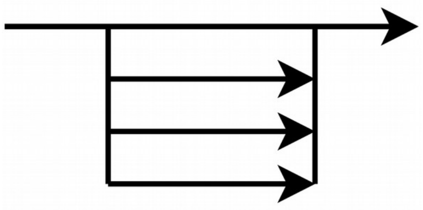
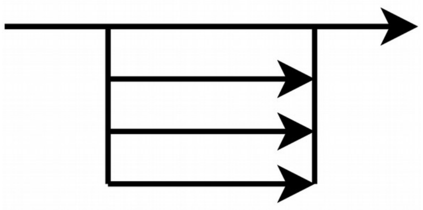
Pthreads Introduction

Parallel Computing

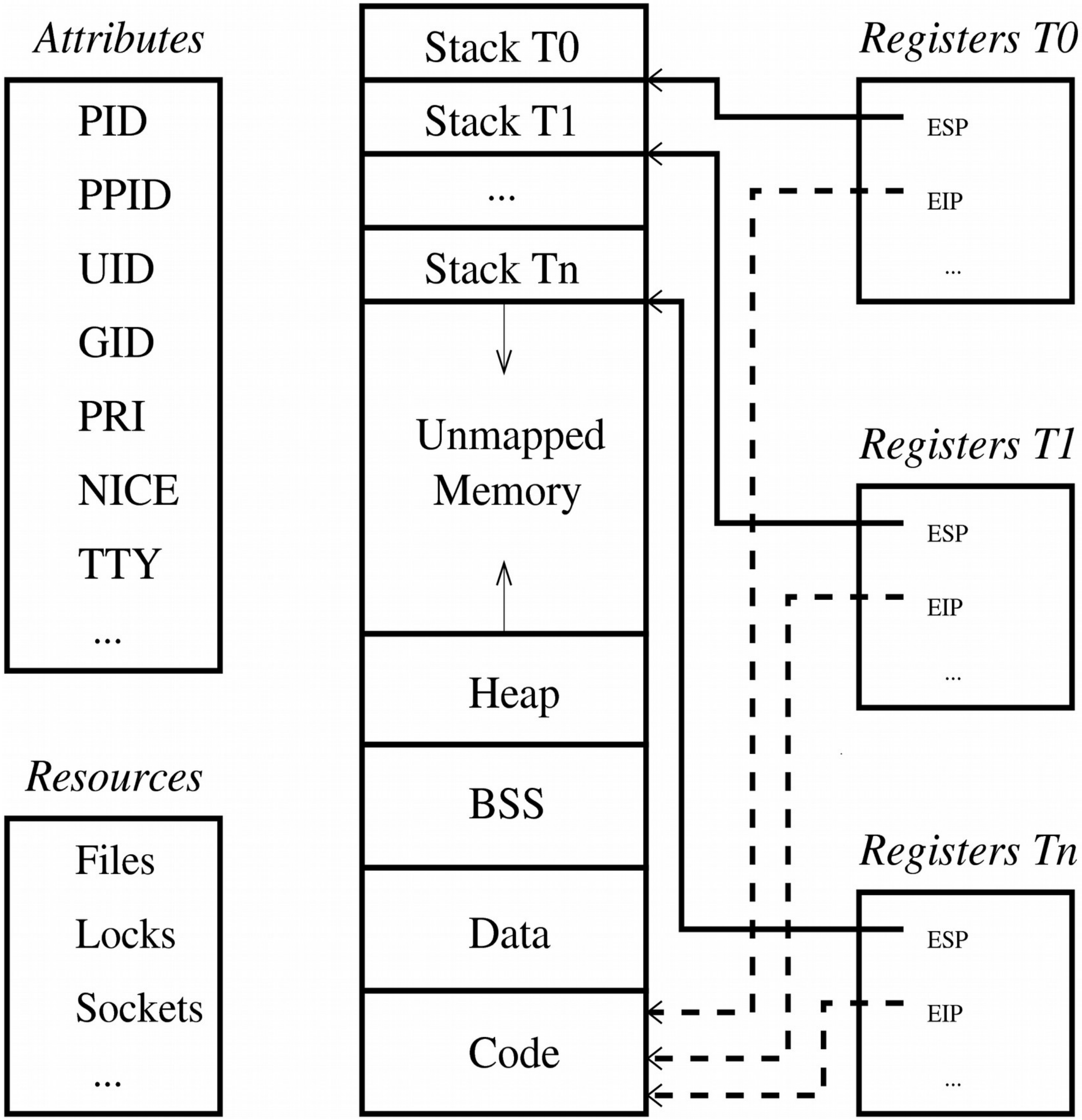
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria





Threads vs. Processes

- Process can have multiple threads
- Thread: “lightweight” process
- Threads share address space, file descriptors, sockets,...
- Per-thread stack, program counter, registers: thread's *context*
- Switching threads more efficient than switching processes
  - “lightweight” context



Benefits of Threading

Parallelism

- computing independent tasks at the same time
  - speed-up (Amdahl's Law!)
- need multiprocessor HW for “true” parallelism
- exploiting capabilities of modern multi-core processors

Concurrency

- progress despite of blocking (overlapping) operations
- no multiprocessor HW needed
- “illusion” of parallelism
  - analogy: multiple running processes in multi-tasking operating systems

Threaded programming model

- shared-memory (no message passing)
- sequential program: implicit, strong synchronization via ordering of operations
- threaded program: explicit code constructs for synchronizing threads
- synchronization clearly designates dependencies
- better understanding of “real” dependencies

Costs of Threading

Overhead (Synchronization, Computation)

- directly: more synchronization → less parallelism, higher costs
- indirectly: scheduling, memory architecture (cache coherence), operating system, calling C library,...

Programming discipline

- “thinking in parallel”
- careful planning
- avoidance of
  - deadlocks: circular waiting for resources
  - races: threads' speed (scheduling) determines outcome of operation

Debugging and Testing

- nondeterminism: timing of events depends on threads' speed (scheduling)
- bugs difficult to reproduce
  - e.g. what thread is responsible for invalid memory access?
- probe effect: adding debugging information can influence behaviour
- how to test possible interleavings of threads?

When (not) to Use Threads?

Pro threads

- independent computations on decomposable data
  - Example: `arraysum`
- frequently blocking operations, e.g. waiting for I/O requests
- server applications

Contra threads

- highly sequential programs: every operation depends on the previous one
- massive synchronization requirements

Challenges in Threaded Programming

- (applies to parallel computation in general)
- Amdahl's Law is optimistic (ignores underlying HW, operating system,...)
- keeping the sequential part small: less synchronization
- increasing the parallel part: data decomposition

Pthreads Basics

Parallel Computing

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

POSIX Threads

POSIX: Portable Operating System Interface

- IEEE standards defining API of software for UNIX-like operating systems

POSIX threads (Pthreads)

- standard approved 1995, amendments
- functions for
  - creating threads
  - synchronizing threads
  - thread interaction
- *opaque* data types for
  - thread identifiers
  - synchronization constructs
  - attributes
  - ...
- header file `pthread.h`
- compilation: `gcc -pthread -o prog prog.c`

References:

- D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997
- http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

(P)Threads in Linux

How can a thread-library be implemented?

Abstraction levels:

- threads: created by a user program
- kernel entity: “process”, scheduled by operating system
- processor: physical device, gets assigned kernel entities by scheduler

Design decision: how to map threads to kernel entities?

M-to-1:

- all threads of process mapped to one kernel entity
- fast scheduling (in library), but no parallelism

M-to-N:

- threads of process mapped to different kernel entities
- two-level scheduling (library and kernel) incurs overhead, but allows parallelism

1-to-1:

- each thread mapped to one kernel entity
- scheduling in kernel, less overhead than in M-to-N case, allows parallelism
- used in most modern Linux systems: *Native POSIX Threads Library (NPTL)*

