Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 10: Virtual memory

Nicole Hamilton

https://web.eecs.umich.edu/~nham/

nham@umich.edu

Agenda

1. Midterm.
2. Strategie for P2.
3. Review of deadlocks.
4. Virtual memory.

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 next Friday and in review session weekend before exam.

Try sample exams in exam setting before then.

Writing good P2 test cases

Need targeted test cases and stress tests.

Targeted test case:

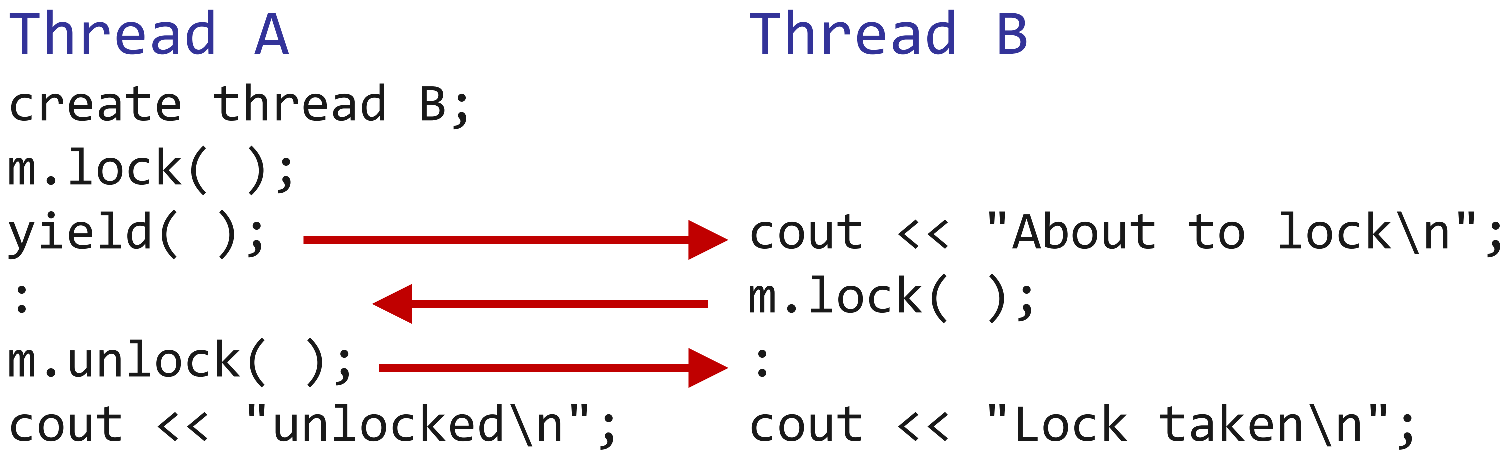
- Tests one or a few specific things.
- Failure tells you exactly what’s wrong.
- Autograder gives you results for your tests (!)

Stress tests:

- Lots of concurrency, activities.
- Tests for rare, non-deterministic interleavings.
- Failure doesn’t say why – run under debugger?

Targeted test case

Want to test that `lock( )` blocks when mutex held, finishes when the lock is released. Does this work?

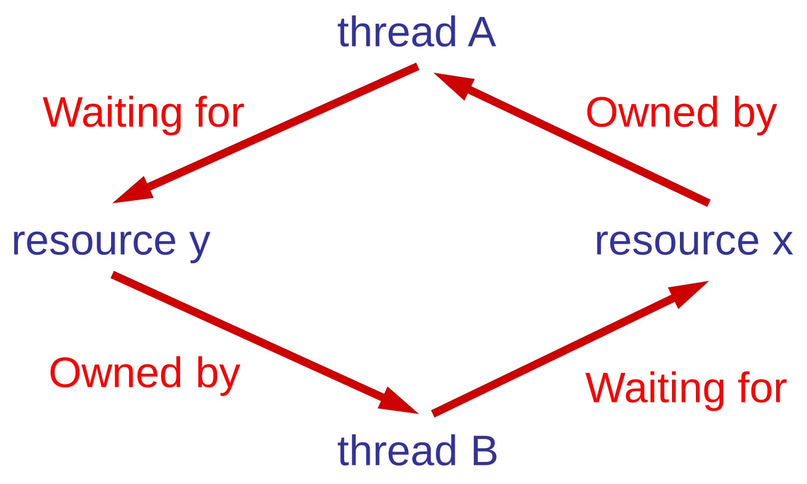


```
Thread A                    Thread B
create thread B;
m.lock( );
yield( );                   cout << "About to lock\n";
:                           m.lock( );
m.unlock( );                :
cout << "unlocked\n";       cout << "Lock taken\n";
```

Not a valid test. No way to know B ever runs before A finishes. Would at minimum have to add a `yield( )`.

Waits-for graph

Cycle represents a deadlock.



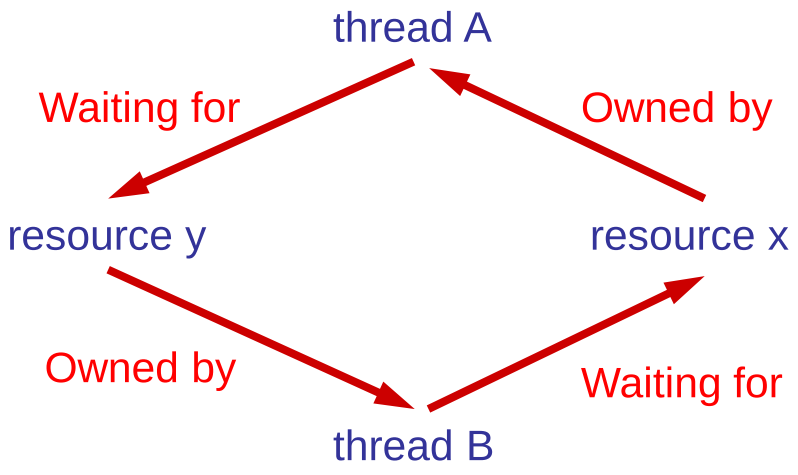
Waits-for graph

Thread A

```
1  x.lock( );
3  y.lock( );
   ...
   y.unlock( );
   x.unlock( );
```

Thread B

```
2  y.lock( );
4  x.lock( );
   ...
   x.unlock( );
   y.unlock( );
```



Coping with deadlocks

Alternatives:

1. Ignore.

   Typical OS strategy for application deadlocks.

   Do deadlocked apps consume CPU?

2. Detect and fix.

   Use waits-for graph to detect.

   How to fix?

   - Could kill threads but not always safe to do so.
   - Invariants can be broken while a thread holds a lock.

   Databases can rollback to a previous representation invariant state and restart, but general purpose rollback is costly, difficult.

3. Prevent them from occurring.

Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests.

Four necessary conditions for deadlock

1. Limited resources: Not enough to serve all threads simultaneously.
2. No preemption: Can’t force threads to give up resources.
3. Hold and wait: Threads hold resources while waiting to acquire others.
4. Cyclical chain of requests. This can be avoided.

Eliminating circular chain

Define a global order over all resources, e.g., by numbering them.

The numbering can be arbitrary but is usually least precious first to most precious last (so you hold the most precious resources the shortest amount of time.)

All threads acquire resources in this order.

The thread with the highest numbered resource can always run.

