Stack and Heap Allocation
Any program you run has, associated with it, some memory which is divided into:

- Code Segment
- Data Segment (Holds Global Data)
- Stack (where the local variables and other temporary information is stored)
- Heap

The Heap grows downwards

The Stack grows upwards

Diagram:

- Code Segment
- Data Segment
- Stack
- Heap
When we have a declaration of the form “int a;”:

- a variable with identifier “a” and some memory allocated to it is created in the stack. The attributes of “a” are:
  - Name: a
  - Data type: int
  - Scope: visible only inside the function it is defined, disappears once we exit the function
  - Address: address of the memory location reserved for it. Note: Memory is allocated in the stack for a even before it is initialized.
  - Size: typically 4 bytes
  - Value: Will be set once the variable is initialized

Since the memory allocated for the variable is set in the beginning itself, we cannot use the stack in cases where the amount of memory required is not known in advance. This motivates the need for HEAP
We know what a pointer is. Let us say we have declared a pointer “int *p;” The attributes of “a” are:

- Name: p
- Data type: Integer address
- Scope: Local or Global
- Address: Address in the data segment or stack segment
- Size: 32 bits in a 32-bit architecture

We saw how a fixed memory allocation is done in the stack, now we want to allocate dynamically. Consider the declaration:

- “int *p;”. So the compiler knows that we have a pointer p that may store the starting address of a variable of type int.
- To point “p” to a dynamic variable we need to use a declaration of the type “ p = new int;”
Dynamic variables are never initialized by the compiler, so it is a good practice to initialize it.

In more compact notation:

```cpp
int *p;
p = new int;
*p = 0;

int *p = new int(0);
```
Internal representation of the earlier declaration would be as follows:

Now we can delete the dynamic variable from the heap using

- “delete p;” Now 2 bytes of memory were freed but the pointer p isnit erased and it can be used to initialize another dynamic variable.
C++ also allows us to allocate arrays in heap. This can be done as follows:

```cpp
int *ap = new int[20];

To initialize the array, we do the following

for (int i = 0; i <= 19; i++)
    ap[i] = 0;
```
Be wary of....

Deleting a dynamic variable that has already been Deleted

```cpp
int *p = new int(0);
delete p;
delete p;
```

Deleting a dynamic variable that has not yet been allocated

```cpp
int *p = new int(0);
```

Assign value to an unallocated dynamic variable

```cpp
char *p;
StringCopy(p, 'Hello');
```

Assigning a value to a deleted dynamic variable

```cpp
char *str = new char[100];
delete [] str;
char *test = new char[100];
strcpy(str, "surprise !");
cout << test;
delete [] test;
```
Compiler Techniques for Single Processor Tuning

An Introduction
Compiler is the primary tool of computer program optimization:

- structure of the compiler and the compilation process
- compiler optimizations
- steering of the compilation process - compiler options
- structure of the run time libraries and scientific libraries
- computational domain and computation accuracy
Compiler manages processor resources:

- registers
- integer/floating-point execution units
- load/store/prefetch for data flow in/out of processor
- the implementation details of processor and system architecture are built into the compiler

User Program (C/C++/Fortran, etc.)
- high level representation

Compilation process

Intermediate representation

Solving:
- data dependencies
- control flow dependencies
- parallelization
- compactification of code
- optimal scheduling of the code

Machine instructions
There are no source-to-source optimizers or parallelizers.
Source code is translated to WHIRL (Winning Hierarchical Intermediate Representation Language);
- same IR for different levels of representation
- whirl2f and whirl2c translates back into Fortran or C from IRs
Inter-Procedural analyzer requires final translation at link time
Compiler Optimizations

• **Global Optimizer:**
  – dead code elimination
  – copy propagation
  – loop normalization
    • stride one loops
    • single induction variable
  – memory alias analysis
  – strength reduction

• **Inter-Procedural Analyzer:**
  – cross-file function inlining
  – dead function elimination
  – dead variable elimination
  – padding of variables in common blocks
  – inter-procedural constant propagation

• **Loop Nest Optimizer:**
  – loop unrolling (outer)
  – loop interchange
  – loop fusion/fission
  – loop blocking
  – memory prefetch
  – padding local variables

