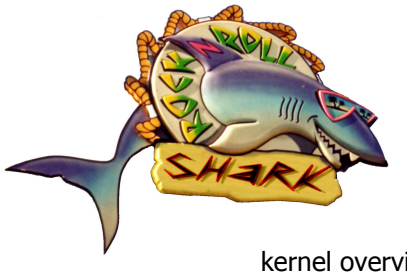
## *ReTiS Lab*

**Scuola Superiore S. Anna**

**Pisa**




kernel overview

Paolo Gai
Evidence Srl




## summary

- introduction
- walking through the architecture
- shark user interface
- libraries and drivers
- installing the kernel




## part I

introduction




## what is S.Ha.R.K.?

- S.Ha.R.K. is an open source real-time kernel mainly developed at Scuola Superiore S. Anna, Italy and at the Robotic Lab of the University of Pavia, Italy
- it supports:
  - modular interface for the specification of scheduling algorithms
  - device drivers for the most common hardware
  - advanced time handling




## objectives

- simplicity of the development
- flexibility in the modification of the scheduling policy
- predictability
- adherence to the POSIX standard

## the POSIX standard

- a standard for the programming interface of UNIX systems
  - standard C library
  - process and thread primitives, scheduling
  - file and I/O primitives
  - synchronization (semaphores, mutex, condition variables, message passing)
  - shared memory
  - signals and timers




## the POSIX standard (2)

- standards
  - 1003.1a the core
  - 1003.1b real-time extensions
  - 1003.1c thread extensions
  - others sporadic server, timers, ecc...
- real time profiles
  - 1003.13 subsets of the 1003.1 standard




## POSIX 1003.13 profiles

- PSE51 minimal realtime system profile
  - no file system
  - no memory protection
  - monoprocess multithread kernel
- PSE52 realtime controller system profile
  - PSE51 + file system + asynchronous I/O
- PSE53 dedicated realtime system profile
  - PSE51 + process support and memory protection
- PSE54 multi-purpose realtime system profile
  - PSE53 + file system + asynchronous I/O




## S.Ha.R.K. And POSIX

- implements 90% of POSIX PSE52
  - standard C library
  - file system
  - pthread library
  - not asynchronous I/O
  - not locale and setjmp support
- implemented through modules and by redefining the standard primitives




## S.Ha.R.K. and freedom

- S.Ha.R.K. is **free** software
  - it is distributed under the GPL license
- the word free stands for **freedom**
- **3** kinds of freedom
  - to distribute it
  - to read, to modify and to enhance it
  - to obtain the same kind of freedom everywhere




## supported platforms

- GNU gcc compiler
- host operating systems
  - MS-DOS / Windows (DJGPP)
  - Linux (GCC)
- multiboot image format (COFF/ELF)
- target configuration
  - MS-DOS with a DOS extender
  - GRUB

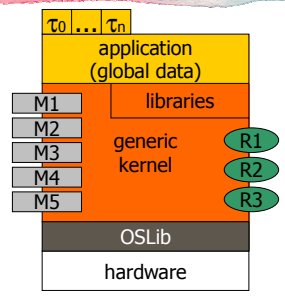
# part II

walking through the architecture




# architecture

application
libraries
resource modules
aperiodic servers modules
scheduling modules
generic kernel
OSLib
hardware







# tasks and instances

- a task is a concurrent activity
  - executed by the kernel
  - into a private stack
  - implemented by a C function
- a task execution can be divided in instances
  - instances can be periodic (e.g., one every second)
  - for example, a *clock* task has one instance/second
  - the `task_endcycle` function signal the end of a task instance




# a typical task body

```
void * body(void *arg)
{
  /* initialization part */
  ...
  for (;;) {
    /* the instance */
    ...
    task_endcycle();
  }
  return myvalue;
}
```




# scheduling algorithms: Round Robin

- this is the tradictional and simplest scheduling algorithm used in most OS
- every task:
  - is inserted into a ready queue
  - has a Quantum
  - consumes the Quantum when it executes
- when the Quantum finishes, the task is inserted at the end of the ready queue, and the next task in it is executed



