„Systemnahe Programmierung“ (BA-INF 034) Wintersemester 2023/2024

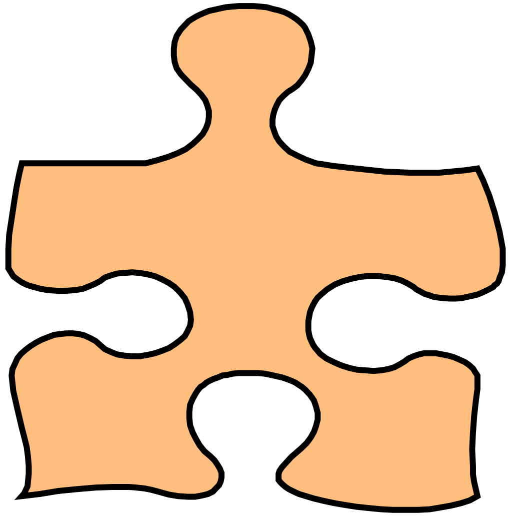
Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4
Universität Bonn

E-Mail: *{matthew, matthias.wuebbeling}@cs.uni-bonn.de*
Sprechstunde: nach Vereinbarung

2. Fortgeschrittene Konzepte der Systemprogrammierung

2. Betriebssysteme/Threads
„Kommunikation innerhalb von Prozessen bzw. zwischen Prozessen eines Rechners“



Teil 2

- 2.1. Processes
- 2.2. Threads
- 2.3. Interprocess Communication IPC
- …

Literature

- Mark Mitchell, Jeffrey Oldham, and Alex Samuel. Advanced Linux Programming. New Riders Publishing. First edition, 2001. Chapter 4. http://www.advancedlinuxprogramming.com/
- Abraham Silberschatz, Peter Baer Galvin, und Greg Gagne. Operating System Concepts. John Wiley & Sons. Eighth edition, 2008. Chapters 4 and 7.
- Andrew S. Tanenbaum. Modern Operating Systems. Prentice Hall. Third edition, 2007. Chapters 2 and 3.
- W. Richard Stevens, Stephen A. Rago. Advanced Programming in the UNIX Environment. Addison-Wesley. Second Edition, 2005. Chapter 11.
- Blaise Barney. POSIX Threads Programming Tutorial. Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/

2.2. Threads – Outline

- **2.2.1. Fundamentals**
- Threads in Linux
- Thread Synchronization
- Deadlocks
- Important Threading Mechanisms

2.2.1. Motivation

- A process – as discussed so far – is a program executed sequentially with a single thread of control

- Processes allow parallelism…
  - Different applications are executed in parallel (separate processes)
  - A single application can consist of several (cooperating) processes

- … but: Processes are **independent** execution units
  - Communication among processes is relatively complex
  - Switching context from process to process is expensive

- Idea: Support multiple threads of control within a single process

more on communication between processes in subsection 2.3. IPC

Threads (1)

- Threads provide **multiple execution flows** within one process
  - Threads operate within the environment of a process
  - All threads run independently, executing in parallel
- Threads are **less independent** than processes
  - Share the same address space
  - Can access the same global variables
- **No built-in protection** between threads
  - Data is shared among threads
  - Threads can access (and manipulate!) each others stack!
  - A thread might interfere with the execution of other threads

Threads (2)

- All **threads share**
  - Address space
  - Set of open files
  - Set of child processes
  - Sets of alarms and signals

- **Each thread needs a separate**
  - Stack – Stores local variables of the functions in execution
  - Program counter – “Pointer” to the next command in program code
  - Registers – Data currently used by the processor when executing the thread
  - State – Stores whether the thread is executing, blocked, …

Benefits of Using Threads

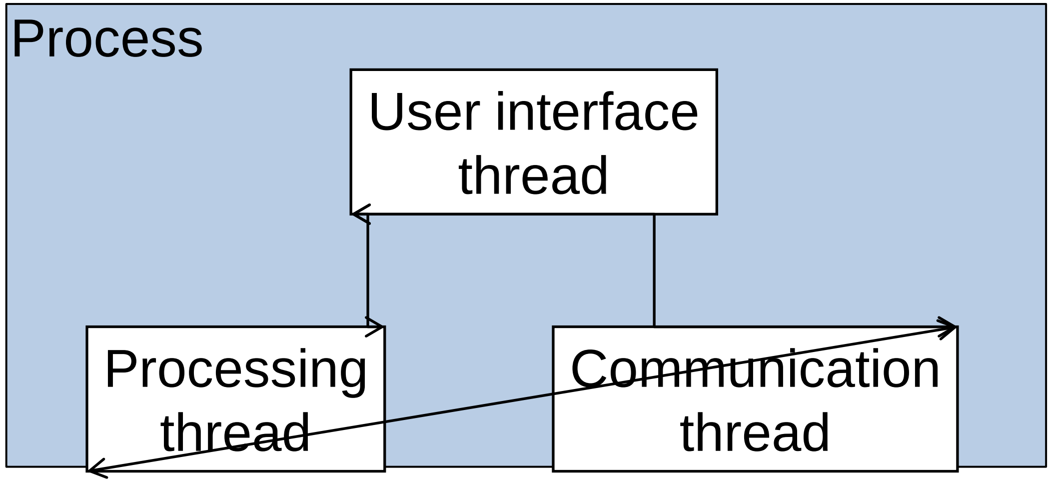
- **Parallel execution** of multiple activities
  - Sharing of processor time
  - Utilization of the processor while some activities are blocked

- Utilization of **multiprocessor / multicore systems**
  - Multiple threads of control can run truly in parallel on multiprocessor or multicore architectures

- **Responsiveness**
  - Long-running computations in a background thread do not block the main application thread

- **Resource sharing**
  - Threads operate in the same address space and can therefore easily share resources

Threads – Example Scenario



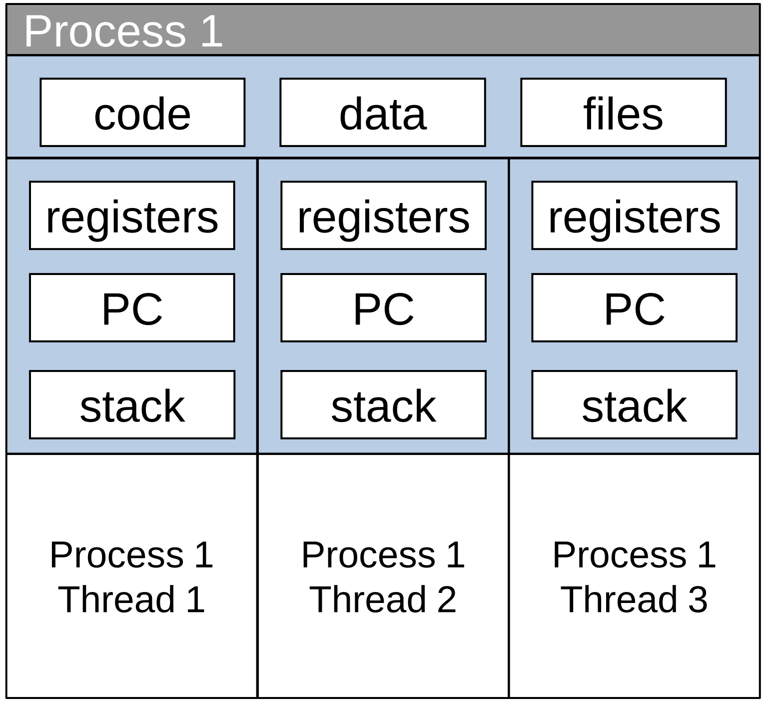
- User interface thread:
  - **Should remain responsive** even while the application processes data
  - Might be idle over longer periods of time

- Processing thread
  - Responsible for time-consuming computations
  - Requires a lot of CPU time

- Communication thread:
  - Sending, receiving and processing messages
  - Blocked while accessing external resources

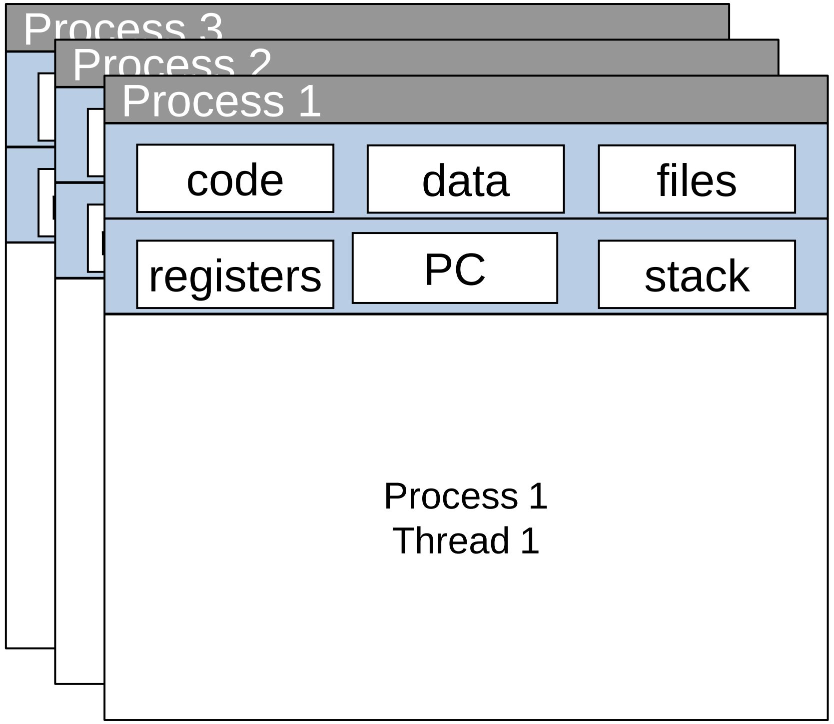
Threads vs. Processes (1)

Thread model



- All threads within a process share a common address space
- Fast context switches
- Efficient communication and cooperation

Process Model



- Processes are isolated from each other
- Communication and cooperation is complex

Threads vs. Processes (2)

- **Advantages** over Processes
  - Low context switching overhead
  - Shared state
  - Efficient communication and cooperation among threads
  - Easier to program?

- **Disadvantages**
  - Shared state: Threads can interfere with each others operation
  - Errors in one thread (e.g., a crash) can affect all others
  - More difficult to synchronize?

