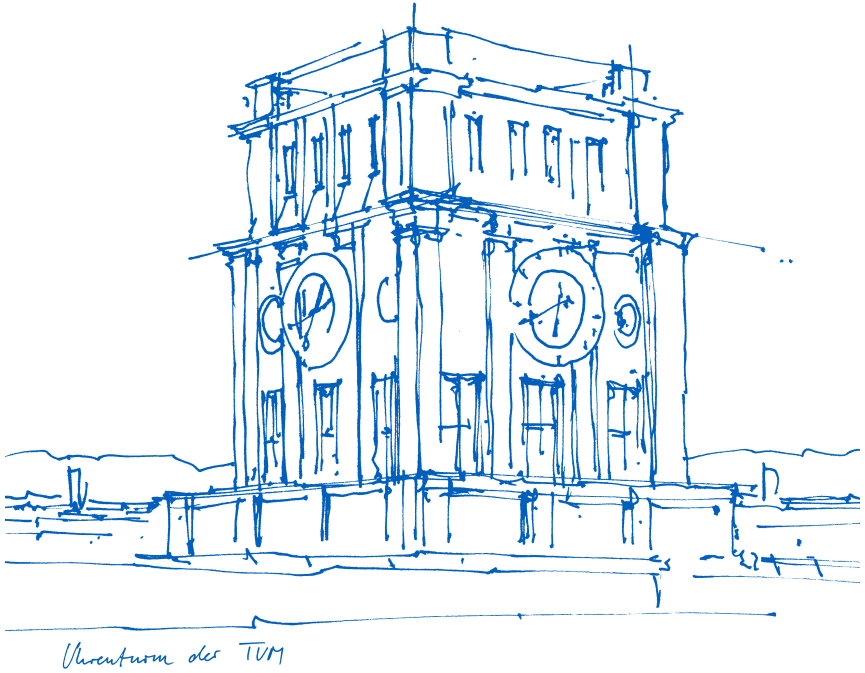
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik



Lecture 2:
Threading Concepts
Threading APIs
POSIX Threads

Practical Information Lecture

Lecture slides will be available after the lectures at

http://parprog.lrr.in.tum.de/ (for now)

Moodle (soon)

Exam

- Final exam will cover lecture and exercises
- 24.07.2017, 16:00 (Please check the final date in TUM-Online)
- Repetition exam tbd.

Do not forget to register for the exam in TUM-Online

Please be interactive and ask questions!

Feel free to send me comments as we go along: schulzm@in.tum.de

Exercises are important!!!

Contents & Schedule (tentative!)

Lectures/Exercises: Monday 16:15, IHS-2

Exercises/Lectures: Wednesday 8:15, MW-0350

	Monday	Wednesday
9./11.4.	Basics / Introduction	Threading / Pthread
16./18.4.	**Exercise**	**Exercise**
23./25.4.	OpenMP Basics	Shared Memory / Dependencies
30.4./2.5.	**Exercise**	**Exercise**
7./9.5.	OpenMP Advanced	**Exercise**
14./16.5.	HPC Architectures and Concepts	**Exercise**
21./23.5.		**Exercise**
28./30.5.	MPI Basics	**Exercise**
4./6.6.	Distributed Memory / Networks	MPI Advanced
11./13.6.	**Exercise**	Tuning and Tools
18./20.6.	Scaling / Mapping	**Exercise**
25./27.6.	**Exercise**	**Exercise**
2./4.7.	Accelerator/GPU Programming	Tasks/PGAS/Future Trends
9./11.7.	**Exercise**	**Exercise**

Summary From Last Time

Parallel processing

- Multiple tasks working together to finish a (a) larger problem (b) faster
- Goal has to be efficiency

Parallelism is becoming more and more common place

- No longer niche HPC
- Multi-/Many-core developments catapult this to every system

Programming in parallel

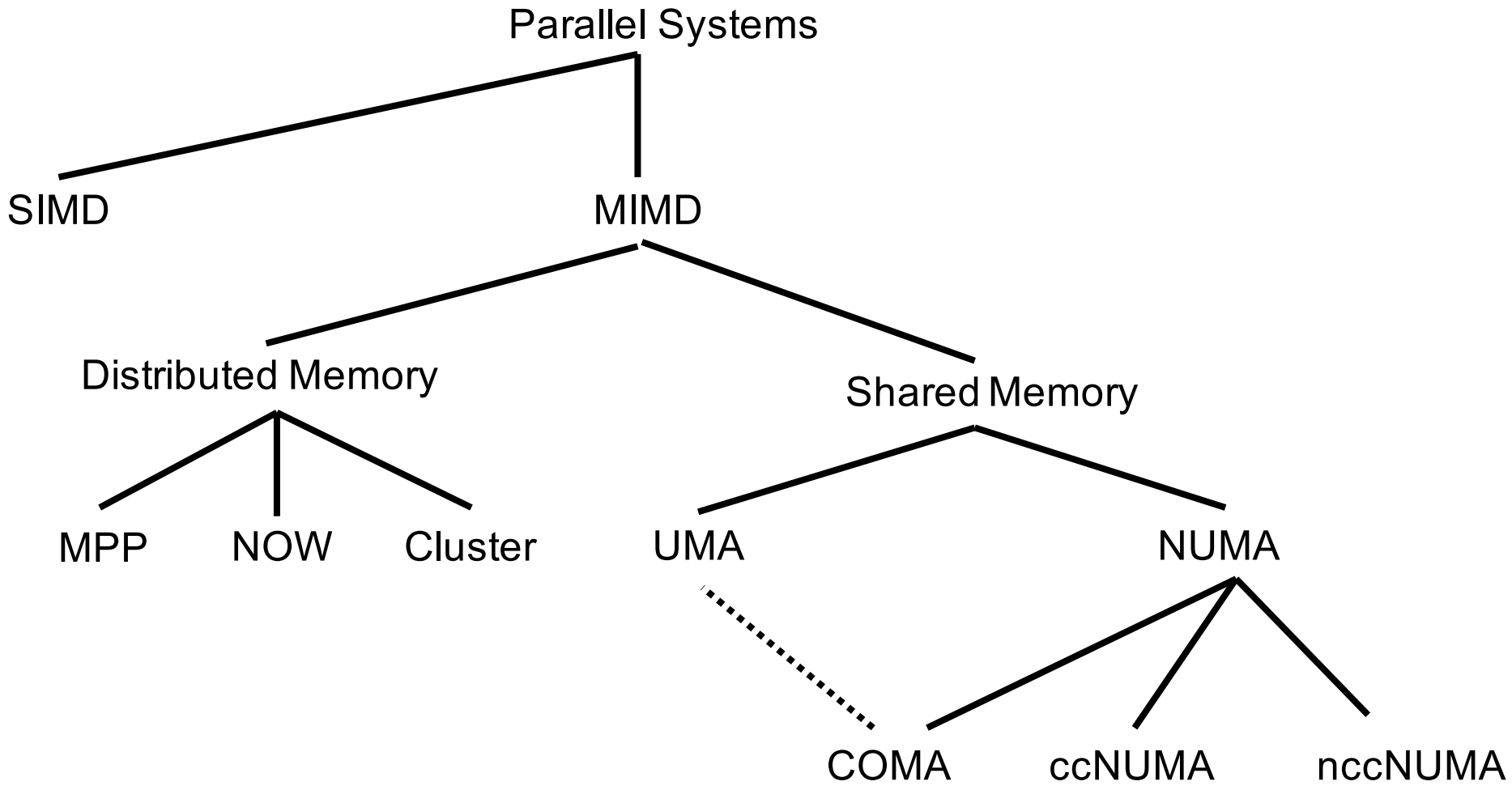
- Decomposition of work and data using choice of best fitting pattern
- Mapping to architectures critical

Wide choice of parallel programming models

- Strong connection to underlying architectures, but not 1:1 match
- Hybrid programming is becoming the norm

Think parallel!

