Prosesler

Dr. Öğr. Üyesi Ertan Bütün

Bu dersin içeriği hazırlanırken Operating System Concepts (Silberschatz, Galvin and Gagne) kitabı ve
Prof. Dr. M. Ali Akcayol'un (Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü) ders sunumlarından faydalanılmıştır.

https://codex.cs.yale.edu/avi/os-book/OS10/index.html

http://w3.gazi.edu.tr/~akcayol/BMOS.htm

Konular

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Amaçlar

- Tüm hesaplamaların (computing) temelini oluşturan proses kavramını tanıtmak.
  - Hesaplama (computing) – genel olarak bilgisayar kaynaklarını kullanarak bir amaca yönelik faaliyet olarak tanımlayabiliriz.
- Proseslerin zamanlanması (scheduling), oluşturulması, sonlandırılması ve haberleşmesi dahil olmak üzere çeşitli özelliklerini açıklamak
- Prosesler arası iletişim için kullanılan shared memory ve message passing yöntemlerini tanımlamak
- İstemci-sunucu sistemlerinde iletişimi tanımlamak

Proses kavramı

- Bir işletim sistemi çeşitli programları yürütür. İlk bilgisayarlar, işleri toplu yürüten **batch** sistemlerdi, ardından kullanıcı programlarını veya görevlerini **time-shared** (zaman paylaşımlı) olarak çalıştıran sistemler ortaya çıktı.
  - Batch systems: Batch isteminin kullanıcıları, bilgisayarla doğrudan etkileşime girmezlerdi. Her kullanıcı işini delikli kartlar (punch cards) gibi çevrimdışı bir cihazda hazırlar ve bilgisayar operatörüne sunardı. İşlemeyi hızlandırmak için benzer ihtiyaçlara sahip işler bir araya toplanır ve grup olarak çalıştırılırdı. Batch sistemlerde mekanik I/O cihazlarının hızı CPU'dan çok yavaş olduğu için CPU genellikle boştadır.
  - Time-shared: Birden fazla proses, CPU tarafından aralarında geçiş yapılarak yürütülür, ancak bu geçişler o kadar sık gerçekleştirilir ki kullanıcı bunu fark etmez. Böylece CPUnun boşta kaldığı zaman azaltılmış olur.

Proses kavramı

- Günümüz işletim sistemleri, birden çok programın hafızaya yüklenmesine ve eşzamanlı çalıştırılmasına izin verir.
- **Çalışmakta olan programa proses denilmektedir.**
- **CPU, aralarında geçiş yaparak tüm prosesleri eş zamanlı çalıştırılabilir.**
- Bir sistem, tek kullanıcılı bile olsa, birden fazla uygulamayı (Word, Excel, Web Browser, …) birlikte çalıştırabilir.

Proses ile Program Arasındaki Farklar

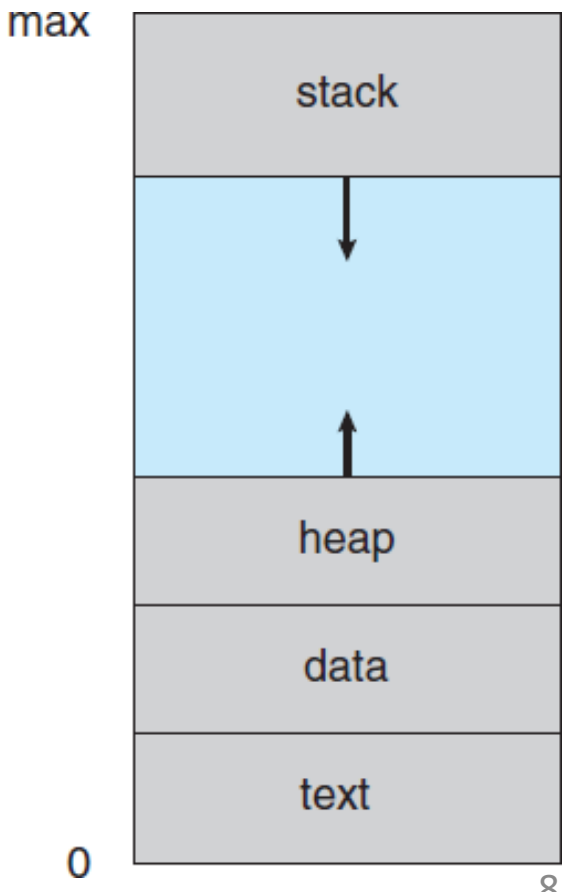
Program	**Proses**
Program, belirli bir görevi tamamlamak için tasarlanmış bir dizi komut içerir.	Proses, yürütülmekte olan bir programın bir örneğidir.
Program, ikincil bellekte bulunduğu için **pasif** bir varlıktır.	Proses, yürütülme sırasında oluşturulduğu için ve ana belleğe yüklendiği için **aktif** bir varlıktır.
Programın herhangi bir kaynak gereksinimi yoktur, yalnızca komutların saklanması için depolama alanı gerektirir.	Prosesin yüksek kaynak gereksinimi vardır, ömrü boyunca CPU, bellek adresi, I/0 gibi kaynaklara ihtiyaç duyar.
Programın herhangi bir kontrol bloğu yoktur.	Proses, Process Control Block (PCB) adı verilen kendi kontrol bloğuna sahiptir.

Proses ile Program Arasındaki Farklar

Program	Proses
Programın iki mantıksal bileşeni vardır: kod ve veri.	Program verilerine ek olarak, bir proses, yönetimi ve yürütülmesi için gerekli olan ek bilgileri de gerektirir.
Bir program, manuel olarak silinmediği sürece kalıcı hafızada saklandığından daha uzun bir kullanım ömrüne sahiptir.	Proses, görev tamamlandıktan sonra sonlandırıldığı için daha kısa bir ömre sahiptir.
Program kendini değiştirmez.	Birçok proses tek bir programı çalıştırabilir. Program kodu aynı olabilir ancak program verileri farklı olabilir.

Proses kavramı

- Bir prosesin anlık durum bilgisi program counter ve process registers ile tutulur.
  - program counter - instruction pointer olarak da bilinir, prosesin bir sonraki yürüteceği komutunun adresini tutar.
  - process registers - bir interrupt olduğunda prosesin register bilgileri kaydedilir, proses tekrar running durumuna geçtiğinde register bilgileri yeniden yüklenerek en son kaldığı yerden devam eder.
- Bir prosesin bileşenleri:
  - text section - program kodudur
  - data section - global değişkenleri saklar.
  - heap - prosese runtime'da dinamik olarak ayrılan bellektir.
  - stack - fonksiyonlar çağrıldığında geçici verileri saklar: fonksiyon parametreleri, return adresleri, lokal değişkenler vb.



Proses kavramı

- **Aynı program birden fazla proses ile ilişkili olabilir** (birden fazla eşzamanlı çalışan Web browser).
- **Her prosesin stack, data section ve heap kısımları farklıdır.**
- Program çalışma süresi boyunca boyutları değişmediği için text ve data sections boyutları sabittir.

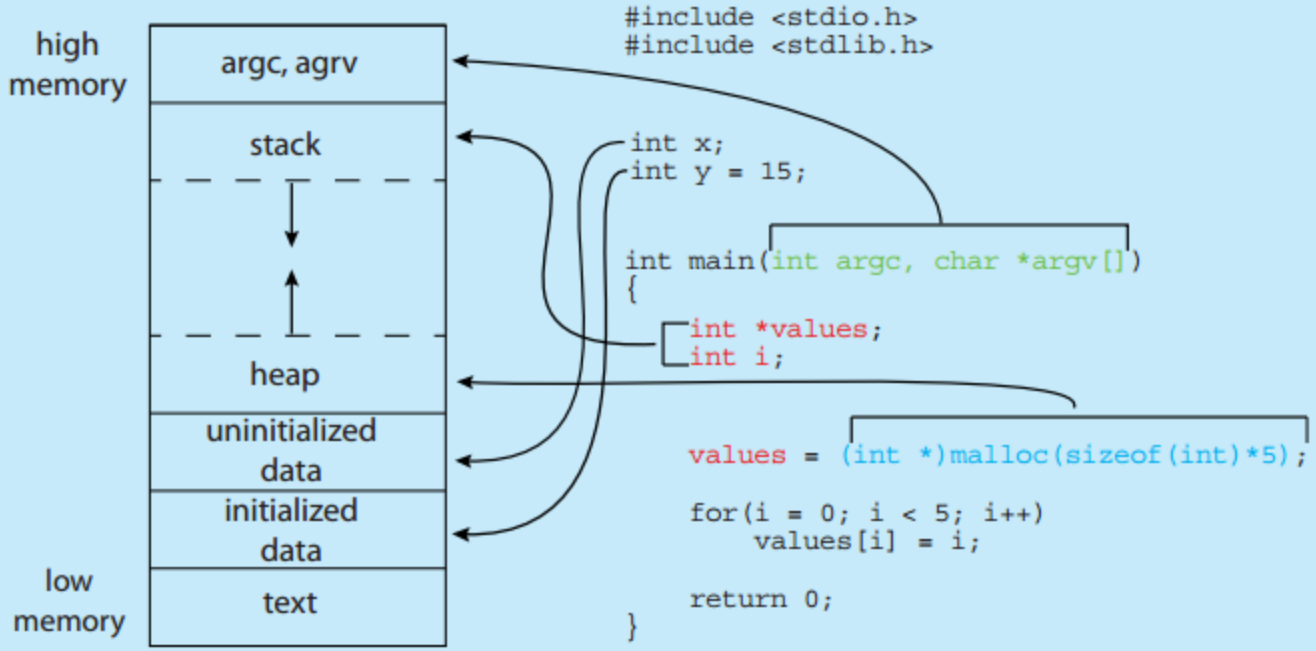
Proses kavramı

- stack ve heap sections, program yürütülürken dinamik olarak büyüyüp küçülebilir.
  - Bir fonksiyon her çağrıldığında, fonksiyon parametrelerini, yerel değişkenleri ve dönüş adresini içeren bir activation record, stack'a eklenir; fonksiyondan geri dönüldüğünde, stack'tan activation record çıkarılır.
  - Benzer şekilde, bellek dinamik olarak tahsis edildikçe heap büyüyecek ve bellek sisteme geri döndüğünde küçülecektir.
- stack ve heap sections birbirine doğru büyümesine rağmen, işletim sistemi bunların birbiriyle çakışmamasını sağlamalıdır.

