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).**

*E. W. Dijkstra, Co-operating sequential processes, in F. Guneys (editor), *Programming Languages*, pp. 43-112, New York, Academic Press, 1968.

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

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,

43

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

process P_i

if it is not my turn, I wait

enter

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

exit

I am done, it is your turn now

- **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!**

Attempt I: 2/3

process P_i

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

enter

exit

if it is not my turn, I wait

I am done, it is your turn now

- **Mutual Exclusion**
- **P_i in its CS if `turn=i`.**
- **P_j in its CS if `turn=j`.**
- **If P_i and P_j are BOTH in their CSs, then `turn=i` and `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.**

Attempt I: 3/3

process P_i

if it is not my turn, I wait

```
do {
    while (turn != i);   enter
    critical section
    turn = j;            exit
} 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 = ?

Attempt II: 1/4

```
bool  flag[2] = FALSE;

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

I am interested

wait for you

enter

exit

I am not interested

- 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.

Attempt II: 2/4

```
bool  flag[2] = FALSE;
```

I am interested

```
do {
```

wait for you

enter

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

critical section

exit

```
    flag[i] = FALSE;
```

```
}
```

I am not interested

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

Attempt II: 3/4

`bool  flag[2] = FALSE;`

I am interested

`do {`

wait for you

enter

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

critical section

exit

```
flag[i] = FALSE;
```

`}`

I am not interested

- **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

```
bool  flag[2] = FALSE;

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

      critical section

      flag[i] = FALSE;

}
```

I am interested

wait for you

enter

exit

I am not interested

- **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 `flag[j]` when P_j exits, P_j can come back fast before P_i can check `flag[j]` again, and set `flag[j]` to `TRUE`. Then, no one can enter.**
- **We need to do more in the `while`.**

Attempt III: 1/6

- **Consider the algorithm below:**

Process 0: P_0

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

Process 1: P_1

```
flag[1] = TRUE;
while (flag[0]) {
  flag[1] = FALSE;
  while (flag[0])
     ;
  flag[1] = TRUE;
}
flag[1] = FALSE;
```

yield!

re-test

interested again if you are not

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

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

// critical section

flag[i] = FALSE;
```

- **Mutual Exclusion**
- **If P_i is in its critical section, then `flag[i]` is `TRUE` and `flag[j]` is `FALSE`.**
- **If both processes are in their critical sections, `flag[i]` and `flag[j]` are both `TRUE` and `FALSE`.**
- **Contradiction.**

Before the `while` condition is met, `flag[i]` is always set to `TRUE`

Attempt III: 3/6

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

// critical section

flag[i] = FALSE;
```

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

Attempt III: 4/6

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

// critical section

flag[i] = FALSE;
```

- **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

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

// critical section

flag[i] = FALSE;
```

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

Bounded waiting also fails.
Find execution sequences yourself

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.  }
     // critical section
8.  flag[i] = FALSE;
```

P_0	P_1	flag[0]	flag[1]	Comment
Both Processes Start		F	F	
f[0] = T	f[1] = T	T	T	
while(f[1])		T	T	P_0's line 2 while
f[0] = F	while(f[0])	F	T	P_1's line 2 while
while(f[1])	P_1 enters CS	F	T	P_0 loops line 6
	P_1 exits CS	F	T	
	f[1] = F	F	F	P_1 resets f[1]
f[0] = T		T	F	P_0's line 6
	f[1] = T	T	T	P_1 comes back
f[0] = F		F	T	P_0's next iteration
	while(f[0])	F	T	P_1's line 2 while
while(f[1])	P_1 enters CS	F	T	P_0 loops line 6
	P_1 exits CS	F	T	
	f[1] = F	F	F	P_1 resets f[1]
f[0] = T		T	F	P_0's line 6
	f[1] = T	T	T	P_1 comes back

P_1 enters twice and can enter again & again

Attempt IV: 1/10

- Variable `turn` being `i` or `j` can be considered as a "**scheduler**":

```
Bool   flag[2] = { FALSE, FALSE };
int    turn;                  // initialized to i or j
```

Process *i*

```
flag[i] = TRUE;               // I am interested
while (flag[j]) {             // If you are not interested, I enter
   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

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

<u>Process *i*</u>

turn is **NOT** used

```
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;
```

- **<u>Mutual Exclusion: 1</u>**
- If `flag[j]` is `FALSE`, P_i enters immediately.
- If `flag[j]` is `TRUE`, P_i enters the `while`.
- At the end of the `while`, `flag[i]` is reset to `TRUE`.
- Thus, if P_i enters, we have <u>`flag[i]=TRUE` **and** `flag[j]=FALSE`</u>.
- `turn` does not play a role as its value does not affect who can enter.

