

ID1217 Concurrent Programming
Lecture 2

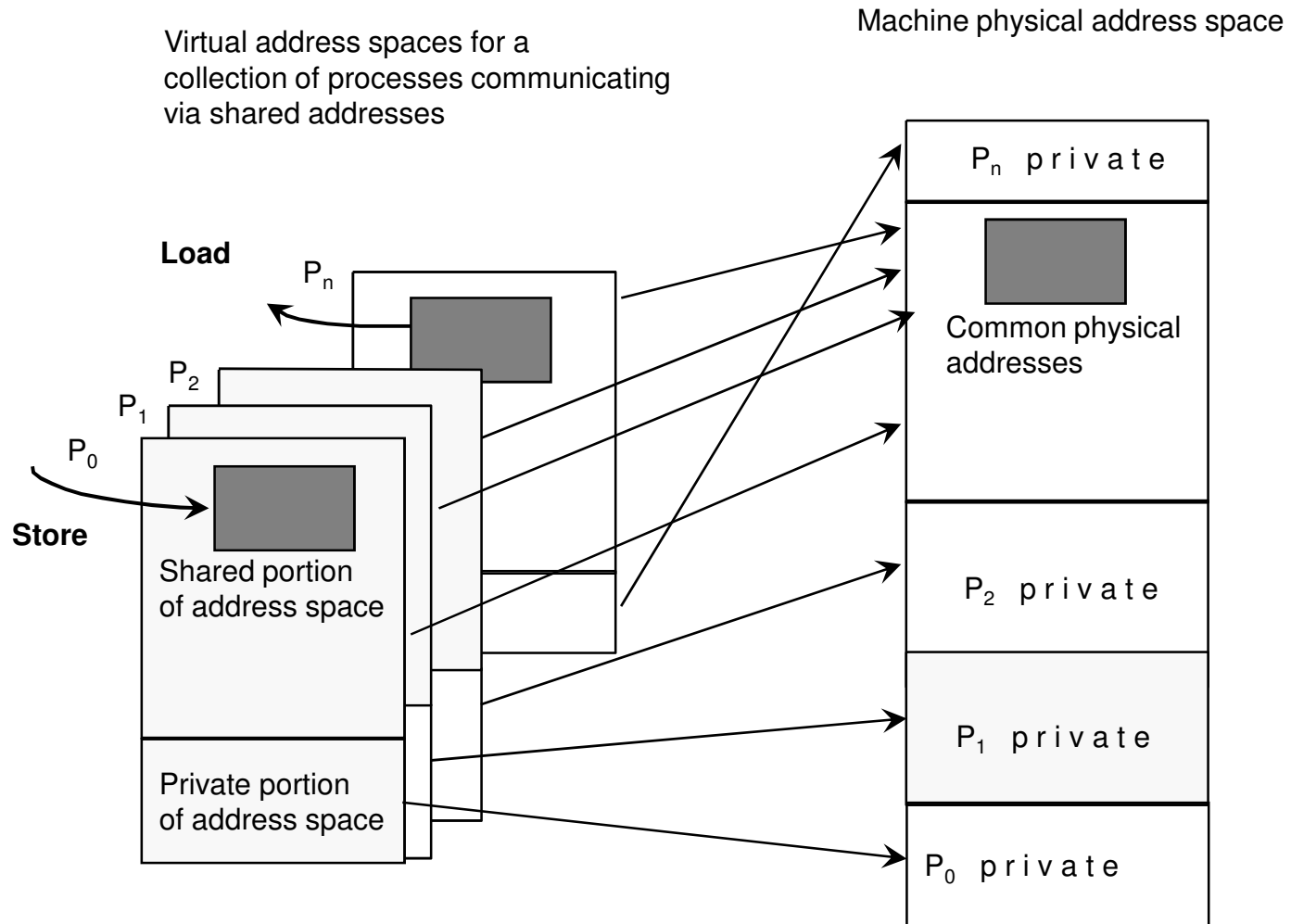
Shared Memory Programming:
Processes and Synchronization

Vladimir Vlassov
KTH/ICT/ECS
VT 2008

Outline (Section 2.1-2.5)

- Processes
 - Atomic actions
 - Process histories.
 - Interleaving semantics and nondeterminism of concurrent execution.
- Synchronization
 - Atomicity.
 - The await statement.
 - At-Most-Once property.
- Examples of parallelization and synchronization
 - Finding patterns in a file.
 - Find the maximum element in an array.