Pthread Lifecycle: States

Ready

able to run, waiting for processor

Running

on multiprocessor possibly more than one at a time

Blocked

thread is waiting for a shared resource

Terminated

system resources partially released

but not yet fully cleaned up

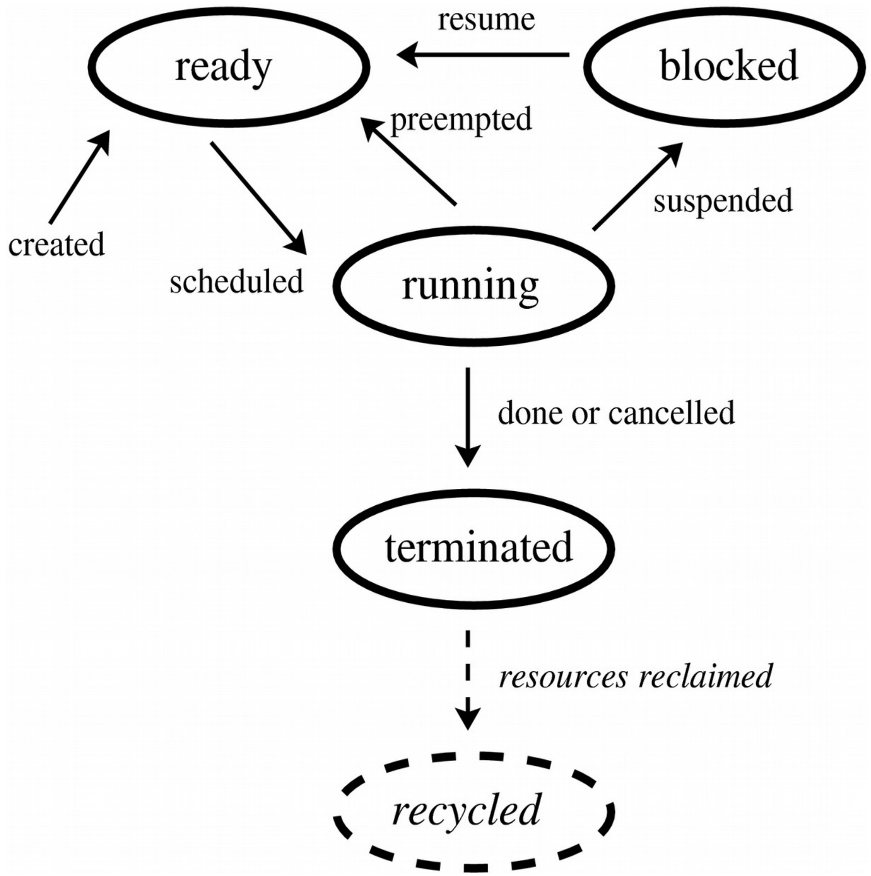
thread's own memory is obsolete

can still return value

(Recycled)

all system resources fully cleaned up

controlled by the operating system



Pthread Creation

`int pthread_create(arg0, arg1, arg2, arg3)`

- `arg0: pthread_t *tid_ptr`
  - where to store thread ID of type `pthread_t`
- `arg1: const pthread_att_t *attr`
  - may set certain attributes at startup
  - ignored for the moment: always pass `NULL` → set default attributes
- `arg2: void *(*start)(void *)`
  - pointer to thread's startup function
  - takes exactly one `void*` as argument
- `arg3: void *arg`
  - actual parameter of thread's startup function
- returns zero on success, else error code

thread ID is stored in `*tid_ptr`

- `pthread_t pthread_self()` returns ID of current thread
- `int pthread_equal(pthread_t tid1, pthread_t tid2)` compares IDs

Example: `helloworld`

`main`-Thread

Process creates thread which executes `main`-function → “`main`-thread”

`main`-thread behaves slightly differently from ordinary threads:

- termination of `main`-thread by returning from `main` causes process to terminate
  - all threads of process terminate
  - Example: `helloworld`
- calling `pthread_exit(...)` in `main`-thread causes process to continue
  - all created threads continue
  - recall lifecycle: `main`-thread terminates → resources partially released
    - Attention: stack may be released!
  - memory errors: dereferencing pointers into `main`-thread's (released) stack
  - Example: `helloworld_buggy`

Pthread Termination

generally: thread terminates if startup function returns

`int pthread_exit(void *value_ptr)`

- causes thread to terminate (special semantics in `main`-thread)
- implicitly called if thread's startup function returns (except in `main`-thread)
- `value_ptr` is the thread's return value (see `pthread_join(...)`)

`int pthread_detach(pthread_t tid)`

- resources of `tid` can be reclaimed after `tid` has terminated
- default: not detached
- any thread can detach any thread (including itself)

