

Süleyman Gölbol
76210123

Programiranje III
Parallel and distributed programming

INTRODUCTION

Janez Žibert, Jernej Vičič

UP FAMNIT

Basic information

- ▶ Associate professor:
 - ▶ izr. prof. Jernej Vičič
 - ▶ contact: jernej.vicic@upr.si
 - ▶ contact hours (ZOOM): just make an appointment (I prefer evenings)
- ▶ Teaching assistant:
 - ▶ Sead Jahić
 - ▶ contact: sead.jahic@famnit.upr.si
- ▶ Technical data:
 - ▶ CT: 6
 - ▶ Lectures: 45 hours, lab exercises 45 hours, remaining 90 hours individual work (45 hours for project work)
 - ▶ Compulsory 2nd year course
 - ▶ Program pillar: programming

Objectives and competencies

- ▶ Prior knowledge to be acquired by the student:
 - ▶ Successfully completed coursework: Programming I and Programming II.
- ▶ Learning Objectives:
 - ▶ get acquainted with the concepts of parallel and distributed programming.
 - ▶ use the knowledge of parallel and distributed programming in solving practical tasks.
 - ▶ learn about tools, programming languages and techniques for the development of parallel systems.

Course implementation

- ▶ **Basic literature:**
 - ▶ book: **Foundations of Multithreaded, Parallel and Distributed Programming**, G. R. Andrews, Addison Wesley, 2000 <http://www.cs.arizona.edu/~greg/mpdbook/>
 - ▶ Lecture notes (e-classroom)
 - ▶ Janez Žibert (only in Slovene):
http://temena.famnit.upr.si/files/files/skripta_vzporedno_programiranje.pdf
- ▶ **Course implementation:**
 - ▶ Lectures (13 lectures)
 - ▶ Lab exercises
 - ▶ **3 homeworks:** solving a selected problem in three ways:
 - ▶ 1. sequential (normal) program
 - ▶ 2. parallel program
 - ▶ 3. distributed program

Grading

Homeworks/Projects	40%
Written exam	30%
Oral exam	30%

- ▶ Each grading:
 - ▶ it is necessary to achieve a minimum number of points that represent the lower limit for the student to complete this part of the task, for example, homework ≥ 40 points, exam ≥ 50
- ▶ Homework:
 - ▶ It is necessary to complete: **all three tasks ≥ 40**

Overview

- ▶ Introduction. Basic terms and concepts of parallel and distributed programming.
- ▶ Parallel programming (shared memory programming):
 - ▶ processes and interprocess synchronization
 - ▶ synchronization mechanisms when using shared memory:
 - ▶ locks, flags, barriers, conditional variables, semaphores, monitors
- ▶ Distributed programming (distributed memory programming)
 - ▶ by exchanging messages,
 - ▶ with the RPC (remote procedure call) method and the "rendezvous" principle,
 - ▶ with the RMI method (remote method invocation)
- ▶ Examples of parallel computing.
- ▶ Overview of selected tools for parallel and distributed programming.

Lectures

Lecture	Overview	Reading
1	INTRODUCTION Basic terms and concepts of parallel and distributed programming	Chapter 1
2	Processes and synchronisation	2.1 – 2.5
3	Crytical sections, Locks.	3.1-3.3
4	Barriers. Flags. Data and procedural parallelism.	3.4 – 3.6

Učni načrt

5	Semaphores	4
6	Pthreads library.	4.6, 5.5 <i>Pthreads API Manual</i>
7	Monitors, conditional variables	5
8	Miltithread programming in Java. Java and monitors.	5.4
9	Parallel and distributed programming examples – I	11

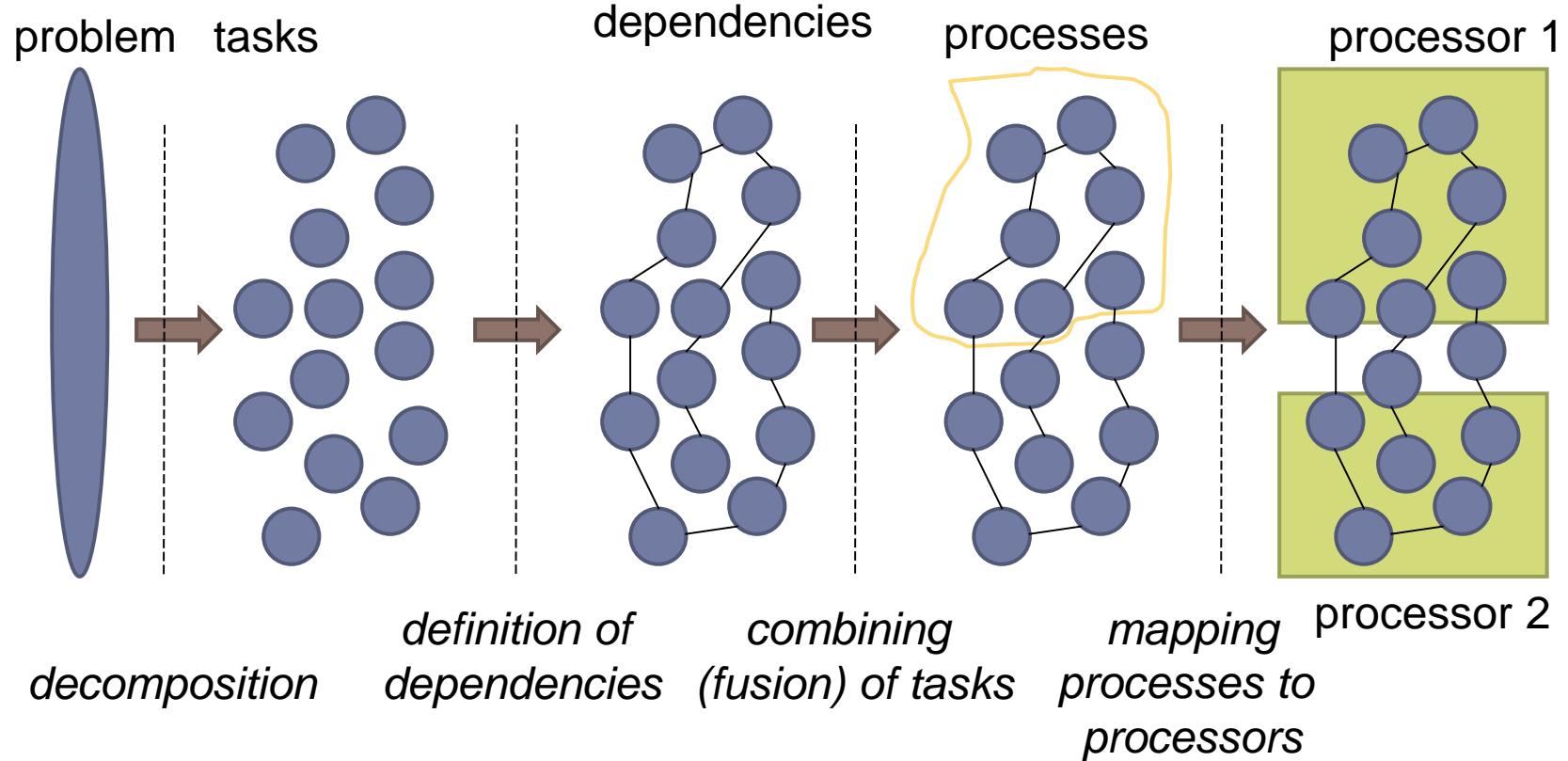


Učni načrt

10	Distributed programming with message passing	7.1 – 7.5
11	Intro into MPI	7.8 <u>A User's Guide to MPI</u>
12	Parallel and distributed programming examples – II	11
13	MapReduce	
14	General Purpose GPU programming	



Planning a parallel algorithm



Problem decomposition

- ▶ Solving the problem by dividing it into individual subproblems - tasks:
 - ▶ tasks can be defined statically or dynamically (if necessary),
 - ▶ the number of tasks depends on the problem that we have to solve and can change during the problem solving procedure,
 - ▶ decomposition according to the data, according to subproblems.
 - ▶ the number of jobs should be at least one magnitude greater than the number of processors we will be using,
- ▶ Objective:
 - ▶ define the tasks so that they can be solved simultaneously,
- ▶ Example:
 - ▶ The loops in a sequential program can be performed in parallel.

Definition of dependencies

- ▶ Tasks after decomposition should be solved concurrently, but they are not necessarily independent:
- ▶ a task may require the data/results of other tasks for its solution,
- ▶ It is necessary to determine dependencies between tasks and determine the course of communication:
 - ▶ communication channels: who communicates, who expects data, who computes data,
 - ▶ communication and synchronization algorithms.

combining (fusion) of tasks

- ▶ The result of the split and integration is a large number of tasks with defined relationships.
- ▶ At this stage, the following may appear:
 - ▶ the communication burden is too high,
 - ▶ work between jobs is unevenly distributed,
 - ▶ increase of program load required to synchronize tasks
 - ▶ Therefore, at this stage:
 - ▶ combine small tasks into a larger one in order to solve the problems of communication and synchronization.
- ▶ General idea:
 - ▶ join those tasks that can not progress at the same time.

Translation of the algorithm to the computer

- ▶ After the first three phases we already have a parallel program.
- ▶ Here we determine:
 - ▶ which processor will compute a selected process,
 - ▶ which processes will be run on the same processor.
- ▶ The best solution:
 - ▶ one that minimizes the time the program is running.
 - ▶ The problem of process allocation to processors is NP full.
- ▶ Recommendations:
 - ▶ assign tasks that can progress simultaneously to different processors.
 - ▶ assign tasks that intensively communicate with each other, the same processor.

Programming III

Parallel and distributed programming

Basic terms and concepts parallel and distributed programming

Janez Žibert, Jernej Vičič

UP FAMNIT

Overview (1. chapter in the book)

- ▶ What is a parallel program?
- ▶ Computer architectures for running parallel programs.
- ▶ Basic concepts in (parallel) programming: task, process, process state, ...
- ▶ Interprocess communication:
 - ▶ communication with shared memory,
 - ▶ communication through linking structures.
- ▶ Basic concepts of parallel/distributed programming:
 - ▶ iterative parallelism,
 - ▶ recursive parallelism,
 - ▶ principle manufacturer/consumer,
 - ▶ client/server principle,
 - ▶ principle of interaction among equivalent entities.

Sequential an parallel program

▶ Sequential program:

- ▶ represents a sequence of commands - operations performed one after the other in order to solve (perform) a particular task,
 - ▶ a running program is called a process.

▶ Parallel program:

- ▶ it involves two or more processes involved in solving a particular task:
 - ▶ each process implements its sequence of commands,
 - ▶ processes participate/communicate with each other through shared memory or with the help of messages
 - ▶ running:
 - ▶ simultaneously on single-processor systems (task-switching),
 - ▶ parallel to multiprocessor or multi-computer systems.

Forms of programs with regard to the running of processes

- ▶ **Simultaneous computing applications**
 - ▶ processes are run simultaneously on one processor (or multiple processes per processor)
 - ▶ communication is done by reading and writing to shared memory,
 - ▶ applications:
 - ▶ modern operating systems,
 - ▶ graphic interfaces,
 - ▶ real-time systems.
- ▶ **Distributed computing applications**
 - ▶ processes are run/executed on several computers that are connected to a network,
 - ▶ communication is done through the exchange of messages (by receiving and transmitting messages),
 - ▶ applications:
 - ▶ wherever there is a requirement for distributed resources:
 - ▶ file and data servers,
 - ▶ www servers, p2p, iptv, ...
- ▶ **Parallel computing applications**
 - ▶ processes run parallel on multiple processors (as many processes as processors)
 - ▶ communication:
 - ▶ using shared memory or with messages
 - ▶ applications:
 - ▶ speeding up procedures for processing large amounts of data:
 - ▶ modeling: optimization, statistical modeling, image and video processing (graphics cards).

Why parallel programs?

▶ Motivations:

- ▶ speeding up procedures (due to the division of work between multiple processors, computers)
- ▶ requirements for the distributed operation of individual services (client/server, p2p),
- ▶ "natural" approach to solving certain problems:
 - ▶ multitasking in operating systems,
 - ▶ graphical interfaces,,
- ▶ increasing scalability of individual programs/applications

Efficiency of parallel programs

- ▶ In parallel programming, we are looking for such algorithms that lead us to the desired goal rather than a strictly sequential solution to the problem.
- ▶ Measuring the speedup:

- ▶ actual speedup:

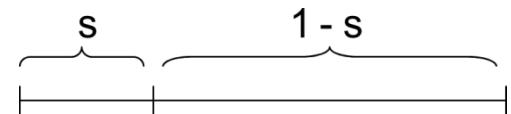
$$S(N) = \frac{\text{sequential algorithm on 1 proc. of a parallel computer}}{\text{parallel algorithm on } N \text{ proc. of a parallel computer}}$$

- ▶ relative speedup:

$$S(N) = \frac{\text{parallel algorithm on 1 proc. parallel computer}}{\text{parallel algorithm on } N \text{ proc. parallel computer}}$$

Factors limiting the speedup

- ▶ problem decomposition costs
 - ▶ increment of the number of instructions - increasing the program load (additional operations, new variables, ...),
 - ▶ uneven load distribution (uneven distribution of data, unbalanced complexity of suboperations),
 - ▶ an increase in the communication burden (synchronization)
 - ▶ sequential nature of the problem:
- ▶ Amdahl Law:
 - s - the degree of sequentiality of the problem, the percentage of time we need for solving the sequential part of the program.
The speedup of the problem is limited by $1/s$.



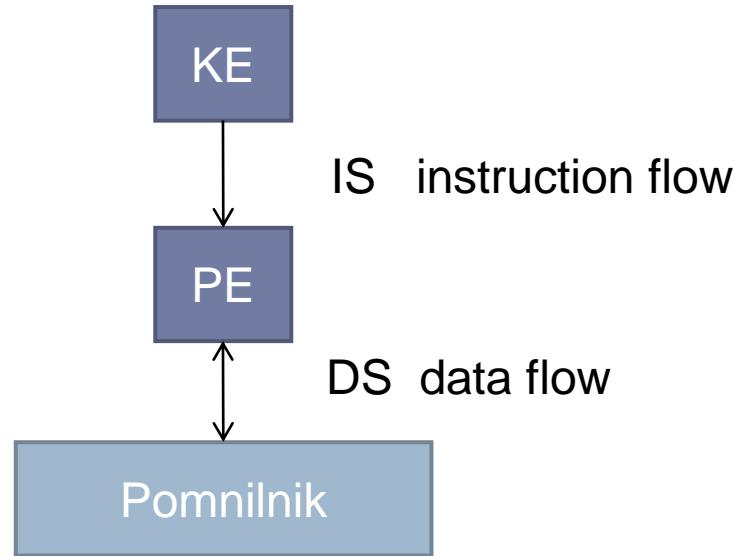
$$S(n) = \frac{T}{\frac{T(1-s)}{n} + T \cdot s}$$

$$\lim_{n \rightarrow \infty} \frac{T}{\frac{T(1-s)}{n} + T \cdot s} = \frac{1}{s}$$

Computer models for parallel programming

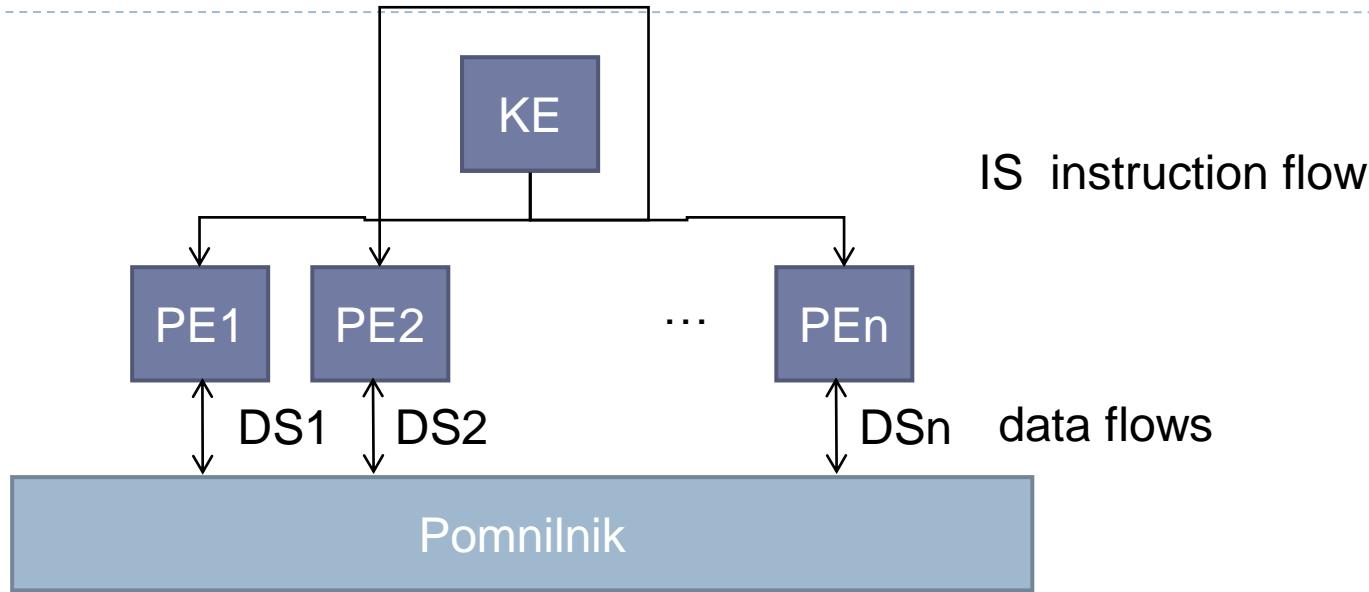
- ▶ The computer executes commands over the data:
 - ▶ the sequence of commands (flow of commands) specifies the operations that the computer executes one after another,
 - ▶ the sequence of data (data flow) determines over what operations are executed,
- ▶ Given the number of simultaneous flows of commands and data, we divide the computers into the following models (Flynn):
 - ▶ SISD (Single Instruction Single Data Stream)
 - ▶ SIMD (Single Instruction Multiple Data Stream)
 - ▶ MISD (Multiple Instruction Single Data Stream)
 - ▶ MIMD (Multiple Instruction Multiple Data Stream)
 - ▶ shared memory,
 - ▶ distributed memory.

SISD computer



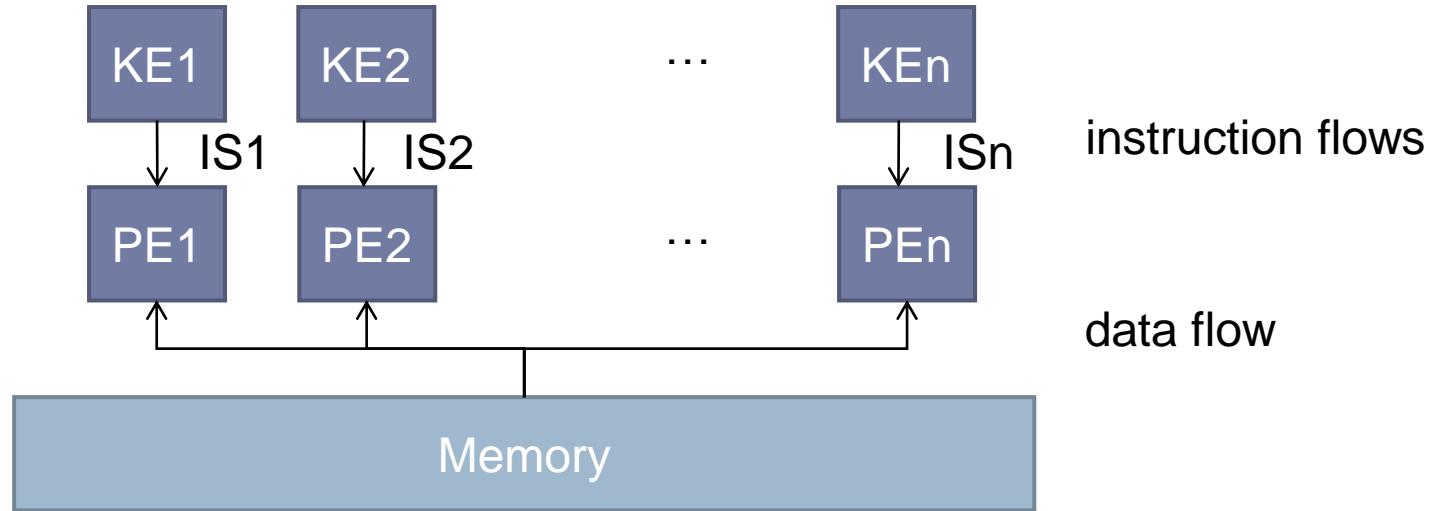
- ▶ von Neumannov type computer
- ▶ sequential computer:
 - ▶ the computer executes instructions one-by-one
- ▶ Example: add N numbers

SIMD computer



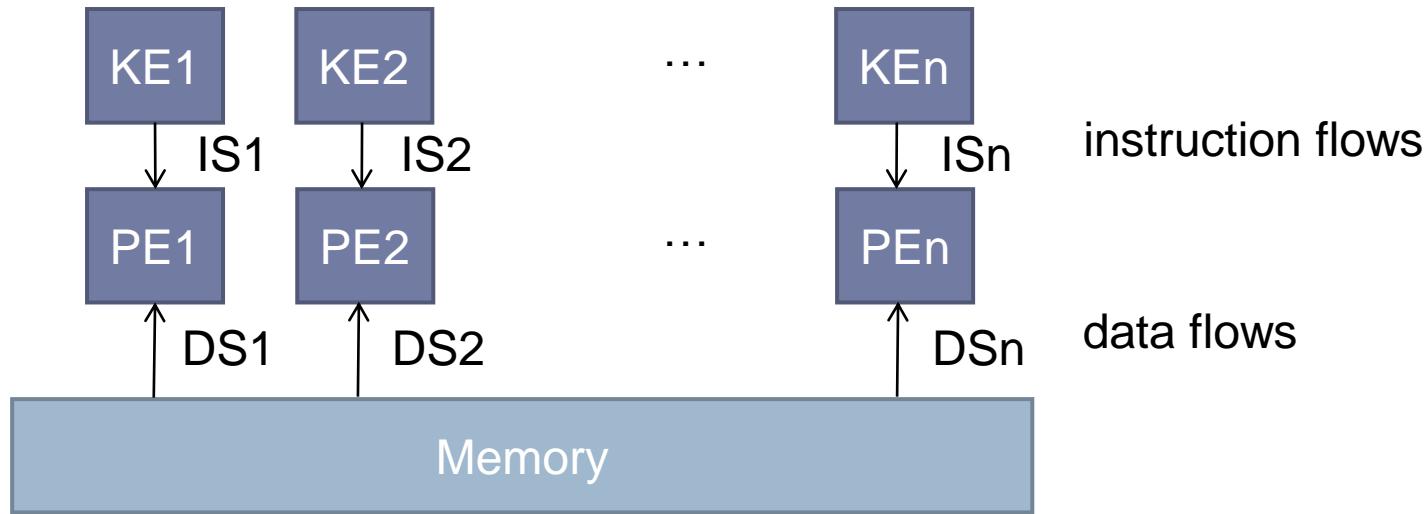
- ▶ the computer executes at a given moment one operation over more than one data,
- ▶ simultaneous access to data via vectors or arrays
- ▶ Example: simultaneously adding multiple pairs of numbers

MISD computer



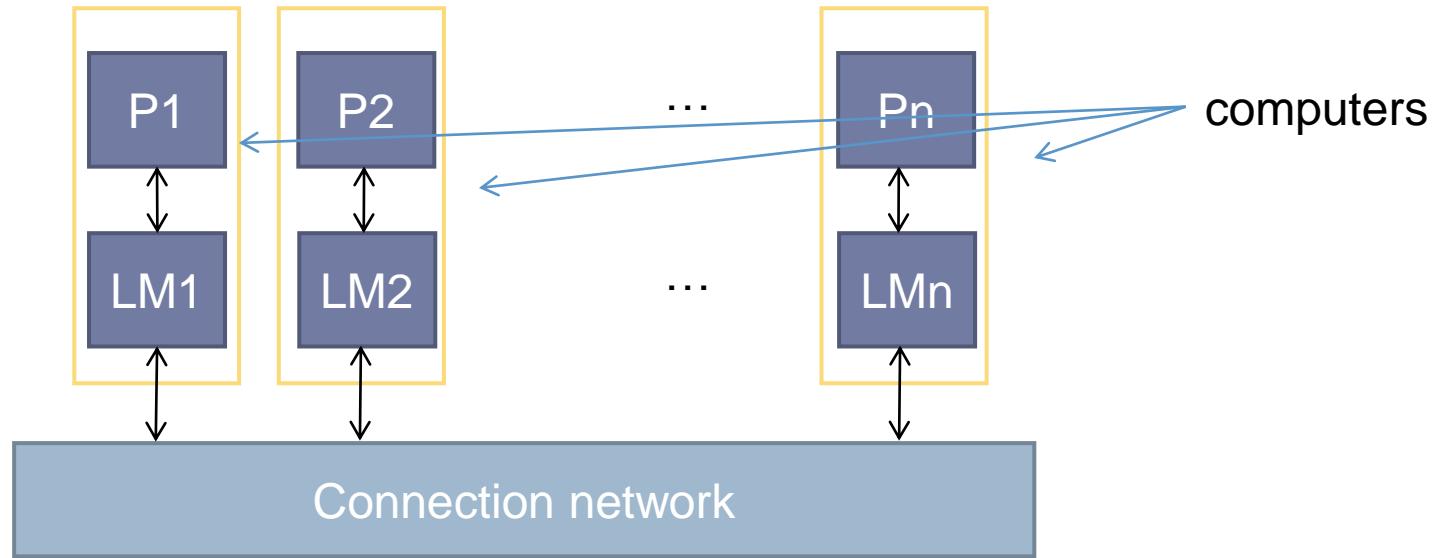
- ▶ the computer executes at a given moment several operations over one datam
- ▶ the model was not implemented
- ▶ Example: Is the number N prime?

MIMD computer with shared memory



- ▶ the computer executes at a given moment a number of operations over different data,
- ▶ Multi-processor system:
 - ▶ processors share shared memory,
 - ▶ processors access global memory through a common bus (bottleneck if we have a lot of processors).
 - ▶ processors communicate with each other by reading and writing to a shared memory.

MIMD computer with distributed memory

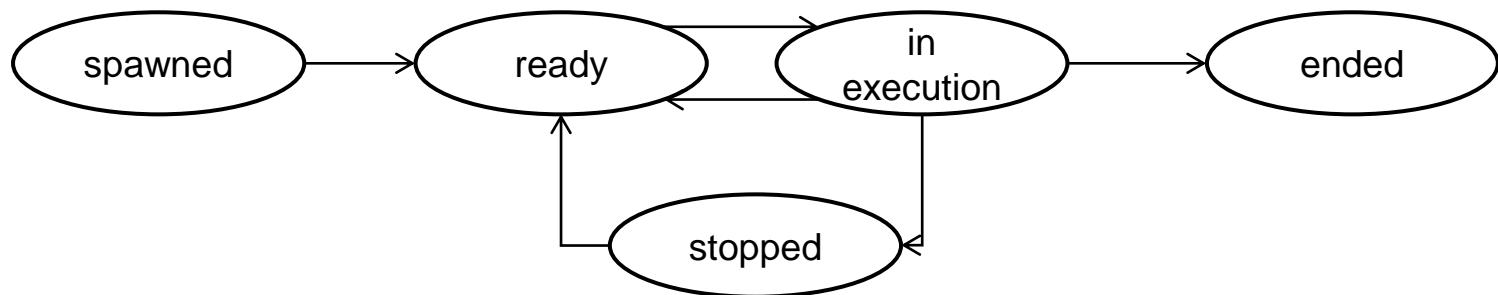


▶ Multi computer system:

- ▶ system of multiple computers connected to a connection network (do not share memory!),
- ▶ they communicate with each other by exchanging messages,
- ▶ various structures of multi-computing systems:
 - ▶ same/different performance computers (processors),
 - ▶ topology of interconnection networks (networks, hypercube, ...),
 - ▶ resource management: centralized, distributed,
 - ▶ types of connections: myrinet, infiniband, ethernet...

Basic concepts in parallel programming: PROCESS

- ▶ **task** – unit of execution of a computer program performing a task,
- ▶ at PP: commands (operations) in a task are executed in succession,
- ▶ **process** – computer program in execution (when the program starts running it becomes a process):
 - ▶ can perform one or more tasks,
 - ▶ during the execution the process changes state:

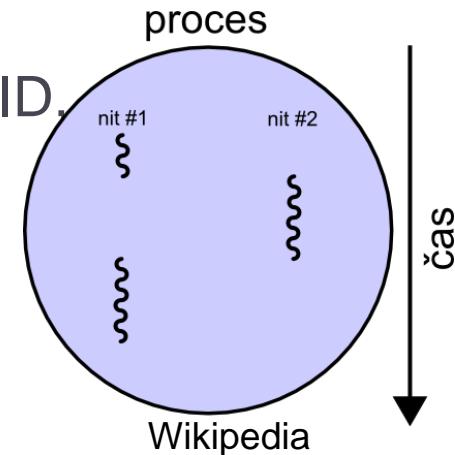


PROCESSES in operating systems

- ▶ In a multi-tasking OS, there are several processes over a certain period, each of them is at a certain stage of progression.
- ▶ In the running state: one process per processor,
- ▶ In standby mode or stopped: any number,
- ▶ In each OS, we have a scheduler:
 - ▶ it determines which process to run next (and for how long),
- ▶ Process switching in the processor is performed by the dispatcher:
 - ▶ context switch:
- ▶ Process context - all that is needed to implement the process:
 - ▶ address space: text, data, stack, shared memory ...,
 - ▶ control data,
 - ▶ process ID, program counter, stack pointer, registers, ...

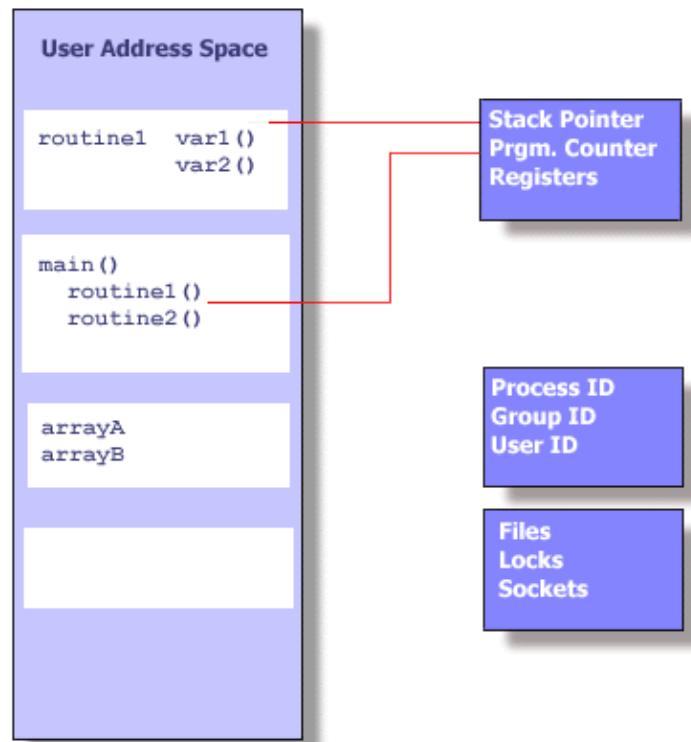
PROCESSES and THREADS

- ▶ Each process can use one or more threads.
- ▶ A thread is a new process that shares the father's address space:
 - ▶ has its own stack, its own registers, has its own ID.
 - ▶ with the other threads of the common process:
 - ▶ shares a common address space and,
 - ▶ global variables.
 - ▶ has access to common files,
 - ▶ has access rights to everything within this process.
- ▶ Example: LINUX fork () and clone ():
 - ▶ fork() creates a new process with a new process context,
 - ▶ clone() creates a new process with its own identity,
 - ▶ but with the sharing of data structures of the father.

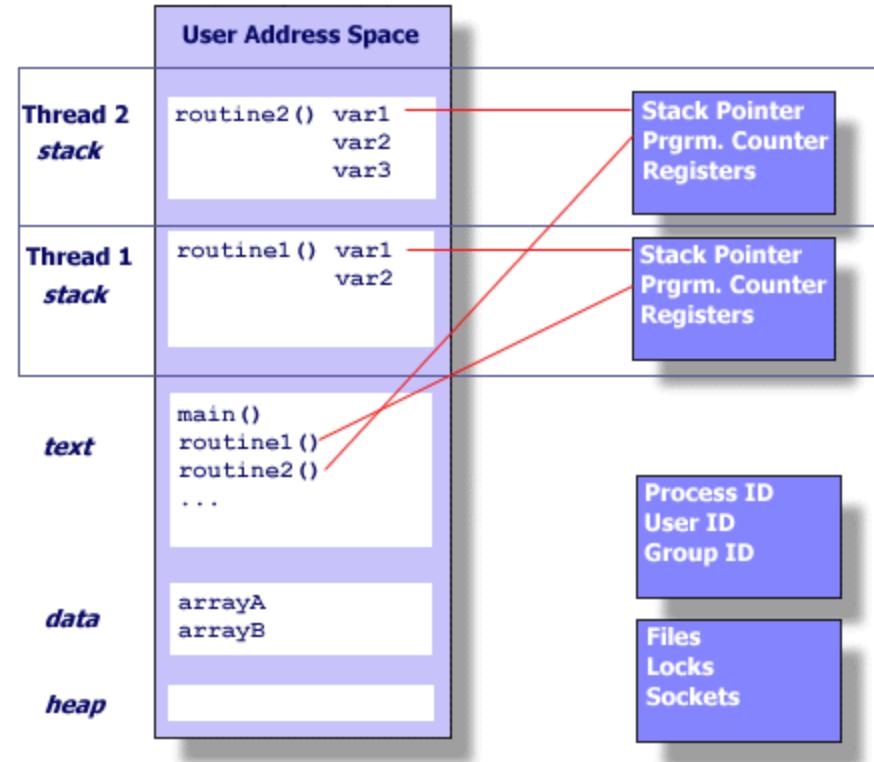


PROCESSES and THREADS

UNIX processes

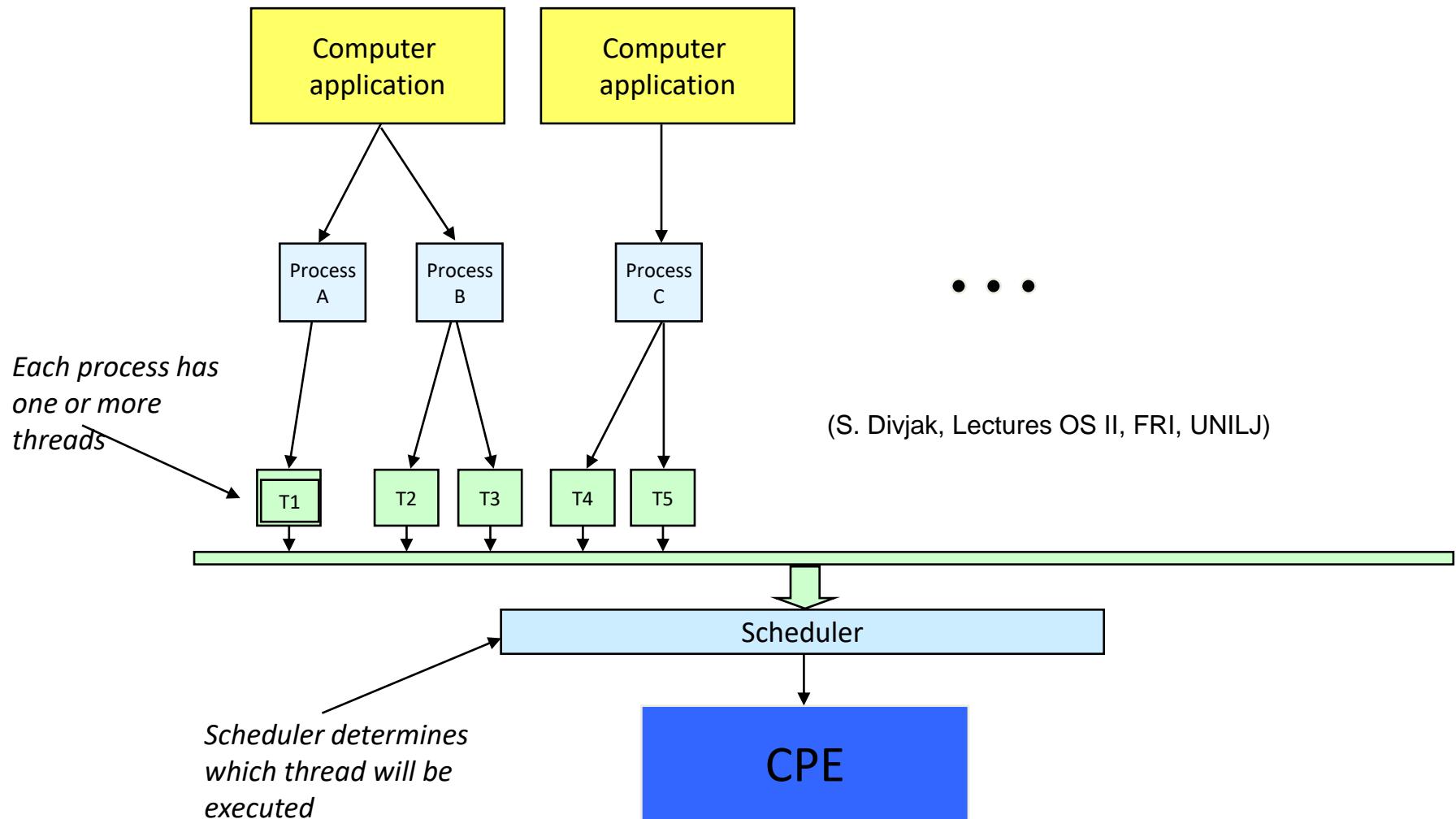


Threads in UNIX processes



<https://computing.llnl.gov/tutorials/pthreads/>

Processes in one-processor system



Types of parallel programming

- ▶ In parallel programming, we determine how processes communicate with one another (interprocess communication) in solving a specific task and what methods of synchronization are used.
- ▶ Programming using shared memory
 - ▶ processes share common memory,
 - ▶ communication is done by reading and writing data into common (global) variables (data structures),
 - ▶ explicit synchronization is needed when processes can be write/read from common variables.
- ▶ Distributed memory programming
 - ▶ processes do not have shared memory,
 - ▶ they share interconnection structures (communication channels),
 - ▶ communication is done by sending and receiving messages via communication channels,
 - ▶ synchronization is implicit: in order for the message to be accepted, it must be sent.

Synchronisation and communication mechanisms

- ▶ Synchronisation mechanisms (shared memory):
 - ▶ locks,
 - ▶ barriers,
 - ▶ condition variables,
 - ▶ semaphores,
 - ▶ monitors.
- ▶ Communication mechanisms (distributed memory):
 - ▶ message exchange
 - ▶ asynchronous and synchronous message exchange,
 - ▶ with calls to perform remote procedures (operations)
 - ▶ with the RPC (remote procedure call) method and the "rendezvous",
 - ▶ with the RMI method (remote method invocation).

Pseudo code, used in the book

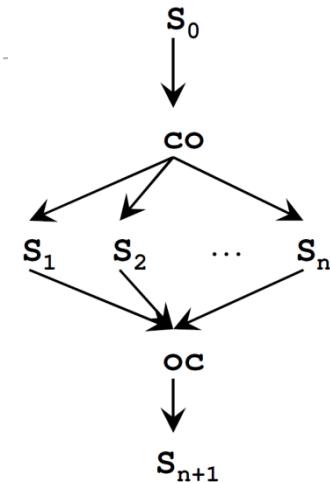
- ▶ **C-like language.**
- ▶ Declarations – as in **C** or/and **matlab**
 - ▶ variables: basic types (`int`, `double`, `char`, ...)
 - ▶ arrays: `int c[1:n]`, `double c[n,n]`
 - ▶ process declaration `process` – start execution of one or more processes in the background.
- ▶ Instructions:
 - ▶ assignment `a= y*x + f(z)`
 - ▶ flow control: `if`, `while`, `for` (as in **C** or Java)
 - ▶ `for` loops (**matlab**)
 - `for [i=0 to n-1, j=0 to n-1]`
 - `for [i=1 to n by 2]` #all odd numbers from 1 to n
 - `for [i=1 to n st i != x]` #all numbers from 1 to, except `i==x` (`st` = “such that”)
 - ▶ **concurrent execution** – `co`
 - the command begins with the simultaneous execution of two or more threads waiting for it until they are finished
 - ▶ **synchronisation** – `await`
 - the command is waiting until a condition is met (it will be presented below)

CO

branching form

- each thread is defined in one branch
- threads are executed in parallel
- at the `oc` we wait until
- all threads finish

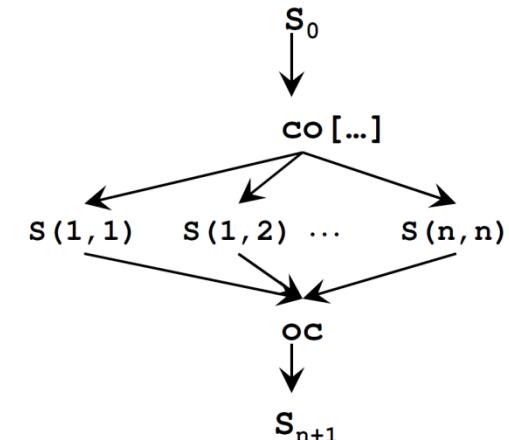
```
s0;  
co s1; # thread 1  
|| ...  
|| sn; # thread n  
oc;  
sn+1;
```



using iterative variables

- threads are defined by a combination of iterative variables,
- defined in the `co` command
- threads are executed in parallel,
- at the `oc` point we wait until all the threads finish

```
s0;  
co [i=1 to n, j=1 to n] { # n x n threads  
    s(i, j);  
}  
oc;  
sn+1;
```



Declaration of a **process**

- ▶ Syntax is similar to `co`.

- ▶ Two forms:

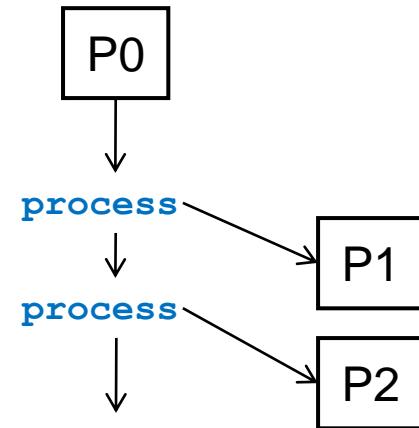
- ▶ one process

```
process bar1 {  
    for [i=1 to n]  
        write(i);  
}
```

- ▶ many processes

```
process bar2 [i= 1 to n] {  
    write(i);  
}
```

- ▶ The process is executed at the declaration and is run in the background.
- ▶ the program continues its execution,
- ▶ we do not wait for the processes to be completed.
- ▶ Within the process we can define local variables and use global variables.



Concepts of parallel/distributed programming

- ▶ iterative parallelism,
- ▶ recursive parallelism,
- ▶ principe producer/consumer,
- ▶ principe client/server,
- ▶ principe interaction among equals.

Iterative parallelism

- ▶ Parallelization of independent iterations in sequential procedures.
- ▶ For example. parallelization of loops, where the loop iteration is performed independently of other iterations.
- ▶ Example: multiplication of matrices $C = A * B$

▶ **sequential program**

```
double a[n,n], b[n,n], c[n,n]
for [i=0 to n-1] {
    for [j=0 to n-1] {
        c[i,j] = 0;
        for [k=0 to n-1]
            c[i,j] = c[i,j]+a[i,k]*b[k,j];
    }
}
```

parallel program

```
double a[n,n], b[n,n], c[n,n]
co [i=0 to n-1] {
    for [j=0 to n-1] {
        c[i,j] = 0;
        for [k=0 to n-1]
            c[i,j] = c[i,j]+a[i,k]*b[k,j];
    }
} oc;
```

- ▶ substitute **for** and **co**,
- ▶ we get n threads for each line i , that is executed at the same time,
- ▶ we could parallelize on rows and columns and get (n^2 threads)
 - ▶ **co [i=0 to n-1, j=0 to n-1]**

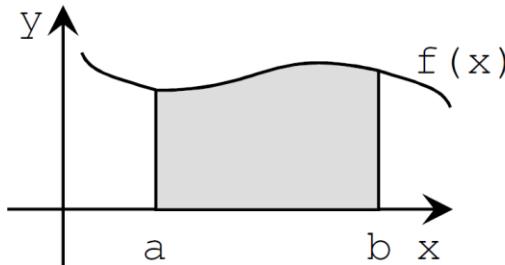
Recursive parallelism

- ▶ Parallelization of independent recursive calls.
- ▶ we have a program that is recursively executed on data
- ▶ recursions are independent among themselves => each recursive call is performed in its thread.
- ▶ Examples:
 - ▶ quick sort, recursive calculation of a given integral, ...wherever we can use recursion according to the "divide and conquer" principle - so where we can use the same process on smaller pieces of data to solve the problem on all data.

Approximation of the calculation of a definite integral

▶ Task:

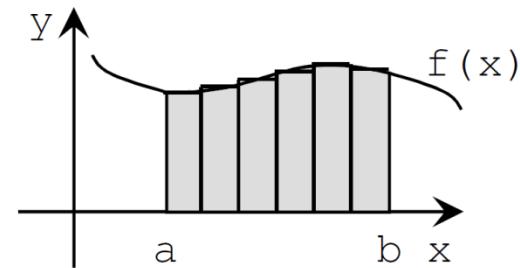
- ▶ Approximate the calculation of a definite integral of a function on an interval!



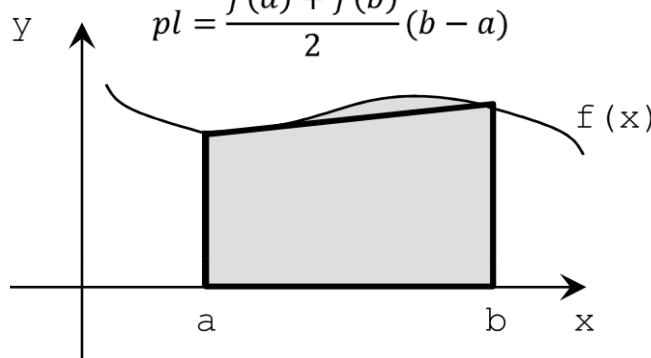
▶ Iterative procedure (sequential program)

- ▶ use a trapezoidal rule

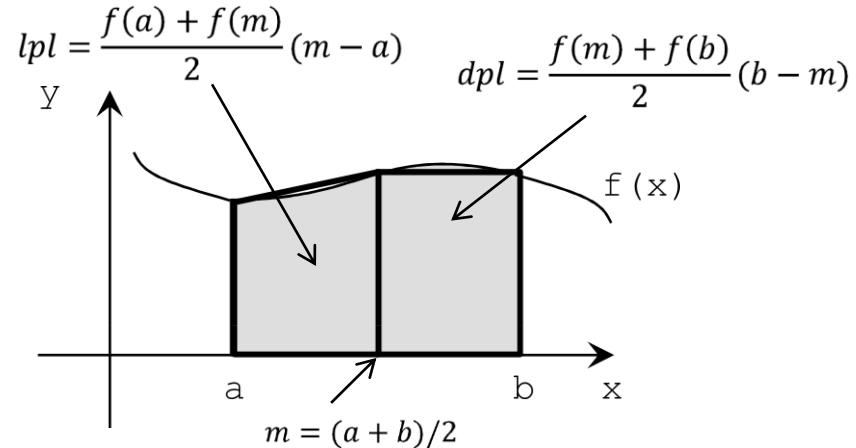
```
double f1 = f(a), fr = f(b), pl = 0.0;  
double dx = (b-a)/ni;  
for [x = (a+dx) to b by dx] {  
    fr = f(x);  
    pl = pl + (f1+fr)*dx/2;  
    f1 = fr;  
}
```



Recursive calculation of a definite integral



first approx
 pl



second approx
 $lpl + dpl$

► Recursion:

- ▶ calculate area pl on a given interval
- ▶ split the interval into two halves and calculate left and right area, lpl and rpl
- ▶ recursion is stopped when $| (lpl+dpl) - pl | < \text{eps}$

Recursive approximation of the calculation of a definite integral

Sequential program

```
double ninteg(double l,d,f1,fd,pl) {  
    double m = (l+d)/2;  
    double fm = f(m);  
    double lpl = (f1+fm)*(m-l)/2;  
    double dpl = (fm+fd)*(d-m)/2;  
    if (abs((lpl+dpl)-pl) > eps) {  
        lpl = ninteg(l,m,f1,fm,lpl);  
        dpl = ninteg(m,d,fm,fd,dpl);  
    }  
    return (lpl+dpl);  
}
```

Parallel program

```
double ninteg(double l,d,f1,fd,pl) {  
    double m = (l+d)/2;  
    double fm = f(m);  
    double lpl = (f1+fm)*(m-l)/2;  
    double dpl = (fm+fd)*(d-m)/2;  
    if (abs((lpl+dpl)-pl) > eps) {  
        co lpl = ninteg(l,m,f1,fm,lpl);  
        || dpl = ninteg(m,d,fm,fd,dpl);  
        oc;  
    }  
    return (lpl+dpl);  
}
```

starting call

```
pl = ninteg(a,b,f(a),f(b),(f(a)+f(b))/(b-a))
```

- ▶ Recursive calls are independent and can be executed in parallel.

