IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Principles of Parallel and Concurrent Programming

Synchronization and Mutual Exclusion

Rocco De Nicola

January 2026

GSSI - L'Aquila and IIT CNR - Pisa

Concurrent Processing Mode

Atomic Actions, States and Histories

- A (sequential or concurrent) program consists of atomic statements causing well defined, atomic, state changes depending on the computer architecture.
- An execution of a sequential program , i.e. a process is a sequence of atomic actions also called events inducing a sequence of consecutive states.
- The state of a process can only be observed from the outside between two atomic actions.
- An execution of a concurrent program consists of several processes/threads running physically in parallel (multiprocessing) or virtually in parallel (multitasking).
- The trace (history) of a concurrent system/program is one interleaving of the traces of its processes/threads into one, single trace.

Process Histories

The sequence of states makes up a process history. A history is a trace of ONE execution:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$$

and transitions are the consequences of atomic actions a_i that induce a state change.

Atomic Actions

- Atomic actions are indivisible sequence of state transitions made atomically
- Fine-grained atomic actions are machine instructions (read, write, swap, etc.) with atomicity guaranteed by HW
- Coarse-grained atomic actions are a sequence of fine-grained atomic actions executed indivisibly (atomically). Internal state transitions are not visible from "outside".
- <statements> denotes a list of statements to be executed atomically.

Interleaving Semantics of Concurrent Execution

The concurrent execution of several processes can be viewed as the interleaving of histories of the processes, i.e., the interleaving of sequences of atomic actions of different processes.

Individual histories:

P1: $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$

P2: $p_0 \xrightarrow{b_1} p_1 \xrightarrow{b_2} \ldots \xrightarrow{b_n} p_n$

P3: $r_0 \xrightarrow{c_1} r_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} r_n$

The parallel execution of P1 , P2 e P3 yields many traces, eg.:

- $p_0||q_0||r_0 \xrightarrow{a_1} s_1||q_0||r_0 \xrightarrow{b_1} s_1||q_1||r_0 \xrightarrow{b_2} s_1||q_2||r_0 \xrightarrow{c_1} \ldots$
- $p_0||q_0||r_0 \xrightarrow{b_1} s_0||q_1||r_0 \xrightarrow{c_1} s_0||q_1||r_1 \xrightarrow{b_2} s_0||q_2||r_0 \xrightarrow{c_2} \ldots$
- $\ldots$

That represent the concurrent histories of P1 , P2 and P3.

Hardware assumptions

- Values of basic types, e.g., int, are stored in memory locations, e.g., words, that are read and written as atomic actions;
- Values of program variables are manipulated by loading them into registers (LOAD), modifying the register value (EXECUTE) and storing the results back into memory (STORE);
- each process has its own set of registers

  real registers in a multiprocessing implementation
  logical registers in a multiprogramming implementation, where register values are saved and restored when switching processes
- when evaluating a complex expression, e.g., $z = x * (y + 1)$, intermediate results are stored in registers or in memory private to the executing process, e.g., on a private stack.

Special machine instructions

In addition to reads and writes of single memory locations, most modern CPUs provide additional special indivisible (atomic) instructions (in the single-CPU case), e.g. (where x is a variable and r is a register.):

- Exchange instruction
  - `SWAP(int x, int y) = <z = x , x = y, y = z>` ($x \longleftrightarrow y$)
- Increment and Decrement instructions:
  - `INC(int x) = <x = x + 1; r = x>`
  - `DEC(int x) = <x = x - 1; r = x>`
- Test-and-Set instruction used to write to a memory location and test and return its old value as a single atomic (i.e. non-interruptible) operation:

```
function TestAndSet(boolean lock)
boolean <temp = lock; lock = true; return temp>
```

Find the Maximum I

- Given array $a[1{:}n]$ of positive integers. Find the maximum value m

$$(\forall\, j : 1 \le j \le n :\ m \ge a[j]) \ \wedge\ (\exists\, j : 1 \le j \le n :\ m = a[j])$$

  - When program terminates, m is at least as large as every element of a
  - m is one of elements of a
- Sequential program:

```
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m) m = a[i];
```

- How to parallelize? – Examining all elements in parallel
- Parallel program without synchronization:

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

  - Program is incorrect because of the race condition
  - Synchronization is needed

