

CPSC 213

Introduction to Computer Systems

Unit 2d

Virtual Memory

Reading

- ▶ Companion

- 5

- ▶ Text

- 2ed: 9.1-9.2, 9.3.2-9.3.4

- 1ed: 10.1-10.2, 10.3.2-10.3.4

Multiple Concurrent Program Executions

- ▶ So far we have
 - a single program
 - multiple threads
- ▶ Allowing threads from different program executions
 - we often have more than one thing we *want* to do at once(ish)
 - threads spend a lot of time blocked, allowing other threads to run
 - but, often there aren't enough threads in one program to fill all the gaps
- ▶ What is a program execution
 - an instance of a program running with its own state stored in memory
 - compiler-assigned addresses for all static memory state (globals, code etc.)
 - security and failure semantics suggest memory isolation for each execution
- ▶ But, we have a problem
 - there is only one memory shared by all programs ...

Virtual Memory

▶ Virtual Address Space

- an abstraction of the *physical* address space of main (i.e., *physical*) memory
- programs access memory using virtual addresses
- hardware translates virtual address to physical memory addresses

▶ Process

- a program execution with a private virtual address space
- associated with authenticated user for access control & resource accounting
- running a program with 1 or more threads

▶ MMU

- memory management unit
- the hardware that translates virtual address to physical address
- performs this translation on **every** memory access by program

Implementing the MMU

▶ Lets think of this in the simulator ...

- introduce a class to simulate the MMU hardware

```
class MMU extends MainMemory {
    byte []      physicalMemory;
    AddressSpace currentAddressSpace;

    void setAddressSpace (AddressSpace* as);

    byte readByte (int va) {
        int pa = currentAddressSpace.translate (va);
        return physicalMemory.read (pa);
    }
}
```

- currentAddressSpace is a hardware register
- the address space performs virtual-to-physical address translation

Implementing Address Translation

```
class MMU extends MainMemory {  
    byte []      physicalMemory;  
    AddressSpace currentAddressSpace;  
  
    void setAddressSpace (AddressSpace* as);  
  
    byte readByte (int va) {  
        int pa = currentAddressSpace.translate (va);  
        return physicalMemory.read (pa);  
    }  
}
```

▶ Goal

- translate any virtual address to a unique physical address (or none)
- fast and efficient hardware implementation

▶ Lets look at a couple of alternatives ...

Base and Bounds

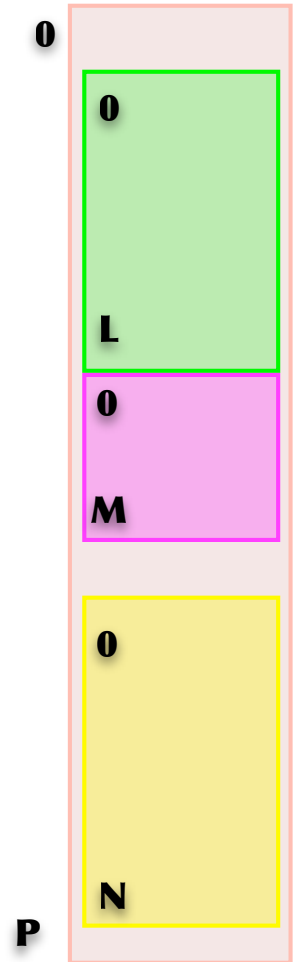
▶ An address space is

- a single, variable-size, non-expandable chunk of physical memory
- named by its base physical address and its length

▶ As a class in the simulator

```
class AddressSpace {
    int baseVA, basePA, bounds;

    int translate (int va) {
        int offset = va - baseVA;
        if (offset < 0 || offset > bounds)
            throw new IllegalArgumentException ();
        return basePA + offset;
    }
}
```



