CMSC 313 Lecture 13

- Project 3 Questions
- Project 4
- C functions continued
- Virtual Memory

Project 3: An Error-Correcting Code — Due: Thursday October 14, 2004

Objective

The objectives of this programming project are 1) for you to gain familiarity with data manipulation at the bit level and 2) for you to write more complex assembly language programs.

Background

In Project 2, we saw that checksums can be used detect corrupted files. However, there is not much we can do after we have detected the corruption. An error-correcting code is able to fix errors, not just detect them.

In this project, we will use a 31-bit Hamming code that can correct a 1-bit error in each 32-bit codeword. Each 32-bit codeword encodes 3 bytes of the original data. The format of the codeword is shown on the next page.

Assignment

Write an assembly language program that encodes the input file using the codeword format described below. As in Project 2, use Unix input and output redirection:

```
./a.out <ifile >ifile.ham
```

Some details:

- Your program must read a block of bytes from the input. You should not read from the input one byte at a time or three bytes at a time. (That would be terribly inefficient.)
- You may assume that when the operating system returns with 0 bytes read that the end of the input file has been reached. On the other hand, you may not assume that the end of the file has been reached when the operating system gives you fewer bytes than your block size. Similarly, you may not assume that the operating system will comply with your request for a number of input bytes that is divisible by 3.
- The 32-bit codewords must be written out in little-endian format.

The C source code for two programs `decode.c` and `corrupt.c` are provided in the GL file system in the directory: `/afs/umbc.edu/users/c/h/chang/pub/cs313`. These two programs can be used to decode an encoded file and to corrupt an encoded file. You can use these programs to check if your program is working correctly. Both programs use I/O redirection.

Record some sample runs of your program using the Unix `script` command. You should show that you can encode a file using your program, then decode it and obtain a file that is identical to the original. Use the Unix `diff` command to compare the original file with the decoded file. You should also show that this works when the file is corrupted.

Implementation Notes

- The parity flag PF is set to 1 if the result of an instruction contains an even number of 1's. Unfortunately, PF only looks at the lowest 8 bits of the result. For this project, you will need to compute 32-bit parities. Here's a simple way to comput the parity of the EAX register.

```
mov     ebx,eax
shr     eax,16
xor     ax,bx
xor     al,ah
jp      even_label
```

  Note that the EAX and EBX registers are modified in this process, so you may need to use different registers.

- A main issue in this project is handling the "extra characters" at the end of a block of input after you have processed all the 3-byte "groups". E.g., if your block size 128, then you will have 2 characters left over after processing 42 three-byte groups (42 x 3 = 126). These 2 extra characters must be grouped with the first character of the next block (if there is a next block). Think about this situation *before* you begin coding.

- Another main issue is the last 32-bit word output by your program. Note that the bits m1 and m0 must be set *before* you compute the parity bits p4, p3, p2, p1 and p0.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj3`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj3 encode.asm typescript
```

Codeword format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	p4	b0	c7	c6	c5	c4	c3	c2	p3	c1	c0	m1	p2	m0	p1	p0	0

bit 0 is not used and always holds a 0.

1st byte of data = a7 a6 a5 a4 a3 a2 a1 a0

2nd byte of data = b7 b6 b5 b4 b3 b2 b1 b0

3rd byte of data = c7 c6 c5 c4 c3 c2 c1 c0

p4, p3, p2, p1 and p0 are used to ensure that these bit positions have an even number of 1's:

```
p0:  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31
p1:  2  3  6  7 10 11 14 15 18 19 22 23 26 27 30 31
p2:  4  5  6  7 12 13 14 15 20 21 22 23 28 29 30 31
p3:  8  9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
p4: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

m1 and m0 are only used in the last word of the encoded file. They depend on the original file size (in number of bytes).

m1 m0 = 00 if the file size mod 3 is 0

m1 m0 = 01 if the file size mod 3 is 1

m1 m0 = 10 if the file size mod 3 is 2

Project 4: An Error-Correcting Code, Part 2 Due: Thursday October 21, 2004

Objective

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

Assignment

Modify your assembly language program from Project 3 so that it can be called from a C program as a C function with the following function prototype:

```
char *encode(char *A, int n, int *mptr) ;
```

Here `A` is a pointer to a sequence of bytes in memory that should be encoded into the Hamming Code format from Project 3. The parameter `n` is the number of bytes in `A`. The result of the codewords must be stored in a memory location that is dynamically allocated. Your assembly language program must call `malloc()` to obtain a block of memory of the correct size. The address of this block of memory is the return value from `encode()`. The size of the block must be stored in the location specified by `mptr`.

Your program must work with the C main program `p4main.c` which is available in the following directory in the GL file system:

```
/afs/umbc.edu/users/c/h/chang/pub/cs313
```

This C program reads bytes from `stdin` and stores them in a dynamically allocated memory location. It then calls your `encode()` function and writes the resulting codewords to `stdout`. Thus, if your assembly language implementation of `encode()` works correctly, the program resulting from compiling it with `p4main.c` behaves exactly like the encoding program in Project 3.