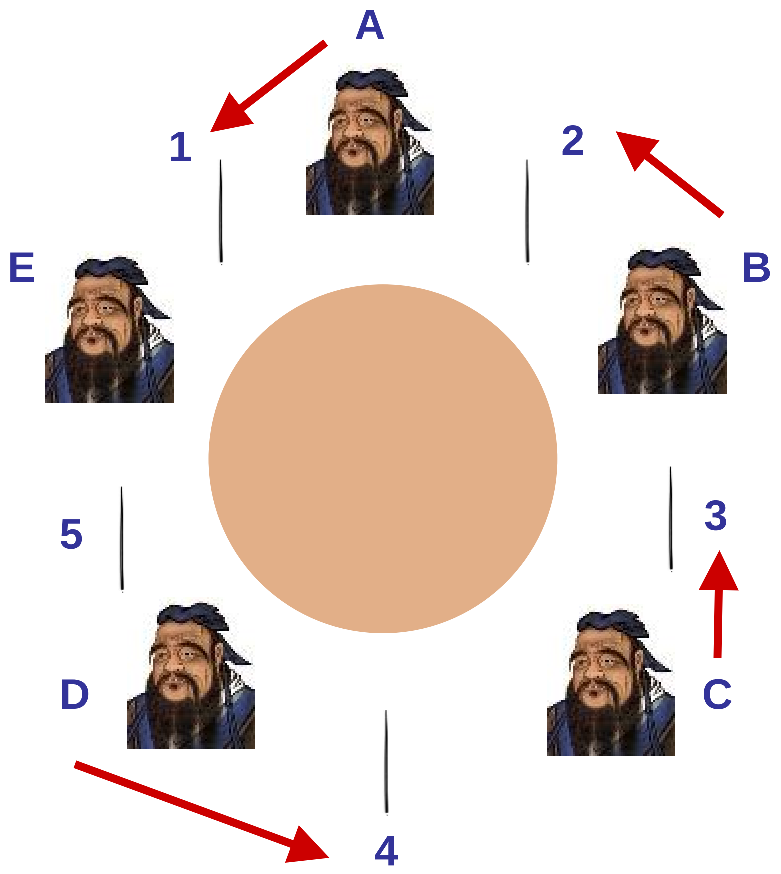
Dining philosophers

Solution:

Pick up the lower number chopstick first, then pick up the higher number chopstick.

Might get to a situation where E blocks but D proceeds with the highest numbered chopstick, then C, etc., until A finishes, then E proceeds.

No deadlock.



Eliminating the circular chain

But what if you already hold the higher number resource but also want the lower number resource?

You must first give up the higher number, then take them both in order.

You must rewrite your code to ensure you follow the pattern. It's a bug and you have to fix it.

Thread A

```
x.lock( );
y.lock( );
...
y.unlock( );
x.unlock( );
```

Thread B

```
y.lock( );
x.lock( );
...
x.unlock( );
y.unlock( );
```

⇨

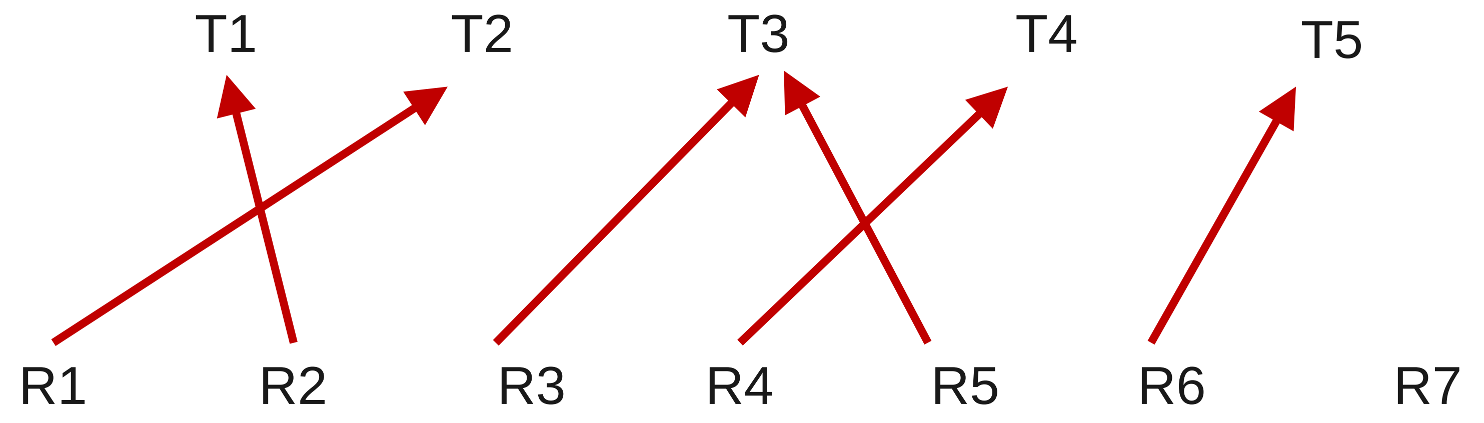
Thread B

```
y.lock( );
y.unlock( );
x.lock( );
y.lock( );
...
y.unlock( );
x.unlock( );
```

Global ordering of resources

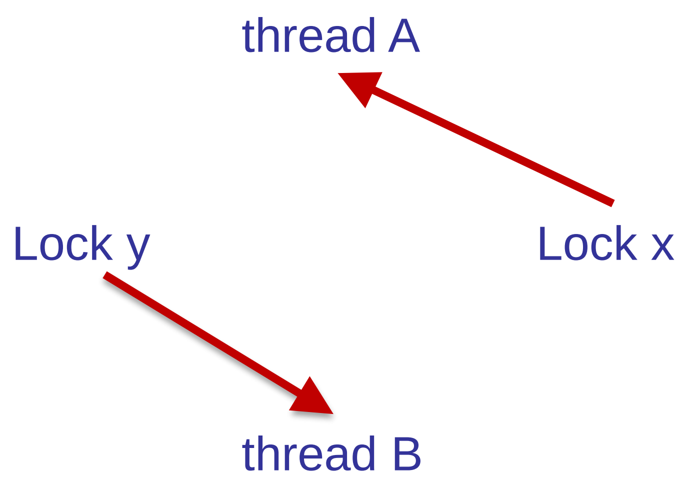
If every thread acquires resources in order

How can we be sure that *some* thread can progress?



Preventing deadlock

What if we don't grant resources that will lead to cycle in waits-for-graph?



Thread A	Thread B
`x.lock( );`	`y.lock( );`
`y.lock( );`	`x.lock( );`
`...`	`...`
`y.unlock( );`	`x.unlock( );`
`x.unlock( );`	`y.unlock( );`

Threads and Concurrency

Physical reality:

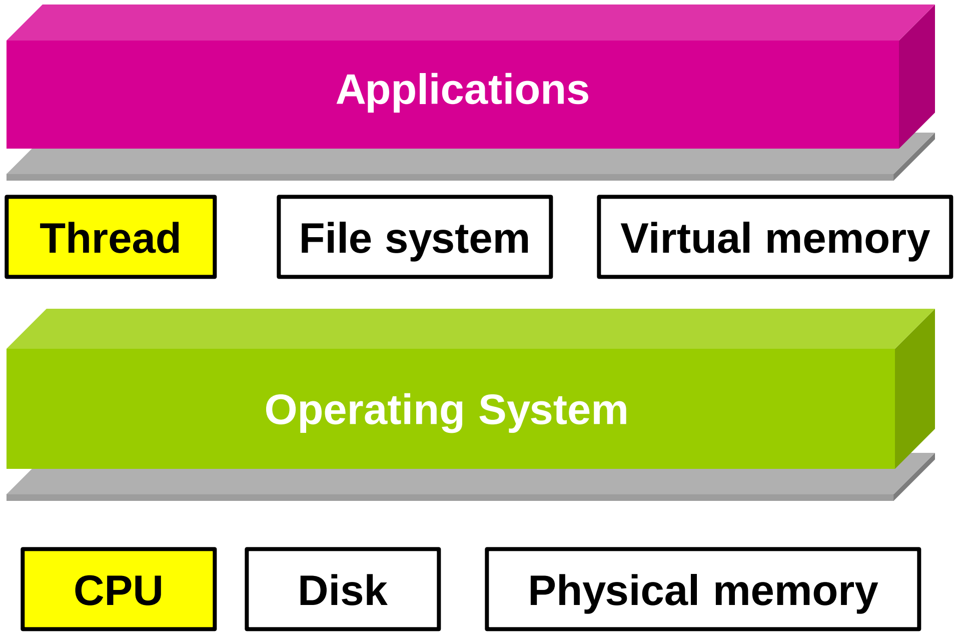
- Limited # of CPUs operating independently
- Low-level H/W support: interrupts and test_and_set

Abstraction:

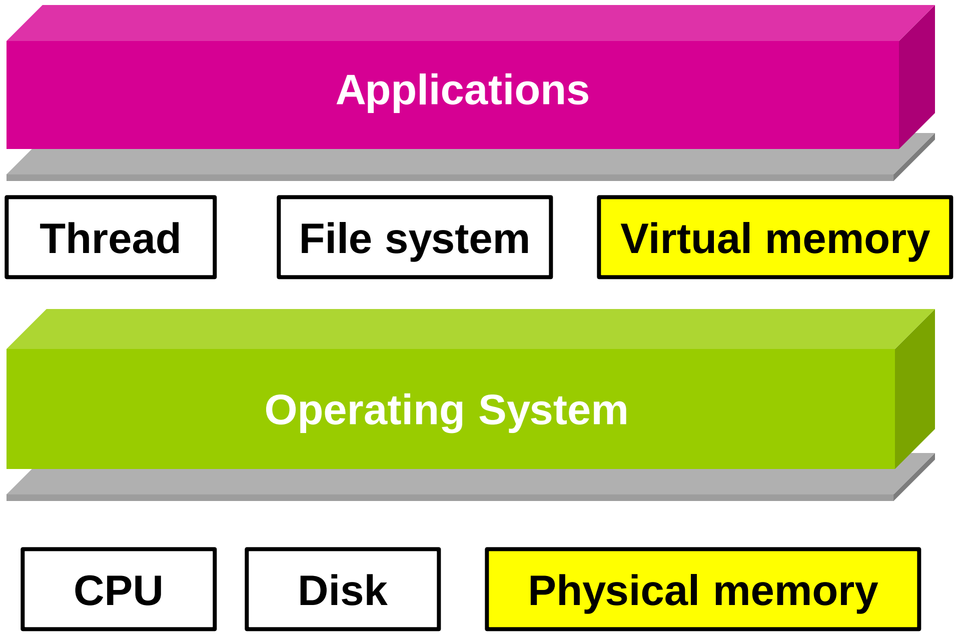
- Threads can assume infinite CPUs
- Locks for mutual exclusion, condition variables for ordering constraints, and semaphores for both

Over-constrained synchronization → deadlock

OS abstractions



OS Abstractions



Memory management

Recall: Process = Set of threads + address space

Address space

- All the memory space the process can use as it runs

Hardware interface:

- Physical memory shared between processes

Potential abstractions:

- Allow direct access to addresses in physical memory?
- Partition physical memory across processes?

Abstraction provided by OS

Virtual memory. Address space can be larger than physical memory.

Address independence. Same numeric address can be used in different address spaces, yet remain logically distinct.

Protection. One process can’t access data in another process’s address space (actually controlled sharing).

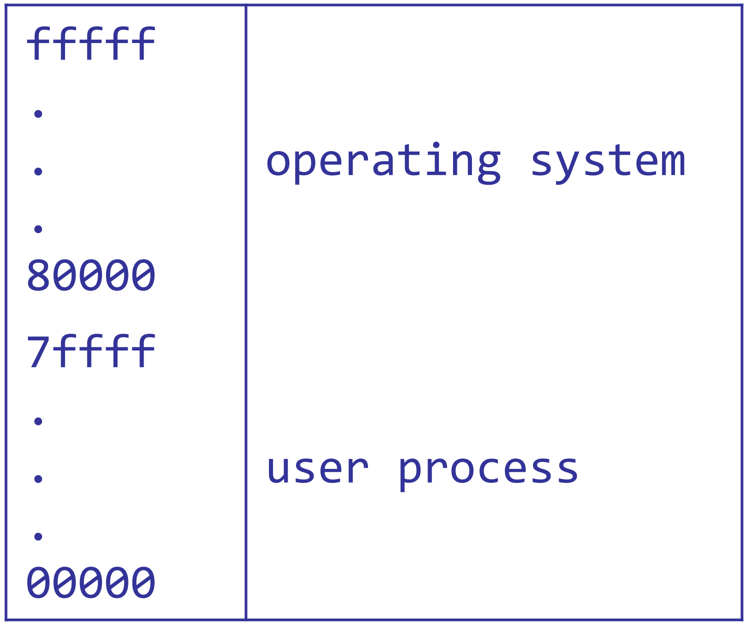
Uni-programming

Consider a machine that runs one process at a time.

Partition the memory, reserving space for the OS.

Always load the process into same spot in memory.

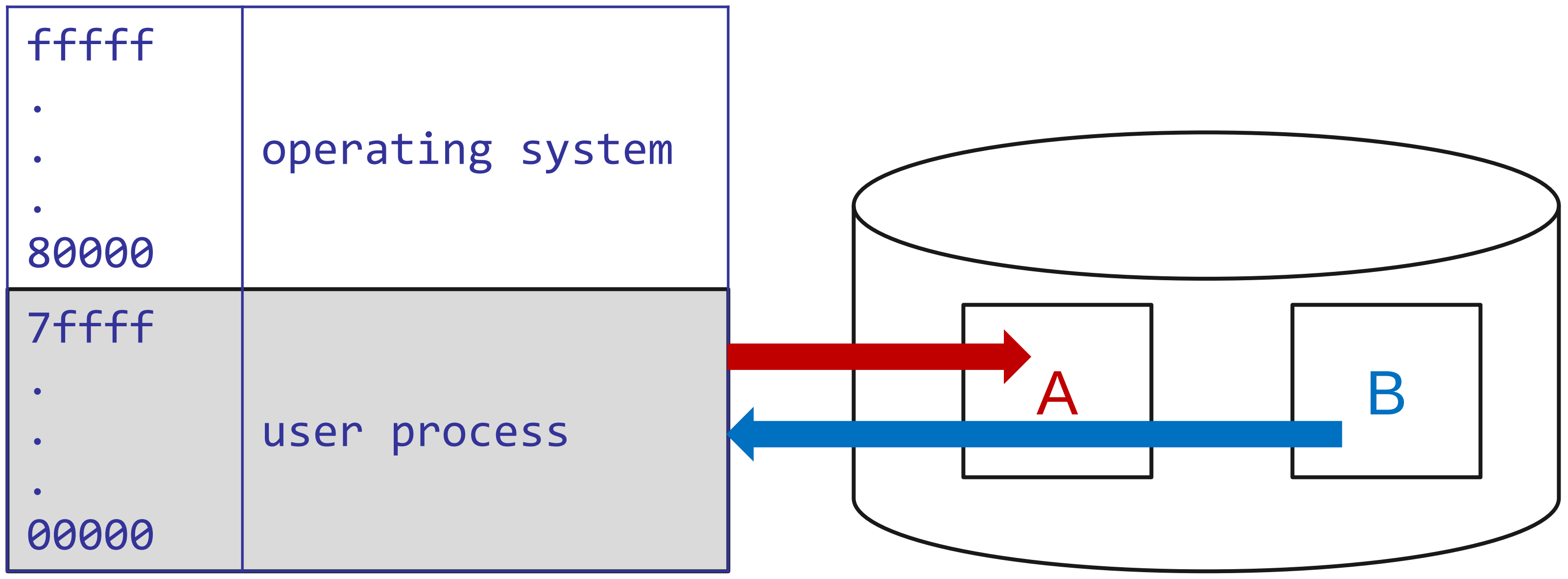
Virtual address = physical address.



Problems?

Could run multiple programs but only by swapping them back-and-forth to disk.

Context switch



Physical memory

Disk

Context switch would be very slow so you couldn't afford to do it often or you'd enter a condition called *thrashing*, where the only work the system does is swapping processes in and out of memory.

Uni-programming summary

Address space abstractions:

- Virtual memory: No
- Address Independence: Yes
- Protection: Yes

Pro: Simple (early OSes for PCs used this)

Con: Expensive context switch

Multi-programming

Allow >1 address space in physical memory.

Programs written assuming address range starts at 0.

Only 1 process can start at physical address 0.

Implies address translation necessary for address independence.

Static address translation

Compiler generates addresses starting at 0.

Linker-loader adds an offset to instructions as they're loaded into memory.

For example,

```
MOV 0x10, %eax
```

becomes

```
MOV 0x20010, %eax
```

fffff . 40000	operating system
3ffff . 20000	user process 1
1ffff . 00000	user process 2

Register offset addressing

Arrays, structs, pointers use indexing.

Any issues with this?

`%ebx` could be negative or greater than the partition size!

Difficult to trap invalid accesses.

Could add dynamic checks before loads/stores.

