BIL 222 System Programming - Spring 2024

Instructor: Ammar Daskin

Hafta - 1

Elimizde OS işletim sistemleri var

Linux
Windows
MacOS
Android
IOS
Pardus
Debian
Xinu

Bir Linux var, bir de Unix var

Unix çok eski 70 lere gidiyor

MacOS da Unix destekliyor

Android → Üst tarafı Java
→ Alt tarafı Linux

processler nasıl cpu'da nasıl schedule ediyor

User permissionları nasıl veriliyor

File permission falan

Kernelde belirleniyor

Örneğin pardusta tamamen Türkçeleştirilmiş user-interface var. Dolayısıyla tamamen Türkçe görüyorum. Ama alt tarafta linux kernel kullanıyor. Debian'da ws de yine böyle oluyor.

Sistem programı nedir? Çekirdek sistem kütüphanelerini kullanarak program yazmaktır.

Kernel tarafından bize sunulan sistem kütüphaneleri var. Biz direk bu sistem kütüphanelerini kullanarak program yazacağız

Sistem programları direk olarak OS kernel ile irtibata geçiyor.

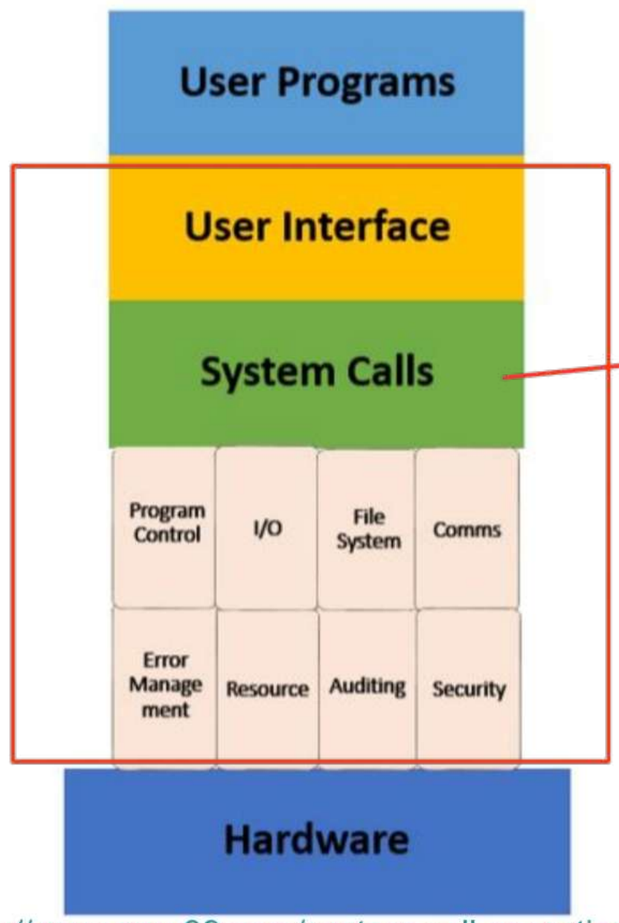
Shell, Text editor, debugger, system daemons, network server

Linux'ta sistem programı yazarken kullanılacak temel şeylerden bir system-call'

User printf yazdı → Bir şekilde kernel ile iletişime geçildi
Çünkü donanıma ekrana bir işlem yaptırılacak

System call → System Library'leri üzerinden
kernel ile iletişime geçip sonu yazdır demesi gerekiyor

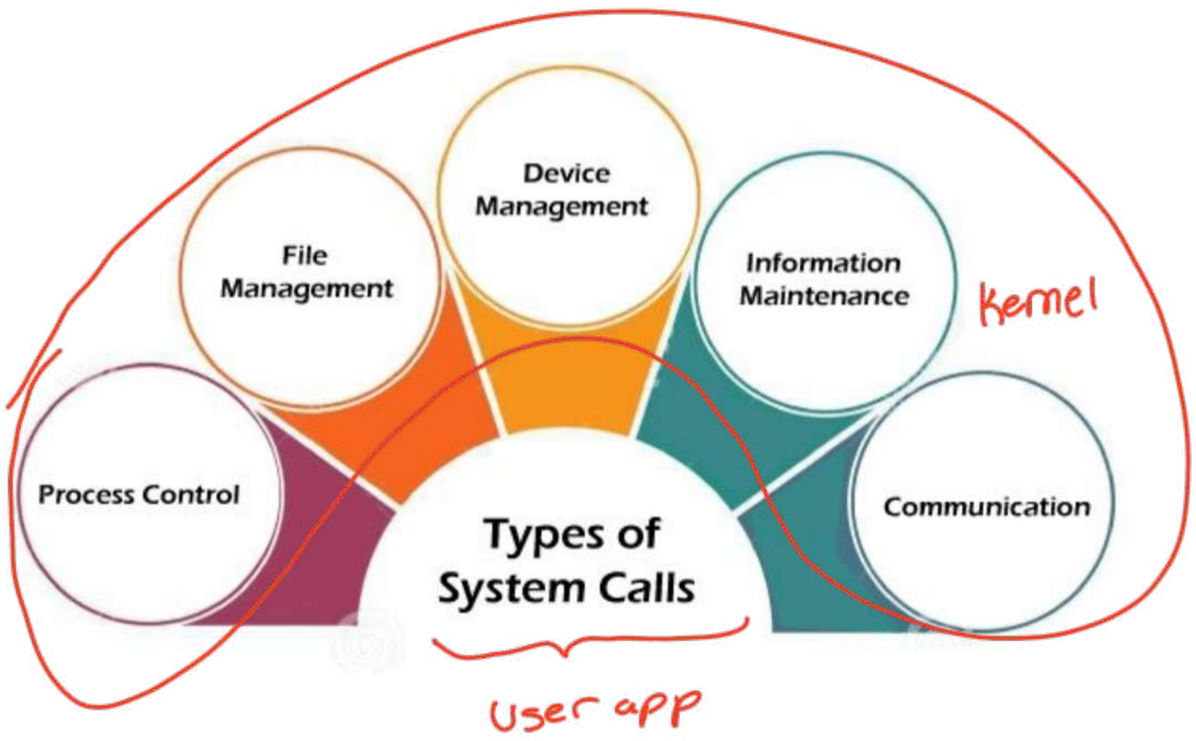
Computer OS Structure



OS
Kernel ile haberleşmeyi sağlıyor
Kernel hardware ile haberleşmeyi sağlar

https://www.guru99.com/system-call-operating-system.html#2

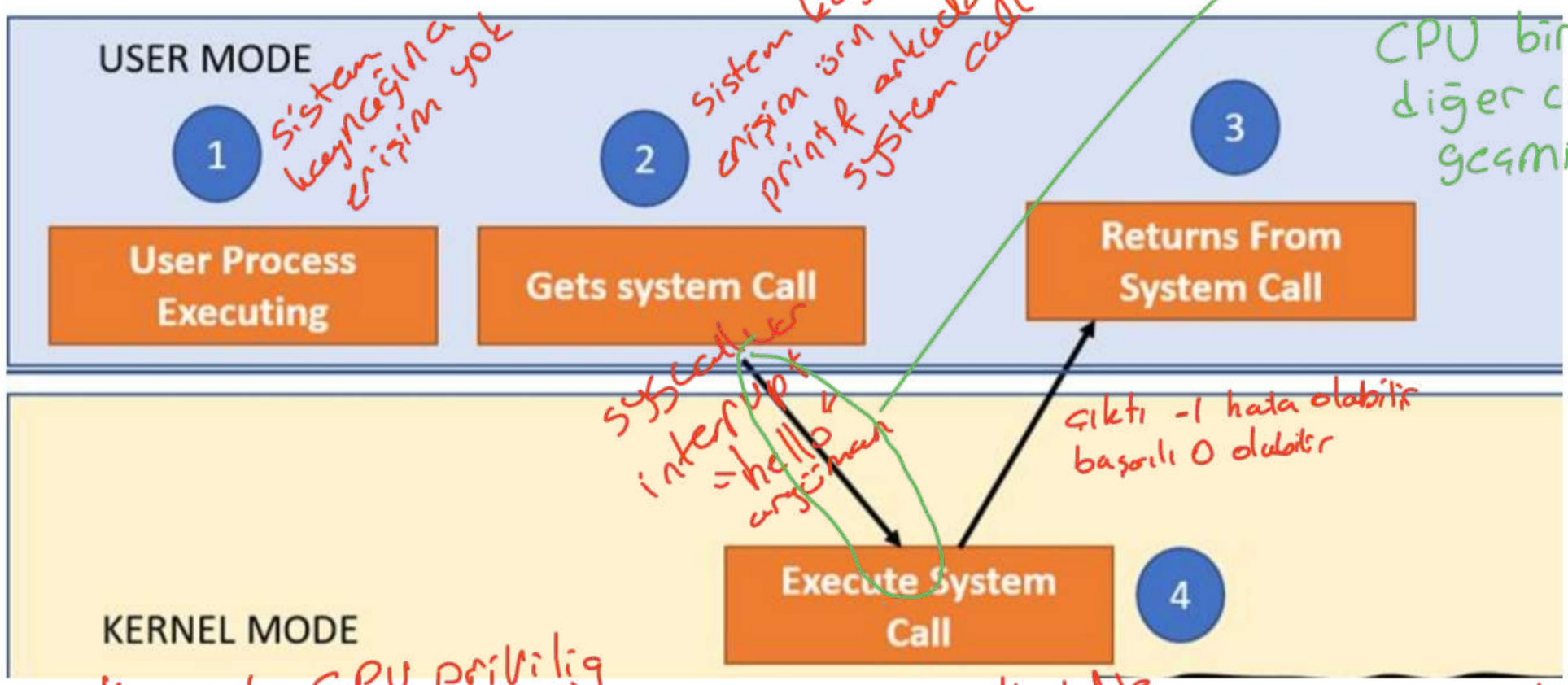
What type of system calls we can use in our system programs



https://www.javatpoint.com/system-calls-in-operating-system

OS dersinde tamamen kendin içini irdeleyip anlamaya çalışacaz.

OS services the system calls!

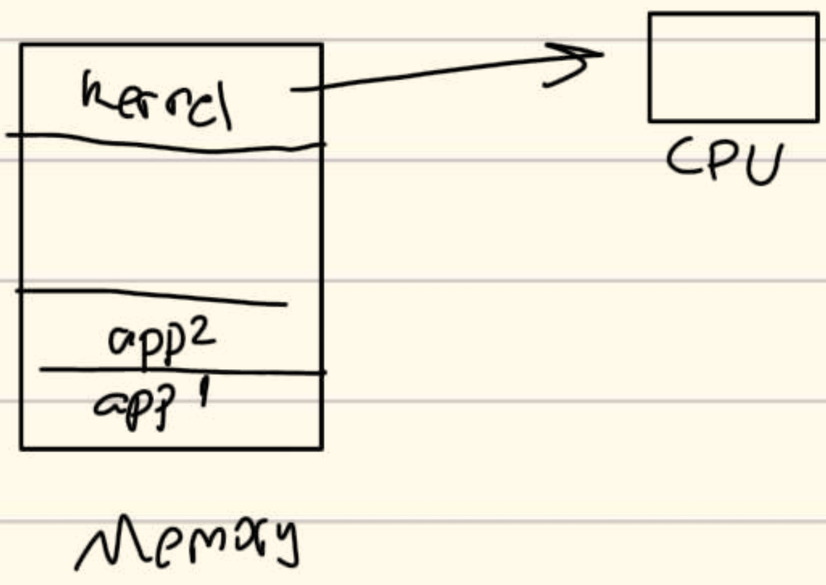


CPU'nun farklı farklı çalışma modları var. CPU nun genelde 2 modu var.

privilige (güvenli) mod
Sistem library
network kartı port erişim
sistem kaynakları.

user mod
CPU burada User applar toplama çıkarma gibi basic matematiksel işlemler yapar.

Privilig modda CPU'da çalışırken hem kernel'in datasına erişebiliyor-sunuz hem de user in datasına



OS bütün sistem kaynak-ların kontrol ediyor

API (Application Programming Interface) Bir programdan diğer programı nasıl çağırırsın onu belirliyor.

Sistem programlarını yazarken örneğin benim fonksiyonum write

write (... , ...)

bu şekilde API olarak tanımladım. Çağırma prototipi böyle.

Belirli API lere uyarak tasarlanınca bu şekilde portability sağlanıyor

POSIX standartları

Portable Operating System Interface → İşletim sistemi gereksinimleri

→ Hangi API leri sağlamalı

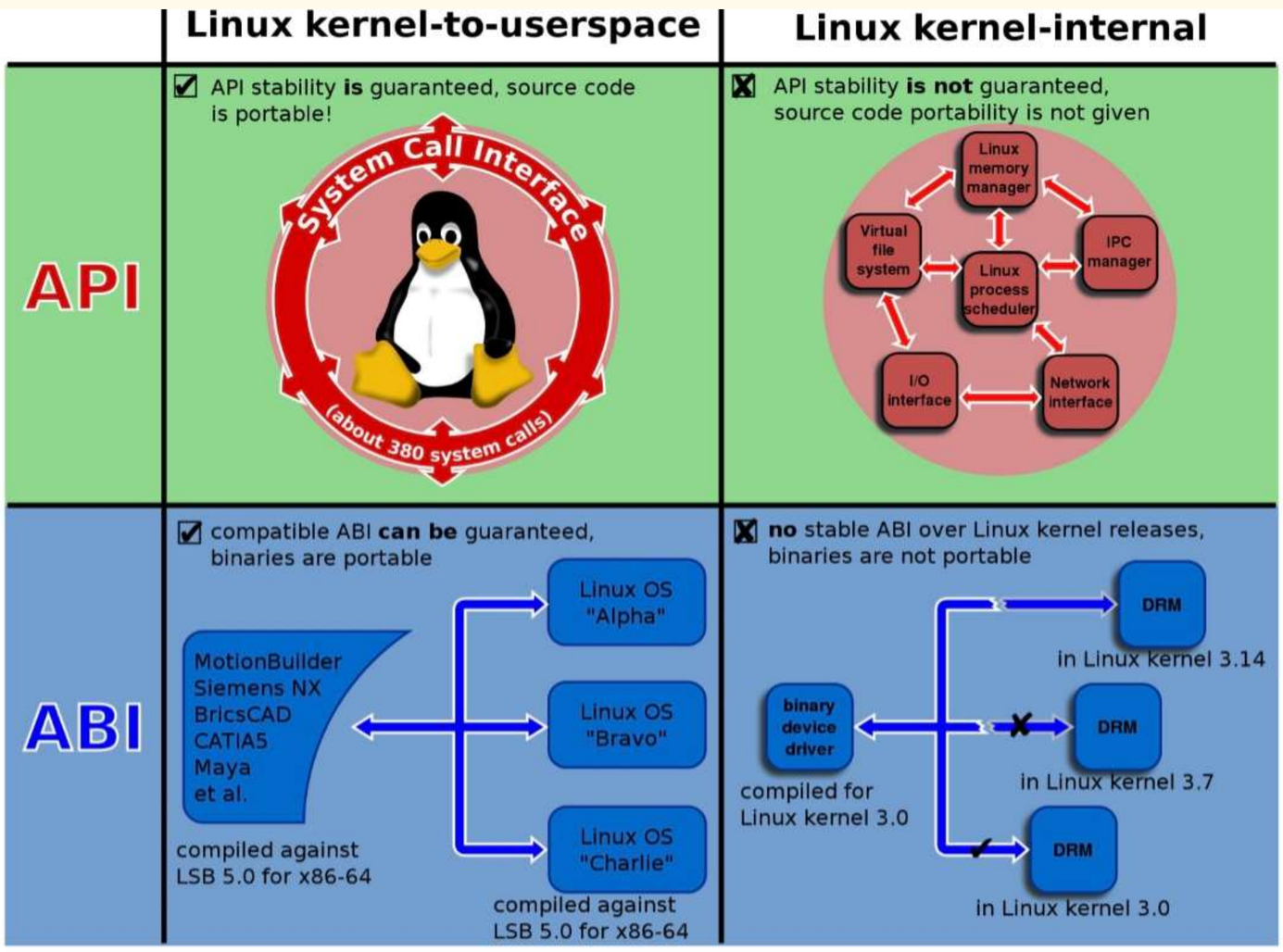
belirler. Buna uyan POSIX desteklemiş olur.

MacOS posix uyumlu olduğu için genelde Linux'da yazılan MacOS da da çalışır.

