Lecture IN-2147
Parallel Programming

Martin Schulz

Technical University of Munich

Department of Computer Engineering

Lecture 2: Threading

- Seg. 6: Introduction into Parallel Architectures
- Seg. 7: Concept of Threading
- Seg. 8: The Pthreads API
- Seg. 9: Thread Synchronisation
- Seg. 10: Modern Thread APIs

Recap: Lecture 1- Introduction

Parallel processing
- Multiple tasks working together to finish a (a) larger problem (b) faster
- Goal has to be efficiency
- Multi-/Many-core developments catapult this out of a niche into every system

Programming in parallel
- Decomposition of work and data using choice of best fitting pattern
- Hybrid models are likely the best choice for individual applications
- Mapping to architectures critical

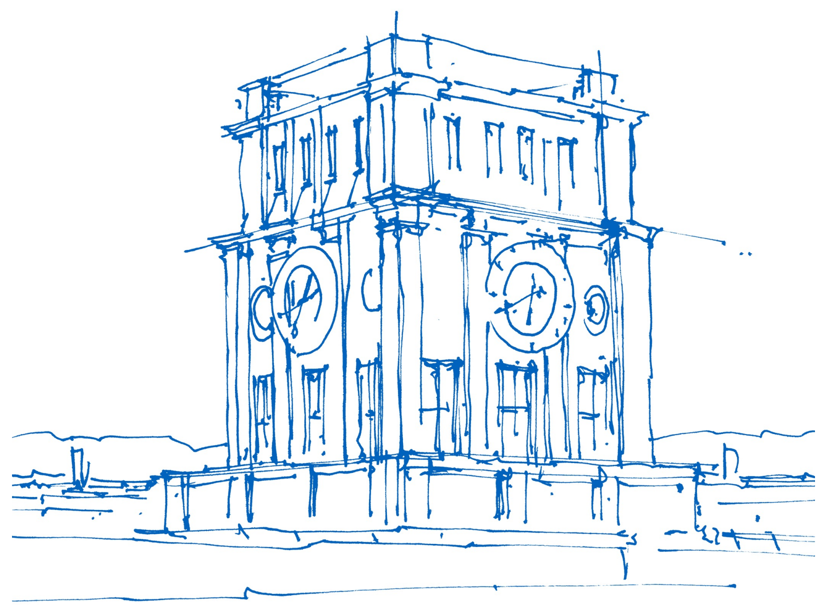
Metrics for Determining Success of Parallelization
- Speed-Up and Efficiency
- Amdahl’s law to determine theoretical upper bounds
- Also here: exceptions in from of super-linear speed-ups caused by architectures

Think parallel!

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 6
Introduction into Parallel Architectures

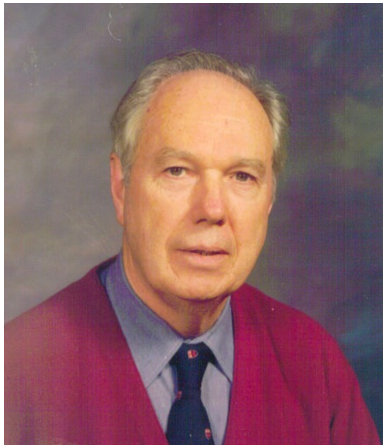


Uhrenturm der TUM

Flynn’s Classification

M. Flynn, Very High-Speed Computing Systems, Proceedings of the IEEE, 54, 1966

	Single Data	Multiple Data
Single Instruction	**SISD** Sequential Processing	**SIMD** Pipelines, Vectors, GPUs
Multiple Instruction	**MISD** ??? / Systolic Arrays	**MIMD** MPP Systems Clusters



Michael Flynn
Born 1938

Relevant items covering parallel systems

- SIMD (Single Instruction Multiple Data):
  Synchronized execution of the same instruction on a set of data
- MIMD (Multiple Instruction Multiple Data):
  Asynchronous execution of different instructions

Classification

Parallel Systems

SIMD

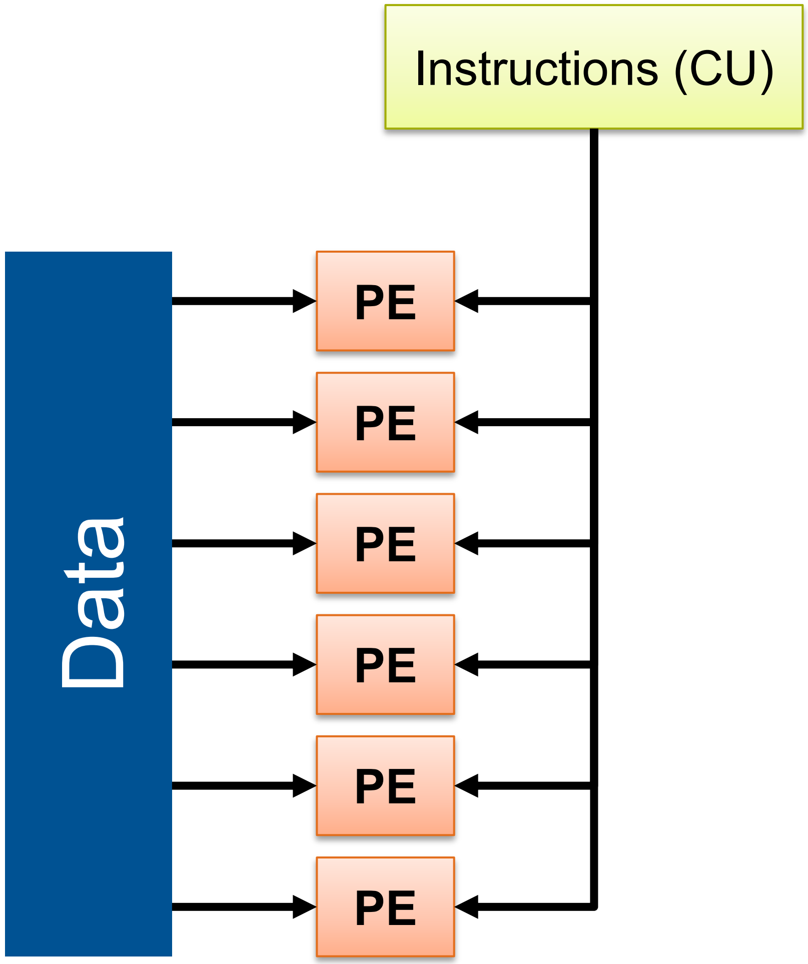
MIMD

SIMD Systems

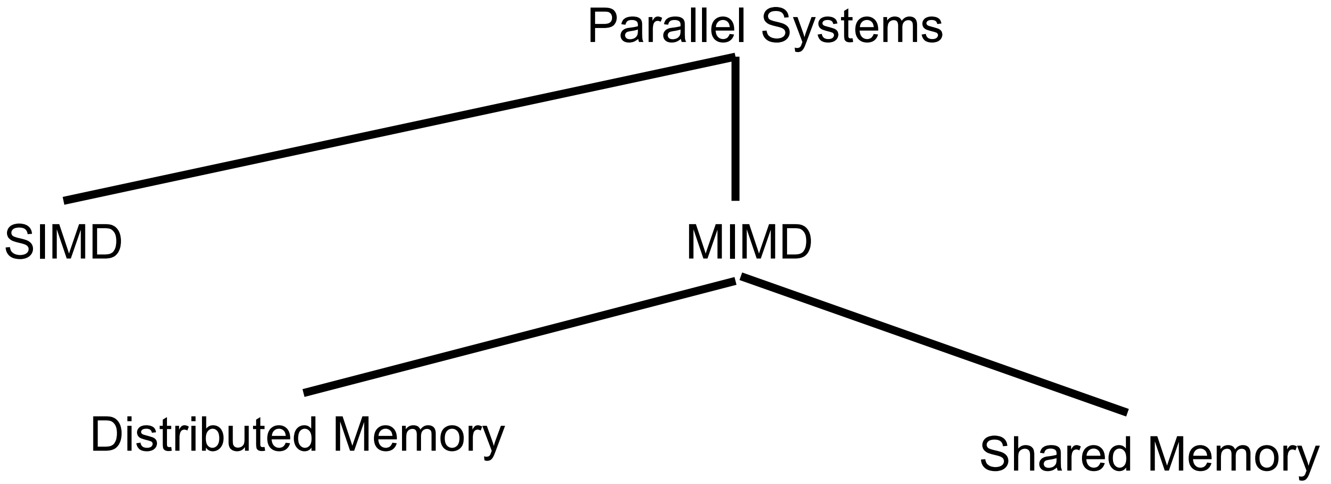
One instructions operates
on a many data streams

