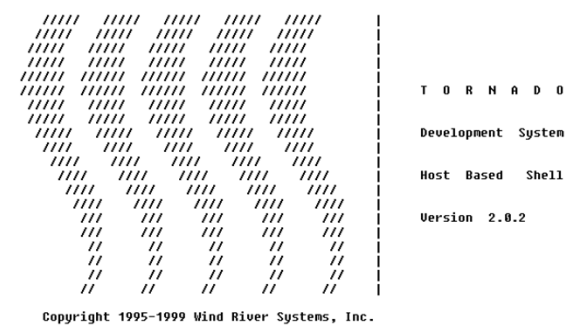
UNIVERSITY OF OSLO

FYS 4220 – 2011 / #2

Real Time and Embedded Data Systems and Computing

Concurrency and concurrent systems



T.B. Skaali, Department of Physics, University of Oslo)

Concurrency

- Computer science defines concurrency as a property of systems where several processes are executing at the same time, and may or may not interact with each other.
  - The above definition are neither very precise nor complete!
    - What does ”at the same time” mean?
      - In a single processor it is obvious that only one process can execute at a time. This means that concurrent activities must be given time slices such that it seems that they are running in parallell.
    - What is meant by ”interact with each other” ?
      - Will be covered in later lectures – interprocess communication
    - What is the relation between Real Time and Concurrent systems?
    - What is the environment of a concurrent system: processors, communication, etc?
    - How are common resources distributed?
      - A classic: http://en.wikipedia.org/wiki/Dining_philosophers_problem

True Concurrency vs. Pseudo-Concurrency

- In a single processor, concurrent external activities must be ”mapped” as pseudo-concurrent sequential processes.The timing requirements is met when all the sequential proceesses can react within the given deadlines. This may be difficult to prove for hard Real-Time systems.
- True concurrency requires parallell processing in separate processors, either a multi-processor system or multi-CPUs implemented as soft cores in a FPGA.
- Whatever the solution, process-process interaction (communication) greatly complicates the implementation and analysis of a concurrent system.

UNIVERSITY OF OSLO

Real Time - Concurrent systems

- Definition of a Real Time system, ref. lecture #1:
  - *A Real-Time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period, and the correctness depends not only on the logical result but also the time it was delivered*
- A dedicated (embedded) system using a micro-controller to read out data at a fixed rate or from interrupts is a Real Time system, but is not a concurrent system if all processing is done within the same process
- Vice versa: a concurrent system where the correctness does not really depend on timing constraints, is not a Real Time system
- However, in general a Real Time system is a concurrent system, where each Real Time activitiy is mapped into a process (or a thread, or a ”task” in VxWorks dialect)

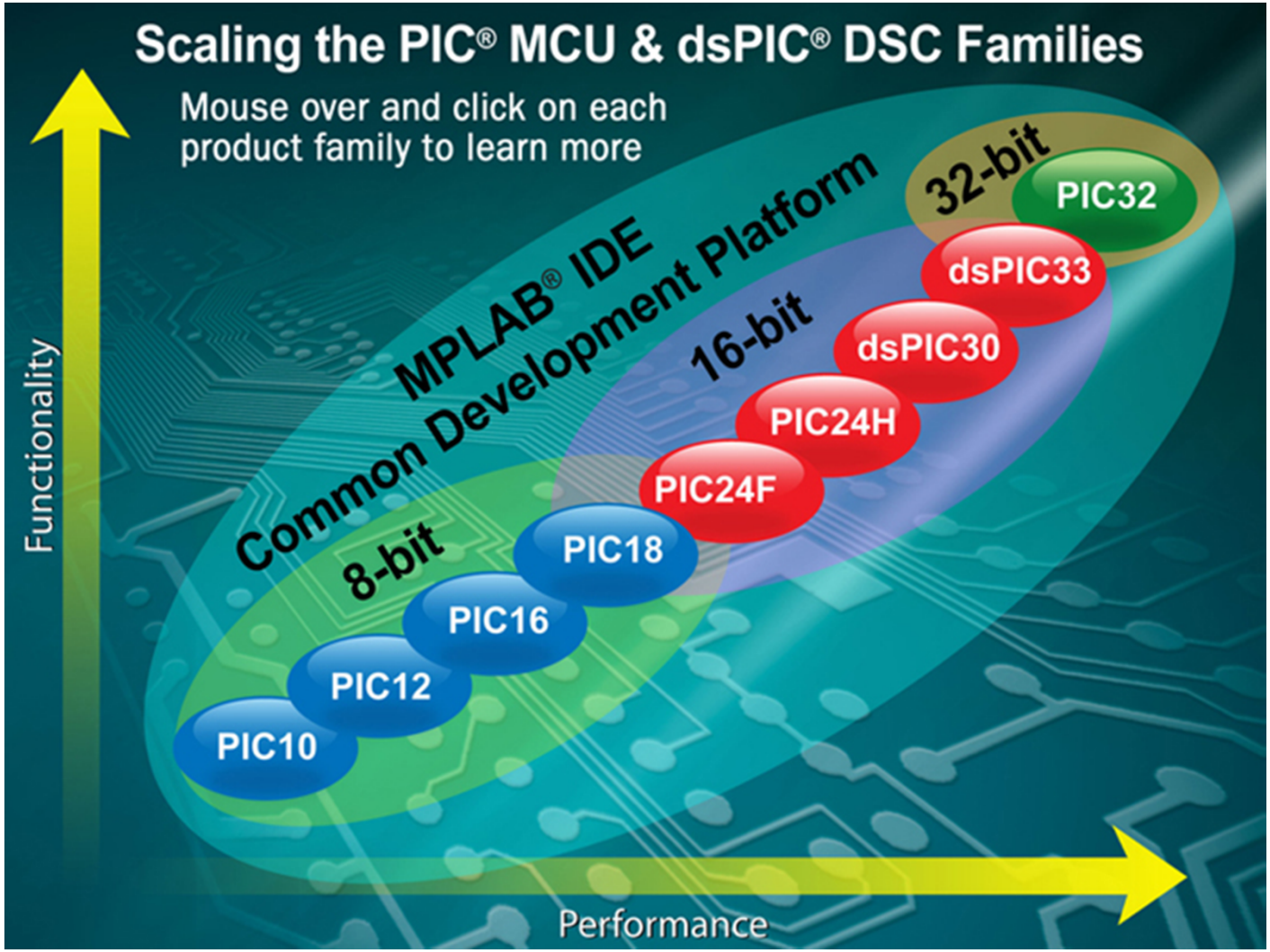
Concurrent programming

- The name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems
- For Real-Time systems the physical activities are mapped into a number of separately executing programs, processes
  - Note the distinction between the program code and the execution of this code within the context of a process. The same code can be executed by several processes, operating on separate data.
- All Real-Time systems are inherently concurrent — physical devices operate in parallel in the real world
- So, how are the synchronization and communication problems solved?
  - Note that these problems are basically the same whether one implements the processes on a single computer, or if the activities are spread on separate processors!

