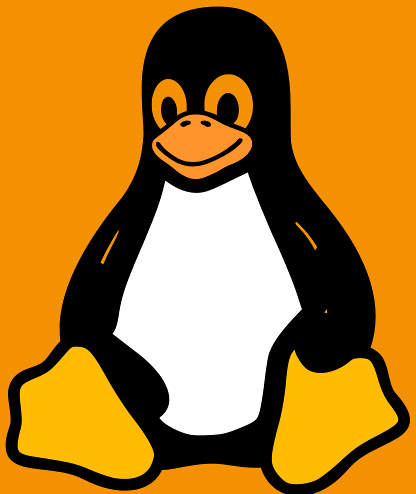
Prof. Dr. Florian Künzner

Technical University of Applied Sciences Rosenheim, Computer Science



source: iconspng.com

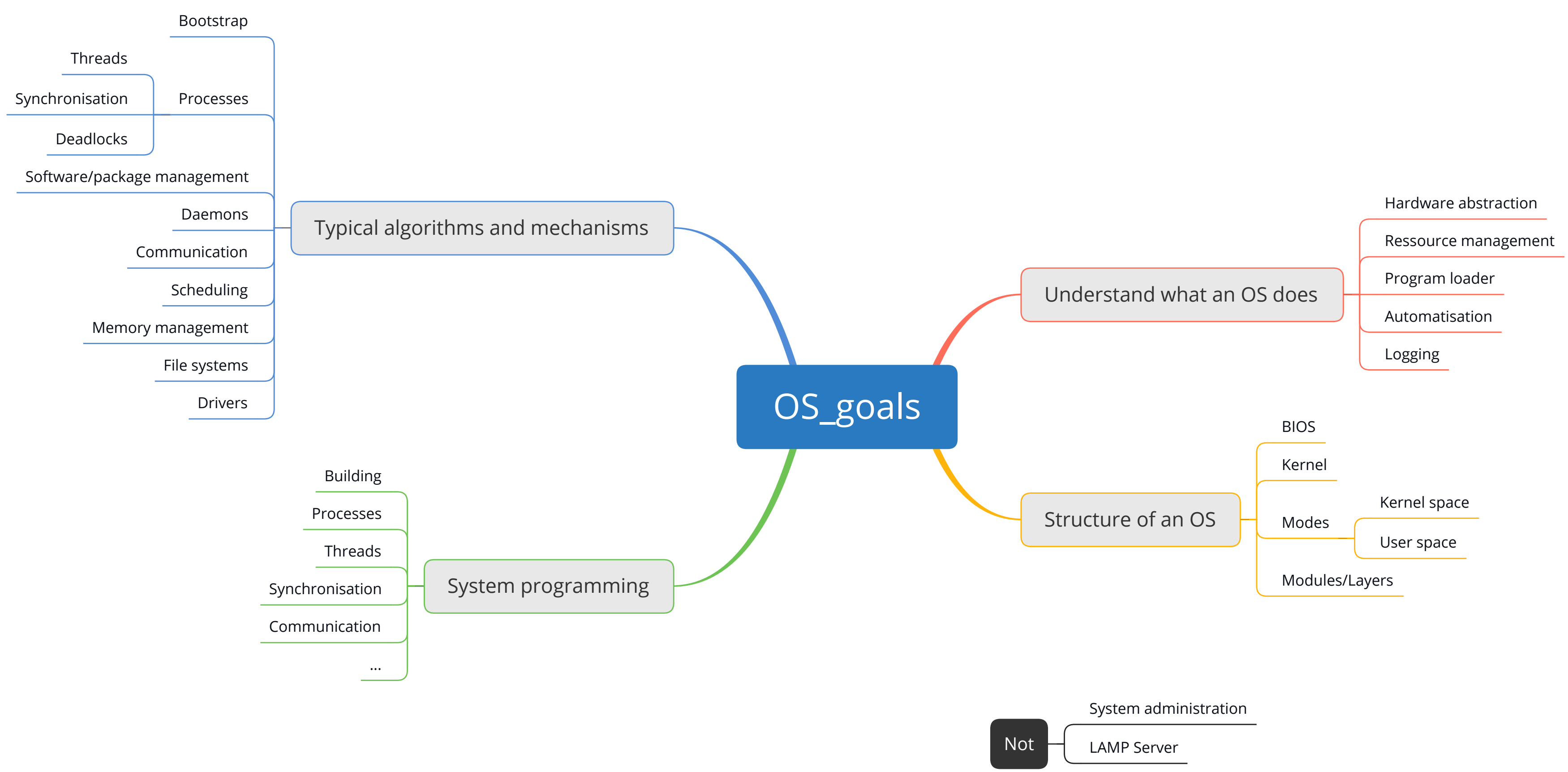
OS 7 – Synchronisation 2

The lecture is based on the work and the documents of Prof. Dr. Ludwig Frank

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Goal



CAMPUS Rosenheim
Computer Science

Goal

OS::Synchronisation

- Producer-consumer problem
- Reader-writer problem
- Monitor concept

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Intro

Standard problems...

- Mutual exclusion (last week – check!)
- Producer-consumer problem
- Reader-writer problem

CAMPUS Rosenheim
Computer Science

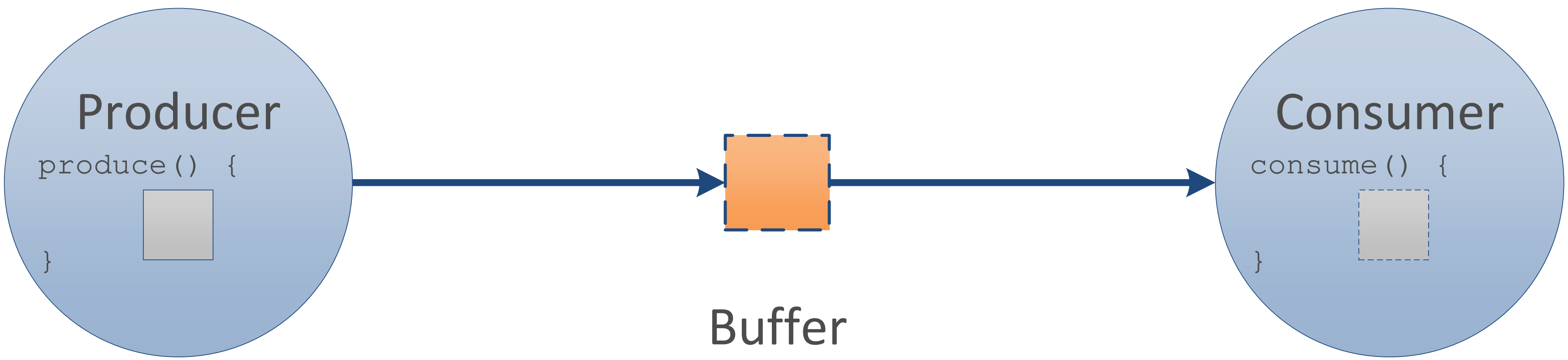
Producer-consumer

Producer-consumer problem

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Producer-consumer (1): illustration



CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Producer-consumer (1): facts

- **One or more** processes "**produce**" something
- **One or more** processes "**consume**" something
- There is a **buffer** with **one place** to **store** the produced "**artefact**"
- **Producer**
  - If it delivers an "artefact" it **can immediately produce** the **next**
  - If the buffer is full, the **producer waits until the buffer is free**
- **Consumer**
  - If it has consumed an "artefact" it **can immediately fetch** the **next**
  - If the buffer is empty, the **consumer waits until the buffer is full**

