

Outline for Today's Lecture

- Last time: Finished up Processor Management and Concurrent Programming.
- Objective: [Major context switch] Memory Management
 - Review of 104

1

Issues

- Exactly what kind of object is it that we need to load into memory for each process?
What is the definition of 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?

2

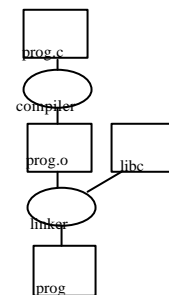
More Issues

- 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?

3

From Program to Executable

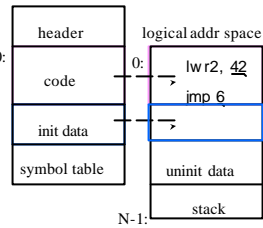
- The executable file resulting from compiling your source code and linking with other compiled modules contains
- machine language instructions (as if addresses started at zero)
 - initialized data
 - how much space is required for uninitialized data



4

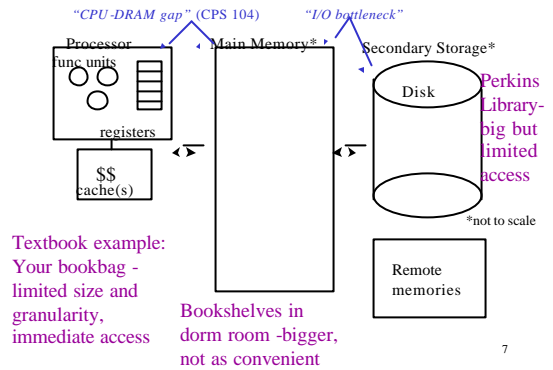
Executable to Address Space

- 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?



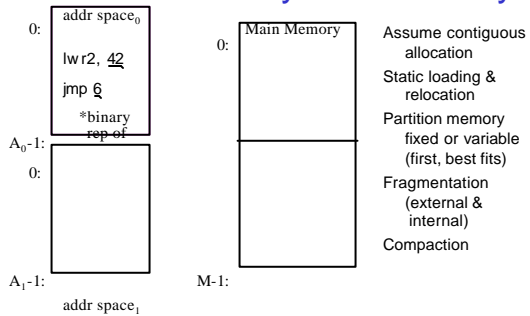
5

Memory Hierarchy



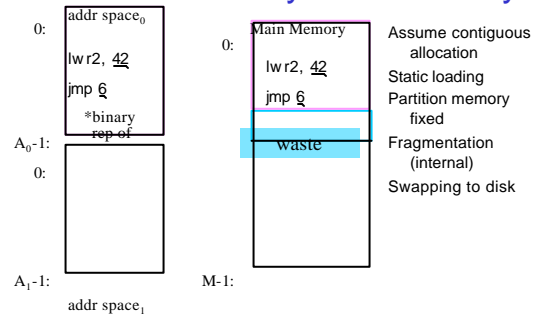
7

Allocation to Physical Memory



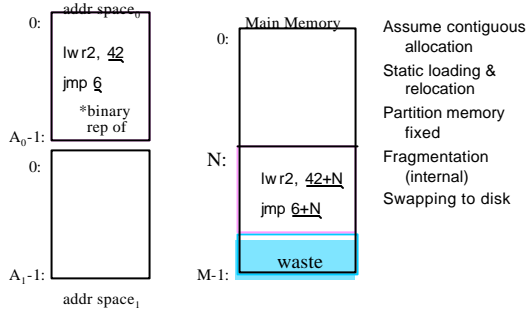
8

Allocation to Physical Memory



9

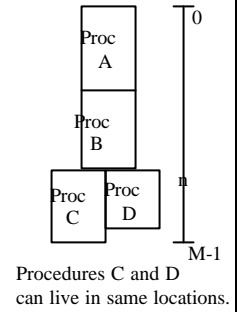
Allocation to Physical Memory



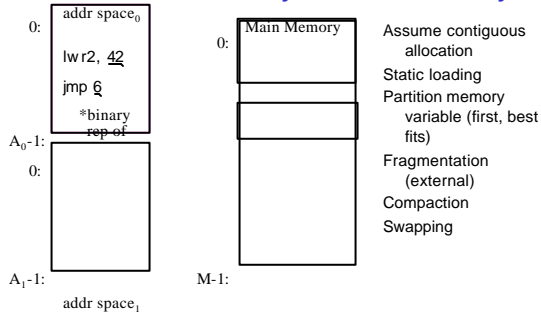
10

Overlays

- Explicit (user controlled) movement between levels of the memory hierarchy.
- Explicit calls to load regions of the address space into physical memory.
- Based on identifying 2 areas in the address space that are not needed at the same time.

