Memory Management Basics

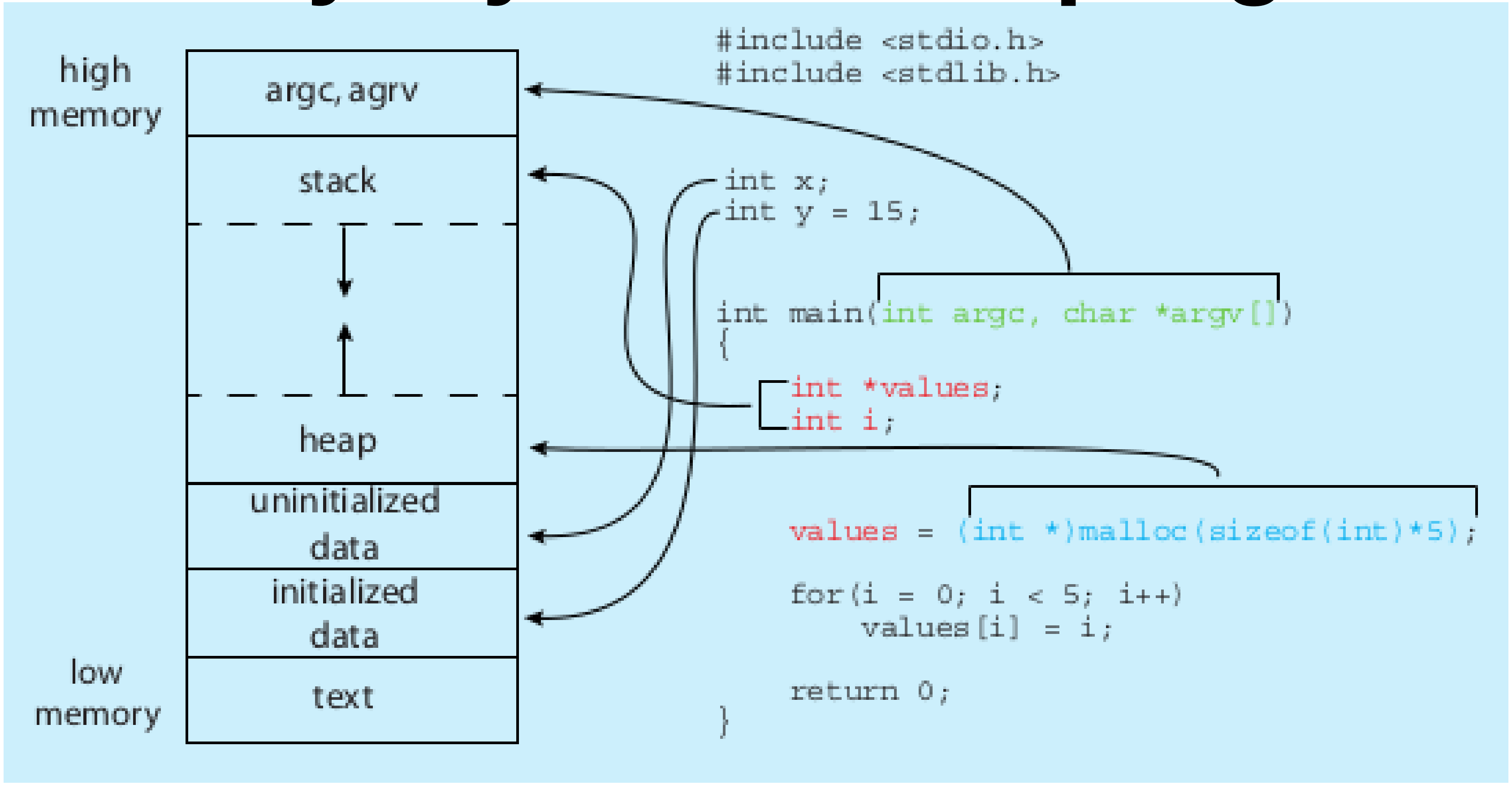
Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
     int *values;
     int i;

     values = (int *)malloc(sizeof(int)*5);

     for(i = 0; i < 5; i++)
          values[i] = i;

     return 0;
}
```

```
$ size /bin/ls
   text       data      bss       dec       hex    filename
 128069     4688     4824    137581   2196d  /bin/ls
```

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other’s code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

