Page Tables Revisited

Learning Outcomes

- An understanding of virtual linear array page tables, and their use on the MIPS R3000.
- Exposure to alternative page table structures beyond two-level and inverted page tables.

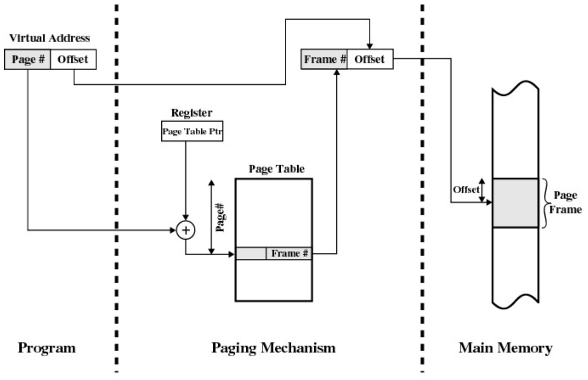
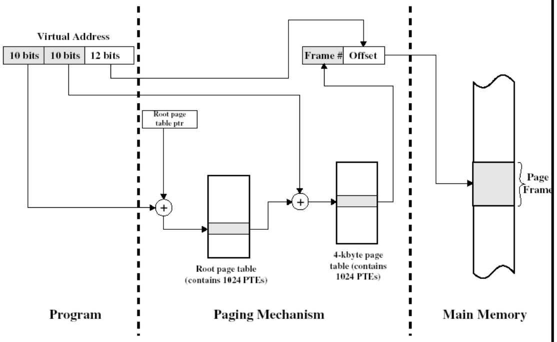


Figure 8.3 Address Translation in a Paging System

Two-level Translation



R3000 TLB Refill

- Can be optimised for TLB refill only
  - Does not need to check the exception type
  - Does not need to save any registers
    - It uses a specialised assembly routine that only uses k0 and k1.
  - Does not check if PTE exists
    - Assumes virtual linear array – see extended OS notes
- With careful data structure choice, exception handler can be made very fast

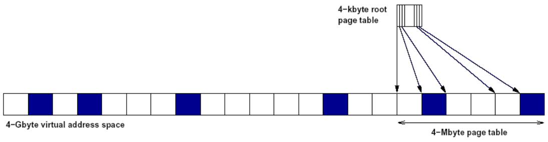
- An example routine

```
mfc0 k1,C0_CONTEXT
mfc0 k0,C0_EPC # mfc0 delay
               #  slot
lw k1,0(k1) #  may double
   # fault (k  = orig EPC)
nop
mtc0 k1,C0_E   YLO
nop
tlbwr
jr k0
rfe
```

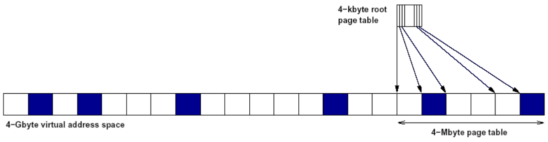
How does this work?

Virtual Linear Array page table

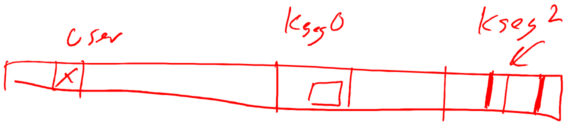
- Assume a 2-level PT
- Assume 2nd-level PT nodes are in virtual memory
- Assume all 2nd-level nodes are allocated contiguously ⇒ 2nd-level nodes form a contiguous array indexed by page number



Virtual Linear Array Operation

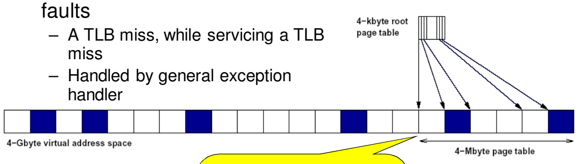


- Index into 2nd level page table *without* referring to root PT!
- Simply use the full page number as the PT index!
- Leave unused parts of PT unmapped!
- If access is attempted to unmapped part of PT, a *secondary page fault* is triggered
  - This will load the mapping for the PT from the root PT
  - Root PT is kept in physical memory (cannot trigger page faults)



