Lecture 7:
Procedure Call and Stack
Address Space (User)

Objects in the address space

instructions

local variables

function call return address

dynamic memory

malloc()

global and static variables

stack

heap

data

text

0x0000 0000
exe C

depend
 DATA
 init
 symbols

elf format

stack

heap

BSS

data

text

and start running it!
Process address space in xv6

Virtual

- 4 Gig
- Device memory
- Unused if less than 2 Gig of physical memory
- Free memory
- Kernel data
- Kernel text
- Program data & heap
- User stack
- User data
- User text

Physical

- Memory-mapped 32-bit I/O devices
- Unused if less than 2 Gig of physical memory
- Extended memory
- I/O space
- Base memory

0x100000
640K

PHYSTOP
4 Gig

0
0x1000000
0xFE000000

Youjip Won
$2^n - 1$ → 

\[ \%\text{esp} \rightarrow 0x0B63 \]

\%esp: points to the stack top

stack
push abc

$2^n - 1 \rightarrow \quad 2^n - 1 \rightarrow$

%esp $\rightarrow$ OxB63 $\rightarrow$ %esp $\rightarrow$ Oxa04

stack

- get value from abc
- %esp = %esp - 4
- stores the value at the address pointed by %esp.
%esp: the address of contents to pop

- pop abc

2^n - 1 ➔

%esp ➔ 0x0B63

- save the value @ %esp to variable 'abc'.
- %esp <= %esp - 4

Stack

2^n - 1 ➔

%esp ➔ 0x0B63

0xAB64

Stack
Procedure Call Overview

...<set up args>
call

Caller

Callee
Procedure Call Overview

Caller

... <set up args> call
<cleanup up args> <find return val>
...

Callee

<create local vars>
...
<set up return val>
<destroy local vars>
return
Call Convention

- what to pass
  - caller $\rightarrow$ callee: parameters
  - callee $\rightarrow$ caller: return value

- address
  - caller: where to jump
  - callee: where to return

- saving the registers
  - caller and callee may execute on the same CPU
  - we need to save to registers used by the caller
  - method 1: caller saves
  - method 2: callee saves

stack frame
stack frame

- contents
  - parameters
  - local variables of the callee
  - register values of the caller
  - return address

- allocated when the function is called
- deallocated when the function returns.
Stack Frame

procA()
{
    procB()
    
    procC()
}

main()
{
    procA()
}

%esp ->

stack

stack frame for procA

stack frame for procB

stack frame for procC
Organization of stack frame

- **ebp (base pointer register)**
  - the address of the beginning of the stack frame, remain unchanged while the function executes.

- **esp (stack pointer)**
  - address of stack top, keep changing while the function executes.

Stack grows.
Procedure call

- Push return address on stack
- Jump to label
- return address: address of instruction after call
- in the figure below, what is the return address? 0x8038553

```
0x804854e:   e8 3d 06 00 00    call    0x8048b90 <main>
0x8038553:  50                pushl   %eax
```
**Details!**

- **Procedure return**: `ret`
  - Pop the return address from the stack.
  - Jump to address.
  - Return value: by convention, save it to `%eax` register (it is not mandatory)
  - When the return value requires more than 4 byte, save the address of the return value at `%eax`.
  - Caller saves the `%eax` before calling.

```
0x804854e:  e8 3d 06 00 00    call    0x8048b90 <main>
0x8038553:  50                pushl   %eax
```
### Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 0x8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>0x8048553</td>
<td>50</td>
<td>push %eax</td>
</tr>
</tbody>
</table>

#### Diagram

- **%esp**: program counter
- **%eip**: Program counter

(source: [https://courses.cs.washington.edu/courses/cse351/12aw/l07-procedures.pdf](https://courses.cs.washington.edu/courses/cse351/12aw/l07-procedures.pdf))
Procedure Call Example

0x804854e : e8 3d 06 00 00  call  0x8048b90  <main>
0x8048553 : 50  push  %eax

%eip: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

```
0x804854e : e8 3d 06 00 00 call 0x8048b90 <main>
0x8048553 : 50 push %eax
```

%eip: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

0x804854e : e8 3d 06 00 00 call 0x8048b90 <main>
0x8048553 : 50 push %eax

%esp: program counter

(%source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

```
0x804854e: e8 3d 06 00 00 call 0x8048b90 <main>
0x8048553: 50 push %eax
```

---

(call 8048b90)

