Real-Time Systems Programming

Summer-Semester 2002
Lecture 16
7 June 2002

Concurrency

The following foils were adapted from the ones provided by Burns & Wellings, gratefully acknowledged here!

The 5 Minute Review Session

1. *Fault avoidance:* What are the limits of testing ?
2. *Fault tolerance:* What are the limits of HW redundancy ?
3. What types of *voting* exist ?
4. What are the *requirements* on an exception handling facility?
5. What *aspects* are there of exception handling?

Characteristics of RTS

- Large and complex
- ***Concurrent control of separate system components***
- Facilities to interact with special purpose hardware
- Guaranteed response times
- Extreme reliability
- Efficient implementation

Aim

- To illustrate the requirements for concurrent programming
- To demonstrate the variety of models for creating processes
- To lay the foundations for studying inter-process communication

Contents

1. What is Concurrency ?
2. Why do we need it ?
3. Cyclic executives
4. The run-time support system
5. Types of processes
6. Process states
7. Process representations

Concurrent Programming

> ***Concurrent Programming*** *is the name given to programming notation and techniques for expressing* ***potential parallelism*** *and solving the resulting synchronization and communication problems.*
>
> ***Implementation of parallelism*** *is a topic in computer systems (hardware and software) that is essentially* ***independent*** *of concurrent programming.*
>
> *Concurrent programming is important because it provides an abstract setting in which to study parallelism* ***without getting bogged down in the implementation details****.*
>
> *Ben-Ari, 1982*

•Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming.

•The topic has been around for a long time (can we traced back to Conway and Dijkstra in the early 60's)

•Perhaps its defining moment was the publications of Dijkstra's paper on Co-operating Sequential Processesin 1965

•First concurrent programming language Simula 66?

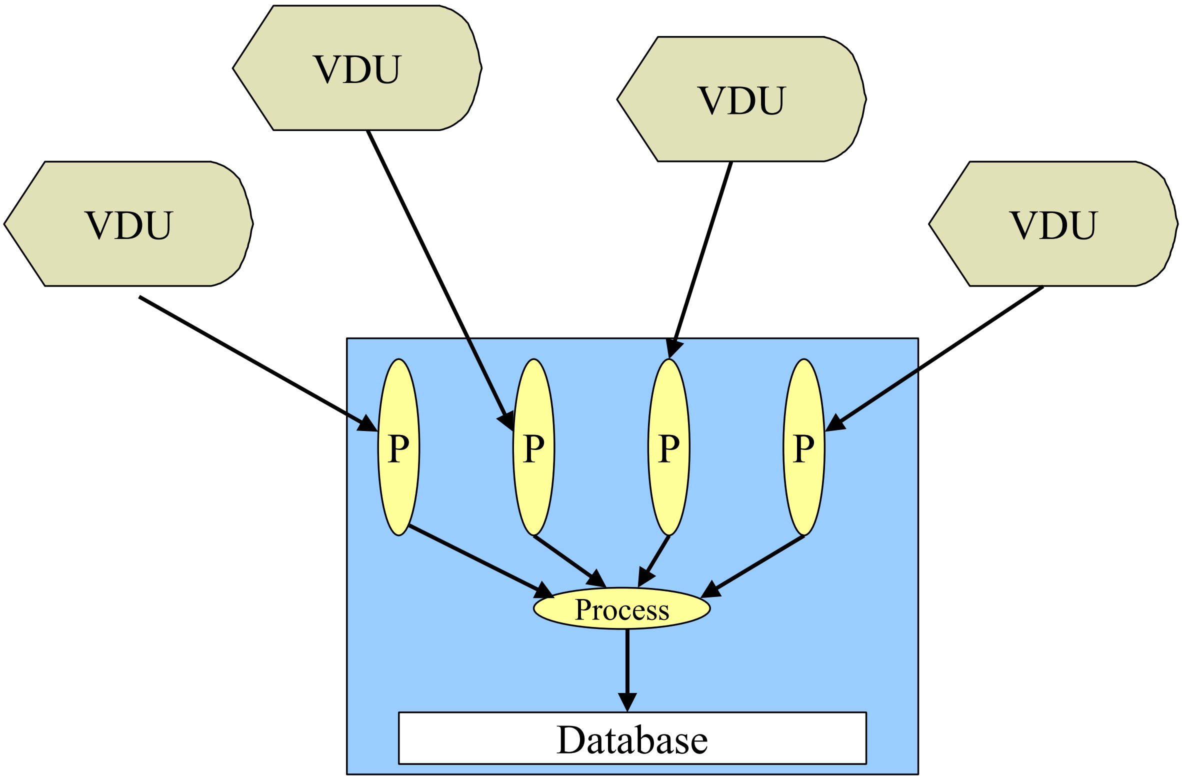
•Not until 1983 that we see the first international standard concurrent programming language - Ada.

