P6 (PentiumPro, II, III, Celeron) memory system

- 32 bit address space
- 4 KB page size
- L1, L2, and TLBs
  - 4-way set associative
- inst TLB
  - 32 entries
  - 8 sets
- data TLB
  - 64 entries
  - 16 sets
- L1 i-cache and d-cache
  - 16 KB
  - 32 B line size
  - 128 sets
- L2 cache
  - unified
  - 128 KB -- 2 MB

processor package
Overview of P6 memory read

CPU

virtual address (VA)

VPN
VPO

TLBT TLBI

TLB (16 sets, 4 entries/set)

VPN1 VPN2

TLB miss

PDE
PTE

Page tables

L1 hit

L1 (128 sets, 4 lines/set)

L1 miss

physical address (PA)

32 result

L2 and DRAM
P6 2-level page table structure

Page directory
- 1024 4-byte page directory entries (PDEs) that point to page tables
- one page directory per process.
- page directory must be in memory when its process is running
- always pointed to by PDBR

Page tables:
- 1024 4-byte page table entries (PTEs) that point to pages.
- page tables can be paged in and out.
# P6 page directory entry (PDE)

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page table physical base addr</td>
<td>Avail</td>
<td>G</td>
</tr>
</tbody>
</table>

**Page table physical base address**: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

- **Avail**: available for system programmers
- **G**: global page (don’t evict from TLB on task switch)
- **PS**: page size 4K (0) or 4M (1)
- **A**: accessed (set by MMU on reads and writes, cleared by software)
- **CD**: cache disabled (1) or enabled (0)
- **WT**: write-through or write-back cache policy for this page table
- **U/S**: user or supervisor mode access
- **R/W**: read-only or read-write access
- **P**: page table is present in memory (1) or not (0)

<table>
<thead>
<tr>
<th>31</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available for OS (page table location in secondary storage)</td>
<td>P=0</td>
</tr>
</tbody>
</table>
# P6 page table entry (PTE)

- **Page base address**: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)
- **Avail**: available for system programmers
- **G**: global page (don’t evict from TLB on task switch)
- **D**: dirty (set by MMU on writes)
- **A**: accessed (set by MMU on reads and writes)
- **CD**: cache disabled or enabled
- **WT**: write-through or write-back cache policy for this page
- **U/S**: user/supervisor
- **R/W**: read/write
- **P**: page is present in physical memory (1) or not (0)

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page physical base address</td>
<td>Avail</td>
<td>G 0 D A CD WT U/S R/W P=1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available for OS (page location in secondary storage)</td>
<td>P=0</td>
</tr>
</tbody>
</table>
How P6 page tables map virtual addresses to physical ones

Virtual address

word offset into page directory

word offset into page directory

word offset into page table

word offset into physical and virtual page

page directory

page table

physical address of page base (if P=1)

physical address of page table base (if P=1)

PDBR

physical address of page directory

PDE

PTE

PPN

PPO
Representation of Virtual Address Space

Simplified Example
- 16 page virtual address space

Flags
- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?
Common Case: TLB – No OS Involved

CPU

VPN

VPO

20

12

virtual address (VA)

TLB (16 sets, 4 entries/set)

TLB hit

TLB miss

L1 hit

L1 miss

L1 (128 sets, 4 lines/set)

Page tables

PDBR

PDE

PTE

PPN

PPO

physical address (PA)

32 result

L2 and DRAM

CT

Cl

CO

20

7

5
Uncommon Case: Not in TLB

CPU

virtual address (VA)

VPN1, VPN2

TLB (16 sets, 4 entries/set)

TLB hit

TLB miss

PDE, PTE

Page tables

Page directory (PDBR)

PPN, PPO

Physical address (PA)

L1 (128 sets, 4 lines/set)

L1 hit

L1 miss

L2 and DRAM

result

32

CT, CI, CO

CPU

VPN, VPO

TLB (16 sets, 4 entries/set)
Translating with the P6 page tables (case 1/1)

Case 1/1: page table and page present.

**MMU Action:**
- MMU build physical address and fetch data word.
- OS action
  - none

---

Mem

- VPN
- VPN1
- VPN2
- VPO
- PDE \( p=1 \)
- PTE \( p=1 \)
- Data page

Disk
Translating with the P6 page tables (case 1/0)

Case 1/0: page table present but page missing.

MMU Action:
- page fault exception
- OS’s handler receives the following args:
  - VA that caused fault
  - fault caused by non-present page or page-level protection violation
  - read/write
  - user/supervisor
Translating with the P6 page tables (case 1/0, cont)

OS Action:
- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk and copy to virtual page
- Restart faulting instruction by returning from exception handler.

Probably Later
Lets another process run while the disk is getting the page
Translating with the P6 page tables (case 0/1)

Case 0/1: page table missing but page present.

Introduces consistency issue.

- potentially every page out requires update of disk page table.

Linux disallows this

- if a page table is swapped out, then swap out its data pages too.
Translating with the P6 page tables (case 0/0)

Case 0/0: page table and page missing.

MMU Action:
- page fault exception
Translating with the P6 page tables  
(case 0/0, cont)

OS action:
• swap in page table.
• restart faulting instruction by returning from handler.