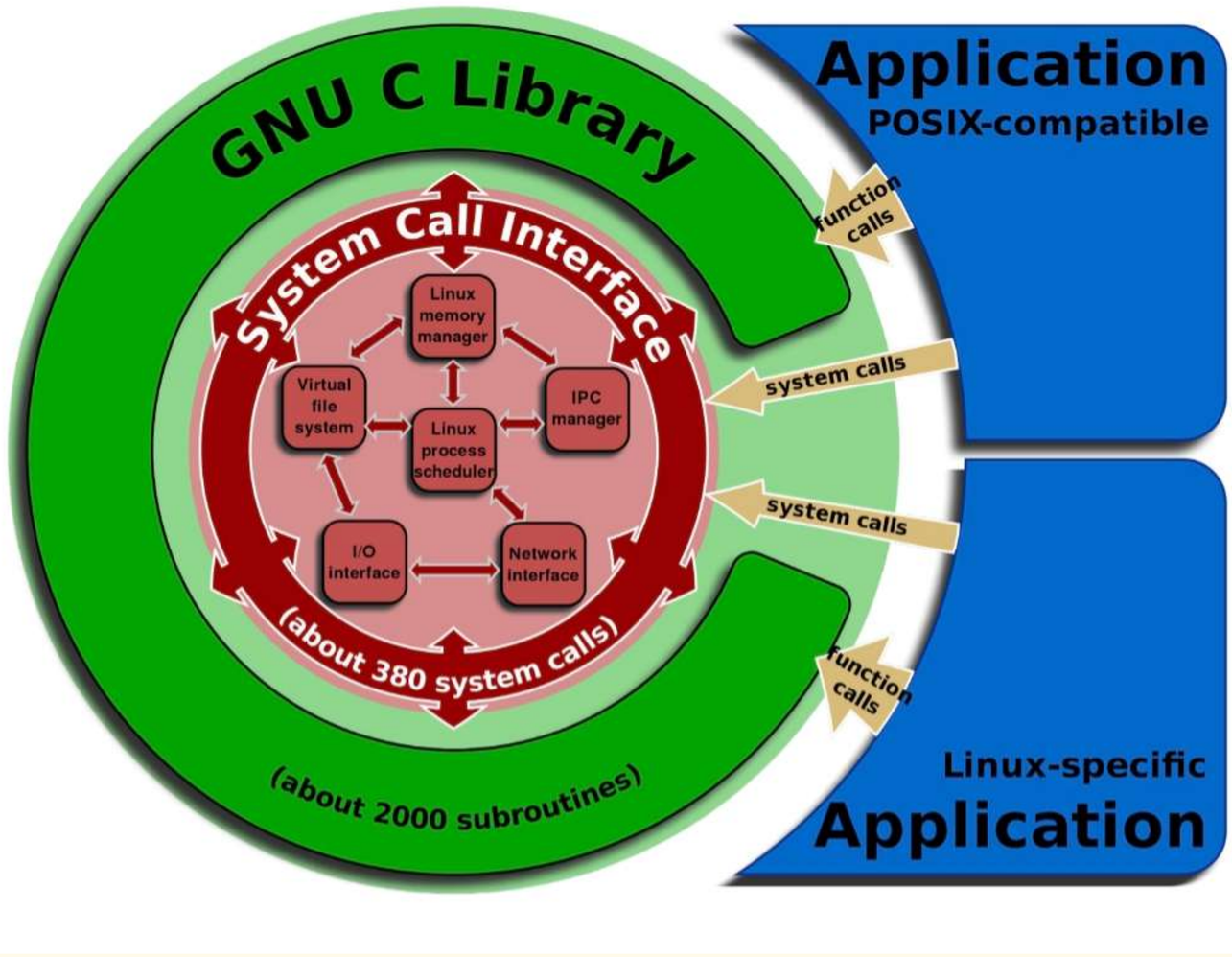
Android tam anlamıyla POSIX desteklemez. Linux dağıtımı değildir Linux çekirdeği kullanan ayrı bir işletim sistemidir. Kendi hafif C kütüphanesi olan BIONIC i kullanır

ABI (Application Binary Interface)

Bu biraz daha low level. Bir mimaride yaptığınız ABI diğer bir mimaride de çalışıyor.



https://en.wikipedia.org/wiki/System_call



C'de program yazarken bir tane source dosyanız oluyor. (kaynak dosyası)

```c
#include <stdio.h>
int main() {
        printf("hello world");
        return 0;
}
```

GNU Compiler Collection

`gcc -o aprogram main.c`

`./aprogram`

aprogram bir binary dosyası çalıştırılabilir bir dosya

- `-o <çıktı dosya ismi>` : Derlenen dosyanın adını belirler.
- `-Wall` : Tüm uyarıları gösterir.
- `-g` : debug hata ayıklama bilgisi ekler. gdb ile debug yapabilirsin
- `-c` : Sadece object dosyası üretir.
- `-I<klasör>` : Header dosyanın bulunduğu dizini belirtir.
- `-L<klasör>` : Kütüphanelerin bulunduğu dizini belirtir.
- `-l<kütüphane` : Bağlanacak kütüphaneyi belirtir.

Bitişik yazılıyorlar -lm gibi. matematik kütüphanesi

- `-std=c99` veya `-std=c11` : Hangi C standardında derleyeceğini belirtir.

main.c kaynak dosyasıdır (source file). Çalıştırılabilir dosya değildir. Bunlardan bir program dosyası aprogram gibi

Bir program dosyası memorye uygulamayı load etmek için tüm gerekli bilgilere sahiptir. Çalıştırılması için şunlara ihtiyaç duyar

- machine instructions
- initialized data içerisinde init data varsa onları içerecek
- List of library dependencies printf mesela stdio.h librarysinden geldi
- programın kullanacağı memory sectionları

Compiling a c program

```
$ cat main.c

#include <stdio.h>

int main() {

    printf("Hello World!\n");

    return 0;

}


$ gcc main.c -o main

$ ./main

Hello World!

$
```

Preprocessing:

- The preprocessor expands all preprocessor directives.

Parsing: ; falan unutursan hata veriyor

- The compiler parses the text file for function declarations, variable declarations, etc.

Assembly Generation: Assembly kodu üretir

- The compiler then generates assembly code for all the functions after some optimizations if enabled.

Assembler (compiling assembly code): Assembly kodundan binary kod üretir

- The assembler turns the assembly into 0s and 1s and creates an object file. This object file maps names to pieces of code.

Static Linking: Birden fazla dosya fonksiyon varsa bunların linklenmesi yapılıyor

- The linker takes a series of objects and static libraries and resolves references of variables and functions from one object file to another.
- The linker finds the main method and makes that the entry point for the function.
- The linker also notices when a function is meant to be dynamically linked.
- The compiler also creates a section in the executable that tells the operating system that these functions need addresses right before running.

Sistem kütüphaneleri linkleniyor

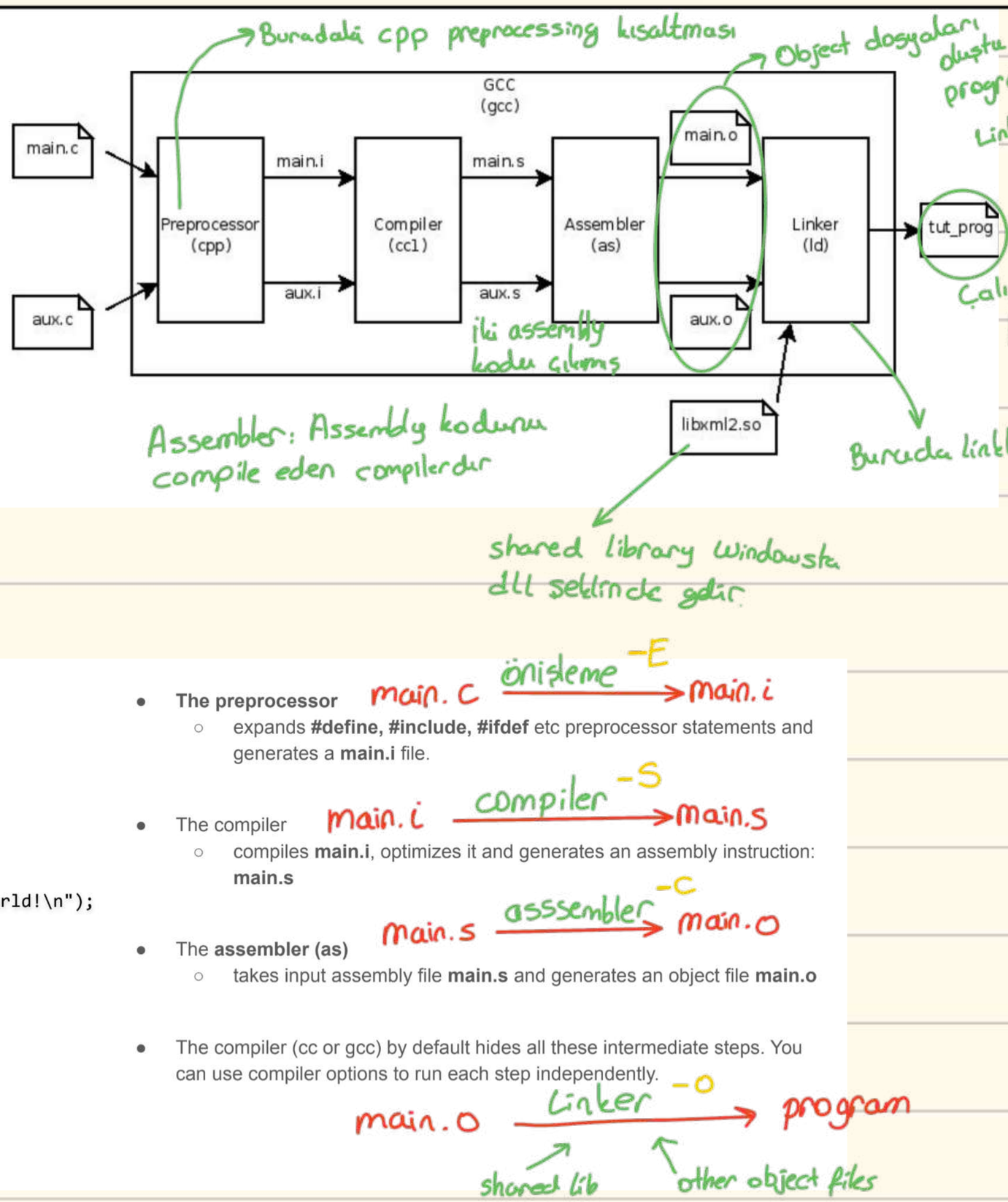
Dynamic Linking: As the program is getting ready to be executed, the operating system looks at what libraries that the program needs and links those functions to the dynamic library.

- The program is run.

∝ `gcc -E main.c -o main.i`

preprocess yaptiktan sonra bes # include <stdio.h> icerigini atde extend eder

```
$ gcc main.c aux.c -o tut_prog
$ ./main
Hello World!
$
```




```
$ cat main.c
#include <stdio.h>
int main() {
      printf("Hello World!\n");
      return 0;
}


$ gcc main.c -o main
$ ./main
Hello World!
$
```

- **The preprocessor** main.c —önişleme (-E)→ main.i
  - expands **#define, #include, #ifdef** etc preprocessor statements and generates a **main.i** file.
- The compiler main.i —compiler (-S)→ main.s
  - compiles **main.i**, optimizes it and generates an assembly instruction: **main.s**
- The **assembler (as)** main.s —asssembler (-c)→ main.o
  - takes input assembly file **main.s** and generates an object file **main.o**
- The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

main.o —Linker (-o)→ program (shared lib, other object files)

```
$ cat main.c
#include <stdio.h>
int main() {
      printf("Hello World!\n");
      return 0;
}


$ gcc main.c -o main
$ ./main
Hello World!
$
```

- $ gcc -E main.c > main.i
  - -E stops compiler after running preprocessor
- $ gcc -S main.c
  - -S stops compiler after assembly file generated
  - Assembler uses ".s" file to generate an object file
- $ gcc -c main.c
  - Generates object file main.o
  - main.o has undefined symbols like printf we don't know where it is placed
  - $ nm main.o
- $ gcc -o main main.c
  - Generates executable main
  - printf does not have a value yet until the program is loaded
  - $ nm main.o

Gcc options
https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html#Option-Summary

main.o gibi object dosyaları direk cat ile incelenmez. cat ile her sisteme özgü değişik çıktılar verir.

nm main.o // sembol table ı gösterir.

İçerisinde kullanmış olduğum semboller fonksiyonlar falan varsa

man nm : man'i kullanmak Manual page'i onları gösteriyor.

görüntülememizi sağlar

```
#include <stdio.h>
#define X 10
int print_integer(int);
int main(){
      int a = X;
      printf("Hello World!")
      print_integer(5);
      return 0;
}
```

declaration dur. definition DEĞİL!
bu fonksiyon henüz bir yerde tanımlanmadı.

> gcc -o a1 main.c

Başarısız
undefined reference to 'print_integer'

> gcc -c main.c

main.o object dosyası oluşur.

Burada hata vermez çünkü linker henüz çalışmadı.

aux.c

```
#include <stdio.h>

int print_integer(int a){
    printf("integer: %d\n", a);
    return 0;
}
```

gcc -o myprog main.c aux.c

bu şekilde direk program üretilebilir.

> gcc -o aux aux.c

Hata verir içinde main dosyası yok diye

> gcc -c aux.c

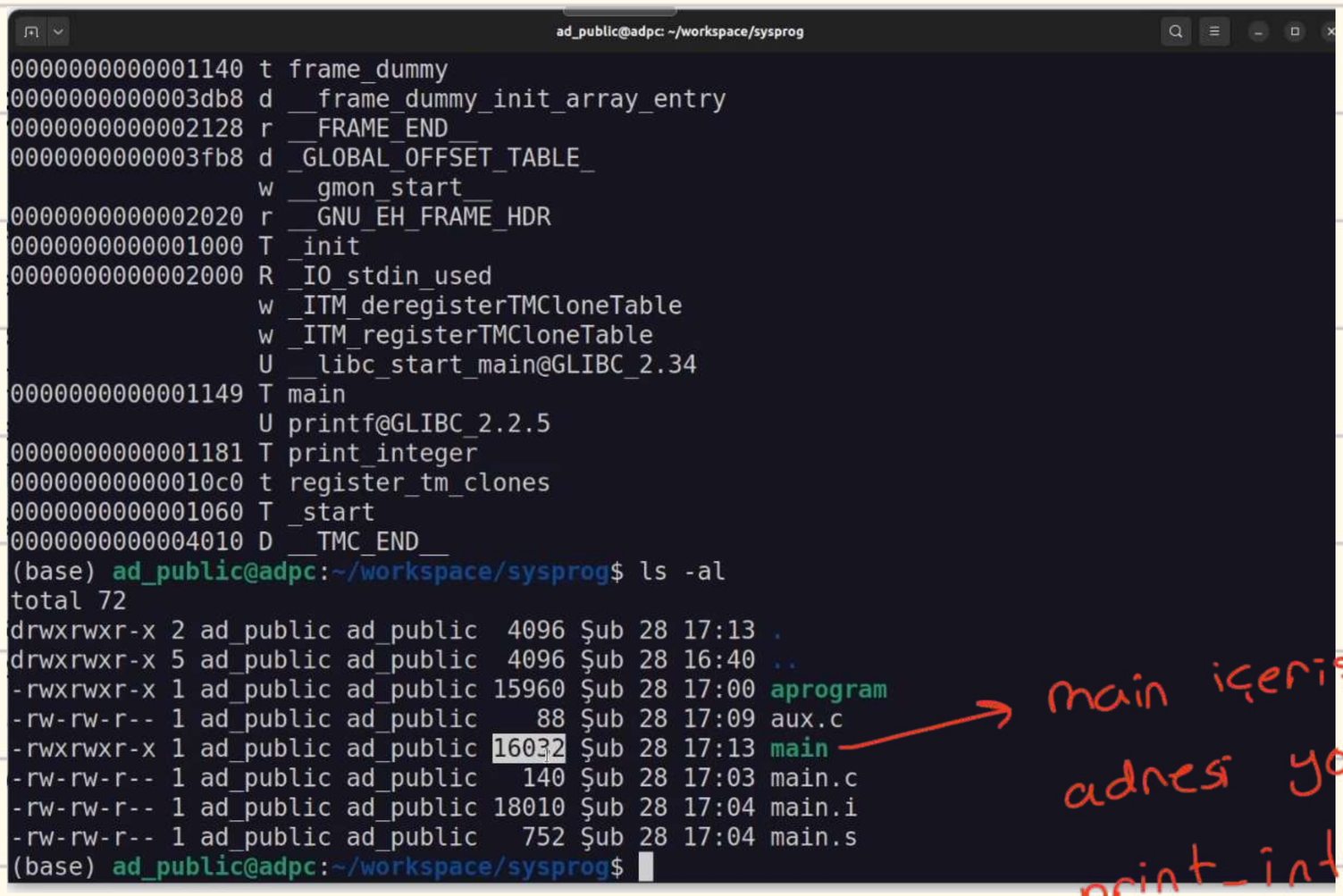
Bu bana bir tane object aux.o üretecek

> gcc -o main main.o aux.o

Böylece başardı şekilde çalışır

Linker aşamasında bu gibi ifadeler linkleniyor.
Bu static linkleme olarak adlandırılıyor çünkü derleme esnasında yapılır

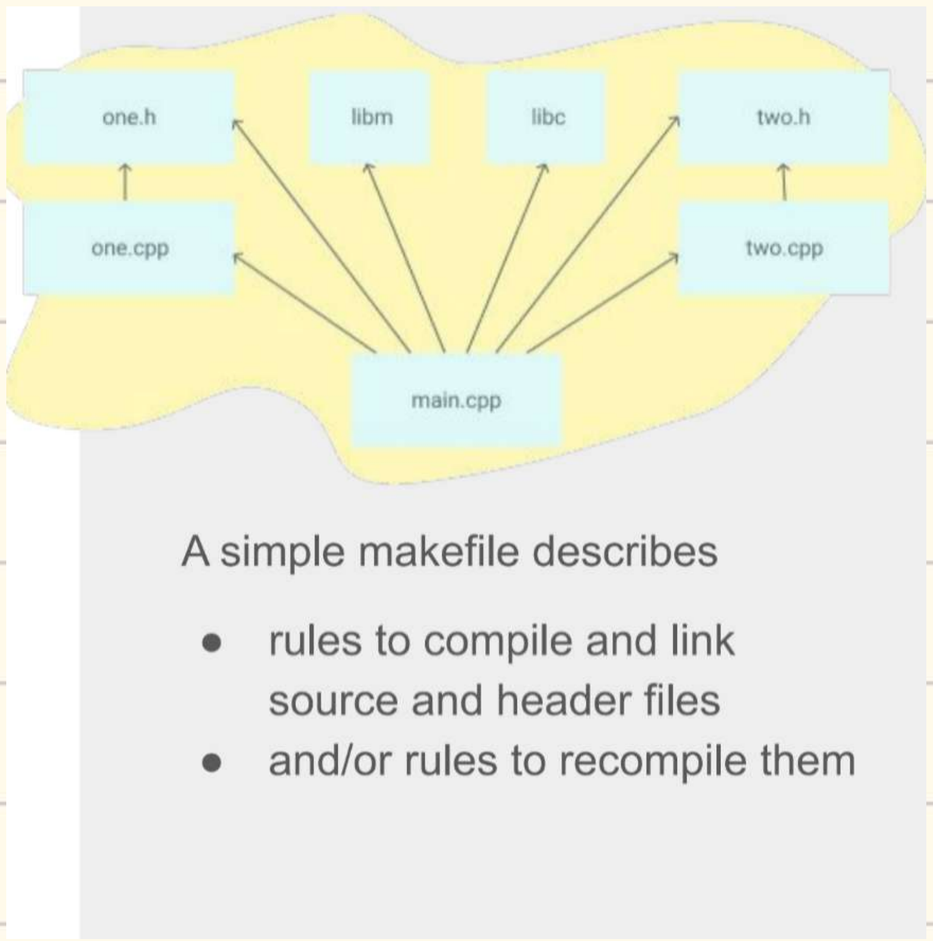
printf stdio.h shared libraryси içerisinde tanımlanmıştır
Bu dynamic linkleme olarak adlandırılır. Çünkü runtime esnasında linkleme gerçekleşir.



main içerisinde printf memoryde adresi yok
print_integer adresi var

Hafta - 2 Makefiles

https://makefiletutorial.com
https://www.gnu.org/software/make/manual/make.html#Introduction



A simple makefile describes

- rules to compile and link source and header files
- and/or rules to recompile them

- Bazen C dosyaları idare edilemicek kadar çok olabiliyor. Dolayısıyla compile ederken belli düzenlere sokmamız gerekebiliyor.

