Unit III : Address Spaces

- Early Systems, Multiprogramming and Time Sharing, The Address Space,

- Memory API**:** Types of Memory, The malloc() Call, The free() Call, Common Errors, Underlying OS Support,

- Segmentation, Fine-grained vs. Coarse-grained Segmentation, Free-Space Management**,**

- Paging, A Memory Trace, Faster Translations, (TLBs), TLB Basic Algorithm, Example: Accessing An Array, Who Handles The TLB Miss?, TLB Issue: Context Switches, Replacement Policy,

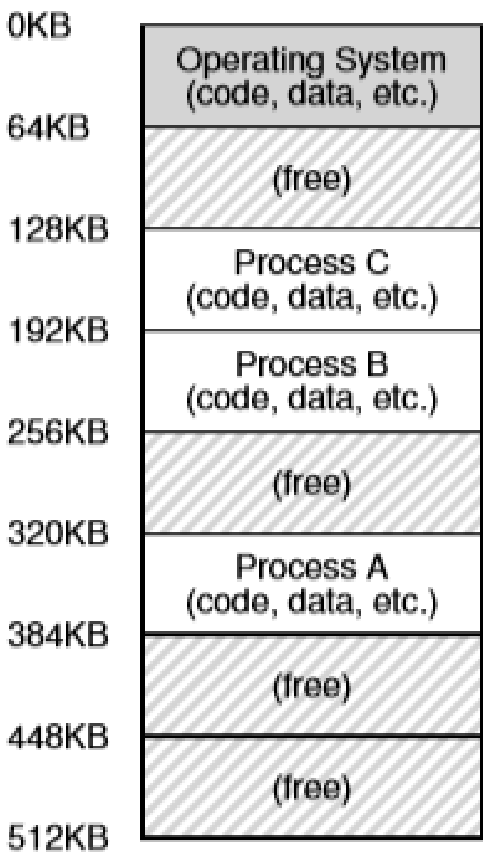
- Hybrid Approach: Paging and Segments, Beyond Physical Memory: Mechanisms**,** Swap Space, The Present Bit, The Page Fault, What If Memory Is Full?, Page Fault Control Flow, When Replacements Really Occur, The Linux Virtual Memory System.

Why Virtualize Memory ?

- Because real view of memory is messy.
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU.
  - Many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user

Multiprogramming and Time Sharing

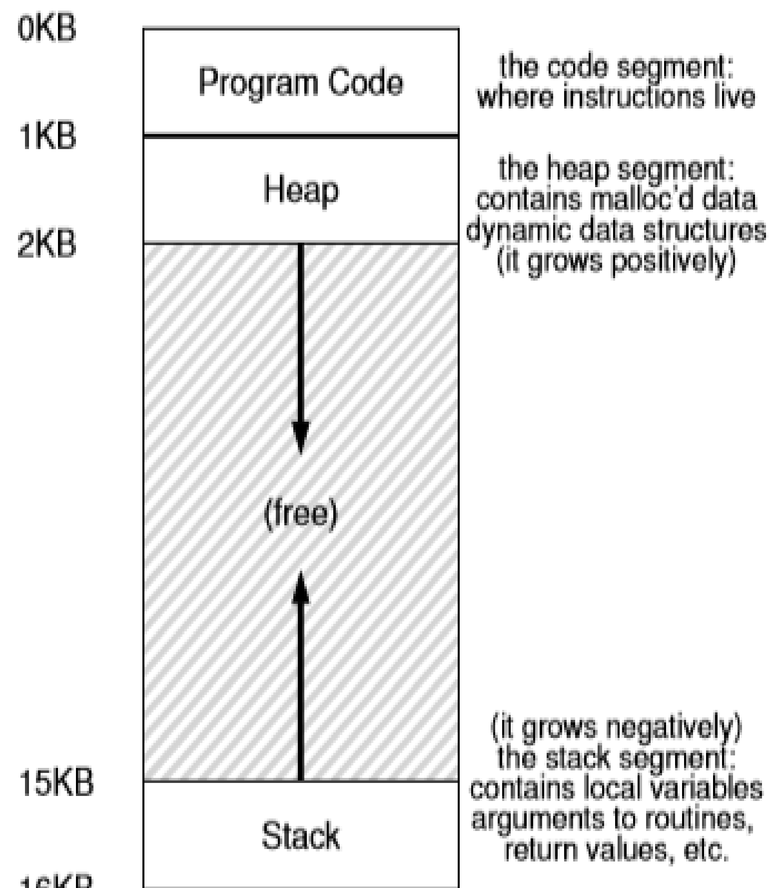
- **Load multiple processes** in memory.
  - Execute one for a short while.
  - Switch processes between them in memory.
  - Increase utilization and efficiency.



Three Processes: Sharing Memory

Abstraction of Virtual Address Space

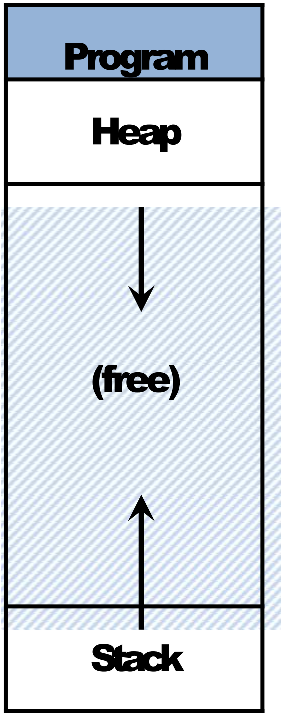
- OS creates an **abstraction** of physical memory, called as Address Space.
- Virtual Address Space: Every process assume it has access to large space of memory from address 0 to MAX
  - The address space contains all about a running process.
  - That is consist of program code, heap, stack etc.
- CPU issues load and store to virtual addresses



