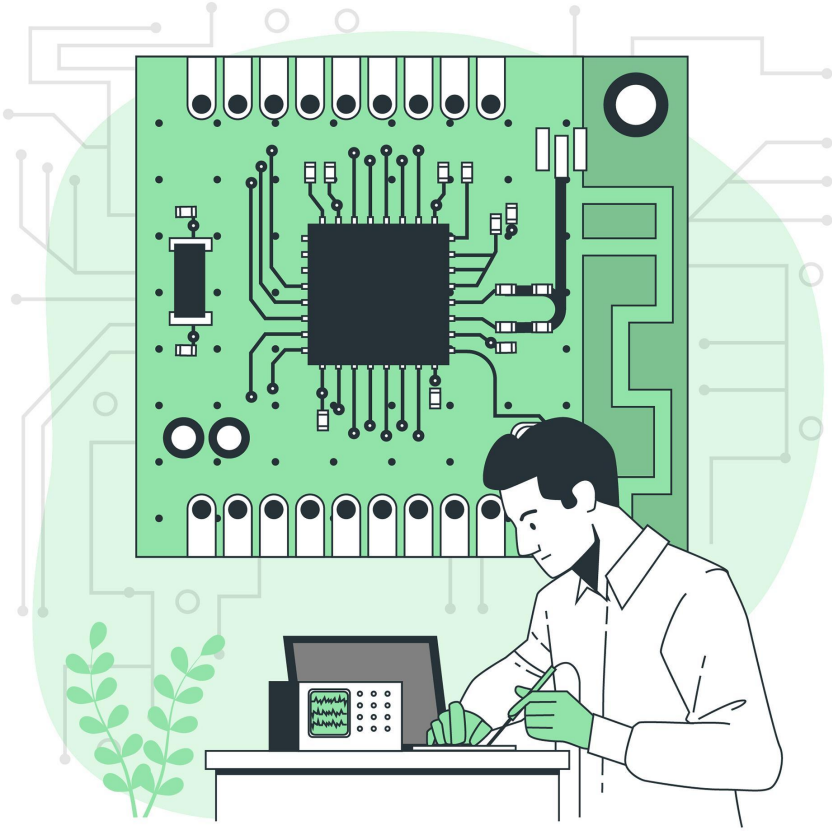
02
Süreçler
(Processes)



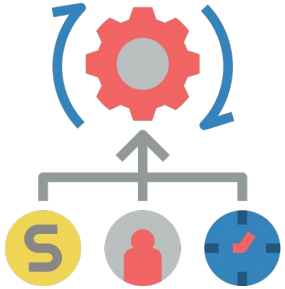
Süreç Nedir?

1. **PROGRAM**
   a. **Diskte bulunan kod**
2. **ÇALIŞAN PROGRAM = SÜREÇ**
   a. **Programın çalışma anındaki aktif durumunu**
   b. **Sürecin çalışma hızı sabit değildir.**
   c. **Her sürecin kendi CPU'su vardır.**
3. **SOYUTLAMA**
   a. **Süreç, birçok detayı (bellek yönetimi, I/O vb.) gizler.**

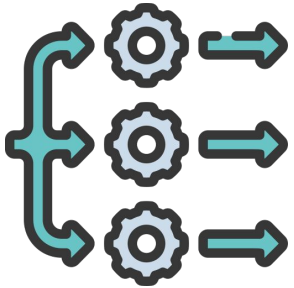
Süreç Özellikleri

İletişim

Kaynak İzolasyonu



Çoklu Görev



Process Oluşturma Süreci

Fork

Çağrıldığı sürecin (ebeveyn sürecin) kopyasını (çocuk süreç) oluşturur.

Ebeveyn ve çocuk süreçleri ayrı bellek alanlarına sahiptir.

Bir programı çalıştırmadan önce süreç kendi kopyasını oluşturur

Process Oluşturma Süreci

Execve

Mevcut sürecin bellek içeriğini yeni bir programla değiştirir.

Çağrılan sürecin belleği yeni programın kodu ve verisiyle doldurulur.

Süreç ID'si değişmez, ancak farklı bir programı çalıştırmaya başlar.

"fork" ile bir süreç kopyalandıktan sonra, çocuk sürecin farklı bir programı çalıştırması için kullanılır.

Process Oluşturma Süreci

CreateProcess

Hem süreci oluşturur hem de belirtilen programı yeni süreçte çalıştırır. UNIX'teki "fork" ve "execve" kombinasyonunun birleşik bir versiyonu gibidir.

Process ve Program Farkı

CPU: Yemeği yapan aşçı

Program: Yemek tarifi

Süreç: Aşçının tarifi takip ederek yemeği yapma eylemi

Process ve Program Farkı

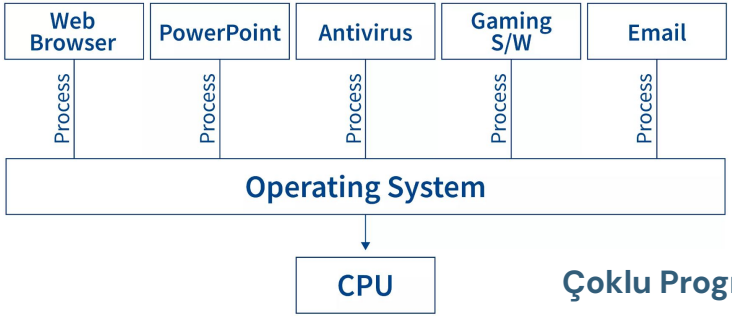
Kesme:

Aşçı yemek yaparken çocuğunu arı sokarsa, yemek yapmayı bir süreliğine bırakır ve çocuğa yardım etmeye başlar.

Bu, CPU'nun bir süreçten (yemek yapma) diğerine (ilk yardım) geçiş yapmasına benzer.

Acil durum çözüldüğünde, aşçı yemeğe kaldığı yerden devam eder.

Çoklu Programlama ve Çekirdek



Çoklu Programlama (Multiprogramming):

Tek bir CPU'nun, birden fazla işlemi, eşzamanlı olarak yürütmek için işlemler arasında hızla geçiş yapması yöntemidir.

İşletim sistemi, çalışan süreçler arasında düzenli aralıklarla geçiş yaparak, CPU'nun sürekli olarak kullanılmasını sağlar.

Sistem kaynaklarını daha verimli kullanır.

