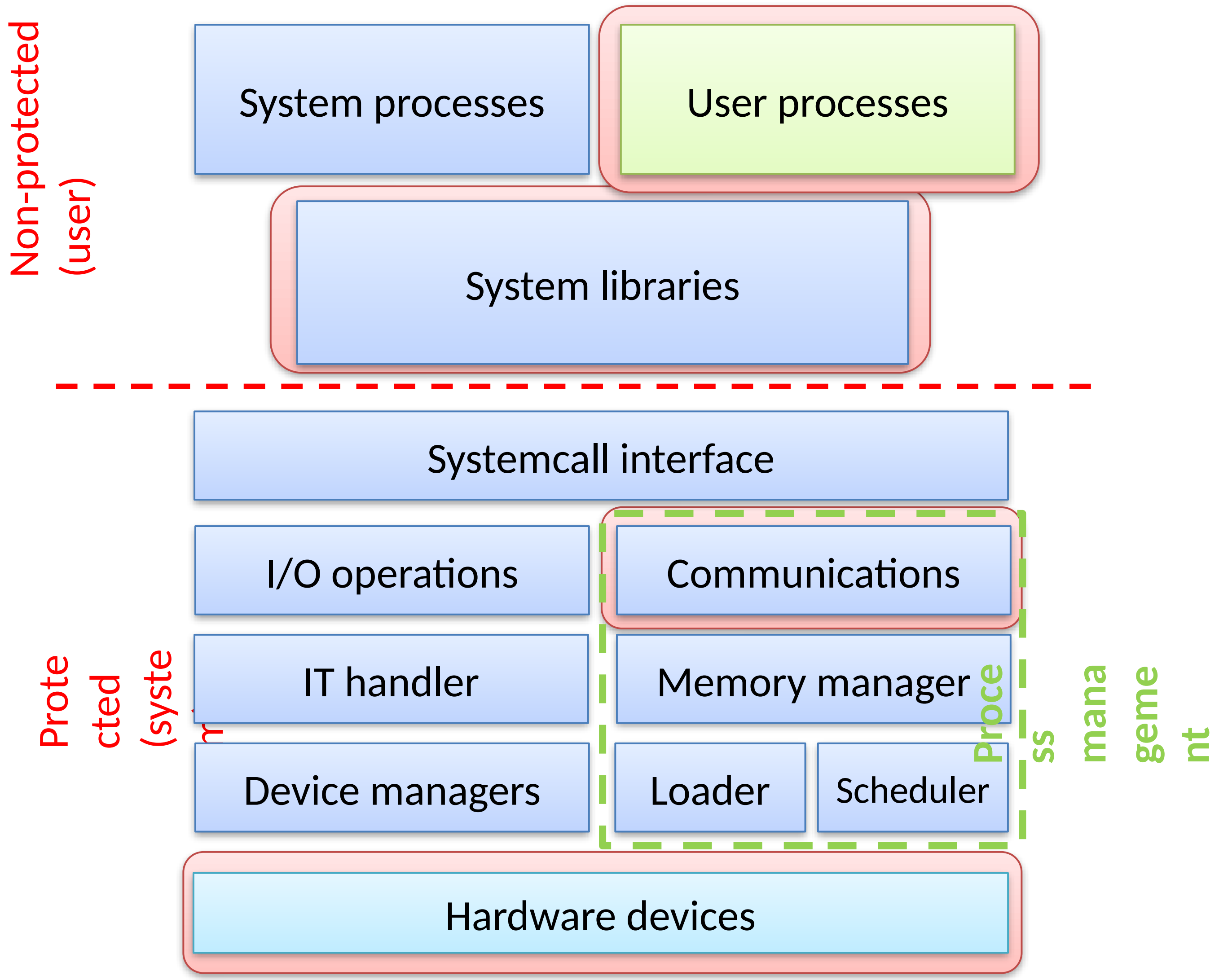
Operating Systems – Synchronization between tasks