An Example Address Space

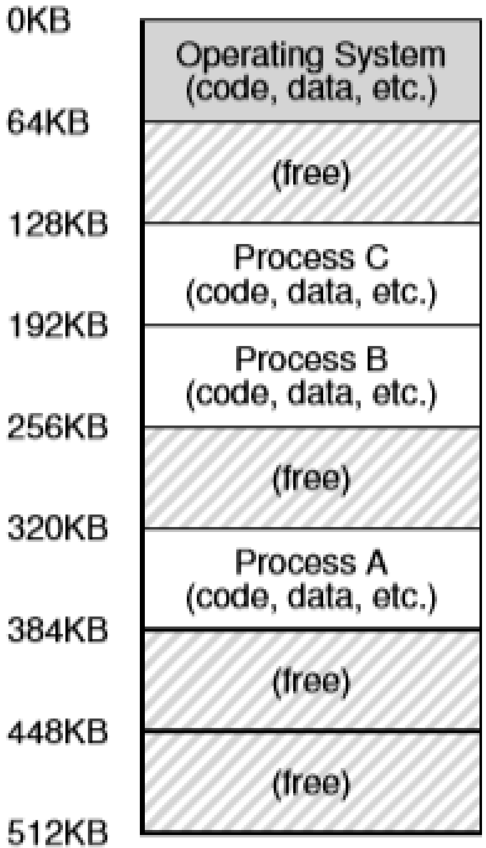
Address Space(Cont.)

- **Program**
  - **Where instructions and static data live**
- **Heap**
  - **Dynamically allocate memory.**
    - `malloc/free` **in C language**
    - `new/delete` **in object-oriented language**
- **Stack**
  - **Store return addresses or values.**
  - **Contain local variables arguments to routines.**



Address Space

- Observe how processes A, B and C are loaded into memory
- When the OS does this, we say the OS is virtualizing memory, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space (say 32-bits or 64-bits); the reality is quite different.



Three Processes: Sharing Memory

Goals of VM system

- Transparency:
  - The program shouldn't be aware of the fact that memory is virtualized.

- Efficiency:
  - The OS should strive to make the virtualization as efficient as possible, both in terms of time and space.
  - In implementing time-efficient virtualization, the OS will have to rely on hardware support, including hardware features such as TLBs

- Protection:
  - Protection enables us to deliver the property of isolation among processes.

Contiguous Memory

- Total size of memory is 10 KB
  - P1 requires 2 KB.
  - P2 requires 5 KB

(4KB)	P1 (2KB)	(4KB)

- We can not store P2 in contiguous location,
- **Space is available but can no be allotted, Called as External Fragmentation.**
- **Advantages: Address translation and access is easy (only need to remember base address)**
- **Disadvantage: External Fragmentation**

Non-Contiguous Allocation

- Total size of memory is 10 KB
  - P1 requires 2 KB.
  - P2 requires 5 KB (provides link)

(4KB)	**P1 (2KB)**	**(4KB)**

We can not access the data element directly

Advantages: No external fragmentation.

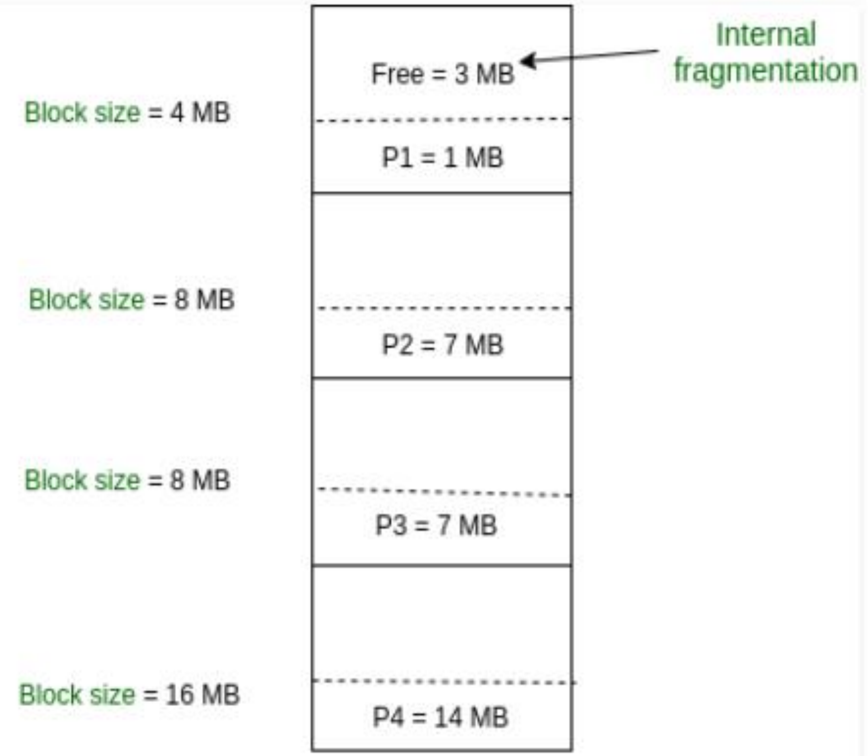
Contiguous : Fixed Partitioning

- Contiguous Technique can be divided into:
  - Fixed (or static) partitioning
  - Variable (or dynamic) partitioning

- Fixed Size Partitioning:

- This is the oldest and simplest technique used to put more than one processes in the main memory.

