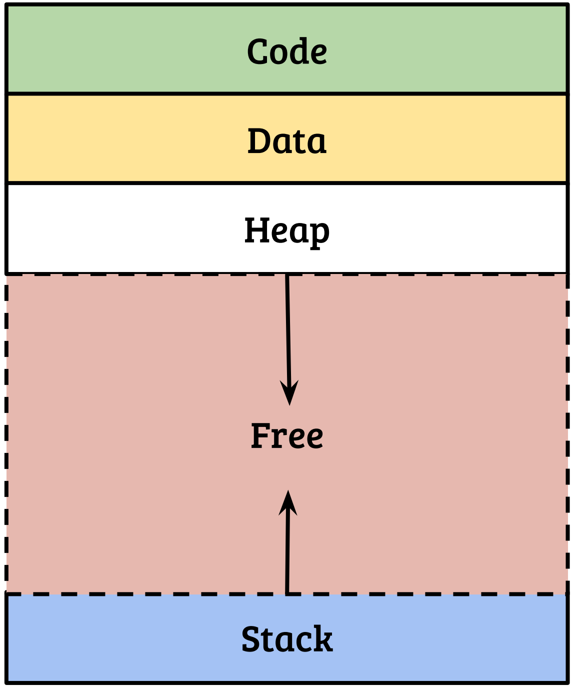
CS330: Operating Systems

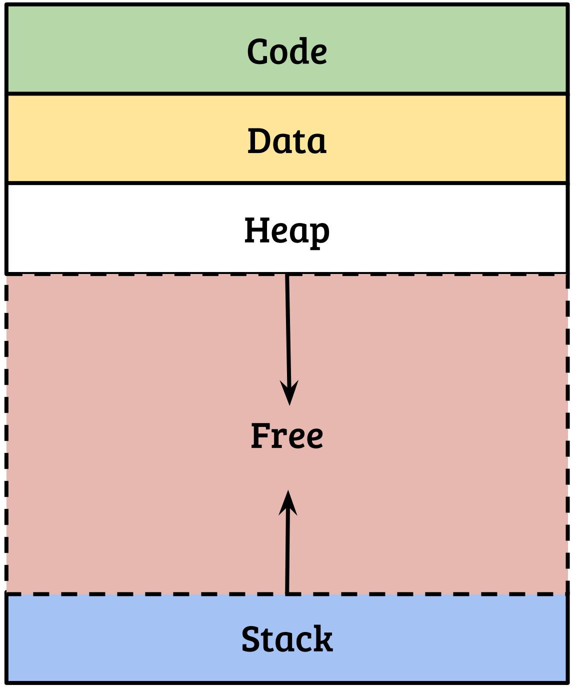
Virtual memory: Address translation

Recap: Process address space



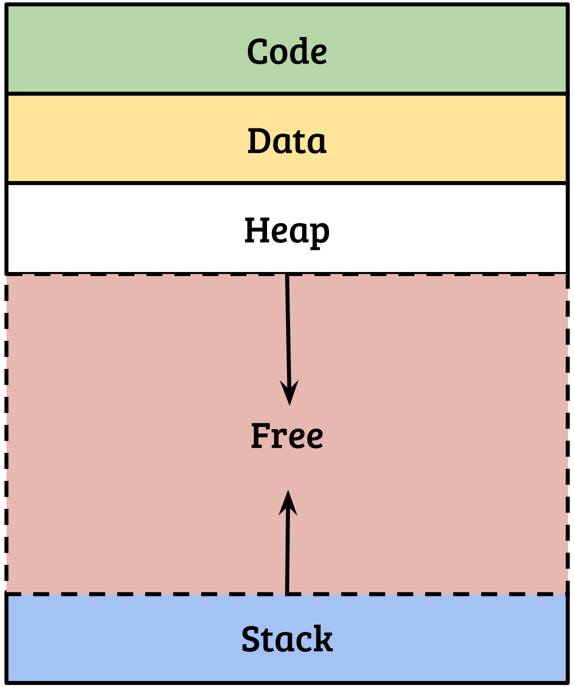
- Address space provides a unique view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view

Recap: Process address space



- Address space provides a unique view of memory to *all processes*
    - Address space is virtual
    - OS enables this virtual view
- User can organize/manage virtual memory using OS APIs
    - No control on physical memory!

