1 – Threads, Kernels and Scheduling

List of Slides

1  Threads, Kernels and Scheduling
4  Operating Systems Taxonomy
5  Some Tradeoffs
6  Structure of Monitor like OS
7  Process State in Monolithic Kernels
8  Multitasking Using Processes
9  Granularity of Parallelism
10 Cooperative Multitasking Using Threads
11 Library Support of Threads
12 Uses of threads
13 Advantages of Threading
14 User Vs. Kernel level Thread Support
15 User vs. Kernel Level Threads
16 Solaris Kernel level Thread Support
17 Contents of a Thread
18 Library Support Revisited
19 Solaris Thread Support
20 System Call Semantics
21 Signal Semantics 1 of 2
22 Signal Semantics 1 of 2
23 POSIX Threads Life Cycle
24 Mapping Threads to LWPs
25 Thread Scheduling
26 Thread Context Switch Causes
27 Thread States
28 Thread Suspension
29 Pthread Synchronization
30 Pthread Mutex Usage
32 POSIX Semaphores
34 POSIX Condition Variables
36 POSIX threads Compiling and Linking
37 Simultaneous Multithreading
38 Multiprocessor Scheduling
39 Gang Scheduling
Operating systems are frequently described according to what features are maintained in the kernel. Some categories include:

1. Monolithic Kernel — High level features encapsulated in the kernel. E.g. U*ix, Linux, VMS, MVS, VM
2. Microkernel — High level features implemented as privileged processes outside the kernel. E.g. Mach, L4, QNX
3. Exokernel — Kernel only exists to multiplex hardware all all other features in user space libraries. E.g. MIT exopc and exos.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Exokernel</th>
<th>Microkernel</th>
<th>Monolithic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personality</td>
<td>Kernel</td>
<td>Server Proc.</td>
<td>Kernel</td>
</tr>
<tr>
<td>Kernel Crossings</td>
<td>Few</td>
<td>Many</td>
<td>Large</td>
</tr>
<tr>
<td>Process State</td>
<td>Small</td>
<td>Kernel</td>
<td>Kernel</td>
</tr>
<tr>
<td>Protection</td>
<td>Libos</td>
<td>Libos</td>
<td>Libos &amp; kernel</td>
</tr>
<tr>
<td>VM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context Switch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4 – Structure of Monitor like OS

DOS like systems provide very basic single user support.

![Diagram of OS structure]
In traditional monolithic kernels, the process state increased in size due to features being moved inside the kernel.
Multitasking using processes involves maintaining separate state for each task.
7 – Granularity of Parallelism

Stallings [2] describes parallelism according how high a synchronization cost (in instructions) can be tolerated:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Synch. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine</td>
<td>Inherent in the instruction stream</td>
<td>&lt; 20</td>
</tr>
<tr>
<td>Medium</td>
<td>Shared Memory (threads)</td>
<td>20 – 200</td>
</tr>
<tr>
<td>Coarse</td>
<td>Shared Memory (processes)</td>
<td>200 – 2000</td>
</tr>
<tr>
<td>Very Coarse</td>
<td>Distributed Processes (Network)</td>
<td>2000 – 10^6</td>
</tr>
<tr>
<td>Independent</td>
<td>Unrelated processes</td>
<td>∞</td>
</tr>
</tbody>
</table>
8 – Cooperative Multitasking Using Threads

Threads share large amounts of common state information (although they have distinct stacks, priorities, program counters, and registers) and having greatly reduced context switching overhead.
9 – Library Support of Threads

Libraries (such as the POSIX pthreads library) support multithreading of applications.
10 – Uses of threads

Lewis and Berg [?, ?] list some inherently multithreaded (MT) programs:

1. Independent Tasks
2. Servers
3. Repetitive Tasks

And some not so obvious examples:

1. Numerical Programs
2. Tuning Existing Code

![Diagram of multithreaded processes]
11 – Advantages of Threading

The following “wins” can be obtained through multithreading an application:

1. Better performance via Parallelism (SMP architectures)
2. Increased application throughput.
3. More Responsive Applications
4. Reduced interprocess communication
5. Efficient Resource Utilization
6. Simplified realtime processing
7. Simplified Signal Handling
8. Ease of exploiting concurrency in distributed objects
9. Improved software structure
10. A single source ports to multiple platforms
12 – User Vs. Kernel level Thread Support

Some operating systems have user level threads (ULT). ULTs have thread scheduling and control implemented as a user level library which runs inside the user process (e.g. FreeBSD 4.0).

Others (e.g. Linux, Solaris, SGI Irix) support kernel level threads (KLT).

Is one more powerful than the other?

What are the relative costs?
13 – User vs. Kernel Level Threads

1. User Level Thread Advantages – Since Scheduling is done at the application layer:
   (a) ULT Scheduling decisions can be application specific.
   (b) ULTs do not need kernel level support, and have libraries which tend to be easy to port.