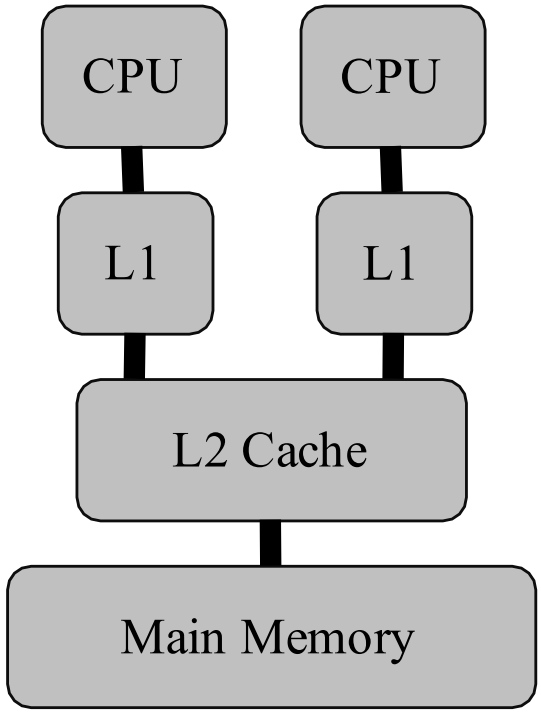
Classification



Shared Memory

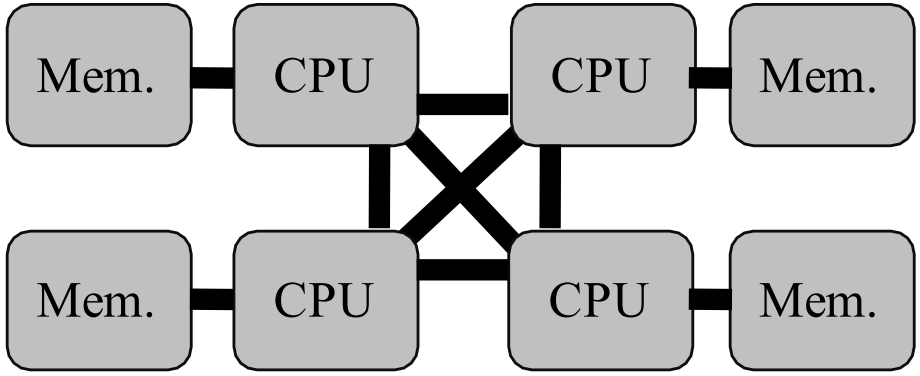
Uniform Memory Access – UMA :
(symmetric multiprocessors - SMP):

- Centralized shared memory
- Accesses to global memory from all processors have “same” latency.
- Transition from bus to crossbars



Non-uniform Memory Access Systems - NUMA
(Distributed Shared Memory Systems – HW-DSM):

- Memory is distributed among the nodes
- Local accesses much faster than remote accesses.



Shared Memory Models Match Shared Memory

Assume a global address space with random access

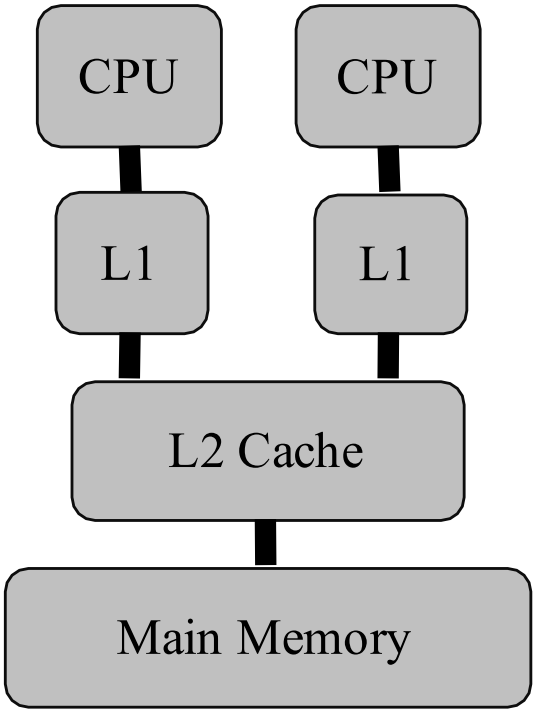
- Any read/write can reach any memory cell
- This is also for NUMA systems, but locality gets tricky
- Most models assume cache coherency

Communication through memory accesses

- Load/Store operations to arbitrary addresses
- Pass data from PE to the next

Synchronization constructs to coordinate accesses

- Need to ensure consistency
  - Data synchronization
- Need to ensure control flow
  - Control synchronization



Examples: **POSIX threads**, OpenMP, …

What is a Thread?

Independent stream of execution

- Own PC
- Own Stack

Hardware threads

- Implementation of an execution stream in hardware
  (think realization of a von Neumann machine in hardware)
- Separate Control Unit executing a sequence of instructions

Software threads

- Programming abstraction that represents a stream of execution
- Seen by programmer

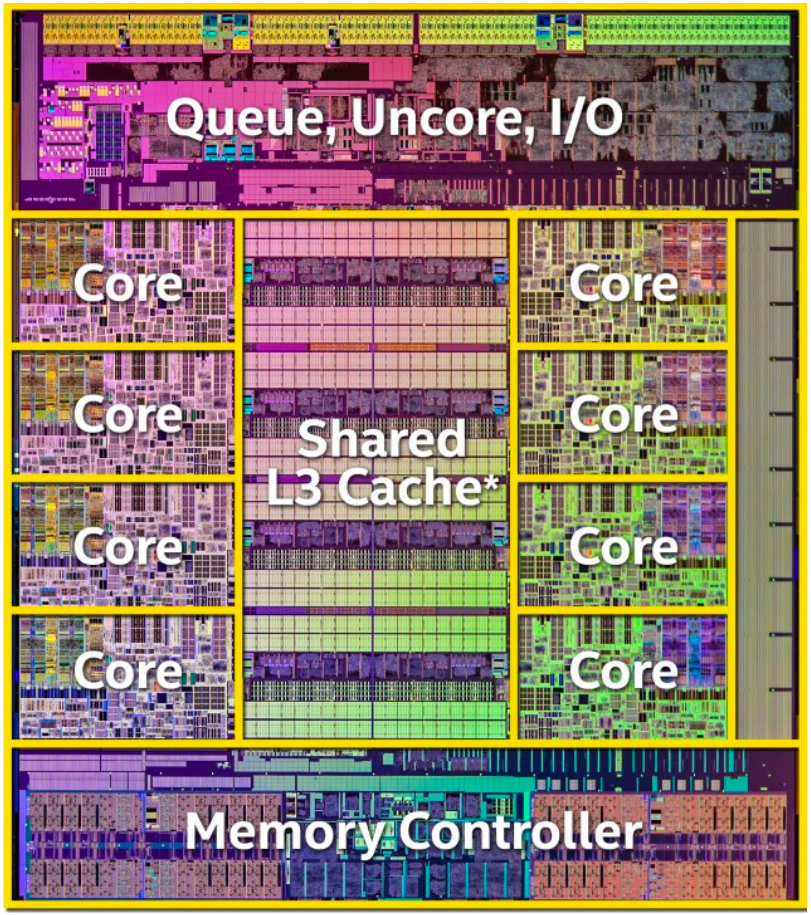
Hardware Threads in the Parallel Case

Traditional view
- One processor = one hardware thread (i.e., one control unit)