Find the Maximum II

- First attempt to synchronize: execute cond. updates atomically

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

  - Not efficient (overconstrained): all the updates are serialized just like in the sequential program but executed in arbitrary order.
- Observations:
  - Read and test can be executed in parallel for each **i**
  - Updates of **m** require atomicity (mutual exclusion) for serialization
  - Can use second test in critical section to avoid races

Find the Maximum III

- Read and test in parallel without mutual exclusion. Those who passed the test, execute conditional update atomically (with mutual exclusion)

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

- Lessons learned
  - Synchronization is required whenever processes read and write shared variables (to preserve dependencies)
  - Atomicity helps to provide mutual exclusion
  - Test followed by an atomic test-and-update is a useful combination for conditional updates
    - Helps to avoid races among concurrent conditional updates that depend on the same condition

Asynchronous process execution

Asynchrony

In multiprocessing systems the processes usually have little or no control over the way their atomic actions are interleaved.

Advantage:

Applications programmer can ignore the problems of timesharing the processes

Disadvantage:

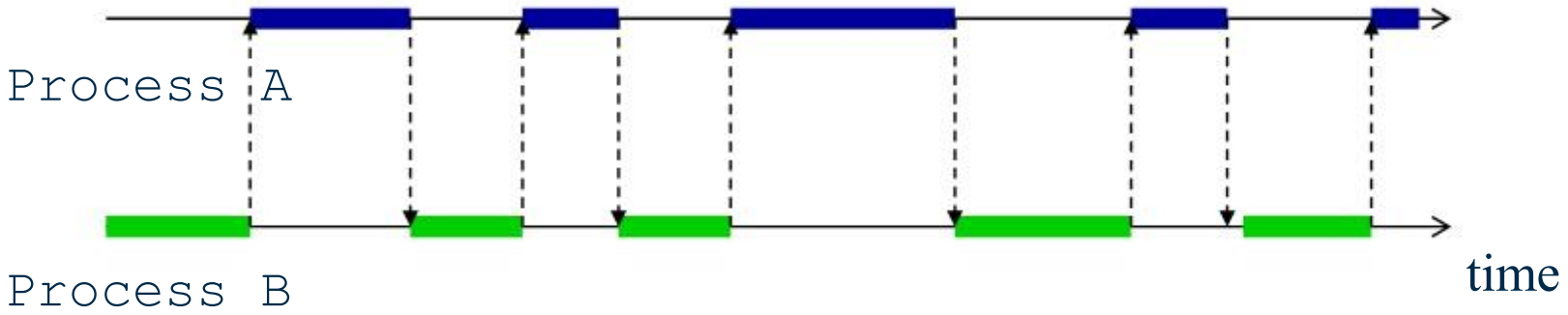
Since processes run asynchronously, we can not make any assumption on their relative speed, about which one starts first, when they will be suspended etc.

Important Consideration

Each program statement or machine instruction ultimately reduces to a sequence of atomic actions on the shared memory and the effect of executing a set of atomic actions in parallel is equivalent to executing them in some arbitrary serial order.

Concurrent Execution and Interleaving

Consider a multiprogramming implementation of a concurrent program consisting of two processes:



The switching between processes occurs voluntarily or in response to interrupts. The processor executes a sequence of instructions which is an interleaving of the instruction sequences from each process.

Process switching does not affect the order in which instructions are executed by each process.

Nondeterminism of Concurrent Execution

Different interleavings (histories) can be observed on different concurrent executions.

- A concurrent program of n processes each of m atomic actions can produce $(n \times m)!/(m!)^n$ different histories.
  If n = 3, m = 2 we have 90 different histories.
- $(n * m)!/(m!)^n$ is obtained considering that $(n * m)!$ represents the possible sequences obtained by permuting$(n * m)$ atomic actions and that from this number we need to remove all the sequences obtained by permuting the sequences of the actions of the individual processes.
- In general, if we have n processes that execute $m_1, m_2, \ldots m_n$, actions we have that the number of possible histories is :

$$(\sum_{i=1}^{n} m_i)!/\prod_{i=1}^{n}(m_i)!$$

Example of Nondeterminism

Given the program

```
int y = 0; z = 0;
co x = y + z; || y = 1; z = 2; oc}
```

let us consider the possible runs that lead to four different final values of variable `x`:

Trace 1: `x = y{0} + z{0}; y = 1; z = 2;`
final value x = 0

Trace 2: `y = 1; x = y{1} + z{0}; z = 2;`
final value x =1

Trace 3: `y := 1; z := 2; x := y{1} + z{2};`
final value x = 3

Trace 4: `load y{0} to R1; y := 1; z := 2;`
`add z{2} to R1{0}; store R1 to x;`
final value x = 2

Synchronization

Synchronization is a mechanism to delay processes to

- Reduce the entire set of possible histories to those which are desirable/correct.
- Preserve (true) dependences between processes
- Avoid race conditions, if any.

