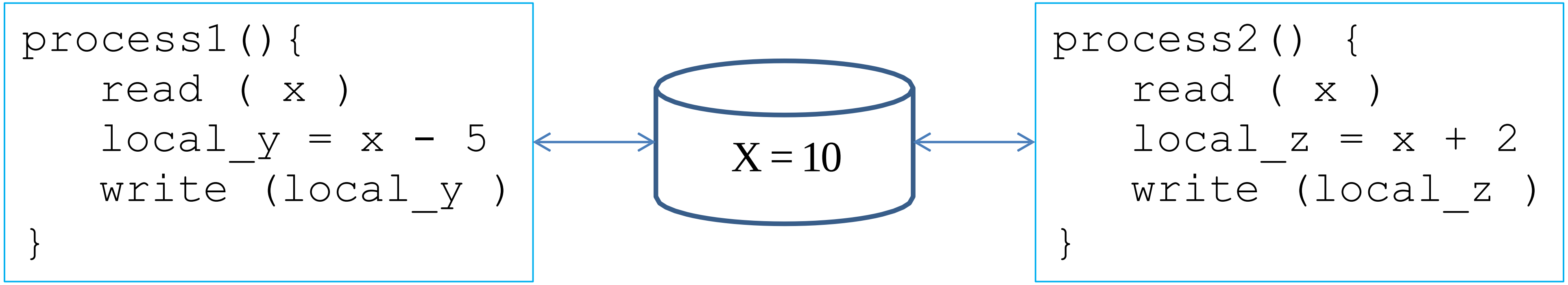
Synchronisation of Processes

- Mutual Exclusion
  - Avoid simultaneous access to resources
    - Ensure that only one process at a time may execute a “critical” course of actions (read and write of shared resource)
- Condition synchronisation
  - Enforce a strict sequence of actions across processes
    - Processes wait for particular conditions to hold, before they proceed with execution

Example



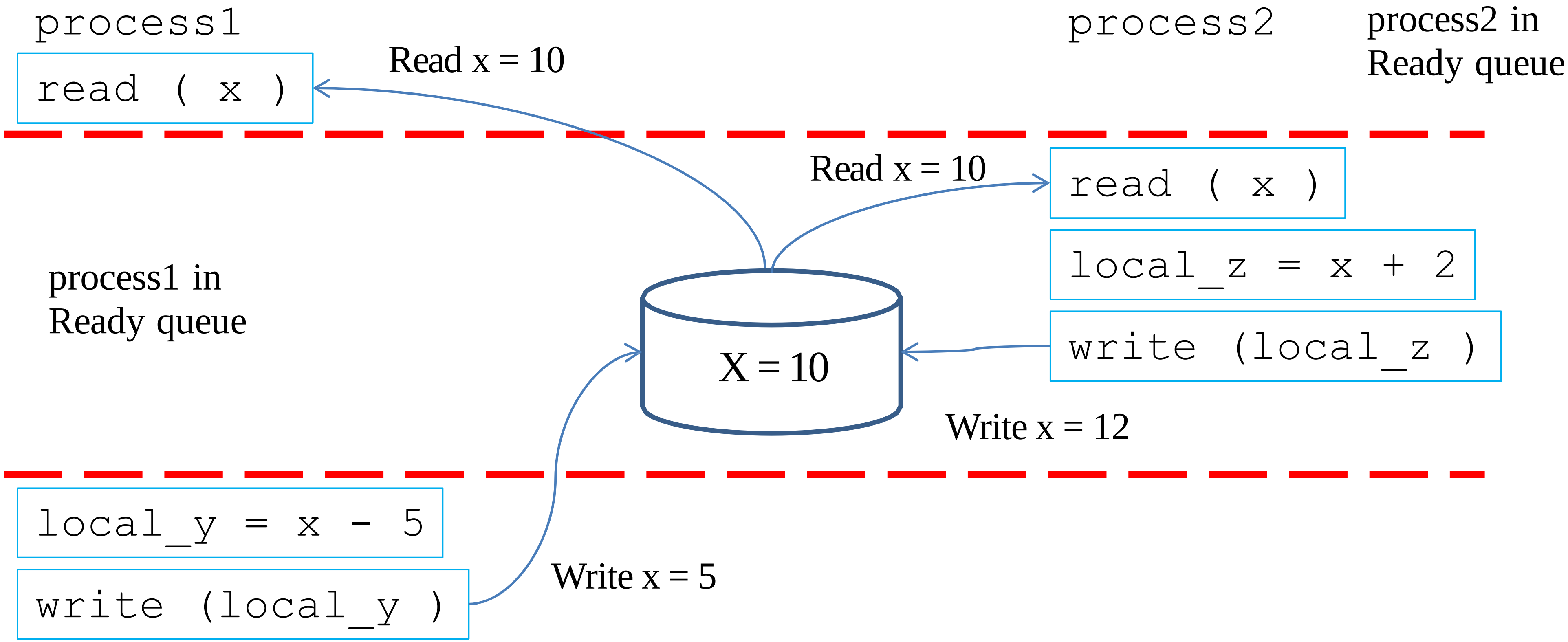
```
process1(){
    read ( x )
    local_y = x - 5
    write (local_y )
}
```

X = 10

```
process2() {
    read ( x )
    local_z = x + 2
    write (local_z )
}
```

- We expect
  - –When process1 finishes, shared variable x is reduced by 5
  - –When process2 finishes, shared variable x is increased by 2