A simple Makefile

prerequisite : An input file to create target

recipe : An action (a terminal command)

```
target … : prerequisites …
    recipe
    …
    …
```

Makefile

target → prerequisites, şuan boş
action

```
hello:
    echo "Hello, World"
```

Bu şekilde bir kullanım ile baştan aşağı çalışır Makefile'ı arayıp çalıştırır.

Run with make command

```
$ make
echo "Hello, World"
Hello, World
```

```
$ make hello
echo "Hello, World"
Hello, World
```

Bu şekilde kullanarak sadece target'ı çalıştırabiliriz.

main.c

```c
#include <stdio.h>
#define X 10
int print_integer(int);
int main(){
    int a = X;
    printf("Hello World!")
    print_integer(5);
    return 0;
}
```

aux.c

```c
#include <stdio.h>
int print_integer(int a){
    printf("integer: %d\n", a);
    return 0;
}
```

Makefile

```
hello:
    echo "Hello world!"
```

↓ geliştirelim

```
main:
    gcc main.c aux.c
cleanall:
    rm main.o aux.o a.out
cleanobj:
    rm main.o aux.o
```

```
main: aux.o main.o
    gcc main.o aux.o -o program
aux.o:
    gcc -c aux.c
main.o:
    gcc -c main.c
```

Multiple Target

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

Makefile

```
some_file: other_file
    echo "This will always run, and runs second"
    touch some_file

other_file:
    echo "This will always run, and runs first"
```

```
$ make
echo "This will always run, and runs first"
This will always run, and runs first
echo "This will always run, and runs second"
This will always run, and runs second
touch some_file
```

Variables

```
files := file1 file2
some_file: $(files)
	echo "Look at this
variable: " $(files)
	touch some_file

file1:
	touch file1
file2:
	touch file2

clean:
	rm -f file1 file2
some_file
```

Automatic variables

```
hey: one two
	# Outputs "hey", since this is the target name
	echo $@

	# Outputs all prerequisites newer than the target
	echo $?

	# Outputs all prerequisites
	echo $^

	touch hey

one:
	touch one

two:
	touch two

clean:
	rm -f hey one two
```

direk variableları otomatik algılıyor

Variablelar tanımlayıp farklı farklı assigment işlemleri yapılabilmektedir.

Fancy rules

Implicit Rules

```
CC = gcc # Flag for implicit rules
CFLAGS = -g # Flag for implicit rules. Turn on debug info

# Implicit rule #1: blah is built via
# the C linker implicit rule
# Implicit rule #2: blah.o is built via
#the C compilation implicit rule, because blah.c exists

blah: blah.o

blah.c:
	echo "int main() { return 0; }" > blah.c
clean:
	rm -f blah*
```

Static pattern rules

targets...: target-pattern: prereq-patterns ...

commands

```
objects = foo.o bar.o all.o
all: $(objects)

# These files compile via implicit rules
foo.o: foo.c
bar.o: bar.c
all.o: all.c

all.c:
	echo "int main() { return 0; }" > all.c

%.c:
	touch $@

clean:
	rm -f *.c *.o all
```

hepsini require etti

For more see https://makefiletutorial.com/

Implicit Rulelar ile otomatik işlemler yapılabilmektedir.

Implicit Rule kendi kendine gizli tanımlanmış işlemler aslında yani kapalı rule gibi de düşünülebilir

Preprocessor

```
/* Before preprocessing */
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]
```

```
/* After preprocessing */
char buffer[10
```

Object Like MACRO

```
#include <stdio.h>
#define X 1, \
          2, 3
int a = X
```

↓

```
int a = 1, 2, 3
```

```
#include <stdio.h>
#define X 1,2,3
int a =X
```

↓

```
int a =1,2,3
```

```
#include <stdio.h>
#define X 1, \
          2, \
          3
int a = X
```

↓

```
int a= 1,2,3
```

Object Like MACROS: Şimdiye kadar görmüş olduğumuz bu makroların hepsi object like yani bir objeyi değiştiriyorlar. Örneğin

```
#define NUMBERS 1,2,3
int x[] = {NUMBERS};
```

Array olmuş oldu → (preprocesing) → `int x[] = {1,2,3};`

Function Like Macros: fonksiyon çağrısı gibi görünürler. C'de birçok fonksiyon var. O fonksiyon için farklı isim gibi substitionlarla farklı şeyler tanımlayabiliyorum

```
#define lang_init c_init
lang_init()
```

Aslında yaptığımız şey aynı sadece fonksiyon üzerinde çalışıyorum → (preprocessing) → c_init

Macros with arguments

comparison (karşılaştır)

x, y den küçükse True ise X

y, x'den küçükse False ise y

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))

x = min(a, b);
/* →  x = ((a) < (b) ? (a) : (b));*/

y = min(1, 2);
/* →  y = ((1) < (2) ? (1) : (2)); */

z = min(a + 28, *p);
/* →  z = ((a + 28) < (*p) ? (a + 28) : (*p)); */
```

Burada parantez kullanarak (X) şeklinde yapmamız çok önemli

Böylece a+28 diye bir argüman gönderince parantez dikkate alır

```
#define foo(x) x, "x"
foo(bar)

/*
→ bar, "x"
*/
```

Seni korur

Macro pitfalls-misnesting

```
#define twice(x) (2*(x))
#define call_with_1(x) x(1)
call_with_1 (twice)

/*after preprocessing
    → twice(1)
     → (2*(1))
*/
```

```
//the use of unbalanced open parentheses in a
macro body is just confusing, and should be
avoided.
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)

/*→ fprintf (stderr, "%s %d", p, 35)
*/
```

https://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros

X++ : Önce işlem yapılıyor sonra x'in değeri memory'de 1 artırılıyor

++x : Önce x'in değerini artırıyor. Daha sonra işlem yapıyor

Macro pitfalls-Operator Precedence Problems

```
#define ceil_div(x, y) (x + y - 1) / y


a = ceil_div (b & c, sizeof (int));


/*after preprocessing
    → a = (b & c + sizeof (int) - 1) / sizeof (int);
*/

Solution: each occurrence of a macro argument name
should have parentheses around it.
```

```
/*What is the problem here?*/
#define min(a,b) a < b ? a : b

int main() {
     int x = 4;
     if(min(x++, 5)) printf("%d is six", x);

     int r = 10 + min(99, 100);
     printf("r is %d", r);
     return 0;
}
```

https://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros

Bazen duplication fonksiyon çağrılarını göndericerseniz!

Macro pitfalls-duplication side effects

```
/*What is the problem here?*/
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))

int main() {
     int next = min (x + y, foo (z));
/*after preprocessing
    → next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
the expression foo (z) has been substituted twice into the macro expansion.
*/
     return 0;
}
```

Normalde fonksiyonu bir kere çağırıp yapabilecekken

- Fonksiyonu iki kere çağırmış oldum boş yere
- Beklenmeyen hataya sebep olabilir.
- Performans düşebilir

https://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros

Macro pitfalls-swallowing semicolon

```
#define SKIP_SPACES(p, limit)  \
{ char *lim = (limit);         \
  while (p < lim) {            \
    if (*p++ != ' ') {         \
      p--; break; }}}


SKIP_SPACES (p, lim);
/* ";" causes problems before else */

if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

```
/*solution*/
#define SKIP_SPACES(p, limit)     \
do { char *lim = (limit);         \
     while (p < lim) {            \
       if (*p++ != ' ') {         \
         p--; break; }}}          \
while (0)

SKIP_SPACES (p, lim);
/*after preprocessing
    expands into

do {...} while (0);
*/
```

https://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros

Macro conditionals

```
int main() {
     #ifdef __GNUC__
     return 1;
     #else
     return 0;
     #endif
}

$ gcc -E main.c >main.i
...
int main() {
 return 1;
}

In different compiler you would get return 0
```

Eger GNUC compiler ise şunu yap

Değilse şunu yap

Mimariye göre farklı şeyler yaptırılabilir.

C keywords

`extern`, `break`, `const`, `continue`

`do{} while{};`

`enum`

`for (initialization; check; update) {}`

`if{} else{}`

`inline`

`goto (do not use in your programs)`

`restrict` çok önemli değil sınav için

`return`

`signed`, `unsigned`

`sizeof` -an operator

`static`

`struct, union, typedef`

`switch case`

Bunu kullanma

Extern

```
extern int print_integer(a);
extern int b;
int main() {
     int a=5;
     print_integer(b);
     printf("min: %d\n", min(a+2, a*3);
     return 0;
}

int b = 5;
int print_integer() { .... }
```

Bu belki tanımlanmış olabilir.

Ama eğer biz en başa bunu extern olarak belirtmezsek program hata verebilir.

Benzer şekilde fonksiyonun definitionu farklı bir dosyada ise object dosyası oluşturuken hata verebilir. Extern hata vermesini engelliyor. bunu ileride göreceğiz. Şikayet etme!

Static

main.c

```
static int b = 100;
extern int print_integer(int);
int main(){
        printf("min: %d\n", b);
        print_integer(1000);
        return 0;
}
```

> gcc -c main.c bu komut çalışır ve bir adet main.o object dosyası elde ederim.

Ama gcc main.o aux.o -o main komutu sonrası HATA alınır

Bunun sebebi static kullanımı ile locale edilmesi

aux.c

```
int b=5;
static int print_integer(int a){
        printf("%d\n", a);
        return 0;
}
```

gcc -c aux.c ile aux.o object file ını oluşturduk.

nm aux.o yu incelersek

```
00000000 D b
         U printf
00000000 t print_integer
```

static 1. kullanım şekli

static yazınca t küçüldü. T büyük ise global t lowercase ise local anlamına geliyor. Yani hangi dosyadaysa o dosyaya local oluyor.

main.c

```c
#include <stdio.h>
static int b = 100;
extern int print_integer(int);
int main(){
    int a = 5;
    print_integer(1);
    print_integer(2);
    print_integer(3);
    return 0;
}
```

aux.c

```c
#include <stdio.h>
int b=5;
int print_integer(int a){
    static int counter =1
    counter++;
    printf("Verilen : %d - counter : %d \n", a, counter);
    return 0;
}
```

```
gcc -c main.c
gcc -c aux.c
gcc main.o aux.o -o program
```

```
./program
Verilen: 1  counter: 2
Verilen:2  counter:3
Verilen:3  counter: 4
```

counter bir sefer tanımlanıyor, Definition oldudtan sonra her fonksiyon çagrildiginda aynı definition üzerinden devam ediyor
Ve burada scope u sadece fonksiyonun içerisinde

C operators

In order of precedence

[], Super bracket

->,

.,

+/-a,

*a, direference

&a, address

++, --,

sizeof,

arithmetic operators {+,-,*,%,/},

>>/<<,

<=/>=,

==/!=, &&, ||,

!, &, |, ~, Hata yapmamak için parantez kullan

?:,

a,b

C Data types and Operators

Char, short (short int), int, long (long int), float, double
Fixed width integers
[u]int*width*_t

int { 32 bit makine 2 byte
      64 bit makine 4 byte

```c
char a; /*1-byte data*/
short a; /*at least 2 bytes*/

int a; /*at least 2/4 bytes*/
long a;  /*at least 4/8 bytes*/
float a; /*IEEE-764 single precision floating point: 4 bytes*/
double a; /*IEEE-764 single precision floating point: 8 bytes*/
```

Shift operators

```
unsigned short uns = -127; // 1111111110000001
short sig = 1; // 0000000000000001
uns << 2; //1111111000000100
sig << 2; // 0000000000000100
uns >> 2; // 1111111111100000
sig >> 2; // 0000000000000000
```

In right shift;

- If the operand on the left is signed, then the integer is sign-extended. This means that if the number has the sign bit set, then any shift right will introduce 1s on the left.
- If the number does not have the sign bit set, any shift right will introduce zeros on the left.
- If the operand is unsigned, zeros will be introduced on the left either way.

```
int a = 5
int * ptr;
ptr = & a;
printf("a: %d, %p, %p", a, ptr, &a);
```

a : 5 , adres , adres

```
printf("a: %d, %d \n", a, *ptr);
```

a: 5 , 5

```
printf("a: %d, %d \n", a, ptr[0]);
```

a: 5 , 5

$*ptr = *(ptr+0) = ptr[0]$

Pointer in da adresini tutmak mümkün

```
int a = 5
int * ptr;
```

→ türü (int *) bunun adresini tutmak için

```
ptr = &a
(int *)* q = &ptr
```

q 'dan a 'ya nasıl ulaşırım?

```
printf("a: %d", *(*q));
```

$**q = q[0][0] = a$

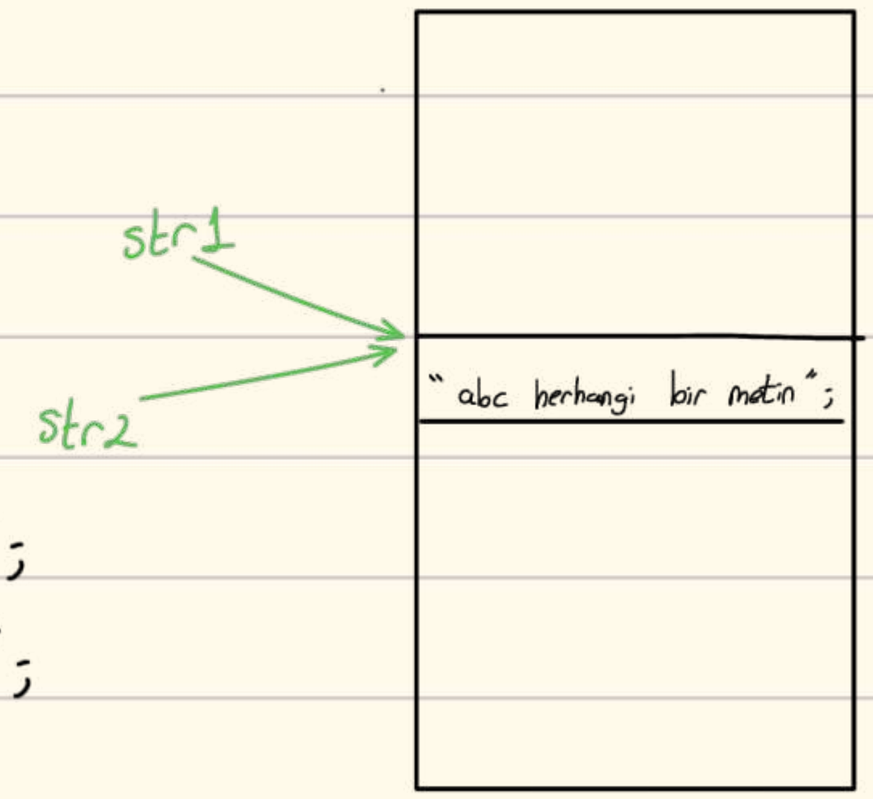
NOT: void pointerlar üzerinde artırma, azaltma işlemleri yapılamamaktadır.

```
Char *ptr = "hello";
```

String constant

3. HAFTA

```
#include <stdio.h>
int main() {
    char a[4] = {'a', 'b', 'c', '\0'};
    char b[4] = {'a', 'b', 'c', '\0'};
```

