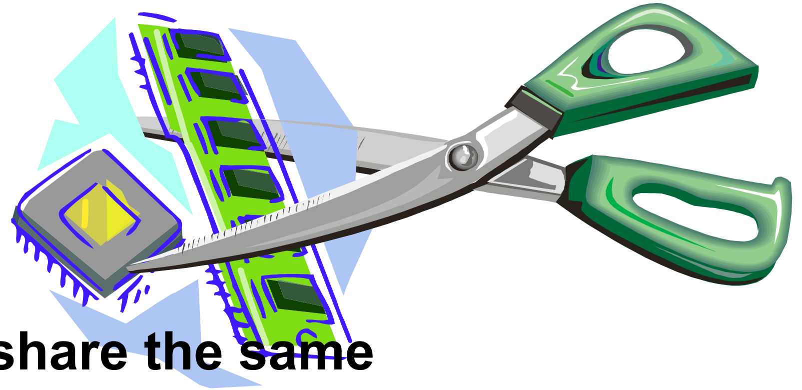
MEMORY MANAGEMENT – PART I

RONG ZHENG

MULTIPROGRAMMING



Physical Reality: Processes/Threads share the same hardware

- Need to multiplex CPU (CPU Scheduling)
- Need to multiplex use of Memory (Today)

Why worry about memory multiplexing?

- The complete working state of a process and/or kernel is defined by its data in memory (and registers)
- Consequently, cannot just let different processes use the same memory
- Probably don't want different processes to even have access to each other's memory (protection)

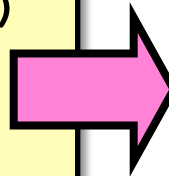
OBJECTIVES OF MEMORY MANAGEMENT

- **Abstraction of exclusive and contiguous logical memory space to processes**
  - Might be larger than the amt of physical memory
- **Allow sharing of memory space across cooperating processes**
- **Protection: ever wonder what is a segmentation fault?**
- **Efficiency**
  - a single memory access typically involves multiple instructions even I/O operations
  - physical memory should be well utilized

BINDING OF INSTRUCTIONS AND DATA TO MEMORY

Process view of memory

```
data1:   dw    32
               …
start:   lw    r1,0(data1)
         jal   checkit
loop:    addi r1, r1, -1
         bnz   r1, loop
         …
checkit: …
```



Physi

```
Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000
```

```
0x0300  00   20
    …       …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
  …
0x0A00
```

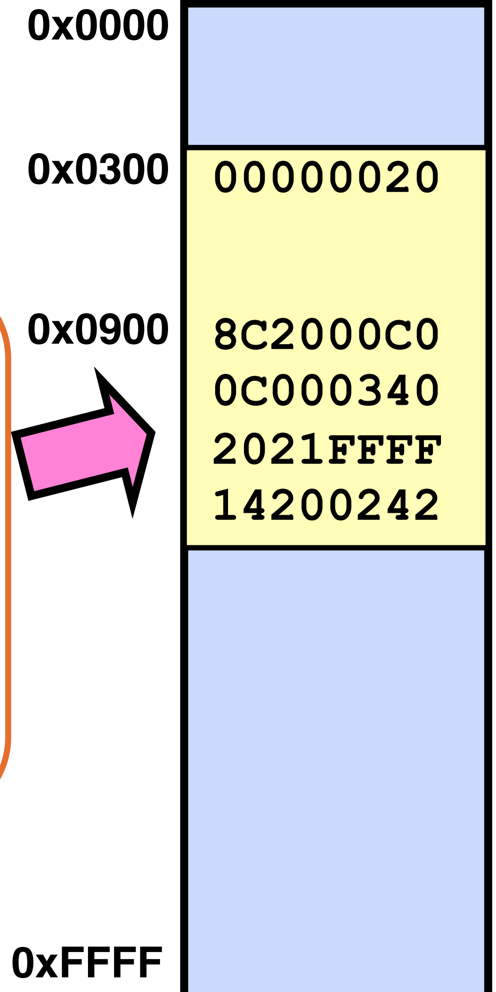
BINDING OF INSTRUCTIONS AND DATA TO MEMORY

Process view of memory

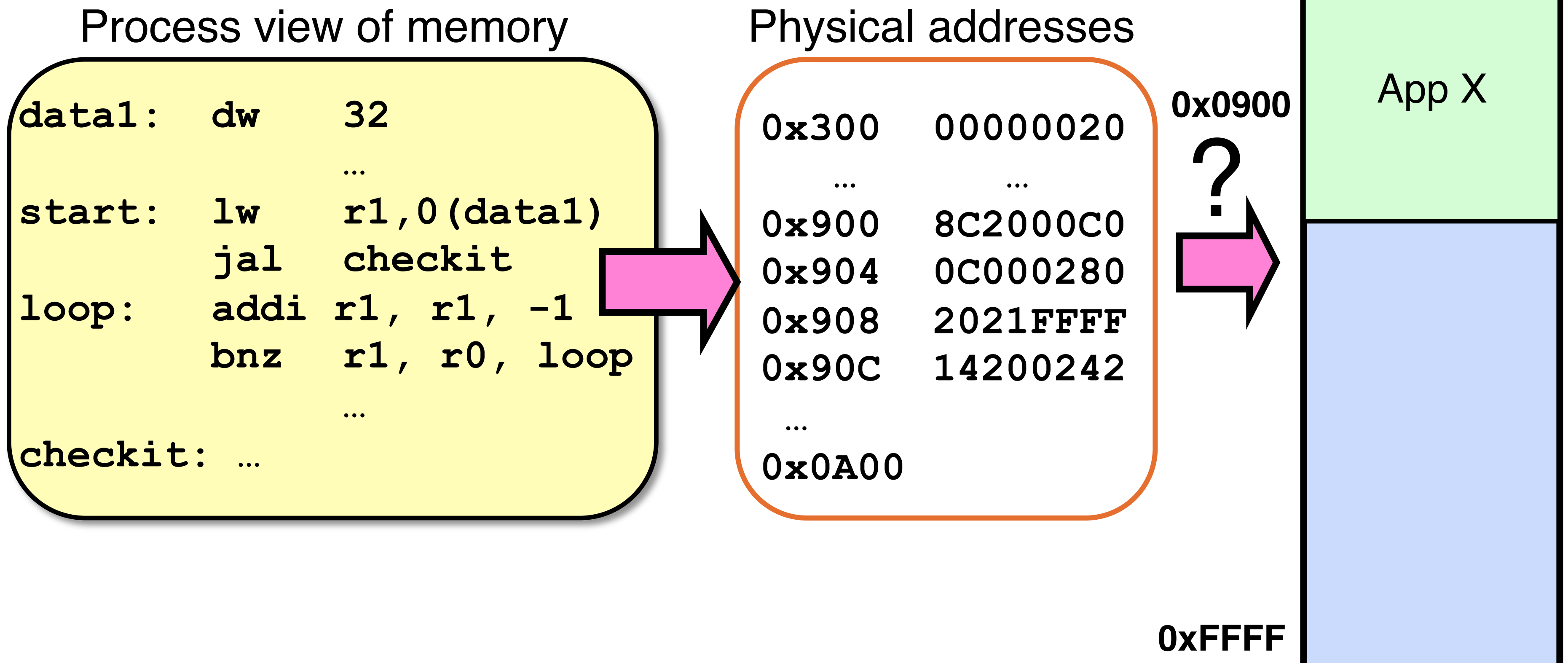
```
data1:   dw     32
                …
start:   lw     r1,0(data1)
         jal    checkit
loop:    addi r1, r1, -1
         bnz    r1, loop
         …
checkit: …
```

Physical addresses

```
0x0300  00000020
   …        …
0x0900  8C2000C0
0x0904  0C000280
0x0908  2021FFFF
0x090C  14200242
 …
0x0A00
```

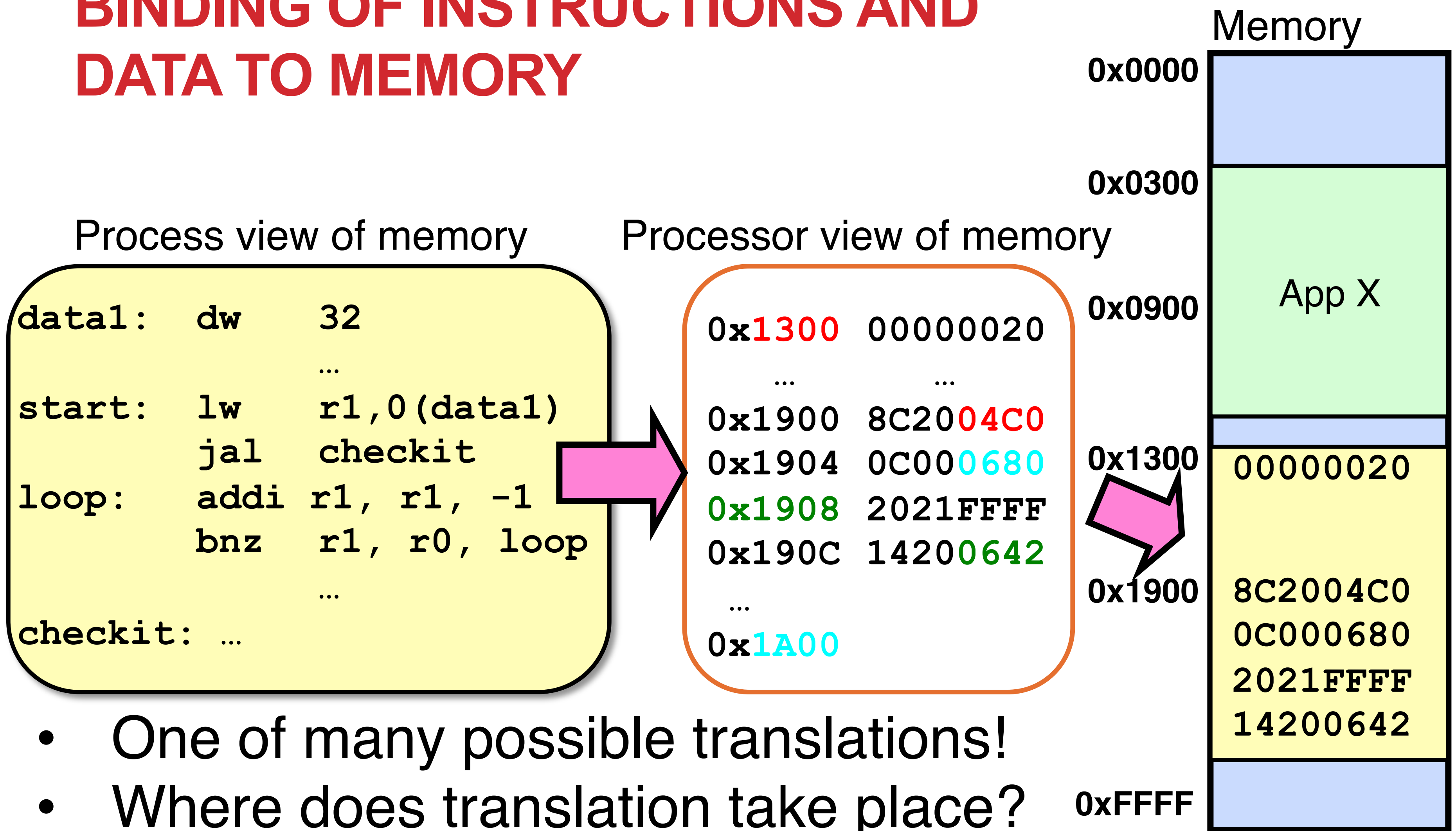


BINDING OF INSTRUCTIONS AND DATA TO MEMORY



Need address translation!

BINDING OF INSTRUCTIONS AND DATA TO MEMORY



- One of many possible translations!
- Where does translation take place?

Compile time, Link/Load time, or Execution time?