Vector processing
SIMD instructions

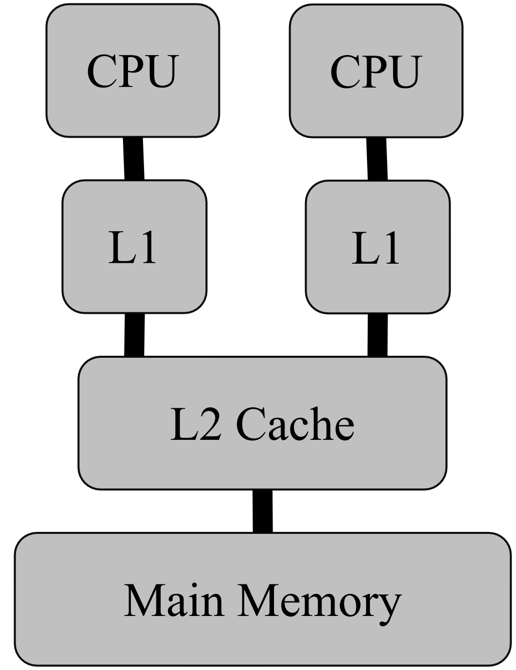
GPGPU processing
fits the same model



Classification



Shared Memory

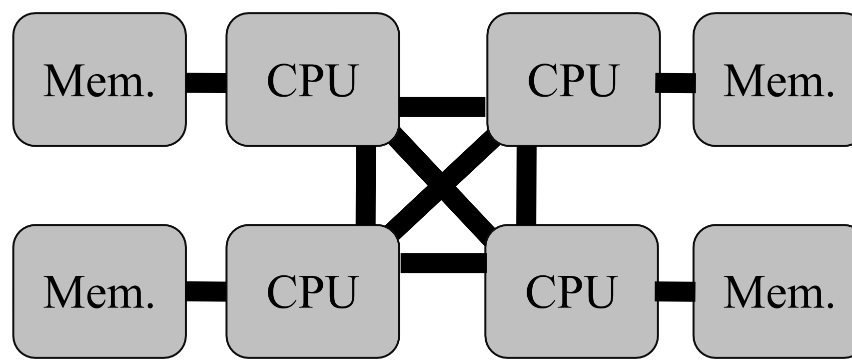


Uniform Memory Access – UMA :
(symmetric multiprocessors - SMP):

- 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):

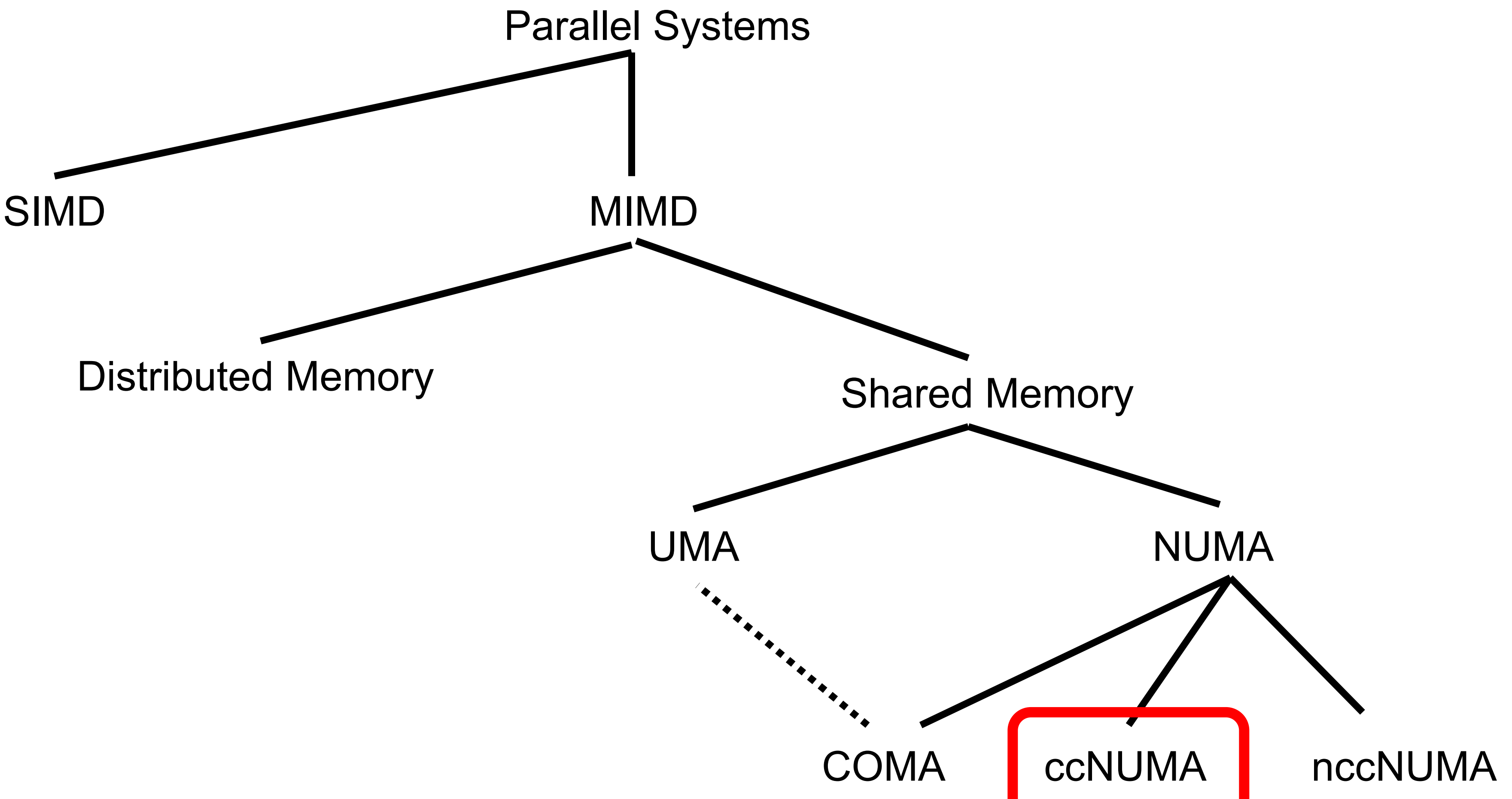
- Memory is distributed among the nodes
- Local accesses much faster than remote accesses.



