Parallel Programming (3)

Shared Memory

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

This does NOT repeat previous information
There are a few reminders, but that is all

It goes into more details on shared memory issues

Some of these issues apply to PGAS, too
E.g. memory consistency and synchronisation

Overview

Start with basic concepts and problems

Commonly used shared–memory environments

Some useful techniques and methodologies

Guidelines for practical programming

Memory models and data consistency

Information for debugging and tuning

SMP and Threading

SMP stands for Shared Memory Processor
All threads can access all the memory

- It does NOT guarantee consistency!

We will return to this minefield later

It used to mean Symmetric Multi–Processing
That use still occasionally crops up

Threads share all memory within a process
Can execute in parallel, but that is about all

Other Courses

The Computer Laboratory has a 16 lecture course:

http://www.cl.cam.ac.uk/teaching/1314/...
...../ConcDisSys/

8 lectures on explicit shared–memory threading

Non–trivial and not generally advised but, if you do:

- Allow 30–50 hours for practical work

Key to Success (1)

General threading is too hard for mere mortals
See Hoare's Communicating Sequential Processes!

- First key to success is a good design paradigm
Nowadays, often called a programming pattern

- Second key to success is strict discipline
Important for all programming, critical for this

- Incorrect programs often seem to work
Pass all testing, nothing flagged by debuggers etc.
⇒ But, in real use, often get wrong answers

Key to Success (2)

Will describe why this is so hard, later

- Even top experts don't trust their intuition

For some code, possible to prove correctness
But not feasible for most practical programs

- So solution is to use a proven design

And program within the restrictions of that design
Use of patterns was described in previous lecture

⇒ Remember data races are the main problem

Reminder: Data Races

If two threads/processes access same location:

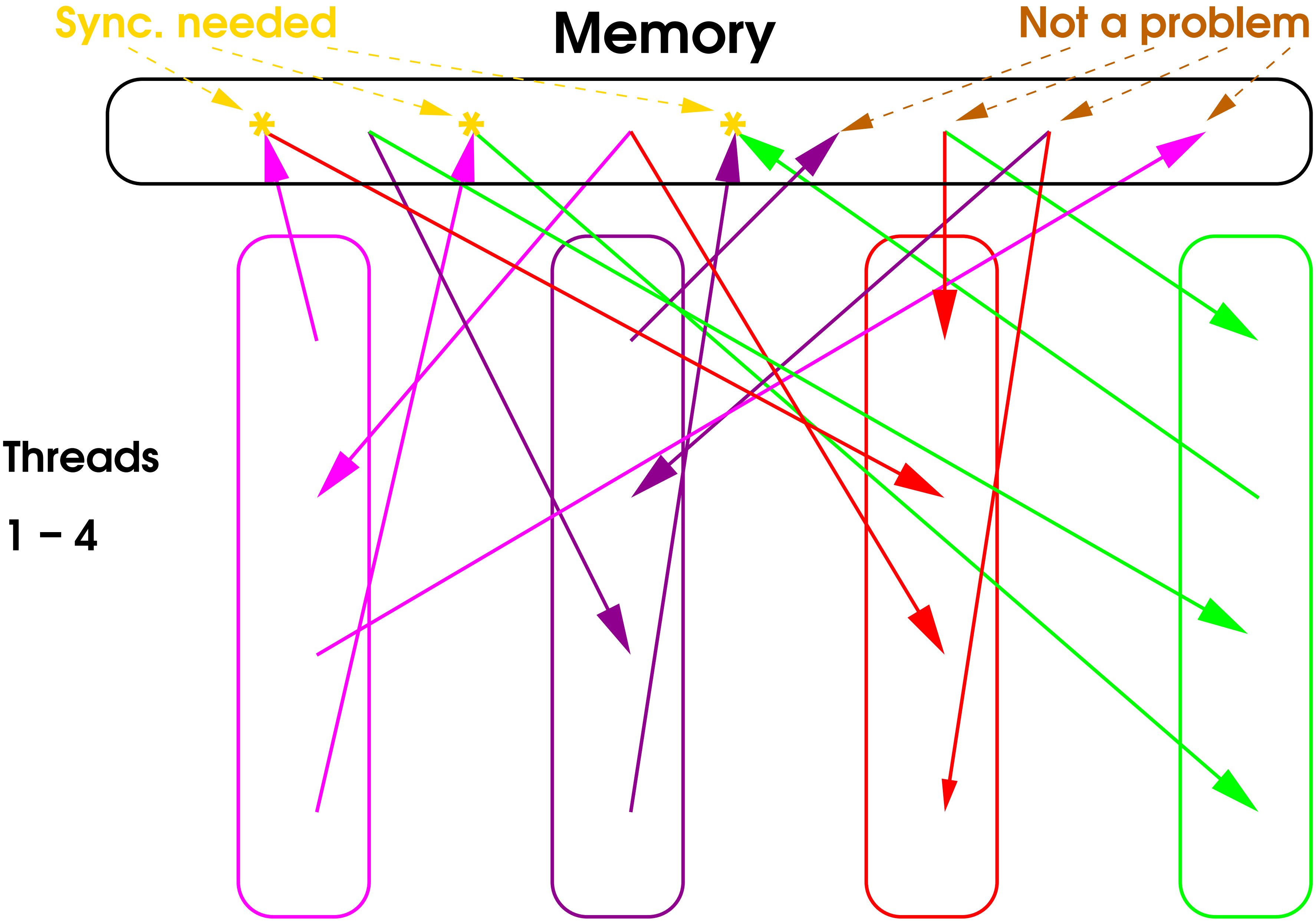
- Either all accesses must be reads
- Or both threads must be synchronised
- Or all accesses must be atomic or locked

