Concurrency 1

Shared Memory

Catuscia Palamidessi
INRIA Futurs and LIX - Ecole Polytechnique

The other lecturers for this course:

Jean-Jacques Lévy (PPS)
James Leifer (INRIA - Rocq)
Eric Goubault (CEA)

http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2004/

Outline

1. Motivation
2. Overview of the course
3. Concurrency in Shared Memory: Effects and Issues
4. Critical Sections and Mutual Exclusion
   - Some attempts to implement a critical section
   - Some famous algorithms
   - Semaphores
   - The dining philosophers
   - Exercises

Motivation

Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation

Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

(S = speedup, b = sequential part, N processors)

Motivation
Why Concurrency?

- Programs for multi-processors
- Drivers for slow devices
- Human users are concurrent
- Distributed systems with multiple clients
- Reduce latency
- Increase efficiency, but Amdahl's law

$$S = \frac{N}{b * N + (1 - b)}$$

($S =$ speedup, $b =$ sequential part, N processors)

Overview of the course

09-28	CP	Shared memory: atomicity
10-05	CP/JJL	Shared memory: verification, report on Ariane 501
10-12	CP	CCS: syntax and transitions, coinduction
10-19	CP	CCS: weak and strong bisimulations, axiomatization
10-26	CP	CCS: examples, Hennessy-Milner logic
11-02	JL	π-calculus: syntax; reduction, transitions, strong bisimulation
11-09	JL	π-calculus: sum, abstractions, data structures, bisimulation proofs
11-16	JL	π-calculus: bisimulation “up to”, congruence, barbed bisimulation
11-23	Review	
11-30	MT exam	
12-07	JL	π-calculus: comparison between equivalences
12-14	JJL	Expressivity of the pi-calculus and its variants
12-21	vacation	
12-28	vacation	
01-04	JJL	Distributed pi-calculus
01-11	JJL	Problems with distributed implementation
01-18	EG	True concurrency versus interleaving semantics
01-25	EG	Event structures and Petri nets
02-01	EG	Application to the semantics of CCS
02-08	EG	Comparison of the expressiveness of different models
02-15	Review	
02-22	Final exam	

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

  $S = [x := 1;]$ and $T = [x := 2;]$

- After the execution of $S \mid\mid T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

  $$S = [x := 1;] \quad \text{and} \quad T = [x := 2;]$$

- After the execution of $S \,||\, T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

$$S = [x := 1;] \quad \text{and} \quad T = [x := 2;]$$

- After the execution of $S \,||\, T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

  $S = [x := 1;]$ and $T = [x := 2;]$

- After the execution of $S \mid\mid T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

  $S = [x := 1;]$ and $T = [x := 2;]$

- After the execution of $S \,||\, T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

  $S = [x := 1;]$ and $T = [x := 2;]$

- After the execution of $S \,||\, T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Non-determinism

- Note: we assume that the update of a variable is *atomic*
- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider two simple processes

$$S = [x := 1;] \quad \text{and} \quad T = [x := 2;]$$

- After the execution of $S \,||\, T$, we have $x \in \{1, 2\}$
- Conclusion:
  - Result is not unique.
  - Concurrent programs are not described by functions.

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \mid\mid x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \mid\mid \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \parallel x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \parallel \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \mid\mid x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \mid\mid \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \parallel x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \parallel \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \mid\mid x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \mid\mid \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked
- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \mid\mid x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \mid\mid \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Implicit Communication

- Let x be a global variable. Assume that at the beginning $x = 0$
- Consider the two processes

$$S = [x := x + 1; x := x + 1 \mid\mid x := 2 * x]$$
$$T = [x := x + 1; x := x + 1 \mid\mid \text{wait } (x = 1); x := 2 * x]$$

- After the execution of S, we have $x \in \{2, 3, 4\}$
- After the execution of T, we have $x \in \{3, 4\}$
- T may be blocked

- Conclusion: The parallel subcomponents of a program may interact via their shared variables

Input-output behavior

- Let x be a global variable.
- Consider the two processes

  $S = [x := 1]$ and $T = [x := 0; x := x + 1]$

- S and T are the same function on memory state.
- However, $S \mid\mid S$ and $T \mid\mid S$ are different "functions" on memory state.
- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

  $S = [x := 1]$ and $T = [x := 0; x := x + 1]$

- S and T are the same function on memory state.
- However, $S \mid\mid S$ and $T \mid\mid S$ are different "functions" on memory state.

- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

$$S = [x := 1] \quad \text{and} \quad T = [x := 0; x := x + 1]$$

- S and T are the same function on memory state.
- However, $S \mid\mid S$ and $T \mid\mid S$ are different "functions" on memory state.

- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

$$S = [x := 1] \quad \text{and} \quad T = [x := 0; x := x + 1]$$

- S and T are the same function on memory state.
- However, $S \,||\, S$ and $T \,||\, S$ are different "functions" on memory state.
- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

  $S = [x := 1]$ and $T = [x := 0; x := x + 1]$

- S and T are the same function on memory state.
- However, $S \,||\, S$ and $T \,||\, S$ are different “functions” on memory state.
- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

$$S = [x := 1] \quad \text{and} \quad T = [x := 0; x := x + 1]$$

- S and T are the same function on memory state.
- However, $S \mid\mid S$ and $T \mid\mid S$ are different "functions" on memory state.

- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq \mathit{Null} + 2^{\mathit{action} \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Input-output behavior

- Let x be a global variable.
- Consider the two processes

  $S = [x := 1]$ and $T = [x := 0; x := x + 1]$

- S and T are the same function on memory state.
- However, $S \,||\, S$ and $T \,||\, S$ are different "functions" on memory state.

- A process is an *atomic action*, followed by a process:

$$\mathcal{P} \simeq Null + 2^{action \times \mathcal{P}}$$

- Part of the concurrency course aims at giving sense to this equation.

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Atomicity

- Let x be a global variable. Assume that at beginning $x = 0$
- Consider the process $S = [x := x + 1 \mid\mid x := x + 1]$
- After the execution of S we have $x = 2$.

- However $[x := x + 1]$ may be compiled into $[A := x + 1; x := A]$
- So, S may behave as $[A := x + 1; x := A] \mid\mid [B := x + 1; x := B]$, which, after execution, gives $x \in \{1, 2\}$.

- To avoid such effect, $[x := x + 1]$ has to be *atomic*
- Atomic statements, aka *critical sections* can be implemented via *mutual exclusion*

Outline

The problem

- Let $P_0 = [\cdots ; C_0; \cdots]$ and $P_1 = [\cdots ; C_1; \cdots]$
- We intent C_0 and C_1 to be critical sections, i.e. they should not be executed simultaneously.

Attempt n.1

- Use a variable *turn*. At beginning, $turn = 0$.

P0

```
...;
  while turn != 0  do ;
  C0;
  turn := 1;
  ...
```

P1

```
...;
  while turn != 1  do ;
  C1;
  turn := 0;
  ...
```

- However the method is unfair, because P_0 is privileged. Worse yet, until P_0 executes its critical section, P_1 is blocked.

Attempt n.1

- Use a variable *turn*. At beginning, $turn = 0$.

P0

```
...;
   while turn != 0  do ;
   C0;
   turn := 1;
   ...
```

P1

```
...;
   while turn != 1  do ;
   C1;
   turn := 0;
   ...
```

- However the method is unfair, because P_0 is privileged. Worse yet, until P_0 executes its critical section, P_1 is blocked.

Attempt n.2

- Use two boolean variables a_0, a_1.
  At beginning, $a_0 = a_1 =$ false .

P0

```
...;
while a1 do ;
a0 := true ;
C0;
a0 := false ;
...
```

P1

```
...;
while a0 do ;
a1 := true ;
C1;
a1 := false ;
...
```

- Incorrect. It does not ensure mutual exclusion.

