Virtual Memory

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

Agenda

- Address spaces
- Segmentation
- Paging
- Swapping
- User space memory management
- Case study: Linux+x86 VM

§ Address spaces

Programs → Processes

- Compilers transform programs into binary images that contains executable machine code and static data (e.g., constants/globals)
- The kernel can turn these binaries into active, running processes
  - Via limited direct execution and the scheduler/dispatcher, multiple processes can run concurrently on timeshared CPUs
  - But how do processes share memory?
  - How are memory addresses encoded into programs?

```c
#include <stdio.h>

unsigned int glob = 0xdeadbeef;

int main() {
  printf("0x%lx\n", (unsigned long)&main);
  printf("0x%lx\n", (unsigned long)&glob);
  printf("0x%x\n", glob);
  return 0;
}
```

```
0x40052d
0x601040
0xdeadbeef
```

```
> objdump -d a.out
000000000040052d <main>:
  40052d:   55                      push   %rbp
  40052e:   48 89 e5                mov    %rsp,%rbp
  400531:   b8 2d 05 40 00          mov    $0x40052d,%eax
  400536:   48 89 c6                mov    %rax,%rsi
  400539:   bf 04 06 40 00          mov    $0x400604,%edi
  40053e:   b8 00 00 00 00          mov    $0x0,%eax
  400543:   e8 c8 fe ff ff          callq  400410
  400548:   b8 40 10 60 00          mov    $0x601040,%eax
  40054d:   48 89 c6                mov    %rax,%rsi
  400550:   bf 04 06 40 00          mov    $0x400604,%edi
  400555:   b8 00 00 00 00          mov    $0x0,%eax
  40055a:   e8 b1 fe ff ff          callq  400410
  40055f:   8b 05 db 0a 20 00       mov    0x200adb(%rip),%eax # 601040 <glob>
  400565:   89 c6                   mov    %eax,%esi
  400567:   bf 0b 06 40 00          mov    $0x40060b,%edi
  40056c:   b8 00 00 00 00          mov    $0x0,%eax
  400571:   e8 9a fe ff ff          callq  400410
  400576:   b8 00 00 00 00          mov    $0x0,%eax
  40057b:   5d                      pop    %rbp
  40057c:   c3                      retq
```