Details depend on the interface you are using

- Critical to read specification carefully

Consistency rules are the memory model problem
Will return to that towards the end

Shared–Memory Threading



Sync. needed

Memory

Not a problem

Threads

1 – 4

When Updates Happen

Updates may not transfer until you synchronise
But they may, which is deceptive

Memory will synchronise itself automatically

- Now, later, sometime, mañana, faoi dheireadh

So incorrect programs often work – usually
But may fail, occasionally and unpredictably

Makes it utterly evil investigating data races

- Any diagnostics will often cause them to vanish

C++ and Threading (1)

- This applies to ALL threading in C++

C++ and threading do not mix at all well
⇒ Don’t underestimate the trickiness of this area

E.g. constructors, destructors and copy assignment
may be called when you don’t expect

Also C++ 11 changed the allowable optimisations
Code reordering interacts subtly with threading

C++ and Threading (2)

- Worse, templates are like macros on steroids
Specification depends on the implementation (sic)

- Biggest problem is containers and similar
No explicit specification of links with memory model

- Formal rules imply they are almost unusable
People rely on the ‘‘But everybody knows’’ rule
Requires undocumented interpretation of wording

- Problems more likely to appear in optimised code
Very rarely bugs in the compiler’s optimiser

C++ and Threading (3)

Some fairly safe rules in lecture 7 of:
OpenMP/

- Reliable for GNU/Intel on Linux and x86 – for now

Less so on other systems, if optimised or in future
You are on your own with libraries like Boost

If you have trouble, do not reduce optimisation, but:

- Simplify and clean up your code
- Reduce updates to shared containers etc.
- See if explicit synchronisation helps

Status Report

Have covered basic concepts and problems

Now commonly used shared–memory environments

Fully Shared Memory

Mainly OpenMP, Java and POSIX/Microsoft threads

- Most SMP libraries implemented using OpenMP

See later about the consequences of this

- But C++ 11 has now added its own threading

CilkPlus is a promising new model for C++

Also a lot of specialist and potential ones
A very active area in computer science research
⇒ Expect new environments all the time

OpenMP (1)

A language extension, not just a library

Designed by a closed commercial consortium

“Open” just means no fee to use specification

Dating from about 1997, still active

http://www.openmp.org

Specifications for Fortran, C and C++

Most compilers have some OpenMP support

- This is the default to use for SMP HPC

Unfortunately the specification is ghastly

OpenMP (2)

- The compiler handles the synchronisation
Covers up problems in underlying implementation
E.g. ambiguities in the threading memory model

- Mainly directives in the form of comments
They indicate what can be run in parallel, and how
Also a library of utility functions

OpenMP permits (not requires) autoparallelisation
I.e. when the compiler inserts the directives
Available in many Fortran compilers, less so in C

OpenMP (3)

- Easiest way of parallelising a serial program
Can just modify the areas that take the most time

- Can usually mix SMP libraries and OpenMP
Start with adding calls to parallel library functions
And set compiler options for autoparallelisation

- Then use SIMD or SPMD directives
Finally, worry about more advanced parallelism

Too good to be true? I am afraid so, but still useful

OpenMP (4)

- Inserting directives trickier than it seems

Make even a minor mistake, and chaos ensues

There is a two–day course on OpenMP
OpenMP/

- Teaches SIMD and ‘pure’ SPMD mode

Limited synchronization, locking or atomic
Will get the best diagnostics and other help

- One lecture on tasking, but not always given

C++ Threads (1)

- Latest C++ 2011 standard defines threading
The design is OK, but some compilers don’t have it
Memory model has been proven to be consistent

- Ordinary memory accesses must not conflict
Atomic memory accesses impose synchronisation

- Default is sequentially consistent – so not scalable
More scalable uses for expert masochists only

- Basic facilities are very low–level, confusingly so
Usable only by experts writing simple primitives

C++ Threads (2)

- Some bad ideas – e.g. cross−thread exceptions

Avoid fancy ones (e.g. memory_order, try_lock)
Issues too complicated for this course

Inherited C facilities are full of gotchas

- May not behave as you expect (i.e. not work)

Includes all I/O – so synchronise each stream

- Library has only very low−level facilities

Containers etc. are still entirely serial

Issues with those were described above

C++ Threads (3)

- Using it will be harder than using OpenMP
The C standard is copying it – if anyone cares!

- I really don't recommend it for most people
Except for small, encapsulated primitives

- Boost has its own threading, but don't use it
No memory model and superseded by standard
Same applies to most other C++ threading libraries

⇒ If you must use raw threading, use C++ 11

Java Threads

The first version was a failure, and was redesigned
Memory model was inconsistent and almost unusable

⇒ Yes, experts have trouble with this area!

Reports of the merits of the second one are mixed

http://docs.oracle.com/javase/tutorial/...
.../essential/concurrency/

C#

This was ‘fast tracked’ into a standard
I.e. essentially no external technical debate over it

The threading facilities are very low–level
Plus some rather crude loop parallelism

volatile accesses in a single thread are ordered
⇒ But with NO cross–thread consistency

- That is almost unusable – see later for why

Microsoft Threads

In 2007, finding the API was easy enough

- But I failed to find a proper specification

Or even a decent tutorial, like Java's

Searching threads "memory model"
From *.microsoft.com had 167 hits, but ...

I was told that it was then about to change

- In 2014, it seems to be specified only for C#

POSIX (1)

- C90 and C99 are entirely serial languages
  POSIX is inconsistent with C standard

- Legal C optimisations break POSIX threads
  Often need to disable optimisation to make them work

- Neither C99 nor POSIX defines a memory model
  Reasons why one is essential are covered later

http://www.opengroup.org/onlinepubs/...
.../009695399/toc.htm

POSIX (2)

- No way of synchronising non–memory effects
Even simple ones like clock() values and locales

- I/O is said to be thread–safe – NO, IT ISN’T!

