

UNIVERSIDAD POLITÉCNICA DE VALENCIA



DEPARTAMENTO DE INGENIERÍA DE SISTEMAS,
COMPUTADORES Y AUTOMÁTICA

Stand-Alone RTLinux Internals

Vicente Aurelio Esteve LLoret

viesllo@inf.upv.es

Índice general

1. Introduction	1
2. SA-RTL file organization	3
2.1. Directories	4
3. Design for x86 architecture	5
3.1. Boot Process	6
3.1.1. bootsect.S File	6
3.1.2. setup.S File	6
3.1.3. head.S File	6
3.2. Makefile File	6
3.2.1. Steps to insert new OS functionalities	6
3.3. Interrupt Managment	6
3.4. Timers	6
3.4.1. 8254	6
3.4.2. APIC	6
4. Implemented Functions	7
4.1. Funciones Posix	7
4.2. Funciones No Posix	9
5. Developers Guide	11
5.1. Makefile	12
5.2. config.in	12
6. Debugging using GDB Agent	15
6.1. Design	16
6.2. Time Meassure using Breakpoints	16
6.3. New implemented commands	16
7. MiniTracer, Low Overhead Tracer for remote systems	19
7.1. Design	20
7.2. RTL TRACE macro	20
7.3. <i>gdb2vcd.pl</i> script	20

Índice de figuras

5.1. xconfing Main Menu	12
5.2. xconfig POSIX Menu	13
6.1. DDD Window	18
7.1. Tracer Visualization with gtkwave	22

Índice de cuadros

6.1. calc_rdtsc_overhead routine	16
6.2. Interrupt 3 Handler	17
6.3. Time between 2 consecutive breakpoints	17
7.1. Macro RTL_TRACE	21
7.2. Hexadecimal dump of tracerbuf	21
7.3. VCD File dumped	22

Capítulo 1

Introduction

The Stand-Alone RTLinux is an independent implementation of the real time operative system RTLinux. RTLinux is distributed in a package including kernel patch, for several kernel versions, and chargeable modules. The Stand-Alone has his own boot code for that it doesn't depend on the Linux kernel for its own functionality. The RTLinux independation provides us certain advantages and disadvantages:

- We have a RTOS with the same functionality that RTLinux but reducing considerably the OS memory overhead. The most reduced compilation of Linux for x86 architecutre is about 200Kb, In Stand-Alone implementation is only about 25Kb. A too high price in space to use only certain Linux funcionalities.
- RTLinux portability to architectures without paging is impossible since the current Linux portings only covers architectures with paging managment.
- The Stand-Alone RTLinux doesn't have paging managment what allows us to decrease undesirable effects ¹ that produce systems with MMU like TLB and page misses.
- POSIX Standard compliant like RTLinux.
- Interrupt virtualization system of RTLinux although interesting it can fail with modules that use CLI/STI instructions without using the kernel macros, with certain programs like VMWare or with RING3 programs with IOPL=3 that use CLI/STI instructions.

The disadvantages are:

- The communication between Remote System and Host is slow. I've been trying to solve this problem in the Tracer with a new aproach, We will see it later.
- You can no longer use Linux Driver. In RTLinux, development of Linux Kernel also is usefull to RTLinux, being able to have drivers with a non specific RTLinux implementation.
- FIFO interprocess communication mechanism proviedes a quick and simple communication between the Real Time System and Linux because both reside in the same machine. In Stand-ALone this philosophy changes, although it is possible its implementation, is needed a remote communication protocol among 2 computers to carry it out.

¹In real time systems

Capítulo 2

SA-RTL file organization

2.1. Directories

arch/i386	Rutinas dependientes de la arquitectura x86
arch/i386/boot	Código de arranque
arch/i386/boot/tools	Fuentes del programa “build” necesario para la generación del archivo binario
arch/i386/kernel	Rutinas de manejo de interrupciones y timers dependientes de la arquitectura
debug	Funciones para obtener información del sistema
Documentation	Información general sobre el OS
include	Includes
include/i386	Includes dependiente de la arquitectura x86
include/lib	Includes para las funciones de librería del directorio lib
init	Punto de entrada del OS en C. Inicialización del HW, Planificador , y llamada a la inicialización de tareas
tasks	Aquí residen las tareas y la función init_tasks que se encarga de crear e inicializar las distintas tareas
lib	Funciones de librería, funciones matemáticas, manejo de strings etc
main	Gestión de interrupciones, parte no dependiente de la arquitectura
modules	En este directorio se almacenan temporalmente los archivos objeto durante el proceso de compilación
schedulers	Código del planificador
scripts	Código de mkdep (para calcular las dependencias) y IUs para configurar las variables de compilación (generan el archivo .config).