Very expensive for general-purpose languages like C and C++.

```
# Add 0x10 to %ebx and load
# value at that address.

    MOV %ebx(0x10), %eax
```

Dynamic address translation

Problem is application gets “last move.”

1. Compiler generates machine code (app).
2. Linker-loader translates addresses (OS).
3. Register values used to calculate addresses (app).

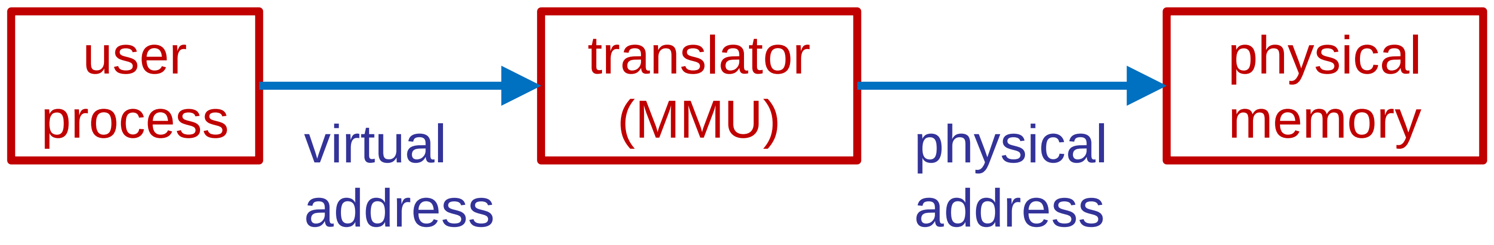
Dynamic translation: system has the last move.

Hardware (MMU) translates all memory references.

Virtual address: address used by the process.

Physical address: address in physical memory.

Dynamic address translation



Address independence

Virtual addresses are scoped to 1 process.

Protection

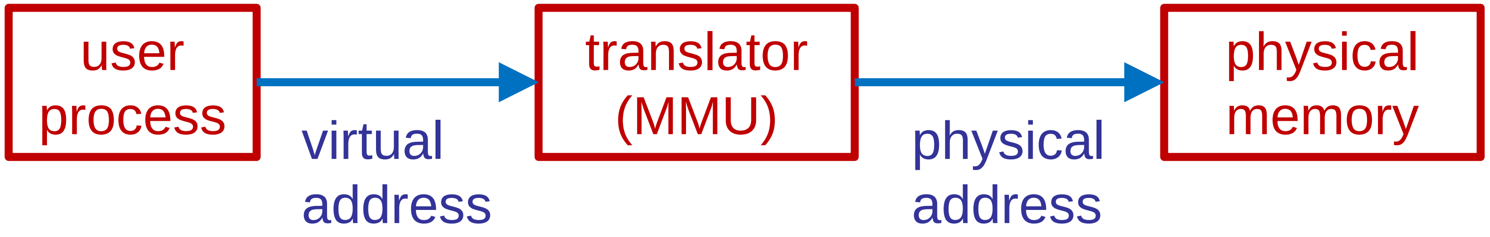
One process can’t refer to another’s address space.

Virtual memory

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

Allows changing translations on the fly.

Dynamic address translation

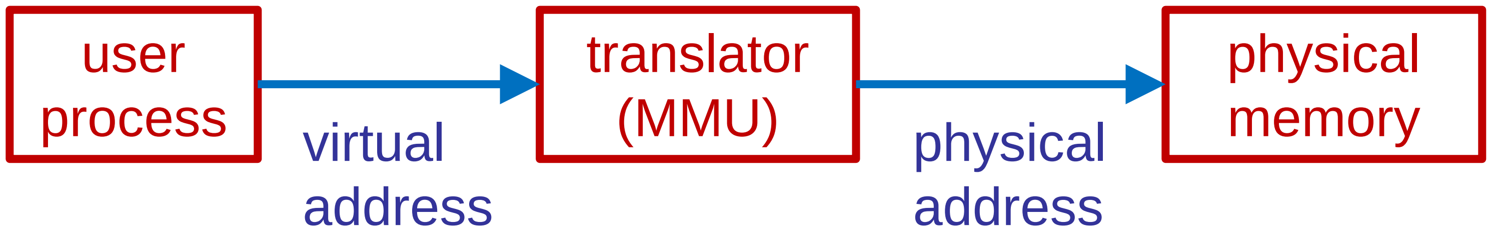


Many ways to implement the translator.

Tradeoffs

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

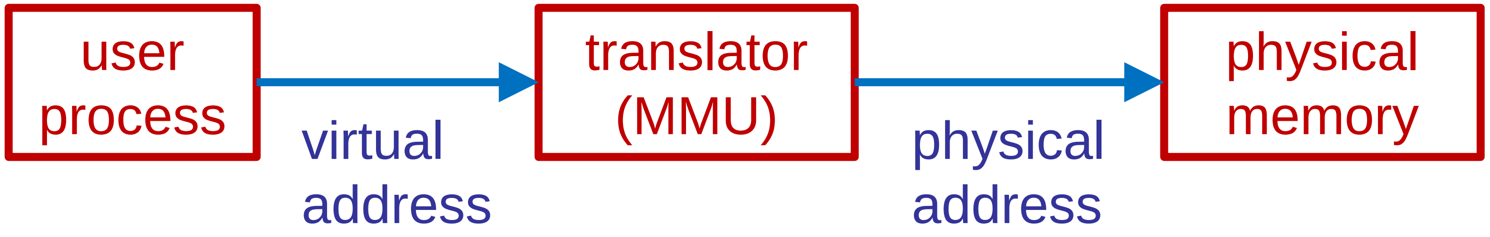
Dynamic address translation



MMU strategies we'll discuss:

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

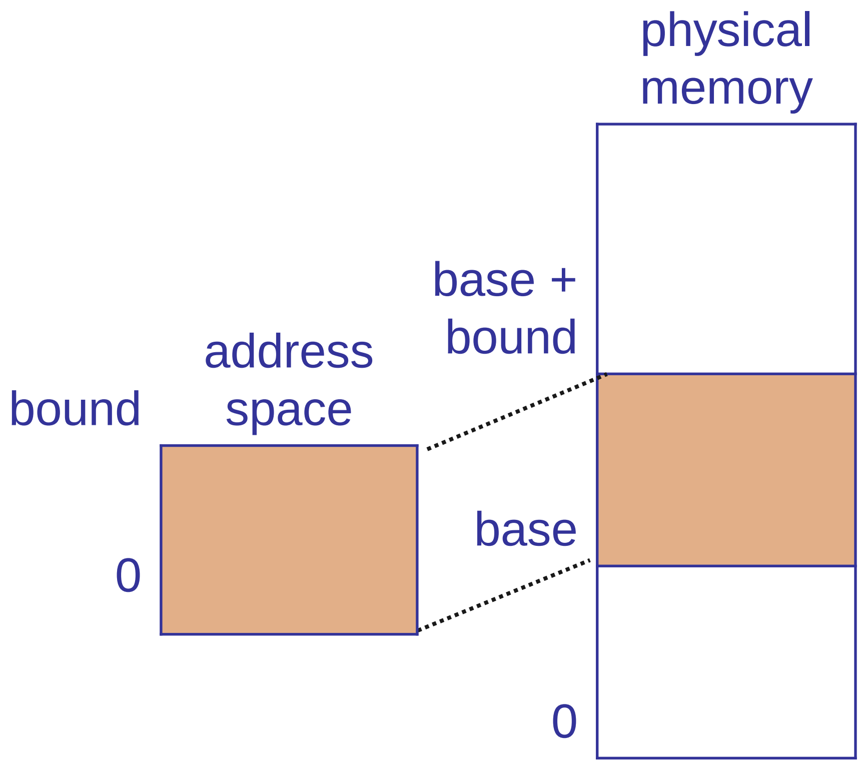
Dynamic address translation



MMU strategies we'll discuss:

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

Base and bounds



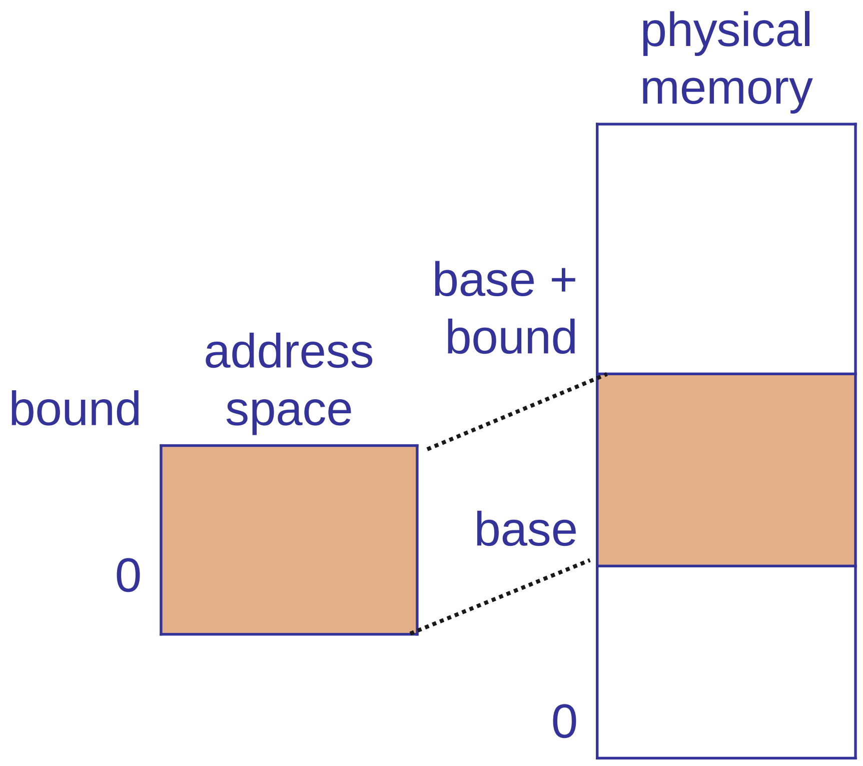
Load each process into a contiguous region of physical memory.

Prevent process from accessing data outside its region.

Base register: starting physical address.

Bound register: size of region.

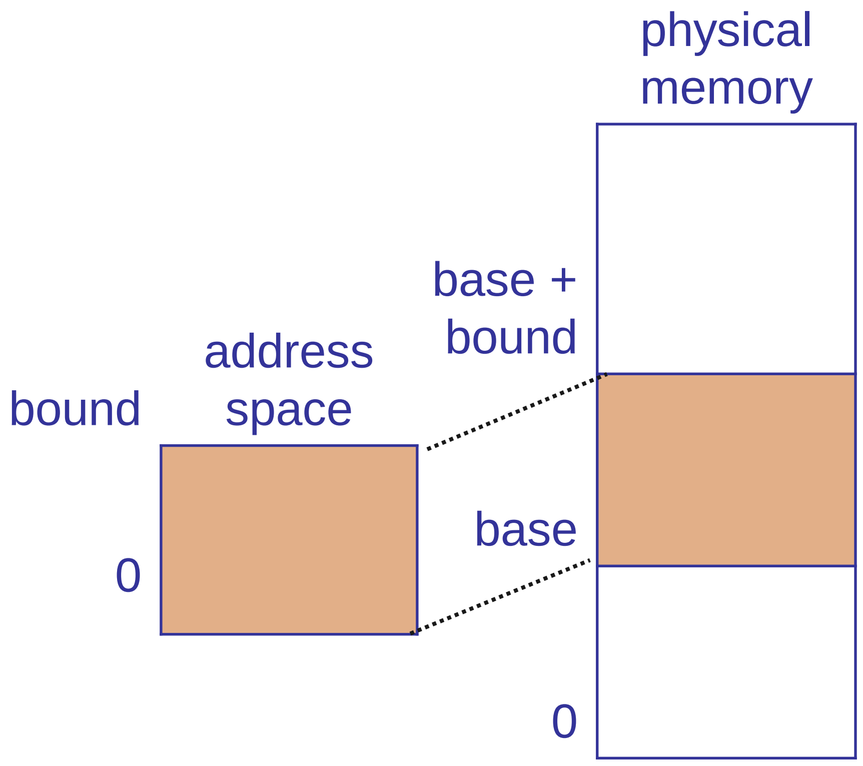
Base and bounds



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

What happens on a context switch?

Base and bounds



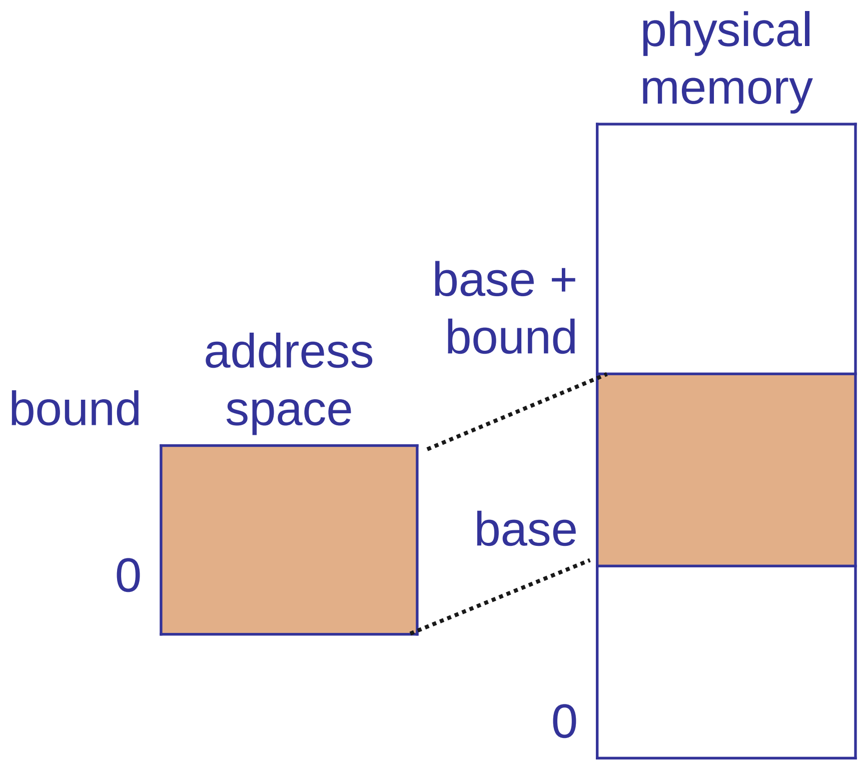
Pros:

1. Fast.
2. Simple hardware support.

Cons?

1. Can the address space be larger than physical memory?
2. Any undesirable effects as address spaces created and destroyed over time?
3. How would you grow the address space?
4. How would you share parts of the address space across processes? Why would you want to do this?

Base and bounds



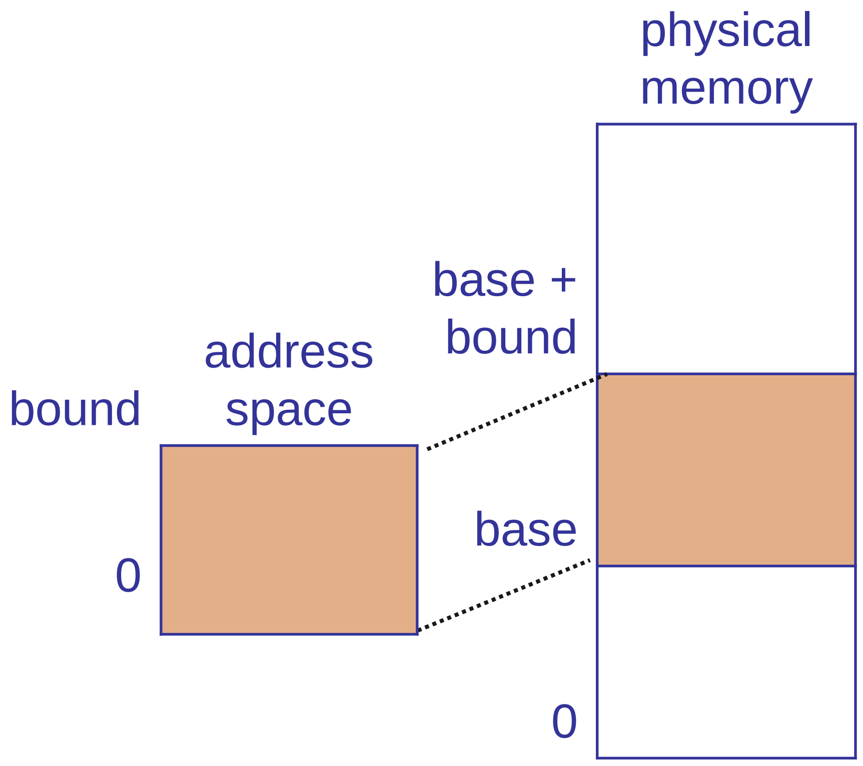
Pros:

1. Fast.
2. Simple hardware support.

Cons?

