EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

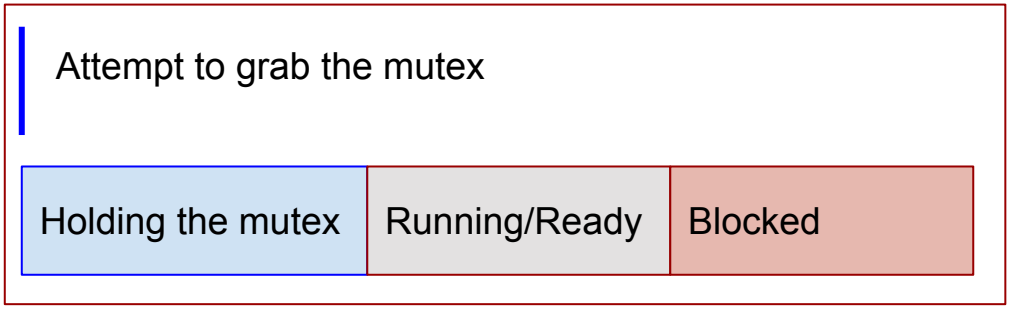
Challenge 3 discussion

- Add features to thread library to support synchronization
- Features:
  - pthread_barrier
  - pthread_mutex

Mutex

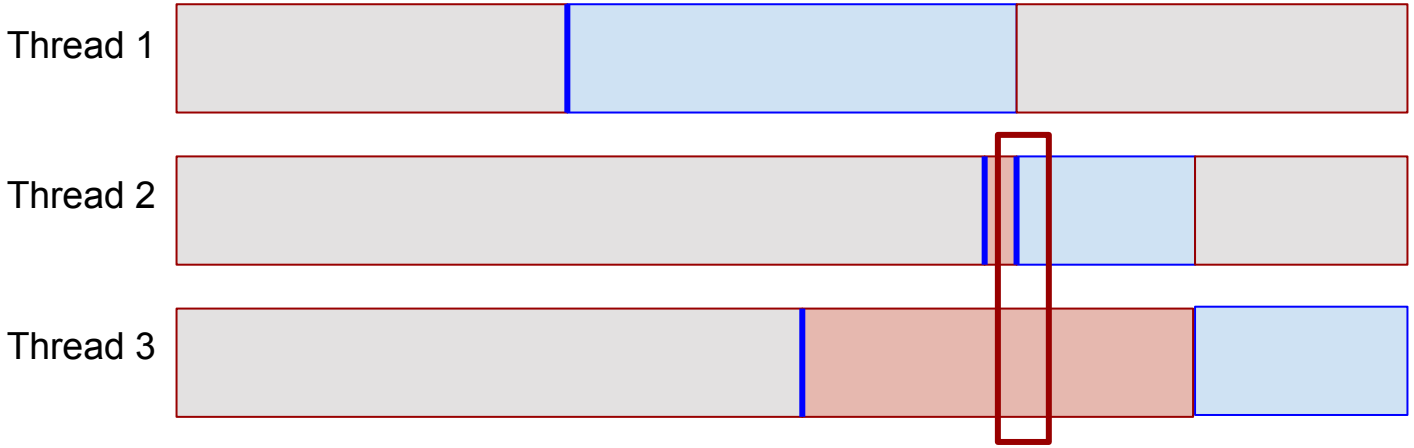
- **`Functions:`**
  - **`pthread_mutex_init`**
  - **`pthread_mutex_destroy`**
  - **`pthread_mutex_lock`**
  - **`pthread_mutex_unlock`**
- Attributes will always be NULL

Mutexes



Example: A mutex is used by multiple threads

Consider adding a BLOCKED state to your threads.



Either could get the mutex. 3 was unlucky. You can choose which thread wins.

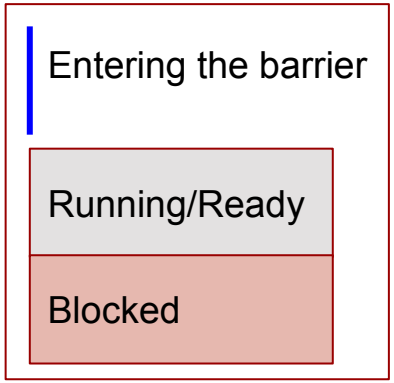
Discussion

- Does it matter who runs next?
- Are you going to schedule everyone, and have them retry, or hand off mutex to a thread?