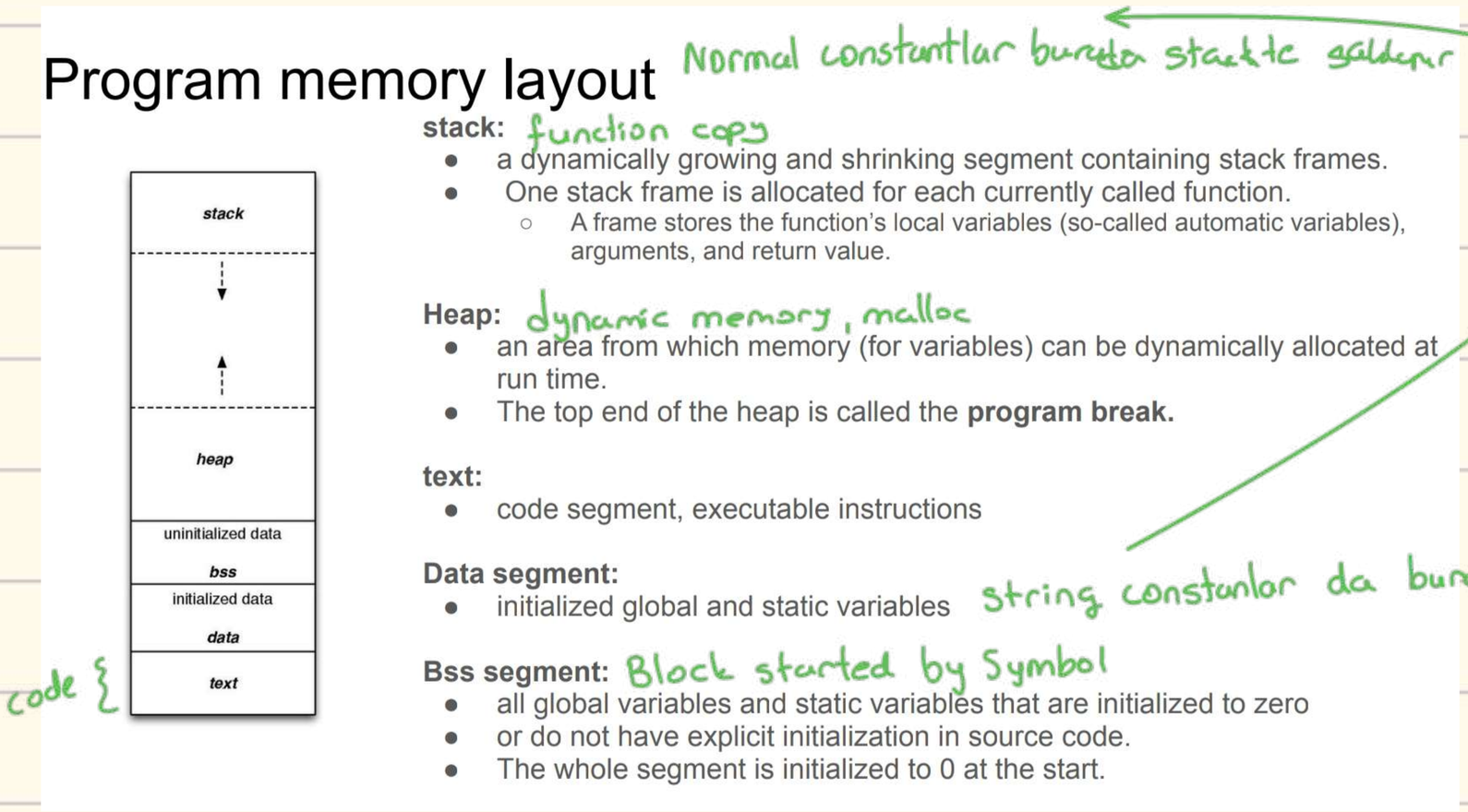


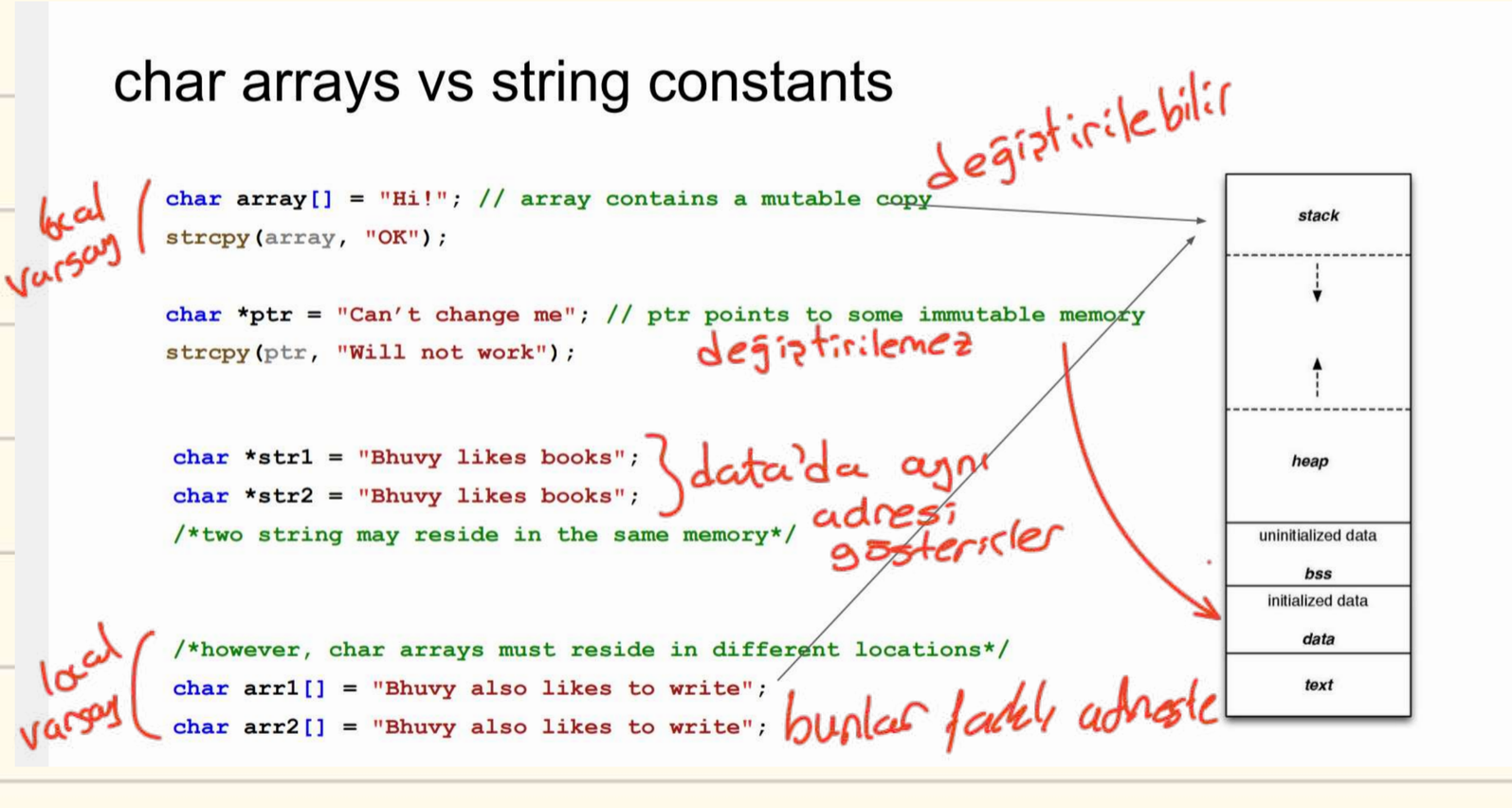

```
    char *str1 = "abc herhangi bir metin";
    char *str2 = "abc herhangi bir metin";
```

`str1[0] = 'A';` Hata verir

`a[0] = 'A';` Çalışır

Çünkü "abc herhangi bir metin" string constanttır. String constantlar memoryde bir defa tanımlanıyor





Constant Pointer vs Pointer to constant

Constant pointer vs Pointer to constant

```c
/*a constant variable*/
const int a = 5;
const int b = 10;

/*a pointer to a constant integer*/
const int* ptr;

ptr = &a;
*ptr = 10; /*cannot do*/
```

```c
/*a constant pointer to int*/
int c = 15, d = 20;
int *const ptr2 = &c;

/*cannot change the pointer value*/
ptr2 = &d;
```

```c
/*constant pointer to a constant*/
const int *const ptr3 = &a;
```

const tells the compiler that the variable cannot be written (is constant).
This will result in a compile-time error: This has no influence on the memory space where the variable is located. In Run-time it may change its value.

Pointer to Functions

Pointers to functions

```c
#include <stdio.h>
int intcmp(int *a, int *b){
    return *a > *b;
}
int main(){
    int x = 4, y = 5;
    /*declaration*/
    int (*comp)(void *, void *);

    /*assignment with type casting*/
    comp = (int (*)(void *, void *)) &intcmp;

    /*function call*/
    int z = (*comp)(&x, &y);

    printf("%d\n", z);
    return 0;
}
```

comp pointeri declare edilmiş bu int döndüren void * void * türünü bilmediğimiz iki argüman alır.

```
int (*comp) (void *, void*)
```

- comp is a pointer to a function
- The function should have two void*
- The function returns int

24-25/93

```
int *comp(void *, void*)/*WRONG*/
```

- comp is a function
  - returning an int pointer

int parametreli fonksiyon void'e cast ediliyor

Pointers to functions are often used to implement **polymorphism** in "C". (parametre falan)

tek bir pointer ile prototipleri aynı olan birden fazla fonksiyonu typecasting yaparak çağırabiliriz

```
int main (){
        printf (" %p - %p \n", &main , main);
}
```

Her ikisi de aynı adres

0x56535afd149 - 0x56535afd149

Fonksiyonu Pointer ile işaret etme?

```
int main (){
    int *p = main;
}
```

Benim tutmuş olduğum değer bir integer adresi main integer değil! Fonksiyon

Ben öyle bir syntax uyduracayım ki bu da fonksiyonların adresi olsun. pointer ile fonksiyon adresi

```
int main () {
    int (*p) () = main;
    printf ( "%p - %p \n", main , p);
    return 0;
}

int main ( int argc , char **argv ) {
    int (*p)(int , char **) = main ;
    printf ( "%p - %p \n", main , p);
    return 0;
}
```

Pointerın fonksiyonu işaret ettiğini belli etmek için parantez alıyorum

int * p

int ,

```
int main (){
    printf (" %p - %p \n", &main , main);
}
```

Her ikisi de aynı adres

```
int main (int argv)

int(*p)(void) = main
```

```
int main () {
  return 0;
}
```

```
int main (){
    int *p = main;
}
```

Benim tutmuş olduğum değer bir integer adresi main integer değil! Fonksiyon

Ben öyle bir syntax uyduracayım ki bu da fonksiyonların adresi olsun. pointer ile fonksiyon adresi

```
int main () {
    int (*p) () = main;
    printf ("%p - %p \n", main , p);
    return 0;
}
```

Pointerın fonksiyonu işaret ettiğini belli etmek için parantez alıyorum

```
int main ( int argc, char **argv ) {
    int (*p)(int, char **) = main;
    printf ("%p - %p \n", main , p);
    return 0;
}
```

Soru:

```
int intcmp (int *a, int *b){
        return a> b;
}

int x = 5;
int y = 3;
```

(tanımlama)

> Buna uygun function pointer declare et

> typecasting ile assign et

> Fonksiyonu pointer üzerinden çağır

declare

```
int (*p)(void *, void *);
```

declare

```
int(*p)(int*, int*);
```

assign typecast

```
p = (int (*)(void*, void *))intcmp;
```

assign

```
p = intcmp
```

function call

```
int z = (*p)(&x, &y);
int z = p(&x, &y);
```

function call

```
int z=(*p)(&x, &y);
int z= p(&x, &y);
```

Burada otomatik olarak int * ifadesi void * ifadesine implicit cast edilmiştir

```
z = *p((void*)&x, (void*)&y);
```

implicit cast bunu yapar.

Structs and Pointers

Structs and pointers

```c
#include <stdio.h>
typedef struct{
   int a1;
   int a2;
} pair;
```

```c
int main(){
   pair obj;
   pair zeros;
   zeros.a1 = 0;
   zeros.a2 = 0;

   pair *ptr = &obj;

   obj.a1 = 1;
   obj.a2 = 2;

   *ptr = zeros;
   printf("a1: %d, a2: %d\n", ptr->a1, ptr->a2);
   return 0;
}
```

```c
#include <string.h>
int main () {
    struct User {
        char name[10];
        int id;
    };
    struct User u1;
    strncopy (u1.name, "isim1", 10);
    u1.id = 5;

    struct User *p = &u1;
    printf ("%.10s \n id: %.10d \n", p->name, p->id);
```

Tip tanımlaması

int r; 8bit
int g; 8bit
int b; 8bit

little endian : bgr en significant sağda
bigendian : rgb en significant solda

Hafta-4 Slayt 2

C functions and System Calls

Common C functions

Herşey dosya POSIX standart mantra

POSIX mantra; "**Everything is a file**"

- Everything is a **file descriptor**
  - an integer.

```
int file_fd = open(...);

int network_fd = socket(...);

int kernel_fd = epoll_create1(...);
```

- These are all references to **the kernel's file descriptor table (a table of objects)**.
- File descriptors are simply pointers to these objects.

Kernel içerisindeki descriptor table

Operations on these objects

- open, close, allocation, deallocation
  - are done through **system calls.**

Her şey dosya olduğu için biz bu dosyaları integer olan file descriptor ile gösteririz

file descriptor her şeyi gösterir.

The program interacts with these objects using the API specified through system calls and library functions.

Her bir processin tutulduğu bir process table vardır. PT üzerinden sistem takip edilir.

Her bir process işlemler yapıyor. Mesela dosya açtı kapadı env izinleri falan

Process Table

hangi processler var

hangileri çalışıyor

Process Control Table

process için

file_descriptor

Hangi dosyalar açık? Hangi dosyalar kapalı Global table Kernel kısmında bulunur.

printf , fopen, scanf, fread ... Bu fonksiyonlar farklı:

printf: standart output'a output veriyor

fopen: harddiskten bir tane dosya açıyorum

System calls

User processes

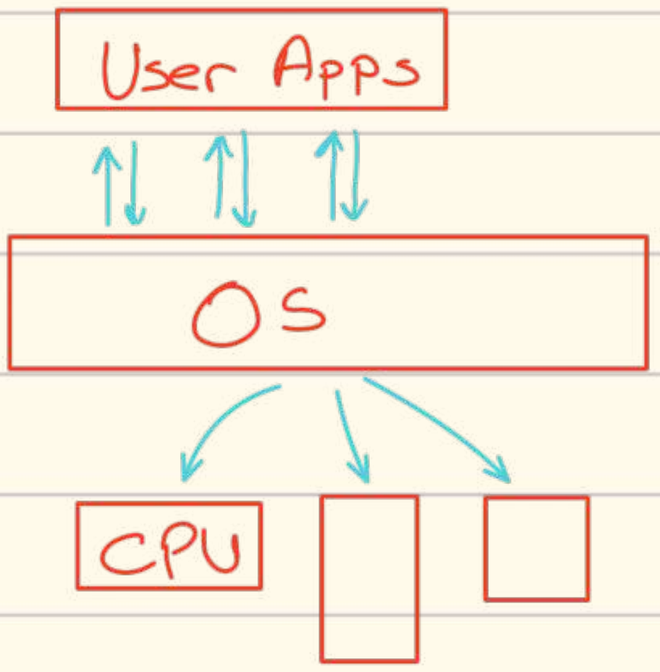
- cannot perform privileged operations
- request OS to do so on their behalf by issuing **system calls**

A **system call** is a request made by **an *active process (through traps or software interrupts)*** for a service performed by the kernel.

Examples:

- input/output (i.e., any movement of information to or from the combination of the CPU and main memory)
- creation of a new process.
- And many more https://man7.org/linux/man-pages/man2/syscalls.2.html

User programlarının privileged işlemleri yapmasına izin verilmiyor
yetki gerektiren işlemler



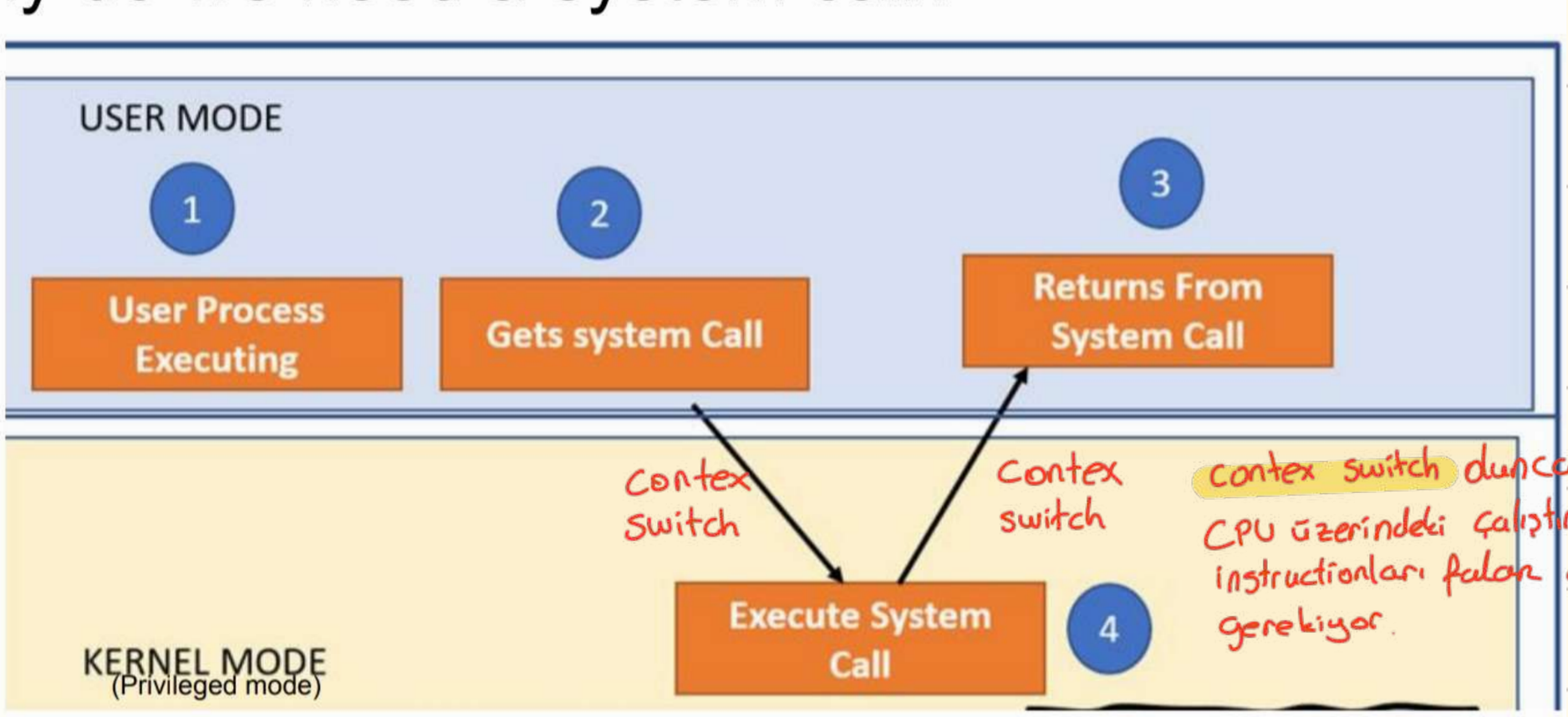
API (Application Program Interface)

Hardware'e erişimi OS yapıyor

User App API üzerinden request yapmalı

System call aktif bir process tarafından yapılan bir requesttir
Kernel'dan bir servis ister

Why do we need a system call?



context switch dunca CPU üzerindeki çalıştırdığım instructionları falan değiştirmem gerekiyor.

How do we switch(transition) to kernel mode?