Example



`process1`

`read ( x )`

Read x = 10

`process2`

process2 in
Ready queue

Read x = 10

`read ( x )`

`local_z = x + 2`

`write (local_z )`

process1 in
Ready queue

X = 10

Write x = 12

`local_y = x - 5`

`write (local_y )`

Write x = 5

- Context switches may occur at any time
- Process 2 has its result overwritten by process 1
- Process 1 operates with outdated information

Race Condition

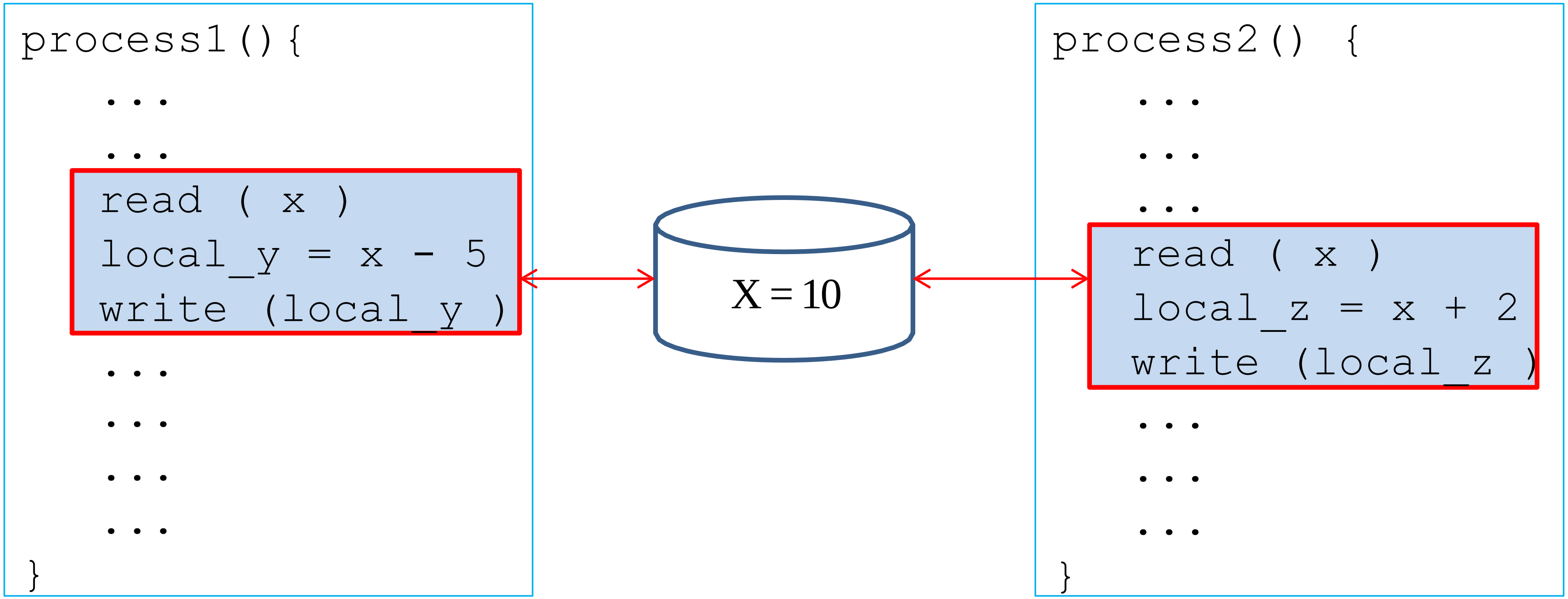
- Occurs when multiple processes / threads read and write shared data items
- The processes "race" to perform their read/ write actions
- The final result depends on the order of execution
  - The "loser" of the race is the process that performs the last update and determines the final value of a shared data item

Race Condition

- Why do race conditions occur
  - “whenever the state of a shared resource depends on the precise execution order of the processes”
  - Scheduling: Context switches at arbitrary times during execution
  - Outdated Information: Processes / Threads operate with “stale” copies of memory values in registers / local variables
    - Other processes may already have changed the original value in the shared memory location
- How can we avoid race conditions?

Critical Section

- Critical Section
  - Part of the program code that accesses a shared resource



```
process1(){
    ...
    ...
    read ( x )
    local_y = x - 5
    write (local_y )
    ...
    ...
    ...
    ...
}
```

```
process2() {
    ...
    ...
    ...
    read ( x )
    local_z = x + 2
    write (local_z )
    ...
    ...
    ...
}
```

Critical Section

- Critical Section
  - Part of the program code that accesses shared resource
- A program will consist of critical and non critical sections
- In order to avoid race conditions, we have to control the concurrent execution of critical sections
  - Strict serialisation – mutual exclusion

Critical Section

```
process ()
{
   entry_protocol()
      critical_section()
   exit_protocol()
}
```

- Entry protocol:
  - Process requests entry to critical section
  - Process has to communicate that it entered critical section
- Exit protocol:
  - process communicates to other processes that it leaves critical section

