Operating Systems and Concurrency

Lecture 8: Concurrency

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

- 25/10/2018, 16:00 - 17:00 will be revision
  - Please send me an email with topics that you would like to re-visit
  - We will go through past exam questions
- Make sure you download the last version of the slides before the lectures

Recap
Last Lecture

- Examples of **concurrency issues** (e.g. `counter++`)
- Root causes of **concurrency issues** (parallel code, race conditions, registers vs. variables)
- **Critical sections**, **mutual exclusion**, and **deadlocks**

- **Software based**: Peterson's solution
- **Hardware based**: disable **interrupts**, `test_and_set()`, `swap_and_compare()`

- **OS based**:
  - Mutexes
  - Semaphores

- **Monitors:** software construct within the programming languages

Peterson's Solution
Software Solution

- Peterson's solution is a **software based solution** which worked well on **older machines**
- Two **shared variables** are used:
  - `turn`: indicates **which process is next** to enter its critical section
  - `boolean flag[2]`: indicates that a **process is ready** to enter its critical section
- Can be **generalised to multiple processes**
- Peterson's solution for two processes **satisfies all "critical section requirements"** (mutual exclusion, progress, fairness)

Peterson’s Solution

Software Solution

```
do {
    flag[i] = true; // i wants to enter critical section
    turn = j;       // allow j to access first
    while (flag[j] && turn == j);
    // whilst j wants to access critical section
    // and its j's turn, apply busy waiting

    // CRITICAL SECTION, e.g. counter++

    flag[i] = false;

    // remainder section
} while (...);
```

Figure: Peterson’s solution for process *i*

Peterson’s Solution
Software Solution

```
do {
      flag[j] = true; // j wants to enter critical section
      turn = i;       // allow i to access first
      while (flag[i] && turn == i);
      // whilst i wants to access critical section
      // and its i’s turn, apply busy waiting

      // CRITICAL SECTION, e.g. counter++

      flag[j] = false;

      // remainder section
} while (...);
```

Figure: Peterson’s solution for process *j*

Peterson’s Solution
Software Solution

```
flag[i] = false;                    flag[j] = false;
...                                 ...
do {                                do {
   flag[i] = true;                     flag[j] = true;
   turn = j;                           turn = i;
   while (flag[j] && turn == j);       while (flag[i] && turn == i);
   counter++;                          counter++;
   flag[i] = false;                    flag[j] = false;
} while (...);                      } while (...);
```

Process i   *Process j*

Peterson’s Solution
Software Solution

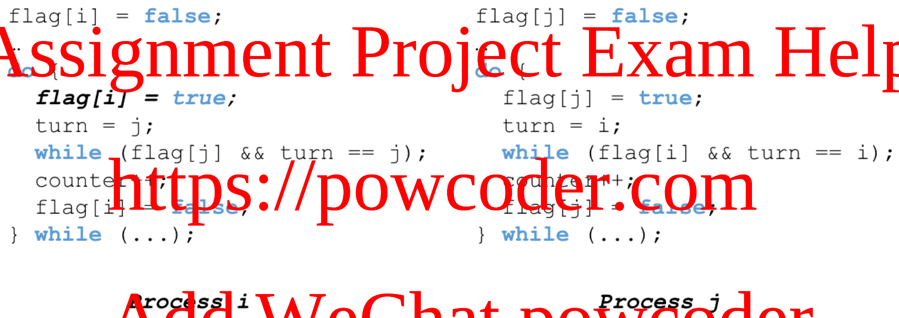
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

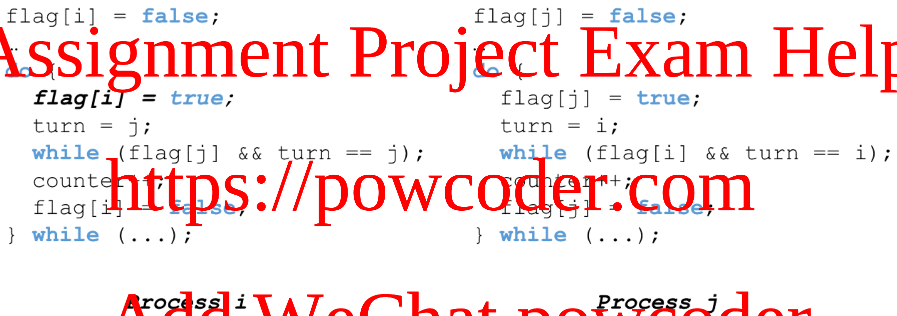
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

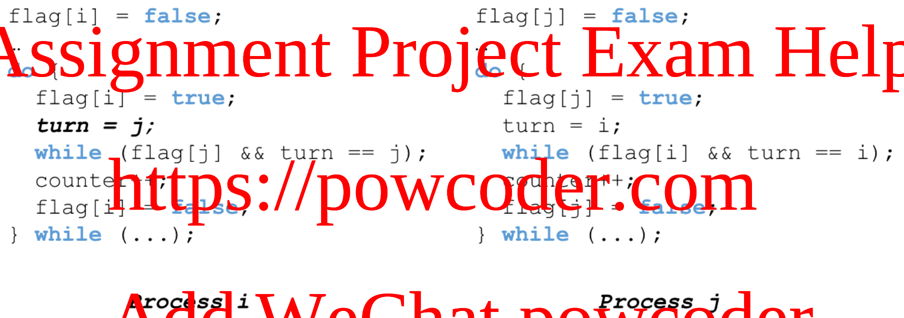
Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  counter++;
  flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  counter++;
  flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