Boards with multiple sockets
- One hardware thread per socket

This all changed with multi-/many-core
- A processor now has multiple cores
- Each core has its own hardware thread

To add to the confusion
- OSes will report hardware-threads or cores as processors
- No distinction between sockets (by default)



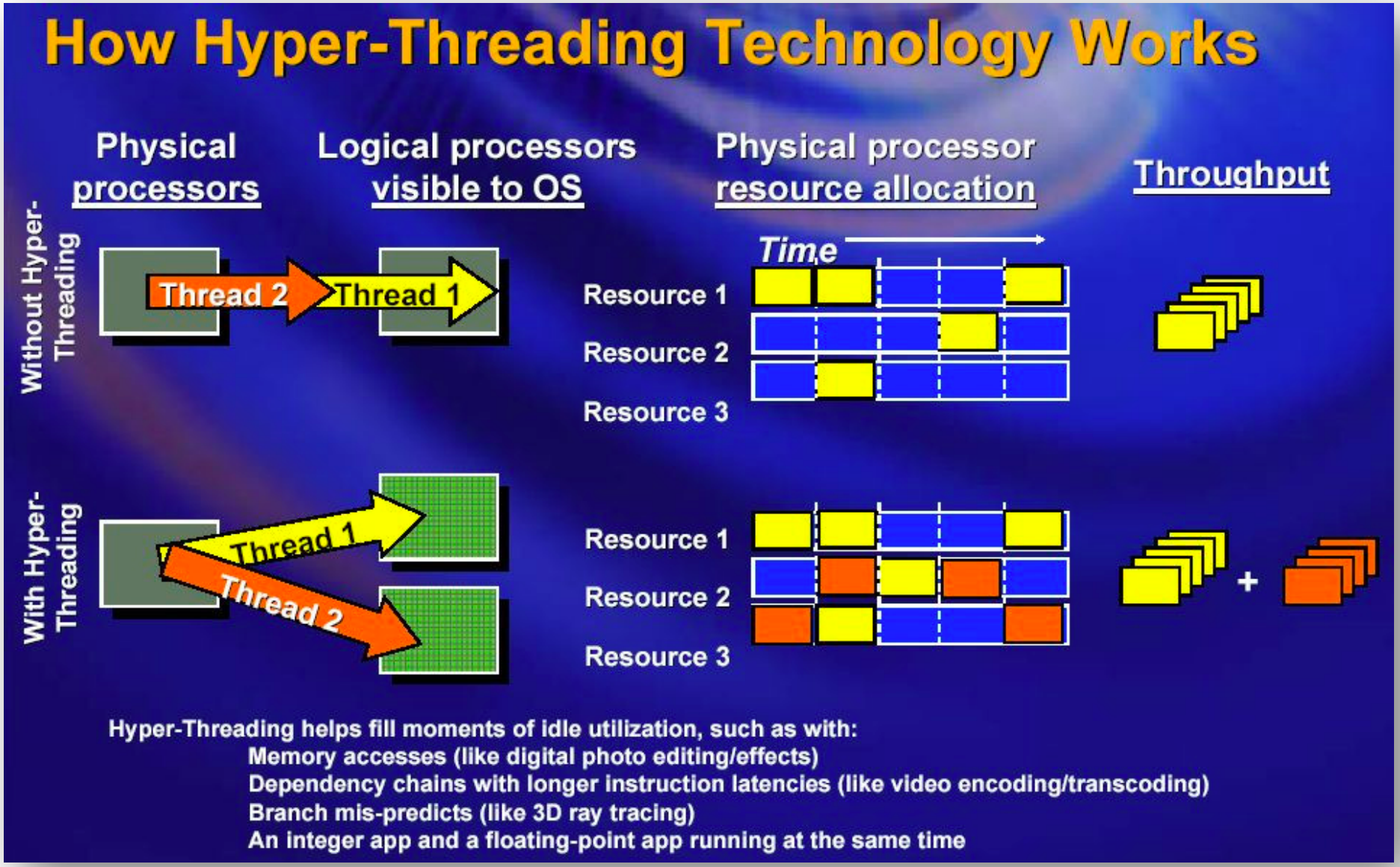
Die picture of an Intel Xeon Processor

cat /proc/cpuinfo (under Linux)

One entry for each “CPU”

i.e., hardware thread

processor : 23
vendor_id : GenuineIntel
cpu family : 6
model : 85
model name : Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
stepping : 4
microcode : 0x2000043
cpu MHz : 2100.000
cache size : 16896 KB
physical id : 1
siblings : 12
core id : 13
cpu cores : 12
apicid : 58
initial apicid : 58
fpu : yes
fpu_exception : yes
cpuid level : 22
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb invpcid_single intel_pt rsb_ctxsw spec_ctrl retpoline kaiser tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx avx512f rdseed adx smap clflushopt clwb avx512cd xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window hwp_pkg_req
bugs : cpu_meltdown spectre_v1 spectre_v2
bogomips : 4191.73
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:



Source: Intel

Using Hyperthreading / SMT

Hyperthreads will also show up as “CPUs”
- BIOS initializes them along with cores and sockets as boot
- Not distinguishable by default
- Changes require reboot

Useful for regular “concurrent” workloads
- E.g., multiple concurrent programs on a laptop
- Resource sharing beneficial

For parallel programming, this is problematic
- Multiple instances of same program with same resource requirements
- Heavy impact on speedup
- Need to be careful on what to schedule on a HT/SMT and what not
- Note: many HPC centers turn this off by default

Still could be useful for background tasks
- System daemons
- I/O operations

What is a Thread?

Independent stream of execution

- Own PC
- Own Stack

Hardware threads

- Implementation of an execution stream in hardware
  (think realization of a von Neumann machine in hardware)
- Separate Control Unit executing a sequence of instructions

Software threads

- Programming abstraction that represents a stream of execution
- Seen by programmer

To execute a program

- Programmer defines a software thread
- Software thread gets mapped to hardware thread for execution
  (by OS and/or runtime)

Software Threading Basics

Traditional view

- Operating systems maintain processes
- Processes get scheduled to available hardware threads (aka. processors)
- Each process has one execution stream

Processes maintain isolation for protection

- Separate address spaces and files
- Coupled with user IDs
- Communication only via IPC (Inter-Process Communication)

Threading was intended to make this easier

- Sharing of data without protection boundaries
- Cooperative concurrency to support asynchronous behavior
  - Example: I/O in the background, GUI threads
- OS still responsible for scheduling (at least for system-level threads)
  - Enables preemption and progress

