Parallelism
Synchronous mechanism

Peter Levinsky IT, Roskilde

15.09.2024

Time consuming operations

Two categories

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

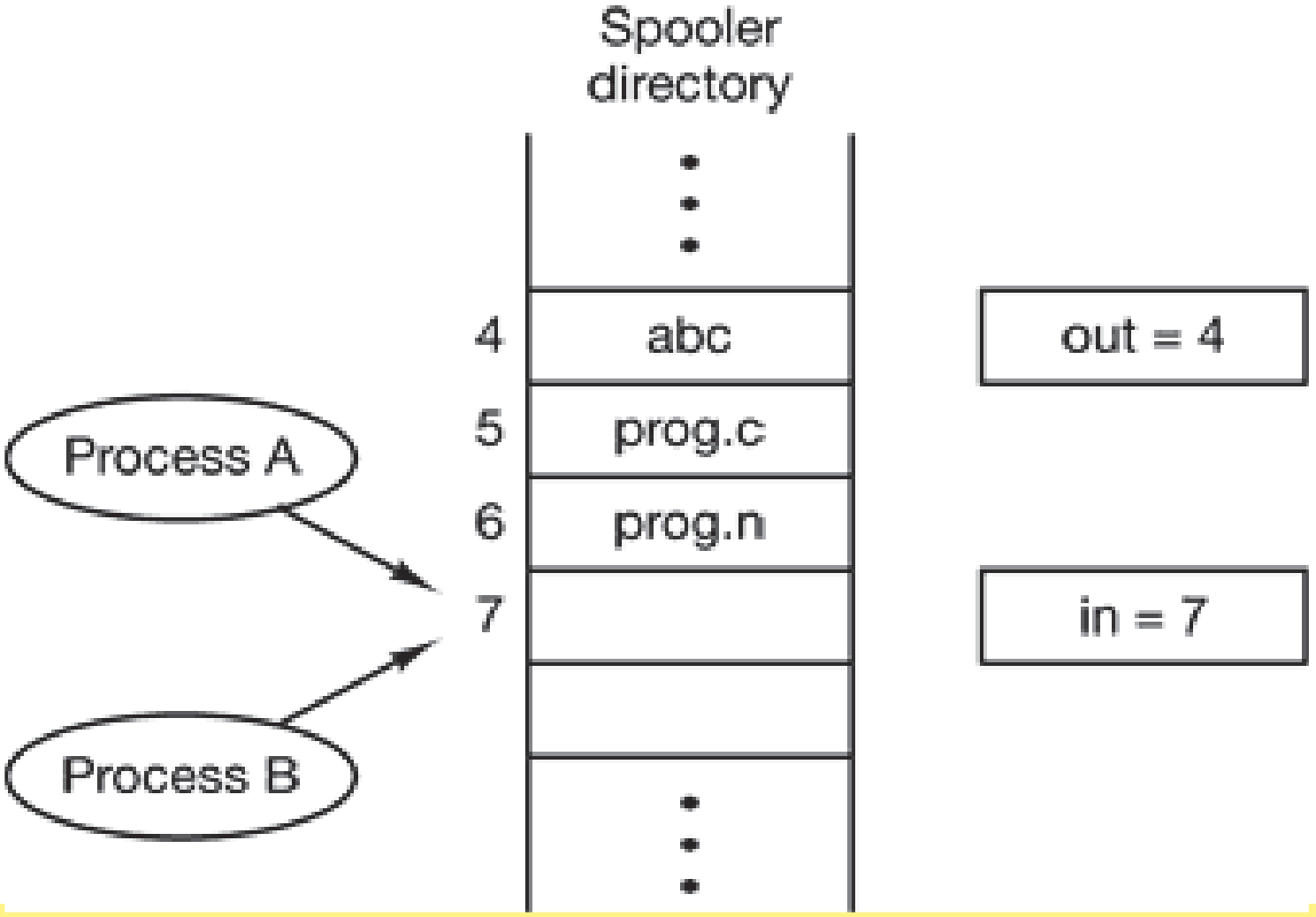
Parallelism in C#

Levels of parallelism:

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

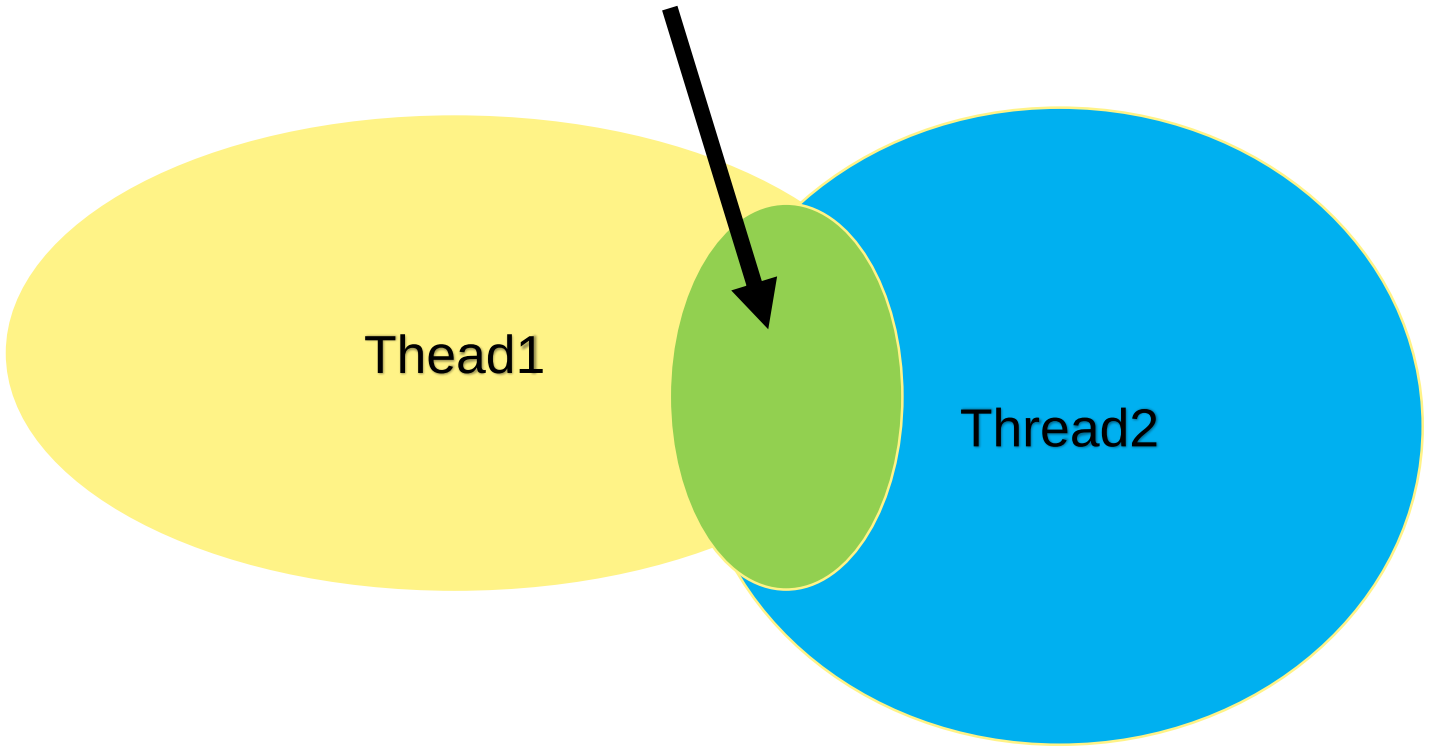
Synchronous Mechanism

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 (e.g. bounded buffer)

