Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

## Algorithms and Programming IV

# (Concurrency with) Threads

**Summer Term 2021 | 23.04.2021**

**Barry Linnert**

# Objectives of Today‘s Lecture

- Performance from a user perspective
- Limits to concurrency by processes
- Introduction of Threads
- Usage of shared data
- Characterization of term Critical section
  - Solutions to protect the Critical section

Freie Universität Berlin

Concepts of Non-sequential and Distributed Programming

# CONCURRENCY WITH THREADS

# Requirements for Programs

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

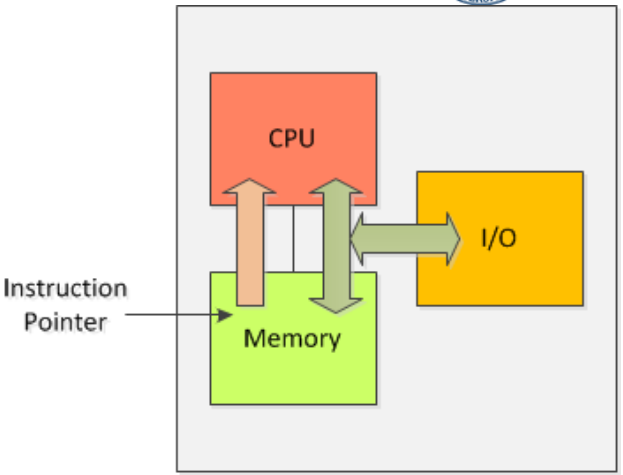
- A program should comply with certain requirements about its behavior.
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …

# Correctness

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

# Requirements for Programs

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

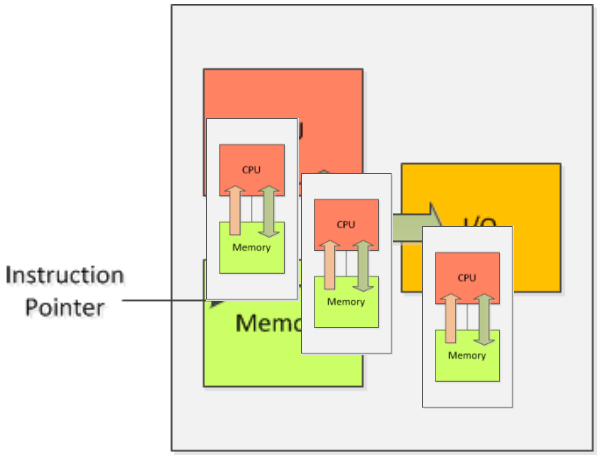



- A program should comply with certain requirements about its behavior.
  - Non-functional requirements, such as
    - Performance
    - Usability
    - Security
    - …

# Performance

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

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



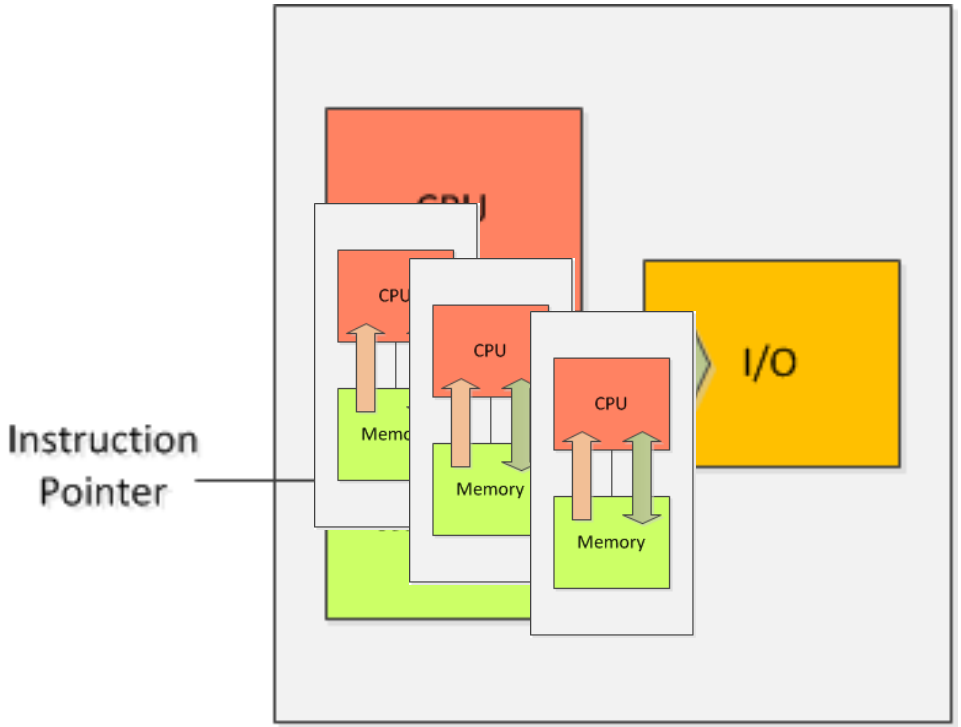

- Perspective of the user
  - the fastest possible processing of your own program
  - short response time

# Performance from the User Perspective

- Process switching comes with an overhead.
- But, if the program is designed to use more resources than CPU (and memory) this usage of the different resources may be performed in parallel (parts of the program are executed concurrently).
- Problem has to be split into reasonable pieces (e.g. via divide and conquer approaches).
- The usage of all of the resources may lead to a reduction of idle time as well as to a reduction of the response time of the entire program.
- The uniform progress of the processes of the program (and all other processes) is ensured by the operating system. It needs to correspond to goals and possibilities of the operating system (scheduling).

