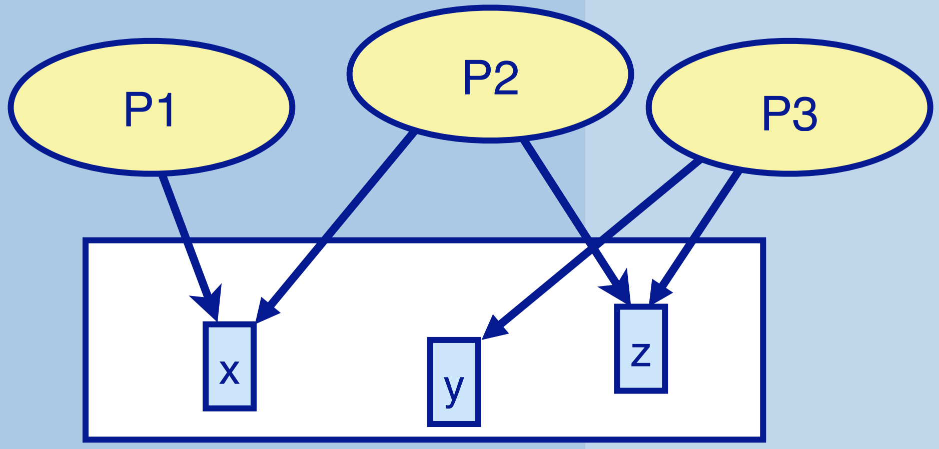
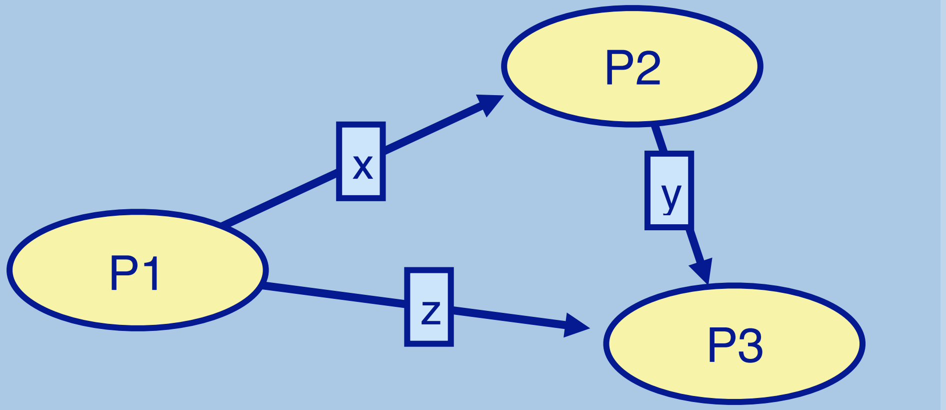
CP — Concurrency: State Models and Design Patterns

1. Introduction

Oscar Nierstrasz





Concurrent Programming

Lecturer	Prof. Oscar Nierstrasz
Assistants	Pascal Gadient, Mohammadreza Hazhirpasand
Lectures	Wednesday @ 10h15-12h00
Exercises	Wednesday @ 12h00-13h00
WWW	http://scg.unibe.ch/teaching/cp/

This is a note (a hidden slide). You will find some of these scattered around the PDF versions of the slides.

Roadmap

- Course Overview
- Concurrency and Parallelism
- Challenges: Safety and Liveness
- Expressing Concurrency
  - Process Creation
  - Communication and Synchronization

Goals of this course

> Introduce *basic concepts of concurrency*
>
> —safety, liveness, fairness

> Present tools for *reasoning* about concurrency
>
> —LTS, Petri nets

> Learn the *best practice* programming techniques
>
> —idioms and patterns

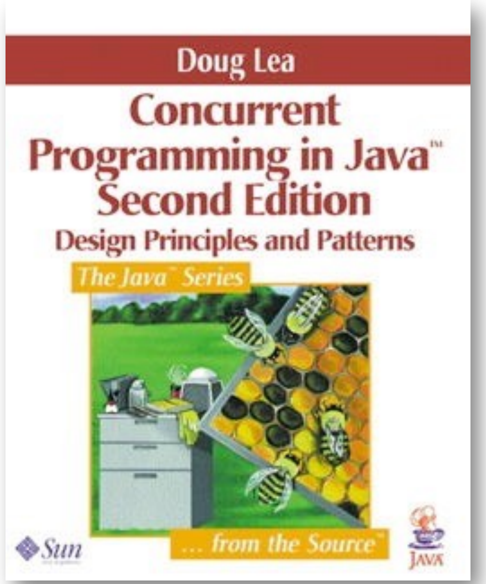
> Get *experience* with the techniques
>
> —lab sessions

Schedule

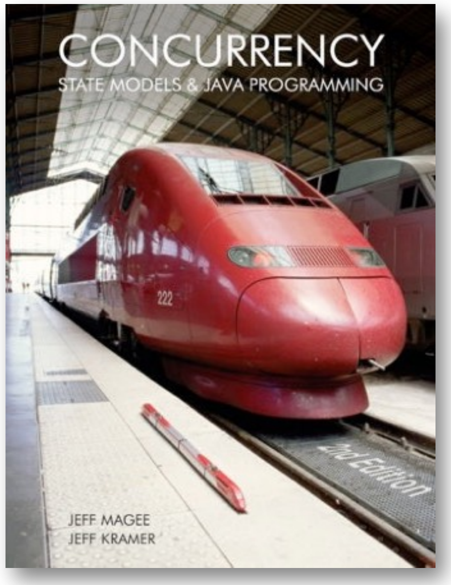
1	22-Sep-21	Introduction
2	29-Sep-21	Java and Concurrency
3	6-Oct-21	Safety and Synchronization
4	13-Oct-21	Safety Patterns + Transactional Memory
5	20-Oct-21	Liveness and Guarded Methods
6	27-Oct-21	Lab session
7	3-Nov-21	Liveness and Asynchrony
8	10-Nov-21	Condition Objects
9	17-Nov-21	Fairness and Optimism
10	24-Nov-21	Lab session
11	1-Dec-21	Petri Nets
12	8-Dec-21	Architectural Styles for Concurrency
13	15-Dec-21	TBA
14	22-Dec-21	Exam

Texts

Doug Lea: **Concurrent Programming in Java: Design Principles and Patterns**, Addison-Wesley, 1996