▶ Problems

But, Address Space Use May Be Sparse

▶ Issue

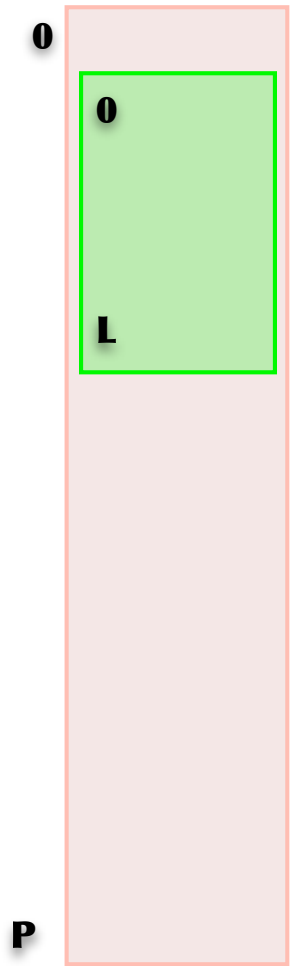
- the address space of a program execution is divided into regions
- for example: code, globals, heap, shared-libraries and stack
- there are large gaps of unused address space between these regions

▶ Problem

- a single base-and-bounds mapping from virtual to physical addresses
- means that gaps in virtual address space will waste physical memory
- this is the **Internal Fragmentation** problem



▶ Solution

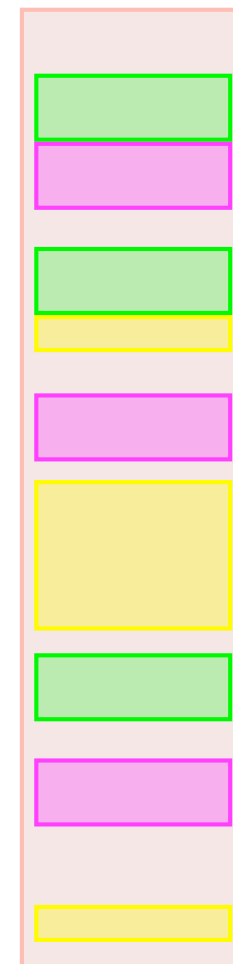


Segmentation

- ▶ An address space is
 - a set of segments
- ▶ A segment is
 - a single, variable-size, non-expandable chunk of physical memory
 - named by its base virtual address, physical address and length
- ▶ Implementation in Simulator

```
class AddressSpace {
    Segment segment[];

    int translate (int va) {
        for (int i=0; i<segments.length; i++) {
            int offset = va - segment[i].baseVA;
            if (offset >= 0 && offset < segment[i].bounds) {
                pa = segment[i].basePA + offset;
                return pa;
            }
        }
        throw new IllegalArgumentException (va);
    }
}
```



- ▶ Problem

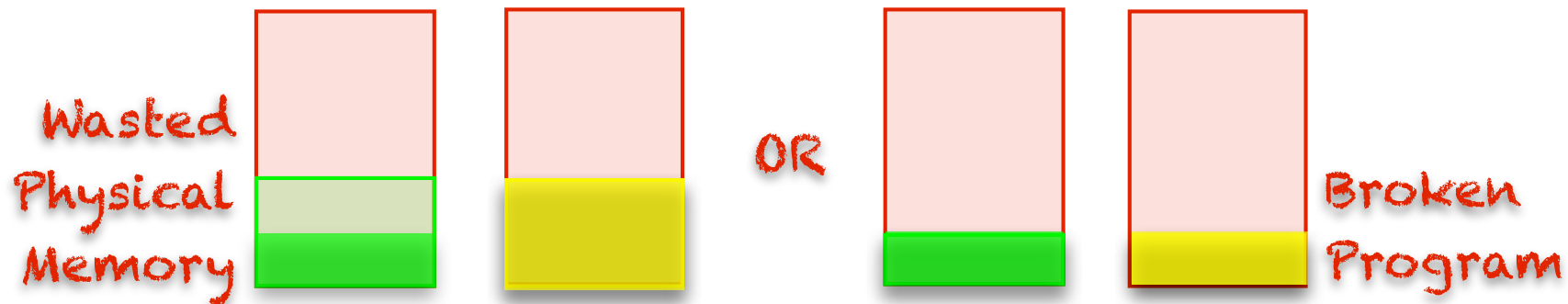
But, Memory Use is Not Know Statically

▶ Issue

- segments are **not expandable**; their size is static
- some segments such as stack and heap change size dynamically

▶ Problem

- segment size is chosen when segment is created
- too large and internal fragmentation wastes memory
- too small and stack or heap restricted



▶ Solution

- allow segments to expand?

