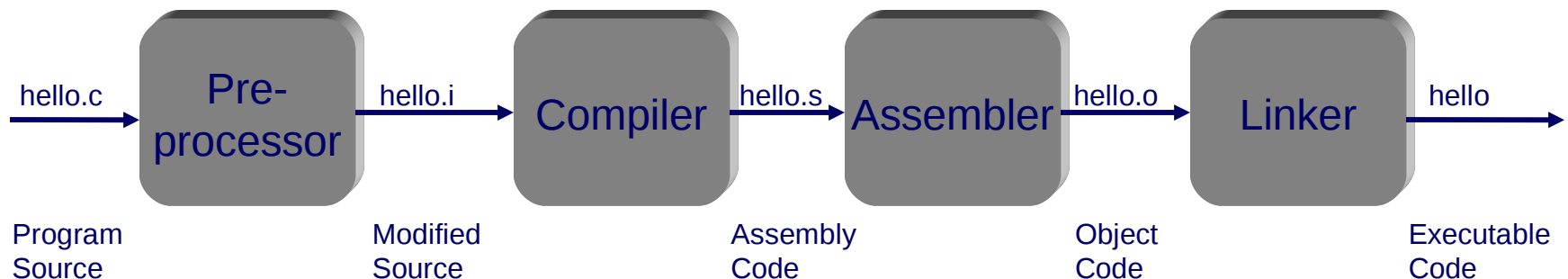


Review of C and x86

How software works

gcc compiler driver pre-processes, compiles, assembles and links to generate executable

- Links together object code (i.e. game.o) and static libraries (i.e. libc.a) to form final executable
- Links in references to dynamic libraries for code loaded at load time (i.e. libc.so.1)
- Executable may still load additional dynamic libraries at run-time



Static libraries

Suppose you have utility code in x.c, y.c, and z.c that all of your programs use

- **Link together individual .o files**

```
gcc -o hello hello.o x.o y.o z.o
```

- **Create a library libmyutil.a using ar and ranlib and link library in statically**

```
libmyutil.a : x.o y.o z.o
```

```
ar rvu libmyutil.a x.o y.o z.o
```

```
ranlib libmyutil.a
```

```
gcc -o hello hello.c -L. -lmyutil
```

- **Note: library code copied directly into binary**

Dynamic libraries

Avoid having multiple copies of common code on disk

- **Problem: libc**
 - “gcc program.c -lc” creates an a.out with entire libc object code in it (libc.a)
 - Almost all programs use libc!
- **Solution: Have binaries compiled with a reference to a library of shared objects versus an entire copy of the library**
 - Libraries loaded at run-time from file system
 - “ldd <binary>” to see which dynamic libraries a program relies upon
 - gcc flags “-shared” and “-soname” for handling and generating dynamic shared object files

The linking process (ld)

Merges object files

- Merges multiple relocatable (.o) object files into a single executable program.

Resolves external references

- References to symbols defined in another object file.

Relocates symbols

- Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
 - References in both code and data
 - » code: `a();` /* reference to symbol a */
 - » data: `int *xp=&x;` /* reference to symbol x */

Executables

Various file formats

- Linux = Executable and Linkable Format (ELF)
- Windows = Portable Executable (PE)

ELF

Standard binary format for object files in Linux

One unified format for

- Relocatable object files (.o),
- Shared object files (.so)
- Executable object files

Better support for shared libraries than old a.out formats.

More complete information for debuggers.

ELF Object File Format

ELF header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program header table

- Page size, virtual addresses of memory segments (sections), segment sizes, entry point

.text section

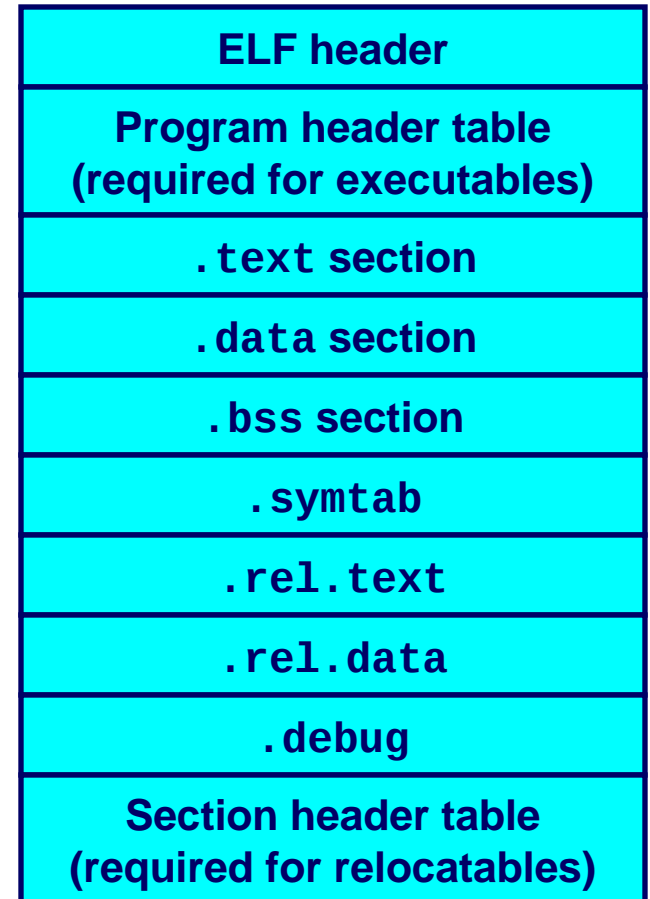
- Code

.data section

- Initialized (static) data

.bss section

- Uninitialized (static) data
- “Block Started by Symbol”



ELF Object File Format (cont)

.symtab section

- Symbol table
- Procedure and static variable names
- Section names and locations

.rel.text section

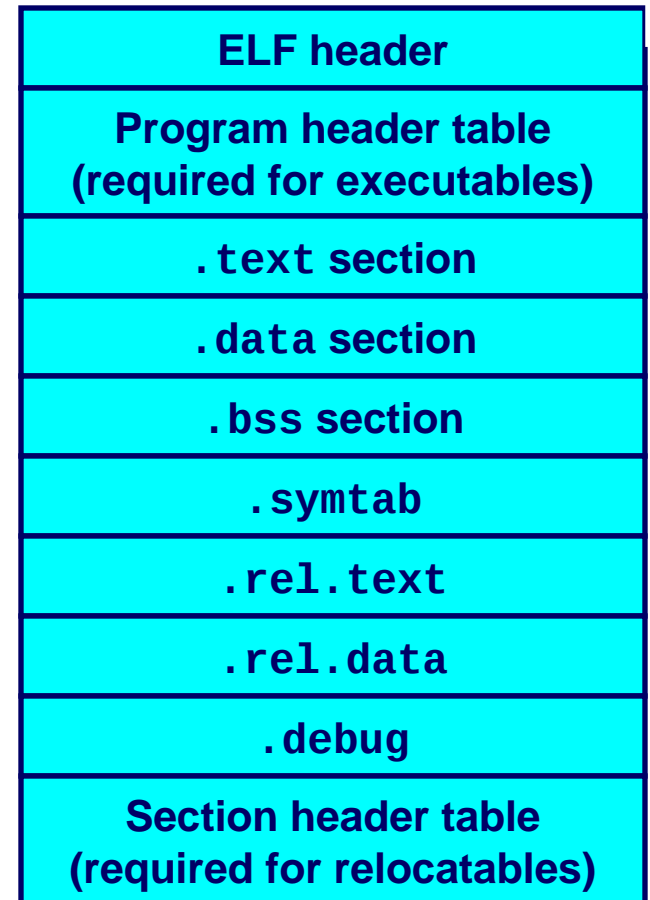
- Relocation info for .text section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

.rel.data section

- Relocation info for .data section
- Addresses of pointer data that will need to be modified in the merged executable

.debug section

- Info for symbolic debugging (gcc -g)



PE (Portable Executable) file format

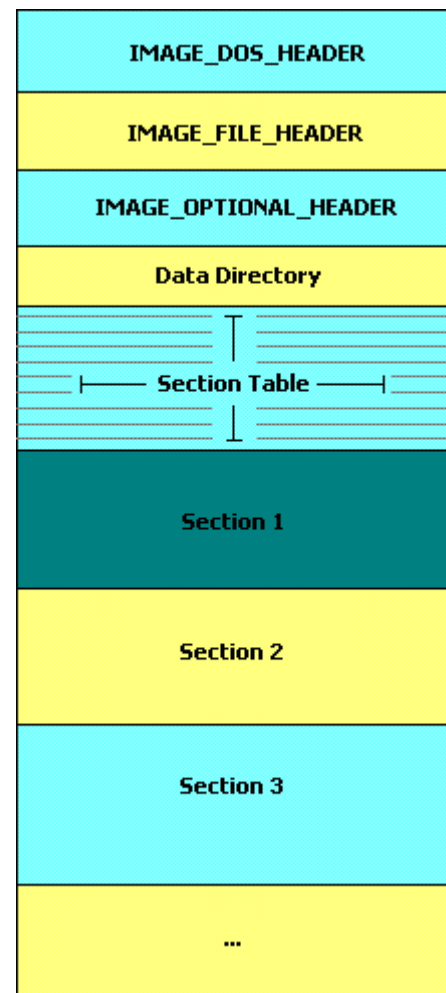
Windows file format for executables

Based on COFF Format

- Magic Numbers, Headers, Tables, Directories, Sections

Disassemblers

- Overlay Data with C Structures
- Load File as OS Loader Would
- Identify Entry Points (Default & Exported)



Example C Program

m.c

```
int e=7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

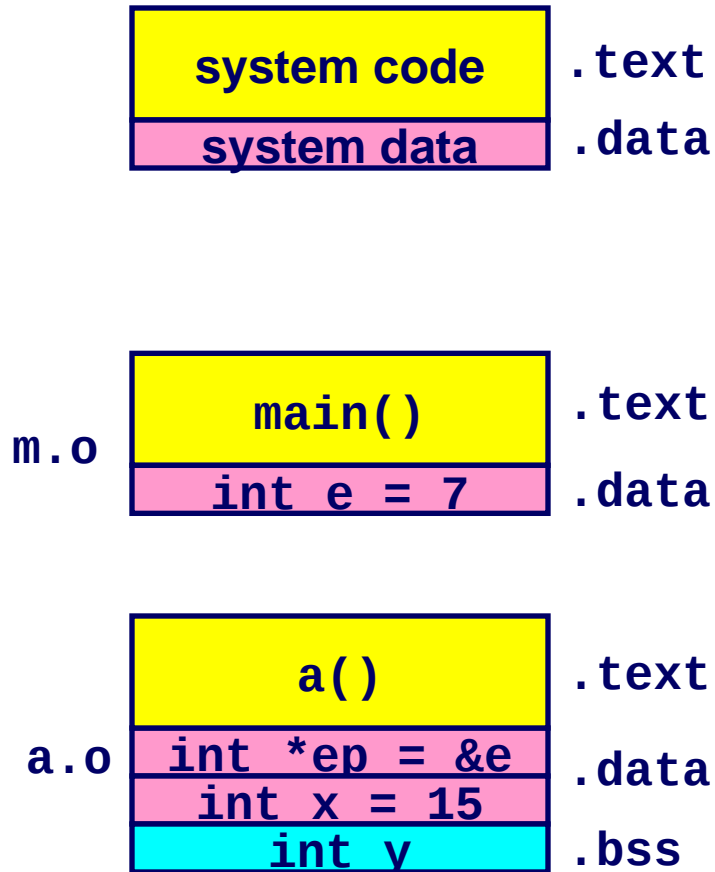
```
extern int e;

int *ep=&e;
int x=15;
int y;

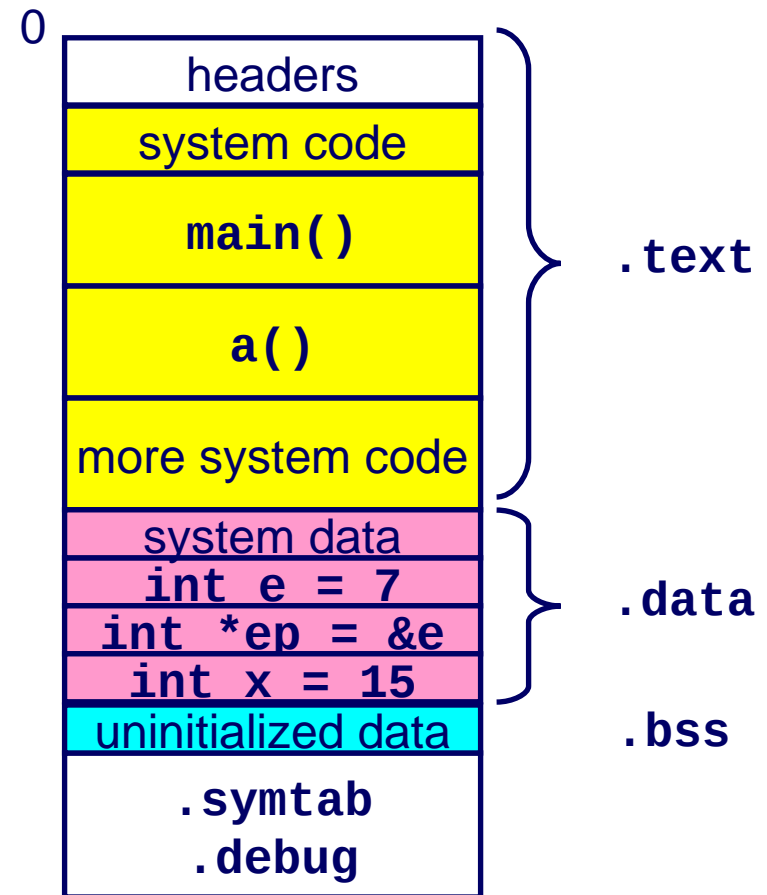
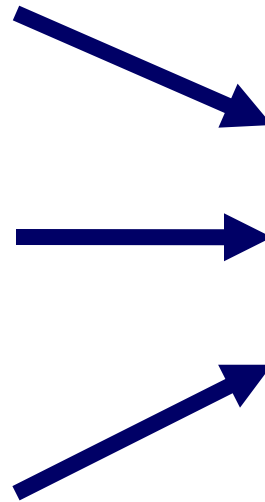
int a() {
    return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File

Relocatable Object Files



Executable Object File



Program execution

Operating system provides

- Protection and resource allocation
- Abstract view of resources (files, system calls)
- Virtual memory
 - Uniform memory space abstraction for each process
 - Gives the illusion that each process has entire memory space

How does a program get loaded?

