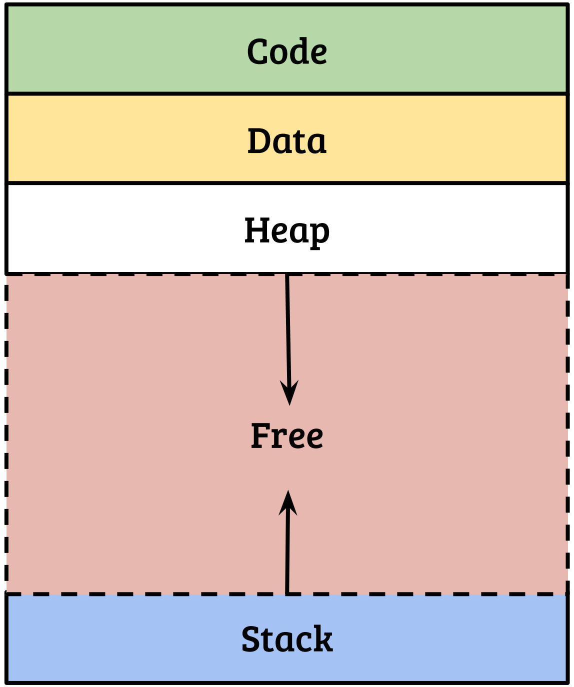
CS330: Operating Systems

Virtual Memory: Address translation

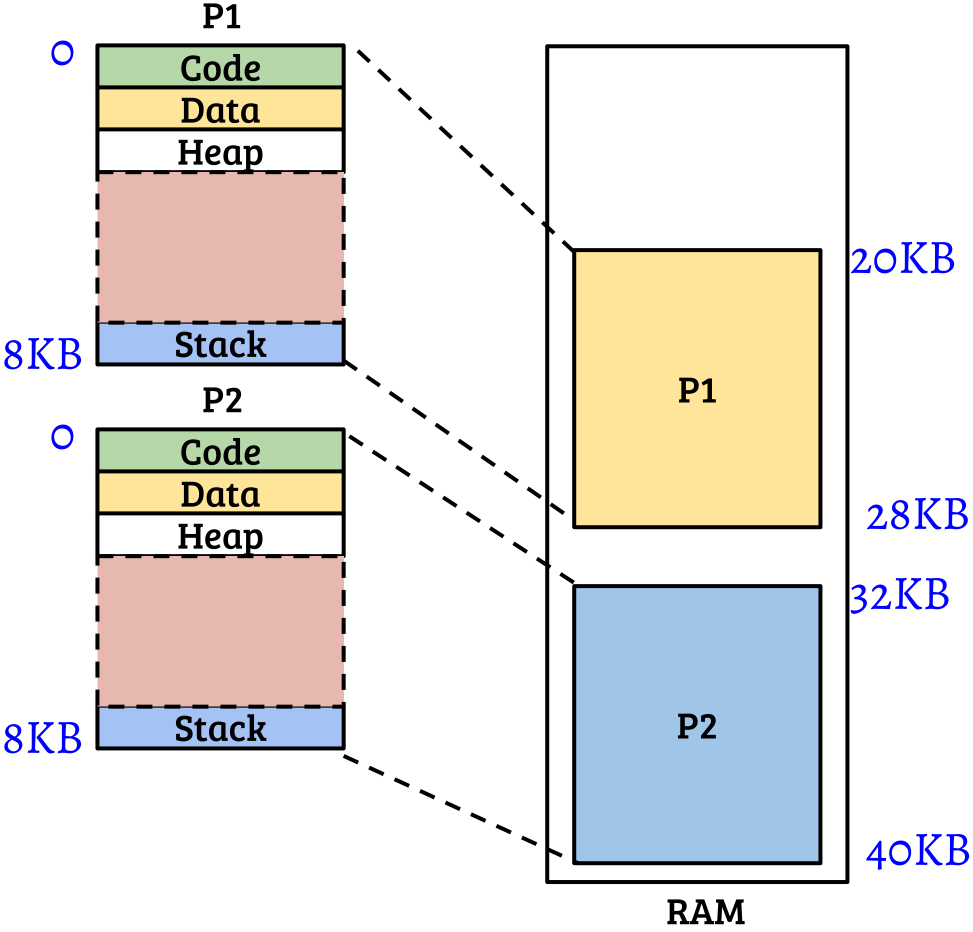
Recap: Process address space



- Address space abstraction provides the same 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!

