Implementation of TinyOS3

Lab notes

Vasilis Samoladas

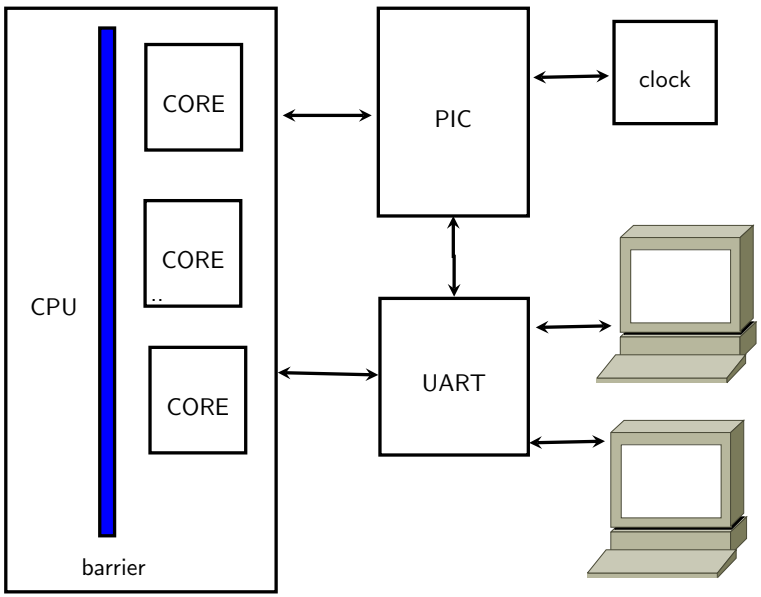
Technical University of Crete

Year 2017–18

Part I

The Virtual Machine

The Virtual Machine (VM) API



File: `bios.h`

```
void vm_boot(interrupt_handler bootfunc,
      uint cores, uint serialno);

uint cpu_core_id;  // This is a variable
uint cpu_cores();
void cpu_core_barrier_sync();
void cpu_ici(uint core);

void cpu_core_halt();
void cpu_core_restart(uint core);
void cpu_core_restart_one();
void cpu_core_restart_all();

void cpu_interrupt_handler(interrupt, handler);
void cpu_disable_interrupts();
void cpu_enable_interrupts();

void cpu_initialize_context(ctx,sp,size,func);
void cpu_swap_context(oldctx,newctx);

void bios_set_timer(TimerDuration usec);
void bios_cancel_timer();
TimerDuration bios_clock();

uint bios_serial_ports();
void bios_serial_interrupt_core(serial,intno,core);
int bios_read_serial(uint serial, char* ptr);
int bios_write_serial(uint serial, char value);
```

Running the VM

cpu_boot

```
typedef void interrupt_handler();
typedef unsigned int uint;

void vm_boot(interrupt_handler bootfunc,
    uint cores, uint serialno);
```

Example:

```
#include <bios.h>
#include <stdio.h>

void bootfunc() {
  fprintf(stderr, "Hello␣from␣core␣%u\n", cpu_core_id())
}
int main() {
  vm_boot(bootfunc, 4, 0);
  return 0; }
```

Information about the cores and syncing

Which core are we on?

```
uint cpu_core_id;    /* Contains core id */
uint cpu_cores();    /* Returns the number of cores.*/
```

Synchronizing the cores

```
void cpu_core_barrier_sync();
```

Interrupts

Operations on each core

```
typedef enum Interrupt
{
        ICI, ALARM,
        SERIAL_RX_READY, SERIAL_TX_READY
} Interrupt;

void cpu_interrupt_handler(Interrupt interrupt,
                           interrupt_handler handler);

void cpu_disable_interrupts();
void cpu_enable_interrupts();

void cpu_core_halt();              // The HLT instruction
void cpu_core_restart(uint c);     // PIC-supported
void cpu_core_restart_one();       // PIC-supported
void cpu_core_restart_all();       // PIC-supported
```

Interrupts and PIC

Sending ICI (Inter-Core Interrupt)

```
void cpu_ici(uint core);
```

Setting ALARM interrupt

```
void bios_set_timer(TimerDuration usec);
void bios_cancel_timer();
```

Routing serial port interrupts to cores

```
void bios_serial_interrupt_core(uint serial,
                                Interrupt intno,
                                uint core);
```

Serial ports

How many are there?

