CMSC 313 Lecture 13

- Project 4 Questions
- Reminder: Midterm Exam next Monday 10/20
- Virtual Memory

Project 4: C Functions

Due: Tue 10/14/03, Section 0101 (Chang) & Section 0301 (Macneil)
Fri 10/17/03, Section 0201 (Patel & Bourner)

Objective

The objective of this programming exercise is to practice writing assembly language programs that use the C function call conventions.

Assignment

Convert your assembly language program from Project 3 as follows:

1. Convert the program into one that follows the C function call convention, so it may be called from a C program. Your program should work with the following function prototype:
   The intention here is that the first parameter is a pointer to the records array and the second parameter has the number of items in that array.

   ```
   void report (void *, unsigned int) ;
   ```

   The intention here is that the first parameter is a pointer to the records array and the second parameter has the number of items in that array.

2. Modify your program so it uses the strncmp() function from the C library to compare the nicknames of two records. The function prototype of `strncmp()` is:

   ```
   int strncmp(const char *s1, const char *s2, size_t n) ;
   ```

   The function returns an integer less than, equal to, or greater than zero if s1 (or the first n bytes thereof) is found, respectively, to be less than, to match, or be greater than s2.

3. Modify your program so that it prints out the entire record (not just the `realname` field) of the record with the least number of points and the record with the alphabetically first nickname. You must use the `printf()` function from the C library to produce this output. The output of your program would look something like:

   ```
   Lowest Points: James Pressman (jamieboy)
     Alignment: Lawful Neutral
     Role: Fighter
     Points: 57
     Level: 1
   First Nickname: Dan Gannett (danmeister)
     Alignment: True Neutral
     Role: Ranger
     Points: 7502
     Level: 3
   ```

A sample C program that should work with your assembly language implementation of the `report()` function is available on the GL file system: `/afs/umbc.edu/users/c/h/chang/pub/cs313/records2.c`

Implementation Notes

- Documentation for the printf() and strncmp() functions are available on the Unix system by typing `man -S 3 printf` and `man -S 3 strncmp`.
- Note that the strncmp() function takes 3 parameters, not 2. It is good programming practice to use `strncmp()` instead of `strcmp()` since this prevents runaway loops if the strings are not properly null terminated. The third argument should be 16, the length of the `nickname` field.

- As in Project 3, you must also make your own test cases. The example in `records2.c` does not fully exercise your program. As before, your program will be graded based upon other test cases. If you have good examples in Project 3, you can just reuse those.
- Use `gcc` to link and load your assembly language program with the C program. This way, `gcc` will call `ld` with the appropriate options:

```
nasm -f elf report2.asm
gcc records2.c report2.o
```

- Notes on the C function call conventions are available on the web:

```
http://www.csee.umbc.edu/~chang/cs313.f03/stack.shtml
```

- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit at least 4 files: your assembly language program, at least 2 of your own test cases and a typescript file of sample runs of your program. The class name for submit is `cs313_0101`, `cs313_0102` or `cs313_0103` for respectively sections 0101 (Chang), 0201 (Patel & Bourner) or 0301 (Macneil). The name of the assignment name is `proj4`. The UNIX command to do this should look something like:

```
submit cs313_0103 proj4 report2.asm myrec1.c myrec2.c typescript
```