Virtual Linear Array Page Table

- Use Context register to simply load PTE by indexing a PTE array in virtual memory
- Occasionally, will get double faults
  - A TLB miss, while servicing a TLB miss
  - Handled by general exception handler



PTEbase in virtual memory in kseg2
• Protected from user access

---

c0 Context Register

31 … 21	20 … 2	1 … 0
PTEBase	Bad VPN	0

- c0_Context = PTEBase + 4 * PageNumber
  - PTEs are 4 bytes
  - PTEBase is the base local of the page table array (note: aligned on 4 MB boundary)
  - PTEBase is (re)initialised by the OS whenever the page table array is changed
    - E.g on a context switch
  - After an exception, c0_Context contains the address of the PTE required to refill the TLB.

Code for VLA TLB refill handler

```
mfc0 k1,C0_CONTEXT
mfc0 k0,C0_EPC      # mfc0 delay slot
lw k1,0(k1)         # may double fault
                    # (k0 = orig EPC)
nop
mtc0 k1,C0_ENTRYLO
nop
tlbwr
jr k0
rfe
```

Load PTE address from context register

Load address of instruction to return to

Move the PTE into EntryLo.

Load the PTE.
Note: this load can cause a TLB refill miss itself, but this miss is handled by the general exception vector. The general exception vector has to understand this situation and deal with in appropriately

Write EntryLo into random TLB entry.

Return from the exception

Software-loaded TLB

- Pros
  - Can simplify hardware design
  - provide greater flexibility in page table structure
- Cons
  - typically have slower refill times than hardware managed TLBs.

Design Tradeoffs for Software-Managed TLBs

David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest Trevor Mudge & Richard Brown

ISCA '93 Proceedings of the 20th annual international symposium on computer architecture

---

Trends at the time

- Operating systems
  - – moving functionality into user processes
  - – making greater use of virtual memory for mapping data structures held within the kernel.
- RAM is increasing
  - – TLB capacity is relatively static
- Statement:
  - – Trends place greater stress upon the TLB by increasing miss rates and hence, decreasing overall system performance.
  - – True/False? How to evaluate?

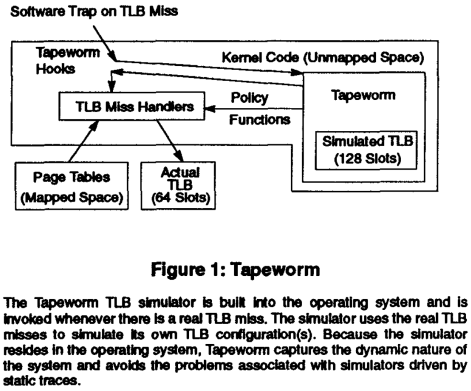


Figure 1: Tapeworm

The Tapeworm TLB simulator is built into the operating system and is invoked whenever there is a real TLB miss. The simulator uses the real TLB misses to simulate its own TLB configuration(s). Because the simulator resides in the operating system, Tapeworm captures the dynamic nature of the system and avoids the problems associated with simulators driven by static traces.

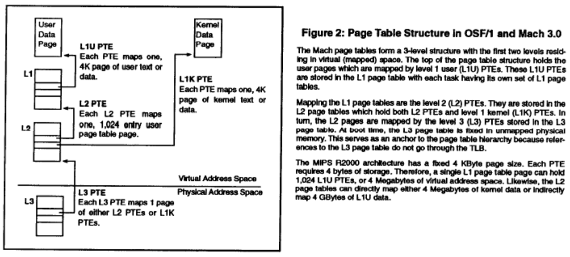
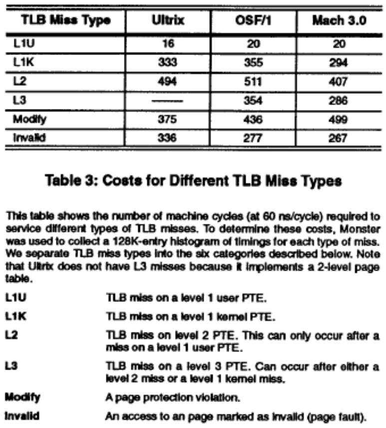


