Implementing Threads

Knut Omang
Ifi/Oracle
21 Sep, 2011

(with slides from several people)

ORACLE

Today:

- Implementing Threads
  - user level
  - kernel level
  - hybrid
    - scheduler activation
  - pop-up threads
- Understanding the hardware
  - effect of cache misses
  - context switch performance

ORACLE

Threads and Address Spaces

- Thread
  - A sequential line of execution within a process
- Address space
  - All the state needed to run a program
  - Provide illusion that program is running on its own machine (protection)
  - There can be more than one thread per address space

ORACLE

The Thread Model

Per process items	Per thread items
Address space (MMU,page table)	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	
Items shared by all threads in a process	Items private to each thread

ORACLE

Threads vs. multiple processes

- Thread-to-thread context switch within process less expensive – partly shared state
- fine grained communication easier since shared address space

ORACLE

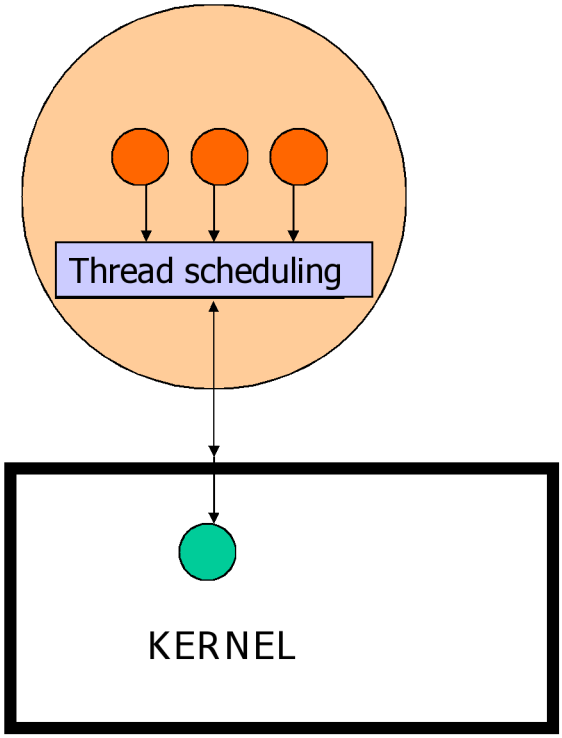
Why Threads?

- Utilize multiple cores/multiple CPUs
  - Few plausible alternatives…
- As an abstraction to simplify programming
  - Separate independent tasks
    - GUI vs I/O
  - Just simplify programming model

ORACLE

Implementation of threads

User level threads



single threaded kernel

ORACLE

User Level Thread Packages

- Implementing threads in user space
  - Kernel knows nothing about them: appears to be single-threaded application
  - Threads are switched by fast 'private' runtime system
  - Processes can implement custom scheduling algorithms
  - Blocking system calls in one thread block all threads of the process

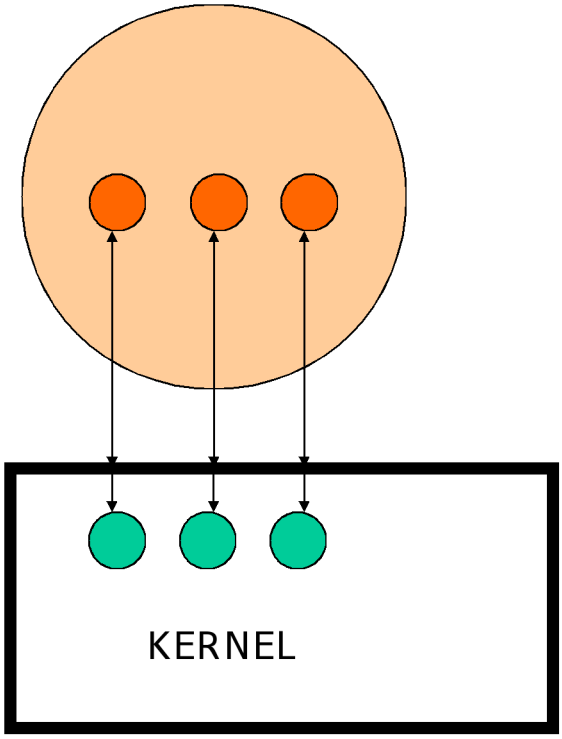
→ prohibit blocking calls or write jackets around library calls
(ex. via LD_PRELOAD)

  - A page fault in one thread will block all threads of the process
  - No clock interrupts can force a thread to give up CPU, spin locks cannot be used
  - Hard to utilize multiple CPUs….