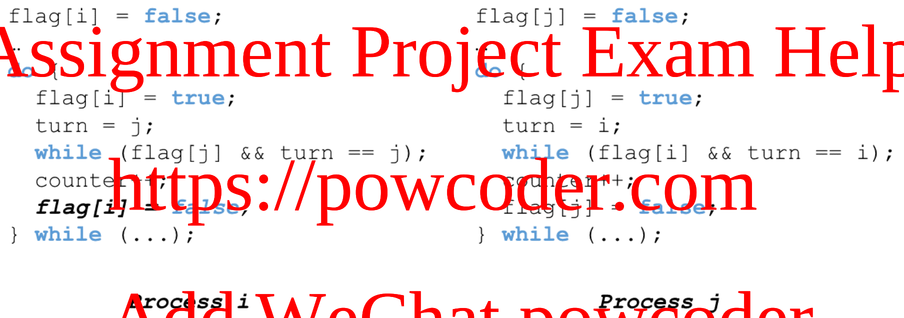
Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);

          Process i
```

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);

          Process j
```

Peterson’s Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

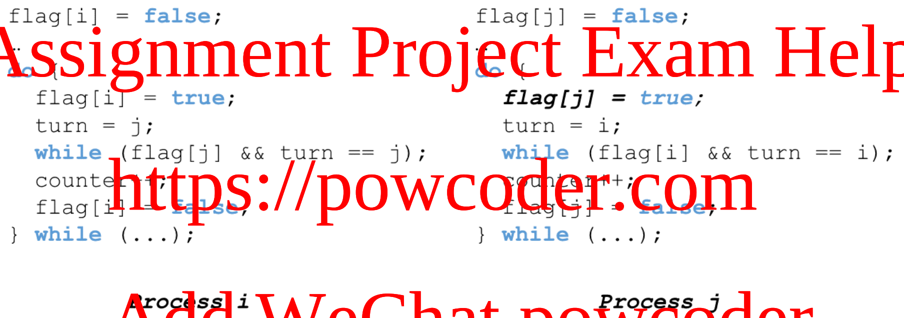
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

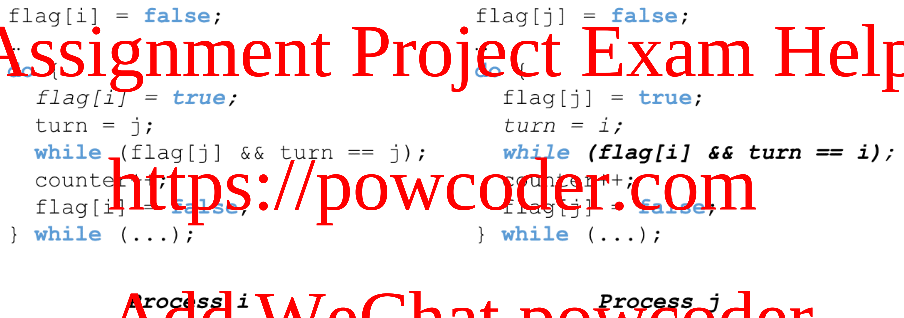
Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Software Solution

```
flag[i] = false;
...
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  counter++;
  flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  counter++;
  flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Software Solution

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Mutual exclusion requirement

- **Mutual exclusion requirement:** the variable turn can have **at most one value at a time**
  - Both `flag[i]` and `flag[j]` are `true` when they want to enter their critical section
  - Turn is a **singular variable** that can store **only one value**
  - Hence, either `while(flag[i] && turn == i)` or `while(flag[j] && turn == j)` is true and at most **one process can enter its critical section** (mutual exclusion)

Peterson's Solution

Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Mutual Exclusion Requirement

```
flag[i] = false;                        flag[j] = false;
...                                     ...
do {                                    do {
   flag[i] = true;                         flag[j] = true;
   turn = j;                               turn = i;
   while (flag[j] && turn == j);           while (flag[i] && turn == i);
   counter++;                              counter++;
   flag[i] = false;                        flag[j] = false;
} while (...);                          } while (...);
```

Process i   *Process j*

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  counter++;
  flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  counter++;
  flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

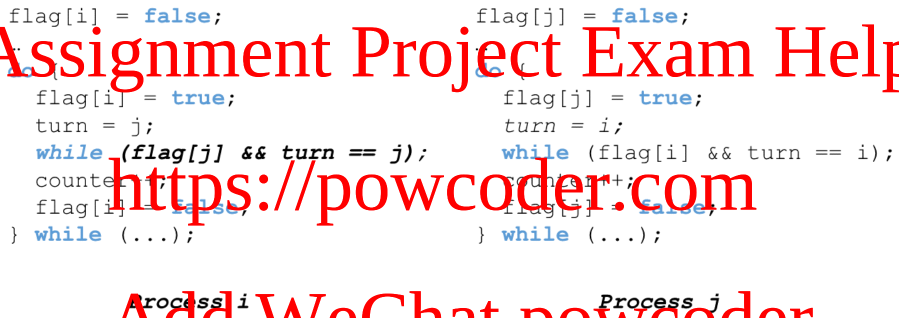
Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Mutual Exclusion Requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process *i*

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process *j*

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Mutual Exclusion Requirement