Attempt n.2

- Use two boolean variables a_0, a_1.
  At beginning, $a_0 = a_1 =$ false .

P0

```
...;
while a1 do ;
a0 := true ;
C0;
a0 := false ;
...
```

P1

```
...;
while a0 do ;
a1 := true ;
C1;
a1 := false ;
...
```

- Incorrect. It does not ensure mutual exclusion.

Attempt n.3

- Use two boolean variables a_0, a_1.
  At beginning, $a_0 = a_1 =$ false .

P0

```
...;
a0 := true ;
while a1 do ;
a0 := true ;
C0;
a0 :=false ;
...
```

P1

```
...;
a1 := true ;
while a0 do ;
a1 := true ;
C1;
a1 := false ;
...
```

- We may get a deadlock. Both P_0 and P_1 may block.

Attempt n.3

- Use two boolean variables a_0, a_1.
  At beginning, $a_0 = a_1 =$ false .

P0

```
...;
a0 := true ;
while a1  do ;
a0 := true ;
C0;
a0 :=false ;
...
```

P1

```
...;
a1 := true ;
while a0  do ;
a1 := true ;
C1;
a1 := false ;
...
```

- We may get a deadlock. Both P_0 and P_1 may block.

Outline

Dekker's Algorithm (early Sixties)

- The first correct mutual exclusion algorithm
- Use both the variable *turn* and the boolean variables a_0 and a_1. At beginning, $a_0 = a_1 =$ false , $turn \in \{0, 1\}$

P0

```
...;
a0 := true ;
while a1  do
    if turn != 0  begin
        a0 := false ;
        while turn != 0  do ;
        a0 := true ;
        end ;
C0;
turn := 1; a0 := false ;
...
```

P1

```
...;
a1 := true ;
while a0  do
    if turn != 1  begin
        a1 := false ;
        while turn != 1  do ;
        a1 := true ;
        end ;
C1;
turn := 0; a1 := false ;
...
```

- A variant of Dekker's algorithm for the case of *n* processes was presented by Dijkstra (CACM 1965).

Dekker's Algorithm (early Sixties)

- The first correct mutual exclusion algorithm
- Use both the variable *turn* and the boolean variables a_0 and a_1. At beginning, $a_0 = a_1 =$ false , $turn \in \{0, 1\}$

P0

```
...;
a0 := true ;
while a1  do
    if turn != 0  begin
        a0 := false ;
        while turn != 0  do ;
        a0 := true ;
        end ;
C0;
turn := 1; a0 := false ;
...
```

P1

```
...;
a1 := true ;
while a0  do
    if turn != 1  begin
        a1 := false ;
        while turn != 1  do ;
        a1 := true ;
        end ;
C1;
turn := 0; a1 := false ;
...
```

- A variant of Dekker's algorithm for the case of *n* processes was presented by Dijkstra (CACM 1965).

Peterson's Algorithm (IPL 1981)

- The simplest and most compact mutual exclusion algorithm in literature
- Use both the variable *turn* and the boolean variables a_0 and a_1. At beginning, $a_0 = a_1 =$ false , $turn \in \{0, 1\}$

P0

```
...;
a0 := true ;
turn := 1;
while a1  and turn != 0  do ;
C0;
a0 := false ;
...
```

P1

```
...;
a1 := true ;
turn := 0;
while a0  and turn != 1  do ;
C1;
a1 := false ;
...
```

Correctness of Peterson's Algorithm (1/2)

- To show the correctness it is convenient to add two variables, pc_0, pc_1, which represent a sort of program counters for P_0 and P_1.
  At beginning $pc_0 = pc_1 = 1$

P0

...;
$\{\neg a_0 \wedge pc_0 \neq 2\}$
a0 := true ; pc0 := 2;
$\{a_0 \wedge pc_0 = 2\}$
turn := 1; pc0 := 1;
$\{a_0 \wedge pc_0 \neq 2\}$
while a1 and turn != 0 do ;
$\{a0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2)\}$
C0;
a0 := false ;
$\{\neg a_0 \wedge pc_0 \neq 2\}$
...

