
Linux® Threads & The Frequency Based Scheduler

*Jeff Hollensen
Software Engineer
Concurrent Computer Corporation*

Abstract: RedHawk™'s Frequency Based Scheduler provides tools, libraries, commands, and kernel support for scheduling independent real-time processes with critical time constraints or synchronization requirements. Typically, such processes communicate via shared memory. Multi-threaded processes can take advantage of the same real-time scheduling features without having to partition their address space or resort to shared memory paradigms.

Brief Overview of the Frequency Based Scheduler

The term “Frequency Based Scheduler” (often abbreviated as FBS) refers to a collection of commands, tools, application programming interfaces, and operating system kernel support which together define, control, and monitor the scheduling of individual real-time processes. The FBS lends itself well to applications with cyclic scheduling requirements, such as flight simulators, which are often triggered from an external interrupt source or real-time clock. The FBS can also be used to synchronize individual processes without cyclic scheduling requirements.

The creation of a *scheduler* involves specification of a set of parameters which include the number of processes to be scheduled, the frequency (if any) of individ-

The scheduling attributes can be dynamically monitored and adjusted as processes execute by any of the FBS interfaces. The execution performance of individual processes on a scheduler can be monitored as well.

How Scheduling Works

Processes which are members of a scheduler continue their execution unimpeded until they reach a *cycle point*. A cycle point is defined by the user application by calling a simple API function called *fbswait(3)*. This function call causes the application to block and yield the CPU until the next FBS cycle is triggered by the interrupt source. Based on the process' period, it will then continue execution after returning from the *fbswait()* call.

In the simplest form, the following code sequence shows the ease of integrating an application with the FBS:

```
#include <fbsched.h>
main ()
{
    while (fbswait() == 0) {
        // do something
    }
}
```

Multi-threaded Processes

Multi-threaded applications may also be scheduled via the FBS in one of two ways:

- *Main Thread Scheduling*
- *Multi-threaded Scheduling*

The *Main Thread Scheduling* technique is a simple extension of the basic use of the FBS. The main thread can spawn additional threads which operate independently of the FBS. In such a case, the main thread would call *fbswait()* at its cycle point. Additional threads can execute asynchronously of the main thread, or use standard Linux thread interfaces to synchronize with the main thread. ***Only*** the main thread may call *fbswait()* using this technique.

The *Multi-threaded Scheduling* technique is more interesting. With this method, the main thread may or may not be a member of a scheduler. After creating additional threads, each individual thread that is to become a member of the scheduler

will join a pre-existing scheduler via a call to the API function *fbsschedself(3)*. A required parameter to *fbsschedself()* is a structure which defines the cyclic scheduling attributes of the calling thread, including its starting cycle and period. Other scheduling attributes of the thread, including its policy, priority, and CPU affinity may be set externally via FBS utilities or by calling standard POSIX® API functions.

Once a thread becomes a member of a scheduler, its execution performance can be monitored and its scheduling attributes can be monitored and manipulated by any of the interfaces provided with the FBS, including:

- rtcp command
- API function calls
- NightSim graphical tool

How Are Threads Scheduled?

Threads which are members of a scheduler execute in exactly the same manner as independent processes. They define their cycle point via a call to *fbswait()*.

The implementation is simplified by the nature of Linux threads. Linux threads are simply *cloned* processes. A cloned process is a full-weighted process which shares attributes of its parent process; typically its entire address space, file descriptors, etc. Threads are created by the *pthread* library *pthread_create()* function by calling the *clone(2)* system service call. As RedHawk Linux adapts to changes in threads implementations which are underway within the Linux community, it will ensure that bound threads will always be able to be scheduled via the FBS.

How Do Threads and Processes Differ With Respect to the FBS?

In reality, once a thread becomes a member of a scheduler, the distinction is lost. The distinction is actually in how a *process*, be it a single-threaded process or a thread, becomes associated with a scheduler. There are two basic mechanisms by which a process becomes a member of a scheduler:

- A process is created by executing an executable program file
- An existing process joins a scheduler by executing an API call itself

In the former case, any of the three FBS interfaces (rtcp command, API, or NightSim) can specify that a named executable program file should be exe-

cuted (essentially an *exec(2)* system call) and that resultant process becomes a member of a scheduler.

In the latter case, only the API is available for joining an FBS.

