Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Concurrency

Summer Term 2023 | 26.04.2023

Barry Linnert

Objectives of Today‘s Lecture

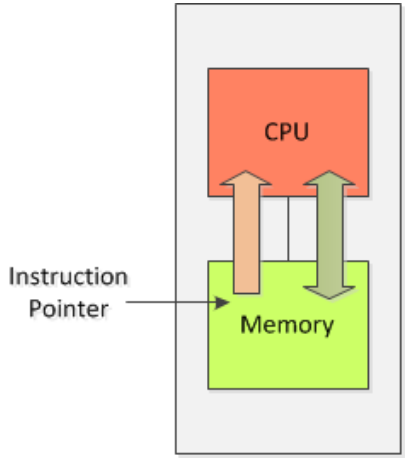
- Concurrency through processes
- Interrupts and process switching
- Concurrency and effects on determinism
- Atomic and non-atomic operations

Concepts of Non-Sequential and Distributed Programming

CONCURRENCY

Requirements for Programs

- A program should do what it is expected to do!
  - Functional requirements, such as
    - Scope of functions
    - Correctness



- A program should comply with certain requirements about its behavior.
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …

Performance

- There are (at least) two different perspectives to discuss the topic performance:

- Perspective of the service provider
  - use all resources to run as many programs as possible
  - maximum utilization of resources – especially CPU

- Perspective of the user
  - the fastest possible processing of your own program
  - short response time

What is the most important perspective?

Are there any dependencies between these two perspectives?

Limitations to Performance

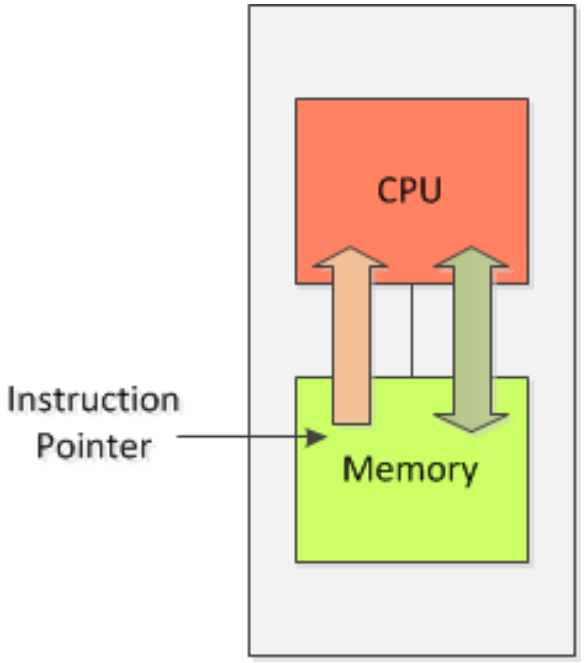
- Programs often use resources other than CPU or the main memory only.
- These other resources may have an impact on performance too.

- These other resources frequently consists of Input/Output devices.
- Thus I/O operations may have a significant impact on the performance since they are depending on
  - the access times to the input/output device
  - events that are introduced by the input/output devices
    - e.g. input by user

What is the perspective I/O does impact?

Repeat: Process

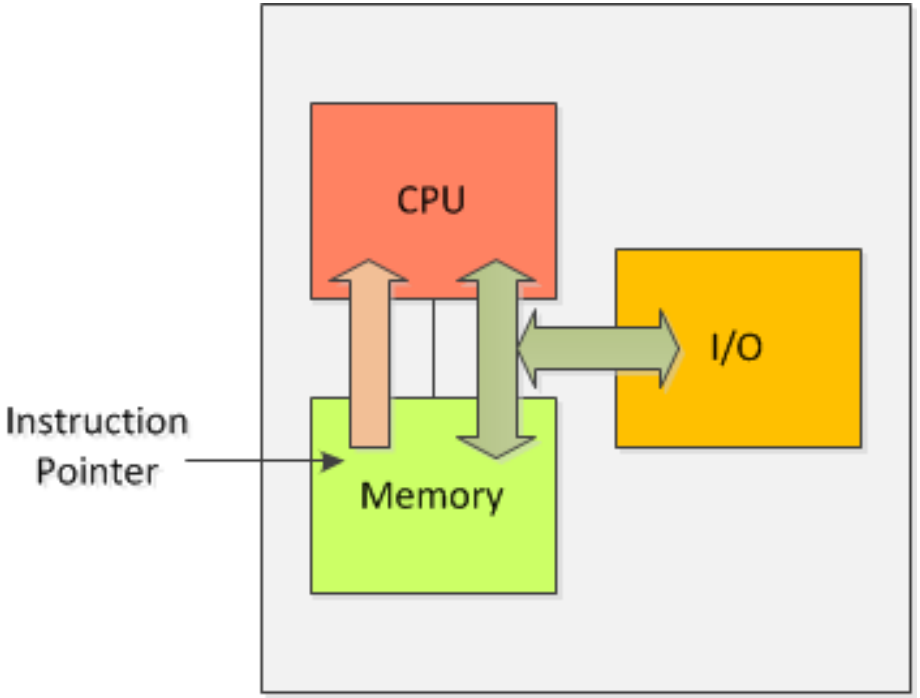
- The fact having our programming model corresponding with the machine model is ensured by the concept of the process.
- The operations of our program are translated to machine instruction manipulating the state of the system.
- The sequential execution of the machine instructions corresponds to the sequential execution of our operations programmed.



