Chapter 8

Memory Management

Basic Management Techniques

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

May 20/27, 2020

8.1

Purpose and Contents

The purpose of this chapter

1. presents basic role and functionality of a memory manager
2. describes different memory management techniques
    - contiguous allocation, swapping, paging, segmentation

8.2

Bibliography

- Andrew Tannenbaum, *Modern Operating Systems*, second edition, Prentice Hall, 2001, pg. 190 – 263.

8.3

Contents

8.4

1 Context and Definitions

Context

- program's execution phases
    - user writes the program
    - compiler (assembler) and linker generates the executable
    - OS loads the executable in memory ⇒ process
    - process gets executed by the CPU

  - on-chip MMU (memory management unit) accesses the memory
- main memory (RAM) and registers are
  - the only storage directly accessible by CPU (i.e. MMU)
  - though, some levels of cache also, but we do not talk about this
- if some needed data (e.g. instructions, operands) is not in memory
  - it must be moved there before the CPU can operate on it

8.5

Memory Needs

- ideally, we would like the memory to be
  - very large
  - very fast
  - non-volatile
- in reality, there is a hierarchy of different types of memory (speed vs. capacity vs. cost)
  - register access is faster than memory access
    - while register access lasts one CPU cycle, a memory access may need more CPU cycles
  - faster memory, named *cache*, is used between registers and main memory
  - slower memory (storage area) is used for higher capacity and non-volatility needs
- memory protection must be provided
  - OS and processes must be protected from one another

8.6

Memory Manager

- an OS component
- keeps track of which parts of memory are in use and which are not
- allocates and releases areas of main memory to processes
- influences how each process sees the system's memory
  - based on hardware support
- coordinates how the different types of memory are used
  - moves processes between different memory layers (i.e. main memory and disk), to improve process performance
  - invalidates cache entries for performance and security reasons
- provides memory protection
  - based on hardware support

8.7

Basic Memory Configuration

- only one user program (process) in memory at one moment
  - usually at a known, established location
- the OS also resides in memory
  - so, actually we have two processes in memory
- disadvantages
  - a limited configuration
  - supporting multiprogramming is difficult

8.8

Basic Memory Configuration. Illustration

8.9

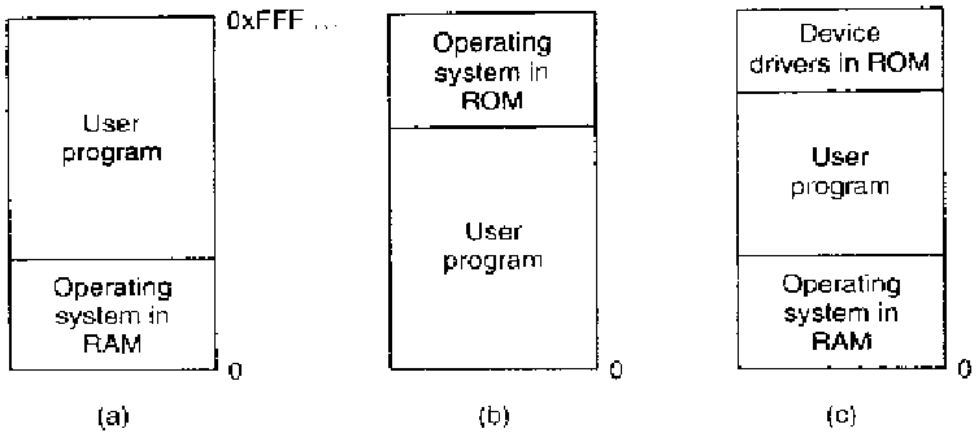


Figure 1: Tanenbaum, "Modern OS", Figure 4.1

A More Realistic Configuration

- **multiprogramming** systems
  - more processes loaded simultaneously in memory
- improves CPU (and other resources) utilization
- ⇒ **different processes loaded in different places in memory**, i.e. at different memory addresses

8.10

A More Realistic Configuration. Illustration

8.11

A More Realistic Configuration. Problems

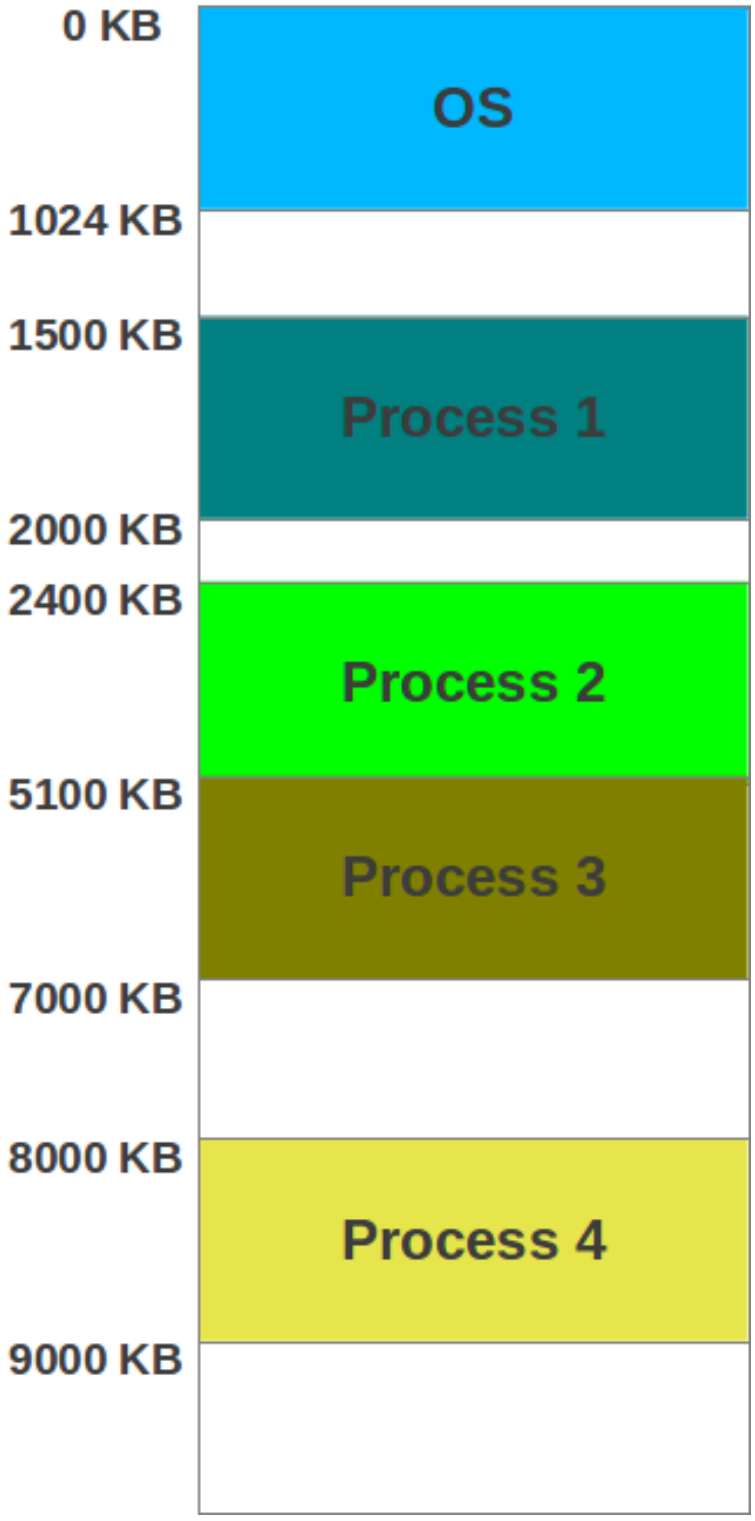
- memory addresses one program / process uses cannot be the same like those of other process
- when one program's executable is generated, the compiler
  - mono-programming case
    - knows where that program would be loaded in memory
    - knows which are the memory addresses for that program
  - multi-programming case
    - does not know where that program will be loaded in memory
    - does not know which are the memory addresses for that program