Jeff Magee, Jeff Kramer, **Concurrency: State Models & Java Programs**, Wiley, 1999

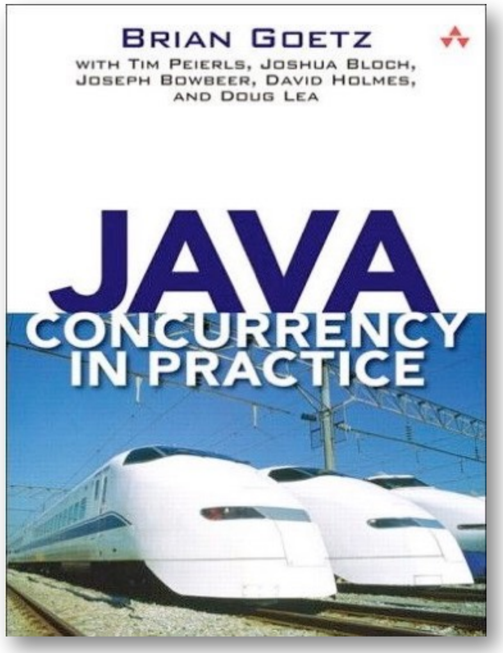


Concurrency raises complex issues for programming software systems. In this course we take a pragmatic approach to balancing theory and practice. The book by Magee and Kramer provides us with *theoretical foundations* needed to understand the problems posed by concurrency and the *formal tools* we need to reason about it. The book by Lea, on the other hand, focuses on *common design patterns* used to manage concurrency and reduce its complexity.

Further reading

Brian Goetz et al,

Java Concurrency in Practice,
Addison Wesley, 2006.


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE

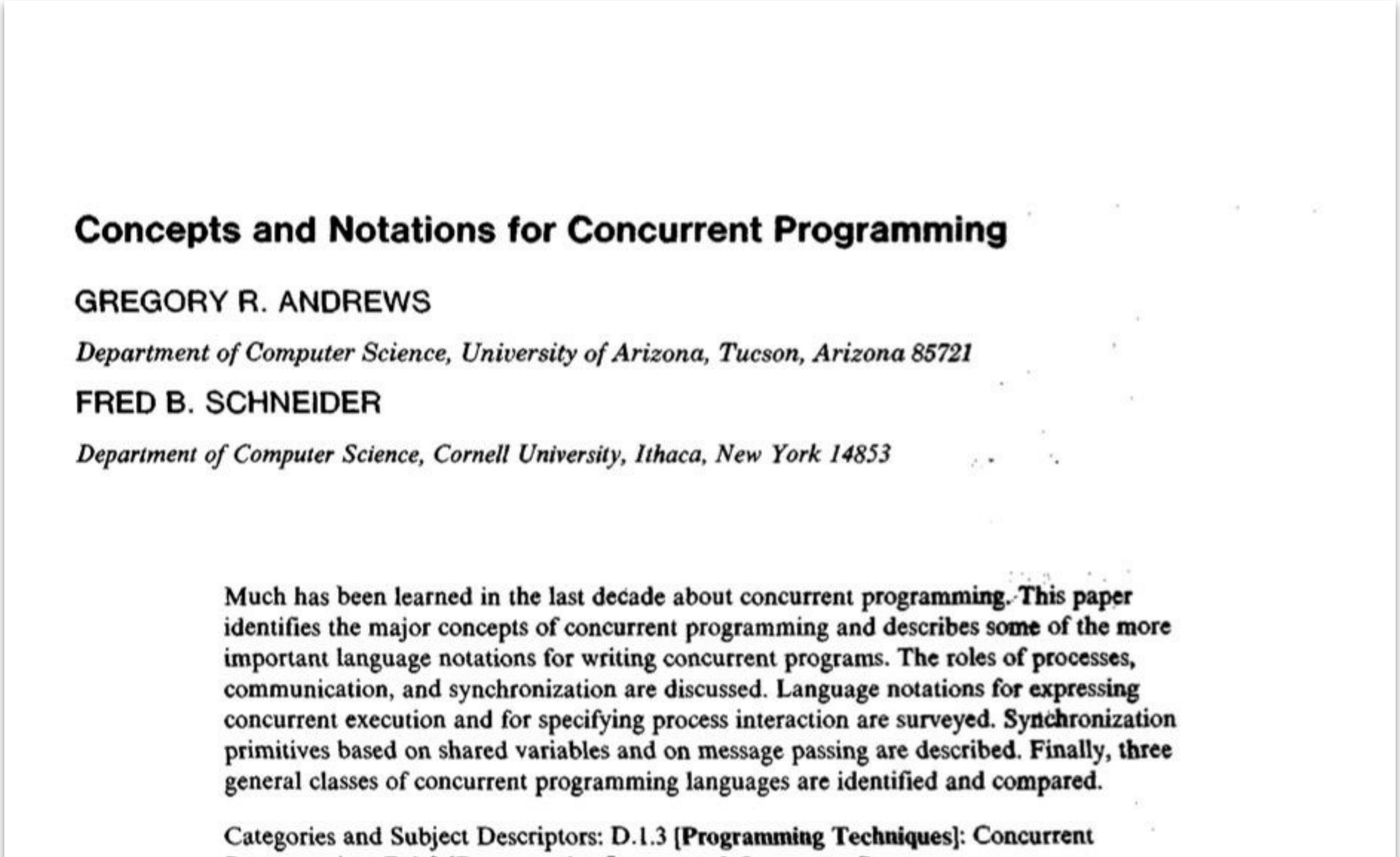
The book by Goetz provides a complement to the other two books, by presenting some of the more modern concurrency features and libraries offered by the Java platform.

Roadmap

- Course Overview
- **Concurrency and Parallelism**
- Challenges: Safety and Liveness
- Expressing Concurrency
  - Process Creation
  - Communication and Synchronization

A Survey of Concurrent Programming Concepts and Notations

Gregory R. Andrews and Fred B. Schneider.
Concepts and Notations for Concurrent Programming.
In ACM Computing Surveys 15(1) p. 3—43, March 1983.



