Operating Systems
سیستمهای عامل

اسلایدهای شماره ۲

دکتر خانمیرزا

h.khanmirza@kntu.ac.ir

دانشکده کامپیوتر

دانشگاه صنعتی خواجه نصیرالدین طوسی

مفاهیم اساسی در سیستم عامل

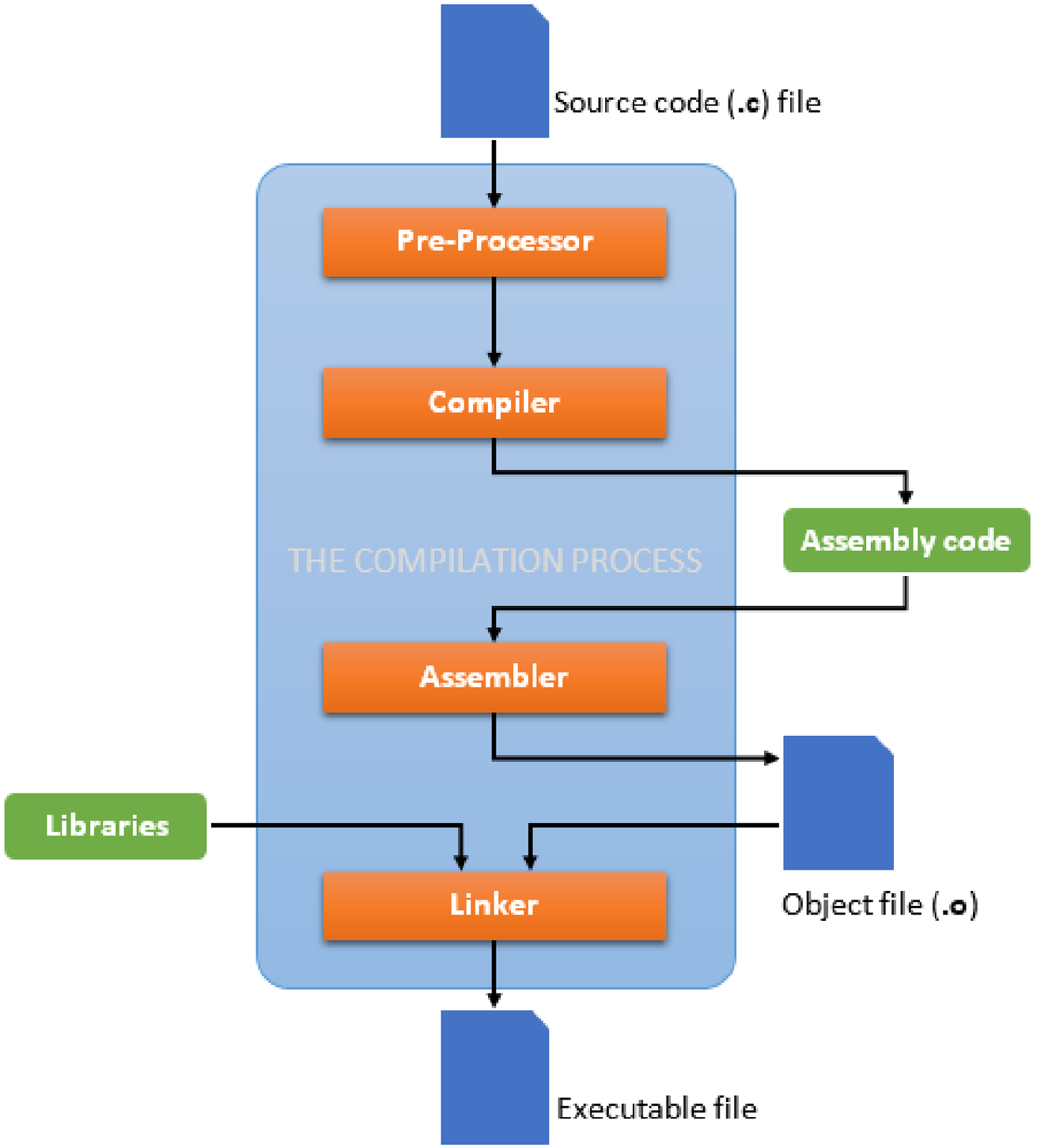
- در تمامی سیستم عامل‌ها چهار مفهوم پیاده‌سازی می‌شود که ابتدا به معرفی آنها می‌پردازیم.

- یکی از سرویسهایی که از سیستم عامل انتظار می‌رود اجرای برنامه‌هاست.
- قبل از معرفی مفاهیم اساسی ابتدا به معرفی ساختار برنامه‌ها و فرآیندها می‌پردازیم

مراحل کامپایل برنامه‌ها

- بعد از نوشتن شدن، برنامه برای قابل اجرا شدن باید کامپایل (compile) شود.
- کامپایل یک برنامه شامل چند گام است
  - اینجا بیشتر برنامه‌ها به زبان C مد نظر است.

مراحل کامپایل برنامه‌ها



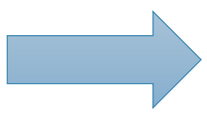
https://medium.com/@laura.derohan/compiling-c-files-with-gcc-step-by-step-8e78318052

مرحله اول پیش‌پردازش (pre-process)

- حذف کامنتها از کد برنامه
- اضافه شدن کد فایلهای با پسوند h. که در ابتدای برنامه include شده‌اند
  - روش کار کردن با فایلهای include یا کتابخانه‌ها متفاوت است. کتابخانه‌ها همانطور که توضیح داده خواهد شد به برنامه متصل (link) میشوند ولی برنامه‌های include شده بطور کامل در فایل برنامه کپی شده و یک فایل جدید بزرگ ساخته و همین فایل کامپایل می‌شود.
- جایگذاری ماکروها (`#define, #ifdef`)
  - ماکروها بطور کامل تبدیل به کد می‌شوند یعنی به شکل رشته در برنامه جایگذاری می‌شوند

- خروجی این مرحله یک فایل با پسوند i. است.
  - این خروجی با دستور `gcc -E test.c` قابل دیدن است

```
1  #define __x 10
2
3  int y = 11;
4  if(y < __x){
5
6  }
```



```
1  #define __x 10
2
3  int y = 11;
4  if(y < 10){
5
6  }
```

مرحله اول پیش‌پردازش (pre-process)

```c
#include <stdio.h>

#define __x 11

int main(void){

  int input;
  scanf("%d", &input);

  if(input < __x)
    printf("Your input was less than threshold");
  else
    printf("Your input was greater than threshold");

  return 0;
}
```

File size: 227 bytes

```
__attribute__((availability(watchos,introduced=4.0)));
# 367 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 3 4
extern const int sys_nerr;
extern const char *const sys_errlist[];

int asprintf(char ** restrict, const char * restrict, ...) __attribute__((__format__ (__printf__, 2, 3)));
char *ctermid_r(char *);
char *fgetln(FILE *, size_t *);
const char *fmtcheck(const char *, const char *);
int fpurge(FILE *);
void setbuffer(FILE *, char *, int);
int setlinebuf(FILE *);
int vasprintf(char ** restrict, const char * restrict, va_list) __attribute__((__format__ (__printf__, 2, 0)));
FILE *zopen(const char *, const char *, int);

FILE *funopen(const void *,
                int (* _Nullable)(void *, char *, int),
                int (* _Nullable)(void *, const char *, int),
                fpos_t (* _Nullable)(void *, fpos_t, int),
                int (* _Nullable)(void *));
# 407 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_stdio.h" 1 3 4
# 31 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_stdio.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_common.h" 1 3 4
# 32 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_stdio.h" 2 3 4
# 42 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_stdio.h" 3 4
extern int __sprintf_chk (char * restrict, int, size_t,
     const char * restrict, ...);
# 52 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/secure/_stdio.h" 3 4
extern int __snprintf_chk (char * restrict, size_t, int, size_t,
      const char * restrict, ...);
extern int __vsprintf_chk (char * restrict, int, size_t,
      const char * restrict, va_list);
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
       const char * restrict, va_list);
# 408 "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 3 "os_test.c" 2
```

