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.
An Important Assumption: 3/3

This is Dijkstra’s paper. It was a technical report before published as a paper.

Several Examples will be discussed in terms of:
(1) Mutual Exclusion, (2) Progress, and (3) Bounded Waiting.
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!

```c
process P_i
    do {
        if it is not my turn, I wait
        while (turn != i);
        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 $P_i$

do {
  while ($\text{turn} \neq i$);  
  enter
  critical section
  turn = j;  
  exit
} while (1);

If it is not my turn, I wait
I am done, it is your turn now
Attempt I: 3/3

- **Progress**
- If \( P_i \) sets \( \text{turn} \) 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!**

```plaintext
process \( P_i \)
do {
  if it is not my turn, I wait
  while (\( \text{turn} \neq i \));
  \text{critical section}
  \text{exit}
  \text{turn} = j;
} while (1);
I am done, it is your turn now
Bound = ?
```
Shared variable 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.
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.

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

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

- **Progress**
- If both $P_i$ and $P_j$ set flag[i] and 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: $P_0$
- $\text{flag}[0] = \text{TRUE};$
- while (flag[1]) {
  - flag[0] = FALSE;
  - while (flag[1]) {
    - flag[0] = TRUE;
  }
- in critical section
- flag[0] = FALSE;

Process 1: $P_1$
- $\text{flag}[1] = \text{TRUE};$
- while (flag[0]) {
  - flag[1] = FALSE;
  - while (flag[0]) {
    - flag[1] = TRUE;
  }
- out of critical section
- flag[1] = FALSE;

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

Flags are initialized to FALSE
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.
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.
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

flag[i] = FALSE;
```

Bounded waiting also fails.
Find execution sequences yourself
Attempt III: 6/6

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>flag[0]</th>
<th>flag[1]</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. flag[i] = TRUE;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. while (flag[j]) {</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. flag[i] = FALSE;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. while (flag[j])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. flag[i] = TRUE;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. }</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. flag[i] = FALSE;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

// critical section

<table>
<thead>
<tr>
<th>Both Processes Start</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0] = T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>f[1] = T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

P₀’s line 2 while

<table>
<thead>
<tr>
<th>while(f[1])</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0] = F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>while(f[0])</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

P₁’s line 2 while

<table>
<thead>
<tr>
<th>while(f[1])</th>
<th>P₁ enters CS</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0] = T</td>
<td>P₁ exits CS</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>f[1] = F</td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

P₀ loops line 6

<table>
<thead>
<tr>
<th>while(f[1])</th>
<th>P₁ enters CS</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0] = F</td>
<td>P₁ exits CS</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>f[1] = T</td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

P₁ comes back

<table>
<thead>
<tr>
<th>while(f[0])</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(f[1])</td>
<td>P₁ enters CS</td>
<td>F</td>
</tr>
<tr>
<td>f[0] = F</td>
<td>P₁ exits CS</td>
<td>F</td>
</tr>
<tr>
<td>f[1] = F</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

P₀’s next iteration

<table>
<thead>
<tr>
<th>while(f[1])</th>
<th>P₁ enters CS</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>f[0] = F</td>
<td>P₁ exits CS</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>f[1] = T</td>
<td></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

P₀ loops line 6

<table>
<thead>
<tr>
<th>P₀ enters twice and can enter again &amp; again</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

P₁ comes back
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
while (flag[j]) {
    if (turn == j) { // If you are, is it your turn?
        flag[i] = FALSE; // it is your turn, not interested
        while (turn == j) // wait until it is not your turn
            ;
        flag[i] = TRUE; // I am interested AGAIN
    }
} // Then, loop back and retry!

Critical Section
turn = j; // upon exit, you have the turn
flag[i] = FALSE; // and I am not interested
```
**Attempt IV: 2/10**

### Mutual Exclusion: 1
- If $\text{flag}[j]$ is FALSE, $P_i$ enters immediately.
- If $\text{flag}[j]$ is TRUE, $P_i$ enters the while.
- At the end of the while, $\text{flag}[i]$ is reset to TRUE.
- Thus, if $P_i$ enters, we have $\text{flag}[i]=\text{TRUE}$ and $\text{flag}[j]=\text{FALSE}$.
- $\text{turn}$ does not play a role as its value does not affect who can enter.

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

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

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 3/10

- **Mutual Exclusion: 2**
  - $P_i$ enters, $\text{flag}[i] = \text{TRUE}$ and $\text{flag}[j] = \text{FALSE}$.
  - $P_j$ enters, $\text{flag}[j] = \text{TRUE}$ and $\text{flag}[i] = \text{FALSE}$.
  - If $P_i$ and $P_j$ are in the critical section, $\text{flag}[i]$ and $\text{flag}[j]$ are both TRUE and FALSE.
  - This is impossible and mutual exclusion holds.
Attempt IV: 4/10

- **Progress:** 1
- **Outsider Issue**
- If $P_i$ is entering and $P_j$ is not, then $P_j$ has set `turn` to $i$ and `flag[j]` to false.
- In this case, $P_i$ reaches the `while` and enters the critical section.
- `turn` is NOT used.
- Therefore, an outsider will not affect those waiting to enter.