The simplest system – a single process

- Some measuring/control tasks may be so trival that a single execution thread solution is sufficient
- Typically this is a case where a measurement of a single value is done at regular intervals
- The time of the next measurement is then determined by polling the value of a timer, eventually that the thread is woken up by a timer interrupt
- A typical implementation is to use a microcontroller, they are available from 4-bits architecture and upwards
- A well known product is the PIC™ microcontroller family from Microchip *http://www.microchip.com*

PICs are popular with both industrial developers and hobbyists alike due to their low cost, wide availability, large user base, extensive collection of application notes, availability of low cost or free development tools, and serial programming (and re-programming with flash memory) capability. Microchip announced on February 2008 the shipment of its six billionth PIC processor.



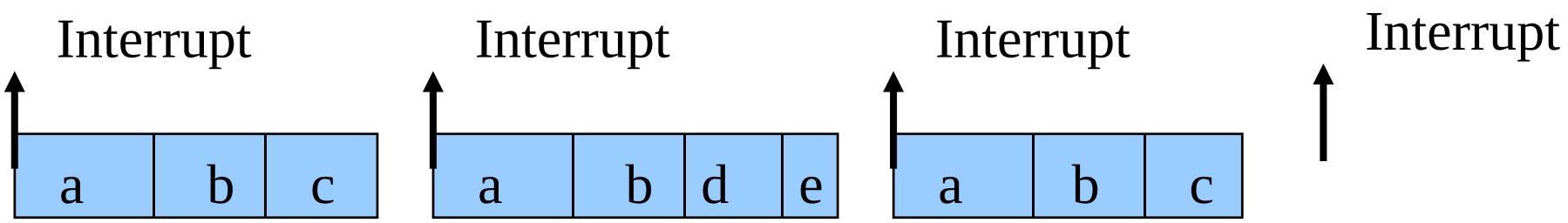
An simple alternative to concurrent programming?

- An alternative is to use sequential programming techniques
- The programmer must construct the system so that it involves the cyclic execution of a program sequence to handle the various concurrent activities, see principle on next two pages [Ref. 2]
- This complicates the programmer's already difficult task and involves him/her in considerations of structures which are irrelevant to the control of the activities in hand
- The resulting programs will be more obscure and inelegant
- 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
- However, for a small dedicated embedded system this approach may be the simplest one!

Cyclic Executive

```
loop
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
  procedure_for_e;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_c;
  wait_for_interrupt;
  procedure_for_a; procedure_for_b; procedure_for_d;
end loop;
```

Time-line for Process Set



Concurrent programming

- Real-Time / embedded software systems may be anything from a few thousand to millions of lines of code
- Languages used: numerous, a few examples:
  - CORAL 66 (Computer On-line Real-time Applications Language), from 1964, based on Algol 60
  - RTL/2: 1972, based on Algol 68
  - JOVIAL (Jules Own Version of the International Algorithmic Language) : similiar to Algol, 1959, devloped for military aircraft electronics, still in use
  - Then, there is C and C++, probably the most popular languages today
- However, these languages are not Real-Time systems *as such*, they depend on support from an operating system!

Concurrent languages

- High level concurrent languages, requires no Operating System support!
- Ada, CHILL, Modula-2, Real-Time Java
  - **Ada** is a structured, statically typed, imperative, and object-oriented high-level computer programming language. It was originally designed by a team led by Jean Ichbiah of CII Honeywell Bull under contract to the United States Department of Defense during 1977–1983 to supersede the hundreds of programming languages then used by the DOD. Ada addresses some of the same tasks as C or C++, but Ada is strongly-typed (even for integer-range), and compilers are validated for reliability in mission-critical applications, such as avionic software. Ada was named after Ada Lovelace, who is often credited with being the first computer programmer. Ada is an international standard; the current version (known as Ada 2005) is defined by joint ISO/ANSI standard (ISO-8652:1995), combined with major Amendment ISO/IEC 8652:1995/Amd 1:2007. (*ref. Wikipedia*)

Ada ”Hello World”

- File: hello_world_1.adb

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Hello is
begin
        Put_Line("Hello, world!");
end Hello;
```

The **with** statement adds the package Ada.Text_IO to the program. This package comes with every Ada compiler and contains all functionality needed for textual Input/Output. The **with** statement makes the declarations of Ada.Text_IO available to procedure Hello. This includes all types of Ada.Text_IO, the subprograms of Ada.Text_IO and everything else that is declared in Ada.Text_IO for public use. In Ada, packages can be used as toolboxes. Text_IO provides a collection of tools for textual input and output in one easy-to-access module. Here is a partial glimpse at package Ada.Text_IO

```
package Ada.Text_IO is
        type File_Type is limited private;
        -- more stuff
        procedure Open(File : in out File_Type;
                       Mode : File Mode;
                       Name : String;
                       Form : String := "");
        -- more stuff
        procedure Put_Line (Item : String);
        -- more stuff
end Ada.Text_IO;
```

(ref. Wikipedia)

UNIVERSITY OF OSLO

Real-Time Java

- Java executes in a virtual machine: it is platform independent
- Java and Real-Time Java:
  http://en.wikipedia.org/wiki/Real-Time_Java
  http://java.sun.com/developer/technicalArticles/Interviews/Bollella_qa2.html

- **17.1.1 Hello.java (ref. Forelesning H. Haugerud, Ifi; 19/3-07)**