# Virtualization of the Processor

- That is our current machine model (execution model):



# LIMITS TO CONCURRENCY BY PROCESSES

# Program with several Processes

- To get some positive effect to the performance of the solution the problem is to be split into reasonable pieces. Approaches, such as divide and conquer may be used.
- To perform the partitioning of the data representing the problem into reasonable pieces the parent process can perform the splitting before the forking of the (working) child processes.
- Thus, the parent process can provide the data for the child processes due to the duplication of the address space (by the operating system).

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

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

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

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

  return 0;
}
```

# Program with several Processes

- To get some positive effect to the performance of the solution the problem is to be split into reasonable pieces. Approaches, such as divide and conquer may be used.
- To perform the partitioning of the data representing the problem into reasonable pieces the parent process can perform the splitting before the forking of the (working) child processes.
- Thus, the parent process can provide the data for the child processes due to the duplication of the address space (by the operating system).

- The result of the calculation of every piece of the problem is transmitted by the return value of the child process.

Is this approach sufficient?

# Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

# Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

```
// simple matrix multiplication


int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];

  int i, j, h;

  // init data                                  // doing the work


                                                return 0;
}
```

```c
// simple matrix multiplication

#include <stdlib.h>

int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];

  int i, j, h;

  // init data
  srand ((unsigned) time (NULL));

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand()
        / (RAND_MAX - 1)) * 100);


    }
  }
```

```c
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      mb[i][j] = (int)(((double)
        rand() / (RAND_MAX - 1)) * 100);


    }
  }

  // doing the work


  return 0;
}
```

```c
// simple matrix multiplication
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];

  int i, j, h;

  // init data
  srand ((unsigned) time (NULL));

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         ma[i][j]);
    }
  }

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      mb[i][j] = (int)(((double)
         rand() / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         mb[i][j]);
    }
  }

  // doing the work

  return 0;
}
```

```c
// simple matrix multiplication
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];
  int result = 0;
  int i, j, h;

  // init data
  srand ((unsigned) time (NULL));

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         ma[i][j]);
    }
  }

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      mb[i][j] = (int)(((double)
         rand() / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         mb[i][j]);
    }
  }

  // doing the work
  for (h = 0; h < 2; h++) {
    for (i = 0; i < 2; i++) {

      for (j = 0; j < 3; j++) {
        result += (ma[h][j] * mb[j][i]);
      }

    }
  }
  return 0;
}
```

```c
// simple matrix multiplication
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];
  int result = 0;
  int i, j, h;

  // init data
  srand ((unsigned) time (NULL));

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         ma[i][j]);
    }
  }

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      mb[i][j] = (int)(((double)
         rand() / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         mb[i][j]);
    }
  }

  // doing the work
  for (h = 0; h < 2; h++) {
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("%d,%d : ", h, i);
      for (j = 0; j < 3; j++) {
        result += (ma[h][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
  }
  return 0;
}
```

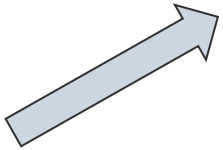
```c
// simple matrix multiplication
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int ma[2][3];
  int mb[3][2];
  int result = 0;
  int i, j, h;

  // init data
  srand ((unsigned) time (NULL));

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         ma[i][j]);
    }
  }

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 2; j++) {
      mb[i][j] = (int)(((double)
         rand() / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         mb[i][j]);
    }
  }

  // doing the work
  for (h = 0; h < 2; h++) {
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("%d,%d : ", h, i);
      for (j = 0; j < 3; j++) {
        result += (ma[h][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
  }
  return 0;
}
```

```
// simple matrix multiplication
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...
```

```
// doing the work
for (h = 0; h < 2; h++) {
  for (i = 0; i < 2; i++) {
    result = 0;
    printf ("%d,%d : ", h, i);
    for (j = 0; j < 3; j++) {
      result += (ma[h][j] * mb[j][i]);
    }
    printf (" %d \n", result);
  }
}
```

```
return 0;
}
```

```c
// simple matrix multiplication
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...
```



```c
// doing the work
for (h = 0; h < 2; h++) {
  for (i = 0; i < 2; i++) {
    result = 0;
    printf ("%d,%d : ", h, i);
    for (j = 0; j < 3; j++) {
      result += (ma[h][j] * mb[j][i]);
    }
    printf (" %d \n", result);
  }
}
```

The outermost for loop is a good candidate to split the problem (work) into reasonable pieces.

