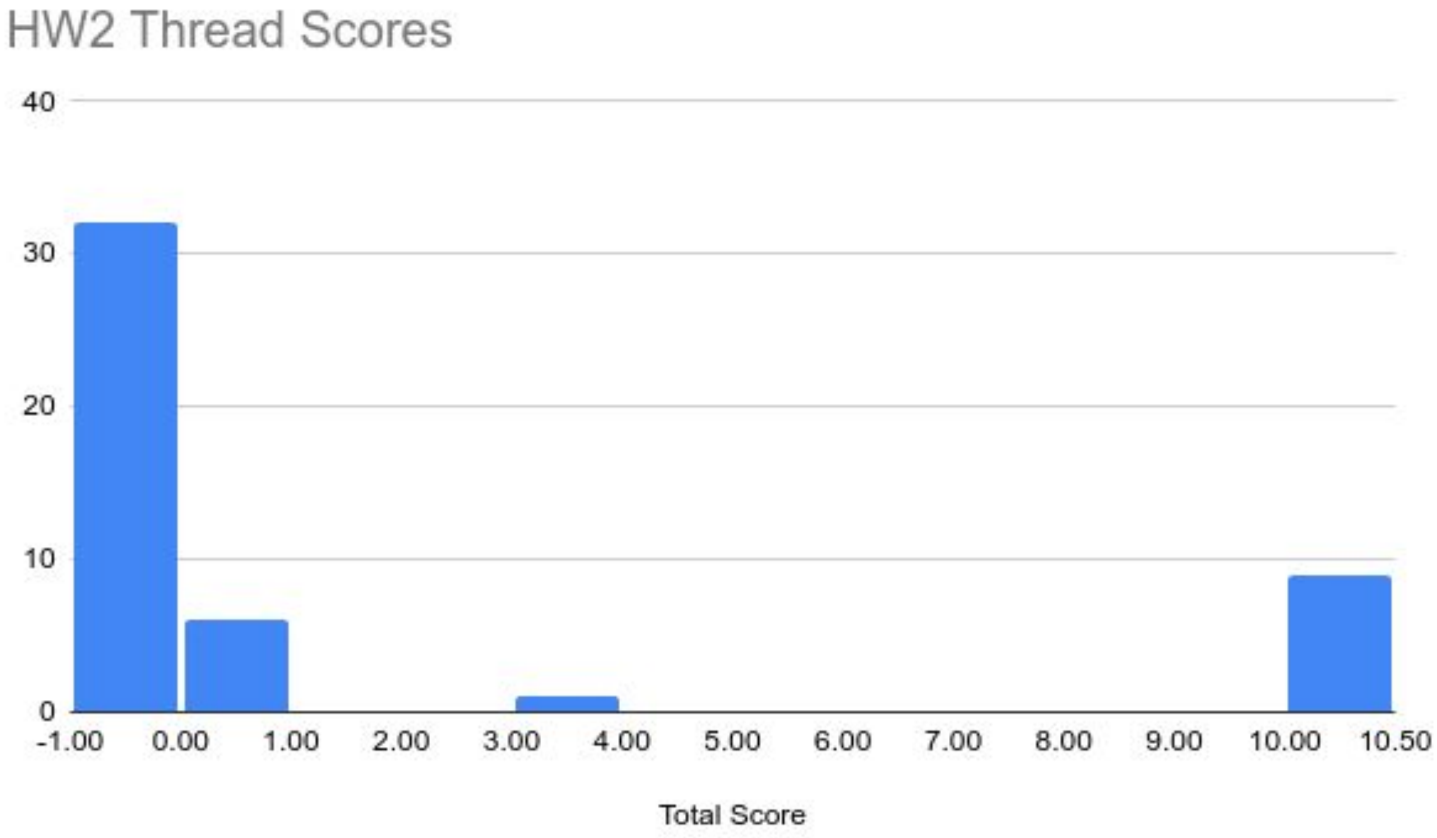
EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Status of assignment 2



Hacking overview

- discuss evolving tools document
  - please comment, we will add you to authors for significant contributions
- what we will cover in hacking:
  - compilers, function prototype
  - simple makefile
  - real implementation of dining philosophers
  - example of tests
  - example of debugging
- Before we have fun...

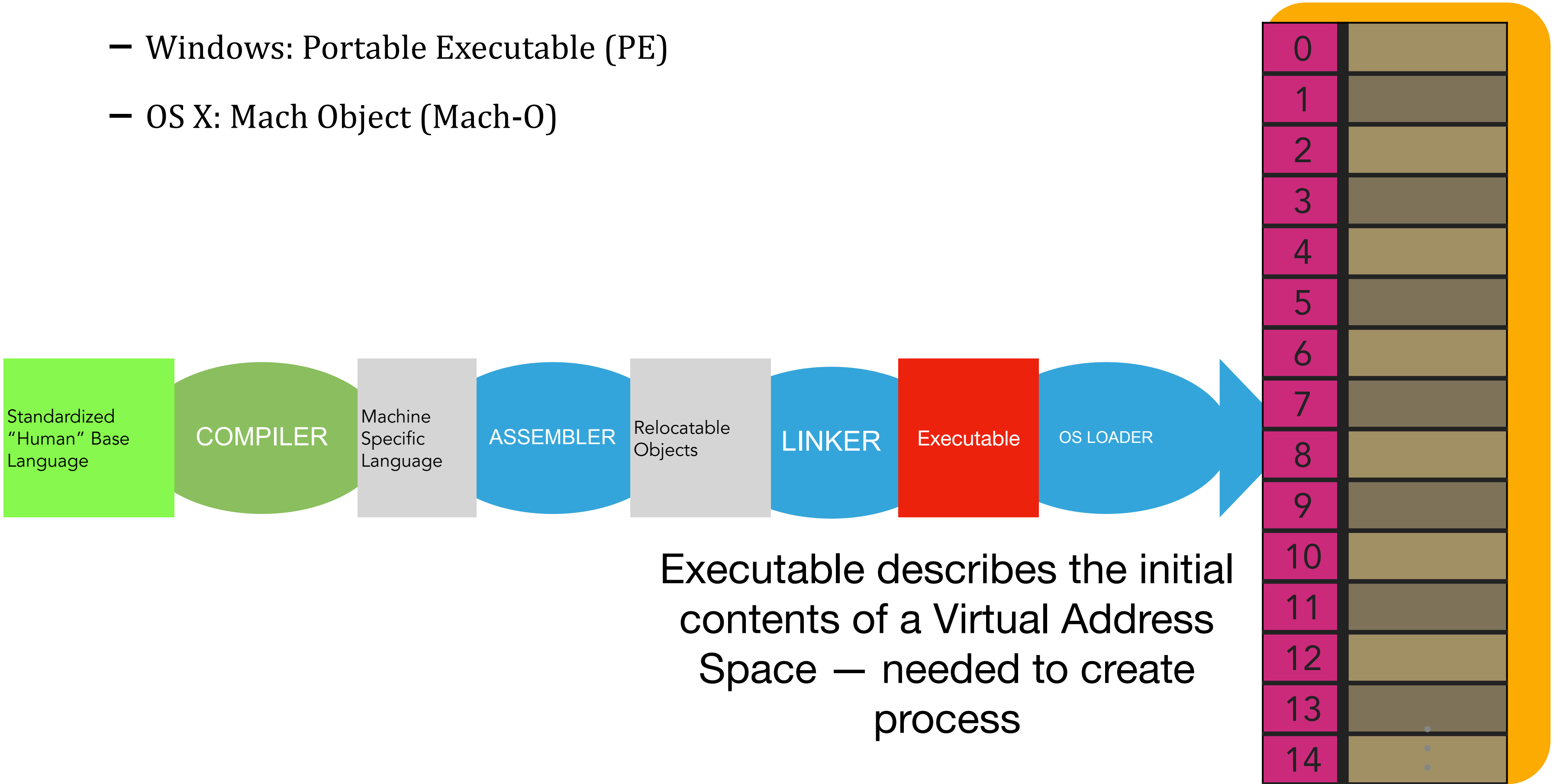
Compiler/linking

- Why write your code in multiple files?
  - modularity: lots of small files rather than one monolithic mess
  - efficiency: change one source file, compile, then relink don't re-compile other programs
- Requires:
  - header files that describe symbols/prototypes of functions shared
  - makefile that records the dependencies

Review compile, link...

Executable file that an OS can use to launch a process

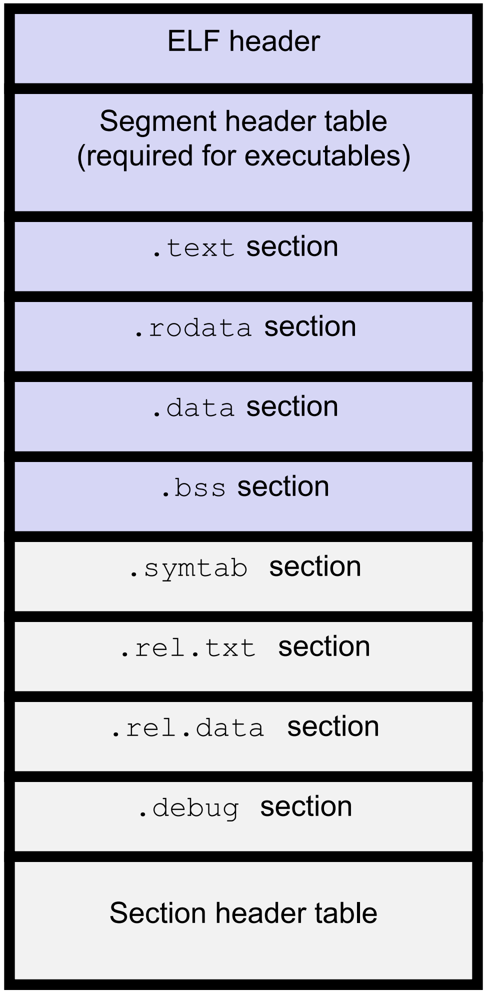
- Compiler produces a special kind of file that your OS can use to create/load a Virtual Address Space/Process
  - LINUX: Executable Linkable Format (ELF)
  - Windows: Portable Executable (PE)
  - OS X: Mach Object (Mach-O)



Executable describes the initial contents of a Virtual Address Space — needed to create process

ELF Object File Format

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"



ELF header

Segment header table
(required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

`.rel.data` section

`.debug` section

Section header table

Getting concrete

- Compiler creates a ".o" elf file with bunch of undefined symbols
- Linker:
  - resolves symbols between the different ".o" files
  - decides where everything is going to go in the final executable
  - patches up references to point to where it put things
  - if same symbol (function, data...) in two files, blows up
- Linker/compiler are all gcc with right flags

Now lets have fun

Memory Management

Memory

- Paraphrase of Parkinson’s Law, ‘‘*Programs expand to fill the memory available to hold them*.’’
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

The Memory Manager

- The portion of the OS that allocates, frees, and tracks the usage of RAM is the *memory manager*
- Fundamental jobs of the memory manager:
  - Abstracting a memory model the is conducive writing, running and debugging programs.
  - Managing the private virtual address space of all processes.
  - Protecting each virtual address space from others.
  - Managing all of the physical memory on the system.

Before Memory Management

- Early computers had no abstraction for memory
- You ask for data at address 0x1234, you get the data stored at physical memory location 0x1234
- This is often called the *physical* memory model because every address refers directly to a physical location in memory
- The physical memory model provides no separation or protection between processes.

Memory

- RAM is one of the main resources managed by an operating system:
  - fast
  - small(compared to disk)
  - expensive(compared to storage)
  - *volatile* storage (does not persist across reboots)
    - Though perhaps new technology like NVRAM will change that …
- The portion of the OS that allocates, frees, and tracks the usage of RAM is the *memory manager*

Physical Memory Model Organization

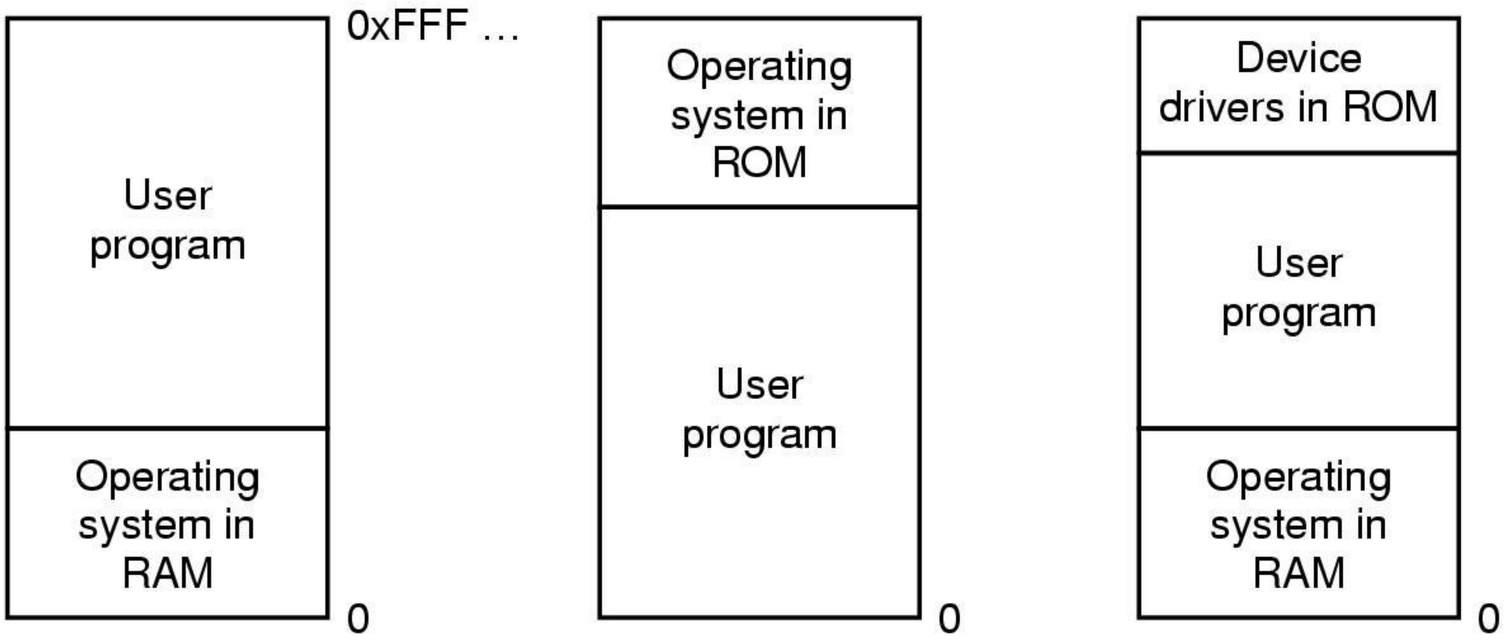
Even with such a simple model, there are still decisions to be made:

- Where do we put the OS/kernel code?
- Where do user programs go?
- How do we support multiple processes and programs?
- How do we protect them from each other?

