Operating System Support for a Parallel DBMS with an Hierarchical Shared-Nothing Architecture

Leonid B. Sokolinsky
Chelyabinsk State University
Russian Fed.
sokolinsky@acm.org

ADBIS'99

This work was supported by the Russian Foundation for Basic Research under Grant 97-07-90148

The subject of my presentation is the Operating System Support for a Parallel DBMS with an Hierarchical Shared-Nothing Architecture.

The prototype of a parallel DBMS for Russian MBC supercomputer was designed in Chelyabinsk State University. This prototype was called Omega system.

Contents

- Omega hardware architecture
- Omega operating system support
- Omega operating system structure
- A model for thread scheduling
- Thread manager implementation
- Conclusion

ADBIS'99 L.B. Sokolinsky

During my presentation I concern the following subjects

Omega hardware architecture
Omega operating system support
Omega operating system structure
A model for thread scheduling
Thread manager implementation

And then I make a short conclusion .

Contents

- **Omega hardware architecture**
- Omega operating system support
- Omega operating system structure
- A model for thread scheduling
- Thread manager implementation
- Conclusion

ADBIS'99 L.B. Sokolinsky

The first topic is “Omega hardware architecture”

Three levels of the hierarchy

- 1st level: Processor modules
- 2nd level: Omega clusters
- 3rd level: Omega system

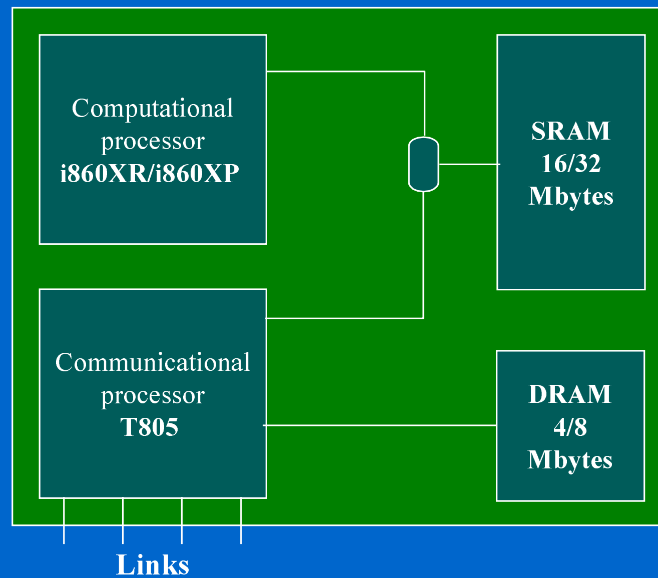
ADBIS'99 L.B. Sokolinsky

Omega hardware architecture has a three levels of hierarchy:

The 1st level is presented by processor modules

The 2nd level is presented by Omega clusters

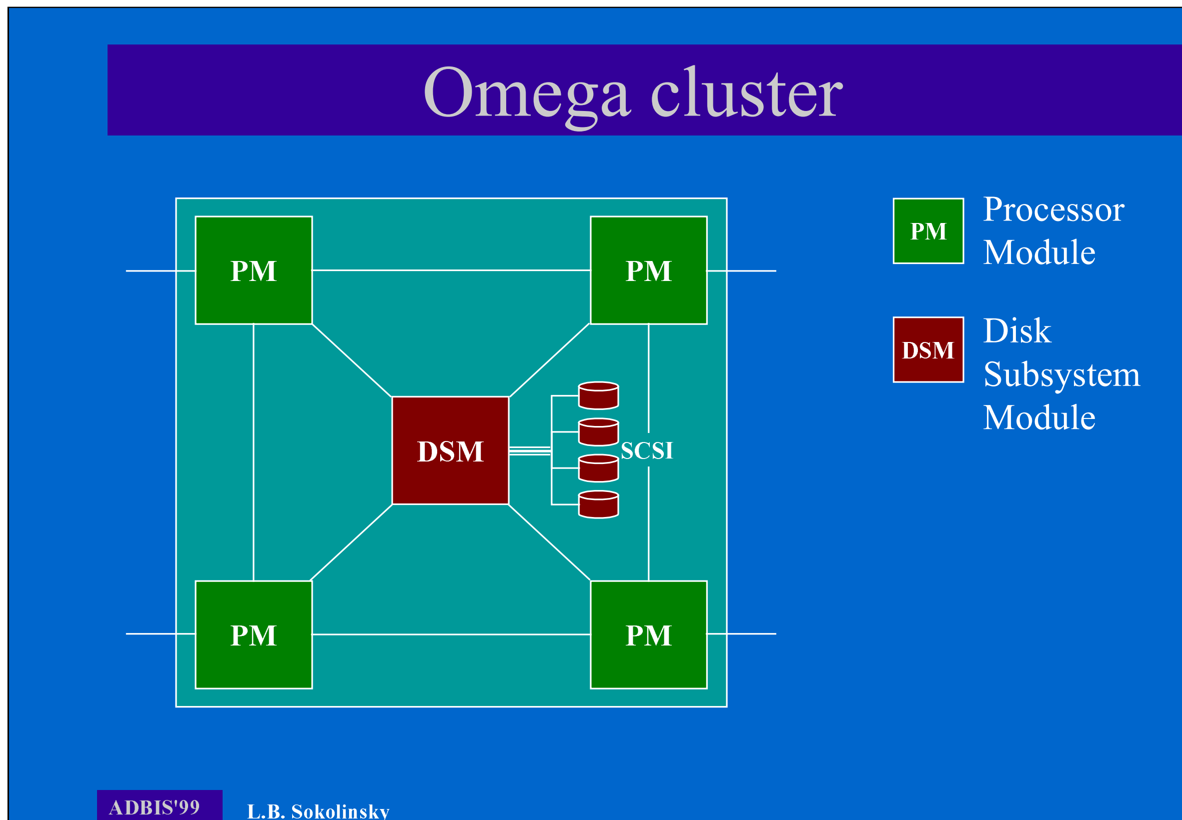
The 3rd level is presented by Omega system



The MBC processor module includes two processor devices: computational processor and communicational processor.

Computational and communicational processors share common memory 16 or 32 Megabytes. The communicational processor has 4 or 8 Mbytes of private memory. These two processors exchange data in shared memory.

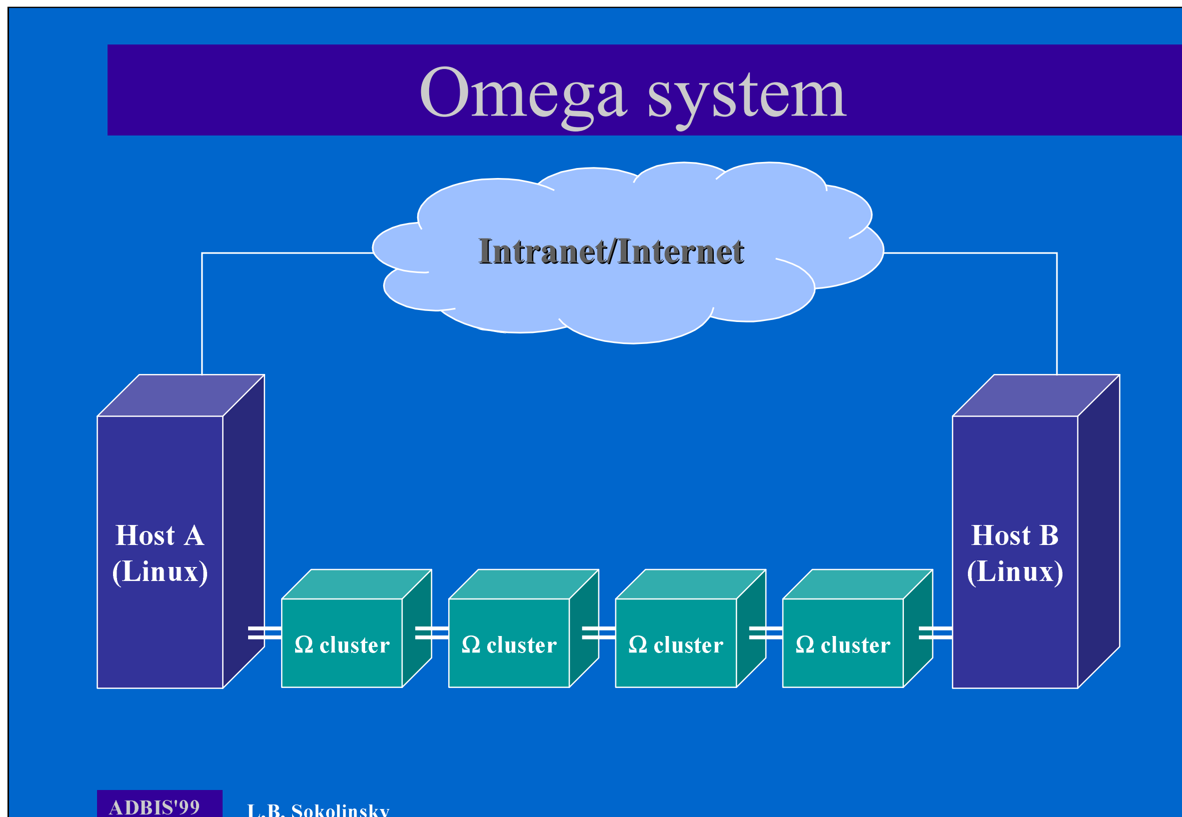
The communicational processor has four serial links for board-to-board communications.