Condition synchronization:

Delays a process until a certain condition is true.

Example:

If process A produces data and process B prints them, B has to wait until A has produced some data before starting to print.

Interference and Mutual Exclusion

Interference

If instructions from different processes are arbitrarily interleaved, any interleaving which is not explicitly prohibited is allowed. Some interleavings might have unwanted results due to unwanted interference on read and write shared variables of processes.

Mutual exclusion

To avoid interference, we need to ensure that no two processes access a shared variable at the same time by marking parts of code as critical and requiring that no two processes execute it at the same time (critical section).

Let CS and CS' be time intervals in which two different processes execute the critical section, then Mutual exclusion means that for each such pair:

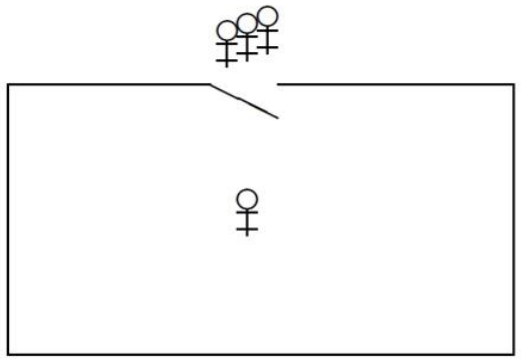
$$CS \longrightarrow CS' \text{ or } CS' \longrightarrow CS.$$

Critical sections

A critical section is a section of code belonging to a process in a concurrent program that:

- accesses a shared resource, e.g a variable, a communication channel, a file, etc.;
- for correct behaviour of the program only one process may access the shared resource at a time.

A critical section in which at most one process at a time may be in.



Our problem

Provide the protocols for opening and closing the door to the critical section, to ensure that a process waits in front of its critical section when this is occupied by others.

Archetypical mutual exclusion

Any program consisting of n processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

```
// Process 1          // Process 2    ...    // Process n
init1;                init2;                 initn;
while(true) {         while(true) {          while(true) {
  crit1;                crit2;                 critn;
  rem1;                 rem2;                  remn;
}                     }                      }
```

where *initi* denotes any (non-critical) initialisation, *criti* denotes a critical section, *remi* denotes the (non-critical) remainder of the program, with $i = 1, 2, ...n$.

Mutual exclusion via interrupts

On a mono-processor, coarse grained atomic actions (hence also critical sections) can be implemented at the hardware level by disabling interrupts. But disabling interrupts has many disadvantages:

- it is possible only in privileged mode (what would happen if was possible in user mode?);
- it excludes all other processes, and thus reduces concurrency;
- it does not work in multiprocessing implementations (disabling interrupts is local to one processor)

Disabling interrupts is not a very useful from the point of view of an application programmer and is only useful in specific situations like:

1. developing operating systems;
2. writing software for embedded systems;
3. using simple (single-user) processors

Basic mutual exclusion

Atomic machine instructions can be used to solve some very simple mutual exclusion problems directly, e.g.:

Single-Word Readers and Writers

If many processes read and/or write to a shared variable and this is stored in a single word, then the memory unit ensures mutual exclusion for all accesses to the variable.

Shared Counter

Several processes each increment a shared counter. If the counter can be stored in a single word, INC (if available!) can be used to update the counter, ensuring mutual exclusion.

Problems with (fine-grained) atomic actions

Relying on fine-grained atomic actions is not very valuable for the applications programmer:

- atomic actions do not work for multiprocessor implementations of concurrency unless one can lock memory
- the set of atomic actions (special instructions) varies from machine to machine
- It cannot be assumed that a compiler will generate a specific sequence of machine instructions from a given high-level statement
- the range of things one can do with a single machine instruction is limited (no critical section with more than one instruction)

Solution:

Atomic actions can be used to implement, on specific platforms, higher-level synchronisation primitives and protocols.

