Scheduler Activation:

Slide 2:

We know that threads play a critical role for concurrency programming, and it can be supported by user-level or kenel threads. However, each approaches has its own advandtages and disadvantages.

The user level threads offer great performance. It is managed by runtime libraries within applications, they operate without kernel intervention. They also have high flexibility, allowing customization to specific user or language requirements. But it can lead to poor performance and even incorrect behaviors during I/O activities and page faults.

In contrast, kernel-level threads have worse performance than the user level threads, but it doesn’t have such restrictions as user-level threads.

Slide3

Specifically,

— In the common case when thread operations do not need kernel

intervention, performance is same as user-level management sys.

— In the infrequent case when the kernel must be involved, such as on

processor reallocation or I/O, our system can mimic the behavior of a

kernel thread management system:

— No processor idles in the presence of ready threads.

— When a thread traps to the kernel to block (for example, because of a

page fault), the processor on which the thread was running can be

used to run another thread from the same or from a different address

space.

Slide5.

The vir. Multiprocessor is a bridge between the kernel and user-level threadd system.

The kernel has complete control of allocating the processors of each address space, while the user level system controls which thread to run on the allocated processors.

When the kernel has to change the number of processors, it notifies the user-level thread system. On the other hand, the user-level thread system notifies the kernel when its application needs more or few processors.

Slide 6.

The mechanism of the kernel notifies the user-level thread system is called “scheduler activation”. It sends a notification to the user-lvl thread at certain points. It also provides space in kernel to save context of user threads when kernel stops it.

Slide 7.

Slide 8

This example illustrate how schedular activation works when IO req/complete happens

T1, kernel upcall, remove threads from rdy list

T2, IO start, Thread1 block, kernel upcall, new scheduled act. Created, run thread3

T3, IO end, kernel notify and preempted thread2, scheduled act. A B discarded

T4, kernel upcall, take thread off the ready queue

Slide 10:

All of the above features were implemented by modifying the Topaz OS on the DEC SRC Firefly multiprocessor workstation. This device has 6 cores—enough to compare performance between the new system and both user and kernel-level threads, though some effects may not be clear without even more cores to work with. The modifications required took about ¼ as much code as the original OS, most of which was related to the kernel’s allocation of processors to processes. The scheduling policy used was that all equal-priority processes are given an even amount of cores, with timesharing only used to divide the last couple cores if the numbers didn’t divide evenly.

Slide 11:

A few specific optimizations were made to improve the performance of the new system. For one, cores that idle in one process are allowed to spin for a time before that core is reallocated. This is to prevent cores from being taken away over something small like a single I/O. Another optimization is to reuse old activations’ allocated structures to create new activations. This avoids kernel overhead in allocating new memory.

Finally, a new scheme was developed for handling cases where the user thread is interrupted during a critical section. The kernel keeps a copy of each critical section’s code with a “return to activation” added at the end of each. This copy is run instead of the original when an activation finishes up a critical section for a thread that was interrupted. By doing this, we avoid any overhead in the more common case where no interruption occurs.

Slide 12:

The new system was found to be nearly as fast as old user-level threads in cases where the application fits in memory. It is thus an order of magnitude faster than kernel threads in this case. The small slowdown was due to some additional bookkeeping being done in the new system.

Slide 13:

Kernel operations were found to be around 5x slower than normal. This is concerning since these operations can be limiting in many applications. However, the authors believe that they could improve their implementation substantially, possibly to match the standard performance. The present implementation maintains a more complex state than is theoretically necessary and hasn’t been optimized on the assembly level with as much care as the original Topaz system.

Slide 14:

When tested on a practical application that fits in memory, the new system was faster than either user or kernel-level threads alone. Kernel-level threads were particularly slow due to the overhead of visiting the kernel more often than necessary. User-level threads were also a bit slower due to previously mentioned inefficiencies.

Slide 15:

The new system was also faster on a practical application that doesn’t fit in memory. User-level threads were particularly slow in this case because they often wait unnecessarily for another user-level thread that has been blocked for I/O.

Slide 16:

Some similar implementations have been created before. For example, asynchronous I/O is a paradigm that allows user-level threads to avoid blocking unnecessarily by not blocking for I/O at all. However, this requires the programmer to use a different interface, which makes development unnecessarily difficult.

Other OS’s have been implemented that allow some level of communication between user and kernel like what is described here. Psyche and Symunix are two examples. However, such implementations have generally failed to implement the degree of optimizations necessary to make this paradigm worthwhile.

Slide 17:

To Conclude:

User-level threads offer flexibility and low-overhead to the developer, but have issues such as unnecessary blocking that limit their performance in many scenarios.

Kernel-level threads offer the generality desired for more optimized parallel computations, but suffer from significant overhead for many operations.

The new activation-based threading system combines the best of both worlds by allowing processes and the kernel to communicate useful information. This is used to allow processes to dictate their own optimal processor usage while minimizing time spent in wasteful kernel operations. As a result, the new system was found to be significantly more efficient in a variety of applications.

Mesa:

Slide 2.

The use of monitors has been discussed in the previous papers. However, the number of problems still exists and are addressed in this paper.

With the high demand of concurrency used in the program, the author wanted to develop a model for controlling concurrency using monitor for:

* Local concurrent programming. An application can be implemented as a group of synchronized processes.
* Global resource sharing: Independent applications share the processor.
* Replace interrupts: Without using interrupt, the request for a software attention to a device can be replaced by a wakeup mechanism.

Slide 3.