Shared Address Space Model



Process State. Actions. Process History

- ***Process (thread)*** is an abstract entity that performs tasks assigned to processes
- ***Process state*** is formed of values of variables at a point in time.
- Each process executes a sequence of statements.
- Each statement consists of one or more ***atomic (indivisible) actions*** which transform one state into another
 - Some actions allow processes to communicate with each other
- Sequence of states makes up a process history
- The ***process history*** is a trace of ONE execution.

$$P: Q_0 \xrightarrow{a_1} Q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} Q_m$$

Atomic Actions

- ***Atomic action*** – indivisible sequence of state transitions made atomically
- Fine-grained atomic actions
 - Machine instructions (read, write, swap, etc.) – atomicity is guaranteed by HW
- Coarse-grained atomic actions
 - A sequence of fine-grained atomic actions executed indivisibly (atomically)
 - Internal state transitions are not visible “outside”
 - For example, a critical section of a code
- Notation: **< statements >** – a list of statements to be executed atomically.

Interleaving Semantics of Concurrent Execution

- The concurrent execution of several processes can be viewed as the interleaving of histories of the processes
 - i.e., the interleaving of sequences of atomic actions of different processes.
- Individual histories:
 - Process 1: $\mathbf{s}_0 \rightarrow \mathbf{s}_1 \rightarrow \mathbf{s}_2 \rightarrow \dots \rightarrow \mathbf{s}_n$
 - Process 2: $\mathbf{p}_0 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \dots \rightarrow \mathbf{p}_n$
 - Process 3: $\mathbf{q}_0 \rightarrow \mathbf{q}_1 \rightarrow \mathbf{q}_2 \rightarrow \dots \rightarrow \mathbf{q}_n$
- Interleaved concurrent histories:
 - Trace 1: $\mathbf{s}_0 \rightarrow \mathbf{p}_0 \rightarrow \mathbf{s}_1 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \mathbf{q}_0 \rightarrow \mathbf{s}_2 \rightarrow \mathbf{q}_1 \rightarrow \dots$
 - Trace 2: $\mathbf{p}_0 \rightarrow \mathbf{s}_0 \rightarrow \mathbf{q}_0 \rightarrow \mathbf{q}_1 \rightarrow \mathbf{s}_1 \rightarrow \mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \mathbf{q}_2 \rightarrow \dots$

Nondeterminism of Concurrent Execution

- The behavior of a concurrent program is not reproducible because different interleavings (histories) can be observed on different concurrent executions.
 - Each process executes at its own rate
 - Multiprogramming causes context switching
- A concurrent program of n processes each of m atomic actions can produce $(n \times m)! / (m!)^n$ different histories.
 - For example, $n = 3$, $m = 2$ give 90 different histories.
- Impossible to show the correctness of a program by testing (“run the program and see what happens”).

Example of Nondeterminism

- A program:

```
int y = 0; z = 0;  
co x = y + z; || y = 1; z = 2; oc
```

- Trace 1:

```
x = y{0} + z{0}; y = 1; z = 2;      {x == 0}
```

- Trace 2:

```
y = 1; x = y{1} + z{0}; z = 2;      {x == 1}
```

- Trace 3:

```
y := 1; z := 2; x := y{1} + z{2};    {x == 3}
```

- Trace 4:

```
load y{0} to R1; y := 1; z := 2;  
add z{2} to R1{0}; store R1 to x; {x == 2}
```


Synchronization

- Synchronization is a mechanism to delay a process until it may proceed
 - Allows reducing the entire set of possible histories to those which are desirable (correct).
 - To preserve (true) dependences between processes
 - To avoid race conditions if any
- Two kinds of synchronization:
 - *Mutual exclusion*
 - guarantees that only one process executes its critical section at a time
 - *Condition synchronization*
 - delays a process until a certain condition is true.

