Interaction and cooperation among processes

Motivation

Simultaneously active processes are said to be executed *concurrently* or in *parallel*. They can operate completely independently, but some *interaction* between parallel processes is often needed. The origin of this need for interaction can be one of the following:

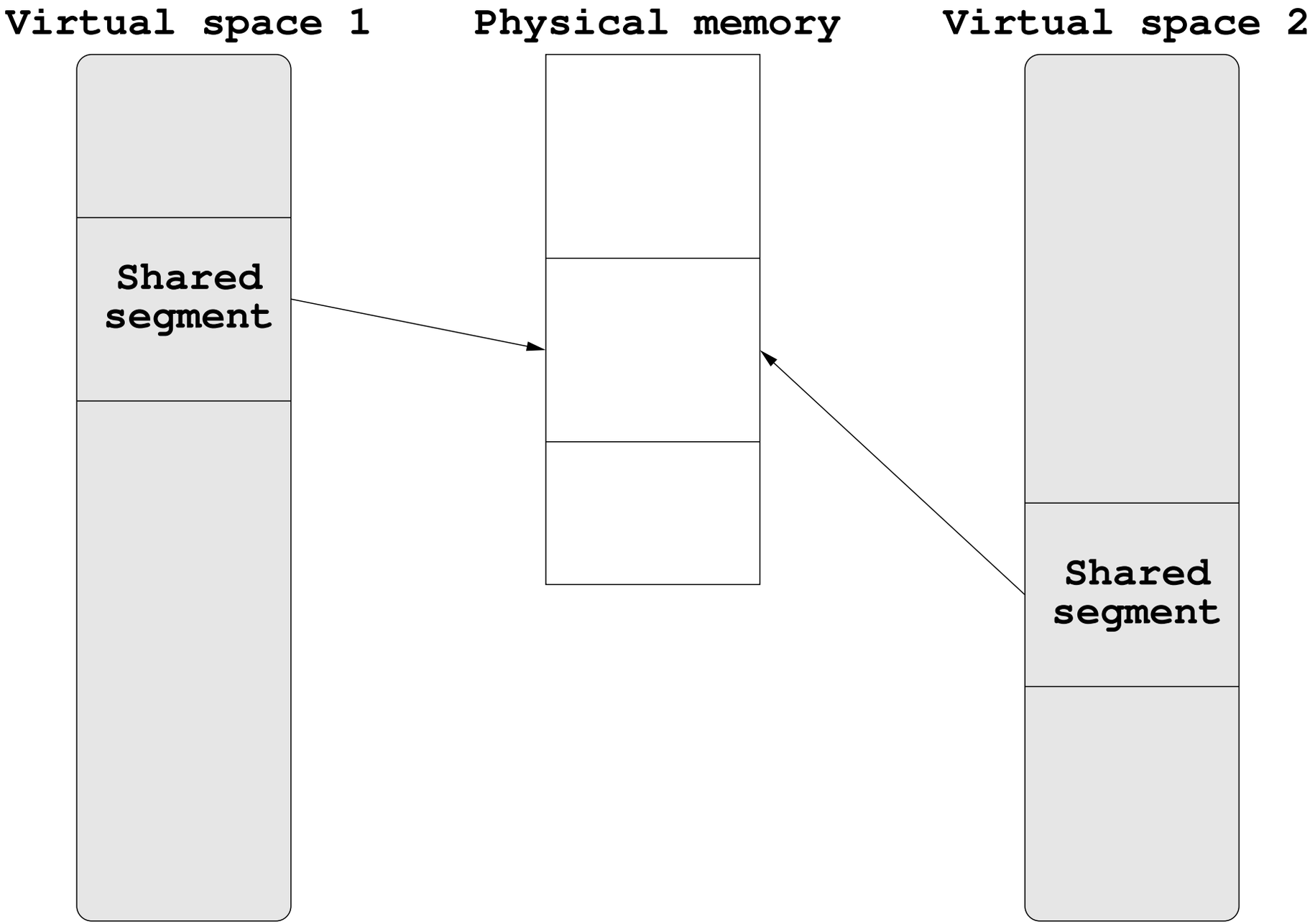
- The processes are cooperating towards a common goal, for example
  - parallel computing on a multiprocessor machine,
  - cooperation between processes running on different machines in the context of a distributed application,
  - a program is written as parallel tasks, for instance in the case of a control system;
- Sharing resources, for example a disk-based file system.