---

%esp: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

0x8048591 : c3 ret

%eip: program counter

%esp
0x104

%eip
0x8048b91

0x100
0x104
0x108
0x10c
0x110

0x8048553

0x8048b91

ret

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

0x8048591 : c3  ret

%eip: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

0x8048591 : c3  ret

%eip: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Procedure Call Example

0x8048591 : c3  ret

%eip: program counter

(source: https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)
Linux stack frame
IA32/Linux Stack Frame

- **Stack Frame ("Top" to Bottom)**
  - "Arguments": parameters for function about to be called
  - Local variables
  - Saved register context (when reusing registers)
  - Old %ebp

- **Caller’s role in setting up the Stack Frame**
  - Return address
    - Pushed by call instruction
    - Arguments for this call

(source: [https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf](https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf))
int
main() {
    char *argv[3];

    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;

    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
data segment: constants, strings ("echo", "hello")

<table>
<thead>
<tr>
<th>e</th>
<th>c</th>
<th>h</th>
<th>o</th>
<th>\0</th>
<th>h</th>
<th>e</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x65</td>
<td>0x63</td>
<td>0x68</td>
<td>0x6f</td>
<td>0x00</td>
<td>0x68</td>
<td>0x65</td>
<td>0x6c</td>
</tr>
<tr>
<td>l</td>
<td>o</td>
<td>!</td>
<td>\0</td>
<td></td>
<td>0x68</td>
<td>0x65</td>
<td>0x6c</td>
</tr>
<tr>
<td>0x6c</td>
<td>0x6f</td>
<td>0x21</td>
<td>0x00</td>
<td>0x68</td>
<td>0x65</td>
<td>0x6c</td>
<td>0x6c</td>
</tr>
</tbody>
</table>

0x7e5: 0x6f 0x20 0x77 0x6f 0x72 0x6c 0x64 0x21

0x7ed: 0x0a 0x00 0x28 0x6e 0x75 0x6c

<table>
<thead>
<tr>
<th>code segment: exec() starts at 0x000002e0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000002e0 &lt;+0&gt;:  mov  $0x7,%ax</td>
</tr>
<tr>
<td>0x000002e3 &lt;+3&gt;:  add  %al,(%bx,%si)</td>
</tr>
<tr>
<td>0x000002e5 &lt;+5&gt;:  int  $0x40</td>
</tr>
<tr>
<td>0x000002e7 &lt;+7&gt;:  ret</td>
</tr>
</tbody>
</table>
Revisiting *sysprog*

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo",argv);
    exit();
}
```

**Calling** `exec` **from** `main`

```
main:
    ...
    lea -0x14(%ebp),%eax  # 1
    push %eax             # 2
    push $0x7d5           # 3
    call 2e0 <exec>       # 4
    ...
```

**Start address of exec**
Revisiting sysprog

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;

    printf("hello world!\n");
    exec("echo", argv);
    exit();
}
```

Calling `exec` from `main`

```assembly
main:
    ...
    lea -0x14(%ebp),%eax    # 1
    push %eax                # 2
    push $0x7d5              # 3
    call 2e0 <exec>         # 4
    ...

lea address regr
save the virtual address of "address" to regr.

save the start address of `argv` array to `%eax`.```
Revisiting sysprog

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;

    printf(1,"hello world!\n");
    exec("echo",argv);
    exit();
}
```

Calling `exec` from `main`

```assembly
ingore
main:
    ...
    lea -0x14(%ebp),%eax  # 1
    push %eax             # 2
    push $0x7d5           # 3
    call 2e0 <exec>       # 4
    ...
```

push the second argument (argv) to stack frame.
Revisiting sysprog

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo",argv);
    exit();
}
```

**Calling exec from main**

```
main:
  ...
  lea -0x14(%ebp),%eax     # 1
  push %eax                # 2
  push $0x7d5              # 3
  call 2e0 <exec>          # 4
  ...
```

1. push the first argument to stack
2. the first argument is string —> push the start address of “echo”, 0x7d5, to stack.
Revisiting sysprog

```c
int main() {
    char *argv[3];

    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;

    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
```

### Calling `exec` from `main`

```
main:
    ...
    lea -0x14(%ebp),%eax  # 1
    push %eax            # 2
    push $0x7d5           # 3
    call 2e0 <exec>      # 4
    ...
```

1. Jump to 0x2e0 which is the address of `exec`
2. Just before jump, push the address of the next instruction (return address)
Calling `exec` from `main`

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
```