Specifying Synchronization: The Await Statement

< **await (B) S; >**

- Wait for **B** to be true, then execute **S** atomically
- Combines condition synchronization with mutual exclusion
 - Executed as an atomic action
 - B is guaranteed to hold when S begins
 - S is guaranteed to terminate

< **await (B) ; >**

- Wait for a condition **B** to be true.
- Used to express condition synchronization

< **S; >**

- Execute a list of statements **S** atomically (indivisibly).
- Can be used to express mutual exclusion

Assumptions on Machine Architecture

- RISC architecture
 - Values are manipulated by loading them into registers, operating on them, and storing them back to memory
- Atomic memory operations
 - Values of the basic types are stored in memory elements that are read and written as atomic actions
- Process context
 - Each process has its own set of registers and stack
- All temporaries are stored in private memory
 - Any intermediate results are stored in registers or in process' private memory (on stack)

The At-Most-Once Property

- Unnecessary explicit coarse-grained atomicity causes unnecessary overhead and may reduce parallelism.
- A *critical reference* is a reference to a variable which is (can be) changed by another process.
- A statement $\mathbf{x} = \mathbf{e}$ appears to be executed atomically when it satisfies the *At-Most-Once (AMO) property*, i.e. if
 - either (a) \mathbf{e} contains at most one critical reference and \mathbf{x} is not referenced by another process,
 - or (b) \mathbf{e} contains no critical references and \mathbf{x} may be read by another processes.

Implication of the AMO Property

< S; > ≡ S;

- as long as **S** contains at most one critical reference seeing by other processes,
- In this case, one cannot tell the difference, so **S** will appear to execute atomically w.r.t. to other processes.

< await (B); > ≡ while (not B) continue;

- if **B** contains (reads) at most one critical reference

Parallelizing a Sequential Program

- How? Identify independent and dependent tasks, assign tasks to processes, identify shared variables, synchronize processes.
- Data accessed by a process:
 - Read set (rs) – variables that are only read
 - Write set (ws) – variables that are written (and possibly read)
- ***Independence***: processes are independent iff the write set of one proc is disjoint from the read and write sets of another proc:

$$ws_1 \cap (rs_2 \cup ws_2) = \emptyset \text{ AND } ws_2 \cap (rs_1 \cup ws_1) = \emptyset$$

Dependencies Between Atomic Actions w.r.t. to a Shared Variable

- Assume two atomic actions access the same variable
- Dependences:
 - *True dependence*: read after write, e.g. $x := 1; a := x;$
 - *Anti dependence*: write after read, e.g. $a := x; x := 1;$
 - *Output dependence*: write after write, e.g. $x := 1; x := 2;$
 - *Input dependence*: read after read, e.g. $a := x; b := x;$
- If the actions are executed by different processes, all dependencies must be preserved by synchronization
 - Input dependences do not require synchronization
 - Some anti and output dependencies can be *false dependencies* caused by reuse of memory – To avoid, rename (disjoin) variables
 - An output dependence does not need to be preserved (ordered) if the order of writes is not important, e.g. OR-parallelism

Example of Parallelization and Synchronization: Finding Patterns in a File

- Find all instances of a pattern in a file: **grep pattern filename**

- Sequential program:

```
string line;  
open the file; read a line of input from the file into line;  
while (!EOF) { # EOF is end of file  
    look for pattern in line;  
    if (pattern is in line) print line;  
    read next line from the file into line;  
}
```

- Parallelization:

- Tasks: read a line, compare with the pattern, print if match
- Apply Producer-Consumer paradigm:
 - P reads a line from the file and stores it to a shared buffer
 - C gets a line from the buffer, tests it against pattern, prints if match.