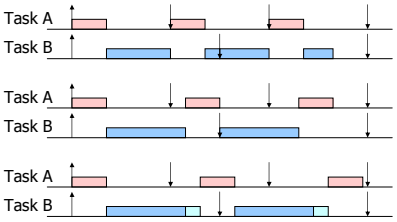

# RM and EDF

- Rate Monotonic (RM)
  - tasks are periodic or sporadic
  - the priority of a task is proportional to the period
- Earliest Deadline First (EDF)
  - tasks are periodic or sporadic
  - the priority of a tasks is the *deadline*, that is the absolute start time of the next instance
- the task with the lowest priority is always executed

## RM and EDF (2)

- an example:

- Rate Monotonic




- EDF...




- ...and there is still space!

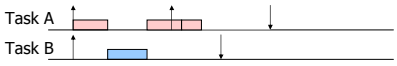






## Constant Bandwidth Server

- every task has a:
  - Period
  - Mean Execution Time (MET)
- tasks are scheduled using EDF
- when a task instance executes more than the MET, its deadline is postponed by one period

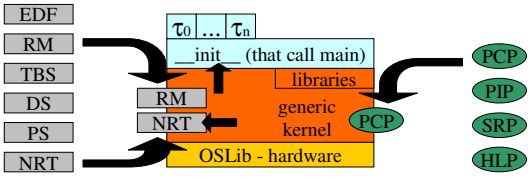






## system initialization

- the modules are registered during the system initialization
- the **main()** function is called into an NRT task

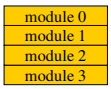






## modules organization

- **modules** are registered in the kernel in a given order
- module at level 0 has higher priority than the module at level 1

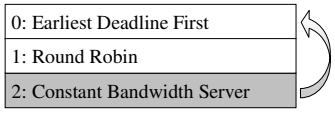



- a task that is assigned to a module is scheduled in background with respect to tasks of higher priority modules




## modules organization (2)

- suppose the following scenario:
  - periodic tasks / aperiodic tasks
  - Hard Real Time using EDF
  - Soft Real Time using CBS
  - Non Real Time using Round Robin







## the initialization file

```
TIME __kernel_register_levels__(void *arg)
{
  struct multiboot_info *mb =
       (struct multiboot_info *)arg;

  INTDRIVE_register_level(Q,T,FLAG);
  EDF_register_level(EDF_ENABLE_ALL);
  RR_register_level(RRTICK, RR_MAIN_YES, mb);
  CBS_register_level(CBS_ENABLE_ALL, 1);
  dummy_register_level();

  SEM_register_module();
  CABS_register_module();
  return TICK;
}
```

## the __init__ task

- an user application start with the `main()` function, usually understanding a set of default initialized services
  - keyboard, file system, semaphores
- the `__init__` task is created at startup into the Round Robin module
  - it initializes a set of devices
  - it calls the main function




## the __init__ task (2)

```
void *__init__(void *arg)
{
  struct multiboot_info *mb =
      (struct multiboot_info *)arg;

  device_drivers_init();

  set_shutdown_task();
  sys_atrunlevel(call_shutdown_task, NULL,
                 RUNLEVEL_SHUTDOWN);

  HARTPORT_init();
  call_main__(mb);
  return (void *)0;
}
```




## the main() function

- syntax (ANSI C)

  `int main(int argc, char **argv)`

- is called by the `__init__` task
- used to start an application
- when the `main()` ends, the system DOES NOT shutdown
- `main()` is a function like all the others




## the main() function (2)

- is usually used for:
  - create and activate the application tasks
  - init the devices not initialized into `__init__`
  - set the exception handlers
- the main() function:
  - may terminate or may check for the exit conditions
  - usually does not have an endless busy cycle (a busy cycle inhibits the JET of the dummy, useful for load control)




## a real example

- the example shows:
  - interaction between CBS and EDF
  - independence of the application from the scheduling policy (two initialization files)
  - use of the
    - graphic library
    - keyboard
    - exceptions
    - JET functions




## part III

S.Ha.R.K. user interface

## time handling

- two data structures to handle the time
  - `TIME` microseconds
  - `struct timespec` seconds+nano (POSIX)
- to get the current time since startup

