Recap

How to virtualize memory?

- Base-and-bounds:

- The address space of a process is mapped to physical memory as a whole
- A pair of base and bounds registers for a process

- Segmentation:

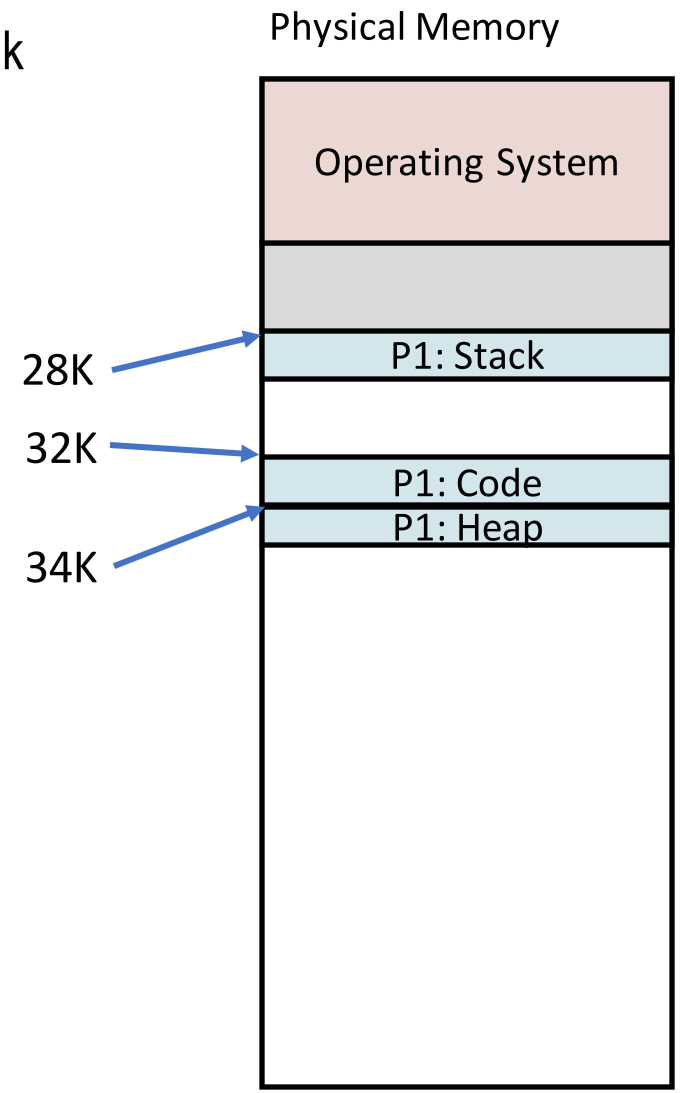
- Divide an address space to multiple (say, 3) segments
- Each segment is mapped to physical memory separately
- Multiple (say, 3) pairs of base and bounds (size) registers are needed for a process

Hardware Requirements for Segmentation

Address space divided to 3 segments: Code (+ static data); Heap; Stack

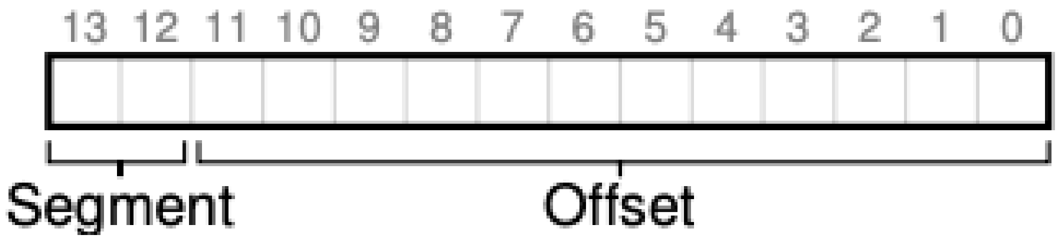
Registers for the start and size of each segment

Segment	Base register	Size register
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



How to Translate Addresses?