Producer/consumer principle

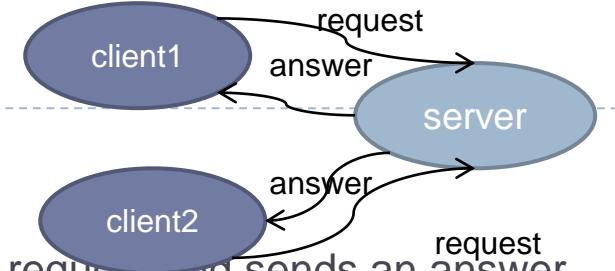
- ▶ The producer produces data
- ▶ The consumer uses the data
- ▶ It is a reciprocal relationship, the production and consumption of data is carried out simultaneously.
- ▶ The flow of data is directed from the producer to the consumer.
- ▶ In the middle we can have "filters".
- ▶ Example

```
$ cat list-1 list-2 list-3 | sort | uniq > final.list
    # Concatenates the list files,
    # sorts them,
    # removes duplicate lines,
    # and finally writes the result to an output file.
```

- ▶ each program reads from the input and writes to the output, the program uses the data produced by the previous program and produces data for its successor,
- ▶ data between programs (processes) are stored in a buffer organized by a FIFO queue.

Client/server principle

- ▶ Client-server relationship
 - ▶ the client requires a service,
 - ▶ the server that provides the service responds to the request and sends an answer
 - ▶ mutual communication: demand, response
- ▶ Parallelism must be enabled on the server side, which must "service" multiple clients:
 - ▶ multithread process, sometimes synchronization between services is also needed,
 - ▶ there can be many clients.
- ▶ Implementation:
 - ▶ in distributed systems: communication by sending messages,
 - ▶ with remote procedure calls (RPC, RMI, ...)
 - ▶ in shared memory: requirements performed as program functions (eg open, read, write), need for synchronization
- ▶ Example, a file server
 - ▶ client requests: **open, read, write, close**
 - ▶ server responses: data from the file, request execution status (it succeeded, problems ...)



Interaction of equals

- ▶ Here we solve a problem by distributing the work among several "workers", all of whom perform approximately the same amount of work.
- ▶ Workers solve their problem with their work and cooperation.
- ▶ Usually:
 - ▶ we solve a problem that involves the processing of input data,
 - ▶ the data is divided into smaller parts
 - ▶ then the same procedure is carried out on every part of the data (workers).
- ▶ Normally, communication between these procedures is required for the final solution.

Interaction of equals

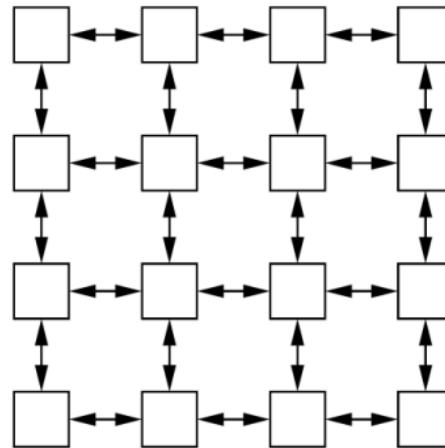
- ▶ Methods of work distribution and interaction:
 - ▶ coordinator, workers



- ▶ circular interaction

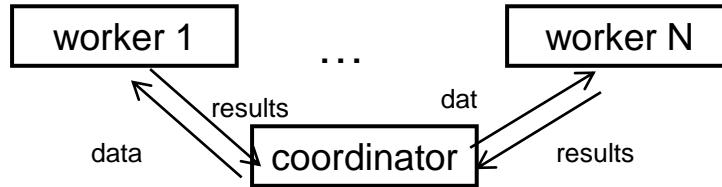


- ▶ grid interaction



- ▶ ...

Coordinator/workers: matrix multiplication



▶ Worker

```
process worker[i = 0 to n-1] {
    double a[n];      # row i of matrix a
    double b[n,n];    # all of matrix b
    double c[n];      # row i of matrix c
    receive initial values for vector a and matrix b;
    for [j = 0 to n-1] {
        c[j] = 0.0;
        for [k = 0 to n-1]
            c[j] = c[j] + a[k] * b[k,j];
    }
    send result vector c to the coordinator process;
}
```

▶ Coordinator

```
process coordinator {
    double a[n,n];    # source matrix a
    double b[n,n];    # source matrix b
    double c[n,n];    # result matrix c
    initialize a and b;
    for [i = 0 to n-1] {
        send row i of a to worker[i];
        send all of b to worker[i];
    }
    for [i = 0 to n-1]
        receive row i of c from worker[i];
    print the results, which are now in matrix c;
}
```

Circular interaction: matrix multiplication



```
process worker[i = 0 to n-1] {
    double a[n];           # row i of matrix a
    double b[n];           # one column of matrix b
    double c[n];           # row i of matrix c
    double sum = 0.0;       # storage for inner products
    int nextCol = i;        # next column of results
    receive row i of matrix a and column i of matrix b;
    # compute c[i,i] = a[i,*] × b[*,i]
    for [k = 0 to n-1]
        sum = sum + a[k] * b[k];
    c[nextCol] = sum;
    # circulate columns and compute rest of c[i,*]
    for [j = 1 to n-1] {
        send my column of b to the next worker;
        receive a new column of b from the previous worker;
        sum = 0.0;
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        if (nextCol == 0)
            nextCol = n-1;
        else
            nextCol = nextCol-1;
        c[nextCol] = sum;
    }
    send result vector c to coordinator process;
}
```

Programming III

Parallel and distributed programming

Processes and synchronization

Janez Žibert, Jernej Vičič UP FAMNIT

Overview (chapter 2)

- ▶ **Processes:**
 - ▶ execution of process commands - indivisible execution
 - ▶ state and intertwining of states of simultaneously running processes
- ▶ **Synchronization**
 - ▶ atomicity of commands,
 - ▶ introducing the **await** command
- ▶ **Examples of parallelization and synchronization**
 - ▶ copy an array between the producer and the consumer
 - ▶ search for the maximum element in the array
 - ▶ search for a cross-section of two tables
- ▶ **Characteristics of running parallel programs**
 - ▶ safe execution,
 - ▶ liveliness of a process,
 - ▶ scheduling of a process,
 - ▶ sorting process execution.

Process: state, atomic operation

- ▶ The **process** is a unit of a program that performs a specific task.
- ▶ The **state** of a process is represented by the values of all the variables used by the process for performing its work at a given time point.
- ▶ Each process implements a specific sequence of commands.
- ▶ Each instruction consists of one or more **operations (executed with no interruption)** that change one state of the process to another.
- ▶ **(Non-breakable)** operation - change one process state to another

Atomic instructions

- ▶ **Atomic instruction** – a sequence of onstructions (passages from state to state) in the execution of a process, which is performed as an indivisible whole.
- ▶ Machine commands, that are implemented directly in the processor architecture (**read, write, swap**) are executed as atomic commands.
- ▶ In our case: implementation in program
 - ▶ `<instructions ;>` - a sequence of instructions that must be executed as an atomic instruction,
 - ▶ therefore: the atomic command must be executed as one indivisible operation of the process,
 - ▶ example: critical areas (in more detail below).

Processes and machine atomic instructions

- ▶ Let's assume the basic model of the computer that we use to implement the processes:
 - ▶ the values of the basic types of variables (**int, char, ...**) are stored as basic memory elements (words) and access them with atomic commands: **read, write**,
 - ▶ operations over values of variables are executed in registers, load into the register, perform the operation (eg add), save the result to the memory (store),
 - ▶ each process has its own set of registers and its own stack. When the process with the dispatcher is changed (according to the scheduler), the whole set of registers and the stack is replaced (the context of the process is replaced),
 - ▶ each intermediate result in the execution of complex process commands is stored in the registers or in the process stack.

Interlacing the states of concurrently running processes

- ▶ Each process executes out a sequence of atomic operations, and causes changes in its states.
 - ▶ Each process has its own course of changing states.
- ▶ Several simultaneously executing processes, each with its own course :
 - ▶ Process 1: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n$
 - ▶ Process 2: $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$
 - ▶ Process 3: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots \rightarrow q_n$
- ▶ Multiple joint execution paths in the execution of concurrent processes:
 - ▶ path 1: $s_0 \rightarrow p_0 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_0 \rightarrow s_2 \rightarrow q_1 \rightarrow \dots$
 - ▶ path 2: $p_0 \rightarrow s_0 \rightarrow q_0 \rightarrow q_1 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2 \rightarrow \dots$

Concurrent execution of processes is non-deterministic!

- ▶ What will be the intertwining of operations between concurrently performed processes is impossible to predict:
 - ▶ each time we have a different history of changing states (operations' sequences),
 - ▶ the sequence of operations can not be repeated => non-deterministic behavior.
- ▶ Example: take a parallel program with n processes and each process executes m (atomic) operations, then we have $(n \cdot m)!/(m!^n)$ possible paths:
 - ▶ $n=3, m=2, 90$ possible program executions
- ▶ Zato tudi:
 - ▶ it is impossible to prove the correctness of the program operation only by testing ("the program is to run and we observe what is happening")

Example of non-deterministic behavior of a parallel program

- ▶ Program:

```
int y = 0; z = 0;  
co x = y + z; || y = 1; z = 2; oc
```

- ▶ Possibility 1:

```
x = y{0} + z{0}; y = 1; z = 2; {x == 0}
```

- ▶ Possibility 2:

```
y = 1; x = y{1} + z{0}; z = 2; {x == 1}
```

- ▶ Possibility 3:

```
y := 1; z := 2; x := y{1} + z{2}; {x == 3}
```

- ▶ Possibility 4:

```
load y{0} to R1; y := 1; z := 2;  
add z{2} to R1{0}; store R1 to x; {x == 2}
```

Critical references and at most once property

- ▶ **Critical reference** in an expression is a reference to a variable in an expression that is changed by another process.
- ▶ **At most once property:**
if $x := e;$
 - ▶ (1) e has at one critical reference and x is not read by another process.
 - ▶ (2) e does not include critical references, x can be read by many processes.

Examples of critical references and at most once property

```
int x = 0, y = 0;  
co x = x + 1; || y = y + 1; oc
```

- ▶ At most once property holds.

```
int x = 0, y = 0;  
co x = y + 1; || y = y + 1; oc
```

- ▶ At most once property holds.

```
int x = 0, y = 0;  
co x = y + 1; || y = x + 1; oc
```

- ▶ At most once property does not hold.

Synchronization

- ▶ Synchronization is essential to ensure the desired operation of parallel programs.
- ▶ **Synchronization is a mechanism by which we can stop and resume processes.**
 - ▶ In this way, we can control the interaction between processes and allow only those states that lead to the final (desired) solution of the programs.
- ▶ Two synchronization mechanisms (controls):
 - ▶ the principle of **mutual exclusion**
 - ▶ we provide access to critical areas to only one process at a time
- ▶ **The conditional synchronization principle**
 - ▶ stop the execution of the process until a certain condition has been met.

Instruction `await` and types of synchronization

- ▶ **<await (B) S;>**
 - ▶ Wait until the condition B becomes true, then immediately execute commands S atomically. Combination of the principle of mutual exclusion and conditional synchronization:
 - ▶ It is performed as an atomic instruction. Condition B is definitely satisfied when the execution of the command S begins. The S commands are atomically executed and definitely end.
- ▶ **<await (B);>**
 - ▶ Wait until the condition B becomes **true**.
 - ▶ Principle of **conditional synchronization**.
- ▶ **<S;>**
 - ▶ Instructions S are executed atomically and definitely end.
 - ▶ Usage: in critical section to ensure **mutual exclusion**.

Examples of `<await (B) s;>`

- ▶ **`<await (s>0) s = s-1;>`**
 - ▶ wait until `s` becomes positive,
 - ▶ reduce `s` for 1
 - ▶ `s` is reduced only if it is positive!!!
(nothing happened in the meantime)
- ▶ **`<await (s>0)>`**
 - ▶ wait until `s` not positive
- ▶ **`<s = s-1;>`** atomic instruction
 - ▶ `s` decrease by 1,
 - ▶ instruction executes atomically: operations load to register (`load`), subtract (`sub`) and write back to memory (`store`) are executed as one “transaction”,
 - ▶ process states that are altered by these operations are invisible to other processes!!Therefore there is no intertwining among these states and all the other processes.

Implementation of `<await (B) S;>`

- ▶ Instruction `<await (B) S;>` is very hard to implement in the full form (very expensive).
- ▶ We will use this instruction as a theoretic instruction in definition of parallel programming concepts:
 - ▶ Concrete implementation will vary from example to example..
- ▶ Special cases:
 - ▶ `< S;> ≡ S;`
 - ▶ In case all instructions in `S` fulfill the at-most-once property.
 - ▶ `<await (B)> ≡ while(not B);`
 - ▶ In case all instructions in `B` fulfill the at-most-once property.

Example <await (B) S;>

- ▶ Parallel program:

```
int x = 1, y = 2, z = 3;  
co x = x + 1;  
|| y = y + 2;  
|| z = x + y;  
|| <await (x > 1) x = 0; y = 0; z = 0;>  
oc
```

- ▶ Does the parallel program end?
- ▶ What are the final values of variables x, y and z?
- ▶ Let's not forget: the addition is not an atomic operation, but we must first read the value from memory into the register, add it, and then write it back to the memory.

The final result of the parallel program

- ▶ Parallel program:

```
int x = 1, y = 2, z = 3;  
co x = x + 1;  
|| y = y + 2;  
|| z = x + y;  
|| <await (x > 1) x = 0; y = 0; z = 0;>  
oc
```

- ▶ Program ends.
- ▶ Possible values for variables are:

```
x == {0}  
y == {0,2,4}  
z == {0,1,2,3,4,5,6}
```

Example of copying the table according to the producer/consumer principle

- ▶ **Task:**
 - ▶ Assume that one program (producer) has a local array of integers $a[n]$.
 - ▶ Copy the contents of the array $a[n]$ to the array $b[n]$ that is available (initialized) in another program (consumer).
 - ▶ The program does not have access to a local array in the other program.
 - ▶ Communication between programs runs through a common (global) variable `buf`.
- ▶ **Solution:**
 - ▶ Producer writes to `buf`, consumer reads from `buf`.
 - ▶ They do this alternately: first, the producer writes the 1st element from the a array, then the consumer reads the `buf` value and writes it to the 1st place in array b , then the producer writes the 2nd element from the a to `buf`, the consumer reads it and writes it to 2. place in array b , etc.
 - ▶ Synchronization is required.
 - ▶ When can we write in the `buf` and when can we read from it?
 - ▶ We do this with two counters, a p counter, which counts the entries in the `buf`, and a counter c that counts reads from `buf`.

Example of copying the table according to the manufacturer/consumer principle

```
int buf, p = 0, c = 0;  
process Producer {  
    int a[n];  
    while (p < n) {  
        <await (p == c);>  
        buf = a[p];  
        p = p+1;  
    }  
}  
process Consumer {  
    int b[n];  
    while (c < n) {  
        <await (p > c);>  
        b[c] = buf;  
        c = c+1;  
    }  
}
```

- ▶ Independence of both processes.
- ▶ Common variable is buf.
- ▶ Synchronization:
 - ▶ mutual exclusion:
 - ▶ access to the buf always takes only one process,
 - ▶ conditional synchronization
 - ▶ the producer is waiting until the buf is ready for re-entry
 - ▶ the consumer waits for the buf to get a new value
 - ▶ execution of synchronization:
 - ▶ with counters p and c condition:
 - ▶ $c \leq p \leq c + 1$

Parallel search for the maximum element in the array

- ▶ **Task:**

- ▶
 - Search for the max element in the array $a[n]$.
Suppose that the elements are positive int numbers.

- ▶ **Solution:**

Select into variable m some value that is smaller than every value in $a[n]$.

Traverse the array a with index $i = 0$ to $n-1$ and for $a[i]$ check if it is greater than m .

Change if true: $m = a[i]$.

Parallel search for the maximum element in the array

▶ Sequential program

```
int m = 0;  
for [i = 0 to n-1]  
    if (a[i] > m) m = a[i];
```

▶ Parallel program

```
int m = 0;  
co [i = 0 to n-1]  
    if (a[i] > m) m = a[i]; oc
```

- ▶ Is this enough?
- ▶ Does the program work correctly?
 - ▶ No.
 - ▶ The processes are not independent.
 - ▶ Each read and write the variable m. So: at the beginning, all processes start and each compares its value with m, because m is smaller than all the values in the table, everyone wants to change the value of m with its value.
 - ▶ Of course, this computer runs sequentially => at the end the m gets the value of the process that is done last.
 - ▶ ***race condition***
- ▶ **Synchronization is needed.**

Parallel search for the maximum element in the array

- ▶ First try:
 - ▶ we make comparison and assignment as an atomic command

```
int m = 0;
co [i = 0 to n-1]
  < if (a[i] > m) m = a[i]; > oc
```
 - ▶ We got rid of the race condition. Why?
 - ▶ What about efficiency?
Same as sequential program, processes execute one at a time, but the order is arbitrary.
- ▶ Observations:
 - ▶ Reading m and a[i] can be performed in parallel for each i,
 - ▶ writing in m must be carried out successively, so we must ensure that the writing is excluded in to this critical variable. - a critical part of the execution.
 - ▶ How to get rid of the race condition?

Parallel search for the maximum element in the array

▶ Second try:

- ▶ we perform a non-atomic comparison and then compare and arrange as an atomic command

```
int m = 0;  
co [i = 0 to n-1]  
  if (a[i] > m)  
    < if (a[i] > m) m = a[i]; > oc
```

- ▶ In the first comparison, it is not necessary to ensure inter-process exclusion because we only read variables.
- ▶ At the second comparison and assignment atomicity is required. This ensures the mutual exclusion of the processes in the assignment. Let's get rid of the race condition.
- ▶ Performance is better than before: it is not necessary to implement an atomic command (which is not cheap!), in case the condition $a[i] > m$. The principle of neutron comparisons and atomic comparisons and events is often used to avoid the problem of competition between processes.

Example: `await` and `if`

- ▶ Look at this example:

```
int x = 0;
co <await (x != 0) x = x - 2 >;
|| <await (x != 0) x = x - 3 >;
|| <await (x == 0) x = x + 5 >;
oc
```

- ▶ (a) Does the program end? If true, what are the possible ending values for `x`. If false, why?
- ▶ (b) Change `await` with `if` and remove `<>`. Now the program finishes. What are the possible values for `x`?

Solution

- ▶ (a) Program finishes. End value of $x = 0$. Last process executes first, then the first two in arbitrary order.
- ▶ (b)

```
int x = 0;  
co if (x != 0) x = x - 2;    //s1  
|| if (x != 0) x = x - 3;    //s2  
|| if (x == 0) x = x + 5;    //s3  
oc
```

Possible paths and final values of x :

conditions S1 and S2 are false, S3	$x == 5$
condition S2 is false; S3, S1;	$x == 3$
condition S1 is false; S3, S2;	$x == 2$
S3; S1 S2	$x == \{0, 2, 3\}$

Example 3: Cross section of two arrays

- ▶ Task:
 - ▶ take two integer arrays $a[1:m]$ and $b[1:n]$, both sorted in ascending order.
 - ▶ (a) Make a sequential program that counts those elements from both arrays that have the same values. So we count how many elements are in the cross-section of both arrays.
 - ▶ (b) Translate sequential program into parallel!

Example 3: Cross section of two arrays

- ▶ Linear time:

- ▶ sequential program:

```
int i = 0, j = 0, count = 0;
for (i=0; i < m; i++)
    for (j=0; j < n; j++)
        if (a[i] == b[j]) count++;
```

- ▶ sequential program better solution

```
int i = 0, j = 0, count = 0;
while (i < m && j < n) {
    if (a[i] < b[j]) i++;
    else if (a[i] > b[j]) j++;
    else {count++; i++; j++;}
}
```

- ▶ parallel program:

```
int i = 0, j = 0, count = 0, A, B;
while (i < m && j < n) {
    A = a[i]; B = b[j];
    if (A < B) i++;
    || if (A > B) j++;
    || if (A == B) {count++; i++; j++;}
    oc;
}
```

Kthere is no need for synchronization because the exclusive conditions.

At each step, exactly one process will be executed, this is the only place where the change in the values can happen ...

What did we learn?

- ▶ Synchronization is required when multiple processes access common variables (reads or writes to common variables).
- ▶ By atomic execution of commands, we ensure the mutual exclusion of interlacing the execution of commands between processes. Ensure mutual exclusion.
- ▶ The principle of non-atomic comparisons and atomic comparisons (test and test-and-set) is often used to avoid the problem of competition between processes:
 - ▶ in the event that several conditional assignments are carried out simultaneously on the basis of the same condition.
 - ▶ Example:
 - ▶ Search for the maximum element in the table

Characteristics of running parallel programs

- ▶ In a parallel program, two or more simultaneous processes are involved. Execution of processes is intertwined. The order of the states of the parallel program is called state path. A setting can have multiple paths.
- ▶ Questions:
 - ▶ How to ensure the correctness of the implementation?
 - ▶ How to ensure that the program always ends?
 - ▶ How to ensure that the program always ends properly?

Characteristics of running parallel programs

- ▶ Definition:
- ▶ The property of the (parallel) program is a feature P , which is valid for every state of all possible paths of execution of the program.
- ▶ Two important properties ensure the correct operation and stopping of the execution of programs:
- ▶ *safety properties:*
 - ▶ There is no bad state in the execution path.
- ▶ *liveness properties:*
 - ▶ In some moment the execution goes into a good state.

“Good” and “bad” process states

- ▶ Suppose that P is a property:
 - ▶ "Good" process state - state in which property P is valid.
 - ▶ "Bad" state of the process - a state in which the P property is not valid.
- ▶ Safe program execution for P :
 - ▶ The P property is valid in every state in all possible program execution paths.
 - ▶ The P property is globally unchanged according to the program being executed - invariant.
- ▶ Liveliness of the execution od the program for P :
 - ▶ In all possible stages of program execution, we get to the state where P is valid.

Examples of process execution properties

- ▶ Safe execution property
 - ▶ **Partial correctness**
 - ▶ If we assume that the program ends, comes to the final state of implementation, then this final state correctly solves the task that the program solves.
 - ▶ **Mutual exclusion** of the execution in critical sections
 - ▶ Ensuring that at any time of the program execution at most one process is executed in critical sections.
 - ▶ **Preventing deadlock**
 - ▶ Deadlock: process is waiting for a condition to become true, but this never happens.
- ▶ Liveliness property
 - ▶ **Program termination**
 - ▶ All possible program execution paths include a final number of states. The program always ends.
 - ▶ **Ensuring entry into the critical area**
 - ▶ In the event that a process at the time of the execution requires access to a critical section, this access must be granted at some point.
- ▶ **Full program correctness:** stopping + partial correctness
 - ▶ The program always ends, the final state gives the correct answer to the task that we are solving.

Example: copy array producer/consumer

```
int buf, p = 0, c = 0;  
process Producer {  
    int a[n];  
    while (p < n) {  
        <await (p == c);>  
        buf = a[p];  
        p = p+1;  
    }  
}  
process Consumer {  
    int b[n];  
    while (c < n) {  
        <await (p > c);>  
        b[c] = buf;  
        c = c+1;  
    }  
}
```

- ▶ Preventing the deadlock - proof: by contradiction (Reductio ad absurdum)
- ▶ Globaly unchangeable property:
- ▶ $PC: c \leq p \leq c+1$
- ▶ await in producer:
 $WP: PC \wedge (p < n) \wedge (p \neq c)$
- ▶ await v consumer:
 $WC: PC \wedge (c < n) \wedge (p \leq c)$
- ▶ Deadlock:
 $WC \wedge WP = (p \neq c) \wedge (p == c) = \text{false}$
- ▶ Deadlock is impossible in this case.

Liveliness of program execution and process scheduling

- ▶ Liveliness property is based on the assumption that someone/something guarantees the execution of the program.
- ▶ How the program is executed and in which order the processes are executed in the program determines the process scheduler.
- ▶ What do we expect from the process scheduler?
- ▶ That sorting is "fair". This means that the execution of individual (atomic) instructions of the process must occur sometime.

Scheduling types

```
bool continue = true;  
co while (continue);  
  || continue = false;  
oc
```

- ▶ Example scheduling type: the processor executes a process all the way until the process finishes, then this program can never end:
 - ▶ first process starts and never ends
 - ▶ **livelock** – program is runs, but it does not progress.
- ▶ If the second process gets some running time, the program finishes.
- ▶ This depends on the type of scheduling.
- ▶ Fair way to schedule unconditional instructions:
Each atomic instruction that does not include a condition will sooner or later finish.
- ▶ Example: round robin on single-procesor system, parallel execution on multiprocessor system.

Scheduling method with conditional atomic commands

- ▶ **Weakly fair scheduling:**
 - ▶ It is fair in the case of unconditional atomic commands.
 - ▶ In the case of conditional atomic commands: the execution of a conditional atomic command is guaranteed if the condition of the conditional atomic command becomes true and remains true till the execution of the command.
 - ▶ Example: round-robin
- ▶ **Strong fair scheduling:**
 - ▶ It is fair in the case of unconditional atomic commands.
 - ▶ In the case of conditional atomic commands: the execution of a conditional atomic command is guaranteed if the condition of the conditional atomic command becomes true infinite number of times (it does not necessarily remain true to the execution, it is important that it is infinitely true).
 - ▶ Not possible to implement in practice.

Example: scheduling of razvrščanja pogojnih atomarnih ukazov

```
bool continue = true, try = false;
co while (continue) {try = true; try = false;}
  || <await (try) continue = false;>
  oc
```

- ▶ In the case of strictly fair scheduling, the parallel program will end.
- ▶ In the case of weakly fair scheduling, the parallel program does not end.
- ▶ Why?

Programming III

Parallel and distributed programming

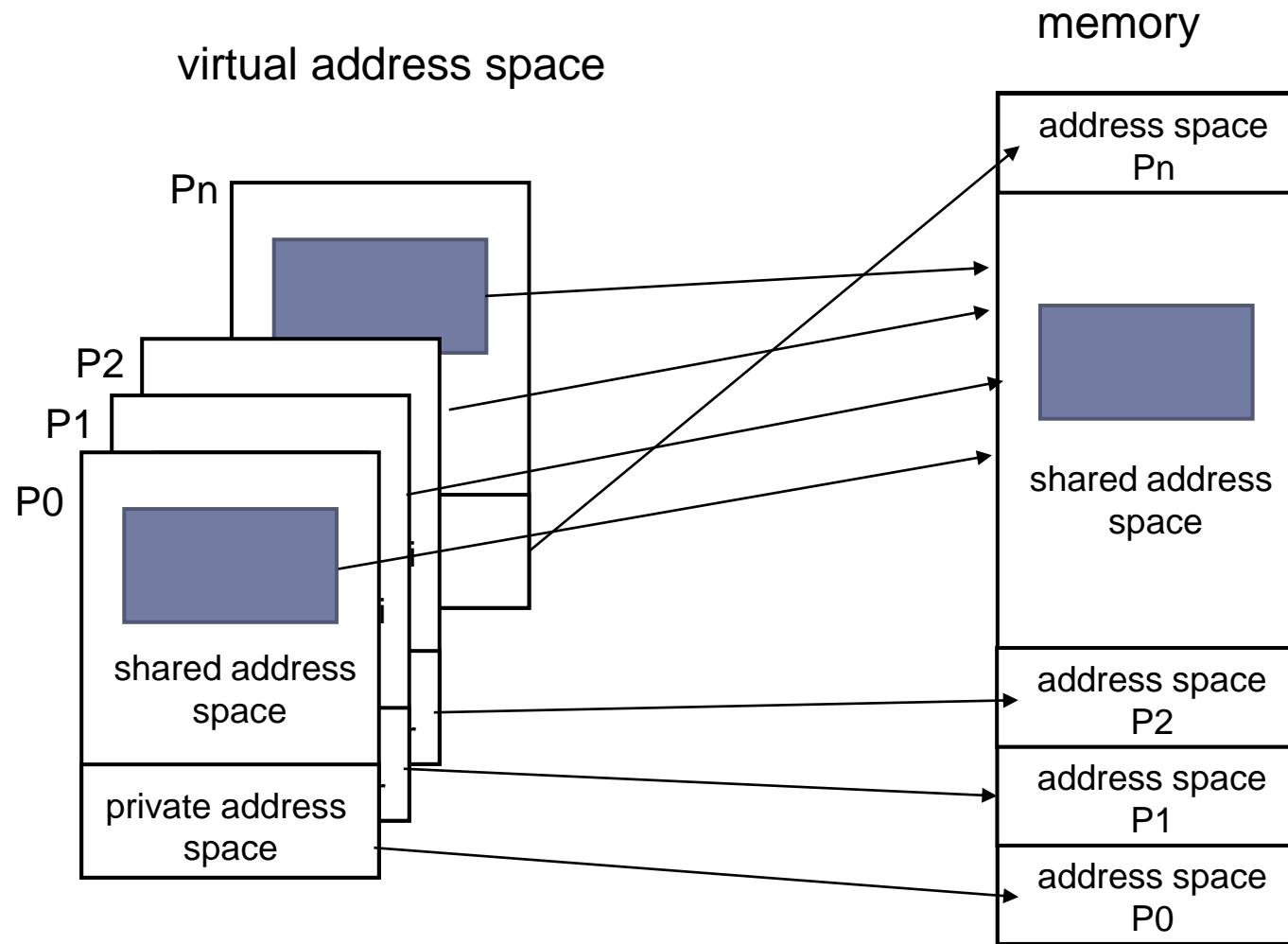


Critical sections.
Locks.

Overview (chapter 3.1-3.3)

- ▶ Critical sections:
 - ▶ Access to the CS
- ▶ Locking principle - locks
 - ▶ spinlocks,
 - ▶ establishing the order of access to critical areas:
 - ▶ Peterson algorithm,
 - ▶ waiting list algorithm,
 - ▶ baker's algorithm.
- ▶ Implementation of the **await** with locks.

Shared memory address space



Shared variables

- ▶ Example:

```
int x = 0;  
co x++; || x++; oc
```

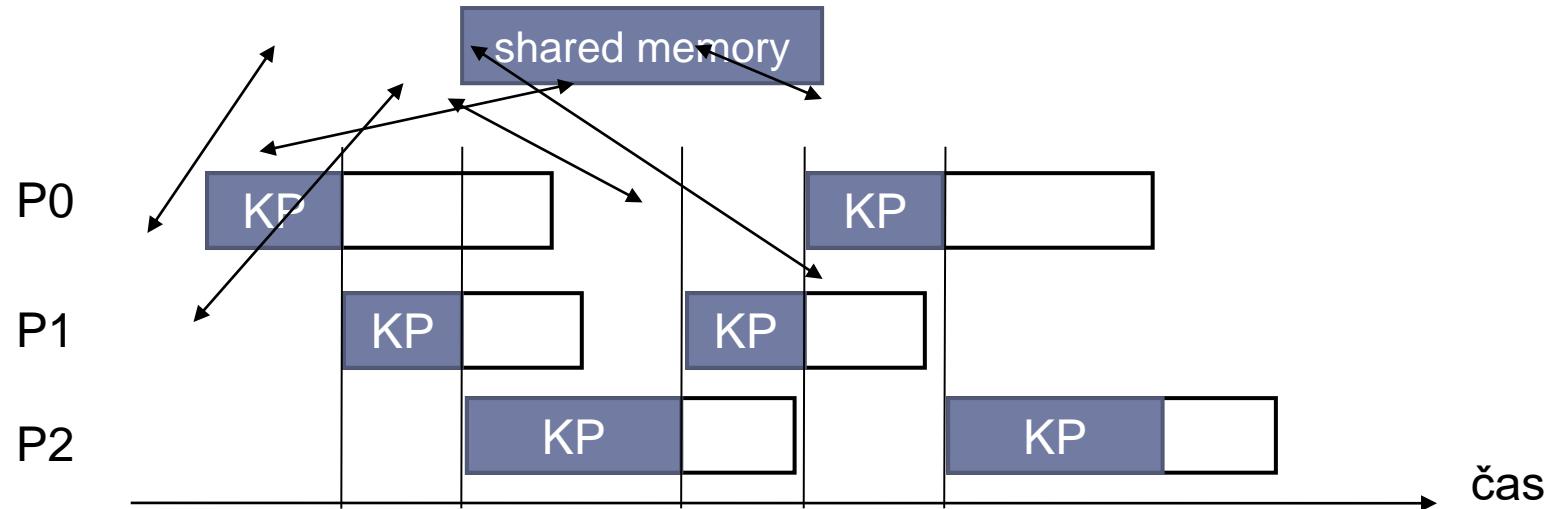
- ▶ What are the possible final values for **x**? **x == {1, 2}**
- ▶ Why **x == 1**?

- ▶ Remember:

```
P0 : ...; load x to REG; inc REG, store REG to x; ...  
P1 : ...; load x to REG; inc REG, store REG to x; ...
```

Program's critical section

- ▶ The sequence of commands that access common variables is called critical section of the program.
- ▶ Critical section of a program can only be executed by one process at a time, so the sequence of critical commands must be performed indivisibly as an atomic command.
- ▶ **The problem of accessing critical sections:** The mechanism for regulating the access of several simultaneous processes to the critical section of the program. Entering and executing a critical section must be guaranteed only for one process at the same time.



Program's critical section

```
process P[i=0 to n-1] {
    while (true) {
        CSentry;           //entry protocol
        critical section; //critical section
        CSexit;           //exit protocol
        non-critical section;
    }
}
```

- ▶ What should we do to implement the entry and exit protocols?
 - ▶ **Mutual exclusion of CS**
 - ▶ A maximum of one process can enter the CS at the same time, perform the CS and exit the CS.
 - ▶ **Preventing deadlock**
 - ▶ At least one of the processes competing to enter the CS will be enabled.
 - ▶ **Preventing the retention of CS**
 - ▶ If one process is awaiting entry into the CS, but there is no other process, then this process can begin immediately the CS.
 - ▶ **Enabling access to CS**
 - ▶ Every process that is waiting to enter the CS will enter eventually.

Locking the CS access - locks

- ▶ What are the options for executing the CS?
 - ▶ Either one of the processes is executing the CS, or none.
 - ▶ We use the lock principle: when we enter the CS we lock the CS, when we exit, unlock CS.
- ▶ **Lock** - a logical variable that is true when one of the processes in the CS; and it is false when there is no process in the CS:
 - ▶ Locking and unlocking the lock is performed in the entry and exit protocol (CSentry, CSexit).
- ▶ We can use one or more locks:
 - ▶ two extremes:
 - ▶ one lock for all shared variables; can be ineffective, we lose parallelism
 - ▶ each shared variable has its own lock;
 - ▶ high level of parallelism, high burden of synchronization,
 - ▶ therefore, we always weigh the degree of parallelism and the burden of synchronization.

Lock implementation

```
bool lock = false;  
process CS1 {  
    while (true) {  
        (await (!lock) lock = true;) /* entry */  
        critical section;  
        lock = false; /* exit */  
        noncritical section;  
    }  
}  
process CS2 {  
    while (true) {  
        (await (!lock) lock = true;) /* entry */  
        critical section;  
        lock = false; /* exit */  
        noncritical section;  
    }  
}
```

- ▶ Solution for 2 processes.
- ▶ Can this be abstracted to n processes? How??
- ▶ How can we implement the `await` instruction?

Lock implementation in the `await` statement

- ▶ There are machine instructions (which are executed as atomic commands on a computer) that can help us perform the lock in the `await` command.
- ▶ Examples:
 - ▶ TS (Test and Set)

```
bool TS (bool lock) {  
    < bool initial = lock; // save the starting value  
    lock = true;           // set lock to true  
    return initial; >     // return starting value
```

- ▶ FA (Fetch and Add)

```
int FA (int var, int incr) {  
    < int tmp = var;       // save the var value  
    var = var + incr;     // increment var for incr  
    return tmp; >         // return starting value of var
```

- ▶ other

CS implemented with TS lock

```
bool lock = false;           /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (TS(lock)) skip; /* entry protocol */
        critical section;
        lock = false;          /* exit protocol */
        noncritical section;
    }
}
```

▶ Check the 4 CS properties:

- ▶ mutual exclusion: YES
- ▶ deadlock prevention: YES
- ▶ preventing the retention of execution: YES
- ▶ ensuring the CS enter: DPARTIAL

(in the case of strictly fair scheduling YES,
otherwise NO, in practise YES.)

Spin lock

```
bool lock = false;                      /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (TS(lock)) skip;      /* entry protocol */
        critical section;
        lock = false;              /* exit protocol */
        noncritical section;
    }
}
```

- ▶ Special lock implementation:
spin lock:
- ▶ processes keep spinning in a loop and wait for the lock to open.

The weakness of the TS lock in a loop

```
    bool lock = false;          /* shared lock */
    process CS[i = 1 to n] {
        while (true) {
            while (TS(lock)) skip; /* entry protocol */
            critical section;
            lock = false;          /* exit protocol */
            noncritical section;
        }
    }
```

- ▶ Variable **lock** is shared among all processes.
- ▶ The **while** loop writes to the **lock** variable in every loop.
- ▶ Assuming that each process is running on its processor and the lock is stored in the cache of this process (for faster access), after each TS execution, the common memory - all the caches of other processors must be refreshed = **memory contention**
- ▶ The price of the **TS** is much higher than simple reading of the shared **lock**.

Improvement: TTS

```
bool lock = false;           /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (lock) skip;      /* entry protocol */
        while (TS(lock)) {
            while (lock) skip;
        }
        critical section;
        lock = false;           /* exit protocol */
        noncritical section;
    }
}
```

- ▶ Implementation: Test and Test & Set
 - ▶ The TS command is only implemented if we have the opportunity to make progress.
 - ▶ Reduces the number of TS operations and, consequently, the requirement to continually refresh the shared memory.
 - ▶ For this reason, we only read the value, which is cheaper than TS.
 - ▶ The exit protocol only puts lock in its original state.

Establishing the order of execution of the CS

- ▶ Spinlocks:
 - ▶ an effective way of ensuring the implementation of the CS,
 - ▶ how it works:
 - ▶ processes are waiting in a loop to unlock the lock,
 - ▶ the process that first detects the unlocked lock goes into the execution of the CS,
 - ▶ the others loop forward.
 - ▶ what can happen is:
 - ▶ a process that just executed the CS can once again regain the possibility of entering CS,
 - ▶ That's not fair.
 - ▶ Ranking of processes before the lock:
 - ▶ processes are placed in a queue in front of the lock and waiting to be selected
 - ▶ when the lock is unlocked, the process that is the first in queue goes into CS,
 - ▶ several possible approaches.

Peterson algorithm

- ▶ Suppose we have two processes that access the CS.
- ▶ Consider:
 - ▶ We set the lock to enter CS.
 - ▶ Remember, which of the processes was last executed by the CS.
 - ▶ If the lock is unlocked and the process was not the last in the execution of the CS, it can begin to execute the CS.
- ▶ Idea:
 - ▶ The new "lock" to enter the CS is thus composed of two parts: the logical variable which is our "old" lock and the variable which manages the order of entry.

Peterson algorithm

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
        last = 1; in1 = true;      /* entry protocol */
        <await (!in2 or last == 2);>
        critical section;
        in1 = false;             /* exit protocol */
        noncritical section;
    }
}
process CS2 {
    while (true) {
        last = 2; in2 = true;      /* entry protocol */
        <await (!in1 or last == 1);>
        critical section;
        in2 = false;             /* exit protocol */
        noncritical section;
    }
}
```

- ▶ Process CS1:
 - ▶ process waits to enter CS,
 - ▶ if process CS2 is executing CS (in2==true)
 - ▶ if it just came from CS and CS2 is next (last==1).
- ▶ Process CS2:
 - ▶ vice versa.
- ▶ The program meets all four features of the CS.

Peterson algorithm with no atomic instructions

- ▶ Why is it possible to even if we have 2 critical references in the entry protocols?
 - ▶ Condition: `<await (!in2 or last==2)>`
- ▶ Process CS1:
 - ▶ Suppose the process is waiting to enter CS. Denimo, da je pogoj == **true**. It can enter the CS.
 - ▶ If **in2==false**, se lahko zgodi, da je v naslednjem trenutku **in2** že **true**. V tem primeru je proces CS2 že postavil **last** na 2. Zato je pogoj še vedno **true**, čeprav je **in2** spremenil svojo vrednost.
 - ▶ Če je **last == 2**, pogoj še vedno **true**. **last** ostane 2, dokler CS1 ne izvede KP.
 - ▶ Torej: če proces CS1 misli, da je pogoj **true**, je dejansko pogoj **true** (lahko izvede v vstop v KP).
- ▶ Symmetric for CS2.
- ▶ So we do not need atomic conditional instructions, we can negate the condition in the while loop.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
        last = 1; in1 = true;      /* entry protocol */
        while (in2 and last == 1) skip;
        critical section;
        in1 = false;             /* exit protocol */
        noncritical section;
    }
}
process CS2 {
    while (true) {
        last = 2; in2 = true;      /* entry protocol */
        while (in1 and last == 2) skip;
        critical section;
        in2 = false;             /* exit protocol */
        noncritical section;
    }
}
```

The process of waiting tickets

- ▶ Peterson algorithm works well for 2 processes,
 - ▶ The generalization to n processes is quite complicated. It is necessary to keep the locking array `in[i]` and the visit array `last[n]` for each process separately. Intuitively the solution is quite incomprehensible. There is a lot of synchronization between shared variables.
- ▶ An alternative:
 - ▶ simulate the waiting queue at the doctor's office (in the shop, itd.)
 - ▶ when the patient enters the waiting room, he takes a leaf with a number that is the last used number + 1,
 - ▶ the patient waits until his number appears on a display.
 - ▶ **Process of waiting tickets.**

The process of waiting tickets

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        <turn[i] = number; number = number + 1;>
        <await (turn[i] == next);>
        critical section;
        <next = next + 1;>
        noncritical section;
    }
}
```

- ▶ The essential requirement is that each process gets its own number.
 - ▶ We are on the safe side: we use atomic commands.
 - ▶ Non-atomic implementation:
 - ▶ 1. atomic instruction: problems; we have to write to a new variable and increment the first variable
 - ▶ can be solved with a new CS and a lock, but we do not get a fair solution.
 - ▶ 2. atomic instruction: `while (turn[i] != Next) skip;`
 - ▶ 3. atomic command: we can drop `<>` because the output protocol can be executed in only one process
 - ▶ Another problem: Counters number and next can grow indefinitely rising across all boundaries.
-
- ▶ 19 Parallel and distributed programming, Janez Žibert, Jernej Vičič UP FAMNIT

The process of waiting tickets with no atomic instructions

- ▶ Machine (atomic) instruction FA (Fetch and Add)

```
int FA (int var, int incr) {
    < int tmp = var;          // save starting value of var
    var = var + incr;        // povecaj var za incr
    return tmp; >           // return the starting value of var
}
```

- ▶ waiting tickets with FA instruction

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = FA(number,1);      /* entry protocol */
        while (turn[i] != next) skip;
        critical section;
        next = next + 1;           /* exit protocol */
        noncritical section;
    }
}
```

Bakers algorithm

- ▶ What if we do not have the FA command available?
- ▶ Need to implement a new CS and ensure mutual exclusion with locks, which leads to an unfair solution to the waiting room.
- ▶ Alternative procedure: Bakers algorithm a procedure similar to that of a waiting list, but the patient (in our case, the buyer at baker store) does not receive a ticket with a number which is one larger than the number of the last patient (buyer), but one that is greater than the largest of all patients numbers in lobby (customers in the shop). A patient (buyer) with the smallest number of patients (customers) in the waiting room is accepted (served).
- ▶ difference with the previous procedure: we do not run global counters, but compare the numbers between the current numbers that are available.

