Open RazzmatazZ Laboratory (OrzLab)
http://orzlab.blogspot.com/

深入淺出 Hello World – Part II

探索記憶體模型與系統呼叫

Mar 25, 2007

Jim Huang(黃敬群 /**jserv**)
Email: <**jserv.tw@gmail.com**>
Blog: **http://blog.linux.org.tw/jserv/**

注意

- 本議程針對 x86 硬體架構，至於 ARM 與 MIPS 架構，請另行聯絡以作安排
- 簡報採用創意公用授權條款 (Creative Commons License: **Attribution-ShareAlike**) 發行
- 議程所用之軟體，依據個別授權方式發行
- 系統平台
  - Ubuntu feisty (development branch)
  - Linux kernel 2.6.20
  - gcc 4.1.2
  - glibc 2.5

大綱

- 探索 Linux 記憶體模型
- 深入 syscall
- 再探動態連結

探索 Linux 記憶體模型

- Memory Section/Region
- Linux Memory Allocation
- i386 stack/register/call

Memory section (1)

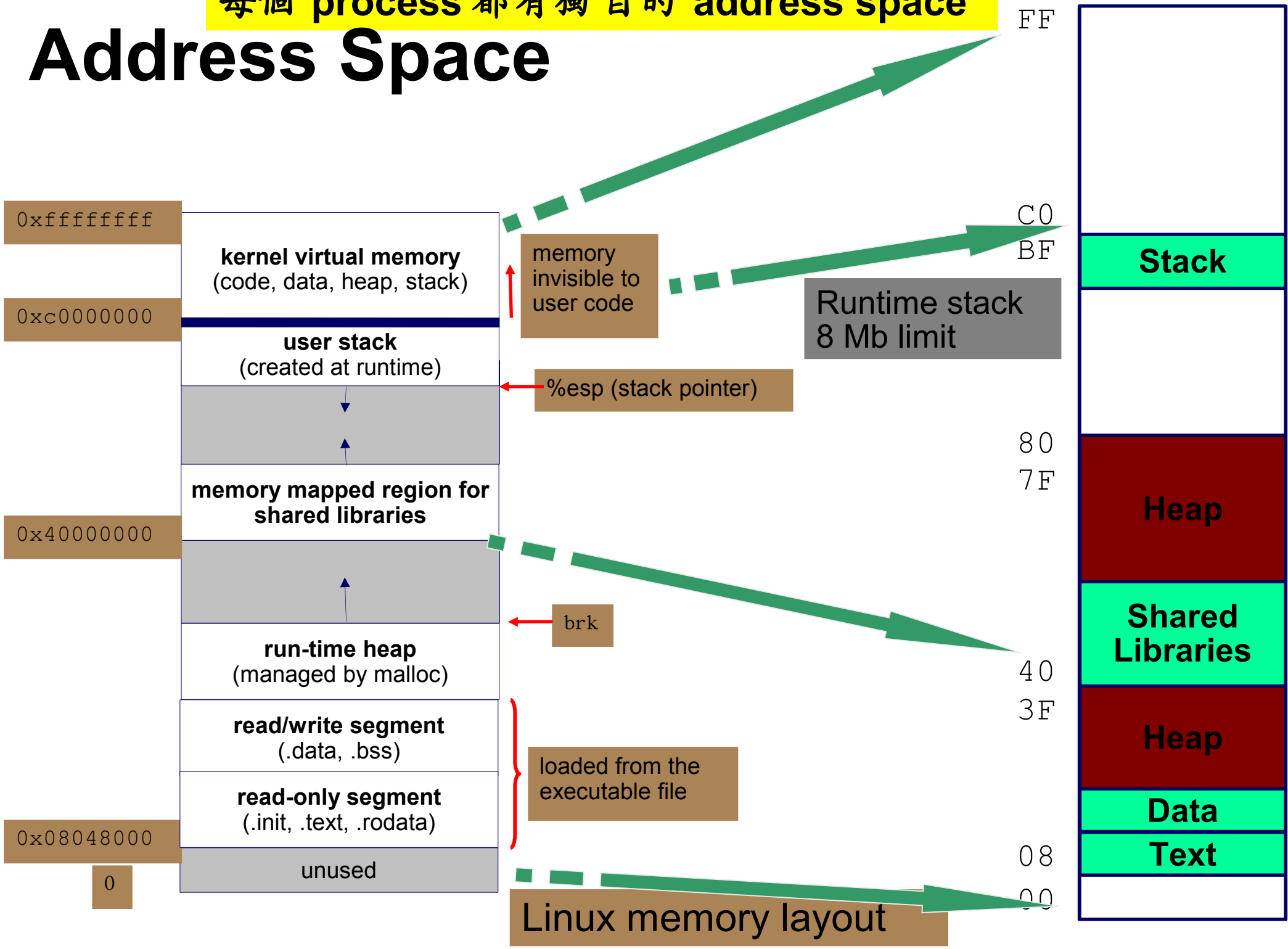
- 概念： Modular programming
- Module 就是特定 I/O 介面的黑盒子，由 re-locatable object code 組成
- Assembler 與 Linker 層面來說，程式由若干 **sections** (absolute or relocatable blocks of code or data) 集合而成
- 來自不同 module 、卻有相同名稱的 sections ，特稱爲”**partial sections**”
- Linker 的職責就是將所有 partial sections 結合爲 sections

Memory section (2)

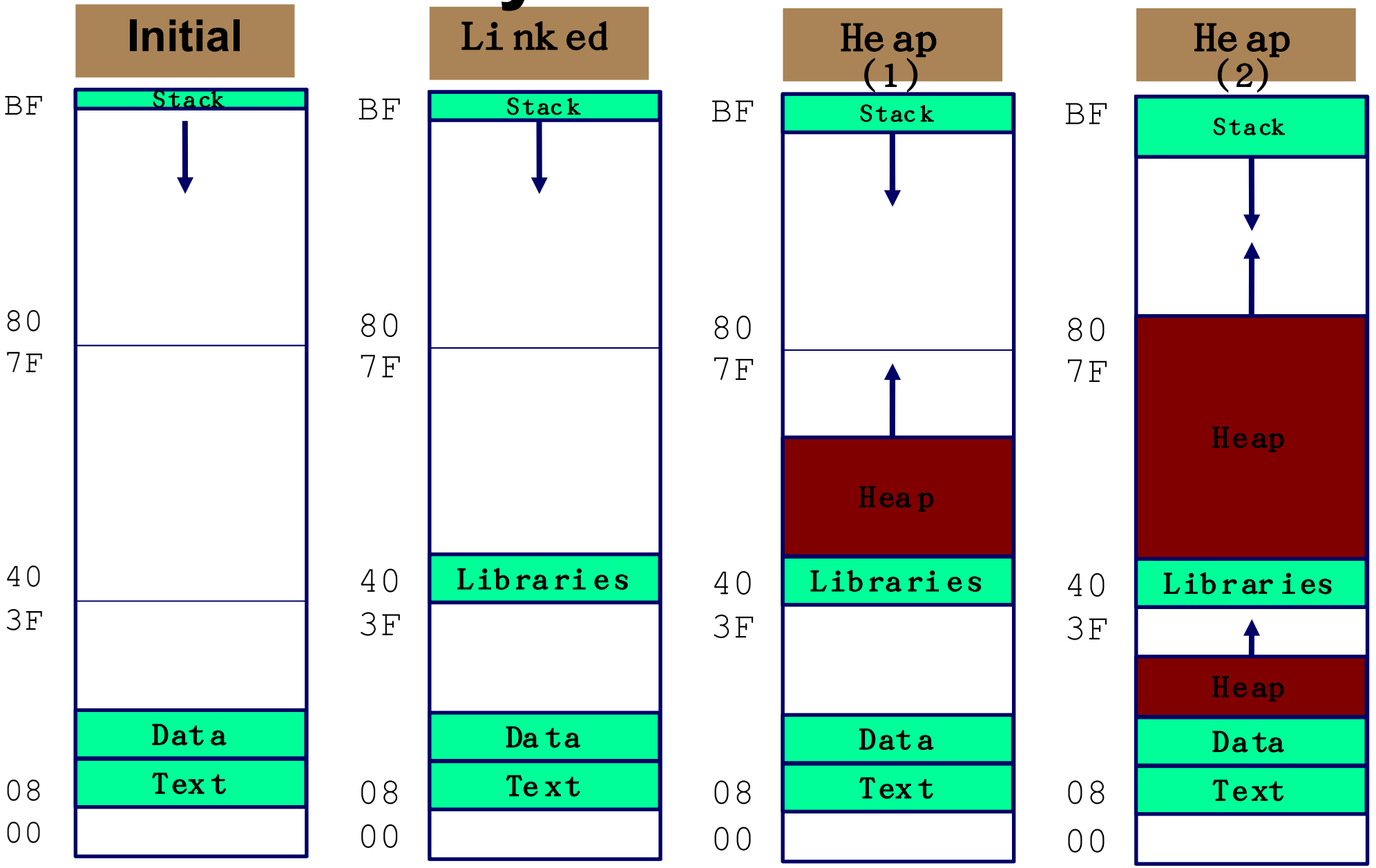
- Absolute sections
  - 需要定位於特定的 memory address
  - 不可結合到其他 section
- Module
  - 由一個或多個 code sections 所組成
- Program
  - 包含單一 absolute module ，並整合其他 absolute/relocatable section
- Linker
  - all external references to symbols in other modules are **resolved** by the linker/locator – the end product is a single absolute object module *(the "program")*

