Operating Systems and Multiprogramming:

Multiprogramming, POSIX Threads

Jost Berthold
berthold@diku.dk
February 2014
Department of Computer Science
University of Copenhagen

Multithreading and Multiprocessing

- In the real world, many things happen at the same time (concurrently)
- Likewise, a system can be modelled by many concurrent actions.

Multithreading and Multiprocessing

- In the real world, many things happen at the same time (concurrently)
- Likewise, a system can be modelled by many concurrent actions.
- Modern hardware supports simultaneous execution (multicore processors)
- Concurrent programming is rightfully considered very complicated

Contents and Goals of this Part

Contents and Goals of this Part

1. What is Multi-Threaded Programming?
   - Motivation and Terminology
2. Threads and Context Switch
   - User Threads and Kernel Threads
   - Context Switching
3. C with POSIX Threads (Pthreads)
   - Our First Multi-Threaded Programs
   - Mutual Exclusion with Pthread Locks
   - Conditional Synchronisation: Pthread Condition Variables

Goals:

- Explain use of concurrency and threads in operating systems
- Understand how thread context switches are implemented
- Use PThreads for concurrent programming

Multi(-threaded) Programming / Multiprocessing

Multiprogramming (and Multi-Threaded Programming)

The term multiprogramming generally means to share hardware resources of a computer system among several executing units. More specifically, multi-threaded programming describes the coordinated use of several threads of execution in one OS process.

Multi(-threaded) Programming / Multiprocessing

Multiprogramming (and Multi-Threaded Programming)

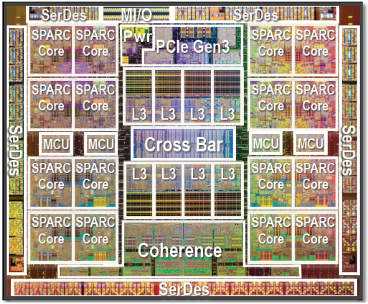
The term multiprogramming generally means to share hardware resources of a computer system among several executing units.
More specifically, multi-threaded programming describes the coordinated use of several threads of execution in one OS process.

Multiprocessing

The term multiprocessing (most often) describes execution of computer programs on several CPUs within a single computer system.

Flynn's taxonomy: (1968)	Single Instruction	Multiple Instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Multiprocessing hardware



Oracle Sparc T5

16 cores, 128 HW threads

MIMD, Multicore

NVidia Geforce GTX Titan

2688 "Cuda cores"

SIMD, Accelerator

Our focus here (OSM): concurrent execution on multicore

Concurrent and parallel programming

Concurrent programming:

- The CPU can be efficiently used during waiting periods (for memory or disk access).
- Even a busy system remains responsive.
- The system is easier to model in separate units than in one block that does everything

Concurrent and parallel programming

Concurrent programming:

- The CPU can be efficiently used during waiting periods (for memory or disk access).
- Even a busy system remains responsive.
- The system is easier to model in separate units than in one block that does everything
  (even on sequential hardware, time-sharing).

Parallel programming:

- Computations can finish faster if parallel hardware is used.
- Concurrent threads can compute partial results simultaneously.

Concurrent and parallel programming

Concurrent programming:

- The CPU can be efficiently used during waiting periods (for memory or disk access).
- Even a busy system remains responsive.
- The system is easier to model in separate units than in one block that does everything
(even on sequential hardware, time-sharing).

Parallel programming:

- Computations can finish faster if parallel hardware is used.
- Concurrent threads can compute partial results simultaneously.

Concurrency and Parallelism often confused, to be distinguished!

Concurrency in an operating system

System (kernel) concurrency

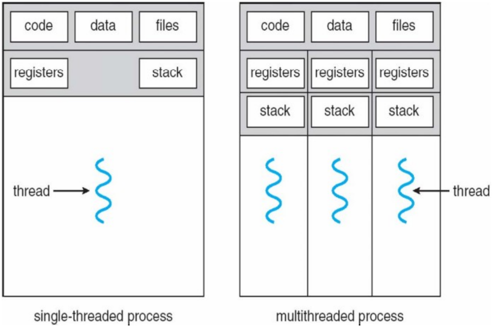
- OS management tasks can run in the background;
- OS manages several applications (user processes) to share hardware (run concurrently);
- The entire system remains responsive.

User-level concurrency

- The OS provides an API to run sub-processes concurrently;
- A user process can use several threads...
  - to do a task more quickly (parallelism)
  - or to stay responsive (GUI programs, concurrency);
- Applications can be structured using concurrent threads.

Multiple user processes vs. multiple threads

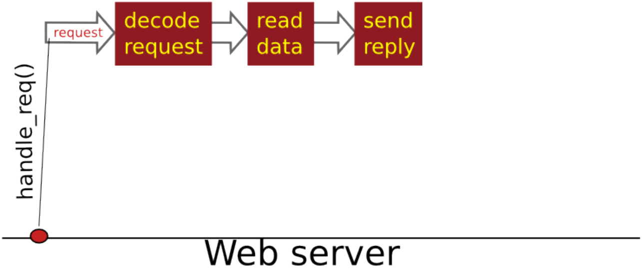
- Every thread in a process has its own register and stack.
- Data, code and files are shared between all threads in a process.



[Silberschatz et al., 2013]

Example: A webserver

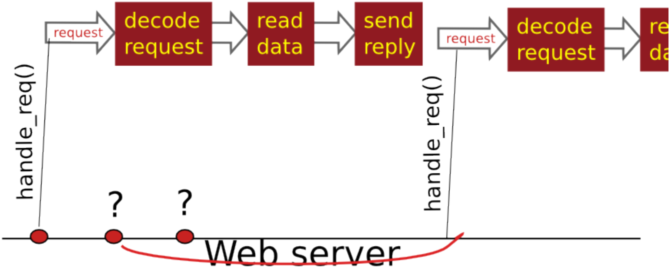
- receive HTTP requests
- decode request
- read data from disk
- respond via network



- Sequential processing

Example: A webserver

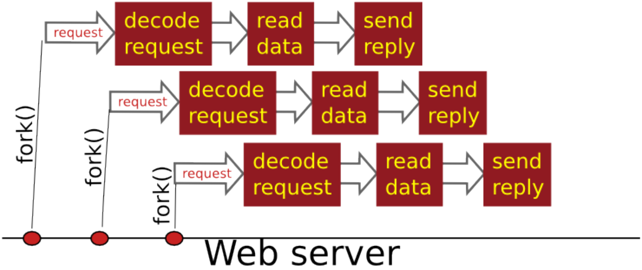
- receive HTTP requests
- decode request
- read data from disk
- respond via network



- Sequential processing not acceptable

Example: A webserver

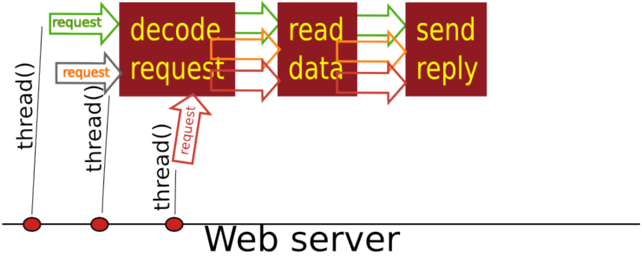
- receive HTTP requests
- decode request
- read data from disk
- respond via network



- Sequential processing not acceptable
- Using `fork`: process overhead, no shared cache

Example: A webserver

- receive HTTP requests
- decode request
- read data from disk
- respond via network



- Sequential processing not acceptable
- Using `fork`: process overhead, no shared cache
- Using threads: hides I/O latency, enables caching
  In practice: thread pool with limited threads

Limited Multi-Threading: Thread Pool

Want to limit number of threads

- Memory consumption grows with threads
- Each thread incurs administration cost in the system

Limited Multi-Threading: Thread Pool

Want to limit number of threads

- Memory consumption grows with threads
- Each thread incurs administration cost in the system

Create and reuse a number of worker threads

- Unnecessary ("idle") threads can be sent to sleep,
- saves thread creation (administration) cost

Number of threads can be adjusted to deal with changing load

- Create additional threads when load increases.
- Terminate threads when too many existing threads are idle.

Outline

User threads and kernel threads

Thread support can be realised at two levels:

User Threads

- Support for threads in user processes/programs
- managed by a runtime system, above the kernel;
- Threads share memory in one user process.

User threads and kernel threads

Thread support can be realised at two levels:

User Threads

- Support for threads in user processes/programs
- managed by a runtime system, above the kernel;
- Threads share memory in one user process.

Kernel Threads

- Support for running multiple threads in the kernel
- managed by the kernel itself.
- Virtually all modern OS support kernel threads.
- All kernel threads have access to (all) kernel memory.

User threads and kernel threads

Thread support can be realised at two levels:

User Threads

- Support for threads in user processes/programs
- managed by a runtime system, above the kernel;
- Threads share memory in one user process.

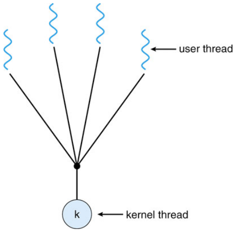
Kernel Threads

- Support for running multiple threads in the kernel
- managed by the kernel itself.
- Virtually all modern OS support kernel threads.
- All kernel threads have access to (all) kernel memory.

Several options for relating user and kernel threads.

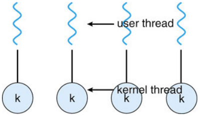
Mapping User-Threads to Kernel Threads

Many-to-One

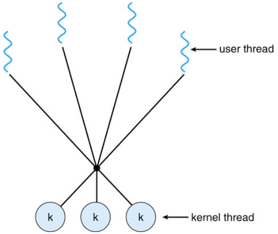


- Interpreters (CPython, older Java VMs)
- lightweight threads (small overhead)

One-to-One

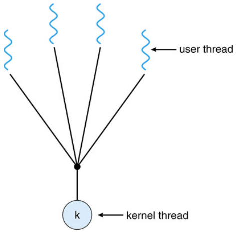


Many-to-Many



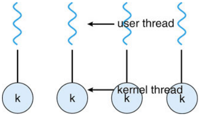
Mapping User-Threads to Kernel Threads

Many-to-One



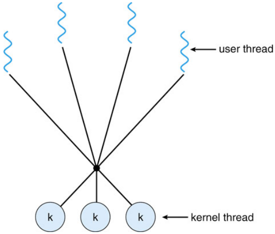
- Interpreters (CPython, older Java VMs)
- lightweight threads (small overhead)

One-to-One



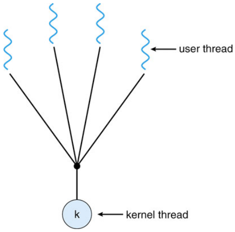
- Windows (Win32API)
- Two stacks per thread, hierarchical structure

Many-to-Many



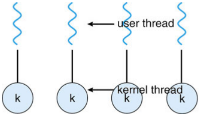
Mapping User-Threads to Kernel Threads

Many-to-One



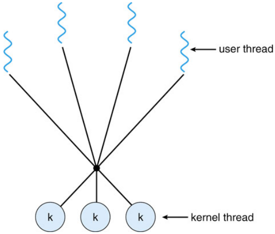
- Interpreters (CPython, older Java VMs)
- lightweight threads (small overhead)

One-to-One



- Windows (Win32API)
- Two stacks per thread, hierarchical structure

Many-to-Many



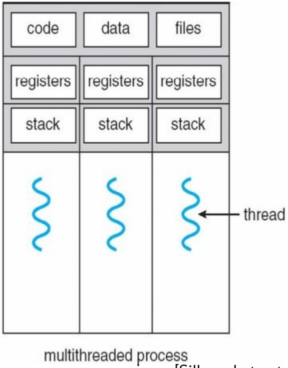
- Linux – using tasks instead of threads and processes (`clone()` sys. call).
- Effectively a thread pool.

Thread context switching

(Kernel) Thread under execution

- CPU Registers
- Stack for Kernel code
- Stack for User code (if applicable)
- Status register, program counter

Thread context saved at context switch.



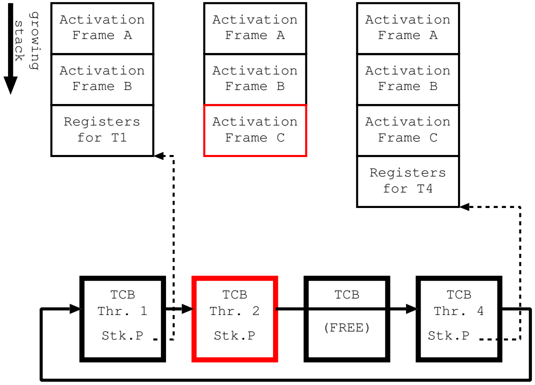
[Silberschatz et al., 2013]

- Context saved on top of the C stack
- Context pointer in the kernel's Thread Control Block (TCB)

Threads and Stacks under Execution

C example

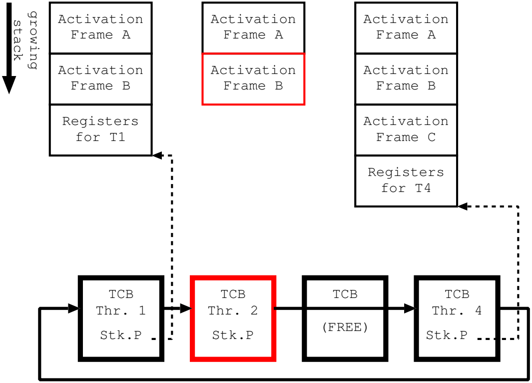
```
int C() {
  ...
}
int B() {
  int a = C();
  return a;
}
int A() {
  int a1, a2;
  a1 = B();
  a2 = C();
}
```



Threads and Stacks under Execution

C example

```
int C() {
  ...
}
int B() {
  int a = C();
  return a;
}
int A() {
  int a1, a2;
  a1 = B();
  a2 = C();
}
```

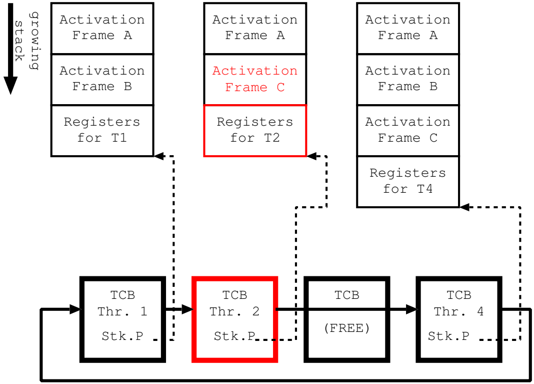


- Executing thread adds and removes C stack frames.

Threads and Stacks under Execution

C example

```
int C() {
  ...
}
int B() {
  int a = C();
  return a;
}
int A() {
  int a1, a2;
  a1 = B();
  a2 = C();
}
```

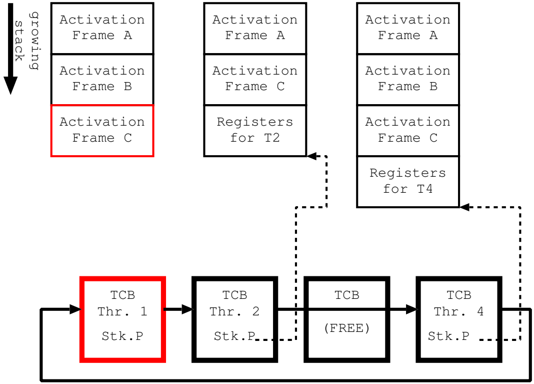


- Executing thread adds and removes C stack frames.
- Context switch (on interrupts and exceptions):
  Registers and context saved, interrupt/exception handled

Threads and Stacks under Execution

C example

```
int C() {
  ...
}
int B() {
  int a = C();
  return a;
}
int A() {
  int a1, a2;
  a1 = B();
  a2 = C();
}
```



- Executing thread adds and removes C stack frames.
- Context switch (on interrupts and exceptions):
  Registers and context saved, interrupt/exception handled
- After handling event: resume (maybe other) thread.

Context Switching Code in Buenos

- All interrupts and exceptions lead to context switch (executing interrupt handlers or exception handlers)
- Special interrupts that lead to "nothing but" context switch:
  - Timer interrupt: Time slice of running thread expired.
  - Software interrupt: Running thread requests a context switch.

Interrupt handlers then call scheduler to select a new thread.

buenos/kernel/cswitch.S

```
cswitch_switch:
    <find context_t structure for running thread>
    j cswitch_context_save # save registers, status, pc
    nop
    <init stack for action to perform>
        # After this we can call C-functions
    <change base mode if appropriate>
    <set up arguments to *_handle>
    <call *_handle>
    <find (maybe different) context_t structure to resume>
    j cswitch_context_restore # restore registers, status, pc
    nop
    eret
```

Threads and Context Switching – Idle Thread

- Most of the time: OS is waiting for instructions
- Also when executing user programs: often waiting for I/O

Threads and Context Switching – Idle Thread

- Most of the time: OS is waiting for instructions
- Also when executing user programs: often waiting for I/O

- When no threads can be run:
  OS executes a special idle thread (does nothing).

buenos/kernel/idle.S

```
_idle_thread_wait_loop:
    wait  # Enter sleep mode until an interrupt occurs
    j _idle_thread_wait_loop
    .end    _idle_thread_wait_loop
```

- No stack space needs to be reserved, no registers to save.
- When running this thread, always call scheduler on interrupt handling.

Outline

POSIX Threads

POSIX: Portable Operating System Interface

- Standardisation of UNIX-based (and other) systems
- started around 1985, later adopted by IEEE (*IEEE 1003*).
- Defines standard API for OS and tool program interfaces.
- Current version: POSIX:2008 (IEEE 1003.1-2008)

POSIX Threads

POSIX: Portable Operating System Interface

- Standardisation of UNIX-based (and other) systems
- started around 1985, later adopted by IEEE (*IEEE 1003*).
- Defines standard API for OS and tool program interfaces.
- Current version: POSIX:2008 (IEEE 1003.1-2008)

POSIX Threads (Pthreads)

- POSIX standard extension 1003.1c (1995) for multithreading
- Thread API specification (creation, control, synchronisation).
- Implementations provided by vendors – full or partial–
  (often as an addition to native thread support).

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it's me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}
```

What the thread should do

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it’s me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}
```

What the thread should do

Fct. with `void*` arg. and return type

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it's me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}

int main(int argc, char** argv) {
  int i, r;
  pthread_t threads[NUM_TREADS];
  int* thread_data = (int*) malloc(sizeof(int)*NUM_TREADS);
  if (!thread_data) error();

  for(i = 0; i < NUM_TREADS; i++) {
    thread_data[i] = i;
    r = pthread_create(&threads[i], NULL, thread_function, (void*)(thread_data+i)
    if (r != 0) { error(); }
  }
  for(i = 0; i<NUM_TREADS; i++) {
    r = pthread_join(threads[i], NULL);
    if (r != 0) { error(); }
  }
  free(thread_data);
  return 0;
}
```

What the thread should do

Fct. with `void*` arg. and return type

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it's me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}

int main(int argc, char** argv) {
  int i, r;
  pthread_t threads[NUM_TREADS];
  int* thread_data = (int*) malloc(sizeof(int)*NUM_TREADS);
  if (!thread_data) error();

  for(i = 0; i < NUM_TREADS; i++) {
    thread_data[i] = i;
    r = pthread_create(&threads[i], NULL, thread_function, (void*)(thread_data+i)
    if (r != 0) { error(); }
  }
  for(i = 0; i<NUM_TREADS; i++) {
    r = pthread_join(threads[i], NULL);
    if (r != 0) { error(); }
  }
  free(thread_data);
  return 0;
}
```

What the thread should do

Fct. with `void*` arg. and return type

Array holding thread IDs

Allocate space for arguments

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it's me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}

int main(int argc, char** argv) {
  int i, r;
  pthread_t threads[NUM_TREADS];
  int* thread_data = (int*) malloc(sizeof(int)*NUM_TREADS);
  if (!thread_data) error();

  for(i = 0; i < NUM_TREADS; i++) {
    thread_data[i] = i;
    r = pthread_create(&threads[i], NULL, thread_function, (void*)(thread_data+i));
    if (r != 0) { error(); }
  }
  for(i = 0; i<NUM_TREADS; i++) {
    r = pthread_join(threads[i], NULL);
    if (r != 0) { error(); }
  }
  free(thread_data);
  return 0;
}
```

What the thread should do

Fct. with `void*` arg. and return type

Array holding thread IDs

Allocate space for arguments

Create threads to execute function

Our first Multi-threaded Program

```
#include <pthread.h>

void* thread_function(void* data) {
  printf("Hello, it's me, thread no. %d\n", *(int*)data);
  pthread_exit(NULL);
}

int main(int argc, char** argv) {
  int i, r;
  pthread_t threads[NUM_TREADS];
  int* thread_data = (int*) malloc(sizeof(int)*NUM_TREADS);
  if (!thread_data) error();

  for(i = 0; i < NUM_TREADS; i++) {
    thread_data[i] = i;
    r = pthread_create(&threads[i], NULL, thread_function, (void*)(thread_data+i)
    if (r != 0) { error(); }
  }
  for(i = 0; i<NUM_TREADS; i++) {
    r = pthread_join(threads[i], NULL);
    if (r != 0) { error(); }
  }
  free(thread_data);
  return 0;
}
```

What the thread should do

Fct. with `void*` arg. and return type

Array holding thread IDs

Allocate space for arguments

Create threads to execute function

Wait for created threads in main

Pthreads Creation and Management

Thread creation: (thread executes start routine, attributes influence execution)

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

Return Results: (to parent thread)

```
void pthread_exit(void *arg);
```

Wait for thread: (+collect return value)

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Our Second Multi-threaded Program

Matrix Multiplication in Several Threads.

What can be done in parallel?

$$\begin{pmatrix} a_{00} & a_{01} & \ldots & a_{0n} \\ a_{10} & a_{11} & \ldots & a_{1n} \\ a_{20} & a_{21} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{k0} & a_{k1} & \ldots & a_{kn} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} & \ldots & b_{0m} \\ b_{10} & b_{11} & \ldots & b_{1m} \\ \vdots & \vdots & & \vdots \\ b_{n0} & b_{n1} & \ldots & b_{nm} \end{pmatrix}$$

Our Second Multi-threaded Program

Matrix Multiplication in Several Threads.

- Each thread computes one row of the result.
- Compute all results in the row simultaneously.
- Result space allocated before.

Thread Parameter struct

```
typedef struct ttask {
  double* row_a;
  double* matrix_b;
  int a_length, b_columns;
  double* row_result;
} TTask;
```

Our Second Multi-threaded Program

Matrix Multiplication in Several Threads.

- Each thread computes one row of the result.
- Compute all results in the row simultaneously.
- Result space allocated before.

Thread Parameter struct

```
typedef struct ttask {
  double* row_a;
  double* matrix_b;
  int a_length, b_columns;
  double* row_result;
} TTask;
```

Thread function:

```
void *rowmult(void *arg) {
  int i, j;
  TTask *tt = arg;
  for(j=0; j < tt->b_columns; j++)
    tt->row_result[j] = 0.0;
  for (i=0; i < tt->a_length; i++)
    for (j=0; j < tt->b_columns ; j++)
      tt->row_result[j] += tt->row_a[i] * tt->matrix_b[ i*(tt->b_columns)+j];
  return NULL;
}
```

$$(a_0\ a_1\ \dots\ a_n) \cdot \begin{pmatrix} b_{00} & b_{01} & \dots & b_{0m} \\ b_{10} & b_{11} & \dots & b_{1m} \\ \vdots & \vdots & & \vdots \\ b_{n0} & b_{n1} & \dots & b_{nm} \end{pmatrix}$$

Another multithreaded program...

Concurrent Money Transfers

- Global variable for account balance.
- Concurrent threads do money transfers

User withdraws at an ATM:		Balance:		State transfers SU electronically:
−500 kr.	⇒	324.17 kr.	⇐	+5.486 kr.

The Mutual Exclusion Problem

Simple example: Concurrent Money Transfers with global balance.

User withdraws at an ATM:		Balance:		State transfers SU electronically:
-500 kr.	$\Rightarrow$	324.17 kr.	$\Leftarrow$	$+5.486$ kr.

The Mutual Exclusion Problem

Simple example: Concurrent Money Transfers with global balance.

User withdraws at an ATM:

−500 kr.

⇒ Balance: 324.17 kr. ⇐

State transfers SU electronically:

+5.486 kr.

ATM: −500 kr.

```
changeBalance(Kr amount) {
 Kr bal;

 bal = balance ;

 balance  = bal + amount;

 return;
}
```

Online: +5.486 kr.

```
changeBalance(Kr amount) {
 Kr bal;

 bal = balance ;

 balance  = bal + amount;

 return;
}
```

?????? kr.

The Mutual Exclusion Problem

Simple example: Concurrent Money Transfers with global balance.

User withdraws at an ATM:

−500 kr.

⇒ Balance: 324.17 kr. ⇐

State transfers SU electronically:

+5.486 kr.

ATM:−500 kr.

```
changeBalance(Kr amount) {
 Kr bal;

 bal = balance ;

 balance  = bal + amount;

 return;
}
```

⋮

⋮

⋮

⋮

⋮

⋮

−175.83kr.

5310.17kr.

5810.17kr.

Online:+5.486 kr.

```
changeBalance(Kr amount) {
 Kr bal;

 bal = balance ;

 balance  = bal + amount;

 return;
}
```

Synchronisation

Concurrency creates problems that do not exist in a sequential world.

- Program behaviour depends on system load and external factors.
- Access to shared data needs to be protected.
- Threads can block each other when data is protected.

Synchronisation between threads that share resources

- Exclusive access to a shared resource (mutual exclusion).
- Waiting for conditions needed before starting execution.

Some terminology

Race Condition:

- Result of executing a multi-threaded program depends on the order of thread execution.

Critical Section:

- Section that changes resources

  . . . shared by several threads.

Several threads execute:

```
changeBalance(Kr amount) {
 Kr bal;

 bal = balance;
 balance = bal + amount;

 return;
}
```

Some terminology

Race Condition:

- Result of executing a multi-threaded program depends on the order of thread execution.

Critical Section:

- Section that changes resources

  . . . shared by several threads.

Mutual Exclusion:

- At most one thread is allowed inside the critical section.
- Synchronisation between the threads

  (it is realized via shared memory!)

Several threads execute:

```
start:
 ...
 /* synchronise */
 /* critical: using
   shared resources */

 /* non-critical */
 goto start
```

Mutual Exclusion with Pthread Locks

- In Pthreads API: locks (also called "mutex" – mutual exclusion)
- Mutex open: critical section free
- Mutex locked: other thread inside
- When locked, threads trying to lock will be suspended until lock open again.

Mutual Exclusion with Pthread Locks

- In Pthreads API: locks (also called "mutex" – mutual exclusion)
- Mutex open: critical section free
- Mutex locked: other thread inside
- When locked, threads trying to lock will be suspended until lock open again.

Initialisation:	`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
Locking:	`int pthread_mutex_lock(pthread_mutex_t *mutex);` `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
Unlocking:	`int pthread_mutex_unlock(pthread_mutex_t *mutex);`
Cleaning up:	`int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Pthread Locks: Usage

- Can protect critical sections by one global lock.
- Different locks for different protected data.

Correct lock usage is responsibility of the programmer.

Pthread Locks: Usage

- Can protect critical sections by one global lock.
- Different locks for different protected data.

Correct lock usage is responsibility of the programmer.

Linux Pthread implementation supports different kinds of locks:

- Linux Fast Mutex:
  - Threads can unlock a mutex locked by another thread.
  - Threads trying to obtain a lock a second time get stuck.
- Linux Errorcheck Mutex
  - Return error codes in the above cases.
- Linux Recursive Mutex
  - Allow the same thread to lock a second time.
    (called reentrant lock implementation).
  - The mutex must be unlocked as many times as it was locked.

Locks Are Sometimes Not Enough...

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  memcpy(buffer ,
         line, 80);
  pthread_mutex_unlock(m);
} while (1);
```

Shared Memory

```
char* buffer;
pthread_mutex_t m;
      ⋮
              ⋮
```

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  write(stdout,
        buffer , 80);
  pthread_mutex_unlock(m);

} while (1);
```

Locks Are Sometimes Not Enough. . .

Shared Memory

```
char* buffer;
pthread_mutex_t m;
```

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  memcpy(buffer ,
         line, 80);
  pthread_mutex_unlock(m);
} while (1);
```

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  write(stdout,
        buffer , 80);
  pthread_mutex_unlock(m);

} while (1);
```

- A file is printed line-wise by two interacting threads.

Locks Are Sometimes Not Enough...

Shared Memory

```
char* buffer;
pthread_mutex_t m;
```

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  memcpy(buffer ,
         line, 80);
  pthread_mutex_unlock(m);
} while (1);
```

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  write(stdout,
        buffer , 80);
  pthread_mutex_unlock(m);

} while (1);
```

- A file is printed line-wise by two interacting threads. One thread (producer) reads lines and copies them to a shared `buffer`. A second thread (consumer) reads and prints it.

Locks Are Sometimes Not Enough. . .

Shared Memory

```
char* buffer;
pthread_mutex_t m;
```

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  memcpy(buffer ,
         line, 80);
  pthread_mutex_unlock(m);
} while (1);
```

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  write(stdout,
        buffer , 80);
  pthread_mutex_unlock(m);

} while (1);
```

- A file is printed line-wise by two interacting threads. One thread (producer) reads lines and copies them to a shared `buffer`. A second thread (consumer) reads and prints it.
- Buffer is protected, but lines can be dropped or printed twice.
- Threads should execute in lock-step (alternating).

Pthread Condition Variables

- Threads can wait and be suspended on a certain condition. (for example: Input available, buffer free, . . .)
- Other threads can signal the condition.
  If any threads are waiting on the condition variable, one of them will become active. Otherwise, it has no effect.
- Should always be used together with a mutex. (Why?)

Pthread Condition Variables

- Threads can wait and be suspended on a certain condition. (for example: Input available, buffer free, . . .)
- Other threads can signal the condition. If any threads are waiting on the condition variable, one of them will become active. Otherwise, it has no effect.
- Should always be used together with a mutex. (Why?)

Initialisation:	`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
Waiting	`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);` `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
Signal:	`int pthread_cond_signal(pthread_cond_t *cond);`
Cleaning up:	`int pthread_cond_destroy(pthread_cond_t *cond);`

Producer and Consumer

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  while (turn != 0 )
   pthread_cond_wait(c,m);
  memcpy(buffer,
         line, 80);
  turn = 1;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

Shared Memory

```
char* buffer;
pthread_mutex_t *m;
pthread_cond_t *c;
int turn = 0;
```

⋮

⋮

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  while (turn != 1 )
    pthread_cond_wait(c, m);
  write(stdout,
        buffer, 80);
  turn = 0;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

Producer and Consumer

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  while (turn != 0 )
   pthread_cond_wait(c,m);
  memcpy(buffer,
         line, 80);
  turn = 1;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

Shared Memory

```
char* buffer;
pthread_mutex_t *m;
pthread_cond_t *c;
int turn = 0;
```

⋮

⋮

⋮

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  while (turn != 1 )
    pthread_cond_wait(c, m);
  write(stdout,
        buffer, 80);
  turn = 0;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

Producer and Consumer

File Reader reads lines:

```
char line[80];
do {
  read(handle, line,80);
  pthread_mutex_lock(m);
  while (turn != 0 )
   pthread_cond_wait(c,m);
  memcpy(buffer,
         line, 80);
  turn = 1;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

Shared Memory

```
char* buffer;
pthread_mutex_t *m;
pthread_cond_t *c;
int turn = 0;

    ⋮
            ⋮
    ⋮
```

Line Printer prints lines:

```
do {

  pthread_mutex_lock(m);
  while (turn != 1 )
    pthread_cond_wait(c, m);
  write(stdout,
        buffer, 80);
  turn = 0;
  pthread_cond_signal(c);
  pthread_mutex_unlock(m);
} while (1);
```

- Condition variable does not really test anything !
  Meaningful only together with a real condition.
- Always use with a lock: shared data may change during test.

Summary

- Concurrency is very common in operating systems.
- Threads can be at user or kernel level, with different mappings.
- PThreads, a widespread library for concurrent programming,
- . . . offers threading support and synchronisation mechanisms.

So far for today

Next time: more on synchronisation.

Blaise Barney (2011).
POSIX Threads Programming.
Lawrence Livermore National Laboratory (LLNL).
Online Tutorial, https://computing.llnl.gov/tutorials/pthreads/ (see Absalon).

IEEE (2004).
POSIX Threads (IEEE Std 1003.1, 2004 Edition).
IEEE.
http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2013).
Operating system concepts.
Wiley, 9th edition.

More Condition Variable Examples

Waiting for a counter to reach a limit.

See LLNL tutorial example.

More Condition Variable Examples

Waiting for a counter to reach a limit. See LLNL tutorial example.

Matrix Multiplication using a Thread Pool

- Limited number of threads work on tasks (compute one row).

- Main thread pushes tasks on the stack, worker threads pop them. Stack protected by mutex lock.
- Two problems arise.

Worker Thread:

```
do {
  pthread_mutex_lock(st_m);


  tt = *stack_pop(st);

  pthread_mutex_unlock(st_m);

  rowmult(tt); // solve task

} while (1); // forever
```

More Condition Variable Examples

Waiting for a counter to reach a limit. See LLNL tutorial example.

Matrix Multiplication using a Thread Pool

- Limited number of threads work on tasks (compute one row).

Worker Thread:

```
do {
  pthread_mutex_lock(st_m);


  tt = *stack_pop(st);

  pthread_mutex_unlock(st_m);

  rowmult(tt); // solve task

} while (1); // forever
```

- Main thread pushes tasks on the stack, worker threads pop them. Stack protected by mutex lock.
- Two problems arise.
    - What if main thread too slow?
    - When and how does the system terminate?

More Condition Variable Examples

Waiting for a counter to reach a limit. See LLNL tutorial example.

Matrix Multiplication using a Thread Pool

- Limited number of threads work on tasks (compute one row).

- Main thread pushes tasks on the stack, worker threads pop them. Stack protected by mutex lock.
- Two problems arise.
  - What if main thread too slow?

    Workers wait on a condition variable when stack empty.

  - When and how does the system terminate?

Worker Thread:

```
do {
  pthread_mutex_lock(st_m);
  // if empty, wait

  tt = *stack_pop(st);

  pthread_mutex_unlock(st_m);

  rowmult(tt); // solve task

} while (1); // forever
```

More Condition Variable Examples

Waiting for a counter to reach a limit. See LLNL tutorial example.

Matrix Multiplication using a Thread Pool

- Limited number of threads work on tasks (compute one row).
- Main thread pushes tasks on the stack, worker threads pop them. Stack protected by mutex lock.
- Two problems arise.
  - What if main thread too slow?

    Workers wait on a condition variable when stack empty.
  - When and how does the system terminate?

    Use a global `finishing` flag set by the main thread.

Worker Thread:

```
do {
  pthread_mutex_lock(st_m);
  // if empty, wait
  // if finishing, stop
  tt = *stack_pop(st);

  pthread_mutex_unlock(st_m);

  rowmult(tt); // solve task

} while (1); // forever
```