Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 11: Segmentation and Paging

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Midterm.
2. Virtual memory.
3. Segmentation.
4. Paging.

Agenda

1. Midterm.
2. Virtual memory.
3. Segmentation.
4. Paging.

Midterm exam

Online using *Crabster.org* Wed Jun 24 3:00 to 5:00 pm EDT.

If you need an accommodation, please let us know soon.

Material for midterm:

1. All the lecture topics from start until end of lecture 9 on deadlock.
2. All the labs on these topics.
3. Projects 1 and 2.

Midterm exam

Two sample exams posted on web page.

Solutions in lab section this Friday.

Review session Sat Jun 20 12:00 noon to 3:00 pm EDT.

Agenda

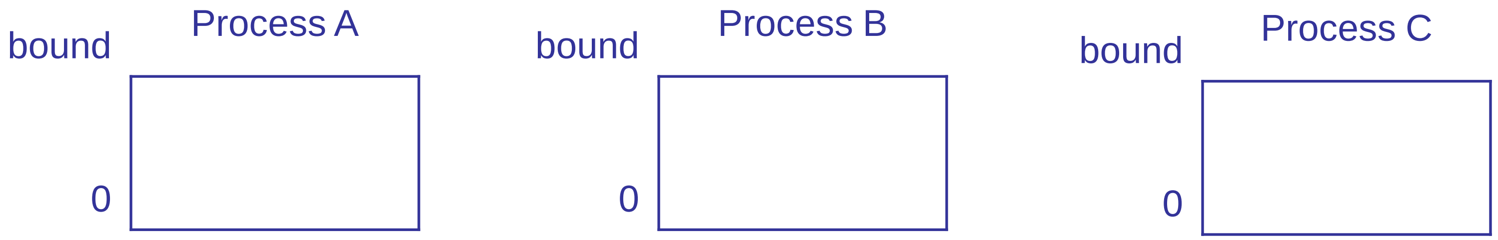
1. Midterm.
2. Virtual memory.
3. Segmentation.
4. Paging.

Address Spaces

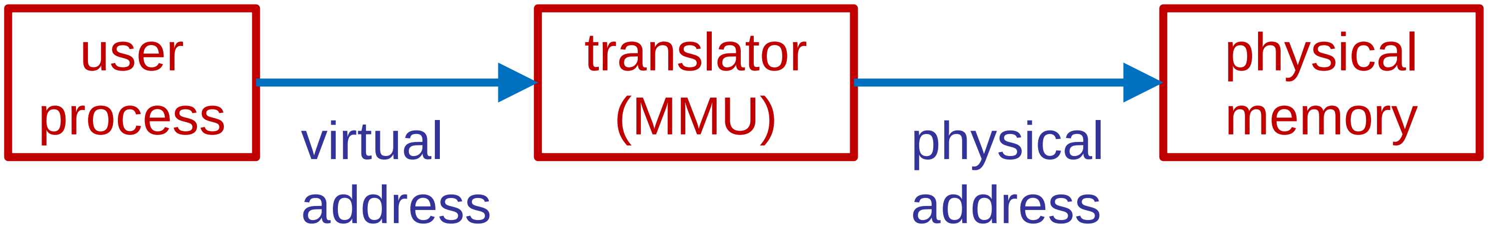
Hardware interface:

- All processes share physical memory

OS abstraction:



Dynamic address translation



Address independence

Virtual addresses are scoped to 1 process.

Protection

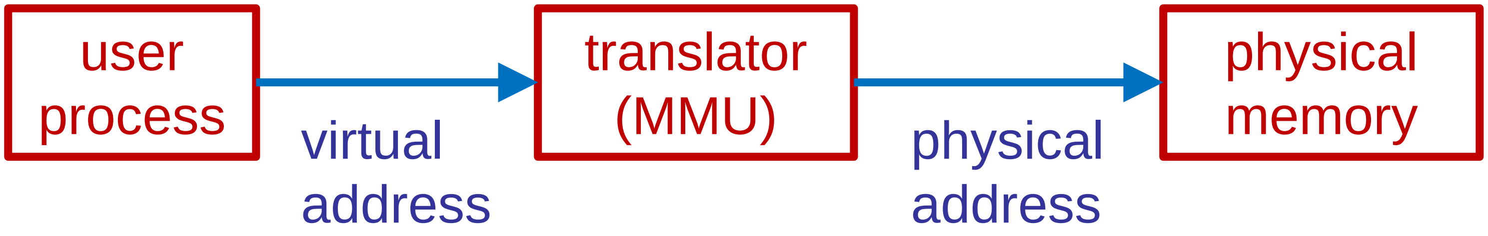
One process can't refer to another's address space.

Virtual memory

VA only needs to be in physical mem. when accessed.

Allows changing translations on the fly.

Dynamic address translation

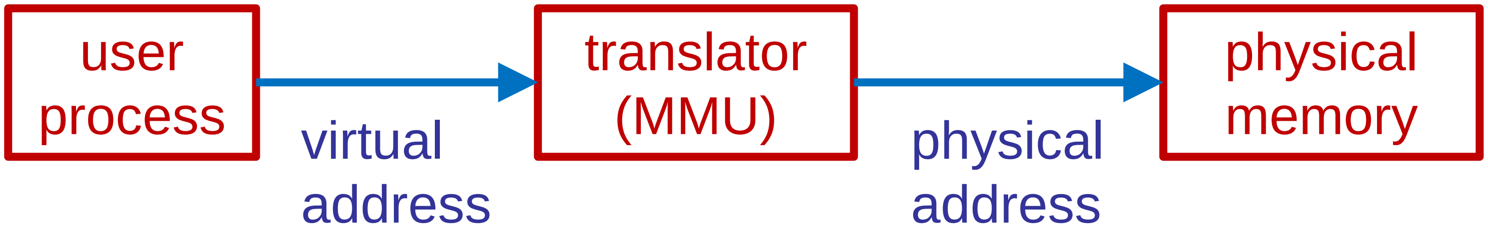


Many ways to implement the translator.