Included Part

```
int main(void){

 int input;
 scanf("%d", &input);

 if(input < 11)
  printf("Your input was less than threshold");
 else
  printf("Your input was greater than threshold");

 return 0;
}
```

File size: 23 Kbytes

مرحله دوم: کامپایل (compile)

- کامپایلر نتیجه گام قبلی را گرفته و کد میانی (Intermediate Representation) تولید می‌کند.
- معمولا این کد به زبان اسمبلی است.
- اگر بخواهید خروجی این گام را ببینید کافی است که فرمان `gcc -S test.c` را وارد کنید.

```
  .section  __TEXT,__text,regular,pure_instructions
  .build_version macos, 10, 15  sdk_version 10, 15
  .globl  _main                   ## -- Begin function main
  .p2align  4, 0x90
_main:
  .cfi_startproc
## %bb.0:
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register %rbp
  subq  $32, %rsp
  movl  $0, -4(%rbp)
  leaq  L_.str(%rip), %rdi
  leaq  -8(%rbp), %rsi
  movb  $0, %al
  callq _scanf
  cmpl  $11, -8(%rbp)
  movl  %eax, -12(%rbp)         ## 4-b
  jge LBB0_2
## %bb.1:
  leaq  L_.str.1(%rip), %rdi
  movb  $0, %al
  callq _printf
  movl  %eax, -16(%rbp)         ## 4-b
  jmp LBB0_3
LBB0_2:
  leaq  L_.str.2(%rip), %rdi
  movb  $0, %al
  callq _printf
  movl  %eax, -20(%rbp)         ## 4-byte Spill
LBB0_3:
  xorl  %eax, %eax
  addq  $32, %rsp
  popq  %rbp
  retq
  .cfi_endproc
                                ## -- End function
  .section  __TEXT,__cstring,cstring_literals
L_.str:                         ## @.str
  .asciz  "%d"

L_.str.1:                       ## @.str.1
  .asciz  "Your input was less than threshold"

L_.str.2:                       ## @.str.2
  .asciz  "Your input was greater than threshold"


.subsections_via_symbols

```

مرحله سوم: ترجمه به کد ماشین

- در مرحله سوم کد اسمبلی توسط برنامه assembler به کد ماشین تبدیل می‌شود.
- خروجی این مرحله یک فایل o. است و برای دیدن آن دستور `gcc -c test.c` را وارد کنید.
- این فایل حاوی object code است.
- البته که این فایل قابل خواندن نیست!

os_test.o

```
œˆÌ¸ à(__text__TEXT](8Ä__cstring__TEXT]LÖ__compact_unwind__LD∞ ÿh__eh_frame__TEXT—@¯
h2

p†
PUHâÂHÉÏ «E¸Hç=GHçu¯∞ËÉ}¯
âEÙçHç=+∞ËâE♦ÈHç=8∞ËâEÏ1¿HÉƒ ]√%dYour input was less than thresholdYour input was greater
than threshold]zRx
ê$ˇˇˇˇˇˇˇˇ]AÜC
N-G8-1-_main_printf_scanf
```

مرحله چهارم: لینک (link)

- در این مرحله هنوز برخی اسامی (symbols) هستند که مشخص نشده‌اند (resolve). یعنی این اسامی به آدرس تبدیل نشده‌اند.
  - اسم یا یک متغیر است و یا نام یک تابع (جایی از کد) که در هر دو حالت باید به آدرس حافظه تبدیل شود.

- اگر چندین فایل C قرار است در یک فایل باینری کامپایل شده و یک فایل اجرایی را بسازند object codeهای این چند فایل در هم ادغام شده و برخی اسامی resolve می‌شوند.
  - `gcc file1.c file2.c`

- گاهی اوقات ما از کتابخانه‌ها در برنامه‌هایمان استفاده می‌کنیم. در این حالت کد کامپایل شده آن کتابخانه‌ها در زمان اتصال به برنامه ضمیمه شده و آدرسهای اسامی جایگذاری می‌شود.
  - مقایسه کنید با روش include

مرحله چهارم: لینک (link)

- اتصال کتابخانه‌ها به کد به دو شکل انجام می‌شود
  - روش استاتیک (Static Library)
  - روش پویا (Dynamic Library)

- در اتصال کتابخانه‌های استاتیک، کد کامپایل شده در کد باینری قابل اجرای برنامه کپی می‌شود

- در اتصال پویا فقط نام کتابخانه مشخص شده و فایل کتابخانه‌ها باید در کنار فایل اجرایی قرار بگیرد.
  - مانند فایلهای dll در ویندوز و so در لینوکس
  - فراخوانی و استفاده از کتابخانه‌های پویا در زمان اجرا (run time) محقق می‌شود.

ساختار برنامه‌ها

- اطلاعات برنامه اجرایی در بخشهای (section) مختلفی قرار داده می‌شود.
  - این بخش‌ها برای سرویس‌دهی مناسب در حافظه از هم جدا می‌شوند
  - مثلا بخش کد برنامه هیچگاه عوض نمی‌شود ولی مقدار متغیرها در زمان اجرا عوض می‌شود

- یک برنامه حداقل دارای دو بخش است
  - بخش code و یا TEXT
  - در بخش کد دستورات ماشین ترجمه شده قرار می‌گیرند
  - بخش Data
  - در این بخش داده‌های static و global (سراسری) قرار می‌گیرند
  - بخش DATA خود به دو بخش تقسیم می‌شود
    - بخش متغیرهای مقداردهی شده در زمان کامپایل (initialized data)
    - بخش متغیرهای بدون مقدار اولیه در زمان کامپایل (uninitialized data - BSS)

- این بخشها را می‌توان با کمک دستورات shell مشاهده کرد

ساختار برنامه‌ها

- دستور objdump که عمدتا برای بررسی و دیدن برنامه‌های کامپایل شده بکار می‌رود

```
> objdump --section-headers a.out

a.out: file format Mach-O 64-bit x86-64


Sections:
Idx Name          Size      Address          Type
  0 __text        0000005d 0000000100000ed0 TEXT
  1 __stubs       0000000c 0000000100000f2e TEXT
  2 __stub_helper 00000024 0000000100000f3c TEXT
  3 __cstring     0000004c 0000000100000f60 DATA
  4 __unwind_info 00000048 0000000100000fac DATA
  5 __got         00000008 0000000100001000 DATA
  6 __la_symbol_ptr 00000010 0000000100002000 DATA
  7 __data        00000008 0000000100002010 DATA
```

ساختار برنامه‌ها

- دستور size که اندازه هر کدام از بخشهای برنامه را نشان می‌دهد

```
> size os_test.o

__TEXT __DATA __OBJC others dec hex
233    0      0      32     265 109

> size a.out
__TEXT        __DATA        __OBJC        others        dec    hex
4096   4096   0      4294975488   4294983680   100004000
```

- دقت کنید که اندازه بخشهای برنامه قبل از لینک و بعد از لینک متفاوت است. چرا؟

ساختار برنامه‌ها

- اگر در برنامه قبلی متغیر input را به صورت سراسری تعریف کنیم

```c
#include <stdio.h>

#define __x 11

int input;

int main(void){
  scanf("%d", &input);
  if(input < __x)
    printf("Your input was less than threshold");
  else
    printf("Your input was greater than threshold");

  return 0;
}
```

ساختار برنامه‌ها

- برای برنامه جدید

```
> size os_test2.o

__TEXT  __DATA  __OBJC others dec hex
239     4       0      32     275 113
```