More exotic

- COMA (Cacho Only)
- NCC-NUMA (non cache coherent)

Classification



Diversity in Parallel Programming Models

Driven by architecture developments (at least traditional and at the low level)

Most attached to an existing sequential programming language
- Most common: C, C++, Fortran
- Scripting languages are becoming more relevant
- APIs or language extensions

SIMD or Vector Programming
- Often in the form of pragmas (many vectorizing compilers)
- CUDA, OpenCL, ... as separate languages (but again built on base language)

Shared Memory Programming Models
- MIMD models that match shared memory architectures

Message Passing Programming Models
- MIMD models that match distributed memory architectures

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

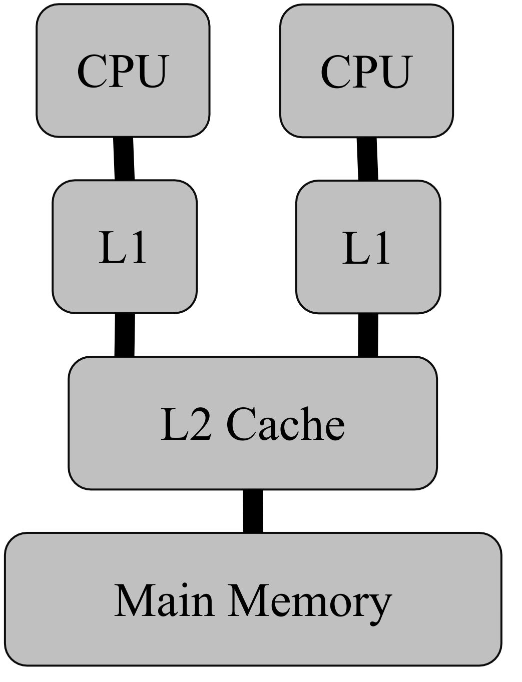
Communication through memory accesses

- 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

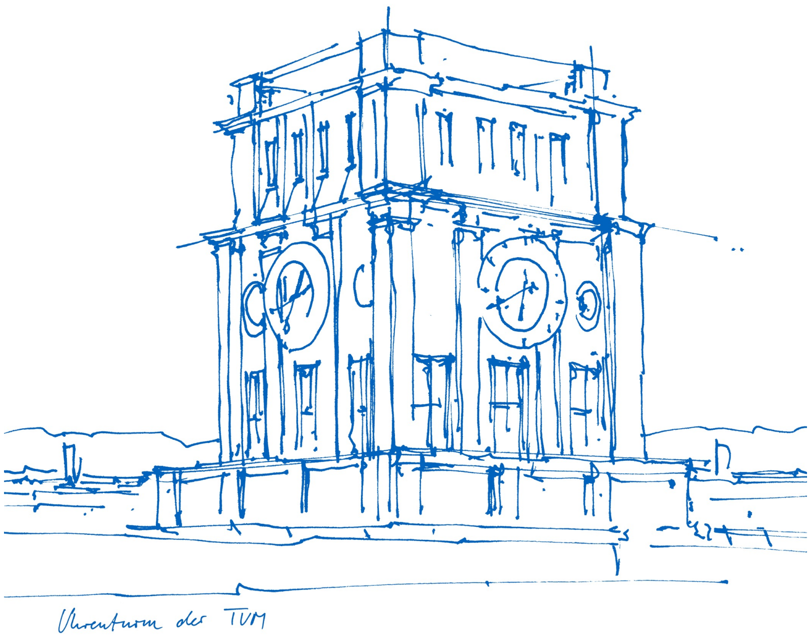
Examples: POSIX threads, OpenMP, …



Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 7
The Concept of Threading



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

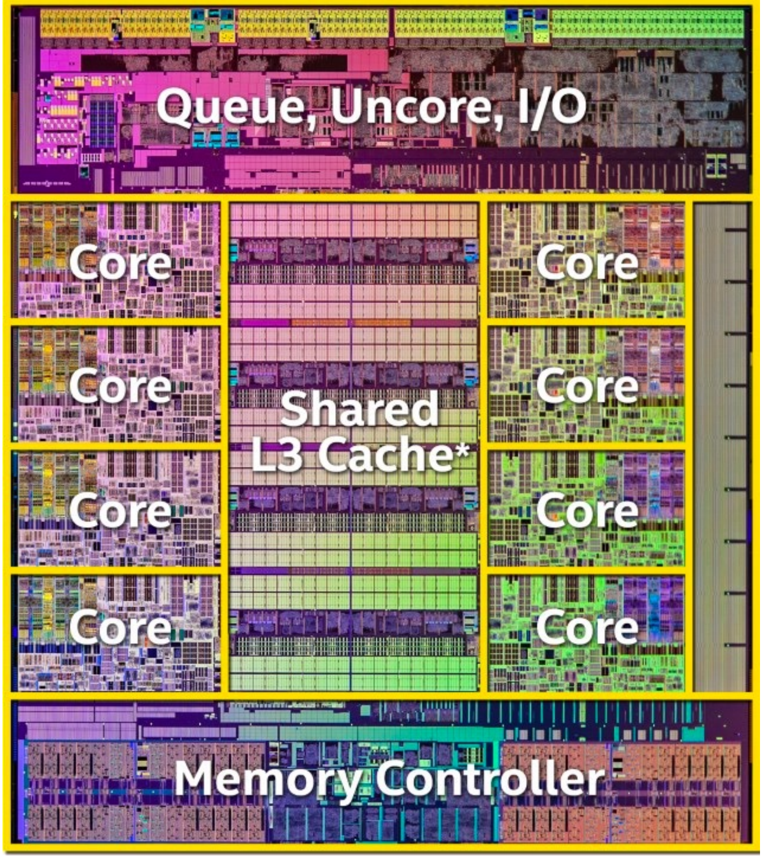
Hardware Threads in the Parallel Case

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 (or even multiple ones → SMT)