Tradeoffs

1. Flexibility (sharing, growth, virtual memory)
2. Size of data needed to support translation
3. Speed of translation

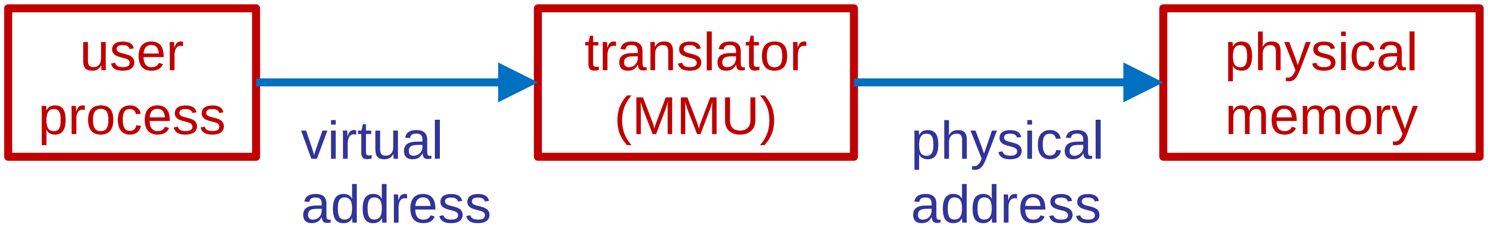
Dynamic address translation



MMU strategies we'll discuss:

1. Base and bounds.
2. Segmentation.
3. Paging.

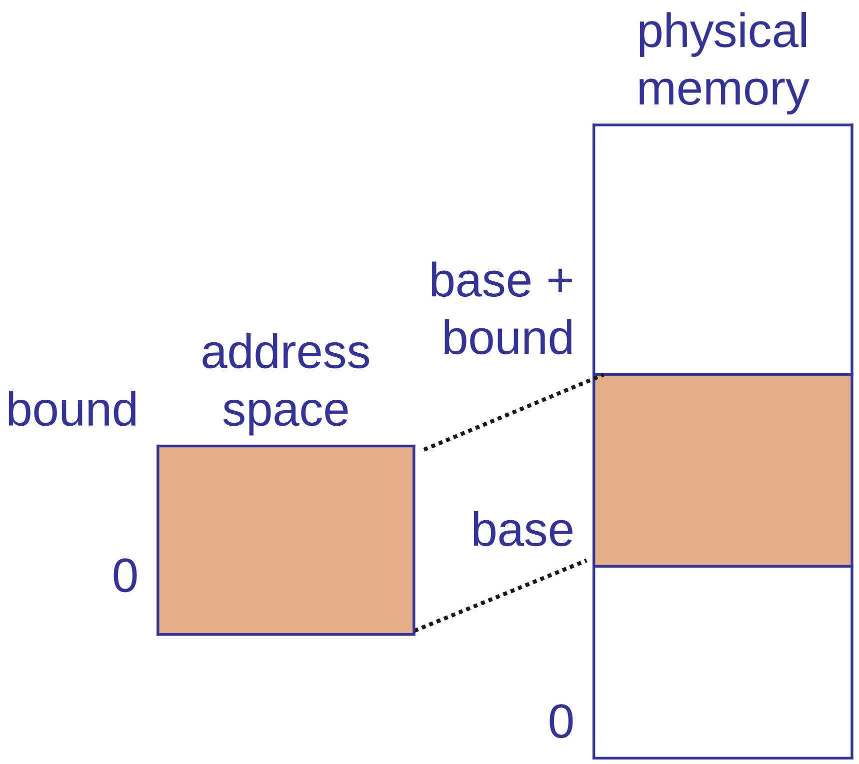
Dynamic address translation



MMU strategies we'll discuss:

1. Base and bounds.
2. Segmentation.
3. Paging.

Base and bounds



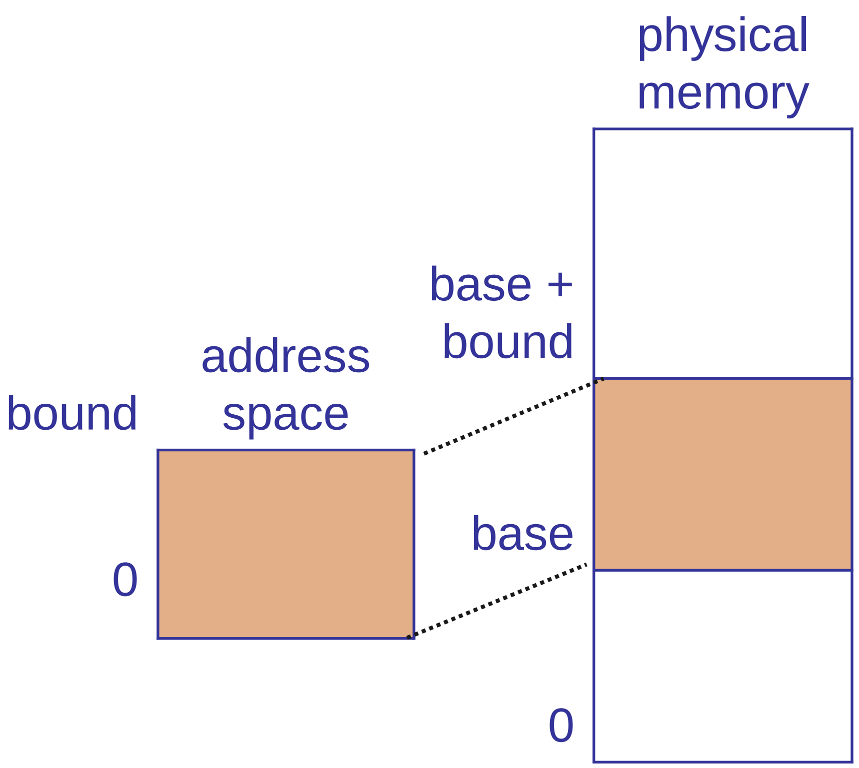
Load each process into a contiguous region of physical memory.

