ID1217 Concurrent Programming
Lecture 4

Critical Sections. Locks. Condition Variables.

ROYAL INSTITUTE OF TECHNOLOGY

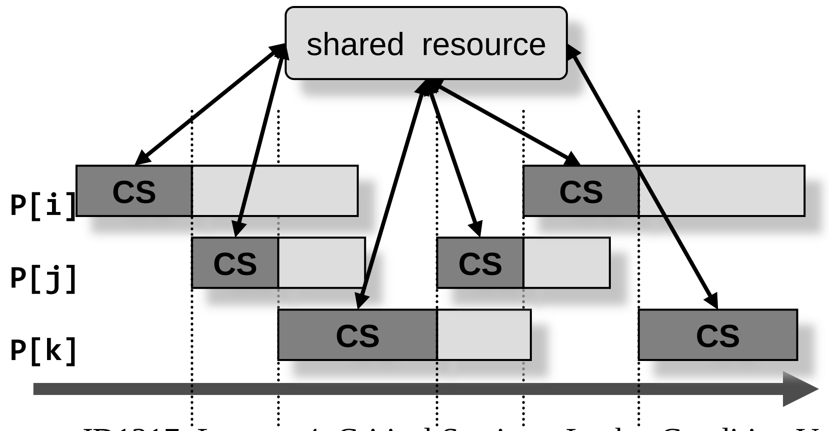
Vladimir Vlassov
KTH/EECS

Outline

- Critical sections. The critical section problem
- Locks
  - Unfair spin locks
  - Fair (queuing) locks
  - Implementing the await statements using locks
- Condition variables
  - Busy waiting versus blocking
  - Implementing the await statements using locks and condition variables

Accessing Shared Data

- Assume, a shared counter **`x`** is initially 0; Two concurrent threads increment **`x`**; Expected result: **`x == 2`**
  - Process histories:
    - P1: …; load value of **`x`** to reg; incr reg; write reg to **`x`**; …
    - P2: …; load value of **`x`** to reg; incr reg; write reg to **`x`**; …
  - Without synchronization, the final result can be **`x == {1,2}`**
- The statements accessing shared variables are **critical sections** that should be executed one at a time, i.e. with mutual exclusion (atomically)



ROYAL INSTITUTE OF TECHNOLOGY

The Critical Section Problem

- **Critical section** is a section of code that accesses a shared resource (e.g. a shared variable) and can be executed by only one process at a time
- **The Critical Section (CS) Problem**: To find a mechanism that guarantees execution of critical sections one at a time, i.e. with mutual exclusion.
  - Arises in many concurrent programs. For example: shared linked lists in OS, database records, shared counters, etc.

ROYAL INSTITUTE OF TECHNOLOGY

Model for the CS Problem

```
process P[i = 0 to n-1] {
   while (true) {
      CSentry:  entry protocol;
      critical section;
      Csexit:  exit protocol;
      non-critical section;
        }
}
```

- The task is to design entry and exit protocols with the following properties.
- Mutual exclusion (correctness)
  - At most one proc at a time is entering, executing and exiting its CS.
- Absence of deadlocks and livelocks (safety)
  - One of the competing proc succeeds to enter.
- Absence of unnecessary delay (efficiency, liveness)
  - A proc is not prevented from entering if others do not compete
- Eventual entry (fairness, liveness)
  - Every proc attempting to enter its CS will eventually enter

Impossibility Result

- Assume ***N*** processes
- Result 1: At least ***N*** MRSW (Multi-Reader/Single-Writer) registers are needed to solve deadlock-free mutual exclusion.
- Result 2: At least ***N*** MRMW (Multi-Reader/Multi-Writer) registers are needed to solve deadlock-free mutual exclusion.
- **Thus, to solve the CS problem for *N* processes using only reads and writes requires O(*N*) memory locations**

ROYAL INSTITUTE OF TECHNOLOGY

Solving the CS Problem Using Locks

- Observation: there are two states:
  1. Some process in its critical section;
  2. No process in CS.
- The states can be represented by a Boolean variable a.k.a. a basic lock
- **Locks** provide a solution for the CS problem
  - "Lock the resource" in CSentry
  - "Unlock the resource" in CSexit