2. Kernel Level Thread Advantages — Since scheduling is done by the OS
   (a) Systems calls that stop a thread stop only that thread and not the entire process.
   (b) If additional physical processors exist, the kernel can schedule threads there (which cannot happen using ULTs).
14 – Solaris Kernel level Thread Support

In Solaris, lightweight processes serve as virtual processors for executing threads. A process needs to keep track of its LWPs.

Traditional UNIX Process Structure

<table>
<thead>
<tr>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>UID GID EUID EGID CWD...</td>
</tr>
<tr>
<td>Signal Dispatch Table</td>
</tr>
<tr>
<td>Memory Map</td>
</tr>
<tr>
<td>File Descriptors</td>
</tr>
<tr>
<td>Priority</td>
</tr>
<tr>
<td>Signal Mask</td>
</tr>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>Kernel Stack</td>
</tr>
<tr>
<td>CPU State</td>
</tr>
</tbody>
</table>

Solaris 2 Process Structure

<table>
<thead>
<tr>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>UID GID EUID EGID CWD...</td>
</tr>
<tr>
<td>Signal Dispatch Table</td>
</tr>
<tr>
<td>Memory Map</td>
</tr>
<tr>
<td>File Descriptors</td>
</tr>
</tbody>
</table>

LWP 2

| LWP ID |
| Priority |
| Signal Mask |
| Registers |
| Kernel Stack |
| ... |

LWP 1

| LWP ID |
| Priority |
| Signal Mask |
| Registers |
| Kernel Stack |
| ... |
15 – Contents of a Thread

Threads have their own stack, identifier, priority, signal mask and registers (in POSIX).

```assembly
#PROLOGUE
save %sp,%rp %sp
sethi %HLID137,%rp0
sethi %HLVAR_REG1,%rp1
id [%co+0x0(LID137)].
sethi %HLID134,%rp2
sethi %HL(v.16),%rp0
ld [%co+0x0(LID134)].
or %rp0,%lo(v.16),%rp0
call _s_wide 1
spilld [%co+0x0(LP1)]
sethi %HLID126,%rp1
or %rp0,%lo(v.16),%rp0
call _s_wide 1
spilld [%co+0x0(LP1)]
sethi %HLID126,%rp1
or %rp0,%lo(LID126),%rp1
```

--- Stack Frame ---
Return Address
Input Arguments:
"The Cat in the hat"
Local variables
"F"
3.14159265358969
Stack Frame Pointer

--- Stack Frame ---
Return Address
Input Arguments:
"came back!"
Local variables
"Rural Route 2"
1.414

--- Stack Frame ---
Return Address

--- Stack Frame ---
Return Address

Thread Structure

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Priority</th>
<th>Signal Mask</th>
<th>CPU Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Stack Pointer

Program Counter

Code (not part of the thread)
The threads library maintains internal threads structures and maps threads onto LWPs.
The Solaris kernel transparently maps LWP’s on to physical processors, and (in conjunction with the library) threads on to LWPs.

**Solaris Multithreaded Model**

![Diagram of Solaris Multithreaded Model]
A system call does the following:

1. The process traps to the kernel.
2. The trap handler runs in kernel mode, preserving registers.
3. It sets the stack pointer to the kernel stack of the process.
4. The kernel runs the system call.
5. The kernel copies results into the user space.
6. The kernel updates process structures.
7. The process reverts to user mode.
Unix uses signals for asynchronous programming. Prior to a program receiving a signal it will:

1. call `sigaction()` to specify a signal handler.
2. call `sigprocmask()` to enable/disable a particular signal.

afterwards (3) the program will be ready to receive signals and run the job, then:

4. The signal (e.g `SIGUSR1`) can be received from some other program interrupting the program.
5. The handler is invoked from the interrupt vector.
6. The handler either returns (and the program continues) or terminates the program.
So a signal could have the states:
21 – POSIX Threads Life Cycle

In POSIX threads (pthreads), the following calls are provided for thread creation, deletion, and waiting for termination:

1. `pthread_create` — starts a thread
2. `pthread_exit` — terminates a thread
3. `pthread_join` — waits for child thread completion

If you detach a thread, the parent is not expected to wait for it (and the child might survive the parent’s death).
The mapping of threads to LWPs is central to scheduling. The following alternatives are used:

1. Many threads on one LWP (many to one).
2. One thread per LWP (one to one).
3. Many threads to many LWPs (strict, i.e. one to one not allowed)
4. Many to many (a user may request one to one for a thread).
23 – Thread Scheduling

The kind of scheduling induced follows from the mapping model:

1. Process Contention Scope or Unbound Threads — many to many model

2. System Global Scheduling, System Contention Scope or Bound Threads — one to one model.

POSIX calls these the scheduling contention scope, and supports both. Win32 supports only global scheduling.
24 – Thread Context Switch Causes

In Process Contention Scope (many to many) context switching can happen due to:

1. Synchronization — the most common cause
2. Preemption — higher priority thread arrives
3. Yielding — voluntary yield to other threads of same priority.
4. Time Slicing — times out (if vendor supported).
25 – Thread States

A thread is in one of the following states:

1. Active — it is on an LWP
2. Runnable — ready but needing an LWP
3. Sleeping — Blocked on a synchronization variable.
4. Stopped — (NOT in POSIX) asynchronously suspended by another thread.
5. Zombie — A dead thread pending resource reclamation.
26 – Thread Suspension

Dave Butenhof (POSIX standards group) states that thread suspension is dangerous since threads can be suspended in systems calls and the machine state is very uncertain. Relatively few applications need thread suspension.

