Embedded System Design and Modeling

IX. SchedulingCommunication Models

Models of communications

- Consider multiple models working in parallel composition
  - i.e. multiple models, processes, threads etc
- How do they communicate?
- Typical scenarios:
  - two models (FSM's) on the same machine, executed on same CPU thread
  - two models on the same machine, executed on parallel threads/CPUs (parallelization)
  - two models on different machines
- It is a general topic in multi-threaded programming (working with threads, processes etc)

Models of communications

Two communication paradigms:

- Communicate via shared memory / variables
  - both processes read/write some variable **directly**
  - one process writes it, the other process reads it
- Communicate via message passing
  - blocking (synchronous)
  - non-blocking (asynchronous)

Shared memory

- **Shared** variables = variables which can we written / read by both models
- Potential problems:
  - What happens if both models try to access (read or write) the variable **at the same time**?
  - What happens if a thread is interrupted right in the middle of a read/write operation
- Answer: possibly something bad. Might end up with an incorrect value
- Solution: access to shared variable must be via **atomic operations** or guarded with a **mutex**

Shared Memory

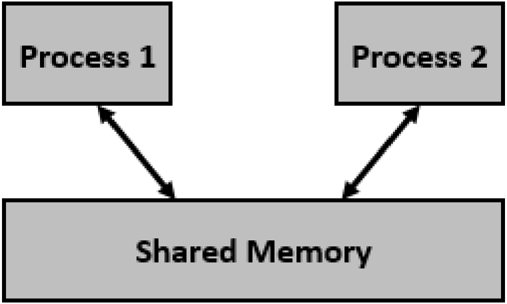


Figure 1: Shared Memory illustrated

[1]

[1] image from https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_shared_memory.htm

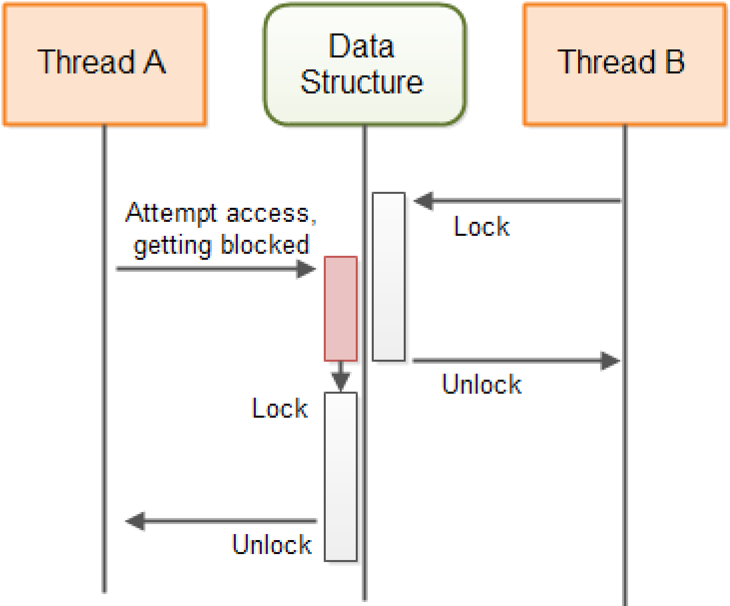
Shared memory

- **Atomic** operation = an operation that is indivisible (once it starts, it can't be interrupted until it ends)
  - it is either fully done, or not done
- Typical atomic operations:
  - setting / getting a value for a built in datatype, e.g
    - `a = 5;`
    - `is_Enabled = False;`
- Non-atomic operations: everything else
  - calling a function
  - e.g. inserting/removing an element in a vector
  - setting multiple variables (can be interrupted inbetween)
  - ...