1. Can the address space be larger than physical memory?
2. Any undesirable effects as address spaces created and destroyed over time?
3. How would you grow the address space?
4. How would you share parts of the address space across processes? Why would you want to do this?

Must be physical memory



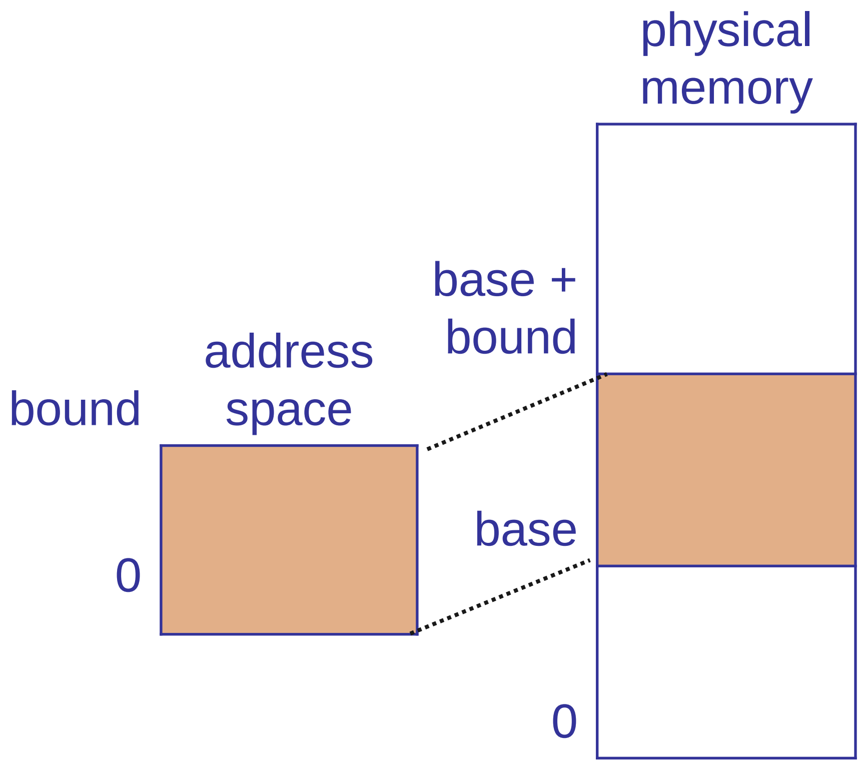
Load each process into a contiguous region of physical memory.

Prevent process from accessing data outside its region.

Base register: starting physical address.

Bound register: size of region.

Base and bounds



Pros:

1. Fast.
2. Simple hardware support.

Cons?

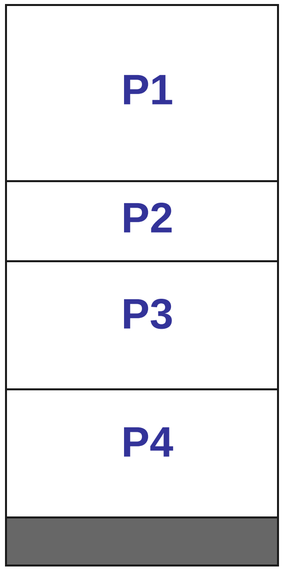
1. Can the address space be larger than physical memory?
2. Any undesirable effects as address spaces created and destroyed over time?
3. How would you grow the address space?
4. How would you share parts of the address space across processes? Why would you want to do this?

External fragmentation

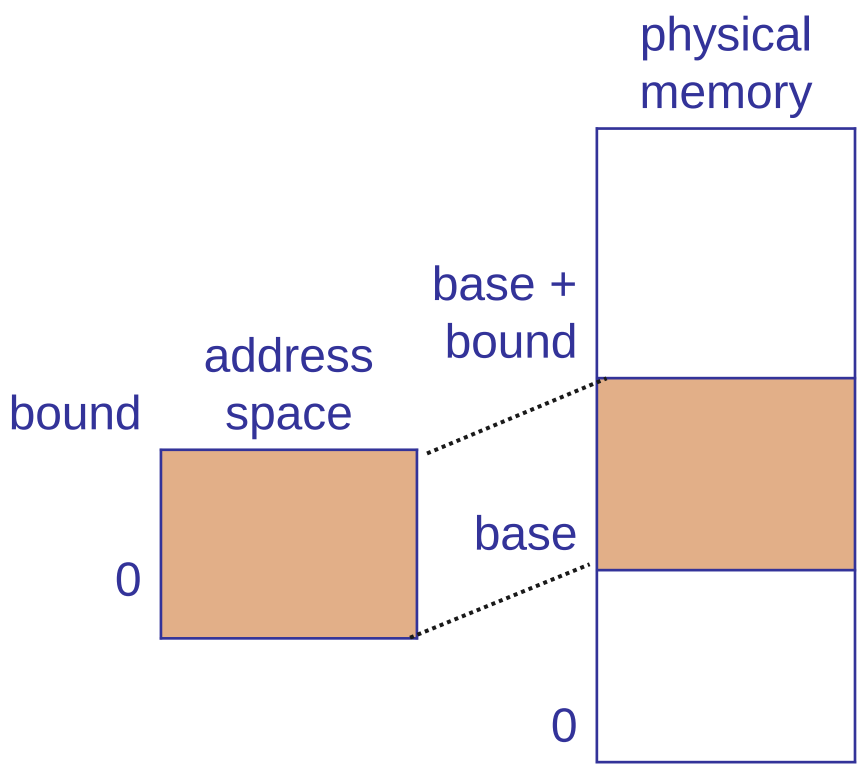
Processes come and go, leaving a mishmash of available memory regions.

Wasted memory between allocated regions.

Must move things around, compacting memory periodically to solve.



Base and bounds



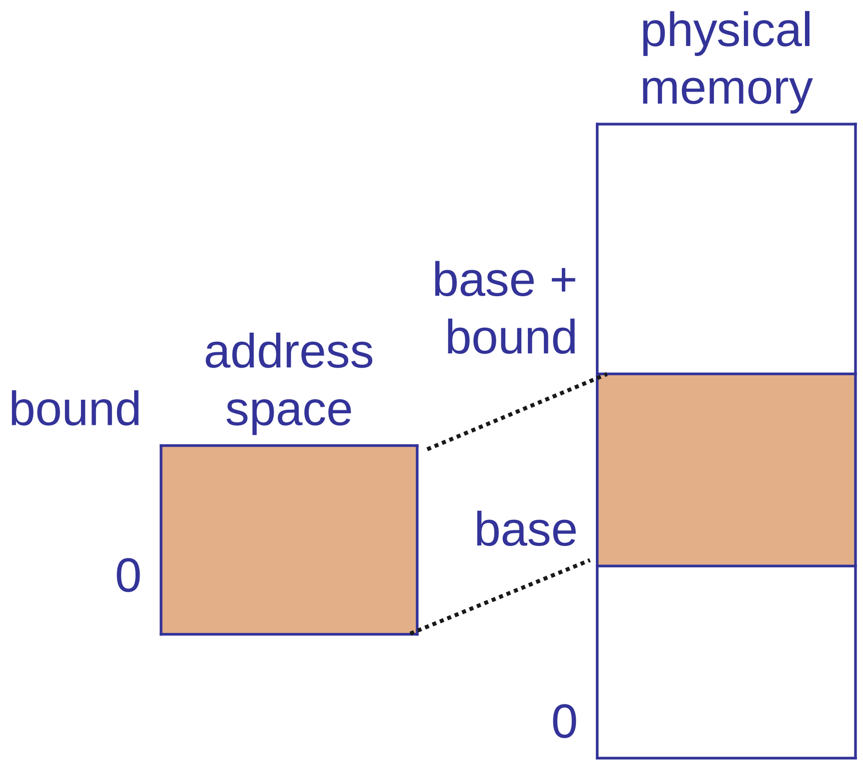
Pros:

1. Fast.
2. Simple hardware support.

Cons?