- Some aspects (e.g. signals) are simply broken
They are implementation–dependent and unreliable

- Microsoft threads even less well specified

Solution is to be very defensive – see later

Other Threading

Can use raw threads in OpenMP if you really insist
CilkPlus was covered in previous lecture

DON'T use the latest C standard (C11)
Almost nobody is interested in new C standards
⇒ Will be untested and unreliable, at best

Many other languages have threading, too

Ada and Haskell said to be fairly good
I haven't looked at them in any depth

Status Report

Done commonly used shared–memory environments

Now some useful techniques and methodologies

Synchronisation

- KISS – Keep It Simple and Stupid

If you need separation or ordering, code it

- Barriers and reductions are by far the easiest

One–sided synchronisation is much trickier

- Critical sections / locked transactions are next

But fancy use of them can be quite tricky

- Whether atomic is easy depends on consistency

Beyond that, it gets rapidly harder

Barriers (1)

- Barriers are multi–thread synchronisation
All threads (or a subset) execute the same barrier
All involved threads synchronise data, and restart

- Much the simplest when synchronising all threads
Data should be write once or read many
If a thread updates data, no other thread accesses it

- Watch out if using barriers on a subset of threads
Any access on remaining threads can be a data race
Also consistency issues, just as for pairwise methods

Barriers (2)

P = Q = R = S = T = 0

Thread A		Thread B
P = 1 → OK		Q = 2 → OK
print S → 0		R = 4 → OK
U = 8 → OK		print T → 0
Barrier	===	Barrier
print Q → 2		print P → 1
R = 5 → OK		U = 9 → OK
T = 7 → OK		S = 6 → OK

All statements in top half precede all in bottom half

Reductions (1)

A special, easy to use, form of atomic update
E.g. accumulating a sum, but also other operations
They are also likely to be more efficient

- Initialise the accumulator before a barrier

- Accumulate updates during the parallelism

But do not use its value or do anything else with it

- Read the accumulator only after next barrier

Reductions (2)

P = Q = 0

Thread A		Thread B		Thread C
Barrier	===	Barrier	===	Barrier
P += 12		Q *= 23		P += 45
Q *= 67		P += 89		P += 1
Barrier	===	Barrier	===	Barrier

Print P, Q

Transactions (1)

A very important technique in parallel programming
Just a compound action packed to be atomic

- Makes it much easier to avoid data races

- A transaction must include all data accesses

Only exception is for globally read–only data
Yes, a read–only transaction can often be needed

Can be implemented in many ways, not covered here
Main ones include locking and retry on conflict

Critical Sections

A structured form of locked transactions
Can also code using explicit locks – e.g. in OpenMP

```
CRITICAL      ! Use a hidden lock
        < Perform actions >
END CRITICAL     ! Unlock it
```

```
omp_set_lock ( & lock ) ;
        < Perform actions >
omp_unset_lock ( & lock ) ;
```

C++ Example

```
typedef vector<double> dp ;
dp arr(7 , 0.0) ;
```

Thread A	Thread B	Thread C
Lock	Lock	Lock
cout << dp[3] ;	dp[3] = 7 ;	arr = dp(5 , 2.3) ;
Unlock	Unlock	Lock

- Note that the first action is read–only

The order the sections execute is unpredictable

Transactions (2)

- Lock all parallel accesses to objects

Using a single section name or global lock is easy
All accesses will be sequentially consistent

- You usually need multiple locks for efficiency

Data consistency is not guaranteed
And nested locks will often cause deadlock

Details depend on language and parallel mechanism
Can give only some rough guidelines here

Transactions (3)

Generally, in order to minimise consistency problems:

- Access all related data under a single lock
- Keep data under separate locks independent

Can also access data used only by that thread
Simplest approach is often to copy in and copy out

Use other global data only if globally read–only

- Being read–only under the lock is not enough

Invalid Transaction Pairs

Using lock A in thread X: analyse(limit)
Using lock B in thread Y: limit = calc()

Using lock A in thread X: analyse(limit)
In unlocked code in thread Y: limit = calc()

In unlocked code in thread X: analyse(limit)
Using lock B in thread Y: limit = calc()

Using lock A in thread X: lwb = upb – delta
Using lock B in thread Y: upb = lwb + delta

One-Sided Synchronisation (1)

Other mechanisms include mutexes, semaphores,
condition variables, fancy locking etc.
Very environment–dependent but, in summary:

- Almost all synchronise only a pair of threads

And they are almost always one–sided – i.e.:
execution before the mechanism on one thread
precedes execution after it on the other

- Tricky to use when several threads are involved

Data consistency (see later) often not as expected
Can usually assume mechanisms are transitive

One-Sided Synchronisation (2)

P = Q = R = S = T = 0

Thread A		Thread B
P = 1 → OK		Q = 2 → undefined
print S → 0		R = 4 → undefined
U = 8 → OK		print T → undefined
Mechanism	⇒⇒⇒	Mechanism
print Q → undefined		print P → 1
R = 5 → undefined		U = 9 → OK
T = 7 → undefined		S = 6 → OK

Statements in top left precede those in bottom right

- There is NO ordering top right to bottom left

Transitivity

P = Q = 0

Thread A		Thread B		Thread C
P = 1				
Mechanism	⇒	Mechanism		
		Q = 1		
		Mechanism	⇒	Mechanism
				print P

