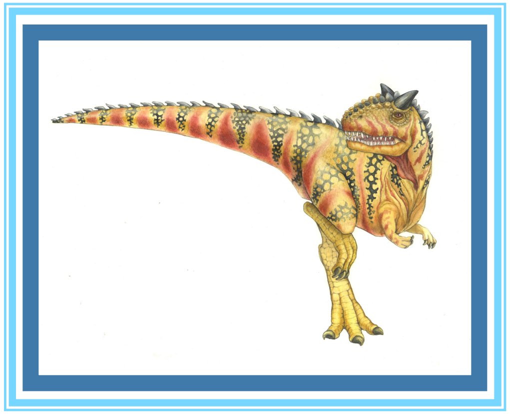
Chapter 3: Processes



Silberschatz, Galvin and Gagne ©2013

Chapter 3: Processes

- Process Concept
- Process Scheduling(İşlem programlama-zamanlama)
- Operations on Processes (Süreç üzerindeki işlemler)
- Interprocess Communication (İşlemler arası haberleşme)
- Examples of IPC Systems
- Communication in Client-Server Systems

Silberschatz, Galvin and Gagne ©2013

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

- İşlem kavramını (çalışmakta olan bir program) tanıtmak
- İşlemlerin pek çok özelliklerini tanıtmak: zamanlama, oluşturma, sonlandırma ve iletişim
- İstemci-sunucu sistemlerinde iletişimi açıklamak

Process Concept

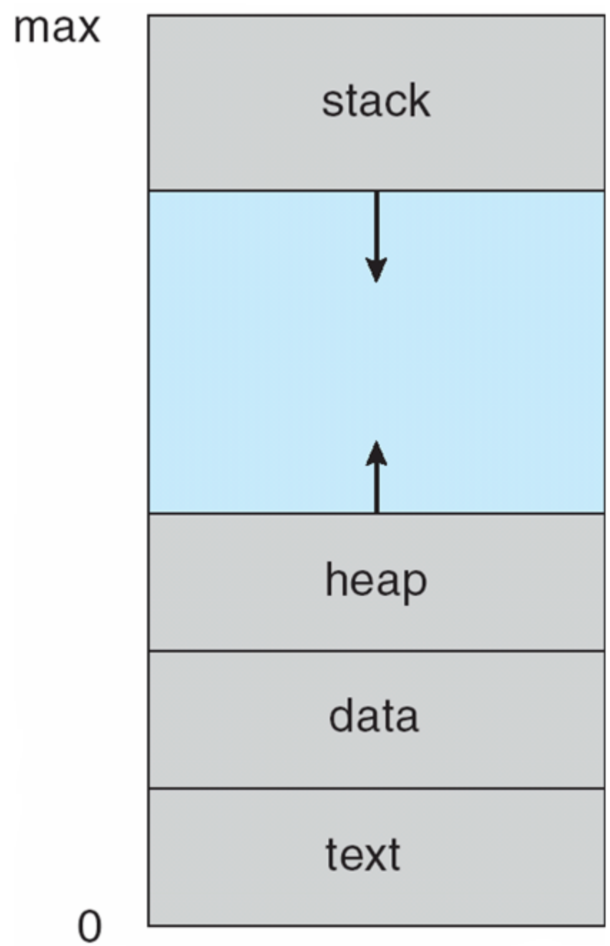
- An operating system executes a variety of programs:
  - Batch system – **jobs** (peşpeşe tanımlanan çağrılar ve program)
  - Time-shared systems – **user programs** or **tasks** (zaman paylaşımlı kullanıcı programları ve görevler… Tek çekirdekte aynı zamanda 1 işlem)
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion (RAM'da yüklü iş)

Silberschatz, Galvin and Gagne ©2013

Process Concept

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
  - (Kod akışı: program sayacı, hangi işin çalışacağını gösterir, yığında lokal değişken fonksiyon parametresi, dönüş adresi, veri bölümünde global değişkenler gibi, heap: çalışma anındaki dinamil bellek)

Process in Memory



Process Concept (Cont.)