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

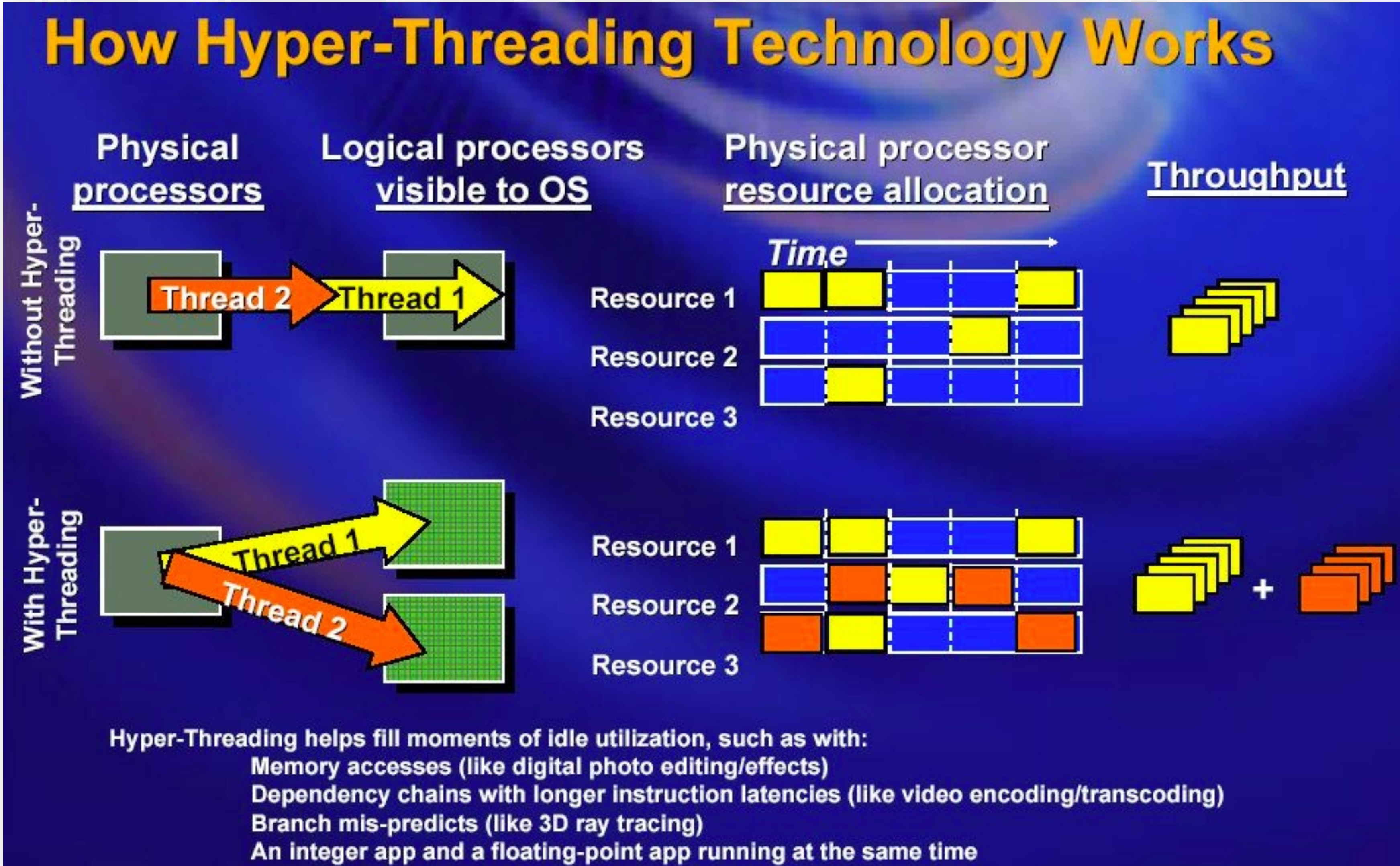


Die picture of an Intel Xeon Processor

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
- Some architectures may require it, though

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

Most Common Model: Fork/Join Model

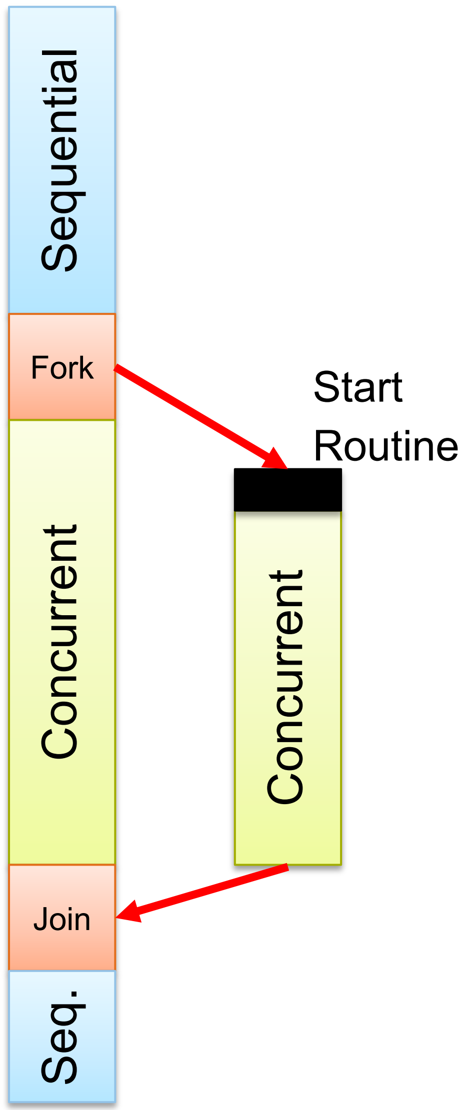
A main thread executes in a process
- Sequential execution on a HW thread

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

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

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

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



Different APIs and Threading Implementations

How to implement the fork operation?

- How can the fork be specified by the programmer?
- Is this an OS operation?
  - Mapping to OS abstractions
- Is this a runtime (user-level) runtime mechanism?
  - Understanding thread properties

Similar for join operation

How to pass parameters into a new thread?

How to retrieve results?