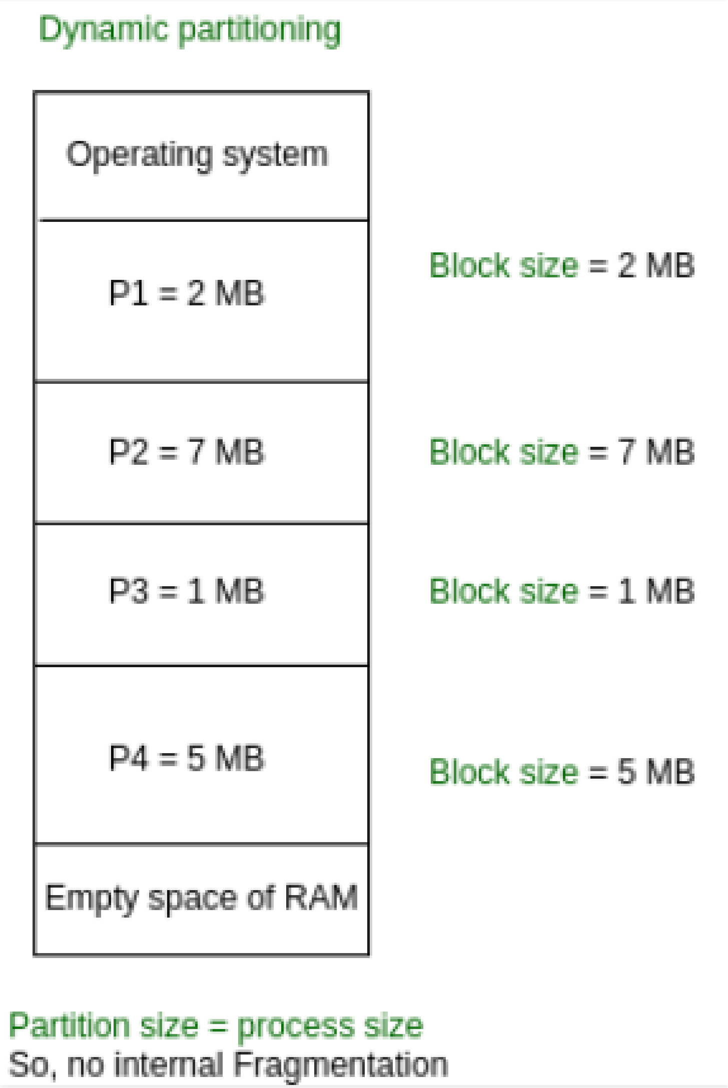
- In this partitioning, number of partitions (non-overlapping) in RAM are **fixed but size** of each partition may or **may not be same**



Fixed size partition

Contiguous : Variable-sized Partitioning

- The size of partition will be equal to incoming process.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.
- No internal fragmentation but external fragmetantion.
  - For example, suppose in above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB.
  - Let's suppose process P5 of size 3MB comes



- For both the schemes(fixed and variable size partitioning) , the operating system must keep list of each memory location noting which are free and which are busy.

- Then as new jobs come into the system, the free partitions must be allocated.

- These partitions may be allocated by 4 ways:

    - First Fit Memory Allocation
    - Best Fit Memory Allocation
    - Worst Fit Memory Allocation

Memory API

Memory API

- Memory allocation interfaces in UNIX system.
- In UNIX/C programs, understanding how to allocate and mange memory is critical in building robust and reliable software.
- What interfaces are commonly used?
- What mistakes should be avoided?

Types of Memory

- In C program there are two types of memory allocated

- Stack: Allocation and deallocation is managed implicitly by the compiler.
  - Sometimes, called as automatic memory.
  - When you return from the function, compiler deallocates the memory

```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

- For long-lived memory:
  - Malloc: Allocation and deallocation is manged by programmer

```
void func() {
  int *x = (int *) malloc(sizeof(int));
  ...
}
```

Compiler knows to make room for pointer to an integer when it sees (int *)
When it request malloc(), it requests space for an integer on the heap

Memory API: malloc()

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap.
  - Argument
    - `size_t size` : size of the memory block(in bytes)
    - `size_t` is an unsigned integer type (defined in C standard).
  - Return
    - Success : a void type pointer to the memory block allocated by `malloc`
    - Fail : a null pointer

Sizeof()

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Two types of results of `sizeof` with variables
  - The actual size of `'x'` is known at run-time.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - The actual size of `'x'` is known at compile-time.

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```

Memory API: free()

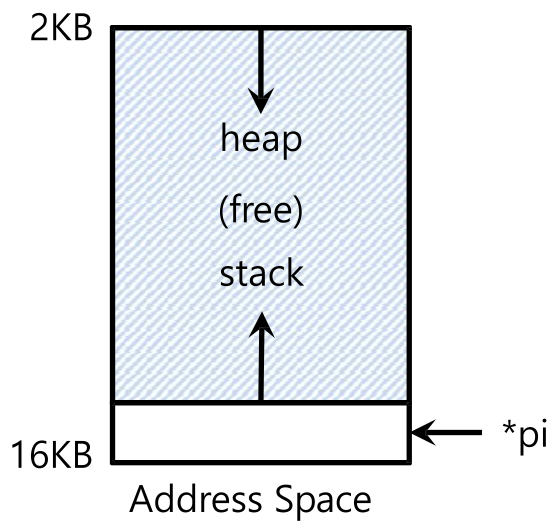
```c
#include <stdlib.h>