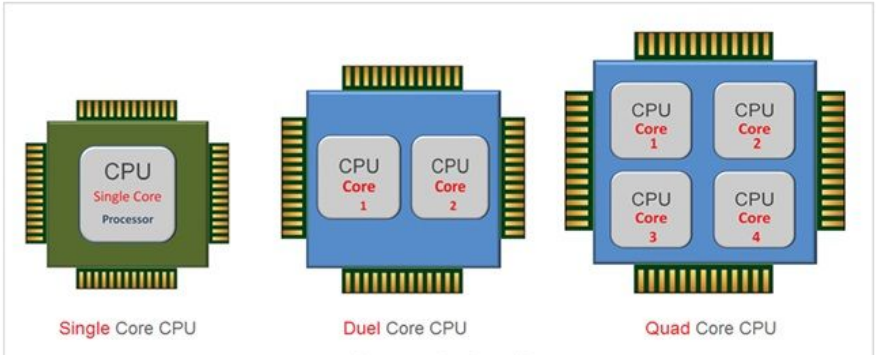
Çoklu Programlama ve Çekirdek

Çok Çekirdekli İşlemciler (Multicore Processors):

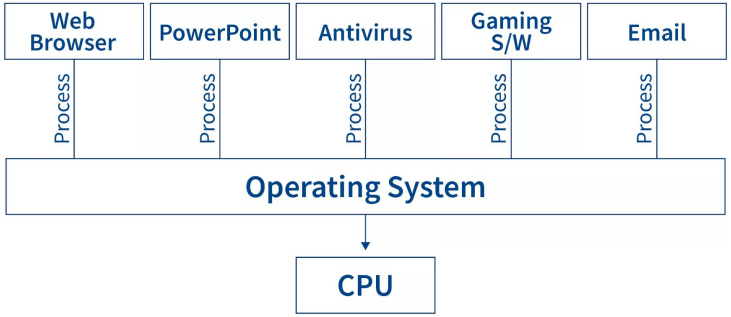
Tek bir işlemci çipinde birden fazla işlemci çekirdeği bulunduran işlemcilerdir.

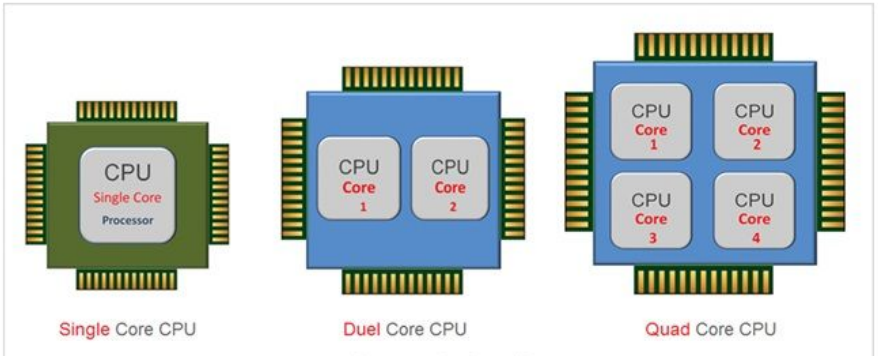
Her bir çekirdek, bağımsız bir işlemci olarak hareket edebilir.

Performansı artırma ve enerji verimliliğini optimize etme gibi avantajlara sahiptir.



Çoklu Programlama ve Çekirdek





Bilgisayarın aynı anda birçok işlemi gerçekte ve etkili bir şekilde yürütmesine olanak tanır.

Yüksek performanslı hesaplama gereksinimlerini karşılamak için önemlidir.

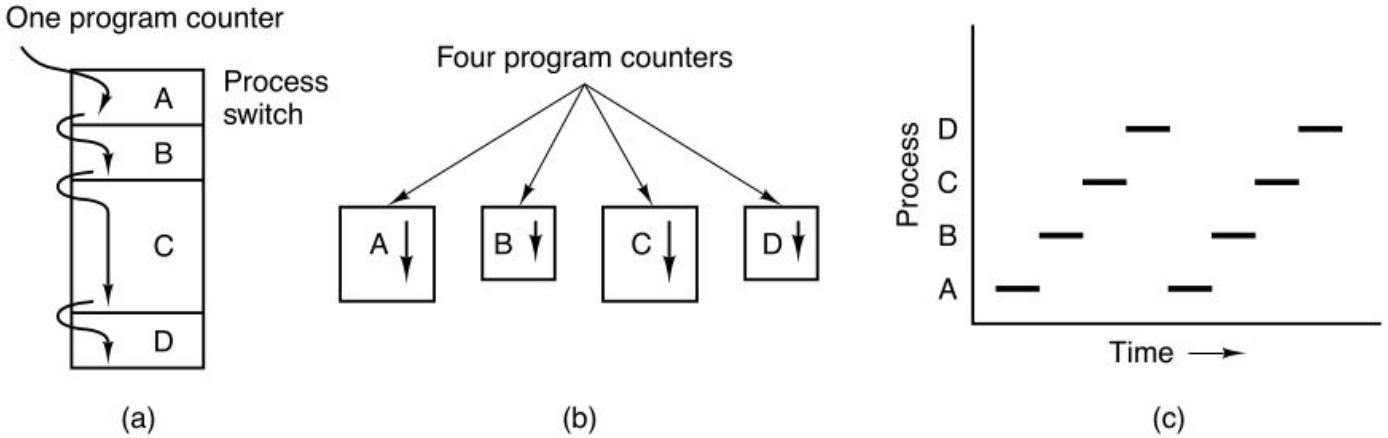
Özellikle veri bilimi, yapay zeka ve grafik tasarım gibi alanlarda sıkça kullanılır.

Çoklu Programlama ve Çekirdek

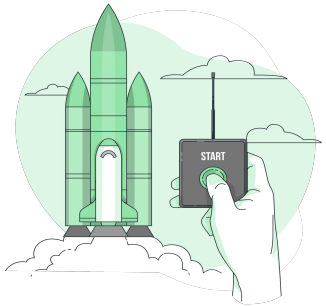
1 Çekirdek

4 Çekirdek

Process Modeli



Process Oluşturma

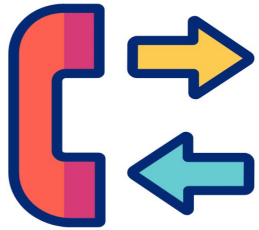


Sistem Başlangıcı

Bilgisayarınızı başlattığınızda, işletim sistemi otomatik olarak bir dizi temel süreç başlatır.

Bu süreçler, sistem kaynaklarını yönetmek, kullanıcı girişlerini beklemek ve arka planda çalışan hizmetleri başlatmak için gereklidir.

Process Oluşturma



Sistem Çağrısı

Çalışan bir program, başka bir görevi yerine getirmek veya kaynaklara erişmek için işletim sistemine talepte bulunabilir.

Bu, genellikle bir programın başka bir programı çalıştırması veya bir hizmeti başlatması gerektiğinde gerçekleşir.

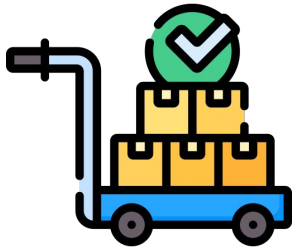
Örneğin, bir metin düzenleyici, yazdırma işlemi için işletim sistemine bir talepte bulunabilir.