- قبلا بخش داده اندازه 0 داشت ولی حالا برای یک متغیر سراسری integer اندازه ۴ بایت در نظر گرفته شده است

ساختار برنامه‌ها

- اگر object code را decompile بکنیم متوجه خواهیم شد که برنامه جدید دارای دو بخش است

```
> objdump -D os_test2.o

_main:
       0: 55 pushq %rbp
       1: 48 89 e5 movq %rsp, %rbp
       4: 48 83 ec 10 subq $16, %rsp
       8: c7 45 fc 00 00 00 00 movl $0, -4(%rbp)
....


Disassembly of section __DATA,__common:
_input:
...

> objdump –t os_test2.o

SYMBOL TABLE:
0000000000000110 g     O __DATA,__common _input
0000000000000000 g     F __TEXT,__text _main
0000000000000000         *UND* _printf
0000000000000000         *UND* _scanf
```

ساختار برنامه‌ها

- در این چهار گام برنامه نوشته شده کامپایل شده و به یک فایل قابل اجرا تبدیل می‌شود.
- زمانی که می‌خواهیم یک فایل را در سیستم عامل اجرا کنیم گام لود (load) انجام می‌گیرد

- لود کردن برنامه در حقیقت خواندن محتوای برنامه از دیسک و کپی آن به حافظه است به طریقی که قابل اجرا باشد

- این کار توسط برنامه لودر (loader) انجام می‌گیرد که بخشی از سیستم عامل است.

- جزئیات عملکرد لودرها بر اساس قالب فایل اجرایی و نیز سیستم عامل متفاوت است

- فایلهای اجرایی قالبهای مختلفی دارند
  - a.out
  - ELF
  - COFF

ساختار برنامه‌ها

- بر اساس عملکرد لودرهای مختلفی وجود دارد

- لودر مطلق (Absolute Loader): این لودر همواره برنامه‌ها را در جای مشخصی از حافظه لود می‌کند.

- لودر جابجا کننده (Relocating Loader): این لودرها در زمان لود کردن، آدرس‌های داخل باینری را تغییر می‌دهند تا با آدرس لود مشخص شده برای لود باینری در حافظه تطابق داشته باشند.
  - برنامه‌ها معمولا از آدرس صفر کامپایل می‌شوند
  - ممکن است بخشهای مختلف برنامه در جاهای مختلفی از حافظه لود شود که بعدا صحبت خواهد شد.

- لودر پویا (Dynamic Linking Loader):
  - مشکل اصلی لینک استاتیک این است که تمامی کتابخانه‌های مشترک به صورت تکراری در تمامی فایلهای اجرایی کپی می‌شوند که این باعث می‌شود فایلهای اجرایی بزرگ شده و حجم زیادی در دیسک و حافظه اشغال کنند.
  - مثلا بیشتر برنامه‌ها به کتابخانه‌های گرافیکی نیاز دارند و این کتابخانه‌ها حجم زیادی دارند.
  - سیستم عامل‌ها کتابخانه‌های سیستمی را در محلی از حافظه لود کرده و برنامه‌هایی را که بدان نیاز دارند در زمان لود به آنها لینک می‌کنند
  - برای این کار اسامی و آدرسهای فایل باینری را در زمان لود تغییر می‌دهند
  - خود کتابخانه‌های مشترک (shared libraries) با قالب PIC (Position Independent Code) کامپایل می‌شوند

ساختار فرآیندها

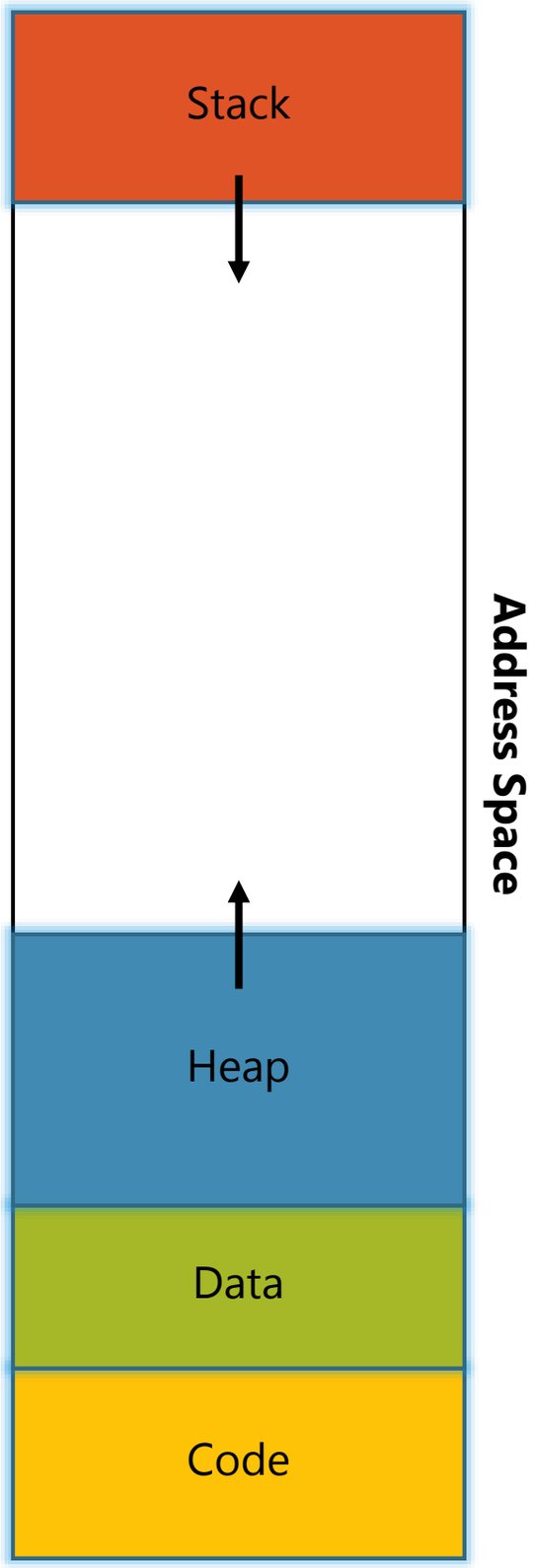
- پس از لود، برنامه را فرآیند می‌نامیم
- لودرها علاوه بر تصحیح آدرسها و کپی برنامه ساختار فرآیند در حافظه را برای برنامه‌ها تشکیل می‌دهند
- فرآیندها علاوه بر بخشهای کد و داده حداقل دارای دو بخش دیگر نیز هستند:

- بخش heap:
  - در این بخش متغیرهایی که اندازه آنها در زمان کامپایل مشخص نیست و در زمان اجرا مشخص می‌شود قرار داده می‌شود.
  - عمدتا داده‌هایی که با فراخوانی malloc ایجاد می‌شوند از این نوع هستند:

```
int x;
scanf (“%d”, &x);
int* p = (int *) malloc(x * 4);
```