```
$ cat Hello.java
class Hello
{
        public static void main(String args[])
        {
                System.out.println("Java: Hello world!");
        }
}
```

Some terminology

- A concurrent program is a collection of autonomous sequential processes, executing (logically) in parallel
- Each process has a single thread of control
- The actual implementation (i.e. execution) of a collection of processes usually takes one of three forms.

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

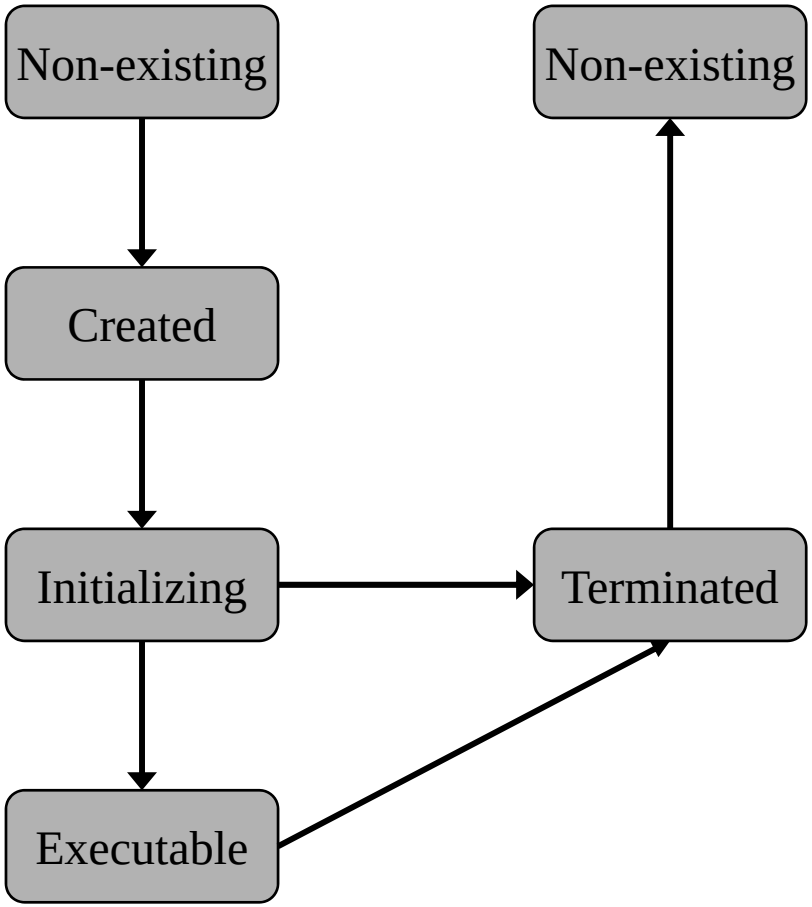
  – processes multiplex their executions on several processors which do not share memory

(ref. B&W)

Operating system environment

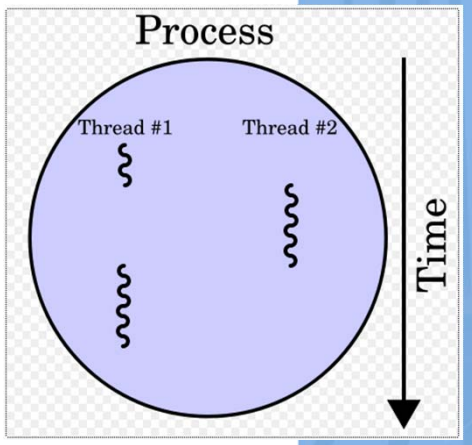
- All (real) operating systems (OS) provide processes
- A key issue is to avoid destructive interference between processes
- The OS therefore implements an environment which can be called a virtual machine. A user process can not directly access other processes, and neither can access memory space outside its own space if a memory protection scheme is implemented
  - Memory protection requires hardware support by the processor
- However, in Real-Time applications it may be useful (however potentially dangerous but quite interesting…) that a process can access any part of the physical memory. Such a system is often called a ”flat memory”. Used by VxWorks.

Process states – a simple model



Processes vs. threads

- A key characteristic for Real-Time OS is the context switching time, i.e. the time needed for freezing one process and starting another one
  - However, a process context switch is relatively costly in time, because much information needs to to stored when a process is freezed, and the same amount of information is to be retrieved when before another process can be started. This is especially true in a memory protected environment
- Simply said: a *thread* is a process within a process. As such it has access to the same memory space, and the context switching from one thread to another will be shorter than for process to process



Multithreading

- Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process' resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a *single* process to enable *parallel execution* on a *multiprocessor* system.
- This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines. This is because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require atomic operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

(ref. Wikipedia)

Execution of Real Time processes – how?

- A process can be executed, or resumed after a wait, by:
  - User/operator command
  - Data **interrupt**, for instance from the Analog-to-Digital converter
  - By a clock (timer), typical for a **periodic** execution
    - Timer interrupt
    - On a given date and time
  - By a **signal** or **message** from another process
    - In this case one talks about a **parent-child** relationship. A child can in due course start up a new child, and so on.
- If several processes want to execute at the same time, a mechanism is required to select which process shall get the CPU
  - The situation is somewhat blurred in a multi-cpu processor, there one can of course have truly parallell execution, provided that all other resources are available. To simplify the discussion, let us consider single CPU systems for the time being

Concurrent systems environment

- In order that processes are executed truly simultaneous, they must run on separate processors
  - In most cases this is not feasible because it may imply a very large number of processors, and interprocess communication may become a bottleneck in itself
- An appearance of concurrency may be implemented by interleaving separate processes on a single processor with a multi-tasking OS
  - The requirement is that the system has sufficient resources for that the processes can be executed according to their timing specifications
    - But, how can one know that?
  - Since the processes can interact with each other while executing, the number of possible execution paths in the system can be extremely large, and the resulting behavior can be very complex.

Difficulties with concurrency – ”a can of worms”

- Policy for allocation of shared resources between processes of different priorities:
  - CPU
  - Memory
  - Input/output devices
- Race condition may result in unpredictable behavour
- Mutual exclusion can prevent race conditions, but may in turn lead to:
  - Deadlock
    - The processes are locked in a passive state. Two processes compete for two resources, and after each of them have reserved one of the resources they discover that the other resource is taken!
  - Starvation
    - The processes execute without advancing, for instance by endless testing on a flag

A VxWorks example - I

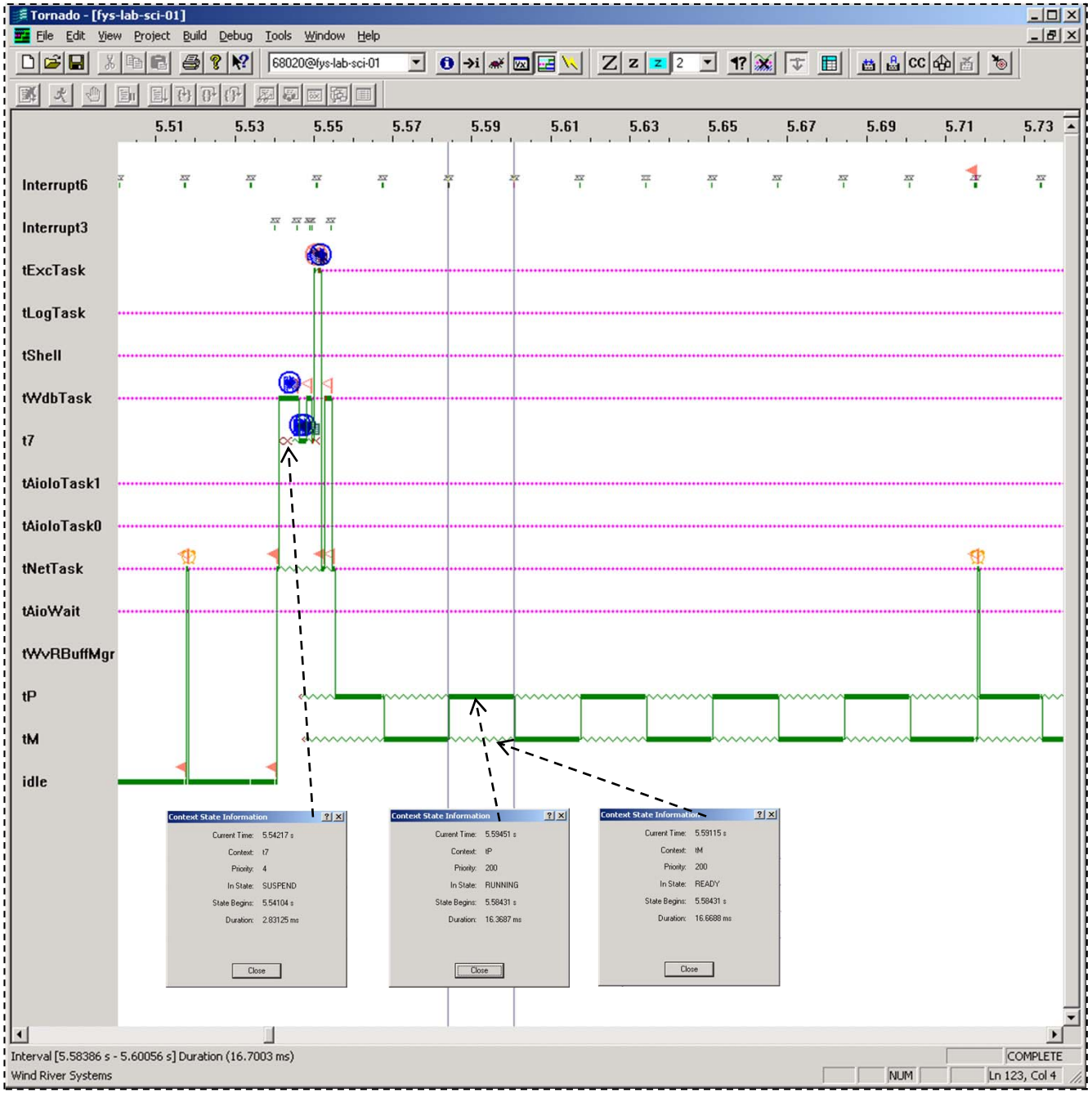
- VxWorks case: three processes are started from routine *Alfred*, using the same code of routine *controller()*