1. Can the address space be larger than physical memory?
2. Any undesirable effects as address spaces created and destroyed over time?
3. How would you grow the address space?
4. How would you share parts of the address space across processes? Why would you want to do this?

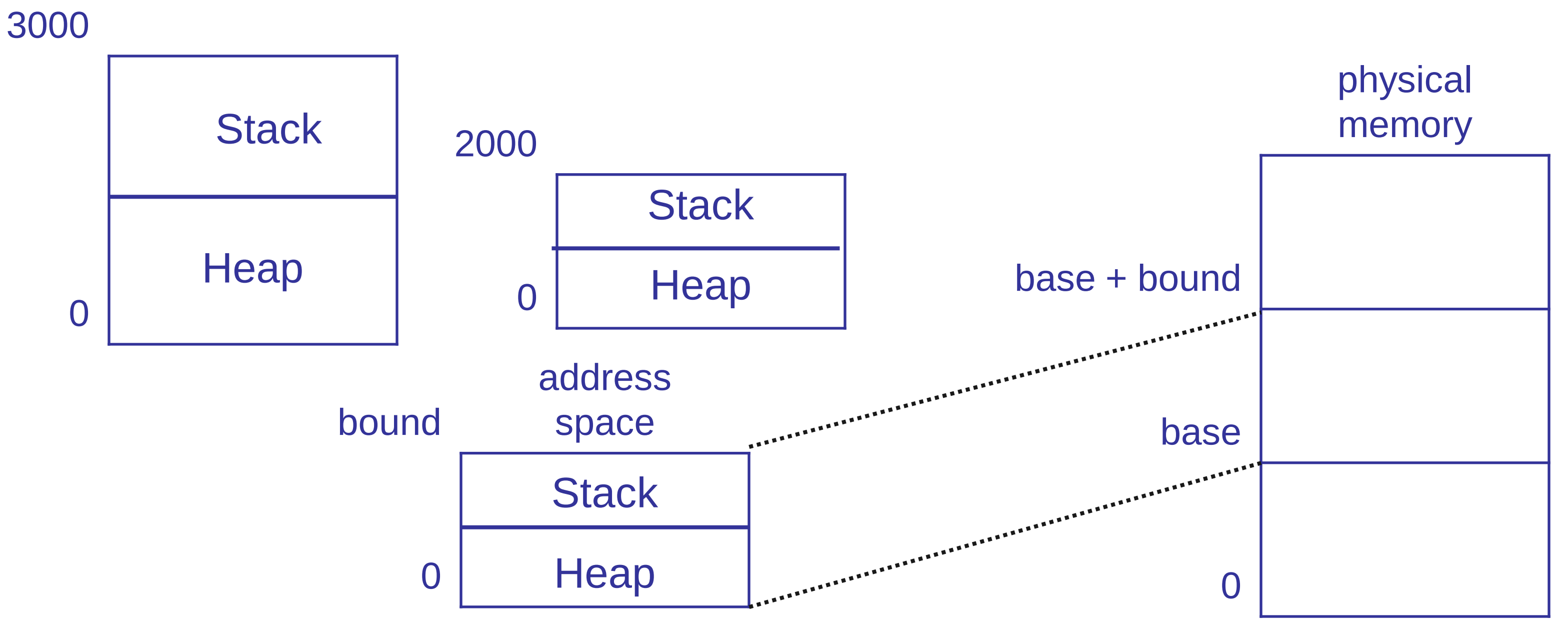
Difficult to grow the address space



Load each process into a contiguous region of physical memory.

There are no holes in the memory unless we create them.

It’s a contiguous address space

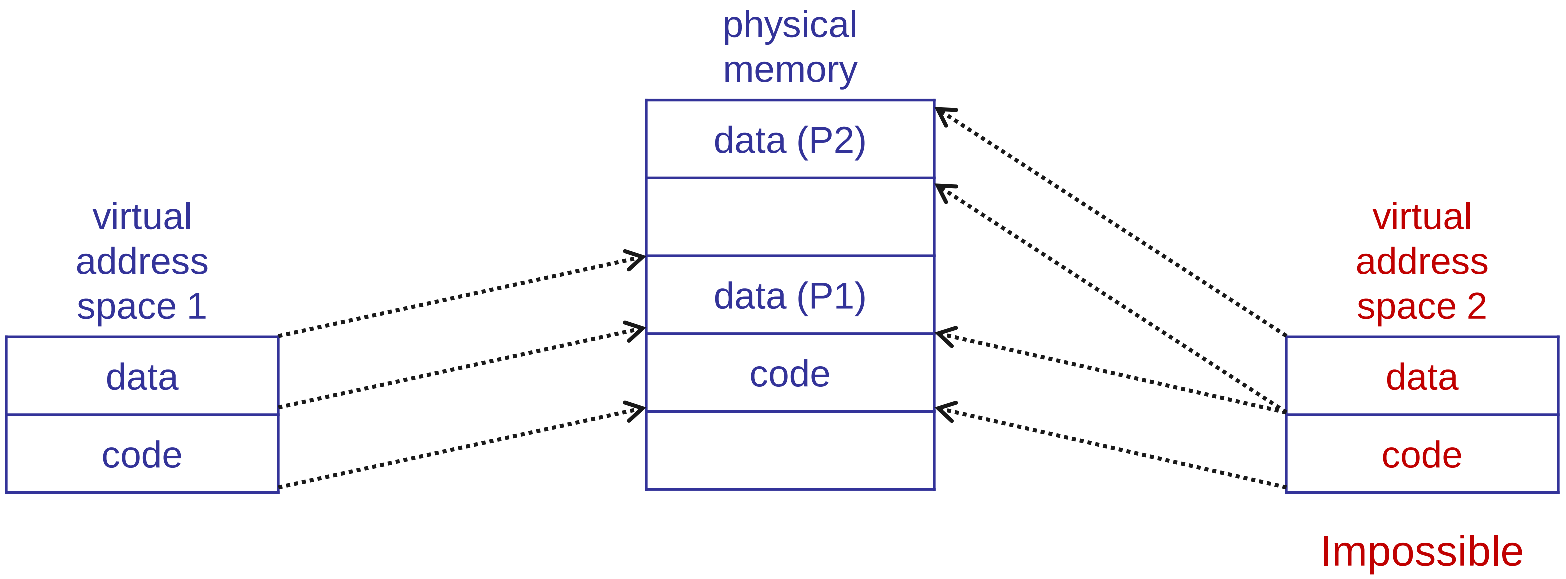


How can stack and heap grow independently?

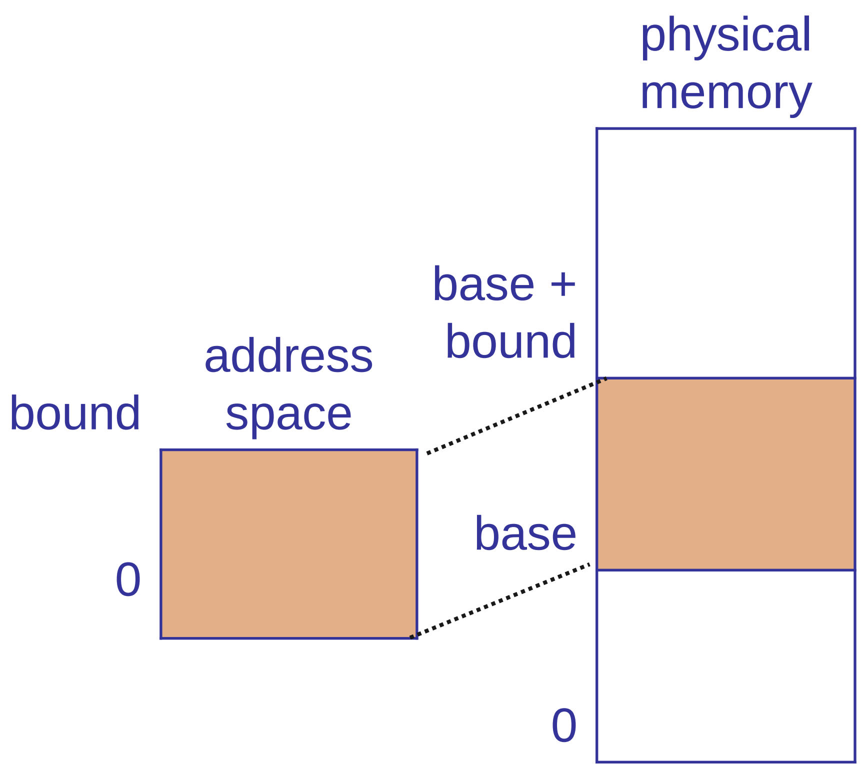
There are no “holes” in the address space, so growing the heap by 1000 bytes requires adding 1000 to every address in stack unless you leave a lot unused physical memory in the middle.