Attempt IV: 3/10

```
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;
```

- **Mutual Exclusion: 2**
- **P_i enters, `flag[i] = TRUE` and `flag[j] = FALSE`.**
- **P_j enters, `flag[j] = TRUE` and `flag[i] = FALSE`.**
- **If P_i and P_j are in the critical section, `flag[i]` and `flag[j]` are both `TRUE` and `FALSE`.**
- **This is *impossible* and mutual exclusion holds.**

Attempt IV: 4/10

```
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;
```

- **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.**

Attempt IV: 5/10

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

Pi's view

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;
```

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

Attempt IV: 6/10

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

Pj's view

Process *i*

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

skipped

Critical Section

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

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

Attempt IV: 7/10

```
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;
```

- __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 `turn` to `i` and `flag[j]` to false.
- Hence, P_i enters, waiting for **0** round!

Attempt IV: 8/10

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

<u>***Process i***</u>

```
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;
```

- **Bounded Waiting: 2**
- **If P_j is in the CS, `flag[j]` is true.**
- **When P_j exits, it sets `turn` to `i` and `flag[j]` to false.**
- **If P_i fails to see `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 `flag[j]` being false, P_i enters, waiting for 0 round.**

Attempt IV: 9/10

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

Process *i*

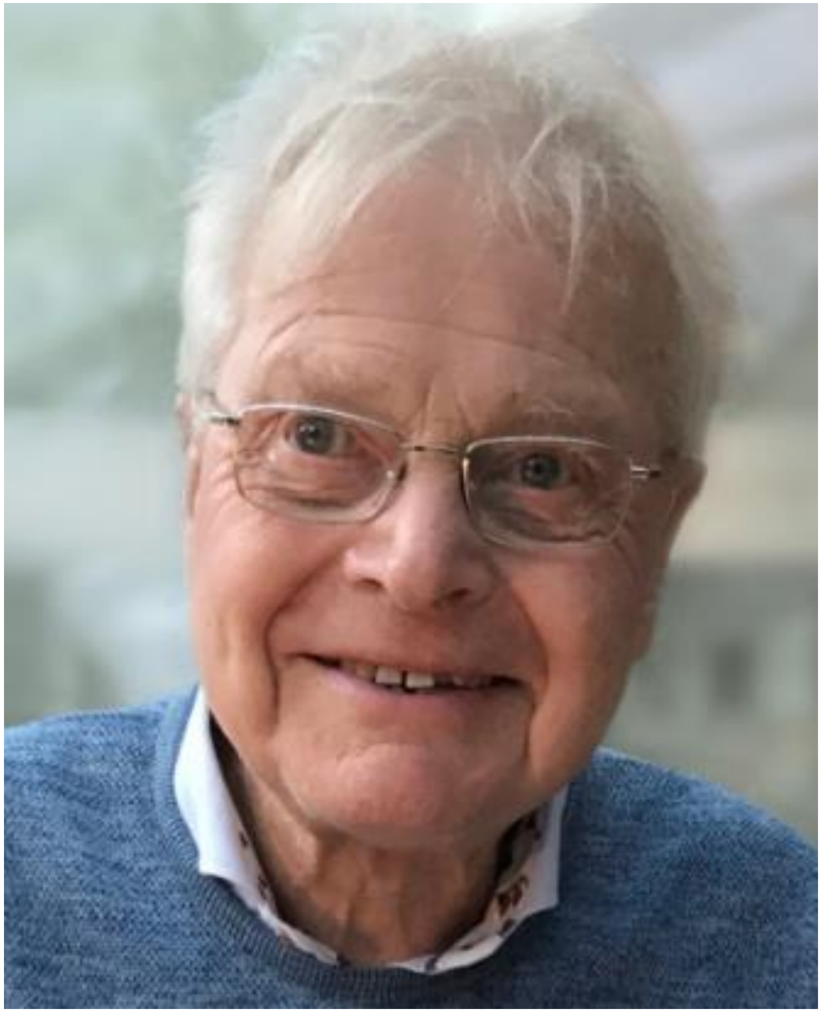
```
flag[i] = TRUE;
while (flag[j]) {
   if (turn == j) {
      flag[i] = FALSE;
      while (turn == j)
         ;
      flag[i] = TRUE;
   }
}
```

Critical Section

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

- **Bounded Waiting: 3**
- If P_i 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**.

Attempt IV: 10/10



- 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 `flag[i] = TRUE` **and** `flag[j] = FALSE`.
- Thus, P_j can enter the critical section if and only if `flag[j] = TRUE` **and** `flag[i] = FALSE`.
- As a result, if P_i and P_j are both in the critical section, (`flag[i] = TRUE` **and** `flag[j] = FALSE`) and (`flag[j] = TRUE` **and** `flag[i] = FALSE`) are both true.
- Hence, `flag[i]` and `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

```
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

```
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: 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.**

Attempt IV: 5/8

```
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;
```

- **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!**

Attempt IV: 6/8

```
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;
```

- **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)**.

Attempt IV: 7/8

```
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

- **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.