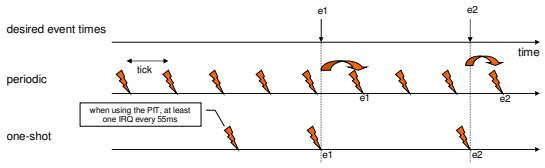
```
TIME sys_gettime(struct timespec *t)
```

- more precise timings can be obtained using the Pentium TSC
- there is no abstraction of tick




## time handling (2)

- the PC timer can be programmed to raise an interrupt ( ) every (max) 55 ms
- periodic, oneshot, oneshot with APIC
  - a timer interrupt raised every x us
  - a timer interrupt raised only when needed







## time handling (3)

- OSLib event abstraction
  - a function called at a specified time
  - the source can be a timer or an interrupt from an external interface
  - events runs at the highest interrupt priority, and cannot be preempted
  - events are used by the scheduling algorithms to implement asynchronous behavior like: deadline checks, periodic reactivations, capacity exhaustions




## system lifecycle

- the i386 starts in real mode
- call to `__kernel_register_levels__`
- the i386 goes in protected mode
- `RUNLEVEL_INIT`: call registered functions
- the first task is created, `__init__` is called
- the Application runs
- `RUNLEVEL_SHUTDOWN`: call registered functions
- user task killed
- `RUNLEVEL_BEFORE_EXIT`: call registered functions
- the i386 goes back in real mode
- `RUNLEVEL_AFTER_EXIT`: call registered functions
- back to DOS or reset!




## system shutdown

- when the Application finishes, S.Ha.R.K.
  - returns to DOS if called with the eXtender X
  - halt the PC if called with GRUB
- to finish an application you have to
  - finish (or kill) all the user tasks
  - call `exit()`




## call a function at exit time

- sys_atrunlevel

```
int sys_atrunlevel(
  void (*func_code)(void *),
  void *parm, BYTE when);
```

- the when parameter can be:
  - `RUNLEVEL_INIT`
  - `RUNLEVEL_SHUTDOWN`
  - `RUNLEVEL_BEFORE_EXIT`
  - `RUNLEVEL_AFTER_EXIT`

## task, threads and POSIX

- a task can be thought as a POSIX thread
- S.Ha.R.K. implement
  - cancellation, cleanup handlers, thread specific data, join, semaphores, mutexes, condition variables

  in a way similar to POSIX PSE51
- POSIX is implemented through modules and name redeclarations
- primitive names task_* become pthread_*




## tasks and models

- each task is composed by:
  - a model
  - a body `void *mybody(void *arg)`
- the model encapsulates the QoS requirements of the task to the system
  - period, deadline, wcet
- there are a predefined set of task models
- the user can create his/her own models




## models

```
HARD_TASK_MODEL mp;
hard_task_default_model(mp);
hard_task_def_ctrl_jet(mp);
hard_task_def_arg(mp, arg);
hard_task_def_wcet(mp, mywcet);
hard_task_def_mit(mp,myperiod);
hard_task_def_usemath(mp);
```



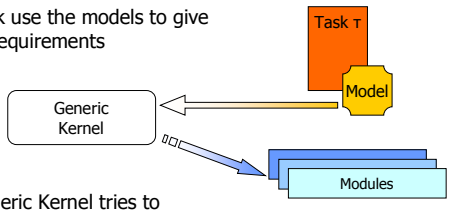

## models (2)

```
SOFT_TASK_MODEL mp;
soft_task_default_model(mp);
soft_task_def_arg(mp, arg);
soft_task_def_group(mp, mygroup);
soft_task_def_met(mp, mymet);
soft_task_def_period(mp,myperiod);
soft_task_def_usemath(mp);
```




## task model

- each task use the models to give its QoS requirements



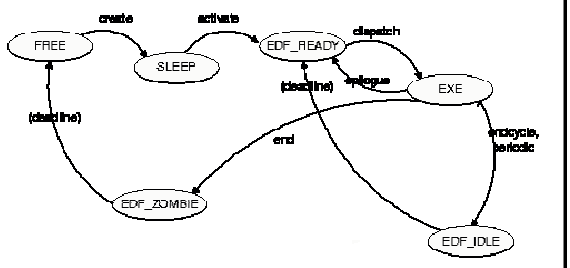

- the Generic Kernel tries to find a module that can handle the model
  - a model is not interpreted by the Generic Kernel