ORACLE

Implementation of threads

1-1 model and light weight processes



Multithreaded kernel

ORACLE

Single vs multithreaded kernel

Multithreaded kernel:

- Each kernel thread has
  - a user stack
  - a private kernel stack
- Pros
  - concurrent accesses to system services
  - works on a multiprocessor
- Cons
  - More memory
  - More synch overhead

Single threaded kernel:

- Each kernel thread has
  - a user stack
  - a shared kernel stack with other threads
- Pros
  - less memory
- Cons
  - serial access to system services
  - contention for kernel managed resources

ORACLE

A modern thread API

- Thread manipulation
  - create/cancel
  - join (wait for child(ren) to terminate)
- Mutual exclusion
  - lock (acquire), unlock (release)
- Condition variables/monitors
  - wait, signal, broadcast
- Scheduler hints
  - yield,exit, sched.policy, signal policy/send…
  - thread affinity

ORACLE

User Level Thread Packages

- Implementation options
  - Libraries
    - Basic system libraries (“invisible”)
    - Additional system libraries
    - Additional user libraries
  - Language features
    - Early Java (1.0 – 1.2)
    - ADA
    - Numerous research projects..

ORACLE

Many names/ways of thinking about (quasi-)parallelism – not new..

- Co-routines (Simula-67)
  - Call/detach
- Event-driven programming
  - Inner loop processing events
  - Everything becomes events…
  - Asynchronous interfaces needed
- Continuations (from functional languages)
- User level threads…

ORACLE

Kernel Level Thread Packages

- Implementing threads in the kernel
  - Threads created/destroyed by kernel calls
    - optimization by recycling threads
  - Kernel table per process, one entry per thread
  - Kernel does scheduling
    - clock interrupts available
    - blocking calls and page faults no problem
  - But: Performance penalty of thread mgmt in kernel:
    - User/kernel switch overhead

ORACLE

Solution: schemes to collaborate between user/kernel mode

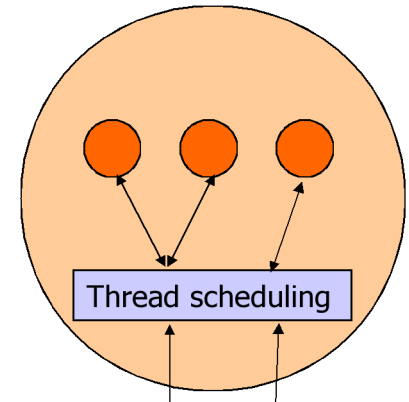
”Typical” schema:

- Let kernel and user mode communicate
- Let user mode library code decide when a full process switch is needed and when ’fast paths’ can be taken
  - Recent example: futexes – Linux 2.6

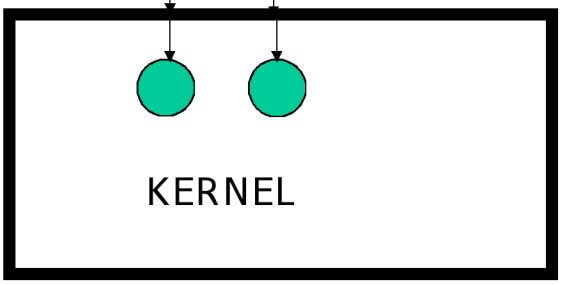
ORACLE

Implementation of threads

Hybrid model
(M on N)



Multithreaded kernel



ORACLE

Scheduler Activations - Design

- Combine advantages of kernel space implementation with performance of user space implementations
- Scheduler activations provide an interface between the kernel and the user-level thread package:
  - Kernel is responsible for processor allocation and notifying the user-level of events that affect it.
  - User-level is responsible for thread scheduling and notifies the kernel of events that affect processor allocation decisions.
  - Avoid unnecessary transitions between user and kernel space

