Duke Systems

CPS 310 / ECE 353

The View From Your Process

Jeff Chase

Duke University

Key points for today

- Virtualize!
  - CPU, memory, and “disk” (or other I/O devices)
- Example: C/Ux programming environment
- Where do program variables live?
  - Your args/env
  - Your stack
  - Your heap
- Under the hood: finding your stuff

Timeslicing in action

Virtualizing the CPU

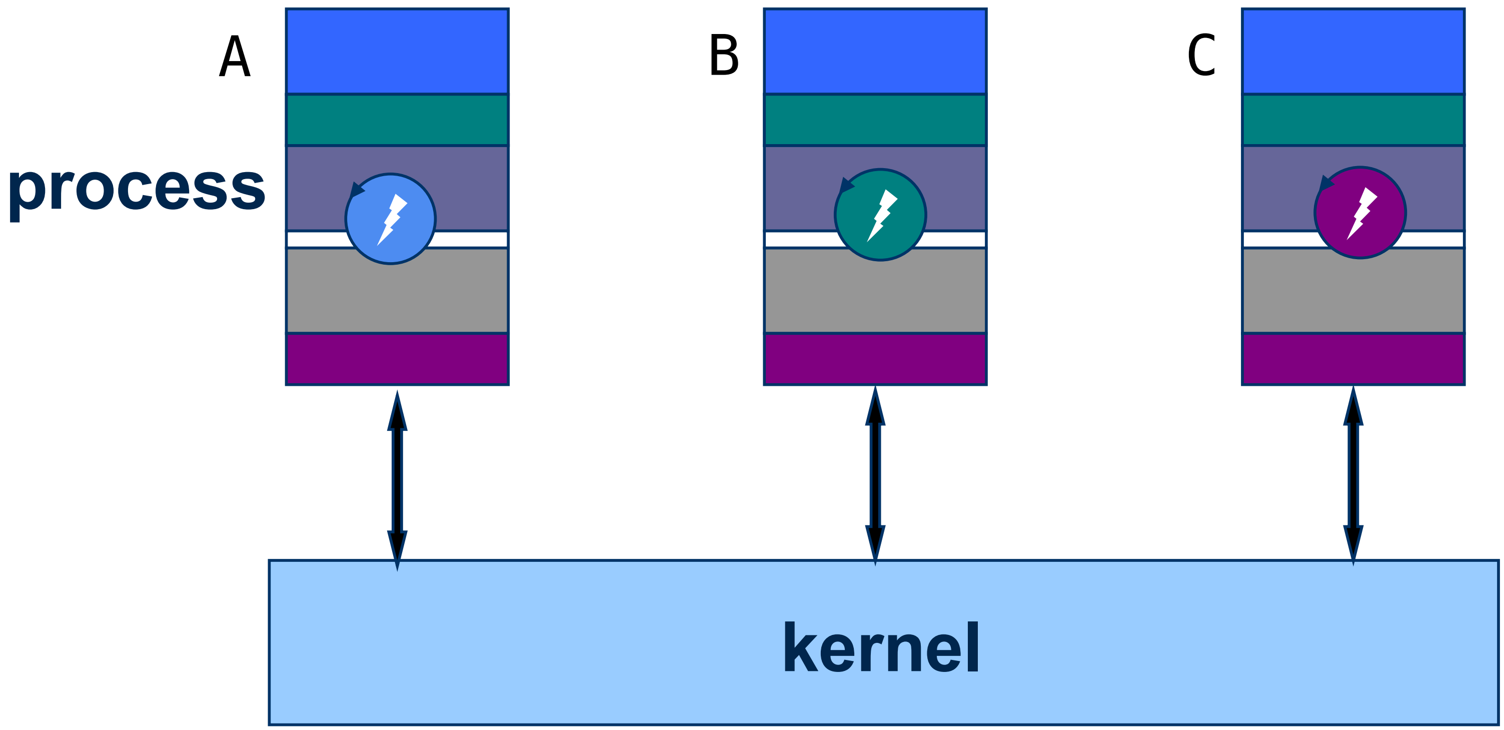
1. Compile and link.
2. Run 3x &.

```
chase$ cc –o democ democ.c
chase$ ./democ A & ./democ B & ./democ C
[1] 54922
[2] 54923
./democ C
./democ A
./democ B
./democ B
./democ A
./democ C
./democ A
./democ B
./democ C
[1]-  Done ./democ A
[2]+  Done ./democ B
chase$
```

democ.c

```c
int
main(int argc, char *argv[])
{
  int i, j;
  for (i=0; i<3; i++) {
    printf("%s %s\n",
              argv[0], argv[1]);
    for (j=0;
            j<500000000; j++);
  }
  return 0;
}
```

The story so far...



A **process** is a running **program** instance.

OS **kernel** multiplexes the computer among processes.

Kernel launches processes and provides services to them.

Virtualizing memory

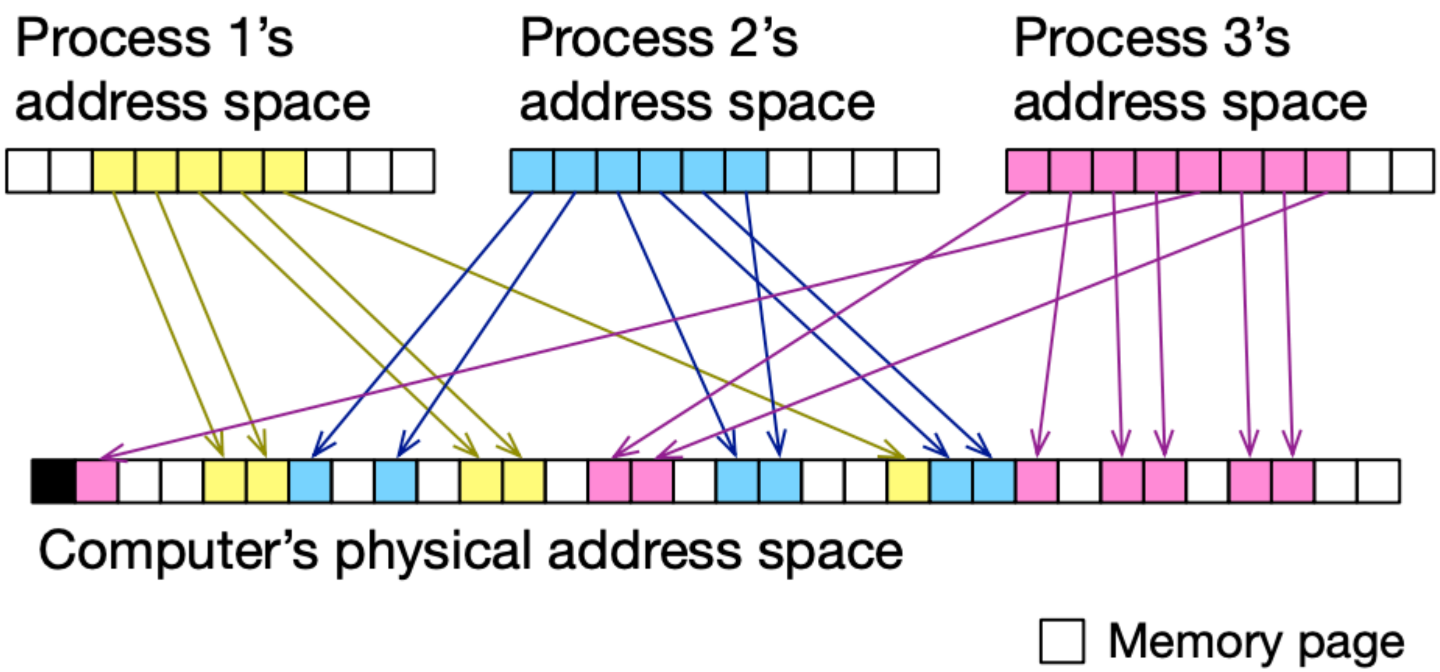


Figure 11: The virtual memory abstraction gives each process its own virtual address space. The operating system multiplexes the computer's DRAM between the processes, while application developers build software as if it owns the entire computer's memory.

Graphic from **Intel SGX Explained** (Costan and Devadas, IACR Cryptology, 2016)

A peek at virtual memory (VM)