If you cannot convert your assembly language program from Project 3 (e.g., if your program for Project 3 does not work), then you must implement a similar `encode()` function that copies the bytes in `A`, but inserts the byte `0xFF` after every three bytes. It must also pad the resulting memory block with `0xFF` so the total length is divisible by 4. Implementing this version of the project will incur a 10% penalty.

Implementation Notes

- Look up `malloc()` if you haven't used it in a while. Remember that calling `malloc()` from your assembly language program will clobber the EAX, ECX and EDX registers.
- You should check for the possibility that `malloc()` returns 0 because the system does not have as much memory as you have requested.
- You will most likely need to use the EBX, ESI and EDI registers. You should push them on the stack to save them. Remember that when you pop them off the stack, you must pop them off in the opposite order.
- You will need to pre-compute the size of the memory block that holds the resulting codewords. Look up the DIV instruction. Note that it does not support many addressing modes.
- Your program should not alter the sequence of bytes given to you. This might change the way you handle the "extra bytes" at the end.
- As in Project 3, you can use `decode`, `corrupt` and `diff` to check if your program produced the correct results.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj4`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj4 p4encode.asm typescript
```

Linux/gcc/i386 Function Call Convention

- Parameters pushed right to left on the stack
  - first parameter on top of the stack
- Caller saves EAX, ECX, EDX if needed
  - these registers will probably be used by the callee
- Callee saves EBX, ESI, EDI
  - there is a good chance that the callee does not need these
- EBP used as index register for parameters, local variables, and temporary storage
- Callee must restore caller's ESP and EBP
- Return value placed in EAX

A typical stack frame for the function call:

```
int foo (int arg1, int arg2, int arg3) ;
```

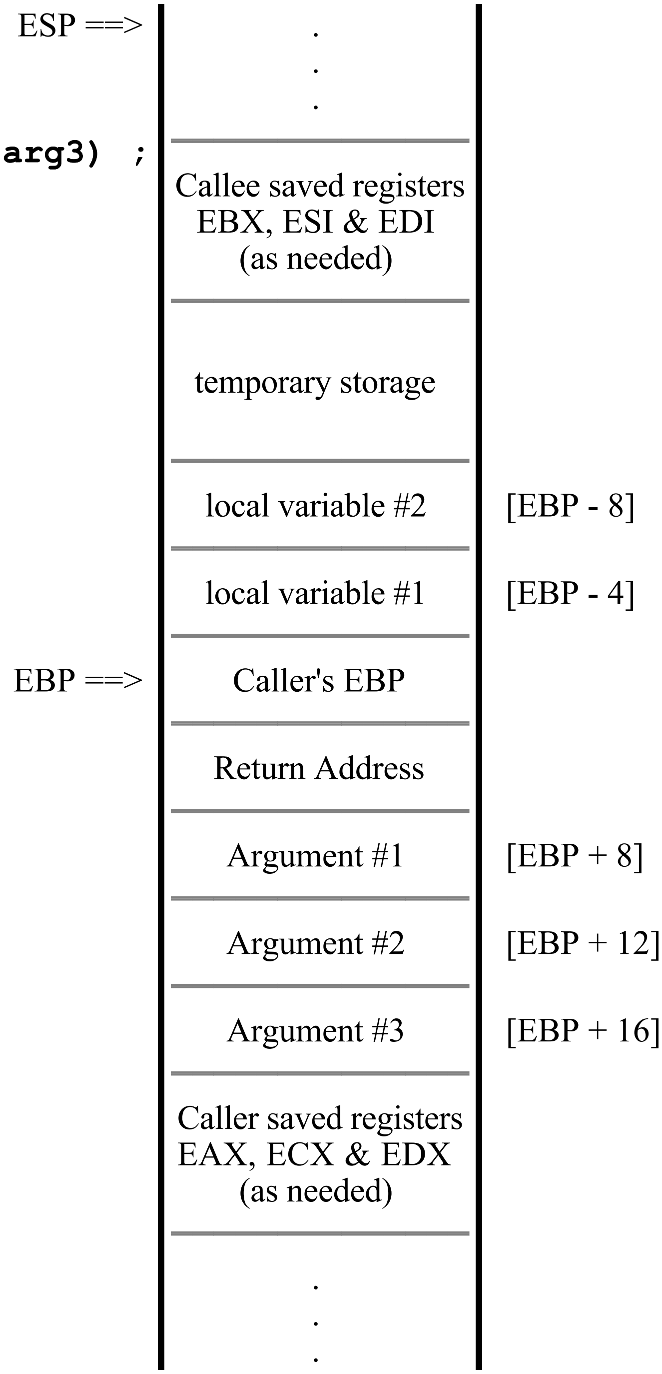


Fig. 1

```
; File: printf1.asm
;
; Using C printf function to print
;
; Assemble using NASM:  nasm -f elf printf1.asm
;
; C-style main function.
; Link with gcc:  gcc printf1.o
;

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Hello, world: %c", 10, 0    ; The string to print.