If it is guaranteed to print 1, it’s transitive

- Almost always the case, but check specification

Low-level Synchronisation (1)

- Underneath, they use fences to synchronise data

These bring memory up–to–date on one thread
The mechanism ensures that the fences are ordered

Thread A calls a fence and then sends a message
Thread B receives the message and calls a fence

The message is usually internal to the mechanism
In such cases, it can be anything (e.g. a signal)

- You don’t need to worry about what it is

Low-level Synchronisation (2)

Thread A		Thread B
Fence		
Send	⇒	Receive
		Fence

- Using fences yourself is generally not advised
They are tricky to get right, and truly evil to debug
You must use above logic to use them correctly

Warning: there are also release/acquire fences
Makes writes visible to reads – no more!
They are fiendishly hard to use correctly

Release/Acquire Synchronisation

P = Q = R = S = T = 0

Thread A		Thread B
P = 1 → OK		Q = 2 → undefined
print S → undefined		R = 4 → undefined
U = 8 → undefined		print T → undefined
Mechanism	⇒⇒⇒	Mechanism
print Q → undefined		print P → 1
R = 5 → undefined		U = 9 → undefined
T = 7 → undefined		S = 6 → undefined

Writes in top left precede reads in bottom right

- There is NO other ordering

Status Report

Done some useful techniques and methodologies

Now onto guidelines for practical programming

Atomic (1)

- The term atomic is seriously ambiguous

It means an action happens completely or not at all

But, if two atomic updates happen at the same time,
Can one of them simply disappear into thin air?

Even on one variable, are they always consistent?

- For multiple variables, the answer is usually NO

- It is critical to know the precise specification

- And it may be efficient or highly inefficient

Atomic (2)

The following is what you can usually rely on:

- Synchronisation also affects atomic accesses
The rest of this is about unsynchronised accesses

- Accessing both atomically and not is undefined
Unless they are properly synchronised, of course
Just having an atomic data type is rarely enough

- There may be no consistency between variables
Describing the possibilities is too hard for this course

Atomic (3)

- A thread sees its own actions happen in order
Applies even to changes to several atomic variables

- For a single atomic variable:
All threads see a consistent order of changed values

- All updates will happen precisely once
Order is constrained solely by above conditions

- All changes eventually happen (not for PGAS)
An explicit synchronisation will force that

Atomic (4)

Safest uses are (in decreasing order of safety):

- Atomic variables that are only written to
Last values can then be read after synchronisation

- A single thread changes a single atomic variable
All other threads may read its value

- Any thread can write to a single atomic variable
Only a single thread will read its value

⇒ Check the specification of your environment

Implicit Atomicity

I.e. for loads and stores on hardware data

- The answer is sometimes – it’s not simple

Probably OK, IF <type> is scalar and aligned
But are you sure that they have the right alignment?

Available types depend on compiler and hardware

- Never structures (e.g. complex), and
  not always floating–point or even pointers

- FAR better to use explicitly atomic operations

Heterogeneous Accesses

Killer is different methods of access of any form

- Don't mix atomic actions and other access

- Don't mix external transfers and atomic access

Includes RDMA, GPU transfers and even I/O

- Don't even mix scalar and vector accesses

SSE is consistent, now, but used not to be
And I have used other vector extensions that weren't

Shared Memory I/O (1)

- POSIX is seriously self–inconsistent
I/O is thread–safe (2.9.1) but not atomic (2.9.7)
Can you guess what that means? I can’t

- SIGPIPE etc. may go to any thread
Not necessarily all threads – it’s system–dependent

- dup’ed descriptors share only some state
Sockets and other FIFOs are even more complex

Shared Memory I/O (2)

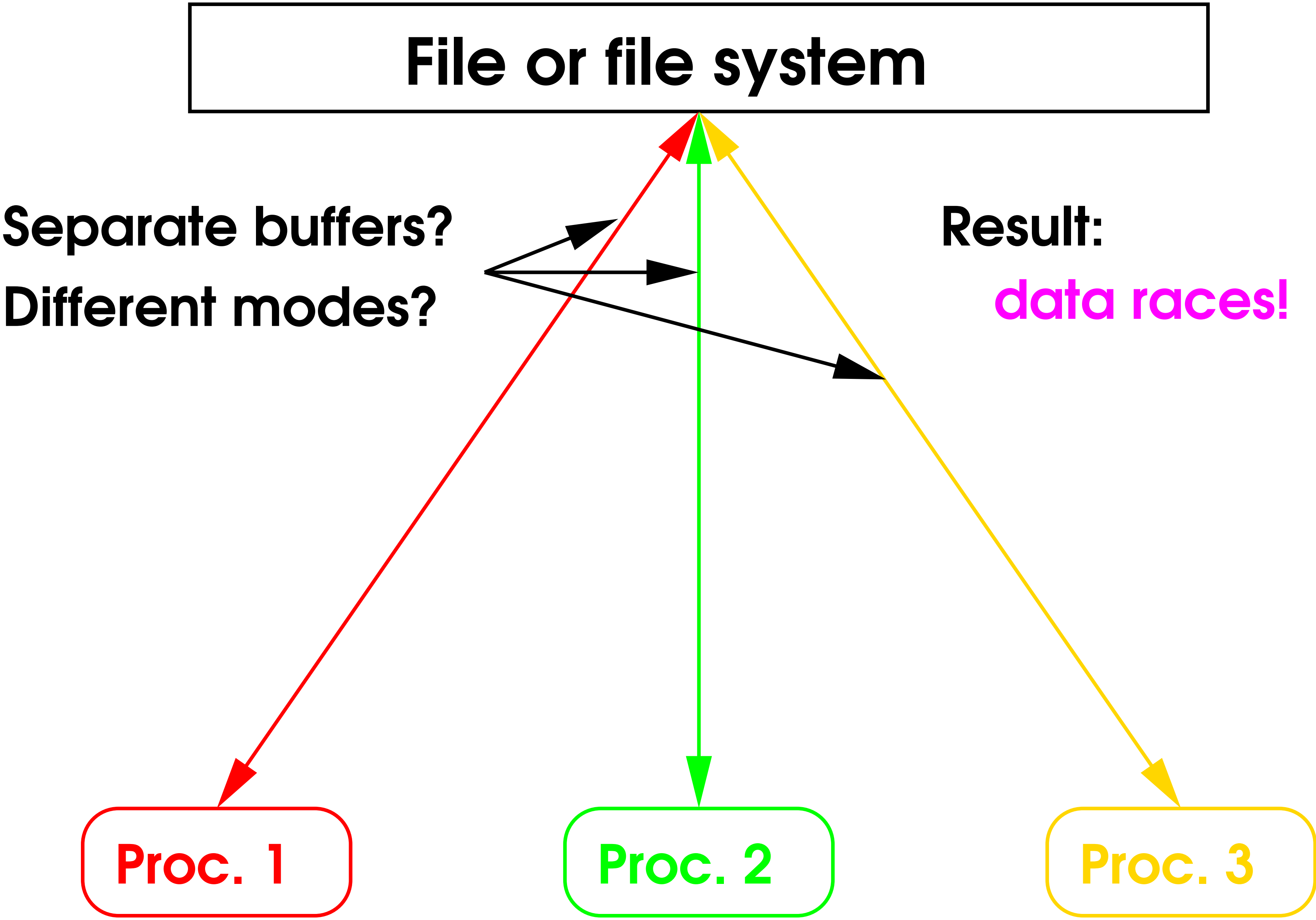
Most other interfaces are built on top of POSIX
C++ and Java are a bit better, but not much
But essentially all have the following problem:

- Each thread may use temporary buffers

Means I/O transfers interleave in very strange ways
This is typically a low–probability phenomenon
Shows up mainly in large files or heavy I/O use

MPI users see this effect much more often

Shared I/O Descriptors



Shared Memory I/O (3)

An example of the problem:

Thread A	Thread B
print "Oh, joy"	Print "Oh, heck"

Could print OOh, heh, jocky or other variations

- Input and binary can fail more severely

Serious data and even file corruption are likely

Shared Memory I/O (4)

- Near–bulletproof (i.e. safest) solution:

Do all I/O from the initial thread
Do all state changing (e.g. locales) there, too

But following is also fairly reliable in practice:

- Serialise (lock) all I/O on any single file

- Serialise (lock) all opens, closes and related directory operations (e.g. rename, readdir)

Usually not needed, but these are fairly rare

Shared Memory I/O (5)

- Avoid two descriptors to the same file

Changes to one may or may not propagate

Fairly safe for ordinary disk files if read–only
Both constraints are essential, though

Aside: problems can occur even between processes
for shared descriptors, sockets etc.

Keep It Simple And Stupid

Program and System State (1)

Locales, any modes, setvbuf, setenv and lots more

- Best to set them early in initialisation

I.e. in initial thread, before starting any other threads
Almost all modes are inherited by other threads

- Otherwise synchronised and in initial thread

- Synchronisation alone has lots of gotchas

Don’t assume consistency between two items
See later for this incredibly deceptive minefield

Program and System State (2)

- Related operations may cause race conditions
E.g. getpid/setpgrp and even popen/close

⇒ Above may not work for signal handling
I have had signal races in synchronised code
There be dragons

- See Critical Guidelines in OpenMP course
Similar rules apply to all threading mechanisms

Aside: POSIX allows each thread separate properties
Don’t go there – that way madness lies

Exceptions

SerC++, errno, IEEE 754 or other

- Handle them only in thread that causes them

Any thread switch will corrupt them

- Watch out for implicit thread switching

Can happen in C++, OpenMP and CilkPlus

- Never use setjmp/longjmp etc. across threads

⇒ And leave genuine signal handling to experts

An expert knows how badly it is broken

Status Report

Done guidelines for practical programming

Now onto memory models and data consistency

Consistency Failure (1)

- Consistency failure is a more subtle problem
We assume events happen in some unknown order
This is sequential consistency – no time warps

- That is not guaranteed by modern systems
Different threads can see incompatible event orders
Very similar to same issue in special relativity

- Some environments have “out of thin air” effects
I.e. breaches of causality, which are even worse

Consistency Failure (2)

Too complicated (and evil) to cover thoroughly
Suitable key phrases to look up include:

Data Races / Race Conditions
Sequential Consistency / Causal Consistency
Strong and Weak Memory Models
Mutual Exclusion
Dekker's Algorithm

And dozens of others more – see the references

Main Consistency Problem

Thread 1

A = 1

print B

Thread 2

B = 1

print A

Now did A get set first or did B ?

0 – i.e. A did   0 – i.e. B did

Intel x86 allows that – yes, really

So do Sparc , POWER and ARM

Also Caused by Optimisation

```
Thread A        Thread B
                X = A
A = 1           B = 1
print B         print A
```

Might well be optimised into:

```
Thread A        Thread B
                <register> = A
A = 1           B = 1
print B         print <register>
```

Another Consistency Problem

Thread 1

A = 1

Thread 2

B = 1

Thread 3

X = A

Y = B

print X, Y

Now, did A get set first or did B ?

