Exercise 2

Process and Thread synchronisation

Fabian Fröschl 2023 - fabianf96@googlemail.com

Why do we need that?

- Processes can be executed in parallel, and processes may be multithreaded
  - Data sharing might be necessary among processes/threads
  - Data inconsistencies may occur => Can lead to software malfunctions that are not checked => Must be avoided!
- Different theoretically described problems exist
- Different handling mechanisms exist

The Producer/Consumer (Bounded Buffer) problem

- Two processes/threads exist, one makes data available (Producer), one makes use of that data (Consumer)
  - Can be extended to multiple Producers and Consumers
- Queue between Producers/Consumers serves as buffer for data
  - Consumers need time for processing in which Consumers may or may not add data
- Problem: Assure that Consumers do not try to get data from empty queue + Consumers do not add to full queue => Causes exceptions

The Reader/Writer problem

- One or more Writers exist which create data into a shared data set
  - Can read, update and delete data
- One or more Readers exist which **only** read data from shared data set
  - Can read data
- Goal: Uphold data consistency and keep data up to date
  - Problem: Multiple readers should be able to read at the same time
  - Problem: Only a single writer can access data set at the same time

The Reader/Writer problem

- Two variations exist:
  - 1: Readers have priority => Readers are not kept waiting **except** for when a Writer is accessing and writing
    - Can lead to Writer starvation => Data in the data set might be outdated (=stale)
  - 2: Writers always get immediate access to data set except for when another Writer is accessing and writing
    - Can lead to Reader starvation => Data is kept up to date but is never read and used
- Goal: Assure data consistency without starvation

Race condition

- Simple solution: Use while loops that use a shared control variable?
  - Works conceptually if only code is considered
  - Can still lead to inconsistencies in practice
    - CPU needs time to process control variable from one process/thread
    - While CPU is processing the control variable it may be accessed by another process/thread => Now the second thread works with the non-updated value
    - When CPU is done with both processes/threads the value of the control variable is inconsistent!
    - This happens because CPU buffers data in registers for computation
  - This is called a **race condition** => Avoidance is one goal of synchronisation
    - The outcome of an execution depends on order of access and cannot be controlled

The Critical Section Problem

- Goal: Avoid race conditions

- Each one of n processes has a **critical section**
  - Part of code where shared resources are accessed
  - When one process enters its critical section, no others may to enter theirs
  - A critical section is always defined by the shared resource

- Structure of process around critical section
  - Entry section: Get permission to enter critical section
  - Exit section: Return permission and continue with remainder section

The Critical Section Problem

- Critical Section Problem = Design a protocol to synchronize critical sections without race conditions and starvation

- Requirements for a solution:
  - **Mutual Exclusion**: If one process/thread is executing its critical section, no other is allowed to enter its
  - **Progress**: If entering the critical section is possible and a process/thread wants to enter its, it must be allowed to do so in finite time
  - **Bounded Waiting**: A bound must exist for how often a process/thread that wants to execute its critical section is postponed in favour of another one

The Critical Section Problem - Solutions

- Interrupt-based solution
  - Use OS interrupts (traps)
  - Disable interrupts in entry section, enable them in the exit section
  - Can lead to starvation if any process remains in its critical section for a long time => And the interrupts are blocked => **No OS services available for other apps**

The Critical Section Problem - Solutions

- Peterson´s solution for two processes:
  - Two control variables are used
    - A Boolean array containing a flag per process signalling intention to enter critical section
    - An integer serving as an identifier for a process
  - Entry section:
    - Process 1 sets own flag in array to signal intention to enter
    - Priority is given **to the process 2** via its id
    - Empty while loop blocks execution until process 2 does not intend to enter its critical section or gives priority to process 1
    - Same logic for process 2, only it gives priority to process 1 and checks process 1 in while loop

The Critical Section Problem - Solutions

- Peterson´s solution for two processes:
  - Fulfils all requirements to solve the critical section problem (can be proven mathematically)
  - But:
    - Assumes that load and store instructions in CPU are atomic (= Cannot be broken down and interrupted) to work
    - Because of this it may not work with modern compilers => More abstraction from the Operating System

The Critical Section Problem - Solutions

- Mutex locks
  - OS service providing a lock for critical sections => Only one thread allowed to hold it at any time
  - Thread/Process **acquires** lock in entry section and **releases** it in exit section
  - Availability is shown via a Boolean flag
  - Busy waiting => Threads/Processes that want to acquire a lock in use must loop when trying to acquire => **Spinlock**

The Critical Section Problem - Solutions

- Semaphore
  - On instantiation n permissions to access the critical section are created
  - Calls to **acquire/wait** in entry section succeed whenever a permit is free => (n-=1)
  - Calls to **release/signal** in exit section make a permit available => (n+=1)
  - Semaphore with 1 permit = **Binary Semaphore** = Mutex lock

The Critical Section Problem - Solutions

- Monitor (Thread safe class/object in Java)
  - In Java, any class has an in-built monitor object => Is made available to a function by using the keyword **synchronized**
  - **Wait()** is called in entry section which blocks execution until no other thread is in its critical section
  - Once a waiting thread receives a signal to continue it can enter its critical section => Signalling done via call to **notify()** or **notifyAll()** in exit section

The Dining-Philosophers problem

- Shared data (= bowl of rice)
- 5 philosophers (= processes)
- 5 chopsticks (= limited resources)
- All need to access data at times but need two chopsticks to do so

Deadlock

- Process level: Process waits for an event that will never occur
- System level: Multiple processes deadlocked and wait for each other
- Comes up because of resources are scarce and there might be a bigger need for resources than there are available resources

Deadlock - Conditions

- Four conditions must hold simultaneously for deadlock:
  - Mutual exclusion: A resource is only available for one process at a time
  - Hold and wait: A process P_x holds a resource and is waiting for a resource which another process P_y is holding at the same time
  - No pre-emption: A resource cannot be taken from a process once it holds it
  - Circular wait: Processes waiting for occupied resources form a circle

- Resource-Allocation graph can be used to visualize the situation
  - Can show if a system is not deadlocked or if deadlock **is possible**

Deadlock - Solutions

- Deadlock prevention
  - Avoid any of the four conditions to be fulfilled
  - Mutual exclusion always holds for non-shareable resources (e.g., chopsticks)
  - Other three conditions can be avoided but avoiding hold and wait may introduce starvation

- Deadlock detection
  - System can enter deadlock and in that case a recovery scheme to free resources is run

Deadlock - Solutions

- Deadlock avoidance
  - Safe state exists if there is an order of execution where all processes get resources they need
  - Avoidance algorithm controls state of system resources and manages assignment of resources to processes to keep system in a safe state
  - Processes must inform the system about their **maximum need** before start
  - If the system is always in a safe state, no deadlock can happen
  - E.g., Resource-Allocation-Graph or **Banker´s algorithm**