V.5.5. Using the MMU

Hardware

Intel case

- segmentation
  - cannot be disabled
  - but can be "avoided" through software
- pagination
  - can be enabled/disabled

The Operating System

Windows/Linux cases

- segmentation
  - not used in practice
  - all segments are sized such that each one covers the entire memory
- pagination
  - pages of 4 KB
  - Windows can also use pages of 4 MB

Utility of the MMU (1)

Advantages:

- protection to errors
- an application cannot impair another application's working
- checking is performed in hardware
  - safe mechanism
  - higher speed

Utility of the MMU (2)

Drawbacks

- complex management
- memory occupied by the dedicated data structures
  - descriptor table
  - page table
- lower speed - doubles the number of memory accesses (or even worse)

Utility of the MMU (3)

Conclusions

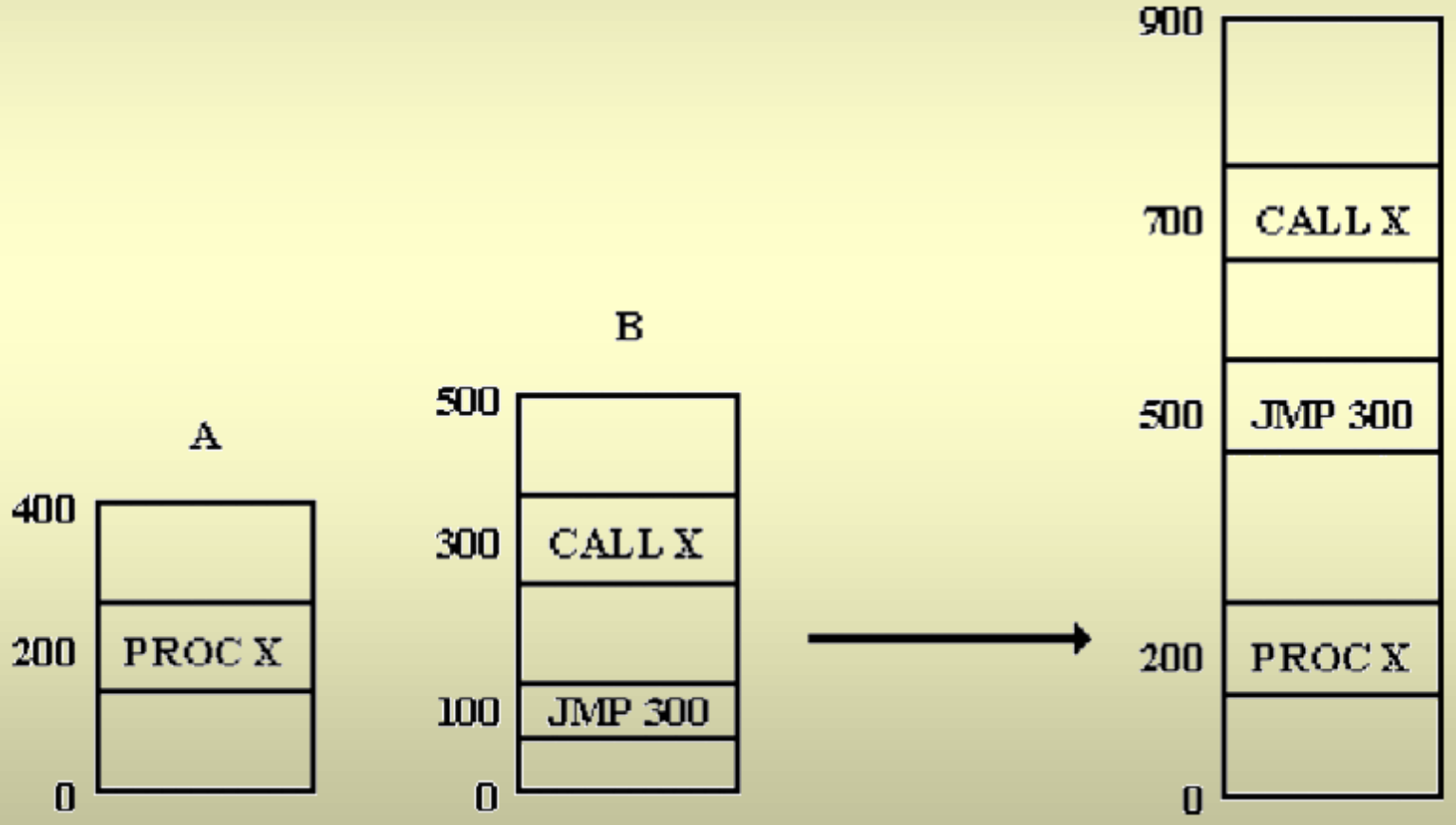
- the loss of performance can be compensated by using caches
- current processors provide enough speed
- multitasking systems - high risk of interferences
- MMU's mechanisms should be used

