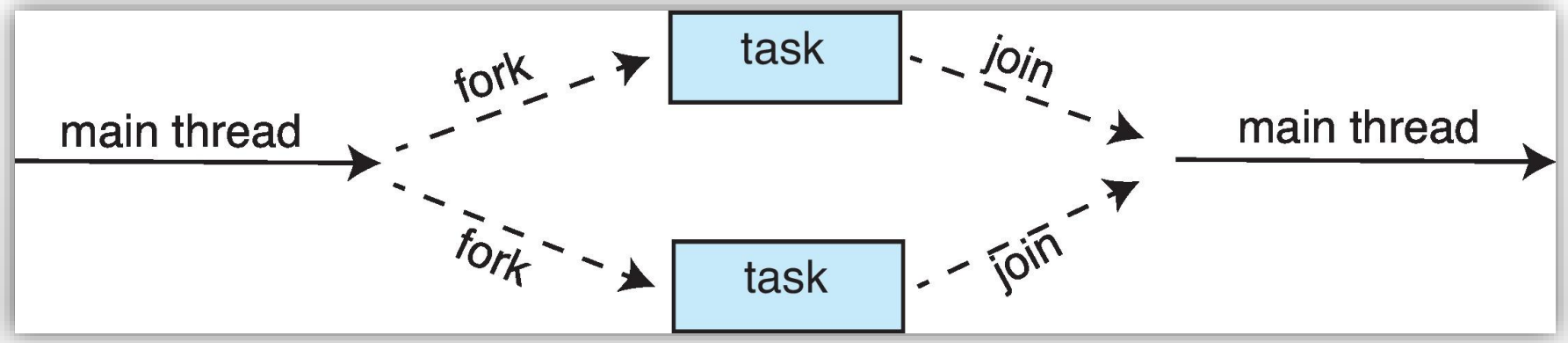
- İş parçacığı sayısı arttığında, programın doğruluğunun kontrolü zorlaşır.
- Çalışma zamanı kütüphaneleri tarafından oluşturulur ve yönetilir.
- Yöntemler
  - İş parçacığı havuzları (*thread pool*),
  - *fork - join,*
  - *OpenMP,*
  - Görev dağıtımı (*grand cental dispatch*),
  - Intel İş Parçacığı Oluşturma Yapı Taşları (*intel threads building blocks*).

İş Parçacığı Havuzu

- Bir havuzda belirli sayıda iş parçacığı tutulur.
- Havuzda mevcut bir iş parçacığıyla bir isteğe hizmet vermek, yeni bir iş parçacığı oluşturmaktan hızlıdır.
- Uygulamadaki iş parçacığı sayısı havuzun boyutu ile sınırlıdır.

fork - join

- .



