Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T07 Memory Virtualization
Address Spaces

Referência principal
Ch.13 of Operating Systems: Three Easy Pieces by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 22 de agosto de 2018

Arthur João Catto, PhD
2º semestre de 2018

Memory Virtualization

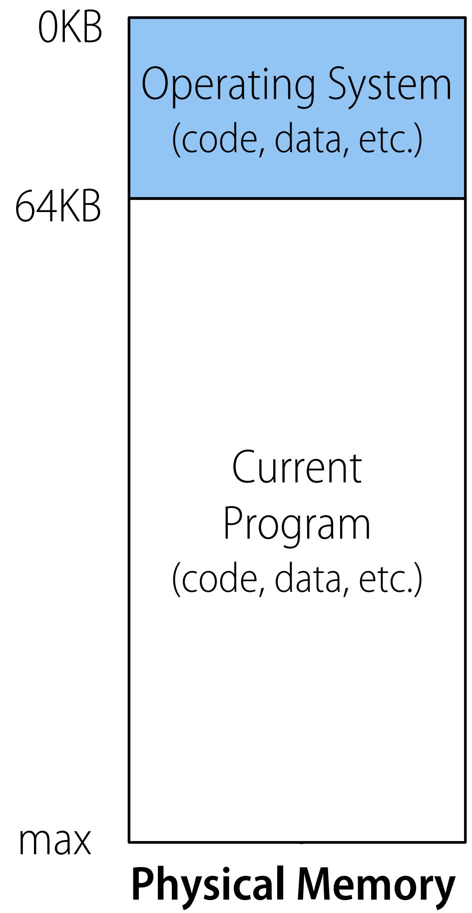
- What is memory virtualization?
  - OS virtualizes the physical memory.
  - OS provides an illusory (aka virtual) memory space to each process.
  - To the process, it looks like having the whole memory (or more!) at its disposal.

Benefits of Memory Virtualization

- Ease of use in programming
- Memory efficiency in terms of *time* and *space*
- A guarantee of isolation for processes as well as the OS
  - Protection from errant accesses of other processes

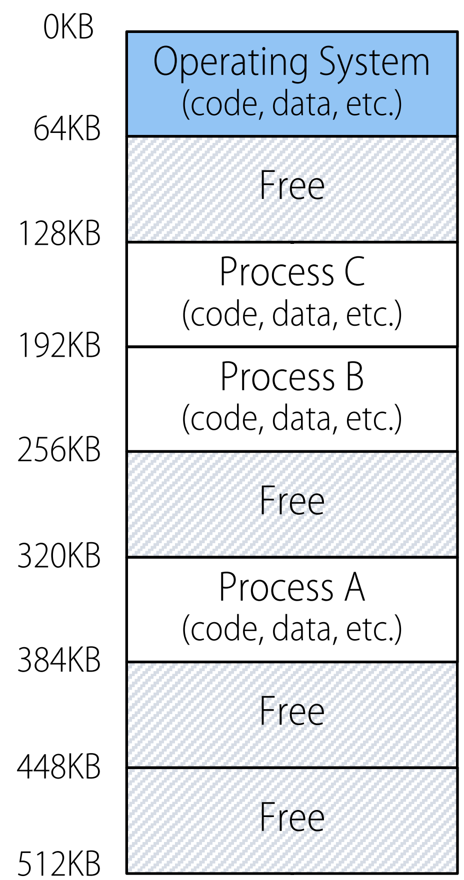
OS in early systems

- Only one process loaded in memory each time.
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

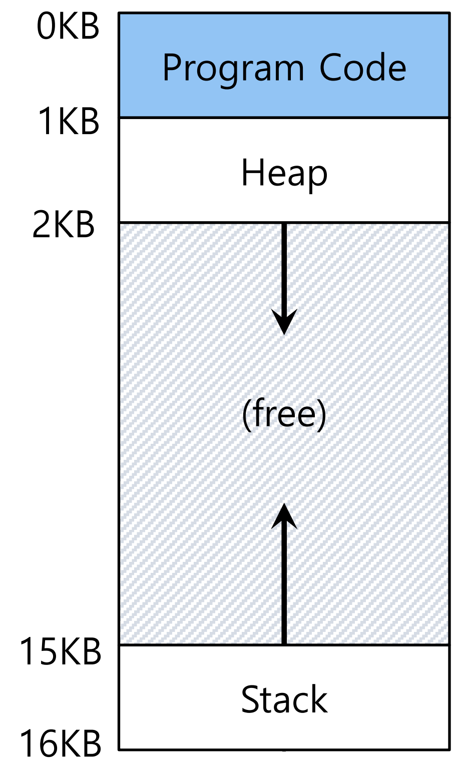
- **Several processes are loaded** in memory at the same time.
  - A process is executed for a short while.
  - System switches cyclically among ready processes in memory.
  - Utilization and efficiency are increased.
- **An important protection issue** is caused:
  - Errant memory accesses from other processes



Physical Memory

Address Space

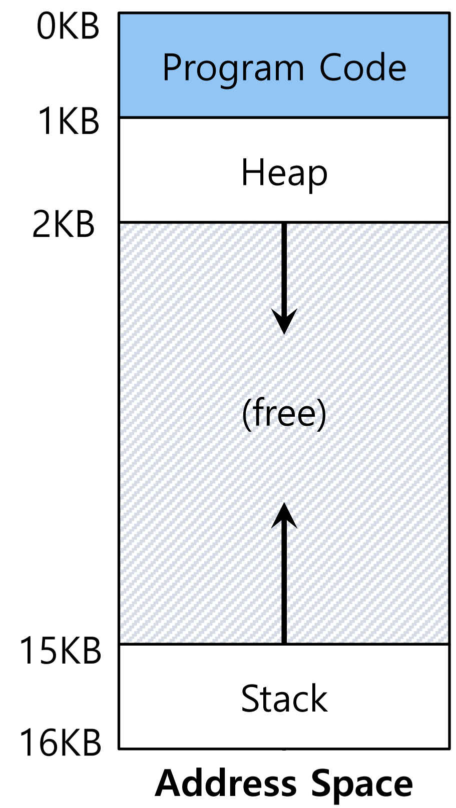
- OS creates an abstraction of physical memory.
  - The address space contains all about a running process.
  - That consists of program code, heap, stack and etc.



Address Space

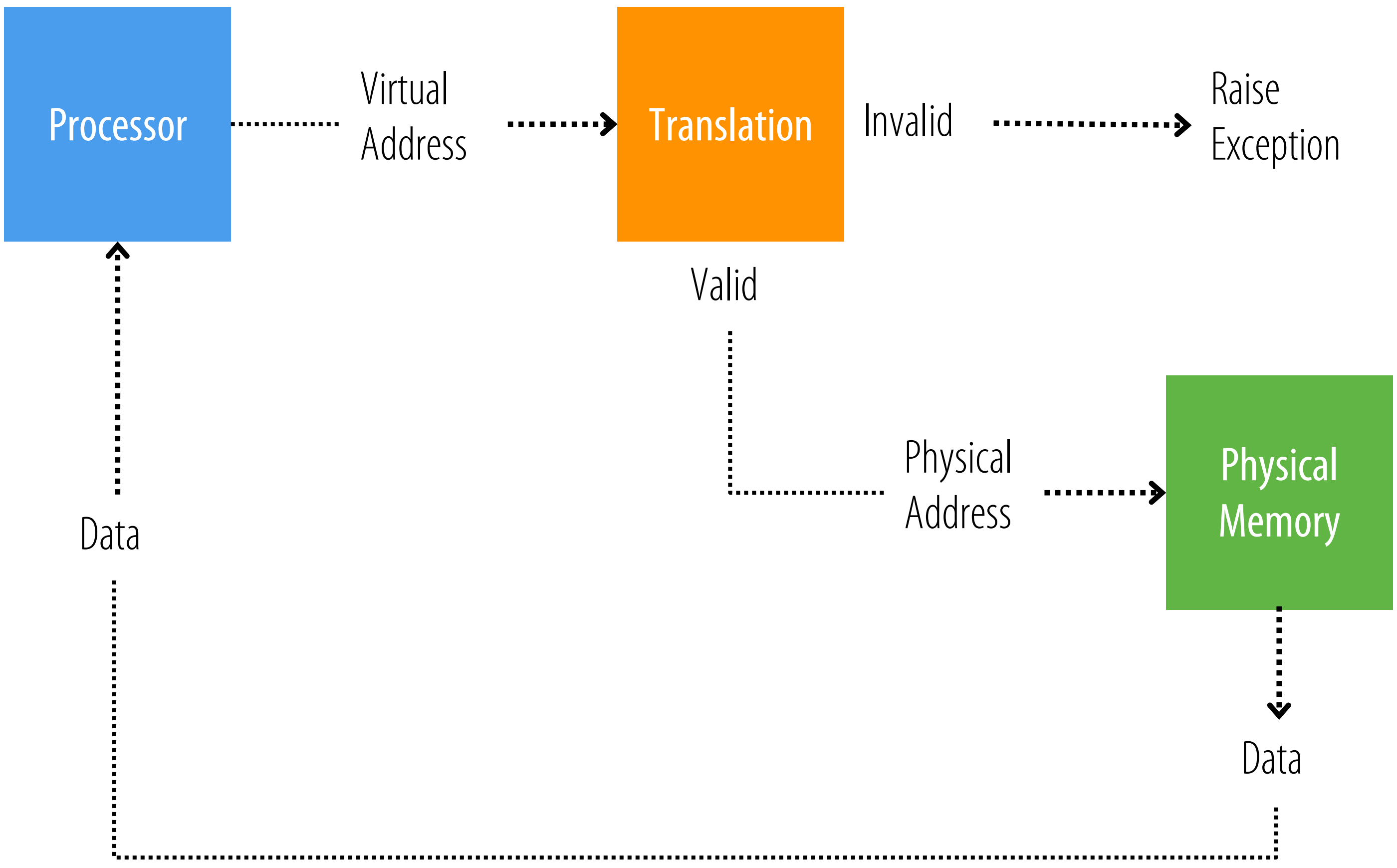
Address Space (cont.)