- Due to the similarities of threads and processes, threads are often called **lightweight processes**

Performance comparison – processes vs. threads

- *„ … the following table compares timing results for the* ***`fork()`*** *subroutine and the* **`pthreads_create()`** *subroutine.“*

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

- *„Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.“*

Table and *quotations* from:
Blaise Barney. POSIX Threads Programming Tutorial. Lawrence Livermore National Laboratory.
https://computing.llnl.gov/tutorials/pthreads/

UNIVERSITÄT BONN

Operating System Support for Threads

- **Threading support** can be provided on the
  - User level –> User threads
  - Kernel level –> Kernel threads

- **Multithreading** on the user level can be implemented following different models
  - Many-to-One (cf. slides 15 ff.)
  - One-to-One
  - Many-to-Many

User Threads

- Implemented by a thread library at the user level
  - Provides support for creating, managing and scheduling without support from the kernel

- Kernel is unaware of threads
  - Sees only a single thread of execution – the process

- **Advantages**
  - Fast to create and manage (No kernel involvement required)

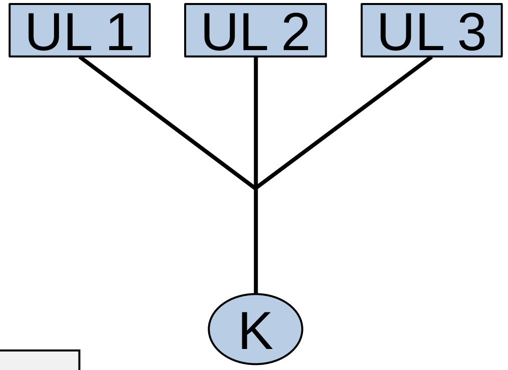
- **Disadvantages**
  - Cannot take advantage of multiprocessors or multicores
  - Blocking system call in one thread blocks all threads

Kernel Threads

- Implemented as part of the core operating system
- Kernel responsible for creation, scheduling and management
- **Advantages**
  - Able to take advantage of multiprocessors or multicores
  - Kernel can schedule another thread upon blocking on a system call in one thread
- **Disadvantages**
  - Slower to create and manage than user threads

Multithreading Models (1)

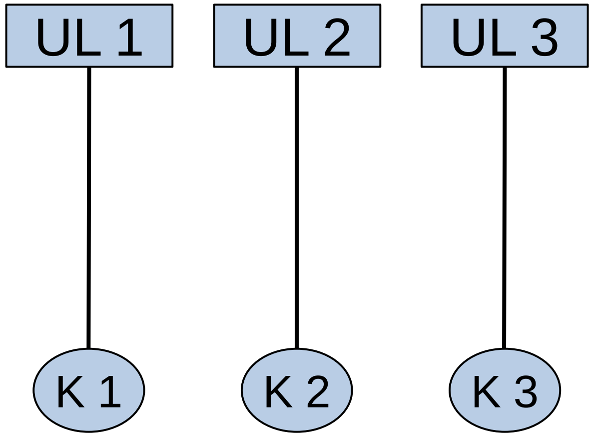
- Many-to-One
  - Many user-level threads are mapped to a single kernel-level thread
  - Has all the advantages and disadvantages of user threads
  - Example: GNU Portable Threads




```
int main(void) {
        int thread = 0;
        while(1) {
        if thread == 0 {
        thread0();
        thread = 1;
        }
        else {
        thread1();
        thread = 0;
        }
        }
}
```

Multithreading Models (2)

- One-to-One
  - Maps each user-level thread to a kernel-level thread
  - Has all the advantages and disadvantages of kernel threads
  - Example: **Linux, Windows, Solaris**

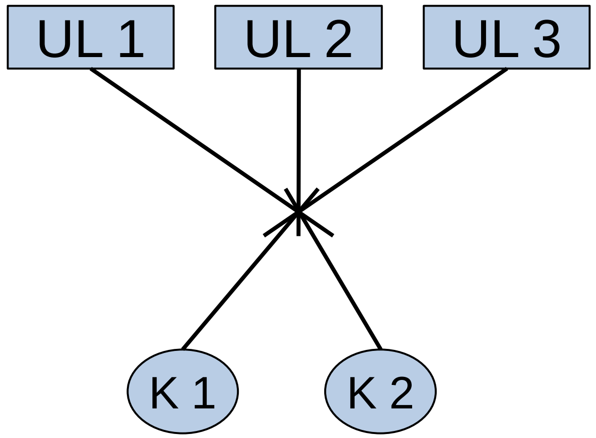


Multithreading Models (3)

- Many-to-Many
  - Multiplexing many user-level threads to many kernel-level threads
  - Number of kernel-level threads ≤ number of user-level threads

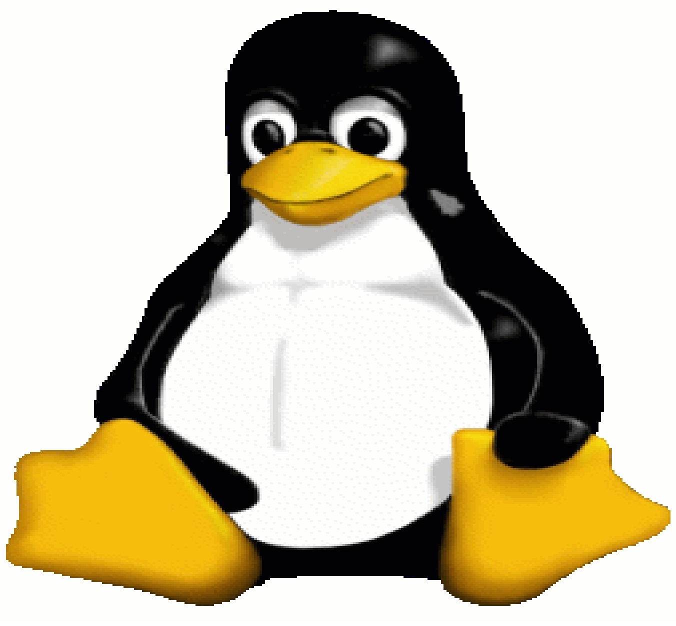
  - Mapping decision based on various criteria:
    - Application properties
    - Hardware properties (e.g., number of cores)

    –> Most flexible model
  - Example: Older versions of Solaris
  - Golang revives N:M mapping



Overview

- Fundamentals
- **2.2.2. Threads in Linux**
- Thread Synchronization
- Deadlocks
- Important Threading Mechanisms



2.2.2. Implementation of Threads in Linux (1)

- On the operating system level, Linux does not really distinguish between processes and threads
  - Uses the term task to cover both
  - Difference: Threads share their address space with each other

- A new thread is created using the `clone()` system call
  - Generates a new child process just like the `fork()` system call
  - Allows the child process to share parts of its execution context with the calling process
  - Flags allow to configure what is to be shared, e.g., the memory space, the file descriptor table, the signal handlers, …

Implementation of Threads in Linux (2)

- Effects
  - Threads in Linux are usually based on the **one-to-one multithreading model**
  - Originally, each thread not only had a different thread ID but also a different process ID – this has changed in newer kernel versions…

- **What does this mean for the programmer?**
  - `clone()` system call is usually hidden behind a user-level thread implementation like `Pthreads`
  - Usually: No need to worry about the implementation – but always good to be aware of how it is mapped internally.

Implementation of Threads in Linux (3)

- (Recently) new possibility: C11 (this is NOT C++11 !)
  - Features the new `<threads.h>`
  - APIs very similar to `pthread`
  - Support for: Threading, Mutual Exclusion, Condition variables, Thread-local storage
  - Link with API: https://en.cppreference.com/w/c/thread
  - **Support in `glibc`** since Version 2.28 but…
  - Glibc 2.3 supports “Native POSIX Thread Library” providing pthread_* syscalls in Linux.

POSIX Threads

- **POSIX:** "Portable Operating System Interface" for software compatible with variants of the UNIX operating system
  - Family of standards from the area of operating systems
  - Administered by the IEEE (Institute of Electrical and Electronics Engineers)

- POSIX Threads (or **PThreads**)
  - Standard programming interface for threads
  - Defines procedures for creating, manipulating and destroying threads
  - Implementations exist for various operating systems (albeit not all fully standards compliant)

POSIX Threads in Linux

- Implementations provided by the GNU C library on Linux
  - Originally: LinuxThreads (Now obsolete but still available)

  - Since the Linux 2.6 kernel: NPTL (Native POSIX Threads Library)

  - Note: The description in the "Advanced Linux Programming" book is still based on Linux Threads!
- Functions and data types are declared in `<pthread.h>`
- Included in the library `libpthread`
  - Add `-lpthread` as a parameter to your linker call

PThreads Overview

- Tasks
  - Creating threads
  - Managing thread attributes
  - Passing parameters to threads
  - Exiting threads
  - Joining threads
  - Cancelling threads

–> Explained in detail on the following slides

PThread Thread Creation

- Create a new thread, specify which function it should execute and initiate the execution of the new thread:

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

  - `thread` – Data structure representing the created thread (= thread identifier)
  - `attr` – The attributes of the new thread. If set to `NULL`, then the default attributes will be used.
  - `start_routine` – Pointer to the thread function executed by the newly created thread
  - `arg` – Arguments passed to the start routine

- Return value: 0 if OK, error code otherwise
- PThread library executes clone for each create internally

PThread Thread Function

- Initial function executed by a new thread
  - Might call any other function
  - Returning from this function is one way of terminating the thread.

- Function of type: `void* threadFunctionName (void*)`
  - Any function of this type qualifies as a thread function
  - Allows any type for both the argument and the return value –> Use casting for accessing thread-specific data

- A function pointer of type `void* (*) (void*)` must be provided to create a new thread
  - New thread executes the thread function specified by the function pointer

PThread Thread Creation – Example

declaration of thread function

```
#include <pthread.h>
#include <stdio.h>

pthread_t workThreadID;

void *workThreadFunction(void *thread_arg) {
        printf("Your work thread says: Hello world!\n");
        return ((void *) 0);
}

int main(void) {
        pthread_create(&workThreadID, NULL, workThreadFunction, NULL);
        printf("Your main thread says: Hello world!\n");
        sleep(2);
        return 0;
}
```

creation of thread
(no attributes, no args)

PThread Thread Attributes (1)

- Can be used to **configure the fundamental behavior** of a new thread

- Initializing
  - `int pthread_attr_init(pthread_attr_t *attr);`
  - Initializes `attr` with the default value for all attributes

- Destroying
  - `int pthread_attr_destroy(pthread_attr_t *attr);`

- The available set of thread attributes is implementation dependent.

- Accessing attributes (`set` & `get`)
  - `int pthread_attr_set`***`AttrName`***` ( pthread_attr_t *attr, AttrType t);`
  - `int pthread_attr_get`***`AttrName`***` ( pthread_attr_t *attr, AttrType *t);`

PThread Thread Attributes (2)

- **Detach state** (exact semantics will be explained later – cf. slide 34 ff.)
  - `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);`
  - `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);`
  - `detachstate` – either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`

- **Thread Scheduling**
  - `pthread_attr_setschedpolicy` / `pthread_attr_getschedpolicy`
  - `pthread_attr_setscope` / `pthread_attr_getscope`
  - `pthread_attr_setinheritsched` / `pthread_attr_getinheritsched`

- **Stack size and stack address**
  - `pthread_attr_setstacksize` / `pthread_attr_getstacksize`
  - `pthread_attr_setstackaddr` / `pthread_attr_getstackaddr`

PThread Thread Attributes – Example

- Creating a thread in the detached state:

```
int main(void) {
    pthread_attr_t wtAttr;

    pthread_attr_init(&wtAttr);
    pthread_attr_setdetachstate(&wtAttr, PTHREAD_CREATE_DETACHED);
    pthread_create(&workThreadID, &wtAttr, workThreadFunction, NULL);
    pthread_attr_destroy(&wtAttr);

    [...]
}
```

- Remark: Changes to `wtAttr` after calling `pthread_create` have no effect on the thread

PThread Thread Parameter Passing

Function of type: `void* threadFunctionName (void*)`

- Useful to **pass data to a newly created thread**

- Each thread function expects exactly one parameter of type `void*`
  - Flexible type of the argument
  - Limitation: Only a single parameter possible

- Solution:
  - Define a structure for each thread that contains the expected parameters
  - Pass a pointer to the structure as the single argument to the thread

- Be careful: **Consider the lifetime of variables** passed to a thread using pointers

PThread Thread Parameter Passing – Example

```
struct wtParams {
        int param1;
        char param2;
};

void* workThreadFunction(void* thread_arg) {
        struct wtParams* wtp = (struct wtParams*) thread_arg;
        printf("Param 1: %d\n", wtp->param1);
        printf("Param 2: %c\n", wtp->param2);
        return ((void*) 0);
}

int main(void) {
        struct wtParams args;
        args.param1 = 4;
        args.param2 = 'x';
        pthread_create(&workThreadID, NULL, workThreadFunction, &args);
        [...]
}
```

argument passed and used as a pointer!

in this case: pointer to global struct

- Be careful: **Consider the lifetime of variables** passed to a thread using pointers (do not do this at home)!