Alternative Methods

While it might be convenient for the *rtep* command or NightSim tool to have the latter capability, it would require insight and integration into the user's application. Creation of threads requires specification of user-defined function addresses executed by the user's process. Neither the *rtep* command nor the NightSim tool have the ability to take control of a process or make function calls on its behalf. NightView, RedHawk's real-time debugger, includes a "hot patch" capability which could be used to execute the required calls to *pthread_create()* and *fbsschedself()*. A hot patch consists of adding machine code representing a user-defined expression (including function calls!) to a running process at a user-specified location (line number or address) without stopping the process. When the process executes at the specified location, the new machine code is executed after which the process resumes its normal instructions. Hot patches are extremely powerful and efficient. The only drawback of this technique is that NightView must initially take control of the process before hot patches can be inserted. Currently, acquiring control of a process requires that it be launched from within NightView or that NightView attach to it -- both operations require the process to be stopped for a short period of time before real-time execution can continue (subsequently unimpeded by NightView).

A Multi-threaded Example

The following is a simple example of a program that creates two threads which join an existing scheduler. For this example, assume the scheduler identifier is passed to the program on the command line. Error checking has been omitted for brevity:

```
#include <pthread.h>
#include <fb Sched.h>

int fbs_id;
#define NUM_THREADS 2
pthread_t  threads[NUM_THREADS];
int        identity[NUM_THREADS];
int        results[NUM_THREADS];

work() {}
```

```
void *
thread_main (void * formal)
{
    int whoami = *(int*) formal;
    struct fbssched_buf buf;
    char thread_name[80];
    int * result = &results[whoami];

    buf.version = FBSSCHED_BUF_V1;
    buf.param   = 0;
    buf.period  = 1;
    buf.cycle   = 0;
    buf.ab      = 0;
    buf.fpid    = -1;
    sprintf (thread_name, "thread_%d", whoami);

    (void) fbsschedself(fbs_id, thread_name, &buf);

    // The main execution loop for this thread
    while (fbswait() == 0) {
        work();
    }

    pthread_exit (result);
}

int main (int argc, char * argv[])
{
    int      i;
    void     * exit_code;

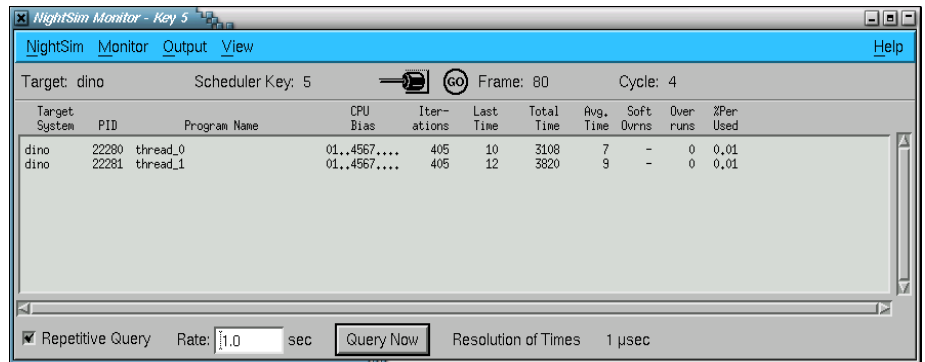
    fbs_id = atoi(argv[1]);
    for (i=0; i<NUM_THREADS; ++i) {
        identity[i] = i;
        threads[i] = -1;
        (void) pthread_create (&threads[i],
                               NULL,
                               thread_main,
                               &identity[i]);
    }
    for (i=0; i<NUM_THREADS; ++i) {
        if (threads[i] != -1) {
```

```

        (void) pthread_join (threads[i],
                            &exit_code);
    }
}
}
// Program End

```

Once the threads are created and make their call to *fb Schedself()*, the NightSim graphical utility can be used to monitor their execution performance or to dynamically adjust their scheduling attributes (priority, CPU affinity, period). The following figure is an example of NightSim's Monitor window monitoring the execution of threads in the example above:



Conclusion

Integrating RedHawk's Frequency Based Scheduler into a multi-threaded program is as simple as encoding two FBS API calls into individual bound threads. The ease of integration combined with the flexibility and real-time performance of the Frequency Based Scheduler make porting or designing multi-threaded real-time applications with RedHawk Linux worth serious investigation.

