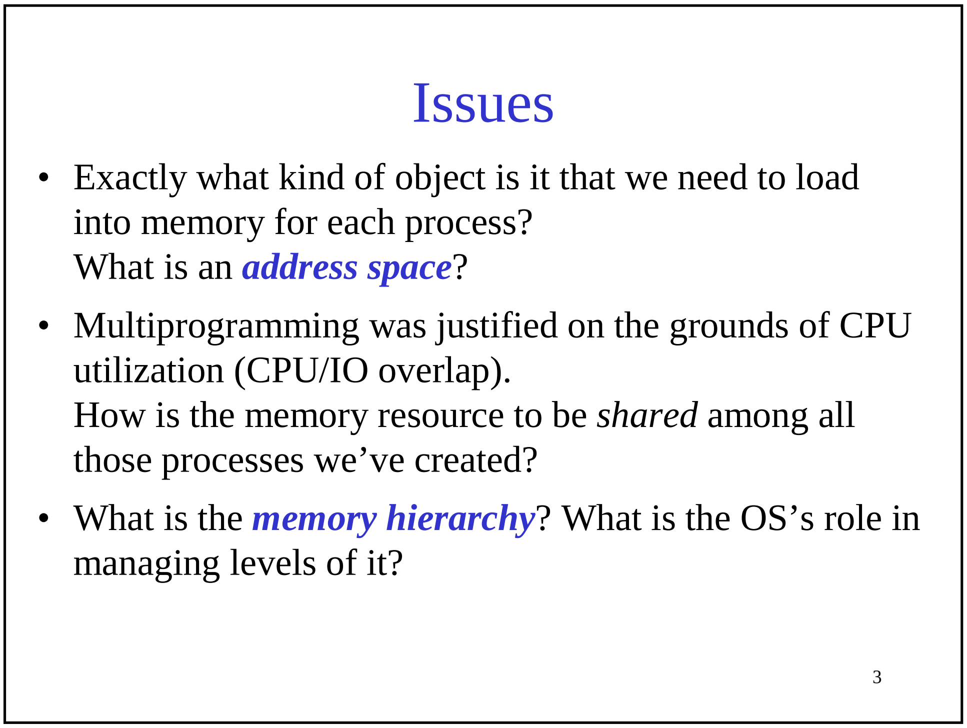
Outline

- Objectives
  - – Review of undergrad material w.r.t. memory management
  - – Linux details from ch 10,13 sprinkled along the way
- Administrative details
  - – Upcoming midterm (next Monday)

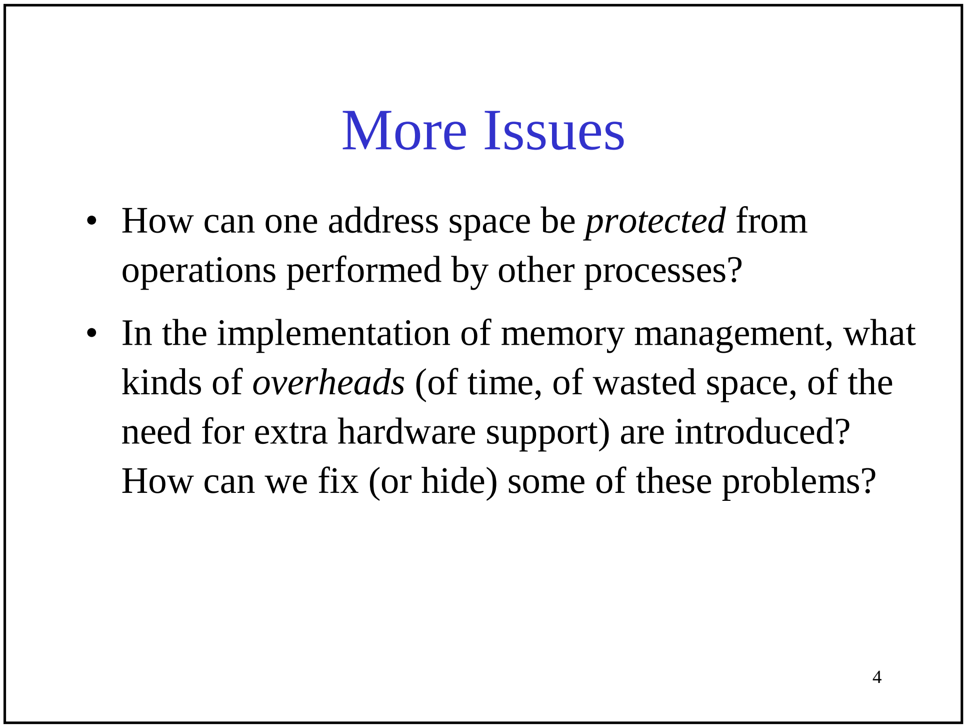
Review of Memory Management

- The traditional memory hierarchy,
  the virtual memory abstraction.
- Hardware and software mechanisms to support the abstraction.
- Management policies.
- Where are the opportunities for current research?
  The underlying assumptions that are changing.



Issues

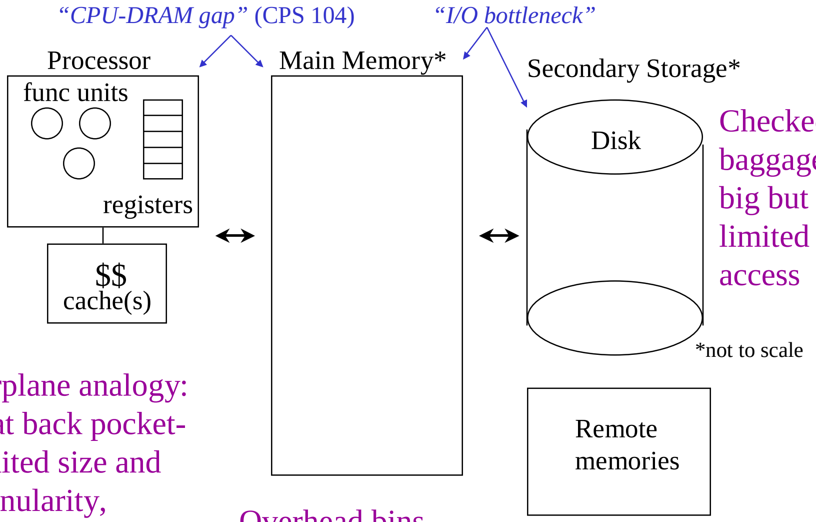
- Exactly what kind of object is it that we need to load into memory for each process?
  What is an ***address space***?
- Multiprogramming was justified on the grounds of CPU utilization (CPU/IO overlap).
  How is the memory resource to be *shared* among all those processes we've created?
- What is the ***memory hierarchy***? What is the OS's role in managing levels of it?



More Issues

- How can one address space be *protected* from operations performed by other processes?
- In the implementation of memory management, what kinds of *overheads* (of time, of wasted space, of the need for extra hardware support) are introduced? How can we fix (or hide) some of these problems?

Memory Hierarchy



Checked baggage- big but limited access

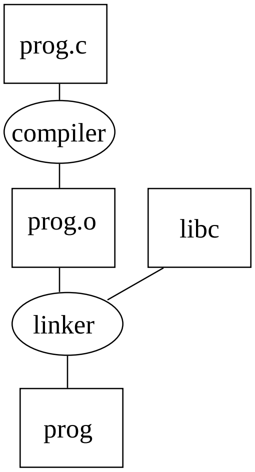
Airplane analogy: Seat back pocket- limited size and granularity, immediate access

Overhead bins- bigger, not as convenient

From User Program to Executable

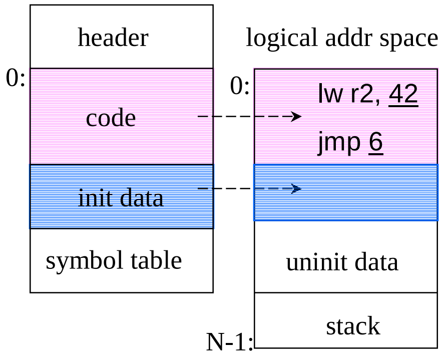
The executable file resulting from compiling your source code and linking with other compiled modules contains

- machine language instructions (as if addresses started at zero)
- initialized data
- how much space is required for uninitialized data



Executable to User Address Space

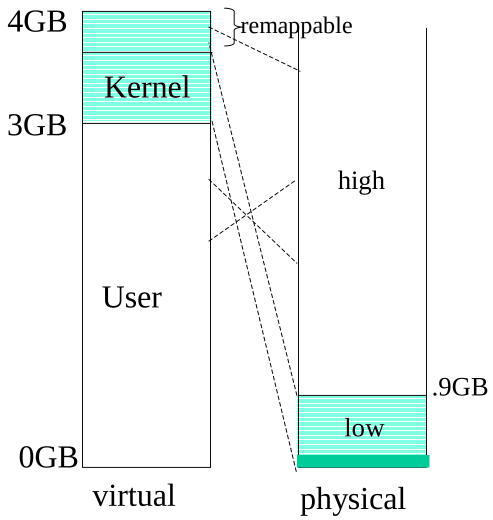
- In addition to the code and initialized data that can be copied from executable file, addresses must be reserved for areas of uninitialized data and stack when the process is created
- When and how do the real ***physical addresses*** get assigned?



