Operating Systems Lab '2016

Jürgen Schönwälder

September 22, 2016

http://cnds.eecs.jacobs-university.de/courses/osl-2016/

Part: Preface

Course Content

- Essential development tools
- Shells
- Linking and libraries
- Concurrent programming
- Kernel modules
- Filesystems
- Device driver
- Network filtering
- I/O programming

Course Objectives

- Gain practical experience with systems programming above and below the system call interface of operating systems
- Learn how to write concurrent programs
- Gain understanding how kernel programming differs from normal application development

Grading Scheme

- Homework assignments (100%)
- Start work on the assignment during the lab
- Complete assignment at home and submit solution via jgrader
- Defend your homework in an interview (random selection)

$\Rightarrow$ Consult the course web page for submission instructions and grading details.

Expectations

- Knowledge of C is essential (C++ does not help you here)
- Basic familiarity with Unix command line tools is required
- Programming and debugging via a terminal session will be needed

Part: Introduction

2 Terminology

Operating System

- General term for the collection of software that manages the resources of a computer including base software tools such as the shell (a command line interpreter), file utilities, and a graphical user interface.
  - Example: Debian GNU/Linux
- Central software that manages the computer's resources (processor, memory, storage devices, I/O devices).
  - Example: Linux kernel
- The family of Unix systems provides some well-defined services to (user space) programs.

System Calls vs. Library Calls

- An Operating System provides service points through which programs request services from the kernel.
- The so called system calls provide a well-defined and limited number of entry-points directly into the operating system kernel.
- On Unix systems, for each system call there is a function of the same name in the C library; user space programs usually call these library functions to invoke system calls.
- Note that not all library functions are wrappers for system calls.

System Calls vs. Library Calls

- Which of the following are system calls?
  - `printf()`
  - `write()`
  - `strcpy()`
  - `time()`

- From a system programmer's point of view, there is a big distinction between system calls and library calls
- From an application programmer's perspective, the distinction is often not important
- Knowing the system call and library interfaces well is crucial for writing efficient code

Program

- Exists as source code (written using an editor)
- Translated into native machine code (ignoring non-systems programming languages for the moment)
- Stored in a certain format in file of the a filesystem
- Loaded into memory and executed when requested

Process

- A program being executed is called a process
- A process has a unique identifier (of type `pid_t`)
- A process allocates and 'owns' various resources
- Memory is divided into segments:
  - text: machine instructions of the program
  - data: static variables
  - heap: dynamically allocated data structures
  - stack: automatically allocated local variables, management of function calls

Thread

- Threads are individual control flows, typically within a process (or within a kernel)
- Multiple threads share the same address space and other resources
  - Fast communication between threads
  - Fast context switching between threads
  - Often used for very scalable server programs
  - Multiple CPUs can be used by a single process
  - Threads require synchronization (see later)
- Linux provides thread support in the kernel
- Some kernel functions are running as separate thread inside the kernel

Shell

- The Unix shell is a command interpreter that provides the traditional Unix-like command-line user interface
- The Bourne Again Shell (`bash`), an open-source re-implementation of the Bourne Shell (`sh`), is a popular shell on Linux systems
- Unless noted differently, examples assume `bash` as a shell
- Shell scripts are sequences of shell commands stored in a file
- For portability reasons, it is often recommended to stick to the commands supported by the original Bourne Shell

Part: Tools

3. bash
4. strace and ltrace
5. ps and top
6. lsof and watch

bash: redirects

- Every process has some standard I/O channels:
  - standard input (readable)
  - standard output (writeable)
  - standard error (writeable)
- Unix processes refer to their I/O channels via file descriptors (small non-negative integers)
- The standard input has file descriptor 0, the standard output has file descriptor 1, the standard error has file descriptor 2
- For an interactive shell, the standard input, output, and error file descriptors initially refer to a terminal device
- The file descriptors can be changed to refer to other I/O channels (e.g., files) via redirection

bash: redirects

Symbol	Function
>	Redirect output to a file
>>	Redirect output by appending to a file
<	Redirect input from a file
n>	Redirect file descriptor n to a file
n>>	Redirect file descriptor n by appending to a file
n<	Redirect file descriptor n from a file

bash: redirects

```
$ echo "a" > /tmp/file.txt
$ echo "b" 1> /tmp/file.txt
$ echo "c" >> /tmp/file.txt
$ echo "d" 1>> /tmp/file.txt
$ cat < /tmp/file.txt
$ cat 0< /tmp/file.txt
```

bash: pipelines

- Pipelines connect the standard output I/O channel of a process to the standard input I/O channel of a subsequent process
- The `tee` utility program copies standard input to standard output and copies data to zero or more files
- The `less` utility program allows users to page through text one screenful at a time (less is more)

```
$ cat < /tmp/file.txt | less
$ cat < /tmp/file | tee /tmp/copy | less
```

bash: parameters and variables

```
$ a=foo
$ echo $a
foo
$ echo $PATH
/home/schoenw/bin:/usr/local/bin:/usr/bin:/bin
$ echo $HOME
/home/schoenw
```