text (code) addresses

global data address

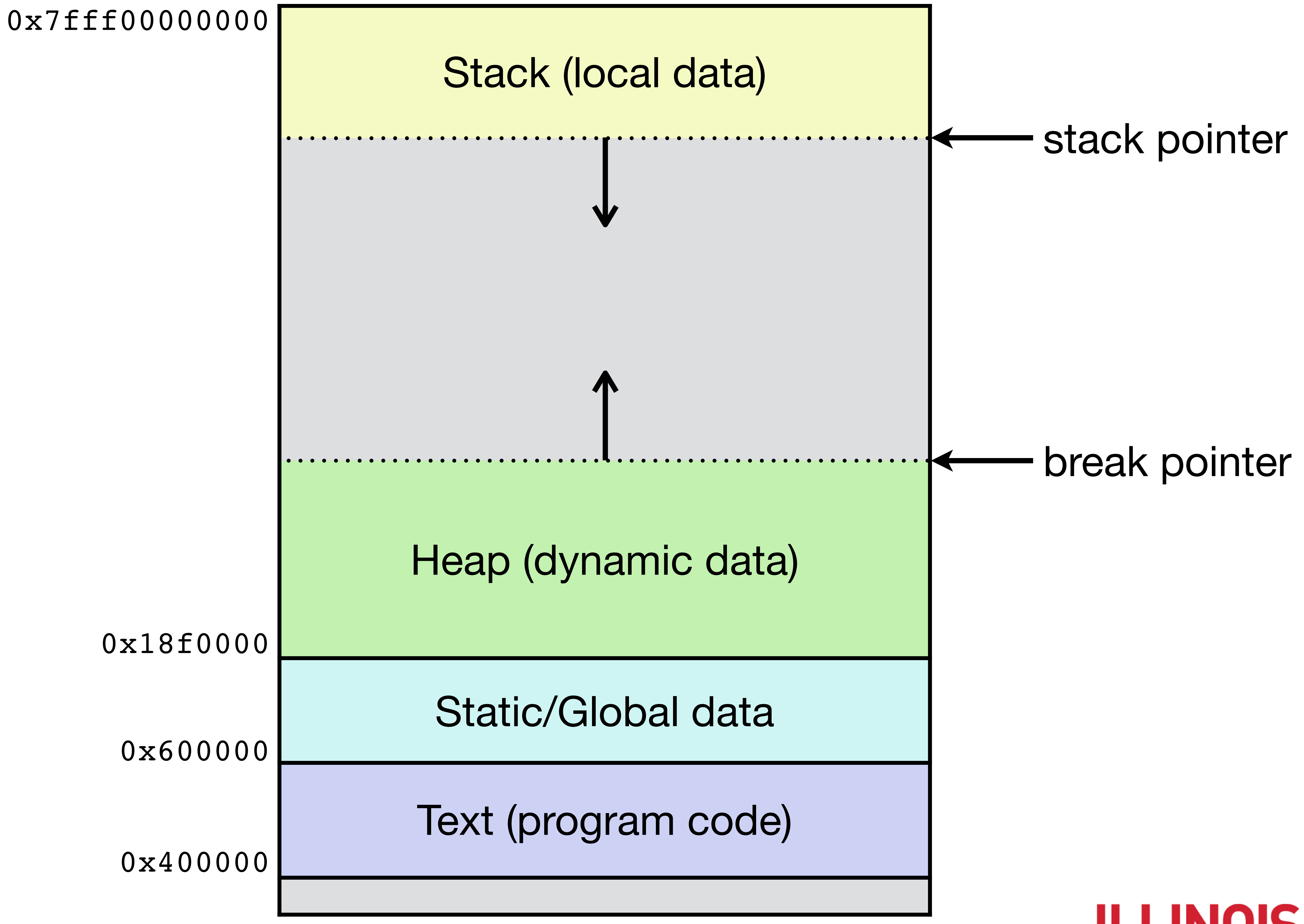
```
> objdump -h a.out
a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
 12 .text         000001b2  0000000000400440  0000000000400440  00000440  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 23 .data         00000014  0000000000601030  0000000000601030  00001030  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          00000004  0000000000601044  0000000000601044  00001044  2**0
                  ALLOC
```

“Hardcoded” addresses

- At compile time, the linker embeds fixed addresses into the binary
  - E.g., for function calls, global variables/constants, jump tables, etc.
- When `exec`-ed, the OS loads sections of the binary into memory at these pre-established locations
  - Text & Data sections are initialized with contents of binary
  - BSS section is zero-initialized
  - Starting addresses for initially empty stack & heap are also established

Uniform process address space



Address space = a lie!

- If processes are simultaneously accessing physical memory …
  - Not all text sections can begin at `0x400000`
  - Not all data sections can begin at `0x600000`
  - Not all heaps can begin at `0x18f0000`
  - Not all stacks can begin at `0x7fff00000000`
- Uniform process address spaces are an illusion created by the kernel
  - To simplify program loading and execution (among other reasons)

Timesharing?

- Can we timeshare memory like we do the CPU?
  - Yes, but …
    - Need to swap process address space contents between disk and memory on every context switch
    - Prohibitively expensive!!!
- May work for non-preemptive FCFS/batch systems where processes are expected to use up most or all of physical memory
- Space-sharing is the only generally viable (performant) solution

Software relocation

- Simple option: rewrite all addresses at load-time (in software), so that processes can occupy memory simultaneously (space-sharing)

```
000000000040052d <main>:
  40052d:   55                      push   %rbp
  40052e:   48 89 e5                mov    %rsp,%rbp
  400531:   b8 2d 05 40 00          mov    $0x40052d,%eax
  400536:   48 89 c6                mov    %rax,%rsi
  400539:   bf 04 06 40 00          mov    $0x400604,%edi
  ...
```

```
000000000060052d <main>:
  60052d:   55                      push   %rbp
  60052e:   48 89 e5                mov    %rsp,%rbp
  600531:   b8 2d 05 60 00          mov    $0x60052d,%eax
  600536:   48 89 c6                mov    %rax,%rsi
  600539:   bf 04 06 60 00          mov    $0x600604,%edi
  ...
```

- Issues?

