Part III
Synchronization
Software and Hardware Solutions

*Computers are useless. They can only give answers.*

*Pablo Picasso*
Software Solutions for Two Processes

- Suppose we have two processes $P_0$ and $P_1$.
- Let one process be $P_i$ and the other be $P_j$, where $j = 1 - i$. Thus, if $i = 0$, then $j = 1$ and if $i = 1$, then $j = 0$.
- We will design enter-exit protocols for a critical section to ensure mutual exclusion.
- We will go through a few unsuccessful attempts and finally yield a correct one.
- These solutions are pure software-based.
An Important Assumption: 1/3

- We have the following assumption*: 
  
  ➢ Inspecting the current value of a shared variable and assigning a new value to such a shared variable are to be regarded as indivisible, non-interfering actions (i.e., atomic).

An Important Assumption: 2/3

- **What does this mean?**

  - When two processes assign a new value to the same shared variable simultaneously, the assignments are done sequentially.

  - When a process checks the value of a shared variable with an assignment to it by the other one, the former process will find either the old or the new value.

  - These variables could be in registers.

  - However, expression evaluation is **NOT** atomic.
Co-operating Sequential Processes

E. W. DIJKSTRA
Department of Mathematics, Technological University, Eindhoven, The Netherlands

INTRODUCTION

This chapter is intended for all those who expect that in their future activities they will become seriously involved in the problems that arise in either the design or the more advanced applications of digital information processing equipment; they are further intended for all those who are just interested in information processing.

The applications are those in which the activity of a computer must include the proper reaction to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in process control, traffic control, stock control, banking applications, automatization of information flow in large organizations, centralized computer service, and, .
A Few More Assumptions

- The following assumptions are made about the behavior of the processes

  - Nothing is assumed about the remainder code except that it cannot influence the behavior of other processes.
  - Shared objects in an entry or an exit code may not be referred to in a remainder code of a critical section.
  - A process cannot fail or loop while executing the entry code, critical section and exit code. Whenever it is scheduled it must take a step.
  - A process can take only a finite number of steps in its critical section and exit code.
  - While the collection of processes is concurrent, individual processes are sequential.
Attempt I: 1/3

- Shared variable `turn`, initialized to `i` or `j`, controls who can enter the critical section.
- Since `turn` is either `i` or `j`, only one can enter.
- However, processes are forced to run in an alternating way.
- Not good!

```
process P_i

do {
    if it is not my turn, I wait
    while (turn != i);
    enter
    critical section
    turn = j; exit
} while (1);

I am done, it is your turn now
```
Attempt I: 2/3

- **Mutual Exclusion**
  - $P_i$ in its CS if $\text{turn} = i$.
  - $P_j$ in its CS if $\text{turn} = j$.
  - If $P_i$ and $P_j$ are BOTH in their CSs, then $\text{turn} = i$ and $\text{turn} = j$ must BOTH be true.
  - This is absurd, because a variable can only hold one and only one value (i.e., cannot hold both $i$ and $j$) at any time.

### Process $P_i$

```
process $P_i$
do {
    if it is not my turn, I wait
    while (turn != i);
    critical section
    turn = j;
    exit
}
```
Attempt 1: 3/3

process \( P_i \)

\[
\text{do } \{ \text{if it is not my turn, I wait} \\
\text{while} \ (\text{turn} \neq i); \text{enter} \\
\text{critical section} \\
\text{turn} = j; \text{exit} \\
\} \text{ while (1);}
\]

I am done, it is your turn now

- **Progress**
  - If \( P_i \) sets turn to \( j \) on exit and will not use the critical section for some time, \( P_j \) can enter but cannot enter again.