8.12

C Source Code Example (`memory-layout.c`)

```
int x;

int inc(int v)
{
  int aux;

  aux = v;
  aux++;

  return aux;
}


int main(int argc, char **argv)
{
  x = 0;

  x = inc(x);

  return x;
}
```

8.13


0 KB
OS
1024 KB
1500 KB
Process 1
2000 KB
2400 KB
Process 2
5100 KB
Process 3
7000 KB
8000 KB
Process 4
9000 KB

Compilation and Executable Displaying Commands

```
# Compile and generate executable
#    "-m32" for x86 (32-bit) architecture
#    "-g"   compile for debugging,
#           including symbol and source code references
#    "-fno-stack-protector"
#           do not include code for stack protection
gcc -Wall -Werror -g -m32 -fno-stack-protector \
    memory-layout.c -o memory-layout

# Display executable's contents
#    "-d"   disassemble code
#    "-S"   show code source intermixed with disassembly
#    "-j .text"
#           for the code section
#    "--disassembler-options=intel"
#           usig Intel syntax
objdump --disassembler-options=intel -d -S \
        -j .text memory-layout
```

8.14

The Resulted Executable (`memory-layout`)

```
080483db <inc>:
#include <stdio.h>

int x;

int inc(int v)
{
 80483db:   55                      push   ebp
 80483dc:   89 e5                   mov    ebp,esp
 80483de:   83 ec 10                sub    esp,0x10
  int aux;

  aux = v;
 80483e1:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 80483e4:   89 45 fc                mov    DWORD PTR [ebp-0x4],eax
  aux++;
 80483e7:   83 45 fc 01             add    DWORD PTR [ebp-0x4],0x1

  return aux;
 80483eb:   8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
}
 80483ee:   c9                      leave
 80483ef:   c3                      ret


080483f0 <main>:


int main(int argc, char **argv)
{
 80483f0:   55                      push   ebp
 80483f1:   89 e5                   mov    ebp,esp
  x = 0;
 80483f3:   c7 05 1c a0 04 08 00    mov    DWORD PTR ds:0x804a01c,0x0
 80483fa:   00 00 00

  x = inc(x);
 80483fd:   a1 1c a0 04 08          mov    eax,ds:0x804a01c
 8048402:   50                      push   eax
 8048403:   e8 d3 ff ff ff          call   80483db <inc>
 8048408:   83 c4 04                add    esp,0x4
 804840b:   a3 1c a0 04 08          mov    ds:0x804a01c,eax

  return x;
 8048410:   a1 1c a0 04 08          mov    eax,ds:0x804a01c
}
 8048415:   c9                      leave
 8048416:   c3                      ret
```

8.15

Questions

- **Question 1**: Who is responsible for establishing / specifying memory location a process will be loaded at, i.e. memory addresses that process will use?
- due to uniformity and transparency reasons, we could have
  - the compiler / process presumes (sees) one kind of memory layout
    - e.g. supposes the process will be loaded at the beginning of the memory, i.e. memory address 0 (zero)
  - the operating system and the MMU sees another kind of memory layout
- $\Rightarrow$ there could be
  - logical (virtual) memory layout $\Rightarrow$ **logical (virtual) memory addresses**
  - real (physical) memory layout $\Rightarrow$ **real (physical) memory addresses**
- **Question 2**: Who is responsible for translating logical addresses to physical addresses?