Virtual Address:



12

0x3000

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

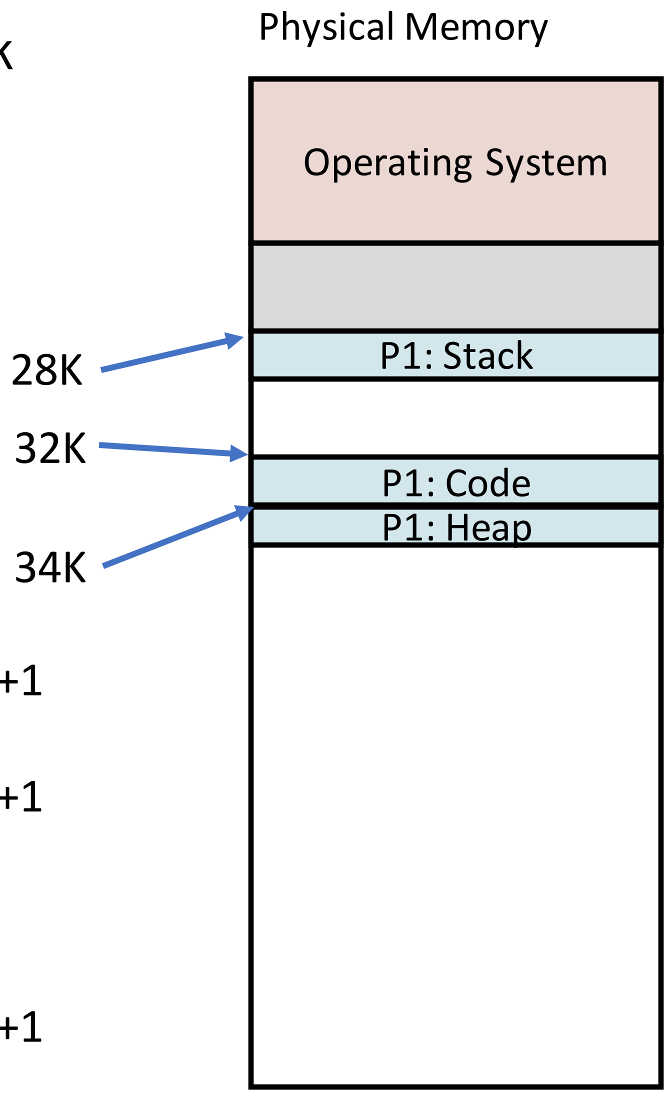
0x0fff

Hardware Requirements for Segmentation

Address space divided to 3 segments: Code (+ static data); Heap; Stack

Registers for the start and size of each segment

Segment	Base register	Size register
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



Virt. address: 00010000000001=>Phy address: 33K+1

Seg#=0, offset=010000000001=1K+1<2K (valid), base(0)+offset=32K+1K+1

Virt. address: 01010000000001=>Phy address: 35K+1

Seg#=1, offset=010000000001=1K+1<2K (valid), base(1)+offset=34K+1K+1

Virt. address: 10110000000001=>Phy address:

Seg#=2, offset=110000000001=3K+1>2K invalid!

Virt. address: 10010000000001=>Phy address:29K+1

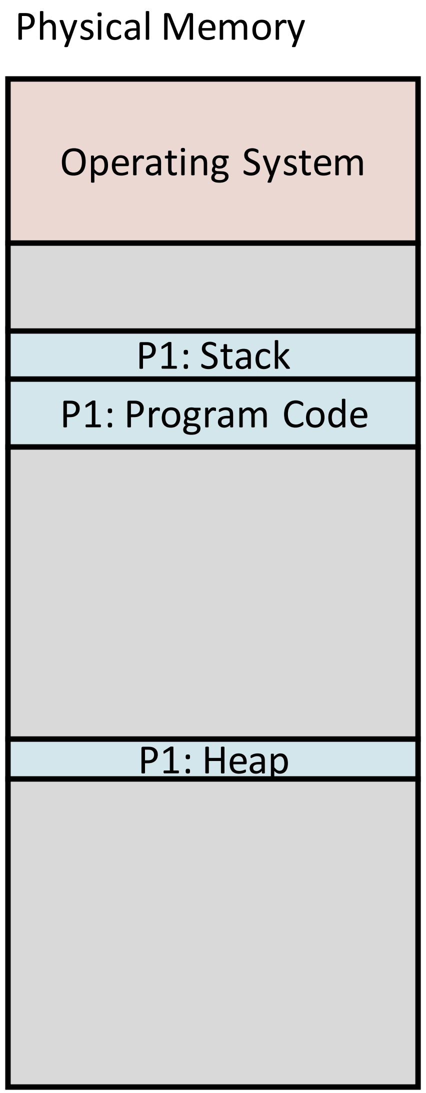
Seg#=2, offset=010000000001=1K+1<2K (valid), base(2)+offset=28K+1K+1

Question

What if physical memory runs out of space for a segment and needs to relocate it? Will pointers in the program need to be updated?

No, address space does not depend on where segments are located in physical memory.

Only base and bounds registers change.