- A parameter is an entity that stores values.
- A variable is a parameter denoted by a name.
- Variables can be local or be exported to the environment.

bash: comments and quoting

```
$ # this is a comment
$ echo foo    bar
foo bar
$ echo "foo    bar"
foo    bar
$ echo 'foo    bar'
foo    bar
$ echo 'foo    "bar"'
foo    "bar"
$ echo "foo    'bar'"
foo    'bar'
$ echo "$SHELL" '$SHELL'
/bin/bash $SHELL
$ echo $?
0
```

- Characters enclosed in single quotes are treated as a word
- Characters in double quotes are teeated as a word but substitutions still apply

bash: pathname expansions

```
$ cd ~
$ pwd
/home/schoenw
$ HOME=/tmp
$ cd ~
$ pwd
/tmp
$ ls *.[ch]
```

- The character ~ expands to the path of the home directory
- Glob-style pattern matching is used to expand file names
- There is a limit on the number of arguments that can be passed to a command (see `xargs` if you hit the limit)

bash: command substitution

```
$ d=‘date‘
$ echo $d
Wed Aug 31 16:17:53 CEST 2016
$ d=$(date)
$ echo $d
Wed Aug 31 16:21:17 CEST 2016
$ echo=echo
$ $echo $($echo $($echo $echo))
echo
$ $echo ‘$echo \‘$echo $echo\‘‘
echo
```

- Execute a program and substitute the result (standard output)
- Backquotes indicate command substitution, but difficult to nest
- The simpler syntax is the `$(command)` notation

job control

```
$ echo hi &
[1] 8640
$ hi
$ sleep 5 &
[1] 8664
$ sleep 5 &
[2] 8667
$ jobs
[1]-  Running                 sleep 5 &
[2]+  Running                 sleep 5 &
$ fg %1
$
```

- Commands can execute as jobs in the background
- Jobs are identified by a number, `%n` refers to job `n`
- Background jobs can be brought back into the foreground

strace — trace system calls

```
$ strace /bin/true
execve("/bin/true", ["/bin/true"], [/* 21 vars */]) = 0
brk(0)                                  = 0xef7000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe138660000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=34548, ...}) = 0
mmap(NULL, 34548, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe138657000
close(3)                                = 0
```

- Traces system calls
- Very useful to quickly figure out what a program might be doing
- Provides summary and timing statistics

ltrace — trace library calls

```
$ ltrace /bin/true
__libc_start_main(0x401390, 1, 0x7ffcbaa25d78, 0x403e20 <unfinished ...>
exit(0 <no return ...>
+++ exited (status 0) +++
```

- Traces library calls (and optionally also system calls)
- Very useful to quickly figure out what a program might be doing
- Allows to optain some timing information
  (always measure, never trust your intuition)
- It is possible to trace already running programs
- Does not work for statically linked libraries

ps — snapshot of the current processes

```
$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
schoenw   7263  0.0  0.2  20564  2436 pts/0    R+   15:26   0:00 ps ux
schoenw  28572  0.0  0.3  35632  3680 ?        Ss   10:12   0:00 /lib/systemd/sy
schoenw  28573  0.0  0.1  58368  1572 ?        S    10:12   0:00 (sd-pam)
schoenw  28575  0.0  0.4 101764  4140 ?        S    10:12   0:01 sshd: schoenw@p
schoenw  28576  0.0  0.5  25668  6012 pts/0    Ss   10:12   0:00 -bash
schoenw  28708  0.0  1.5  49660 15912 pts/0    T    10:12   0:06 emacs ../mps.c
```

