Operating Systems and Concurrency

Concurrency 2

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Coursework

- The coursework and supporting source code is now available on Moodle.
- The recommended submission deadline is **Monday 11th December, 2023**.
- The latest submission deadline is **Thursday 4th January, 2024**.
- No late submissions after that deadline!
- There will be a short introduction **next** weeks lab on **Friday 27th October, 2023**.

A question from last time

Question (paraphrased)

The slides last time defined deadlocks and critical sections in terms of processes - was this a typo?

A question from last time

Question (paraphrased)

The slides last time defined deadlocks and critical sections in terms of processes - was this a typo?

Answer

It doesn't really make very much difference. For the definitions in question, you need two key things:

- **Multiple sequences of instructions being executed concurrently** - both threads and processes satisfy this property.
- **Shared resources** - sharing is easier for threads. There are mechanisms to share resources between processes, including resources that **satisfy mutual exclusion**. A standard example would be **named semaphores**.

Recap

Last Lecture

- Examples of **concurrency issues** (e.g. `counter++`)
- Root causes of **concurrency issues** - Unprecictable order of execution.
- Potential solutions - **Critical sections** and enforcing **mutual exclusion**
- Futher problems - **deadlocks**.

Mutual Exclusion

Approaches for Mutual Exclusion

- **Software based**: Peterson's solution
- **Hardware based**: `test_and_set()`, `swap_and_compare()`
- **OS based**: Blocking in the kernel.

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
  - `bool 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;
…
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

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, at most one of `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;
…
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;
…
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

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

    $\Rightarrow$ `flag[j] == false`

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

    $\Rightarrow$ *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;
…
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

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
    $\Rightarrow$ `flag[i] == flag[j] = true`
    $\Rightarrow$ `turn` is either *i* or *j* $\Rightarrow$ assuming that `turn == i` $\Rightarrow$
    `while(flag[j] && turn == j)` terminates and *i* enters section
    $\Rightarrow$ *i* finishes critical section $\Rightarrow$ `flag[i] = false` $\Rightarrow$
    `while(flag[i] && turn == i)` terminates and *j* enters critical section.
    Even if it loops back round again, it will set $turn = j$, letting the other thread in first.

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

Atomic Instructions

Hardware approaches

- Implement `test_and_set()` and `swap_and_compare()` instructions as a **set of atomic (= uninterruptible) instructions**
  - Reading and setting the variables appears as **a single instruction**
  - If `test_and_set()` / `compare_and_swap()` are **called simultaneously**, they will be **executed sequentially**
- They can be used in in combination with **lock variables**, assumed to be `true` (or `1`) if the lock is **in use**

Atomic Instructions

Hardware approaches

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

---

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

```
THREAD 2
...
...
bool rv = *bIsLocked;
...
...
*bIsLocked = true;
return rv;

---

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
...
bool rv = *bIsLocked;
*bIsLocked = true;
return rv;
...
...
...

---

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

```
THREAD 2
...
...
...
...
bool rv = *bIsLocked;
*bIsLocked = true;
return rv;

---

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 const old_value = *iIsLocked;
  if(old_value == expected)
   *iIsLocked = new_value;
  return old_value;
}
```

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

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

Mutual Exclusion

Hardware approaches

- `test_and_set` and `compare_and_swap` are rather low level, and require busy waiting.
- The **OS may use these hardware instructions** to implement **higher level mechanisms** for mutual exclusion, i.e. **mutexes** and **semaphores**

Mutual Exclusion

Hardware approaches

- `test_and_set` and `compare_and_swap` are rather low level, and require busy waiting.
- The **OS may use these hardware instructions** to implement **higher level mechanisms** for mutual exclusion, i.e. **mutexes** and **semaphores**

Questions

We have two operations, `test_and_set` and `compare_and_swap` that have use as concurrency primitives. This raises some questions:

- What sort of primitive operations are useful?
- Are some better than others?
- Are there instructions that are best for concurrency support?

Mutexes

OS approaches

- **Mutexes** are an abstraction for providing **mut**ual **ex**clusion.
- A mutex provides an interface with two functions:
  - `acquire(&mutex)`: called **before entering** a critical section, returns when nobody else is in the critical section.
  - `release(&mutex)`: called **after exiting** the critical section, allows other threads to acquire the mutex. **Should only be called after a matching acquire**.
- Details of this interface may vary - names, or using methods in an object-oriented language.
- How exactly this interface is provided is an **implementation detail**.
  - Under naive assumptions, we could use Peterson's algorithm.
  - We could use atomic hardware operations and busy waiting.
  - The operating system could block threads that are trying to acquire a mutex that is not available.
  - Hybrid solutions combining several strategies might be chosen to optimise performance, depending on design assumptions.

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Pthreads Mutexes

Example

```
int counter = 0;
pthread_mutex_t lock;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++) {
    pthread_mutex_lock(&lock); // acquire
    counter++;
    pthread_mutex_unlock(&lock); // release
  }
  return 0;
}
int main() {
  pthread_t tid1 = 0, tid2 = 0;
  pthread_mutex_init(&lock,NULL);
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Test your understanding

- Can you implement mutual exclusion on concurrent hardware without either operating system or hardware support?
- Would a mutex implemented using Peterson's algorithm work on modern hardware? (Worth trying as an exercise)
- Would you need to use mutexes / a critical region to protect code that is only reading from variables?