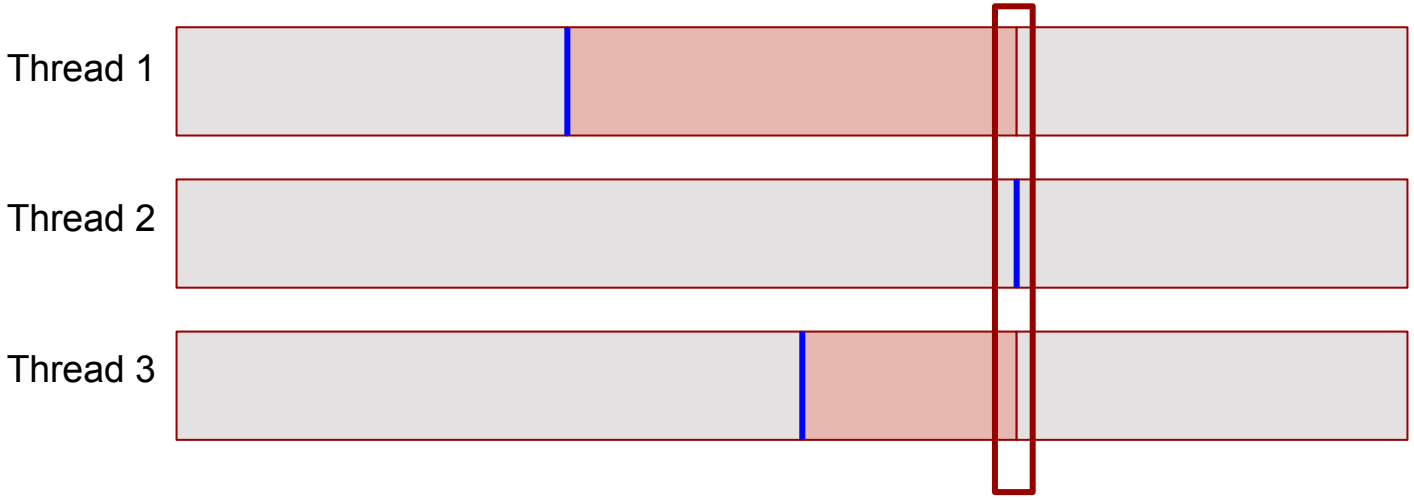
Barrier

- **`Functions:`**
  - **`pthread_barrier_init`**
  - **`pthread_barrier_destroy`**
  - **`pthread_barrier_wait`**
- Attributes will always be NULL

Barriers



Example: A barrier is initialized to a value of 3



Discussion

- What do threads get after barrier completes?
- Do they all get the same return code?
- What is the state of the barrier after it is complete?
- Hint, read the man pages....

Challenges

- You are going to be manipulating data structures in, e.g., your implementation of pthread_wait ... what could go wrong?
- What if a signal happens while you are doing that?
- How can you manipulate these data structures atomically?
- Internal lock/unlock that uses sigprocmask to mask/unmask alarm.

Other challenges

- Need to keep track of the list of threads blocked on mutex or waiting for barrier
- Look at the include file at the data structures, figure out what you want to hook your state to... :
  - `pthread_barrier_t, pthread_mutex_t`

How are you going to test?

- How will you test that a mutex is held after you call lock?
- How will you test that a thread has been added to barrier?
- Remember: `extern enum pstate_e get_status_phil(int p);`
- You can't modify the pthread prototypes, but you can add your own functions..
- Advise look at tests from
  - ec440-collaboration/lecture_demos/snippets/sync/

Review

Memory Abstractions

- The total set of addresses a program can possibly refer to is called its *address space*
- Trouble awaits if all programs have the same address space - i.e., physical memory mode
  - No protection from each other – errors in one program can cause damage to others
  - Programs must be written cooperatively, knowing about where others are located in memory
- Virtual memory: an abstraction, so that each process has a *private address space*: make 0x1234 in Program A different from 0x1234 in Program B

Memory Management

- The portion of the OS that allocates, frees, and tracks the usage of RAM is the *memory manager*
- Fundamental jobs of the memory manager:
  - Managing virtual memory abstraction of each process
  - Managing all of the physical memory on the system.
- Segmentation is a simple approach to provide a virtual address space for each process.
  - To support Segmentation the CPU has *base* and *limit* registers
  - Each time a memory address is referenced, the CPU transparently adds the *base* to it and verifies that *base* + address ≤ *limit*
  - A linear virtual address space translates to directly to a linear physical address space

Review from last lecture(cont)

- The entire process must be in memory when using segmentation.
- The virtual address space must be smaller than physical memory when using segmentation.
- whole processes must be swapped out/in using segmentation.
- Segmentation causes fragmentation of physical memory.
- We use compaction to coalesce free regions of physical memory.
- We keep track of free physical memory using bitmaps or linked lists.