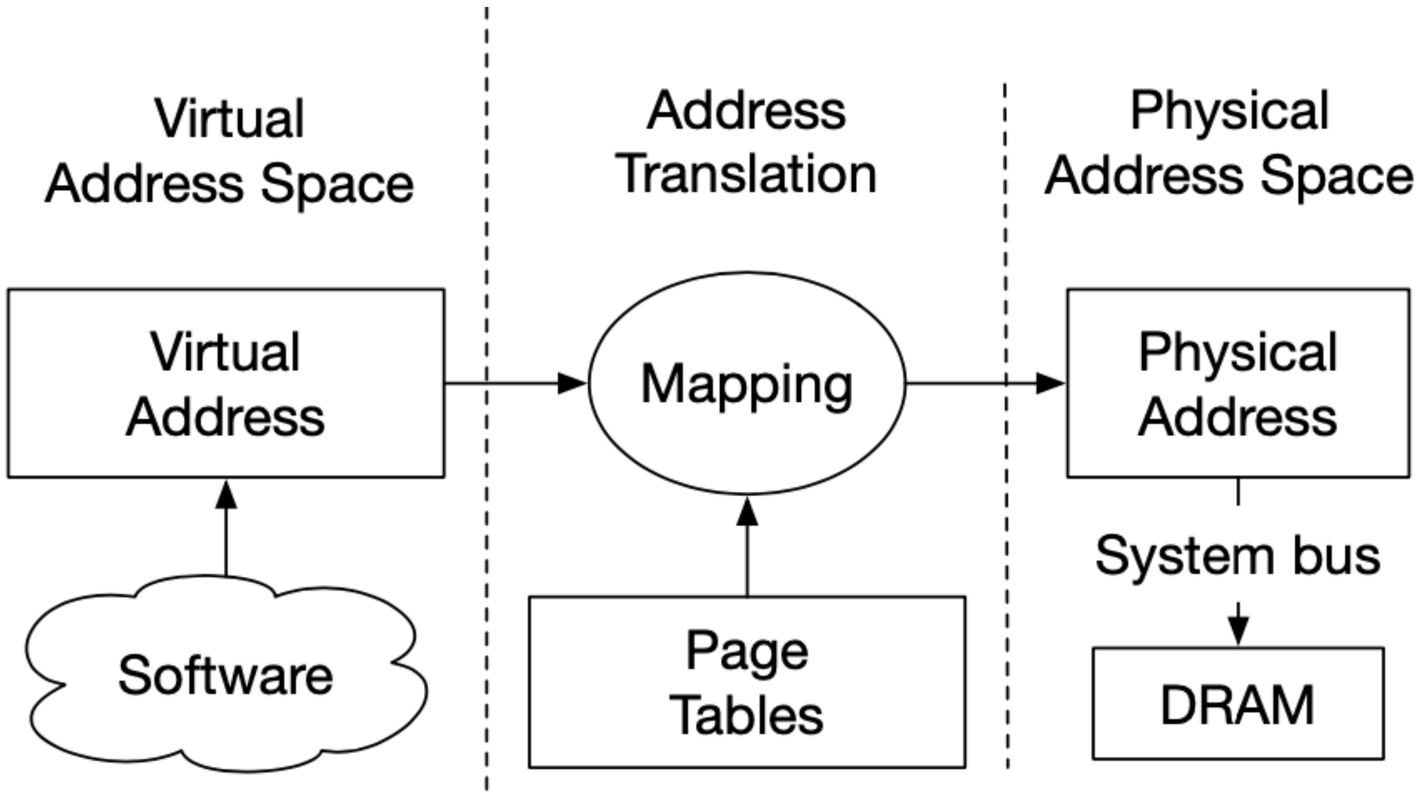
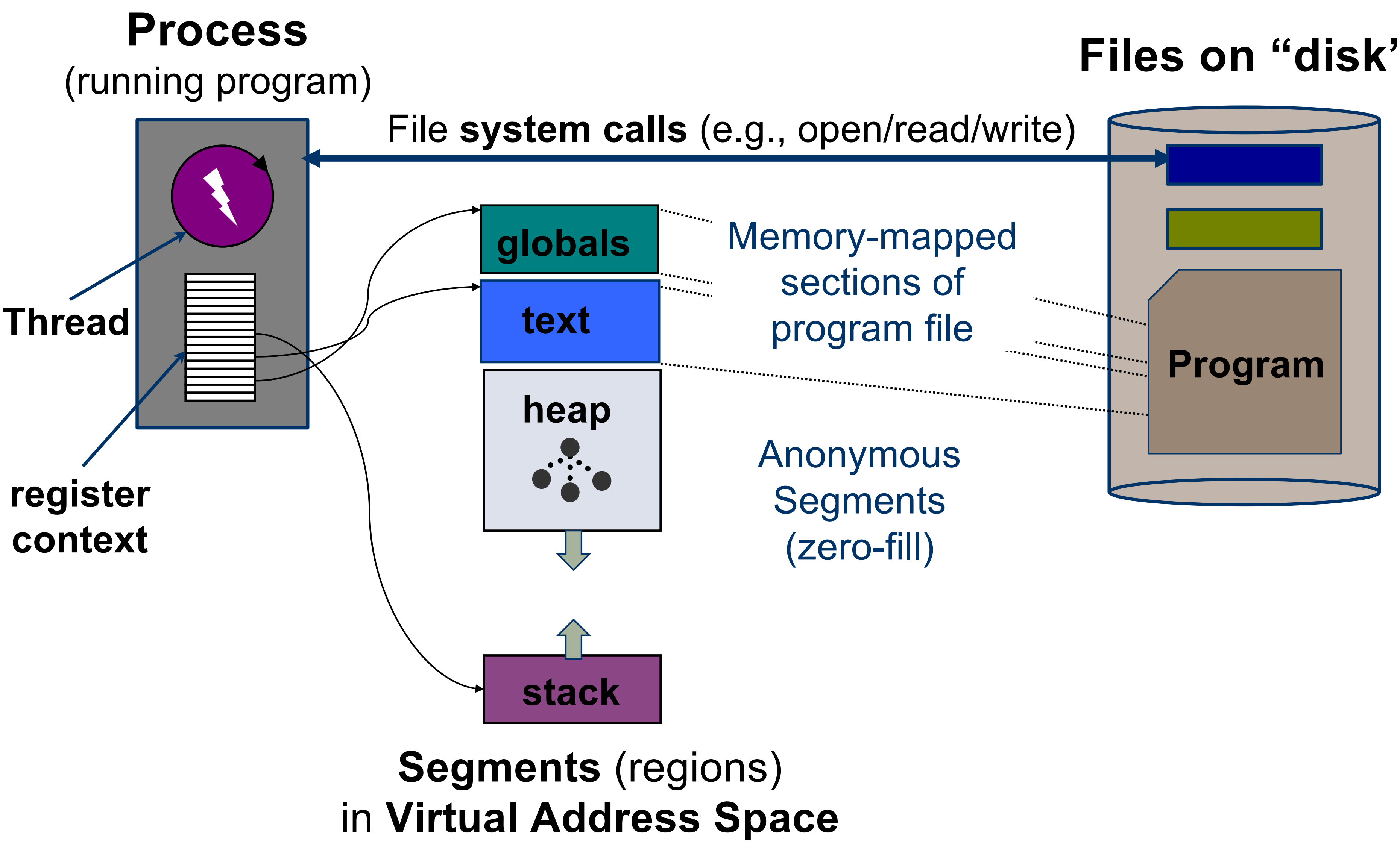


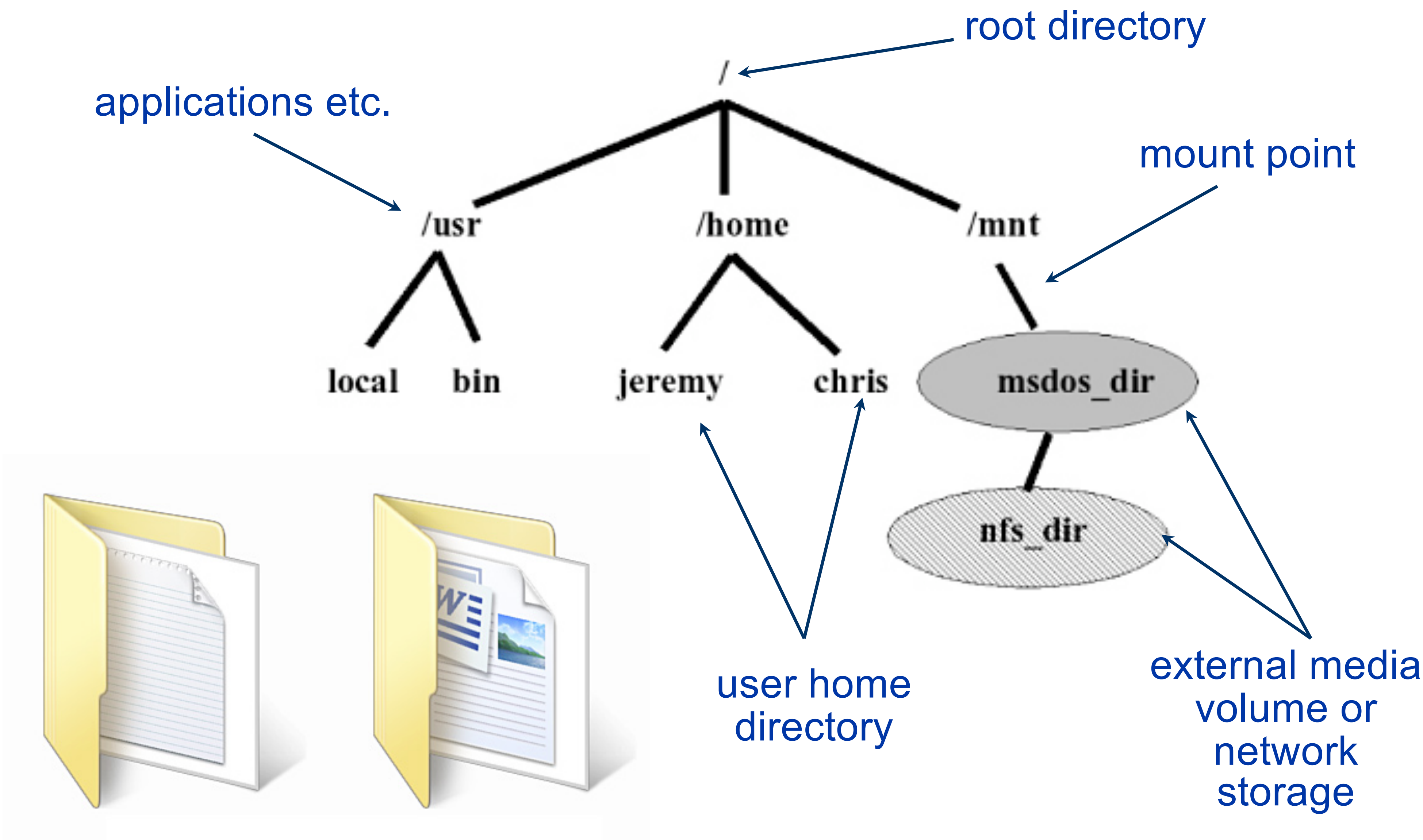
Figure 10: Virtual addresses used by software are translated into physical memory addresses using a mapping defined by the page tables.