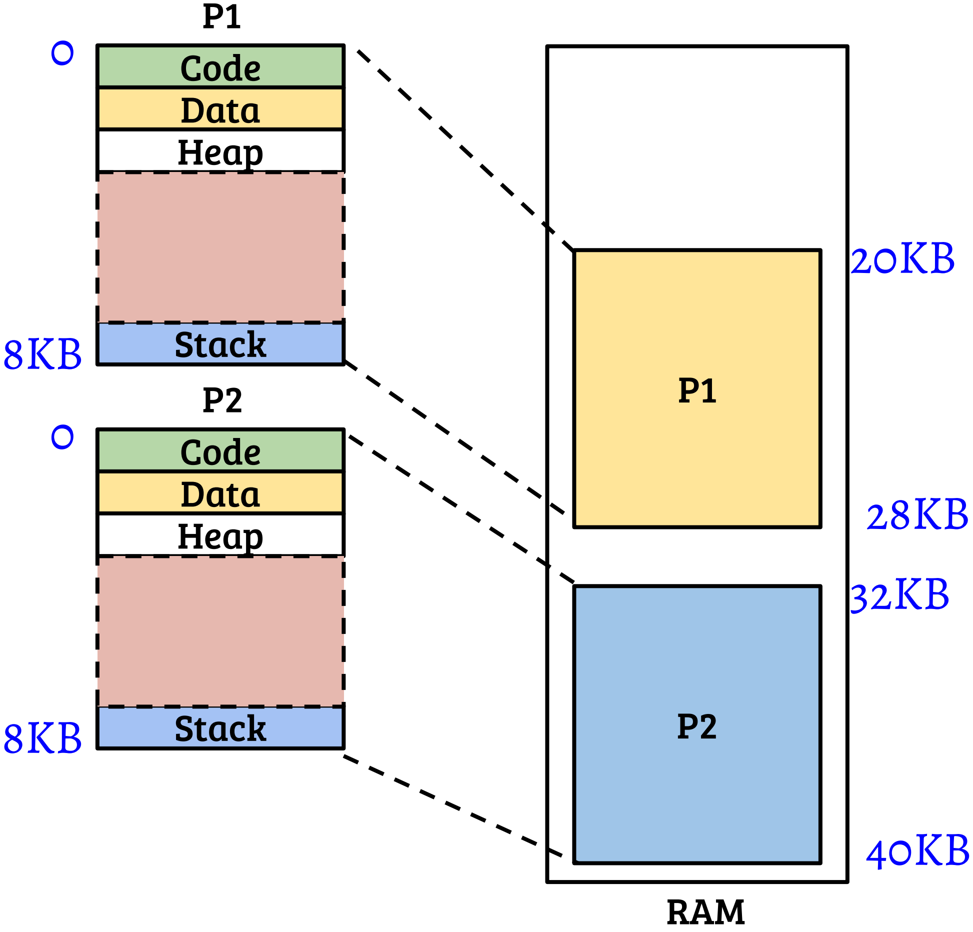
Recap: Process address space



- Address space provides a unique view of memory to *all processes*
  - Address space is virtual
  - OS enables this virtual view
- User can organize/manage virtual memory using OS APIs
  - No control on physical memory!

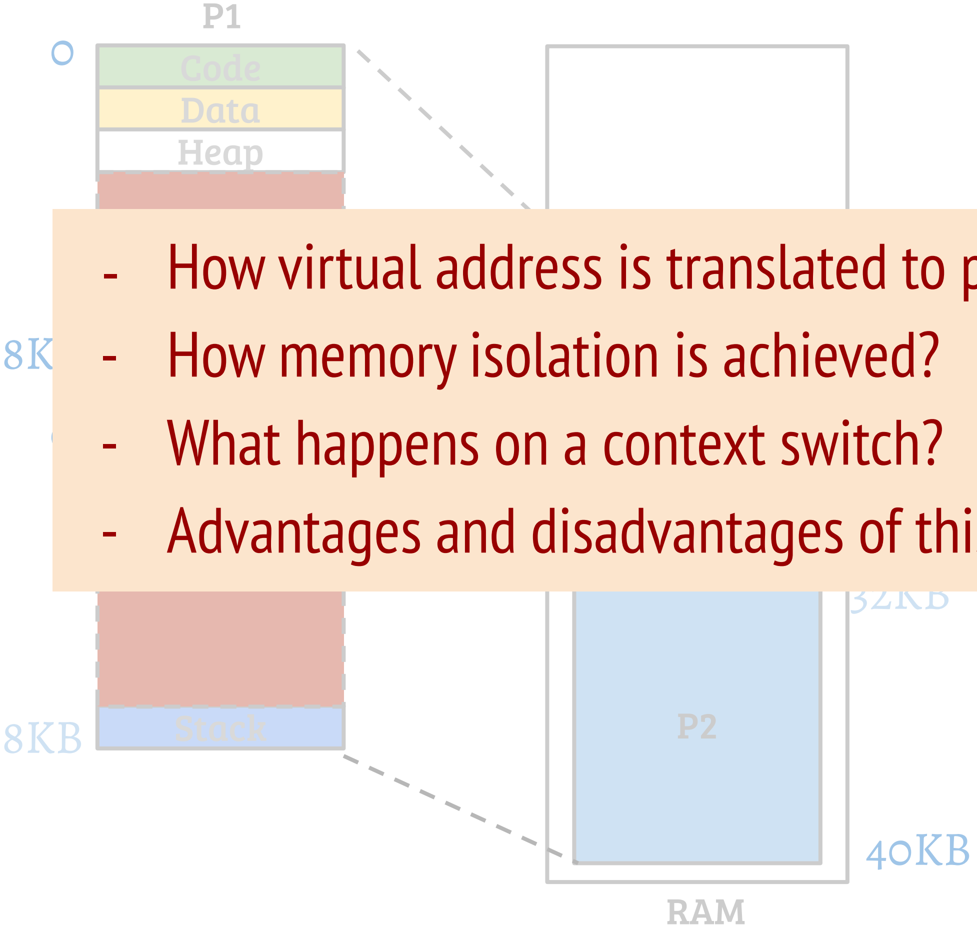
Today's agenda: Virtual to physical address translation

Translation at address space granularity



- Physical memory of same size as the address space is allocated to each process
- Physical memory for a process can be at any address, but should be contiguous

Translation at address space granularity



- Physical memory of same size as

contiguous

- How virtual address is translated to physical address?
- How memory isolation is achieved?
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Role of the compiler

