Systems Programming

Kernel Development

H. Turgut Uyar   Şima Uyar

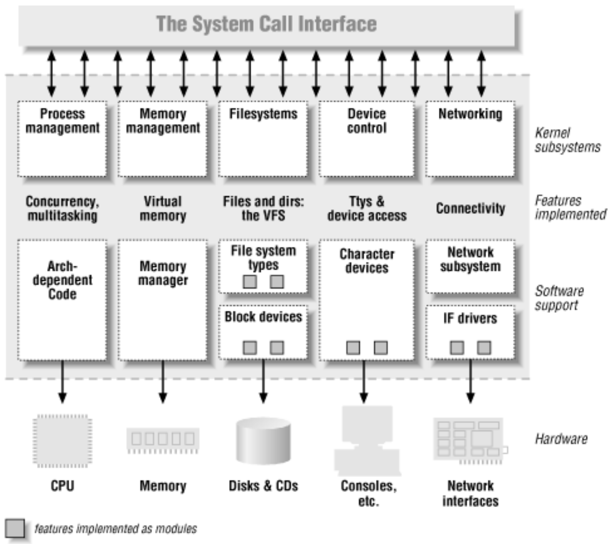
2001-2014

Topics

Kernel

- provides programs with a consistent view of the hardware
- protects against unauthorized access to resources
- kernel runs in supervisor mode (kernel space), applications run 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: organized as layers
  - most functionality in user space
  - too much communication overhead

Kernel Development

- recompile the kernel
- reboot the computer
- test the new kernel
- reboot to the original kernel

- very slow development cycle!
- no external libraries

Example: Adding a System Call

- add an entry to the system call table:
  system call number, name, function to invoke, ...
- add prototype to the system calls header file
- implement system call

Example: Adding a System Call

- new system call: add two integers
- add an entry to the system call table

arch/x86/kernel/syscall_table_32.S

```
.long sys_mycall
```

- append entry for system call

arch/x86/include/asm/unistd_32.h

```
#define __NR_mycall 333
```


Example: Adding a System Call

- add prototype to the system calls header file

include/linux/syscalls.h

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

- implement system call

mycall.c

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


Example: Test Program

```
#define __NR_mycall 333

int main(int argc, char **argv)
{
    int x1 = 10, x2 = 20, y;

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


Data Transfer

- special functions for transferring data between kernel space and user space
- kernel → user:
  `copy_to_user(user_buf, kernel_buf, length)`
- user → kernel:
  `copy_from_user(kernel_buf, user_buf, length)`

Example: Data Transfer

- new system call: get the time passed since 1970
- kernel structure for representing time

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

- global variable that keeps the current time

```
struct timeval xtime;
```


Example: Data Transfer

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

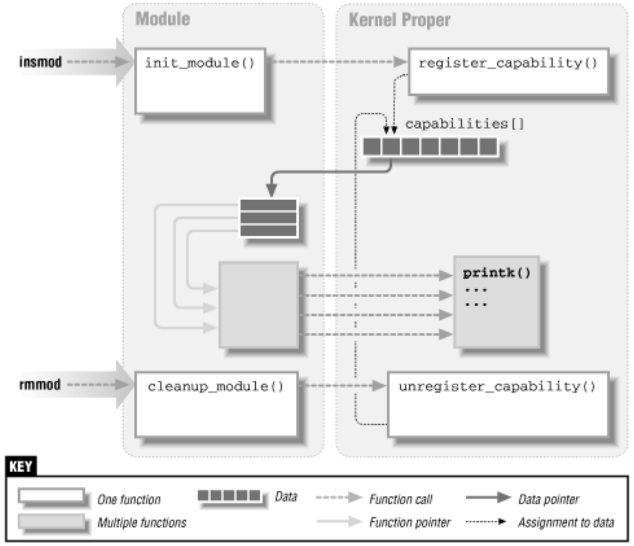

Example: Test Program

```
#define __NR_ptime 334

int main(int argc, char **argv)
{
    struct timeval utime;
    int res;

    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
- modules added or removed at runtime
- no need to reboot: faster development cycle

Module Registry



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);
```


Example: Hello, world!

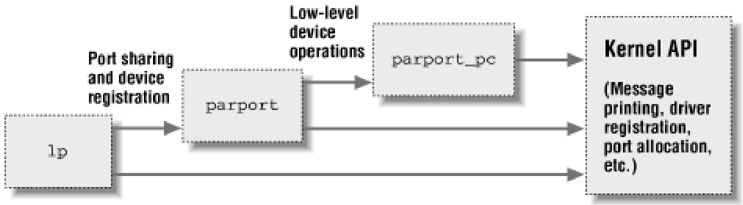
```
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world!\n");
    return 0;
}

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


Kernel Symbol Table

- kernel symbol table contains addresses of global symbols
- when loading a module:
- unresolved symbols are linked to the kernel symbol table
- exported symbols become part of the kernel symbol table

Module Stacking

- modules can use symbols exported by other modules



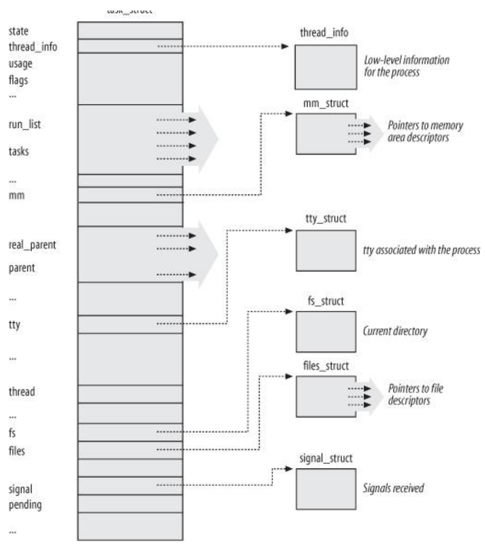
Reading Material

- Corbet-Rubini-Hartman, 3/e
  - Chapter 2: Building and Running Modules

Process Descriptor

- a process descriptor for each process: `struct task_struct`
- process state
- process identification (pid, uid, euid, ...)