Capítulo 3

Design for x86 architecture

3.1. Boot Process

3.1.1. bootsect.S File

3.1.2. setup.S File

3.1.3. head.S File

3.2. Makefile File

3.2.1. Steps to insert new OS functionalities

3.3. Interrupt Managment

3.4. Timers

3.4.1. 8254

3.4.2. APIC

Capítulo 4

Implemented Functions

4.1. Funciones Posix

Actualmente se han portado las siguientes llamadas posix:

- Pthread Managment
 - pthread_self
 - pthread_create
 - pthread_attr_init
 - pthread_equal
 - pthread_kill
 - pthread_getschedparam
 - pthread_setschedparam
 - pthread_attr_getinheritsched
 - pthread_attr_setinheritsched
 - pthread_attr_getschedparam
 - pthread_attr_setschedparam
 - pthread_attr_getstackaddr
 - pthread_attr_setstackaddr
 - pthread_attr_getstacksize
 - pthread_attr_setstacksize
 - pthread_attr_init
 - pthread_attr_destroy
 - pthread_attr_getdetachstate
 - pthread_attr_setdetachstate
- Scheduler Managment
 - sched_get_priority_max
 - sched_get_priority_min
- Semaphores
 - sem_init

- sem_wait
- sem_post
- sem_destroy
- sem_getvalue
- Mutexs
 - pthread_mutexattr_destroy
 - pthread_mutexattr_init
 - pthread_mutexattr_getprioceiling
 - pthread_mutexattr_setprioceiling
 - pthread_mutexattr_getprotocol
 - pthread_mutexattr_setprotocol
 - pthread_mutexattr_getpshared
 - pthread_mutexattr_setpshared
 - pthread_mutexattr_gettype
 - pthread_mutexattr_settype
 - pthread_mutex_destroy
 - pthread_mutex_init
 - pthread_mutex_getprioceiling
 - pthread_mutex_setprioceiling
 - pthread_mutex_lock
 - pthread_mutex_unlock
- Conditional Variables
 - pthread_cond_init
 - pthread_cond_destroy
 - pthread_cond_broadcast
 - pthread_cond_wait
 - pthread_cond_signal
 - pthread_condattr_getpshared
 - pthread_condattr_setpshared
 - pthread_condattr_init
 - pthread_condattr_destroy
- Time Managment
 - usleep
 - gethrtime
- Spin Locks
 - pthread_spin_destroy
 - pthread_spin_init
 - pthread_spin_trylock
 - pthread_spin_lock
 - pthread_spin_unlock