8.16

Address Binding

- binding = the way different types of addresses are related
- compile time
  - if starting memory address the program will be loaded at execution is known, than *absolute code* can be generated
  - if not (common case), *relocatable code* must be generated
- load time
  - binding is done at load time
  - if starting address changes, the program must be reloaded
- execution time (runtime)
  - if process can be moved during its execution, binding must be delayed until runtime
  - hardware support is needed for such a scheme to work

8.17

Practice (1)

1. Having the following partial contents of an executable file (i.e. an instruction accessing a memory location), illustrate, for the following two cases, the corresponding contents (i.e. how the instruction looks like) of the physical memory allocated to the process executing the given code (instruction), considering that process to be loaded in memory starting from address `0x1000000`:
   (a) load-time binding;
   (b) runtime binding.

```
mov     DWORD PTR ds:0x804a018,0x0
```

8.18

Logical (Virtual) vs. Physical Address Space

- **address space** = the set of all memory addresses accessible by a process
  - **virtual address space (VAS)**
  - **physical address space (PAS)**
- compiler generates virtual (logical) addresses (VAs) $\rightarrow$ VAS
- CPU, at runtime, tries accessing VAs $\rightarrow$ VAS
- MMU accesses physical addresses (PAs) $\rightarrow$ PAS

1. the compile-time binding method generates identical VAS and PAS
   - $\Rightarrow$ no need for VA to PA translation
2. the load-time binding method generates different VAS and PAS
   - $\Rightarrow$ VA to PA translation done by OS loader
3. the execution-time binding method generates different VAS and PAS
   - $\Rightarrow$ VA to PA translation done by MMU

8.19

General Layout of a Process' VAS

8.20

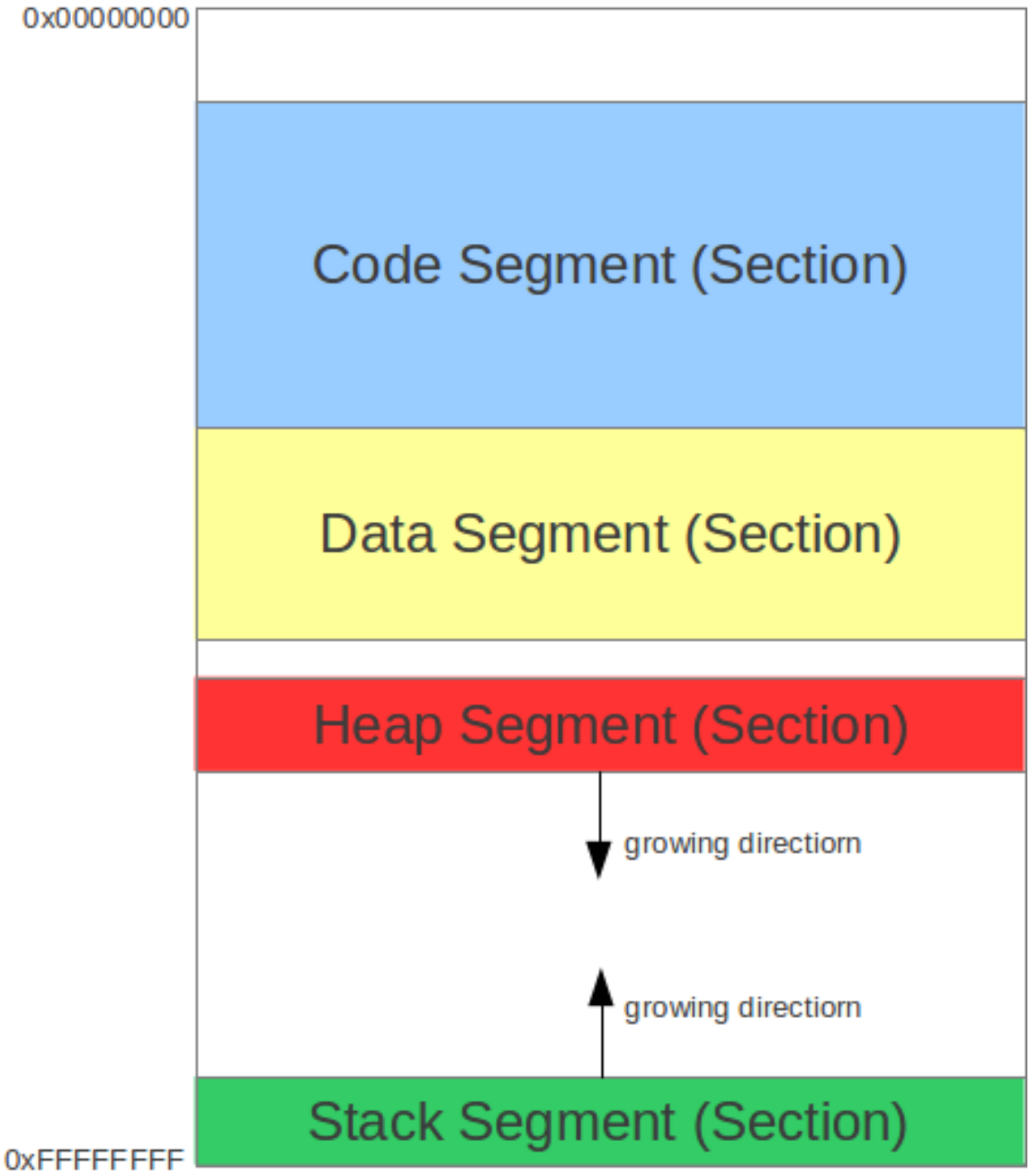
2 Basic Memory Management Techniques

2.1 Contiguous Allocation

General Rules

- each process is allocated a single contiguous area
- OS and processes must be allocated different areas
- usually the OS is mapped in low memory and is resident

8.21


0x00000000
Code Segment (Section)
Data Segment (Section)
Heap Segment (Section)
growing directiorn
growing directiorn
Stack Segment (Section)
0xFFFFFFFF

Memory Mapping (Binding) and Protection

- could be done dynamically based on the base and limit values
  - it is a hardware support
  - there are two registers: base and limit
  - each memory address is compared to the limit and added the base register
  - each process is associated different base and limit values, depending on where the process is allocated memory and its size
  - the two registers could be set only by the OS
- supports moving of processes in memory due to OS or processes variation
- problem: does not support mapping of virtual address space with holes

8.22

Allocation Strategy

- multiple fixed-size (not necessarily equal) partitions
  - two allocation strategies
    - a different waiting queue for each partition
    - one global queue for all partitions
  - the number of concurrent processes depends on the number of partitions
  - too rigid
- multiple variable-size partitions
  - each process is allocated only the space it needs
  - the system keeps track of available partitions
    - they could be ordered in some way to help finding the good one
    - they might be merged, when released

8.23

Allocation Strategy (cont.)