Machine Model

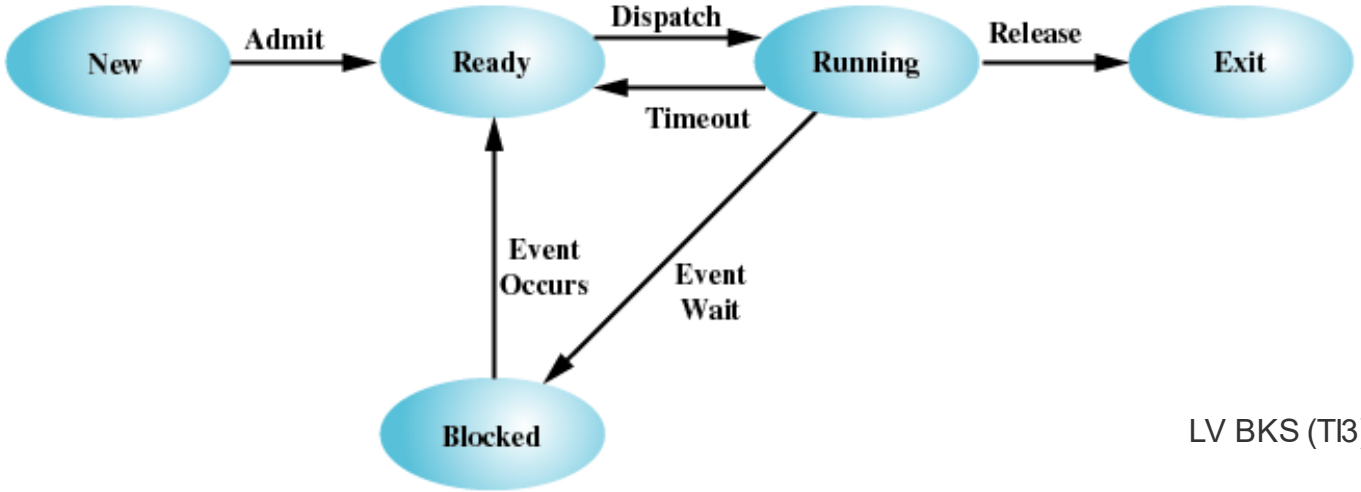
- With respect to performance of the whole system we have to consider the I/O operations, too.
- Our machine model needs to be extended.
- While waiting on response of the I/O device the process cannot run:

  The process is blocked.



Process States

- The process states describe how the system handles the process based on the process execution, i.e. I/O operation, and the system's properties and preferences.
- You discussed this in the lecture course "Betriebs- und Kommunikationssysteme" (BKS/TI3) already.

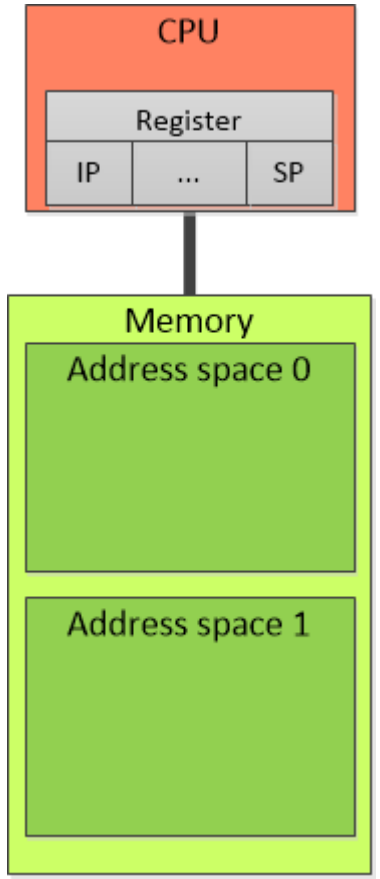


LV BKS (TI3)

Concurrency

- The introduction of concurrent process execution helps to reduce idle time of the CPU as another process can be executed, if the first process is blocked.

- Concurrency comes with some requirements to the system:
  - Separation of memory areas by introducing address spaces
  - Ability of process generation
    - via System call



Process Generation

```
// Program in C forking another process



int main(void) {



                                                    }
```

LV BKS

Process Generation


```
// Program in C forking another process
#include <stdlib.h>


int main(void) {

  pid_t pid;

  pid = fork();


}
```

LV BKS

Process Generation

```
// Program in C forking another process
#include <stdlib.h>

#include <stdio.h>

int main(void) {

  pid_t pid;

  pid = fork();

  if (pid == 0) {
    printf("Child process running.\n");

    // Do something...

    printf("Child process done.\n");

  }
```

```
  else if (pid > 0) {
    printf("Parent process, waiting for
      child %d...\n", pid);



    printf("Child process %d
      terminated,        %d.\n", pid,
                         );

  }
  else {
    printf("fork() failed\n");

  }
}
```

LV BKS

Process Generation

```c
// Program in C forking another process
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
  int status;
  pid_t pid;

  pid = fork();

  if (pid == 0) {
    printf("Child process running.\n");

    // Do something...

    printf("Child process done.\n");
    exit(123);
  }
  else if (pid > 0) {
    printf("Parent process, waiting for
      child %d...\n", pid);

    pid = wait(&status);

    printf("Child process %d
      terminated, status %d.\n", pid,
      WEXITSTATUS(status));

    exit(EXIT_SUCCESS);
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }
}
```

LV BKS

Process Generation

```c
// Program in C forking another process
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
  int status;
  pid_t pid;

  pid = fork();

  if (pid == 0) {
    printf("Child process running.\n");

    // Do something...

    printf("Child process done.\n");
    exit(123);
  }
  else if (pid > 0) {
    printf("Parent process, waiting for
      child %d...\n", pid);

    pid = wait(&status);

    printf("Child process %d
      terminated, status %d.\n", pid,
      WEXITSTATUS(status));

    exit(EXIT_SUCCESS);
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }
}
```

LV BKS

Process Generation with fork ()

- With `fork()` two identical processes are created.
  - `fork()` is a call to the operating system – system call
  - It's defined in the POSIX standard.
  - It creates a new address space (within the main memory) for the child process with a copy of the address space of the parent process.

- The processing of the process starts or continues after `fork()`.

