Thread scheduling and related interfaces in NetBSD 5.0

Mindaugas Rasiukevicius
The NetBSD Project
rmind at netbsd org

May 4, 2009

Introduction

A lot of new features were implemented in the NetBSD 5.0 release, and many improvements were made in the areas of scheduling, threading and symmetric multiprocessing (SMP). Like other modern UNIX-like operating systems, NetBSD supports traditional processes created by *fork(2)* and native POSIX threads (*pthreads*). Prior to the 5.0 release, user threading on NetBSD was implemented using a mechanism called *scheduler activations (SA)*. The SA implementation was complicated, scaled poorly on multiprocessor systems and had no support for real-time applications. In NetBSD 5.0 these deficiencies have been addressed by replacing SA with an entirely new, scalable 1:1 threading model. In a 1:1 model each user thread (*pthread*) has a kernel thread called a *light-weight process (LWP)*. Inside the kernel, both processes and threads are implemented as *LWPs*, and are served the same by the scheduler, as in Solaris, and other systems. In this article, we will review the scheduling of threads, related application programming interfaces, and utilities in the NetBSD operating system. Since the focus of this article is to introduce readers to new interfaces, we will not go into the details of implementations inside the kernel.

Real-time and scheduling classes

A real-time system is a *predictable* system which aims to meet certain time constraints (deadlines). Failure to meet these time constraints usually indicates hardware failure. Systems can be classified as either *hard* or *soft* real-time systems. Hard real-time systems shall meet the requirements unconditionally. That is, their predictability is deterministic. Soft real-time systems are not deterministic; they can tolerate some latencies, but their objective is to minimize them. NetBSD 5.0 provides *soft* real-time extensions.

According to the POSIX standard, at least the following three scheduling policies (classes) should be provided to support the POSIX real-time scheduling extensions:

- *SCHED_OTHER*: Time-sharing (TS) scheduling policy, the default policy in NetBSD.
- *SCHED_FIFO*: First in, first out (FIFO) scheduling policy.
- *SCHED_RR*: Round-robin scheduling policy.

The standard defines algorithms for the real-time *SCHED_FIFO* and *SCHED_RR* policies, and leaves *SCHED_OTHER* as an implementation-defined policy; that is, it is specific to the operating system. All three policies are provided in NetBSD 5.0 and fit in the following in-kernel priority model:

Kernel (RT)	192 .. 223	32 levels	Software interrupts.
User (RT)	128 .. 191	64 levels	Real-time user threads (SCHED_FIFO and SCHED_RR policies).
Kernel threads	96 .. 128	32 levels	Internal kernel threads (*kthreads*), used by the I/O, VM and other kernel subsystems.
Kernel	64 .. 95	32 levels	Kernel priority for user processes/threads, which is temporarily given when process/thread enters the kernel-space and blocks (sleeps).
User (TS)	0 .. 63	64 levels	Time-sharing range, where user processes/threads run by default (SCHED_OTHER policy).