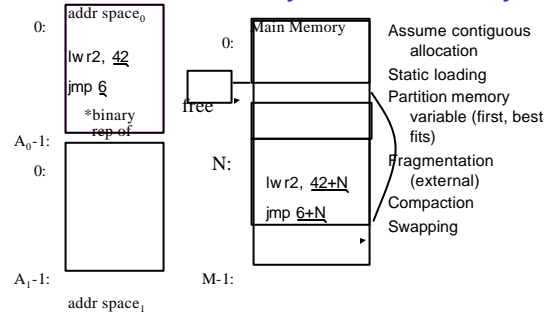


Allocation to Physical Memory



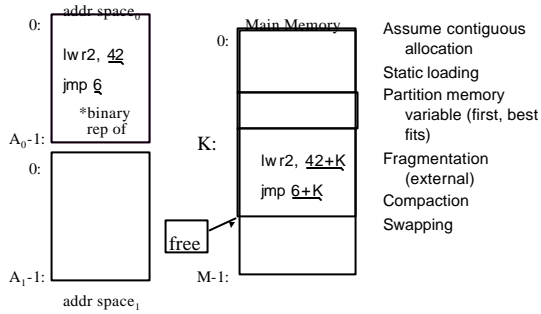
12

Allocation to Physical Memory



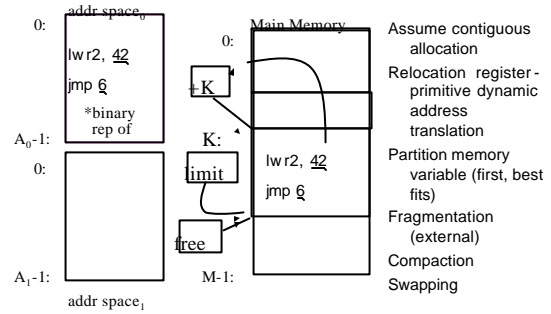
13

Allocation to Physical Memory



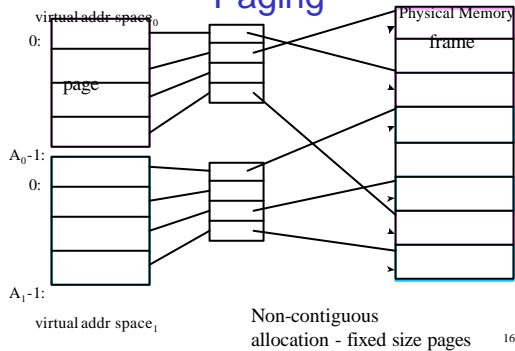
14

Allocation to Physical Memory



15

Paging



16

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).

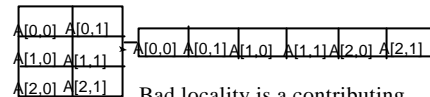
Locality

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

Assume:
arrays laid
out in rows



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

10/4/2001

19

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.

10/4/2001

21

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.

22

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.

23

Nachos Address Spaces

Remember this from Scheduler:Run code?

```

if (userprogram) {
    currentThread->SaveUserState();
    currentThread->space->SaveState();
}
    
```

24

A Page Table Entry (PTE)

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

25

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 **Memory Management Unit** 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.

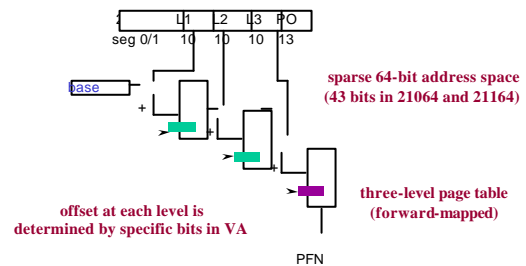
Memory Management Unit (MMU)

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

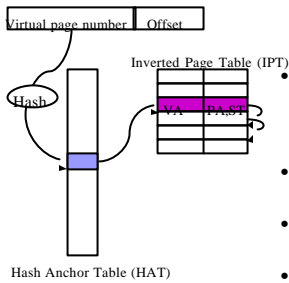
The OS Directs the MMU

- 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”.