Concepts and Notations for Concurrent Programming

GREGORY R. ANDREWS

Department of Computer Science, University of Arizona, Tucson, Arizona 85721

FRED B. SCHNEIDER

Department of Computer Science, Cornell University, Ithaca, New York 14853

Much has been learned in the last decade about concurrent programming. This paper identifies the major concepts of concurrent programming and describes some of the more important language notations for writing concurrent programs. The roles of processes, communication, and synchronization are discussed. Language notations for expressing concurrent execution and for specifying process interaction are surveyed. Synchronization primitives based on shared variables and on message passing are described. Finally, three general classes of concurrent programming languages are identified and compared.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent

Although this survey is now very old, it still offers an excellent and very readable introduction to the main concepts of concurrent programming. Most of the content of this introductory lecture is based on this survey.

DOI: 10.1145/356901.356903

Locally available:

http://scgresources.unibe.ch/Literature/CP/Andr83aSurvey.pdf

Concurrency

> A sequential program has a *single thread of control.*

—Its execution is called a process.

> A concurrent program has *multiple threads of control.*

—These may be executed as *parallel processes*.

Beware, the words “thread” and “process” have multiple, overlapping, and inconsistent meanings in Computer Science.

Generally we will use the word “process” to mean *an instance of a running software program*, with its own address space. A “thread” refers to a *“thread of control”* within a given process. A thread, then, does not have its own address space. A process may contain multiple threads.

Parallelism

A concurrent program can be executed by:

Multiprogramming:	processes *share one or more processors*
Multiprocessing:	each process runs on its own processor but with *shared memory*
Distributed processing:	each process runs on *its own processor* connected by a network to others

Assume only that all processes make positive finite progress.

Why do we need concurrent programs?

> *Reactive programming*
>
> —minimize response delay; maximize throughput

> *Real-time programming*
>
> —process control applications

> *Simulation*
>
> —modelling real-world concurrency

> *Parallelism*
>
> —speed up execution by using multiple CPUs

> *Distribution*
>
> —coordinate distributed services

Reactive programming is used in applications characterized by asynchronous events. Programs react to asynchronous streams of data. Events could be user events, or data streams from social media, or even financial data.

In *real-time programming*, programs need to handle events within real-time constraints. Examples are process control applications that react to sensors on an automated factory floor.

Simulation programs can be structured as processes each of which simulates a real-world entity.

Parallelism is closely related to concurrency, but focuses on speeding up algorithms by splitting them up into parts that can run in parallel. Concurrent programs in general don't necessarily have parallelism as a goal, but rather as an inherent aspect of the problem to be solved.

Distribution is also closely related to concurrency, but focuses rather on distributing tasks across a number of more loosely connected devices.

Roadmap

- Course Overview
- Concurrency and Parallelism
- **Challenges: Safety and Liveness**
- Expressing Concurrency
  - Process Creation
  - Communication and Synchronization

Difficulties

But concurrent applications introduce complexity:

Safety

> concurrent processes may *corrupt shared data*

Liveness

> processes may *"starve"* if not properly coordinated

Non-determinism

> the same program run twice may give *different results*

Run-time overhead

> thread construction, context switching and synchronization *take time*

Concurrency and atomicity

Programs P1 and P2 execute concurrently:

```
  { x = 0 }
P1: x := x+1
P2: x := x+2
  { x = ? }
```

What are possible values of x after P1 and P2 complete?

What is the intended final value of x?

Consider carefully all possible interleavings of P1 and P2.

NB: what exactly are you assuming to be the *atomic actions* of P1 and P2?

Safety

Safety = *ensuring consistency*

A safety property says *"nothing bad happens"*

- ***Mutual exclusion:*** shared resources must be *updated atomically*
- ***Condition synchronization:*** operations may be *delayed* if shared resources are in the wrong state
  - *(e.g., read from empty buffer)*

One way of thinking about safety is that the data of our program are not corrupted. This is clearly related to the notion of a *class invariant* in object-oriented programming. Safety (of an object) is ensured if the class invariant is respected.

Another way of thinking about safety is that the program does not reach a “*bad state*”. The “bad state” can manifest itself as inconsistent memory, or perhaps some other undesirable property.

Liveness

Liveness = *ensuring progress*

A liveness property says *“something good happens”*

- ***No Deadlock:*** some process can always access a shared resource
- ***No Starvation:*** all processes can eventually access shared resources

Safety can be ensured trivially simply by not doing anything. (Some bureaucracies function like this.) But a useful program should actually do something useful, which is why we need both safety and liveness.

There can be *many degrees of liveness*. “No deadlock” is fairly weak, as it states that some progress is made in the system. “No starvation” is stronger since it ensures that all components will make some progress eventually.

Roadmap

> Course Overview
> Concurrency and Parallelism
> Challenges: Safety and Liveness
> **Expressing Concurrency**
—**Process Creation**
—Communication and Synchronization

Expressing Concurrency

A programming language must provide mechanisms for:

Process creation

> how do you specify *concurrent processes*?

Communication

> how do processes *exchange information*?

Synchronization

> how do processes *maintain consistency*?

