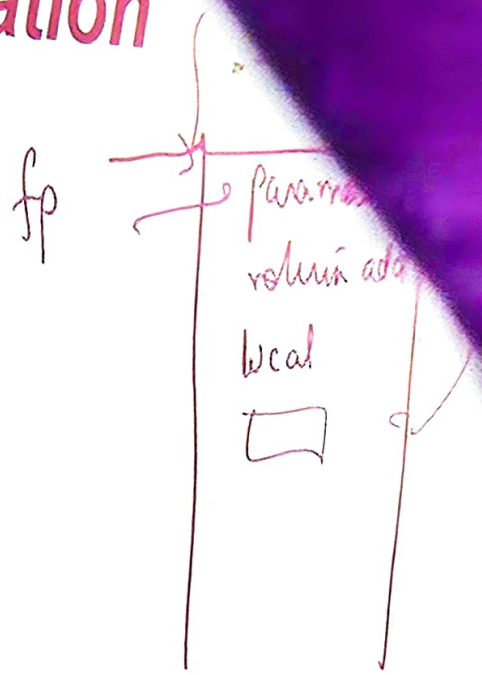
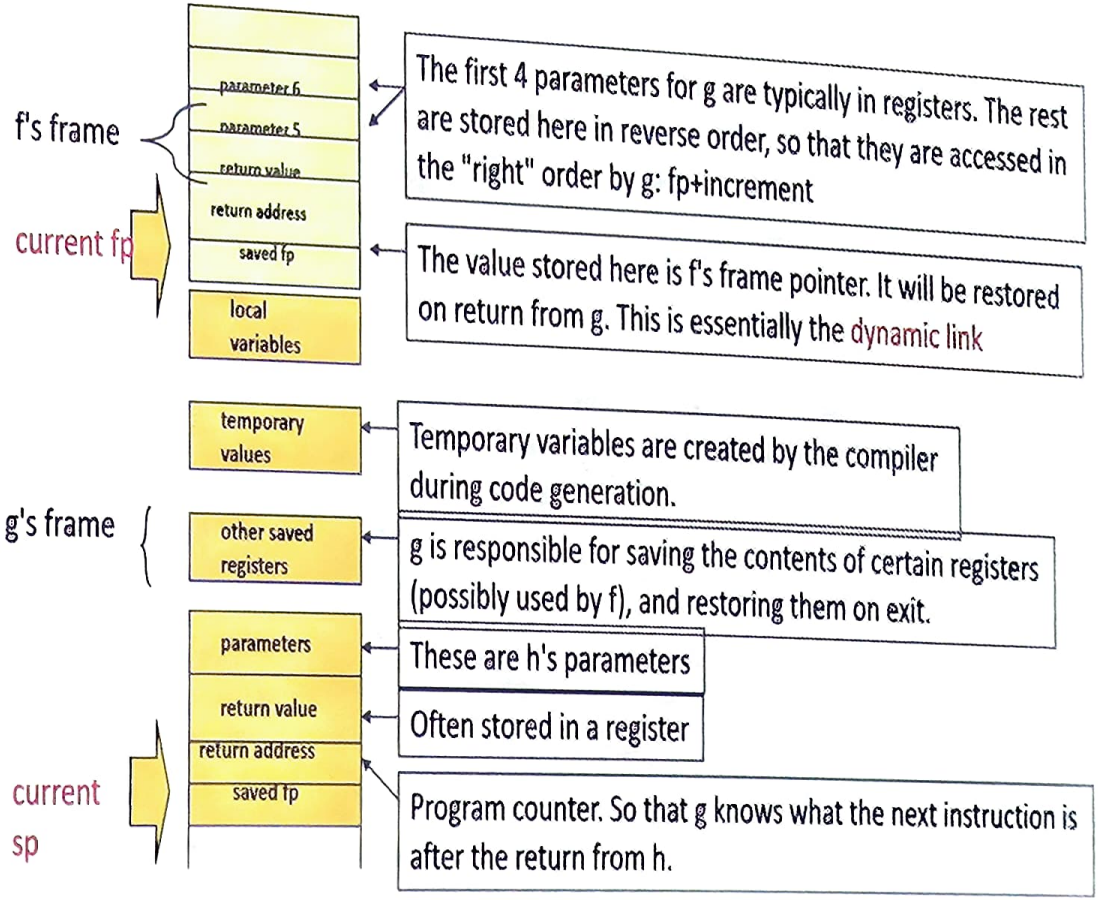
Storage organization

- Stack frames. What's typically in them?
  - parameters
  - local variables
  - temporary values
  - return value
  - stack pointer
  - frame pointer
  - return address
    - (saved pc)
  - dynamic link
    - Reference to the frame of the caller, used to restore the stack after the call.
  - static link
    - Reference to the nearest frame of the lexically enclosing procedure. Used for non-local accesses in languages that support statically nested scopes.