Figure 2: Page Table Structure in OSF/1 and Mach 3.0

The Mach page tables form a 3-level structure with the first two levels residing in virtual (mapped) space. The top of the page table structure holds the user pages which are mapped by level 1 user (L1U) PTEs. These L1U PTEs are stored in the L1 page table with each task having its own set of L1 page tables.

Mapping the L1 page tables are the level 2 (L2) PTEs. They are stored in the L2 page tables which hold both L2 PTEs and level 1 kernel (L1K) PTEs. In turn, the L2 pages are mapped by the level 3 (L3) PTEs stored in the L3 page table. At boot time, the L3 page table is fixed in unmapped physical memory. This serves as an anchor to the page table hierarchy because references to the L3 page table do not go through the TLB.

The MIPS R2000 architecture has a fixed 4 KByte page size. Each PTE requires 4 bytes of storage. Therefore, a single L1 page table page can hold 1,024 L1U PTEs, or 4 Megabytes of virtual address space. Likewise, the L2 page tables can directly map either 4 Megabytes of kernel data or indirectly map 4 GBytes of L1U data.



TLB Miss Type	Ultrix	OSF/1	Mach 3.0
L1U	16	20	20
L1K	333	355	294
L2	494	511	407
L3	——	354	286
Modify	375	436	499
Invalid	336	277	267

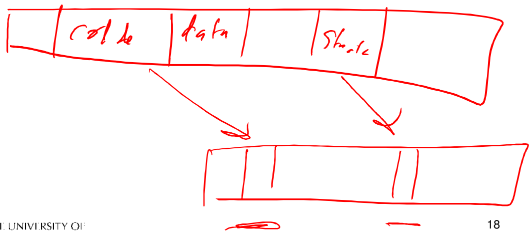
Table 3: Costs for Different TLB Miss Types

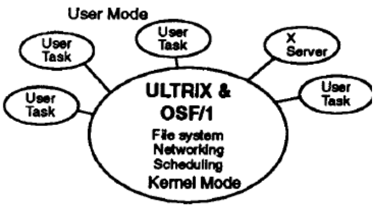
This table shows the number of machine cycles (at 60 ns/cycle) required to service different types of TLB misses. To determine these costs, Monster was used to collect a 128K-entry histogram of timings for each type of miss. We separate TLB miss types into the six categories described below. Note that Ultrix does not have L3 misses because it implements a 2-level page table.

L1U	TLB miss on a level 1 user PTE.
L1K	TLB miss on a level 1 kernel PTE.
L2	TLB miss on level 2 PTE. This can only occur after a miss on a level 1 user PTE.
L3	TLB miss on a level 3 PTE. Can occur after either a level 2 miss or a level 1 kernel miss.
Modify	A page protection violation.
Invalid	An access to an page marked as invalid (page fault).

Note the TLB miss costs

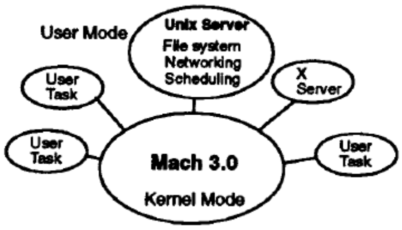
- What is expected to be the common case?





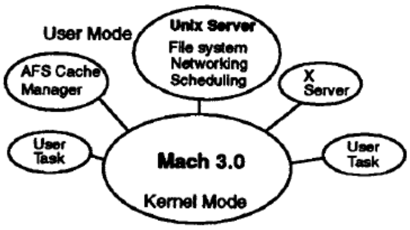
ULTRIX & OSF/1

File system, networking, scheduling and Unix interface reside inside a monolithic kernel. Kernel text resides in unmapped space. Ultrix places most kernel data structures in unmapped space while OSF/1 uses mapped space for many of its kernel data structures.