•Indeed is not universally accepted that programming languages should be concurrent

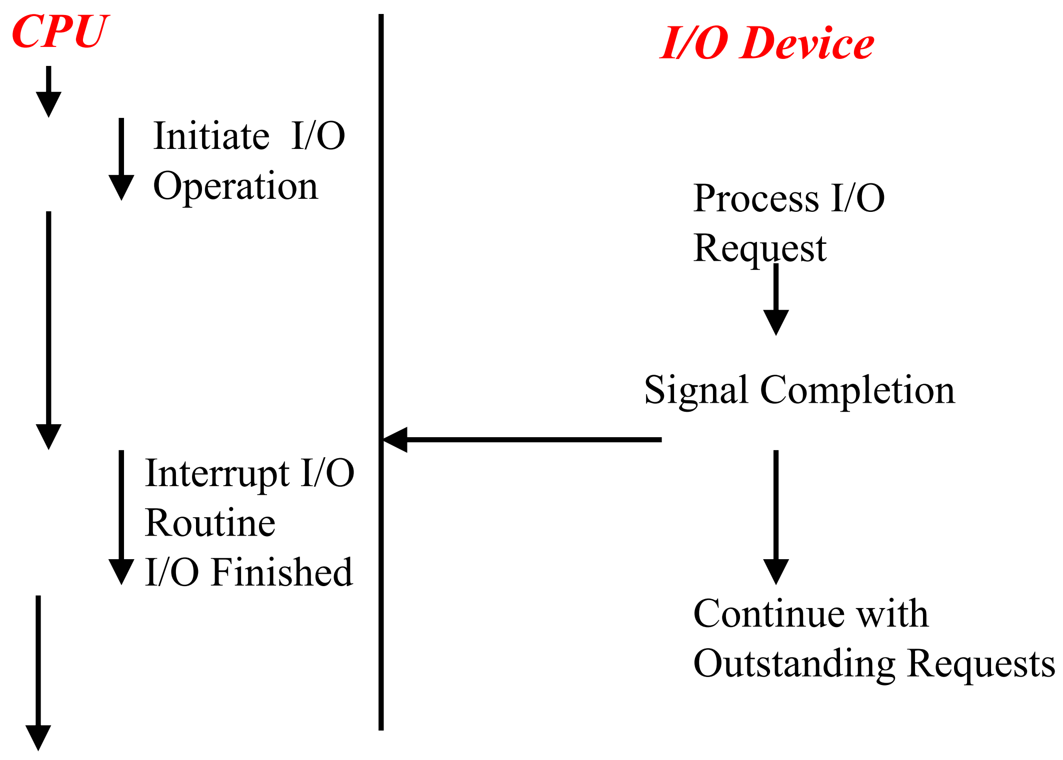
Why Concurrency ?

- ***To model the parallelism in the real world***
- Virtually all real-time systems are inherently concurrent — devices operate in parallel in the real world
- Another reason: To allow the expression of potential parallelism so that ***more than one computer*** can be used to solve the problem

Example: Airline Reservation System



Example: CPU and I/O Devices



Example: Autonomous Vehicle

- *Mars rover:* Control software may (concurrently)
  - Take in the surrounding terrain to map a path
  - Get sensor inputs on which wheels touch the ground
  - Control the power put out to any motor
  - Sample air, temperature, light, scoop up little pieces of Mars
  - Get input from humans back on earth („*unjam that stoopid antenna!*“)
- Inputs may be ***related*** or not
- SW has to provide ***timely response to all inputs***

Terminology

- A ***concurrent program***:
  - Collection of autonomous sequential ***processes***
  - Execute (logically) in parallel
  - Each process has single ***thread of control***
- Alternatives for ***implementation*** (i.e. execution) of a collection of processes :
  - ***Multiprogramming***
  - ***Multiprocessing***
  - ***Distributed Processing***