void free(void* ptr)
```

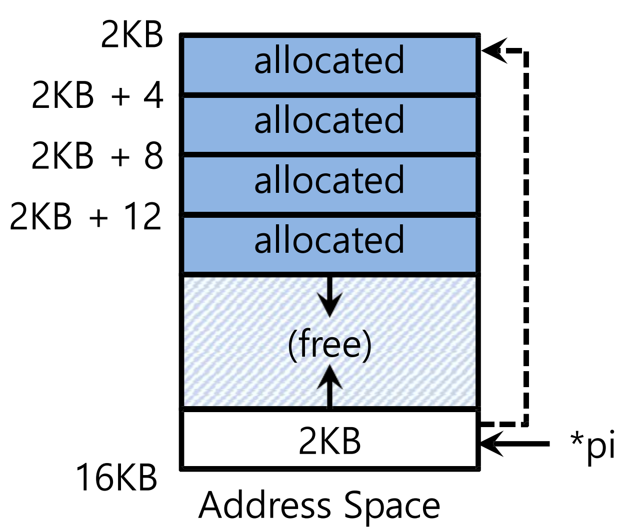
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating

pointer

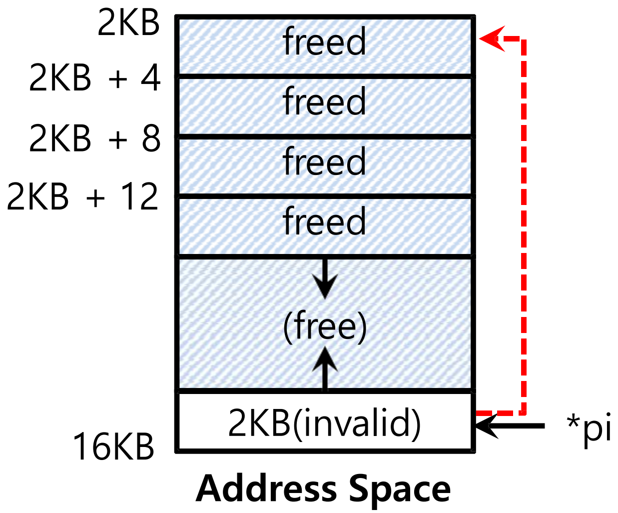



```c
int *pi; // local variable
```

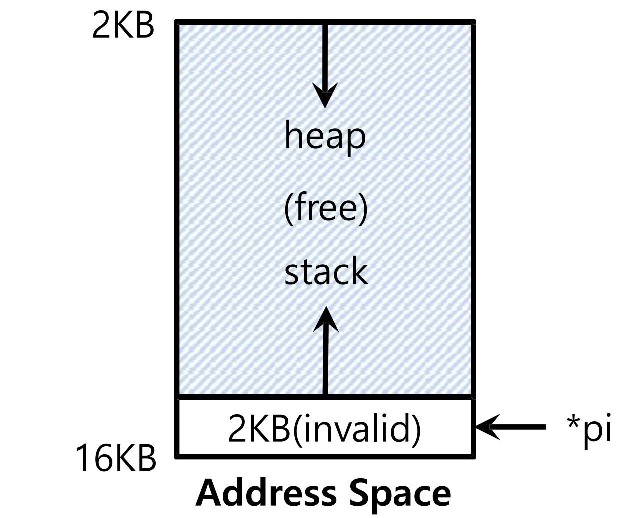



```c
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing




```
free(pi);
```



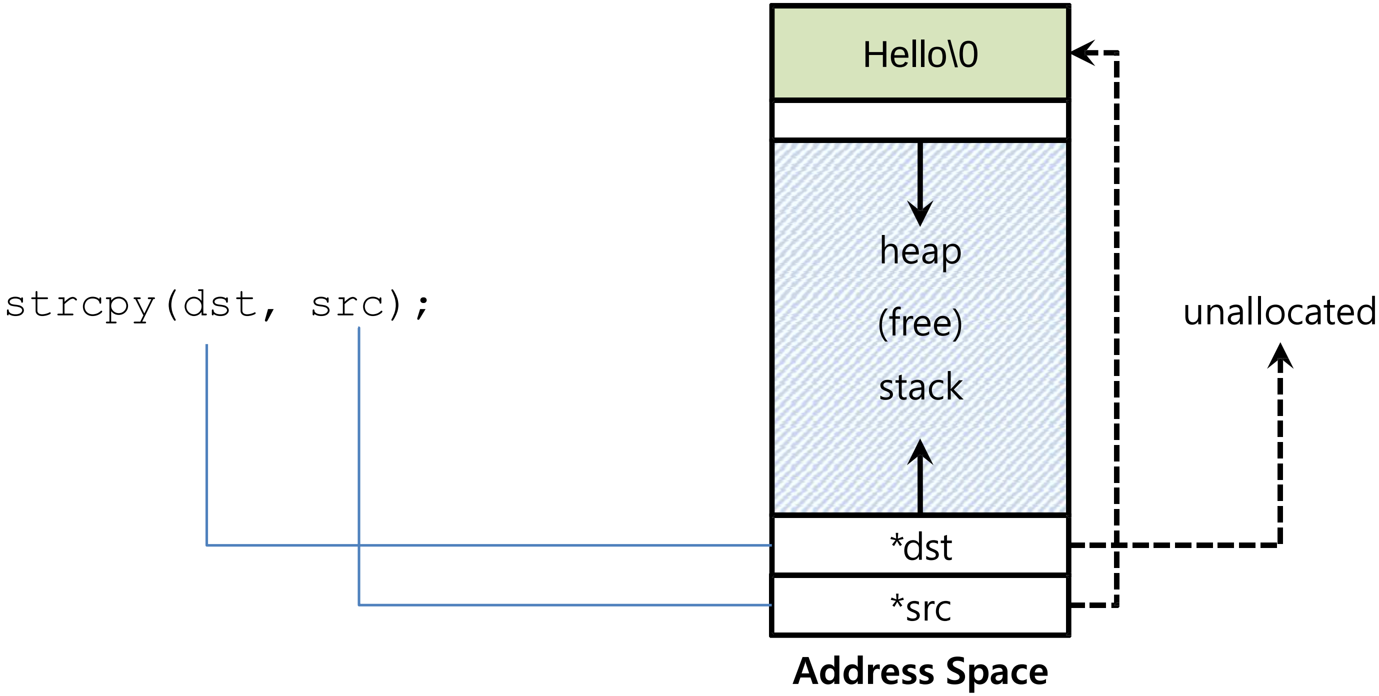
Common errors (that arise in the use of malloc() and free())