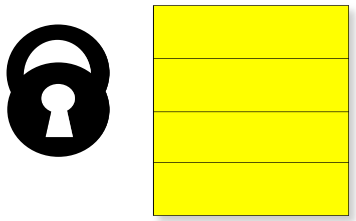
What Locks Protect

- Locks protect shared variables in critical sections.
  - One can say ”protects critical sections”
- A lock is associated with shared variable(s) it protects.
- For example (using the Pthread library):

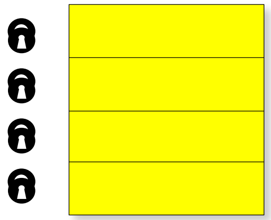
```
int counter; // global counter
pthread_mutex_t lock; // lock that protects the counter
. . .
pthread_mutex_lock(&lock);
counter++; // critical section w.r.t. counter
pthread_mutex_unlock(&lock)
```

How Many Locks to Use

- One single lock can be used to protect several shared variables.
- Extreme case: **One lock for ALL shared variables**
  - Easy to use, but inefficient, decreases parallelism



- Extreme case: **Each shared variable has its own lock**
  - Large degree of parallelism can be achieved, but may cause large synchronization overhead (memory and time)
  - Deadlock prone; order of locking is important



- A better solution is in between: Should **tradeoff** the number of locks (synchronization overhead) for the number of variables protected by a single lock (available parallelism).

Types of Locks

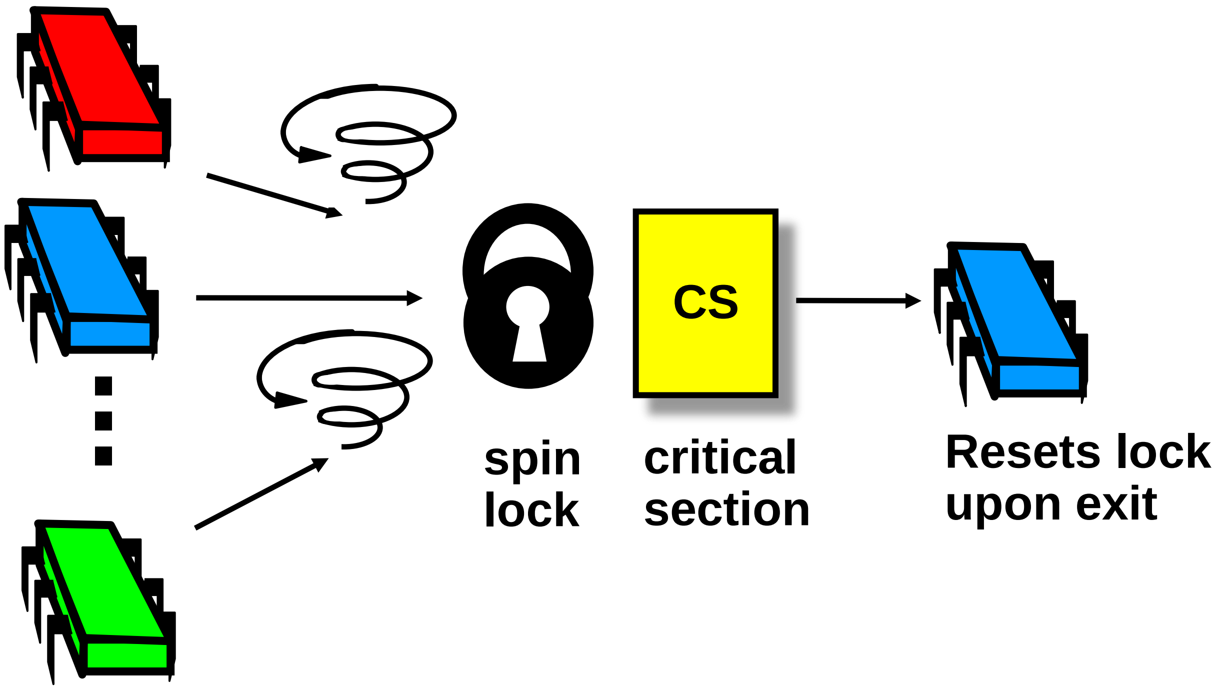
our focus

- What should you do if you can't get a lock?
- Keep trying – **Spin locks**
  - ***Spinning*** or ***busy-waiting***: Repeatedly testing the lock
  - Good if delays are short
  - Spinning makes sense only on multiprocessors
  - Need time-slicing on uni-processors
  - Examples: T&S lock, counter lock, filter lock, bakery lock
- Give up the processor – **Blocking**
  - Good if delays are long as blocking (context switch) is expensive
  - Always good on uniprocessor
  - **Java's built-in synchronization is blocking**
- Many operating systems mix both strategies, spinning for a short time and then blocking.

Types of Spin Locks

- **Unfair locks** – unfair but efficient
  - Short latency and low memory demand
  - Poor fairness, may cause starvation
    - The same process may acquire the lock several times consecutively, while others are waiting (spinning) for the lock
  - Good in case of low contention, i.e. for a small number of competing processes
  - Examples: Test&set lock, test-test&set lock, test&set lock with backoff
- **Queuing (FIFO) locks** – fair but more expensive
  - Longer latency, more memory – the price for fairness
  - Examples: tie-breaker lock (a.k.a. filter lock), ticket lock, bakery lock

Basic Spin-Lock



KTH VETENSKAP OCH KONST
ROYAL INSTITUTE OF TECHNOLOGY

A Basic (Unfair) Spin-Lock

- A basic (unfair) spin-lock is a Boolean variable that indicates whether or not one of the processes is in its critical section:

`lock` =
- `true` – some process is in its CS (the lock is "locked")
- `false` – no process in CS (the lock is "unlocked")

```
boolean lock = false;

procedure Lock( var boolean lock ) {
   < await (!lock) lock = true; >
}

procedure Unlock( var boolean lock ) {
   lock = false;
}
```

ROYAL INSTITUTE
OF TECHNOLOGY

Critical Sections Using a Basic Spin-Lock

```
boolean lock = false;
process P[i = 0 to n-1] {
   while (true) {
      Lock(&lock);    // CSentry
       critical section;
       Unlock(&lock); // CSexit
       non-critical section;
    }
  }

procedure Lock( var boolean lock ) {
   <await (!lock) lock = true;>
  }

procedure Unlock( var bool lock ) {
   lock = false;
  }
```

Implementing a Basic Spin-Lock Using a <Read-Modify-Write> Operation

- We have a problem! Lock requires atomicity is its own implementation:

```
<await (!lock) lock = true;>
```

- HW support – a special atomic memory instruction **RMW** **<Read-Modify-Write>**
  - Examples: test&set, swap, fetch&increment
  - For example, using t&s (in pseudo-code):
    ```
    while (test&set(lock)) continue;
    ```
  - **Allows to overcome the $\Omega(N)$ lower bound on memory locations needed to solve the CS problem using only reads and writes (see Slide 6)**

- Unlock is implemented using the ordinary store operation

```
lock = false;
```

ROYAL INSTITUTE OF TECHNOLOGY

Test-and-Set Spin-Lock

- Using Test&Set instruction (t&s)

```
lock:   t&s register, location        // try to lock the lock
        bnz lock                      // if not 0, try again
        ret                           // return to caller

unlock: st location, #0         // write 0 to the lock
        ret                     // return to caller
```

Here: **bnz**: branch-non-zero – if the result of the previous instruction is nonzero, then goto to a specified label;

ret: return from the function

Drawback of the Simple Test-and-Set Lock

- Causes high memory contention while waiting for the lock:
  - t&s is treated as a write operation – invalidates cached copies of the lock
  - Unsuccessful t&s generate memory accesses (bus traffic)
  - Also wasting CPU time because of spinning (busy waiting)

Enhancements to the Simple Test-and-Set Lock

- SW solutions:
  - Test-and-Set lock with (exponential) backoff
  - Test-Test-and-Set lock
- Improved HW primitives:
  - Instructions: Load-Locked (LL) and Store-Conditional (SC)

Test-and-Set Lock with Backoff

```
lock:   ld     reg2, #1               // initial delay
retry:  t&s    reg1, location         // try to lock
        beqz   done                   // if 0, goto to done
        sll    reg2, reg2, 1          // increase delay by 2
        pause  reg2                   // delay by value in reg2
        j      retry                  // jump to retry
done:   ret                           // return control to caller
```

- Back off (pause) after unsuccessful t&s (attempt to lock)
- Allows to reduce frequency of issuing test&sets while waiting.
- Don't back off too much or will be backed off when lock becomes free
- Exponential backoff works well empirically: i-th time = $k*c^i$ (e.g. 2^i)
  - Here: **beqz**: branch-equal-zero – if the result of the previous instruction is zero, then jump to a specified label;
    sll: shift-logical-left (equivalent to multiply by 2)
    ret: return from the function
    j: jump to a specified label

ROYAL INSTITUTE
OF TECHNOLOGY

Test-Test-and-Set Lock

- Idea: Keep testing with ordinary load. When value changes (to 0), do test&set.

```
lock:   ld     reg1, location         // load of lock
        bnz    lock                   // if not 0, spin
        t&s    reg1, location         // try to lock
        bnz    lock                   // if not 0, retry
        ret                           // return control to caller
```

- Slightly higher latency, much less memory contention

Performance of Test-and-Set Locks

- Uncontended latency
  - Low if repeatedly accessed by same processor; independent of *n*
- Traffic
  - Lots if many processors compete; poor scaling with *n*
  - Each t&s generates invalidations, and all rush out again to t&s
- Storage: Very small (single variable); independent of *n*
- Fairness: Poor, can cause starvation

- Test&set with backoff similar, but less traffic
- Test-and-test&set: slightly higher latency, much less traffic

ROYAL INSTITUTE OF TECHNOLOGY

Queuing Locks

- Unfair spin-locks are efficient (low latency and memory demand) but unfair
  - When a lock becomes free, spinning processes rush to grab the lock in an arbitrary order; one succeeds, others fail and spin again.
  - The same process can grab the lock again.
- **Queuing spin-locks** provide fair solution to the CS problem
  - Waiting processes are queued on the lock (at the CS entrance);
  - Released lock is passed to the proc in the head of the queue;
  - Examples : tie-breaker (a.k.a. filter), ticket, bakery locks.
- Consider the ticket and the bakery locks

The Ticket Algorithm

- Works like a waiting line at a post office or a bank.
- Two shared counters per lock:
  - **`number`** to be “drawn” by one proc at a time;
  - **`next`** to indicate which proc can enter its critical section.
- CS enter (lock the lock):
  - Read a number from **`number`** and increment **`number`**; wait until **`next`** is equal to its number drawn, then enter CS.
- CS exit (unlock the lock):
  - Increment **`next`** that allows the next waiting proc (if any) to enter its CS.
- Global invariant:

TICKET: $\texttt{next} > 0 \land$
$(\forall\ i{:}\ 1 \le i \le n{:}\ (P[i]\ \text{in its critical section}) \Rightarrow (\texttt{turn}[i] == \texttt{next}) \land$
$(\texttt{turn}[i] > 0) \Rightarrow (\forall\ j{:}\ 1 \le j \le n,\ j \ne i{:}\ \texttt{turn}[i] \ne \texttt{turn}[j]))$

Critical Sections Using the Ticket Algorithm

```
int number = 1, next = 1, turn[0:n-1] = ([n]0);
process P[i = 0 to n-1] {
   while (true) {
       < turn[i] = number++; >  // “draw” a number
      while (turn[i] != next); // <await (turn[i]==next)>
      critical section;
      next = next + 1;           // Csexit
      non-critical section;
   }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Implementing the Ticket Lock

- `turn[i]` can be a local variable `turn` in the Lock procedure.
- The ticket lock can be implemented as a structure with fields `number` and `next`.

```
typedef struct { int number; int next} ticketlock;
procedure Lock( var ticketlock lock ) {
       int turn;
       <turn = lock.number++;>      // draw a number
  while (turn != lock.next) ;       // wait for my turn
}
procedure Unlock( var ticketlock lock ) {
  lock.next++;
}
```

- The ticket lock requires a special atomic memory instruction for number drawing – ***fetch&increment***

The Bakery Algorithm

- The Ticket algorithm is fair if fetch&add is available
  - Otherwise requires mutual exclusion for number drawing that can be unfair.
- The ***Bakery algorithm*** works like a line in a bakery without a number-drawing machine – a proc looks around and takes a number one larger then any other.
  - Requires an integer array **`turn[n]`** per lock
  - Does not need a special instruction
  - Global invariant of the Bakery algorithm:

***BAKARY*: (∀ i: 1 ≤ i ≤ n: (P[i] in its critical section) ⇒(turn[i] > 0) ∧**
(∀ j: 1 ≤ j ≤ n, j ≠ i: turn[j] == 0 ∨ turn[i] < turn[j])

Implementation of The Bakery Algorithm

```
int turn[1:n] = ([n]0);
process CS[i = 1 to n] {
   while (true) {
      turn[i] = 1;
      turn[i] = max(turn[1:n]) + 1;
      for [j = 1 to n such that j != i]
         while (turn[j] != 0 and
                 (turn[i],i) > (turn[j]),j) skip;
      critical section;
      turn[i] = 0;
      non-critical section;
   }
} // Here (a,b) > (c,d) if a > c or (a == c and b > d)
```

ROYAL INSTITUTE OF TECHNOLOGY

Some Tips on How to Program CS

- Check for the AMO property
  - A section may appear to be executed atomically – does need explicit atomicity
- Try to avoid critical sections by replicating (or splitting) shared data
- Try to make CS as short as possible.
- Try to use different locks for different shared data.
  - Tradeoff between the size of CS and the number of locks.
- Which lock to use?
  - The simple t&s lock works quite well in the case of low contention (i.e. a small number of competing processes).
  - An enhanced spin lock (t-t&s, t&s with backoff) should be used in the case of high contention (a large number of competing processes).
  - Queuing locks, e.g. ticket, bakery, should be use if fairness is an issue.

Example: Find the Maximum (revisited)

- The parallel program outline using **`await`** for synchronization:

```
int a[n];
int m = 0;
…
co [i = 0 to n-1]
   if (a[i] > m)
      〈 if (a[i] > m) m = a[i]; 〉 oc
```

- The program outline using a spin lock:

```
int m = 0;
boolean lock = false;
…
co [i = 0 to n-1]
   if (a[i] > m) {
     Lock(&lock); if (a[i]) > m) m = a[i]; Unlock(&lock);
   }
oc
```

ROYAL INSTITUTE OF TECHNOLOGY

Implementing Await Statements

- Observation: Mutual exclusion guarantees atomicity
- Any solution to the CS problem, CSenter and CSexit, can be used to implement the await statements:

```
< S; >
< await(B); >
< await(B) S; >
```

  - The statement can be treated as a critical section for a set of shared variables referenced in B and S.
  - Use a lock to protect the shared variables referenced in B and S.

Implementing Await Statements (cont' d)

Atomic action	Implementation

`< S;>`	`Lock(&lock); // CSenter`
	`S;`
	`Unlock(&lock); // CSexit`

`<await(B);>`	`Lock(&lock);`
	`while (!B) {Unlock(&lock); Delay; Lock(&lock); }`
	`Unlock(&lock);`

`<await(B) S;>`	`Lock(&lock);`
	`while (!B) {Unlock(&lock); Delay; Lock(&lock); }`
	`S;`
	`Unlock(&lock);`

- To avoid a deadlock, a proc awaiting **B**, must repeatedly exit and enter its CS to allow other processes to alter variables in **B**

Waiting for Synchronization: Busy Waiting versus Blocking

- **<await (B) ... >**
- **Busy waiting**
  - A proc spins on a condition
  - Requires time slicing on a shared CPU to avoid deadlock
  - Shortcomings of busy waiting
    - Inefficiency: wasting CPU time
    - Complexity
      - Difficult to specify conditions for waiting
      - Difficult to reuse synch variables, e.g. flags
- **Blocking**
  - A proc is blocked on synchronization to be resumed by another proc
  - Controlled by the OS kernel or the runtime system
  - Shortcomings of blocking
    - Long context switch time
  - Blocking synchronization mechanisms:
    - Condition variables
    - Semaphores – to be considered later

Condition Variables

- **Condition variable** is an opaque object that represents a queue of suspended processes waiting to be resumed (when the queue is signaled)
  - A mechanism to wait and to signal conditions in condition synchronization
  - Allows to suspend and to resume processes holding locks
- Operations:
  - Blocking wait
  - Signal to resume
- Locks and condition variables – synchronization mechanisms in Pthreads
  - Locks – for mutual exclusion synchronization
  - Condition variables – for condition synchronization
- Implicit condition variables are used in Java monitors

Operations on Condition Variables

- Declaration: `cond cv;`
- Operations:

`wait(cv, lock)`	Release the lock and wait at end queue
`wait(cv, rank, lock)`	Release lock and wait in order of increasing value of rank
`signal(cv)`	Awaken process at front of queue then continue
`signal_all(cv)`	Awaken all processes on queue then continue
`empty(cv)`	True if queue is empty; false otherwise
`minkrank(cv)`	Value of rank of proc at front of wait queue

  – When used in monitors, `lock` is not specified, the implicit monitor lock is assumed

ROYAL INSTITUTE OF TECHNOLOGY

Signaling Disciplines

- ***Signal and Continue (SC)*** – the signaling process continues, the resumed process reacquires the lock
- ***Signal and Wait (SW)*** – the signaling process passes the lock to the resumed process and reacquires the lock
- ***Signal and Urgent Wait (SUW)*** – as SW but the signaling process is placed to the head of the lock queue

Implementing Await Statements using Locks and Condition Variables

```
boolean lock = false; cond cv;
```

Atomic action	Implementation	Signal condition

`< S;>`	`Lock(&lock);`
	`S;`
	`Unlock(&lock);`	

`<await(B);>`	`Lock(&lock);`
	`while (!B) wait(cv, &lock);`
	`Unlock(&lock);`
		`Lock(&lock);`
		make `B` `true`;
		`signal(cv);`
		`Unlock(&lock);`

`<await(B) S;>`	`Lock(&lock);`
	`while (!B) wait(cv, &lock);`
	`S; Unlock(&lock);`	

- See example in the lecture on pthreads

ROYAL INSTITUTE OF TECHNOLOGY

Exercise: The Water Molecule Problem

- Suppose hydrogen (H) and oxygen (O) atoms are bouncing around in space trying to group together into water molecules. This requires that two hydrogen atoms and one oxygen atom synchronize with each other.
- Let the H and O atoms be simulated by concurrent processes (threads). Each H atom calls a procedure Hready when it wants to combine into a water molecule. Each O atom calls another procedure Oready when it wants to combine.
- Develop the two procedures, Hready and Oready, using locks and condition variables for synchronization.
  - An H atom has to delay in Hready until another H atom has also called Hready and one O atom has called Oready. Then one of the processes should call a procedure makeWater. After makeWater return, all three processes should return from their calls of Hready and Oready.

A Possible Solution

```
// shared variables
// number of atoms ready
int nh = 0, no = 0;
// lock to protect counters
bool  entry = false;
// conditions: waiting points
cond H, O;
procedure Hready() {
   lock(entry);
   if (nh > 0 and no > 0) {
       makeWater();
       nh--;
       no--;
       signal(H);
       signal(O);
   } else {
       nh++;
       wait(H, entry);
   }
   unlock(entry);
}
```

```
procedure makeWater() {
   this procedure makes a water
   molecule, for example just
   counts the molecules
}


procedure Oready() {
   lock(entry);
   if (nh > 1) {
       makeWater();
       nh = nh - 2;
       signal(H);
       signal(H);
   } else {
       no++;
       wait(O, entry);
   }
   unlock(entry);
}
```