• **Automatic Parallelizer**
  – loop level work distribution

• **Code Generator:**
  – software pipelining
  – inner loop unrolling
  – if-conversion
  – read/write optimization
  – recurrence breaking
  – instruction scheduling inside *basic blocks*
Optimization Levels

Compilation speed degrades with higher optimization

- **-O0**: turn off all optimizations
- **-O1**: only local optimizations
- **-O2** or **-O**: extensive but conservative optimizations
- **-O3**: aggressive optimizations, LNO, *software pipelining*
- **-ipa**: inter-procedural analysis (only at -O2 and -O3)
- **-apo**: automatic parallelization option (same as -pfa)
- **-g[0|3]**: debugging switch:
  - **-g0** forces -O0
  - **-g3** to debug with -O3
<table>
<thead>
<tr>
<th>Option</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>-r10000</td>
<td>Generate optimal instruction schedule for the R10000 proc</td>
</tr>
<tr>
<td>-r8000</td>
<td>Generate optimal instruction schedule for the R8000 proc</td>
</tr>
<tr>
<td>-O[0</td>
<td>1</td>
</tr>
<tr>
<td>-Ofast=[ipXX]</td>
<td>Select best optimization for the given architecture</td>
</tr>
<tr>
<td></td>
<td>XX machine (output of the <code>hinv -c processor</code> command)</td>
</tr>
<tr>
<td></td>
<td>27 Origin2000 (all cpu frequencies and cache sizes)</td>
</tr>
<tr>
<td></td>
<td>35 Origin3000 (all cpu frequencies and cache sizes)</td>
</tr>
</tbody>
</table>

Optimizations may differ on the version of the compiler. Currently:

- `-O3 -IPA -TARG:platform=ip27 -n32`
- `-OPT:Olimit=0:roundoff=3:div_split=ON:alias=typed`

*(thus -Ofast switch invokes the Interprocedural Analyzer)*

- `-mp` Enable multi-processing directives
- `-mpio` Support I/O from a parallel region
- `-apo` Invoke automatic parallelization option
<table>
<thead>
<tr>
<th>Option</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d8/d16</td>
<td>Double precision variables as 8 or 16 bytes</td>
</tr>
<tr>
<td>-r8</td>
<td>Convert REAL to REAL<em>8 and COMPLEX to COMPLEX</em>16 (1)</td>
</tr>
<tr>
<td>-i8</td>
<td>Convert INTEGER to INTEGER*8 and LOGICAL to 8 byte sizes (1)</td>
</tr>
<tr>
<td>-static</td>
<td>Local variables will be initialized in fixed locations on the heap (-static_threadprivate makes static variables private to each thread)</td>
</tr>
<tr>
<td>-col[72</td>
<td>120]</td>
</tr>
<tr>
<td>-Dname</td>
<td>Define name for the pre-processor</td>
</tr>
<tr>
<td>-Idir</td>
<td>Define include directory dir</td>
</tr>
<tr>
<td>-alignN</td>
<td>Assume alignment on the N=8,16,32,64,128 bit boundary</td>
</tr>
<tr>
<td>-G0</td>
<td>Put all static data into indirect address area</td>
</tr>
<tr>
<td>-xgot</td>
<td>make big tables for static data and program addresses</td>
</tr>
<tr>
<td>-multigot</td>
<td>Automatic choice of table sizes for static variables and addresses</td>
</tr>
<tr>
<td>-version</td>
<td>Show compiler version</td>
</tr>
<tr>
<td>-show</td>
<td>Put the compiler in verbose mode: all switches are displayed</td>
</tr>
</tbody>
</table>

(1) Note: explicit sizes are preserved, i.e. REAL*4 remains 32 bit
## Options: Debugging

<table>
<thead>
<tr>
<th>Option</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-g</code></td>
<td>Disable optimization and keep all symbol tables</td>
</tr>
<tr>
<td><code>-DEBUG:</code></td>
<td>the DEBUG group option (man DEBUG_GROUP):</td>
</tr>
</tbody>
</table>
| • `check_div=n` | `n=1` (default) check integer divide by zero  
  