Example (cont'd): How to Synchronize

- True/anti dependence between P and C via the shared buffer.
- Let the buffer provide place for only one string
- Synchronization:
 - Mutual exclusion: The buffer is accessed by one process at a time
 - Condition synchronization
 - To put a line, Producer waits for the buffer to be empty
 - To get a line, Consumer waits for a buffer to become full
- Let's use two counters:
 - **p** – number of produced lines (stored to the buffer by Producer)
 - **c** – number of consumed (fetched from the buffer by Consumer)
- Synchronization requirement expressed as a predicate:

$$PC: c \leq p \leq c + 1$$

Example (cont'd): Parallel Program

```
# shared variables:
string buf; /* buffer (input line) */
bool done = false; /* termination */
int p = 0, c = 0; /* counters */
Process Producer { /* reads new lines */
    string line;
    open the file;
    while (true) {
        read a line of input from the file into line;
        if (EOF) { done = true; break; }
        /* wait for buffer to be empty */
        < await (p == c); >
        buf = line;
        p++; /* signal that buffer is full */
    }
};

Process Consumer { /* finds patterns */
    string line;
    while (true) {
        /* wait for buffer to be full */
        /* or done to be true */
        < await ((p > c) || done); >
        if (p > c) {
            line = buf;
            c++; /* signal that buf is empty*/
            look for pattern in line;
            if (pattern is in line) write line;
        }
        if (done) break;
    }
}
```

Example of Synchronization: Find the Maximum

- Given array $a[1:n]$ of positive integers. Find the maximum value m
 $(\forall j: 1 \leq j \leq n: m \geq a[j]) \wedge (\exists j: 1 \leq j \leq n: m = a[j])$
 - When program terminates, m is at least as large as every element of a
 - m is one of elements of a
- Sequential program:

```
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m) m = a[i];
```
- How to parallelize? – Examining all elements in parallel
- Parallel program without synchronization:

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m) m = a[i]; oc
```

 - Program is incorrect because of the race condition
 - Synchronization is needed

Example (cont'd): Concurrent Program

- First attempt to synchronize: execute cond. updates atomically

```
int m = 0;  
co [i = 0 to n-1  
    < if (a[i] > m) m = a[i]; > oc
```

- Not efficient (overconstrained): all the updates are serialized just like in the sequential program but executed in an arbitrary order.
- Observations:
 - Read and test can be executed in parallel for each **i**
 - Updates of **m** require atomicity (mutual exclusion) for serialization
 - Can use second test in critical section to avoid races

Example (cont'd): Solution for Concurrent Program

- Read and test in parallel without mutual exclusion. Those who passed the test, execute conditional update atomically (with mutual exclusion)

```
int m = 0;
co [i = 0 to n-1]
  if (a[i] > m)
    < if (a[i] > m) m = a[i]; > oc
```

- Lessons learned
 - Synchronization is required whenever processes read and write shared variables (to preserve true, anti, and output dependences)
 - Atomicity helps to provide mutual exclusion
 - Test followed by an atomic test-and-update is a useful combination for conditional updates
 - Helps to avoid races among concurrent conditional updates that depend on the same condition

Problem 1: Non-determinism.

Possible final values

- Consider the following fragment of a program outline:

```
var x = 1, y = 2, z = 3;  
co x = x + 1;  
  || y = y + 2;  
  || z = x + y;  
  || <await (x > 1) x = 0; y = 0; z = 0 >  
oc
```

- Assume that atomic actions in the first three arms of the **co** statement are reading and writing individual variables.
- Does the co-statement terminate? What are the possible final values of x, y and z? Explain.

Solution to Problem 1