Thread 4

Y = B

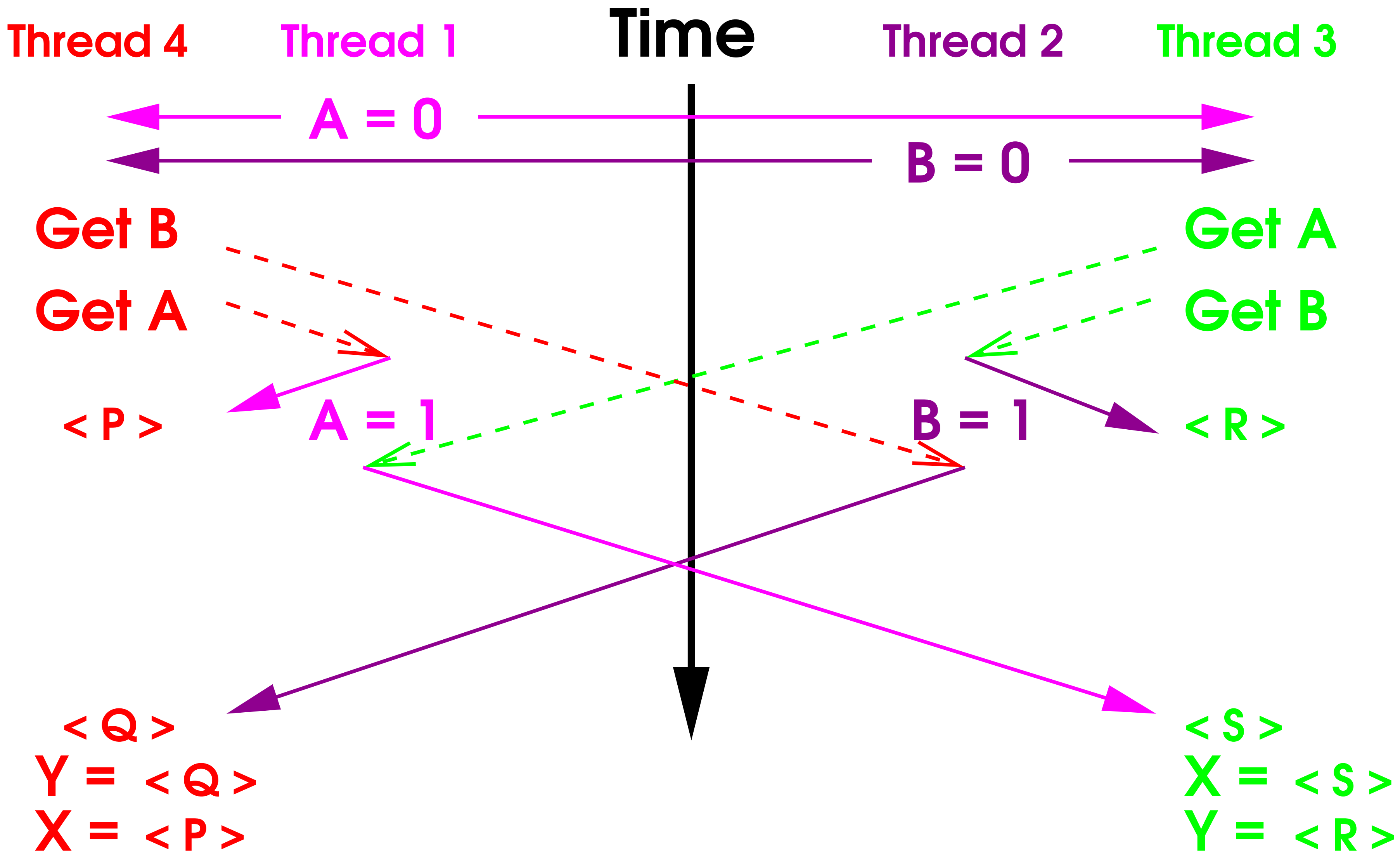
X = A

print X, Y

1 0 – i.e. A did

0 1 – i.e. B did

How That Happens



<x> means a temporary location

Another Consistency Problem

Thread 1

```
print B
A = 1
```

Thread 2

```
print A
B = 1
```

Now did A get set first or did B ?

1 – i.e. B did 1 – i.e. A did

POWER and ARM allow that

Memory Model Issues (1)

Some of you will think that I am exaggerating
But everything I say is based on actual experience

To disbelievers, masochists and mathematicians:

http://www.cl.cam.ac.uk/teaching/1314/R204/

http://www.cl.cam.ac.uk/~pes20/...
...../weakmemory/index.html

Also compiler optimisation can add extra gotchas
Language–, environment– and compiler–specific

Memory Model Issues (2)

Most people will use Intel or AMD CPUs

Intel(R) 64 and IA–32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, 8.2 Memory Ordering

http://developer.intel.com/products/...

.../processor/manuals/index.htm

Follow the guidelines here, and can ignore them

- Start to be clever and you had better study them

Best Approach (1)

- Easiest to use language that prevents problems

No current one does that automatically and safely

- Most can help, if you code in a disciplined way

⇒ This course has described how to do that

Above all, KISS – Keep It Simple and Stupid

- Work with your environment, not against it

Actually, a good rule for all programming

Even Hoare regards this as tricky and deceptive

Best Approach (2)

- Never assume more than explicitly specified

A common aphorism is There Ain't No Sanity Clause
The Damned also made So Who's Paranoid?