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	**int $0x80**	**eax**	**eax**	**edx**	**-**	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
...						
x86-64	**syscall**	**rax**	**rax**	**rdx**	**-**	**5**
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

CPU çeşitleri

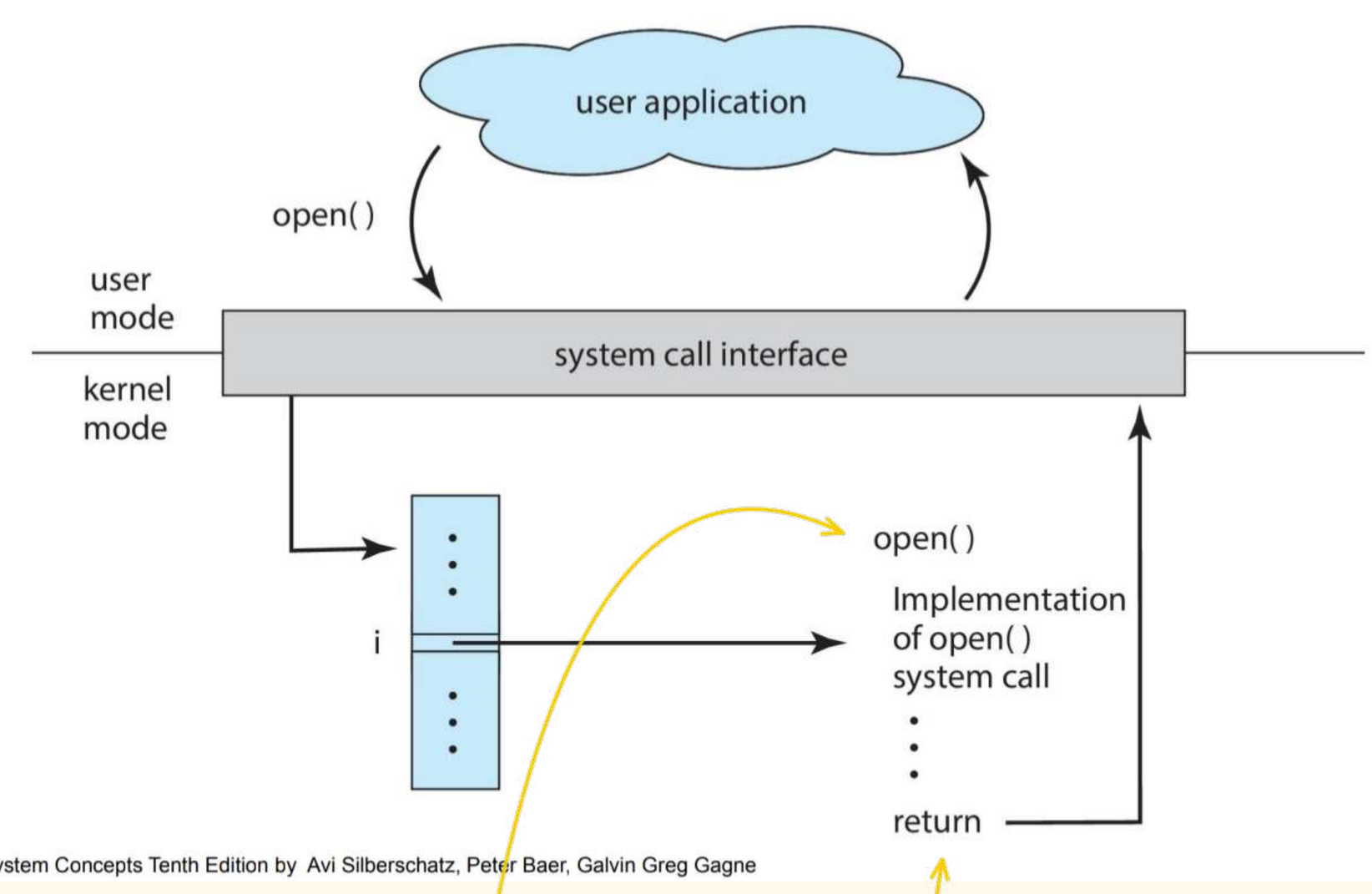
https://man7.org/linux/man-pages/man2/syscall.2.html

Bazı sistemlerde direk trap olarak yazılabiliyor bazıların instructionları belli

int $0x80

Interrupt genelde donanımsal olarak anlaşılır. Örn kamera görüntü aldı. CPU'yu interrupt etti. CPU işini bırakıp kameraya cevap veriyor

API – System Call – OS Relationship



Operating System Concepts Tenth Edition by Avi Silberschatz, Peter Baer, Galvin Greg Gagne

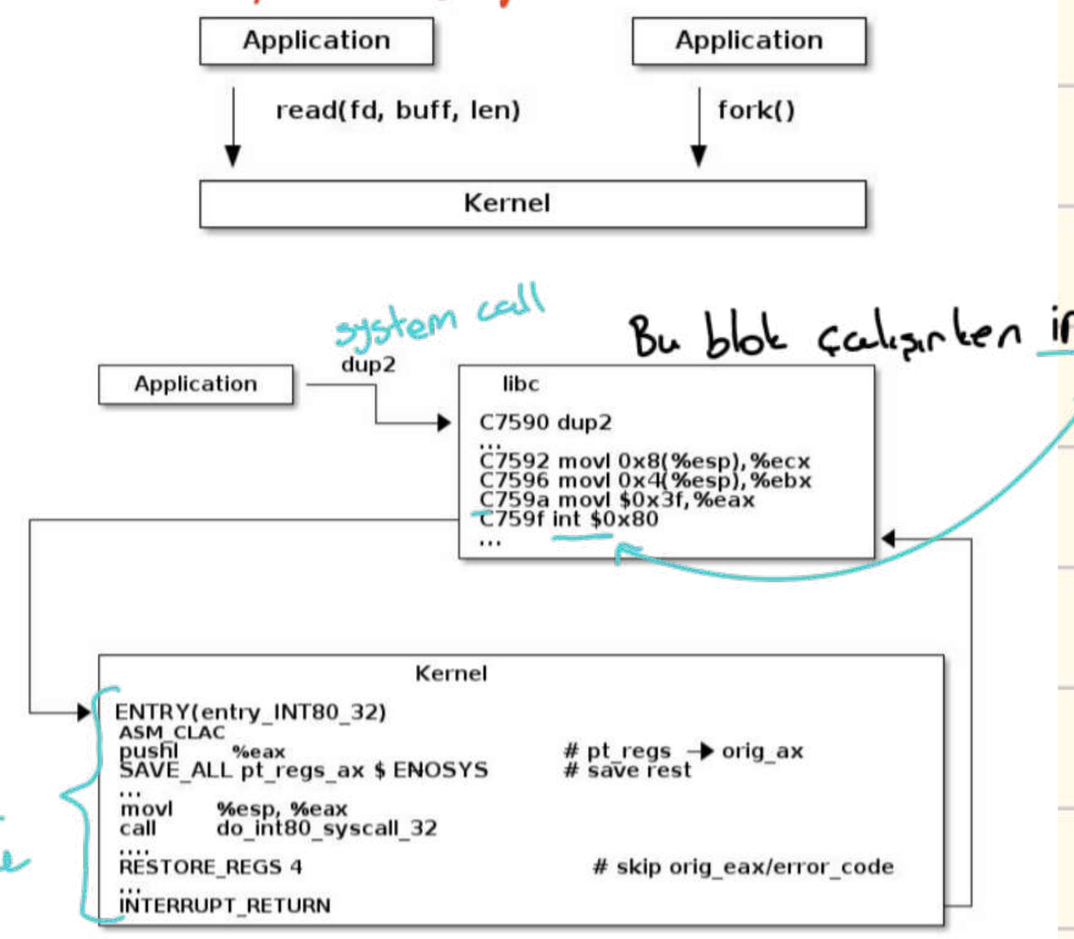
Mesela user application open() system call dedi. Open ile ilgili ben neyi çalıştırmalıyım kernel bunu buluyor ve bunu return ediyor, sistem kaldığı yerden devam ediyor

Standart output, input, file descriptor falan varsa onların üzerinde kopyalama falan yapmak için kullanılan bir şey

Example dup2() on x86

System calls are specific assembly instructions that

- setup information to identify the system call and its parameters
- trigger a kernel mode switch
- retrieve the result of the system call



https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

syscall genelde trap yapar ama bazı mimarilerde software interrupt

trap vs exception vs interrupt

Exception bir işlemi yaparken karşılaşılan beklenmeyen durum

An exception is an unexpected event occurs because of **the execution of an instruction** (synchronous event).

- Division by zero
- **CPU detects this:**
- **Exception Types** 3 Exception var
  - Traps,
    - Breakpoint, Trap, Syscalls, division by zero
  - faults,
    - Memory faults(Page fault),
  - aborts

Hardware interrupt is an unexpected event from outside the processor.

Software interrupt (e.g., int 0x80) is generated by the application programs(similar to traps).

$\frac{a}{b} = c$ b=0 olursa (zero division)
Exception → Trap

https://stackoverflow.com/questions/3149175/what-is-the-difference-between-trap-and-interrupt

Exceptionlar software tarafından geliyor. Bir özelliği de bunlar hep Instruction çalışması sonucu oluyor. Ne zaman olduğunu neden olduğun bildiğim için (synchronous event)

Interruplar'ın çeşidi hardware ve software olmak üzere iki tanedir Hardware interrupt'ın ne zaman geldiğini bilmiyorum. Dolayısıyla bu bir (asynchronous event)

C stderr, error messages

```c
#include <stdio.h>
#include <stdlib.h>
main(){
   int dividend = 20, divisor = 0, quotient;
   if (divisor == 0){
       fprintf(stderr, "Division by zero! Exiting...\n");
       exit(-1); /*=exit(EXIT_FAILURE)*/
   }

   quotient = dividend / divisor;
   fprintf(stderr, "Value of quotient : %d\n", quotient);
   exit(0); /*=exit(EXIT_SUCCES)*/
}
```

Handling Errors: errno

- C does not provide direct support for error handling (also known as exception handling).
- -1 and NULL are used to indicate errors
- In Linux, global variable errno is assigned to the code
- errno can be used to identify the problem

strerror() and **perror()** can be used to display the error message associated with **errno**

errno value	Error
1	/* Operation not permitted */
2	/* No such file or directory */
3	/* No such process */
4	/* Interrupted system call */
5	/* I/O error */
6	/* No such device or address */
7	/* Argument list too long */
8	/* Exec format error */
9	/* Bad file number */
10	/* No child processes */
11	/* Try again */
12	/* Out of memory */
13	/* Permission denied */

```c
#include <errno.h>
#include <stdio.h>
#include <string.h>
int main(){
   FILE *f = fopen("/does/not/exist.txt", "r");
   if (NULL == f){
       fprintf(stderr, "Errno is %d\n", errno);

       fprintf(stderr, "Description is %s\n", strerror(errno));

       perror("Description is ");
   }
}
```

Common Bugs in C Programs

```c
void mystrcpy(char *dest, char *src){
   // void means no return value
   while (*src)   {
       dest = src;  src++;  dest++;
   }
}
```

while src != '\0'
*dest = *src
dest ++;
src ++;
dest* = src*

```c
while( *src ) {*dest = *src; src ++; dest++; }
*dest = *src;
```

```c
while( (*dest++ = *src++ )) {};/*bad coding practice!*/
/***************************************/
```

```c
int *p = malloc(sizeof(int));
free(p);
*p = 123; /* Oops! - Dangling pointer!
Writing to memory we don't own anymore */

free(p);  /* Oops! - Double free! */
```

```c
p = NULL; // No dangling pointers
```

```c
/*buffer overflow/underflow*/
#define N (10)
int i = N, array[N];
for( ; i >= 0; i--) array[i] = i;


gets(array); /*WRONG!: Let's hope the input is shorter than my array!*/

/*Do NOT use gets, very unsafe!!!*/
/*alternative fgets*/
```


```c
/*Strings require strlen(s)+1 bytes*/
/* returns a copy of 'input' */
char *strdup(const char *input){
   char *copy;
   copy = malloc(sizeof(char *)); /* nope! this allocates space for
    a pointer, not a string */
```

Bu direk pointer için olan ayırıyor

```c
   copy = malloc(strlen(input)); /* Almost...but what about the
   null terminator? */
```

"/0", eklemedik

```c
   copy = malloc(strlen(input) + 1); /* That's right. */
   strcpy(copy, input);              /* strcpy will provide the null terminator */
   return copy;
}
```

Şuan doğru oldu

```c
/*Using uninitialized variables*/
int myfunction(){
   int x;
   int y = x + 2;

/*Assuming Uninitialized memory will be zeroed:
Automatic (temporary variables) and heap allocations may contain
random bytes or garbage.*/
void myfunct(){
   char array[10]; Stackte automatic variable
   char *p = malloc(10); Heap dynamic, free etmedigin sürece elinde kalır
```

```c
/*Equality vs equal*/
int p1, p2;
p1 = p2 = 0;

int answer = 3; // Will print out the answer.
if (answer = 42){
   printf("The answer is %d", answer);
}

if (42 = answer){ /*compiler  will catch the
error*/
   printf("The answer is %d", answer);
}
```

```c
/*Extra Semicolons*/
for (int i = 0; i < 5; i++);
   printf("Printed once");

while (x < 10);
x++; // X is never incremented
```

Using a debugger

Using a debugger

GDB is short for the GNU Debugger.

A program that helps you track down errors by interactively debugging them.

It can start and stop your program, look around, and put in ad hoc constraints and checks.

-g option add debugging information to your program

```
gcc -g -o hello hello.c
```

Programa breakpointler koyabiliyor-sunuz. gbb için -g debugging information gdb kullanmak için bu opsiyon ile çalıştırmalıyız

Memory Leak ⟶ Valgrind

Intro to gdb

`gdb programname`

or

`gdb programname core`

to examine the coredump as if the program was running and had just faulted.

In GDB

run or **r: runs the program**

bt: backtrace when a segfault occurs

p: print the value of a variable

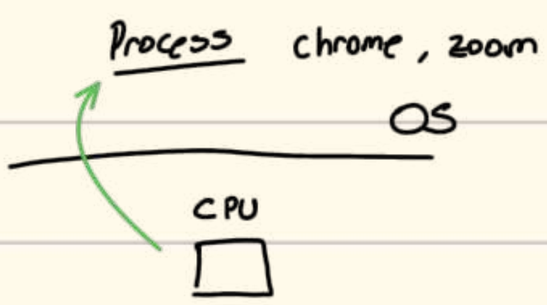
Google gdb cheat sheet to get a list of commands in gdb

Setting breakpoints programmatically

```c
int main() {
      int val = 1;
      val = 42;
      asm("int $3"); /* generates interrupt*/
      val = 7;
}
```

```
$ gcc main.c -g -o main
$ gdb --args ./main
(gdb) r
[...]
Program received signal SIGTRAP,
Trace/breakpoint trap.
main () at main.c:6
6
val = 7;
(gdb) p val
$1 = 42
```

FILE I/O: Universal I/O Model



File descriptors

bir system call

```
int file_fd = open(...);
```

file descriptor aslında bir integer (0- max)

- opening a file returns a **file descriptor**,
  - An **integer** from **0** to **max**
  - Each process has
    - **0 STDIN_FILENO** for stdin
    - **1 STDOUT_FILENO** for stdout
    - **2 STDERR_FILENO** for stderr

Her bir process için

subsequent operations (reading, writing, and so on) are done by using the file descriptor

```
write(file_fd, "Hello!", 6);
char c = 0;
read(file_fd, &c, 1);
```

file_descriptor numarası Hangi dosya ise oraya 6 karakterli "Hello!"

1 karakter okuyacağım

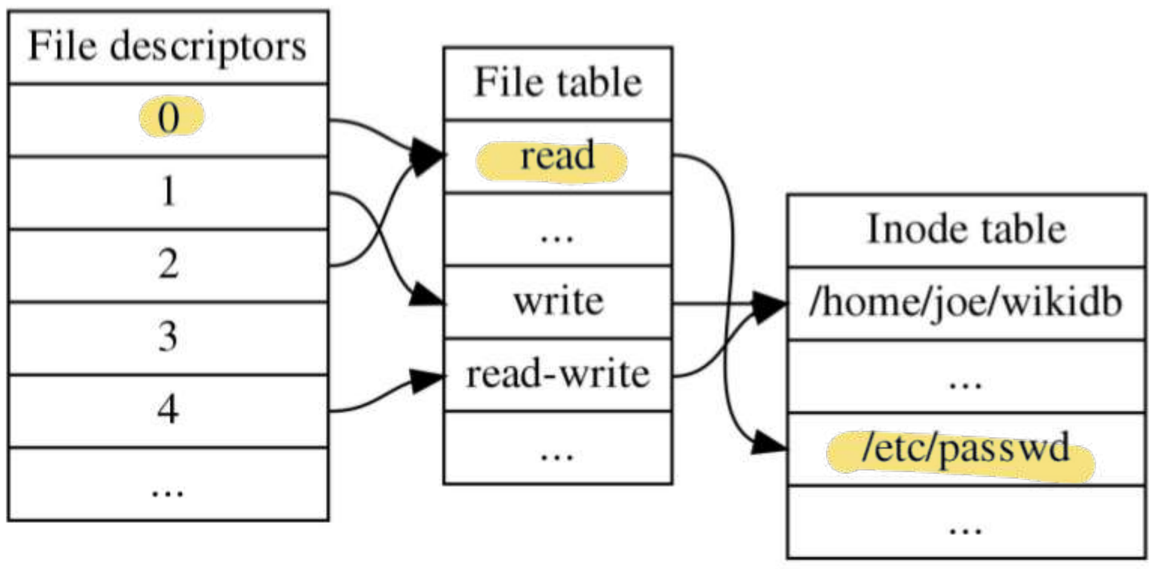
write(0, ... , ...) ise 0 stdin e yazmaya çalışıyorsun

write(1, ... , ...) ise 1 stdout a yazmaya çalışıyorsun

write(2, ... , ...) ise 2 stderr e yazmaya çalışıyorsun

HER ŞEY DOSYA!