4.2. Funciones No Posix

- Thread Managment
 - `pthread_wait_np`
 - `pthread_make_periodic_np`

Capítulo 5

Developers Guide

5.1. Makefile

5.2. config.in

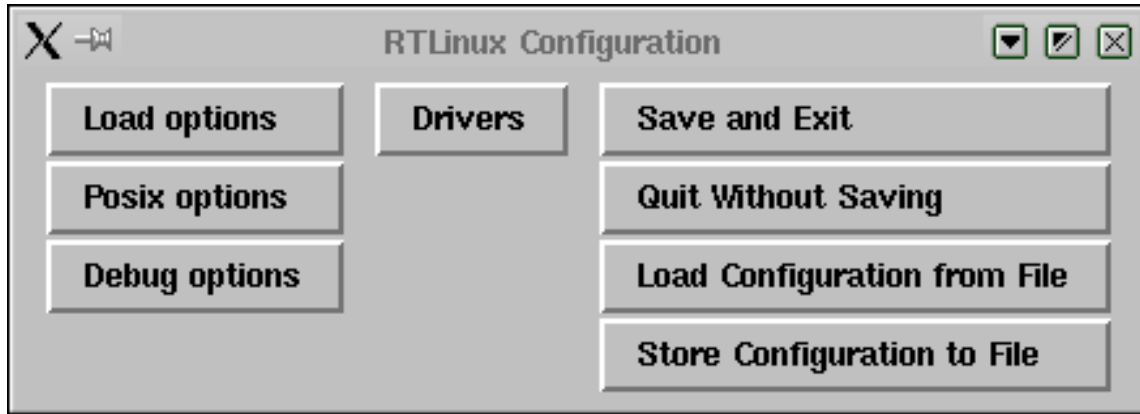


Figura 5.1: xconfging Main Menu

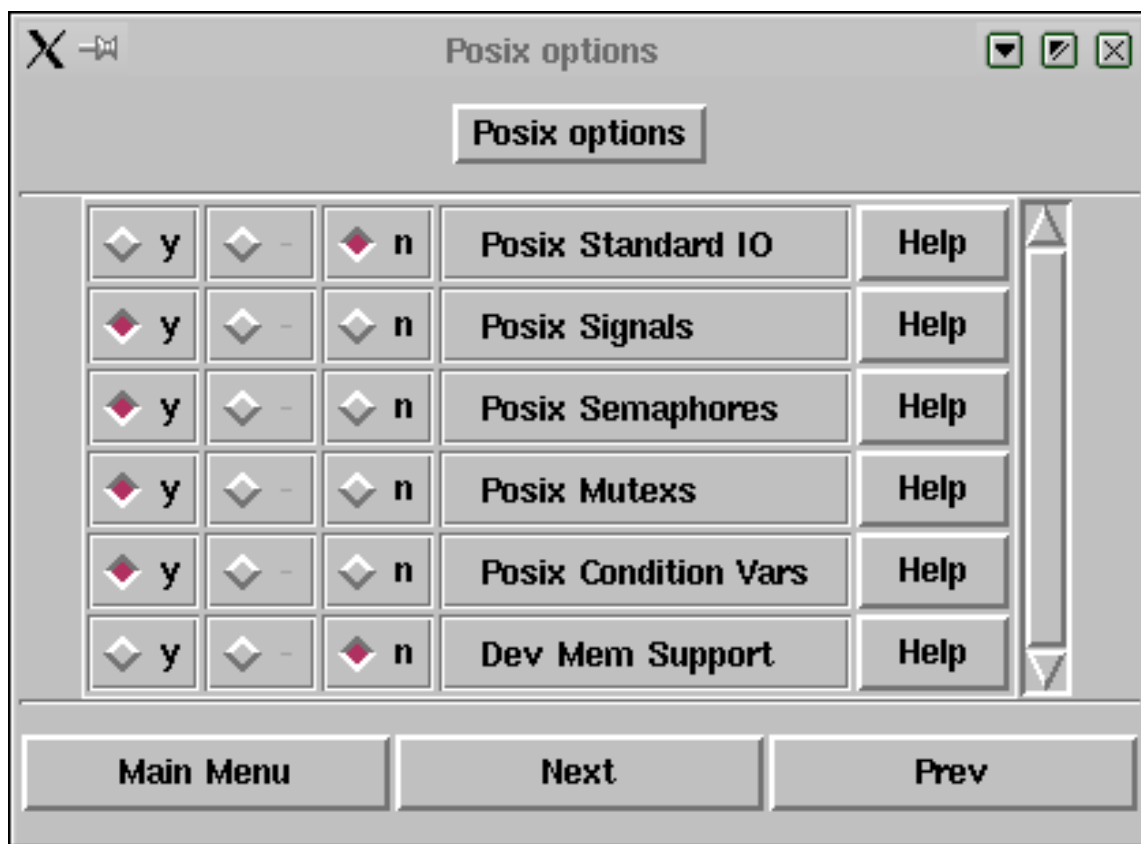


Figura 5.2: xconfig POSIX Menu

Capítulo 6

Debugging using GDB Agent

6.1. Design

GDB Agent provides GDB connectivity using serial port. With few basic remote commands like read/write registers and read/write memory gdb is able to manage debugging ¹. The serial port access and register access make Agent Code architecture dependent.

The ELF file that we get after compilation must have debugging information ². For that, We compile kernel with -g option that it's activated automatically when we turn on GDB Agent option with xconfig .