  `n=2` check integer overflow  
  
  `n=3` check integer divide by zero and overflow  |
| • `subscript_check` | *(default ON)* to check for subscripts out of range  
  
  C/C++: produces trap #8  
  
  f77: aborts run and dumps core  
  
  f90: aborts run if setenv F90_BOUNDS_CHECK_ABORT |
| • `verbose_runtime` | *(default OFF)* to give source line number of failures  |
| • `trap_uninitialized` | *(default OFF)* initialize all variables to 0xFFFFA5A5  
  
  when used as pointer - access violation  
  
  when used as fp values - NaN causes fp trap |

**Example:**

```
f77 -n32 -mips4 -g file.f  
  -DEBUG:subscript_check:verbose_runtime=ON  
  -DEBUG:check_div=3  -DEBUG:trap_uninitialized=ON
```
1. Produce executable `a.out` with default compilation options:
   
   ```
   f77 source.f
   cc source.f
   ```

   be aware of the defaults setting (e.g. `/etc/compiler.defaults`)
   same flags for Fortran and C

2. Options for debugging:
   
   ```
   f77/cc -o prog -n32 -g -static source.f
   ```

2. Explicit setting of ABI/ISA/Processor, highest opt:
   
   ```
   f77/cc -o prog -n32 -mips4 -r10000 -O3 source.f
   ```

3. Detailed control of the optimization process with the
   group options:
   
   ```
   f77/cc -o prog -64 -mips4 -O3 -Ofast=ip27
   -OPT:round=3:IEEE_arith=3 -IPA:dfe=on ...
   ```
Compiler performs many sophisticated optimizations on the source code under certain assumption about the program. Typically:

- program data is large (does not fit into the cache)
- program does not violate language standard
- program is insensitive to roundoff errors
- all data in the program is alias-ed, unless it can be proved otherwise

If one or more of these assumptions does not hold, compiler should be tuned to the program with the compiler options. Most important:

- OPT for general optimizations assumptions
- LNO for the Loop Nest optimizer options
- IPA for the Inter-Procedural Analyzer options

Additional options that help to tune the compiler properly:

- TENV, TARG for the target machine and environment description
  -TENV:align_aggregates=x (bytes)
- LIST, DEBUG for the listing and debugging options
Compiler is the primary tool of program optimization

• Compilation is the process of lowering the code representation from high level to low, i.e. processor level
• The MipsPro compiler targets the MIPS R1x000 processor and has built in the features of the processor and Origin architecture
• A large number of options exist to steer the compilation process
  – ABI, ISA and optimization options selections
  – setting of assumptions about the program behaviour
• There are optimized and parallelized libraries of subroutines for scientific computation
• When programming for a digital computer, it is important to remember the limitations due to limited validity range of the floating point calculations
15-213
“The course that gives CMU its Zip!”

Linking
February 28, 2008

Topics
- Static linking
- Dynamic linking
- Case study: Library interpositioning
Example C Program

main.c

```c
int buf[2] = {1, 2};
int main()
{
    swap();
    return 0;
}
```

swap.c

```c
extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```
Static Linking

Programs are translated and linked using a **compiler** driver:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```

Source files

```
main.c
```

Translators

(cpp, ccl, as)

```
swap.c
```

Translators

(cpp, ccl, as)

```
main.o
```

Separately compiled relocatable object files

```
swap.o
```

Fully linked executable object file
(contains code and data for all functions defined in main.c and swap.c)