Managing interaction between parallel processes: Interprocess communication

For now, we consider processes executed on a single, possibly multiprocessor, machine.

- Process interaction will take place with system calls designed for this purpose.
- A first interprocess communication mechanism is the creation of *shared memory segments*, i.e. physical memory segments that appear in the virtual memory of several processes.
- Once a shared memory segment has been created, the processes can interact through this shared memory.
- The problem to be solved is to develop programming techniques that can exploit this form of interaction.

A shared memory segment



Understanding shared memory

- The basic operations on shared memory are reads and writes (`LD` et `ST`).
- The `LD` and `ST` instructions are executed atomically (without any possible interaction). Indeed,
  - on a single processor, switching between processes always occurs at instruction boundaries;
  - on a shared memory multiprocessor, a mechanism guaranteeing exclusive access to the bus connecting the processors to the memory is essential.
- The behavior of the processes will thus depend on the order in which the `LD` and `ST` instructions of the different processes are executed.
- In most cases, this order is unknown and will vary from one execution to another.

A process interaction example

In the program below, we assume that, for each process, `r2` contains the address of a word in shared memory that has been initialized to 0.

Process 1	Process 2
`LD(r2,0,r0)`	`LD(r2,0,r0)`
`ADDC(r0,1,r0)`	`ADDC(r0,1,r0)`
`ST(r0,0,r2)`	`ST(r0,0,r2)`

Depending on whether the execution order is

```
LD(r2,0,r0)
ADDC(r0,1,r0)
ST(r0,0,r2)
                LD(r2,0,r0)
                ADDC(r0,1,r0)
                ST(r0,0,r2)
```

or

```
LD(r2,0,r0)
                LD(r2,0,r0)
ADDC(r0,1,r0)
                ADDC(r0,1,r0)
ST(r0,0,r2)
                ST(r0,0,r2)
```

the word in shared memory will have the value 2 or the value 1 after the execution of the processes. Note that since each process operates in its own context, registers are not shared.

Possible execution orders: The interleaving model

- If process interaction is limited to the atomic shared memory accesses `LD` and `ST`, the result of a concurrent execution will be the same as that of a serialization (an *interleaving*) of the processes' instructions.

- This is obvious on a machine with a single processor; on a multiprocessor it is a consequence of the fact that the operations `LD` and `ST` are atomic and that these operations are thus executed sequentially. Since there is no interaction linked to other instructions, executing them in parallel or sequentially yields the same result.

- The serialization that will actually be observed will depend on the process scheduler and on the state of the machine when the program is executed.

- Given that these items are not perfectly known and can vary depending on the execution, we will assume that *any* serialization can be observed. This is what is called the *interleaving model*:

  > **The result of the execution of parallel processes can be that of any interleaving of the elementary instructions executed by these processes.**

- Thus, when writing a program that will be executed as parallel processes (a *parallel program*), one must make sure that it is correct for all interleavings of the elementary instructions it is composed of.

Writing parallel programs

- The interleaving model is a simple and safe abstraction of the behavior of parallel programs, but it is not easy to write programs that are correct for this model.

- Consider the case of a shared data structure. If several processes are simultaneously applying operations composed of several elementary actions, it is hard to ensure that these simultaneous operations will be performed correctly.

- The solution is to turn these operations on the shared data structure into *atomic* operations, in other words to ensures that only one process at a time can modify the date structure.

- For doing this, we need an algorithm to guarantee *mutual exclusion*.

Mutual exclusion

- The *mutual exclusion* problem consists in guaranteeing that code sections, called *critical sections*, of parallel programs are never executed simultaneously.

- Precisely, it is required that when a process enters a critical section, no other process can do the same before this process has finished executing the critical section.

- Mutual exclusion is a basic building block that makes it possible to write complex parallel programs.