```
flag[i] = false;                     flag[j] = false;
...                                  ...
do {                                 do {
   flag[i] = true;                      flag[j] = true;
   turn = j;                            turn = i;
   while (flag[j] && turn == j);        while (flag[i] && turn == i);
   counter++;                           counter++;
   flag[i] = false;                     flag[j] = false;
} while (...);                       } while (...);
```

Process i *Process j*

Peterson’s Solution

Mutual Exclusion Requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

- **Progress**: any process must be able to enter its critical section at some point in time
  - Processes/threads in the “**remaining code**” **do not influence** access to critical sections
  - If process *j* does **not want to enter** its critical section

    ⇒ `flag[j] == false`

    ⇒ `while(flag[j] && turn == j)` will terminate for process *i*

    ⇒ *i* enters critical section

Peterson’s Solution
Progress requirement

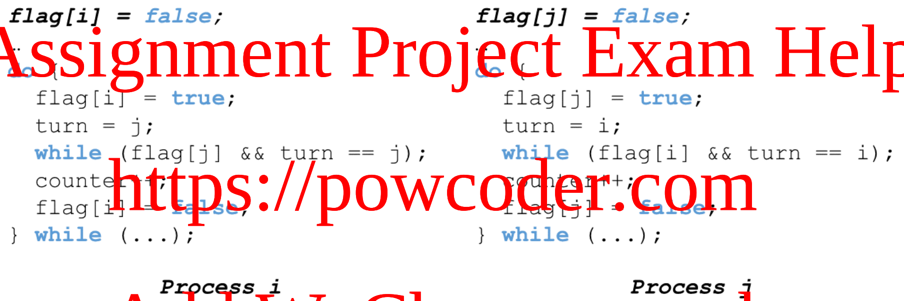
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Progress requirement



```
flag[i] = false;                     flag[j] = false;
...                                  ...
do {                                 do {
   flag[i] = true;                      flag[j] = true;
   turn = j;                            turn = i;
   while (flag[j] && turn == j);        while (flag[i] && turn == i);
   counter++;                           counter++;
   flag[i] = false;                     flag[j] = false;
} while (...);                       } while (...);

          Process i                            Process j
```

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

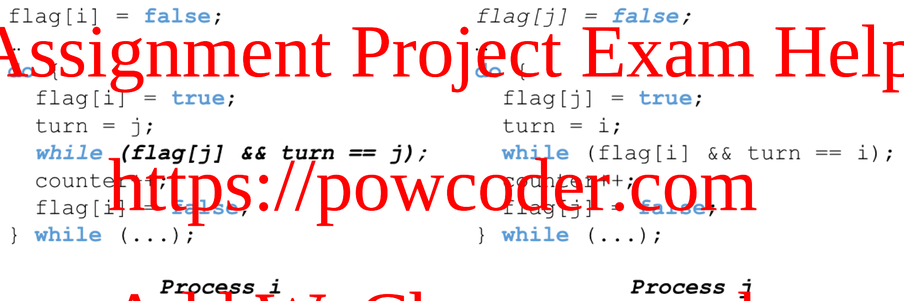
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);

            Process i
```

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);

            Process j
```

Peterson’s Solution
Progress requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

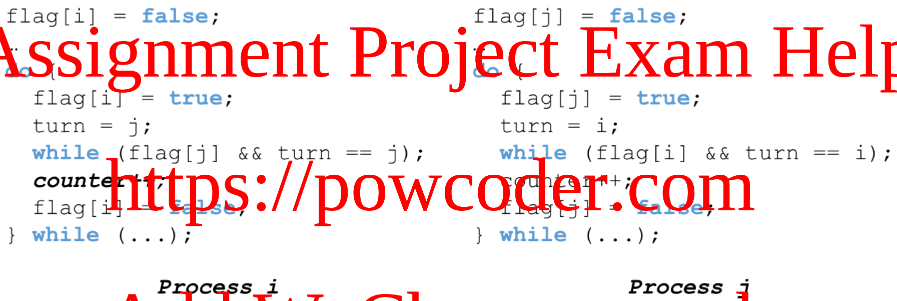
Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

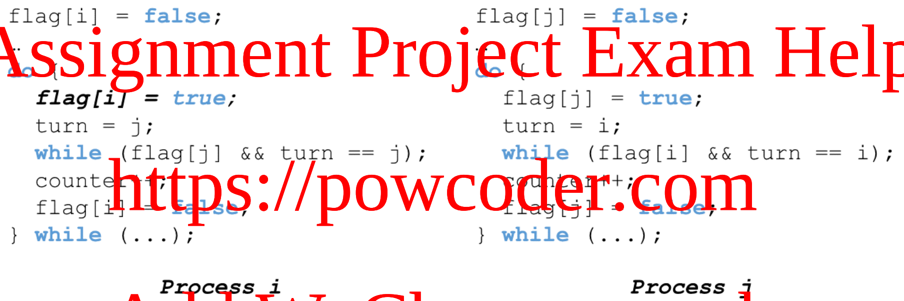
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Progress requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution

Progress requirement

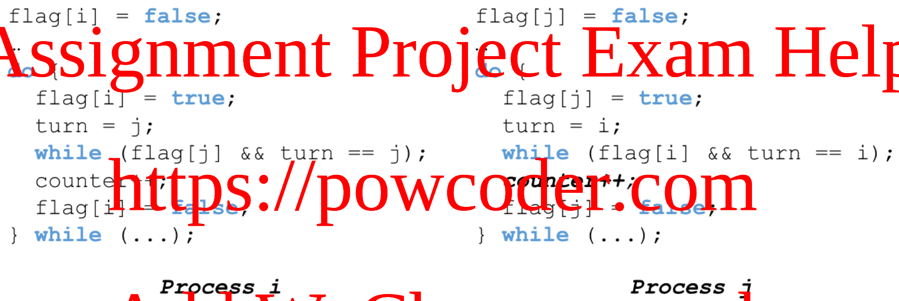
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;                        flag[j] = false;
...                                     ...
do {                                    do {
  flag[i] = true;                         flag[j] = true;
  turn = j;                               turn = i;
  while (flag[j] && turn == j);           while (flag[i] && turn == i);
  counter++;                              counter++;
  flag[i] = false;                        flag[j] = false;
} while (...);                          } while (...);
```