There had been several problems with monitor addressed in this paper. Including.

Program structure: Mesa organizing programs into modules, how to design?

=> Monitor module and condition variables

Dynamic process creation:

=> Just prefix a function with 'FORK' to create a new process

Dynamic allocation of monitors:

=> Create an object -> create a monitor -> associate the monitor with the object

The consequence of wait call in a nested monitor call is not clear

=> Deadlock

Exceptions: How should the system handle exceptions when monitors are used?

Exception handler should release the mutex first and then handle the exception

Scheduling: How should monitors interact with process scheduling?

Priority inversion can happen. Suggestion: priority boosting

Input-output: Can and should I/O devices be treated like any other signalling in a system that provides monitors?

Naked notify

Slide 4.

The paper also purposed two other ways rather than using monitors:

1)Msg passing

2)Shared mem

But rejected each of it because:

1. Will need more effort compared to the monitor approach
2. It would not work on multi processors, and a separate mutex mechanism is needed.

Slide 5.

Processes in Mesa can be dynamically created using the prefix by ‘’fork’

Processes are treated like any other value, they can be passed as arguments or assigned to variables.

Slide 6.

The shared data is protected by the monitor, there are two methods to access the data:

* Entry procedures: Processes can only perform operation for data from calling it.
* Internal procedures : Can only call from monitor procedures.

Only one process can be inside monitor and access the shared data.

If a random order of calling entry procedures is not acceptable, other provisions must be made in the program outside the monitor.

Slide 7.

The monitor module has three kinds of procedure: entry, internal, and external. The first two are the monitor procedures, and execute with a monitor lock held.

Slide 9.

If a procedure P1 called another P2, P2 call P3 … until Pn. If Pn generate a exception that can only be handled by P1.

P1 abandoned computation of P2…Pn, and continew exec in P1.

When this happens, an UNWIND exception is generated for giving a chance for P2…Pn to release the monitor lock if needed.

Slide 10:

In Mesa, as opposed to Hoare semantics, condition variables are only used to give “hints” to when threads should be woken up. Each thread independently verifies that its desired condition holds each time it wakes up, so it is ok to wake them up spuriously on occasion. This means that we can often use simplistic checks to decide when to do wakeups; the threads themselves will make the final decision as to whether it is actually a correct time for them to proceed.

Slide 11:

There are a few ways besides normal notifications to wake up waiting threads. On some systems, a timeout is established that wakes each thread up if it hasn’t been notified after a certain period. This can be useful if we want a thread to “give up” if a notification isn’t happening in a reasonable period. Another option is for one process to send an abort signal to another. This causes the aborted thread to wake up and abort, giving us the ability to clean up waiting threads if we choose. Finally, there is the option to broadcast a notification to all threads waiting on a certain condition. This can be useful if more than one thread may be able to proceed when a condition is met.

Slide 12:

Monitors can be used by external devices to wake up threads that handle their interactions with the processor. The issue with this is that we don’t want a potentially slow external device holding a monitor lock that is needed frequently by actual processors. Instead, we use the “wakeup-waiting switch” trick by writing to a fixed location in memory to perform a notification. This avoids race conditions without needing to acquire the monitor lock.

Slide 13:

A common issue with synchronization tools (such as monitors) used alongside a priority scheduler is priority inversion. This occurs when a high-priority process waits for a low-priority process that is then preempted by a mid-priority process. The high-priority process is then effectively waiting for the mid-priority process. This can be fixed by the familiar method of priority donation, allowing the monitor to run with the priority of the highest priority thread that is waiting and not just that of the thread that is actually running.

Slide 14:

In Mesa, a process is represented by its call stack and its state. The state determines whether it is currently running or waiting on a queue (read, monitor lock, condition variable, or fault). Each queue is sorted by the process’s priorities so that every structure respects those priorities.

Slide 15:

Monitors are used as part of the life cycle of processes. A “rebirth” condition variable is kept so that processes can wait on it when they are ended. New processes are created by modifying the information in one of the available process slots and the notifying that process’ rebirth. Abort and Yield functions are also implemented so that processes can control their use of processor time.

Slide 16:

One downside of monitors is that they have overhead to maintain their states. However, we generally observer that this overhead is small compared to most operations they are used to control.

[go over data]

Slide 17:

Pilot is an OS that was developed for personal computers in Mesa. It uses ~40 monitors and 15 default processes to coordinate its function. Previously mentioned features such as naked notifies and fault queues are used in various systems, and processes wake up handlers using notifications. Due to the complexity of the system’s synchronization, numerous deadlocks have surfaced over its use. However, none have proven difficult to fix with the monitor paradigm. There is also the issue that signals such as “unwind” must be handled by monitors, which can incur overhead and be difficult to implement.

Slide 18:

Violet is a distributed database manager with a particular application to calendars. It uses a hierarchy of processes that make calls to lower processes to perform system functions. Monitors are used to coordinate the processes and alert them when data has changed (so work needs to be done).

Slide 19:

Gateway is an interface for passing packets between two or more existing networks. It makes use of large-scale parallelism with numerous processes for device drivers, sockets, a dispatcher, and many other abstractions. These processes use monitors to notify each other as part of the pathways used in each operation.

Slide 20:

To Conclude:

Satisfactory monitor implementations are difficult to make because of the number of edge cases and applications that need to be considered in their design. Mesa monitors have succeeded in this respect, finding use in a variety of complex and large-scale systems as coordinators between processes. Though bugs have inevitable been found, the paradigm has proven robust enough to resolve them all handily.