```
(void) controller  (par)
{
--- do something according to ”par”
}

(void) Alfred()
{
…..
/* ”P1”  task name, pr1 priority, stack space, (FUNCTR)<start_address>, par   */
taskSpawn (”P1", pri1, 0, stack1, (FUNCPTR)controller, par1,…);
taskSpawn (”P2", pri2, 0, stack2, (FUNCPTR)controller, par2,…);
taskSpawn (”P3", pri3, 0, stack3, (FUNCPTR)controller, par3,…);
 …..
 }
```

A VxWorks example - II

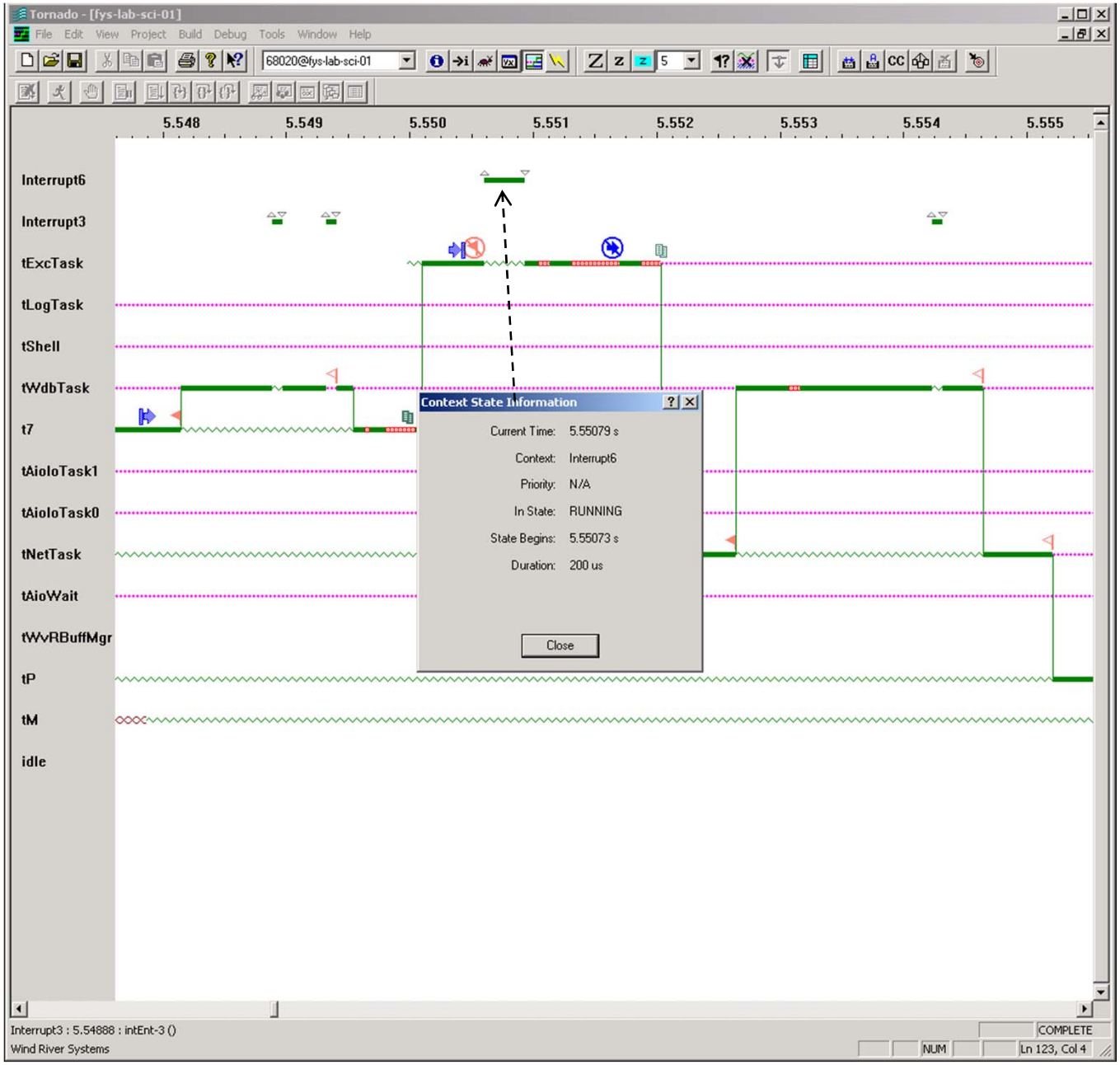
- Now, how are the concurrent sequential VxWork tasks P1, P2 and P3 executed? (In a non-Real-Time system one would not worry too much about that, just hoping that they could enevtuall carry out their jobs)
  - Assuming the very simplistic case with 1) no interaction between the tasks and 2) no resource competition except for the CPU, the execution will be determined by the scheduling algorithm, where the process priority is a key parameter
    - Scheduling will be covered in a later lecture
  - If 1) or 2) are not true then there is no clear-cut answer!
  - Below is part of a code that will be discussed in a coming lecture, ”plusx” and ”minusx” are program units. An excerpt of the processor activity is shown on next page

```
#define                 P_PRI       200
#define                 M_PRI       200