7

Linux View of Memory

- Physical memory
- Kernel memory -- v.m. visible from kernel mode
  - ZONE_DMA
  - ZONE_NORMAL 1-to-1 mapped
  - ZONE_HIGHMEM explicitly mapped into address space
- User memory – visible from user mode
  - ZONE_HIGHMEM



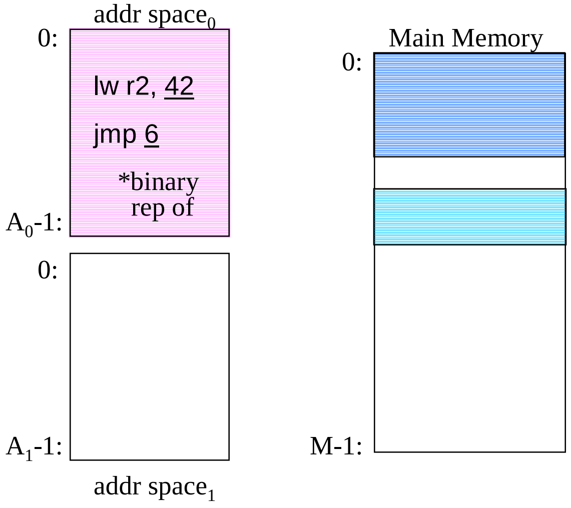
Allocation Mechanisms

- Acquiring regions of physical memory and mapping them (implicitly or explicitly) into the virtual address space
- Kernel allocators
  - Page grained – `alloc_pages` and `get_zeroed_page`
    `kmap` into kernel address space
  - Sub-page-size – rtn logical addresses
    - `kmalloc` – physical contiguous chunks
    - `vmalloc` – virtually contiguous

Slab Allocator

- Creates cache of pre-allocated and recycled data structures. Essentially free-lists of various kinds
- `kmem_cache_alloc` – gets an object of the appropriate type from the cache
  - Inodes, task_structs, etc.

Allocation to Physical Memory



Assume contiguous
allocation

Static loading

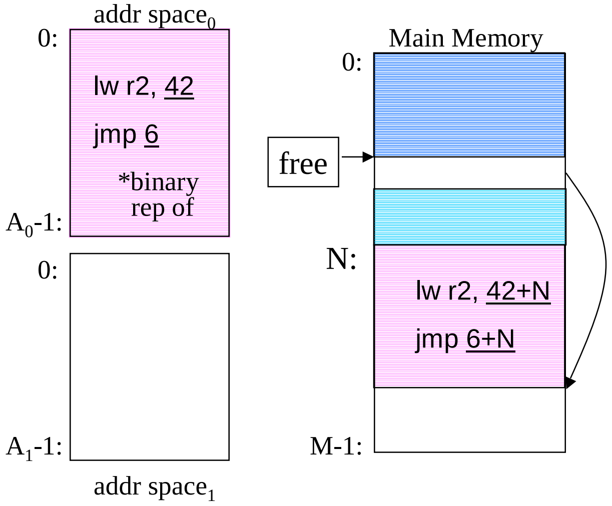
Partition memory
variable (first, best
fits)

Fragmentation
(external)

Compaction

Swapping

Allocation to Physical Memory



Assume contiguous
allocation

Static loading

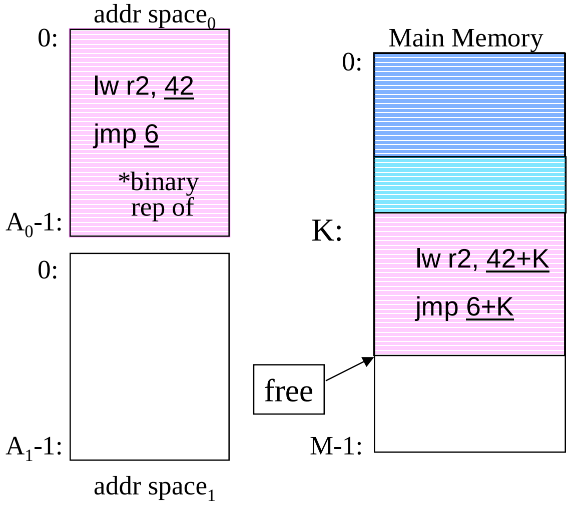
Partition memory
variable (first, best
fits)

Fragmentation
(external)

Compaction

Swapping

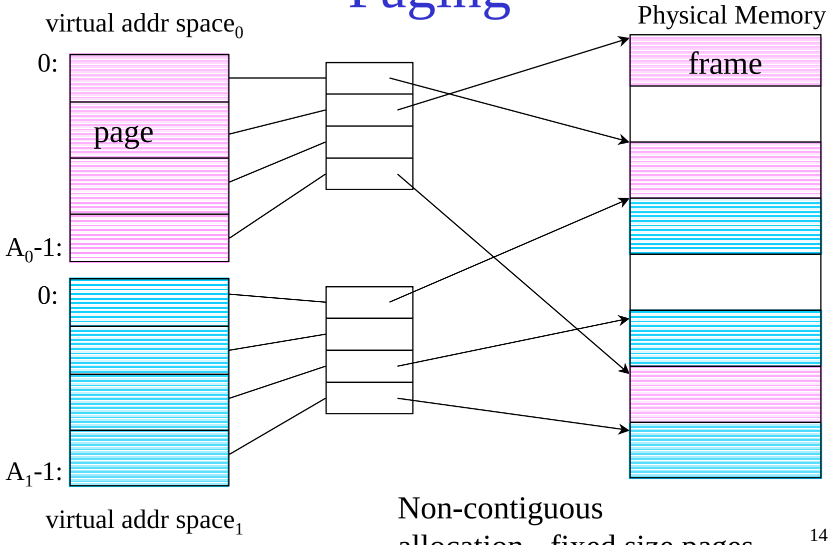
Allocation to Physical Memory



- Assume contiguous allocation
- Static loading
- Partition memory variable (first, best fits)
- Fragmentation (external)
- Compaction
- Swapping

13

Paging



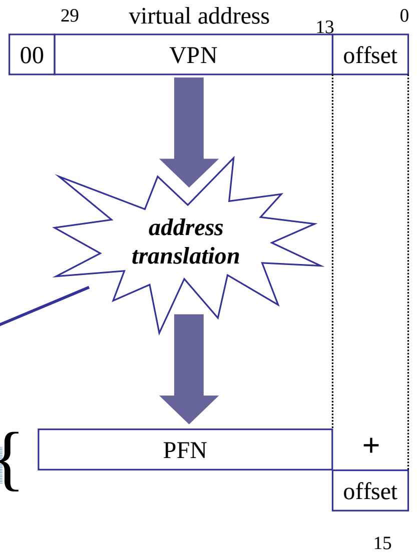
Non-contiguous allocation - fixed size pages

14

Paging

Dynamic address translation

- another case of indirection (as "the answer")
- TLB to speed up lookup (another case of **caching** as "the answer")



Deliver exception to OS if translation is not valid and accessible in requested mode.

physical address

Virtual Memory

- System-controlled movement up and down in the memory hierarchy.
- Can be viewed as automating overlays - the system attempts to dynamically determine which previously loaded parts can be replaced by parts needed now.
- It works only because of ***locality*** of reference.
- Often most closely associated with paging (needs non-contiguous allocation, mapping table).

Locality

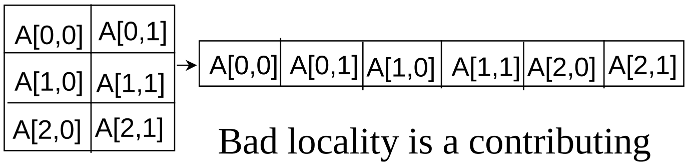
- Only a subset of the program’s code and data are needed at any point in time. Can the OS predict what that subset will be (from observing only the past behavior of the program)?
- ***Temporal*** - Reuse. Tendency to reuse stuff accessed in recent history (code loops).
- ***Spatial*** - Tendency to use stuff near other recently accessed stuff (straightline code, data arrays). Justification for moving in larger chunks.

