Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV
Parallelization

Summer Term 2023 | 08.05.2023

Barry Linnert

Objectives of Today‘s Lecture

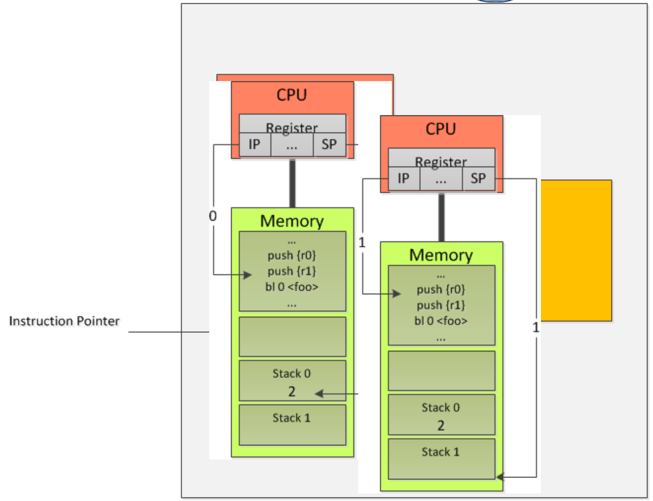
- more (about) Locks
- Parallelization
  - Machine model / Execution model
  - Hardware support
- POSIX Mutex
- Minimization of the critical section

Concepts of Non-sequential and Distributed Programming

PARALLELIZATION

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 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

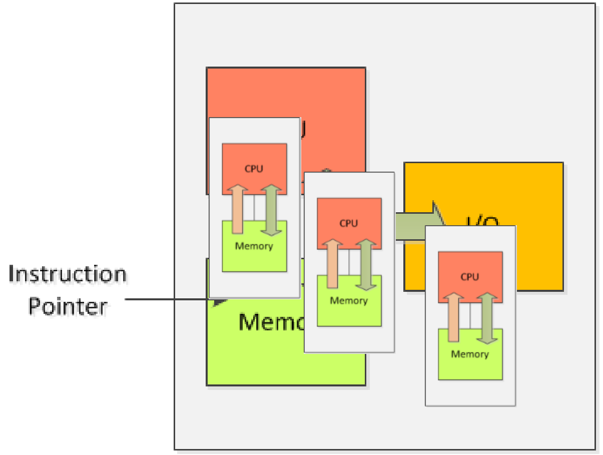
Requirements for Solutions to protect the Critical Section

- 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. ✓

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

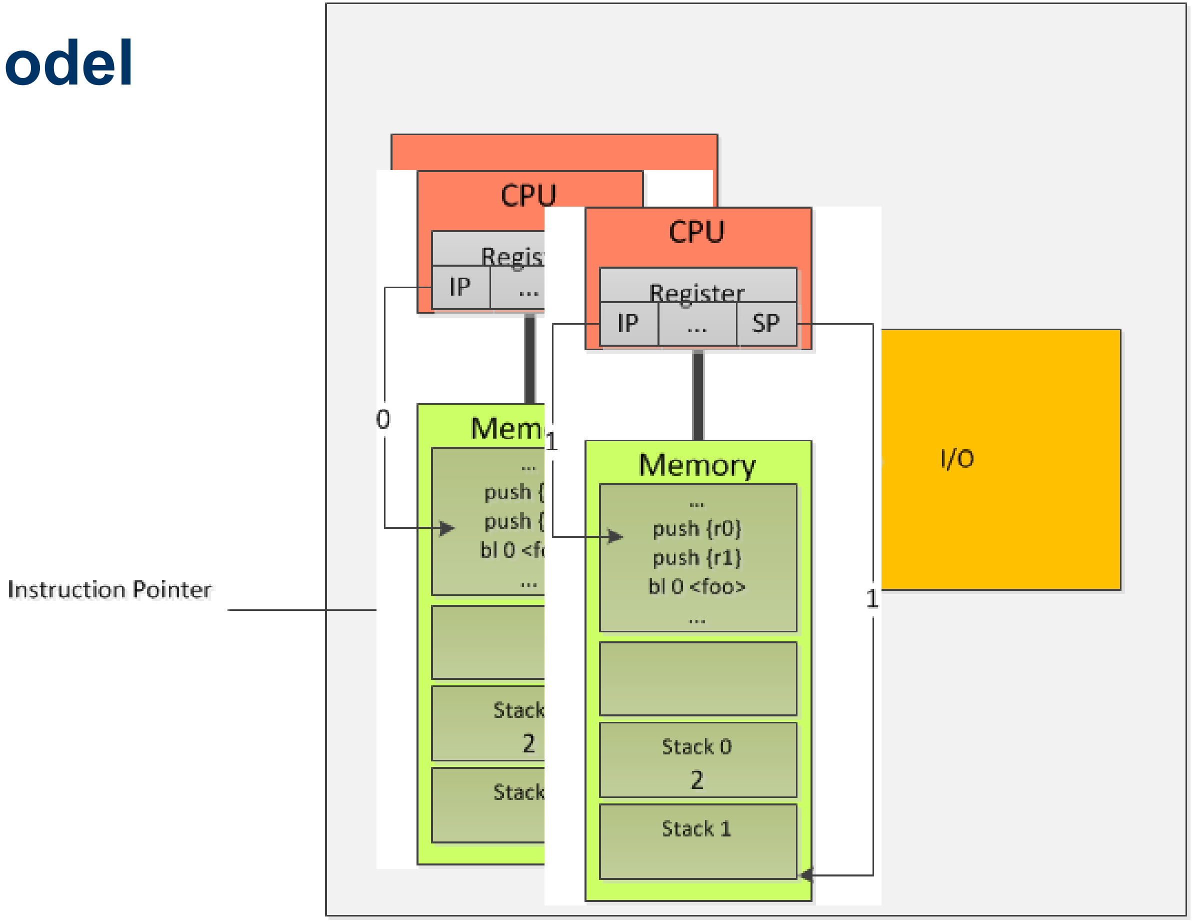


- Perspective of the user
  - the fastest possible processing of your own program
  - short response time

Performance from the User Perspective

- 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).