The Critical Section Problem

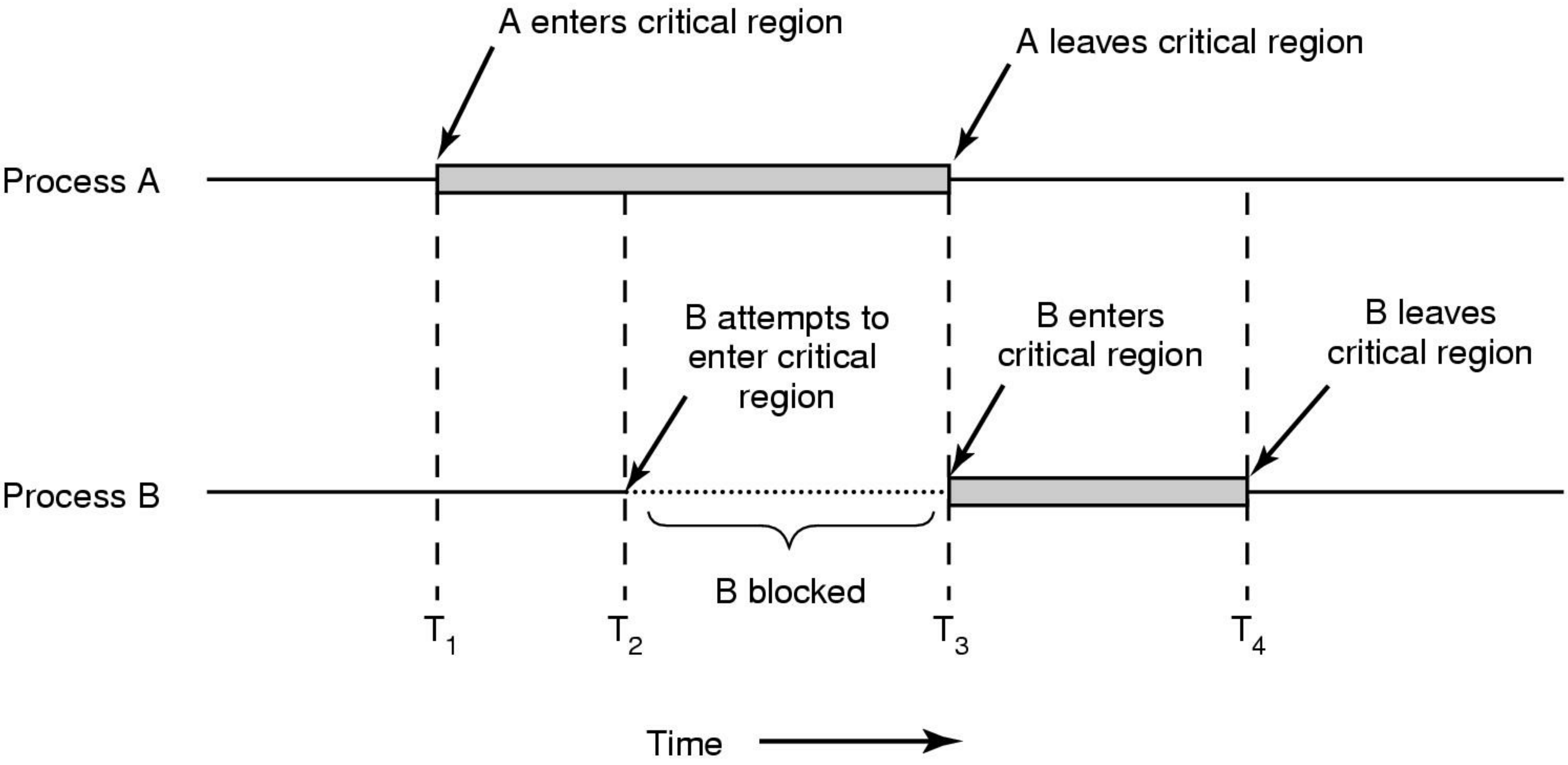
- Avoid race conditions by enforcing mutual exclusion between processes
- Control entry to and exit from critical section
  - We need a Critical Section Protocol:
    - Entry section: Each process must request permission for entering a critical section
      - Requires Interprocess communication
      - has to wait / is suspended until entry is granted
    - Exit section:
      - Requires interprocess communication
      - process communicates that it leaves critical section
- Avoid deadlock and starvation:
  - Enforcing mutual exclusion may result in deadlocks and starvation – has to be solved

Achieve Mutual Exclusion

- Arrange the execution of processes such that
  - Mutual Exclusion: only one of them is executing its critical section.
  - Handle scheduler preemption: This one process can finish the execution of its critical section, even if it is preempted or interrupted
  - Any other process sharing the resource has to wait or is blocked, in the meantime, from accessing it

Mutual Exclusion

- Mutual Exclusion during critical sections



Deadlock and Starvation

- Enforcing mutual exclusion creates two new problems
  - Deadlocks
    - Processes wait forever for each other to free resources
  - Starvation
    - A process waits forever to be allowed to enter its critical section
- Implementing mutual exclusion has to account for these problems

Solutions

- Software
  - Use shared lock variables to control access to critical section
  - Busy waiting
- Hardware
  - Disable interrupts
  - Processor provides special instructions
- Higher operating system constructs
  - Semaphores, Monitor, message passing
  - Involvement of scheduler, processes are suspended

Software Solutions for Mutual Exclusion

Solving the Critical Section Problem

Requirements

for Solutions to the Critical Section Problem, Mutual Exclusion

- Serialisation of access:
  - Only one process at a time is allowed in the critical section for a resource
- Progress (Liveness, no deadlock):
  - A process that halts in its noncritical section must do so without interfering with other processes currently waiting to enter their critical section
  - Only processes currently waiting to enter their critical section are involved in the selection of the one process that may enter
  - A process remains inside its critical section for a finite time only
- Bounded waiting (no starvation):
  - A process waiting to enter a critical section, must be guaranteed entry (with some defined limited waiting time)
    - Scheduling algorithm has to guarantee that process is eventually scheduled and can progress