Good & Bad Locality

```
for (i = 0; i++; i<n)
   for (j = 0; j++; j<m)
        A[i, j] = B[i, j]
```

```
for (j = 0; j++; j<m)
   for (i = 0; i++; i<n)
        A[i, j] = B[i, j]
```

Assume: arrays laid out in rows



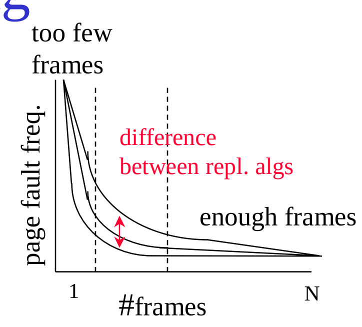
A[0,0]	A[0,1]
A[1,0]	A[1,1]
A[2,0]	A[2,1]

A[0,0]	A[0,1]	A[1,0]	A[1,1]	A[2,0]	A[2,1]

Bad locality is a contributing factor in ***Thrashing*** (page faulting behavior dominates).

Thrashing

- Page faulting is dominating performance
- Causes:
  - Memory is overcommited - not enough to hold locality sets of all processes at this level of multiprogramming
  - Lousy locality of their programs
  - Positive feedback loops reacting to paging I/O rates



- Load control is important (how many slices in frame pie?)
  - LT/RT strategy **(limit # processes in load phase)**

Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?

   Address translation must be fast.
3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted from memory, where do we put it?
7. How do we decide which pages to evict from memory?

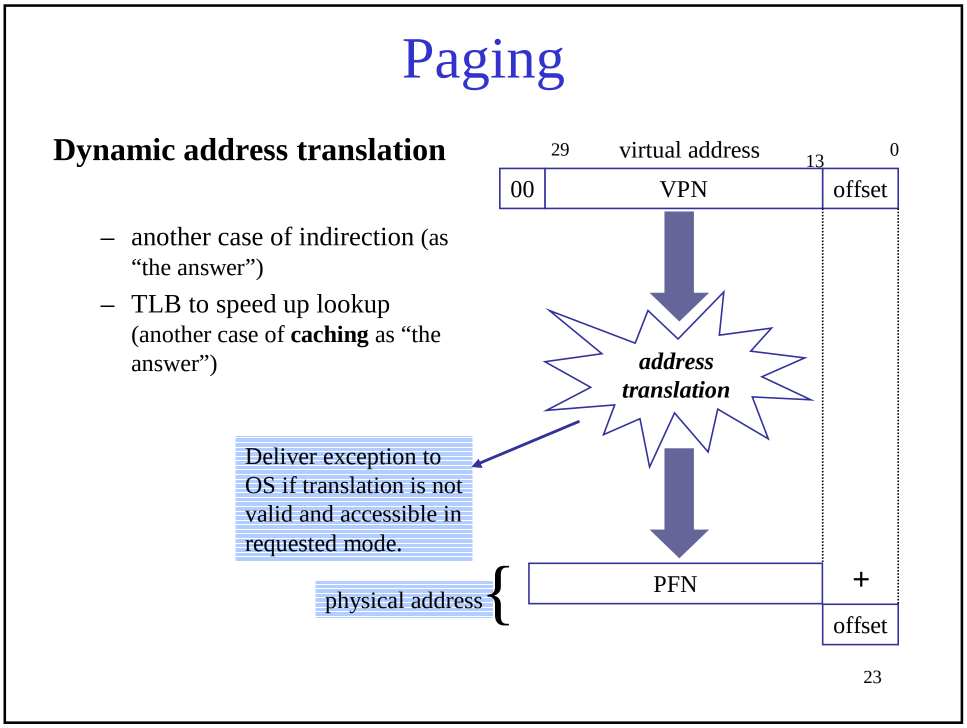
   Page replacement policy should minimize overall I/O.

Virtual Memory Mechanisms

- Hardware support - beyond dynamic address translation needed to support paging or segmentation (e.g., table lookup)
  - mechanism to generate page fault on missing page.
  - restartable instructions
- Software
  - Data to support replacement, fetch, and placement policies.
  - Data structure for location in secondary memory of desired page.

Role of MMU Hardware and OS

- VM address translation must be very cheap (on average).
  - Every instruction includes one or two memory references.
    - (including the reference to the instruction itself)
- VM translation is supported in hardware by a ***M***emory ***M***anagement ***U***nit or ***MMU.***
  - The addressing model is defined by the CPU architecture.
  - The MMU itself is an integral part of the CPU.
- The role of the OS is to *install* the virtual-physical mapping and *intervene* if the MMU reports a violation.



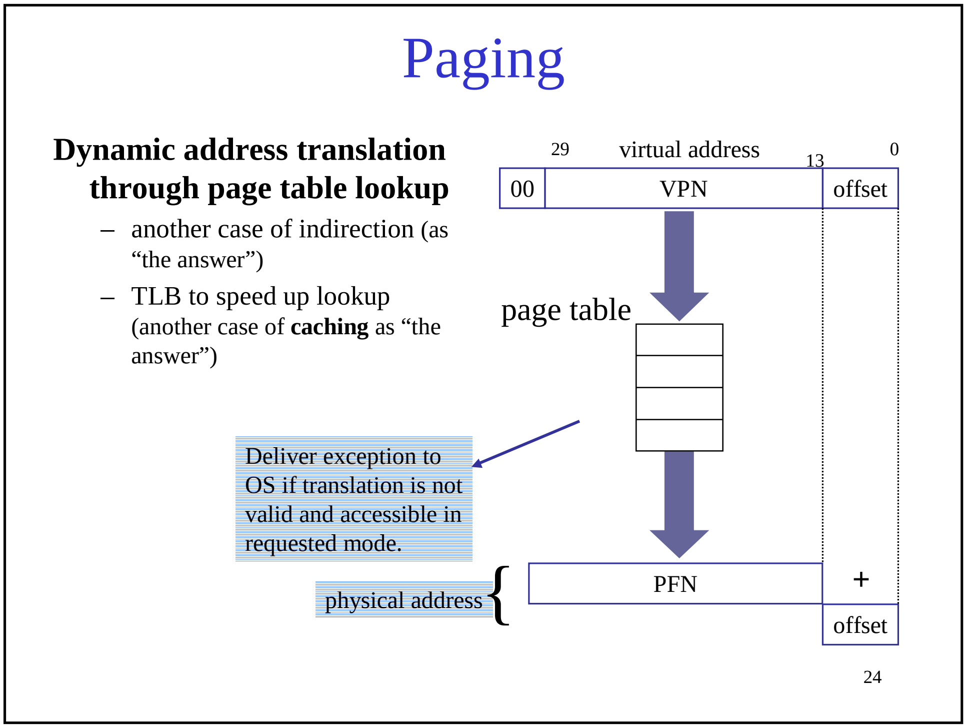
Paging

Dynamic address translation

- – another case of indirection (as "the answer")
- – TLB to speed up lookup (another case of **caching** as "the answer")

Deliver exception to OS if translation is not valid and accessible in requested mode.

23



Paging

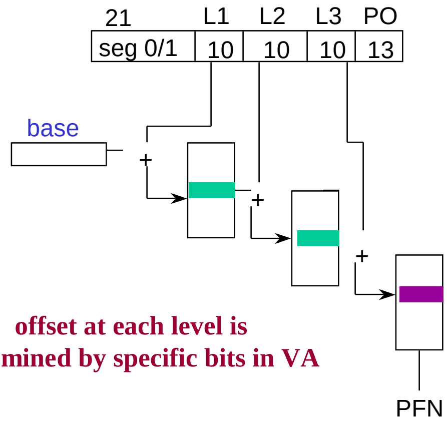
Dynamic address translation through page table lookup

- – another case of indirection (as "the answer")
- – TLB to speed up lookup (another case of **caching** as "the answer")

Deliver exception to OS if translation is not valid and accessible in requested mode.