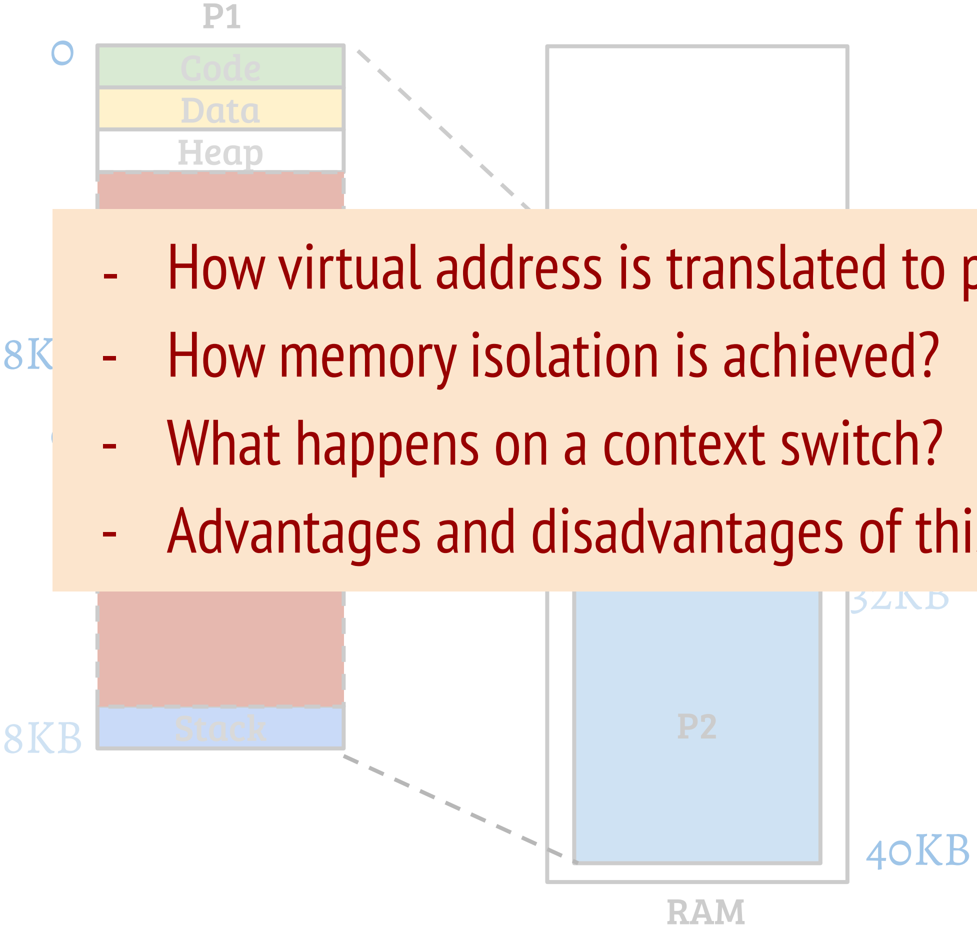
Agenda: Virtual to physical address translation

Translation at address space granularity



- Physical memory of same size as the address space size 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

ISA: commonly used addressing modes (x86)

- At a high-level, instructions contains two parts: opcode and operand
  - ISA defines binary encoding of opcodes, mode and register operands (more complex in practice)
- Operands can be specified in multiple ways
  - Register: mov %rcx, %rax
  - Immediate: mov $5, %rax
  - Absolute: mov 8000000, %rax
  - Indirect: mov (%rcx), %rax
  - Displacement: mov -16(%rbp), %rax