Storage organization

- Typical stack layout. Assume function f calls g, and g is about to call h



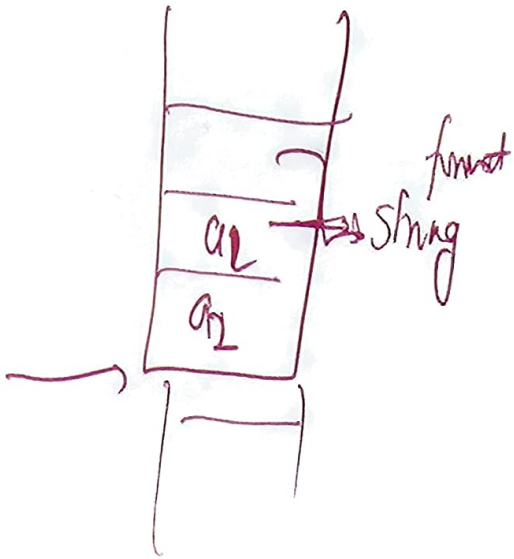
$a_1 = 1$

$a_2 = 2$

$a_3 = 3$

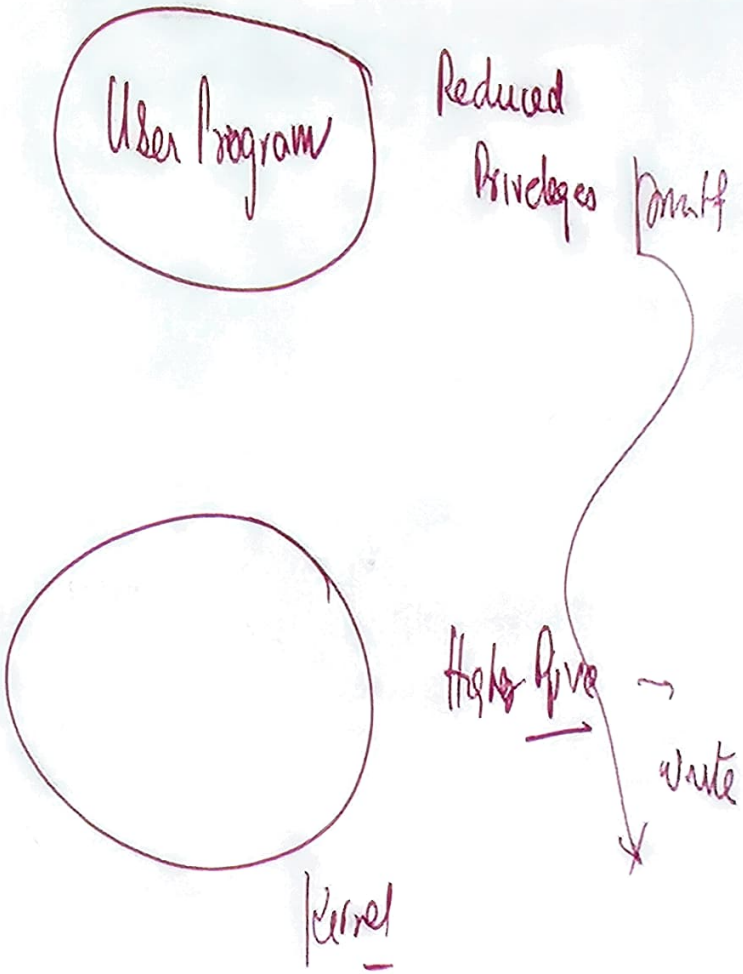
printf ("%d", a1..)

printf ("%d, %d, %p, \n", a1, a2...)