/* enable or disable Wind round-robin*/
    kernelTimeSlice (1);
    Pid  = taskSpawn ("tP",   P_PRI,  0, 1000, (FUNCPTR)plusx,     0,0,0,0,0,0,0,0,0,0);
    Mid = taskSpawn ("tM", M_PRI, 0, 1000, (FUNCPTR)minusx, 0,0,0,0,0,0,0,0,0,0);
}
```



UNIVERSITY OF OSLO

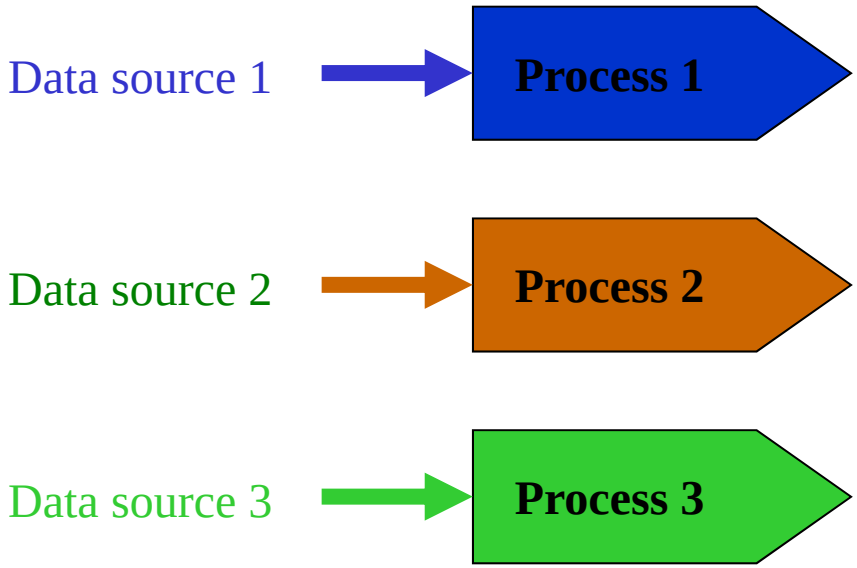
After tP and tM have been dispatched from process interactive process t7 through tExcTask they execute on a round-robin basis with time slice 16.67ms, which is the period of the Real-Time clock



Blown up part showing Interrupt processing and task execution. **Interrupt6** is Real-time 60Hz clock, **Interrupt3** is net operations, the host-target communication runs over Ethernet.

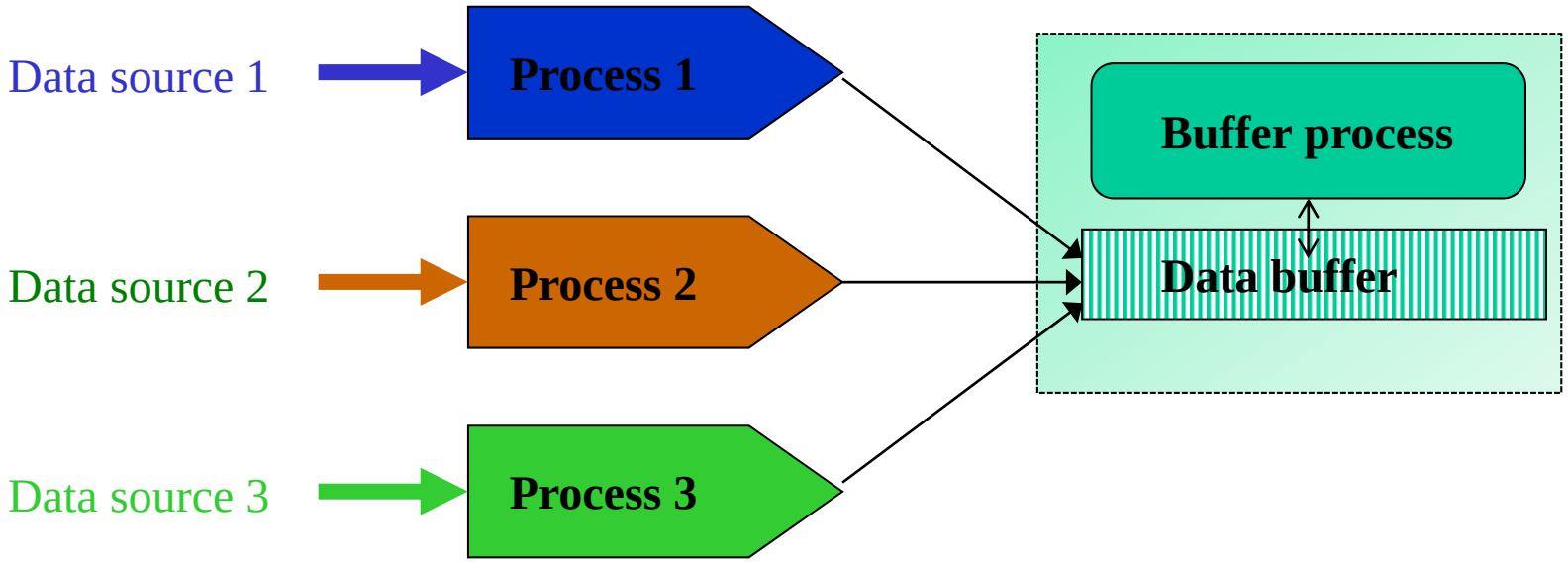
Case: multiple independent data streams

- The three data processing streams are independent, there is no communication between them.



Case: multiple interacting data streams

- The three data processing streams are coupled via the requirement of writing data to a common buffer. This is a very different problem: how is the access to the common buffer controlled, and how does the processes know that a data record has been properly received by the buffer process?

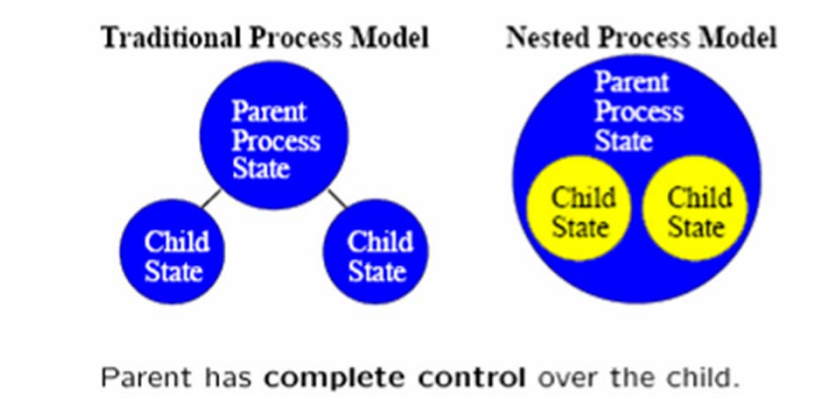


Concurrent programming constructs

- The mapping of concurrent execution through the process concept
- Process synchronization
- Interprocess communication
- Interaction of processes:
  - Independent
    - No communication with other processes
  - Cooperating
    - Synchronizing their activities through communication
  - Competing
    - Resource allocation

Concurrent execution

- Models of concurrencies:
  - **Structure**
    - **Static**: number of processes is fixed and known at Build time
    - **Dynamic**: processes are created (and deleted) at any time
  - **Level of parallelism**
    - **Flat**:processes are defined at the outmost level of the program text, as for C/POSIX
    - **Nested**: processes are allowed to be defined within other processes, as for Ada and Java



POSIX

- **POSIX** (or "Portable Operating System Interface“ is the collective name of a family of related standards specified by the IEEE to define the Application Programming Interface (API) for software compatible with variants of the Unix operating system. Originally, the name stood for IEEE Std 1003.1-1988. The family of POSIX standards is formally designated as **IEEE 1003** and the international standard name is ISO/IEC 9945. The standards emerged from a project that began in 1985. The term *POSIX* was suggested by Richard Stallman in response to an IEEE request for a memorable name!