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

- Producer / Consumer

  Send data from producer to consumer – like a production line

- The Readers and Writers Problem

  Many reader (shared) one writer (exclusive) – like a Database

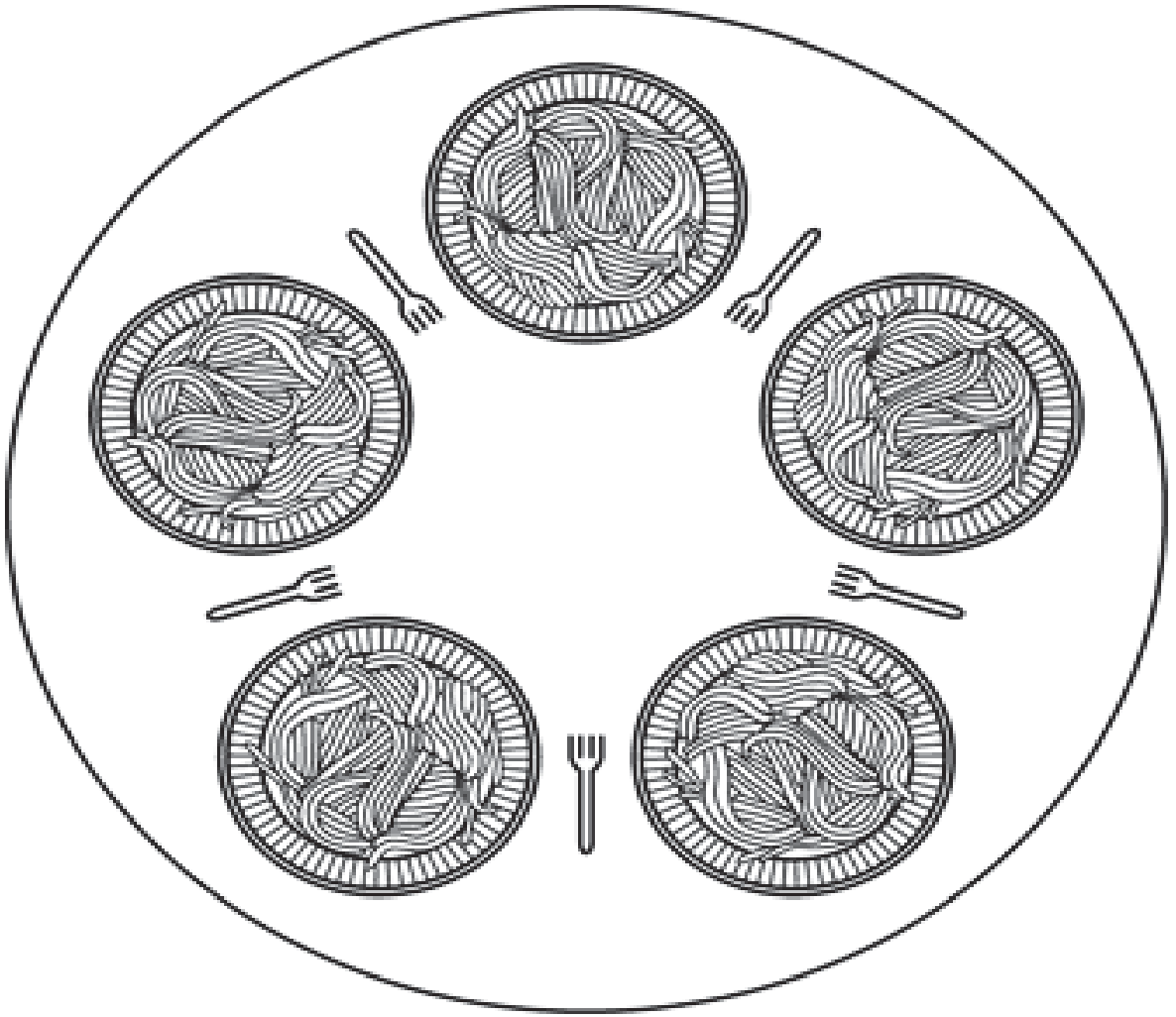
- The Sleeping Barber Problem

  A limited queue to one resource

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 */
  }
}
```

Solution using semaphores

```
void philosopher (int i)/* i: philosopher number, from 0 to N-1 */
{
  while (TRUE) {/* repeat forever */
  think();/* philosopher is thinking */
  take_forks(i);/* acquire two forks or block */
  eat();/* yum-yum, spaghetti */
  put_forks(i);/* put both forks back on table */
}

void test(i)/* i: philosopher number, from 0 to N-1 */
{
if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT]
!= EATING) {
state[i] = EATING;
up(&s[i]);
}
}
```

```
void take_forks(int i)
{
   down(&mutex); /* enter critical region */
   state[i] = HUNGRY;/* record fact that philosopher i is hungry */
   test(i);/* try to acquire 2 forks */
   up(&mutex); /* exit critical region */
   down(&s[i]); /* block if forks were not acquired */
}
void put_forks(i)/* i: philosopher number, from 0 to N-1 */
{
  down(&mutex); /* enter critical region */
  state[i] = THINKING;/* philosopher has finished eating */
  test(LEFT); /* see if left neighbor can now eat */
  test(RIGHT);/* see if right neighbor can now eat */
  up(&mutex); /* exit critical region */
}
```