The Linux Clone call

```
int clone(int (*fn)(void *), void *child_stack, int flags,
void *arg, ... /* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

Shared call for process and thread creation

Flags	CLONE_FILES	Parent/Child share file descriptor table
	CLONE_NEWIPC	Establish new IPC name space
	CLONE_NEWNET	Establish new network name space
	CLONE_NEWUSER	Create new child under new UID
	CLONE_THREAD	Parent/Child in same thread group
	CLONE_VM	Parent/Child in same memory space
	…	

Threads and processes are scheduled by the same scheduler

User-Level Threads

Alternative: implement threads as part of a user-level library

- Maintain own PC and stack
- Switch between threads as needed
- No kernel support or modifications necessary

Advantages

- Easier to implement and support
- Lighter weight

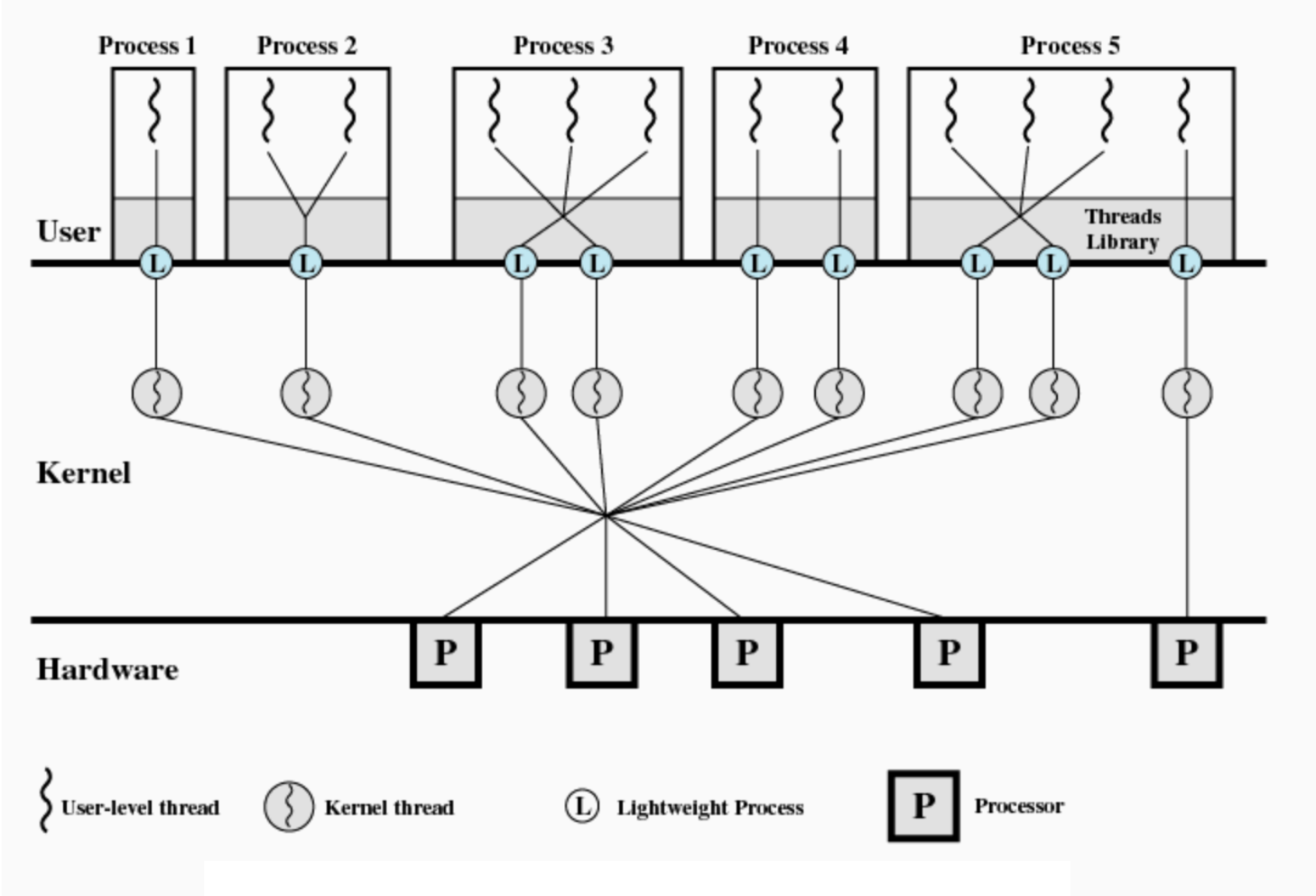
Disadvantages

- No scheduling guarantees
- Preemption and progress is very hard to guarantee, if not impossible
- Often requires explicit `yield()` calls required

Use cases

- Light-weight, fine-grained task based parallel programming
- Closely coordinated activities with well-define switchpoints

Hybrid Models (e.g., Solaris)



From: Brian Vinter, University of Copenhagen, http://www.diku.dk/~vinter/xmp/lecture2.pdf

The Windows View of the World

A process is a data structure with

- Memory space
- File descriptors
- Network interfaces
- BUT: a process cannot execute on its own

A process contains one more threads

- Threads have a stack and a program counters
- Threads execute

Additional: Fibers

- User-level threads
- User scheduled and cannot be preempted
- Hierarchical compared to main threads

Why Talk about Threads in Parallel Programming?

The traditional use case is concurrency

- BUT: threads can be (and have been) used for parallel programming

Threads expose hardware threads to programmer

- Native interface to access parallelism in the architecture

Motivation 1

- Exploit hardware threads as directly as possible
- Direct mapping to resources
- Understanding bottlenecks and overheads where they occur

Motivation 2

- Runtime systems for parallel programming environments build on them
  - For example, OpenMP implementations run on top of native thread packages
- Properties of threads and how they are used have large impact on performance

How to Program with Threads?

Some of the more common APIs

- POSIX Threads
- Win32 Threads
- Solaris Threads
- Java Threads

Many custom/research packages

- Often mapped to native APIs
- Often user-level threads

Motivation for custom APIs

- Lower overhead
- Customized for particular tasks
- Custom hardware with special properties

POSIX Threading

POSIX: Portable Operating System Interface

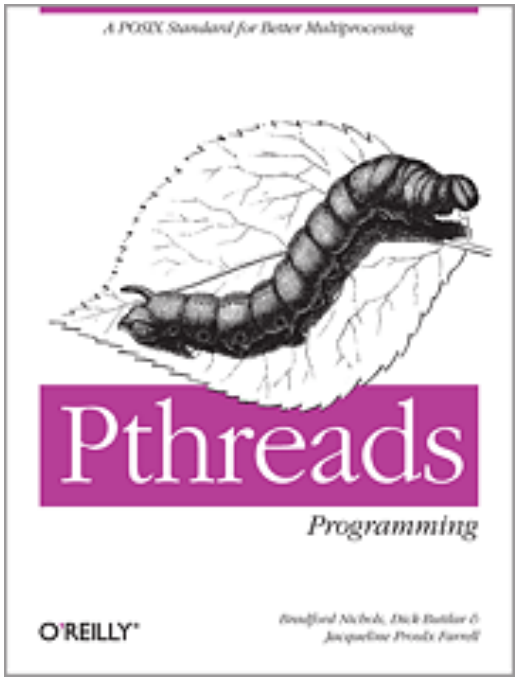
- Family of IEEE Standards
- Ensure compatibility between OSs
- Defines and API, Shells, Utilities

POSIX threads: IEEE Std 1003.1c-1995

- Contains about 100 procedures (Prefix pthread_)
- Standardized access to threads
  - Creation, destruction
  - Coordination and synchronization
  - Thread management
- Available on most systems
- Typically used for system level threads

Programming with pthreads

- `#include <pthread.h>`
- On some systems compile with „`-lpthreads`“



By Dick Buttlar, Jacqueline Farrell, Bradford Nichols
Publisher: O'Reilly Media
Release Date: January 2013

Also:
https://computing.llnl.gov/tutorials/pthreads/

Fork/Join Model

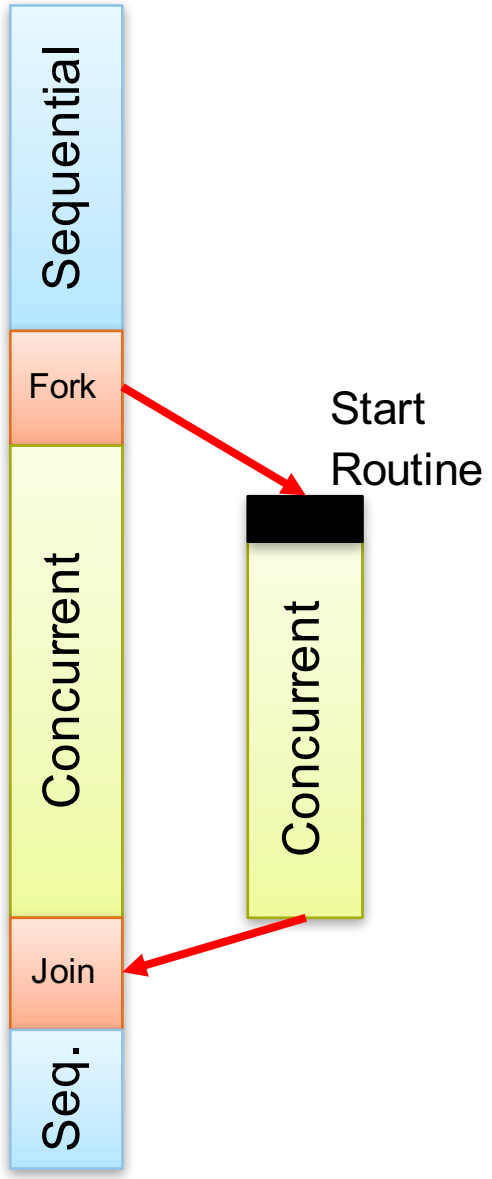
A master thread executes
- Sequential execution on a HW thread

Thread want to spawn a second thread
- E.g., to start handing off work

“Forks” a new thread
- Starts a new thread
- New threads start to a specified routine

Both threads operate concurrently
- One one ore more hardware threads
- Location (at first) transparent

At the end of the parallel regions:
- Master waits for other thread to complete
- Continues sequential execution



POSIX Thread Create