- **Forgetting To Allocate Memory**
- **Not Allocating Enough Memory**
- **Forgetting to Initialize**
- **Forgetting To Free Memory/Memory Leak**
- **Freeing Memory Before You Are Done With It/Dangling Pointer**
- **Freeing Memory Repeatedly/Double free**
- **Calling free() Incorrectly/Invalid Free**

Forgetting To Allocate Memory

- Incorrect code :segmentation fault

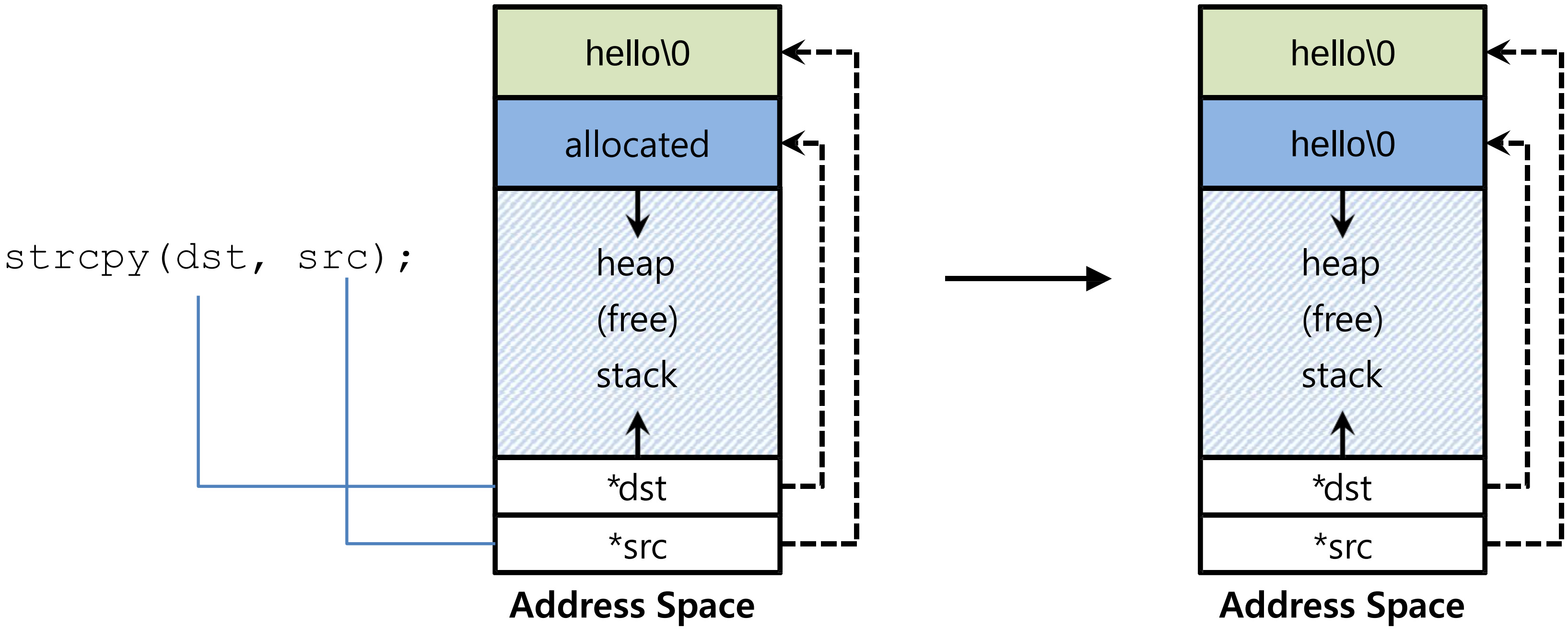
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

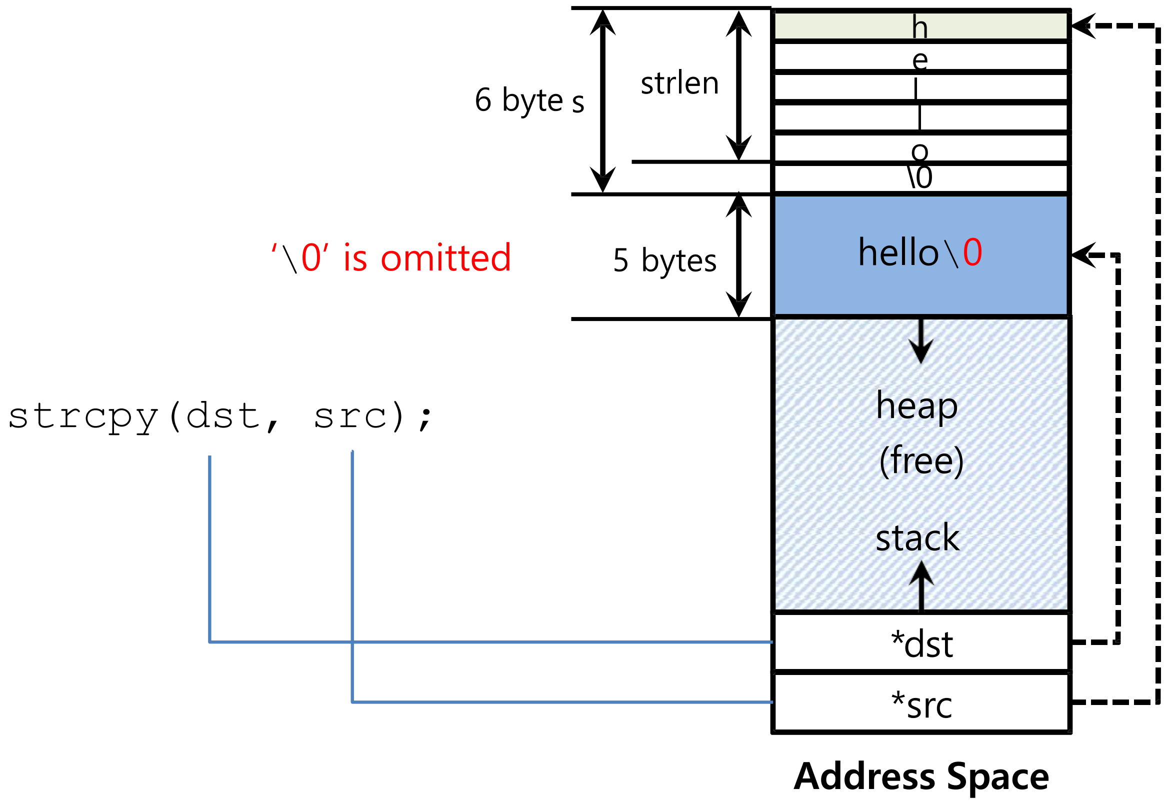
```
char *src = "hello";//character string constant
char *dst =(char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);    //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

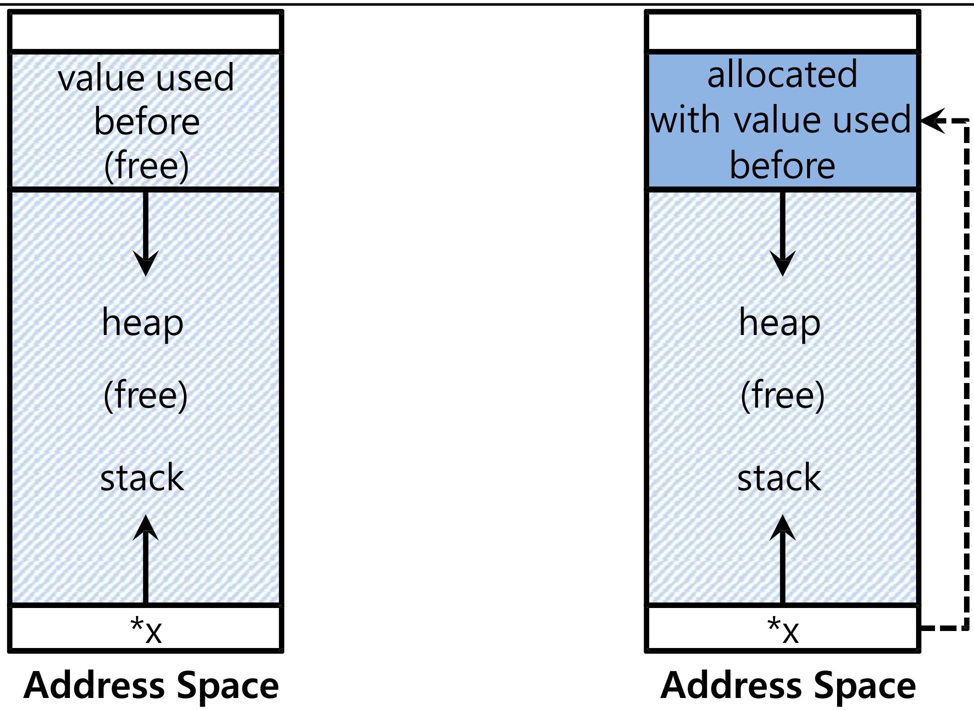
```
char *src = "hello"; //character string constant
char *dst =(char *)malloc(strlen(src)); // too
small  strcpy(dst, src); //work properly
```



Forgetting to Initialize allocated memory

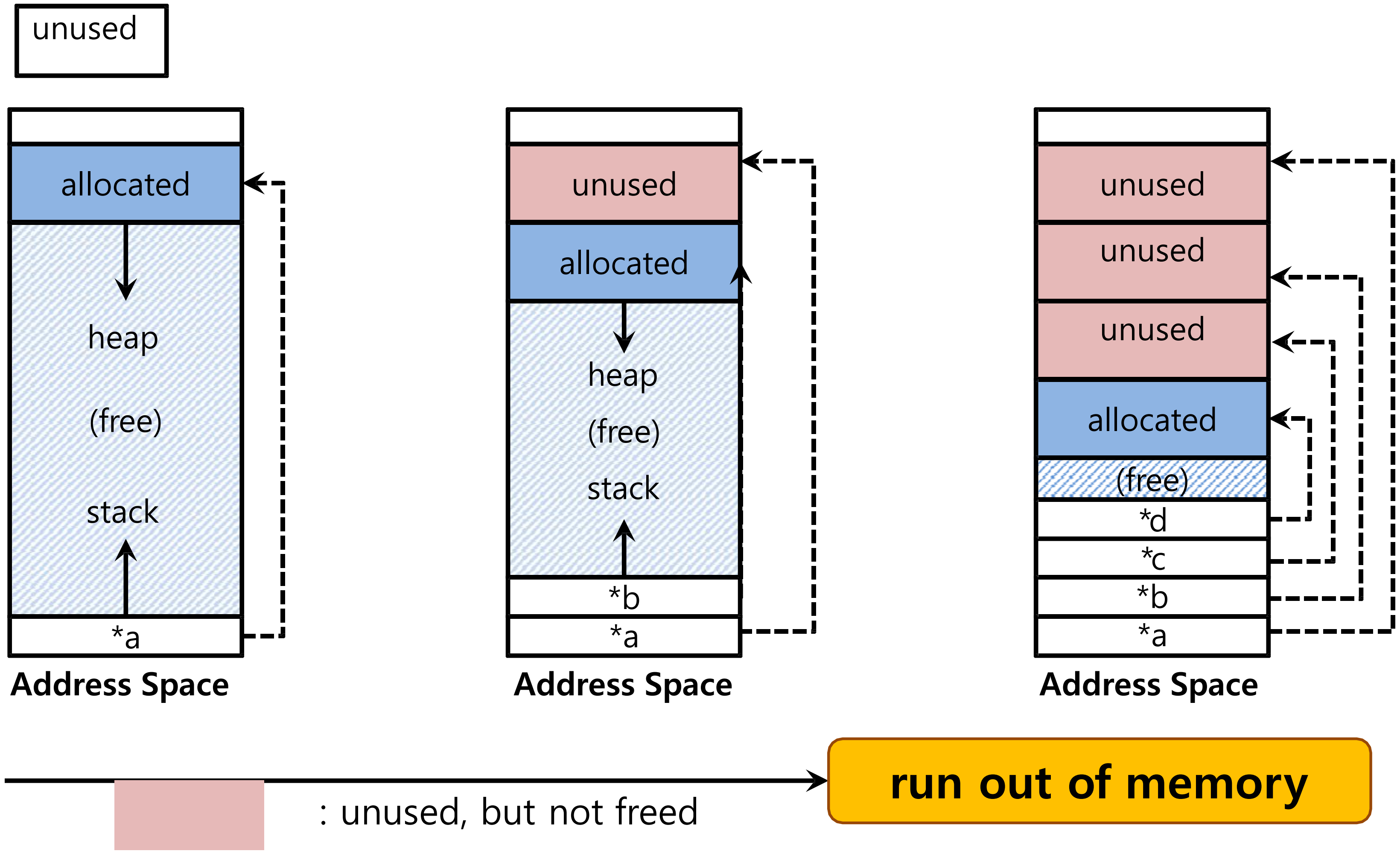
- Encounter an uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory
access
```



Memory Leak

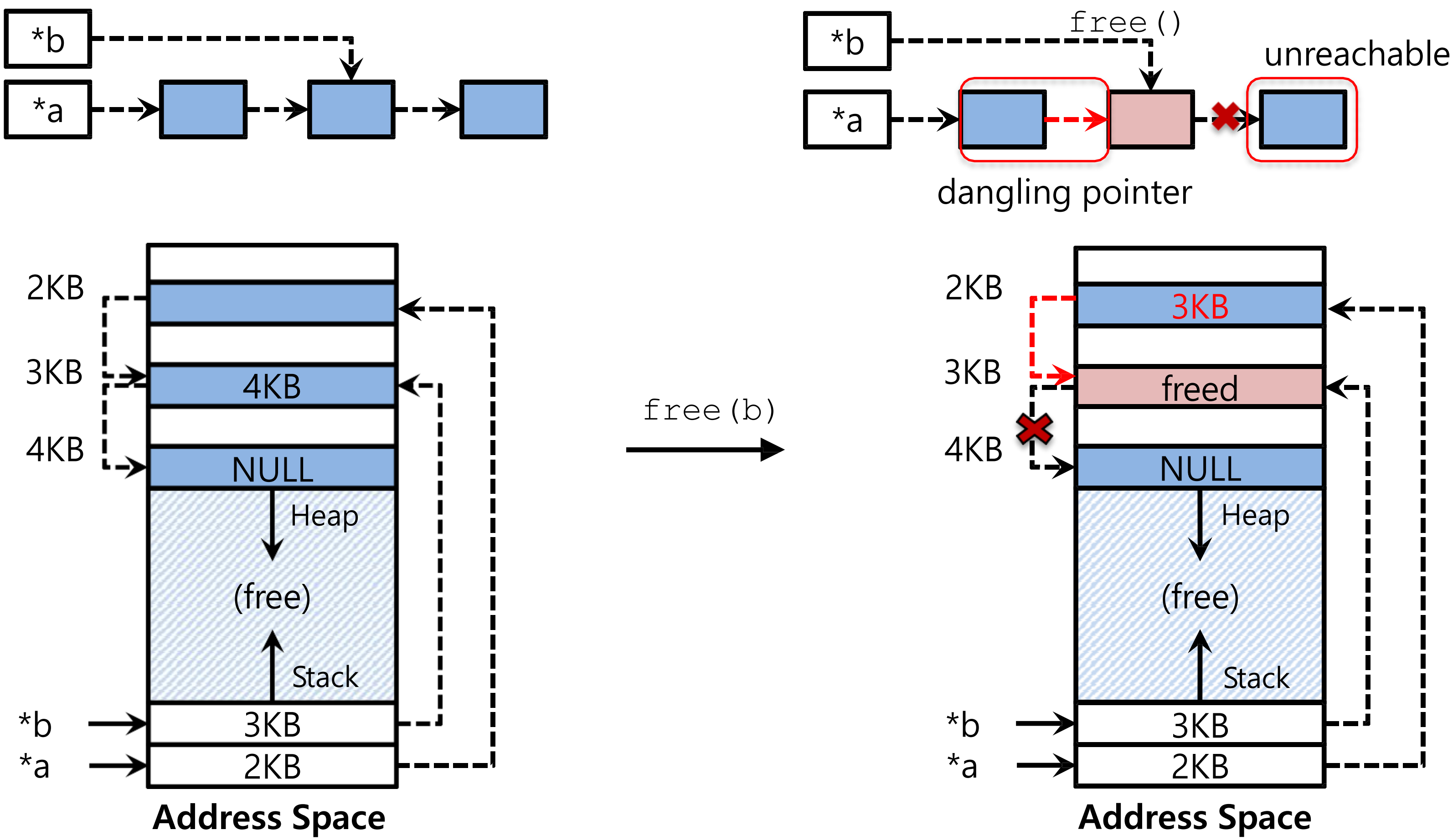
- A program runs out of memory and eventually dies.
- It occurs when you forget to free memory



- In long running applications or systems (such as OS) it is a huge problem.
- Slowly leaking memory leads one to run out of memory, because of restart is required.
- So, whenever we have done with chunk of memory it should be freed

Dangling Pointer

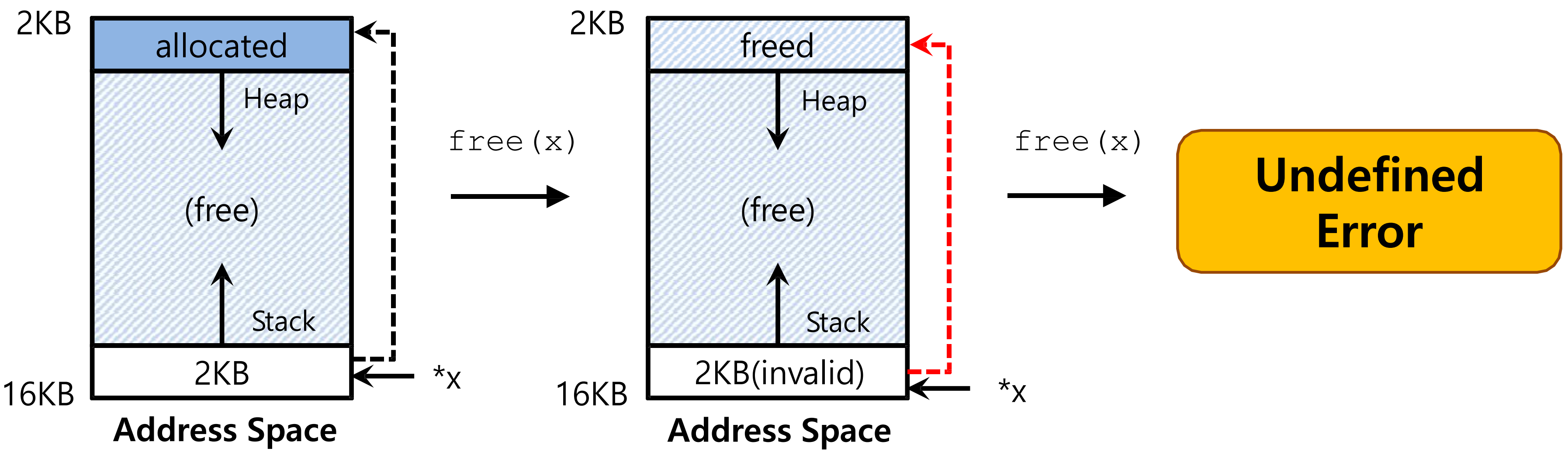
- Freeing memory before it is finished using
  - A program accesses to memory with an invalid pointer



Double Free

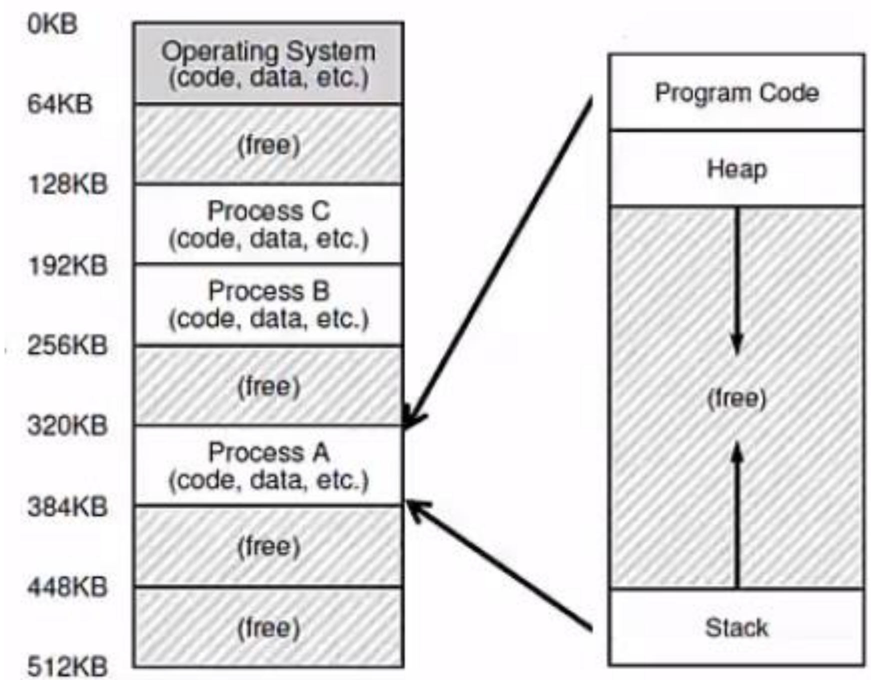
- Free memory that was freed already.

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



How is actual memory reached?

- When program is running: CPU fetches load and store instructions
- All of these accesses happen to virtual addresses.
- But in real life, program is not stored from 0 to MAX
- When you access a certain address, somebody translated from VA to PA.
- CPU issues load/stores to VA but memory hardware accesses PA
  - OS allocates memory and tracks location of processes.
- Translation done by memory hardware called memory management Unit(MMU)



Memory Allocation System Call

- Malloc implemented by c library
  - Algorithms for efficient memory allocation and free space management
- To grow heap, c library uses the brk/sbrk system call
- C library will take care of it.
- A program can also allocate a page sized memory using mmap() system call
  - Gets anonymous page from OS.
  - Gives page in the memory image of the process