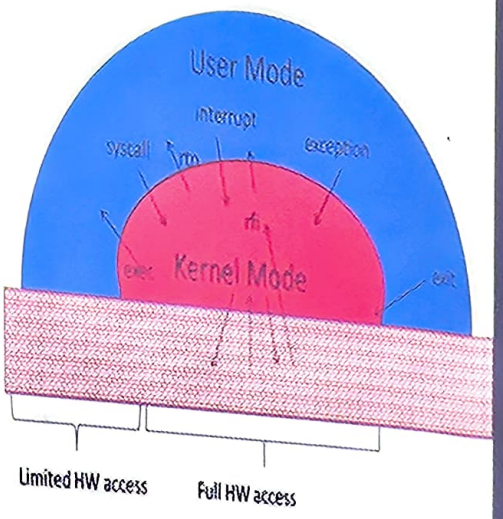
Dual Mode

- **bit** of state (user/kernel mode)
- Certain actions only permitted in kernel mode
  - Which page table entries can be accessed
- User => Kernel mode sets kernel mode, saves user PC
  - Only transition to *OS-designated addresses*
  - OS can save the rest of the user state if necessary
- Kernel => User mode sets user mode, restores user PC, ...


User Program
Reduced
Privileges
Kernel

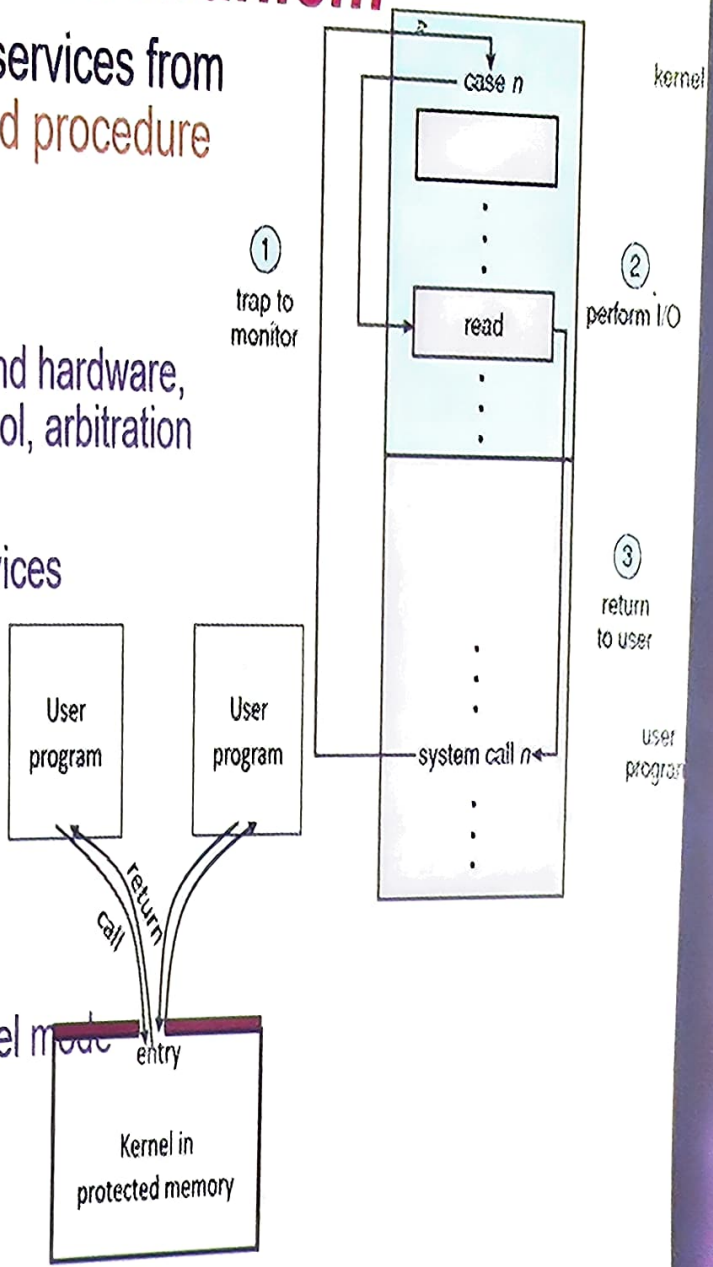
Privileged Mode

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - eg. Timer, I/O device
  - Independent of user process
- Trap
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- All 3 *exceptions* are an UNPROGRAMMED CONTROL TRANSFER
  - Where does it go?



