Concurrency and Parallelism

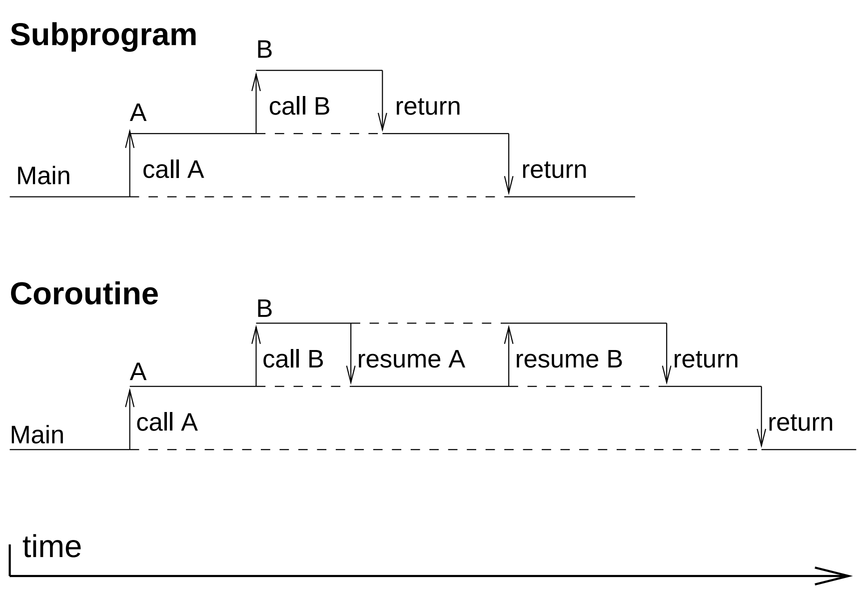
- Concurrency and Parallelism are independent
- (true) parallelism is a hardware concept:
  - >1 processors; operations overlap in time
  - shared or local memory
- quasi-parallel:
  - single processor/memory
  - OS simulates parallelism via time-slicing
- concurrency is a software concept:
  - program objects are considered concurrent if they *could* be executed in parallel
  - concurrent operations *do not necessarily have to be executed in parallel*
  - many "real world" problems are easiest to program using a concurrent execution model
    - ◇ classic example: Operating system

Processes, Threads and Tasks

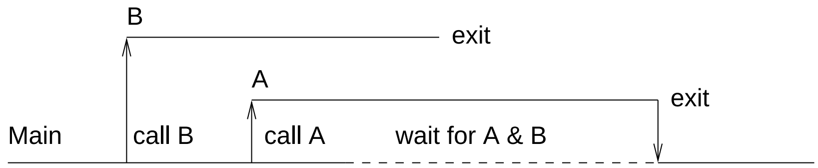
- a **process** is a program being executed; it contains.
  - program instructions
  - state (one program counter, one RTS, data, ...)
  - one *thread* of control per process
  - also called: **heavyweight process**, traditional process
- concurrent programs have possibly many active processes
  $\rightarrow$ they are described as *multithreaded*
- multithread programs have > 1 control thread
  - parallel processors – one process/CPU
    - ◇ OS manages synchronisation & context switching
  - single processor: many **threads** per process
    - ◇ each thread has own PC, stack, but share globals
    - ◇ controller *within the process* manages synchronisation & context switching
    - ◇ also called **lightweight process** or **task**

Standard control flow

- many units are *active*
- only one unit is executing at any time
- subroutines: master/slave relationship
- coroutines:
  – sharing of processing task
  – multiple entry points



Concurrent execution



- Main, A and B are independent
  – 3 tasks: Main, A, B
- Main waits for A & B to finish

Major concurrency issues

1. competition: e.g. ≥ 2 processes need to write database
2. cooperation: e.g. ≥ 2 processes share the processing task
3. *safety* and *liveness* of processes

- (1, 2) require:*synchronisation* of actions of concurrent processes
- (2) requires: *communication*: exchange of data between concurrent processes

Safety

Do all possible sequences result in the same state?

Consider two processes

P1: `t := x; x := t+1` ($\equiv$ `x:=x+1`)
P2: `u := x; x := u+2` ($\equiv$ `x:=x+2`)

If P1 & P2 run sequentially the final state is the same

P1 then P2 $\equiv$ P2 then P1 $\equiv$ `x := x+3`

Consider the cases if P1 & P2 run concurrently

Process 1	Process 2
`t := x`	
	`u := x`
	`x := u+2`
`x := t+1`	

$\equiv$ `x := x+1`

Process 1	Process 2
`t := x`	
	`u := x`
`x := t+1`	
	`x := u+2`

$\equiv$ `x := x+2`

Process 1	Process 2
`t := x`	
`x := t+1`	
	`u := x`
	`x := u+2`

$\equiv$ `x := x+3`

Conclusion:

- *Interleaving* can lead to inconsistent state
- this is a competition synchronisation problem
  – both P1 and P2 compete to write to `x`

Liveness

Can a process complete in reasonable time?

