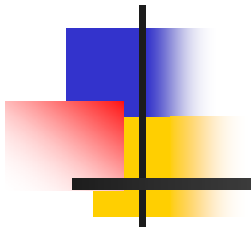
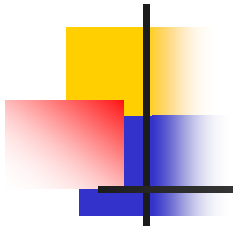


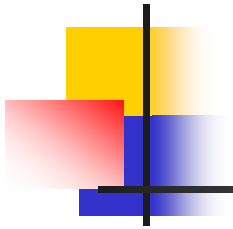
Concurrent Programming

Alexandre David
1.2.05
adavid@cs.aau.dk



Disclaimer

- Overlap with PSS
  - Threads
  - Processes
  - Scheduling
- Overlap with MVP
  - Parallelism
  - Thread programming
  - Synchronization
- But good summary of concepts in a practical context.

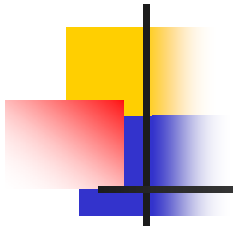


Disclaimer’

- The ADA examples are irrelevant w.r.t. the language itself.

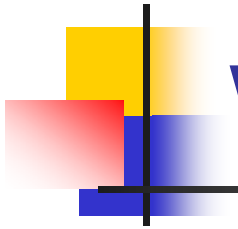
- ⇒ Extract general concepts and constructs.
- ⇒ Map them to different languages.

- We are not ADA experts and we do not need to be.



Overview

- Introduction to concurrent programs.
  - Amdahl's law.
- Tasks & task support.
- Examples.



Why concurrent programming?

- Natural parallelism in real-world.
  - Reflect inherent parallelism of systems.
- Use more efficiently processors.
  - CPU speeds >> input/output: busy wait, polling, or blocking are not good.
- Parallel computations.
  - The speed of light + heat limit processor speeds.
  - The catch: Amdahl's law.

Amdahl's law

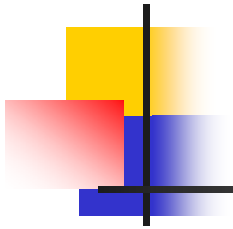
- Inherent sequential costs will limit speedup.

> If a problem of size W has a serial component W_S then the speedup $S \leq W/W_S$ for any p.

Size W corresponds to the serial execution time.
T_P=serial part+(non-serial part)/p
$S=T_S/T_P=W/(W_S+(W-W_S)/p) \leq W/W_S$.

For $W_S=(1-P)W$ we have $S \leq 1/(1-P)$

Note: Problem size here = execution time to abstract from particular problem complexities.



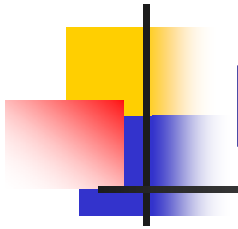
Processes/Threads/Tasks

- Process = program + context + memory space.
  - Thread = program + parent (shared) memory space + private stack.

OS/platform view

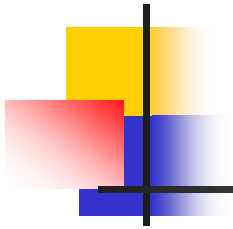
- Task: Abstraction on OS and execution means.
  - Program/procedure.

Logical view



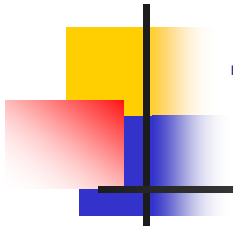
Processes/Threads/Tasks

- DO **NOT** CONSIDER TASKS=THREADS like the book is suggesting.
- Reasoning at the task level is better.
  - Better granularity.
  - Logical view.
  - Abstraction from the platform.
  - Up to a library/language to map tasks to threads.
  - See Intel's Threading Building Blocks, nice read.



So how do we do it?