MULTI-STEP PROCESSING OF A PROGRAM FOR EXECUTION

Preparation of a program for execution involves components at:

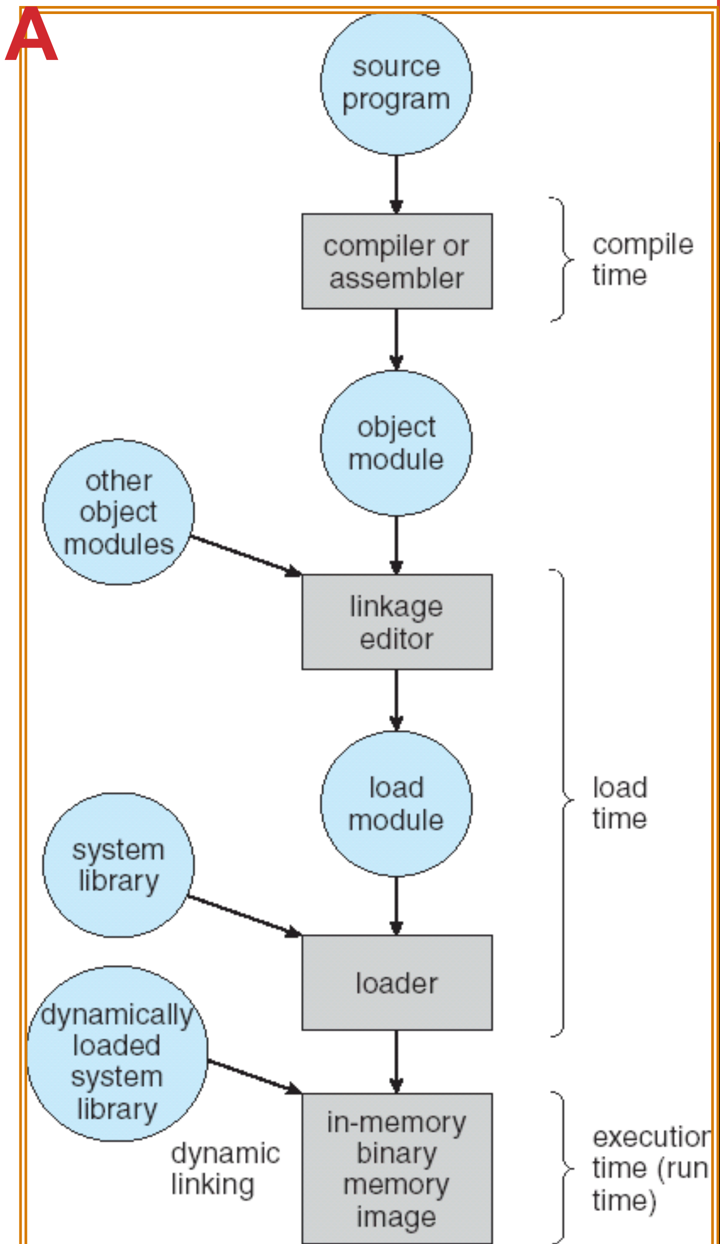
- Compile time (i.e., “gcc”)
- Link/Load time (unix “ld” does link)
- Execution time (e.g. dynamic libs)

Addresses can be bound to final values anywhere in this path

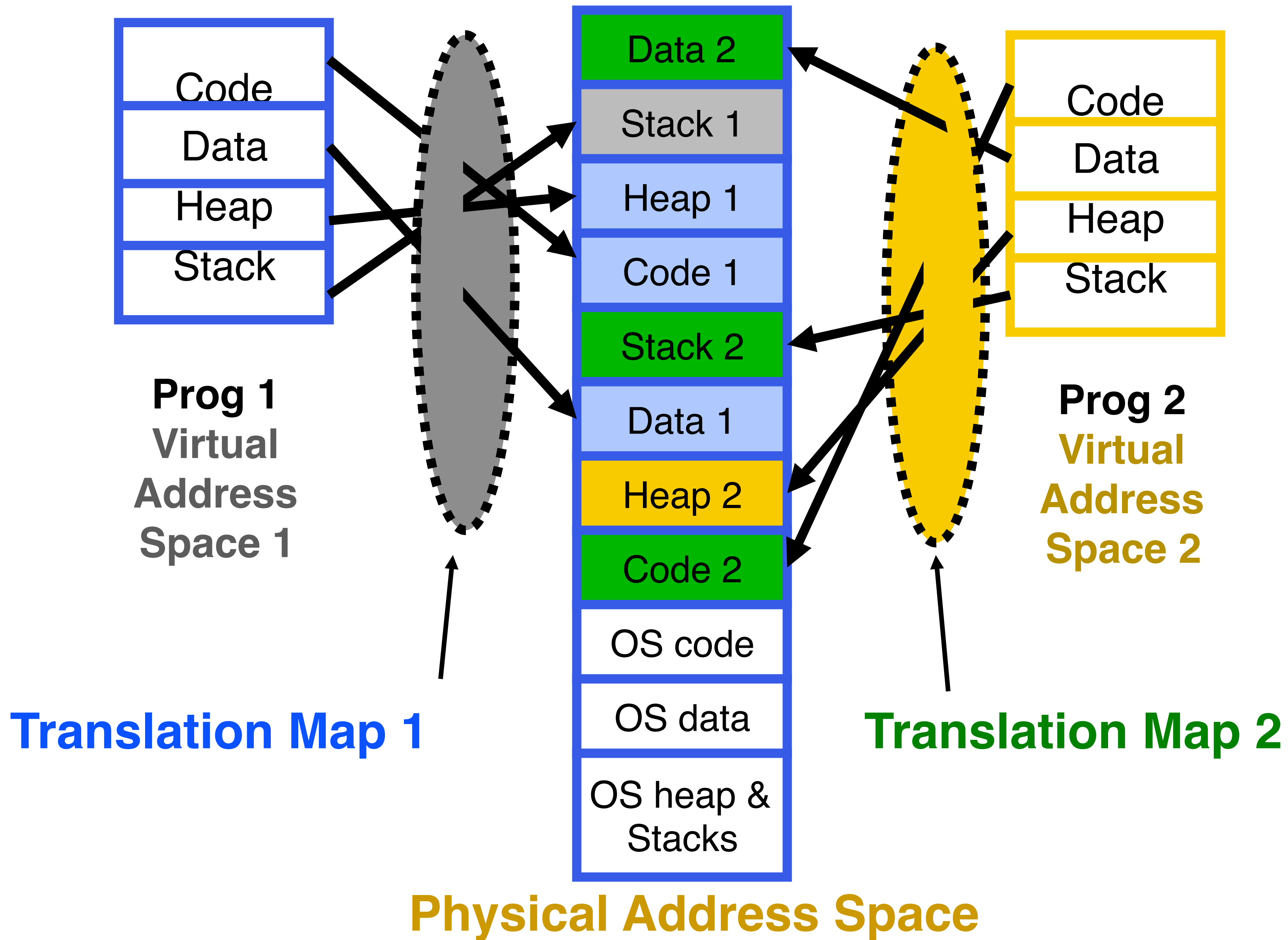
- Depends on hardware support
- Also depends on operating system

Dynamic Libraries

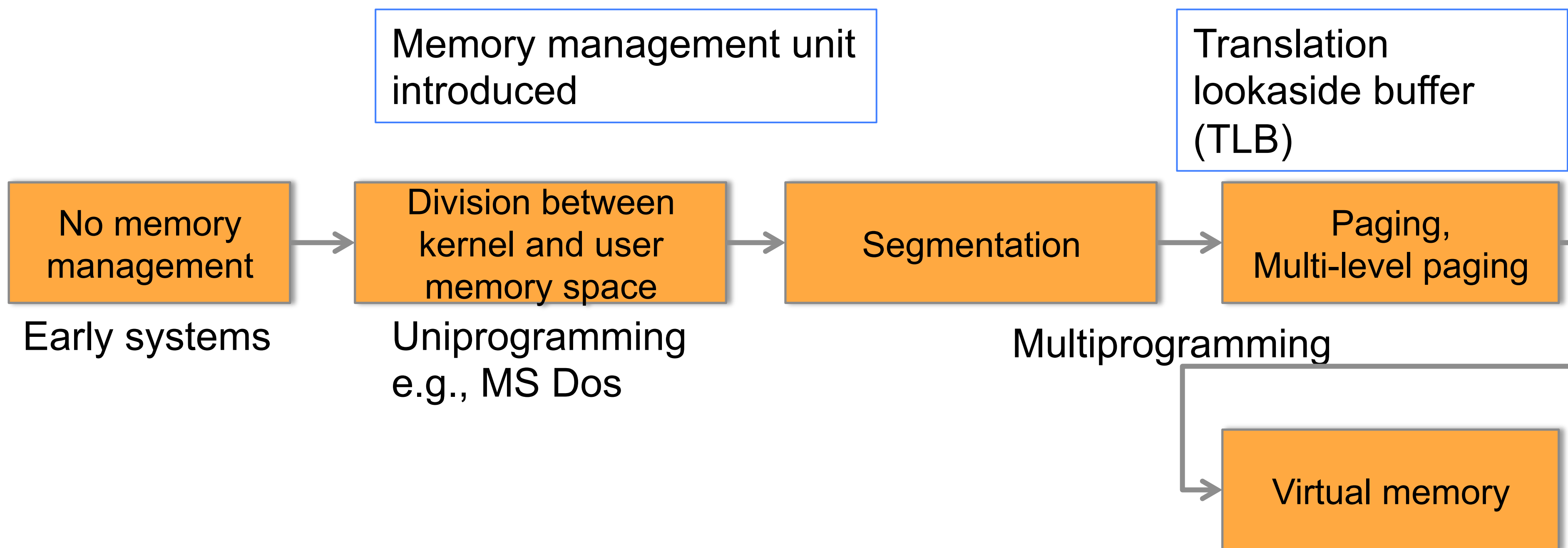
- Linking postponed until execution
- Small piece of code, stub, used to locate appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes routine



EXAMPLE OF GENERAL ADDRESS TRANSLATION



EVOLUTION OF MEMORY MANAGEMENT



Both hardware support and software implementation evolve over time

NO MEMORY MANAGEMENT

The very first computers had no operating system whatsoever

Each programmer

- Had access to whole main memory of the computer
- Had to enter the bootstrapping routine loading his or her program into main memory

Advantage:

- Programmer is in total control of the whole machine.

Disadvantage:

- Much time is lost entering manually the bootstrapping routine.

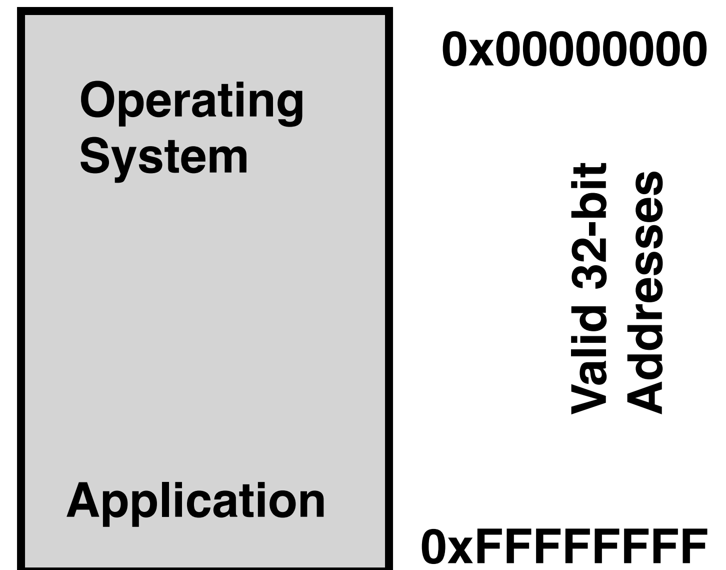
UNIPROGRAMMING

Every system includes a memory-resident monitor

- Invoked every time a user program would terminate
- Would immediately fetch the next program in the queue (batch processing)

Should prevent user program from corrupting the address space of kernel processes

Must add a Memory Management Unit (MMU)



UNIPROGRAMMING

Assuming that the OS occupies memory locations 0 to START – 1

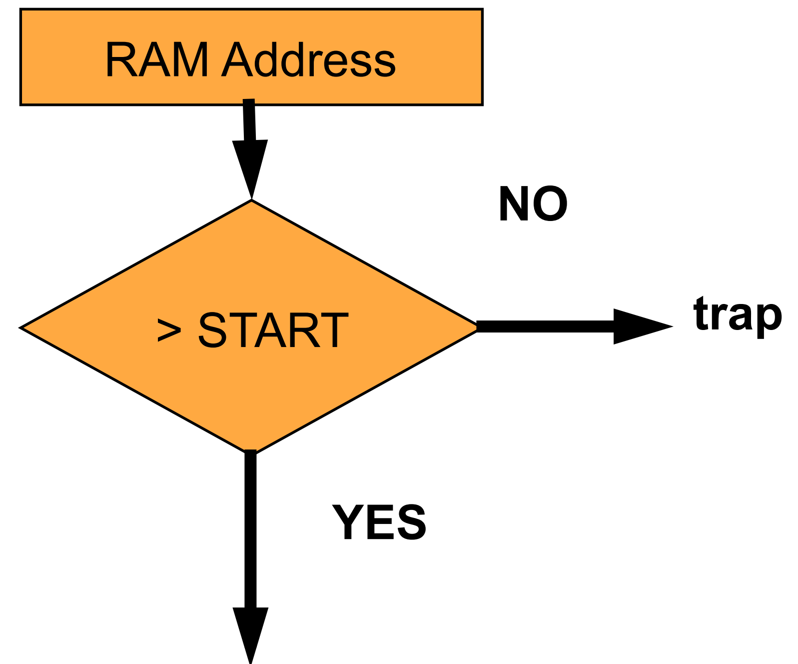
MMU will prevent the program from accessing memory locations 0 to START – 1

Advantage:

- No time is lost re-entering manually the bootstrapping routine

Disadvantage:

- CPU remains idle every time the user program does an I/O.