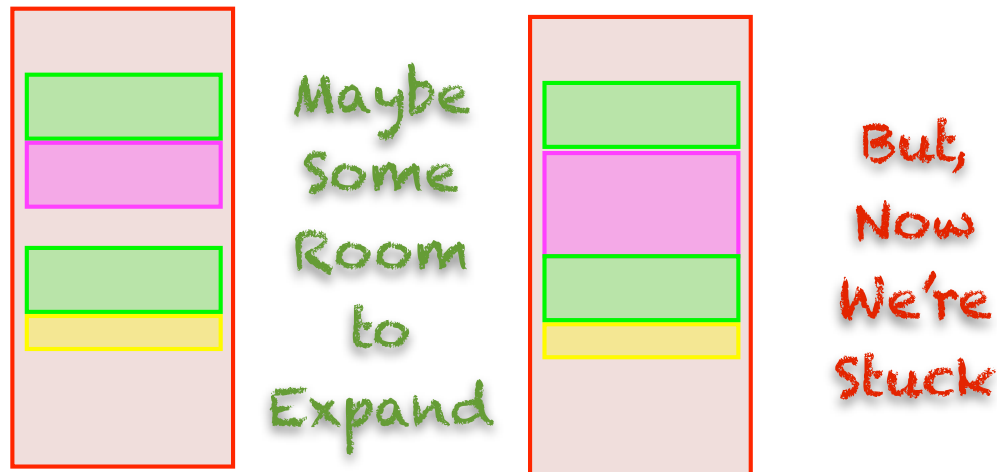
But, There May Be No Room to Expand

▶ Issue

- segments are contiguous chunks of physical memory
- a segment can only expand to fill space between it and the next segment

▶ Problem

- there is no guarantee there will be room to expand a segment
- the available memory space is not where we want it (i.e., adjacent to segment)
- this is the **External Fragmentation** problem



▶ Solution

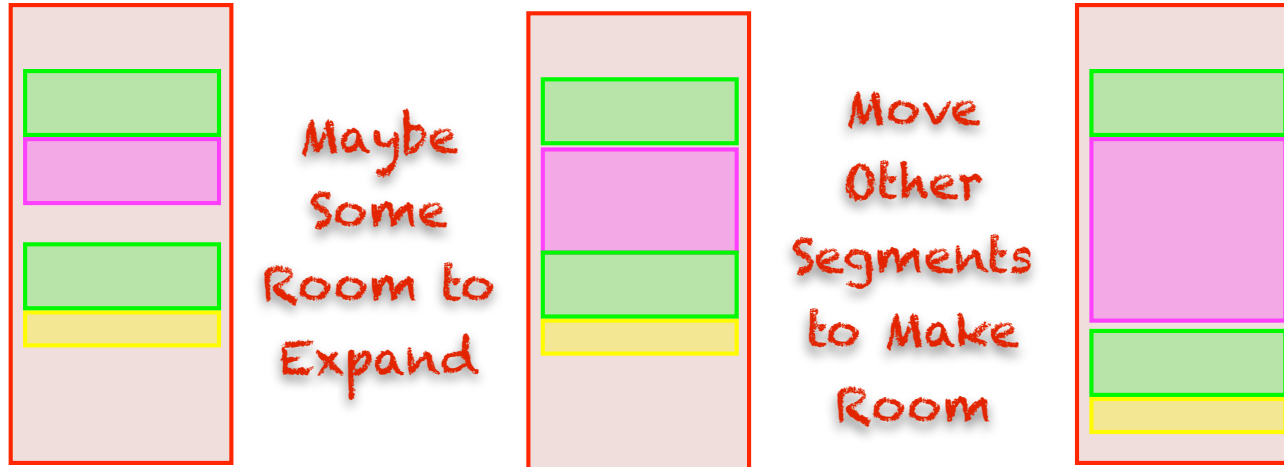
But, Moving Segments is Expensive

▶ Issue

- if there is space in memory to store expanding segment, but not where it is
- could move expanding segment or other segments to make room
- external fragmentation is resolved by moving things to consolidate free space