Fds for a single process

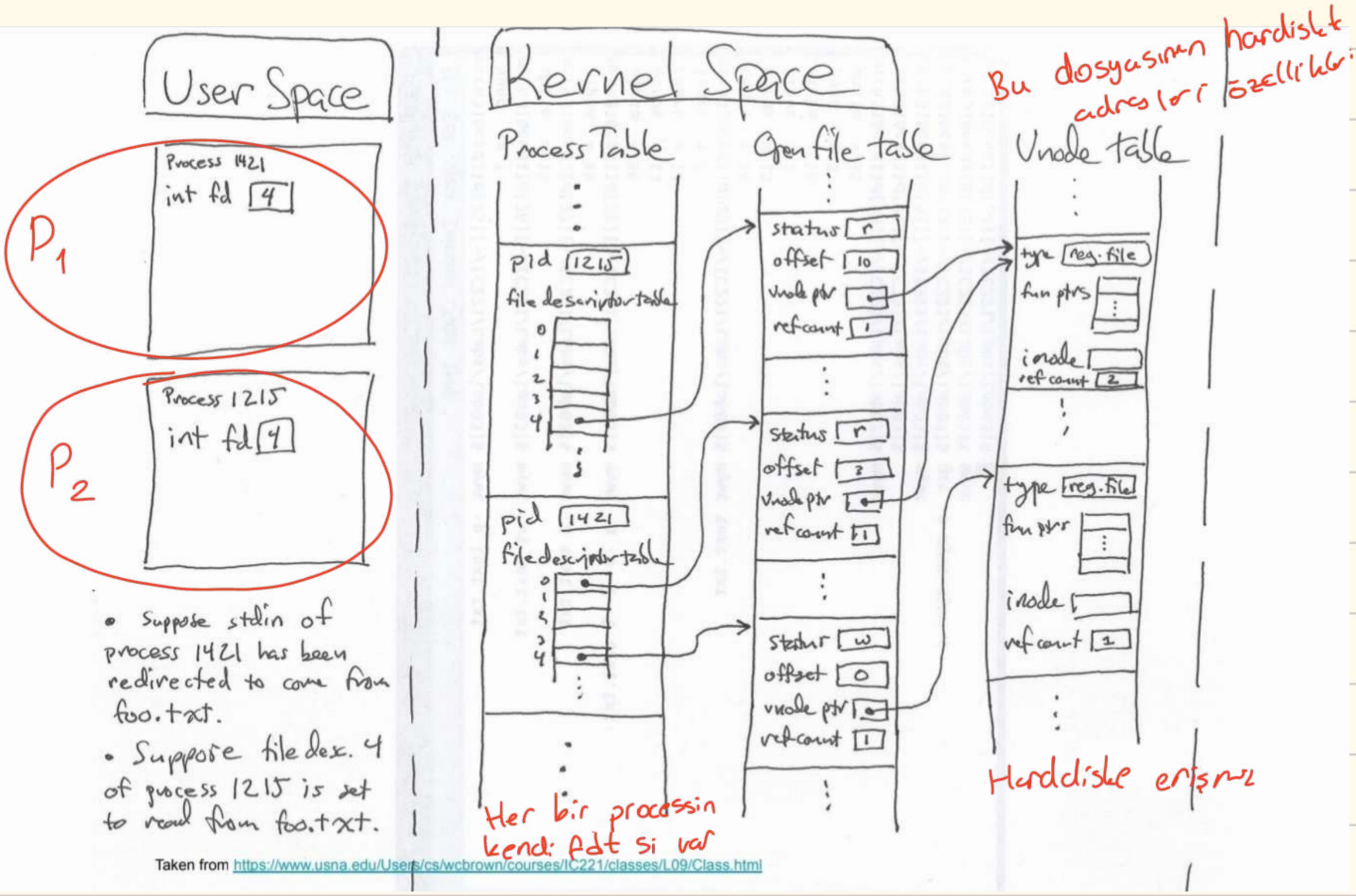


https://en.wikipedia.org/wiki/File_descriptor

- **file table:** table of the list of open files per-process
  - maintained by the kernel
  - **file descriptors(fds):** nonnegative integers that are used to index the entries of the file table.
  - Each entry in the list contains information about an open file,
    - including a pointer to an in-memory copy of the file's backing inode and associated metadata, such as the file position and access modes.

- Note that multiple file descriptors can refer to the same file table entry (dup system call),

dosya açtım fd tutuyorum. Yukarıdaki tek bir process için



Her bir process için fd, oft, vnode table → harddis

More on File descriptors

- File descriptors point to more than files
  - Remember “everything is file” in POSIX
  - Anything you can read or write (accessing devices)
- the operating system keeps track of them
- file descriptors may be reused between processes,
  - but inside of a process, they are unique.

offset değiştirilerek cursor oynatabiliriz

- File descriptors may have a notion of position. These are known as **seekable streams**.
  - A program can read a file on disk completely because the OS keeps track of the position in the file, an attribute that belongs to your process as well.
- Other file descriptors point to network sockets and various other pieces of information, that are **non-seekable streams.**

Network soketleri gibi şeylerde bir live akış olduğu için non-seekable streams

read() system call

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- attempts to read up to **count** bytes from file descriptor **fd** into the memory-buffer pointed by **buf**.

➔ Returns a value indicating number of bytes read from the file

- `0` indicates `EOF`
- `-1` indicates `an error`
- `> 0`, indicates the number of bytes read

read için bir file_descriptor

buffer adresi

fd üzerinden count byte kadar veriyi memory'deki buffer üzerine oku

- dosya sonuna ulaşıldı EOF

pozitif sayı Okunan byte sayısı

-1 Bir hata oluştu

read() system call

```
#include <fcntl.h>     /* open */
#include <sys/types.h>/* needed on some UNIX */
#include <sys/stat.h> /* needed on some UNIX */
#include <stdio.h>     /* perror */
#include <unistd.h>    /* read,close,write */
int main(){
   int fd = open("file.txt", O_RDONLY);
   if(fd == -1){
       perror("cannot open file: ");
       return -1;
   }
```

```
ssize_t ret; char buf[BUFSIZ];
ret = read (fd, buf, BUFSIZ);
if (ret == -1) {
    perror ("cannot read");
    close(fd);
    return -1;
}

buf[ret] = '\0';

printf("%s", buf);
close(fd);
return 0;
}
```

write() system call

```
#include <unistd.h>     /* read, write, close */
ssize_t write (int fd, const void *buf, size_t len);
```

- writes up to **len bytes** from the memory pointed at by **buf**

➔ Returns a value indicating number of bytes written to file

- **>-1** indicates number of bytes written
- **-1** indicates an error

int fd = open("file.txt", O_RDWR)

Burada dosya RDWR ile açıldığı için write kullanıldığında baştan dosyanın üzerine yazılacaktır.

file.txt → file.txt

SelamNaber → HelloNaber

write(fd, "Hello", len)

write() system call

```
#include <fcntl.h>     /* open */
#include <sys/types.h> /* needed on some UNIX */
#include <sys/stat.h>  /* needed on some UNIX */
#include <stdio.h>     /* perror */
#include <unistd.h>    /* read, write, close */
#include <string.h>
int main(){
   int fd = open("file.txt",
                 O_WRONLY | O_APPEND | O_CREAT,
                 S_IRWXU);
   if (fd == -1){
       perror("cannot open file: ");
       return -1;
   }
```

```
ssize_t ret;
char buf[100] = "hello";
int len = strlen(buf);
ret = write(fd, buf, len);
if (ret == -1){
    perror("cannot write!");
    close(fd);
    return -1;
}
else if (ret != len){
    printf(" written data "
           "less than len! error?");
}
close(fd);
return 0;
}
```

fopen vs open

fopen, fd'ye bir pointer

```
FILE *fdopen(int fd, const char *mode);
FILE *fopen(const char *path, const char *mode);
```

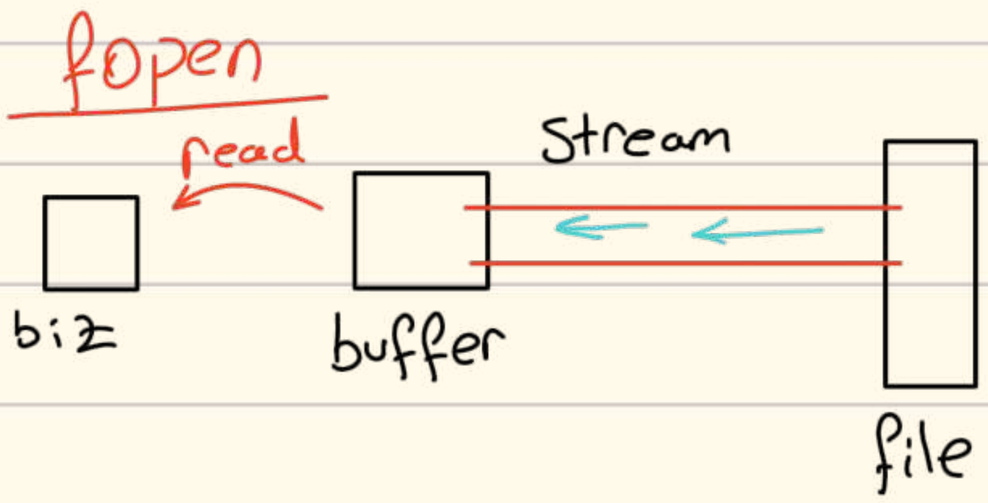
fopen() mode	open() flags
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

```
FILE *f = fopen("afile.txt", "r+");
```

- ➔ Highlivel glibc function: stdio.h,
- ➔ Provides stream with buffer

```
int fd = open("afile.txt", O_RDWR);
```

- ➔ Low-level system call..
- ➔ You need to handle most stuff.



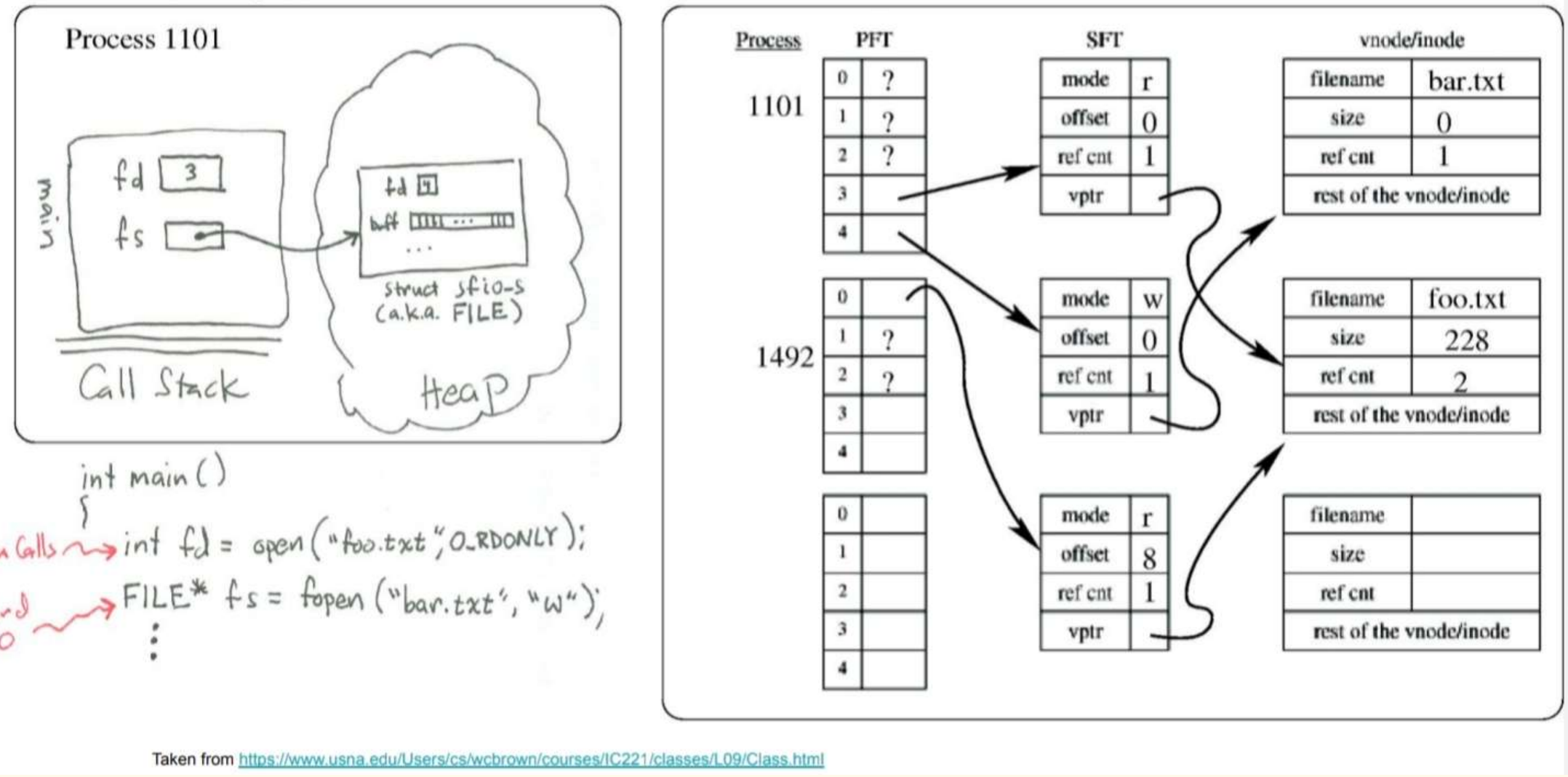
Line-oriented buffer. bufferda biriktiriyor `\n` geldiğinde dosyaya gidiyor, fflush ile bufferi boşaltabiliyoruz

open

~~stream degil~~ byte,byte

open biraz daha low level bazı şeyleri kendimiz handle etmeye çalışıyoruz.

Buffer gibi işlemleri kendiniz halletmeniz gerekiyor.



Taken from https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

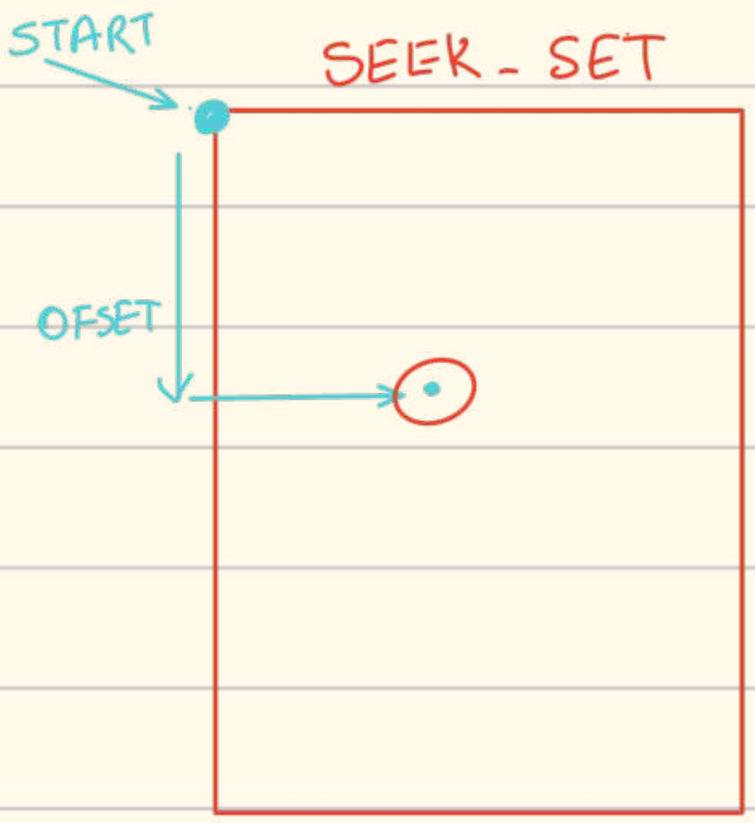
repositioning the file offset

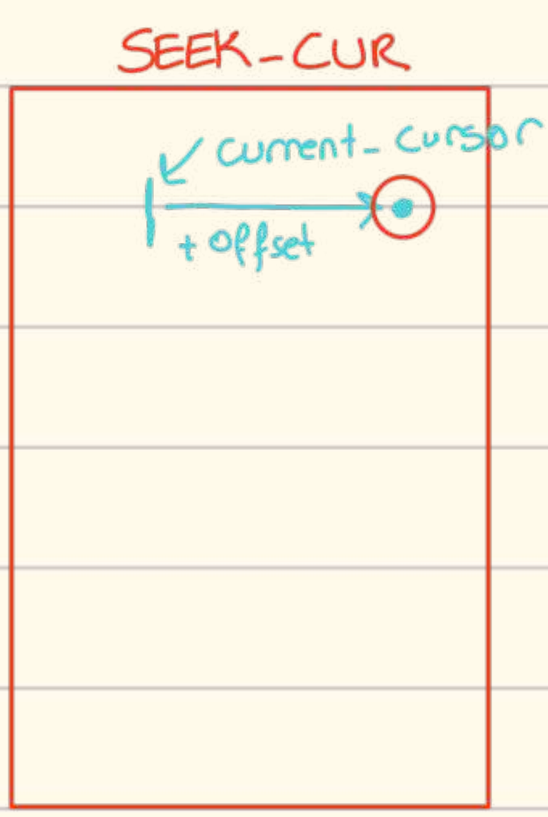
```c
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

SEEK_CUR yazınca mevcut durum + offset
SEEK_END yazınca sizeof file + offset
SEEK_SET yazınca başlangıçtan + offset

whence:

- SEEK_SET
  - The file offset is set to offset bytes.
- SEEK_CUR
  - The file offset is set to **its current location + offset** bytes.
- SEEK_END
  - The file offset is set to **the size of the file + offset** bytes.





read() system call

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- attempts to read up to **count** bytes from file descriptor **fd** into the memory-buffer pointed by **buf**.

➔ Returns a value indicating number of bytes read from the file
- `0` indicates `EOF`
- `-1` indicates `an error`
- `> 0`, indicates the number of bytes read

read() system call