- Arguments for new computation
- Results of computation

Runtime interaction

- Synchronization between threads while they are running

The Linux Clone call (System Threads)

```
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

Arguments passed via “arg”, return via function return

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

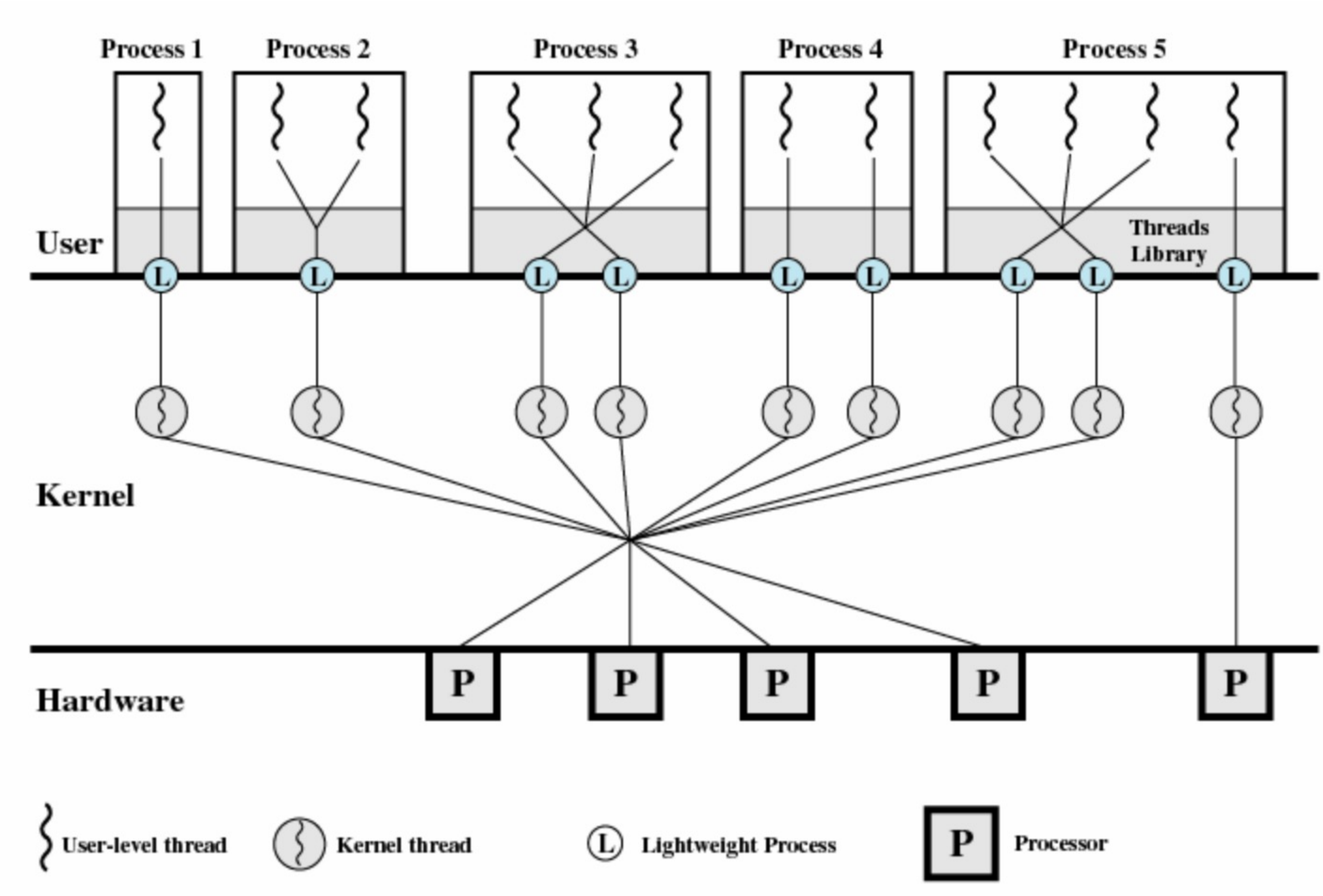
Disadvantages

- 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

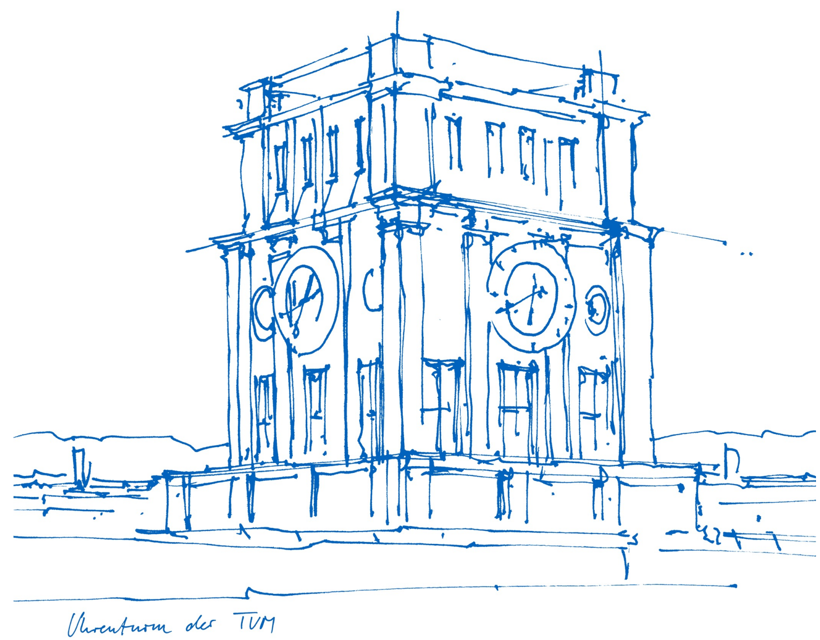
- Threads have a stack and a program counters
- Threads execute

Additional: Fibers

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

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering



Segment 8
The POSIX Thread API

POSIX Threading

POSIX: Portable Operating System Interface

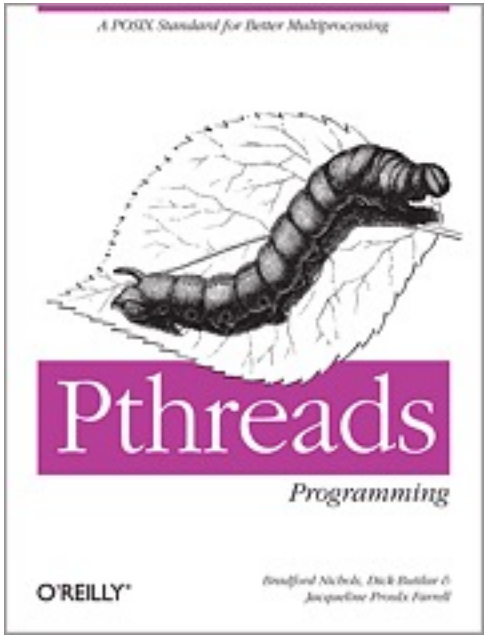
- Family of IEEE Standards
- Ensure compatibility between OSs
- Defines and API, Shells, Utilities

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/

POSIX Thread Create

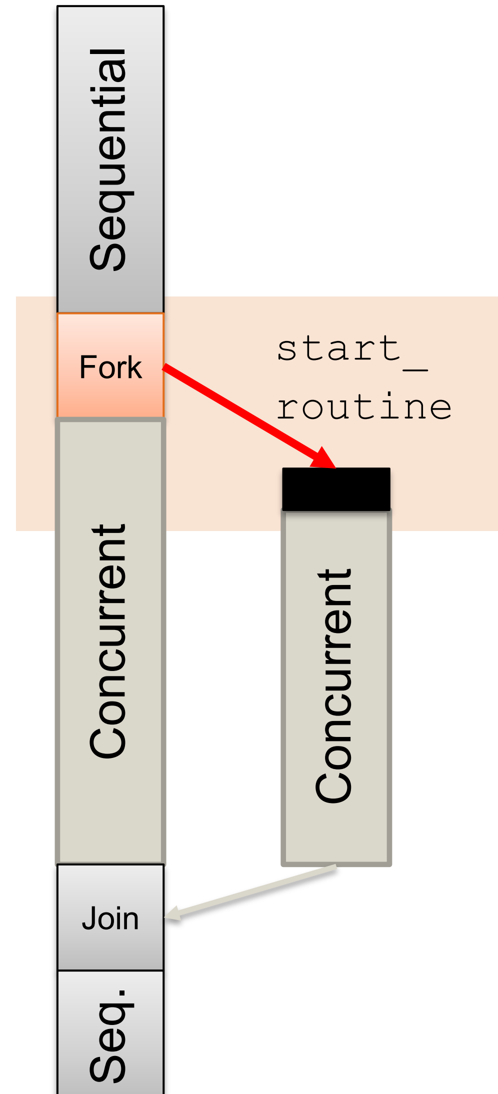
```
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