Process Oluşturma

Kullanıcı Talebi

Kullanıcılar doğrudan bir programı başlatarak veya bir simgeye tıklayarak yeni bir süreç başlatabilir.

Process Oluşturma

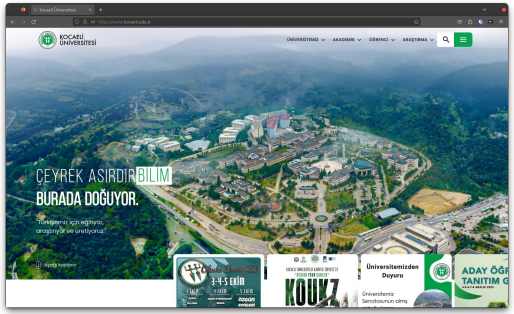


Toplu Çağrı

Özellikle sunucu sistemlerinde, belirli zamanlarda otomatik olarak çalıştırılan bir dizi görev bulunabilir.

Bu toplu işler, genellikle büyük veri işleme görevleri veya düzenli bakım rutinleri için kullanılır.

Ön Plan ve Arka Plan Süreçleri



Ön Plan Süreçleri:

Kullanıcılarla doğrudan etkileşimde bulunan ve genellikle kullanıcının doğrudan taleplerine yanıt olarak çalışan süreçlerdir.

Bu süreçler, kullanıcının eylemlerine doğrudan yanıt verir.

Kullanıcı, bu süreçleri sonlandırabilir veya duraklatabilir.

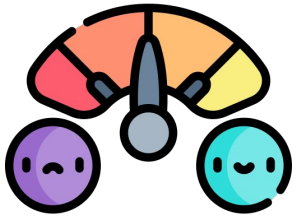
Ön Plan ve Arka Plan Süreçleri

Arka Plan Süreçleri (Daemon'lar):

Kullanıcı etkileşimi olmadan, genellikle belirli görevleri otomatik olarak yerine getiren süreçlerdir.

Bu süreçler, genellikle sistem başlatıldığında otomatik olarak başlar ve kullanıcının doğrudan bilgisi olmadan çalışır.

Process Sonlandırma



Normal Çıkış

Normal Çıkış (Gönüllü):

Süreç, atanmış görevini başarıyla tamamladığında kendi isteğiyle sonlandırılır.

UNIX'te bu sonlandırma "exit" sistem çağrısı,

Windows'ta ise "ExitProcess" fonksiyonu ile gerçekleştirilir.

Process Sonlandırma

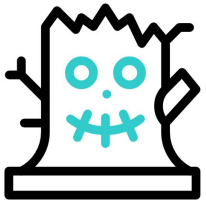
Hata Nedeniyle

Hata Nedeniyle Çıkış (Gönüllü):

Süreç, beklenmedik bir durumla karşılaştığında veya belirli bir hata durumunda sonlandırılabilir.

Örneğin, bir derleyiciye mevcut olmayan bir dosyanın derlenmesi söylendiğinde,

Process Sonlandırma



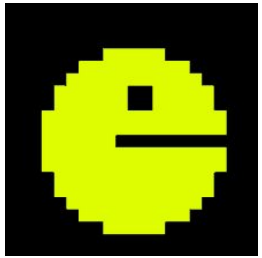
Ölümcül Hata

Ölümcül Hata (Gönüllü Olmayan):

Süreç, programın içerdiği bir hata nedeniyle sonlandırılabilir.

Geçersiz bir komutun çalıştırılması, tanımlanmamış bir belleğe erişim, sıfıra bölme

Process Sonlandırma



Başka Bir Süreç

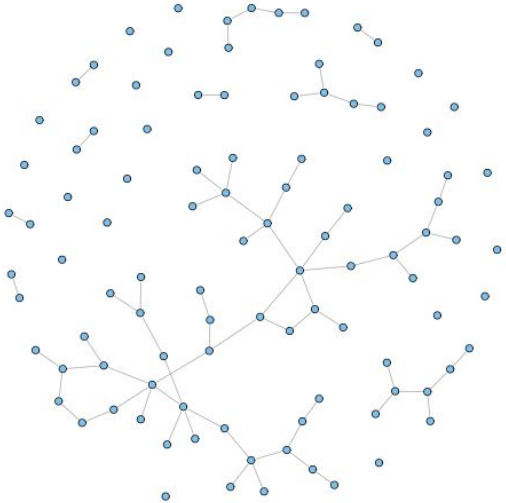
Başka Bir Süreç Tarafından Sonlandırma (Gönüllü Olmayan):

Bir süreç, başka bir süreci sonlandırma yetkisine sahip olabilir.

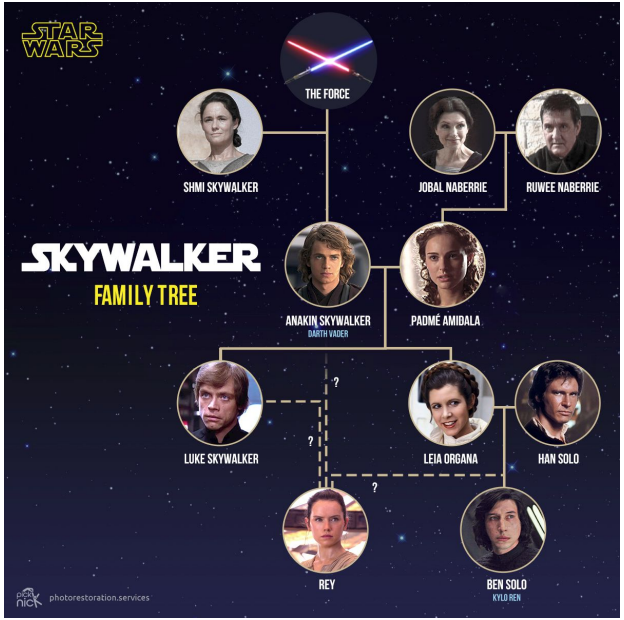
UNIX'te bu işlem "kill" sistem çağrısıyla,

Windows'ta ise "TerminateProcess" fonksiyonu ile gerçekleştirilir.