- Using only shared memory, mutual exclusion is harder to achieve than one might think at first sight.

- We will consider the case of 2 processes.

Mutual exclusion : general schema

Consider two processes whose structure is the following.

```
#define True  = 1
#define False = 0
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */

         /* entry protocol */
  crit1: /* critical section */
         /* exit protocol */
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */

         /* entry protocol */
  crit2: /* critical section */
         /* exit protocol */
}
```

Both processes cannot simultaneously be at locations `crit1` and `crit2`.

Mutual exclusion : first try

```
int Turn = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         while (Turn == 2) {};
  crit1: /* critical section */ ;
         Turn = 2;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section  */
         while (Turn == 1) {};
  crit2: /* critical section */
         Turn = 1;
}
```

This solution guarantees mutual exclusion, but imposes a strict alternation between the two processes, which is too restrictive.

Mutual exclusion : second try

```
int c1,c2  = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         while (c2 == 0) {};
         c1 = 0;
  crit1: /* critical section */ ;
         c1 = 1;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         while (c1 == 0) {};
         c2 = 0;
  crit2: /* critical section */
         c2 = 1;
}
```

Mutual exclusion is not guaranteed:

c1	c2	
1	1	P1 checks c2
1	1	P2 checks c1
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	*Ouch!* !

mutual exclusion : third try

```
int c1,c2  = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         c1 = 0;
         while (c2 == 0) {};
  crit1: /* critical section */ ;
         c1 = 1;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         c2 = 0;
         while (c1 == 0) {};
  crit2: /* critical section */
         c2 = 1;
}
```

The program can get stuck :

c1	c2	
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 checks c2
0	0	P2 checks c1

Such a blocked situation is called a *deadlock* (*étreinte fatale* in French).

mutual exclusion : fourth try

```
int c1,c2  = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         c1 = 0;
         while (c2 == 0)
         {   c1 = 1;
             /* wait*/;
             c1 = 0;
         };
  crit1: /* critical section */ ;
         c1 = 1;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         c2 = 0;
         while (c1 == 0)
         {   c2 = 1;
             /* wait */;
             c2 = 0;
         };
  crit2: /* critical section */
         c2 = 1;
}
```

In this solution, if a process finds out that it cannot proceed to its critical section, it gives up for a while before trying again. A blocked situation is still possible, but it will persist only if there is a perfect symmetry between the execution of the two processes.

c1	c2	
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 chekcs c2
0	0	P2 checks c1
0	0	P1 : c1 = 1
1	0	P2 : c2 = 1
1	1	P1 : c1 = 0
0	1	P2 : c2 = 0
0	0	P1 checks c2
0	0	P2 checks c1

Mutual exclusion : Dekker's algorithm

```
int c1,c2, Turn  = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         c1 = 0;
         while (c2 == 0)
         {  c1 = 1;
            while (Turn == 2) {};
            c1 = 0;
         };
  crit1: /* critical section */ ;
         Turn = 2;
         c1 = 1;
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         c2 = 0;
         while (c1 == 0)
         {  c2 = 1;
            while (Turn == 1) {};
            c2 = 0;
         };
  crit2: /* critical section */
         Turn = 1;
         c2 = 1;
}
```

The symmetry of the previous solution id broken thanks to the variable `Turn`.

Is Dekker's algorithm correct?

- Apparently yes, but how can one show this rigorously?