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);
```

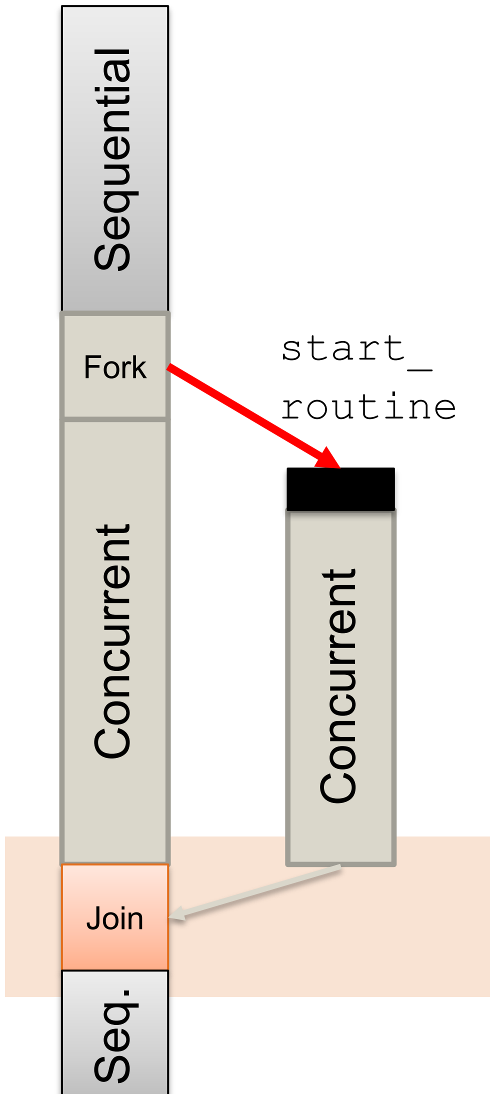
Waits for another thread to terminate

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, …

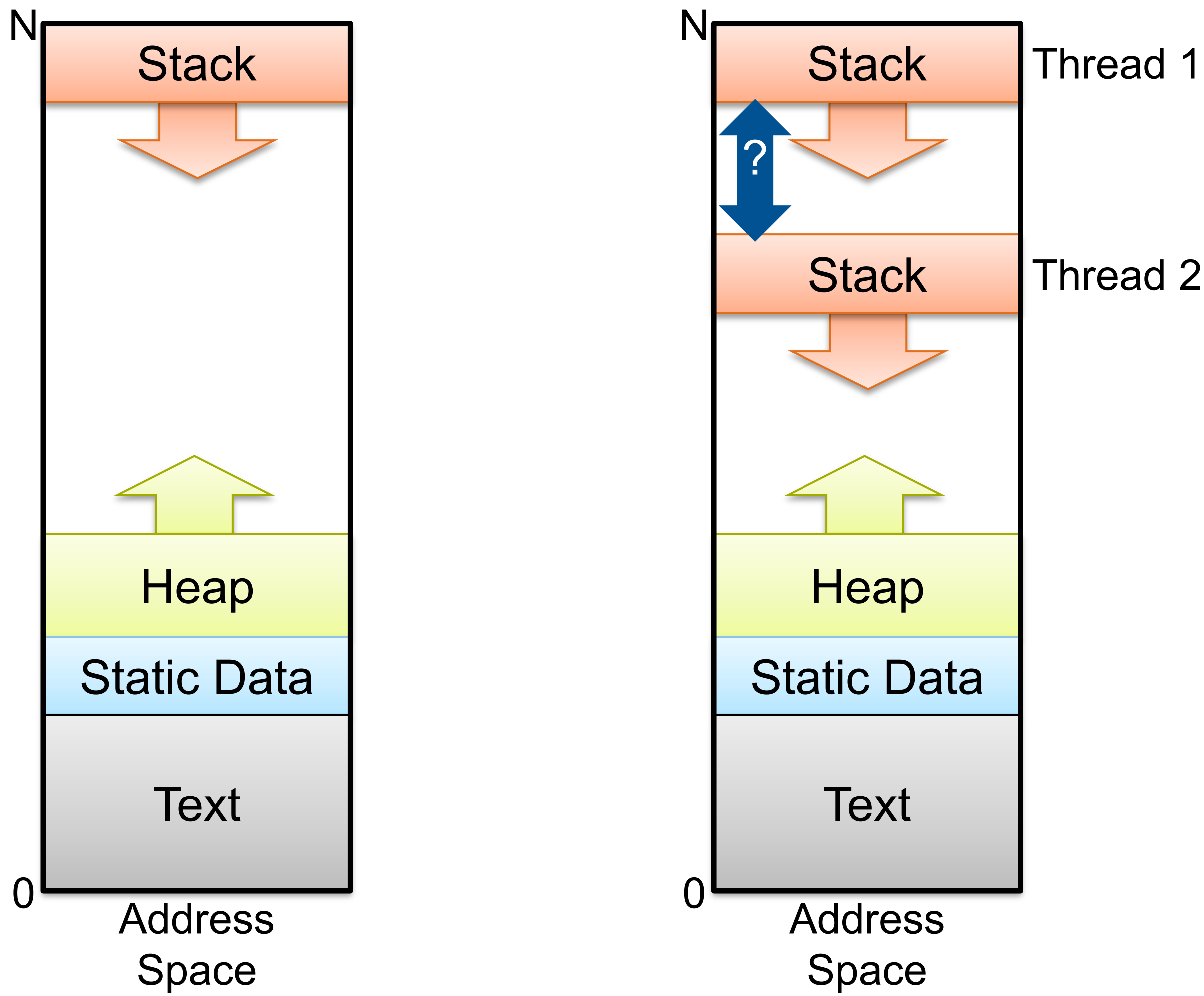
Thread management and special routines:

- 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)

```c
#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);
}
```

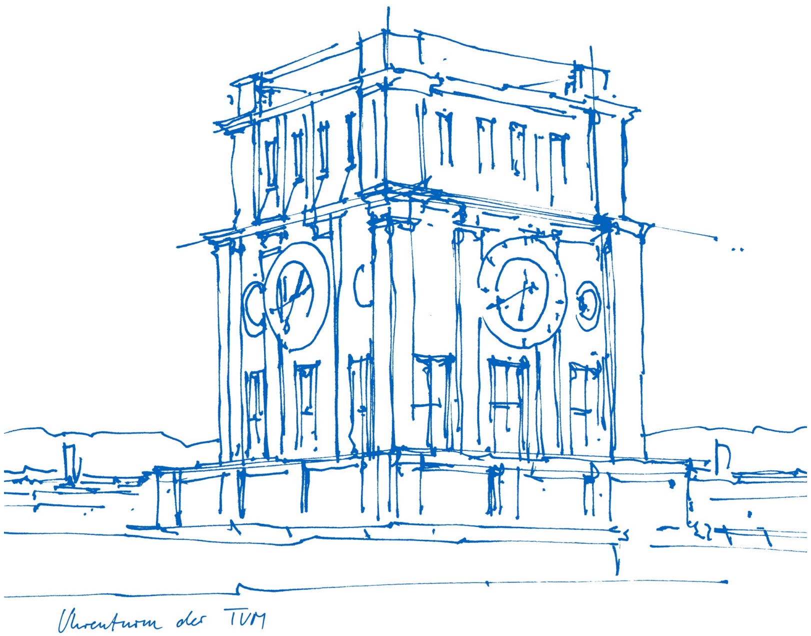
Source: Wikipedia

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
```

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering



Segment 9
Thread Synchronization

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 (passive waiting for resources)
- Condition variables (active signaling, covered in background material)

Other constructs can be built on top of them

Locks for 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
- Fairness: Every process eventually gets the mutex (and on average “equally fast”)

Spin-Locks

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

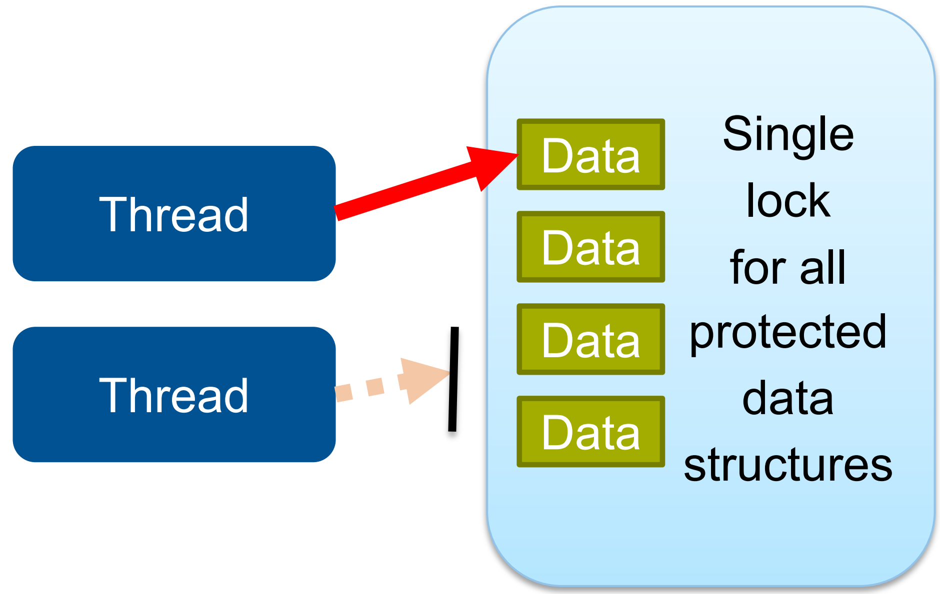
Yielding Locks

- If lock is taken, yield hardware thread
- Implementation typically via runtime system
- Advantage: low resource usage
- Disadvantage: slow response time
- Generally bad for HPC