24

Alpha Page Tables (Forward Mapped)



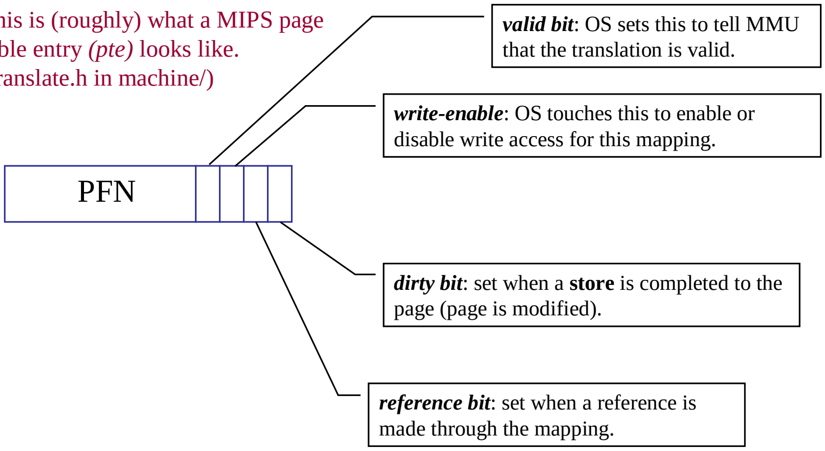
sparse 64-bit address space (43 bits in 21064 and 21164)

three-level page table (forward-mapped)

offset at each level is determined by specific bits in VA

A Page Table Entry (PTE)

This is (roughly) what a MIPS page table entry *(pte)* looks like. (translate.h in machine/)



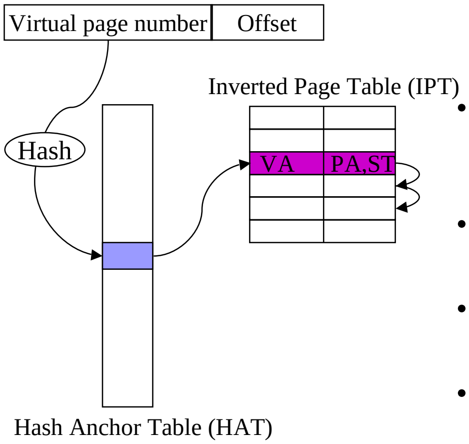
valid bit: OS sets this to tell MMU that the translation is valid.

write-enable: OS touches this to enable or disable write access for this mapping.

dirty bit: set when a **store** is completed to the page (page is modified).

reference bit: set when a reference is made through the mapping.

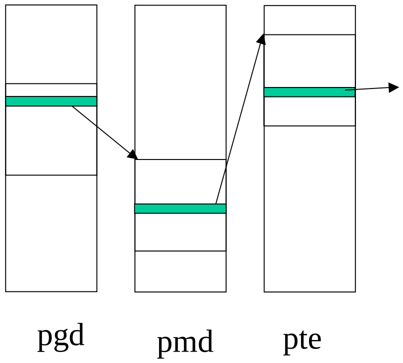
Inverted Page Table (HP, IBM)



- One PTE per page frame
  - only one VA per physical frame
- Must search for virtual address
- More difficult to support aliasing
- Force all sharing to use the same VA

Linux Page Table

- 3-levels
  - pgd – page directory
  - pmd – page middle directory
  - pte – page table entries



Memory Management Unit (MMU)

- Input
  - virtual address
- Output
  - physical address
  - access violation (exception, interrupts the processor)
- Access Violations
  - not present
  - user v.s. kernel
  - write
  - read
  - execute

The OS Directs the MMU

- The OS controls the operation of the MMU to select:

  (1) the subset of possible virtual addresses that are valid for each process (the process *virtual address space*);

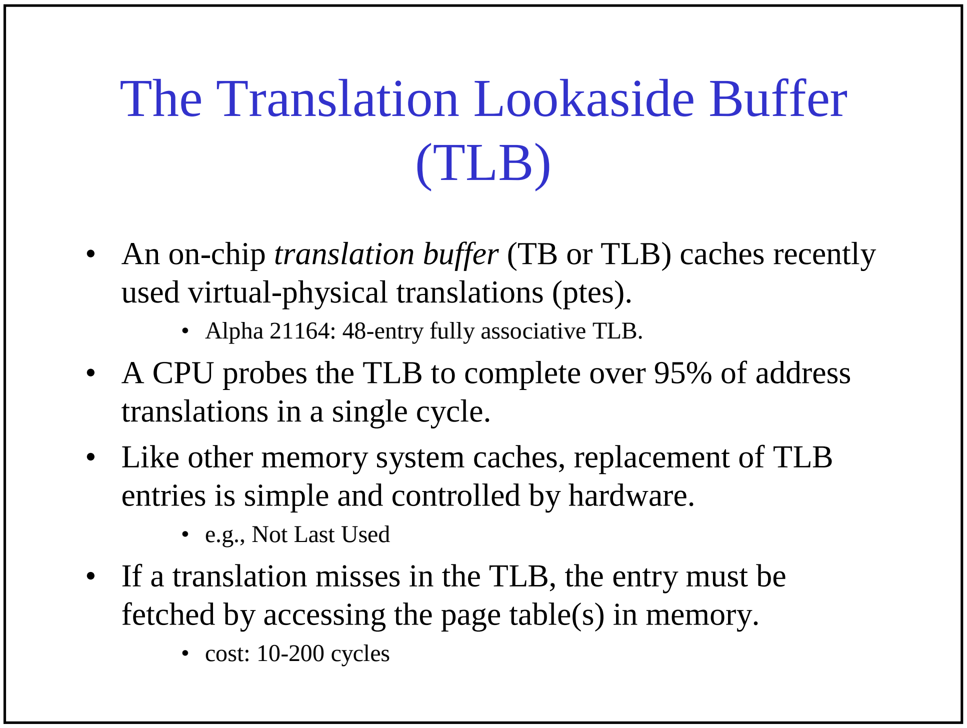
  (2) the physical translations for those virtual addresses;

  (3) the modes of permissible access to those virtual addresses;

  - read/write/execute

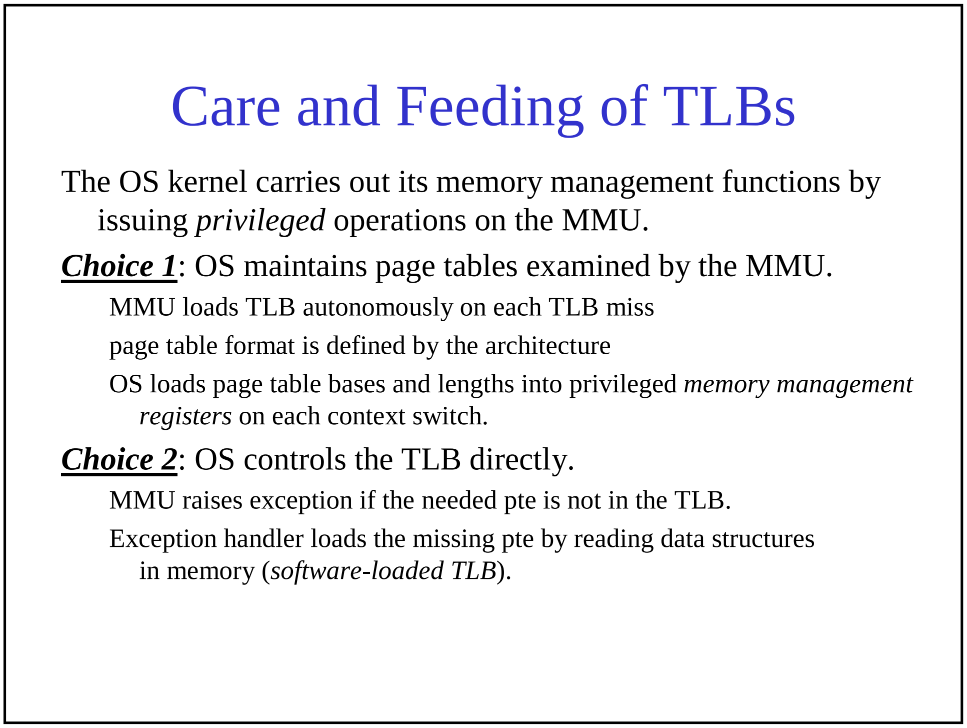
  (4) the specific set of translations in effect at any instant.

  - need rapid context switch from one address space to another
- MMU completes a reference only if the OS “says it’s OK”.
  - MMU raises an exception if the reference is “not OK”.



The Translation Lookaside Buffer (TLB)

- An on-chip *translation buffer* (TB or TLB) caches recently used virtual-physical translations (ptes).
  - Alpha 21164: 48-entry fully associative TLB.
- A CPU probes the TLB to complete over 95% of address translations in a single cycle.
- Like other memory system caches, replacement of TLB entries is simple and controlled by hardware.
  - e.g., Not Last Used
- If a translation misses in the TLB, the entry must be fetched by accessing the page table(s) in memory.
  - cost: 10-200 cycles



Care and Feeding of TLBs

The OS kernel carries out its memory management functions by issuing *privileged* operations on the MMU.