Graphic from **Intel SGX Explained** (Costan and Devadas, IACR Cryptology, 2016)

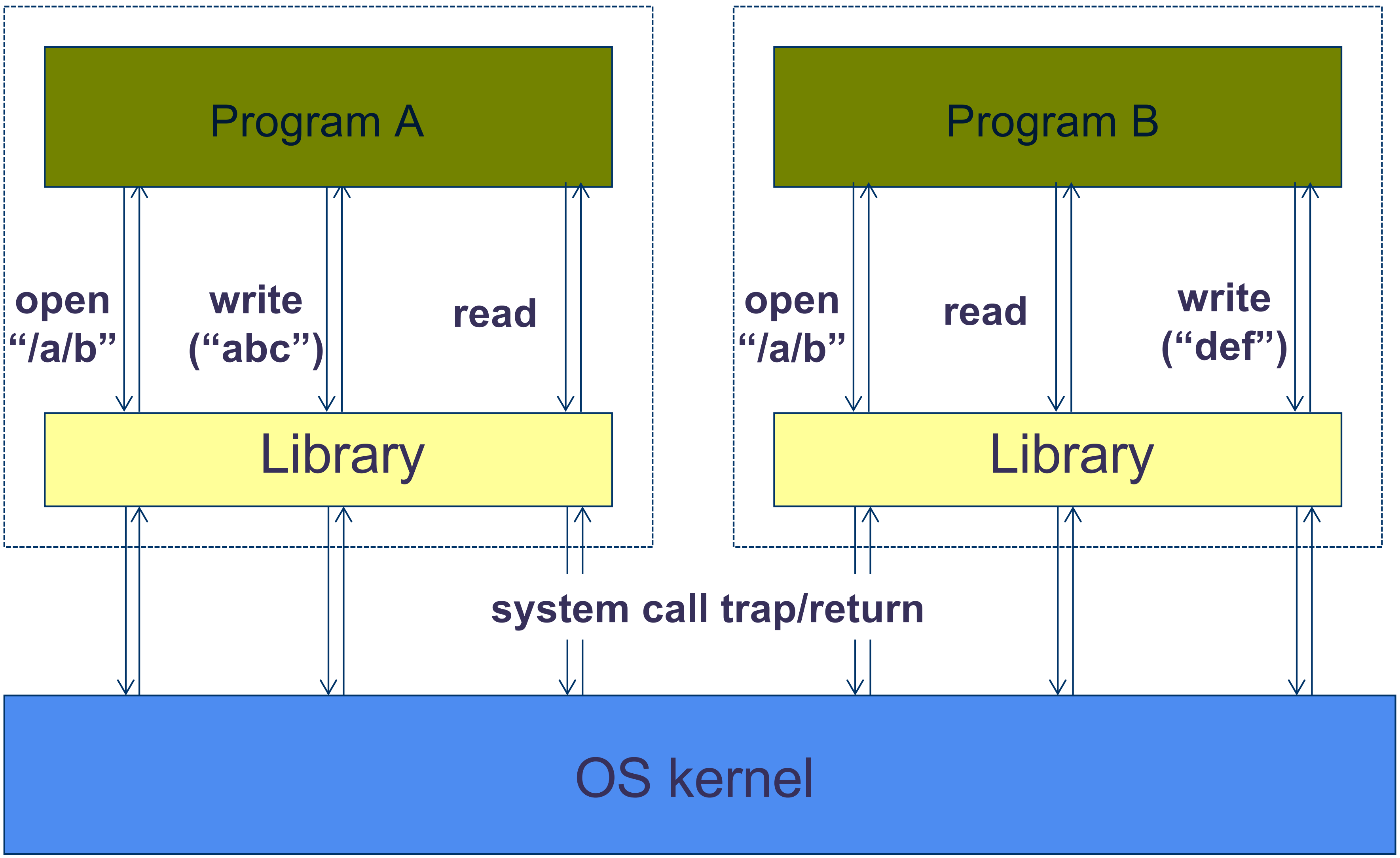
VM and files


Process
(running program)
Files on “disk”
File **system calls** (e.g., open/read/write)
Thread
register context
globals
text
heap
stack
Memory-mapped sections of program file
Anonymous Segments (zero-fill)
Program
Segments (regions) in **Virtual Address Space**

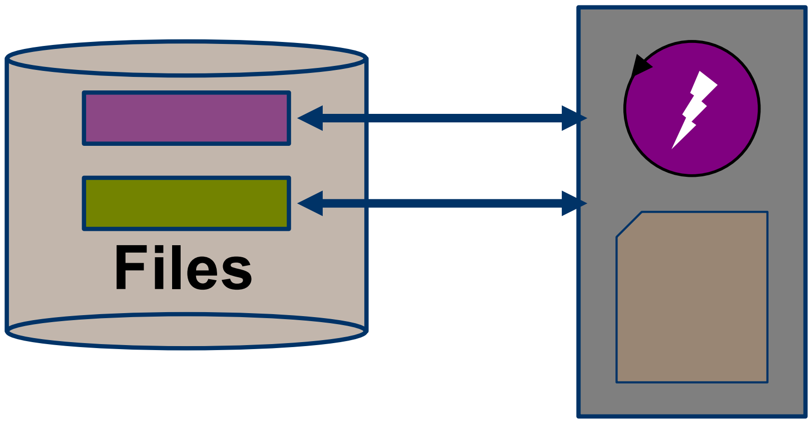
Files: hierarchical name space



The file abstraction



Files: open and chdir



fd = **open**(pathname, <options>);
write(fd, "abcdefg", 7);
read(fd, buf, 7);
lseek(fd, offset, SEEK_SET);
close(fd);

- Syscalls like **open** interpret a file **pathname** relative to the **current directory** of the calling process.
- Pathname starting with "/" is relative to the root directory.
- Change current directory with the **chdir** syscall.

"." in a pathname means current directory (curdir).
".." in a pathname means parent directory of curdir.

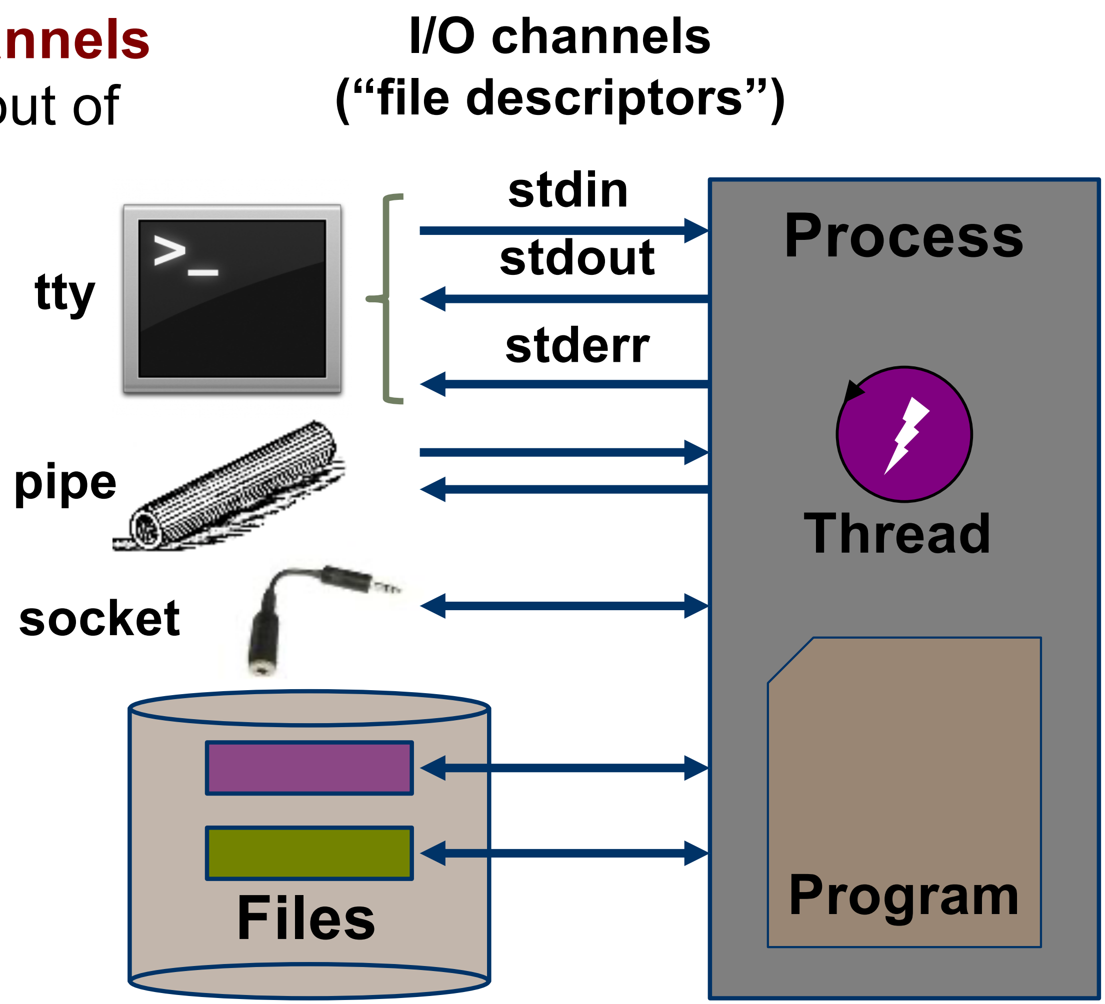
Unix process view: data

A process has multiple **channels** for data movement in and out of the process (I/O).

The channels are typed.

