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

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.