Choice 1: OS maintains page tables examined by the MMU.

MMU loads TLB autonomously on each TLB miss

page table format is defined by the architecture

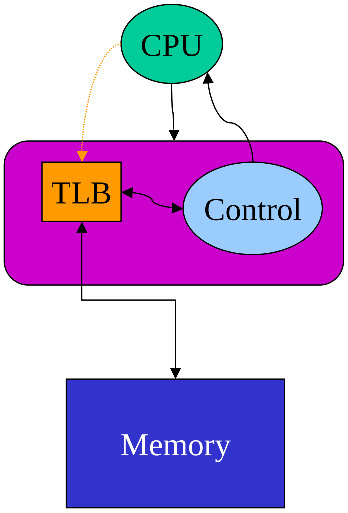
OS loads page table bases and lengths into privileged *memory management registers* on each context switch.

Choice 2: OS controls the TLB directly.

MMU raises exception if the needed pte is not in the TLB.

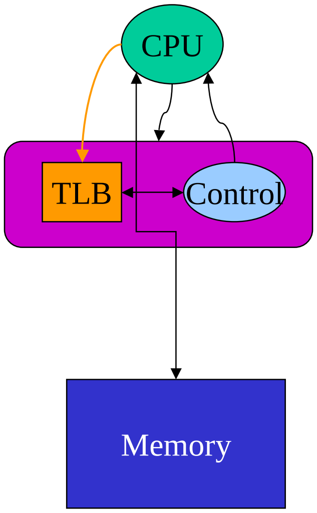
Exception handler loads the missing pte by reading data structures in memory (*software-loaded TLB*).

Hardware Managed TLBs



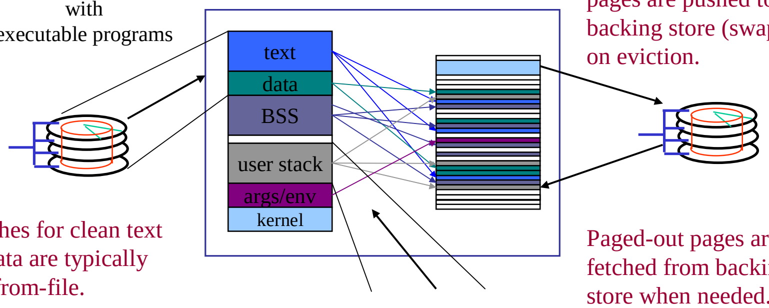
- Hardware Handles TLB miss
- Dictates page table organization
- Compilicated state machine to “walk page table”
  - Multiple levels for forward mapped
  - Linked list for inverted
- Exception only if access violation

Software Managed TLBs



- Software Handles TLB miss
- Flexible page table organization
- Simple Hardware to detect Hit or Miss
- Exception if TLB miss or access violation
- Should you check for access violation on TLB miss?

Where Pages Come From



file volume
with
executable programs

Modified (dirty) pages are pushed to backing store (swap) on eviction.

Fetches for clean text or data are typically fill-from-file.

Paged-out pages are fetched from backing store when needed.

Initial references to user stack and BSS are satisfied by zero-fill on demand.

Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?

   Address translation must be fast.
3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted from memory, where do we put it?
7. How do we decide which pages to evict from memory?

   Page replacement policy should minimize overall I/O.

Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?

   Indirection through MMU is the way to get to physical memory and the protection bits in the PTEs come into play.

2. If a page is in memory, how do we find it?

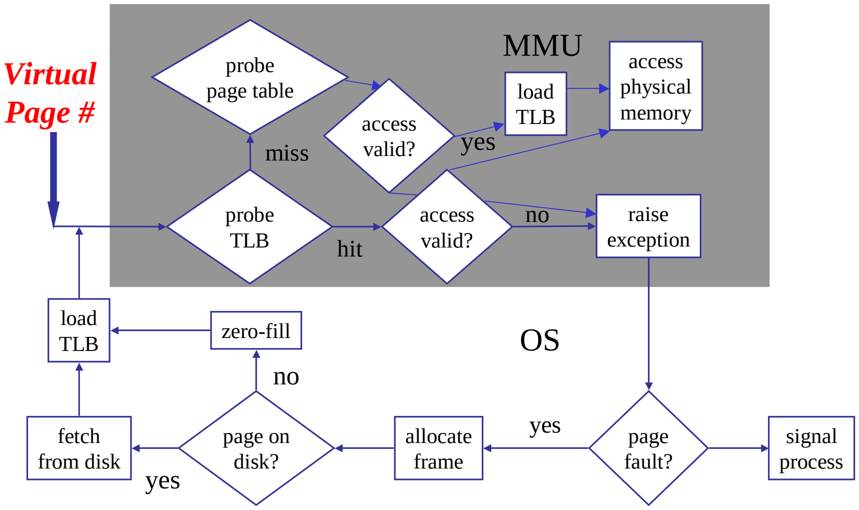
   Address translation must be fast.

   TLB

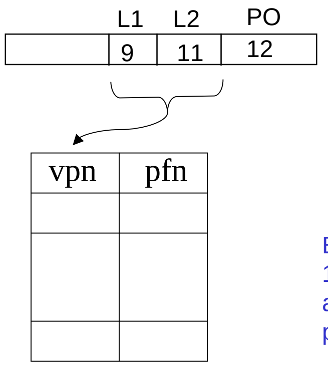
3. If a page is not in memory, how do we find it?

   A miss in the TLB and then an invalid mapping in the page table signify non-resident page - creating an exception (page fault) Another table will give location in backing store.

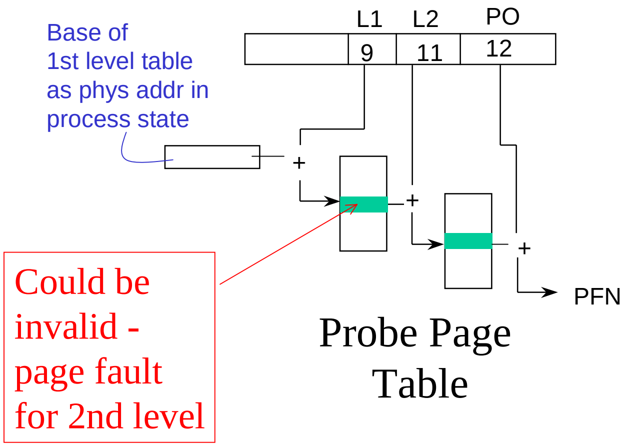
Completing a VM Reference



Addition



Probe TLB



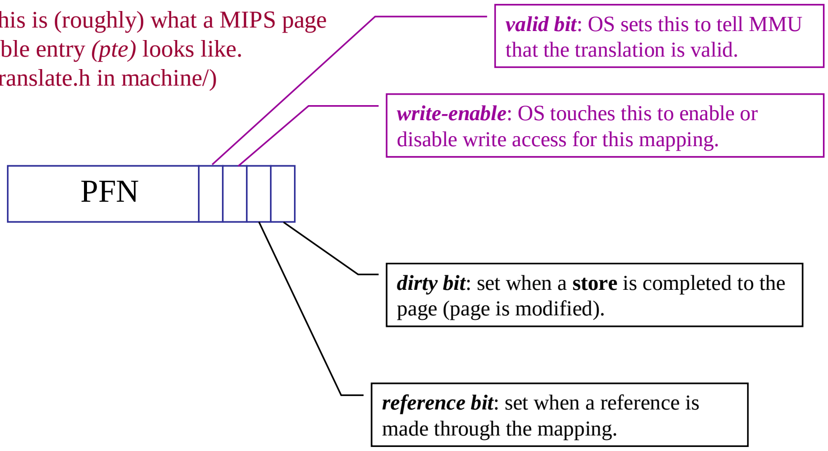
Base of 1st level table as phys addr in process state

Could be invalid - page fault for 2nd level

Probe Page Table

A Page Table Entry (PTE)

This is (roughly) what a MIPS page table entry *(pte)* looks like. (translate.h in machine/)



valid bit: OS sets this to tell MMU that the translation is valid.

write-enable: OS touches this to enable or disable write access for this mapping.

dirty bit: set when a **store** is completed to the page (page is modified).

reference bit: set when a reference is made through the mapping.

Questions for Paged Virtual Memory

1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?

   Address translation must be fast.
3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted from memory, where do we put it?
7. How do we decide which pages to evict from memory?