ساختار فرآیندها

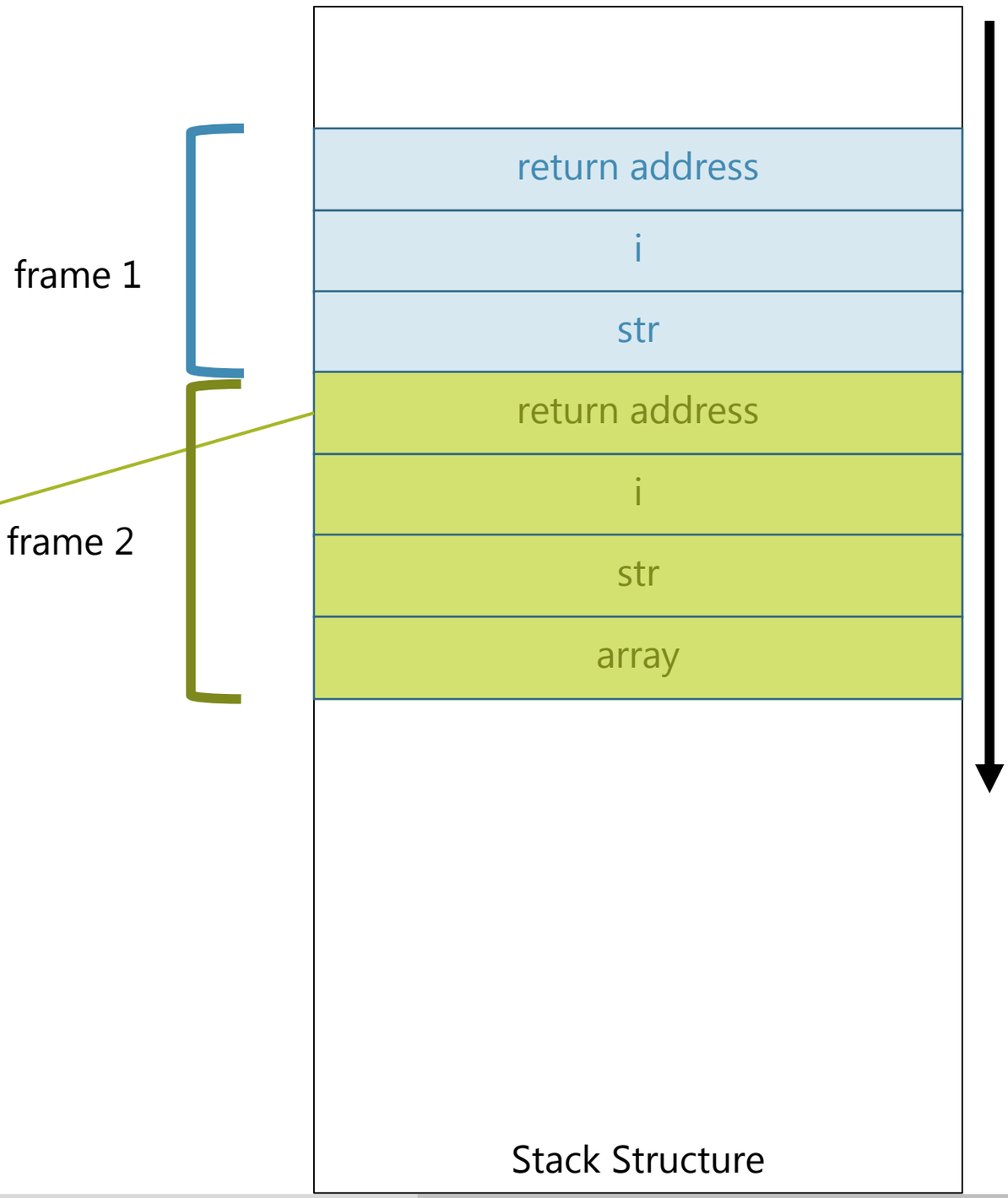
- بخش پشته (stack):
  - تمامی متغیرهای دیگر که از نوع محلی (local) در داخل توابع تعریف می‌شوند و نیز آرگومانها در پشته قرار می‌گیرند.
- چون اندازه این دو بخش متغیر است در دو بخش فضای آدرس قرار داده می‌شوند تا بتوان از کل فضای آدرس استفاده بهینه نمود
- به شکل تاریخی همیشه پشته در انتهای آدرس قرار گرفته و همیشه به سمت آدرسهای کمتر رشد می‌کند.



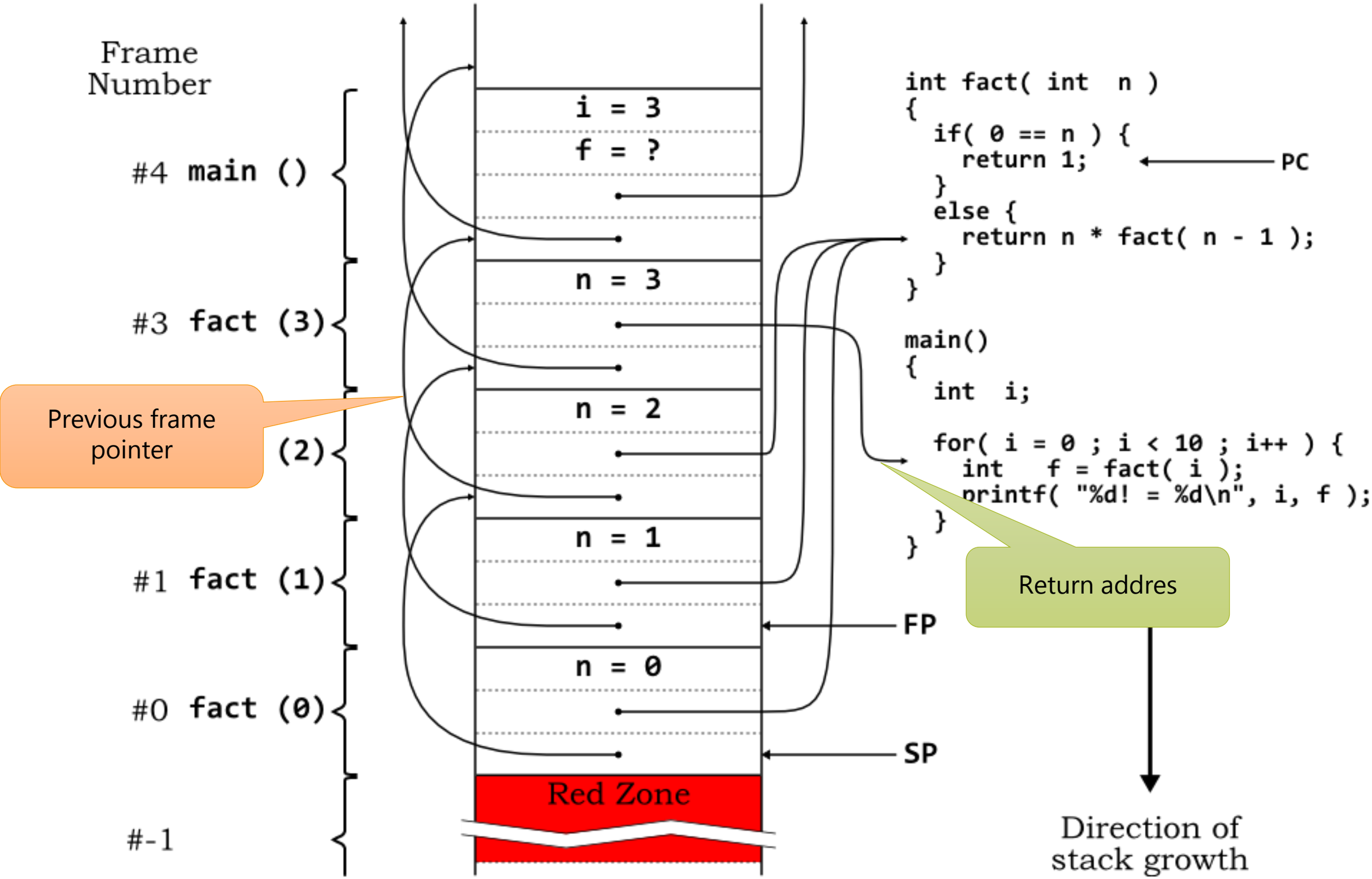
ساختار فرآیندها

- بخش پشته بخش بسیار مهمی از برنامه است که فراخوانی تودرتوی توابع را ممکن می‌کند.

```
1  int b(int n, char* string)
2  {
3    int array[] = {1,2,3,n};
4    printf("%s, %d", string, n);
5  }
6
7  void main(void) {
8    int  i = 0;
9    char* str = "quiet";
10   b(i, str)
11 }
```