Process Hiyerarşisi

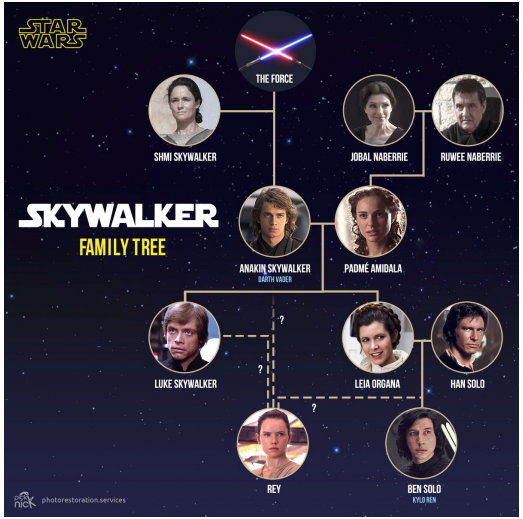


Windows



Unix

Process Hiyerarşisi



Unix

UNIX'te Süreç Hiyerarşisi:

UNIX'te, bir süreç, çocuk süreçleri ve bu çocuk süreçlerin oluşturduğu diğer süreçlerle birlikte bir süreç grubu oluşturur.

Klavyeden bir sinyal gönderildiğinde (örn. CTRL-C tuş kombinasyonu ile), bu sinyal şu anda klavye ile ilişkilendirilmiş süreç grubundaki tüm üyelere gönderilir.

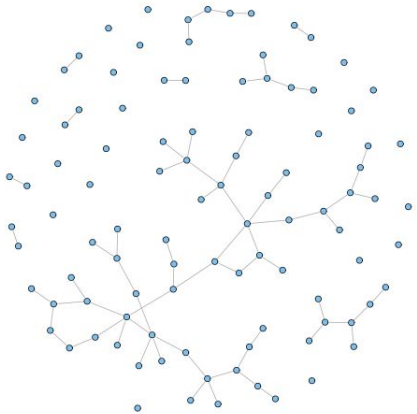
Bilgisayar başlatıldığında, "init" adında özel bir süreç çalıştırılır.

Bu süreç, her terminal için yeni bir süreç oluşturur.

Bu süreçler, kullanıcı giriş yaptığında bir kabuk (shell) çalıştırır. B

Bu kabuk, kullanıcının komutlarına yanıt olarak daha fazla süreç başlatabilir.

Process Hiyerarşisi



Windows

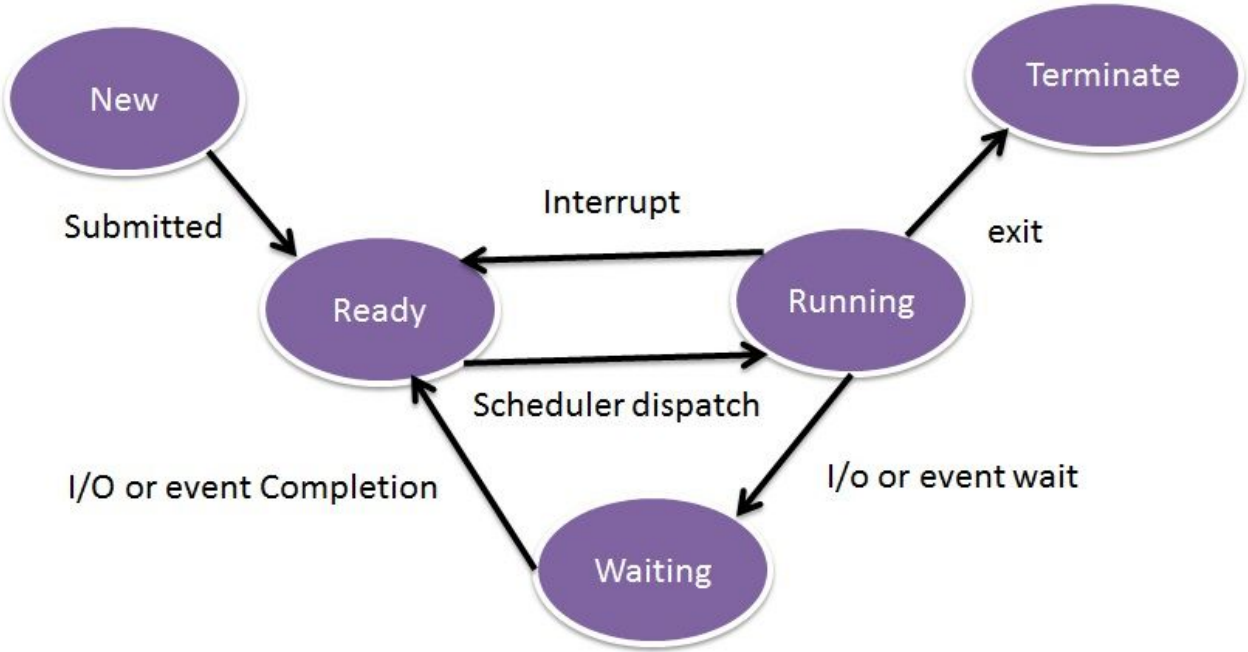
Windows'ta Süreç Hiyerarşisi:

Windows'ta süreç hiyerarşisi kavramı bulunmamaktadır. Tüm süreçler eşittir ve birbirinden bağımsızdır.

Ancak, bir süreç oluşturulduğunda, ebeveyn sürece, çocuğu kontrol etmek için kullanabileceği özel bir belirteç (token) verilir.

Ancak, ebeveyn bu belirteci başka bir sürece aktarabilir, bu da hiyerarşik yapıyı geçersiz kılar.

Process Durumları



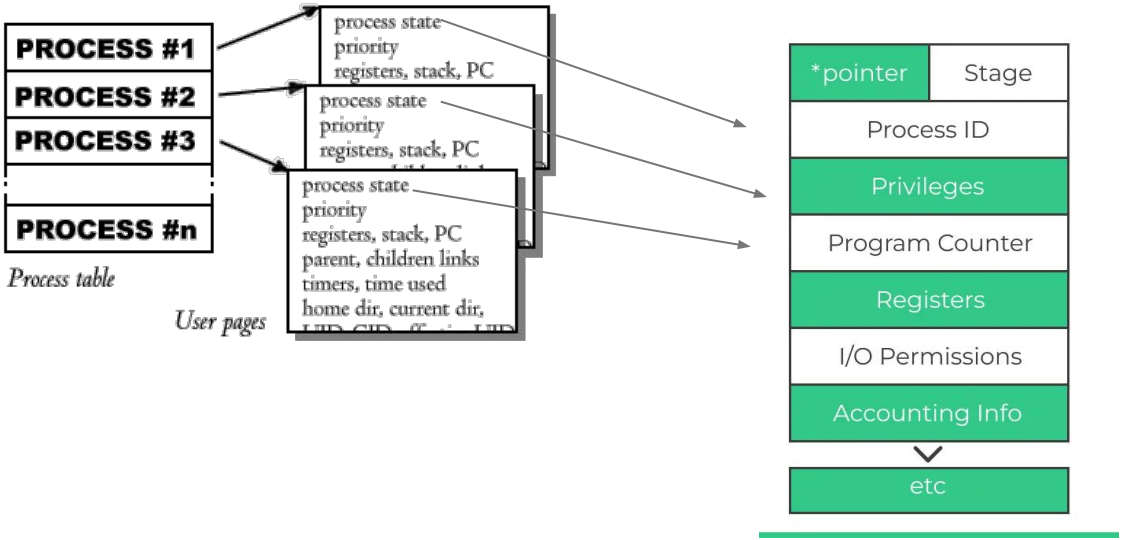
Process Durumları

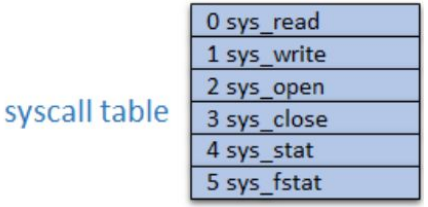
Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Process Durumları