Hans Boehm (and others) have noted that some software (such as Garbage collectors) requires one thread to pause another’s execution. (Thread debuggers are another example).

Win32 supports suspension (with associated risks) while POSIX does not (with some loss of functionality).

Moral of the story: if you use/need thread suspension be VERY CAREFUL!
Pthreads has the following (simple) synchronization variables:

1. Mutexes

2. Semaphores (in the real-time standard, not threads)

3. Condition Variables.

In some implementations, it is possible to use thread variables to synchronize across processes.
A mutex provides single threaded access to a resource. It is considered a programming error if a thread unlocking a mutex is not the locking thread.

1. `int pthread_mutex_init(&prt_lock, &mattr)`

2. `int pthread_mutex_lock(m)`

3. `int pthread_mutex_unlock(m)`

Where `pthread_mutex_t m;` and `pthread_mutexattr_t mattr;`.
POSIX semaphores are part of the real time standard and may not be present on many architectures (e.g. FreeBSD). Their syntax looks like:

1. `int sem_init(sem_t *sem, isshared, value)`
2. `int sem_destroy(sem_t *sem)`
3. `sem_t *sem_open(char *name, int o_flag, ...)`
4. `int sem_close(sem_t *sem);`
5. `sem_unlink(const char *name);`
6. `int sem_wait(sem_t *sem);`
7. `int sem_trywait(sem_t *sem);`
8. `int sem_post(sem_t *sem);`
9. `int sem_getvalue(sem_t *sem, int *sval);`
POSIX condition variables are used to synchronize on predicate which is shared between threads. Be sure to guard evaluation and update of the predicate using a mutex.

1. `int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *pshared);`
2. `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
3. `int pthread_cond_destroy(pthread_cond_t *cond);`
4. `int pthread_cond_broadcast(pthread_cond_t *cond);`
5. `int pthread_cond_signal(pthread_cond_t *cond);`
6. `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
7. `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct
timespec *abstime);
Compiling and linking programs using pthreads varies across architectures, however the following seem to be common place.

1. Use compiler flag \texttt{-D\_REENTRANT} or set macro. (to compile in MT safe prototypes).

2. Link with a threads library (on FreeBSD \texttt{-lcr}, on other U*ixes \texttt{-lthread}).
32 – Simultaneous Multithreading

Simultaneous Multithreaded Architectures (SMT) use hardware thread scheduling on a single processor [?, 1].

- Problem: Hardware misses stall pipelined processors too long.
- Solution: So do a hardware context switch (bank switch registers).
- Challenges: Typically requires rewrite of software to be multithreaded (hard).
- Approach: Chappell and Stark’s [?] Simultaneous Subordinate Multithreading (SSMT) uses hardware (microcode) to implicitly generate multiple threads (is limited).
33 – Multiprocessor Scheduling

Several scheduling approaches have been used:

- Load Sharing — Send available work to first idle processor
- Dedicated Processor Assignment — Each task runs on an assigned processor from start to finish.
- Gang Scheduling — Schedule all cooperating threads to have the same time slice (its all or nothing).
- Dynamic Scheduling — Allow programs to vary their number of threads during execution
- Processor Affinity Scheduling — Schedule a well behaved thread on the same processor, banish poorly behaved threads to some other processor (e.g. for cache reasons).
Gang scheduling is ubiquitous for high performance computing, and may be necessary for some real time applications:

- Useful if the task has the structure:

  ```c
  for (i = 1; i < NUMBER_OF_STEPS; ++i) {
      for (j = 1; j < NUMBER_OF_PARALLEL_TASKS; ++j) {
          do 1/j part of the task in parallel
      }
      synchronize all parallel tasks
      any sequential stuff to complet the ith step
  }
  ```

  ] because waiting for synchronization of blocked jobs is reduced.

- Reduces how often the schedule has to be recomputed.

- To ensure fairness, weighting by number of processors required can be done. E.g.

  Consider a machine with 4 processors and 3 jobs, 2 needing 4 processors and 1 needing 1
processor.

Give each task needing 4 processors weight of $4/(1 + 4 + 4) = 4/9$ access to the processor and the remaining jobs $1/9$ of the time.
References