Péter Györke
http://www.mit.bme.hu/~gyorke/
gyorke@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The main blocks of the OS and the kernel (recap)



Parallel job execution

- The basic goal of the OS is to support user job execution
  - Jobs are executed by tasks (maybe more than one)
  - Executing jobs may require the executor tasks to **cooperate**
  - A modern OS is multiprogrammed ▭ executing jobs parallel
- Task implementations: processes and threads
  - Threads in the same process are using shared memory ▭ **competitive** environment
  - Processes are separated
    - Communication has to be synchronized
    - Parallel running processes may have to compete for the resources
- Current systems require parallel programming
  - The clock frequency is almost reached its technological boundary
  - Multithreaded execution is the design principle ▭ Multicore CPUs
  - This also requires a new programming principle

Competition and cooperation between tasks

- Tasks may operate independently from each others
  - Not influencing each others operation
  - Asynchronous execution
  - This separation is made possible by the OS
  - The resources (CPU, RAM, HW devices) are used by more than one task ▭
    - Conflicts may appear
    - Execution dependencies created
  - These conflicts have to be solved by the OS

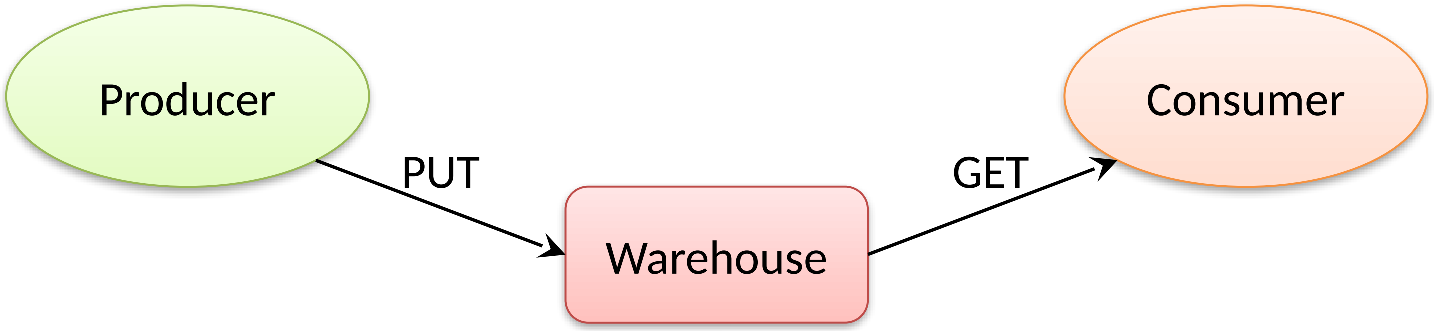
- The user jobs also require the tasks to cooperate
  - The job is decomposed to separate tasks
  - These tasks have to cooperate ▭ communicate and synchronize with each other
  - The OS provide services for this

- Remark: A single processor system is also a competitive environment

Simple example: producer – consumer problem

- The description of the problem
  - The producer creates a product which is stored in a warehouse (in a variable)
  - The consumer consumes the product from the warehouse
  - The producer and the consumer are working simultaneously
    - They may work separately in time
    - They may work with different rates

- Problems to solve
  - Granting the consistency of the warehouse data structure
  - The consumer shouldn't check for products in an infinite loop
  - The producer shouldn't place new product in the warehouse while it's full