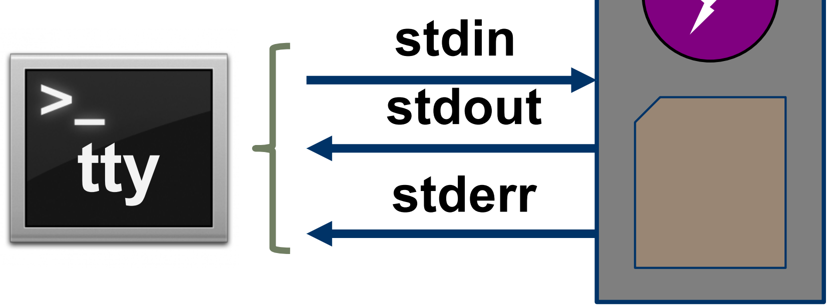
Each channel is named by an I/O **descriptor** (called a “file descriptor”).

A file descriptor is an integer value assigned by the kernel (e.g., at **open**).



Standard I/O descriptors

I/O channels
("file descriptors")



Open files or other I/O channels are named within the process by an integer **file descriptor** value.

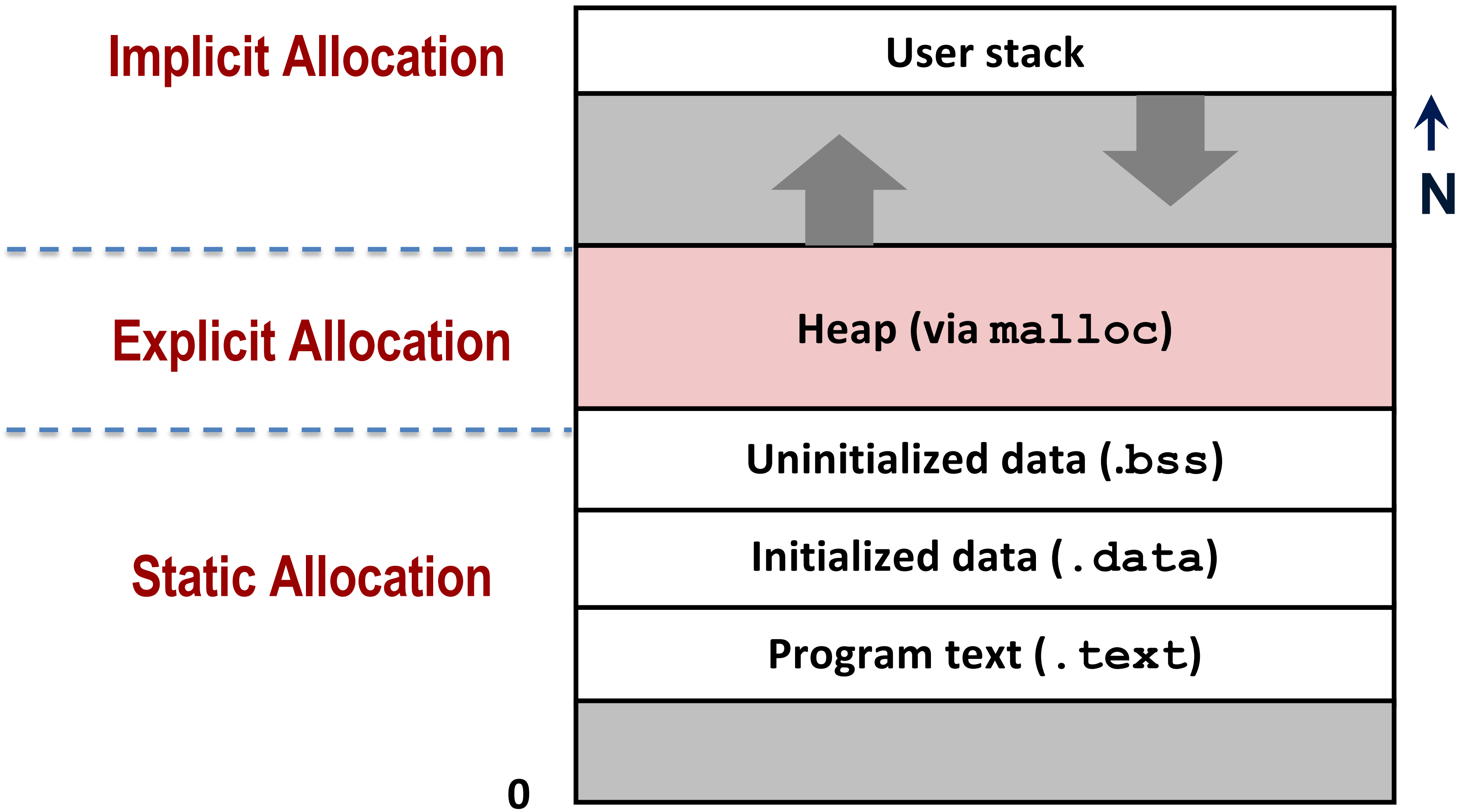
Standard descriptors for primary input (stdin=0), primary output (stdout=1), error/status (stderr=2).

By default they are bound to the **controlling tty**.

```
count = read(0, buf, count);
if (count == -1)  {
            perror("read failed");  /* writes to stderr */

            exit(1);
}
count = write(1, buf, count);
if (count == -1) {
            perror("write failed");  /* writes to stderr */
            exit(1);
}
```

Where do your variables live?



Command line arguments (C)

```
chase$ cc -o args args.c
chase$ ./args 1 2 3 4 5 6
arguments: 7
0: ./args
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
```

```c
#include <stdio.h>

int
main(int argc, char* argv[])
{
  int i;

  printf("arguments: %d\n", argc);
  for (i=0; i<argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }
}
```

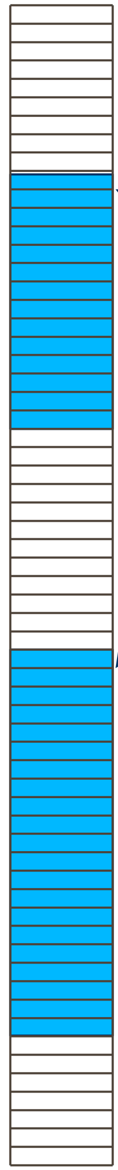
OS copies arguments into the VAS, passes count+address to main().

Heap: dynamic memory

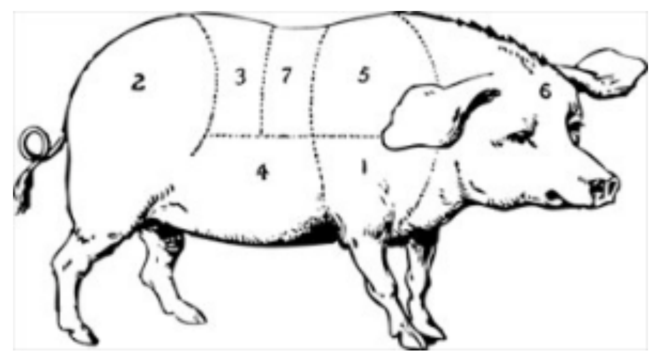
Heap segment. A contiguous chunk of memory obtained from OS kernel. E.g., with Unix *sbrk*() syscall, or *mmap*()

A **runtime library** obtains the block and manages it as a "heap" for use by the programming language environment, to store dynamic objects.

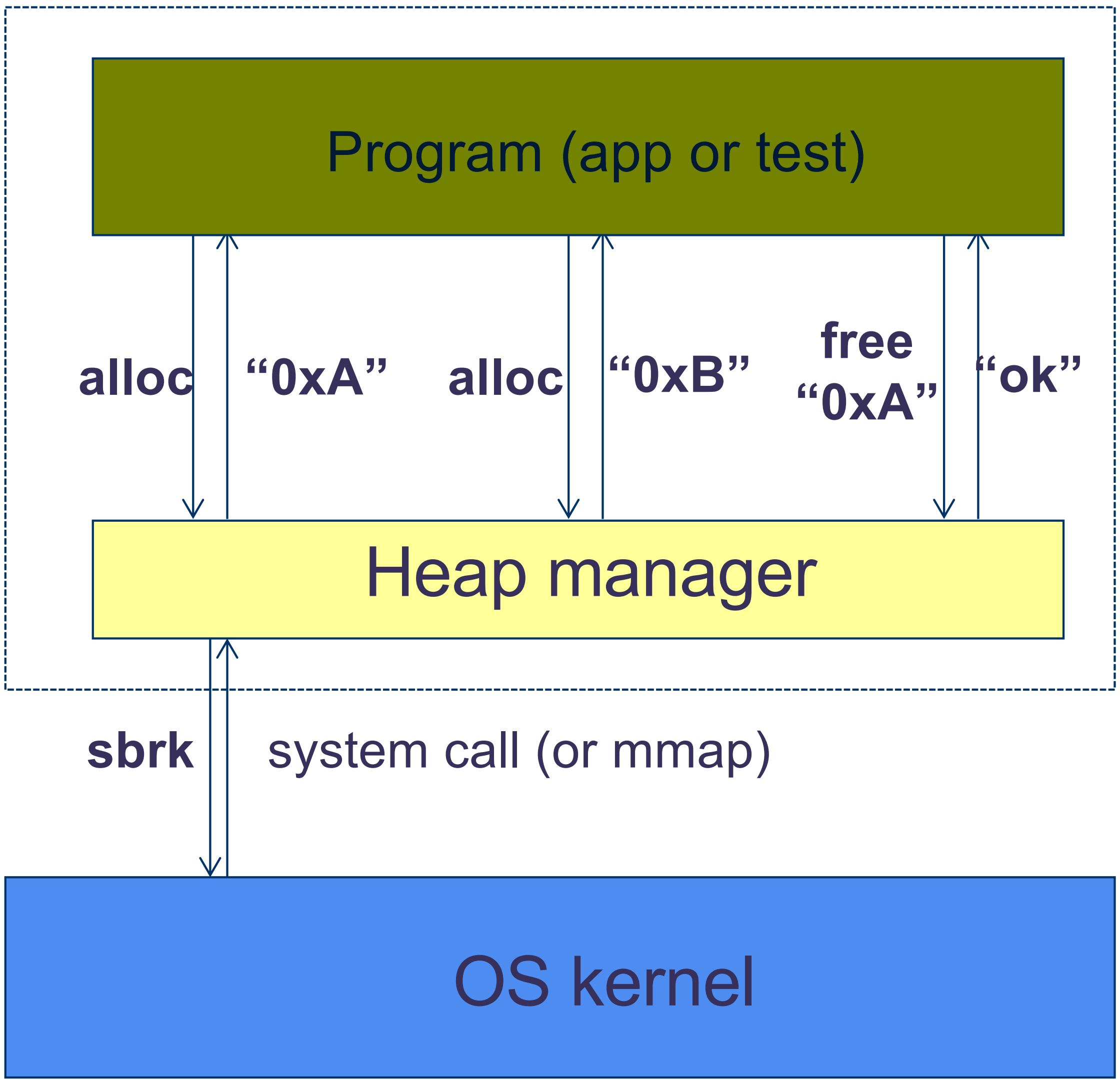
E.g., with Unix *malloc* and *free* library calls, or new in Java or C++.