- Provides a snapshot of the processes running on a system
- May include information about kernel threads
- Can show information about the threads of multi-threaded programs
- Provides basic statistics about resource usage

top — interactive process viewer

```
top - 14:50:46 up 468 days, 15:16,  2 users,  load average: 0.00, 0.01, 0.05
Tasks:  78 total,   1 running,  76 sleeping,   1 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1024468 total,   903192 used,   121276 free,   168280 buffers
KiB Swap:  2045196 total,    36412 used,  2008784 free.   572224 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 3623 schoenw   20   0   25660   5892   3156 S  0.0  0.6   0:00.08 bash
 3632 schoenw   20   0   27080   2808   2404 R  0.0  0.3   0:00.01 top
28572 schoenw   20   0   35632   3680   3184 S  0.0  0.4   0:00.00 systemd
28573 schoenw   20   0   58368   1572      0 S  0.0  0.2   0:00.00 (sd-pam)
28575 schoenw   20   0  101764   4140   3096 S  0.0  0.4   0:01.83 sshd
28576 schoenw   20   0   25668   6012   3172 S  0.0  0.6   0:00.48 bash
28708 schoenw   20   0   49660  15912   7268 T  0.0  1.6   0:06.95 emacs
```

- Displays a sorted list of processes, updated periodically
- The sorting order and the properties shown can be configured
- Several different versions out there with slightly different features
- Essential tool to quickly can an idea "why the system is slow"

lsof — list open files

```
$ lsof /dev/tty
COMMAND   PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
emacs   28708 schoenw    3u   CHR    5,0      0t0 5600 /dev/tty
```

- Shows the list of open files for a process or all processes
- Shows a list of processes that have opened a certain file
- Provides information about open network connections
- Written to run on many platforms (not only Linux)

watch — execute a program periodically

```
$ watch -d date
Every 2.0s: date                                        Wed Aug 31 14:57:45 2016

Wed Aug 31 14:57:45 CEST 2016
```

- Executes a command periodically
- Turns simple commands into something you can “watch”
- Can highlight differences between the last and the current command output

Part: Linking

C Compilation Process

```
C preprocessor -> expanded C code     (gcc -E hello.c)
      v                v
  C compiler   -> assembler code      (gcc -S hello.c)
      v                v
   assembler   -> object code         (gcc -c hello.c)
      v                v
    linker     -> executable          (gcc hello.c)
```

- Compiling C source code is traditionally a four-stage process.
- Modern compilers often integrate stages for efficiency reasons.

Reasons for using a Linker

- Modularity
  - Programs can be we written as a collection of small files
  - Building a collection of easily reusable functions
- Efficiency
  - Separate compilation of a subset of small files saves time on large projects
  - Smaller executables by linking only functions that are actually used

What does a Linker do?

- Symbol resolution
  - Programs define and reference symbols (variables or functions)
  - Symbol definitions and references are stored in object files
  - Linker associates each symbol reference with exactly one symbol definition
- Relocation
  - Merge separate code and data sections into combined sections
  - Relocate symbols from relative locations to their final absolute locations
  - Update all references to these symbols to reflect their new positions

Object Code Files Types

- Relocatable object files (.o files)
  - Contains code and data in a form that can be combined with other relocatable object files
- Executable object files
  - Contains code and data in a form that can be loaded directly into memory
- Shared object files (.so files)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically at either load time or runtime

Executable and Linkable Format

- Standard unified binary format for all object files
- ELF header provides basic information (word size, endianess, machine architecture, ...)
- Program header table describss zero or more segments used at runtime
- Section header table provides information about zero or more sections
- Separate sections for `.text`, `.rodata`, `.data`, `.bss`, `.symtab`, `.rel.text`, `.rel.data`, `.debug` and many more

- The `readelf` tool can be used to read ELF format
- The tool `objdump` can process ELF formatted object files

Linker Symbols

- Global symbols
  - Symbols defined by a module that can be referenced by other modules
- External symbols
  - Global symbols that are referenced by a module but defined by some other module
- Local symbols
  - Symbols that are defined and referenced exclusively by a single module
- Tools:
  - The traditional tool `nm` displays the (symbol table) of object files in a traditional format
  - The newer tool `objdump -t` does the same for ELF object files

Strong and Weak Symbols and Linker Rules

- Strong Symbols
  - Functions and initialized global variables
- Weak Symbols
  - Uninitialized global variables
- Linker Rules:
  - Rule 1: Multiple strong symbols are not allowed
  - Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol
  - Rule 3: If there are multiple weak symbols, pick an arbitrary one

Linker Puzzles

- Link time error due to two definitions of `p1`:

```
a.c: int x; p1() {}
b.c:        p1() {}
```