Controlling Processes after Generation with fork ()

- Processes can be distinct by return value:
  - process ID (pid) of the child process ⇨ current process is parent process
  - process ID (pid) == 0 ⇨ current process is the child process

- End of the child process and feedback to parent process can be given with `exit(status).`

- Parent process can wait for the termination of the child process with `wait(&status)`

Example: two Programs running two Processes

```c
// first program in C

int main (void)
{
  int a = 0;
  int b = 0;

  a = 2;
  b = 3;

  a = a * b;

  return a;
}
```

```c
// second program in C

int main (void)
{
  int a = 0;
  int b = 0;

  a = 3;
  b = 4;

  a = a * b;

  return a;
}
```

Example: one Program runs two Processes

Freie Universität Berlin

```
// Program in C forking another process
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
  int status;
  pid_t pid;
  int a, b = 0;

  pid = fork();
  if (pid == 0) {
    // child process is calculating
    a = 2;
    b = 3;
    a = a * b;
    printf (" result of child
      process: %d \n", a);
    exit(123);
  }
  else if (pid > 0) {
    // parent process is calculating
    a = 3;
    b = 4;
    a = a * b;
    printf (" result of parent
      process: %d \n", a);

    pid = wait(&status);
    printf("Child process %d
      terminated, status %d.\n", pid,
      WEXITSTATUS(status));
    exit(EXIT_SUCCESS);
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }
}
```

Example: one Program runs two Processes

Freie Universität Berlin

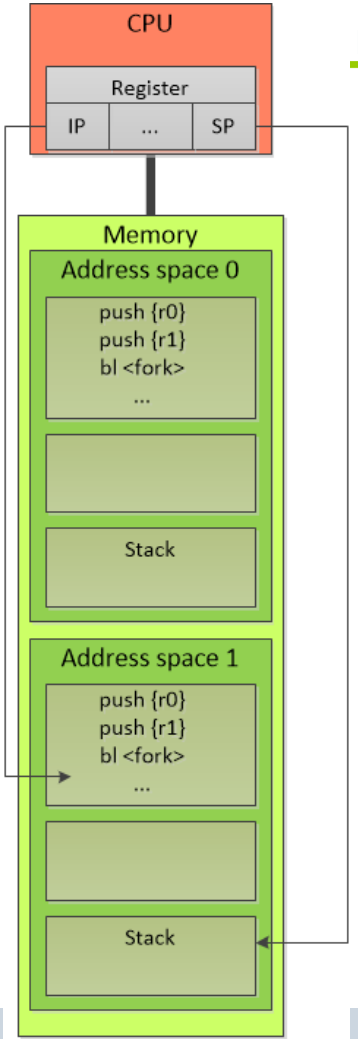
```
// Program in C forking another process
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
  int status;
  pid_t pid;
  int a, b = 0;

  pid = fork();
  if (pid == 0) {
    // child process is calculating
    a = 2;
    b = 3;
    a = a * b;
    printf (" result of child
      process: %d \n", a);
    exit(123);
  }
  else if (pid > 0) {
    // parent process is calculating
    a = 3;
    b = 4;
    a = a * b;
    printf (" result of parent
      process: %d \n", a);

    pid = wait(&status);
    printf("Child process %d
}
```

The two processes are programmed using the same variables.
Does the write operation executed by one process have any effect to the value of the variable of the other process?

Processes in Execution

- After their generation, both processes have their own address space in the main memory and can be executed one after another or concurrently.

- The operating system decides which of the process is executed first.



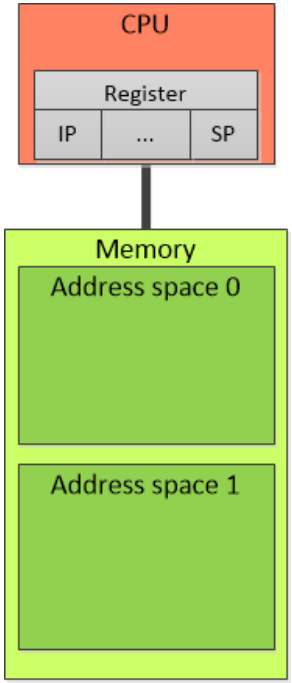
Separated Address Spaces

- Due to the fork system call all the memory areas of the parent process are copied containing data and program code.
- Thus parent and child process got their own copy of memory content. This is called address space and every process works the address space that it is mapped onto.
- The same address used by different processes are represented by different parts (addresses) of the main memory.
- Thus, a process cannot access the memory of another process, i.e., address space.
  - … until there is some kind of special mechanism; please see course “Betriebssysteme”
- Separation of address spaces makes is easy to implement a multitude of processes.

- Please note, the separation of address spaces is crucial for **security**!

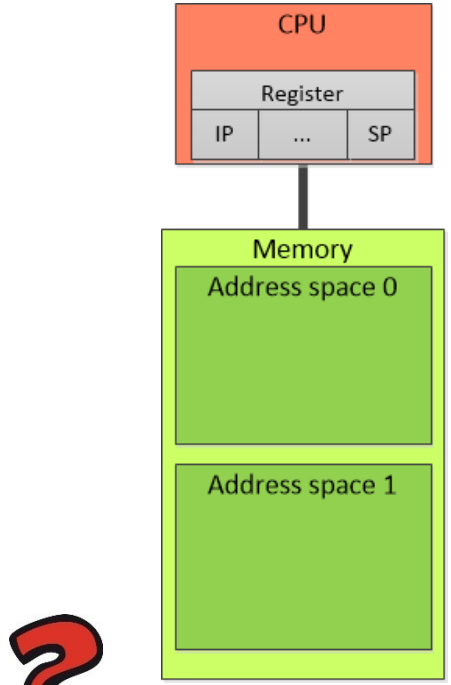
Concurrency

- The introduction of concurrent process execution helps to reduce idle time of the CPU as another process can be executed, if the first process is blocked.
- Concurrency comes with some requirements to the system:
  - Separation of memory areas by introducing address spaces
  - Ability of process generation
    - via System call



Concurrency

- The introduction of concurrent process execution helps to reduce idle time of the CPU as another process can be executed, if the first process is blocked.
- Concurrency comes with some requirements to the system:
  - Separation of memory areas by introducing address spaces
  - Ability of process generation
    - via System call
    - A superordinate (parent) process takes over the process generation and a process hierarchy can be created.
  - The change between processes (of different programs) has to be performed.