- which free partition to allocate?
  - *first-fit*, *next-fit*, *best-fit*, *worst-fit*
- what happens when not enough memory available?
  - another process from input queue is allocated memory
  - all memory allocations are delayed until enough memory available for the next process in input queue

8.24

Fragmentation

- external
  - free space is divided in small areas
  - solutions: compaction or allocation of processes in more areas
  - compaction is possible only in case of dynamic relocation
- internal
  - in practice it is too expensive to keep track of small areas (few bytes) $\Rightarrow$ allocate memory in terms of blocks

8.25

Practice (2)

2. Supposing an OS uses contiguous allocation strategy for memory management and processes `P1`, `P2`, and `P3` was allocated contiguous memory areas starting at `0x1000000`, `0x3000000`, `0x6000000`, respectively, illustrate the evolution of the base register's value when processes are scheduled in the following order: `P2`, `P1`, `P2`, `P3`.

8.26

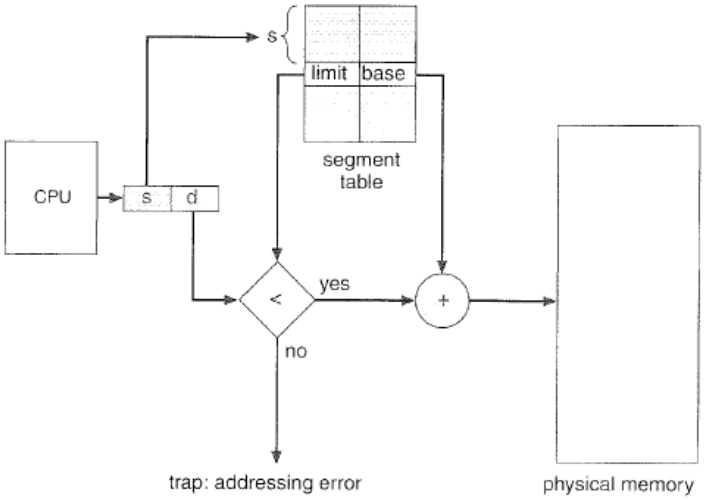


Figure 2: Silberschatz, p. 304

2.2 Segmentation

Method's Principles

- different user view of a process (program)
  - not a single contiguous area of bytes, but
  - more different areas (segments) of variable sizes
  - separated by unused areas, i.e. not a contiguous view of the address space
  - data are referred to using their offset in the corresponding segment
- logical address space is a collection of segments
  - each segment has a name (identifier) and a length
  - a logical address consists of a segment identifier and an offset
  - each segment could contain a particular (different) types of data
  - examples: code, global variables, heap, stack, standard C library, other libraries

8.27

Address Binding (Translation)

- segment table
  - a generalization of the base and limit registers used for contiguous allocation
- each segment is allocated an entry in the segment table
  - the entry contains the base and limit values for that segment
- segment table usually in memory
  - not so many CPU registers
  - CPU contains only the address and size of the segment table

8.28

Hardware Illustration

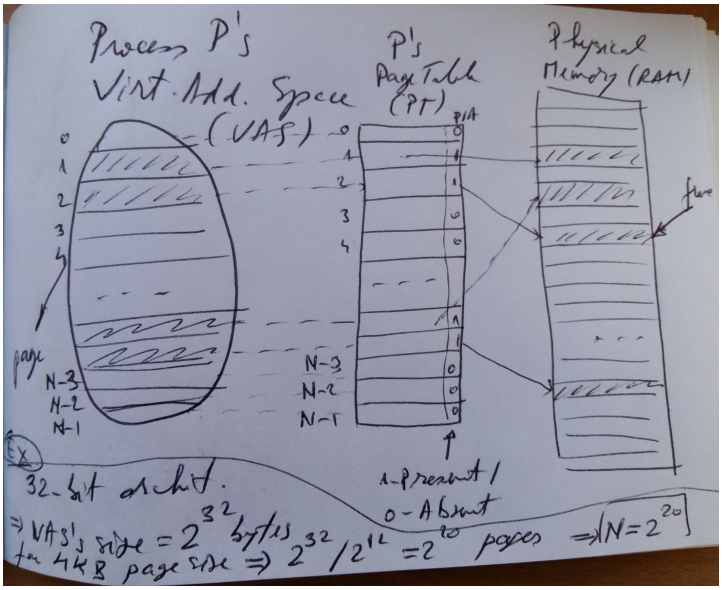
8.29

2.3 Paging

Definition

- divide memory (and swap area) in fixed-size blocks
- allocation is done in terms of those fixed-size blocks
  - physical memory is divided in *frames*
  - logical memory is divided in *pages*
- page size is defined by hardware
  - typically a power of 2

8.30



Method's Principles

- allows allocation in non-contiguous areas
  - all pages / frames of the same size
  - $\Rightarrow$ any available page (free area) could be allocated
- avoids external fragmentation
- though suffers by internal fragmentation
  - not all allocated space (in a page) is used
  - the unused ("free"), yet allocated, space is lost
  - reduce it: decrease page size $\Rightarrow$ overhead for page table and I/O operations
- common page sizes: $4KB - 8KB$

8.31

Address Binding (Translation)

- paging is "similar" to segmentation $\rightarrow$ each page a segment
- $\Rightarrow$ page table = a collection of base and limit registers
  - one pair (entry) for each page $\Rightarrow$ page table (PT)
  - though the limit is not needed as all pages have the same size
- difference by segmentation
  - there is an entry in the PT for each virtual page in the process
  - even for pages in unused areas
  - though, such pages are marked as unmapped (invalid)
  - paging is orthogonal to segmentation