- An irrelevant process blocks other processes from entering a critical section. **Not good!**

- Does bounded waiting hold? **Exercise!**
  
  **Bound = ?**
Shared variable $\text{flag}[i]$ is the “state” of process $P_i$: interested or not-interested.

$P_i$ indicates its intention to enter, waits for $P_j$ to exit, enters its section, and, finally, changes to “I am out” upon exit.
**Attempt II: 2/4**

- **Mutual Exclusion**
  - \( P_i \) is in CS if \( \text{flag}[i] \) is TRUE AND \( \text{flag}[j] \) is FALSE.
  - \( P_j \) is in CS if \( \text{flag}[j] \) is TRUE AND \( \text{flag}[i] \) is FALSE.
  - If both are in their CSs, \( \text{flag}[i] \) and \( \text{flag}[j] \) must be both TRUE and FALSE at the same time.
  - This is absurd.

```cpp

I am interested

do {
  flag[i] = TRUE;
  while (flag[j]);
  enter
  critical section
  exit
  flag[i] = FALSE;
  I am not interested
} while (flag[j]);
```

do {
  flag[i] = TRUE;
  while (flag[j]);
  flag[i] = FALSE;
} while (true);

- **Progress**
- If both $P_i$ and $P_j$ set $\text{flag}[i]$ and $\text{flag}[j]$ to TRUE at the same time, then both will loop at the while forever and no one can enter.

- **A decision cannot be made in finite time (i.e., not deadlock-free).**
Attempt II: 4/4

- **Bounded Waiting**

  Suppose \( P_j \) is in its critical section and \( P_i \) is waiting to enter.

  If \( P_i \) fails to detect the change of \( \text{flag}[j] \) when \( P_j \) exits, \( P_j \) can come back fast before \( P_i \) can check \( \text{flag}[j] \) again, and set \( \text{flag}[j] \) to TRUE. Then, no one can enter.

- We need to do more in the while.
Consider the algorithm below:

Process 0: $P_0$

```plaintext
flag[0] = TRUE;
while (flag[1]) {
    flag[0] = FALSE;
    while (flag[1]) {
        flag[0] = TRUE;
    }
    flag[0] = FALSE;
    in critical section
}
```

Process 1: $P_1$

```plaintext
flag[1] = TRUE;
while (flag[0]) {
    flag[1] = FALSE;
    yield!
    re-test
    while (flag[0]) {
        flag[1] = TRUE;
    }
    flag[1] = FALSE;
    if you are not interested again
}
```

Then, set myself to interested again and loop back

Wait while you are interested
Set myself to not-interested
While you are interested, do the following:

I am interested
Attempt III: 2/6

- **Mutual Exclusion**
  - If $P_i$ is in its critical section, then $\text{flag}[i]$ is TRUE and $\text{flag}[j]$ is FALSE.
  - If both processes are in their critical sections, $\text{flag}[i]$ and $\text{flag}[j]$ are both TRUE and FALSE.

- **Contradiction.**

```c
flag[i] = TRUE;
while (flag[j]) {
    flag[i] = FALSE;
    while (flag[j]) ;
    flag[i] = TRUE;
}
// critical section
flag[i] = FALSE;
```

Before the while condition is met, $\text{flag}[i]$ is always set to TRUE.
Attempt III: 3/6

- **Progress**
- **Outsider Issue:** Suppose $P_j$ is not entering (i.e., elsewhere) and $P_i$ is waiting to enter.
- **Because** $\text{flag}[j]$ is FALSE, $P_i$ enters.

```c
flag[i] = TRUE;
while (flag[j]) {
    flag[i] = FALSE;
    while (flag[j]) {
    
    }
    flag[i] = TRUE;
}

// critical section

flag[i] = FALSE;
```
**Attempt III: 4/6**

- **Progress**
- **Finite Decision Time:** Suppose $P_i$ and $P_j$ are waiting to enter, and the critical section is empty.
- If $P_i$ and $P_j$ execute their corresponding statements in a fully synchronized way, both processes loop forever.
- **Progress fails.**

```c
flag[i] = TRUE;
while (flag[j]) {
    flag[i] = FALSE;
    while (flag[j])
    ;
    flag[i] = TRUE;
}
// critical section
flag[i] = FALSE;
```
Attempt III: 5/6

- **Bounded Waiting**
  - If after $P_i$ sets $\text{flag}[i]$ to FALSE, then $P_j$ has a chance to break its outer while and enter.
  - After $P_j$ sets $\text{flag}[j]$ to FALSE upon exit, $P_i$ may break its inner while. However, it is possible before it sets $\text{flag}[i]$ to TRUE, $P_j$ loops back, breaks its outer while, and enters.

```c
flag[i] = TRUE;
while (flag[j]) {
    flag[i] = FALSE;
    while (flag[j]) {
    }
    flag[i] = TRUE;
}
```