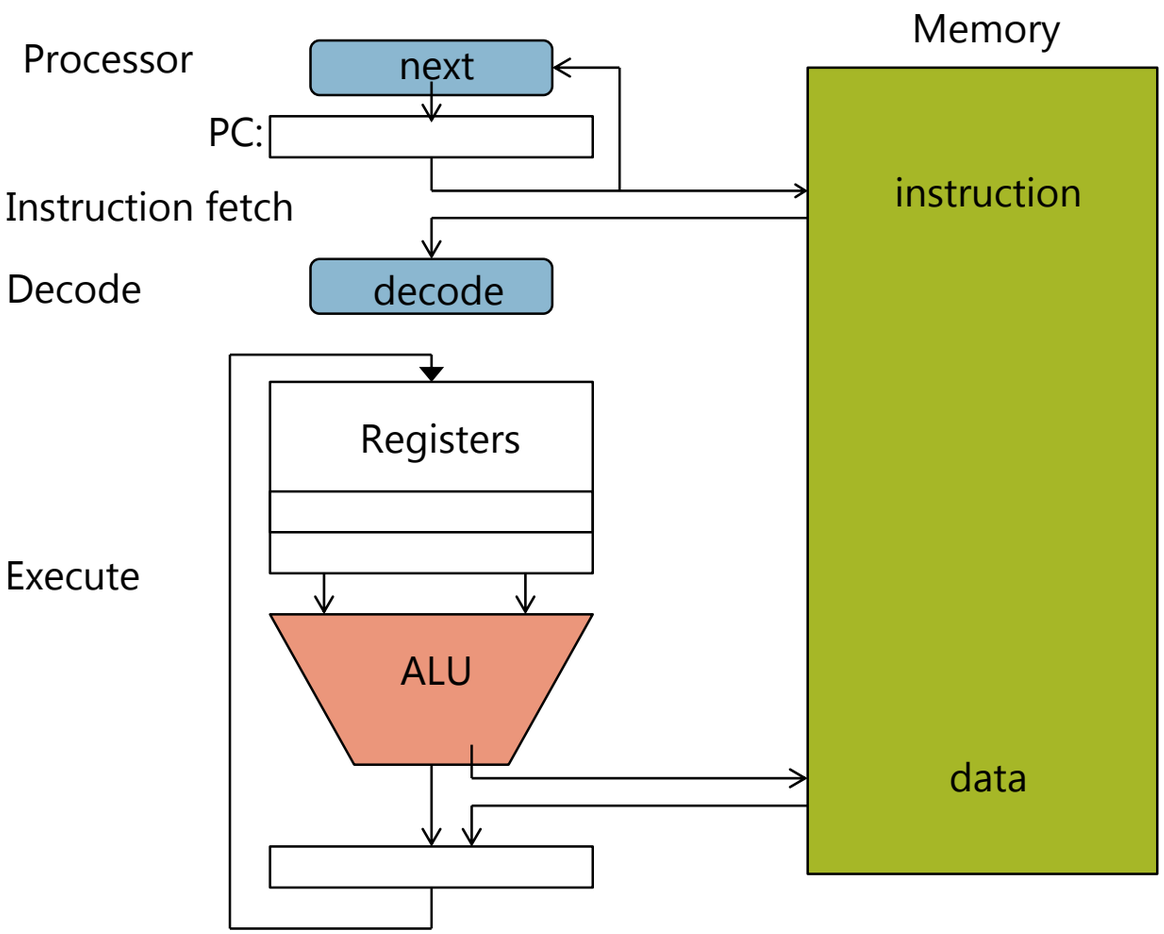
ساختار فرآیندها



مفاهیم اساسی در سیستم عامل

مفهوم اول – ریسمان

- چرخه اجرای فرآیند



- خواندن دستور از حافظه از محل PC (fetch)
- رمزگشایی (decode)
- اجرای دستور با کمک ثباتها
- نوشتن نتایج در ثبات و یا حافظه
- مقدار جدید PC (Program Counter)

- طبق این چرخه برای اجرای دستورات نیاز به ثباتها و برخی مقادیر در حافظه است

مفهوم اول – ریسمان

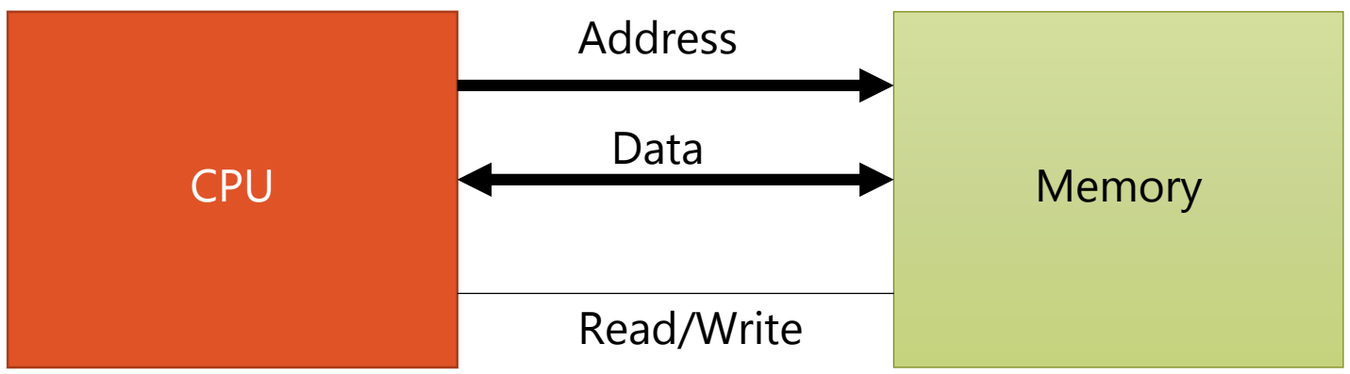
- ریسمان (thread)
- یک زمینه اجرایی یکتا که شامل PC، ثباتها، flagهای اجرایی و پشته است
- یک ریسمان در حال اجراست اگر مقادیر آن ریسمان در ثباتهای پردازنده باشد
- برخی ثباتها مقادیر دستورات را نگه میدارند و برخی دیگر زمینه اجرایی را مثل ثبات stack pointer

- مجموع آنچه برای ریسمان گفته شد وضعیت فعلی اجرایی ریسمان را مشخص می‌کند
- اگر این مقادیر را جایی ذخیره کنیم و بعد از مدتی همین مقادیر را در ثباتها و حافظه قرار دهیم اجرای ریسمان از همان محلی که مانده بود بدون کم و کاست ادامه می‌یابد

مفهوم دوم – فضای آدرس

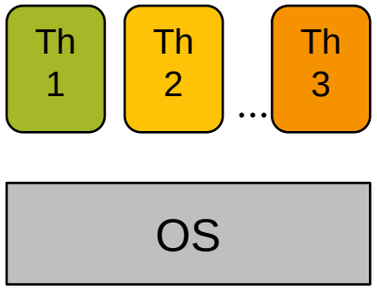
- به بخشی از حافظه که توسط یک ریسمان قابل دسترسی است فضای آدرس گفته می‌شود.