Independent Direction of Segment Growth

We can even allow segments to grow in different directions

A set of registers can indicate if a segment grows up or down

Segment	Base register	Size register (max virt.: 4K)	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	0
Stack	28K	2K	1

Virtual address: (starting from 4K)

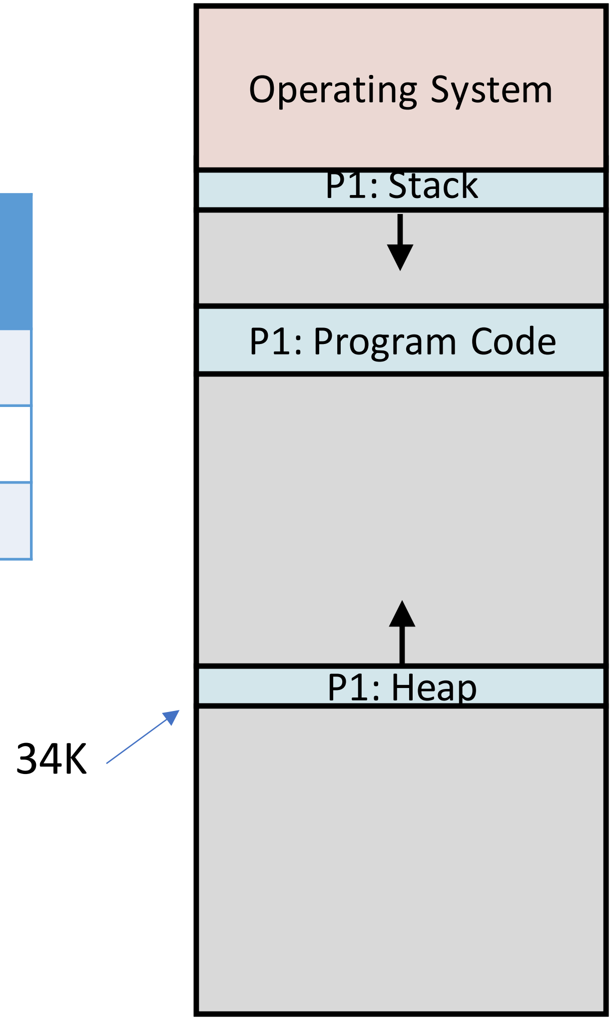
01110000000000 (seg:1, offset:3K)

Heap

It grows negatively

Physical address: 34K – 4K + 3K = 33K

Physical Memory

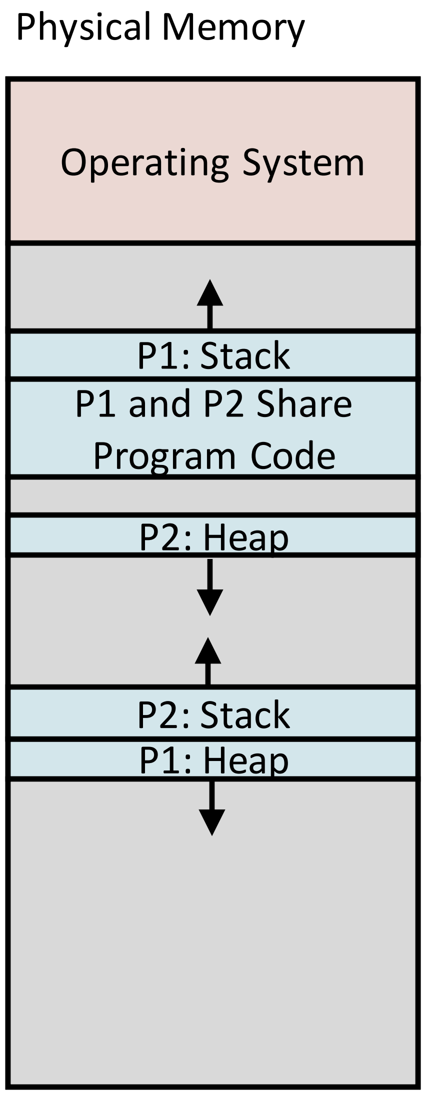


Sharing

Protection registers can enable **sharing**

Example: two processes are executing the same code. If code segment is read-only no danger of processes corrupting each other

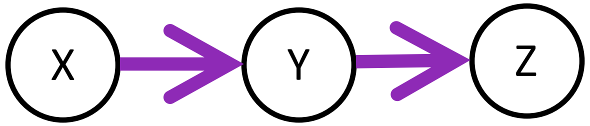
Segment	Base register	Size register	Grows Positive?	Protection
Code	32K	2K	1	Read-execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write