Example: P1 and P2 both require exclusive access to X and Y

"Safe" interleaving where locking occurs in same order

Process 1	Process 2
`lock X`	
	`lock X` *(wait)*
`lock Y`	
`update X,Y`	
`unlock Y`	
`unlock X`	*(continue)*
	`lock Y`
	`update X,Y`
	`unlock X`
	`unlock Y`

Deadlock can occur if locking operations or in different order

Process 1	Process 2
`lock X`	
	`lock Y`
`lock Y` *(wait)*	
	`lock X` *(wait)*

- both examples illustrated competition concurrency
- the problem is one of *synchronisation*
- cooperation concurrency is similar but may also involve data exchange as well.
- data exchange: via shared data or messages
- a range of language features have been proposed to facilitate synchronisation and safe data exchange
  - *semaphores*: a synchronisation mechanism only
  - *monitors*: synchronisation + shared data
  - *message passing*: synchronisation + messages

Synchronisation via Semaphores

- proposed by Dijkstra (Algol68, Modula-2)
- two primitives: P (`wait`) and V (`signal`)
- a semaphore s is an integer variable
  `wait`(s) = wait until $s > 0$; $s := s - 1$
  `signal`(s) = $s := s + 1$
- implementation requires for each semaphore
  – FIFO queue of processes + integer variable
- usage: call `wait` before entering a *critical region* that requires exclusive access; call `signal` on exit from region.
- programmer is responsible for correct usage
- use of semaphores does not guarantee safeness of liveness; semaphores just provide a locking mechanism
- semaphores can be implemented in any language provided that wait and signal can be implemented as *atomic* operations.

Semaphore example

P1: `wait(s); t := x; x := t+1; signal(s);`

P2: `wait(s); u := x; x := u+2; signal(s);`

Main: `var s:sem; s:=1; start(P1); start(P2)`

Note: semaphore s specifies the number of processes that can be in the critical region. Normally set initially to 1 to guarantee exclusive access.

Process 1	Process 2	s
		1
`wait(s)`		0
`t := x`		0
	`wait(s)` *(wait)*	0
`x := t+1`		0
`signal(s)`		1
	(resume)	0
	`u := x`	0
	`x := u+2`	0
	`signal(s)`	1

Monitors

- Combine shared data and semaphore mechanism into a single ADT-like unit
- competition synchronisation is handled by allowing data access only via a monitor function.
  – no need for error prone semaphore
- cooperation synchronisation must be provided by semaphores
  – each language has different semaphore implementation
- Modula-2, Concurrent Pascal provides these facilities

Synchronisation via Message passing

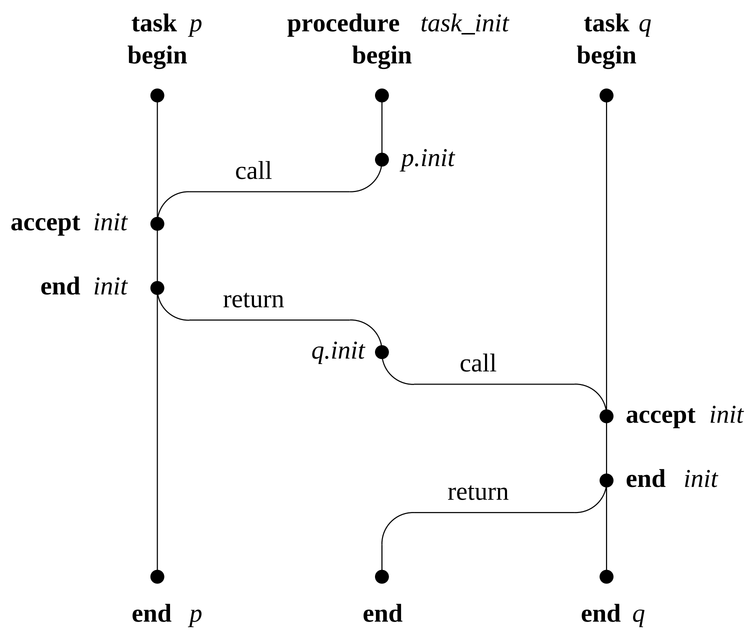
- synchronisation is via a *rendezvous*
  French: “meeting”
- rendezvous is a procedure call
- *client* calls the rendezvous
- *server* receives/accepts the rendezvous
- rendezvous can pass data as arguments
- ‘caller’ suspended during rendezvous