Like case 0/1 from here on.
Linux organizes VM as a collection of “areas” (Hardware Independent)

- **pgd:**
  - page directory address
- **vm_prot:**
  - read/write permissions for this area
- **vm_flags**
  - shared with other processes or private to this process
Linux page fault handling

Is the VA legal?
• i.e. is it in an area defined by a `vm_area_struct`?
  • if not then signal segmentation violation (e.g. (1)) (or extend stack)

Is the operation legal?
• i.e., can the process read/write this area?
  • if not then signal protection violation (e.g., (2))

If OK, handle fault
• e.g., (3)
  • Must also update page tables
Memory mapping

Creation of new VM area done via “memory mapping”

• create new vm_area_struct and page tables for area
• area can be backed by (i.e., get its initial values from) :
  – regular file on disk (e.g., an executable object file)
    » initial page bytes come from a section of a file
  – nothing (e.g., bss)
    » initial page bytes are zeros
• dirty pages are swapped back and forth between a special swap file.

Key point: no virtual pages are copied into physical memory until they are referenced!

• known as “demand paging”
• crucial for time and space efficiency
User-level memory mapping

void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

- map len bytes starting at offset offset of the file specified by file description fd, preferably at address start (usually 0 for don’t care).
  - File can be anonymous (all zeros, not actually stored, “demand zero”)
  - prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
  - flags: MAP_PRIVATE, MAP_SHARED, MAP_ANON
- return a pointer to the mapped area.

- Example: fast file copy
  - useful for applications like Web servers that need to quickly copy files.
  - mmap allows file transfers without copying into user space.

- Example: Sharing
  - Map same file into multiple addresses spaces
mmap() example: fast file copy

```c
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * mmap.c - a program that uses mmap
 * to copy itself to stdout
 */

int main() {
    struct stat stat;
    int i, fd, size;
    char *bufp;

    /* open the file and get its size*/
    fd = open("./mmap.c", O_RDONLY);
    fstat(fd, &stat);
    size = stat.st_size;

    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
                MAP_PRIVATE, fd, 0);

    /* write the VM area to stdout */
    write(1, bufp, size);
}
```
To run a new program \( p \) in the current process using `exec()`:  

- free `vm_area_struct`'s and page tables for old areas.  
- create new `vm_area_struct`'s and page tables for new areas.  
- stack, bss, data, text, shared libs.  
- text and data backed by ELF executable object file.  
- bss and stack initialized to zero.  
- set PC to entry point in `.text`  
- Linux will swap in code and data pages as needed.
Fork() revisited

To create a new process using fork:

• make copies of the old process’s mm_struct, vm_area_struct’s, and page tables.
  — at this point the two processes are sharing all of their pages.
  — How to get separate spaces without copying all the virtual pages from one space to another?
    » “copy on write” technique.

• copy-on-write
  — make pages of writeable areas read-only
  — flag vm_area_struct’s for these areas as private “copy-on-write”.
  — writes by either process to these pages will cause page faults.
    » fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

• Net result:
  — copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).
Dynamic Memory Allocation – beyond the stack and globals

• Stack
  • Easy to allocate (decrement esp)
  • Easy to deallocate (increment esp)
  • Automatic
  • Can pass values to called procedures, but not up to callers

• Global variables
  • Statically allocated
  • Have to decide at compile time how much space you need

• Allocation on the heap
  • Dynamically allocated
  • Independent of procedure calls
  • But must be carefully managed
    – Automatically: garbage collection
    – Manually: malloc/free or new/delete
Dynamic Memory Allocation

| Application | Dynamic Memory Allocator | Heap Memory |

Explicit vs. Implicit Memory Allocator

- **Explicit**: application allocates and frees space
  - E.g., `malloc` and `free` in C
- **Implicit**: application allocates, but does not free space
  - E.g. garbage collection in Java, ML or Lisp

Allocation

- In both cases the memory allocator provides an abstraction of memory as a set of blocks
- Doles out free memory blocks to application
Process memory image

- kernel virtual memory
- stack
- Memory mapped region for shared libraries
- run-time heap (via malloc)
- uninitialized data (.bss)
- initialized data (.data)
- program text (.text)

Allocators request additional heap memory from the operating system using the sbrk() function.

memory invisible to user code

%esp

the “brk” ptr
Malloc package

#include <stdlib.h>

void *malloc(size_t size)

• if successful:
  – returns a pointer to a memory block of at least size bytes, aligned to 8-byte boundary.
  – if size==0, returns NULL
• if unsuccessful: returns NULL

void free(void *p)

• returns the block pointed at by p to pool of available memory
• p must come from a previous call to malloc or realloc.

void *realloc(void *p, size_t size)

• changes size of block p and returns ptr to new block.
• contents of new block unchanged up to min of old and new size.
void foo(int n, int m) {
    int i, *p;

    /* allocate a block of n ints */
    if ((p = (int *) malloc(n * sizeof(int))) == NULL) {
        perror("malloc");
        exit(0);
    }
    for (i=0; i<n; i++)
        p[i] = i;

    /* add m bytes to end of p block */
    if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++)
        p[i] = i;

    /* print new array */
    for (i=0; i<n+m; i++)
        printf("%d\n", p[i]);

    free(p); /* return p to available memory pool */
}