CONTIGUOUS ADDRESS SPACE AND FIXED PARTITION

Multiprogramming with fixed partitions

- Requires I/O controllers and interrupts

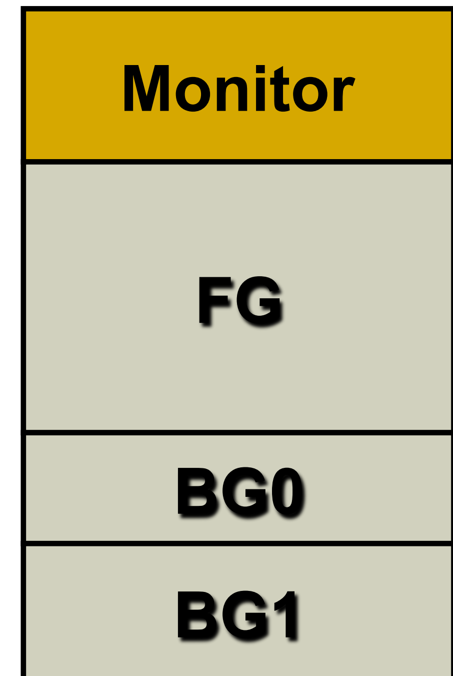
OS dedicates multiple partitions for user processes

- Partition boundaries are fixed

Each process must be confined between its first and last address

Computer often had

- A foreground partition (FG)
- Several background partitions (BG0, . . .)



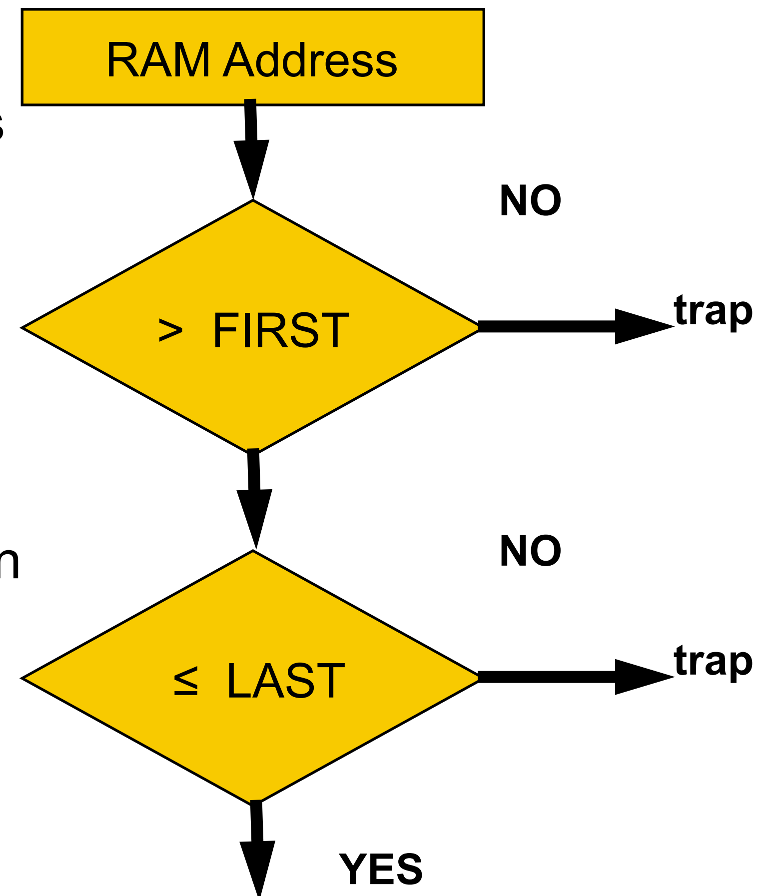
CONTIGUOUS ADDRESS SPACE AND FIXED PARTITION

Advantage:

- No CPU time is lost while system does I/O

Disadvantages:

- Partitions are fixed while processes have different memory requirements
- Many systems were requiring processes to occupy a specific partition



CONTIGUOUS ADDRESS SPACE AND VARIABLE PARTITION

Multiprogramming with variable partitions

OS allocates contiguous extents of memory to processes

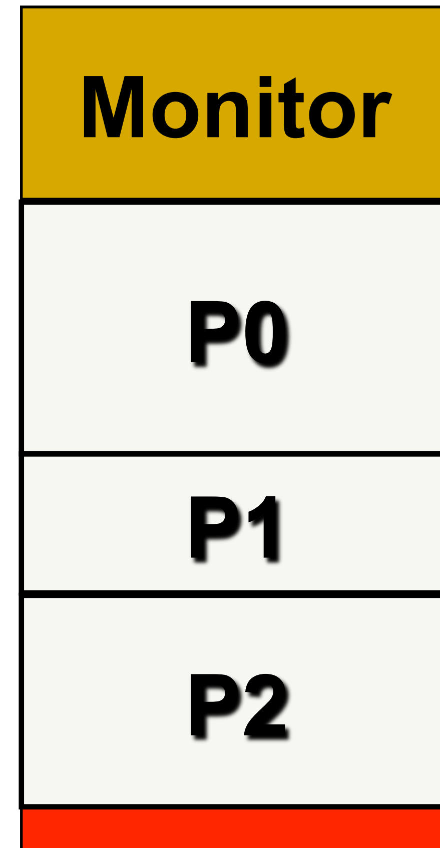
- Initially each process gets all the memory space it needs and nothing more

Processes that are swapped out can return to any main memory location

EXTERNAL SEGMENTATION

Initially everything works fine

- Three processes occupy most of memory
- Unused part of memory is very small

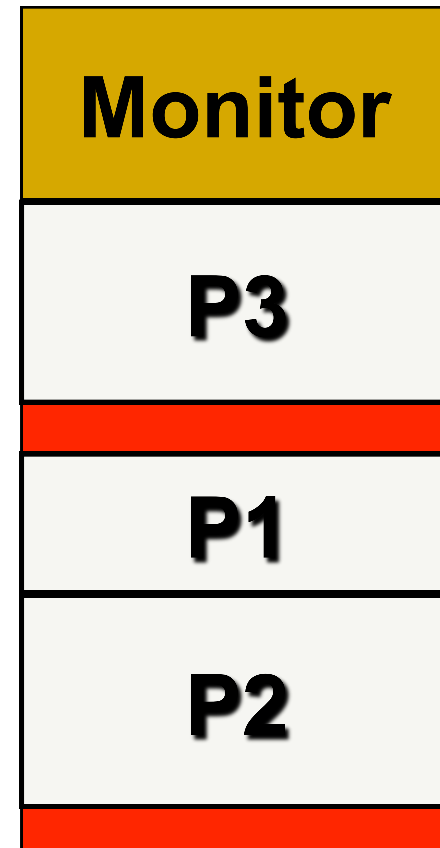


EXTERNAL FRAGMENTATION (CONT'D)

When P0 terminates

- Replaced by P3
- P3 must be smaller than P0

Start wasting memory space

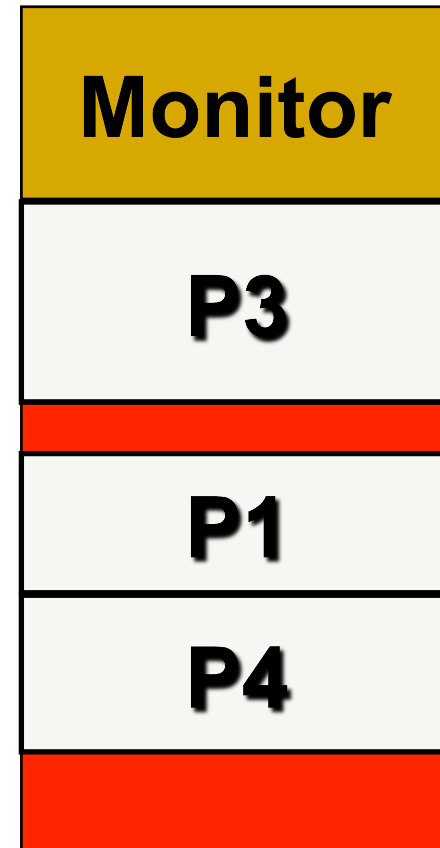


EXTERNAL FRAGMENTATION (CONT'D)

When P2 terminates

- Replaced by P4
- P4 must be smaller than P0 plus the free space

Start wasting more memory space



EXTERNAL FRAGMENTATION

Happens in all systems using multiprogramming with variable partitions

Occurs because new process must fit in the hole left by terminating process

- Very low probability that both process will have exactly the same size
- Typically the new process will be a bit smaller than the terminating process

AN ANALOGY

Replacing an old book by a new book on a bookshelf

New book must fit in the hole left by old book

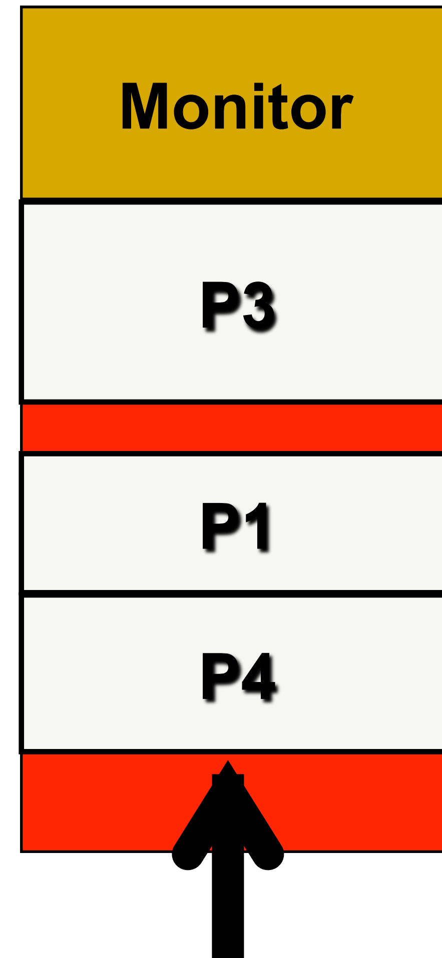
- Very low probability that both books have exactly the same width
- We will end with empty shelf space between books

Solution: push books left and right

Other situations fragmentation occurs in computer systems?