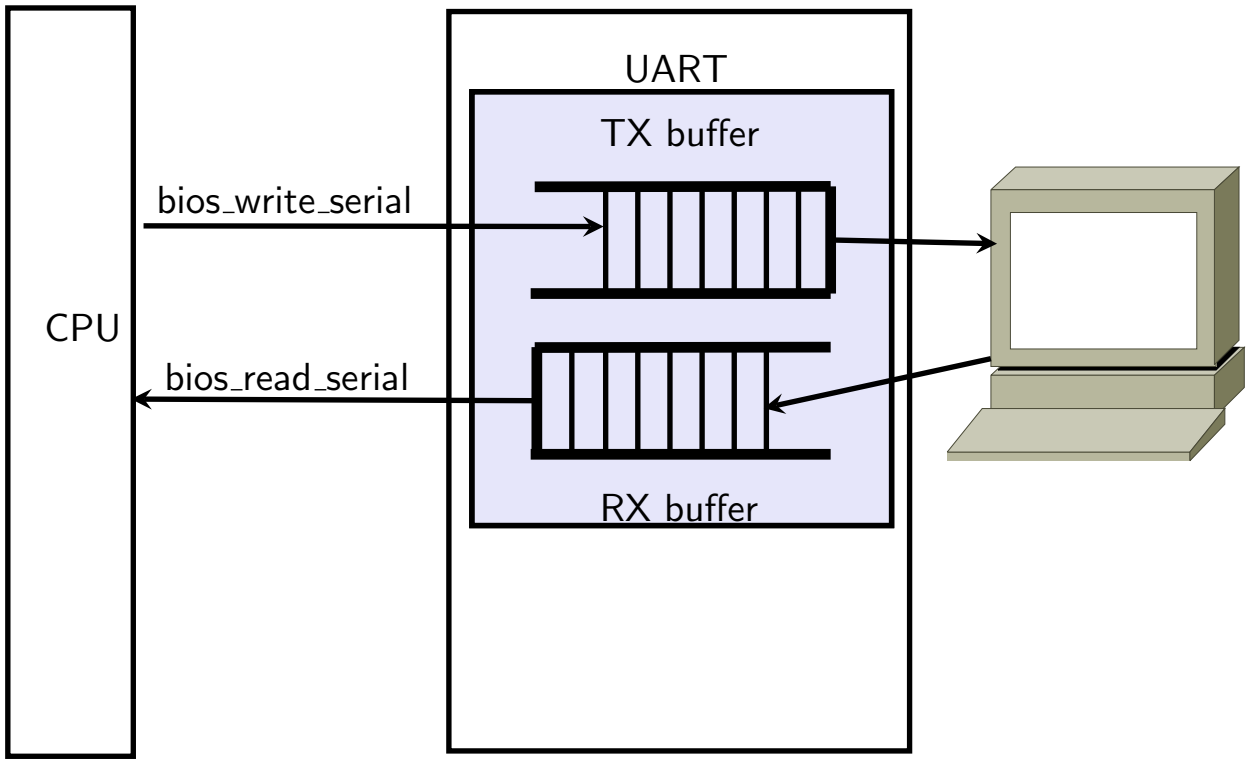
```
uint bios_serial_ports();
```

Performing I/O

```
int bios_read_serial(uint serial, char* ptr);

int bios_write_serial(uint serial, char value);
```

Functionality of serial ports



NOT-READY

- TX buffer full
- RX buffer empty

Interrupt is raised

- When a NOT-READY buffer becomes READY, or
- Timeout: after $\approx$ 300 msec of inactivity.