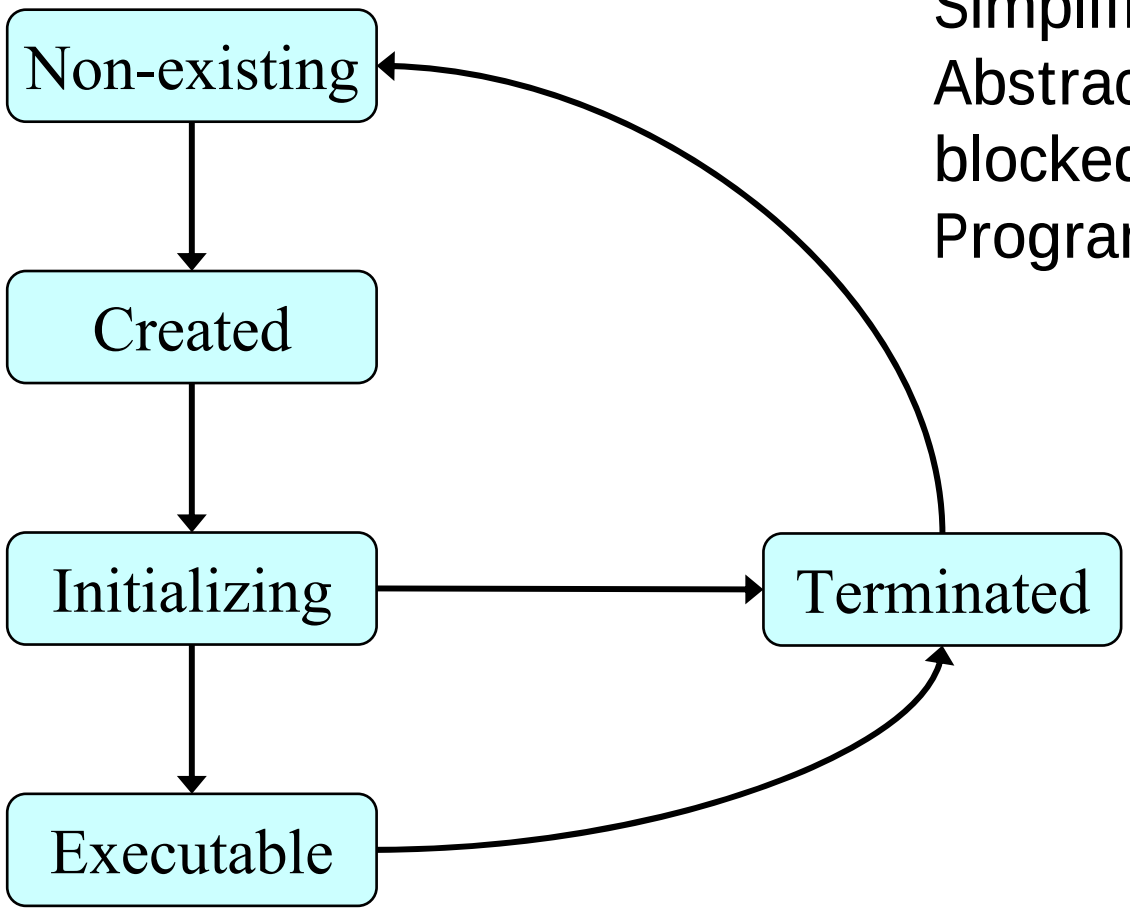
- MVP main topic.
- Hints here:
  - Decompose problems into tasks.
  - Run tasks in parallel.
  - Synchronize tasks whenever needed.
  - Be careful with shared resources.



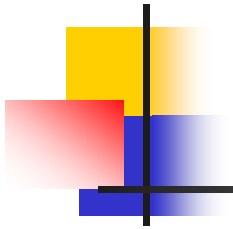
Terminology

- A concurrent program is a collection of autonomous sequential tasks, executing (logically) in parallel.
  - Each task is mapped to a single thread.
  - One thread may execute several tasks.
- Possible execution:
  - Tasks run on one processors (multiprogramming).
  - Tasks run on an SMP machine (multiprocessing).
  - Tasks run on different machines (distributed processing).

Task States



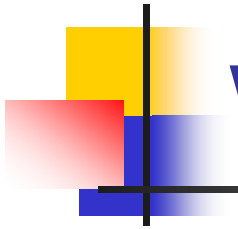
Simplified.
Abstract running, blocked, suspended...
Programmer's view.



Support for parallelism

- Library level
  - C/C++, pthreads.
- Language level
  - Java/ADA, language threads.

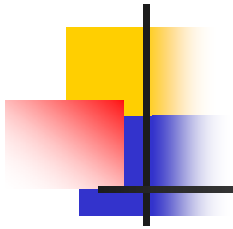
- Different structures: static/dynamic.
- Different levels: flat/nested.