Simplest Memory Management

Sounds easy to do, but what are the downsides?

- Run only one user process at a time
- Operating/system and device drivers resident or in ROM



Running Multiple Programs Without a Memory Abstraction

Addresses need to be *fixed-up* when programs are loaded into memory

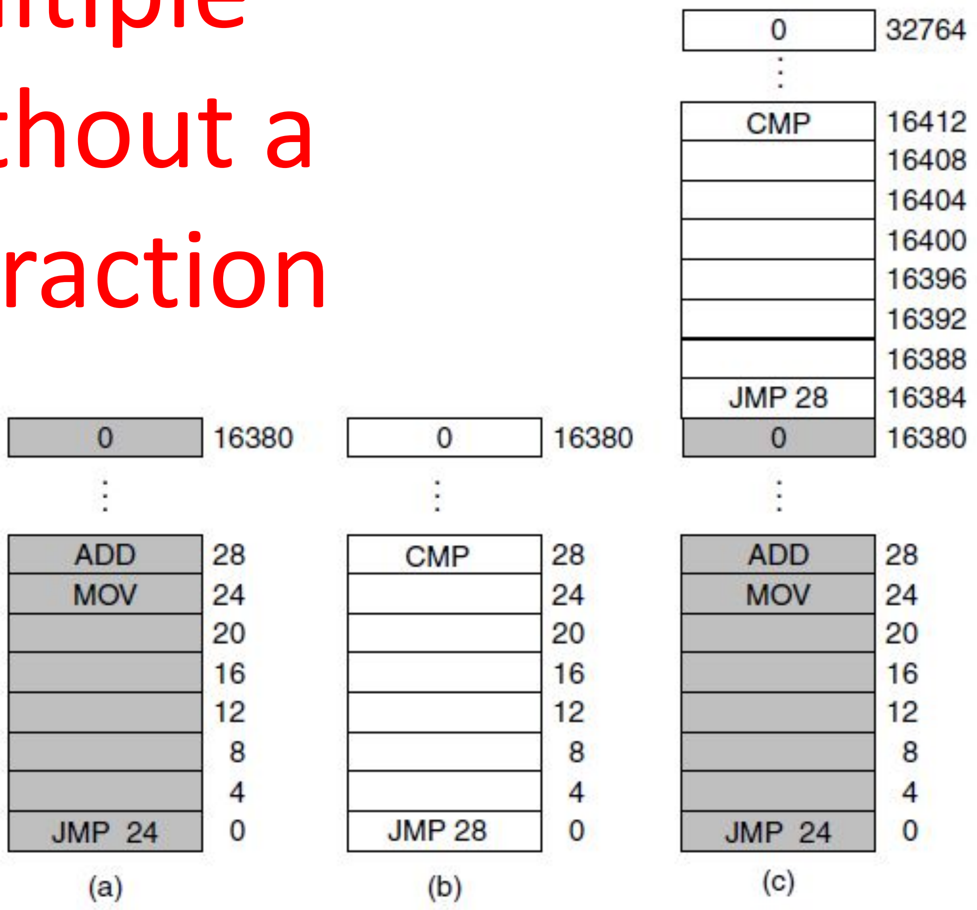
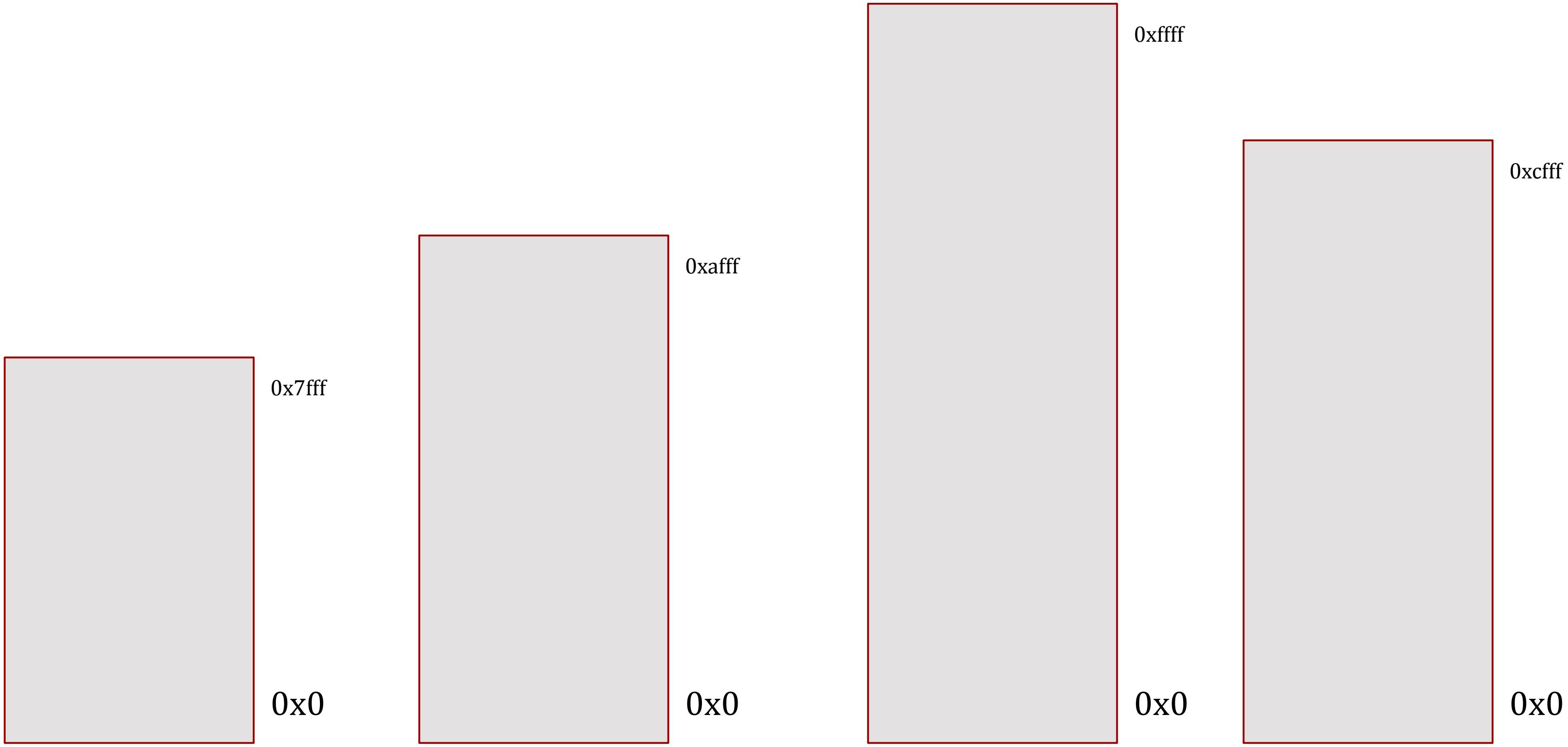


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

What we really want: Virtual Address Spaces

- An abstraction, so that each process has a *private address space*: make 0x1234 in Program A different from 0x1234 in Program B
- Different sizes for different process needs.
- Look and act like an array of bytes starting at address 0x0 ending at end if desired size.
- Each virtual address space maps to a different physical address.
- Simplest scheme with first computers (and 16/32 bit mode x86) is segmentation where you use base (and limit) registers. Still used heavily today in embedded systems.