Solution to Critical Section Problem

- Critical sections must be protected by some form of a "lock"
- Lock
  - A shared data item
  - Processes have to "acquire" such a lock before entering a critical section
  - Processes have to "release" a lock when exiting critical section

```
process ()
{
   acquire lock

        critical_section() ;

   release lock

        remainder_section() ;
}
```

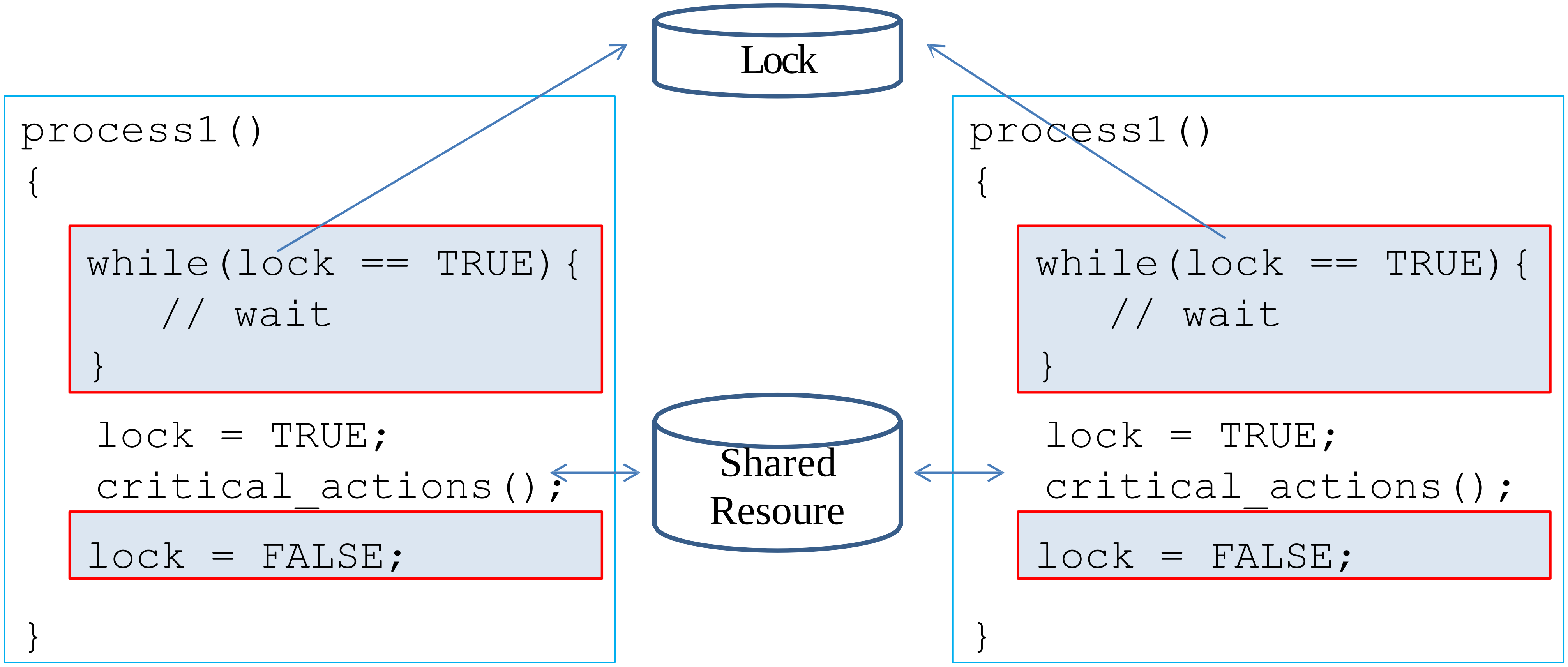
Lock Variables

- Use of shared memory for interprocess communication
- Shared variable “lock”, also called a “mutex”
- Used to indicate whether one of the competing processes has entered critical section
  - If lock == 0 (FALSE), then lock is not set
  - If lock == 1 (TRUE), then lock is set
- All processes that compete for a shared resource, also share this lock variable
  - A process checks the lock
    - If lock is not set, process sets lock and enters critical section
    - If lock is set, process waits
- Problem
  - As lock variable is itself a shared resource, race conditions can occur

Shared Lock / Mutex

- A shared lock (shared variable) is used
- Two states: TRUE ... Critical section is locked, FALSE ... Critical section is unlocked