每個 process 都有獨自的 address space

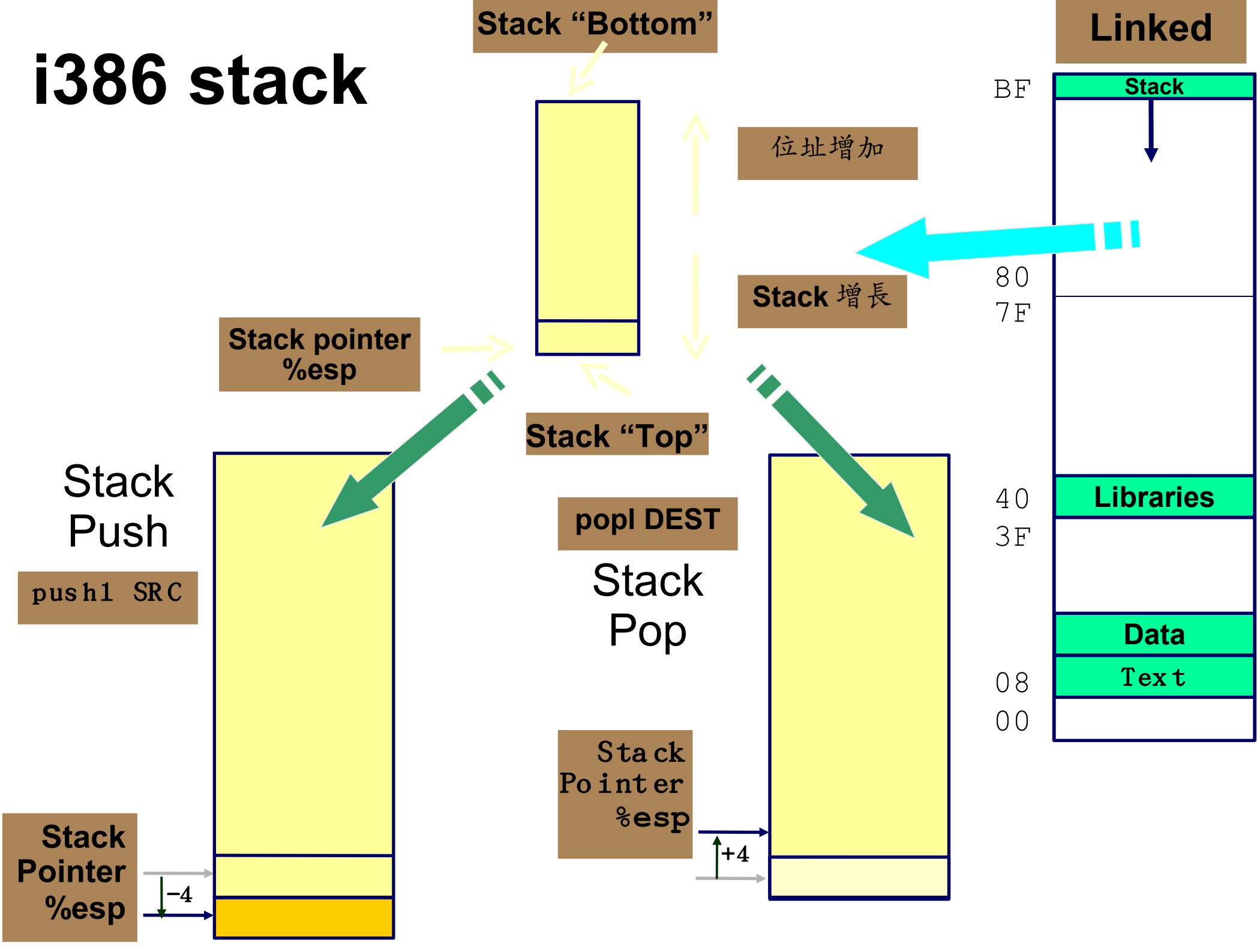
Address Space



Linux Memory Allocation



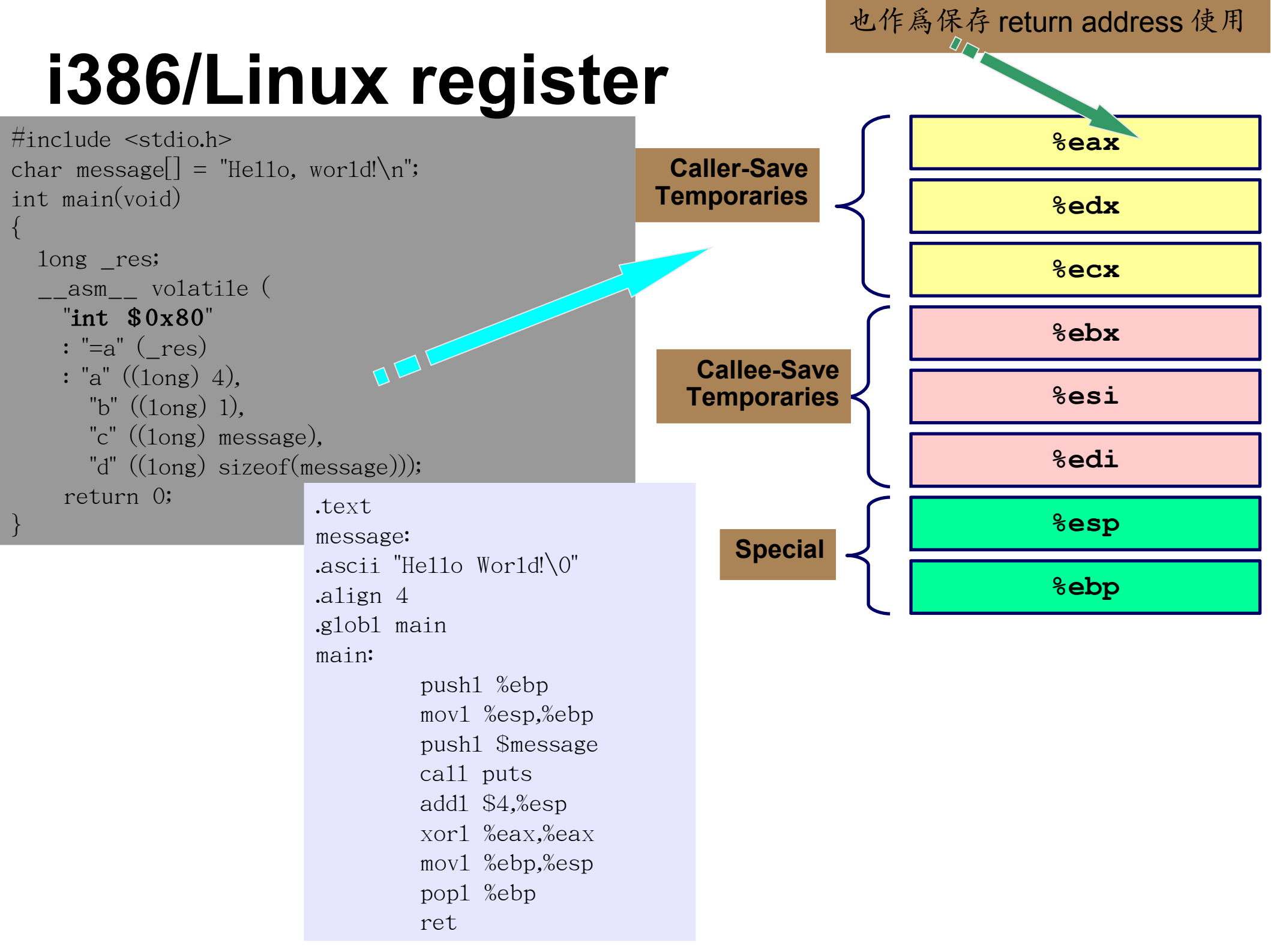
i386 stack


Stack "Bottom"
位址增加
Stack 增長
Stack pointer
%esp
Stack "Top"
Stack
Push
pushl SRC
popl DEST
Stack
Pop
Stack
Pointer
%esp
-4
+4
Linked
BF
Stack
80
7F
40
Libraries
3F
Data
08
Text
00

i386/Linux register

```
#include <stdio.h>
char message[] = "Hello, world!\n";
int main(void)
{
  long _res;
  __asm__ volatile (
    "int $0x80"
    : "=a" (_res)
    : "a" ((long) 4),
      "b" ((long) 1),
      "c" ((long) message),
      "d" ((long) sizeof(message)));
    return 0;
}
```

```
.text
message:
.ascii "Hello World!\0"
.align 4
.globl main
main:
        pushl %ebp
        movl %esp,%ebp
        pushl $message
        call puts
        addl $4,%esp
        xorl %eax,%eax
        movl %ebp,%esp
        popl %ebp
        ret
```



i386 call(1)

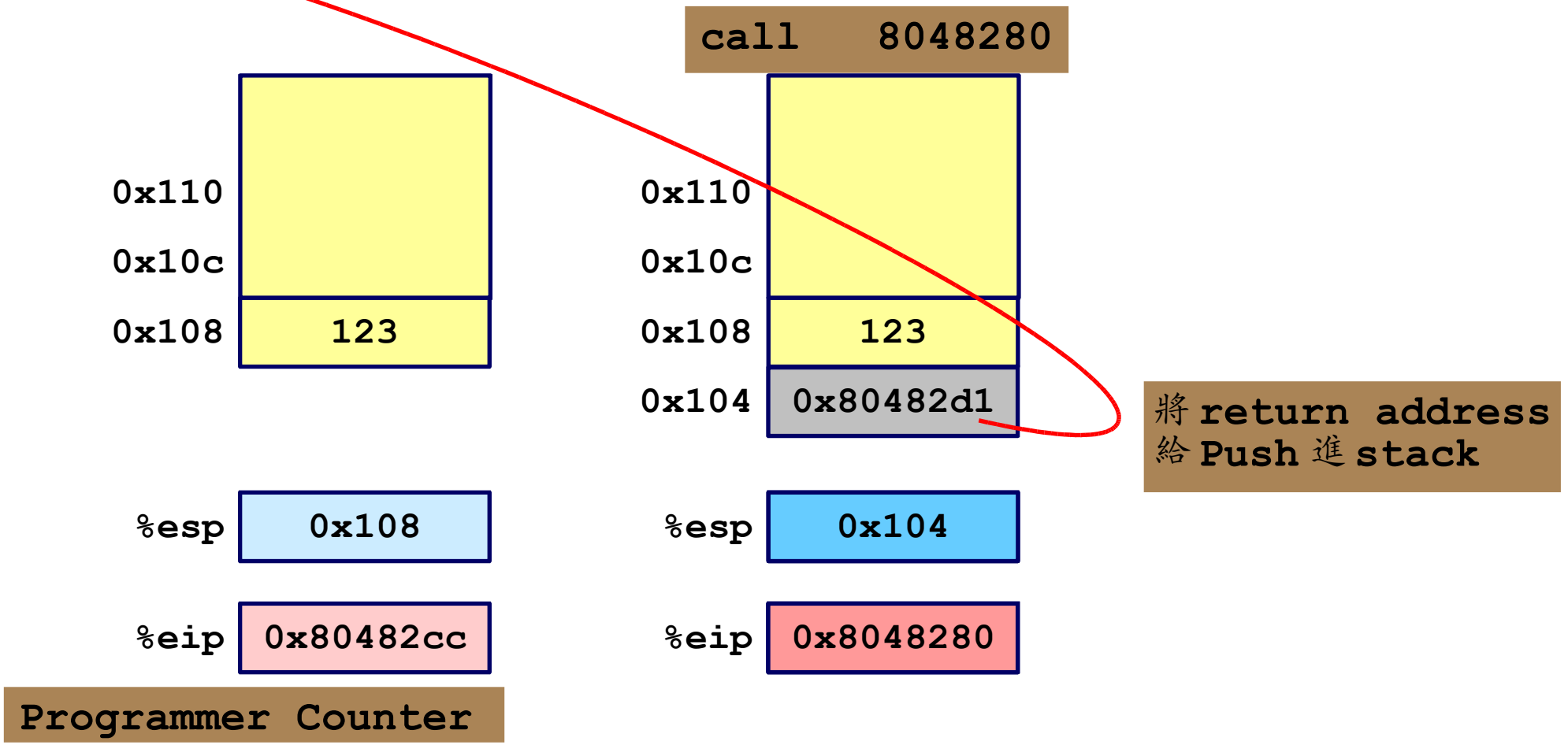
```
Disassembly of section .text:

080482b0 <_start>:
 80482b0:      31 ed                   xor    %ebp,%ebp
 80482b2:      5e                      pop    %esi
 80482b3:      89 e1                   mov    %esp,%ecx
 80482b5:      83 e4 f0                and    $0xfffffff0,%esp
 80482b8:      50                      push   %eax
 80482b9:      54                      push   %esp
 80482ba:      52                      push   %edx
 80482bb:      68 84 83 04 08          push   $0x8048384
 80482c0:      68 89 83 04 08          push   $0x8048389
 80482c5:      51                      push   %ecx
 80482c6:      56                      push   %esi
 80482c7:      68 50 83 04 08          push   $0x8048350
 80482cc:      e8 af ff ff ff          call   8048280 <__libc_start_main@plt>
 80482d1:      f4                      hlt
 80482d2:      90                      nop
 80482d3:      90                      nop

08048280 <__libc_start_main@plt>:
 8048280:      ff 25 44 95 04 08 jmp    *0x8049544
 8048286:      68 00 00 00 00    push   $0x0
 804828b:      e9 e0 ff ff ff    jmp    8048270 <_init+0x18>
```