Part II

The TinyOS3 kernel

System calls [tinyos.h]

Concurrency control

- Mutex_Lock, Mutex_Unlock
- Cond_Wait, Cond_TimedWait, Cond_Signal, Cond_Broadcast

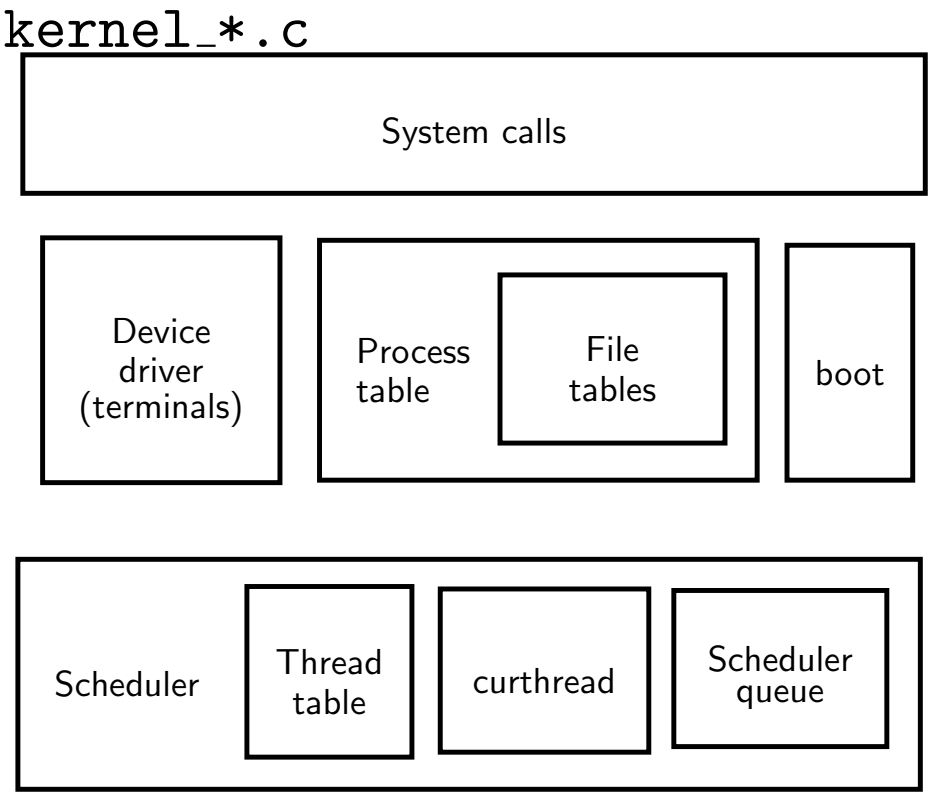
Process control

- Exec Exit Wait_Child
- GetPid GetPPid

Input/Output

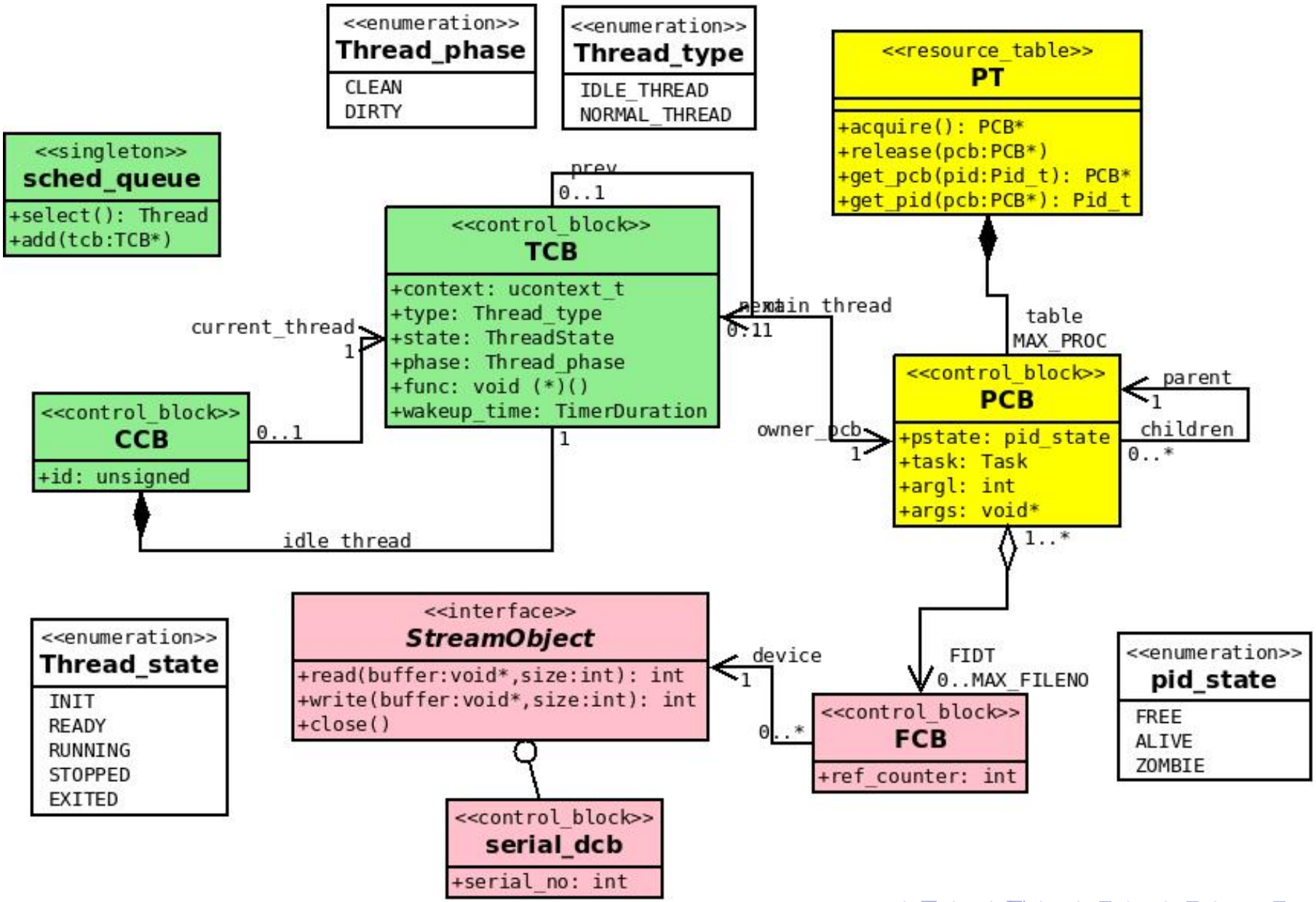
- OpenNull OpenTerminal
- GetTerminalDevices
- Read Write
- Dup2 Close

Parts of the kernel



kernel_cc.h	Concurrrency control and preemption
kernel_sched.h	Scheduler and threads
kernel_proc.h	Process table and system calls
kernel_streams.h	File table and system calls
kernel_dev.h	Device table and drivers
kernel_init.c	Initialization