Synchronization between tasks

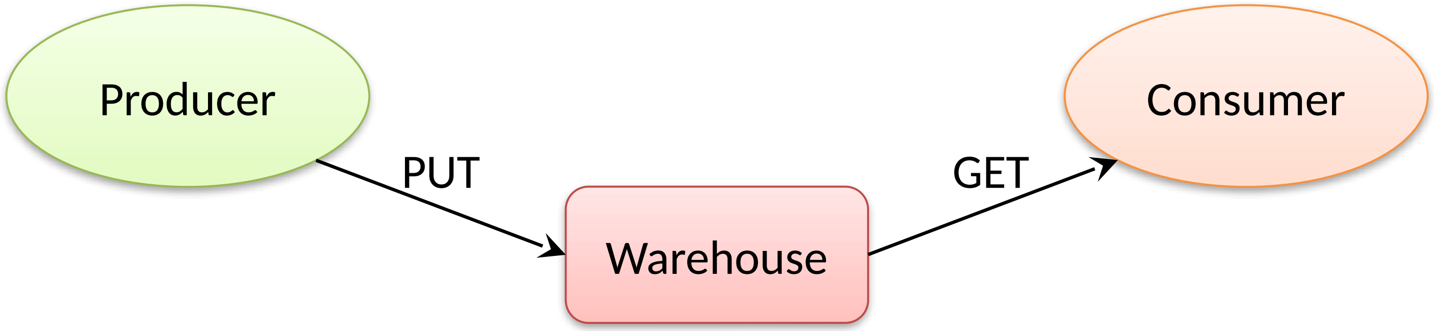
- Synchronization means coordination between tasks by constraining operation execution in time
  - The execution of specific tasks can be slowed down (temporally stopped) in order to achieve combined operation
- Basic application of synchronization
  - In competitive environments: using shared resources
  - In cooperation: communication
- The „price" of synchronization
  - It may cause performance degradation
  - The waiting tasks are not „useful"
    - They cannot wait for I/O operations, they are blocked
  - No sync.: no waiting, erroneous behavior is possible
  - Bad sync. scheme: too much waiting, bad resource utilization

The basic forms of synchronization

- Mutual exclusion
  - **Critical section**: an instruction sequence of the tasks, which are cannot executed simultaneously
  - Shared resources are protected with this method, so **competitive situations** can be managed
  - Pessimistic method: locks the resource in every case
  - E.g.: while the printer is printing, cannot start a new print
- Rendezvous
  - Specific operations of the tasks should start at the same time
  - **Cooperation** scheme to synchronize the operation of sub-tasks
  - E.g.: Send() and Receive() methods of direct (non buffered) messaging
- Precedence
  - The operations of the tasks should be executed in a predefined order
  - **Cooperation** scheme
  - E.g.: business intelligence SW: the data cleaner operations should run before the data analysis operations

Synchronization in the producer – consumer problem

- Mutual exclusion
  - The integrity of the warehouse data structure should be granted
  - While one party is writing, others shouldn't access the warehouse
  - More than one producer or consumer can use the same warehouse
- Precedence
  - The consumer should wait while the product is generated



The critical section

- The critical section is an instruction sequence with restricted execution: only one task can execute them at the same time
- N-critical section: N number of executions are possible at same time (e.g.: N number of resources are available)
- The rules of the restriction (N=1 case)
  - Entering
    - It is forbidden to enter, when another task is in the critical section
    - If no tasks in the critical section, only that task can enter, which are executing the instructions before the critical section
    - If a task wants to enter the critical section it may preceded by other tasks, but number of failed tries are limited
  - Exiting
    - The critical section should finished in finite time
    - Common programming mistake: the tasks don’t leave the critical section (releasing the resources)

Hardware support for synchronization

- Simple solution: disable interrupts when a task executes the critical section
  - ▭ no preemption is possible, mutual exclusion is realized
  - It makes the system cooperative (non preemptive)
  - It may used in single processor systems
  - Disabling the interrupts may lead to omitting important events
  - Cannot use it in a multiprocessor environment
    - Other operations also disabled on the other CPU-s
- Good solution: **atomic** (non interruptible) memory instruction pairs
  - **Test-and-set-lock (TSL)**: Sets the lock value to true
    - `while( TSL(lock) ) { }`
    - Waits until the lock is released and engages the new lock for the current task
    - The lock is usually a binary variable (true-false)
  - **Compare-and-swap (CAS)**: A variable is only modified when it has a specified value
    - `while ( CAS(var, a, b) == a ) {}`
    - Waits until `var` value is a, then sets it to b
    - Works on larger variables also (e.g. arrays)