MEMORY LAYOUT OF A C PROGRAM

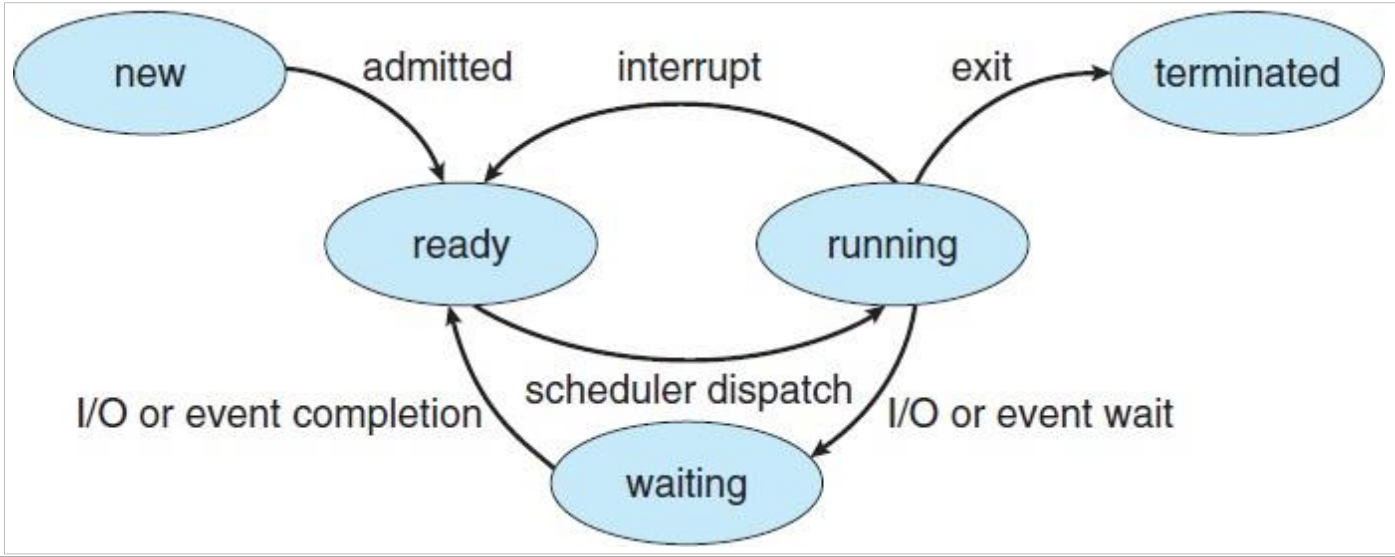
The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



Proses Durumları (Process states)

- **Bir proses çalıştığı sürece durum değiştirir.**
  - New: Proses oluşturulmaktadır.
  - Running: Komutlar çalıştırılmaktadır.
  - Waiting: Proses bir olayın gerçekleşmesini beklemektedir (I/O, bir cihazdan geribildirim).
  - Ready: Proses çalışmak için CPU'ya atanmak üzere bekliyor.
  - Terminated: Proses çalışmasını sonlandırır.*

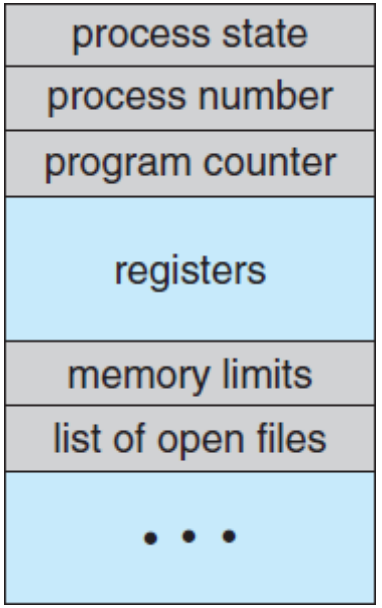


Process Control Block

- Process control block (PCB veya task control block); prosesle ilgili process state, program counter, bellek tahsisi, prosesin kullandığı dosyalar, prosesin kullandığı kaynaklar, prosesin scheduling bilgileri gibi proses hakkında tüm önemli bilgileri içeren bir veri yapısıdır.
- Her proses, işletim sisteminde bir PCB tarafından temsil edilir.
- İşletim sistemi bir prosesin oluşturulmasında, proses bir durumdan başka bir durumuna geçtiğinde (switching) prosesin daha sonra hiç durdurulmamış gibi kaldığı yerden devam ettirilmesinde, prosesin yürütülmesinin izlenmesinde PCB'leri kullanır.
- PCB, proses yönetiminde merkezi bir role sahiptir: işletim sisteminin çoğu bileşeni, özellikle scheduling ve kaynak yönetimi ile ilgili bileşenleri gerektiğinde PCB'lerin içeriğine erişir ve bu bilgiler ışığında kararları verirler ve gerektiğinde içerikler güncellenir.

Process control block

- Bir PCB'de aşağıdaki bilgiler tutulur:
  - Process ID/number : Her prosese özgün benzersiz bir proses ID/number atanır.
  - Process state: Durum bilgisi: new, ready, running, waiting, halted olabilir.
  - Program counter: Bu proses için sonraki komutun adresini gösterir.
  - CPU register'ları: Prosesle ilişki olan tüm register içerikleri tutulur. Bir interrupt olduğunda prosesin register bilgileri kaydedilir, proces tekrar running durumuna geçtiğinde bu registerlerdeki bilgiler ile en son kaldığı yerden devam eder.
  - CPU-scheduling information: Proses önceliğini içerir.
  - Memory-management information: proses için ayrılan bellek bilgilerini içerir
  - Accounting information: CPU kullanım oranları, zaman limitleri, account bilgileri ve proses numaralarını içerir.
  - I/O status information: Proseslere tahsis edilmiş I/O cihazları ile açık durumdaki dosyaları içerir.



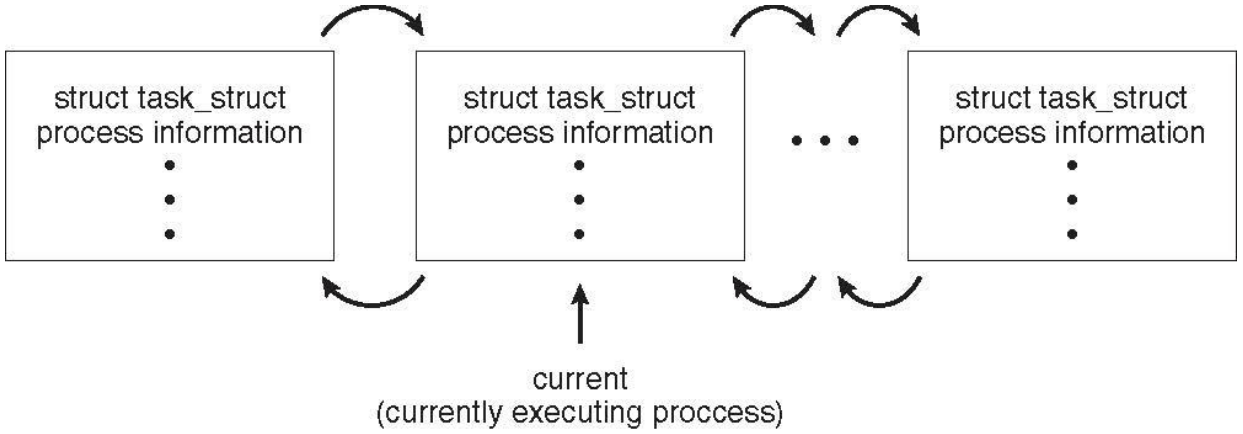
Threads

- **Tek thread ile bir proses kontrol edilir ve birden fazla görev aynı anda yapılamaz** (bir Word programında karakter girişi ile yazım denetleyiciyi aynı anda yapılamaz).
- **Modern işletim sistemlerinde bir proses ile birden fazla thread çalıştırılmasına izin verilir. (**bir Word programında bir thread kullanıcının klavyeden karakter girişini yönetirken diğer bir thread yazım denetleyiciyi çalıştırır**)**
  - **Bu özellik multicore işlemcilerde çok faydalıdır** ve birden çok thread eşzamanlı çalıştırılır.
  - Birden çok thread ile çalışan sistemlerde, PCB ile her bir thread'e ait bilgiler saklanır.

Process Representation in Linux

- Linux işletim sistemindeki PCB, C'de **task_struct** adında bir stucture ile temsil edilir. (`<include/linux/sched.h>`)
- Linux çekirdeğinde aktif prosesler, task_struct içinde doubly linked list veri yapısında tutulur.

```
pid t_pid;                      /* process identifier */
long state;                     /* state of the process */
unsigned int time_slice         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space of this process */
```