Memory Management continued

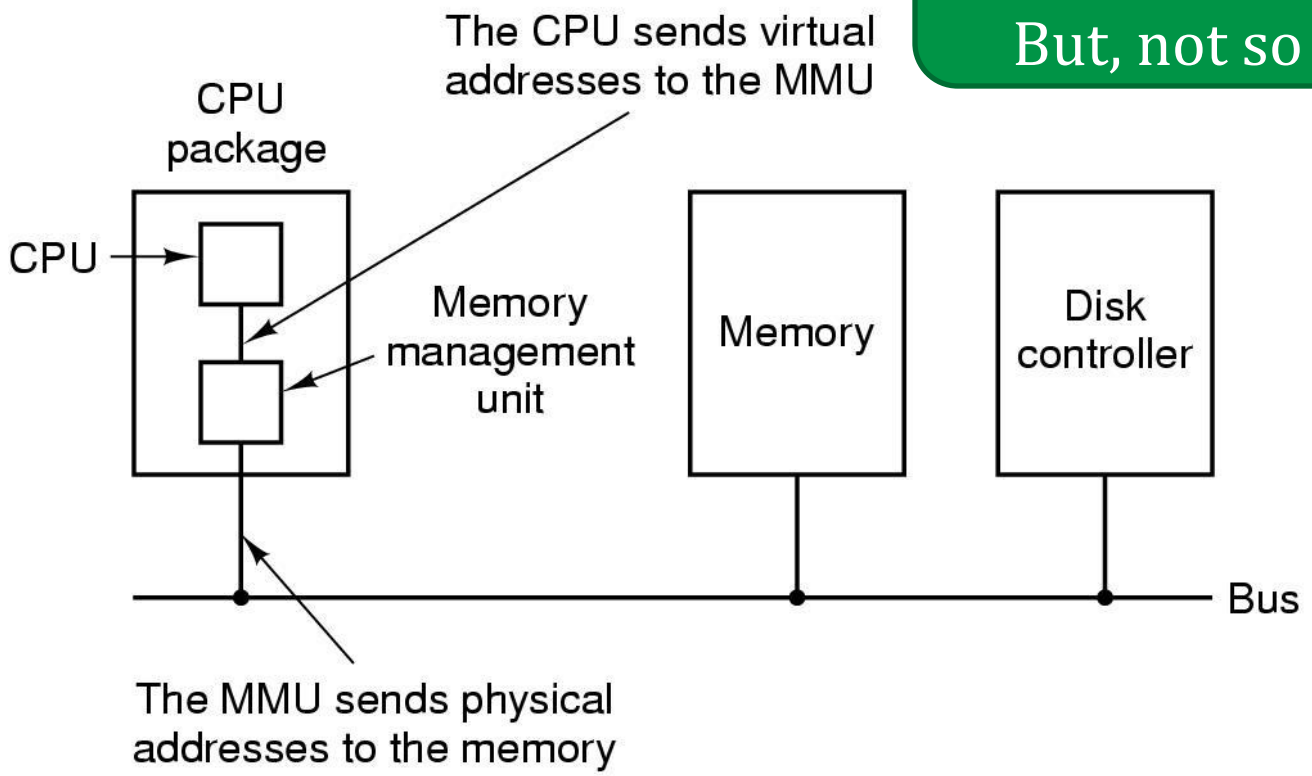
Paging

- In most situations, *paged virtual addressing* technique is used rather than just segmentation
- Maps contiguous virtual addresses to a discontiguous set of physical pages
- OS (with help from the hardware) manages the mapping between pages and page frames
- Each memory access no longer refers directly to physical memory, but instead is *mapped* to some actual physical address

Memory Management Unit (MMU)

Automatically performs the mapping from virtual addresses to physical addresses

Done!
See you next time.
But, not so quick!