Prevent process from accessing data outside its region.

Base register: starting physical address.

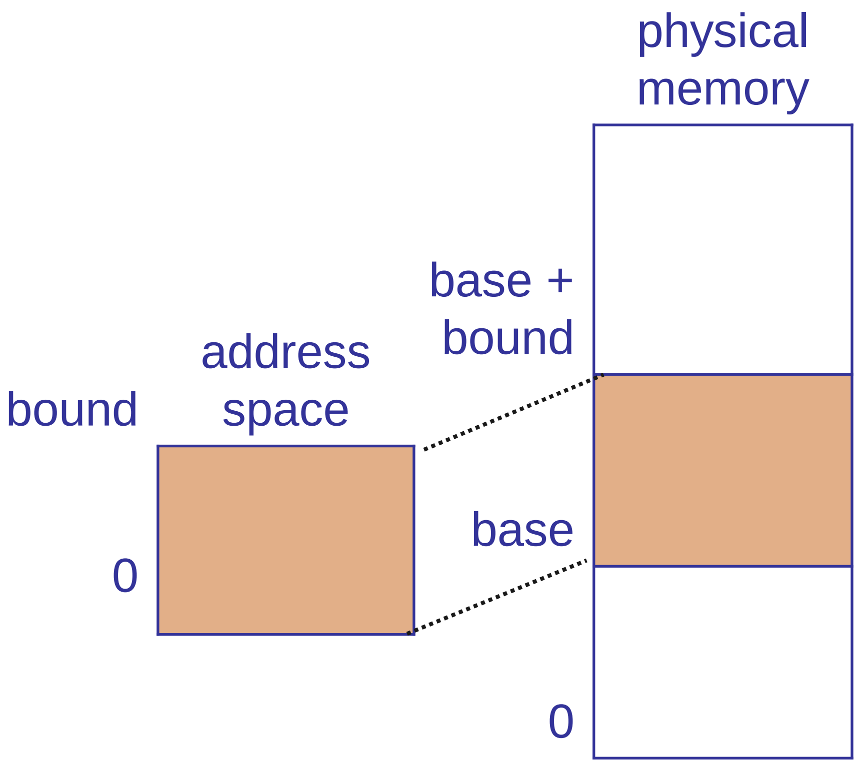
Bound register: size of region.

Base and bounds




```
MMU translation( )
   {
   if ( virtual address > bound )
      {
      trap to the kernel;
      (probably) kill the
         process (core dump);
      }
   else
      physical address = base +
         virtual address;
   }
```

Base and bounds



Pros:

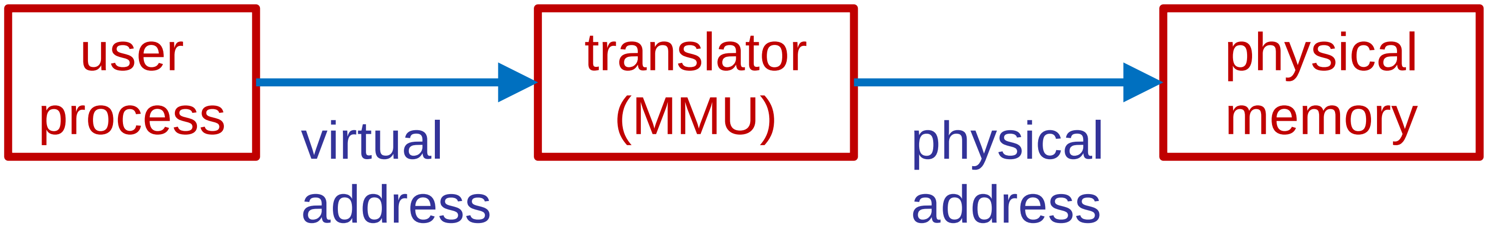
1. Fast.
2. Simple hardware support.

Cons:

1. No virtual memory.
2. External fragmentation.
3. Hard to selectively grow parts of address space.
4. No controlled sharing.

Root cause: Each address space must be contiguous in memory.

Dynamic address translation



Break the requirement that the process space be contiguous.

