

# Multithreaded programming

Jan Faigl

Department of Computer Science

Faculty of Electrical Engineering

Czech Technical University in Prague

Lecture 08

**B3B36PRG – Programming in C**



# Overview of the Lecture

- Part 1 – Multithreaded Programming

Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

Synchronization Mechanisms

POSIX Threads

C11 Threads

Debugging



# Part I

## Part 1 – Multithreaded Programming



# Outline

## Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

Synchronization Mechanisms

POSIX Threads

C11 Threads

Debugging



## Terminology – Threads

- Thread is an independent execution of a sequence of instructions.
  - It is an individually performed computational flow.

*Typically a small program that is focused on a particular part.*

- Thread is running within the process.
  - It shares the same memory space as the process.
  - Threads running within the same memory space of the process.
- Thread **runtime environment** – each thread has its separate space for variables.
  - Thread identifier and space for synchronization variables.
  - Program Counter (PC) or Instruction Pointer (IP) – address of the performing instruction.

*Indicates where the thread is in its program sequence.*
  - Memory space for local variables **stack**.



## Where Threads Can be Used?

- Threads are lightweight variants of the processes that share the memory space.
- There are several cases where it is useful to use threads; the most typical situations are.
  - **More efficient usage of the available computational resources.**
    - When a process waits for resources (e.g., reads from a periphery), it is blocked, and control is passed to another process.
    - Thread also waits, but another thread within the same process can utilize the dedicated time for the process execution.
    - Having multi-core processors, we can speed up the computation using more cores simultaneously by **parallel algorithms**.
  - **Handling asynchronous events.**
    - During blocked i/o operation, the processor can be utilized for other computations.
    - One thread can be dedicated for the i/o operations, e.g., per communication channel, another thread for computations.



## Examples of Threads Usage

### ■ Input/output operations

- Input operations can take significant portions of the runtime, which may be mostly some waiting, e.g., for user input.
- During the communication, the dedicated CPU time can be utilized for computationally demanding operations.

### ■ Interactions with Graphical User Interface (GUI)

- Graphical interface requires an immediate response for a pleasant user interaction with our application.
- User interaction generates events that affect the application.
- Computationally demanding tasks should not decrease the interactivity of the application.

*Provide a nice user experience with our application.*



# Outline

Introduction

**Multithreaded applications and operating system**

Models of Multi-Thread Applications

Synchronization Mechanisms

POSIX Threads

C11 Threads

Debugging





# Threads and Processes

## Process

- Computational flow.
- Has own memory space.
- Entity (object) of the OS.
- Synchronization using OS (IPC).
- CPU allocated by OS scheduler.
  - Time to create a process.

## Threads of a process

- Computational flow.
  - Running in the same memory space of the process.
  - User or OS entity.
  - Synchronization by exclusive access to variables.
  - CPU allocated within the dedicated time to the process.
- + Creation is faster than creating a process.



# Multi-thread and Multi-process Applications

- Multi-thread application.
  - + Application can enjoy a higher degree of interactivity.
  - + Easier and faster communications between the threads using the same memory space.
  - It does not directly support scaling the parallel computation to the distributed computational environment with different computational systems (computers).
- Even on single-core single-processor systems, multi-thread applications may better utilize the CPU.