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp

        push    dword 97                ; an 'a'
        push    dword msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 8                  ; pop stack

        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op

        ret
```

```
linux3% nasm -f elf printf1.asm
linux3% gcc printf1.o

linux3% a.out
Hello, world: a
linux3% exit
```

```
; File: printf2.asm
;
; Using C printf function to print
;
; Assemble using NASM:  nasm -f elf printf2.asm
;
; Assembler style main function.
; Link with gcc: gcc -nostartfiles printf2.asm
;

%define SYSCALL_EXIT  1

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Hello, world: %c", 10, 0    ; The string to print.


        SECTION .text                   ; Code section.

        global _start
_start:
        push    dword 97                ; an 'a'
        push    dword msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 8                  ; pop stack

        mov     eax, SYSCALL_EXIT       ; Exit.
        mov     ebx, 0                  ; exit code, 0=normal
        int     080H                    ; ask kernel to take over
```

```
linux3% nasm -f elf printf2.asm
linux3% gcc -nostartfiles printf2.o
linux3%

linux3% a.out
Hello, world: a
linux3%
```

```
// File: arraytest.c
//
// C program to test arrayinc.asm
//

void arrayinc(int A[], int n) ;

main() {

int A[7] = {2, 7, 19, 45, 3, 42, 9} ;
int i ;

   printf ("sizeof(int) = %d\n", sizeof(int)) ;

   printf("\nOriginal array:\n") ;
   for (i = 0 ; i < 7 ; i++) {
      printf("A[%d] = %d  ", i, A[i]) ;
   }
   printf("\n") ;

   arrayinc(A,7) ;

   printf("\nModified array:\n") ;
   for (i = 0 ; i < 7 ; i++) {
      printf("A[%d] = %d  ", i, A[i]) ;
   }
   printf("\n") ;

}
```

```
linux3% gcc -c arraytest.c
linux3% nasm -f elf arrayinc.asm
linux3% gcc arraytest.o arrayinc.o
linux3%
linux3% a.out
sizeof(int) = 4

Original array:
A[0] = 2  A[1] = 7  A[2] = 19  A[3] = 45  A[4] = 3  A[5] = 42  A[6] = 9

Modified array:
A[0] = 3  A[1] = 8  A[2] = 20  A[3] = 46  A[4] = 4  A[5] = 43  A[6] = 10
linux3%
```

```
; File: arrayinc.asm
;
; A subroutine to be called from C programs.
; Parameters: int A[], int n
; Result: A[0], ... A[n-1] are each incremented by 1


        SECTION .text
        global arrayinc

arrayinc:
        push    ebp                     ; set up stack frame
        mov     ebp, esp

        ; registers ebx, esi and edi must be saved, if used
        push    ebx
        push    edi

        mov     edi, [ebp+8]            ; get address of A
        mov     ecx, [ebp+12]           ; get num of elts
        mov     ebx, 0                  ; initialize count

for_loop:
        mov     eax, [edi+4*ebx]        ; get array element
        inc     eax                     ; add 1
        mov     [edi+4*ebx], eax        ; put it back
        inc     ebx                     ; update counter
        loop    for_loop

        pop     edi                     ; restore registers
        pop     ebx

        mov     esp, ebp                ; take down stack frame
        pop     ebp

        ret
```

```
// File: cfunc3.c
//
// Example of C function calls disassembled
// Return values with  more than 4 bytes
//

#include <stdio.h>

typedef struct {
   int part1, part2 ;
} stype ;


// a silly function
//
stype foo(stype r) {

   r.part1 += 4;
   r.part2 += 3 ;
   return r ;
}


int main () {
   stype r1, r2, r3  ;
   int n ;

   n = 17 ;
   r1.part1 = 74 ;
   r1.part2 = 75 ;
   r2.part1 = 84 ;
   r2.part2 = 85 ;
   r3.part1 = 93 ;
   r3.part2 = 99 ;

   r2 = foo(r1) ;

   printf ("r2.part1 = %d, r2.part2 = %d\n",
    r1.part1, r2.part2 ) ;

   n = foo(r3).part2 ;
}
```

```
        ;FILE "cfunc3.c"
gcc2_compiled.:
SECTION .text
        ALIGN 4
GLOBAL foo
        GLOBAL foo:function
foo:                                    ; comments & spacing added
        push  ebp                       ; set up stack frame
        mov   ebp,esp

        mov   eax, [ebp+8]              ; addr to store return value
        add   dword [ebp+12],4          ; r.part1 = [ebp+12]
        add   dword [ebp+16],3          ; r.part2 = [ebp+16]