Stack frame contents just before calling `exec`

```
main:
0x38:  ...      
0x3d:  lea  -0x14(%ebp),%eax
0x3e:  push  %eax
0x3f:  push  $0x7d5
0x44:  call  2e0 <exec>
0x49:  ...      
```

Stack size; 16 byte aligned

```
0x2fd8 0x2fe4  %ebp
0x2fd4  0       
0x2fda  0       
0x2fdd  0x7da   argv[0] = “echo”
0x2fe8  0x7d5   argv[1] = “hello”
0x2fc0  0       
0x2fc4  0x7da   argv[2] = ‘\0’
0x2fc8  0       
0x2fc8  0x2fd8  0x2fda  0x2fe4  %ebp
0x2fcc  0       
0x2fd0  0       
0x2fd4  0       
0x2fb8  0       
```

Padding

%esp

%ebp

%eax

%esp

0x2fd8

0x2fb8

0x2fd8
Calling `exec` from `main`

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
```

**prepare the first parameter for `exec()`**
(save the starting address of `argv` to `eax` register)

```
main:
0x38: ...  
0x3d: lea -0x14(%ebp),%eax
0x3e: push %eax
0x3f: push $0x7d5
0x44: call 2e0 <exec>
0x49: ...  
```

%ebp

%esp

Padding

%eax

%esp

%ebp

argv[0] = “echo”

argv[1] = “hello”

argv[2] = ‘\0’

Calling `exec` from `main`

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
```

```
main:
0x38:   ...  
0x3d:   lea  -0x14(%ebp),%eax  
0x3e:   push  %eax  
0x3f:   push  $0x7d5  
0x44:   call  2e0 <exec>  
0x49:   ...  
```

```
push the first parameter to the stack
4 byte
```

```
0x2fd8  0x2fe4  %ebp
0x2fd4  0  
0x2fd0  0  
0x2fcc  0  
0x2fc8  0x7da  argv[2] = ‘\0’
0x2fc4  0x7d5  argv[1] = “hello”
0x2fc0  0  
0x2fbc  0  
0x2fb8  0  
0x2fb4  0x2fc4  argv[0] = “echo”
```

```
%eax  0x2fc4  %esp  0x2fb4  %ebp  0x2fd8
```

```
Youjip Won
KAIST OSLab Operating Systems Laboratory
```
Calling `exec` from `main`

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf(1,"hello world!\n");
    exec("echo", argv);
    exit();
}
```

### Calling `exec` from `main`

- `main: 0x38: ...`
- `0x3d: lea -0x14(%ebp),%eax`
- `0x3e: push %eax`
- `0x3f: push $0x7d5`
- `0x44: call 2e0 <exec>`
- `0x49: ...`
calling exec

Calling exec from main

```c
int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello!";
    argv[2] = 0;
    printf("hello world!\n");
    exec("echo", argv);
    exit();
}
```

call exec which is at 0x2e0 (push the return address)

```
main: 0x38: ...
0x3d: lea -0x14(%ebp),%eax
0x3e: push %eax
0x3f: push $0x7d5
0x44: call 2e0 <exec>
0x49: ...
```
Hello example

```c
int hello(int a, int b, int c)
{
    printf("Hello %d %d %d\n", a, b, c);
    return 0;
}

int main(int argc, char *argv[])
{
    int a = 1, b = 2, c = 3;
    hello(1, 2, 3);
    c = a + b;
    return 0;
}
```

Calling hello() from main

```assembly
0x00000051 <+41>:    push   $0x3
0x00000053 <+43>:    push   $0x2
0x00000055 <+45>:    push   $0x1
0x00000057 <+47>:    call   0x0 <hello>
0x0000005c <+52>:    add    $0x10,%esp
```
Hello example

```c
int hello(int a, int b, int c)
{
    printf("Hello %d %d %d\n", a, b, c);
    return 0;
}

int main(int argc, char *argv[])
{
    int a = 1, b = 2, c = 3;
    hello(1, 2, 3);
    c = a + b;
    return 0;
}
```

(gdb) x/32x $esp
0x2fa0: 0x00000000 0x00000000 0x00002fd8 0x0000005c
0x2fb0: 0x00000001 0x00000002 0x00000003 0x00000000
0x2fc0: 0x00000000 0x00000003 0x00000002 0x00000001
0x2fd0: 0x00000000 0x000002fe8 0x00003fa8 0xffffffff
0x2fe0: 0x00000000 0xffffffff 0x00000001 0x00002ff0
0x2ff0: 0x00002ff8 0x00000000 0x6c6c6568 0x0000006f
0x3000: Cannot access memory at address 0x3000
Practice - Examine the stack of the sysprog example

1. Write below code and save as sysprog.c