Software relocation

- Once loaded, cannot easily relocate process in memory
  - Software relocation would be complex and time-consuming (and perhaps impossible, without runtime type information)
- If a process accidentally/maliciously uses a bad address, it could access another process's (or the kernel's) address space!
  - Pure software relocation makes address space *protection* difficult

Hardware address translation

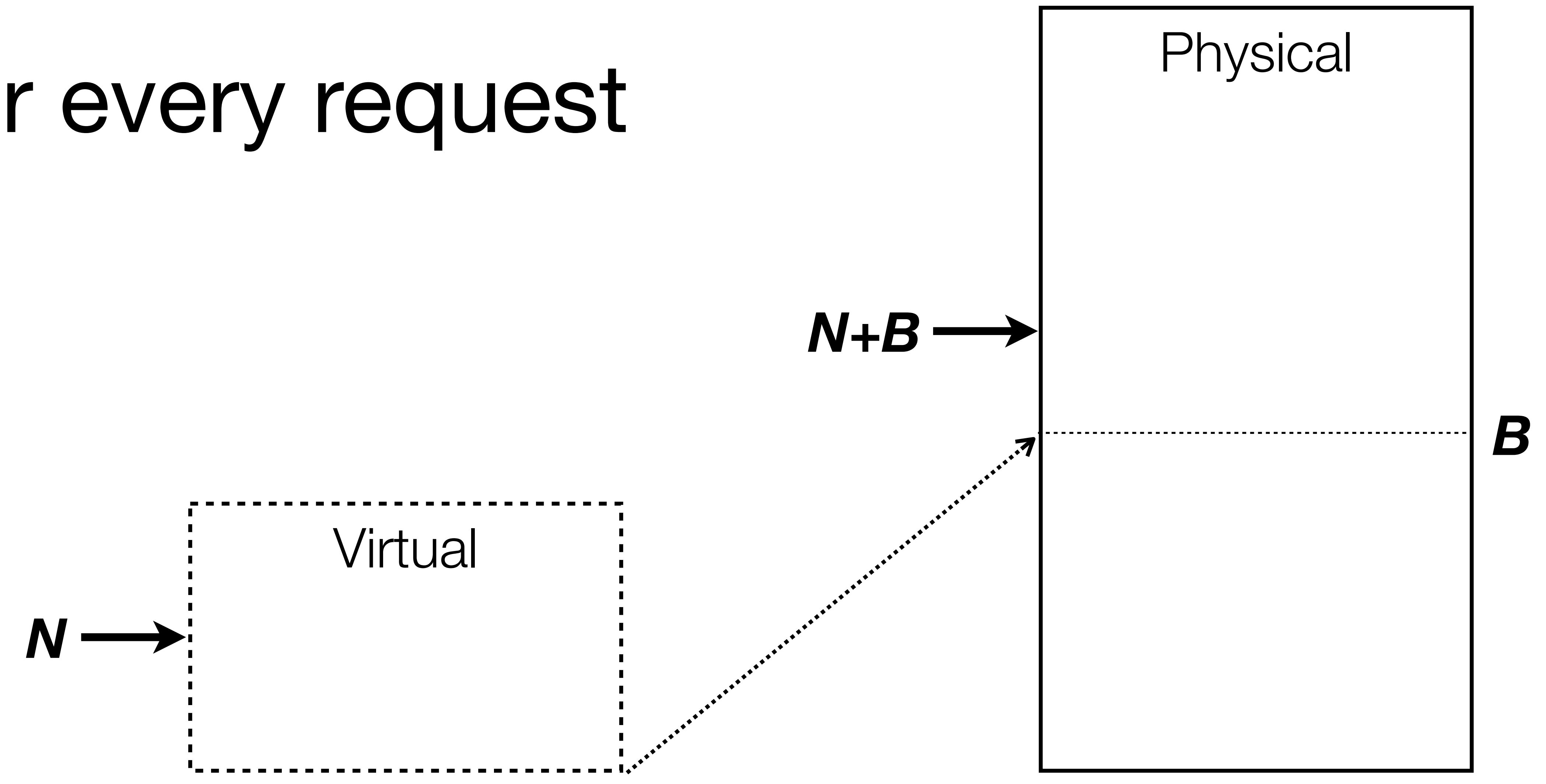
- To support fast, dynamic translation and address space protection, hardware support for address translation is needed
- Idea:
  - All process memory requests (on the CPU) are for **virtual** addresses
  - Hardware translates each request to a **physical** address
    - Kernel sets up mapping from virtual address spaces to physical memory
    - Translation hardware is the memory management unit (MMU)