        ; return value
        ;
        mov   edx, [ebp+12]             ; get r.part1
        mov   ecx, [ebp+16]             ; get r.part2
        mov   [eax],edx                 ; put r.part1 in return valu
        mov   [eax+4],ecx               ; put r.part2 in return valu
        jmp L1
L1:
        mov   eax,eax                   ; does nothing
        leave                           ; bye-bye
        ret 4                           ; pop 4 bytes after return
.Lfe1:
```

```
        GLOBAL   foo:function (.Lfe1-foo)
SECTION         .rodata
.LC0:
        db      'r2.part1 = %d, r2.part2 = %d',10,''
SECTION .text
        ALIGN 4
GLOBAL main
        GLOBAL main:function
main:                                   ; comments & spacing added
        push  ebp                       ; set up stack frame
        mov  ebp,esp
        sub  esp,36                     ; space for local variables

        ; initialize variables
        ;
        mov  dword [ebp-28],17          ; n = [ebp-28]
        mov  dword [ebp-8],74           ; r1 = [ebp-8]
        mov  dword [ebp-4],75
        mov  dword [ebp-16],84          ; r2 = [ebp-16]
        mov  dword [ebp-12],85
        mov  dword [ebp-24],93          ; r3 = [ebp-24]
        mov  dword [ebp-20],99

        ; call foo
        ;
        lea  eax, [ebp-16]              ; get addr of r2
        mov  edx, [ebp-8]               ; get r1.part1
        mov  ecx, [ebp-4]               ; get r1.part2
        push  ecx                       ; push r1.part2
        push  edx                       ; push r1.part1
        push  eax                       ; push addr of r2
        call foo
        add  esp,8                      ; pop r1
                                        ; ret 4 popped r2's addr

        ; call printf
        ;
        mov  eax, [ebp-12]              ; get r2.part2
        push  eax                       ; push it
        mov  eax, [ebp-8]               ; get r2.part1
        push  eax                       ; push it
        push  dword .LC0                ; string constant's addr
        call printf
        add  esp,12                     ; pop off arguments
```

```
        ; call foo again
        ;
        lea  eax, [ebp-36]              ; addr of temp variable
        mov  edx, [ebp-24]              ; get r3.part1
        mov  ecx, [ebp-20]              ; get r3.part2
        push  ecx                       ; push r3.part2
        push  edx                       ; push r3.part1
        push  eax                       ; push addr of temp var
        call foo
        add  esp,8                      ; pop off arguments

        ; assign to n
        ;
        mov  eax, [ebp-32]              ; get part2 of temp var
        mov  [ebp-28],eax               ; store in n