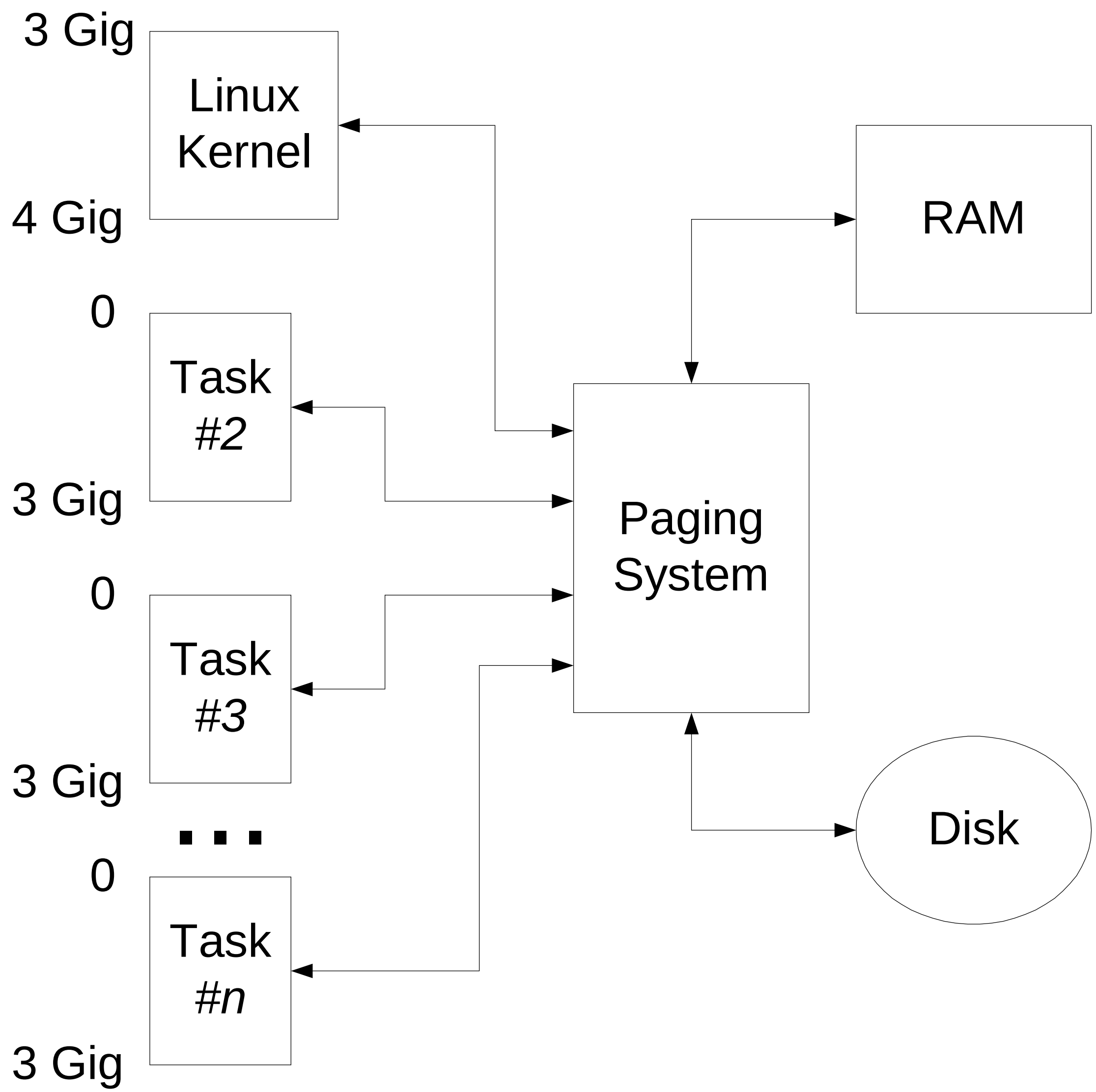
Last Time

- Linux/gcc/i386 Function Call Convention
- Now we know where our C programs store their data, right???

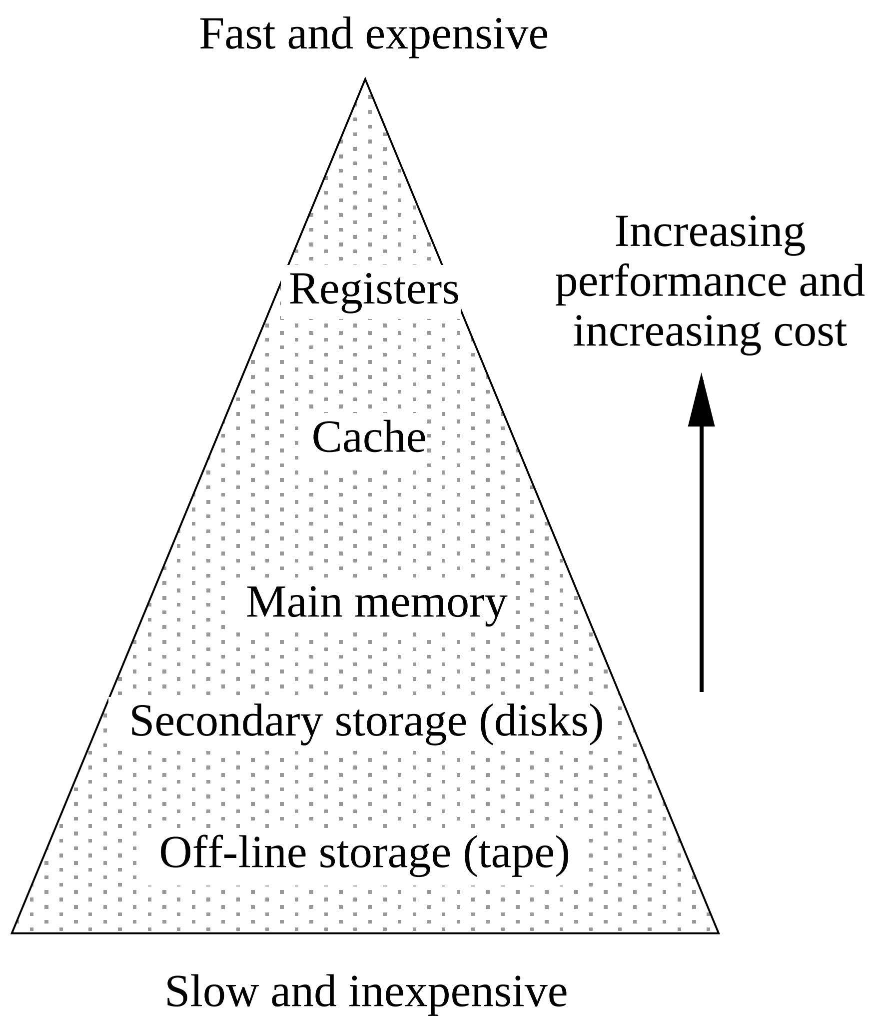
```
int global ;

int main() {

   int *ptr, n ;

   printf ("Address of main:   %08x\n", &main ) ;
   printf ("Address of global variable:  %08x\n", &global ) ;
   printf ("Address of local variable:   %08x\n", &n ) ;

   ptr = (int *) malloc(4) ;
   printf ("Address of allocated memory: %08x\n", ptr) ;
}
```

Linux Virtual Memory Space

- Linux reserves 1 Gig memory in the virtual address space
- The size of the Linux kernel significantly affects its performance (swapping is expensive)
- Linux kernel can be customized by including only relevant modules
- Designating kernel space facilitates protection of
- The portion of disk used for paging is called the swap space