Machine model



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.

```
// 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);
}
```

```
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);
}
```

```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
  ...
```

```c
// 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);
}
```

Twofold Lock with Mutual Precedence

- The twofold lock with mutual precedence approach protects the critical section reliably.

- It prevents deadlocks as the own lock is released in order to give the other thread a chance to get the lock before the lock is acquired by the requesting thread.
- 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]) {
    _lock[tid] = 0;
    sleep (1);
    _lock[tid] = 1;
  }
  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);
}
```

Twofold Lock with Mutual Precedence

- The approach of the twofold lock with mutual precedence meet the requirements, but comes with some additional limitations or problems:

- The requesting thread sleeps a little time to give the other thread the chance to acquire the lock (after returning from the critical section). But the sleeping increases the response time.

- Mutual precedence can lead to a loop: "After you!" – „After *you*!“
  - This is called a livelock and can lead to starvation as no thread will get the lock even if they could.
  - The problem can be reduced in case the sleeping lasts longer to give the other thread more time to get the lock.

```
// 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;
}
```

```
// 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);
}
```

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. ✓
- The solution should provide low overhead. ?
  - There is no (excessive) waiting in order to enter the critical section.

Twofold Lock with Priorities

- To overcome the problem of the livelock („After you!“ – „After *you*!“) an order to access the lock in case of a conflict can be implemented.

- The order is represented by priorities.

- The priority is determined based on the sequence of accessing the lock.
  - This resembles the queueing approach where the last one requesting the resource is last one in the queue.
  - The thread with the lowest priority is the first in queue and is the first to get the lock.

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

#define NUM_THREADS   2
int account[2];
int prio[2];  // init with 0 at main()

int lock (long tid) {

  prio[tid] = prio[NUM_THREADS-1-tid]+ 1;

  while ((prio[NUM_THREADS-1-tid] != 0)&&
        (prio[NUM_THREADS-1-tid] <
                            prio[tid]))
    ;

  return 0;
}

int unlock (long tid) {
  prio[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 Priorities

- Checking and setting of the priority is a critical section by itself.

- 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: 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: 100
 account_1: 100

...
 account_0: 1568
 account_1: -2317
```

Twofold Lock with Priorities and Reservation

- The easiest way to overcome the problems with the twofold lock with priorities approach is to use a fixed priority, so the check and set of the priority does not take place.
- But, this could lead to starvation of the thread with lower priority (higher value of the priority variable).
- To reduce the effect of the fixed priority, the dynamic determination of the priority can be used as introduced by the twofold lock with priorities, but has to be protected.
- Thus, an additional variable is introduced announcing the attempt by a thread to get in line of access to the lock. This is implemented by the variable `interested`.

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

#define NUM_THREADS   2
int account[2];
int prio[2]; // init with 0 at main()
char interested[2]; // init with 0 at main()

int lock (long tid) {
  interested[tid] = 1;
  prio[tid] = prio[NUM_THREADS-1-tid]+ 1;
  interested[tid] = 0;
  while (interested[NUM_THREADS-1-tid])
    ;
  while ((prio[NUM_THREADS-1-tid] != 0)&&((prio[NUM_THREADS-1-tid] < prio[tid])||
           ((prio[NUM_THREADS-1-tid] == prio[tid])&&((NUM_THREADS-1-tid) < tid))))
    ;
  return 0;
}
int unlock (long tid) {
  prio[tid] = 0;
  return 0;
}
```

Twofold Lock with Priorities and Reservation

- The critical section is protected!
- But, due to the prioritization based on the thread IDs there is no fair access to the critical section in case of a conflict.

```
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: 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. ✓
- The solution should provide low overhead. ?
  - There is no (excessive) waiting in order to enter the 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. ✓
- The solution should provide low overhead. ?
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ?

Lock with Alternate Access

- The simple solution to get a fair access to the critical section is to have an alternate access.

- Every thread gets access after the other one is leaving the critical section.

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

int favoured = 1;

int lock (long tid) {




  while (favoured == (NUM_THREADS-1-tid))
    ;




  return 0;
}
int unlock (long tid) {
  favoured = NUM_THREADS - 1 - 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);
}
```

Lock with Alternate Access

- The critical section is protected!
- In case of unbalanced usage of the critical section the requesting thread has to busy wait potentially a long time.
  - Thus, the requirement of low overhead is not achieved.
- If one thread terminates, the other thread will not get access to the critical section anymore.

```
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: 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. ✓
- The solution should provide low overhead. ?
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ?

Twofold Lock with Mutual Access – Dekker

- The turn-taking approach of the lock with alternate access algorithm can be combined with the “classical” locking approach.
- So in case one thread is alive and is using the critical section the other one is favored to get the lock to enter the critical section.
- If only one thread is requesting the critical section, the lock variable is used to protect the critical section independent of the decision which thread is favored (as the other thread doesn’t require access to the critical section).

- One of the first correct algorithms for mutual exclusion introduced by Th. J. Dekker in 1962 or 1963.

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

int favoured = 1;

int lock (long tid) {


    if (favoured != tid) {

      while (favoured != tid)
        ;

    }

  return 0;
}
int unlock (long tid) {
  favoured = NUM_THREADS - 1 - 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);
}
```

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS   2
int account[2];
char _lock[2]; // init with 0 at main()
int favoured = 1;

int lock (long tid) {
  _lock[tid] = 1;
  while (_lock[NUM_THREADS - 1 - tid]) {
    if (favoured != tid) {
      _lock[tid] = 0;
      while (favoured != tid)
        ;
      _lock[tid] = 1;
    }
  }
  return 0;
}
int unlock (long tid) {
  favoured = NUM_THREADS - 1 - 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);
}
```

Twofold Lock with Mutual Access – Dekker

- The critical section is protected!
- A fair access in case of concurrent requests is provided by mutual prioritization.
- There is no starvation even in case one of the threads is terminating.
- The **twofold lock with mutual access (Dekker) approach meets** 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: 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. ✓
- The solution should provide low overhead. ✓
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ✓

Twofold Lock with Mutual Access – Peterson

- A more elegant version of the approach was introduced by Gary L. Peterson in 1981.

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS   2
int account[2];
char _lock[2]; // init with 0 at main()
int favoured = 1;

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

  while ((_lock[NUM_THREADS-1-tid])&&
        (favoured == (NUM_THREADS-1-tid)))
    ;

  return 0;
}

int unlock (long tid) {
  favoured = NUM_THREADS - 1 - 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);
}
```

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. ✓
- The solution should provide low overhead. ✓
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ✓

Multiple Locks with Mutual Access – Peterson

- The algorithm using twofold locks with mutual access by Peterson can be extended to be used by more than two threads.
- The number of threads requesting access to the critical sections has to be known in advance.

- Approach implements the following idea:
  - A thread with the thread ID tid enters a waiting room as the last thread.
  - As long as thread tid is the last thread in the waiting room **and** there is a thread waiting in a waiting room further along the way to get access to the critical section, thread tid waits (active).
  - If another thread enters the waiting room or there is no thread waiting in a waiting room closer to the critical section, the thread tid moves to the next waiting room until it reaches the last waiting room and is allowed to enter the critical section.

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

#define NUM_THREADS   4

int account[NUM_THREADS];
int level[NUM_THREADS]; // init with 0
int last[NUM_THREADS]; // init with 0



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

```
int lock (long tid) {
  int i, j, loop;

  for (i = 1; i < NUM_THREADS; i++) {
    level[tid] = i;
    last[i] = tid;








  }
  return 0;
}

// accounting
...
```

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

#define NUM_THREADS    4

int account[NUM_THREADS];
int level[NUM_THREADS]; // init with 0
int last[NUM_THREADS]; // init with 0




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

```
int lock (long tid) {
  int i, j, loop;

  for (i = 1; i < NUM_THREADS; i++) {
    level[tid] = i;
    last[i] = tid;
    loop = 1;
    while ((loop)&&(last[i] == tid)) {
      j = 0;







    }
  }
  return 0;
}

// accounting
...
```

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

#define NUM_THREADS    4

int account[NUM_THREADS];
int level[NUM_THREADS]; // init with 0
int last[NUM_THREADS]; // init with 0




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

```
int lock (long tid) {
  int i, j, loop;

  for (i = 1; i < NUM_THREADS; i++) {
    level[tid] = i;
    last[i] = tid;
    loop = 1;
    while ((loop)&&(last[i] == tid)) {
      j = 0;
      loop = 0;
      while ((j < NUM_THREADS)&&
             ((level[j] < i)||(j == tid)))
        j++;


    }
  }
  return 0;
}

// accounting
...
```

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

#define NUM_THREADS   4

int account[NUM_THREADS];
int level[NUM_THREADS]; // init with 0
int last[NUM_THREADS]; // init with 0




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

```c
int lock (long tid) {
  int i, j, loop;

  for (i = 1; i < NUM_THREADS; i++) {
    level[tid] = i;
    last[i] = tid;
    loop = 1;
    while ((loop)&&(last[i] == tid)) {
      j = 0;
      loop = 0;
      while ((j < NUM_THREADS)&&
            ((level[j] < i)||(j == tid)))
        j++;
      if (j < NUM_THREADS)
        loop = 1;
    }
  }
  return 0;
}

// accounting
. . .
```

Multiple Locks with Mutual Access – Peterson

- The critical section is protected!
- Multiple threads are handled.
- The **multiple locks with mutual access (Peterson) approach meets** 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 !
In main: creating thread 2
Hello World! It's me, thread #1 !
In main: creating thread 3
Hello World! It's me, thread #2 !
Hello World! It's me, thread #3 !
 account_0: -5011
 account_1: 4241
 account_2: 6877
 account_3: -5707

...
```

Multiple Locks with Mutual Access – Lamport

- A similar algorithm was introduced by Leslie Lamport in 1974.
- As the idea behind the approach envisions a bakery with tickets for the customers, the approach also known as **Lamport‘s bakery**.

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0







int unlock (long tid) {

  return 0;
}
```

```
int lock (long tid) {


















  return 0;
}
```

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0
```

```
int lock (long tid) {
  int i, max = 0;
  enter[tid] = 1;
  for (i = 0; i < NUM_THREADS; i++) {
    if (max < tickets[i])
      max = tickets[i];
  }









  return 0;
}
```

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

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0
```

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

```
int lock (long tid) {
  int i, max = 0;
  enter[tid] = 1;
  for (i = 0; i < NUM_THREADS; i++) {
    if (max < tickets[i])
      max = tickets[i];
  }
  tickets[tid] = max + 1;
  enter[tid] = 0;
  for (i = 0; i < NUM_THREADS; i++) {




  }
  return 0;
}
```

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0




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

```
int lock (long tid) {
  int i, max = 0;
  enter[tid] = 1;
  for (i = 0; i < NUM_THREADS; i++) {
    if (max < tickets[i])
      max = tickets[i];
  }
  tickets[tid] = max + 1;
  enter[tid] = 0;
  for (i = 0; i < NUM_THREADS; i++) {
    if (i != tid) {



    }
  }
  return 0;
}
```

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0
```

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

```
int lock (long tid) {
  int i, max = 0;
  enter[tid] = 1;
  for (i = 0; i < NUM_THREADS; i++) {
    if (max < tickets[i])
      max = tickets[i];
  }
  tickets[tid] = max + 1;
  enter[tid] = 0;
  for (i = 0; i < NUM_THREADS; i++) {
    if (i != tid) {
      while (enter[i])
        ;






    }
  }
  return 0;
}
```

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

#define NUM_THREADS   6

int account[NUM_THREADS]; // init with 0
char enter[NUM_THREADS]; // init with 0
int tickets[NUM_THREADS]; // init with 0




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

```c
int lock (long tid) {
  int i, max = 0;
  enter[tid] = 1;
  for (i = 0; i < NUM_THREADS; i++) {
    if (max < tickets[i])
      max = tickets[i];
  }
  tickets[tid] = max + 1;
  enter[tid] = 0;
  for (i = 0; i < NUM_THREADS; i++) {
    if (i != tid) {
      while (enter[i])
        ;
      while ((tickets[i] != 0)&&
        ((tickets[tid] > tickets[i])||
        ((tickets[tid] == tickets[i])&&
        (tid > i))))
        ;
    }
  }
  return 0;
}
```

Multiple Locks with Mutual Access – Lamport

- The critical section is protected!
- Multiple threads are handled.
- The **multiple locks with mutual access (Lamport) approach meets** all requirements for a solution to protect a critical section.

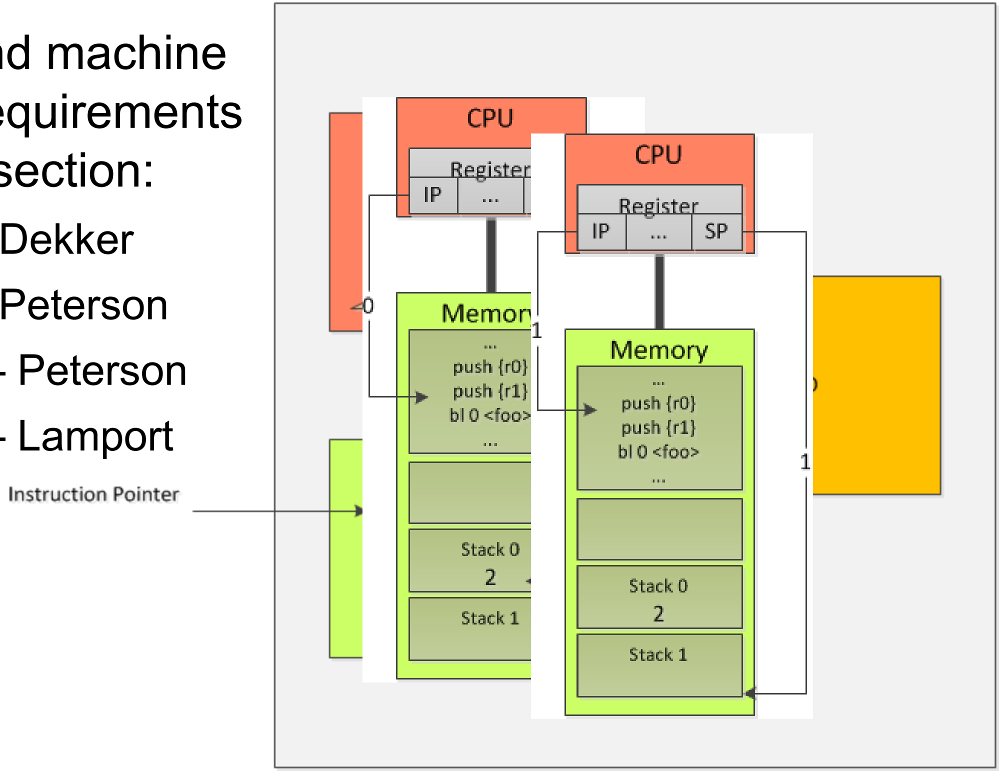
```
In main: creating thread 0
Hello World! It's me, thread #0 !
In main: creating thread 1
Hello World! It's me, thread #1 !
In main: creating thread 2
Hello World! It's me, thread #2 !
In main: creating thread 3
Hello World! It's me, thread #3 !
In main: creating thread 4
Hello World! It's me, thread #4 !
In main: creating thread 5
Hello World! It's me, thread #5 !
 account_0: -37603
 account_1: -24997
 account_2: 4667
 account_3: 81509
 account_4: 14189
 account_5: -37165
```

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. ✓
- The solution should provide low overhead. ✓
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ✓

Machine model

- Based on the current execution and machine model the approaches meet the requirements for solutions to protect the critical section:
  - Twofold Lock with Mutual Access – Dekker
  - Twofold Lock with Mutual Access – Peterson
  - Multiple Locks with Mutual Access – Peterson
  - Multiple Locks with Mutual Access – Lamport

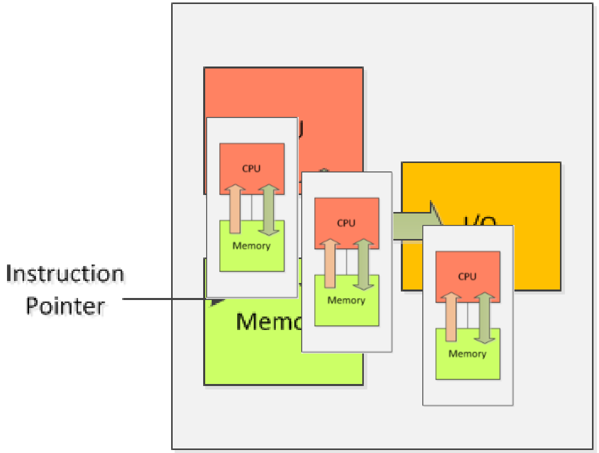


Performance

- 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



Concepts of Non-sequential and Distributed Programming

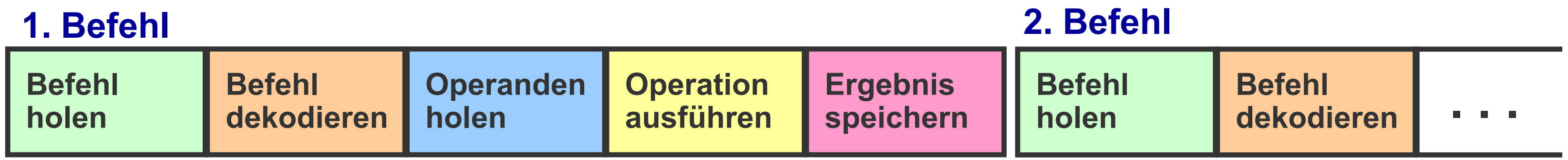
PARALLELIZATION

Parallelization

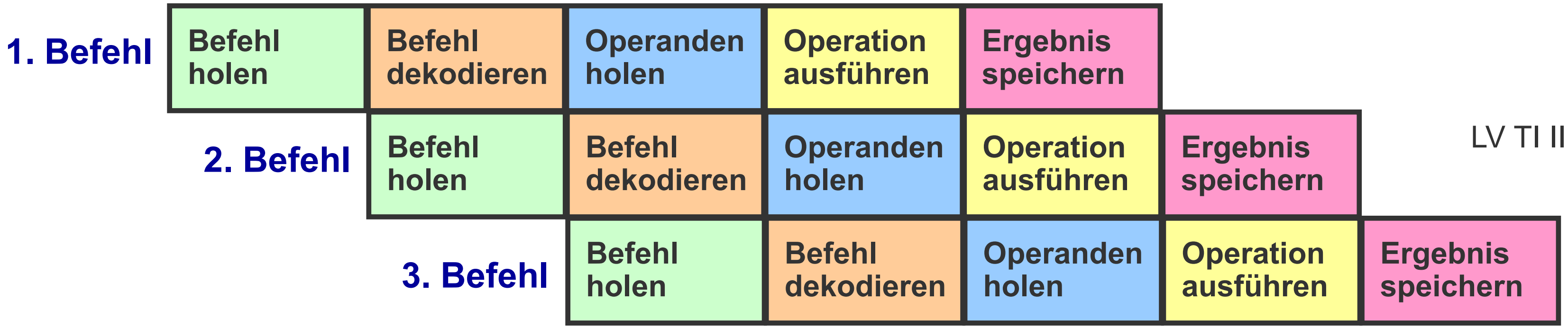
- Current computer architectures differ from the machine model used to discuss the different approaches to protect the critical section.
- Is the machine model (which is used until now) sufficient to cover the real behavior of real machines?

Pipelining

Sequentielle Ausführung:

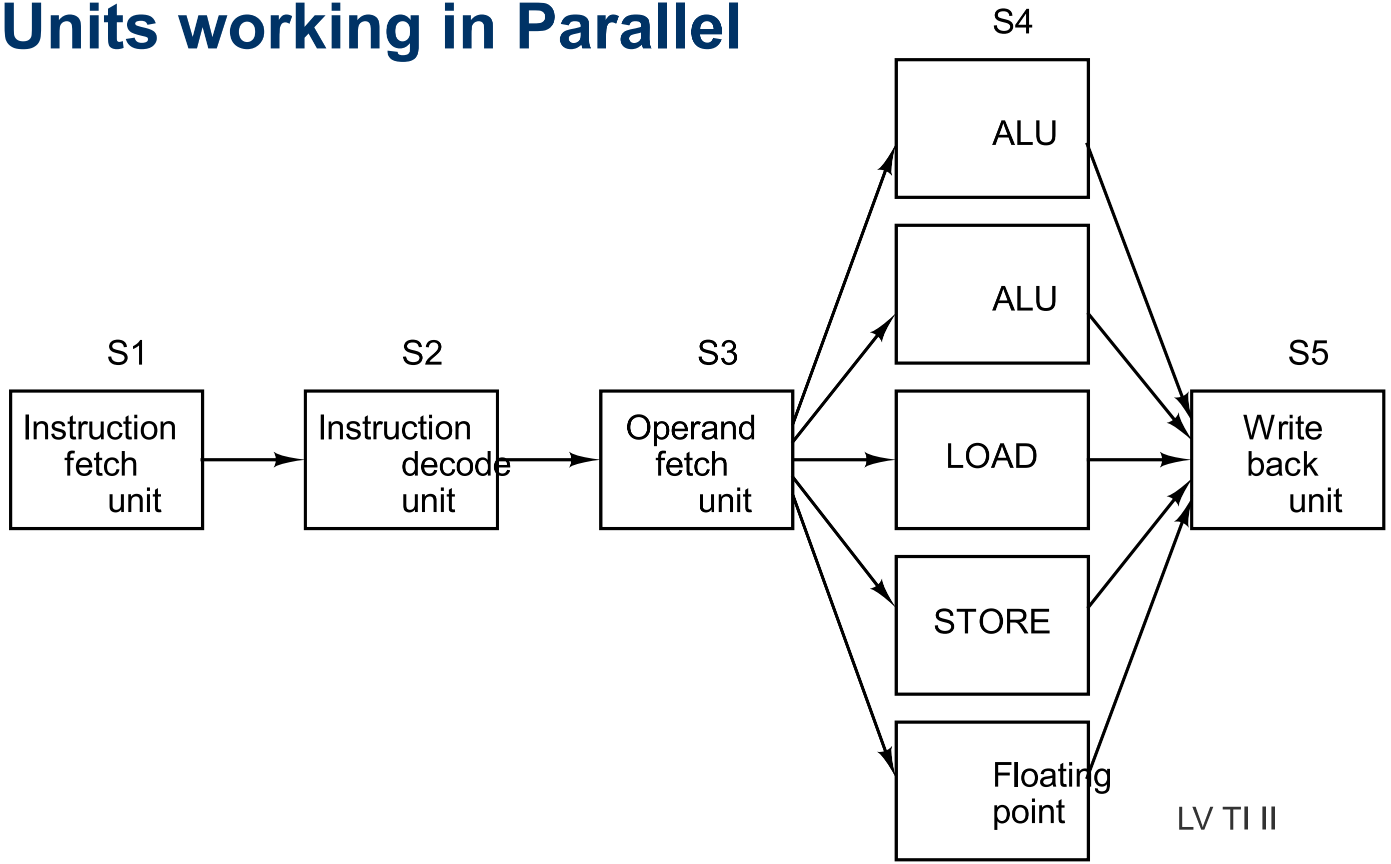


Pipelining:



LV TI II

Execution Units working in Parallel



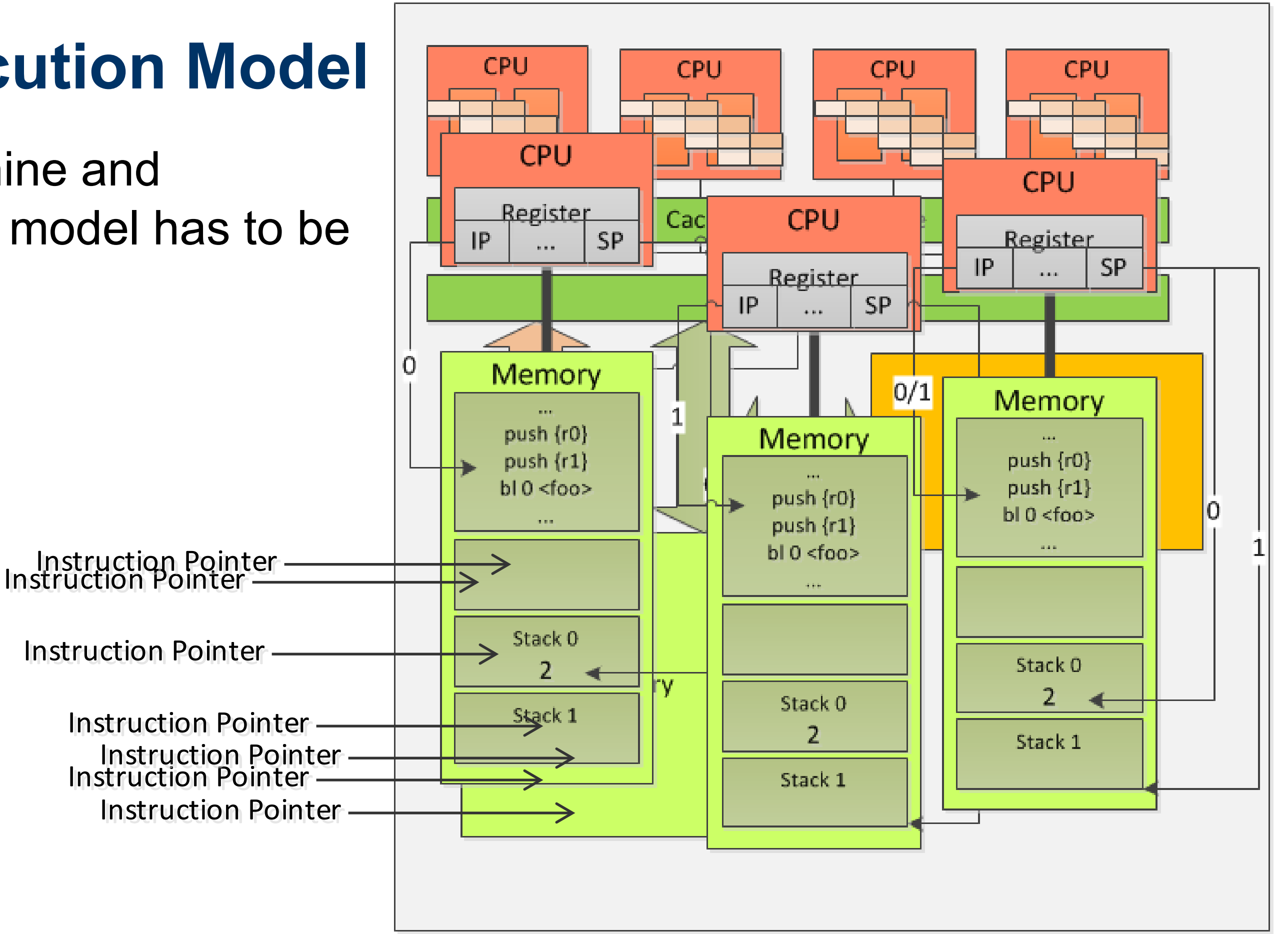
LV TI II

Optimizations of current Compilers and Processors

- There are lots of optimizations implemented in current compilers and processors:
  - Out of order execution
    - Reorganize operations to use as many processing units as possible.
  - Speculative execution
    - Execution of all operation sequences after a conditional jump.
    - The operations that do not have to/should not be executed due to the condition are already processed.
  - Cache hierarchies
    - During a write operation, data is first written to the local cache and is not available there to other processing units of different processor/cores.
    - Writing back data to the shared memory takes place with a delay (Write-back Cache).

Machine and Execution Model

- The machine and execution model has to be adapted:



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

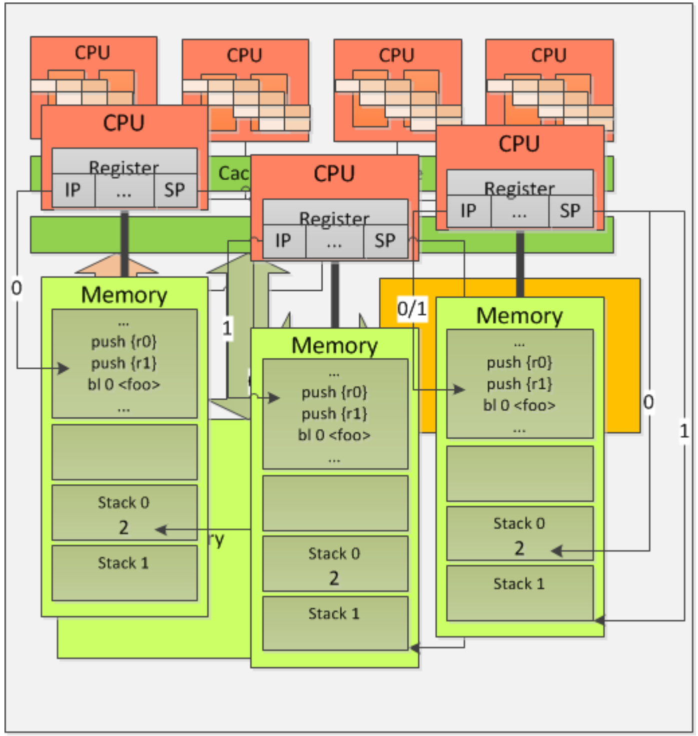
Requirements for Solutions to protect the Critical Section

- 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.
- The solution should provide low overhead.
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair.

With the adaption of the machine and execution model all of the solutions to protect the critical section have to be checked if their still meet the requirements!

Requirements for Solutions to protect the Critical Section II

- Based on the **adapted execution and machine model** the approaches **don't meet** the requirements for solutions to protect the critical section:
  - Twofold Lock with Mutual Access – Dekker
  - Twofold Lock with Mutual Access – Peterson
  - Multiple Locks with Mutual Access – Peterson
  - Multiple Locks with Mutual Access – Lamport



Hardware Support to Protect the Critical Section

- The crucial part of protection of the critical section using locks is to simulate the behavior of an atomic operation for the check and set of the lock variable.
- Most hardware architectures provide support for this task by introducing atomic operations combining the tasks of checking and setting a value to a memory address:
  - Test-and-Set
    - XCHG – x86
  - Fetch-and-Add
    - XADD – x86
  - Compare-and-Swap
    - CMPXCHG – x86
  - Hardware (Cache / Memory) Monitors

Hardware Support to Protect the Critical Section II

- The hardware support comes with some drawbacks such as:
  - To execute the instructions as atomic operations, the pipeline has to be emptied. Thus, the benefit of higher performance is reduced.
  - Different hardware architectures provide different approaches and instructions supporting the atomic instructions. The support can be complex.

- Example: ARM Cache Monitors for Locks
  - http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf

```
locked     EQU    1
unlocked  EQU   0
; lock_mutex
EXPORT lock_mutex
lock_mutex PROC
          LDR     r1, =locked
1         LDREX   r2,    [r0]
          CMP     r2, r1          ; Test if mutex is locked or unlocked
          BEQ     %f2             ; If locked - wait for it to be released, from 2
          STREXNE r2, r1, [r0]    ; Not locked, attempt to lock it
          CMPNE   r2, #1          ; Check if Store-Exclusive failed
          BEQ     %b1             ; Failed - retry from 1
; Lock acquired
          DMB                     ; Required before accessing protected resource
          BX      lr
2    ; Take appropriate action while waiting for mutex to become unlocked
          WAIT_FOR_UPDATE
          B       %b1             ; Retry from 1
          ENDP
```

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.
- The solution should provide low overhead.
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair.

There is some part of the (new) execution and machine model that was considered only briefly and may help to overcome the problems with the current execution and machine model – the **Operating System**!

Operating System Support for Mutual Exclusion

- Operating systems provide support for mutual exclusion to protect a critical section.
- The operating system usually implements the protection using hardware architecture based instruction, such as test-and-set.
- As the operating system is aware of the state of the process or thread it can reduce the overhead by blocking the requesting process or thread rather than implementing busy waiting. The process or thread is deblocked if the critical section is released and the lock is unset.

- The POSIX standard provides the POSIX Mutex.

POSIX Mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex,
       const pthread_mutexattr_t *mutexattr);
```

Initialization of the lock for access to the critical section,

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Request for the lock to get access to the critical section,

Thread sleeps as long as the lock is not available

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

like `pthread_mutex_lock()` but without sleeping

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Release of the critical section and unset of the lock

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS     6
// shared data
int account[NUM_THREADS];



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





    account[tid]                 -= amount;
    account[acc_nr]              += amount;



  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);
}
```

```c
// simple accounting with pthreads
#include ...
#define NUM_THREADS     6
// shared data
int account[NUM_THREADS];


// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount, acc_nr, error = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    // calc acc_nr and amount


    account[tid]              -= amount;
    account[acc_nr]           += 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     6
// shared data
int account[NUM_THREADS];
pthread_mutex_t lock;

// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount, acc_nr, error = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    // calc acc_nr and amount
    // try to enter the critical section
    _error = pthread_mutex_lock(&lock);
    account[tid]                -= amount;
    account[acc_nr]             += amount;
    _error = pthread_mutex_unlock(&lock);
  }
  pthread_exit (NULL);
}
```

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

  // init lock
  pthread_mutex_init(&lock, NULL);

  // init data

  // creating threads

  // joining threads

  // output results

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

POSIX mutex

- The critical section is protected!
- Multiple threads are handled.
- The **POSIX mutex meets** all requirements for a solution to protect a critical section.

```
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 !
Hello World! It's me, thread #2 !
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3 !
Hello World! It's me, thread #4 !
In main: creating thread 5
Hello World! It's me, thread #5 !
 account_0: -82978
 account_1: 3782
 account_2: 66872
 account_3: 13514
 account_4: 26006
 account_5: -26596
...
```

Java – synchronized

- Other programming languages provide interfaces to solutions or solutions by their own to protect critical sections.

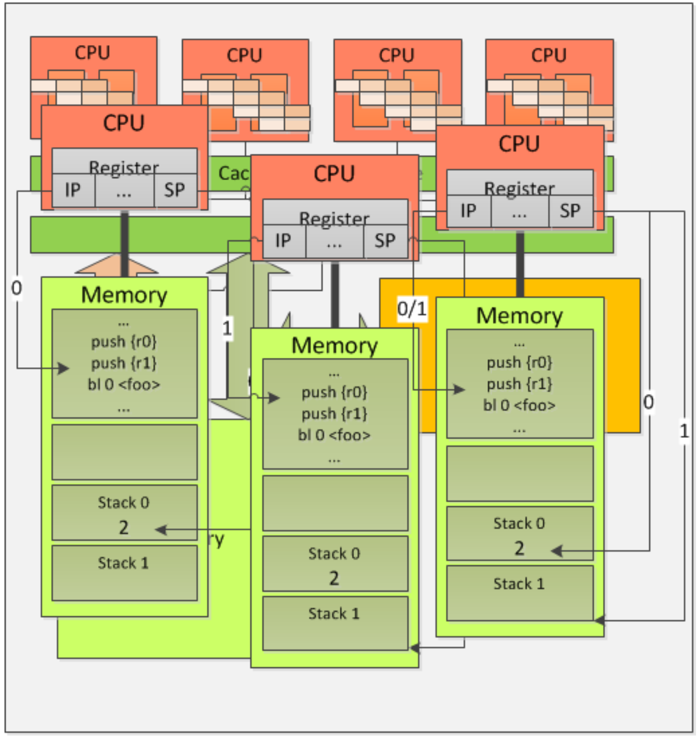
- Example: Java

```
synchronized(MyKlassenName.class)
{
        // statements
}

public static synchronized void method()
{
        // statements
}
```

Performance Aspects

- Sequential processing always leads to reduction of utilization of the processing units – at least from user's perspective.
- Therefore, it is always recommended to **minimize the critical section**!



```
// simple accounting with pthreads
#include ...
#define NUM_THREADS     6
// shared data
int account[NUM_THREADS];
pthread_mutex_t lock;

// accounting
void *bank_action (void *threadid)
{
  long tid;
  int i, amount, acc_nr, error = 0;
  tid = (long) threadid;
  for (i = 0; i < 300000; i++) {
    // calc acc_nr and amount
    // try to enter the critical section
    _error = pthread_mutex_lock(&lock);
    account[tid]              -= amount;
    account[acc_nr]           += amount;
    _error = pthread_mutex_unlock(&lock);
  }
  pthread_exit (NULL);
}
```

- Until now the critical section consists of the two operations:

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

Minimization of the Critical Section

- As the value of `amount` remains unchanged the critical section consists actually of the operation on the single date :

```
account[tid] = account[tid] - amount;
```

  - The critical sections consists of reading `account[tid]` and writing the new value after subtraction of `amount` to `account[tid]`.
  - Thus, only the one operation has to be protected.

```
// simple accounting with pthreads
#include ...
#define NUM_THREADS     6
// shared data
int account[NUM_THREADS];
pthread_mutex_t lock;

// accounting
void *bank_action (void *threadid)
{
  ...
  for (i = 0; i < 300000; i++) {
    // calc acc_nr and amount
    _error = pthread_mutex_lock(&lock);
    account[tid]    -= amount;
    _error = pthread_mutex_unlock(&lock);

    _error = pthread_mutex_lock(&lock);
    account[acc_nr] += amount;
    _error = pthread_mutex_unlock(&lock);
  }
  pthread_exit (NULL);
}
```

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. ✓
- The solution should provide low overhead. ✓
  - There is no (excessive) waiting in order to enter the critical section.
- The access to the critical section should be fair. ✓

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 the operations of the critical section by lock variables (lock/mutex) using the operating system and hardware
    - 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).

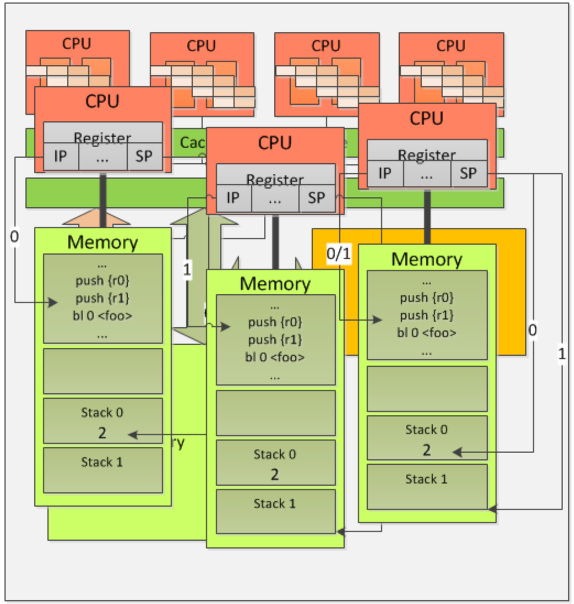
Requirements for Programs

- 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

Modeling with Petri nets

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