- So, POSIX is not an OS, but an API! The API includes Real-Time Services, Threads interface, Real-Time extensions, etc

POSIX upgrade 1b: Real-Time extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynch and Synch I/O
- Memory Locking
- Some fully POSIX compliant OS: LynxOS (RTOS), MAC OS X, Windows XP and Vista
- Mostly compliant: Linux
- Some compliant: VxWorks

Process initialization

- When a process is created, it may need to be supplied with information required for its execution.
- Two ways of supplying this information are:
- 1) Pass the information in the form of parameters to the process, as in the VxWorks *taskSpawn(….)*
  - *Name*
  - *Entry point*
  - *Priority*
  - *Memory (stack) space*
  - *Application specific parameters*
- 2) Communicate with the process after it has started its execution

Process termination

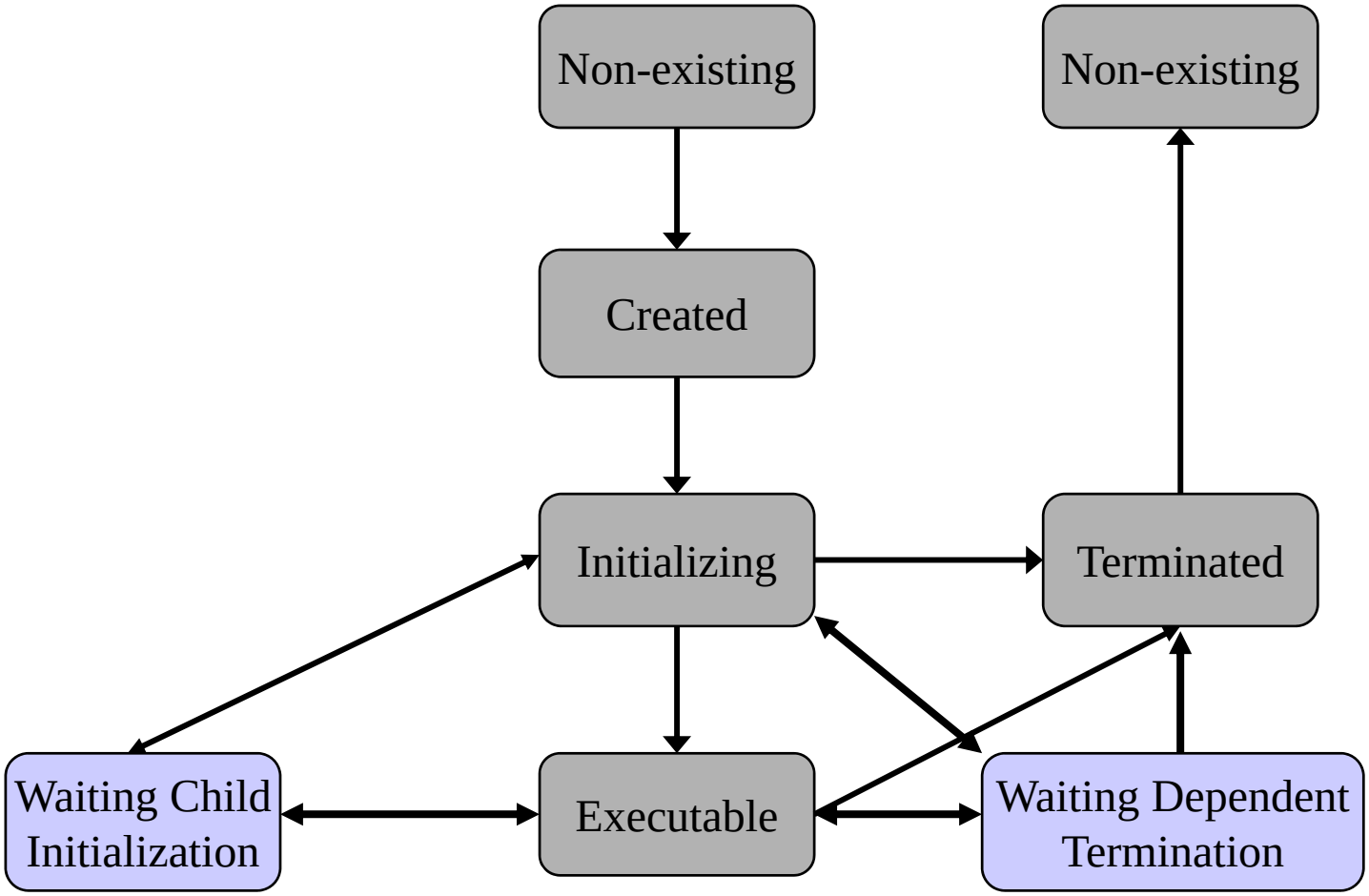
- The circumstances can be summarized as follows:
  - Completion of execution
  - Self-terminate statement
  - Abortion through the explicit action from another process
  - Occurrence of an error condition which is not trapped
  - When no longer needed
  - Or the process runs forever
- For process management, cf. VxWorks *taskLib*

Parent/child and nested processes

- These structures greatly complicates the rules of interaction between the processes:
  - Hierarchies of processes can be created and inter-process relationships formed
  - For any process, a distinction can be made between the process (or block) that created it and the process (or block) which is affected by its termination
  - The relationship is know as **parent/child** and has the attribute that the parent may be delayed while the child is being created and initialized
  - A parent program cannot terminate until all its processes have terminated
    - How can the parent know this? Follow a coming lecture!

UNIVERSITY OF OSLO

Process States extended



Fork and Join – the general concept

- The fork specifies that a designated routine should start executing concurrently with the invoker
- Join allows the invoker to wait for the completion of the invoked routine