```c
return 0;
}
```

```
// simple matrix mult. with 2 procs
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...
```

```
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("0,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[0][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
```

```
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("1,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[1][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
```

```
  return 0;
}
```

```
// simple matrix mult. with 2 procs
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...

  pid = fork();

  if (pid == 0) {
    // child process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("0,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[0][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
    exit(0);
  }
  else if(pid > 0) {
    // parent process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("1,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[1][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }


  }
  else {
    printf ("fork() failed\n");
    exit (EXIT_FAILURE);
  }
  return 0;
}
```

```
// simple matrix mult. with 2 procs
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...

  pid = fork();

  if (pid == 0) {
    // child process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("0,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[0][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
    exit(0);
  }
  else if(pid > 0) {
    // parent process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("1,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[1][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
    pid = wait(&status);
    printf ("\n  %d\n",
        WEXITSTATUS(status));
    exit(EXIT_SUCCESS);
  }
  else {
    printf ("fork() failed\n");
    exit (EXIT_FAILURE);
  }
  return 0;
}
```

```
// simple matrix mult. with 2 procs
#include ...
int main (int argc, char *argv[]) {
  int ma[2][3];
  int mb[3][2];
  ...
  // init data
  ...

  pid = fork();

  if (pid == 0) {
    // child process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("0,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[0][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
    exit(0);
  }
  else if(pid > 0) {
    // parent process is doing work
    for (i = 0; i < 2; i++) {
      result = 0;
      printf ("1,%d : ", i);
      for (j = 0; j < 3; j++) {
        result += (ma[1][j] * mb[j][i]);
      }
      printf (" %d \n", result);
    }
    pid = wait(&status);
    printf ("\n  %d\n",
        WEXITSTATUS(status));
    exit(EXIT_SUCCESS);
  }
  else {
    printf ("fork() failed\n");
    exit (EXIT_FAILURE);
  }
  return 0;
}
```

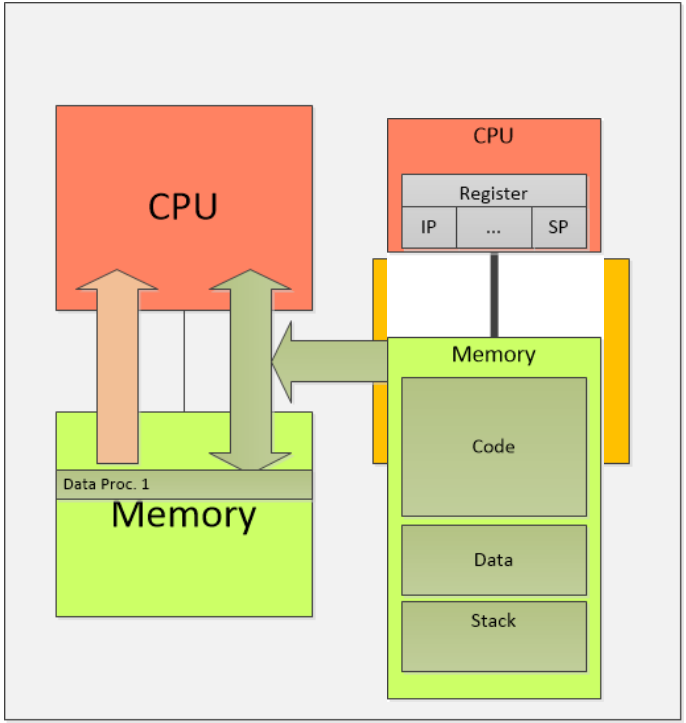
# Program with several Processes

- To get some positive effect to the performance of the solution the problem is to be split into reasonable pieces. Approaches, such as divide and conquer may be used.
- To perform the partitioning of the data representing the problem into reasonable pieces the parent process can perform the splitting before the forking of the (working) child processes.