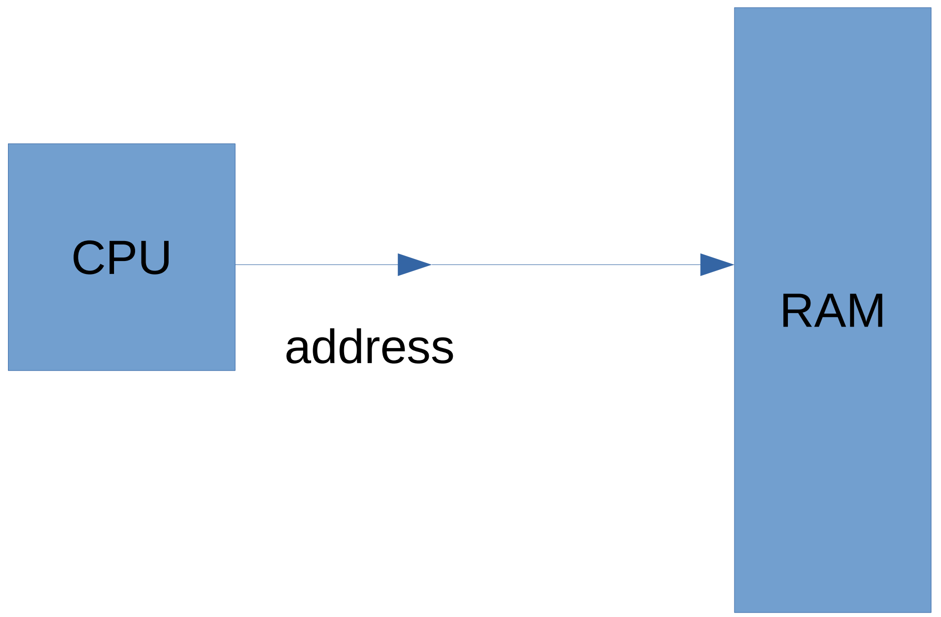
- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

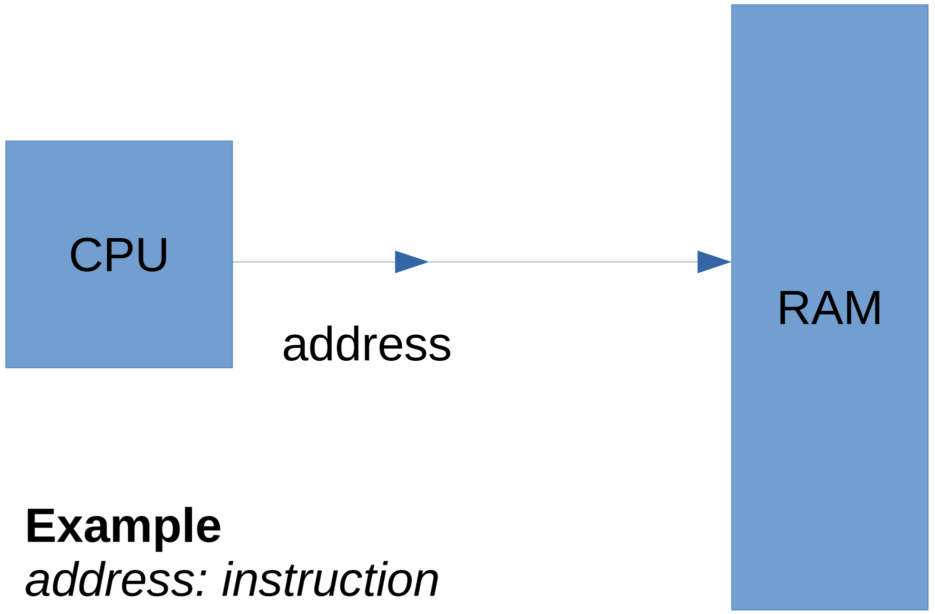
- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case



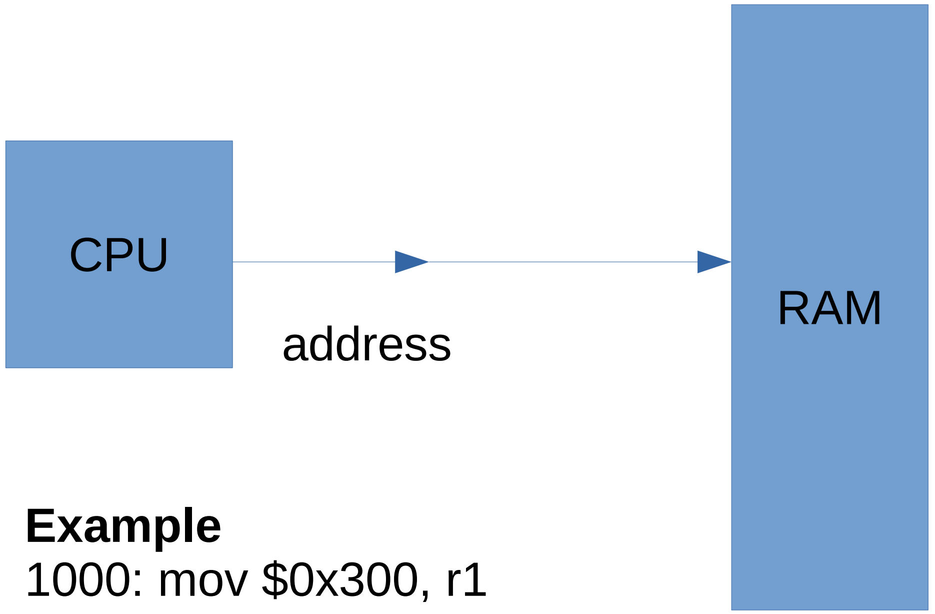
- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

```
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000
```

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

Simplest case

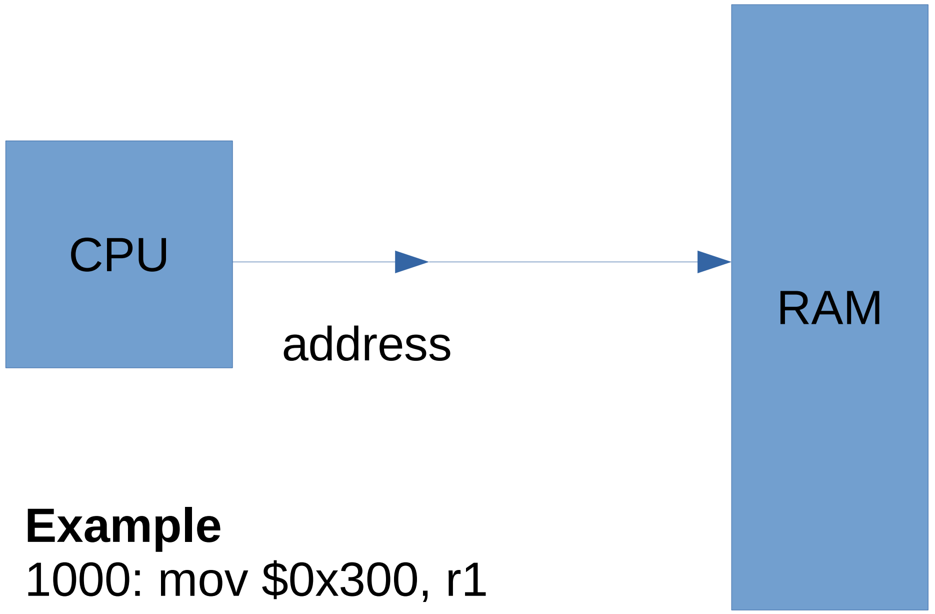


Example

```
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000
```