   Page replacement policy should minimize overall I/O.

Policies for Paged Virtual Memory

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

(1) ***fetch policy*****:** When are pages brought into memory?
- prepaging: reduce page faults by bring pages in before needed
- on demand: in direct response to a page fault.

(2) ***replacement policy***: How and when does the system select victim pages to be evicted/discarded from memory?

(3) ***placement policy****:* Where are incoming pages placed? Which frame?

(4) ***backing storage policy*****:**
- Where does the system store evicted pages?
- When is the backing storage allocated?
- When does the system write modified pages to backing store?
- Clustering: reduce seeks on backing storage

Fetch Policy: Demand Paging

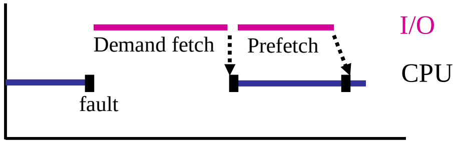
- Missing pages are loaded from disk into memory at *time of reference* (*on demand*).
  The alternative would be to prefetch into memory in anticipation of future accesses (need good predictions).
- Page fault occurs because *valid bit* in page table entry (PTE) is *off*. The OS:
  – allocates an empty frame*
  – initiates the read of the page from disk
  – updates the PTE when I/O is complete
  – restarts faulting process

* Placement and possible Replacement policies

2/22/2005 44

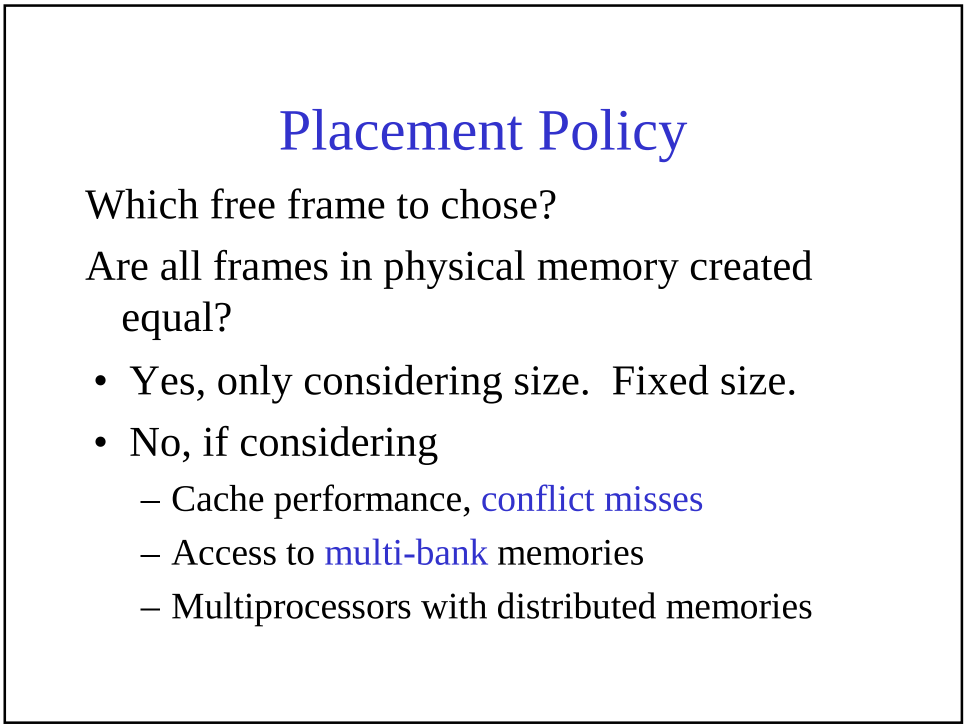
Prefetching Issues

- Pro: overlap of disk I/O and computation on resident pages. Hides latency of transfer.
  – Need information to guide predictions
- Con: bad predictions
  – Bad choice: a page that will never be referenced.
  – Bad timing: a page that is brought in too soon



Impacts:

– taking up a frame that would otherwise be free.
– (worse) replacing a useful page.
– extra I/O traffic

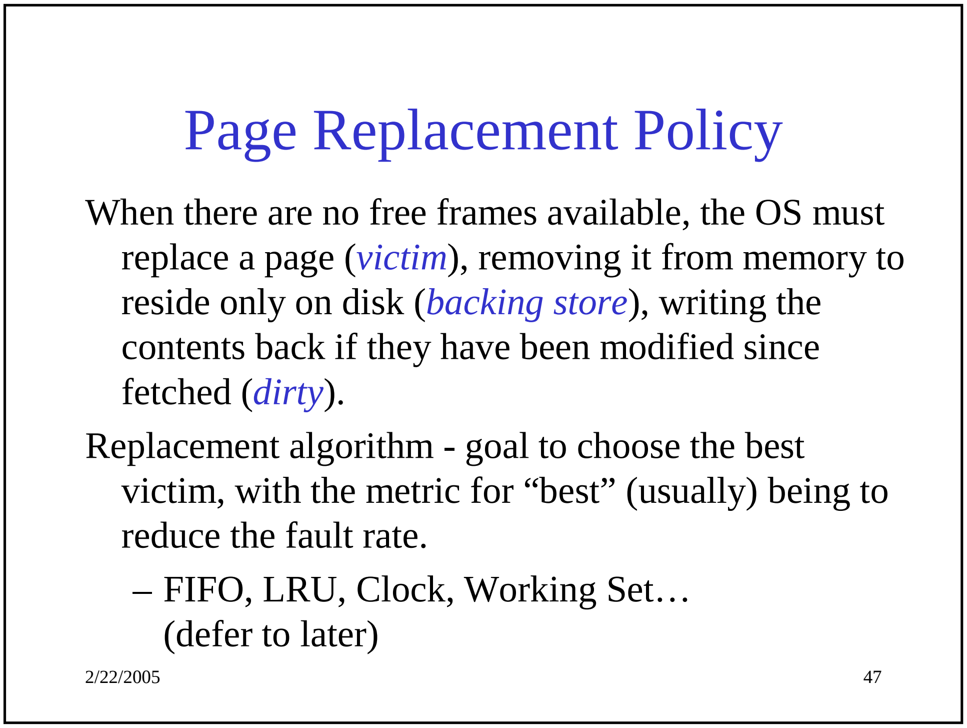


Placement Policy

Which free frame to chose?

Are all frames in physical memory created equal?

- Yes, only considering size. Fixed size.
- No, if considering
  - Cache performance, conflict misses
  - Access to multi-bank memories
  - Multiprocessors with distributed memories



Page Replacement Policy

When there are no free frames available, the OS must replace a page (*victim*), removing it from memory to reside only on disk (*backing store*), writing the contents back if they have been modified since fetched (*dirty*).

Replacement algorithm - goal to choose the best victim, with the metric for “best” (usually) being to reduce the fault rate.

- FIFO, LRU, Clock, Working Set… (defer to later)

The Page Caching Problem (aka Replacement Policy)

- Each thread/process/job utters a stream of page references.
  - Model execution as a *page reference string:* e.g., “abcabcdabce..”
- The OS tries to minimize the number of faults incurred.
  - The set of pages (the *working set*) actively used by each job changes relatively slowly.
  - Try to arrange for the *resident set* of pages for each active job to closely approximate its working set.
- Replacement policy is the key.
  - Determines the resident subset of pages..

Replacement Algorithms

Assume fixed number of frames in memory assigned to this process:

- Optimal - baseline for comparison - future references known in advance - replace page used furthest in future.
- FIFO
- Least Recently Used (LRU)

  stack algorithm - don’t do worse with more memory.
- LRU approximations for implementation

  Clock, Aging register

LRU

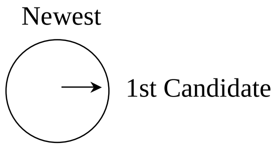
- At fault time: replace the resident page that was last used the longest time ago
- Idea is to track the program’s temporal locality
- To implement exactly: we need to order the pages by time of most recent reference
  (per-reference information needed –> HW, too $$)
  - timestamp pages at each ref, stack operations at each ref
- Stack algorithm - doesn’t suffer from Belady’s anomaly -- if i > j then set of pages with j frames is a subset of set of pages with i frames.