Konular

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Scheduling

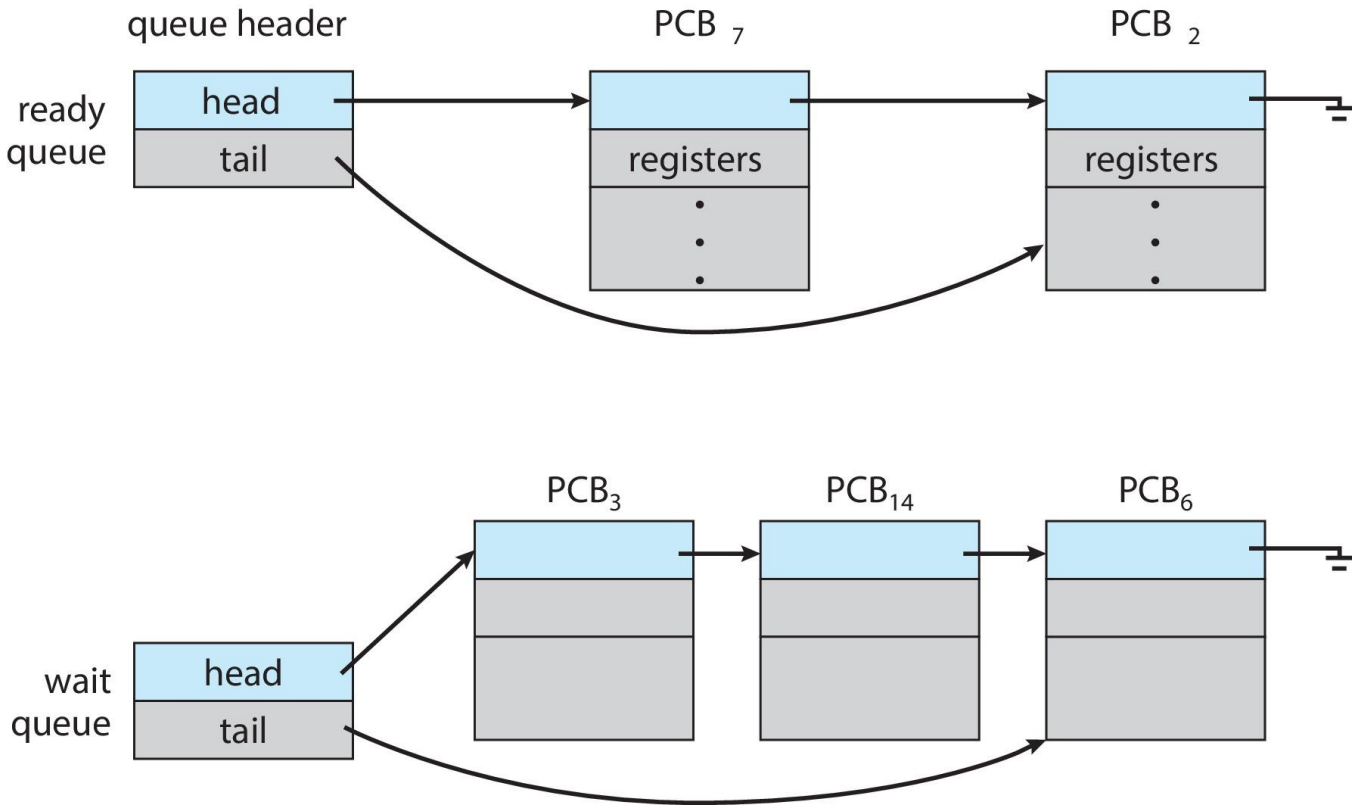
- **Multiprogramming sistemlerin** temel amacı, CPU kullanımını maksimuma çıkaracak şekilde prosesleri çalıştırmaktır**.**
- **Time sharing sistemlerde** ise, kullanıcıların her programla çalışırken etkileşime girebilmesi için bir CPU çekirdeği prosesler arasında sıklıkla değiştirilir.
- Multiprogramming ve timesharing sistemlerde bu amaçların gerçekleştirilebilmesi için **process scheduler** kullanılır.
- **Bir prosesi CPU'da çalışması için process scheduler seçer.**
- Her bir CPU çekirdeğinde aynı anda ancak bir proses çalışabilir, multicore bir sistemde aynı anda birden çok proses çalışabilir.

Scheduling queues

- **process scheduler** proseslerin zamanlama kuyruklarını (scheduling queue) yönetir:
  - job queue – sistemdeki tüm proseslerdir.
  - ready queue – hafızaya alınmış ve ready olarak CPU çekirdeğinde çalışmayı bekleyen proseslerdir.
  - device queue – I/O bekleyen proseslerdir.

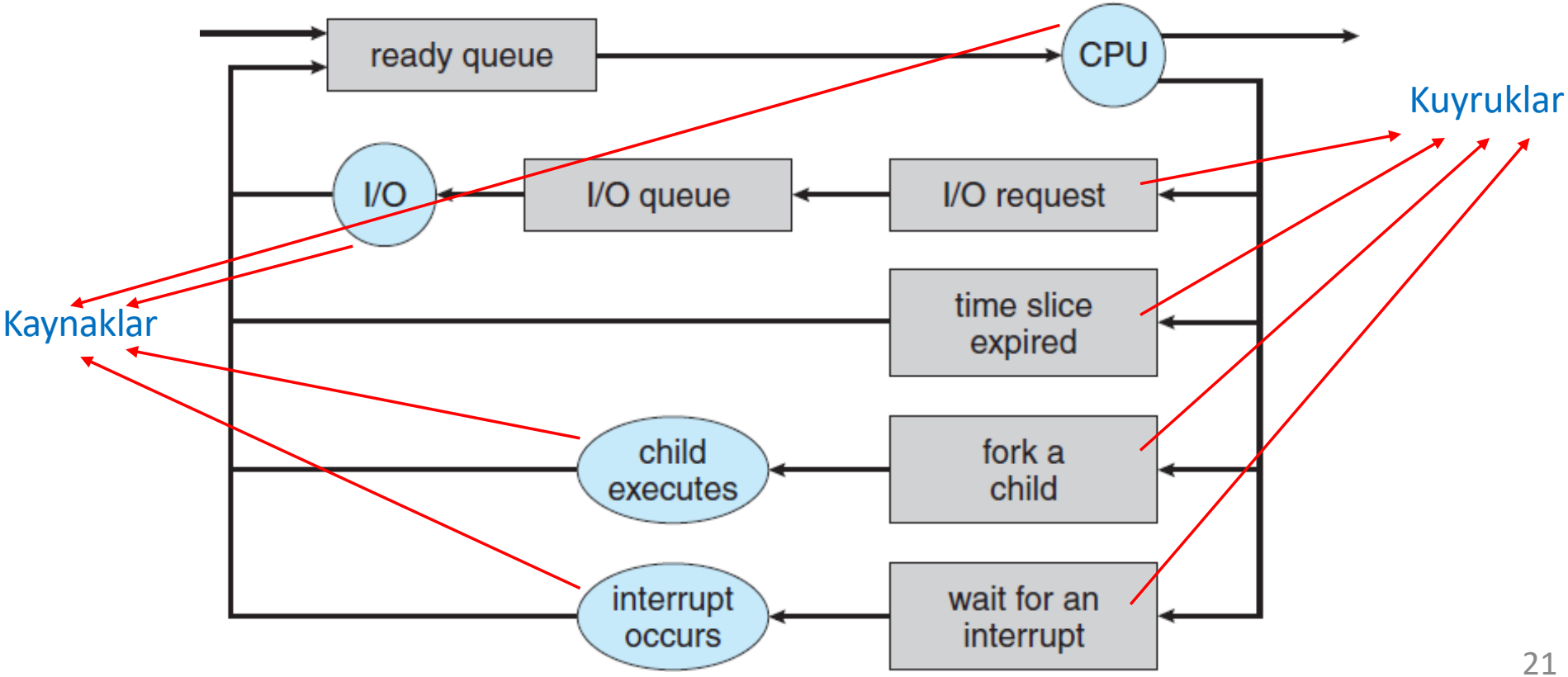
Scheduling queues

- Şekilde Proses scheduling kuyrukları için örnek diyagram verilmiştir.



Scheduling queues

- Proses scheduling için şekilde verilen queueing diagram, yaygın kullanılan bir gösterimdir.



Scheduling queues

- Proses bir CPU çekirdeği tahsis edilip proses CPU'da yürütülürken birkaç olaydan biri gerçekleşebilir:
  - Bir proses I/O isteğinde bulunursa, I/O kuyruğuna aktarılır.
  - Bir proses yeni bir child proses oluşturabilir ve daha sonra child prosesin sonlandırılmasını beklerken bir bekleme kuyruğuna alınabilir.
  - Proses, bir interrupt sonucunda veya zaman diliminin sona ermesi sonucunda (time slice expired) CPU çekirdeğinden zorla çıkarılabilir ve hazır kuyruğuna geri konulabilir.

Schedulers

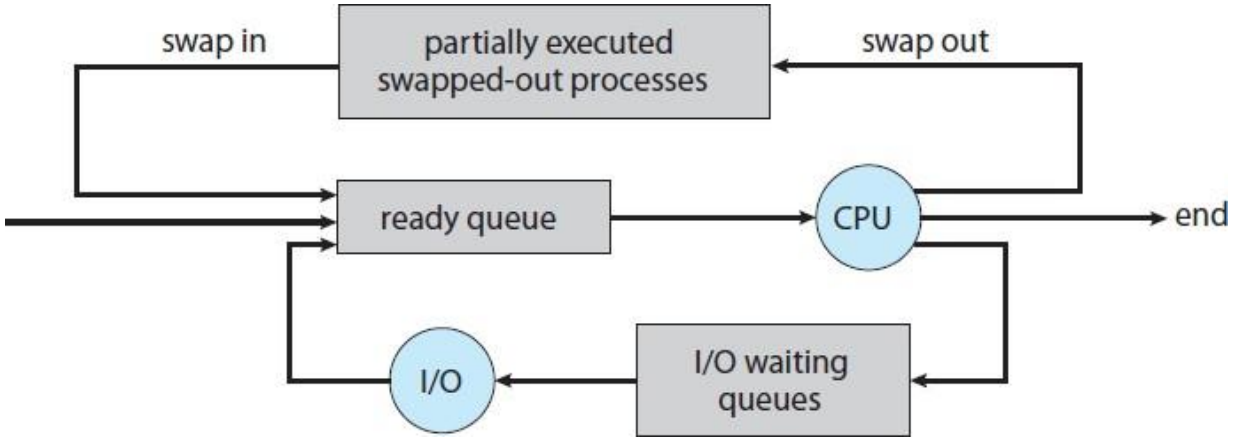
- Bir proses, çalışma süresi boyunca farklı kuyruklara alınabilir.
- Kuyruktaki proseslerin seçilmesi scheduler tarafından gerçekleştirilir.
- Genellikle çok sayıda proses çalıştırılmak üzere sisteme gönderilir. Bu prosesler disk üzerinde biriktirilir ve daha sonra çalıştırılır.
- Long-term scheduler (veya job scheduler) disk üzerindeki bu işleri seçerek çalıştırılmak üzere hafızaya yükler.
- Short-term scheduler (veya CPU scheduler) bu işlerden çalıştırılmak üzere hazır olanları seçerek CPU'yu onlara tahsis eder.
- Short-term scheduler çok kısa aralıklarla (<100ms) ve sıklıkla çalıştırılır. Long-term scheduler ise dakika seviyesindeki aralıklarla çalıştırılır.

Schedulers

- Multiprogramming ve timesharing sistemlerde, uygun bir şekilde prosesleri yürütmek için proseslerin genel davranışları dikkate alınır.
- Genel olarak prosesler I/O-bound ve CPU-bound olarak iki gruba ayrılır:
  - I/O-bound prosesler I/O işlemleri için daha fazla süre ayırırlar.
  - CPU-bound prosesler CPU ile hesaplama işlemleri için daha fazla süre ayırırlar.