▶ Problem

- moving is possible, but expensive
- to move a segment, all of its data must be copied
- segments are large and memory copying is expensive



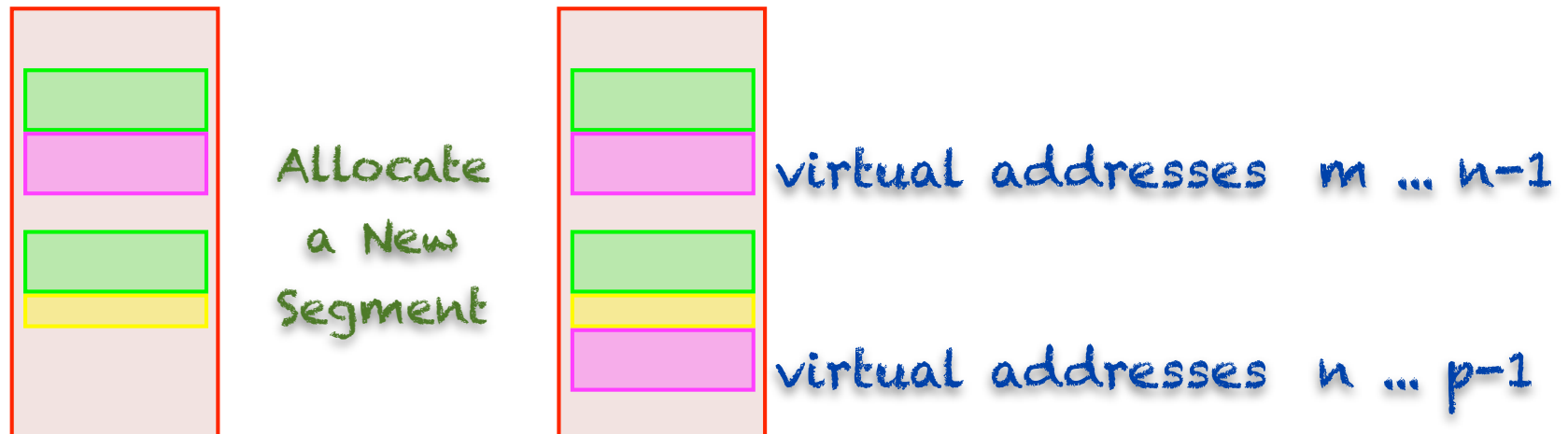
Expand Segments by Adding Segments

▶ What we know

- segments should be non-expandable
- size can not be effectively determined statically

▶ Idea

- instead of expanding a segment
- make a new one that is adjacent virtually, but not physically

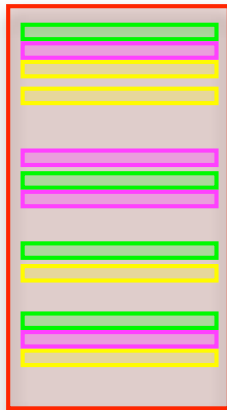


▶ Problem

- oh no! another problem! what is it? why does it occur?

Eliminating External Fragmentation

- ▶ The problem with what we are doing is
 - allocating variable size segments leads to external fragmentation of memory
 - this is an inherent problem with variable-size allocation
- ▶ What about **fixed sized allocation**
 - could we make every segment the same size?
 - this eliminates external fragmentation
 - but, if we make segments too big, we'll get internal fragmentation
 - so, they need to be fairly small and so we'll have lots of them



- ▶ Problem

Translation with Many Segments

- ▶ What is wrong with this approach if there are many segments?

```
class AddressSpace {
    Segment segment[];

    int translate (int va) {
        for (int i=0; i<segments.length; i++) {
            int offset = va - segment[i].baseVA;
            if (offset > 0 && offset < segment[i].bounds) {
                pa = segment[i].basePA + offset;
                return pa;
            }
        }
        throw new IllegalArgumentException (va);
    }
}
```

- ▶ Now what?

- is there another way to locate the segment, when segments are fixed size?