`int pthread_join(pthread_t tid, void **value)`

- returns when `tid` has terminated (or already terminated), caller blocks
- optionally stores `tid`'s return value in `*value`
  - return value from calling `pthread_exit(...)` or returning from startup function
- joined thread will be implicitly detached
- detached threads can not be joined

Pthread Termination - Examples

Example: `helloworld_join`

Returning values from threads

- returning values from threads via `pthread_join(...)`
  - example: `returnval`
  - but: waiting for termination often not needed
  - good practice to release system resources as early as possible
- alternative to `pthread_join(...)`: custom return mechanism
  - threads store their return values on the heap
  - Example: `returnval_heap`
    - problem: need to notify main-thread somehow that all threads have written results
- error: joining a detached thread
  - resources are (may be or not) already released
  - join should fail
  - Example: `returnval_buggy`
- error: returning pointer to local variable
  - Example: `returnval_buggy`

Pthread Lifecycle Revisited (1/2)

Creation

- process creation → `main`-thread creation
- `pthread_create(...)`: new threads are ready
  - no synchronization between `pthread_create(...)` and new thread's execution

Startup

- `main`-thread's `main` function called after process creation
- newly created threads execute startup function

Running

- ready threads are eligible to acquire processor → will be running
- scheduler assigns timeslice to ready thread → threads will be preempted
- switching threads → context (registers, stack, pc) must be saved

Blocking

- running threads may block, e.g. to wait for shared resource
- blocking threads become ready (not running) again

Pthread Lifecycle Revisited (2/2)

Termination

- generally: when thread returns from startup function
- `pthread_exit`
- can also explicitly be cancelled by `pthread_cancel(...)`
- (optional cleanup handlers are called)
- only thread's ID and return value remain valid, other resources might be released
- terminated threads can still be joined or detached
  - joined threads will be implicitly detached, i.e. all its system resources will be released

Recycling

- occurs immediately for terminated, detached threads → all resources released

Creating and Using Threads: Pitfalls

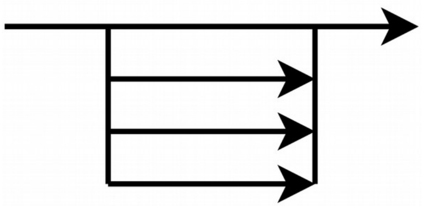
- Sharing pointers into stack memory of threads
  - perfectly alright, but handle with care
    - passing arguments
    - returning values
- Resources of terminated, non-detached threads can not fully be released
  - large number of threads → performance problems?
  - should join or detach threads
- Relying on the speed/order of individual threads
  - do not make any assumptions!
  - need mechanism to notify threads that certain conditions are true
    - example: `returnval_heap`
  - must prevent threads from modifying shared data concurrently
    - example: `sum`

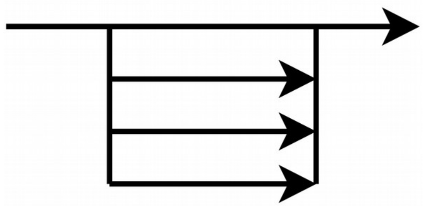
→ Synchronization

Pthreads Synchronization

Parallel Computing

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria





The Need for Synchronization

Threads operating on *shared data* concurrently:

- scheduling determines outcome of operations → race conditions
- can lead to violations of data invariants
  - integrity of data structures: queues, buffers,...

Classical example: concurrent transactions on bank account

Thread 1	Thread 2	Balance
read balance: €1000		€1000
	read balance: €1000	€1000
	set balance: €(1000 – 200)	€800
set balance: €(1000 – 100)		€900
give out cash: €100		€900
	give out cash: €200	€900

Thread *notification*

- inform one or more threads that certain condition has become true
- example: `returnval_heap`

Basic Pthread Synchronization Mechanisms

Controlling access to shared data

- **mutex**: mutual exclusion
- special kind of semaphore
- *locking* a mutex allows mutually exclusive access to shared data
- A mutex can be locked (“owned”) by exactly one thread at a time
  - lock attempt on already locked mutex will block calling thread until mutex unlocked