// critical section

```c
flag[i] = FALSE;
```
Attempt III: 6/6

1. flag[i] = TRUE;
2. while (flag[j]) {
3.   flag[i] = FALSE;
4.   while (flag[j])
5.     ;
6.   flag[i] = TRUE;
7. }
8. flag[i] = FALSE;

Both Processes Start

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>flag[0]</th>
<th>flag[1]</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>f[0] = T</td>
<td>f[1] = T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>while(f[1])</td>
<td></td>
<td>T</td>
<td>T</td>
<td>P_0’s line 2 while</td>
</tr>
<tr>
<td>f[0] = F</td>
<td>while(f[0])</td>
<td>F</td>
<td>T</td>
<td>P_1’s line 2 while</td>
</tr>
<tr>
<td>while(f[1])</td>
<td></td>
<td>F</td>
<td>T</td>
<td>P_0 loops line 6</td>
</tr>
<tr>
<td>P_1 enters CS</td>
<td></td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>P_1 exits CS</td>
<td></td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>f[1] = F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>P_1 resets f[1]</td>
</tr>
<tr>
<td>f[0] = T</td>
<td></td>
<td>T</td>
<td>F</td>
<td>P_0’s line 6</td>
</tr>
<tr>
<td>f[1] = T</td>
<td></td>
<td>T</td>
<td>T</td>
<td>P_1 comes back</td>
</tr>
<tr>
<td>f[0] = F</td>
<td></td>
<td>F</td>
<td>T</td>
<td>P_0’s next iteration</td>
</tr>
<tr>
<td>while(f[0])</td>
<td></td>
<td>F</td>
<td>T</td>
<td>P_1’s line 2 while</td>
</tr>
<tr>
<td>while(f[1])</td>
<td></td>
<td>F</td>
<td>T</td>
<td>P_0 loops line 6</td>
</tr>
<tr>
<td>P_1 enters CS</td>
<td></td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>P_1 exits CS</td>
<td></td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>f[1] = F</td>
<td></td>
<td>F</td>
<td>F</td>
<td>P_1 resets f[1]</td>
</tr>
<tr>
<td>f[0] = T</td>
<td></td>
<td>T</td>
<td>F</td>
<td>P_0’s line 6</td>
</tr>
<tr>
<td>f[1] = T</td>
<td></td>
<td>T</td>
<td>T</td>
<td>P_1 comes back</td>
</tr>
</tbody>
</table>
Variable turn being i or j can be considered as a "scheduler":

```c
int turn;  // initialized to i or j

Process i
flag[i] = TRUE;  // I am interested
if (flag[j]) {
    // If you are not interested, I enter
    if (turn == j) {  // Or, if you are, is it your turn?
        // it is your turn, not interested
        flag[i] = FALSE;
        while (turn == j)  // wait until it is not your turn
            ;
        flag[i] = TRUE;  // I am interested AGAIN
    }  // Then, I enter!
}

Critical Section
turn = j;  // upon exit, you have the turn
flag[i] = FALSE;  // and I am not interested
```
**Attempt IV: 2/8**

- **Mutual Exclusion: 1**
  - **IMPORTANT**: We have an *if* rather than a *while*. This is important!
  - If $\text{flag}[j]$ is FALSE, $P_i$ enters immediately.
  - If $\text{flag}[j]$ is TRUE, $P_i$ goes into the *then* part of the *if*.
  - Thus, whether $\text{flag}[j]$ is TRUE or FALSE, $P_i$ can have a chance to enter.
  - *turn* serves as a scheduler: who runs next.

```c
Bool flag[2] = { FALSE, FALSE }; int turn;

Process i
flag[i] = TRUE;
if (flag[j]) {
  if (turn == j) {
    flag[i] = FALSE;
    while (turn == j);
    flag[i] = TRUE;
  }
}

Critical Section
turn = j;
flag[i] = FALSE;
```
Mutual Exclusion: 2
- If flag[j] is FALSE, P_i enters immediately.
- If flag[j] is TRUE, execution enters then.
- P_i enters if turn is i.
- If turn is j, then P_i waits until turn becomes i.
- Therefore, P_i is in its critical section, we have:
  - flag[j] is FALSE
  - Or turn is i.
  - flag[i] is TRUE
Attempt IV: 4/8