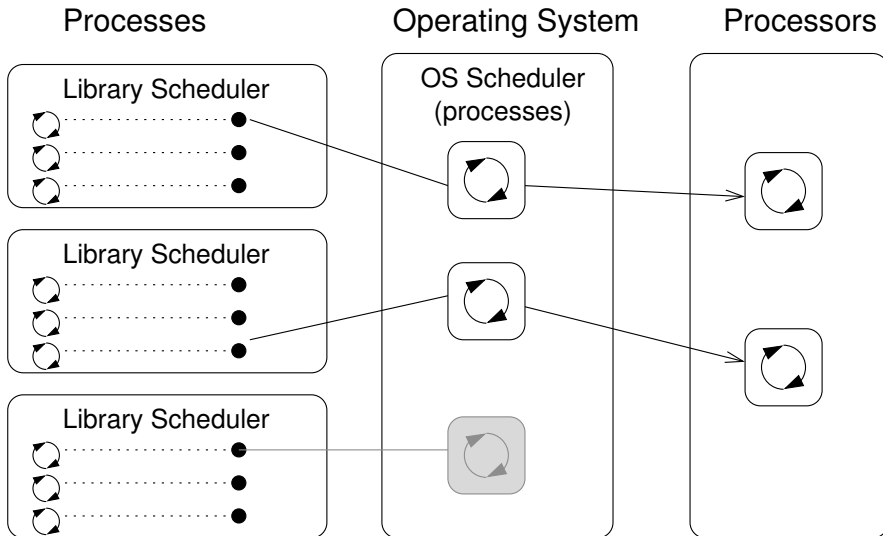


# Threads in the Operating System

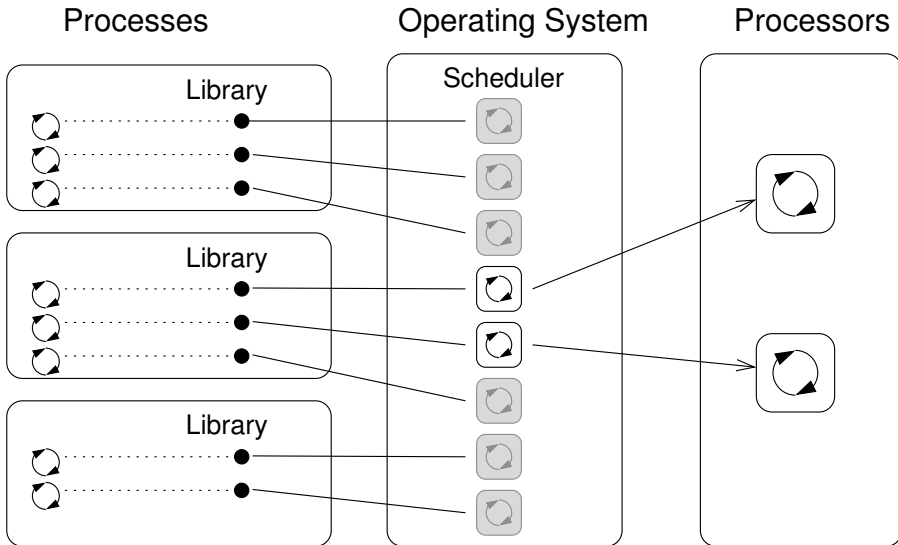
- Threads are running within the process, but regarding the implementation, threads can be in user space or OS entities.
  - **User space of the process** – threads are implemented by a user-specified library.
    - Threads do not need special support from the OS.
    - Threads are scheduled by the local scheduler provided by the library.
    - Threads typically cannot utilize more processors (multi-core).
  - **OS entities** that are scheduled by the system scheduler.
    - It may utilize multi-core or multi-processor computational resources.



# Threads in the User Space



# Threads as Operating System Entities



# User Threads vs Operating System Threads

## User Threads

- + Do not need the support of the OS.
- + Creation does not need (an expensive) system call.  
Expensive is relative to the cost of creating a thread, system thread, and process.
- Execution priority of threads is managed within the assigned process time.
- Threads cannot run simultaneously (pseudo-parallelism).

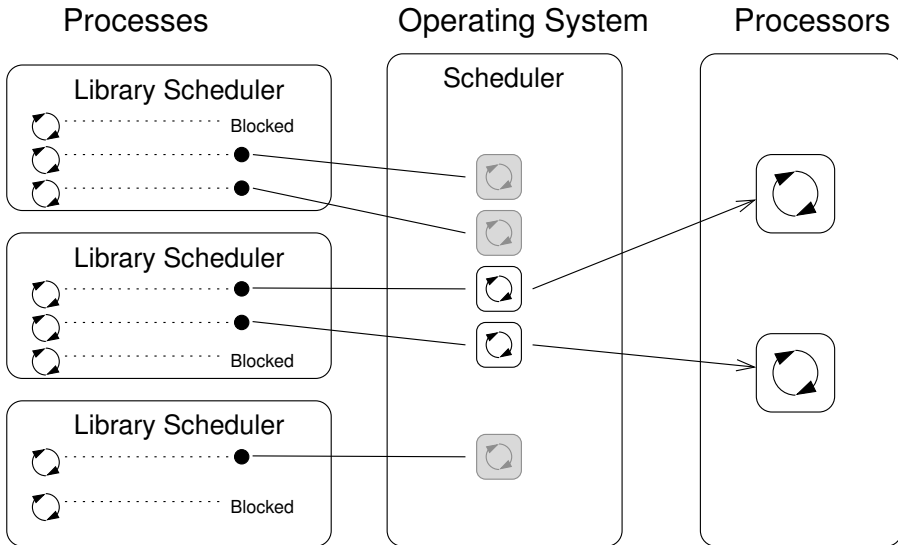
*A high number of threads scheduled by the OS may increase overhead. However, modern OS uses  $O(1)$  schedulers, which are scheduling processes independently of the number of processes. Scheduling algorithms based on complex heuristics.*