X86 ISA: examples

- Access local variables using %rbp (examples)
- long a = 100, b =20, c;
    - mov $100, -8(%rbp); mov $20, -16(%rbp)
- c = a + b;
    - mov -8(%rbp), %rax; mov -16(%rbp), %rcx;
    - add %rcx, %rax; mov %rax, -24(%rbp)
- PC relative jump/call
    - jmp 0x20(%rip)
    - call -0x20(%rip)

Role of the compiler

Simple function

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

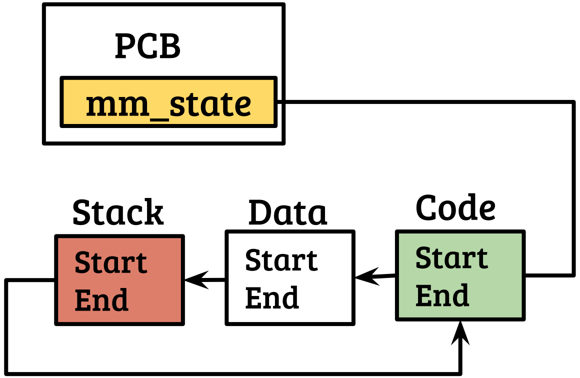
Compiled assembly

```
func:
10:   push %rbp;
12:   mov %rsp, %rbp;
16:   mov $100, -8(%rbp);
20:   mov -8(%rbp), %rax
24:   add $10, %rax
29:   mov %rax, -8(%rbp)
33:   pop %rbp;
35:   ret;
```

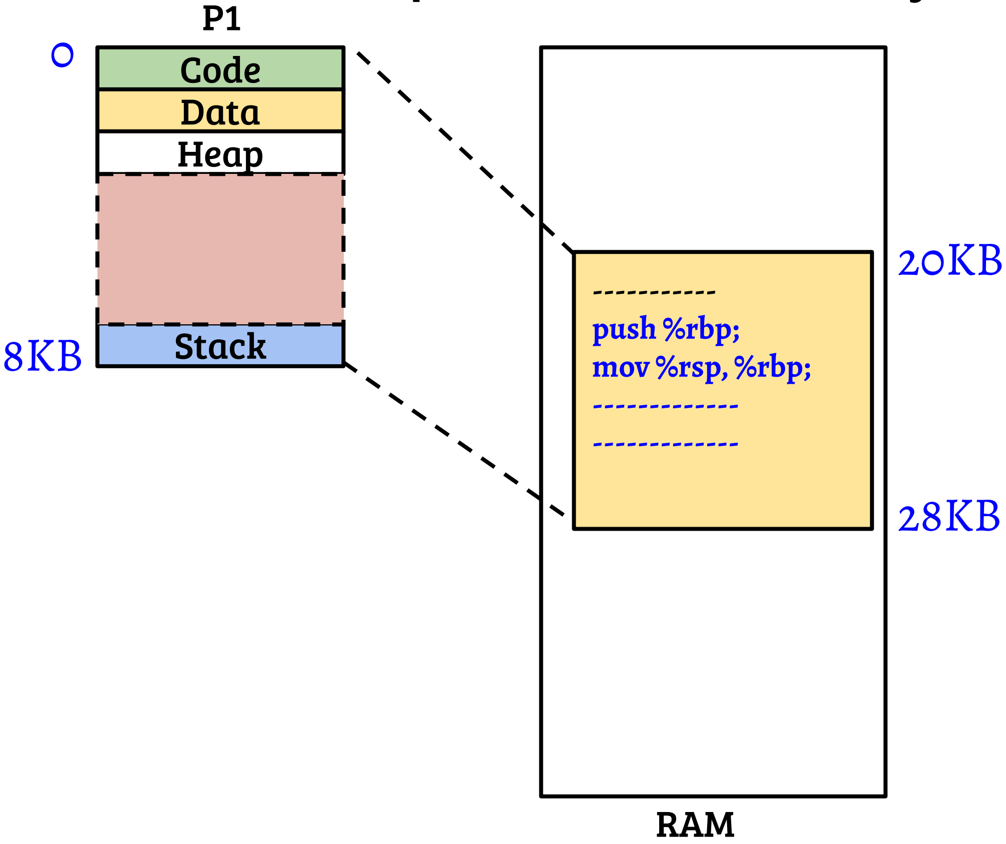
- Compiler can genera
  code assuming start
  the code address as
- Compiler does not k
  the stack address, bl
  uses the registers (r