Free Memory

Segments are in contiguous regions of physical memory

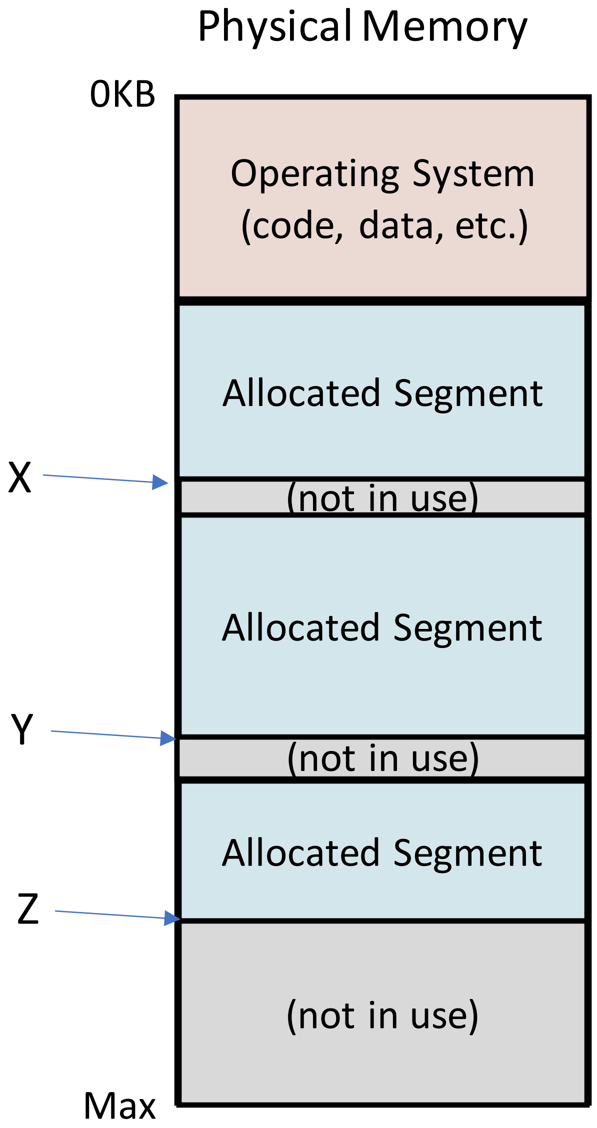
To allocate a new segment, OS must keep a list of free memory



Simple solution is a linked list of free regions of memory

On new allocation search for first open spot that has sufficient memory (**first fit** strategy)

Best fit strategy searches for smallest region of free memory that will fit the segment