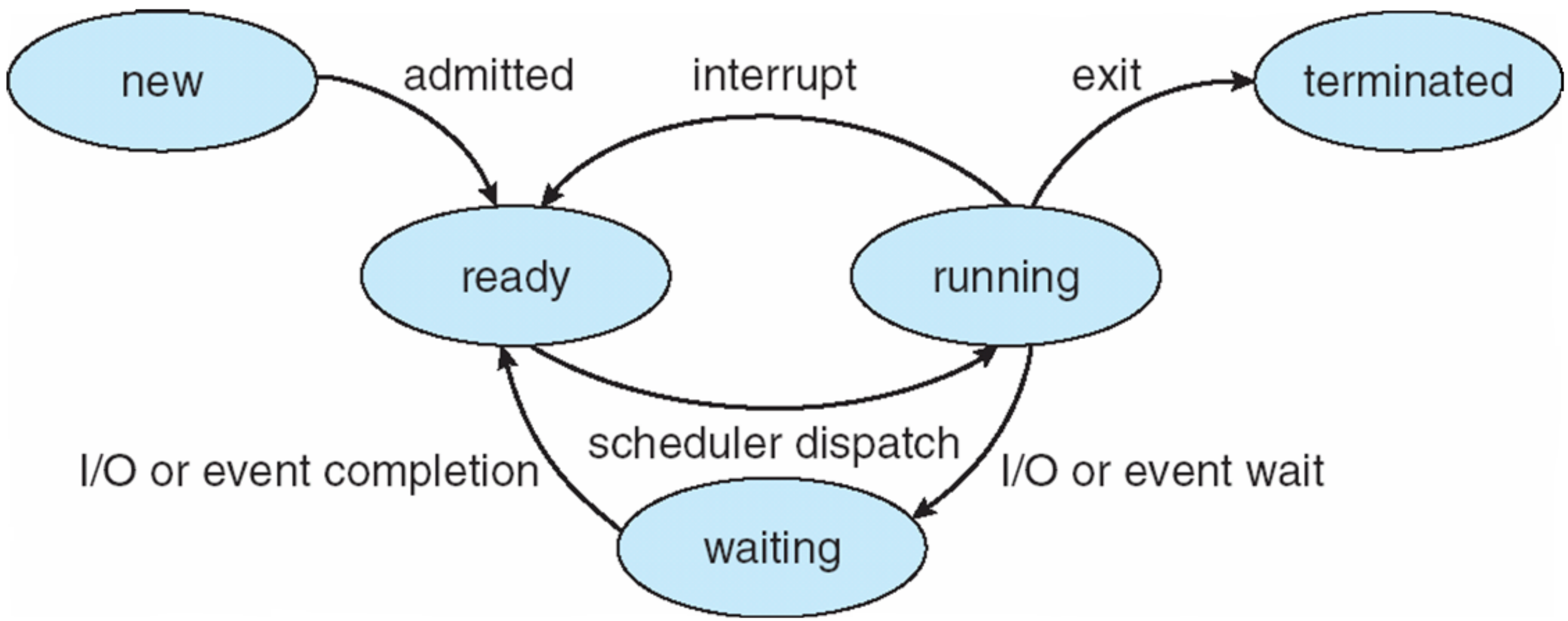
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

  - Passive process diskte, aktif bellekte
  - Bir program içerisinde birden fazla process olabilir

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

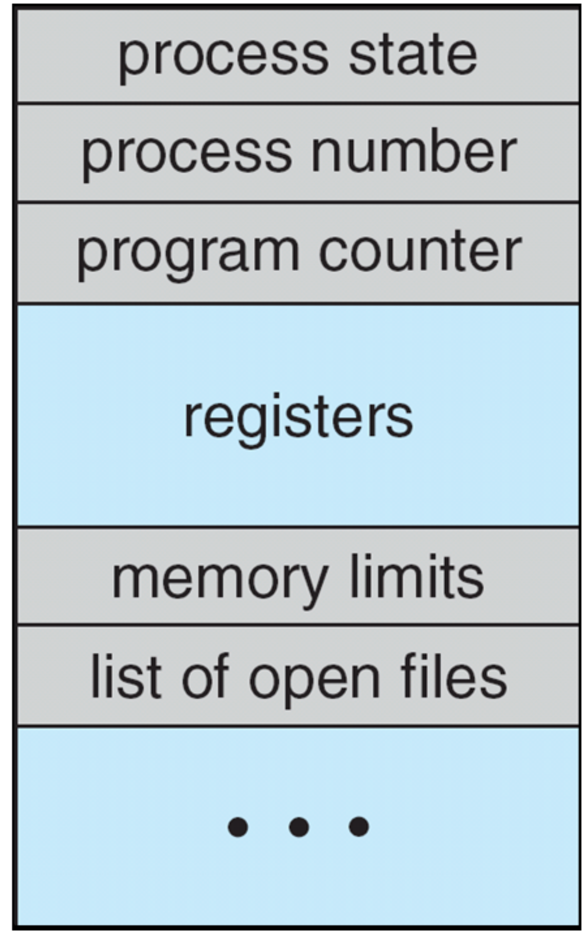


Process Control Block (PCB)