```
int pthread_create(pthread_t *thread, const pthread_attr_t
*attr, void *(*start_routine) (void *), void *arg);
```

Create a new Pthread

Returns 0 if successful

Arguments:

- thread = returns thread ID
- attr = attributes for the new thread
- start_routine = routine that executes the new thread
- arg = argument passed to the new thread

POSIX Thread Join

```
int pthread_join(pthread_t thread, void **retval);
```

Waits for another thread to terminate

Returns 0 if successful

Arguments:

thread = ID of thread to wait for

retval = return value from the terminated thread

POSIX Threads - Details

Attributes

- Set of properties defining thread behavior
- Examples: bound/unbound, scheduling policy, …

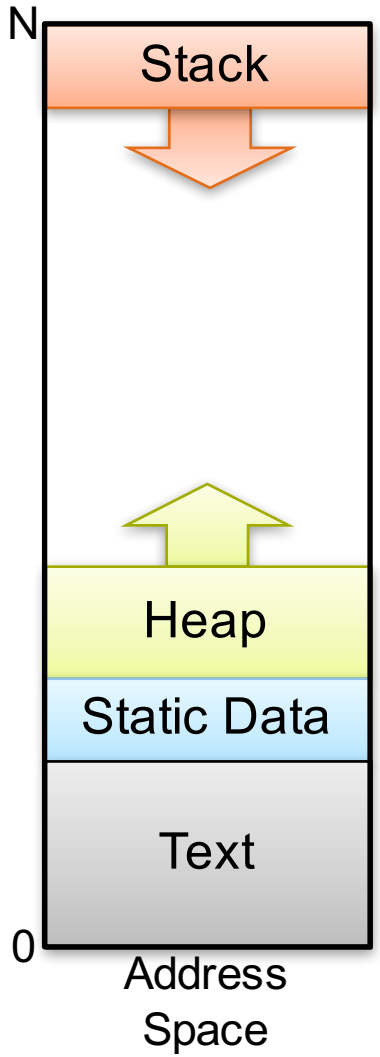
Thread management and special routines:

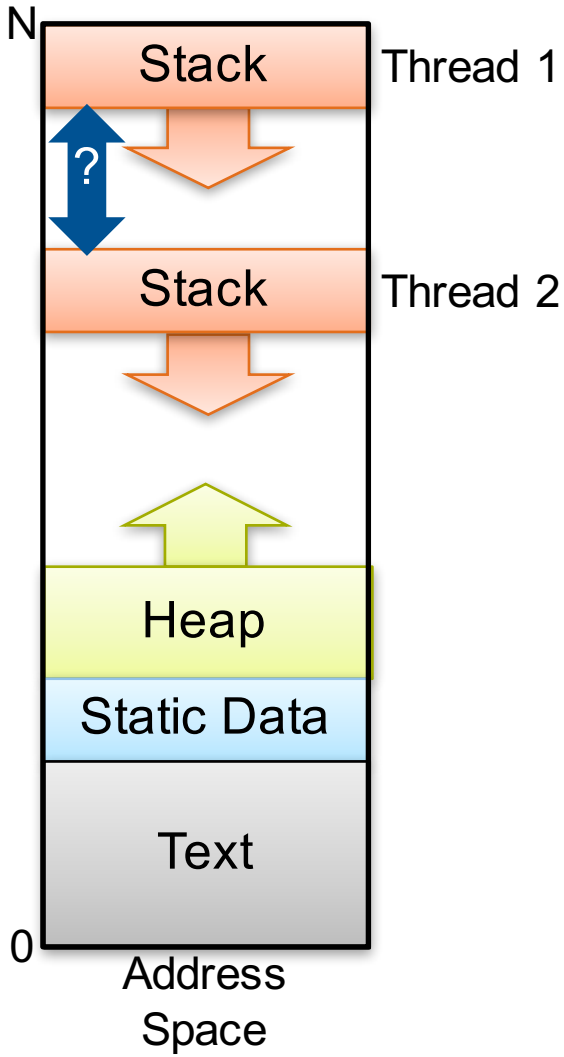
- Get own thread ID: `pthread_self()`
- Compare two thread IDs: `pthread_equal(t1,t2)`
- Run a particular function once in a process: `pthread_once(ctrl, fct)`

Stack management

- Routines to set and get the stack size
- Routines to set and get the stack address

Stack Structure





Fork/Join Example (part 1)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define NUM_THREADS 5

void* perform_work(void* argument)
{
        int passed_in_value;
        passed_in_value = *((int*) argument);
        printf("Hello World! It's me, thread with argument
                %d!\n", passed_in_value);
        return NULL;
}
```

Source: Wikipedia

Fork/Join Example (part 2)

```
int main(int  argc, char** argv)
{
        pthread_t threads[NUM_THREADS];
        int thread_args[NUM_THREADS];
        int result_code; unsigned index; /

        / create all threads one by one
        for (index = 0; index < NUM_THREADS;  ++index)
        {
                thread_args[  index  ] = index;
                printf("In main: creating thread %d\n", index);
                result_code = pthread_create(&threads[index],
                        NULL,
                        perform_work, &thread_args[index]);
                assert(!result_code);
        }
```