Initialisation: `lock = FALSE;`

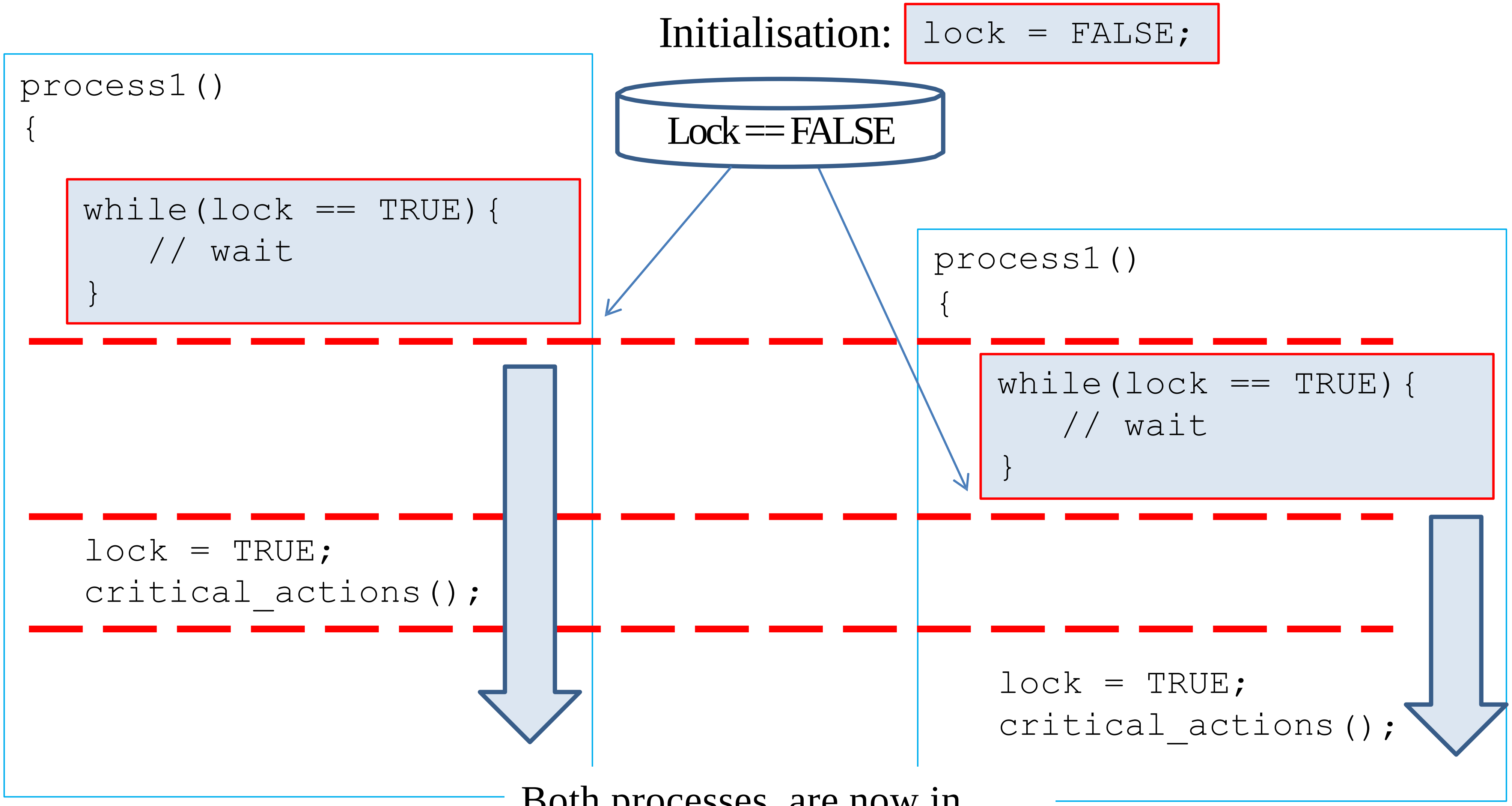


```
process1()
{
  while(lock == TRUE){
     // wait
  }
  lock = TRUE;
  critical_actions();
  lock = FALSE;

}
```

```
process1()
{
  while(lock == TRUE){
     // wait
  }
  lock = TRUE;
  critical_actions();
  lock = FALSE;

}
```

Shared Locks, Problem

- Context switch, no mutual exclusion



Initialisation:

```
lock = FALSE;
```

```
process1()
{
   while(lock == TRUE){
      // wait
   }

   lock = TRUE;
   critical_actions();
```

```
process1()
{
   while(lock == TRUE){
      // wait
   }

   lock = TRUE;
   critical_actions();
```

Both processes are now in their critical sections !

Implementing Mutual Exclusion

- Busy waiting
  - Also called “polling” or “spinning”
    - A process continuously evaluates whether a lock has become available
    - Lock is represented by a data item held in a shared memory (IPC via shared memory)
    - Process consumes CPU cycles without any progress
  - May impact on performance on singleprocessor systems
    - A process busywaiting may prevent another process holding the lock from executing and completing its critical section and from releasing the lock
  - Can be implemented at application level, established algorithms exist that guarantee mutual exclusion, independence from operating system
  - Spin locks are used at kernel level (special HW instructions)

Software Solutions

Strict Alternation

- Busywaiting Strategy
  - Process waits for its turn
- Strict alternation between two processes
  - Use a “token” as shared variable:
    - value is process ID
    - indicates which process is the next to enter critical section, set by previous process
- For two processes P0 and P1 (can be extended to n processes)
- Entry to critical section
  - Process P_i busywaits until token == i (its own process ID)
- Exit from critical Section
  - Process P_i sets token to next process ID

Strict Alternation

Process 0

```
while(TRUE){

    while(turn != 0){
        // wait
    }

      Critical_Section

    turn = 1;

      Non_Critical_Section
      ...
}
```

Global Variable

```
int turn ;
```

Process 1

```
while(TRUE){

    while(turn != 1){
        // wait
    }

      Critical_Section

    turn = 0;

      Non_Critical_Section
      ...
}
```

- Mutual exclusion guaranteed
- Lifeness / Progression problem:
  - Both process depend on a change of the “turn” variable
  - If one of the processes is held up in its noncritical section, it cannot do that and will block the other process

Problem of Strict Alternation

- Violates the progress requirement:
  - Shared variable “turn” is only altered in critical section
  - One process may be held up in its noncritical section
  - This eventually blocks the other process, as the shared variable “turn” is not altered any more
- Alternative approach:
  - Processes announce that they want to enter critical section with flags, one flag per process

