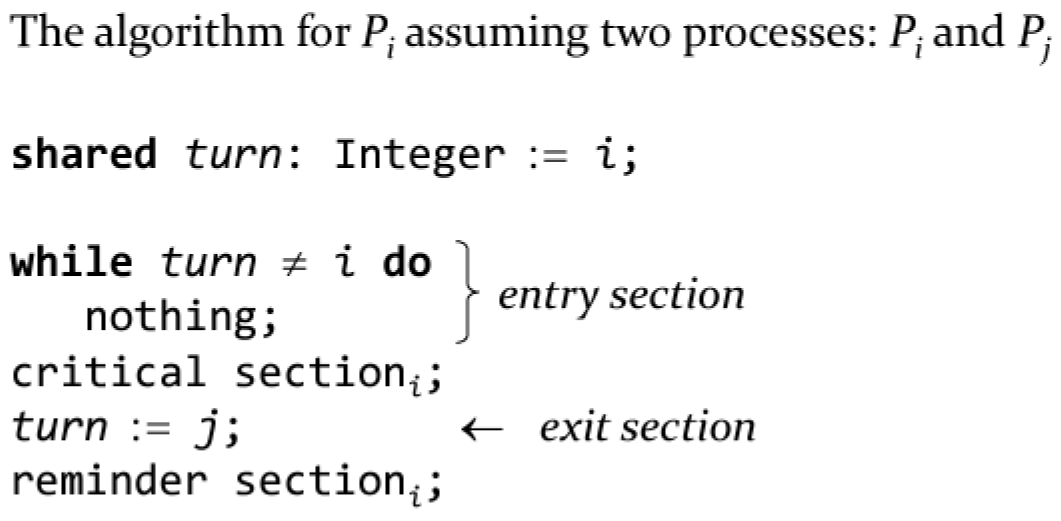
Concurrency additional material

From the book:

Solution 1



The algorithm for P_i assuming two processes: P_i and P_j

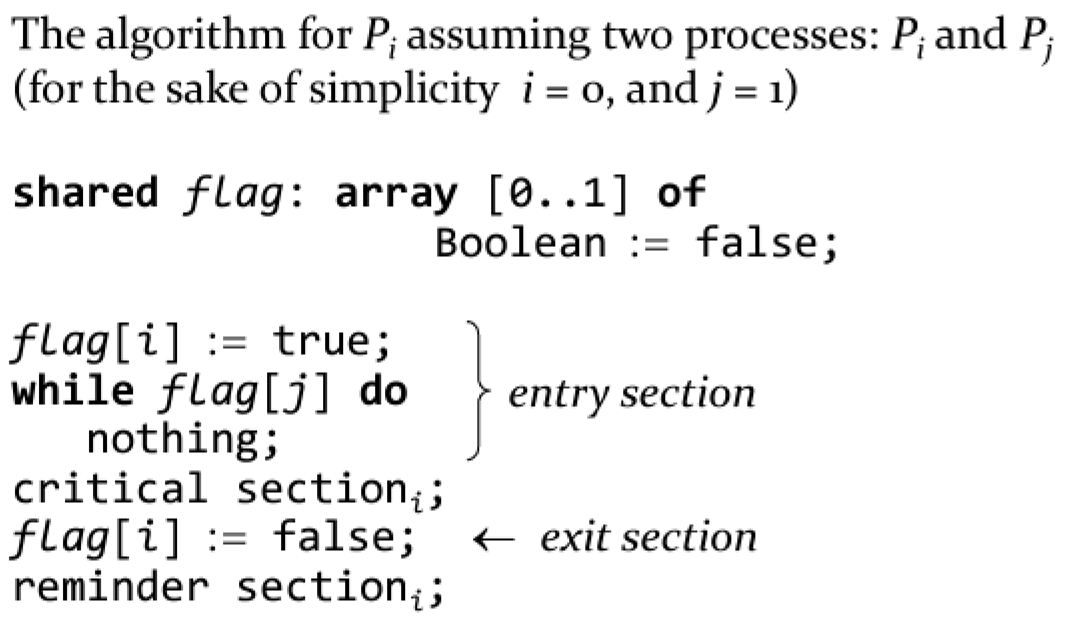
```
shared turn: Integer := i;

while turn ≠ i do      } entry section
   nothing;
critical section_i;
turn := j;             ← exit section
reminder section_i;
```

- Correctness of the solution
  - ✅Mutual exclusion
  - ✅Progress
  - ✅Lockout-freedom
- Explanation:

  This solution has some drawbacks. For example if one of the processes dies while in the critical section, then the other one will never enter it.