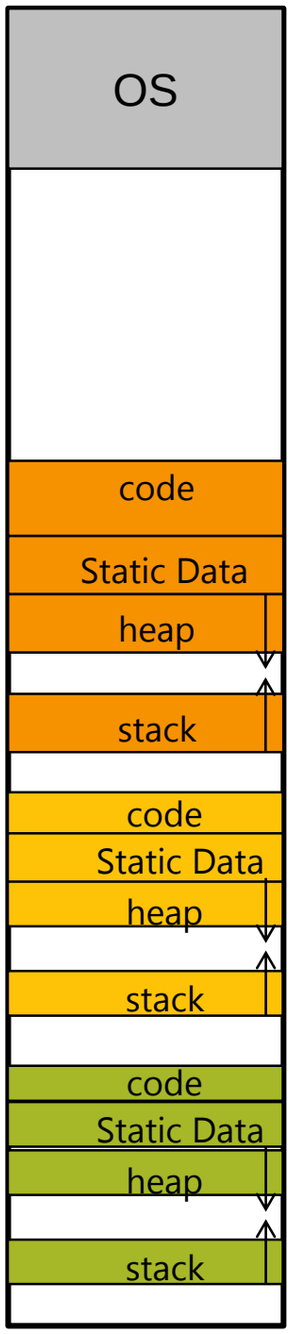
- مثلا اگر فضای آدرس یک ریسمان ۳۲ بیتی است یعنی از آدرس 0 تا آدرس $2^{32} - 1$)حدود ۴ گیگابایت) برای یک ریسمان قابل دسترسی است
- این آدرسها بر اثر اجرای دستورات تولید شده و به خط آدرس حافظه منتقل می‌شود
- انتظار می‌رود در طول مدت اجرای یک ریسمان تمامی آدرسهایی که از پردازنده تولید می‌شود همه در محدوده‌ی فضای آدرس تعیین شده باشد.



چند برنامگی

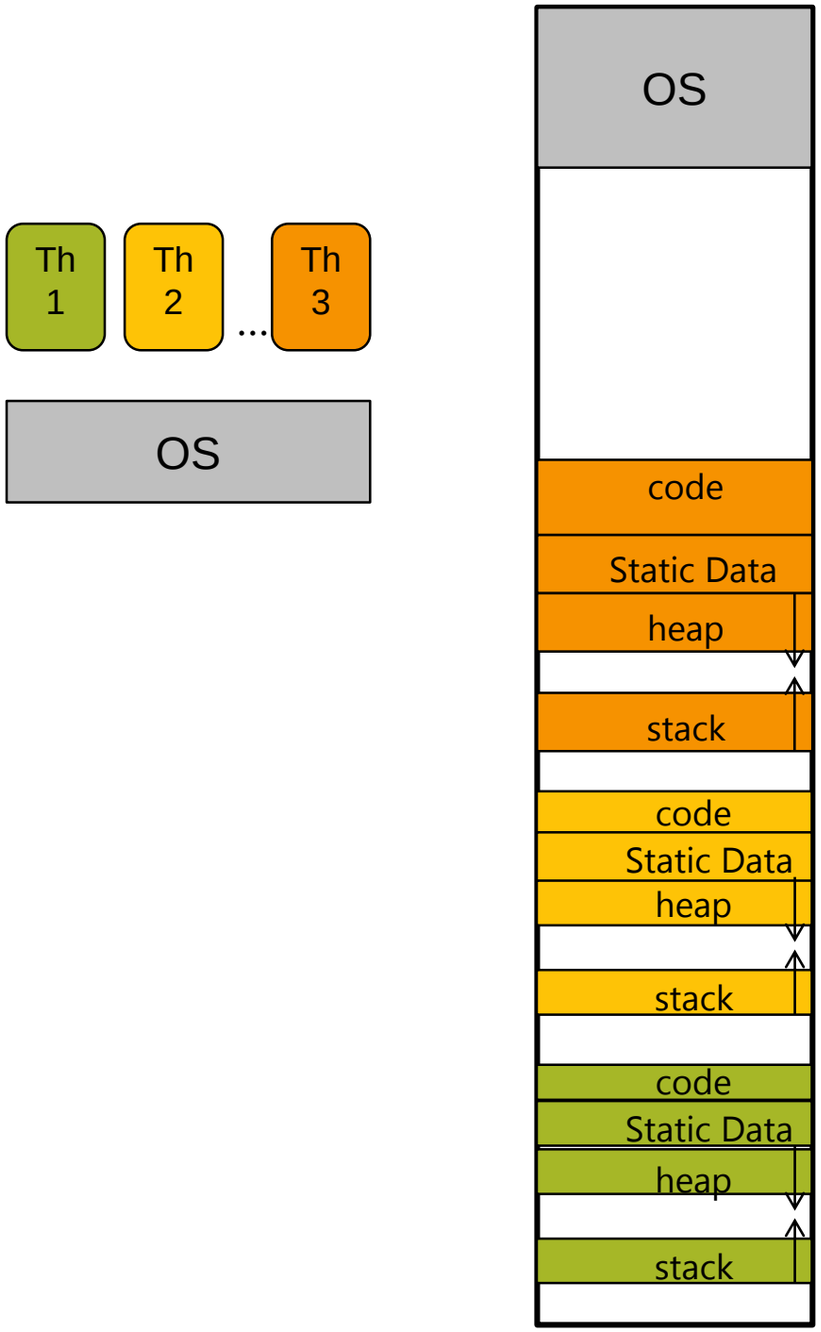
- اگر فقط یک پردازنده داشته باشیم و یک ریسمان برای اجرا وجود داشته باشد مشکلی نیست
- اگر یک پردازنده و چند ریسمان برای اجرا داشته باشیم چگونه آنها را اجرا کنیم؟
- بدیهی است که در ابتدا زمینه اجرایی همه آنها باید در حافظه باشد





چند برنامگی

- روش اجرای دسته‌ای (batch)
- در این روش ریسمانها در یک صف قرار می‌گیرند
- یکی از ریسمانها انتخاب شده و تا انتها اجرا می‌شود
- پس از آن ریسمان بعدی انتخاب می‌شود
- در سیستمهای قدیمی این روش معمول بوده چرا که آنها با کاربر در تعامل نبوده‌اند.

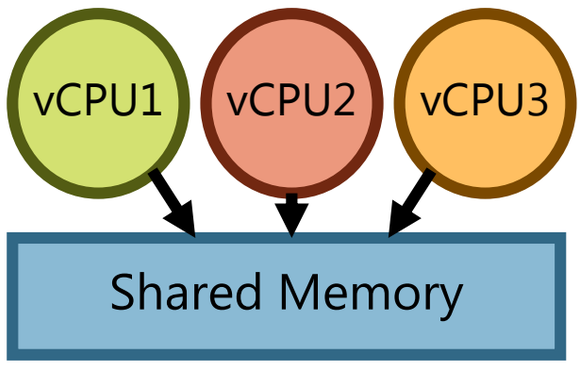


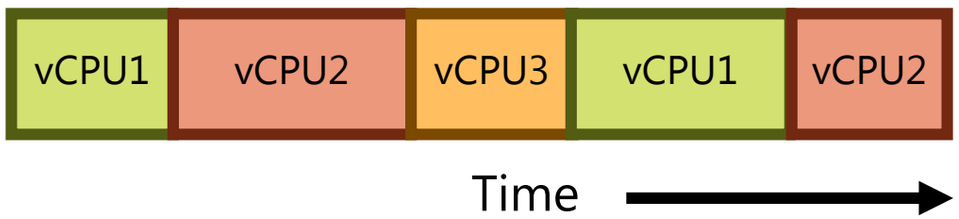
چند برنامگی

- روش اجرای همزمانی (concurrency)

- به هر ریسمان یک زمینه اجرایی مجازی یا پردازنده مجازی اختصاص می‌دهیم
  - زمینه اجرایی مجازی یک ساختمان داده‌ها در حافظه است که اطلاعات زمینه اجرایی در آن ذخیره می‌شود (مقادیر ثباتها و ...)