GDB Agent manages his own interrupts INT 0x3³ and INT 0x1 ⁴. We can't use RTLinux functions in Agent code because if we put breakpoints in these functions system can go into ugly infinity loop.

Int 1 and Int 3 handler are executed completely with interruption disable. We do it defining the 1 and 3 IDT entries like interrupt-gates. The processor disable automatically them when we go into the handler and enable when we go out the handler with the *iret* asm instruction.

we execute int3 and int1 handler with interrupts inhibit, therefore it force us to read serial port by polling. It is possible to do interrupt read when GDB leaves code execute freely using the command *cont*.

6.2. Time Measure using Breakpoints

GDB Agent can get time between two consecutive breakpoints. For that, interrupt handler has been modified like shows 6.2 table.

We get time when we leave handler and when we get into it.

Overhead introduced by interrupt handler call can be measured with `calc_rdtsc_overhead()` routine. We can see in 6.1 his beautiful simplicity.

This time is stored in a 64bits GDB Agent variable called *breakpoint_dif.time*. We can visualizate it using `printf` or display GDB commands like we can see in 6.3 table.

```
void calc_rdtsc_overhead(void)
{
    __asm("int $3\n\t");
    __asm("int $3\n\t");
};
```

Cuadro 6.1: `calc_rdtsc_overhead` routine

6.3. New implemented commands

New commands can be implemented using GDB/Agent protocol. For that, we can use GDB instruction *maintenance packet* "newcommand", "newcommand" is a string that identify a new GDB command. This instruction send a command to the Agent, the agent can return a string with specific information.

- Fcacheon Enable remote system cache.

¹Breakpoints insertion, step by step execution etc

²Function, labels, variable address

³Breakpoints

⁴Step by Step execution


```

#define BUILD_BREAKPOINT_INT() \
asmlinkage void breakpoint_interrupt(void); \
__asm__( \
    "\n" __ALIGN_STR"\n" \
    "breakpoint_interrupt:\n\t" \
    "pushl %eax\n\t" \
    SAVE_ALL \
    "rdtsc \n\t" \
    "movl %eax,(breakpoint_end_time) \n\t" \
    "movl %edx,(breakpoint_end_time+4) \n\t" \
    "call \"SYMBOL_NAME_STR(Breakpoint_Handler)\"\n\t" \
    "rdtsc \n\t" \
    "movl %eax,(breakpoint_start_time) \n\t" \
    "movl %edx,(breakpoint_start_time+4) \n\t" \
    RESTORE_ALL \
    "iret \n\t");

```

Cuadro 6.2: Interrupt 3 Handler

```
printf "%x%x \n",breakpoint_dif_time>>32,breakpoint_dif_time
```

Cuadro 6.3: Time between 2 consecutive breakpoints

- Fcacheoff Disable remote system cache. This command is usefull to measure worst case time.