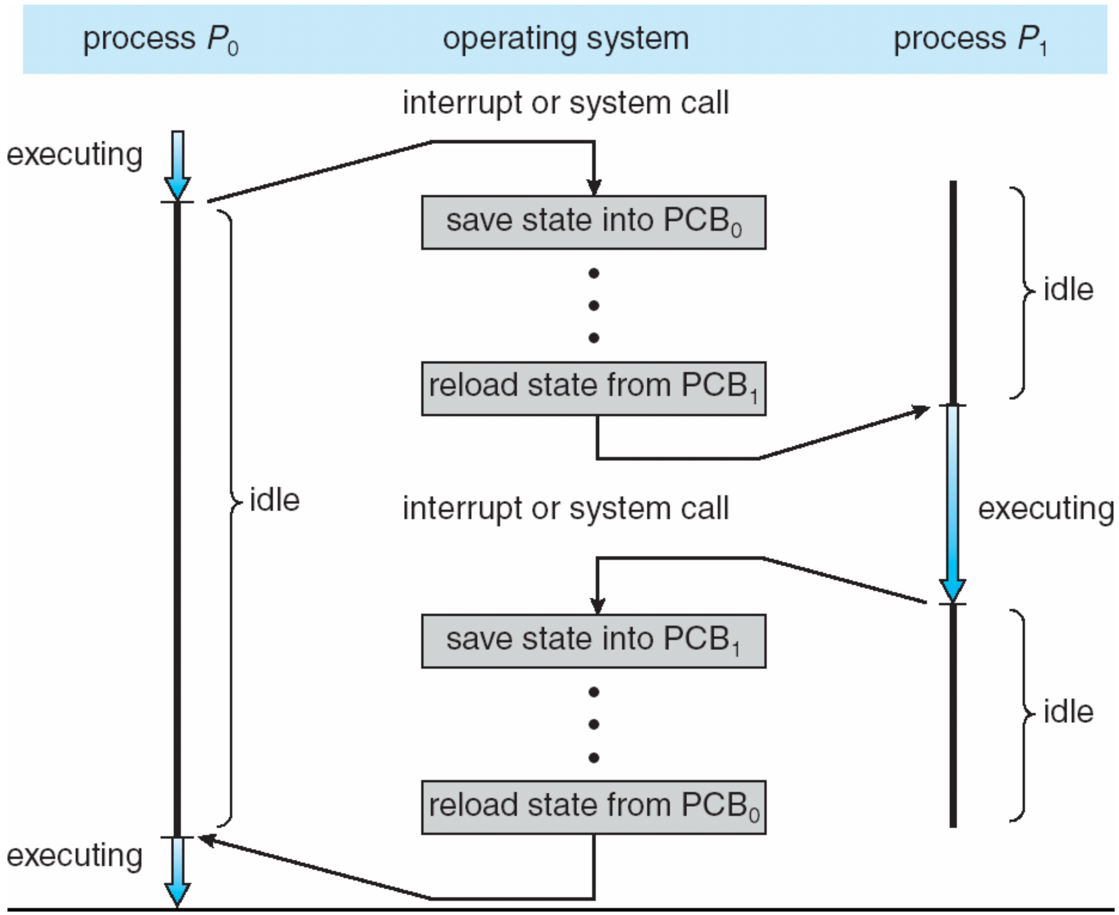
Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc : çalışıyor, bekletiliyor
- Program counter – location of instruction to next execute (koddaki yer)
- CPU registers – contents of all process-centric registers(registerda neler var)
- CPU scheduling information- priorities, scheduling queue pointers (kuyruk sırası)
- Memory-management information – memory allocated to the process (bellekte ayrılav yer)
- Accounting information – CPU used, clock time elapsed since start, time limits (çalışma süresi)
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



Threads

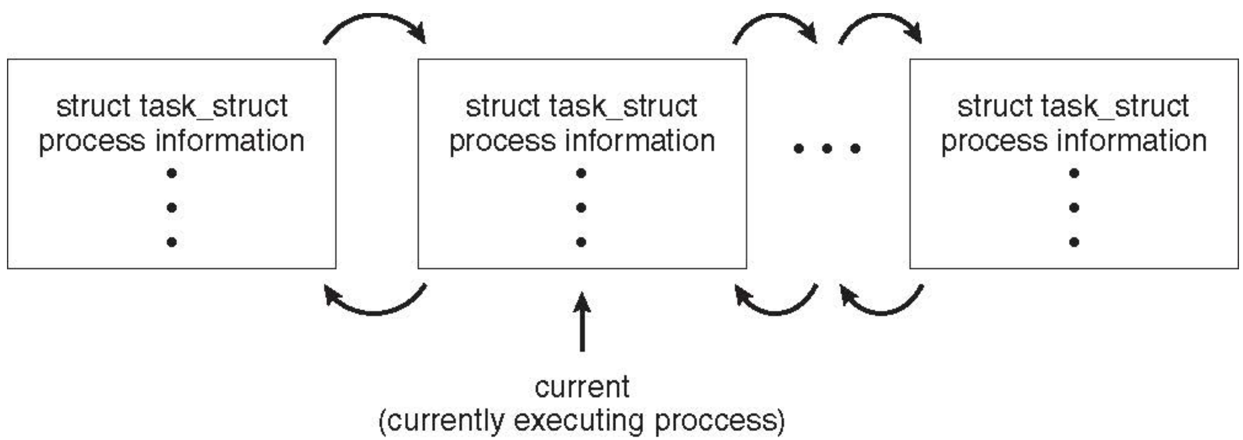
- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

- Process’lerin alt işlemleri(alt parçalara bölünmesi)
- Örn: Mail yazarken, gelen mail var mı kontrol et.

Process Representation in Linux

Represented by the C structure `task_struct`

```
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 */
```