## task creation and activation

- a task can be created...
  - `PID task_createn(char *name, TASK (*body)(…), TASK_MODEL *m, ...)`
  - `PID task_create(char *name, TASK (*body)(…), TASK_MODEL *m, RES_MODEL *r)`
- ...then activated...
  - `int task_activate(PID pid)`
- ...and finally killed!
  - `int task_kill(PID pid)`

## task states for an EDF scheduler







## groups

- each task is identified by a group number
- all the tasks with the same group number can be activated and killed atomically
  - `int group_activate(WORD g)`
  - `int group_kill(WORD g)`
- tasks can also be created and guaranteed atomically in group
  - see the group creation howto




## job execution time (JET)

- S.Ha.R.K. allows the monitoring of the task execution time
  - `int jet_getstat(PID p, TIME *sum, TIME *max, int *n, TIME *curr);`
  - `int jet_delstat(PID p);`
  - `int jet_gettable(PID p, TIME *table, int n);`
- JET must be enabled before task creation

```
soft_task_def_ctrl_jet(mp);
```




## kernel exceptions

- mapped on the RT-signal SIGHEXC
- when a module or a device raises an exception the signal is thrown
  - if you need to redefine a signal handler, send me a mail ;-)
- here are some exception numbers
  - deadline miss (7)
  - WCET exaustion (8)
  - (see `include/bits/errno.h`)




## POSIX cancellation

- specifies how a task reacts to a **kill** request
- there are two different behaviors:
  - deferred cancellation

    when a kill request arrives to a task, the task **does not die**. the task will die only when it will execute a primitive that is a **cancellation point**. this is the default behavior of a task.
  - asynchronous cancellation

    when a kill request arrives to a task, the task dies. the programmer **must** ensure that all the application data structures are coherent.




## cancellation states and cleanups

- the user can set the cancellation state of a task using:

```
int task_setcancelstate(int state, int *oldstate);
int task_setcanceltype(int type, int *oldtype);
```

- the user can protect some regions providing destructors to be executed in case of cancellation

```
int task_cleanup_push(void (*routine)(void *),
  void *arg);
int task_cleanup_pop(int execute);
```

## cancellation points

- the **cancellation points** are primitive that can potentially block a task; if when they are called there is a kill request pending the task will die.
  - `task_testcancel`, `pthread_testcancel`, `sem_wait`, `cond_wait`, `pthread_cond_wait`, `nanosleep`, `task_endcycle`, and others are cancellation points
  - `mutex_lock`, is NOT a cancellation point




## task cancellation

- to kill a task, use:

```
int task_kill(PID p); (or pthread_cancel)
int group_kill(WORD g);
```

- the flag `NO_KILL` can be specified at task creation to inhibit task cancellation.
- POSIX signals kills by default the process and not the threads (tasks). That is, the default signal handler end the whole application.




## mutual exclusion

- allows to execute some code in an **atomic** way with respect to:
  - all the kernel activities
    (the code executes with disabled interrupts)
  - all the tasks
    (the code executes with disabled preemption)
  - only the tasks that share the same resources
    (the code is inserted in the middle of 2 primitives lock/unlock; semaphores / mutexes / CABs)




## low level mutual exclusion

- obtained disabling interrupts
  - nothing can interrupt the code
  - **warning:** no more than a few us!
  - `kern_fsave(), kern_frestore()`
  - `kern_cli(), kern_sti()`
- obtained disabling preemption
  - only interrupts can preempt the running task
  - the scheduler is disabled (priority inversion!)
  - `task_preempt() , task_nopreempt()`




## POSIX semaphores

- used to implement
  - mutual exclusion
  - synchronization
- extends the POSIX semaphores implementing a multi-unit blocking wait
- cancellation points
  - `sem_wait` and `sem_xwait` are cancellation points
  - a non-cancellation point semaphore exists
    - internal semaphores (see the S.Ha.R.K. manual)