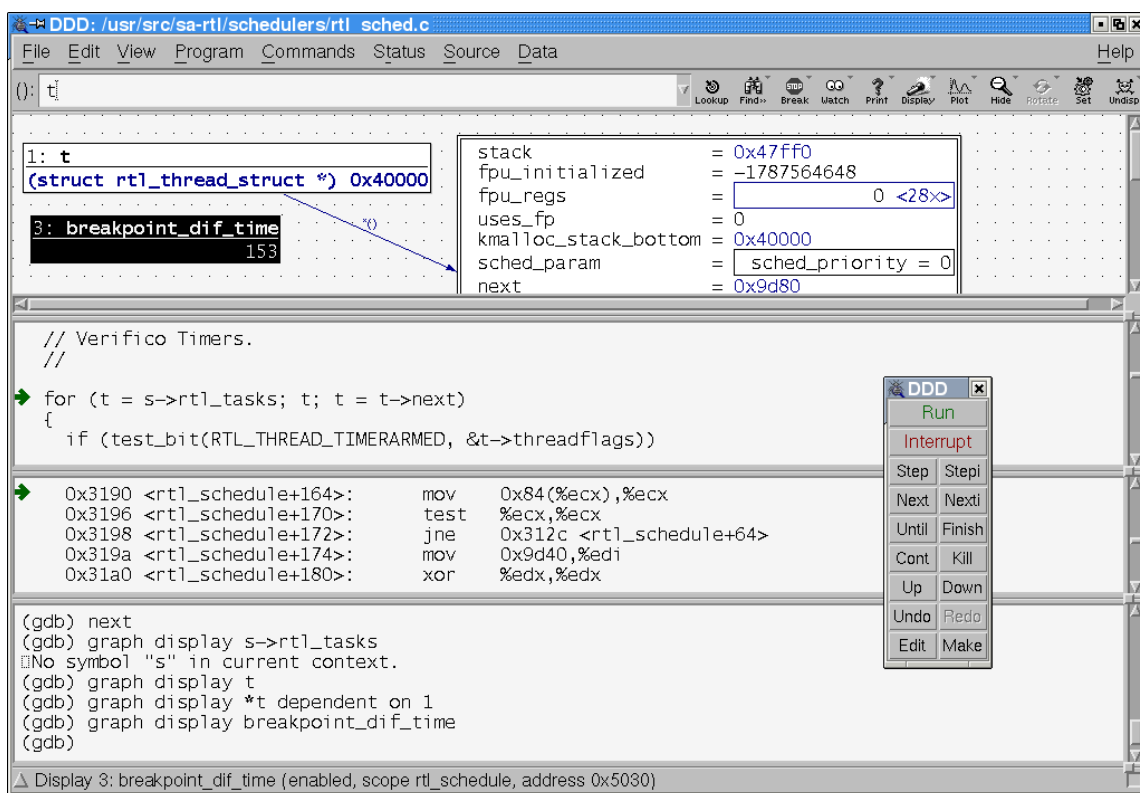


Figura 6.1: DDD Window

Capítulo 7

MiniTracer, Low Overhead Tracer for remote systems

7.1. Design

Tracer allow us to register events like context switch, critic sections, interruptions etc. Trying to avoid overhead introduced by sending this information from remote system we store it in a buffer the event information and when buffer is full we execute a breakpoint to stop processor allowing us to get all these information with *x* GDB command. We can redirect standard output to a file¹ and process this gdb information to another format for his graphical visualization. For that we use *gdb2vcd.pl* script.

7.2. RTL_TRACE macro

La rutina RTL_TRACE permite almacenar un evento en el buffer del tracer. Implementada en forma de macro² y en ensamblador permite reducir considerablemente el overhead introducido. El hecho de implementar esta función en ensamblador puede provocar críticas en los puristas de la portabilidad pero lamentablemente hay otros factores que influyen en la no portabilidad de esta macro. En la implementación para x86 se ha optado por registrar el tiempo mediante la instrucción *rdtsc* y descargar el byte menos significativo y el más significativo para almacenar el identificador de evento y el valor enviado junto al evento³. En arquitecturas con timers de menor resolución es posible reducir el tamaño de evento a 32 bits con lo que disminuiríamos la sobrecarga introducida por el tracer a costa de tener que modificar la implementación del script *gdb2vcd.pl*. Podemos ver el código de la macro RTL_TRACE en la tabla 7.1.

7.3. *gdb2vcd.pl* script

The program *gdb2vcd.pl* was written in Perl, because it's very usufull text managment utility, open log file dumped by *GDB*. GDB dump will be similar to 7.2 table.

It split temporal information, event number and event value and return in the standard output a VCD file that can show us trace information in a graphical way. In 7.1 we can see graphical trace information.

¹Using *gdb rtlinux --tee log*

²Evitamos la sobrecarga introducida por la llamada a una función, call, acceso a parametros en la pila etc

³Como un identificador de tarea en cambios de contexto o un 1/0 para marcar la entrada y salida de ciertas rutinas

```

#if CONFIG_RTL_TRACER
#define RTL_TRACE(event,value)  {
    asm volatile(" rdtsc                                \n"\
        "  shll $8,%%edx                                \n"\
        "  movl %0,%%ebx                                \n"\
        "  movb %%b1,%%al                                \n"\
        "  movl %1,%%ebx                                \n"\
        "  movb %%b1,%%dl                                \n"\
        "  movl (%2),%%edi                               \n"\
        "  movl %%eax,(%%edi)                           \n"\
        "  movl %%edx,4(%%edi)                          \n"\
        "  addl $0x8,%%edi                               \n"\
        "  movl %%edi,(%2)                              \n"\
        "  cmpl %3,%%edi                                 \n"\
        "  jnz  no_overflow%=                            \n"\
        "    int  $0x3                                  \n"\
        "    movl %4,(%2)                              \n"\
        "no_overflow%=:                                \n"\
        :: "r" (value),                                \
        "i" (event),                                  \
        "m" (tracer_ptr),                             \
        "i" (&tracerbuf[TRACERBUFSIZE]), \
        "i" (&tracerbuf[0])                \
        : "eax","edx","ebx","edi","cc");
};
#else
#define RTL_TRACE(event,value)
#endif // CONFIG_RTL_TRACER