Implementing critical sections with TSL and CAS operators

Test-and-set lock (TSL)

```
//non protected section
while(TSL(lock)){}
//critical section
lock = FALSE;
//further non protected
sections
```

Compare-and-swap (CAS)

```
//non protected section
while(CAS(lock, 0, 1) == 0) { }
//critical section
lock = 0;
//further non protected
sections
```

- Critical sections can be implemented with TSL and CAS hardware instructions
- Problems with this implementation
  - The task is in a while loop: busy waiting
  - In a single CPU system it prevents other tasks to run, and release the lock(s)
  - A blocking operation is better, so the task enters waiting state

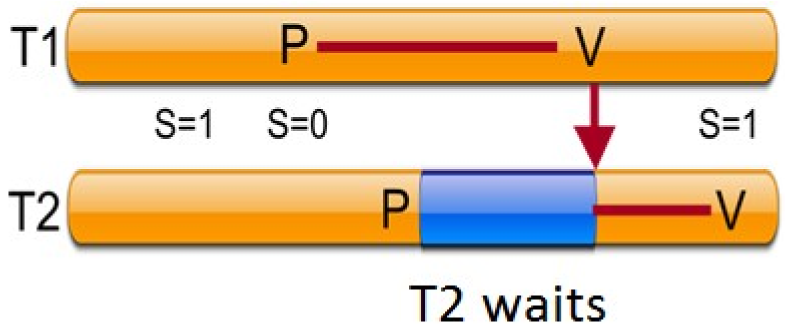
Overview of the locking methods

- Lock bit
  - Single bit, accessing it is atomic, e.g.: TSL
- Mutex (mutual exclusion lock)
  - A tool for implementing critical sections
- Semaphore
  - A data structure with two atomic operations: `wait(P)` and `signal(V)`
  - There are single and multiple access (N != 1) variants
- Spinlock (spinning lock)
  - Busy waiting lock bit, mutex or semaphore
  - E.g.: TSL and CAS instructions
  - It can implemented by SW only also
  - It should be used only when the lock is used for a short time
- ReaderWriterLock
  - Any number of readers may enter the critical section
  - If a writer is entering, it will be blocked until all of the readers are left the critical section
- RecursiveLock
  - The task which has the lock can re-lock the same lock withoud blocking
  - It is useful for recursive functions

The semaphore

- Semaphore (S): data structure with two atomic operations
- Binary semaphore: {0,1}
  - For protecting the critical section. The mutex is a binary semaphore
- Counter type semaphore: {0, 1, 2, …}
  - For protecting N-critical section

- P(S) operation: waits until the S value can be decremented by one
  - This is a blocking operation, so the task will enter into waiting state (good solution, others can run)
- V(S) operation: increments the S value by one
  - Release operation (leaving the critical section)
  - Non blocking



Implementation of the semaphore

- Data structures
  - Counter: `int count;`
  - FIFO queue: `queue_t waiting;`
  - Lock bit for the data structures: `lock_t lock;`
- Operations
  - If count == 0, the task calling P() will entered into waiting state
  - Calling V() will make the first waiting task into ready-to-run state
  - Providing the data structure integrity: e.g.: TSL

P()