Process i ***Process j***

Peterson's Solution

Fairness/bounded waiting requirement

- **Fairness/bounded waiting**: fairly distributed waiting times/processes cannot be made to wait indefinitely
  - If P_i and P_j both want to enter their critical section
    ⇒ `flag[i] == flag[j] = true`
    ⇒ `turn` is either *i* or *j* ⇒ assuming that `turn == i` ⇒
    `while(flag[j] && turn == j)` terminates and *i* enters section
    ⇒ *i* finishes critical section ⇒ `flag[i] = false` ⇒
    `while(flag[i] && turn == i)` terminates and *j* enters critical section

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Progress requirement

```
flag[i] = false;
...
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  counter++;
  flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  counter++;
  flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;                      flag[j] = false;
...                                   ...
do {                                  do {
   flag[i] = true;                       flag[j] = true;
   turn = j;                             turn = i;
   while (flag[j] && turn == j);         while (flag[i] && turn == i);
   counter++;                            counter++;
   flag[i] = false;                      flag[j] = false;
} while (...);                        } while (...);

            Process i                             Process j
```

Peterson’s Solution
Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson's Solution
Progress requirement

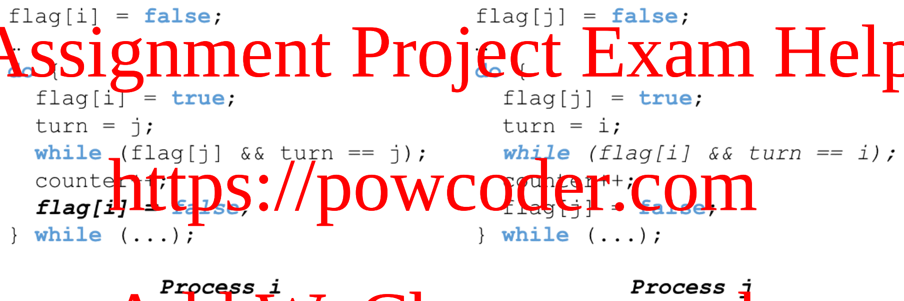
```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement



```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);

              Process i
```

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);

              Process j
```

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   counter++;
   flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
   counter++;
   flag[j] = false;
} while (...);
```

Process j

Peterson’s Solution
Progress requirement

```
flag[i] = false;                          flag[j] = false;
...                                       ...
do {                                      do {
  flag[i] = true;                           flag[j] = true;
  turn = j;                                 turn = i;
  while (flag[j] && turn == j);             while (flag[i] && turn == i);
  counter++;                                counter++;
  flag[i] = false;                          flag[j] = false;
} while (...);                            } while (...);

              Process i                                 Process j
```

Peterson’s Solution

Progress requirement

```
flag[i] = false;
...
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  counter++;
  flag[i] = false;
} while (...);
```

Process i

```
flag[j] = false;
...
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  counter++;
  flag[j] = false;
} while (...);
```

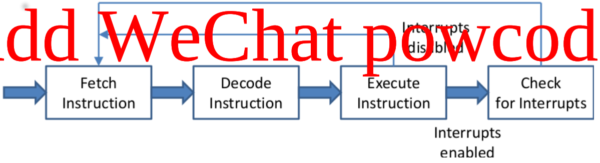
Process j

Disabling Interrupts

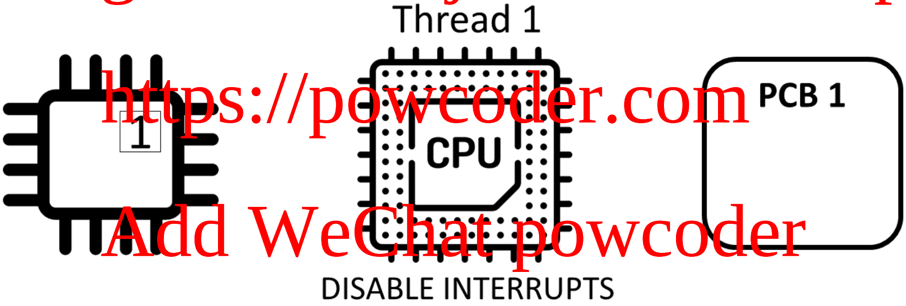
Hardware Approaches

- **Disable interrupts** whilst **executing a critical section** and prevent interruption (i.e., interrupts from timers, I/O devices, etc.)
    - Think of the `counter++` example

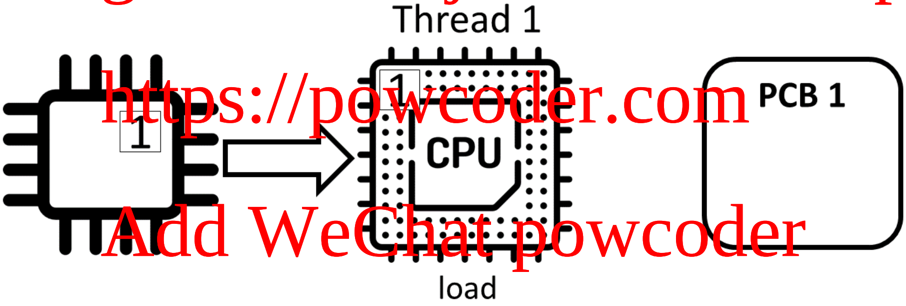
```
register = counter;
register = register + 1
counter = register;
```



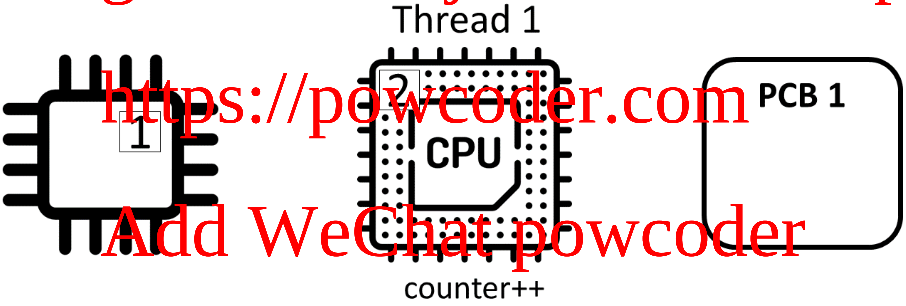
Disabling Interrupts
Hardware Approaches



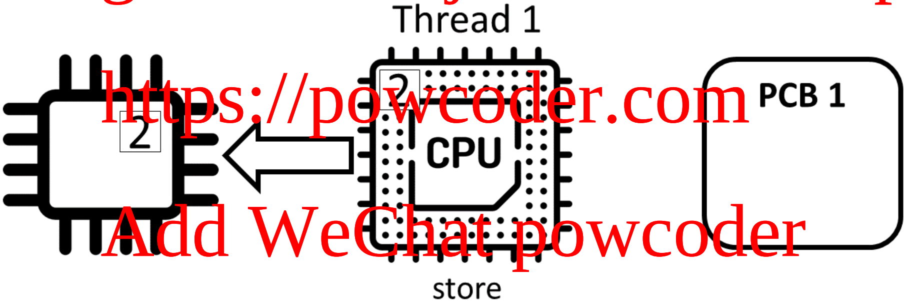
Disabling Interrupts
Hardware Approaches



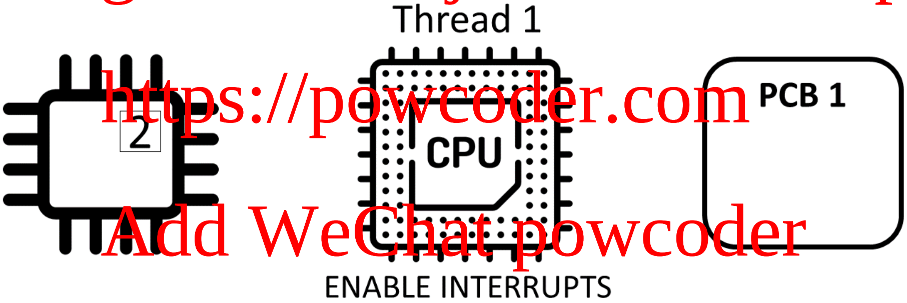
Disabling Interrupts
Hardware Approaches



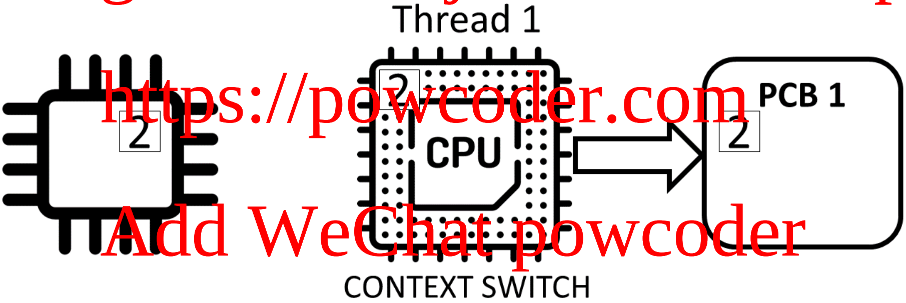
Disabling Interrupts
Hardware Approaches



Disabling Interrupts
Hardware Approaches

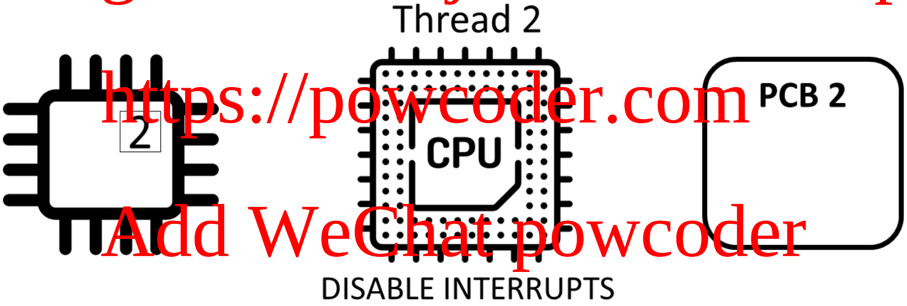


Disabling Interrupts
Hardware Approaches

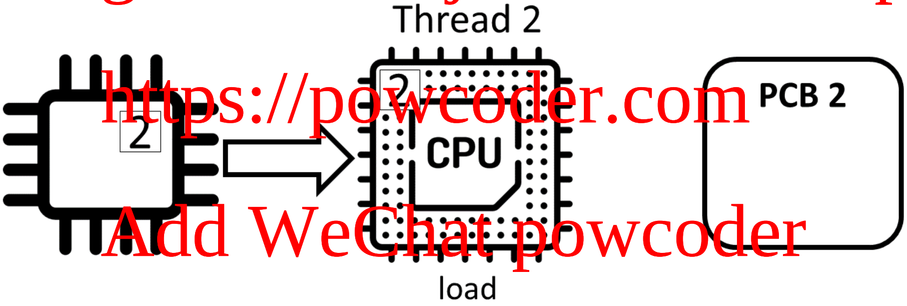


Disabling Interrupts

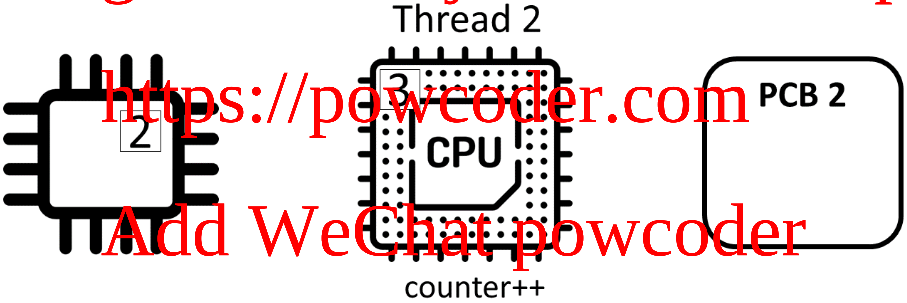
Hardware Approaches



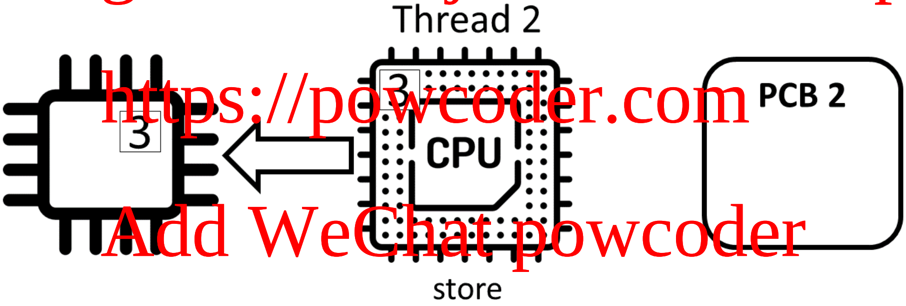
Disabling Interrupts
Hardware Approaches



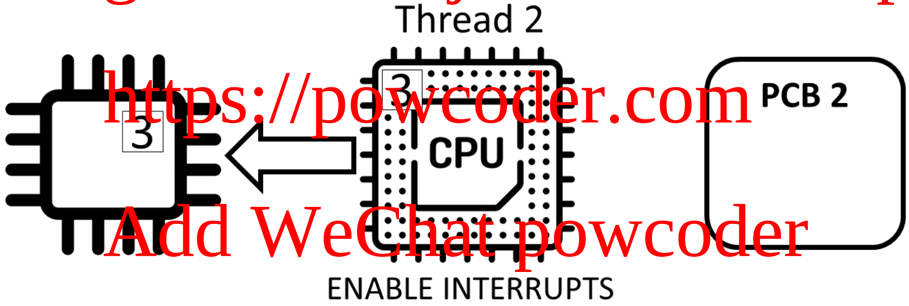
Disabling Interrupts
Hardware Approaches



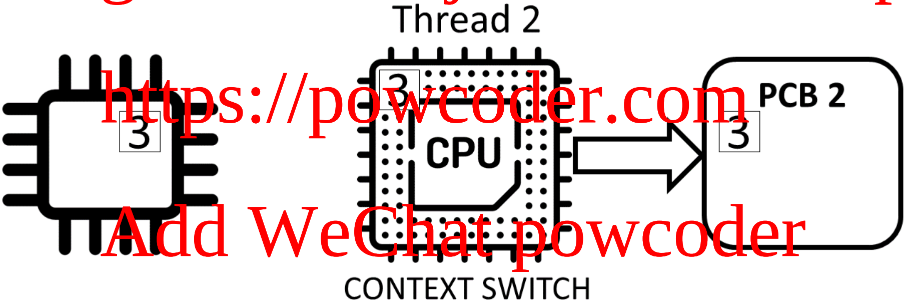
Disabling Interrupts
Hardware Approaches



Disabling Interrupts
Hardware Approaches

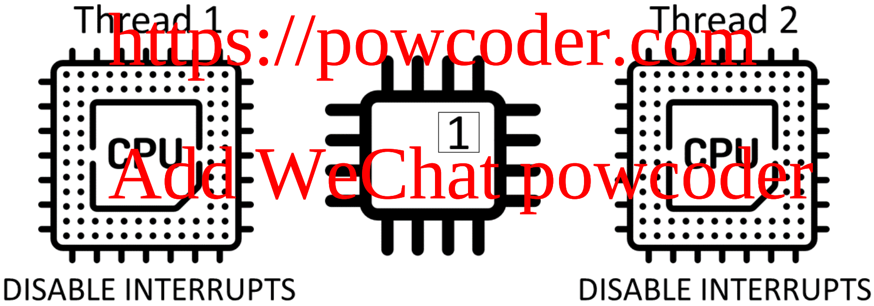


Disabling Interrupts
Hardware Approaches



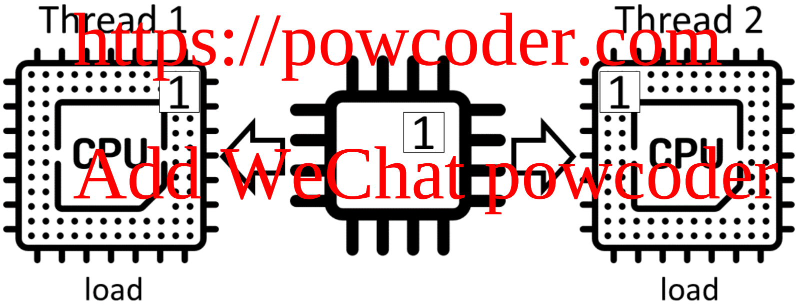
Disabling Interrupts
Hardware Approaches

- Disabling interrupts “may” be appropriate on a **single CPU machine**
- This **is insufficient on modern multi-core/multi-processor machines**



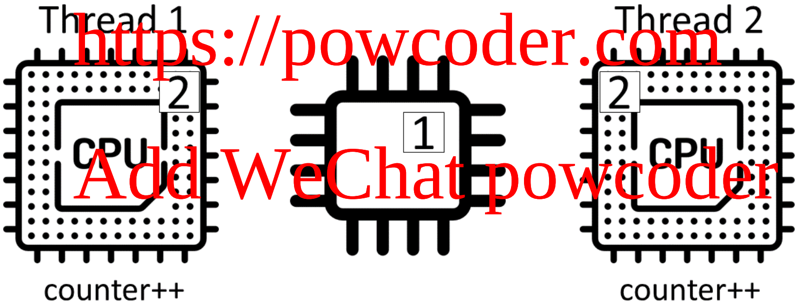
Disabling Interrupts
Hardware Approaches

- Disabling interrupts “may” be appropriate on a **single CPU machine**
- This is **insufficient on modern multi-core/multi processor machines**



Disabling Interrupts
Hardware Approaches

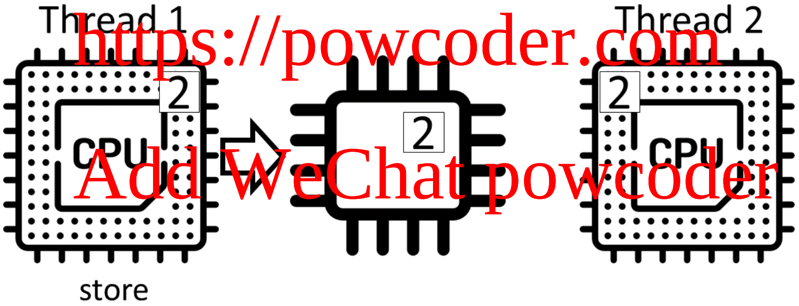
- Disabling interrupts “may” be appropriate on a **single CPU machine**
- This **is insufficient on modern multi-core/multi processor machines**



Disabling Interrupts

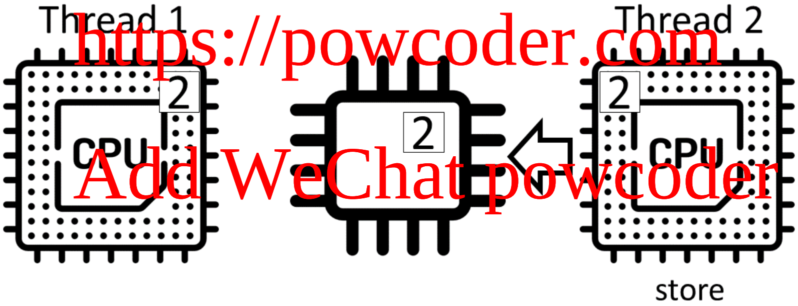
Hardware Approaches

- Disabling interrupts “may” be appropriate on a **single CPU machine**
- This is **insufficient on modern multi-core/multi processor machines**



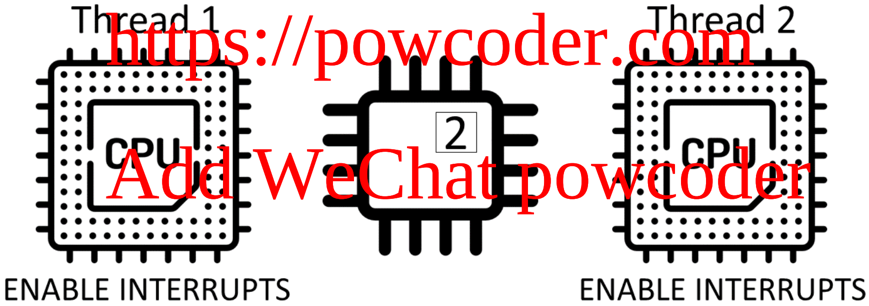
Disabling Interrupts
Hardware Approaches

- Disabling interrupts "may" be appropriate on a **single CPU machine**
- This is **insufficient on modern multi-core/multi processor machines**



Disabling Interrupts
Hardware Approaches

- Disabling interrupts "may" be appropriate on a **single CPU machine**
- This is **insufficient on modern multi-core/multi processor machines**



Atomic Instructions

Hardware approaches

- Implement `test_and_set()` and `swap_and_compare()` instructions as a **set of atomic (= uninterruptible) instructions**
  - Reading and setting the variable(s) is done as **one "complete" set of instructions**
  - If `test_and_set()` / `compare_and_swap()` are **called simultaneously**, they will be **executed sequentially**
- They are used in in combination with **global lock variables**, assumed to be `true (1)` if the lock is **in use**

Atomic Instructions

Hardware approaches

```
// Test and set method
boolean test_and_set(boolean * bIsLocked) {
    boolean rv = *bIsLocked;
    *bIsLocked = true;
    return rv;
}
```

```
// Example of using test and set method
do {
        // WHILE the lock is in use, apply busy waiting
        while (test_and_set(&bIsLocked));
        // Lock was false, now true

        // CRITICAL SECTION
        ...
        lock = false;
        ...
        // remainder section
} while (...)
```

Atomic Instructions

Hardware approaches: test_and_set() and compare_and_swap()

- Test and set must be **atomic/UN-interruptable**

```
THREAD 1                           THREAD 2
...                                ...
boolean rv = *bIsLocked;           ...
...                                boolean rv = *bIsLocked;
*bIsLocked = true;                 ...
return rv;                         ...
...                                *bIsLocked = true;
...                                return rv;


---                                ---

while (test_and_set(&bIsLocked));  while (test_and_set(&bIsLocked));
```

Atomic Instructions

Hardware approaches: test_and_set() and compare_and_swap()

- Test and set must be **atomic/UN-interruptable**

```
THREAD 1                            THREAD 2
...                                 ...
boolean rv = *bIsLocked;            ...
*bIsLocked = true;                  ...
return rv;                          ...
...                                 boolean rv = *bIsLocked;
...                                 *bIsLocked = true;
...                                 return rv;


---                                 ---

while (test_and_set(&bIsLocked));   while (test_and_set(&bIsLocked));
```

Atomic Instructions

Hardware approaches: compare_and_swap()

```
// Compare and swap method
int compare_and_swap(int* iIsLocked, int expected, int new_value) {
  int temp = *iIsLocked;
  if(*iIsLocked == expected)
    *iIsLocked = new_value;
  return temp;
}
```

```
 // Example using compare and swap method
 do {
         // While the lock is in use (i.e. == 1), apply busy waiting
         while (compare_and_swap(&iIsLocked, 0, 1) != 0);
         // Lock was false, now true

         // CRITICAL SECTION
         ...
         lock = 0;
         ...
         // remainder section
 } while (...);
```

- `test_and_set()` and `compare_and_swap()` are **hardware instructions** and (usually) **not directly accessible** to the user
  - Rembember, the OS hides the "bare metal" from the user

- Other disadvantages include:
  - **Busy waiting** is used
  - **Deadlock** is possible, e.g. when two locks are requested in opposite orders in different threads

- The **OS uses the hardware instructions** to implement **higher level mechanisms/instructions** for mutual exclusion, i.e. **mutexes** and **semaphores**

Mutex Locks
OS approaches

- **Mutexes** are an approach for mutual exclusion **provided by the operating system** containing a **boolean** lock variable to indicate availability
  - The lock variable is set to **true** if the lock **is available** (process can enter critical section), **false** if **not**

- Two **atomic functions** are used to **manipulate the mutex**:
  - `acquire()`: called **before entering** a critical section, boolean set to **false**
  - `release()`: called **after exiting** the critical section, boolean set to **true** again

Mutex Locks
OS approaches

```
acquire() {
   while (!available)
      ;   // busy wait
   available = false;
}
```

Figure: Conceptual implementation of acquire()

```
release() {
   available = true;
}
```

Figure: Conceptual implementation of release()

Mutex Locks
OS approaches (Cont'ed)

- `acquire()` and `release()` must be **atomic instructions**
  - No **interrupts** should occur between reading and setting the lock
  - If interrupts can occur, the follow sequence could occur:

```
T_i => lock available
...                                T_j => lock available
...                                T_j sets lock
T_i sets lock                      ...
```

- The **process that acquires** the lock **must release** the lock (in contrast to semaphores – see later)

- The key **disadvantage** of mutex locks is that calls to `acquire()` result in **busy waiting** (although this appears to be OS dependent)
  - **Detrimental for performance** on single CPU systems

- The key **advantages** of mutex locks include:
  - **Context switches** can be avoided (short critical sections)
  - **Efficient on multi-core/multi-processor** systems when **locks are held for a short time** only

Mutex Locks in Linux
Example

```
//includes here
int sum = 0;
pthread_mutex_t lock;

void * calc(void * number_of_increments)
{ int i;
  for(i = 0; i < *((int*) number_of_increments);i++)
  { pthread_mutex_lock(&lock);
    sum++;
    pthread_mutex_unlock(&lock);
  }
}
int main()
{ int iterations = 50000000;
  pthread_t tid1,tid2;
  pthread_mutex_init(&lock,NULL);
  // no error checking for clarity/brevity
  pthread_create(&tid1, NULL, calc, (void *) &iterations);
  pthread_create(&tid2, NULL, calc, (void *) &iterations);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("the value of sum is: %d\n", sum);
}
```

Mutex Locks in Linux
Example

```
THREAD 1                 THREAD 2

...                      ...
mutex_lock               ...
sum++                    mutex_lock
mutex_unlock             // busy wait
mutex_lock               sum++
// busy_wait             mutex_unlock
sum++                    mutex_lock
mutex_unlock             // busy wait
mutex_lock               sum++
...                      ...
```

- Software based approach: **Peterson's** solution (software)
- Hardware based approaches:
  - disabling interrupts
  - atomic instructions: (`test_and_set`, `compare_and_swap`)
- OS based approach: **Mutexes**