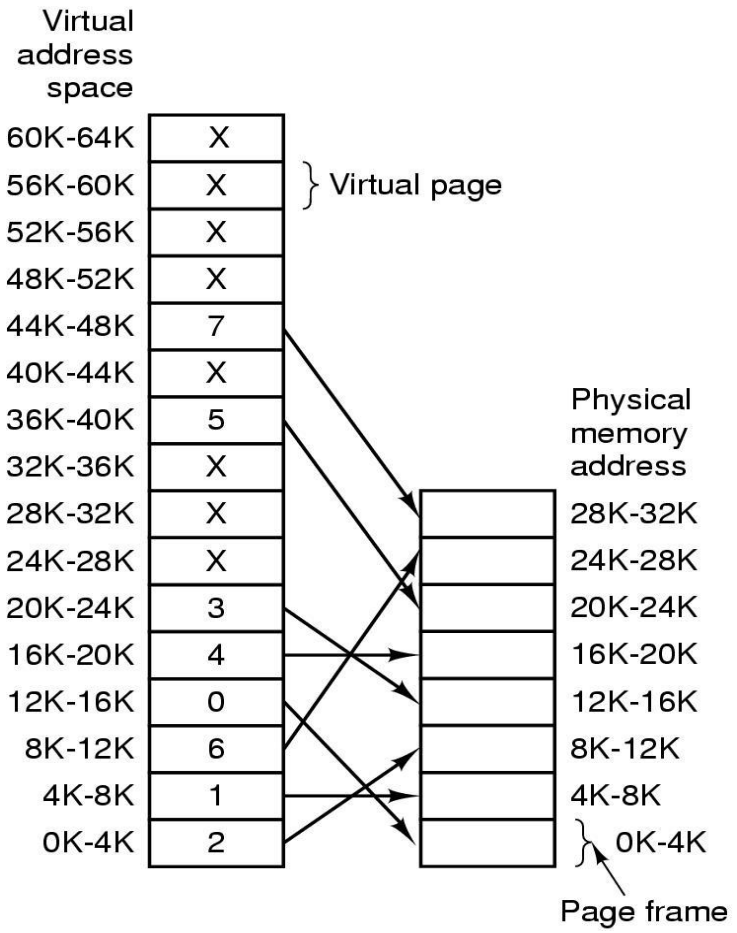


Paging

- Both virtual and physical memory is broken up into fixed-sized units (commonly, $0x1000_{16}$ == $4{,}096_{10}$ bytes)
  - Virtual units called *pages*
  - Physical units called *page frames*
- Page sizes can vary though:
  - 32-bit x86 supports 4KB and 4MB pages
  - 64-bit x86_64 supports 4KB, 2MB, and 1GB pages

Mapping Pages to Page Frames

- Virtual memory: 64KB
- Physical memory: 32KB
- Page size: 4KB
- # Virtual memory pages: 16
- # Page Frames: 8



Outline

- Handling miss in the translation
- Information in page table
- Page table organizations:
  - Single-level page table
  - Multi-level page tables
  - Inverted page tables
- Cache the translations

Page Faults

- What happens if we try to access a page that is not mapped?
- The MMU notices, and we raise a CPU exception called a *page fault*
- Control is passed to the OS (via interrupt) to decide what to do
  - Kill the process if the virtual address is not allowed(segmentation fault)
  - Find some physical page to map to it

Programs Bigger than Memory

- Note that this gives us a way to have programs that don't all fit into memory at once
- We can just map in the parts of the program we're using right now
- If we reference a page that isn't mapped we incur a page fault which locates the appropriate page and maps it.
- If all physical memory is in use we can free another page by ``*paging*'' it to disk.
  - way cheaper than swapping whole process

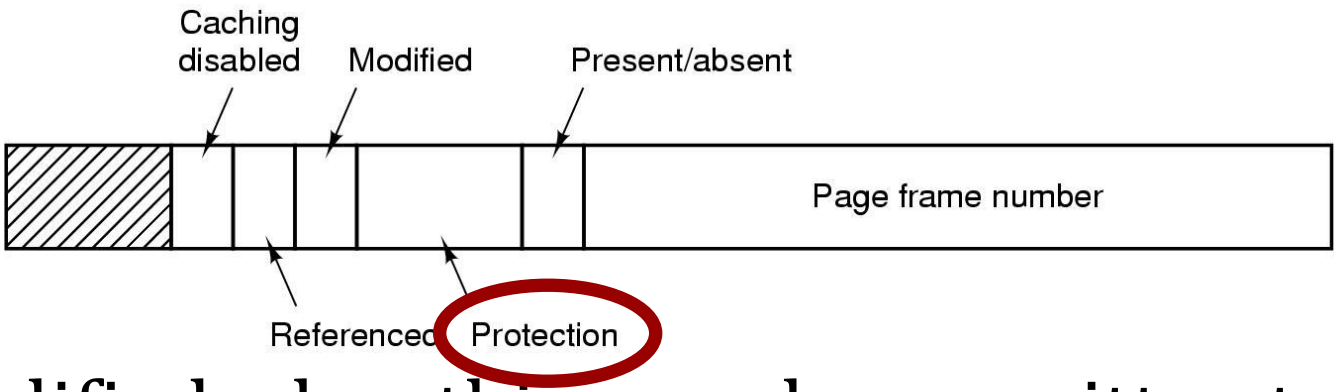
Types of Page Faults

- **Minor page fault** – can be serviced by just creating the right mapping
- **Major page fault** – must load in a page from disk to service
- **Segmentation fault** – invalid address accessed; can't service so we usually just kill the program (dynamic stack expansion uses this but just extend the stack section, we don't kill the process)