Thread notification

- `pthread_join(...)`: very limited, no notification
- **condition variables:** threads block until notified that condition has become true
- <u>always</u> combined with a mutex protecting the condition's data
  - testing and setting the condition must be performed under locked mutex
- multiple threads can block on a condition variable or be notified at a time
  - e.g. multiple consumers waiting at an empty queue of items
  - e.g. producer inserts items and notifies waiting consumers

Synchronization in Java:

- `synchronized` blocks and methods, `wait()` and `notify()`,`notifyAll()`

Pthread Mutexes (1/2)

Represented as variables of type `pthread_mutex_t`

- never copy mutexes!
- share mutexes by passing pointers

Static or dynamic allocation and/or initialization

- static initialization
  - macro `PTHREAD_MUTEX_INITIALIZER`
  - set default attributes
    - e.g. process/system-wide mutexes, real-time scheduling, priority-aware mutexes,...
    - attributes are beyond our scope
- dynamic initialization
  - `pthread_mutex_attr_t` for setting mutex's attributes
  - `int pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
    - pass NULL for `attr` to get default attributes
  - `int pthread_mutex_destroy(pthread_mutex_attr_t *attr)`
    - mutex becomes invalid, but can be re-initialized
- dynamic allocation and initialization
  - allocate mutexes on heap and initialize dynamically

Pthread Mutexes (2/2)

`int pthread_mutex_lock(pthread_mutex_t *mutex)`

- mutex is currently unlocked: caller will own mutex
- mutex is currently locked: caller blocks until mutex is unlocked
  - deadlock: recursively locking a mutex (unless mutex is set to be recursive)

`int pthread_mutex_trylock(pthread_mutex_t *mutex)`

- mutex is currently unlocked: caller will own the mutex
- mutex is currently locked: caller does <u>not</u> block
  - caller can e.g. enter alternative branch

`int pthread_mutex_timedlock(...*mutex, ...*expire)`

- mutex is currently unlocked: caller will own mutex
- `struct timespec *expire`: absolute timeout for blocking

`int pthread_mutex_unlock(pthread_mutex_t *mutex)`

- among multiple blocking threads, exactly one is selected to own mutex
- error: caller does not own mutex
- error: mutex is unlocked already

Example: `sum, prodcons`

Pthread Condition Variables (1/2)

Represented as variables of type `pthread_cond_t`

- like for mutexes: analogous functions for initialization, attributes,...
  - `PTHREAD_COND_INITIALIZER, int pthread_cond_init(...),...`

Always associated with exactly one mutex

- but: different condition variables may use same mutex
- condition must be tested and set under protection of mutex
- mutex must be properly locked and unlocked
- suggested usage pattern:

```
mutex_lock();
while (!condition) {
  mutex_unlock();
  non_busy_wait_until_notified();
  mutex_lock();
}
/* critical region: do some work... */
mutex_unlock();
```

Managed by Pthread condition variables (similar to Java):

- set of waiting threads, (un)locking the mutex, notification of waiting threads

Pthread Condition Variables (2/2)

Waiting on a condition variable

`int pthread_cond_wait(pthread_cond_t *cond, ... *mutex)`

caller must own mutex, will then block until notified

mutex is automatically unlocked before waiting and locked again if call returns

Notifying waiting threads

`int pthread_cond_signal(pthread_cond_t *cond)`

caller notifies one arbitrary thread waiting on `cond`

notified thread wakes up and locks mutex (its call of `pthread_cond_wait` returns)

`int pthread_cond_broadcast(pthread_cond_t *cond)`

caller notifies all threads waiting on `cond`

notified threads wake up (in arbitrary order) and contend for mutex

notifying threads need not own mutex (but recommended)

`pthread_cond_timedwait(... *cond, ... *mutex, ... *expire)`

`struct timespec *expire`: absolute timeout for waiting

if timed out or notified: call will return with mutex locked again

Examples: `prodcons_cond, returnval_heapcond`

Pthread Barriers

Represented as variables of type `pthread_barrier_t`

Synchronizing pool of threads at a specific point

`int pthread_barrier_init(...,unsigned int cnt)`

- must be called before using barrier
- `cnt`: number of threads waiting (calls of ..._`wait(...)`) before all can continue

`int pthread_barrier_destroy(pthread_barrier_t *b)`

- reset barrier to invalid state
- must call `pthread_barrier_init(...)` before using again

`int pthread_barrier_wait(pthread_barrier_t *b)`

- Calling thread will wait (i.e. block) until `cnt` threads have called ..._`wait(...)`
- Waiting threads are then released in arbitrary order
- Returns non-zero to exactly one arbitrary thread and 0 otherwise

Example: `simple-barrier`

In Java 1.5 or higher: CyclicBarrier

Memory Visibility

When will changes of shared data be visible to other threads?