```c
#include <fcntl.h>     /* open */
#include <sys/types.h>/* needed on some UNIX */
#include <sys/stat.h> /* needed on some UNIX */
#include <stdio.h>     /* perror */
#include <unistd.h>    /* read,close,write */
int main(){
   int fd = open("file.txt", O_RDONLY);
   if(fd == -1){
       perror("cannot open file: ");
       return -1;
   }
   ssize_t ret; char buf[BUFSIZ];
   ret = read (fd, buf, BUFSIZ);
   if (ret == -1) {
       perror ("cannot read");
       close(fd);
       return -1;
   }

   buf[ret] = '\0';

   printf("%s", buf);
   close(fd);
   return 0;
}
```

fd'yi kapatmayı unutma

fopen → stream

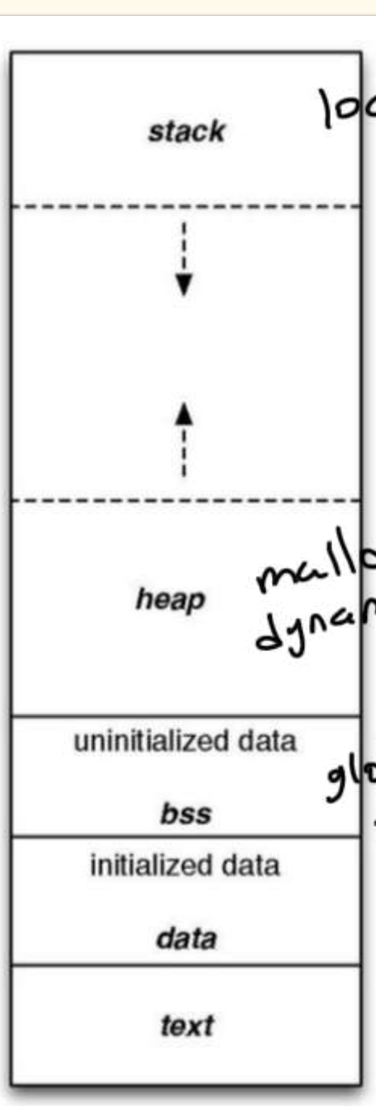
open → byte, byte

```
#define INPUT_SIZE 256
char inbuf[INBUF_SIZE] = {'\0'};
int byte;
while(1){
    write(1,"$",2);
    if((nbyte = read(0,inbuf,255)) <= 0){
        perror("input <=0 byte");
}else{
        inbuf[nbyte+1] = '\0'
}
 char *token=strtok(inbuf," \t\n");
```

Process Content

Processes are isolated! Each process gets:

- A stack
  - The stack is the place where automatically allocated variables and function call return addresses are stored.
- A heap
  - Manually controlled memory allocation for objects (malloc)
- A data segment
  - Initialized data segment
    - int global = 1
  - Uninitialized data segment / BSS
    - int global; //assumed to be zero
- Text segment
  - Executable instructions
  - Readable but not writable



local variable, normal constant

malloc
dynamic

global
static

string constant

Other contents:

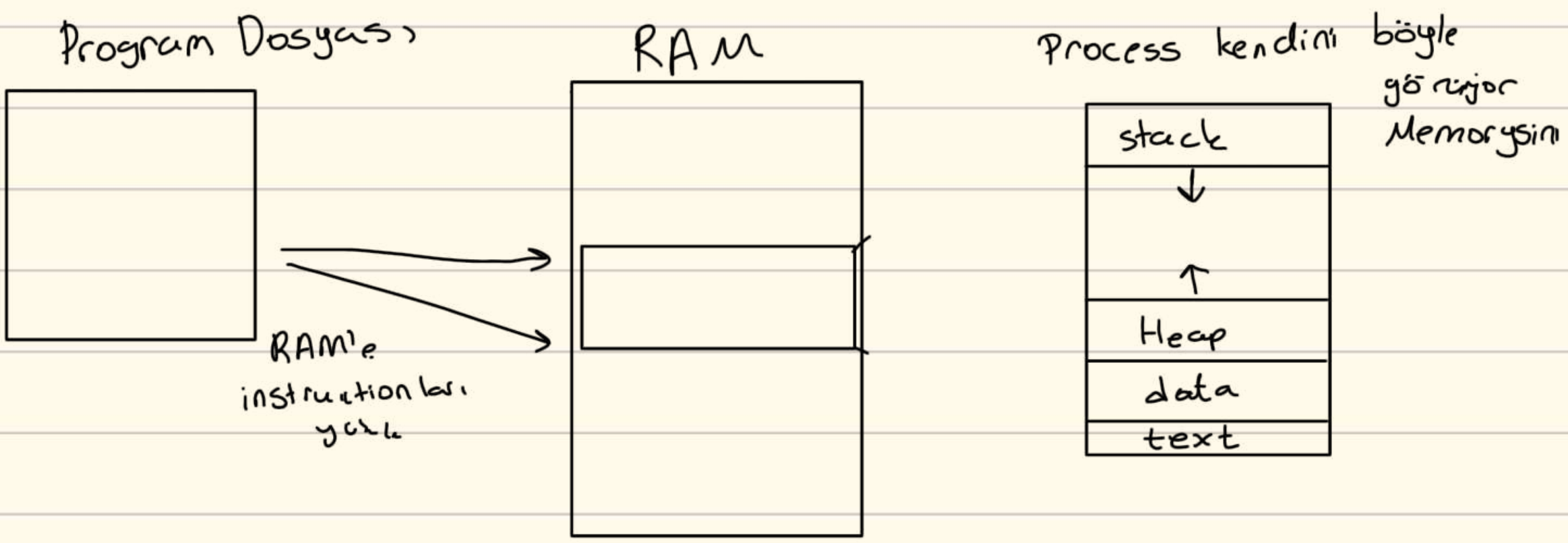
- Running state
- File descriptors
- Permissions
- Arguments
- Environment variables

```c
/*taken from the book linux programming interface*/
#include <stdio.h> #include <stdlib.h>
char globbuf[65536];              /* Uninitialized data segment */
int primes[] = {2, 3, 5, 7};      /* Initialized data segment */

static int square(int x) {        /* Allocated in frame for square() */
   int result;                    /* Allocated in frame for square() */
   result = x * x;
   return result;                 /* Return value passed via register */
}

static void docalc(int val) {     /* Allocated in frame for docalc() */
   printf("The square of %d is %d\n", val, square(val));
   if (val < 1000) {

       int t;                     /* Allocated in frame for doCalc() */
       t = val * val * val;
       printf("The cube of %d is %d\n", val, t);
   }
}
```



Process ID (pid)

Each process is represented by a unique identifier

To list processes in Linux

```
$ps aux
```

```
$ top
```

- The first process kernel executes after booting is init process with pid 1
- When there are no other processes running, kernel runs idle process with pid 0
- For other processes, pid is allocated in a linear fashion

Bütün processlerin kendi id s var

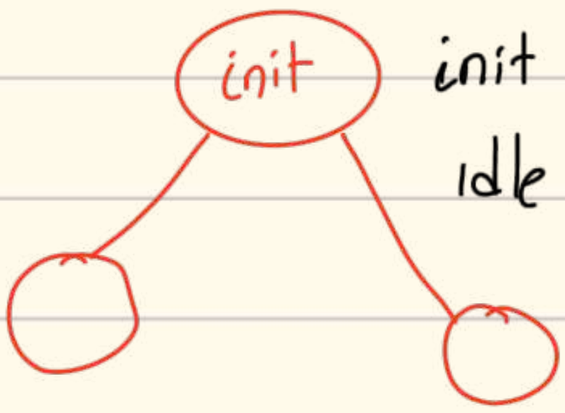
$ ps ile process'leri görebiliriz

PID	TTY	TIME	CMD
12558	ttys000	...	...
12988	ttys000	...	...

$ kill -9 PID ile processler öldürülebilir.

> Bilgisayarı ilk açtığımızda boot sonrası init process oluşur pid = 1

> Eğer çalışcan başka bir process yoksa pid si 0 olan idle process çalışır



init process ID 1

idle process ID 0

execl execv

getpid(): kendi ID sini döndürür

gitppid(): parent process ID sini döndürür

```
execl ("/usr/bin/vim", "vim", NULL);
```

path — program

→ yeni bir process oluşturur. Şuan çalışan processten bagimsız

```
char *args[] = {"./program1", NULL}
execv(args[0], args);
/* programın geri kalanı execv başarılı ise çalışmaz */
```

→ Sistem call başarılı olursa bu process bitip yeni bir process gibi execv'yi çalıştırır. Ve yeni process eski process'in fd lerini devralıyor

close ile stdout kapat

close(1); stdout'u kapat

int fd = open("yeni.txt", O_CREAT | O_WRONLY | O_TRUNC, 0666);

en düşük availability kuralı sebebiyle fd 1 değerini alacaktır. Çünkü biz 1 stdout'u kapattık.

printf("fd: %d \n", fd); Buradaki çıktı fd=1 yeni.txt üzerine yazılacaktır

Open en düşük ulaşılabilir file descriptoru veriyor.

fork()

fork ile yeni bir process oluşturabiliyoruz. Her bir processin memorysi; stack heap, datası var. Buna ek olarak process'in state'si file descriptor'u vardı. Yeni oluşan bu child process, parent'ın virtual address space'inde neyi varsa onun bir replikası olur. Copy on write ile replika oluşturulur

```
int main() {
    pid_t pid;
    pid = fork();
    if(pid >0)
        printf("I am the parent of pid %d\n", pid);
    else if (pid ==0)
        printf("I am the child! \n");
    else if (pid == -1)
        perror("fork");
}
```

child process oluştu. Geri kalan bölümleri hem child hem parent oluşturuyor.

fork işlemi ile beraber child, parentın stack heap memorysi her şeyini kopyalar sadece pid id si farklı

Burası Eksik Kalde

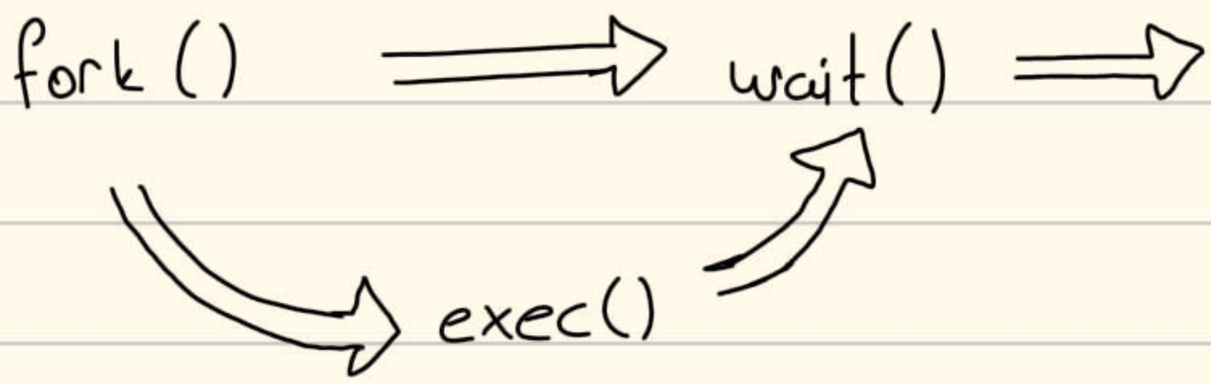
waitpid

Eğer spesifik bir process'i bekliyorsak waitpid kullanılır

```
pid_t waitpid(pid_t pid, int *status, int options);
```

↳ -1 verilirse herhangi bir child'ı, bekler wait() gibi

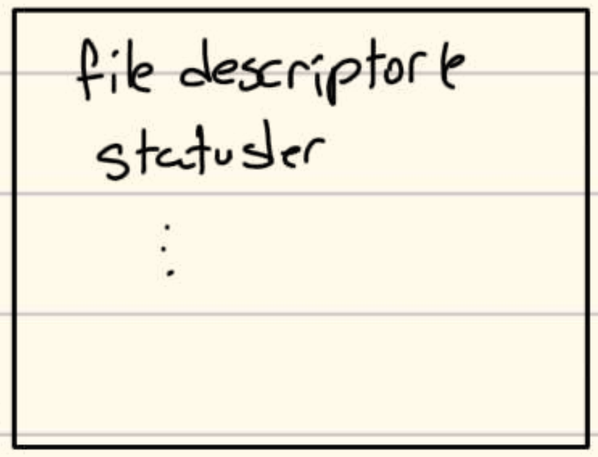
fork - exec - wait



Zombie process

Eğer child processleri wait ile beklemezseniz PCB (Process Control Block)

PCB da statusler saklanıyor ve sistem birisi wait yapar diye pid status saklar. Child process terminate oldu ama hiç wait yapmadı ise zombie process olur



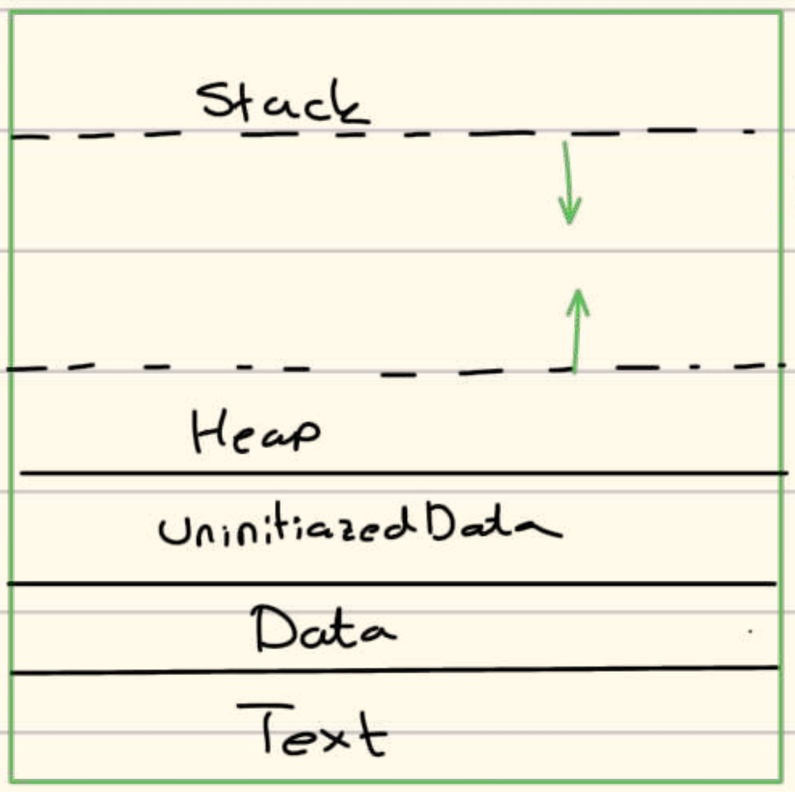
Orphan process

Orphan process'de ise child'ın parenti gitmiş sadece kendisi kalmış ama kendisi kaldığı için bunu init'e bağlıyoruz

Burası Eksik kaldı

Memory Allocation

malloc dynamic memory allocation. Program execution yaparken memorye ihtiyacı olursa bunu kullanıyor.



F() {
int a[100];
} stackte yer ayrılıyor

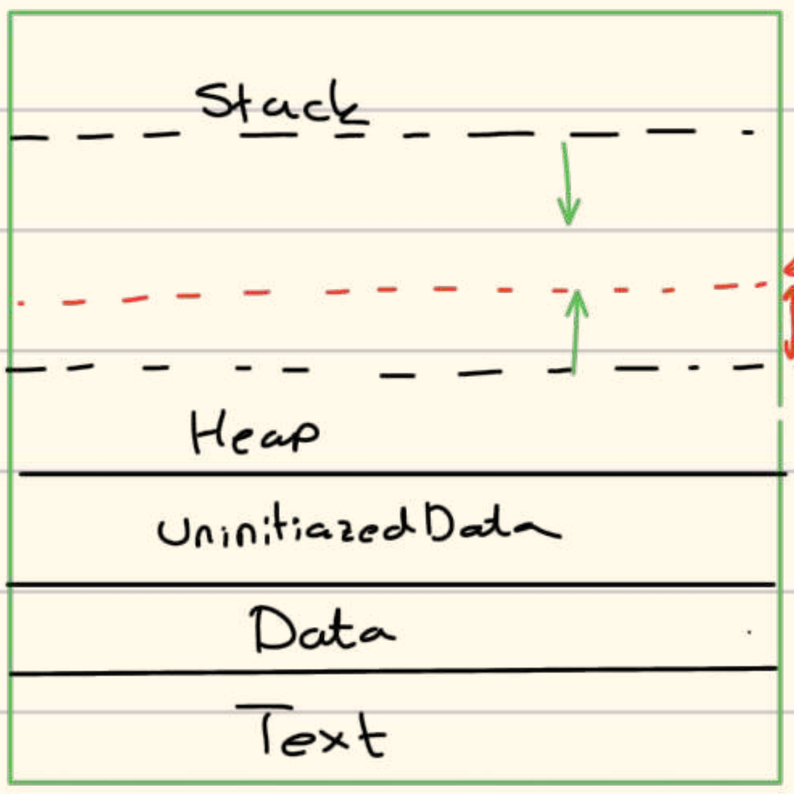
F() {
~~int a[n];~~
}

Heap ve stack birbirine doğru büyüyor. Bir fonksiyonun parametreleri fonksiyon stackde kopyalanıyor. fonksiyonun return adresi falan stack'de

malloc system call Değil! Kendine özgü implementasyonu var. Heap'i dinamik bir şekilde idare eder. Malloc $\xrightarrow{\text{alanister}}$ OS $\xrightarrow{\text{system call}}$ sbreak()

int brk(void *end_data_addr) heap'in sonunu end_data_addr'e çekiyor.

sbreak(1024): 1024 kaydır. Başlangıç pointerının adresini dön. Yani bir nevi heapi genişletmiş oluyoruz. sbreak(0) ile başlangıç adresini alabiliriz



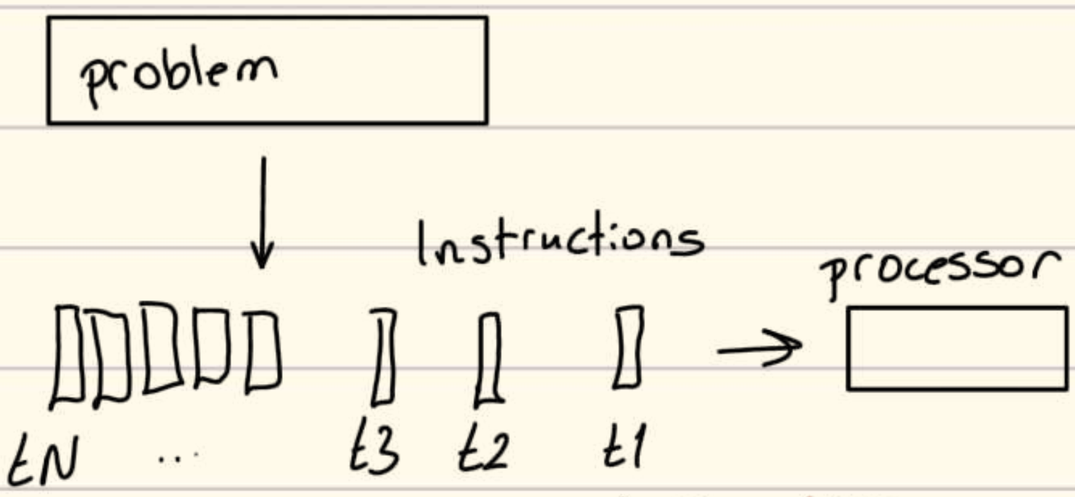
sbrk(0) ile heap'in üst tarafının adresini alırız. Yani heap'in sınırını öğreniriz.

genişletmek için içine değer yazmamız gerekli

C++ da malloc ve new-delete kullanılıyor

garbage-collector var

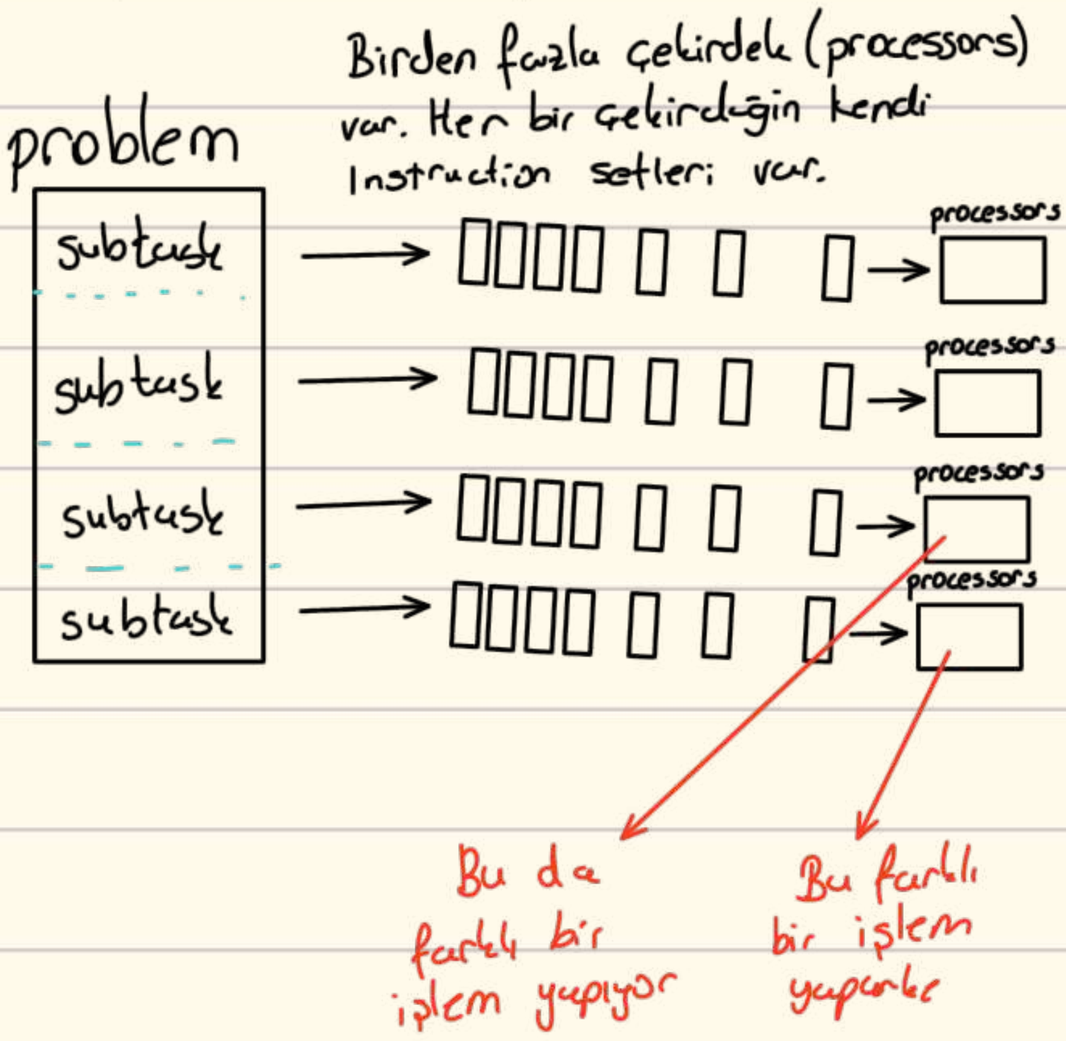
Serial Computing



Sıraya değişmiyor

Instructionların sırasını değiştirmeden olduğu gibi çalıştırma.

Parallel Computing



Parallelism: Problemi subtasklere bölmek

Parallel Computing Concurrent subtaskleri çoklu işlemcide çalıştırmak

Concurrency Bağımsız

Operasyonlar eğer aynı anda birbirlerinden bağımsız ise biz buna concurrent deriz. Concurrent tasklerde birisi önce birisi sonra farklı farklı sıralarda çalışabilirler

Mutexes "Mikrafon"

- thread'in dataya erişimini senkronizasyon yaparak kontrol eder.
- Bir çok thread'in shared memory'e aynı anda erişmesini engeller

Condition Variable

Karşı tarafı uyandırma mekanizması var. Hangi condition'u bekliyorsa o condition olunca haber verilmesi durumudur.