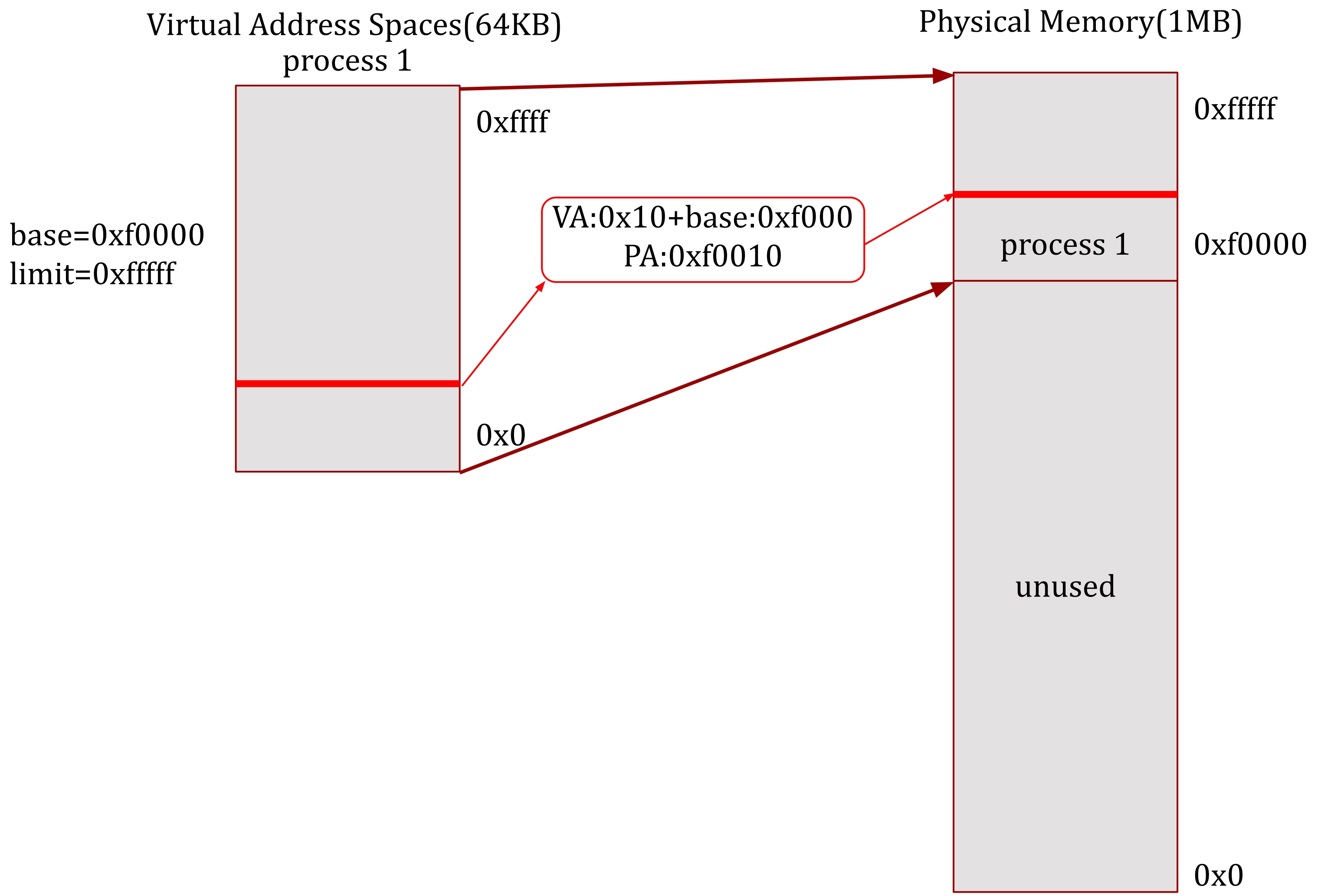
Virtual Address Spaces



Simple Segmentation

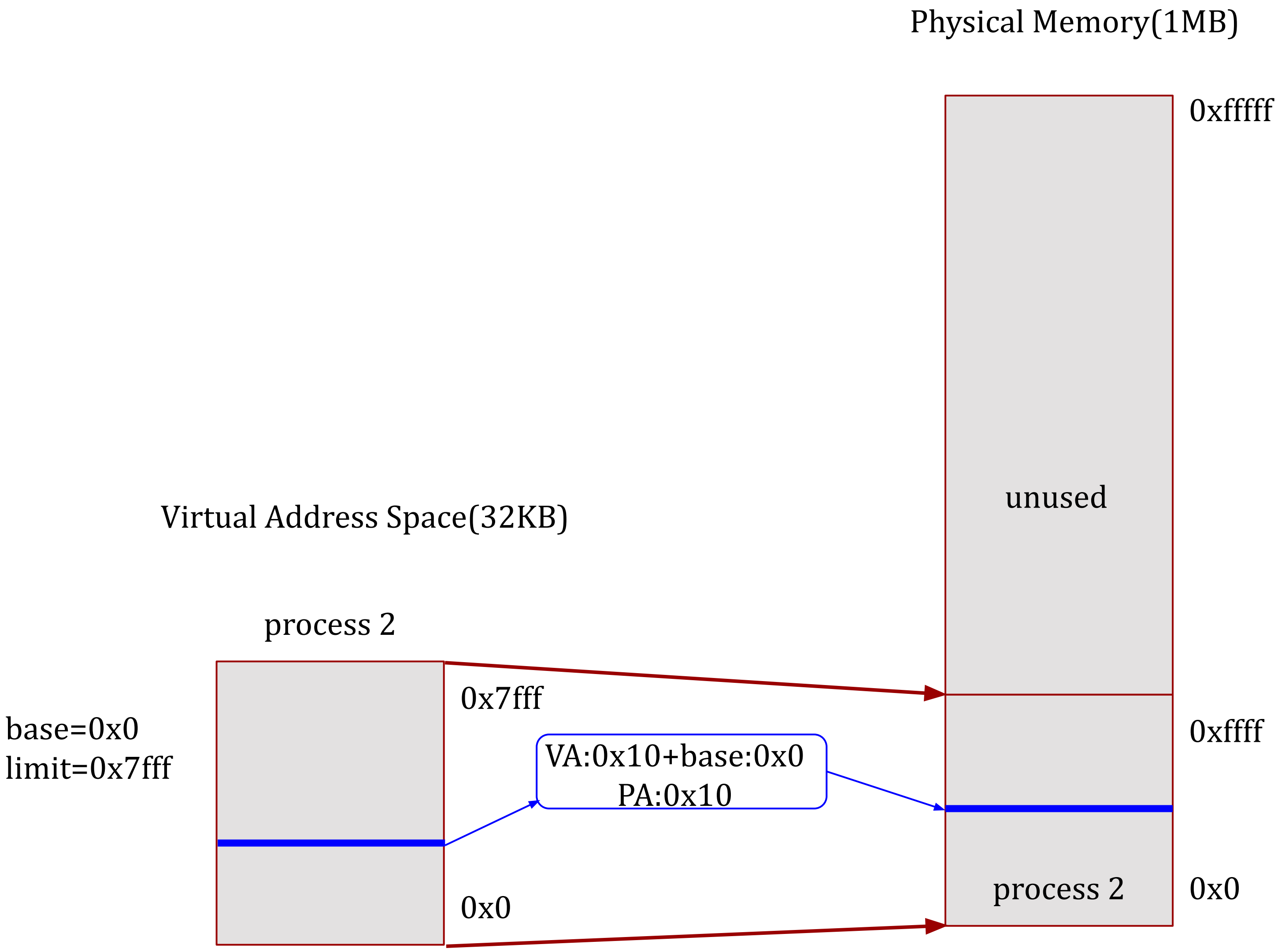
- To support Segmentation the CPU has *base* and *limit* registers
- Each time a memory address is referenced, the CPU transparently adds the *base* to it and verifies that *base* + address ≤ *limit*
  - The CPU adding of the *base* to the address is known as translation.
  - If that *base* + address > *limit* a *segfault* results.
- We refer to the memory address as a “virtual address” and the *base*+address is a “physical address”.
- Each process with a separate address space has different base and limits.
  - Context switching to a new process includes loading the process specific *base* and *limit* registers
  - every process can have a virtual address space similar to other processes.
  - but map to different physical addresses.
- Downside:
  - Memory access was slightly slower because of the address+base addition but hardware resolves this issue.
  - Virtual address space size can not exceed physical memory size.

Simple Segmentation Translation



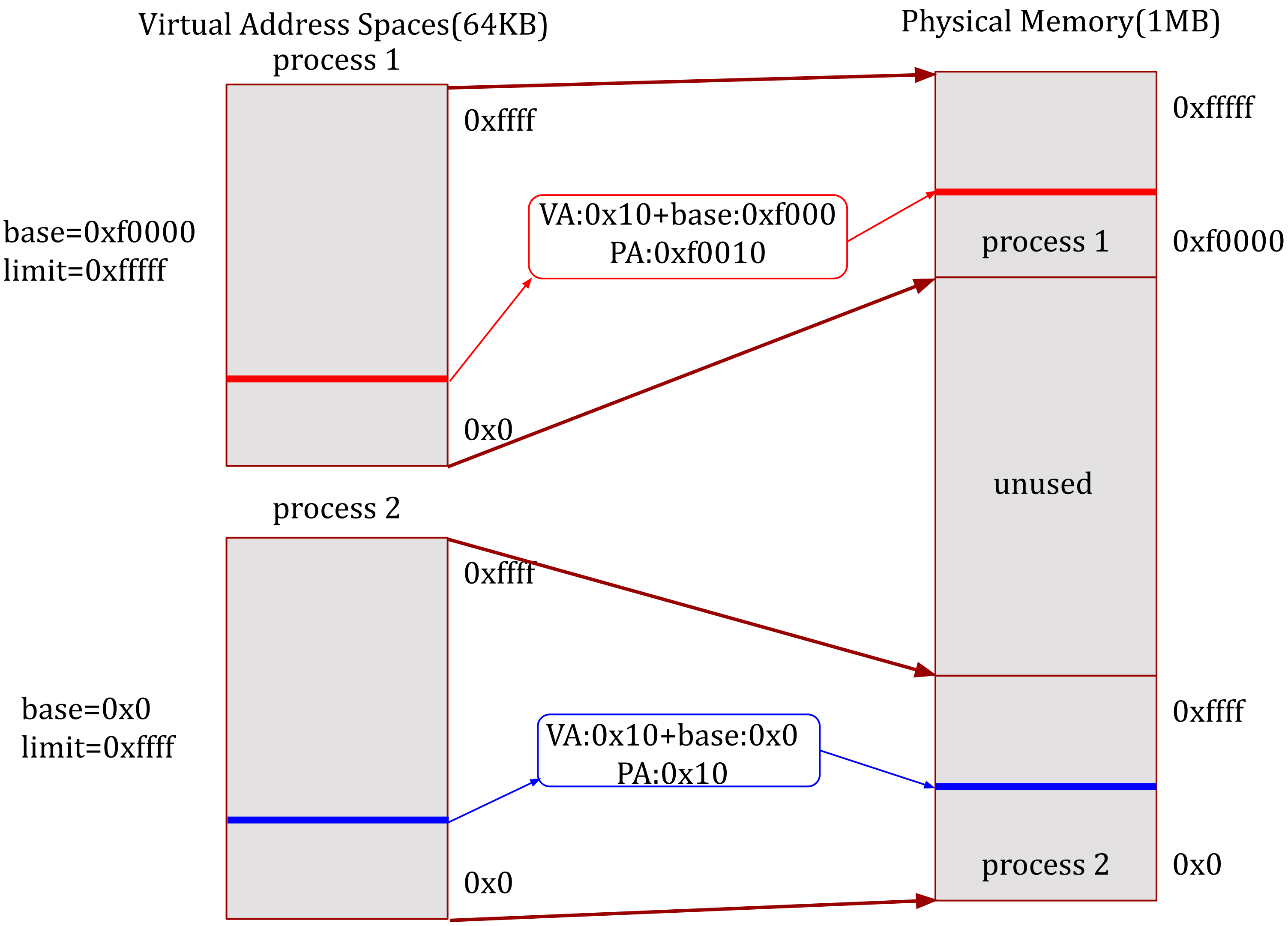
source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Simple Segmentation Translation



source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Simple Segmentation Translation

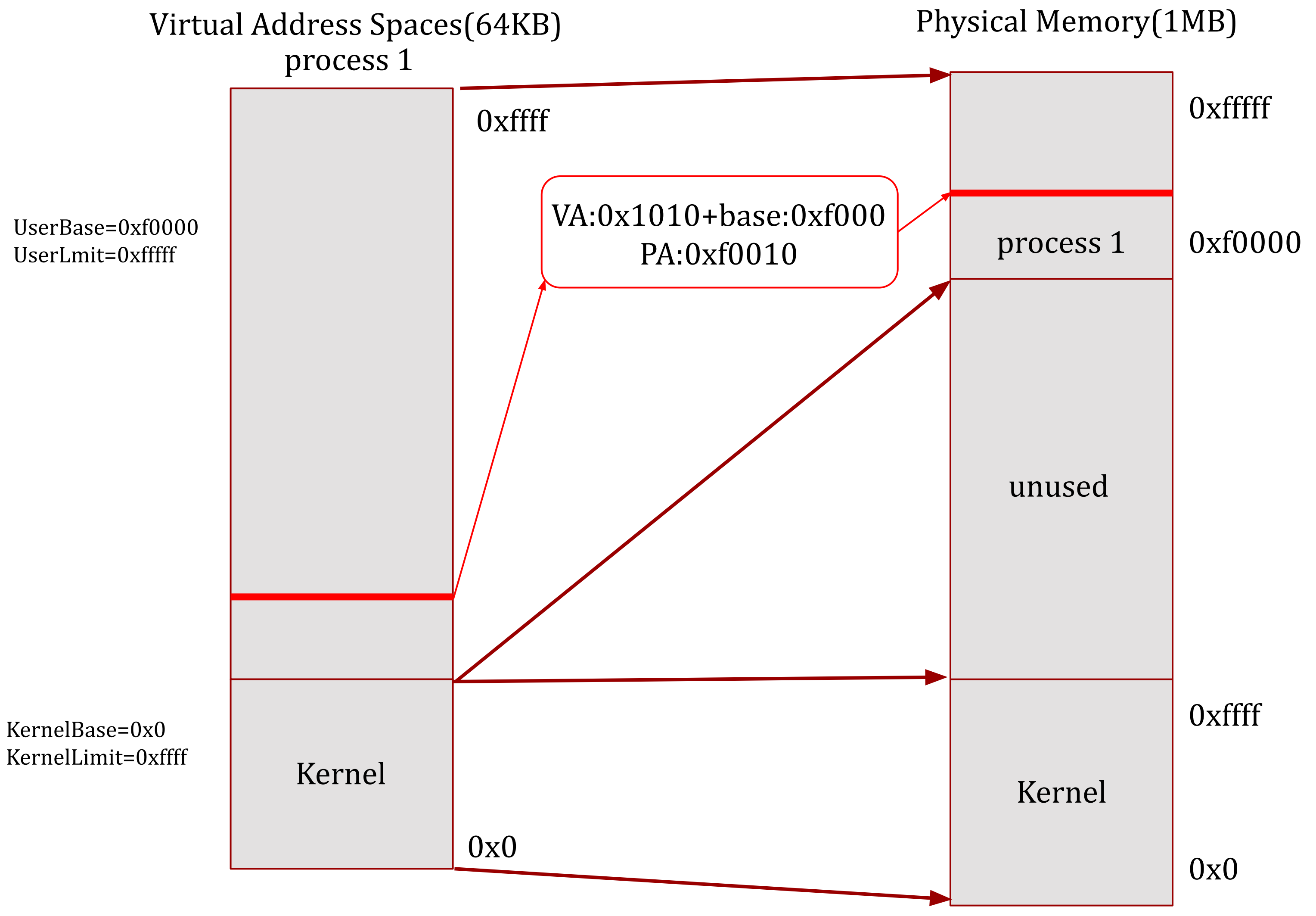


source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Real World Segmentation