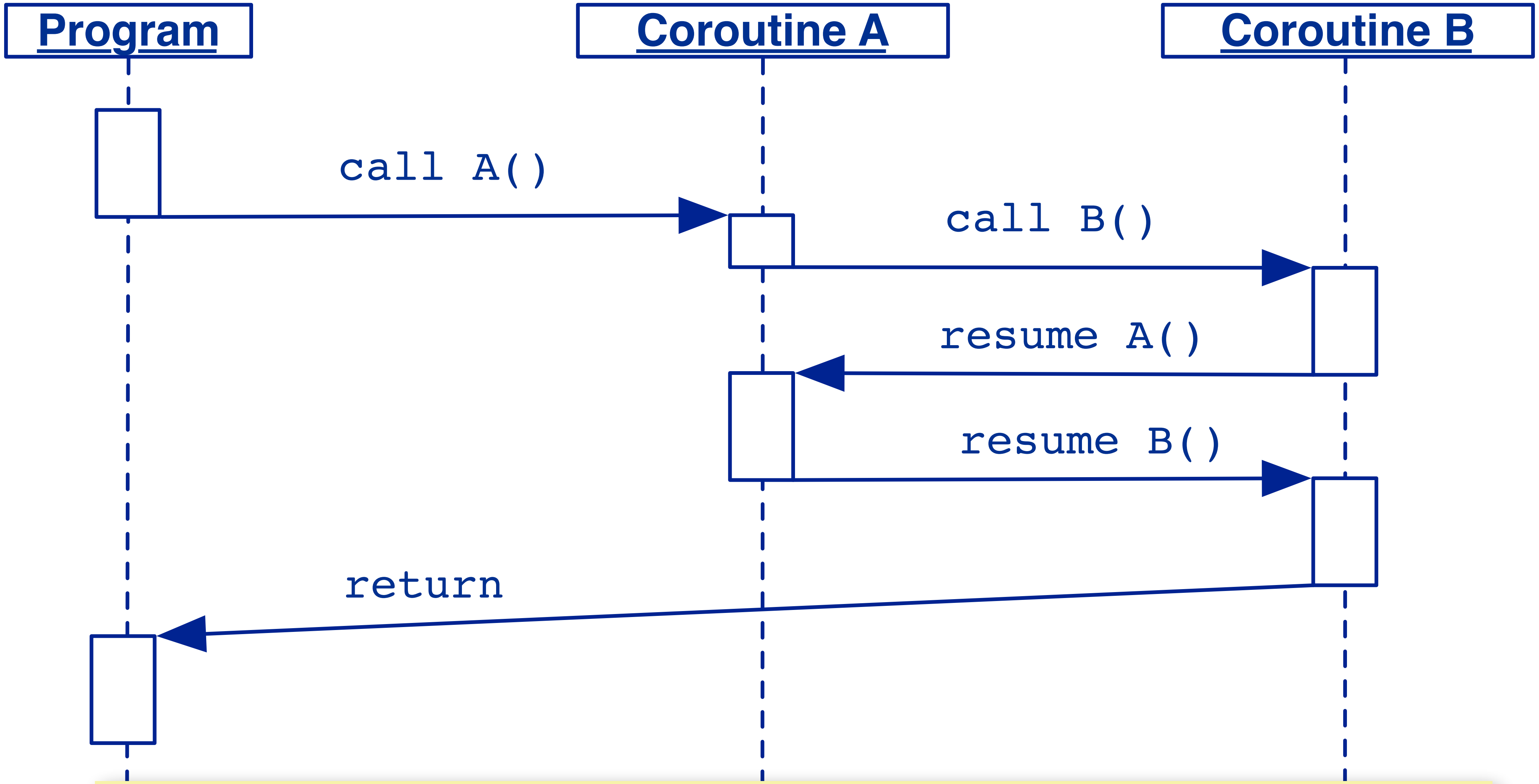
Process Creation

Most concurrent languages offer some variant of the following:

- Co-routines
- Fork and Join
- Cobegin/coend

Co-routines

Co-routines are only *pseudo-concurrent* and require *explicit transfers of control*

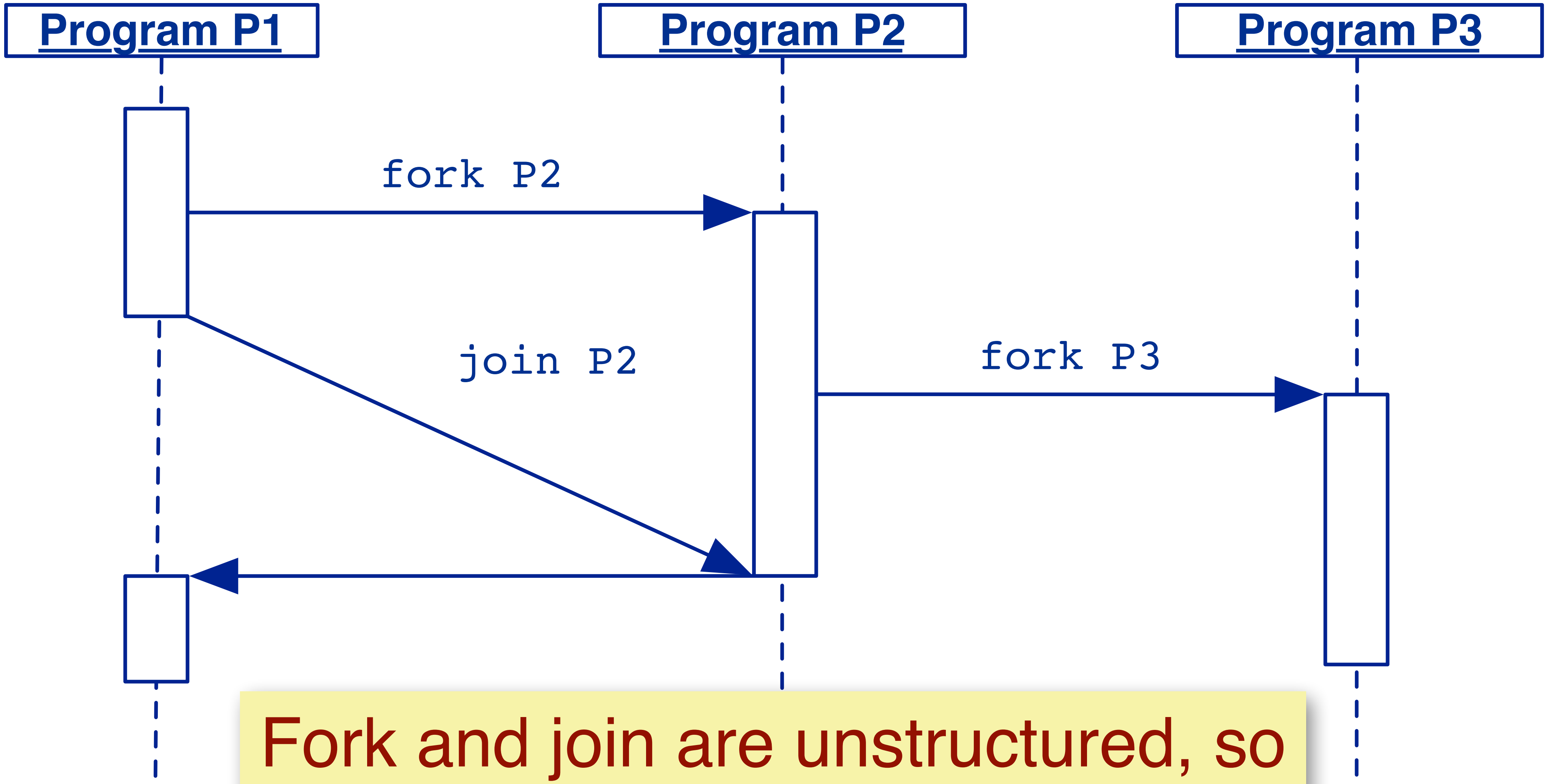


Co-routines can be used to implement most higher-level concurrent mechanisms.