Mutual exclusion protocols

To solve the mutual exclusion problem, we adopt a standard Computer Science approach by relying on specific protocols:

- Designed to be used by concurrent processes to achieve mutual exclusion and avoid interference
- Consisting of a sequence of instructions to be executed before entering (entry protocol) and after leaving (exit protocol) the critical section
- Defined using standard sequential programming primitives or some common special instructions.

Fine-grained atomic actions are used to implement higher-level coarse grained synchronisation primitives and protocols.

General Structure of a Mutual Exclusion Protocol

There are many ways to implement such a protocol. We assume that each of the n processes have the following form:

```
// Process i
initᵢ;
while(true) {
    // entry protocol
    critᵢ;
    // exit protocol
    remᵢ;
}
```

Necessary properties for any solution of the CS problem

- Mutual Exclusion: at most one process at a time is executing its critical section
- Deadlock Freedom: if no process is in its critical section and some processes want to enter their critical sections, one succeeds (No Livelock)
- Eventual Entry: a process that is attempting to enter its critical section will eventually succeed (No Starvation)

A first solution with await

The await statement can be used for achieving synchronisation; it is a conditional conceptual corse grained atomic action

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

The process executing this instruction blocks until B becomes true then atomically executes S.

- B is a boolean expression specifying the waiting condition
- S is a sequence of sequential statements that is guaranteed to terminate
- B is guaranteed to be true when execution of S begins, no internal state of S is visible to other process

(Under specific circumstances) it can be implemented as:

```
<while (not B) do Skip od; S>
```

This is called a spin loop, because the while statement has an empty body, and repeatedly checks B until it becomes true.

Implementing `await` with a Spin Loop

The statement `<await (B) S;>` can be implemented as `<while (not B) do Skip od; S>` only under the following circumstances:

- `B` is a side-effect-free boolean expression
- `S` does not modify any variable appearing in `B`
- `S` is sequential, non-blocking, and guaranteed to terminate
- Execution of `S` is atomic: no intermediate state of `S` is visible to other processes
- The scheduler is fair, so `B` will eventually be re-evaluated
- Busy waiting is acceptable (short waiting time or no blocking primitives available)

Otherwise, the spin-loop implementation is not equivalent to `await`.

Critical section: A solution using 2 variables

To guarantee the mutual exclusion property we need a way to indicate whether a process is in or out of its critical section.

We develop a solution for two processes **P1** and **P2**

- Let in1 and in2 be Boolean variables that are initially false
- When **P1** is in its critical section we set in1 = true
- When **P2** is in its critical section we set in2 = true
- The bad state we want to avoid is the one in which both in1 and in2 are true

Critical section problem: 1st solution

```
// Process P1

init1;
while(true) {
    <await (!in2) in1=true>
    crit1;
    in1 = false // exit
    rem1;
}
```

```
// Process P2

init2;
while(true) {
    <await (!in1) in2=true>
    crit2;
    in2 = false // exit
    rem2;
}
```

The solution has the following properties:

- Mutual Exclusion: yes!
- Absence of Deadlock: yes!
- Eventual Entry: not guaranteed!
  - Additional Assumption Needed: Eventual Exit and Fair CPU scheduling.

Critical section problem: 1st solution - a variant

- The previous solution used two variables to record the status of the two process.
- There are only two interesting states: a process is in critical section or none is. One variable is sufficient to specify this!.

```
// Process CS1                          // Process CS2

init1;                                  init2;
while(true) {      // entry             while(true) {
  <await (!lock) lock=true;>              <await (!lock) lock=true;>
  crit1;                                    crit2;
  lock = false; // exit                     lock = false // exit
  rem1;                                     rem2;
}                                       }
```

- Works also for n-processes.
- Still uses the coarse grained primitive `< await ... >`

Critical section problem: using Test and Set

The atomic (possibly fine-grained) Test and Set (TS) instruction can be used to implement the entry protocol.

```
// Process Pi (1<i<n)

init1;
while(true) {# entry protocol
  while (TS(lock))
  skip //spin until lock is acquired;

  crit1;

  lock = false; # exit protocol

  rem1;
}
```

- The await statements are replaced by while-loops that do not terminate until lock is false, and TS returns false
- This solution is referred as a spin lock because the process keep spinning while waiting for the lock to be cleared.

Properties of the Test and Set Solution

- Mutual Exclusion: YES. if two processes try to enter their critical sections, only one succeeds in changing the value of lock from false to true.
- Absence of Deadlock: YES. If any number of processes are in their entry protocols, and no process is inside a critical section, lock is false and one of the processes will enter
- Eventual Entry: NOT guaranteed. Could be guaranteed if the CPU scheduling policy is strongly fair and it is guaranteed that any process that enters its critical section eventually exits.