## Operating System Threads

- + Threads can be scheduled in competition with all threads in the system.
- + Threads can run simultaneously (on multi-core or multi-processor system – true parallelism).
- Thread creation is more complex (system call).



# Combining User and OS Threads



# Outline

Introduction

Multithreaded applications and operating system

**Models of Multi-Thread Applications**

Synchronization Mechanisms

POSIX Threads

C11 Threads

Debugging





## When to use Threads?

- Threads are advantageous whenever the application meets any of the following criteria.
- It consists of several independent tasks.
- It can be blocked for a certain amount of time.
- It contains a computationally demanding part (while it is also desirable to keep interactivity).
- It has to respond to asynchronous events promptly.
- It contains tasks with lower and higher priorities than the rest of the application.
- The main computation part can be speeded up by a parallel algorithm using multi-core processors.



## Typical Multi-Thread Applications

- **Servers** – serve multiple clients simultaneously. It may require access to shared resources and many i/o operations.
- **Computational application** – having a multi-core or multi-processor system, the application runtime can be decreased by using more processors simultaneously.
- **Real-time applications** – we can utilize specific schedulers to meet real-time requirements.

Multi-thread applications can be more efficient than complex asynchronous programming; a thread waits for the event vs. explicit interrupt and context switching.



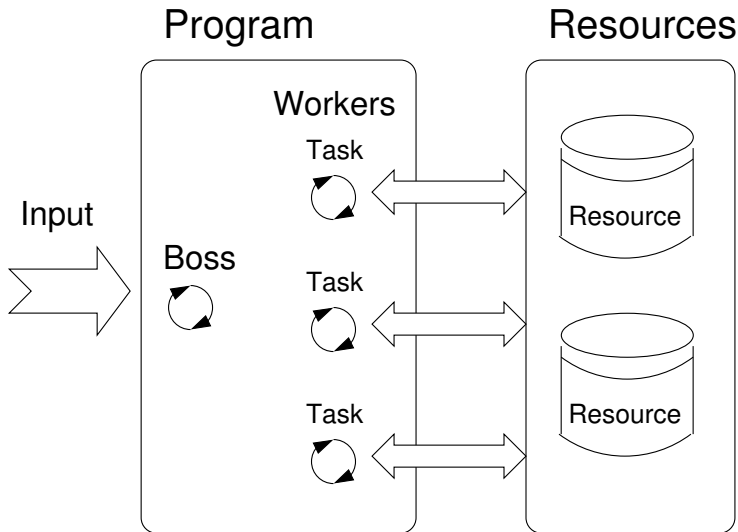
# Models of Multithreaded Applications

- Models address the creation and division of the work to particular threads.
  - **Boss/Worker** – the main thread control division of the work to other threads.
  - **Peer** – threads run in parallel without a specified manager (boss).
  - **Pipeline** – data processing by a sequence of operations.

*It assumes a long stream of input data and particular threads work in parallel on different parts of the stream*



# Boss/Worker Model



## Boss/Worker Model – Roles

- The main thread is responsible for managing the requests. It works in a cycle.
  1. Receive a new request.
  2. Create a thread for serving the particular request.

*Or passing the request to the existing thread.*
  3. Wait for a new request.
- The output/results of the assigned request can be controlled by a particular working thread or the main thread.
  - Particular thread (worker) solving the request.
  - The main thread uses synchronization mechanisms (e.g., event queue).