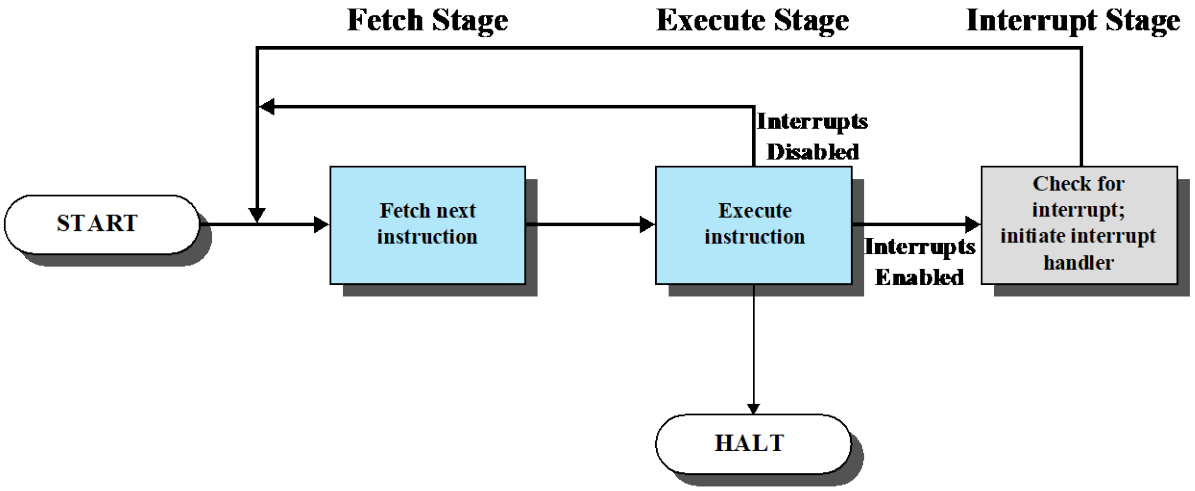


Concurrency (*cont.*)

- The mechanism to change between processes (of different programs) has to be invoked when
  - the process performing an I/O operation is blocked,
  - the I/O operation is finished and the process should run again to receive the result of the operation and release the I/O device.

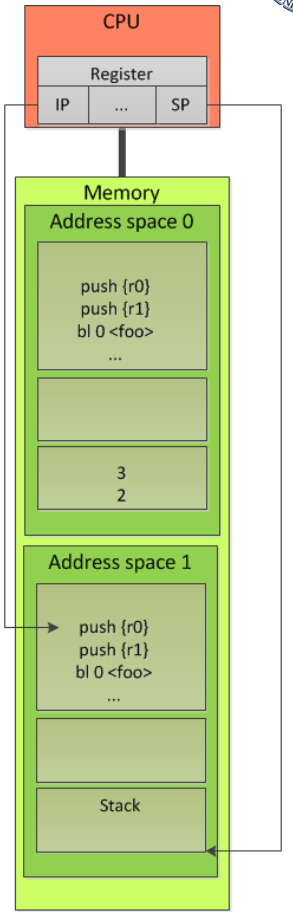
  - Without deblocking and switching of processes the using process will not release the I/O device and no other process is able to use it as well.
  - This would lead to starvation of some processes.
    - Starvation means the process is not able to run as it will get access to a resource needed for processing.

Process Change by Interrupts



LV BKS

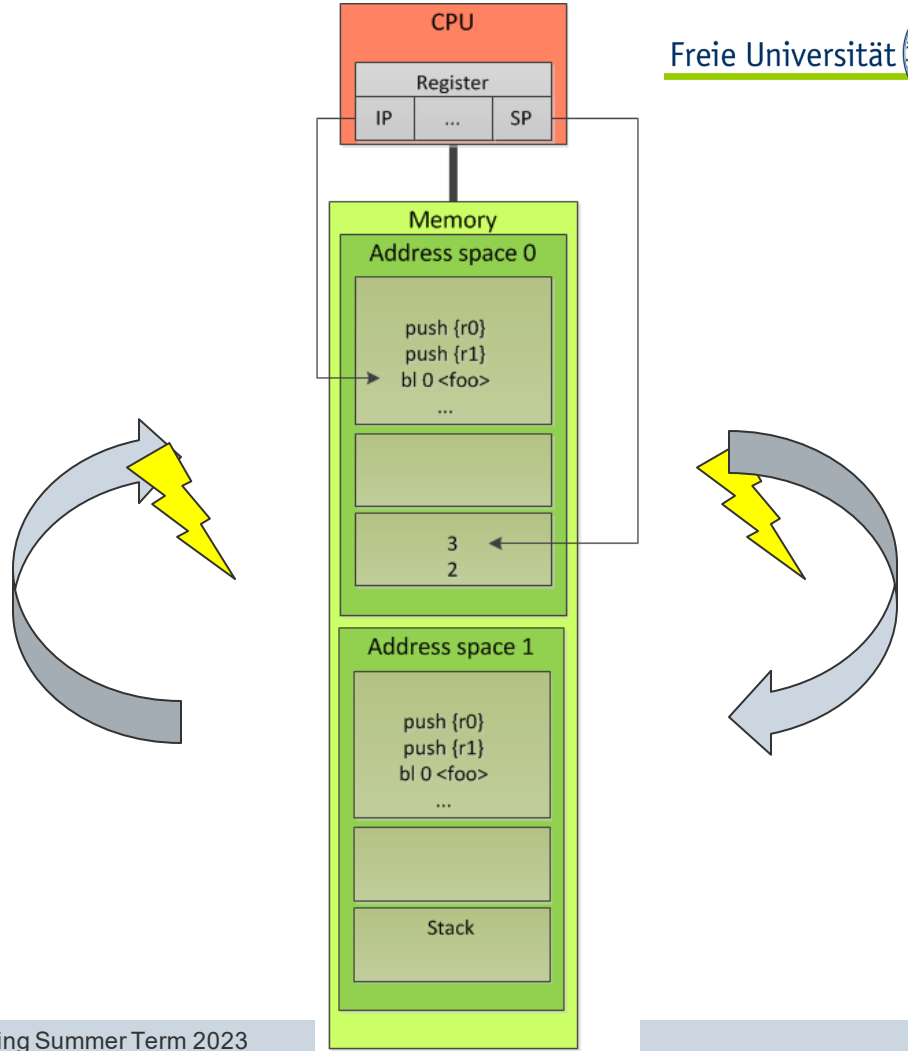
Figure 1.7 Instruction Cycle with Interrupts



Interrupts

- Interrupts interrupt execution of current process.
- The handling of an interrupt provides the possibility to change the state of a process…
  - Deblock the former blocked process.
- … and to schedule another process.
  - Switch to another process.