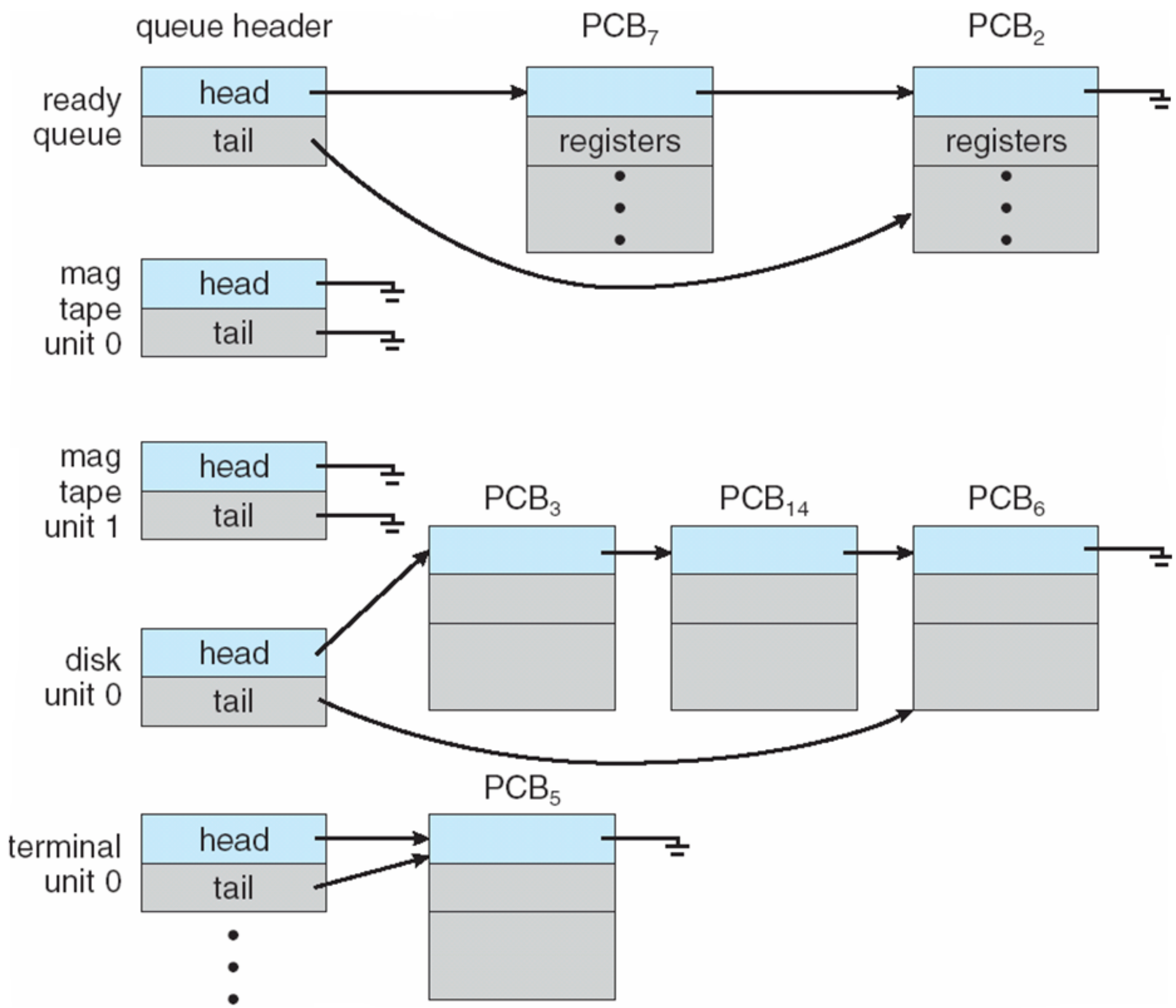


Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system (işletim sistemine ait işlemler , batch gibi)
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute(bellekte bekletilenlerin sırası)
  - **Device queues** – set of processes waiting for an I/O device(Cihazlardan bilgi bekleyenler…)
  - Processes migrate among the various queues

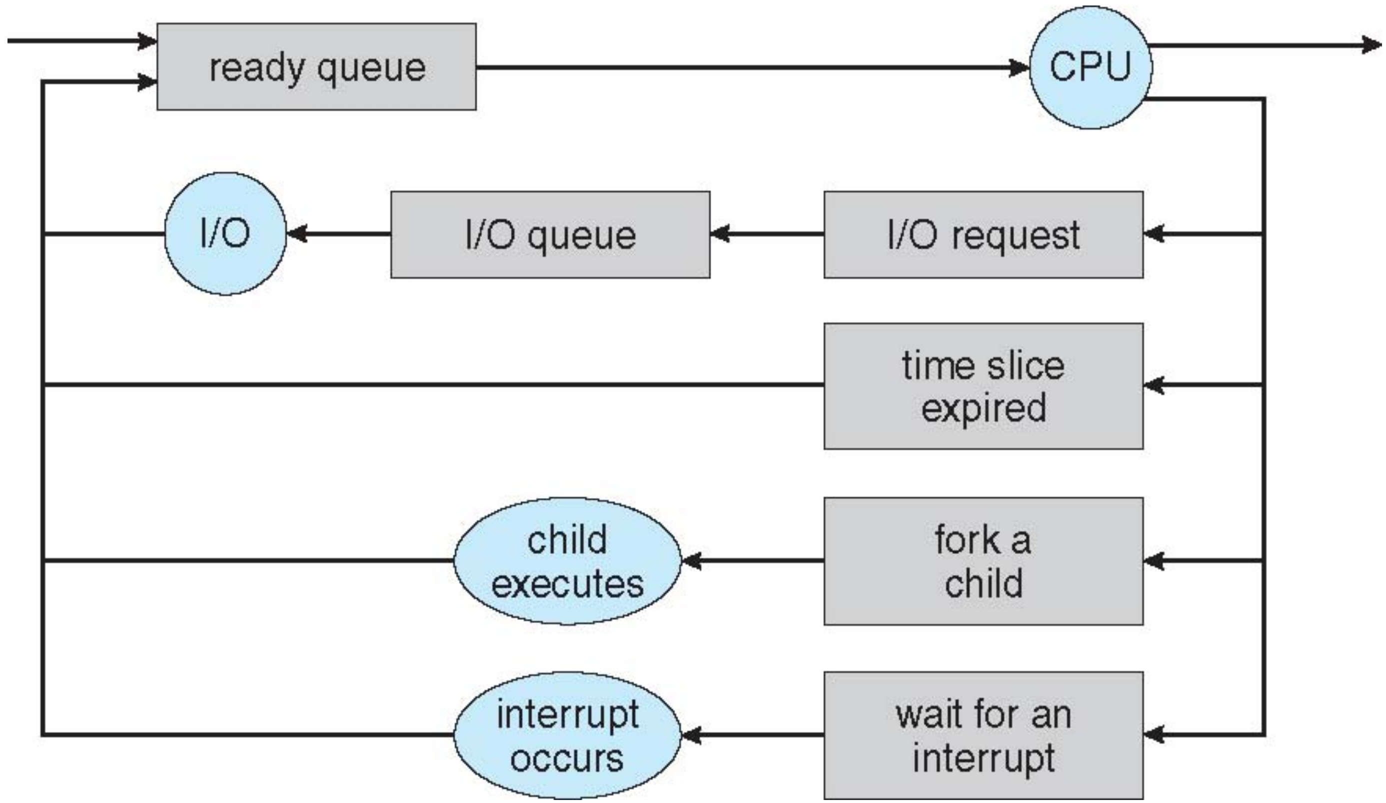
  - Process’ler hangi sırayla çalışacak?