_start 爲該 Image 的 **entry point**

- call LABLE
  - 使用 stack 來實現 procedure call
  - 先 PUSH 返回位址，然後 JUMP 到 LABLE

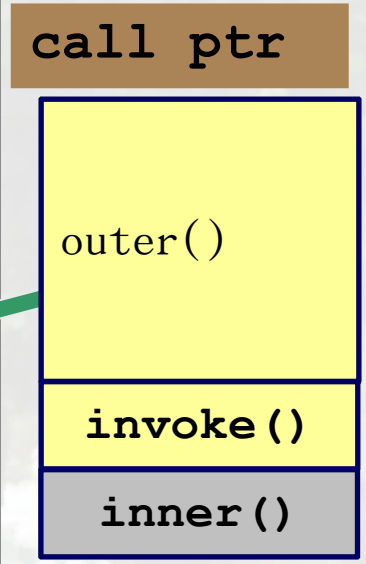
i386 call(2)

```
Disassembly of section .text:

080482b0 <_start>:
...
80482cc:      e8 af ff ff ff      call   8048280 <__libc_start_main@plt>
80482d1:      f4                  hlt
```



GCC Inner Function

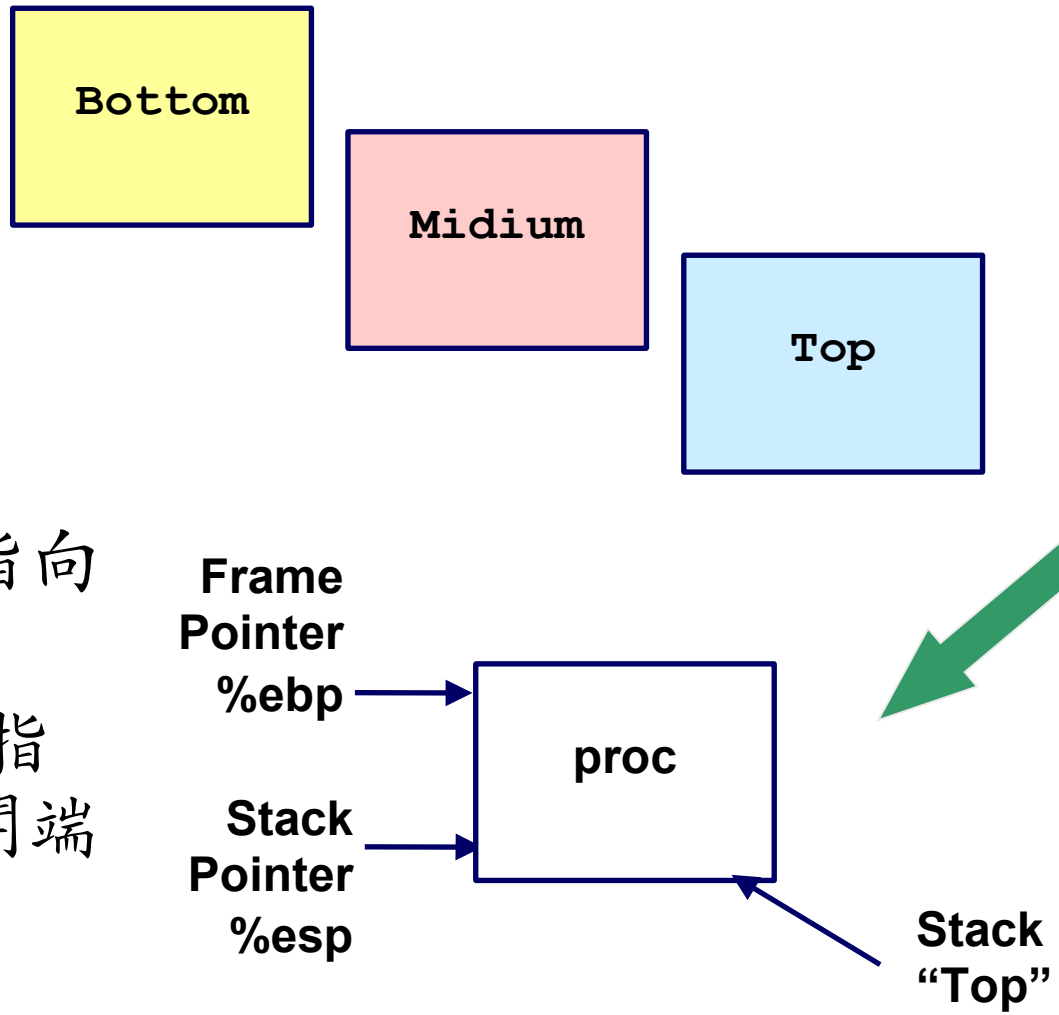
```
$ cat hello.c
#include <stdio.h>

void invoke(void (*func)()) {
    func();
}
void outer() {
    char *hello = "Hello World!\n";
    void inner() {
        printf("%s", hello);
    }
    invoke(inner);
}

int main() {
    outer();
}
$ ./hello
Hello World!
```



將 return address
給 Push 進 stack

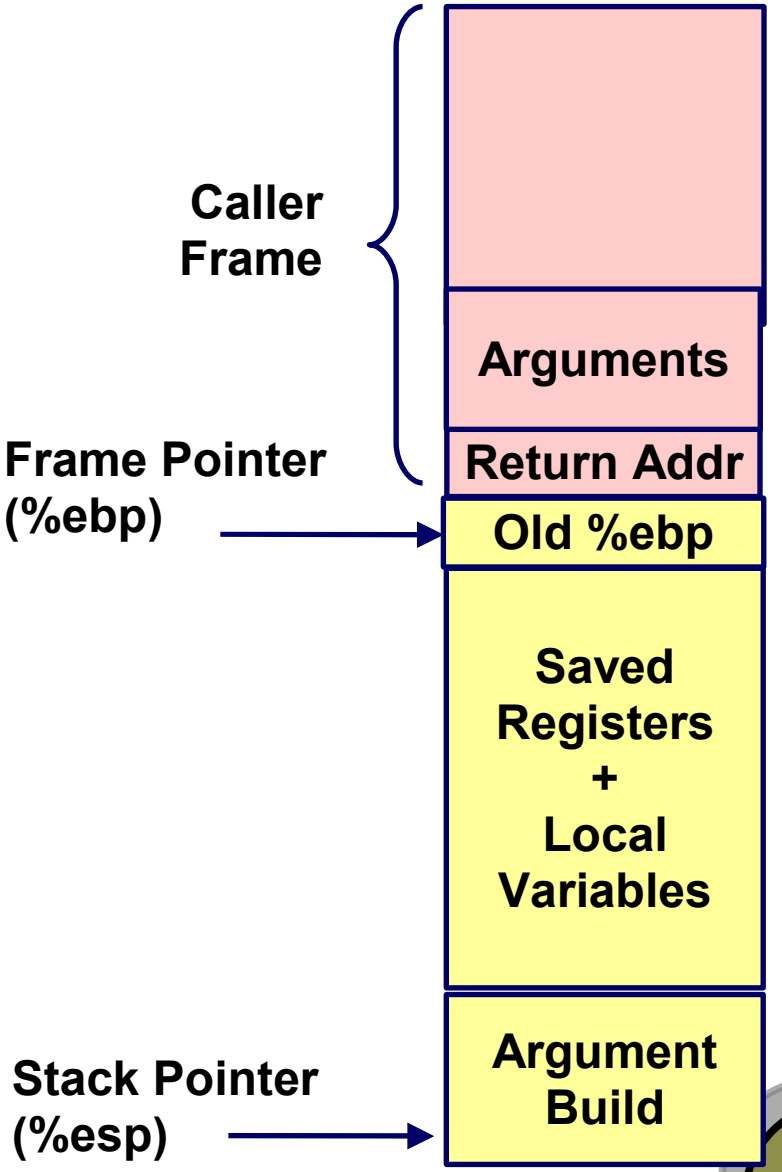
Stack Frame

- 要素
  - Local variables
  - Return information
  - Temporary space
- Pointers
  - Stack Pointer %esp 指向 Stack 的頂端
  - Frame pointer %ebp 指向 Current Frame 的開端