Process Switch



Process Switch performed by the Operating System

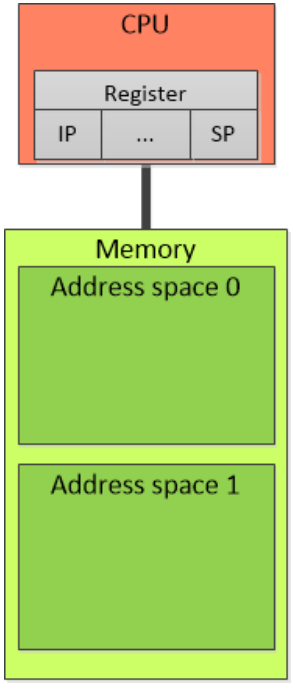
- The interrupt and the interrupt handling gives control of the CPU to the operating system.

- The operating system saves the processing status (context) of the currently running process (e.g. content of the CPU registers).
- The OS changes the state of the process if needed.
- To continue a process, the operating system restores the saved processing status (context) of the process.

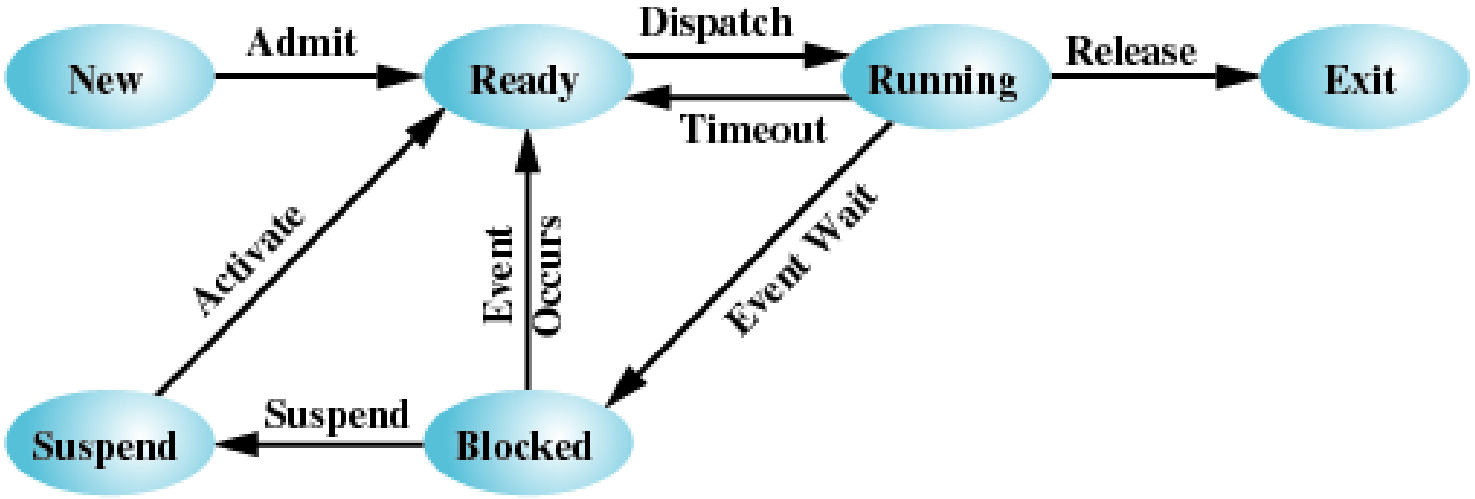
- For the ongoing process these activities are fully transparent (not visible).

Concurrency

- The introduction of concurrent process execution helps to reduce idle time of the CPU as another process can be executed, if the first process is blocked.
- Concurrency comes with some requirements to the system:
  - Separation of memory areas by introducing address spaces ✓
  - Ability of process generation ✓
    - via System call
    - A superordinate (parent) process takes over the process generation and a process hierarchy can be created.
  - The change between processes (of different programs) is to be performed ✓
    - by separation of the processing states and saving of the context of process.



Process States – extended



LV BKS

Effects on Process Execution

- Process switch caused by an interrupt can occur at any time.
  - changing the running process
- What does "anytime" mean?
- Let‘s have a look at our example of two processes:

Effects on Process Execution – Example

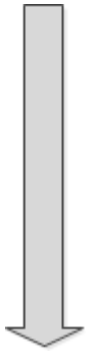
```c
// first program in C

int main (void)
{
  int a = 0;
  int b = 0;

  a = 2;
  b = 3;

  a = a * b;

  return a;
}
```

```c
// second program in C

int main (void)
{
  int a = 0;
  int b = 0;

  a = 3;
  b = 4;

  a = a * b;

  return a;
}
```

Effects on Process Execution – Example

- Execution may take place as follows:

time



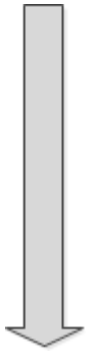
```
a = 2;
b = 3;
a = a * b;
```

```
a = 3;
b = 4;
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



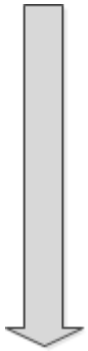
```
a = 2;
b = 3;

a = a * b;
```

```
a = 3;

b = 4;
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



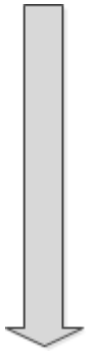
```
a = 2;
b = 3;


a = a * b;
```

```
a = 3;
b = 4;

a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



```
a = 2;
b = 3;
```

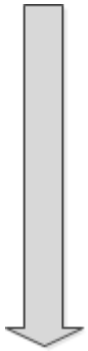
```
a = 3;
b = 4;
a = a * b;
```

```
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



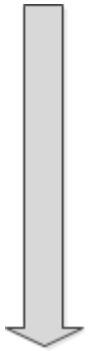
```
a = 2;

b = 3;
a = a * b;
```

```
a = 3;

b = 4;
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



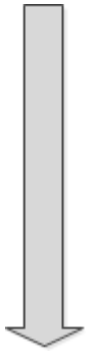
```
a = 2;

b = 3;
a = a * b;
```

```
a = 3;
b = 4;

a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



```
a = 2;
```

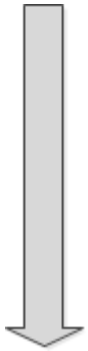
```
b = 3;
a = a * b;
```

```
a = 3;
b = 4;
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



```
a = 2;
b = 3;
a = a * b;
```

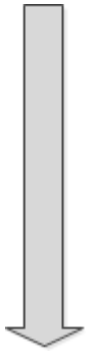
```
a = 3;
```

```
b = 4;
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



```
a = 2;
b = 3;
a = a * b;
```

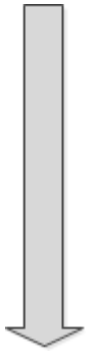
```
a = 3;
b = 4;
```

```
a = a * b;
```

Effects on Process Execution – Example

- … or this way:

time



```
a = 2;
b = 3;
a = a * b;
```

```
a = 3;
b = 4;
a = a * b;
```

Effects on Determinism

- Does the uncertainty about the sequence of execution of the instructions of the different processes has an effect on determinism?

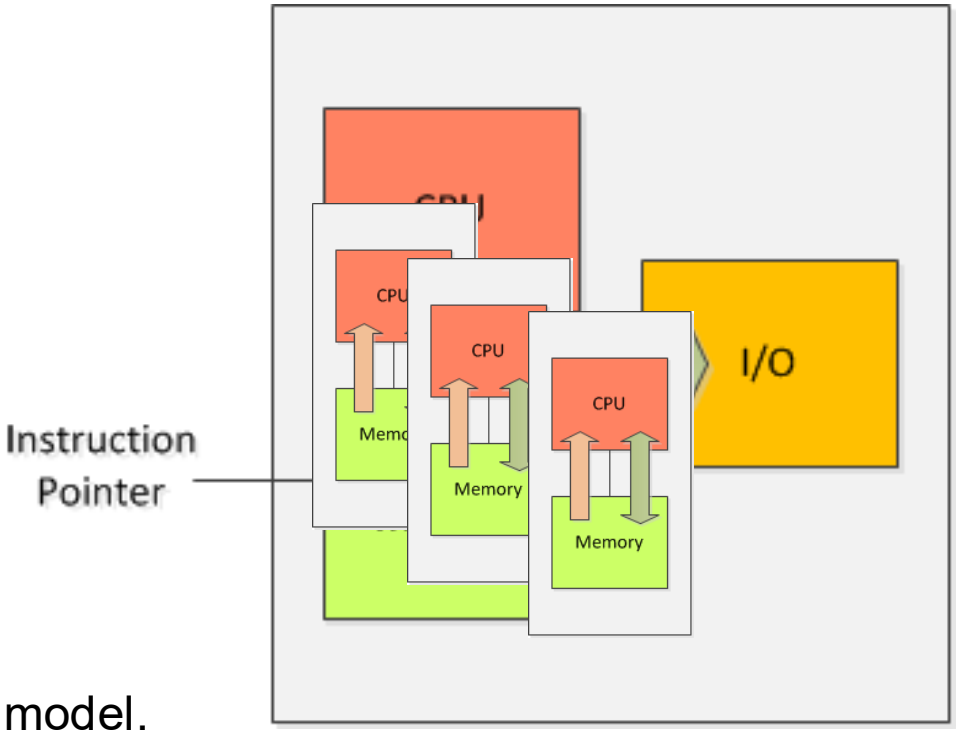
- Well, with respect to the system as a whole the answer is (obviously): **Yes!**

- But, for the single process the sequence of execution of the instructions does not change.
- So, with a view at the single process the answer is: **No!**
  - The state of the execution of the process is frozen while other processes are running and continues at the next instruction at the time the process gets the CPU again.

https://www.telegraph.co.uk/content/dam/film/Frozen/frozen1_3139289a-xlarge.jpg

Virtualization of the Processor

- For every process the systems behaves as it is the only process in the system.
- The process can be seen as a virtualization of the processor.
  - There are as many virtual processors available as processes running.
  - That's not the same concept as the virtual processor of a virtual machine.
    - for more see course "Betriebssysteme"
- Thus, we have to extend our machine model.



Atomic Operations and Machine Model

```
// first program in C

int main (void)
{
  int a = 0;
  int b = 0;

  a = 2;
  b = 3;

  a = a * b;

  return a;
}
```

```
00000000 <main>:
   0:   e52db004        push    {fp}; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e3a03000        mov     r3, #0
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e3a03000        mov     r3, #0
  18:   e50b300c        str     r3, [fp, #-12]
  1c:   e3a03002        mov     r3, #2
  20:   e50b3008        str     r3, [fp, #-8]
  24:   e3a03003        mov     r3, #3
  28:   e50b300c        str     r3, [fp, #-12]
  2c:   e51b3008        ldr     r3, [fp, #-8]
  30:   e51b200c        ldr     r2, [fp, #-12]
  34:   e0030392        mul     r3, r2, r3
  38:   e50b3008        str     r3, [fp, #-8]
  3c:   e51b3008        ldr     r3, [fp, #-8]
  40:   e1a00003        mov     r0, r3
  44:   e24bd000        sub     sp, fp, #0
  48:   e49db004        pop     {fp}; (ldr fp, [sp], #4)
  4c:   e12fff1e        bx      lr
```

Atomic and Non-Atomic Operations

- Atomic (indivisible) operations are executed completely before the process can be interrupted.

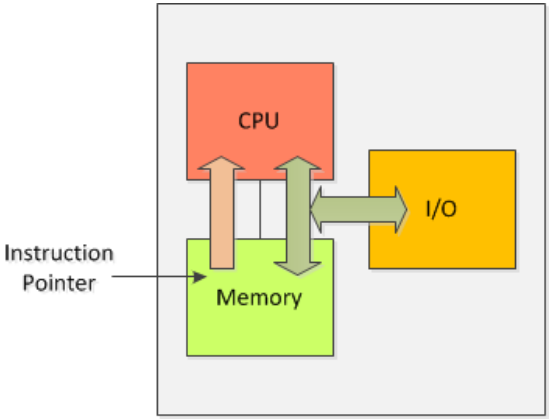
- Non-atomic operations can be interrupted before the their execution is completed.
  - These operations usually consist of more than one machine instruction. But that may differ with the architecture of the processor.

- Processor architectures often offer the possibility to execute certain operations atomically.

SUMMARY

Do we meet the Requirements for Programs?

- A program should do what it is expected to do!
  - Functional requirements, such as
    - Scope of functions
    - Correctness



- A program should comply with certain requirements about its behavior!
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …

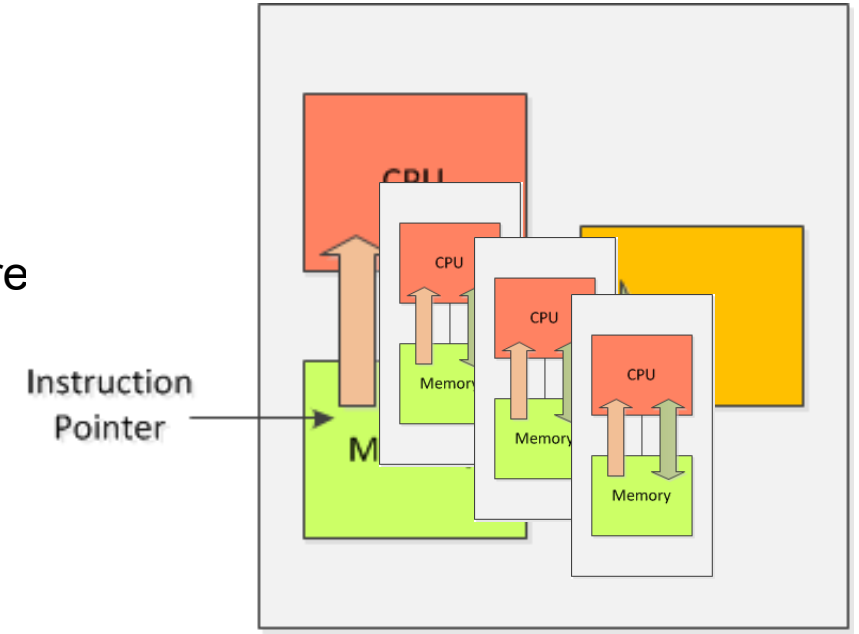
Correctness

- Correctness is ensured based on
  - Correct implementation of commands and functions
    - compiler/interpreter, HW
  - Correct execution of the set of commands and instructions
    - Sequential processing of the instructions of the programs as separate processes
    - Programming model and machine model (execution model) correspond to each other
  - Check with
    - Hoare logic (calculus)
    - Simulation
    - Testing

Requirements for Programs

- A program should do what it is expected to do!
  - Functional requirements, such as
    - Scope of functions
    - Correctness

- A program should comply with certain require about its behavior.
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …



Performance

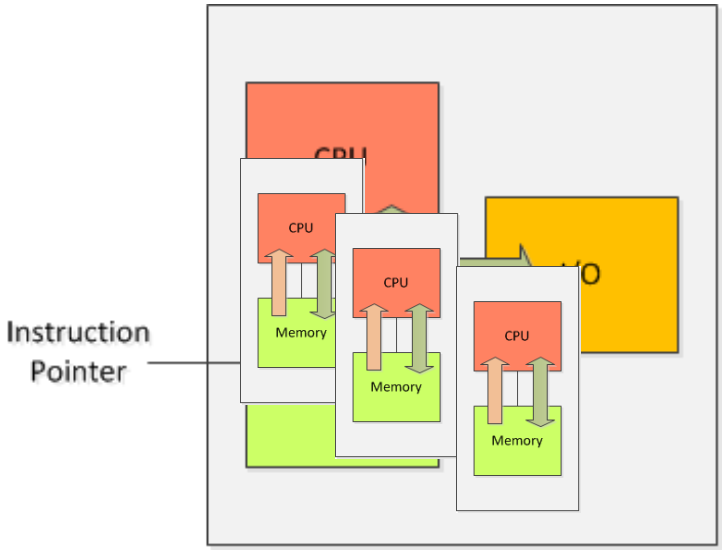
- There are (at least) two different perspectives on *performance*:

- Perspective of the service provider
  - use all resources to run as many programs as possible
  - maximum utilization of resources – especially CPU

- Perspective of the user
  - the fastest possible processing of your own program
  - short response time

Performance Perspective of the Service Provider

- While one process is blocked due to an I/O operation another process can use the CPU.
- Therefore, no resources are wasted and all processes are able to continue their work.



Performance

- There are (at least) two different perspectives on *performance*:

- Perspective of the service provider
  - use all resources to run as many programs as possible
  - maximum utilization of resources – especially CPU

- Perspective of the user
  - the fastest possible processing of your own program
  - short response time

Performance from the User Perspective

- Process switching comes with an overhead.

- But, if the program is designed to use more resources than CPU (and memory) this usage of the different resources may be performed in parallel.
  - Parts of the program are executed concurrently.
- Problem has to be split into reasonable pieces.
  - Divide and conquer approaches
- The usage of all of the resources may lead to a reduction of idle time.
  - Reduction of the response time of the entire program.
- The uniform progress of the processes of the program (and all other processes) is ensured by the operating system.
  - According to the goals and possibilities of the operating system (scheduling)

Let‘s make a more reasonable Example then:

```
// Program in C forking another process
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void) {
  int status;
  pid_t pid;
  int a, b = 0;

  pid = fork();
  if (pid == 0) {
    // child process is calculating
    a = 2;
    b = 3;
    a = a * b;
    printf (" result of child
      process: %d \n", a);
    exit(123);
  }
  else if (pid > 0) {
    // parent process is calculating
    a = 3;
    b = 4;
    a = a * b;
    printf (" result of parent
      process: %d \n", a);

    pid = wait(&status);
    printf("Child process %d
      terminated, status %d.\n", pid,
      WEXITSTATUS(status));
    exit(EXIT_SUCCESS);
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }
}
```

```c
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data




  pid = fork();

  if (pid == 0) {
    // child process is calculating



    exit(?);
  }
  else if (pid > 0) {
    // parent process is calculating



  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }

  // handling results by remaining
  // parent process



  return 0;
}
```

```c
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      data[i][j] = (i + 1) * j;
    }
  }
  pid = fork();

  if (pid == 0) {
    // child process is calculating



    exit(?);
  }
  else if (pid > 0) {
    // parent process is calculating



  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }

  // handling results by remaining
  // parent process



  return 0;
}
```

```c
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      data[i][j] = (i + 1) * j;
    }
  }
  pid = fork();

  if (pid == 0) {
    // child process is calculating
    for (j = 0; j < 3; j++) {
      result += data[0][j];
    }
    exit(?);
  }
  else if (pid > 0) {
    // parent process is calculating
    for (j = 0; j < 3; j++) {
      result += data[1][j];
    }
  }
  else {
    printf("fork() failed\n");
    exit(EXIT_FAILURE);
  }

  // handling results by remaining
  // parent process

  return 0;
}
```

```c
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      data[i][j] = (i + 1) * j;
    }
  }
  pid = fork();

  if (pid == 0) {
    // child process is calculating
    for (j = 0; j < 3; j++) {
      result += data[0][j];
    }
    exit(result);
  }
  else if (pid > 0) {
    // parent process is calculating
    for (j = 0; j < 3; j++) {
      result += data[1][j];
    }
  }
  else {
    printf("fork() failed\n");
    exit(EXIT_FAILURE);
  }

  // handling results by remaining
  // parent process
  pid = wait(&status);
  printf ("\n Result: %d\n",
        result + WEXITSTATUS(status));

  return 0;
}
```

```
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      data[i][j] = (i + 1) * j;
    }
  }
  pid = fork();

  if (pid == 0) {
    // child process is calculating
    for (j = 0; j < 3; j++) {
      result += data[0][j];
    }
    exit(result);
  }
  else if (pid > 0) {
    // parent process is calculating
    for (j = 0; j < 3; j++) {
      result += data[1][j];
    }
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }

  // handling results by remaining
  // parent process
  pid = wait(&status);
  printf ("\n Result: %d\n",
        result +  WEXITSTATUS(status));

  return 0;
}
```

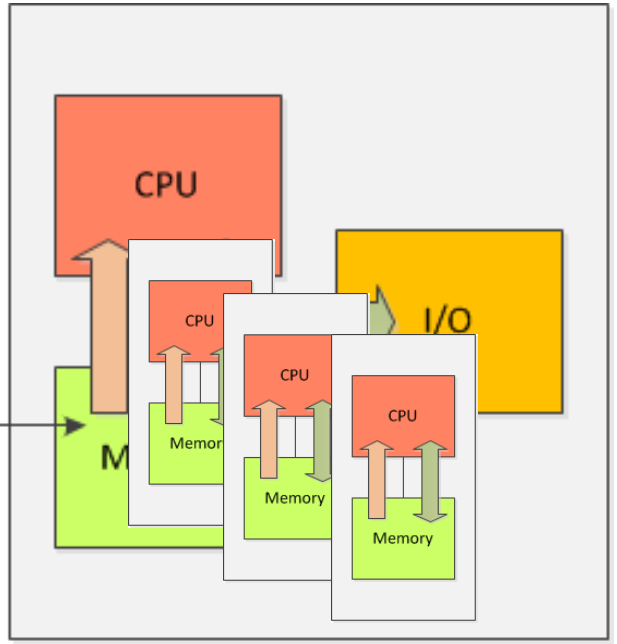
```
// Example with two processes
#include ...
int main(void) {
  int data[2][3];
  int i, j;
  int status, result = 0;
  pid_t pid;

  // init data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      data[i][j] = (i + 1) * j;
    }
  }
  pid = fork();

  if (pid == 0) {
    // child process is calculating
    for (j = 0; j < 3; j++) {
      result += data[0][j];
    }
    exit(result);
  }
  else if (pid > 0) {
    // parent process is calculating
    for (j = 0; j < 3; j++) {
      result += data[1][j];
    }
  }
  else {
    printf("fork() failed\n");
     exit(EXIT_FAILURE);
  }

  // handling results by remaining
                                    );
  return 0;
}
```

How does this implementation have any impact on the performance of the program?

Requirements for Programs

- A program should do what it is expected to do!
  - Functional requirements, such as
    - Scope of functions
    - Correctness

- A program should comply with certain requirements about its behavior.
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …



Concepts of Non-sequential and Distributed Programming

TAKE AWAYS

You should know…

- that the concurrent execution of processes have an impact on performance of the system and the program,
- that you can use the fork() system call to create new processes,
- that the concurrent execution of the processes has an effect to determinism and determined execution of the program, but not from the view of the process,
- that there is a difference between atomic and non-atomic operations.

Concepts of Non-sequential and Distributed Programming

NEXT LECTURE

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Concurrency with Threads

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2023)