Outline

- Handling miss in the translation
- Information in page table
- Page table organizations:
  - Single-level page table
  - Multi-level page tables
  - Inverted page tables
- Cache the translations

Page Table Entries (PTEs)



- Modified – has this page been written to?
  - If so, we will need to write to disk before evicting
- Referenced – has anyone used this page?
- Caching disabled – used if physical page is used for device I/O

Protection

- Because the OS can give processes different virtual address spaces, we have already solved the problem of *isolation*
- But we may want to protect process from themselves in some cases:
  - Detect programmer errors before they do damage
  - Prevent attacks that exploit software vulnerabilities

Protection

Simplest protection is to mark pages as read-only or read/write

- Now, if someone attempts to modify read-only code or data, a page fault will occur

Some processors (in x86-land, starting with the AMD64 in 2003) have a bit to prevent code from being executed on a certain page

- This has been called variously the NX bit, the XD bit, Data Execution Prevention (DEP)
- The idea is to prevent buffer overflows from being exploitable – the attacker won't be able to run his own code because it will be in a data region

Locality of Reference

- Multilevel tables can hold many pages but they still require multiple index lookups for each memory access
  - If we have to do 2 table lookups for every memory access, we've just made memory 3x slower
- Most programs use a subset of their memory pages (loops, sequential executions, updates to the same data structures, etc.) and the set changes slowly
  - *Locality of reference*
  - *Working set*
- The CPU keeps a small cache of mappings that it can translate directly without consulting the page tables

Outline

- Handling miss in the translation
- Information in page table
- Page table organizations:
  - Single-level page table
  - Multi-level page tables
  - Inverted page tables
- Cache the translations

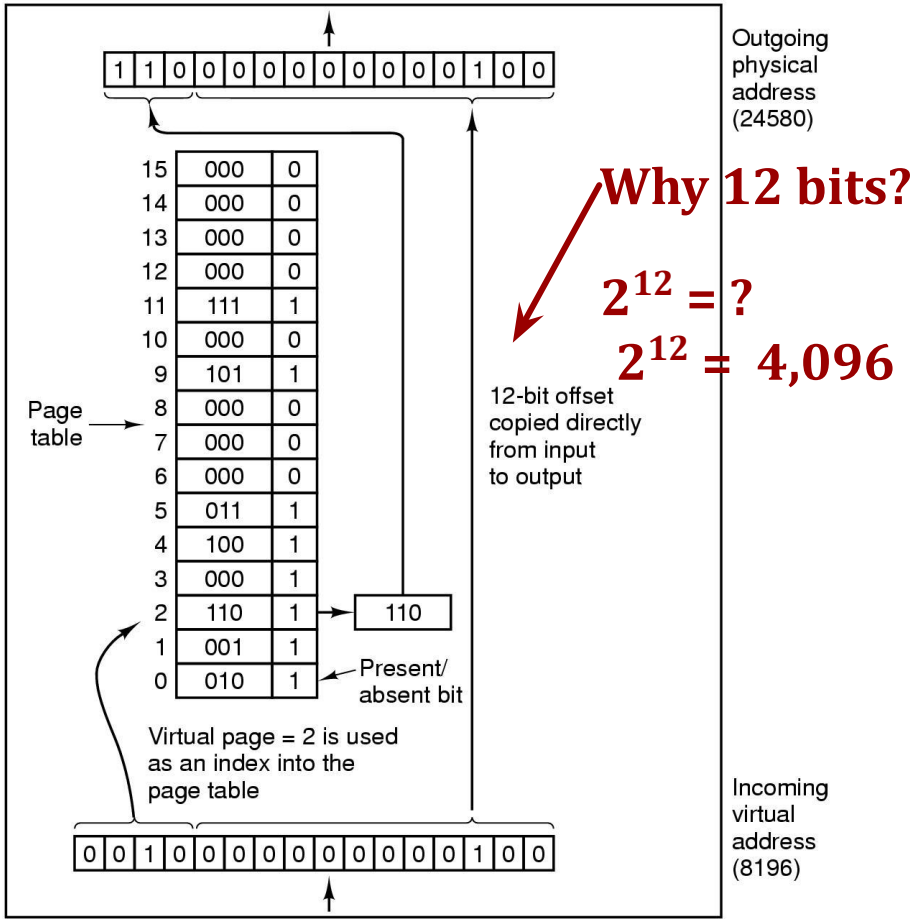
Page Tables

- The MMU has to maintain information about the virtual -> physical mapping
- In the simplest case, this could be a simple array that stores the physical page number for each virtual page number
- The virtual address would then be split into two parts: an index into the mapping table, and then the offset within the page
- As we said in the segmentation lecture, a page table entry is functionally equivalent to a segment register