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

İşletim Sistemi Veri Yapıları





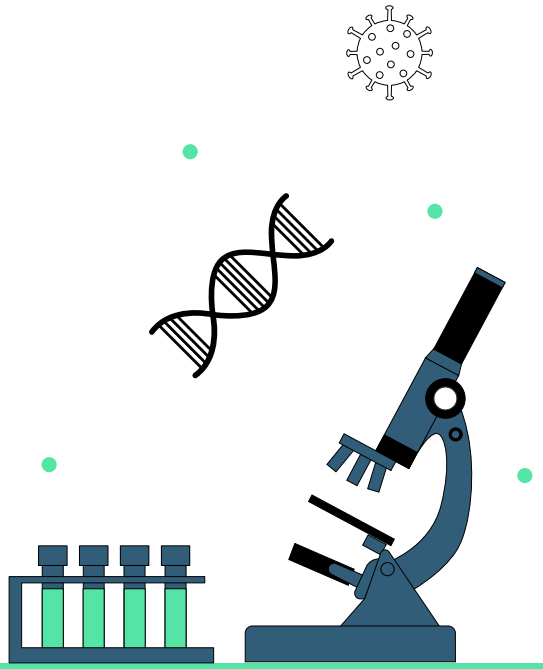
Örnek Bir Süreç (xv6)

xv6 process header

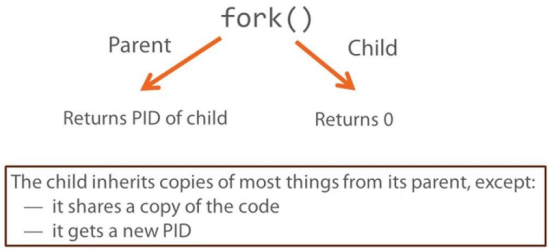
```
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

02

Process API



Fork ()

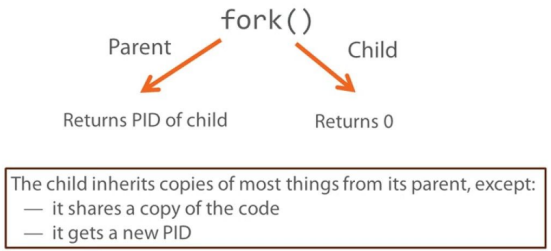


Yeni bir süreç oluşturmak için kullanılan bir sistem çağrısıdır.

Kopyalama:

`fork()` çağrısını yapan ebeveyn süreç çocuk süreç olarak kopyalanır.

Fork ()



Yeni bir süreç oluşturmak için kullanılan bir sistem çağrısıdır.

Kopyalama:

`fork()` çağrısını yapan ebeveyn süreç çocuk süreç olarak kopyalanır.

Dönüş Değerleri:

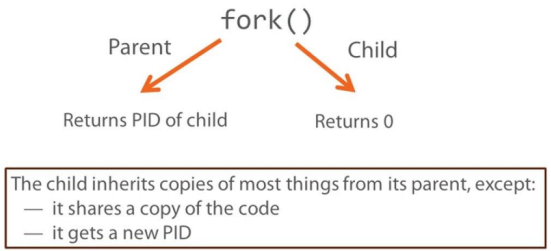
`fork()` fonksiyonunun geri dönüş değeri;

Çocuk süreç için sıfırdır.

Ebeveyn süreç için, çocuk işlemin süreç kimliği (PID)dir.

Eğer `fork()` başarısız olursa, sadece ebeveyn sürece negatif bir değer döner.

Fork ()



Yeni bir süreç oluşturmak için kullanılan bir sistem çağrısıdır.

Kopyalama:

`fork()` çağrısını yapan ebeveyn süreç çocuk süreç olarak kopyalanır.

Dönüş Değerleri:

`fork()` fonksiyonunun geri dönüş değeri;

Çocuk süreç için sıfırdır.

Ebeveyn süreç için, çocuk işlemin süreç kimliği (PID)dir.

Eğer `fork()` başarısız olursa, sadece ebeveyn sürece negatif bir değer döner.

Çocuk süreç, `exec()` fonksiyonunu çağırarak başka bir programı çalıştırır.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
    } else{
        // parent goes down this path(orginal process)
        printf("hello, I am parent of %d (pid:%d)\n",rc,(int) getpid());

    }
    return 0;

}
```

ebeveyn sürecin (parent process) PID'sini (process ID) ekrana yazdırır.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
    } else{
        // parent goes down this path(orginal process)
        printf("hello, I am parent of %d (pid:%d)\n",rc,(int) getpid());

    }
    return 0;

}
```

Ebeveyn sürecin bir **kopyasını** (çocuk süreç - child process) oluşturur ve çalıştırır.

Ebeveyn süreç (parent process): `fork()` çağrısı başarılı olursa, yeni oluşturulan çocuk sürecin PID'sini geri döner.

Çocuk süreç (child process): `fork()` çağrısı, çocuk süreç içinde `0` döndürür.

Başarısızlık durumu: Eğer `fork()` başarısız olursa, `-1` döndürülür.

Bu durumda, `rc` değişkeni **fork()** çağrısından dönen değeri tutar.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n",getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
    } else{
        // parent goes down this path(orginal process)
        printf("hello, I am parent of %d (pid:%d)\n",rc,(int) getpid());

    }
    return 0;

}
```

`fork()` çağrısının başarısız olması durumunda çalışır. Yani, yeni bir süreç oluşturulamadığında bu hata mesajını verir.

`fprintf(stderr, "fork failed\n");`: Hata mesajını standart hata akışına (stderr) yazar.

`exit(1);`: Programı hata kodu ile sonlandırır.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
    } else{
        // parent goes down this path(orginal process)
        printf("hello, I am parent of %d (pid:%d)\n",rc,(int) getpid());

    }
    return 0;

}
```

çocuk sürecin çalıştığı durumdur. Eğer `rc == 0` ise, bu kod çocuk süreçte çalışacaktır.

`printf("hello, I am child (pid:%d)\n", (int) getpid());`:
Çocuk sürecin PID'sini ekrana yazdırır.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
    } else{
        // parent goes down this path(orginal process)
        printf("hello, I am parent of %d (pid:%d)\n",rc,(int) getpid());

    }
    return 0;

}
```

Eğer `rc > 0` ise, bu blok **ebeveyn sürecin** çalıştığı kısımdır. Bu durumda `rc`, çocuk sürecin PID'sini içerir.

`printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());:`

`getpid()`: Ebeveyn sürecin kendi PID'si.

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }









   return 0;
}
```

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   int rc2 = fork();              // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }




   return 0;
}
```

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   int rc2 = fork();              // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

   printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), rc1, rc2);

   return 0;
}
```

P0;

pid=262589 ppid=262565 | rc1=262593 rc2=262594

ppid=262565: P0'ın ebeveyni (terminal vb.)

rc1=262593 (ilk fork'ta doğan C1'in PID'si)

rc2=262594 (ikinci fork'ta doğan P0C2'nin PID'si)

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   int rc2 = fork();              // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

   printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), rc1, rc2);

   return 0;
}
```

P0C2;

pid=262594 ppid=262589 | rc1=262593 rc2=0

rc1, P0'da ilk fork'tan beri tutulur

rc2=0 ikinci fork'un çocuk tarafı

ppid=262589: ebeveyni P0

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   int rc2 = fork();              // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

   printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), rc1, rc2);

   return 0;
}
```

C1;

pid=262593 ppid=1651 | rc1=0 rc2=262595

rc1=0 ilk fork'un çocuk tarafı

rc2=262595 ikinci fork'un çocuğu C1C2'nin PID'si.

ppid=1651 Parent (P0) sonlandı (wait yok) Yetim Süreç

Fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
   printf("root pid=%d\n", (int)getpid());

   int rc1 = fork();              // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   int rc2 = fork();              // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

   printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), rc1, rc2);

   return 0;
}
```

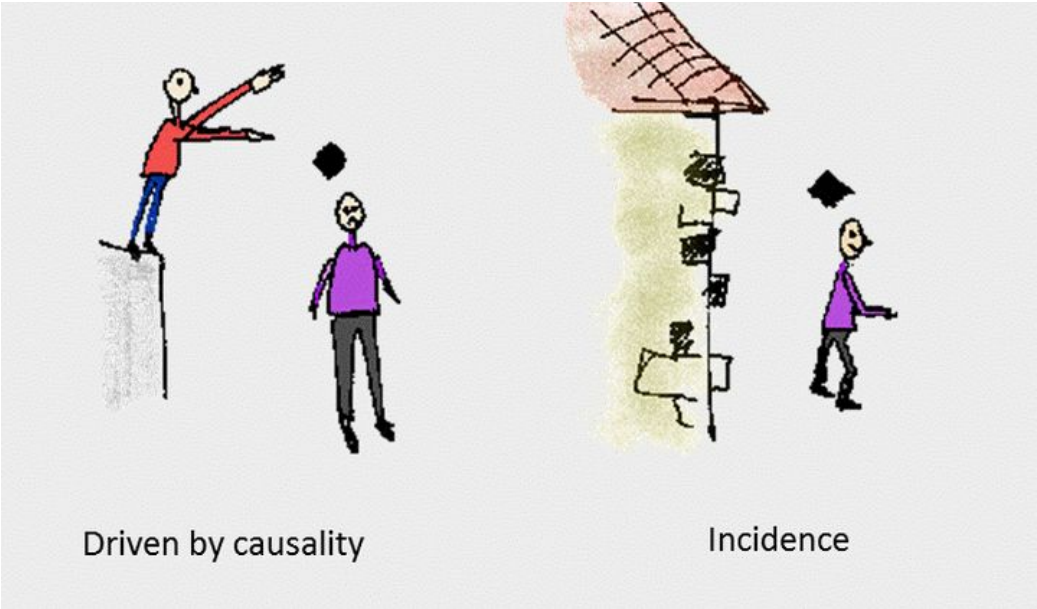
```
C1C2;
pid=262595 ppid=1651 | rc1=0 rc2=0
```

`rc1=0` P0'da ilk fork'tan beri tutulur

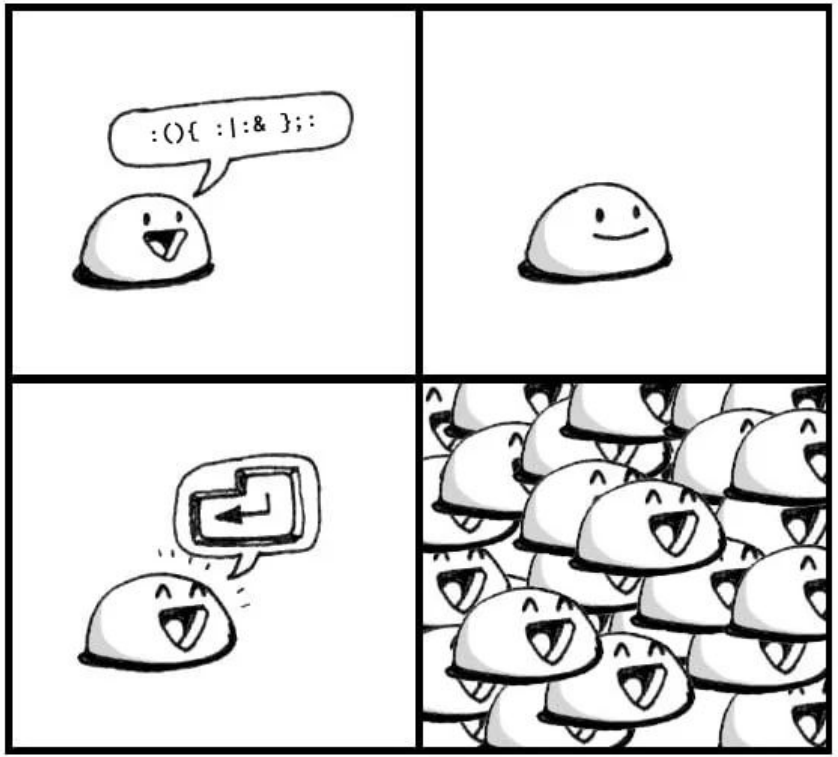
`rc2=0` ikici fork'un fork'unun çocuk tarafı