V.6. Creating and Executing Programs

Creation of a Program - Stages

- compiling
  - translate the commands written in a source language into processor instructions
- linking
  - handles aspects regarding program's memory management

Creating an Executable File from Multiple Source Modules



Relocation Problem

- jump instruction - jump address is no longer correct
- modules are compiled independently - each one starts from address 0
- also affects instructions which access data (memory addresses)
- addresses are relocated (moved) compared to compilation time

External Reference Problem

- function X - called from another module than the one it is defined in
- at compilation time
  - we know it is defined in another module
  - it is impossible to determine what is the address where the function will be located in the final program

Creating the Programs

- can one write a program made of a single module?
- not always
- very complex programs - modularity
- function libraries - separate modules
  - precompiled
  - source code is not available

Stages of Program Creation

- compile the modules
  - source file → object file
  - object files contain information necessary at linking time
- update the links
  - object files → executable files
  - uses the information in the object files

Structure of an Object File (1)

1. header
   - identification information
   - information about the other parts of the file
2. entry point table
   - contains the names of the symbols (variables and functions) in the current module that may be used in other modules

Structure of an Object File (2)

3. external reference table
   - contains the names of the symbols defined in other modules, but used in the current module
4. the code itself
   - resulting from compilation
   - the only part that will be found in the executable file

Structure of an Object File (3)

5. relocation dictionary
   - contains information for locating the code instructions the require modifying the addresses they work with
   - variants
     - bitmap
     - linked list

The Linker (1)

1. builds a table with all object modules and their sizes
2. based on this table, assigns start addresses to object modules
   - start address of a module = the sum of the sizes of previous modules

The Linker (2)

3. determines the instructions that access the memory and adds a relocation constant to each address
   - relocation constant = start address of the module it belongs to
4. determines the instructions that call functions/access data from other modules and inserts the appropriate addresses

Program Execution

- what is a program's start address when loaded into memory?
- cannot know at creation time
- all program addresses depend on the start address
- conclusion: relocation problem shows up again when the program is started

Solution 1

- The executable file contains relocation information
  - this information is used by the operating system when the program is loaded into memory
  - in order to update memory accesses
  - example: the DOS operating system

Solution 2

- Use a relocation register
  - always loaded with the start address of the current program
  - on each memory access, the value of the relocation register is added to the address indicated by the instruction
  - depends on hardware
    - not all processors have a relocation register

Solution 3

- Programs only contain relative memory addresses (related to the program counter)
  - position-independent programs
  - can be loaded into memory at any address
  - very hard to write
    - relative jump instructions - restricted
    - instructions that work with relative data addresses - do no exist on most processors

Solution 4

- Memory pagination
  - the program can be moved anywhere in the physical memory
  - the program believes it starts at address 0, even though that is not true
  - depends on hardware support (the pagination mechanism)

Shared Libraries (1)

Dynamic linking

- some functions and variables may not be permanently included into the program
  - only when they are needed
- some functions and variables may be shared by more programs

Shared Libraries (2)

Utility of dynamic linking

- functions that handle exceptional situations
  - very rarely called
  - useless memory consumption
- functions used by many programs
  - only one copy on the disk
  - only one instance loaded into memory

Shared Libraries (3)

Types of dynamic linking

- implicit
- explicit

Implicit Linking

- uses import libraries
  - statically linked within the executable file
  - indicate the shared libraries that are necessary to the program
- upon program launching
  - the operating system checks for the necessary import libraries
  - loads into memory the missing shared libraries (some may already be in memory, due to other programs)

Explicit Linking (1)

- the program makes a specific system call
- requests the linking of a certain shared library
- if the library is not already present in memory, it is loaded
- the link to a shared library may be created of destroyed at any moment

Explicit Linking (2)

- example - Windows case

```
//explicit linking of a module
hLib=LoadLibrary("module");
//get a pointer to a function
fAddr=GetProcAddress(hLib,"func");
(fAddr)(2,3,8);//function call
FreeLibrary(hLib);//module release
(fAddr)(2,3,8);//error, the
  function is no longer available
```

Explicit Linking (3)

- example - Linux case

```
//explicit linking of a module
hLib=dlopen("module",RTLD_LAZY);
//get a pointer to a function
fAddr=dlsym(hLib,"func");
(fAddr)(2,3,8);//function call
dlclose(hLib);//module release
(fAddr)(2,3,8);//error, the
  function is no longer available
```