```
var x = 1, y = 2, z = 3;  
co x = x + 1  
  || y = y + 2  
  || z = x + y  
  || <await (x > 1) x = 0; y = 0; z = 0 >  
oc
```

- The possible final values of x, y and z are

x == {0}

y == {0,2,4}

z == {0,1,2,3,4,5,6}

Problem 2: Await versus If

- Consider the following program fragment that spawns three threads in the co-statement:

```
int x = 0;
co <await (x != 0) x = x - 2;>
  || <await (x != 0) x = x - 3;>
  || <await (x == 0) x = x + 5;>
oc
```

- Does the program terminate? If so, what are the possible final values of **x**? If not, why not?
- Suppose the **await** statements are changed to **if** statements – namely, that **await** is replaced by **if** and the angle brackets are deleted. Now the program is sure to terminate. What are the possible final values of **x**?

Solution to Problem 2

(a) Yes, it does terminate. The final value of x is zero. The last statement executes first, then the other two in either order.

(b)

```
int x = 0;
co if (x != 0) x = x - 2; // S1
|| if (x != 0) x = x - 3; // S2
|| if (x == 0) x = x + 5; // S3
oc
```

Possible histories and corresponding final values are the following:

S1 and S2 tests fail then S3	$x == 5$
S2 test fails; S3; S1	$x == 3$
S1 test fails; S3; S2	$x == 2$
S3; S1 S2	$x == \{0, 2, 3\}$

Problem 3: Intersection of two arrays

- Given integer arrays $a[1:m]$ and $b[1:n]$, assume that each array is sorted in ascending order.
 - (a) Develop an outline of a sequential program to compute the number of values that appear in both a and b , i.e. the intersection of a and b .
 - (b) Develop an outline of a parallel program based on your sequential program.
Use the `co` statement for concurrent execution and the `await` statement for synchronization (if needed).

Possible Solution to Problem 3

- (a) We can do a linear search in the arrays to compare the elements and find the common ones.

– The following sequential solution is inefficient:

```
int i = 0, j = 0, count = 0;
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        if (a[i] == b[j]) count++;
```

– The outline of a more efficient sequential program:

```
int i = 0, j = 0, count = 0;
while (i < m && j < n) {
    if (a[i] < b[j]) i++;
    else if (a[i] > b[j]) j++;
    else { count++; i++; j++; }
}
```

- (b) The outline of a parallel program:

```
int i = 0, j = 0, count = 0, A, B;
while (i < m && j < n) {
    A = b[i]; B = b[j];
    co if (A < B) i++;
    || if (A > B) j++;
    || if (A == B) { count++; i++; j++; }
    oc
}
```

- There is no need of any synchronization because the conditions are disjoint. Only one of them in every iteration will increment the shared variable(s).

Properties of a Program

- A property of a program is an attribute of ALL histories of a program, e.g., correctness.
- Two kinds of properties:
 - *Safety properties*
 - A safety property is one in which the program never enters a “bad” state
 - *Liveness properties*
 - A liveness property is one in which the program eventually enters a “good” (desirable) state

Examples of Safety and Liveness Properties

- Safety properties:
 - Partial correctness
 - The final state is correct, assuming that the program terminates
 - Mutual exclusion of critical sections
 - “Bad” when more than one proc. are in critical sessions
 - Absence of deadlocks
 - Deadlock: Wait for a condition that never occurs
- Liveness properties:
 - Termination
 - Every history (trace) is finite.
 - Eventually enter a critical section (if any)
- ***Total correctness*** combines termination and partial correctness
 - A program always terminates with a correct answer

Proving Properties

Three main approaches

1. Testing or debugging

- Run and see what happens
- Limited to considered cases

2. Operational reasoning

- “Exhaustive case analysis”
- Considers enormous number of histories: $n \cdot m! / (m!)^n$
- Helps in development

3. *Assertional reasoning*

- Based on axiomatic semantics: axioms, inference rules, assertions
- Work is proportional to the number of atomic actions