Attempt IV: 8/8

```
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

- **Bounded Waiting: 2**
- Bounded waiting fails

P_i	P_j	turn	f_i	f_j
	CS	j	T	T
f_i=F		j	F	T
	t=i	i	F	T
t=i?	f_j=F	i	F	F
	come back	i	F	F
	f_j=T	i	F	T
	enter			

Attempt V: A Combination 1/12
Peterson's Algorithm

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

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

    critical section

    flag[i] = FALSE;
    ⋮
}
```

I am interested

yield to you first

enter

I am done

exit

wait while you are interested **and** *it is your turn.*

G. L. Peterson, Myths about the Mutual Exclusion Problem, *Information Processing Letters*, Vol. 12 (1981), No. 3 (June), pp. 115-116.

Attempt V: Mutual Exclusion 2/12

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);
```

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

Attempt V: Mutual Exclusion 3/12

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);
```

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

Attempt V: Mutual Exclusion 4/12

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);
```

- If processes P_i and P_j are both in their critical sections, then we have:
  - `flag[i]` and `flag[j]` are both `TRUE`.
  - `flag[i] && turn == i` and `flag[j] && turn == j` are both `FALSE`.
  - Therefore, `turn == i` and `turn == j` must both be `FALSE`.

they are both TRUE

Attempt V: Mutual Exclusion 5/12

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);
```

- **Because `turn == i` and `turn == j` are both `FALSE`, `turn == j` and `turn == i` are both `TRUE`.**
- **This is impossible, because a variable (i.e., `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.**

Attempt V: Progress 8/12

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);
```

- **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.**

Attempt V: Bounded Waiting 9/12

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);
```

- **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.**

Attempt V: Bounded Waiting 10/12

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.

Attempt V: Bounded Waiting 11/12

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 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. 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.

Attempt V: Bounded Waiting 12/12

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 3**: If P_j is *in* its critical section, `turn` must be `j` and P_i waits for at most one round.

P_i	P_j	flag[i]	flag[j]	turn	Comments
flag[i]=T	flag[j]=T	TRUE	TRUE	?	
while (…)		TRUE	TRUE	j	P_j enters
	Critical Sec				P_j in CS
	flag[j]=F	TRUE	FALSE	j	P_j exits
	flag[j]=T	TRUE	TRUE	j	P_j returns
	turn = i	TRUE	TRUE	i	P_j yields
	while (…)	TRUE	TRUE	i	P_j loops
Critical Sec					P_i enters

If P_i can see this, it enters

P_i has a chance to enter here.

if P_j comes back fast

One More Example: 1/4

- Consider the following simple algorithm:

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

interested

equal to the other

not equal to the other

not interested

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**:

```
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

interested

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

not interested

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

- If P_0 is in CS, `flag[0]` is `TRUE`, `flag[1]` is `FALSE` **OR** `turn[0] != turn[1]`
- If P_1 is in CS, `flag[1]` is `TRUE`, `flag[0]` is `FALSE` **OR** `turn[0] == turn[1]`
- If P_0 and P_1 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**:

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

Process P_0 | **Process P_1**

interested

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

Critical Section

not interested

```
flag[0] = FALSE;                   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 P_0 is entering.

```
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

interested

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

not interested

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

- 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 `FALSE` when P_1 exits, P_0 enters (P_0 waits for 1 round). Or, P_1 comes back to set `flag[1]` to `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

```
do {
                    entry
    disable interrupts
    critical section
    enable interrupts
                    exit
} while (1);
```

- **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

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

```
bool  lock = FALSE;

do {
     ⋮
     while (TS(&lock));      entry
     critical section
     lock = FALSE;           exit
     ⋮
} while (1);
```

A process is in its critical section if the `TS` instruction returns `FALSE`.

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

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**!

```
bool  lock = FALSE;

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

Compare-and-Swap: 1/2

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

```
bool  lock = FALSE;

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

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

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

Compare-and-Swap: 2/2

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

```
int   count = 0;

done = FALSE;
while (!done) {
   val = *count;
   done = CS(&count,val,val+1);
}
```

- **`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