System Call Mechanism

- User-level processes (clients) request services from the kernel (server) via special "protected procedure calls"
- System calls provide:
  - An abstraction layer between processes and hardware, allowing the kernel to provide access control, arbitration
  - A virtualization of the underlying system
  - A well-defined "API" (ASI?) for system services
- Why?
  - User code can be arbitrary
  - User code cannot modify kernel memory
- Solution principle
  - Makes a system call with parameters
  - The call mechanism switches code to kernel mode
  - Execute system call
  - Return with results



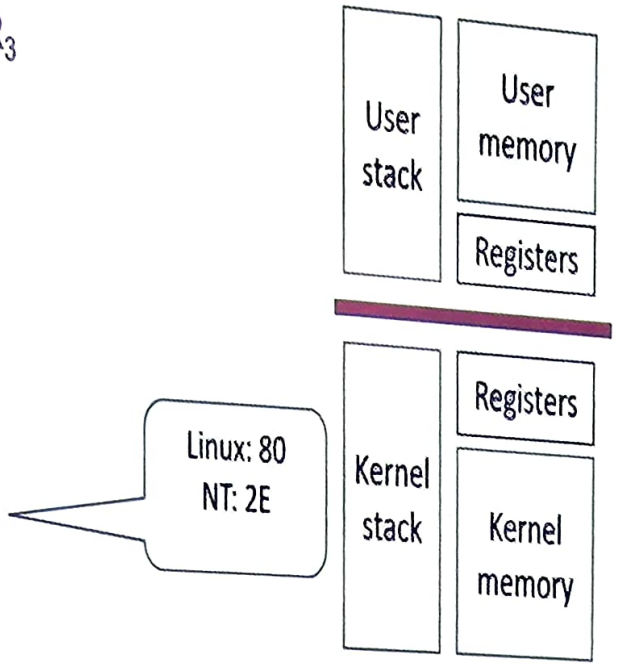
System Call

- Initiating a system call is known as "trapping into the kernel" and is usually effected by a software initiated interrupt to the CPU
  - Example: Intel ("int 80h"), ARM ("swi")
- CPU saves current context, changes mode and transfers to a well-defined location in the kernel
- System calls are designated by small integers; once in the kernel, a dispatch table is used to invoke the corresponding kernel function
- A special assembly instruction (Intel "iret") is used to return to user mode from kernel mode

Library Stubs for System Calls

- Use read(fd, buf, size) as an example:

```
int read( int fd, char * buf, int size)
{
   move fd, buf, size to      R1, R2, R3
   move READ to R0
   int $0x80
   move result to Rresult
}
```



System Call Entry Point

- Assume passing parameters in registers

  EntryPoint:

  switch to kernel stack

  save context

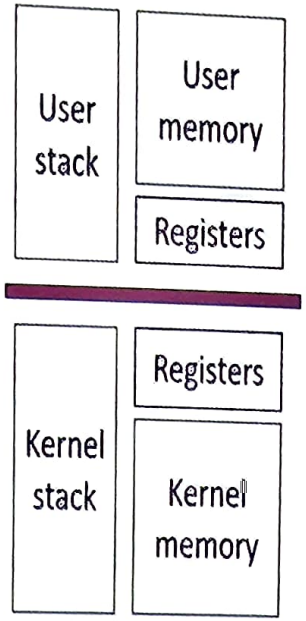
  check R_0

  call the real code pointed by R_0

  restore context

  switch to user stack

  iret (change to user mode and return)



Implementing System Calls

- Initiating a system call is known as "trapping into the kernel" and is usually effected by a software initiated interrupt to the CPU
  - Example: Intel ("int 80h"), ARM ("swi")
- CPU saves current context, changes mode and transfers to a well-defined location in the kernel
- System calls are designated by small integers; once in the kernel, a dispatch table is used to invoke the corresponding kernel function
- A special assembly instruction (Intel "iret") is used to return to user mode from kernel mode