L2:
        leave                           ; bye-bye
        ret
.Lfe2:
        GLOBAL   main:function (.Lfe2-main)
        ;IDENT "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2
release)"
```

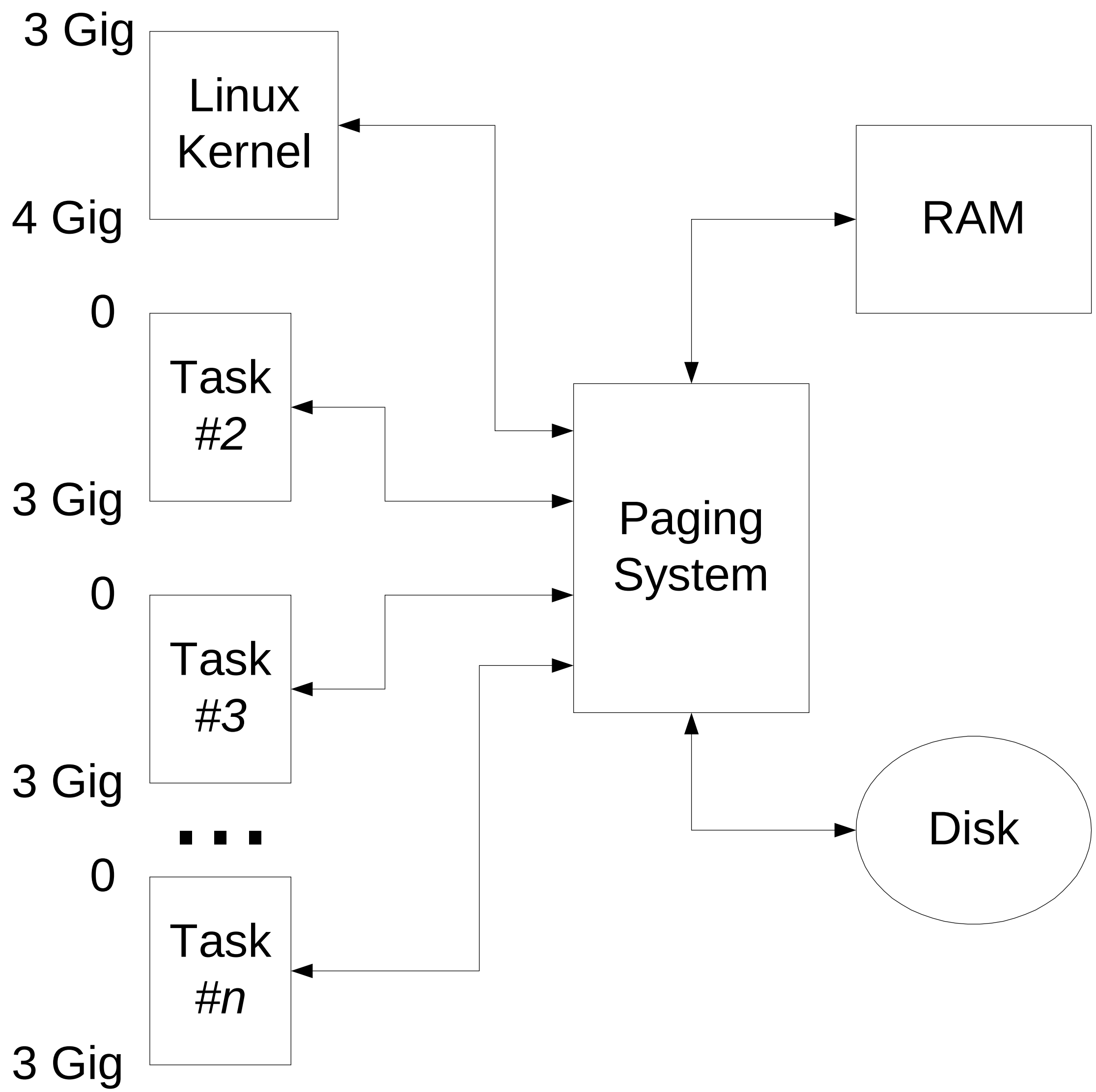
Memory Map

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

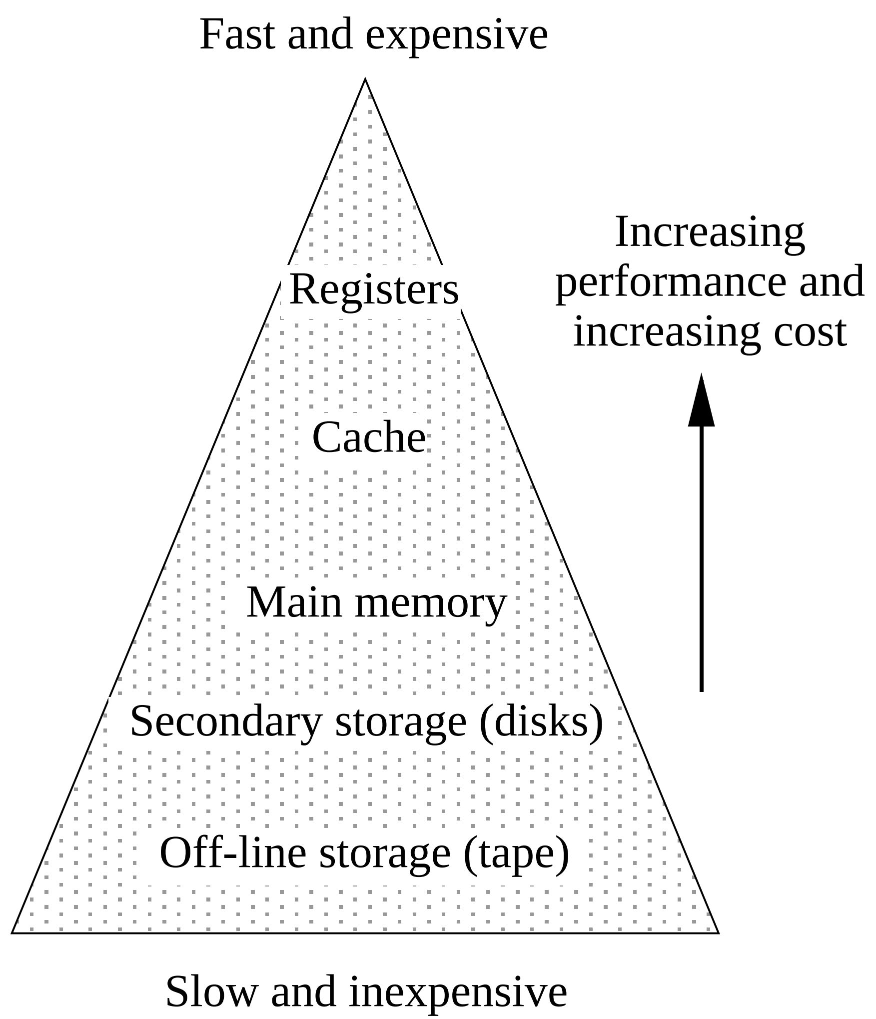
```
int global ;