Kernel data


<<enumeration>>
Thread_phase
CLEAN
DIRTY
<<enumeration>>
Thread_type
IDLE_THREAD
NORMAL_THREAD
<<resource_table>>
PT
+acquire(): PCB*
+release(pcb:PCB*)
+get_pcb(pid:Pid_t): PCB*
+get_pid(pcb:PCB*): Pid_t
<<singleton>>
sched_queue
+select(): Thread
+add(tcb:TCB*)
prev
0..1
<<control_block>>
TCB
+context: ucontext_t
+type: Thread_type
+state: ThreadState
+phase: Thread_phase
+func: void (*)()
+wakeup_time: TimerDuration
current_thread
1
main thread
0..1
table
MAX_PROC
<<control_block>>
PCB
+pstate: pid_state
+task: Task
+argl: int
+args: void*
parent
1
children
0..*
owner_pcb
1
<<control_block>>
CCB
+id: unsigned
0..1
1
idle thread
1..*
<<interface>>
StreamObject
+read(buffer:void*,size:int): int
+write(buffer:void*,size:int): int
+close()
device
1
0..*
FIDT
0..MAX_FILENO
<<control_block>>
FCB
+ref_counter: int
<<enumeration>>
Thread_state
INIT
READY
RUNNING
STOPPED
EXITED
<<enumeration>>
pid_state
FREE
ALIVE
ZOMBIE
<<control_block>>
serial_dcb
+serial_no: int

Booting the kernel [kernel_init.c]

```
void boot(unsigned int ncores, unsigned int nterm,
          Task boot_task, int argl, void* args);
```

1. Saves `boot_task`, `argl` and `args` in global vars.
2. Calls `vm_boot(kernel_boot, ncores, nterm)`

`kernel_boot()`

1. Core 0 initializes all kernel data.
2. Core 0 prepares the initial process: `Exec(boot_task, argl, args)`.
3. All cores enter the scheduler: `run_scheduler()`.

The idle thread

- Each core has a special thread called the idle thread.
- The TCBs of the idle threads are initialized by hand!

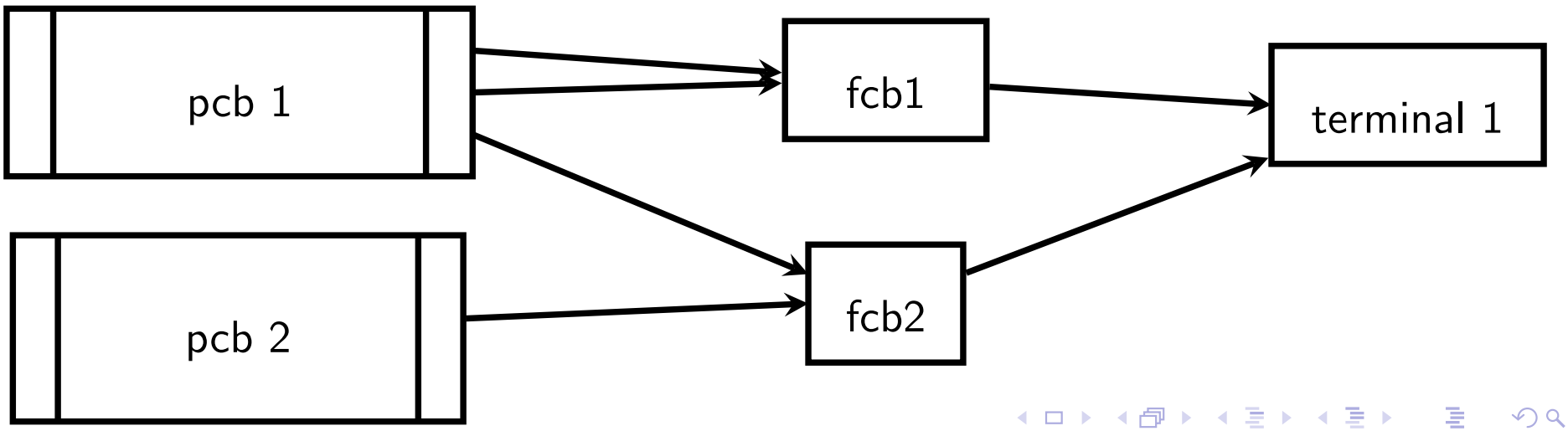
```
void idle_thread()
{
  yield();     /* First time we enter */

  while(active_threads>0) {
    cpu_core_halt();  /* Wait for interrupt */
    yield();          /* Call scheduler */
  }

  /* No more active threads, leave the scheduler! */
  bios_cancel_timer();
  cpu_core_restart_all();
}
```

Files and devices [kernel_dev.c and kernel_streams.c]

- Files, also known as streams, are not the same as devices (e.g., terminals).
- Device Control Blocks (DCBs) are "created" at boot and they live during the life of the VM.
- A new File Control Block (FCB) is created every time we call `OpenNull` or `OpenTerminal`.
- Multiple processes can use the same FCB, and multiple file ids may refer to it.
- `Exec` copies the File Table of the parent to the child process.