Ada example

```
procedure task_init is
    task type emitter is          -- declare "emitter" as a task
        entry init (c:character)  -- that will accept a rendezvous
    end emitter;                  -- called "init

    p,q: emitter;                 -- create 2 tasks

    task body emitter is          -- emitter implementation
        me: character;
    begin
        accept init(c: character) do  -- rendezvous
            me := c;                  -- critical region
        end init;
        put(me); new_line;
    end emitter;

begin
    p.init('p'); q.init('q'); put('r');
end task_init;
```

Produces output ‘p’ then ‘q’ then ‘r’

Control threads from example

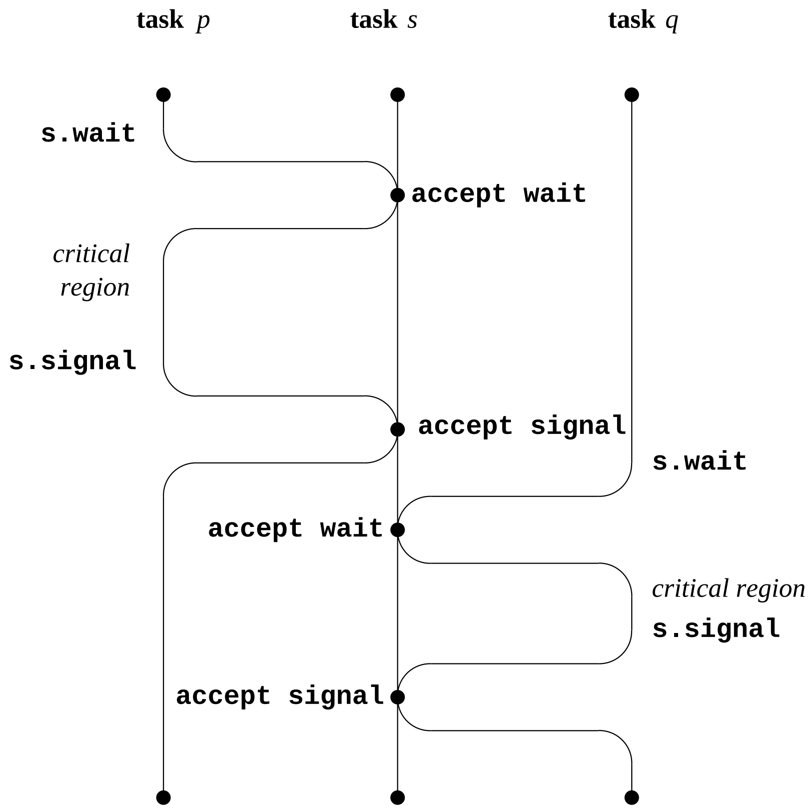


Implementing Semaphores via message passing

```
task type binary_semaphore is
    entry wait;
    entry signal;
end binary_semaphore;

type sem_ptr is access binary_semaphore;

task body binary_semaphore is
begin
    loop
        accept wait;
        accept signal;
    end loop;
end binary_semaphore;

s : sem_ptr;

begin
    s := new binary_semaphore;
    ...
    s.wait;
    ...
    s.signal;
    ...
end;
```

Example

Tasks p and q use s to synchronise. p accesses shared data within critical region



Java threads

- Java objects can be concurrent if they either
  – inherit from predefined class `Thread`, or
  – implement the interface `Runnable`
- a threaded object defines two methods overriding those in `Thread`
  – `run` – which contains the body of the concurrent task, and
  – `start` – which calls `run`
- a task is started by calling `start` which immediately returns
- The Java runtime system contains a scheduler to control execution of runnable tasks.
- Threads can change priority (`getPriority`/`setPriority`)
- competition synchronisation is via `synchronized` methods or blocks
  – only one synchronised method/block can execute at a given time
  – thus private data in a class cannot have multiple readers/writers
  – similar functionality to a monitor

```
class Foo {
    private int xyz[100];     // protected data goes here
    public synchronized void writer (...) {...}
    public synchronized int  reader (...) {...}
}
```

- cooperation synchronisation uses `wait()` and `notify()`
  - `wait()` is called if thread cannot continue (perhaps because some other process is in a critical region); caller is suspended and placed in queue
  - `notify()` is called when a thread has completed a critical operation; other waiting tasks can then recheck and possibly continue.
- see Sebesta for examples.

Evaluation

- semaphores are primitive and error prone, but are sufficient
- monitors provide safer competition synchronisation
  – Java `synchronized` methods provide similar capability
- rendezvous does not need shared data
  $\Rightarrow$ can be used for true concurrency/distributed systems
- all cooperation synchronisation mechanisms are subject to programmer error: possible to generate deadlock situations

See Sebesta for further examples (e.g. Java, Ada95)