MMU strategies we'll discuss:

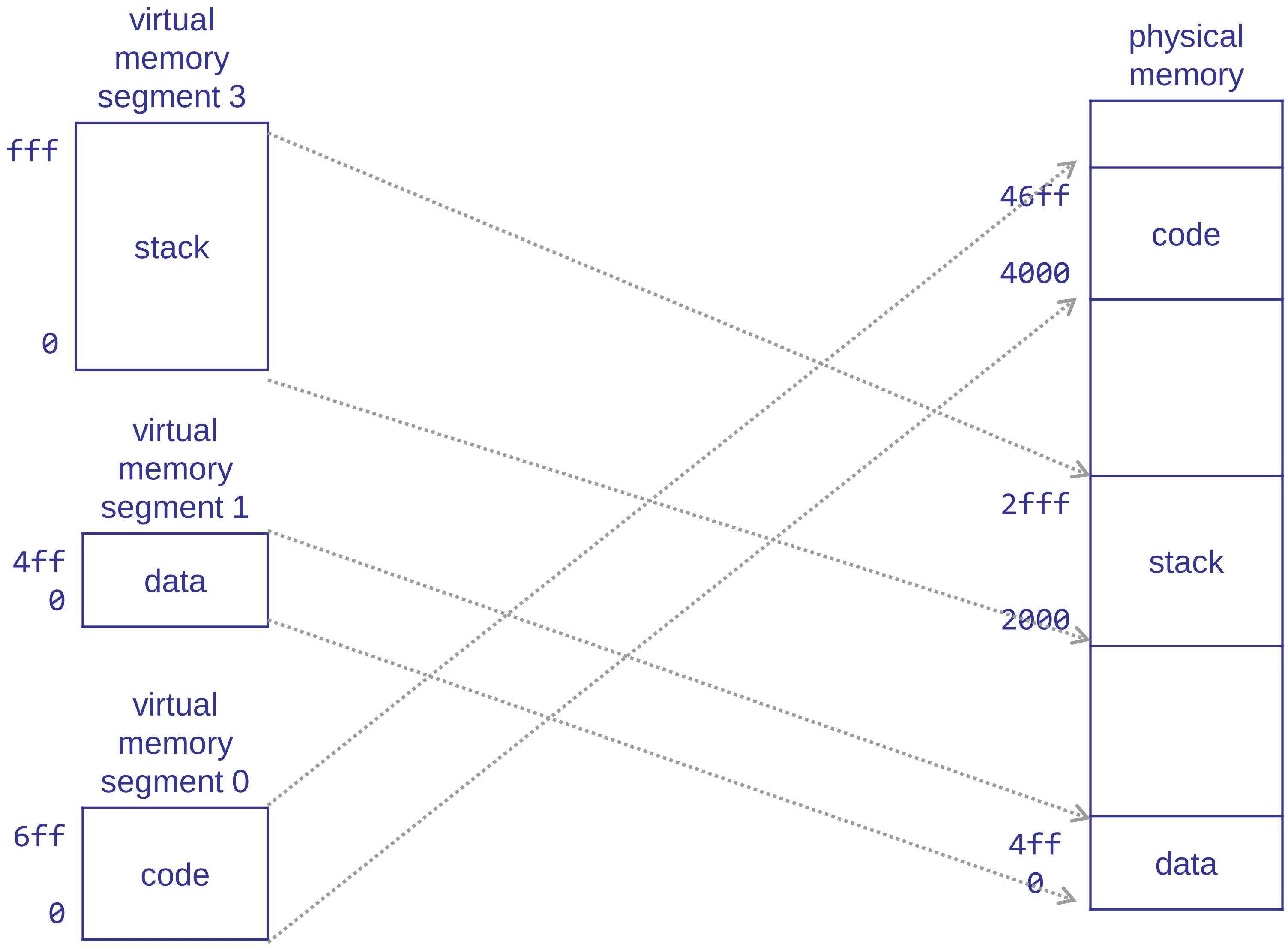
1. Base and bounds.
2. Segmentation.
3. Paging.

Segmentation

Divide address space into segments, regions of memory that are:

1. Contiguous in physical memory.
2. Contiguous in virtual address space.
3. Variable size.

Segmentation



Segmentation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Virtual address is of the form: (segment #, offset)

Physical address = base for segment + offset

Ways to specify the segment number:

1. High bits of address
2. Special register
3. Implicit to instruction opcode

Segmentation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Virtual address is of the form: (segment #, offset)

Physical address = base for segment + offset

Ways to specify the segment number:

1. High bits of address
2. Special register
3. Implicit to instruction opcode

Segmentation: Translation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Physical address for virtual address (`3, 100`)?

`2100`

Physical address for virtual address (`0, ff`)?

`40ff`

Physical address for virtual address (`2, ff`)?

Physical address for virtual address (`1, 2000`)?

Valid vs. invalid addresses

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Not all virtual addresses are valid.

- Valid → address is part of virtual address space.
- Invalid → virtual address is illegal to access.
  - Accessing invalid address causes trap to OS.

Reasons for virtual address being invalid?

- Invalid segment number.
- Offset within valid segment beyond bound.

Protection

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Different segments can have different protection.

Code is usually read only (allows fetch, load,...).

Stack and data are usually read/write (allows load, store,...).

Was this fine-grained protection possible in base and bounds?

What must be changed on a context switch?

Segmentation

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

Parts of the address space can grow separately.

How would you grow a segment?

If there's contiguous free space, can simply extend the bound.

Otherwise, must move it, perhaps compacting memory.

Benefits of Segmentation

Easy to share part of address space.