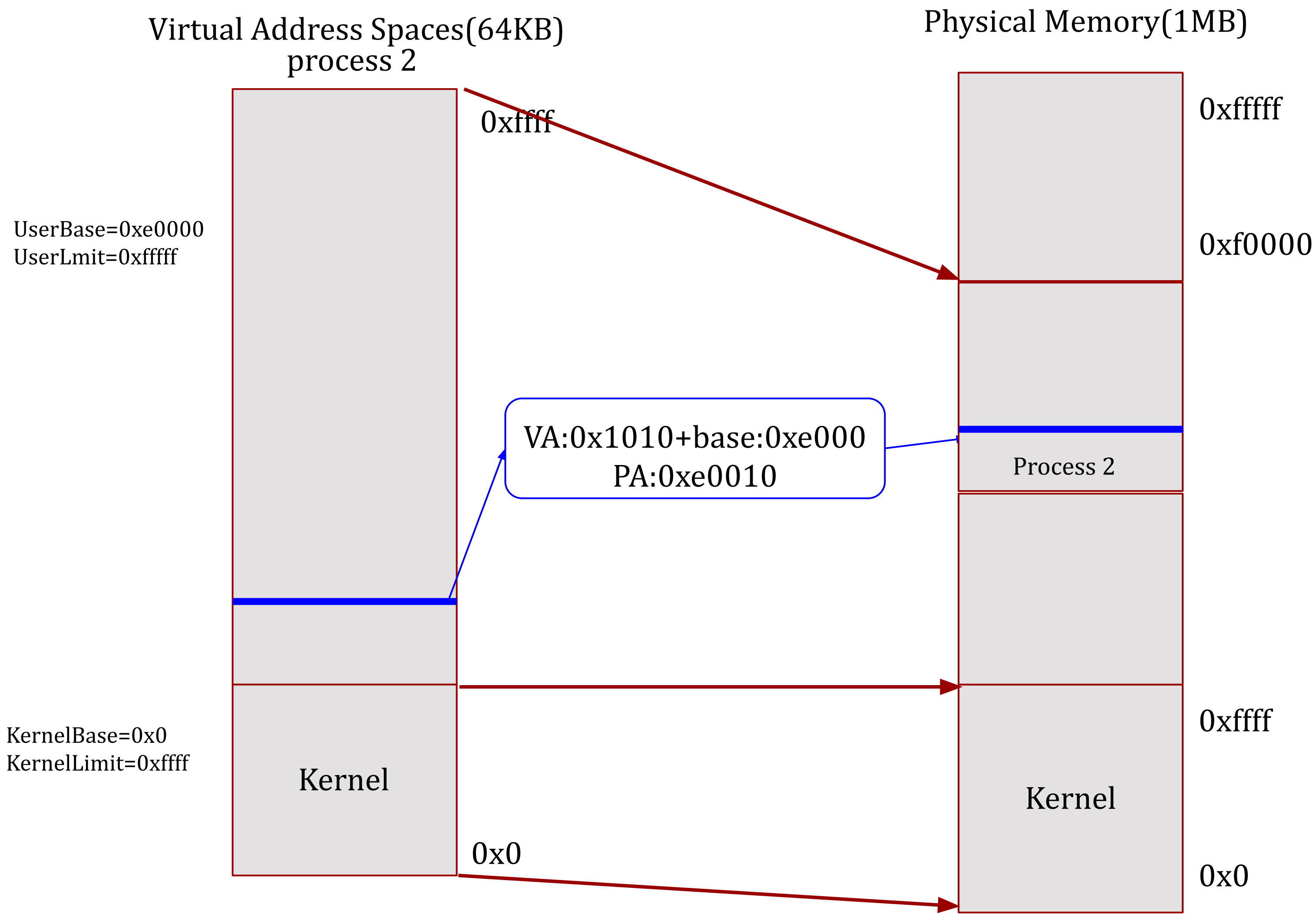
- In the real world there are 2 sets of segmentation registers:
  - User Segmentation Registers
  - Kernel Segmentation Registers
- Rather than duplicating the kernel it is shared and mapped into every process address space
  - Same virtual window
  - Usually privileged
  - System calls jump into/out of this window
- Context switch code changes User Segment Registers but not the Kernel Segment Registers

Process1 Segmentation Translation



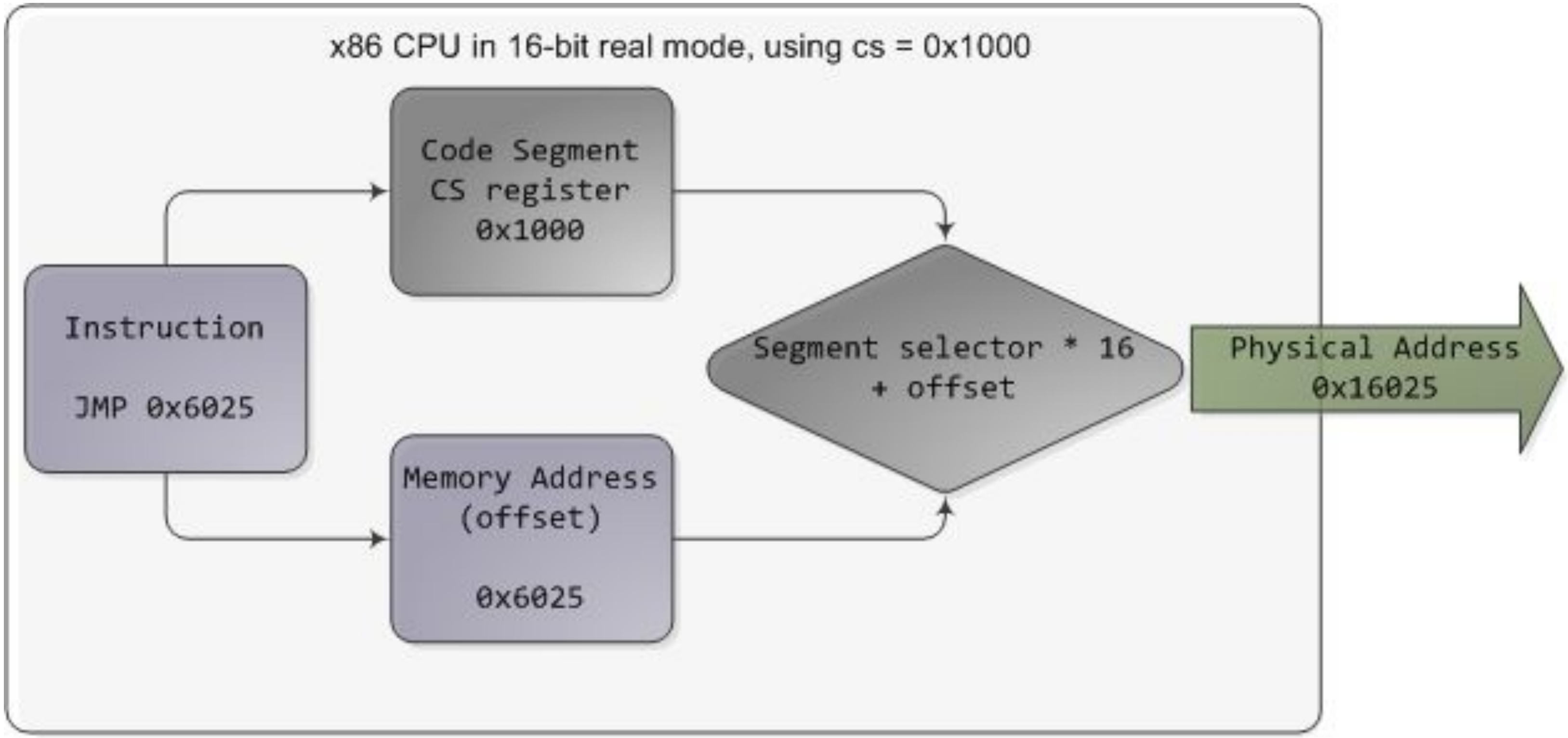
source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Process2 Segmentation Translation



source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

x86 Real Mode Segmentation



source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Base and Limit Registers

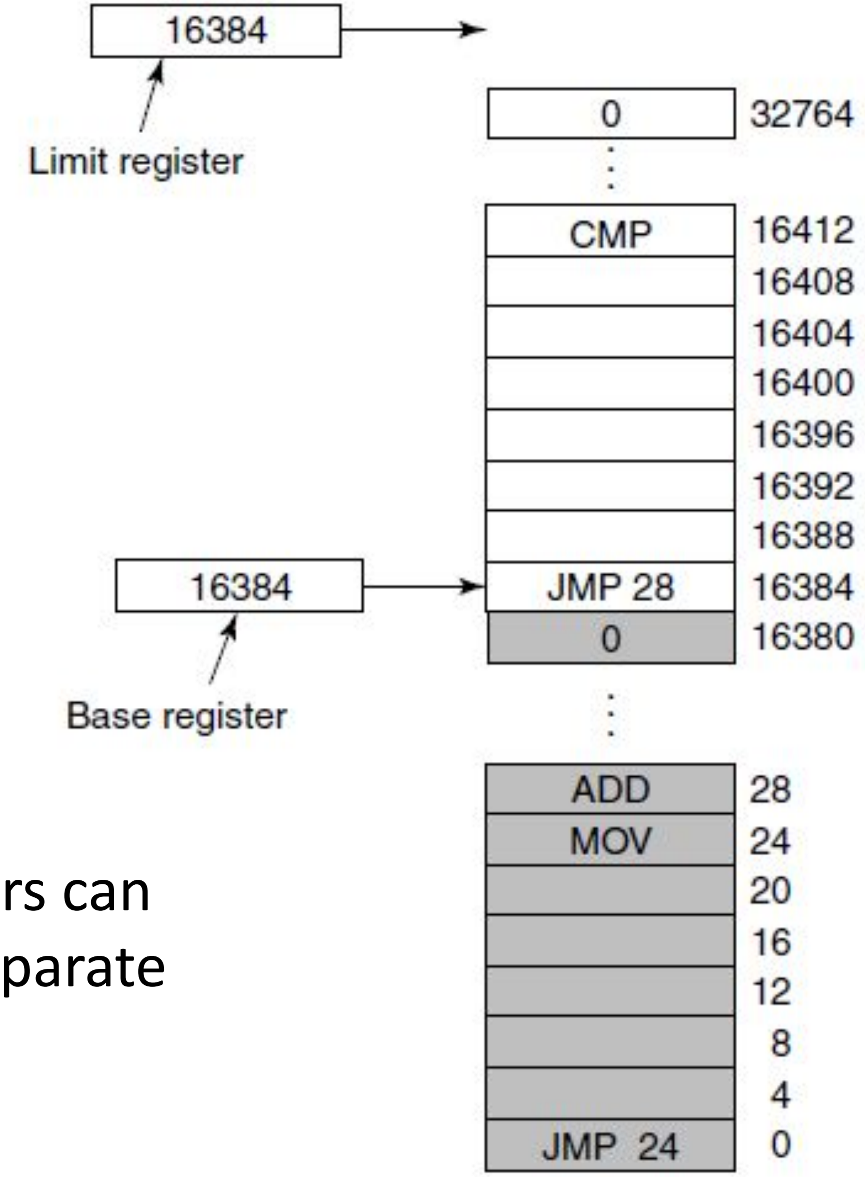
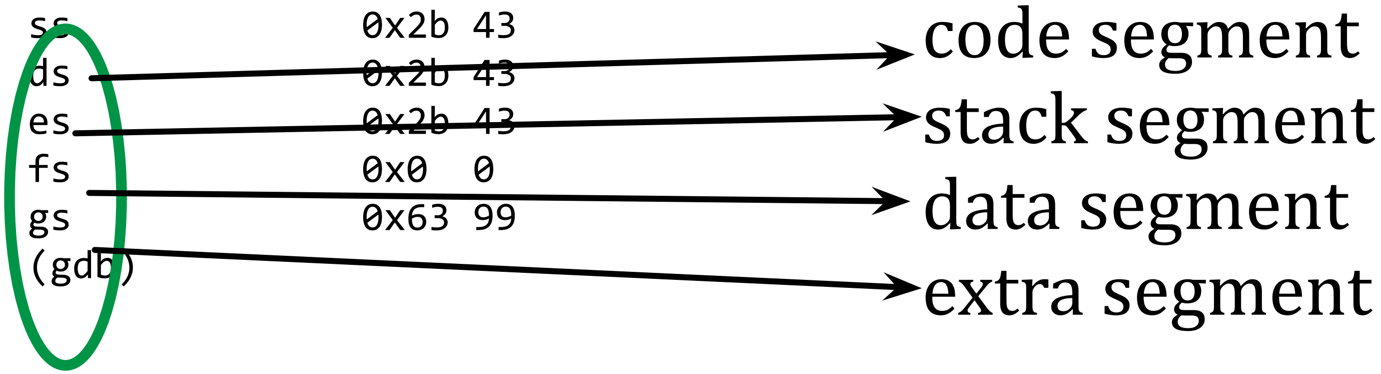