You just have billions
rather than 6

Memory Management Unit

- Addresses are split into a page number and an offset
- Page numbers are used to look up a table in the MMU with as many entries as the number of virtual pages
- Each entry in the table contains a bit that states if the virtual page is actually mapped to a physical one
- If it is so, the entry contains the number of physical page used
- If not, a *page fault* is generated and the OS has to deal with it



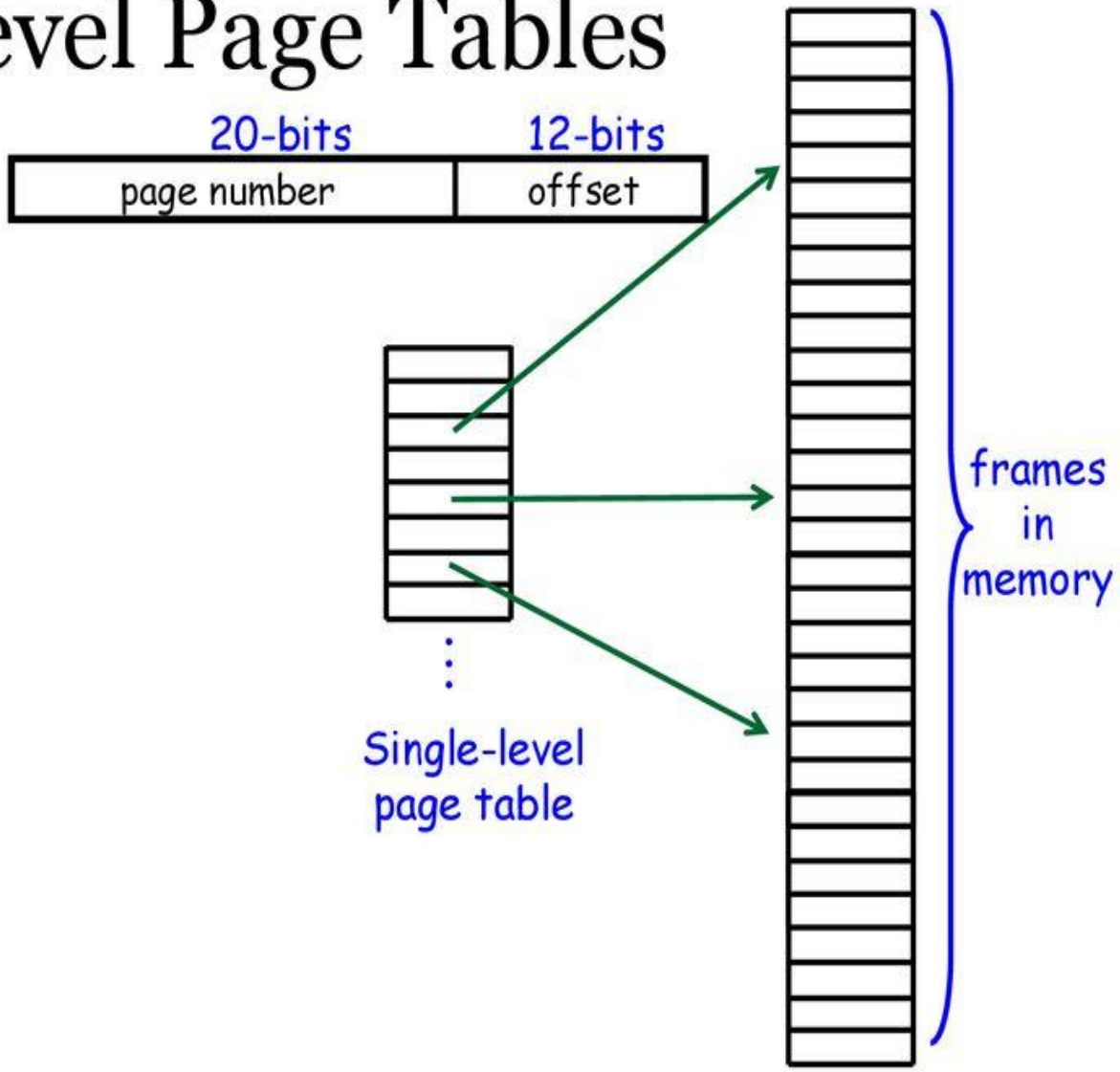
Page Tables

- Page tables contain one entry for each virtual page
- If virtual address space is big (e.g., 32 bit and 64 bit addresses) the table can become of very large and unmanageable size
- Solution: instead of keeping them in the MMU move them to main memory
- Page tables are stored in pages of physical memory called *page table pages.*
- Problem we will talk about:
  - What if memory is only sparsely used (e.g., used-mem << address space == 4GB) (multiple levels)
  - Page tables are used each time an access to memory is performed. Adding a level of indirection, may reduce performance (need cache)
  - Each process has its own page table, they are not shared(think about the memory requirements of thousands or even millions of processes, each with a pafe table of billions of entries!!!)