- Reference to the same uninitialized variable `x`:

```
a.c: int x; p1() {}
b.c: int x; p2() {}
```

- Reference to the same initialized variable `x`:

```
a.c: int x=1; p1() {}
b.c: int x;   p2() {}
```

- Writes to the double `x` likely overwrite `y`:

```
a.c: int x; int y; p1() {}
b.c: double x;     p2() {}
```

Static Libraries

- Collect related relocatable object files into a single file with an index (called an archive)
- Enhance linker so that it tries to resolve external references by looking for symbols in one more more archives
- If an archive member file resolves a reference, link the archive member file into the executable (which may produce additional references)
- Archive format allows incremental updates

- Example:

```
ar -rs libfoo.a foo.o bar.o
```

Shared Libraries

- Static linking duplicates library code by copying it into executables
- Bug fixes in libraries require to re-link all executables
- Solution: Delay the linking until program start and then link against the most recent matching versions of the required libraries
- At traditional link time, an executable file is prepared for dynamic linking (i.e., information is stored which shared libraries are needed) while the final linking takes place when an executable is loaded into memory
- First nice side effect: Library code can be stored in memory shared by multiple processes
- Second nice side effect: Programs can load additional code dynamically while the program is running
- Caveat: Loading untrusted libraries can lead to real surprises

Interpositioning

- Intercept library calls for fun and profit
- Examples:
  - Debugging: tracing memory allocations / leaks
  - Profiling: study typical function arguments
  - Sandboxing: emulate a restricted view on a filesystem
  - Hardening: simulate failures to test program robustness
  - Privacy: add encryption into I/O calls
  - Hacking: give a program an illusion to run in a different context
  - Spying: oops

Compile-time Interpositioning

- Change symbols are compile to so that library calls can be intercepted
- Typically done in C using `#define` pre-processor substitutions, sometimes contained in special header files
- This technique is restricted to situations where source code is available
- Example:

```
#define malloc(size) dbg_malloc(size, __FILE__, __LINE__)
#define free(ptr) dbg_free(ptr, __FILE__, __LINE__)

void *dbg_malloc(size_t size, char *file, int line);
void dbg_free(void *ptr, char *file, int line);
```

Link-time Interpositioning

- Tell the linker to change the way symbols are matched
- The GNU linker supports the option `--wrap=symbol`, which causes references to `symbol` to be resolved to `__wrap_symbol` while the real symbol remains accessible as `__real_symbol`.
- The GNU compiler allows to pass linker options using the `-Wl` option.
- Example:

```
/* gcc -Wl,--wrap=malloc -Wl,--wrap=free */
void * __wrap_malloc (size_t c)
{
    printf("malloc called with %zu\n", c);
    return __real_malloc (c);
}
```

Load-time Interpositioning

- The dynamic linker can be used to pre-load shared libraries
- This may be controlled via setting the `LD_PRELOAD` environment variable
- Example:

  ```
  LD_PRELOAD=./libmymalloc.so vim
  ```

Load-time Interpositioning Example 1/2

```
/*
 * gcc -Wall -fPIC -DPIC -c datehack.c
 * ld -shared -o datehack.so datehack.o -ldl (Linux)
 * ld -dylib -o datehack.dylib datehack.o -ldl (MacOS)
 *
 * LD_PRELOAD=./datehack.so date (Linux)
 * DYLD_INSERT_LIBRARIES=./datehack.dylib date (MacOS)
 *
 * See fakeroot <http://freecode.com/projects/fakeroot> for a project
 * making use of LD_PRELOAD for good reasons.
 *
 * http://hackerboss.com/overriding-system-functions-for-fun-and-profit/
 */

#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <unistd.h>
#include <sys/types.h>
```

Load-time Interpositioning Example 2/2

```
struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
    time_t t = *timep - 60 * 60 * 24;
    return orig_localtime(&t);
}

void
_init(void)
{
    orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Part: Threads

Concurrency

- Increasing processor clock speed increases the heat produced
- Hence, it is not possible to clock processors arbitrarily fast
- To achieve further speed improvements, it is necessary to use multiple processors
- Concurrent programs that can take advantage of multiple processors
- Writing concurrent programs is difficult (synchronization and coordination problems)
- Even with concurrent programs and multiple processors, there are limits on the speed that can be achieved

Speedup and Efficiency and Amdahl's Law

Let T_n be the execution time of a task with n processors.

- The speedup S_n is defined as $S_n = T_1/T_n$.
- The efficiency E_n is defined as $E_n = S_n/n$.

Let p the portion of a program that can benefit from multiple processors and $1 - p$ the portion of a program that is strictly sequential.

- The expected speedup $\hat{S}$ for a task is given by:

$$\hat{S} = \frac{1}{(1-p) + \frac{p}{n}}$$

A consequence of Amdahl's law is that there is limit for the expected speedup, i.e., after reaching the certain point, adding more processors does not improve the speedup significantly anymore.

POSIX API (pthreads)

```
#include <pthread.h>

typedef ... pthread_t;
typedef ... pthread_attr_t;

int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void * (*start) (void *),
                   void *arg);
void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);
int pthread_join(pthread_t thread, void **retvalp);

int pthread_cleanup_push(void (*func)(void *), void *arg)
int pthread_cleanup_pop(int execute)
```

POSIX Mutex Locks

```
#include <pthread.h>

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abstime);
```

- Mutex locks are a simple mechanism to achieve mutual exclusion in critical sections

POSIX Condition Variables

```
#include <pthread.h>

typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *condattr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           struct timespec *abstime);
```

- Condition variables can be used to bind the entrance into a critical section protected by a mutex to a condition

POSIX Barriers

```
#include <pthread.h>

typedef ... pthread_barrier_t;
typedef ... pthread_barrierattr_t;

int pthread_barrier_init(pthread_barrier_t *barrier,
                         pthread_barrierattr_t *barrierattr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- Barriers block threads until the required number of threads have called `pthread_barrier_wait()`

POSIX Message Queues

```
#include <mqueue.h>

typedef ... mqd_t;

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);

int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- Message queues can be used to exchange messages between threads and processes running on the same system efficiently

POSIX Message Queues

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
                 size_t msg_len, unsigned int msg_prio,
                 const struct timespec *abs_timeout);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr,
                        size_t msg_len, unsigned int *msg_prio,
                        const struct timespec *abs_timeout);

int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

- Notifications about message arrivals can be delivered in different ways, e.g., as signals or in a thread-like fashion

POSIX Semaphores

```
#include <semaphore.h>

typedef ... sem_t;

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);

sem_t* sem_open(const char *name, int oflag);
sem_t* sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- Unnamed semaphores are created with (`sem_init()`)
- Named semaphores are created with (`sem_open()`)

Atomic Operations in Linux (2.6.x)

```
struct ... atomic_t;

int  atomic_read(atomic_t *v);
void atomic_set(atomic_t *v, int i);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);

int atomic_add_negative(int i, atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_inc_and_test(atomic_t *v)
int atomic_dec_and_test(atomic_t *v);
```

- The `atomic_t` is essentially 24 bit wide since some processors use the remaining 8 bits of a 32 bit word for locking purposes

Atomic Operations in Linux (2.6.x)

```
void set_bit(int nr, unsigned long *addr);
void clear_bit(int nr, unsigned long *addr);
void change_bit(int nr, unsigned long *addr);

int  test_and_set_bit(int nr, unsigned long *addr);
int  test_and_clear_bit(int nr, unsigned long *addr);
int  test_and_change_bit(int nr, unsigned long *addr);
int  test_bit(int nr, unsigned long *addr);
```

- The kernel provides similar bit operations that are not atomic (prefixed with two underscores)
- The bit operations are the only portable way to set bits
- On some processors, the non-atomic versions might be faster

Spin Locks in Linux (2.6.x)

```
typedef ... spinlock_t;

void spin_lock(spinlock_t *l);
void spin_unlock(spinlock_t *l);
void spin_unlock_wait(spinlock_t *l);
void spin_lock_init(spinlock_t *l);
int  spin_trylock(spinlock_t *l)
int  spin_is_locked(spinlock_t *l);

typedef ... rwlock_t;

void read_lock(rwlock_t *rw);
void read_unlock(rwlock_t *rw);
void write_lock(rwlock_t *rw);
void write_unlock(rwlock_t *rw);
void rwlock_init(rwlock_t *rw);
int  write_trylock(rwlock_t *rw);
int  rwlock_is_locked(rwlock_t *rw);
```

Semaphores in Linux (2.6.x)

```
struct ... semaphore;

void sema_init(struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);

void down(struct semaphore *sem);
int  down_interruptible(struct semaphore *sem);
int  down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);
```

- Linux kernel semaphores are counting semaphores
- `init_MUTEX(s)` equals `sema_init(s, 1)`
- `init_MUTEX_LOCKED(s)` equals `sema_init(s, 0)`