  Warning: If Wawrzyniak doesn't assume that a process needs to complete the critical section (for example it could crash) then the progress and lockout-freedom properties do not hold.

Solution 2



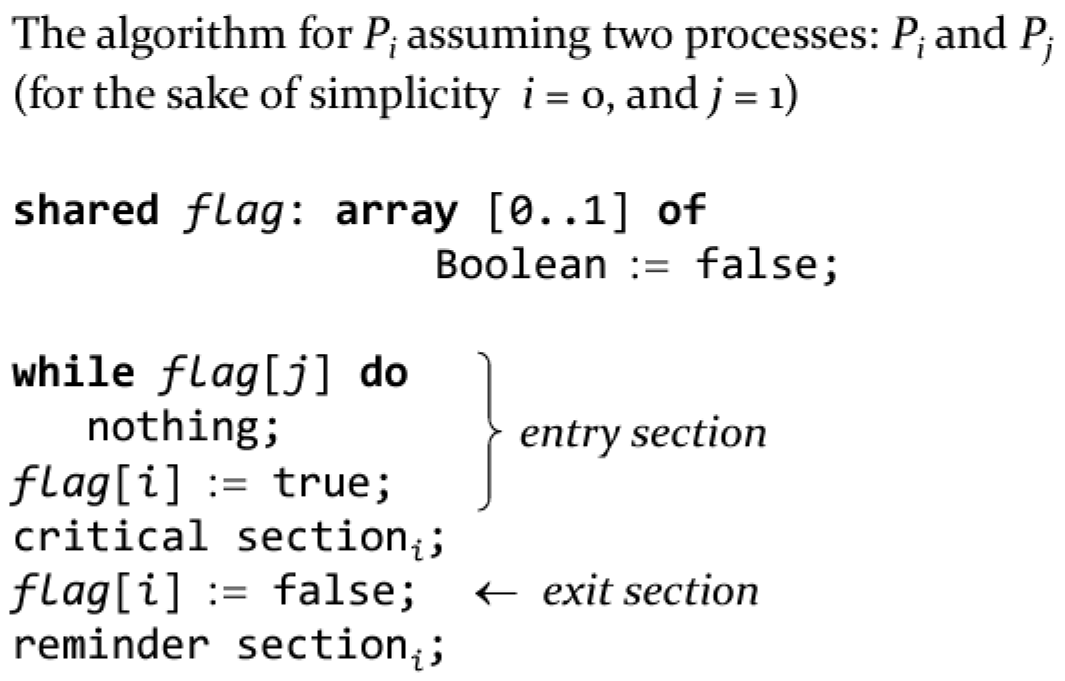
The algorithm for P_i assuming two processes: P_i and P_j
(for the sake of simplicity $i = 0$, and $j = 1$)

```
shared flag: array [0..1] of
                    Boolean := false;

flag[i] := true;         }
while flag[j] do         } entry section
   nothing;              }
critical section_i;
flag[i] := false;      ← exit section
reminder section_i;
```

- Correctness of the solution
    - ✅Mutual exclusion
    - ❌Progress
    - ❌Lockout-freedom
- Explanation:

  If one thread will set it’s flag to true, and another thread will set it’s own flag to true right after the first one then no thread will be able to proceed.