Source: Wikipedia

Fork/Join Example (part 3)

```
        // wait for each thread to complete
        for (index = 0; index < NUM_THREADS; ++index)
        {
                // block until thread 'index' completes
                result_code = pthread_join(threads[index], NULL);
                                assert(!result_code);
                printf("In main: thread %d has completed\n",
                        index);
        }
        printf("In main: All threads completed successfully\n");

        exit(EXIT_SUCCESS);
}
```

Source: Wikipedia

Fork/Join Example (Output)

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread with argument 0!
In main: creating thread 2
Hello World! It's me, thread with argument 1!
In main: creating thread 3
Hello World! It's me, thread with argument 2!
In main: creating thread 4
Hello World! It's me, thread with argument 3!
Hello World! It's me, thread with argument 4!
In main: thread 0 has completed
In main: thread 1 has completed
In main: thread 2 has completed
In main: thread 3 has completed
In main: thread 4 has completed
In main: All threads completed successfully
```

Synchronization Between Threads

Unless we deal with pure concurrency

- Independent tasks
- No dependencies

we need to coordinate between threads

Examples

- Enforce common completion of tasks
- Enforce happens before relationships
- Guard updates to common data structures

In POSIX threads, there are two main concepts

- Locks / Mutual Exclusion
- Condition variables

Other constructs can be built on top of them

Concept 1: Mutual Exclusion

Problem: concurrent access to shared resources

- Shared variables, memory locations
- Access to I/O
- Two or more threads concurrent updates can lead to inconsistencies

Classic example: depositing money into a bank account

```
int account = 100;
void deposit(int money)
{
        account = account + money;
}
```

Thread Alice: `deposit(200);`
Thread Bob: `deposit(100);`

Possible final values for `account`:

400
- both succeed

300
- both read
- Bob writes first

200
- both read
- Alice writes first

POSIX Thread Locks

Initialization of a lock:

- Global variable of type `pthread_mutex_t`
- Initialize to `PTHREAD_MUTEX_INITIALIZER`
- Can also be done dynamically: `pthread_mutex_init/destroy()`

Lock a mutex

- `pthread_mutex_lock(&mutex);`
- Blocks until mutex is granted

Unlock a mutex

- `pthread_mutex_unlock(&mutex);`
- Returns immediately

Lock a mutex, if available

- `pthread_mutex_trylock(&mutex);`
- Returns immediately

Also locks can have attributes

POSIX Lock Example

```
int account = 100;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void deposit(int money)
{
        pthread_mutex_lock(&mutex);
        account = account + money;
        pthread_mutex_unlock(&mutex); return 0;
}
```

Note:

- mutex is a standalone variable
- Not explicitly associated with the memory it protects
- User/programmer responsibility

Lock Implementations

Criteria for implementations

- Correctness: Guarantees mutual exclusion
- Every process eventually gets the mutex
- Fairness

Spin-Locks

- If lock is taken, actively wait by “spinning” on a flag
- Advantage: fast response time
- Disadvantage: blocks the hardware threads, uses resources, costs energy
- Bad for concurrent executions

Yielding Locks

- If lock is taken, yield hardware thread
- Advantage: low resource usage
- Disadvantage: slow response time
- Bad for HPC

Implementation of a Lock with Hardware Support

Most systems support some form of atomic memory operations
- Multiple operations executed atomically
- Cannot not be interrupted by thread switches

Most common examples:
- Test and set
  - Sets a flag to 1 and returns the old value
- Compare and swap
  - Set new value, if old value has a given value

Lock implementation with “test and set”

```
volatile int *mutex;
void lock()
{
     while (test_and_set(&mutex)==1));
}
```

Implementation Without Hardware Support

Peterson's Algorithm

G. L. Peterson: "Myths About the Mutual Exclusion Problem", *Information Processing Letters* 12(3) 1981, 115–116

Example for two threads (assumes tid=0 or tid=1)

```
volatile int flag intent_mutex[2] = {0,0};
volatile int thread;

intent_mutex[tid]=1;  /* thread wants to enter */
thread = 1-tid;       /* other thread is next */

while (intent_mutex[1-tid] && thread==1);
       /* if lock is taken and not my turn, wait */

/* critical section */

intent_mutex[tid]=0;
```

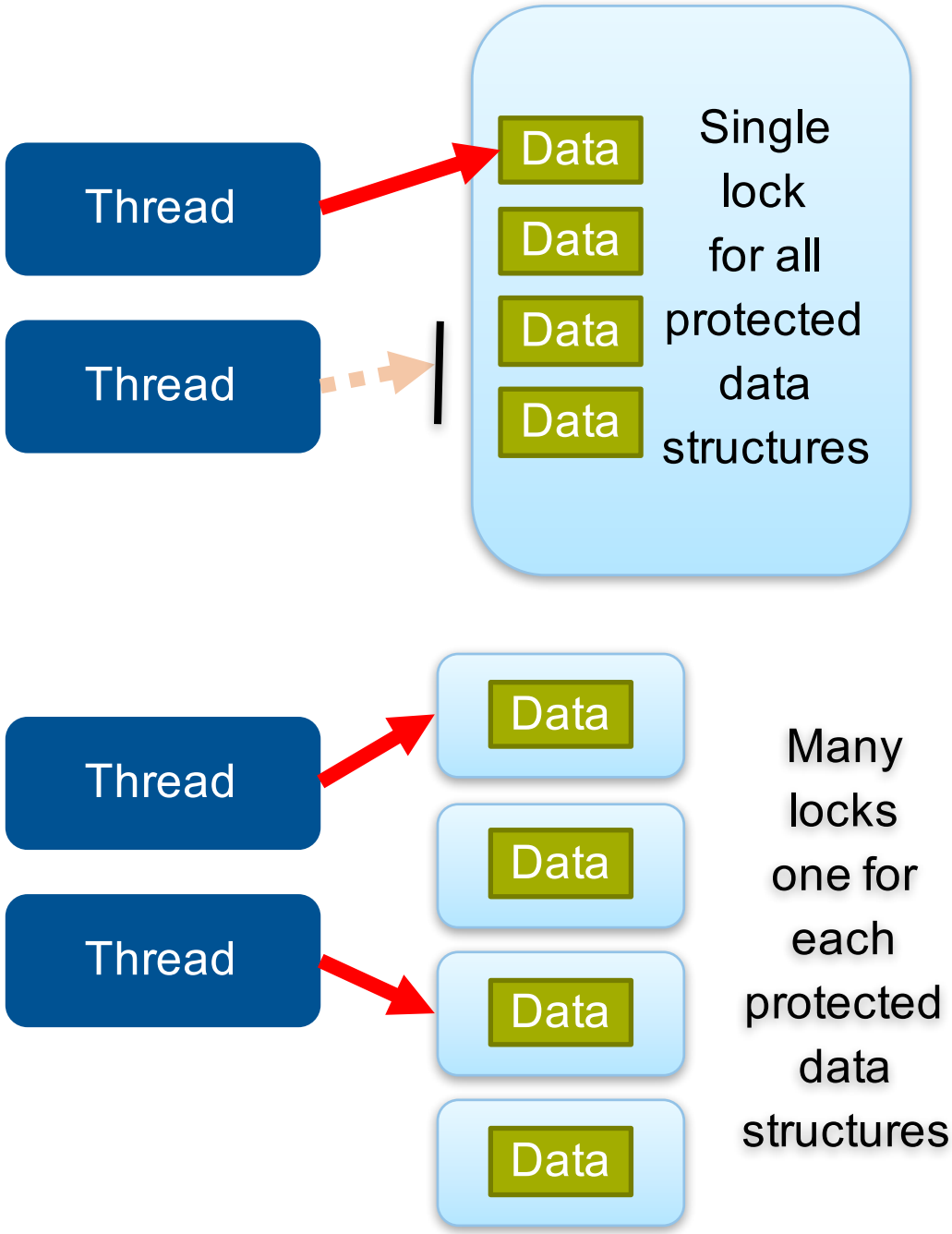
Still assumes some atomicity and memory consistency

Lock Granularity

Coarse grained locking
- One single lock for all data
- Limits concurrency
- Eases implementations
- Older OSes use this

Fine grained locking
- One lock for each data element
- Maximizes concurrency (multiple threads can have locks)
- Requires many locking calls
- May require multiple locks at the same time

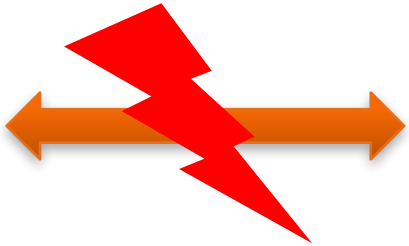
Hybrid versions often useful



Danger: Deadlocks

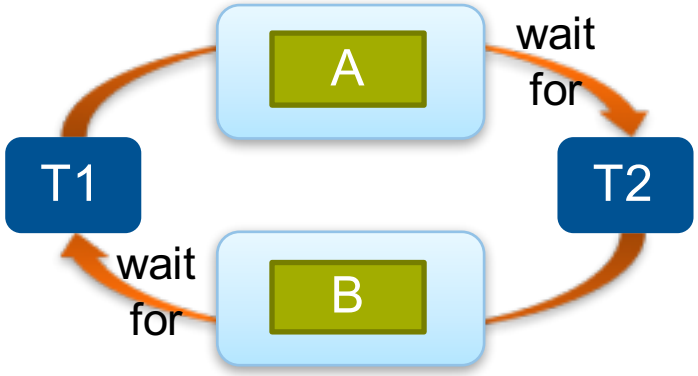
```
Thread 1                    Thread 2