Primary goals

- Transparency
  - Processes aren’t involved in (or aware of) the translation process
- Efficiency
  - Time (speed/throughput) & Space (utilization)
- Protection
  - Processes cannot access data outside their own address spaces
    - Isolated from each other and the kernel

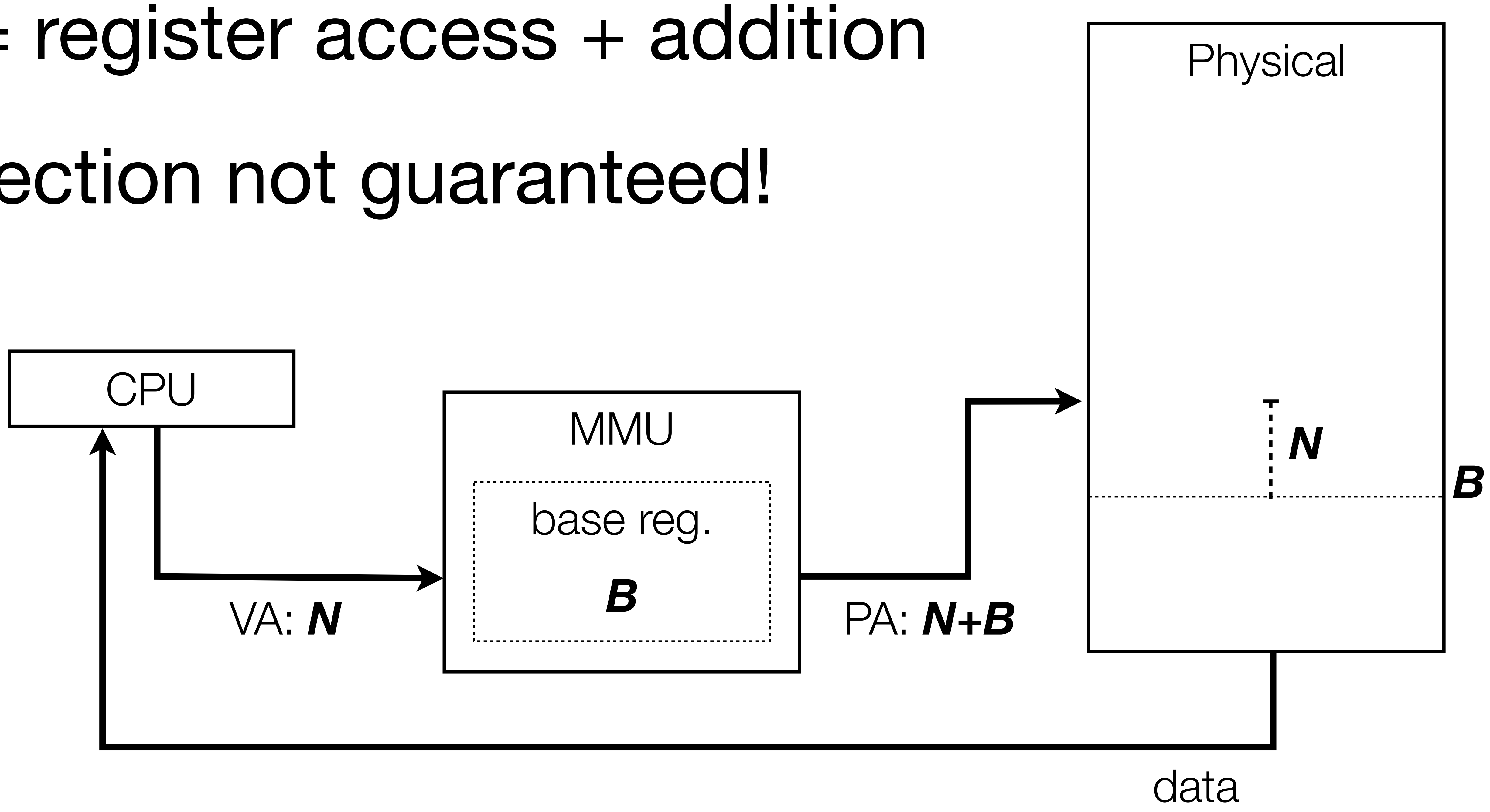
Simple implementation: Relocation

- Assumptions:
  - Fixed size process address spaces
  - Process address space < Physical memory
- Relocation requires a uniform shift for every request