Bakers algorithm

```
int turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
    while (true) {
        <turn[i] = max(turn[1:n]) + 1;>
        for [j = 1 to n st j != i]
            <await (turn[j] == 0 or turn[i] < turn[j]);>
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

- ▶ Each process receives a number which is one greater than the maximum process number among all the processes waiting to enter the CS.
- ▶ In the CS, the process that has the smallest number among all the processes waiting can enter.
- ▶ All four properties are met.

Bakers algorithm with no atomic instructions

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
        for [j = 1 to n st j != i]
            while (turn[j] != 0 and
                   (turn[i],i) > (turn[j],j)) skip;
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

$$(a,b) > (c,d) \iff \begin{cases} a > c & \text{if } a > c \text{ or } a = c \text{ and } b > d \\ \text{false} & \text{otherwise} \end{cases}$$

- ▶ **max** is not executed atomically, two processes could get the same number. in this case the priority is selected by the process number.
- ▶ ANALOGY: two buyers that got the same number are server in the birth date order.

Bakers algorithm : example 1

Why?

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
        for [j = 1 to n st j != i]
            while (turn[j] != 0 and
                   (turn[i],i) > (turn[j],j)) skip;
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

- ▶ two processes without **turn[i]=1**:

Process CS1

```
turn[2] == 0
turn[1] = 1      // max+1
turn[2] == 1
CS1 enters in CS // (1,1) < (1,2)
```

Process CS2

```
turn[1] == 0
turn[2] = 1      //max + 1
CS2 enters in CS //turn[1] == 0
```

Both processes are in CS.

Process 2 overtook process 1 (race condition).

Bakers algorithm : example 2

Why?

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
        for [j = 1 to n st j != i]
            while (turn[j] != 0 and
                   (turn[i],i) > (turn[j],j)) skip;
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

- ▶ two processes with **turn[i]=1**:

Process CS1

```
turn[1] = 1 //NOVO!
turn[2] == 0

turn[1] = 2 // max+1
turn[2] == 2
CS1 enters CS // (2,1) < (2,2)
```

Process CS2

```
turn[2] = 1 //NOVO!
turn[1] == 1 // ne 0, kot prej
turn[2] = 2 //max + 1
CS2 DOES NOT enter CS //turn[1] < turn[2]
```

Process CS1 enters CS. Process 2 overtook process 1, but cannot enter CS.

When process CS1 exits CS, it sets turn[1]=0, which means that CS2 can enter CS.

Summary

- ▶ Spin locks:
 - ▶ effective if there are not too many processes (memory synchronization problem),
 - ▶ implemented with atomic machine instructions (TS, combination TTS, FA)
 - ▶ fair dispatching is not guaranteed,
 - ▶ it could happen that the process exiting the CS reenters the CS,
 - ▶ it is not possible to enter CS for all processes waiting to enter in the event of weak fairness of the process scheduler.
- ▶ Locks that keep track of the entering order:
 - ▶ the lock keeps track of the entering order of the processes,
 - ▶ used where we need “fair ” (partly controlled) access to CS.
 - ▶ Algorithms:
 - ▶ Peterson,
 - ▶ waiting list principle,
(needs FA instruction fo fair execution),
 - ▶ baker’s algorithm.

Example: search for the max element in an array

- ▶ Parallel program with atomic instruction:

```
int a[n];
int m = 0;
...
co [i = 0 to n-1]
  if (a[i] > m)
    <if (a[i] > m) m = a[i]; >
oc
```

- ▶ Implementation with lock using **await**

```
int a[n];
int m = 0;
bool lock = false;
...
co [i = 0 to n-1]
  if (a[i] > m)
    <await (!lock) lock = true;>
    if (a[i] > m) m = a[i];
    lock = false;
oc
```

- ▶ Implementation with a spinlock with machine TS

```
int a[n];
int m = 0;
bool lock = false;
...
co [i = 0 to n-1]
  if (a[i] > m)
    while (TS(lock)) continue;
    if (a[i] > m) m = a[i];
    lock = false;
oc
```

Implementation of the `await` with locks in CS

- ▶ The atomicity of the CS implementation is assured by mutually excluding the processes that perform CS.
- ▶ This is achieved by an appropriate entry and exit protocols of the CS:

```
CSenter;  
kritično področje;  
CSexit;
```

- ▶ one of the mechanisms: locks
- ▶ How can we use this mechanism for the implementation of:
 - ▶ `< S;>`
 - ▶ `< await(B) ; >`
 - ▶ `< await(B) S;>`
- ▶ Atomic instructions in `< >` can be observed as CS.
- ▶ Shared variables in `B` and `S` must be shielded by locks.

Implementation of the `await` with locks in CS

Atomic instruction	Implementation
<code>< S ; ></code>	<code>CSenter;</code> <code>S ;</code> <code>CSexit;</code>
<code>< await(B) ; ></code>	<code>CSenter;</code> <code>while (!B) { CSexit; Delay; CSenter; } ;</code> <code>CSexit;</code>
<code>< await(B) S ; ></code>	<code>CSenter;</code> <code>while (!B) { CSexit; Delay; CSenter; } ;</code> <code>S ;</code> <code>CSexit;</code>

In general, the commands in B and S can have several shared variables.

Why: `CSenter;`

```
  while ( !B ) continue;  
  S ;  
  CSexit;
```

is this not good enough? When is this not OK?

Different types of synchronization of CS

▶ *Busy waiting*

- ▶ the process is waiting for synchronization (to enter the CS) in the loop, until the condition for synchronization (for entering the CS) is fulfilled,
- ▶ example: locks
- ▶ Weaknesses:
 - ▶ waiting in the loop "steals" processor time

▶ *Blocking wait:*

- ▶ the process waiting for synchronization (to enter the CS) is stopped - it does not work at all,
- ▶ it is awoken from sleep by another process,
- ▶ implementation in the OS.
- ▶ Two mechanisms:
 - ▶ conditional variables (built-in monitors),
 - ▶ semaphores.

Condition variable

- ▶ *Condition variable is a data structure that is made of a queue of stopped processes and operations on this queue that manipulate processes.*
- ▶ Operations:
 - ▶ lock a process (the process stops and is put into a waiting queue)
 - ▶ send a signal to awaiken – resume execution of a process
 - unlock (take the process from waiting queue and change its status to executed)
 - ▶ other common operations over the queue:
 - ▶ check if the queue is empty,
 - ▶ find the element qith the lowest priority (whatever that is).

Operations on condition variables

- ▶ Declaration of a condition variable **cond cv**
- ▶ Operations:

wait (cv, lock)	release the lock and put process at the end of the waiting queue
signal (cv)	awaken the process that is at the begining of the queue
empty (cv)	true , if the queue is empty

- ▶ Implementation
 - ▶ monitors
 - ▶ in the Pthreads library

Signalization types in `cond cv`

- ▶ What to do if a process sends a signal for execution to another process?
 - ▶ Two processes could continue the execution (but only one can). Which one?
- ▶ Signal and Continue (SC)
 - ▶ process, that sends the signal continues with execution,
 - ▶ process, that receives the signal waits for execution,
 - ▶ process, that receives the signal is awaikened and put into the waiting queue.
- ▶ Signal and Wait (SW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put into the waiting queue.
- ▶ Signal and Urgent Wait (SUW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put at the start of the waiting queue.

Programming III

Parallel and distributed programming

Barriers. Flags.

Janez Žibert, Jernej Vičič UP FAMNIT

Overview (chapter 3.4-3.6)

- ▶ **Barriers:**
 - ▶ barrier with a counter,
 - ▶ barriers with flags and coordinator,
 - ▶ possible barrier settings:
 - ▶ asymmetric layout,
 - ▶ binary tree,
 - ▶ symmetrical layout of bulkheads
- ▶ **Data parallelism**
 - ▶ example: differential equations with Jakobijevo method,
 - ▶ *bag of tasks principle.*

Coordination points and iterative procedures

- ▶ In the iterative processes, we usually perform some calculations over the data in several steps:
 - ▶ that is:
 - ▶ go through the data and perform some calculations (1st iteration),
 - ▶ second pass through the data using the information from the first step (2nd iteration), etc.
 - ▶ So we have:
 - ▶ data loop (in each step we go through all data)
 - ▶ time loop (iteration steps)
 - ▶ parallelization: usually data loops, but usually no time loops, the calculations depend on the previous iteration step.

Coordination points and iterative procedures

- ▶ The need to coordinate the processes of the current step so that we can progress to the next step of the process, eg:
 - ▶ various numerical iterative processes that converge to some sought-after solution:
 - ▶ we calculate from each input data a new value that becomes the input data for the next calculation step.
 - ▶ In this case:
 - ▶ to progress to the next iteration step, we must wait until all processes that count new data in the current step are finished.
 - ▶ We say that the processes of the current step have to reach some point so that the program can continue.
 - ▶ We call such points the points of coordination or barrier of the program.

Barriers

- ▶ **Barrier** is a point in the program that all processes must reach in order for the program to continue.

```
bool done = false;
process P[i = 0 to n-1] {
    while (!done) {
        instructions to solve task i;
        Barrier(i, n)           // wait till all tasks are finished
    }
}
```

- ▶ Barrier setting architectures:
 - ▶ centralized: barriers with counters, flags (coordinator),
 - ▶ decentralized: tree shaped, symmetric barriers
- ▶ Barrier implementations:
 - ▶ locks, flags, counters – busy wait,
 - ▶ locks with condition variables – stop wait,
 - ▶ semaphores – stop wait.

Barrier with counter

- ▶ Use common counter that counts how many processes reached the barrier.

```
int count = 0;
Barrier(int n) {
    < count = count + 1; >    //somebody came here
    while (count < n);        //wait till everybody arrived here
}
```

- ▶ This can be done if we have machine instruction FAFA (fetch and add):

```
int count = 0;
Barrier(int n) {
    FA(count, 1); //somebody came here
    while (count < n); //wait till everybody arrived here
}
```

- ▶ Machine instruction FA:

```
int FA (int var, int incr) {
    < int tmp = var;
    var = var + incr;
    return tmp; >
}
```

Weaknesses of the barrier with counter

- ▶ In each step of the iteration we have to set the counter to 0.
 - ▶ This can be made in the `Barrier` function when all the processes reach the barrieršele, ko vsi procesi prispejo do pregrade:
 - ▶ Use two counters, one is set at the beginning to 0, the other on the number of processes,
 - ▶ the first counter increases, the other decreases by one. When all processes arrive at a barrier, we replace the roles of the counters for the next iteration.
- ▶ All shared counters must be incremented/decremented atomically. We need special instructions for this.
- ▶ Memory contention:
 - ▶ Every time the counter is changed, all processes must refresh the local copies of the counter variable in the cache. In the worst case, all $n-1$ processes must do this.
 - ▶ That's why it's suitable when there are few processes.

Flag signalization principle

- ▶ Let's have a flag:
 - ▶ flag signals an event
 - ▶ when the flag is raised, the event occurred (for example, we came to the barrier)
 - ▶ when the flag has been dropped, the event has not yet happened.
 - ▶ Signalization with flags in a barrier:
 - ▶ a process enters the barrier and raises its flag, this signalizes the other processes to come into the barrier.
 - ▶ When all processes have raised their flags (they are all in the barrier), they can continue to work.
- ▶ Basic principle of flag synchronization:
 - ▶ The process that is waiting for the second process to raise its flag (and thus inform something) can drop the flag to this process.
 - ▶ The flag must not be raised if it is not previously dropped.
- ▶ An example of using one flag for two processes:
 - ▶ process 1 raises the flag for process 2 (sends a signal),
 - `while (flag) continue; flag = 1;`
 - ▶ process 2 drops the flag (receives the signal):
 - `while (!flag) continue; flag = 0;`

Barrier with flags usage

- ▶ Idea:
 - ▶ Use an array of flags **arrive**[1:n], that mark the arrivals to the barrier (one flag for each process):
 - ▶ Each process that comes to the barrier says: I arrived. The flag for the process i is raised: **arrive**[i] = 1.
 - ▶ Use an array of flags **continue**[1:n] to indicate when the process can continue to run (one flag for each process).
 - ▶ Condition: All flags in **continue** are set to 1 if all flag flags in **arrive** are already set to 1.
 - ▶ We introduce an additional process - a coordinator waiting for all flags in **arrive** to be set to 1, and then set the flags in **continue** to 1.
 - ▶ The process continues to run when **continue**[i] == 1.

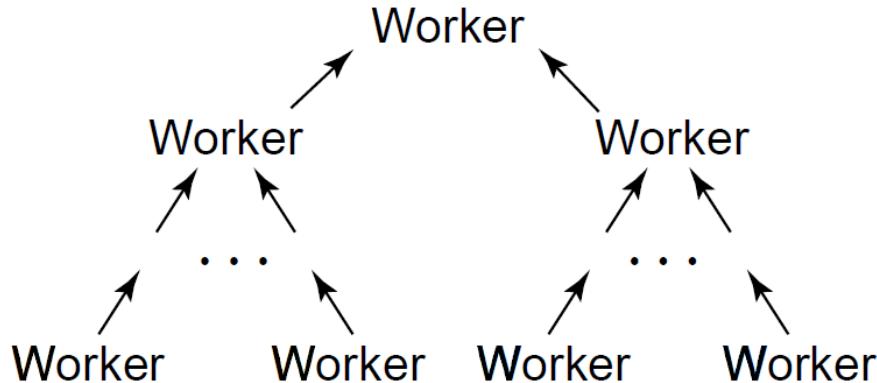
Barrier with flags and coordinator

```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        arrive[i] = 1;
        <await (continue[i] == 1);>
        continue[i] = 0;
    }
}
process Coordinator {
    while (true) {
        for [i = 1 to n] {
            <await (arrive[i] == 1);>
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}
```

Disadvantages of the barrier with the coordinator

- ▶ First:
 - ▶ Additional process (additional processor) – coordinator, $2n$ flags
- ▶ Second:
 - ▶ usually: workers have almost equal load and finish at almost the same time,
 - ▶ coordinator waits in a loop (busy wait) for processes to reach the barrier (loosing parallel time).

Organization of processes and barriers in a binary tree



- ▶ Each node in a binary tree carries out the work of one worker - it executes one process.
- ▶ The work of the coordinator is distributed among workers - the nodes of the tree.
 - ▶ Flags **arrive** are transmitted up the tree.
 - ▶ The **continue** flags are passed down the tree.
- ▶ Synchronization:
 - ▶ the worker in the node is waiting for the **arrive** flags of its children to become 1 when the local work is finished it communicates to the parent node,
 - ▶ when the children of the root node set the flags **arrive** to 1, this means that all workers have finished their work, when the root node finishes the work it communicates to its children to continue their work.

Organization of the processes and barriers in a binary tree

```
leaf node L:  arrive[L] = 1;
              ⟨await (continue[L] == 1);⟩
              continue[L] = 0;

interior node I:  ⟨await (arrive[left] == 1);⟩
                  arrive[left] = 0;
                  ⟨await (arrive[right] == 1);⟩
                  arrive[right] = 0;
                  arrive[I] = 1;
                  ⟨await (continue[I] == 1);⟩
                  continue[I] = 0;
                  continue[left] = 1; continue[right] = 1;

root node R:  ⟨await (arrive[left] == 1);⟩
                  arrive[left] = 0;
                  ⟨await (arrive[right] == 1);⟩
                  arrive[right] = 0;
                  continue[left] = 1; continue[right] = 1;
```

Properties

- ▶ No coordinator – one less process.
- ▶ The number of flags is the same: $2n$
- ▶ Waiting for the next step of the iteration is no longer $O(n)$ as it was in the case of the coordinator, but $O(\log n)$.
- ▶ There is an increase in the communication. Number of signals is increased.
- ▶ The work is no longer equally distributed among workers:
 - ▶ inner nodes do more work than the root and leaves.
 - ▶ **asymmetric work distribution.**

Symmetrical layout of barriers between processes

- ▶ Idea:
 - ▶ Define a barrier between two processes.
 - ▶ Use this barrier to implement a barrier between n processes in several phases, so that at the end we review all processes:
 - ▶ in each phase (step) of performing the synchronization, we review a few pairs of processes,
 - ▶ in the next phase, we review other pairs, taking into account synchronization from the first phase,
 - ▶ etc.
 - ▶ If we do this wisely: in the end, all processes will be synchronized.
 - ▶ Each process will be synchronized on average with $\log(n)$ processes.
 - ▶ We need $\log(n)$ phases.
- ▶ This is the principle of a symmetrical barrier
 - ▶ the same amount of work is done in each process.

Two process barrier

- ▶ Processes i and j signal the completion of the work by setting the `arrive[i]` and `arrive[j]` to 1. Process i (j) then waits for the process j (i), `arrive[j]==1` (`arrive[i]==1`), and it sets it back to 0: `arrive[j] = 0` (`arrive[i]=0`).

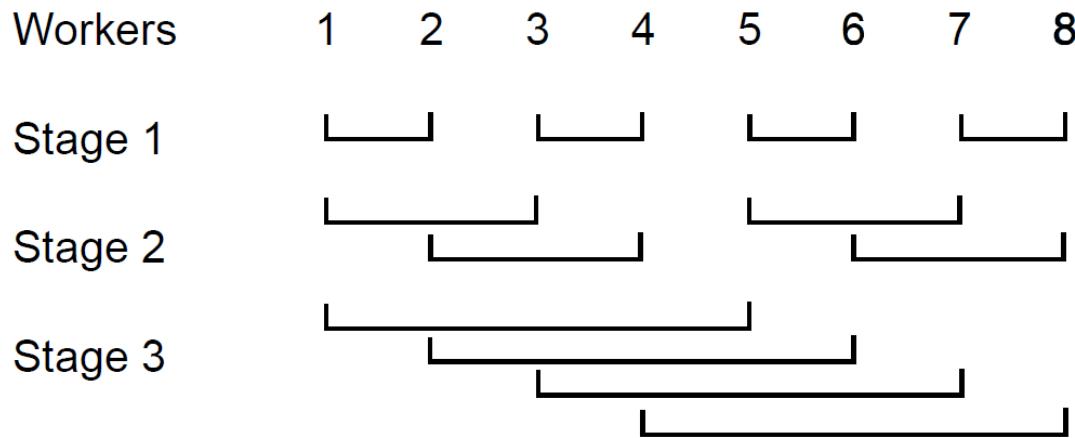
```
/* barrier code for worker process W[i] */
<await (arrive[i] == 0);> /* key line -- see text */
arrive[i] = 1;
<await (arrive[j] == 1);>
arrive[j] = 0;

/* barrier code for worker process W[j] */
<await (arrive[j] == 0);> /* key line -- see text */
arrive[j] = 1;
<await (arrive[i] == 1);>
arrive[i] = 0;
```

- ▶ First line is needed to signal the end of work from previous phase.
- ▶ The last line is sets the flag to 0 for the next phase.

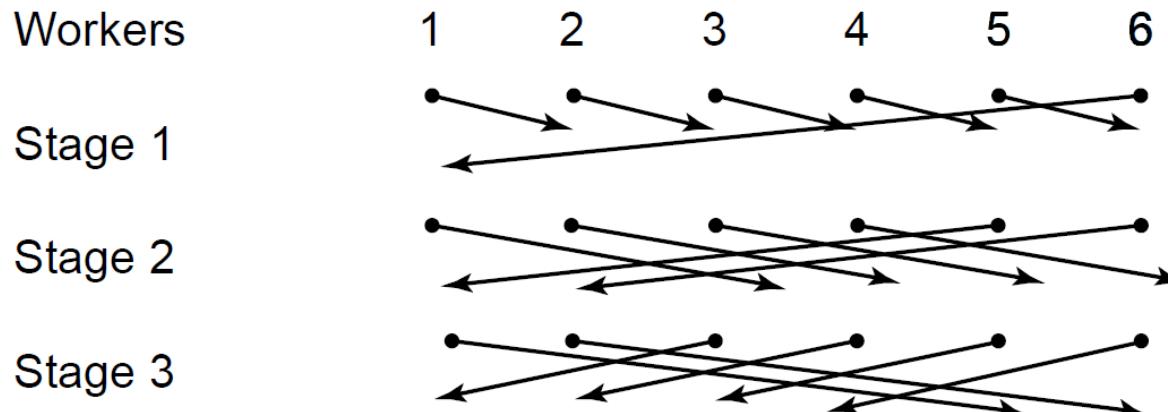
Simmetric butterfly barrier setting

- ▶ Useful if the number of processes (n) is a power of 2.
- ▶ Number of phases: $\log_2 n$
 - ▶ the processes that are distant 2^{k-1} (according to the process ID) are synchronized in phase k ,
 - ▶ each process is synchronized with $\log_2 n$ processes.



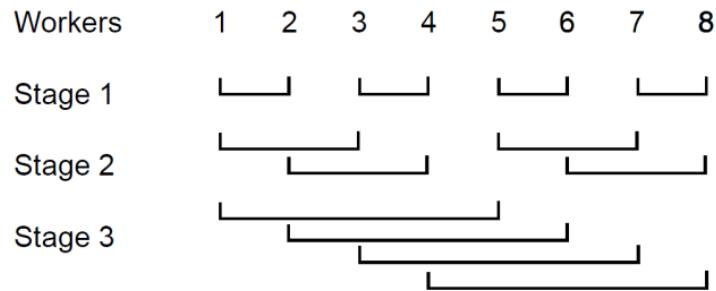
Simmetric barrier with distribution

- ▶ Suitable for any number of processes.
- ▶ Number of phases $\lceil \log_2 n \rceil$
 - ▶ in phase k : barrier between process i and process $(i+2^{k-1}) \bmod n$,
 - ▶ process in each phase distributes information about its arrival to processes on the right:
 - ▶ it first raises its flag in the barrier, and then waits for the process flag to be raised to the right so that it can be dropped.
- ▶ Each process synchronizes with $\lceil \log_2 n \rceil$ other processes.



The problem of overtaking

- ▶ The problem of overtaking occurs, because at the same time we perform multiple synchronizations between two processes in several phases. A deadlock can happen.



- ▶ Let's look at an example of a butterfly barrier:
 - ▶ suppose that in phase 1 processes 3 and 4 synchronize,
 - ▶ process 1 in phase 1 is waiting for process 2 (it is delayed). Process 1, therefore, puts its flag on the `arrive[1]` to 1 and waits for `arrive[2]` to become 1.
 - ▶ In phase 2, process 3 continues its synchronization with the process 1. It waits for it to set `arrive[1]` to 1. This is already done, so it sets up `arrive[1]` to 0 and continues its synchronization with process 7 in phase 3.
- ▶ Deadlock: process 3 synchronized with process 1 and left the barrier, while process 2 is waiting for entry into the barrier with process 1, which will never happen.

How can we prevent this?

- ▶ First idea:
 - ▶ We use different flags for each phase of synchronization.
 - ▶ Too much coordination. Too many flags.
- ▶ Second idea:
 - ▶ each flag should keep track of the synchronization phase.
 - ▶ flags should not be a binary value (0,1), but should be also include information about the phase,
 - ▶ You can go past the barrier when:

```
# barrier for process i
for [s = 0 to num_stages] {
    arrive[i] = arrive[i] + 1;
    # wait for neighbour j
    while (arrive[j] < arrive[i]) skip;
}
```

```
< await (arrive[i] == 0); >
arrive[i] = 1;
< await (arrive[j] == 1); >
arrive[j] = 0;
```



Forms of parallelism

- ▶ Data parallelism
 - ▶ the parallelization of the process with regard to the number of data to be processed at the same time,
 - ▶ the data can be divided into groups, where we can perform individual tasks (same operations) and thus solve the task.
 - ▶ example: divide and conquer,
 - ▶ synchronization with barriers is required
 - ▶ in the case of common variables, CS synchronization with locks or something similar.
- ▶ Procedural parallelism
 - ▶ parallelization of the procedure with respect to the number of tasks (operations) that can be performed at the same time.
 - ▶ example: pipes in Unix: the end result is the result of a sequence of tasks that are executed on the input data
 - ▶ CS synchronization is required: writing of previous task to the shared memory that the next task will need (read) for the execution.
- ▶ The combination of both
 - ▶ it is a combination of both parallelisms
 - ▶ it is also called asynchronous parallelism
 - ▶ synchronization with barriers and synchronization of CS is required.

Example of data parallelism: partial sums of numbers in an array

- ▶ Task:
 - ▶ We have an array of numbers $a[0:n-1]$. Calculate all partial sums in the array. Partial sum till index i is a sum of all numbers in the array $a[0 : i]$.
- ▶ Sequential program:

```
sum[0] = a[0];
for [i = 1 to n-1]
    sum[i] = sum[i-1] + a[i];
```

Example of data parallelism: partial sums of numbers in an array

- ▶ How to parallelize a given problem?
- ▶ Idea:
 - ▶ We add simultaneously pairs of numbers in the array:
 - ▶ eg. neighboring pairs of numbers: $a[0] + a[1]$, $a[2] + a[3]$, ...
 - ▶ concurrent add of pairs is continued in the following steps:
 - ▶ eg. we can proceed by adding the sum $(a[2] + a[3])$ to the sum $(a[0] + a[1])$, etc. Principle divide and conquer.
 - ▶ However, we do not get all partial sums of numbers, although we reduce the number of counting steps to $O(\log n)$. therefore:
 - ▶ so:
 - in the first step, $\text{sum}[i] = a[i]$ za vse i
 - second step: add all pairs of partial sums that are distant by 1 $\text{sum}[i] + \text{sum}[i-1]$ for all $i >= 1$
 - third step: add all pairs of partial sums that are distant the distance of the previous step multiplied by 2 $\text{sum}[i] + \text{sum}[i-2]$ for all $i >= 2$...
 - after $\lceil \log_2 n \rceil$ steps we get all partial sums.

example:	starting values for a	1	2	3	4	5	6
	sum after first step	1	3	5	7	9	11
	sum after second step	1	3	6	10	14	18
	sum after third step	1	3	6	10	15	21

Example of data parallelism: partial sums of numbers in an array

```
int a[n], sum[n], old[n];
process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i]; /* initialize elements of sum */
    barrier(i);
    ## SUM: sum[i] = (a[i-d+1] + ... + a[i])
    while (d < n) {
        old[i] = sum[i]; /* save old value */
        barrier(i);
        if ((i-d) >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
        d = d+d; /* double the distance */
    }
}
```

▶ **barrier(i):**

- ▶ barrier that gets an argument process ID – serial number,
- ▶ the process returns from the barrier when all processes reach the barrier,
- ▶ we use one of the algorithms for implementing barriers,
- ▶ two barriers:
 - ▶ first is needed because all processes must write the value of the sum till this step as the old sum,
 - ▶ We need another because we need to obtain all the partial sums before increasing the distance for the next step.

Data parallelism: concurrent calculations over geometrically arranged data

- ▶ Geometrically arranged data are data that are organized into a certain geometric structure:
 - ▶ eg. represent a network of points.
- ▶ Problem: the solution to the problem must be calculated at each point. Normally in several iterations, where in each iteration the same procedure is repeated on the newly calculated data (to the final solution or convergence to the solution).
 - ▶ Such procedures can be parallelized.
 - ▶ We divide the data into subsections and calculate the solution for each point in this subsection.
 - ▶ Calculations in subsections can be performed concurrently.
- ▶ Formulation of the problem:
 - initialization ;
 - while** (condition to break)
 - calculate new values in each point;**

- ▶ Examples:
 - ▶ image processing: e.g. looking for points with the same luminosity in the image, Hough's transformation,
 - ▶ PDE solving, e.g. using Jacobi's iterative method.

PDE solving: Laplace differential equation

- ▶ Laplace differential equation with the Dirichlet boundary conditions is defined in 2D as:

$$\frac{\partial^2 \phi}{\partial^2 x} + \frac{\partial^2 \phi}{\partial^2 y} = 0$$

- ▶ ϕ represents an unknown quantity that spreads over the surface, e.g. temperature values on the edges of the surface are constant.
- ▶ The task is to calculate the distribution of this amount over the inner surface over a period of time (or until the distribution is no longer changing),
 - ▶ assumptions: 2D surface, we are looking for an (approximated) numerical solution.
- ▶ Parallelization:
 - ▶ we use one of the iterative methods of solving the PDE, e.g. Jacobi's iterative method,
 - ▶ the network of 2D points of the area is divided into subsections, e.g. strips, squares and for each such subsection we use our worker (process),
 - ▶ a concurrent calculation is performed for each subsection.

Paralelization of the Jacobi's iterative method

- ▶ Jacobi's iterative method:

- ▶ the new value at each point is calculated as the average of the values of the adjacent four points

$$\phi_{i,j}^k = \frac{1}{4}(\phi_{i-1,j}^{k-1} + \phi_{i+1,j}^{k-1} + \phi_{i,j-1}^{k-1} + \phi_{i,j+1}^{k-1})$$

- ▶ the process is carried out until the new and old values differ by less than ε .

- ▶ Parallel program:

```
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;
process Grid[i = 1 to n, j = 1 to n] {
    while (not converged) {
        newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                         grid[i,j-1] + grid[i,j+1]) / 4;
        check for convergence
        barrier(i);
        grid[i,j] = newgrid[i,j];
        barrier(i);
    }
}
```

Bag of tasks principle

- ▶ Data parallelism: divide and conquer
 - ▶ distribute data to smaller parts, perform some work on them, the final solution is a combination of the solutions to these tasks
 - ▶ suitable for recursive procedures,
 - ▶ parallelization: simultaneous execution of tasks on distributed data
- ▶ Alternative: Bag of tasks principle
 - ▶ processes share a bag with prepared tasks,
 - ▶ there are more tasks than processes:
 - ▶ at the beginning all processes get their task, when they finish working, they take a new task
 - ▶ this continues until the bag is empty and all tasks have been solved.

```
bool done = false;  
shared bag of tasks;  
process Worker[w = 1 to P] {  
    while (bag of tasks is not empty or !done) {  
        <get a task from the bag;>  
        execute the task;  
        possibly generate new task(s) and <put it to the bag;>  
    }  
}
```

- ▶ this is a form of data parallelism or combined parallelism

Bag of tasks principle

- ▶ Suitable when we anticipate a fixed number of processes.
- ▶ Equal distribution of work between different processes.
- ▶ Since we have a common bag, this is shared between processes and therefore one of the methods for ensuring mutual exclusion must be used:
 - ▶ lock (that locks the bag access),
- ▶ When does the program end?
 - ▶ when the bag is empty and all the tasks are complete,
 - ▶ tasks are completed when all processes (workers) are waiting for the next task.
- ▶ Examples:
 - ▶ parallel multiplication of matrices,
 - ▶ recursive calculation of a definite integral.

Example 1: multiplication of matrices according to the principle of task bag

- ▶ Number of processes (= number of processors)
 $\ll n$

- ▶ Bag of tasks

```
<row = nextRow++;>
```

- ▶ Take a new task:

- ▶ can be implemented with the atomic instruction FA
- ▶ bag is empty when `row >= n`

- ▶ End of program:

- ▶ when all workers jump out of the loop

- ▶ Barrier:

- ▶ for loop (when all rows are calculated)
- ▶ the print of the `c` matrix after the barrier (in one of the workers)
- ▶ barrier can be implemented with counter `done`

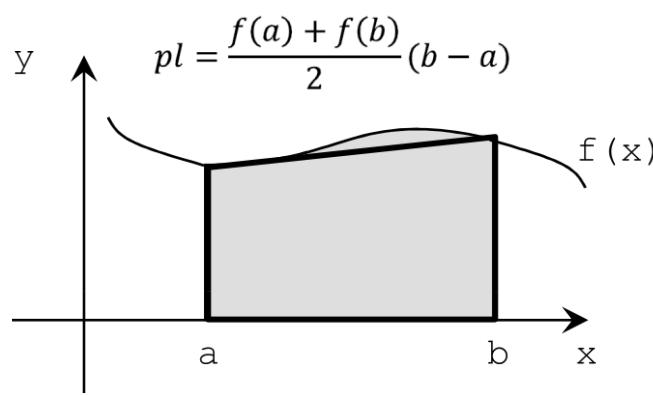
```
int nextRow = 0; # the bag of tasks
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
    int row;
    double sum; # for inner products
    while (true) {
        # get a task
        < row = nextRow; nextRow++; >
        if (row >= n)
            break;
        compute inner products for c[row,*];
    }
}

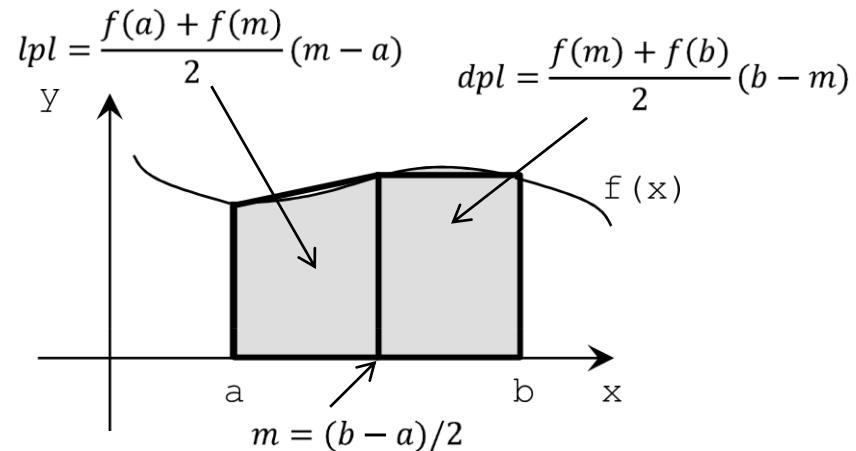
int done = 0;
if (done == n)
    print matrix c;
<done++>;
```

Parallel calculation of definite integrals

► The recursive algorithm:



first approximation
 pl



second ...
 $lpl + rpl$

► Recursion:

- on a given interval calculate the area pl ,
- the interval is divided into two halves and left and right areas are calculated lpl and rpl ,
- recursion is stopped when $|lpl + rpl - pl| < \text{eps}$.

Parallel calculation of definite integrals according to the bag of tasks principle

- ▶ **Bag:**
 - ▶ list of quintuples (a, b, f(a), f(b), area)
- ▶ **Process:**
 - ▶ we take one quintuple from the bag
 - ▶ calculate the area
 - ▶ return two quintuples (left and right side) back into the bag
 - ▶ repeat until the sum of the left and right areas differ from the total plate by less than eps.
- ▶ **Special features:**
 - ▶ the bag may also be empty, but we still have to wait for execution (compared to the previous calculation of the matrix product when we finished when the bag was empty)
 - ▶ we still have to “play” according to the principle of the task bag:
 - ▶ the program ends when the bag is empty and all workers are waiting for new tasks.

Parallel calculation of definite integrals according to the bag of tasks principle

- ▶ Bag is represented as a queue of quintuples.
- ▶ bag principle:
 - ▶ program ends when
 - ▶ **size** = 0 and
 - ▶ **idle** = n (idle counts how many workers are idle - resting)
- ▶ We have multiple atomic instructions (shared variables). This could be done with locks with no atomic instructions and CS.
- ▶ Except one:
 - ▶ this one will be done with monitors and semaphores (later).

```
type task = (double left, right, fleft, fright, lrarea);  
queue bag(task);      # the bag of tasks  
int size;            # number of tasks in bag  
int idle = 0;          # number of idle workers  
double total = 0.0;    # the total area  
  
compute approximate area from a to b;  
insert task (a, b, f(a), f(b), area) in the bag;  
count = 1;  
  
process Worker[w = 1 to PR] {  
    double left, right, fleft, fright, lrarea;  
    double mid, fmid, larea, rarea;  
    while (true) {  
        # check for termination  
        < idle++;  
        if (idle == n && size == 0) break; >  
        # get a task from the bag  
        < await (size > 0)  
        remove a task from the bag;  
        size--; idle--; >  
        mid = (left+right) / 2;  
        fmid = f(mid);  
        larea = (fleft+fmid) * (mid-left) / 2;  
        rarea = (fmid+fright) * (right-mid) / 2;  
        if (abs((larea+rarea) - lrarea) > EPSILON) {  
            < put (left, mid, fleft, fmid, larea) in the bag;  
            put (mid, right, fmid, fright, rarea) in the bag;  
            size = size + 2; >  
        } else  
            < total = total + lrarea; >  
    }  
    if (w == 1)    # worker 1 prints the result  
        printf("the total is %f\n", total);  
}
```

Programming III

Parallel and distributed programming

Semaphores

Janez Žibert, Jernej Vičič UP FAMNIT

Overview (chapter 4)

- ▶ Semaphores - Part 1 (sections 4.1, 4.2):
 - ▶ definition of a semaphore,
 - ▶ types of semaphores: mutex, composite binary, general
 - ▶ synchronization with mutex semaphores:
 - ▶ access to critical section
 - ▶ implementation of barrier
 - ▶ implementation of conditional synchronization
 - ▶ the principle of passing the baton
 - ▶ synchronization with composite mutex semaphores
 - ▶ implementation of the producer/consumer problem
 - ▶ synchronization with general semaphores
 - ▶ synchronization with a buffer with limited space
 - ▶ Semaphores - second part (sections 4.3 - 4.5)
 - ▶ the implementation of selective mutual exclusion with semaphores
 - ▶ the problem of five philosophers,
 - ▶ the problem of reading and writing shared data.
- ▶ Conditional variables

Semaphores

- ▶ Railroad semafors:
 - ▶ red light means that there is another train on the track, we must stop,
 - ▶ green light: the track ahead of us is free, we can continue driving,
 - ▶ when we pass the semafor a red light is lit at the semafor (the line is busy) and remains lit while our train is on this critical section of the line,
 - ▶ when the train leaves the critical section (!), the green light turns on again
 - ▶ semafors are set up to prevent accidents (more than one train in the critical section)
- ▶ Semaphores represent the mechanism for conditional synchronization (red, green light) between processes (trains) in order to ensure mutual exclusion of process access to critical areas (prevent accidents in the critical section of the line).
- ▶ History:
 - ▶ Edsger Dijkstra (Dutch scientist) 1968,
 - ▶ implementation in the THE operating system (Technological University of Eindhoven) ,
 - ▶ definition of two semafor operations:
 - ▶ P from dutch “Proberen” – try or “Passeren” – drive through (semafor in CS)
 - ▶ V from dutch “Verhogen” – increment or “Vrijgeven” – release (CS)

Semaphore definition

- ▶ Semaphore is a data structure that includes an integer variable **s**, and two atomic operations:
 - ▶ try to enter (CS) `P(s) : < await (s > 0) s = s - 1; >`
 - ▶ release (CS) `V(s) : < s = s + 1; >`
- ▶ The value of **s** is always **non-negative**.
- ▶ Declaration and initialization:
 - ▶ `sem s; #semaphore, value of s = 0;`
 - ▶ `sem s = expr; #semaphore, value of s = expr;`
 - ▶ `sem s[1:n] ([n] expr) #array of semaphores s[n] = expr;`

Semaphore implementation

- ▶ How to implement the P and V operation in a semaphore? The await command can be implemented by waiting in sleep (eg conditional variables) or by busy waiting (locks). We use the implementation of waiting by stopping the process in semaphores. That's why we use the FIFO queue.
- ▶ Implementation of P(s):

```
if (s > 0) s = s-1;  
else {  
    stop the process and put it in a waiting queue sem_wait_queue.  
}
```

- ▶ Implementation of V(s):

```
if (empty(sem_wait_queue) ) s = s+1;  
else {  
    take a process from sem_wait_queue and put it into execution  
    queue ready_queue.  
}
```

Semaaphore types

- ▶ **Binary semaphore (mutex)**
 - ▶ the semaphore value can only be 0 or 1
 - ▶ usage:
 - ▶ mutual exclusion of access to critical section,
 - ▶ conditional synchronization (similar to flags).
- ▶ **Composed binary semaphore**
 - ▶ several binary semaphores,
 - ▶ at a given moment only one of the semaphores can be 1, all the others are 0,
 - ▶ then the sum of the traffic lights is 0 or 1.
 - ▶ usage:
 - ▶ for mutual exclusion and conditional synchronization, example: producer/consumer, use of limited buffer
- ▶ **(General) counting semaphore**
 - ▶ the value of the semaphore can be any nonnegative integer value
 - ▶ usage:
 - ▶ count how many times the CS is accessed, how many resources are already used, etc.,
 - ▶ for conditional synchronization.

Critical sections and binary semaphores - MUTEXes

- ▶ The critical section is defined by `<S; >`.
- ▶ Instead of `<>` brackets, a binary semaphore can be used and the brackets are replaced by operations `P` and `V`.
- ▶ Such a binary semaphore is called mutex, because it ensures the mutual exclusion of process access to CS.

```
sem mutex = 1;

process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```

The mutex semaphore is initially set to 1 = access to the CS is allowed.

Using semaphores for signalization

- ▶ Binary semaphores can also be used to communicate between processes: to signal the state of a particular event or state of a certain condition:
 - ▶ usually the value of the semaphore 0 signals that the event has not yet occurred, or that the condition is not met,
 - ▶ the process signals (informs) that the event has been executed with operation V - operation V "sends" the signal (to all other processes),
 - ▶ the process waits for a signal (message) that the event happened, in operation P - the process "receives" the signals in operation P.
- ▶ Usage: barriers

- ▶ a barrier between 2 processes can be implemented with two binary semaphores, that communicate the arrival and control the exit of the barrier.
- ▶ Generalization to n processes?

```
sem arrive1 = 0, arrive2 = 0;  
  
process Worker1 {  
    ...  
    V(arrive1);      /* signal arrival */  
    P(arrive2);      /* wait for other process */  
    ...  
}  
  
process Worker2 {  
    ...  
    V(arrive2);      /* signal arrival */  
    P(arrive1);      /* wait for other process */  
    ...  
}
```

Barrier with coordinator

Converts a barrier with flags into a barrier with binary semaphores:

```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);
process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        arrive[i] = 1;
        <await (continue[i] == 1);>
        continue[i] = 0;
    }
}
process Coordinator {
    while (true) {
        for [i = 1 to n] {
            <await (arrive[i] == 1);>
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}
```

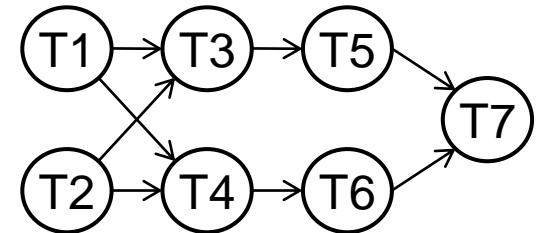
Barrier in binary tree

Converts a barrier with flags into a barrier with binary semaphores:

```
leaf node L:  arrive[L] = 1;  
             ⟨await (continue[L] == 1);⟩  
             continue[L] = 0;  
  
interior node I:  ⟨await (arrive[left] == 1);⟩  
                  arrive[left] = 0;  
                  ⟨await (arrive[right] == 1);⟩  
                  arrive[right] = 0;  
                  arrive[I] = 1;  
                  ⟨await (continue[I] == 1);⟩  
                  continue[I] = 0;  
                  continue[left] = 1; continue[right] = 1;  
  
root node R:  ⟨await (arrive[left] == 1);⟩  
                  arrive[left] = 0;  
                  ⟨await (arrive[right] == 1);⟩  
                  arrive[right] = 0;  
                  continue[left] = 1; continue[right] = 1;
```

Example

- ▶ Ensuring the correct order of execution with semaphores.
- ▶ Suppose that the given graph describes the order in which the tasks must be performed:
- ▶ Each task is completed with a process:



```
process Ti (i = 1, ..., 7) {  
    wait for predecessors, if any;  
    execute the task;  
    signal successor, if any;
```

That means that process T5 waits the process T3 for the execution and when it performs the tasks, sends a message to process T7.

- ▶ Write a program that will execute the tasks with processes in the order shown in the graph. Synchronization between processes is carried out with semaphores. Minimize the number of semaphores.

Solution

- ▶ Solution with 5 semaphores:

```
sem S[3:7] = ([5] 0);           // init
T1:: ... v(S[3]); v(S[4]);
T2:: ... v(S[3]); v(S[4]);
T3:: p(S[3]); p(S[3]); ... v(S[5]);
T4:: p(S[4]); p(S[4]); ... v(S[6]);
T5:: p(S[5]); ... v(S[7]);
T6:: p(S[6]); ... v(S[7]);
T7:: p(S[7]); p(S[7]); ...
```

- ▶ Reduce the number of semaphores to 4:

- ▶ T7 is executed after T3 and T4, one of the semaphores signaling the ending of these two processes can be reused, say S[3].

```
sem S[3:6] = ([4] 0);           // init
T1:: ... v(S[3]); v(S[4]);
T2:: ... v(S[3]); v(S[4]);
T3:: p(S[3]); p(S[3]); ... v(S[5]); v(S[5]);
T4:: p(S[4]); p(S[4]); ... v(S[6]);
T5:: p(S[5]); ... v(S[3]);
T6:: p(S[6]); ... v(S[3]);
T7:: p(S[5]); p(S[3]); p(S[3]); ...
```

Using binary semaphores for signalization

- ▶ Mutual exclusion for CS `<S ;>`

- ▶ Using MUTEX:

```
sem mutex = 1;    // condition for entering CS
P(mutex);
S;
V(mutex);
```

- ▶ Conditional synchronization `<await (B) S ;>`

- ▶ Using two semaphores: one for access to CS and one for signaling that the condition B is fulfilled.

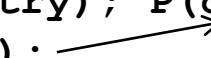
```
sem entry = 1;    // condition to enter CS
sem cond = 0;      // semaphore for condition
P(entry);
while (!B) {V(entry); P(cond); P(entry);}
S;
V(entry);
```

- ▶ This can be improved using the “passing the batton” method.

Synchronization with the “passing the baton” method

- ▶ Observe this example:
 - ▶ Process W, waits for the entry to CS when the **B** condition is fulfilled:
`W: P(entry); while (!B) {V(entry); P(cond); P(entry);}`
 - ▶ Two semaphores are needed for execution of the process W, **cond** and **entry**
 - ▶ Process S, sets **B = true** and signals this event.
`S: P(entry); B = true; V(cond); V(entry);`
 - ▶ Process S uses semaphore **cond** to signal that **B** is true, semaphore **entry** is used to access CS. The values of both semaphores are changed!
 - ▶ Double **P(entry)**! That allows the process **S** to pass the semaphore **entry** (allow entry to CS) to the process **W** using semaphore **cond**, that signals that the **B** is true.
- ▶ Method “passing the baton”: baton is the allowing to enter the CS.

```
W: P(entry); while (!B) {V(entry); P(cond); }  
S: P(entry); B = true; V(cond);
```



Passing the baton method example

```
sem entry = 1,/* controls entry to CS */
    cond = 0; /* to signal the condition */
int dp = 0; /* counts delayed processes */
process W[i = 0 to n - 1] {
    ...
    P(entry);
    if (!B) {
        dp++;
        V(entry);
        P(cond); /* delays on cond */ ←
    }
    S;
    if (dp > 0) {
        dp--;
        V(cond); /* pass the "entry baton" */
    } else V(entry);
    ...
}
```

```
process S {
    ...
    P(entry);
    change B to true;
    if (dp > 0) {
        dp--;
        V(cond); ←
    } else V(entry);
    ...
}
```

Compound binary semaphores

- ▶ The composite binary semaphore consists of several binary semaphores, where at one time only one of the traffic lights can have a value of 1, others have a value of 0.

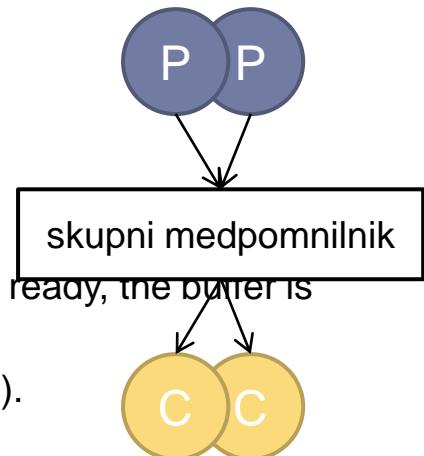
$$0 \leq s_0 + s_1 + \dots + s_{n-1} \leq 1$$

- ▶ enable the execution of mutually exclusive access to CS and signaling for conditional synchronization of several processes
- ▶ useful in cases when we monitor (report) the state of the variables (or data structures) that alternately change their values:
 - ▶ for example, two binary semaphores **full** and **empty** can form a composite binary semaphore that can be used to communicate if a data structure (eg buffer) is full or empty.

Example

- ▶ Producer/consumer problem: shared buffer synchronization:

- ▶ producer (P) is feeding the buffer,
- ▶ consumer (C) is removing items from buffer,
- ▶ there can be multiple producers and consumers,
- ▶ simplified problem when only one shared variable - buffer:
 - ▶ variable is initially empty (no data), the producer must wait until it is ready, the buffer is empty (no data),
 - ▶ the consumer must wait until the variable is not full (full of new data).



- ▶ Solution:

- ▶ ensure the mutual exclusion of writing and reading the shared variable,
- ▶ conditional synchronization is required:
 - ▶ C waiting for new data in the buffer, waiting until there are new data in the variable,
 - ▶ P waits for some space to be released in the buffer, waiting for the variable to be empty,
- ▶ use a compound binary semaphore for conditional synchronization and mutual exclusion of reading and writing in a variable.

The producer/consumer method – solution with binary semaphores

- ▶ Introduce two binary semaphores **full** and **empty**, which form a composite binary semaphore.
- ▶ At most one of them can be 1: $0 \leq \text{full} + \text{empty} \leq 1$
- ▶ If the producer is waiting for **empty==1** when writing to the buffer. When write to **buf** is finished, it sets **full=1**.
- ▶ Consumer waits for **full==1**, then reads the **buf**. When read it sets **empty=1**.