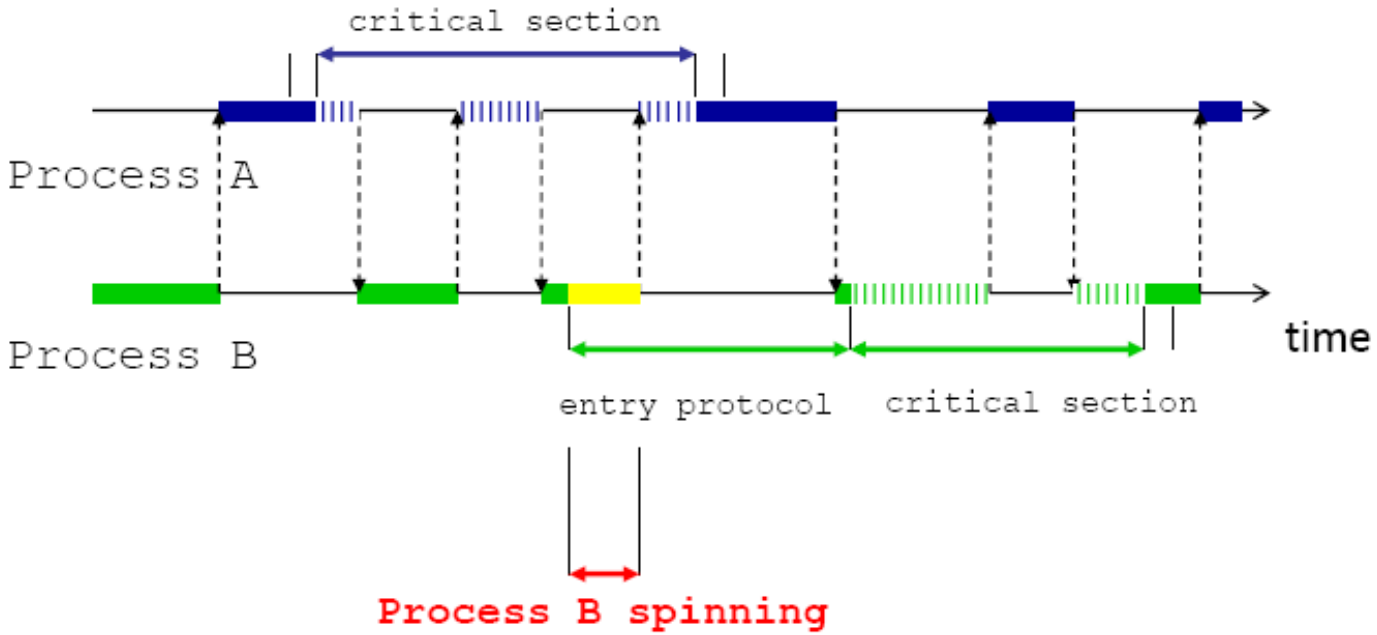
Strong Fairness

A strongly fair scheduling policy guarantees that if a process is infinitely often enabled to execute an action, it will eventually do it.

Weak Fairness

A weakly fair scheduling policy guarantees that if a process is permanently enabled to execute an action, it will eventually do it.

0verhead of Spin Lock



- Time spent spinning is necessary to ensure mutual exclusion but it wastes CPU time: B cannot do useful work while A is in its critical section
- If the critical sections are large relative to the rest of the program, a concurrent program will be slowed down
- When many processes are contending access a large amount of CPU could be wasted

Test and Test and Set

A spin-lock implementation of the entry protocol that reduces concurrent use of a "demanding" atomic operation.

Concurrent simple checks of lock, then, and only if lock is false, other TS checks are performed to enable only one process to set `lock` to `true`.

```
bool lock=false;          /* shared lock */
process CS [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;
   }   }
```

Towards a live solution

Solving the critical section problem without special atomic instructions (TS), and without special (strongly fair) schedulers but using turn variable to let process to control each other progress.

```
shared variable Int turn = 1;
Process P1:                          Process P2:
Init1;                               Init2;
while(true){                         while(true){
   while(turn=2){SKIP};                 while(turn=1){SKIP};
   Crit1;                               Crit2;
   turn=2;                              turn=1;
   rem1;     }                          rem2;     }
```

WRONG: Mutual exclusion works, but processes must strictly alternate their turn. If one of them idles in the non-critical section, the other wanting to enter unnecessarily waits.