- در مدت زمان کارکرد سیستم، هر زمینه اجرایی مجازی در زمانهای کوتاه و به دفعات در زمینه اجرایی فیزیکی و واقعی (که همان پردازنده است) لود شده و اجرا می‌شود
  - زمینه اجرایی فیزیکی در زمینه اجرایی مجازی ذخیره شده
  - سپس زمینه اجرایی مجازی بعدی در پردازنده لود شده و اجرا می‌شود

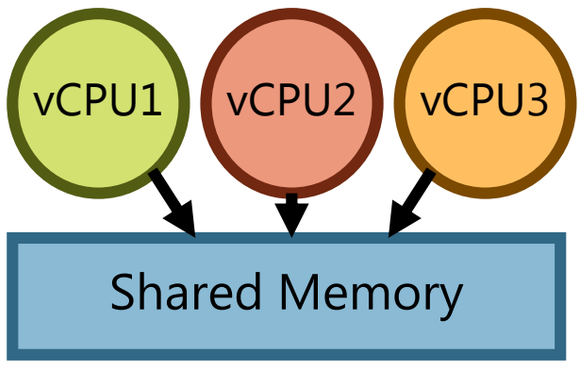


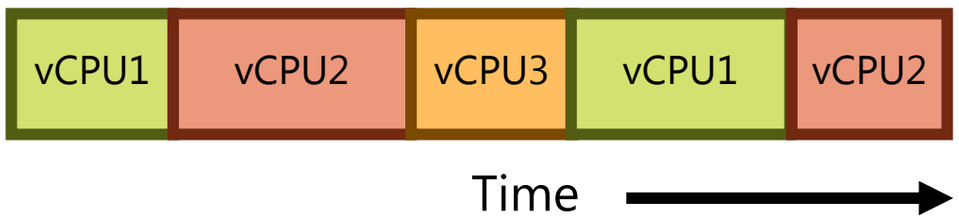


چند برنامگی

- اگر این تغییر زمینه به میزان کافی سریع باشد می‌توان این توهم را در ریسمانها ایجاد کرد که همه به طور موازی در حال اجرا هستند

- زمینه اجرایی فیزیکی چگونه بین زمینه‌های مجازی به اشتراک گذاشته می‌شود؟
  - با سخت‌افزار تایمر
  - با روش داوطلبانه yield
  - با IO





چند برنامگی

- مساله همزمانی

- این مساله در مورد تمامی منابع سیستم صادق است چرا که ما معمولا از هر منبع به تعداد و مقدار محدودی داریم
  - یک دیسک – یک کارت شبکه – یک کارت صدا – یک کارت گرافیک - ...

- در بیشتر این موارد از همزمانی استفاده می‌شود
  - یک abstraction برای هر منبع
  - تسهیم زمانی بین چند ریسمان

- گاهی اوقات تسهیم زمانی ممکن نیست مثل کارت شبکه؟

چند برنامگی

- در تعریف سیستم تا بحال هم زمینه‌های اجرایی مجازی و یا پردازنده‌های مجازی به همه منابع به صورت مشترک دسترسی دارند
  - همه منابع IO
  - حافظه (RAM) – که شامل بخشهای کد،داده و پشته هر ریسمان می‌شود
- در این تعریف ریسمانها به منابع همه دسترسی دارند
  - دسترسی به بخش داده‌های یکدیگر
  - دسترسی به بخش کد یکدیگر
- این مدل برای به اشتراک گذاری منابع بین چند ریسمان بسیار خوب است
- اما از جهت محافظت (protection) اصلا خوب نیست
  - دستکاری عمدی یا سهوی (بخاطر باگ) ریسمان دیگر

چند برنامگی

- نکته جالب این است که تا سال ۲۰۰۰ مدل بدون محافظت رایج بوده است
- سیستمهای نهفته (embedded) که همین حالا هم دارای چنین سیستمی هستند
- سیستم عاملهای windows 3.1 و MacOsهای قدیمی
  - که بسیار crash میکردند و مدل چندبرنامگی در آنها فقط با yield بود.
  - به عبارتی ریسمانها باید آنقدر سریع خودشان و به شکل داوطلبانه در بین کد yield میکردند تا این مکانیسم کار کند.

چند برنامگی

- آیا ریسمانها به حافظه سیستم عامل هم دسترسی دارند؟
- آیا سیستم عامل می‌تواند از خود در برابر دسترسیهای غیرمجاز محافظت کند؟
- آیا سیستم عامل می‌تواند از دسترسی غیرمجاز ریسمانها به یکدیگر هم محافظت کند؟

- دقت کنید که خود سیستم عامل یک نرم‌افزار است و باید بر روی پردازنده اجرا شود.
  - زمانی که یک ریسمان دیگر در حال اجراست سیستم عامل به عملکرد آن ریسمان نظارت ندارد
  - سیستم محافظت در سخت‌افزار پیاده‌سازی می‌شود

- اگر قرار بود سیستم عامل نظارت داشته باشد این مدل چطور باید پیاده‌سازی می‌شد و چه مشکلاتی بهمراه داشت؟

مفهوم سوم: فرآیند

- یک محیط اجرایی با دسترسیهای محدود است
- یک فضای آدرس محافظت شده با چندین ریسمان

- ریسمانها در داخل فضای آدرس فرآیند اجرا می‌شوند و به منابع فرآیند دسترسی مشترک دارند
- فضای آدرس فرآیندها محافظت شده است و چند فرآیند نمی‌توانند به منابع همدیگر دسترسی داشته باشند
- سیستم عامل هم به همین روش از دسترسی فرآیندهای دیگر محافظت می‌شود

- چرا هر دو مفهوم ریسمان و فرآیند وجود دارد؟

- ریسمانها منابع کمتری مصرف می‌کنند و در اجرا کاراتر هستند و فرآیندها محافظت را برقرار می‌کنند (بعدا بیشتر صحبت خواهد شد)