OpenMP

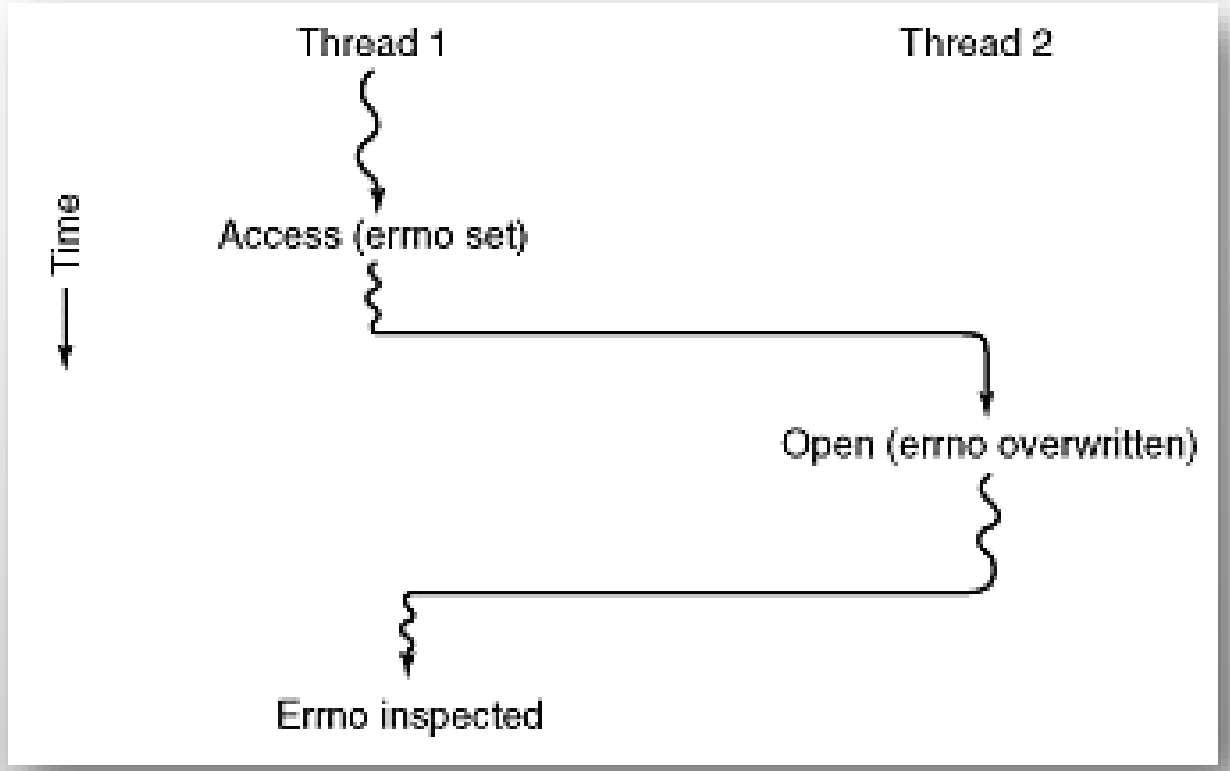
- C, C++, FORTRAN dilleri için bir API.
- Derleyici yönergeleri kümesi. (*compiler directives*)
- Paylaşımlı bellek ortamlarında paralel programlama için destek sağlar.
- Paralel bölgeleri (paralel olarak çalışabilen kod blokları) tanımlar.
- `#pragma omp paralel`
- İşlemci çekirdek sayısı kadar iş parçacığı oluşturur.

Grand Central Dispatch

- macOS ve iOS işletim sistemleri için geliştirilen Apple teknolojisi.
- C, C++ ve Objective-C dillerine, API'ye ve çalışma zamanı kütüphanesine yönelik eklemeler.
- Paralel çalıştırılacak kod blokları tanımlanmasına izin verir.
- İş parçacığı oluşturma işlemlerini yönetir.
- "^{ }" içerisine yazılır.
  - `^{ printf("Ben bir bloğum"); }`
- Kod blokları görev dağıtım kuyruğuna yerleştirilir.
  - Kuyruktan çıkan havuzdaki kullanılabilir iş parçacığına atanır.