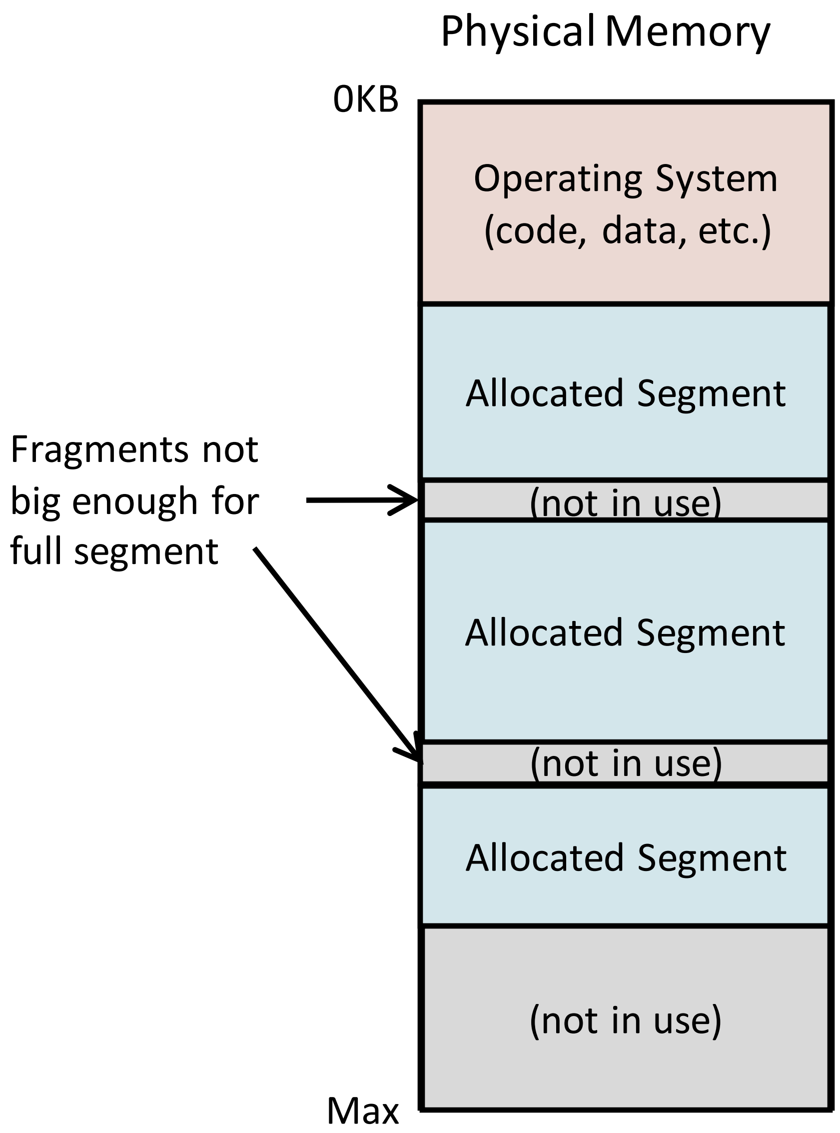
Fragmentation

Segments are in contiguous regions of physical memory

Gaps result in **external fragmentation** (wasted physical memory)

Not big enough to fit a full segment, so can't be used

Compaction used to reclaim the fragments



L9: Paging

(based on Ch 18)

Slides revised from
Matthew Tancreti
Iowa State University

Segmentation ==> Paging

Segmentation made code, stack and heap independently relocatable

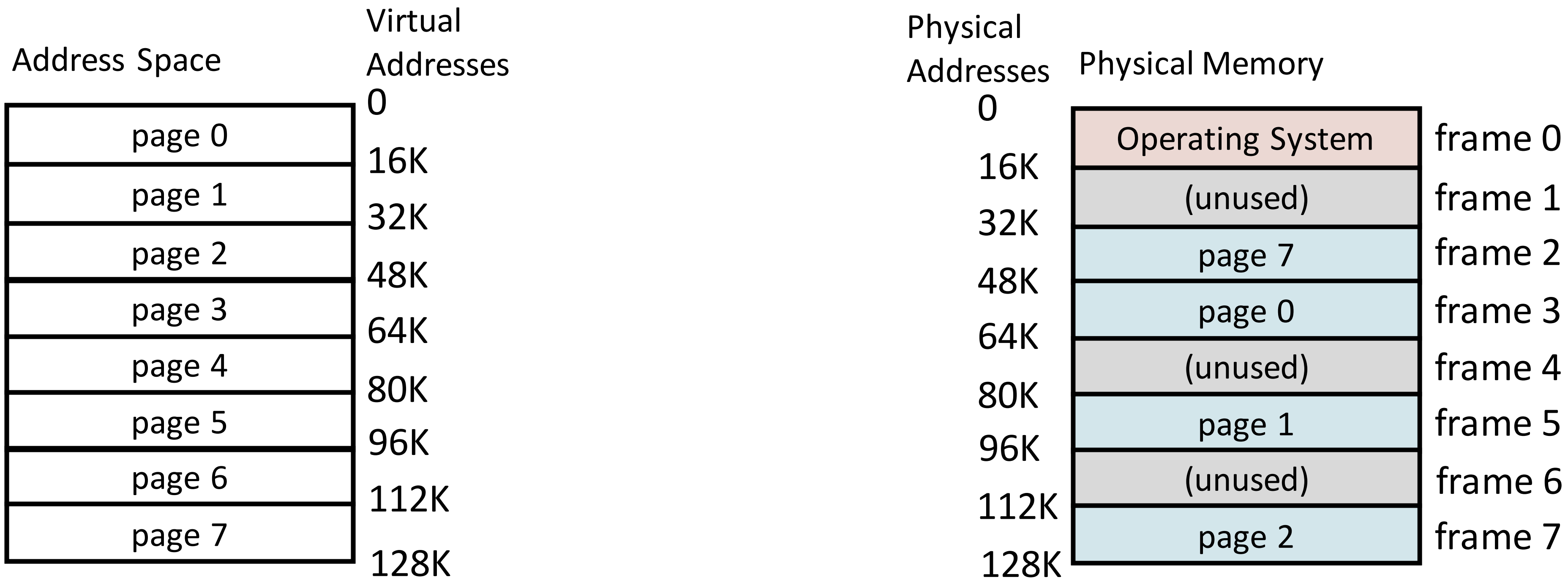
Segments can become arbitrarily large contiguous regions of memory

Resulted in **external fragmentation**

What if we divide the address space into equal size pages?