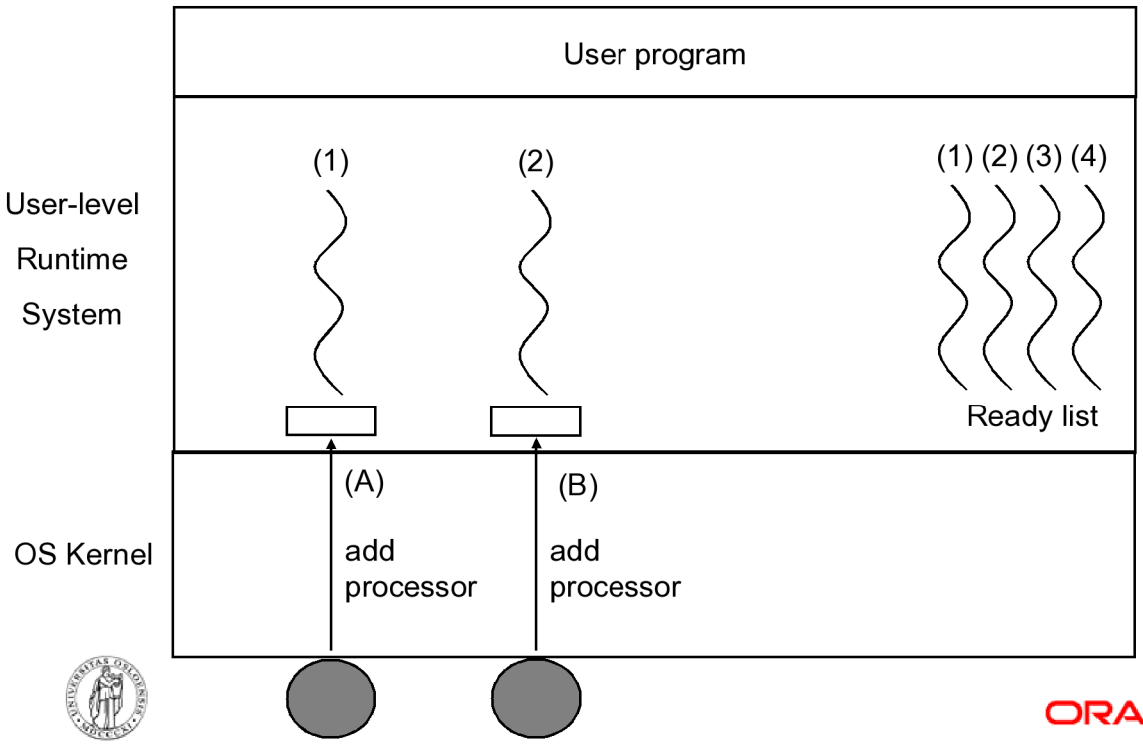
ORACLE

Scheduler Activations - Implementation

- Kernel assigns virtual processors to each process
- User level runtime system allocates threads to processors
- The kernel informs the process's runtime system via an upcall when one of its blocked threads becomes runnable again
- Upcalls: Implemented similar to signals in UNIX - async event
- Runtime system can schedule
- Runtime system has to keep track when threads are in or are not in critical regions
- Example of hybrid solution
- Objection: Upcalls violate the layering principle