```
// 1st thread                          lock her zaman olmalı crit section için
mutex_lock(m1);
/* critical section entry
while(need_to_wait_1) {
    cond_wait(c1, m);
}
/* critical section end
cond_signal(c2); // or broadcast
mutex_unlock(m1);
```

```
// 2nd thread
mutex_lock(m1);
/* critical section entry
while(need_to_wait_1) {
    cond_wait(c2, m);
}
/* critical section end
cond_signal(c1); // or broadcast
mutex_unlock(m1);
```

Ben critical section'a girsem bile bir işlem yapamıyor olabilirim

thred 1 video alıyor — thread 2 video işliyor — thread 2, thread1 videoyu almadan işleyemez. lock sadece shared memory'e aynı anda erişmeni engeller

- Bazen mikrafon sende olsa bile söyleyecek sözün yoktur. Condition Variable bu problemi çözer. Senin beklemekte olduğun condition

m1: mutex lock is to control the access to this critical session

c1: condition variable is for one condition

c2: condition variable is for another condition

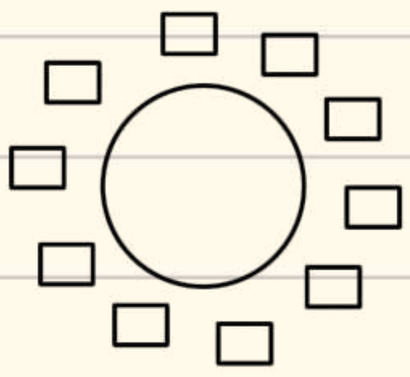
```
int pthread_cond_wait(pthread_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond); //tek bir waiting thred'i uyandır
int pthread_cond_broadcast(pthread_cond_t *cond); //tüm waiting thread'leri uyandır.
```

condition variable, mutex sadece threadlerde kullılabiliyor

Mutexlerde herkes participant olmalı yani dahil olmalı

Semaphores

```
int sem_init( sem_t *sem,
              int pshared,
              unsigned int value);
```



10 tane mikrafon var diyelim

10 tane kişi olsun. mutex 1 mikrafona erişimi kontrol ediyordu.

İlk gelen 10 process'e bu kaynakları sandalyeleri verceğim.

```
sem_t   s;
sem_init ( &s, 0, 10); // 10 müsait kaynağa sahibim
```

0 threadler ile kullılacağını belirtir.
1 olursa process ile kullanılacak demek

```
// Bir thread her zaman kaynağı ayırmak ister
sem_wait ( &s) ; // s'in değerini bir azalt.

    ⋮

// Kaynakla iş bittikten sonra
sem_post ( &s); // s'nin değeri bir artsın.
```

Senkronizasyon & Deadlocklar

Reader - writer problem

Review of producer - consumer problem

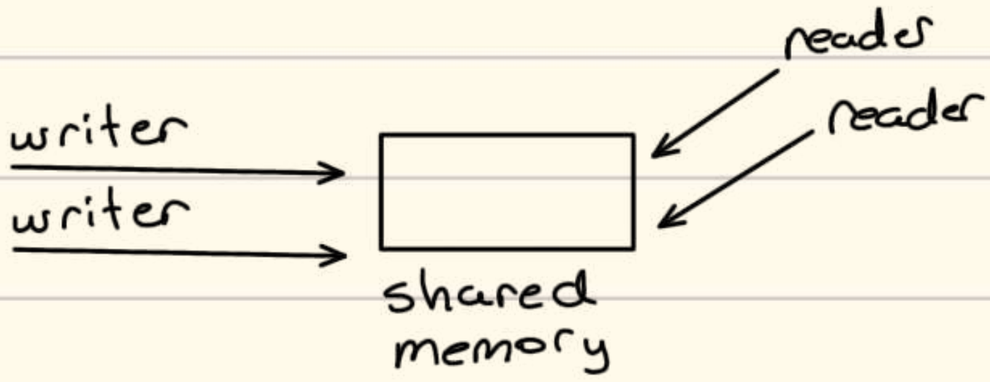
Ring buffer (Queque) implementation

Reader - Writer problem (Multiple writer - Multiple reader)

Reader: Sadece datayı okur ve data üzerinde yazma yapmaz.

Writer: Data üzerinde hem okuma hem yazma yapar

Problem: Ortak (Shared) memory'e hem yazan hem de okuyanlar var.



Sadece readerlar olsa senkronizasyona gerek yok. Çünkü datayı değiştirmiyorlar dolayısıyla bir şey eklemiyorlar.
Ama writing işlemi varsa datada bozulma data corruption olmaması için senkronizasyon gerekiyor.

Reader - Writer Problem: 1. Çözüm Mutex

```
void read() {
    lock(&m)
    // do read stuff
    unlock(&m)
}

void write() {
    lock(&m)
    // do write stuff
    unlock(&m)
}
```

Bir tane mikrafon var. Kim mikrafonu sahipse o konuşur data corruption yok

mutual exclusion var

t1 (read) ⟷ t2 (read) Bu iki thread de read operasyonunu çalıştırıyor.

Bunların arasında da senkronizasyon var. Ama ben istemiyorum!

readerlar birbirlerini beklemesin

Issue: Only one reader/writer at a time

Readers block other readers

Reader - Writer Problem: 2. Çözüm Spinlocks

```
void read() {
    while (writing) { /*spin*/ }
    reading = 1;
    // Read operation
    reading = 0;
}

void write() {
    while (reading || writing) { /*spin*/ }
    writing = 1;
    // Write operation
    writing = 0;
}
```

Yazma işlemi varsa spin

Birden fazla threadin buraya girmesi va

Başka writer ile senkronizasyo

Issiue: Race condition, multiple readers and the writers at the same time

Reader - Writer Problem: 3. Çözüm Condition Varıable

```
void read() {
    lock (&m);
    while (writing)
        cond_wait ( &cv , &m);
    reading++
    unlock ( &m)
    // reading part
    lock (&m)
    reading --
    cond_signal ( &cv);
    unlock (&m)
}
```

```
void writer () {
    lock (&m);
    while (reading, writing)
        cond-wait (&cv, &m);
    writing ++
    // writing
    writing --
    cond_signal ( &cv );
    unlock (&m)
```

Issiue: Eğer reader sayısı çok fazlaysa writer hiç çalışmayabilir. Ve starvation olabilir Bu problemin çözümü için writer'a öncelik verilebilir.

Reader - Writer Problem: Final

Final Solution (Writer Priority)

Writer Code

```
void writer() {
    lock(&m);
    writers++;
    while (reading || writing)
        cond_wait(&cv, &m);
    writing++;
    unlock(&m);

    // Write here

    lock(&m);
    writing--;
    writers--;
    cond_broadcast(&cv);
    unlock(&m);
}
```

Reader Code

```
void read() {
    lock(&m);
    while (writers) // Check waiting writers
        cond_wait(&cv, &m);
    while (writing) // Ensure no active writer
        cond_wait(&cv, &m);
    reading++;
    unlock(&m);

    // Read here

    lock(&m);
    reading--;
    cond_signal(&cv);
    unlock(&m);
}
```

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Example

a. Thread 1 holds lock A and requests lock B
b. Thread 2 holds lock B and requests lock A

Deadlock occurs when

- Threads that are already holding locks request new locks
- The requests for new locks are made concurrently
- Two or more threads form a circular chain in which each thread waits for a lock which is held by the next thread in the chain

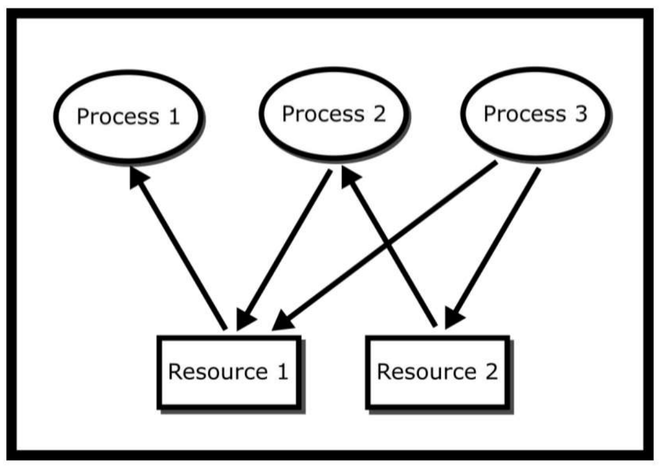
Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result.

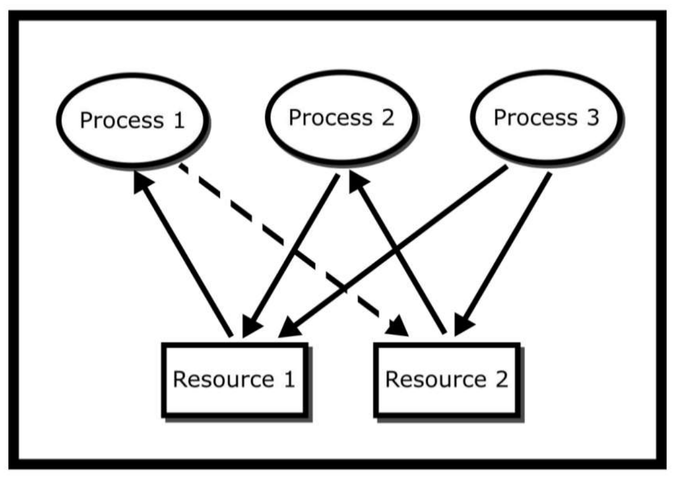
Deadlock: Resource allocation graphs



If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will deadlock.

We can detect a deadlock by traversing the graph and searching for a cycle using a graph traversal algorithm, such as the Depth First Search (DFS).

Deadlock: Resource allocation graphs



If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will deadlock.

We can detect a deadlock by traversing the graph and searching for a cycle using a graph traversal algorithm, such as the Depth First Search (DFS).

Coffman conditions

Deadlock can happen if and only if the four Coffman conditions are satisfied.

karşılıklı Dışlama

1. **Mutual Exclusion:** No two processes can obtain a resource at the same time. Kritikal section da sadece 1 thred olabiliyorsa
2. **Circular Wait:** There exists a cycle in the Resource Allocation Graph: Circular Wait
   a. there exists a set of processes {P1,P2,. . . } such that P1 is waiting for resources held by P2, which is waiting for P3,. . . , which is waiting for P1.
3. **Hold and Wait:** Once a resource is obtained, a process keeps the resource locked. Resource alıp bekliyorsam
4. **No Pre-emption:** Nothing can force the process to give up a resource. Neden beklion diye uyaran yoksa

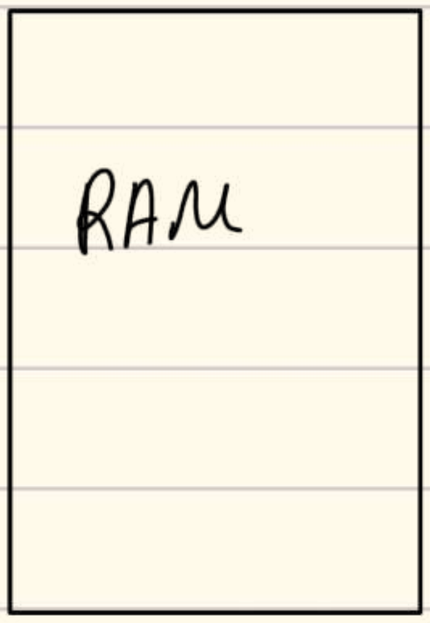
Proof in the book

See also simulations in https://en.wikipedia.org/wiki/Deadlock#Individually_necessary_and_jointly_sufficient_conditions_for_deadlock

Virtual Memory & IPC

> mmap

> Pipes



Virtual memory büyüklüğü OS 32 bit ise o zaman bir register 32 yani max 2^{32} adresleri yapabiliyorum

Processlerin memorysini organize ederken bunu kullanıyorum.

Virtual memory'i kitap olarak düşünüyorum. Kitabımın içinde sayfalarım var. Pagelere sahip ve pagelerin sayfa numaraları var. Bu kez processleri falan saklarken hangi sayfada olduğuna bakıyorum.

Terminology

A **page** is a block of virtual memory.

- A typical block size on Linux is 4KiB
  - or 2^12 addresses,
- one can find examples of larger blocks

a **frame** or sometimes called a **'page frame'** is a block of physical memory or RAM – Random Access Memory.

- A frame is the same number of bytes as a virtual page or 4KiB on our machine.

Page Number	Frame Number
0	42
1	30
2	24
•••	•••

Figure 9.1: Explicit Frame Table

Page table

32 bit PC için 1tane sayfa 4KiB ise 2^{12} ile adreslenir Dolayısıyla 2^{20} sayfa olabilir

- 32-bit computers
  - 2^32 address/2^12(address/page) = 2^20 pages.

- 64-bit computer
  - 2^64 address/2^12(address/page) = 2^52 pages =~ 10^15

- To access a particular byte in a frame, an MMU goes from the start of the frame and adds the offset

Page Number	Frame Number
0	42
1	30
2	24
•••	•••

Networking

TCP UDP , Port numbers, Binding port

Sockets , Programming Interface API

HTTPS, Web Sockets

Devamı kaldı

Eksik