The operating system creates a new process.

- Including among other things, a virtual memory space
- Important: any hardware-based debugger must know OS state in page tables to map accesses to virtual addresses

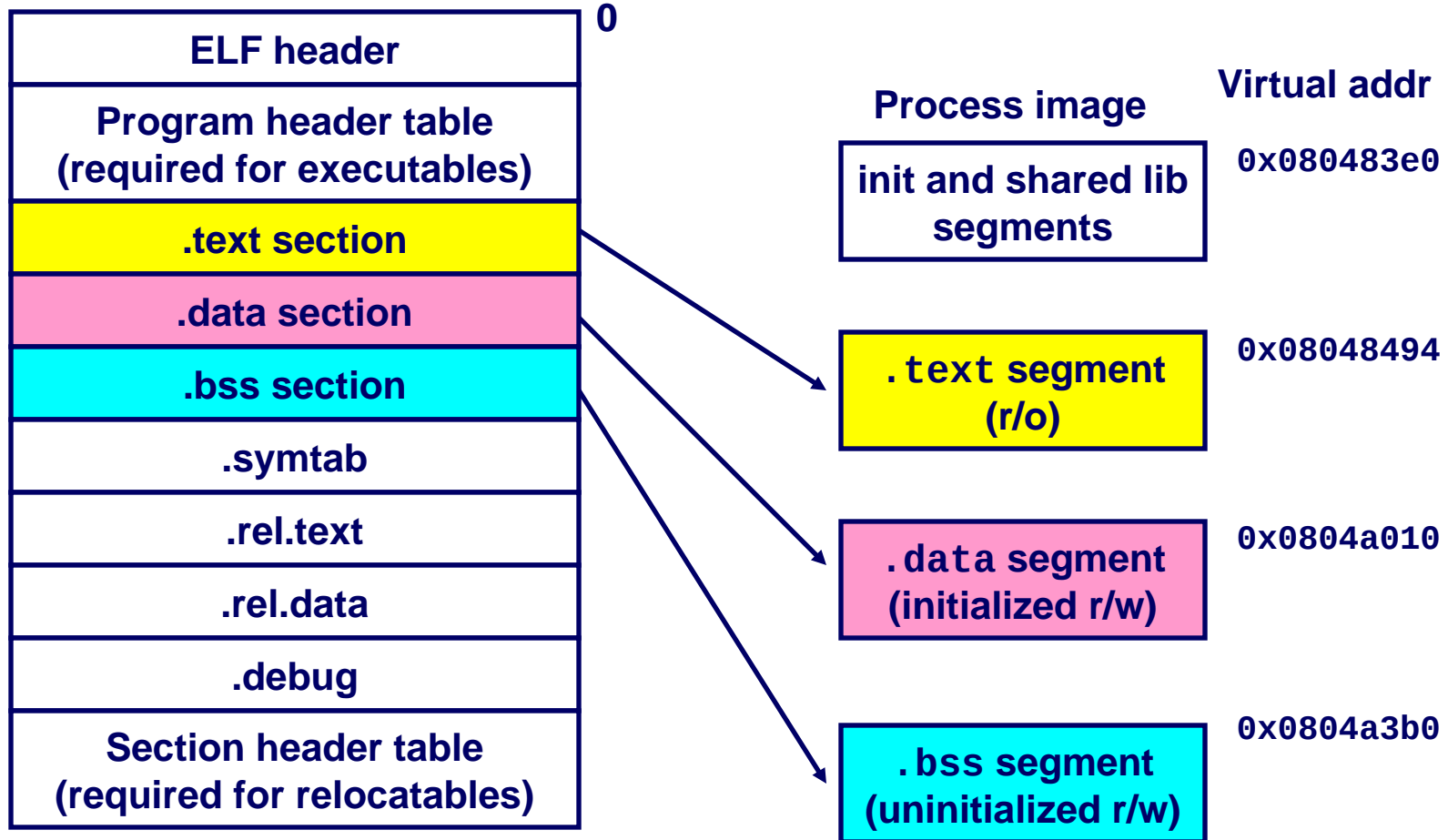
System loader reads the executable file from the file system into the memory space.

- Reads executable from file system into memory space
 - Executable contains code and statically link libraries
 - Done via DMA (direct memory access)
 - Executable in file system remains and can be executed again
- Loads dynamic shared objects/libraries into memory
- Resolves addresses in code given where code/data is loaded

Then it starts the thread of execution running

Loading Executable Binaries

Executable object file for
example program p

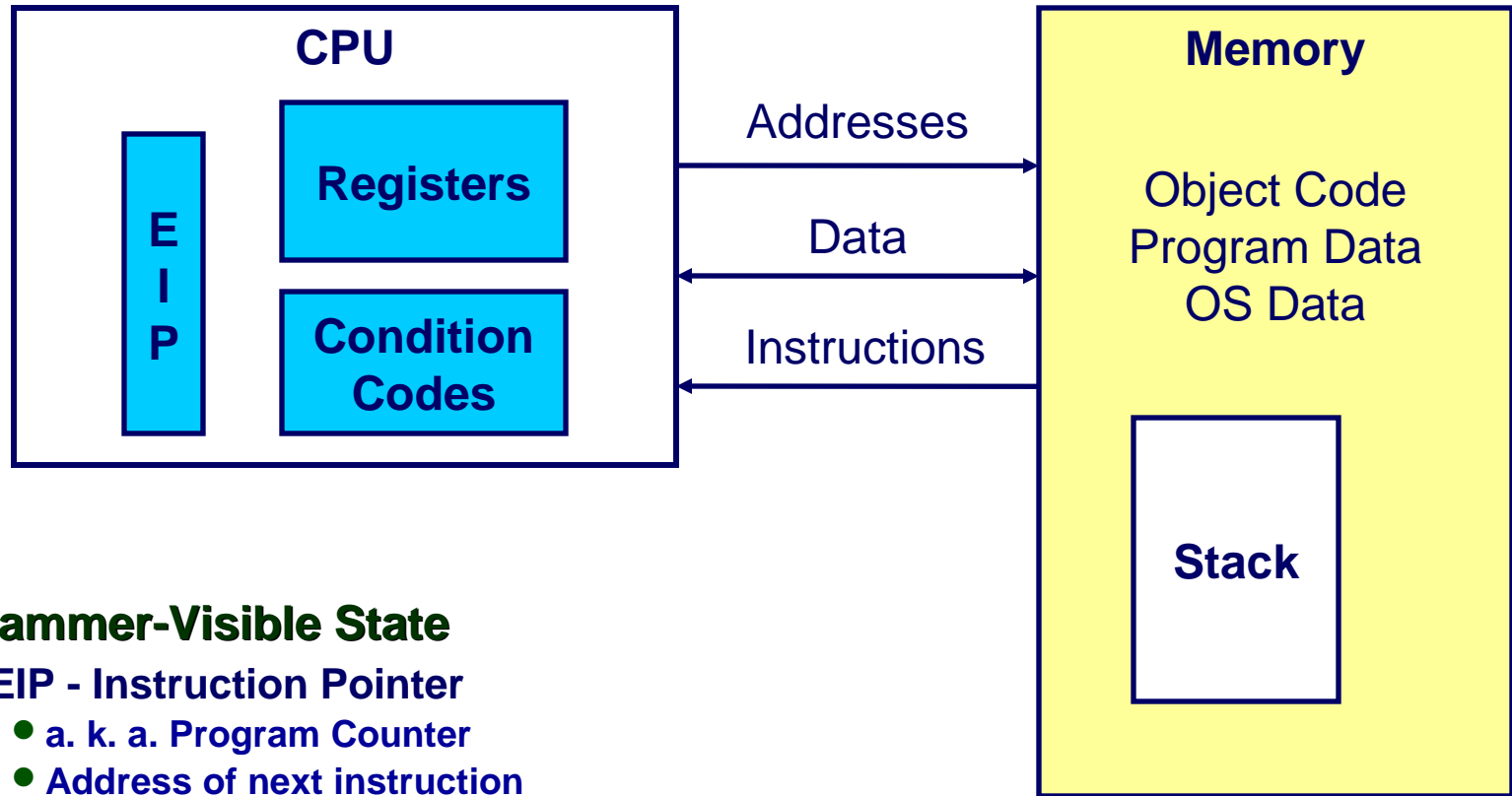


More on relocation

Assembly code with relative and absolute addresses

- **With VM abstraction, old linkers decide layout and can supply definitive addresses**
 - Windows “.com” format
 - Linker can statically bind the program to virtual addresses
 - Now, they provide hints as to where they would like to be placed
- **But....this could also be done at load time (address space layout randomization)**
 - Windows “.exe” format
 - Loader rewrites addresses to proper offsets
 - **System needs to force position-independent code**
 - » Force compiler to make all jumps and branches relative to current location or relative to a base register set at run-time
 - **ELF uses Global Offset Table**
 - » Symbol addresses obtained from GOT before access
 - » Can be targetted for hooks!
 - » Implementation determines exploit

Program execution



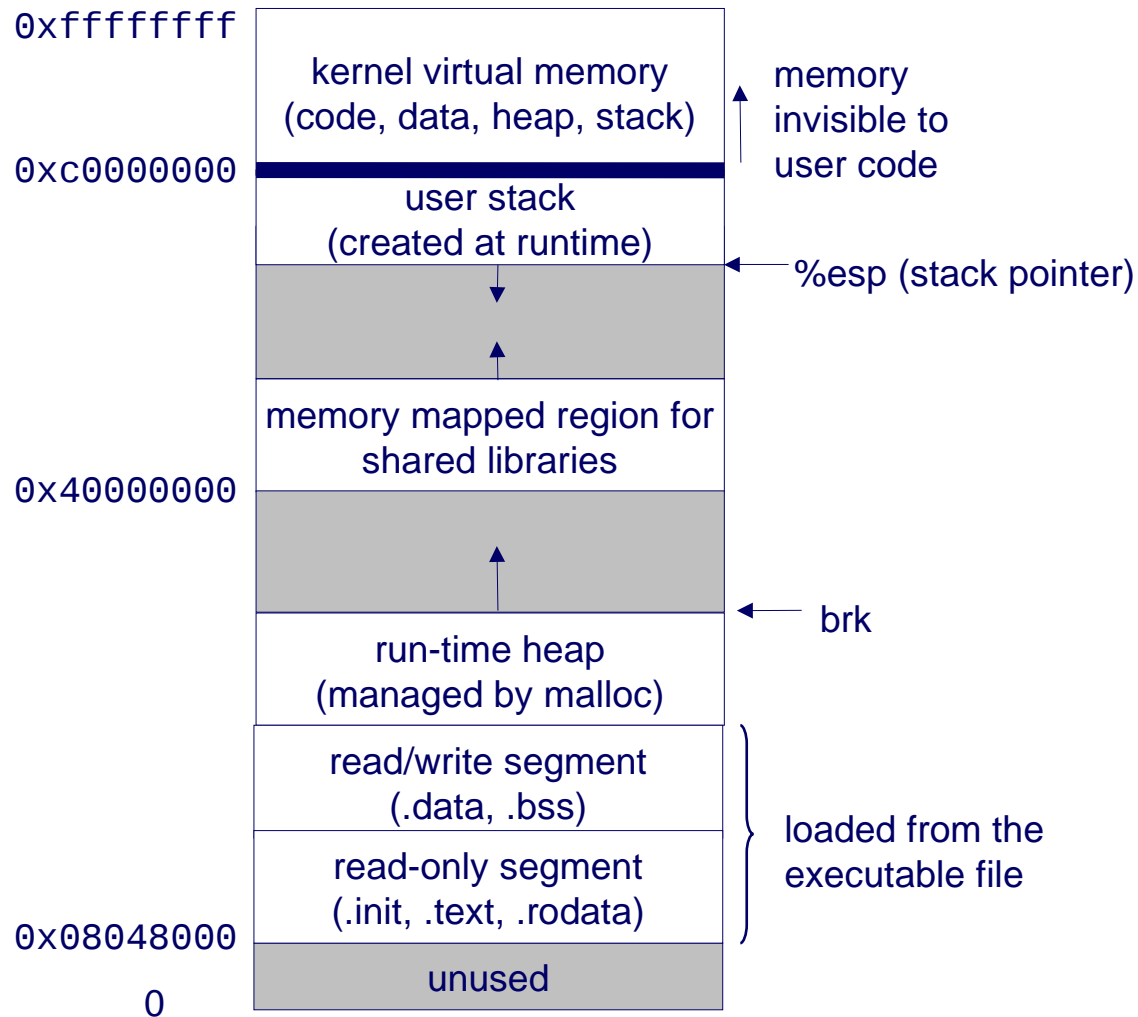
Programmer-Visible State

- EIP - Instruction Pointer
 - a. k. a. Program Counter
 - Address of next instruction
- Register File
 - Heavily used program data
- Condition Codes
 - Store status information about most recent arithmetic operation
 - Used for conditional branching