## POSIX semaphores (2)

```
sem_t mutex;

int main()
{ …
  sem_init(&mutex,0,1);
  …
  sem_getvalue(&mutex, &val);
}
```

semaphore definition

0 is ignored (see the POSIX standard)

1 is the initial semaphore value

read the value of the semaphore counter

## POSIX semaphores (3)

```
void *demotask(void *arg)
{
  …
  sem_wait(&mutex);
  <critical section>
  sem_post(&mutex);
  …
}
```

blocking wait
other primitives:
`sem_trywait` (non-blocking primitive),
`sem_xwait` (decrement by a value >1)

the typical signaling primitive
other primitives:
`sem_xpost` (increment by a value >1)




## mutexes

- used to implement the mutual exclusion
- allow the use of different protocols (PI, PC, SRP, …)
- a structure called mutex attribute must be used to set a mutex protocol
- POSIX have a little different sintax for the mutexes




## mutexes (2)

```
mutex_t mymutex;
int main() {
  …
  PI_mutexattr_t a;
  PI_mutexattr_default(a);
  …
  mutex_init(&mymutex, &a);
  …
}
```

definition of the mutex

definition of the mutex attribute

initialization of the mutex attribute

initialization of the mutex: now the mutex is a priority inheritance mutex




## mutexes (3)

```
void *demotask(void *arg) {
  …
  mutex_lock(&mymutex);
  <critical section>
  mutex_unlock(&mymutex);
  …
}
```

mutex lock: the mutex resource is now locked

mutex unlock: the mutex resource is now free

difference between semaphores and mutexes: the lock/unlock pair **must be** called by the same task




## condition variables

- used to implement synchronization with mutexes
- a little example
  - 1 mutex and 1 condition variable
  - a semaphore implementation using mutex and condition variables




## condition variables (2)

```
struct {
  mutex_t m;
  cond_t  c;
  int number;
} mysem;
void mysem_init(struct mysem *s)
{
  PI_mutexattr_t a;
  PI_mutexattr_default(a);
  mutex_init(&s->m,&a);
  cond_init(&s->c);
  number = 0;
}
```

## condition variables (3)

```
void mywait(struct mysem &s) {  mutex_lock(&s->m);
  while (!number) cond_wait(&s->c,&s->m);
  s->number--;
  mutex_unlock(&s->m);
}
```

the cond_wait MUST always be put into a cycle that test for the condition




## condition variables (4)

```
void mypost(struct mysem *s) {
  mutex_lock(&s->m);
  s->number++;
  cond_signal(&s->c);
  mutex_unlock(&s->m);
}
```

```
cond_broadcast(&s->c);
```




## cancellation and mutexes

- mutexes are **not** cancellation points
- the condition wait **is** a cancellation point
- when a task is killed while blocked on a condition variable, the mutex **is locked again** before dying
  - a cleanup function must be used to protect the task from a cancellation
  - if they are not used, the mutex is left locked, and there are no tasks that can unlock it!




## cancellation and mutexes (2)

```
void cleanup_lock(void *arg)
{ mutex_unlock(&((struct mysem *)arg)->m); }

void mywait_real(struct mysem *s) {
  mutex_lock(&s->m);
  task_cleanup_push(cleanup_lock, (void *)&s);
  while (!number) cond_wait(&s->c,&s->m);
  task_cleanup_pop(0);
  s->number--;
  mutex_unlock(&s->m);
}
```




## part IV

libraries and drivers




## available libraries

- kernel library
  - task handling
  - function names similar to POSIX pthread Lib.
  - RT signals
  - memory allocation
- standard C library
  - independent part provided by the OSLib
  - dependent part provided by the kernel
    - stdio can be used only when the FS is enabled

## available libraries (2)

- drivers
  - keyboard / mouse / Framebuffer / COM / sound blaster framegrabber / HDD / net / PCI / USB / ...
  - written from scratch
  - derived from Linux using some glue code
- portings
  - MPEG audio / video
  - FFTW




## drivers

- a driver should control an interface
- S.Ha.R.K. supports
  - polling
  - interrupt (fast routine or driver task)
  - DMA
- a typical driver should address three stages:
  - initialization / running / shutdown