Simple function

```
func( )
{
  int a = 100;
  a+ = 10;
}
```

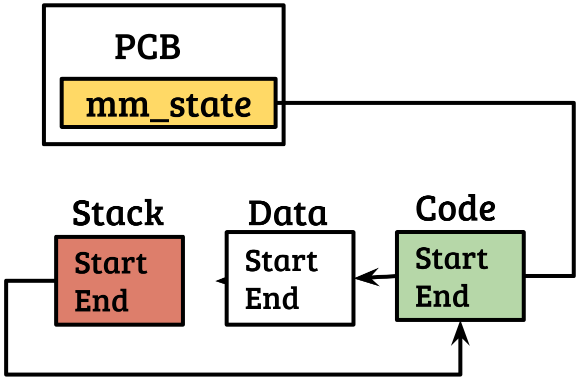
Compiled assembly

```
............
func:
10:   push %rbp;
12:   mov %rsp, %rbp;
15:   mov (%rbp), %rax
18:   add $10, %rax
21:   mov %rax, (%rbp)
24:   pop %rbp;
26:   ret;
............
```

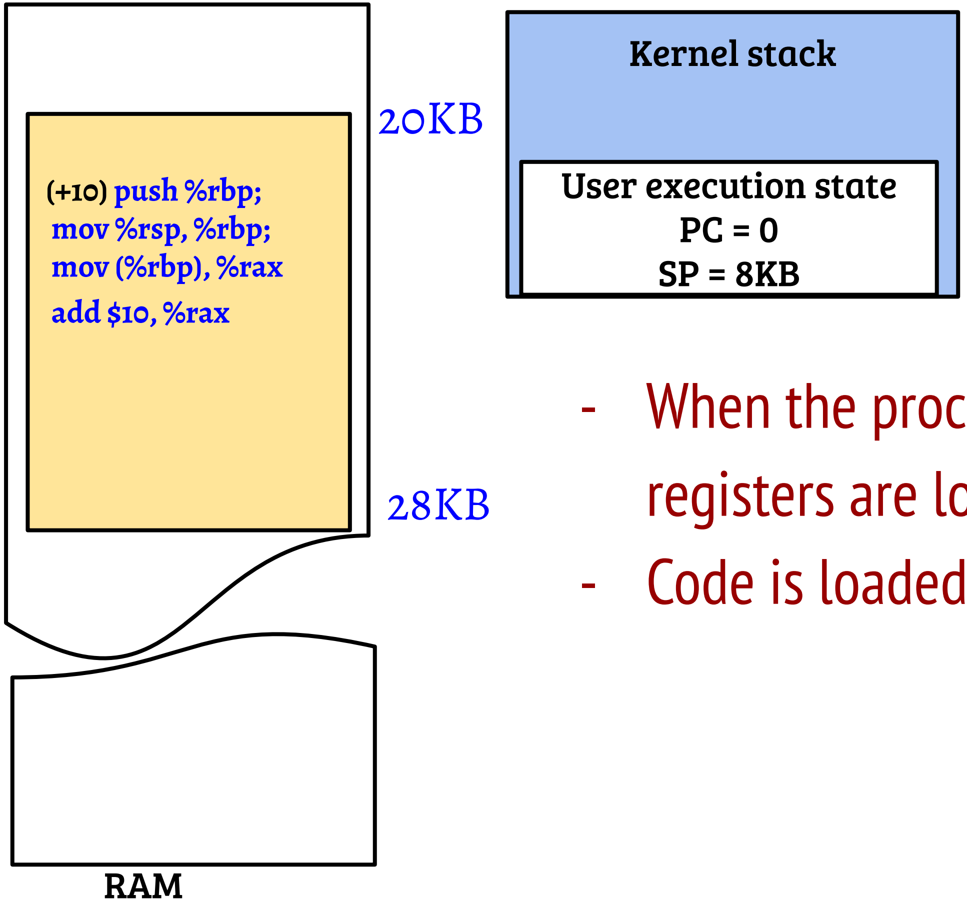
- Compiler generates the code with starting address zero
- Compiler does not know the stack address, blindly uses the registers (rbp, rsp)!

OS during binary load (simplified exec)

```
load_new_executable( PCB *current, File *exe)
{
  reinit_address_space(current → mm_state);
  allocate_phys_mem(current);
  load_exe_to_physmem(current, exe);
  set_user_sp(current → mm_state → stack_start);
  set_user_pc(current → mm_state → code_start);
  return_to_user;
}
```

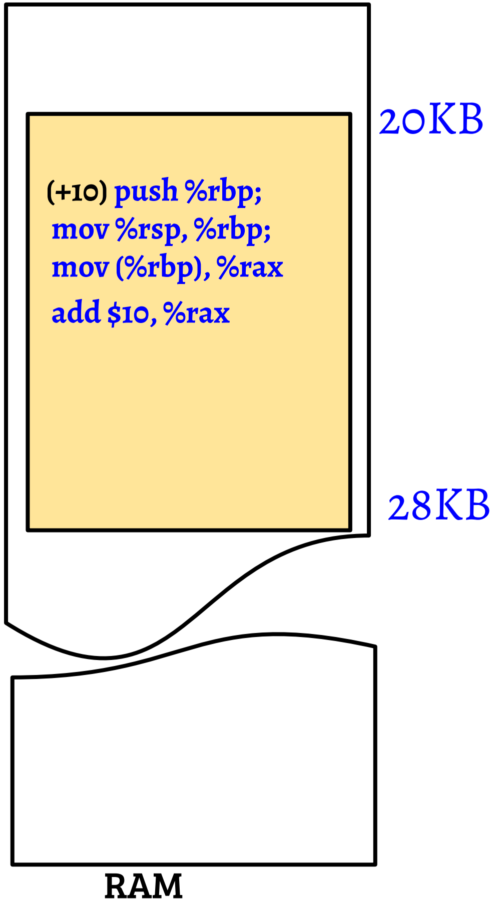


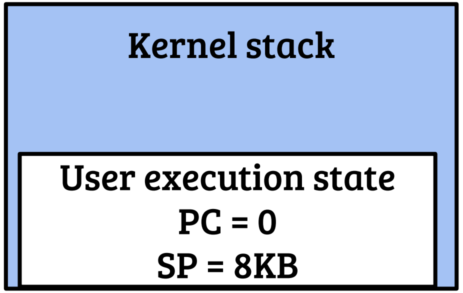
Process state after exec( )



- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)

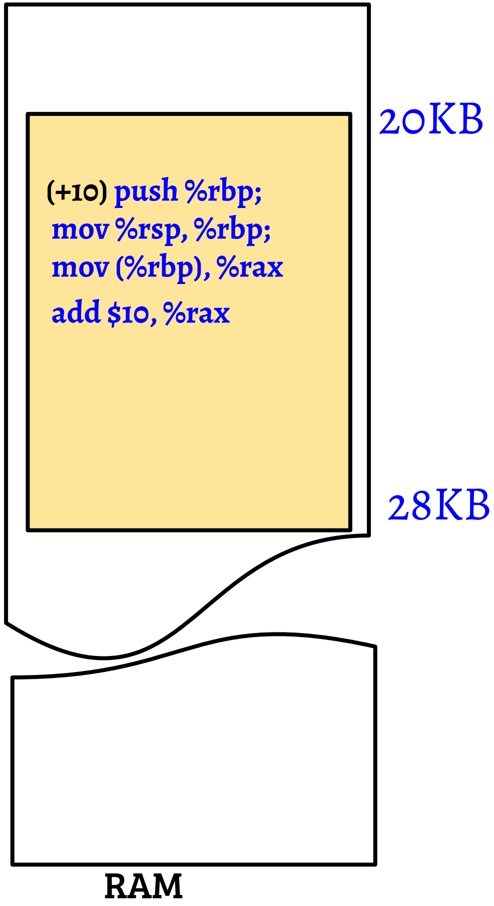
Process state after exec( )

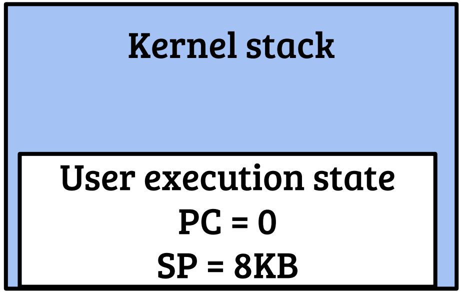


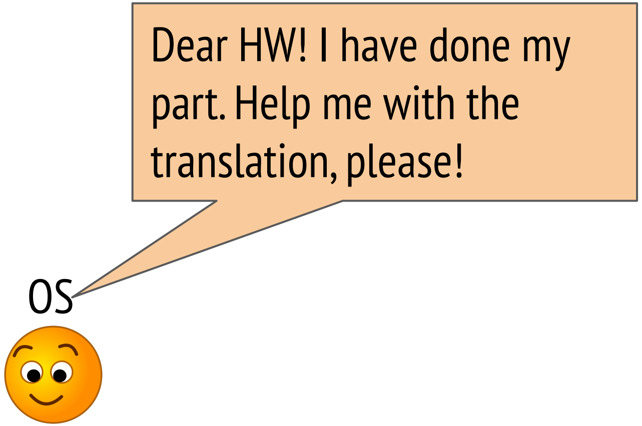


- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of “func” execution
  - Instruction fetch address is 10 (PC = 10)
  - SP will be around 8KB

Process state after exec( )