Lock Granularity

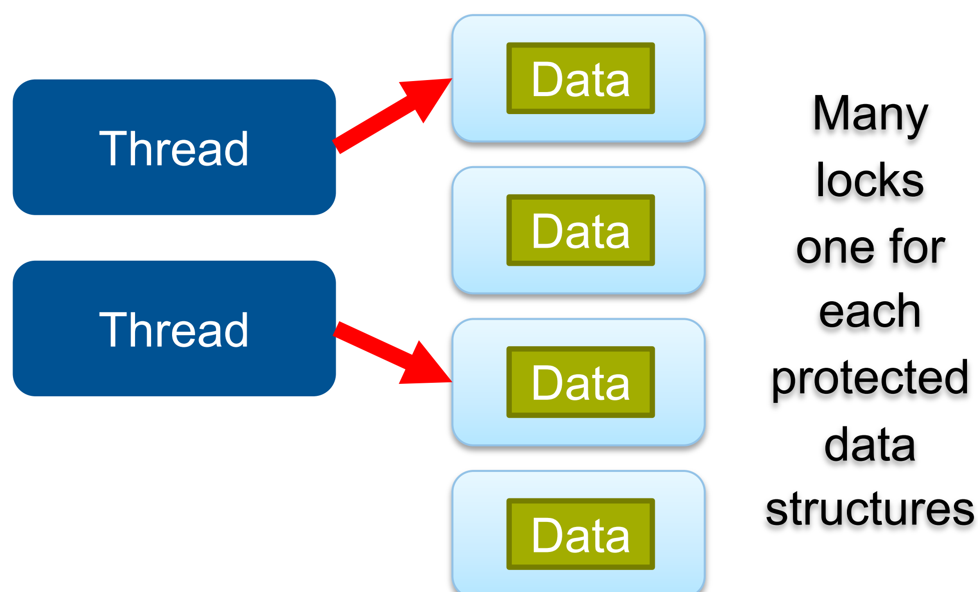
Coarse grained locking

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



Fine grained locking

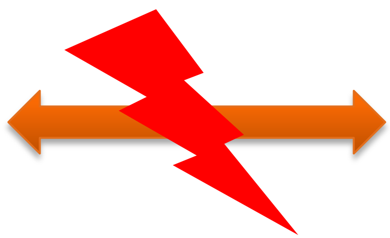
- One lock for each data element
- Maximizes concurrency (multiple threads can have locks)
- Requires many locking calls
- May require multiple locks at the same time



Hybrid versions often useful

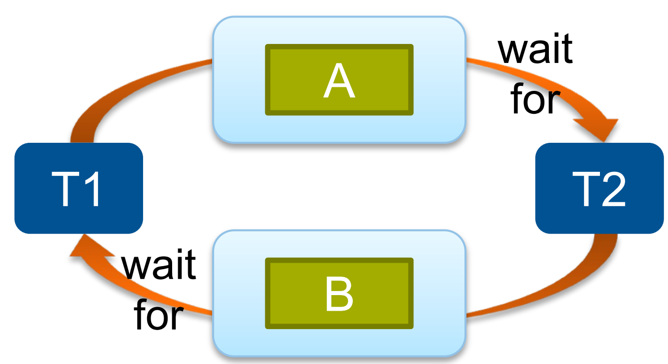
Danger: Deadlocks

```
Thread 1

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



```
Thread 2

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

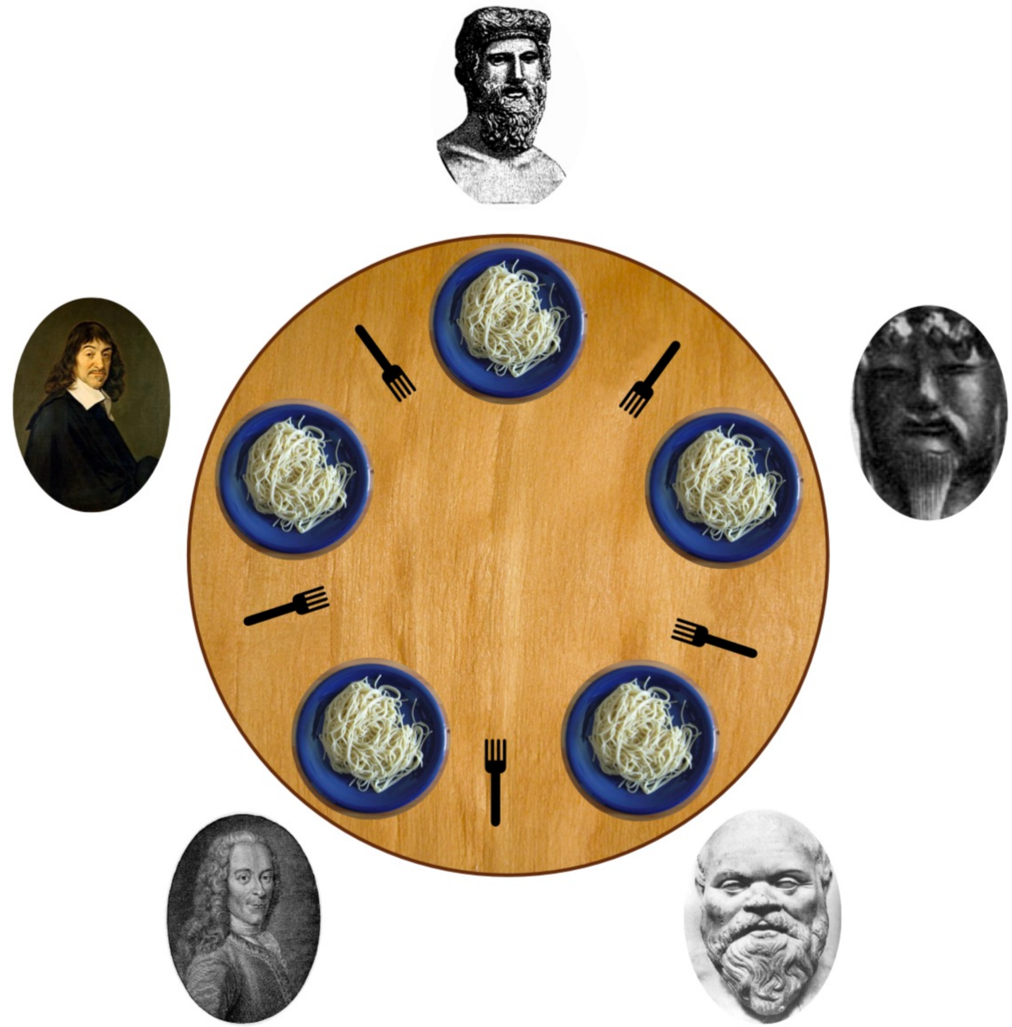
Both threads request lock A and B
- Both end up waiting for the second lock
- Cyclic dependency!