8.32

Mapping One Virtual Address Space With Paging

8.33

Mapping Two Virtual Address Spaces With Paging

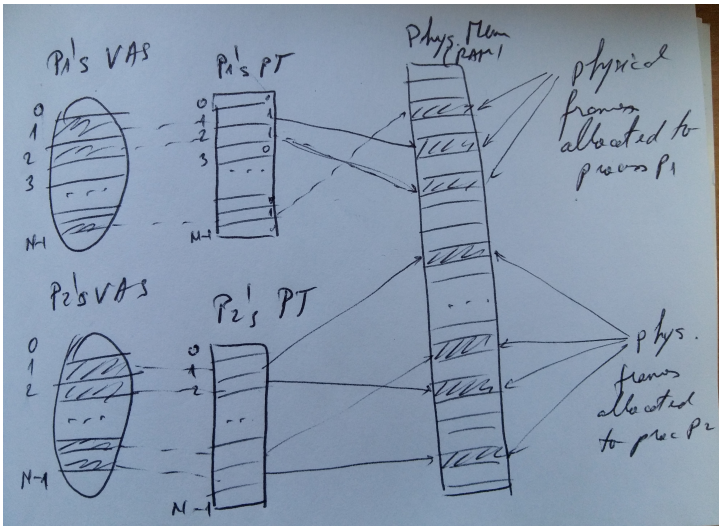
8.34

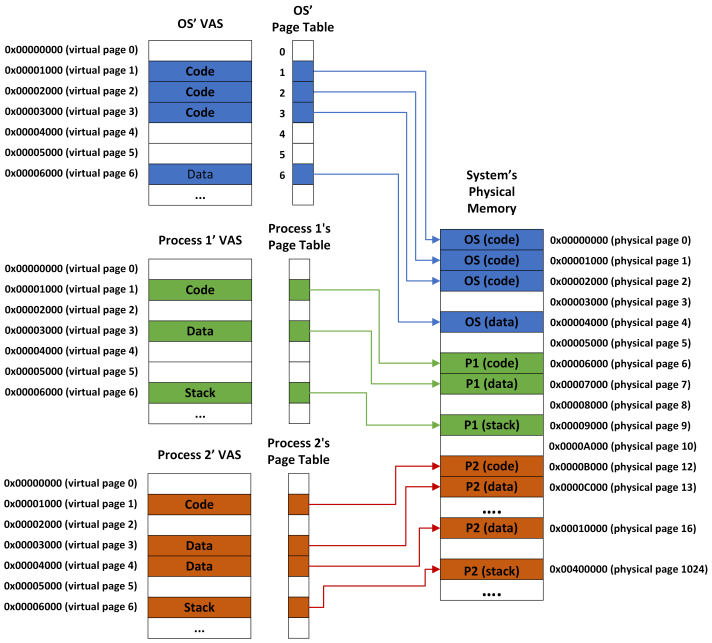
Paging and Page Tables Illustration

8.35

Address Translation During Runtime

8.36





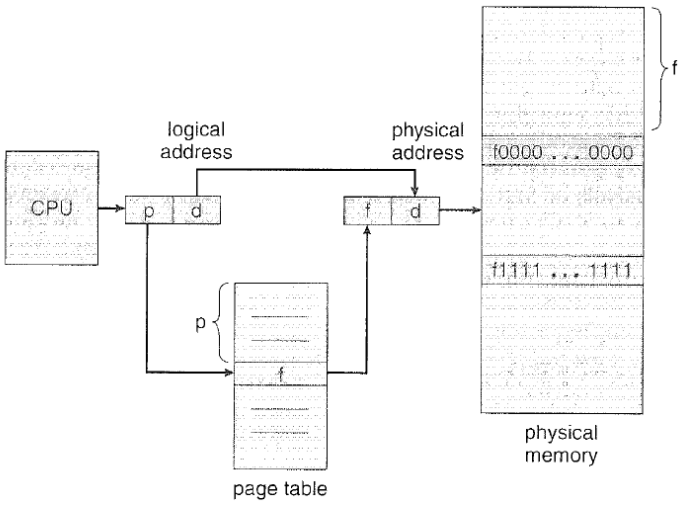


Figure 3: Silberschatz, p. 288

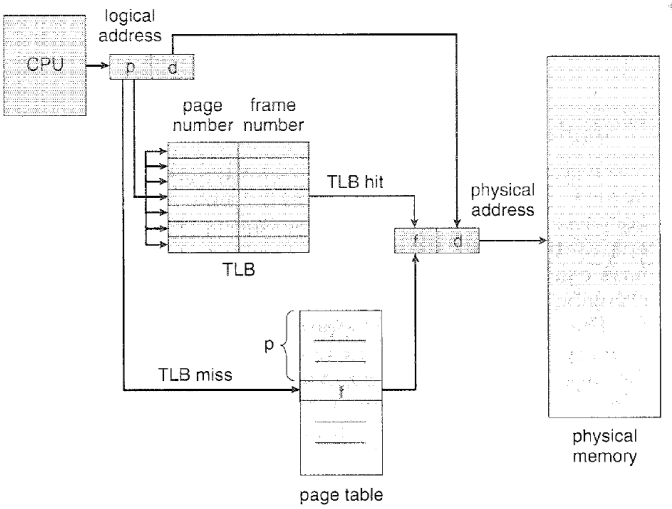


Figure 4: Silberschatz, p. 294

Practice (3)

3. Which is the size in bytes of a process' virtual address space, in case of a system using 32 bits for virtual memory addresses? What about a 64-bit system?
4. Which is the number of entries in the page table for both cases mentioned above?