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



Schedulers(Düzenleyici)

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)

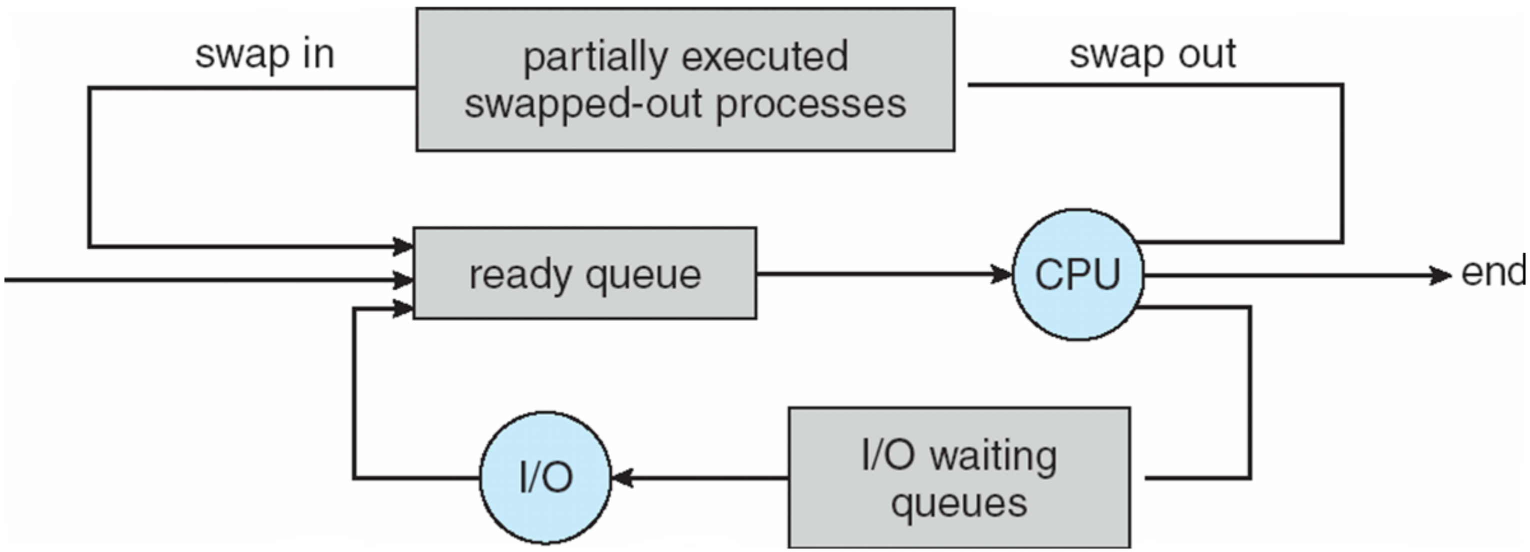
  Kısa suren işlemler için düzenleyici(genellikle sistem işlemleri)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