Memory

- Byte addressable array
- Code, user data, OS data
- Includes stack used to support procedures

Run-time data structures



Registers

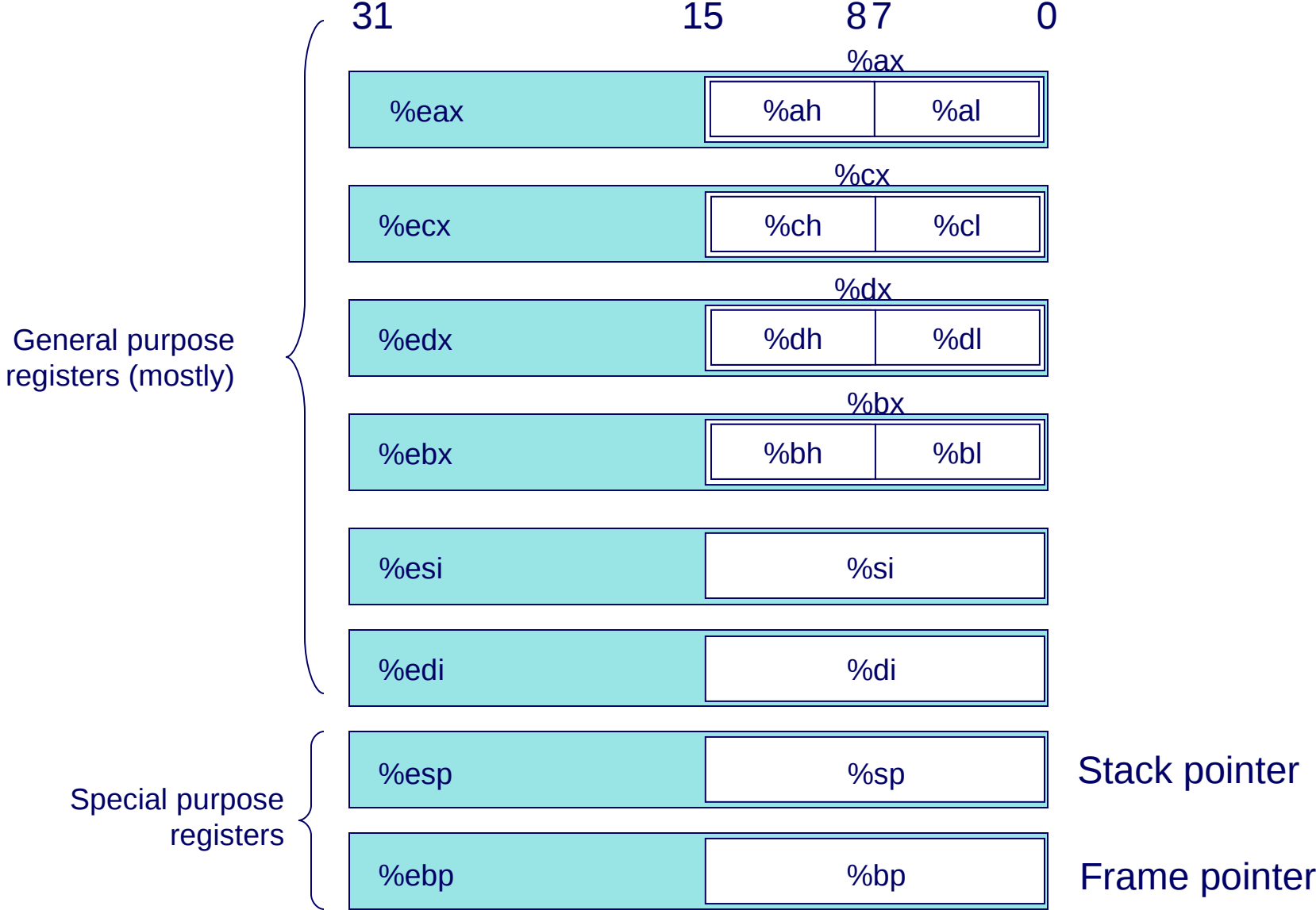
The processor operates on data in registers (usually)

- `movl (%eax), %ecx`
 - Fetch data at address contained in %eax
 - Store in register %ecx
- `movl $array, %ecx`
 - Move address of variable array into %ecx
- Typically, data is loaded into registers, manipulated or used, and then written back to memory

The IA32 architecture is “register poor”

- Few general purpose registers
- Source or destination operand is often memory locations
- Makes context-switching amongst processes easy (less register-state to store)

IA32 General Registers



Operand types

A typical instruction acts on 1 or more *operands*

- `addl %ecx, %edx` adds the contents of `ecx` to `edx`

Three general types of operands

- **Immediate**
 - Like a C constant, but preceded by `$`
 - e.g., `$0x1F`, `$-533`
 - Encoded with 1, 2, or 4 bytes based on instruction
- **Register**: the value in one of the 8 integer registers
- **Memory**: a memory address
 - There are many modes for addressing memory

Operand examples using `mov`

	Source	Destination	C Analog	
movl	Imm	Reg	<code>movl \$0x4,%eax</code>	<code>temp = 0x4;</code>
		Mem	<code>movl \$-147, (%eax)</code>	<code>*p = -147;</code>
	Reg	Reg	<code>movl %eax,%edx</code>	<code>temp2 = temp1;</code>
		Mem	<code>movl %eax, (%edx)</code>	<code>*p = temp;</code>
	Mem	Reg	<code>movl (%eax),%edx</code>	<code>temp = *p;</code>

- Memory-memory transfers cannot be done with single instruction

Addressing Modes

Immediate and registers have only one mode

Memory on the other hand ...

- **Absolute**
 - specify the address of the data
- **Indirect**
 - use register to calculate address
- **Base + displacement**
 - use register plus absolute address to calculate address
- **Indexed**
 - **Indexed**
 - » Add contents of an index register
 - **Scaled index**
 - » Add contents of an index register scaled by a constant

Summary of IA32 Operand Forms

Type	Form	Operand Value	Name
Immediate	\$Imm	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] * s]$	Scaled Indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] * s]$	Scaled Indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] * s]$	Scaled Indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] * s]$	Scaled Indexed

x86 instructions

Rules

- Source operand can be memory, register or constant
- Destination can be memory or register
- Only one of source and destination can be memory
- Source and destination must be same size

Flags set on each instruction

- EFLAGS
- Conditional branches handled via EFLAGS

What's the "l" for on the end?

```
addl 8(%ebp),%eax
```

It stands for "long" and is 32-bits

It tells the size of the operand.

Baggage from the days of 16-bit processors

For x86, x86_64

- 8 bits is a *byte*
- 16 bits is a *word*
- 32 bits is a *double word*
- 64 bits is a *quad word*

IA32 Standard Data Types

C Declaration	Intel Data Type	GAS Suffix	Size in bytes
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
unsigned	Double word	l	4
long int	Double word	l	4
unsigned long	Double word	l	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

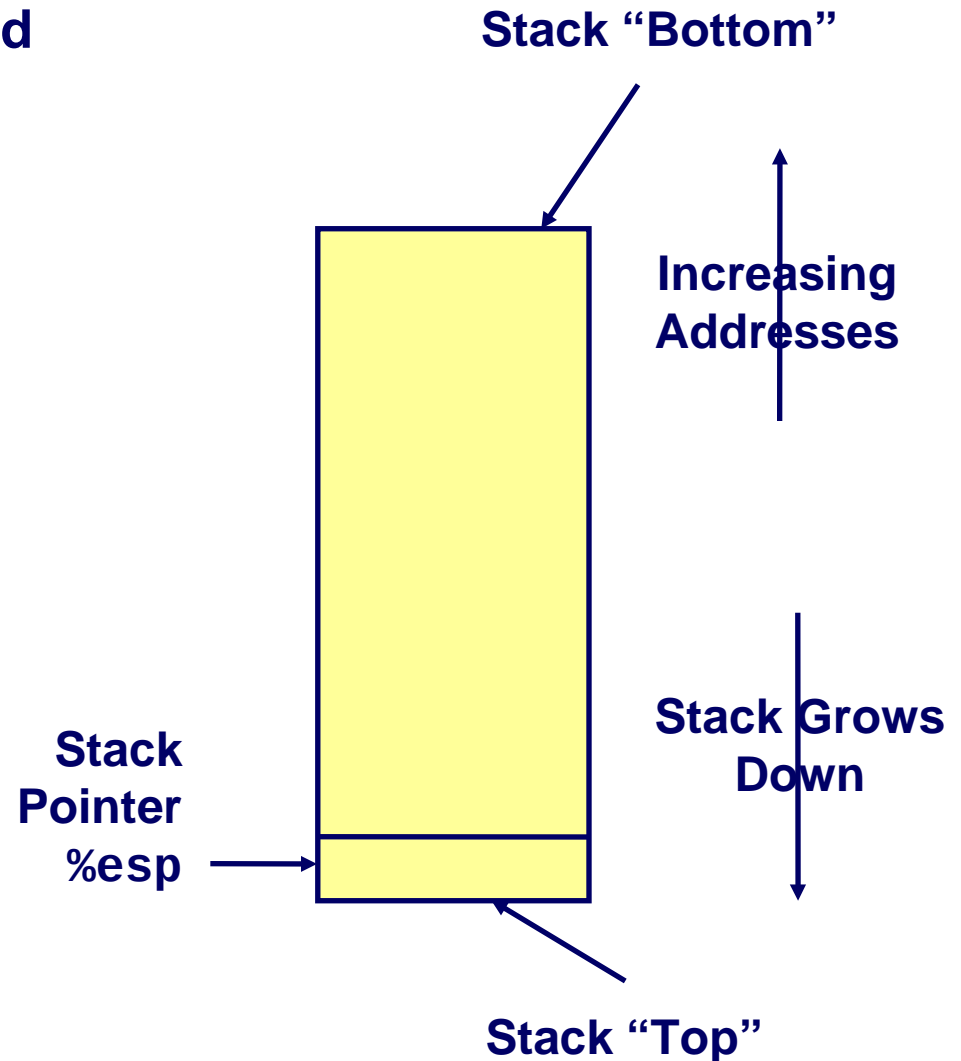
Procedures

Procedure: a unit of code that we can call

- Depending on the programming language, it may be called a procedure, function, subroutine, or method
- A call is like a jump, except it can return
- First of all, we have to understand how a stack works...