UNIVERSITÄT BONN

PThread Ending / Exiting a Thread …

… three ways to do this!

1. **Implicitly**
   - Returning from the thread function (see example on prev. slide)
   - Return value of the thread function is used as return value of the thread

2. **Explicitly using pthread_exit**
   - `void pthread_exit(void *value_ptr);`
     - `value_ptr` – The return value of the thread
   - Terminates the calling thread and makes `value_ptr` available as the return value of the thread

3. Thread can be **cancelled by another thread** in the same process
   - See below

PThread Joining a Thread

- Problem: How to determine when a thread is finished?
  - Example: Main thread should wait until all other threads are done before ending the program (Note: ugly workaround using sleep in the first example – slide 28).
  - Second example: Thread needs return value of a child thread it created previously.

- We need a function that blocks until a thread is finished and then collects the return value of the thread: `pthread_join`

- `int pthread_join(pthread_t thread, void **value_ptr);`
  - `thread` – Identifier of the target thread
  - `value_ptr` – The result value passed to `pthread_exit` by the terminating thread

UNIVERSITÄT BONN

PThread Joining a Thread – Example

```
void* workThreadFunction(void* thread_arg) {
        int answer = 42;
        return ((void*) answer);
}

int main(void) {
        int threadResult;
        pthread_create(&workThreadID, NULL, workThreadFunction, NULL);
        pthread_join(workThreadID, (void*) &threadResult);
        printf("Thread result: %d\n", threadResult);
}
```

return value is integer, but casted to pointer type!

result available in integer variable

- Be careful again: Consider lifetime of variables returned as result!
  - Do not return pointer to data placed on the stack of the thread!
  - Make clear who is responsible for freeing dynamic memory.
  - Do not try this at home!?

Source

PThread Thread Detach State (1)

- A thread can be in one of two states: **`joinable`** (default) or **`detached`**
  - Using the `detachstate` attribute at thread creation time
  - Calling `pthread_detach` to detach a running thread
    - `int pthread_detach(pthread_t thread);`
    - **Inverse operation (reattaching a detached thread) is not possible!**

- Joinable threads
  - Not automatically cleaned up when the thread terminates
  - Requires another thread to call `pthread_join` to clean up the exit state of the thread
  - Allows collecting result data from the thread

PThread Thread Detach State (2)

- Detached threads
  - Immediately cleaned up when the thread terminates
  - Impossible to synchronize on their completion
  - No retrieval of result data possible
  - Only useful if
    - Thread is completely independent or
    - Thread communicates its (result) data to its peers differently (e.g., using shared state)

PThread Cancelling Threads (1)

- Up to now: Every thread runs to completion
  - Returning from its thread function
  - Calling `pthread_exit`

- There can be a need to stop a thread prematurely, for example:
  - All threads must be stopped prior to exiting the program.
  - The user cancelled the operation.
  - The results of the computations of a thread are not required anymore.

- Solution: Cancelling threads
  - `int pthread_cancel(pthread_t thread);`

- Note: Later we discuss whether this is a good solution…

PThread Cancelling Threads (2)

- Each thread manages its own cancelability
  - `int pthread_setcancelstate(int state, int *oldstate);`
    - Enable cancelling by setting state to `PTHREAD_CANCEL_ENABLE`
    - Disable cancelling by setting state to `PTHREAD_CANCEL_DISABLE`
    - `oldstate` returns the previous cancel state of the thread
  - Attempts to cancel a thread are ignored if cancelling is disabled

- Two cancelling types are possible
  - `int pthread_setcanceltype(int type, int *oldtype);`
    - Set to deferred (synchronous) by setting type to `PTHREAD_CANCEL_DEFERRED`
    - Set to asynchronous cancelable by setting type to `PTHREAD_CANCEL_ASYNCHRONOUS`
    - `oldtype` returns the previous cancel type of the thread

PThread Cancelling Threads (3)

- Asynchronously cancelable
  - Thread can be cancelled at any point of its execution – irrespective of the operations it currently performs or the state it is in

- Synchronously cancelable (`DEFAULT`)
  - Thread can only be cancelled at specified cancellation points
  - Explicit cancellation points
    - Specified in the code by the programmer
    - Calling: `void pthread_testcancel(void);`
- Implicit cancellation points
  - Cancellation points are automatically added with other commands, for example `pthread_join`, `pthread_cond_wait`, `sem_wait`, …

PThread Cancelling Threads – Example

```
void* workThreadFunction(void* thread_arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    // Do fancy stuff...
    pthread_testcancel();
    // Do even more fancy stuff...
}

int main(void) {
    pthread_create(&workThreadID, NULL, workThreadFunction, NULL);
    pthread_cancel(workThreadID);
    pthread_join(workThreadID, NULL);
}
```

- Where is the thread cancelled?
  - Which parts have been executed when `pthread_cancel` is called?
  - What if “Do fancy stuff” or one of the functions called in there include an implicit cancellation point?

Cancelling Threads – Disadvantages

- Asynchronous canceling is not applicable in most scenarios
  - We have to deal with aborts at any point of the execution flow.
  - Might leave resources accessed by the thread in an inconsistent state

- Synchronous canceling is difficult to get it right
  - Are all implicit and explicit cancellation points correctly covered?
  - What do the sub routines do that might affect the cancelling behavior?

- Recommendation:
  - Avoid canceling threads in most application scenarios.
  - Explicitly communicate the request to exit to the thread using a shared state variable and react in the thread accordingly.


USE MULTITHREADING, THEY SAID
IT'S EASY, THEY SAID
quickmeme.com

Recap

- Threads as “lightweight processes“
- **Advantages**
  - Low context-switching overhead
  - Shared state and memory
  - Efficient communication among threads
- **Disadvantages**
  - Shared state and memory
  - If threads crash, the entire process is crashed
  - Synchronization
- Various types of threads
  - Kernel vs. user threads
  - 1:1, n:1, n:m mapping

Overview

- Fundamentals
- Threads in Linux
- **2.2.3. Thread Synchronization**
- Deadlocks
- Important Threading Mechanisms

2.2.4. Thread Synchronization Problem

- Threads execute independently of each other
  - No particular ordering of events is guaranteed
  - Each program execution can result in a different execution sequence
  - Possible on multiprocessor / multicore systems: Truly parallel execution of commands

- Threads are not independent of each other
  - Threads accessing the same data
  - One thread building upon the results of another thread
  - Threads requiring the same resources (e.g., a file handler)
  - …

Race Conditions

- The result of a computation critically depends on the sequence or the timing of events
  - Multiple threads access a shared resource
  - Operations are performed in parallel / quasi parallel
  - Some proper order of operations is implicitly required but the system fails to enforce it

Race Conditions – Example (1)

- Adding integer values to a queue

```
struct listNode {
    struct listNode* succ;
    int value;
};

struct listNode* firstNode = NULL;

void addValue(int value) {
    struct listNode* newNode = (struct listNode*) malloc(sizeof(struct listNode));
    newNode->value = value;
    newNode->succ = firstNode;
    firstNode = newNode;
}
```

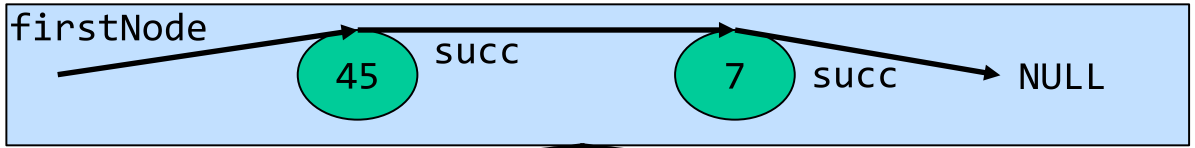
Thread 1: `addValue(42);`

Thread 2: `addValue(13);`

Source

Race Conditions – Example (2)

Start situation:



Sequence of commands

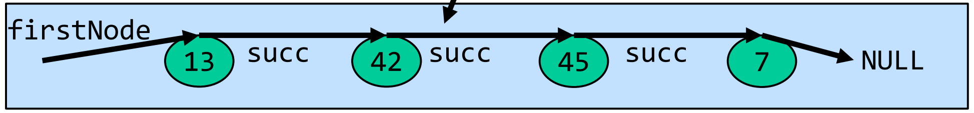
Schedule 1

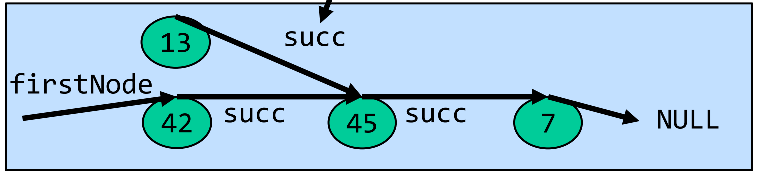
Thread 1: Command 1
Thread 1: Command 2
Thread 1: Command 3
Thread 1: Command 4
Thread 2: Command 1
Thread 2: Command 2
Thread 2: Command 3
Thread 2: Command 4

Schedule 2

Thread 1: Command 1
Thread 1: Command 2
Thread 1: Command 3
Thread 2: Command 1
Thread 2: Command 2
Thread 2: Command 3
Thread 2: Command 4
Thread 1: Command 4

Result






```
void addValue(int value) {
    struct listNode* newNode = (struct listNode*)
    malloc(sizeof(struct listNode));
    newNode->value = value;
    newNode->succ = firstNode;
    firstNode = newNode;
}
```

Critical Sections (also called: Critical Regions) (1)

- Definition: A **critical section** is a part of the code that accesses shared resources that must not be concurrently accessed by more than one thread of execution.
  - Example: Command 3+4 of the function `addValue` together form a critical section

- Insufficient definition: A critical section is a piece of code that can only be executed by one thread at a time
  - Other parts of the code might access the same resource –> All these code parts together belong to the critical section.
  - We have to protect the resources not the code sections!

- We need **synchronization primitives** to prevent the concurrent access to critical sections

Critical Sections (also called: Critical Regions) (2)

- Tanenbaum’s four conditions for a good solution of the critical section problem:
  1. No two threads may be inside their critical section simultaneously.
  2. No assumptions are to be made about speeds or the number of CPUs.
  3. No thread running outside its critical region may block other threads.
  4. No thread should have to wait forever to enter its critical region.