Process 1

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
3	2000	1000	stack segment

Process 2

Segment #	Base	Bounds	Description
0	4000	700	code segment
1	1000	300	data segment
3	500	1000	stack segment

Segmentation

Pros:

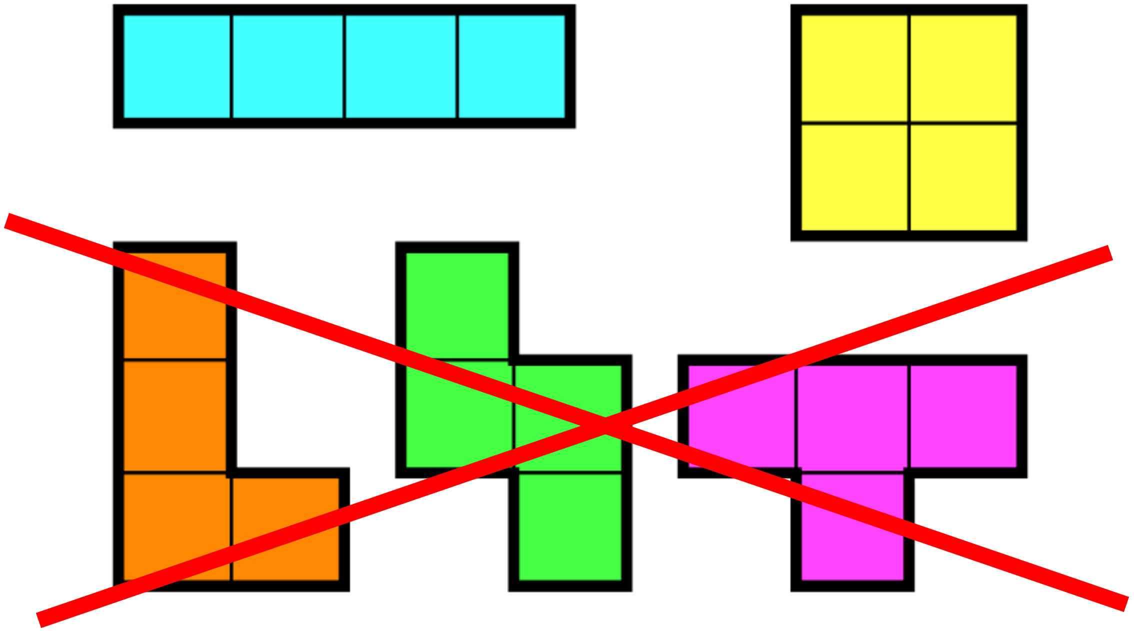
1. Can grow each segment independently.
2. Can share segments across address spaces.

Cons:

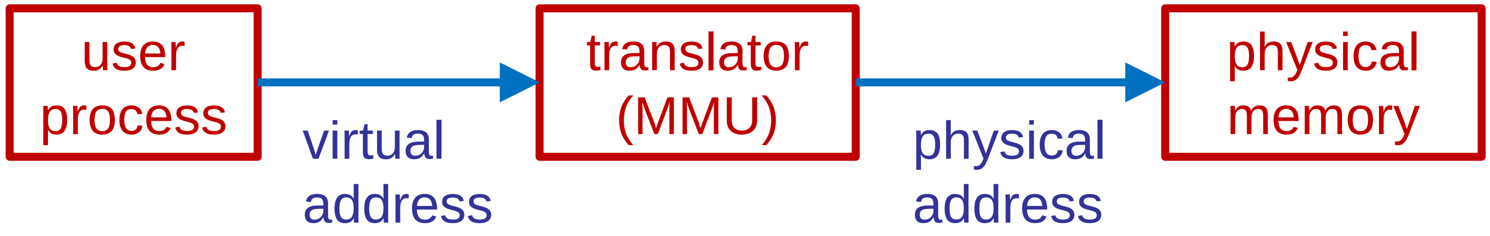
1. Every segment must be smaller than physical memory.
2. Segment allocation is hard.
3. External fragmentation.

Cause: Variable amount of contiguous memory.


GAME
OVER



Dynamic address translation



Break the requirement that the process space be contiguous.

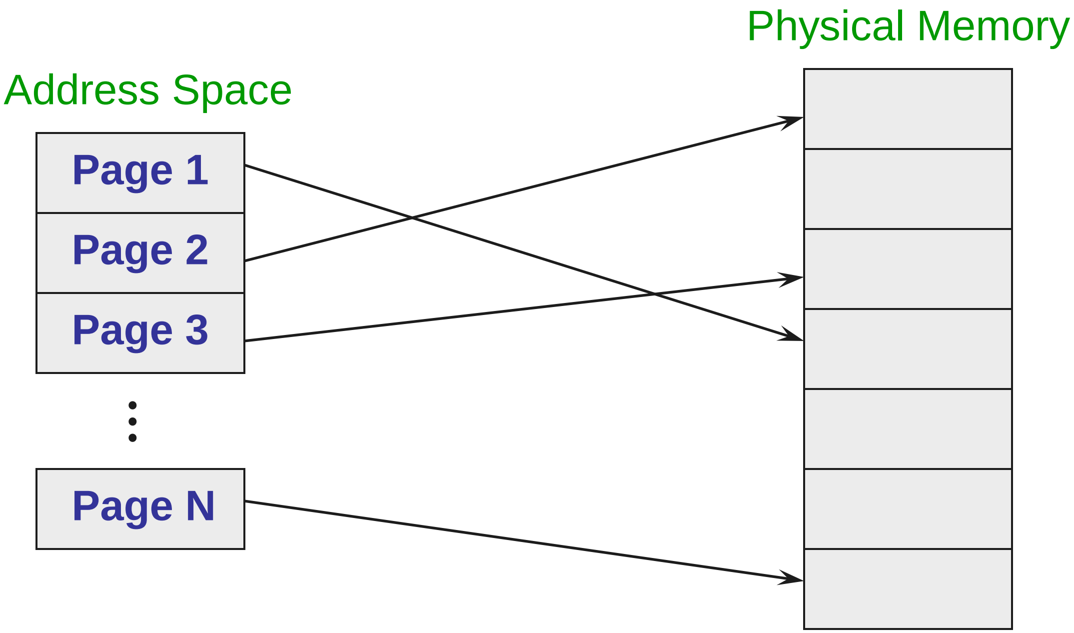
MMU strategies we'll discuss:

1. Base and bounds.
2. Segmentation.
3. Paging.

Paging

Allocate phys. memory in fixed-size units (pages)

- Any free physical page can store any virtual page



Paging

Translation data is the page table.