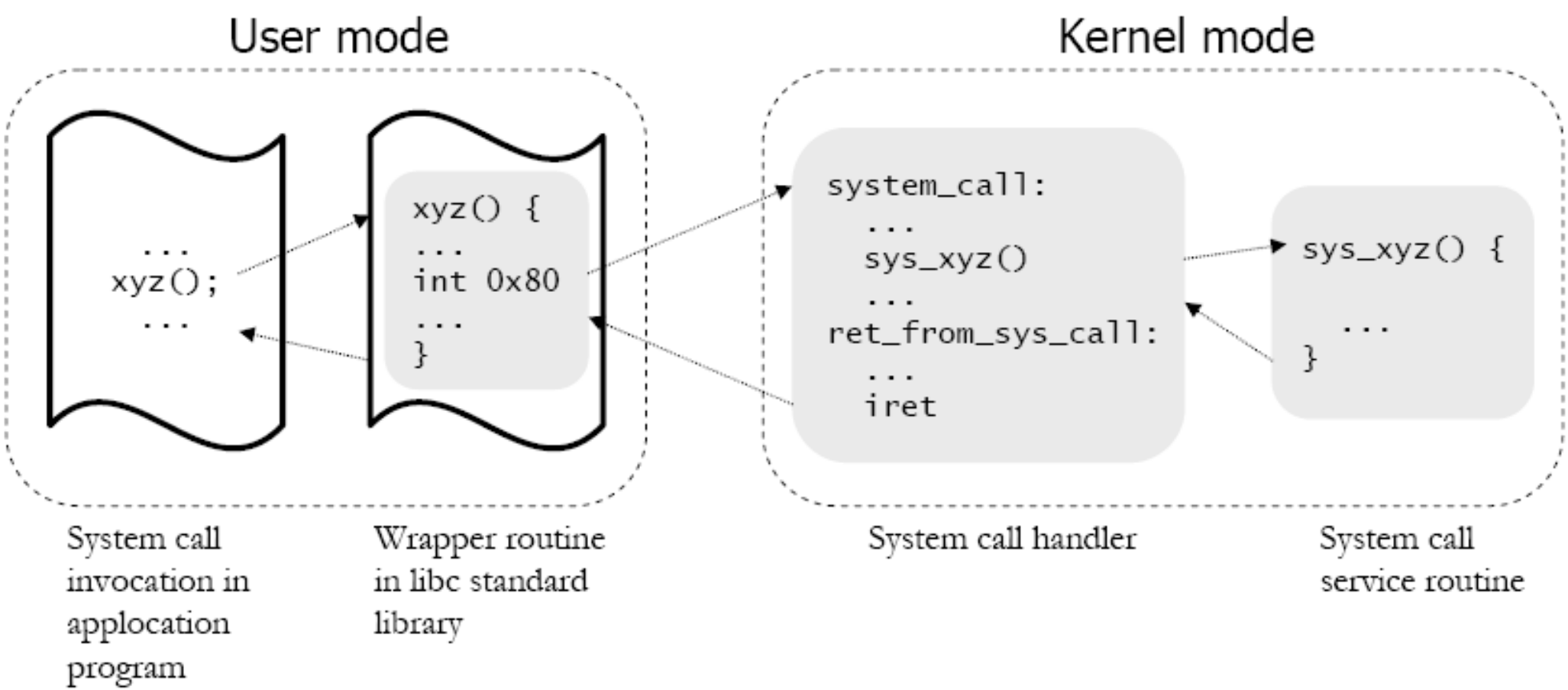
Stack Frame - IA32/Linux

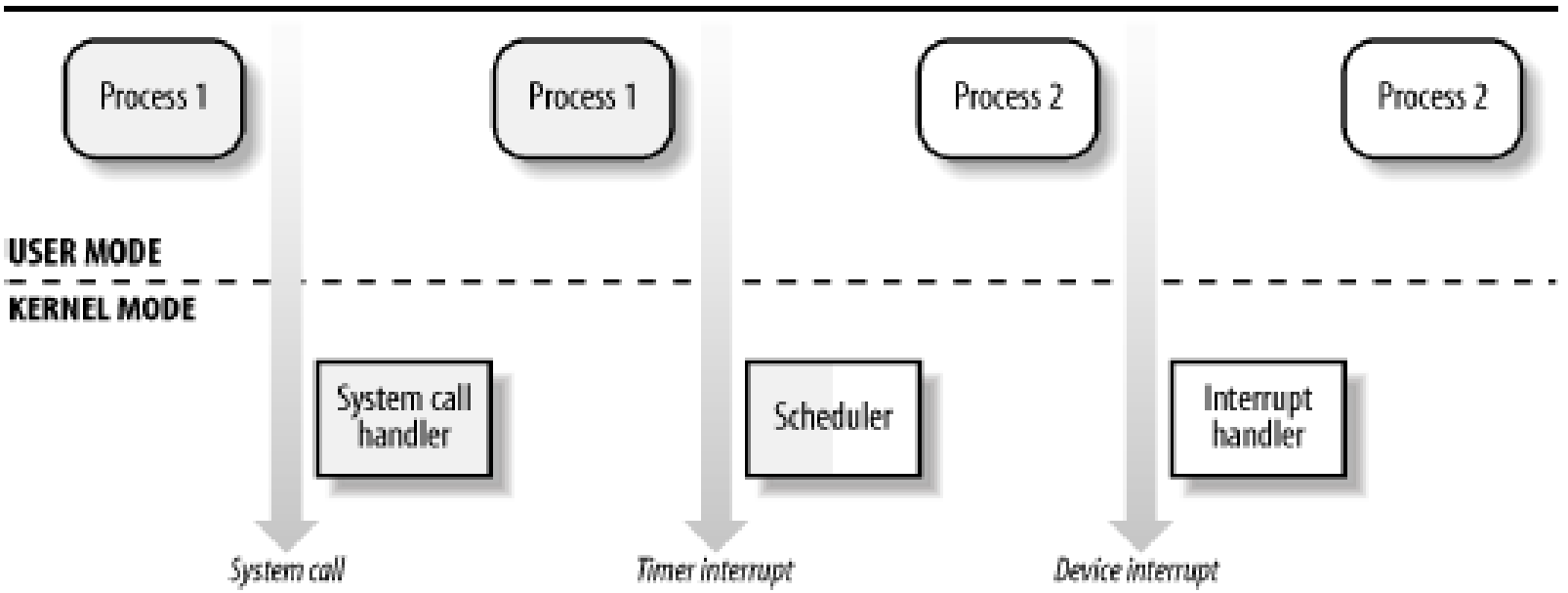
- Current Stack Frame
  - Arugment Build
    - Parameters for function about to call
  - Local variables
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by **call** instruction
  - Arguments for this call



深入 syscall

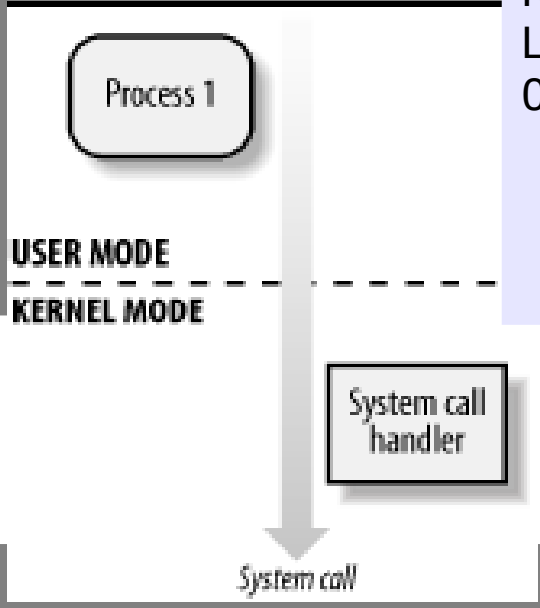
- syscall = system call
- 機制
  - libc wrapper
  - direct syscall
  - syscall function
- Linux kernel 2.6 的 virtual syscall


User mode
Kernel mode
...
xyz();
...
xyz() {
...
int 0x80
...
}
system_call:
...
sys_xyz()
...
ret_from_sys_call:
...
iret
sys_xyz() {
...
}
System call invocation in applocation program
Wrapper routine in libc standard library
System call handler
System call service routine


Process 1
Process 1
Process 2
Process 2
USER MODE
KERNEL MODE
System call handler
Scheduler
Interrupt handler
System call
Timer interrupt
Device interrupt

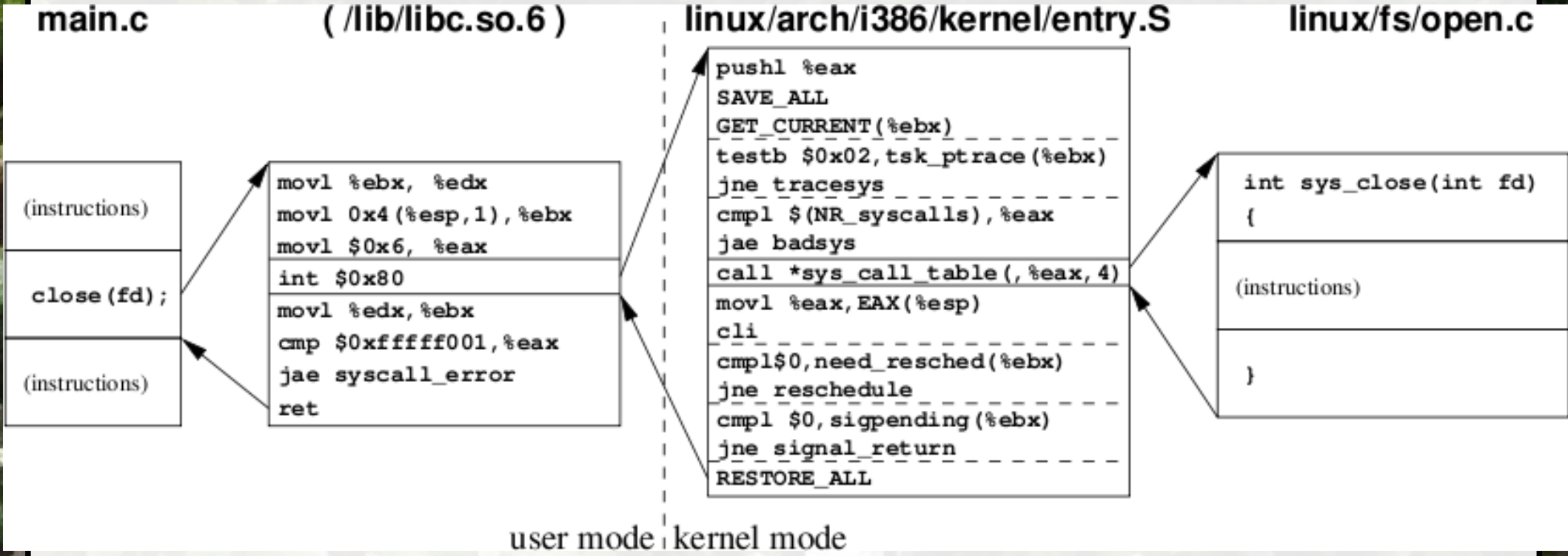
```
$ cat hello-loop.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv)
{
        printf("Hello World!\n");
        while (1) {
                usleep(10000);
        }
        return 0;
}
$ ./hello-loop
Hello World!
```