The second level of hardware hierarchy is presented by the *Ω- cluster*. The Ω- cluster is a shared-disk system whose nodes are MBC processor modules.

Disk Subsystem Module has its own communicational processor with private memory connected to four disk devices by SCSI bus.

One link of each processor module remains free to connect the given Ω- cluster to others.



On the third level of hardware hierarchy, Ω- clusters are composed into *Ω- system* in a shared-nothing manner.

One Ω- system can be scaled up to several hundreds Ω- clusters.

There is no restriction for the interconnect topology in the Ω- system. It can be varied from simple rule box to hypercube.

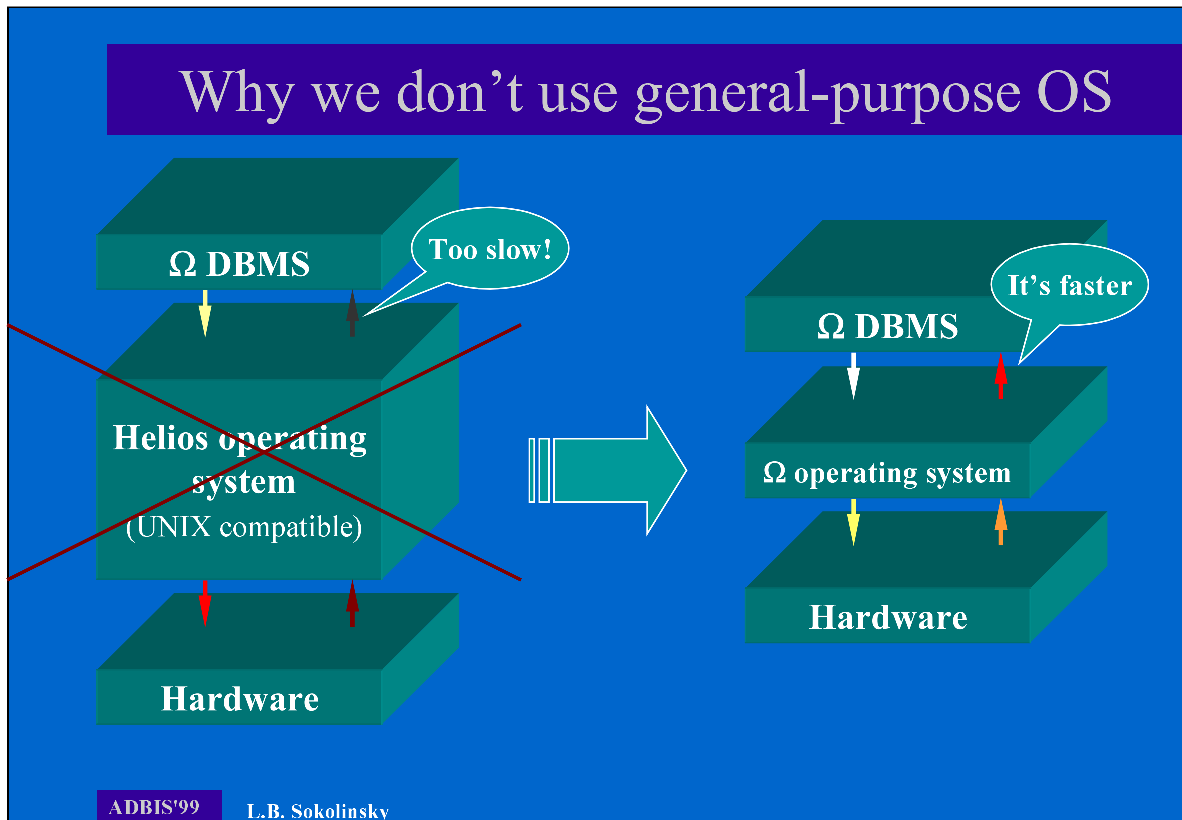
The existence of two hosts is conditioned by fault-tolerance requirements.