Medium Term Scheduling

- Bazı işletim sistemleri, swapping olarak bilinen medium-term scheduler biçimine sahiptir.
- Bazen bir prosesi bellekten çıkarmak ya da CPU'ya geçme rekabetinden çıkarmak avantajlı olabilir, bunun için bazı prosesler swapping ile bellekten diske alınır, böylece multiprogramming derecesi düşürülür.
- swapping tipik olarak yalnızca bellek aşırı yüklendiğinde (overcommitted) ve serbest bırakılması gerektiğinde uygulanır.

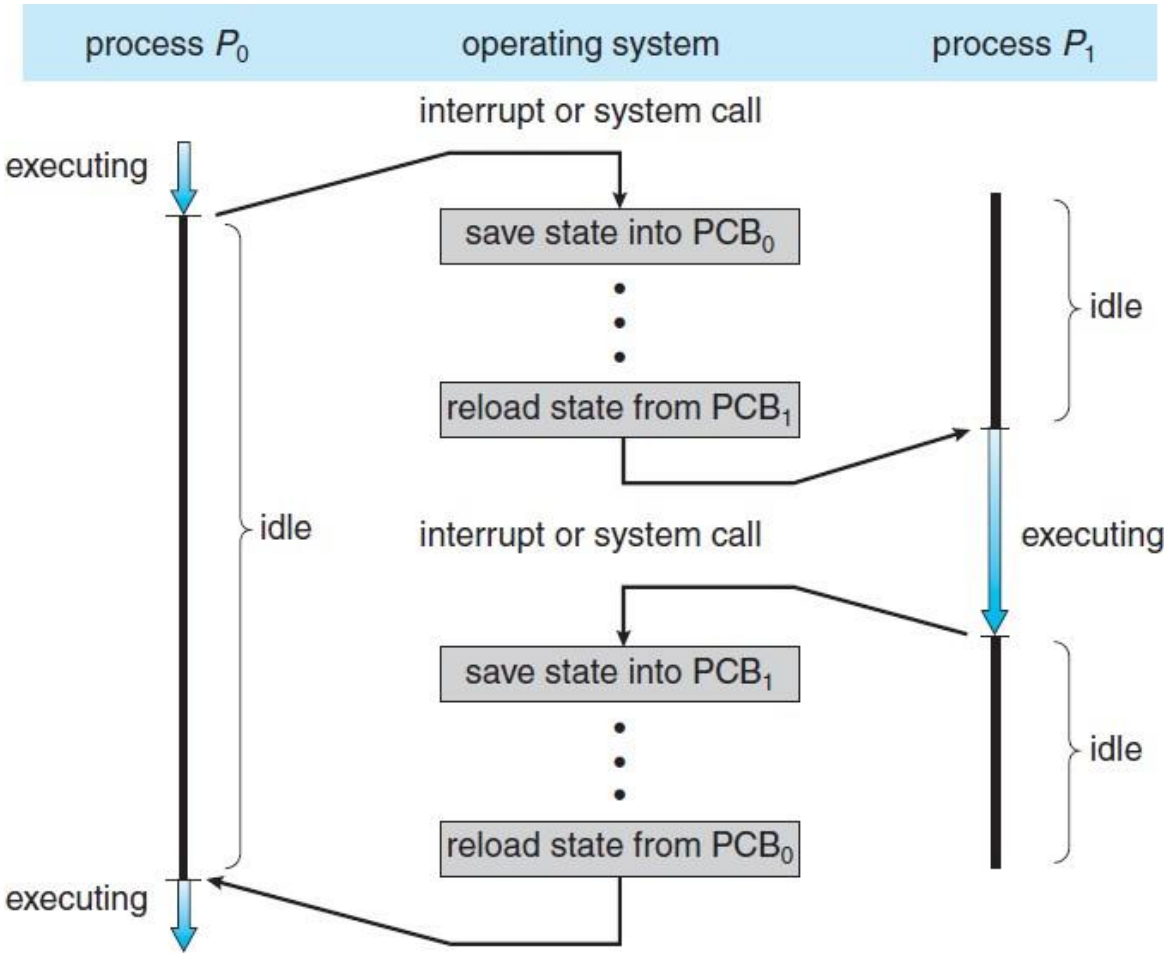


Context Switch

- Bir prosesin CPU'da yürütülürken başka bir prosesin CPU'ya geçmesine karar veridiğinde çıkacak prosesin context bilgilerinin kaydedilmesi ve CPU'ya geçecek prosesin daha önceden kaydedilmiş context bilgilerinin yüklenmesi işlemine **context switch** denir.
- Bir proses için **context** , process control block (PCB) içerisinde saklanır. **context**; CPU register'larının değerlerini, hafıza yönetim bilgilerini, process state bilgisini içerir.
- Context switch süresi, bir iş üretilmediği için **overhead (ek yük)** olarak adlandırılır ve bu süre daha çok donanıma göre değişir (genellikle birkaç milisaniyedir)
- Bazı donanımlarda her CPU için birden çok register seti sağlanır, aynı anda birden fazla **context** yüklenir, **context switch** işlemi basit bir şekilde current register setini gösteren pointer in değiştirilmesiyle sağlanır.

CPU Switch From Process to Process

- Bir context switch, CPU bir prosesten diğerine geçtiğinde oluşur.



Mobil Sistemlerde Multitasking

- Mobil cihazların donanım kısıtı nedeniyle, iOS'un ilk sürümleri kullanıcı uygulamalarına multitasking desteği sağlayamamıştır.
- iOS 4'ten başlayarak Apple, kullanıcı uygulamaları için sınırlı bir multitasking biçimi sağlamıştır, böylece tek bir ön plan uygulamasının (foreground) birden çok arka plan uygulamasıyla (background) aynı anda çalışmasına izin verilmiştir.
- Mobil cihazların kapasiteleri arttıkça (daha büyük bellek kapasiteleri, çok çekirdekli işlemciler ve daha uzun pil ömrü) iOS cihazlar multitasking için daha çok fonksiyonellik sağlayabilmektedirler.
  - Örneğin büyük ekran iPadler split-screen ile aynı anda ekranda iki foreground processe izin verebilirler.

Mobil Sistemlerde Multitasking

- Android işletim sistemi de multitasking destekler.
  - Bir uygulamanın background olarak yürütülmesi gerektiriyorsa, background proses adına çalışan ayrı bir uygulama bileşeni olan **servis** kullanmalıdır.
  - Örneğin bir ses akışı (audio streaming) uygulaması backgrounda geçerse **servis** background uygulaması adına ses verilerini ses aygıtı sürücüsüne göndermeye devam eder.
  - Background proses askıya alınsa bile servis çalışmaya devam edebilir.
  - Servislerin kullanıcı arayüzü yoktur, az bellek kullanırlar.

Konular

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

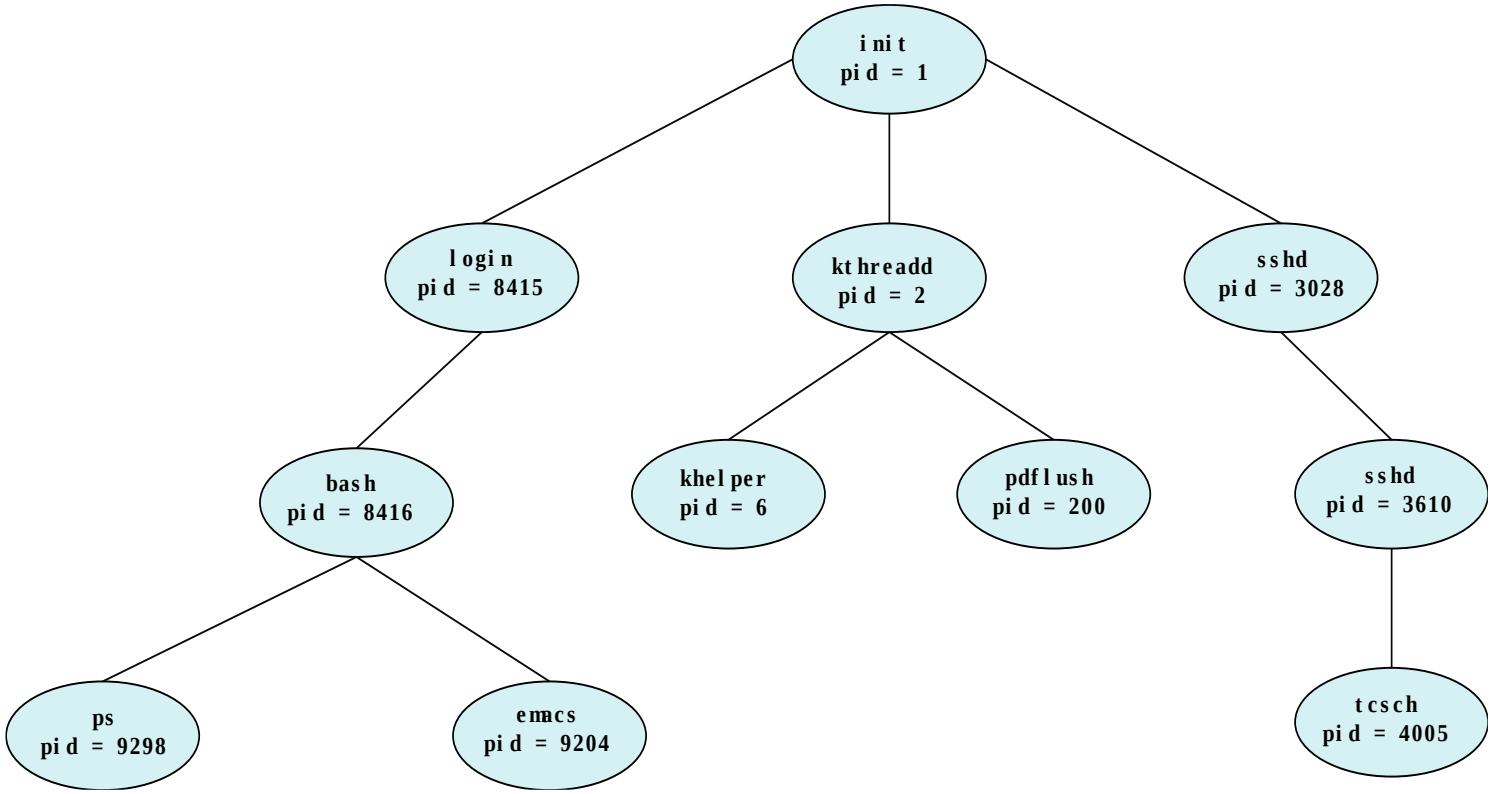
Process Creation

- Prosesler dinamik olarak oluşturulur ve silinirler.
- Bir proses çalışması sırasında birkaç tane başka yeni prosesi oluşturabilir. Prosesi oluşturan **parent process**, yeni oluşturulan **child process** olarak adlandırılır.
  - Child processler başka child processleri de oluşturabilir, bu şekilde bir **process tree** oluşur.
- Unix, Linux ve Windows gibi işletim sistemleri, her proses için **process identifier** (**pid**) değeri atarlar.
  - Her proses için atanan değer unique dir, prosese erişim için kullanılır.