```c
#include "types.h"
#include "user.h"

int main() {
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hi";
    argv[2] = 0;
    printf(1, "Call exec\n");
    exec("echo", argv);
    exit();
}
```

2. Edit Makefile to compile and run it in xv6

```bash
UPROGS=
    _cat
    _echo
    _forktest
    _grep
    _init
    _kill
    _ln
    _ls
    _mkdir
    _rm
    _sh
    _stressfs
    _usertests
    _wc
    _zombie
    _tests
    _sysprog
```

3. Run it with gdb

```bash
jata@jata-desktop:~/unix_kernel_design/xv6-public$ make qemu-nox-gdb
```

```
*** Now run 'gdb'.
```

```
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000
```

jata@jata-desktop:~/unix_kernel_design/xv6-public$ gdb -q
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration of GDB. Attempting to continue with the default i8606 settings.
The target architecture is assumed to be i8606
```
[f000:0xff0] 0x0000: ljmp $0x0000,0x0e05b
```

```
0x00000000 in ?? ()
```

+ symbol-file kernel
```
gdb> ```
4. Add debug symbol of sysprog and continue

   (gdb) add-symbol-file _sysprog 0
   add symbol table from file "_sysprog" at
   .text_addr = 0x0
   (y or n) y
   Reading symbols from _sysprog...done.
   (gdb) 

5. Set breakpoint at main function

   (gdb) break main
   Breakpoint 1 at 0x0: main. (2 locations)
   (gdb) 

   There is two locations because there is two symbol of main function;
   kernel’s main & sysprog’s main
6. Continue xv6

```c
Breakpoint 1, main () at main.c:19
19     {
(gdb) c
Continuing.
[New Thread 2]
[Switching to Thread 2]
The target architecture is assumed to be i8086
[ 1b:  0] 0x1b0 <memmove+32>: add %al,(%bx,%si)
```

```c
Breakpoint 1, main () at sysprog.c:5
5     main() {
(gdb) c
Continuing.
[ 1b:  0] 0x1b0 <memmove+32>: add $0x1,%dx
```

```c
Breakpoint 1, main () at sysprog.c:5
5     main() {
(gdb) c
Continuing.
```

The execution will be stopped 3 times; kernel’s main, initcode.S, init.c
Practice - Examine the stack of the `sysprog` example

7. Run `sysprog` in `xv6`

$ ./sysprog

Breakpoint 1, main () at sysprog.c:5
5     main() {
(gdb) c
Continuing.
[ 1b: 0] 0x1b0 <memmove+32>: ret $0x3901

Breakpoint 1, main () at sysprog.c:5
5     main() {
(gdb)

There is one more breakpoint as execute `sysprog` because shell encounter 0x0 address code executing `sysprog`

address of main in `sysprog`
Practice - Examine the stack of the `sysprog` example

8. dump current stack

```
(gdb) x/24x $esp
0x2fe0: 0xffffffff 0x00000001 0x00002fec 0x00002ff4
0x2ff0: 0x00000000 0x79752f2e 0x61727073 0x00000067
0x3000: Cannot access memory at address 0x3000
(gdb)
```

- fake return addr
- argc
- argv
- argv[0]
Practice - Examine the stack of the `sysprog` example

9. Set breakpoint at 0x44 (call `exec`) and continue

(gdb) br *0x44
Breakpoint 3 at 0x44: file sysprog.c, line 11.
(gdb) continue
Continuing.
[ 1b: 6] 0x1b6 <memmove+38>: pop %bx
Breakpoint 3, 0x00000044 in main () at sysprog.c:11
11 exec("echo", argv);
(gdb)
10. Dump stack; What they are?

(gdb) x/24x $esp
0x2fb0: 0x00000736 0x00002fc4 0x00000000 0x00000000
0x2fc0: 0x00000000 0x00000736 0x0000073b 0x00000000
0x2fd0: 0x00000000 0x00000000 0x000003fc8 0x00000000
0x2fe0: 0xffffffff 0x00000001 0x00002fec 0x00002ff4
0x2ff0: 0x00000000 0x79732f2e 0x6f727073 0x00000067
0x3000: Cannot access memory at address 0x3000

(argv[0] = "echo"
argv[1] = "hello"
argv[2] = '\0')