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

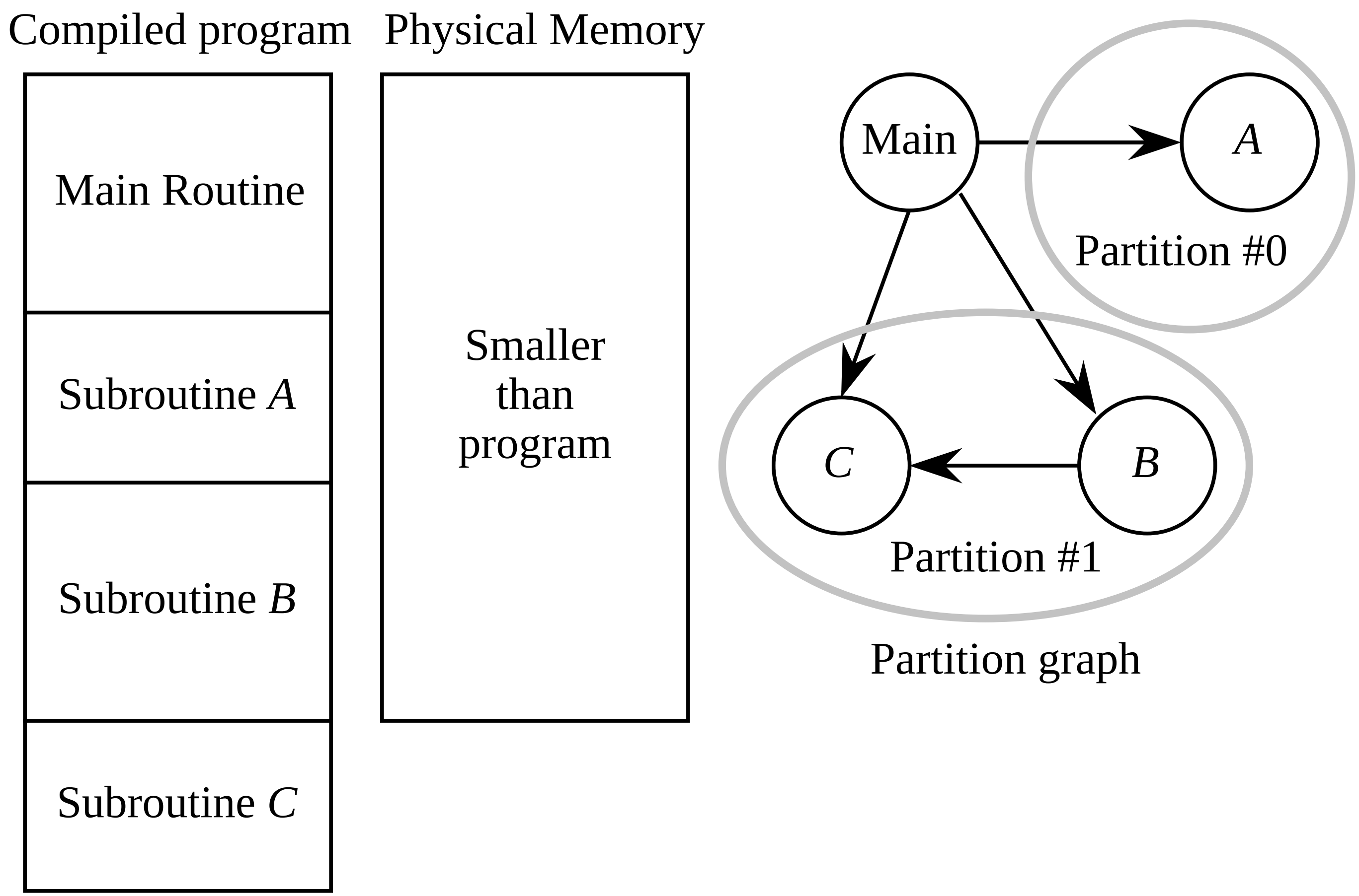


The Memory Hierarchy



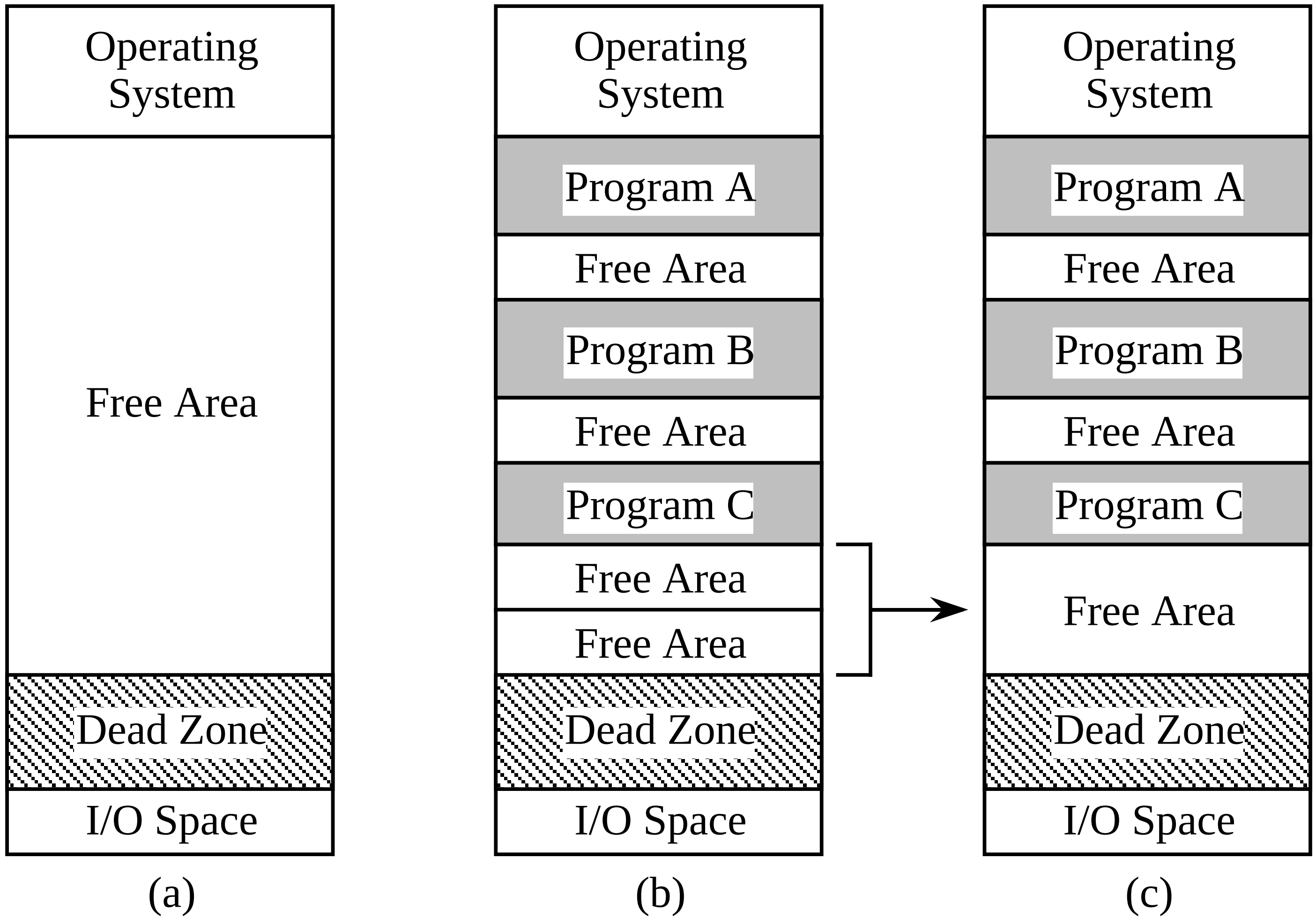
Overlays