`ppid=1651` Parent (P0) sonlandı (`wait` yok) Yetim Süreç

Deterministik / Stokastik

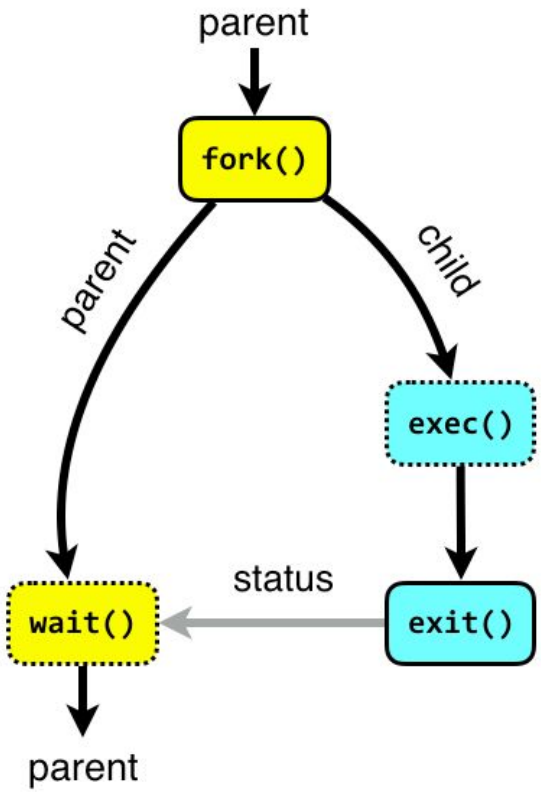


Fork Bomb 💣




```
:(){ :|:& };:
```

wait()



wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

ebeveyn sürecin (parent process) PID'sini (process ID) ekrana yazdırır.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

Ebeveyn sürecin bir **kopyasını** (çocuk süreç - child process) oluşturur ve çalıştırır.

Ebeveyn süreç (parent process): `fork()` çağrısı başarılı olursa, yeni oluşturulan çocuk sürecin PID'sini geri döner.

Çocuk süreç (child process): `fork()` çağrısı, çocuk süreç içinde `0` döndürür.

Başarısızlık durumu: Eğer `fork()` başarısız olursa, `-1` döndürülür.

Bu durumda, `rc` değişkeni **fork()** çağrısından dönen değeri tutar.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n",(int) getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

Eğer `rc` değeri `-1` olursa, bu, `fork()`'un başarısız olduğu anlamına gelir.

`fprintf(stderr, "fork failed\n");`: Standart hata akışına hata mesajı yazar.

`exit(1);`: Programı hatalı şekilde sonlandırır.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

çocuk sürecin çalıştığı durumdur. Eğer `rc == 0` ise, bu kod çocuk süreçte çalışacaktır.

`printf("hello, I am child (pid:%d)\n", (int) getpid());`:
Çocuk sürecin PID'sini ekrana yazdırır.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

`sleep(1);`: Çocuk süreç burada 1 saniye boyunca bekler.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new proce
        printf("hello, I  ild (pid:%d)\n",(int) getpid());
        sleep(1);
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;

}
```

Eğer `rc > 0` ise, bu blok **ebeveyn sürecin** çalıştığı kısımdır.

Bu durumda `rc`, çocuk sürecin PID'sini içerir.

`wait()` fonksiyonu, Ebeveyn süreç, çocuk süreç bitene kadar bekler ve bu fonksiyon çocuk sürecin PID'sini döner.

`wait(NULL)`, çocuk sürecin neden sonlandığını merak etmediğimiz için `NULL` parametresiyle çağrılır.

`printf("hello, I am parent of %d (wc:%d)(pid:%d)\n", rc, wc, (int)getpid());:`

- `rc`: Çocuk sürecin PID'si.
- `wc`: `wait()` : sonlanan çocuk sürecin PID'si.
- `getpid()`: Ebeveyn sürecin kendi PID'si.

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {

   printf("root pid=%d\n", (int)getpid());

   pid_t rc1 = fork();                  // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   pid_t rc2 = fork();                  // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

    printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
           (int)getpid(), (int)getppid(), (int)rc1, (int)rc2);

   // ----- Senkronizasyon: yalnıza KENDİ çocuklarını bekle -----
    // Orijinal ebeveyn: rc1>0 ve rc2>0

   return 0;
}
```

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {

   printf("root pid=%d\n", (int)getpid());

   pid_t rc1 = fork();                  // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   pid_t rc2 = fork();                  // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

    printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
           (int)getpid(), (int)getppid(), (int)rc1, (int)rc2);

   // ----- Senkronizasyon: yalnıza KENDİ çocuklarını bekle -----
    // Orijinal ebeveyn: rc1>0 ve rc2>0

    if (rc1 > 0 && rc2 > 0) {
        int st;
        if (waitpid(rc1, &st, 0) == -1) perror("waitpid rc1");
        if (waitpid(rc2, &st, 0) == -1) perror("waitpid rc2");
    }

    return 0;
}
```

wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {

   printf("root pid=%d\n", (int)getpid());

   pid_t rc1 = fork();                  // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   pid_t rc2 = fork();                  // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

    printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), (int)rc1, (int)rc2);

  // ----- Senkronizasyon: yalnıza KENDİ çocuklarını bekle -----
   // Orijinal ebeveyn: rc1>0 ve rc2>0

   if (rc1 > 0 && rc2 > 0) {
       int st;
       if (waitpid(rc1, &st, 0) == -1) perror("waitpid rc1");
       if (waitpid(rc2, &st, 0) == -1) perror("waitpid rc2");
   }

   // C1 (ilk fork'un çocuğu) aynı zamanda ikinci fork'ta ebeveyn: rc1==0 ve
rc2>0
   else if (rc1 == 0 && rc2 > 0) {
       int st;
       if (waitpid(rc2, &st, 0) == -1) perror("waitpid rc2");
   }
   return 0;
}
```

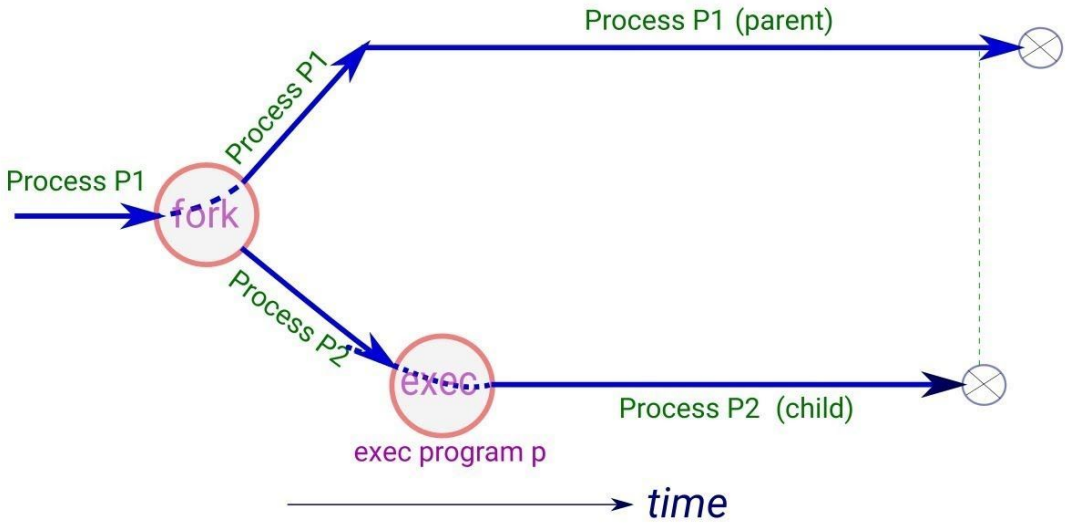
wait()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {

   printf("root pid=%d\n", (int)getpid());

   pid_t rc1 = fork();                  // 1. dallanma
   if (rc1 < 0) { perror("fork"); exit(1); }

   pid_t rc2 = fork();                  // 2. dallanma
   if (rc2 < 0) { perror("fork"); exit(1); }

    printf("pid=%d ppid=%d | rc1=%d rc2=%d\n",
          (int)getpid(), (int)getppid(), (int)rc1, (int)rc2);
```