```

Cuadro 7.1: Macro RTL_TRACE

```

(gdb) x /4xg tracerbuf
0xafa0 <tracerbuf>: 0x00000d0194bd9506 0x00000d0197676305
0xafb0 <tracerbuf+16>: 0x00000d0197676906 0x00000d01ca573200

```

Cuadro 7.2: Hexadecimal dump of tracerbuf

22CAPÍTULO 7. MINITRACER, LOW OVERHEAD TRACER FOR REMOTE SYSTEMS

```
$version
$end
$timescale 1
$end
$var wire 8 0x01 0x01 $end
$var wire 8 0x02 0x02 $end
$var wire 8 0x03 0x03 $end
$var wire 8 0x04 0x04 $end
$var wire 8 0x05 0x05 $end
$var wire 8 0x06 0x06 $end
$dumpvars
#0
b00000110 0x01
#174542
b00000101 0x01
#174548
b00000110 0x01
#3512733
b00000000 0x01
$end
```

Cuadro 7.3: VCD File dumped

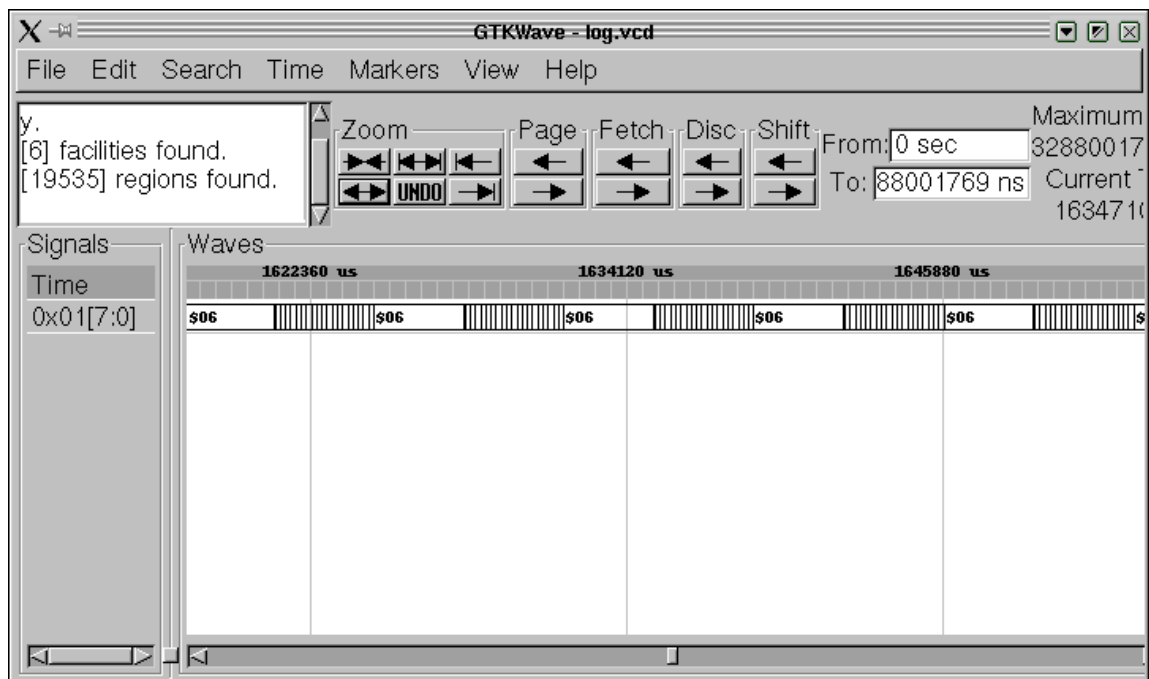


Figura 7.1: Tracer Visualization with gtkwave