- **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.**

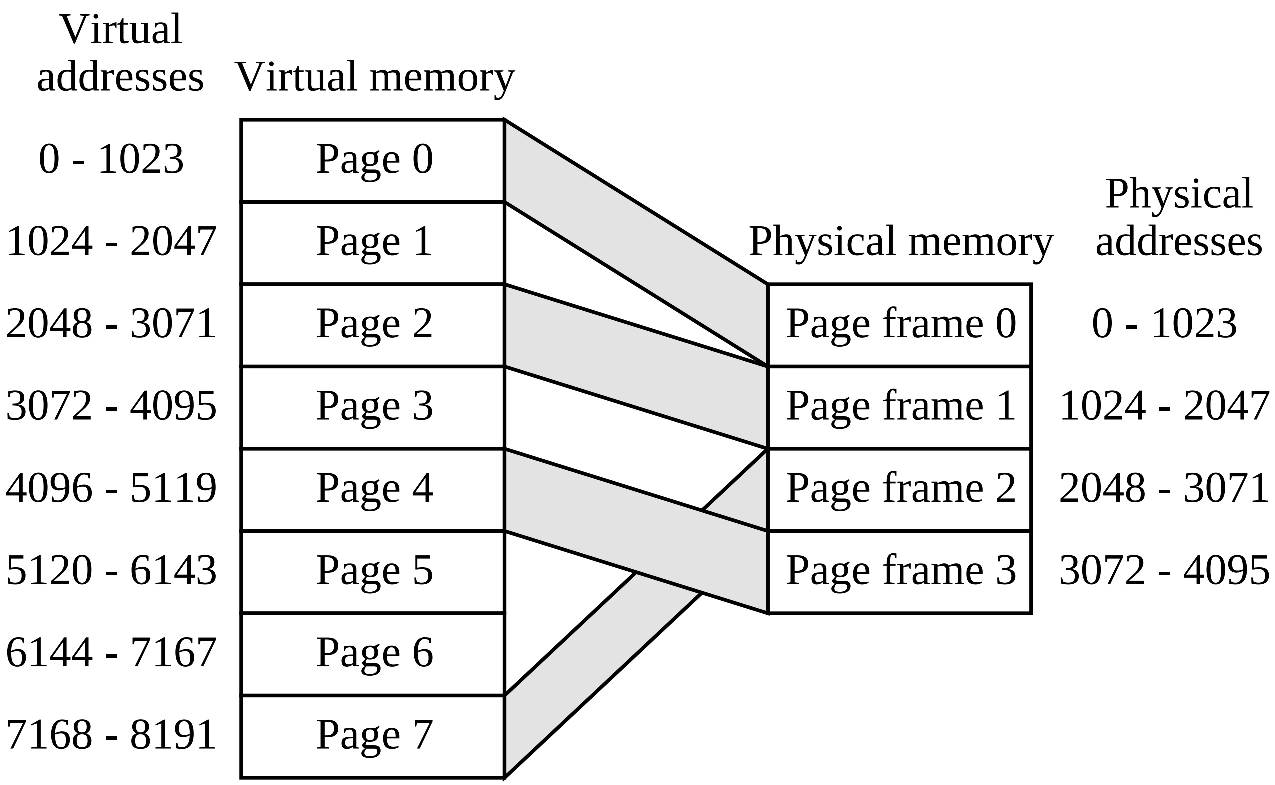


Memory Protection

- 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

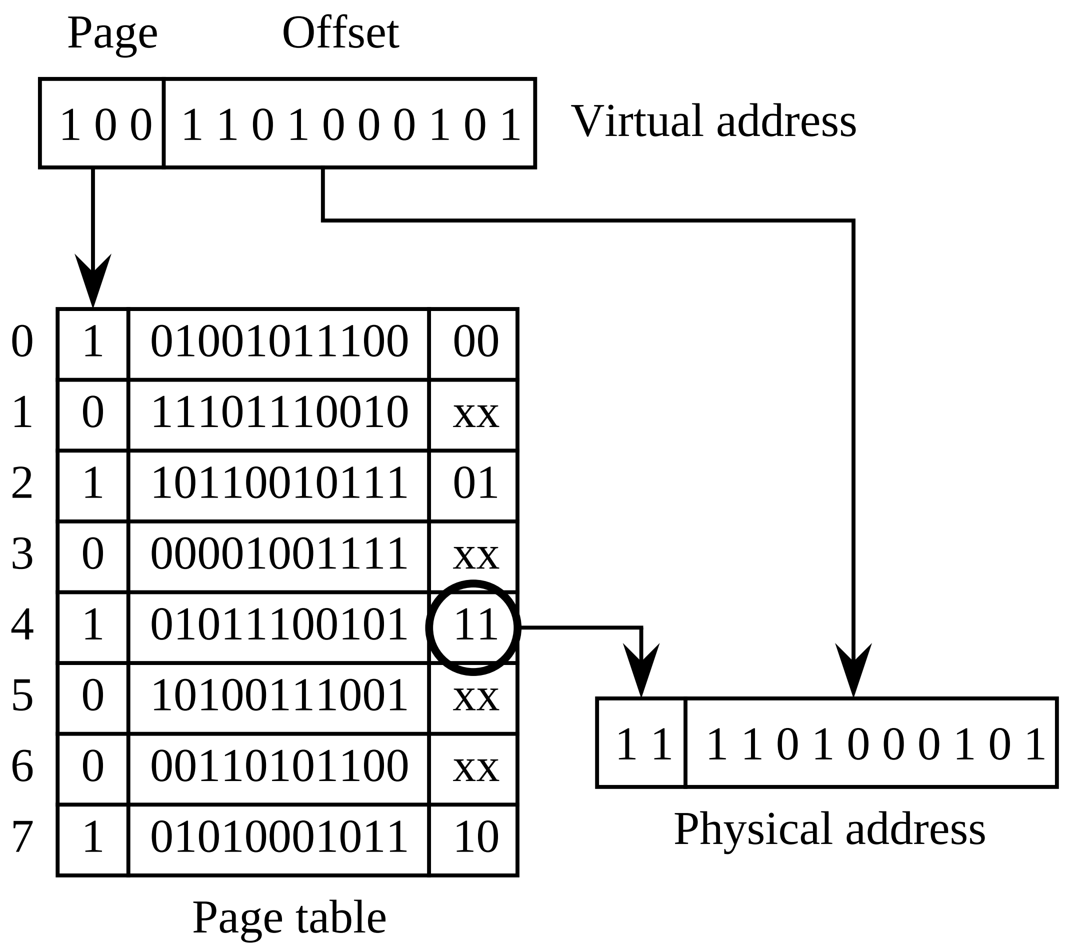
- **The page table maps between virtual memory and physical memory.**

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

Next Time

- Linux page tables
- Interrupts & System Calls