OS during binary load (simplified fork + exec)

```
load_new_executable( PCB *current, File *exe)
{
  verify_executable(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;
}
```

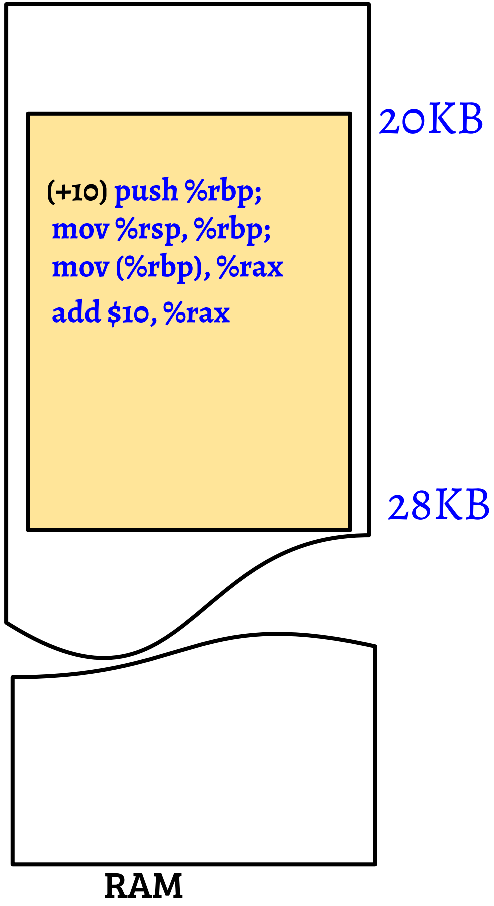


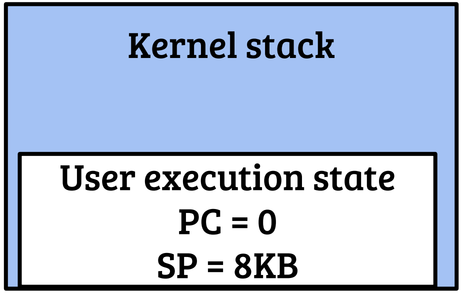
Address space to memory translation



- Physical memory of 8KB is allocated and the code is loaded
- The PCB memory state is updated based on the executable format

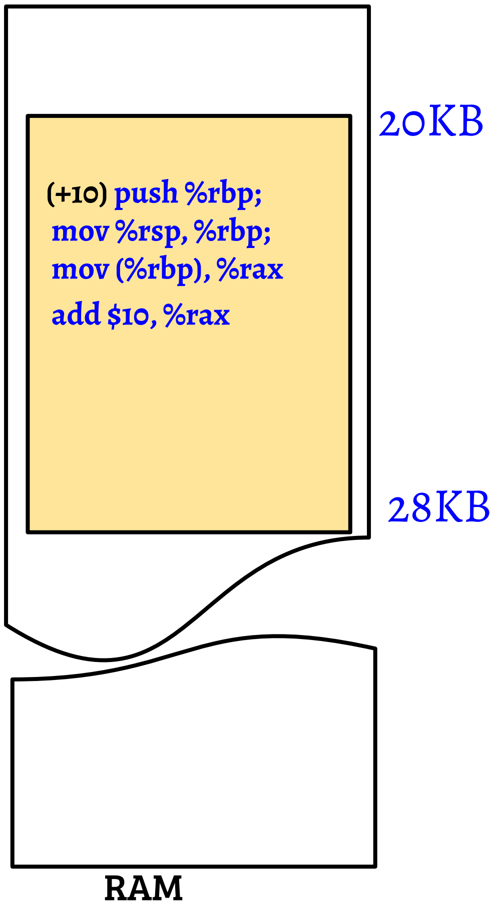
Process state after exec( )