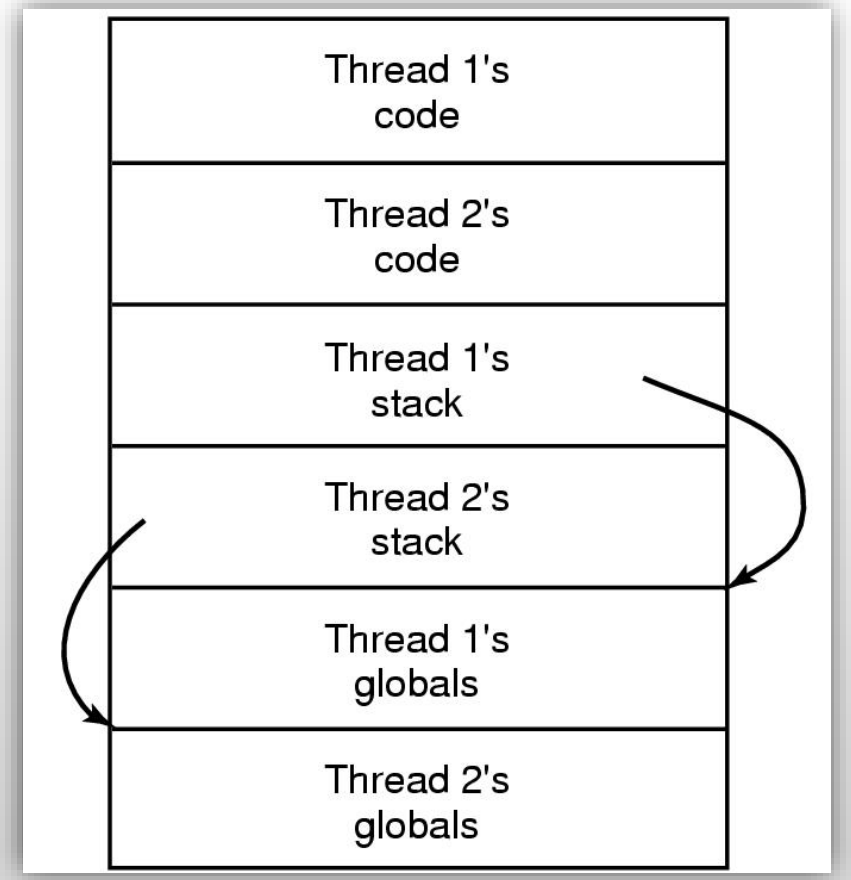
İş Parçacıkları Arasında Çakışma

- Global değişken ile ilgili iş parçacıkları arasında yaşanabilecek çakışma.



Çoklu İş Parçacıklı Programlama

İş parçacıkları kendilerine ait global değişkenlere sahip olabilir.



Problemler

- Yeniden girilmez (*not re-entrant*) kütüphane yordamı.
  - İş parçacığı gelen mesajı bir ara belleğe koyar,
  - Yeni bir iş parçacığı mesajın üzerine yazar.
- Bellekten yer alma programları (*geçici olarak*) tutarsız bir durumda olabilir.
  - Yeni iş parçacığı yanlış işaretçi almış olabilir.
- İş parçacığına özgü sinyalleri uygulamak zor mu?
  - İş parçacıkları kullanıcı düzeyindeyse, çekirdek doğru iş parçacığını adresleyemez.

İş Parçacıklarının Kullanılma Nedeni

- Sistem çağrıları bloke olduğunda paralelliği etkinleştirir (*web sunucusu*).
- İş parçacıkları oluşturmak ve yok etmek, süreçlerden daha hızlı.
- Çok çekirdekli sistemler için gerekli.
- Uygulaması kolay bir programlama modeli.

İş Parçacıklarının Avantajları

- Kullanıcı duyarlılığı:
  - Bir iş parçacığı bloke olduğunda, diğeri kullanıcı G/Ç'sini işleyebilir.
- Kaynak paylaşımı:
  - Bellek (*adres alanı*), açık dosyalar, soketler.
- Hız:
  - İş parçacığı oluşturma 30 kat, bağlam anahtarlama 5 kat daha hızlıdır.
- Donanım paralelliğinden yararlanma:
  - Ağır (*heavy*) süreçler, çoklu işlemcili mimarilerden faydalanabilir.

İş Parçacıklarının Dezavantajları

- Senkronizasyon
  - Paylaşımlı bellek ve değişkenlere erişim kontrol edilmelidir.
  - Program kodunda karmaşıklık ve hataya neden olabilir.
  - Yarış koşullarından ve kilitlenmelerden kaçınmak gerekir.
- Bağımlılık
  - Ağır Ağırlık İşlemde (HWP) iş parçacıkları bağımsız değildir.
  - Adres uzayı paylaşıldığından bellek koruması yoktur.
  - Her iş parçacığının yığınının bellekte ayrı yerde olması amaçlanır.
  - Bir iş parçacığının hatası nedeniyle başka bir iş parçacığının yığınının üzerine yazma yapılabilir.

Pthreads Mutex - Producer

```
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */
void *producer(void *ptr) { /* produce data */
  for (int i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
    while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
    buffer = i; /* put item in buffer*/
    pthread_cond_signal(&condc); /* wake up consumer */
    pthread_mutex_unlock(&the_mutex); /* release access to buffer */
  }
  pthread_exit(0);
}
```

Pthreads Mutex - Consumer

```
void *consumer(void *ptr) { /* consume data */
  for (int i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
    while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
    buffer = 0; /* take item out of buffer */
    pthread_cond_signal(&condp); /* wake up producer */
    pthread_mutex_unlock(&the_mutex); /* release access to buffer */
  }
  pthread_exit(0);
}
```

SON