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
    while (true) {
        ...
        /* produce data, then deposit it in the buffer */
        → P(empty);
        buf = data;
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        /* fetch result, then consume it */
        → P(full);
        result = buf;
        V(empty);
    }
    ...
}
```

Property of compound binary semaphores

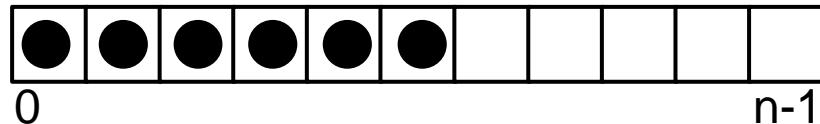
- ▶ Suppose,
 - ▶ that processes use a composite binary semaphore.
 - ▶ that each process has a sequence of commands between operations P and V on (different) binary semaphores that make up the composite binary semaphore.
- ▶ Then, mutual exclusion of the execution of commands between operations P and V in each process is ensured.
 - ▶ Proof:
 - ▶ By definition: $0 \leq s_0 + s_1 + \dots + s_{n-1} \leq 1$
 - ▶ Observation: Each time we perform commands in the area between P and V all semaphores are set to 0.
 - ▶ Which means that once a process has a (true) condition in P, it can continue to execute the CS, while all other processes do not meet the condition in P and therefore await the execution of CS until operation V.

General semaphores

- ▶ The variable in a semaphore can take any non-negative integer value:
 - ▶ the semaphore is initialized to some initial value,
 - ▶ then the counter increases or decreases,
 - ▶ with a counter we can count different things or events:
 - ▶ how many units of space we still have in memory (buffer),
 - ▶ how many data units are in the database,
 - ▶ how many times have we been in CS
- ▶ synchronization with general semaphores:
 - ▶ conditional synchronization:
 - ▶ eg. wait until enough (units) of the buffer space are available, and then you can use them.

Buffer with limited space

- In the case of the producer/consumer problem, we used a buffer that was 1 unit large. What if there are n units?



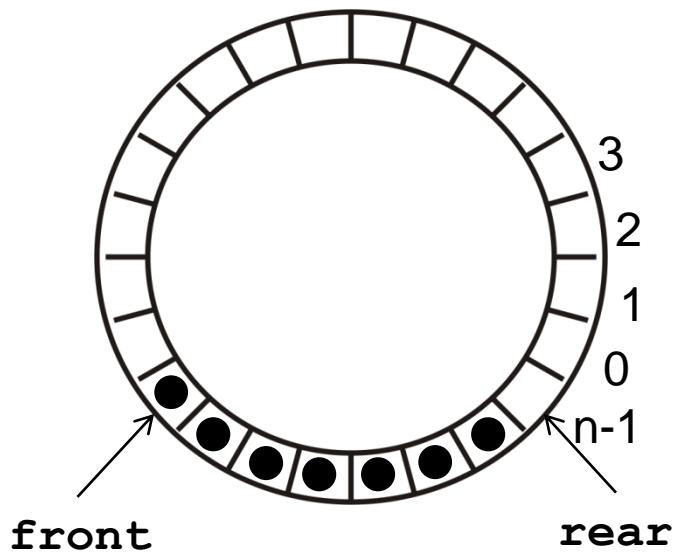
- Buffer size n can be implemented as an array **TypeT**

```
TypeT buf[n];  
int front = 0, rear = 0;
```

- Organized as a circular buffer

- rear** shows the first empty slot
- front** shows the first full slot

```
insert:    buf[rear] = data;  
(deposit)  rear = (rear+1) % n;  
  
take:     result = buf[front];  
(fetch)   front = (front+1) % n;
```



Using limited buffer with many producers/consumers

- ▶ **Synchronization of the buffer access:**

- ▶ the producer must wait until the buffer is empty (it is empty when there is still some space for writing)
- ▶ the consumer must wait until the buffer contains at least one item of information

- ▶ **Two general semaphores are used:**

```
sem empty = n;  
sem full = 0;
```

- ▶ **empty** counts empty spaces
- ▶ **full** counts full spaces

```
typeT buf[n];      /* an array of some type T */  
int front = 0, rear = 0;  
sem empty = n, full = 0; /* n-2 <= empty+full <= n */  
process Producer {  
    while (true) {  
        ...  
        produce message data and deposit it in the buffer;  
        P(empty);  
        buf[rear] = data; rear = (rear+1) % n;  
        V(full);  
    }  
}  
process Consumer {  
    while (true) {  
        fetch message result and consume it;  
        P(full);  
        result = buf[front]; front = (front+1) % n;  
        V(empty);  
        ...  
    }  
}
```

Using limited buffer with many producers/consumers

- ▶ Suppose that several producers write to shared variable and several consumers read the buffer (shared variable).
 - ▶ The insert operation becomes the CS for the producers

```
buf[rear] = data;  
rear = (rear+1) % n;
```

- ▶ operation take becomes CS for the consumers

```
result = buf[front];  
front = (front+1) % n;
```

- ▶ the insert and take operations, are not critical to one another, regardless of the fact that they share the same `buf`, the operations are executed at different places `front` and `rear`.
- ▶ It is necessary to ensure mutual exclusion of the operations that the operation of the operation is mutually exclusive and `insert` and `take`:
 - ▶ `mutexD` to lock variable the `rear` in operation `insert`, which is shared for all producers,
 - ▶ `mutexF` to lock the variable `front` in the `take` operation, which shared by all consumers.

Using limited buffer with many producers/consumers

```
typeT buf[n];      /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion */
process Producer[i = 1 to M] {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        P(mutexD);
        buf[rear] = data; rear = (rear+1) % n;
        V(mutexD);
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        fetch message result and consume it;
        P(full);
        P(mutexF);
        result = buf[front]; front = (front+1) % n;
        V(mutexF);
        V(empty);
        ...
    }
}
```

Programming III

Parallel and distributed programming



Semaphores
(second part)
Conditional variables.



Overview (chapters 4.3 – 4.5, 5.1.2, 5.1.4)

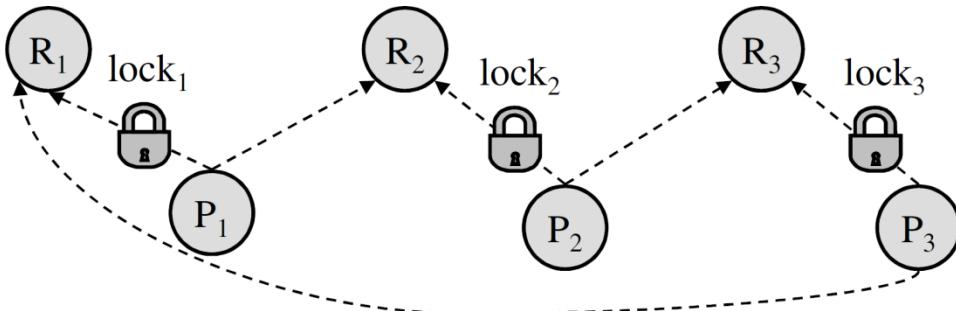
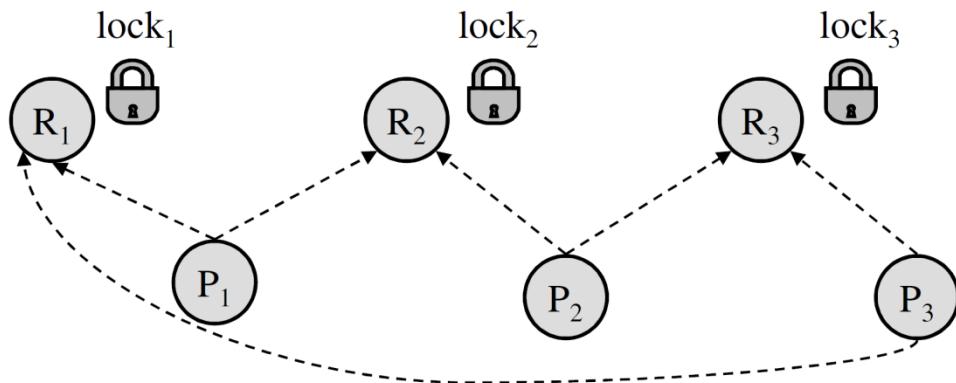
- ▶ the implementation of selective mutual exclusion of processes
 - ▶ the five philosophers problem - a process competing for shared resources,
 - ▶ the problem of reading and writing – processes competing for writing and/or reading shared data.
- ▶ Conditional variables.

Forms of selective mutual exclusion

- ▶ Two forms:
- ▶ Several (equal) processes compete for the use of several different common sources, shared by processes among themselves.
 - ▶ One process may need several different resources for its execution.
 - ▶ Modelling: the principle of solving the five philosophers problem.
- ▶ Several (different) processes access one (the same) shared resource in different ways.
 - ▶ depending on the access method, one process can allow simultaneous access to a shared resource, e.g. reading common variable
 - ▶ but other processes do not, for example, processes that write in a common variable.
 - ▶ Modeling: the principle of solving the problem of reading and writing.

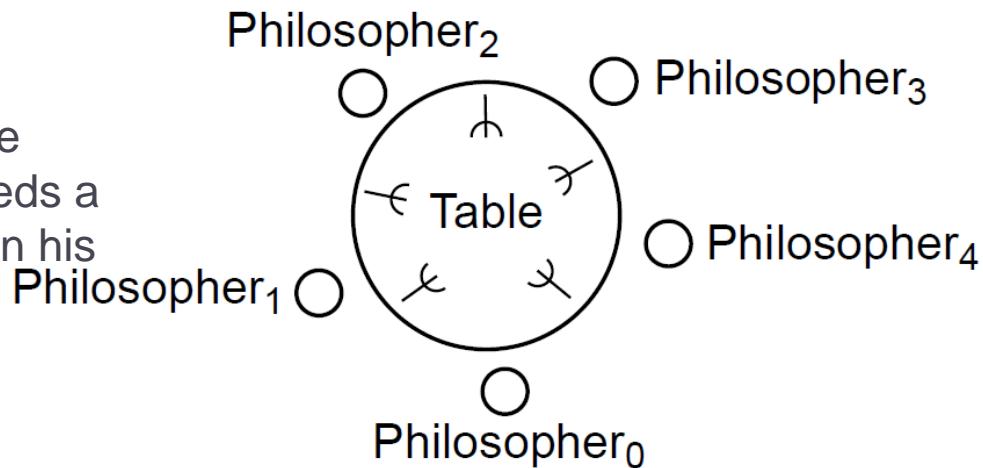
Where is the problem of sharing several common resources at the same time with multiple processes?

- ▶ Say we have several processes (P) and several common sources (R) that are protected by locks.
 - ▶ Each process must lock all the resources that it wants to use exclusively.
- ▶ Deadlock can happen:
 - ▶ example: each process $P[i]$ needs resources $R[i]$ and $R[(i+1)\%n]$.
 - ▶ **deadlock:** each process $P[i]$ locks resource $R[i]$ and tries to lock $R[(i+1)\%n]$, this will never happen because this resource is already locked by process $P[(i+1)\%n]$.



The five philosophers problem

- ▶ The five philosophers problem:
 - ▶ Five philosophers sit at a round table. In their lives, they only do two tasks: they think and eat.
 - ▶ There are only 5 sticks/forks on the table. Each eating philosopher needs a pair of forks, which are available on his left and right.
 - ▶ How to synchronize the eating of philosophers so that the situation does not happen when each person grabs one fork and does not let it go. Therefore no one can eat, and all of them die of hunger?



The five philosophers problem formulation

- ▶ Write a program that simulates the work of the 5 philosophers:

```
process Philosopher[i = 0 to 4] {
    while (true) {
        think;
        acquire a pair of forks;
        eat;
        release the forks;
    }
}
```

- ▶ Assume that the philosopher can eat and think for a long time.
- ▶ We must prevent a deadlock:
 - ▶ every philosopher takes exactly one fork, which means that no one can eat.

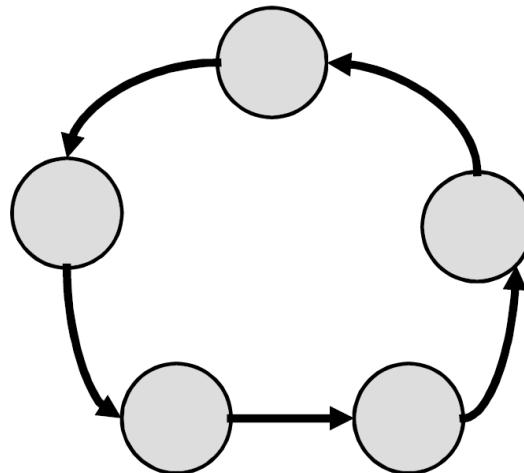
Using binary semaphores

- ▶ We need to implement the action to obtain and release forks.
- ▶ Idea: protect fork with binary semaphore:
 - ▶ each fork can be used by only one philosopher at a time,
 - ▶ the eating phase is CS due to forks, it is necessary to obtain forks from the left and right sides
 - ▶ operation P (left) and P (right) to obtain forks
 - ▶ operation V (left) and V (right) to release forks
- ▶ First attempt to solve:
 - ▶ symmetric solution: first take the fork from the right, then from the left

```
sem fork[5] = ([5] 0);
process Philosopher[i =0 to 4] {
    while(true) {
        P(fork[i]); P(fork[(i+1)%5]);
        eat;
        V(fork[i]); V(fork[(i+1)%5]);
        think;
    }
}
```

Using binary semaphores

- ▶ Is the problem solved correctly?
- ▶ NO. The deadlock can still happen.
- ▶ Circular waiting.



- ▶ How to avoid this?
 - ▶ Break the circle.
 - ▶ before: symmetrical solution: everyone is first waiting for the free right fork, then for the left fork.
 - ▶ Asymmetric solution: Somebody should wait first left and then right. By doing so, we get rid of the circular waiting.

The five philosophers problem solution

- ▶ Four philosophers acquire fork the same as before:
 - ▶ first the fork on the left,
 - ▶ then the fork on the right
- ▶ The last philosopher changes:
 - ▶ first the fork on the right
 - ▶ then the fork on the left

```
sem fork[5] = ([5] 0);
process Philosopher[i = 0 to 3] {
    while(true) {
        P(fork[i]); P(fork[i+1]);
        eat;
        V(fork[i]); V(fork[i+1]);
        think;
    }
}

process Philosopher[4] {
    while(true) {
        P(fork[0]); P(fork[4]);
        eat;
        V(fork[0]); V(fork[4]);
        think;
    }
}
```

The five philosophers problem solution

- ▶ We divide the philosophers into those sitting in the even seats and those that are in the odd seats.
- ▶ “Even” philosophers first take the right fork and then left.
- ▶ “Odd” philosophers do just the opposite.
- ▶ This is another way to prevent the deadlock from circular waiting. All solutions can be generalized to n processes.

```
sem fork[5] = ([5] 0);
process Philosopher[i = 0 to 4] {
    while(true) {
        if (i%2 == 0) {
            P(fork[(i+1)%5]); P(fork[i]);
        } else {
            P(fork[i]); P(fork[(i+1)%5]);
        }
        eat;
        V(fork[i]); V(fork[(i+1)%5]);
        think;
    }
}
```

Reading and writing problem

- ▶ The problem of reading and writing to shared memory is a concrete implementation of the general principle where shared memory is accessed by processes in different ways (some read, others write and read, etc.):
 - ▶ it is necessary to implement the mutual exclusion of the access for various operations in different ways,
 - ▶ this principle is very common in computer applications.
- ▶ Example: database
 - ▶ reading data - here only data is read from a shared database
 - ▶ update the data - here we first check the data in the database (read) and then write new data into the database, if necessary
- ▶ Problem reading/writing to shared data:
 - ▶ it is necessary to develop a synchronization mechanism that will allow data to be written only for one process at the same time, so as to enable
 - ▶ simultaneous reading of data,
 - ▶ mutually exclusion of writing operations with other writing and reading operations.

First solution

- ▶ The problem of reading and writing on a shared database.
- ▶ Implement one binary semaphore, which is used for each operation, that is, for reading and writing to the database.
 - ▶ This ensures the mutual exclusion of all operations (not just writing)
 - ▶ Also excludes simultaneous reading of data.

```
sem rw = 1;
process Reader[i = 1 to M] {
    while (true) {
        ...
        P(rw);      # grab exclusive access lock
        read the database;
        V(rw);      # release the lock
    }
}
process Writer[j = 1 to N] {
    while (true) {
        ...
        P(rw);      # grab exclusive access lock
        write the database;
        V(rw);      # release the lock
    }
}
```

Second solution

- ▶ The fact that the database is locked for all operations when someone reads data is a too strong limit. How do we get rid of it?
- ▶ Idea:
 - ▶ Other processes can read data when a process is reading data. One option: when the first process begins to read the data, this can be allowed to other processes, when the last one of this group ends, we can allow other operations.
 - ▶ So: the process that starts first by reading executes the command P (rw), thereby locking the database, while other processes that read it do not execute this command (go past).
 - ▶ The last process that performs the reading performs the command V (rw), thereby unlocking the database. Other processes (not the last) do not execute this command.
 - ▶ How do we do it?
 - ▶ We introduce an additional counter `nr`, which atomically counts how many readers are currently reading. When `nr == 1`, we have the first process, when `nr == 0` the last one finished reading.
 - ▶ An additional binary `mutexR` semaphore is used for atomic increment and decrement of the counter.

Reading and writing problem solution

```
int nr = 0;          # number of active readers
sem rw = 1;          # lock for access to the database
sem mutexR = 1;      # lock for reader access to nr
process Reader[i = 1 to m] {
    while (true) {
        ...
        P(mutexR);
        nr = nr+1;
        if (nr == 1) P(rw);  # if first, get lock
        V(mutexR);
        read the database;
        P(mutexR);
        nr = nr-1;
        if (nr == 0) V(rw);  # if last, release lock
        V(mutexR);
    }
}
process Writer[j = 1 to n] {
    while (true) {
        ...
        P(rw);
        write the database;
        V(rw);
    }
}
```

Drawbacks of the last solution

- ▶ The solution works. The simultaneous reading and mutual exclusion of writing and reading operations is ensured.
- ▶ However:
 - ▶ We give priority to reading: if a process carries out a reading and there are two new processes coming at the same time, the reader has priority. This can be repeated and always will be given priority to the reader.
 - ▶ The solution is not fair.
- ▶ Implementation of the solution to the problem: Processes should be executed in a queue as requests arise.
 - ▶ It is necessary to keep the order of the database access processes.
 - ▶ We use the conditional synchronization and the principle of passing the baton - the semaphore with which we access the base (reading or writing) is given to the one that is next in line.

Reading and writing problem solution with conditional synchronization

▶ Definition:

- ▶ nr - the number of active processes that read the data from the database (at the beginning 0)
- ▶ nw - the number of active processes that write the data from the database (at the beginning 0)

▶ Solution with atomic instructions

if nobody writes,
we can read

```
int nr = 0, nw = 0;
## RW: (nr == 0 ∨ nw == 0) ∧ nw <= 1
process Reader[i = 1 to m] {
    while (true) {
        ...
        <await (nw == 0) nr = nr+1;>
        read the database;
        <nr = nr-1;>
    }
}
process Writer[j = 1 to n] {
    while (true) {
        ...
        <await (nr == 0 and nw == 0) nw = nw+1;>
        write the database;
        <nw = nw-1;>
    }
}
```

if nobody writes and
nobody reads, we can write

```
if (nw == 0 and dr > 0) {
    dr = dr-1; v(r);    //wake up reader
}
elseif (nr == 0 and nw == 0 and dw > 0) {
    dw = dw-1; v(w);    // wake up writer
}
else
    v(e);                //release the CS lock
```

A solution to the problem of reading and writing with conditional synchronization using semaphores

- ▶ Add also
 - ▶ dr – the number of retained processes that read the data from the database
 - ▶ dw - the number of retained processes that write data to the database
 - ▶ a composite binary semaphore (sem e r w) for carrying out the baton passing

dw = waiting writers
dr = waiting readers

```
if (nw == 0 and dr > 0) {  
    dr = dr-1; V(r);    //wake reader  
}  
elseif (nr == 0 and nw == 0 and dw > 0) {  
    dw = dw-1; V(w);    // wake writer  
}  
else  
    V(e);                //release CS lock
```

```
int nr = 0,      ## RW: (nr == 0 or nw == 0) and nw <= 1  
nw = 0;  
sem e = 1,      # controls entry to critical sections  
r = 0,          # used to delay readers  
w = 0;          # used to delay writers  
# at all times 0 <= (e+r+w) <= 1  
int dr = 0,      # number of delayed readers  
dw = 0;          # number of delayed writers  
  
process Reader[i = 1 to M] {  
    while (true) {  
        # <await (nw == 0) nr = nr+1;  
        P(e);  
        if (nw > 0) { dr = dr+1; V(e); P(r); }  
        nr = nr+1;  
        SIGNAL;      # see text for details  
        read the database;  
        # <nr = nr-1;  
        P(e);  
        nr = nr-1;  
        SIGNAL;  
    }  
}  
  
process Writer[j = 1 to N] {  
    while (true) {  
        # <await (nr == 0 and nw == 0) nw = nw+1;  
        P(e);  
        if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }  
        nw = nw+1;  
        SIGNAL;  
        write the database;  
        # <nw = nw-1;  
        P(e);  
        nw = nw-1;  
        SIGNAL;  
    }  
}
```

A solution to the problem of reading and writing with conditional synchronization using semaphores

▶ Add also

- ▶ dr – the number of retained processes that read the data from the database
- ▶ dw - the number of retained processes that write data to the database
- ▶ a composite binary semaphore (sem e r w) for carrying out the baton passing

▶ Baton passing principle:

- ▶ if no process is writing to the database, the Reader process passes the baton to all other Readers, if there are any, otherwise to all the remaining Writers, if any exist.
- ▶ The Writer process, after writing data to the database, passes the baton to all other Readers, if there are anything else, otherwise, all the remaining Writers, if any exist.

```
int nr = 0, ## RW: (nr==0 or nw==0) and nw<=1
      nw = 0;
sem e = 1, # controls entry to critical sections
      r = 0, # used to delay readers
      w = 0; # used to delay writers
          # at all times 0 <= (e+r+w) <= 1
int dr = 0, # number of delayed readers
      dw = 0; # number of delayed writers
process Reader[i = 1 to M] {
    while (true) {
        # <await (nw == 0) nr = nr+1;>
        P(e);
        if (nw > 0) { dr = dr+1; V(e); P(r); }
        nr = nr+1;
        if (dr > 0) { dr = dr-1; V(r); }
        else V(e);
        read the database;
        # <nr = nr-1;>
        P(e);
        nr = nr-1;
        if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}
process Writer[j = 1 to N] {
    while (true) {
        # <await (nr == 0 and nw == 0) nw = nw+1;>
        P(e);
        if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
        nw = nw+1;
        V(e);
        write the database;
        # <nw = nw-1;>
        P(e);
        nw = nw-1;
        if (dr > 0) { dr = dr-1; V(r); }
        elseif (dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}
```

Exercise1: producer/consumer

- ▶ Suppose we have two operations:
 - ▶ **broadcast (message)** : to all processes that are currently receiving messages (**listen**), send a copy of the message and wait until everyone receives this message,
 - ▶ **listen (x)** : wait for the next message, accept a copy of the message and save it to the local variable **x**
- ▶ The producer broadcasts the message to all consumers who are currently waiting for messages (**listen**). If there is no such consumer, then **broadcast (m)** does not do anything. If there are such consumers, the producer delivers a copy of the message to each of them and is waiting for all consumers to receive it. So both operations stop each other until the message is handed over.
- ▶ Task:
 - ▶ Implement both **broadcast** and **listen** operations. Assume that the message represents one variable int. Use semaphores for synchronization. Assume that we can have multiple producers and multiple consumers, but at any moment only one broadcast operation and one operation can be performed.

Task 1: solution

```
int buffer, waiting = 0;
sem enter = 1; go = 0;

broadcast(m) {
    P(enter);
    buffer = m;
    if (waiting > 0) {
        waiting = waiting - 1;
        V(go);
    } else
        V(enter);
}

listen(x) {
    P(enter);
    waiting = waiting + 1; //register another process that waits
    V(enter);
    P(go); // wait till it is your turn
    x = buffer;
    if (waiting > 0) {
        waiting = waiting - 1;
        V(go); // baton
    } else
        V(enter);
}
```

Exercise 2: sleep and wakeup

- ▶ The Linux core implements two atomic operations:
 - ▶ `sleep()` : stops the execution of a process
 - ▶ `wakeup()` : wakes all stopped processes
- ▶ Process, that executes the `sleep()` operation always stops immediately.
- ▶ Operation `wakeup()` wakes all processes that have been stopped by the `sleep()` operation, **between two `wakeup()` calls**.
- ▶ Implement both operations with semaphores.

Task 2: solution

- ▶ Solution using pass the baton

```
int dp = 0;
sem delay = 0;
sem e = 1;

sleep() {
    P(e);
    dp++;
    V(e);
    P(delay);
    dp--;
    if (dp > 0) V(delay);
    else V(e);
}

wakeup() {
    P(e);
    if (dp > 0) V(delay);
    else V(e);
}
```

- ▶ “Bad” solution: race condition

```
int dp = 0;
sem delay = 0;
sem e = 1;

sleep() {
    P(e);
    dp++;
    V(e);
    P(delay);
}

wakeup() {
    P(e);
    while (dp > 0) {
        dp--;
        V(delay);
    }
    V(e);
}
```

Conditional variables

▶ *busy waiting*

- ▶ the process is waiting for the synchronization (to enter the CS) in the loop, until the condition for synchronization (for entering the CS) is fulfilled,
- ▶ example: locks
- ▶ Weaknesses:
 - ▶ waiting in the loop "steals" the processor time

▶ *blocking*

- ▶ the process waiting for synchronization (to enter the CS) is stopped - it does not work at all
- ▶ it is "waken up" from "sleep" by another process
- ▶ Implementation in the OS.
- ▶ Two mechanisms:
 - ▶ semaphores,
 - ▶ conditional variables (built into monitors).

Conditional variables

- ▶ **conditional variable** is a data structure consisting of a "waiting queue" and operations over the queue that allow work with processes.
- ▶ Operations:
 - ▶ stops a process - locks the execution of a process (the process stops and is put in a queue of stopped processes)
 - ▶ send a signal to continue running the process - unlock the execution of a process (changes the status from stopped process to a process in execution)
 - ▶ other common operations over queues:
 - ▶ whether the queue of stopped processes is empty,
 - ▶ find in the queue stopped processes the process with the lowest priority

Operations on conditional variables

- ▶ Deklaration of a conditional variable `cond cv`
- ▶ Operations:

<code>wait(cv, lock)</code>	release the <code>lock</code> and put process at the end of the waiting queue
<code>wait(cv, rank, lock)</code>	the process is stoped, releas the lock, and put the process into a queue for stopped processes at a place for the rank (the processes are sorted ascending by rank)
<code>signal(cv)</code>	wake up the process that is at the begining of the queue
<code>signal_all(cv)</code>	wake up all the processes of the queue and put them in a queue for execution
<code>empty(cv)</code>	<code>true</code> , if the queue is empty
<code>minrank(cv)</code>	return the rank of a process that is at the beginning of the queue (this is the smallest rank of processes in a queue, since the processes in the queue are arranged from the smallest to the highest ranking)

- ▶ Izvedba (v nadaljevanju)

- ▶ with monitors
- ▶ in Pthreads library

Signalization types in `cond cv`

- ▶ What to do if a process sends a signal for execution to another process?
 - ▶ Two processes could continue the execution (but only one can). Which one?
- ▶ Signal and Continue (SC)
 - ▶ process, that sends the signal continues with execution,
 - ▶ process, that receives the signal waits for execution,
 - ▶ process, that receives the signal is awaikened and put into the waiting queue.
- ▶ Signal and Wait (SW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put into the waiting queue.
- ▶ Signal and Urgent Wait (SUW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put at the start of the waiting queue.

Programming III

Parallel and distributed programming

Library PTHREADS

Janez Žibert, Jernej Vičič UP FAMNIT

Vsebina predavanja (poglavlja 4.6, 5.5, www)

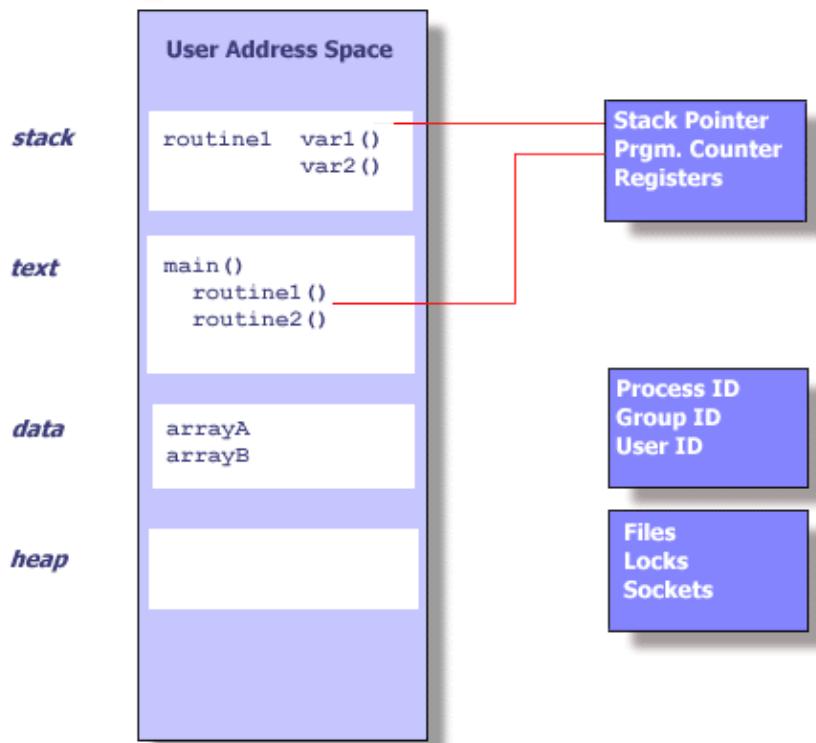
- ▶ Pthreads – IEEE POSIX threads API
- ▶ Pthreads – types and functions
- ▶ Working with threads:
 - ▶ functions for creating and killing threads,
 - ▶ thread functions:
 - ▶ joinable threads
 - ▶ detachable threads
 - ▶ thread synchronization (processes)
 - ▶ locks (mutex locks)
 - ▶ condition variables
 - ▶ semaphores (semaphores)
- ▶ **Examples**
- ▶ Literature:
 - ▶ chapter 4.6, 5.5 in the book
 - ▶ Pthreads manual
 - ▶ <https://computing.llnl.gov/tutorials/pthreads/>

Pthreads = POSIX Threads

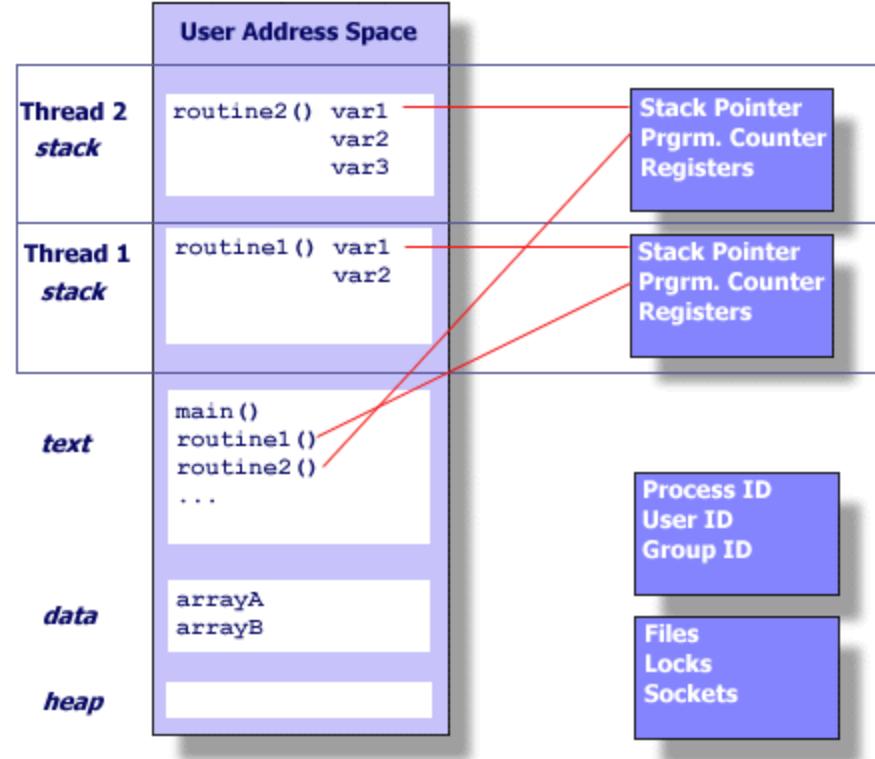
- ▶ Pthreads is a standardized collection of functions for programming language C, that allow multithreaded programming using shared memory model:
 - ▶ Standard:
IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- ▶ Multithreaded programming:
 - ▶ implementation of multiple threads in one (HW) process,
 - ▶ cooperation among threads,
- ▶ The main goal of developing the Pthreads software collection:
 - ▶ developing a single set of tools that allow the development of concurrent computing programs on single-processor and multi-processor systems.

Processes and threads

UNIX processes



Threads in UNIX processes



<https://computing.llnl.gov/tutorials/pthreads/>

Programska knjižnica Pthreads API

- ▶ Knjižnica Pthreads includes:
 - ▶ over 60 functions in the basic collection
 - ▶ around 100 functions with the extensions
- ▶ Thread management (38 functions):
 - ▶ create, exit, detach, join, get/set attributes, cancel, test cancellation, ...
- ▶ Locks (mutex locks, 19):
 - ▶ init, destroy, lock, unlock, try lock, get/set attributes
- ▶ Condition variables (condition variables, 11):
 - ▶ init, destroy, wait, timed wait, signal, broadcast, get/set attributes
- ▶ Read/write locks (read/write locks, 13):
 - ▶ init, destroy, write lock/unlock, read lock/unlock, get/set attributes
- ▶ Signals (3): send a signal to a thread, mask signals
- ▶ Extensions:
 - ▶ semaphores (semaphore.h)

Working with Pthreads

- ▶ Declaration files:

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h> // za delo s semaforji
```

- ▶ Compiling programs with pthreads library:

```
linux$ gcc -lpthread -o test test.c
```

- ▶ Extension: **-lposix4**
- ▶ Program is executed in a normal way:

```
linux$ ./test parameters
```

Variable and function names

Routine Prefix	Functional Group
<code>pthread_</code>	Threads themselves and miscellaneous subroutines
<code>pthread_attr_</code>	Thread attributes objects
<code>pthread_mutex_</code>	Mutexes
<code>pthread_mutexattr_</code>	Mutex attributes objects.
<code>pthread_cond_</code>	Condition variables
<code>pthread_condattr_</code>	Condition attributes objects
<code>pthread_key_</code>	Thread-specific data keys

- ▶ Types: `pthread[_object][_np]_t`
- ▶ Functions: `pthread[_object]_action[_np]`
- ▶ Constants and macros `PTHREAD_PURPOSE[_NP]`

- ▶ **object**: object type – e.g. attr, mutex, ...
- ▶ **action**: operation that is executed on the object, e.g. init, setscope,
- ▶ **np** or **NP**: non-portable extension
- ▶ **PURPOSE**: the purpose of the constant or macro

Basic variable types

type	Description
<code>pthread_attr_t</code>	Thread creation attribute
<code>pthread_cleanup_entry_np_t</code>	Cancelation cleanup handler entry
<code>pthread_condattr_t</code>	Condition variable creation attribute
<code>pthread_cond_t</code>	Condition Variable synchronization primitive
<code>pthread_joinoption_np_t</code>	Options structure for extensions to <code>pthread_join()</code>
<code>pthread_key_t</code>	Thread local storage key
<code>pthread_mutexattr_t</code>	Mutex creation attribute
<code>pthread_mutex_t</code>	Mutex (Mutual exclusion) synchronization primitive
<code>pthread_once_t</code>	Once time initialization control variable
<code>pthread_option_np_t</code>	Pthread run-time options structure
<code>pthread_rwlockattr_t</code>	Read/Write lock attribute
<code>pthread_rwlock_t</code>	Read/Write synchronization primitive
<code>pthread_t</code>	Pthread handle
<code>pthread_id_np_t</code>	Thread ID. For use as an integral type.
<code>struct sched_param</code>	Scheduling parameters (priority and policy)

Declaration and creation of a thread

- ▶ Declaration:

```
pthread_t tid; // nit tid
```

- ▶ Function `pthread_create` creates a new thread:

```
int pthread_create(pthread_t* tid,
                  const pthread_attr_t* attr,
                  void* (*start_routine)(void *),
                  void *arg);
```

- ▶ The program starts in the `main()` function. In the case of `pthread`s, `main()` is treated as a thread, that is, at the beginning of the program one thread is executed with `main()`. Inside `main()` we can create other threads with the above command.
- ▶ With the `pthread_create` command, we make a new thread and the thread starts immediately. The `pthread_create` function can be called anywhere in the program and can be used arbitrarily many times.
- ▶ When done with the `pthread_create` function, a new thread becomes equivalent to all previous ones. There is no hierarchy between threads

Declaration and creation of a thread

- ▶ Arguments of the `pthread_create`:
 - ▶ `* thread`: data about thread created with `pthread_create`,
 - ▶ `* attr`: when `attr = NULL` default values are used, otherwise set attributes with structure `attr`.
 - ▶ `* start_routine`: C routine that the thread executes,
 - ▶ `* arg`: arguments of the routine, that the thread executes. Argument can be only one. It is references by the `start_routine` as a reference to a pointertype (`VOID *`). The pointer is set to `NULL` in the case that the routine does not use arguments. In the case of multiple arguments: prepare a structure and set the pointer to the structure.

Example: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

attr for the thread execution

- ▶ The thread's settings are determined before the function call **pthread_create**
 - ▶ if not (**attr==NULL**), the thread starts with default settings
- ▶ Basic use:
 - ▶ declaration of the attr variable to set the thread
 - ▶ **pthread_attr_t attr;**
 - ▶ initialization of the attr
 - ▶ **pthread_attr_init(&attr);**
 - ▶ setting attr values
 - ▶ **pthread_attr_setattribute(&attr, value);**
 - ▶ delete the attr (when not used):
 - ▶ **pthread_attr_destroy(&attr);**

Possible setting for threads

- ▶ Type of execution:
 - ▶ detachstate – tells whether the thread is detached or joinable, that is, whether the thread is executed with the option of joining or not:
 - ▶ **PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED**
- ▶ Thread stack settings:
 - ▶ stackaddr – set the stack address (pointer VOID), that is used by the thread for the execution
 - ▶ `int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);`
 - ▶ stacksize – determines the minimum amount of address space (memory) for the stack for the implementation of threads
 - ▶ `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);`
- ▶ Scheduling threads:
 - ▶ schedparam – parameters for thread execution, example: priority
 - ▶ **THREAD_PRIORITY_NORMAL**
 - ▶ schedpolicy – scheduling type:
 - ▶ **SCHED_OTHER** (regular, non-real-time scheduling), **SCHED_RR** (real-time, round-robin) **SCHED_FIFO** (real-time, first-in first-out)
 - ▶ inheritsched – determine whether the thread inherits the scheduling attributes from its owner
 - ▶ **PTHREAD_EXPLICIT_SCHED, PTHREAD_INHERIT_SCHED**
 - ▶ scope – determine how to perform scheduling of processes (global or local)
 - ▶ **PTHREAD_SCOPE_SYSTEM** (share scheduling with all processes in OS), **PTHREAD_SCOPE_PROCESS** (share scheduling only with threads in the process)

Example: thread properties

thread_stack_size.c

```
pthread_attr_init(&attr);
pthread_attr_getstacksize (&attr, &stacksize);
printf("Default stack size = %li\n", stacksize);
stacksize = sizeof(double)*N*N+MEGEXTRA;
printf("Amount of stack needed per thread = %li\n", stacksize);
pthread_attr_setstacksize (&attr, stacksize);
printf("Creating threads with stack size = %li bytes\n", stacksize);
for(t=0; t<NTHREADS; t++) {
    rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
```

Input parameters of the starting function

- ▶ **pthread_create** Allows only one argument for the thread function:
 - ▶ the argument must be given to the function as the reference of the type pointer (VOID *)
 - ▶ in case the function does not need the parameters set it to NULL
 - ▶ in the case the function needs more arguments, merge them into a structure and prepare pointer to struct for the function parameter
 - ▶ **Important:** The data of the arguments stored in a structure for the function parameter used by the thread remains in the parent's thread (the one who made this thread), which can therefore change them. Do not change them when it is executed in a thread function with these arguments.

Example: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Thread identification

- ▶ Each thread gets its own identification number at the time of its creation, which can be used to address this thread.

```
pthread_t tid;  
...  
pthread_create(&tid, ...);  
...  
pthread_join(tid, NULL);
```

- ▶ Each thread can ask for the ID:

```
pthread_t mytid = pthread_self();
```

- ▶ To compare two threads:

```
pthread_equal(tid1, tid2);
```

- ▶ == (does not work as tid1,2 are structures)

Interruption of thread operation

- ▶ We can interrupt the operation of a thread in two ways:

- ▶ Explicitly with

```
void pthread_exit(void * return_value);
```

- ▶ thread returns value **return_value**
- ▶ if **main()** ends with **pthread_exit()** function call and all other threads that did not finish can continue
- ▶ if **main()** does not use **pthread_exit()** and **main** ends, all other threads end
- ▶ **pthread_exit()** does not close open files

- ▶ Implicitly, when thread function returns **return(status)**

- ▶ this is the same as calling **pthread_exit()** at the end of the function

Example: Hello World.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Thread join

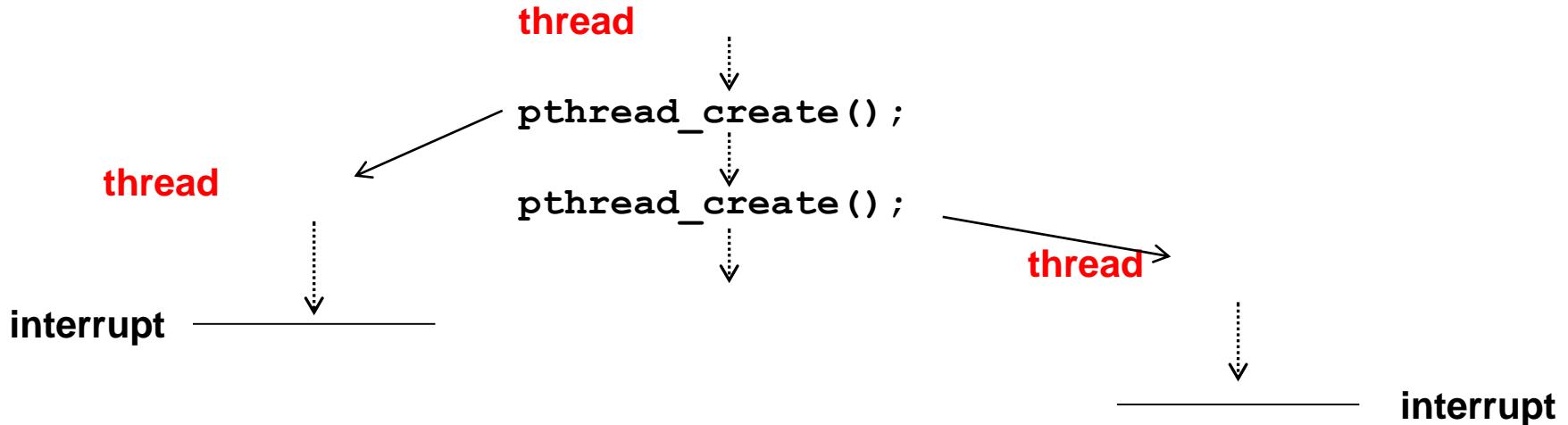
- ▶ In the thread with which we created other threads, we can wait for the completion of these threads with the command:

```
int pthread_join(pthread_t tid, void ** value);
```

- ▶ **tid** – thread id that we are waiting;
- ▶ **value** – naddress of the memory location where the result will be returned by **tid**;
- ▶ The thread in which we use this command stops with this command until the thread we are waiting for ends.
- ▶ It is imperative to provide the ID of the thread that we are waiting for.
- ▶ Waiting for all threads with this command is not possible.
- ▶ The threads can be either joinable or not.

Detached threads

- ▶ The threads that cannot be joined are called detached threads:
 - ▶ the thread that creates a new thread does not wait for this thread to end,
 - ▶ when such a thread is interrupted, it is also destroyed



- ▶ Such threads are created with **pthread_create**:
 - ▶ `pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);`
 - ▶ `pthread_create(&tid, &tattr, ...);`
- ▶ Or explicitly set the type of execution:
 - ▶ `int pthread_detach(pthread_t tid);`

Example: detached threads

detached.c

```
/* Initialize and set thread detached attribute */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

for(t=0;t<NUM_THREADS;t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}
```

Example: all inclusive:)

- ▶ HelloWorld.
- ▶ Program starts 10 threads.
- ▶ Each thread writes:
 - ▶ Hello from ...
- ▶ Most of the time the threads sleep:
 - ▶ each thread sleeps for random time
 - ▶ thus we achieve interweaving threads
- ▶ each thread operates independently.
- ▶ At the end we wait,
 - ▶ for all threads to finish,
 - ▶ print: Goodbye ...
 - ▶ end.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

#define P 10

void *sayhello (void *id)
{
    sleep(rand()%5);
    printf("Hello from thread %d\n", (int) id);
}

int main (int argc, char *argv[])
{
    int i;
    pthread_t thread[P];
    time_t t;

    t = time(NULL); // seed the random number
    srand((int) t); // generator from outside

    printf("Hello from the main thread\n");
    for (i=0; i<P; i++) {
        pthread_create(&thread[i], NULL, sayhello, (void *)i);
    }

    for (i=0; i<P; i++) {
        pthread_join(thread[i], NULL);
    }
    printf("Goodbye from the main thread\n");
    exit(0);
}
```

Synchronization mechanisms in Pthreads

- ▶ **Mutex** – locks, which ensure the mutual exclusion of the implementation of the CS
- ▶ **Conditional variables** – conditional synchronization
- ▶ **KLocks** for **reading** and **writing** problems – provide shared read and exclusive write of shared data
- ▶ **Semaphores** – this is an extension of the standarsd (stand. POSIX 1b)
 - ▶ implementation of semaphores with mutexes and conditional variables
 - ▶ `#include <semaphore.h>`
compile: `-lposix4`

Mutex

- ▶ Mutex – data structure `pthread_mutex_t`
- ▶ It ensures the mutual exclusion of processes/threads at accessing CS:
 - ▶ operations: lock, unlock, trylock
 - ▶ before they are used, they need to be initialized,
 - ▶ in the thread we can find/set the parameters of the mutex,
 - ▶ A mutex is destroyed when we no longer need it.
- ▶ Deklaration:

```
pthread_mutex_t mutex;
```

Mutex functions

- ▶ **`pthread_mutex_init(&mutex, &attr)`**
 - ▶ create and initialize a new mutex lock with parameters `attr`,
 - ▶ `mutex` is unlocket at start
- ▶ **`pthread_mutex_destroy(mutex)`**
 - ▶ destroy `mutex`,
- ▶ **`pthread_mutex_lock(&mutex)`**
 - ▶ lock the mutex lock. If the mutex is already locked, the execution will stop until the mutex is unlocked (then lock it).
- ▶ **`pthread_mutex_trylock(&mutex)`**
 - ▶ try to lock the mutex. If mutex is locked, it returns a value of 0, if not 0. The thread implementation continues.
- ▶ **`pthread_mutex_unlock(&mutex)`**
 - ▶ unlock lock `mutex`. The execution of the thread continues.

Working with mutex

- ▶ Typical mutex usage in pthreads:
 - ▶ initialize of a mutex, ..., create threads, ..., lock mutex, execute the CS, unlock mutex, ..., destroy mutex.
- ▶ Example:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
...
pthread_mutex_lock(&mutex);

CS;

pthread_mutex_unlock(&mutex);

ne-kritično področje;
```

Example 2:

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5 /* default number of threads */

/* shared variables */
double total;
pthread_mutex_t lock;

void *Counter (void *null) {
    int i;
    double *result = (double *) calloc (1, sizeof (double));
    for (i = 0; i < 1000; i++) *result = *result + (double) (random ()%100);
    pthread_mutex_lock (&lock);
    total += *result;
    pthread_mutex_unlock (&lock);
    pthread_exit ((void *) result);
}

int main (int argc, char *argv[]) {
    int n = NUM_THREADS;
    double *result;
    pthread_t thread[NUM_THREADS];
    int t;
    if (argc > 1) n = atoi (argv[1]);
    if (n > NUM_THREADS || n < 1) n = NUM_THREADS;
    for (t = 0; t < n; t++) pthread_create (&thread[t], NULL, Counter, NULL);
    for (t = 0; t < n; t++) {
        pthread_join (thread[t], (void **) &result);
        printf ("Completed join with thread %d. Result =%f\n", t, *result);
    }
    printf ("The total = %f\n", total);
}
```

Conditional variables

- ▶ Conditional variable is defined with `pthread_cond_t`.
- ▶ Used to stop and restart threads in combination with mutex.
 - ▶ So: for conditional synchronization
 - ▶ Always: in combination with mutex
- ▶ Declaration:
`pthread_cond_t cond;`
- ▶ Operations:
 - ▶ `wait`, `signal`, `broadcast`
 - ▶ execution type: Signal and Continue Signaling discipline

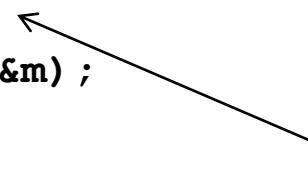
Functions on conditional variables

- ▶ **`pthread_cond_init(&cond, &attr)`**
 - ▶ creates and initialize a new conditional variable cond according to the attr settings
- ▶ **`pthread_cond_destroy(&cond)`**
 - ▶ destroy conditional variable when not used any more
- ▶ **`pthread_cond_wait(&cond, &mutex)`**
 - ▶ qait, until the conditional variab receives a signal to continue,
 - ▶ the thread that executes the command is stopped and put in a waiting queue,
 - ▶ to use this feature, it is necessary to have a mutex bound to this conditional script, locked, this mutex is released during the wait (the other threads can continue to work).
 - ▶ When conditional variable receives a signal, the thread wakes and the mutex locks.
- ▶ **`pthread_cond_signal(&cond)`**
 - ▶ send signal to cond. variable,
 - ▶ prva nit, ki čaka v vrsti pogojne spr. se postavi v vrsto za izvajanje in lahko nadaljuje svoje delo.
the first thread waiting in a row of conditional spr. puts himself in line for implementation and can continue his work.
- ▶ **`pthread_cond_broadcast(&cond)`**
 - ▶ send a signal to all conditional variables that wait in the queue

Working with conditional variables

- ▶ Execute an action in the thread, when $x == 0$

```
action() {  
    ...  
    pthread_mutex_lock(&m);  
    while (x != 0)  
        pthread_cond_wait(&cv, &m);  
        Action;  
    pthread_mutex_unlock(&m);  
    ...  
}  
  
counter() {  
    ...  
    pthread_mutex_lock(&m);  
    x--;  
    if (x == 0)  
        pthread_cond_signal(&cv);  
    pthread_mutex_unlock(&m);  
    ...  
}
```



Example: Sum elements in a matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

Task: $total = \sum_{i=1}^n \sum_{j=1}^n a_{ij}$

Sequential program:

```
double A[n, n];  
...  
total = 0;  
for (i = 1 to n)  
    for (j = 1 to n)  
        total += A[i][j];  
  
write(total);
```

Example: Parallel sum of elements in a matrix

- ▶ Adding along strips.
- ▶ Each worker aggregates elements in one stripe.
- ▶ Worker 0 displays the final result.
- ▶ It is necessary to carry out a barrier.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n\,n-1} & a_{n\,n} \end{pmatrix}$$

strip 0 – worker 0

strip Nw – worker Nw

Example: Parallel sum of elements in a matrix

Solution in
pseudo code

```
double A[n, n];
double sums[Nw] = ([Nw] 0);
int stripSize = n/Nw;      // integer number

process Worker[id= 1 to Nw] {
    int total, i, j, first, last;

    /* determine first and last rows of my strip */
    first = id*stripSize;  last = first + stripSize - 1;

    /* sum values in my strip */
    total = 0;
    for (i = first to last)
        for (j = 0 to n-1)
            total += A[i][j];
    sums[id] = total;

    Barrier();            //wait for all to finish,

    if (id == 0) {        //first worker prints the final result
        total = 0;
        for (i = 0 to n) total += sums[i];
        write("the total is %d", total);
    }
}
```

Barrier for parallel sum of elements in a matrix

Solution in C with PTHREADS

```
pthread_mutex_t barrier;
pthread_cond_t go;
int numArrived = 0;

void Barrier() {
    pthread_mutex_lock(&barrier);
    numArrived++;
    if (numArrived == numWorkers) {
        numArrived = 0;
        pthread_cond_broadcast(&go);
    }
    else
        pthread_cond_wait(&go, &barrier);
    pthread_mutex_unlock(&barrier);
}
```

Example: Parallel sum of elements in a matrix

```
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 10000 /* maximum matrix size */
#define MAXWORKERS 4 /* maximum number of workers */

pthread_mutex_t barrier; /* mutex lock for the barrier */
pthread_cond_t go; /* condition variable for leaving */
int numWorkers;
int numArrived = 0; /* number who have arrived */

/* a reusable counter barrier */
void Barrier() {
    pthread_mutex_lock(&barrier);
    numArrived++;
    if (numArrived == numWorkers) {
        numArrived = 0;
        pthread_cond_broadcast(&go);
    } else
        pthread_cond_wait(&go, &barrier);
    pthread_mutex_unlock(&barrier);
}

int size, stripSize; /* assume size is multiple of numWorkers */
int sums[MAXWORKERS];
int matrix[MAXSIZE][MAXSIZE];
void *Worker(void *);
```

Parallel sum of elements in a matrix

```
/* read command line, initialize, and create threads */
int main(int argc, char *argv[]) {
    int i, j;
    pthread_attr_t attr;
    pthread_t workerid[MAXWORKERS];

    /* set global thread attributes */
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* initialize mutex and condition variable */
    pthread_mutex_init(&barrier, NULL);
    pthread_cond_init(&go, NULL);

    /* read command line */
    size = atoi(argv[1]);
    numWorkers = atoi(argv[2]);
    stripSize = size/numWorkers;

    /* initialize the matrix */
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            matrix[i][j] = 1;

    /* create the workers, then exit */
    for (i = 0; i < numWorkers; i++)
        pthread_create(&workerid[i], &attr, Worker, (void *) i);
    pthread_exit(NULL);
}
```

Parallel sum of elements in a matrix

```
/* Each worker sums the values in one strip of the matrix.
   After a barrier, worker(0) computes and prints the total */
void *Worker(void *arg) {
    int myid = (int) arg;
    int total, i, j, first, last;

    printf("worker %d (pthread id %d) has started\n", myid, pthread_self());

    /* determine first and last rows of my strip */
    first = myid*stripSize;
    last = first + stripSize - 1;

    /* sum values in my strip */
    total = 0;
    for (i = first; i <= last; i++)
        for (j = 0; j < size; j++)
            total += matrix[i][j];
    sums[myid] = total;
    Barrier();
    if (myid == 0) {
        total = 0;
        for (i = 0; i < numWorkers; i++)
            total += sums[i];
        printf("the total is %d\n", total);
    }
}
```

Standard program functions for secure thread work

- ▶ Standard program functions must enable secure thread work.
- ▶ This means that several simultaneous executions of the same function should not affect one another.
- ▶ Two types of standard functions:
 - ▶ re-entrant function: multiple implementations of the same function in the multithreaded program (simultaneously, interlaced, nesting) should not have an impact on the individual implementation,
 - ▶ thread-safe function: these are functions that are either re-entrant or their implementation is protected by some mechanism of mutual exclusion.
- ▶ Use this macro before first include to instruct the compiler to use safe versions:

```
#ifndef _REENTRANT
#define _REENTRANT
#endif

#include <pthread.h>
#include <stdio.h>
...
```

Semafores in PTHREADS

- ▶ Semaphores were added to pthreads as an extension of the basic standard. The pthreads library is implemented with conditional variables and locks - mutex.
- ▶ Work with semaphores:

```
#include <pthread.h>
#include <semaphore.h>
```

- ▶ Compile:

```
-lposix4
```

- ▶ The semaphore is a structure of the `sem_t` form, it can take any nonnegative number, which is changed by the operations P (decreasing) and V (increasing)
 - ▶ if the semaphore value is 0 then the operation P stops the execution of the thread that executed this operation until the semaphore value becomes positive.

Semaphores in PTHREADS

- ▶ Declaration `sem_t sem;`
- ▶ Functions:
 - ▶ `sem_wait(&sem)` – `P(sem)` – decrease semaphore value by 1 if the value is positive.
 - ▶ `sem_post(&sem)` – `V(sem)` – increase the semaphore value.
- ▶ Example:

```
sem=sem_open("ime", O_CREAT, 0644, 1); // sem = 1
...
sem_wait(&sem); // P(sem)
CS
sem_post(&sem); // V(sem)
```

Simple example of P/C with mutex semaphores

```
int buf;
sem full = 0, empty = 1;

process Producer {
    while(1) {
        P(empty);
        buf = data;
        V(full);
    }
}

process Consumer {
    while(1) {
        P(full);
        result = buf;
        V(empty);
    }
}
```

pc.sems.c

P/C: several producers/consumers – buffer size is 1

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
    while (true) {
        ...
        /* produce data, then deposit it in the buffer */
        → P(empty);
        buf = data;
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        /* fetch result, then consume it */
        → P(full);
        result = buf;
        V(empty);
        ...
    }
}
```

pc.sems.multi.c

P/C: several producers/consumers – buffer size is n

```
typeT buf[n];      /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;      /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion */
process Producer[i = 1 to M] {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        P(mutexD);
        buf[rear] = data; rear = (rear+1) % n;
        V(mutexD);
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        fetch message result and consume it;
        P(full);
        P(mutexF);
        result = buf[front]; front = (front+1) % n;
        V(mutexF);
        V(empty);
        ...
    }
}
```

pc.sems.multi_boundbuf.c

The 5 philosophers problem

philosophers.c

Programming III

Parallel and distributed programming

Monitors

Janez Žibert, Jernej Vičič UP FAMNIT

Overview (chapter 5)

- ▶ The introduction, implementation and operation of monitors
 - ▶ what is a monitor?
 - ▶ conditional variables
 - ▶ monitors usage
- ▶ Examples of using monitors:
 - ▶ limited buffer: conditional synchronization
 - ▶ the problem of simultaneous reading and writing: synchronization with signals
 - ▶ scheduling according to the SJN principle: waiting for priority
 - ▶ the barber problem: principle of dating (rendezvous)
 - ▶ organizing disk access - all examples collected together.

Summary ...

- ▶ Synchronization mechanisms:
 - ▶ locks
 - ▶ general CS locking/unlocking mechanism,
 - ▶ waiting for entry is busy wait.
 - ▶ conditional variables
 - ▶ waiting for entry is in sleep,
 - ▶ the process that is waiting is stopped and placed in a queue of paused processes (FIFO type),
 - ▶ when a conditional variable receives a signal, the first process from the queue is set to run
 - ▶ semaphores
 - ▶ a combination of locks and conditional variables
 - ▶ a general principle for solving problems that involve mutual exclusion and conditional synchronization
 - ▶ weaknesses:
 - ▶ programming at a low level: mixing synchronization mechanisms with tasks that solve the parallel problem. There may be errors.
 - ▶ semaphores can be used for both types of synchronization, mutual exclusion and conditional synchronization, which means that the same semaphore can be used for both types of synchronization
 - ▶ monitors
 - ▶ more structured synchronization mechanism,
 - ▶ defined as objects, higher programming level,
 - ▶ both types of synchronization are implemented.

Monitors

- ▶ Monitors combine variables, functions over variables, and initial setup procedures in a common abstract data structure.
 - ▶ Variables:
 - ▶ the values of the monitor variables represent the state of the monitor.
 - ▶ Functions:
 - ▶ Accessing (modifying/displaying) variables is enabled only through the functions defined in monitor procedures.
 - ▶ The procedures in the monitor are executed according to the principle of mutual exclusion.
 - ▶ Initial Setup Procedures:
 - ▶ Monitor Initialization Procedures - Initial Creation and settings. Monitor status at the beginning
 - ▶ In the Java programming language, they are treated as classes.
- ▶ Monitors were first presented in 70ties – T. Hoare:
 - ▶ implementation of a monitor with a combined binary semaphore
- ▶ Monitor popularization: Java.

Monitor definition

- ▶ Treat a monitor as an object that has the following declaration:

```
monitor monitorName {  
    variable declaration;  
    monitor initialization;  
    procedures;  
}
```

- ▶ monitor initialization:
 - ▶ it takes place when a monitor instance is created,
 - ▶ initialization is needed before we can even start using monitor procedures,
 - ▶ variables and procedures are created at initialization and set to the initial values
- ▶ A process can only access monitor procedures in the following way:

```
call monitorName.procedureName (arguments);
```

Synchronization procedures in monitors

- ▶ The principle of mutual exclusion of monitor procedures:
 - ▶ each time a specific monitor is executed(used), the monitor will have its own binary semaphore - lock.
 - ▶ Implementation of mutual exclusion:
 - ▶ before starting the process with the monitor procedure, it must acquire the monitor's lock.
 - ▶ If the lock is already locked, the implementation of the process is stopped and the process is put in a queue.
 - ▶ The lock is unlocked when the process finishes with the monitor procedure.
 - ▶ Conditional synchronization principle:
 - ▶ each monitor can use conditional variables.
 - ▶ The process that uses the monitor can wait for its execution according to the conditional variable that is defined in the monitor,
 - ▶ the process which is waiting for a conditional variable releases the lock.
 - ▶ Another process, which also uses this monitor, can send a signal via a conditional variable to the pending process that is set to run for execution.

Operations on conditional variables

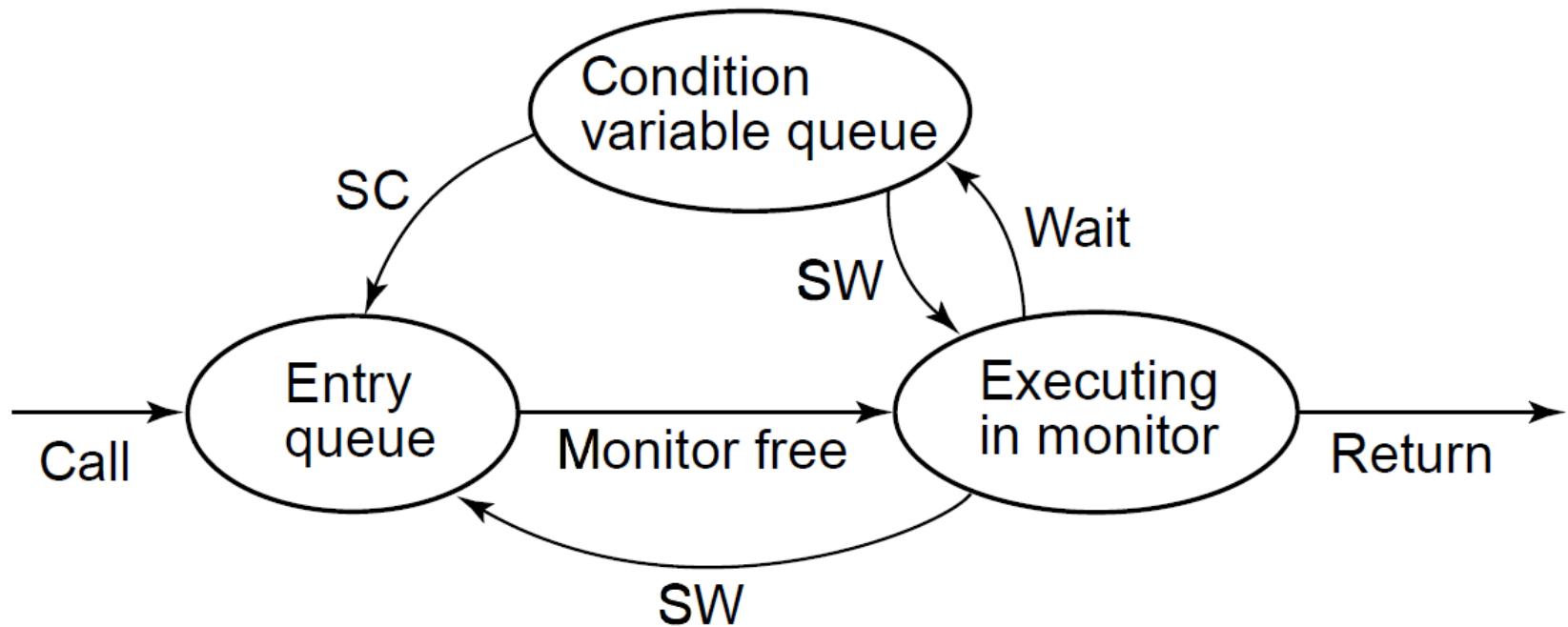
- ▶ Declaration of a conditional variable **cond cv**
- ▶ Operations:

wait(cv, lock)	release the lock and put process at the end of the waiting queue
wait(cv, rank, lock)	the process is stoped, releas the lock, and put the process into a queue for stopped processes at a place for the rank (the processes are sorted ascending by rank)
signal(cv)	wake up the process that is at the begining of the queue
signal_all(cv)	wake up all the processes of the queue and put them in a queue for execution
empty(cv)	true , if the queue is empty
minrank(cv)	return the rank of a process that is at the beginning of the queue (this is the smallest rank of processes in a queue, since the processes in the queue are arranged from the smallest to the highest ranking)

Signalization types in **cond** **cv**

- ▶ What to do if a process sends a signal for execution to another process?
 - ▶ Two processes could continue the execution (but only one can). Which one?
- ▶ Signal and Continue (SC)
 - ▶ process, that sends the signal continues with execution,
 - ▶ process, that receives the signal waits for execution,
 - ▶ process, that receives the signal is awaikened and put into the waiting queue.
- ▶ Signal and Wait (SW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put into the waiting queue.
- ▶ Signal and Urgent Wait (SUW)
 - ▶ process, that sends the signal waits for the execution,
 - ▶ process, that receives the signal starts the execution,
 - ▶ process, that sends the signal is put at the start of the waiting queue.

Synchronization diagram in monitors



SC – signal and continue
SW – signal and wait

Semaphore implementation with monitors

```
monitor Semaphore {  
    int s = 0;    ## s >= 0  
    cond pos;    # signaled when s > 0  
    procedure Psem() {  
        while (s == 0) wait(pos);  
        s = s-1;  
    }  
    procedure Vsem() {  
        s = s+1;  
        signal(pos);  
    }  
}
```

- ▶ Variable s is the value of monitor.
- ▶ procedure Psem is the operation P
 - ▶ when the process calls the Psem procedure, it waits until s == 0, then it can resume execution, decrease the value by 1
- ▶ the procedure Vsem is the operation V
 - ▶ the procedure first increments value by 1, and then signals conditional variable pos to wake up the process that is waiting
- ▶ The difference between SC and SW signaling
- ▶ SW: when the signal(pos) operation is performed in the procedure Vsem a process, which was awakened, is immediately executed,
- ▶ SC: when the signal(pos) operation is performed in the procedure Vsem, the signaled process must wait for the execution. This means that in the meantime, another process can be carried out, which reduces by 1 in the Psem operation, and therefore the original process must check again if it is positive. It's not a FIFO semaphore.
 - ▶ in the SW case we can substitute:
~~while (s==0) wait(pos); z
if (s==0) wait(pos);~~

FIFO semaphore implementation with monitors

```
monitor FIFOsemaphore {
    int s = 0; ## s >= 0
    cond pos; # signaled when s > 0

    procedure Psem() {
        if (s == 0)
            wait(pos);
        else
            s = s-1;
    }

    procedure Vsem() {
        if (empty(pos))
            s = s+1;
        else
            signal(pos);
    }
}
```

- ▶ In order for the previous semaphore to act as a FIFO in both ways of signaling, it is necessary to correct the procedure Vsem:
 - ▶ if there is any process that awaits the conditional variable pos (in the Psem procedure), then signal(pos), but do not increase the semaphore s. Increase s only when no process is waiting.
 - ▶ accordingly, Psem should also be corrected:
 - ▶ if the process is waiting for the pos, then do not reduce it by 1, because Vsem did not increase it, otherwise decrease by 1
- ▶ *Passing the condition*
 - ▶ the procedure that sends the signal, and so wakes up the process that awaits, implicitly also says that it is a positive one
 - ▶ a process that wakes up does not need to check this condition again (in the SC case)
 - ▶ All other processes do not see this condition and therefore do not wait for the conditional test.

Difference between monitors and semaphores

- ▶ operation **wait** in monitor and operation **P** in semaphore stop the process
- ▶ operation **signal** in monitor and operation **V** in semaphore wakes the process
- ▶ But:
 - ▶ 1. difference:
 - ▶ **wait** always stops a process until operation **signal**
 - ▶ operation **P** stops a process only when semaphore is 0
 - ▶ 2. difference :
 - ▶ signal has no effect, if no process waits for a conditional variable.
 - ▶ operation **V** wakes up a process that waits by increasing the semaphore (always doing something)
 - ▶ So: the operation of the signal is not remembered.
 - ▶ Therefore, the conditional synchronization is programmed differently in both cases.

Monitor usage

- ▶ Monitor is the data structure (class) that is used:
 - ▶ to perform atomic operations and commands over common variables that are encapsulated in the monitor.
 - ▶ to control access to shared resources outside monitors.
 - ▶ Later we will look at the use of monitors for various synchronization techniques
 - ▶ limited buffer: conditional synchronization
 - ▶ the problem of reading and writing common data: synchronization with signals
 - ▶ scheduling according to the SJN principle: waiting for priority
 - ▶ the barber problem: rendezvous principle
 - ▶ organization of disk access: all techniques collected together.

Synchronized access to limited buffer between the producer and the consumer

- ▶ This is an example that we have already dealt with. It is the principle of conditional synchronization when accessing a common buffer which is spatially limited.
- ▶ Suppose we have a buffer with the capacity to store n elements.
- ▶ Implement the buffer for communication between the producer and the consumer as a monitor.
- ▶ Monitor Variables:
 - ▶ **buf[n]** , **front** , **rear** , **count** (counts how many buffer slots are currently occupied)
 - ▶ **cond not full** - stops writing from the producer until the buffer is ready to write
 - ▶ **cond not empty** - stops the consumer to read the data from the buffer until new data is in the buffer
- ▶ Monitor Procedures:
 - ▶ **deposit (typeT)** - inserts the data into the buffer (producer)
 - ▶ **fetch (* typeT)** - Write data from the buffer to local variable type **typeT** (consumer)

Implementation of a limited buffer with a monitor

```
monitor Bounded_Buffer {  
    typeT buf[n];      # an array of some type T  
    int front = 0,      # index of first full slot  
        rear = 0;       # index of first empty slot  
        count = 0;       # number of full slots  
    ## rear == (front + count) % n  
    cond not_full,     # signaled when count < n  
        not_empty;     # signaled when count > 0  
  
    procedure deposit(typeT data) {  
        while (count == n) wait(not_full);  
        buf[rear] = data; rear = (rear+1) % n; count++;  
        signal(not_empty);  
    }  
  
    procedure fetch(typeT &result) {  
        while (count == 0) wait(not_empty);  
        result = buf[front]; front = (front+1) % n; count--;  
        signal(not_full);  
    }  
}
```

Usage:

- Producer: `Bounded_Buffer.deposit(a[i]);`
- Consumer: `Bounded_Buffer.fetch(&b[i]);`

Reading/writing shared data problem

- ▶ This is an example of signal synchronization: we will use the `signal_all` operation.
- ▶ Problem: reading and writing data to the database.
- ▶ Selective intercommunication:
 - ▶ At the same time read operations can be performed
 - ▶ writing must be performed atomically with respect to other operations: it is therefore necessary to ensure mutual exclusion with other writing and reading operations.
- ▶ Monitor implementation options:
 - ▶ the database can be performed as one monitor: reading and writing - two monitor procedures
 - ▶ this ensures the exclusive operation of reading and writing operations (we do not allow concurrent readings) - this is not our goal
 - ▶ idea: monitor is used only to control access to database data
 - ▶ a monitor that performs the control of accessing all processes to database data.

Monitor access control implementation

- ▶ The monitor monitors access to the database.
- ▶ Monitor variables:
 - ▶ **nr** - number of current readers,
 - ▶ **nw** - number of current data writers,
 - ▶ **cond oktoread** - a conditional variable that allows you to wait for processes that read data to access the database,
 - ▶ **cond oktowrite** - a conditional variable that allows waiting for the processes that write data to access the database.
- ▶ Monitor Procedures:
 - ▶ **request_read**:
 - ▶ provides access to data reading in the event that writing to the database is not done, otherwise it is waiting;
 - ▶ **request_write**:
 - ▶ provides access to data writing in the event that writing to the database is not done, otherwise it is waiting;
 - ▶ **release_read**:
 - ▶ the last reading process sends a signal to the process that is waiting for writing (if any);
 - ▶ **release_write**:
 - ▶ signal the possibility of continuing work for all the processes that are waiting for implementation.

Monitor implementation

```
monitor RW_Controller {  
    int nr = 0, nw = 0;    ## (nr == 0 ∨ nw == 0) ∧ nw <= 1  
    cond oktoread;      # signaled when nw == 0  
    cond oktowrite;     # signaled when nr == 0 and nw == 0  
  
    procedure request_read() {  
        while (nw > 0) wait(oktoread);  
        nr = nr + 1;  
    }  
  
    procedure release_read() {  
        nr = nr - 1;  
        if (nr == 0) signal(oktowrite);  # awaken one writer  
    }  
  
    procedure request_write() {  
        while (nr > 0 || nw > 0) wait(oktowrite);  
        nw = nw + 1;  
    }  
  
    procedure release_write() {  
        nw = nw - 1;  
        signal(oktowrite);          # awaken one writer and  
        signal_all(oktoread);       # all readers  
    }  
}
```

Usage:

– Read:

```
RW_Controller.request_read();  
read data from database;  
RW_Controller.release_read();
```

– Write:

```
RW_Controller.request_write();  
write data to database;  
RW_Controller.release_write();
```

Scheduling according to the principle SJN

- ▶ Scheduling tasks according to the SJN (Shortest Job Next) principle: distribute the execution of processes according to the expected time of their execution, or depending on the expected time of access to common resources.
- ▶ This is done with semaphores. In this case it will be implemented with monitors, using the command of performing the waiting process of processes according to the priority:
 - ▶ in conditional variable we usually have FIFO queues:
 - ▶ **wait(c)** - processes are put into a queue of paused processes according to the FIFO principle
 - ▶ but we can use the queue in which processes are ranked by rank
 - ▶ **wait(c, rank)** - the process with the lowest rank is waiting in the first place
 - ▶ Monitor where the processes are placed in a queue for a conditional variable according to ranking can be used to perform the scheduling of tasks according to the SJN principle.

Monitor for the implementation of SJN

- ▶ Scheduling tasks according to the SJN principle can be performed as a monitor that governs access to shared resources so that processes are sorted according to the time of use of the shared resource (from the shortest to the longest).
- ▶ Monitor variables:
 - ▶ **boolean free**
 - ▶ which tells whether the source is free to use or not,
 - ▶ **cond turn**
 - ▶ conditional variable where processes wait to execute in SJN principle
- ▶ Monitor procedures:
 - ▶ **request(int time)**
 - ▶ assigns a common resource to the process if the shared resource is free if it is not it puts the process to wait according to the estimated time of use of the source time.
 - ▶ **release:**
 - ▶ signal the process that is waiting in the first place in a conditional variable. The shared resource is free and it can start the execution,
 - ▶ if there are no pending processes, only release the shared resource.

Monitor implementation for SJN

```
monitor Shortest_Job_Next {  
    bool free = true;    ## Invariant SJN:  see text  
    cond turn;           # signaled when resource available  
  
    procedure request(int time) {  
        if (free)  
            free = false;  
        else  
            wait(turn, time);  
    }  
  
    procedure release() {  
        if (empty(turn))  
            free = true  
        else  
            signal(turn);  
    }  
}
```

Usage:

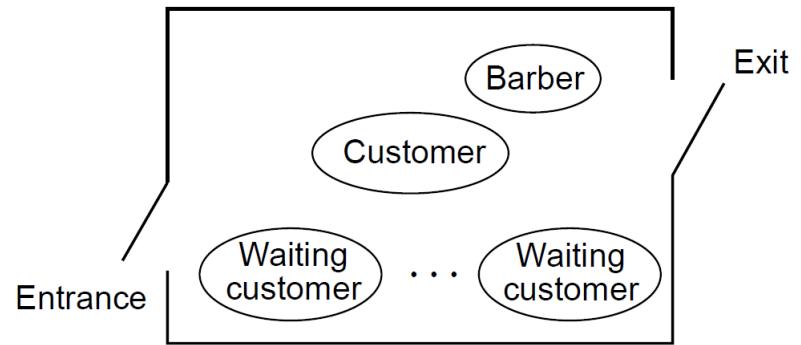
```
Shortest_Job_Next.request(mytime);  
use shared resource;  
Shortest_Job_Next.release();
```

Rendezvous

- ▶ This is an example of using monitors in client-server applications:
 - ▶ One server and several clients.
 - ▶ Interaction is one-on-one: two processes get together and arrange common affairs (rendezvous):
 - ▶ first, they need to agree on a date: signaling with conditional variables.
 - ▶ the process on the server must wait for the client process to "serve it"
 - ▶ the client must wait for the server to be free so that it can "be served"
 - ▶ the server can have a date with all clients, the client only with the server.

Barber problem: rendezvous

- ▶ The barber has his own place where he shaves his clients in the following way:
 - ▶ Wait for the client;
 - ▶ the client is shaved;
 - ▶ signal the client to leave;
 - ▶ wait the client to leave.
- ▶ A client who would like to get the hair
 - ▶ come into the shop;
 - ▶ wait until the barber is available;
 - ▶ get shaved;
 - ▶ leave the shop.
- ▶ The barber is a server, the customer is a client.
- ▶ Rendezvous: The barber shaves the client.
- ▶ Problem: synchronization of barber and client processes
- ▶ Implementation: monitor is a site that allows "dating" between the barber (server) and clients (clients).



Monitor to implement out a barber shop

- ▶ Procedures:
 - ▶ `get_haircut` – used by the client – customer requests the shaving
 - ▶ `get_next_customer` – used by the server – barber requests a new customer
 - ▶ `finished_cut` – used by server – barber end the shaving process
- ▶ Monitor usage:
 - ▶ barber process (server):

```
while (true) {  
    Barber_Shop.get_next_customer();  
    shave the client;  
    Barber_Shop.finished_cut();  
}
```

- ▶ customer process (client):

```
Barber_Shop.get_haircut();
```

Monitor variables

- ▶ Flags – used to signal status:
 - ▶ **barber** – barber is free (server can receive requests)
 - ▶ **chair** – chair is taken (server is serving a request)
 - ▶ **open** – shop door is open (request was completed, the answer is ready)
when **open == 0** – client has left the shop
- ▶ Conditional variables - waiting points:
 - ▶ used to synchronize between the barber and client (rendezvous).
 - ▶ The barber waits:
 - ▶ **chair_occupied** – for the client to seat on the chair: `<await (chair > 0);>`
 - ▶ **customer_left** – for the client that was served to leave:
`<await (open==0);>`
 - ▶ Client waits:
 - ▶ **barber_available** – for the barber to be: `<await (barber > 0);>`
 - ▶ **door_open** – to be served: `<await (door > 0);>`

The implementation of a monitor for the barber problem

▶ Usage:

- ▶ Client

```
Barber_Shop.get_haircut();
```

- ▶ Barber

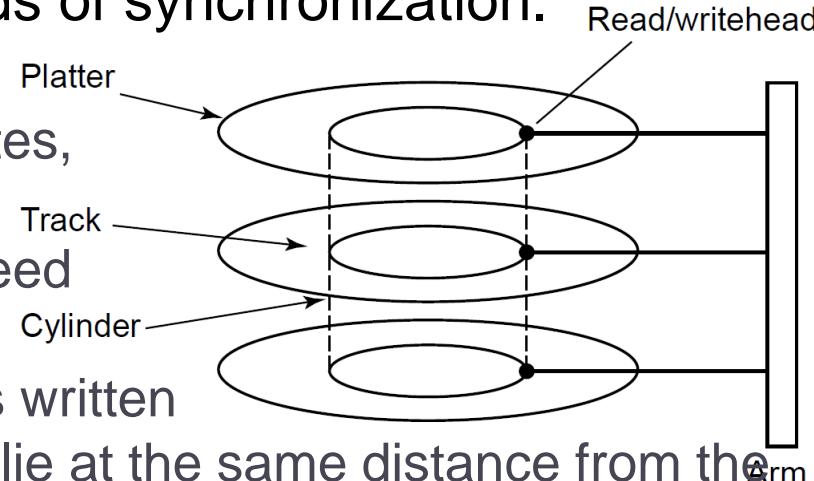
```
while (true) {  
    Barber_Shop.get_next_customer();  
    cut the hair;  
    Barber_Shop.finished_cut();  
}
```

- ▶ The monitor has several waiting points.
- ▶ By doing so, we provide a meeting:server with a single client.

```
monitor Barber_Shop {  
    int barber = 0, chair = 0, open = 0;  
    cond barber_available;      # signaled when barber > 0  
    cond chair_occupied;       # signaled when chair > 0  
    cond door_open;            # signaled when open > 0  
    cond customer_left;        # signaled when open == 0  
    procedure get_haircut() {  
        while (barber == 0) wait(barber_available);  
        barber = barber - 1;  
        chair = chair + 1; signal(chair_occupied);  
        while (open == 0) wait(door_open);  
        open = open - 1; signal(customer_left);  
    }  
    procedure get_next_customer() {  
        barber = barber + 1; signal(barber_available);  
        while (chair == 0) wait(chair_occupied);  
        chair = chair - 1;  
    }  
    procedure finished_cut() {  
        open = open + 1; signal(door_open);  
        while (open > 0) wait(customer_left);  
    }  
}
```

Implementation of the system for scheduling disk access

- ▶ This is a bigger problem than the previous one and combines several techniques and methods of synchronization.
- ▶ Hard disk:
 - ▶ consisting of several roundsplates,
 - ▶ mounted on an axle,
 - ▶ which rotate with a constant speed
 - ▶ recording track:
 - ▶ on one disk where the data is written
 - ▶ cylinder: circles on all discs that lie at the same distance from the axle of rotation
 - ▶ reading/writing:
 - ▶ moving arm, where the read/write heads are attached. The arm moves in such a way that it approaches or moves away from the disk rotation axle, and thus, data can be written/read on all discs at the same distance (cylinders).



Hard disk access

- ▶ Using I/O instructions to transfer data from the memory to the hard disk (and vice versa). The arguments for commands to access the disk are:
 - ▶ opcode - whether it's reading or writing to the disc
 - ▶ disk address space (where data will be read/written)
 - ▶ the address of the memory space (where the data will be written/read)
- ▶ how much data will be transferred between the disc and the memory
- ▶ Accessing data on your hard drive:
 - ▶ put the read/write head at the position of the tape where you will read or write the information
 - ▶ wait for the disc to rotate until it reaches the position on the tape where the data is stored or written,
 - ▶ read or write data from/to disk.

Scheduling Disk Access

- ▶ Assume we have several processes that want to read or write data to/from disk. The execution of commands for accessing the disk should therefore be sorted accordingly.
- ▶ Problem: Efficient deployment of disk access is required.
- ▶ Requirements: Minimize the average access time to the disk.
- ▶ Disk access time consists of:
 - ▶ time of movement of the head to the cylinder
 - ▶ the time of rotation of the disc to reach the desired position on the rotational plate
 - ▶ time of data transfer from memory to disk or vice versa (data transmission time)
- ▶ Relationships between times:
 - ▶ the time of rotation of the disc is much shorter than the time of moving the head to the cylinder, the data transmission time depends on the size of data being transmitted.
- ▶ Our mission: reduce the time of moving the head to the cylinder.

Disk allocation strategies

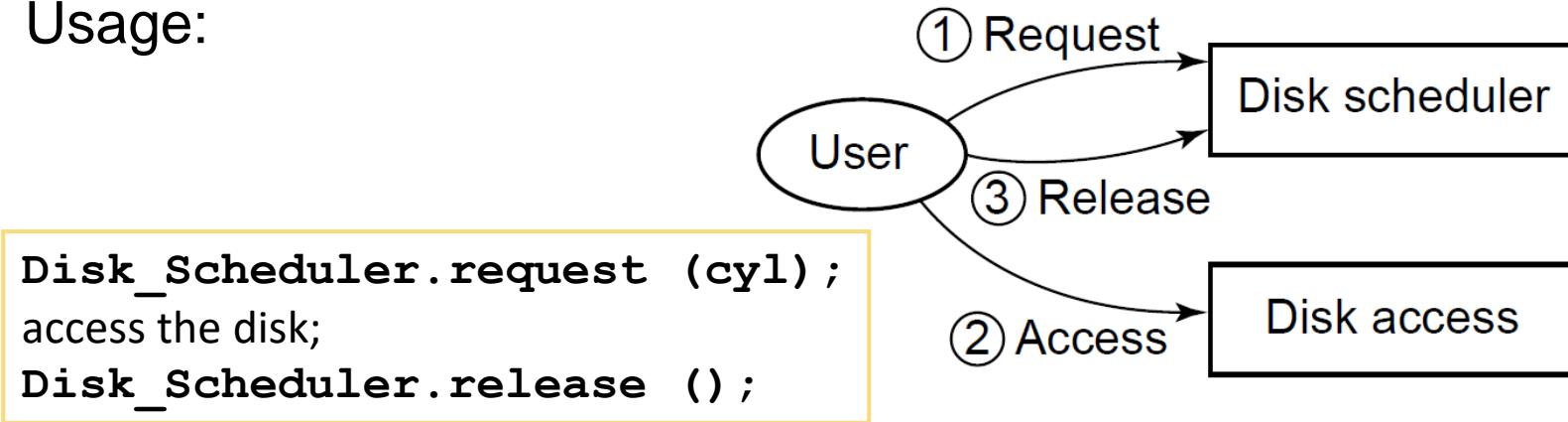
- ▶ **SST (shortest seek time)**
 - ▶ processes that require access to the disk should be arranged in such a way that it is next executed the process that needs the shortest time to move the head,
 - ▶ minimizing the time of movement, but the process is not fair: some processes can starve.
 - ▶ It is used on UNIX operating systems
- ▶ **SCAN (LOOK)**
 - ▶ processes that require access to the disk should be arranged in such a way that the first in turn is the one that needs the shortest time to move the head in the direction of moving the head of one process in front of it:
 - ▶ so first we enable one-way access, then we change the direction and allow access in the other direction (operation principle of the lift)
 - ▶ great differences in waiting time for access
- ▶ **CSCAN (CLOOK)**
 - ▶ the same as for SCAN, but we only have one direction, e.g. from the outside to the inside,
 - ▶ minimize the differences in waiting time for the access.

Possible implementations

- ▶ Let's assume the access to the disk according to the CSCAN strategy.
- ▶ Option 1: autonomous deployment of disk access scheduling as when making access to shared resources:
 - ▶ the implementation of a monitor that controls access to the disk (similar to the problem of reading and writing shared data)
- ▶ Option 2: perform an access scheduler that acts as an intermediary between the processes requiring access and the driver that manages the hard disk: similar to the problem of the barber:
 - ▶ processes requiring access are clients,
 - ▶ the driver that manages the hard drive is the barber,
 - ▶ our problem: we are scheduling the requests for access to the barber according to the CSCAN strategy,
 - ▶ the data we get from the driver is forwarded to the process.
- ▶ Option 3: use two monitors that are nested
 - ▶ two monitors:
 - ▶ one for scheduling access requests, the other for accessing the disk,
 - ▶ the first monitor calls the second – nesting.

1. possibility: autonomous monitor implementation

- ▶ The monitor is intended primarily to control access to the disk.
- ▶ Usage:



- ▶ The process requiring disk access asks the access monitor. If access is not possible, it waits for authorization to start reading or writing to disk. When performing read or write operations, it frees access to other processes.
- ▶ All processes must use these three disk access operations, using the same protocol:
 - ▶ request access, execute access, release access.

Monitor synchronization

- ▶ Monitor procedures:
 - ▶ `request(cyl)` , `release()`
- ▶ Monitor variables:
 - ▶ `int position` – head position (< 0 – means the head is at the starting position)
 - ▶ `cond scan[0:1]` – two conditional variables to perform the organization of waiting for access to the disk in the first pass and the second pass through the disk
 - ▶ `int c` – variable which points to a conditional variable for the first pass through the disk
 - ▶ `int n` – variable which points to a conditional variable for the next pass through the disk
- ▶ The monitor must provide the following features:
 - ▶ In the conditional variables `scan[c]` and `scan[n]` are two stopped queues arranged according to rank,
 - ▶ all elements (all processes) in `scan[c]` must request disk access to the positions that are $> \text{position}$,
 - ▶ all elements (all processes) in `scan[n]` must request disk access to the positions that are $\leq \text{position}$,
 - ▶ if `position== -1`, both queues are empty (`scan[c]` and `scan[n]`)

Monitor implementation

```
monitor Disk_Scheduler { ## Invariant DISK
    int position = -1, c = 0, n = 1;
    cond scan[2];    # scan[c] signaled when disk released
    procedure request(int cyl) {
        if (position == -1)  # disk is free, so return
            position = cyl;
        elseif (position != -1 && cyl > position)
            wait(scan[c], cyl);
        else
            wait(scan[n], cyl);
    }
    procedure release() {
        int temp;
        if (!empty(scan[c]))
            position = minrank(scan[c]);
        elseif (empty(scan[c]) && !empty(scan[n])) {
            temp = c; c = n; n = temp;      # swap c and n
            position = minrank(scan[c]);
        }
        else
            position = -1;
        signal(scan[c]);
    }
}
```

Usage:

```
Disk_Scheduler.request (cyl);
access the disk;
Disk_Scheduler.release ();
```

Scheduling access between the processes and the driver in the intermediate monitor

- ▶ The communication between processes requiring access and the driver can be additionally equipped with a monitor that classifies the accesses according to the sorting strategy.



- ▶ The problem is similar to that of a barber, but it is more complex.
- ▶ Processes use a simpler disk access protocol:
- ▶ `Disk_Interface.use_disk(cyl, op, req and res pointers);`
- ▶ The driver receives requests from the intermediate monitor in this way:

```
Disk_Interface.get_next_request(&request);  
Process request;  
Disk_Interface.finished_transfer(status);
```

Schematic execution of the intermediate monitor

```
monitor Disk_Interface
    permanent variables for status, scheduling, and data transfer
    procedure use_disk(int cyl, transfer and result parameters) {
        wait for turn to use driver
        store transfer parameters in permanent variables
        wait for transfer to be completed
        retrieve results from permanent variables
    }
    procedure get_next_request(someType &results) {
        select next request
        wait for transfer parameters to be stored
        set results to transfer parameters
    }
    procedure finished_transfer(someType results) {
        store results in permanent variables
        wait for results to be retrieved by client
    }
}
```

Usage:

Process, that accesses the disk:

```
Disk_Interface.use_disk(cyl, op,
    req and res pointers);
```

Drive:

```
Disk_Interface.get_next_request(&request);
```

Process the request;

```
Disk_Interface.finished_transfer(status);
```

Implementation of the intermediate monitor

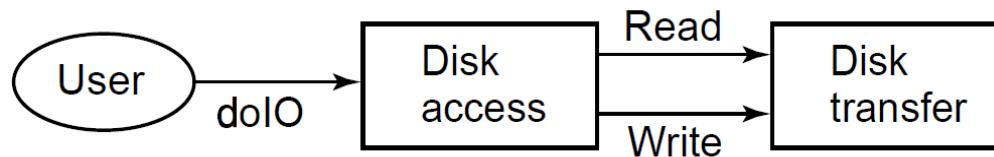
```
monitor Disk_Interface {
    int position = -2, c = 0, n = 1, args = 0, results = 0;
    cond scan[2];
    cond args_stored, results_stored, results_retrieved;
    argType arg_area; resultType result_area;
    procedure use_disk(int cyl; argType transfer_params;
                        resultType &result_params) {
        if (position == -1)
            position = cyl;
        elseif (position != -1 and cyl > position)
            wait(scan[c], cyl);
        else
            wait(scan[n], cyl);
        arg_area = transfer_params;
        args = args+1; signal(args_stored);
        while (results == 0) wait(results_stored);
        result_params = result_area;
        results = results-1; signal(results_retrieved);
    }
    procedure get_next_request(argType &transfer_params) {
        int temp;
        if (!empty(scan[c]))
            position = minrank(scan[c]);
        elseif (empty(scan[c]) && !empty(scan[n])) {
            temp = c; c = n; n = temp;      # swap c and n
            position = minrank(scan[c]);
        }
        else
            position = -1;
        signal(scan[c]);
        while (args == 0) wait(args_stored);
        transfer_params = arg_area; args = args-1;
    }
    procedure finished_transfer(resultType result_vals) {
        result_area := result_vals; results = results+1;
        signal(results_stored);
        while (results > 0) wait(results_retrieved);
    }
}
```

} wait similar as the previous monitor

} send a signal similar as the previous monitor

3. possibility: 2 monitor usage

- ▶ 1. possibility: one monitor: processes must access the disk according to a specific protocol.
 - ▶ procesi morajo dostopati do diska po točno določenem protokolu.
- ▶ 2. possibility : proxy monitor:
 - ▶ monitor med procesi in gonilnikom, monitor between processes and driver,
 - ▶ uporaba enostavna, izvedba kompleksna, easy to use, complex implementation,
 - ▶ predpostavka: imamo gonilnik Assumption: we have a driver
- ▶ 3. possibility: combination of both approaches: 2 monitors
 - ▶ one monitor for scheduling of the process access,
 - ▶ second monitor for actual access



Implementation of disk access with 2 monitors

```
monitor Disk_Access {
    permanent variables as in Disk_Scheduler;
    procedure doIO(int cyl; transfer and result arguments) {
        actions of Disk_Scheduler.request;
        call Disk_Transfer.read or Disk_Transfer.write;
        actions of Disk_Scheduler.release;
    }
}
```

- ▶ The **Disk_Access** monitor is similar to the **Disk_Scheduler** monitor, but it has only one **doIO** procedure where a call from another **Disk_Transfer** monitor is performed that performs reading or writing operations on the disk.
- ▶ Monitor **Disk_Access** calls the **Disk_Transfer** monitor
 - ▶ this is a nested call

Nested monitor calls

- ▶ Nested monitor call:
 - ▶ The process that performs the procedure within one monitor calls the procedure of another monitor, which means that it temporarily leaves the first monitor.
- ▶ Closed call:
 - ▶ keep the exclusivity of execution in the monitor from which we call another monitor - the process requires two locks for this implementation
 - ▶ cons: a deadlock may occur:
 - ▶ for example: the process that performed the nested call stopped by waiting with the wait command, while the other monitor performs the nested call of the first monitor.
- ▶ Open call:
 - ▶ release the lock in monitor from which we call another monitor, the process has exclusive access only to the second monitor.
 - ▶ when the procedure of the other monitor is performed, the process again obtains exclusive access to the execution of the first monitor.
 - ▶ cons: making such a call is more demanding for programming.
- ▶ Monitors for disk access:
 - ▶ **Disk Transfer** monitor call inside the **Disk Access** monitor is nested and must be open type. Otherwise only one process can access both monitors.

Programming III

Parallel and distributed programming

Multithreaded programming in JAVA. Java and monitors.

Janez Žibert, Jernej Vičič UP FAMNIT

Overview (chapter 5.4 in www)

- ▶ Brief presentation of JAVA programming language
- ▶ Multithreaded programming in JAVA
 - ▶ Runnable interface
 - ▶ Class Thread
 - ▶ two ways of implementing threads
- ▶ Thread Synchronization in JAVA
 - ▶ synchronized methods
 - ▶ monitors.
- ▶ Examples:
 - ▶ producer/consumer, use of limited buffer,
 - ▶ writing/reading problem.
- ▶ Parallel programming tools in the `java.util.concurrent` collection
 - ▶ just a review of the methods and classes from the collection
- ▶ References:
 - ▶ Book: chapter 5.4.
 - ▶ multithreaded programming in Java:
<http://java.sun.com/docs/books/tutorial/essential/concurrency/>
 - ▶ `java.util.concurrent`:
<http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>

Threads in JAVA

- ▶ Threads are defined as LW processes:
 - ▶ have their own stack and context of execution,
 - ▶ access all variables within their reach, (scope)
- ▶ Threads are implemented by extending the Thread class or implementing the Runnable interface.
 - ▶ The key methods are **start** and **run**.
- ▶ Both structures are embedded in the standard java library: `java.lang`.

Threads in JAVA

- ▶ Create a thread:
 - ▶ `Thread foo = new Thread();`
- ▶ Start the thread:
 - ▶ `foo.start();`
 - ▶ start is just one of the methods in the Thread class
 - ▶ by starting the thread foo, nothing happens because the start() method calls the run() method in the Thread class. This does nothing work in the Thread class.
- ▶ We have to implement the run method from class **Thread**:
 - ▶ 1. way: extending the **Thread**
 - ▶ 2. way: implementing the **run** in the interface **Runnable**

Threads in JAVA

- ▶ by extending the class **Thread**:

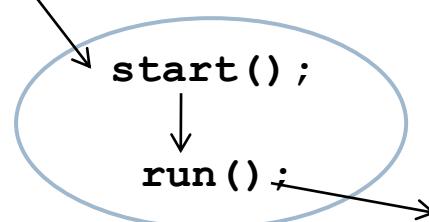
- ▶ Define a new (sub) class of the Thread class by defining the run method and other methods from the Thread class, if necessary.

- ▶ Example:

```
class Simple extends Thread {  
    public void run() {  
        System.out.println("I am a thread.");  
    }  
}
```

- ▶ Usage:

```
Simple s = new Simple();  
s.start(); // execute the run method in object s
```



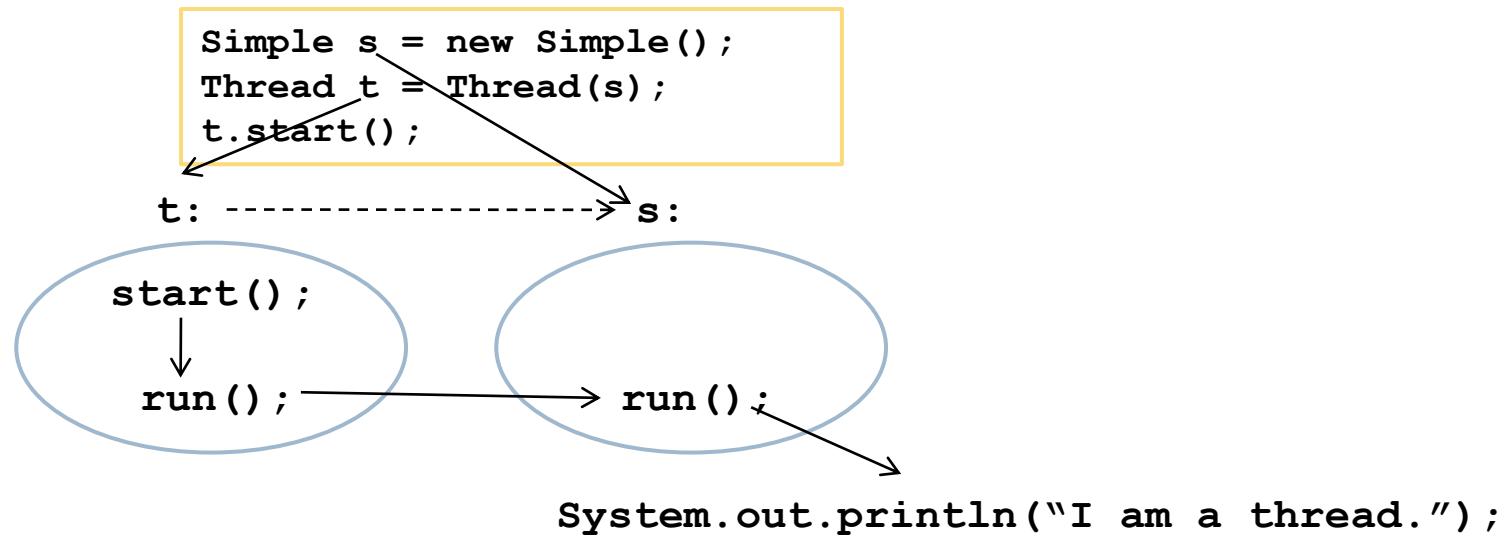
```
System.out.println("I am a thread.");
```

Threads in JAVA

- Implementation of the run method of the interface Runnable

```
class Simple implements Runnable {  
    public void run() {  
        System.out.println("I am a thread.");  
    }  
}
```

- In this case, we create a thread by first creating an object from this class and sending it as a parameter to the constructor of the Thread class. Start the thread with the **start()** method.



Threads in JAVA

- ▶ 1. define a new class either by extending the **Thread** class or implementing the **Runnable** interface.
- ▶ 2. program the run method in the new class: this is the program that the thread will perform.
- ▶ 3. Create an object from the new class with the new operator.
- ▶ 4. Start the execution with the start method.

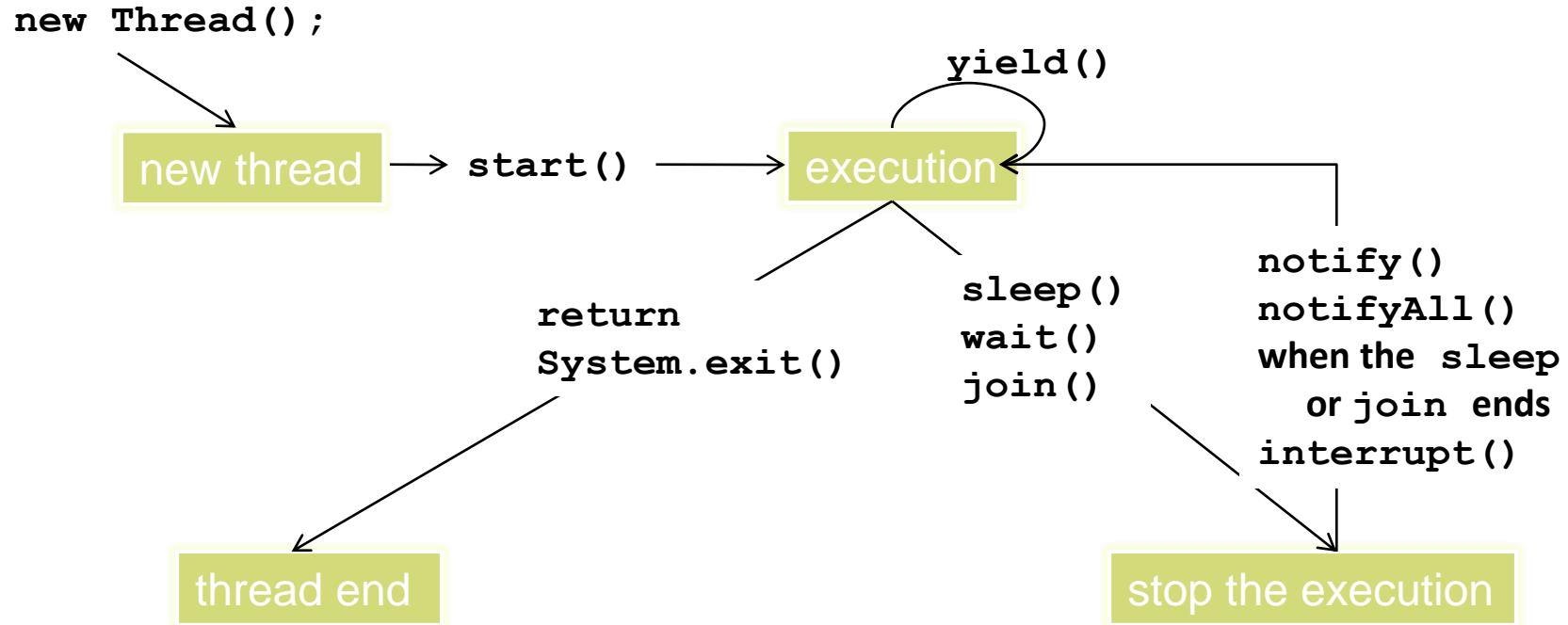
The constructors of the Thread class and thread properties

- ▶ Each thread is created in the java with its own name, belongs to a particular group of threads, and has its own priority. Constructors:
 - ▶ `Thread()`
 - ▶ `Thread(String)`
 - ▶ `Thread(ThreadGroup, String)`
 - ▶ `Thread(Runnable)`
 - ▶ `Thread(ThreadGroup, Runnable)`
 - ▶ `Thread(Runnable, String)`
 - ▶ `Thread(ThreadGroup, Runnable, String)`
- ▶ The properties of the thread can be set or read using the methods in the Thread class:
 - ▶ name, priority, group, thread type (daemon)
 - ▶ The group and the type of thread can not be modified during execution, other properties can be.
- ▶ Priorities:
 - ▶ `MAX_PRIORITY`
 - ▶ `MIN_PRIORITY`
 - ▶ `NORM_PRIORITY`

Thread methods

- ▶ **run()**
 - ▶ This is a method that is overwritten with its code that will be executed in the thread. If we do not implement this method, the thread will not do anything. It is not called directly, but with the start method.
- ▶ **start()**
 - ▶ Start the thread execution: JVM starts to execute the instructions in run method.
- ▶ **join()**
 - ▶ Wait for a thread to finish.
- ▶ **yield()** *getiri salama*
 - ▶ Start context switch.
- ▶ **sleep(long)**
 - ▶ The thread stops its execution for time (in milliseconds), which is specified with the long parameter.
- ▶ **interrupt()**
 - ▶ Stop the thread execution.
- ▶ Setting/query thread properties:
 - ▶ **setPriority(int)** , **getPriority()**
 - ▶ **setName(String)** , **getName()** ,
 - ▶ **setDaemon(boolean)** , **isDaemon()**

Implementation of threads and how the methods work



Communication among threads in JAVA

- ▶ The threads in JAVA are executed simultaneously - at least conceptually.
- ▶ Thread communication:
 - ▶ accessing methods and variables in a shared scope is enabled for all threads;
 - ▶ communication via pipes;
 - ▶ communication through common (shared) objects.
- ▶ The object is shared by several simultaneous threads when multiple threads can use the methods of this object to access the (common) object variables.
 - ▶ The class can be programmed as a monitor if all or some of the methods of the object are declared with the type synchronized, which means that these methods can be implemented by taking into account mutual exclusion.
 - ▶ The class may include methods of the synchronized type and/or the usual methods - these are not implemented by taking into account mutual exclusion.

synchronized methods

- ▶ The word synchronized is reserved for defining mutually exclusive:
 - ▶ methods, if the method is defined with the type synchronized,
 - ▶ the sequence of commands that are captured in the synchronized block.
- ▶ Each object that uses methods (commands) of the synchronized type obtains its own lock (in order to perform mutually exclusive execution).
- ▶ Implementing the method or sequence of commands type synchronized is as follows:
 - ▶ the method (or command sequence) waits to obtain a lock;
 - ▶ the method (or sequence of commands) is performed;
 - ▶ when it stops, the lock is released;
- ▶ So:
 - ▶ method of the synchronized type - the procedure in the monitor
 - ▶ sequence of commands that are captured in synchronized - await command.

Examples of the type `synchronized`

```
class Interfere1 {  
    private int data = 0;  
    public synchronized void update() {  
        data++;  
    }  
}
```

```
class Interfere2 {  
    private int data = 0;  
    public void update() {  
        synchronized(this) {      // lock this object  
            data++;  
        }  
    }  
}
```

Monitors in JAVA

- ▶ Monitors in the java are objects of concrete class implementations that have defined synchronized type methods:
 - ▶ which means that in a multithreaded program, such methods are executed at most one at a time,
 - ▶ Classes may have methods of type synchronized or not
- ▶ Each object - monitor - obtains one lock and one conditional variable:
 - ▶ locks and conditional variables do not need to be explicitly defined, but are generated when the thread or program itself starts,
- ▶ Conditional variable operations are:
 - ▶ in combination with methods of synchronized type:
 - ▶ wait() - monitor procedure wait, release lock and set thread to queue for execution (FIFO type)
 - ▶ notify() - monitor signal procedure, wake the thread that is the first in the queue to put in and put it into the execution queue
 - ▶ notifyall() - monitor signal_all, all threads waiting to wake up and put them in the execution queue
- ▶ A conditional variable in the JAVA is not implemented with a FIFO queue that allows can be sorted by rank (priorities).
- ▶ Signal mode in JAVA: Signal & Continue
- ▶ Nested calls within the monitors are allowed, these are closed type
 - ▶ It can lead to a deadlock: the synchronized type method calls the method synchronized in another object, and that backwards.

Example of a deadlock in nested calls

- ▶ **Review:**
 - ▶ Nested calls within the monitors are allowed, these are closed type
 - ▶ This can lead to a deadlock: the synchronized type method calls the method synchronized in another object, and that backwards.
- ▶ **Example:**
 - ▶ Thread 1 synchronizes on A
 - ▶ Thread 1 synchronizes on B (while synchronized on A)
 - ▶ Thread 1 decides to wait for a signal from another thread before continuing
 - ▶ Thread 1 calls B.wait() thereby releasing the lock on B, but not A.
 - ▶
 - ▶ Thread 2 needs to lock both A and B (in that sequence)
 - ▶ to send Thread 1 the signal.
 - ▶ Thread 2 cannot lock A, since Thread 1 still holds the lock on A.
 - ▶ Thread 2 remain blocked indefinitely waiting for Thread 1
 - ▶ to release the lock on A
 - ▶
 - ▶ Thread 1 remain blocked indefinitely waiting for the signal from
 - ▶ Thread 2, thereby
 - ▶ never releasing the lock on A, that must be released to make
 - ▶ it possible for Thread 2 to send the signal to Thread 1, etc.

Example of a deadlock in nested calls

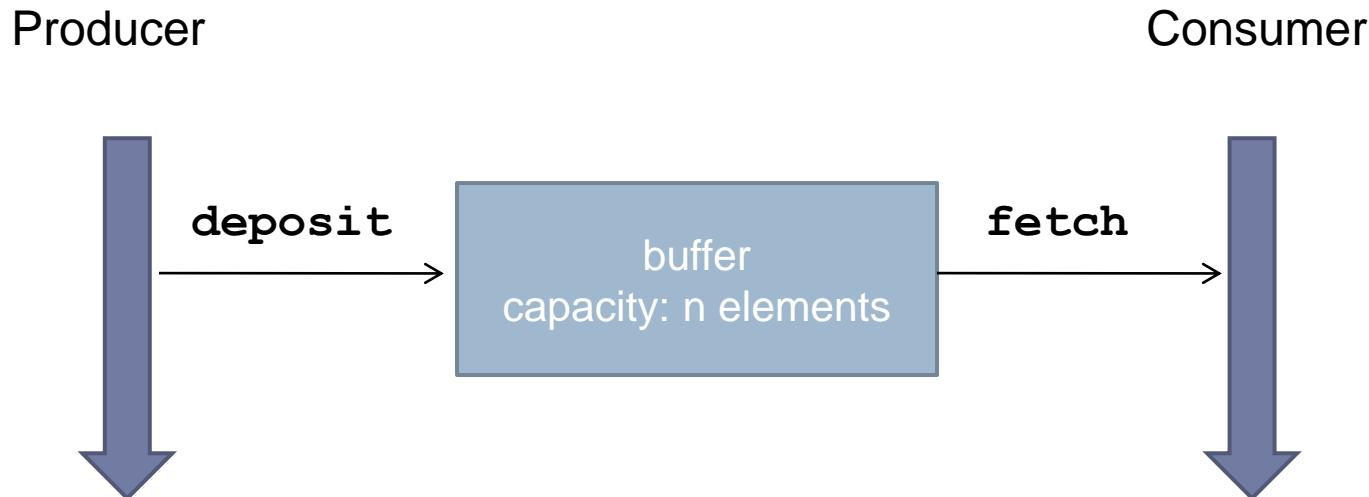
```
▶ //lock implementation with nested monitor lockout problem
▶   public class Lock{
▶     protected MonitorObject monitorObject = new MonitorObject();
▶     protected boolean isLocked = false;
▶
▶     public void lock() throws InterruptedException{
▶       synchronized(this){
▶         while(isLocked){
▶           synchronized(this.monitorObject){
▶             this.monitorObject.wait();
▶           }
▶         }
▶         isLocked = true;
▶       }
▶     }
▶     public void unlock(){
▶       synchronized(this){
▶         this.isLocked = false;
▶         synchronized(this.monitorObject){
▶           this.monitorObject.notify();
▶         }
▶       }
▶     }
▶   }
```

Example of a deadlock in nested calls

```
public static Object cacheLock = new Object();
public static Object tableLock = new Object();
...
public void oneMethod() {
    synchronized (cacheLock) {
        synchronized (tableLock) {
            doSomething();
        }
    }
}
public void anotherMethod() {
    synchronized (tableLock) {
        synchronized (cacheLock) {
            doSomethingElse();
        }
    }
}
```

Example: Synchronize access to a restricted buffer between the producer and the consumer

- ▶ This is an example that we have already dealt with. It is the principle of conditional synchronization when accessing a common buffer which is spatially limited.
- ▶ Suppose we have a buffer with the capacity to store n elements.



Implementation of a monitor for using limited buffer

Implement the buffer for communication between the producer and the consumer as a monitor.

► Monitor Variables:

- ▶ `buf [n]`, `front`, `rare`, `count` (counts how many buffer slots we have currently occupied)
- ▶ `cond not_full` - stops loading from the producer's buffer until the buffer is ready to write (not free)
- ▶ `cond not_empty` - stops the consumer to read the data from the buffer until new data is in the buffer.

► Monitor Procedures:

- ▶ `deposit (typeT)` - inserts the data into the buffer (producer)
- ▶ `fetch (* typeT)` - Write data from the buffer to local variable type `typeT` (consumer)

```
monitor Bounded_Buffer {  
    typeT buf[n];      # an array of some type T  
    int front = 0,      # index of first full slot  
        rear = 0;      # index of first empty slot  
        count = 0;      # number of full slots  
    ## rear == (front + count) % n  
    cond not_full,    # signaled when count < n  
        not_empty;    # signaled when count > 0  
  
    procedure deposit(typeT data) {  
        while (count == n) wait(not_full);  
        buf[rear] = data; rear = (rear+1) % n; count++;  
        signal(not_empty);  
    }  
  
    procedure fetch(typeT &result) {  
        while (count == 0) wait(not_empty);  
        result = buf[front]; front = (front+1) % n; count--;  
        signal(not_full);  
    }  
}
```

Usage:

- Producer: `Bounded_Buffer.deposit(a[i]);`
- Consumer: `Bounded_Buffer.fetch(&b[i]);`

Implementation of a monitor for using limited buffer in JAVA

```
public class Bounded_Buffer {  
    private Object[] buf;  
    private int n, count = 0, front = 0, rear = 0;  
  
    public Bounded_Buffer(int n) {  
        this.n = n;  
        buf = new Object[n];  
    }  
  
    public synchronized void deposit(Object obj) {  
        while (count == n)  
            try { wait(); }  
            catch (InterruptedException e) { }  
        buf[rear] = obj; rear = (rear + 1) % n; count++;  
        notifyAll();  
    }  
  
    public synchronized Object fetch() {  
        while (count == 0)  
            try { wait(); }  
            catch (InterruptedException e) { }  
        Object obj = buf[front]; buf[front] = null;  
        front = (front + 1) % n; count--;  
        notifyAll();  
        return obj;  
    }  
}
```

Methods deposit and fetch are type synchronized, which means they are being executed in mutual exclusion.

Methods wait, notifyAll are executed on conditional variable that is implicitly associated with the monitor.

Implementation of the Producer/Consumer problem

```
class Producer extends Thread {  
  
    private BoundedBuffer buf;  
    private int[] a;  
  
    public Producer (int[] a, BoundedBuffer buf) {  
        this.buf = buf;  
        this.a = a;  
    }  
  
    public void run () {  
        for (int i=0; i < a.length; i++)  
            buf.deposit(a[i]);  
    }  
}  
  
class Consumer extends Thread {  
    private BoundedBuffer buf;  
    public int[] b;  
  
    public Consumer(int[] b, BoundedBuffer buf) {  
        this.buf = buf;  
        this.b = b;  
    }  
  
    public void run () {  
        for (int i = 0; i < b.length; i++)  
            b[i] = buf.fetch();  
    }  
}
```

Implementation of the Producer/Consumer problem

```
class Exchange {  
  
    public static void main(String args[]) {  
  
        int[] a;  
        int[] b;  
  
        int n = Integer.parseInt(args[0], 10);  
  
        a = new int[n];  
        b = new int[n];  
  
        System.out.println("Tabela a:");  
        for(int i = 0; i < n; i++) {  
            a[i] = (int) (Math.random () * 100);  
            System.out.print (a[i]+" ");  
        }  
  
        System.out.println(); System.out.flush();  
  
        BoundedBuffer buf = new BoundedBuffer (5);  
        Producer p = new Producer (a, buf);  
        Consumer c = new Consumer (b, buf);  
  
        p.start ();  
        c.start ();  
    }  
}  
  
try {  
    c.join ();  
} catch (InterruptedException e) { };  
  
b = c.b;  
System.out.println("Tabela b:");  
for(int i = 0; i < n; i++)  
    System.out.print (b[i]+" ");  
System.out.println(); System.out.flush();  
}
```

Example: reading and writing the database

- ▶ A program for reading and writing data from/to a database.
- ▶ The database is only one integer variable.
- ▶ First version:
 - ▶ sequential program, alternately read and write data.

[CaseStudies\Java\Java Osnove\ReadersWriters\rw.seq\rw.seq.java](#)

Example: reading and writing the database

- ▶ Parallel program for reading and writing data from to database.
- ▶ The database is only one integer variable.
- ▶ Second version:
 - ▶ a parallel program for writing and reading,
 - ▶ poor version, we do not prevent mutual exclusion,
 - ▶ just for example, how to make a multithreaded program in Java.

[CaseStudies\Java\Java Osnove\ReadersWriters\rw.par\rw.par.java](#)

Example: reading and writing the database

- ▶ Parallel program for reading and writing data from to database.
- ▶ The database is only one integer variable.
- ▶ Third version:
 - ▶ a parallel program for writing and reading,
 - ▶ we perform mutual exclusion by implementing a new class RWexclusive from the RWbasic class by declaring the read and write methods synchronized.
 - ▶ We also adequately correct the Reader, Writer and Main classes to use RWexclusive instead of RWbasic
 - ▶ this is not the right version because all operations, i.e. writing and reading, are carried out by mutual exclusion.

[CaseStudies\Java\Java Osnove\ReadersWriters\rw.excl\rw.excl.java](#)

Example: reading and writing the database

- ▶ Fourth - final - version:
 - ▶ the read deal,
 - ▶ simultaneous reading of data from the database and exclusive write of data into the database can be performed,
 - ▶ a new class is extended from `RWbasic ReadersWriters` and the code is corrected in the `Reader`, `Writer`, and `Main` classes in order to use the new class.
- ▶ In the `ReadersWriters` class:
 - ▶ we undo the exclusive implementation of the read method,
 - ▶ we add two new methods `startRead` and `endRead`, we use them before and after the data reading operation,
 - ▶ `startRead` method increases variable nr, which tells how many readers read the data,
 - ▶ `EndRead` method reduces variable nr when nr == 0, it informs potential data writers (if any)
 - ▶ `startRead`, `endRead`, and write are synchronized, which means that their execution is excluded.

[CaseStudies\Java\Java Osnove\ReadersWriters\rw.real\rw.real.java](#)

Programming III

Parallel and distributed programming

Distributed programming with message passing

Janez Žibert, Jernej Vičič UP FAMNIT

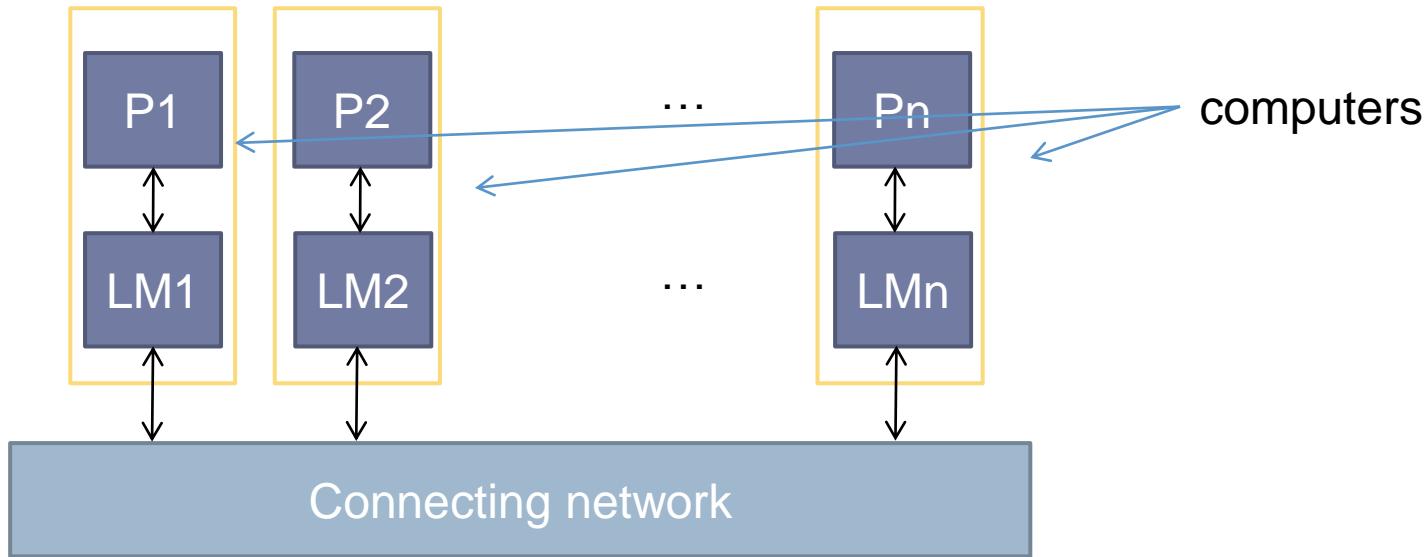
Overview (chapters 7.1 – 7.5)

- ▶ Fundamentals of distributed programming
- ▶ Communication by sending messages
 - ▶ channels for communication and messages
- ▶ Asynchronous communication with messages
 - ▶ communication between the producer and consumer – filters
 - ▶ communication between the server and the client/clients
 - ▶ communication among interacting peers
- ▶ Synchronous communication with messages
- ▶ Comparison between asynchronous and synchronous communication

Distributed programming

- ▶ A distributed program is a program that performs its work in a distributed way between multiple computers or computer systems.
 - ▶ Processes do not share shared memory, we say that the memory is distributed (distributed memory).
 - ▶ Distributed processes communicate with each other via communication channels.
 - ▶ Processes interact with each other by exchanging messages that are transmitted via communication channels.
- ▶ Sending messages enables the exchange of information and data between processes:
 - ▶ the exchange of messages takes place through communication channels shared by distributed processes among themselves (instead of common variables for the shared memory model)
 - ▶ because processes do not share other data, mutual exclusion of implementation is ensured.
- ▶ Communication channels:
 - ▶ Provide one or two-way information flow.
 - ▶ Communication between distributed program processes takes place by receiving and sending messages.
 - ▶ Communication can be asynchronous or synchronous.

Computer architecture with distributed memory



- ▶ Multiple computers connected to the connection network.
- ▶ Each computer represents one node in such a network:
 - ▶ a computer can comprise one or more processor units
 - ▶ Computers do not share shared memory
 - ▶ communication is carried out via the connection network (communication channel abstraction):
 - ▶ networks are different and scalable,
 - ▶ a large number of possible connections between network nodes,
 - ▶ allows a large number of concurrent (and also independent) transactions between nodes in the network.

Mechanisms of communication and the environment for distributed implementation of programs

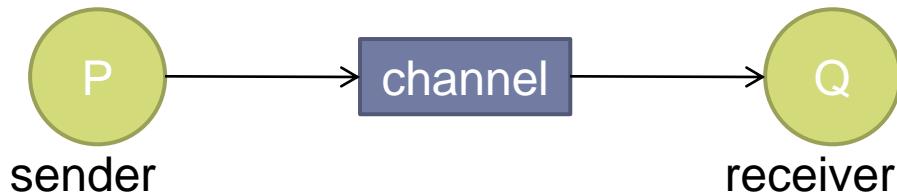
- ▶ Communication mechanisms:
 - ▶ with messages
 - ▶ asynchronous communication with the exchange of messages,
 - ▶ synchronous communication with the exchange of messages,
 - ▶ with calls to perform remote operations
 - ▶ RPC (remote procedure call) in a rendezvous,
 - ▶ RMI (remote method invocation).
- ▶ Distributed programming environment:
 - ▶ Brekeley Sockets API (BSD socket API)
 - ▶ standard for most programming languages for inter-network communication, the basis for communication for distributed programming
 - ▶ RPC API, Java socket API, CORBA, Java RMI
 - ▶ Microsoft .NET
 - ▶ PVM (parallel virtual machine)
 - ▶ MPI (message passing interface)
 - ▶ ...

Forms of distributed distributed applications

- ▶ Processes in distributed computing applications are HW processes (no shared variables) that are run on different computers (processors).
 - ▶ The beginning of the execution of distributed processes is different.
 - ▶ Every process can produce a new process.
 - ▶ A process can perform various tasks.
- ▶ 1. distributed application model: the same program for different data (SPMD):
 - ▶ One program is used to perform several identical tasks on different data.
 - ▶ To split tasks (data) between processes we need to:
 - ▶ known number of processes that will perform a task,
 - ▶ know the size of the data for processing,
 - ▶ define the proper sharing of data between tasks.
- ▶ 2. distributed application model: different programs for different data (MPMD):
 - ▶ different programs are used to perform various tasks (processes) on different data.
 - ▶ Example: server / client

Communication with message exchange

- ▶ Communication between processes is carried out through common communication channels through the exchange of messages:
 - ▶ two basic types of operations: send - send message and receive - receive message
 - ▶ When sending/receiving a message, it means transferring data from the memory of the process that transmits the message to the memory of the process that receives the message.



- ▶ Communication with messages requires synchronization - the message can not be received if it is not already sent.
- ▶ The communication channel can be:
 - ▶ shared buffer in the case of a single-processor or multiprocessor architecture with shared memory,
 - ▶ network connection in the case of distributed distributed memory computer architecture.

Two ways to exchange messages

- ▶ **Synchronous communication with message exchange -** send and receive operation stops the execution of processes until they are done:
 - ▶ The send operation stops the operation of the process – the sender of the message, until the process - receiver of the message - accepts the message.
 - ▶ The receive operation stops the process of the message receiver until the message enters the receiver.
- ▶ **Asynchronous communication with message exchange:**
 - ▶ The send operation does not stop the operation of the process - the transmitter of the message (the process continues its work), and the receive operation stops the operation of the process - the receiver of the message until it receives the message.
 - ▶ In this case, the channel is implemented as an unlimited (very large) FIFO type in which messages are loaded until they reach the receivers.

Basic forms of communication and collaboration between processes in distributed programming

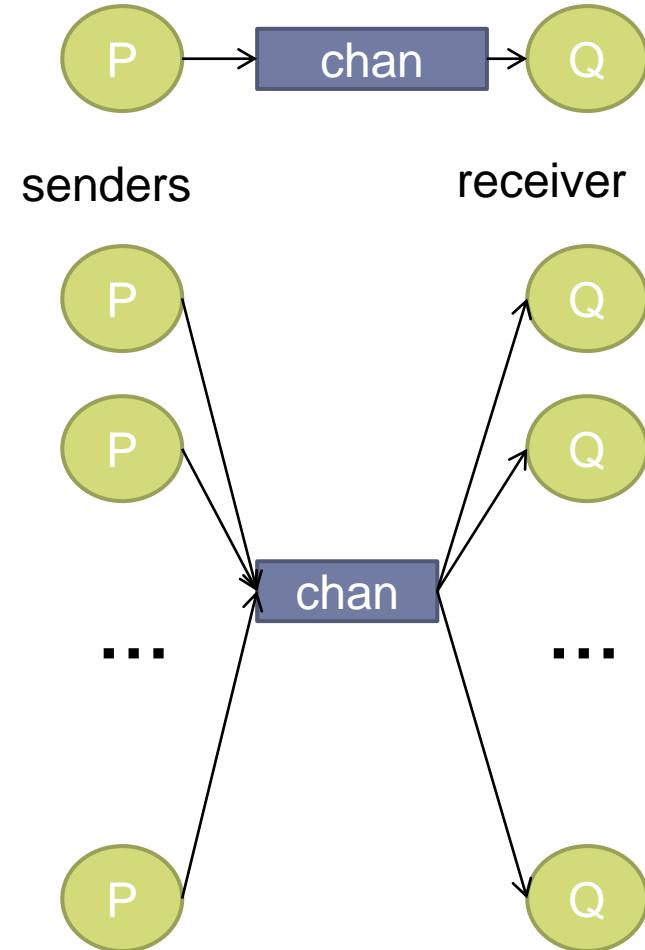
- ▶ **Producer - Consumer (filters) - One-way communication:**
 - ▶ The output of the producer is the consumer's input - the output is the function of the input data - filter
 - ▶ communication: the consumer with the receive operation is waiting for the send operation of the manufacturer
- ▶ **Client / server - two way communication, one process more important than othersthe server responds to client requests,**
 - ▶ communication: the client sends a request message, the server waits for the request message in the receive operation, and then responds to the request with the send operation,
 - ▶ It is a request/response communication.
- ▶ **Interaction of equivalent - two-way communication, all processes are equivalent this is the interaction between equivalent processes.**
 - ▶ Usually, one process initiates an interaction (sends a message), all processes must be ready to receive messages.

Asynchronous communication with exchange of messages

- ▶ **Communication**
 - ▶ Let's assume that we have an unlimited space for storing messages and data outside of the local memory for processes that communicate.
 - ▶ The processes share a common communication channel:
 - ▶ channel is the abstraction of actual physical communication links,
 - ▶ channel is a one-way connection between the transmitter and the receiver,
 - ▶ we can imagine it as a buffer with an unlimited space for storing data in messages.
- ▶ **Basic "atomic" communication operations:**
 - ▶ **send**
 - ▶ sending data (message) from local memory to a common communication channel shared by the process - sender and process - receiver,
 - ▶ when executing the send operation, the process does not stop.
 - ▶ **receive**
 - ▶ receives data (message) from the communication channel and stores it in the local memory,
 - ▶ the receive operation stops the operation of the process until it receives the entire message (all data in the message).

Communication channel

- ▶ The communication channel represents a connection for transmitting messages during the process - sender and process - receiver. We can present a channel as a FIFO queue of messages that have been sent but have not yet been accepted.
- ▶ Operations:
 - ▶ **send** – Attaches the message to the channel queue in the last place
 - ▶ **receive** – Take and delete a message from the first place of the channel queue



Channel declaration `chan`

- ▶ The channel is common to all the processes that use it.
- ▶ Channel Declaration:
 - ▶ `chan ch(type1 field1, ..., typen fieldn);`
 - ▶ list `typei fieldi` represents the data structure from which a message is transmitted over a channel.
 - ▶ names of the variables in `fieldi` can be omitted:
`chan ch(type1, ..., typen);`
- ▶ Channel table:
`chan ch[n] (type1, ..., typem);`
 - ▶ a table of n channels is defined, each transmits messages of the same data structure.
- ▶ Examples:
 - ▶ `chan input(char);`
 - ▶ `chan disk_access(int cylinder, int block, int count, char* buffer);`
 - ▶ `chan result[n](int);`

Operations over channels

- ▶ **send** – sends a message to the channel
`send ch(expr1 , ... , exprn) ;`
 - ▶ the operation does not stop the execution of the process that performs this operation
 - ▶ Operation:
 - values are calculated in `expr1 , ... , exprn`
 - the calculated values are compiled in a message that is the same structure as the channel that transmits messages,
 - the message is pinned to the last place in the channel queue ch

- ▶ **receive** – receives the message from the channel iz kanala
`receive ch(var1 , ... , varn) ;`
 - ▶ the operation stops the execution of the process that performs this operation,
 - ▶ the process is stopped while the message is in the channel
 - ▶ Operation:
 - ▶ the message from the beginning of the channel queue ch is prepared for transmission;
 - ▶ transmission begins;
 - ▶ the data from the message is stored in the variable `var1 , ... , varn`

An example of using channels and operations

```
chan input(char), output(char [MAXLINE]);  
process Char_to_Line {  
    char line[MAXLINE]; int i = 0;  
    while (true) {  
        receive input(line[i]);  
        while (line[i] != CR and i < MAXLINE) {  
            # line[0:i-1] contains the last i input characters  
            i = i+1;  
            receive input(line[i]);  
        }  
        line[i] = EOL;  
        send output(line);  
        i = 0;  
    }  
}
```

The program receives characters from one channel, concatenates them into a string and sends them to the output channel - filter.

Operations over channels

- ▶ Operations **send** and **receive** are atomic:
 - ▶ **send** is analogous **v**(channel as a semaphore)
 - ▶ **receive** is analogous **P**(channel as a semaphore)
- ▶ The sending and receiving of messages is based on the FIFO principle - messages are received in the same order as they were sent.
- ▶ Additional operation:
bool empty (ch) ;
 - ▶ function returns **True**, if the channel ch does not have any messages, else returns **False**
 - ▶ when used it may happen a Race Condition:
 - ▶ eg. the function returns **False** (that is, the channel contains messages), but during this time the message is already submitted to the receivers and the message queue in the channel becomes empty. Attention!

Client/server principle

- ▶ Client is an active process:
 - ▶ Sends messages - requests to the server process through the request channel.
 - ▶ It waits for the server to complete the request, and to receive the result sent by the server back.
 - ▶ The response of the server is transmitted through the response channel.
- ▶ Server is a process that responds to customer requests:
 - ▶ Waiting for requests from customers coming through the request channel.
 - ▶ Processed requests processed and produced response.
 - ▶ The response is sent by the response channel that is shares with the client.
- ▶ In general, we can say that the server response at the client's request is a function of the server's request and status.



Client and server implementation

- ▶ Each client request is mapped to the operation - the sequence of commands for executing this request.
- ▶ The server process is running in one thread. Operations are performed sequentially according to the type of operation that is in the request. Therefore, one channel for request.
- ▶ There can be more customers, so there are also many channels to answer. The server at one time only serves one client.

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
chan request(int clientID, op_kind, arg_type);
chan reply[n] (res_type);
process Server {
    int clientID; op_kind kind; arg_type args;
    res_type results; declarations of other variables;
    initialization code;
    while (true) {      ## loop invariant MI
        receive request(clientID, kind, args);
        if (kind == op1)
            { body of op1; }
        ...
        else if (kind == opn)
            { body of opn; }
        send reply[clientID] (results);
    }
}
process Client[i = 0 to n-1] {
    arg_type myargs; result_type myresults;
    place value arguments in myargs;
    send request(i, opj, myargs);      # "call" opj
    receive reply[i] (myresults);        # wait for reply
}
```

Example of using a client/server: allocation of shared resources

- ▶ Allocation of shared resources:
 - ▶ clients request a share of the common resource (memory, ...);
 - ▶ they use it;
 - ▶ when they are finished, the resource is returned to the resource manager (server);
 - ▶ Basic version: Clients can request and release a shared resource individually (only one thread of the process)
 - ▶ common resources are managed in a certain data structure and accessed by insert/remove operations
- ▶ Operation:
 - ▶ according to the client/server principle
 - ▶ request/answer
 - ▶ two types of operations: **ACQUIRE/RELEASE**
 - ▶ mapped into operations **remove/insert**, that access shared resource
 - ▶ **pending queue**: all shared resource requirements are stored that can not be immediately executed.

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitid);
chan reply[n] (int unitID);

process Allocator {
    int avail = MAXUNITS; set units = initial values;
    queue pending; # initially empty
    int clientID, unitID; op_kind kind;
    declarations of other local variables;
    while (true) {
        receive request(clientID, kind, unitID);
        if (kind == ACQUIRE) {
            if (avail > 0) { # honor request now
                avail--; remove(units, unitID);
                send reply[clientID](unitID);
            } else # remember request
                insert(pending, clientID);
        } else { # kind == RELEASE
            if empty(pending) { # return unitID to units
                avail++; insert(units, unitid);
            } else { # allocate unitID to a waiting client
                remove(pending, clientID);
                send reply[clientID](unitID);
            }
        }
    }
}
```

Example of using a client/server: allocation of shared resources - client

```
process Client[i = 0 to n-1] {
    int unitID;
    send request(i, ACQUIRE, 0)      # "call" request
    receive reply[i](unitID);
    # use resource unitID, then release it
    send request(i, RELEASE, unitID);
    ...
}
```

- ▶ Client
 - ▶ send a message asking for some free units of the shared resource
 - wait to receive these units,
 - ▶ send a message in which it releases units of a common resource – no need to wait.
- ▶ Each client has
 - ▶ one answer channel,
 - ▶ one channel for the request.

Second example of using the client/server principle: file server

- ▶ Example of a **multithreaded server** and an example of **communication in a session**.
- ▶ The file server can be part of a distributed file system:
 - ▶ allows client access to files that are on the server's disk (they can be distributed over multiple disks of the server system).
- ▶ Client requests submitted to the server:
 - ▶ OPEN - the client wants to access the file - after the request is approved, a communication session is opened,
 - ▶ READ/WRITE - the file is being read or written,
 - ▶ CLOSE - the file is closed - the communication session ends
- ▶ The server is multithreaded:
 - ▶ each process processes the requirements of one client session.
 - ▶ the session ends when the server accepts the CLOSE request from the client.

Processes and channels of a file server

- ▶ **n** server processes
 - ▶ `process File_Server[n]`
 - ▶ The server can serve n clients and access n files simultaneously.
 - ▶ Request channels:
 - ▶ One channel for each request **OPEN**:
 - ▶ `chan open(fname, clientID)`
 - ▶ n channels for **READ/WRITE/CLOSE** requests:
 - ▶ `chan access[n] (kind, args)`
 - ▶ Channel with answers:
 - ▶ m channels to respond to **OPEN** requests (one per client)
 - ▶ `chan open_reply[m] (serverID)`
 - ▶ m result channels after accessing files (one per client)
 - ▶ `chan access_reply[m] (results)`

Interaction: file server - client

- ▶ Communication session:
 - ▶ the session opens with the client request after accessing the file:
 - ▶ Client:

```
send open(fname, myid);
receive open_reply[myid] (serverID);
```
 - ▶ Server - process **myid** serves the client with **clientID**:

```
receive open(fname, clientID);
open the file;
if (successful)
    send open_reply[clientID] (myid);
```
 - ▶ Session: sharing requests for accessing data from files and sending access results:
 - ▶ request/response interaction
 - ▶ one client/one server process with identification numbers **clientID** and **serverID**

File server implementation

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n] (int kind, types of other arguments);
chan open_reply[m] (int serverID); # server id or error
chan access_reply[m] (types of results); # data, error, ...

process File_Server[i = 0 to n-1] {
    string fname; int clientID;
    kind k; variables for other arguments;
    bool more = false;
    variables for local buffer, cache, etc.;

    while (true) {
        receive open(fname, clientID);
        open file fname; if successful then:
        send open_reply[clientID] (i); more = true;
        while (more) {
            receive access[i] (k, other arguments);
            if (k == READ)
                process read request;
            else if (k == WRITE)
                process write request;
            else # k == CLOSE
                { close the file; more = false; }
            send access_reply[clientID] (results);
        }
    }
}
```

File client implementation

```
process Client[j = 0 to m-1] {
    int serverID; declarations of other variables;
    send open("foo", j);      # open file "foo"
    receive open_reply[j] (serverID); # get back server id
    # use file then close it by executing the following
    send access [serverID] (access arguments);
    receive access_reply[j] (results);
    ...
}
```

File server improvements

- ▶ **File server enhancements:**
 - ▶ in our case, we can serve a fixed number of clients
 - ▶ alternative: server implementation to serve as many clients as there are (number of processes is not top limited)
 - ▶ complex server implementation
 - ▶ alternative: nr. of file servers is equal to the nr. of disks on the server:
 - ▶ complex execution of the server or clients.
- ▶ **Different approach: NFS (Sun Network File System)**
 - ▶ accessing files is done with remote procedure calls,
 - ▶ similar to normal access to files on a local drive, except that this happens on a remote disk:
 - ▶ the request to access the file is basically required to obtain a file descriptor (file handle in NFS) and the status of the file (write/read),
 - ▶ at each access these (additional) data is transmitted in the remote procedure calls arguments - additional data transfer, but simpler execution of the server and clients
 - ▶ in the case of sudden system interruptions:
 - ▶ server crashes - the client attempts to send requests until he receives a response
 - ▶ client crashes – server does not need to do anything.

Communication between equivalent processes

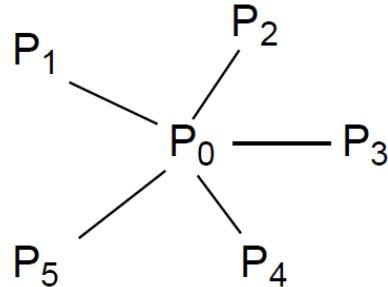
- ▶ Two-way communication between equivalent processes:
 - ▶ communication among servers,
 - ▶ communication among clients,
 - ▶ P2P communication.
- ▶ Applications that use this principle:
 - ▶ Distributed Composite Table (DHT),
 - ▶ file sharing, torrents,
 - ▶ P2P services (DNS),
 - ▶ P2P on grids,
 - ▶ net games.

Example: exchange of values of variables

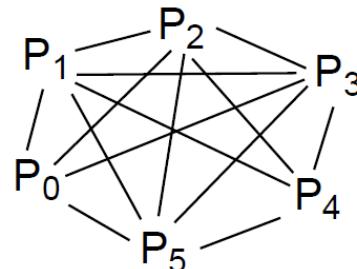
- ▶ A typical problem of interaction between the equivalent.
- ▶ We have n processes
 - ▶ each includes a number that needs to exchange with all other processes.
- ▶ Task:
 - ▶ each process must determine the minimum and maximum value between the numbers that the processes share with each other.
- ▶ Implementation:
 - ▶ by sending messages,
 - ▶ weighing between efficiency and number of exchanged messages.

Possible solutions

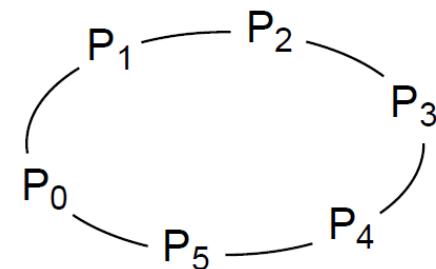
- ▶ **Centralized solution:** The coordinator collects all messages in the center, makes a comparison, and returns the highest and lowest value to all processes.
- ▶ **Symmetric Solution:** Each process connects to all others and thus obtains information about the largest and the smallest number.
- ▶ **Circular Solution:** Processes connect to a circle and information travels round the circle until each process receives information about the largest and the smallest number.



Centralized solution



Symmetric Solution



Circular Solution

Centralized solution

- ▶ One process - coordinator - collects all values, makes a comparison, and sends the solution to all processes back.
- ▶ Total **2(n-1)** messages and n channels:
 - ▶ if we can implement a "broadcast" message we only have n different messages.

```
chan values(int), results[n](int smallest, int largest);
process P[0] {    # coordinator process
    int v;    # assume v has been initialized
    int new, smallest = v, largest = v;  # initial state
    # gather values and save the smallest and largest
    for [i = 1 to n-1] {
        receive values(new);
        if (new < smallest)
            smallest = new;
        if (new > largest)
            largest = new;
    }
    # send the results to the other processes
    for [i = 1 to n-1]
        send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
    int v;    # assume v has been initialized
    int smallest, largest;
    send values(v);
    receive results[i](smallest, largest);
}
```

Simmetric solution

- ▶ Every process
 - ▶ sends its value to all others,
 - ▶ collects all values from other processes,
 - ▶ makes a comparison and determines the maximum and minimum value.
- ▶ Total:
 - ▶ $n(n-1)$ messages and n channels
 - ▶ if we have "broadcast" messages, then we have n messages

```
chan values[n] (int);
process P[i = 0 to n-1] {
    int v;      # assume v has been initialized
    int new, smallest = v, largest = v;  # initial state
    # send my value to the other processes
    for [j = 0 to n-1 st j != i]
        send values[j] (v);
    # gather values and save the smallest and largest
    for [j = 1 to n-1] {
        receive values[i] (new);
        if (new < smallest)
            smallest = new;
        if (new > largest)
            largest = new;
    }
}
```

Circular solution

- ▶ Two rounds of message exchange:
 - ▶ round 1: determine the minimum and the maximum as they would be solved in the pipe; the solution is obtained only when we reach the last process in the pipe;
 - ▶ round 2: propagate the solution to all processes.
- ▶ Work in process $P[i]$ in the first round:
 - ▶ take two values (current min and max) from its predecessor - process $P[i-1]$;
 - ▶ compare these two values with its value and correct min or max accordingly;
 - ▶ send the corrected min and max to its successor - process $P[(i + 1) \bmod n]$;
 - ▶ it is necessary to avoid the deadlock (remember the problem of the five philosophers) - $P[0]$ first send, and then receive data.
- ▶ Work in process $P[i]$ in the second round:
 - ▶ take two values (final min and max) from its predecessor - process $P[i-1]$;
 - ▶ both values are stored and forwarded to the successor -process $P[(i + 1) \bmod n]$;

Circular solution

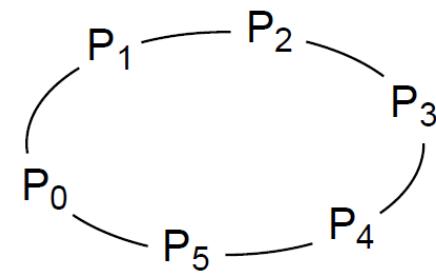
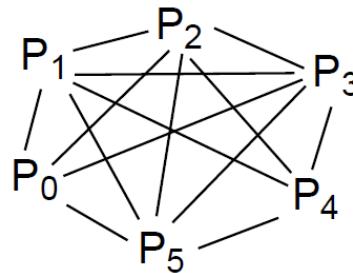
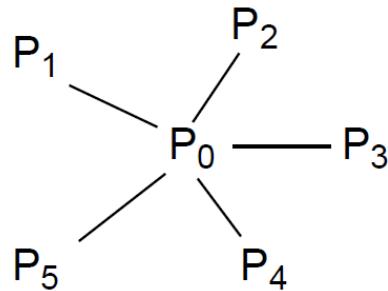
- ▶ Total:
 - ▶ $2n-1$ messages and n channels
 - ▶ "broadcast" messages are not possible

```
chan values[n] (int smallest, int largest);
process P[0] {  # initiates the exchanges
    int v;    # assume v has been initialized
    int smallest = v, largest = v;  # initial state
    # send v to next process, P[1]
    send values[1] (smallest, largest);
    # get global smallest and largest from P[n-1] and
    #   pass them on to P[1]
    receive values[0] (smallest, largest);
    send values[1] (smallest, largest);
}

process P[i = 1 to n-1] {
    int v;    # assume v has been initialized
    int smallest, largest;
    # receive smallest and largest so far, then update
    #   them by comparing their values to v
    receive values[i] (smallest, largest)
    if (v < smallest)
        smallest = v;
    if (v > largest)
        largest = v;
    # send the result to the next processes, then wait
    #   to get the global result
    send values[(i+1) mod n] (smallest, largest);
    receive values[i] (smallest, largest);<
    if (i < n-1)
        send values[i+1] (smallest, largest);
}

}
```

Let's repeat: the interaction between equivalent processes



Centralized solution

Symmetric Solution

Circular Solution

- ▶ Communication between equivalent processes - symmetric solution - $O(n^2)$ messages, processes are easy to program (each process has all the information)
- ▶ A star or circle solution - $O(n)$ messages
- ▶ A circular solution may be ineffective:
 - ▶ we need to carry out two rounds of message exchange in order to get a common global solution.

Synchronous communication with message exchange

- ▶ Both operations: **send** and **receive** stop the execution of the processes until the transmission of the message is carried out:
 - ▶ the send operation stops the process that sends the message until this message is received with the receive operation of the receiver process.
 - ▶ operation:
`synch_send ch(expr1 , ..., exprn);`
 - ▶ operation **receive** stops the receiver process until it receives the entire message from the sender process.
- ▶ Messaging channels can be implemented without buffers for storing message data.
- ▶ Synchronous communication was proposed with the programming language CSP (1978, Tony Hoare).

Restrictions on synchronous communication

- ▶ Example: communication between producers and consumers with synchronous message exchange:
 - ▶ for example, the producer and consumer reach the point of communication at different moments.
 - ▶ Then they have to wait. This increases the total time execution.
Limited parallel execution.

```
channel values(int);

process Producer {
    int data[n];
    for [i = 0 to n-1] {
        do some computation;
        synch_send values(data[i]);
    }
}

process Consumer {
    int results[n];
    for [i = 0 to n-1] {
        receive values(results[i]);
        do some computation;
    }
}
```

- ▶ Another example: synchronous communication in the case of a client and a server
 - ▶ unnecessary suspension of execution:
 - ▶ when the client announces that it no longer needs a shared resource, there is no reason to wait for the server to receive this message.
 - ▶ similarly: when a client sends a write request to an external source (file, graphic display, ...), which is controlled by the server, it is not necessary to wait for the server to receive the request, but only once in the future that request is executed.

Another disadvantage: the possibility of deadlock

- ▶ Example:
 - ▶ two processes are exchanging their values.
- ▶ Deadlock:
 - ▶ Both processes are stopped on the **synch_send** operation
 - ▶ solution: in one process, we replace the order of the send and **receive** operation.
 - ▶ This must be taken care at each synchronous communication: when one process is executed by **synch_send**, the other process must perform the **receive** operation.
 - ▶ In asynchronous communication, we can leave such a symmetrical solution.

```
channel in1(int), in2(int);  
  
process P1 {  
    int value1 = 1, value2;  
    synch_send in2(value1);  
    receive in1(value2);  
}  
  
process P2 {  
    int value1, value2 = 2;  
    synch_send in1(value2);  
    receive in2(value1);  
}
```

Comparison of synchronous and asynchronous communication

- ▶ Advantages and disadvantages of asynchronous messaging:
 - ▶ simpler programming
 - ▶ allows simultaneous execution of distributed processes,
 - ▶ less deadlock situations
 - ▶ But: it needs an unlimited space for storing messages or data outside of the memory of a particular process
 - ▶ hardware and software are therefore more demanding
 - ▶ the amount of space required for storing messages in channels is dependent on communication and is changing, so the allocation of the channel allocation space needs to be organized dynamically.
- ▶ Advantages and disadvantages of synchronous messaging:
 - ▶ Channels can be implemented without message storage interfaces:
 - ▶ the data is transmitted directly from the transmitter's memory to the receiver's memory, no "buffering" and copying
 - ▶ But:
 - ▶ limited concurrency of implementation
 - ▶ greater risk of deadlock situations; it is always necessary to ensure the correct order of transmission and reception of messages.

Programming III

Parallel and distributed programming

Introduction to MPI.

Janez Žibert, Jernej Vičič

UP FAMNIT

Overview (chapter 7.8, additional literature)

- ▶ **Introduction**
 - ▶ Concepts of programming in MPI: processes, communicators, ...
- ▶ **MPI Library:**
 - ▶ functions, types of variables and commands, ...
- ▶ **Basic functions of MPI**
- ▶ **Send and receive command**
 - ▶ synchronous communication, asynchronous communication
- ▶ **Group communication operations:**
 - ▶ barrier, broadcast, gather/scatter, reduce, ...
- ▶ **Examples:**
 - ▶ A simple program for exchanging values between distributed processes
 - ▶ calculation of decimals of the number of PI
 - ▶ computing farm
- ▶ **References:**
 - ▶ book: Chapter 7.8
 - ▶ www: A User's Guideknjiga: poglavje 7.8
 - ▶ www: [A User's Guide to MPI](#)
 - ▶ MPI standard: [MPI: A Message-Passing Interface Standard](#)

MPI: Message Passing Interface

- ▶ The MPI library is designed to work with distributed processes in multiprocessor or multi-computing architectures (cluster, ...):
 - ▶ communication between processes with message exchange - API,
 - ▶ implementation of the API and communication in the library,
 - ▶ possible implementation of distributed programs with the MPI library in the programming languages C and Fortran
- ▶ Purpose:
 - ▶ enable the development of programs for parallel and distributed computing,
 - ▶ enable the implementation of distributed programs on various distributed computer systems and various communication protocols,
 - ▶ MPI is the standard for distributed computing.
- ▶ Implementation of MPI:
 - ▶ MPICH: general implementation, enabled on various computer platforms - takes advantage of individual computer platforms (the most widespread)
 - ▶ UNIX, WindowsIt
 - ▶ works: distributed and shared memory, various topologies of computers, various connections between computers
 - ▶ MPICH2: new version of MPICH,
 - ▶ LAM: intended for use on computer systems that communicate via TCP/IP
 - ▶ There are others.

MPI library

- ▶ The MPI library includes more than 130 features that allow communication between processes:
 - ▶ by exchanging messages (one-to-one, group)
 - ▶ grouping processes and different topologies.
- ▶ On the other hand, only 6 basic functions can be used to effectively program distributed programming:
 - ▶ `MPI_Init`, `MPI_Finalize`,
`MPI_Comm_size`, `MPI_Comm_rank`,
`MPI_Send`, `MPI_Recv`
 - ▶ Or these 6:
 - ▶ `MPI_Init`, `MPI_Finalize`,
`MPI_Comm_size`, `MPI_Comm_rank`,
`MPI_Bcast`, `MPI_Reduce`
- ▶ All functions and data structures have the **`MPI_`** prefix.

Variable types in MPI and link to C types

Data types in MPI	Data types in C
<code>MPI_CHAR</code>	<code>signed char</code>
<code>MPI_SHORT</code>	<code>signed short int</code>
<code>MPI_INT</code>	<code>signed int</code>
<code>MPI_LONG</code>	<code>signed long int</code>
<code>MPI_UNSIGNED_CHAR</code>	<code>unsigned char</code>
<code>MPI_UNSIGNED_SHORT</code>	<code>unsigned short int</code>
<code>MPI_UNSIGNED</code>	<code>unsigned int</code>
<code>MPI_UNSIGNED_LONG</code>	<code>unsigned long int</code>
<code>MPI_FLOAT</code>	<code>float</code>
<code>MPI_DOUBLE</code>	<code>double</code>
<code>MPI_LONG_DOUBLE</code>	<code>long double</code>
<code>MPI_BYTE</code>	<code>8 binary digits</code>
<code>MPI_PACKED</code>	<code>data packed or unpacked with MPI_Pack() / MPI_Unpack</code>

MPI processes

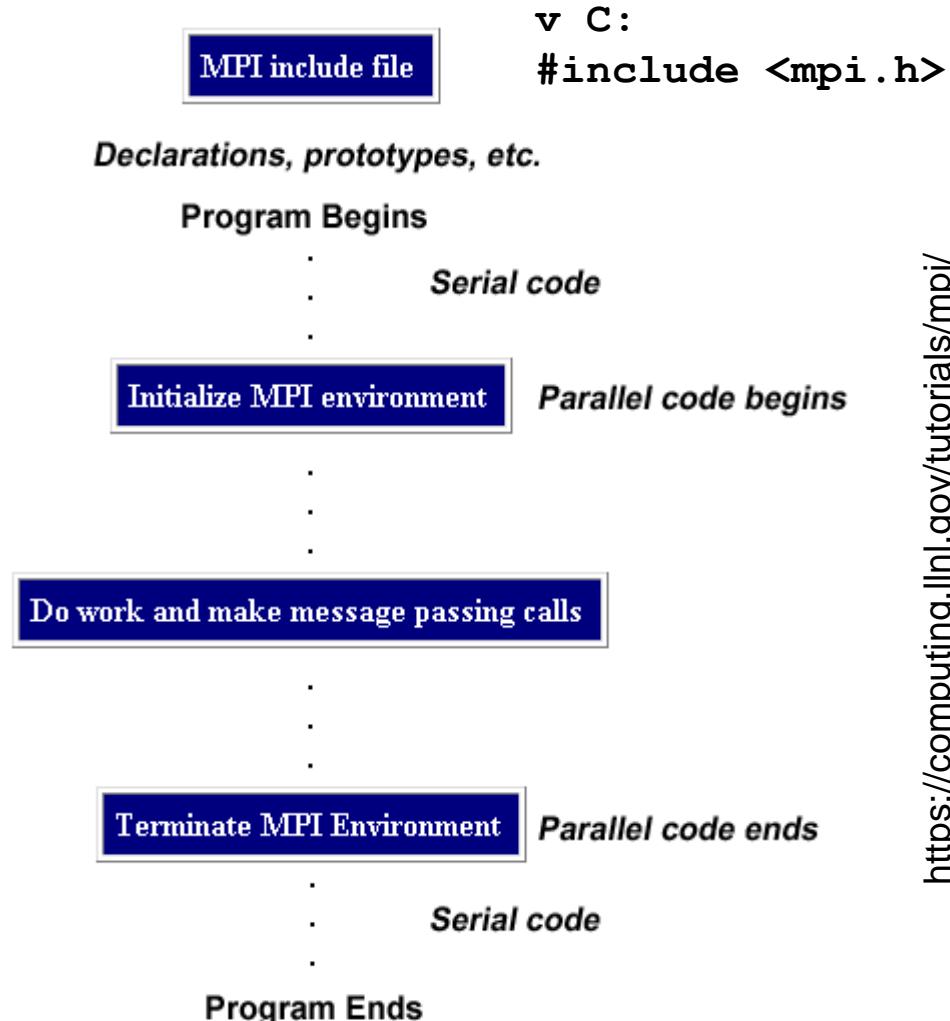
- ▶ **MPI process:**
 - ▶ Our process - the implementation of a program part on one processor (computer).
 - ▶ Each process has its own ID, its rank, and belongs to a group of processes.
 - ▶ Processes can communicate only with processes from the same group, which are handled by one communicator.
 - ▶ Communication within the communicator (communicator context) is done through the exchange of messages.
 - ▶ The process can send messages synchronously or asynchronously.
 - ▶ Each process receives the messages that are sent to it.
 - ▶ Processes communicate with each other using the send/receive functions, which are implemented either to stop the processes or not.
 - ▶ The processes can communicate with each other through group communication (within the communicator) with broadcast, collect/scatter, barrier operations.

MPI processes: communicator, process group, process rank

- ▶ **Communicator:**
 - ▶ Data structure - an object which defines a group of processes that can communicate among themselves.
 - ▶ Communication within a communicator can be limited to individual sub-groups of processes:
 - ▶ the subgroup of processes solves a task by exchanging messages that take place through communication, which is predefined in a communication context within a communicator.
- ▶ **Process group:**
 - ▶ A group of processes that are included in the communicator.
 - ▶ The processes within the group can communicate directly with each other or group (one with all, ...).
 - ▶ Every communication takes place only within the group.
 - ▶ Each process has its rank within a group of processes.
- ▶ **Process Rank:**
 - ▶ Each process has its own ID number, which determines the rank of the process within a process group (in a communicator).

MPI program

- ▶ Structure of an MPI program:
- ▶ One program for all processes.
- ▶ Number of simultaneous processes is predefined at starttime.
 - ▶ it is best to program as many processes as we expect when the program is started (one process per processor).



MPI application

- ▶ Install one of the MPI implementations, eg. MPICH, MPICH2, openmpi
- ▶ Write one program for the application:
 - ▶ one program for all processes
- ▶ Compile the program:
 - ▶ `mpicc -o myprog myprog.c`
 - ▶ Use Makefile for bigger projects:
 - example: `/usr/share/doc/openmpi-doc/examples/Makefile`
- ▶ Start the program:
 - ▶ `mpirun -np 2 myprog`
 - ▶ With option `-help` dobimo vse opcije `mpirun`.

MPI application

- ▶ Start the program on multiple computers:
- ▶ prepare the environment
 - ▶ each computer has the program in the same place,
 - ▶ NFS
 - ▶ ssh keys for accessing computers,
- ▶ `mpirun -np 2 myprog`
- ▶ `mpirun -np 5 -hosts client,localhost myprog`
- ▶ `mpirun -np 5 -hosts 172.16.153.20, 172.16.153.21, 172.16.153.22 myprog`

Basic MPI functions

- ▶ **int MPI_Init(int *argc, char **argv[])**
 - ▶ Run MPI. Initialize everything you need for MPI processes.
- ▶ **int MPI_Finalize()**
 - ▶ Stop MPI. Cleanup.
- ▶ **int MPI_Comm_size(MPI_Comm comm, int *size)**
 - ▶ Determine and return the number of processes in the communicator group:
 - ▶ **comm** – communicator, **MPI_COMM_WORLD** (all processes in the group)
 - ▶ **size** – number of processes in group
- ▶ **int MPI_Comm_rank(MPI_Comm comm, int *rank)**
 - ▶ Defines and returns the rank of the process that is performed in the communicator group:
 - ▶ **comm** – communicator, **MPI_COMM_WORLD**
 - ▶ **rank** – rank of the process in communicator group (value between 0 and **size-1**)
- ▶ **int MPI_Abort(MPI_Comm comm)**
 - ▶ Interrupts all processes in the communicator group.

Example: “Hello world!”

```
10 #include "mpi.h"
11 #include <stdio.h>
12
13 int main (int argc, char *argv[]) {
14
15     int numtasks, rank, rc;
16
17     rc = MPI_Init(&argc,&argv);
18     if (rc != MPI_SUCCESS) {
19         printf ("Error starting MPI program. Terminating.\n");
20         MPI_Abort(MPI_COMM_WORLD, rc);
21     }
22
23     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
24     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
25     printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);
26
27     //***** do some work *****/
28
29     MPI_Finalize();
30 }
31 }
```

Output:

```
Number of tasks= 10 My rank= 1
Number of tasks= 10 My rank= 0
Number of tasks= 10 My rank= 2
Number of tasks= 10 My rank= 3
Number of tasks= 10 My rank= 4
Number of tasks= 10 My rank= 5
Number of tasks= 10 My rank= 7
Number of tasks= 10 My rank= 8
Number of tasks= 10 My rank= 9
Number of tasks= 10 My rank= 6
```

Compile and startup:

```
user@linux:~$ mpicc hello2.c -o hello2
user@linux:~$ mpirun -np 10 hello2
```

Operation send

- ▶

```
int MPI_Send(void *buf, int count, MPI_datatype dt,
             int dest, int tag, MPI_Comm comm)
```
- ▶ Sends a tag **tag** to the dest process, which is defined within the communicator group:
 - ▶ **buf** message data
 - ▶ **count** count of data in message
 - ▶ **dt** data type in message
 - ▶ **dest** process rank
 - ▶ **tag** message tag
 - ▶ **comm** communicator
- ▶ The function returns an integer value that indicates the status of the command execution, most often **MPI_SUCCESS**.
- ▶ **datatype** can be any elementary type of variable (previous slides) or composite structure from these variables
- ▶ **MPI_Type_contiguous** **MPI_Type_vector**, **MPI_Type_indexed**, **MPI_Type_struct**

Operation receive

- ▶

```
int MPI_Recv(void *buf, int count, MPI_datatype dt,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```
- ▶ Accepts a message with **tag** from process **source**, that is defined in the communicator group:
 - ▶ **buf** buffer (data structure) to store message data
 - ▶ **count** data count
 - ▶ **dt** message data type
 - ▶ **source** process rank
 - ▶ **tag** message tag
 - ▶ **comm** communicator
 - ▶ **status** returned operation status
- ▶ When accepting a message, we can also use:
 - ▶ for message tag: **MPI_ANY_TAG**: accept every message, for process rank,
 - ▶ **MPI_ANY_SOURCE**: accept message from any process that sends a message.

Example

- ▶ Have a look at `exchange_mpi.c`!

Reception of messages

- ▶ If we use tags for mass reception of messages **MPI_ANY_TAG** or to receive messages from multiple sources **MPI_ANY_SOURCE**, we can check the state of reception of messages with structure **MPI_Status**, which has three components: **MPI_SOURCE**, **MPI_TAG**, **MPI_ERROR**:

```
MPI_Status status;  
MPI_Recv(..., &status);  
int tag_received = status.MPI_TAG;  
int rank_of_source = status.MPI_SOURCE;  
MPI_Get_count(&status, datatype, &count);
```

- ▶ **MPI_Get_count** returns the number of messages of data type **datatype** have been received.

Example: exchange array values between four processes

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {

    int numtasks, rank, source=0, dest, tag=1, i;
    float a[SIZE][SIZE] =
        {1.0, 2.0, 3.0, 4.0,
         5.0, 6.0, 7.0, 8.0,
         9.0, 10.0, 11.0, 12.0,
         13.0, 14.0, 15.0, 16.0};

    float b[SIZE];

    MPI_Status stat;
    MPI_Datatype rowtype;

    MPI_Init (&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &numtasks);

    MPI_Type_contiguous (SIZE, MPI_FLOAT, &rowtype);
    MPI_Type_commit (&rowtype);
```

```
if (numtasks == SIZE) {
    if (rank == 0) {
        for (i=0; i<numtasks; i++)
            MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
    }

    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
           rank,b[0],b[1],b[2],b[3]);
}
else
    printf("Must specify %d processors. Terminating.\n",SIZE);
MPI_Type_free(&rowtype);
MPI_Finalize();
```

MPI_Type_contiguous

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0
-----	------	------	------

1 element of
rowtype

Example

- ▶ **See `exchange2.c`!**

Example: message exchange (always succeeds)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
    myvalue = 14;
    MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Process 0 sending %d to process 1.\n", myvalue);

    MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf("Process 0 received a value %d from process 1.\n", othervalue);
}

if (rank == 1) {

    MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    printf("Process 1 received a value %d from process 0.\n", othervalue);

    myvalue = 25;
    MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    printf("Process 1 sending %d to process 0.\n", myvalue);

}
```

- ▶ The program will finish, even if there is no buffer for exchange of messages, which means that communication is performed synchronously.

▶ Execution:

- ▶ when send from process 0 is sends data, receive from process 1 receives,
- ▶ both processes can continue their work:
- ▶ When send from process 1 sent the data, receive in process 0 receives. The process finishes.

Example: message exchange (that never finishes)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

    MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf("Process 0 received a value %d from process 1.\n", othervalue);

    myvalue = 14;
    MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Process 0 sending %d to process 1.\n", myvalue);

}

if (rank == 1) {

    MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    printf("Process 1 received a value %d from process 0.\n", othervalue);

    myvalue = 25;
    MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    printf("Process 1 sending %d to process 0.\n", myvalue);

}
```

- ▶ The program will never finish, because both processes want to receive messages first, and hence stop the execution until the message is received.

Example: message exchange (that conditionally succeeds)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

    myvalue = 14;
    MPI_Send(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    printf("Process 0 sending %d to process 1.\n", myvalue);

    MPI_Recv(&othervalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
    printf("Process 0 received a value %d from process 1.\n", othervalue);
}

if (rank == 1) {

    myvalue = 25;
    MPI_Send(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    printf("Process 1 sending %d to process 0.\n", myvalue);

    MPI_Recv(&othervalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    printf("Process 1 received a value %d from process 0.\n", othervalue);
}
```

- ▶ The program will be executed to the end if it is possible to store at least one message in the buffer during communication (depending on the implementation of MPI)
= if the send operations are asynchronous.
- ▶ In this case, the buffer can 1 **int** in size.

Message exchange without stopping processes

- ▶ `int MPI_Isend(void *buf, int count, MPI_datatype dt, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
 - ▶ Sends a message with `tag` to the `dest` process, which is defined within a communicator group. The process can resume work immediately, as the storage space in the local memory is reserved for storing a message that begins to transmit via a communication channel. The message status information is stored in the data structure `&request`.
- ▶ `int MPI_Irecv(void *buf, int count, MPI_datatype dt, int source, int tag, MPI_Comm comm, MPI_Request *request)`
 - ▶ Receives a message tagged `tag` from the process `source`, which is defined within the communicator group. The process can resume work immediately, as it essentially only prepares the memory space for the (later) reception of the message. The message status information is stored in the data structure `&request`.

Message exchange control

- ▶ `int MPI_Test(MPI_Request *request, int *flag,
MPI_Status *status)`
 - ▶ Test if the operation without stopping (Isend, Ireceive) has actually been carried out. If it was executed (the message was completely submitted or received): `flag = true`, otherwise `false`. Command ends immediately.
- ▶ `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
 - ▶ The command waits until the operation linked to `request` is completely executed (finished).
- ▶ `MPI_Iprobe(source, tag, comm, flag, status)`
 - ▶ The process that executes this command checks to see if a message with the `tag` of the process `source` is prepared, the flag is set to `true` if the message is waiting for it, otherwise `false`.
- ▶ `MPI_Probe(source, tag, comm, status)`
 - ▶ The process that executes this command checks to see if a message with the `tag` of the process `source` is prepared, the flag is set to `true` if the message is waiting for it, otherwise `false`. – Same as above BUT the command returns only when message is delivered, otherwise the process that executes this command stops.

Example: send and receive messages without stopping the execution of processes

- ▶ The process is stopped on the send and receive operations. The value for process 1 is obtained only when:
 - ▶ the process 0 has sent the whole message (MPI_Wait),
 - ▶ the process 1 has received the whole message (MPI_Wait).

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {

    myvalue = 14;
    MPI_Isend(&myvalue, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &request);

    //***** do some work *****
    sleep(rand()%10);

    MPI_Wait(&request, &status);
    printf("Process 0 send value %d to process 1.\n", myvalue);

}

if (rank == 1) {

    MPI_Irecv(&myvalue, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);

    //***** do some work *****
    sleep(rand()%10);

    MPI_Wait(&request, &status);
    printf("Process 1 received value %d from process 0.\n", myvalue);

}

MPI_Finalize();
```

Operations for group communication

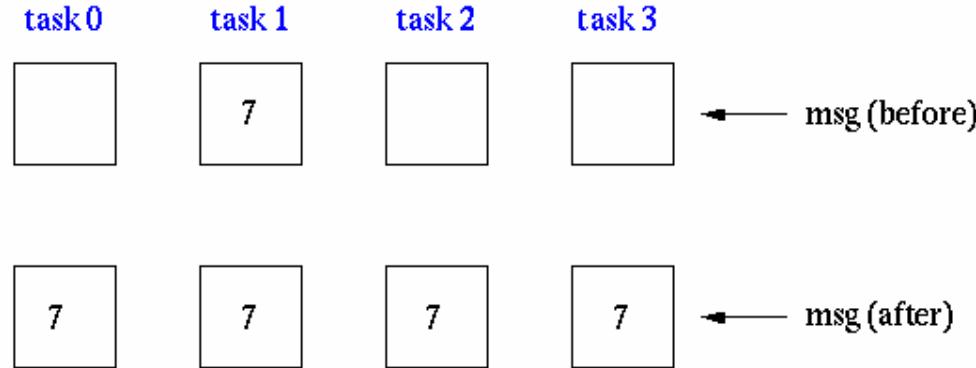
- ▶ Operacije za skupinsko komunikacijo omogočajo interakcijo enega procesa z vsemi ostalimi procesi v isti skupini: *Group communication operations allow one process to interact with all other processes in the same group:*
 - ▶ **`MPI_Barrier()`**
 - ▶ Signals the arrival to barrier point. The function stops the operation of the process until all processes in the group reach this point.
 - ▶ **`MPI_Bcast()`**
 - ▶ Send a copy of the message to all the processes within the group
 - ▶ **`MPI_Scatter()`**
 - ▶ Send messages from the array `a` to other processes: send a message to the process `i`.
 - ▶ **`MPI_Gather()`**
 - ▶ Collect messages from all processes and store them in an array: message from process `i` save to `a[i]`.
 - ▶ **`MPI_Reduce()`**
 - ▶ Collect messages from all processes in the same group and merge them into one message according to the operation specified in the command. Possible operations: `MPI_SUM`, `MPI_MAX`, `MPI LAND` (logicalAND), `MPI_BOR` (binarni OR), ...
 - ▶ **`MPI_Allreduce()`**
 - ▶ Same as `MPI_Reduce`, each process in the group receives the gathered message.

Broadcast

MPI_Bcast

Broadcasts a message to all other processes of that group

```
count = 1;  
source = 1;           broadcast originates in task 1  
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

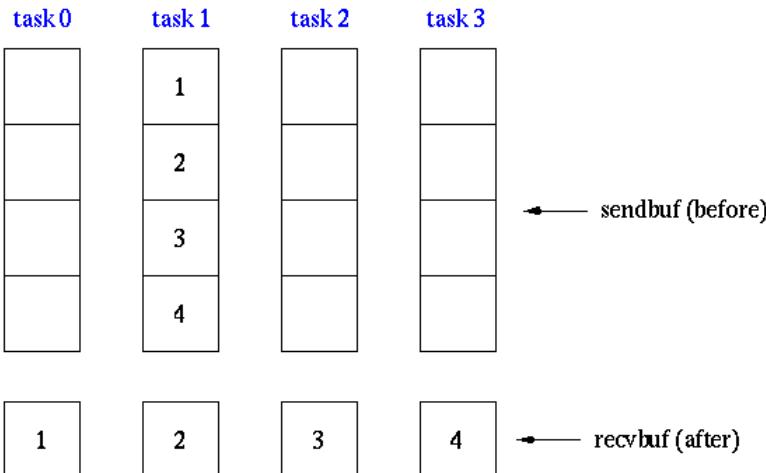


Scatter/Gather

MPI_Scatter

Sends data from one task to all other tasks in a group

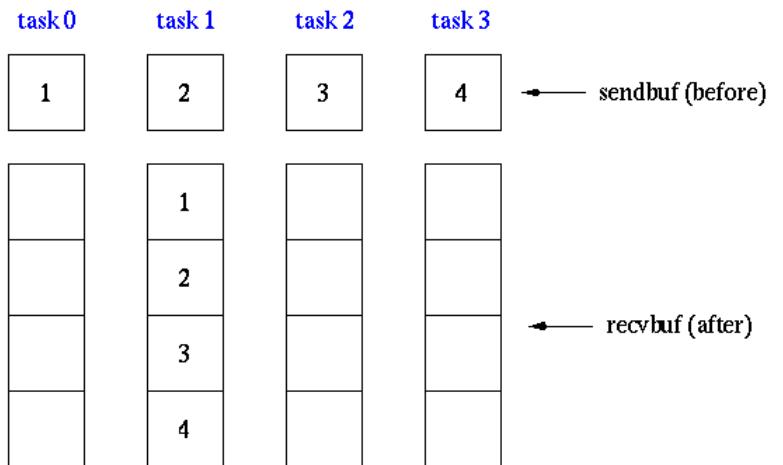
```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           task 1 contains the message to be scattered  
MPI_Scatter(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```



MPI_Gather

Gathers together values from a group of processes

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           messages will be gathered in task 1  
MPI_Gather(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```

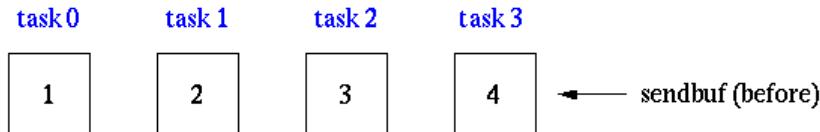


Reduce/Allreduce

MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

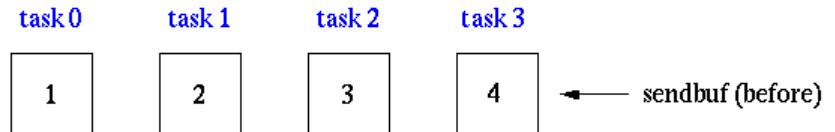
```
count = 1;  
dest = 1;           result will be placed in task 1  
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
           dest, MPI_COMM_WORLD);
```



MPI_Allreduce

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

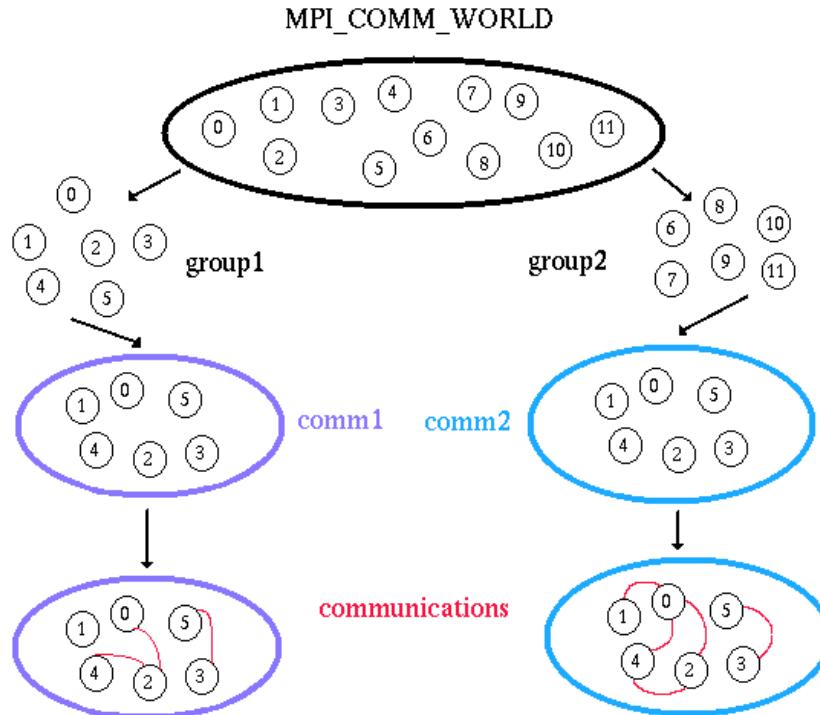
```
count = 1;  
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
              MPI_COMM_WORLD);
```



Examples

- ▶ Usage:
 - ▶ **MPI_Bcast**
 - ▶ **MPI_Gather**
 - ▶ **MPI_Scatter**
 - ▶ **MPI_Reduce**

Groups and communicators



- ▶ Groups of processes that can communicate with one another are grouped in communicators.
 - ▶ Communicators can also be considered as additional identification to messages (not just the tag, but also to which group (communicator) this message belongs).
- ▶ Communicator
MPI_COMM_WORLD groups all processes of an MPI application.
- ▶ MPI has functions for:
 - ▶ to create new communicators, to delete, ...
 - ▶ to work with process ranks within a communicator and work with process groups, ...

Bigger example: Calculation of the PI number

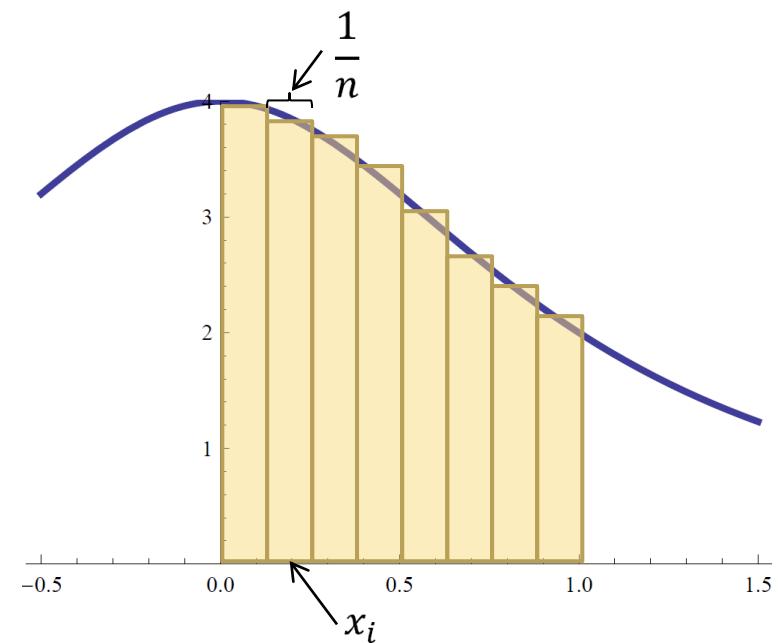
- ▶ The value of definite integral:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

- ▶ Approximation of a definite integral:

$$\frac{1}{n} \sum_{i=1}^n \frac{4}{1+x_i^2} \quad x_i = \frac{1}{n}(i - 0.5)$$

- ▶ Calculate the sum and get an approximation of Pi.
- ▶ n has to be as big as possible to get better accuracy.

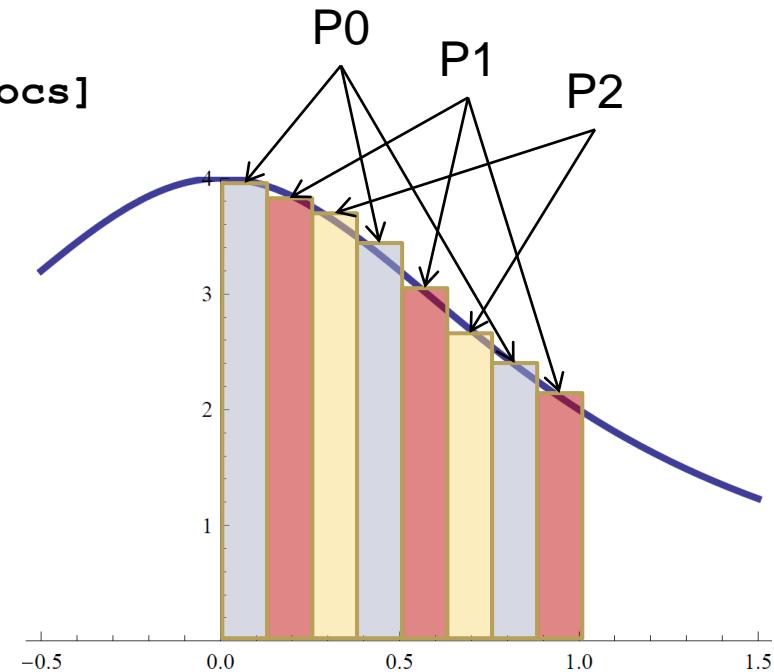


Bigger example: Calculation of the PI number

- ▶ Divide the computation of the sum to all processes:
- ▶ Each process:

```
for[i=rank+1 to n step by numprocs]
```

$$\frac{1}{n} \sum_{i=1}^n \frac{4}{1+x_i^2} \quad x_i = \frac{1}{n}(i - 0.5)$$



- ▶ Then use the function **`MPI_Reduce`** to send to the process 0 all partial sums that are summed together.
- ▶ The final result is the number Pi.

Calculation of the PI number

Each process receives a number of intervals to sum the values.

```
int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
    }
}
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
if (n == 0) break;
```

```
h = 1.0 / (double) n;
```

```
sum = 0.0;
```

```
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
```

```
mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
if (myid == 0)
```

```
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
```

```
}
```

```
MPI_Finalize();
```

All subtotals are transmitted to process 1 and in the meantime already summed up. This is the final value Pi.

Implementation of a computing farm

- ▶ The bag of tasks principle:
 - ▶ one process (coordinator or farmer) shares tasks among workers,
 - ▶ processes workers perform tasks, when a task is completed, they communicate the availability to get a new job,
 - ▶ the number of tasks and task operations are known before the start of the program,
 - ▶ process assignments are dynamic - given the speed and success of an individual worker process.
- ▶ Suppose:
 - ▶ the number of tasks is greater than or equal to the number of processes,
 - ▶ the tasks in our case are merely the sum of integer values, the results are integer values (usually the tasks are more complex and the results are complex data structure). sadece

Implementation of a computing farm

```
#define MAX_TASKS 100
#define NO_MORE_TASKS MAX_TASKS+1

void farmer (int workers);
void worker (int rank);

int main(int argc, char *argv[])
{

    int np, rank;
    time_t t;

    t = time(NULL); // seed the random number
    srand((int) t); // generator from outside

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &np);

    if (rank == 0) {
        farmer(np-1);
    } else {
        worker(rank);
    }

    MPI_Finalize();
}
```

Process no. 0 is the coordinator, all other processes are workers.
The coordinator checks how many workers are available.
Each worker gets its ID, which is just the rank of the worker's process.

Coordinator

Choose a random integer for tasks numbers between 0 and 4, which means, how many work units do workers do.

Send the first n jobs to workers:
Task i is sent to worker i (process $i + 1$),
The job tag is i.

Each further task is forwarded among workers:
a worker that completes a task is assigned a new task.

This is the end (my only friend, the end):
There are no more tasks.
Gather the last results and send another message with tag: **NO_MORE_TASKS**, thus task till will tell workers to stop working.

```
void farmer (int workers)
{
    int i, task[MAX_TASKS], result[MAX_TASKS], temp, tag, who;
    MPI_Status status;

    for (i=0; i<MAX_TASKS; i++) {
        task[i] = rand()%5; // set up some "tasks"
    }

    // Assume at least as many tasks as workers
    for (i=0; i<workers; i++) {
        MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
    }

    while (i<MAX_TASKS) {
        MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

        who = status.MPI_SOURCE;
        tag = status.MPI_TAG;
        result[tag] = temp;
        MPI_Send(&task[i], 1, MPI_INT, who, i, MPI_COMM_WORLD);
        i++;
    }

    for (i=0; i<workers; i++) {
        MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

        who = status.MPI_SOURCE;
        tag = status.MPI_TAG;
        result[tag] = temp;
        MPI_Send(&task[i], 1, MPI_INT, who, NO_MORE_TASKS,
                 MPI_COMM_WORLD);
    }
}
```

Worker

```
void worker (int rank)
{
    int tasksdone = 0;
    int workdone = 0;
    int task, result, tag;
    MPI_Status status;

    MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
             &status);
    tag = status.MPI_TAG;
    while (tag != NO_MORE_TASKS) {
        sleep(task);
        result = rank;
        workdone+=task;
        tasksdone++;
        MPI_Send(&result, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
        MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                 &status);
        tag = status.MPI_TAG;
    }
    printf("Worker %d solved %d tasks totalling %d units of work \n", rank, tasksdone, workdone);
}
```

Worker receives a task from the coordinator process 0. Checks the tag of the message, if it is NO_MORE_TASKS, does nothing, only prints the results.

The worker receives the task from the coordinator - process 0. The task is simulated with a sleep for the number of units. Count how many tasks have been solved and how much work was done. The result of the job is sent to the coordinator (only one integer). Receives a new task from the coordinator. The worker communicates with the coordinator only. Verifies the message tag, if NO_MORE_TASKS, jumps out of the loop.

Programming III

Parallel and distributed programming

Distributed programming, map reduce

Jernej Vičič, Janez Žibert UP FAMNIT

Overview (not in the book)

- ▶ Map-Reduce

Distributed computing

- ▶ **distributed programs** are run on two or more computers,
- ▶ **distributed algorithms** are distributed programs that stop,
- ▶ **distributed systems** are distributed programs running in infinity, usually offering services / tasks

Distributed computing

- ▶ users are distributed,
- ▶ information is distributed,
- ▶ it can be more reliable,
- ▶ it can be faster,
- ▶ it may be cheaper (Cray versus some classrooms).

Distributed computing

- ▶ computers are crashing,
- ▶ network connections are falling,
- ▶ sending data is slow (10Gbit network has 300 micro seconds latency)
- ▶ the 2GHz processor makes 600,000 cycles the throughput is finally large,
- ▶ no global clock (ticks),...

Ozadje MR

- ▶ MR - parallel programming model
 - ▶ Google 2004
 - ▶ It bases on a large number of cheap computers (user - commodity)
 - ▶ Failure is a part of life.
 - ▶ Simple
 - ▶ Scales well

Osnovne komponente

1. MR task scheduling and environment
 - ▶ Running jobs, dealing with moving data, coordination, failures
2. Distributed File System (DFS)
 - ▶ Saving data in a robust way across the network
 - ▶ Moving - delivering data to nodes
 - ▶ Distributed Hash Table (BigTable)
 - ▶ Random access to data that is on the network
 - ▶ **Hadoop** is an open source version that supports 1 and 2
 - ▶ HBase supports 3

Google File System - GFS

- ▶ GFS is a distributed file system,
- ▶ made by Google,
- ▶ it is used by Google,
- ▶ Hadoop is an open source implementation of GFS,
 - ▶ with some limitations.

What is a distributed file system?

- ▶ Common operations on files:
 - ▶ create, read, move, write, find, distributed system,
- ▶ distributed file access,
- ▶ files are stored in a distributed way.

Why is GFS different?

- ▶ Expected high failure of individual nodes,
- ▶ huge files,
- ▶ almost all of the writings are append,
- ▶ reading:
 - ▶ small and random,
 - ▶ large and sequential,
- ▶ we do not need the full POSIX set (symlink, ...),
- ▶ throughput is more important than latency.

HDFS interface

- ▶ Hadoop Distributed File System,

Shell:

```
bin/hadoop dfs -{command}
```

Primer:

```
bin/hadoop dfs -mkdir /foodir
```

```
bin/hadoop -cat /foodir/myfile.txt
```

Java:

```
org.apache.hadoop.dfs.DistributedFilesystem supplies  
create, open, exists, rename, delete, mkdirs,  
listPaths (ls)  
getFileStatus (stat), set & getWorkingDirectory
```

Distributed programming

- ▶ Problems:
 - ▶ communication and coordination
 - ▶ solving computer crashes,
 - ▶ status report,
 - ▶ debug,
 - ▶ optimization,
 - ▶ small chance to reuse the tools.

Typical problem for MapReduce

- ▶ Read large amounts of data.
- ▶ Map: extract important information from individual records.
- ▶ Shuffle and Sort.
- ▶ Reduce: aggregation, filtering, or transforming results.
- ▶ Save the results.

MapReduce

- ▶ Podatki so shranjeni na eni ali več gručah.
- ▶ Datoteke so shranjene v blokih.
- ▶ Velikost blokov je optimizirana za diskovni medpomnilnik (pogosto 64M).
- ▶ Bloki so replicirani po omrežju:
 - ▶ replikacija omogoča fault tolerance
 - ▶ replikacija povečuje verjetnost, da so želeni podatki na istem vozlišču kjer izvira zahteva.
- ▶ Bloki so razdeljeni enakomerno po celotni gruči.

Data is stored on one or more clusters.

Files are stored in blocks.

Block size is optimized for disk buffer (often 64M).

Blocks are replicated over the network:

Replication allows fault tolerance

Replication increases the likelihood that the desired data is on the same node where the request originates.

The blocks are distributed evenly throughout the cluster.

MapReduce - control

- ▶ Tasks and data are centrally controlled:
 - ▶ dashboard for supervision and managing;
 - ▶ Hadoop: this represents a single-point of failure.
- ▶ Possibility to transfer tasks between data centers:
 - ▶ Allows the use of cheap electricity (night tariff);
 - ▶ load-balancing;
 - ▶ disasters, etc.

MapReduce – programming model

- ▶ MR provides a limited version of parallel programming:
 - ▶ coarse-grained;
 - ▶ no interprocess communication;
 - ▶ Communication can be achieved through files.

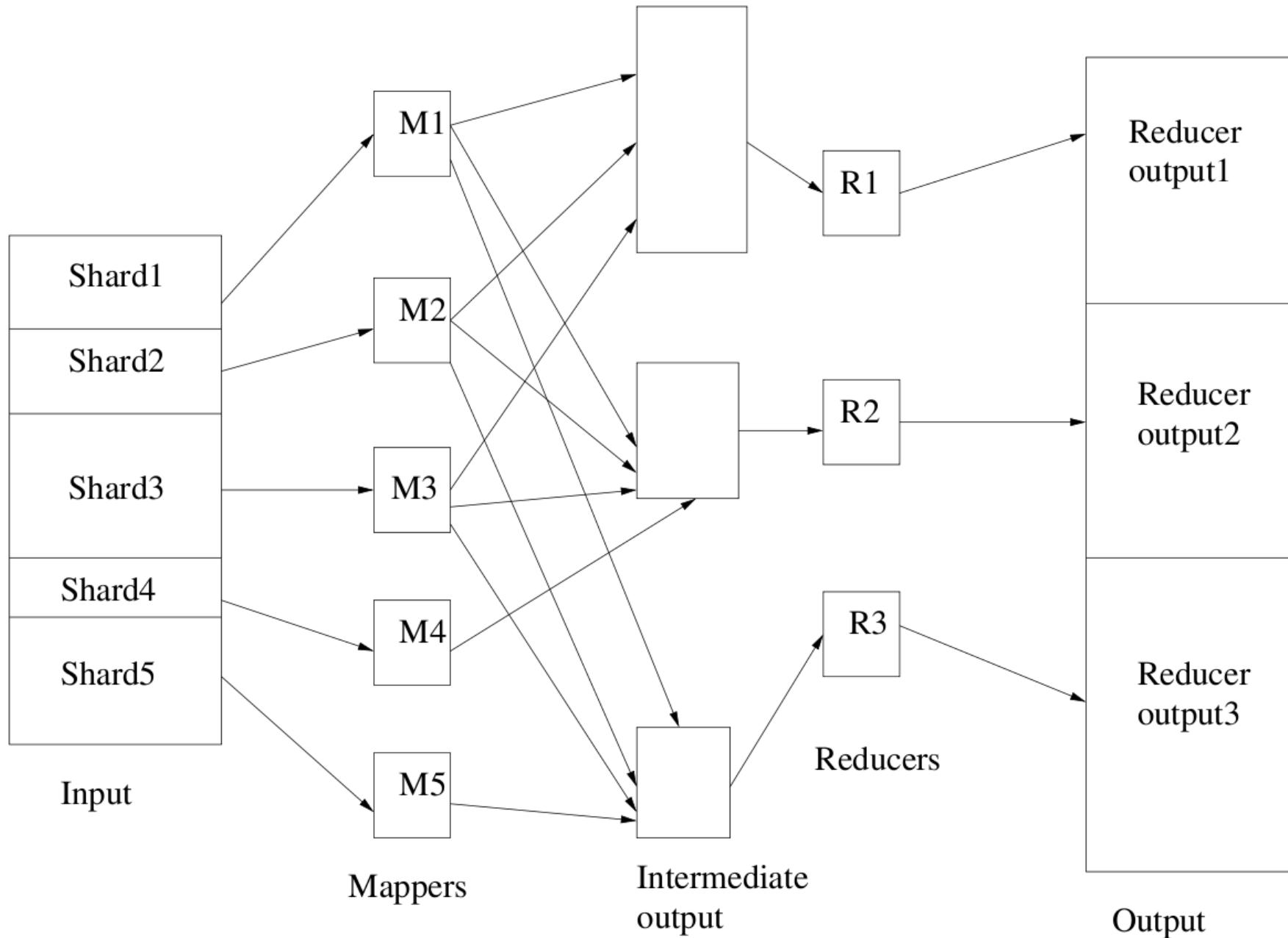
MapReduce – programming model

- ▶ **Mapping:**
 - ▶ input data is split into shards;
 - ▶ Map operation works on each single fragment and “spits” pairs: key-value;
 - ▶ each mapper works in parallel.

The key and the value can represent anything that can be represented by a string - (String).

MapReduce – programming model

- ▶ Reducing:
 - ▶ Each key-value pair is mapped by a hash key.
 - ▶ This pair is sent to the selected reducer.
 - ▶ All keys that are compressed into the same value are sent to the same reducer.
 - ▶ The reducer input is sorted by key.
 - ▶ Sorted input means that related pairs of key values are grouped locally.



MapReduce – programming model

- ▶ Each mapper and reducer is run in parallel.
- ▶ Tasks do not share the environment.
- ▶ Communication between tasks is achieved:
 - ▶ Through external means,
 - ▶ at startup.
- ▶ The number of mappers and reducers cutters is optional PAZI!
- ▶ We can also have 0 reducers.

MapReduce – programming model

- ▶ Tasks read the input sequentially.
- ▶ Sequential reading of the disk is much more effective than random access.
- ▶ Reduction begins after the end of the mapping (individual task, not the whole data).

MapReduce – example - WordCount

- ▶ Count the number of words in the collection of documents!
- ▶ A maper counts the words in each shard.
- ▶ The Reducer collects individual partial sums and gives the final sum.

MapReduce – example - WordCount

- ▶ **Mapper:**
 - ▶ for each shard: produce pairs \langle word, 1 \rangle .
 - ▶ Key is the word.
 - ▶ Value is 1.

MapReduce – example - WordCount

- ▶ Reducer:
 - ▶ Each Reducer will see all the instances of certain words (hashing).
 - ▶ Sequential reading of the input of the reducer gives the partial results of counting the word.
 - ▶ Partial counts can be added to the final count.

MapReduce – example - WordCount

▶ **Input:**

- ▶ the cat
- ▶ the dog

▶ **Output of the Mapper:**

key	value
the	1
cat	1
the	1
dog	1

Reducer 1 input

the, 1

the, 1

dog, 1

Reducer 2 input

cat, 1

Reducer 1 output

the, 2

dog, 1

Reducer 2 output

cat, 1

MapReduce – example- source code

- ▶ WordCount.java

MapReduce – example - start

- ▶ WordCount Hadoop:
 - ▶ use HDFS.
 - ▶ Select the program MR.
 - ▶ Start the job

MapReduce – example start

- ▶ Upload file to HDFS:
 - ▶ `hadoop dfs -mkdir data`
 - ▶ `hadoop dfs -put file.txt data/`
- ▶ check:
 - ▶ `hadoop dfs -ls data/`

MapReduce – example - start

- ▶ Upload files to HDFS:
 - ▶ `hadoop dfs -mkdir data`
 - ▶ `hadoop dfs -put file.text data/`
- ▶ Check:
 - ▶ `hadoop dfs -ls data/`
- ▶ Start:
 - ▶ `hadoop jar hadoop*examples*.jar wordcount data/ data/output`

Programming III

Parallel and distributed programming

GPU

Jernej Vičič

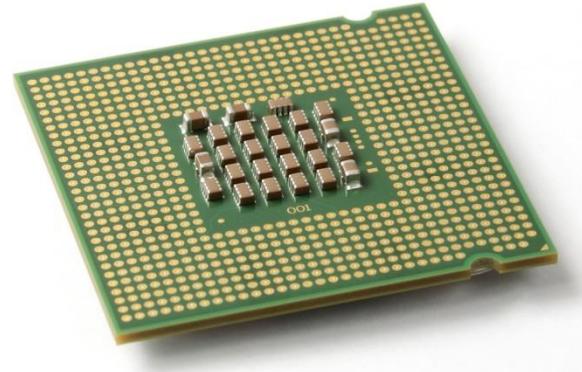
Overview (not in the book)

- ▶ GPU



CPU

- ▶ central processing unit
- ▶ Usually, applications use the CPU for primary calculations
 - ▶ strong, general purpose,
 - ▶ Restricted to Moore's Law,
 - ▶ Proven technology.



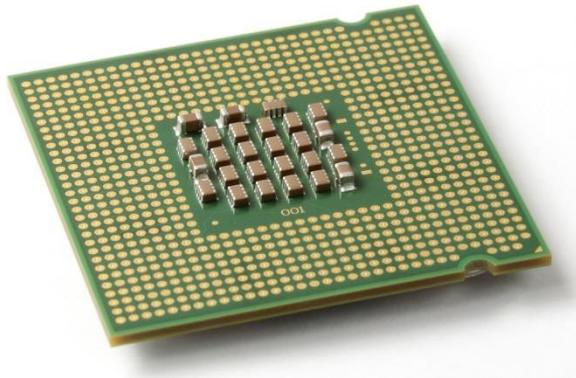
GPU

- ▶ Designed for graphics,
- ▶ For graphics problems: it's **MUCH FASTER** than CPU,
- ▶ What about other problems?



This lecture in 30 seconds

- ▶ For certain problems use:



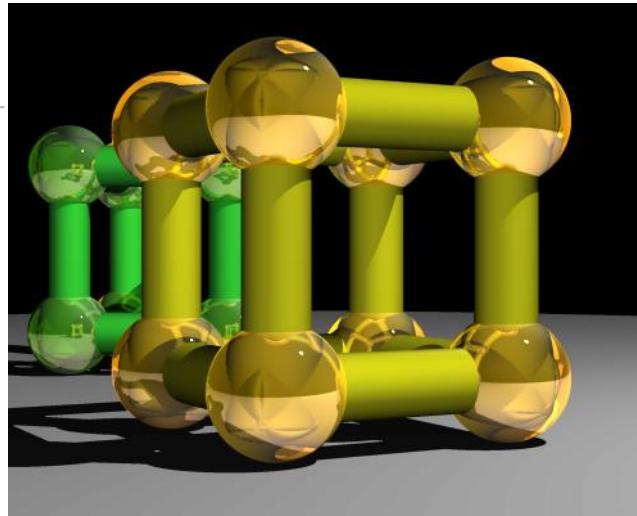
This lecture in 60 seconds

- ▶ GPU: hundreds (thousands) of cores!
against. 2,4,8 (16, 32) CPU cores.
- ▶ Suitable for hingly (massivelly) parallel problems:
Speedup: 10x, 100x+



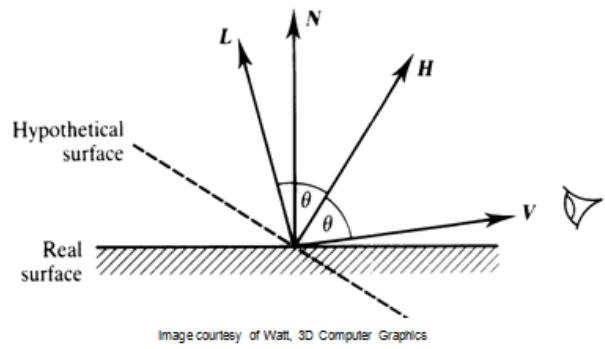
GPU – motivation

A lot of calculations are
“the same”!



Example: Raytracing:

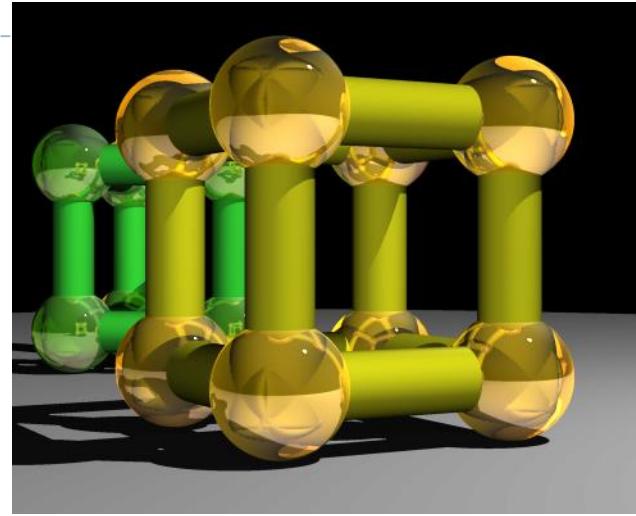
Objective: tracking light beams, calculating object interaction, and creating a realistic image



Watt, 3D Computer Graphics (from
<http://courses.cms.caltech.edu/cs171/>)

GPU – motivation

A lot of calculations are
“the same”!



Example: Raytracing:

Objective: tracking light beams, calculating object interaction, and creating a realistic image

```
for all pixels (i,j):  
    calculate ray point and direction in 3d space  
    if ray intersects object:  
        calculate lighting at closest object  
    store color of (i,j)
```

GPUs – motivation

A lot of calculations are
“the same”!



Example: simple shading:

for all pixels (i, j) :

replace previous color with new color
according to rules



"Example of a Shader" by TheReplay - Taken/shaded with YouFX webcam software, composited next to each other in Photoshop. Licensed under CC-BY-SA 3.0 via Wikipedia http://en.wikipedia.org/wiki/File:Example_of_a_Shader.png#/media/File:Example_of_a_Shader.png

GPUs – motivation

A lot of calculations are

“the same”!

$$T_{\mathbf{v}} \mathbf{p} = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{bmatrix} = \mathbf{p} + \mathbf{v}$$

Transformations (camera, perspective, ...):

for all vertices (x, y, z) in scene:

obtain new vertex $(x', y', z') = T(x, y, z)$



Overview

- ▶ motivation
- ▶ short history
- ▶ “simple problem”
- ▶ “simple rešitev“



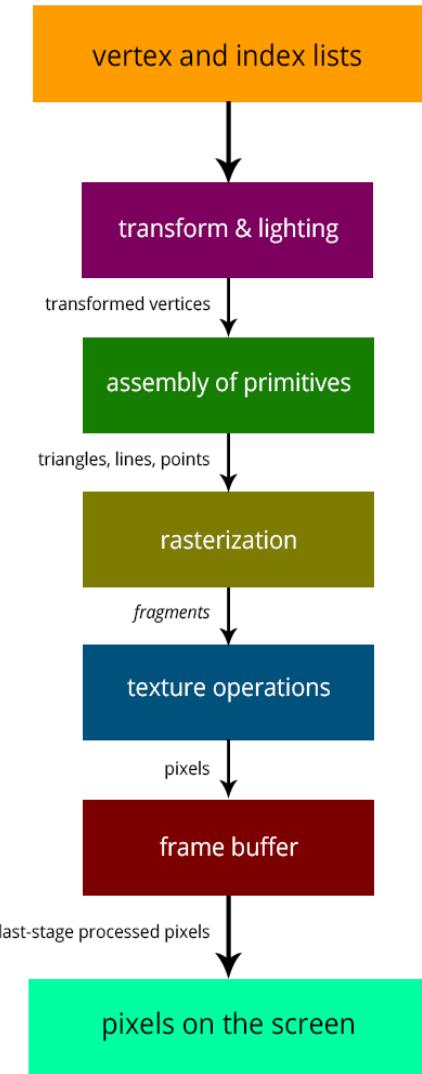
GPU – short history

Pipelines with fixed functionality
predefined functions,
limited functionality.



<http://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on-cms-21469>

Source: Super Mario 64, by Nintendo



GPU – short history

Shaders:

- ▶ Can implement proprietary functions!
- ▶ GLSL (C-like language) – OpenGL Shading Language
- ▶ Can be used for general programming - general-purpose programming!



GPU – short history

- ▶ CUDA (Compute Unified Device Architecture)
 - ▶ General-purpose parallel computing platform for NVIDIA GPUs
- ▶ OpenCL (Open Computing Language)
General heterogenous computing framework

...

Used as C extensions! (and other languages...)



GPU today

- ▶ “General-purpose computing on GPUs” (GPGPU)



Overview

- ▶ motivation
- ▶ short history
- ▶ “simple problem”
- ▶ “simple solution “



Simple problem...

Sum two arrays

$A[] + B[] \rightarrow C[]$

On CPU:

```
float *C = malloc(N * sizeof(float));  
for (int i = 0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Sequential, can we do better?



Simple problem...

On CPU (multi-threaded, pseudocode):

(allocate memory for C)

Create # of threads equal to number of cores on processor
(around 2, 4, perhaps 8)

(Indicate portions of A, B, C to each thread...)

...

In each thread,

For (i from beginning region of thread)

 C[i] <- A[i] + B[i]

 //lots of waiting involved for memory reads, writes, ...

 wait for threads to synchronize...

A few times faster – 2-8x



Simple problem...

How many threads? How Speed Increases With Increasing The System (scaling)?

Context switch:

- Expensive on CPU
- Cheap on GPU



Simple problem...

GPU:

(allocate memory for A, B, C on GPU)

Create the “kernel” – each thread will perform one (or a few) additions

Specify the following kernel operation:

For (all i's assigned to this thread)

$C[i] \leftarrow A[i] + B[i]$

- Start **~20000 (!)** threads
- Wait of the threads to synchronize (do their job)...



GPU: advantages

- ▶ Parallelism / LOTS of cores
- ▶ Low price for context switching.
 - ▶ We can afford a lot of threads!



Overview

- ▶ motivation
- ▶ short history
- ▶ “simple problem”
- ▶ “simple solution”



GPU Computing: step by step

- ▶ Prepare input on computer - host (CPU-accessible memory)
- ▶ Allocate memory for input on GPU
- ▶ Allocate memory for output on host
- ▶ Allocate memory for output on GPU
- ▶ Copy input from host to GPU
- ▶ Start GPU kernel
- ▶ Copy output from GPU to host

(Copy can be asynchronous)



Kernel

- ▶ „Our“ parallel function
- ▶ Simple implementation

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```



Indexing

- ▶ Assemble the index from block ID and thread ID in the block:
 - ▶ Unique thread ID!

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```



Kernel call

```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



Kernell call (2)

```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```



Overview

- ▶ For many massively parallel problems ...
 - ▶ For these problems, the GPU allows huge speedups!
- ▶ Difficult problems are simplified,
- ▶ Impossible problems (almost) mastered.



Disclaimer

- ▶ What will this lesson present:
 - ▶ Fast-paced introduction
 - ▶ “Know enough to be dangerous”
- ▶ Missing parts int next year (elective course).



The example problem...

Sum two arrays:

$A[] + B[] \rightarrow C[]$

Aim: To find out what's going on ...



CUDA source code

```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



Basic recipe (again)

- ▶ Prepare input on computer - host (CPU-accessible memory)
- ▶ Allocate memory for input on GPU
- ▶ Allocate memory for output on host
- ▶ Allocate memory for output on GPU
- ▶ Copy input from host to GPU
- ▶ Start GPU kernel
- ▶ Copy output from GPU to host

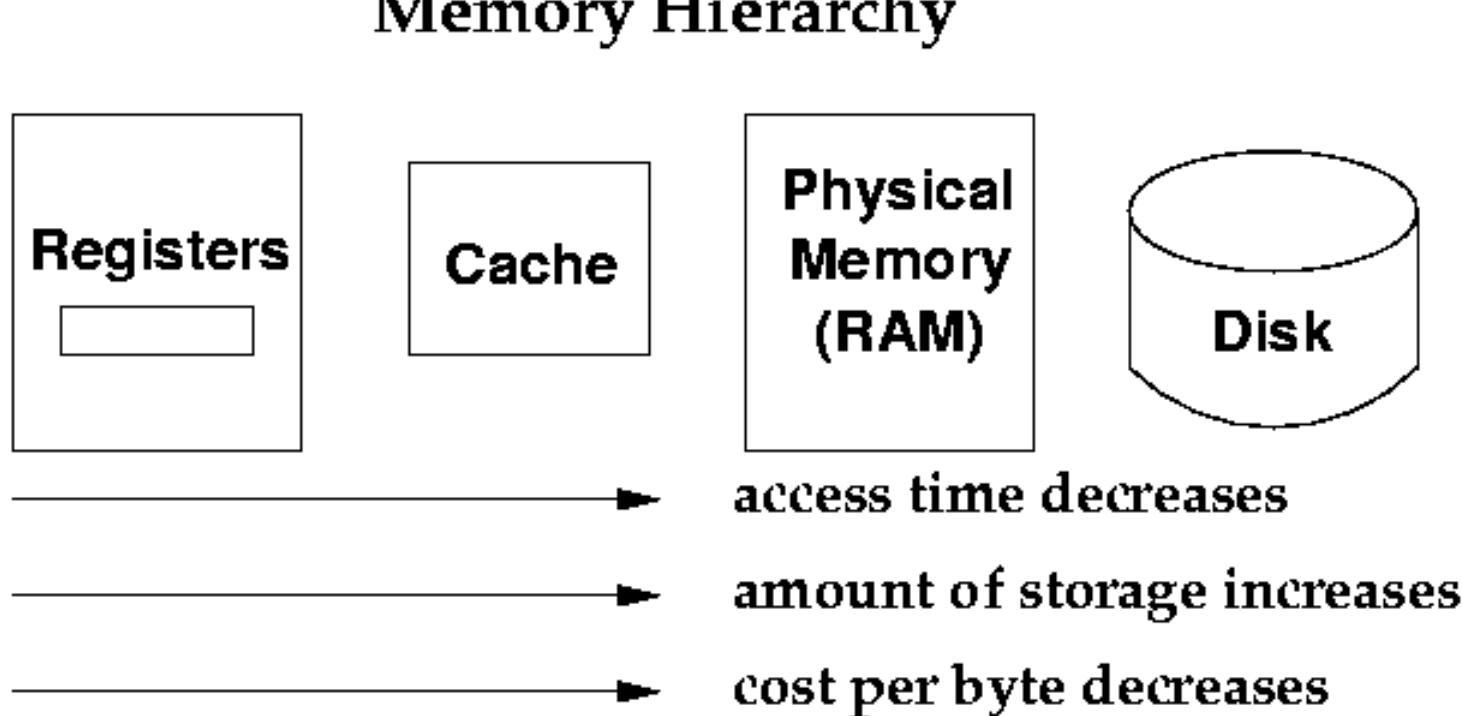
(Copy can be asynchronous)



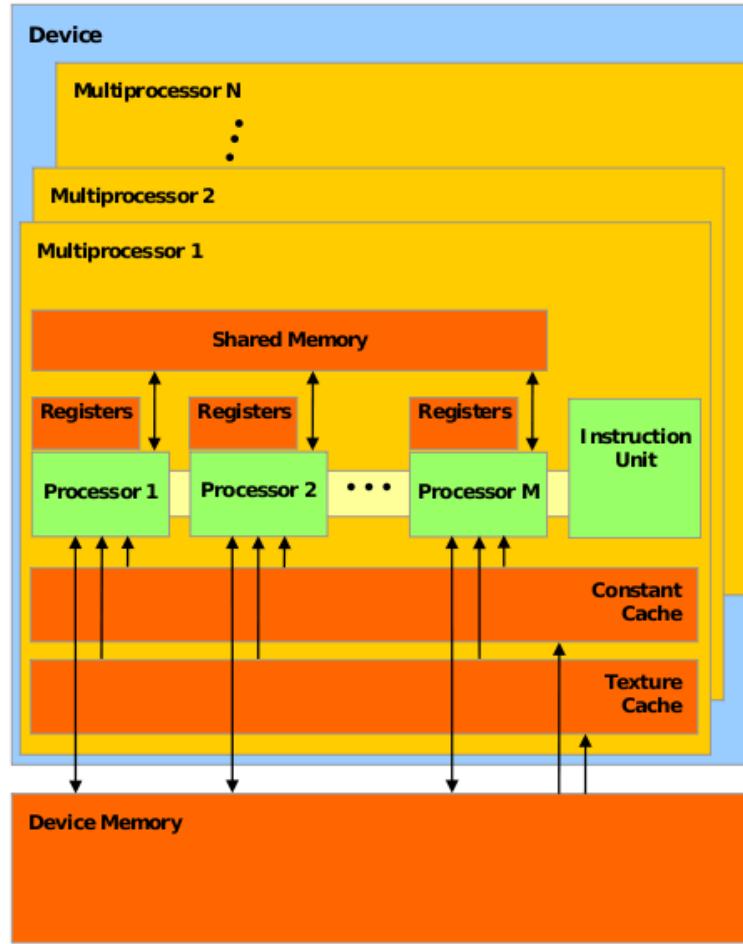
```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



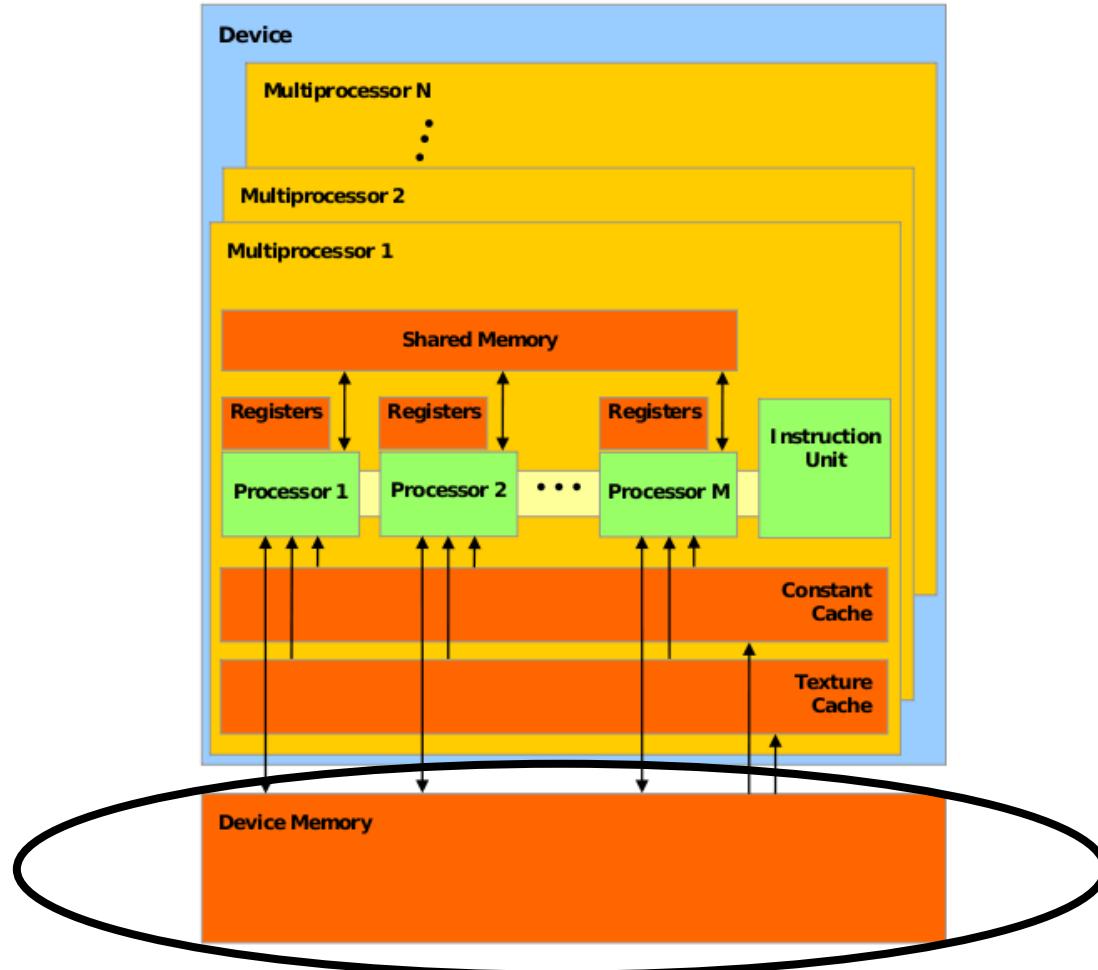
Classic memory hierarchy



GPU



GPU



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



Pointers

- ▶ Difference between CPU and GPU pointers?



Pointers

- ▶ Difference between CPU and GPU pointers?
 - ▶ there is no difference, pointers are only addresses in memory,



Pointers

- ▶ Difference between CPU and GPU pointers?
 - ▶ there is no difference, pointers are only addresses in memory,
 - ▶ the programmer must take care of what is going on.



Pointers

- ▶ Best practice:
 - ▶ Special naming conventions:

e.g. “dev_” prefix



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size)
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);

    // Allocate memory on the GPU for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    → cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    → cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    → cudaMalloc((void **) &dev_c, size*sizeof(float));
```



Memory allocation

- ▶ CPU (host memory)...

```
float *c = malloc(N * sizeof(float));
```

Allocate #bytes in argument



Memory allocation

- ▶ GPU (global memory):

```
float *dev_c;  
cudaMalloc(&dev_c, N * sizeof(float));
```

- ▶ Function signature:

```
cudaError_t cudaMalloc (void ** devPtr, size_t size)
```

- ▶ Allocate #bytes in arg2

- ▶ arg1 is a pointer to a pointer in GPU memory!

- ▶ Parameter to a change function,
- ▶ Result after a successful call: allocated memory on the location given by dev_c on the GPU,
- ▶ Return value is an error code that is "checked".



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    → cudaMalloc((void **) &dev_a, size*sizeof(float));  
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    → cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b  
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    // Allocate memory on the GPU for our outputs:  
    → cudaMalloc((void **) &dev_c, size*sizeof(float));
```



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    → cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    → cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice); // and dev_b  
    → cudaMalloc((void **) &dev_c, size*sizeof(float));  
  
    // Allocate memory on the GPU for our outputs:  
    → cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);
```



Memory copy

- ▶ CPU (host memory)...

```
// source pointers, destination pointers point to memory locations
memcpy(destination, source, N);
```

- ▶ Signature:

```
void * memcpy (void * destination, const void * source, size_t num);
```

- ▶ Copy *num* bytes from source to destination



Memory copy

- ▶ `cudaMemcpy()` is generally useful:

CPU -> GPU

GPU -> **CPU**

GPU -> GPU

CPU -> CPU



Memory copy

Signature:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,  
enum cudaMemcpyKind kind)
```



Memory copy

Podpis:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,  
enum cudaMemcpyKind kind)
```

Vrednosti:

cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice



Memory copy

Signature:

```
cudaError_t cudaMemcpy(void *destination, void *src, size_t count,  
enum cudaMemcpyKind kind)
```

Specifies the treatment of dst. src src. such as CPU or GPU addresses

Values:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice



```
void cudaAddVectors(const float* a, const float* b, float* c, size_t size){  
    //For now, suppose a and b were created before calling this function  
  
    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be  
    // arrays on the GPU.  
  
    float * dev_a;  
    float * dev_b;  
  
    float * dev_c;  
  
    // Allocate memory on the GPU for our inputs:  
    cudaMalloc((void **) &dev_a, size*sizeof(float));  
    → cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);  
  
    → cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);  
    // and dev_b  
  
    // Allocate memory on the GPU for our outputs:  
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```



Memory overview

- ▶ CPU vs GPU pointers
- ▶ `cudaMalloc()`
- ▶ `cudaMemcpy()`



```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```



First part

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    → const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    → const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    → cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

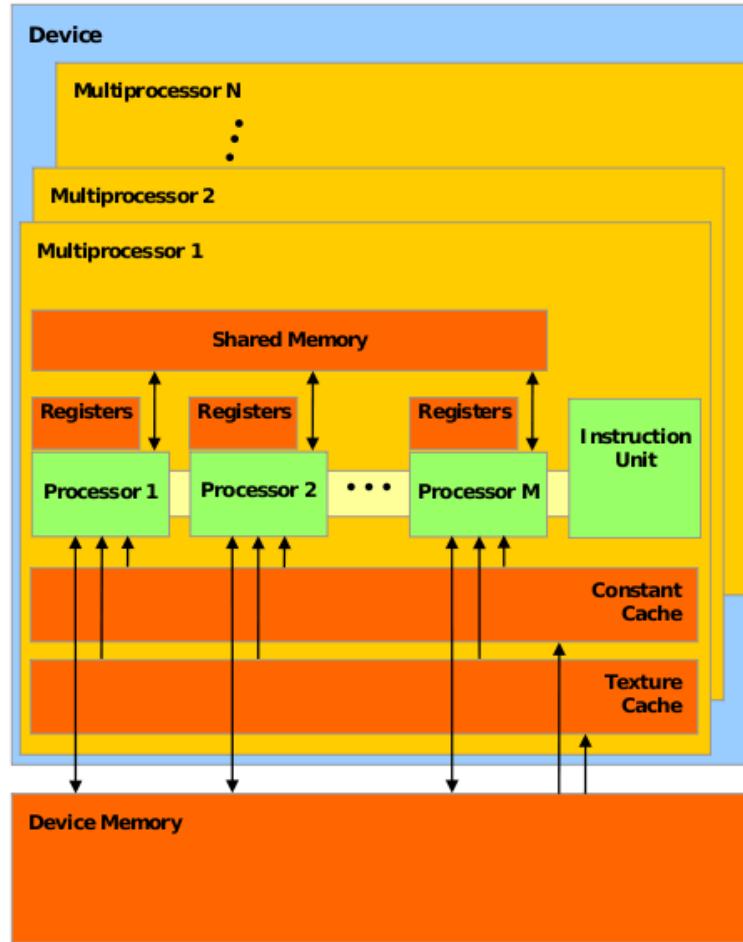


Remember...

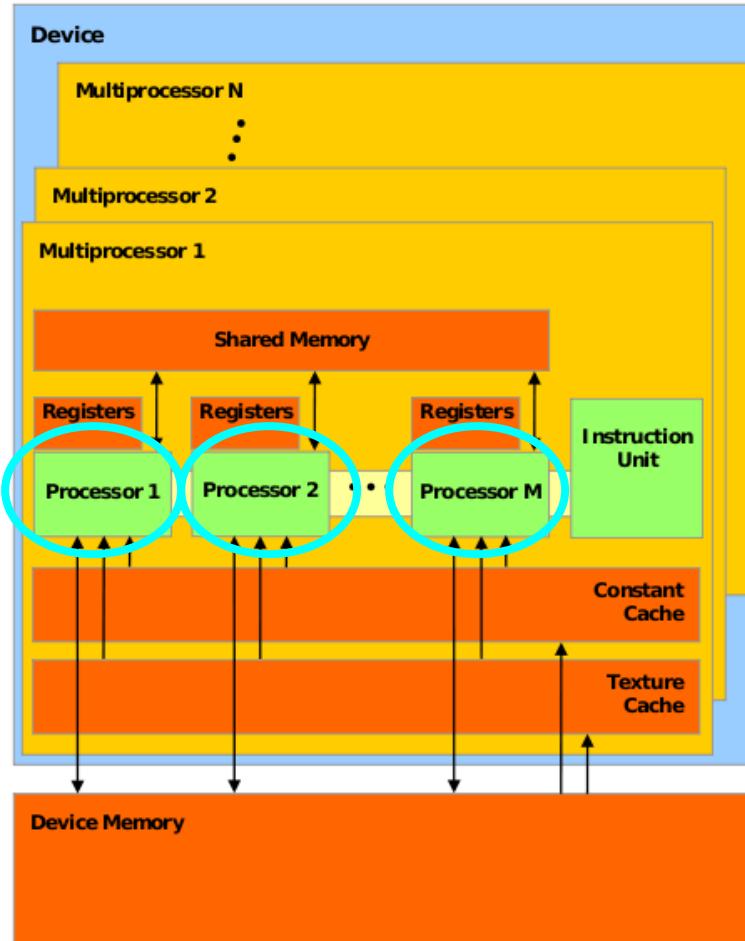
- ▶ GPU...
- ▶ lots of cores
- ▶ Useful for massively parallel problems



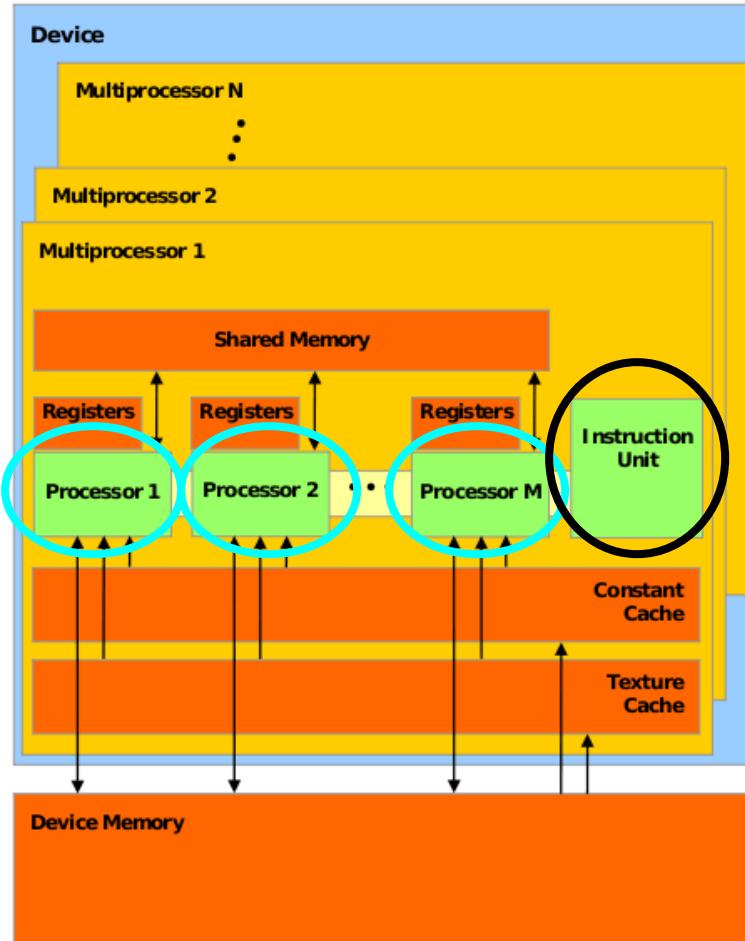
GPU guts



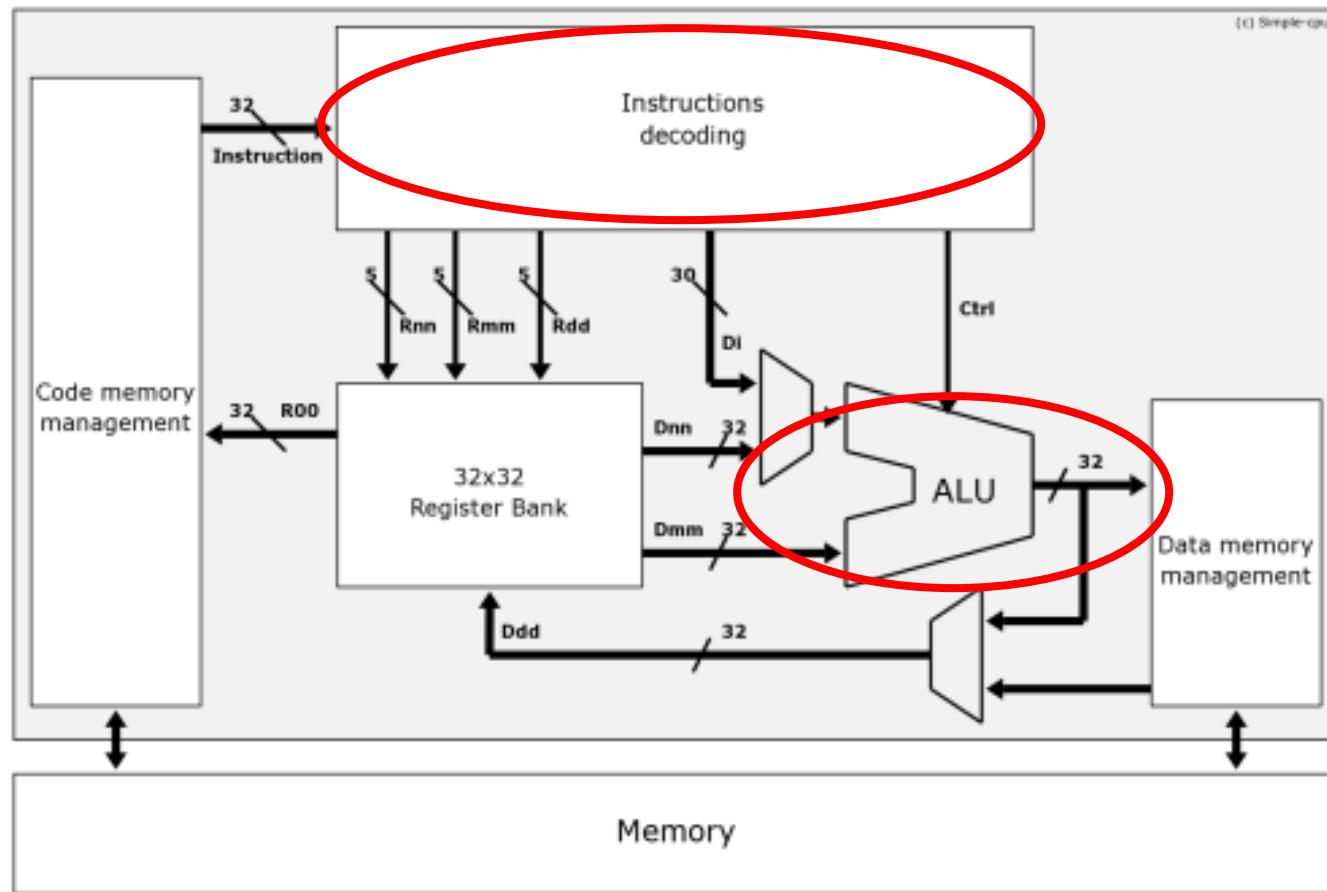
GPU guts



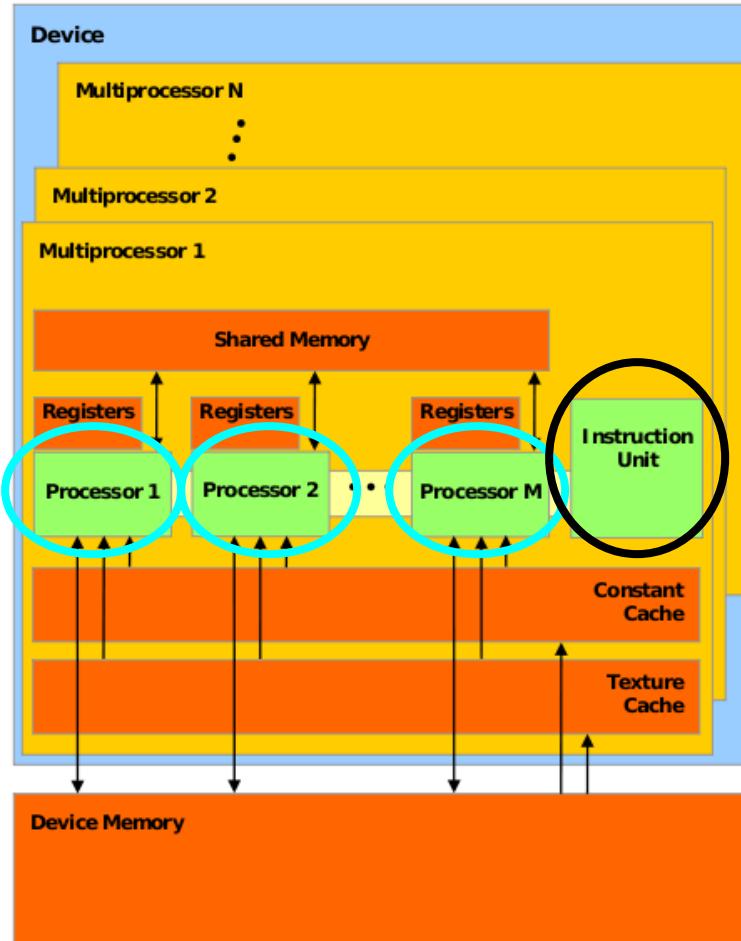
GPU guts



CPU guts



GPU guts

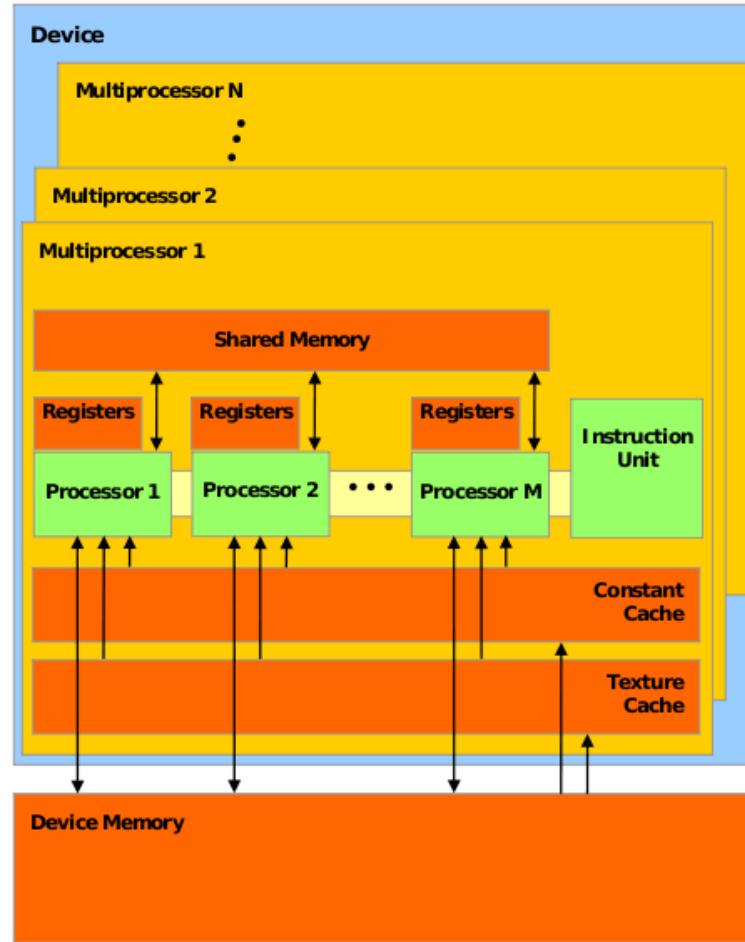


Warps egrilik, çarpıklık, sapma, çelik tel

- ▶ Thread groups that are parallelly executed:
EXECUTE SAME INSTRUCTIONS!
- ▶ “warp”
 - ▶ (32 threads still for amm CUDA implementations)



GPU guts

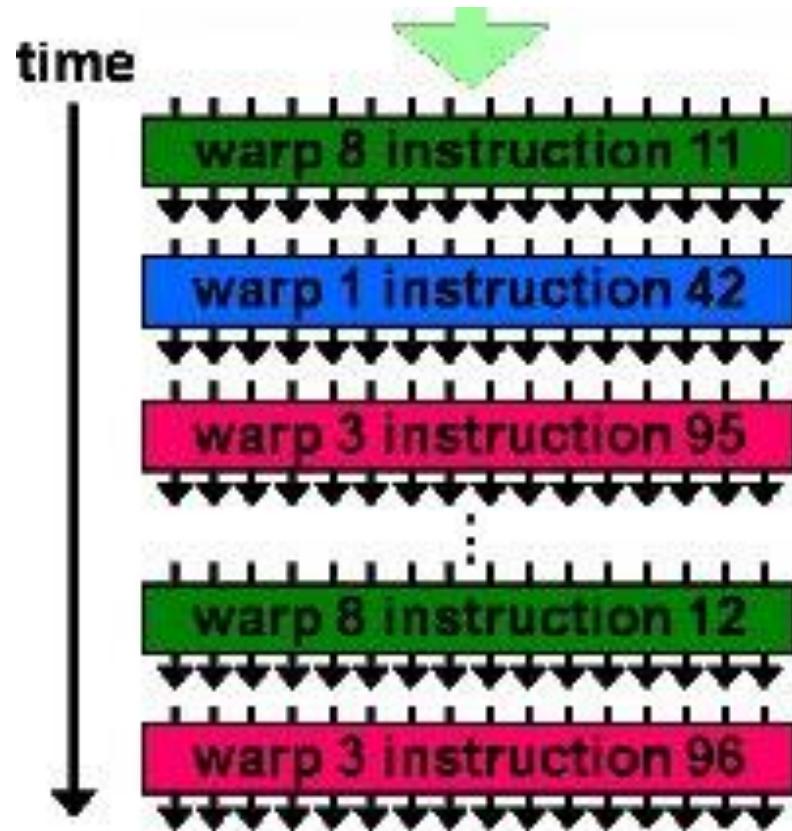


Blocks

- ▶ A group of threads that will be run on a single multiprocessor
- ▶ It contains several warps
- ▶ It has max. limit (depending on GPU, e.g. 512 or 1024)



Multiprocessor execution timeline



Thread groups

- ▶ A *grid* (all threads are started...):
... contains blocks <- assigned to multiprocessors
 - Each block contains warps <- executed at the same time
 - Each warp contains threads



Second part

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    → const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    → const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    → cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```



- ▶ **Lesson 1:**

- ▶ **Start a lot of threads!**

- ▶ Remember: context switch is cheap
 - ▶ Cover latency

- ▶ **Start enough blocks!**

- ▶ Occupy SMs

e.g. Don't call:

```
kernel<<<1,1>>>(); // 1 block, 1 thread per block
```

Call:

```
kernel<<<50,512>>>(); // 50 blocks, 512 threads per block
```



► Lesson 2:

Multiprocessors execute warps (of 32 threads)

Block sizes of $32 \times n$ (integer n) are best

e.g. Don't call:

```
kernel<<<50,97>>>(); // 50 blocks, 97 threads per block
```

Call:

```
kernel<<<50,128>>>(); // 50 blocks, 128 threads per block
```



Summary (processor internals)

- ▶ Kernel call:
 - ▶ Number of threads per block,
 - ▶ number of blocks.



```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
→ cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```



Kernel argument passing

- ▶ Similar to the transfer of arguments in C functions,
- ▶ Rules:
 - ▶ do not download pointers from the host,
 - ▶ small variables are OK (for example, individual ints),
 - ▶ there is no download per reference.



Kernel function

- ▶ Start a lot of threads,
- ▶ threads have a mechanism for unique IDs:
- ▶ Thread index within block
- ▶ Block index

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```



-
- ▶ Out of bounds issue:
 - ▶ If $\text{index} > (\#\text{elements})$, illegal access!

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```



- ▶ Out of bounds issue:
- ▶ If $\text{index} > (\#\text{elements})$, illegal access!

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        c[index] = a[index] + b[index];
    }
}
```

- ▶ #Threads issue:
 - ▶ Cannot start 1e9 threads!
 - ▶ Threads must process any number of elements.

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        c[index] = a[index] + b[index];
    }
}
```

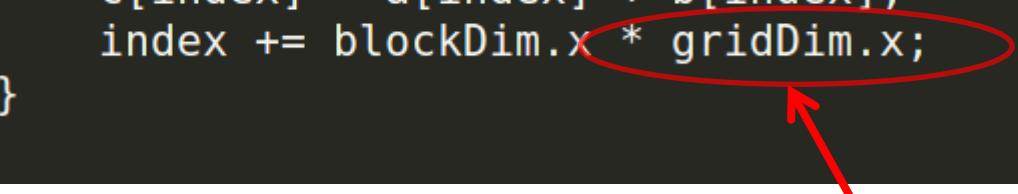


- ▶ #Threads issue:
- ▶ Cannot start 1e9 threads!
- ▶ Threads must process any number of elements.

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    while (index < size) {
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
    }
}
```

- ▶ #Threads issue:
- ▶ Cannot start 1e9 threads!
- ▶ Threads must process any number of elements.

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c, int size) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    while (index < size) {
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
    }
}
```



Kernel start

```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

→ cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```



Kernel start

```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```



-
- ▶ `cudaFree()`
 - ▶ Equivalent to `free()` function from host,
 - ▶ (as on host) release memory after usage.



Example: SAXPY - Single Precision a x plus y

- ▶ instead of "Hello world",
- ▶ look at the serial implementation,
- ▶ look at CUDA implementation,
- ▶ make two arrays size 2^{30} elements,
- ▶ calculate SAXPY for each index,
- ▶ do this on CPU = i5,
- ▶ do this on GPU = GTX 1050ti,
- ▶ measure time ...



Overview

- ▶ GPU global memory:
 - ▶ Pointers (CPU vs GPU)
 - ▶ cudaMalloc() in cudaMemcpy()
- ▶ GPU processor details:
 - ▶ Hierarchy of thread groups
 - ▶ startup parameters
- ▶ Threads in kernel