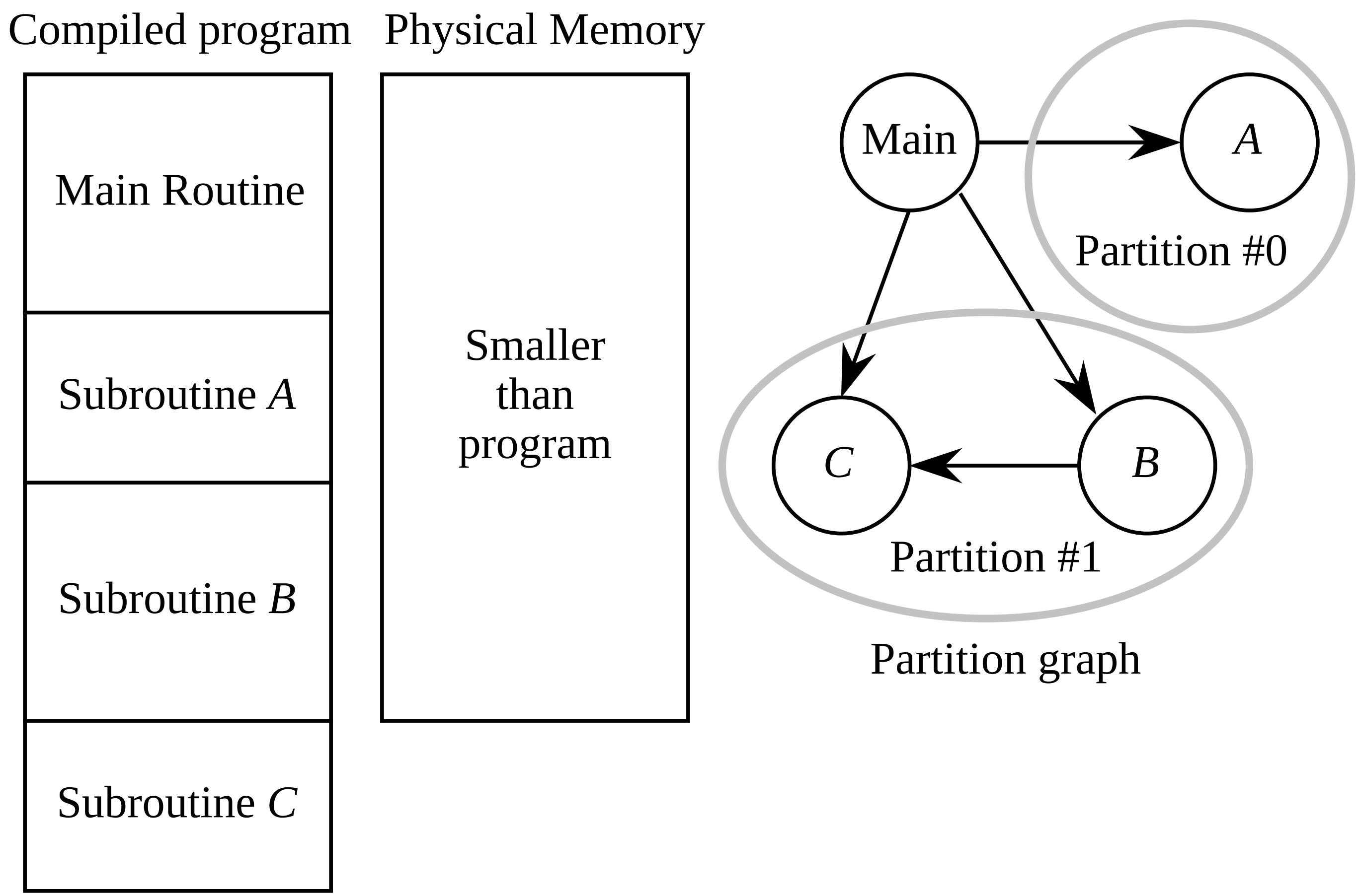


The Memory Hierarchy



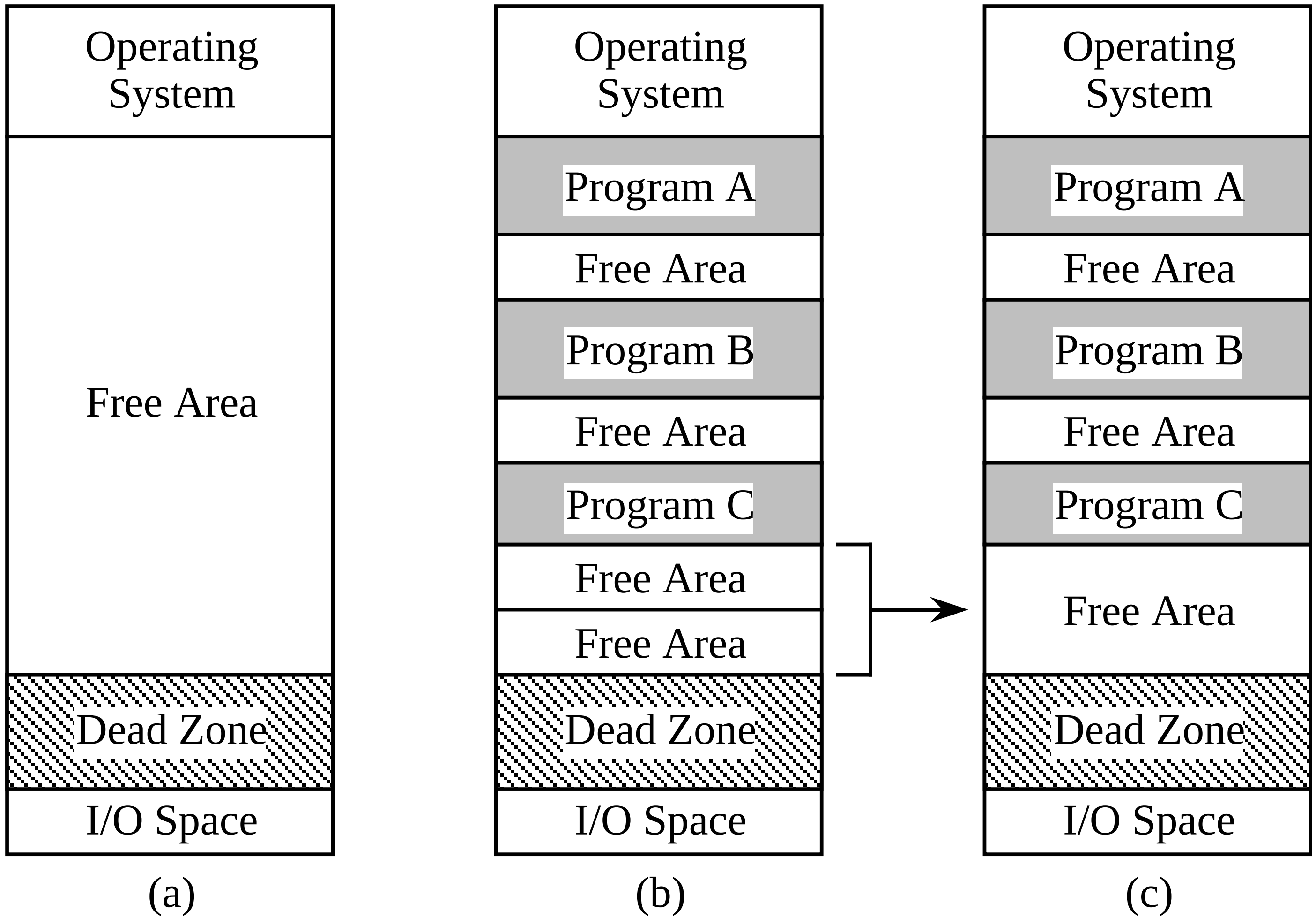
Overlays

- **A partition graph for a program with a main routine and three subroutines:**



Fragmentation

- **(a) Free area of memory after initialization; (b) after fragmentation; (c) after coalescing.**