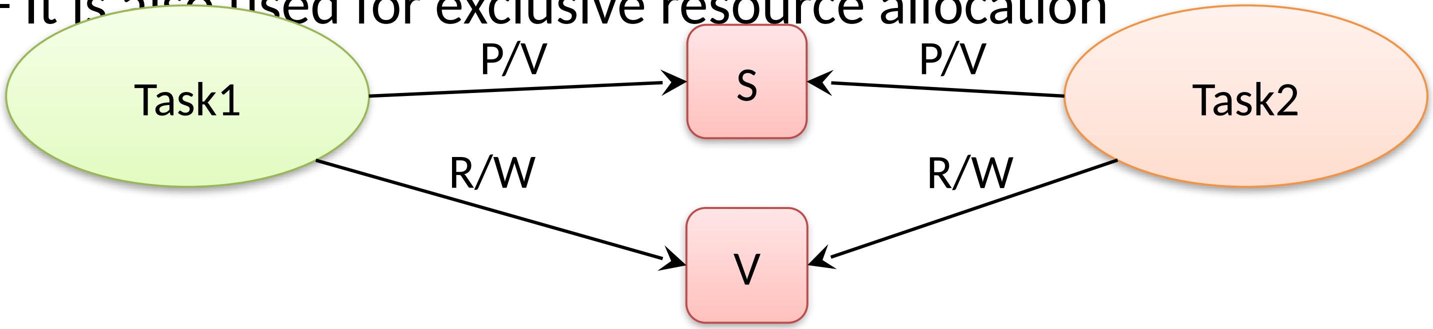
```
while (test_and_set(lock)) { }
if (count == 0) {
  fifo_add(waiting, T1);
  sched_block(T);
} else {
  count--;
}
lock = 0;
```

V()

```
while (test_and_set(lock)) { }
if (is_empty(waiting)) {
  count++;
} else {
  T2 = fifo_get_first(waiting);
  sched_wakeup(T2);
}
lock = 0;
```

Classic synchronization problem: reader-writer problem

- Description of the problem
  - The variable V is read and write by multiple independent tasks
  - Inconsistent states has to avoided
- Implementation
  - The critical section is protected by a semaphore
  - Leaving the critical section releases the semaphore
  - It is also used for exclusive resource allocation



Task1

```
P(S);
// reading/writing variable
V(S);
```

Task2

```
P(S);
// reading/writing variable
V(S);
```

The problem of multiple readers

- Description of the problem
  - The solution for the reader-writer problem is inefficient when the number of readers are high compared to the number of writers
  - It is unnecessary to block the readers while, the variable is not written
- Implementation
  - The readers are signaling the read, but not blocked if no write operation is active
  - The writers will be blocked when other reader/write operation is active

Start reading

```
P(reader_mutex);
++readerCount;
if (readerCount == 1) {
  P(writerLock);
}
V(reader_mutex);
```

Finish reading

```
P(reader_mutex);
--readerCount;
if (readerCount == 0) {
  V(writerLock);
}
V(reader_mutex);
```

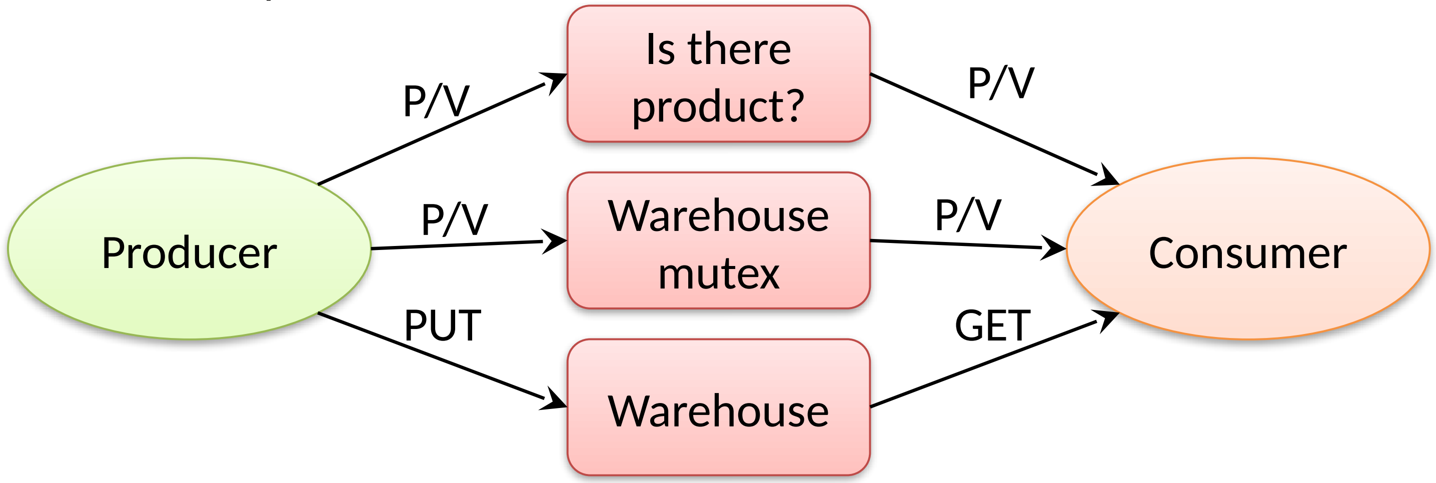
Writing

```
P(writerLock);
// writing variable
V(writerLock);
```