Paging

Address space divided into equal sized pages that can be stored in frames



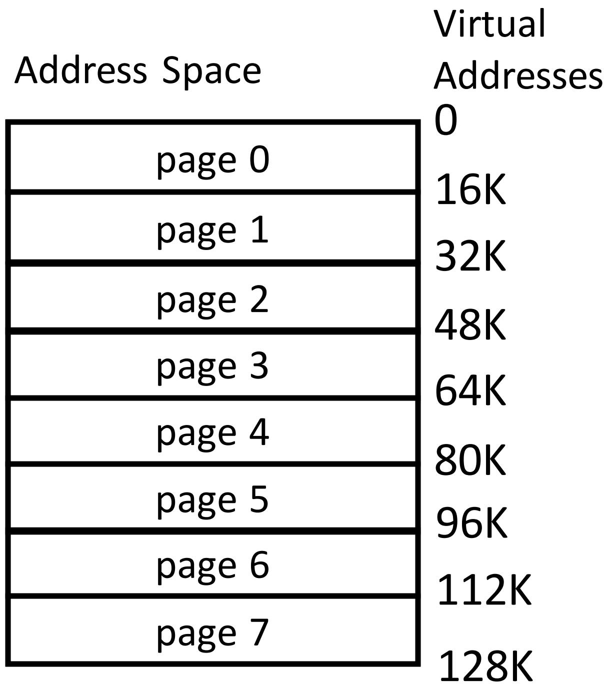
Page Table

One page table for each process

Virtual Page Number (VPN) is the index of the table

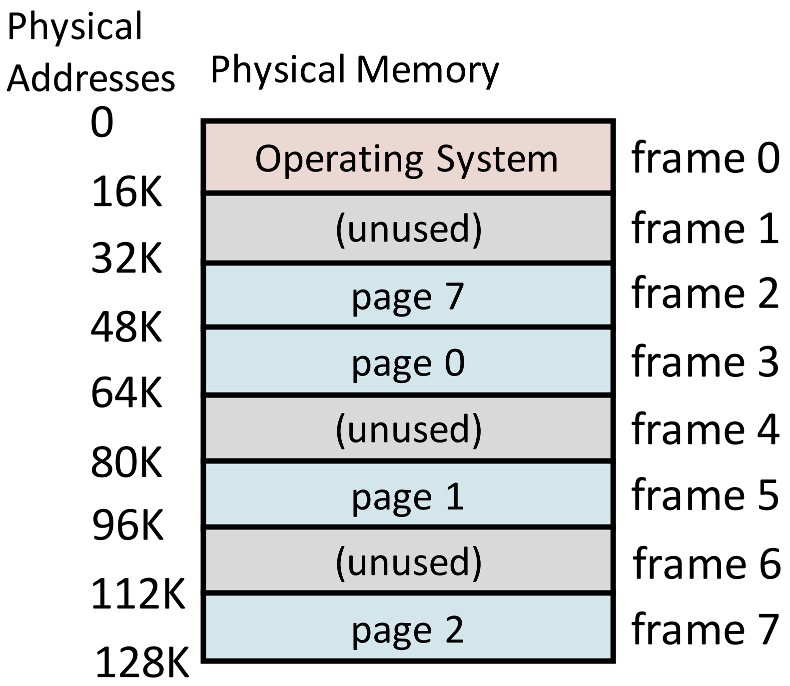
Physical Frame Number (PFN) points to the frame in physical memory

Valid bit indicates if table entry is valid (not all of address space needs to be mapped)



Page Table

VPN	PFN	valid
0	3	1
1	5	1
2	7	1
3	-	0
4	-	0
5	-	0
6	-	0
7	2	1



Virtual Address Bits

Virtual address divided into VPN and offset

	VPN		offset			
Virtual Address	0	1	0	1	0	1

4

```
VPN = VirtualAddress >> NUM_OFFSET_BITS

Offset = VirtualAddress & OFFSET_MASK
```

001111

Virtual Address VPN Bit Size

	VPN		offset			
Virtual Address	0	1	0	1	0	1
	m - n bits		n bits			

If total address space is 2^m bytes and page size is 2^n bytes then

VPN is m-n bits

offset is n bits

Question: If address space is 1GB and page size is 4KB how many bits is the VPN?