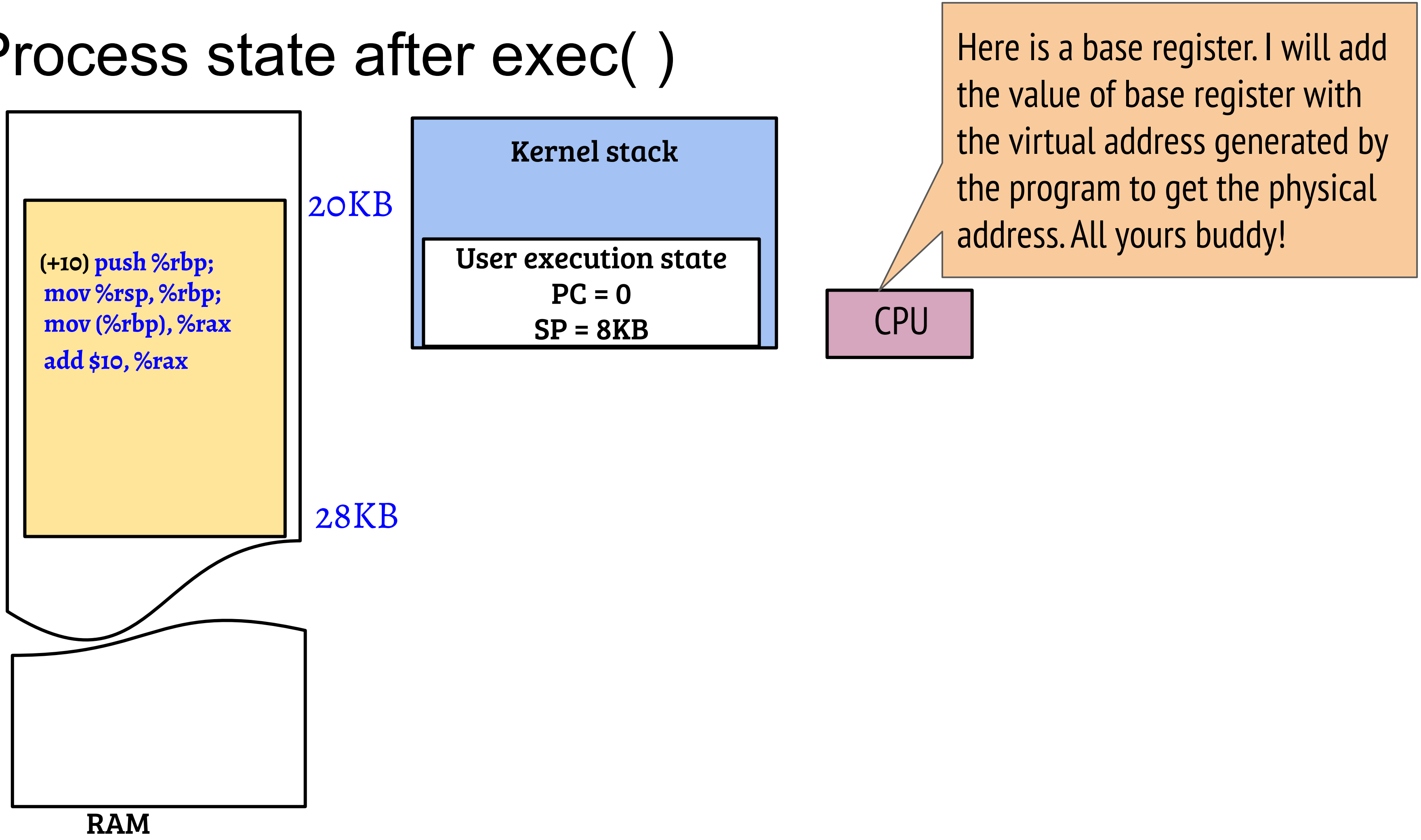




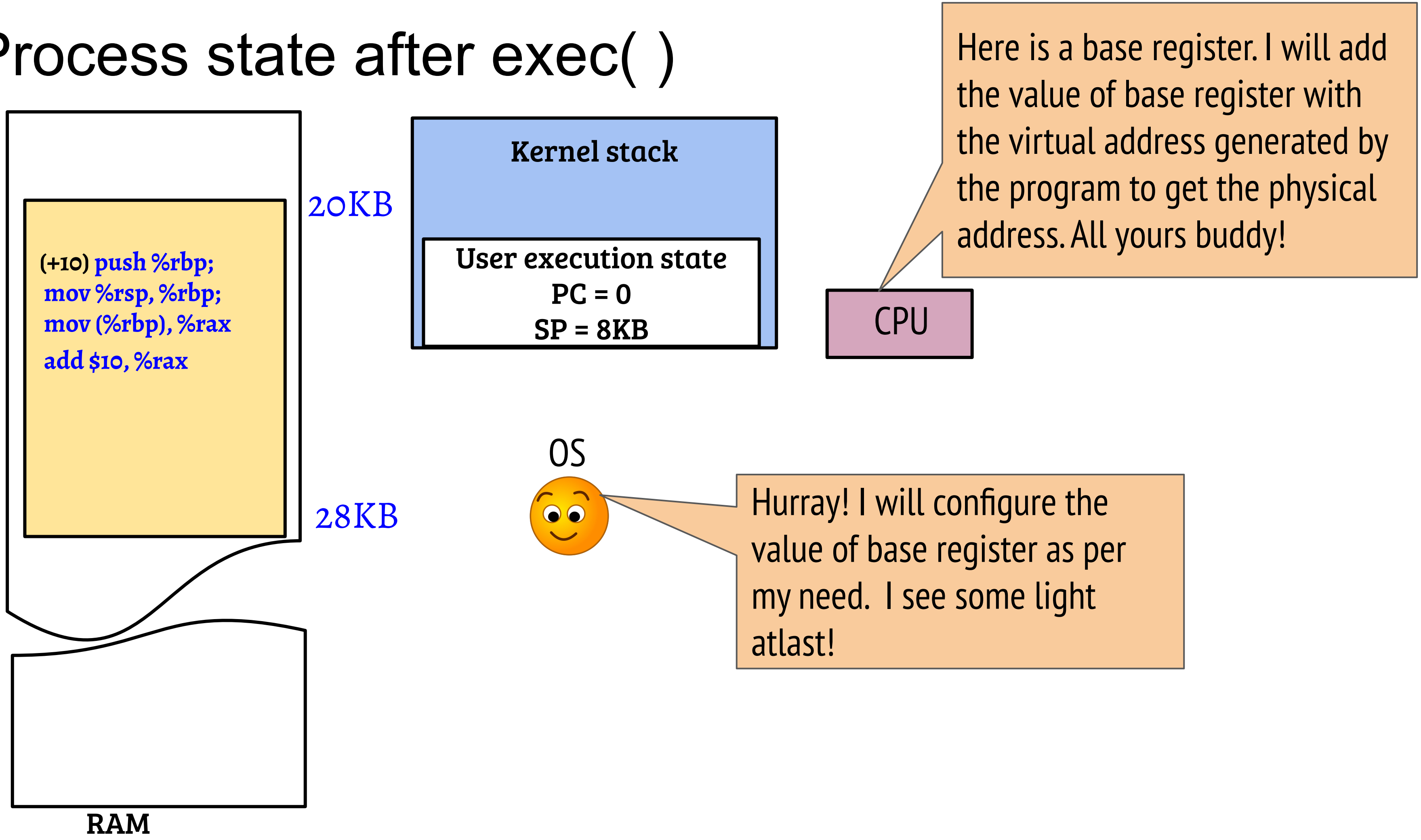


- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of “func” execution
    - Instruction fetch address is 10
    - RSP will be around 8KB