LRU Approximations for Paging

- Pure LRU and LFU are prohibitively expensive to implement.
  - most references are hidden by the TLB
  - OS typically sees less than 10% of all references
  - can’t tweak your ordered page list on every reference
- Most systems rely on an *approximation* to LRU for paging.
  - periodically *sample* the reference bit on each page
    - visit page and set reference bit to zero
    - run the process for a while (the *reference window*)
    - come back and check the bit again
  - reorder the list of eviction candidates based on sampling

Clock Algorithm

- Maintain a circular queue with a pointer to the next candidate (clock hand).
- At fault time: scan around the clock, looking for page with usage bit of zero (that’s your victim), clearing usage bits as they are passed.
- We now know whether or not a page has been used *since the last time the bits were cleared*



Newest

1st Candidate

Practical Considerations

- Dirty bit - modified pages require a writeback to secondary storage before frame is free to use again.
- Variation tries to maintain a healthy pool of clean, free frames
  - on timer interrupt, scan for unused pages, move to free pool, initiate writeback on dirty pages
  - at fault time, if page is still in frame in pool, reclaim; else take free, clean frame.

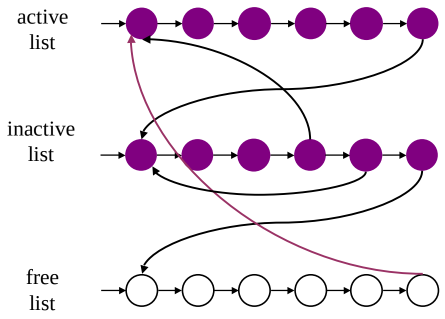
The Paging Daemon

- Most OS have one or more system processes responsible for implementing the VM page cache replacement policy.
  - A *daemon* is an autonomous system process that periodically performs some housekeeping task.
- The *paging daemon* prepares for page eviction before the need arises.
  - Wake up when free memory becomes low.
  - Clean dirty pages by pushing to backing store.
    - *prewrite* or *pageout*
  - Maintain ordered lists of eviction candidates.
  - Decide how much memory to allocate to UBC, VM, etc.

FIFO with Second Chance (Mach)

- ***Idea***: do simple FIFO replacement, but give pages a "second chance" to prove their value before they are replaced.
  - Every frame is on one of three FIFO lists:
    - *active*, *inactive* and *free*
  - Page fault handler installs new pages on tail of active list.
  - "Old" pages are moved to the tail of the inactive list.
    - Paging daemon moves pages from head of active list to tail of inactive list when demands for free frames is high.
    - Clear the refbit and protect the inactive page to "monitor" it.
  - Pages on the inactive list get a "second chance".
    - If referenced while inactive, *reactivate* to the tail of active list.

Illustrating FIFO-2C



Restock inactive list by pulling pages from the head of the active list: knock off the reference bit and inactivate.

Inactive list scan:
1. Page on inactive list has been referenced? Return to tail of active list (*reactivation*).

2. Page at head of inactive list has not been referenced? *pmap_page_protect* and place on tail of free list.

3. Dirty page on inactive list? Push and return to inactive list tail.

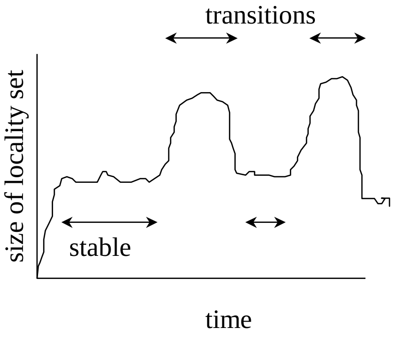
Paging daemon scans a few times per second, even if not needed to restock free list.

Consume frames from the head of the free list.

If free shrinks below threshhold, kick the paging daemon to start a scan.

Variable / Global Algorithms

- Not requiring each process to live within a fixed number of frames, replacing only its own pages.
- Can apply previously mentioned algorithms globally to victimize any process's pages
- Algorithms that make number of frames explicit.



Variable Space Algorithms

- Working Set

  Tries to capture what the set of active pages currently is. The whole working set should be resident in memory for the process to bother running. WS is set of pages referenced during window of time (now-t, now).

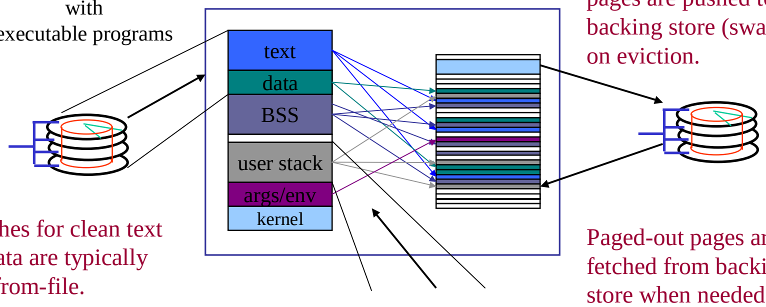
  – Working Set Clock - a hybrid approximation

- Page Fault Frequency

  Monitor fault rate, if pff > high threshold, grow # frames allocated to this process, if pff < low threshold, reduce # frames. Idea is to determine the right amount of memory to allocate.

Backing Store = Disk

file volume
with
executable programs



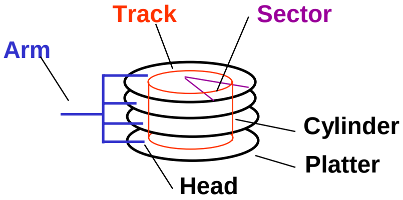
Modified (dirty) pages are pushed to backing store (swap) on eviction.

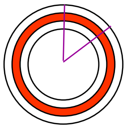
Fetches for clean text or data are typically fill-from-file.

Paged-out pages are fetched from backing store when needed.

Initial references to user stack and BSS are satisfied by zero-fill on demand.

Rotational Media





Access time = seek time + rotational delay + transfer time

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

Layout issues: clustering

A Case for Large Pages

- Page table size is inversely proportional to the page size
  - memory saved
- Transferring larger pages to or from secondary storage (possibly over a network) is more efficient
- Number of TLB entries are restricted by clock cycle time,
  - larger page size maps more memory
  - reduces TLB misses

A Case for Small Pages

- Fragmentation
  - not *that* much spatial locality
  - large pages can waste storage
  - data must be contiguous within page

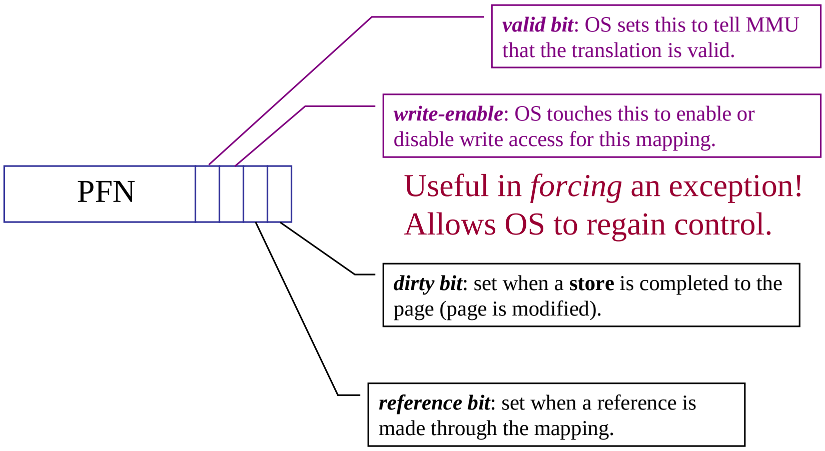
MMU Games

Vanilla Demand Paging

- Valid bit in PTE means non-resident page. Resulting page fault causes OS to initiate page transfer from disk.
- Protection bits in PTE means page should not be accessed in that mode (usually means non-writable)

What *else* can you do with them?

A Page Table Entry (PTE)



valid bit: OS sets this to tell MMU that the translation is valid.

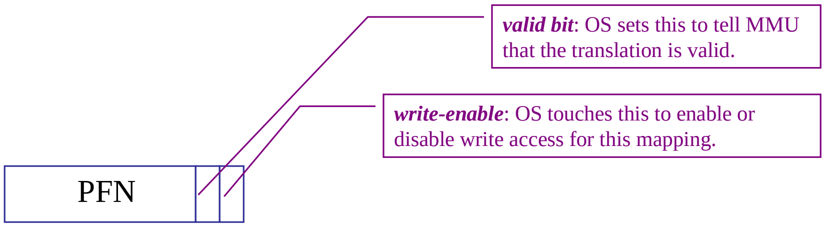
write-enable: OS touches this to enable or disable write access for this mapping.

Useful in *forcing* an exception! Allows OS to regain control.

dirty bit: set when a **store** is completed to the page (page is modified).

reference bit: set when a reference is made through the mapping.

Simulating Usage Bits



valid bit: OS sets this to tell MMU that the translation is valid.

write-enable: OS touches this to enable or disable write access for this mapping.