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case



Example
```
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000
```

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

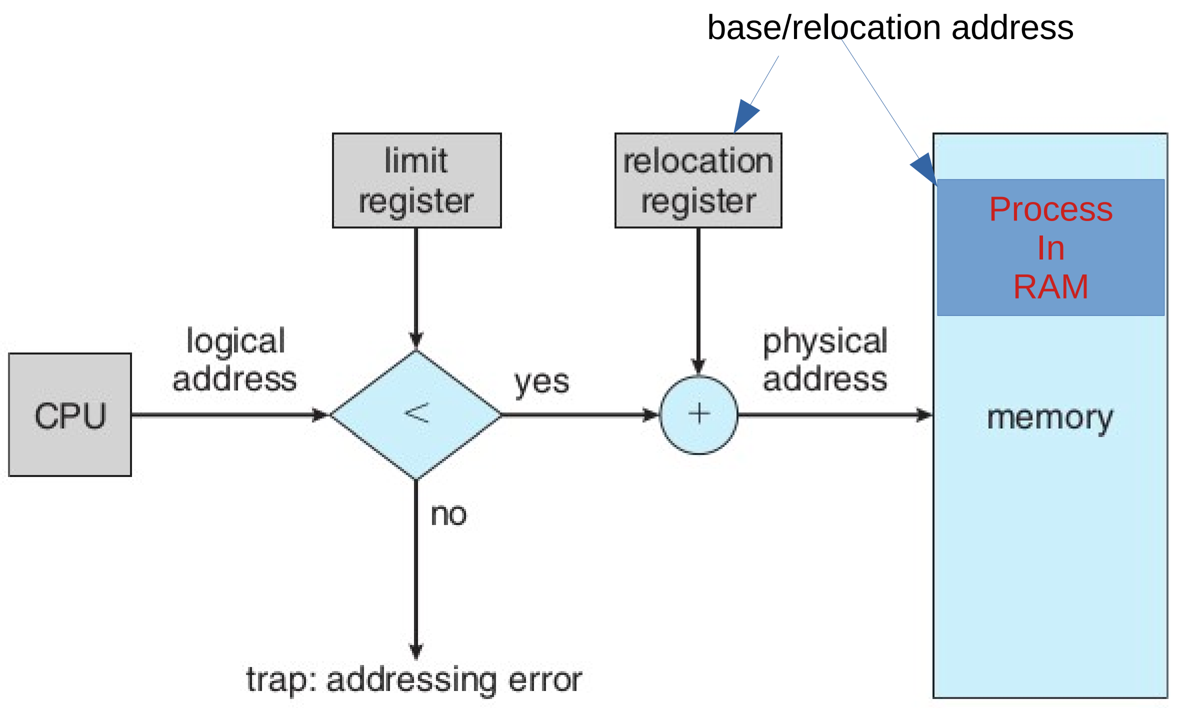


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

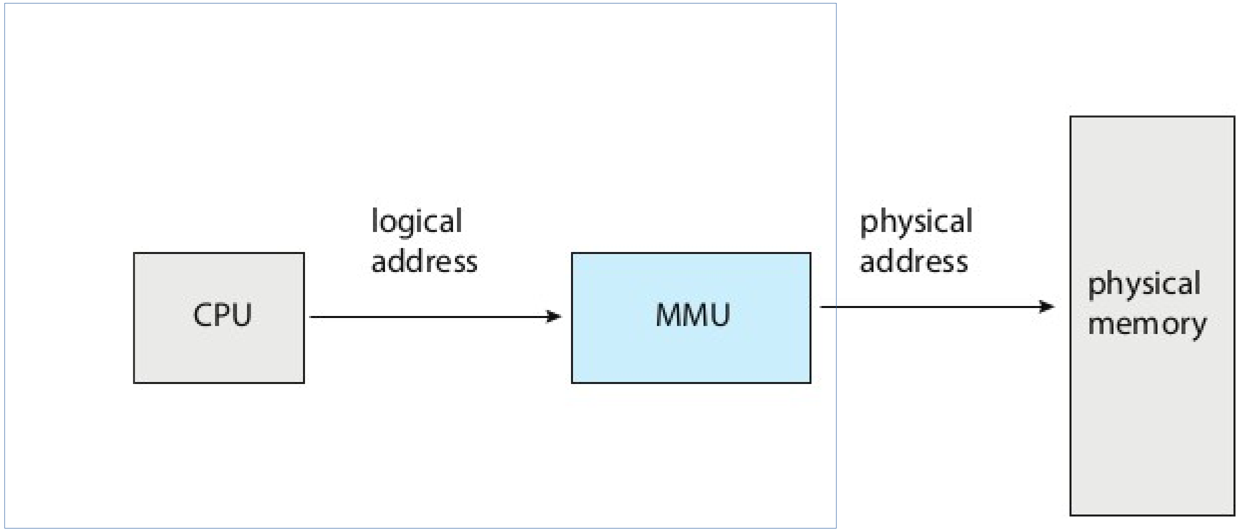
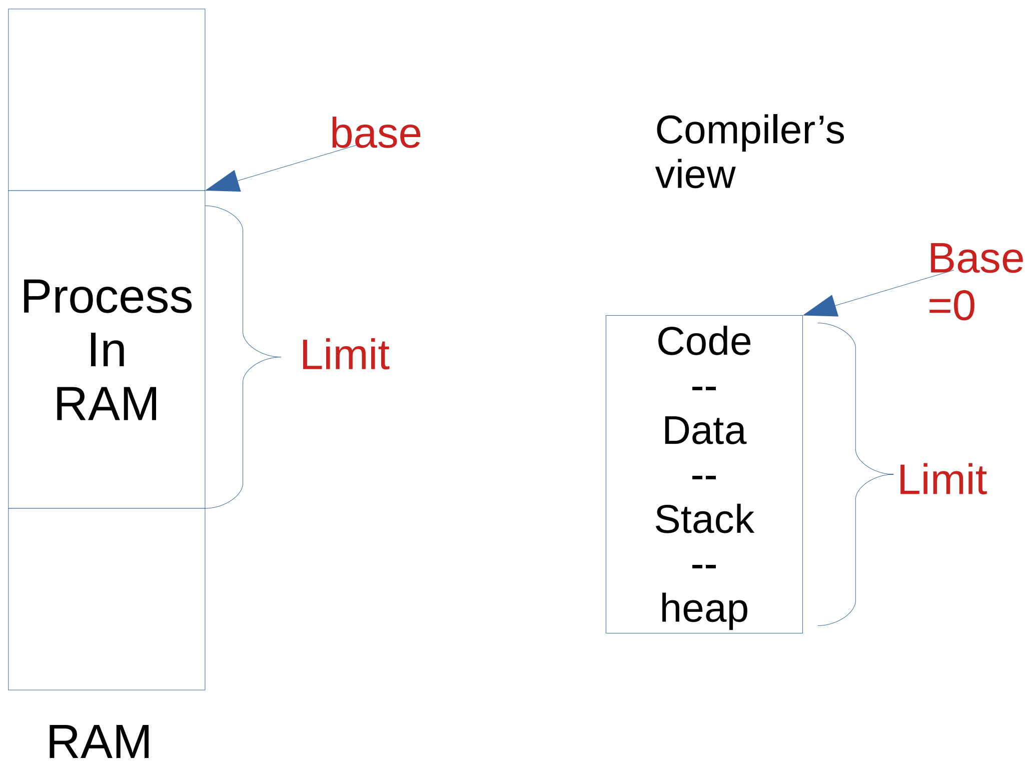


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme



- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

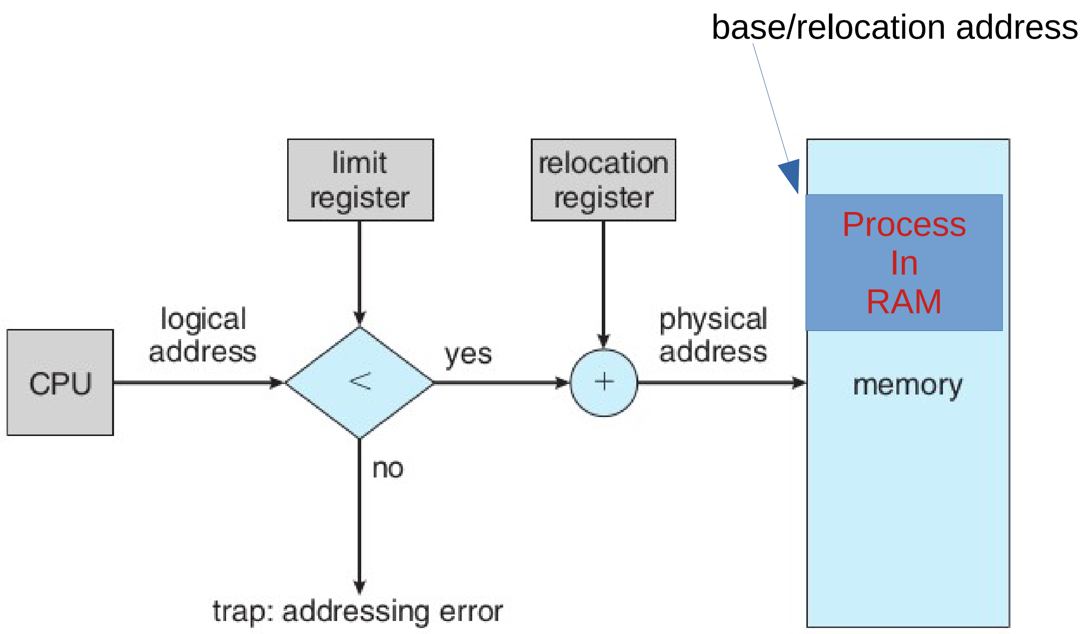