Single Level Page Tables

- Contains one entry for each page frame
- Virtual address space is designed to be far larger than ever required (32 bits ;-))
- HW requires it to be in physically contiguous memory
  - a 4GB virtual address space requires 1million 4-byte entries
  - a 4TB virtual address space requires 1 billion entries
  - Reliably providing large regions of physically contiguous memory is difficult for the kernel!

Single-Level Page Tables

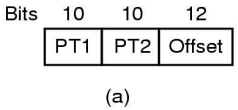


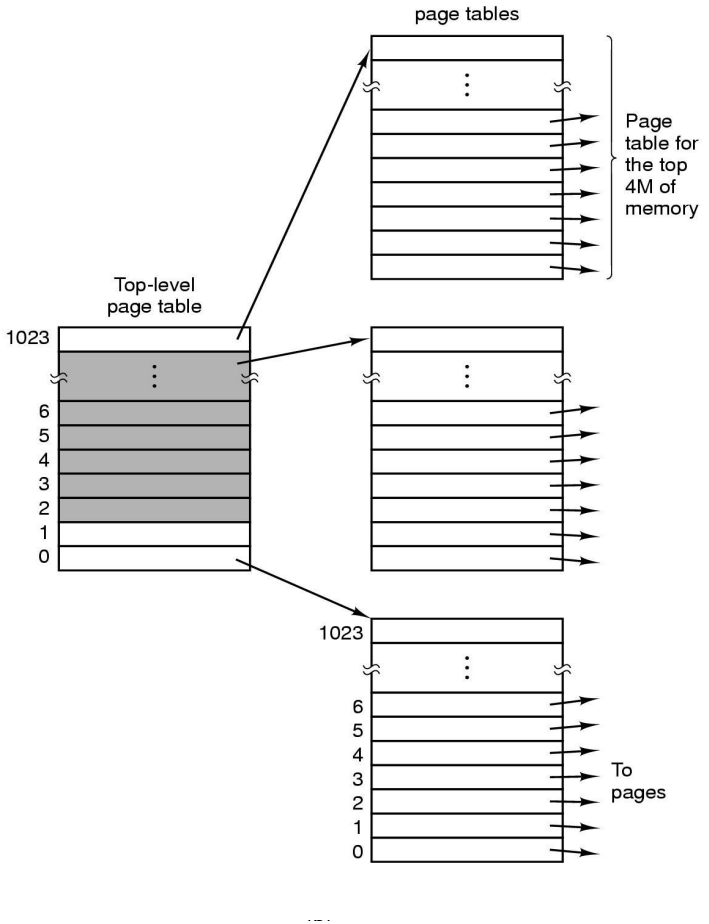
Multilevel Page Tables

- Single level is very inefficient if the virtual address space is expected to be sparse (i.e., not many mapped pages)
- Instead, multi-level page tables are used
  - The virtual address now has multiple indices
  - This allows us to only allocate tables for portions of the space that are used
- Page tables themselves are stored in memory!

Multilevel Page Tables

- 32 bit virtual address
- PT1: Top-level index, 10 bits
- PT2: Second-level index, 10 bits
- Offset: 12 bits
- Page size: 4KB
- Second-level maps 4MB (1024 entries of 4KB)
- Top-level maps 4GB (1024 entries of 4MB)