Process Creation

- Bir parent process, yeni bir child process oluşturduğunda, yeni child process CPU, hafıza, dosyalar ve I/O cihazları gibi kaynaklara ihtiyaç duyar.
- Parent-child process kaynak paylaşımı türleri
  - Parent ve child processler tüm kaynakları paylaşabilir.
  - Child processlere parent process in kaynaklarının bir kısmı paylaştırabilir.
  - Parent ve child process, kaynaklarını paylaşmaz, işletim sistemi tarafından child process'e yeni kaynak tahsis edilebilir.
- Bir parent process, çalışmaya devam etmek için child processlerin tümünün veya bir kısmının sonlanmasını bekleyebilir veya parent process child processle eşzamanlı çalışmasını sürdürebilir.

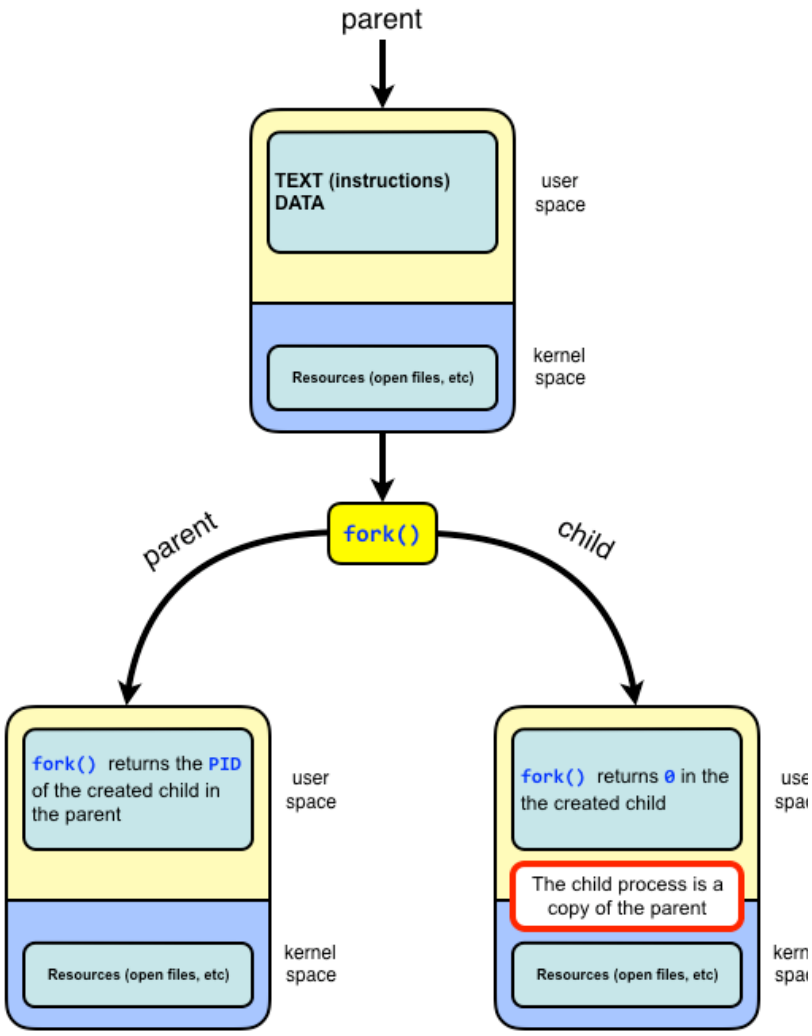
A Tree of Processes Linux



Process Creation

- Child processler için adress-space seçenekleri:
  - Child process, parent processin program ve datasının bir kopyasına sahip olabilir.
  - Child process, yükelenen yeni bir programa sahip olabilir.

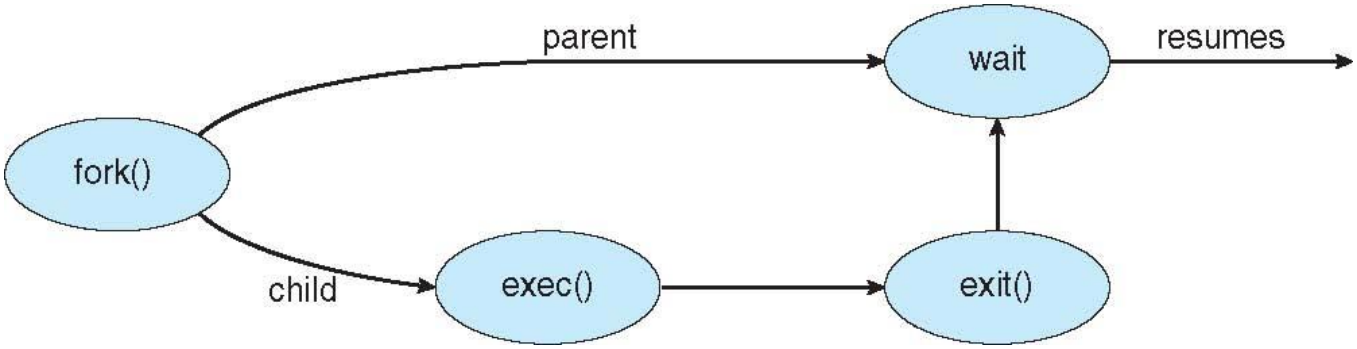
Creating a separate process using the UNIX fork()




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Creation

- Bir önceki örnekte parent process wait sistem çağrısı kullanılmıştı. Child process tamamlanınca (exit) parent process kaldığı yerden devam edecektir.



Process işlemleri

Process creation - Windows

Yeni process için sistem çağrısı

Yeni process "mspaint.exe"

Yeni child process özellikleri (Window size, giriş/çıkış dosyaları, ...)

Process identifier

Process oluşturma hatası

Parent process, child process'i bekliyor.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Bir proses son komutunu da çalıştırıp tamamlandığında işletim isteminden exit() sistem çağrısı ile silinmeyi ister.
  - Sonlanan proses, durum bilgisini (genellikle bir tamsayıdır) parent process'e wait () sistem çağrısı ile döndürürebilir.
  - Prosesin kullandığı kaynaklar işletim sistemi tarafından serbest bırakılır.

Process Termination

- Bir parent process çeşitli nedenlerden dolayı child process'i **abort()** sistem çağrısı ile sonlandırabilir:
  - Child process kaynak kullanım sınırını aştığında, parent process tarafından sonlandırılabilir. (Kaynak aşımının belirlenmesi için parent process'in child processlerin durumunu inceleyecek bir mekanizmaya sahip olması gerekir.)
  - Child process'in yaptığı işe gerek kalmayabilir.
  - parent process sonlandırılırsa, işletim sistemi child process'lerin devam etmesine izin vermeyebilir.

Process Termination

- Bazı sistemlerde, parent process sonlandırıldığında bir child processing var olmasına izin verilmez. Bu durumda:
  - Parent process normal veya anormal şekilde sonra erdiğinde tüm child processler sonlandırılmalıdır. Bu işleme **cascading termination** denir.
  - Sonlandırma, işletim sistemi tarafından başlatılır.
- Parent process, wait () sistem çağrısını kullanarak bir child processin sonlandırılmasını bekleyebilir. Wait çağrısı, durum bilgisini ve sonlandırılan prosesin pid'sini döndürür.

```
pid_t pid;
int status;
pid = wait(&status);
```

Process Termination

- Bir proses sona erdiğinde, kullanmış olduğu kaynakları işletim sistemi tarafından serbest bırakılır.
- Ancak process table, prosesin çıkış durumunu içerdiğinden proses tablosundaki içerik, parent process wait() sistem çağrısını çağırıncıya kadar orada kalmalıdır.
- Bir proses sonlandırılmış ancak parenti henüz wait() 'i çağırmamışsa, bu proses **zombie** proses olarak bilinir. Tüm prosesler sonlandırıldığında bu duruma geçer, ancak genellikle sadece kısa süre **zombie** olarak var olurlar. Parent wait() 'i çağırdığında, **zombie** prosesin tanımlayıcısı ve proses tablosu serbest bırakılır.

Process Termination

- Parent process wait()i çağırmadan kendisi sonlanırsa child processler **orphan (kimsesiz)** proses olarak bilinir.
- Geleneksel UNIX sistemleri, **init processi** (parent root process) **orphan** processlere yeni parent atayarak bu senaryoyu ele alır.
- **init process** periyodik olarak wait()'i çağırır, böylece **orphan** processin çıkış durumunun toplanmasına ve **orphan** proses tanımlayıcısını ve proses tablosunu serbest bırakmasına izin verir.

Android Process Hierarchy

- Sınırlı bellek gibi kaynak kısıtlamaları nedeniyle, mobil işletim sistemleri, sınırlı sistem kaynaklarını geri kazanmak için mevcut işlemleri sonlandırmak zorunda kalabilir.
- Android, rastgele bir prosesi sonlandırmak yerine, proseslerin önem sırasını (**importance hierarchy**) belirlemiştir.
- Yeni veya daha önemli bir proses için kaynakları kullanılabilir hale getirmek üzere proseslerin sonlandırılması gerektiğinde, prosesler artan önem sırasına göre sonlandırılır.