```
P
```
Why Linkers?

Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.

- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library
Why Linkers? (cont)

Reason 2: Efficiency

- Time: Separate Compilation
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.

- Space: Libraries
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.
What Do Linkers Do?

Step 1. Symbol resolution

- Programs define and reference *symbols* (variables and functions):
  - `void swap() {...} /* define symbol swap */`
  - `swap(); /* reference symbol swap */`
  - `int *xp = &x; /* define xp, reference x */`

- Symbol definitions are stored (by compiler) in *symbol table*.
  - Symbol table is an array of structs
  - Each entry includes name, type, size, and location of symbol.

- Linker associates each symbol reference with exactly one symbol definition.
What Do Linkers Do? (cont)

Step 2. Relocation

- Merges separate code and data sections into single sections

- Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.

- Updates all references to these symbols to reflect their new positions.
Three Kinds of Object Files (Modules)

1. Relocatable object file (.o file)
   - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
   - Each .o file is produced from exactly one source (.c) file

2. Executable object file
   - Contains code and data in a form that can be copied directly into memory and then executed.

3. Shared object file (.so file)
   - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
   - Called *Dynamic Link Libraries* (DLLs) by Windows
Executable and Linkable Format (ELF)

Standard binary format for object files

Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux

One unified format for
  - Relocatable object files (.o),
  - Executable object files
  - Shared object files (.so)

Generic name: ELF binaries
ELF Object File Format

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

Segment header table
- Page size, virtual addresses memory segments (sections), segment sizes.

.text section
- Code

.data section
- Initialized global variables

.bss section
- Uninitialized global variables
  “Block Started by Symbol”
  “Better Save Space”
- Has section header but occupies no space

<table>
<thead>
<tr>
<th>ELF header</th>
<th>Segment header table (required for executables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text section</td>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
<td>.symtab section</td>
</tr>
<tr>
<td></td>
<td>.rel.txt section</td>
</tr>
<tr>
<td></td>
<td>.rel.data section</td>
</tr>
<tr>
<td></td>
<td>.debug section</td>
</tr>
</tbody>
</table>

Section header table
## ELF Object File Format (cont)

### ELF header
- Segment header table
  - Required for executables

### Section header table
- Offsets and sizes of each section

### .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)

### Section header table
- Offsets and sizes of each section

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Segment header table</td>
</tr>
<tr>
<td>(required for executables)</td>
</tr>
<tr>
<td>.text section</td>
</tr>
<tr>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
</tr>
<tr>
<td>.symtab section</td>
</tr>
<tr>
<td>.rel.text section</td>
</tr>
<tr>
<td>.rel.data section</td>
</tr>
<tr>
<td>.debug section</td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>
Linker Symbols

Global symbols
- Symbols defined by module \( m \) that can be referenced by other modules.
- Ex: non-static C functions and non-static global variables.

External symbols
- Global symbols that are referenced by module \( m \) but defined by some other module.

Local symbols
- Symbols that are defined and referenced exclusively by module \( m \).
- Ex: C functions and variables defined with the static attribute.

Key Point: Local linker symbols are not local program variables
Resolving Symbols

Def of global symbol `buf`

```
main.c
int buf[2] = {1,2};
int main()
{
    swap();
    return 0;
}
```

Def of local symbol `bufp0`

```
static int *bufp0 = &buf[0];
```

Ref to external symbol `buf`

```
extern int buf[];
```

Def of global symbol `swap`

```
void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

Ref to external symbol `swap`

Ref to external symbol `buf`

Linker knows nothing of `temp`
Relocating Code and Data

Relocatable Object Files

- **System code**
  - main.o
    - main()
    - int buf[2]={1,2}
- **System data**
  - swap.o
    - swap()
    - int *bufp0=&buf[0]
    - int buf[2]={1,2}
    - int *bufp1

Executable Object File

0

- **Headers**
- **System code**
  - main()
  - swap()
- **More system code**
  - int buf[2]={1,2}
  - int *bufp0=&buf[0]
- **Uninitialized data**
  - int *bufp1
- **System data**
- **.text**
- **.data**
- **.bss**
- **.symtab**
- **.debug**
int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}

Disassembly of section .data:

00000000 <buf>:
  0: 01 00 00 00 02 00 00 00

Source: objdump
extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

Disassembly of section .text:

00000000 <swap>:
  0: 55                  push   %ebp
  1: 8b 15
      00 00 00 00 00
      mov    0x0,%edx
  3: R_386_32 bufp0
      mov   0x4,%eax
  7: a1
      00 00 00 00
      mov    0x0,%eax
  8: R_386_32 buf
    mov   %esp,%ebp
  c: 89 e5
    mov    %esp,%ebp
  e: c7 05
      00 00 00 00 00 04
      movl   $0x4,0x0
  15: 00 00 00
  10: R_386_32
    mov   %esp,%ebp
  14: R_386_32 buf
    mov   %ebp,%esp
  18: 89 ec
      mov   (%edx),
    %ecx
  1c: 89 02
      mov   (%edx),
    (%edx)
  1e: a1
      00 00 00 00 00
      mov   0x0,%eax
  1f: R_386_32
    mov   %esp,%ebp
  23: 89 08
    mov   %ecx,
    (%eax)
  25: 5d
    pop   %ebp
  26: c3
      ret
extern int buf[];
static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
Executable After Relocation (text):