```
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 the F has finished (if it has not already done so)
- Fork and join notation can be found in UNIX/POSIX

The UNIX system call fork()

- System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:
  - If **fork()** returns a negative value, the creation of a child process was unsuccessful.
  - **fork()** returns a zero to the newly created child process.
  - **fork()** returns a positive value, the ***process ID*** of the child process, to the parent
- UNIX will give an exact copy of the parent's address space and give to the child

Fork() example I – page 1 of 1

UNIVERSITY OF OSLO

Examples from ”UNIX Multiprocess Programming
http://www.csl.mtu.edu/cs4411/www/Home.html
http://www.csl.mtu.edu/cs4411/www/NOTES/process/process.html

```
/* ------------------------------------------------------------------ */
/* PROGRAM  fork-01.c                                                 */
/*    This program illustrates the use of fork() and getpid() system */
/* calls.  Note that write() is used instead of printf() since the    */
/* latter is buffered while the former is not.                        */
/* ------------------------------------------------------------------ */

#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

Fork() example II – page 1 of 2

UNIVERSITY OF OSLO

```
/* ------------------------------------------------------------------ */
/* PROGRAM fork-02.c                                                  */
/*   This program runs two processes, a parent and a child.  Both of */
/* them run the same loop printing some messages.  Note that printf()*/
/* is used in this program.                                           */
/* ------------------------------------------------------------------ */

#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}
```

Fork() example II – page 2 of 2