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

x86 Segment Registers

```
(gdb) info registers
eax            0x1  1
ecx            0xffffd840   -10176
edx            0xffffd864   -10140
ebx            0xf7fa6000   -134586368
esp            0xffffd824   0xffffd824
ebp            0xffffd828   0xffffd828
esi            0x0  0
edi            0x0  0
eip            0x8048409    0x8048409 <main+14>
eflags         0x286    [ PF SF IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0  0
gs             0x63 99
(gdb)
```



code segment

stack segment

data segment

extra segment

Physical Memory Management

Ideally memory would be

- Large
- Fast
- Non volatile

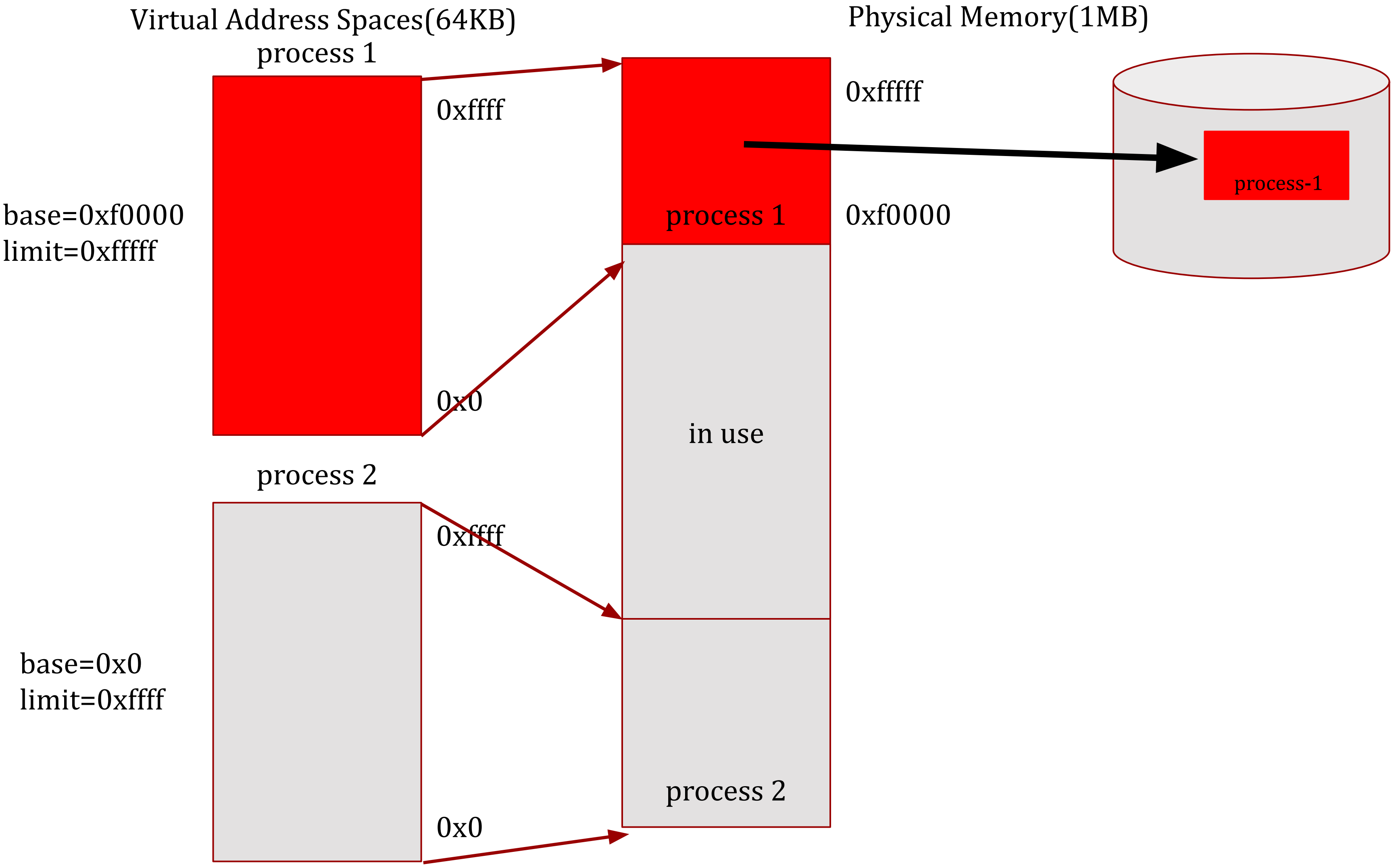
In practice, a memory hierarchy is used

- Small amount of fast, expensive memory (cache)
- Some medium-speed, medium price (main memory, RAM)
- Gigabytes of slow, cheap disk storage

Application/Segment Swapping

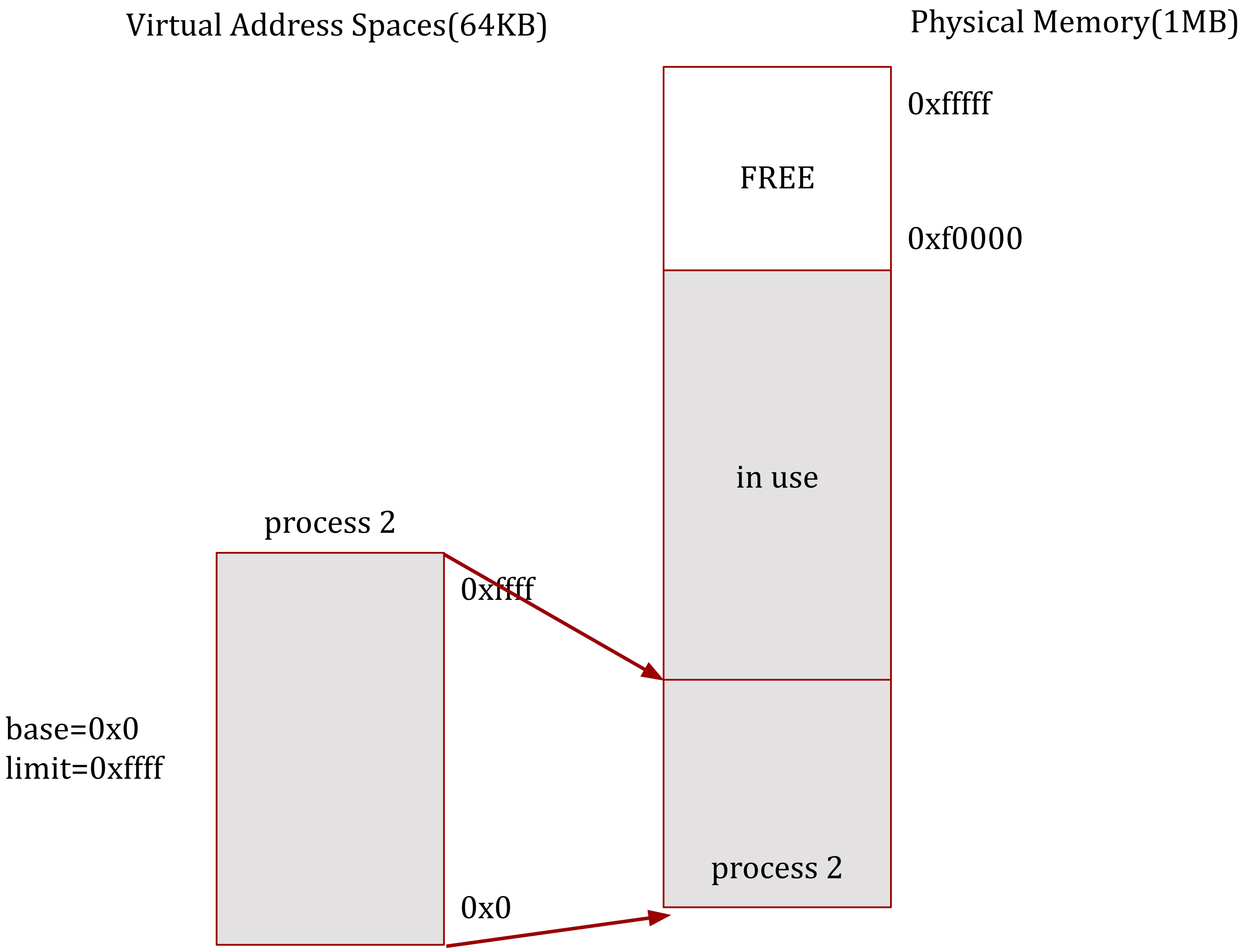
- Most of the time there is not enough memory to hold all the active processes at the same time
- Swapping is the process of saving a task to disk
- In swapping processes are loaded to and discarded from memory following the needs of execution
- Address must be relocated each time either by hardware or by software
- After a while memory can get fragmented and may need compaction, which is computationally expensive
- A process may also try to get bigger and bigger and bigger

Swap Process-1's entire physical memory out to file on disk



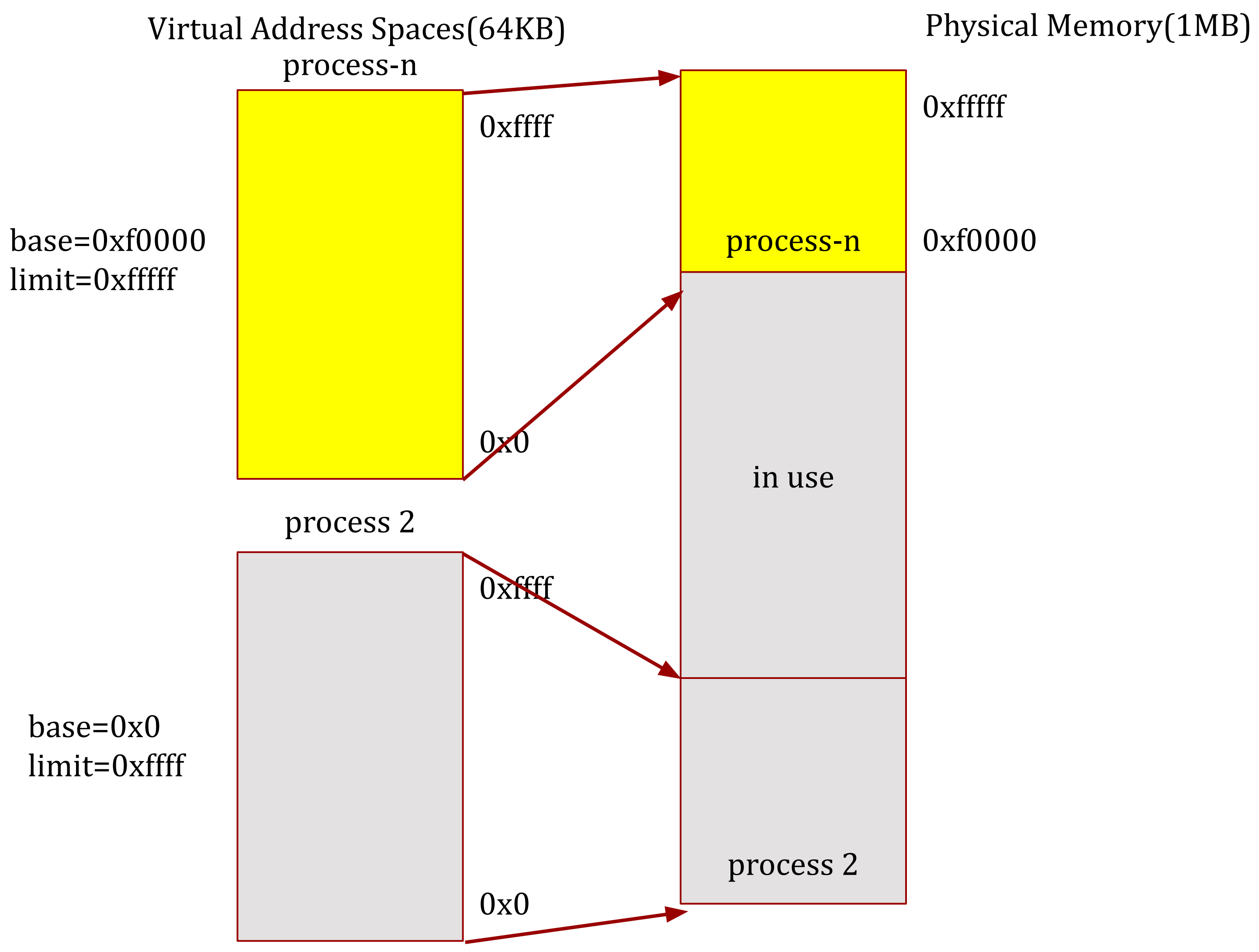
source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Process-1’s physical memory now FREE



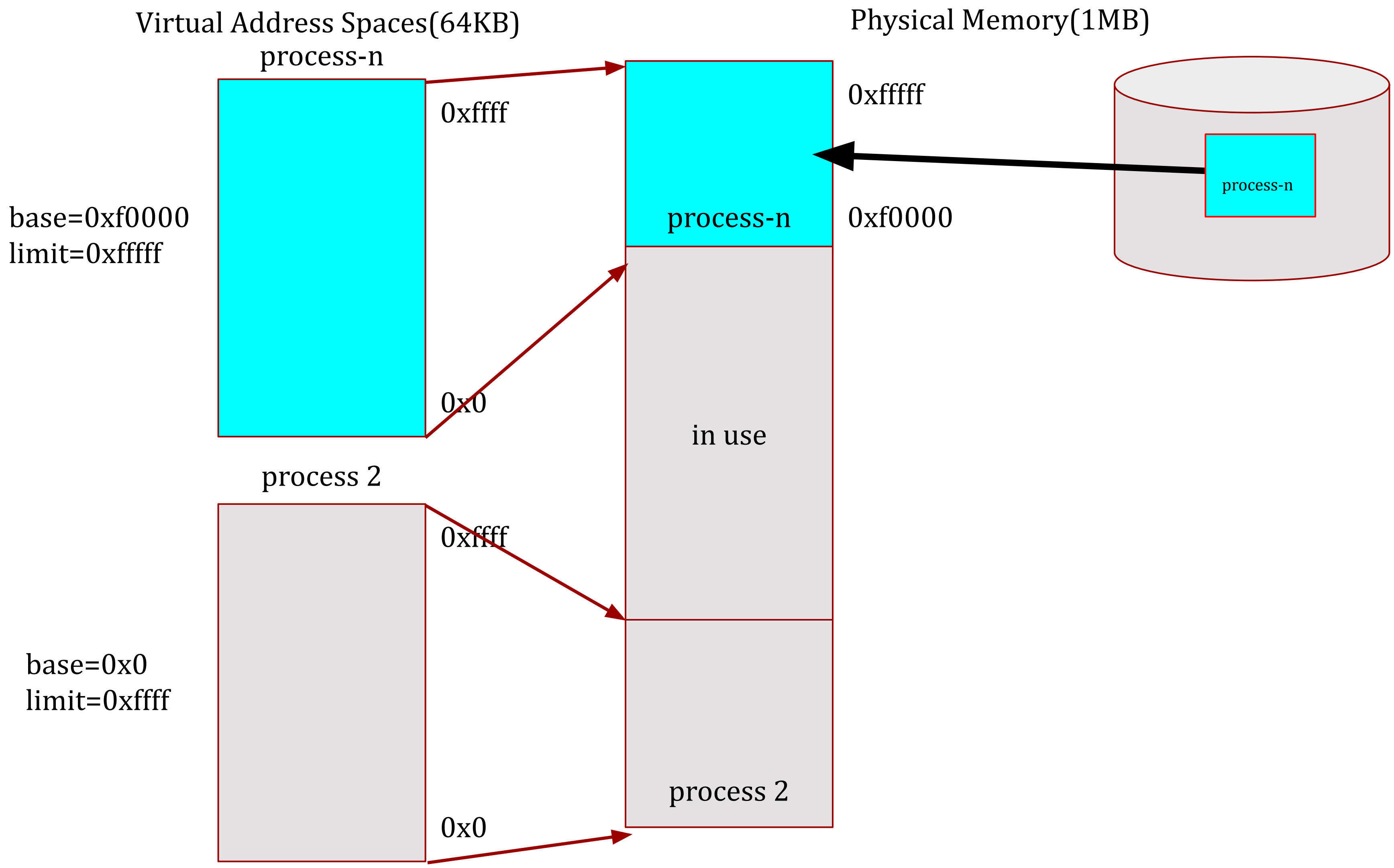
source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Create new virtual address for Process-n