Use an Array of Flags

- Busywaiting Strategy
  - Process waits for entering critical section
- Processes announce that they want to enter critical section
  - Use of flags, one flag per process
    - Flag_i == TRUE: process i wants to enter critical section
    - Flag_i == FALSE: process i is outside critical section

Use an Array of Flags

Global Variables

```
boolean flag[2];
```

```
flag[0]=FALSE
flag[1]=FALSE
```

Process 0

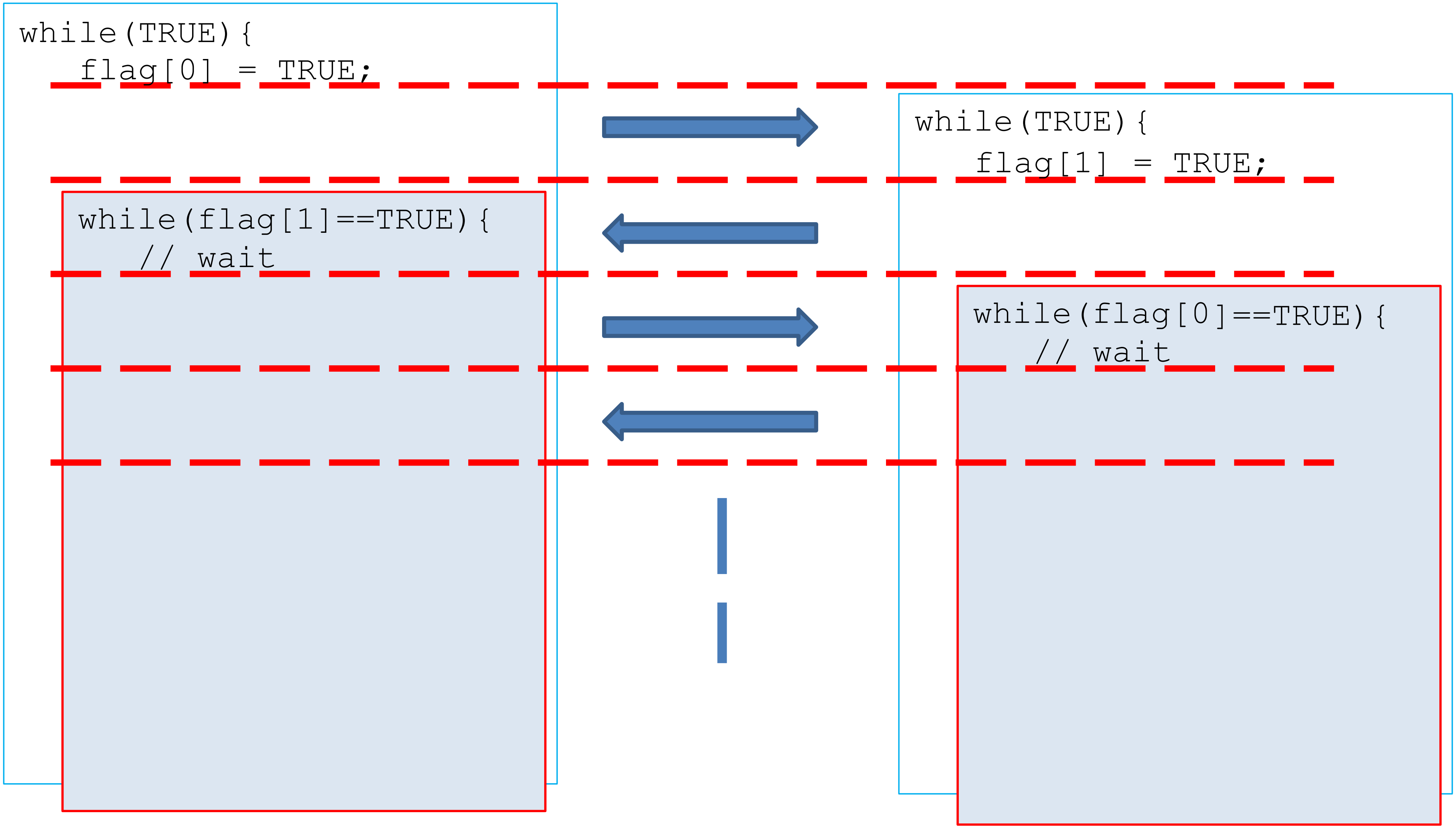
```
while(TRUE){
   flag[0] = TRUE;
   while(flag[1]==TRUE){
      // wait
   }
     Critical_Section
   flag[0] = FALSE;
     Non_Critical_Section
     ...
}
```

Process 1

```
while(TRUE){
   flag[1] = TRUE;
   while(flag[0]==TRUE){
      // wait
   }
     Critical_Section
   flag[1] = FALSE;
     Non_Critical_Section
     ...
}
```

- Mutual exclusion guaranteed
- Problem: Deadlock may occur due to context switch

Use an Array of Flags
Deadlock



Process 0

```
while(TRUE){
   flag[0] = TRUE;
```

```
while(flag[1]==TRUE){
   // wait
```

Process 1

```
while(TRUE){
   flag[1] = TRUE;
```

```
while(flag[0]==TRUE){
   // wait
```

Dekker's Algorithm

- Busywaiting Strategy
  - Process waits for entering critical section
- Use of shared memory variables for communication between processes
- Works for two processes
- Combines strict alternation with using flags for announcing entry into CS
- Avoids progression, deadlock and starvation issues
  - Use of flags to indicate intention to enter CS
  - Use of "turn" variable for specifying which process is supposed to enter the CS