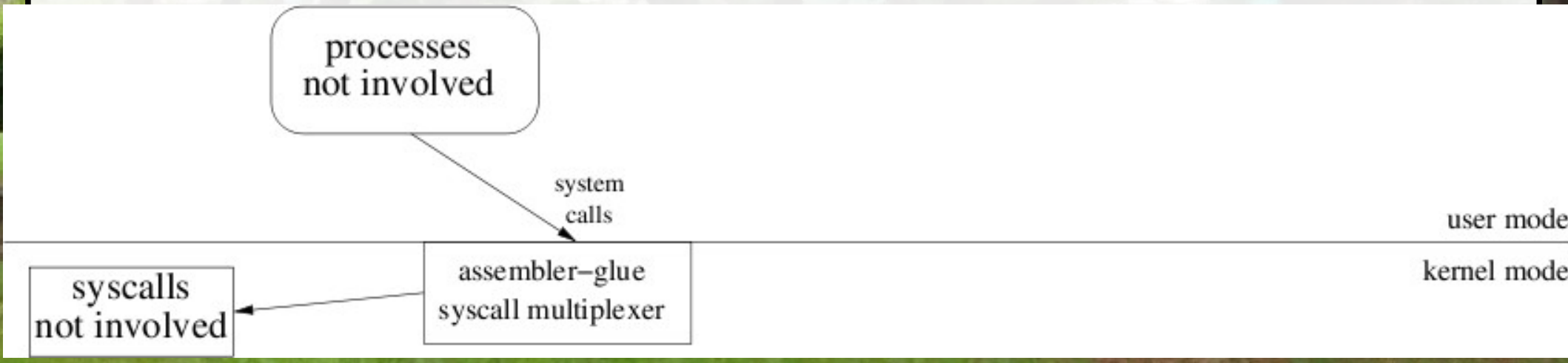
```
$ pidof hello-loop
6987
$ gdb
(gdb) attach 6987
Attaching to process 6987
Reading symbols from
/home/jserv/HelloWorld/samples/hello-loop...done.
Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1".
Reading symbols from
/lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
```




```
(gdb) bt
#0  0xffffe410 in __kernel_vsyscall ()
#1  0xb7e37ef0 in nanosleep () from /lib/tls/i686/cmov/libc.so.6
#2  0xb7e6f93a in usleep () from /lib/tls/i686/cmov/libc.so.6
#3  0x080483ad in main () at hello-loop.c:7
```

典型 系統呼叫 (1)


main.c
(/lib/libc.so.6)
linux/arch/i386/kernel/entry.S
linux/fs/open.c
(instructions)
close(fd);
(instructions)
movl %ebx, %edx
movl 0x4(%esp,1),%ebx
movl $0x6, %eax
int $0x80
movl %edx,%ebx
cmp $0xfffff001,%eax
jae syscall_error
ret
pushl %eax
SAVE_ALL
GET_CURRENT(%ebx)
testb $0x02,tsk_ptrace(%ebx)
jne tracesys
cmpl $(NR_syscalls),%eax
jae badsys
call *sys_call_table(,%eax,4)
movl %eax,EAX(%esp)
cli
cmpl$0,need_resched(%ebx)
jne reschedule
cmpl $0,sigpending(%ebx)
jne signal_return
RESTORE_ALL
int sys_close(int fd)
{
(instructions)
}
user mode
kernel mode


processes
not involved
system
calls
user mode
kernel mode
assembler–glue
syscall multiplexer
syscalls
not involved

典型 系統呼叫 (2)

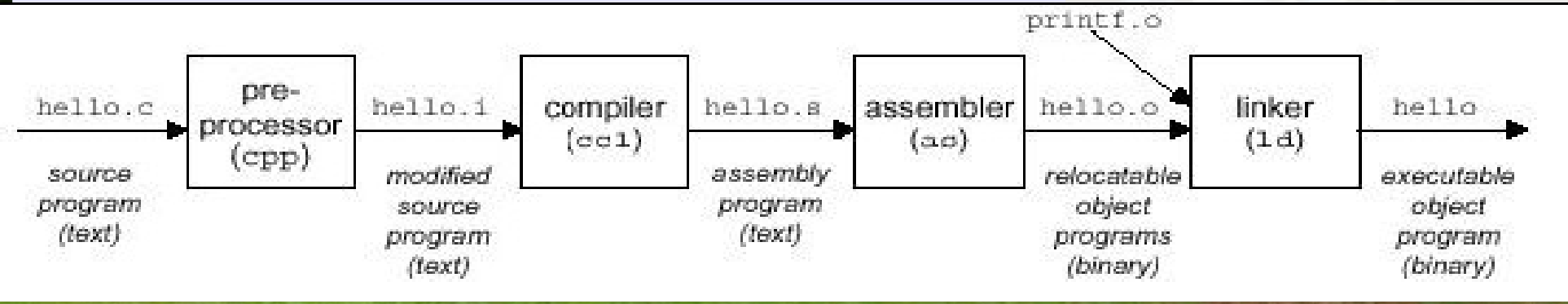
- 在 x86 保護模式，處理 int 中斷指令時， CPU 需要作繁複的查表與確認動作，才得以切換執行權，當 Kernel 執行系統呼叫完畢後，以 iret 返回，該指令恢復目前的 stack ， 並回到原本的執行
  - CPL <= DPL
- 由 Ring3 進入 Ring0 的過程浪費許多 CPU 週期

回頭看 hello.c 的編譯過程

```
$ gcc -v -o hello hello.c
Using built-in specs.
Target: i486-linux-gnu
...
 /usr/lib/gcc/i486-linux-gnu/4.1.2/collect2 --eh-frame-hdr
-m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o hello
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../lib/crt1.o
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../lib/crti.o
/usr/lib/gcc/i486-linux-gnu/4.1.2/crtbegin.o -L
/usr/lib/gcc/i486-linux-gnu/4.1.2 -L/usr/lib/gcc/i486-
linux-gnu/4.1.2 -L/usr/lib/gcc/i486-linux-
gnu/4.1.2/../../../../lib -L/lib/../lib -L/usr/lib/../lib
/tmp/ccyj1YoV.o -lgcc --as-needed -lgcc_s --no-as-
needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/i486-linux-gnu/4.1.2/crtend.o
/usr/lib/gcc/i486-linux-gnu/4.1.2/../../../../lib/crtn.o
$ wc -c hello
6749 hello
```

crt*.o
C Runtime object files

```
$ cat hello.c
int main() {
   printf("Hello World\n");
   return 0;
}
$ strace ./hello
...

write(1, "Hello World\n", 12Hello World.) = 12
exit_group(0)                           = ?
Process 14211 detached
```



系統呼叫 (1)

```
$ cat hello-write.c
#include <unistd.h>
int main()
{
    write(1, "Hello World\n", 12);
    return 0;
}
$ ./hello-write
Hello World
```

- 改用 POSIX 之 write 系統呼叫
- 但還不是最「直接」的使用系統呼叫
- 透過 libc wrapper 實現

```
/* Write N bytes of BUF to FD.  Return the number written, or -1.

   This function is a cancellation point and therefore not marked with
   __THROW.  */
extern ssize_t write (int __fd, __const void *__buf, size_t __n) __wur;
```

(/usr/include/unistd.h)

系統呼叫 (2)

```
$ cat hello-syscall.c
#include <asm/unistd.h>

static int errno;
_syscall1(int, exit, int, status);
_syscall3(int, write,
          int, fd,
          const void*, buf,
          unsigned long, count);

int main()
{
    write(1, "Hello World\n", 12);
    exit(0);
}
$ ./hello-syscall
Hello World
```

- linux-2.6.17/include/asm-i386/unistd.h
- gcc -o hello-syscall \
  -D__KERNEL__ \
  -fno-builtin \
  -fomit-frame-pointer \
  -I/lib/modules/`uname -r`/build/include \
  hello-syscall.c

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("push %%ebx ; movl %2,%%ebx ; int
          $0x80 ; pop %%ebx" \
        : "=a" (__res) \
        : "0" (__NR_##name),"ri" ((long)(arg1)) :
          "memory"); \
__syscall_return(type,__res); \
}
```

```
extern ssize_t write (int __fd, __const void *__buf, size_t __n) __wur;
```

(/usr/include/unistd.h)

系統呼叫 (3)

```
$ cat hello-syscall2.c
#include <asm/unistd.h>

static int errno;
_syscall1(int, exit, int, status);
_syscall3(int, write,
    int, fd, const void*, buf,
    unsigned long, count);

void hello()
{
      write(1, "Hello World\n", 12);
      exit(0);
}
```

```
$ gcc -c -Os -D__KERNEL__ \
          -fno-builtin -fomit-frame-pointer \
          -I/lib/modules/`uname -r`/build/include \
          hello-syscall2.c
$ ld -o hello-syscall2 \
          -e hello \
          hello-syscall2.o
$ wc -c hello-syscall2
977 hello-syscall2
```

```
#include <stdio.h>
char message[] = "Hello,
world!\n";
int main(void) {
  long _res;
  __asm__ volatile (
    "int $0x80"
    : "=a" (_res)
    : "a" ((long) 4),
      "b" ((long) 1),
      "c" ((long) message),
      "d" ((long) sizeof(message)));
    return 0;
}
```