Mutual Exclusion: 3

If $P_i$ and $P_j$ are both in their critical sections:

- For $P_i$: $flag[j]$ is FALSE OR turn is i.
- For $P_j$: $flag[i]$ is FALSE OR turn is j.
- But $flag[i]$ and $flag[j]$ are both TRUE before entering.
- Thus, turn being i and turn being j must both hold. A contradiction.

```c
Bool flag[2] = { FALSE, FALSE }; int turn;

Process i
flag[i] = TRUE;
if (flag[j]) {
    if (turn == j) {
        flag[i] = FALSE;
        while (turn == j);
    }
    flag[i] = TRUE;
}

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 5/8

- **Progress:** 1/2
- **Outsider Issue:**
  
  - If $P_j$ is not interested and $P_i$ tries to enter, because $flag[j]$ was set to FALSE when $P_j$ exited, $P_i$ enters. **No outsider issues!**

```c
int turn;

Process i
flag[i] = TRUE;
if (flag[j]) {
    if (turn == j) {
        flag[i] = FALSE;
        while (turn == j)
            ;
        flag[i] = TRUE;
    }
}

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 6/8

- Progress: 2/2
- Finite Decision Time:
  - If $P_i$ and $P_j$ are both waiting to enter, and the CS is empty, then $\text{flag}[i]$ and $\text{flag}[j]$ are both TRUE and the determining factor is the value of turn.
  - Only the while can cause infinite decision time.
  - Because the value of turn is not modified before exit, the test in the while takes finite time, and decision time is finite (i.e., deadlock free).

```c
Bool flag[2] = { FALSE, FALSE }; int turn;

Process i
flag[i] = TRUE;
if (flag[j]) {
    if (turn == j) {
        flag[i] = FALSE;
        while (turn == j);
        flag[i] = TRUE;
    }
}

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 7/8

- Bounded Waiting: 1
- Upon exit, $P_j$ sets $\text{turn}$ to $i$ and $\text{flag}[j]$ to FALSE.
- When $P_i$ sees $\text{turn}$ being $i$, before $P_i$ can reset $\text{flag}[i]$ back to TRUE, $P_i$ may be switched out and a fast $P_j$ may come back and enter again.
- This can happen over and over.
- Thus, there is no way to determine a possible bound.

```c
bool flag[2] = { FALSE, FALSE }; int turn;

process i
flag[i] = TRUE;
if (flag[j]) {
    if (turn == j) {
        flag[i] = FALSE;
        while (turn == j);
        flag[i] = TRUE;
    }
}
critical section
turn = j;
flag[i] = FALSE;
context switch can happen here
```
Attempt IV: 8/8

Bounded Waiting: 2

Bounded waiting fails

```c
Bool flag[2] = { FALSE, FALSE }; int turn;

Process i
flag[i] = TRUE;
if (flag[j]) {
    if (turn == j) {
        flag[i] = FALSE;
        while (turn == j)
            flag[i] = TRUE;
    }
}

Critical Section
turn = j;
flag[i] = FALSE;
```

Table:

<table>
<thead>
<tr>
<th></th>
<th>P_i</th>
<th>P_j</th>
<th>turn</th>
<th>f_i</th>
<th>f_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>CS</td>
<td>j</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>f_i=F</td>
<td>j</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>t=i</td>
<td>i</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>t=i?</td>
<td>f_j=F</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f_j=T</td>
</tr>
</tbody>
</table>

Diagram:
- Context switch can happen here.
Attempt V: A Combination

Peterson’s Algorithm