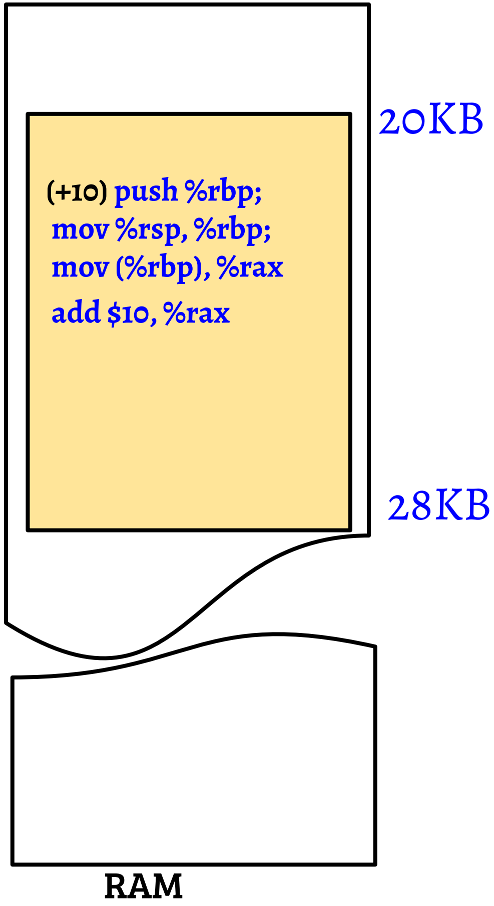
Process state after exec( )

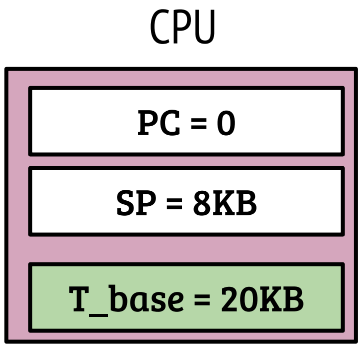


Process state after exec( )



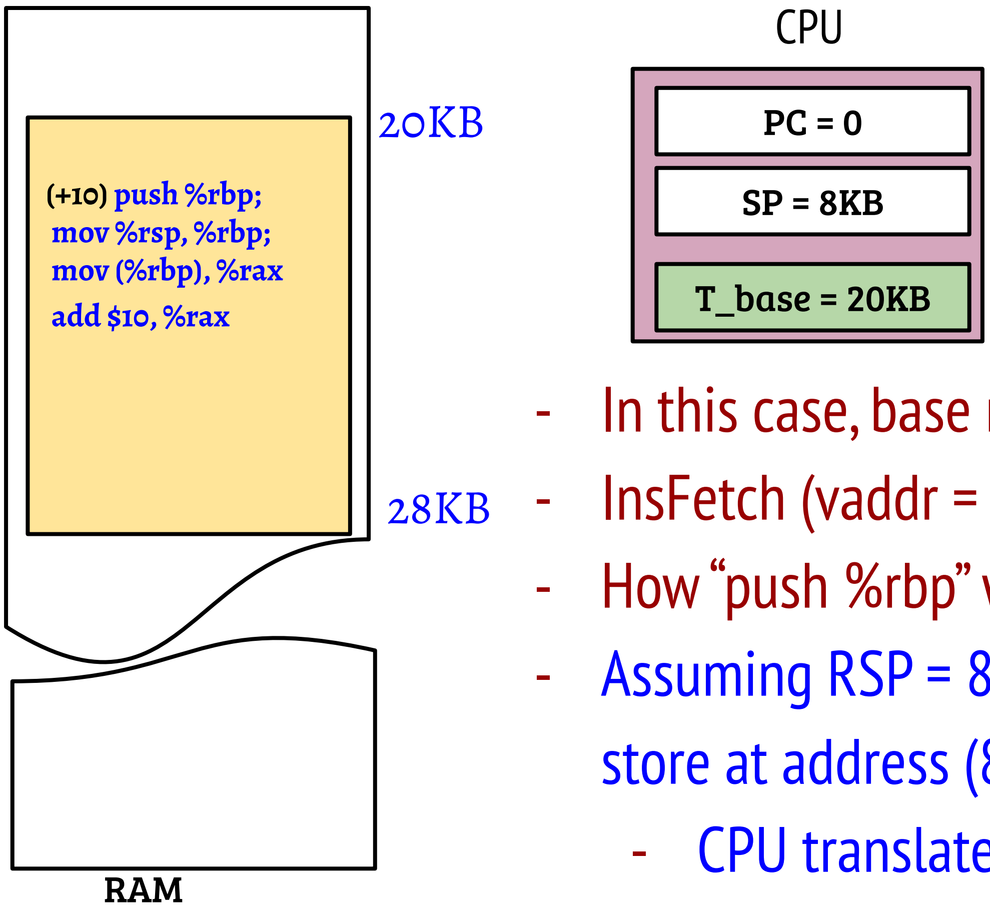
Translation





- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How “push %rbp” works?