Mach 3.0

File system, networking, and Unix interface reside inside the monolithic Unix Server. Kernel text and some data reside in unmapped virtual space but the Unix Server is in mapped user space.



Mach 3.0 + AFSout

Same as standard Mach 3.0, but with increased functionality provided by a server task. The AFS Cache Manager is either inside the Unix Server or in its own, user-level server (as pictured above).

Measurement Results

System	Total Run Time (sec)	L1U	L1K	L2	L3	Invalid	Modify	Total
Ultrix	583	9,021,420	135,847	3,828	———	16,191	115	9,177,401
OSF/1	892	9,817,502	1,509,973	34,972	207,163	79,299	42,490	11,691,398
Mach3	975	21,466,165	1,682,722	352,713	556,264	165,849	125,409	24,349,121
Mach3+AFSIn	1,371	30,123,212	2,493,283	330,803	690,441	168,429	127,245	33,933,413
Mach3+AFSOut	1,517	31,611,047	2,712,979	1,042,527	987,648	168,128	127,505	36,649,834

Table 5: Number of TLB Misses

System	Total TLB Service Time (sec)	L1U	L1K	L2	L3	Invalid	Modify	% of Total Run Time
Ultrix	11.82	8.66	2.71	0.11	———	0.33	0.00	2.03%
OSF/1	51.85	11.78	32.16	1.07	4.40	1.32	1.11	5.81%
Mach3	80.01	25.76	29.68	8.61	9.55	2.66	3.75	8.21%
Mach3+AFSIn	106.56	36.15	43.98	8.08	11.85	2.70	3.81	7.77%
Mach3+AFSOut	134.71	37.93	47.86	25.46	16.95	2.69	3.82	8.88%

Table 6: Time Spent Handling TLB Misses

These tables show the number of TLB misses and amount of time spent handling TLB misses for each of the operating systems studied. In Ultrix, most of the TLB misses and TLB miss time is spent servicing L1U TLB misses. However, for OSF/1 and various versions of Mach 3.0, L1K and L2 misses can overshadow the L1U miss time. The increase in Modify misses is due to OSF/1 and Mach 3.0's use of protection to implement copy-on-write memory sharing.

Specialising the L2/L1K miss vector

Type of PTE Miss	Counts	Previous Total Cost from Table 6 (sec)	New Total Cost (sec)	Time Saved (sec)
Mach3+AFSIn				
L1U	30,123,212	36.15	36.15	0.00
L2	330,803	8.08	0.79	7.29
L1K	2,493,283	43.98	2.99	40.99
L3	690,441	11.85	11.85	0.00
Modify	127,245	3.81	3.81	0.00
Invalid	168,429	2.70	2.70	0.00
Total	33,933,413	106.56	58.29	48.28

Table 7: Recomputed Cost of TLB Misses Given Additional Miss Vectors (Mach 3.0)

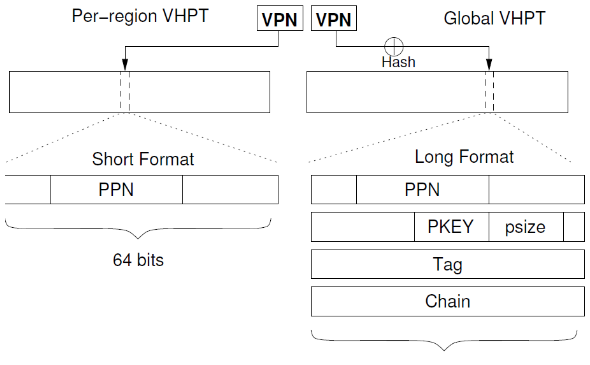
Supplying a separate interrupt vector for L2 misses and allowing the uTLB handler to service L1K misses reduces their cost to 40 and 20 cycles, respectively. Their contribution to TLB miss time drops from 8.08 and 43.98 seconds down to 0.79 and 2.99 seconds, respectively.