```c
bool flag[2] = FALSE; // process Pi
int turn;

do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

    flag[i] = FALSE;
} while (true);
```

I am interested

yield to you first

I am done

enter

wait while you are interested and it is your turn.

exit

If $P_i$ is in its critical section, then it sets

- $\text{flag}[i]$ to TRUE
- $\text{turn}$ to $j$ (but $\text{turn}$ may not be $j$ after this point because $P_j$ may set it to $i$ later).
- and waits until $\text{flag}[j] && \text{turn} == j$ becomes FALSE
If $P_j$ is in its critical section, then it sets

- $\text{flag}[j]$ to TRUE
- $\text{turn}$ to $i$ (but $\text{turn}$ may not be $i$ after this point because $P_i$ may set it to $j$ later).
- and waits until $\text{flag}[i] \land \text{turn} == i$ becomes FALSE
If processes $P_i$ and $P_j$ are both in their critical sections, then we have:

- $\text{flag}[i]$ and $\text{flag}[j]$ are both TRUE.
- $\text{flag}[i] \land \text{turn} == i$ and $\text{flag}[j] \land \text{turn} == j$ are both FALSE.

Therefore, $\text{turn} == i$ and $\text{turn} == j$ must both be FALSE.
Because $\text{turn} == i$ and $\text{turn} == j$ are both FALSE, $\text{turn} == j$ and $\text{turn} == i$ are both TRUE.

This is impossible, because a variable (i.e., $\text{turn}$) cannot hold two different values at the same time (i.e., $i$ and $j$).

Therefore, we have a contradiction and mutual exclusion holds.
Attempt V: Mutual Exclusion 6/12

- We normally use the proof-by-contradiction technique to establish the mutual exclusion condition.
- To do so, follow the procedure below:
  - Find the condition $C_0$ for $P_0$ to enter its CS
  - Find the condition $C_1$ for $P_1$ to enter its CS
  - If $P_0$ and $P_1$ are in their critical sections, $C_0$ and $C_1$ must both be true.
  - From $C_0$ and $C_1$ being both true, we should be able to derive an absurd result.
  - Therefore, mutual exclusion holds.
Attempt V: Mutual Exclusion 7/12

- We care about the conditions \( C_0 \) and \( C_1 \). The way of reaching these conditions via instruction execution is usually un-important.

- Never use an execution sequence to prove mutual exclusion. In doing so, you make a serious mistake, which is referred to as proof-by-example.

- You may use a single example to show a proposition being false. However, you cannot use a single example to show a proposition being true. That is, \( 3^2 + 4^2 = 5^2 \) cannot be used to prove \( a^2 + b^2 = c^2 \) for any right triangles.
If $P_i$ and $P_j$ are both waiting to enter their critical sections, since the value of `turn` can only be $i$ or $j$ but not both, one process can pass its **while** loop with one comparison (i.e., decision time is finite).

If $P_i$ is waiting and $P_j$ is not interested in entering its CS:

- Since $P_j$ is **not interested** in entering, `flag[j]` was set to **FALSE** when $P_j$ exits, and $P_i$ enters.
- Thus, the process that is not entering does not influence the decision.
If $P_i$ wishes to enter, we have three cases:

1. $P_j$ is outside of its critical section.
2. $P_j$ is in the entry section.
3. $P_j$ is in its critical section.
**CASE I:** If $P_j$ is outside of its critical section, $P_j$ sets $\text{flag}[j]$ to FALSE when it exits its critical section, and $P_i$ may enter.

- In this case, $P_i$ does not wait. Or, $P_i$ waits for 0 turn.

```
process $P_i$
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
flag[j] = TRUE;
turn = i;
while (flag[i] && turn == i);
```