```
$ gcc -c -E  -Os -D__KERNEL__ -fno-builtin -fomit-frame-pointer \
-I/lib/modules/`uname -r`/build/include hello-syscall2.c
...
int write(int fd,const void* buf,unsigned long count) { long __res; __asm__ volatile
("push %%ebx ; movl %2,%%ebx ; int $0x80 ; pop %%ebx" : "=a" (__res) : "0"
(4),"ri" ((long)(fd)),"c" ((long)(buf)), "d" ((long)(count)) : "memory"); do { if
((unsigned long)(__res) >= (unsigned long)(-(128 + 1))) { errno = -(__res); __res =
-1; } return (int) (__res); } while (0); };
...
```

系統呼叫 (4)

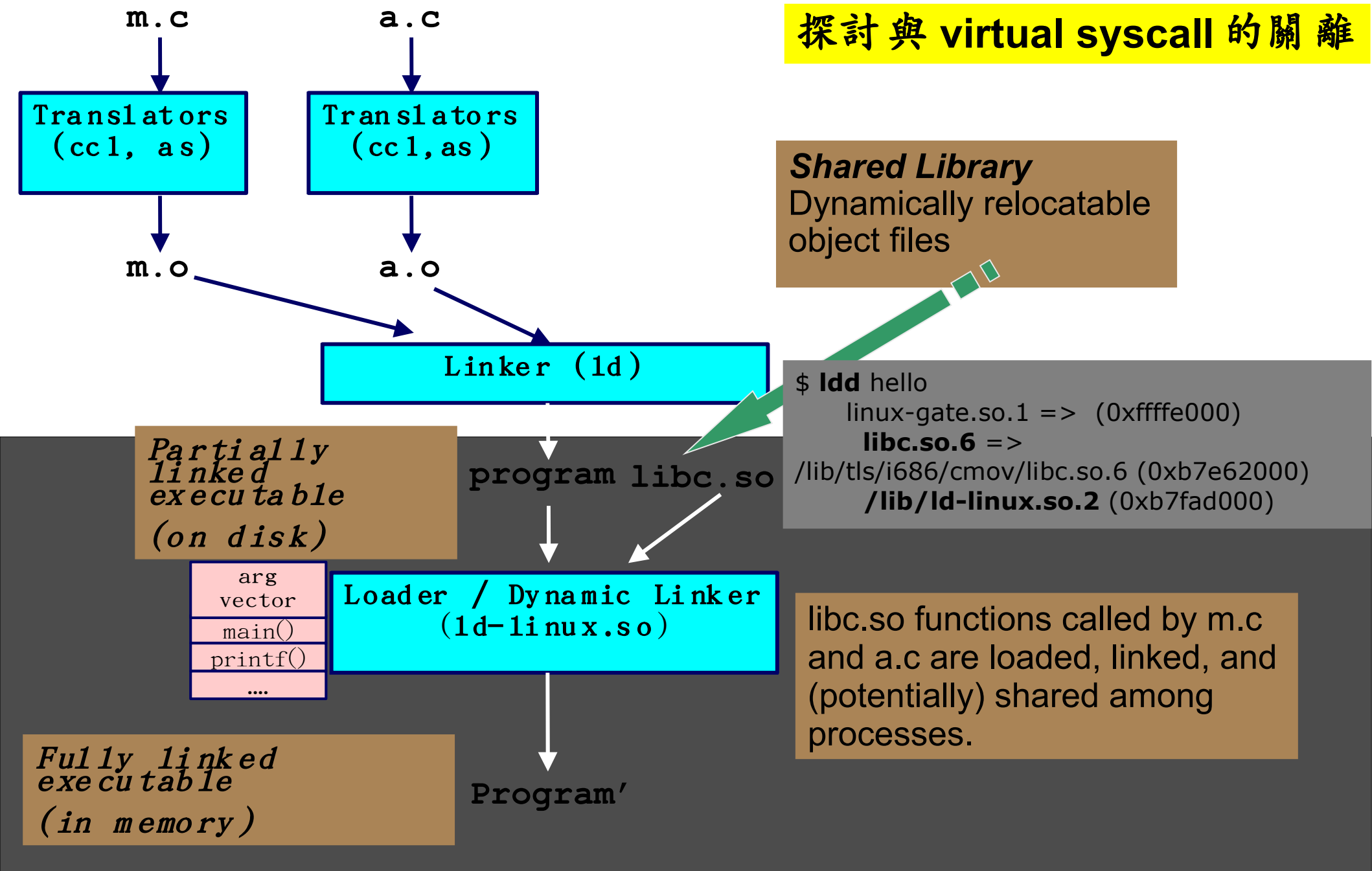
```
$ cat hello-syscall3.c
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

int main()
{
        int ret;
        ret = syscall(__NR_write, 1, "Hello World\n", 12);
        return 0;
}
$ ./hello-syscall3
Hello World
```

```
#include <stdio.h>
char message[] = "Hello,
world!\n";
int main(void) {
  long _res;
  __asm__ volatile (
    "int $0x80"
    : "=a" (_res)
    : "a" ((long) 4),
      "b" ((long) 1),
      "c" ((long) message),
      "d" ((long) sizeof(message)));
    return 0;
}
```

```
$ head /usr/include/bits/syscall.h
/* Generated at libc build time from kernel syscall list.  */

#ifndef _SYSCALL_H
# error "Never use <bits/syscall.h> directly; include <sys/syscall.h>
instead."
#endif

#define SYS__llseek __NR__llseek
#define SYS__newselect __NR__newselect
#define SYS__sysctl __NR__sysctl
```

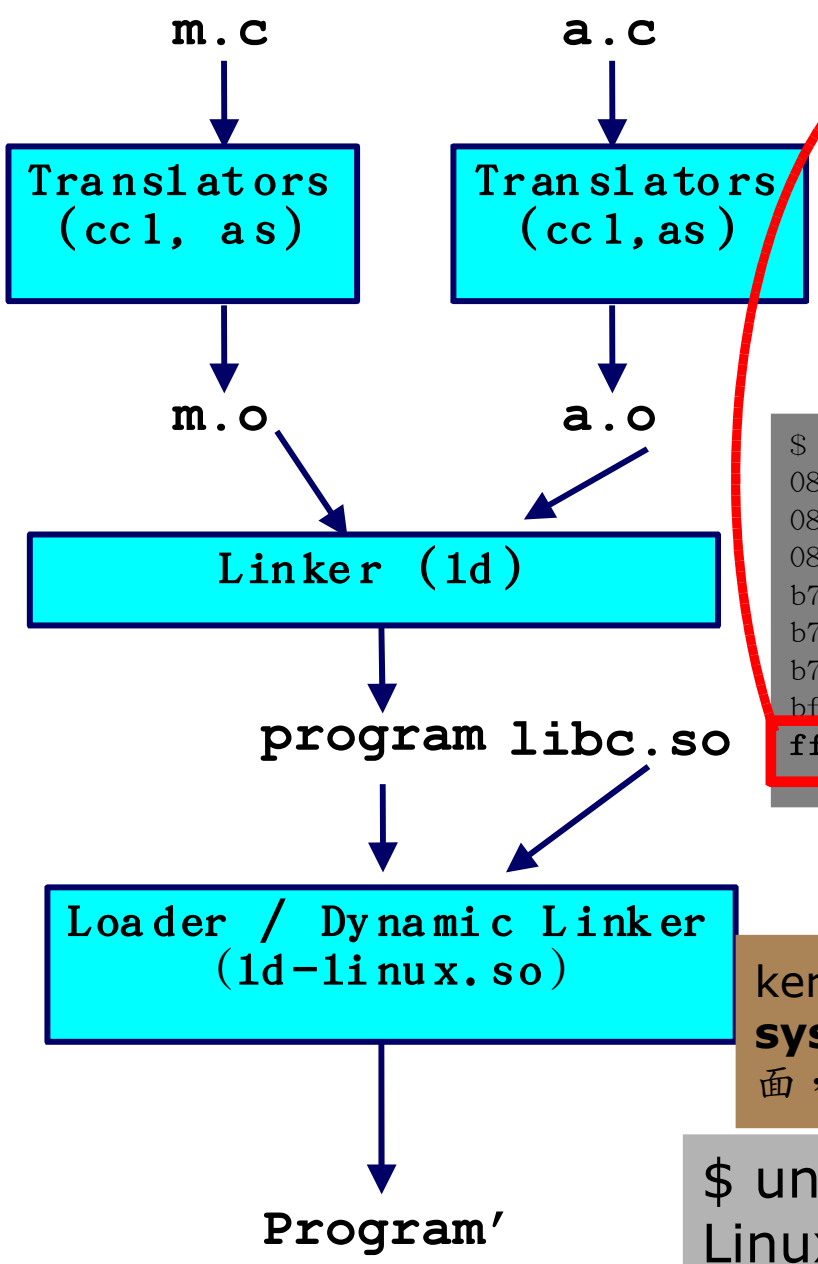
```
$ head -n 12 /usr/include/asm-
i386/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call
numbers.
 */

#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
```

Dynamically Linked Shared Libraries(1)



探討與 virtual syscall 的關離

Dynamically Linked Shared Libraries(2)




```
$ ldd hello
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 =>
/lib/tls/i686/cmov/libc.so.6 (0xb7e62000)
        /lib/ld-linux.so.2 (0xb7fad000)
```

/proc/self/maps
Memory map

```
$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 31222     /bin/cat
0804c000-0804d000 rw-p 00003000 31222     /bin/cat
0804d000-0806e000 rw-p 0804d000 0             [heap]
b7f90000-b7f92000 rw-p b7f90000 0
b7f92000-b7fab000 r-xp 00000000 295771    /lib/ld-2.4.so
b7fab000-b7fad000 rw-p 00018000 295771    /lib/ld-2.4.so
bfd95000-bfdab000 rw-p bfd95000  0             [stack]
ffffe000-fffff000 ---p 00000000  0             [vdso]
```

vdso : Virtual DSO

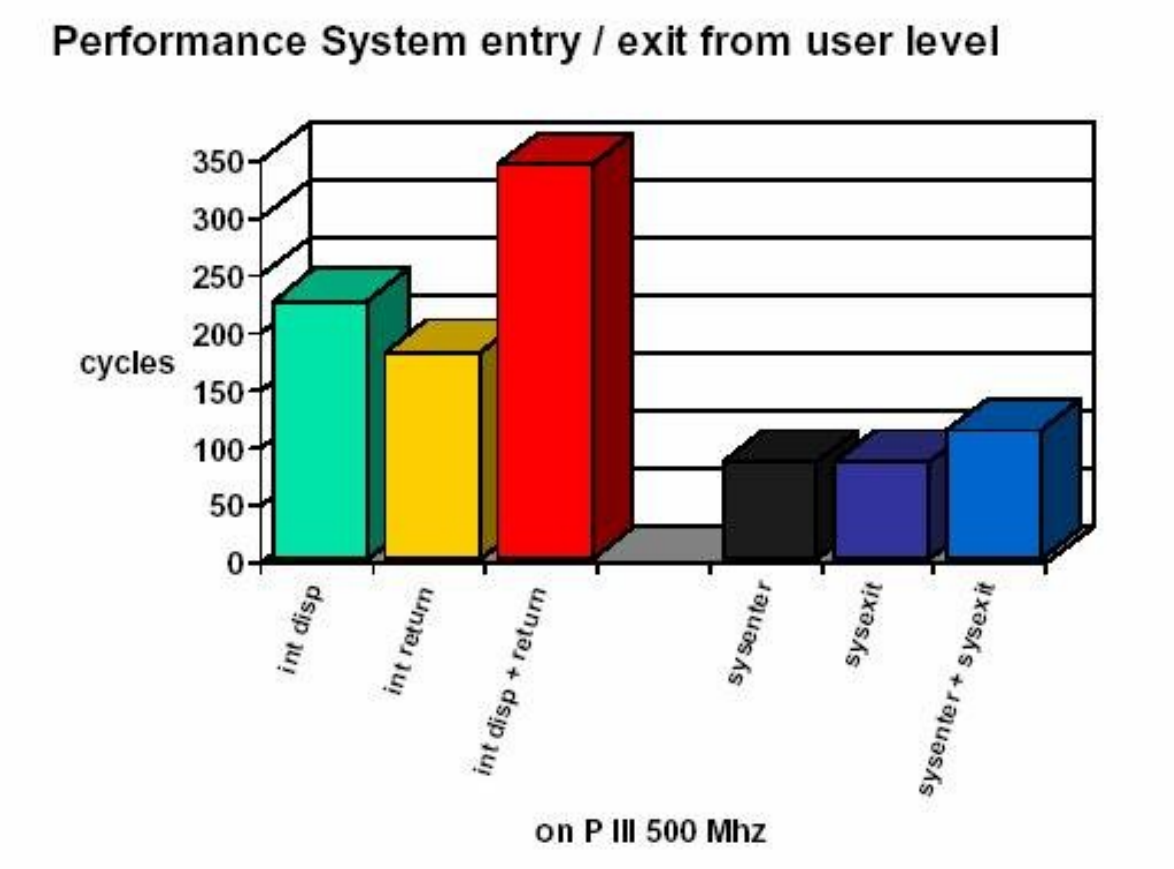
kernel 2.6 支援 Pentium III+ 的快速系統呼叫 **sysenter / sysexit** 機制。核心提供原先的 int 0x80 ，與新的 **sysenter** 雙介面，並虛擬印射了 **linux-gate.so.1** 來實作