Thread Synchronization

- Goals
  - Prevent concurrent access to critical sections
  - Provide control over the ordering of relevant system events
- Approaches

  2.2.3.1. **Mutex** variables
  2.2.3.2. **Condition** variables
  Primary means of implementing thread synchronization
  2.2.3.3. Semaphores
  2.2.3.4. Barriers
  2.2.3.5. Read-write locks
  –> Explained in detail on the following slides

2.2.3.1. Mutex Variables – Motivation

- **Mutual Exclusion**
- **Goal:** Protect a critical section against simultaneous access by two or more threads
    - Thread is only allowed to enter if no other thread is in the critical section
    - Once a thread leaves a critical section, it should be accessible to other threads again

Mutex Variables – Basic Concept

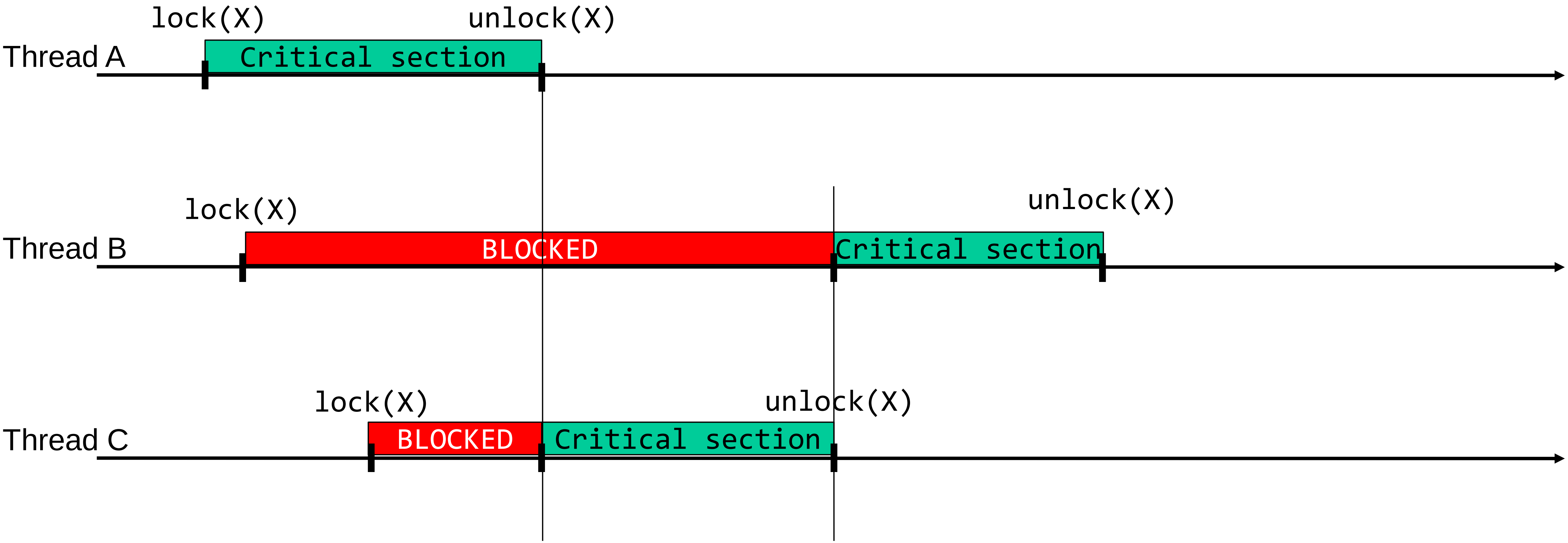
- Critical regions are **protected with** the help of **locks**
  - Thread must acquire lock before entering the critical region
  - Thread releases lock upon leaving the critical region

- **Invariant: At any given time, only one thread can have a lock on a mutex variable**

- Thread A requests a lock on mutex X. Two cases:
  - No thread holds a lock on X: A acquires the lock and is able to proceed
  - Another thread already holds a lock on X: A is blocked

- Thread A releases its lock on mutex X
  - Unblock one of the threads waiting for X (if such a thread exists)

Mutex Variables – Example



Note: No FIFO guarantees provided!

PThread Mutexes

- Data structures
  - `pthread_mutex_t` – Mutex variable data type
  - `pthread_mutexattr_t` – Mutex attributes data type

- Initializing a mutex variable
  - Dynamically
    - `int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);`
  - Statically at declaration time
    - `pthread_mutex_t mutexVar = PTHREAD_MUTEX_INITIALIZER;`

- Destroying a mutex variable
  - `int pthread_mutex_destroy (pthread_mutex_t *mutex);`

PThread Mutex Attributes

- Initializing
  - `int pthread_mutexattr_init(pthread_mutexattr_t *attr);`
  - Initializes attr with the default value for all attributes

- Destroying
  - `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`

- Important mutex attribute: Setting / getting the type of a mutex
  - `int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type);`
  - `int pthread_mutexattr_gettype (const pthread_mutexattr_t *attr, int *type);`
  - Example:
    - `pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);`

PThread Mutex Types

- `PTHREAD_MUTEX_NORMAL`
  - Attempting to lock a mutex twice from the same thread results in a deadlock; behavior when unlocking unlocked mutexes undefined
  - Fast mutex – no checks or counters required
- `PTHREAD_MUTEX_ERRORCHECK`
  - Relocking a locked mutex or unlocking an unlocked mutex results in an error
- `PTHREAD_MUTEX_RECURSIVE`
  - Multiple locks from the same thread possible – requires the same number of calls to `pthread_mutex_unlock`
- `PTHREAD_MUTEX_DEFAULT`
  - Recursive locking and unlocking error behavior implementation dependent

PThread Locking and Unlocking Mutexes

- Acquiring a lock on a mutex
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
    - Blocks the thread until the lock can be acquired
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
    - Never blocks the thread but indicates the success of the locking attempt in its return value
      - `0` – Mutex has been locked successfully.
      - `EBUSY` – Mutex has not been locked, because it was already locked.
- Releasing a lock on a mutex
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

PThread Mutexes – Example

- Revisiting the race condition example:
- Only one thread can access the marked code `(*)` at a time – others are blocked
- Prevents race condition in adding values to the queue
- No guarantees which thread can add its node first!

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void addValue(int value) {
    struct listNode* newNode = (struct listNode*) malloc(sizeof(struct listNode));
    newNode->value = value;
    pthread_mutex_lock(&mutex);
(*) newNode->succ = firstNode;
(*) firstNode = newNode;
    pthread_mutex_unlock(&mutex);
}
```

2.2.3.2. Condition Variables – Motivation

- Thread might need to wait for an event or a certain condition before proceeding
  - Thread can be put to sleep while waiting (with others continuing their work)
  - Thread should be notified once the condition is fulfilled

- Examples:
  - Waiting for a computation being started by the user
  - Waiting for data being added to a linked list
  - Consumer thread waiting for data input prepared by a producer thread

First Idea: Busy Waiting (Spinning)

- Goal: Waiting for an event using only mutexes
- Idea: Check repeatedly to see if a condition is true

```
void* waitingThread(void* thread_arg) {
    int localFlag = 0;
    while(!localFlag) {
    pthread_mutex_lock(&threadFlagMutex);
    localFlag = threadFlag;
    pthread_mutex_unlock(&threadFlagMutex);
    }
    // Condition fulfilled – Do work!
}
```