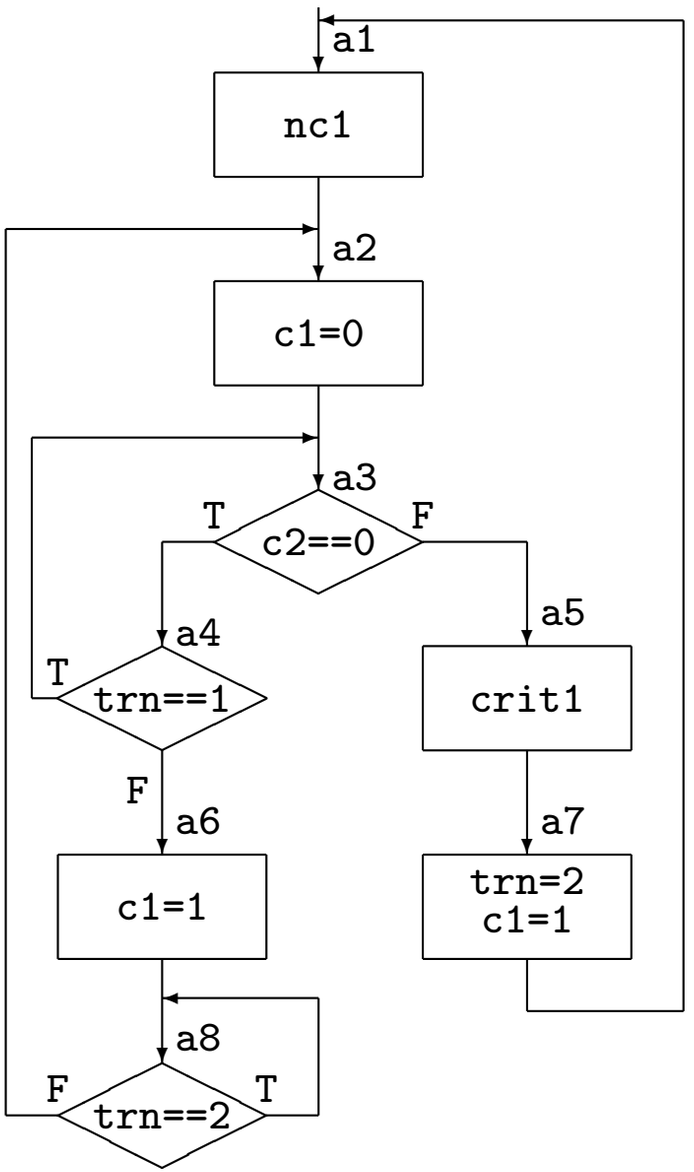
- One approach is to systematically examine all possible interleavings of the actions of the processes.

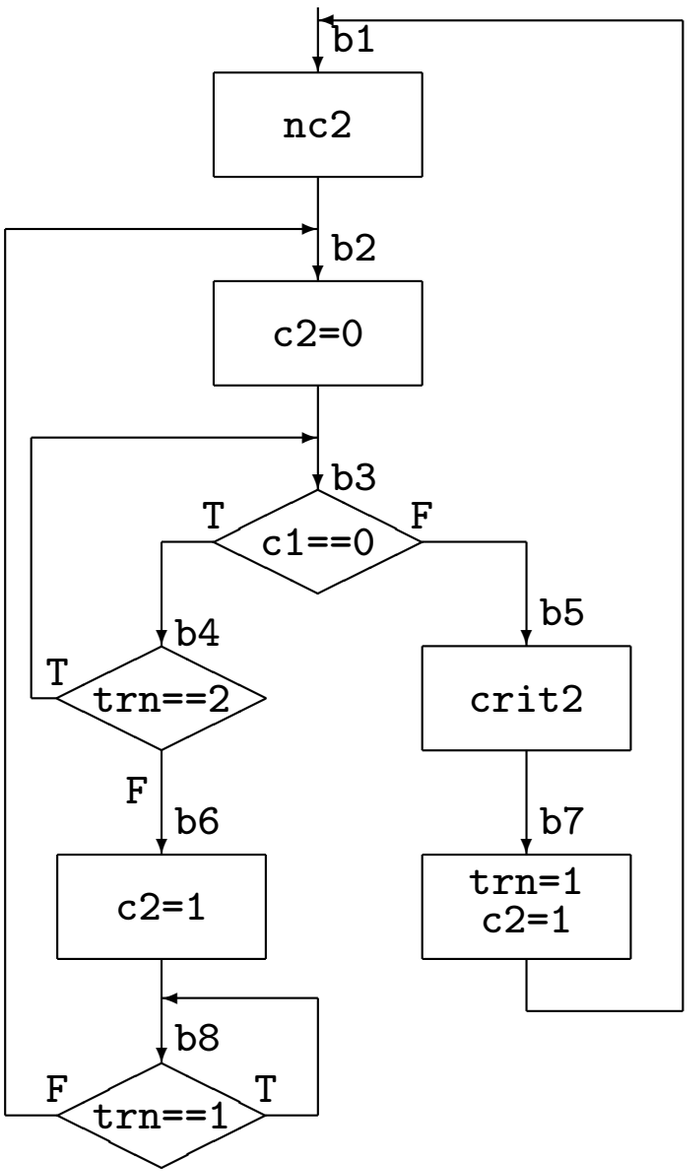
- For doing this, it is convenient to present the program in which the the control structure and the actions of the process are explicit.