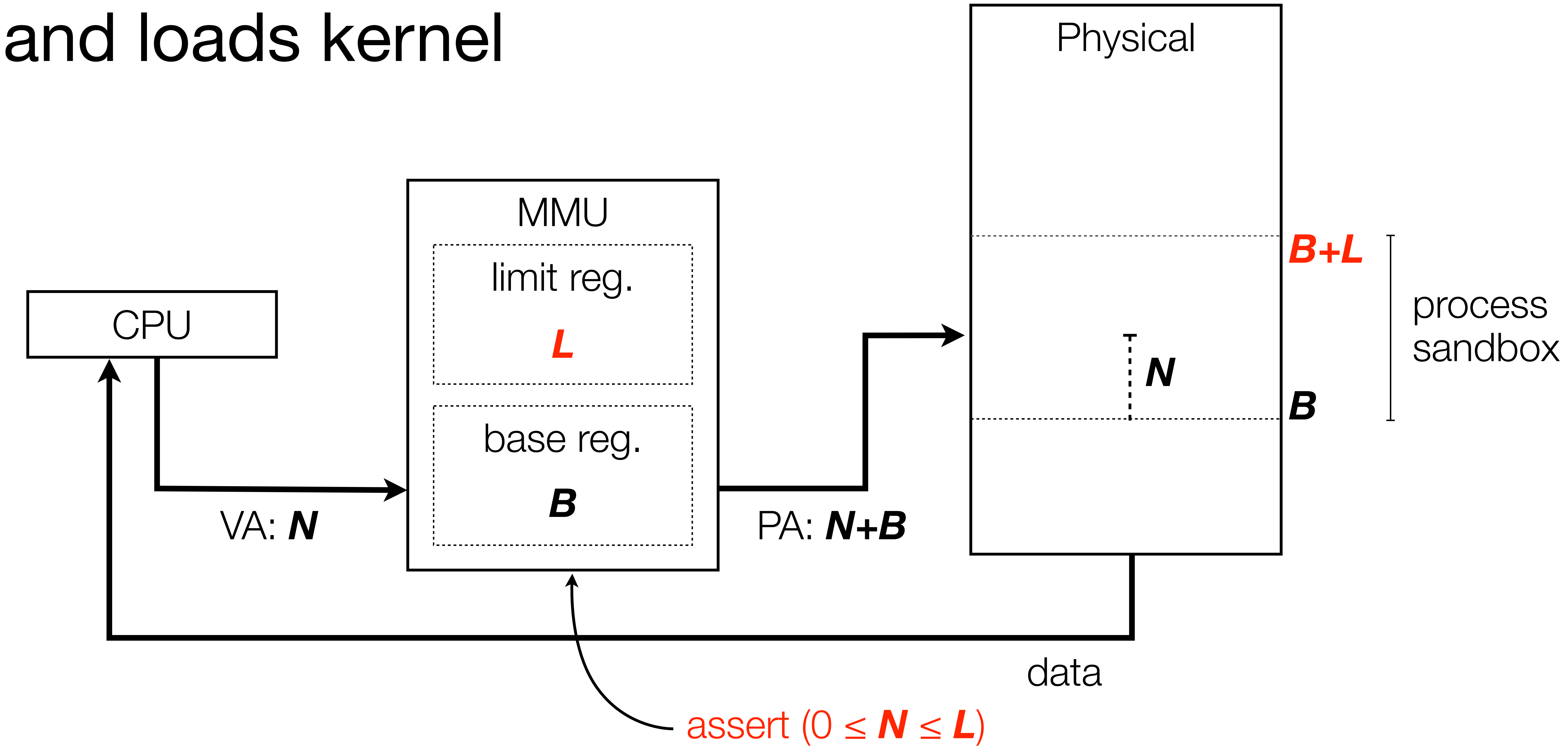
Base address

- Kernel maintains base address of each process in PCB
  - Load into base (address) register in MMU on each context switch
  - Relocation = register access + addition
- Problem: protection not guaranteed!



Base + Limit registers

- Incorporate a limit register to enforce memory protection
- Assertion failure triggers fault (software exception) and loads kernel



Analysis

- **Fast translation** via hardware relocation
  - Register access & addition/comparison
- **Protection** is enforced
- But address spaces are **mapped monolithically** — i.e., unused portions reserve physical space
  - Results in **poor utilization**