Systems Programming

Kernel Development

H. Turgut Uyar Şima Uyar

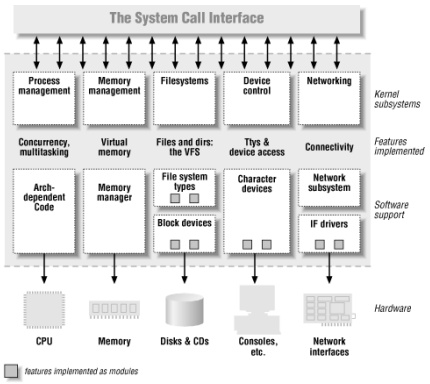
2001-2009

Topics

Kernel

- provides programs with a consistent view of the hardware
- protection against unauthorized access to resources
  - different levels of protection
  - kernel executes in supervisor-mode (kernel-space), applications execute in user-mode (user-space)
- switching to kernel-space:
  - system calls: synchronous, in the process context
  - interrupts: asynchronous

Kernel Subsystems



Kernel Subsystems

- process management
  - creating and destroying processes
  - communication between processes
  - scheduling
- memory management
  - virtual address space for each process
- filesystems
  - structured filesystem on top of unstructured hardware
- device control
- networking
  - delivering data packets across program and network interfaces
  - routing and address resolution

Kernel Architecture

- *monolithic*
  - all functionality in one big chunk of code
- *microkernel*
  - kernel organized as layers
  - most of the functionality in user-space
  - too much communication overhead

Getting Ready

- `sudo apt-get update`
- `sudo apt-get install linux-source`
- `tar xjvf /usr/src/linux-source-2.6.28.tar.bz2`
- `sudo apt-get install kernel-package`
  utilities for compiling the kernel
- `sudo apt-get install fakeroot`
  allows compiling the kernel without becoming root
- `sudo apt-get install libncurses5-dev`
  used by script which configures the kernel

Configuring

- copy `config.txt` file to `linux-source-2.6.28`
- rename as `.config`
- `make oldconfig`
  modifies the config file based on changes between kernel versions
- `make menuconfig`
  for configuring the kernel
  - as exercise: remove the floppy disk support from the kernel
  - [*M*]: module, [∗]: included in kernel, [ ]: disabled

Compiling

- `make-kpkg clean`
  cleans up all from previous kernel compiles
- ```
  fakeroot make-kpkg -initrd
           -append-to-version=-custom
           kernel_image kernel_headers
  ```
- two files in parent directory:
  linux-image-2.6.28.10-custom_....deb
  linux-headers-2.6.28.10-custom_....deb

Installing

- sudo dpkg -i linux-image-2.6.28.10-custom_....deb
- sudo dpkg -i linux-headers-2.6.28.10-custom_....deb
- uninstalling:
  sudo dpkg -r linux-image-2.6.28.10-custom
  sudo dpkg -r linux-headers-2.6.28.10-custom

System Calls

- used by application programs to request service from operating system
- execute in kernel mode
- each has a unique number

System Call Interface

- wrapper functions usually have same name
- system call interface may change when sizes of structures or arguments change
- user programs unaware of change because of wrapper functions
  ```

System Calls in Linux

list of system calls

- man 2 syscalls
- man 2 unimplemented

invoking a system call

- **int** syscall (**int** no, ...);

Preliminaries

- new system call becomes part of kernel
- need to re-compile the kernel
- step-by-step procedure may be different for different distributions

Steps

- define and write new system call
- modify system header files
- modify system call table
- compile kernel
- reboot to new kernel
- test new system call

System Call Example

Example

- a new system call to add two parameters

System Call Table

- append entry to file:
  `arch/x86/kernel/syscall_table_32.S`

Example

.**long** sys_mycall

System Calls Header File

- add system call prototype to file:
  `include/linux/syscalls.h`

Example

asmlinkage **int** sys_mycall(**int** i, **int** j);

Standard Header

- append entry for system call to file: `arch/x86/include/asm/unistd_32.h`

Example

#define __NR_mycall 333

Implement System Call

Example (mycall/mycall.c)