- One then explores the possible executions of the program, while remembering the states that have been reached in order to stop the exploration when reaching a state that has previously been reached.

Dekker with explicit control (slightly modified version)

Initializations : `c1 = 1; c2 = 1; trn = 1`



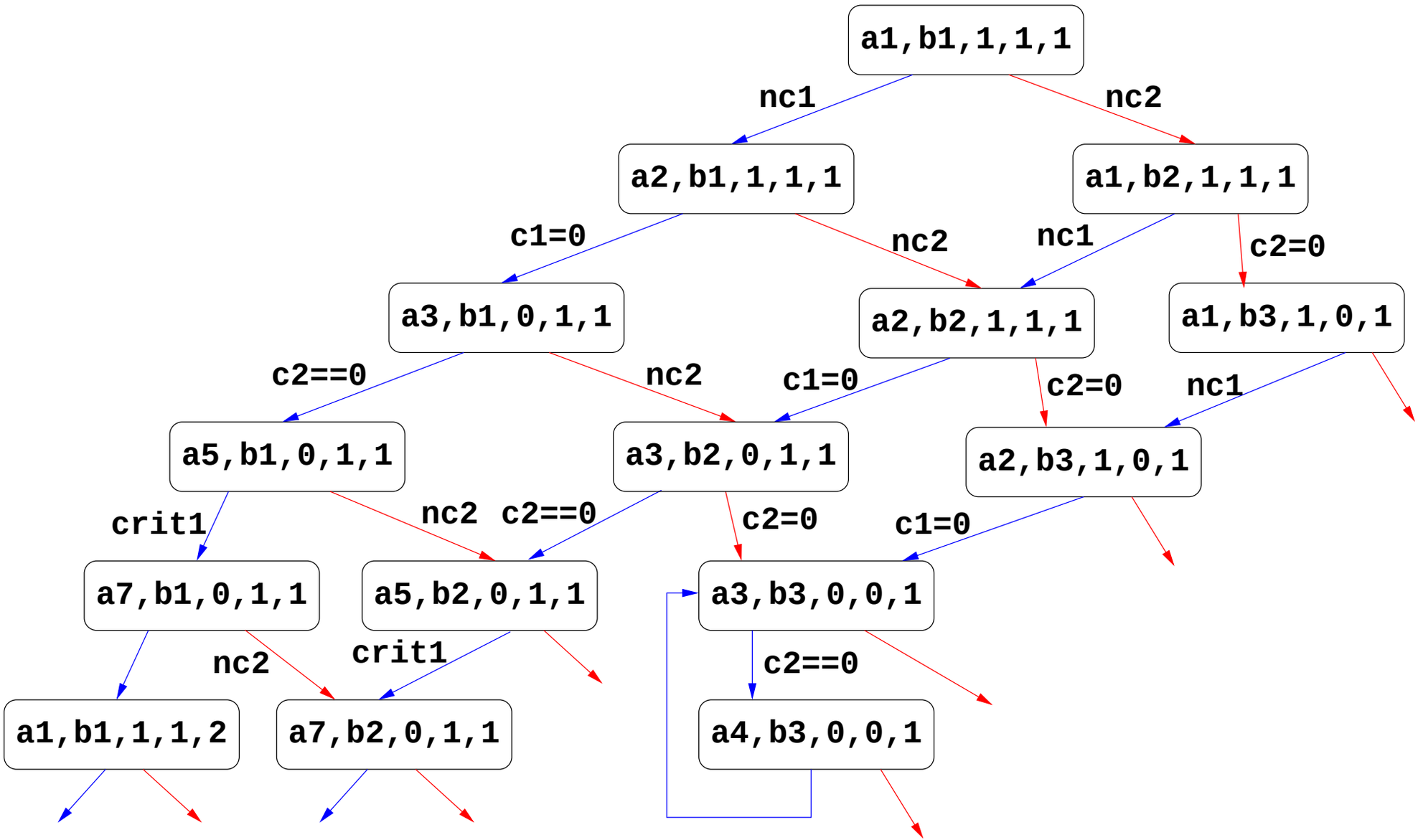


Exploring the executions of Dekker's algorithm

- During the execution of the algorithm, a state of the system is characterized by
  - a control location for each process,
  - a value for the three variables c_1, c_2 et trn.
- A state is thus a 5-tuple

  (`loc1,loc2,c1,c2,trn`)
- The initial state is (`a1,b1,1,1,1`)
- It is then sufficient, starting from the initial state, to generate new states by considering all possible actions.
- Exploring the executions (the possible states or *state space*) stops when no new state can be reached, which is bound to happen since there are at most $8 \times 8 \times 2 \times 2 \times 2 = 512$ states.

Exploring the state space of Dekker's algorithm



Interpreting state space exploration

- The mutual exclusion property is satisfied if it is impossible to reach a state in which the locations are `a5-a7` and `b5-b7`.

- It is also possible to use state space exploration to check if a process that wishes to do so (reaches `a3` or `b3`) will eventually reach its critical section.

- To do this, one must, for example, check that every path starting from a state in which Process 1 is at `a3` will eventually reach `a5`.

- However, one can see that this is not always the case. Why ?

Fairness

- The correct operation of Dekker's algorithm depends on the fact that each process will always eventually progress.

- This property depends on the process scheduler, all the detail of which are not always known.

- We will not model the process scheduler, but we will correct the interleaving model with a constraint that represent a minimal condition that has to be satisfied by any process scheduler: a *fairness hypothesis*.

- A simple fairness hypothesis that is sufficient to reason about Dekker's algorithm is the following.

  Every process that has the possibility of executing an instruction will always eventually do so.

Safety properties - Liveness properties

Among the properties that can be specified for a concurrent programs one distinguishes the two following categories.