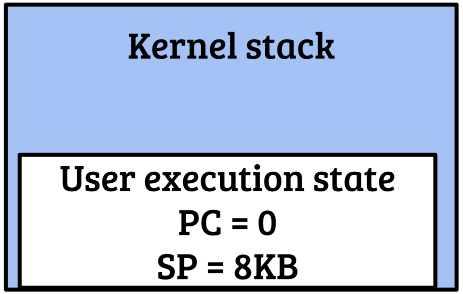


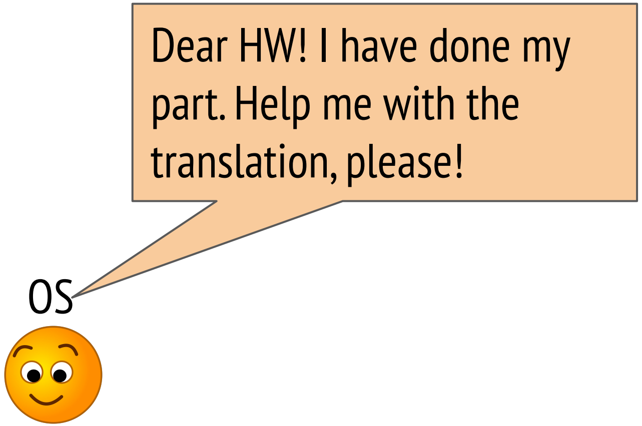


- When the process returns to user space, the registers are loaded with virtual addresses
- PC = 0 and SP = 8KB

Process state after exec( )