- Multiprogramming: Processes multiplex their executions on a single processor
- Multiprocessing: Processes multiplex their executions on a multiprocessor system where there is access to shared memory
- Distributed Processing: Processes multiplex their executions on several processors which do not share memory

Doing Multiple Things “at the Same Time“

1. Use one process
   - ***Cyclic executive***
   - Equivalent to ***sequential programming***
2. Use signals to emulate multiple processes
   - A signal handler similar to an independent, asynchronous flow of control – ***but not quite***

Doing Multiple Things “at the Same Time“

3. Use many processes

- Dedicated process for each activity
- Have to ***coordinate*** processes
- Potential problems with ***performance***, ***scalability***

4. Use not quite so many processes

- Careful combination of activities into processes
- May ***simplify*** programming effort

5. Use threads

- In POSIX.4a: ***pthreads***

The Cyclic Executive

- Traditional programming languages (***Pascal***, ***C***, ***Fortran***, ***Cobol***) are sequential
- Emulate concurrency by ***cyclic execution*** of a program sequence to handle the various concurrent activities
- ***Example***: Video game controller

```
while (1) {
    /* Read keyboard */
    /* Recompute player positions */
    /* Update the display */
}
```

- May work for small RT applications, with things happening in synch

•It is up to the programmer to construct his/her system so that it involves the cyclic execution of a program sequence to handle the various concurrent tasks.

•This complicates the programmer's already difficult task and involves him/her in considerations of structures which are irrelevant to the control of the tasks in hand;

•The resulting programs will be more obscure and inelegant;

•It makes proving program correctness more difficult;

•It makes decomposition of the problem more complex;

•Parallel execution of the program on more than one processor will be much more difficult to achieve;

•The placement of code to deal with faults is more problematic.

The Cyclic Executive

- Loop is infinite while
  - Spins as fast as possible
  - May consume more resources than needed (busy wait)
    - May *increase power consumption*
    - May be unacceptable in shared environment
- Loop has to run fast enough to service the input with highest frequency
  - May be difficult to achieve
  - Assumes that tasks all have *harmonic* frequencies
- May also periodically poll for other work from within a long computation
  - The classical Macintosh programming model

The Cyclic Executive – Disadvantages

- Resulting programs ***more obscure*** and inelegant
- ***Decomposition*** of the problem becomes more complex
- Parallel execution of the program on more than one processor difficult
- Placement of code to deal with ***faults*** is more problematic

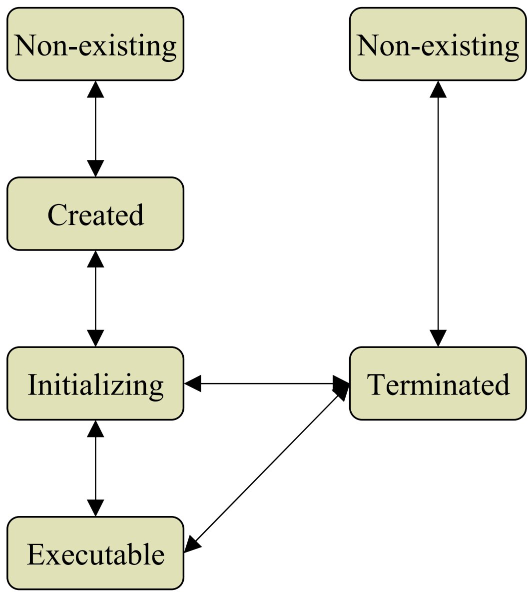
Emulating Multitasking with Signal Handler

- A POSIX signal is a SW analogue to a HW ***interrupt***
- However, signal handler cannot synchronize execution with any of the other signal-simulated tasks – ***there really is just one task !***
- If the signal handler blocks:
  - Entire program blocks
  - Results in a ***hung application***
- ***More on signals later***

•When a process receives a signal:
- Switch control to signal handler

•Upon completion of the signal handler:
- Return control to where the signal occurred
- Is similar to exceptions with resumption

Process States



- A process may
  - never terminate
  - fail during initialization
- Executable processes may not execute due to lack of resources (e.g., the CPU)