IA32 Stack

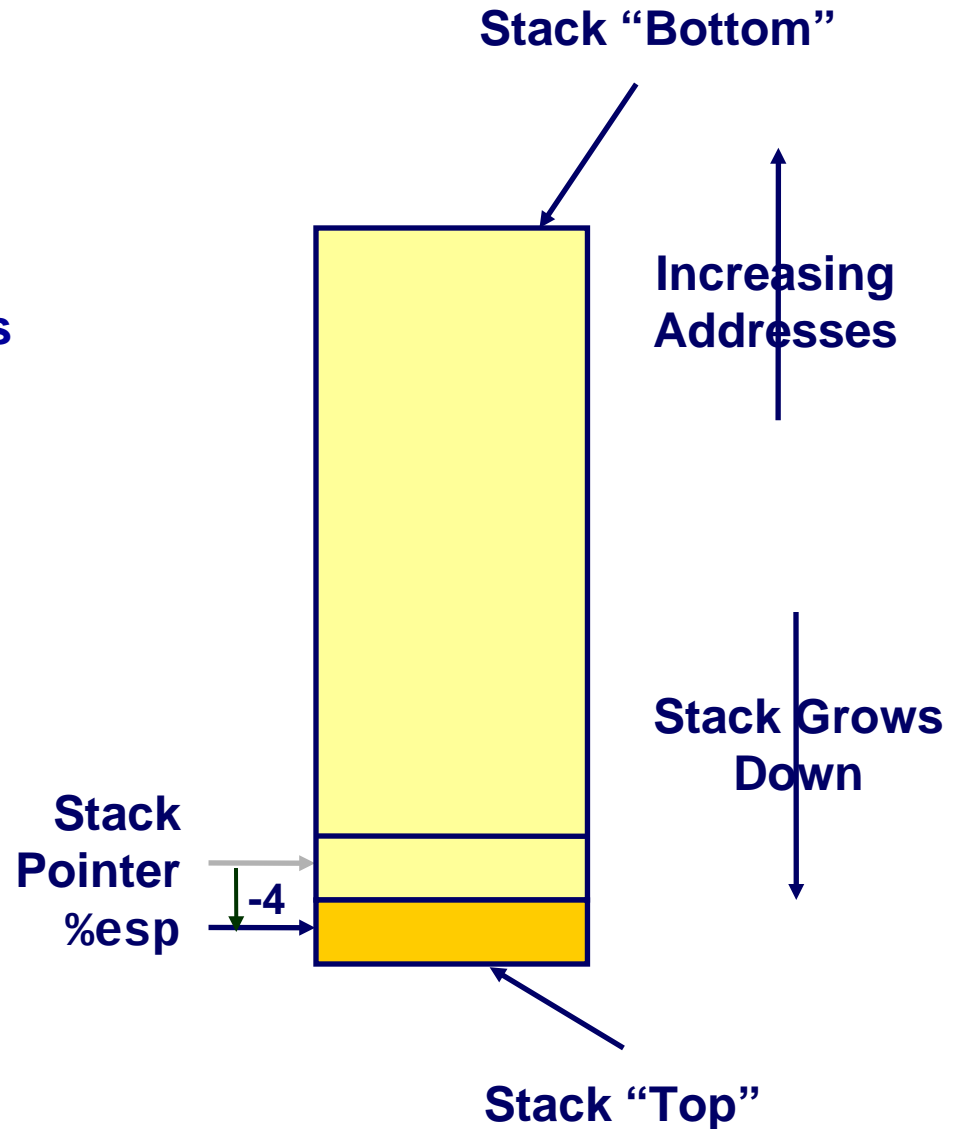
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
 - address of top element



IA32 Stack Pushing

Pushing

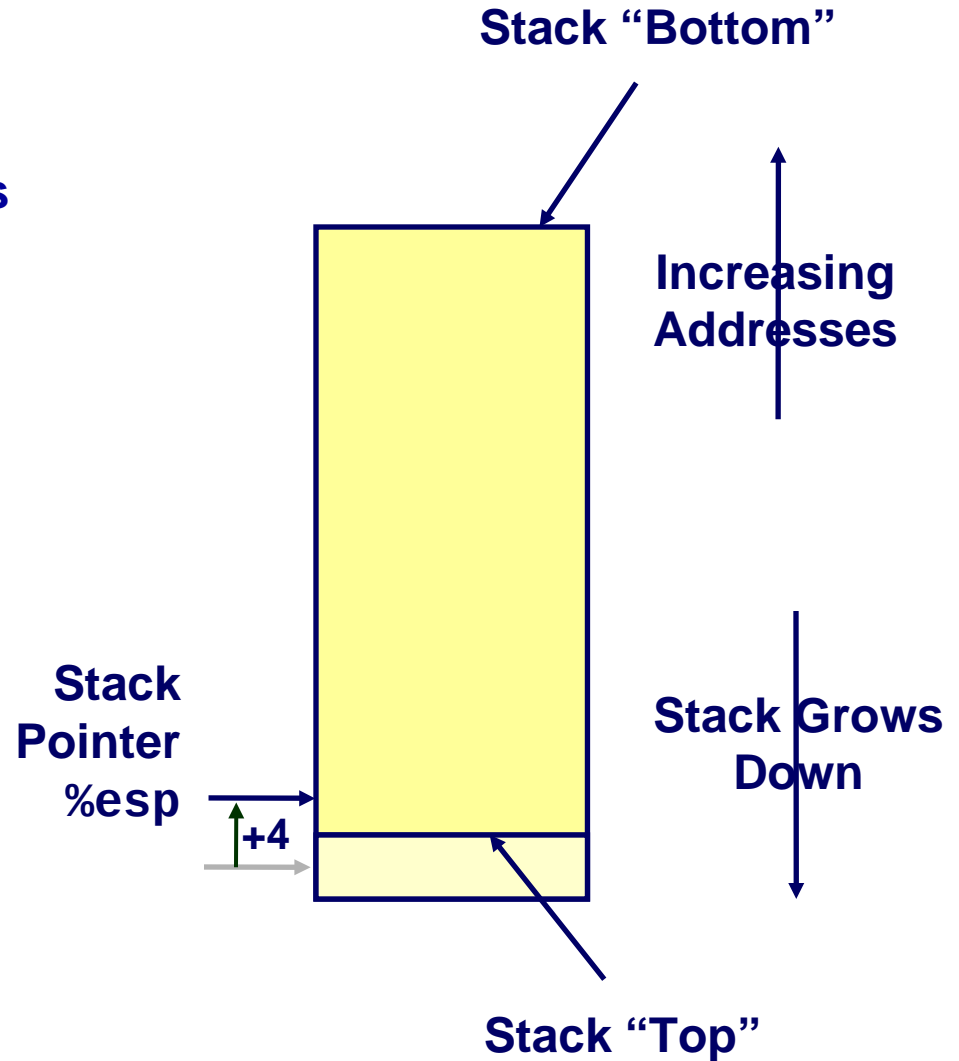
- `pushl Src`
 - Decrement `%esp` by 4
 - Fetch operand at `Src`
 - Write operand at address given by `%esp`
- e.g. `pushl %eax`
`subl $4, %esp`
`movl %eax, (%esp)`



IA32 Stack Popping

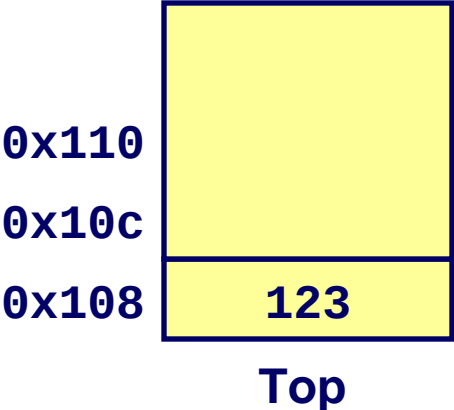
Popping

- `popl Dest`
 - Read operand at address given by `%esp`
 - Write to `Dest`
 - Increment `%esp` by 4
- e.g. `popl %eax`
`movl (%esp), %eax`
`addl $4, %esp`

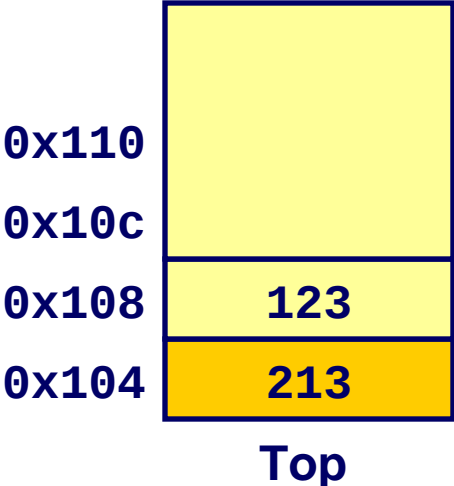


Stack Operation Examples

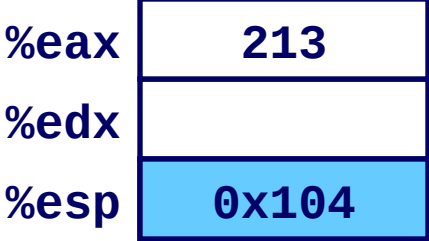
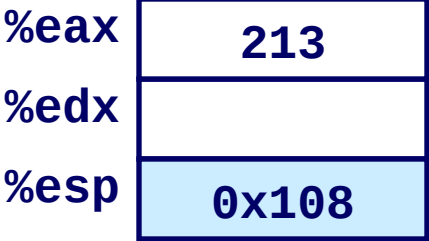
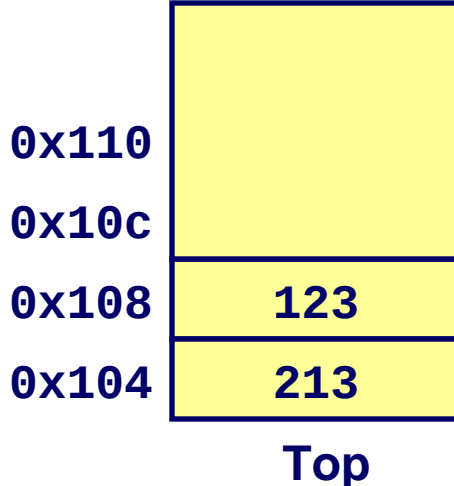
Initially



pushl %eax



popl %edx



Procedure Control Flow

Procedure call:

`call label`

- Push address of next instruction (after the call) on stack
- Jump to *label*

Procedure return:

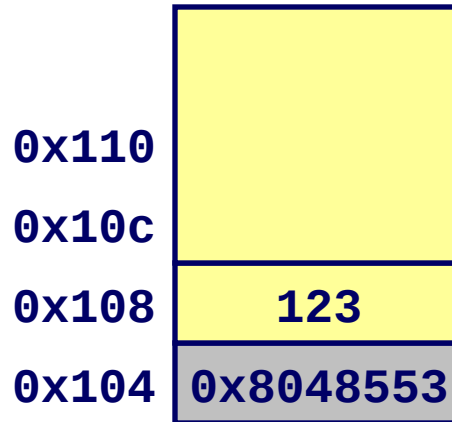
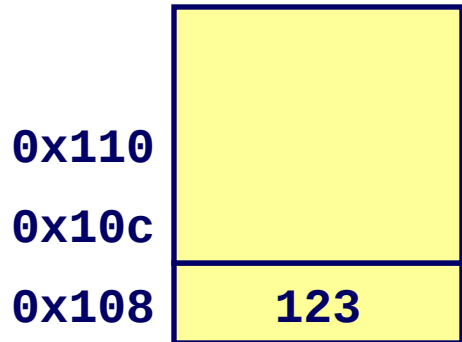
- `ret` Pop address from stack into eip register

Procedure Call Example

804854e: e8 3d 06 00 00
8048553: 50

call 8048b90 <main>
next instruction

call 8048b90



%esp 0x108

%esp 0x104

%eip 0x804854e

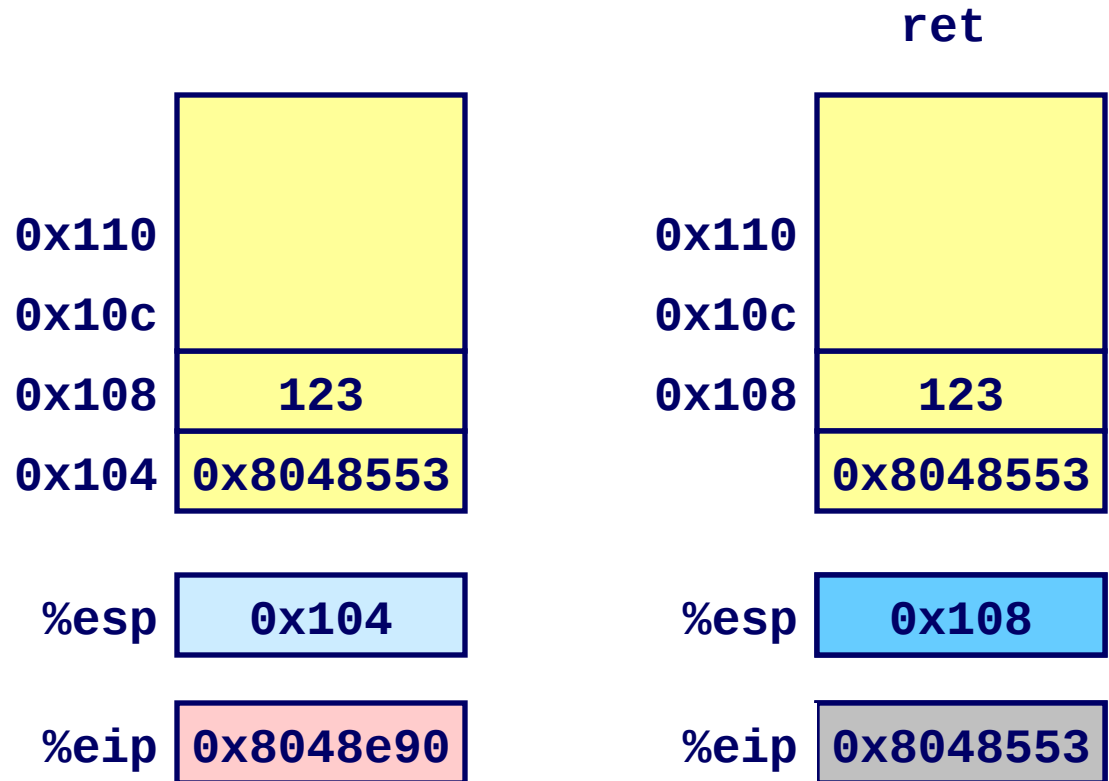
%eip 0x8048b90

%eip is program counter

Procedure Return Example

8048e90: c3

ret



%eip is program counter

Procedure Control Flow

When procedure foo calls who:

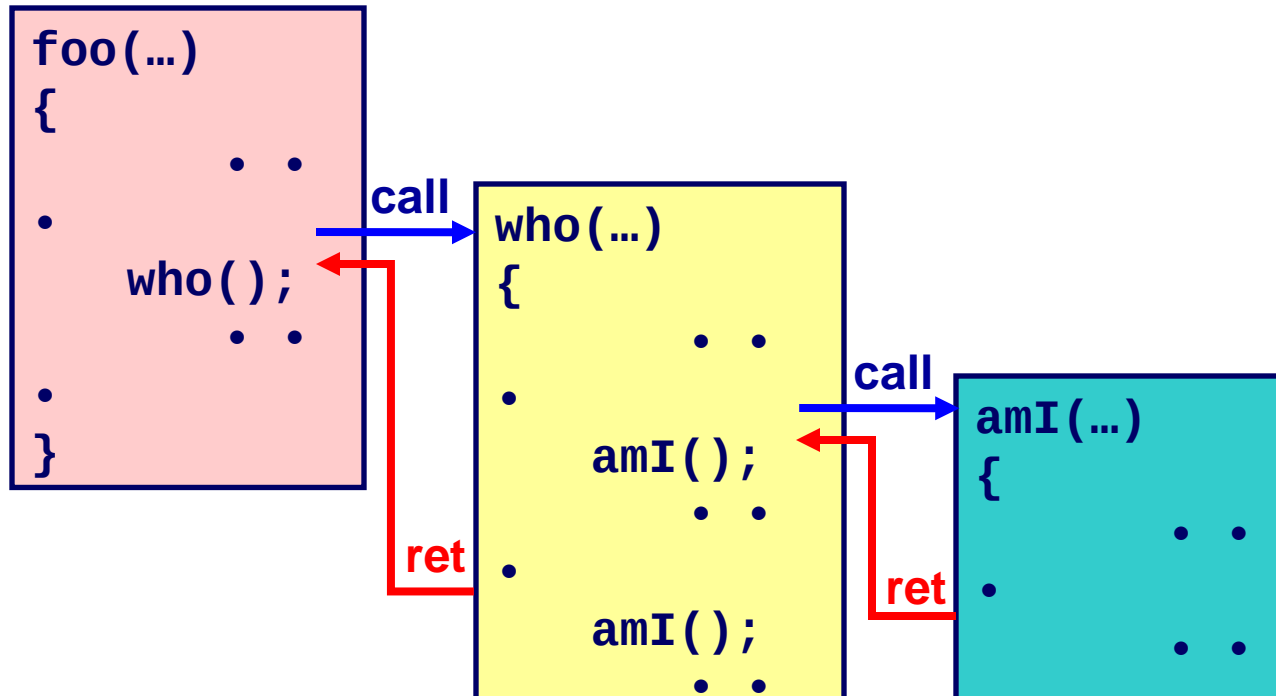
- foo is the *caller*, who is the *callee*
- Control is transferred to the 'callee'

When procedure returns

- Control is transferred back to the 'caller'

Last-called, first-return (LIFO) order

- Naturally implemented via the stack



Procedure calls and stack frames

How does the 'callee' know where to return later?

- Return address placed in a well-known location on stack within a "stack frame"

How are arguments passed to the 'callee'?

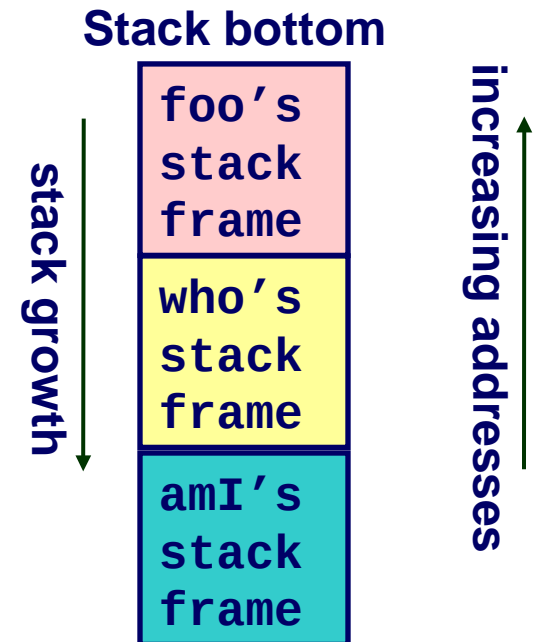
- Arguments placed in a well-known location on stack within a "stack frame"

Upon procedure invocation

- Stack frame created for the procedure
- Stack frame is pushed onto program stack

Upon procedure return

- Its frame is popped off of stack
- Caller's stack frame is recovered



Call chain: foo => who => amI

Keeping track of stack frames

The stack pointer (%esp) moves around

- Can be changed within procedure
- Problem
 - How can we consistently find our parameters?
- The *base pointer* (%ebp)
 - Points to the base of our current stack frame
 - Also called the frame pointer
 - Within each function, %ebp stays constant

Most information on the stack is referenced relative to the base pointer

- Base pointer setup is the programmer's job
 - Actually usually the compiler's job

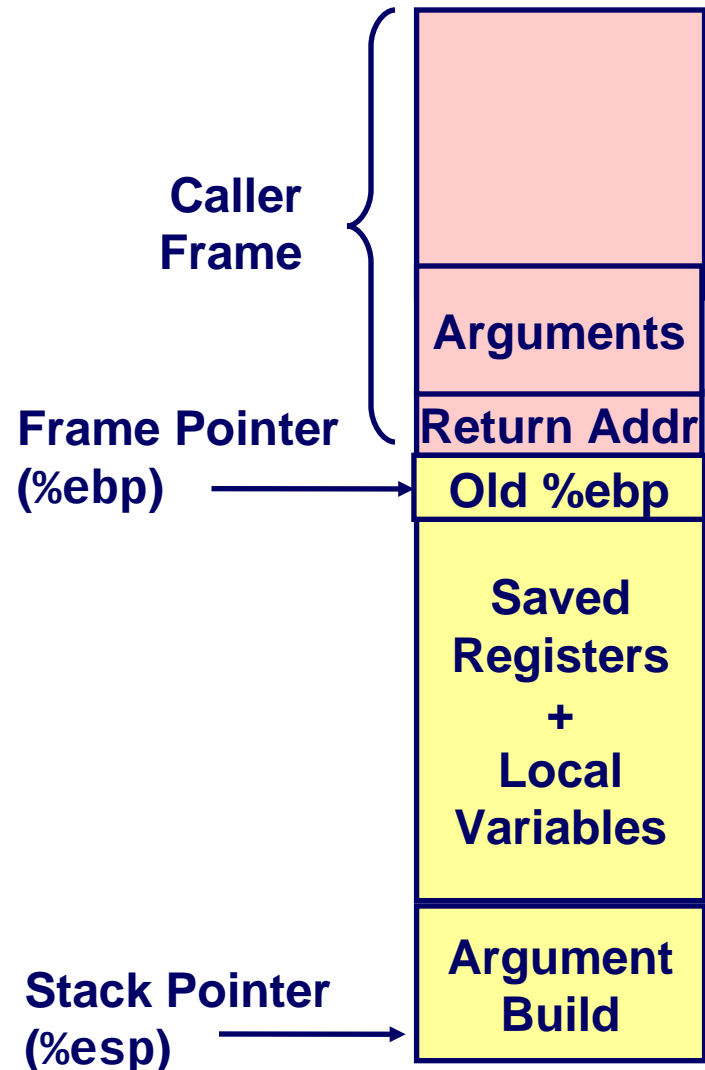
IA32/Linux Stack Frame

Current Stack Frame (Yellow) (From Top to Bottom)

- Parameters for function about to be called
 - “Argument build” of caller
- Local variables
 - If can't keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame (Pink)

- Return address
 - Pushed by `call` instruction
- Arguments for this call
 - “Argument build” of callee
- etc...



swap

```
int zip1 = 15213;
int zip2 = 91125;

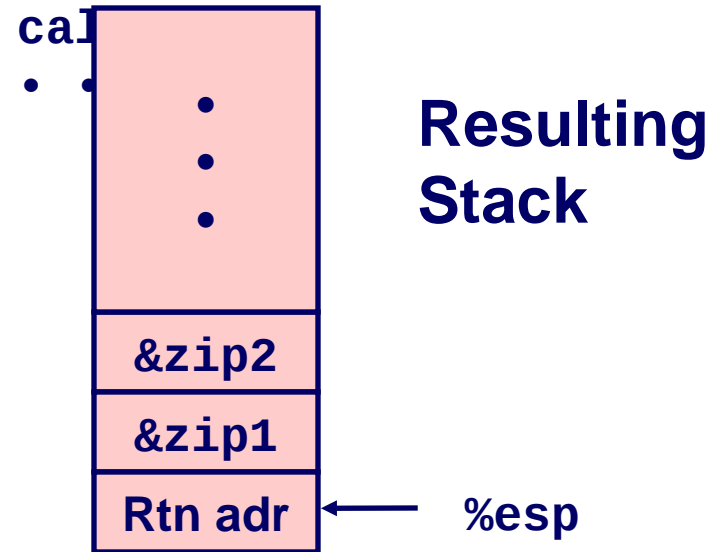
void call_swap()
{
    swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

call_swap:

```
    . . .
    pushl $zip2    # Global
Var
    pushl $zip1    # Global
Var
```



swap

swap:

```
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
} Setup

    movl 12(%ebp),
%ecx
    movl 8(%ebp),
%edx
} Body

    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
} Finish

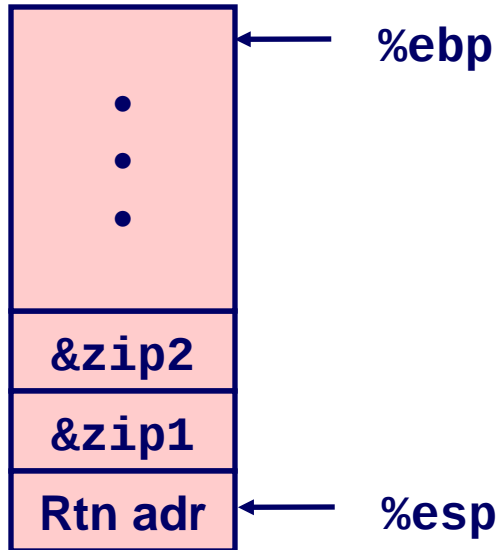
    movl -4(%ebp),
%ebx

    movl %ebp,%esp
    popl %ebp
    ret
```

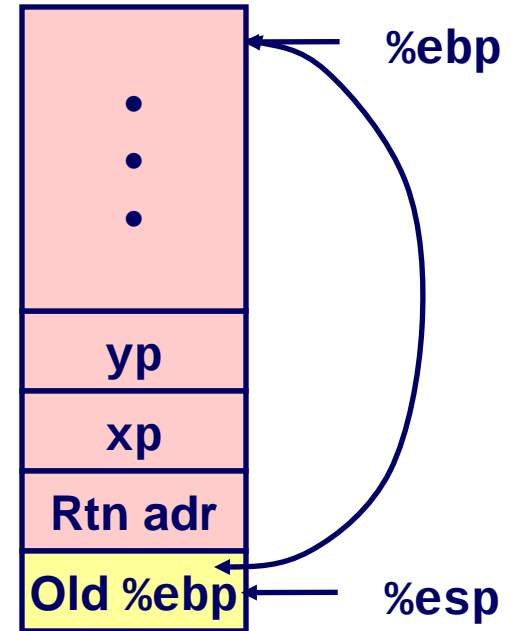
```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap Setup #1

Entering Stack



Resulting stack

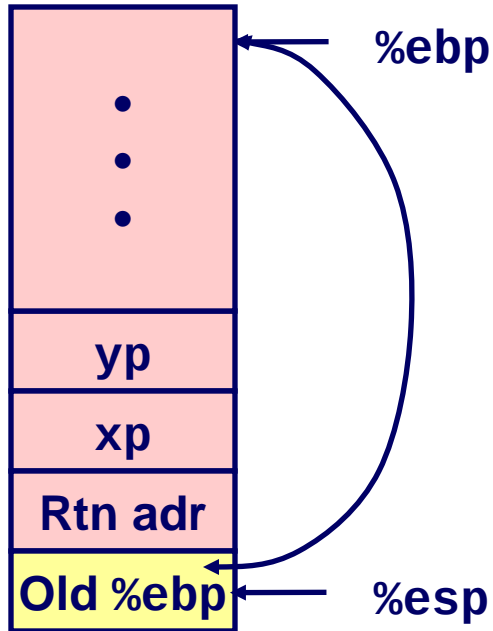


swap:

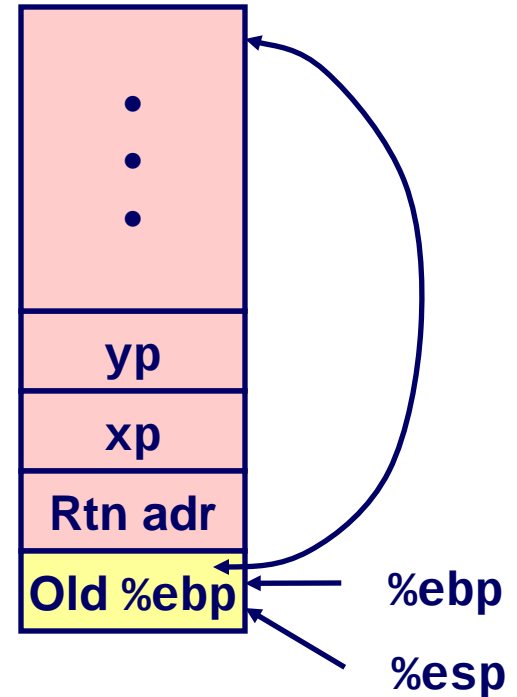
```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

swap Setup #2

Stack before instruction



Resulting stack

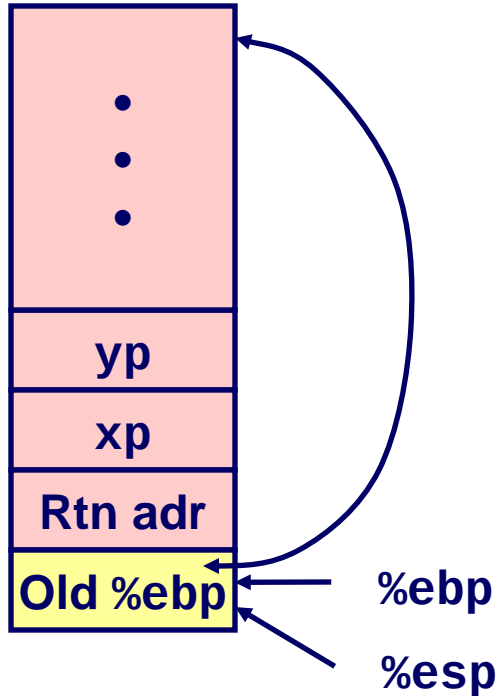


swap:

```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

swap Setup #3

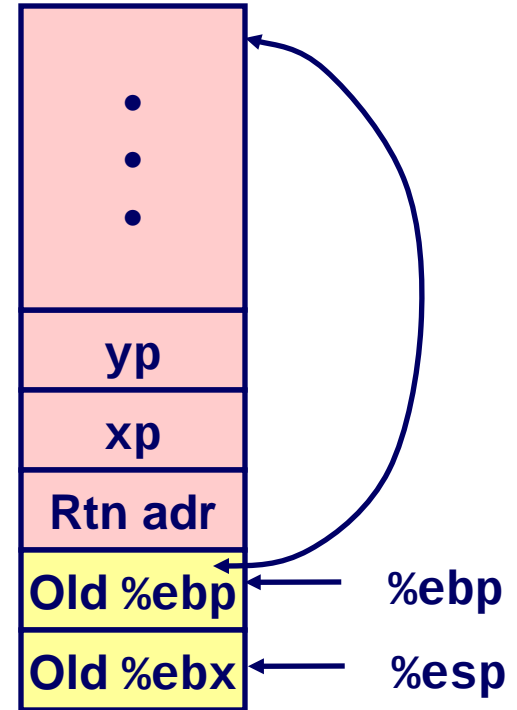
Stack before instruction



swap:

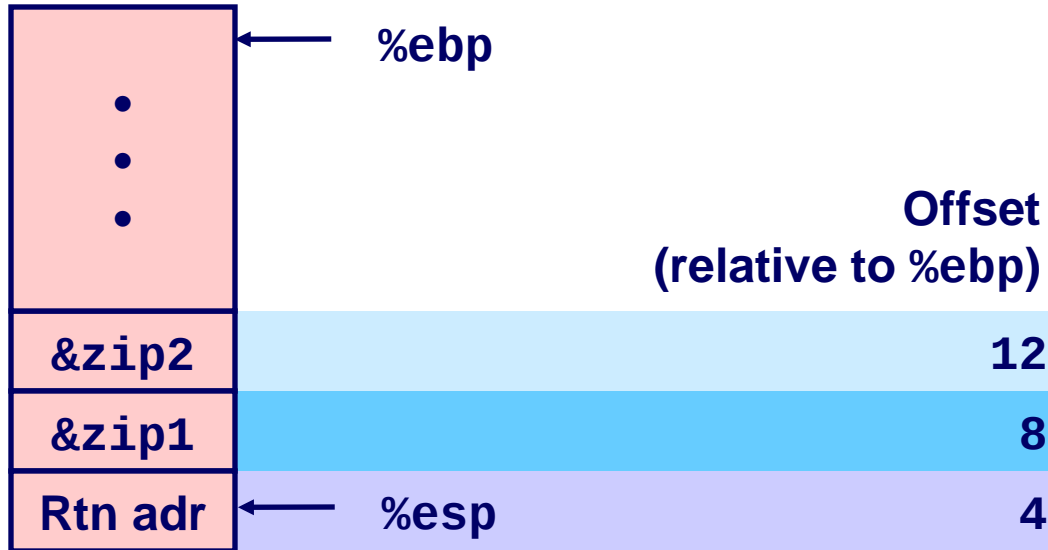
```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

Resulting Stack

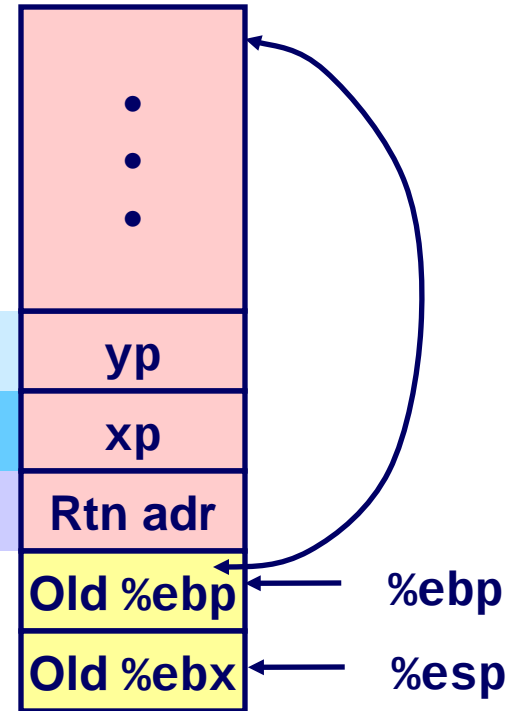


Effect of swap Setup

Entering Stack



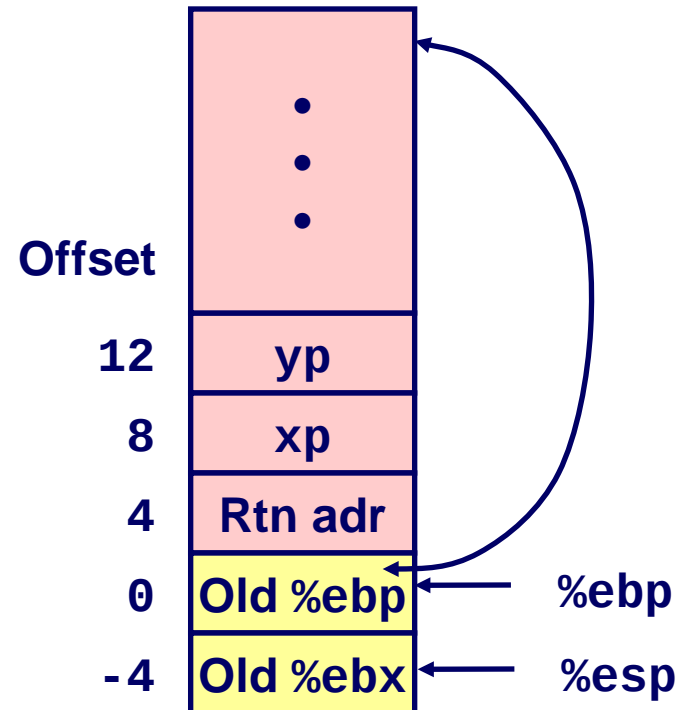
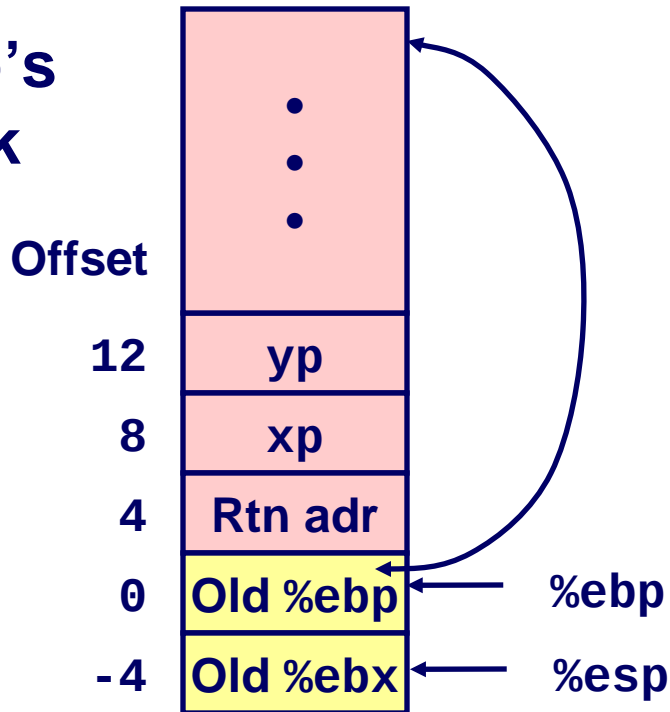
Resulting Stack



```
movl 12(%ebp),%ecx # get yp  
movl 8(%ebp),%edx # get xp  
... } Body
```

swap Finish #1

swap's
Stack



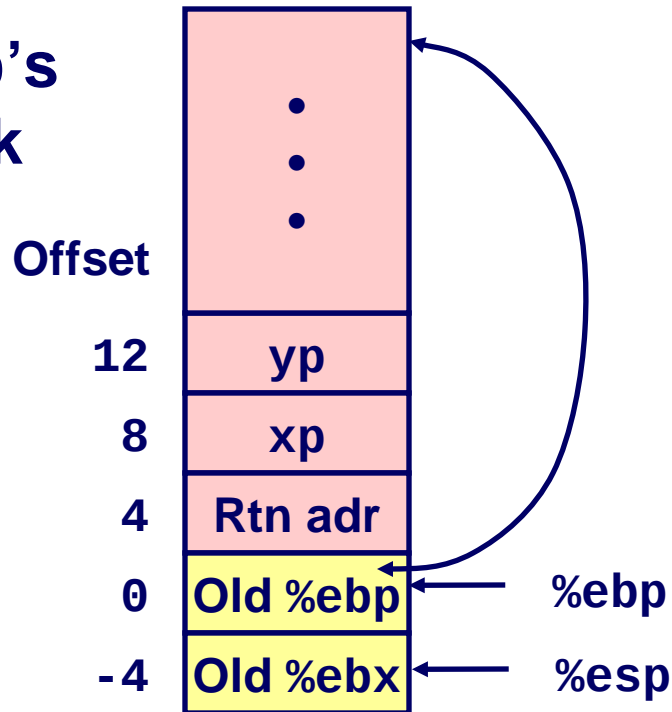
```
movl -4(%ebp), %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```