What you can do

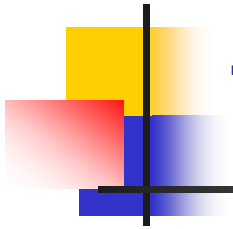
- Start tasks (or threads).
- Synchronize tasks
  - semaphores, mutex, condition variables.
- Use inter-process communication (IPC)
  - message passing, pipes, shared memory.
- Stop tasks.
  - Exception, exit, cancel, never.

- Constraints: dependencies (task dependency graph), shared resources (exclusive access).



Nested tasks

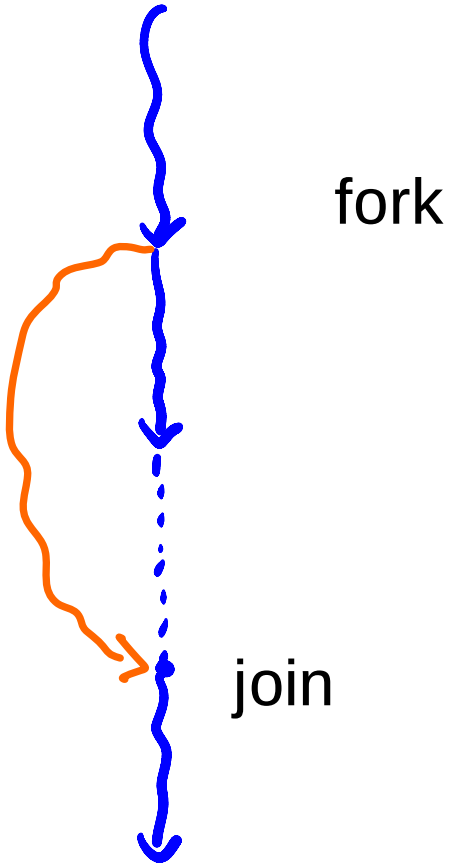
- Hierarchies of tasks.
- Relationship:
  - Parent/child tasks.
  - Guardian/dependent: Logical block entered by the guardian (parent) that creates tasks (children).
    - The guardian may exit the block when all dependent have terminated.
    - Structure: fork()...wait() or create_thread()...join().

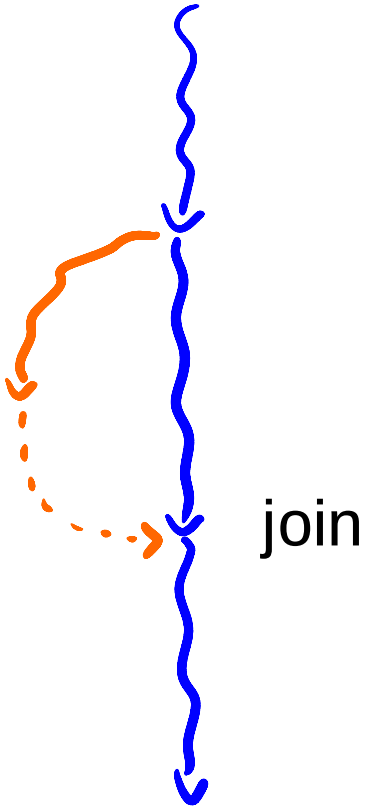


Task representation

- Fork & join
- cobegin
- task declaration

Fork & join concepts



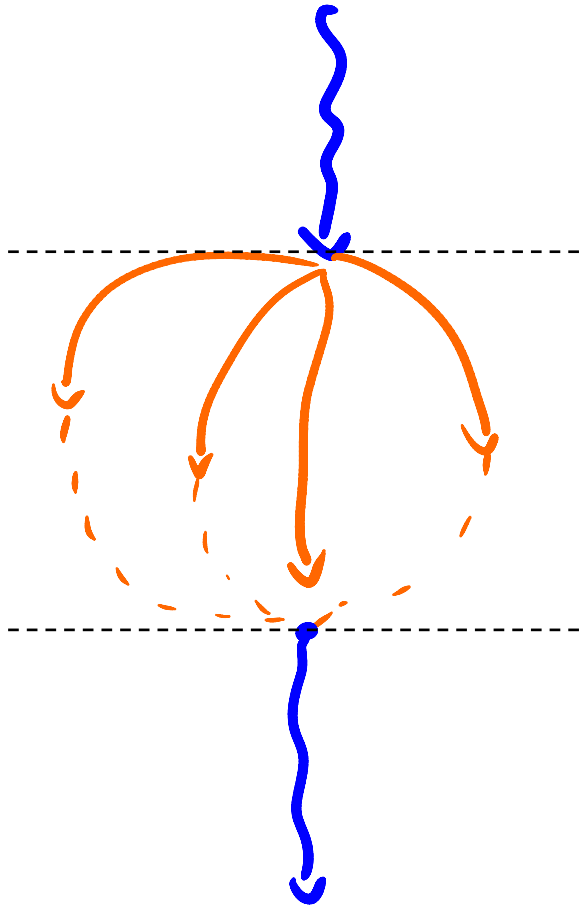


map to...