Solution 2 — modification



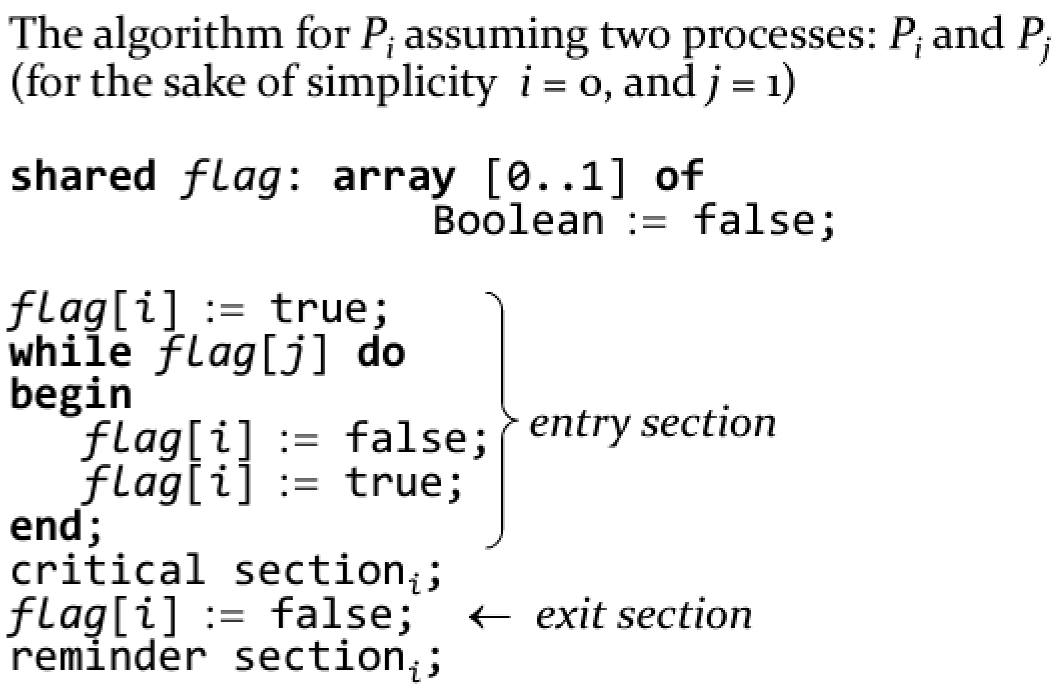
The algorithm for P_i assuming two processes: P_i and P_j (for the sake of simplicity $i = 0$, and $j = 1$)

```
shared flag: array [0..1] of
                        Boolean := false;

while flag[j] do        }
   nothing;             } entry section
flag[i] := true;        }
critical section_i;
flag[i] := false;       ← exit section
reminder section_i;
```

- Correctness of the solution
  - ❌Mutual exclusion
  - ✅Progress ??? Not sure
  - ✅Lockout-freedom ??? Not sure
- Explanation:

  If both thread enter the while loop simultaneously then both will proceed to the critical section at the same time, thus the mutual exclusion property doesn't hold.

Solution 3



The algorithm for P_i assuming two processes: P_i and P_j (for the sake of simplicity $i = 0$, and $j = 1$)

```
shared flag: array [0..1] of
                  Boolean := false;

flag[i] := true;           ⎫
while flag[j] do           ⎪
begin                      ⎪
   flag[i] := false;       ⎬ entry section
   flag[i] := true;        ⎪
end;                       ⎭
critical section_i;
flag[i] := false;          ← exit section
reminder section_i;
```

- Correctness of the solution
  - ✅Mutual exclusion
  - ❌Progress ??? Not sure
  - ❌Lockout-freedom
- Explanation:

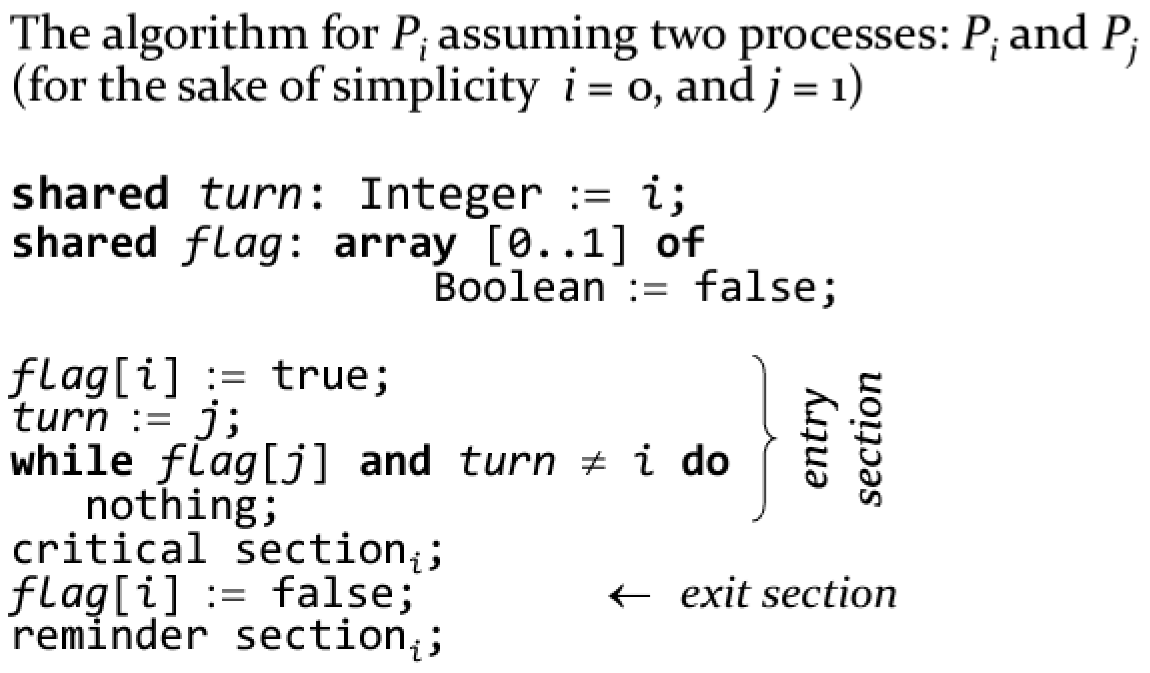
  While the mutual exclusion property hold there is an interleaving that results in a infinite sequence of checking the flag and unchecking. Consider:

Process 1	Process 2
`flag = true`	
	`flag = true`
`flag = false`	
	`flag = false`
`flag = true`	
	`flag = true`

  This sequence could be extended infinitely, therefore neither the lockout-freedom property nor the progress property holds.

My solutions:

Solution 4

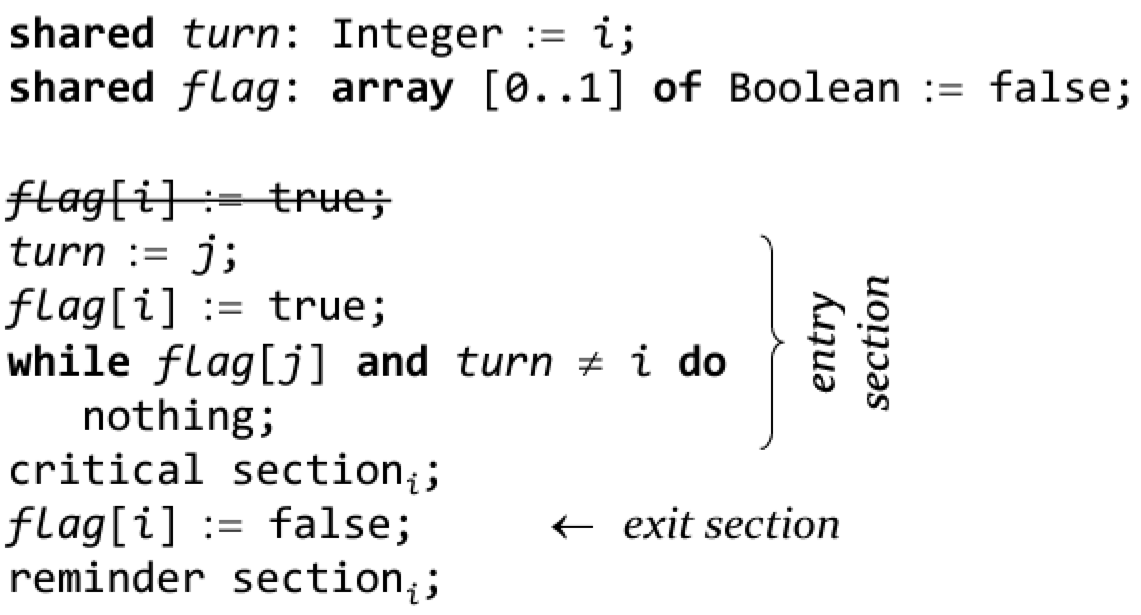


The algorithm for P_i assuming two processes: P_i and P_j (for the sake of simplicity $i = 0$, and $j = 1$)

```
shared turn: Integer := i;
shared flag: array [0..1] of
                 Boolean := false;

flag[i] := true;                        }
turn := j;                              }  entry
while flag[j] and turn ≠ i do           }  section
   nothing;                             }
critical section_i;
flag[i] := false;            ← exit section
reminder section_i;
```

- Correctness of the solution
  - ✅Mutual exclusion
  - ✅Progress
  - ✅Lockout-freedom