The Run-Time Support System

- RTSS similar to ***scheduler*** in an operating system
- RTSS is logically between hardware and application software
- Scheduling algorithm of RTSS affects ***temporal behavior*** of the SW
- For well-constructed programs, the ***logical behavior*** should not depend on the RTSS

RTSS Implementations

An RTSS may be implemented as

- Software structure ***programmed*** as part of the application (approach of ***Modula-2***)
- Standard software system generated with the program object code by the ***compiler***
  - Typical for ***Ada*** and ***Java*** and co-design systems such as ***MetroPOLIS***
- ***Hardware structure*** microcoded into the processor for efficiency
  - E.g., an ***occam2*** program running on the transputer has such a run-time system

Processes and Threads

- All operating systems provide ***processes***
- Processes execute in their own ***virtual machine*** (VM)
  - avoids interference from other processes
- Recent OSs provide mechanisms for creating ***threads***:
  - co-exist within the same VM
  - have unrestricted access to their VM
- The programmer and the language must provide the protection from interference
- Threads may be provided ***transparently*** to the OS
  - ***Example***: ***Windows 2000***
    - ***Threads*** are visible to the kernels
    - ***Fibers*** are invisible

Concurrent Programming Constructs

- Fundamental facilities:
  - Expression of ***concurrent execution*** through the notion of process
  - Process ***synchronization***
  - Inter-process ***communication***
- Processes may be
  - independent
  - cooperating
  - competing

Properties of Processes

- Process ***structure***
  - ***Static***: fixed and know at compile time
  - ***Dynamic***: created at run time

- Process ***level***
  - ***Nested***: processes can be defined at any level
  - ***Flat***: processes defined only at outermost level

Language	Structure	Level
Concurrent Pascal	static	flat
occam2	static	nested
Modula 1/2	dynamic	flat
C/POSIX	dynamic	flat
Ada	dynamic	nested
Java	dynamic	nested

Properties of Processes

- Process ***granularity***
  - ***Coarse*** (***Ada***, ***POSIX*** processes/threads, ***Java***)
  - ***Fine*** (***occam2***)
- Process ***initialization***
  - ***Parameter passing*** – at process creation
  - ***IPC*** – after a proces has started executing

Processes Termination

Processes can terminate in a number of ways:

- ***Completion*** of execution of the process body
- ***Suicide***, by execution of a ***self-terminate*** statement
- ***Abortion***, through the explicit action of another process
- Occurrence of an ***untrapped error condition***
- *Never*: processes are assumed to be non-terminating loops

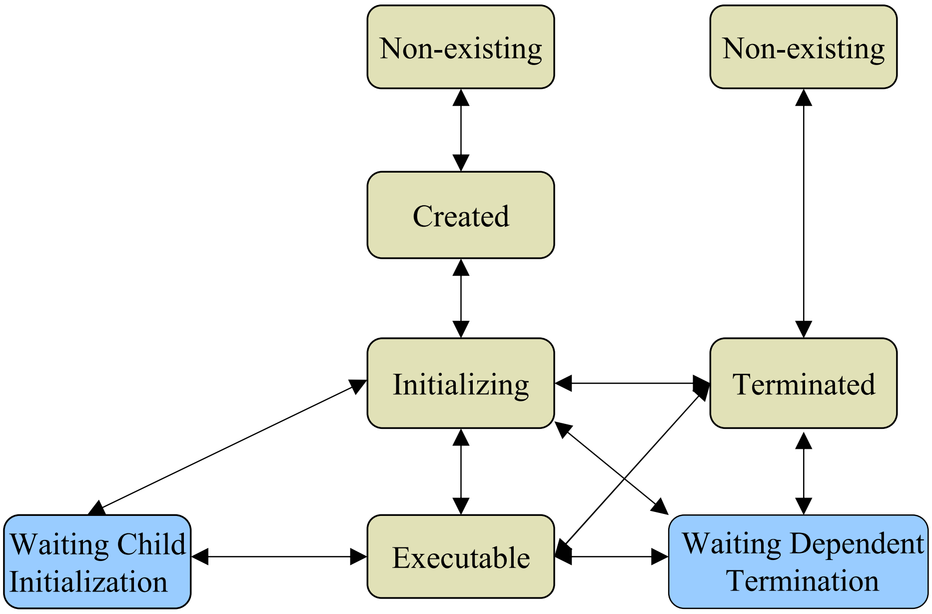
Nested Processes

- ***Hierarchies*** of processes can be created and inter-process relationships formed
- ***Parent/child*** relationship:
  - ***Parent*** process (or block) creates ***child***
  - Parent process may be delayed while child is created and initialized
- Guardian/dependent relationship:
  - ***Dependent*** process (or block) terminates if its ***guardian*** (or ***master***) terminates
  - Dependent process may depend on
    - Guardian process itself or
    - An inner block of the guardian