Process Descriptor



Process List

- doubly linked list of all process descriptors
- `tasks` field of `task_struct`
- `current` macro gives the process descriptor of the running process: e.g. `current->pid`

Process 0

- a.k.a. the *idle process* or the *swapper*
- the first entry in the process list
- created during the initialization stage of the kernel
- the only process created without using the *fork* system call
- the ancestor of all processes

Process 0

- uses a statically allocated data structure
  - process descriptor stored in the `init_task` variable
  - initialized by the `INIT_TASK` macro
- executes the `start_kernel()` function
  - initializes all data structures needed by kernel
  - enables interrupts
  - creates process 1 (commonly known as the *init process*)
- executes the `cpu_idle()` function

Creating Processes

- *fork* is implemented as the `clone` system call
- `do_fork()` function handles the `clone` system call:

- allocates a pid for the child process
- uses `copy_process()` to set up the process descriptor and other kernel data structures for new process
  - uses `dup_task_struct()` to allocate a new process descriptor and to copy parent process' process descriptor info
- adjusts some parameters of parent and child processes
- returns pid of child process

Destroying Processes

- through the _exit() system call
- uses the do_exit() function

Synchronization

- critical sections and race conditions also exist for kernel code
- synchronization is needed

- several kernel level synchronization primitives
- primitive must be chosen based on requirements of operation

Synchronization Primitives

- atomic read-modify-write operations
- memory barriers (to avoid instruction reordering)
- spin locks (locks with busy waiting)
- kernel semaphores (lock with blocking wait)
- interrupt disabling (local CPU)

Atomic Operations

- instructions that execute atomically
- no interrupts

- to implement counters
- to atomically perform an operation and test results: e.g. atomic_dec_and_test

```
typedef struct {
    volatile int counter;
} atomic_t;
```

Memory Barriers

- kernel may reorder assembly instructions for optimization
- reordering must be avoided when synchronization is needed
- barrier ensures that the instructions before the primitive are completed before the instructions after the primitive

- read memory barrier: `rmb()`
- write memory barrier: `wmb()`
- memory barrier: `barrier()` – same as `wmb()`

Spin Locks

- for locking access to shared data (critical sections)
- for multiprocessor environments
- uses busy waiting
  - kernel resources usually locked for very short periods
  - more time consuming to release and reacquire cpu

- represented by a `spinlock_t` structure
- macros used for working with spin locks
- read and write spin locks to increase concurrency: `rwlock_t` structure

Semaphones

- sleeping locks
- suited for locks that are held for a long time
- not optimal for locks that are held for short periods

- kernel preemption not disabled, i.e. no adverse effects on scheduling latency
- allows arbitrary number of simultaneous lock holders: counting semaphores

- two atomic operations: `P()` - `V()`
  `down()` - `up()`

Scheduling

- divide the finite resource of processor time between the runnable processes on the system

- conflicting goals:
  - fast process response time (low latency)
  - maximal system utilization (high throughput)

- processor bound processes - I/O bound processes
  - Linux favors I/O bound processes, i.e. optimizes for low latency

O(1) Scheduler

- constant-time algorithm for timeslice calculation and per processor runqueues
- scalable
- ideal for large server workloads
- problems for interactive processes

CFS Scheduler

- Completely Fair Scheduler
- aims at improving scheduling for interactive processes

Linux Scheduler

- different algorithms to schedule different types of processes
- scheduler classes with priorities
- iterate over each scheduler class in order of priority
- CFS for normal processes
- two policies for real time processes:
  - `SCHED_FIFO`
  - `SCHED_RR`

CFS

- assign processes a *proportion* of processor
- nice value (priority) acts as weight to determine proportion of processor time
- preemptive (based on proportions of processor time consumed)

CFS

- *timeslice* proportional to process' weight
  over sum of weights of all runnable processes
- targeted latency
- minimum granularity

CFS Implementation

- for process accounting:
  `struct sched_entity`
- member of `struct task_struct`
- *virtual runtime* (`vruntime`):
  actual runtime (in ns) of a process
  normalized by the number of runnable processes
- in a perfectly multitasking system all processes should have
  the same virtual runtime
- updated periodically by the system timer
  and also whenever a process becomes runnable or is blocked

CFS Implementation

- the runnable process with the smallest `vruntime`
  is selected to run
- red-black tree to manage list of runnable processes:
  search in $O(log\ n)$
  - leftmost node has lowest `vruntime`
  - leftmost node is cached

Reading Material

- Linux Kernel Development, 3rd Edition
  - Author: Robert Love
  - Publisher: Addison-Wesley Professional
  - Year: 2010
  - Chapters: 3, 4, 5, 9 and 10
  - accessible on Safari e-books through the ITU Library