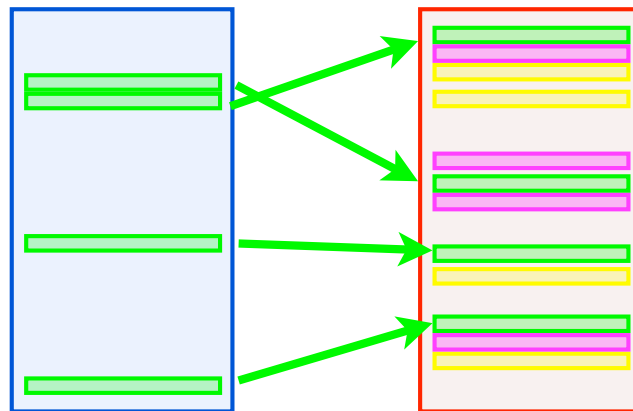
Paging

▶ Key Idea

- Address Space is divided into set of fixed-size segments called pages
- number pages in virtual address order
- page number = virtual address / page size

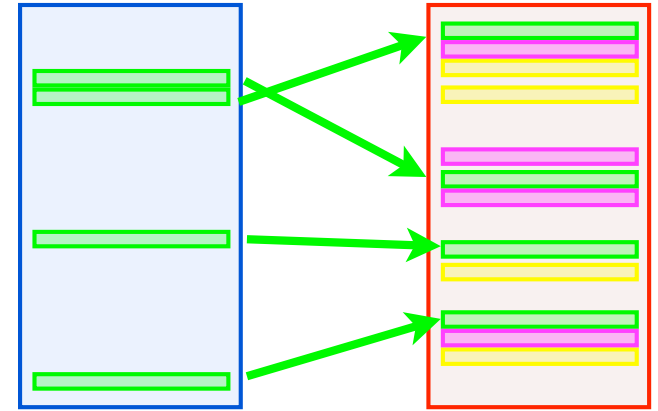
▶ Page Table

- indexed by virtual page number (vpn)
- stores **base physical address** (actually address / page size (pfn) to save space)
- stores **valid flag**, because some segment numbers may be unused



▶ New terminology

- **page** a small, fixed-sized (4-KB) segment
- **page table** virtual-to-physical translation table
- **pte** page table entry
- **vpn** virtual page number
- **pfm** physical page frame number
- **offset** byte offset of address from beginning of page



▶ Translation using a Page Table

```
class PageTableEntry {
    boolean isValid;
    int     pfn;
}
```

```
class AddressSpace {
    PageTableEntry pte[];

    int translate (int va) {
        int vpn    = va / PAGE_SIZE;
        int offset = va % PAGE_SIZE;
        if (pte[vpn].isValid)
            return pte[vpn].pfn * PAGE_SIZE + offset;
        else
            throw new IllegalArgumentException (va);
    }
}
```

▶ The bit-shifty version

- assume that page size is 4-KB = 4096 = 2^{12}
- assume addresses are 32 bits
- then, vpn and pfn are 20 bits and offset is 12 bits
- pte is pfn plus valid bit, so 21 bits or so, say 4 bytes
- page table has 2^{20} pte's and so is 4-MB in size

▶ The simulator code

```
class PageTableEntry {  
    boolean isValid;  
    int     pfn;  
}
```

```
class AddressSpace {  
    PageTableEntry pte[];  
  
    int translate (int va) {  
        int vpn     = va >>> 12;  
        int offset = va & 0xfff;  
        if (pte[vpn].isValid)  
            return pte[vpn].pfn << 12 | offset;  
        else  
            throw new IllegalArgumentException (va);  
    }}  
}}
```

Question

- ▶ Consider this page table

```
0x00000000  
0x80000007  
0x80000321  
0x8000006b  
0x8000005a  
0x80000040  
0x00000000
```

- ▶ Is 0x43a0 a valid virtual address and if so what is the corresponding physical address?
 - (A) Not valid
 - (B) 0x43a0
 - (C) 0x5a3a0
 - (D) 0x73a0
 - (E) 0x3a0

Translation and Exceptions

▶ Virtual-to-Physical translation

- occurs on every memory reference
- handled by hardware (sometimes with some software)
- aided by a cache of recent translations
- but, in general requires reading page table entry from memory

▶ Page fault

- is an exception raised by the CPU
- when a virtual address is invalid
- an exception is just like an interrupt, but generated by CPU not IO device
- page fault handler runs each time a page fault occurs

▶ Handling a page fault

- extending the heap or stack, handler can just deliver a new zero-filled page
- what about the code, global variables, or existing parts of heap or stack?