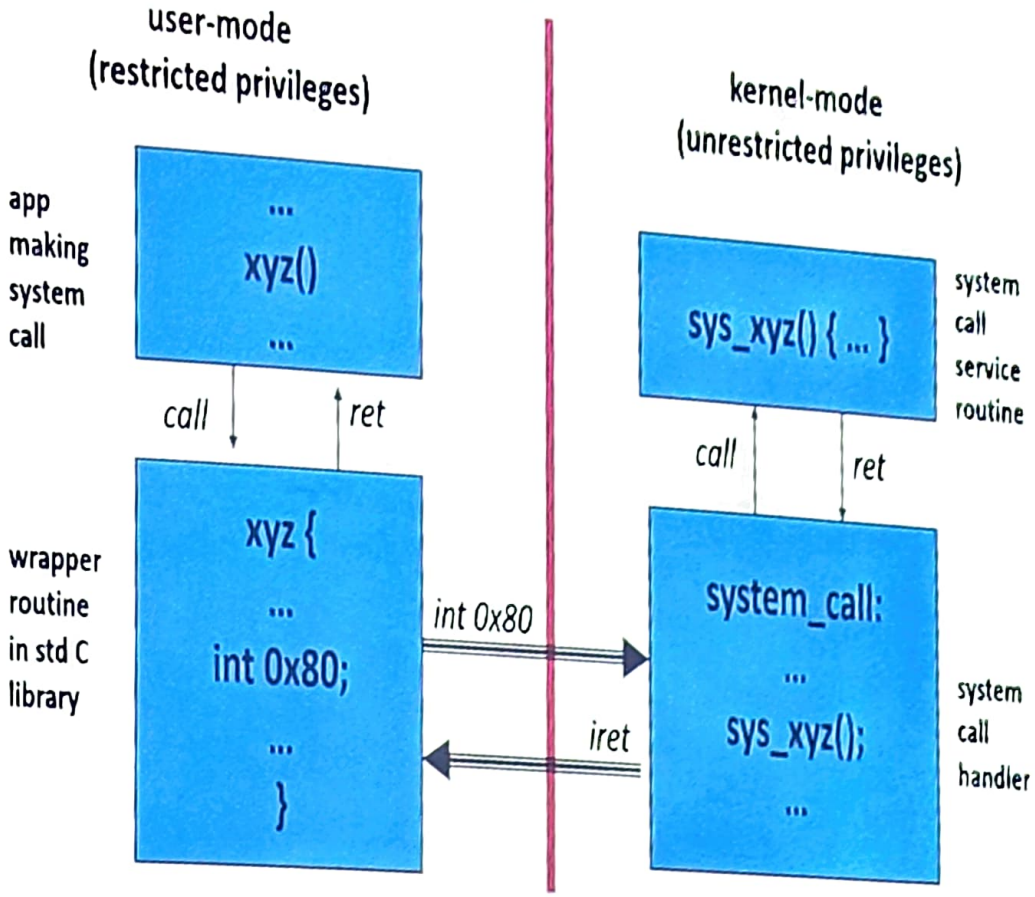
System Calls vs. Library Calls

- System calls can only be initiated by assembly code (special software interrupt instructions)
- Processes normally call library "wrapper routines" that hide the details of system call entry/exit
- Library calls are much faster than system calls
  - If you can do it in user space, you should
- Library calls (man 3), system calls (man 2)
- Some library functions:
  - never call syscalls (strlen),
  - some always call syscalls (mmap),
  - some occasionally call syscalls (printf)

Trapping into the Kernel

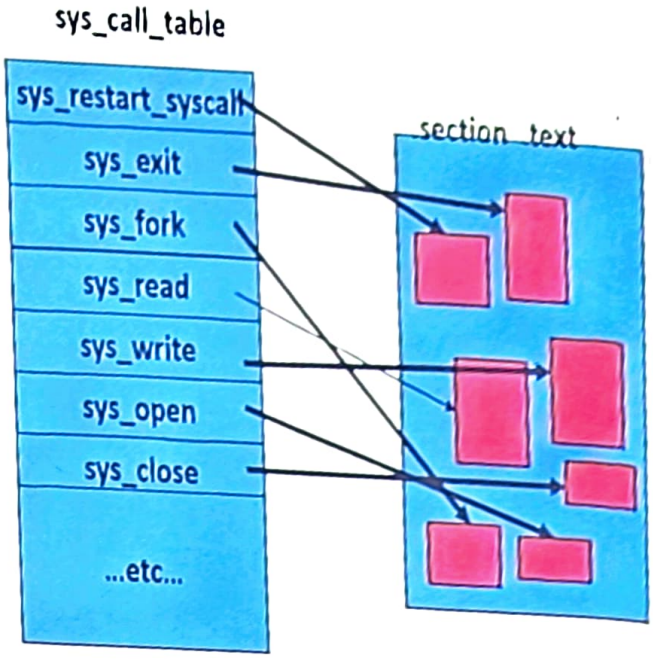
- Trapping into the kernel involves executing a special assembly instruction that activates the interrupt hardware just like a device interrupt would but it is done by software instead so it is known as a "software interrupt"
- Intel uses the "int" (interrupt) instruction with the operand "80h" (80 hex = 128), indicating the interrupt handler that should be executed
- "int 80h" ultimately causes control to transfer to the assembly label: system_call in the kernel (found in arch/kernel/i386/entry.S)
- Every process in Linux has the usual user-mode stack as well as a small, special kernel stack that is part of the kernel address space; trapping involves switching the stack pointer to the kernel stack (and back)
- Early UNIXes had about 60 system calls, Linux 2.6 has about 300; Solaris more, Window more still