```
process $P_j$
flag[j] = TRUE;
turn = i;
while (flag[i] && turn == i);
```
CASE 2: If \( P_j \) is in the entry section, depending on the value of \( \text{turn} \), we have two cases:

- If \( \text{turn} \) is \( i \) (e.g., \( P_i \) sets \( \text{turn} \) to \( j \) before \( P_j \) sets \( \text{turn} \) to \( i \)), \( P_i \) enters immediately. \( P_i \) waits for 0 \( \text{turn} \).

- Otherwise, \( P_j \) enters, and \( P_i \) stays in the while loop, and we have **CASE 3**. In this case, \( P_i \) waits for at least one \( \text{turn} \).
**CASE 3**: If $P_j$ is in its critical section, $\text{turn}$ must be $j$ and $P_i$ waits for at most one round.

<table>
<thead>
<tr>
<th>$P_i$</th>
<th>$P_j$</th>
<th>flag[i]</th>
<th>flag[j]</th>
<th>turn</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{flag}[i]=T$</td>
<td>$\text{flag}[j]=T$</td>
<td>TRUE</td>
<td>TRUE</td>
<td>?</td>
<td>$P_j$ enters</td>
</tr>
<tr>
<td>while (...)</td>
<td>$\text{flag}[j]=T$</td>
<td>TRUE</td>
<td>TRUE</td>
<td>$j$</td>
<td>$P_j$ in CS</td>
</tr>
<tr>
<td></td>
<td>$\text{flag}[j]=F$</td>
<td>TRUE</td>
<td>FALSE</td>
<td>$j$</td>
<td>$P_j$ exits</td>
</tr>
<tr>
<td></td>
<td>$\text{flag}[j]=T$</td>
<td>TRUE</td>
<td>TRUE</td>
<td>$j$</td>
<td>$P_j$ returns</td>
</tr>
<tr>
<td></td>
<td>$\text{turn} = i$</td>
<td>TRUE</td>
<td>TRUE</td>
<td>$i$</td>
<td>$P_j$ yields</td>
</tr>
<tr>
<td></td>
<td>while (...)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>$i$</td>
<td>$P_j$ loops</td>
</tr>
</tbody>
</table>

$P_i$ has a chance to enter here. If $P_j$ comes back fast, $P_i$ enters.
One More Example: 1/4

Consider the following simple algorithm:

```c
bool flag[2] = { false, false }; // global flags
bool turn[2] = { false, true }; // global turn variable

process p_0
flag[0] = true;
turn[0] = turn[1];
repeat
    until (!flag[1] || turn[0] != turn[1]);
critical section
flag[0] = false;

process p_1
flag[1] = true;
turn[1] = !turn[0];
repeat
    until (!flag[0] || turn[0] == turn[1]);
critical section
flag[1] = false;
```

- `P_0` waits for the two `turn` values being not equal.
- `P_1` waits for the two `turn` values being equal.
- `P_0` is interested in the current turn, while `P_1` is not interested in the current turn.
One More Example: 2/4

- **Mutual Exclusion:**

```plaintext
```

**Process P₀**

```plaintext
flag[0] = TRUE;
turn[0] = turn[1];
repeat
    until (!flag[1] || turn[0] != turn[1]);
```

**Critical Section**

```plaintext
flag[0] = FALSE;
```

**Process P₁**

```plaintext
flag[1] = TRUE;
turn[1] = !turn[0];
repeat
    until (!flag[0] || turn[0] == turn[1]);
```

**Critical Section**

```plaintext
flag[1] = FALSE;
```

- If P₀ is in CS, flag[0] is TRUE, flag[1] is FALSE OR turn[0] != turn[1]
- If P₁ is in CS, flag[1] is TRUE, flag[0] is FALSE OR turn[0] == turn[1]
- If P₀ and P₁ are in both in CS, flag[0] and flag[1] are TRUE in the until
- Thus, turn[0] and turn[1] are equal and not equal to each other
- This is a contradiction!
One More Example: 3/4

- **Progress:**

```plaintext

Process P0
flag[0] = TRUE;
turn[0] = turn[1];
repeat
  until (!flag[1] ||
    turn[0] != turn[1]);

Critical Section
flag[0] = FALSE;

Process P1
flag[1] = TRUE;
turn[1] = !turn[0];
repeat
  until (!flag[0] ||
    turn[0] == turn[1]);

Critical Section
flag[1] = FALSE;