```
$ uname -a
Linux venux 2.6.15-23-686 #1 SMP PREEMPT
Tue May 23 14:03:07 UTC 2006 i686 GNU/Linux
```

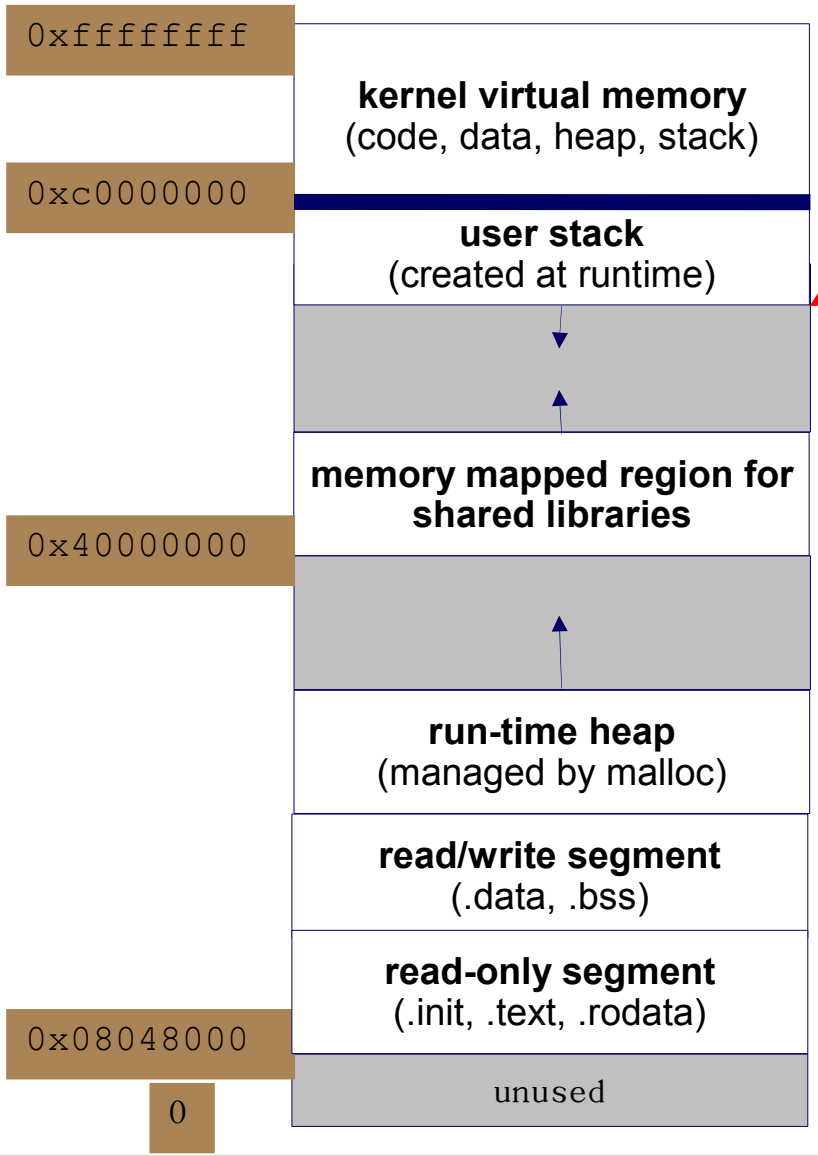
int80 vs. sysenter/sysexit

耗費時間	User-Mode	Kernel-Mode
以 **int80** 為基礎的系統呼叫	17.500ms	7.00ms
以 **systenter/ sysexit** 為基礎	**9.833ms**	**6.833ms**

Intel® Pentium® III CPU, 450 MHzProcessor Family: 6 Model: 7 Stepping: 2



Dynamically Linked Shared Libraries(3)



memory invisible to user code

%esp (stack pointer)

brk

loaded from the executable file

```
$ ldd hello
    linux-gate.so.1 =>  (0xffffe000)
    libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e62000)
    /lib/ld-linux.so.2 (0xb7fad000)
```

/proc/self/maps
Memory map

```
$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 31222     /bin/cat
0804c000-0804d000 rw-p 00003000 31222     /bin/cat
0804d000-0806e000 rw-p 0804d000 0             [heap]
b7f90000-b7f92000 rw-p b7f90000 0
b7f92000-b7fab000 r-xp 00000000 295771    /lib/ld-2.4.so
b7fab000-b7fad000 rw-p 00018000 295771    /lib/ld-2.4.so
bfd95000-bfdab000 rw-p bfd95000 0             [stack]
ffffe000-fffff000 ---p 00000000 0             [vdso]
```

vdso Virtual DSO

auxiliary vector (auxv)

argv ➔ program arguments

envp ➔ environment variables

適用新建立的 Process

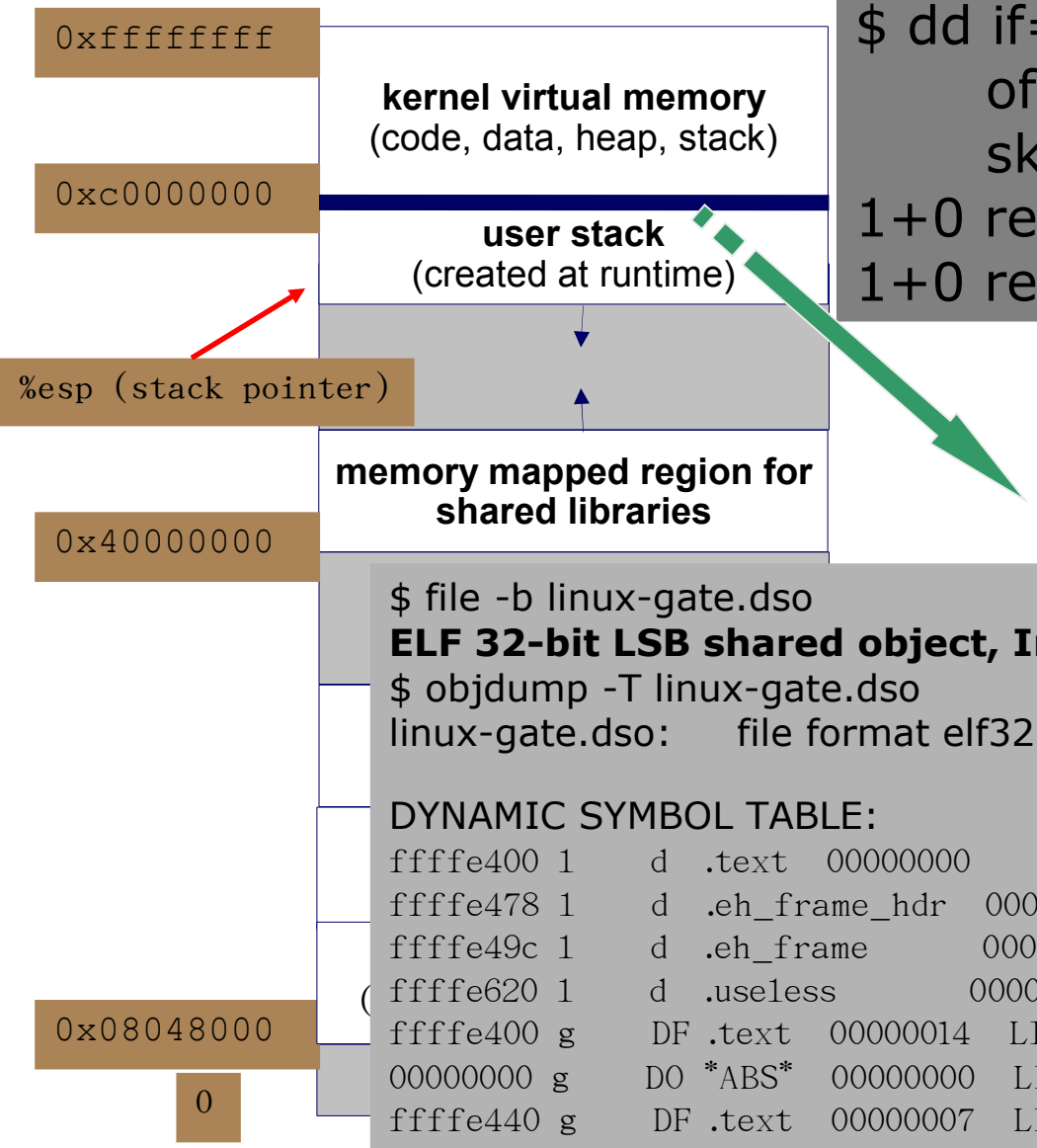
```
$ gdb -c core -q
Core was generated by `hello'.
Program terminated with signal 11, Segmentation fault.
#0  0xffffe777 in ?? ()
(gdb) bt
#0  0xffffe777 in ?? ()
```

VA stack exploit

Ref: Exploiting with linux-gate.so.1 . by Izik

```
# JMP *%ESP @ linux-gate.so.1jmp    = "\x77\xe7\xff\xff"
```

Dynamically Linked Shared Libraries(4)



/proc/self/mem
Memory snapshot