```
void setFlag(int newFlagValue) {
    pthread_mutex_lock(&threadFlagMutex);
    threadFlag = newFlagValue;
    pthread_mutex_unlock(&threadFlagMutex);
}
```

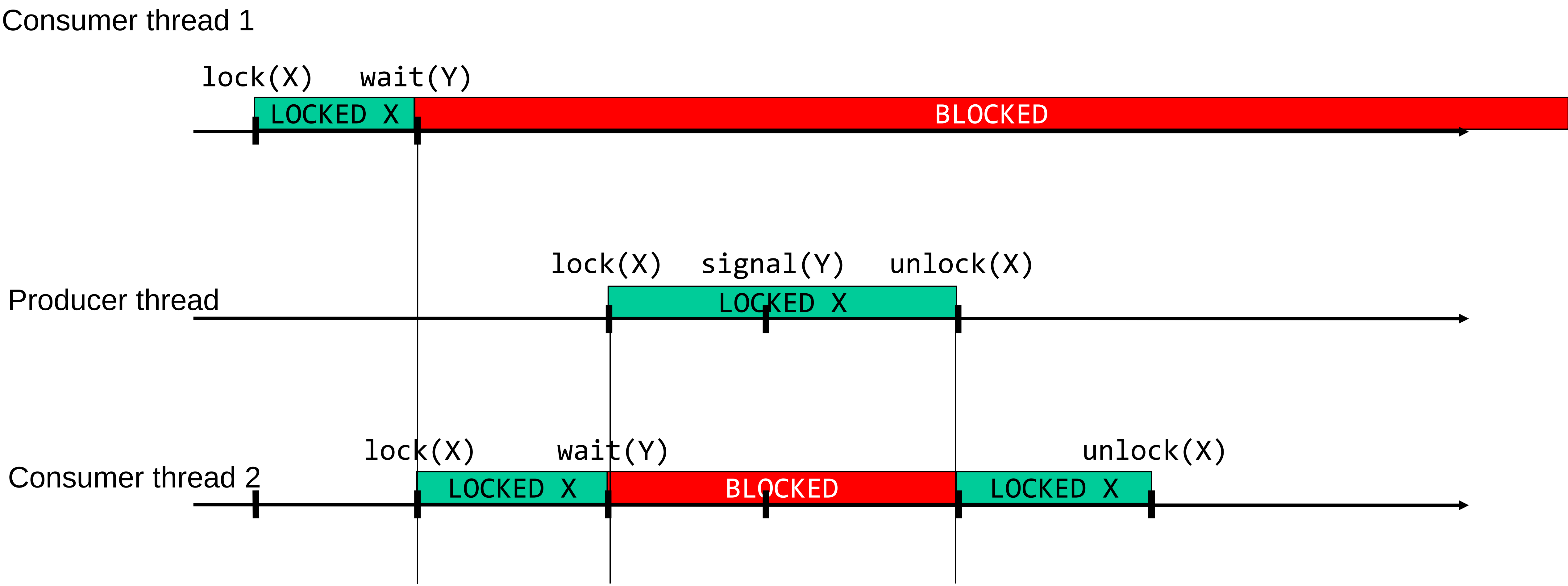
Main disadvantage:
Consumes processor time without performing any work!

Condition Variables – Basic Concept

- A thread can wait on a condition variable
- A waiting thread is blocked until another thread signals the same condition variable either
  - Waking a single thread or
  - Waking all waiting threads by broadcasting the signal

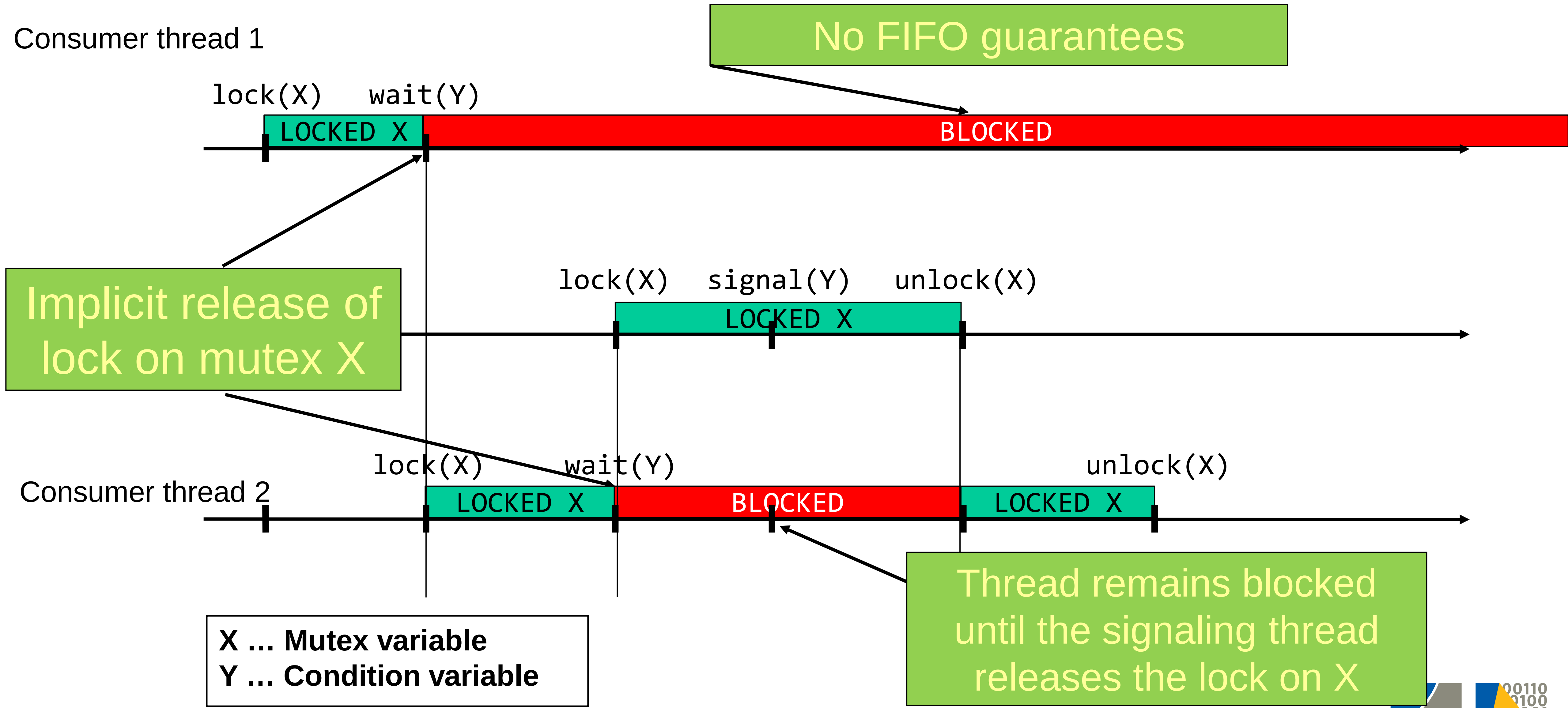
- **Condition variables** must always be **used together with a mutex**
  - Lock mutex before waiting on condition variable
  - Waiting on the condition variable implicitly unlocks the mutex
  - Lock on the mutex is reacquired automatically when the thread is unblocked
  - Acquire lock on mutex before signalling the condition variable

Condition Variables – Example



X … Mutex variable
Y … Condition variable

Condition Variables – Example



X … Mutex variable
Y … Condition variable

PThread Condition Variables

- Data structures
  - `pthread_cond_t` – Condition variable data type
  - `pthread_condattr_t` – Condition attributes data type

- Initializing a condition variable
  - Dynamically
    - `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
  - Statically at declaration time
    - `pthread_cond_t condVar = PTHREAD_COND_INITIALIZER;`

- Destroying a condition variable
  - `int pthread_cond_destroy(pthread_cond_t *cond);`

PThread Condition Variable Attributes

- Allows to configure the properties of a new condition variable

- Initializing
  - `int pthread_condattr_init(pthread_condattr_t *attr);`
  - Initializes `attr` with the default value for all attributes

- Destroying
  - `int pthread_condattr_destroy(pthread_condattr_t *attr);`

- There is usually no need to set any condition variable attributes to special values

UNIVERSITÄT BONN

PThread Waiting On a Condition Variable

- `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - `cond` – Condition variable to wait on
  - mutex – Mutex variable associated with `cond`
  - Precondition: `mutex` must be locked by the calling thread
  - Effect: Atomically releases `mutex` and blocks the calling thread on the condition variable `cond` until `cond` "is signalled"
  - Postcondition: `mutex` is again locked by the calling thread

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond, &mutex);
// Signal has been received
// Do actual work here...
pthread_mutex_unlock(&mutex);
// ... or here
```

PThread Condition Variable Timed Wait (1)

- `int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
  - `abstime` – Absolute time value limiting how long the calling thread can remain blocked waiting for the signal

- Behavior identical to `pthread_cond_wait`, except
  - Wait time for the signal is limited
  - Returns `ETIMEDOUT` if `abstime` is passed without `cond` being signaled

- Note: `pthread_cond_timedwait` expects an **absolute time value** not the length of a time interval!

PThread Condition Variable Timed Wait (2)

- Example:

```
int result;
struct timespec tspec;
struct timeval tval;

pthread_mutex_lock(&mutex);
result = gettimeofday(&tval, NULL);
tspec.tv_sec = tval.tv_sec;
tspec.tv_nsec = tval.tv_usec * 1000;
tspec.tv_sec += 5;
result = pthread_cond_timedwait(&cond, &mutex, &tspec);
if (result == ETIMEDOUT) {
    printf("Timeout!\n");
} else {
    printf("Signal received!\n");
}
pthread_mutex_unlock(&mutex);
```

- Waits for a maximum of 5 seconds
- But: Can remain blocked longer as it needs to reacquire the lock on the mutex.

PThread Signaling on a Condition Variable

- `int pthread_cond_signal(pthread_cond_t *cond);`
  - Effect: Unblocks at least one of the threads waiting for `cond` (if there are threads waiting)

- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Effect: Unblocks all threads waiting for `cond`

- For predictable scheduling behavior:
  - Lock `mutex` variable `mutex` before signalling on a condition variable and release lock on `mutex` afterwards.
  - Signaled threads remain blocked until lock on `mutex` is released again.
  - Only one of the signaled threads at a time gets the lock on `mutex`.

UNIVERSITÄT BONN

PThread Condition Variables – Waking Threads (1)

- Problem
  - `pthread_cond_signal` unblocks **AT LEAST** one thread
  - Specification allows “spurious wakeups” from `pthread_cond_wait` and `pthread_cond_timedwait`
  - Returning from `pthread_cond_wait` or `pthread_cond_timedwait` **DOES NOT IMPLY ANYTHING** about the value of the predicate

- Solution
  - Always reevaluate the predicate upon returning from a wait or a timed wait

PThread Condition Variables – Waking Threads (2)

- Example

```
pthread_mutex_lock(&threadFlagMutex);
while(!threadFlag) {
    pthread_cond_wait(&threadFlagCondition, &threadFlagMutex);
}
pthread_mutex_unlock(&threadFlagMutex);
// Condition fulfilled - Do work!
```

- Remark: Also covers the case where an event is signaled before the thread waits on the condition variable

- Works similarly for `pthread_cond_timedwait`
    - Using absolute time values is actually helpful here!
    - Idea: Only repeat if “`result != ETIMEDOUT`”

2.2.3.3. Semaphores – Motivation

- Some applications need to limit access to sections **based on a counter**, for example:
  - Limited number of N resources available –> Only proceed if not all of them are in use.
  - Count the number of entries in a queue –> Only proceed if the queue is not empty.

- Goal: **Implement a shared counter** that controls the access to critical sections

- Generalization of the mutex approach which can be seen as a binary semaphore

Semaphores – Basic Concept

- Use a **counter to synchronize threads**
  - Value > 0: Thread is allowed to proceed
  - Value = 0: Thread is blocked

- Wait operation
  - If counter is zero, then block the calling thread
    (the block will be released when the semaphore value is >0)
  - Counter value is decremented by 1
  - wait returns

- Post operation
  - Increment counter value by 1
  - Unblock one of the waiting threads (if any) if the counter was zero before

Semaphores – Application Examples

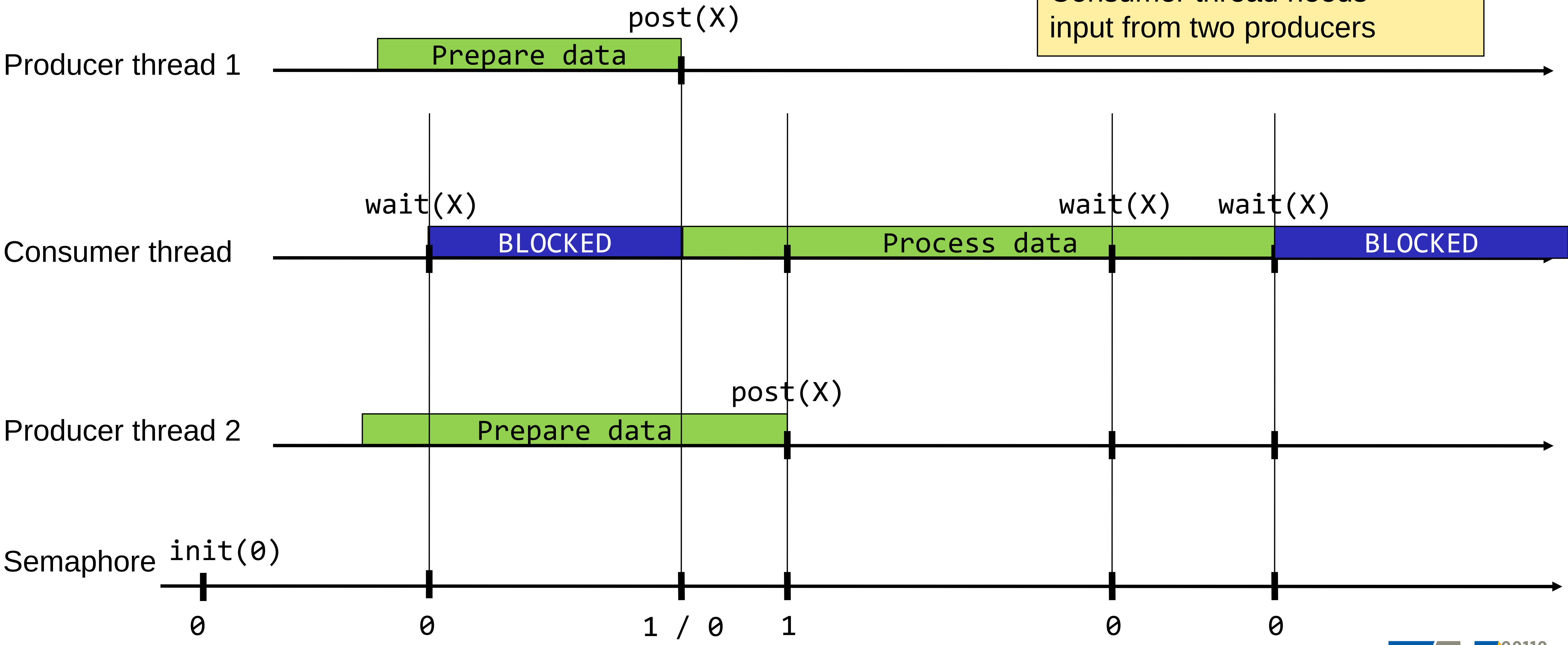
- **Mutual exclusion = Mutex cf. 2.2.3.1.**
  - Initialize counter with 1
  - Process entering critical section calls wait operation
  - Process leaving critical section calls post operation

- Variants of the mutual exclusion problem
  - E.g., no more than X threads are allowed to enter a critical section at a time.