## the filesystem

- FAT16 filesystem - allow the usage of the standard C file operations
  - see `demos/oldexamples/fs/initfs.c`
  - see `demos/mesaref`
- if the application is loaded through the X extender you can use some DOS callbacks into `__kernel_register_levels__` and into the `RUNLEVEL_AFTER_EXIT` functions
  - see `oslib/ll/i386/x-dos.h`
  - see `demos/dosfs`




## the console

- direct output to the text mode video memory
- supports 25 and 50 lines text mode
- `#include "ll/i386/cons.h"`
- `void set_visual_page(int page);`
- `void set_active_page(int page);`
- `int get_visual_page(void);`
- `int get_active_page(void);`
- `void place(int x,int y);`
- `void cursor(int start,int end);`
- `void clear(void);`
- `void scroll(void);`




## the console (2)

- `void cputc(char c);`
- `void cputs(char *s);`
- `int  cprintf(char *fmt,...);`
- `void putc_xy(int x,int y,char attr,char c);`
- `char getc_xy(int x,int y,char *attr,char *c);`
- `void puts_xy(int x,int y,char attr,char *s);`
- `int printf_xy(int x,int y,char attr, char *fmt,...);`
- `Colors (BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE)`




## input layer: keyboard

- initialized into `__init__`
- handles keyboard, mouse, joystick, speaker, event debugger
- to read a key use

  `int  keyb_getch(BYTE wait)`
  - wait can be BLOCK or NONBLOCK
  - the returned value is the key pressed
  - also the keycode can be read, using `keyb_getcode`

## input layer: keyboard (2)

- to set the italian keyboard use

```
keyb_set_map(KEYMAP_IT);
```

- to assign an event to a key use

```
KEY_EVT k;
k.flag = CNTR_BIT;
k.scan = KEY_C;
k.ascii = 'c';
keyb_hook(k,endfun);
```




## input layer: mouse

```
MOUSE_PARMS mouse = BASE_MOUSE;
mouse_def_task(mouse,(TASK_MODEL *)&mouse_nrt);
mouse_init26(&mouse);
mouse_setlimits(xmin, ymin, xmax, ymax);
mouse_setposition(320,280);
mouse_setthreshold(2);
mouse_grxshape(img, mask, bpp);
mouse_grxcursor(cmd, bpp);
mouse_on();
mouse_hook(my_mouse_handler);
```




## input layer: joystick / buzzer

```
JOY26_init();
JOY_enable();
JOY_disable();
JOY_getstatus(x0, y0, x1, y1, buttons);


SPEAK26_init();
speaker_sound(hz, ticks);
speaker_mute();
```




## graphic primitives

- Frame Buffer
  - a set of primitives allows the drawing of simple shapes
- MESA libraries are also supported
  - provides a complete library for 3D graphics
- if the graphic card access the video memory using banks, the graphics primitives have to be run in **mutual exclusion**




## graphic primitives (2)

```
FB26_init();
FB26_open(device);
FB26_use_grx(device);
FB26_setmode(device,"640x480-16");
FB26_close(device);

grx_box(x1, y1, x2, y2, GREEN);
grx_plot(x, y, color);
grx_line(x1, y1, x2, y2, color);
grx_text("Goofy", x, y, color, fore, back);
grx_disc(x, y, radius, color);
grx_close();
```




## part V

installing the kernel

## MS-DOS/Windows hosts

- download mindjgpp.zip, sharkXXX.zip, unzip32.exe, and then:

```
1) Download unzip32.exe, mindj333.zip and shark14.zip from the S.Ha.R.K. web
site.
2) unzip32 -o mindj333.zip -d c:
3) cd c:\djgpp
4) install.bat
5) setvar.bat
   (this script automtically set the environement variables
    for DJGPP, you must run this files every time you reboot
    and start a compile session)

Now DJGPP is installed and ready to compile shark

   6) unzip32 -o shark14.zip -d c:
   7) cd c:\shark
   8) Edit shark.cfg:
This step is needed to setup the compiler options and to optimize the kernel for
   faster and more precise time computation.
```