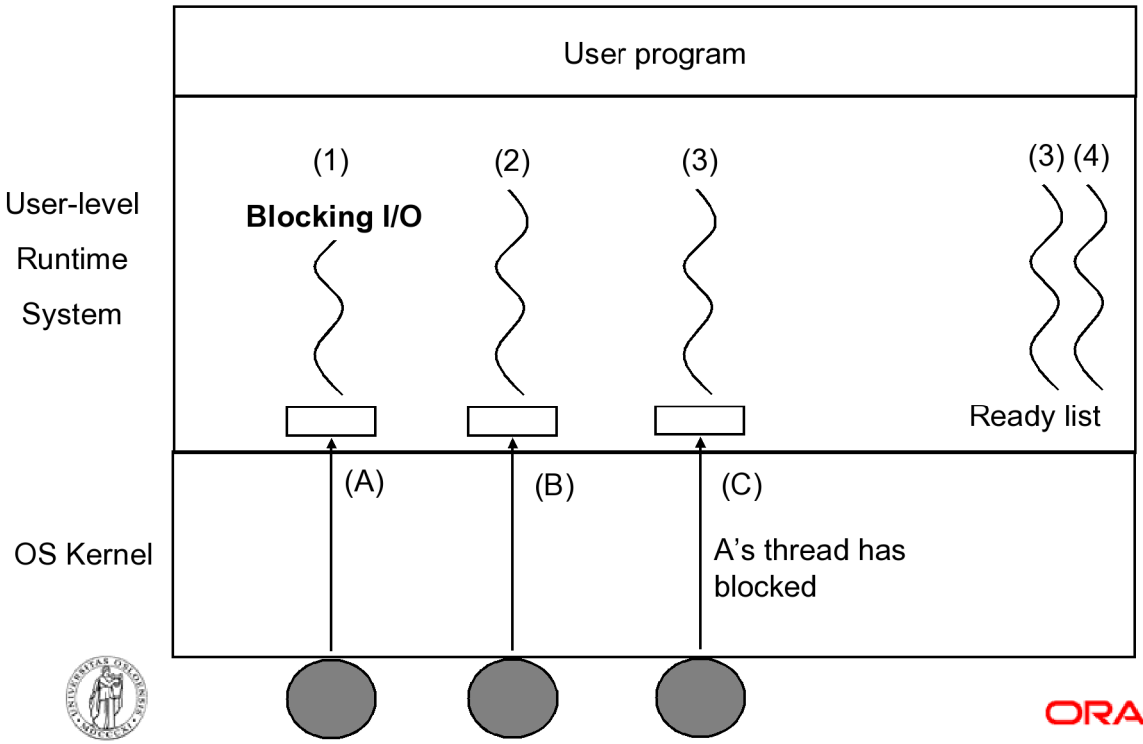
ORACLE

Scheduler Activations



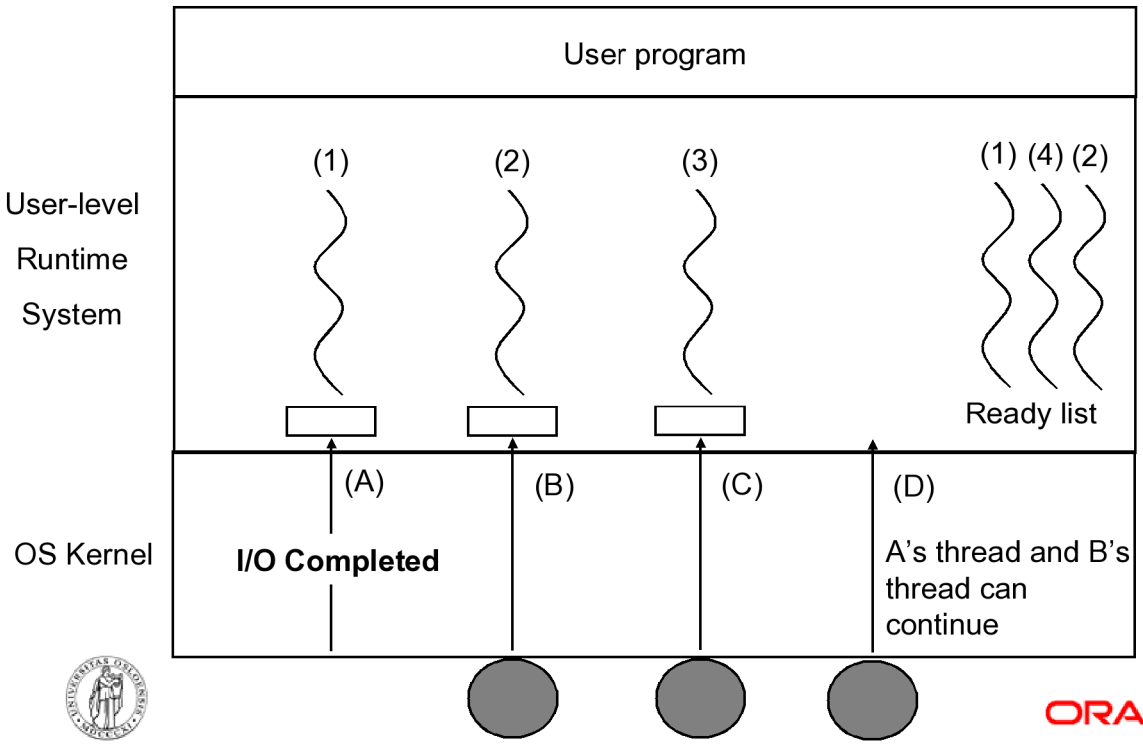
ORACLE

Scheduler Activations



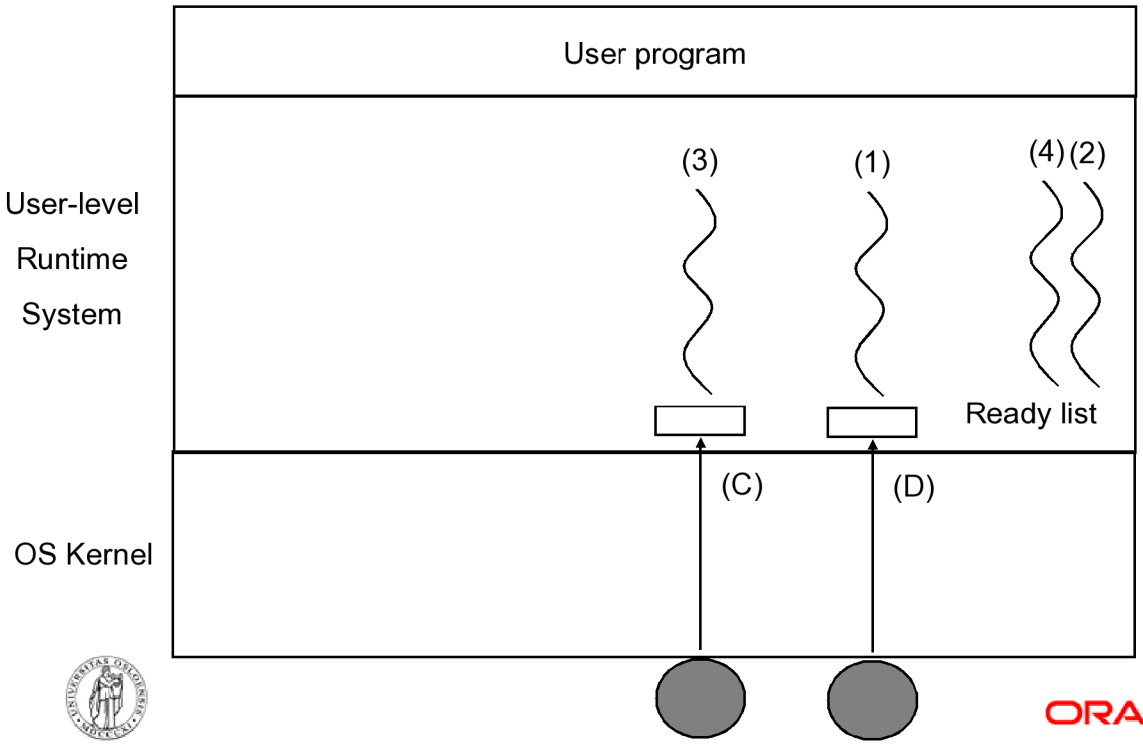
ORACLE

Scheduler Activations



ORACLE

Scheduler Activations



ORACLE

Scheduler Activations (SA) papers

- Nathan J. Williams: ”An Implementation of Scheduler Activations on the NetBSD Operating Systems”, in Proceedings of Freenix/Usenix 2002
- Earlier implementations of scheduler activations in Taos, Mach 3.0, BSD/OS, Digital Unix (now Compaq Tru64 Unix)

ORACLE

SA - Kernel Interface - I

- Application → scheduler activation system, i.e., by system calls:
  - `sa_register( )`
  - `sa_setconcurrency( )`
  - `sa_enable( )`
  - `sa_yield ( )`
  - `sa_preempt( )`
- Scheduler activation → application, i.e., by upcall:
  -
```
void sa_upcall(int type,
               struct sa_t *sas[],
               int events,
               int interrupted,
               void *arg);
```

ORACLE

SA - Kernel Interface - II

- Events that generated upcalls:
  - `SA_UPCALL_NEWPROC`
  - `SA_UPCALL_PREEMPTED`
  - `SA_UPCALL_BLOCKED`
  - `SA_UPCALL_UNBLOCKED`
  - `SA_UPCALL_SIGNAL`
  - `SA_UPCALL_USER`

ORACLE

SA - Kernel Interface - III

- Stacks:
  - Any upcall code needs to store local variables, return address, etc.
  - Using stack of preempted thread?
    - New processor allocations
    - Makes thread management more difficult
  - Each upcall got its own stack
  - System call `sa_stack( )`
- Signals:
  - Support the POSIX signal model
  - Kernel does not know about specific threads
  - Signals are handed to the application with an upcall

ORACLE

Context switch Performance

Taken from Anderson et al 1992

Operation	User level threads	Kernel-level threads	Processes
Null fork	34µs	948µs	11,300µs
Signal-wait	37µs	441µs	1,840µs

Observations

•Look at relative numbers as computers are faster in 2009 vs. 1992

•**Fork: 1:30:330**

•Time to fork off around 300 user level threads ~time to fork off one single process

•Fork off 5000 threads/processes: 0.005s:0.15s:1,65s. OK if long running application. BUT we are now ignoring other overheads when actually running the application.

•**Signal/wait: 1:12:50**

•Assume 20M signal/wait operations: 0,3min:4 min:16,6min. **Not** OK.

Why?

•Thread vs. Process Context switching

•Cost of crossing protection boundary

•User level threads less general, but faster

•Kernel level threads more general, but slower

•Can combine: Let the kernel cooperate with the user level package

ORACLE

Memory subsystem numbers – more up-to-date (double writes)

CPU	1 level cache access time	Memory access time	Linux bogomips* cores	'Instr' per cache miss
AMD-K6, 0.5 GHz	12 ns	80 ns	~1000	80
Athlon XP 1600+, 1.4 GHz	2.5 ns	14 ns	~2800	39
AMD Athlon 64 X2, 2.3 GHz	3.0 ns	12 ns	~4000	55
Intel Xeon 2.1 GHz (2 core)	1.0 ns	5 ns	~8400	30
AMD Phenom X4, 2.6 GHz	1.3 ns	2.2 ns	~20800	11

Measuremens using cachebench:
http://icl.cs.utk.edu/projects/llcbench/cachebench.html

ORACLE

Context switch overhead (newer hardware)

CPU	Context switch with minimal process	Context switch w/ 16KB array (stride 512)	'Instr' per switch (stride 512)
AMD-K6, 0.5 GHz	6.1 µs	7.3 µs	7300
Athlon XP 1600+, 1.4 GHz	2.3 µs	3.7 µs	10359
AMD Athlon 64 X2, 2.3 GHz	3.2 µs	5.0 µs	23000
Intel Xeon 2.1 GHz (2 core)	0.8 µs	1.7 µs	10707
AMD Phenom X4, 2.6 GHz	1.5 µs	2.5 µs	19500

Test code from
http://www.cs.rochester.edu/u/cli/research/switch.htm

ORACLE

Context switch overhead

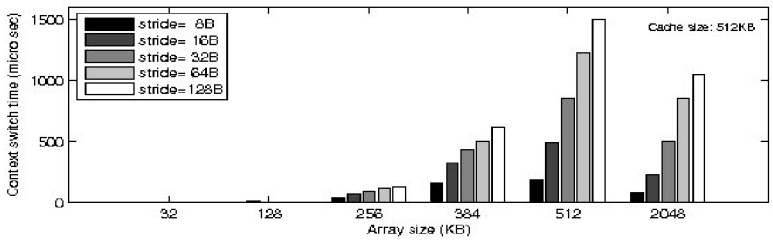


Figure 2: The effect of the access stride on the cost of context switch

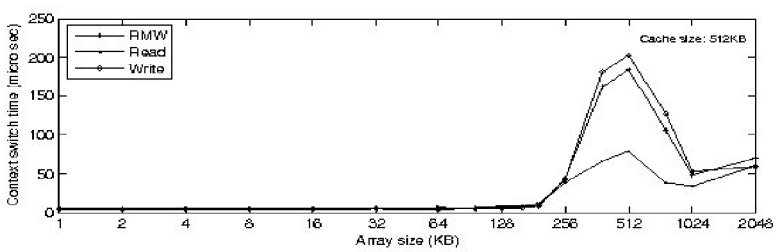
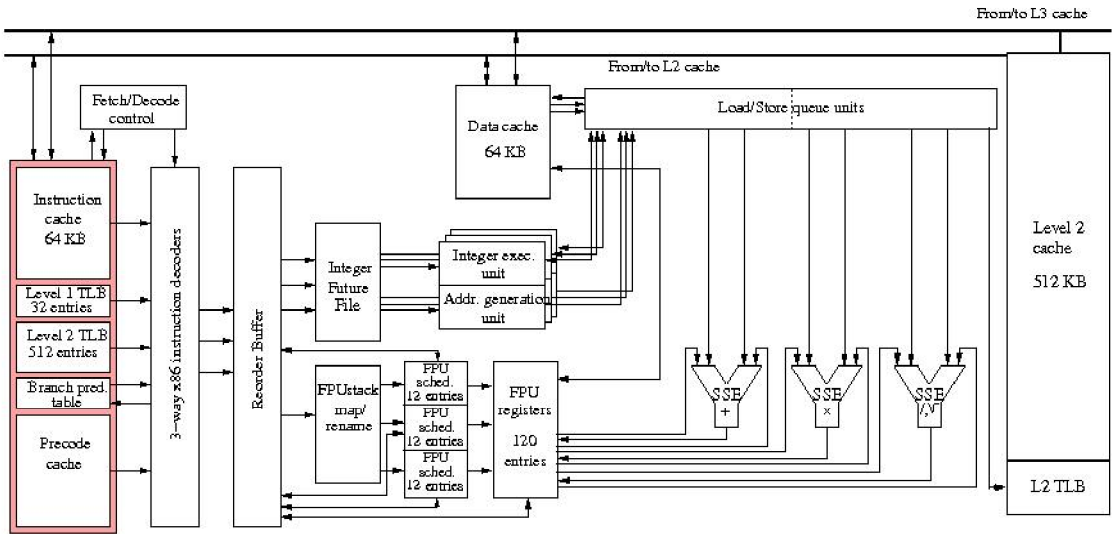


Figure 1: The effect of data size on the cost of the context switch

ORACLE

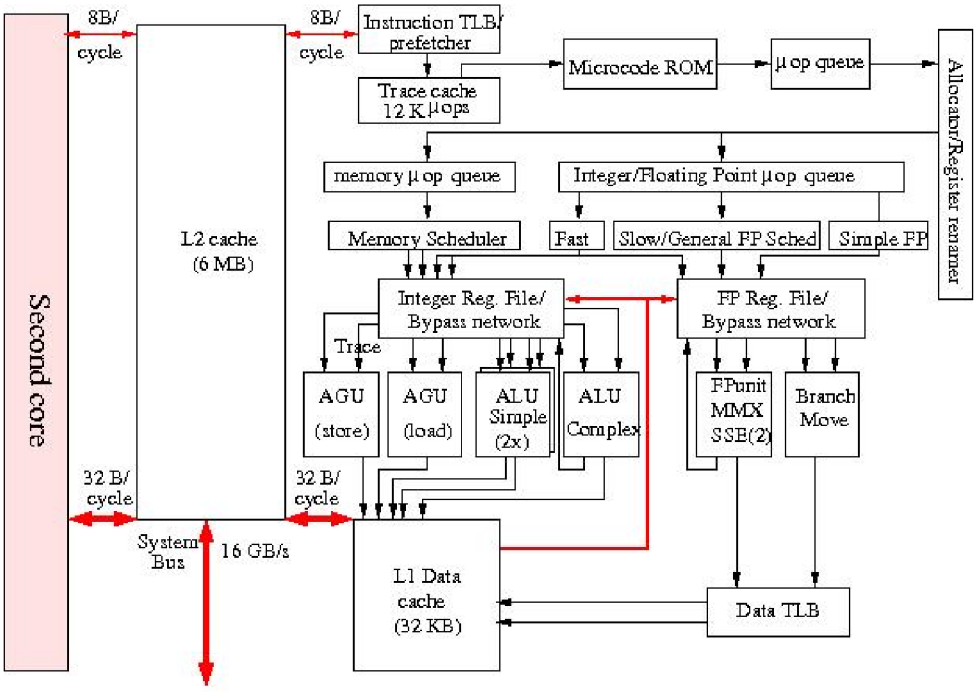
AMD Phenom again..



Internals of an AMD Phenom core

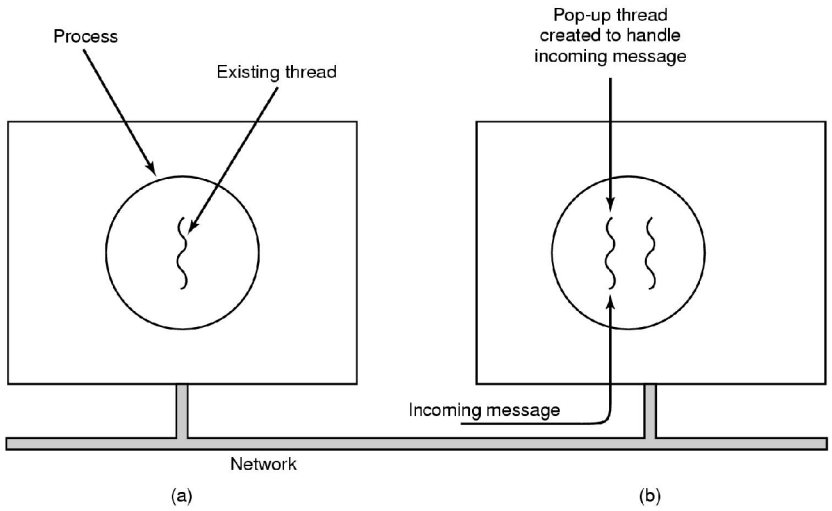
ORACLE

Intel Xeon



ORACLE

Pop-Up Threads



- Creation of a new thread when message arrives

(a) before message arrives
(b) after message arrives

ORACLE

Pop-Up Threads

- Reacting fast to external events
  - Packet processing is meant to last a short time
  - Packets may arrive frequently
- Questions with pop-up threads
  - How to guarantee processing order without loosing efficiency?
  - How to manage time slices? (process accounting)
  - How do schedule these threads efficiently?

ORACLE

Thread Cancellation

- Terminating a thread before it has finished
- Reason:–
  - Some other thread may have completed the joint task
  - E.g., searching a database
- Issues:
  - Other threads may be depending cancelled thread for resources, synchronization, etc.
  - May not be able to cancel one until all can be cancelled

Thread Cancellation (continued)

- Two general approaches:
  - *Asynchronous cancellation* terminates the target thread immediately
  - *Deferred cancellation* allows the target thread to periodically check if it should cancel itself

- `pthreads` provides *cancellation points*

Thread support in some OS'es

- SunOS 4 (Solaris 1):
  - single threaded kernel, no native thread support
- Linux <= 2.2.x
  - single threaded kernel – processes as threads
- Windows NT,2000,XP
  - lightweight processes
  - multithreaded kernel
- Solaris >= 2
  - lightweight processes
  - multithreaded kernel
  - Hybrid M lwps / N threads model, 'bound/unbound'
- Linux 2.4.x / LinuxThreads
  - Limited multithreading – still processes as threads at user level – only a single kernel thread per process
- Linux 2.6.x / Native Posix Threads Library (NPTL)
  - Kernel threads
  - 1 thread = 1 lwp
  - futexes

ORACLE

Existing Thread Packages

- All have
  - Thread creation and destruction
  - Switching between threads
- All specify mutual exclusion mechanisms
  - Semaphores, mutexes, condition variables, monitors

ORACLE

Some existing thread packages

- POSIX Pthreads (IEEE 1003.1c) for all/most platforms
  - Some implementations may be user level, kernel level or hybrid
- GNU PTH
- LinuxThreads (linux kernel 2.4 and before)
- JAVA for all platforms
  - User level, but uses OS time slicing
- Win32 for Win95/98 and NT
  - kernel level thread package
- OS/2
  - kernel level

- Basic idea in most packages
  - Simplicity, fancy functions can be built using simpler one

ORACLE

Interprocess(-thread) communication

"True" multithreading
  - preemptive scheduling of threads/processes
  - multiple CPUs
- Introduces non-determinism:
  - different executions of the same program with same input may produce different results
- Non-determinism wrt. computing results usually a bad idea
  - race conditions!

ORACLE

Real life example:
Process queueing/dequeuing in linux 2.2.x

(From the linux 2.2.16 kernel source:)

```
/* Note that we only need a read lock for the wait queue (and thus do
 * not have to protect against interrupts), as the actual removal from
 * the queue is handled by the process itself.
 */
```

Goal: an implementation of monitors (condition queues) in Linux (kernel level)

Why?

- Linux 2.2 offered low level primitives only (abstracted and simplified)

spin_lock/spin_unlock -- mutual exclusion locks based on busy waiting
spin_lock_irqsave/spin_unlock_irqrestore -- mutual exclusion: interrupt disabling+spin
enqueue/dequeue(task,queue) -- add/remove myself to/from a process queue
schedule() -- invoke scheduler (yield)
wake_up_next(queue) -- next process in "queue" put back on the run queue

ORACLE

Condition variables implementation

```
/* assuming lock is held and
 * interrupts turned off */

void cond_wait(cond c, mutex lock)
{
   enqueue(current,c.queue);
   spin_unlock_irqrestore(lock);
   schedule();
   dequeue(current, c.queue);
   spin_lock_irqsave(lock);
}
```

```
/* assuming lock is held and
 * interrupts turned off */

void cond_signal(cond c)
{
   wake_up_next(c.queue);
}
```

ORACLE

Example case (implementation)

Usage: resource management

```
...
spin_lock_irqsave(lock);
if (<my resource not available>)
    cond_wait(c, lock);
<grab resource>
spin_unlock_irqrestore(lock);

...
spin_lock_irqsave(lock);
<release resource>
cond_signal(c);
spin_unlock_irqrestore(lock);
```

Linux impl. of enqueue/wake_up

```
global mutex queue_lock;
void enqueue(task t, queue q)
{
  spin_lock_irqsave(queue_lock);
  < do the queueing of t in q>
  spin_unlock_irqrestore(queue_lock);
}

task dequeue(queue t)
{
   task t;
   spin_lock(queue_lock);
   t := pop(queue);
   spin_unlock(queue_lock);
   return t;
}
```

ORACLE

Example case: proc.1/proc.2/interrupt,p.1 scenario on dual processor system

process A
(processor 1)

(inside tcp/ip stack)
....
dequeue(A,tcp)
  spin_lock(queue_lock)
    <holds queue_lock..>
     ...Interrupted!...

interrupt context
(executing within A)
(processor 1)

spin_lock_irqsave(L)
  ...spinning on L...

process B
(processor 2)

<holds lock L, int.off>
cond_wait
   enqueue(B,res)
      ...spinning on queue_lock....!

ORACLE

Remember: Safe interprocess communication...

- Murphy's law:
  - *Anything that can go wrong will eventually go wrong!*
  - *There is no limit to the complexity of error scenarios..*
- No assumptions about thread speed (time independence)
  - "Ole-Johan's semicolons" – the semicolon where it all may go wrong...
- Forward progress (but not necessarily for all threads)
- With preemptive scheduling:
  - a thread might lose control at any point!

ORACLE

Important parallel programming lesson:

- The lower the probability of something bad happening, the harder it is to track down!
- Or: a bug that happens frequently is an easy one to reproduce (and hopefully fix..)
  - Eg. it is actually a good thing (during development…)
- Never hide a bug by reducing the chance for it to happen (unless you can make the chance 0…)
  - Don't blame it on cosmic rays… ☺

ORACLE