Android Process Hierarchy

- Önem sırasına göre proseslerin hiyerarşisi (azalan sıraya göre):
  - **Foreground process** - Kullanıcının halihazırda etkileşimde bulunduğu uygulamayı temsil eden ekranda görünen mevcut proses.
  - **Visible process** — Doğrudan ön planda görünmeyen ancak foreground processin atıfta bulunduğu yani kullanıcının farkında olduğu bir etkinliği **(activity)** gerçekleştiren bir proses (yani, foreground processte durumu görüntülenen bir etkinliği gerçekleştiren bir proses)*
  - **Service process** — Background processe benzer ancak kullanıcının görebileceği bir etkinlik **(activity)** gerçekleştiren bir prosestir. (müzik akışı gibi)
  - **Background process** — Bir etkinlik **(activity)** gerçekleştiren ancak kullanıcı tarafından görülemeyen bir proses.
  - **Empty process** - Herhangi bir uygulamayla ilişkilendirilmiş etkin bileşen içermeyen bir proses*

Multiprocess Architecture – Chrome Browser

- Bazı web tarayıcıları single process olarak çalışır.
  - Single process olarak çalışan web tarayıcı uygulamalarında bir web sitesi soruna neden olursa, tarayıcının tamamı (tüm sekmeler) kilitlenebilir veya çökebilir.
- Google Chrome Tarayıcı, 3 farklı proses türü ile multiprocess olarak tasarlanmıştır:
  - **Browser** process - kullanıcı arayüzünü, diski ve network I/O'yu yönetir.
  - **Renderer** process - web sayfalarını render eder; HTML, Javascript ve resimleri işlerler. Açılan her web sitesi için yeni bir renderer oluşturulur.
  - **Plug-in** process - kullanılan her plug-in türü (Flash veya QuickTime gibi) için bir plug-in proses oluşturulur.

Multiprocess Architecture – Chrome Browser

- Multiprocess yaklaşımının avantajı, web sitelerinin birbirinden ayrı olarak çalışmasıdır. Bir web sitesi çökerse, yalnızca onu render eden proses etkilenir; diğer tüm prosesler zarar görmeden kalır.
- Ayrıca render process disk ve network I/O'ya erişimin kısıtlı olduğu **sandbox** alanda çalışır, böylece güvenlik açıklarının etkisi en aza indirgenir.

Konular

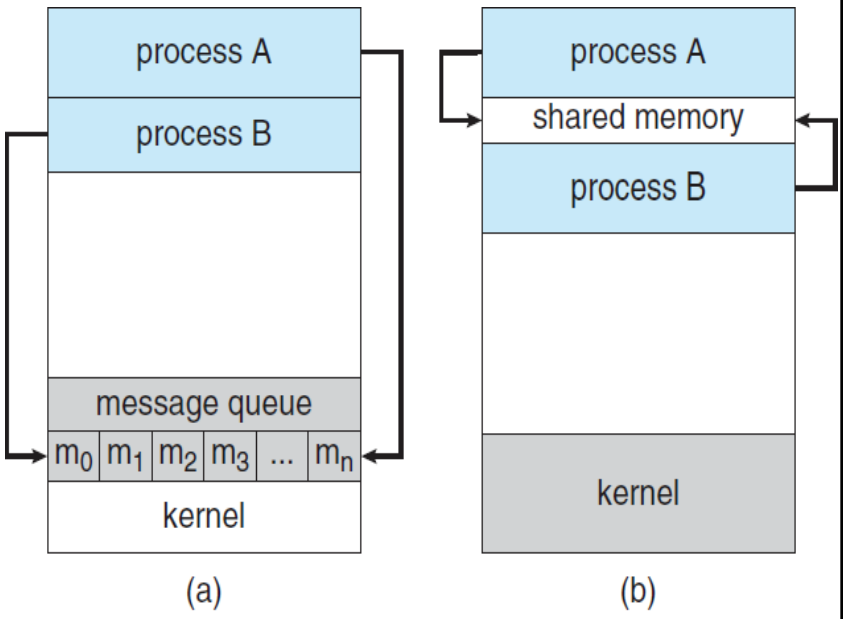
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Interprocess Communication (IPC)

- Prosesler işletim sisteminde independent process veya cooperating process olarak çalışırlar.
- Independent process'ler diğer prosesleri etkilemezler ve onlardan etkilenmezler.
- Cooperating process'ler diğer prosesleri etkilerler ve onlardan etkilenirler (diğer proseslerle veri paylaşımı yaparlar).
- Cooperating proseslere birkaç nedenden dolayı ihtiyaç duyulur:
  - **Information sharing**: Birkaç uygulama aynı bilgi parçasıyla ilgilenebileceğinden, bu tür bilgilere eşzamanlı erişime izin verecek bir ortam sağlamalıyız.
  - **Computation speedup**: Görevleri hızlı yürütmek için multicore işlemcili bilgisayarlarda görevler parçalar halinde eşzamanlı yürütülürler.
  - **Modularity**: Sistem modüler (prosesler, thread'ler) halinde oluşturulabilir ve bu modüller arasında iletişim yapılabilir.
  - **Convenience**: Bir kullanıcı farklı işleri (müzik dinleme, metin yazma, compile, …) aynı anda gerçekleştirebilir.

Interprocess Communication (IPC)

- Cooperating process'ler aralarında veri alışverişine imkan tanıyan bir prosesler arası haberleşme (interprocess communication, IPC) mekanizmasına ihtiyaç duyarlar.
- Cooperating process'ler shared memory ve message passing modelleri ile veri aktarımı yaparlar.
  - **Shared memory** modelinde, hafızada bir bölge prosesler arasında paylaştırılır.
  - **Message passing** modelinde, haberleşme prosesler arasında mesaj alışverişi ile sağlanır.



Communications models. (a) Message passing. (b) Shared memory.

Shared Memory vs Message Passing

- Her iki model de işletim sistemlerinde yaygındır, ve birçok sistem her ikisini de uygular.
- **Message passing** , daha küçük miktarlarda veri alışverişi için kullanışlıdır, çünkü çakışmalardan kaçınılması gerekmez. Message passing dağıtık bir sistemde uygulanması **Shared memory** den daha kolaydır.
- **Shared memory, message passing** den daha hızlı olabilir , çünkü **message** passing sistemleri tipik olarak sistem çağrıları kullanılarak uygulanır ve bu nedenle daha fazla zaman alan kernel müdahalesi gerektirir.
- **Shared memory** sistemlerinde, sistem çağrıları yalnızca paylaşılan bellek bölgelerini kurmak için gereklidir. Paylaşılan bellek oluşturulduktan sonra, tüm erişimler rutin bellek erişimleri olarak değerlendirilir ve kernelden herhangi bir yardım gerekmez.

IPC in **Shared Memory**

- Shared Memory yöntemi ile IPC gerçekleştirmek için, bir paylaşılan hafıza bölgesi oluşturulmalıdır.
- Tipik olarak, bir paylaşılan bellek bölgesi, paylaşılan bellek bölümünü oluşturan prosesin adres alanında bulunur. Bu paylaşılan bellek bölümünü kullanarak haberleşmek isteyen diğer prosesler, bunu adres alanlarına eklemelidir.
- Verilerin biçimi ve konumu bu prosesler tarafından belirlenir ve işletim sisteminin kontrolü altında değildir.
- Bu yöntemin en kritik noktası kullanıcı proseslerinin paylaşılan belleğe eriştiklerinde eylemlerini **senkronize etmelerine** izin verecek bir mekanizmanın sağlanmasıdır.

Producer-Consumer Problem

- Producer-consumer problemi, bounded buffer olarak da bilinir, cooperating processler için bilinen bir paradigmadır: Bir producer process, bir consumer process tarafından tüketilen bilgileri üretir.
- Örneğin; compiler bir programı derler ve assembly kod üretir, assembler bu kodu alır ve object kod üretir, loader ise bu kodu giriş olarak alır.

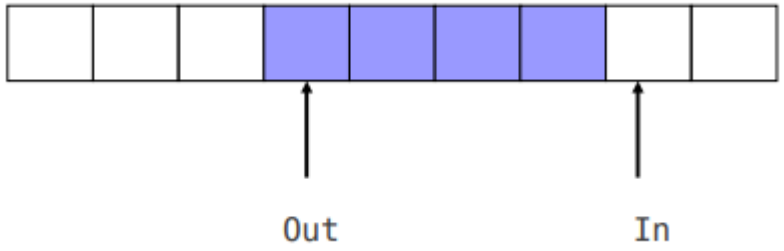
Producer-Consumer Problem – Shared Memory Solution

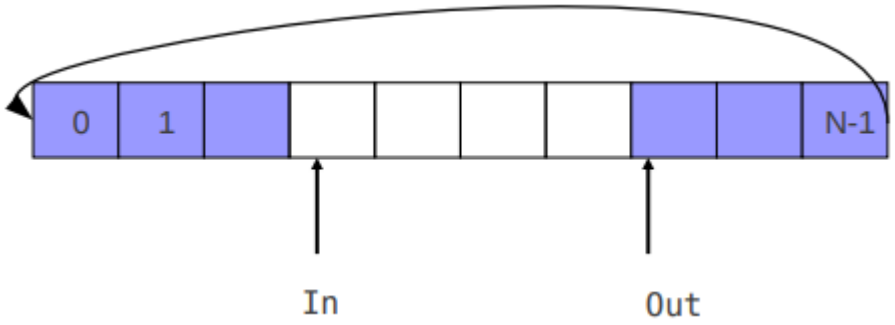
- Producer-consumer problemi shared memory ile çözülebilir. Bu yöntemde producer ve consumer proseslerinin eş zamanlı olarak çalışmasına izin vermek için, producer tarafından doldurulabilen ve consumer tarafından boşaltılabilen bir buffere ihtiyaç vardır.
- Bu buffer, producer ve consumer processleri tarafından paylaşılan bir bellek bölgesinde yer alacaktır. producer, consumer başka bir ürünü tüketirken bir ürünü üretebilir.
- **Producer ve consumer senkronize olmalıdır, böylece consumer henüz üretilmemiş bir ürünü tüketmeye çalışmaz.**
- İki tür buffer kullanılabilir:
  - Unbounded buffer: buffer boyutuna pratik bir sınır getirilmez. consumer yeni veriler beklemek zorunda kalabilir, ancak producer her zaman yeni veriler üretebilir.
  - Bounded buffer: sabit bir boyutu vardır. consumer buffer boş ise producer de buffer dolu ise beklemelidir.