8.37

Hardware Support for Paging

- registers to store page table entries
  - advantage: fast
  - disadvantage: appropriate for small tables
- page-table base register (PTBR)
  - page table kept in memory
  - advantage: appropriate for large page tables
  - disadvantage: slower, due to additional memory accesses

8.38

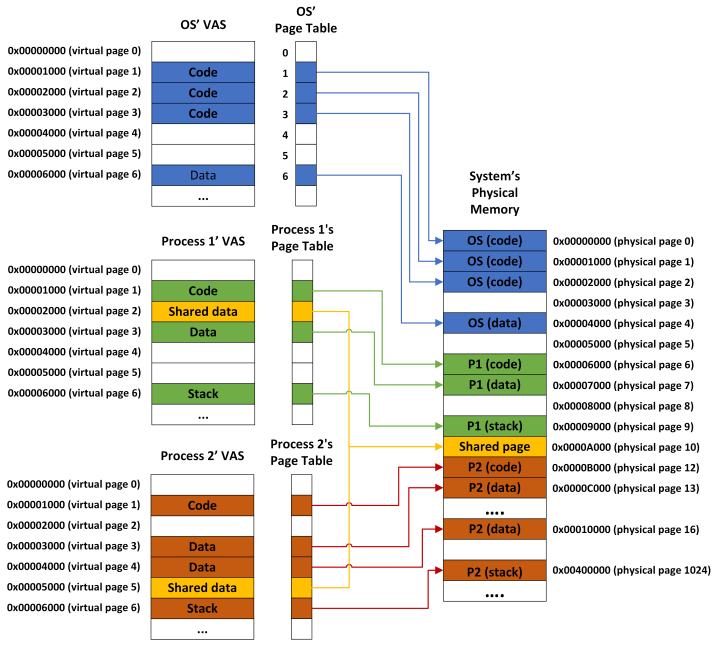
Hardware Support for Paging (cont.)

- translation look-aside buffer (TLB)
  - associative, **high-speed memory**
  - a **TLB entry** consists of a **key (VA)** and a **value (PA)**
  - contains some (most used) page table entries (PTEs)
  - advantage: a key (VA) is searched very fast (in parallel on all entries)
  - disadvantage: expensive and small (typically 64 – 1024 entries) $\Rightarrow$ **TLB miss**
  - replacement algorithms must be used
  - some entries could be *locked* (typically the OS's entries)
  - a TLB entry could also contain an address-space identifier (ASID) to be able to store page of different processes simultaneously
  - when not having ASID, the TLB must be flushed at each context (process) switch

8.39

Paging Mechanism in Hardware with TLB

8.40



Protection

- based on some bits (flags) associated to each page and supported by the hardware
  - read (R), write (W), execute (X)
  - 1 $\rightarrow$ allows the operation
  - 0 $\rightarrow$ disallow the operation
- valid / present bit
  - 1 $\rightarrow$ legal / present page
  - 0 $\rightarrow$ illegal / non-present page

8.41

Shared Pages

- same physical pages (frames) shared between different processes
- the shared memory could be mapped on different areas in each process' VAS
- easy to be done using page tables
- useful especially for shared reentrant code
  - the read-only nature of shared code should not be let to the correctness of code
  - the OS should enforce this property
- used provide memory-based inter-process communication mechanisms

8.42

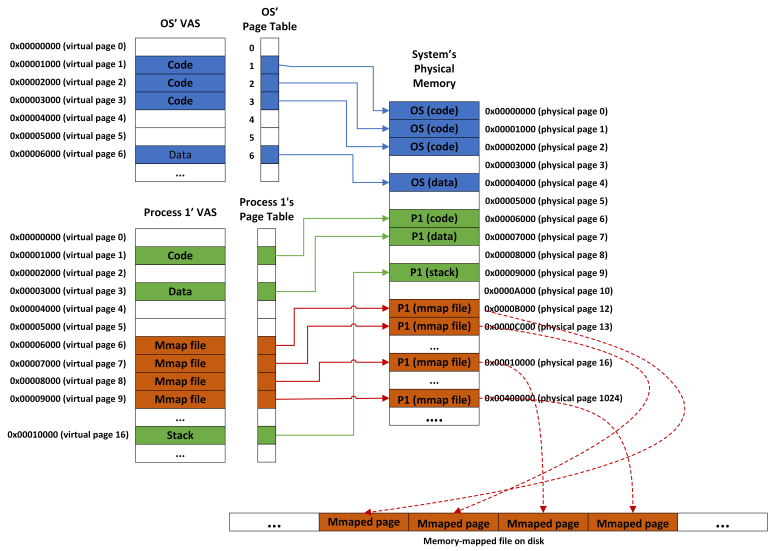
Shared Memory Illustration

8.43

Memory Mapped Files

- map parts of a file into a process' virtual address space
- provide a memory-like interface to a file contents
- advantage: transparency, performance
- when used with memory lazy-loading mechanisms
  - only accessed parts of the file are loaded in memory (transparently done by OS)
  - advantage: efficiently manage huge files
- limitations
  - file could not be extended

8.44



Memory-Mapped File Illustration

8.45

Practice (4)

4. You are given the following C code of a process running on a system that uses paging with pages of 4KB in size. You have to illustrate on the process's page table the way the allocated memory referred to by the "*p*" pointer could be mapped in the physical memory, using non-consecutive physical memory frames (pages).

```
1 char *p;
2 p = (char*) malloc (4097 * size(char));
3 printf("p = %p\n", p);    // displays p = 10240
```

5. What will be the effect of the following instructions, executed after the ones given above.

```
1 *p = 0;
2 p[4096] = 10;
3 p[4100] = 20;
4 p[2*4096+1] = 30;
```

6. How much space is lost due to internal fragmentation?
7. How much space is lost due to external fragmentation?

8.46

2.4 Swapping

Definition

- method to extend the main memory
- swap out (to some backing store) to make space for other process
- swap in (back in memory) to be available for execution
- there could be different strategies for swap out/in depending on the scheduling algorithm
  - round-robin
  - priority based
- swapping granularity
  - *hole process level*
    - it is too slow to swap out/swap in an entire process at CPU switch
  - *parts-of-process level*
    - more feasible in practice
    - besides, running a process does not require the entire process to be in memory, but only the parts needed for current instruction

8.47

Restrictions

- binding method
  - if binding is done at compile-time or load-time, swap (back) in must be done at the same position (address) from where the process was swapped out
  - if binding is done at run-time, the process can be swapped in (back) at any position
- transfer speed (time) to and from backing store (normally the dominant term in switching time)
  - it would be useful for the system to know exactly how much memory a process needs, to swap in only that part, in order to reduce the swap in (transfer) time