Answer: 1GB = 2^{30}, 4KB = 2^{12}, therefore VPN is 30 - 12 = 18 bits

Check: 4KB * 2^{18} = 1GB

Physical Address Bits

	PFN		offset			
Physical Address	0	1	0	1	0	1
	m - n bits		n bits			

If physical memory is 2^m bytes and frame (also page) size is 2^n bytes then

PFN is m-n bits

offset is n bits

Therefore, calculate physical address as:

```
PhysAddr = PFN * frame_size + offset
```

or in binary arithmetic:

```
PhysAddr = (PFN << NUM_OFFSET_BITS) | offset
```

Keep in mind, total physical memory and address space size may not be the same

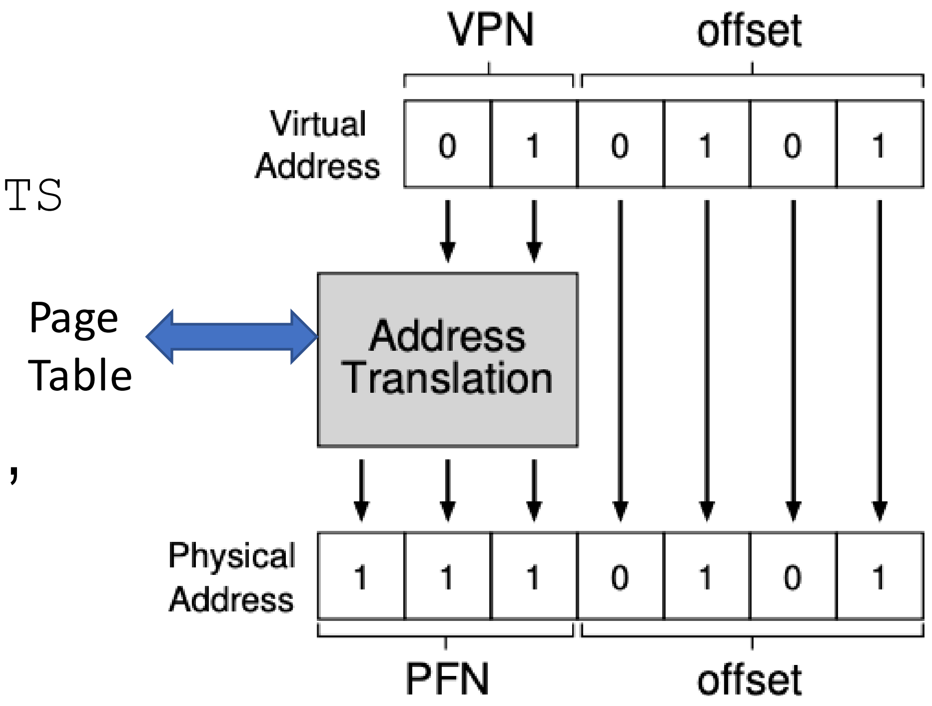
Question: Assume the page size is 4KB, what is the address for frame 10 and offset 128?

Answer: 4KB * 10 + 128 = 41,088

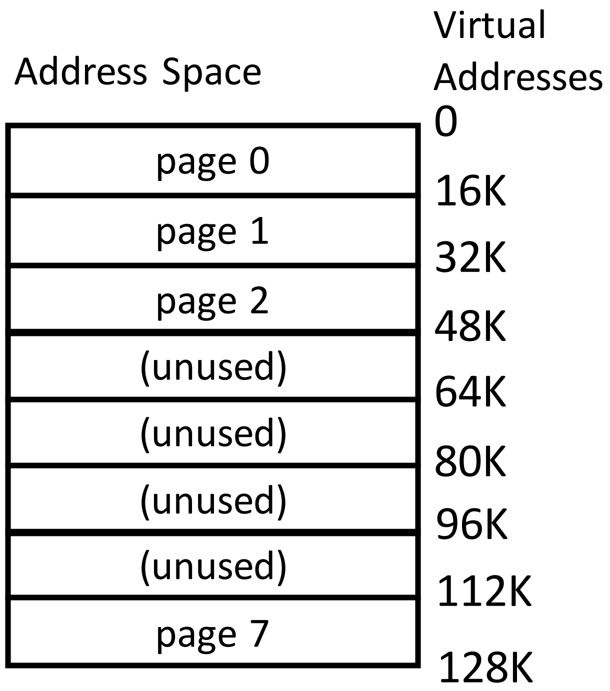
Address Translation

Steps performed **in hardware** (MMU):