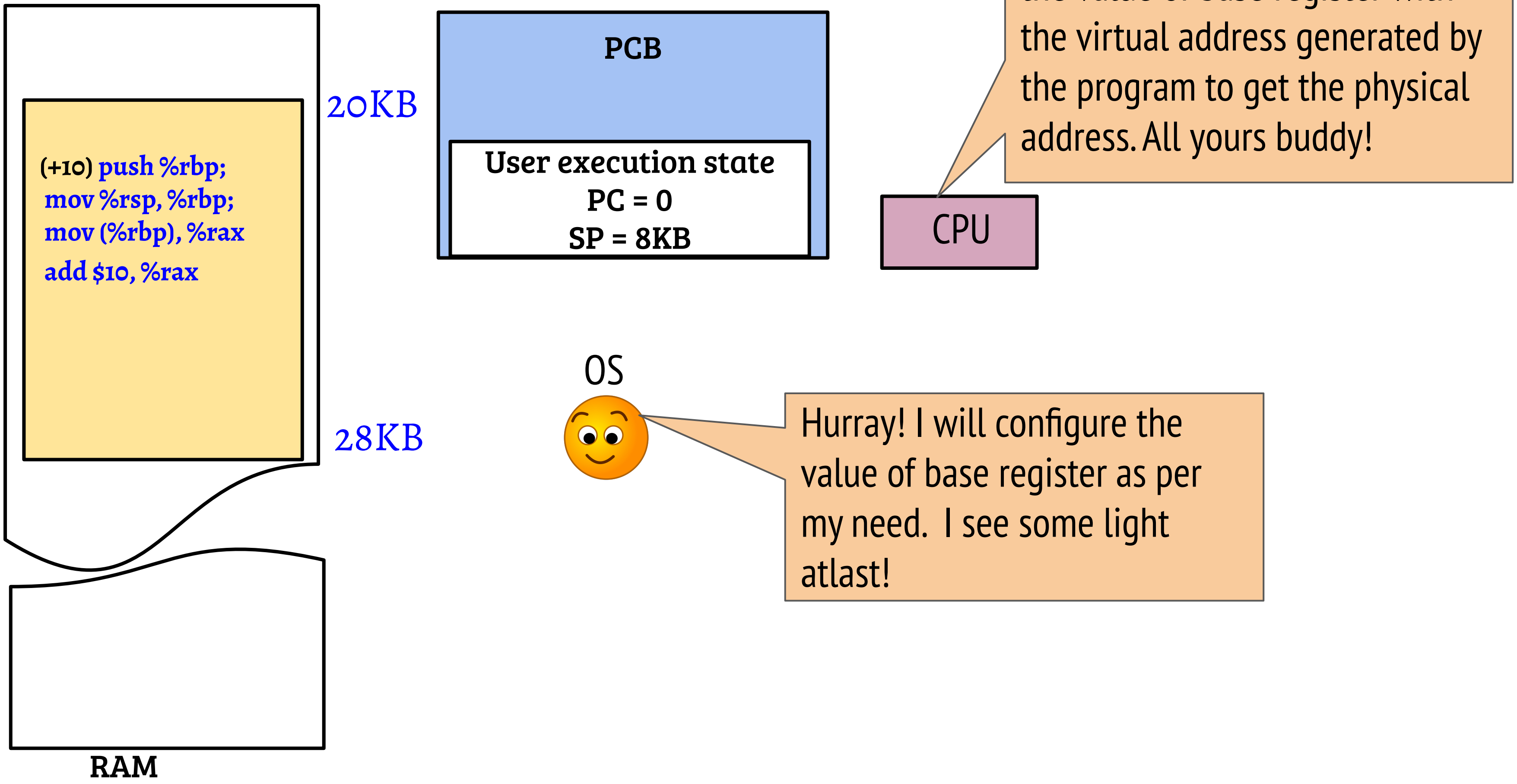




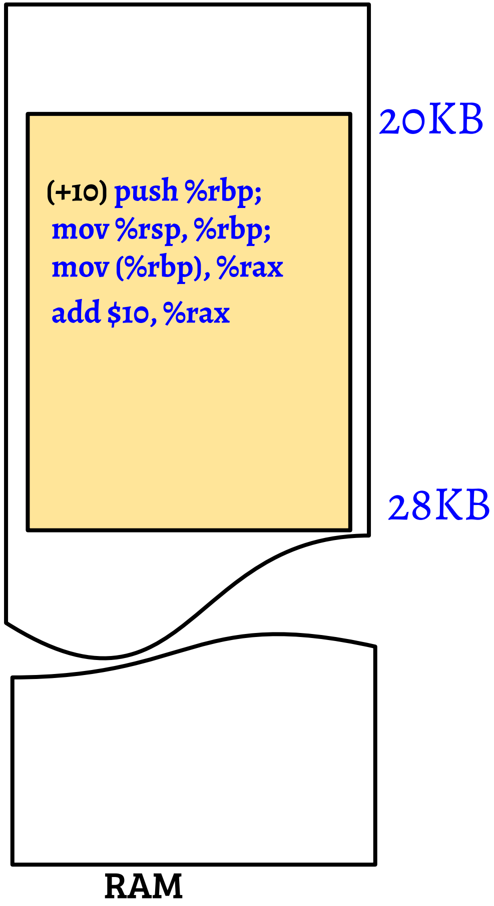


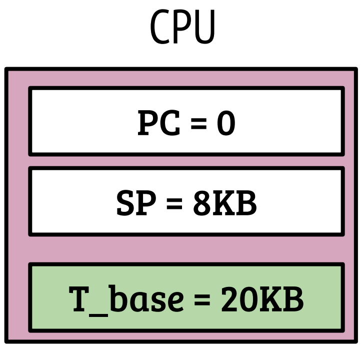
- When the process returns to user space, the registers are loaded with virtual addresses
- Code is loaded into physical memory (@20KB)
- At the start of "func" execution
  - Instruction fetch address is 10
  - SP will be around 8KB

Process state after exec( )