080483b4 <main>:
  80483b4:      55                      push   %ebp
  80483b5:      89 e5                   mov    %esp,%ebp
  80483b7:      83 ec 08                sub    $0x8,%esp
  80483ba:      e8 09 00 00 00         call   80483c8 <swap>
  80483bf:      31 c0                   xor    %eax,%eax
  80483c1:      89 ec                   mov    %ebp,%esp
  80483c3:      5d                      pop    %ebp
  80483c4:      c3                      ret

080483c8 <swap>:
  80483c8:      55                      push   %ebp
  80483c9:      8b 15 5c 94 04 08       mov    0x804945c,%edx
  80483cf:      a1 58 94 04 08          mov    0x8049458,%eax
  80483d4:      89 e5                   mov    %esp,%ebp
  80483d6:      c7 05 48 95 04 08 58  movl   $0x8049458,0x8049548
  80483dd:      94 04 08                mov    %edx,%ecx
  80483de:      89 02                   mov    %eax,(%edx)
  80483e2:      8b 0a                   mov    (%edx),%ecx
  80483e4:      89 02                   mov    %eax, (%edx)
  80483e6:      a1 48 95 04 08          mov    0x8049548,%eax
  80483eb:      89 08                   mov    %ecx, (%eax)
  80483ed:      5d                      pop    %ebp
  80483ee:      c3                      ret
Executable After Relocation (.data)

Disassembly of section .data:

08049454 <buf>:
  8049454: 01 00 00 00 02 00 00 00

0804945c <bufp0>:
  804945c: 54 94 04 08
Program symbols are either strong or weak

- **strong**: procedures and initialized globals
- **weak**: uninitialized globals
Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.
(Each item can be defined only once)

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker will pick an arbitrary one.
- Can override this with gcc -fno-common
Packaging Commonly Used Functions

How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

Awkward, given the linker framework so far:

- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient

- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer
Static Libraries

Solution: static libraries (.a archive files)

- Concatenate related relocatable object files into a single file with an index (called an archive).

- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.

- If an archive member file resolves reference, link into executable.
Creating Static Libraries

Archiver allows incremental updates:
  • Recompile function that changes and replace .o file in archive.

```
unix> ar rs libc.a \n     atoi.o printf.o ... random.o
```

C standard library
Commonly Used Libraries

\texttt{libc.a (the C standard library)}
- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

\texttt{libm.a (the C math library)}
- 1 MB archive of 226 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
e_asinl.o
...
```
Linking with Static Libraries

Translators (cpp, cc1, as)

main2.c vector.h

Archiver (ar)

libvector.a

libc.a

addvec.o multvec.o

p2

Fully linked executable object file

Static libraries

printf.o and any other modules called by printf.o

Relocatable object files

Linker (ld)
Using Static Libraries

Linker’s algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters!
- Moral: put libraries at the end of the command line.

bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
Loading Executable Object Files

Executable Object File

<table>
<thead>
<tr>
<th>ELF header</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table (required for executables)</td>
<td></td>
</tr>
<tr>
<td>.text section</td>
<td></td>
</tr>
<tr>
<td>.data section</td>
<td></td>
</tr>
<tr>
<td>.bss section</td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.rel.text</td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>Section header table (required for relocatables)</td>
<td></td>
</tr>
</tbody>
</table>

Kernel virtual memory

User stack (created at runtime)

Memory-mapped region for shared libraries

Run-time heap (created by malloc)

Read/write segment (.data, .bss)

Read-only segment (.init, .text, .rodata)

Unused

Memory invisible to user code

%esp (stack pointer)

brk

Loaded from the executable file

15-213, S’08