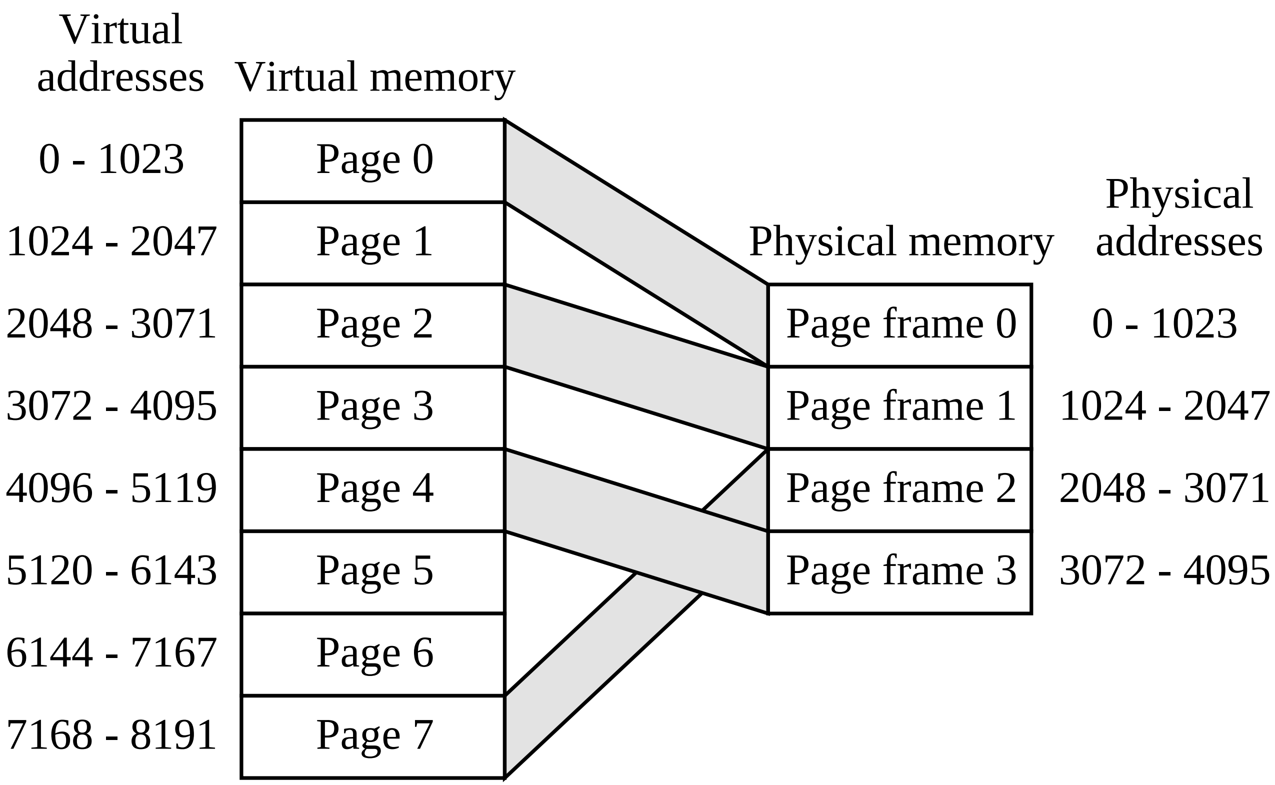


Memory Protection

- **Prevents one process from reading from or writing to memory used by another process**
- **Privacy in a multiple user environments**
- **Operating system stability**
  - **Prevents user processes (applications) from altering memory used by the operating system**
  - **One application crashing does not cause the entire OS to crash**

Virtual Memory

- Virtual memory is stored in a hard disk image. The physical memory holds a small number of virtual *pages* in physical *page frames*.
- A mapping between a virtual and a physical memory:



Page Table

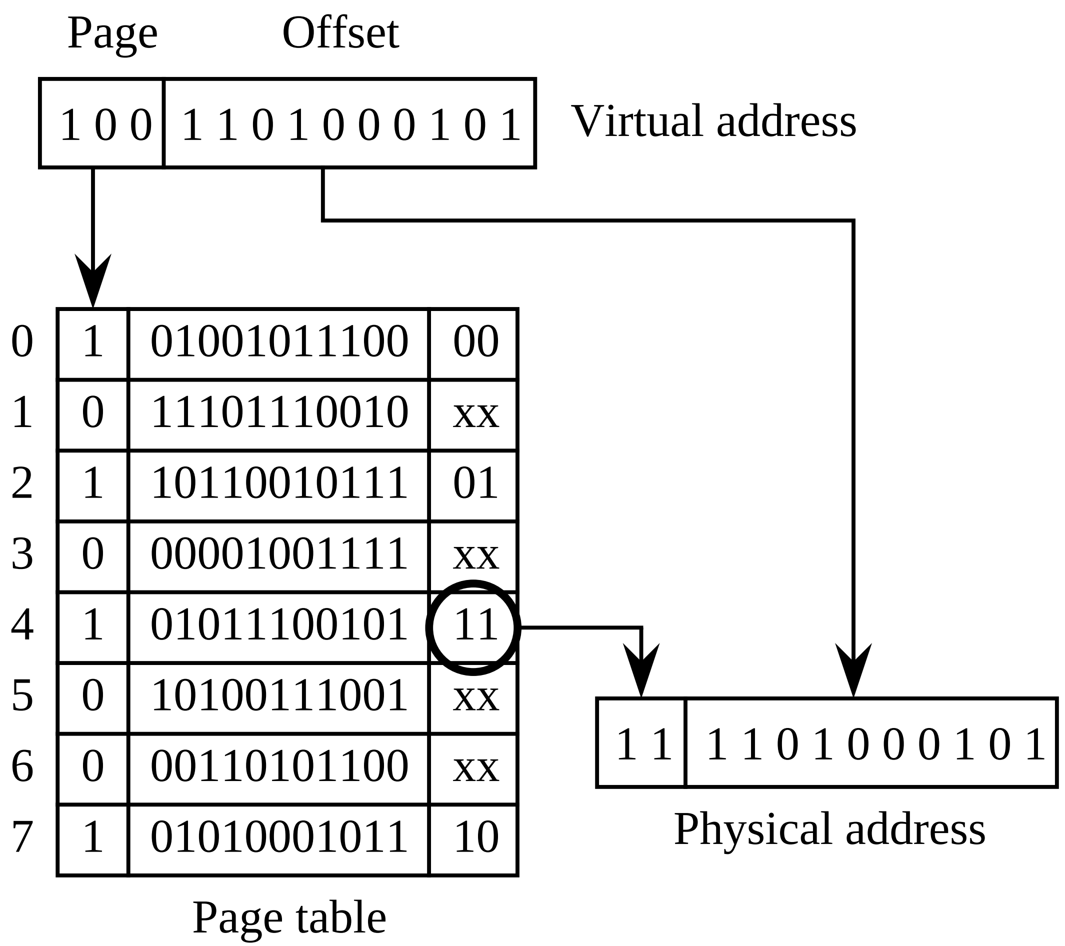
- **The page table maps between virtual memory and physical memory.**