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continous segment
  - Fill in the 'base' register with the actual address of the process in RAM.
  - Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

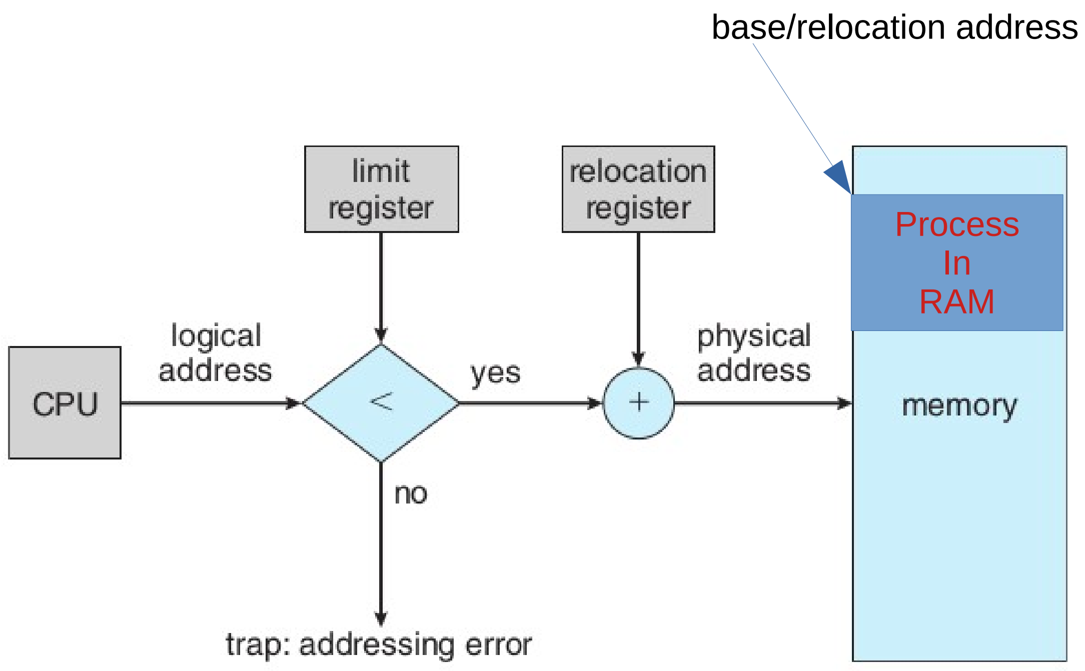
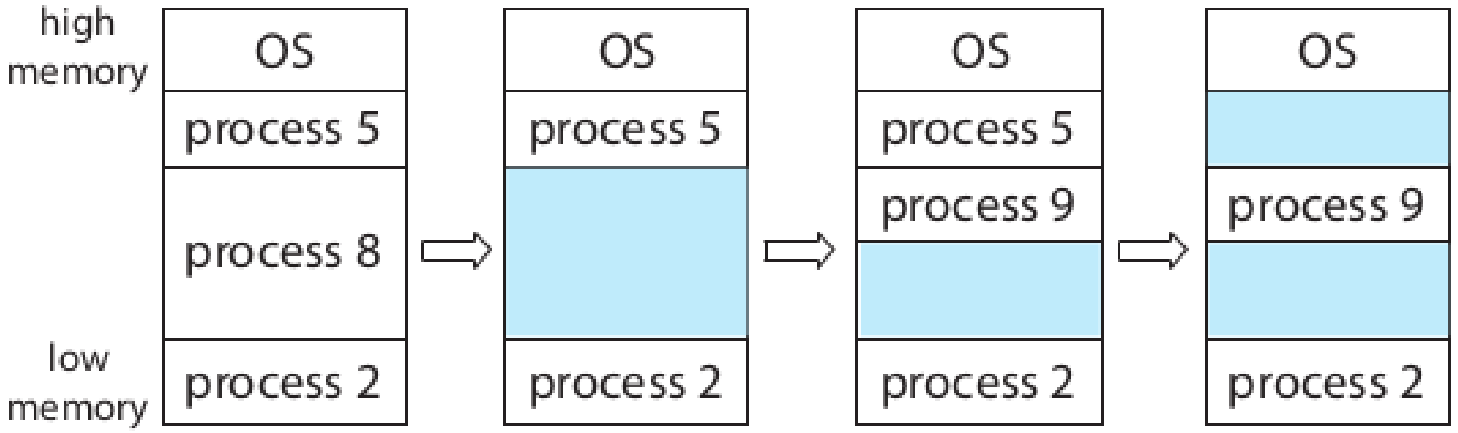


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **“Relocatable code”** – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme



Process 8 called exit()

Process 9 forked

Process 5 terminated

It should be possible to have relocatable code even with “simplest case”