Co-routines are a rather old-fashioned mechanism to explicitly hand off control between multiple threads. They are no longer used in any mainstream programming language, but sometimes you will find them offered as library functions for low-level programs (e.g., assembler) as building blocks for creating higher-level concurrency constructs.

Fork and Join

Fork can be used to create any number of processes:
Join waits for another process to terminate.



Fork and join are unstructured, so require care and discipline!

The “fork and join” mechanism is the most common way to create processes or threads. For example, it is used by both the Unix O/S to create (heavyweight) processes and by Java to create (lightweight) threads.

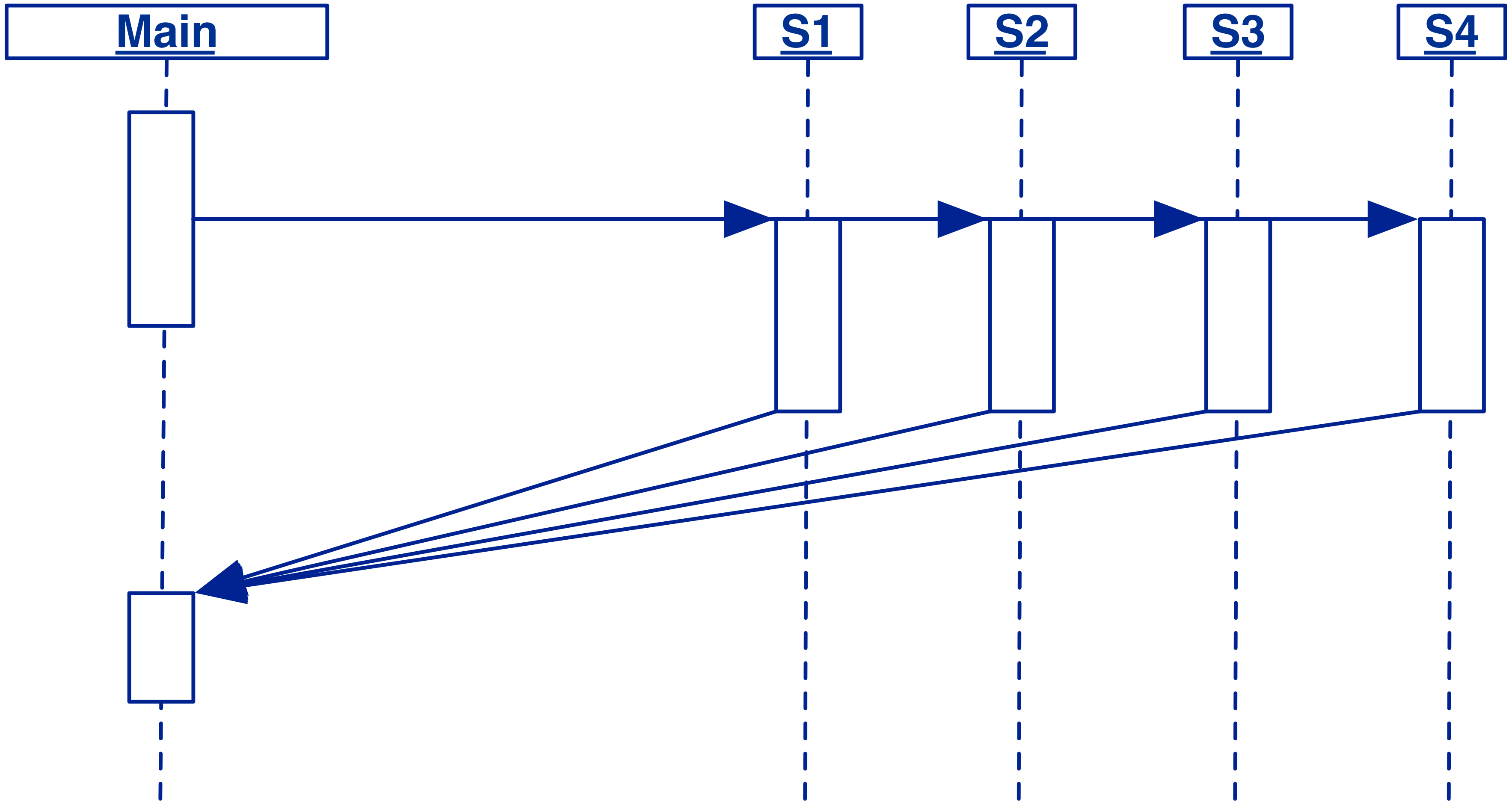
The “join” mechanism provides the way for a parent process or thread to wait until the completion of any of its children.

Cobegin/coend

Cobegin/coend blocks are better structured:

```
cobegin S1 || S2 || ... || Sn coend
```

but they can only create a *fixed number of processes*.
The caller continues when all of the coblocks have terminated.



The “cobegin/coend” statement was introduced by Dijkstra as more structured alternative to fork and join. Its main disadvantage is that only a fixed number of processes or threads can be created.

