Part III
Synchronization
Software and Hardware Solutions

Computers are useless. They can only give answers.

Pablo Picasso
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 have to design an enter-exit protocol for a critical section to ensure mutual exclusion.

We will go through a number of unsuccessful attempts and finally obtain a correct one.

These solutions are pure software-based.
An Important Assumption: 1/3

- We have the following assumption*:
  - Inspecting the present value of a common variable and assigning a new value to such a common 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 common variable simultaneously, the assignments are done sequentially.
  - When a process checks the value of a common 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.
  - But, expression evaluation is **NOT** atomic.
An Important Assumption: 3/3

This is Dijkstra’s paper. It was a technical report a year earlier.

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,
**Attempt I: 1/3**

- Shared variable `turn` controls who can enter the critical section.
- Since `turn` is either 0 or 1, only one can enter.
- However, processes are forced to run in an *alternating* way.
- *Not good!*

```c
process P_i

do { 
    while (turn != i); 
    critical section 
    turn = j; 
} while (1); 

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

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

- **Mutual Exclusion**
- \( P_0 \) in its CS if \( \text{turn}=0 \).
- \( P_1 \) in its CS if \( \text{turn}=1 \).
- If \( P_0 \) and \( P_1 \) are \textit{BOTH} in their CSs, then \( \text{turn}=0 \) and \( \text{turn}=1 \) must \textit{BOTH} be true.
- This is absurd, because a variable can only hold one and only one value (i.e., cannot hold both 0 and 1) at any time.
### 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.

- Thus, an irrelevant process can block other processes from entering a critical section. **Not good!**

- Does bounded waiting hold? **Exercise!**

```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 II: 1/5

- **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.

```cpp

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

**Critical section**

- *I am interested*
- *wait for you*
- *enter*
- *critical section*
- *exit*
- *I am not interested*
**Attempt II: 2/5**

- **Mutual Exclusion**
- \( P_0 \) is in CS if \( \text{flag}[0] \) is \text{TRUE} \text{ AND } \text{flag}[1] \) is \text{FALSE}.
- \( P_1 \) is in CS if \( \text{flag}[1] \) is \text{TRUE} \text{ AND } \text{flag}[0] \) is \text{FALSE}.
- If both are in their CSs, \( \text{flag}[0] \) and \( \text{flag}[1] \) must be both \text{TRUE} \text{ AND } \text{FALSE} \) at the same time.
- This is absurd.

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

*I am interested*  
*I am not interested*  
*wait for you*  
*enter*  
*critical section*  
*exit*
Attempt II: 3/5

- **Progress**
- If both $P_0$ and $P_1$ set flag[0] and flag[1] 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.**
### Attempt II: 4/5

- **Progress**
- The two statements of the enter section have to be executed *atomically* (i.e., no interleaving and as a single unit).
- A `flag[]` assignment cannot run between the `flag[]` and `while` of the other process.
- Otherwise, Progress will not be met.

```c

I am interested

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

    critical section

    flag[i] = FALSE;
}

I am not interested

wait for you
```
Attempt II: 5/5

- **Bounded Waiting**
- Suppose that the two statements in the enter section form an atomic “instruction”.
- If \( P_0 \) enters first and finally exits, if \( P_1 \) fails to detect the value change of flag[0], \( P_0 \) can enter again!
- This may repeat multiple times and hence .....

```c

do {
  I am interested
  flag[i] = TRUE;
  while (flag[j]);
  critical section
  flag[i] = FALSE;
}

I am not interested
```
Peterson’s Algorithm

1/12

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

doi{
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    flag[i] = FALSE;
}
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.
Attempt III: Mutual Exclusion 3/12

- 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
Attempt III: Mutual Exclusion

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.
- $\text{flag}[i] \&\& \text{turn} == i$ and $\text{flag}[j] \&\& \text{turn} == j$ are both FALSE.
- Therefore, $\text{turn} == i$ and $\text{turn} == j$ must both be FALSE.
### Attempt III: 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>

- Since $\text{turn} == i$ and $\text{turn} == j$ are both FALSE and since $\text{turn}$ is set to $j$ (by $P_i$) or $i$ (by $P_j$) before entering the critical section, only one of $\text{turn} == i$ and $\text{turn} == j$ can be FALSE but not both.

- 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 $C_1$ both being 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 *prove-by-example*.

You may use a single example to show a proposition being false. But, 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 $\text{turn}$ 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, $\text{flag}[j]$ was set to FALSE when $P_j$ exits and $P_i$ enters.
- Thus, the process that is not entering does not influence the decision.
Attempt III: Bounded Waiting

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.
CASE 2: If $P_j$ is in the entry section, depending on the value of turn, we have two cases:

- If turn is $i$ (e.g., $P_i$ sets turn to $j$ before $P_j$ sets turn to $i$), $P_i$ enters immediately.
- Otherwise, $P_j$ enters and $P_i$ stays in the while loop, and we have CASE 3.
**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></td>
</tr>
<tr>
<td>while (...)</td>
<td>TRUE</td>
<td>TRUE</td>
<td>j</td>
<td>$P_j$ enters</td>
<td></td>
</tr>
<tr>
<td>Critical Sec</td>
<td></td>
<td>$P_j$ in CS</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></td>
<td>$P_i$ enters</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$P_i$ has a chance to enter here.

if $P_j$ comes back fast
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)
    - Swap
    - Compare and Swap (CS)
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.

```
    do {
        disable interrupts
        critical section
        enable interrupts
    } while (1);
```
**Test-and-Set Instruction: 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);
```

**critical section**
Test-and-Set Instruction: 2/2

- A process is in its critical section if the TS instruction returns FALSE.
- If two processes $P_0$ 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.
- Hence, one of them, say $P_0$, executes the TS instruction 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);
```

```c
while (lock == FALSE);
```

```c
\textbf{critical section}
```

```c
lock = TRUE;
```

```c
\textbf{critical section}
```

```c
} while (1);
```
Problems with Software and Hardware Solutions

- All of 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