Contents

- Omega hardware architecture
- **Omega operating system support**
- Omega operating system structure
- A model for thread scheduling
- Thread manager implementation
- Conclusion

ADBIS'99 L.B. Sokolinsky

The next topic is “Omega operating system support”.



As an operating system, MBC system uses Helios operating system.

Helios is a UNIX compatible distributed operating system.

Helios is a general-purpose operating system and all services provided by Helios turn out too slow for DBMS needs. For example, the latent part of message passing initiation amounts up to 3000 bytes.

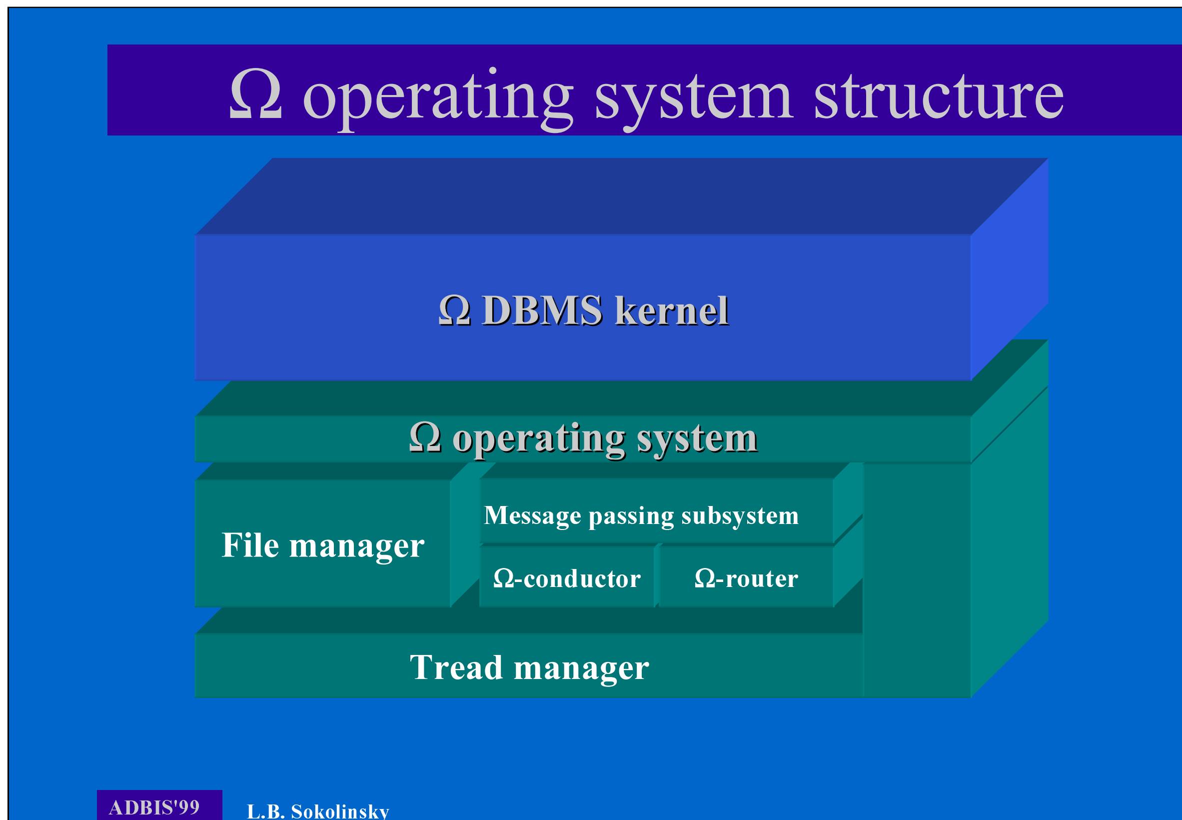
It motivated us to develop an operating system that can meet the Omega parallel DBMS requirements.