```
        pid = fork();
        if (pid == 0)
             ChildProcess();
        else
             ParentProcess();
}

void  ChildProcess(void)
{
        int   i;

        for (i = 1; i <= MAX_COUNT; i++)
             printf("   This line is from child, value = %d\n", i);
        printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
        int   i;

        for (i = 1; i <= MAX_COUNT; i++)
             printf("This line is from parent, value = %d\n", i);
        printf("*** Parent is done ***\n");
}
```

The POSIX fork() call

LYNUXWORKS™
Open. Reliable. Safe. Secure.

• CORPORATE • PRODUCTS • SUPPORT • HOME
• ALLIANCES • INDUSTRIES • TRAINING search go

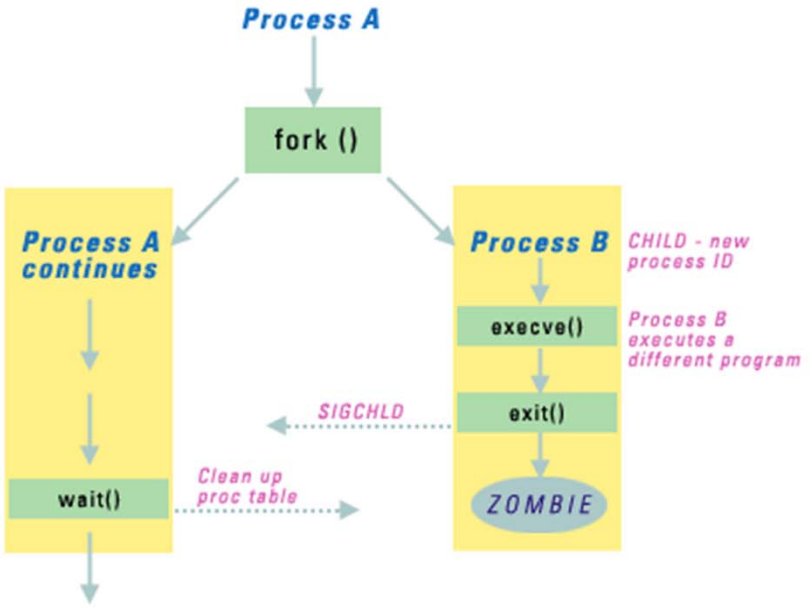
Processes and Parent-Child Relationships

The fork() call

NEXT: THREADS IN POSIX

To find out if an operating system might support the POSIX®.1 specification, ask "Does it support the fork() call?"

Fork() calls are scattered throughout Unix®/Linux® code. To support fork(), an operating system must first support the concept of parent-child relationships between processes. If fork() and other common POSIX.1 calls are not supported, the vast majority of your Unix/Linux code will not port to that operating system without major rewrites.



Understanding fork()

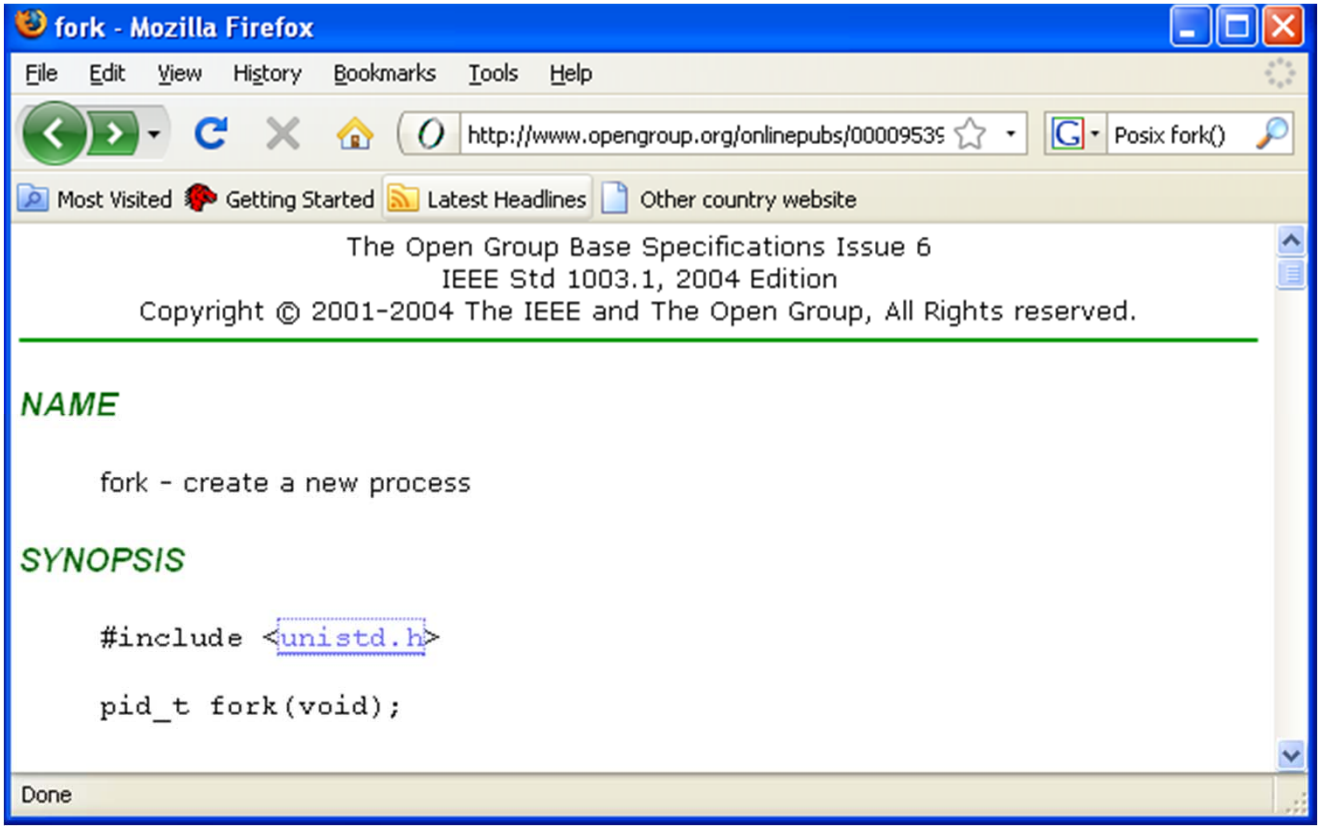
The fork() system call creates a new process called a child. The original process is called the parent, and the child is a near-exact copy of the parent. The child's run time is set to zero and file locks are not inherited.

The child has its own process ID and its own copy of the parent's file descriptors.

A parent process can recover the exit status of a child using the wait() or waitpid() function.

POSIX fork()

- IEEE Std 1003.1
  http://www.opengroup.org/onlinepubs/000095399/functions/fork.html



POSIX Threads Programming

- As a starter, let us have a look at the beginning of this tutorial from Lawrence Livermore National Laboratory https://computing.llnl.gov/tutorials/pthreads
  - A PDF version of the tutorial is on the FYS4220 web page
- An exercise with VxWorks pthreads will come later