Invoking System Calls



The system-call jump-table

- There are approximately 400+ system-calls
- Any specific system-call is selected by its ID-number (it's placed into register %eax)
- It would be inefficient to use if-else tests or even a switch-statement to transfer to the service-routine's entry-point
- Instead an array of function-pointers is directly accessed (using the ID-number)
- This array is named '**sys_call_table[]**'



Assembly language (.data)

```
        .section    .data
sys_call_table:
        .long       sys_restart_syscall
        .long       sys_exit
        .long       sys_fork
        .long       sys_read
        .long       sys_write
        // …etc (from 'arch/i386/kernel/entry.S')
```

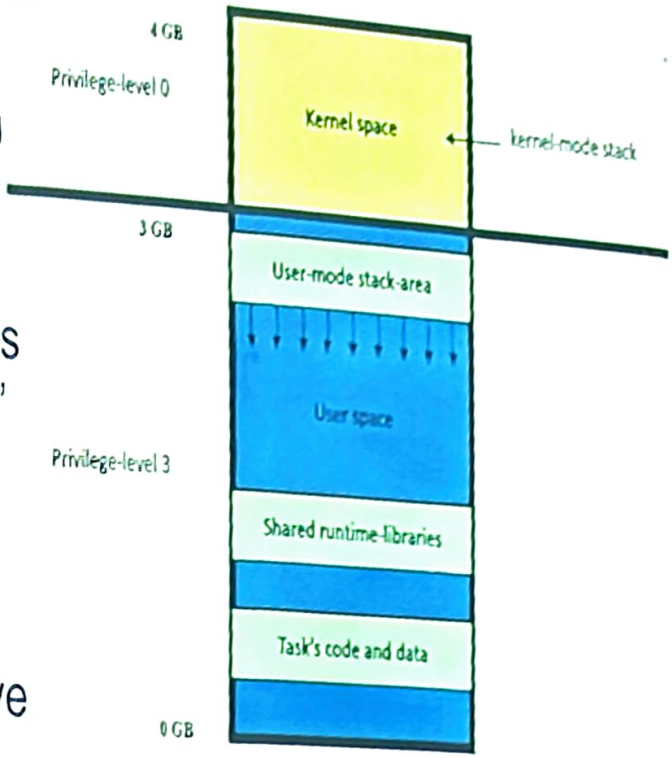

Assembly language (.text)

```
        .section    .text
system_call:
  // copy parameters from registers onto stack...
        call    sys_call_table(, %eax, 4)
        jmp     ret_from_sys_call

ret_from_sys_call:
  // perform rescheduling and signal-handling...
        iret            // return to caller (in user-mode)
```

Some Details

- Syscall Naming Convention
  - Usually a library function "foo()" will do some work and then call a system call ("sys_foo()")
  - In Linux, all system calls begin with "sys_"
  - Often "sys_foo()" just does some simple error checking and then calls a worker function named "do_foo()"
- Process Address Space
- Return Values
  - System calls return specific negative values to indicate an error
    - Most system calls return -errno



Syscall Parameters

- For simplicity and security, syscall parameters are passed in registers across the kernel boundary
- Also consistency with interrupts and exception handlers
- The first parameter is always the syscall #
  - eax on Intel
- Linux allows up to six additional parameters
  - ebx, ecx, edx, esi, edi, ebp on Intel
- System calls that require more parameters package the remaining params in a struct and pass a pointer to that struct as the sixth parameter
- Note that syscalls are C functions with asmlinkage because the syscall "interrupt handler" is written in assembly; it sets up the stack frame, copying parameters from registers to the stack and then calls the appropriate sys_ function

