

# FSMLabs RTLinux PSDD: Hard Real-time with Memory Protection

After a significant design effort including several false starts, RTLinux Process Space Development Domain (PSDD) was released in 2001. The challenges were to make the system fit cleanly within the RTLinux POSIX threads API, to minimize additional overhead, to prevent the new system from interfering at all with the existing unprotected thread system, and to produce a reliable and maintainable design.

## 1. REAL-TIME AND MEMORY PROTECTION

RTLinux was designed to run at the highest possible speed with the lowest possible overhead. For most of its existence as a product, RTLinux did not offer memory protection for threads on the grounds that the overhead in terms of context switches and requests for system services was unacceptable. The rapid increase in speed and cache size of modern microprocessors, however, and requirements of customers in aerospace and manufacturing test forced a re-evaluation. After a significant design effort including several false starts, RTLinux Process Space Development Domain (PSDD) was released in 2001. The challenges were to make the system fit cleanly within the RTLinux POSIX threads API, to minimize additional overhead, to prevent the new system from interfering at all with the existing unprotected thread system, and to produce a reliable and maintainable design.

PSDD was initially aimed at FSMLabs customers in aerospace manufacturing test. Several of these customers use test frameworks that mix-and-match software simulations and control programs for physical components. Since simulation codes are often very complex and come from multiple suppliers, it is necessary to ensure that coding errors in any simulation component cannot result in undetected changes to data or code of other software - especially software controlling large and powerful physical components. Memory protection is necessary but not in itself a complete solution. PSDD tools include methods for incorporating Fortran and C++ simulations in control loops, a secondary frame or slot scheduler, full support for SMP, and protection against scheduling overruns.

## 2. OVERVIEW OF PROCESS SPACE REAL-TIME

RTLinux is based on a design technique in which a non-real-time operating system runs as a preemptible thread of a real-time kernel. The non-real-time kernel is usually Linux, but NetBSD is also an option. Standard RTLinux threads share a single physical address space. Process Space threads are embedded in the protected address space of UNIX process. Process Space threads are invisible to the UNIX scheduler and remain under the control of a real-time scheduler. The

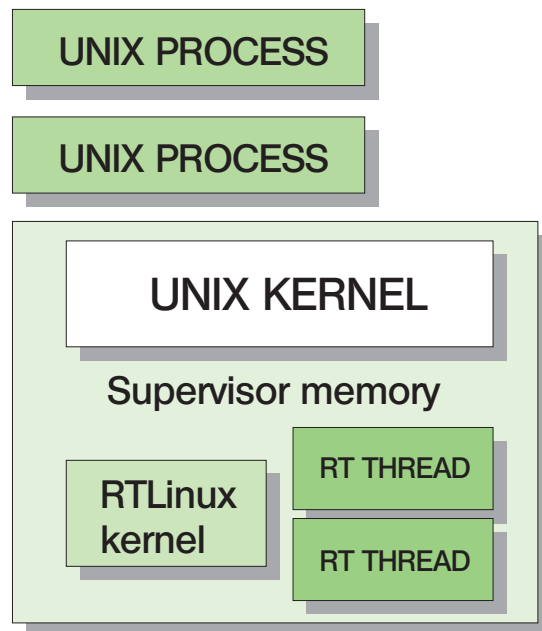


Figure 1.

UNIX home process continues to run under control of the UNIX scheduler and to have full access to the shared address space. An address violation in a Process Space thread will stop execution and invoke the debugger. It is possible to embed multiple process space real-time threads in a single process and to use multiple processes as homes. The home thread may construct UNIX shared memory regions before creat-

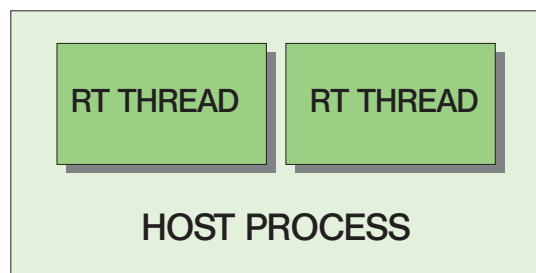


Figure 2.

ing real-time threads so that process space threads can communicate via shared memory.