Pthreads standard guarantees basic *memory visibility rules*

- thread creation
  - memory state before calling `pthread_create(...)` is visible to created thread
- mutex unlocking (also combined with condition variables)
  - memory state before unlocking a mutex is visible to thread which locks same mutex
- thread termination (i.e. entering state “terminated”)
  - memory state before termination is visible to thread which joins with terminated thread
- condition variables
  - memory state before notifying waiting threads is visible to woke up threads

Memory barriers:

- instructions issued implicitly to ensure memory visibility rules for pthreads
- impose order on memory accesses
- all memory accesses issued before barrier must complete before any access issued after the barrier can complete

`volatile` variables do <u>not</u> guarantee memory consistency!

Hints and Pitfalls (1/4)

Always wait in a loop on a condition variable (applies to any thread library)

- condition should be re-evaluated after waking up → why?
- intercepted wakeups
  - another thread might acquire mutex before the woke up thread and reset condition
- notification on weak predicates (programmer's responsibility)
  - e.g. notify if *n* <= *value*, but "tight" condition is *n* < *value* → unnecessary notifications
- spurious wakeups
  - library: more efficient to notify multiple threads at `pthread_cond_signal(...)`
  - programming errors: notification although the condition is false
  - pthread standard does not prevent wakeups without any notifying thread [Butenhof'97]

Beware of deadlocks

- threads wait for mutexes in circular fashion
- fixed locking hierachy: always lock mutexes in fixed order
- try and back off: unlock all mutexes in a set if one lock fails, then start again later
  - can lead to starvation: thread "polls" for mutex and never waits
- Example: `deadlock_backoff`

Hints and Pitfalls (2/4)

Beware of "badly optimizing" the use of condition variables

- lost wakeups: thread waits although condition is true
  - like `prodcons_cond`: producer signals only if buffer becomes non-empty → error
- do not share condition variables between predicates
  - do not know which predicate a notified thread was waiting for

Speed/order of threads

- do not assume anything!
- adding `sleep(...)` is not a bug fix (but can "hide" synchronization problems)

Hints and Pitfalls (3/4): Performance Concerns

Number of threads:

- cost of thread creation and context switches is system-dependent

Synchronization prevents concurrency and parallelism

- best solution: do not share too much (Example: `arraysum`)

Own mutexes for shortest possible time → reduces waiting time

Massive (un)locking of mutexes is expensive

- Example: `freq-locking`

Mutexes and condition variables consume memory

- Mutex: 40 (24) bytes in 64-bit (32-bit) environment
- Condition variables: 48 bytes in 32- and 64-bit environment

Hints and Pitfalls (4/4): Performance Concerns

Fine-grain locking

- using many “small” mutexes increases concurrency and locking overhead
- Example: `locked-array/many-locks`

Coarse-grain locking

- using few “big” mutexes decreases concurrency and locking overhead
- Example: `locked-array/big-lock`

Lock chaining

- e.g. lock(m1), lock(m2), unlock(m1), lock(m3), unlock(m2),...
- e.g. concurrent linked list: locking entire list or single nodes

Read/write locks: allow concurrent reads

- multiple readers may concurrently read if no writer is active
- one writer prevents any other writer or reader from accessing

Advanced Topics

Thread-specific data

- static data where each thread has a private value associated with a key

Attributes

- for threads, mutexes and condition variables

Cancellation

- cancel threads either immediately or at special cancellation points
- held resources need to be cleaned up properly (cleanup handlers)

Realtime scheduling

- setting scheduling policy and priorities, priority-aware mutexes

Thread-safe libraries

- how to make libraries thread-safe?
- must interfaces be changed?
- often inefficient: one “big” internal mutex protecting entire functions
- problem: functions which maintain internal state across calls

Spinlocks vs. mutexes

- busy waiting vs. non-busy waiting