MEMORY COMPACTION

When external fragmentation becomes a problem we push processes around in order to consolidate free spaces

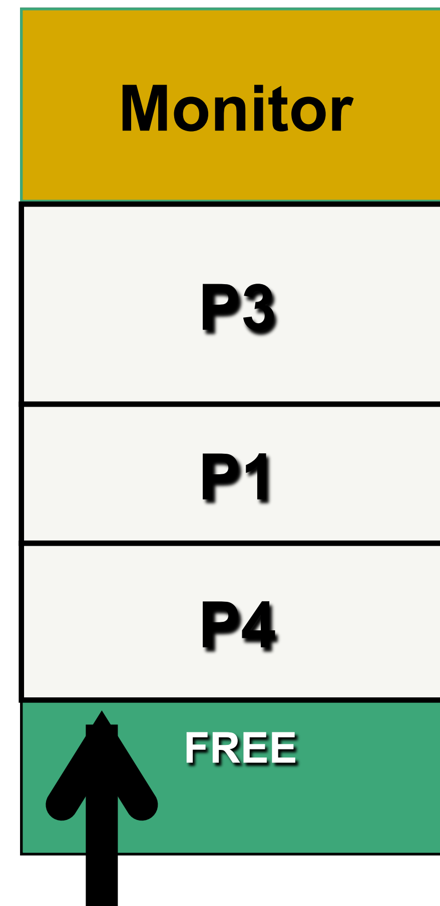


MEMORY COMPACTION

Works very well when memory sizes were small

large overhead with more processes and large memory sizes

Problematic if address binding is done at compilation or loading stage



SEGMENTATION

Non-contiguous allocation

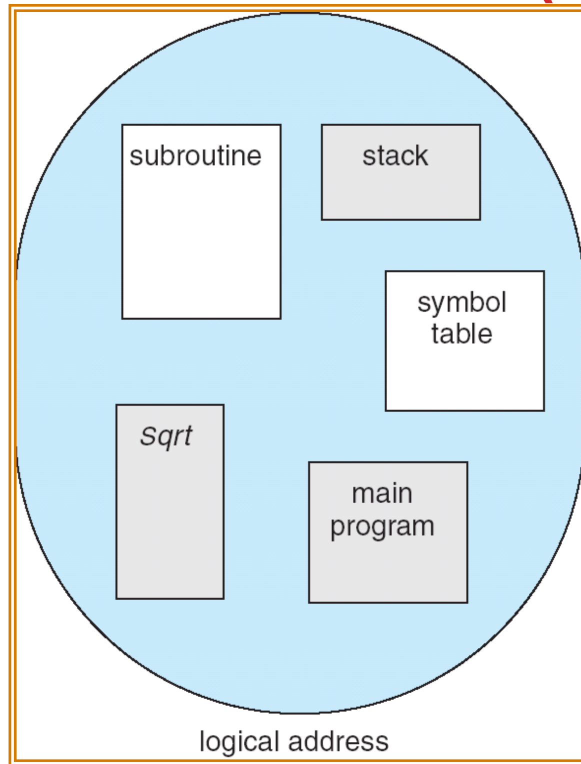
Partition physical memory into fixed-size entities

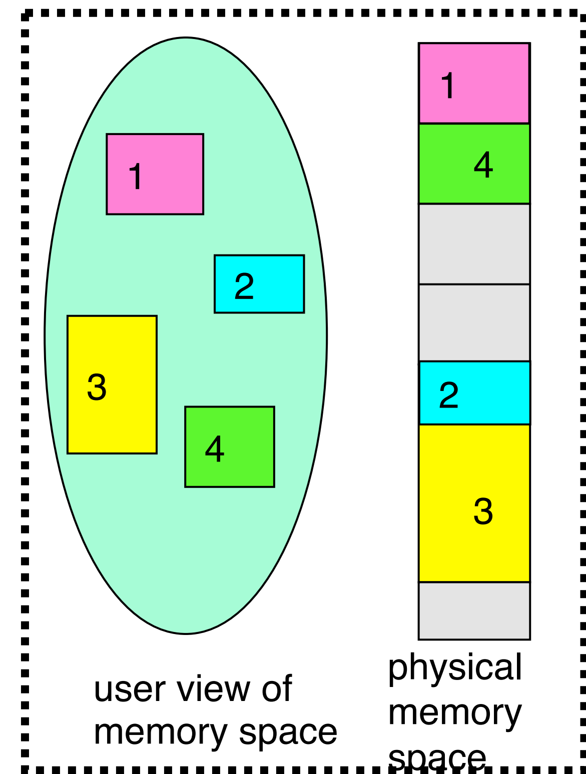
- Page frames

Allocate non-contiguous page frames to processes

Let the MMU take care of the address translation

SEGMENTATION (CONT'D)





Logical View: multiple separate segments

- Typical: Code, Data, Stack
- Others: memory sharing, etc

Each segment is given region of contiguous memory

- Has a base and limit
- Can reside anywhere in physical memory

COFF IN NACHOS

The Common Object File Format (COFF) -- a specification of a format for executable, object code, and shared library computer files used on Unix systems.

Table 9: COFF Header format

Offset	Size	Field	Description
0	2	Machine	Number identifying type of target machine
2	2	NumberOfSections	Number of sections; indicates size of the Section Table, which immediately follows the headers.
4	4	TimeDateStamp	Time and date the file was created.
8	4	PointerToSymbolTable	File offset of the COFF symbol table or 0 if none is present.
12	4	NumberOfSymbols	Number of entries in the symbol table. This data can be used in locating the string table, which immediately follows the symbol table.

COFF header

COFF table entry .text

COFF table entry .rodata

COFF table entry .data

COFF table entry .bss

Table 10: Section table entries in COFF

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded ASCII string. There is no terminating null if the string is exactly eight characters long. For longer names, this field contains a slash (/) followed by ASCII representation of a decimal number: this number is an offset into the string table. Executable images do not use a string table and do not support section names longer than eight characters. Long names in object files will be truncated if emitted to an executable file.
8	4	VirtualSize	Total size of the section when loaded into memory. If this value is greater than Size of Raw Data, the section is zero-padded. This field is valid only for executable images and should be set to 0 for object files.
12	4	VirtualAddress	For executable images this is the address of the first byte of the section, when loaded into memory, relative to the image base. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.

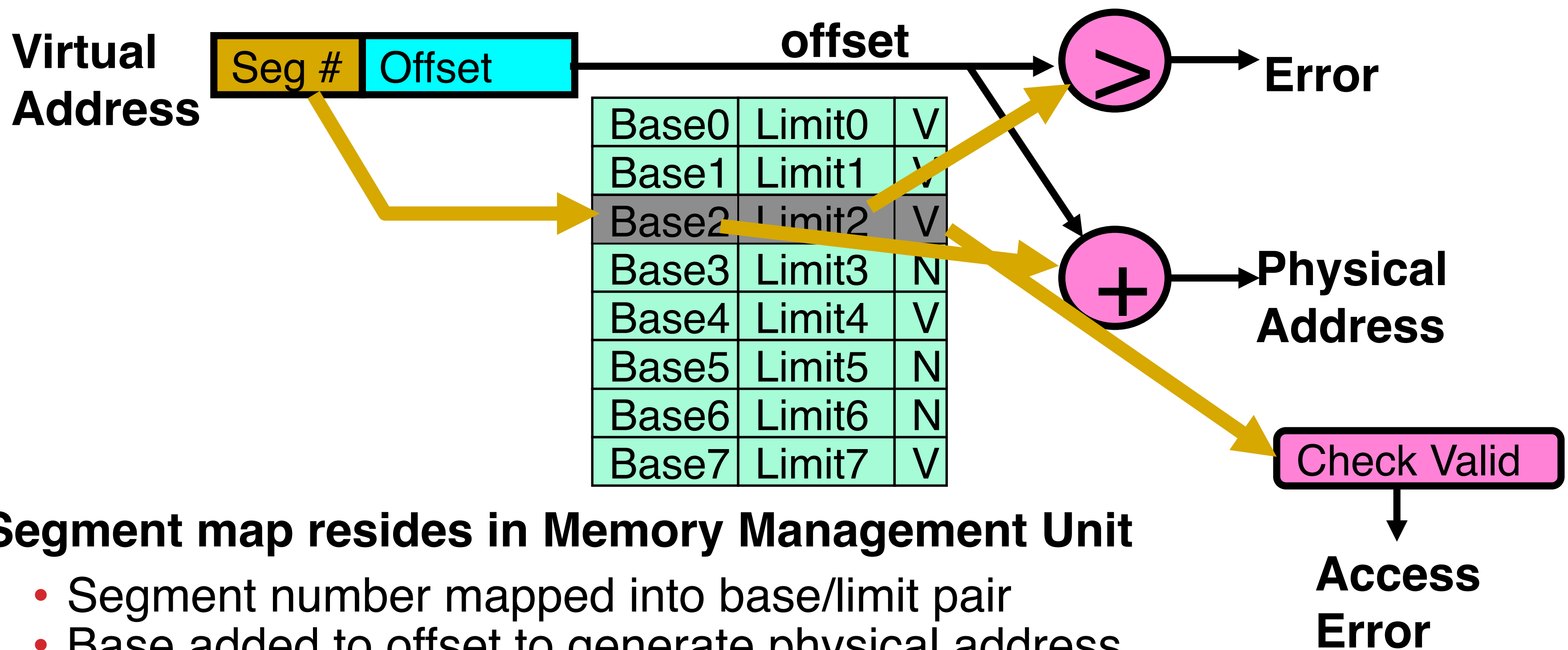
```
#define Dim     20

nachos -d ac -x matmult.coff
initializing .text section (3 pages)
…
initializing .bss section (5 pages)
```

```
#define Dim     50

nachos -d ac -x matmult.coff
initializing .text section (3 pages)
…
initializing .bss section (30 pages)
```

SEGMENTATION (CONT'D)



Segment map resides in Memory Management Unit

- Segment number mapped into base/limit pair
- Base added to offset to generate physical address
- Error check catches offset out of range

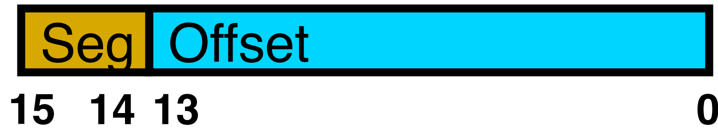
As many chunks of continguous physical memory as entries

- Segment addressed by portion of virtual address

What is "V/N" (valid / not valid)?

- Can mark segments as invalid; requires check as well

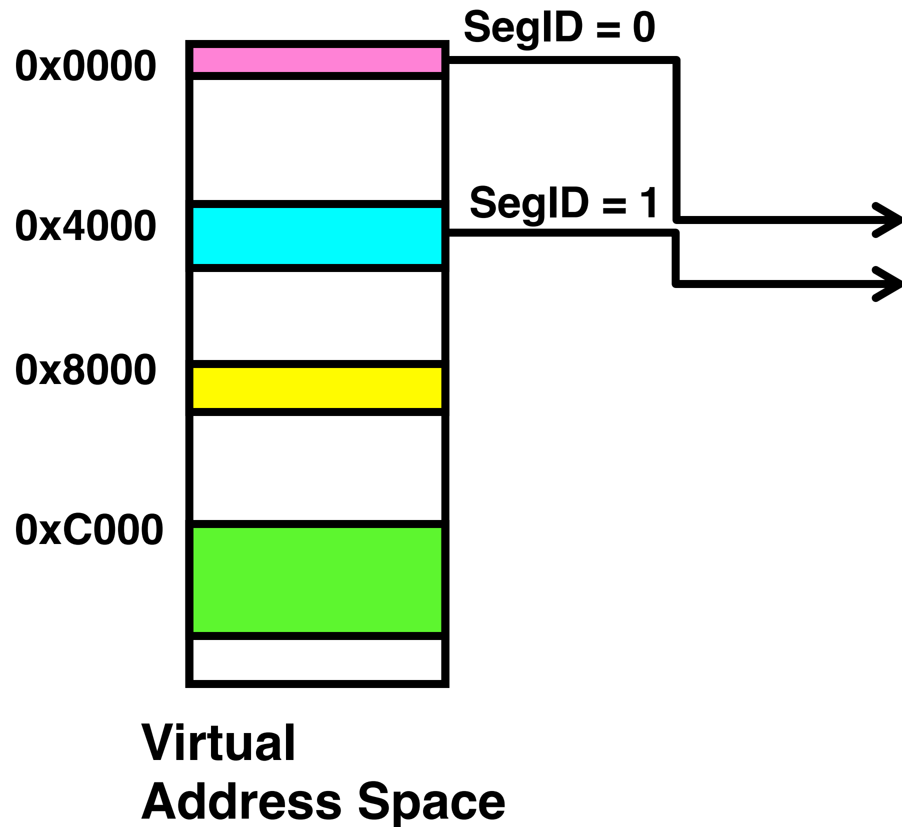
EXAMPLE: FOUR SEGMENTS (16 BIT ADDRESSES)