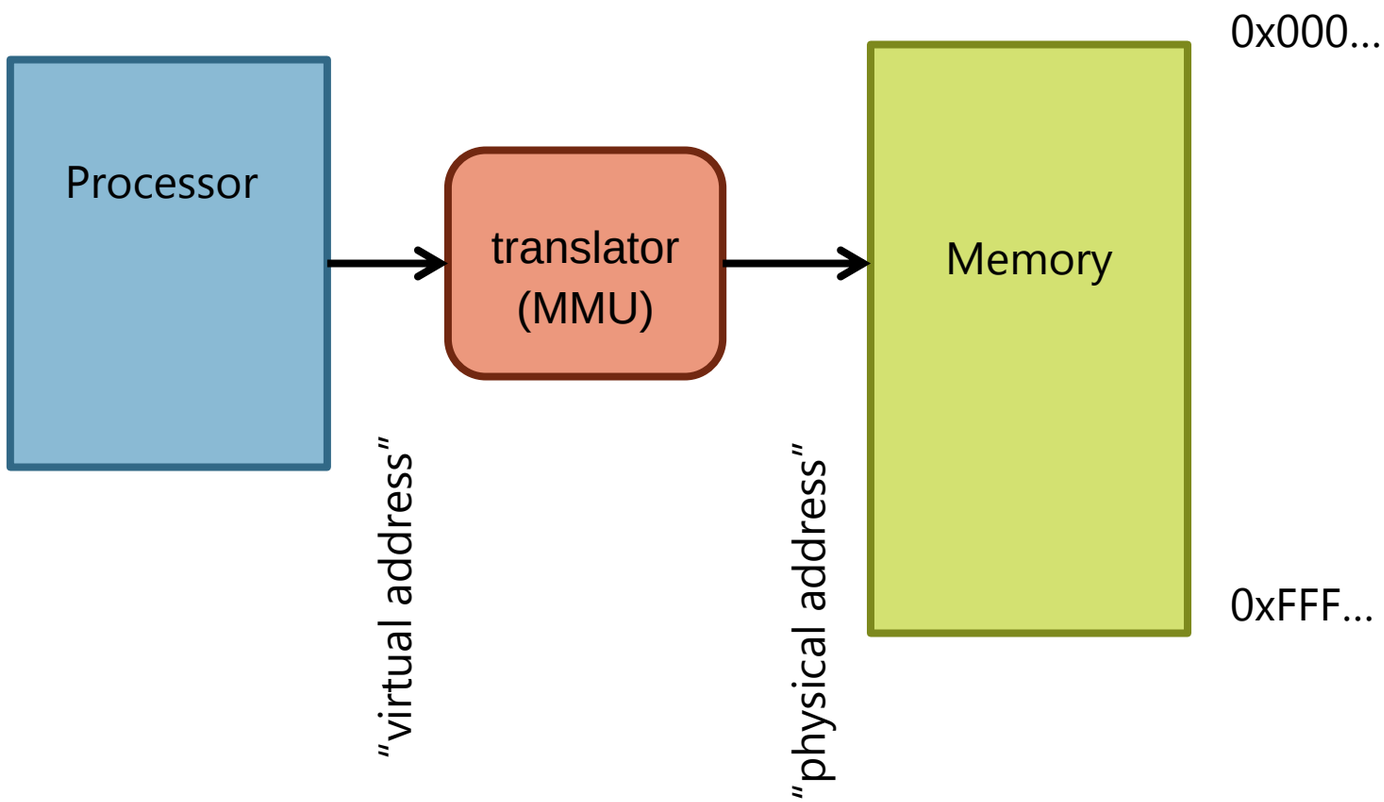
محافظت

- محافظت (Protection)
  - قابلیت اطمینان (reliability)
    - اگر سیستم عامل محاظفت نشود براحتی داده‌های آن دستکاری شده و در نتیجه کل سیستم از کار میفتد
  - امنیت (security)
    - محدود کردن فرآیندها در مورد کارهایی که می‌توانند انجام دهند
  - حریم خصوصی (privacy)
    - فرآیندها فقط به داده‌های خودشان دسترسی داشته باشند
  - انصاف (fairness)
    - فرایندها به سهم خود از منابع وادار شوند

محافظت

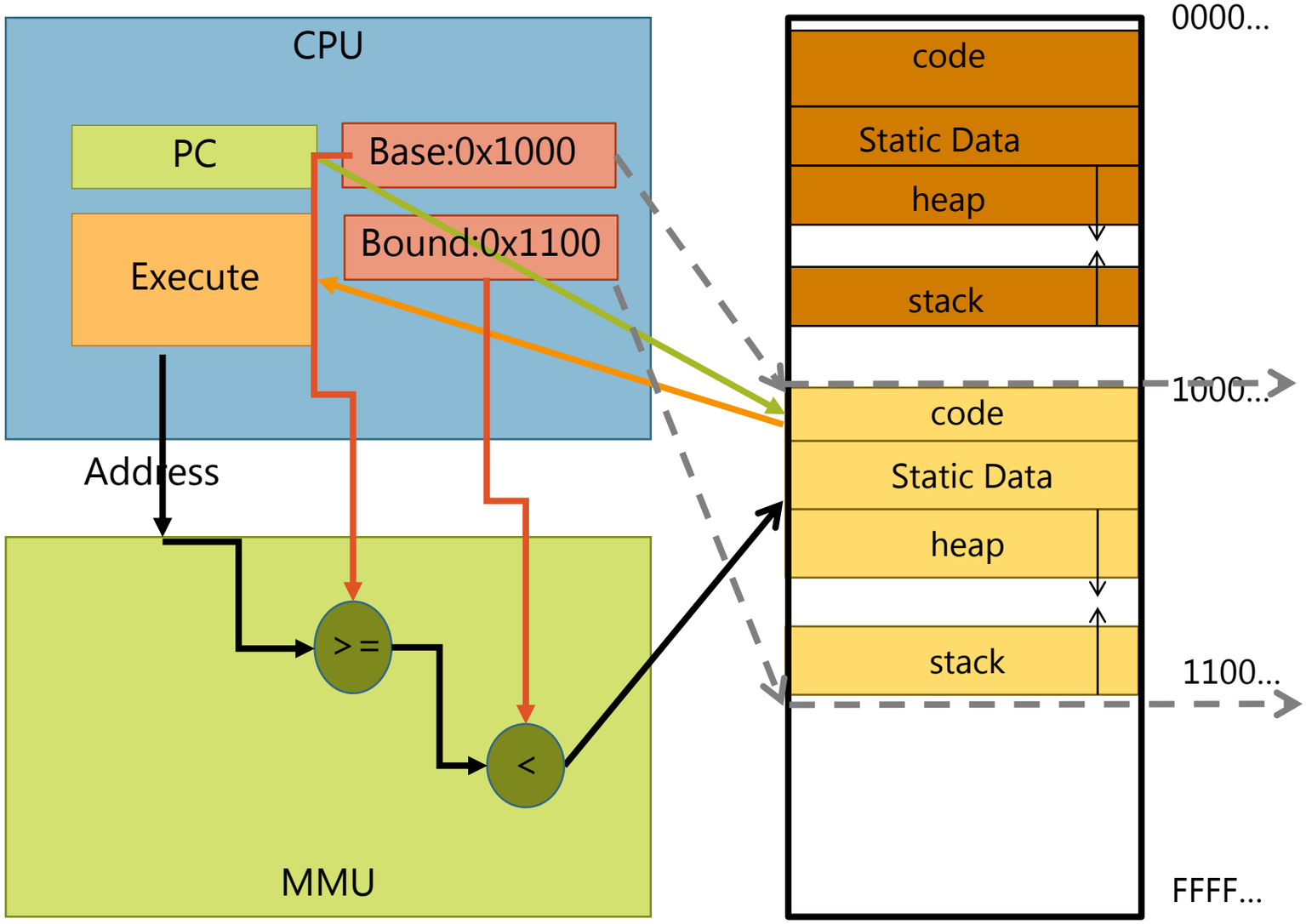
- محافظت چطور پیاده‌سازی می‌شود؟
  - به هر فرآیند یک فضای آدرس مجازی اختصاص داده می‌شود
  - آدرس مجازی تولید شده در پردازنده توسط یک سخت‌افزار ترجمه‌ی آدرس (MMU) که مابین پردازنده و حافظه اصلی قرار دارد به آدرس صحیح و واقعی ترجمه می‌شود
- ترجمه آدرس (Address Translation): تبدیل آدرس مجازی به آدرس فیزیکی

- مکانیزمهای دیگری هم هست که در اجرای دوگانه بحث خواهد شد



محافظت

- ساده‌ترین نوع محافظت تعیین یک محدوده آدرس برای هر فرآیند است
- این مکانیزم به Base & Bound (B&B) معروف است
- فضای آدرس هر فرآیند به صورت یکپارچه و پشت سرهم و بین دو آدرس است
- پردازنده دارای دو ثبات است که ابتدا و انتهای محدوده قابل دسترسی فرآیند در آنها مشخص می‌شود.
- ابتدا و انتهای فضای آدرس فرآیند در زمان لود و توسط سیستم عامل مشخص می‌شود.



محافظت

- در محافظت B&B
  - برنامه‌ها از آدرس صفر کامپایل می‌شوند
  - ولی در زمان لود در محل غیر صفر از حافظه لود می‌شوند
  - بنابراین نیاز به لودرهای جابجاکننده نیاز است

- اگر یک فرآیند بخواهد به آدرس مشخصی از فرآیند دیگر دسترسی داشته باشد؟
  - اصولا نمی‌داند که آن فرآیند در چه آدرسی لود شده است
  - حتی آدرس واقعی خودش را هم نمی‌داند – این اطلاعات در کرنل سیستم عامل ذخیره شده است