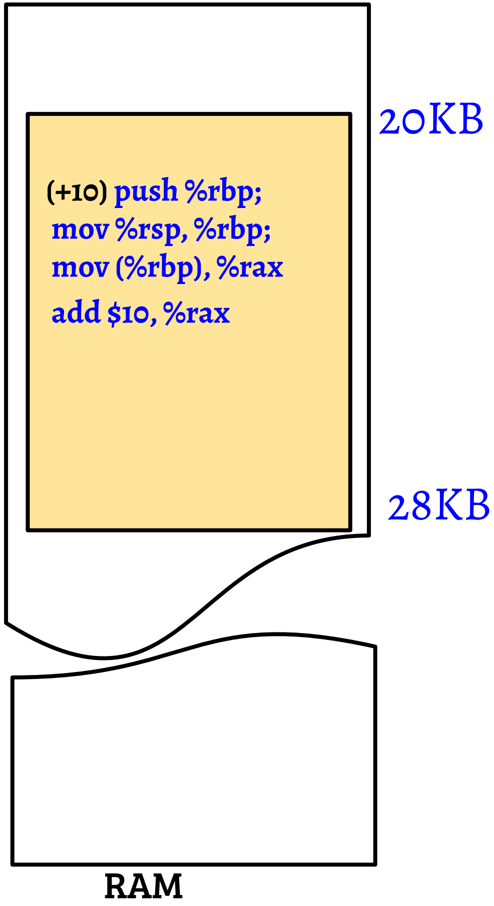
Translation

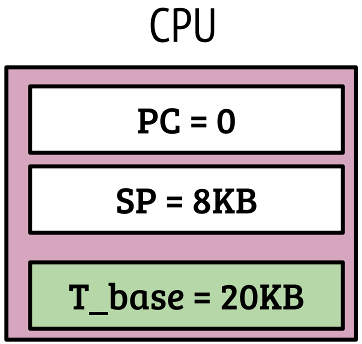




- In this case, base register value should be 20KB
- InsFetch (vaddr = 10) ⇒ InsFetch (paddr = 20KB +10)
- How "push %rbp" works?
- Assuming RSP = 8KB, "push %rbp" results in a memory store at address (8KB - 8)
  - CPU translates the address to (28KB - 8)

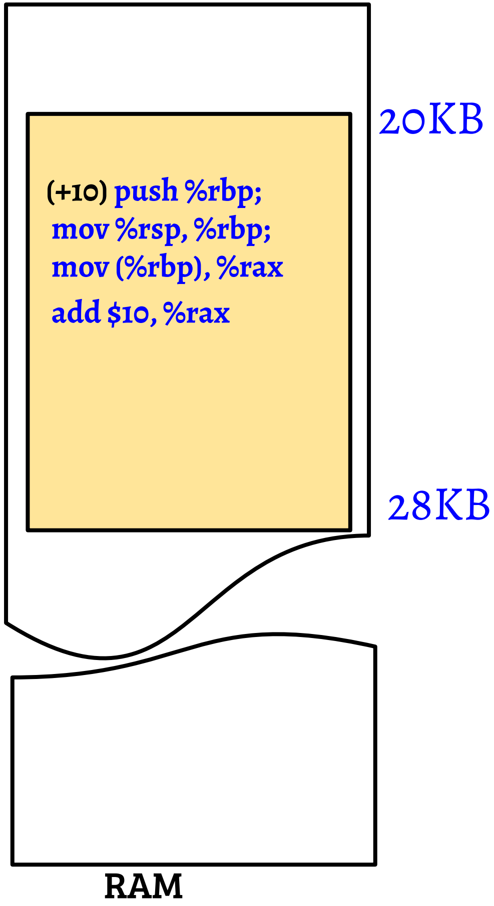
Isolation: How to stop illegal access?

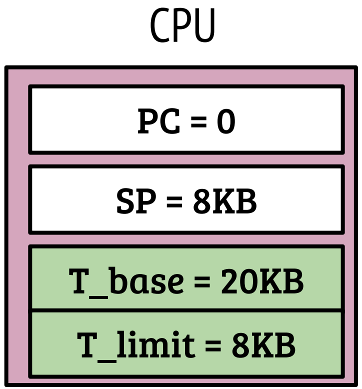




How can I stop a program from accessing VA = 20KB? If not stopped, the program gets access to physical address 40KB. I do not want to break my promise of isolation.

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

- The hardware raises a fault if some program violates the limit.
- The OS fault handler may kill the process
- (WE-1) T_base and T_limit values across processes

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
- (WE-2) User-to-OS context switching

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 ⇒ 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