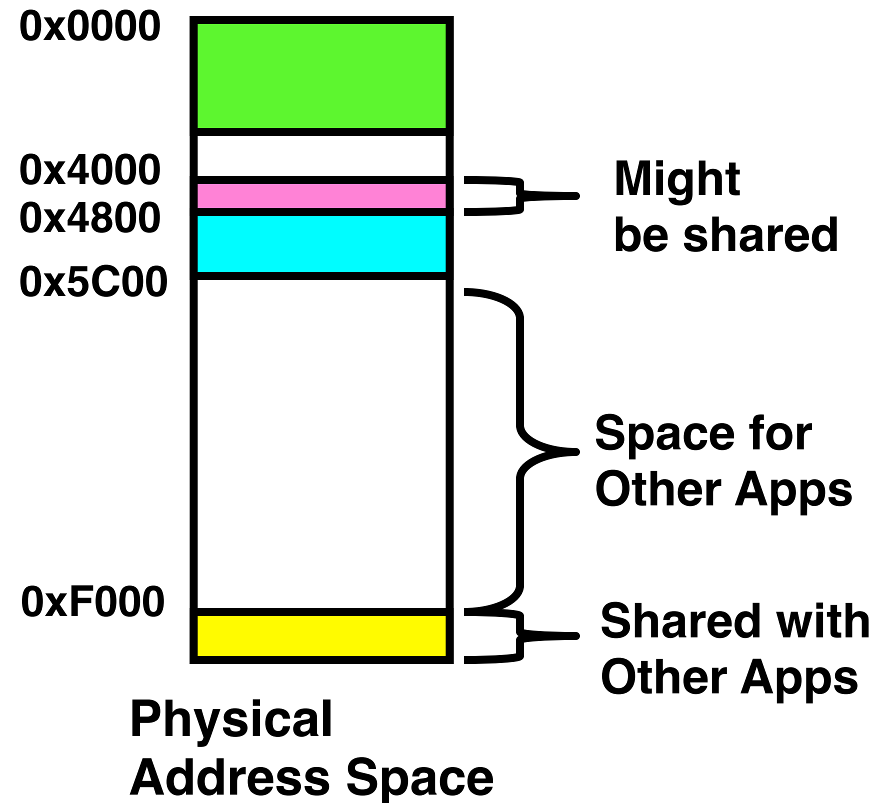
Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

segment table



Virtual Address Space



Physical Address Space

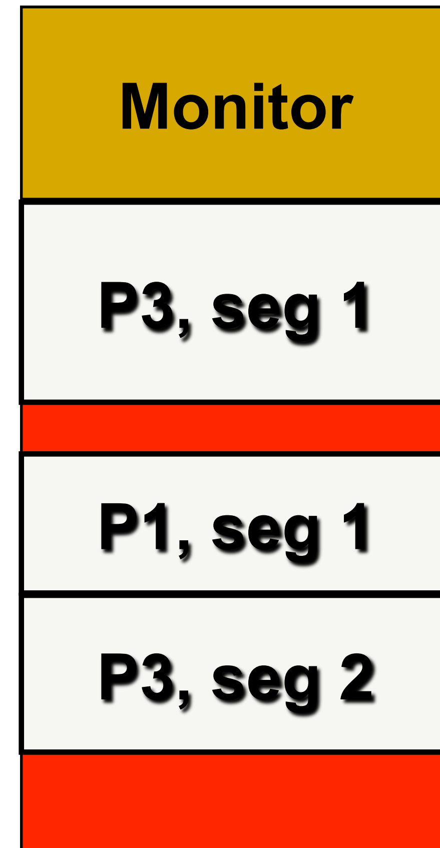
PROBLEMS WITH SEGMENTATION

How to partition

Size of segments vary

- Can still suffer from external fragmentation

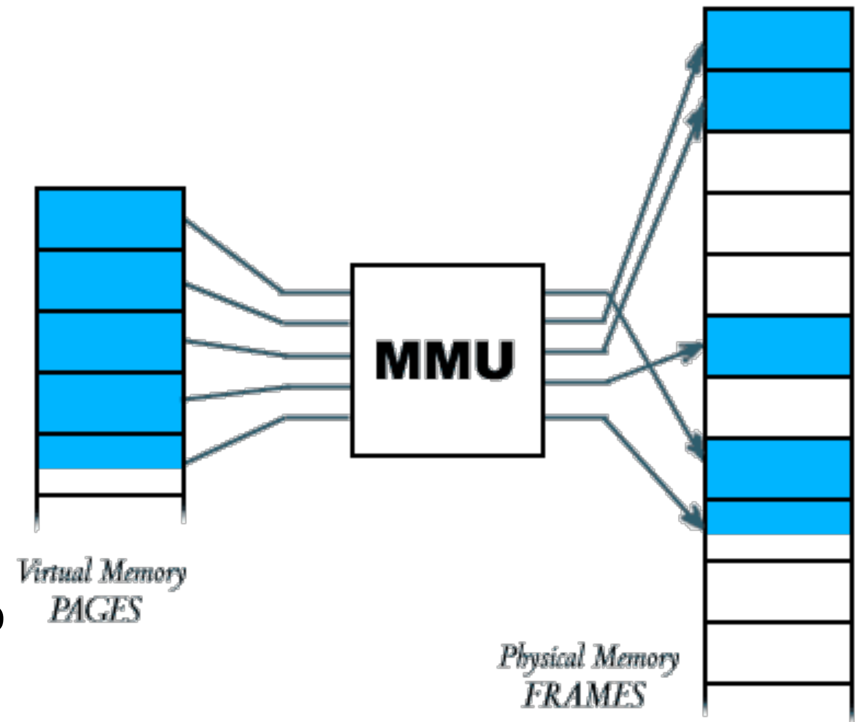
No easy for sharing



PAGING

The problem of external fragmentation in segmentation is due to the mismatch between physical and virtual memory

- holes in physical memory that no process/segment can fit



Basic idea: equal sized pages in physical and virtual memory

- How big the sizes?
- How to look up the physical page from a virtual page?
- Where to store such information?
- Internal fragmentation and validity of pages

68451 MMU used with Motorola 68010

PAGE TABLE

A page table consists of a collection of pages

- Resides in memory
- One page table per process

A page table entry (PTE) contains

- A page frame number
- Several special bits

Assuming 32-bit addresses, all fit into four bytes

THE SPECIAL BITS

Valid bit:

- 1 if page is in main memory,
- 0 otherwise

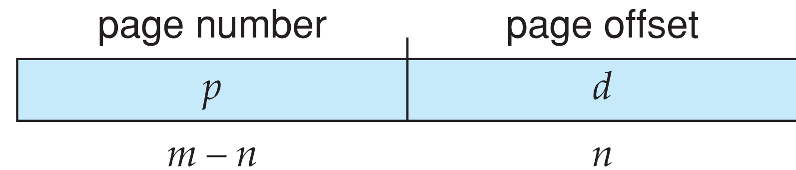
Dirty bit: 1 if page has been modified since it was brought into main memory, 0 otherwise

- A dirty page must be saved in the process swap area on disk before being expelled from main memory
- A clean page can be immediately expelled

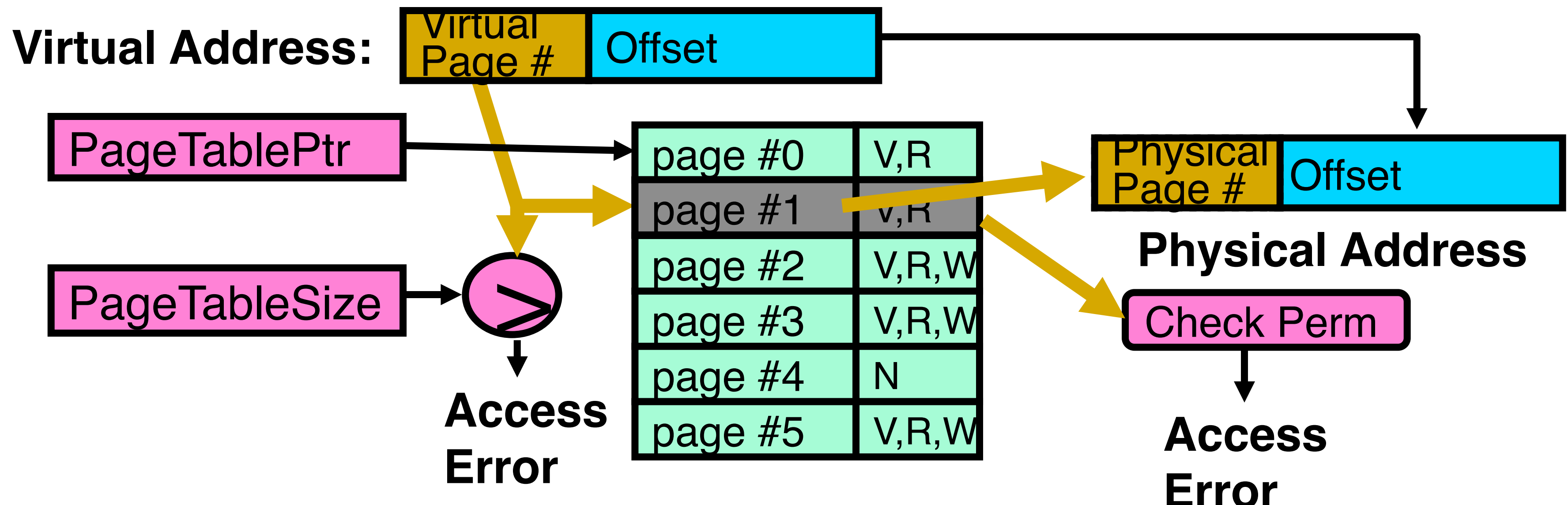
Page-referenced bit:1 if page has been recently accessed, 0 otherwise

- Often simulated in software

…



IMPLEMENTATION OF PAGING

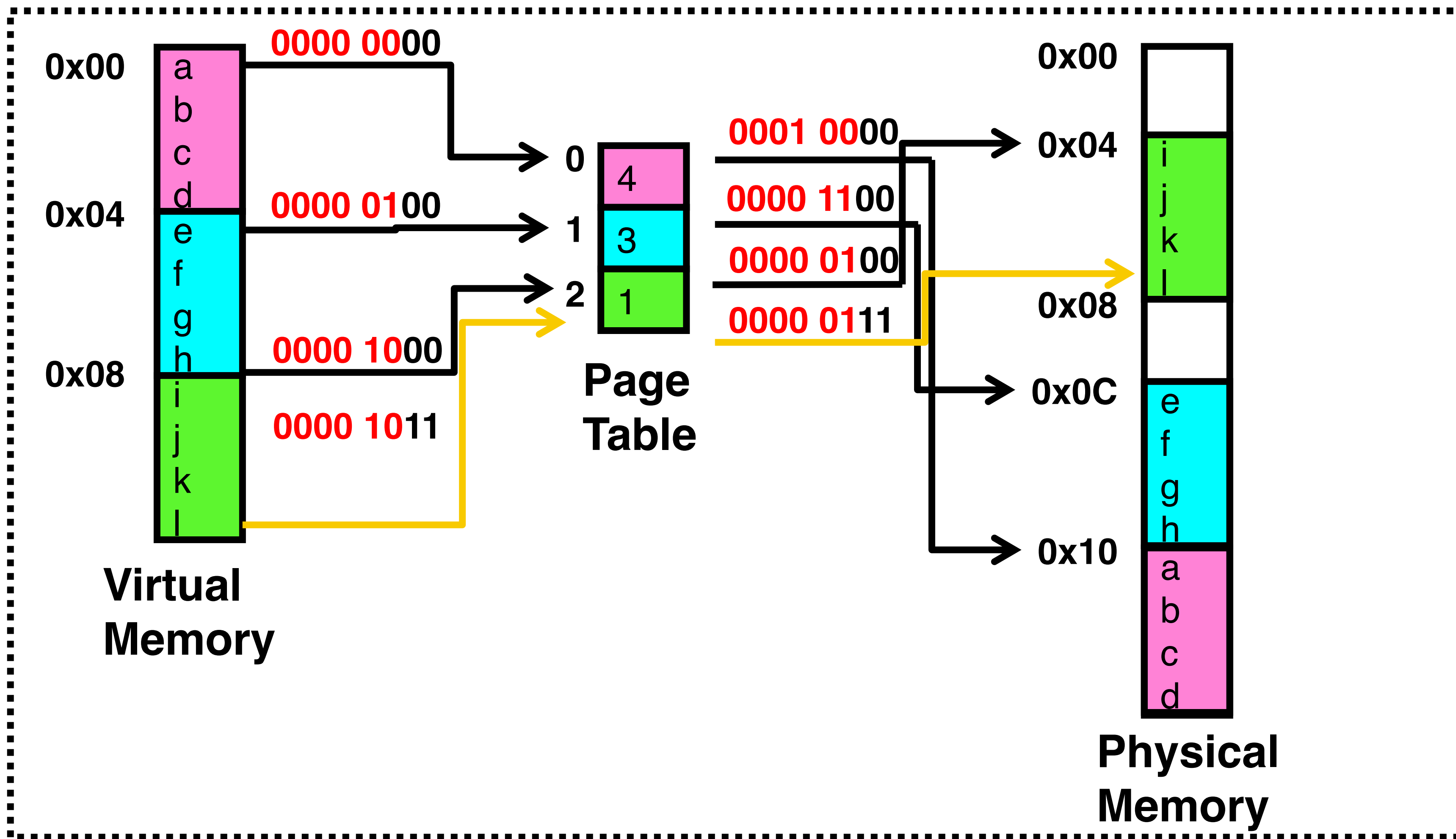


Virtual address mapping

- Offset from Virtual address copied to Physical Address
  - Example: 10 bit offset → 1024-byte pages
- Virtual page # is all remaining bits → look up the page table entry
  - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
  - Physical page # copied from table into physical address
- Check Page Table bounds and permissions