Contents

- Omega hardware architecture
- Omega operating system support
- **Omega operating system structure**
- A model for thread scheduling
- Thread manager implementation
- Conclusion

ADBIS'99 L.B. Sokolinsky

The next topic is “Omega operating system structure”.



The Omega OS includes the following subsystems: a thread manager, a message passing subsystem and a file manager.

The message passing subsystem consists of Omega-conductor and Omega-router.

Contents

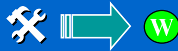
- Omega hardware architecture
- Omega operating system support
- Omega operating system structure
- **A model for thread scheduling**
- Thread manager implementation
- Conclusion

The next topic is “A model for thread scheduling”. This model was used in tread manager.

Producer/consumer paradigm

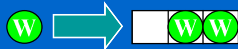
Producer

1. Make new widget



2. If buffer is full, go to sleep

3. Put widget to buffer



4. If buffer was empty, wake consumer

Consumer

1. If buffer is empty, go to sleep

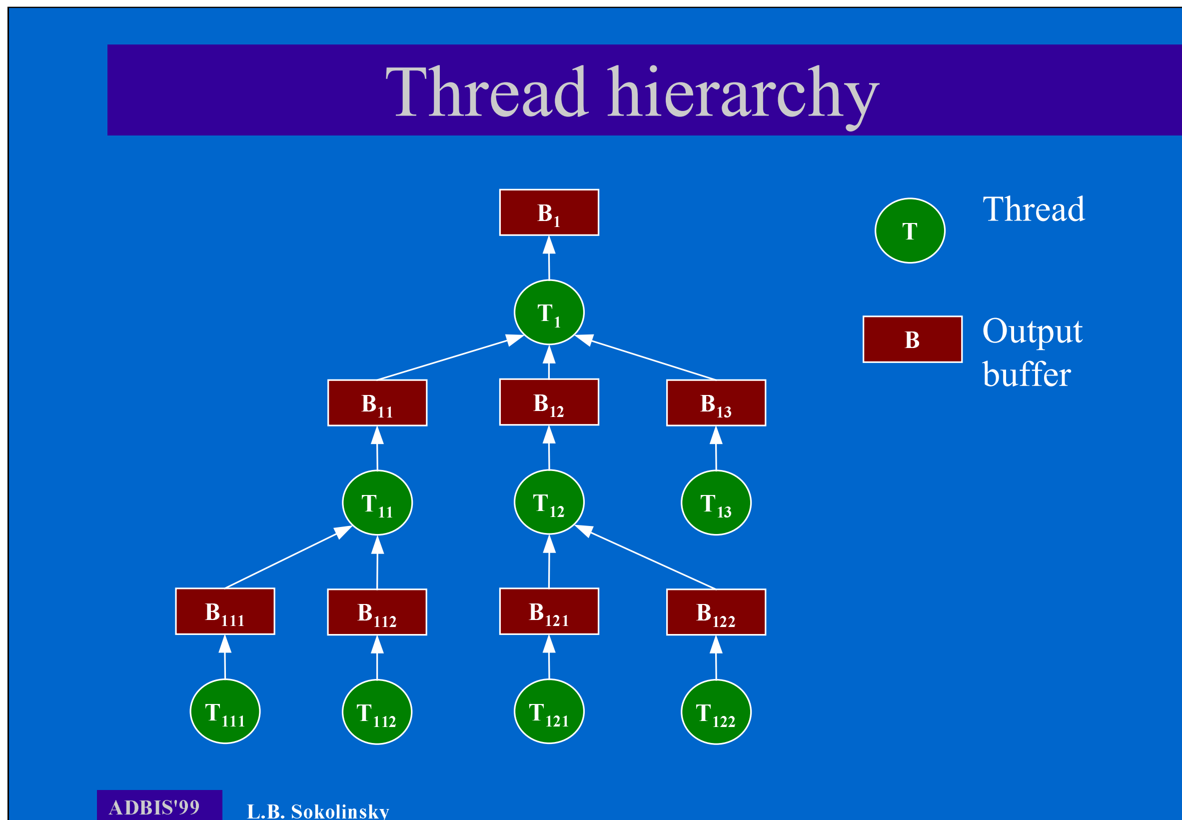
2. Take widget from buffer

3. If buffer was full, wake producer

4. Consume the widget

ADBIS'99 L.B. Sokolinsky

Our model is based on producer/consumer paradigm. According to this paradigm, the producer acts as it shown in the left frame, and consumer acts as it shown in the right frame.