```cpp
bool flag[2] = {false, false};
int turn;

// Process i
flag[i] = true;
while (flag[j]) {
    if (turn == j) {
        flag[i] = false;
        while (turn == j);
        flag[i] = true;
    }
}

// Critical Section

turn = j;
flag[i] = false;
```
Attempt IV: 5/10

- **Progress:** 2 (turn=j)
- **Finite Decision Time**
- If $P_i$ and $P_j$ are both entering, $\text{flag}[i] = \text{flag}[j] = \text{TRUE}$ and both enter the while.
- If $\text{turn} = j$, $P_i$ loops here after setting $\text{flag}[i]$ to FALSE.
- This is equivalent to “waiting for my turn (i.e., $\text{turn} = i$).”

```plaintext
int   turn;

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

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 6/10

- **Progress:** 3 \(\text{(turn=j)}\)
- **Finite Decision Time**
  - \(P_j\) checks if it is its turn (i.e., \(\text{turn} = j\)).
  - Because \(\text{turn}\) is \(j\), \(P_j\) loops back to check if \(\text{flag}[i]\) is \(\text{FALSE}\).
  - Thus, \(P_j\) loops around while \((\text{true})\) and if \((\text{false})\) until \(\text{flag}[i]\) is \(\text{FALSE}\) because \(\text{turn}\) is \(j\).
  - Because \(P_i\) only needs 3 statements to set \(\text{flag}[i]\) to \(\text{FALSE}\), \(P_j\) takes finite time to enter.

```c
int turn;

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

Critical Section
turn = i;
flag[j] = FALSE;
```
**Attempt IV: 7/10**

- **Bounded Waiting: 1**
- Suppose $P_i$ is entering. $P_j$’s location dictates what we have:
  1. $P_j$ is not interested
  2. $P_j$ is in the CS
  3. $P_j$ is entering.
- If $P_j$ is not interested, it has already set $\text{turn}$ to $i$ and $\text{flag}[j]$ to false.
- Hence, $P_i$ enters, waiting for 0 round!

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

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

Critical Section
turn = j;
flag[i] = FALSE;
```
**Attempt IV: 8/10**

- **Bounded Waiting: 2**
  - If $P_j$ is in the CS, $\text{flag}[j]$ is true.
  - When $P_j$ exits, it sets $\text{turn}$ to $i$ and $\text{flag}[j]$ to false.
  - If $P_i$ fails to see $\text{flag}[j]$ being false, $P_j$ can come back quickly and compete against $P_i$ to enter. This is **Case 3**.
  - If $P_i$ sees $\text{flag}[j]$ being false, $P_i$ enters, waiting for **0** round.

```c
int turn;

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

Critical Section
turn = j;
flag[i] = FALSE;
```
Attempt IV: 9/10

- **Bounded Waiting:** 3
- If \( P_j \) and \( P_j \) are competing to enter, they both set their `flag[]` to true.
- The value of `turn` dictates who can enter. Because `turn` can only be \( i \) or \( j \), either \( P_i \) or \( P_j \) enters.
- If \( P_j \) enters, then we have **Case 2** and \( P_i \) enters because \( P_j \) sets `turn` to \( i \) upon exit.
- Thus, \( P_i \) waits for at most one around.

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

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

// Critical Section
turn = j;
flag[i] = FALSE;
```
This algorithm was due to Prof. Dr. Th. J. (Dirk) Dekker (March 1, 1927 - November 25, 2021) in 1965 and is usually referred to as Dekker’s algorithm.

Prof. Dr. Dekker was a Dutch mathematician.

Th. J. Dekker = Theodorus Jozef Dekker.

Dekker’s algorithm is the first known correct solution to the mutual exclusion problem.
Attempt IV: 4/n

- **Mutual Exclusion:** 3
  - Thus, $P_i$ can enter the critical section **if and only if** $\text{flag}[i] = \text{TRUE and flag}[j] = \text{FALSE}$. 
  - Thus, $P_j$ can enter the critical section **if and only if** $\text{flag}[j] = \text{TRUE and flag}[i] = \text{FALSE}$. 
  - As a result, if $P_i$ and $P_j$ are both in the critical section, $(\text{flag}[i] = \text{TRUE and flag}[j] = \text{FALSE})$ and $(\text{flag}[j] = \text{TRUE and flag}[i] = \text{FALSE})$ are both true.
  - Hence, $\text{flag}[i]$ and $\text{flag}[j]$ are both true and false, which is impossible.
  - $P_i$ and $P_j$ cannot both be in the critical section.
Attempt IV: 5/8

- **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 $flag[i]$ and $flag[j]$ 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
int turn;

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

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

- **Bounded Waiting: 1**

- Upon exit, $P_j$ sets turn to $i$ and flag[$j$] to FALSE.