Page #	Present bit	Disk address	Page frame
0	1	01001011100	00
1	0	11101110010	xx
2	1	10110010111	01
3	0	00001001111	xx
4	1	01011100101	11
5	0	10100111001	xx
6	0	00110101100	xx
7	1	01010001011	10

Present bit:
0: Page is not in physical memory
1: Page is in physical memory

Using the Page Table

- **A virtual address is translated into a physical address:**



Using the Page Table (cont')

- The configuration of a page table changes as a program executes.
- Initially, the page table is empty. In the final configuration, four pages are in physical memory.

After fault on page #1

0	0	01001011100	xx
1	1	11101110010	00
2	0	10110010111	xx
3	0	00001001111	xx
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

After fault on page #2

0	0	01001011100	xx
1	1	11101110010	00
2	1	10110010111	01
3	0	00001001111	xx
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

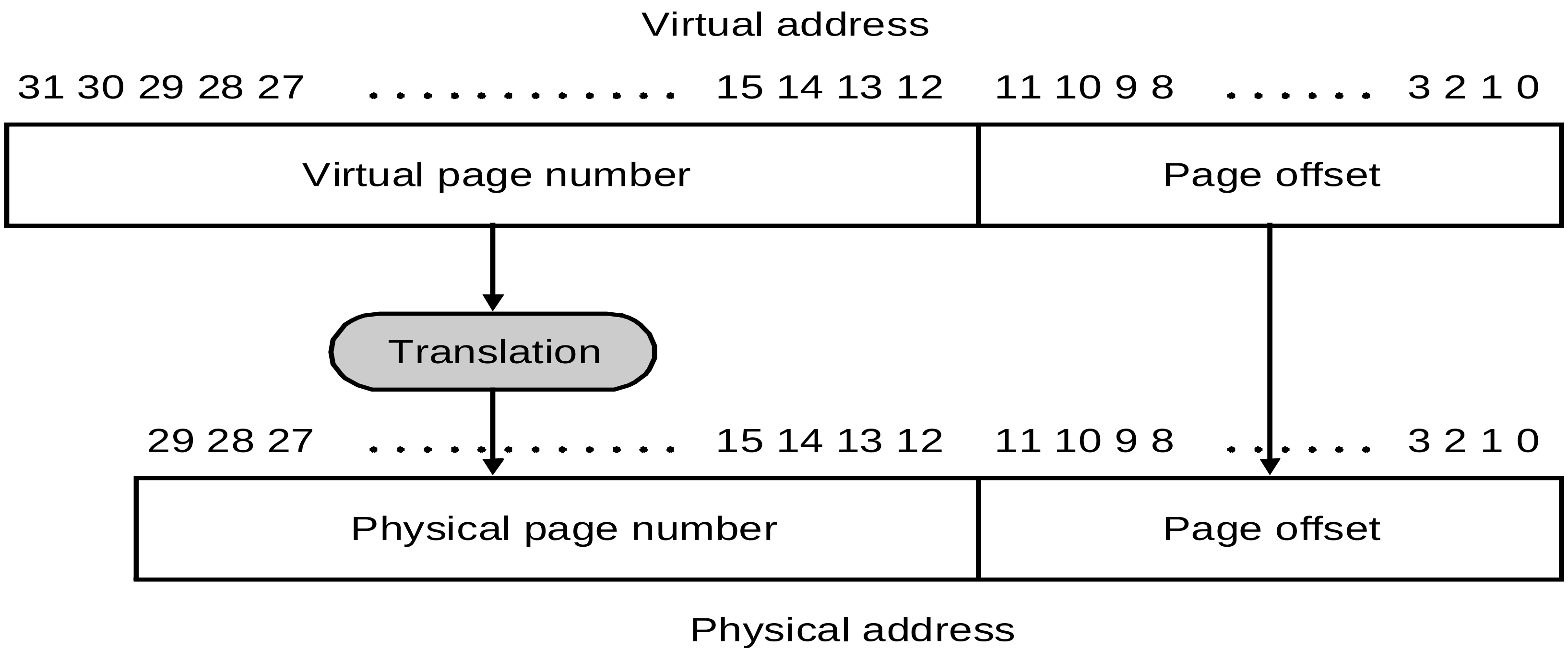
After fault on page #3

0	0	01001011100	xx
1	1	11101110010	00
2	1	10110010111	01
3	1	00001001111	10
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