1. Compute page number
   `VPN = VirtualAddress >> NUM_OFFSET_BITS`
2. Look up page in page table to find frame number
3. Confirm that access to page is allowed (e.g., valid bit set)
4. Compute physical address
   PhysAddr = (PFN << NUM_OFFSET_BITS) | offset

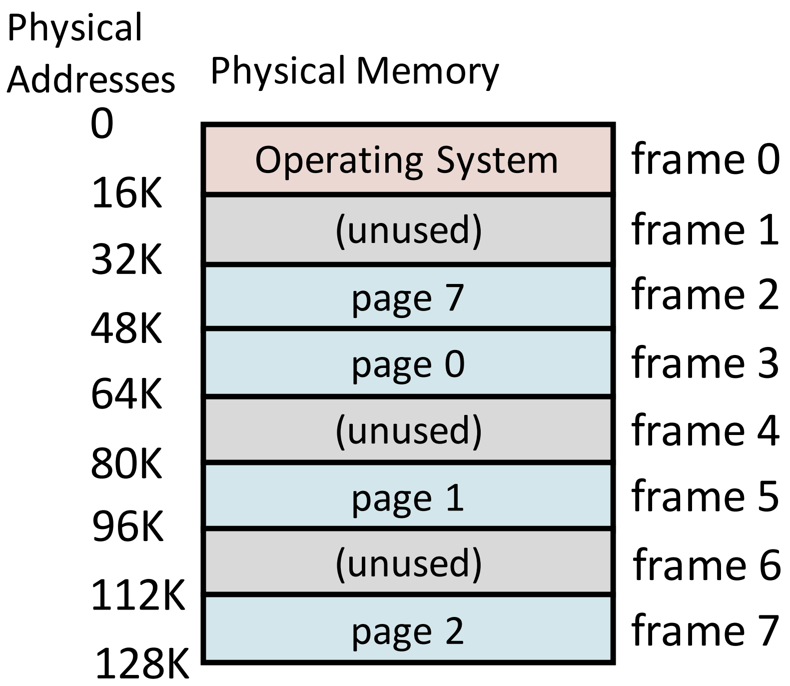


Examples



Page Table

VPN	PFN	valid
0	2	1
1	5	1
2	7	1
3	-	0
4	-	0
5	-	0
6	-	0
7	2	1



Virt. Address: 01000000000000010 => Phy. Address: 11100000000000010

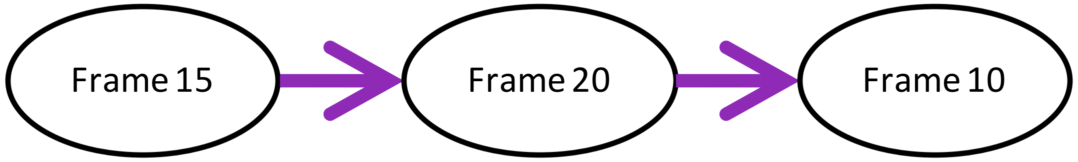
VPN: 2 => check Page Table => PFN: 7

Virt. Address: 11100000000000010 => Phy. Address:

Virt. Address: 10100000000000011 => Phy. Address:

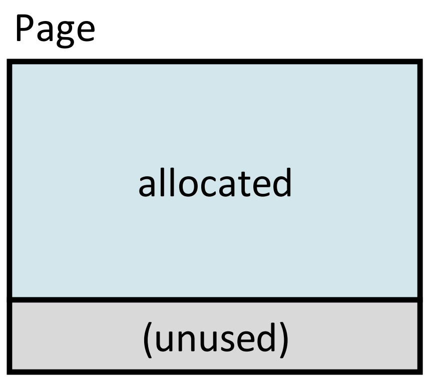
Free Memory Management

OS needs to know which frames are not in use, simple method is linked list



No external fragmentation - no unusable gaps between frames

Pages can have **internal fragmentation** - unused portion of page



Page Size

Tradeoff

- Small page size means bigger page table (more page table entries)
- Bigger page size means more internal fragmentation

Typical page size is 8KB in Linux

Process Control Block (struct proc) in xv6

proc.h

```c
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

enum procstate { UNUSED, USED, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
```

```c
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on channel
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // proc_tree_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Concern

Each process has its own page table which the OS stores in frames in physical memory

Linux page size is 8KB, on pyrite we saw the address space is 140TB

~17 million page table entries for every process!

Linux reduces size with multi-level (3-level tree structure) page tables

Page table lookup is slow, every memory access requires additional memory access(es)! How to speed up memory?