An Implementation of Scheduler Activations on the NetBSD Operating System
by
Nathan J. Williams

Presented By: Udaykumar Batchu
A Brief Introduction

• Importance of “Thread Programming” in application development.

• Two major Thread Implementations currently available in Unix like systems are:
  ✓ User-level Threads.
  ✓ Kernel-level Threads.

• Design of a two-level thread system known as *Scheduler Activations*. 
Thread Systems: In-detail

- **User-level Threads**
  - Purely implemented at user-level.
  - Kernel is unaware of threads.
  - Can not make use of multiple CPU’s.
  - Also called as “N:1” thread system.

- **Kernel Threads**
  - OS Kernel is aware of the threaded nature.
  - Kernel is responsible for everything.
  - Multiprocessor parallelism can be exploited.
  - Also called as “1:1” thread system.
Advantages and Disadvantages

• Kernel threads provide better concurrency, but are more expensive in time and space.
• Relatively easy to implement, given OS support for kernel execution entities.
• E.g.; Linux, IRIX, Windows NT.
• User-level threads can be implemented without kernel support.
• E.g.; GNU PTH Thread Library, FSU Threads.
• Conclusion: Both are in one way or the other not suitable. So a combination of both of these known as “N:M” systems is derived to achieve the result.
**N:M THREAD SYSTEM**

- Maps some no. N of application threads onto a no. M of kernel entities (usually N>M).
- Also known as “two-level” thread systems.
- N:M thread systems are more complicated and effective.
- Implemented by associating groups of threads with single kernel entities.
- Both AIX and Solaris use N:M thread system.
- The best implementation is, one put forward by Anderson et al. called: *scheduler activations* model.
Scheduler Activations (SA)

- Provides the application with a set of virtual processors.
- This has many advantages such as:
  - kernel resource usage is kept small.
  - Voluntary thread switching is cheap.
  - Application concurrency is fully-maintained.
- The SA kernel communicates with the application about the happenings with a mechanism known as: *upcall.*
SA contd…

- All the events are communicated by an upcall, like a change in the size of the virtual processor set.
- Performing an upcall for allocation of a new Virtual Processor is done by: upcall handler.
- It is enabled with performing any user-level thread bookkeeping.
- The upcall handler decides which thread to run next, when a preemption occurs.
Kernel Interface

The application interface to SA consists of: system calls.

• `sa_register()`: kernel entry point usage.
• `sa_setconcurrency()`: level of concurrency available.
• `sa_enable()`: starts the system by invoking an upcall.
• `sa_preempt()` and `sa_yield()`: These do the management of processors.
Upcalls

- **Def**: An interface used by the SA for communication.
- The *upcall* procedure is called when an event occurs.
- The signature of an upcall is:

```c
void sa_upcall (int type,
                struct sa_t *sas[],
                int events,
                int interrupted,
                void *arg);
```
contd...

- **type**: the event which triggered the upcall.
- **sas**: points to an array of `struct sa_t`, the activations involved in the event.
- **events**: activations that are directly involved in the event. `(sas[1] – sas[events])`
- **interrupted**: the activations that are stopped for delivering an upcall.
  
  `(sas[events+1] – sas[events+interrupted])`
The role of **Stacks**

- Upcalls must be allocated their own set of stacks.
- Needs a STACK for the storage of it’s local variables, return addresses and so on.
- This can be done by: `sa_stacks()`.  
- The application has to keep track of all the calls.  
- Periodical calling of `sa_stacks()` is needed to recycle stacks.
The role of **Signals**

- The method of handling signals is similar to: POSIX signal model.
- Signals are handed to the application via the *upcall*.
- The `struct_siginfo_t` structure is used, with *arg* parameter pointing to it.
Kernel Implementation

• The kernel implementation involves two things:

  ✓ Separation of traditional process context from execution context.

  ✓ The separation of machine-dependent and machine-independent code.
LWP (light-weight process)

• The first step involved:

  ➢ Relocating the parts of classic BSD struct proc to a new structure, struct lwp.

  ➢ Conversion process replaced variables: struct proc *p with struct lwp *l.

  ➢ Scheduler was converted to handle scheduling LWPs rather than processes.

  ➢ Also, the existence and concurrent execution of several LWPs within a single process.

  ➢ Signal delivery and process exit were significantly affected.
Original Net BSD Process States
New NetBSD Process states and LWP States

- SIDL
- SACTIVE
- SDEAD
- SZOMB
- SSTOP
- LSIDL
- LSDEAD
- LSZOMB
- LSSUSP
- LSRUN
- LSONPROC
- LSSLEEP
- LSSTOP
contd...

• The second step involved:

  ➢ Porting of machine-dependent parts of NetBSD to work with LWPs.

  ➢ Also the splitting of parts of old struct proc into new struct proc and struct lwp.

  ➢ A change to the cpu_switch( ) function, by using a new routine called cpu_preempt( ).
Scheduler Activations (restated)

When an execution context is blocked by a call to \texttt{tsleep()} function, the change here is:

- A call to \texttt{sa\_switch()} instead of \texttt{mi\_switch()}.

- The “preempted” upcall should also to be taken care.
Thread Implementation

- Support user-level concurrency.
- Spin locks are to be handled carefully, as critical sections are protected with spin locks.
- The thread implementation also has a machine dependent component.
Performance Analysis

HBench-OS test

The test is done on a: 500MHz Digital Alpha 21164 System.

<table>
<thead>
<tr>
<th></th>
<th>before SA</th>
<th>after SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid</td>
<td>0.631</td>
<td>0.601</td>
</tr>
<tr>
<td>getrusage</td>
<td>4.053</td>
<td>4.332</td>
</tr>
<tr>
<td>timeofday</td>
<td>1.627</td>
<td>1.911</td>
</tr>
<tr>
<td>sbrk</td>
<td>0.722</td>
<td>0.687</td>
</tr>
<tr>
<td>sigaction</td>
<td>1.345</td>
<td>1.315</td>
</tr>
</tbody>
</table>

The idea of the test is to: explore the performance of the system. Five of the tests measure system call latency.
Measurement of Thread Operation costs:
Done on: Apple iBook 500MHz G3 processor with 256k of L2 cache.

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>PTH</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>12us</td>
<td>96us</td>
<td>90us</td>
</tr>
<tr>
<td>Mutex</td>
<td>0.4us</td>
<td>0.3us</td>
<td>0.6us</td>
</tr>
<tr>
<td>Context</td>
<td>225us</td>
<td>166us</td>
<td>82us</td>
</tr>
</tbody>
</table>
Conclusions and future work

- Successfully presented the design and implementation of a two-level thread scheduling system based on scheduler activations.

- The integration with the main BSD source tree will be a good work and as always performance tuning.