In our model, every task is presented by a root thread.

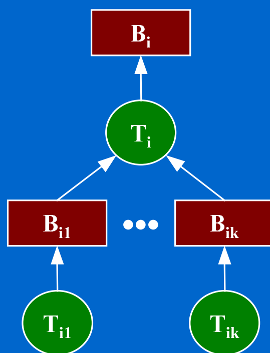
The root thread can generate an arbitrary amount of the child threads. Each child thread can generate its own set of child threads. Thus all generated threads form a hierarchy.

We assume that each thread produces some data *granules* for its father. To produce its own granule, the thread needs the granules from its children.

A output buffer is associated with each thread. This buffer is organized as a queue.

The supplier thread puts the produced granules to its output buffer as the consumer thread consumes the granules from its input buffers.

A model for thread scheduling



- ***Time slice*** ⇔ time of one data granule producing in any thread
- ***T–factor*** τ_i ⇔ output buffer fullness ($\forall i$: $0 \le \tau_i \le 1$)
- ***Factorfunction*** $f_i(t)$ ⇔ calculates T-factor for T_i in time t
- T_i is ***disjunctive*** ⇔ T_i can produce an output by consuming only one input from any of $T_{i1}, \ldots, T_{ik}$
- T_i is ***conjunctive*** ⇔ T_i can produce an output by consuming inputs from all of $T_{i1}, \ldots T_{ik}$

- T_i is ***suspended*** in time t ⇔ $f_i(t)=1$
- ***Disjunctive*** thread T_i is ***blocked*** in time t ⇔ $\sum_{j=1}^{k} f_{ij}(t) = 0$
- ***Conjunctive*** thread T_i is ***blocked*** in time t ⇔ $\prod_{j=1}^{k} f_{ij}(t) = 0$
- T_i is ***ready to run*** ⇔ T_i isn't suspended and T_i isn't blocked

ADBIS'99 L.B. Sokolinsky

Now we described a new model for thead scheduling.

The time to be needed for producing one granule by any thread is a quantum of system time slicing.

Define T–factor as the output buffer fullness.

Define a factorfunction as a function which calculates T-factor.

Define the thread as disjunctive if it can produce an output by consuming only one input from any of its children.

Define the thread as conjunctive if it can produce an output only by consuming inputs from all of its children.

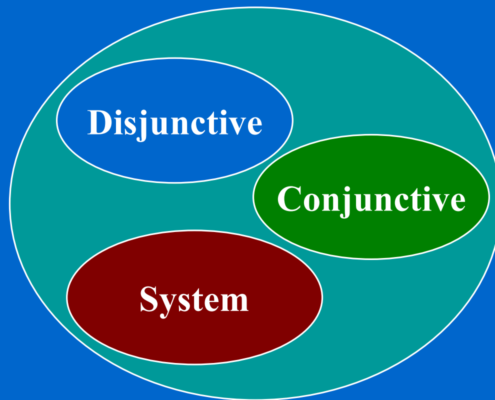
We'll say, the thread is in suspended mode if its T-factor equals zero.

We'll say, the disjunctive thread is in blocked mode if all of its child T-factors equal zero.

We'll say, the conjunctive thread is in blocked mode if any of its child T-factor equals zero.

The thread is in ready-to-run mode if it isn't in suspended or blocked modes.

Kinds of threads



- *System thread* ⇔ has output buffer of length 1, and does not have any child threads. Define the type of all other threads as a *user*.
- Thread tree is in *normal form* if it includes only disjunctive and conjunctive threads
- **Proposition.** Any thread tree can be converted into some normal form. (The proof is following from the theorem about conjunctive normal form)

In our model we distinguish system and user threads. System thread has output buffer of length 1, and does not have any child threads. Define the type of all other threads as a user.

User threads can be divided into three classes: disjunctive, conjunctive and mixed.

We'll say that thread tree is in a normal form if it includes only disjunctive and conjunctive threads.

We can formulate the following proposition. Any thread tree can be converted into some normal form. The proof is following from the theorem about conjunctive normal form in mathematical logic. This proposition shows that the set of disjunctive and conjunctive threads is quite representative.