- Code
  - Is the area where instructions live
- Heap
  - Dynamically allocated memory.
    - `malloc` in C language
    - `new` in object-oriented language
- Stack
  - Keeps return addresses or values.
  - Contains local variables and arguments to routines.



Virtual Address Translation

- **Every address** in a running program is virtual.
  - OS translates the virtual address to a physical address



Example: a small C program that prints out addresses

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

int main(int argc, char *argv[]){
    printf("pid:%d code  is at %p\n",
            (int) getpid(), (void *) main);
    printf("pid:%d heap  is at %p\n",
            (int) getpid(), (void *) malloc(1));
    int x = 3;
    printf("pid:%d stack is at %p\n",
            (int) getpid(), (void *) &x);
    return 0;
}
```

Example: a small C program that prints out addresses

```
SUP080:atom arthur.catto$ gcc -o addresses2 addresses2.c -Wall
SUP080:atom arthur.catto$ ./addresses2
pid:1464 code  is at 0x105e41e90
pid:1464 heap  is at 0x7f8038402910
pid:1464 stack is at 0x7ffee9dbe9cc
SUP080:atom arthur.catto$ ./addresses2 & ./addresses2
[1] 1467
pid:1468 code  is at 0x10f2b9e90
pid:1468 heap  is at 0x7fe091c02910
pid:1468 stack is at 0x7ffee09469cc
pid:1467 code  is at 0x10560fe90
pid:1467 heap  is at 0x7ffb35402910
pid:1467 stack is at 0x7ffeea5f09cc
```

Example: a small C program that prints out addresses

```
SUP080:atom arthur.catto$ ./addresses2 & ./addresses2 & ./addresses2
[2] 1469
[3] 1470
pid:1470 code  is at 0x10e010e90
pid:1470 heap  is at 0x7ff06d400080
pid:1470 stack is at 0x7ffee1bef9cc
pid:1471 code  is at 0x1075cde90
pid:1469 code  is at 0x103bf6e90
pid:1469 heap  is at 0x7f9e09402910
pid:1469 stack is at 0x7ffeec0099cc
pid:1471 heap  is at 0x7f9e2bd00000
pid:1471 stack is at 0x7ffee86329cc
[1]   Done                    ./addresses2
[2]-  Done                    ./addresses2
[3]+  Done                    ./addresses2
SUP080:atom arthur.catto$
```