AN EXAMPLE

8-bit address, 4 byte pages



SHARED MEMORY

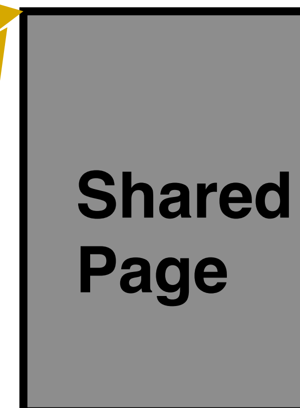
Virtual Address (Process A):

PageTablePtrA

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

PageTablePtrB

page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W



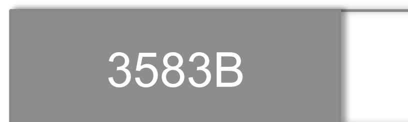
This physical page appears in address space of both processes

Virtual Address (Process B):

SELECTING THE RIGHT PAGE SIZE

Increasing the page size

- Increases the length of the offset
- Decreases the length of the page number
- Reduces the size of page tables
    - Less entries
- Increases internal fragmentation: unused space in an allocated frame



4KB

4KB seems to be a good choice

SUMMARY OF PAGING

Pros

- Simple memory allocation
- Easy to share

Con i) Internal fragmentation: may not use up the last page

Con ii) Each logical memory access ➔ two memory accesses

- Solution: use special hardware cache called translation look-aside buffer (TLB)

Con iii)

- For small page size, more page table entries needed (e.g, 1K pages, 32-bit, 4 million page table entries)
    - Page table entries needed whether in use or not
- Possible solutions:
  - Combining segmentation with paging
  - Multi-level page tables (tradeoff space complexity with time complexity)
  - Inverted page table
  - Include page table entries only when needed (which are in use?)
  - Variable page size

TRANSLATION LOOK-ASIDE BUFFER (TLB)

A high-speed cache is set up for page table entries (key,value)

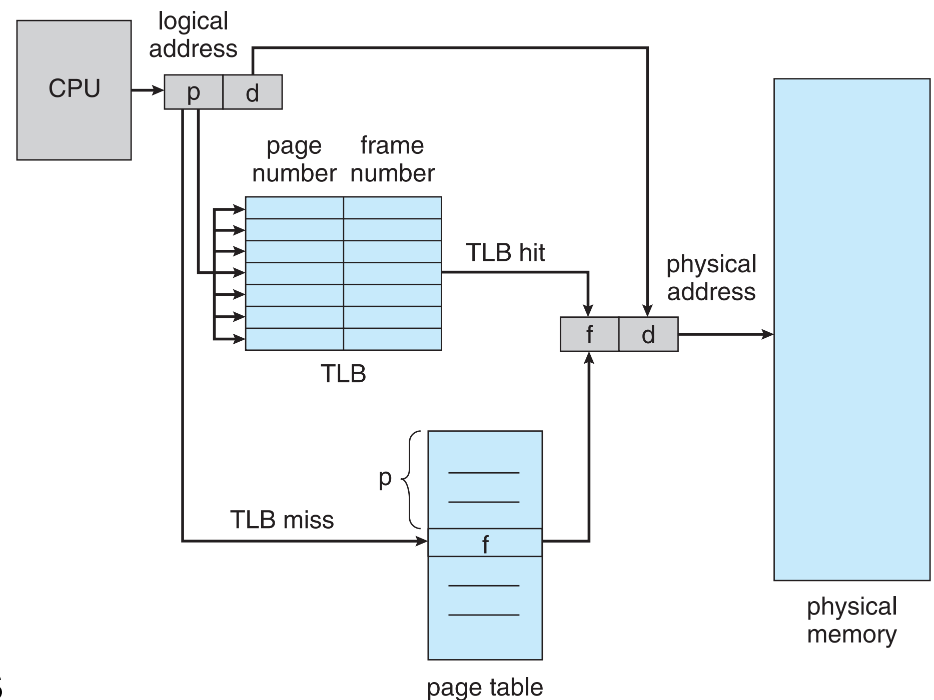
- Can concurrently check with multiple entries

Contains page table entries that have been most recently used

TLB miss requires more time

Replacement policy for TLB

- Flush TLB during context switch
- TLB miss loads new entries into the TLB

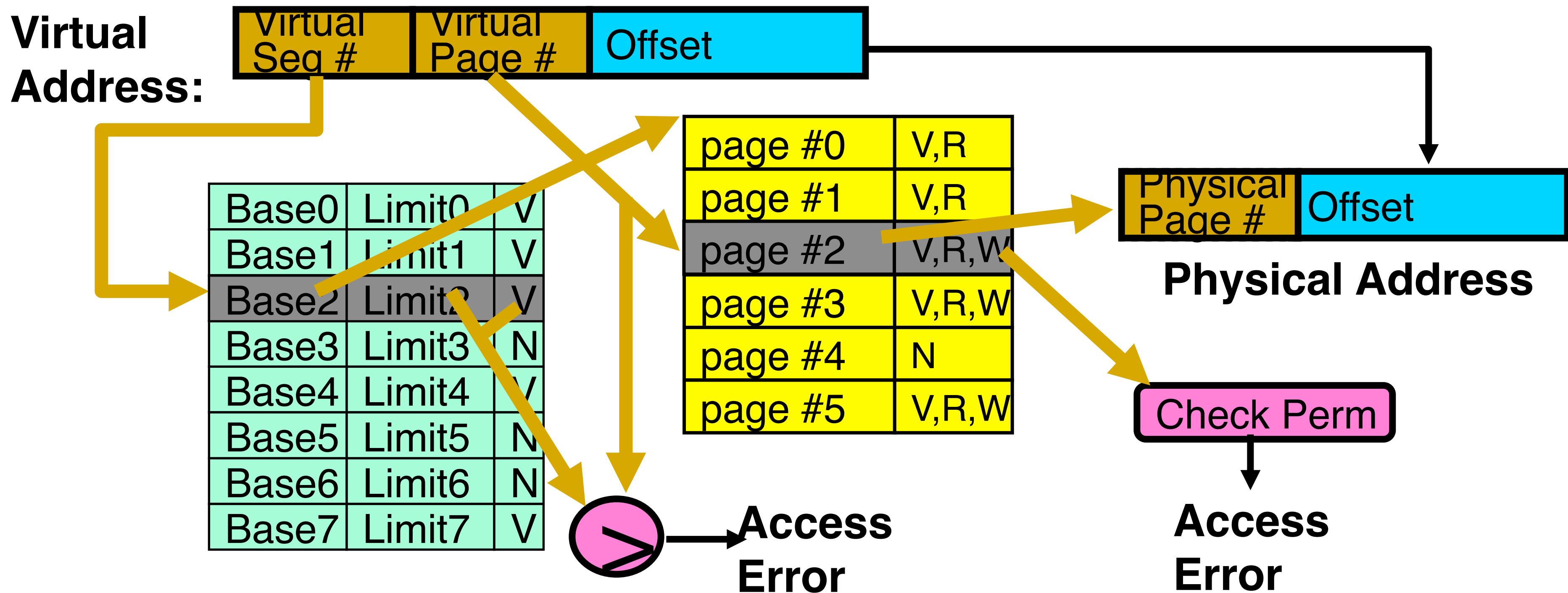


PERFORMANCE IMPLICATIONS

T_m be the main memory access time, p be the probability of TLB hit

- **Access time using TLB = $2(1-p)T_m + p*T_m$**
  - e.g., $p = 0.99 \rightarrow 1.01T_m$
- **Compared to $2T_m$ using page table only**

COMBINING PAGING WITH SEGMENTATION



Used with Intel 32-bit architecture

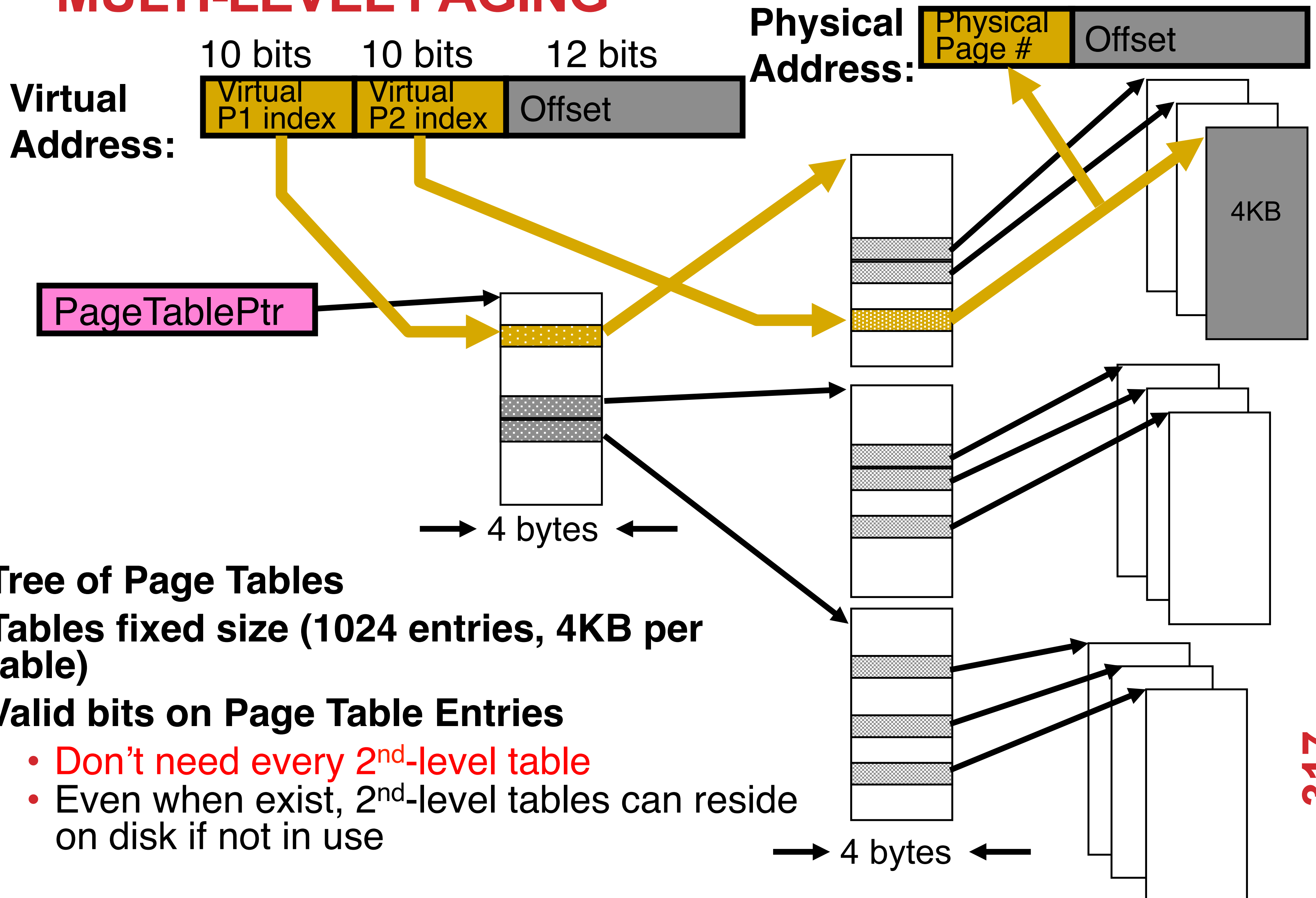
Segment map on MMU, page table in memory

EXAMPLE: SEGMENTED PAGE

- **Given a memory size of 256 = 2^8 addressable bytes,**
- **a page table indexing 8=2^3 pages,**
- **a page size of 32 bytes = 2^5, and**
- **8 = 2^3 logical segments**

1. **How many bits is a physical address?** 8 bits
2. **How many bits for the segment number, page table, and offset?** 3 3 5
3. **How many bits is a virtual address?** 11 bits
4. **How many segment table entries do we need?** 8