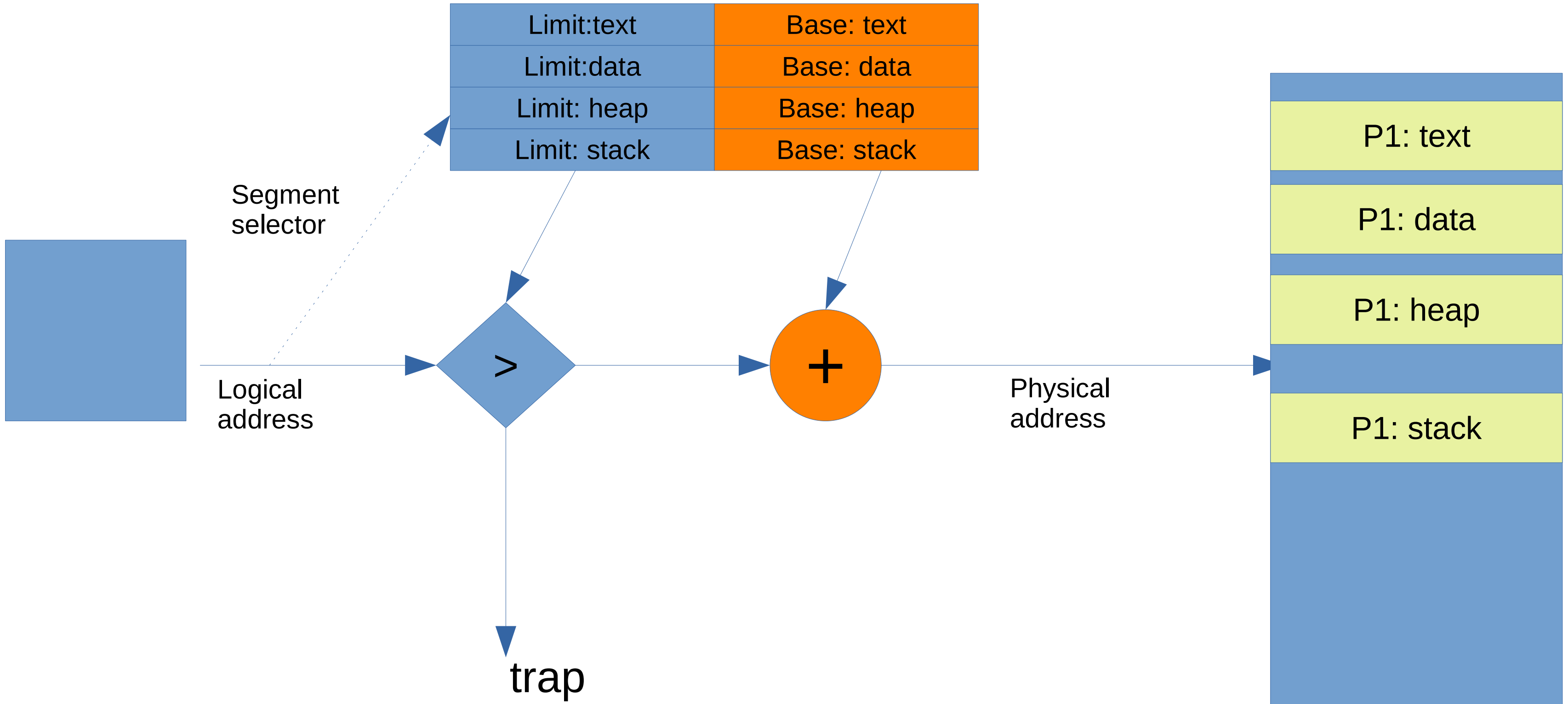
By doing extra work during “loading”.

How?

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme:
Multiple base +limit pairs



Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it’s in memory?
  - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more “base+limits” in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

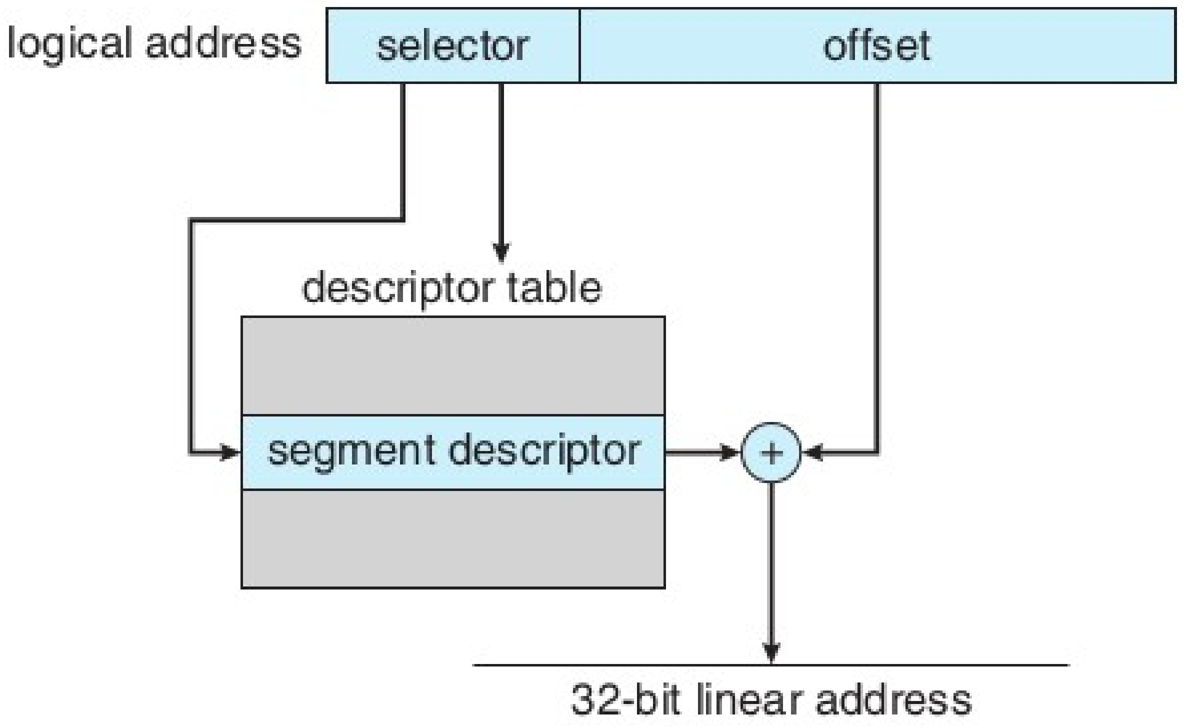


Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the "segment"
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continous in memory!