- When $P_i$ sees turn being $i$, before $P_i$ can reset 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
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;
```

![Diagram showing the algorithm with a table and process flow](image-url)
### Peterson’s Algorithm

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

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

**Critical Section**

I am interested

I am done

yield to you first

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

process $P_i$

```plaintext
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process $P_j$

```plaintext
flag[j] = TRUE;
turn = i;
while (flag[i] && turn == i);
```
Attempt V: Mutual Exclusion

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

<table>
<thead>
<tr>
<th>Process $P_i$</th>
<th>Process $P_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag[i] = TRUE;</td>
<td>flag[j] = TRUE;</td>
</tr>
<tr>
<td>turn = j;</td>
<td>turn = i;</td>
</tr>
<tr>
<td>while (flag[j] &amp;&amp; turn == j);</td>
<td>while (flag[i] &amp;&amp; turn == i);</td>
</tr>
</tbody>
</table>

- 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 \text{TRUE}.
  - $\text{flag}[i] && \text{turn} == i$ and $\text{flag}[j] && \text{turn} == j$ are both \text{FALSE}.
  - Therefore, $\text{turn} == i$ and $\text{turn} == j$ must both be \text{FALSE}. 

```plaintext
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
flag[j] = TRUE;
turn = i;
while (flag[i] && turn == i);
```
Because \( \text{turn} = \text{i} \) and \( \text{turn} = \text{j} \) are both FALSE, \( \text{turn} = \text{j} \) and \( \text{turn} = \text{i} \) are both TRUE.

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

Therefore, we have a contradiction and mutual exclusion holds.
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.
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.

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

process \( P_j \)
flag[j] = TRUE;
turn = i;
while (flag[i] && turn == i);
```
CASE I: If $P_j$ is outside of its critical section, $P_j$ sets 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.
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 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 turn.
**CASE 3**: If \( P_j \) is in its critical section, 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>flag([i])=T</td>
<td>flag([j])=T</td>
<td>TRUE</td>
<td>TRUE</td>
<td>?</td>
<td>( P_j ) enters</td>
</tr>
<tr>
<td>while (...)</td>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td>( j )</td>
<td>( P_j ) enters</td>
</tr>
<tr>
<td>( P_j ) is in CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flag([j])=F</td>
<td>TRUE</td>
<td>FALSE</td>
<td>( j )</td>
<td>( P_j ) exits</td>
<td></td>
</tr>
<tr>
<td>flag([j])=T</td>
<td>TRUE</td>
<td>TRUE</td>
<td>( j )</td>
<td>( P_j ) returns</td>
<td></td>
</tr>
<tr>
<td>turn = ( i )</td>
<td>TRUE</td>
<td>TRUE</td>
<td>( i )</td>
<td>( P_j ) yields</td>
<td></td>
</tr>
<tr>
<td>while (...)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>( i )</td>
<td>( P_j ) loops</td>
<td></td>
</tr>
<tr>
<td>Critical Sec</td>
<td>( P_i ) enters</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( P_i \) has a chance to enter here. if \( P_j \) comes back fast
One More Example: 1/4

- Consider the following simple algorithm:

```cpp
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
```
One More Example: 2/4

- **Mutual Exclusion:**

```c

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

Critical Section
flag[0] = FALSE;              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!

интересно
One More Example: 3/4

- Progress:

```

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;
```

- Outsider Issue: If P_1 is not interested, it sets flag[1] to FALSE and P_0 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 P₀ is entering.

```c

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

Critical Section
flag[0] = FALSE;              flag[1] = FALSE;

➢ If P₁ is not interested, P₀ waits for 0 round and enters.
➢ If P₁ is competing, P₁ enters if turn[0] = turn[1]. If P₀ detects flag[1] being changed to FALSE when P₁ exits, P₀ enters (P₀ waits for 1 round). Or, P₁ comes back to set flag[1] to TRUE and negate (i.e., modify) turn[1]. Then, P₀ enters.
➢ If P₁ is in CS, this is the second half of the above. This, P₀ 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.
**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**

- **P₀** is in its CS, if **TS** returns **FALSE**.
- **P₁** is in its CS, if **TS** returns **FALSE**.
- If **P₀** and **P₁** are in their critical sections, they both got the **FALSE** return value from **TS**.
- **P₀** and **P₁** cannot execute their **TS** instructions at the same time because **TS** is atomic. Their **TS** are executed sequentially.
- Hence, if **P₀** executes the **TS** before the other, once **P₀** finishes its **TS**, the value of **lock** becomes **TRUE**. **P₁** cannot get a **FALSE** return value and cannot enter its CS.
- We have a **contradiction**!

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

```
lock = FALSE;
}
while (1);
```
**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 {
    ... 
    entry
    while(!CS(&lock,FALSE,TRUE))
    ;
critical section
    lock = FALSE;
    exit
    } while (1);
```

A process is in its critical section if the CS instruction returns TRUE.
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.
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