Nested Processes

- ***Guardian is not allowed to exit from a block*** until all dependent processes of that block have terminated
  - A process cannot exist outside of its scope
- ***Guardian cannot terminate*** until all its dependents have terminated
- ***A program cannot terminate*** until all its processes have terminated
- A parent of a process may also be its guardian
  - E.g. with languages that allow only static process structures – such as ***occam2***
- With dynamic nested process structures, the parent and the guardian may or may not be identical (***Ada***)

Process States



Processes and Objects

- *Active* objects
  - Undertake spontaneous actions (***active agents***)
- *Reactive* objects
  - Only perform actions when invoked (***passive data***)
  - ***Resources***
    - Can control order of actions and access to internal states
    - May for example be used by only one agent at a time
    - Accessability may also depend on internal state
  - *Passive* objects
    - No control over order

Resources

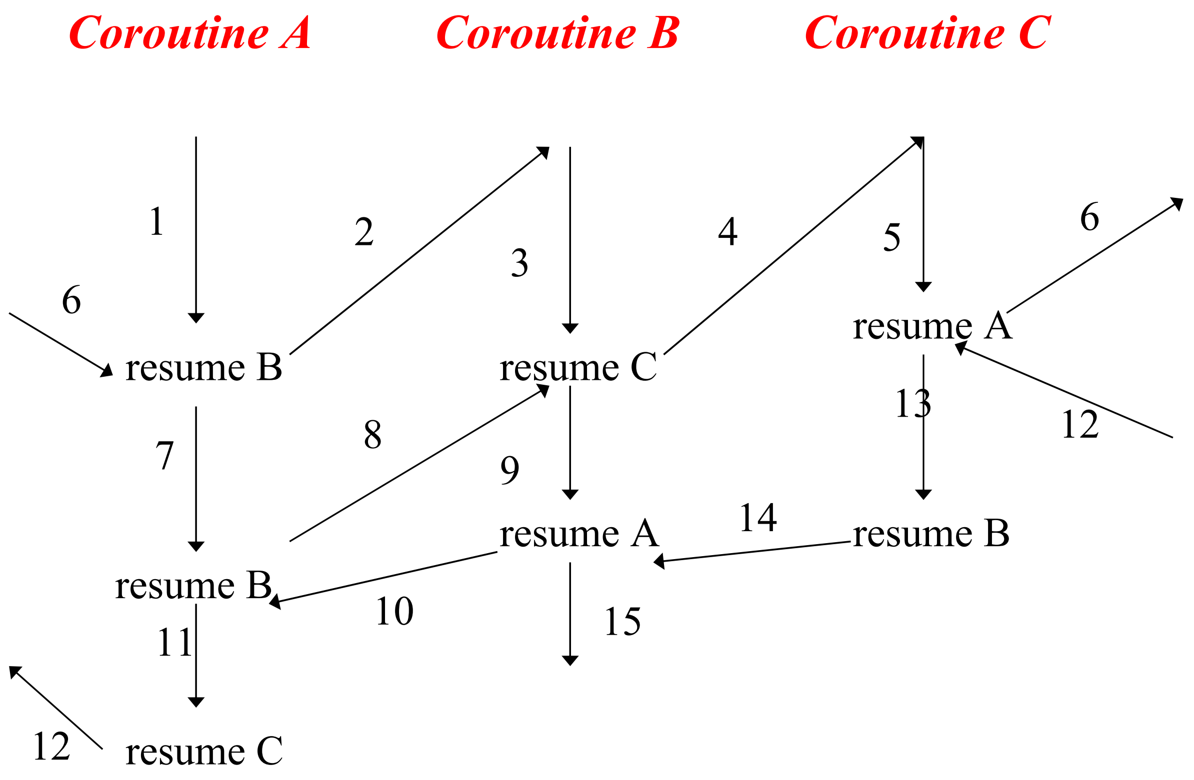
- *Implementation of resources requires control agent*
- ***Protected*** (or ***synchronized***) resources
  - ***Passive resource controller***
  - ***Pro***: efficiency
  - ***Con***: inflexibility
- ***Server***
  - ***Active resource controller***
  - ***Pro***: flexibility
  - ***Con***: may lead to proliferation of processes
- ***Ada***, ***Java***, ***POSIX*** support all
- ***Occam2*** supports only servers