```
$ dd if=/proc/self/mem \
     of=linux-gate.dso bs=4096 \
     skip=1048574 count=1
1+0 records in
1+0 records out
```

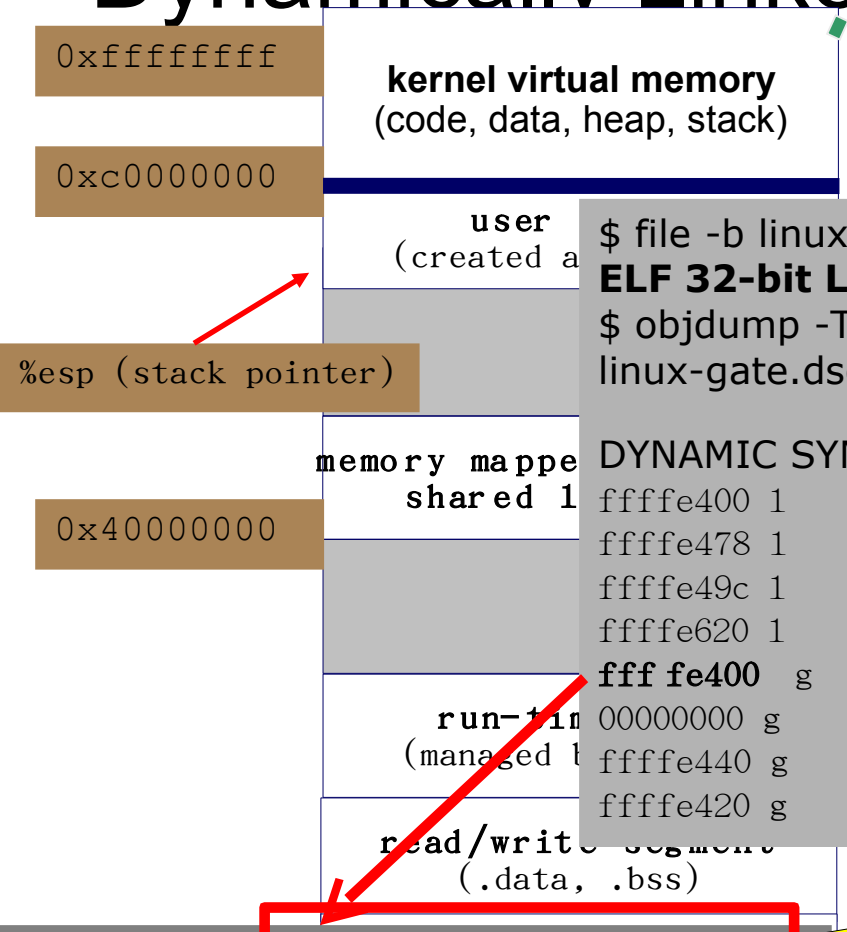
2^{20} = **1048576**

virtual system call

```
$ file -b linux-gate.dso
ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), stripped
$ objdump -T linux-gate.dso
linux-gate.dso:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
ffffe400 l    d  .text  00000000              .text
ffffe478 l    d  .eh_frame_hdr  00000000              .eh_frame_hdr
ffffe49c l    d  .eh_frame  00000000              .eh_frame
ffffe620 l    d  .useless  00000000              .useless
ffffe400 g    DF .text  00000014  LINUX_2.5   __kernel_vsyscall
00000000 g    DO *ABS*  00000000  LINUX_2.5   LINUX_2.5
ffffe440 g    DF .text  00000007  LINUX_2.5   __kernel_rt_sigreturn
ffffe420 g    DF .text  00000008  LINUX_2.5   __kernel_sigreturn
```

```
$ LD_TRACE_LOADED_OBJECTS=1 ./hello
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e9b000)
        /lib/ld-linux.so.2 (0xb7fe6000)
```

Dynamically Linked Shared Libraries(5)



virtual system call

```
$ file -b linux-gate.dso
ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), stripped
$ objdump -T linux-gate.dso
linux-gate.dso:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
ffffe400 l    d  .text  00000000              .text
ffffe478 l    d  .eh_frame_hdr  00000000              .eh_frame_hdr
ffffe49c l    d  .eh_frame      00000000              .eh_frame
ffffe620 l    d  .useless       00000000              .useless
ffffe400 g     DF .text  00000014  LINUX_2.5   __kernel_vsyscall
00000000 g     DO *ABS*  00000000  LINUX_2.5   LINUX_2.5
ffffe440 g     DF .text  00000007  LINUX_2.5   __kernel_rt_sigreturn
ffffe420 g     DF .text  00000008  LINUX_2.5   __kernel_sigreturn
```

__kernel_vsyscall 進入點

```
$ objdump -d --start-address=0xffffe400 \
--stop-address=0xffffe414 linux-gate.dso

linux-gate.dso:     file format elf32-i386
Disassembly of section .text:

ffffe400 <__kernel_vsyscall>:
ffffe400:       51              push   %ecx
ffffe401:       52              push   %edx
ffffe402:       55              push   %ebp
ffffe403:       89 e5           mov    %esp,%ebp
ffffe405:       0f 34           sysenter
```

```
ffffe407:       90              nop
ffffe408:       90              nop
ffffe409:       90              nop
ffffe40a:       90              nop
ffffe40b:       90              nop
ffffe40c:       90              nop
ffffe40d:       90              nop
ffffe40e:       eb f3           jmp    ffffe403
<__kernel_vsyscall+0x3>
ffffe410:       5d              pop    %ebp
ffffe411:       5a              pop    %edx
ffffe412:       59              pop    %ecx
ffffe413:       c3              ret
```

虛擬系統呼叫

```
$ cat hello.c
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>
int main() {
        int ret;
        char message[] = "Hello, world!\n";
        __asm__ volatile (
                "call *%2"
                : "=a" (ret)
                : "a" (__NR_write),
                  "S" (0xffffe400),
                  "b" ((long) 1),
                  "c" ((long) message),
                  "d" ((long) sizeof(message)));
        return 0;
}
$ ./hello
Hello, world!
```

```
$ objdump -d --start-address=0xffffe400 \
--stop-address=0xffffe414 linux-gate.dso

linux-gate.dso:     file format elf32-i386
Disassembly of section .text:

ffffe400 <__kernel_vsyscall>:
ffffe400:       51              push   %ecx
ffffe401:       52              push   %edx
ffffe402:       55              push   %ebp
ffffe403:       89 e5           mov    %esp,%ebp
ffffe405:       0f 34           sysenter
```

以 **procedure call** 的形式呼叫了 **system call**

再探動態連結

- glibc 的 HWCAP 機制
- Rundll32.exe on Linux

glibc 的 HWCAP 機制 (1)

- $ COLUMNS=200 dpkg -l | grep libc6
  - ii **libc6** 2.5-0ubuntu2 GNU C Library: Shared libraries
  - ii libc6-dev 2.5-0ubuntu2 GNU C Library: Development Libraries and Header Files
  - ii **libc6-i686** 2.5-0ubuntu2 GNU C Library: Shared libraries [**i686 optimized**]
- 一般 i386 與 i686 最佳化的 glibc 如何共存?
- 特定之函數，如數學運算，如何在執行時期針對硬體挑選最佳的實做?
- 引入 HWCAP (Hardware Capacities) 的機制
  - LD_SHOW_AUXV (AUXiliary Vector)

glibc 的 HWCAP 機制 (2)

```
$ ldd hello
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e8d000)
        /lib/ld-linux.so.2 (0xb7fe5000)
$ realpath /lib/libc.so.6
/lib/libc-2.5.so
$ strace -f ./hello
execve("./hello", ["./hello"], [/* 28 vars */]) = 0
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1311200, ...}) = 0
...
```

cmov = Pentium4 SSE2 Conditional MOVe

glibc 的 HWCAP 機制 (3)

```
$ LD_SHOW_AUXV=1 ./hello
AT_SYSINFO:      0xffffe400
AT_SYSINFO_EHDR: 0xffffe000
AT_HWCAP:    fpu vme de pse tsc msr mce cx8 sep mtrr pge mca cmov pat
clflush dts acpi mmx fxsr sse sse2 tm pbe
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x8048034
AT_PHENT:        32
AT_PHNUM:        7
AT_BASE:         0xb7fca000
AT_FLAGS:        0x0
AT_ENTRY:        0x80482b0
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_PLATFORM:     i686
Hello World!
```

Rundll32.exe on Linux(1)

- MS-Windows Rundll.exe 允許透過命令列載入 DLL 並呼叫其中的 function
  - Rundll32.exe DllFileName FuncName
- izik(http://www.tty64.org) 針對 x86/Linux 撰寫 Runlib32
  - ./runlib libc.so.6,puts \""Hello World"\"

```
$ ./runlib -v -x printf-out libc.so.6,puts \""Hello World"\"
puts[<0xb7ed8610>]@libc.so.6[]
------------------------------------------------------
* Stack Generated (1 parameters, 4 bytes)
--------------------------------------
Generated Assembly
-----------------
 * pushl $0xbfce7c9a
 * call 0xb7ed8610
```

```
Streams Buffers
---------------
 * Standart Output (STDOUT) : 15 bytes
 * Standart Error  (STDERR) : 0 bytes

Function Result
---------------
 * Pointer: No
 * Value: 12
$ cat printf-out
Hello World
```

Rundll32.exe on Linux(2)

src/lib.c

```
/*
 * Manually pushing the function arguments to
* the stack
 */

if (ptr->stack) {
        for (j = 0; j < ptr->stack->stack_items; j++) {
                __asm__ __volatile__ (\
                        "pushl %0 \n" \
                        : /* no output */ \
                        : "r" (ptr->stack->stack[j]) \
                        : "%eax" \
                        );
        }
}
```

```
/*
 * Make the CALL!
 */
ret = (unsigned long) ptr->fcn_handler();

/*
 * Be polite, let's clean the stack afterward
 */
if (ptr->stack) {
        ptr->stack->stack_items *= sizeof(long);
        __asm__ __volatile__ (\
                "addl %0, %%esp \n" \
                : /* no output */ \
                : "r" (ptr->stack->stack_items) \
                : "%esp" );
        ptr->stack->stack_items /= sizeof(long);
}

s_errno = errno;
signal(SIGSEGV, SIG_DFL);
```

Rundll32.exe on Linux(3)

- Trampoline (Assembly/Machine_code-C interfacing)
- 類似技術廣泛應用於:
  - ffcall (GNUstep/Objective-C)
  - libffi (GNU GCC)
  - JIT compiler (Kaffe, Hotspot, ...)
  - Boot-straping code in dynamic programming language engine

參考資料

- 《Binary Hacks》， O'Reilly Japan
- Linkers and Loaders
  - http://www.iecc.com/linker/
- Startup state of a Linux/i386 ELF binary
  - http://asm.sourceforge.net/articles/startup.html
- IA32 上 Linux 內核中斷機制分析
  - http://www.whitecell.org/list.php?id=23
- FFCALL
  - http://www.haible.de/bruno/packages-ffcall.html
- Linux 2.6 對新型 CPU 快速系統調用的支持
  - http://www-128.ibm.com/developerworks/cn/linux/kernel/l-k26ncpu/