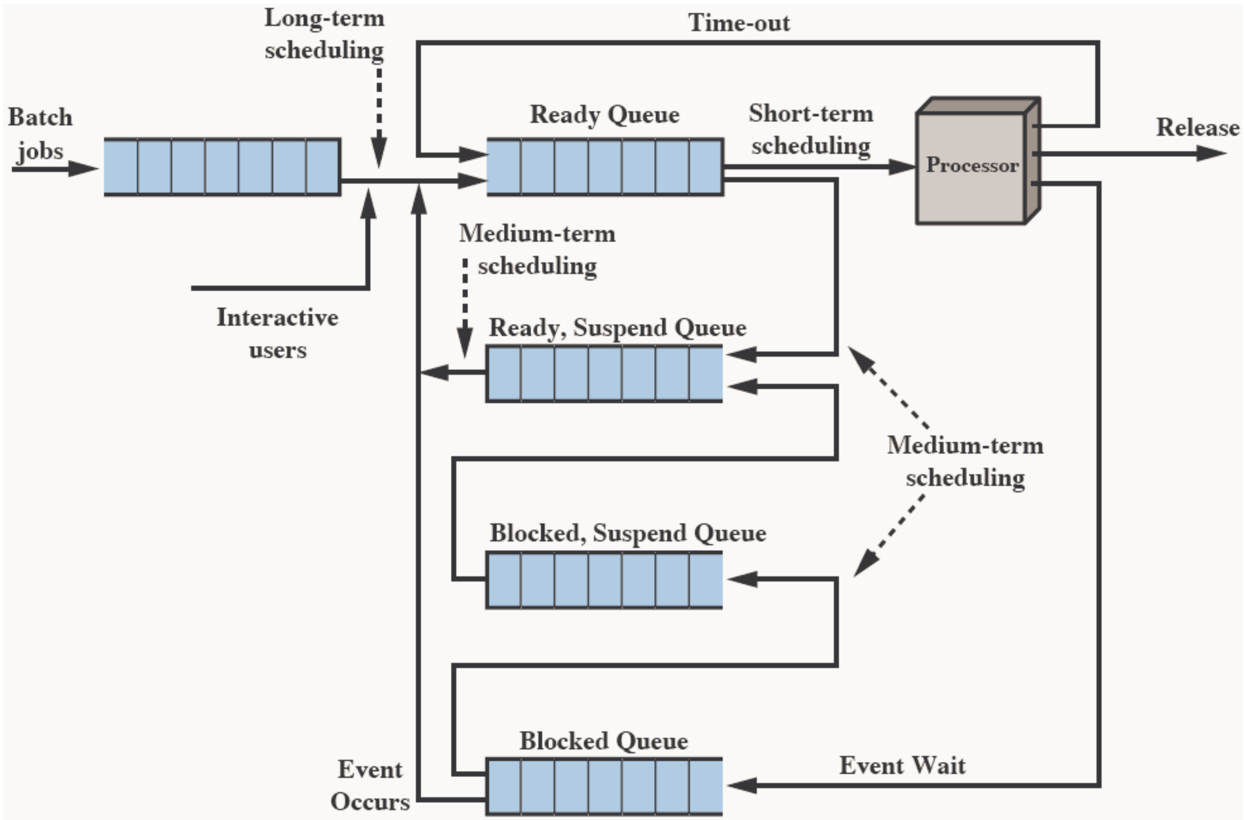
  Uzun zaman alanlar için ayrı bir düzenleyici(I/O işlemleri, disk yedekleme)
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

- Long Term (Job) Düzenleyici: Sisteme dahil olan proseslerden hangilerinin hazır kuyruğuna seçileceğine karar verir.
- Short Term (CPU) Düzenleyici: Hazır kuyruğundan hangi prosesin seçilip CPU’ya gönderileceğine karar verir.

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**
  - **Diskin RAM gibi kullanıldığı düzenleyici**




Long-term
scheduling
Time-out
Batch
jobs
Ready Queue
Short-term
scheduling
Processor
Release
Medium-term
scheduling
Interactive
users
Ready, Suspend Queue
Medium-term
scheduling
Blocked, Suspend Queue
Blocked Queue
Event Wait
Event
Occurs

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

İki tip çalışan process: foreground: üstte sistem kaynaklarını kullanan görünür uyg. background: arkada açık olan sistem kaynağı elinden almış uyg. Altta çalışan uyg diğer uygulamalara servis sunabilir.