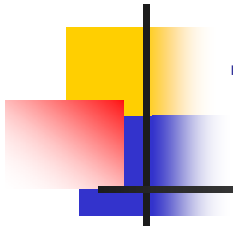
Parent calls join 1st.

Child finishes 1st.

Cobegin concept



```
cobegin
  statement;
  statement;
  ...
coend
```

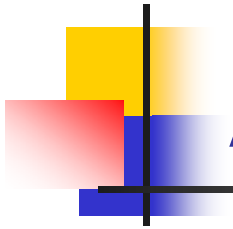


Task declaration

- Explicit declaration of tasks.
  - Task A() {…} Task B() {…}
  - Main program setups parameters (periods) and enters a loop – may listen to events there.

Language vs. OS supported concurrency

- + language
  - +readable +maintainable programs
  - $\neq$ types of OS, same lang. $\rightarrow$ +portable programs
  - possible to have no OS
- - language
  - Models of concurrency are language specific $\rightarrow$ delicate to mix $\neq$ languages.
  - May be difficult to implement efficiently the language model of concurrency of top of an OS.
  - POSIX standard for OS API, better portability.



Ada

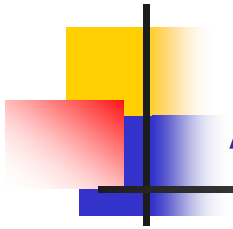
- Unit of concurrency = task.
  - Explicitely declared.
  - Created implicitly.
- Tasks communicate & synchronize via
  - rendezvous (~sync message passing)
  - protected units (~monitor, condition variable)
  - shared variables

Example Task Structure

```
task type Server (Init :
  Parameter) is
  entry Service;
end Server;
task body Server is
begin
  ...
  accept Service do
    -- Sequence of
statements;
  end Service;
  ...
end Server;
```