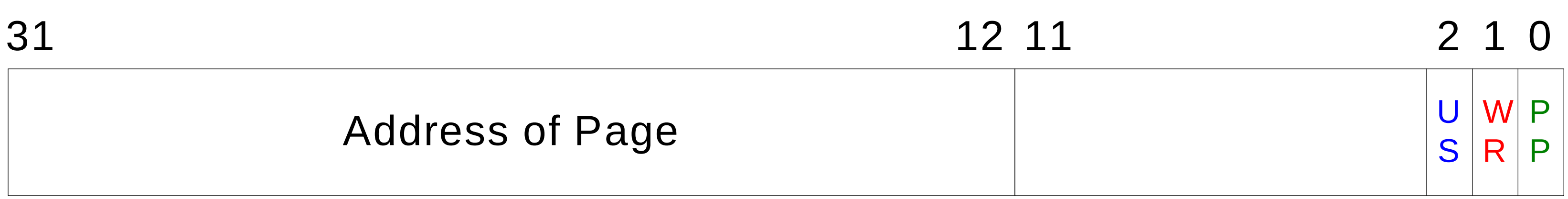
Final

0	0	01001011100	xx
1	0	11101110010	xx
2	1	10110010111	01
3	1	00001001111	10
4	1	01011100101	11
5	1	10100111001	00
6	0	00110101100	xx
7	0	01010001011	xx

Virtual Addressing

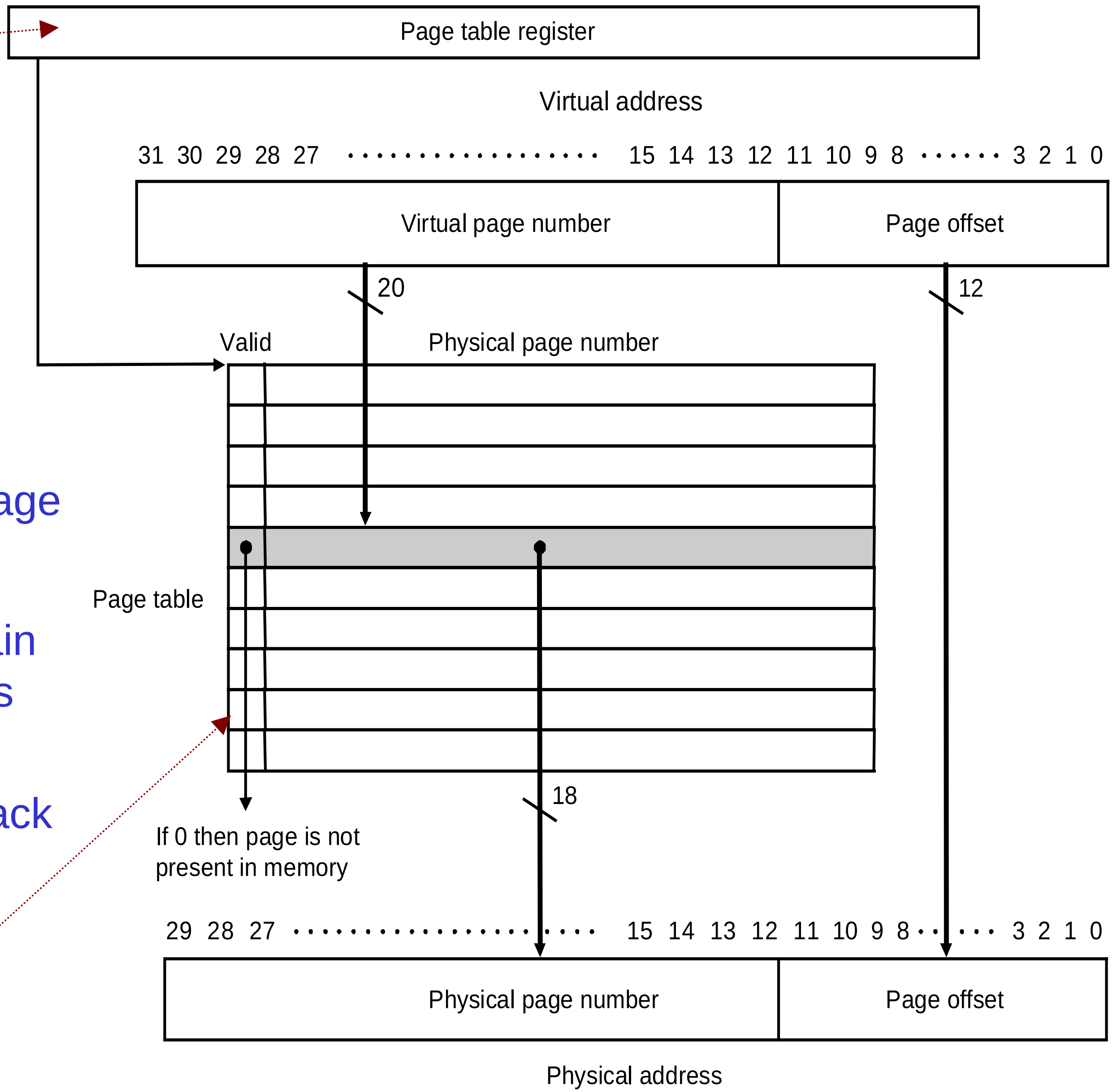


- Page faults are costly and take millions of cycles to process (disks are slow)
- 80386 Page attributes:
  - RW: read and write permission
  - US: User mode or kernel mode only access
  - PP: present bit to indicate where the page is