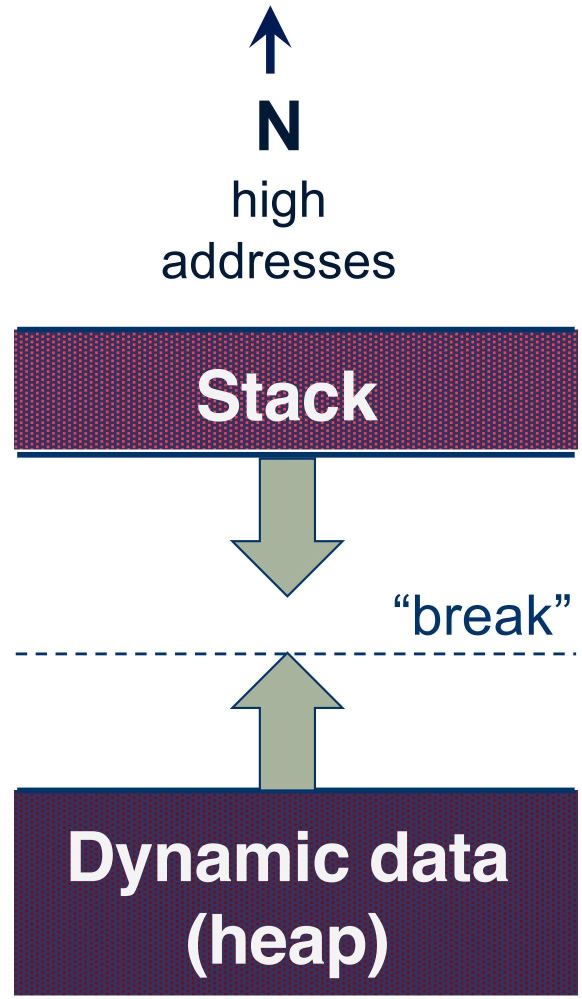


Allocated heap blocks for structs or objects. Align!



Heap manager



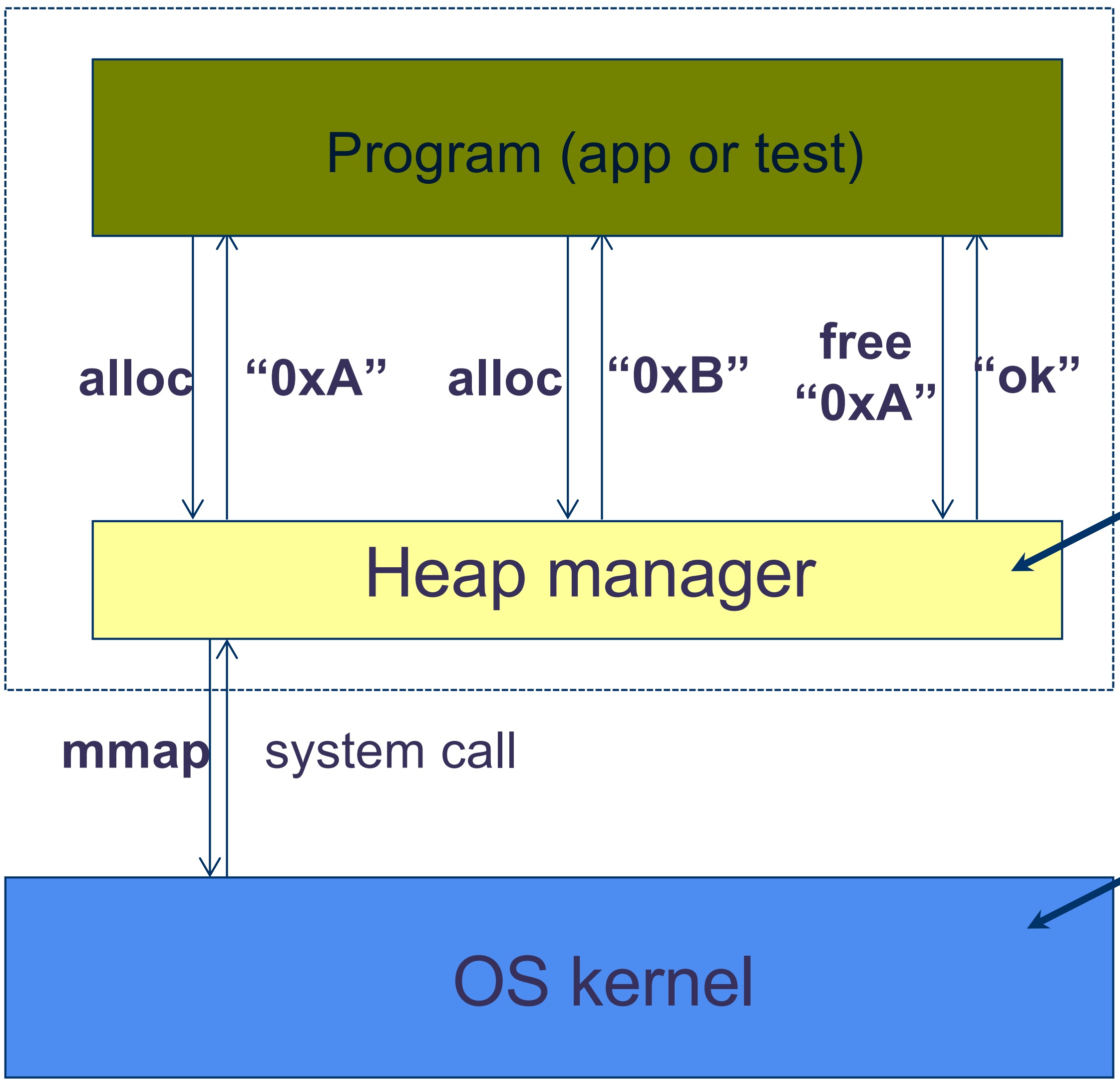


sbrk(X) increments program's data/heap segment size by X bytes

The heap manager contract

1. User program calls **malloc** API to **allocate** a block of any desired size to store some dynamic data.
2. Heap manager returns a pointer to a block. The program uses that block for its purpose. The block's memory is reserved exclusively for that use.
3. Program calls heap manager to **free** (**deallocate**) the block when the program is done with it.
4. Once the program frees the block, the heap manager may reuse the memory in the block for another purpose.
5. User program is responsible for initializing the block, and deciding what to store in it. Initial contents could be old. Program must not try to use the block after freeing it.