LOCK FOR A                  LOCK FOR B
LOCK FOR B                  LOCK FOR A
READ A and B                READ A and B
UNLOCK A                    UNLOCK A
UNLOCK B                    UNLOCK B
```



Both threads request lock A and B

- Both end up waiting for the second lock
- Cyclic dependency!



Classic Example: Dining Philosophers

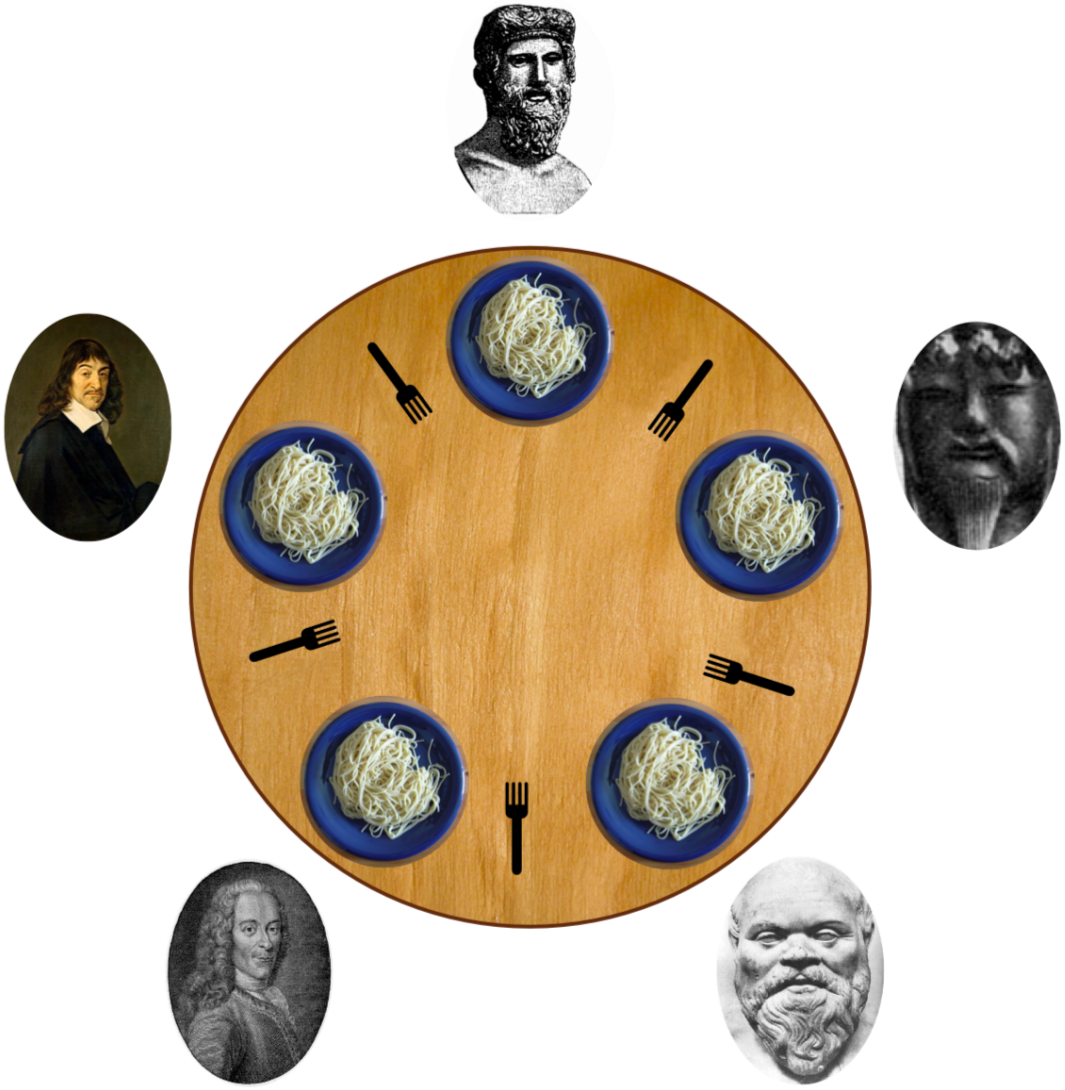
5 philosophers sit at a table and

- Think
- Think until left fork is free
- Pick up left fork
- Think until right fork is free
- Pick up right fork
- Eat for some time
- Put down both forks

Translated to locks:

Philosopher P

- `Compute`
- `Lock(lock[P])`
- `Lock(lock[(P+1) % 5]`
- `Compute`
- `Unlock(lock[P])`
- `Unlock(lock[(P+1) % 5]`



Source: Wikipedia

Strategies for Deadlock Avoidance

Easy option: only hold one lock at a time

Central arbiter to ask for permission

Order among all locks and only allocate in that order