8.48

3 Security Issues

Overview

- common programmers' mistakes related to memory
  - some lead to **program crash**
  - some lead to **program misbehavior**
  - some turn into **vulnerabilities** that could be **exploited** by an **attacker**
    - e.g. lead to execution of unintended code, in attacker's advantage
- possible solutions or mitigation techniques
  - compiler based
  - OS based
- (some) explained in relation to paging, though independent of it
  - based on the fact that a page table does not map all pages, but only the ones really filled with the program code and data
  - the virtual pages (VP) in holes are marked as invalid in the corresponding page table entries (PTE)

8.49

Usage of a Non-Initialized Pointer

- **global** pointer
  - this will be automatically **initialized with 0, i.e. NULL**
    - as all global variables are
  - lead to **program crash**, due to invalid memory access, i.e. "*page fault*" exception
    - usually the page 0 (so VA zero) is not mapped
- **local** pointer, i.e. declared in a function
  - allocated (as any local variable) on the stack $\Rightarrow$ its **initial value is undefined**
  - lucky case: its value corresponds to an unmapped virtual page
    - generates a "*page fault*" exception
    - $\Rightarrow$ lead to **program crash**
  - unlucky case: its value corresponds to a mapped page
    - obviously not to a page desired by the user
    - accesses and possible changes go undetected until maybe later time
    - read / overwrite random data in the program
    - $\Rightarrow$ very **tricky bugs**
    - $\Rightarrow$ exploitations: **data leakage**, **execute unintended code**

8.50

Usage of a Non-Initialized Pointer. Example

```
#include <stdio.h>

#define MAX_LEN 4096

int* global_pointer;

int rec_function(int no)
{
    char* local_pointer;

    printf("global_pointer = %p, local_pointer = %p\n", global_pointer, local_pointer);

    //local_pointer[0] = 0;

    if (no > 0)
        rec_function(no-1);

    return 0;
}

int main(int argc, char **argv)
{
    rec_function(9);
    printf("------\n");
    rec_function(9);

    return 0;
}
```

```
$ gcc -Wall uninitialized-pointers.c -o uninitialized-pointers.exe
uninitialized-pointers.c: In function 'rec_function':
uninitialized-pointers.c:11:2: warning: 'local_pointer' is used uninitialized in this function [-Wuninitialized]
  printf("global_pointer = %p, local_pointer = %p\n", global_pointer, local_pointer);
  ^
$ ./uninitialized-pointers.exe
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = 0x7ff4fdffa168
global_pointer = (nil), local_pointer = 0x400650
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = (nil)
global_pointer = (nil), local_pointer = 0x7fffffd2
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffcf
global_pointer = (nil), local_pointer = 0x7fffffd2
global_pointer = (nil), local_pointer = 0x7fffffd2
------
global_pointer = (nil), local_pointer = 0x7ffe2d8c4e30
global_pointer = (nil), local_pointer = 0x7ff4fda8381b
global_pointer = (nil), local_pointer = 0x7ff4fda83409
global_pointer = (nil), local_pointer = 0x7ff4fda81bff
global_pointer = (nil), local_pointer = 0x7ff4fdb002c0
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
global_pointer = (nil), local_pointer = 0x7fffffc9
```

8.51

Usage of a Non-Initialized Pointer. Coding Recommendation

- initialize with NULL a pointer, when declared
- assure a pointer is assigned a valid value (i.e. initialized) before using it

```
int* global_pointer = NULL;

int rec_function(int no)
{
    char* local_pointer = NULL;
    ...

    global_pointer = (char*) malloc(sizeof(int));
    if (global != NULL) {
        // use it
    }

    local_pointer = (char*) global_pointer;
    if (local_pointer != NULL) {
        // use it
    }
}
```

8.52

Use After Free

- context
  - **dynamically allocated memory**
  - its address returned

  - got it in a pointer variable
  - **free the allocated memory**, passing the pointer as argument
    * though the **pointer value is not affected**
    * $\Rightarrow$ the previously used memory address still available in program
    * the virtual page the (released) memory was located could be still valid (mapped)
- **vulnerability: use the "*dangling pointer*" as if still valid**
  - lucky case: the program crashes, due to invalid memory
  - unlucky case: the program overwrites other data
- **could be exploited** by attackers to
  - **leak application's secrets**
  - **execute unintended code**

8.53

Use After Free. Example

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

#define MAX_LEN 4096

int main(int argc, char **argv)
{
    char* pointer_1, *pointer_2;

    pointer_1 = malloc(MAX_LEN * sizeof(char));
    printf("pointer_1 = %p\n", pointer_1);

    for(int i=0; i<MAX_LEN; i++)
        pointer_1[i] = 'A';
    printf("pointer_1[0] = %c\n", pointer_1[0]);

    free(pointer_1);

    pointer_2 = malloc(MAX_LEN * sizeof(char));
    printf("pointer_1 = %p, pointer_2 = %p\n", pointer_1, pointer_2);

    for(int i=0; i<MAX_LEN; i++)
        pointer_2[i] = 'B';

    printf("pointer_1[0] = %c\n", pointer_1[0]);

    return 0;
}
```

```
$ gcc -Wall use-after-free.c -o use-after-free.exe

$ ./use-after-free.exe
pointer_1 = 0x8f2010
pointer_1[0] = A
pointer_1 = 0x8f2010, pointer2 = 0x8f2010
pointer_1[0] = B
```

8.54

Use After Free. Coding Recommendation

- initialize to NULL a pointer to a just released memory
- make sure to never reuse a dangling pointer (i.e. pointing to a released memory)

```
char* pointer;

pointer = malloc(...);

...