These priorities are only used in the kernel, and internals are revealed only to provide a better understanding of the scheduling system as a whole. The POSIX standard requires at least 32 priority levels for the real-time scheduling policies. NetBSD 5.0 provides 64 priority levels, which are internally mapped to the appropriate kernel range shown above. It is not portable to depend on any of these constants, therefore using them is strongly discouraged. Applications should determine the priority range of the specific policy using the following functions defined by the standard:

```
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

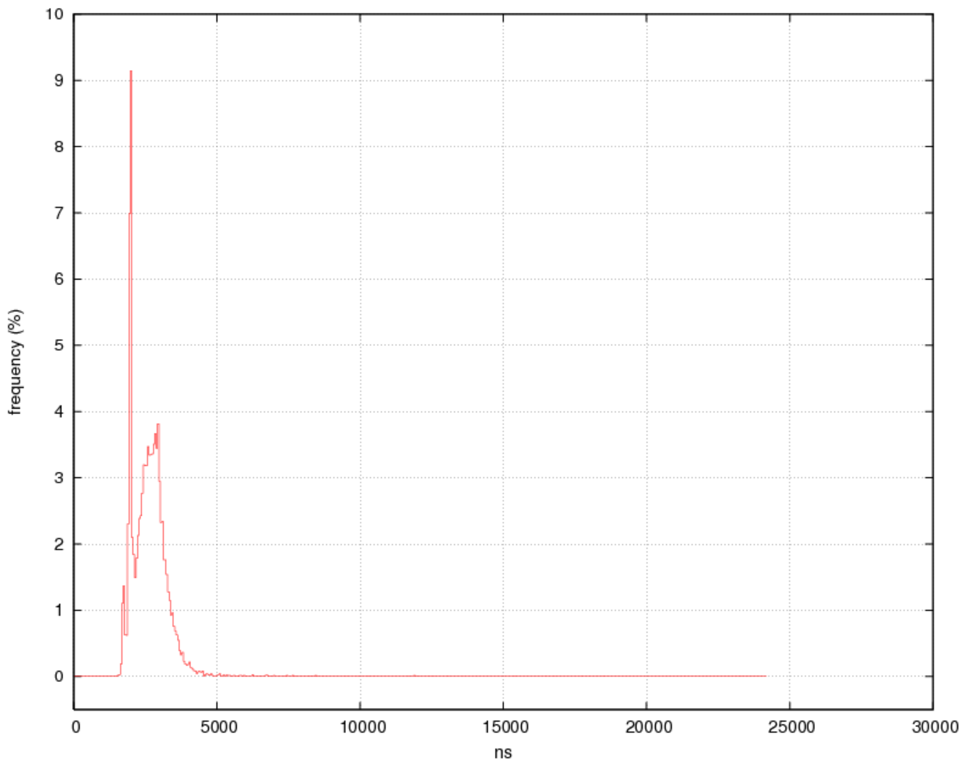
In NetBSD, the run queue of LWPs is implemented as a traditional *multi-level feedback queue*, like in many UNIX-like operating systems. The default *SCHED_OTHER* policy is either the original 4.4BSD scheduling or traditional UNIX time-sharing approach like in Solaris, depending on which scheduler is chosen. It can be chosen using the **SCHED_4BSD** or **SCHED_M2** kernel options. One of the main objectives of a time-sharing queue is to give a priority boost for I/O bound threads (which allows the kernel to provide good response times for interactive tasks) and ensure fairness. Each LWP has a priority and time-quantum (amount of time allocated by the scheduler to run on the CPU). In a time-sharing queue, both values are dynamically calculated by the scheduler.

Threads running with the *SCHED_FIFO* policy, which is a real-time policy, have a fixed priority. That is, the kernel does not change the priority dynamically. Only the super user can set the scheduling policy to *SCHED_FIFO*. Under this policy, a thread runs until completion, or it:

- Voluntarily gives up the CPU (yields).
- Blocks on I/O operation or other resources (memory allocation, locks, etc).
- Gets preempted by a higher priority real-time thread.

SCHED_RR works in the same way as *SCHED_FIFO*, except threads running with this policy have a *time-quantum (time-slice)*, which by default is 100 ms. Thus, unless it encounters one of the three cases above, the thread finishes when its *time-quantum* expires. Threads at the same priority level are processed in a round-robin way.

Figure 1: Real-time thread dispatch latency (threads bound to *pset*), 8 core Xeon with background load.



A simple latency test with two *SCHED_FIFO* threads bound to the same CPU when the system is under load (running *./build.sh -j16*) has shown that NetBSD with kernel preemption enabled tends to respond within 5 microseconds (see Figure 1). Microsecond-level latencies for real-time threads are usually what real-time operating systems, like QNX or VxWorks, guarantee. As we see, response times of real-time threads in NetBSD 5.0 are similar to other real-time operating systems.

The following C code fragment illustrates how to use the *SCHED_FIFO* policy and set highest priority for the current (calling) thread:

```
struct sched_param sp;

memset(&sp, 0, sizeof(struct sched_param));
sp.sched_priority = sched_get_priority_max(SCHED_FIFO);

if (pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp)) {
        errx(EXIT_FAILURE, "pthread_setschedparam: %s", strerror(error));
}
```

See the *sched(3)* and *pthread_schedparam(3)* manual pages for a full description of scheduling functions.

Thread affinity

Thread affinity is the ability to bind threads to run only on a specified processor (or processors). This functionality is relevant to SMP systems and provides an effective way to increase *CPU utilization*, avoid CPU *cache thrashing*, ensure *concurrency*, and guarantee fast response times. These are of particular benefit to real-time applications.

Internally, the scheduler (this part of functionality is also called the *dispatcher*) tries to balance between two opposite tasks - utilize all processors and maintain thread affinity. In NetBSD 5.0, this is achieved by means of per-CPU run-queues, thread migration and balancing mechanisms. However, balancing and migration decisions are heuristic and based on observations of thread behaviour made by the kernel, meaning that the decisions made might not be optimal in all given situations. For example, the scheduler is not aware of the workload a thread is going to process, or of the relationships between threads. Because developers have such knowledge of their applications, interfaces to control affinity are provided. There are two different interfaces in NetBSD to implement thread affinity: **dynamic CPU sets** and **processor sets**.

Dynamic CPU sets is an interface similar to the CPU sets found in Linux and recent FreeBSD systems. It allows binding (pinning) a thread to the processors expressed via affinity mask (a simple bit-mask). Here is a C code fragment which creates a dynamic CPU set, adds the first processor (cpu0) to the set, and sets the affinity mask to the current (calling) thread:

```
cpuset_t *cset;
pthread_t pth;
cpuid_t ci;

cset = cpuset_create();
if (cset == NULL) {
        err(EXIT_FAILURE, "cpuset_create");
}

/* Set the first processor (cpu0). */
ci = 0;
cpuset_set(ci, cset);

/* Set affinity mask to the current thread. */
pth = pthread_self();
error = pthread_setaffinity_np(pth, cpuset_size(cset), cset);
if (error) {
        errx(EXIT_FAILURE, pthread_setaffinity_np: %s", strerror(error));
}

/*
 * At this point, the current thread runs on the first processor (cpu0).
 * The set can be destroyed now (or re-used for other thread, for example),
 * since it is already "applied" to the thread.
 */
cpuset_destroy(cset);

perform_work();
```

Wrappers to set and get the process affinity are also provided:

```
int sched_getaffinity_np(pid_t pid, size_t size, cpuset_t *cpuset);
int sched_setaffinity_np(pid_t pid, size_t size, cpuset_t *cpuset);
```

Note that while the thread is bound to the first processor, any other process/thread could also run on this processor, and share the time. A full description of dynamic CPU sets API could be found in the *affinity(3)* and *cpuset(3)* manuals. Unfortunately, neither POSIX nor other standards define any interfaces to control thread affinity and interfaces provided by various operating systems differ, so these calls are not expected to be portable.

Processor sets is an interface which allows assigning processors to threads. Assigned processors are forced to run only bound threads (or processes). Thus, processor sets are more of a resource pool based solution. A similar interface is found in the Solaris and HP-UX operating systems. Here is a C code fragment which forces a processor to run only the current (calling) process:

```
psetid_t psid;
cpuid_t ci;

if (pset_create(&psid) == -1) {
        err(EXIT_FAILURE, "pset_create");
}

/*
 * Assign cpu0 to the processor set.
 */
ci = 0;
if (pset_assign(psid, ci, NULL) == -1) {
        err(EXIT_FAILURE, "pset_assign");
}

/*
 * Bind the current process to the processor-set.
 */
if (pset_bind(psid, P_PID, P_MYID, NULL) == -1) {
        err(EXIT_FAILURE, "pset_bind");
}

/*
 * At this point, the first processor (cpu0) runs _only_ the current process.
 */
perform_work();

/*
 * Destroy the processor set.
 * This operation releases all assigned processors and bound threads.
 */
if (pset_destroy(psid) == -1) {
        err(EXIT_FAILURE, "pset_destroy");
}
```

Note that unlike the previous example with dynamic CPU sets, in this example the first processor (cpu0) would run only one bound process, and no other processes. A full description of the processor sets API can be found in the *pset(3)* manual page. This interface is not defined by any standards either. However, the API is expected to be nearly compatible with Solaris and HP-UX.

To find out how many processors are configured in the system, there is a standard option to *sysconf(3)*:

```
ncpu = sysconf(_SC_NPROCESSORS_CONF);
```

The flexibility to choose between two interfaces with different approaches, found in various UNIX-like operating systems, allows developers to use the interface which best suits the needs of their applications.

Controlling the scheduling of processes and threads

NetBSD also provides two utilities to control scheduling and thread affinity. These utilities can be used by administrators and developers. The *schedctl(8)* command allows changing scheduling priority and class, and setting thread/process affinity. The following examples illustrate some possible uses of this utility.

Show scheduling information about PID "123":

```
# schedctl -p 123
  LID:              1
  Priority:         43
  Class:            SCHED_OTHER
  Affinity (CPUs):  <none>
```

Set the affinity to CPU 0 and CPU 1, policy to SCHED_RR, and priority to 63 for thread whose ID is "1" in process whose ID is "123":

```
# schedctl -p 123 -t 1 -A 0,1 -C SCHED_RR -P 63
```

Run the *top(1)* command with real-time priority:

```
# schedctl -C SCHED_FIFO top
```

Traditional utilities like *top(1)* (with the -t option to enable thread-view) and *ps(1)* (with the -l option) can be used to monitor the general thread activity.

To control processor sets, *psrset(8)* is provided, which is nearly compatible with the one found in Solaris. The following example illustrates how to create a processor set, assign CPU 9 to it, and run *httpd* on that processor set.

Print current processor sets:

```
# psrset
system processor set 0: processor(s) 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Create a new processor set, which is assigned an ID of 1, and add CPU 9 to processor set 1:

```
# psrset -c
1

# psrset -a 1 9
```

Print current processor sets. Note that there is now a user-created processor set with an ID of 1, and that CPU 9 is now in processor set 1 and no longer in 0:

```
# psrset
system processor set 0: processor(s) 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15
user processor set 1: processor(s) 9
```

Execute *httpd* within processor set 1:

```
# psrset -e 1 /etc/rc.d/httpd start
```

Note that *top(1)* shows *httpd* running on CPU 9:

```
# top | grep httpd
29469 www       85    0   164K 1468K select/9    0:04  0.68%  0.68% httpd
15229 www       85    0   164K 1404K select/9    3:15  0.00%  0.00% httpd
18574 www       85    0   164K 1688K select/9    2:16  0.00%  0.00% httpd
14208 www       85    0   164K 1336K select/9    2:15  0.00%  0.00% httpd
```

A small *cpuctl(8)* utility is also available, which can change the state of CPU to online/offline, as well as identify its model and features. Example output from a machine with *"AMD Shanghai"* (codename) processors:

```
# cpuctl list
Num  HwId Unbound LWPs Interrupts     Last change
---- ---- ------------ -------------- ----------------------------
0    0    online       intr           Sat Nov  8 16:33:40 2008
1    1    online       intr           Sat Nov  8 16:33:40 2008
2    2    online       intr           Sat Nov  8 16:33:40 2008
3    3    online       intr           Sat Nov  8 16:33:40 2008
4    4    online       intr           Sat Nov  8 16:33:40 2008
5    5    online       intr           Sat Nov  8 16:33:40 2008
6    6    online       intr           Sat Nov  8 16:33:40 2008
7    7    online       intr           Sat Nov  8 16:33:40 2008
8    8    online       intr           Sat Nov  8 16:33:40 2008
9    9    online       intr           Sat Nov  8 16:33:40 2008
10   a    online       intr           Sat Nov  8 16:33:40 2008
11   b    online       intr           Sat Nov  8 16:33:40 2008
12   c    online       intr           Sat Nov  8 16:33:40 2008
13   d    online       intr           Sat Nov  8 16:33:40 2008
14   e    online       intr           Sat Nov  8 16:33:40 2008
15   f    online       intr           Sat Nov  8 16:33:40 2008

# cpuctl identify 0
cpu0: AMD Unknown AMD64 CPU (686-class), 2999.67 MHz, id 0x100f40
cpu0: features 178bfbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR>
cpu0: features 178bfbff<PGE,MCA,CMOV,PAT,PSE36,CFLUSH,MMX>
cpu0: features 178bfbff<FXSR,SSE,SSE2,HTT>
cpu0: features2 802009<SSE3,MONITOR,CX16,POPCNT>
cpu0: features3 efd3fbff<SCALL/RET,NOX,MXX,FFXSR,P1GB,RDTSCP,LONG,3DNOW2,3DNOW>
cpu0: features4 37ff<AHF,CMPLEGACY,SVM,EAPIC,ALTMOVCR0,LZCNT,SSE4A,MISALIGNSSE,3DNOWPREFETCH,OSVW,IBS,SKINIT,WDT>
cpu0: "AMD Engineering Sample"
cpu0: I-cache 64KB 64B/line 2-way, D-cache 64KB 64B/line 2-way
cpu0: L2 cache 1MB 64B/line 16-way
cpu0: ITLB 32 4KB entries fully associative, 16 4MB entries fully associative
cpu0: DTLB 48 4KB entries fully associative, 48 4MB entries fully associative
cpu0: L3 cache 2MB 64B/line direct-mapped
cpu0: Initial APIC ID 0
cpu0: AMD Power Management features: 0x1f9<TS,TTP,HTC,STC,100,HWP,TSC>
cpu0: family 0f model 04 extfamily 01 extmodel 00
```

This utility can also be useful to developers who are writing multithreaded applications and want to test them under a variety of CPU conditions.

Conclusion

NetBSD is a general-purpose operating system, with a scalable 1:1 threading implementation and flexible interfaces to control scheduling and threads. This scalable implementation allows it to provide high quality POSIX real-time extensions and reliably suit the needs of embedded systems. NetBSD also provides modern solutions to ensure good performance for threaded workloads, which are increasingly prevalent in today's world of multi-core and multi-processor systems.

http://www.NetBSD.org/