CAMPUS Rosenheim
Computer Science

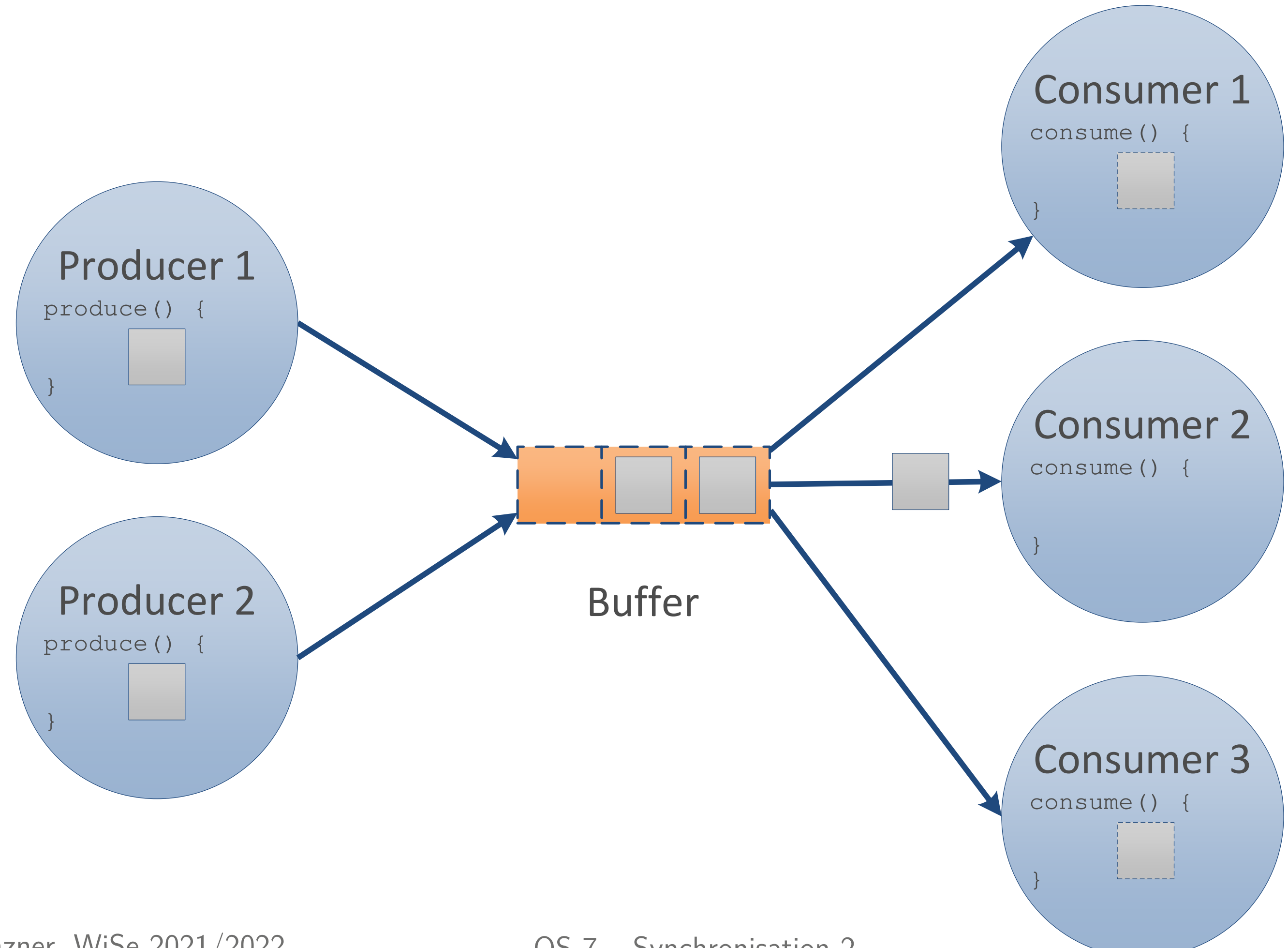
Technische Hochschule Rosenheim
Technical University of Applied Sciences

Producer-consumer (1): pseudo C code

```
int buffer = 0;
seminit(buffer_free, 1);
seminit(buffer_full, 0);

void producer() {
    while(1) {
        int artefact = produce();

        P(buffer_free);
          buffer = artefact;
        V(buffer_full);


    }
}

void consumer() {
    while(1) {


        P(buffer_full);
          int artefact = buffer;
        V(buffer_free);

        consume(artefact);
    }
}

int main() {
    //start threads...
}
```

CAMPUS Rosenheim
Computer Science

Producer-consumer (2): illustration



CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Producer-consumer (2): facts

- **One or more** processes "**produce**" something
- **One or more** processes "**consume**" something
- There is a **buffer** with **N places** to **store** the produced "**artefacts**"
- **Producer**
  - If it delivers an "artefact" it **can immediately produce** the **next**
  - If the buffer is full, the **producer waits until the buffer has a free place**
- **Consumer**
  - If it has consumed an "artefact" it **can immediately fetch** the **next**
  - If the buffer is empty, the **consumer waits until the buffer contains at least one artefact**