Petersons' solution

Peterson's algorithm puts together the ideas of the two previous faulty solutions:

- Enforces mutual exclusion
- Uses only simple shared variables and no special instructions such as Test-and-Set
- Uses two flags variables: c1 for process P1, c2 for process P2 . The value of 1 of a flag indicates that the owner process wants to enter the critical section.
- A variable turn holds the ID of the process whose turn it is
- Entrance to the critical section is granted for process P1 if P2 does not want to enter its critical section or if P2 has given priority to P1 by setting turn to 1.

Peterson's Coarse-Grained solution

```
bool in1=false, in2=false; int last = 1
process CS1  {
  while (true) {
    in1=true; last=1;                  /* entry protocol */
    <await (!in2 or last==2);>         /* entry protocol */
       critical section;
    in1=false;                        /* exit protocol */
      noncritical section;    }   }
process CS2  {
  while (true) {
    in2=true; last=2;                  /* entry protocol */
    <await (!in1 or last==1);>         /* entry protocol */
      critical section;
    in2=false;                        /* exit protocol */
      noncritical section;    }   }
```

Correctness of the solution

Main properties:

Mutual exclusion: Is guaranteed by the atomic test of `in1` and `last` (and of `in2` and `last`) from the fact that `last` is a boolean.

No Livelock: Here a key rôle is played by the shared variable `last` and is used to "break the tie".

Eventual Entry: Also this property is guaranteed by `last` because it permits keeping track of the process that last was in its critical section

Generalization to n processes

The generalization to a generic number of processes is possible, but the solution is rather elaborate. Later we will report the generalizarion of the fine-grained version.

Petersons' Fine-Grained solution

```
shared bool in1=false, in2=false;
shared int last = 1   /* we could have last = 2 */
 process CS1
  while (true)
   {in1=true; last=1; while (in2 and last == 1) {skip};
     critical section;
     in1= false;
     noncritical section }
 process CS2
  while (true) {
   {in2=true; last=2; while (in1 and last==2) {skip};
    critical section;
    in2=false;
    noncritical section }
```

Petersons' Fine-Grained solution - Correcteness

No need to evaluate atomically the delay condition because, in principle the two corresponding instructions of CS1:

```
(*)  in1=true; last=1; <await (!in2 or last==2);>
(**) in1=true; last=1; while (in2 and last==1) skip;
```

1. could yield different outcomes if the tests on `in2` and `last` are not executed atomically: the value of `in2` could be `false` but it could become `true` before the evaluation of `last`, because in the mean time CS2 could have entered his critical section.
2. In this case CS1 could be allowed to enter the critical section, while CS2 is still inside.
3. However,
   - CS2 before trying to enter has certainly set `last=2`,
   - `in1` has been set to `true` and has not been changed (only CS1 can change it!),
   - Hence, CS2 is blocked and cannot enter.

Peterson's for n-processes

The generalization of Peterson's algorithm to n-processes, `CS[0]` to `CS[n-1]`, relies on the same basic idea of the version for two processes

- It uses arrays `in[0..n-1]` and "last[0..n-1]"
- `in[i]` indicates which stage "CS[i]" is executing
- `last[j]` indicates which thread last began stage `j`

Entry section contains an outer `FOR loop (j)` with $n-1$ stages

The inner `FOR loop (k)` in `CS[i]` checks every other thread

- Waits for every other thread with a higher or equal number
- Once another thread enters stage j or all processes ahead of `CS[i]` have exited, `CS[i]` can proceed to next stage

This algorithm guarantees mutual exclusion, livelock-freeness, and ensures eventual entry

Peterson's for n-processes

Figure 2.7 The `Filter` lock algorithm.