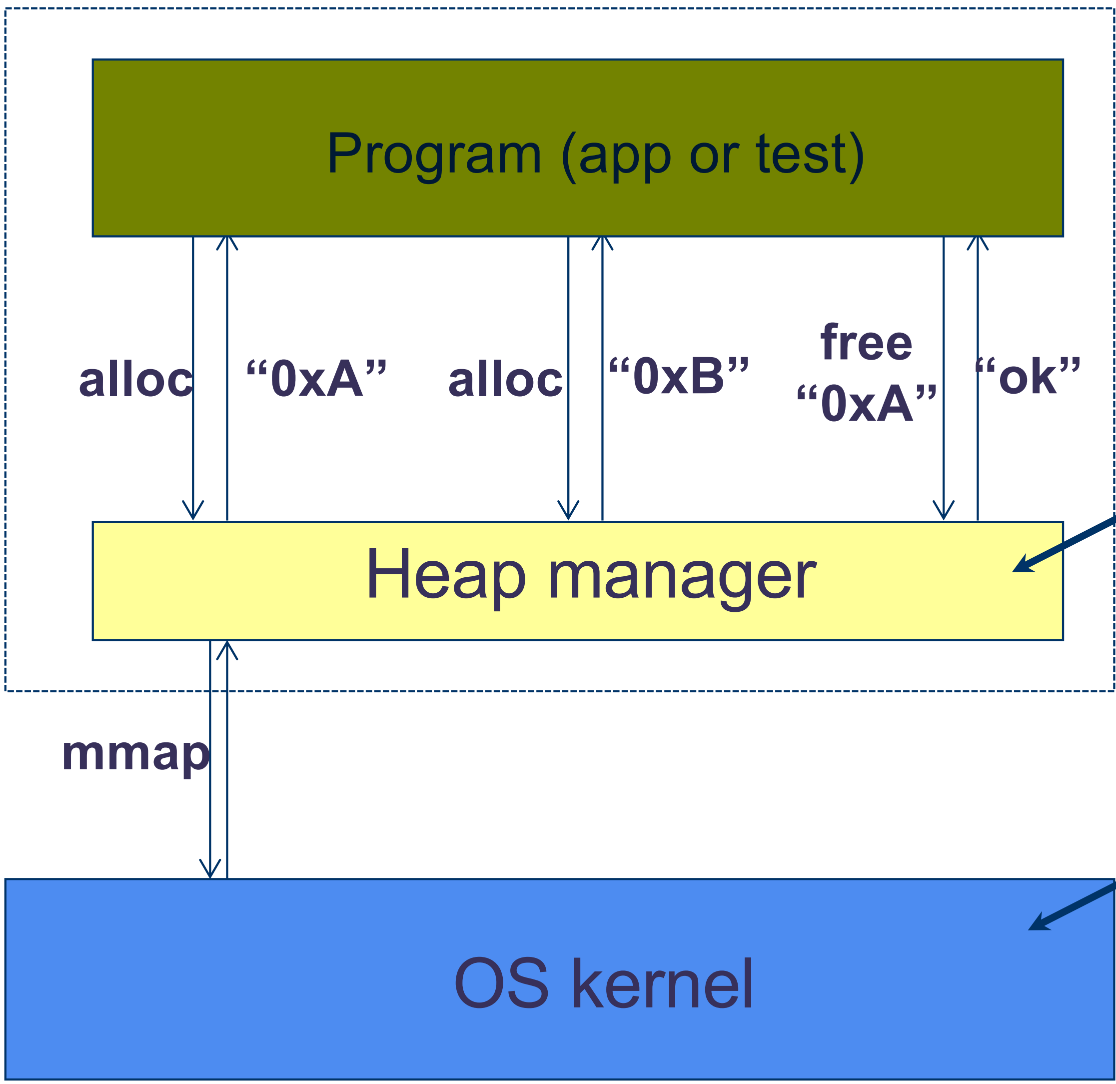
Memory management: two levels



Allocate and free heap blocks of varying sizes, with arbitrary lifetimes.

Allocate and free VM regions of varying sizes, with arbitrary lifetimes.

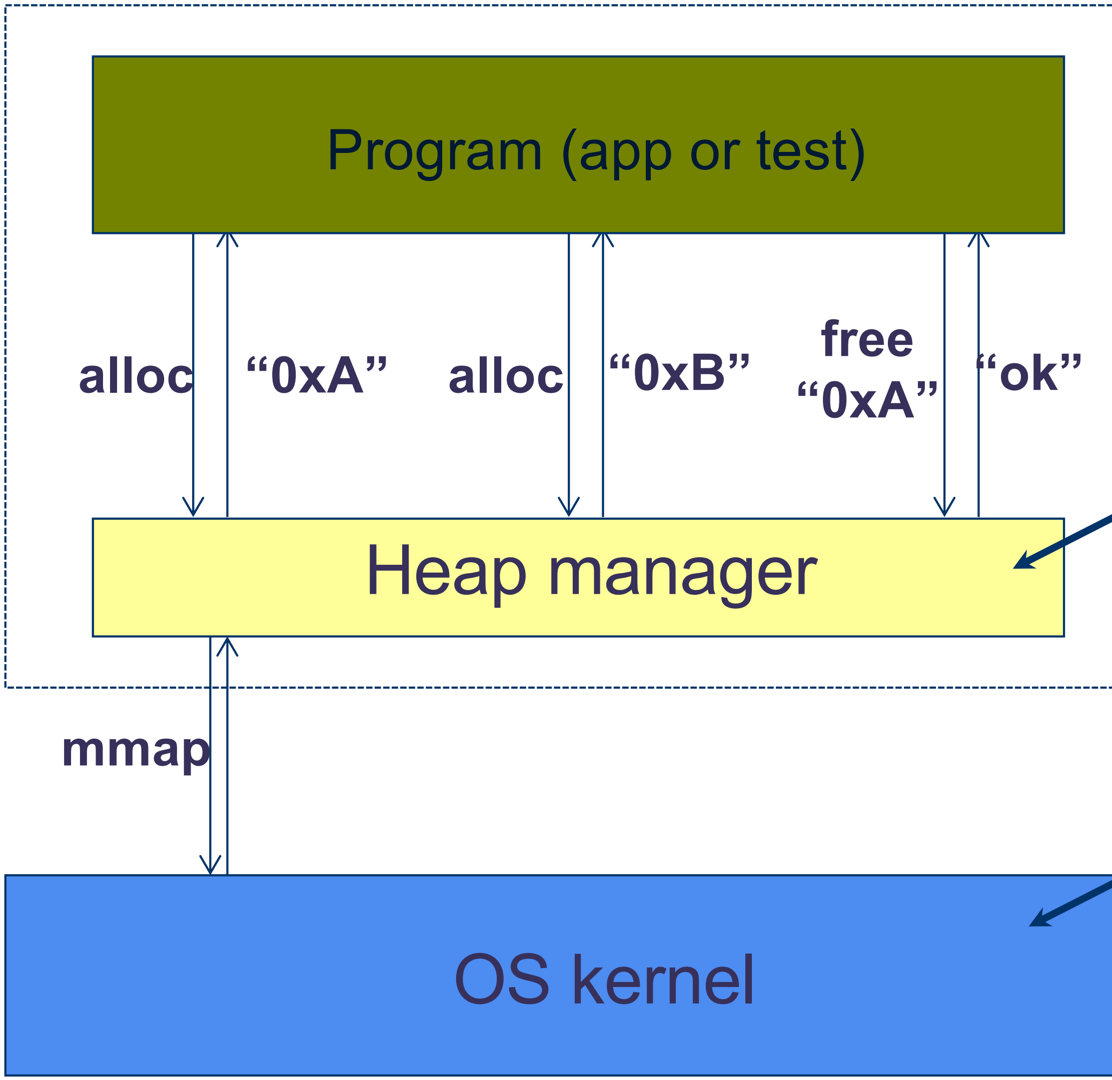
Memory management: user/kernel



User code: works for the user and program.

Kernel code: works for the system and all of its users and programs.

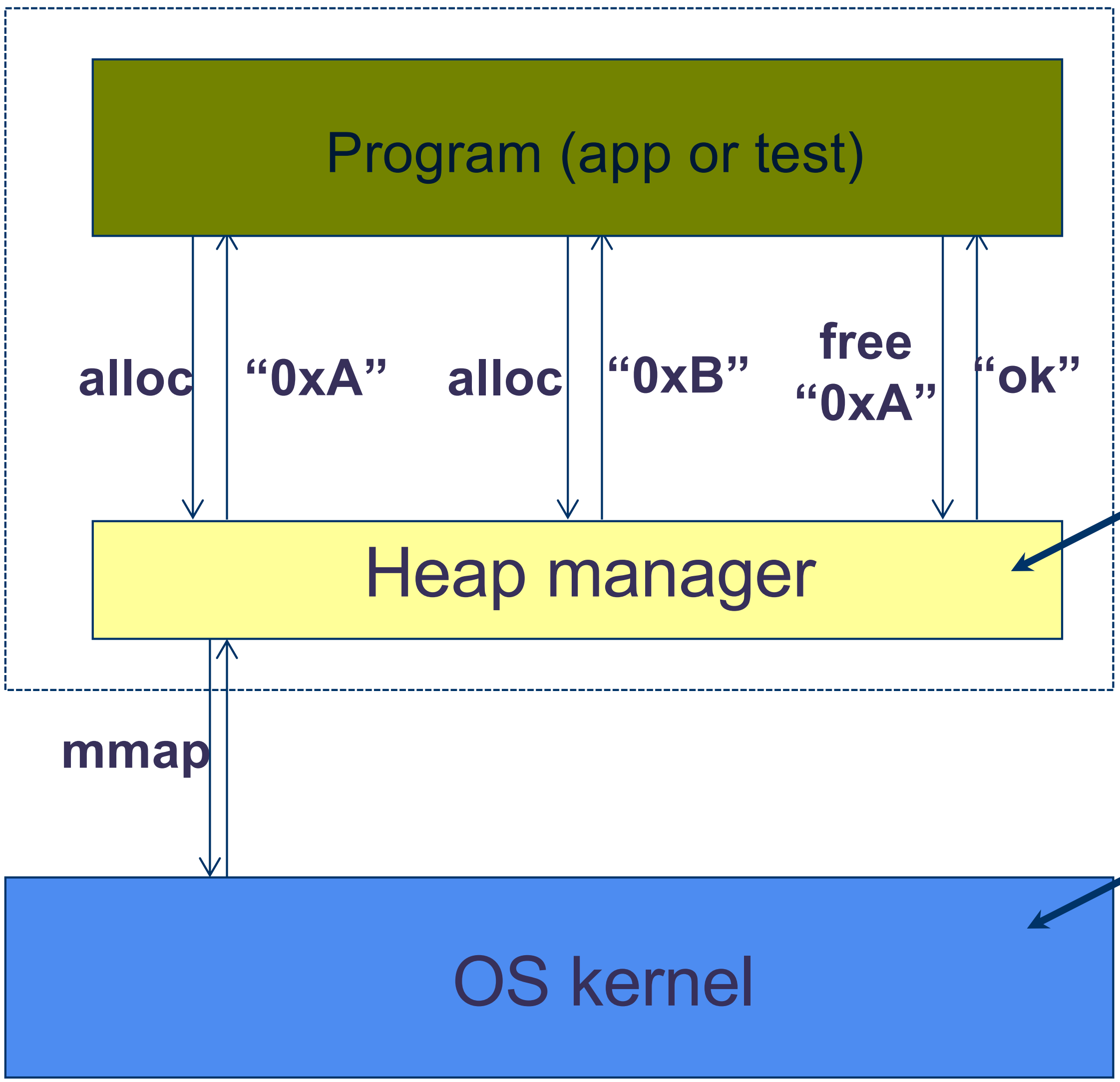
Memory management: trust



The program trusts it.
It trusts the program.