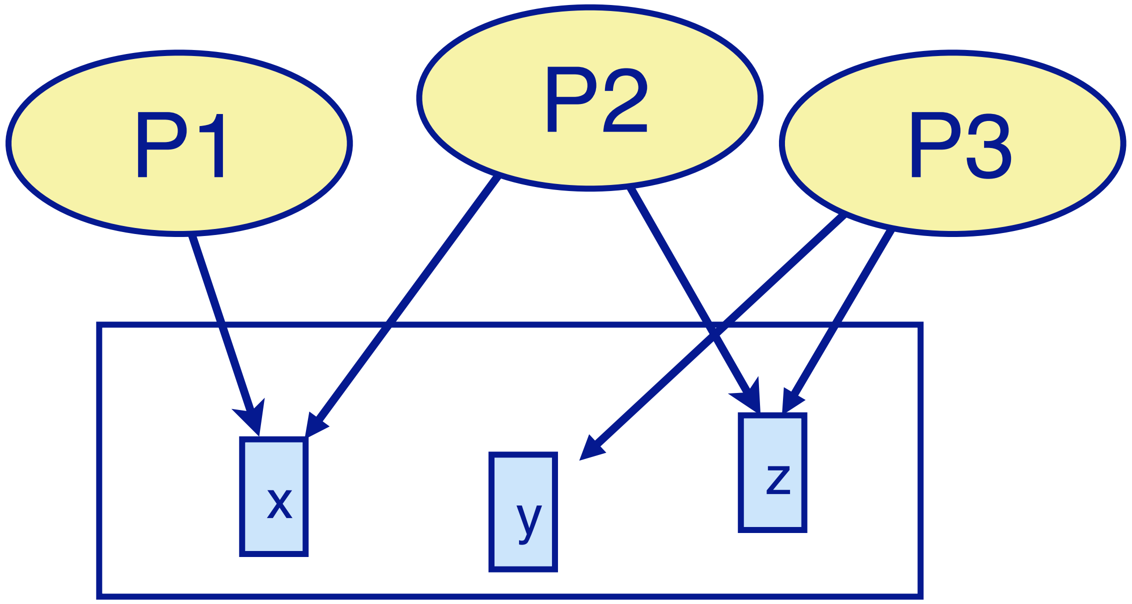
Roadmap

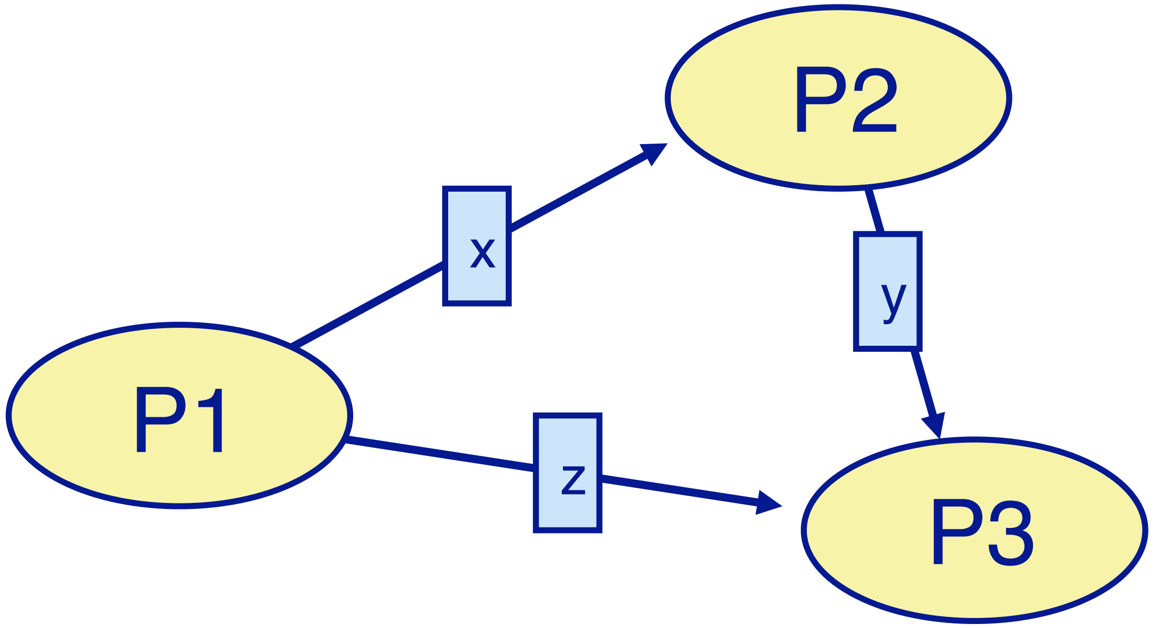
> Course Overview
> Concurrency and Parallelism
> Challenges: Safety and Liveness
> **Expressing Concurrency**
  —Process Creation
  —**Communication and Synchronization**

Communication and Synchronization

In approaches based on shared variables, processes *communicate indirectly*.

Explicit synchronization mechanisms are needed.





In message passing approaches, communication and synchronization are combined.

Communication may be either synchronous or asynchronous.

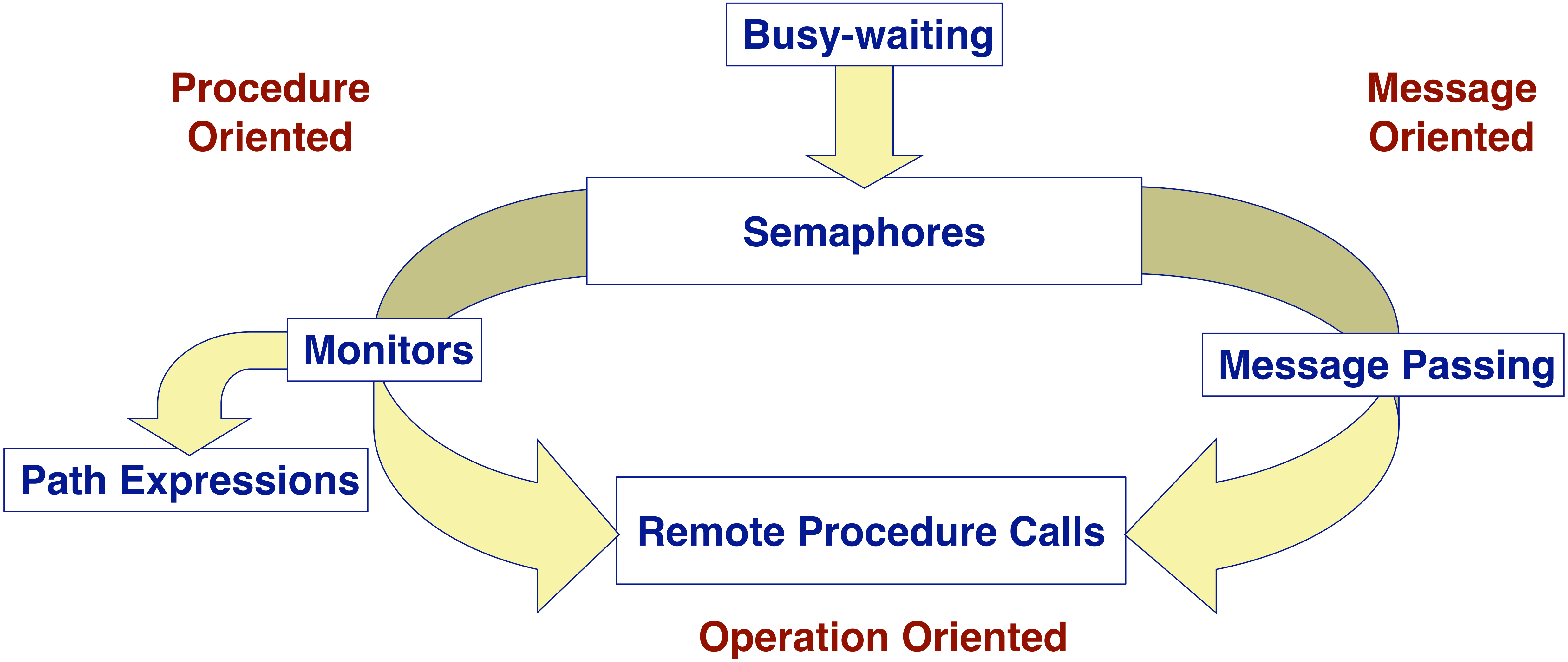
There are two fundamentally different ways of thinking about communication and synchronization in concurrent programs, but they are *functionally equivalent*. It is relatively straightforward to implement or simulate one approach with the other.