Translation

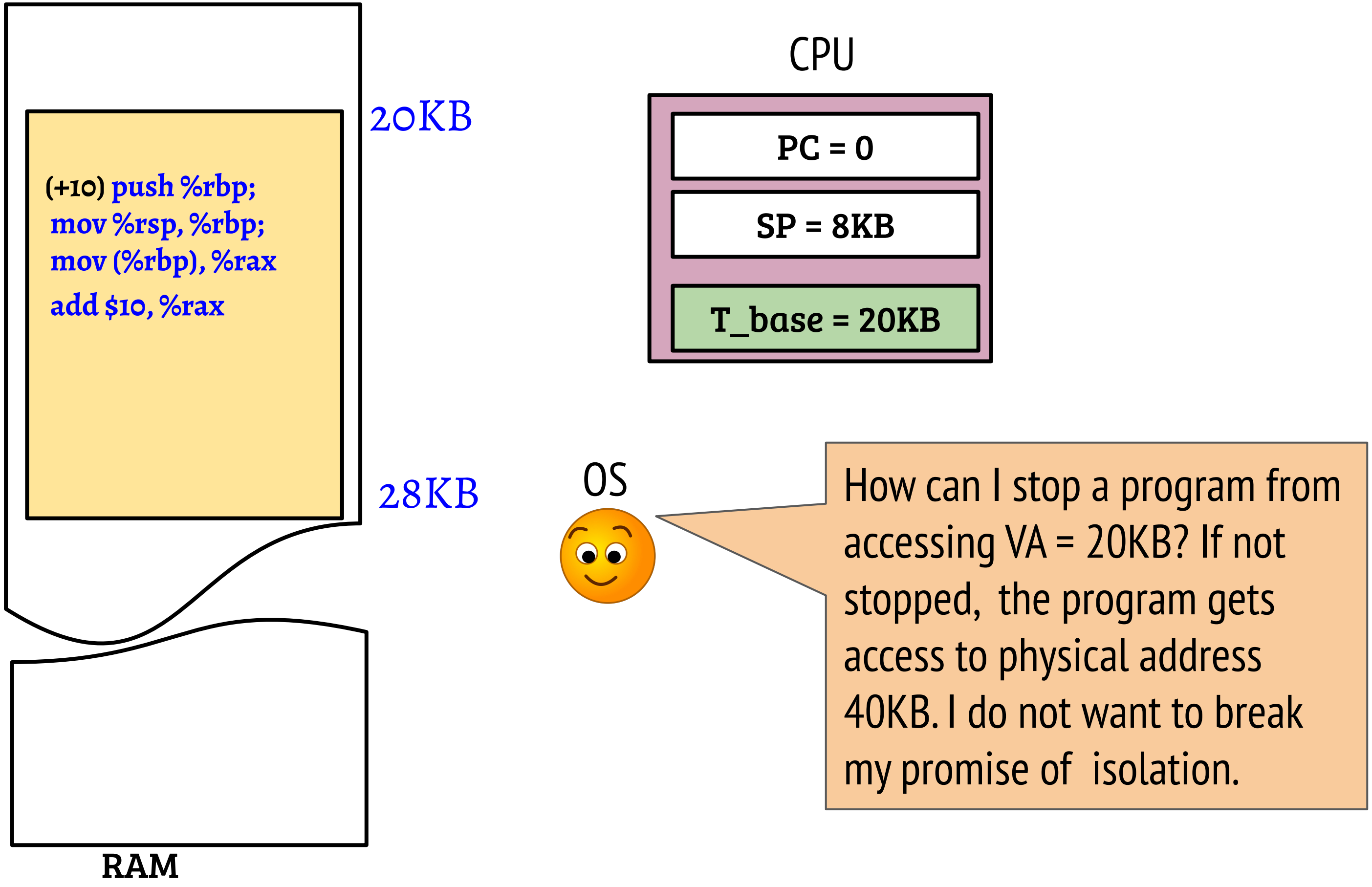


- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How "push %rbp" works?
- Assuming RSP = 8KB, "push %rbp" results in a memory store at address (8KB - 8)
    - CPU translates the address to (28KB - 8)

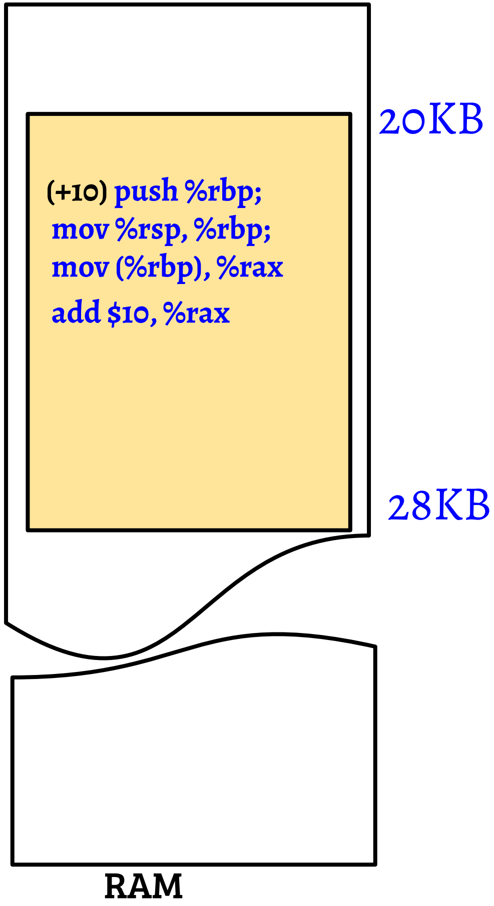
Translation at address space granularity

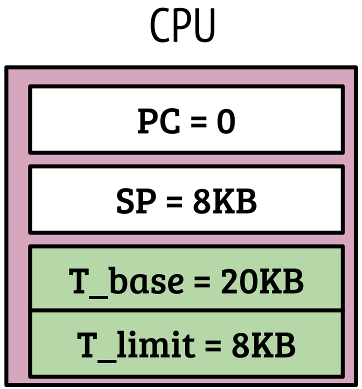
- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location. The hardware performs the translation using the base value.
- How memory isolation is achieved?
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Isolation: How to stop illegal access?