- Turn off both valid bit and write-protect bit
- On first reference - fault allows recording the reference bit information by OS in an auxillary data structure. Set it valid for subsequent accesses to go through HW.
- On first write attempt - protection fault allows recording the dirty bit information by OS in aux. data structure.

Copy-on-Write

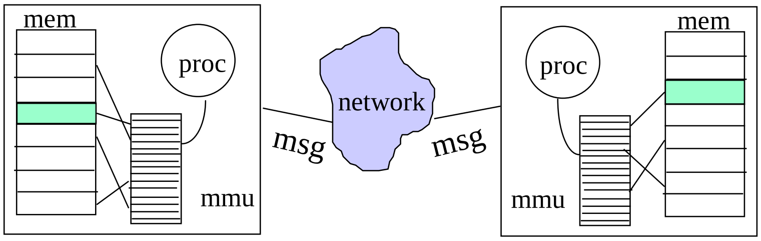
- Operating systems spend a lot of their time copying data.
  - particularly Unix operating systems, e.g., *fork()*
  - cross-address space copies are common and expensive
- *Idea*: defer big copy operations as long as possible, and hope they can be avoided completed.
  - create a new *shadow* object backed by an existing object
  - shared pages are mapped readonly in participating spaces
    - read faults are satisfied from the original object (typically)
    - write faults trap to the kernel
      - make a (real) copy of the faulted page
      - install it in the shadow object with writes enabled

Things Change

- Myth that placement is irrelevant
- View that OS is concerned only with the main-secondary levels of memory hierarchy
- New architectures / new views of the memory “hierarchy”
- Scale - larger address spaces
- Workload assumptions
  - New things to do with memory management

Distributed Shared Memory (DSM)

Allows use of a shared memory programming model (shared address space) in a distributed system (processors with only local memory)



DSM Issues

- Can use the local memory management hardware to generate fault when desired page is not locally present or when write attempted on read-only copy.
- Locate the page remotely - current “owner” of page (last writer) or “home” for page.
- Page sent in message to requesting node (read access makes copy; write migrates)
- Consistency protocol - invalidations or broadcast of changes (update)
  – directory kept of caches holding copies

DSM States

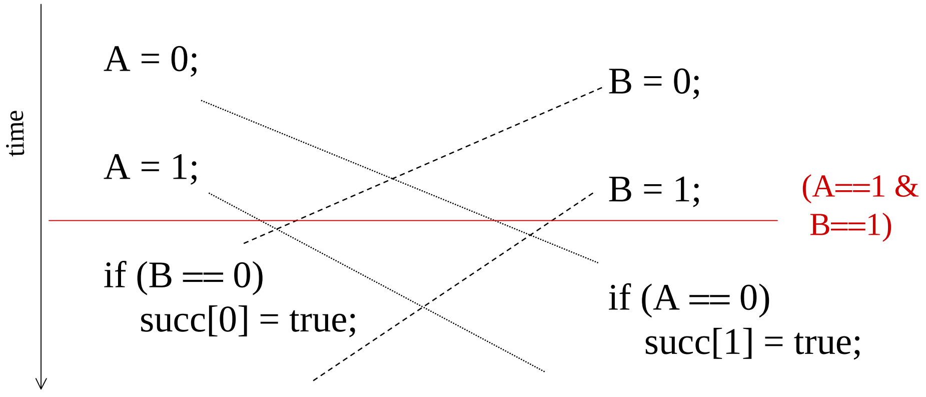
Forced faults are key to consistency operations

- Invalid local mapping, attempted read access -
  data flushed from most recent writer,
  set write-protect bit for all copies.
- Invalid local mapping, attempted write access -
  migrate data, invalidate all other copies.
- Local read-only copy, write-fault -
  invalidate all other copies

Consistency Models

- Sequential consistency
  - All memory operations *appear* to execute one at a time. A write is considered *done* only when invalidations or updates have propagated to all copies.
- Weaker forms of consistency
  - Guarantees associated with synchronization primitives; at other times, it doesn’t matter
  - For example:
    acquire lock - make sure others’ writes are done
    release lock - make sure all my writes are seen by others

Example - the Problem



time

A = 0;

A = 1;

if (B == 0)
    succ[0] = true;

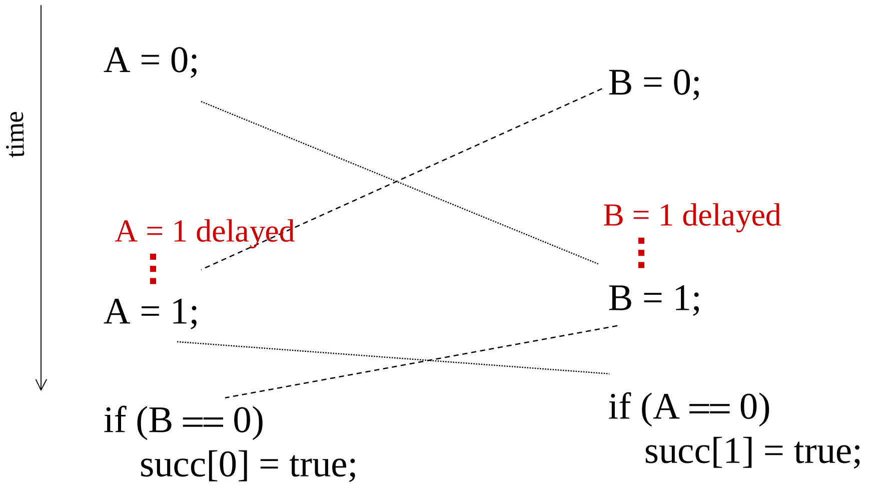
B = 0;

B = 1;

(A==1 & B==1)

if (A == 0)
    succ[1] = true;

Example - Sequential Consistency



time

A = 0;

A = 1 delayed

A = 1;

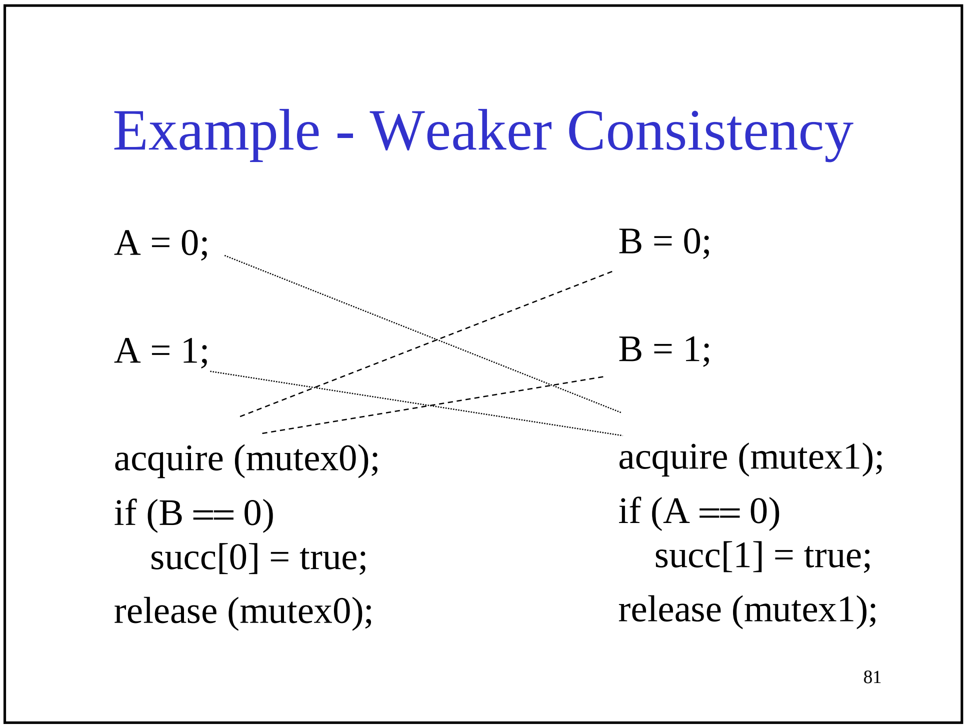
if (B == 0)
    succ[0] = true;

B = 0;

B = 1 delayed

B = 1;

if (A == 0)
    succ[1] = true;



Example - Weaker Consistency

```
A = 0;

A = 1;

acquire (mutex0);
if (B == 0)
   succ[0] = true;
release (mutex0);
```

```
B = 0;

B = 1;

acquire (mutex1);
if (A == 0)
   succ[1] = true;
release (mutex1);
```