Alpha Page Tables (Forward Mapped)



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

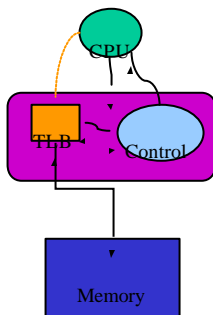
30

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

31

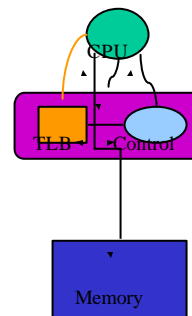
Hardware Managed TLBs



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

32

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?

33

Questions for Paged Virtual Memory

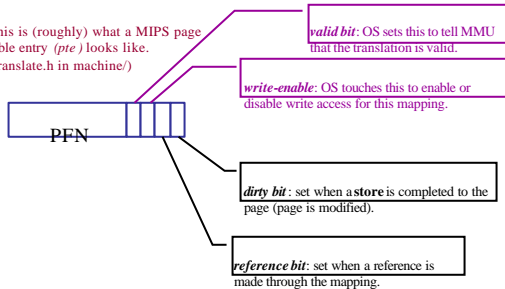
1. How do we prevent users from accessing protected data?
2. If a page is in memory, how do we find it?
Address translation must be fast.
3. If a page is not in memory, how do we find it?
4. When is a page brought into memory?
5. If a page is brought into memory, where do we put it?
6. If a page is evicted 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.

A Page Table Entry (PTE)

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



38

Care and Feeding of TLBs

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

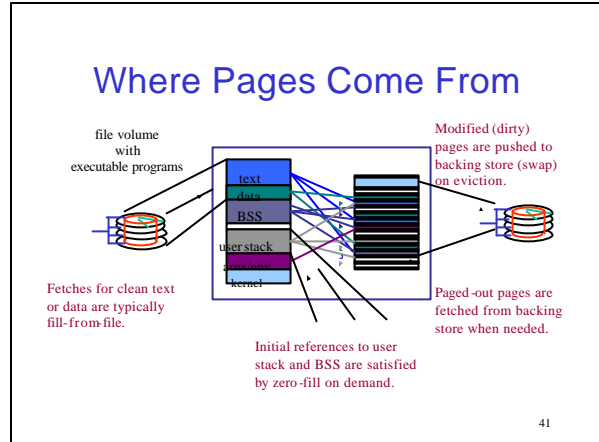
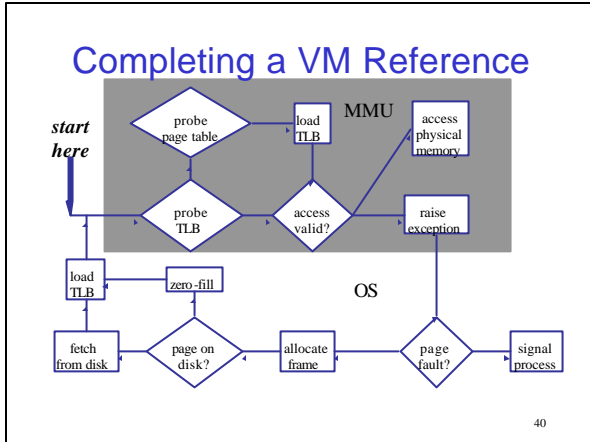
Choice 1: OS maintains page tables examined by the MMU.

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*).



- ### 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

10/4/2001

45

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

46

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)

10/4/2001

47

Placement Policy

Which free frame to choose?

Are all frames in physical memory created equal?