free(pointer);
pointer = NULL;
```

8.55

Return the Address of a Local Variable

- context
  - **local variables are allocated on the stack**
  - at function call (enter)
  - at function return, the stack space is released and could be reused
- **mistake**: a function **returns the address of a local variable**

- lucky case: stack is not reused when accessing the variable
  - program runs normally
- unlucky case: stack reused, i.e. other functions called
  - ⇒ **bugs**, i.e. **program misbehavior**
  - ⇒ **vulnerabilities**, like **data leakage**, **unintended code execution**

8.56

Return the Address of a Local Variable. Example

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

#define MAX_LEN 4096

int* compute_power(int base, int exp)
{
    int result;

    result = 1;
    for (int i=0; i<exp; i++)
        result *= base;

    printf("Inside the function: &result = %p, 2^10 = %d\n", &result, result);

    return &result;
}

int main(int argc, char **argv)
{
    int* power;

    power = compute_power(2, 10);

    printf("Outside the function: power = %p\n", power);
    printf("Outside the function: 2^10 = %d\n", *power);
}
```

```
gcc -Wall return-as-result-the-address-on-stack.c -o return-as-result-the-address-on-stack.exe
return-as-result-the-address-on-stack.c: In function 'compute_power':
return-as-result-the-address-on-stack.c:16:9: warning: function returns address of local variable [-Wreturn-local-addr]
  return &result;
         ^
```

```
$ ./return-as-result-the-address-on-stack.exe
Inside the function: &restul = 0x7fffc7905990, 2^10 = 1024
Outside the function: power = (nil)
Segmentation fault (core dumped))
```

8.57

Return the Address of a Local Variable. Coding Recommendation

Never do this!

8.58

Buffer Overflow

- **vulnerability** (maybe the **most famous**)
  - declare **/ allocate a memory buffer of a certain size**
  - **write more space than allocated**
  - ⇒ overwrites other adjacent memory locations, e.g. variables, control data
- lucky case
  - touches addresses in non-mapped pages
  - ⇒ generate page fault and **program crash**
- unlucky case
  - touches addresses only in mapped (so, valid) pages
  - **overwrites other program's data**
  - could generate tricky bugs
- **could be exploited** to
  - reveal application's secrets (**data leakage**)
  - **execute unintended code**
- types
  - stack smashing
  - heap smashing

8.59

Buffer Overflow. The "*Classical*" Example

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    strcpy(dst, src);  // vulnerability!
}
```

- a vulnerability when `src`'s contents and length controlled by the user (i.e. attacker)
- $\Rightarrow$ could copy more than allocated MAX bytes, overwriting the bytes following in memory to "`dst`"

8.60

Buffer Overflow. Non Nul-Terminated String Example

```
#define MAX 10

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

    read(fd, buf, MAX);

    int len = strlen(buf);       // vulnerability!
    printf("buf = %s, len = %d\n", buf, len); // vulnerability!
}
```

- "`read`" just reads bytes, not working with nul-terminated strings
- string functions go in memory until finding a 0
- maybe the most encountered mistake in the (first) lab assignment solutions

8.61

Buffer Overflow. Another "*Classical*" (off-by-one) Example

```
#define MAX 10

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

    int n = read(fs, buf, MAX);
    buf[n] = '\0'; // vulnerability!

    printf("buf = %s\n", buf);
}
```

- when $n = MAX \Rightarrow$ "`buf[MAX] = 0`" overwrites the next byte after "`buf`"
- due to the good intention to terminate a string with 0 (nul), to be correctly handled by string functions

8.62

Buffer Overflow. Non Nul-Terminated String (Subtle) Example

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    strncpy(dst, src, MAX);  // vulnerability!
    printf("dst = %s\n", dst);
}
```

- does not terminate "`dst`" with 0 when "`strlen(src) >= MAX`"
- best solution: `strcpy_s`

8.63

Buffer Overflow. Coding Recommendation

- use safe functions for string manipulation
  - e.g. `strncpy`, `strcpy_s` instead of `strcpy`
- take care of string functions, read carefully their specification
- check correctly buffer limits

```
#define MAX 10

int main(int argc, char **argv)
{
    char dst[MAX];
    char *src;

    if (argc != 2)
        exit(1);

    src = argv[1];
    int err = strcpy_s(dst, MAX, src);
    if (err != 0) {
        // error
        exit(1);
    }
    // dst always nul-terminated and not overflowed
    printf("dst = %s\n", dst);
}
```

8.64

Buffer Overflow. Solutions

- fundamental: **write correct (secure) code**
  - most effective (**remove the cause**)
  - **do not trust user-provided data**, but **validate it**
- **compiler-based** (**mitigate effects**)
  - stack canaries
  - control-flow integrity
- **OS-based** (**mitigate effects**)
  - process **isolation**
  - address space layout randomization (**ASLR**)
  - no-execution pages / data execution prevention (**DEP**)
- **hardware-based** (**mitigate effects**)
  - Intel MPX
  - needs compiler support

8.65

More Information

- vulnerability description
  - http://cwe.mitre.org
  - http://cwe.mitre.org/data/slices/2000.html
  - http://cwe.mitre.org/top25/index.html
- vulnerability databases
  - http://www.cvedetails.com/
  - http://cve.mitre.org/
  - http://www.securityfocus.com/
  - https://www.exploit-db.com/

8.66

4 Conclusions

What we talked about

- *virtual address space* (VAS) vs. *physical address space* (PAS)
  - virtual address (VA) vs physical address (PA)
- binding (translating) VAs to PAa
  - *compile time* vs *load time* vs *run time*
- *contiguous allocation*
  - base and limit
  - + simple
  - - rigid, external fragmentation, no support for sparse VAS
- *segmentation*
  - more contiguous areas, each with its base and limit
  - + supports sparse VAS
  - - external fragmentation
- *paging*
  - uniform, fixed-size allocation units, i.e. pages
  - + supports sparse VAS, fine-grained memory sharing, no external fragmentation
  - - internal fragmentation

8.67

What we talked about (cont.)

- *swapping*
- memory related-*vulnerabilities*

8.68

Lessons Learned

1. multiprogramming implies usage of virtual memory addresses (VAs)
2. low-level access to memory, like C allows, is both useful and dangerous

8.69