PSDD was designed with simulation in mind so that a simulation can run in the UNIX home process and one or more real-time threads can collect data and updates the simulation data. Suppose we need to collect data from a sensor every 500 microseconds and then log and display the data. The real-time thread would be a loop: collect data; put data in a queue; and wait for the next period before starting the loop again. The data handling would be taken care of by the Linux process. The interface would be the very efficient lock-free queue. When there is no data to output, the Linux process just sleeps for a second or so. The queue is sized so that there is enough space in it to absorb more than 10 seconds worth of data - which should be more than enough. For a more sophisticated program, we might want to have more elaborate overflow handling. The code outline is as follows.

```
main(){
    create a lock free queue
    lock memory
    create a real-time thread
    loop:
        get data from queue
        if queue was not empty
            output data to file
            output data to display
        else sleep(1)
        goto loop
}

rt_thread(){
    get time
    loop:
        get data
        put data on queue
        sleep for 500 microseconds
        goto loop
}
```

Code sample 1. Pseudo code for a data collection program.

The Linux process can take advantage of buffered I/O, graphics libraries, and all the other facilities of the UNIX system. The real-time components themselves are down to the simplest essence. The RTLinux programming environment is designed to allow the real-time components to be stripped of everything but the time critical software because getting timing right is difficult enough on its own. As much as possible is delegated to the non-RT system. The complete code in code sample 1 shows the simplicity of the system.

The API of PSDD threads is a subset of the standard RTCore Pthreads API (see code sample 2). In fact, PSDD can be used as a prototyping system for kernel real-time threads. Once software has been made to work in process space, it can be recompiled to work, without modification, in kernel space - as long as it does not depend on access to ordinary user space memory.

```
/* create a queue for data samples
   the queue length should be a power of 2 */
DEFINE_OWQTYPE(sample,32768,int,-1,-1)
DEFINE_OWQFUNC(sample,32768,int,-1,-1)
thread_t thread;

int main(void) {
    sample myq;
    void *device;
    FILE *LOGFILE = fopen("mylogfile","w");
    mlockall(MCL_CURRENT | MCL_FUTURE); // required.
    device = map_device(); // device dependent.
    /* now make the real-time thread */
    rtl_pthread_create(&thread, NULL, &thread_code, NULL);
    while (1) {
        long data;
        if( (n = sample_deq(&myq))>0){
            fwrite(LOGFILE,&n,sizeof int);
            display(n);
        }else sleep(1);
    }
}
```

The process space real-time thread itself is just a loop.

```
void *thread_code(void *param) {
    struct timespec next;
    rtl_clock_gettime(RTL_CLOCK_REALTIME, &next);
    while (1) {
        int data;
        timespec_add_ns(&next, 500000);
        rtl_clock_nanosleep(RTL_CLOCK_REALTIME,
            RTL_TIMER_ABSTIME,
            &next, NULL);
        data = *(int *)device & 0xffff;
        sample_enq(&myq,n); // ignore overflow.
    }
    return NULL;
}
```

Code sample 2.

### 3. THE FRAME SCHEDULER

The standard RTLinux scheduler is purely priority driven and preemptive. A process space frame scheduler provides a second tier scheduler for applications needing a slot or frame scheduler. The frame scheduler imposes a cyclical schedule, guards against certain error conditions, and allows for dynamic modification of scheduling parameters.

The frame scheduler itself is implemented by the combination of a process space real-time task and its home process. There may be several slot schedulers running concurrently on the same machine. It is up to the user to ensure that there are no conflicts between the schedules.

The frame scheduler supports hard real time scheduling of user space tasks in terms of frames and minor cycles. There is a fixed number of minor cycles per frame. Minor cycles can be either time-driven or interrupt-driven. For each task, it is possible to specify task priority, the CPU to schedule the task on, the starting minor cycle number within the frame, and the run frequency in terms of minor cycles. (For example, if there are 10 minor cycles in a frame, the starting minor cycle is 2, and the run frequency is 3, the task will run at the following minor cycles: 2, 5, 8, 2, 5, 8, ...). If there are mul-

Function Groups	Description
rti_clock_gettime, rti_usleep, rti_clock_nanosleep, rti_nanosleep	POSIX clock and sleep functions
rti_open, rti_close, rti_ioctl, rti_lseek, rti_read, rti_write	POSIX file IO
rti_cpu_exists, rti_getcpuid	RTLinux multiple CPU support
rti_thread_attr_init, rti_thread_attr_destroy, rti_thread_attr_setcpu_np, rti_thread_attr_getcpu_np, rti_thread_attr_setfp_np, rti_thread_attr_getfp_np, rti_thread_attr_setschedparam, rti_thread_attr_getschedparam, rti_thread_attr_setstackaddr, rti_thread_attr_getstackaddr, rti_thread_attr_setstacksize, rti_thread_attr_getstacksize	POSIX thread creation attributes
rti_thread_create, rti_thread_cancel, rti_thread_exit, rti_thread_join, rti_thread_equal, rti_thread_kill, rti_sched_get_priority_max, rti_sched_get_priority_min, rti_thread_self	POSIX thread control functions
rti_sem_destroy, rti_sem_getvalue, rti_sem_init, rti_sem_post, rti_sem_timedwait, rti_sem_trywait, rti_sem_wait	POSIX semaphore support
rti_printf	Message logging

Table 1. The PSDD API.

multiple tasks ready at the start of a minor cycle, the task with a higher priority is run first.

Threads running under the frame scheduler are generally written as loops of the following form.

```

loop:
    fsched_block(); // wait for minor cycle
    do work
    go to loop
    
```

Code sample 3.

The main routine calls the scheduler to initialize and then start real-time threads.

The thread code itself does not contain any scheduling information. Scheduling information is supplied when attaching a new task to the scheduler via command line interface. This approach allows the user to change schedules and control algorithms without recompiling. The user manipulates the frame scheduler via the "fsched" command.

Typically, running frame scheduler programs is accomplished with a shell script like the following:

```

fsched create
sleep 1
fsched config -mpf 10 -dt 50
fsched attach -n user1 -f 3 -smc 2 -p 1
fsched start
    
```

Code sample 4.

Here we create a frame scheduler, configure it to each 50 milliseconds period with 10 minor cycles per frame. Then we attach a user program to execute starting at minor cycle 2 of each frame with the frequency of 3 minor cycles. The task runs at priority 1. Finally, the whole system is started with the *fsched start* command.

## 4. TECHNICAL OVERVIEW AND API

PSDD is a subsystem within RTLinux that establishes a system call connection to the Unix thread and a means of changing the memory map context for RTCore threads. A process space thread is created by a call to *rti\_thread\_create* from within a Unix process - called the home process of that thread. All pages of the home process must be locked in the physical RAM, e.g. with *mlockall(2)*. The *rti\_thread\_create* call is propagated into supervisor mode RTLinux and a standard real-time thread is created and marked as a process space thread. Whenever the RTCore scheduler activates a process space thread,

the scheduler restores the memory management context of the home process.

The RTLinux paradigm of separating real-time and non-real-time application components is extended to UNIX process space with PSDD. Typically, the main() program performs application-specific initialization, locks down process pages in memory, creates some real-time threads using *rti\_thread\_create*, and then proceeds to interact with them or just sleeps. Note that real-time threads execute in the same address space as the process, so shared memory is automatically available.

PSDD fully supports SMP. Applications can schedule threads on any available CPU in the system. In addition, user space implementation makes adding support for other programming languages a relatively easy task. While RTLinuxCore only supports C, PSDD applications can be written in either C or C++. Other languages may be made available if needed. (Table 1)

In addition to the standard POSIX scheduling support, PSDD includes a frame scheduler facility. It is specifically designed to handle periodically repeating (frame-based) schedules. One of the key features of the frame scheduler is separation of the scheduling information from the control algorithm. This is useful in simulation and control applications - users can change the schedules without recompiling their programs. PSDD frame scheduler also provides special support for sharing memory between user space processes that can be used for building modular systems.

```
fsched config -mpf minor_cycles_per_frame -dt
dt_per_minor_cycle [-s sched_id] [-i interrupt_source]
```

*Code sample 5. This command sets the scheduler parameters: number of minor cycles in a frame, the length of a minor cycle, and the interrupt source.*

```
fsched [-s sched_id] startstop
```

*Code sample 6. start and stop the frame scheduler.*

```
fsched [-s sched_id] pauseresume
```

*Code sample 7. pauses and resumes the execution at the next minor cycle.*

```
fsched attach [-s sched_id] -n program -p priority -f run_freq -
smc starting -cpu cpu_number -args arguments passed to user
process"
```

*Code sample 8. attach a program to the frame scheduler. "program" is the name or path of the executable to start. "priority" can lie between 1 (min) and 255 (max). If the CPU is not specified, the default CPU is used. The task starts the execution at the "starting" minor cycle number of the next frame with "run\_freq" frequency.*

```
fsched debug -p pid
```

*Code sample 9. break in the user process "pid". The break happens at the next minor cycle, and all scheduling activity stops. After that, it is possible to attach to the process with GDB and perform source-level debugging.*

```
fsched info [-s sched_id] [-n average_runs]
```

*Code sample 10. display the statistical information about the schedulers and tasks. For each task, fsched info displays execution statistics: last, running average, min and max execution times in microseconds, total number of execution cycles, and number of overruns. Percentage of the current CPU time used by the RT tasks is also displayed.*

## 5. CREDITS

PSDD was primarily designed by Michael Barabanov with a great deal of assistance from Cort Dougan. While a very early version of RTLinux ran almost entirely in user space, the initial target Intel 486/33MHz systems were slow at crossing protection domains. RTLinux was then redesigned to run in kernel space. As processor and memory speed increased, especially in relation to the bus and interrupt controllers that came to dominate timings, user mode became more plausible. Several years ago, Piere Cloutier developed LXRT a very interesting user mode capability for RTAI - a "fork" of the open RTLinux project. LXRT was interesting for a couple of reasons and FSMLabs began a research effort to find a way to provide protected memory to threads in a way compatible with our goals of

standardized, minimal API and a high performance maintainable and portable design.

As a first effort, Cort Dougan implemented an RTLinux user mode facility called PSC (Process Space Control). PSC allowed users to install real-time interrupt handlers in the space of Linux processes and it was successfully used by many FSMLabs customers. PSC was really designed to make it easier to migrate code from systems like VxWorks and PSOS. Linux threading was much faster than what is offered in those systems, so a threaded program could easily be moved to run in Linux user space, with ISR's hooking to the PSC features. Steve Rosenbluth, at the time at the Jim Henson Creature Shop, helped considerably in debugging and improving the interface to PSC. Eventually, we decided that PSC was too limited, too different from the standard RTCore programming environment, and too narrowly focused to the porting problem. Redesign lead to PSDD.

PSDD in its current form, and the frame scheduler in particular, benefited greatly from the advice and experience of Dean Anneser and other members of the Real-Time Test Systems group at United Technologies in East Hartford Connecticut. The requirements of United Technology's test system, refined during 15 years of application to a wide variety of jet engine test, drove development. Our theory is that design of operating system components should be motivated by support of applications. ■

*Extra background information on Victor Yodaiken and Michael Barabanov is not available.*

## 6. APPENDIX: FRAME SCHEDULER

```
void rt_thread(void*arg) {
    /* thread executed in hard real-time */
    while (1) {
        /* block the execution until the next run */
        fsched_block();
        user_code();
    }
}

int main(int argc, char **argv) {
    struct fsched_task_struct task_desc;
    application_init();
    // initialize RT subsystem
    fsched_init(argc, argv, &task_desc, NULL);
    // start real-time thread
    fsched_run(rt_thread, &task_desc);

    /* main thread sleeps forever; hard RT thread is running*/
    while (1) {
        sleep(1);
    }
}
```

*Code sample 11.*