- Consumer waiting for input (see next slide)
- …

Semaphores – Example

Example application:
Consumer thread needs
input from two producers



PThread Semaphores

- Requires: `#include <semaphore.h>`
- Data structures
    - `sem_t` – Semaphore data type

- Initializing a semaphore
    - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
    - `sem` – The semaphore
    - `pshared` – Indicates whether the semaphore is shared among processes
    - `value` – The initial value of the semaphore

- Destroying a semaphore
    - `int sem_destroy(sem_t *sem);`

PThread Operating on Semaphores

- Waiting on a semaphore
  - `int sem_wait (sem_t *sem);`
    - Blocks the thread until the lock on the semaphore can be acquired
  - `int sem_trywait (sem_t *sem);`
    - Never blocks the thread but indicates the success of the locking attempt in its return value
    - `0` – Semaphore has been locked successfully.
    - `EAGAIN` – Semaphore has not been locked, because it was already locked.

- Releasing a lock on a semaphore
  - `int sem_post(sem_t *sem);`

PThread Semaphores Example

- Using a semaphore to implement mutex functionality

```
void* workThread(void* thread_arg) {
    sem_wait(&workThreadSem);
    // Critical section
    sem_post(&workThreadSem);
}

void init() {
    sem_init(&workThreadSem, 0, 1);
    [...]
}
```

  - Counter of the semaphore is always zero while a thread is within the critical section.

–> No other thread is able to enter until the thread calls `sem_post` which sets the semaphore counter back to one.

Semaphores

Brinch Hansen (1973): “The semaphore is an elegant synchronizing tool for an ideal programmer who never makes mistakes. But unfortunately of using semaphores incorrectly can be quite serious“

2.2.3.4. Barriers – Motivation

- Threads need to "meet up" at certain points of the program execution
    - Only when all threads have arrived at this point, the execution of the individual threads can continue.
    - Threads "wait for each other"

- Example applications:
    - Threads operating on different parts of a large problem synchronizing their (intermediate) results

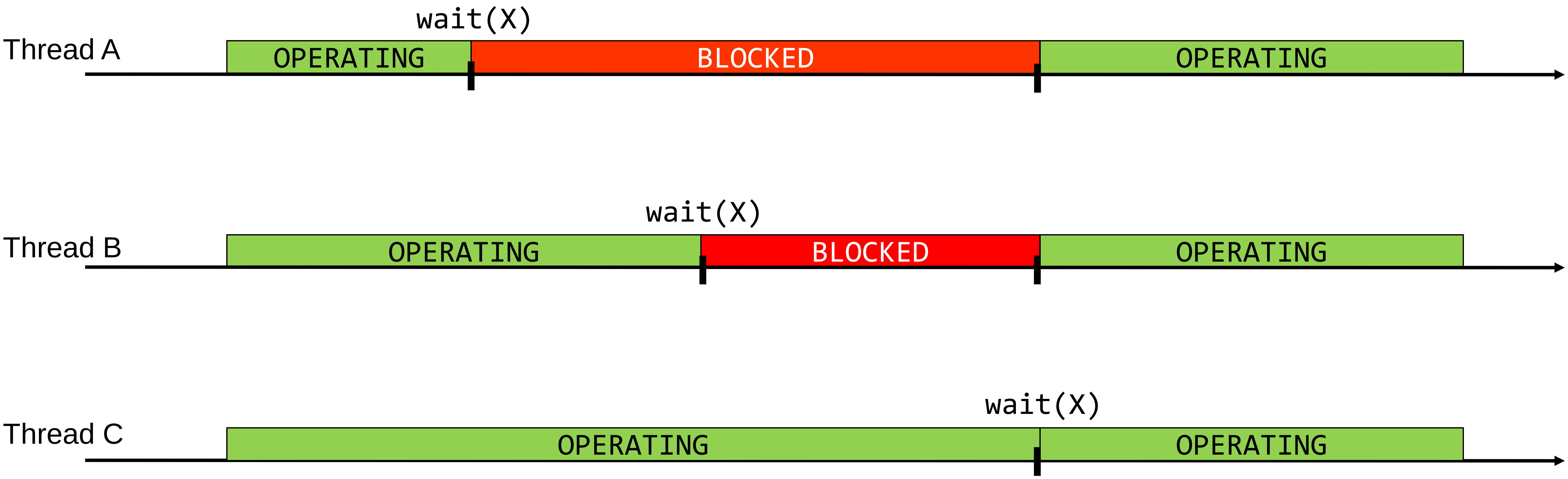
Barriers – Basic Concept

- Barrier is initialized with the number of threads participating nBarr

- Barrier manages a counter to record the number of threads already waiting at the barrier

- Wait operation
  - Increment counter: `counter++`
  - If (`counter < nBarr`) then block calling thread
  - If (`counter == nBarr`) then unblock all threads and re-initialise counter with 0

Barriers – Example



`init` n_{barr} `= 3`

PThread Barriers (1)

- Data structures
  - `pthread_barrier_t` – Barrier data type
  - `pthread_barrierattr_t` – Barrier attributes data type

- Initializing a barrier
  - `int pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count);`
  - count – The **number of threads** that must call `pthread_barrier_wait` for the threads to become unlocked

- Destroying a barrier
  - `int pthread_barrier_destroy(pthread_barrier_t *barrier);`

PThread Barriers (2)

- Waiting on the barrier
  - `int pthread_barrier_wait(pthread_barrier_t *);`
- Example

thread(s) waiting here

init with 5

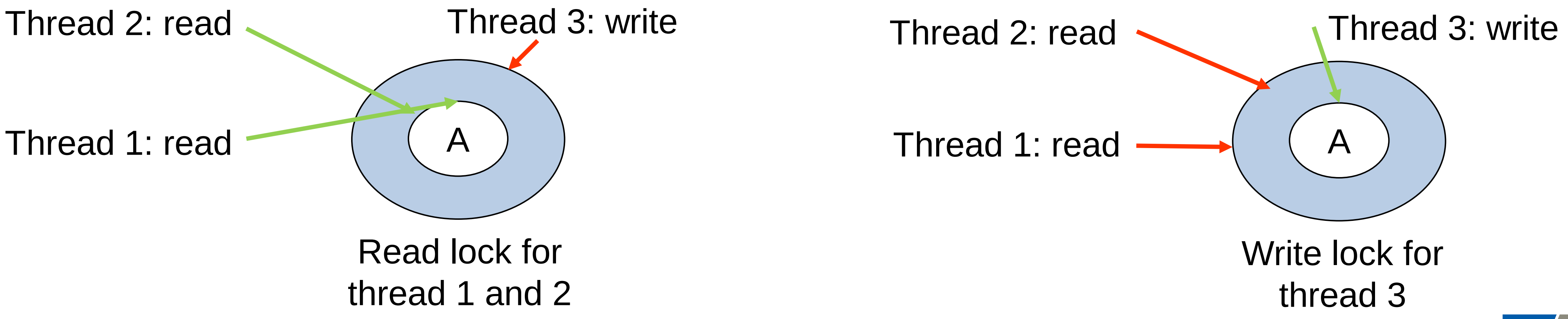
```
void* workThread(void* thread_arg) {
    // Do a lot of work here...
    pthread_barrier_wait(&allThreadsFinishedBarrier);
    printf("All threads finished!\n");
}

void init() {
    int i;
    pthread_barrier_init(&allThreadsFinishedBarrier, NULL, 5)
    for (i=0; i<5; i++) {
    pthread_create(&threadID[i], NULL, workThreadFunction, NULL);
    }
    [...]
}
```


UNIVERSITÄT BONN

2.2.3.5. Read-Write Locks

- Motivation:
  - Reading data concurrently from multiple threads is not harmful.
  - Only **write operations can interfere** with concurrent read or write operations

- **Idea:** Treat locking requests for reading shared data differently than locking requests for writing data



Read-Write Locks – Basic Concept

- Servicing a request for a read lock on X
  - if no other thread holds a lock on X
  - if other threads only hold read locks on X
  - Otherwise: Requesting thread is blocked

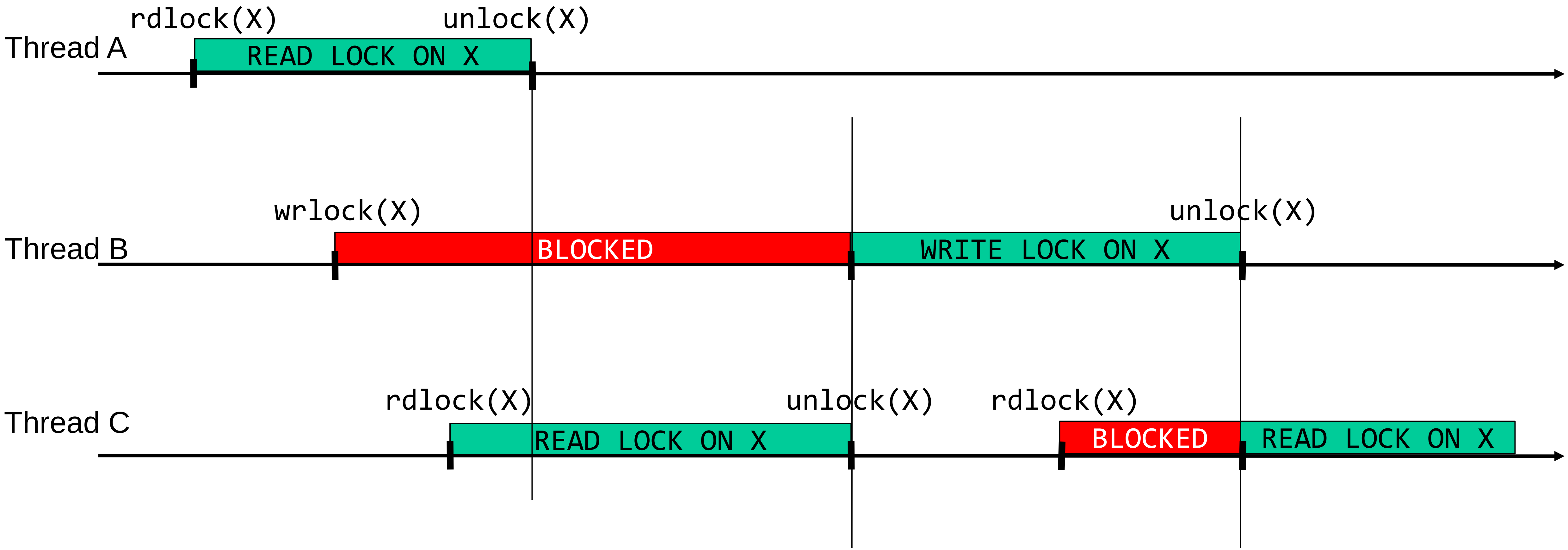
- Servicing a request for a write lock on X
  - only if no other thread holds a lock (of any kind) on X
  - Otherwise: Requesting thread is blocked

Lock set

		Null	Read	Write
Lock requested	Read	+	+	-
	Write	+	-	-

\+ request will acquire lock
\- request will block thread

Read-Write Lock – Example



PThread Read-Write Lock

- Data structures
  - `pthread_rwlock_t` – Read-write lock data type
  - `pthread_rwlockattr_t` – Read-write lock attributes data type

- Initializing a read-write lock
  - Dynamically
    - `int pthread_rwlock_init (pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);`
  - Statically at declaration time
    - `pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;`

- Destroying a read-write lock
  - `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);`

PThread Read-Write Lock – Locking and Unlocking

- Acquiring a read lock
  - `int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);`
  - `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`

- Acquiring a write lock
  - `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
  - `int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);`

- Releasing a lock
  - `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`
  - No different functions required for releasing read and write locks!