Tracing System Calls

- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- The ptrace() system call is used (same call used by debuggers for single-stepping applications)
- Use the "strace" command (man strace for info)
- You can trace library calls using the "ltrace" command

Blocking System Calls

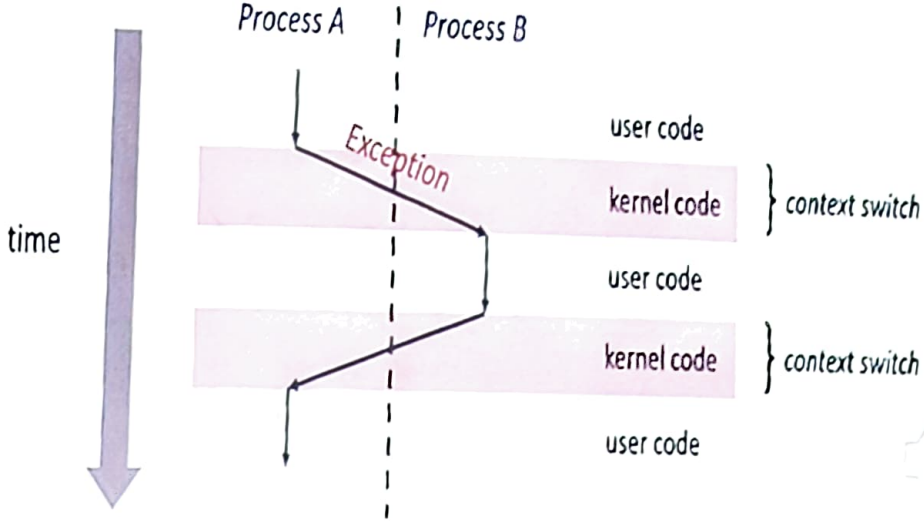
- System calls often block in the kernel (e.g. waiting for IO completion)
- When a syscall blocks, the scheduler is called and selects another process to run
- Linux distinguishes "slow" and "fast" syscalls:
  - Slow: may block indefinitely (e.g. network read)
  - Fast: should eventually return (e.g. disk read)
- Slow syscalls can be "interrupted" by the delivery of a signal (e.g. Control-C)
- Intel has a hardware optimization (sysenter) that provides an optimized system call invocation

Measuring Context Switch

- First Graded Assignment
  - Add a system call to display helloWorld
    - Your program should take a parameter -☺