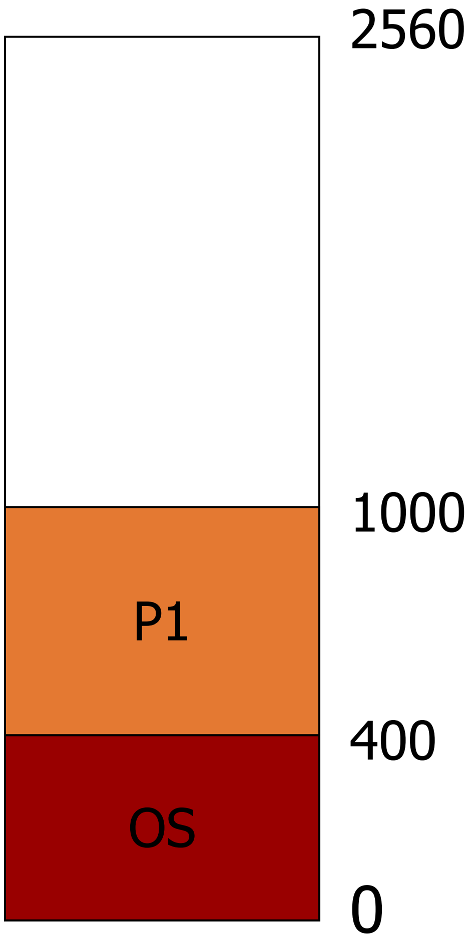
source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

Swap Process-n's entire physical memory in from disk



source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

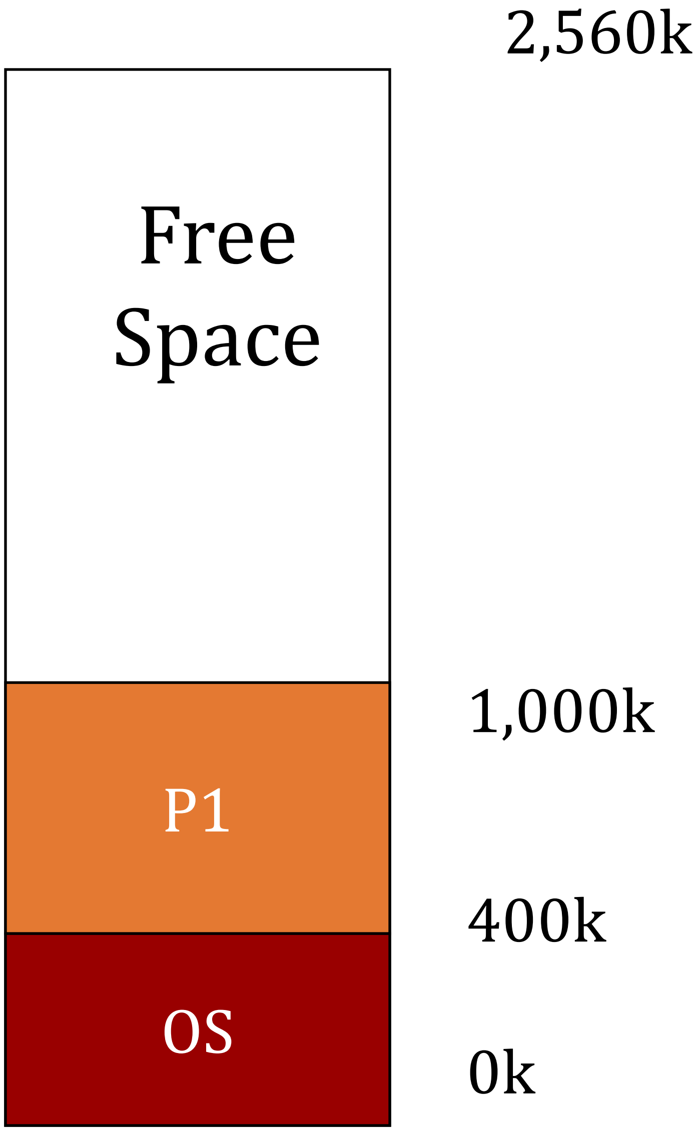
An Example



P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P2 can be loaded

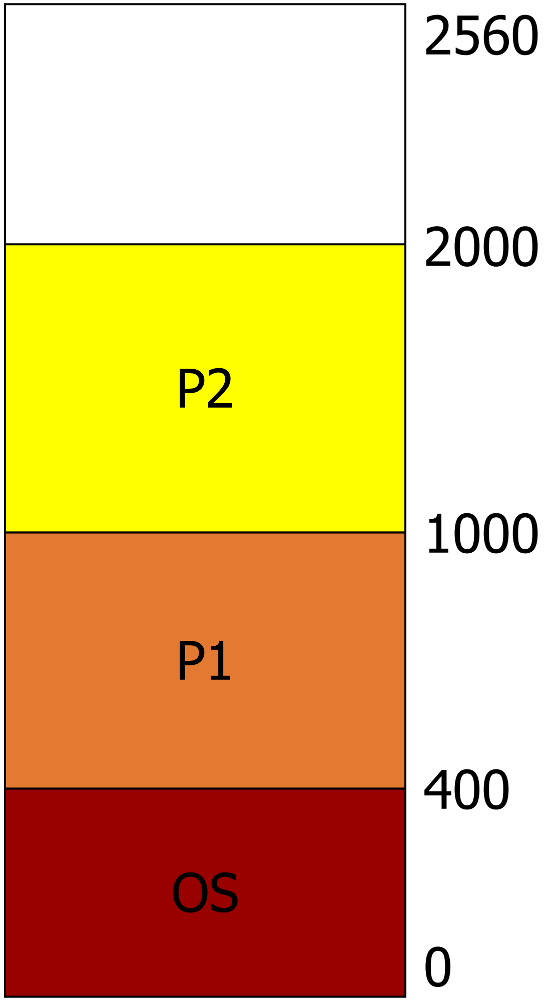
An Example



Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P2 can be loaded

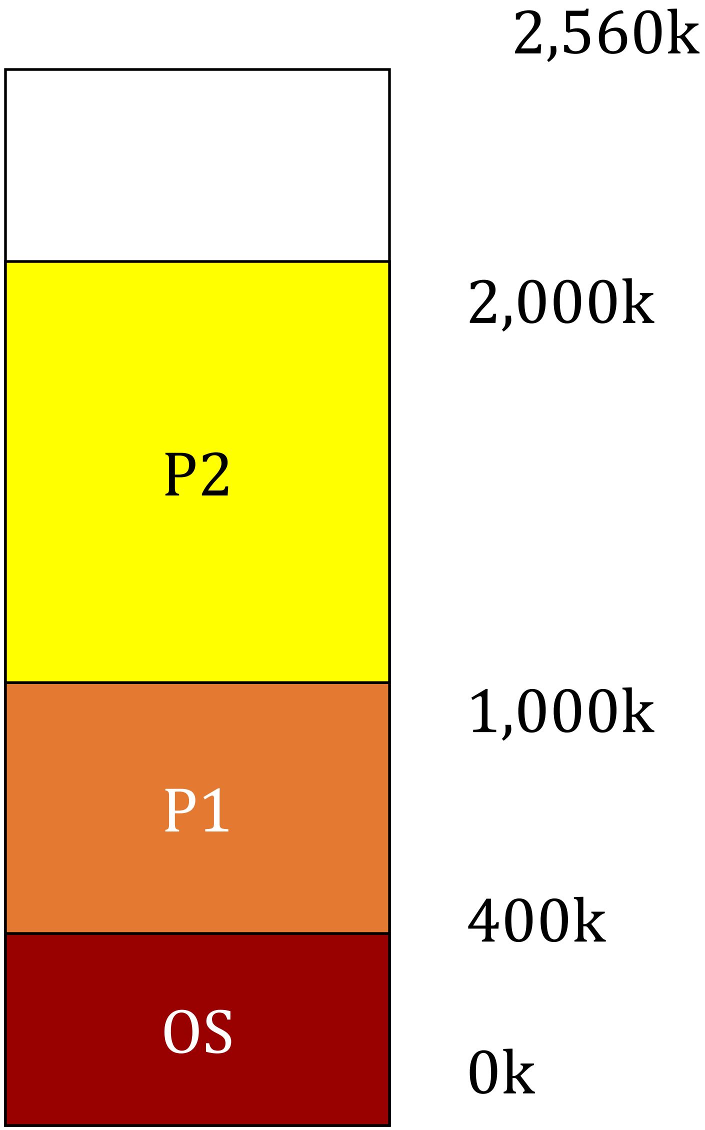
An Example



P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P3 can be loaded

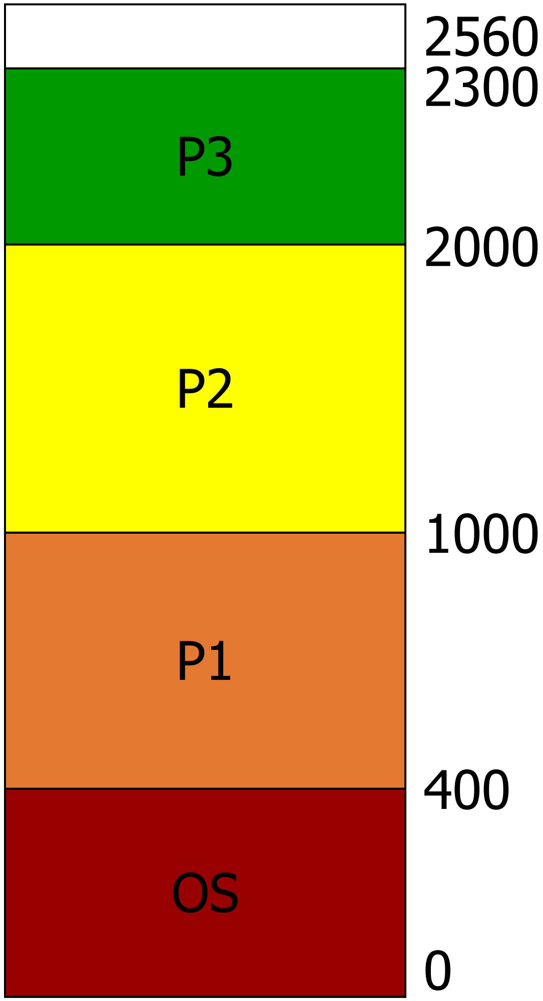
An Example



Process	Memory	Time
P1	600k	10s
P2	1,000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P3 can be loaded

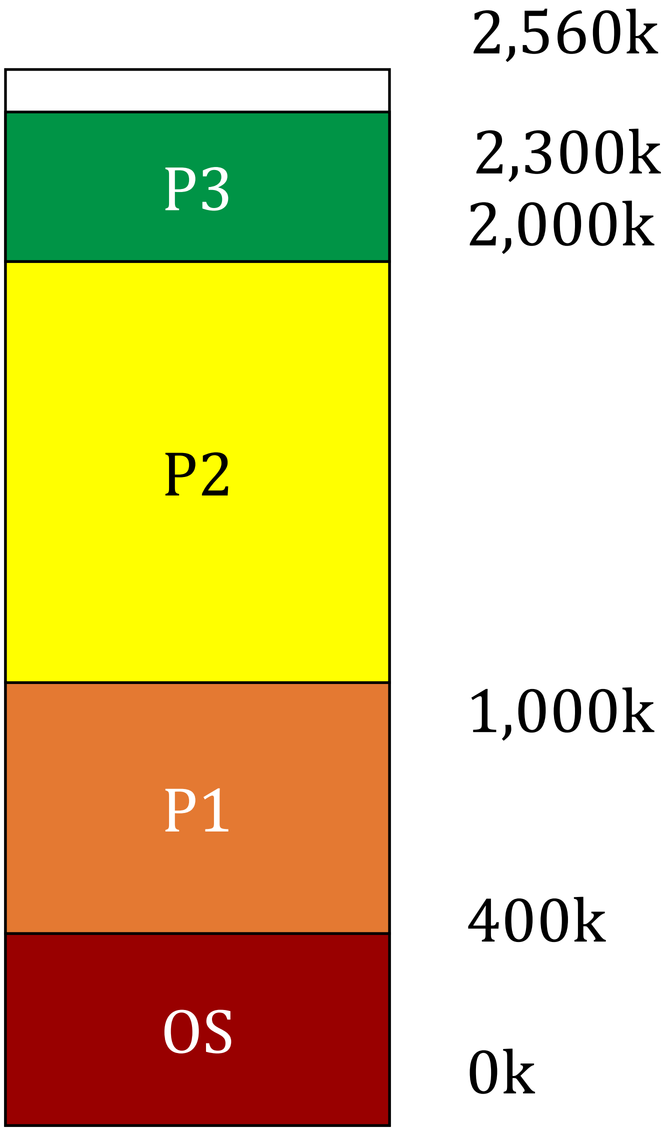
An Example



P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P4 & P5 need to wait!