```
#include <linux/kernel.h>

asmlinkage int sys_mycall(int i, int j)
{
    return i + j;
}
```


Modify Makefile

- add your directory to Makefile
- add a Makefile to your directory

Example (Makefile)

core−y += .... mycall/

Example (mycall/Makefile)

obj−y := mycall.o

Sample Test Program

Example

```
#include <stdio.h>
#include <sys/syscall.h>

#define __NR_mycall 333

int main(void)
{
    int x1=10, x2=20, y;

    y = syscall(__NR_mycall, x1, x2);
    printf("%d\n", y);
    return 0;
}
```


Another Example

Example

- a new system call to obtain the time in seconds which has passed since 1970

System Call Table

- append entry to file: `arch/x86/kernel/syscall_table_32.S`

Example

.long sys_ptime

System Calls Header File

- add system call prototype to file: `include/linux/syscalls.h`

Example

```
asmlinkage int sys_ptime(struct timeval *);
```


Standard Header

- append entry for system call to file: `arch/x86/include/asm/unistd_32.h`

Example

```
#define __NR_ptime 334
```


struct timeval

Example

```
struct timeval
{
    long tv_sec;          /* seconds */
    long tv_usec;         /* microseconds */
};
```


Implement System Call

Example (ptime/ptime.c)

```
#include <linux/kernel.h>

asmlinkage int sys_ptime(struct timeval *tm)
{
  copy_to_user(tm, &xtime,
               sizeof(struct timeval));
  return 0;
}
```


Modify Makefile

- add your directory to Makefile
- add a Makefile to your directory

Example (Makefile)

```
core-y += .... ptime/
```

Example (ptime/Makefile)

```
obj-y := ptime.o
```


Sample Test Program

Example

```
#include <stdio.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <sys/time.h>

#define __NR_ptime 334
```

Sample Test Program

Example

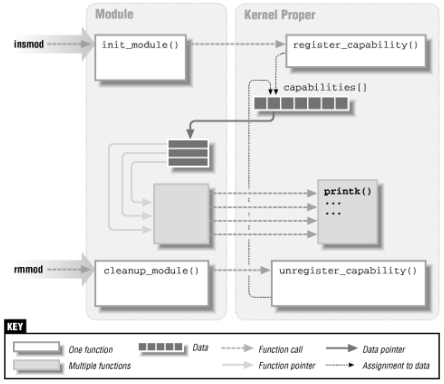
```
int main(void)
{
    struct timeval utime;
    int result;

    res = syscall(__NR_ptime, &utime);
    printf("%d\n", (int) utime.tv_sec);
    sleep(2);
    res = syscall(__NR_ptime, &utime);
    printf("%d\n", (int) utime.tv_sec);
    return 0;
}
```


Modular Kernel

- monolithic architecture
- kernel extended at runtime
- easier and safer development

Module Registry



Modules vs Applications

- an application performs a task,
  a module registers itself in order to serve future requests
- an application runs in user-space,
  a module runs in kernel-space
  - an application can call external functions,
    a module can only call kernel functions

Module Example

Example (Hello world)

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual␣BSD/GPL");

static int hello_init(void) { ... }

static void hello_exit() { ... }

module_init(hello_init);
module_exit(hello_exit);
```


Module Example

Example (Hello world)

```
static int hello_init(void)
{
  printk(KERN_ALERT "Hello,␣world!\n");
  return 0;
}

static void hello_exit()
{
  printk(KERN_ALERT "Goodbye,␣cruel␣world!\n");
}
```

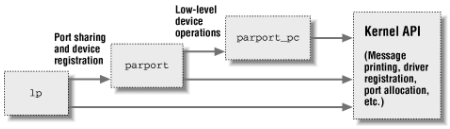
Kernel Symbol Table

- kernel symbol table contains addresses of global kernel symbols
- on loading:
  - link unresolved symbols in module to the kernel symbol table
  - exported module symbols become part of the kernel symbol table

Module Stacking

- modules can use symbols exported by other modules

Example



Reading Material

- Corbet
  - Chapter 2: Building and Running Modules