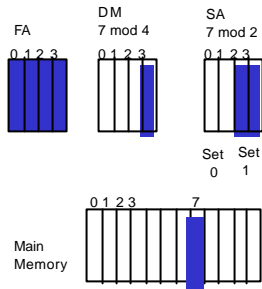
- Yes, only considering size. Fixed size.
- No, if considering
 - Cache performance, *conflict misses*
 - Access to *multi-bank* memories
 - Multiprocessors with distributed memories

48

Review: Cache Memory

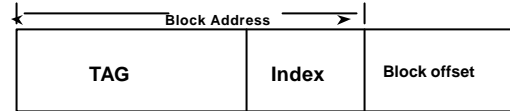
Block 7 placed in 4
block cache:

- Fully associative, direct mapped, 2-way set associative
- S.A. Mapping = Block Number Modulo Number Sets
- DM = 1-way Set Assoc



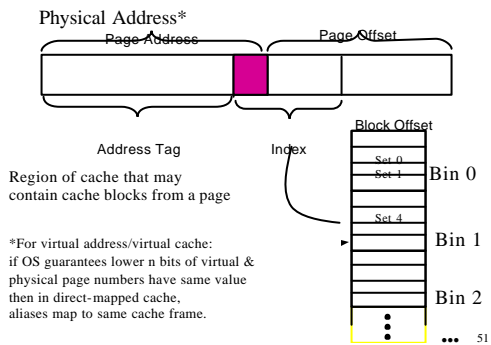
Cache Indexing

- **Tag on each cache block**
 - No need to check index or block offset
- **Increasing associativity shrinks index, expands tag**



Fully Associative: No index
Direct-Mapped: Large index

Bins



Virtual Memory and Physically Indexed Caches

-
- Random vs careful mapping
 - Selection of physical page frame dictates cache index
 - Overall goal is to minimize cache misses (conflict)
 - inter- and intra-address space

Careful Page Mapping

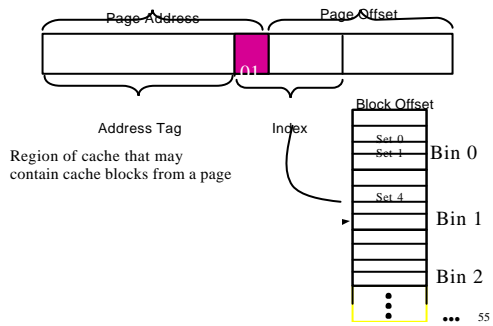
- Select a page frame such that cache conflict misses are reduced
 - only choose from *pool* of available page frames (no replacement induced)
- static
 - “smart” selection of page frame *at page fault time*
 - shown to reduce cache misses 10% to 20%
- dynamic
 - move pages around (copying from frame to frame)

53

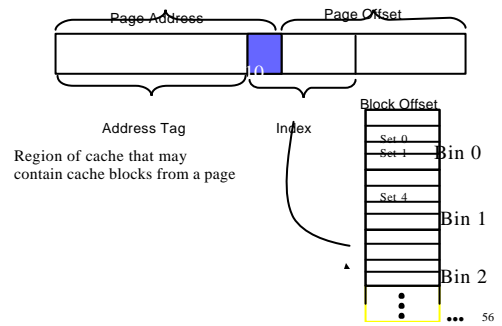
Page Coloring

- Make physical index match virtual index
- no conflicts for sequential pages
- Possibly many conflicts between processes
 - address spaces all have same structure (stack, code, heap)
 - modify to xor PID with address (MIPS used variant of this)
- Simple implementation
- Pick arbitrary page if necessary

54



55

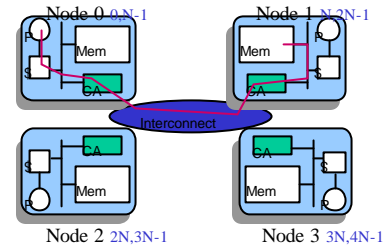


56

Bin Hopping

- Allocate sequentially mapped pages (time) to sequential bins (space)
- Can exploit temporal locality
 - pages mapped close in time will be accessed close in time
- Search from last allocated bin until bin with available page frame
 - Better fallback plan when small number in pool
- Separate search list per process
- Simple implementation

57



- Each node could be small scale MP
- Each node owns some of physical memory
- OS can allocate physical memory anywhere in system
- if remote: install mapping to remote data, migrate and install mapping to local data, or replicate and install mapping to local copy

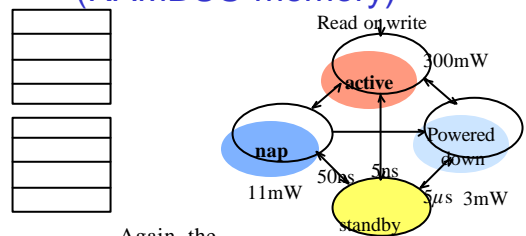
58

Hot Spots

Problem of creating a “hot spot” at one of the node memories - essentially analogous to the cache conflict miss problem motivating the page coloring and bin hopping ideas.
Reuse a good idea.