Mutex (lock)

- **Mutex** (or **lock**) = a mechanism for ensuring only one process accesses a given resource (e.g. variable) at one time
  - A process first **acquires** the mutex, if it is available
  - Only afterwards it accesses the variable
  - While the mutex is acquired, no other process can access it
  - The process **releases** the mutex when it's done with the variable
  - The code between acquiring and releasing the mutex is known as a **critical section**
- Mutexes are provided by the operating system, and are used in code via library functions provided by the OS
- Example: Python

Mutex (lock)



Thread A
Data Structure
Thread B
Lock
Attempt access,
getting blocked
Unlock
Lock
Unlock

Mutex (lock)

```python
lock = threading.Lock()
def thread_function_1():

    # Acquire lock
    with lock:
        print("Thread 1 acquired lock. Writing...")
        write_shared_memory()

    # Lock is released
    # In Python this happens automatically
    #   when exiting the `with` context manager
```

Mutex (lock)

```python
def thread_function_2():

    # Acquire lock
    with lock:
        print("Thread 2 acquired lock. Reading...")
        read_shared_memory()

    # Lock is released
    # In Python this happens automatically
    #   when exiting the `with` context manager
```

Shared memory

- There can be multiple writers, multiple readers of the shared data
- It is up to the designer to ensure the synchronization between all the participants

Message passing

Message passing

- Communication is achieved explicitly via **messages** which are sent and received
- Two variants:
  - blocking
  - non-blocking

Message passing

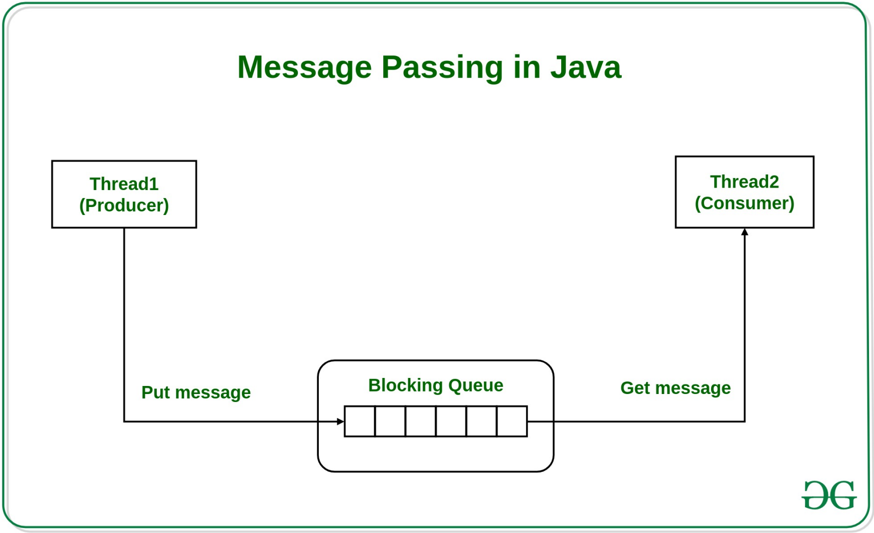


Figure 3: Message Passing illustrated

Message passing: blocking

Message passing: **blocking (synchronous)**

- When the sender sends, it **waits** for the receiver to acknowledge that is has received the data
- When the receiver reads, it **waits** for the data
- Basically, the earlier one waits for the other one
- Works like a courier

Message passing: non-blocking

Message passing: **non-blocking (asynchronous)**

- There is a sender process and a receiver process
- When the sender sends, it **stores** the data somewhere, and goes on
- When the receiver reads, it **collects** (if available) the data and goes on
- Neither process waits
- Works like the post office

Message passing

Comparing blocking vs non-blocking:

- Storage:
  - Non-blocking communication needs a storage mechanism (FIFO, LIFO, Queue, list etc.)
    - This storage space may overflow => need to have safety mechanisms in place to avoid **buffer overflow**
  - Blocking communication does not need any special soneeds a storage space (FIFO, LIFO, Queue, list etc.)
- Delays:
  - Non-blocking communication doesn't delay the sender nor the receiver
  - Blocking communication delays one of the processes until the other one is ready
- Examples:...