```
root pid=16216
pid=16216 ppid=16184 | rc1=16227 rc2=16228
pid=16228 ppid=16216 | rc1=16227 rc2=0
pid=16227 ppid=16216 | rc1=0 rc2=16229
pid=16229 ppid=16227 | rc1=0 rc2=0
```

```c
  // ----- Senkronizasyon: yalnıza KENDİ çocuklarını bekle -----
   // Orijinal ebeveyn: rc1>0 ve rc2>0

   if (rc1 > 0 && rc2 > 0) {
       int st;
       if (waitpid(rc1, &st, 0) == -1) perror("waitpid rc1");
       if (waitpid(rc2, &st, 0) == -1) perror("waitpid rc2");
   }

   // C1 (ilk fork'un çocuğu) aynı zamanda ikinci fork'ta ebeveyn: rc1==0 ve
rc2>0
   else if (rc1 == 0 && rc2 > 0) {
       int st;
       if (waitpid(rc2, &st, 0) == -1) perror("waitpid rc2");
   }
   return 0;
}
```

exec()


Process P1 (parent)
Process P1
Process P1
fork
Process P2
exec
Process P2 (child)
exec program p
time

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

ebeveyn sürecin (parent process) PID'sini (process ID) ekrana yazdırır.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n",(int) getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

Ebeveyn sürecin bir **kopyasını** (çocuk süreç - child process) oluşturur ve çalıştırır.

Ebeveyn süreç (parent process): `fork()` çağrısı başarılı olursa, yeni oluşturulan çocuk sürecin PID'sini geri döner.

Çocuk süreç (child process): `fork()` çağrısı, çocuk süreç içinde `0` döndürür.

Başarısızlık durumu: Eğer `fork()` başarısız olursa, `-1` döndürülür.

Bu durumda, `rc` değişkeni **fork()** çağrısından dönen değeri tutar.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

`fork()` çağrısının başarısız olması durumunda çalışır. Yani, yeni bir süreç oluşturulamadığında bu hata mesajını verir.

`fprintf(stderr, "fork failed\n");`: Hata mesajını standart hata akışına (stderr) yazar.

`exit(1);`: Programı hata kodu ile sonlandırır.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

çocuk sürecin çalıştığı durumdur. Eğer `rc == 0` ise, bu kod çocuk süreçte çalışacaktır.

`printf("hello, I am child (pid:%d)\n", (int) getpid());`: Çocuk sürecin PID'sini ekrana yazdırır.

Çocuk süreçte bir `exec()` sistemi çağrısı ile yeni bir program başlatılır.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc" (world count)
        myargs[1] = strdup("p3.c"); // argument:file to count
        myargs[2]=NULL; // marks end of array
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

Yeni bir programı çalıştırmak için bir **argüman dizisi** tanımlar. Bu dizi, `execvp()` tarafından kullanılacaktır.

`myargs[0] = strdup("wc");`: `wc` (word count - kelime sayma) programının adını dizinin ilk elemanı olarak yerleştirir. `strdup()` ile `"wc"` stringi heap belleğe kopyalanır.

`myargs[1] = strdup("p3.c");`: `wc` programına argüman olarak verilen dosya adını (`"p3.c"`) tanımlar. Bu, `wc` programının işlem yapacağı dosyadır.

`myargs[2] = NULL;`: Bu, argüman listesinin sonunu belirtir. `exec()` fonksiyonları, NULL ile sonlandırılmış bir argüman listesi bekler.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpid());
        char *myargs[3];
        myargs[0] =strdup("wc"); // program: "wc"          count)
        myargs[1] = strdup("p3.c"); // argume      to count
        myargs[2]=NULL; // marks end of      ray
        execvp(myargs[0],myargs); // run word count
        printf("this shouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

execvp sistemi çağrısı ile `wc` programı çalıştırılır. `exec()` ailesi fonksiyonları, mevcut süreci yeni bir programla değiştirir.

`execvp(myargs[0], myargs);` ile `wc` programı çalıştırılır ve `p3.c` dosyası üzerinde kelime sayımı yapılır.

Eğer `execvp()` başarılı olursa, şu anki çocuk süreç sonlanır ve `wc` programı çalıştırılır. **Başarılı olduğu durumda**, bu noktadan sonra çocuk sürecin kalan kısmı çalıştırılmaz, çünkü süreç tamamen `wc` programı ile değiştirilmiştir.

Eğer `execvp()` başarısız olursa, program kaldığı yerden devam eder.

exec başarılı olursa, "this shouldn't print out" ifadesi asla ekrana yazdırılmaz, çünkü bu nokta çocuk süreç `wc` programı ile değiştirildiği için çalışmaz.

exec()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc=fork();
    if(rc<0){
        //fork failed:exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }else if(rc == 0){
        // child (new process)
        printf("hello, I am child (pid:%d)\n",(int) getpi
        char *myargs[3];
        myargs[0] =strdup("wc"); // progr                (world count)
        myargs[1] = strdup("p3.c"); /                ent:file to count
        myargs[2]=NULL; // m                end of array
        execvp(myargs[0                yargs); // run word count
        printf("this st                ouldn't print out");
    } else{
        // parent goes down this path(orginal process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d)(pid:%d)\n",rc,wc,(int) getpid());

    }
    return 0;
}
```

Eğer `rc > 0` ise, bu blok **ebeveyn sürecin** çalıştığı durumdur.

`int wc = wait(NULL);`:Ebeveyn süreç, çocuk sürecin tamamlanmasını bekler. `wait()` fonksiyonu, çocuk sürecin sonlanmasını bekler ve sonlanan çocuk sürecin PID'sini döndürür.

`printf("hello, I am parent of %d (wc:%d)(pid:%d)\n", rc, wc, (int)getpid());`:Ebeveyn süreç, çocuk sürecin PID'sini (`rc`) ve `wait()` ile dönen değeri (`wc`) ekrana yazdırır. Ebeveyn sürecin kendi PID'si de yazdırılır.