MULTI-LEVEL PAGING



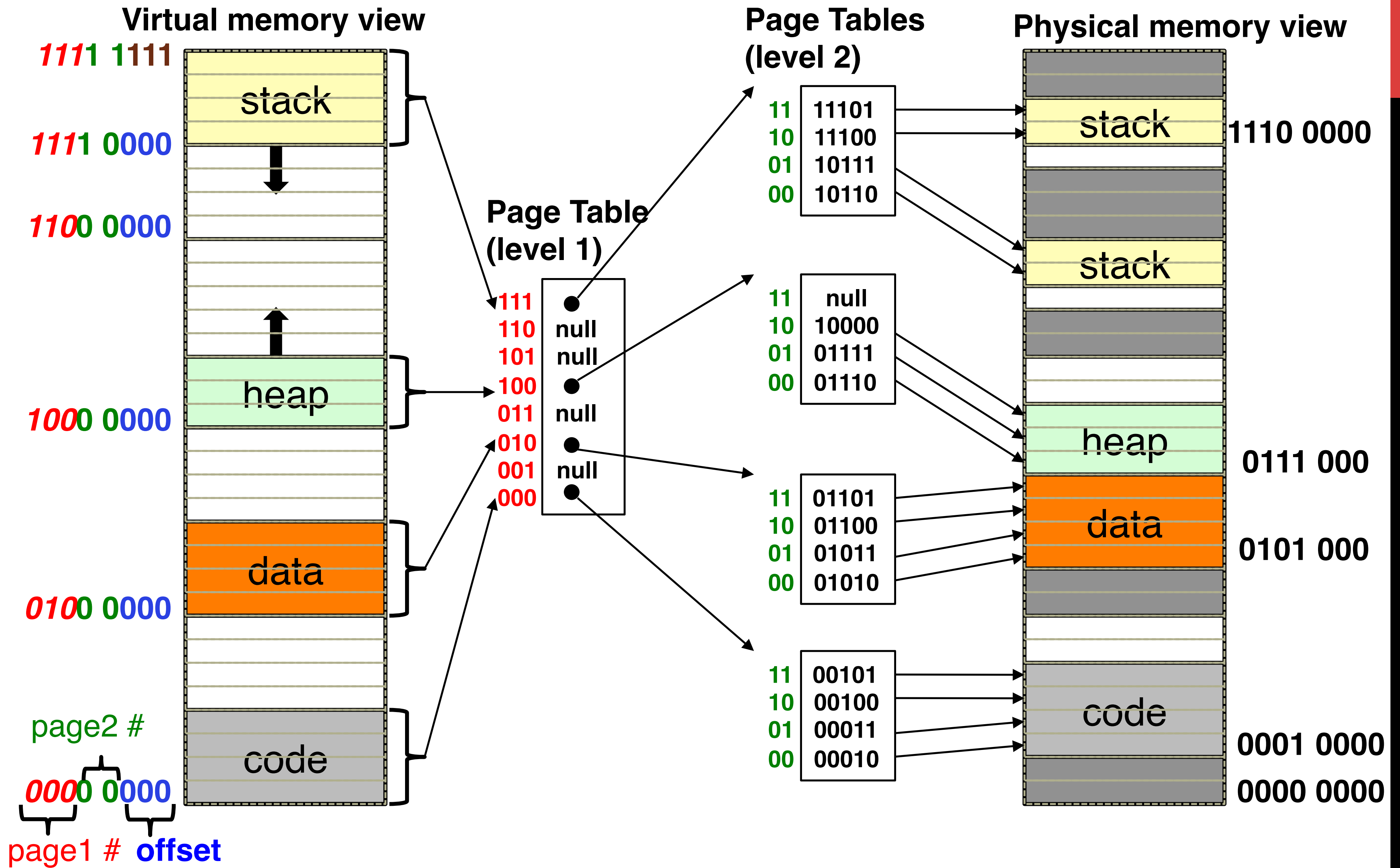
Tree of Page Tables

Tables fixed size (1024 entries, 4KB per table)

Valid bits on Page Table Entries

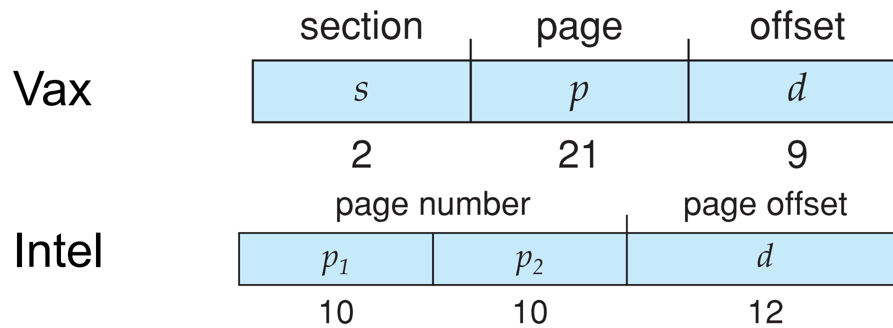
- Don't need every 2nd-level table
- Even when exist, 2nd-level tables can reside on disk if not in use

EXAMPLE (3 BITS FOR LEVEL-1, 2 BITS FOR LEVEL-2

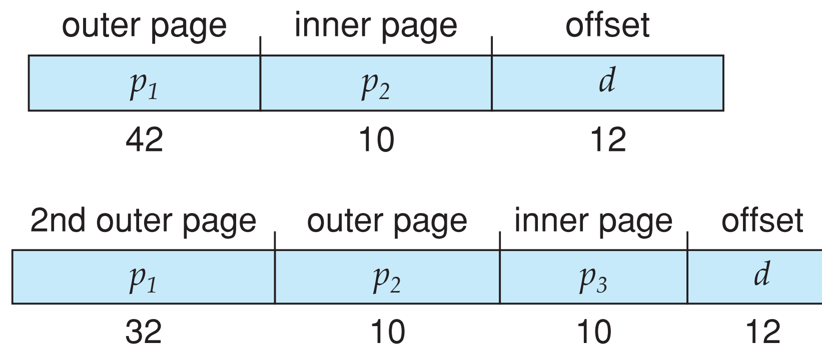


USE OF MULTI-LEVEL PAGING

32-bit machines



64-bit machine



MULTI-LEVEL TRANSLATION ANALYSIS

Pros:

- Only need to allocate as many page table entries as we need for application – size is proportional to usage
  - In other words, sparse address spaces are easy
- Easy memory allocation
- Easy Sharing
  - Share at segment or page level (need additional reference counting)

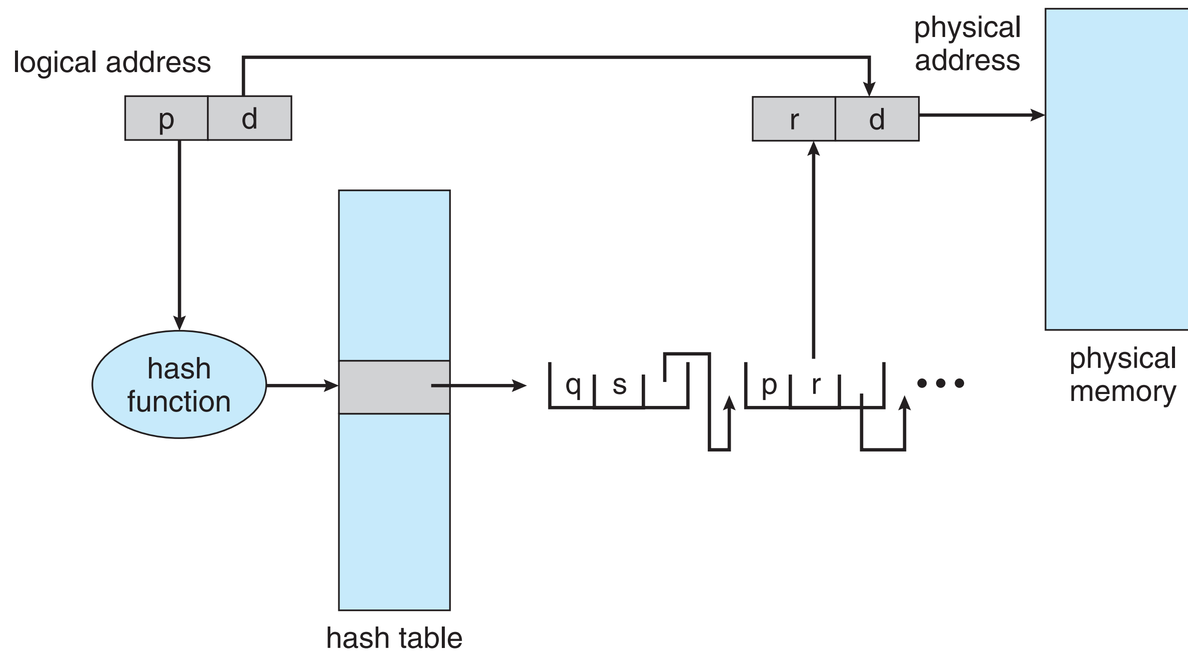
Cons:

- One pointer per page (typically 4K – 16K pages today)
- Page tables need to be contiguous
  - However, previous example keeps tables to exactly one page in size
- Two (or more, if >2 levels) lookups per reference
  - Seems very expensive!

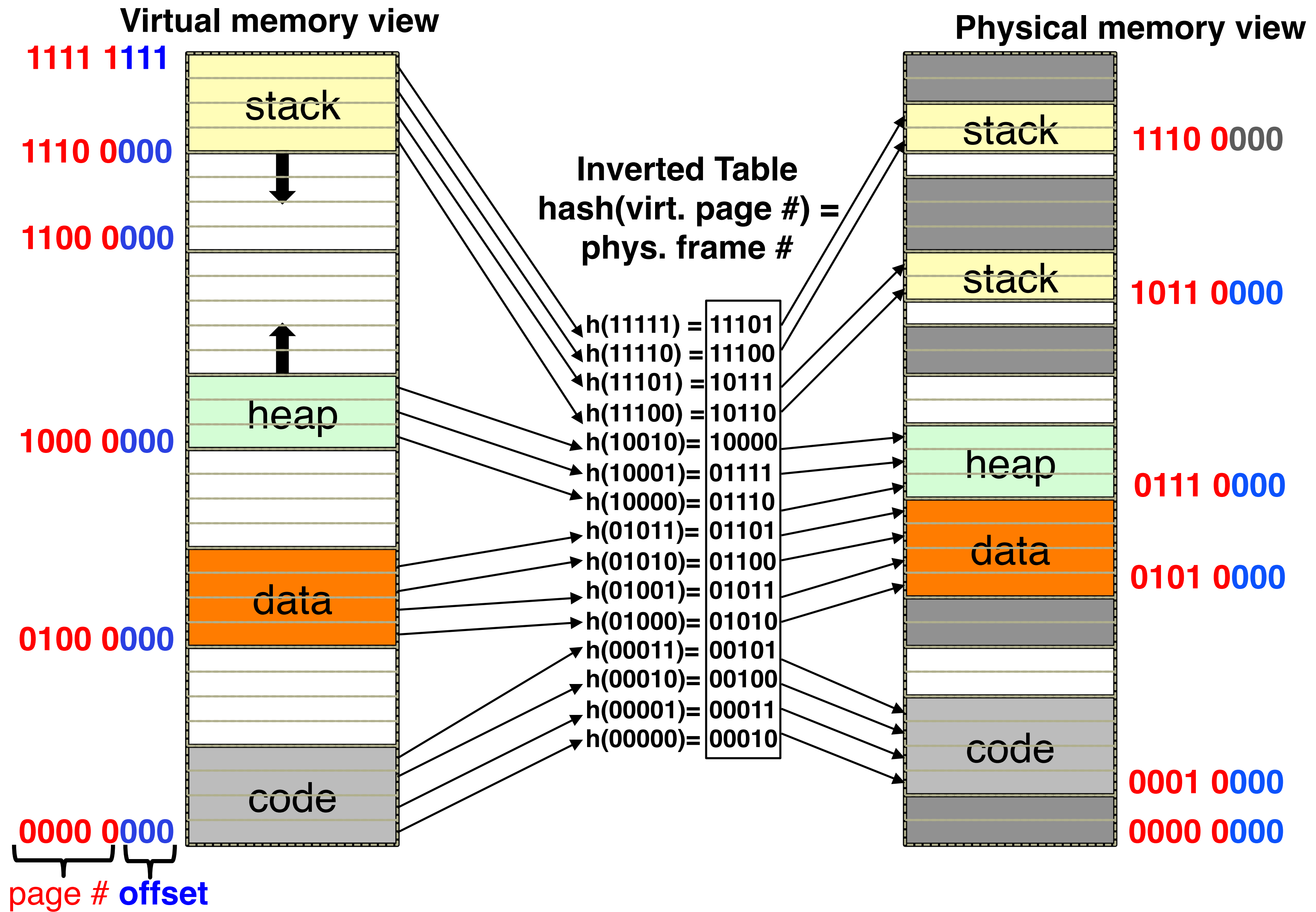
HASHED PAGE TABLES

One entry per page

fast lookup but large table



HASHED PAGE TAKE



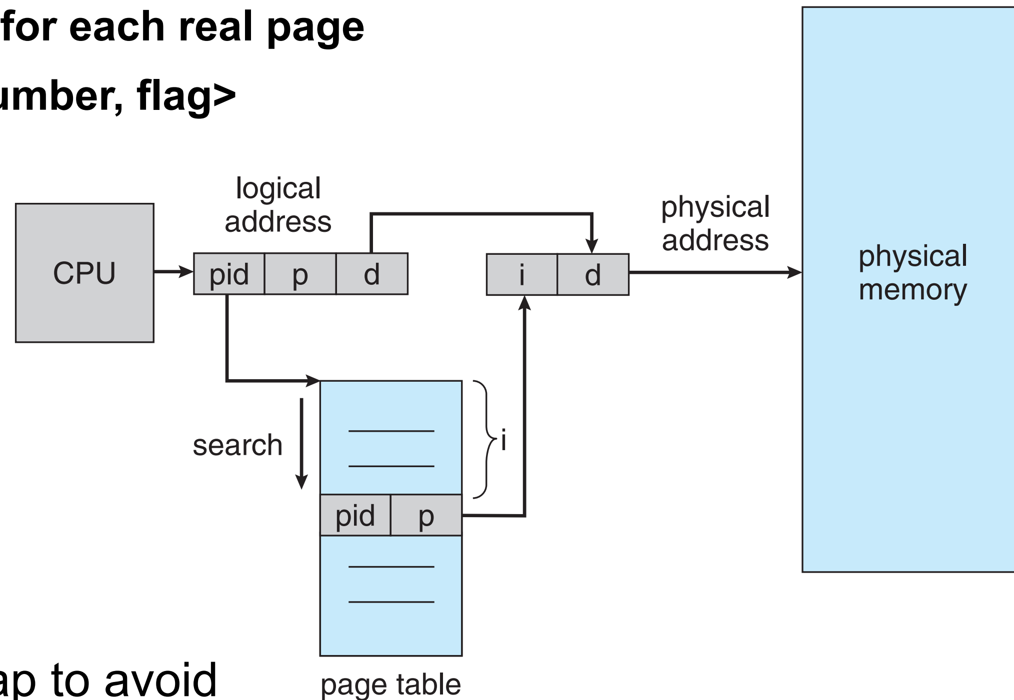
INVERTED PAGE TABLES (IPT)

Previously, with single-level paging, one page table per process

- **64-bit logical address space, 4KB per page → $2^{64-12} = 2^{52}$ entries!**
- **If use multi-level page table, 1024 entries per page → 6 levels! (why)**

In inverted page table, one entry for each real page

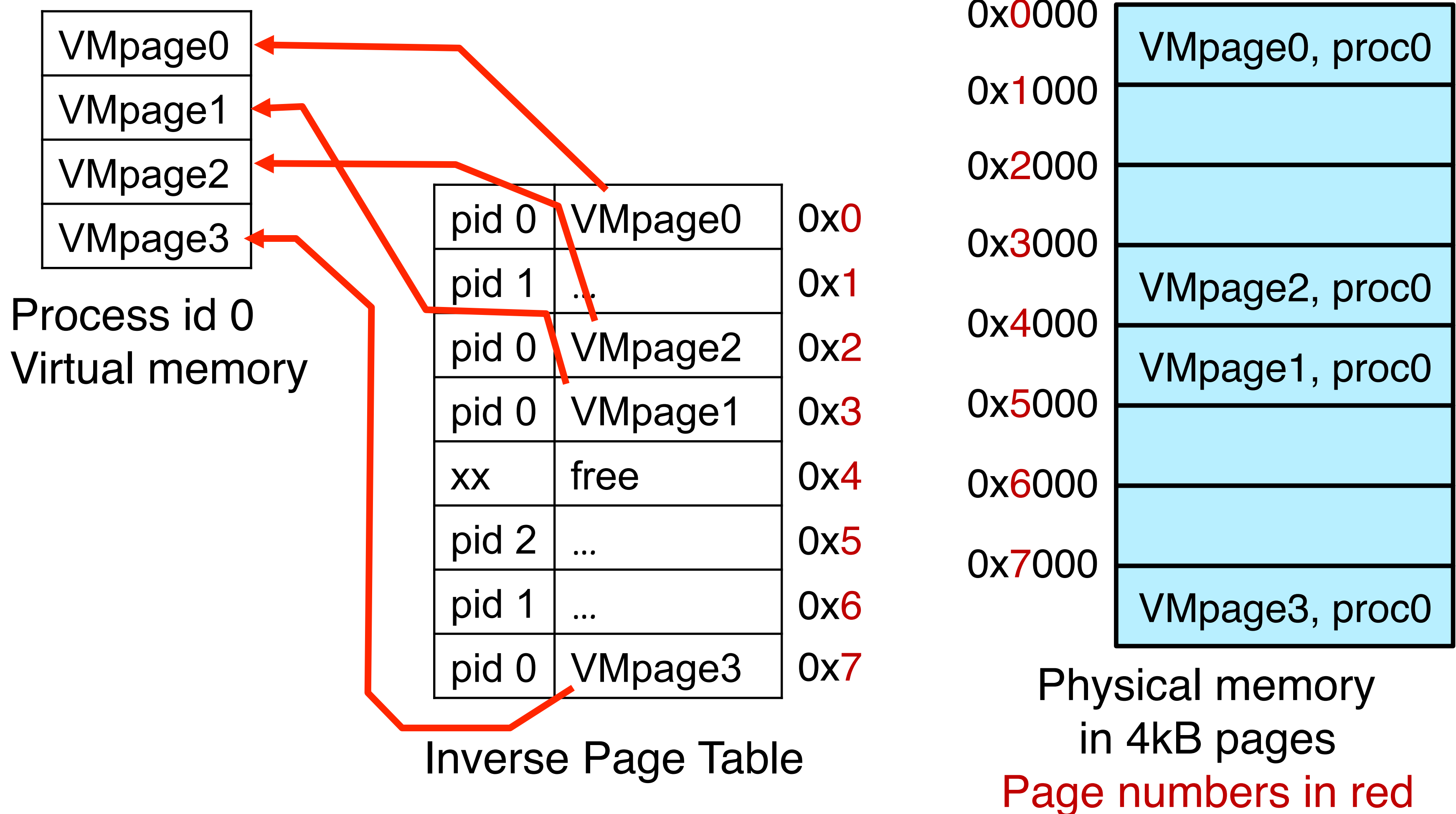
The entry contains <PID, page-number, flag>



Can be combined with a hash map to avoid linear search

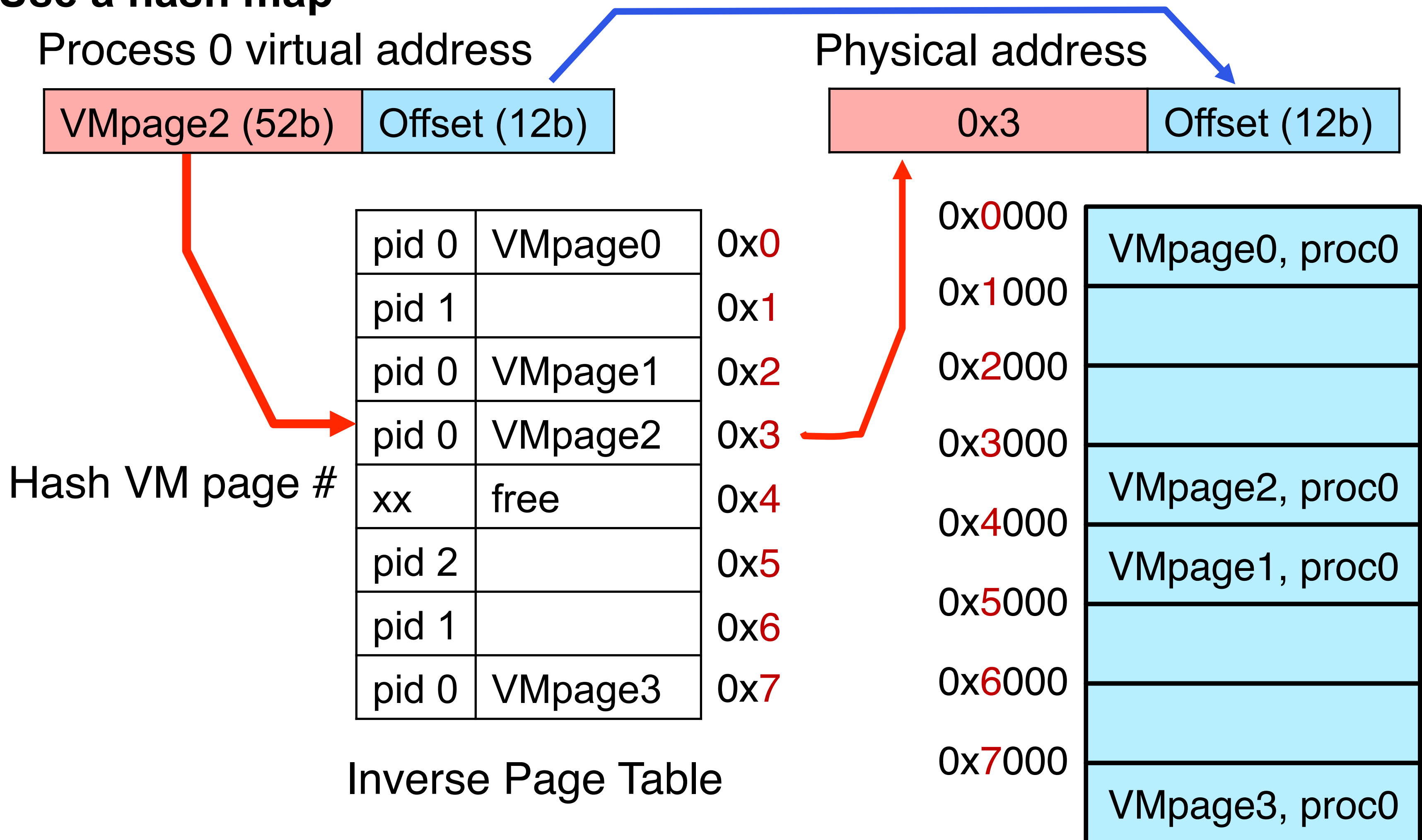
IA64 (INTEL ITANIUM ARCHITECTURE): INVERSE PAGE TABLE (IPT)

Idea: index the page table by physical pages instead of VM



IPT ADDRESS TRANSLATION

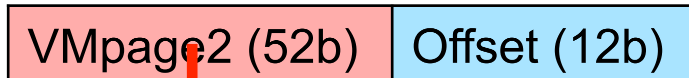
Need an associative map from VM page to IPT entry:
Use a hash map



IPT ADDRESS TRANSLATION

Note: can't share memory: only one hashed entry will match.

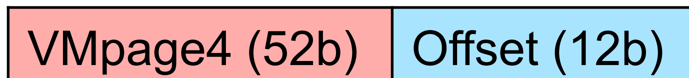
Process 0 address



pid 0	VMpage0	0x0
pid 1		0x1
pid 0	VMpage1	0x2
pid 0	VMpage2	0x3
xx	free	0x4
pid 2		0x5
pid 1		0x6
pid 0	VMpage3	0x7

Inverse Page Table

Process 1 address



IA64: INVERSE PAGE TABLE (IPT)

Pros:

- Page table size naturally linked to physical memory size.
- Only two memory accesses (most of the time).
- Shouldn't need to page out the page table.
- Hash function can be very fast if implemented in hardware.

Cons:

- Can't (easily) share pages.
- Have to manage collisions, e.g. by chaining, which adds memory accesses.

SUMMARY

<table>
<tr><th></th><th>Advantages</th><th>Disadvantages</th><th>Context switch</th></tr>
<tr><td>Segmentation</td><td>Fast context switching: Segment mapping maintained by CPU</td><td>External fragmentation</td><td>Load segment map (typically a collection of segment registers)</td></tr>
<tr><td>Paging (single-level page)</td><td>No external fragmentation, fast easy allocation</td><td>Large table size ~ virtual memory</td><td>Flush TLB
Set Page table base register (PTBR)
Store in PCB page table pointer and limit</td></tr>
<tr><td>Paged segmentation</td><td rowspan="2">Table size ~ # of pages in virtual memory, fast easy allocation</td><td rowspan="2">Multiple memory references per page access</td><td rowspan="2">(Segment registers) pointer to top level page table in PTBR</td></tr>
<tr><td>Multi-level pages</td></tr>
<tr><td>Inverted page table</td><td>Table size ~ # of pages in physical memory</td><td>Lookup time
If combined with hash table, two memory lookups</td><td></td></tr>
</table>