Contents

- Omega hardware architecture
- Omega operating system support
- Omega operating system structure
- A model for thread scheduling
- **Thread manager implementation**
- Conclusion

ADBIS'99 L.B. Sokolinsky

The next topic is “Thread manager implementation”.

Thread manager interface

```
/* Thread manager initialization */
void init(void*, /* Pointer to the factorfunction of the root thread */
          char, /* Thread type (user or system) */
          int   /* Nice (static priority) of the root thread */);

/* New thread creation */
int /* thread identifier */
fork(void*,   /* Pointer to a body of the thread */
     void*,   /* Pointer to a argument list of the thread */
     void*,   /* Pointer to the factorfunction of the thread */
     char,    /* Thread type (user or system) */
     int      /* Nice (static priority) of the thread */);

/* To perform a schedule */
void schedule(void);
```

ADBIS'99 L.B. Sokolinsky

The thread manager interface includes three main functions which are shown on this slide.

The **init**() function performs the initialization of the thread manager and registries the current process as a root thread.

The **fork**() function creates a new thread.

The **schedule**() function calls the scheduler. Each time when a user thread produces a granule, it must execute a scheduling operation call.

Static & dynamic priorities

nice – *static priority* ($-20 \le nice \le 20$). It's assigned by user.

dprty – *dynamic priority*. It's calculated by the system:

$$dprty = K*C + N*f + threshold_nice + nice$$

- ***C*** - counter. It's incremented when thread is scheduled to run. Every ***m*** scheduling cycles, *C* is recounted:
  $C = C*k$.
  k - some fixed value ($0<k<1$).
- ***f*** - value of factorfunction
- ***K***, ***N*** - normalizing constants
- ***threshold_nice*** - threshold value for user threads. For system threads, it equals 0.

ADBIS'99 L.B. Sokolinsky

For thread scheduling, we use the priorities. Each thread has its own *static priority*, which is assigned as a *nice value* during of its creation.

The real scheduling policy is based on a *dynamic priority*. The dynamic priority is a function of the static priority.

We provide a simple formula for dynamic priority calculation:

$$dprty = K*C + N*f + threshold_nice + nice$$

Here *C* is a counter associated with the thread. *C* is incremented every time, when the thread is scheduled to run. Every *m* scheduling cycles, the *C* parameter of each thread is recounted by this formula:

$$C := C*k.$$

f is a value of the factorfunction.

K and *N* are normalizing constants.

threshold_nice - is a threshold value of the dynamic priority for the user threads. For the system threads, it equals zero.

Parameters for the formula

Parameter	Value
k	0.5
m	60
K	0.01
N	0.1
threshold_nice	40

ADBIS'99 L.B. Sokolinsky

We checked the thread manager by several tests. In the formula of dynamic priority calculation, we used the parameter values, which are shown in this slide. These values were obtained by an experimental way.

Scheduling cycle

1. Recount T-factors for all system threads
2. Recount T-factors for active thread and all its sons
3. If there are no “ready-to-run” threads, give control to root thread
4. Recount dynamic priorities for all threads whose status is “ready-to-run”
5. Give control to “ready-to-run” thread whose dynamic priority is highest

ADBIS'99 L.B. Sokolinsky

This slide describes the scheduling cycle.

1. Recount T-factors for all system threads
2. Recount T-factors for active thread and all its sons
3. If there are no “ready-to-run” threads, give the control to root thread
4. Recount dynamic priorities for all threads whose status is “ready-to-run”
5. Give the control to “ready-to-run” thread whose dynamic priority is the highest

Contents

- Omega hardware architecture
- Omega operating system support
- Omega operating system structure
- A model for thread scheduling
- Thread manager implementation
- **Conclusion**

For the conclusion.

The main advantages

1. We can associate a separate user thread with every operation in the query tree
2. We can use system threads for efficient implementation of operating environment subsystems
3. We needn't any primitives for threads synchronization. The synchronization is performed *automatically*
4. Dynamic priority mechanism provides the possibility for accurate and effective scheduling

ADBIS'99 L.B. Sokolinsky

The main advantages of described model are following.

Future work

- Investigate different algorithms for dynamic priority evaluation
  - investigate different formulas
  - in formulas, use T-factors of thread's father and children

ADBIS'99 L.B. Sokolinsky

As a future work we are going: