Parallelism
Synchronous mechanism

Peter Levinsky IT, Roskilde

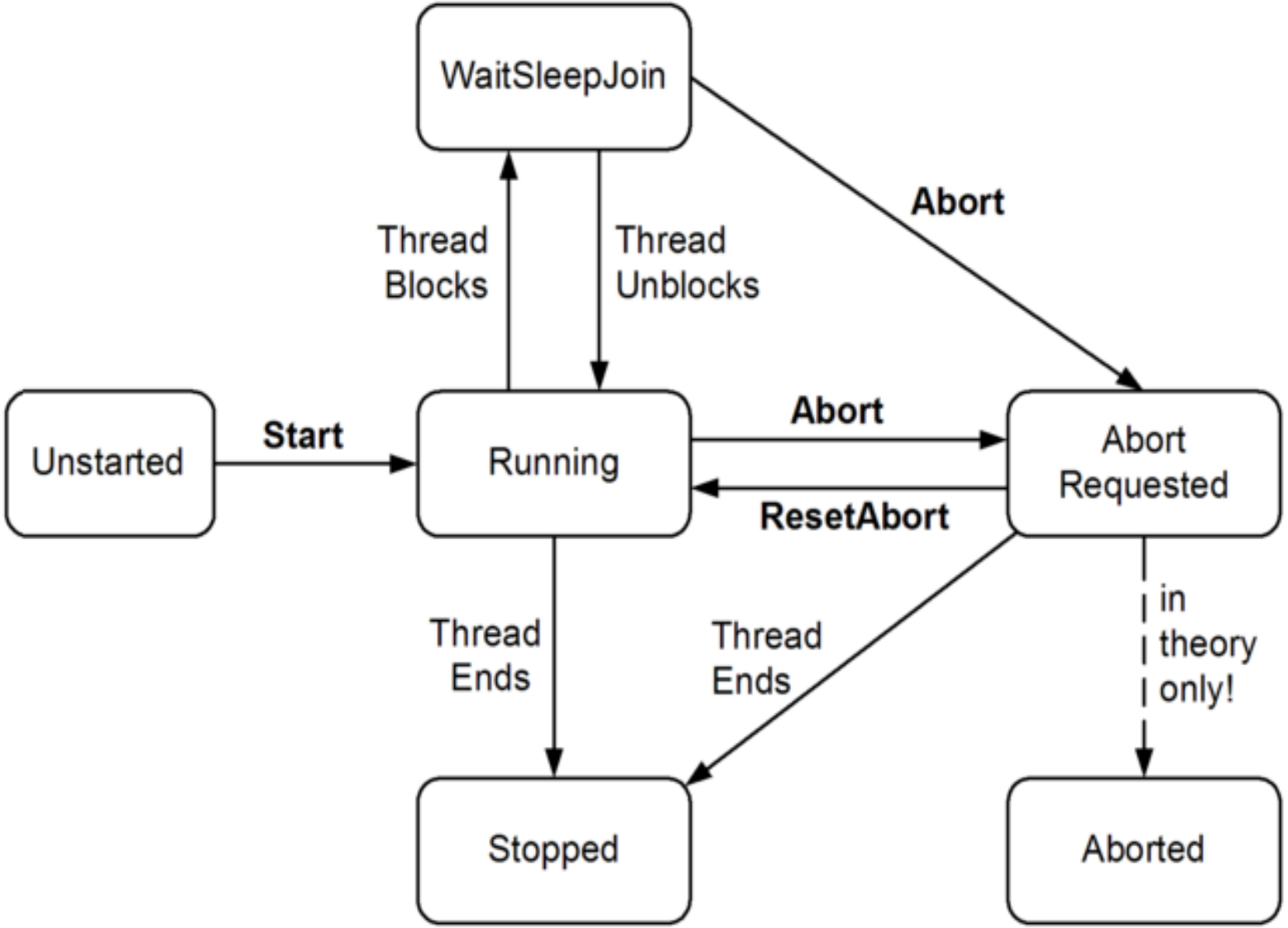
07.03.2022

Time consuming operations

Two categories

- CPU-bound operations
- I/O-bound operations

Thread Life cycle



Thread in C#

```
Thread t = new Thread (-- delegate Method --);
t.Start();
…
t.Join(); // wait here until t is completed
```

? Delegate Method

Thread in C# - executing

```
class ThreadTest
{
  static bool done;     // Static fields are shared between all threads

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    if (!done) { done = true; Console.WriteLine ("Done"); }
  }
}
```

Parallelism in C# - An Overview

Levels of parallelism:

- Thread -- Basic structure for parallelism (in most programming languages)
- Task -- C# smooth variant i.e. Task.Run(<<delegate method>>)
- Parallel.Invoke -- Can start several threads (continues after all thread is completed)
- Parallel.For/Foreach -- Can start several threads in a loop (continues after all thread is completed)
- Plinq -- Can execute a Linq expression in parallel

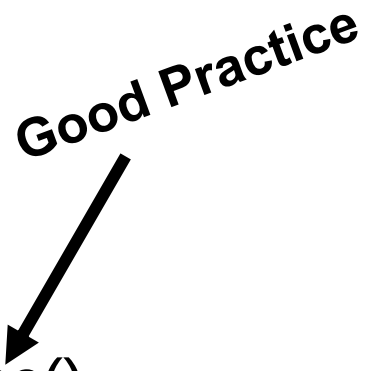
High End Parallelisme **async / await**

- Use of built in features **async / await**
  Do not create a new thread but make use of a coroutine i.e. program continue and ‘jumps’ back to the await call when it is ready.

- Where to use
  - I/O-bound operations – Like network, accessing files etc.

- How to use
  - Method is async – like public async Task<int> DoSomethingAsync()
  - In method body … somewhere

    await ….. return anInteger;



What is Async / Await ?

- The use of Async / Await is **not** directly the same as a **thread** / task !

- But the program will wait at ‘await’ until this job is done
- And you can continue do other stuff in between
  e.g. show information about ‘work in progress’ (Jacob Nielsen – System status)

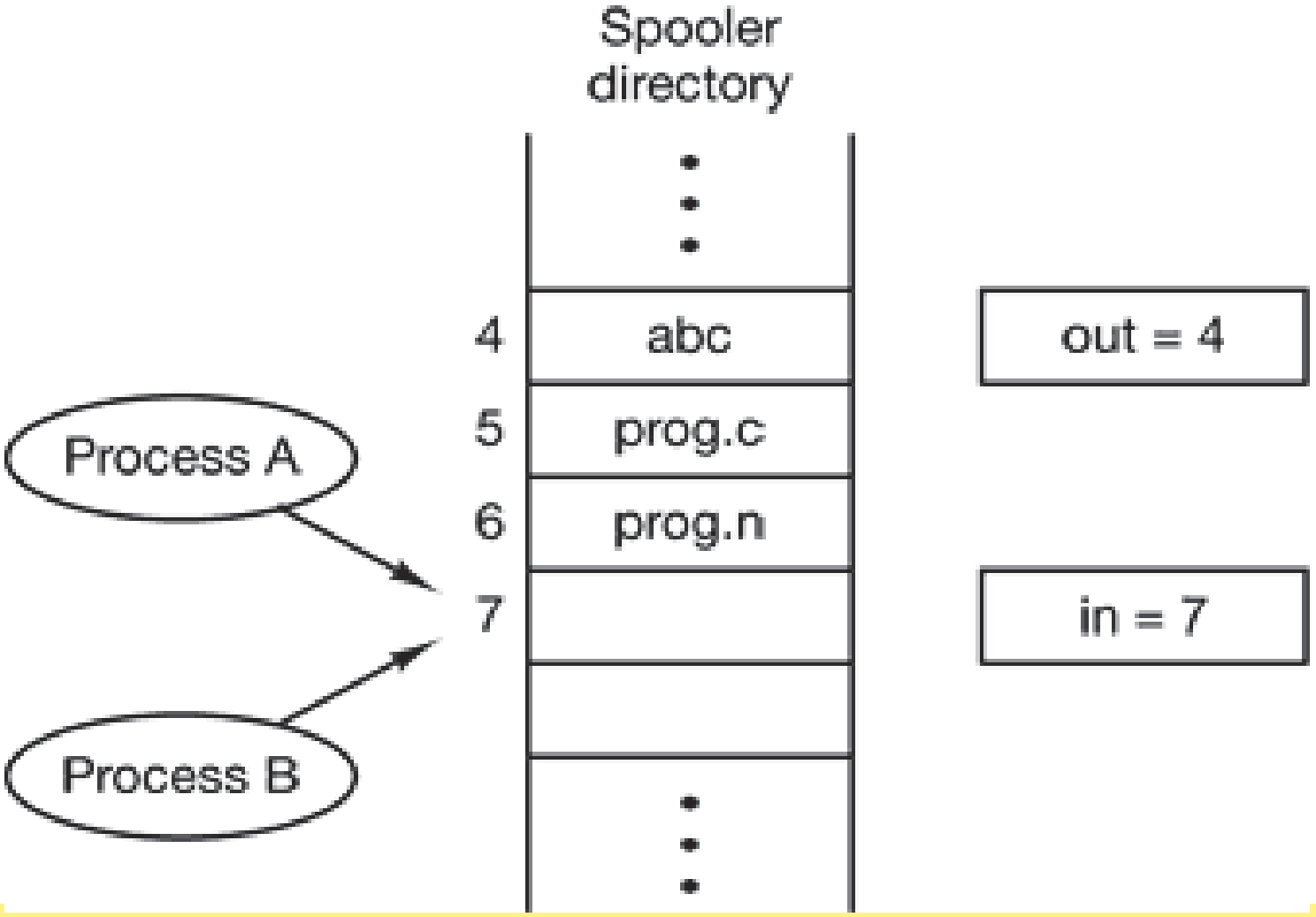
```
Task<List<Picture>> pictures = await ReadPicturesFromFile(“somefile.pic”);
Status = “Getting pictures …”;      // set system status
foreach(var pic in pictures.Result){
    …
}
```

Demo

Opgaver C#Exercises Prog.3.6+3.8

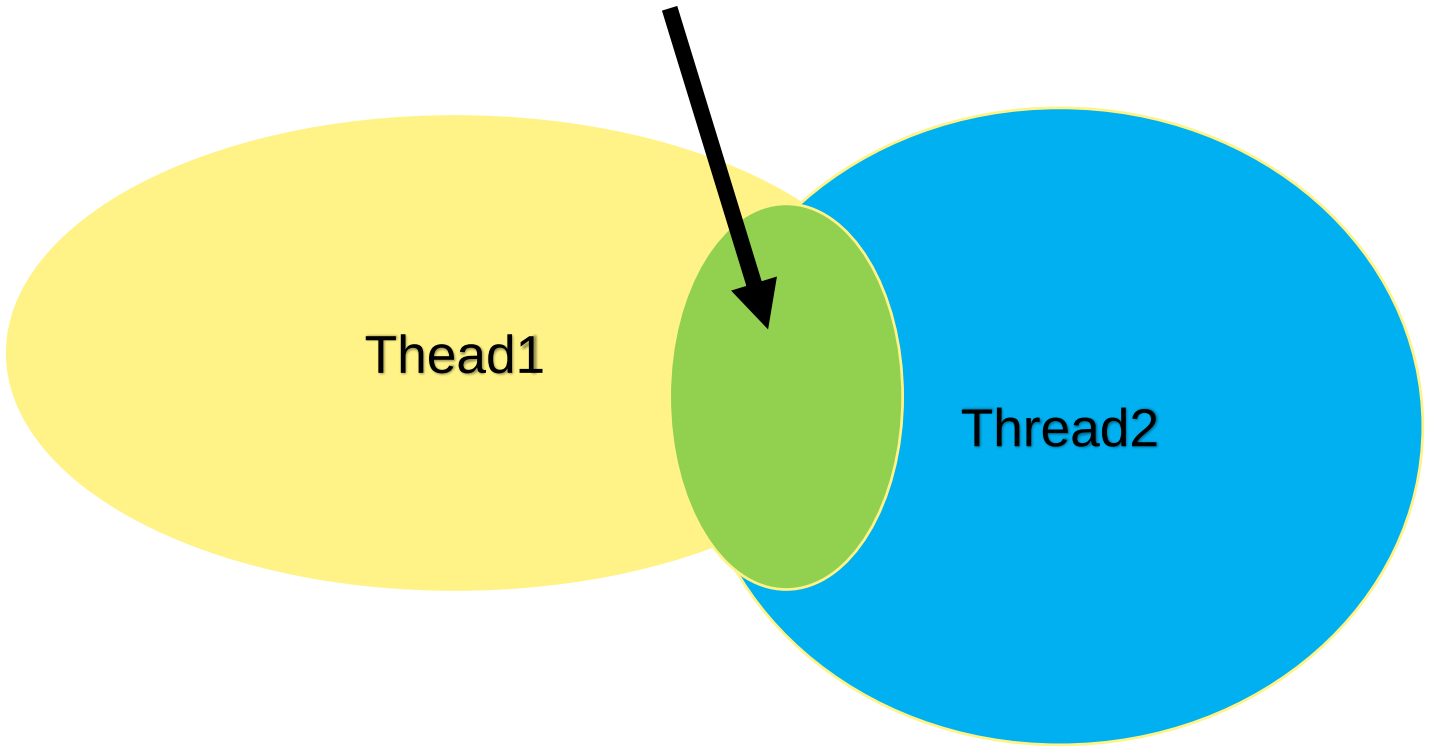
Synchronous Mechanism

Race Conditions:



Critical Regions

Common area (shared data) between several threads



Like ‘done’ in ThreadTest

Control of Critical Sections

A. Mutal Exclusion with busy waiting

   while (x != 0); // do nothing though loop again

   Petersons solution / TSL in machine language

B. Sleep and wakeup

   i. Lock
   ii. Semaphores
   iii. Mutex (binary semaphores)
   iv. Monitors

Overview Sleep and Wait

Lock

Ensure only one thread in block

Semaphore

Down for enter – count down by one if possible otherwise wait
Up for leave – increment by one if not reach roof (counting e.g. max 10)
C# waitOne, Release

Mutex

General like semaphore where roof is one
C# waitOne, ReleaseMutex

Monitor

The monitor are the critical section
Variable => conditions || Wait / signal
C# Enter / Exit

Classic Problems

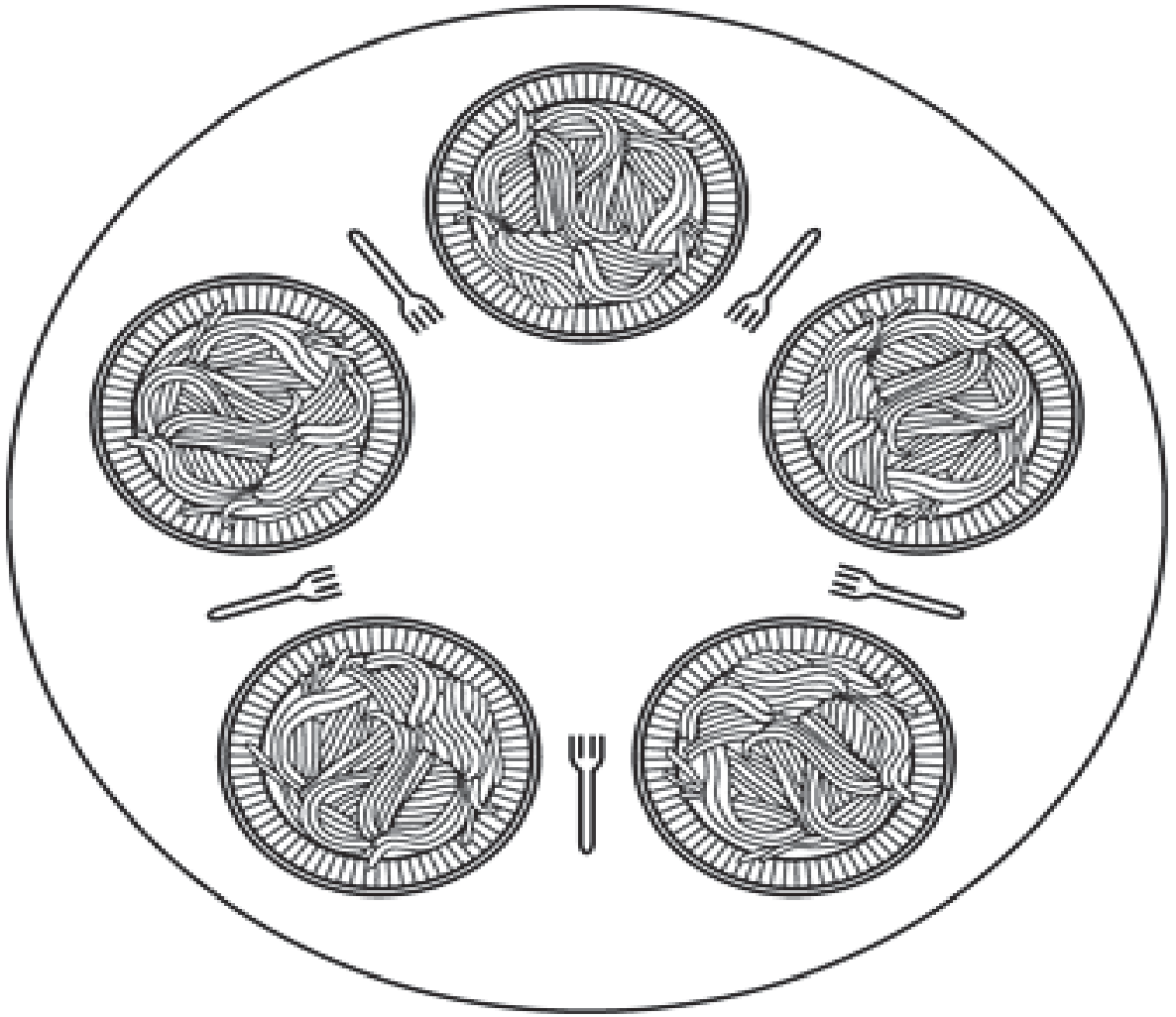
- The Dining Philosophers Problem

  Need two resources

The Dining Philosophers Problem

Philosophers do

Think
Eat



Example code for Dining philosophers

```
#define N 5/* number of philosophers */
void philosopher(int i)/* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think( ); /* philosopher is thinking */
    take_fork(i); /* take left fork */
    take_fork((i+1) % N);/* take right fork; % is modulo operator */
    eat(); /* yum-yum, spaghetti */
    put_fork(i); /* Put left fork back on the table */
    put_fork((i+1) % N);/* put right fork back on the table */
  }
}
```

Demo

Opgaver C#Exercises XXXX