## MS-DOS/Windows hosts (2)

```
   9) make

S.Ha.R.K. is compiled

   10) cd demos
   11) make

The demos are compiled.


If host and target machine are the same and you want to test a demo

   12) cd <demo dir>
   13) x <demo name>

In real DOS environement, you can compile and run a demos without reboot.
```




## Linux hosts

- Download sharkXXX.zip from the website

```
1) Download shark-1.4.tar.bz2 from the S.Ha.R.K. web site
2) tar xvjf shark-1.4.tar.bz2
3) cd shark
4) Edit shark.cfg

This step is needed to setup the compiler options and to optimize the kernel
for faster and more precise time computation.

5) make

S.Ha.R.K. is compiled

6) cd demos
7) make

The demos are compiled. You can run the demos using the FreeDOS bootdisk with
x.exe or directly load a demo thought Grub.
```




## directory tree

- `config`
  - configuration files for the S.Ha.R.K. makefiles
- `distrib`
  - internal scripts for website distributions
- `include`
  - standard include files
  - non-standard includes in separate subdirs
  - oslib includes in `ll`
- `kernel`
  - kernel, tracer
- `modules`
  - scheduling modules




## directory tree (2)

- `oslib`
  - i386 low level source code
  - boot code
  - C library code independent from the architecture
- `drivers`
  - one dir for each device
- `libc`
  - C library dependent from the architecture




## directory tree (3)

- `fs`
  - the file system source code (only FAT 16)
- `port`
  - FFTW, mpeg, mpeg2 and mpg123, mesa, zlib, png, OCERA memory manager
- `lib`
  - compiled libraries
- `demos`
  - some demos that shows the various features of the kernel
- `advdemos, unsupported`
  - other demos, not part of the base distribution

## your application

- use an existing demo as a base for your applications
- a demo typically contains:
  - a makefile
  - an initialization file
  - an application
  - a scheduling module




## your demo application (2)

- the makefile (`demos/mix/makefile`)

```
ifndef BASE
BASE=../..
endif

include $(BASE)/config/config.mk

PROGS=  mix

include $(BASE)/config/example.mk

mix:
	make -f $(SUBMAKE) APP=mix OTHEROBJS="initfile.o"
	    OTHERINCL= SHARKOPT="__OLDCHAR__ __GRX__"
```

your application name (must correspond to a .c file)

other OBJS that compose the application

```
available libraries:
__6025E__     - NI 6025e
__BTTV__      - FRamegrabber
__CM7326__    - CM7326
__CPU__       - CPU Frequency
__DIDMA__     - O(1) Memory Allocator
__FB__        - Frame Buffer
__FFT__       - Fast Fourier Transforms
__FIRST__     - FIRST Framework
__GRX__       - GRX library
__I2C__       - I2C
__INPUT__     - Input layer
__LINUXC26__  - Linux Comp 2.6
__NET__       - Network
__OLDCHAR__   - Keyboard
__OSMESA__    - MESA 3D library
__PCI__       - PCI bus
__PCL833__    - PCL833
__PCLAB__     - PCLAB
__PNG__       - a graphic format
__PPORT__     - Parallel Port
__PXC__       - Framegrabber
__SERVO__     - Servo motors
__SNAPSHOT__  - Snapshot
__TFTP__      - Trivial FTP
__USB__       - USB
__ZLIB__      - compression routines r
... and more!
```




## the multiboot image

- the application is linked statically in a binary image that follows the multiboot standard
- there is not dynamic linking
- all the simbols are resolved at compilation time, and they are allocated at memory addresses >1Mb
- a multiboot image can be booted using GRUB




## running an application

- you need:
  - a PC with MS-DOS
  - a PC with Windows 95/98 (DOS Mode)
  - or a boot floppy (a floppy image is provided on the web site)
- the Kernel can boot
  - using GRUB
    - (no graphical and DOSFS support)
  - using our custom eXtender




## web sites

http://shark.sssup.it

http://lancelot.sssup.it/bugzilla

http://feanor.sssup.it/retis-projects

http://oslib.sourceforge.net




## contact info

Paolo Gai

pj@evidence.eu.com

(please use the Shark Forum for SHaRK related questions!)

http://feanor.sssup.it/~pj

http://www.evidence.eu.com