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

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

```
Thread 2

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

Classic Example: Dining Philosophers

All solutions have drawbacks

One lock

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

Arbiter

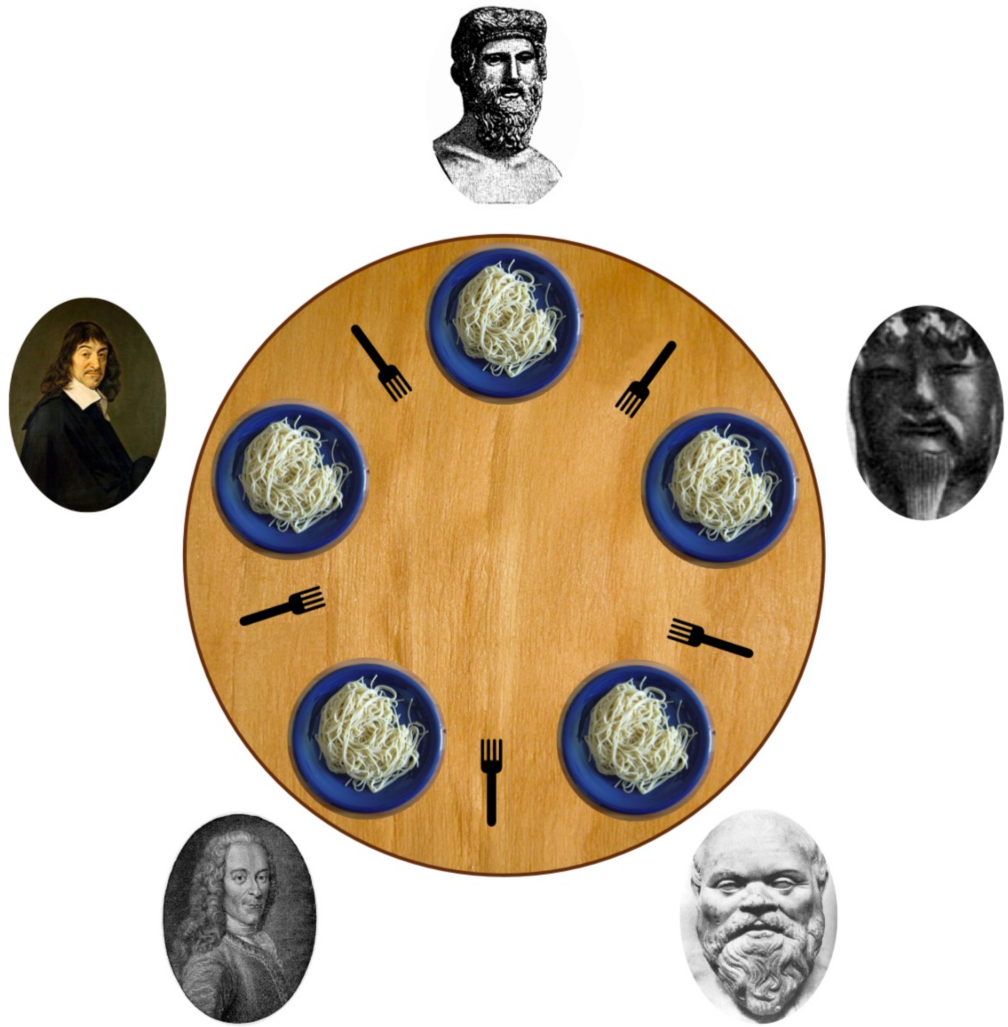
- Need central instance
- Might be complex to implement

Lock order

- Fairness
  (one philosopher may starve)

Custom schemes possible

- Requesting forks from neighbors
- Preference to starving processes



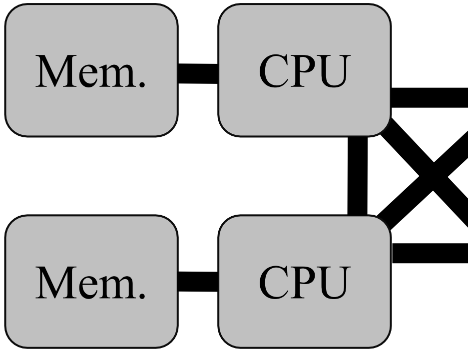
Source: Wikipedia

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`)

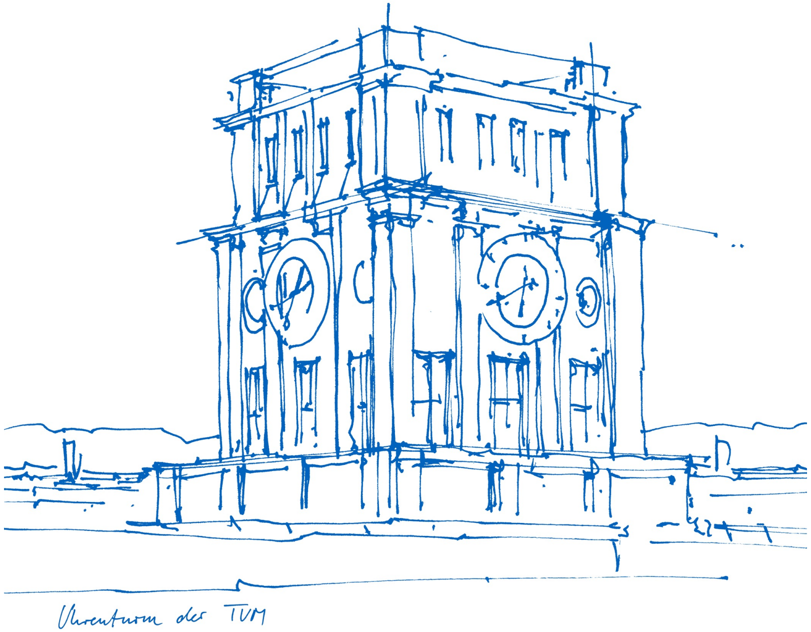


Good use of caches, avoidance of false sharing

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 10
Modern Thread APIs



Different Thread APIs

Native thread APIs per system

- POSIX Threads
- Win32 Threads
- Solaris Threads

Portable standards

- OpenMP

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

Native thread APIs per language

- C++ threads
- Java threads
- Rust threads

(requires a runtime system)

Language APIs can Simplify Usage (e.g., C++)

```
#include <string>
#include <iostream>
#include <thread>
using namespace std;

// The function we want to execute on the new thread.
void task1(string msg)
{ cout << "task1 says: " << msg; }

int main()
{
   // Constructs the new thread and runs it. Does not block.
   thread t1(task1, "Hello");

   // Do other things...

   // Join threads, blocks until completion
   t1.join();
}
```

Task to be run in thread

Fork

Join

On G++, compile with -std=c++0x -pthread.

Source: https://stackoverflow.com/questions/266168/simple-example-of-threading-in-c

Mutex Example C++

```
#include <iostream>
#include <thread
#include <mutex>

std::mutex mtx;

void print_thread_id (int id)
{
   mtx.lock();
   std::cout <<
      "thread #" << id << '\n';
   mtx.unlock();
}
```

Declaration

Lock

Unlock

```
int main ()
{
   std::thread threads[10];

   // spawn 10 threads
   for (int i=0; i<10; ++i)
      threads[i] =
         std::thread(
         print_thread_id,i+1);

   for (auto& th : threads)
      th.join();

   return 0;
}
```

Source: https://www.cplusplus.com/reference/mutex/mutex/lock/

Example: Java threads

```
public class ThreadExample
{
  public static void main(String[] args)
  {
    System.out.println(Thread.currentThread().getName());

    for(int i=0; i<10; i++)
    {
      new Thread("" + i)
      {
        public void run()
        {
          System.out.println("Thread: " + getName() + " running");
        }
      }.start();
} } }
```

New Thread Declaration

Task to be run
in thread (class)

Fork / Thread Start

Class also has a “join” method

Example: Java Locks

```
public class Counter
{
   private Lock lock = new Lock();
   private int count = 0;

   public int inc()
   {
      lock.lock();
      int newCount = ++count;
      lock.unlock();
      return newCount;
   }
}
```

Declaration

Lock

Unlock

Summary

Many thread libraries follow the fork/join model

- Mapped onto language constructs
- Very similar functionality
- Costs may vary widely, though, requiring different tradeoffs

Mutual exclusions is also a very basic primitive

- Given in many models for parallelism and concurrency
- Typical multiple options for locks (try-locks, multi-locks, etc.)

Often additional synchronization mechanisms

- Signaling like in condition variables
- Explicit dependencies

Implementation at the end based on low-level threads

- User level threads in the runtime
- OS threads or abstracts like Pthreads

Exercises with C++ threads