Dekker's Algorithm

Global Variables

```
boolean flag[2];
int turn;
```

```
flag[0]=FALSE
flag[1]=FALSE
```

```
turn = 0; // or 1
```

Process 0

```
P0:
   flag[0] = TRUE;
   while(flag[1] == TRUE){
      if(turn == 1){
         flag[0]=FALSE
         while(turn == 1){
            // wait
         }
         flag[0]=TRUE;
      }
   }
     Critical_Section
   turn = 1;
   flag[0] = FALSE;
     Non_Critical_Section
     ...
```

Process 1

```
P1:
   flag[1] = TRUE;
   while(flag[0] == TRUE){
      if(turn == 0){
         flag[1]=FALSE
         while(turn == 0){
            // wait
         }
         flag[1]=TRUE;
      }
   }
     Critical_Section
   turn = 0;
   flag[1] = FALSE;
     Non_Critical_Section
     ...
```

Dekker’s Algorithm

- Enter critical section
    - If two processes attempt to enter critical section, one process will be allowed to enter, based on “turn” variable
    - If one process is already in critical section, the other will busywait, based on flags
        - Waiting process is also temporarily setting its own flag to FALSE to let other process proceed

Dekker's Algorithm

- Scenario1:
  - Process 0 wants to enter critical section, process 1 has not entered, flag[1] == FALSE
    - Sets flag[0]=TRUE
    - Checks process 1 flag: flag[1]==TRUE or FALSE?
      - Flag[1] == FALSE, process 1 has not entered : process 0 enters critical section
- Scenario 2:
  - Process 0 wants to enter critical section, context switch to process 1
    - Sets flag[0]=TRUE
    - Context switch: process 1 has entered, flag[1] == TRUE
    - Process 1 checks process 0 flag: flag[0]==TRUE
      - Turn == 0: it is process 0's turn, process 1 waits, process 0 enters critical section
      - Turn == 1: it is process 1's turn, process 0 busywaits, also resets its flag[0] so that process 1 can enter critical section
- Scenario 3:
  - Process 0 wants to enter critical section, process 1 has entered, flag[1] == TRUE
    - Context switch to process 0
    - Sets flag[0]=TRUE
    - Checks process 1 flag: flag[1]==TRUE or FALSE?
      - Flag[1] == TRUE: process 1 also tries to enter critical section
      - Turn == 0: it is process 0's turn, process 0 will loop until flag[1] == FALSE, process 1 enters critical section
      - Turn == 1: it is process 1's turn, process 0 busywaits, also resets its flag[0] so that process 1 can enter critical section

Peterson’s Algorithm

- Is equivalent to Dekker’s algorithm
  - Combines strict alteration with flags for indicating interest in entering critical section
  - Simpler than Dekker’s algorithm

Peterson's Algorithm

- Peterson's Solution
  - NonAtomic Locking: works even if there is a race condition
  - Is limited to two processes coordinating their access to critical sections
  - Uses two shared data items for coordinating access to critical section (changes seen by both processes)

```
process ( i )
{
    j = 1 -i ;
    flag[i] = TRUE ;
    turn = j ;
    while (flag[j] &&
               turn == j ) ;

       critical_section() ;

    flag[i] = FALSE ;

       remainder_section() ;
}
```

```
int turn ;
```

Indicates, which of the two processes is allowed to enter

```
boolean flag[2] ;
```

Indicates, which of the two processes is ready to enter (both can be ready at the same time)

Peterson’s Algorithm

Global Variables

```
boolean flag[2];
int turn;
```

```
flag[0]=FALSE
flag[1]=FALSE
```

```
turn = 0; // or 1
```

Process 0

```
while(TRUE) {
    flag[0] = TRUE;
    turn = 1
    while(flag[1] == TRUE &&
              turn == 1){
      // wait
    }
    Critical_Section
    flag[0] = FALSE;
      Non_Critical_Section
      ...
}
```

Process 1

```
while(TRUE) {
    flag[1] = TRUE;
    turn = 0
    while(flag[0] == TRUE &&
              turn == 0){
      // wait
    }
    Critical_Section
    flag[1] = FALSE;
      Non_Critical_Section
      ...
}
```

Peterson’s Algorithm

- Initially:
  - No process in critical region
    - turn = 0, flag[0] = FALSE, flag[1] = FALSE
- Process 0 tries to enter critical section
  - Sets turn = 1 (other process), sets interested[0] =TRUE
  - As flag[1] == FALSE, process enters critical section
- Process 1 tries to enter critical section
  - Sets turn = 0 (other process), flag[1] = TRUE,
  - As flag[0] == TRUE && turn == 0, process waits, until process 0 finishes
- Process 0 exit
  - Sets flag[0] = FALSE
- Process 1 enters critical section ...