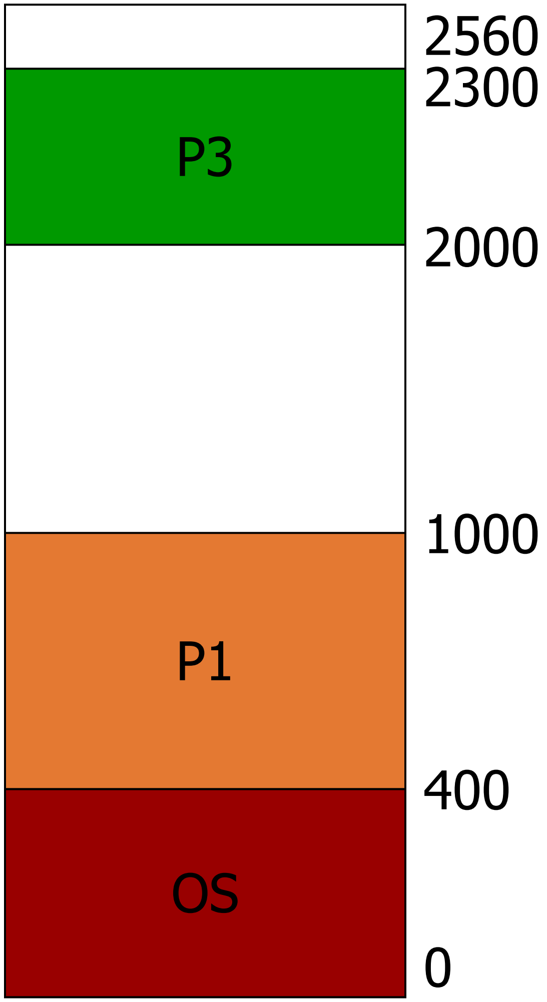
An Example



Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P4 & P5 must wait!

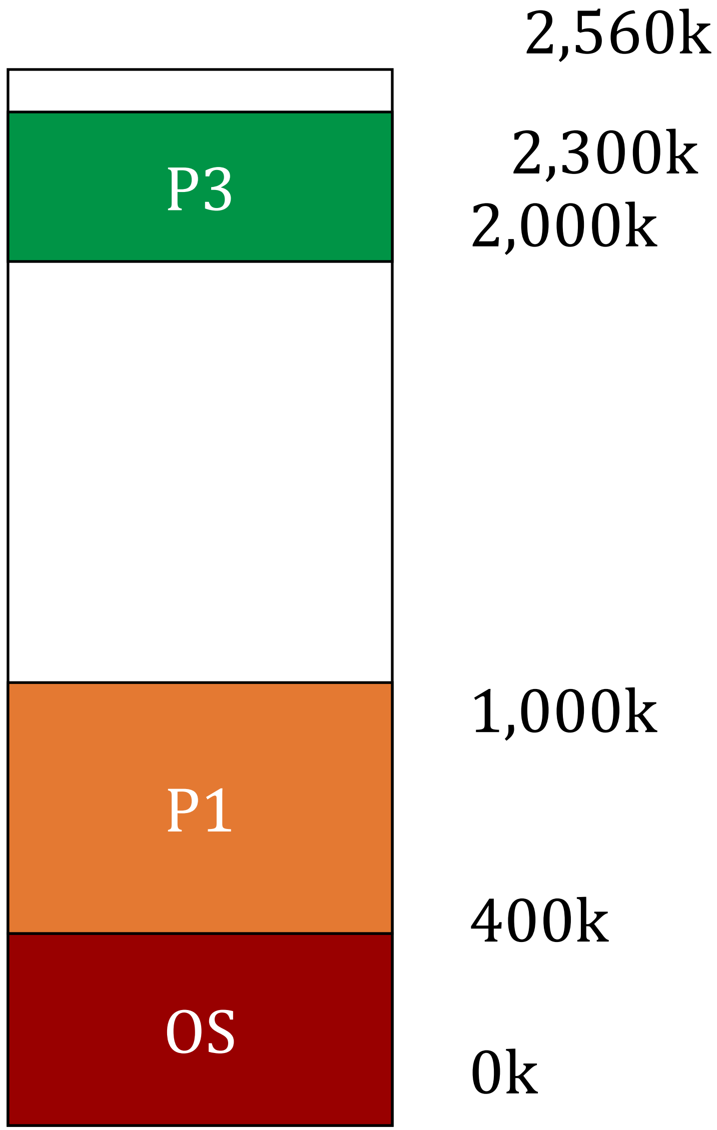
An Example



P1	600k	10s	
P2	1000k	5s	Terminates
P3	300k	10s	
P4	700k	8s	
P5	500k	15s	

P4 can be loaded

An Example

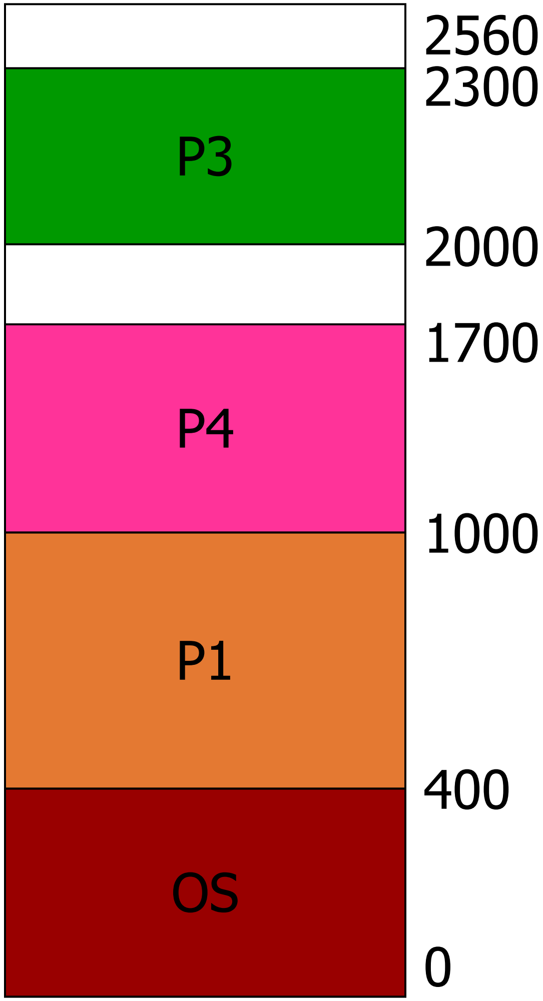


Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

terminates

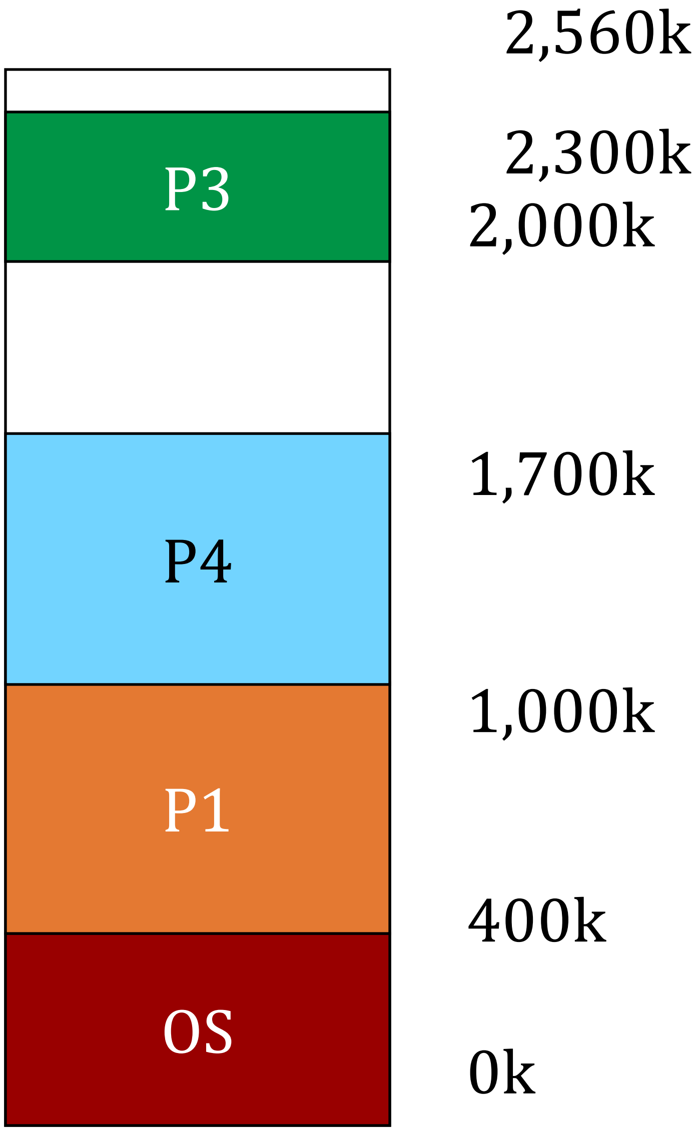
P4 can be loaded

An Example



P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

An Example



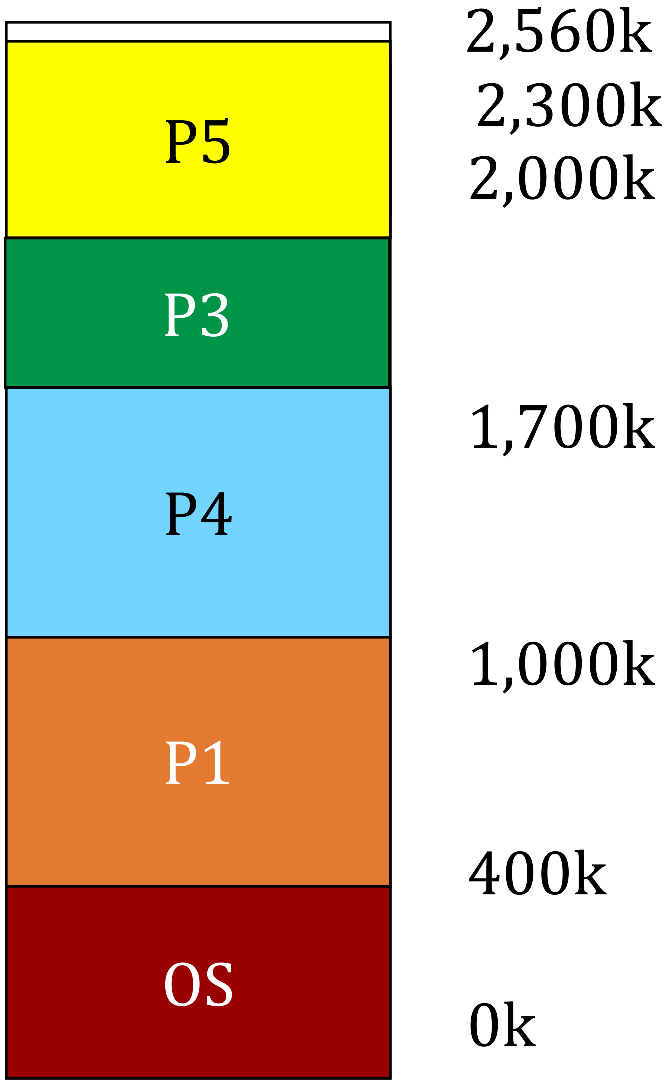
Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P5 cannot be loaded!
300k + 260k free,
500k needed

Memory Compaction

- As a consequence of swapping things in and out of memory, we might *fragment* memory
- This could prevent us from loading a program even though we technically have enough memory for it
- If necessary, we can shuffle things around so that we have one contiguous free space instead of multiple small "holes"
- But: it may be slow! E.g., if it takes us 100ns to read and then write 8 bytes of memory, ~107 seconds to move 8GB