- Thus, the parent process can provide the data for the child processes due to the duplication of the address space (by the operating system).
- But, until now the result of the calculation of every piece of the problem is transmitted by the return value of the child process.
  - That is not sufficient for an number of reasons:
    - The result may consists of more data than just one value (integer).
    - The data may be used by other processes more than once.

Are there any other reasons?

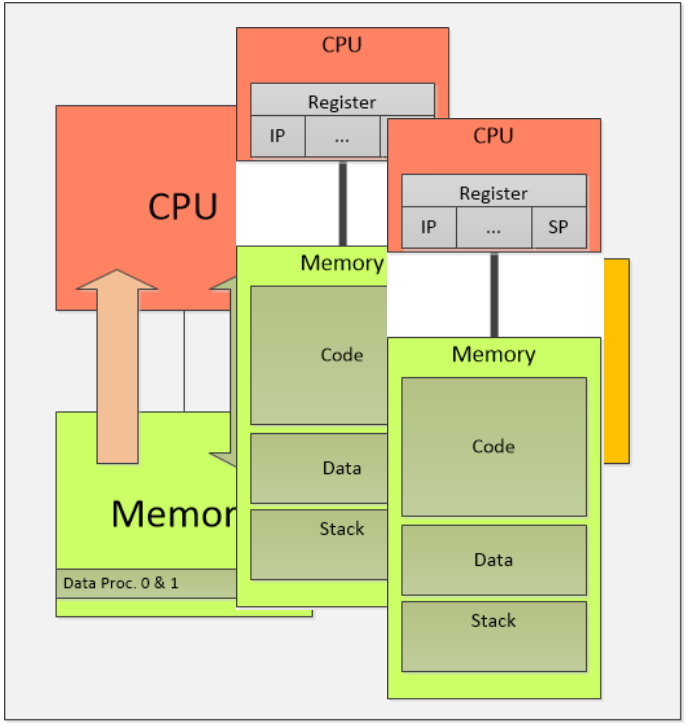
# Concurrent Work on Shared Data I

- A process uses an address spaces of it’s own.
  - The address spaces of different processes are separated by operating system and hardware.
  - The separation of the address spaces provides protection against (unauthorized) access from other processes (of other users) → Security.



# Concurrent Work on Shared Data II

- All of the separated address spaces use the same main memory.
- To solve the problem of data exchange between different program execution paths (we call them processes until now) the address spaces can be used by more than one of these execution paths.
- Using the same address space the different program execution paths may work on the same data and can store the results in common variables or addresses.



Freie Universität Berlin

# THREADS

# Threads

- These „ different program execution paths” are called Threads.

Why does threads have a stack of their own?

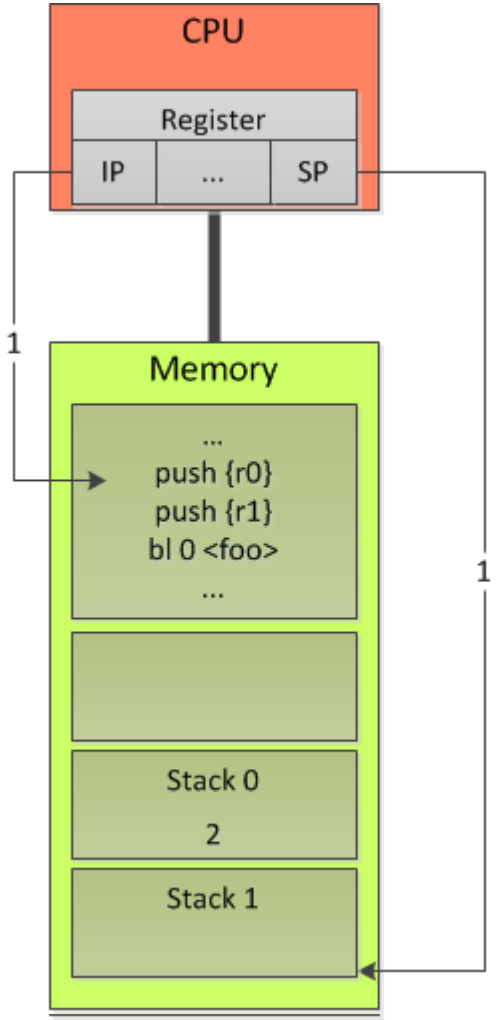
- Threads share an address space, but use a stack by there own.
  - Text/code segment and data segment (heap) are shared between all threads of the group of threads using the same address space.
  - Sometimes the first thread instance linked to a specific address space is called process.
    - Using this characterization the process holds one or more threads.

- To use threads in different environments there is a standardized interface: POSIX
  - pthreads

# Threads

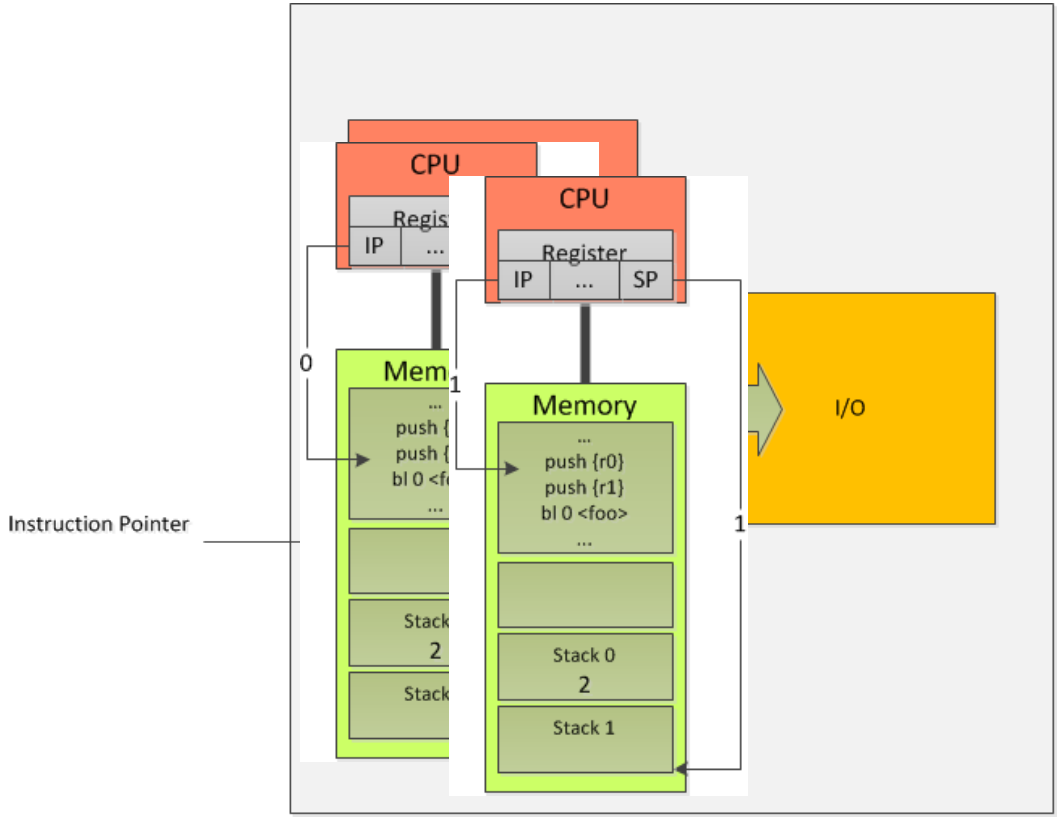
- One shared address space contains text and data segment, but different stacks.

- Switch between threads is performed by operating system (or thread environment).
  - The point in time the switch to assign the CPU to another thread is performed is unpredictable to the thread.
  - In this way the thread resembles a process.



# Machine Model

- The introduction of threads extends our machine model (or execution model):
- Different processes are represented by different address spaces which hold different threads. (The threads share the specific address space of the process.)



# Using Pthreads

- Due to the broad approach and the purpose to have an universal interface the POSIX threads differ in fundamental issues from processes.
- Pthreads have to be managed by the program itself.
- A new pthread does not start right after the invocation, but starts execution at a given address in text/code segment.
  - A function pointer should be used to give the thread a starting point for execution.

```
// simple example for pthreads


#include <stdlib.h>
#include <stdio.h>



void *PrintHello (?)
{



  printf ("Hello World!
         It's me, thread #%ld!\n", ?);



}
```

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


  long t;
  for (t=0; t < NUM_THREADS; t++) {
     printf ("In main:
         creating thread %ld\n", t);


  }



}
```

```c
// simple example for pthreads

#include <pthread.h>
#include <stdlib.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);
}
```

```c
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);

  }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple example for pthreads

#include <pthread.h>
#include <stdlib.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);
}
```

```c
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);
     }
  }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple example for pthreads

#include <pthread.h>
#include <stdlib.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);
}
```

```c
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);
     }
  }

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

# Example: Matrix Multiplication

- Using threads the matrix multiplication program can store the results in shared data.
  - Every thread has access to the shared data and can store the part of the resulting matrix.
- Additionally the input data is shared and is not to be stored more than once.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

```c
// simple matrix mult. with pthreads
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define NUM_THREADS     3

// shared data
int ma[3][3];
int mb[3][3];
int result[3][3];

// calculation
void *matr_mult (void *threadid)
{
  // doing the calculation

  pthread_exit (NULL);
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...

// calculation
void *matr_mult (void *threadid)
{
  long tid;
  int i, j;
  tid = (long) threadid;


  for (i = 0; i < 3; i++) {
    result[tid][i] = 0;
    for (j = 0; j < 3; j++) {
      result[tid][i] += (ma[tid][j] *
         mb[j][i]);
    }
  }
  pthread_exit (NULL);
}
```

```
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...

// calculation
void *matr_mult (void *threadid)
{
  long tid;
  int i, j;
  tid = (long) threadid;
  printf ("Hello World! It's me, thread
        #%ld !\n", tid);
  for (i = 0; i < 3; i++) {
    result[tid][i] = 0;
    for (j = 0; j < 3; j++) {
      result[tid][i] += (ma[tid][j] *
        mb[j][i]);
    }
  }
  pthread_exit (NULL);
}
```

```
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  srand ((unsigned) time (NULL));
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand ()
         / (RAND_MAX - 1)) * 100);


    }
  }
```

```
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      mb[i][j] = (int)(((double) rand ()
         / (RAND_MAX - 1)) * 100);


    }
  }

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  srand ((unsigned) time (NULL));
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      ma[i][j] = (int)(((double) rand ()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         ma[i][j]);
    }
  }
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      mb[i][j] = (int)(((double) rand ()
         / (RAND_MAX - 1)) * 100);
      printf ("%d,%d: %d \n", i, j,
         mb[i][j]);
    }
  }

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  ...
```

```
  // creating threads

  // joining threads

  // output results
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      printf (" %d", result[i][j]);
    }
    printf ("\n");
  }



  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  ...
```

```
// creating threads
for (t = 0; t < NUM_THREADS; t++) {


   rc = pthread_create (&threads[t],
       NULL, matr_mult, (void *)t);




}
// joining threads
for (t = 0; t < NUM_THREADS; t++) {
  pthread_join (threads[t],NULL);
}

// output results
...
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  ...
```

```
// creating threads
for (t = 0; t < NUM_THREADS; t++) {
   printf ("In main: creating thread
       %ld\n", t);
   rc = pthread_create (&threads[t],
       NULL, matr_mult, (void *)t);
   if (rc) {
      printf ("ERROR; return code from
       pthread_create () is %d\n", rc);
      exit (-1);
   }
}
// joining threads
for (t = 0; t < NUM_THREADS; t++) {
  pthread_join (threads[t],NULL);
}

// output results
...
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

```
// simple matrix mult. with pthreads
#include ...
#define NUM_THREADS     3
// shared data
...
// calculation
void *matr_mult (void *threadid) {
  ...
}

int main (int argc, char *argv[]) {
  ...
  // init data
  ...
// creating threads
for (t = 0; t < NUM_THREADS; t++) {
   printf ("In main: creating thread
       %ld\n", t);
   rc = pthread_create (&threads[t],
       NULL, matr_mult, (void *)t);
   if (rc) {
      printf ("ERROR; return code from
       pthread_create () is %d\n", rc);
      exit (-1);
   }
}
// joining threads
for (t = 0; t < NUM_THREADS; t++) {
  pthread_join (threads[t],NULL);
}

// output results
...
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

# Example: Matrix Multiplication – Output

```
0,0: 69
0,1: 8
0,2: 24
1,0: 65
1,1: 23
1,2: 94
2,0: 10
2,1: 33
2,2: 36
0,0: 51
0,1: 5
0,2: 51
1,0: 58
1,1: 1
1,2: 0
2,0: 37
2,1: 7
2,2: 35
```

```
In main: creating thread 0
Hello World! It's me, thread #0 !
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #1 !
 4871 521 4359
 8127 1006 6605
 0 0 0
Hello World! It's me, thread #2 !
```

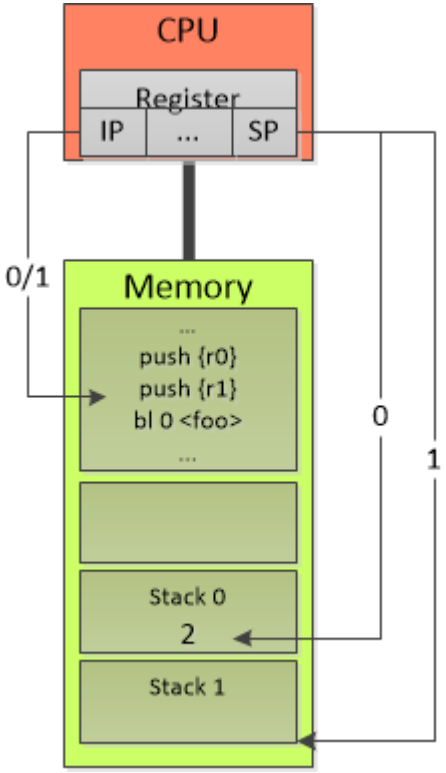
# Example: Matrix Multiplication – Conclusion

- Using threads, the work of the matrix multiplication can be split and distributed easily.
- Shared data reduces the amount of main memory needed.
- Storing and exchange of data between the threads is performed easily by using common (global) variables.

# USING SHARED DATA

# Using Shared Data for Intermediate Results

- Until now the shared data was used for writing final results and for reading input data only.
- Some applications need to work on intermediate results.
- With threads these intermediate results may serve as input data for other threads.



# Example: Accounting

- A bank transfers money from one account to another.
- The amount of money to be debit from one account equals the amount of money that is transferred to the other account.
  - Money does not disappear or is created out of nothing.
- The transfers usually are performed on a multitude of accounts and more than once between different accounts.
- So the tasks may be performed concurrently.

- In our example two different threads transfer money between two accounts.

```c
// simple accounting with pthreads

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define NUM_THREADS     2

// shared data
int account[2];

// accounting
void *bank_action (void *threadid)
{
  // doing transfers

  pthread_exit (NULL);
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount = 0;

  tid = (long) threadid;




    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;

  pthread_exit (NULL);
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount = 0;

  tid = (long) threadid;


  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
  }
  pthread_exit (NULL);
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount = 0;

  tid = (long) threadid;
  printf ("Hello World! It's me, thread
        #%ld !\n", tid);
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
  }
  pthread_exit (NULL);
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data

  // creating threads

  // joining threads

  // output results

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```c
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid) {
  ...
}


int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data
  srand ((unsigned) time (NULL));
  account[0] = account[1] = 100;

  // creating threads

  // joining threads

  // output results
  for (i = 0; i < NUM_THREADS; i++) {
      printf (" account_%d: %d \n", i,
         account[i]);

  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid) {
  ...
}


int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data
  ...
```

```
// creating threads
for (t = 0; t < NUM_THREADS; t++) {
   printf ("In main: creating thread
       %ld\n", t);
   rc = pthread_create (&threads[t],
       NULL, bank_action, (void *)t);
   if (rc) {
      printf ("ERROR; return code from
       pthread_create () is %d\n", rc);
      exit (-1);
   }
}
// joining threads
for (t = 0; t < NUM_THREADS; t++) {
  pthread_join (threads[t], NULL);
}

// output results
...
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS     2
// shared data
int account[2];

// accounting
void *bank_action (void *threadid) {
  ...
}


int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int i, j;

  // init data
  ...
```

```
// creating threads
for (t = 0; t < NUM_THREADS; t++) {
   printf ("In main: creating thread
       %ld\n", t);
   rc = pthread_create (&threads[t],
       NULL, bank_action, (void *)t);
   if (rc) {
      printf ("ERROR; return code from
       pthread_create () is %d\n", rc);
      exit (-1);
   }
}
// joining threads
for (t = 0; t < NUM_THREADS; t++) {
  pthread_join (threads[t], NULL);
}

// output results
...
/* Last thing that main() should do */
pthread_exit(NULL);
}
```

# Example: Accounting – Output

- First run:

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: 3174954
 account_1: -3955026
```

- Second run:

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: -1142039
 account_1: -1270137
```

# Example: Accounting – Conclusion I

- The accounting program with two threads is not correct!
- As the accounts are not balanced there must be a problem with the manipulation of the account values.
  - The two command lines should change the two account variables – one for every account.
  - A given amount of money is taken from on account by subtracting this value from the value stored in the first account variable. The same amount is added to the value of the second account variable.

```
// simple accounting with pthreads
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;

  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
  }

  pthread_exit (NULL);
}
```

# Example: Accounting – Conclusion II

- The amount of money to be transferred is not changed as the variable to hold this value is a local one. So it is used only by one thread. Every thread has it's own variable storing the amount of money to be transferred.

- But, the variables representing the accounts are global variables holding shared data.

```
// simple accounting with pthreads
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;

  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
  }

  pthread_exit (NULL);
}
```

# Example: Accounting – Conclusion III

```
// simple accounting with pthreads
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
```

- Every transfer can be seen as a sequence of at least 8 machine instructions:
  - The high level operation is **not a atomic operation**!
- Thus, the value of one account variable can be changed by the second thread in time the first one is transferring the money from one to another account.
  - There is **no sequential execution**.

```
read_into_reg_r0 (account[tid]);
read_into_reg_r1 (amount);
sub_reg_r0_r1;
store_reg_r0 (account[tid]);

read_into_reg_r0 (account[NUM_THREADS-1-tid]);
read_into_reg_r1 (amount);
add_reg_r0_r1;
store_reg_r0 (account[tid]);
```

- The correct manipulation of the shared variables are critical for the correctness of the whole program.

Freie Universität Berlin

# CRITICAL SECTION

# Critical Section I

- For  a correct execution of the program the usage of the shared data must be performed in a **deterministic** way.
- The foundation for the deterministic execution of the program – or sections of the program – is the **sequential processing** of the operations, commands, and instructions.
  - Atomic operations provide, per se, a sequential processing (of this one operation).

- Definition: A program section (series of operations) that only one thread is permitted to execute at a time to ensure the correct execution of the section and the whole program is called **critical section**.

# Critical Section II

- If more than one thread is processing this program section concurrently, the program may be executed incorrectly.
  - The faulty execution need not occur with every program run.

- Operations of the critical section create inconsistent states on the processed data until all of the operations of the critical section are executed successfully.

# Mutual Exclusion

- To protect the critical section it is to be enforced that only one thread is able to enter the critical section.

- Thus, all other threads have to be excluded from entering the critical section.
- This is called **mutual exclusion**.

# Requirements for Solutions to protect the Critical Section

- The solution has to protect the critical section reliably by mutual exclusion.
- The solution should be used in higher level programming languages.
  - Thus, the solution is usable on different architectures providing portability for the program using it.

# Lock

- A lock is a variable that indicates whether a critical section can be entered by a thread.
- The lock has to be a variable that can be accessed by all of the threads.
- If the lock is set (unequal 0), the requesting thread should not/can’t access the critical section.
- Entering the critical section the thread has to set the lock first.
- After releasing the critical section the lock has to be unset (write 0 to the lock variable).

# Example: Accounting – Critical Section

- Following, the protection of the critical section will be evaluated using the well known example of the transferring money from one account to another.
- The critical section contains the operations on the account variables.

```
// simple accounting with pthreads
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;

  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;

 }

  pthread_exit (NULL);
}
```

# Lock

- Using a global variable as lock variable the lock has to be set by the thread entering the critical section.
- As it is not clear if or when the critical section is free or will released be the other thread the requesting thread has to check the lock in a loop.

```
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
```

```
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;

  }
  pthread_exit (NULL);
}
```

```c
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
char lock = 0;
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);
    // try to enter the critical section
    while (lock)
      ;
    lock = 1;
    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    lock = 0;
  }
  pthread_exit (NULL);
}
```

# Example: Accounting – Lock

- Checking and setting the lock variable is by itself a critical section.

- The **simple lock approach does not meet** all requirements for a solution to protect a critical section.

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: 100
 account_1: 100

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: 6277461
 account_1: -6368446
```

# Twofold Lock

- As the simple lock constitutes a critical section by itself, this critical section has to be protected, too.
  - The critical section consists of checking and setting of the one lock variable which both of the threads try to do concurrently.
- To avoid the concurrent access to the single lock variable every thread gets a lock variable of its own. The thread checks if the other one hold its lock and sets its own if not.
- Checking and setting is now performed on different lock variables.

- To handle the locking more easily the set and unset of the lock variables are encapsulated into functions.

```c
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;

  }
  pthread_exit (NULL);
}
```

```c
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {
  while (_lock[NUM_THREADS - 1 - tid])
    ;
  _lock[tid] = 1;
  return 0;
}

int unlock (long tid) {
  _lock[tid] = 0;
  return 0;
}
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;

  }
  pthread_exit (NULL);
}
```

```c
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {
  while (_lock[NUM_THREADS - 1 - tid])
    ;
  _lock[tid] = 1;
  return 0;
}

int unlock (long tid) {
  _lock[tid] = 0;
  return 0;
}
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // try to enter the critical section
    lock (tid);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    unlock (tid);
  }
  pthread_exit (NULL);
}
```

# Example: Accounting – Twofold Lock

- Still, the checking and setting of the lock variables is itself a critical section.
  - Thus, the processing of lock() and unlock() which can be interrupted and may lead to an inconsistent state of the lock.
- The critical section includes both of the lock variables.
- The **twofold lock approach does not meet** all requirements for a solution to protect a critical section.

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: -2415554
 account_1: 2352894

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: -284407
 account_1: -48137
```

# Twofold Lock with Primary Protection

- The critical section of setting the lock is unprotected as the first thread may require access even if it has released the lock shortly before.
- Thus, it is to prevent the thread releasing the lock gets it again right in time the other thread checks the state of the lock variable.
- The lock variable of the thread is set before the checking of the lock variable of the other thread.

```
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {



  return 0;
}

int unlock (long tid) {

  return 0;
}
```

```
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // try to enter the critical section
    lock (tid);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    unlock (tid);
  }
  pthread_exit (NULL);
}
```

```c
// simple accounting with pthreads

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

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {
  _lock[tid] = 1;
  while (_lock[NUM_THREADS - 1 - tid])
    ;
  return 0;
}

int unlock (long tid) {
  _lock[tid] = 0;
  return 0;
}
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // try to enter the critical section
    lock (tid);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    unlock (tid);
  }
  pthread_exit (NULL);
}
```

# Twofold Lock with Primary Protection

- The threads block each other.
- There is a deadlock!

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #1 !
Hello World! It's me, thread #0 !
```

- How about the requirements for a solution to protect a critical section?

# Requirements for Solutions to protect the Critical Section

- The solution has to protect the critical section reliably by mutual exclusion.
- The solution should be used in higher level programming languages.
  - Thus, the solution is usable on different architectures providing portability for the program using it.
- The solution must not lead to a deadlock.

# Twofold Lock with Primary Protection

- The threads block each other.
- There is a deadlock!

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #1 !
Hello World! It's me, thread #0 !
```

- The **twofold lock with primary protection approach does not meet** all requirements for a solution to protect a critical section.

# Twofold Lock with Mutual Precedence

- To prevent the deadlock the lock has to be released in case the requesting thread will not be able to get the lock.
- Thus, the thread will after acquiring the lock give way to the other thread in case it is requesting the lock, too.
- To give the other thread a chance to get the lock, the first one will wait shortly.

```
// simple accounting with pthreads
#include ...

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {
  _lock[tid] = 1;

  while (_lock[NUM_THREADS - 1 - tid]) {




  }
  return 0;
}

int unlock (long tid) {
  _lock[tid] = 0;
  return 0;
}
```

```
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // try to enter the critical section
    lock (tid);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    unlock (tid);
  }
  pthread_exit (NULL);
}
```

```c
// simple accounting with pthreads
#include ...

#define NUM_THREADS   2

int account[2];
char _lock[2];

int lock (long tid) {
  _lock[tid] = 1;

  while (_lock[NUM_THREADS - 1 - tid]) {
    _lock[tid] = 0;
    sleep (1);
    _lock[tid] = 1;
  }
  return 0;
}

int unlock (long tid) {
  _lock[tid] = 0;
  return 0;
}
```

```c
// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i;
  int amount = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    amount = (int)(((double) rand () /
        (RAND_MAX - 1)) * 100);

    // try to enter the critical section
    lock (tid);

    // critical section
    account[tid]                -= amount;
    account[NUM_THREADS-1-tid] += amount;
    // return from critical section
    unlock (tid);
  }
  pthread_exit (NULL);
}
```

# Twofold Lock with Mutual Precedence

- The critical section is protected!

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0 !
Hello World! It's me, thread #1 !
 account_0: 100
 account_1: 100

In main: creating thread 0
Hello World! It's me, thread #0 !
In main: creating thread 1
Hello World! It's me, thread #1 !
 account_0: 100
 account_1: 100
```

# Requirements for Solutions to protect the Critical Section

- The solution has to protect the critical section reliably by mutual exclusion. ✓
- The solution should be used in higher level programming languages. ✓
  - Thus, the solution is usable on different architectures providing portability for the program using it.
- The solution must not lead to a deadlock. ✓

# Correctness

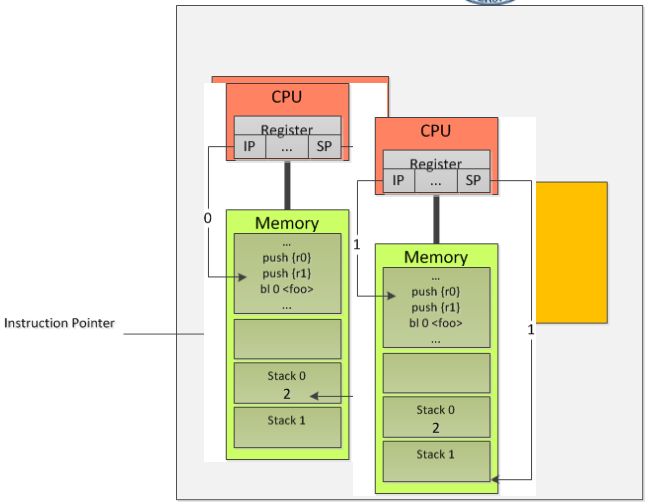
- Correctness is ensured based on
  - Correct implementation of commands and functions
    - compiler/interpreter, HW
  - Correct execution of the set of commands and instructions
    - Sequential processing of the instructions of the critical section by mutual exclusion due to locks
    - Programming model and machine model (execution model) correspond to each other
  - Check with
    - Hoare logic (calculus)
    - Simulation
    - Testing

# Performance from the User Perspective

- *Thread* switching comes with an overhead.
- But, if the program is designed to use more resources than CPU (and memory) this usage of the different resources may be performed in parallel (parts of the program are executed concurrently).
- Problem has to be split into reasonable pieces (e.g. via divide and conquer approaches).
- *The data representing the problem can be shared between all threads by using the same address space.*
- The usage of all of the resources may lead to a reduction of idle time as well as to a reduction of the response time of the entire program.
- The uniform progress of the *threads* of the program (and all other processes) is ensured by the operating system. It needs to correspond to goals and possibilities of the operating system (scheduling).

# Requirements for Programs

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




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

Concepts of Non-sequential and Distributed Programming

# NEXT LECTURE

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

# Parallelization

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2021)