ReaderWriterLock implemented with semaphore

Classic synchronization problem: producer-consumer problem

- Description of the problem: slide 5.
- Solution with semaphores:



Producer

```
while () {
  T = create_product();
  P(warehouse_mutex);
| put(warehouse, T);
  V(warehouse_mutex);
  V(is_product);
}
```

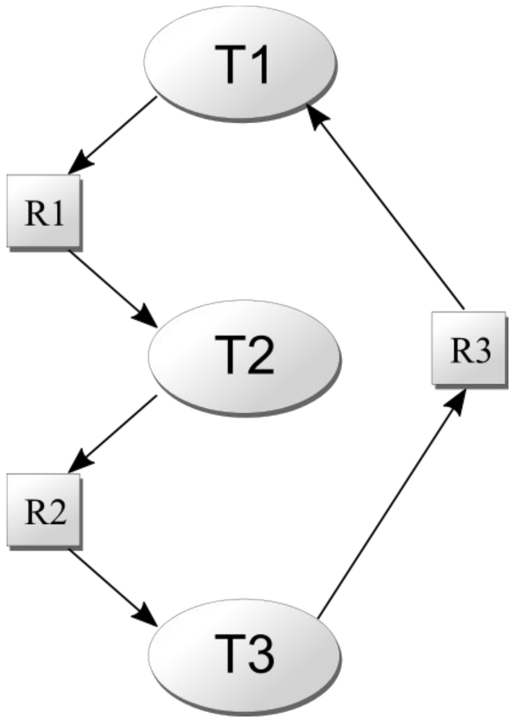
Consumer

```
while () {
  P(is_product);
  P(warehouse_mutex);
| T = get(warehouse);
  V(warehouse_mutex);
  use_product(T)
}
```

A more complex mutual exclusion example

- Description of the problem
  - Three tasks (T1, T2, T3) and three resources (R1, R2, R3)
- Implementation Resource allocation graph

T1
```
P(R3);
P(R1);
V(R1);
V(R3);
```
T2
```
P(R1);
P(R2);
V(R2);
V(R1);
```
T3
```
P(R2);
P(R3);
V(R3);
V(R2);
```



What happens if they start at the same time?

Emergence of a deadlock

- The deadlock is a situation when a set of tasks {T_1, T_2, …,T_N} waits for an event, which can be only generated by a T_i task (1<=i<=N) from the set
  – If this situation is emerged, the task execution stops
- Look at the example above
  – Every execution sequence creates a deadlock?
  – What happens if the tasks are delayed?
  – Can the resource allocation reordered to avoid deadlocks?
- Necessary conditions for deadlocks
  – 1. mutual exclusion
  – 2. busy waiting
  – 3. only voluntarily resource releasing
  – 4. circular waiting
    - T_i waits for a resource locked by T_{i+1} (1<=i<=N) and T_N waits for a resource locked by T_0

Managing deadlocks

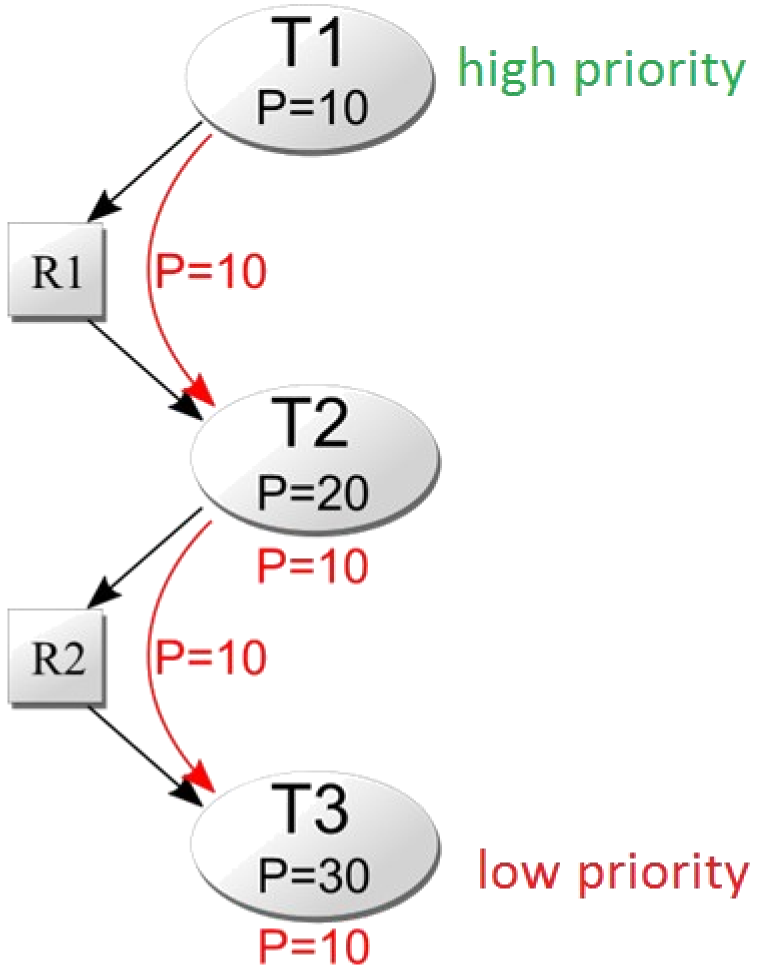
- Ignoring it (ostrich „algorithm”)
  - If the chances of deadlocks are low
  - If a deadlock won’t cause critical errors
  - Balance between the severeness of the error vs. the additional implementation labor
- Detecting and managing it
  - In the case of single resources (N=1):
    - Directed cycles in the resource allocation graph can be detected
    - After detection, the deadlock should be resolved
      - Blocking or delaying tasks, based on other criteria, a precedence should be established between the tasks
      - Blocking tasks may result other failures (inconsistent memory states)
    - Resolving deadlocks is not the task of the OS
- Protect the system against it
  - In the design phase a deadlock-free system should be created
  - At least one of the necessary conditions should be excluded
  - At runtime: only resource reservations are allowed, which won’t cause deadlock
  - Safe state: the system can allocate a resource without any chances of a deadlock
  - Before allocating a resource, the safe state should be checked

Checking the safe state

- At runtime deadlocks can be avoided with safe states
  - The system starting point is a safe state (no allocations)
  - Before every allocation two things should be checked
    - The synchronization condition of the resource allocation
    - The safe state after the allocation
  - If any of these conditions are not met → the task enters into waiting state
- Single instance resources (N^2 complexity graph alg.)
  - The resource allocation graph is prepared
    - The nodes are the tasks and resources
    - The edges shows the allocations and allocation demands
  - Then the graph is extended with a new type of edges
    - Future allocation demands
  - When a task wants to allocate a resource
    - It is checked that if the resource is allocated, no cycles are emerged in the graph
    - The future demands also included
    - If there are cycles → no safe state is guaranteed
- Multiple instance resources
  - Banker's algorithm
  - $M * N^2$ complexity, M is the number of resource types

Further problems of mutual exclusion

- Priority inversion
  - A lower priority task allocated a resource
  - A higher priority task has to wait
  - Solution: priority inheritance
- Starvation by locking
  - One of the tasks is locked a resource continuously
  - Others are blocked
  - Solution: correct the faulty design
- Starvation by waiting
  - In the case of non FIFO waiting
  - A task may stuck in the waiting queue
  - Solution: priority aging
- Performance degradation caused by synchronization
  - Solution: optimistic locking, lock/wait free synchronization



Optimistic resource allocation (locking)

- The pessimistic method prepares for the worst case
  - If the theoretical probability of a conflict is not zero ▭ critical sections defined
- The optimistic method assumes there will be no conflicts
  - It starts the critical section without locking
  - If a conflict is detected: the operations of the critical section is reverted
- The transaction based implementation of the optimistic locking
  - BEGIN: marks the starting state
  - MODIFY: performs the operations
  - VALIDATE: checks the consistency
    - If OK: COMMIT – commits the operations
    - If conflict: ROLLBACK – revert the state to the starting mark
- Pro-s and con-s of the optimistic method
  - If the chances of conflicts are low, the system performance will be improved (no blocking)
  - If the chances are high, the performance will drop, due to the high number of rollbacks
- Implementation examples
  - Transactional memory (HW and SW, e.g.: Intel Haswell TSX and RTM)
  - Database engines, integrated programming structures (C/C++, JAVA)

Lock- and wait-free algorithms

- Lock-free resource management
  - For a data structure to qualify as *lock-free*, if any thread performing an operation on the data structure is suspended at any point during that operation then the other threads accessing the data structure must still be able to complete their tasks.
  - Good resource utilization
- Wait-free resource management
  - A lock-free system which guarantees the operations to be completed regardless of other tasks
- We can avoids deadlocks by using these methods
- It can be proven that synchronization can be implemented with these methods
  - In practice, the overhead may be higher than with waiting
  - It is challenging to create efficient algorithms for the given data structures
    - Waiting queues (ring buffer FIFO) and it's application, chained list, lockless cache
- Challenges of practical implementations
  - Implementing non-atomic operations (e.g. var++; consists of three instructions)
  - The order of the instructions are not guaranteed, the compiler may reorder them, annotations should be given to the compiler to restrict reordering

The basic forms of synchronization

- Mutual exclusion
  - **Critical section**: an instruction sequence of the tasks, which are cannot executed simultaneously
  - Shared resources are protected with this method, so **competitive situations** can be managed
  - Pessimistic method: locks the resource in every case
  - E.g.: while the printer is printing, cannot start a new print

- Rendezvous
  - Specific operations of the tasks should start at the same time
  - **Cooperation** scheme to synchronize the operation of sub-tasks
  - E.g.: Send() and Receive() methods of direct (non buffered) messaging

- Precedence
  - The operations of the tasks should be executed in a predefined order
  - **Cooperation** scheme
  - E.g.: business intelligence SW: the data cleaner operations should run before the data analysis operations

Synchronization between tasks (summary)

- Mandatory synchronization when using shared memory
  - Typically between threads of the same process
    - Supported by programming language structures
    - Usually simple mutexes are used
  - Between processes, usually semaphores are used
- Synchronization of shared resource allocation
  - The chosen method depends on the nature of the resources and tasks
  - Usually counter type semaphores are used
- Protecting the kernel data structure
  - It is necessary because the preemptive and multiprocessor systems
  - For short period locking, spinlocks are used
- Bad designs will lead to faulty operation
  - Deadlocks and starvation may appear, proper design considerations can avoid them
- Synchronization may cause performance degradation
  - Task may blocked, lower CPU utilization
  - Optimistic locking and lock-free methods may be considered