- Divide the continuous process image in smaller chunks (let’s say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

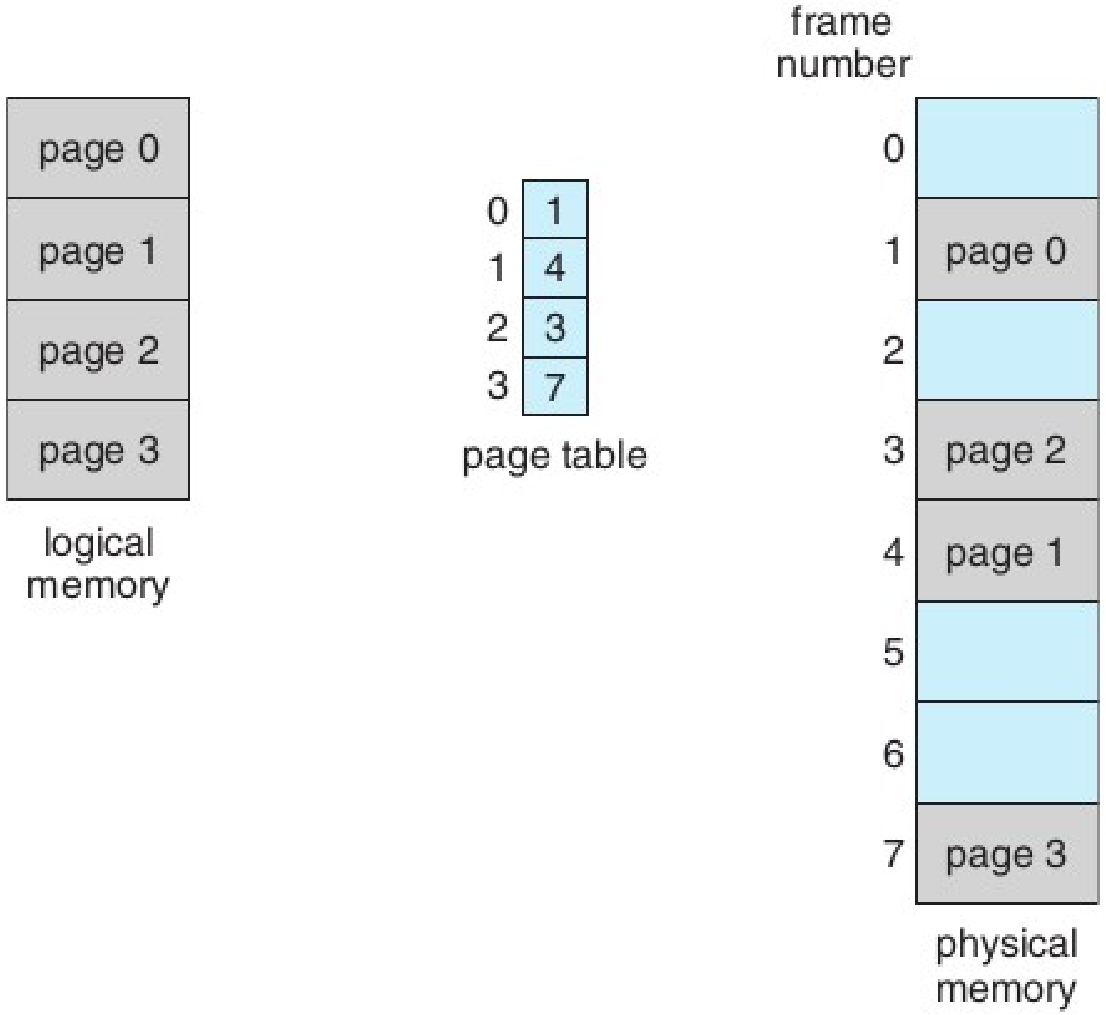


Figure 9.9 Paging model of logical and physical memory.