Parallelism is very like special relativity
The order of events depends on the observer

- Do not believe claims that it is easy

That is often said on The Web of a Million Lies
And in many books by people who should know better

Status Report

Done memory models and data consistency

Now information for debugging and tuning

Debugging (1)

Frequency of failure is O(N^K) for K $\geq$ 2 (often 3 or 4)
N is the rate of cross–thread accesses
The number of threads is also relevant, non–linearly

- And the word probability is critical here
Almost all data races are not reliably repeatable

- Finding them by debugging is very hard
Diagnostics add delay, and bugs often hide
Often same with debugging options or a debugger

Debugging (2)

Many programs ‘work’ because N is small
The MTBF is often measured in weeks or years
Who notices if a Web service fails 0.1% of the time?

In HPC, there are many more data accesses
The MTBF is often measured in hours or days
Complete analyses often take days or weeks

- Solution is to avoid data races while coding

Not easy, but not impossible with discipline

Debugging (3)

- Variables can change value ‘for no reason’
Failures are critically time–dependent
Serial debuggers often misbehave or even crash

- Even parallel debuggers can get confused
Especially if you have an aliasing bug

- A debugger changes a program’s behaviour
Same applies to diagnostic code or output
Problems can change, disappear and appear

- Try to avoid needing a debugger

Debugging (4)

Memo to self: investigate Intel Inspector XE

That is claimed to detect data races

Tuning Shared-Memory (1)

Tuning is extremely difficult, at best

- Generally best done as part of the design

Minimising access has several aspects:
amount transferred, number of transfers,
waiting for data and conflict

- First problem is that most have several limits

Per core, per socket, per board, ...

- Second problem is that conflict is arcane

Need to be an expert on details of that CPU

Tuning Shared-Memory (2)

There are some tools available
E.g. Intel VTune Amplifier XE, perf (under Linux)

- High−level tools analyse parallelism and waiting
Often some measure of amount of data transfers

- Low−level tools use the performance counters
Simplest when investigating serial performance
Experts can use them to investigate conflict

- NO tools to investigate code overhead
Running in parallel can be much slower than in serial

NUMA and Affinity

Stands for Non Uniform Memory Architecture
In this, ‘closer’ means that access is faster

- Some memory is closer to some cores than others

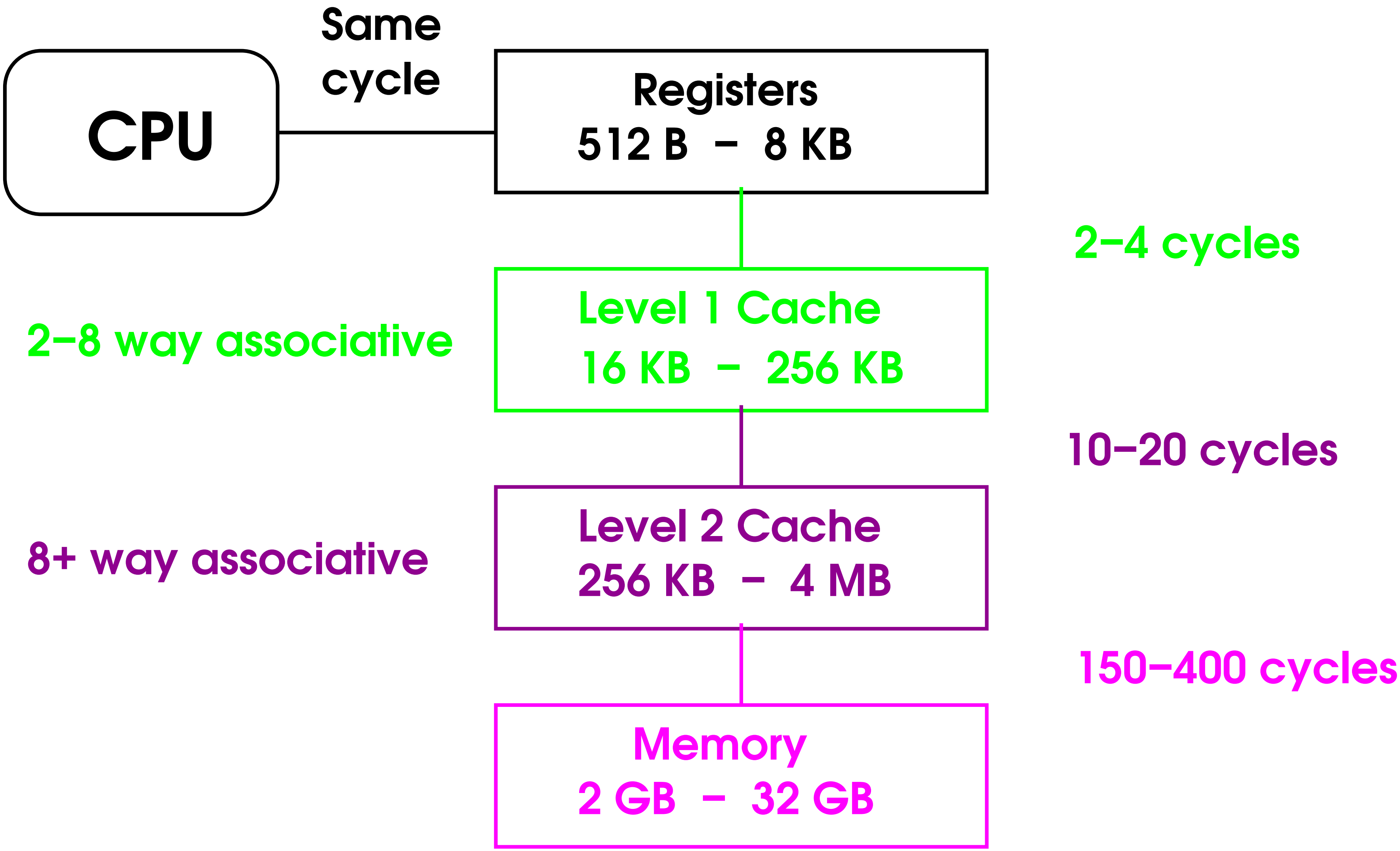
Groups of cores usually share a common access path
The grouping can be different for each level of caching