Kernel code: trusted by everyone. **Trusts no-one.**

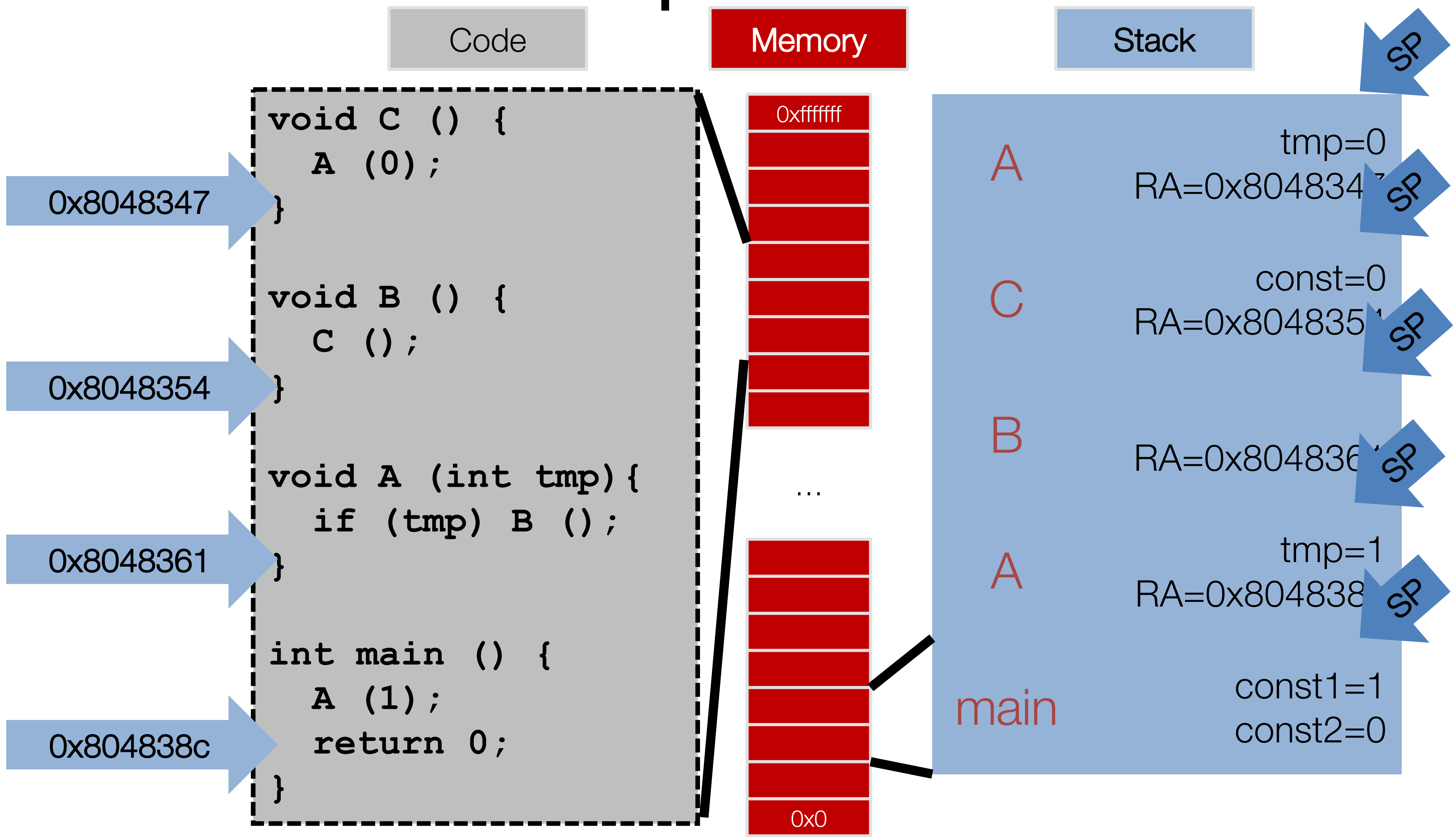
Memory management: isolation



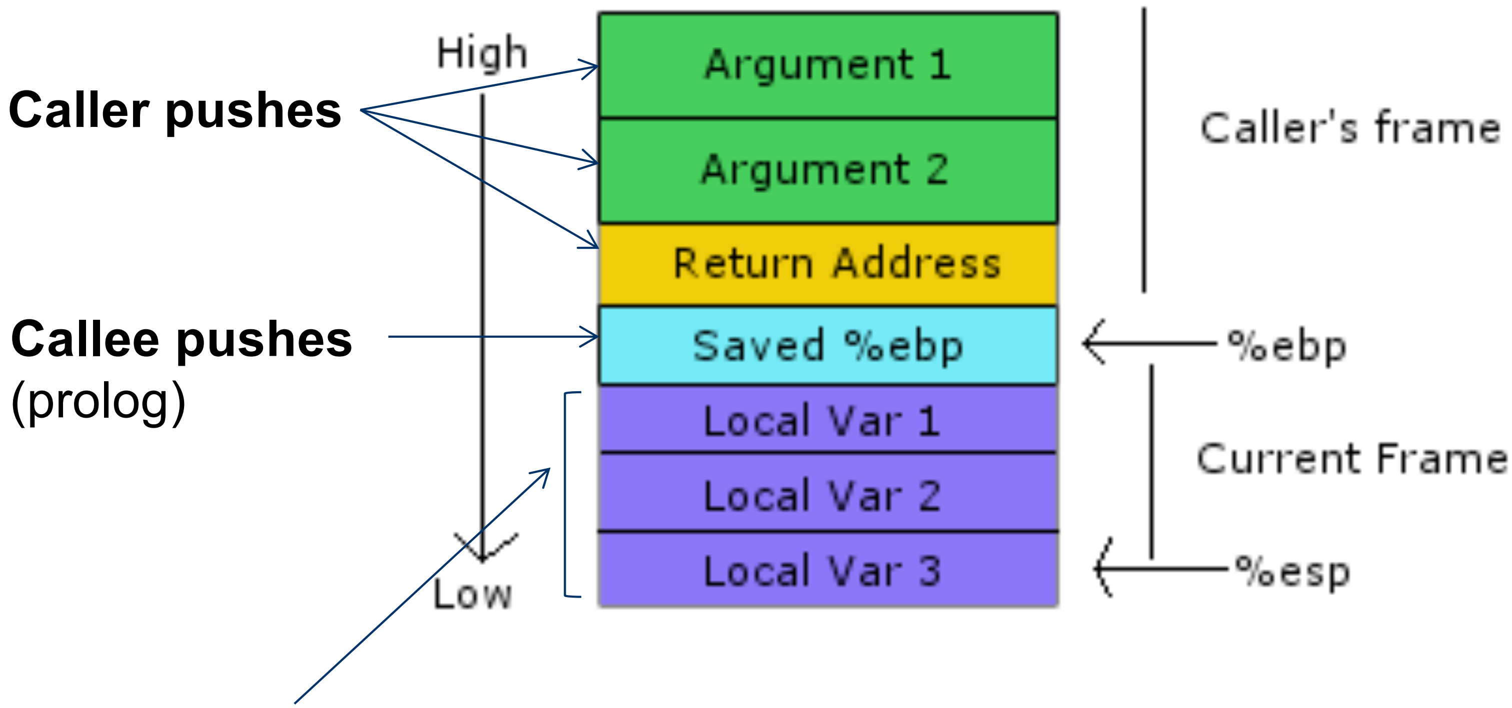
Leaves memory uninitialized.

Never exposes data across process boundaries: zero-fill to "clean" empty memory.

Example stack



IA32 stack discipline



Fixed-size stack frame data area: size determined by compiler of the called procedure. The procedure's compiled code addresses locals and arguments at fixed offsets from %ebp (frame pointer).

Note: callee code may push some callee-saved register values in here too.

Stack layout when calling functions

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
}
```

old base pointer

`low`

return address

`high(N)`

local variables pushed in the same order as they appear in the code

arguments pushed in reverse order of code

Example: C structs, global, stack, heap

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

struct stuff {
  int i;
  long j;
  char c[2];
};
```

Data structure type definition

```
struct stuff gstuff;
```

A global structure

Local variables

Heap allocation

```
int main() {
  struct stuff sstuff;
  struct stuff *hstuffp =
      (struct stuff *) malloc(sizeof(struct stuff));

  gstuff.i = 13;
  gstuff.j = 14;
  gstuff.c[0] = 'z';
  gstuff.c[1] = '\0';
```

Accessing a global structure

```
  printf("%s\n", gstuff.c);

  sstuff.i = 13;
  sstuff.j = 14;…
```

Local data of a procedure (on the stack)

```
  hstuffp->i = 13;
  hstuffp->j = 14;…
}
```

Accessing a heap-allocated struct through a pointer.

```
cc –o structs structs.c
otool –vt structs
```