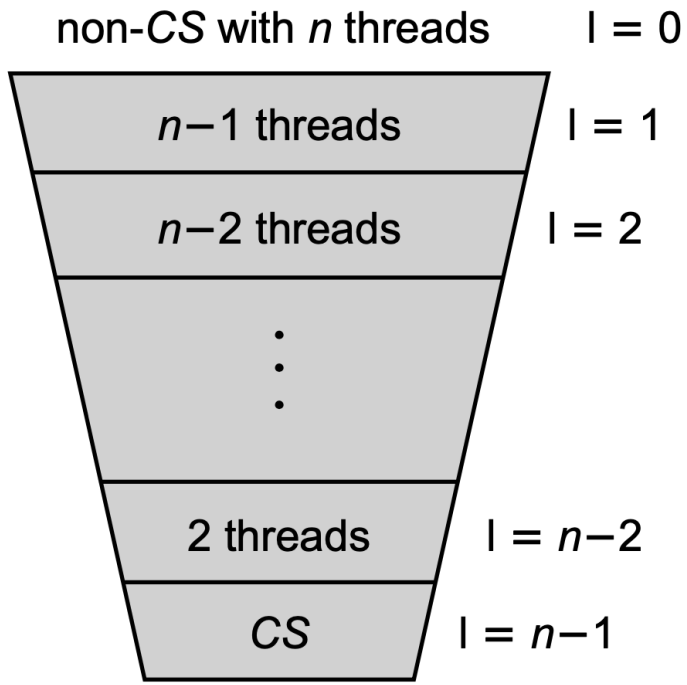


Figure 2.8 There are $n-1$ levels threads pass through, the last of which is the critical section. There are at most n threads that pass concurrently into level 0, $n-1$ into level 1 (a thread in level 1 is already in level 0), $n-2$ into level 2 and so on, so that only one enters the critical section at level $n-1$.

Ticket Algorithm

This algorithm works for any number of processes and satisfies the the three required properties for a solution of the critical section problem. It is much simpler than Peterson's Algorithm.

```
shared int number=1, next=1, turn[1:n]=([n] 0)
process CS[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;
                }           }
```

Unfortunately it needs the Fetch and Add atomic operation that might nor be available on all processors.

Coarse-Grained Bakery algorithm

Yet another algorithm!

First the coarse grained variant:

```
int turn[1:n]=([n] 0)
  process CS[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;
     }
  }
```

Fine-Grained Bakery algorithm

Now, the fine grained variant:

```
shared 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[j],j) =< (turn[i],i)) skip;
      critical section;
      turn[i] = 0;
      noncritical section;
   }      }
```

Lexicographic Ordering

`(a, b) < (c, d)  iff  (a < c) OR (a = c AND b < d)`

Barrier Syncronisation

Barrier Syncronisation is used to coordinate groups of processes especially when using UMA . Each process from the group at the end of its computation is suspended until all the processes in the group have reached a specific point called barrier.

A simple solution with co inside while

```
while(true) {
    co [i=1 to n]
        code to implement task i;
    oc  }
```

This has the disadvantage of creating n processes each iteration.

One of the process to be synchronized

```
process Worker[i=1 to n] {
    while(true) {
        code to implement task i;
        wait for all n processes to complete  }  }
```

Possible approaches to Barrier Syncronisation

- Shared Counter
- Coordinating Process
- Symmetric Barriers

Shared Counter

```
shared int count = 0;
process Worker[i = 1 to n] {
   while (something) {
          do some work;
          <count = count + 1;>
          <await (count == n);>  } }
```

```
   <count = count + 1;>     and     <await(count==n);>
are rendered as
   FA(count, 1);     and     while (count != n) skip;
```

Reusable Barrier

Wrong Solution

```
shared int count = 0;
process Worker[i = 1 to n] {
   while (something) {
          do some work;
          <count = count + 1;>
          <await (count == n); count = 0;>    }  }
```

A process can increment count and proceed to the test only after another process as set the counter to 0.

Reusable Barrier

Correct Solution: Sense Reversing Barrier

```
shared int count = 0; shared boolean sense = false;
process Worker[i = 1 to n] {
private boolean mySense = !sense; ## one per thread
      while (something) {
           do some work;
          < count = count + 1;
            if (count == n) { count = 0; sense = mySense; }
           >
        while (sense != mySense); ## wait
       mySense = !mySense;    }   }
```

The flag `mySense` permits determining when a process can start again its execution.

Process Coordinator

```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);

 process Worker[i=1 to n] {
   while (true) {
      do task i;
      arrive[i] = 1;
      <await (continue[i] == 1);>
      continue[i] = 0;   }

 process Coordinator {
   while (true) {
     for [int i=1 to n] {
        <await (arrive[i] == 1);>
        arrive[i] == 0; }
    for [int i=1 to n] continue[i] = 1;  } }
```

arrive and continue are *flag variables*.