Also other, more complicated, aspects
Area is far too foul to cover in this course

- Hope that the system’s heuristics guess right

Even administrators have very little control over this

A Typical Cache Hierarchy



CPU

Same cycle

Registers

512 B – 8 KB

2–4 cycles

2–8 way associative

Level 1 Cache

16 KB – 256 KB

10–20 cycles

8+ way associative

Level 2 Cache

256 KB – 4 MB

150–400 cycles

Memory

2 GB – 32 GB

Cache Line Sharing

```
           int A[N];
Thread i:  A[i] = <value_i>;
```

- That can be Bad News in critical code
Leads to cache thrashing and dire performance

Each thread’s data should be well separated
Cache lines are 32–256 bytes long

- Don’t bother for occasional accesses
The code works – it just runs very slowly

Thread Affinity

Can link threads or processes to CPU cores
In Linux, look at the taskset command

- Still has to be permitted by the sysadmin

Also other problems, too difficult to cover here

Rarely useful for embarrassingly parallel problems

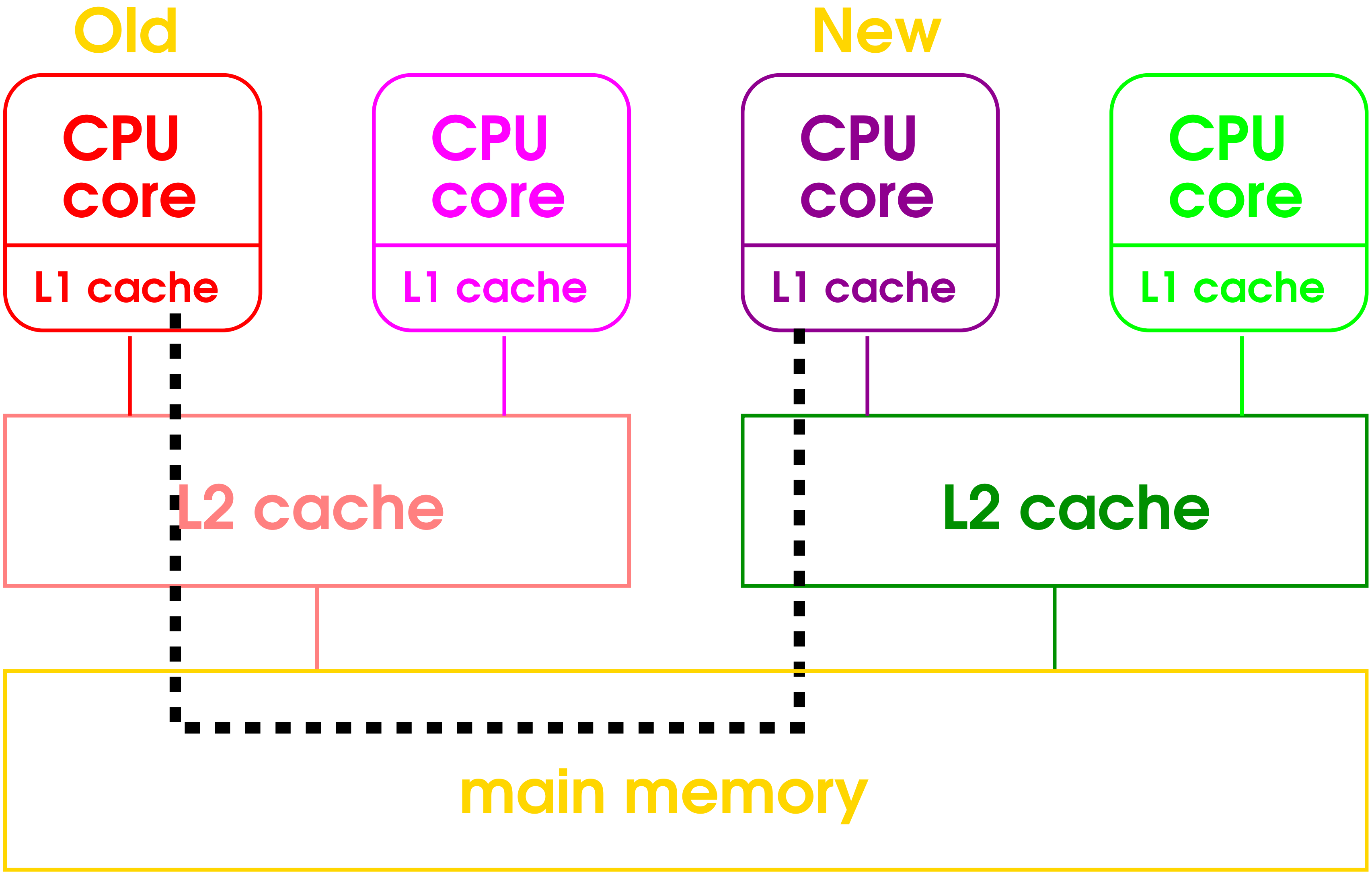
- But can make a huge difference to performance

Sometimes needed to avoid deadlock and livelock

- Transferring between cores can be very slow

Not as bad as distributed memory, but can be close

Moving Ownership



Scheduling Problems

- By FAR the foulest tuning problem
Details are far too complicated to describe here
But one easy solution usually helps!

- Leave some cores free for the system and
Avoid running other daemons, applications etc.

- Using only half the available cores isn't stupid
And don't count hyperthreading etc. in that

- Just adapt the count for best results

The End

Please fill in and hand in the green form