Threads

• Suspending a process
  – suspends all threads of the process since all threads share the same address space
• Termination of a process
  – terminates all threads within the process

Thread States

• States of a thread
  – Spawn
    • when process is spawned
    • thread may spawn other threads
    • each thread has its own:
      – register context, state space, and place in ready queue
  – Block
    • when thread waits for event
      – saves user registers, PC and stack pointer
Thread States

- States of a thread
  - Unblock
    - when blocking event occurs
    - thread is moved to ready queue
  - Finish
    - register context and stack is deallocated

Remote Procedure Call Using Single Thread

What is a RPC?

(a) RPC Using Single Thread
Remote Procedure Call Using Threads

(b) RPC Using One Thread per Server (on a uniprocessor)

Figure 4.3 Remote Procedure Call (RPC) Using Threads

Multithreading

Figure 4.4 Multithreading Example on a Uniprocessor
Basic questions

• What is the difference between this and multiprocessing?
  – kind of looks the same, or...?

• Is there a need to synchronize threads?
  – e.g. two threads insert an element into a linked structure

User-Level Threads (ULT)

• All thread management is done by the application
  – e.g. using threads library
• The kernel is not aware of the existence of threads
User-Level Threads

Thread 2 makes call to I/O that blocks

Time slice expired and other process is executed. Thread 2 is still "running"

Thread 2 is only perceived as running by the thread library

Thread 2 needs action from thread 1 and blocks so thread 1 can execute

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States
Kernel-Level Threads (KLT)

• Often called *lightweight processes*
• Windows is an example of this approach
• Kernel maintains context information for the process and the threads
• Scheduling is done on a thread basis
VAX Running UNIX-Like Operating System

Table 4.1  Thread and Process Operation Latencies (μs) [ANDE92]

<table>
<thead>
<tr>
<th>Operation</th>
<th>User-Level Threads</th>
<th>Kernel-Level Threads</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Fork</td>
<td>34</td>
<td>948</td>
<td>13,560</td>
</tr>
<tr>
<td>Signal Wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

Combined Approaches

- Thread creation is done in user space
- Bulk of scheduling and synchronization of threads done within application
- Example is Solaris
Combined Approaches

Relationship Between Threads and Processes

<table>
<thead>
<tr>
<th>Threads/Processes</th>
<th>Description</th>
<th>Example Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>Each thread of execution is a unique process with its own address space and resources.</td>
<td>Traditional UNIX implementations</td>
</tr>
<tr>
<td>M:1</td>
<td>A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.</td>
<td>Windows NT, Solaris, Linux OS/2, OS/390, MACH</td>
</tr>
<tr>
<td>1:M</td>
<td>A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems</td>
<td>Ke (Clouds), Emerald</td>
</tr>
<tr>
<td>M:N</td>
<td>Combines attributes of M:1 and 1:M cases.</td>
<td>TRIX</td>
</tr>
</tbody>
</table>
Advantages of ULT over KLT

- thread switching does not require kernel mode privileges
  - saves two mode switches (user-to-kernel and kernel-to-user)
- application specific scheduling
  - applications may prefer their own specific scheduling algorithm
- ULT can run on any OS

Disadvant. of ULT vs KLT

- Many OS system calls are blocking.
  - so if ULT executes such call all threads within its process are blocked
- In pure ULT strategy a multithreaded application cannot take advantage of multiprocessing
  - no concurrency