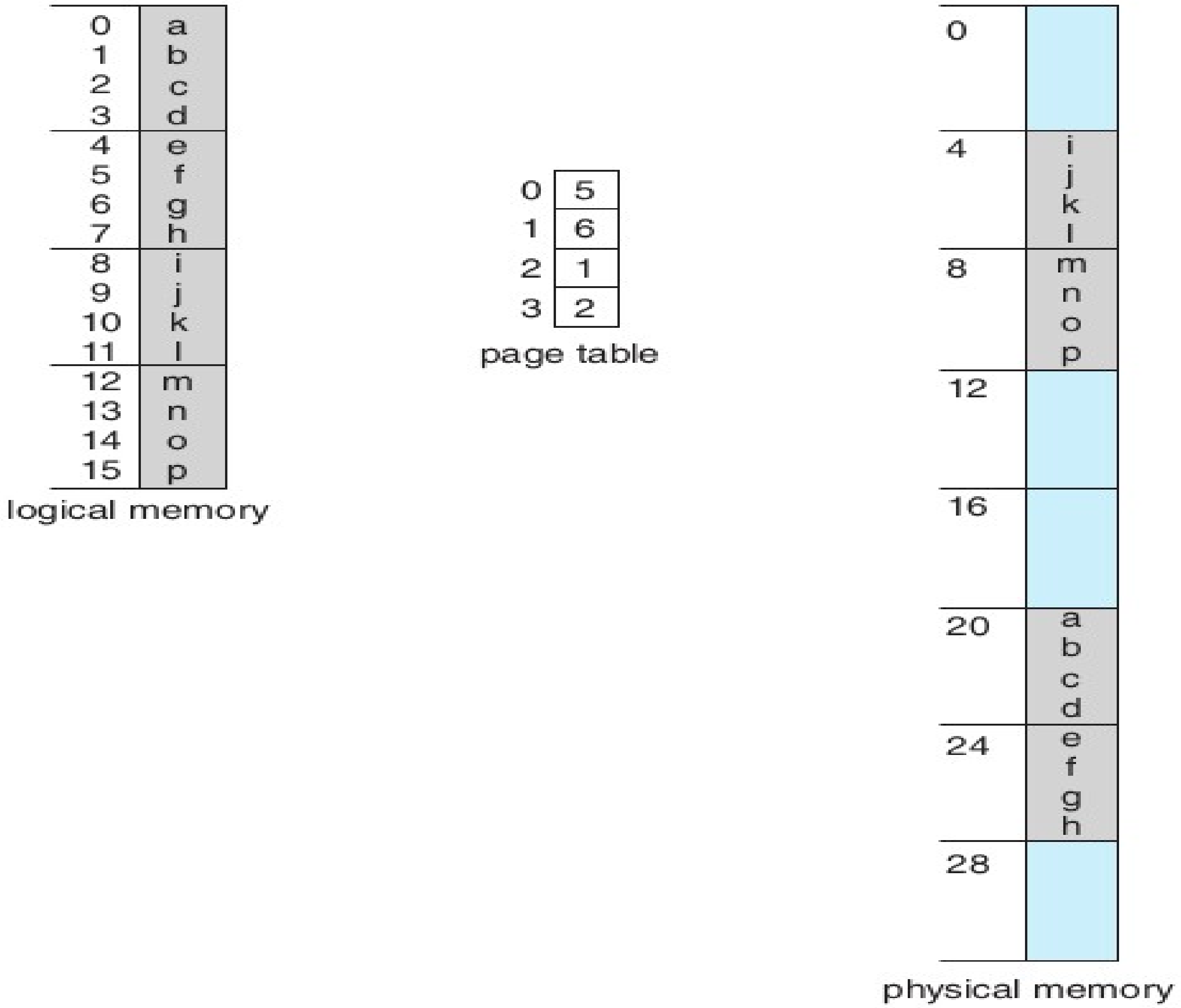


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

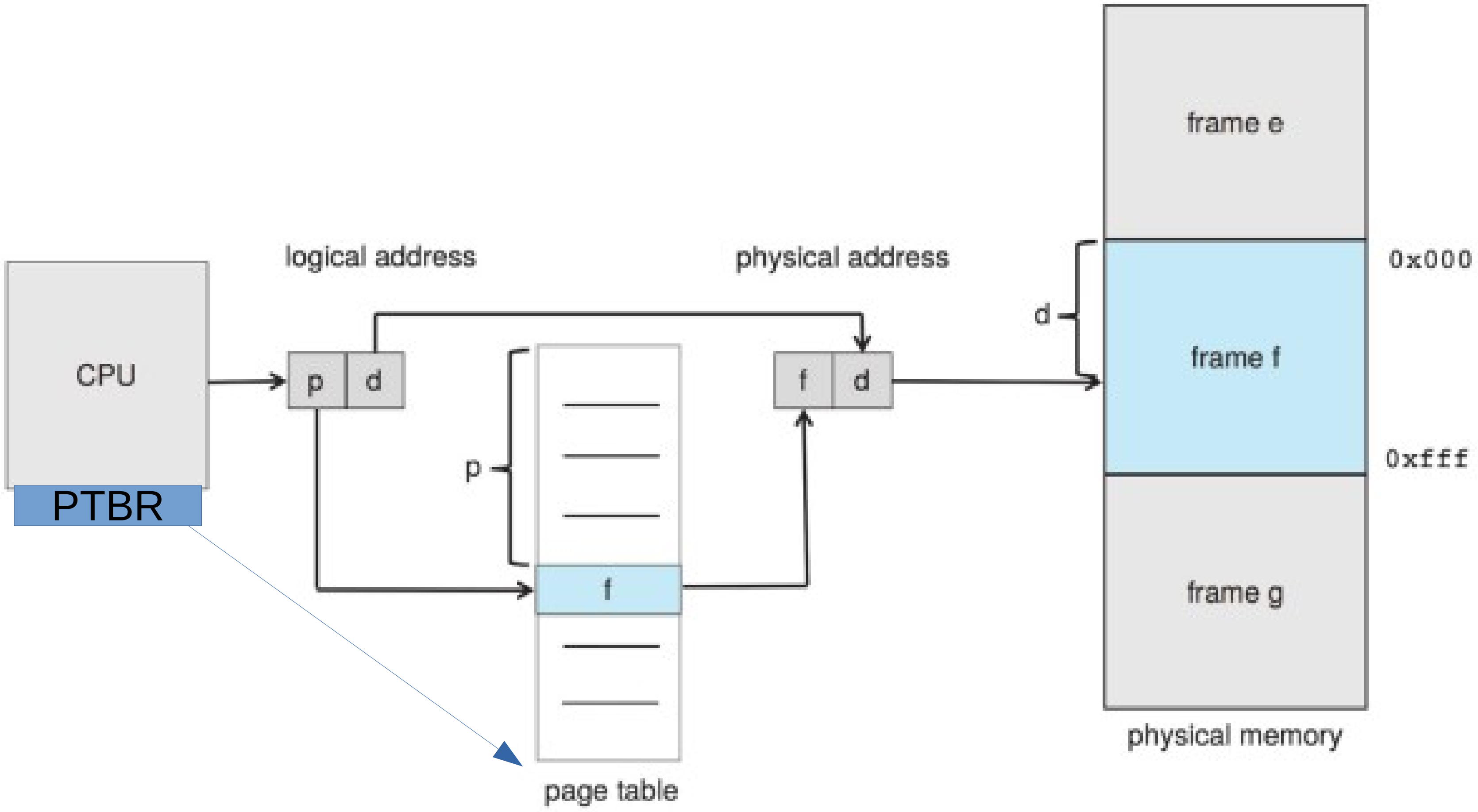


Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management

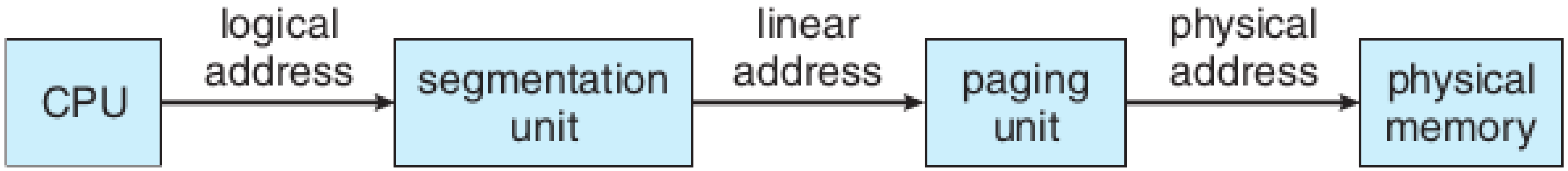


Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

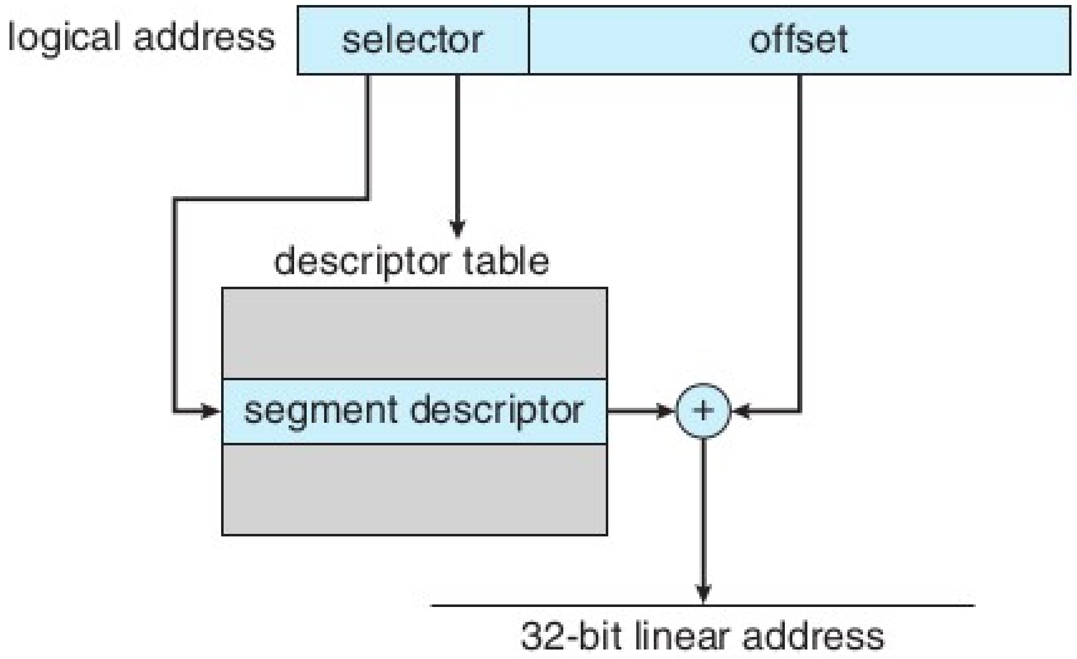


Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

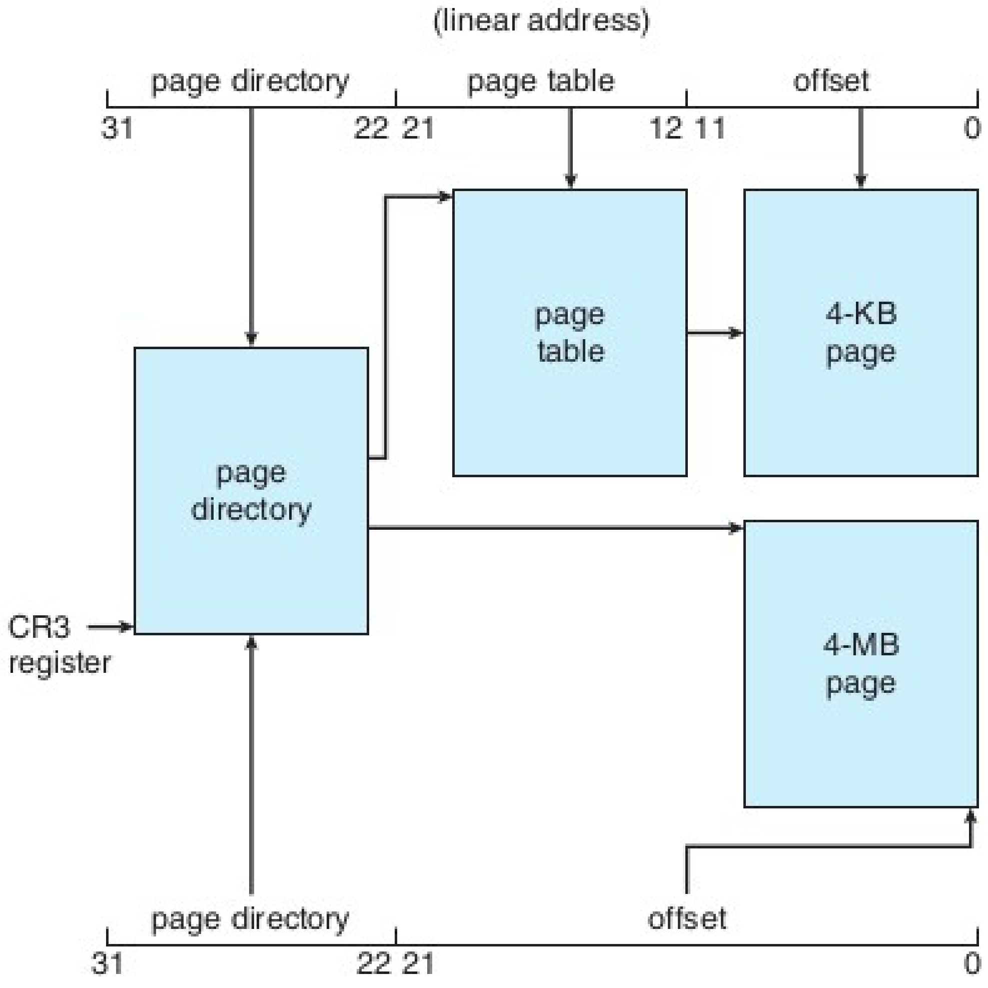


Figure 9.23 Paging in the IA-32 architecture.