Waiting for Threads

- PThreads:
  - `pthread_join` for joinable threads
    - Missing join produces zombies
  - Manual management for detached threads
    - Global variables
    - Counters
    - …

Thread communication

- PThread
  - Return codes
  - Pass by reference
  - Global variables

- Problem:
  - Requires synchronization

Share by communication

- Communicate results instead of using shared memory
    - Used in functional programming
    - Unless it‘s easier to solve with Mutexes
    - In most cases channels are easier than sync methods
    - Very useful in concurrent network development

- Shared data-structures still require traditional synchronization methods
    - Dynamic Lists
    - Arrays
    - Hash-maps

Channels

- Channels for communication between routines
  - Read on one end
  - Write on an other end
  - Block on full buffer
  - Thread-safe

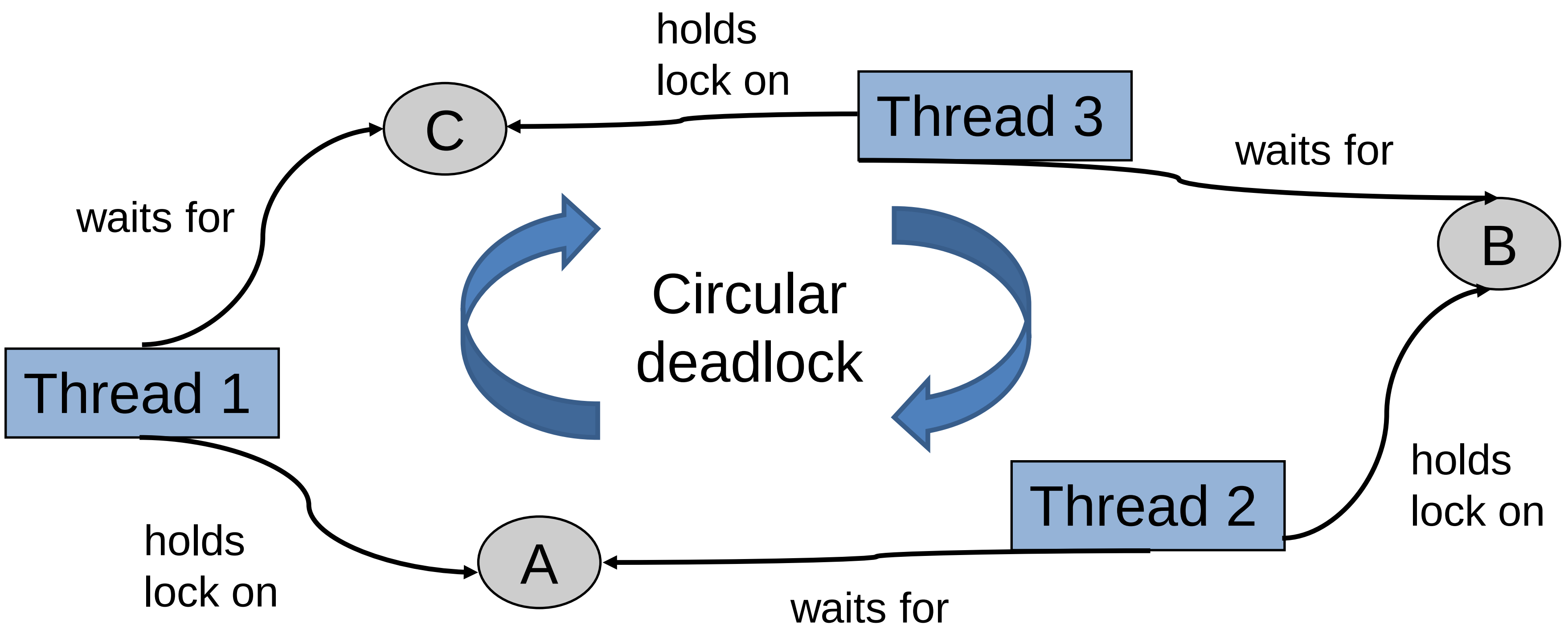
src: http://golangtutorials.blogspot.de/

Overview

- Fundamentals
- Threads in Linux
- Thread Synchronization
- **2.2.4. Deadlocks**
- Important Threading Mechanisms

2.2.4. Deadlocks

- A **deadlock** is a situation when two or more threads (or processes) are waiting for the other to finish some action (e.g., accessing a resource, signaling an event) thereby blocking each others progress.



Deadlock Conditions

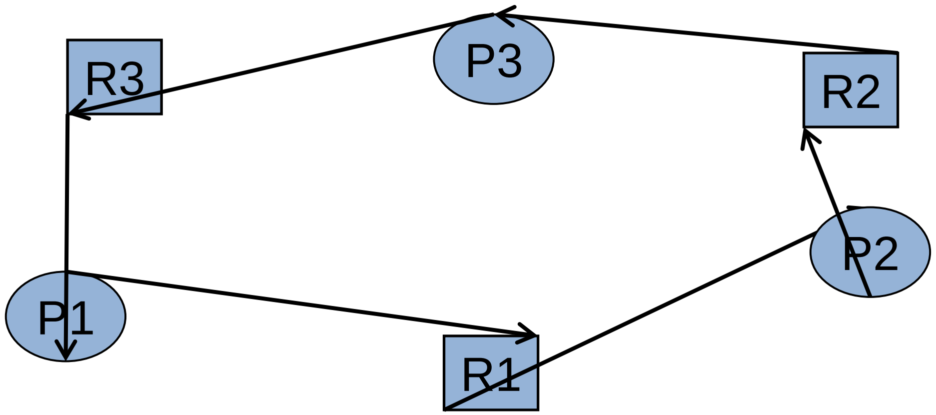
1. **Mutual exclusion:** Resources are exclusive – they cannot be accessed by more than one thread at a time.
2. **Non preemption:** Resources can only be released voluntarily by the holding thread and cannot be forcibly removed.
3. **Hold & wait:** Threads can hold resources while waiting for other resources.
4. **Circular wait:** There exists a closed loop of threads in which each thread waits for a resource held by its successor in the loop.

All four conditions **must be fulfilled for a deadlock to occur**!

Wait-For Graphs (1)

Resource-Allocation Graphs

- Two types of **nodes:**
  - Processes or threads are represented by **circles**
  - Resources are represented by **rectangles**
- **Edges** only possible between different types of nodes
  - Process –> Resource: Process is waiting for resource
  - Resource –> Process: Resource is held by process
- Example:



Cycle in the graph –> Deadlock

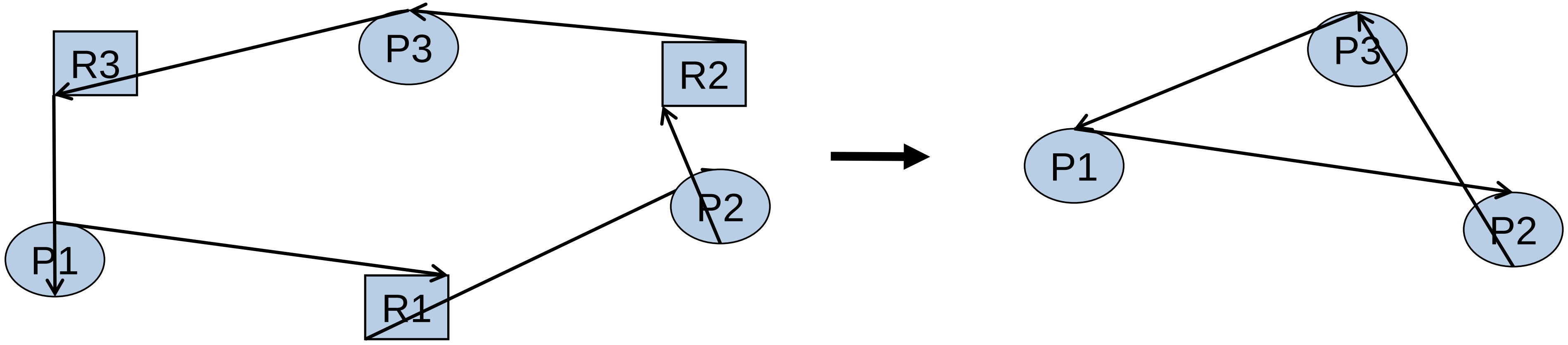
Wait-For Graphs (2)

Process Wait-For Graphs

- Nodes are processes
- Edge P1 –> P2: Process P1 waits for Process P2
- Example:

Cycle in the graph → Deadlock

- Resource-allocation graphs are **easily converted** to process wait-for graphs:



Handling Deadlocks (1)

- **Deadlock prevention** **(more also on slide 105)**
  - Idea: Ensure that one of the necessary deadlock conditions cannot hold
  - Achieve this by constraining how processes / threads can request resources

- **Deadlock avoidance**
  - Supervisor (e.g., the OS) needs information on which resources processes / threads are going to request
  - Decides which resource requests can be satisfied and which have to be delayed

Handling Deadlocks (2)

- **Deadlock detection**
  - Assumption: Deadlock situations can occur
  - Strategy: (Periodically) examine the state of the system to detect deadlock situations
  - Recover from deadlocks (e.g., by terminating some processes / threads)

- **Suspecting deadlocks**
  - Like deadlock detection but already act when a deadlock is suspected
  - Allows for less intricate detection mechanisms

Deadlock Prevention

- Some approaches:
  - Allow threads to lock only one resource at a time → Resource must be released before another resource is requested (**Hold & wait condition**)
  - Force threads to release all held resources if they request another resource that cannot be served immediately (**No preemption condition**)
  - Impose a total ordering of resources → Resources can only be requested in this order (**Circular wait condition**)

- Applicability depends on the system properties

- Deadlock prevention can be expensive!

Deadlocks – Relevance for the Programmer

- Thread synchronization can provide for the first three deadlock conditions
  - Danger of deadlocks is a real problem!
  - Not using thread synchronization is usually not an option.