Observation

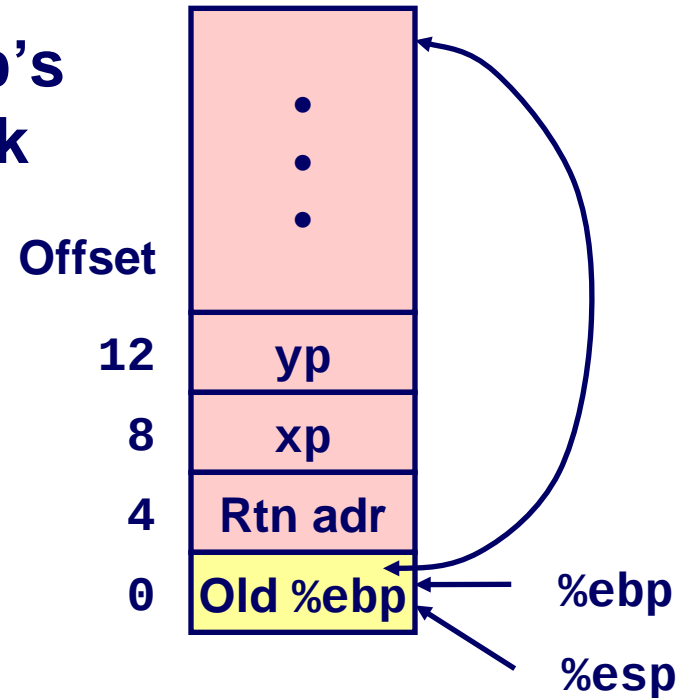
- Saved & restored register `%ebx`

swap Finish #2

swap's
Stack



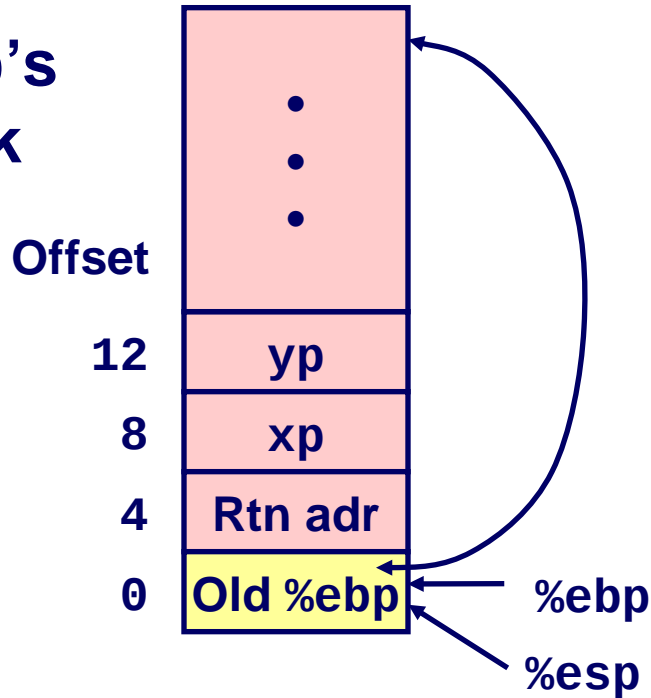
swap's
Stack



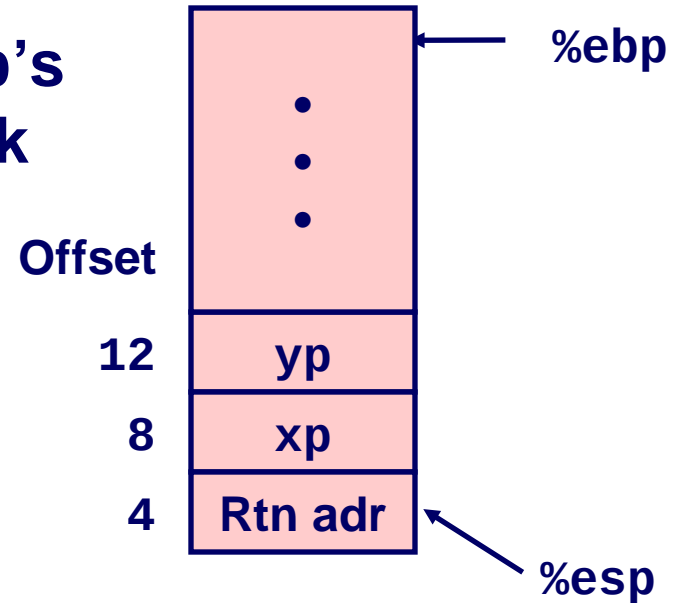
```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

swap's
Stack

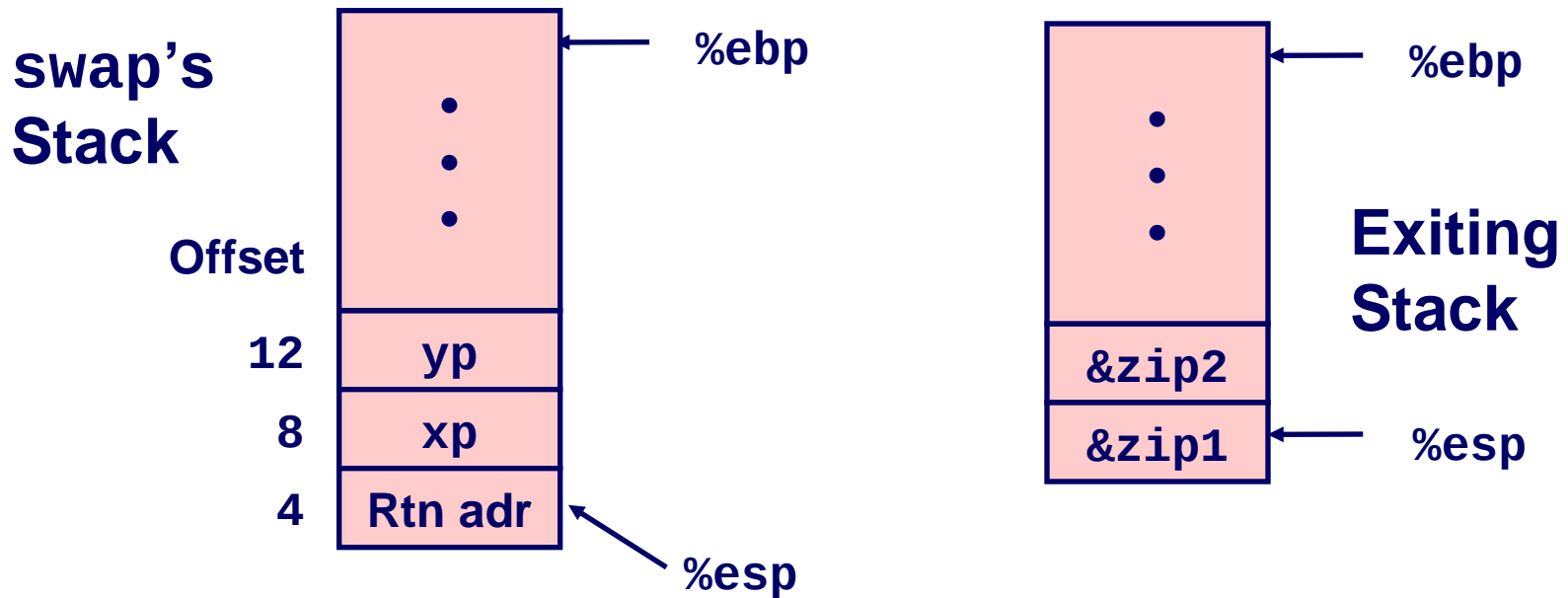


swap's
Stack



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4



Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

Setup

Save old %ebp of caller frame
Set new %ebp for callee (current) frame
Save state of %ebx register from caller

```
movl 12(%ebp),
movl 8(%ebp),
```

Body

Retrieve parameter yp from caller frame
Retrieve parameter xp from caller frame

%ecx

%edx

```
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax, (%edx)
movl %ebx, (%ecx)
```

Perform swap

Finish

Restore the state of caller's %ebx register
Set stack pointer to bottom of callee frame (%ebp)
Restore %ebp to original state

%ebx

```
movl -4(%ebp),
```

Pop return address from stack to %eip

```
movl %ebp,%esp
popl %ebp
```

Equivalent to single leave instruction

Local variables

Where are they in relation to ebp?

- Stored “above” %ebp (at lower addresses)

How are they preserved if the current function calls another function?

- Compiler updates %esp beyond local variables before issuing “call”

What happens to them when the current function returns?

- Are lost (i.e. no longer valid)

Register Saving Conventions

When procedure foo calls who:

- foo is the *caller*, who is the *callee*

Can Register be Used for Temporary Storage?

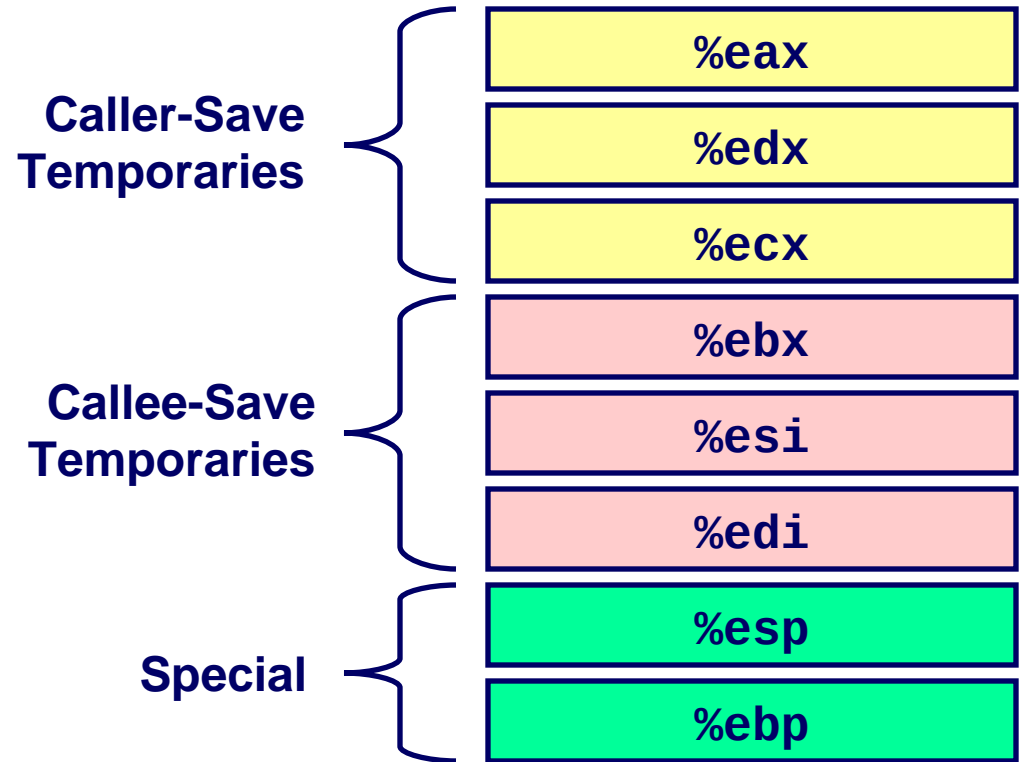
Conventions

- “Caller Save”
 - Caller saves temporary in its frame before calling
- “Callee Save”
 - Callee saves temporary in its frame before using

IA32 Register Usage

Integer Registers

- Two have special uses
%ebp, %esp
- Three managed as callee-save
 - Old values saved on stack prior to using
- Three managed as caller-save
 - Do what you please, but expect any callee to do so, as well
- Return value in %eax



Function pointers

Pointers in C can also point to *code* locations

- **Function pointers**
 - Store and pass references to code

Some uses

- **Dynamic “late-binding” of functions**
 - Dynamically “set” a random number generator
 - Replace large switch statements for implementing dynamic event handlers
 - » Example: dynamically setting behavior of GUI buttons
- **Emulating “virtual functions” and polymorphism from OOP**
 - `qsort()` with user-supplied callback function for comparison
 - » `man qsort`
 - Operating on lists of elements
 - » multiplication, addition, min/max, etc.

Malware leverages this to execute its own code

Using pointers to functions

```
// function prototypes
int doEcho(char*);
int doExit(char*);
int doHelp(char*);
int setPrompt(char*);

// dispatch table section
typedef int (*func)(char*);

typedef struct{
    char* name;
    func function;
} func_t;

func_t func_table[] =
{
    { "echo",    doEcho },
    { "exit",    doExit },
    { "quit",    doExit },
    { "help",    doHelp },
    { "prompt",  setPrompt },
};

#define cntFuncs
(sizeof(func_table) / sizeof(func_table[0]))
```

```
// find the function and dispatch it
for (i = 0; i < cntFuncs; i++) {
    if (strcmp(command, func_table[i].name)==0){
        done = func_table[i].function(argument);
        break;
    }
}
if (i == cntFuncs)
    printf("invalid command\n");
```

Uses in operating system

Interrupt descriptor table

- Pointers to interrupt handler functions
- IDTR points to IDT

System services descriptor table

- Pointers to system call functions

Import address table

- Pointers to imported library calls

Malware attacks all of these

More disassembly

Chapter 8: Reversing

Code patterns in assembly



- Calling conventions (fast vs. standard vs. cdecl)
- ebp omission
- ecx use as C++ this pointer
- C++ vtables (virtual function table)
- WinXP SP2 prologue with patching support
 - For detours
- Exception handlers (FS register)
 - Linked list of functions stored in exception frames on stack

Advanced disassembly

Windows examples

- Largely the same with small modifications
- Size of operands (i.e. dword) specified (not in operator suffix)
- Reverse ordering of operands

Disassembly example

0000	mov ecx, 5		for(int i=0;i<5;i++)
0003	push aHello		{
0009	call printf		printf("Hello");
000E	loop 00000003h		}
0014	...		
0000	cmp ecx, 100h		if(x == 256)
0003	jnz 001Bh		{
0009	push aYes		printf("Yes");
000F	call printf		}
0015	jmp 0027h		else
001B	push aNo		{
0021	call printf		printf("No");
0027	...		}

Disassembly example

```
int main(int argc, char **argv)
{
    WSADATA wsa;
    SOCKET s;
    struct sockaddr_in name;
    unsigned char buf[256];

    // Initialize Winsock
    if(WSAStartup(MAKEWORD(1,1), &wsa))
        return 1;

    // Create Socket
    s = socket(AF_INET, SOCK_STREAM, 0);

    if(INVALID_SOCKET == s)
        goto Error_Cleanup;

    name.sin_family = AF_INET;
    name.sin_port = htons(PORT_NUMBER);
    name.sin_addr.S_un.S_addr = htonl(INADDR_ANY);