In the shared variables approach, processes communicate and synchronize using shared memory. Synchronization mechanisms (such as locks) are needed to ensure safety.

In message-passing approaches, processes directly send messages to each other. Message sending may be *synchronous* (i.e., sending and receiving occur simultaneously) or *asynchronous*. Senders and receivers may be explicitly identified, or some higher-level medium may be used to exchange messages.

Synchronization Techniques

Different approaches are roughly equivalent in expressive power and can be used to implement each other.



Each approach emphasizes a different style of programming.

Andrews and Schneider (in their survey) provide this chart to show how various concurrency approaches are related to each other. The approaches also progress in time from top to bottom. Busy-waiting is one of the oldest techniques, and was supplanted by semaphores in the 1960s. During the 1970s, numerous other approaches were developed, based either on a procedural paradigm, or on a message-passing paradigm. Remote procedure calls sit somewhere in the middle.

Busy-Waiting

Busy-waiting is primitive but effective

Processes *atomically test and set* shared variables.

Condition synchronization is easy to implement:

—to signal a condition, a process *sets* a shared variable

—to wait for a condition, a process *repeatedly tests* the variable

Mutual exclusion is more difficult to realize correctly and efficiently

…

Busy-waiting is one of the oldest mechanisms to support synchronization. As the name suggests, a process repeatedly tests the value of a variable to to determine whether a particular condition holds. The actual low-level primitive is called "test and set", as a process may atomically test whether a variable has a particular value, and then set it. A lock can be modeled by such a variable. If two processes simultaneously try to test and set the lock, then it is guaranteed that just one will succeed.

The other process must then busy-wait, repeatedly attempting a test-and-set until it succeeds to acquire the lock.

Busy-waiting may be effective for hardware, but is not very efficient for software processes which waste cycles as they busy-wait.

See also: https://en.wikipedia.org/wiki/Test-and-set

Semaphores

Semaphores were introduced by Dijkstra (1968) as a higher-level primitive for process synchronization.

A semaphore is a non-negative, integer-valued variable s with two operations:

- **P(s):**
  - delays until s>0
  - then, atomically executes s := s-1
- **V(s)**
  - atomically executes s:= s+1

Semaphores eliminate the need for busy-waiting. A semaphore represents an non-negative integer value with two atomic operations. A V operation always succeeds, and increases the value of the semaphore. A P decreases the value, but only if the current value is not zero.

If the semaphore is zero, then a P will cause the invoking process to wait (without busy-waiting) until another process performs a V.

In case multiple processes are attempting a P, an order may be imposed on waiting processes to ensure fairness (i.e., a queue, or priorities).

Aside: The terms "P" and "V" come from Dutch. V = "verhogen" (to increase), and P = "prolaag" (probeer te verlagen — try to decrease)

See also: https://en.wikipedia.org/wiki/Semaphore_(programming)

Programming with semaphores

Many problems can be solved using binary semaphores, which take on values 0 or 1.

```
process P1
   loop
      P (mutex) { wants to enter }
      Critical Section
      V (mutex) { exits }
      Non-critical Section
   end
end
```

```
process P2
   loop
      P (mutex)
      Critical Section
      V (mutex)
      Non-critical Section
   end
end
```

Semaphores can be implemented using busy-waiting, but usually implemented in O/S kernel.

A “critical section” in a concurrent program is any portion of code that attempts to access a shared resource, i.e., where safety issues may arise.

See also: https://en.wikipedia.org/wiki/Critical_section

Monitors

A monitor encapsulates *resources and operations* that manipulate them:

> operations are invoked like ordinary procedure calls
- —invocations are guaranteed to be *mutually exclusive*
- —condition synchronization is realized using *wait and signal* primitives
- —there exist many variations of wait and signal ...

Monitors were proposed by Per Brinch Hansen and Tony Hoare as a more structured alternative to semaphores. Monitors nicely encapsulate the fact that one normally obtains and releases access rights to a shared resource in a structured way.

The precise semantics of *wait* and *signal vary* from implementation to implementation.

Java supports a variant of monitors.

See also: https://en.wikipedia.org/wiki/Monitor_(synchronization)

Programming with monitors

```
type buffer(T) = monitor
  var
  slots : array [0..N-1] of T;
  head, tail : 0..N-1;
  size : 0..N;
  notfull, notempty:condition;

procedure deposit(p : T);
  begin
     if size = N then
        notfull.wait
     slots[tail] := p;
     size := size + 1;
     tail := (tail+1) mod N;
     notempty.signal
  end
```

```
procedure fetch(var it : T);
  begin
     if size = 0 then
         notempty.wait
     it := slots[head];
     size := size - 1;
     head := (head+1) mod N;
      notfull.signal
  end

begin
  size := 0;
  head := 0;
  tail := 0;
end
```

This is an example of a bounded buffer implemented using monitors in an unnamed programming language. A monitor is a *data type* (i.e., a class) that encapsulates internal data and synchronized operations operating on that data.

The `buffer` type encapsulates an array of items, `head` and `tail` variables that identify the first and last elements within the array (i.e., that cycle around), the current `size` of the buffer (number of elements stored in the array), and two condition variables that express whether the buffer is *not full* or *not empty*.