Process Representation

There are different basic mechanisms for representing concurrent execution:

- Coroutines
- Fork and Join
- Cobegin
- Explicit Process Declaration

Coroutine Flow Control



Coroutines

- Coroutines are similar to subroutines
- *However:*
  - Control is passed in ***symmetric*** rather than in hierarchical way
  - Control is passed with ***resume*** statement
  - There is ***no return*** statement
  - The value of the data local to the coroutine persist between successive calls
  - The execution of a coroutine is supended as control leaves it, to carry on where it left off when it resumed
- ***Example***: ***Modula 2***

So … do coroutines express true parallelism?

Fork and Join

- ***Fork***:
  - Starts a designated routine, concurrently with the invoker
- ***Join*** (POSIX: **`wait`**):
  - Invoker waits for the completion of the invoked routine
- Can be found in ***Mesa*** and ***POSIX***

```
function F return is ...;
procedure P;
    ...
    C:= fork F;
    ...
    J:= join C;
...
end P;
```


•After the fork, P and F will be executing concurrently.

•At the point of the join, P will wait until F has finished (if it has not already done so

Fork and Join

- … allow dynamic process creation and the passing of ***parameters*** to child processes
- Usually child returns single value upon termination
- ***Pro***: ***flexible***
- ***Con***: ***unstructured***
  - ***F***or example, guardian must explicitly rejoin dependants
- ***Example***: ***UNIX***

```
for (I=0; I!=10; I++) {
  pid[I] = fork();
}
wait . . .
```

How many processes were created?

Cobegin

- ***Cobegin*** (or ***parbegin*** or ***par***):
  - A ***structured*** way of denoting the concurrent execution of a collection of statements

```
cobegin
   S1;
   S2;
   S3;
   .
   .
   Sn
coend
```

- Can be found in ***Edison*** and ***occam2***

- S1, S2 etc, execute concurrently
- The statement terminates when S1, S2 etc have terminated
- Each Si may be any statement allowed within the language
  - This includes cobegin (nesting)

Explicit Process Declaration

- To clarify program structure:
  - Routines state whether they will be executed concurrently
  - This does not say ***when*** they will execute!
- Process or task creation may be
  - *Explicit*
  - *Implicit*

```
task body Process is
begin
  . . .
end;
```

Summary I

- The application domains of most real-time systems are ***inherently parallel***
- The inclusion of the notion of process within a real-time programming language makes an enormous difference to its ***expressive power*** and ***ease of use***
- Without concurrency the software must be constructed as a ***single control loop***
  - The structure of this loop cannot retain the logical distinction between systems components
  - It is particularly difficult to give process-oriented timing and reliability requirements without the notion of a process being visible in the code

Summary II

- The use of a concurrent programming language requires a ***run-time support system*** to manage process execution
- Processes have several ***states***:
  - non-existing
  - created
  - initialized
  - executable
  - waiting dependent termination
  - waiting child initialization
  - terminated

Summary III

A process model is characterized by its:

- ***Structure***
  - static
  - dynamic
- ***Level***
  - top level processes only (flat)
  - multilevel (nested)
- ***Initialization***
  - with or without parameter passing
- ***Granularity***
  - fine grain
  - coarse grain

Summary IV

A process model is further characterized by its:

- ***Termination***
  - Natural
  - Suicide
  - Abortion
  - Untrapped error
  - Never
- ***Representation***
  - Coroutines
  - Fork/join
  - Cobegin
  - Explicit process declarations

To Go Further

- Chapter 7 of [Burns and Wellings 2001]
- Chapter 3 of Gallmeister, POSIX.4: Programming for the Real World, O'Really, 1995