Virtual address is split into:

1. Virtual page # (high bits of address, e.g., bits 31-12).

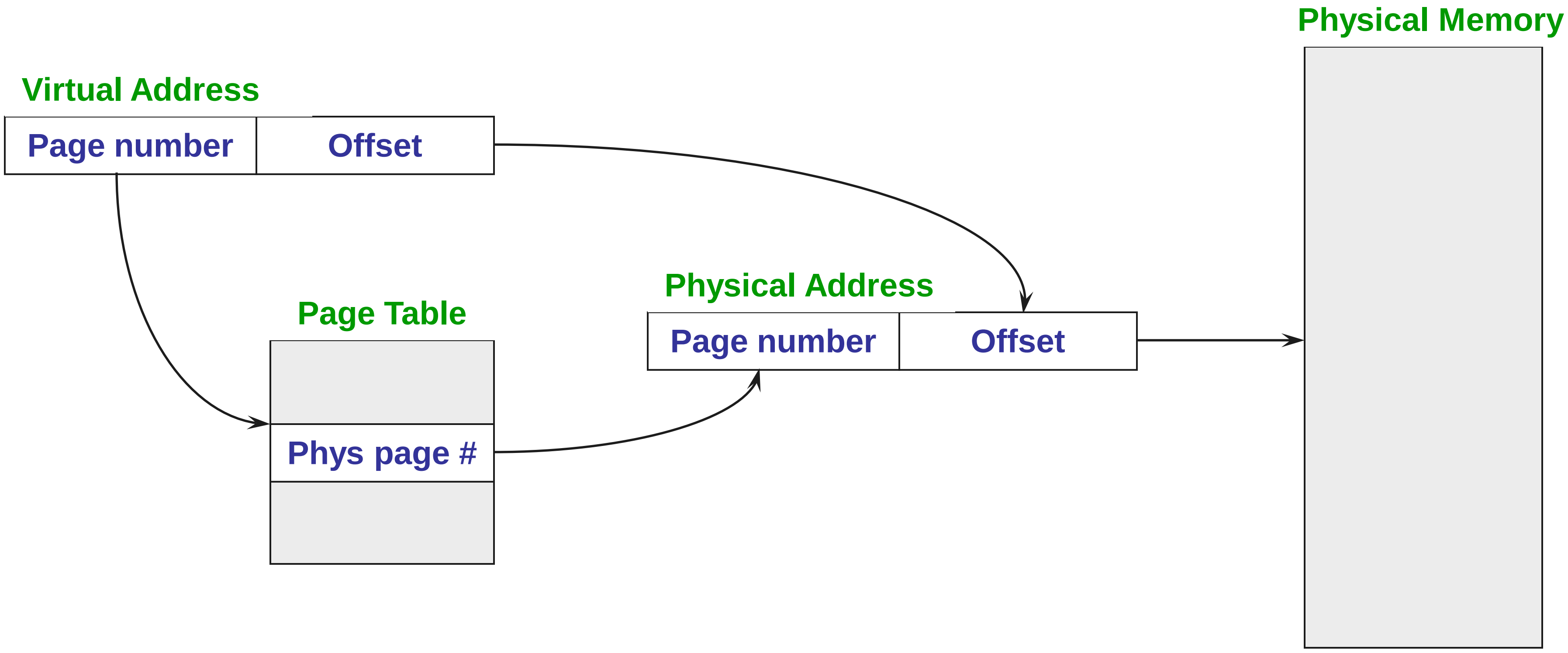
2. Offset (low bits of address, e.g., bits 11-0, for 4 KB page size).

Why no column for bound?

Function to translate VA to PA?

Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048575	invalid

Page Lookups



Paging

Translating virtual address to physical address.

What must be changed on a context switch?

Indirection via Page Table Base Register.

```
MMU_translation( )
   {
   if ( virtual page is invalid )
      trap to OS fault handler;
   else
      {
      physical page # =
         pageTable[ virtual page # ].physPageNum;
      physical address =
         concat( Physical page #, offset );
      }
   }
```

Paging

Each virtual page can be in physical memory or “paged out” to disk.

How does processor know that a virtual page is not in physical memory?

```
MMU_translation( )
   {
   if ( virtual page is invalid )
      trap to OS fault handler;
   else
      {
      physical page # =
         pageTable[ virtual page # ].physPageNum;
      physical address =
         concat( Physical page #, offset );
      }
   }
```

Paging

Each virtual page can be in physical memory or “paged out” to disk.

How does processor know that a virtual page is not in physical memory?

Like segments, pages can have different protections (e.g., read, write, execute).

```
MMU_translation( )
   {
   if ( virtual page is
         invalid or non-resident or protected )
      trap to OS fault handler;
   else
      {
      physical page # =
         pageTable[ virtual page # ].physPageNum;
      physical address =
         concat( Physical page #, offset );
      }
   }
```

Paging

Revised page table:

Virtual page #	**Physical page #**	**Resident**	**Protection**
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Valid versus Resident

Valid → virtual page is legal for process to access.

Resident → virtual page is valid and in physical memory.