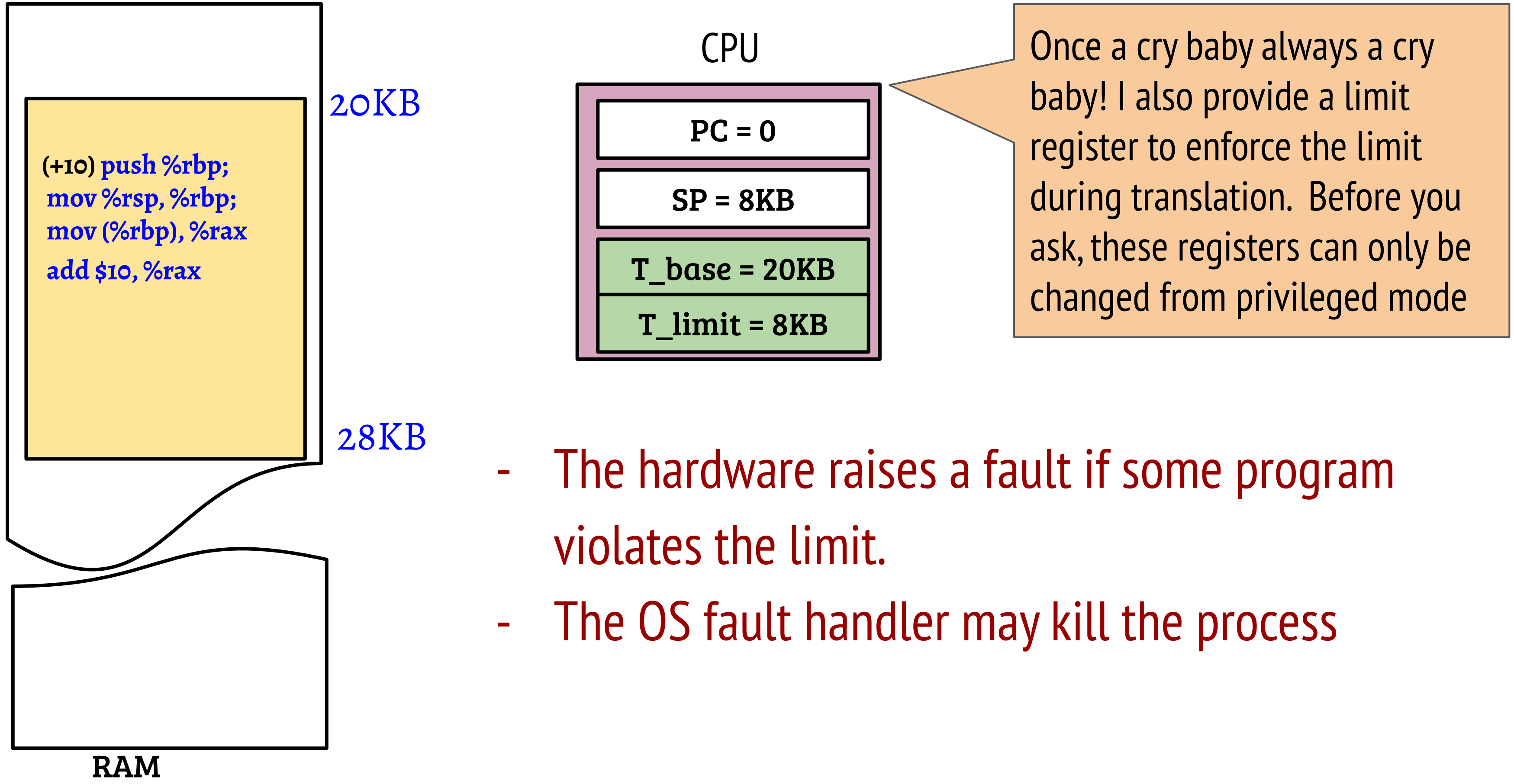
Isolation: How to stop illegal accesses?





Once a cry baby always a cry baby! I also provide a limit register to enforce the limit during translation. Before you ask, these registers can only be changed from privileged mode

Isolation: How to stop illegal accesses?



- The hardware raises a fault if some program violates the limit.
- The OS fault handler may kill the process

Translation at address space granularity

- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location.
  The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Advantages and disadvantages of this scheme

Context switch and translation information

- The base and limit register values can be saved in the outgoing process PCB during context switch
- Loaded from PCB to the CPU when a process is scheduled

Translation at address space granularity

- How virtual address is translated to physical address?
- The OS sets the base register value depending on the physical location.
  The hardware performs the translation using the base value.
- How memory isolation is achieved?
- Limit register can be used to enforce memory isolation
- What happens on a context switch?
- Save and restore limit and base registers
- Advantages and disadvantages of this scheme

Translation at address space granularity: Issues

- Physical memory must be greater than address space size
    - Unrealistic, against the philosophy of address space abstraction
    - Small address space size $\Rightarrow$ Unhappy user

Translation at address space granularity: Issues

- Physical memory must be greater than address space size
  - Unrealistic, against the philosophy of address space abstraction
  - Small address space size ⇒ Unhappy user
- Memory inefficient
  - Physical memory size is same as address space size irrespective of actual usage ⇒ Memory wastage
  - Degree of multiprogramming is very less