Peterson’s Algorithm

- Does it work if both processes enter almost simultaneously?
  - Both will set flag[processID] = TRUE
  - Both try to write the variable turn
  - This is a race condition: if Process 0 is the last to write, it loses the race and will not enter its CS as turn = 1 (Process 0 is really a loser!:)
    - Example: Process 1 wins the race, turn = 1 (set by Process 0)
    - Both processes arrive at the while loop
      - Process 1 immediately continues (as turn = 1)
      - Process 0 is waiting in the while loop (as turn = 1 and flag[1] = TRUE)
- The race condition is not a problem
  - If there is a race condition in terms of updating the shared variable “turn”, one of the two processes will win and be the one to enter the critical section

Peterson's Algorithm
Race Condition

Global Variables

```
boolean flag[2];
int turn;
```

```
flag[0]=FALSE
flag[1]=FALSE
```

```
turn = 0; // or 1
```

Process 0

```
while(TRUE) {
    flag[0] = TRUE;
    turn = 1
    while(flag[1] == TRUE &&
                turn == 1){
      // wait
    }
    Critical_Section
    flag[0] = FALSE;
      Non_Critical_Section
      ...
}
```

Process 1

```
while(TRUE) {
    flag[1] = TRUE;
    turn = 0
    while(flag[0] == TRUE &&
                turn == 0){
      // wait
    }
    Critical_Section
    flag[1] = FALSE;
      Non_Critical_Section
      ...
}
```

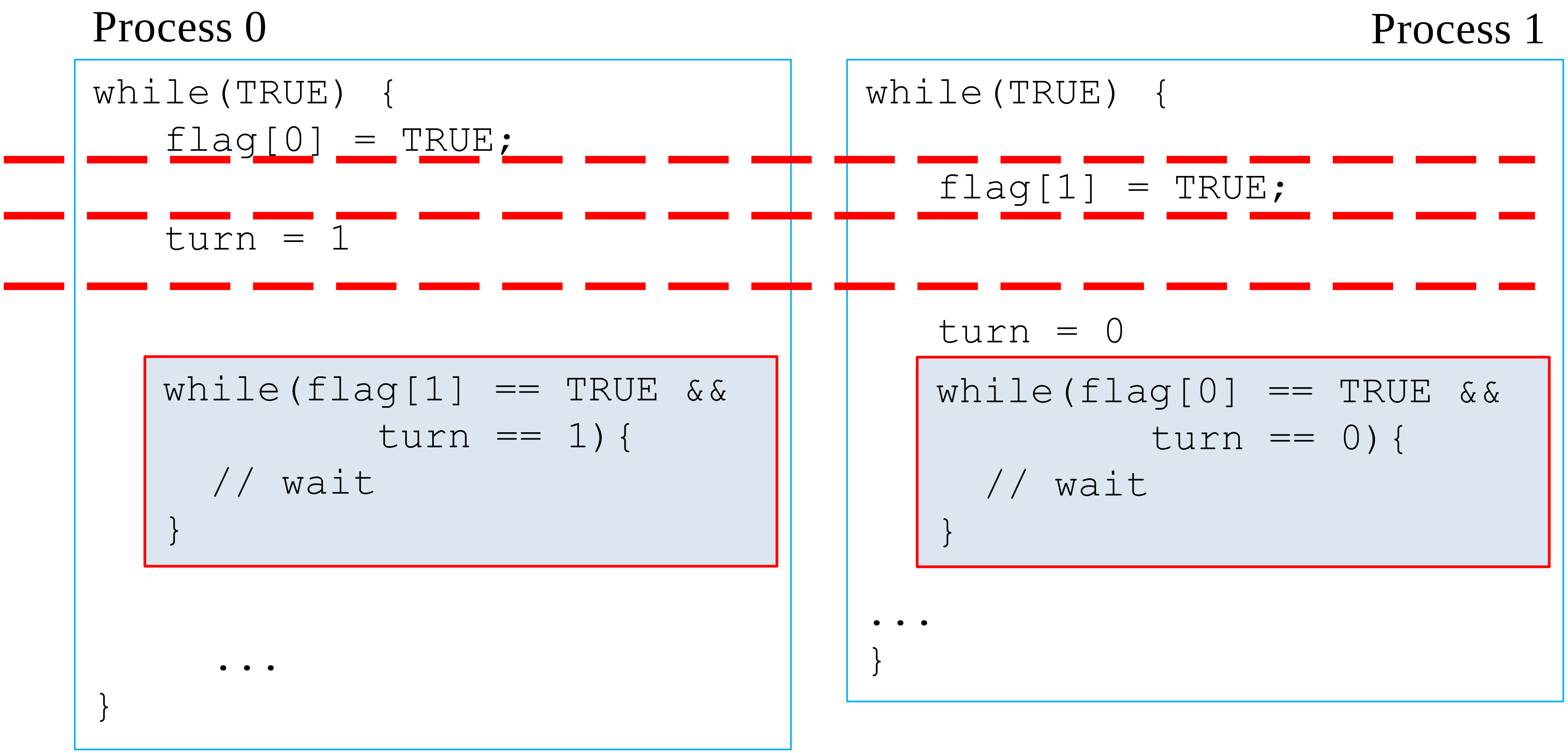
Peterson's Algorithm
Race Condition

Global Variables

```
boolean flag[2];
int turn;
```

```
flag[0]=FALSE
flag[1]=FALSE
```

```
turn = 0; // or 1
```



Process 0

```
while(TRUE) {
    flag[0] = TRUE;

    turn = 1


    while(flag[1] == TRUE &&
                turn == 1){
       // wait
    }


       ...
}
```

Process 1

```
while(TRUE) {

    flag[1] = TRUE;


    turn = 0
    while(flag[0] == TRUE &&
                turn == 0){
       // wait
    }
...
}
```

Peterson’s Algorithm

- Peterson’s Algorithm
  - Is a nonatomic locking algorithm
  - Mutual Exclusion is preserved
    - Even if flag[i] == flag[j] == TRUE (both processes are ready), the variable turn can only be either i or j (only one of them can enter critical section)
  - Progress and Bounded Waiting
    - Progress is guaranteed: If a process indicates interest to enter critical section, it will gain access after the other process is finished
- Problems
  - Solution for only two processes, can be extended to n processes, does not work for unknown number of processes