P1

...;
$\{\neg a_1 \wedge pc_1 \neq 2\}$
a1 := true ; pc1 := 2;
$\{a_1 \wedge pc_1 = 2\}$
turn := 0; pc1 := 1;
$\{a_1 \wedge pc_1 \neq 2\}$
while a0 and turn != 1 do ;
$\{a1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2)\}$
C1;
a1 := false ;
$\{\neg a_1 \wedge pc_1 \neq 2\}$
...

Correctness of Peterson's Algorithm (2/2)

We can prove the correctness by contradiction. If both programs were in their critical section, then the formulas $\{a0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2)\}$ and $\{a1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2)\}$ should be true at the same time, but:

$$\begin{array}{ll} & a_0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2) \\ \wedge & a_1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2) \\ \equiv & turn = 0 \wedge turn = 1 \end{array}$$

Contradiction!

Correctness of Peterson's Algorithm (2/2)

We can prove the correctness by contradiction. If both programs were in their critical section, then the formulas $\{a0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2)\}$ and $\{a1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2)\}$ should be true at the same time, but:

$$\begin{array}{ll} & a_0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2) \\ \wedge & a_1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2) \\ \\ \equiv & turn = 0 \wedge turn = 1 \end{array}$$

Contradiction!

Correctness of Peterson's Algorithm (2/2)

We can prove the correctness by contradiction. If both programs were in their critical section, then the formulas $\{a0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2)\}$ and $\{a1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2)\}$ should be true at the same time, but:

$$\begin{array}{ll} & a_0 \wedge pc_0 \neq 2 \wedge (\neg a_1 \vee turn = 0 \vee pc_1 = 2) \\ \wedge & a_1 \wedge pc_1 \neq 2 \wedge (\neg a_0 \vee turn = 1 \vee pc_0 = 2) \\ \\ \equiv & turn = 0 \wedge turn = 1 \end{array}$$

Contradiction!

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to *N* processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, . . .
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to N processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to *N* processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to *N* processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to *N* processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, . . .
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to N processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to N processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, . . .
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Synchronization in Concurrent/Distributed algorithms

- Dekker's algorithm (early sixties). Quite complex.
- Peterson is simpler and can be generalized to *N* processes more easily
- Both algorithms by Dekker and Peterson use busy waiting
- Fairness relies on fair scheduling
- Many other algorithms for mutual exclusion have been proposed in literature. Particularly by Lamport: barber, baker, ...
- Proofs ? By model checking ? With assertions ? In temporal logic (eg Lamport's TLA)?

Need for higher constructs in concurrent programming.

Outline

Semaphores

A generalized semaphore s is an integer variable with 2 operations

- $acquire(s)$: If $s > 0$ then $s := s - 1$, otherwise suspend on s. (atomically)
- $release(s)$: If some process is suspended on s, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$$[\cdots; acquire(s);\ C_0;\ release(s); \cdots] \ || \ [\cdots; acquire(s);\ C_1;\ release(s); \cdots]$$

Question Consider another definition for semaphore:
$acquire(s)$: If $s > 0$ then $s := s - 1$. Otherwise restart.
$release(s)$: Do $s := s + 1$.
Are these definitions equivalent?

Semaphores

A generalized semaphore s is an integer variable with 2 operations

- $acquire(s)$: If $s > 0$ then $s := s - 1$, otherwise suspend on s. (atomically)
- $release(s)$: If some process is suspended on s, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$$[\cdots ; acquire(s);\ C_0;\ release(s); \cdots] \ || \ [\cdots ; acquire(s);\ C_1;\ release(s); \cdots]$$

Question Consider another definition for semaphore:
$acquire(s)$: If $s > 0$ then $s := s - 1$. Otherwise restart.
$release(s)$: Do $s := s + 1$.
Are these definitions equivalent?