Page Table

Hardware supported

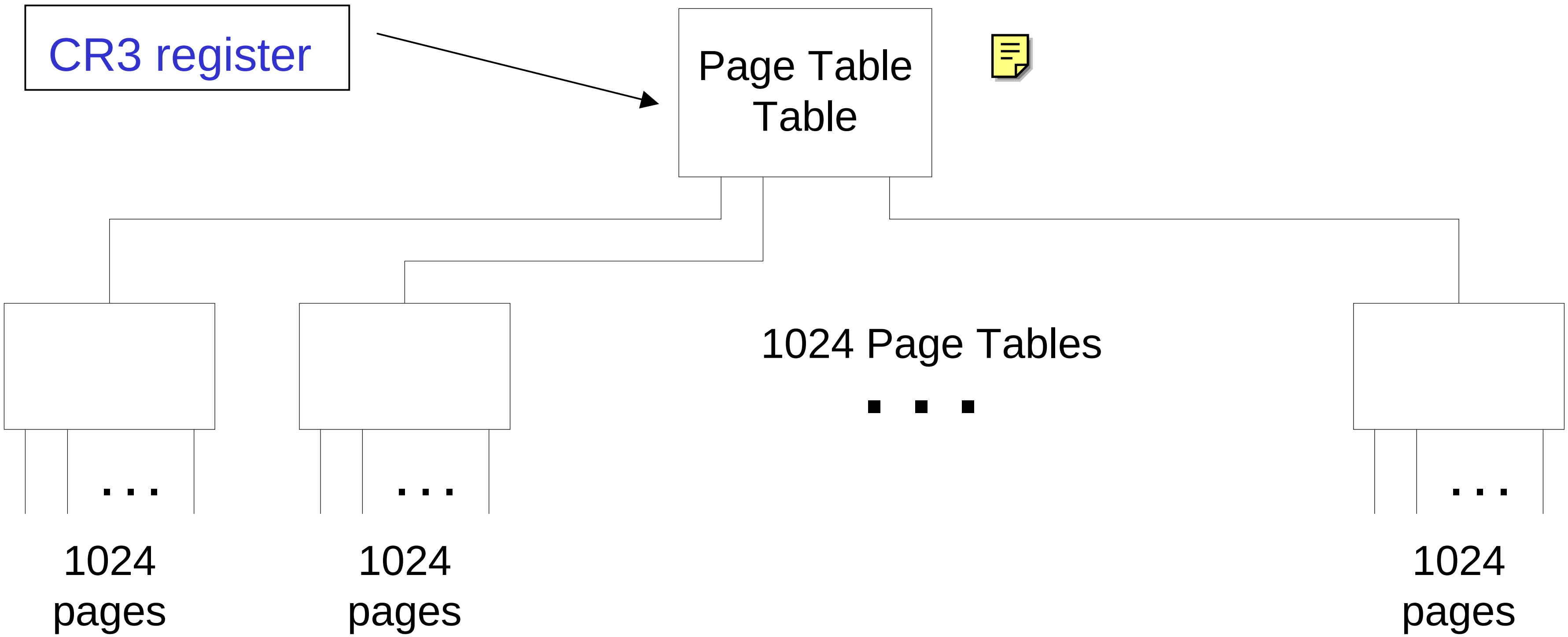


Page table:

- ★ Resides in main memory
- ★ One entry per virtual page
- ★ No tag is requires since it covers all virtual pages
- ★ Point directly to physical page
- ★ Table can be very large
- ★ Operating sys. may maintain one page table per process
- ★ A dirty bit is used to track modified pages for copy back

Indicates whether the virtual page is in main memory or not

Linux 2-Level Page Table



- The CR3 register is designated for pointing to the first level page table
- The CR3 is part of the task state that needs to be saved at preemption

31 ... 22	21 ... 12	11 ... 0
Index into Page Table Table	Index into Page Table	Index into Page

3.7.1. Linear Address Translation (4-KByte Pages)

Figure 3-12 shows the page directory and page-table hierarchy when mapping linear addresses to 4-KByte pages. The entries in the page directory point to page tables, and the entries in a page table point to pages in physical memory. This paging method can be used to address up to 2^{20} pages, which spans a linear address space of 2^{32} bytes (4 GBytes).

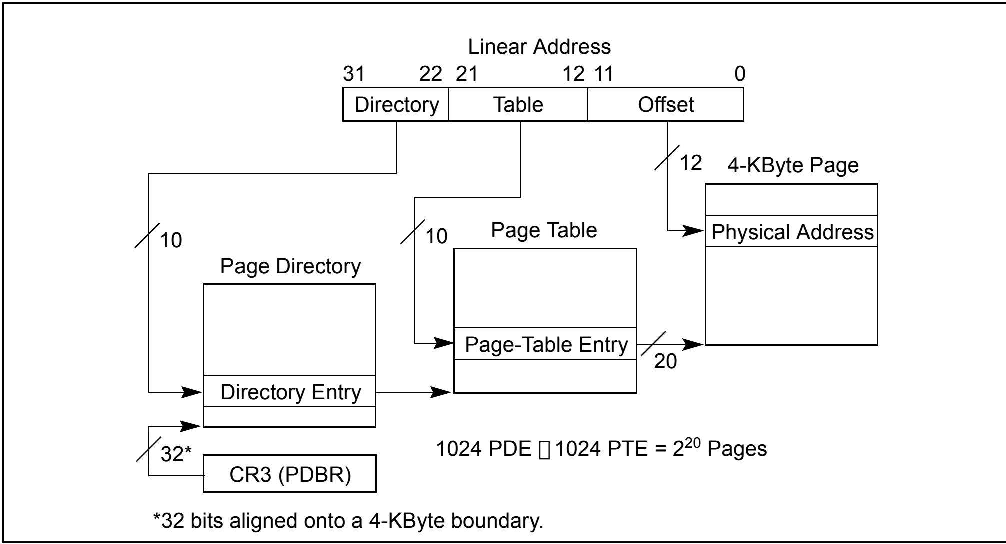


Figure 3-12. Linear Address Translation (4-KByte Pages)