On MacOS, the otool –vt command shows contents of the text section of an executable file. Here are the instructions for the **structs** program. When the program runs, the OS loads them into a contiguous block of virtual memory (the text segment) at the listed virtual addresses.

```
0000000100000ea0 pushq   %rbp
0000000100000ea1 movq    %rsp, %rbp
0000000100000ea4 subq    $48, %rsp
0000000100000ea8 movabsq $24, %rdi
0000000100000eb2 callq   0x100000f42
0000000100000eb7 leaq    182(%rip), %rdi
0000000100000ebe leaq    347(%rip), %rcx
0000000100000ec5 movq    %rcx, %rdx
0000000100000ec8 addq     $16, %rdx
0000000100000ecf movq    %rax, -32(%rbp)
0000000100000ed3 movl    $13, (%rcx)
0000000100000ed9 movq    $14, 8(%rcx)
0000000100000ee1 movb    $122, 16(%rcx)
0000000100000ee5 movb    $0, 17(%rcx)
0000000100000ee9 movq    %rdx, %rsi
0000000100000eec movb    $0, %al
0000000100000eee callq   0x100000f48
0000000100000ef3 movl    $0, %r8d
```

```
0000000100000ef9 movl    $13, -24(%rbp)
0000000100000f00 movq    $14, -16(%rbp)
0000000100000f08 movb    $122, -8(%rbp)
0000000100000f0c movb    $0, -7(%rbp)
0000000100000f10 movq    -32(%rbp), %rcx
0000000100000f14 movl    $13, (%rcx)
0000000100000f1a movq    -32(%rbp), %rcx
0000000100000f1e movq    $14, 8(%rcx)
0000000100000f26 movq    -32(%rbp), %rcx
0000000100000f2a movb    $122, 16(%rcx)
0000000100000f2e movq    -32(%rbp), %rcx
0000000100000f32 movb    $0, 17(%rcx)
0000000100000f36 movl    %eax, -36(%rbp)
0000000100000f39 movl    %r8d, %eax
0000000100000f3c addq    $48, %rsp
0000000100000f40 popq    %rbp
0000000100000f41 ret
```

Simplified…

```
int main() {
  struct stuff sstuff;

  struct stuff *hstuffp =
      (struct stuff *)
      malloc(24);

  gstuff.i = 13;
  gstuff.j = 14;
  gstuff.c[0] = 'z';
  gstuff.c[1] = '\0';

  sstuff.i = 13;
  sstuff.j = 14;…

  hstuffp->i = 13;
  hstuffp->j = 14;…
}
```

```
pushq    %rbp
movq     %rsp, %rbp
subq     $48, %rsp
```

Push a frame on the stack: this one is 48 bytes. **Who decided that?**

```
movabsq $24, %rdi
callq    0x100000f42
movq     %rax, -32(%rbp)
```

Call **malloc**(24): move return value into a local variable (at an offset from the stack base pointer).

```
leaq     347(%rip), %rcx
```

Address global data relative to the code address (%rip=PC). **position-independent**

...address stack data relative to (%rbp)

; move pointer to heap block into %rcx

```
movq     -32(%rbp), %rcx
```

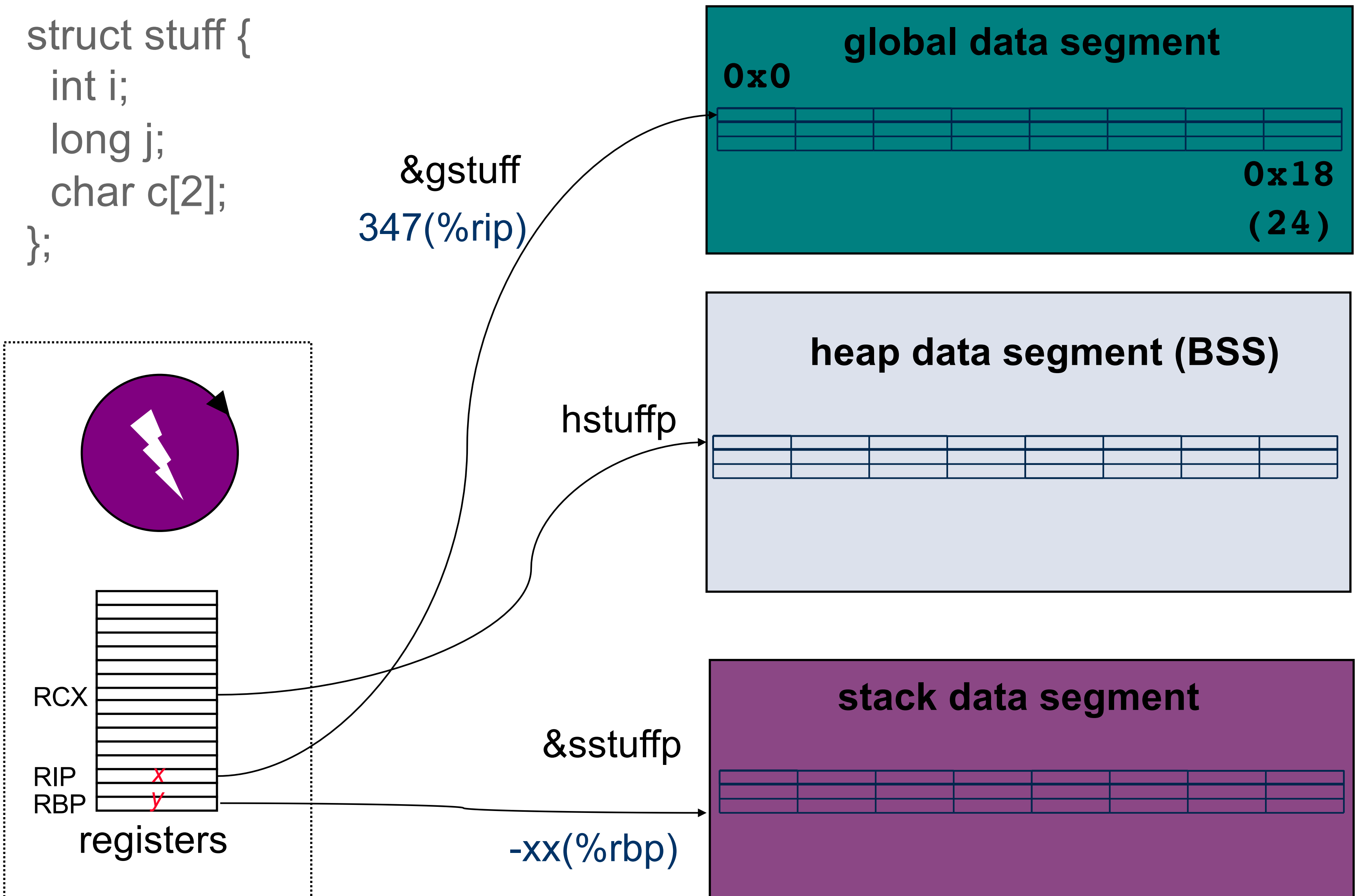
...address heap block relative to (%rcx)

```
addq     $48, %rsp
popq     %rbp
ret
```

Pop procedure frame from the stack before returning from main().

Addressing stuff

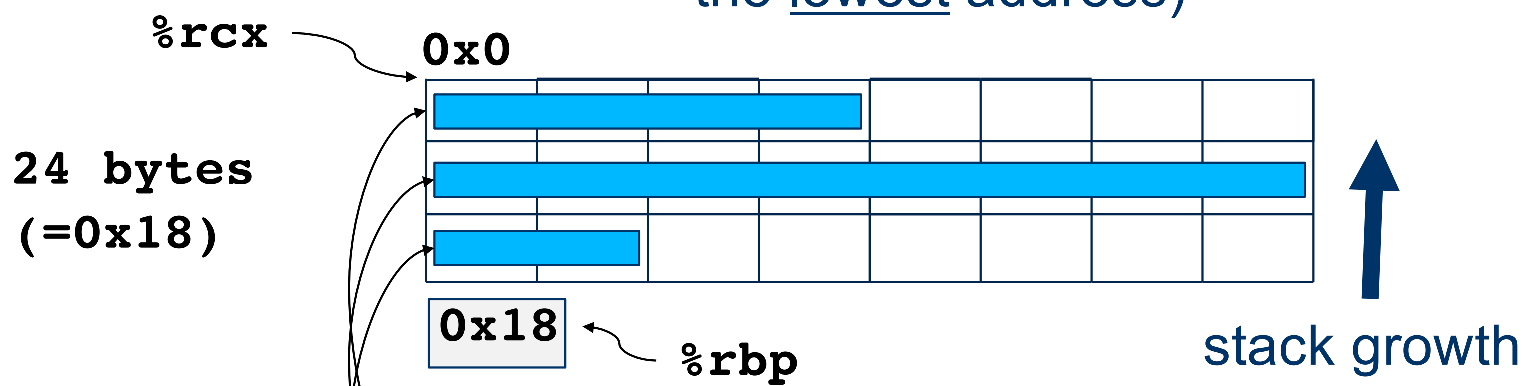
```
struct stuff {
  int i;
  long j;
  char c[2];
};
```



Addressing stuff

struct members stored in memory in order they are declared (hence i is at the <u>lowest</u> address)

```
struct stuff {
  int i;
  long j;
  char c[2];
};
```



sstuff.i = 13;	movl $13, -24(%rbp)
sstuff.j = 14;	movq $14, -16(%rbp)
sstuff.c[0] = 'z';	movb $122, -8(%rbp)
sstuff.c[1] = '\0';	movb $0, -7(%rbp)

hstuffp->i = 13;	movl $13, (%rcx)
hstuffp->j = 14;	movq $14, 8(%rcx)
hstuffp->c[0] = 'z';	movb $122, 16(%rcx)
hstuffp->c[1] = '\0';	movb $0, 17(%rcx)

Better than sbrk

pages may be written

pages may be read

mapping is not backed by a file

```
int prot = (PROT_WRITE | PROT_READ);
int flags = (MAP_SHARED | MAP_ANONYMOUS);
freelist = (metadata_t*)
        mmap(NULL, max_bytes, prot, flags, -1, 0);
```

starting address (here chosen by kernel)

used for mapping a file to memory (here no file)

```
freelist = (metadata_t*)
        sbrk(max_bytes);
```

mmap is a general "swiss army knife" system call to create a new region/segment in the virtual address space, and make it valid for reference.