An Example – Compaction



Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

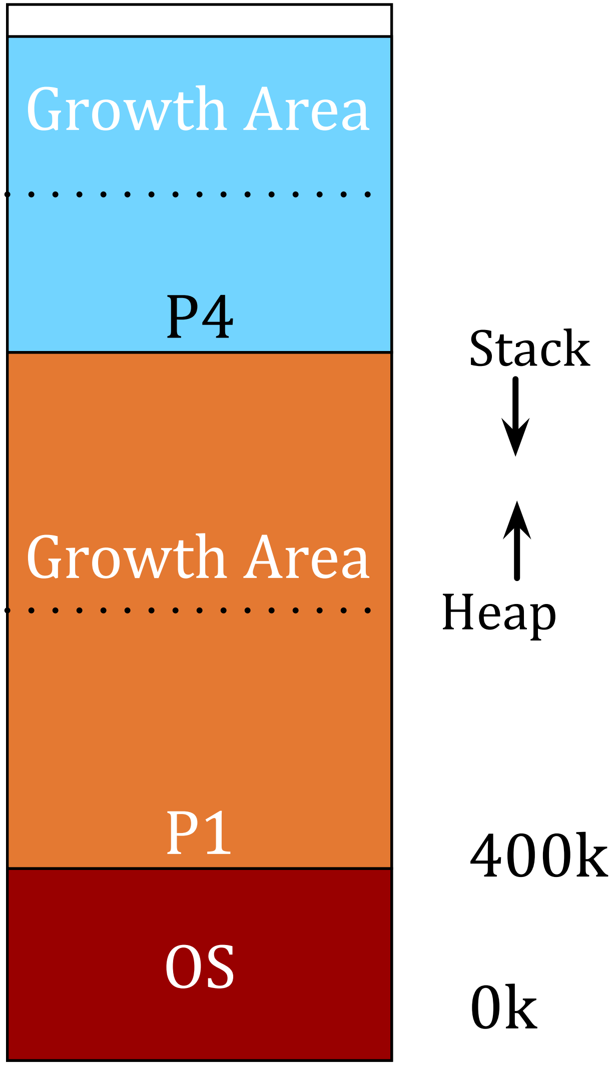
Growing Process Memory

- In general a process will not start off with all the memory it will ever need
  - Function calls will cause it to use more of the stack
  - Dynamically allocated data structures will need space too
- So in this case we will need to grow the memory space allocated to a process

Growing Process Memory

- If we allocate processes right next to each other, then we would have to move or swap them the first time the process grows
- Instead, it makes more sense to start each process with room to grow

Process Growth Area



Keeping Track of Memory

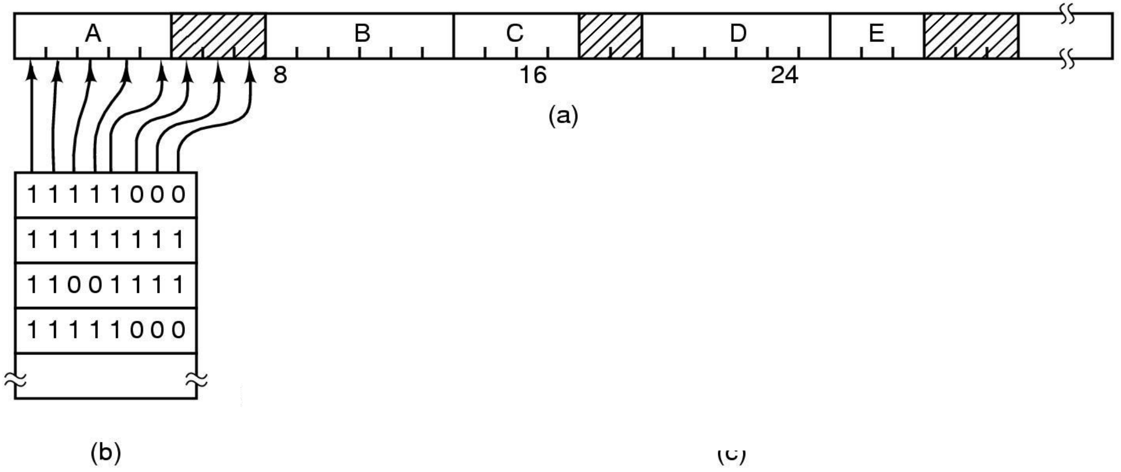
- To decide where to put programs, we need to know what memory is used/free
- This is a job for the OS – maintain a data structure that it can use to know what's available
- Two main structures used for this are *bitmaps* and *lists*

Memory Bitmap

- Basic idea – allocate memory in chunks of size N (the *allocation unit*)
- Store a sequence of bits where bit *i* says whether the *i*th chunk is free
- The allocation unit size is yet another balancing act:
  - Large unit sizes mean fewer bits are needed to describe memory, but may waste memory if process is not exact multiple of N

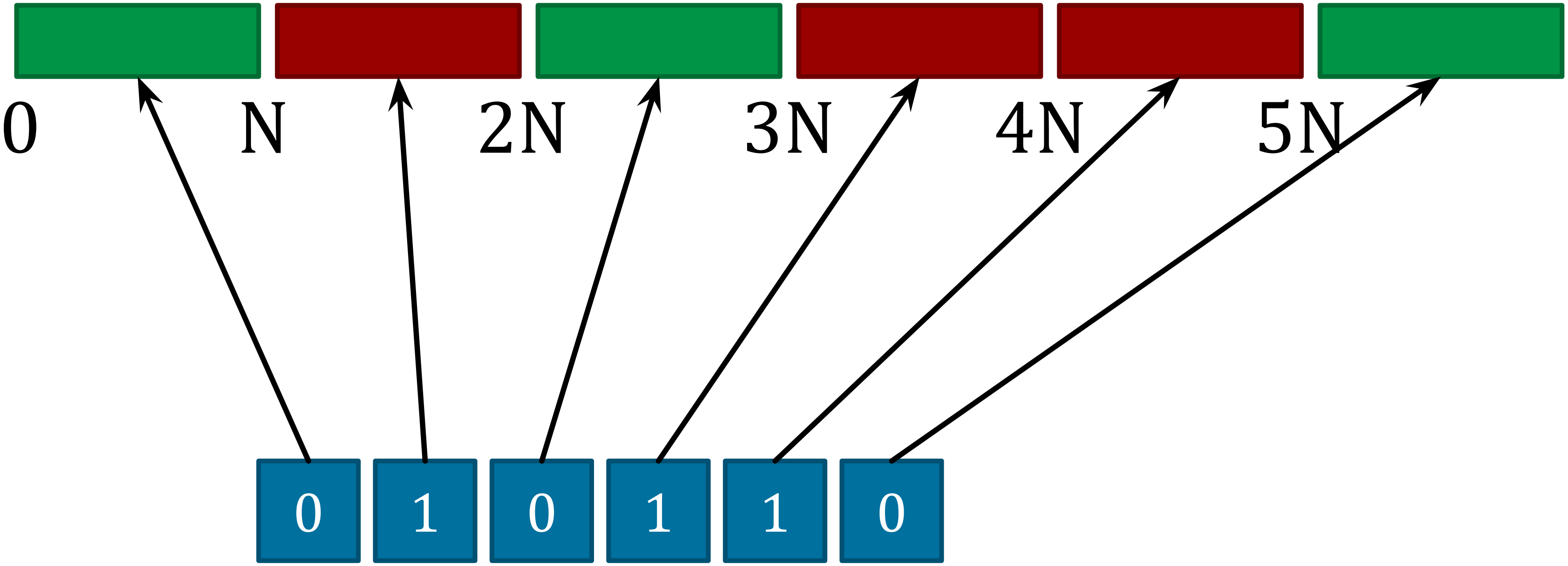
Memory Management with Bitmaps

- Divide memory in allocation units
- Keep track of which units have been used and which ones are free using a bitmap
- Tradeoff:
  - Big allocation units: +small bitmap -may waste memory
  - Small allocation units: -better “fit” +big bitmap



11111000
11111111
11001111
11111000

Memory Bitmap



Suppose N = 8 bytes
Then tracking free/used for 56 bytes
of memory takes only 6 bits
How about N = 4 bytes?

Allocating/Freeing Memory

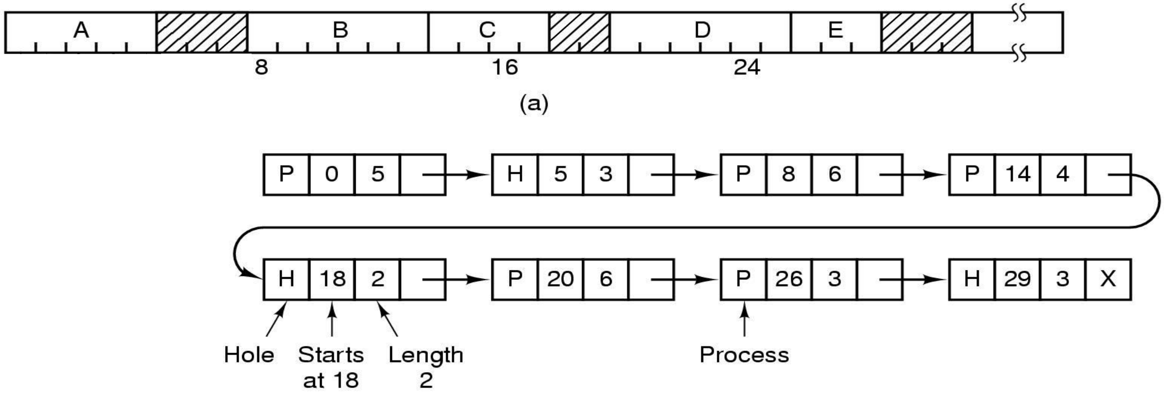
- To mark space as free, just set the corresponding bits to 0
- To find space for a new process K units long, we need to search for a consecutive string of K zeros
- This could be very slow, since most CPUs deal in units of multiple bytes, not bits, and the string of 0s could straddle a byte/word boundary

Memory Management with Linked Lists

Maintain a linked list where each element

- May represent a process (P) or a piece of free memory (“hole”, H)
- Contains number of initial unit
- Contains length of memory block
- Maintains pointer to each element (single- or double-linked)

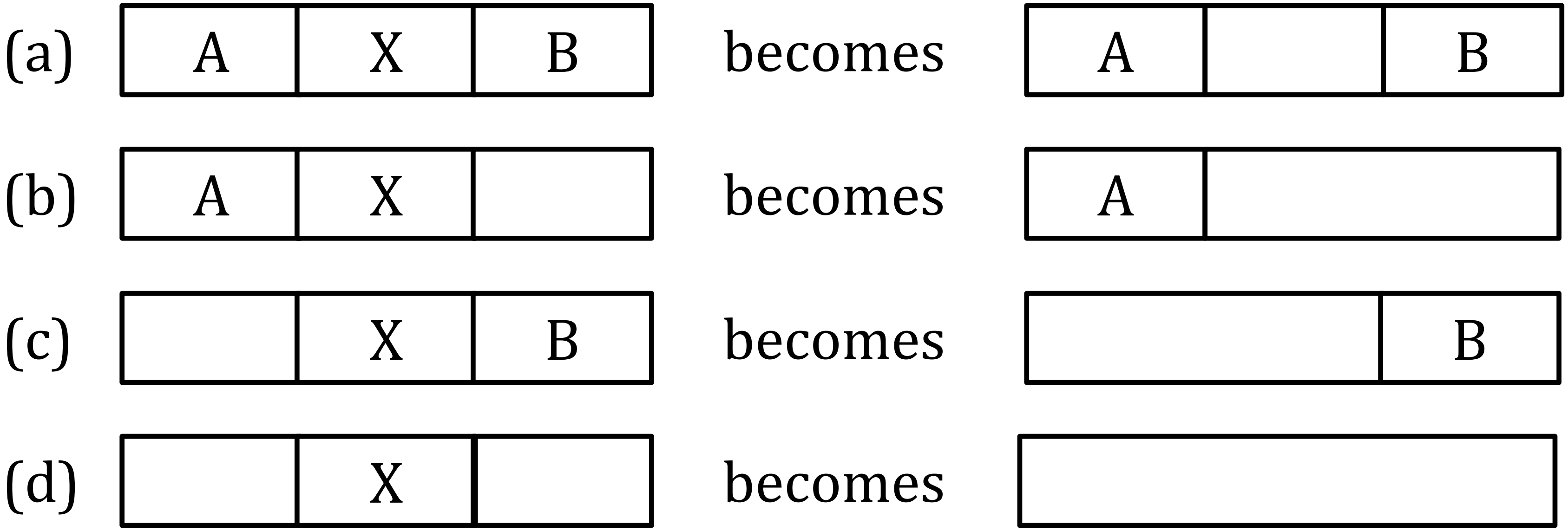
List is usually keep ordered



(a)

Freeing Memory

Before X terminates After X terminates



(a) A X B becomes A B

(b) A X becomes A

(c) X B becomes B

(d) X becomes

Finding Free Memory

Many strategies got find the right place to allocate a process that needs space:

First fit

- Search the list until a suitable hole is found
- Split the hole in a P and an H

Best fit

- Search the entire list and use the smallest hole that fits the program
- Slow (requires complete scan through the list)

Quick fit

- Separate lists, hashed by size or size ranges
- Speeds up search
- Makes compaction difficult

List Management – Optimizations

- Keep a separate freelist of just the unallocated regions
  - One nice trick is that we can actually store the list entries in the unallocated spaces themselves!
- Keep the lists sorted by address, so it's easier to merge free regions later
- Keep the lists sorted by size, so we don't have to search the entire list for the smallest

Modern operating systems use paging and virtual memory, but these techniques remain very relevant for heap management (`malloc`)

Relocation

The programmer cannot be sure where the program will be loaded in memory

1. Address locations of variables, code routines cannot be absolute
2. Must keep a program out of other processes' partitions

Relocation can be done at load time

- Maintain a list of all places in the program where absolute addresses were used (relocations)
- At load time, simply add an offset to all absolute memory references
- Introduced by the IBM 360 in 1965. Why would this still be relevant?
  - Shared libraries used by a program may have to be statically relocated before they are loaded