Can’t share memory

Can’t share part of an address space between processes.



Base and bounds



Pros:

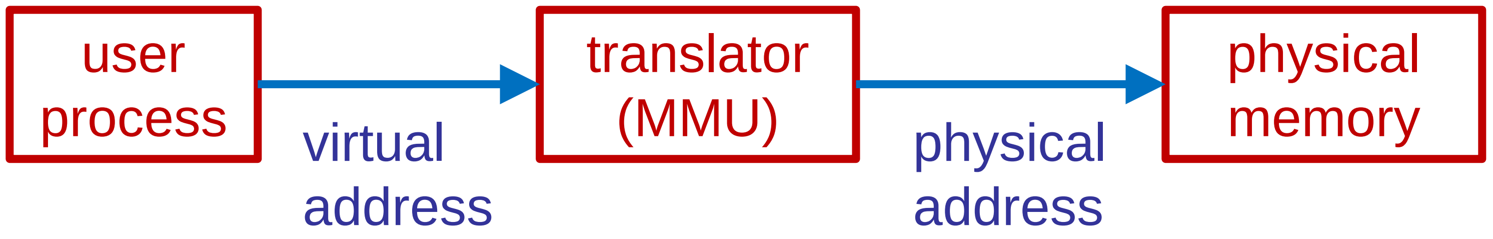
1. Fast.
2. Simple hardware support.

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:

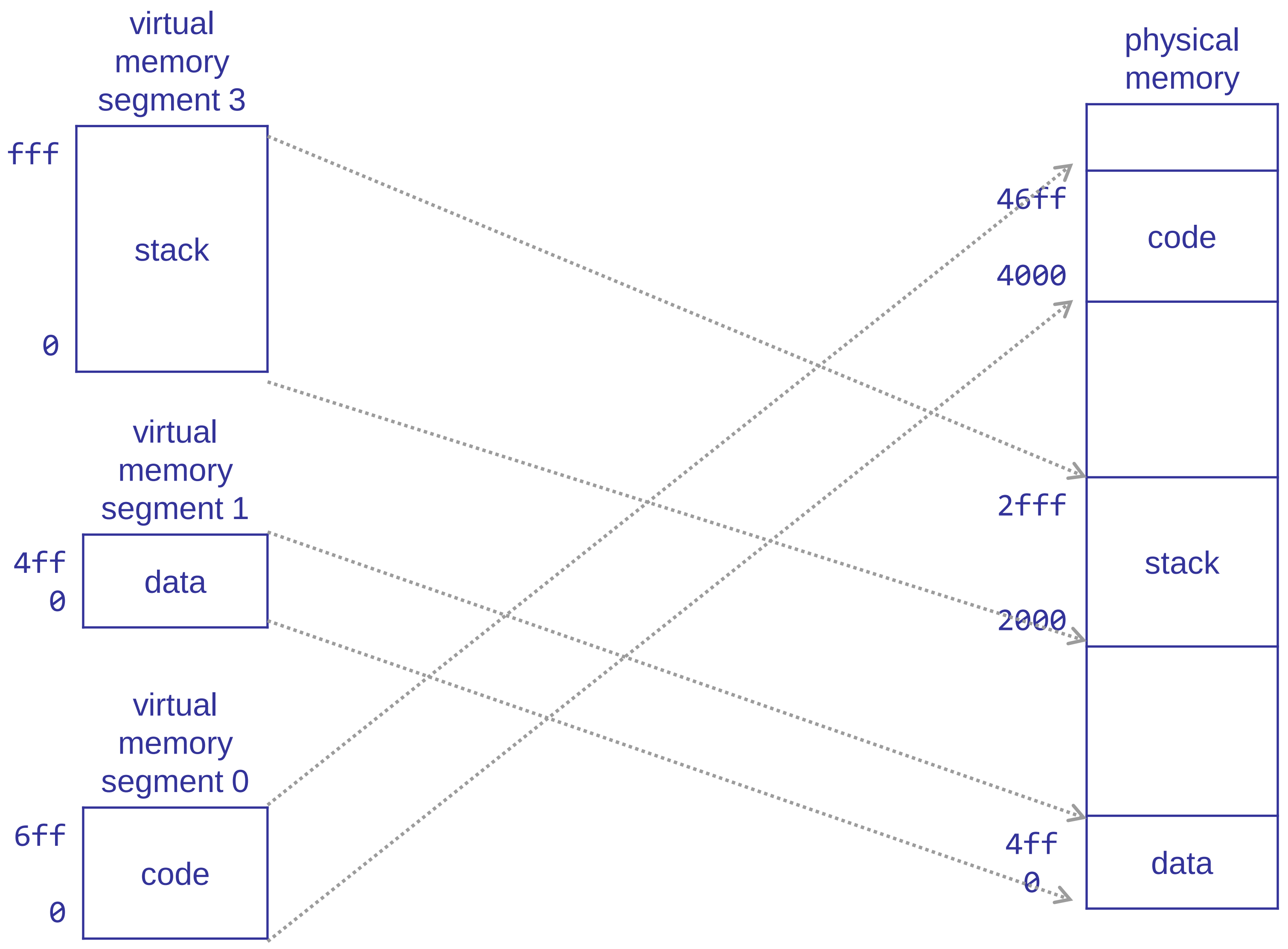
1. Base and bounds.
2. Segmentation.
3. Paging.

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.

Segmentation

How to grow a segment?

Different segments can have different protection

- E.g., code is usually read only (allows fetch, load,...)
- E.g., data is usually read/write (allows load, store,...)
- Fine-grained protection in base and bounds?

What must be changed on a context switch?

Benefits of Segmentation

Multiple areas of address space can grow separately.

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

Drawbacks of Segmentation

1. External fragmentation
2. Can address space be larger than physical memory?

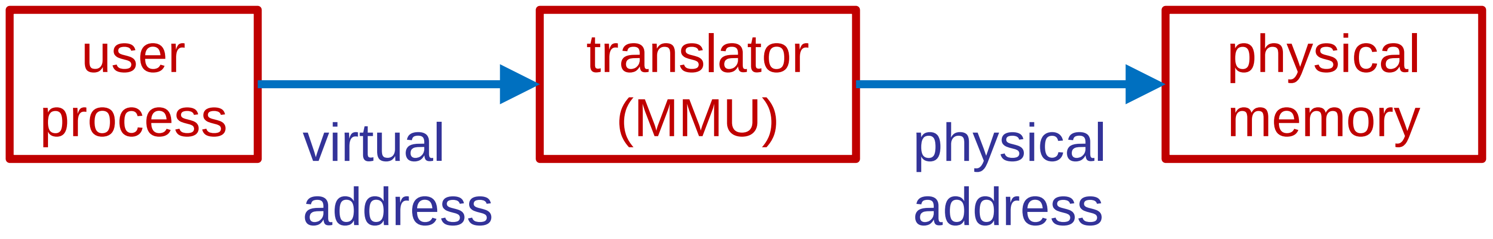
How can we:

1. Make memory allocation easy?
2. Not have to worry about external fragmentation?
3. Allow address space size to be > physical memory?

Next up …

Paging

Next time …



MMU strategies we'll discuss:

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