Producer-consumer (2): pseudo C code

```c
const unsigned int N = 3;
int buffer[N];
seminit(buffer_free, N);
seminit(buffer_full, 0);
seminit(buffer_critical, 1); //binary semaphore
```

```c
void producer() {
    while(1) {
        int artefact = produce();

        P(buffer_free);

        P(buffer_critical);
          store_in_buffer(artefact);
        V(buffer_critical);

        V(buffer_full);


    }
}
```

```c
void consumer() {
    while(1) {


        P(buffer_full);

        P(buffer_critical);
          int artefact = fetch_from_buffer();
        V(buffer_critical);

        V(buffer_free);

        consume(artefact);
    }
}
```

CAMPUS Rosenheim
Computer Science

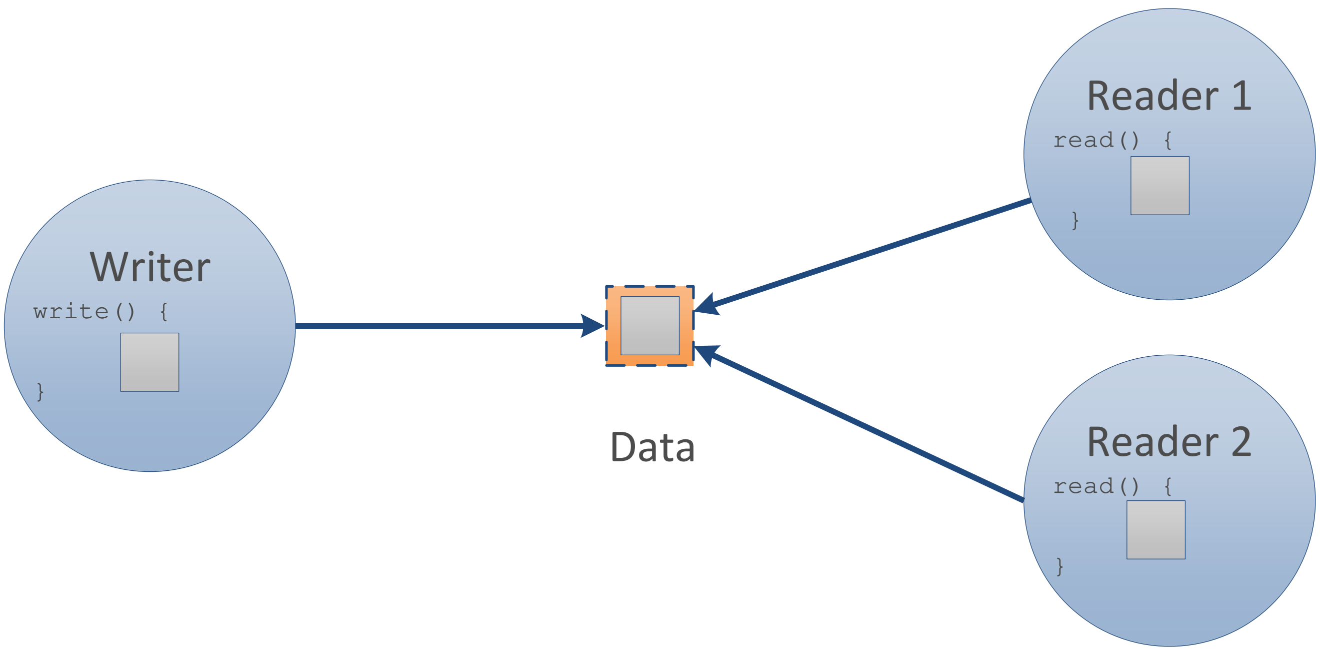
Reader-writer

Reader-writer problem

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Reader-writer: illustration



CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Reader-writer: facts

- **One or more** writers "**write**" something
- **One or more** readers "**read**" something
- There is a **shared area** for the data.
- **Writer**
  - After the data is written, a writer **can immediately collect** the **next** set of data
  - If **no readers** currently **read**, it **can write** the new set of data
  - If **readers** currently **read**, it **waits until all readers have finished**
- **Reader**
  - After the data is fully read, it can **work independently** with the data
  - If a writer is currently writing the readers have to **wait until the writer has fully provided the data**
  - It is not a consuming read, the **data stay in the shared data area**

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Reader-writer: pseudo C code

```c
int shared_data = 0;
int num_active_readers = 0;
seminit(data_access, 1);
seminit(readers, 1);
```

```c
void writer() {
  while(1) {
    int data = produce();
    P(data_access);
      write_data(data);
    V(data_access);
  }
}
```

```c
void reader() {
  while(1) {
    P(readers);
      ++num_active_readers;
      if(num_active_readers == 1) {
        P(data_access);
      }
    V(readers);
    data = read_data();
    P(readers);
      --num_active_readers;
      if(num_active_readers == 0) {
        V(data_access);
      }
    V(readers);
    work_with(data);
  }
}
```

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

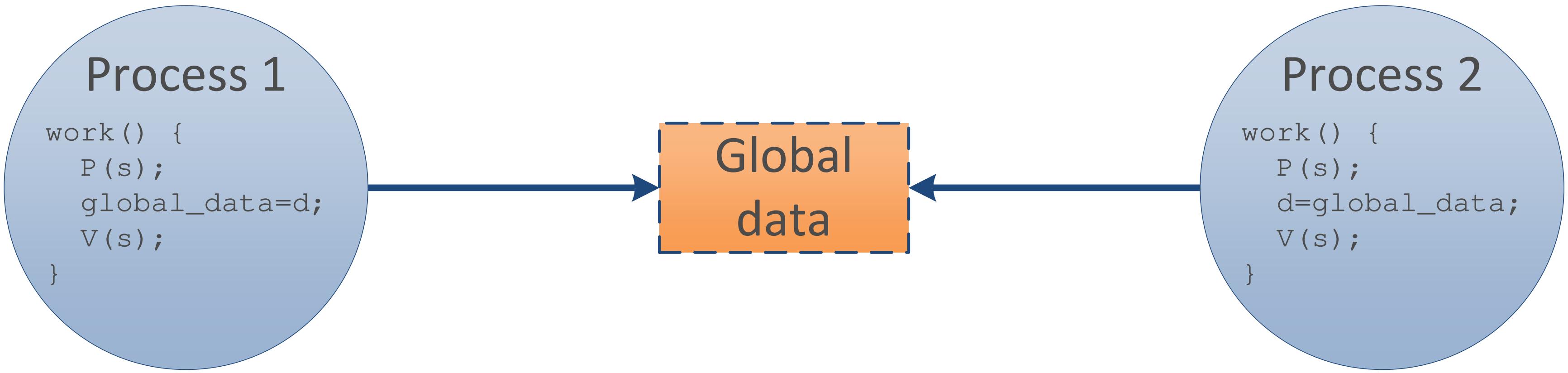
Monitor concept

Monitor concept, a short introduction.

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Problems with “raw” semaphores

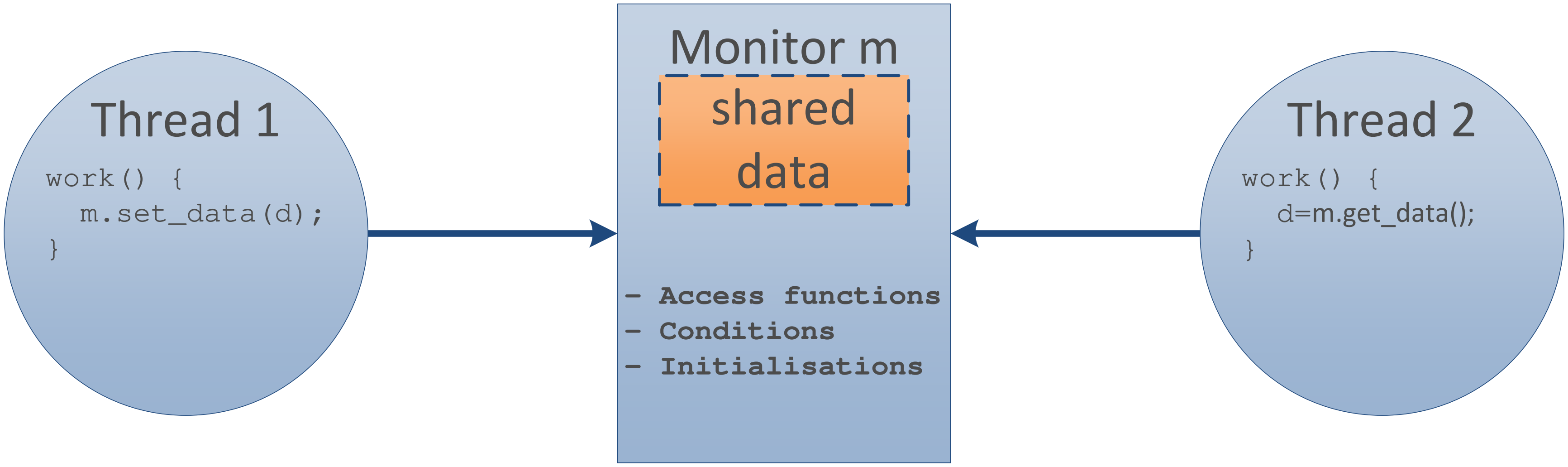


- Implementation is difficult
- Depends on the correctness of all processes/threads
- Verification of correctness is difficult
- Difficult to determine which access functions read or change shared data
- Data is independent of access functions

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

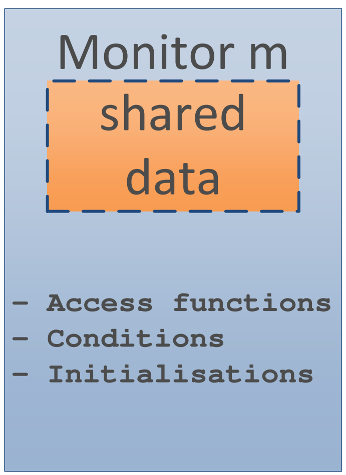
Monitor concept: illustration



CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Monitor concept: facts



- Contains data and access functions
- Does all the initialisation of data
- Checks the conditions internally
- Access to the shared data is only possible via the access functions
- **Only** one **“active“** process/thread **can be inside an access function**

Pro

- Less error prone: less todo for the users (processes/threads)
- Concentration of the difficult know-how inside the monitor

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Monitor concept (light): pseudo C code

Monitor module

```
void* buffer = NULL;
seminit(buffer_free, 1);
seminit(buffer_full, 0);

void set_data(void* data) {
  P(buffer_free);
    buffer = data;
  V(buffer_full);
}

void* get_data() {
  P(buffer_full);
    void* data = buffer;
    buffer = NULL;
  V(buffer_free);
  return data;
}
```

Main module

```
void producer() {
    while(1) {
        void* data = produce();
        set_data(data);
    }
}
void consumer() {
    while(1) {
        void* data = get_data();
        consume(data);
    }
}
```

- This is a lightweight monitor example
- Usually, condition variables and mutexes are used

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Monitor concept: mutex and condition

Idea A mutex controls the access functions of a monitor. The conditions helps to implement the waiting logic.

Operation	Description
`Mutex mutex`	Creates an instance of a mutex. A mutex is like a binary semaphore. The only difference is, that only the calling process/thread can unlock it.
`Condition cond`	Creates a condition variable. A condition variable is a synchronisation primitive that enables a process/thread to wait until a particular condition occurs.
`lock(mutex)`	Locks a mutex. The others wait.
`unlock(mutex)`	Unlocks a mutex.
`wait(cond, mutex)`	Waits until the condition is fulfilled. The mutex is free while waiting.
`signal(cond)`	Signals that the condition is fulfilled. Notifies one.

CAMPUS Rosenheim
Computer Science

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Monitor concept: pseudo C code

Monitor module

```
void* buffer = NULL;
Mutex mutex;
Condition buffer_free, buffer_full;
void set_data(void* data) {
  lock(mutex);
    if(buffer != NULL) { wait(buffer_free, mutex); }
    buffer = data;
    signal(buffer_full);
  unlock(mutex);
}
void* get_data() {
  lock(mutex);
    if(buffer == NULL) { wait(buffer_full, mutex); }
    void* data = buffer;
    buffer = NULL;
    signal(buffer_free);
  unlock(mutex);
  return data;
}
```

Main module

```
void producer() {
  while(1) {
    void* data = produce();
    set_data(data);
  }
}
void consumer() {
  while(1) {
    void* data = get_data();
    consume(data);
  }
}
```

CAMPUS Rosenheim
Computer Science

Summary and outlook

Summary

- Producer-consumer problem
- Reader-writer problem
- Monitor concept

Outlook

- Process communication
- Signals
- Sockets
- Pipes