The `deposit` and `fetch` operations follow a classical monitor style in which a condition variable is checked before performing any operation. The monitor guarantees that only one operation may gain access to the monitor at a time. If the wait condition fails, the “lock” is released and another waiting thread can gain access.

Here, if a thread is blocked on *deposit* because the buffer is full, a *fetch* thread may get in, and signal *notfull* at the end, thus waking up the waiting thread.

Problems with monitors

Monitors are more structured than semaphores, but they are still tricky to program:

- Conditions must be manually checked
- Simultaneous signal and return is not supported

A signalling process is temporarily suspended to allow waiting processes to enter!

- The *monitor state may change* between signal and resumption of signaller
- Unlike with semaphores, *multiple signals are not saved*
- *Nested monitor calls* must be specially handled to prevent deadlock

Path Expressions

Path expressions express the *allowable sequence of operations* as a kind of regular expression:

```
buffer : (put; get)*
```

Although they elegantly express solutions to many problems, path expressions are too limited for general concurrent programming.

It seems as though path expressions a rediscovered every few years. Path expressions are ideal for expressing synchronization that follows a well-defined sequences of states, as in a state machine. (Think of the equivalence of regular expressions and finite state automata.) When synchronization does not fit into this pattern (e.g., if there is an unbounded number of synchronization states), then path expressions fail miserably.

Message Passing

Message passing combines communication and synchronization:

> The sender specifies the *message and a destination*
  - —a process, a port, a set of processes, ...

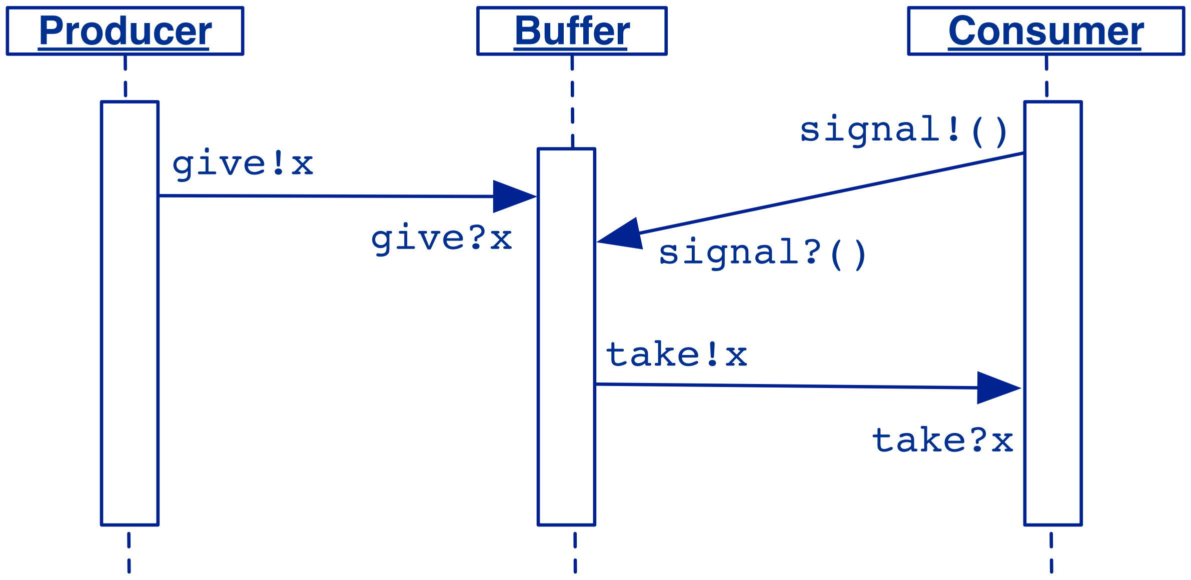
> The receiver specifies *message variables and a source*
  - —source may or may not be explicitly identified

> Message transfer may be:
  - —asynchronous: send operations *never block*
  - —buffered: sender *may block if the buffer is full*
  - —synchronous: sender and receiver *must both be ready*

Send and Receive

In CSP and Occam, source and destination are explicitly named:

```
PROC buffer(CHAN OF INT give, take, signal)
  ...
  SEQ
    numitems := 0 ...
    WHILE TRUE
    ALT
      numitems ≤ size & give?thebuffer[inindex]
        SEQ
          numitems := numitems + 1
          inindex := (inindex + 1) REM size
      numitems > 0 & signal?any
        SEQ
          take!thebuffer[outindex]
          numitems := numitems - 1
          outindex := (outindex + 1) REM size
```



NB: The consumer must signal!any to inform the buffer that it wishes to take?avalue

In this example, a buffer can synchronize either with a producer (as long as the buffer is not full) or with a consumer (as long as it is not empty).

Communication with the producer consists in a single action, in which the producer sends a `give` message with the payload `x` to the buffer.

Communication with the consumer consists in two steps: first the consumer sends a `signal` message, thereby requesting a value; then the buffer sends it the value at the head of the queue with a synchronous `take` message.

Remote Procedure Calls and Rendezvous

In Ada, the caller identity need not be known in advance:

```
task body buffer is ...
begin loop
   select
     when no_of_items < size =>
      accept give(x : in item) do
        the_buffer(in_index) := x;
      end give;
      no_of_items := no_of_items + 1; ...
   or
     when no_of_items > 0 =>
      accept take(x : out item) do
        x := the_buffer(out_index);
      end take;
      no_of_items := no_of_items - 1; ...
   end select;
  end loop; ...
```

In contrast to Occam, Ada tasks will accept messages from unknown senders. This starts to look much more like monitors (or objects), where only the sender needs to know the identity of the receiver.

What you should know!

- *Why do we need concurrent programs?*
- *What problems do concurrent programs introduce?*
- *What are safety and liveness?*
- *What is the difference between deadlock and starvation?*
- *How are concurrent processes created?*
- *How do processes communicate?*
- *Why do we need synchronization mechanisms?*
- *How do monitors differ from semaphores?*
- *In what way are monitors equivalent to message-passing?*

Can you answer these questions?

> *What is the difference between concurrency and parallelism?*
> *When does it make sense to use busy-waiting?*
> *Are binary semaphores as good as counting semaphores?*
> *How could you implement a semaphore using monitors?*
> *How would you implement monitors using semaphores?*
> *What problems could nested monitors cause?*
> *Is it better when message passing is synchronous or asynchronous?*