More on Concurrency and Threads

Knut Omang
Ifi/Oracle
12 Feb, 2014

(with slides from several people)

Today:

- Thread implementation
  - user/kernel/hybrid
  - communication between threads?
- Understanding the hardware
  - effect of cache misses
  - context switch performance

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

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...

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

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

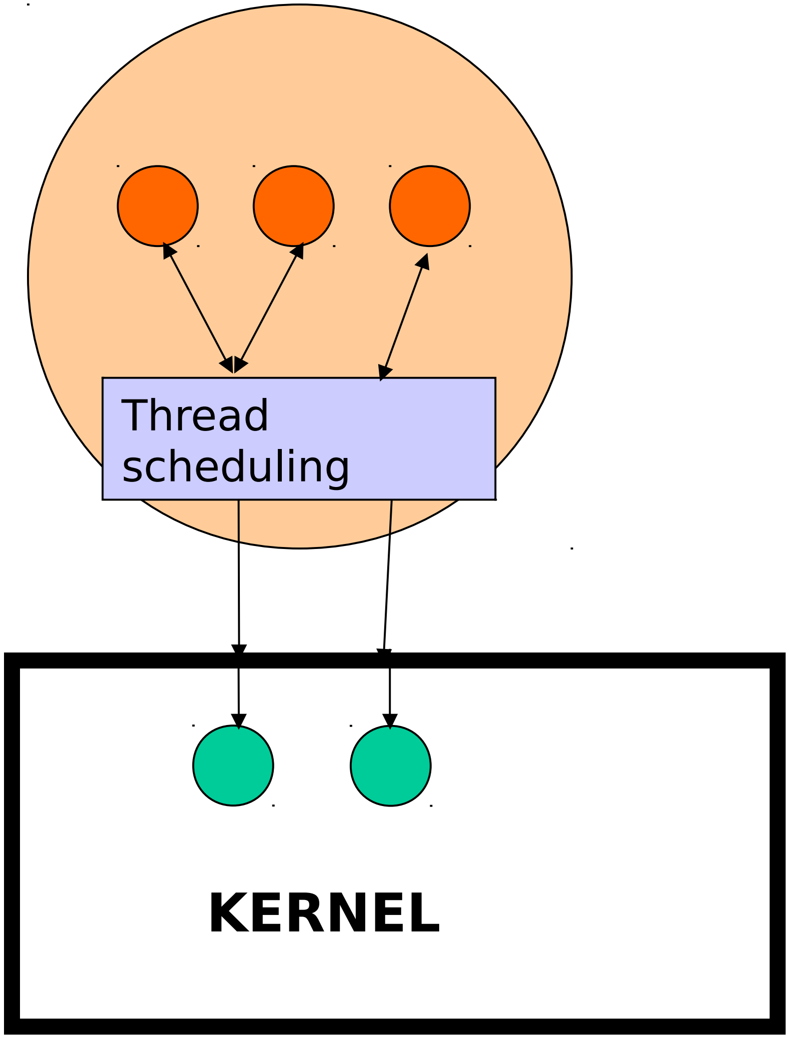
Solution: schemes to collaborate between user/kernel mode

”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
  - Scheduler Activations
  - Futexes – used by Linux NPTL

Implementation of threads

Hybrid model (M on N)



Multithreaded kernel
kernel

Scheduler Activations - Design

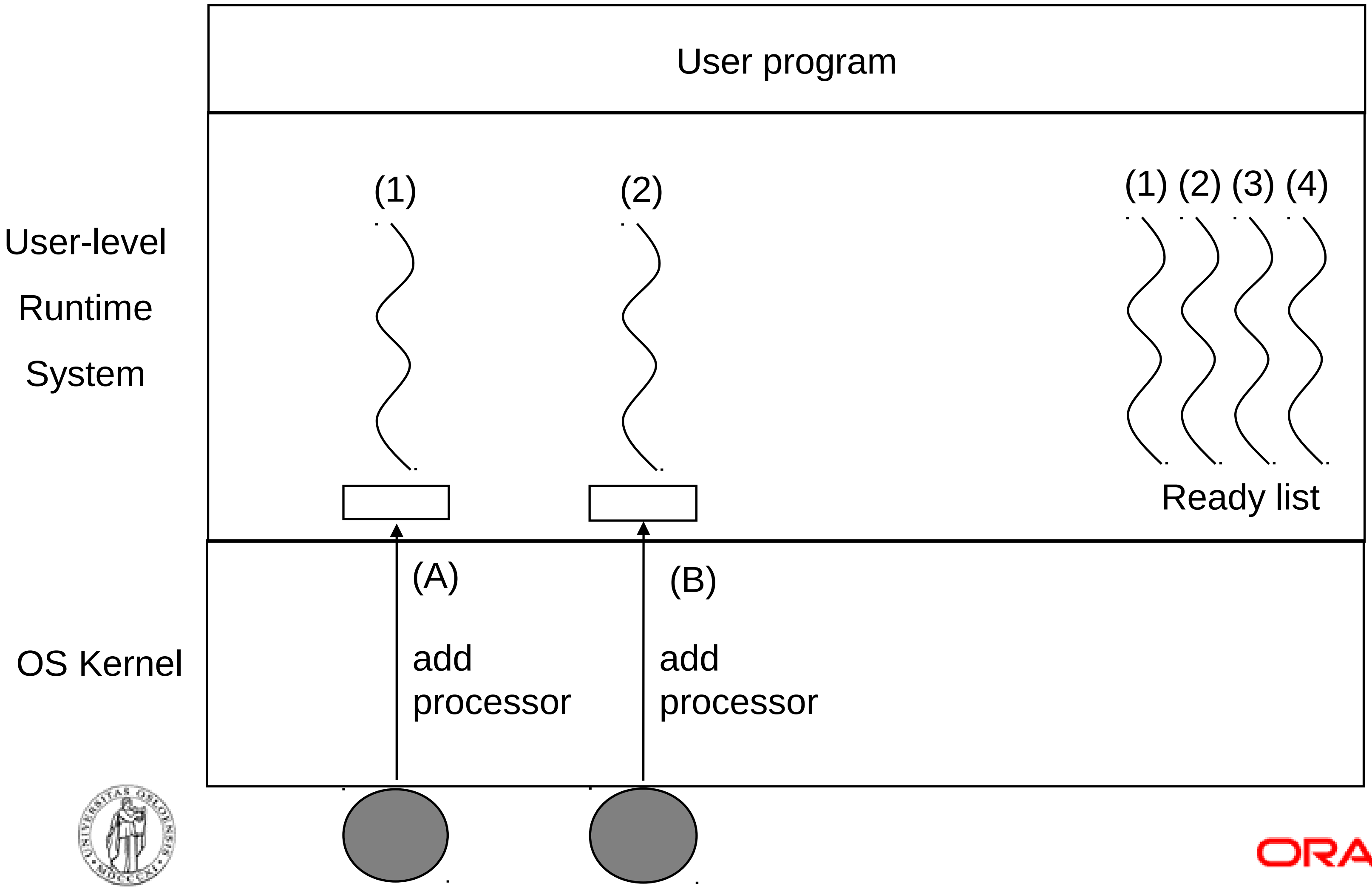
- Combine advantages of kernel space implementation with performance of user space implementations
- Scheduler activations provide an interface between the kernel and the user-level thread package:
  - Kernel is responsible for processor allocation and notifying the user-level of events that affect it.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

Scheduler Activations - Implementation

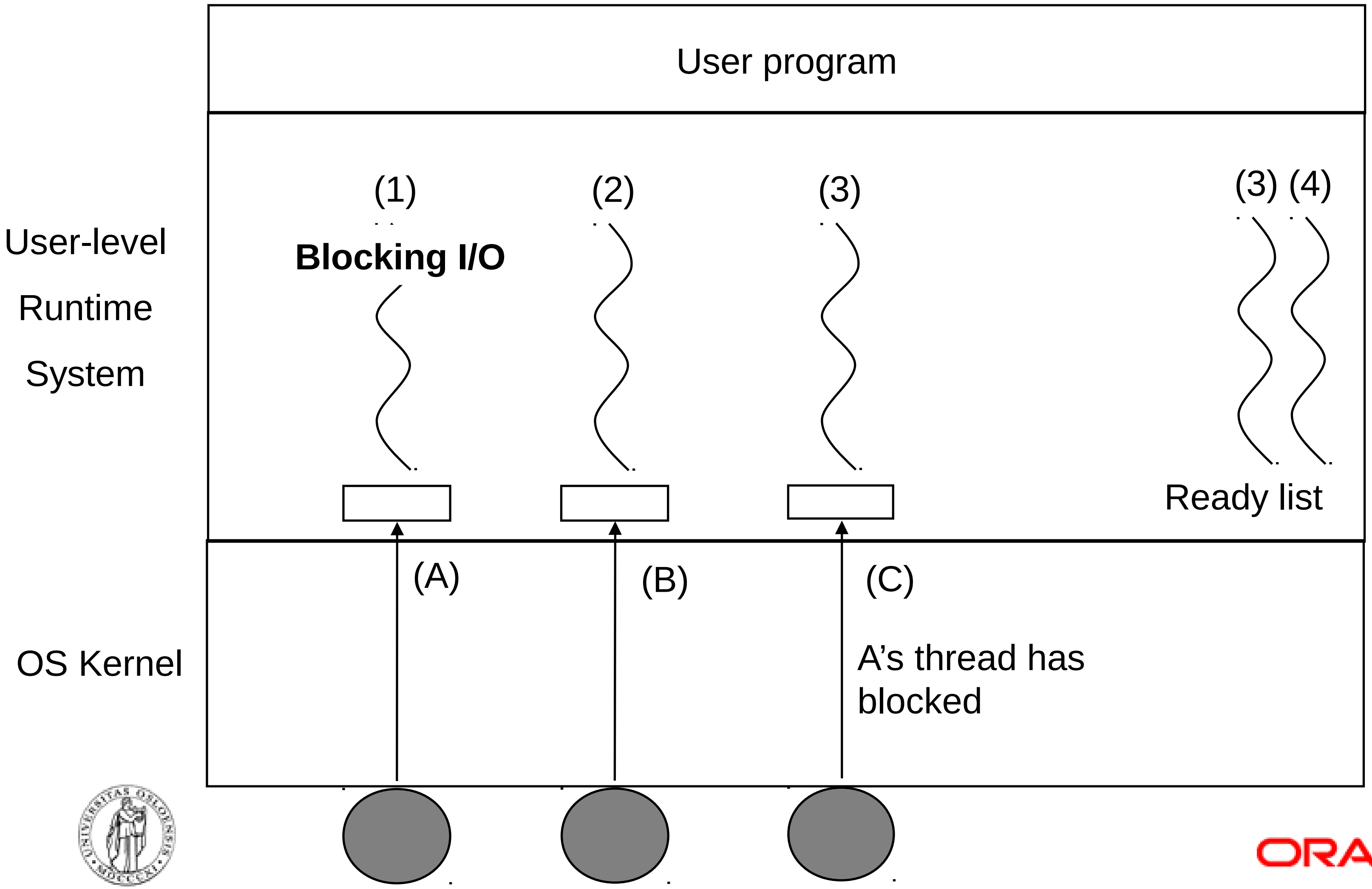
- Kernel assigns virtual processors to each process
- User level runtime system allocates threads to processors
- The kernel informs the process’s runtime system via an upcall when one of its blocked threads becomes runnable again
- Upcalls: Implemented similar to signals in UNIX - async event
- Runtime 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

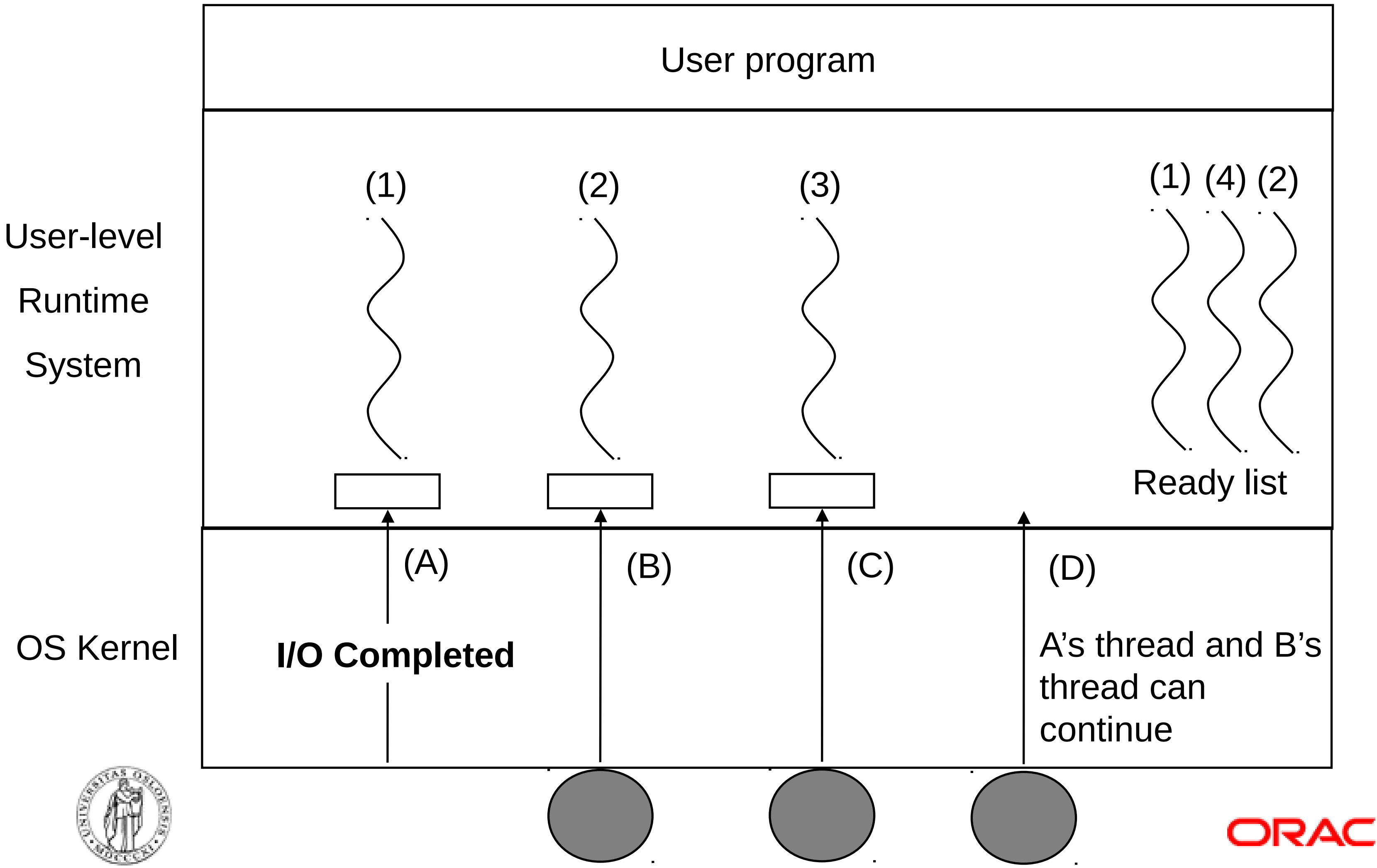
Scheduler Activations



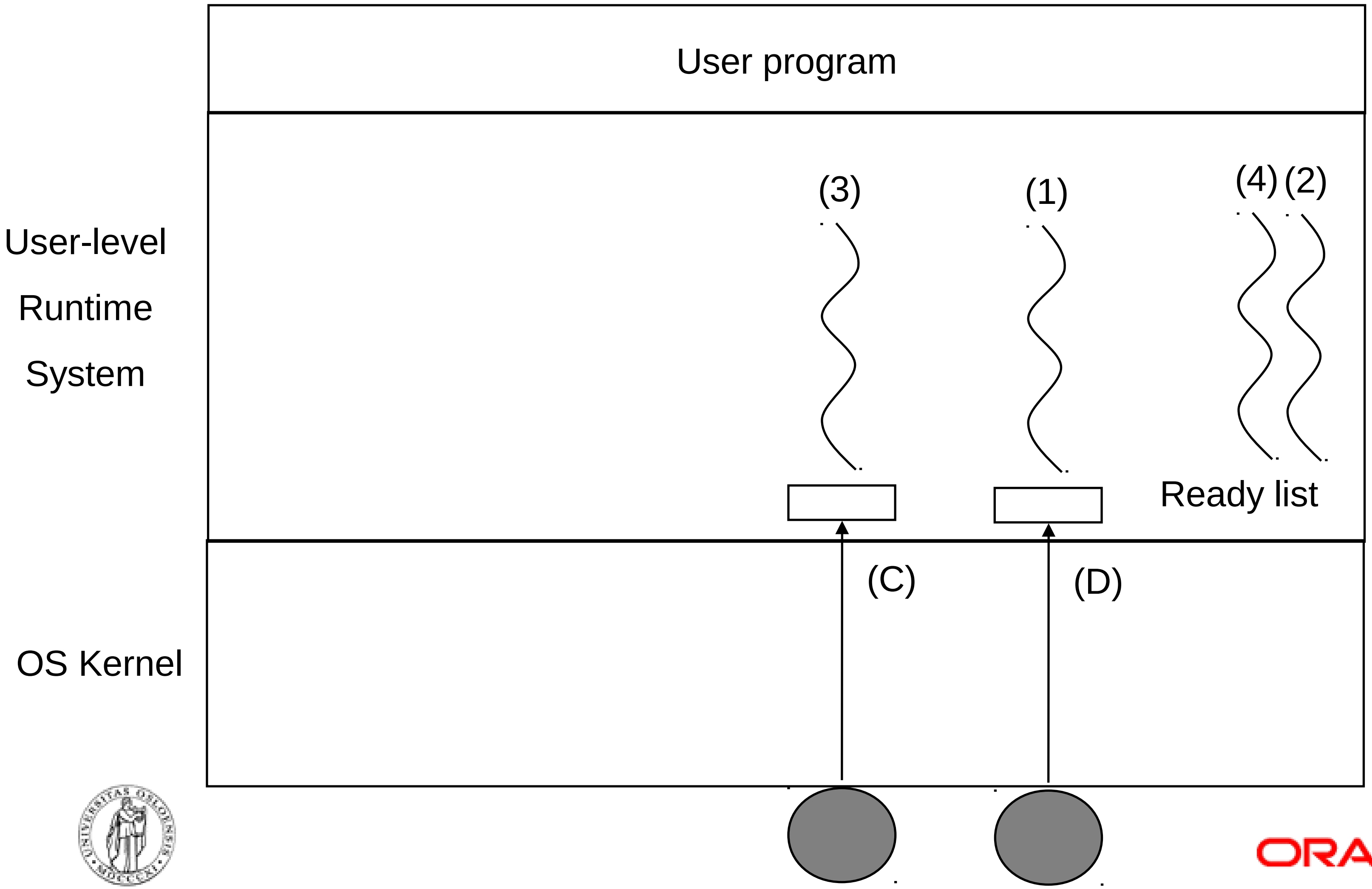
Scheduler Activations



Scheduler Activations



Scheduler Activations



Futex - fast userspace locking

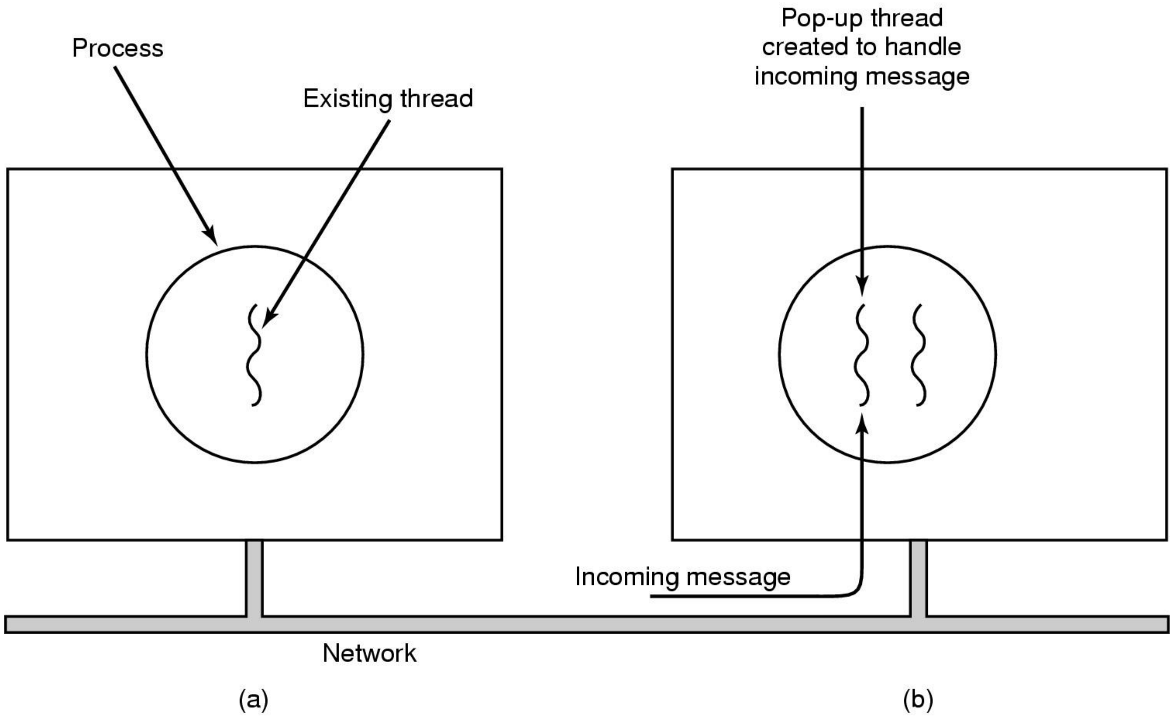
```
int futex(int *uaddr, int op, int val, const
   struct timespec *timeout, int *uaddr2, int
   val3);


op: {FUTEX_WAIT, FUTEX_WAKE}
int stat = 0;
int mutex = 1;
...
void lock(int *mutex)
{
    int gotval =
       atomic_dec_unless(&mutex, -1);
    while (gotval != 0 && !stat)
    {
         stat = futex(&mutex, FUTEX_WAIT,
                              gotval, NULL, NULL, 0);
         gotval =
             atomic_dec_unless(&mutex, -1);
    }
}
```

(~ how it is done!)   Linux >= 2.6

```
void unlock(int *mutex)
{
    int gotval =
       atomic_inc_return(&mutex);
    if (gotval == 0)
    {
         atomic_set(&mutex, 1);
         stat = futex(&mutex, FUTEX_WAKE,
                              1, NULL, NULL, 0);
     }
}
```

Pop-Up Threads



- Creation of a new thread when message arrives

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

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?

Thread Cancellation

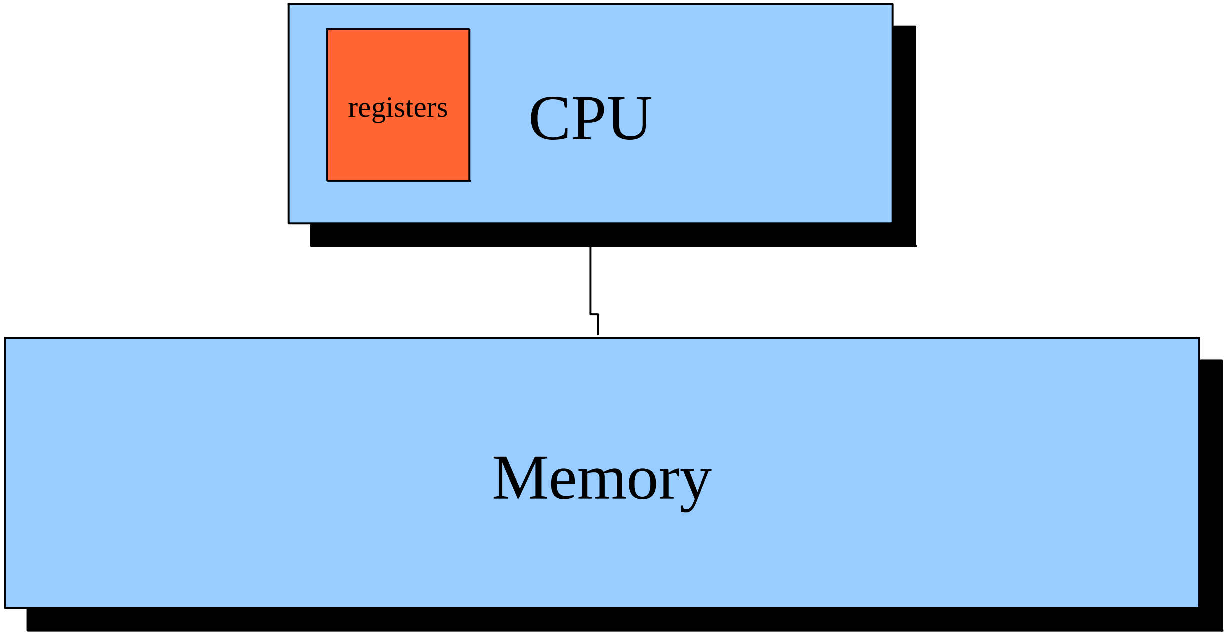
- Terminating a thread before it has finished
- Reason:–
  - Some other thread may have completed the joint task
  - E.g., searching a database
- Issues:
  - Other threads may be depending cancelled thread for resources, synchronization, etc.
  - May not be able to cancel one until all can be cancelled

Thread Cancellation
(continued)

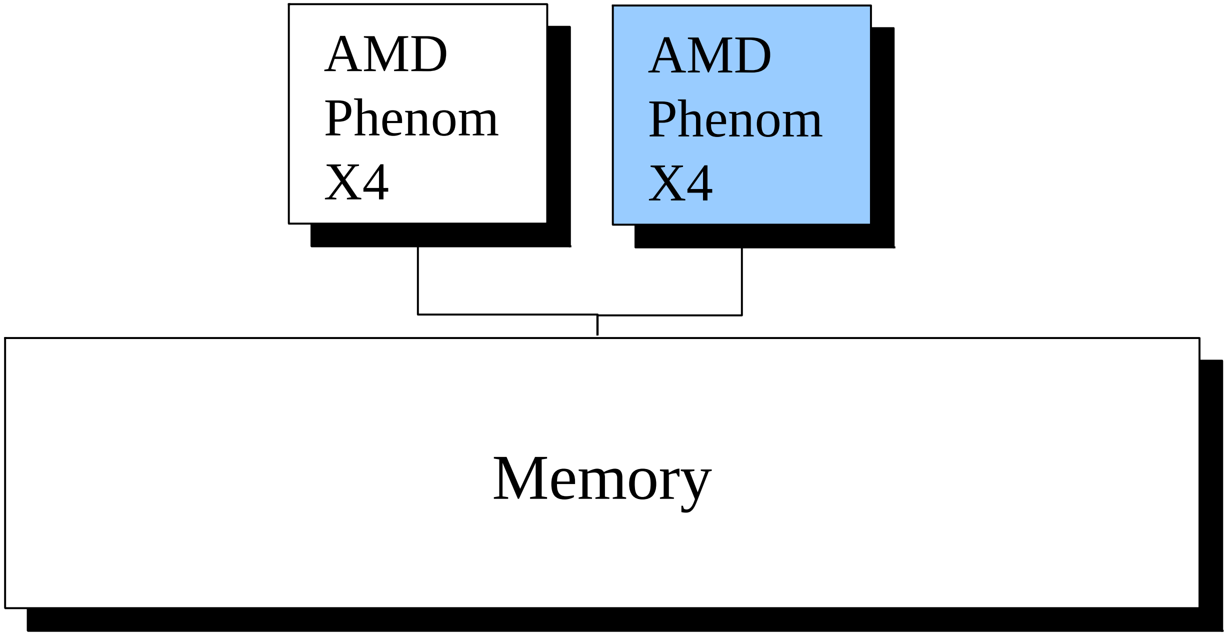
- Two general approaches:
  - *Asynchronous cancellation* terminates the target thread immediately
  - *Deferred cancellation* allows the target thread to periodically check if it should cancel itself

- **`pthreads`** provides *cancellation points*

The simplified memory system



The reality today (simple machine...)

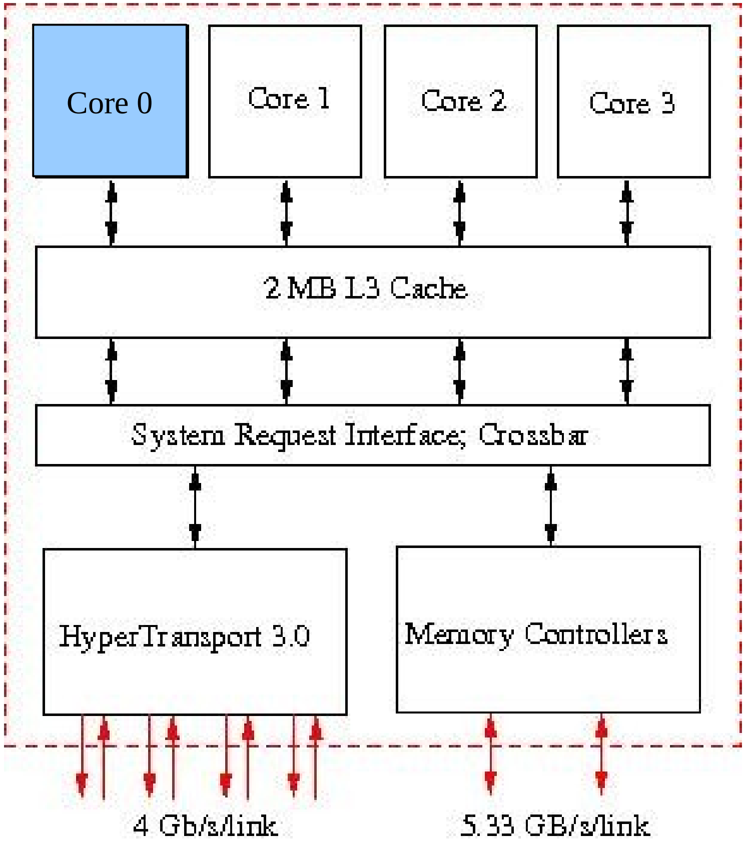


AMD Phenom X4

AMD Phenom X4

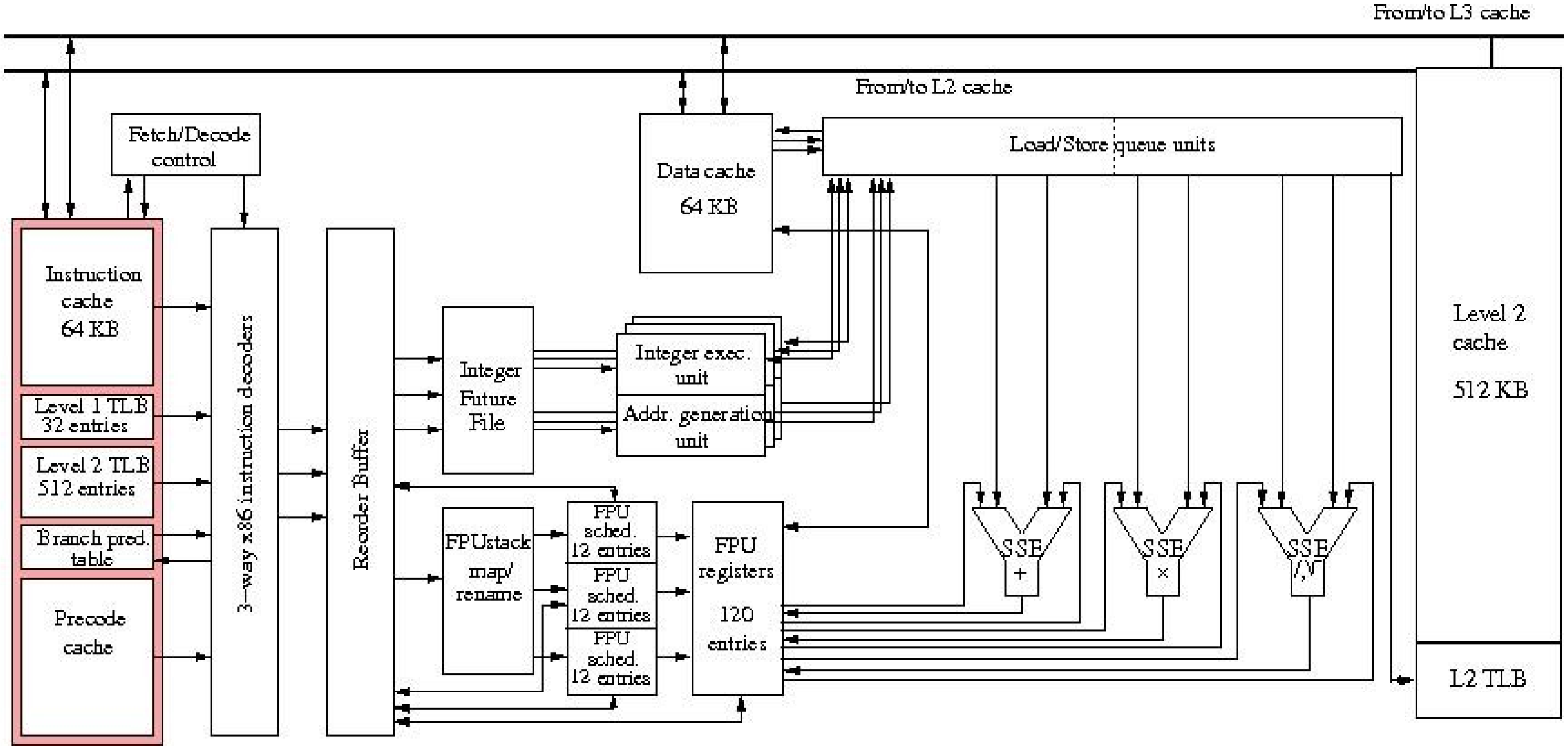
Memory

The reality today (cont.)



Architecture of an AMD Phenom X4

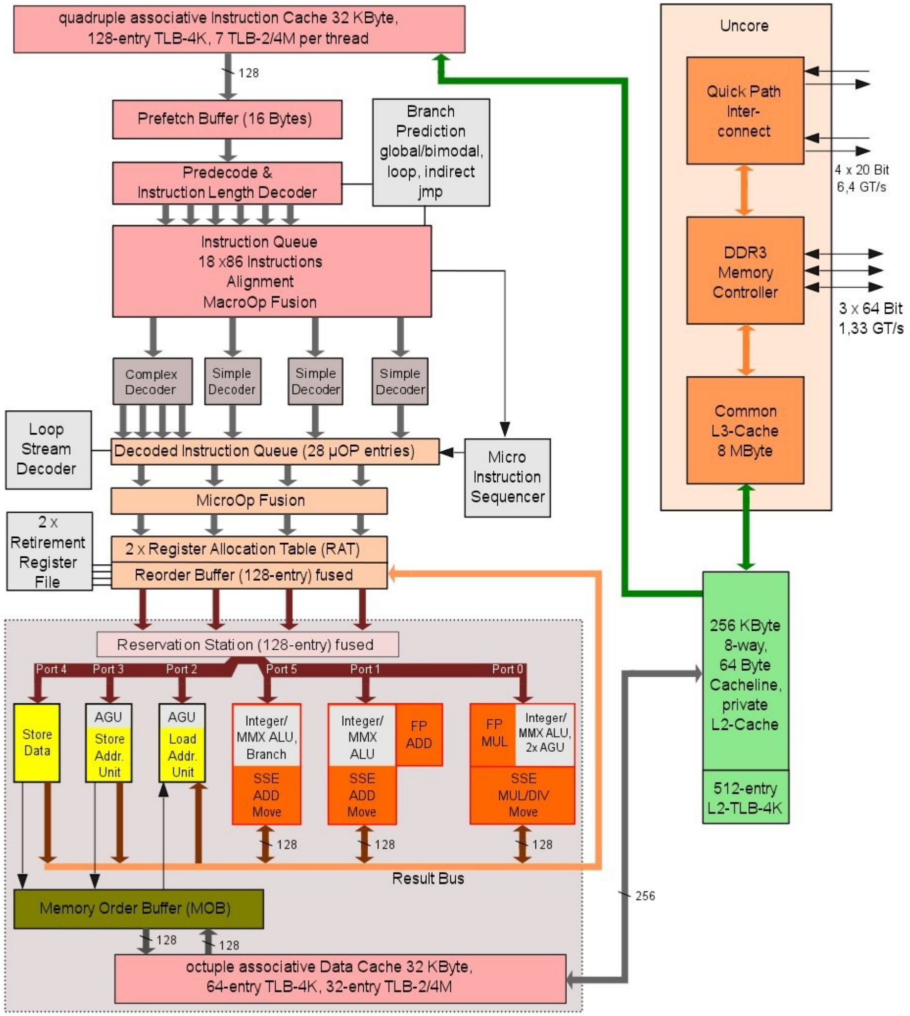
The reality today (cont.)



Internals of an AMD Phenom core

Intel Nehalem (i7)

Intel Nehalem microarchitecture



GT/s: gigatransfers per second

“Shared” memory?

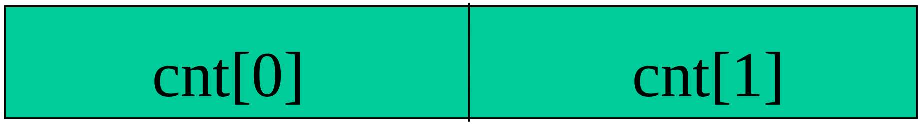
- Processors usually reads/writes caches
- Sharing with another processor/core requires communication
- Optimal performance when optimal communication pattern
- write can be non-blocking
- reads are blocking
- Memory are more or less “local”
- Best if write to 'remote' - read 'locally'

What can we assume about memory accesses?

- “A read from any given address always returns the value of the latest write to that address”
- Read and writes are atomic
- What about order of writes?
- And what goes on during a write?
  - Depends on consistency model
  - Varies between CPUs and memory architectures and settings..

False Sharing

Int cnt[2];
A: cnt[0]++;
B: cnt[1]++;



cnt[0] cnt[1]

cacheline

A False Sharing test

```
volatile int count[2];

#ifdef FALSE
worker(void * arg)
{
  int i,index=(int)arg;
  for(i=0;i<100000000; i++)
    count[index]++;
}
#else
worker(void * arg)
{
  int i,index=(int)arg;
  int temp=0;

  for(i=0;i<100000000; i++)
    temp++;

    count[index]+=temp;
}
#endif
```

```
main()
{
  pthread_t t;

  pthread_create(&t,NULL,worker,NULL);
  worker((void *)1);

  printf("%d %d\n",count[0],count[1]);

}
```

```
False Sharing
13.190u 0.020s 0:06.79 194.5%   0+0k 0+0io 245pf+0w

No False Sharing
  2.690u 0.000s 0:01.36 197.7%    0+0k 0+0io 245pf+0w
```

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

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

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

Context switch overhead

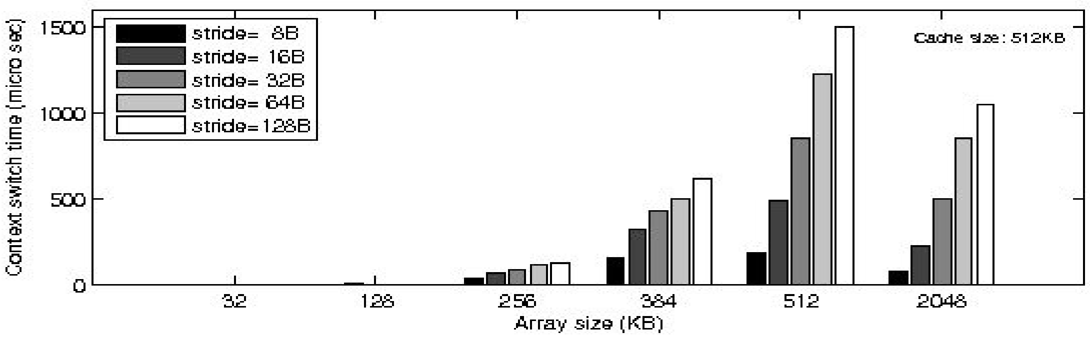


Figure 2: The effect of the access stride on the cost of context switch

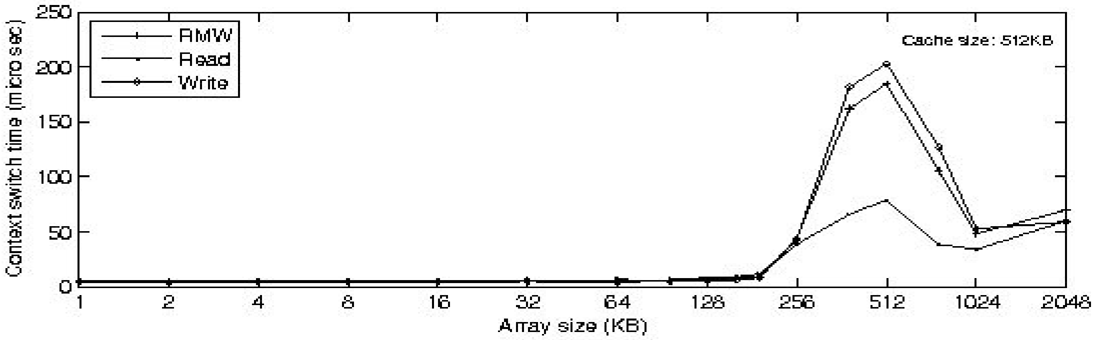


Figure 1: The effect of data size on the cost of the context switch

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!

Safe interprocess communication using shared memory

Based on trust - no way to stop ill-behaved threads!

- Contract between participating threads about usage of memory locations
  - mutual exclusion by means of locks or monitors (condition queues guarded by locks)
  - transactional: do something then rollback if someone else appeared to do it first
  - single writer/single reader schemes:

(efficient message passing in shared memory)

Safe interprocess communication – some important issues:

- Murphy’s law for parallel programming:
  - *Anything that can go wrong will eventually go wrong!*
- No assumptions about thread speed (time independence)
  - “Ole-Johan’s semicolons” – the semicolon where it all may go wrong...
- Forward progress
  - All threads must at some point in time be able to continue
- With preemptive scheduling:
  - a thread might lose control at any point!

Mutual exclusion principle

1. lock(A);
2. <read/modify state protected by A>;
3. unlock(A);

No more than 1 process executing between line 1 and 3 in any case.

- all others must wait

Mutual exclusion

Principle: Serialize access to resource

- self imposed protection

Key issues:

- Protection of data structure rather than code segments!
- Partial monotonic ordering of locks in a system must not be violated!
- Interrupts is a source of problems if not properly implemented!

Mutual exclusion lock usage:

watch out for the implicit partial order between locks!

```
lock(A)
  lock(B)
  unlock(B)
unlock(A)
...
lock(A)
   lock(C)
   unlock(C)
unlock(A)
```

A, B, C part of partial monotonic ordering of all locks

A > B

A > C

means

A must always be grabbed outside of C (parenthetically) if they are to be held simultaneously!

no relation between C and B yet.

Monotonic ordering of locks – why?

Process 1:

```
lock A
lock B
unlock B
unlock A
```

Process 2:

```
lock B
lock C
unlock C
unlock B
```

Process 3:

```
lock C
if <some rare case>
  lock A
  unlock A
fi
unlock C
```

Time independence:

“suppose we have this fast process and this other slow process…”

process 1: (inc, dec: atomic ops)

```
if (!o)
   lock(olock)
   if (!o) o = new object;
   unlock(olock);
inc(o.users);
<using o>
...

dec(o.users);
if (o.users == 0)
   lock(olock);
   if (o.users == 0)
      delete o; o = NULL;
   unlock(olock);
```

Can you see any problems with this algorithm??

Forward progress

- spin lock L:

```
lock L
<use R>
unlock L
```

Is forward progress ensured for all threads calling this code?

Mutual exclusion: drawbacks

- Contention for locks: not very scalable
- Modern architectures:
  - fine grained sharing not good for memory system – cache line ping-pong/false sharing common!
- serialization – tight synchronization
  - critical regions must be kept small to reduce chance of contention!

Transactional memory – "non-blocking synchronization"

- Assumes compare&swap

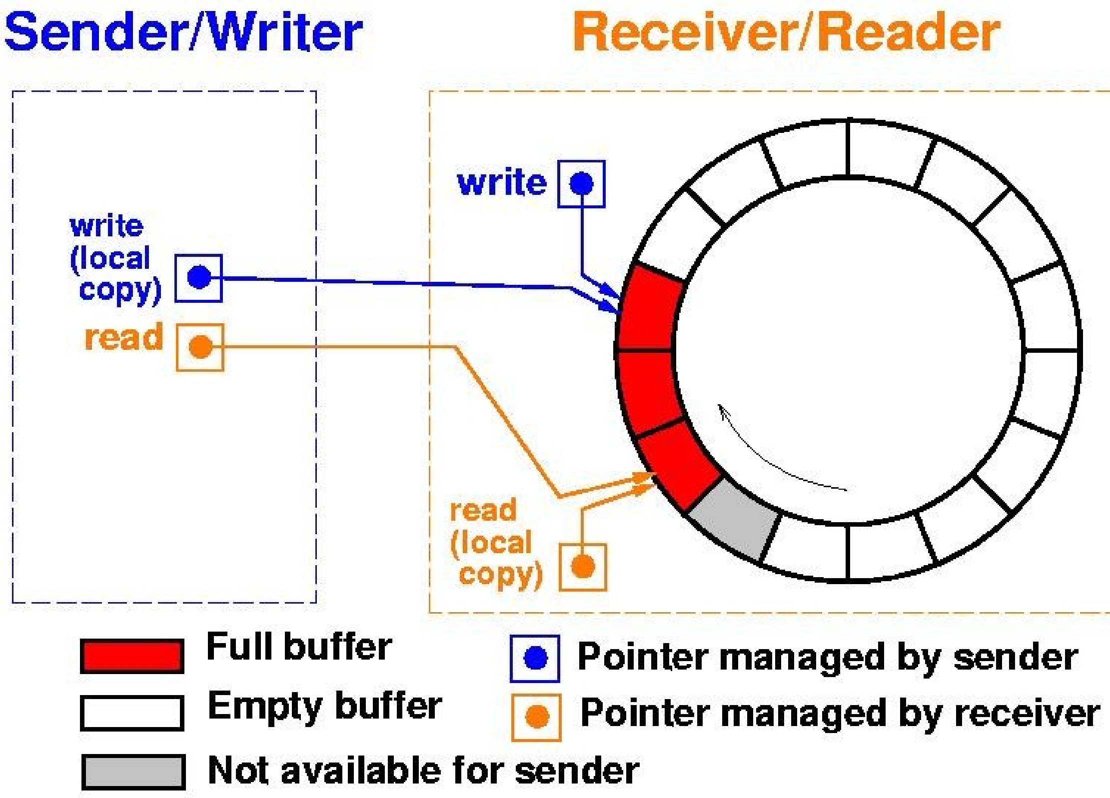
Optimistic approach:

- "usually I am the only one to acquire a resource, recover if someone else appeared to be first"

1. <read/modify state "protected" by A>;
2. commit/rollback(A);

Single writer/reader exclusive read pointer

- Contract: only one process/thread have write access to a particular location:



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

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);
}
```

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;
}
```

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....!

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!

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... ☺