```
Thread 1                    Thread 2

LOCK FOR A                  LOCK FOR A
LOCK FOR B                  LOCK FOR B
READ A and B                READ A and B
UNLOCK A                    UNLOCK A
UNLOCK B                    UNLOCK B
```

Classic Example: Dining Philosophers

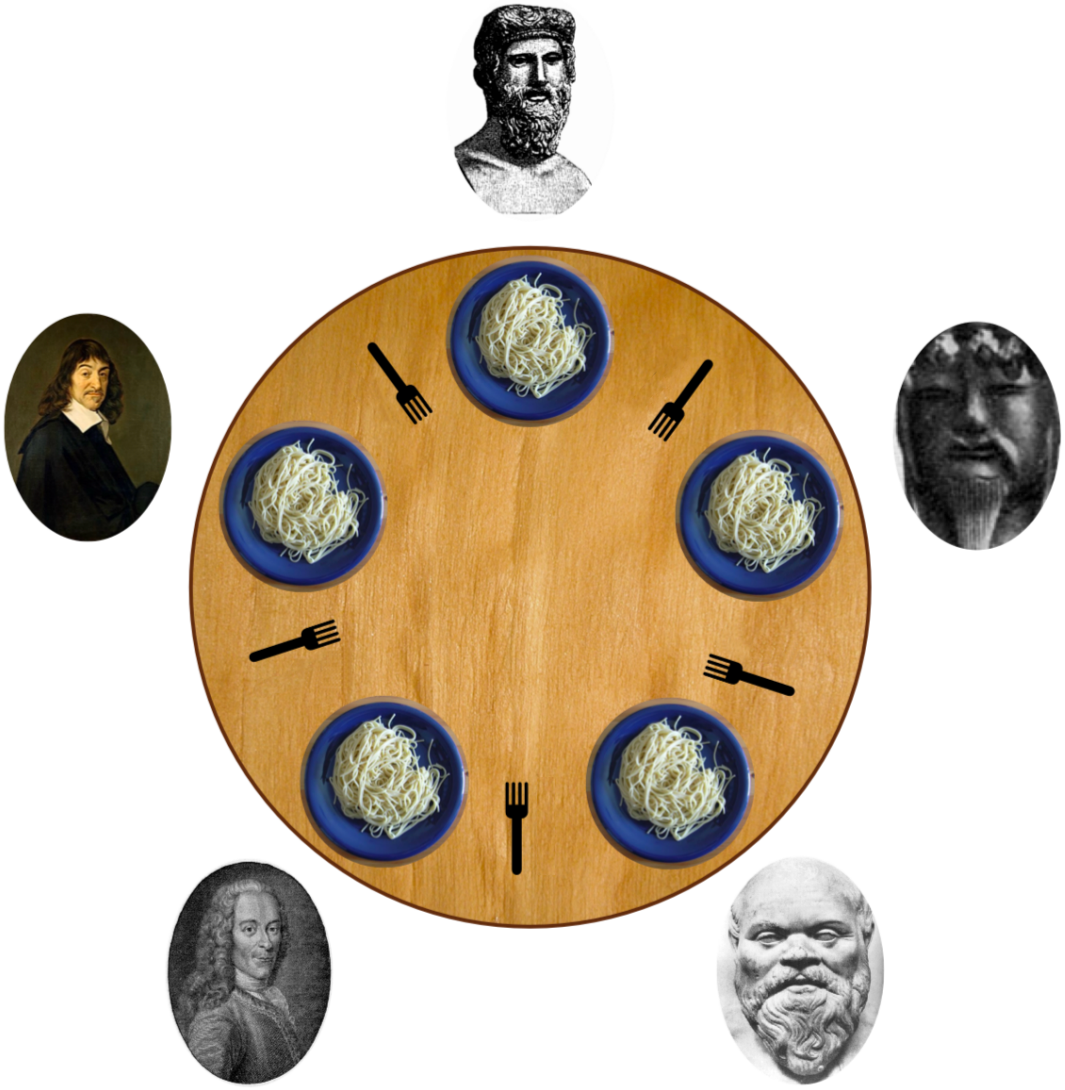
All solutions have drawbacks

One lock
- We still need two forks
- One lock for whole table?

Arbiter
- Need central instance
- Might be complex to implement

Lock order
- Fairness
  (one philosopher may starve)

Custom schemes possible
- Requesting forks from neighbors
- Preference to starving processes



Source: Wikipedia

Concept 2: Condition Variables

Waiting based on the value of a shared variable

- Works in conjunction with a mutex variable to protect that data

One thread needs to wait for a condition to be true

- Blocks until this happens

Second thread checks condition

- Signals other thread when condition is met

Condition Variables in the POSIX Thread API

Initialization of a condition variable:

- Global variable of type `pthread_cond_t`
- Initialize to `PTHREAD_COND_INITIALIZER`
- Can also be done dynamically: `pthread_cond_init/destroy()`

Wait on a condition to be set in another thread

- `pthread_cond_wait(&condition,&mutex);`
- Unlocks mutex, blocks until condition is signaled, acquires lock
- Mutex must be locked before calling

Signal a condition to a waiting thread

- `pthread_cond_signal(&mutex);`
- `pthread_cond_broadcast(&mutex);`
- Send condition to other thread
- Must be followed by an unlock

Also condition variables can have attributes

Example: Condition Variables

Global information

```
int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
```

Waiting thread

```
Pthread_mutex_lock(&mutex);
While (count<=100)
{
   pthread_cond_wait(&cond,&mutex);
}
pthread_mutex_unlock(&mutex);
```

Guard against sporadic wake-ups!

Signaling thread

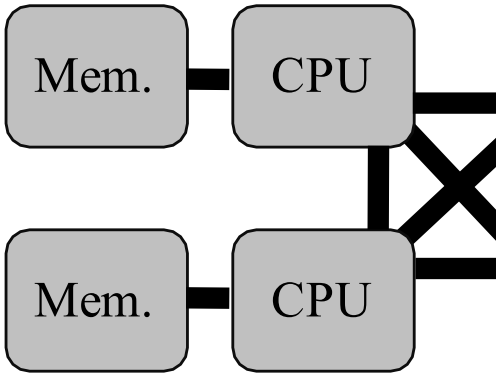
```
for (i=0; i<1000; i++)
{
   pthread_mutex_lock(&count);
   count++;
   if (count==100)
   {
      pthread_cond_signal(&cond);
   }
   pthread_mutex_unlock(&mutex);
}
```

Performance Aspects for Threading

Overheads
- Thread creation and destruction can be expensive operations
- Ensure large parallel regions or “park” threads

Lock contention
- Locks are low-overhead when few threads try access them
- Overhead grows with more threads accessing them more often

Thread pinning
- For system-level threads the OS does scheduling
  - Mapping of SW to HW thread
  - Determines the location of a thread’s execution
- Large impact on performance
  - Determines what is needed to share information
  - NUMA properties
- Pinning, fixing a SW thread to a (group of) HW thread(s)
  - Thread attributes, libraries like libNUMA (see `man numa`)



Good use of caches, avoidance of false sharing (see exercises)

Summary

A thread is a stream of execution

Hardware threads
- Implementation of the “von Neumann” control unit (CU)
- Complicated by multi-/many core and Hyperthreading/SMT

Software threads are abstracting execution streams for the programmer
- Distinguish user- and system-level threads

Widely available and standardized API: POSIX threads
- Routines for thread creation/destruction
- Several synchronization constructs, incl. mutexes and condition variables

Performance challenges
- Reduce thread management overhead
- Lock contention
- Bottlenecks in the memory system
- Missing thread pinning can lead to bad HW/SW mappings