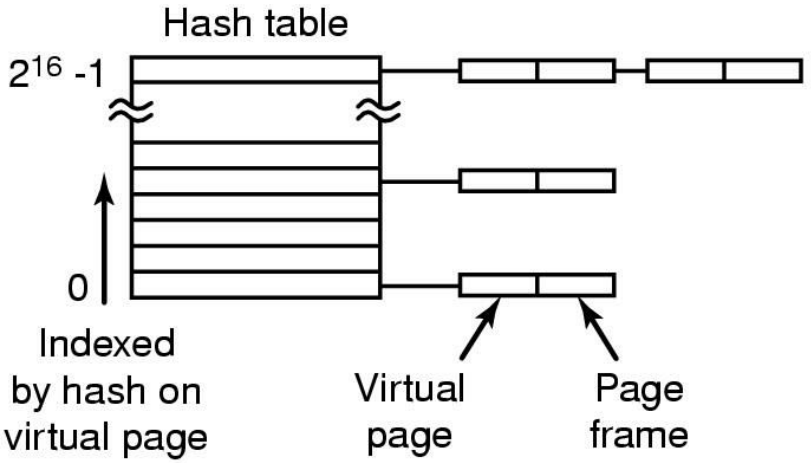


Inverted Page Table

- When virtual pages are too many, maintaining a page table is not feasible
  - In a worst case scenario, with a very sparse virtual address space, the number of page table pages can exceed the number of physical pages mapped in the address space.
  - This is a huge memory consumption problem if many processes are sparse.
- Solution: Inverted Page Table
  - One entry per physical page frame
  - Each entry contains a pair <process, virtual page>
- Address cannot be resolved simply by looking for an index in a table
- When process n accesses page p, the table must be scanned for an entry <n, p>
- Solution
  - TLB should catch most of the accesses
  - Table hashed on virtual address to resolve the mapping

Inverted Page Table

- Introduced by PowerPC, SPARK, Itanium
- Advantages:
  a. less overhead sparse; only one field
  b. can be way more efficient (book wrong)
- On Power, actually table no links…; HW can search in parallel
- Challenge: supporting large pages



Outline

- Handling miss in the translation
- Information in page table
- Page table organizations:
  - Single-level page table
  - Multi-level page tables
  - Inverted page tables
- Cache the translations

Translation Look-aside Buffers

- Translation Look-aside Buffer (TLB)
  - hardware device that allows fast access without using the page table
  - very fast cache that holds a subset of the processes page table entries.
- Small number of entries (e.g., 64-512) accessible as an associative memory(CPU hardware searches all TLB entries in parallel).
- Checked by hardware before doing a page table walk
- If lookup succeeds (hit), the page is accessed directly
- If TLB lookup fails (miss), the page table is used and the corresponding entry in TLB is added
- When an entry is taken out of the TLB, the modified bit is updated in the corresponding entry of the page table
- TLB management can be done both in hardware (MMU) or in software (by the OS)

Translation Look-aside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB on x86

- Can look this up on Linux with the `x86info` command:

```
x86info –c
```

TLB on x86

- Can look this up on Linux with the `x86info` command:

```
Cache info
TLB info
 Instruction TLB: 4K pages, 4-way associative, 64 entries.
 Data TLB: 4KB or 4MB pages, fully associative, 32 entries.
 Data TLB: 4KB pages, 4-way associative, 64 entries
 Data TLB: 4K pages, 4-way associative, 512 entries.
```

- Note that the x86 keeps multiple separate TLBs for code vs. data as well as different page sizes

Software TLB Management

- In some architectures (e.g., SPARC, MIPS), the TLB is managed by software
- TLB entries are explicitly loaded by the OS
- If we look up an address and it's not in the TLB (a *TLB miss*) we raise a TLB fault
- The OS then has to fill in the missing TLB entry

Software TLB Management

- Why manage the TLB explicitly instead of letting hardware do it?
  - The MMU can be much simpler, which saves space on the CPU that can be used for other things
  - Flexibility – the OS can choose its own algorithms for which TLB entries to evict and add
- But: this is generally slower than hardware-managed TLBs

It's a tradeoff!

TLBs and Context Switching

- TLBs map a virtual page to a physical page frame
- But, once we change to a new process, these mappings are no longer valid, and a *TLB flush* occurs
- This makes context switching more expensive – the first few memory accesses a process makes will have to be serviced by walking the page tables (i.e., 3x slower memory)

Tagged TLBs

- On most architectures today, each TLB entry can be associated with a *tag* that says what address space it belongs to.
- Think of this *tag* as being the process PID
- Now we don't have to flush the TLB when switching address spaces
- This can help make context switching faster – some TLB entries might still be valid when we switch back to a process

Global TLBs

- Similar to the Global Segment some TLB entries are marked as “Global” so they don’t get flushed during context switches
- Global TLB entries are used to map the kernel’s virtual address space.
- We never want to flush the kernel’s TLB entries during a context switch, every process needs the kernel.