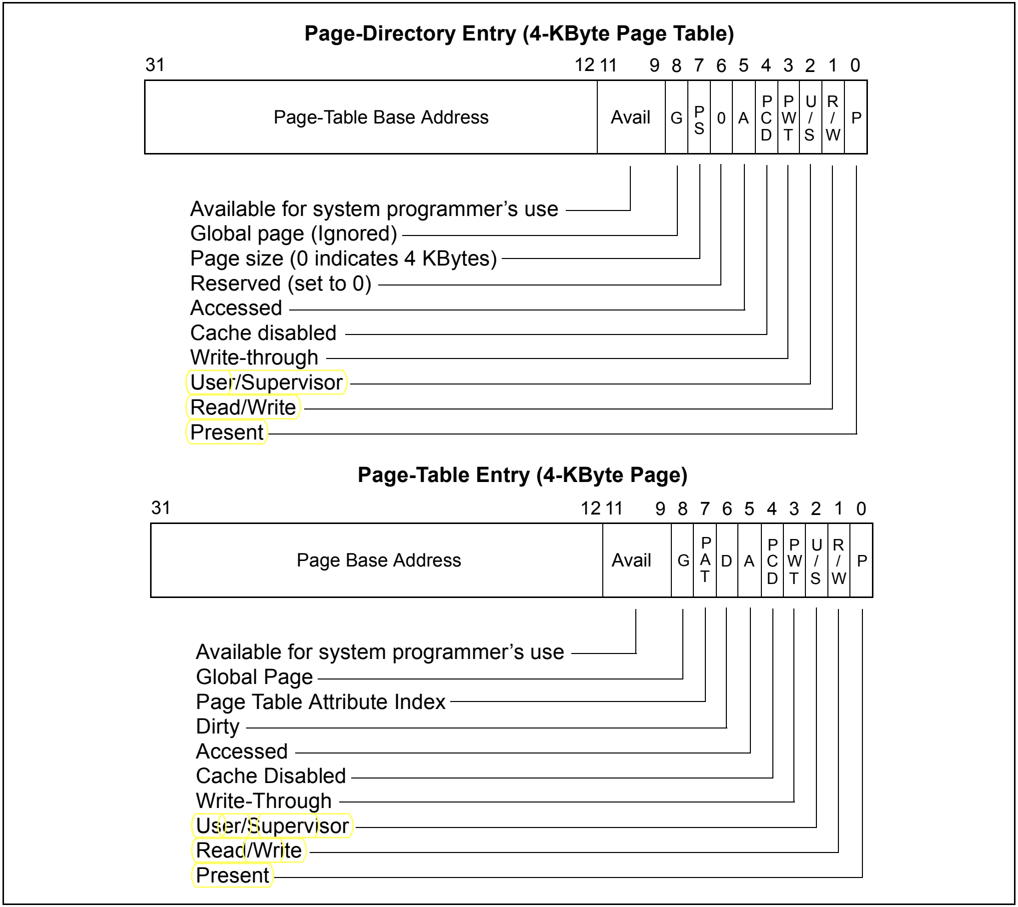
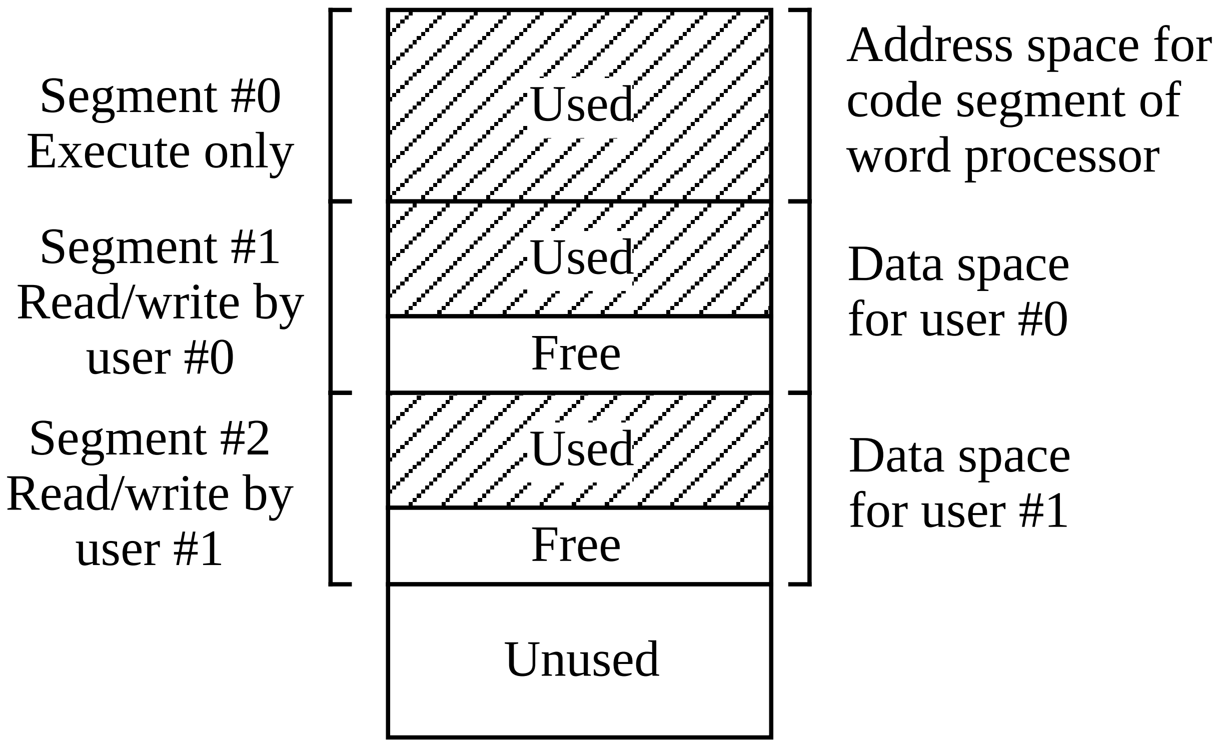


Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses

Segmentation

- **A segmented memory allows two users to share the same word processor code, with different data spaces:**



Translation Lookaside Buffer

- An example TLB holds 8 entries for a system with 32 virtual pages and 16 page frames.

Valid	Virtual page number	Physical page number
1	0 1 0 0 1	1 1 0 0
1	1 0 1 1 1	1 0 0 1
0	- - - - -	- - - -
0	- - - - -	- - - -
1	0 1 1 1 0	0 0 0 0
0	- - - - -	- - - -
1	0 0 1 1 0	0 1 1 1
0	- - - - -	- - - -

Virtual Memory: Problems Solved

- **Not enough physical memory**
  - Uses disk space to simulate extra memory
  - Pages not being used can be swapped out
    (how and when you'll learn in CMSC 421 Operating Systems)
  - Thrashing: pages constantly written to and retrieved from disk
    (time to buy more RAM)
- **Fragmentation**
  - Contiguous blocks of virtual memory do not have to map to contiguous sections of real memory
- **Memory protection**
  - Each process has its own page table
  - Shared pages are read-only
  - User processes cannot alter the page table (must be supervisor)

Virtual Memory: too slow?

- **Address translation is done in hardware**

  In the middle of the fetch execute cycle for:

  ```
  MOV     EAX, [buffer]
  ```

  the physical address of buffer is computed in hardware.

- **Recently computed page locations are cached in the translation lookaside buffer (TLB)**
- **Page faults *are* very expensive (millions of cycles)**
- **Operating systems for personal computers have only recently added memory protection**

Next Time

- **Memory Cache**
- **Interrupts**
- **Review for Midterm Exam**