- **Outsider Issue:** If P1 is not interested, it sets flag[1] to FALSE and P0 enters freely.
- **Finite Decision Time:** If both are trying to enter, testing whether turn[0] is equal to turn[1] takes finite time to choose a candidate to enter.
```
One More Example: 4/4

- **Bounded Waiting**: Assume $P_0$ is entering.

Bool $flag[2] = \{ \text{FALSE, FALSE} \}$; // global flags
Bool $turn[2] = \{ \text{FALSE, TRUE} \}$; // global turn variable

---

**Process $P_0$**

- $flag[0] = \text{TRUE}$;
- $turn[0] = turn[1]$;
- repeat
  - until (!$flag[1] ||
  - $turn[0] != turn[1]$);

**Critical Section**

- $flag[0] = \text{FALSE}$;

---

**Process $P_1$**

- $flag[1] = \text{TRUE}$;
- $turn[1] = !turn[0]$;
- repeat
  - until (!$flag[0] ||
  - $turn[0] == turn[1]$);

---

- **If $P_1$ is not interested**, $P_0$ waits for 0 round and enters.
- **If $P_1$ is competing**, $P_1$ enters if $turn[0] = turn[1]$. If $P_0$ detects $flag[1]$ being changed to \text{FALSE} when $P_1$ exits, $P_0$ enters ($P_0$ waits for 1 round). Or, $P_1$ comes back to set $flag[1]$ to \text{TRUE} and negate (i.e., modify) $turn[1]$. Then, $P_0$ enters.
- **If $P_1$ is in CS**, this is the second half of the above. This, $P_0$ waits for at most 1 round.
Hardware Support

- There are two types of hardware synchronization supports:
  - Disabling/Enabling interrupts: This is slow and difficult to implement on multiprocessor systems.
  - Special privileged, actually atomic, machine instructions:
    - Test and set (TS)
    - Compare and Swap (CS)
    - Swap
Interrupt Disabling

- Because interrupts are disabled, no context switch can occur in a critical section (why?).
- Infeasible in a multiprocessor system because all CPUs/cores must be informed.
- Some features that depend on interrupts (e.g., clock) may not work properly.

```c
do {
    disable interrupts
    critical section
    enable interrupts
} while (1);
```
Test-and-Set: 1/2

- **TS** is atomic.
- **Mutual exclusion** is met as the **TS** instruction is atomic. See next slide.
- However, **bounded waiting** may not be satisfied. **Progress?**

```c
bool TS(bool *key) {
    bool save = *key;
    *key = TRUE;
    return save;
}
```

```c
bool lock = FALSE;

do {
    while (TS(&lock));
    lock = FALSE;
} while (1);
```

A process is in its critical section if the **TS** instruction returns **FALSE**.
Test-and-Set: 2/2

- $P_0$ is in its CS, if $TS$ returns FALSE.
- $P_1$ is in its CS, if $TS$ returns FALSE.
- If $P_0$ and $P_1$ are in their critical sections, they both got the FALSE return value from $TS$.
- $P_0$ and $P_1$ cannot execute their $TS$ instructions at the same time because $TS$ is atomic. Their $TS$ are executed sequentially.
- Hence, if $P_0$ executes the $TS$ before the other, once $P_0$ finishes its $TS$, the value of $lock$ becomes TRUE. $P_1$ cannot get a FALSE return value and cannot enter its CS.
- We have a **contradiction**!

```c
bool lock = FALSE;
do {
  while (TS(&lock));
  lock = FALSE;
} while (1);
```

critical section
Compare-and-Swap: 1/2

- **CS** is atomic.
- **Mutual exclusion** is met as the **CS** instruction is atomic. See next slide.
- However, **bounded waiting** may not be satisfied. **Progress?**

```c
bool CS(int *p, old, new) {
    if (*p != old)
        return FALSE;
    *p = new;
    return TRUE;
}
```

```c
bool lock = FALSE;

do {
    // ...

    while(!CS(&lock, FALSE, TRUE))
        ;

    lock = FALSE;
     }

while (1);
```

A process is in its critical section if the **CS** instruction returns **TRUE**.
Compare-and-Swap: \(2/2\)

- **CS** is useful for building mutual exclusion.
- Because **CS** is atomic, it offers a fast way for updating variables such as doing `count++` and `count--` in a mutually exclusive way.
- It is also very useful in a kernel for implementing locks. You will learn this in an *Operating Systems* course.

```c
bool CS(int *p, old, new)
{
    if (*p != old)
        return FALSE;
    *p = new;
    return TRUE;
}
```

```c
int count = 0;
done = FALSE;
while (!done) {
    val = *count;
    done = CS(&count, val, val+1);
}
```
Problems with Software and Hardware Solutions

- All these solutions use **busy waiting**.
- **Busy waiting** means a process waits by executing a tight loop to check the status/value of a variable.
- Busy waiting may be needed on a multiprocessor system; however, it wastes CPU cycles that some other processes may use productively.
- Even though some systems may allow users to use some atomic instructions, unless the system is lightly loaded, CPU and system performance can be low, although a programmer may “think” his/her program looks more efficient.
- So, we need better solutions.
The End