Microprocessors

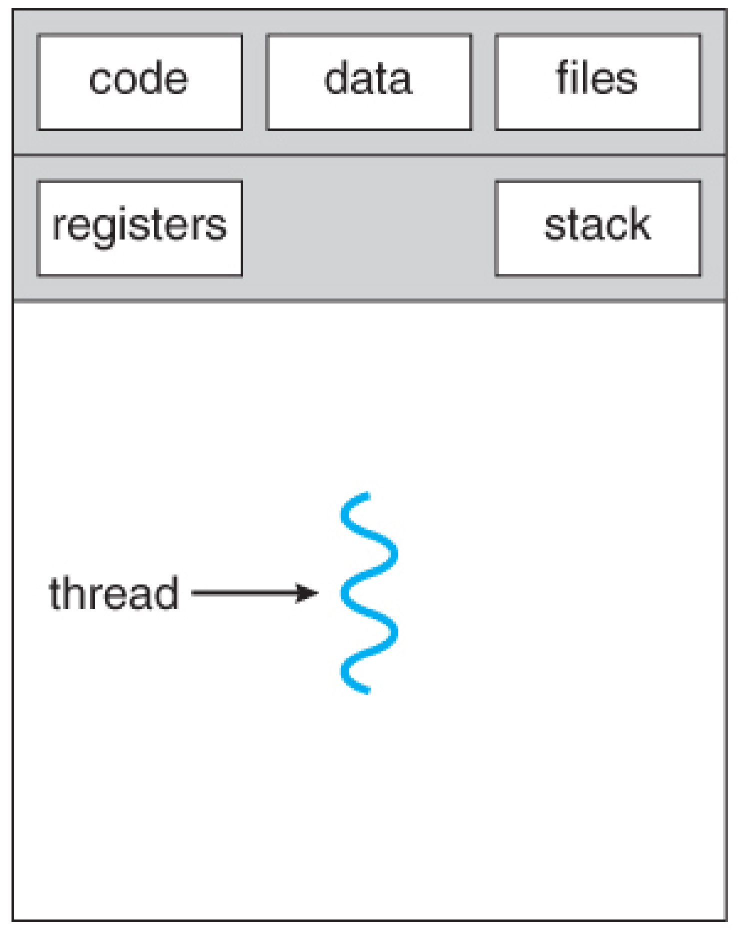
Multiprocessing, threads

Stanislav Vítek
Department of Radioelectronics
Czech Technical University in Prague

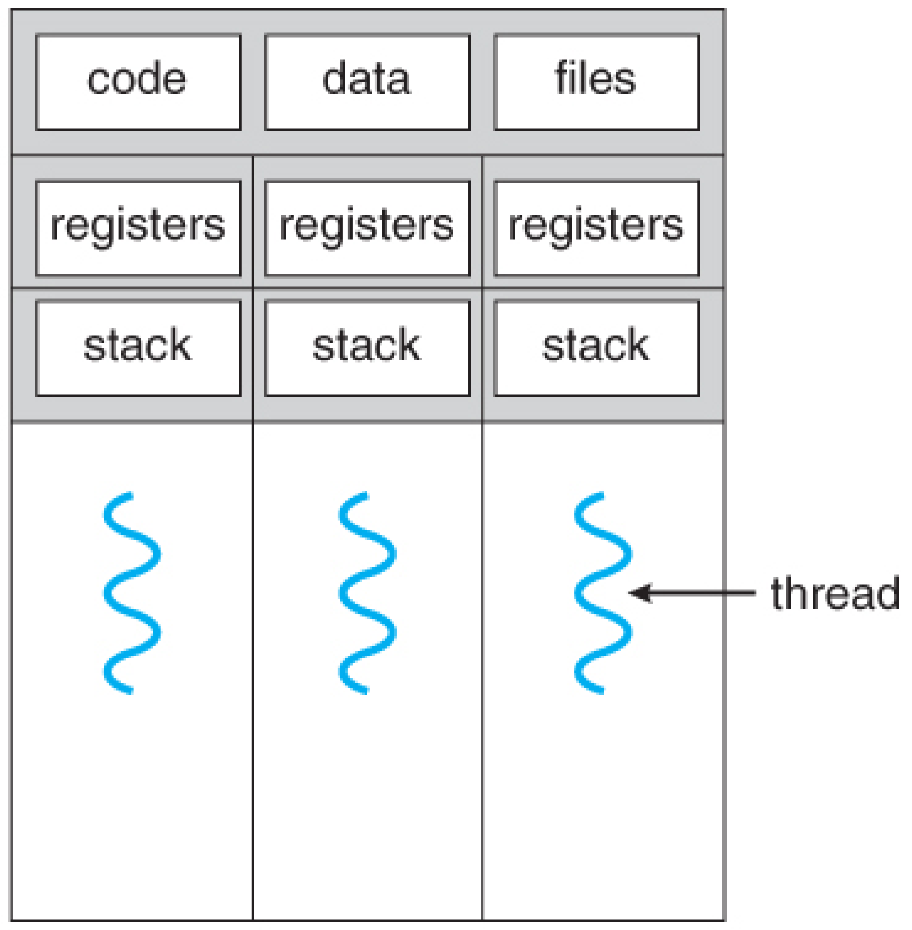
Thread

- **Thread:** An independent execution of a sequence of instructions.
    - It is individually performed computational flow.
    - Typically a small program that is focused on a particular part.
- Thread runs within the process.
    - It shares the same memory space as the process.
    - Threads running within the same memory space of the process.

Single-threaded and multithreaded processes



single-threaded process



multithreaded process

Thread runtime environment

- Each thread has its own separate space for variables.
- Thread identifier and space for synchronization variables.
- Program counter (PC) or Instruction Pointer (IP) -- address of the performing instruction.
- Indicates where the thread is in its program sequence.
- Memory space for local variables (stack).

Where Threads Can be Used?

- Threads are lightweight variants of processes that share the memory space.
- Useful cases for using threads:
  - **More efficient usage of available computational resources:**
    - When a process waits for resources, another thread within the same process can utilize the dedicated time for process execution.
    - Multi-core processors can speed up computation using parallel algorithms.
  - **Handling asynchronous events:**
    - During blocked I/O operation, another thread can be dedicated to I/O operations, while others handle computations.

Examples of Threads Usage

- **Input/output operations:**
    - Input operations can take significant portions of the run-time.
    - Dedicated CPU time can be utilized for computationally demanding operations during communication.
- **Interactions with Graphical User Interface (GUI):**
    - Graphical interface requires immediate response for a pleasant user interaction.
    - Computationally demanding tasks should not decrease interactivity.

Threads and Processes

Process:	Threads of a process:
• Computational flow. • Has its own memory space. • Entity (object) of the OS. • Synchronization using OS (IPC). • CPU allocated by OS scheduler. • Time to create a process.	• Computational flow. • Running in the same memory space of the process. • Synchronization by exclusive access to variables. • CPU allocated within the dedicated time to the process. • Creation is faster than creating a process.

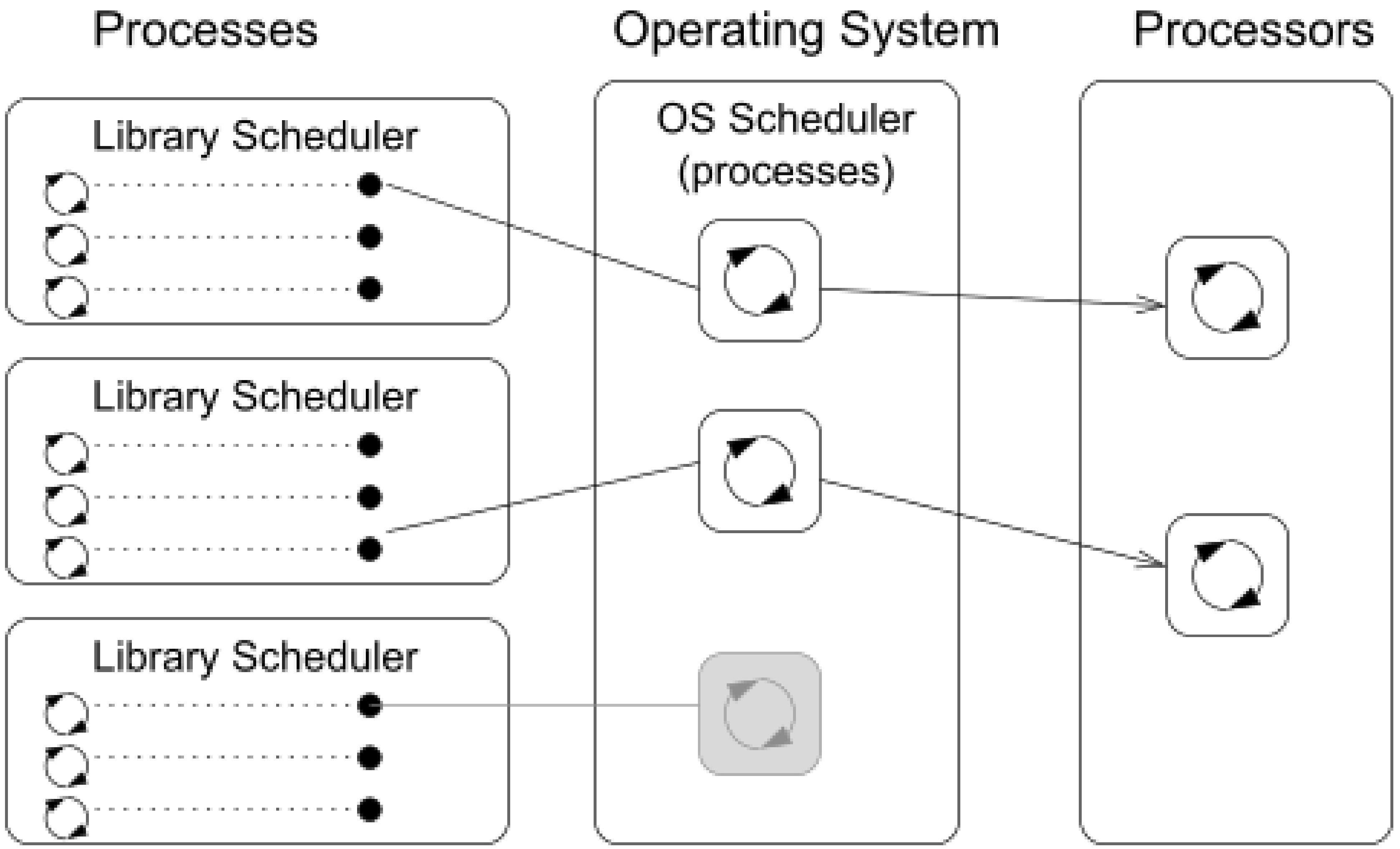
Multi-thread and Multi-process Applications

- **Multi-thread application:**
  - Application can enjoy a higher degree of interactivity.
  - Easier and faster communication between threads using the same memory space.
  - Does not directly support scaling parallel computation to distributed computational environments.
- Even on single-core single-processor systems, multi-threaded applications may better utilize the CPU.

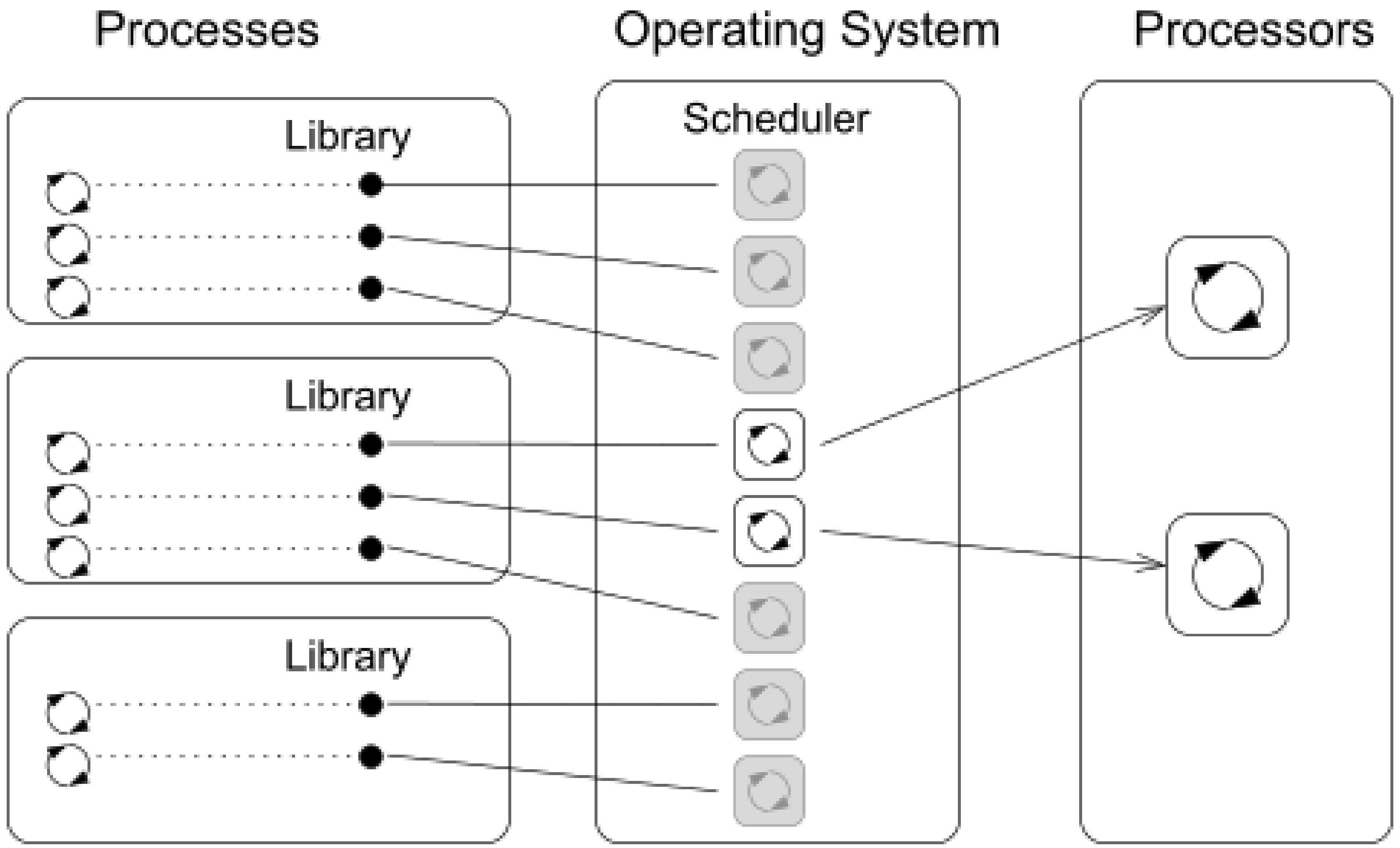
Threads in the Operating System

- Threads are running within the process.
- Regarding the implementation, threads can be:
  - **User space of the process:**
    - Threads are implemented by a user-specified library.
    - Threads do not need special support from the OS.
    - Threads are scheduled by the local scheduler provided by the library.
    - Threads typically cannot utilize more processors (multi-core).
  - **OS entities:**
    - Scheduled by the system scheduler.
    - May utilize multi-core or multi-processor computational resources.

Threads in the User Space



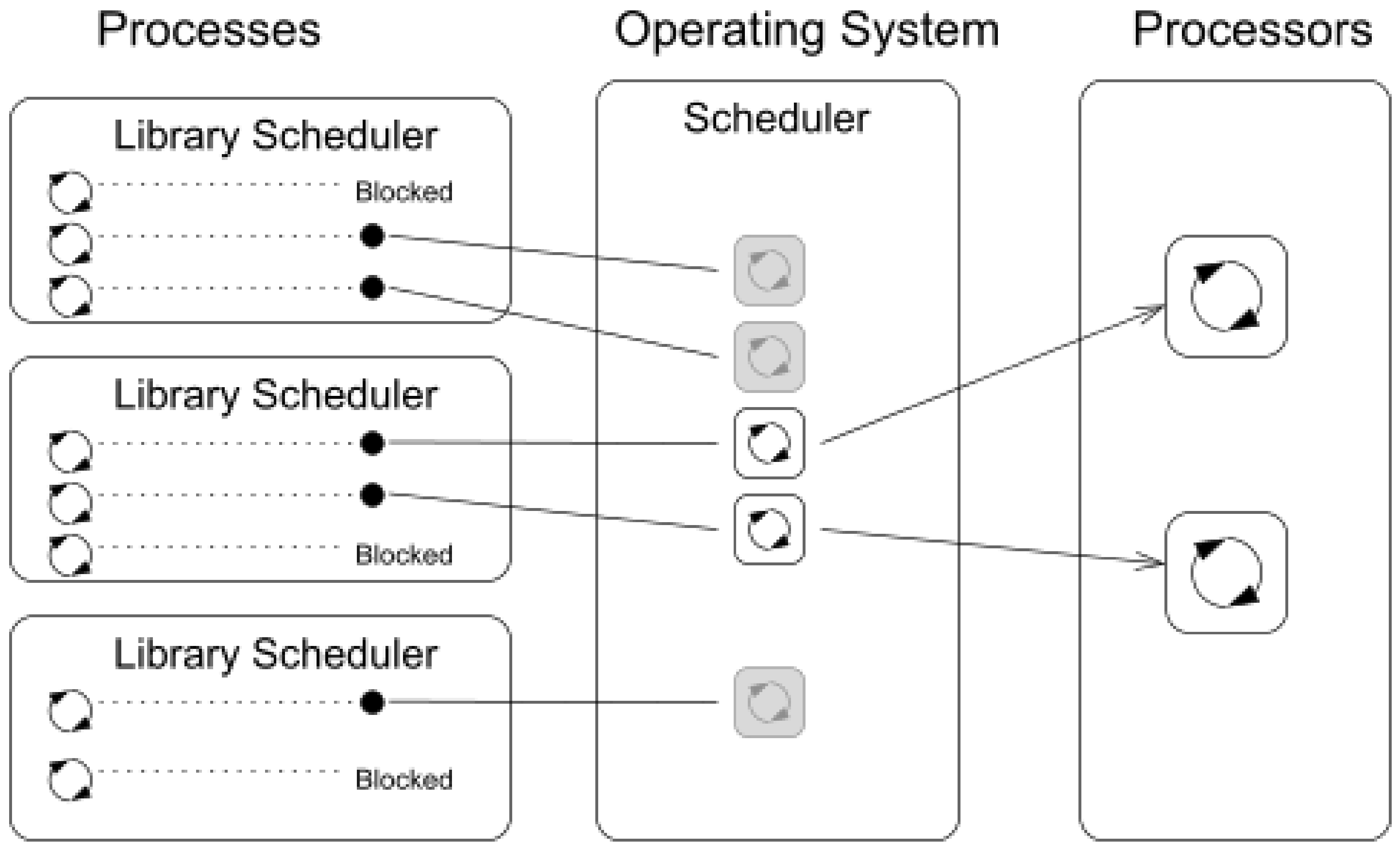
Threads as Operating System Entities



User Threads vs Operating System Threads

- **User Threads:**
  - Do not need support from the OS.
  - Creation does not need an expensive system call.
  - Execution priority of threads is managed within the assigned process time.
  - Threads cannot run simultaneously (pseudo-parallelism).
- **Operating System Threads:**
  - Threads can be scheduled in competition with all threads in the system.
  - Threads can run simultaneously (on multi-core or multi-processor systems -- true parallelism).
  - Thread creation is a bit more complex (system call).

Combining User and OS Threads



Models of Multi-Thread Applications

When to use Threads

- Threads are advantageous whenever the application meets any of the following criteria:
    - It consists of several independent tasks.
    - It can be blocked for a certain amount of time.
    - It contains a computationally demanding part (while keeping interactivity).
    - It has to promptly respond to asynchronous events.
    - It contains tasks with lower and higher priorities than the rest of the application.
    - The main computation part can be speeded up by a parallel algorithm using multi-core processors.

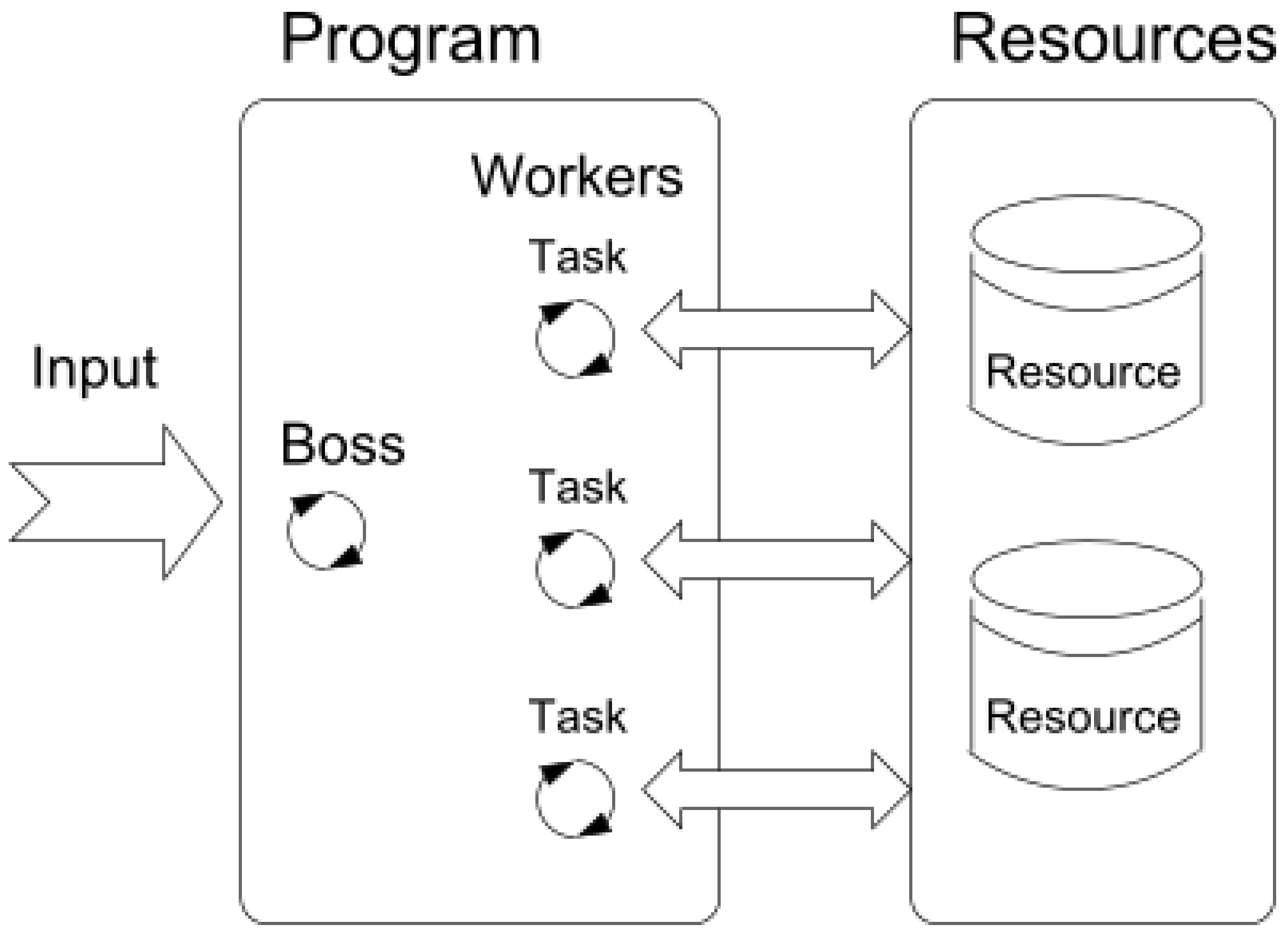
Typical Multi-Thread Applications

- **Servers:**
    - Serve multiple clients simultaneously.
    - May require access to shared resources and many I/O operations.
- **Computational application:**
    - With multi-core or multi-processor systems, the application runtime can be decreased.
- **Real-time applications:**
    - Utilize specific schedulers to meet real-time requirements.
    - More efficient than complex asynchronous programming.

Models of Multithreading Applications

- Models address the creation and division of work among threads.
  - **Boss/Worker:**
    - The main thread controls the division of work to other threads.
  - **Peer:**
    - Threads run in parallel without a specified manager.
  - **Pipeline:**
    - Data processing by a sequence of operations.

Boss/Worker Model

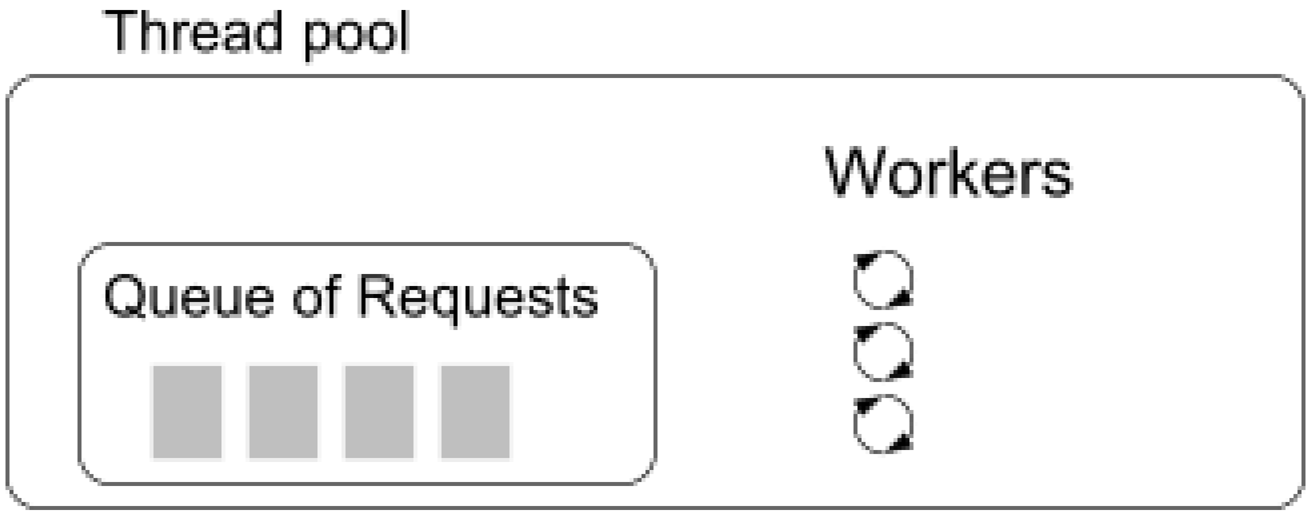


Boss/Worker Model - Roles

- The main thread is responsible for managing requests.
    1. Receive a new request.
    2. Create a thread for serving the particular request (or pass the request to an existing thread).
    3. Wait for a new request.
- Output/results of the assigned request can be controlled by:
    - The particular thread (worker) solving the request.
    - The main thread using synchronization mechanisms (e.g., event queue).

Thread Pool

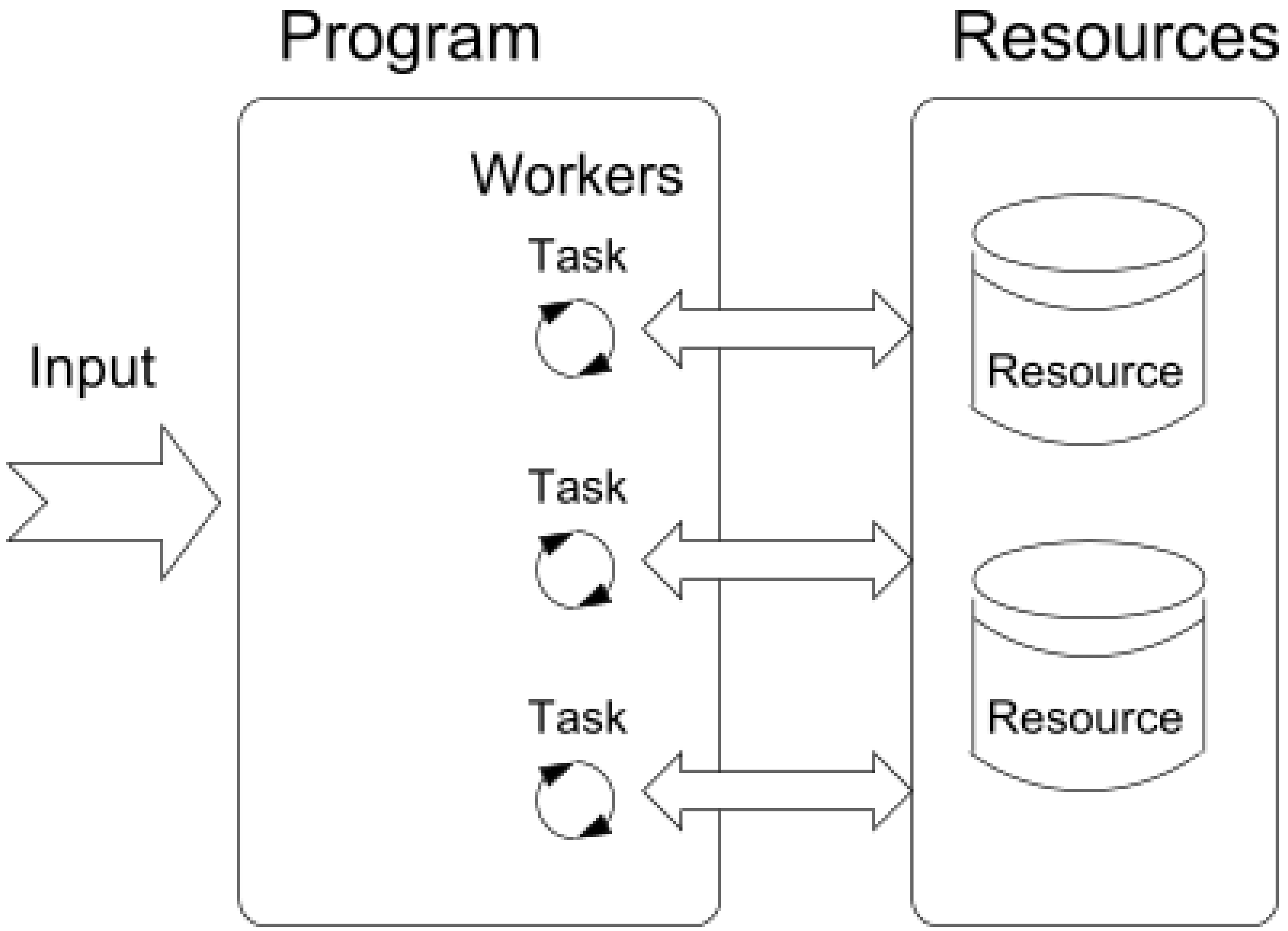
- The main thread creates threads upon receiving a new request.
- Overhead with the creation of new threads can be reduced using the **Thread Pool** with already created threads.
- The created threads wait for new tasks.



Thread Pool

- Properties of the thread pool to consider:
  - Number of pre-created threads
  - Maximal number of requests in the queue
  - Definition of behavior if the queue is full and no threads are available
    - E.g., block incoming requests.

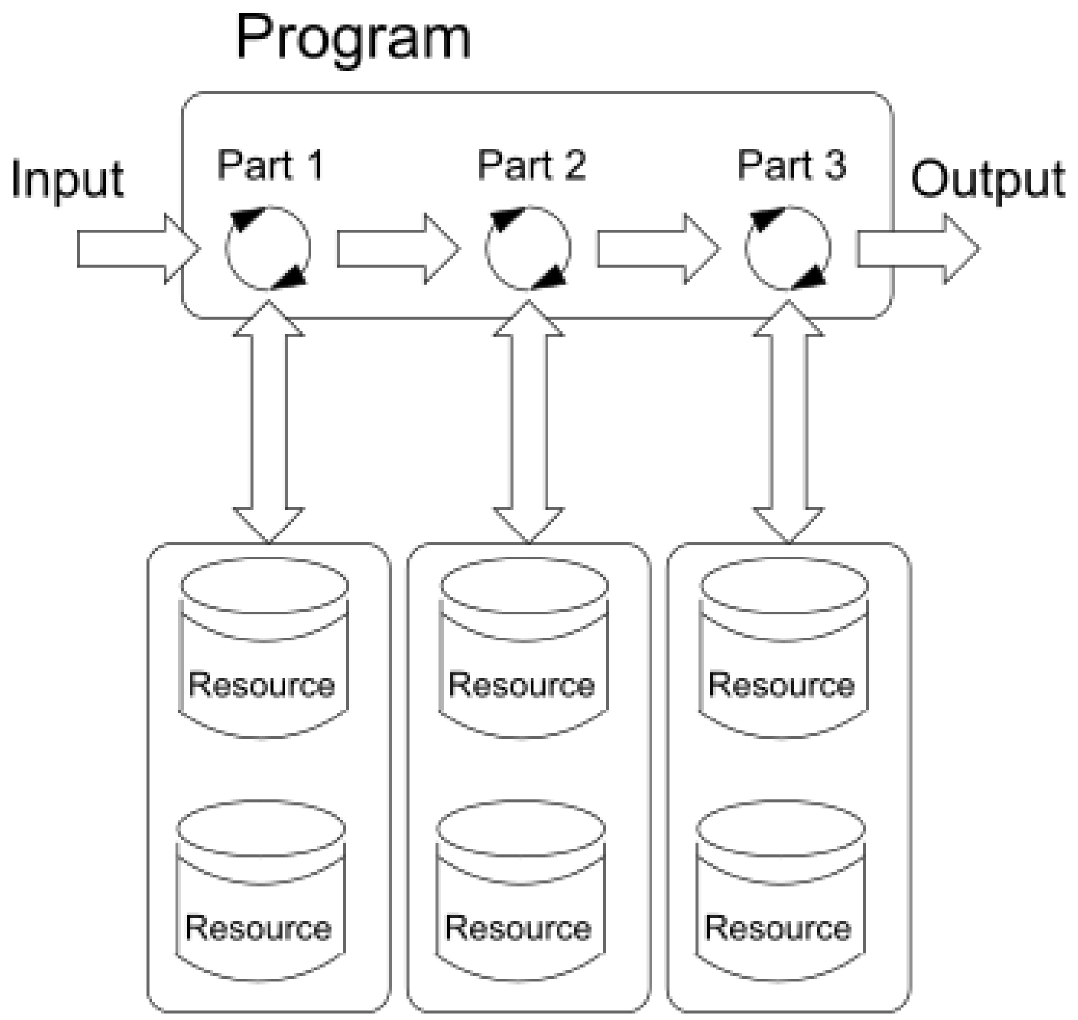
Peer Model


Program
Resources
Workers
Task
Task
Task
Input
Resource
Resource

Peer Model Properties

- It does not contain the main thread.
- The first thread creates all other threads and then:
  - Becomes one of the other threads (equivalent).
  - Suspends its execution and waits for other threads.
- Each thread is responsible for its input and output.

Data Stream Processing -- Pipeline

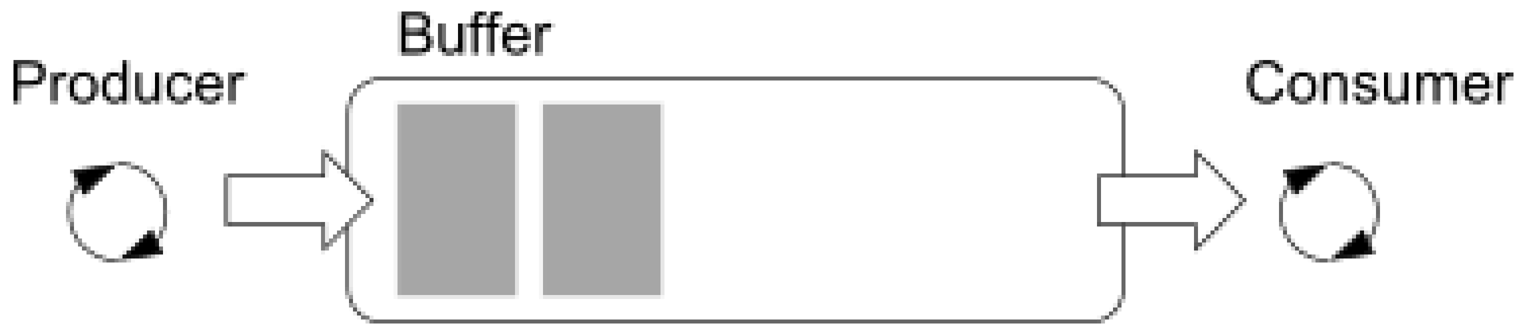


Pipeline Model -- Properties and Example

- A long input stream of data with a sequence of operations (a part of processing).
- Each input data unit must be processed by all parts of the processing operations.
- At a particular time, different input data units are processed by individual processing parts, and the input units must be independent.

Producer--Consumer Model

- Passing data between units can be realized using a memory buffer.
    - Or just a buffer of references (pointers) to particular data units.
- Producer: Thread that passes data to another thread.
- Consumer: Thread that receives data from another thread.
- Access to the buffer must be synchronized (exclusive access).



- Using the buffer does not necessarily mean the data are copied.

Synchronization Mechanisms

Mutex - A Locker of Critical Section

- Mutex is a shared variable accessible from particular threads.
- Basic operations that threads may perform on the mutex:
    - **Lock:** Acquired the mutex to the calling thread.
        - If the mutex cannot be acquired by the thread (because another thread holds it), the thread is blocked and waits for mutex release.
    - **Unlock:** Unlock the already acquired mutex.
        - If there is one or several threads trying to acquire the mutex (by calling lock on the mutex), one of the threads is selected for mutex acquisition.

Generalized Models of Mutex

- Recursive: The mutex can be locked multiple times by the same thread.
- Try: The lock operation immediately returns if the mutex cannot be acquired.
- Timed: Limit the time to acquire the mutex.
- Spinlock: The thread repeatedly checks if the lock is available for acquisition.
    - Thread is not set to blocked mode if the lock cannot be acquired.

Spinlock 1/2

Under certain circumstances, it may be advantageous not to block the thread during the acquisition of the mutex (lock).

- Performing a simple operation on shared data/variable on a system with true parallelism (using a multi-core CPU).
- Blocking the thread, suspending its execution, and passing the allocated CPU time to another thread may result in significant overhead.
- Other threads quickly perform other operations on the data, and thus, the shared resource would be quickly accessible.

Spinlock 2/2

- During locking, the thread actively tests if the lock is free.
    - It wastes CPU time that can be used for productive computation elsewhere.
- Similarly to a semaphore, such a test has to be performed by the TestAndSet instruction at the CPU level.
- *Adaptive mutex* combines both approaches to use *spinlocks* to access resources locked by the currently running thread and block/sleep if such a thread is not running.
    - It does not make sense to use spinlocks on single-processor systems with pseudo-parallelism.

Condition Variable

- *Condition variable* allows signaling a thread from another thread.
- The concept of *condition variable* allows the following synchronization operations:
  - Wait: The variable has been changed/notified.
  - Timed waiting for a signal from another thread.
  - Signaling another thread waiting for the condition variable.
  - Signaling all threads waiting for the condition variable.
    - All threads are awakened, but access to the condition variable is protected by the mutex that must be acquired, and only one thread can lock the mutex.

Example -- Condition Variable

- Example of using a condition variable with a lock (mutex) to allow exclusive access to the condition variable from different threads.

```
Mutex mtx; // shared variable for both threads
CondVariable cond; // shared condition variable

// Thread 1                                    // Thread 2
Lock(mtx);                                     Lock(mtx);
// Before code, wait for Thread 2              ... // Critical section
CondWait(cond, mtx); // wait for cond          // Signal on cond
... // Critical section                        CondSignal(cond, mtx);
UnLock(mtx);                                   UnLock(mtx);
```

Parallelism and Functions

- In a parallel environment, functions can be called multiple times.
- Regarding parallel execution, functions can be:
  - **Reentrant**: At a single moment, the same function can be executed multiple times simultaneously.
  - **Thread-Safe**: The function can be called by multiple threads simultaneously.
- To achieve these properties:
  - *Reentrant function* does not write to static data and does not work with global data.
  - *Thread-safe function* strictly accesses global data using synchronization primitives.

Main Issues with Multithreading Applications

The main issues/troubles with multiprocessing applications are related to synchronization.

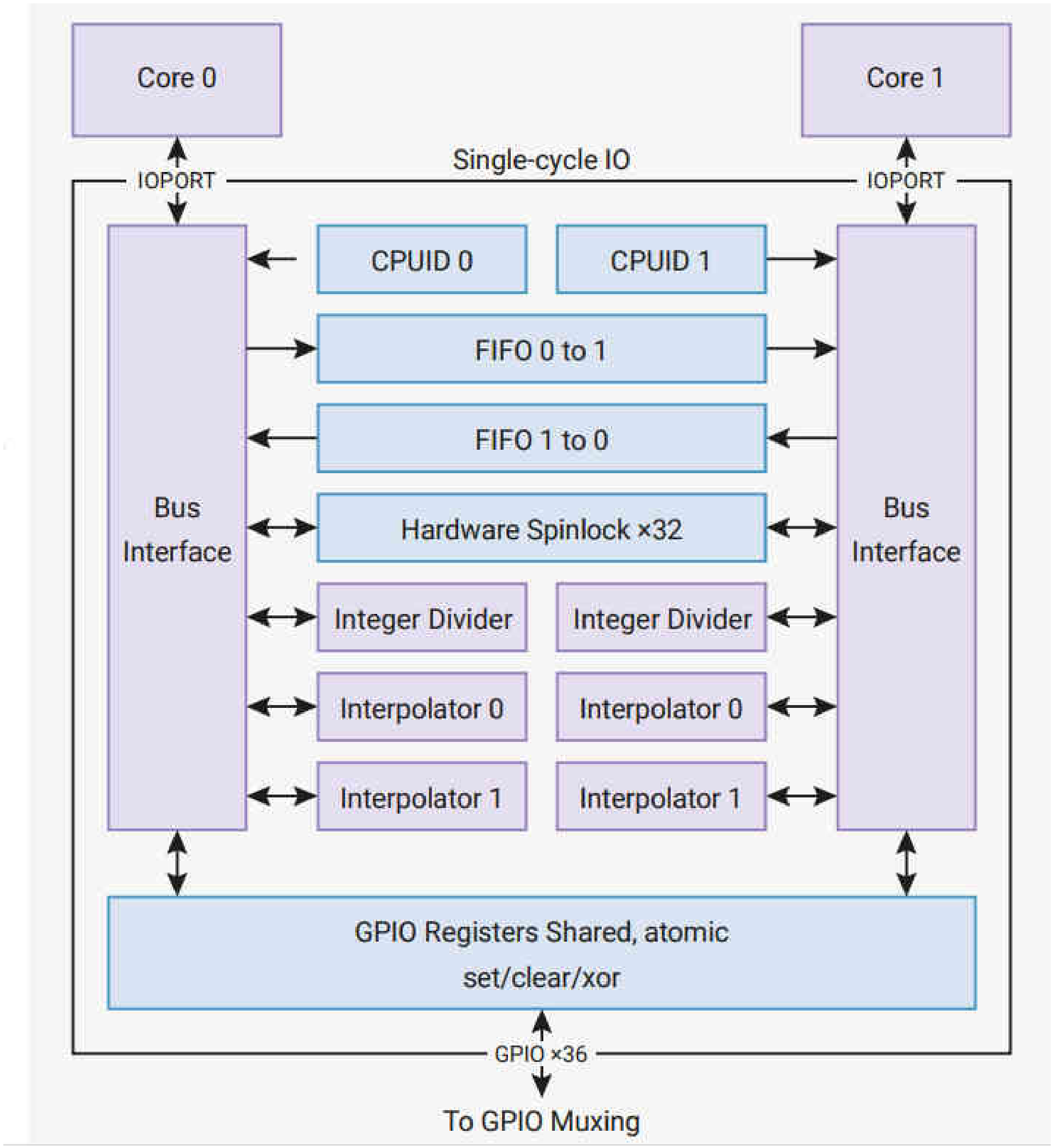
- **Deadlock**: A thread waits for a resource (mutex) that is currently locked by another thread that is waiting for the resource (thread) already locked by the first thread.
- **Race condition**: Access of several threads to shared resources (memory/variables), and at least one of the threads does not use the synchronization mechanisms (e.g., critical section).
    - A thread reads a value while another thread is writing the value.
    - If reading/writing operations are not atomic, data are not valid.

Raspberry Pi Pico Dual Core Programming

with MicroPython

Raspberry Pi Pico

- RP2040 based module has two processing cores, Core 0 and Core 1
- Deafault mode:
    - Core 0 executes all the tasks
    - Core 1 remains idle or on standby mode
- MicroPython provides a _thread package to handle the division and running of our code on separate cores.



Communication between cores

- To make the cores to communicate with each other the Raspberry Pi Pico module is featured with two individual FIFO structures.
- Each core can access only one FIFO structure so both core have their own FIFO structure to write codes which helps in avoiding race condition or writing to the same memory location simultaneously.
- **Semaphores** or **Simple locks** are provided with **_thread** module which is responsible for synchronization between multiple threads.
    - The **allocate_lock()** function provided with **_thread** module is responsible for returning a new lock object.

The _thread package

- Our main Python code will automatically start on **Core 0**
- We can then tell the **_thread** package to start another block of code on **Core 1**

```
def thread_function():
  # code to be running on Core 1

new_thread = _thread.start_new_thread(thread_function, args, [,kwargs])
```

- thread_function is a reference to a standard Python function that contains the code for the new thread. This must be followed by a tuple containing the function arguments and then an optional dictionary or keyword arguments for the function.

Multi-Threading Example

```python
from time import sleep
import _thread

def core0_thread(counter = 0):
    while True:
        print(counter)
        counter += 2
        sleep(1)

def core1_thread(counter = 1):
    while True:
        print(counter)
        counter += 2
        sleep(2)

second_thread = _thread.start_new_thread(core1_thread, ())

core0_thread()
```

Communication between threads

```python
def core0_thread():
    global run_core_1
    # do something
    # signal core 1 to run
    run_core_1 = True
    # wait for core 1 to finish
    while run_core_1:
      pass

def core1_thread():
    global run_core_1
    while True:
        # wait for core 0 to signal start
        while not run_core_1:
            pass
        # do something
        # signal core 0 code finished

run_core_1 = False

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sharing resources

- Sometimes we need to be very careful about who and when a thread can have access to some data, or some resource, e.g. the SPI interface.
  - If both threads try to use or update the same resource at the same time we'll either get corrupted data or potentially crash part of our code.
- Flag commented above is simple a works in well defined situations. When you need more flexible control you need to use a Lock.
  - A Lock (or semaphore in concurrent programming) allows us to control access.
  - We create a Lock object and only the owner of the Lock can use the resource.
  - Every other thread has to wait for the Lock owner to release it before one of the waiting threads can take ownership and get access to the resource.

Sharing resources - race condition

```
def core0_thread():
  while True:
    print('A')
    sleep(0.5)

def core1_thread():
  while True:
    print('B')
    sleep(0.5)

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sharing resources - using lock (mutex)

```
def core0_thread():
  global lock
  while True:
    lock.acquire()
    print('A'); sleep(0.5)
    lock.release()

def core1_thread():
  global lock
  while True:
    lock.acquire()
    print('B'); sleep(0.5)
    lock.release()

lock = _thread.allocate_lock()

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Communication between threads using Queue (FIFO)

- Queue() has a thread-safe implementation with all the required locking mechanism
    - it could basically pass a queue as an argument to the second thread

Core 0 thread:

```
q = queue.Queue()
_thread.start_new_thread(second_core_thread, (q) )

while True:
  msg = q.get()
```

Core 1 thread (second_core_thread):

```
def writer(q):
  queue.put(....)
```