Other performance improvements?

- In Paper
  - Pinned slots
  - Increased TLB size
  - TLB associativity
- Other options
  - Bigger page sizes
  - Multiple page sizes

Itanium Page Table

- Takes a bet each way
- Loading
  - software
  - two different format hardware walkers
- Page table
  - software defined
  - Virtual linear array
  - Hashed



what about the P4?

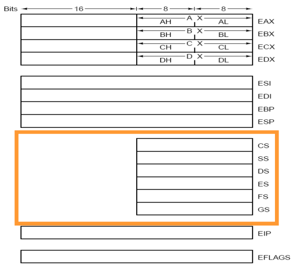
- i.e. 32-bit x86 architecture.

P4

- Sophisticated, supports:
  - demand paging
  - pure segmentation
  - segmentation with paging
- Heart of the VM architecture
  - Local Descriptor Table (LDT)
  - Global Descriptor Table (GDT)
- LDT
  - 1 per process
  - describes segments local to each process (code, stack, data, etc.)
- GDT
  - shared by all programs
  - describes system segments (including OS itself)

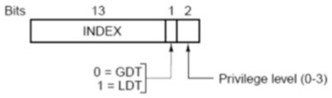
P4

- To access a segment P4
  - loads a selector in 1 of the segment registers
  - …



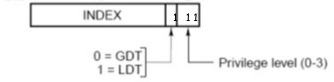
P4

- a P4 selector:



P4

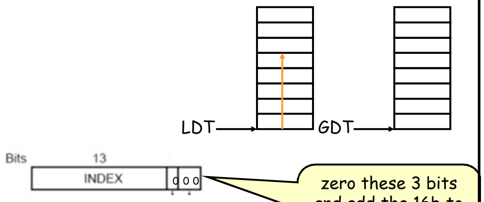
- a P4 selector:



determine LDT or GDT (and privilege level)

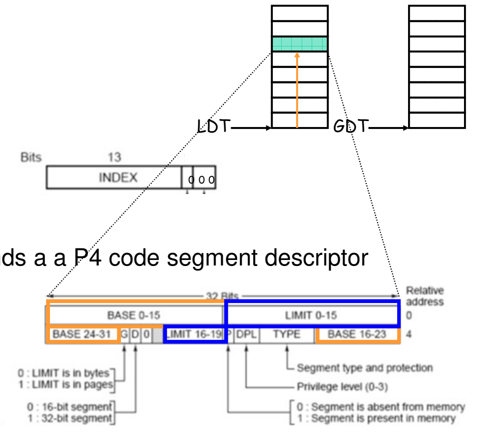
- when selector is in register, corresponding segment descriptor is
  - fetched by MMU
  - loaded in internal MMU registers
- Next, segment descriptor is used to handle memory reference (discussed later)

P4



zero these 3 bits and add the 16b to base address of LDT or GDT

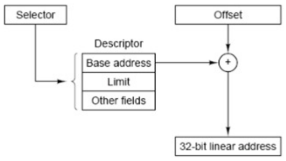
P4



- finds a a P4 code segment descriptor

P4

- calculating a linear address from selector+offset



P4

IF no paging used: we are done

➔ this is the physical address

ELSE

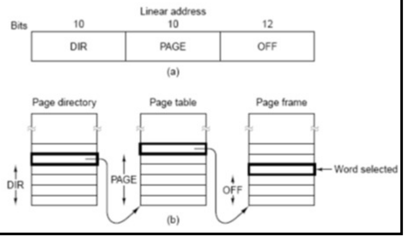
➔ linear address interpreted as virtual address

➔ paging again!

P4 with paging

- every process has page directory
  - 1024 32bit entries
  - each entry points to page table
  - page table contains 1024 32bit entries
  - each entry points to page frame

mapping linear address to physical address with paging



P4

- Many OSs:
  - –BASE=0
  - –LIMIT=MAX
- ➔ no segmentation at all

That is it!