Semaphores

A generalized semaphore s is an integer variable with 2 operations

- *acquire*(s): If $s > 0$ then $s := s - 1$, otherwise suspend on s. (atomically)
- *release*(s): If some process is suspended on s, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$$[\cdots ; acquire(s);\ C_0;\ release(s); \cdots] \ || \ [\cdots ; acquire(s);\ C_1;\ release(s); \cdots]$$

Question Consider another definition for semaphore:
acquire(s): If $s > 0$ then $s := s - 1$. Otherwise restart.
release(s): Do $s := s + 1$.
Are these definitions equivalent?

Semaphores

A generalized semaphore s is an integer variable with 2 operations

- $acquire(s)$: If $s > 0$ then $s := s - 1$, otherwise suspend on s. (atomically)
- $release(s)$: If some process is suspended on s, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$$[\cdots; acquire(s);\ C_0;\ release(s); \cdots]\ ||\ [\cdots; acquire(s);\ C_1;\ release(s); \cdots]$$

Question Consider another definition for semaphore:
$acquire(s)$: If $s > 0$ then $s := s - 1$. Otherwise restart.
$release(s)$: Do $s := s + 1$.
Are these definitions equivalent?

Semaphores

A generalized semaphore s is an integer variable with 2 operations

- $acquire(s)$: If $s > 0$ then $s := s - 1$, otherwise suspend on s. (atomically)
- $release(s)$: If some process is suspended on s, wake it up, otherwise $s := s + 1$. (atomically)

Now mutual exclusion is easy: At beginning, $s = 1$. Then

$$[\cdots; acquire(s);\ C_0;\ release(s); \cdots] \ || \ [\cdots; acquire(s);\ C_1;\ release(s); \cdots]$$

Question Consider another definition for semaphore:
$acquire(s)$: If $s > 0$ then $s := s - 1$. Otherwise restart.
$release(s)$: Do $s := s + 1$.
Are these definitions equivalent?

Outline

1. Motivation
2. Overview of the course
3. Concurrency in Shared Memory: Effects and Issues
4. Critical Sections and Mutual Exclusion
   - Some attempts to implement a critical section
   - Some famous algorithms
   - Semaphores
   - The dining philosophers
   - Exercises

The dining philosophers

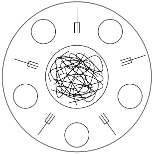
- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.
- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

The dining philosophers

- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.
- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

The dining philosophers

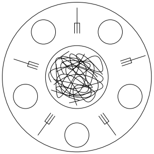
- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.



- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

The dining philosophers

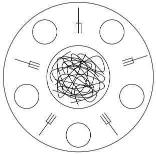
- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.



- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

The dining philosophers

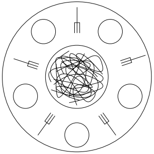
- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.



- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

The dining philosophers

- Problem proposed by Dijkstra for testing concurrency primitives
- 5 philosophers spend their time around a table thinking or eating spaghetti. In order to eat, each philosopher needs two forks. However, there are only 5 forks on the table.



- Desiderata
  - if one philosopher gets hungry, some philosopher will eventually eat (*progress*)
  - if one philosopher gets hungry, he will eventually eat (*starvation-freedom*)

Outline

1. Motivation
2. Overview of the course
3. Concurrency in Shared Memory: Effects and Issues
4. Critical Sections and Mutual Exclusion
   - Some attempts to implement a critical section
   - Some famous algorithms
   - Semaphores
   - The dining philosophers
   - Exercises

Exercises

- (Difficult) Generalize Dekker's algorithm to the case of n processes
- Generalize Petersons's algorithm to the case of n processes
- Implement the Semaphore in Java
- Write a program for the dining philosophers which ensure progress
- Discuss how to modify the solution so to ensure starvation-freedom
- Problem: A certain file is shared by some Reader and some Writer processes: we want that only one writer can write on the file at a time, while the readers are allowed to do it concurrently. Write the code for the Reader and the Writer.