Error to access invalid page, but not to access non-resident page.

Who makes a virtual page resident/non-resident?

Who makes a virtual page valid/invalid?

Why would a process want one of its virtual pages to be invalid?

Picking Page Size

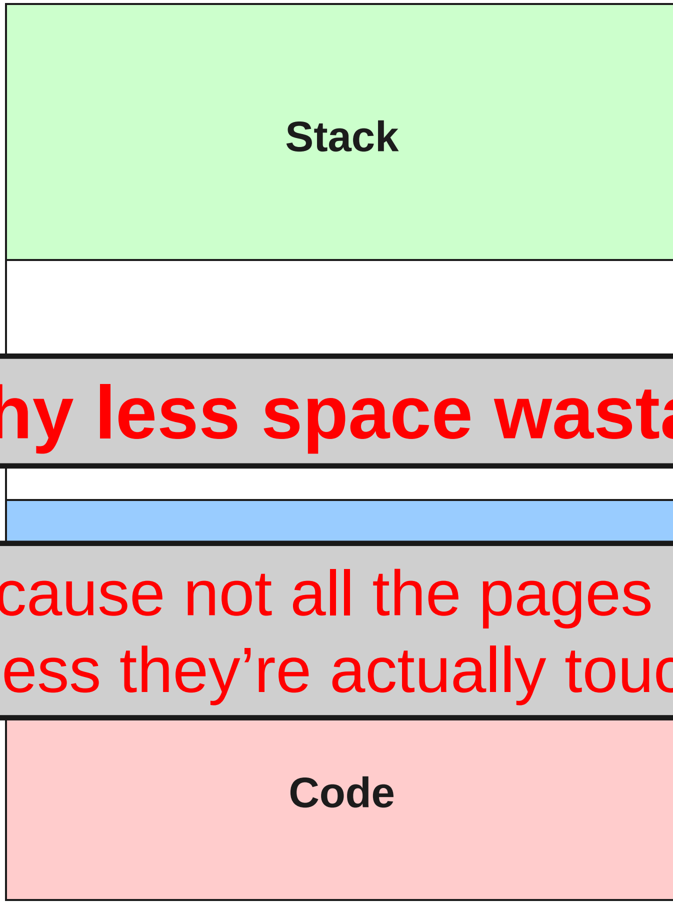
What happens if page size is really small?

What happens if page size is really big?

Typically a compromise, e.g., 4 KB or 8 KB.

Some architectures support multiple page sizes.

Growing Address Space



Virtual page #	Physical page #
0	105
1	15
2	283
...	invalid
1048574	48136
1048575	60

Why less space wastage than base and bounds?

Because not all the pages have to be in physical memory unless they're actually touched.

Paging

Pros

1. Simple memory allocation
2. Flexible sharing
3. Easy to grow address space

Cons

1. 32-bit virtual address, 4 KB pages, 4 byte PTEs
2. Page table size?

Page table size

32-bit address → 2^32 unique addresses

4 KB page → (2^32)/4 KB = 2^20 virtual pages

4 bytes per page table entry → 4 MB page table

25 processes → 100 MB for page tables!

How to reduce page table overhead?

Multi-level Paging

Standard page table is a simple array

Multi-level paging generalizes this into a tree

Example: Two-level page table with 4KB pages

- Index into level 1 page table: virtual address bits 31-22
- Index into level 2 page table: virtual address bits 21-12
- Page offset: bits 11-0