Bounded Buffer Problem – Shared Memory Solution

- Aşağıda shared memory yöntemi ile bounded buffer tanımlaması verilmiştir.
- Paylaşılan buffer, **in** ve **out** isminde iki işaretleyiciye sahip dairesel bir dizi olarak uygulanır. **in,** bufferdeki bir sonraki boş konuma işaret eder; **out**, bufferdeki ilk dolu konuma işaret eder.

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```





Bounded Buffer Problem– Shared Memory Solution

Producer

- Bu gerçekleştirimde aynı anda buffer üzerinde en fazla BUFFER SIZE−1 öğeye izin verilir. (Dolu mu kontrolü -> ((in+1)% BUFFERSIZE)==out)

```
item next_produced;

while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer Problem – Shared Memory Solution

Consumer

```
item next_consumed;

while (true) {
     while (in == out)
          ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next consumed */
}
```

Message Passing in IPC

- Message passing, proseslerin aynı adres alanını paylaşmadan haberleşmesine ve eylemlerini senkronize etmesine izin veren bir mekanizma sağlar.
- Özellikle, haberleşecek proseslerin bir ağ ile bağlı farklı bilgisayarlarda bulunabildiği dağıtık ortamlarda kullanışlıdır. (chat programı)
- Message passing modelinde en az iki işlem tanımlanır:
  - send(message)
  - receive(message)
- Mesaj boyutları sabit uzunlukta veya değişken uzunlukta olabilir.
- P ve Q prosesleri iletişim kurmak istiyorsa,
  - Bir ***communication link*** aralarında kurulmalıdır.
  - send/receive işlemleri üzerinden mesaj alışverişi yapılmalıdır.

Message Passing in IPC

- Gerçekleştirim sorunları:
  - Linkler nasıl kurulacak?
  - Bir link, ikiden fazla prosesle ilişkilendirilebilir mi?
  - Haberleşecek her proses çifti arasında kaç link olabilir?
  - Bir linkin kapasitesi ne olacak?
  - Linkin barındırabileceği mesajın boyutu sabit mi yoksa değişken mi olacak?
  - Bir link tek yönlü mü yoksa çift yönlü mü olacak?

Message Passing in IPC

- Communication link gerçekleştirimi seçenekleri:
  - Fiziksel olarak link gerçekleştirim
    - Shared memory
    - Hardware bus
    - Network
  - Mantıksal olarak link gerçekleştirimi
    - Direct or Indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Haberleşmek isteyen her proses, iletişimin alıcısını veya göndericisini açıkça adlandırmalıdır.
  - send (P, message) – send a message to process P
  - receive(Q, message) – receive a message from process Q
- Communication linkin özellikleri:
  - Haberleşmek isteyen her proses çifti arasında otomatik olarak bir bağlantı kurulur. Proseslerin haberleşmek için yalnızca birbirlerinin kimliğini bilmesi gerekir.
  - Bir link, tam olarak bir çift prosesle ilişkilidir.
  - Her proses çifti arasında tam olarak bir link vardır.
  - Link tek yönlü olabilir, ancak genellikle iki yönlüdür.

Indirect Communication

- Mesajlar mailbox veya portlara gönderilir ve buradan alınır.
  - Her mailbox unique bir id ye sahiptir
  - iki proses yalnızca paylaşılan bir mailboxa sahipse iletişim kurabilir.
- Communication linkin özellikleri:
  - Link, ancak prosesler ortak bir mailboxı paylaşıyorlarsa kurulur.
  - Bir link, birden çok prosesle ilişkili olabilir.
  - Her proses çifti arasında her bir linkin bir mailboxa karşılık geldiği bir dizi farklı bağlantı bulunabilir.
  - Link tek yönlü veya çift yönlü olabilir.
- send ve receive şöyle tanımlanır:
  - **send**(A, message) – send a message to mailbox A
  - **receive**(A, message) – receive a message from mailbox A

Indirect Communication

- P1, P2 ve P3 proseslerinin hepsinin A mailboxı paylaştığını varsayalım. P1 prosesi A'ya bir mesaj gönderirken hem P2 hem de P3, A'dan mesaj almak için receive() çalıştırıyorsa P1 tarafından gönderilen mesajı hangi proses alacak? Bunun cevabı, aşağıdaki yöntemlerden hangisinin seçildiğine bağlıdır:
  - Bir linkin en fazla iki prosesle ilişkilendirilmesine izin verilebilir.
  - Bir receive işlemini yürütmek için bir seferde yalnızca bir prosese izin verilebilir.
  - Sistemin rastgele alıcıyı seçmesine izin verilebilir (P2 veya P3'ten sadece biri mesajı alabilir) . Sistem, hangi prosesin mesajı alacağını seçmek için bir algoritma tanımlayabilir (örneğin, proseslerin mesajları sırayla aldığı round robin). Alıcının kim olduğu gönderene bildirilir.

Synchronization

- Message passing için bir diğer yöntem **blocking/unblocking** yöntemidir. (**synchronous** ve **asynchronous** olarak da bilinir)
- **Blocking**, **synchronous** olarak yürütülür.
  - Blocking send – mesaj receiver tarafından alınıncaya kadar sender bloke olur.
  - Blocking receive - bir mesaj mevcut olana kadar receiver bloke olur.
- **Nonblocking**, **asynchronous** olarak yürütülür.
  - Nonblocking send – gönderici proses mesajı gönderir ve işleme devam eder.
  - Nonblocking receive – receiver geçerli bir mesaj veya boş bir mesaj alır.

Synchronization

- Blocking send() receive() komutları ile producer consumer probleminin çözümü sıradanlaşır.

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Buffering

- Haberleşecek prosesler arasında mesaj alışverişi geçici bir kuyrukla yapılır. Temel olarak, bu tür kuyruklar üç şekilde uygulanabilir:
  1. **Zero capacity** – linkte bekleyen mesaj yoktur, receiver mesajı alana kadar sender bloke edilir.
  2. **Bounded capacity** – kuyruğun n uzunluğunda sabit bir uzunluğu vardır, en fazla n tane mesaj tutulabilir. Kuyruk doluysa kuyrukta boş yer oluncaya kadar sender bloke edilmelidir, kuyruk dolu değilse mesaj kuyruğa yerleştirlir sender beklemeden çalışmasına devam eder.
  3. **Unbounded capacity** – kuyruk sonsuz uzunluktadır, istenilen sayıda mesaj kuyrukta tutulabilir, sender asla bloke edilmez.

Examples of IPC Systems

- POSIX Shared Memory
- Mach communication is message based
- Windows – provides message passing using shared message
- Pipes – earliest IPC mechanisms on UNIX systems

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```
    ftruncate(shm fd, 4096);
    ```
  - Use **mmap()** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **mmap()**
  - Now the process could write to the shared memory
    ```
    sprintf(shared memory, "Writing to shared memory");
    ```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

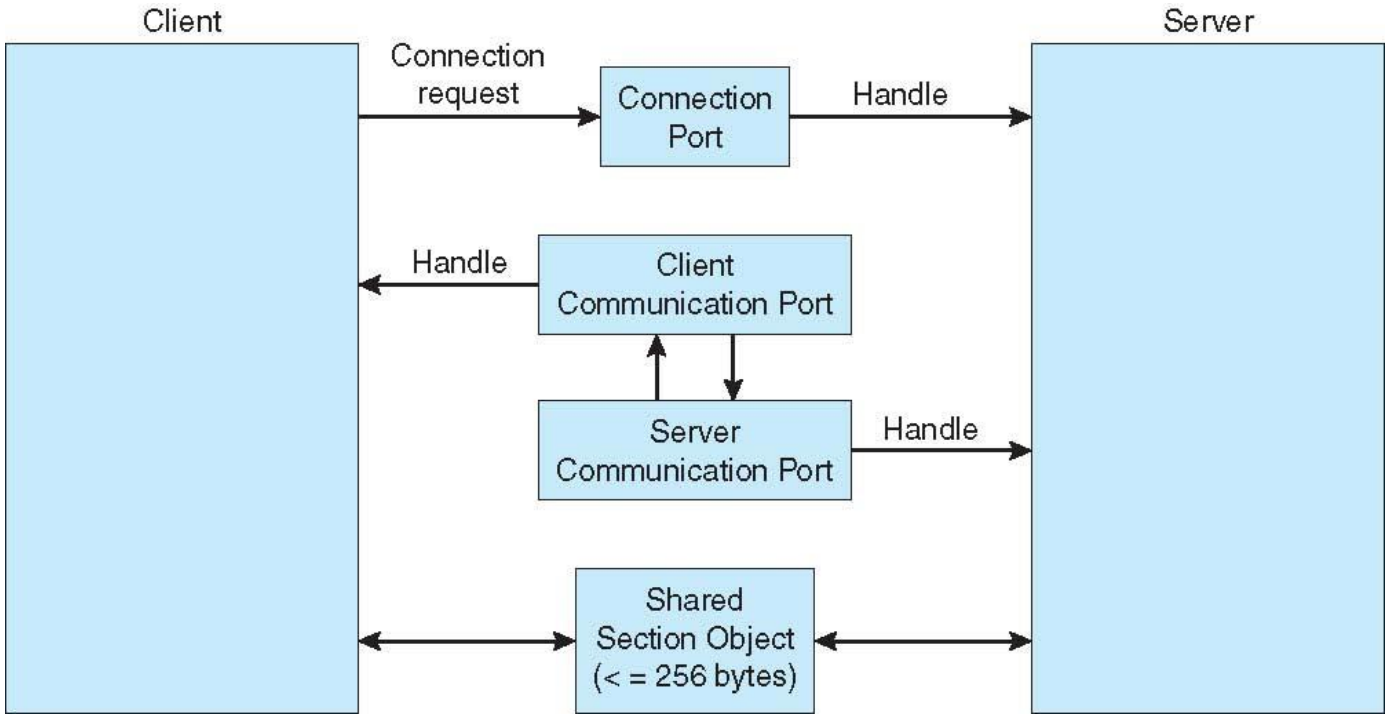
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows

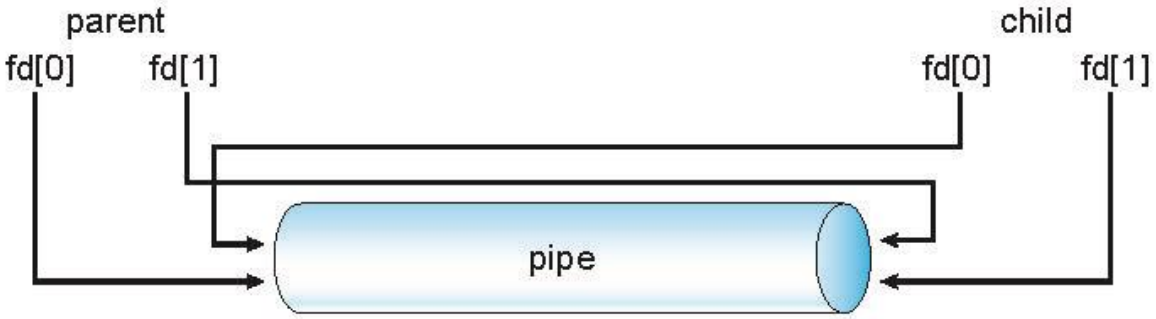


Pipes

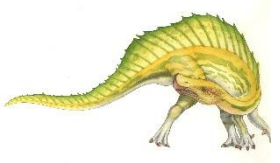
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Communication in Client–Server Systems

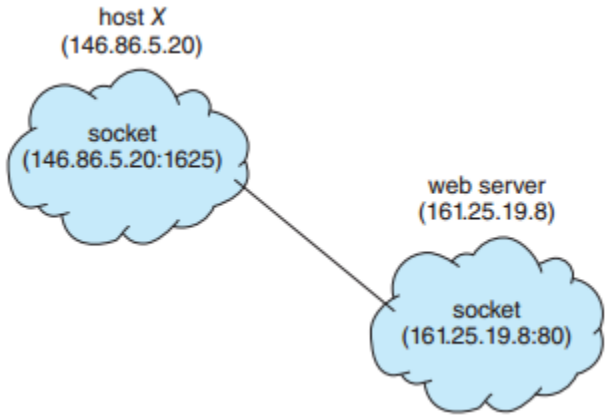
- Shared memory ve message passing, clint-server sistemlerde de prosesler arası iletişim için kullanılabilir. Client-server sistemlerde prosesler arası iletişim için diğer iki yöntem **sockets** ve **remote procedure calls (RPCs)** yöntemleridir.