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

PCB ‘de işlem listesine alma işlemi

Operations on Processes

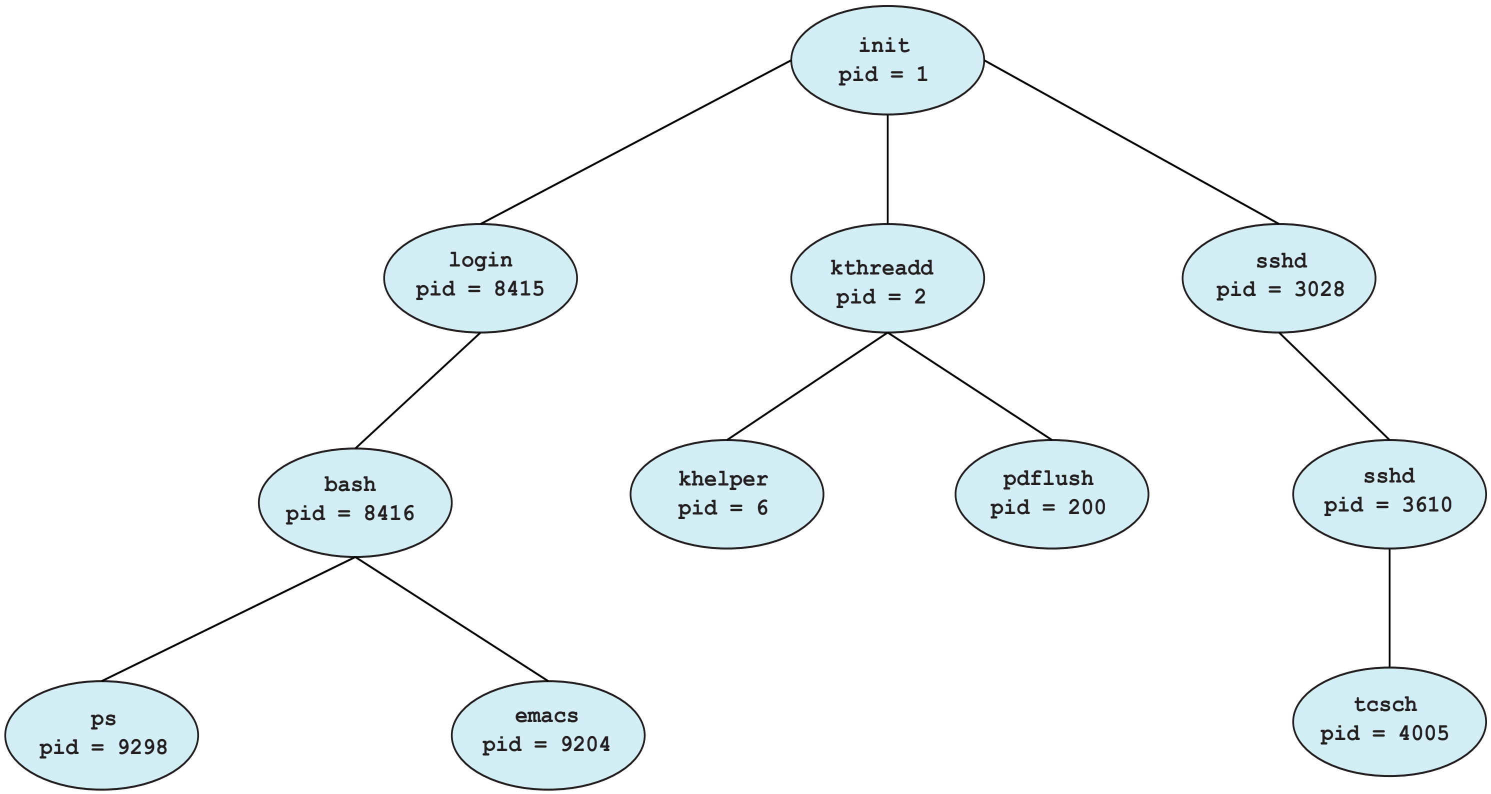
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources(kaynaklar paylaşılır)
  - Children share subset of parent's resources(çocuk, ailenin alt kaynağını paylaşır)
  - Parent and child share no resources(kaynaklar paylaşılmaz)
- Execution options
  - Parent and children execute concurrently(aynı anda çalışır)
  - Parent waits until children terminate(çocuk işlem bitene kadar aile bekler)

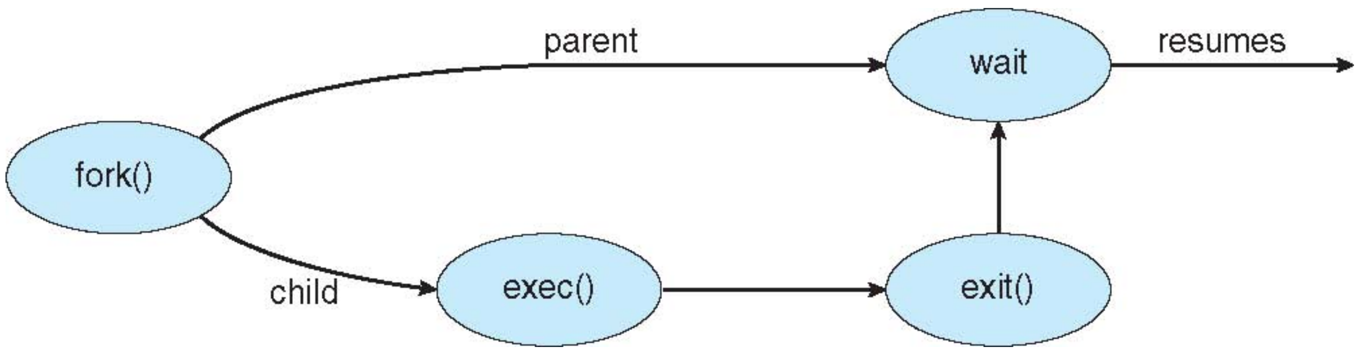
Her process'in bir parent'ı vardır. Her işlemi işletim sisteminde iş yapan kullanıcının işletim sistemi emriyle(ilk process) child process başlatır.

A Tree of Processes in Linux



Process Creation (Cont.)