Part III

Utilities

Testing your code

- From inside the kernel, you can print error messages using

```
1    fprintf(stderr, "My␣message␣is␣...␣\n", ...);
```

- It is an excellent practice to use `assert(...)`.
  You need to `#include <assert.h>`.
  Then, write code like

```
1    assert( curthread->owner->pid > 0 );
```

  `assert` will check the condition and ignore it if it is TRUE. It will abort the program if it is FALSE.
- It is an excellent idea to write unit tests for your code. You can add your tests in file `test_util.c`.

Writing tests [unit_testing.h]

```
#include "unit_testing.h"

BARE_TEST(my_test, "This is a silly test")
{
  ASSERT(1+1==2);
  ASSERT(2*2*2 < 10);
}
TEST_SUITE(all_my_tests, "These are mine")
{
  &my_test,
  NULL
};

int main(int argc, char** argv) {
  return register_test(&all_my_tests) ||
    run_program(argc, argv, &all_my_tests);
}
```

validate_api

Testing your implementation

It can be quite hard to check all the requirements of the TinyOS3 kernel, or understand why some program (e.g., `mtask`, `tinyos_shell`) does not run correctly.

Solution:

Use `validate_api`.
This program is built automatically every time you invoke `make`.

Usage

```
validate_api [--cores=<cores>] [--term=<terminals>]
    [--nofork] [--list] [--verbose] [--help]
    [--usage] [--version]  TEST  ...
```

validate_api

- A number of tests (see them with `--list`)
- Can run a single test, e.g.,
  `validate_api test_boot` or a number of tests
  `validate_api test_boot test_pid_of_init_is_one`
- Can run tests for different combinations of cores, e.g.
  `validate_api -c 1,2,3,4 basic_tests`
  will run all the basic tests for 1,2 and 4 cpus.
- Many other options, see `--help`

Advanced use of `validate_api`

Use `validate_api` in the debugger

To do this, you must run your test with the "nofork" flag.

Write tests for your own code

This is required! No way to catch bugs otherwise.

Linked lists (util.h)

```
rlnode my_list, *p;
rlnode_init(& my_list, NULL);

 p = malloc(sizeof(rlnode));     /* Add a node */
 rlnode_init(p, "Hello");
 rlist_push_front(&my_list, p);

 p = malloc(sizeof(rlnode));  /* Add another node */
 rlnode_init(p, "World");
 rlist_push_back(&my_list, p);

 while(is_rlist_empty(& my_list) {    /* Show all */
   p = rlist_pop_front(& my_list);
   fprintf(stderr, "%s\n", (char*) p->obj);
   free(p);
 }
```

Intrusive lists: avoiding malloc/free

Put rlnodes inside objects.

```
1  typedef struct {
2      int a;
3      /* other stuff */
4      rlnode node;
5  } FooCB;
6
7  /* Initialize FooCB */
8
9  FooCB foo;
10
11 /* Point at containing
12     object
13  */
14 rlnode_init(& foo.node,
15     & foo);
```

```
1  rlnode L1, L2;
2  rlnode_init(&L1, NULL);
3  rlnode_init(&L, NULL);
4
5  /* No malloc/free */
6  rlnode_push_front(&L1,
7                    &foo.node);
8
9  rlnode_pop_back(& L1);
10
11 rlnode_push_back(&L2,
12                   &foo.node);
13 ...
```