    // Bind Socket To Local Port
    if(SOCKET_ERROR == bind(s, (struct
        sockaddr*)&name, sizeof(name)))
        goto Error_Cleanup;

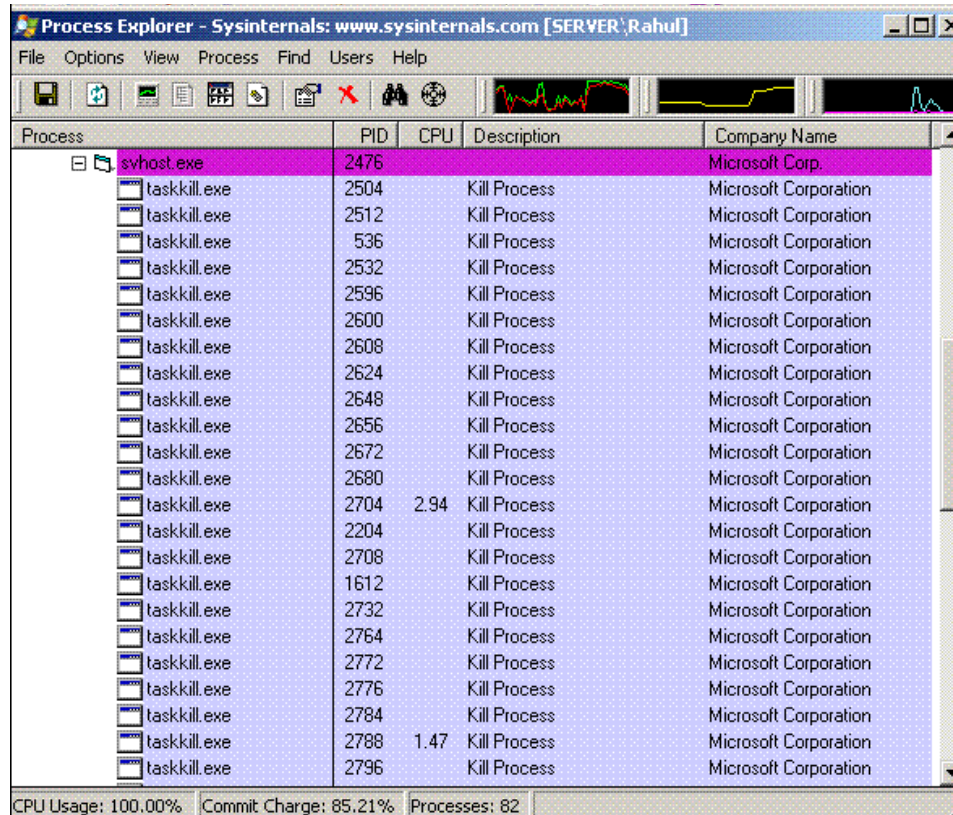
    // Set Backlog parameters
    if(SOCKET_ERROR == listen(s, 1))
        goto Error_Cleanup;
```

```
push ebp
mov ebp, esp
sub esp, 2A8h
lea eax, [ebp+0FFFFFFE70h]
push eax
push 101h
call 4012BEh
test eax, eax
jz 401028h
mov eax, 1
jmp 40116Fh
push 0
push 1
push 2
call 4012B8h
mov dword ptr [ebp+0FFFFFFE6Ch], eax
cmp dword ptr [ebp+0FFFFFFE6Ch], byte
0FFh
jnz 401047h
jmp 401165h
mov word ptr [ebp+0FFFFFFE5Ch], 2
push 800h
call 4012B2h
mov word ptr [ebp+0FFFFFFE5Eh], ax
push 0
call 4012ACh
mov dword ptr [ebp+0FFFFFFE60h], eax
push 10h
lea ecx, [ebp+0FFFFFFE5Ch]
push ecx
mov edx, [ebp+0FFFFFFE6Ch]
push edx
call 4012A6h
cmp eax, byte 0FFh
jnz 40108Dh
jmp 401165h
push 1
mov eax, [ebp+0FFFFFFE6Ch]
push eax
call 4012A0h
cmp eax, byte 0FFh
jnz 4010A5h
jmp 401165h
```

More code snippets

Kills anti-virus, zone-alarm, firewall processes

```
text:00403212      lea     ecx, [ebp-34h]
text:00403215      call   ds:__vbaFreeObj
text:0040321B      mov     edi, ds:__vbaVarDup
text:00403221      lea     edx, [ebp-54h]
text:00403224      lea     ecx, [ebp-44h]
text:00403227      mov     dword ptr [ebp-4Ch], offset aTaskkillImBkav ; "taskkill /im bkav2006.exe"
text:0040322E      mov     dword ptr [ebp-54h], 8
```



The screenshot shows the Process Explorer application window. The title bar reads "Process Explorer - Sysinternals: www.sysinternals.com [SERVER\Rahul]". The menu bar includes "File", "Options", "View", "Process", "Find", "Users", and "Help". The main window displays a list of processes in a table format. The table has columns for "Process", "PID", "CPU", "Description", and "Company Name". The "svchost.exe" process is expanded, showing a list of "taskkill.exe" processes. The CPU usage for these processes is mostly 0.00%, with a few showing 2.94% and 1.47%. The status bar at the bottom indicates "CPU Usage: 100.00%", "Commit Charge: 85.21%", and "Processes: 82".

Process	PID	CPU	Description	Company Name
svchost.exe	2476			Microsoft Corp.
taskkill.exe	2504		Kill Process	Microsoft Corporation
taskkill.exe	2512		Kill Process	Microsoft Corporation
taskkill.exe	536		Kill Process	Microsoft Corporation
taskkill.exe	2532		Kill Process	Microsoft Corporation
taskkill.exe	2596		Kill Process	Microsoft Corporation
taskkill.exe	2600		Kill Process	Microsoft Corporation
taskkill.exe	2608		Kill Process	Microsoft Corporation
taskkill.exe	2624		Kill Process	Microsoft Corporation
taskkill.exe	2648		Kill Process	Microsoft Corporation
taskkill.exe	2656		Kill Process	Microsoft Corporation
taskkill.exe	2672		Kill Process	Microsoft Corporation
taskkill.exe	2680		Kill Process	Microsoft Corporation
taskkill.exe	2704	2.94	Kill Process	Microsoft Corporation
taskkill.exe	2204		Kill Process	Microsoft Corporation
taskkill.exe	2708		Kill Process	Microsoft Corporation
taskkill.exe	1612		Kill Process	Microsoft Corporation
taskkill.exe	2732		Kill Process	Microsoft Corporation
taskkill.exe	2764		Kill Process	Microsoft Corporation
taskkill.exe	2772		Kill Process	Microsoft Corporation
taskkill.exe	2776		Kill Process	Microsoft Corporation
taskkill.exe	2784		Kill Process	Microsoft Corporation
taskkill.exe	2788	1.47	Kill Process	Microsoft Corporation
taskkill.exe	2796		Kill Process	Microsoft Corporation

More code snippets

New variants

- Download worm update files and register them as services
- regsvr32 MSINET.OCX
 - Internet Transfer ActiveX Control

```
text:004021C8      lea     edx, [ebp-4Ch]
text:004021CB      lea     ecx, [ebp-3Ch]
text:004021CE      mov     dword ptr [ebp-44h], offset aRegsvr32Msinet ; "regsvr32 MSINET.OCX"
text:004021D5      mov     dword ptr [ebp-4Ch], 8
-----
```

- Check for updates

```
text:00401C74 aHttpGiftshop_v:
text:00401C74      unicode 0, <http://gift[REDACTED]/update.txt>,0
```

Done with review..

Run-time data structures

Segment Name	Segment Description
.text	This segment contains the executable instructions and is shared among every process running the same binary. This segment usually has READ and EXECUTE permissions only. This section is the one most affected by optimization.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rdata	Sometimes known as .rodata (read-only data) segment. This contains constants and string literals.
.bss	BSS stands for "Block Started by Symbol." It holds un-initialized global and static variables. Since the BSS only holds variables that don't have any values yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but the BSS (unlike the data segment) doesn't take up any actual space in the object file.
.reloc	Stores the information required for relocating the image while loading.
Heap	<p>The heap area is for dynamically allocated memory (<code>malloc()</code>, <code>realloc()</code>, <code>calloc()</code>) and is accessed through a pointer. Everything on a heap is anonymous, thus you can only access parts of it through a pointer. A <code>malloc()</code> request may be rounded up in size to some convenient power of two. Freed memory goes back to the heap, but there is no easy way to give up that memory back to the OS. The heap usually grows up toward the stack.</p> <p>The end of the heap is marked by a pointer known as the "break." You cannot reference past the break. You can, however, move the break pointer (via <code>brk</code> and <code>sbrk</code> system calls) to a new position to increase the amount of heap memory available. This is usually done automatically for you by the system if you use <code>malloc</code> often enough.⁹</p>
Stack	<p>The stack holds local (automatic) variables, temporary information, function parameters, and the like. It acts like a LIFO (Last In First Out) object as it grows downward toward the heap.</p> <p>When a function is called, a stack frame (or a procedure activation record) is created and PUSHed onto the top of the stack. This stack frame contains information such as the address from which the function was called (and where to jump back to when the function is finished (return address)), parameters, local variables, and any other information needed by the invoked function. The order of the information varies by system and compiler, but on Solaris it is described in <code>/usr/include/sys/frame.h</code>. When a function returns, the stack frame is POPped from the stack. The current instruction that is running is pointed to by the IP (Instruction Pointer). The address of the next instruction is held in the PC (Program Counter).</p>

Machine Instruction Example

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

```
0x401046:    03 45 08
```

C Code

- Add two signed integers

Assembly

- Add 2 4-byte integers
 - “Long” words in GCC parlance
 - Same instruction whether signed or unsigned
 - Operands:
 - x: Register%eax
 - y: Memory M[%ebp+8]
 - t: Register%eax
- » Return function value in %eax

Object Code

- 3-byte instruction
- Stored at address 0x401046

Tools for disassembling

objdump

`objdump -d <object_file>`

- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either executable or relocatable (.o) file

gdb Debugger

`gdb p`

`disassemble sum`

- Disassemble procedure

`x/13b sum`

- Examine the 13 bytes starting at sum

Extended Example: simple.c

```
gcc -O2 -c simple.c
```

```
int simple(int *xp, int y)
{
    int t = *xp + y;
    *xp = t;
    return t;
}
```

`_simple:`

<code>pushl %ebp</code>	Setup stack frame pointer
<code>movl %esp, %ebp</code>	
<code>movl 8(%ebp), %edx</code>	get xp
<code>movl 12(%ebp), %ecx</code>	get y
<code>movl (%edx), %eax</code>	move *xp to t
<code>addl %ecx, %eax</code>	add y to t
<code>movl %eax, (%edx)</code>	store t at *xp
<code>popl %ebp</code>	restore frame pointer
<code>ret</code>	return to caller

C, x86 example

```
int a = 1, b = 3, c;  
if (a > b)  
    c = a;  
else  
    c = b;
```

```
00000018: C7 45 FC 01 00 00 00      mov dword ptr [ebp-4],1    ; store a = 1  
0000001F: C7 45 F8 03 00 00 00      mov dword ptr [ebp-8],3    ; store b = 3  
00000026: 8B 45 FC                   mov eax,dword ptr [ebp-4]  ; move a into EAX register  
00000029: 3B 45 F8                   cmp eax,dword ptr [ebp-8]  ; compare a with b (subtraction)  
0000002C: 7E 08                       jle 00000036               ; if (a<=b) jump to line 00000036  
0000002E: 8B 4D FC                   mov ecx,dword ptr [ebp-4]  ; else move 1 into ECX register &&  
00000031: 89 4D F4                   mov dword ptr [ebp-0Ch],ecx ; move ECX into c (12 bytes down) &&  
00000034: EB 06                       jmp 0000003C               ; unconditional jump to 0000003C  
00000036: 8B 55 F8                   mov edx,dword ptr [ebp-8]  ; move 3 into EDX register &&  
00000039: 89 55 F4                   mov dword ptr [ebp-0Ch],edx ; move EDX into c (12 bytes down)
```

More code snippets

Registry modifications for disabling task manager and changing browser default page

```
text:00402B58                                     ; .text:00403E1C↓  
text:00402B58                                     unicode 0, <Software\Microsoft\Windows\CurrentVersion\Po>  
text:00402B58                                     unicode 0, <licies\System>,0  
text:00402BCC                                     unicode 0, <(>,0  
text:00402BD0 aDisableregistr:                   ; DATA XREF: .text:00403C49↓  
text:00402BD0                                     unicode 0, <DisableRegistryTools>,0
```

HKEY_CURRENT_USER\Software\Policies\Microsoft\Internet Explorer\Control Panel,Homepage

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\SystemDisableRegistryTools

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\MainStart Page

HKEY_CURRENT_USER\Software\Yahoo\pager\View\YMSGR_buzz content url

HKEY_CURRENT_USER\Software\Yahoo\pager\View\YMSGR_Launchcast DisableTaskMgr