Demand Paging

▶ Key Idea

- some application data is not in memory
- transfer from disk to memory, only when needed

▶ Page Table

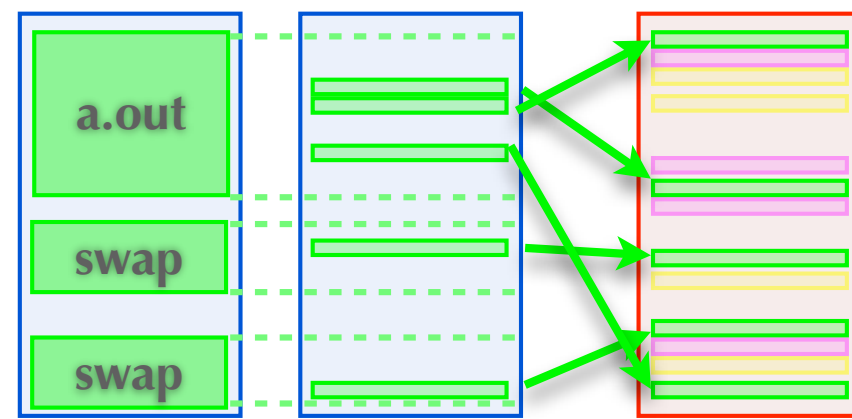
- only stores entries for pages that are in memory
- pages that are only on disk are marked invalid
- access to non-resident page- causes a page-fault interrupt

▶ Memory Map

- a second data structure managed by the OS
- divides virtual address space into regions, each *mapped* to a file
- page-fault interrupt handler checks to see if faulted page is mapped
- if so, gets page from disk, update Page Table and restart faulted instruction

▶ Page Replacement

- pages can now be removed from memory, transparent to program
- a replacement algorithm choose which pages should be resident and swaps out others



Context Switch

▶ A context switch is

- switching between threads from different **processes**
- each process has a private address space and thus its own page table

▶ Implementing a context switch

- change PTBR to point to new process's page table
- switch threads (save regs, switch stacks, restore regs)

▶ Context Switch vs Thread Switch

- changing page tables can be considerably slower than just changing threads
- mainly because caching techniques used to make translation fast

Inter-Process Communication

- ▶ With one process the threads
 - communicate through shared memory
- ▶ Different processes do not share memory
 - they can not communicate in the same way
- ▶ IPC
 - basic mechanism is send and receive unformatted messages
 - a message is an array of bytes
 - sender and receiver have named endpoints (e.g., socket or port)
 - operating system provides the glue
 - the OS can access every processes memory
 - it copies from sender message and into receiver's memory
 - what is send/receive not like?
 - what is send/receive like?

Summary

▶ Process

- a program execution
- a private virtual address space and a set of threads
- private address space required for static address allocation and isolation

▶ Virtual Address Space

- a mapping from virtual addresses to physical memory addresses
- programs use virtual addresses
- the MMU translates them to physical address used by the memory hardware

▶ Paging

- a way to implement address space translation
- divide virtual address space into small, fixed sized virtual page frames
- page table stores base physical address of every virtual page frame
- page table is indexed by virtual page frame number
- some virtual page frames have no physical page mapping
- some of these get data on demand from disk

Address Space Translation Tradeoffs

- ▶ **Single, variable-size, non-expandable segment**
 - internal fragmentation of segment due to sparse address use
- ▶ **Multiple, variable-size, non-expandable segments**
 - internal fragmentation of segments when size isn't known statically
 - external fragmentation of memory because segments are variable size
 - moving segments would resolve fragmentation, but moving is costly
- ▶ **Expandable segments**
 - expansion must be physically contiguous, but there may not be room
 - external fragmentation of memory requires moving segments to make room
- ▶ **Multiple, fixed-size, non-expandable segments**
 - called pages
 - need to be small to avoid internal fragmentation, so there are many of them
 - since there are many, need indexed lookup instead of search