specification

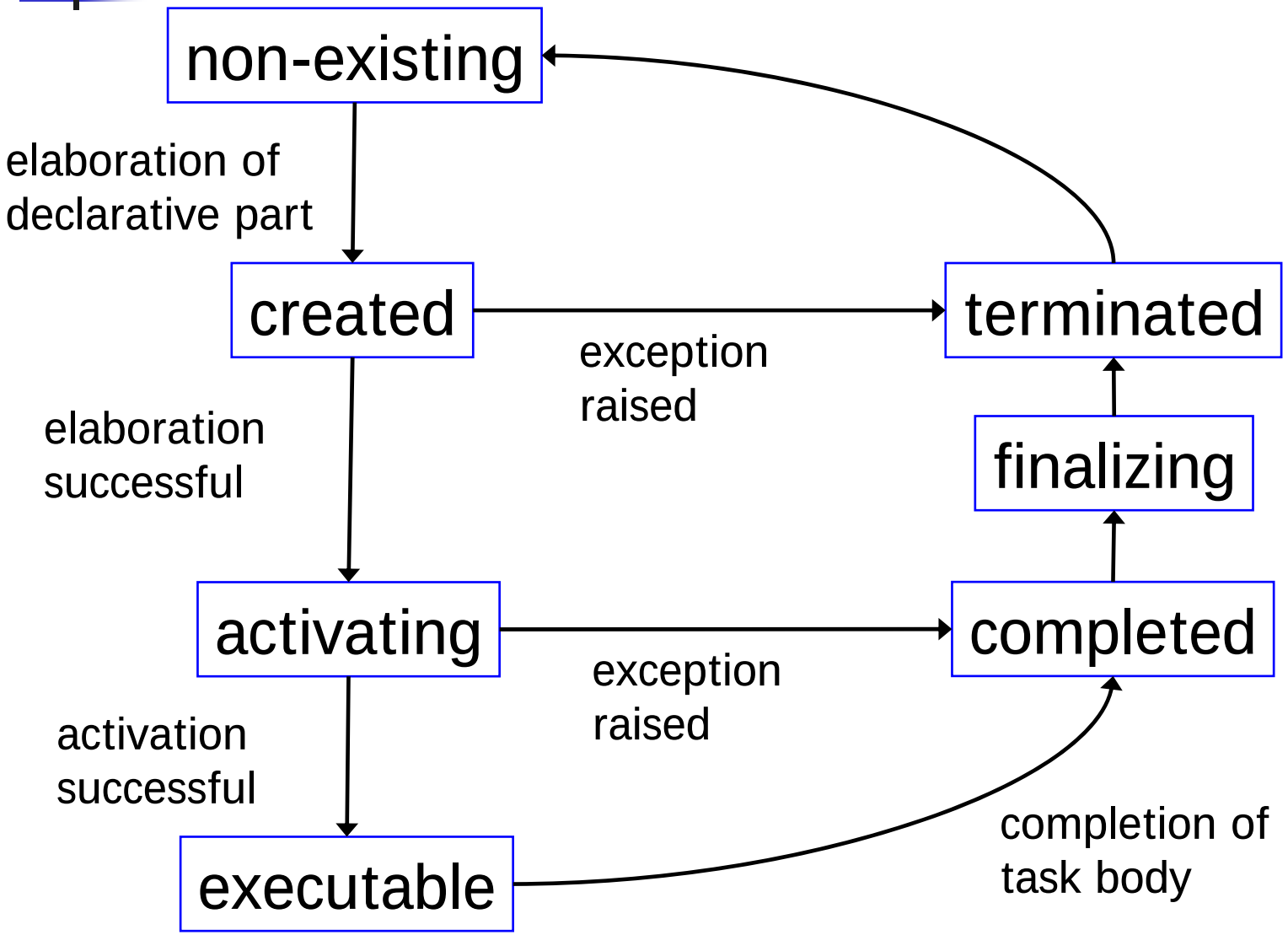
body

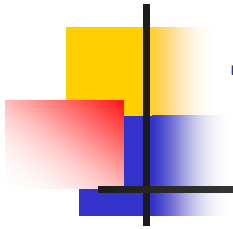


Activation, Execution, & Finalization

- Activation: task declaration, allocation of variables, creation of the task (~constructor).
- Execution: execute statements in the body.
- Finalization: execute finalization code associated with declared objects (~destructor).

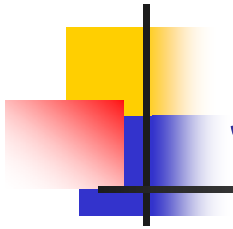
Task states in Ada





Task Identification

- Type in Ada.
- Integer in C (process ID) or type (pthread_t).
- Reference to Thread in Java.
- Way to identify a task to
  - communicate
  - wait for it
  - cancel it.

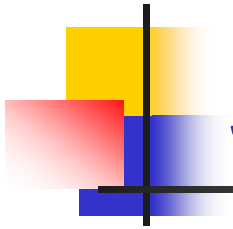


Java

- Class Thread
  - methods run(), start(), isAlive(), join()...
  - and interface Runnable.

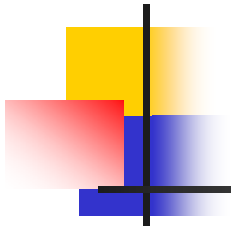
```
public interface Runnable {
    public abstract void run();
}
```

    Running object is Thread so you can either
    - extend Thread, or
    - implement Runnable and create new Thread(runMe).
  - Interface is there to avoid extending Thread (single inheritance in Java).



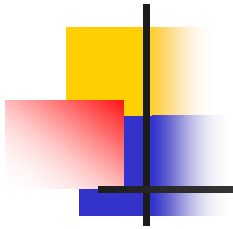
Java threads

- Dynamic thread creation.
- Pass any data to constructor.
- Thread groups (no master/guardian).
- Main program terminates when all threads terminate.
- One thread can wait for another by calling its join().
- Some thread specific exception.



POSIX

- Process
  - fork(): copies current process (everything).
- Thread
  - pthread_create(): create a thread in the address space of its parent process.
- Mutex, condition variable, and semaphores.
- Shared memory
  - threads share the same address space or
  - process can allocate shared memory.



Summary

- Concurrency important:
  - Real-world inherently concurrent.
  - Support by OS or language (better).
  - Modeled as tasks.
    - States.
- Variations in the task model, different mapping to different languages.
  - static/dynamic, flat/nested, fine/coarse,
  - how to terminate,
  - how to declare/create (fork, join, cobegin, explicit declaration).