Multi-level Paging

level 1
page table

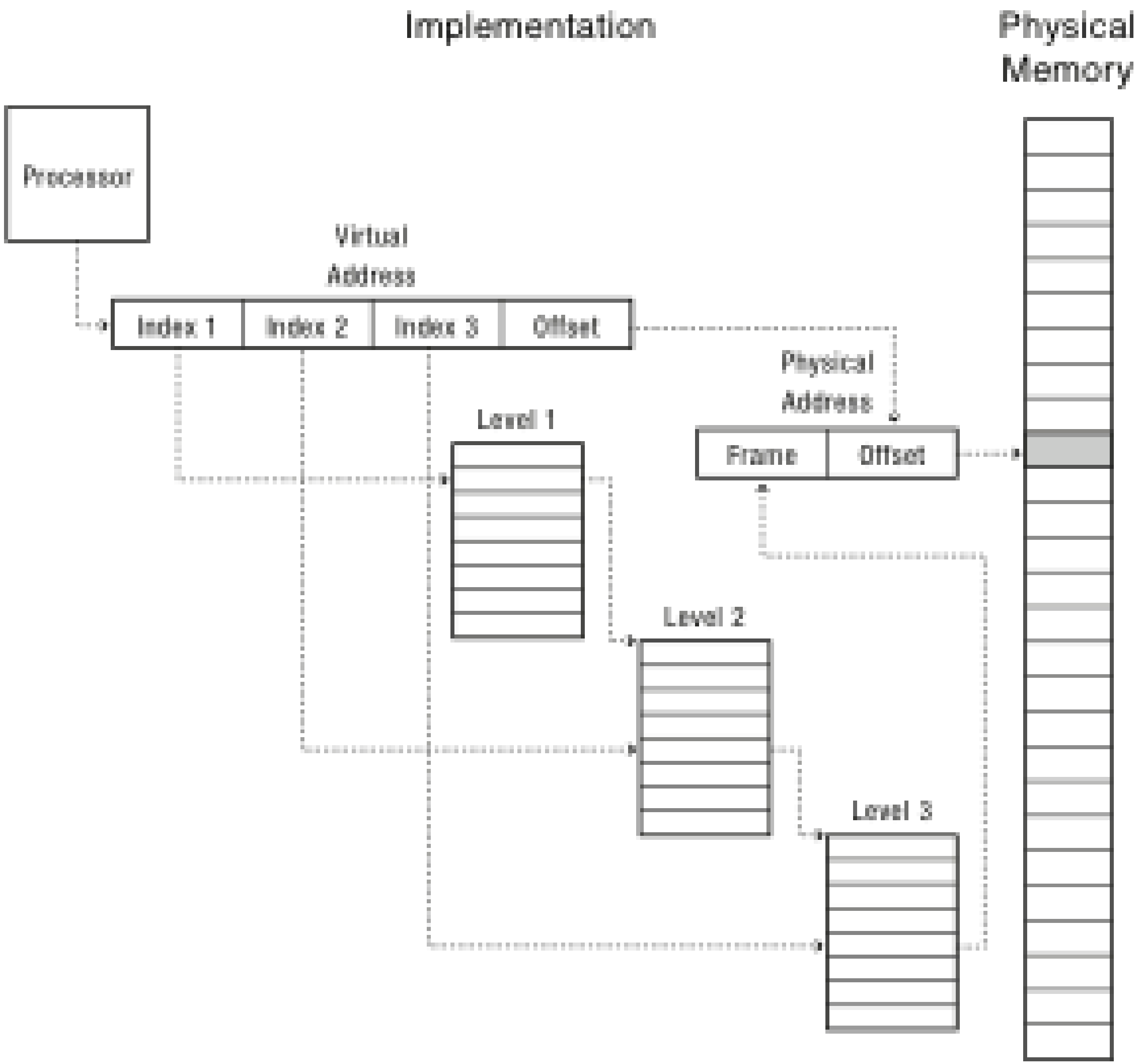
0	1	…	n

level 2
page tables

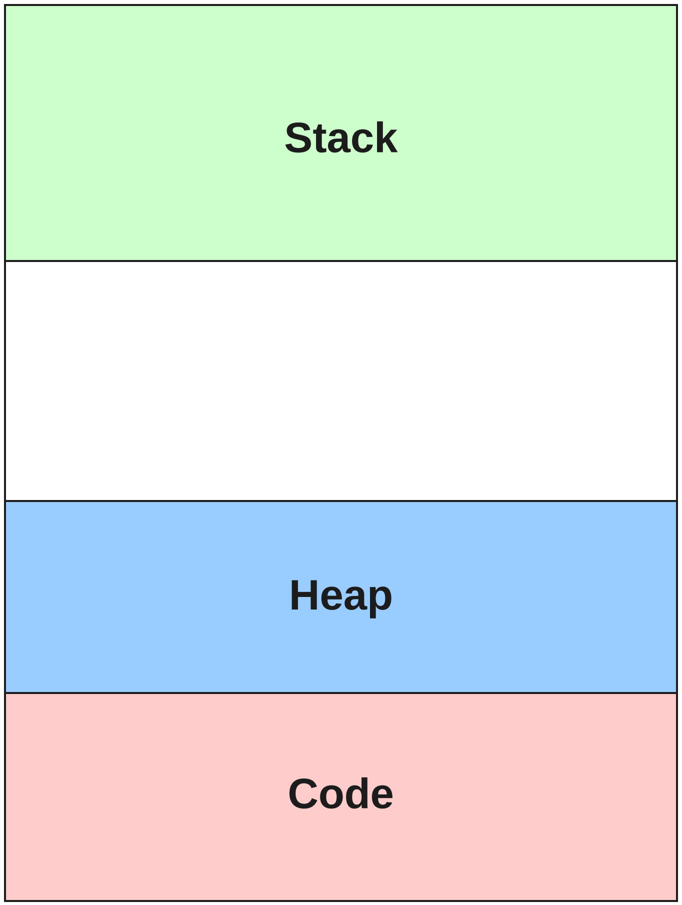
virtual address bits 21-12	physical page #
0	10
1	15
2	20
3	2

virtual address bits 21-12	physical page #
0	30
1	4
2	8
3	3

How does this let translation data take less space?


Implementation
Physical
Memory
Processor
Virtual
Address
Index 1
Index 2
Index 3
Offset
Physical
Address
Frame
Offset
Level 1
Level 2
Level 3

Sparse Address Space



Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Sparse Address Space

Bits 31-22	Physical address
0	0xfffff389
1	Invalid
2	Invalid
…	Invalid
1021	Invalid
1022	Invalid
1023	0xffff7046

Bits 21-12	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid

Bits 21-12	Physical page #
...	invalid
1020	invalid
1021	1078
1022	48136
1023	60

Multi-level paging

How to share memory between address spaces?

What must be changed on a context switch?

Pros

- Easy memory allocation
- Flexible sharing
- Space efficient for sparse address spaces

Cons

- Two or more extra lookups per memory reference

Translation lookaside buffer

- TLB caches virtual page # to PTE mapping
  - Cache hit → Skip all the translation steps
  - Cache miss → Get PTE, store in TLB, restart instruction

Does TLB change what happens on a context switch?

End-to-end look at paging

- New process → allocate new L1 page table
  - All entries in L1 page table invalid
- As process makes virtual pages valid, allocate new L2 page tables and add entries
- To serve load/store on a virtual page:
  - CPU looks up TLB to find PTE for virtual page #
  - If absent, lookup PTE in memory and load TLB
- When process ends, deallocate L1 and L2 page tables

Page replacement

Not at all valid pages can be in phys memory.

How to handle loads/stores on non-resident pages?