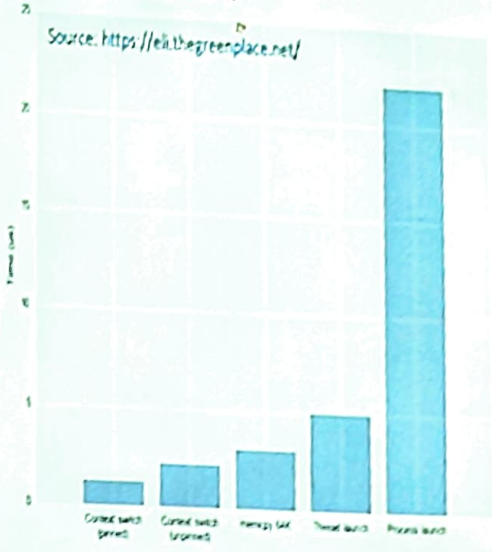
Measuring Context Switch

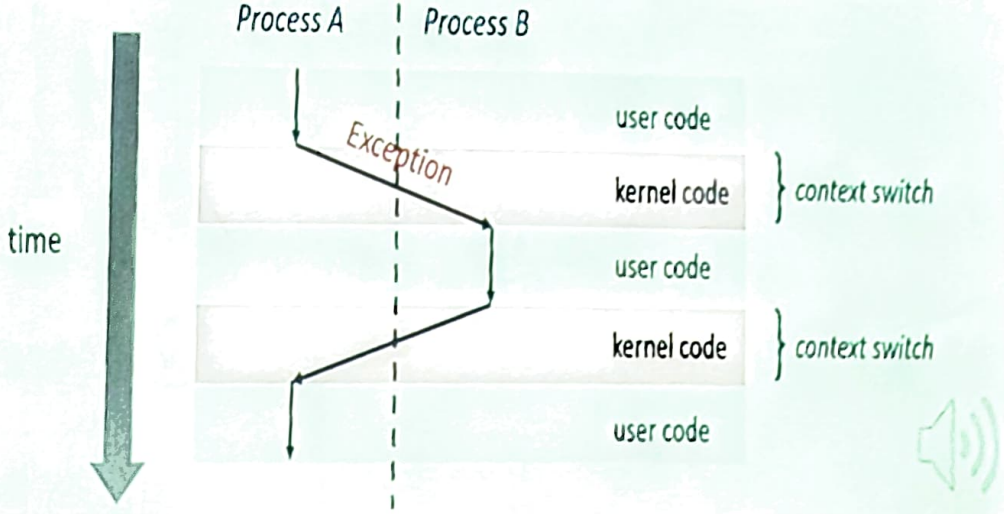
- First Graded Assignment
  - Add a system call to display helloWorld
    - Your program should take a parameter -☺
  - Add a system call to measure context Switch time
    - Need to understand how to measure time



Measuring Context Switch

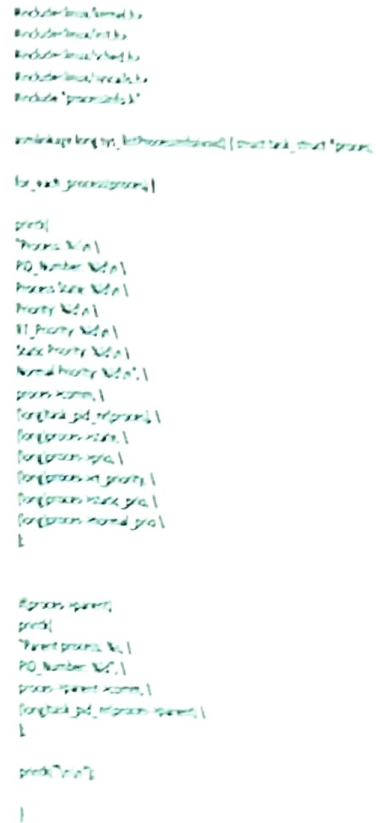
- First Graded Assignment
  - Add a system call to display helloWorld
    - Your program should take a parameter -☺
  - Add a system call to measure context Switch time
    - Need to understand how to measure time





Steps for Introducing a System Call

- Ensure that you are in the linux-xx.xx.y directory
- Create your own directory
  - **mkdir info**
    cd info/
  - **Create processInfo.h**
    - asmlinkage long sys_listProcessInfo(void);
  - Create processInfo.c
  - Write Makefile and change top level Makefile
    - obj-y:=listProcessInfo.o
    - core -y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ info/
  - Alter the /arch/x86/entry/syscalls/syscall_64.tbl
    - 548 common processInfo sys_processInfo
  - Alter /include/linux/syscalls.h
    - Add asmlinkage long sys_hello(void)
  - Compile
    - sudo make && sudo make modules_install && sudo make install
  - Test



directory

```
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/sched.h>
#include<linux/syscalls.h>
#include "processInfo.h"

asmlinkage long sys_listProcessInfo(void) { struct task_struct *proces;

for_each_process(proces) {

printk(
"Process: %s\n \
PID_Number: %ld\n \
Process State: %ld\n \
Priority: %ld\n \
RT_Priority: %ld\n \
Static Priority: %ld\n \
Normal Priority: %ld\n", \
proces->comm, \
(long)task_pid_nr(proces), \
(long)proces->state, \
```

e

Process

- Definition: **execution environment with restricted rights**
  - Address Space with One or More Threads
  - Owns memory (mapped pages)
  - Owns file descriptors, file system context, ...
  - Encapsulates one or more threads sharing process resources
- Application program executes as a process
  - Write Makefile and change top level Makefile
  - Complex applications can fork/exec child processes [later]
- Why processes?
  - Alter the /arch/x86/entry/syscalls/syscall_64.tbl
    - 548 common processInfo sys_processInfo
  - Alter /include/linux/syscalls.h
    - Add asmlinkage long sys_hello(void)
  - Compile
    - sudo make && sudo make modules_install && sudo make install
  - Test