Sockets

- Bir soket haberleşme için uç noktayı (endpoint) tanımlar.
- Bir ağ üzerinde haberleşen iki prosesin her biri bir sokete sahiptir. Habeleşme bir çift soket arasında olur.
- Bir soket, **IP** adresi ve **port** numarasıyla tanımlanır.
- Sunucu, bir portu dinleyerek gelen istekleri bekler. Sunucuya bir istek geldiğinde bağlantıyı tamamlamak için istemci soketinden bir bağlantı kabul eder.
- Spesifik hizmetleri (SSH, FTP ve HTTP gibi) uygulayan sunucular, bilindik portları (well-known ports) dinler. (bir SSH sunucusu port 22'yi dinler; bir FTP sunucusu port 21'i dinler ve bir web veya HTTP sunucusu port 80'i dinler).
- 1024'ün altındaki tüm portlar bilindik (well-known) kabul edilir ve standart hizmetleri uygulamak için kullanılır.
- Tüm prosesler için işletim sisteminin atadığı port numaraları farklı olmak zorundadır.

Sockets

- Bir istemci prosesi bir bağlantı için bir istek başlattığında, kendisine host bilgisayarı tarafından bir port atanır. Bu port 1024'ten büyük rasgele bir sayı olur.
- Şekildeki örnekte, 146.86.5.20 IP adresine sahip host X'teki bir istemci 161.25.19.8 adresinde bir web sunucusuyla (port 80'i dinleyen) bir bağlantı kurmak istemiştir, host X'e 1625'e portu atanmıştır.
- Bağlantı (146.86.5.20:1625) ile (161.25.19.8:80) soket çiftinden oluşur.
- **Tüm bağlantılar tekil(unique) olmalıdır**. Bu nedenle, bu örnek için host X'te başka bir proses aynı web sunucusuyla başka bir bağlantı kurmak isterse, buna 1024'ten büyük ve 1625'en farklı bir port numarası atanacaktır.



Sockets in Java

- Java üç farklı socket türü sağlar:
  - Connection-oriented (TCP)
  - Connectionless (UDP)
  - MulticastSocket – birden çok alıcıya veri gönderilmesine izin verilir.

Sunucu Örneği - Sockets in Java

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

İstemci Örneği - Sockets in Java

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Remote Procedure Calls (RPCs)

- Soket ile haberleşme yaygın ve etkili olmasına rağmen dağıtık prosesler arasında düşük seviyeli bir iletişim biçimi olarak kabul edilir. Bunun bir nedeni, soketlerin iletişim kuran evreler arasında yalnızca yapılandırılmamış byte akışına izin verilmesidir. Verileri yapılandırmak istemcinin veya sunucu uygulamasının sorumluluğundadır.
- **Remote procedure calls (RPC)** daha yüksek düzeyde bir haberleşme yöntemidir.
- RPC ağ bağlantılı sistemlerde prosesler arasında procedür çağrılarını soyutlamak için tasarlanmıştır.
- IPC mesajlarının aksine, RPC iletişiminde alınıp verilen mesajlar iyi yapılandırılmıştır ve bu nedenle artık sadece veri paketleri değildir. Her mesaj, uzak sistemdeki bir portu dinleyen bir RPC arka plan programına (deamon) adreslenir ve her biri yürütülecek fonksiyonu ve bu fonksiyona iletilecek parametreleri belirten bir tanımlayıcı (identifier) içerir.

Remote Procedure Calls (RPCs)

- Örneğin bir sistem kendi mevcut kullanıcılarının diğer sistemler tarafından listelemesine izin vermek isterse, bu sistemde bir porta bağlı böyle bir RPC'yi destekleyen bir arka plan programı (örneğin, 3027 portu) olacaktır. Herhangi bir uzak sistem, sunucudaki 3027 portuna bir RPC iletisi göndererek mevcut kullanıcıların listesini elde edebilir.
- RPC'lerin semantiği, bir istemcinin yerel olarak bir prosedürü çağırdığı gibi uzaktaki bir ana bilgisayardaki prosedürü çağırmasına olanak tanır.
- RPC sistemi, istemci tarafında bir stub sağlayarak haberleşmenin gerçekleşmesine izin veren ayrıntıları gizler.
- Tipik olarak, her bir uzak prosedür için ayrı bir stub mevcuttur. İstemci bir uzak prosedürü çağırdığında, RPC sistemi uygun stub ı çağırır ve uzak prosedüre sağlanan parametreleri iletir.

Remote Procedure Calls (RPCs)

- Bu stub, sunucudaki porta konumlanır ve parametreler paketlenir (marshalling). stub daha sonra, message passing kullanarak sunucuya bir mesaj iletir.
- Sunucu tarafındaki benzer bir stub bu mesajı alır, parametreleri çıkarır ve sunucudaki prosedürü başlatır. Gerekirse, dönüş değerleri aynı teknik kullanılarak istemciye geri gönderilir.
- Windows sistemlerinde stub kodu, istemci ve sunucu programları arasındaki arayüzleri tanımlamak için kullanılan Microsof Interface Definition Language(MIDL) dilinde derlenir.
- Ele alınması gereken bir konu istemci ve sunucu makinelerinde veri temsilindeki farklılıklarla ilgilidir.
  - 32 bit tam sayıların temsilini düşünün: bazı sistemler **big-endian** (önce en önemli byte tutulur) bazıları **little-endian** (önce en önemsiz byte tutulur) olarak bilinir
  - Mimari farklılıkları sorununu çözmek için, birçok RPC sisteminde makineden bağımsız bir veri temsili - External Data Representation (XDL) kullanılır.

Semantic - Remote Procedure Calls (RPCs)

- RPC’de bir diğer önemli konu çağrının semantiğidir.
- Local prosedür çağrıları yalnızca extra durumlarda başarısız olurken, yaygın ağ hatalarının bir sonucu olarak RPC'ler başarısız olabilir veya birden çok kez çoğaltılabilir ve yürütülebilir.
  - Bunun önüne geçmek için mesajlar ***at most once*** semantiği yerine, ***exactly once*** semantiğiyle iletilebilir.
  - ***at most once*** **–** Bu semantik her mesaja bir timestamp eklenerek uygulanabilir. Sunucu, tekrarlanan iletilerin algılandığından emin olmak için önceden işlediği iletilerin timestamp geçmişini tutar, geçmiş kaydı olan mesajlar tekrar geldiğinde bunlar tekrar işletilmez göz ardı edilir.
    - İstemci bir mesajı bir veya daha fazla kez gönderebilir ve yalnızca bir kez yürütüldüğünden emin olabilir.

Semantic - Remote Procedure Calls (RPCs)

- ***exactly once*** **–** bu semantik ile sunucunun isteği asla almama riskinin ortadan kaldırılması hedeflenir.
- Bunu başarmak için, sunucunun "***at most once***" protokolünü uygulaması gerekir, ayrıca istemciye RPC çağrısının alındığını ve yürütüldüğünü de bildirmesi (**acknowledge-ACK**) gerekir.
- İstemci, söz konusu çağrı için ACK'yı alana kadar her RPC çağrısını düzenli aralıklarla yeniden göndermelidir.

Binding - Remote Procedure Calls (RPCs)

- Bir diğer önemli konu istemci ile sunucu portlarının nasıl bağlanacağı (binding) konusudur. Bağlantının oluşturulması için istemci sunucudaki port numaralarını nasıl bilecektir?
- İşletim sistemi genellikle istemci ve sunucuyu bağlamak için bir buluşturma (**rendezvous** also called **matchmaker**) hizmeti sağlar. Rendezvous mekanizması ile binding dinamik olarak gerçekleştirilir:
  - Tipik olarak, bir işletim sistemi, sabit bir RPC portunda bir buluşturma arka plan programı (rendezvous daemon) sağlar.
  - Bu arka plan programla bir istemci, RPC'nin adını içeren bir mesaj göndererek RPC'nin port adresini talep eder,
  - Port numarası döndürülür ve işlem sona erene kadar RPC çağrıları bu porta gönderilebilir.

Execution of RPC