- Address space
  - Child duplicate of parent (parent child'ı çağırır)
  - Child has a program loaded into it (Child program yükler)
- UNIX examples
  - **`fork()`** system call creates new process (sistem yeni bir iş oluşturur)
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program(exec sistemi çağrısı fork () process i çağrılarak yapılır. Exec yeni bir program çalıştırır)



C Program Forking Separate Process

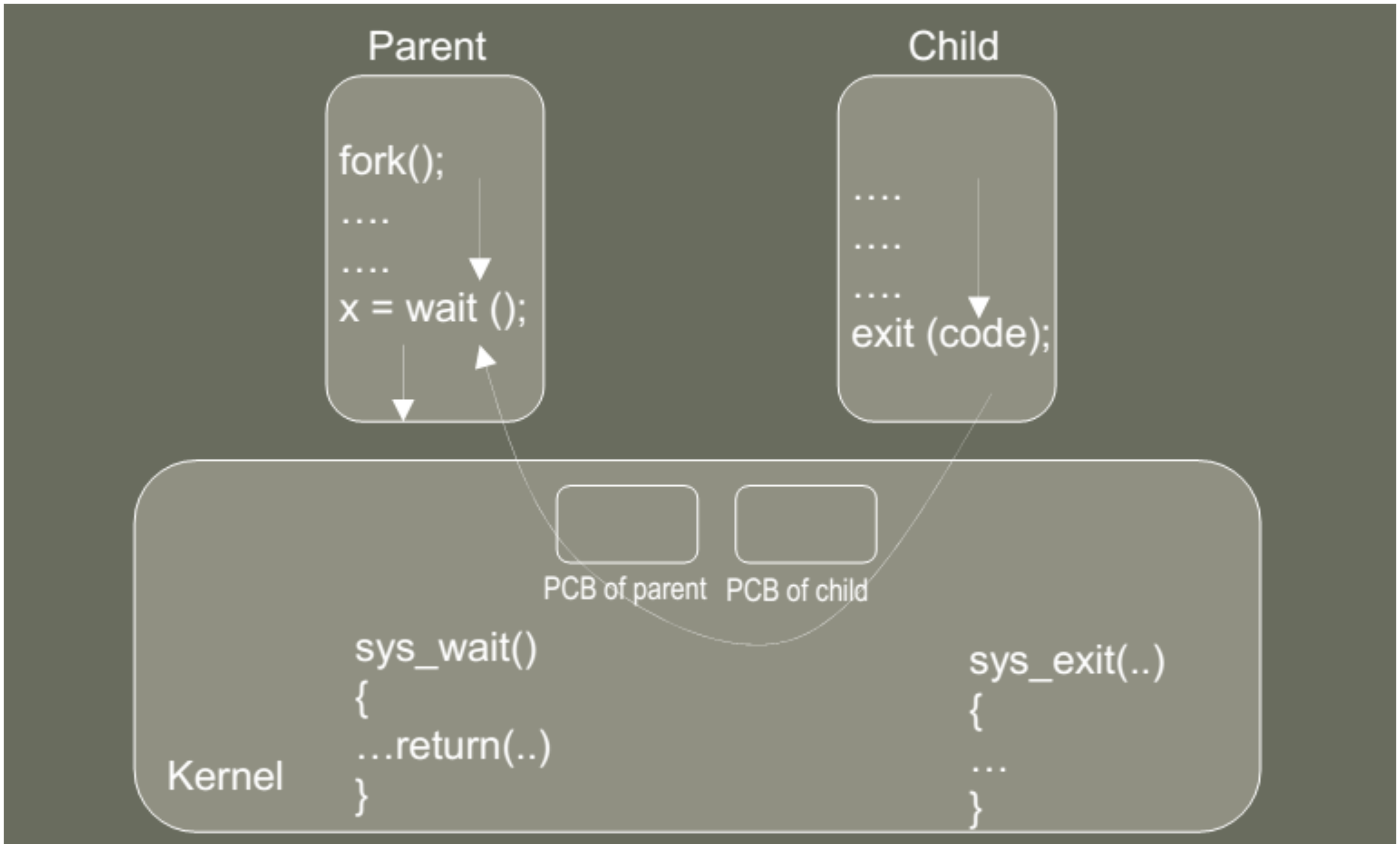
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```

Creating a Separate Process via Windows API

```
#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

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.(işlem bittikten sonra sistem çağrısı(exit komutu) ile sonlandırılır)
  - Returns status data from child to parent (via **`wait()`**(child process ten parent process’e(bekleme durumunda iken) durum bilgisi (return 0) gönderilir)
  - Process’ resources are deallocated by operating system(kaynaklar serbest bırakılır)
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so(exit olmadan parent , child abort edebilir):
  - Child has exceeded allocated resources()(Child kaynak kullanım sınırı aşmış olabilir)
  - Task assigned to child is no longer required(child ihtiyaç kalmamış olabilir)
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates(işletim sistemi parent’ı bitirir ise)


Parent
fork();
....
....
x = wait ();
Child
....
....
....
exit (code);
PCB of parent
PCB of child
sys_wait()
{
...return(..)
}
Kernel
sys_exit(..)
{
...
}

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.

  Parent sonlandırılırsa, tüm child lar sonlandırılır.(Basamaklı sonlandırma)
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

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

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie (parent bekleme yapmazsa)**
- If parent terminated without invoking **`wait`** , process is an **orphan(yetim-parent bekleme yapmadan işlemi bitirirse)**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)(Genellikle browserlar tek process üzerinden koşturulur)
  - If one web site causes trouble, entire browser can hang or crash

    Eğer bir web sitesi için problem oluştuğunda tarayıcı kilitlenebilir veya askıya alınabilir
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing