



T-106.420

Concurrent Programming

Liveness, locks, critical sections



interesting properties

- true for every possible history
- *safety* property
 - the program never enters a *bad* state
 - generalisation of *partial correctness*:
 - if the program terminates, its final state provides the correct answer
- *liveness* property
 - the program eventually enters a *good* state
 - generalization of *termination*:
 - every loop & procedure call terminates, i.e. every history is finite
- *total correctness*
 - partial correctness + termination
 - the program always terminates and with a correct answer
 - classical property of sequential programs



examples properties

- safety property
 - mutual exclusion
 - bad state: more than one process in their respective critical sections concurrently
 - absence of *deadlock*
 - bad state: two or more processes waiting for each other to establish a condition which will not become true
- liveness property (affected by scheduling policies)
 - a process will eventually enter a critical section
 - service request will be eventually honored
 - messages will eventually reach their destination



proving safety properties

- any action of a program is based on program state
- a safety property asserts some soundness property of the program state: "program is working ok"
- *mutual exclusion* property fails to hold, if there is a state when two(or more) processes are in their respective critical sections
- *deadlock*: two or more processes wait for conditions to be established by the other(s), and which are currently false – and consequently, will stay false forever.



proving safety properties (2)

- Method 1
 - let BAD characterize a bad program state
 - program is safe if BAD is false in every state in every possible history
 - show that BAD is not true initially, and can never be entered by any action
- Method 2
 - show that program is always in a GOOD state, where GOOD is equivalent to NOT BAD
 - specify BAD
 - negate BAD to yield GOOD
 - ensure that GOOD is a *global invariant* – a predicate that is true in every state



proving safety properties (3)

- Method 3: *exclusion of configurations*

```
co  # process 1
    ...; {pre(S1)} S1; ...
    {x==1 v x==3}

// # process 2
    ...; {pre(S2)} S2; ...
oc
```

- $\text{pre}(S1) \wedge \text{pre}(S2) == \text{false}$
the two processes can not be at these statements at the same time; *false* characterizes *no* program state
- S2 can not interfere with $\text{pre}(S1)$ and vice versa.



scheduling policies and fairness

- liveness properties depend on *fairness* of scheduling policy
- fairness is a *scheduling guarantee* that a process proceeds when it logically can, i.e. it's not waiting for a condition to hold
- *eligible atomic action* is the next executable action in a process
- several processes \Rightarrow normally several *eligible atomic actions*
- *unconditional atomic action* is one that does not have a delay on a condition; i.e. whenever it's next, it's also eligible
- *scheduling policy* determines which one of the eligible actions will be executed next



scheduling policies and fairness (2)

unconditional fairness

- example:

```
bool continue=true;
co while (continue); #process 1
// continue=false; #process 2
oc
```
- assume a one processor system and a scheduling policy that assigns the processor to a process until it *terminates* or *awaits* for a condition to get *true*
- if process #1 is executed first the program will never terminate, if process #2 gets a chance the program will terminate immediately after it
- scheduling policy is *unconditionally fair (impartial)* if every eligible unconditional atomic action is *executed eventually*, i.e.
 - every infinite history has an infinite number actions from every process
 - every process progresses with a “positive speed”
- examples of unconditionally fair scheduling policy for the above program
 - on a single processor -> e.g. round-robin
 - on a multiprocessor -> fair parallel execution were no processor can monopolize the access to the shared memory i.e. variable `continue`.



scheduling policies and fairness (3)

weak fairness

- assume a program contains a *conditional atomic action*
 - await statement with Boolean condition B
 - a conditional atomic action is eligible iff it's the next one, and B is true
- a scheduling policy is *weakly fair* if
 1. it is *unconditionally fair* and
 2. *every* conditional atomic action *a* that is *continuously eligible* i.e. its condition B becomes true and remains true until *a* gets selected, is executed eventually
- example: `<await (B) S; >` If it's next in its process and B is true and remains true at least until S has been executed
- round-robin and time-slicing are such scheduling policies, because any delayed process will eventually see that its delay condition as true



scheduling policies and fairness (4)

strong fairness

- weak fairness is not sufficient to ensure that an eligible `< await >` statement (i.e. next executable action in its process) eventually executes when its condition B alternates between *true* and *false*
- a scheduling policy is *strongly fair* if
 1. it is *unconditionally fair*
 2. *every* conditional atomic action that is eligible is executed eventually, assuming that its condition is infinitely often true
- example:

```
bool continue=true, try=false;
co while (continue){try=true; try=false;}
// <await (try)continue=false; >
oc
```
- other, more deterministic policies: no-infinite-overtaking, fifo...



deadlock, livelock and starvation

- *deadlock*: Two or more processes wait for conditions to be established by each other, i.e. they form a "waiting circle" and will get stuck independent of any scheduling policy
- *livelock*: a possible trace for several processes, such that no process progresses in bounded time out from a waiting loop
- *starvation*: a trace where other processes may progress and overtake one process indefinitely often, ie. system is not *fair*.
- both starvation and livelock are possible, although in practice improbable system behaviours caused by the combination of weak synchronisation algorithm and a weakly fair scheduling policy. They are not acceptable in hard real-time systems.
- we will get examples of these later.



locks

critical sections



the critical section problem

- classic problem: how to implement the $\langle \dots \rangle$?
 - length of $\langle \dots \rangle$ is arbitrary
- the same in more concrete terms:
 - n processes (of the form below) repeatedly execute a critical & a non-critical code section:

```
process p[i=1 to n]{  
  while (true) {  
    entry protocol;           #  $\langle$   
    critical section;        #   ...  
    exit protocol;           #    $\rangle$   
    noncritical section;  
  }  
}
```



critical section problem (2)

- *basic assumption*: each process entering its critical section will eventually exit from it, ie. a process may not terminate in its critical section
- *safe mutual exclusion*: at most one process at a time is in its critical section
- *absence of deadlock*: if two or more processes are trying to enter their critical sections, at least one will succeed
- *absence of unnecessary delay*: if a process is alone trying to enter an unoccupied critical section (ie. there is no *race condition*) it may do so without any extra delay
- *fairness*: process attempting to enter its critical section will eventually succeed assuming that the scheduler is weakly fair



critical section – solutions (1)

- trivial solution
 - Put the whole `critical section` inside an unconditional `< await >` statement: `<critical section>`
 - Mutual exclusion solved because of the semantics of `< await >`
 - Absence of deadlock (livelock), absence of unnecessary delay, eventual entry are solved if scheduler is *unconditionally fair*
 - But how are those *angle brackets* `< >` implemented? This was not a solution, but the problem!
- we are asked to solve the problem using the "normal computer HW", ie. with traditional machine instructions, like `LOAD` and `STORE`.



critical section (2): coarse-grained

```
bool in1=false, in2=false;
## MUTEX: ¬(in1 ∧ in2) -- global invariant
process p1 {
  while (true) {
    ⟨await (!in2) in1=true; ⟩ /* entry*/
    critical section;
    in1=false;                /* exit */
    noncritical section;
  }
}
process p2 {
  while (true) {
    ⟨await (!in1) in2=true; ⟩ /* entry*/
    critical section;
    in2=false;                /* exit */
    noncritical section;
  }
}
```

So, how is this better than just: ⟨critical section⟩? No way.



critical section(3): spin locks

- generalization of previous solution would use n variables for n processes
- actually 2 interesting states
 - some process in its critical section
 - no process in its critical section
- modify previous example
 - `lock == (in1 v in2)`
 - `lock` used in entry and exit protocols



critical section (4): spin locks

- previous solution generalized for n processes

```
bool lock=false;
process p1 {
  while (true) {
    <await (!lock) lock=true; > /* entry*/
    critical section;
    lock=false;                /* exit */
    noncritical section;
  }
}
process p2 {
  while (true) {
    <await (!lock) lock=true; > /* entry*/
    critical section;
    lock=false;                /* exit */
    noncritical section;
  }
}
```



test & set machine instruction (TS)

- the previous were not real solution, because `< await >` was used
- so, how to implement: `< await (!lock) lock=true; >` ?
- most machines have a *Test-and-Set* (TS) machine instruction, which returns the old value *of a shared variable and assigns a constant value to it* in one atomic HW action:

```
bool TS (bool var) {           /* var is shared and tmp is just
                                temporary and local */
    bool tmp = var;           /* save old value to be returned */
    var = true;              /* set shared variable */
    return tmp;              /* return old value */
}
```

- check out your machine instruction set for instructions satisfying this property



critical section(5): spin locks with TS

```
bool lock=false;           /* shared lock */  
process p[i=1 to n] {  
  while (true) {  
    while (TS(lock)) skip; /* entry protocol */  
    critical section;      /* {lock=true } */  
    lock=false;         /* exit protocol */  
    noncritical section;  
  }  
}
```

- A common paradigm for exit protocols in spin-lock solutions: simply reset the shared variables to their initial values



test & (test & set) (TTS)

- TS(lock) leads to poor performance because every delayed process *continuously* refers to lock.
=> lock is a *hot spot* causing memory contention.
- Even worse: lock is written by TS at every execution!
=> in a shared memory multiprocessors caches on other processors also need to be invalidated or altered continuously.



critical section(6): spin locks with TTS

```
bool lock=false;                                /* shared lock */
process p[i=1 to n] {
  while (true) {                                  /* entry protocol */
    while (lock) skip;                            /* spin while lock set */
    while (TS(lock)) {                          /* try to grab the lock */
      while (lock) skip;                          /* spin again if fail */
    }
    critical section;
    lock=false;                                  /* exit protocol */
    noncritical section;
  }
}
```

- This causes much less memory contention, because `(TS(lock))` is executed only when lock was found to be false just before it in the termination of the `while (lock) skip;` loop.



implementing `<await B S>` statements

```
CSenter;  
while (!B) { CSexit; Delay; CSenter };  
S;  
CSexit;
```



critical section – fair solutions (1)

- fair spin-lock solutions require a strongly fair scheduler
- the solutions we have seen this far couldn't guarantee fairness.
- solutions that require only weakly fair scheduling policies:
 - tie-breaker algorithm
 - ticket algorithm
 - bakery algorithm



tie-breaker algorithm for two processes

- problem: if each process tries to enter its critical section (CS) there is no control over who will succeed; one process can execute over and over again

```
bool in1=false, in2=false;
## MUTEX: ¬(in1 ∧ in2) -- global invariant
process p1 {
while (true) {
    <await (!in2) in1=true; >           /* entry */
    critical section;
    in1=false;                          /* exit */
    noncritical section;
}
}
...
```



tie-breaker algorithm (3)

- How to implement this : `<await (!in2) in1=true; > /* entry*/`
 - try by checking out first if the other process is around:

```
while (in2) skip;
in1=true;           /* and similarly for p2 */
critical section;...
```

=> problem: the two statements are not atomic => *mutual exclusion not ensured*

- reverse the order of the statements “while (in2) skip” and “in1=true”:

```
in1=true;
while (in2) skip;
critical section;...
```

=> problem: mutual exclusion is ensured, but *a deadlock can result*: if in1 and in2 are both true, neither loop will terminate.



tie-breaker algorithm(4):coarse-grained

Break the tie by adding a variable **last** telling “which one was the last to enter the wait for CS”:

```
bool in1=false, in2=false;
int last = 1;
process p1 {
    while (true) {
        in1=true; last=1;                /* entry protocol */
        <await (!in2 or last ==2); > {-in2 or last =2}
        critical section;
        in1=false;                       /* exit */
        noncritical section;
    }
}
process p2 {
    while (true) {
        in2=true; last=2;                /* entry protocol */
        <await (!in1 or last ==1); >{-in1 or last =1}
        critical section;
        in2=false;                       /* exit */
        noncritical section;
    }
}
```

Note: The conditions to hold when p1 and p2 are able to enter their CS are: $\{-in2 \text{ or } last = 2\}$ and $\{-in1 \text{ or } last = 1\}$. Because they can not be true at the same time, the solution is safe.



fine-grained tie-breaker algorithm (5)

- Wait in p1: `<await (!in2 or last ==2);> {¬in2 or last =2}` can be replaced with busy-wait loop if the condition would satisfy the At-Most-Once property:

```
while (in2 and last ==1) skip; {¬in2 or last =2} ?
```

- But it does not, because it refers to `in2` and `last` altered by the other process p2.
- However, the following weaker proof outline of p1 is not interfered by p2:

```
while (in2 and last ==1) skip;  
{¬in2 or last =2 or “in2=true; {p2 is here}last=2;” }  
critical section;
```

Explanation: Assume that p1 finds that `{¬in2}` holds, and drops out of the while-loop. If `in2` is changed to true by p2 after this, it has already done `last=2;` or is just about to do it, so the second part: `{last =2}` will become true. So the whole condition for p1: `{¬in2 or last =2}` will be true before p2 may come to its while loop:

```
while(in1 and last == 2) skip;
```

and will find its wait condition to hold and starts waiting.

- Conclusion: The parts of the while condition can be tested independently.



fine-grained tie-breaker algorithm (8)

also know as Peterson's algorithm

```
bool in1=false, in2=false;
int turn = 1
process p1 {
  while (true) {
    in1=true; turn=2;          /* entry protocol */
    while (in2 and turn==2) skip;
    critical section;
    in1=false;                /* exit */
    noncritical section;
  }
}
process p2 {
  while (true) {
    in2=true; turn=1; /* entry protocol */
    while (in1 and turn==1) skip;
    critical section;
    in2=false;          /* exit */
    noncritical section;
  }
}
```

Warning: There is an error in the book in Figures 3.5 and 3.6!!!



tie-breaker algorithm (8)

fine-grained

- Warning: Figures 3.5 and 3.6 in the book contain a serious error ! Can you find it ?
- Is tie-breaker correct if the order of `in1=true` and `turn=2` is interchanged ?
- Prove about the tie-breaker the properties: mutual exclusion, no deadlock, liveness.
 - Hint: Prove a predicate $H1$ covering the state of CS1 between `in1=true` and `in1=false`, which is not interfered by CS2 and vice versa. Work from $H1 \wedge H1$ towards the properties.
 - To formulate H use "a program counter" to specify part of the process state
- For n processes: generalise from the 2 process solution using following recursion:
 - $SS(1) = CS$
 - $SS(n) = \text{entry}(n); SS(n-1); \text{exit}(n)$
 - where $\text{entry}(n)$ and $\text{exit}(n)$ allow at most $n-1$ concurrent processes in $SS(n-1)$
 - $\text{entry}(2)$ and $\text{exit}(2)$ are from the 2-process solution described



ticket algorithm

- n-process tie-breaker algorithm is complex!
- “queuing numbers” to order waiting processes
- stores, banks employ the method:
 - upon entry a customer draws a queuing **number** from a paper reel (larger than what any previous customer has)
 - over the counter a display indicates which customer is being served **next**



ticket algorithm (2): basic version

```
int number=1, next=1, turn[1:n]=([n] 0)
## predicate TICKET is a global invariant int last = 1
process p[i=1 to n] {
  while (true) {
    ⟨turn[i]=number; number= number+1; ⟩ /* draw the ticket*/
    ⟨await (turn[i]==next); ⟩          /* wait on the counter */
    critical section;
    ⟨next = next +1; ⟩                 /* increment the display */
    noncritical section;
  }
}
```



ticket algorithm (3): invariant

- customers get unique numbers: $\langle \text{turn}[i] = \text{number}; \text{number} = \text{number} + 1; \rangle$
- **number** and **next** are read & incremented as atomic actions
- global invariant, TICKET:

$$\{ (\text{next} > 0) \wedge \\ (\forall i: 1 \leq i \leq n: \\ (\text{p}[i] \text{ in its critical section}) \Rightarrow \\ (\text{turn}[i] == \text{next}) \wedge (\text{turn}[i] > 0) \Rightarrow (\forall j: 1 \leq j \leq n, j \neq i: \text{turn}[i] \neq \text{turn}[j]))) \}$$

- last line : nonzero values of **turn** are unique
- **number** & **next** are unbounded (increment can cause arithmetic overflow; but unlikely), and updated by everybody



ticket algorithm (4): coarse-grained -> fine-grained

- algorithm has 3 coarse-grained atomic actions
 - `<turn[i]=number; number= number+1; >`
 - `<await (turn[i]==next); >`
 - `<next = next +1; >`
- `<turn[i]=number; number= number+1; >`
some machines can do this, e.g. **Fetch-and-Add** instruction:
`FA(var,incr): <int tmp=var; var=var+incr;return(tmp); >`
- `<await (turn[i]==next); >`
only one shared variable referenced => easy to implement as busy-waiting loop
`while (turn[i] !=next) skip;`
- `<next = next +1; >`
implementable as usual *load* and *store* instructions,
because at most one process executes the exit protocol at a time



ticket algorithm (5) fine-grained: draw the ticket with FA

```
int number=1, next=1, turn[1:n]=([n] 0)
process p[i=1 to n] {
  while (true) {
    turn[i]=FA(number, 1); /* entry protocol */
    while (turn[i] !=next) skip;
    critical section;
    next = next +1;          /* exit protocol */
    noncritical section;
  }
}
```



ticket algorithm (6): draw the ticket” with a lower level CS

```
int number=1, next=1, turn[1:n]=[n] 0)
process p[i=1 to n] {
    while (true) {
        CSenter;          /* entry protocol */
        turn[i]=number;
        number=number+1;

        CSexit;

        while (turn[i]!=next) skip;
        critical section;
        next = next +1;      /* exit protocol */
        noncritical section;
    }}
```

- Challenges left:
- CSenter/CSexit can be implemented using Test-and-Set - with usual problems related to HW-mechanisms, i.e. fairness, memory contention...
- memory contention around the shared variable `next`.



bakery algorithm

- Algorithm in case the machine has no Fetch-and-Add instruction or we want avoid number drawing

```
⟨turn[i]=number;  number= number+1; ⟩
```

- Customer with smallest number gets service first, but customers check with each other, not with a central **next** counter



bakery algorithm (2): coarse-grained

```
int turn[1:n]=([n] 0)
## predicate BAKERY is a global invariant
process p[i=1 to n] {
  while (true) {
    ⟨turn[i]=max(turn[1:n]) + 1;⟩
    for [j=1 to n st j != i]
      ⟨await (turn[j]==0 or turn[i] < turn[j]); ⟩
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```



bakery algorithm (3): invariant

- basic idea: The process in its CS has the smallest positive turn value and all the positive turn values are unique"
- this is needs to be atomic for nonzero values of turn to be unique:
 $\langle \text{turn}[i] = \max(\text{turn}[1:n]) + 1; \rangle$
- global invariant, BAKERY:
 $\{\forall i: 1 \leq i \leq n: (\text{p}[i] \text{ in its critical section })$
 $\Rightarrow (\text{turn}[i] > 0) \wedge (\forall j: 1 \leq j \leq n, j \neq i: \text{turn}[j] == 0 \vee \text{turn}[i] < \text{turn}[j])\}$
- Problem to be solved separately: incrementing turn can cause arithmetic overflow, but when ?



bakery algorithm fine-grained (4)

```
int turn1=turn2=0
process p1 {
  while (true) {
    turn1 = turn2 + 1; /*1*/
    while (turn2 !=0 and /*2*/
           turn1 > turn2) skip;
    critical section;
    turn1 = 0;
    noncritical section;
  }
}
process p2 {
  while (true) {
    turn2 = turn1 + 1;
    while (turn1 !=0 and
           turn2 > turn1) skip;
    critical section;
    turn2 = 0;
    noncritical section;
  }
}
```

- Problem: both turn1 and turn2 could be set to 1 concurrently, so they are not unique, i.e. both p1 and p2 could get in the critical section at the same time, i.e. *race condition*.
- An intuitive to solve this:
 - A process which is just about to draw its turn value /*1*/ should be able to postpone the selection of the next process /*2*/ to get into CS
 - In case more than one process will get the same turn value, we should have tie-break rule. This will not hurt the fairness, if a process can not be by passed by others several times.



bakery algorithm fine-grained (5)

Basic idea: A process, p say, just about to choose its turn value has the smallest possible turn value ($=1$), so that it will slow down the selection of the smallest actual turn value and the process to enter CS, until p has got its actual turn value.

```
int turn1=turn2=0
process p1 {
  while (true) {
    turn1 = 1;
    turn1 = turn2 + 1;
    while (turn2 !=0 and
           turn1 > turn2) skip;
    critical section;
    turn1 = 0;
    noncritical section;
  }
}
process p2 {
  while (true) {
    turn2 = 1;
    turn2 = turn1 + 1;
    while (turn1 !=0 and
           turn2 ≥ turn1) skip;
    critical section;
    turn2 = 0;
    noncritical section;
  }
}
```

Both $p1$ and $p2$ may get $turn = 1$ and both will get in. So an asymmetry is introduced based on process id, so that $p1$ overdrives $p2$.

Why does it not affect fairness?

Convince yourself that setting $turn = 1$ is still necessary at the start of the entry procedure!



bakery algorithm (5)

fine-grained for n processes

```
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 st j != i]
      while (turn[j]!=0 and
            (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```



Heuristics for the CS solutions:

- Start from an abstract idea to implement a coarse grain algorithm.
- Split the coarse grain atomicity to get a more fine grained one which can be implemented either directly with normal HW-instructions, or using a more primitive CS-solution
- Break the symmetry (e.g. to avoid deadlocks) by introducing static, dynamic or even random (e.g. Ethernet media access protocol) asymmetry for tie-break arbitration
- Avoid global shared state written by many - or even better: read by many processes to get a truly distributable and scalable solution.
- Check out that you really understand your solution by constructing a *global invariant* and verifying mutual exclusion, deadlock freedom and fairness.