- **The safety properties**. These are the properties that specify the undesirable states are never reached. They do not depend on a fairness hypothesis.

  The liveness properties. These specify that desirable states will inevitably be reached. Usually, these properties are true only if a fairness hypothesis is taken into account.

The limitations of shared memory

- Even if it is possible, synchronizing processes using only shared memory is a difficult.
- Indeed, an algorithm such as Dekker's is non obvious and is not easily extended to more than two processes.
- Another drawback of an algorithm such as Dekker's is that a waiting process still uses the CPU.
- It would be very useful to have another synchronization mechanism that makes a better implementation of mutual exclusion possible.
- We will study *semaphores* which appear as shared variables only accessible through specific system calls.

The semaphore concept

A semaphore is a shared integer variable. Its value is positive or 0 and it can only be accessed through the two operations `wait(s)` and `signal(s)`, where `s` is an identifier representing the semaphore.

- `wait(s)` decrements `s` if `s > 0` ; if not, the process executing the operation `wait(s)` is suspended.

- `signal(s)` increments `s`. The execution of `signal(s)` can have as result (possibly delayed) that a process waiting on the semaphore `s` resumes its execution. Executing a `wait(s)` or a `signal(s)` operation is done without any possible interaction (atomically).

Mutual exclusion with semaphores

Semaphores make a very simple implementation of mutual exclusion possible.

```
semaphore s = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         wait(s);
  crit1: /* critical section */ ;
         signal(s);
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         wait(s);
  crit2: /* critical section */
         signal(s)
}
```

Mutual exclusion and semaphores: notes

- The solution just shown can be directly generalized to any number of processes.

- If the semaphore is initialized to a value k other than 1, one obtains a solution that allows k processes to be simultaneously in their critical section.

- The correct operation of mutual exclusion implemented with semaphores requires fairness in the handling of the processes suspended by the `wait` operation.