59

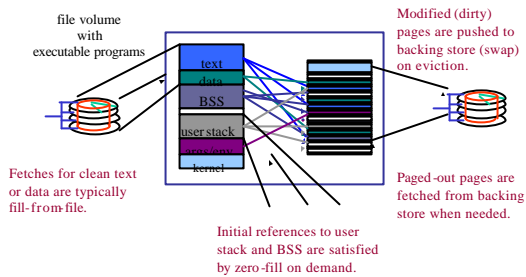
Power Management (RAMBUS memory)



Again, the v.a. → p.a mapping could exploit page coloring ideas

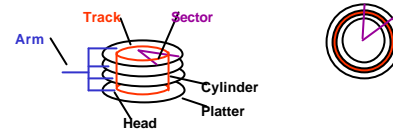
60

Backing Store = Disk



61

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

62

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

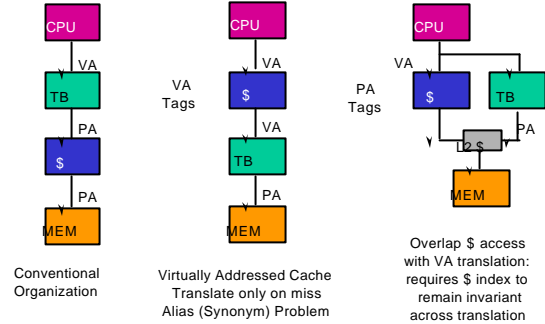
- Fragmentation
 - not *that* much spatial locality
 - large pages can waste storage
 - data must be contiguous within page

64

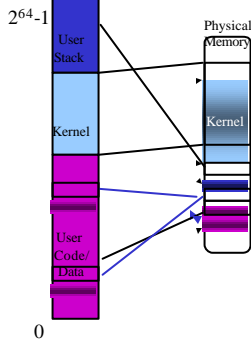
Another problem: Sharing

- The indirection of the mapping mechanism in paging makes it tempting to consider sharing code or data - having page tables of two processes map to the same page, but...
- Interaction with caches, especially if virtually addressed cache.
- What if there are addresses embedded inside the page to be shared?

TLBs and Caches



Aliases and Virtual Caches

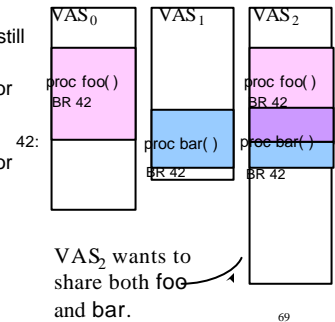


- *aliases* (sometimes called *synonyms*); Two different virtual addresses map to same physical address
- But, but... the virtual address is used to index the virtual cache
- Could have data in two different locations in the cache

67

Paging and Sharing (difficulties with embedded addresses)

- Virtual address spaces still look contiguous.
- Virtual Address Space for Process 0 links foo into pink address region
- Virtual Address Space for Process 1 links bar into blue region
- Then along comes Process 2 ...

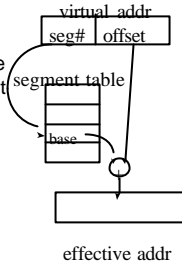


10/4/2001

69

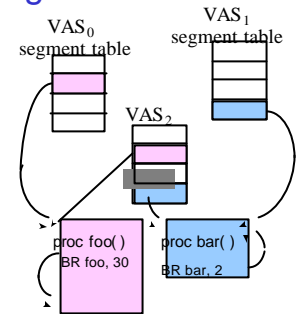
Segmentation

- A better basis for sharing. Naming is by *logical unit* (rather than arbitrary fixed size unit) and then offset within unit (e.g. procedure).
- Segments are variable size
- Segment table is like a bunch of base/limit registers.



Sharing in Segmentation

- Naming is by logical objects.
- Offsets are relative to base of object.
- Address spaces may be sparse as well as being non-contiguous.



10/4/2001

71

Combining Segmentation and Paging

- Sharing supported by segmentation. Programs name shared segments.
- Physical storage management simplified by paging of each segment.