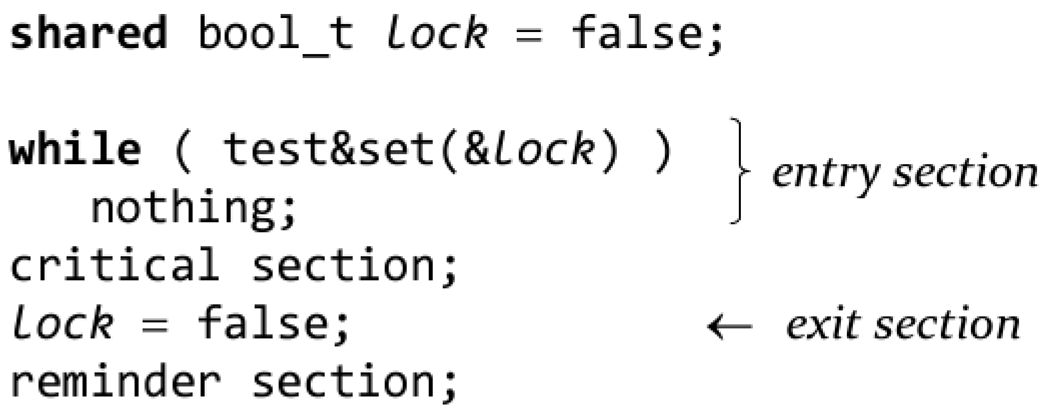
Solution 4 — modification



```
shared turn: Integer := i;
shared flag: array [0..1] of Boolean := false;

~~flag[i] := true;~~
turn := j;
flag[i] := true;
while flag[j] and turn ≠ i do
   nothing;
critical section$_i$;
flag[i] := false;
reminder section$_i$;
```

- Correctness of the solution
  - ❌Mutual exclusion
  - ✅Progress
  - ✅Lockout-freedom

Test and set solution



```
shared bool_t lock = false;

while ( test&set(&lock) )
   nothing;
critical section;
lock = false;
reminder section;
```

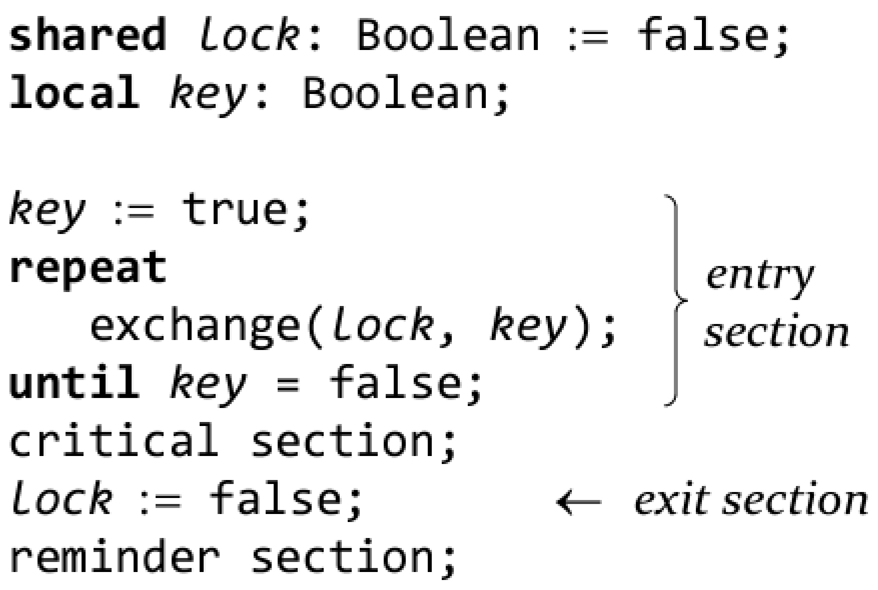
- Correctness of the solution
  - ✅Mutual exclusion

- ✅Progress
- ❌Lockout-freedom

- Explanation:

  The lockout-freedom property is not fulfilled since there is no fairness mechanism. One thread can happen to always be the first to lock the shared variable, therefore the second thread will never get a chance to enter the critical section.

Exchange solution



```
shared lock: Boolean := false;
local key: Boolean;

key := true;                         }
repeat                               }  entry
    exchange(lock, key);             }  section
until key = false;                   }
critical section;
lock := false;               ← exit section
reminder section;
```

- Correctness of the solution
  - ✅Mutual exclusion
  - ✅Progress
  - ❌Lockout-freedom
- Explanation:

  Same as above.