- Multithreaded applications are prime candidates for the deadlock problem as multiple threads compete for resources

- Programmer needs to be aware of the deadlock problem
  - Careful programming can avoid many potential deadlock situations
  - Deadlock prevention is most effectively and efficiently implemented with application knowledge

Overview

- Fundamentals
- Threads in Linux
- Thread Synchronization
- Deadlocks
- **2.2.5. Important Threading Mechanisms**

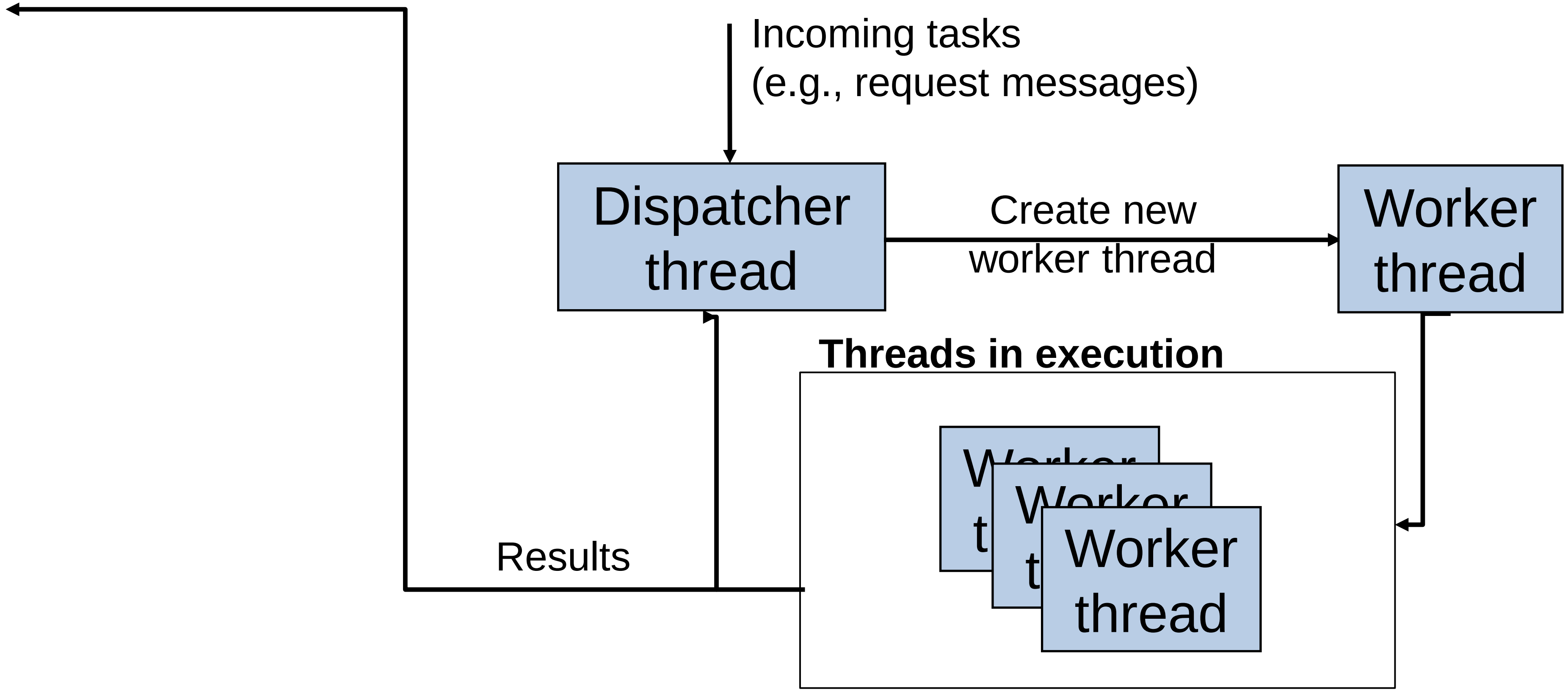
2.2.5. … Worker Threads (1)

- Motivation
  - Programs need to execute long running tasks
  - The application should not block while tasks are executed
    - The GUI of interactive applications should remain responsive
    - More tasks might come in while one task is executed

- Approach
  - Dispatcher thread receives all incoming tasks
  - Dispatcher creates a separate worker thread for each long running task
  - If required: Finished worker threads report their results back to the dispatcher (which might forward the result)
  - Worker threads terminate after completing their task

Worker Threads (2)

- General Model:



Worker Threads (3)

- Advantages
  - Simple mechanism
  - Provides for parallelism in the execution of tasks

- Thread synchronization requirements
  - Worker threads accessing shared data
  - Worker threads returning result data to the dispatcher or the original requester
  - Expensive if many tasks are used
  - Increased complexity

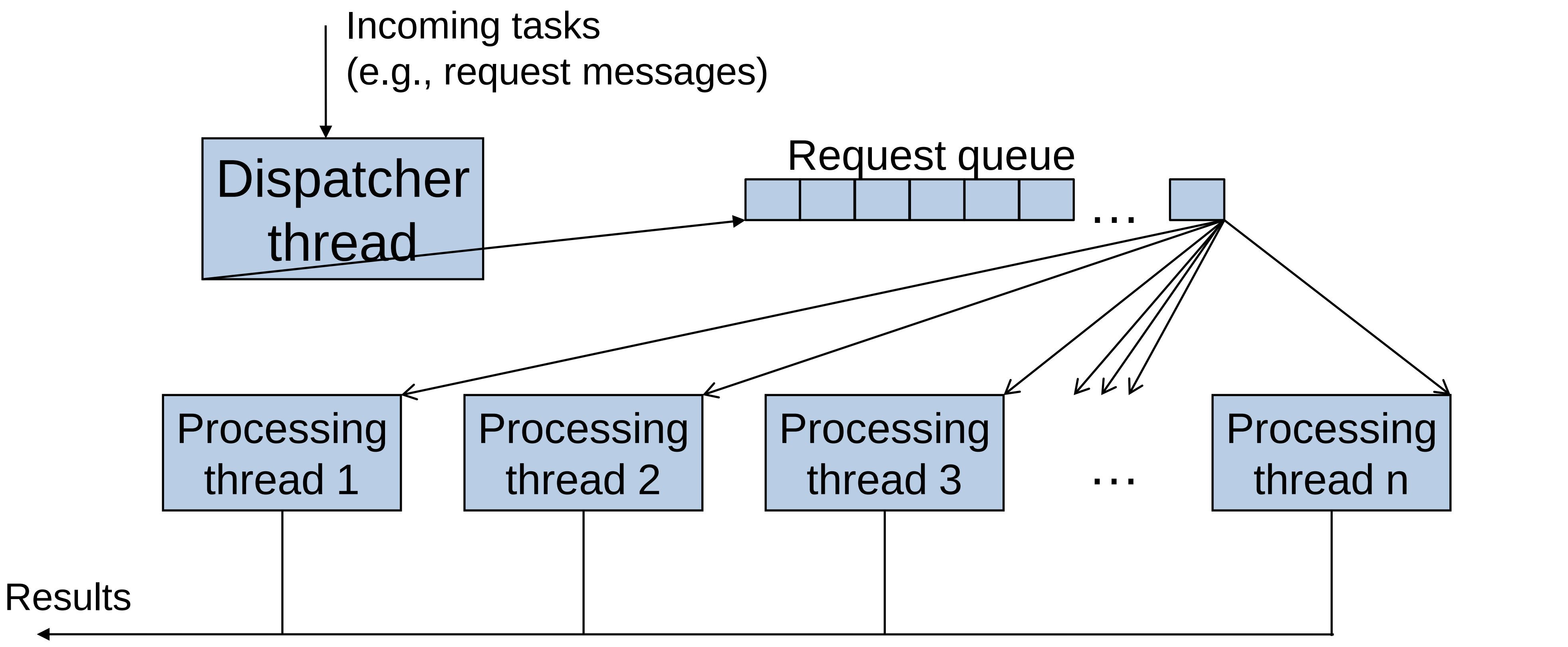
Thread Pools (1)

- Motivation
  - Many applications use threads to perform recurring tasks (e.g., to process incoming messages). Most of them are short-lived.
  - Tasks are independent of each other.
  - Creating threads on-the-fly for each task can be costly.

- Approach
  - Create a fixed number of processing threads (the thread pool) at startup time.
  - Add new tasks to a task queue and notify threads waiting for work.
  - Processing threads becoming available check the queue for tasks and wait if currently no task is available.

Thread Pools (2)

- General model



Thread Pools (3)

- Advantages
  - Efficient use of multithreading

- Thread synchronization requirements
  - Accessing the request queue for adding or removing tasks
  - Processing threads accessing shared data
  - Processing threads returning result data to the dispatcher or the original requester

- Variants
  - Dynamically adjust the number of threads in the pool based on the workload in the system.
  - Use a priority queue to handle tasks waiting for a thread.

Thread-Safe Libraries

- Code or code libraries are thread safe if they function correctly when executed in parallel by multiple threads
  - Multiple threads might call the same functions
  - Multiple threads might call functions accessing shared data
- Example: Function for printing data to a screen (e.g., `printf` !!!)
  - Data might be buffered before output
  - What if two threads are accessing the buffer in parallel?
- Documentation of functions or code libraries should provide information on whether thread safety is provided or not!
- In some cases, special reentrant versions of functions or libraries are available

Achieving Thread Safety

- Ensure **re-entrance of functions**
  - Only use purely local state – no access of shared data
  - Partial execution of the function only affects the stack of the thread
- Use **mutual exclusion**
  - Serialize access to critical sections using thread synchronization primitives
- Use **thread-local data**
  - Create copies of data for each thread and operate only on this thread-local data
- Use **atomic operations for accessing shared data**
- **Share memory by communication**

Thread-Safe Libraries – Pros and Cons

- **Importance** of thread safety
  - Library function calls are used as black boxes within threads
  - It might not be obvious, which functions of libraries a thread uses
  - Interrelation of library functions often unclear

- **Why** is **not** every library made **thread-safe**?
  - Programming overhead
  - Performance overhead of using thread synchronization mechanisms
  - Thread-safety not relevant in many scenarios
  - Programmer using a library has better overview on the concurrency issues in the application (but does not necessarily know the inner workings of the affected library!).

Summary

- Threads are a powerful mechanism for parallel programming
  - More lightweight than processes
  - Allow for efficient communication and cooperation

- Standardized support for thread programming in C is available using the PThreads library

- Synchronizing the execution of threads is a big challenge for parallel programming
  - Many thread synchronization mechanisms
  - Care must be taken to avoid deadlocks
  - Share memory by communicating (if possible)

Summary (2)

- What did we learn in this section?
    - how to create and handle threads (2.2.1. / 2.2.2.)
    - how threads communicate with each other: (2.2.2.)
        - parameters with creating a new thread
        - return of results from a terminating thread
        - shared variables, data structures, …
- how to synchronize threads in critical sections (2.2.3.x)
    - mutex, semaphore (2.2.3.1. + 2.2.3.3.)
    - condition variable (2.2.3.2.)
    - read/write locks (2.2.3.5.)
    - barriers (2.2.3.4.)
    - channels (2.2.3.5.)
- problem of deadlocks (2.2.4.)
- prominent thread programming models (2.2.5.)