## Example – Boss/Worker

```
1 // Boss
2 while(1) {
3     switch(getRequest()) {
4         case taskX:
5             create_thread(taskX);
6             break;
7         case taskY:
8             create_thread(taskY);
9             break;
10    }
11 }
```

```
1 // Task solvers
2 taskX()
3 {
4     solve the task // synchronized
5     usage of shared resources
6     done;
7 }
8 taskY()
9 {
10    solve the task // synchronized
11    usage of shared resources
12    done;
13 }
```



## Thread Pool

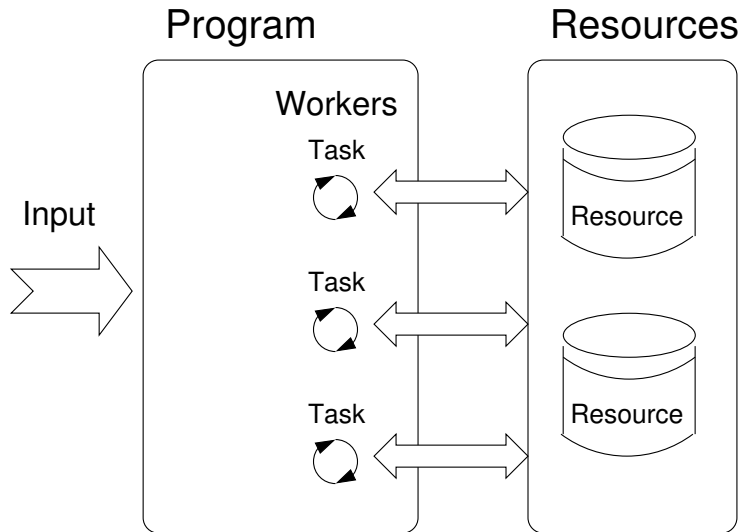
- The main thread creates threads when a new request is received.
- The overhead with the creation of new threads can be decreased using the **Thread Pool** with already created threads.
- The created threads wait for new tasks.



- Properties of the thread pool need to be considered.
  - Number of pre-created threads.
  - Maximal number of the request in the queue of requests.
  - Definition of the behavior if the queue is full and none of the threads is available.  
*E.g., block the incoming requests.*



## Peer Model





## Peer Model Properties and Example

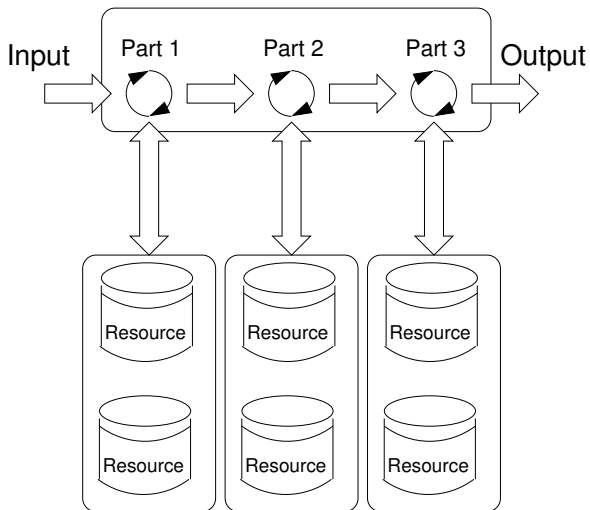
- It does not contain the main thread; the first thread creates all other threads, and then:
  - It becomes one of the other threads (equivalent).
  - It suspends its execution and waits for other threads.
- Each thread is responsible for its input and output.

```
1 // Boss
2 {
3   create_thread(task1);
4   create_thread(task2);
5   .
6   .
7   start all threads;
8   wait to all threads;
9 }
```

```
1 // Task solvers
2 task1()
3 {
4   wait to be executed
5   solve the task // sync. usage of shared resources
6   done;
7 }
9 task2()
10 {
11   wait to be executed
12   solve the task // sync. usage of shared resources
13   done;
14 }
```



## Data Stream Processing – Pipeline Program



## Pipeline Model – Properties and Example

- A long input stream of data with a **sequence of operations** (a part of processing) – each input data unit must be processed by all parts of the processing operations.
- At a particular time, different input data units are processed by individual processing parts – the input units must be independent.

```
1 main()
2 {
3     create_thread(stage1);
4     create_thread(stage2);
5     ...
6     ...
7     wait // for all pipeline;
8 }
10 stage1()
11 {
12     while(input) {
13         get next program input;
14         process input;
15         pass result to next the stage;
16     }
17 }
```

```
1 stage2()
2 {
3     while(input) {
4         get next input from thread;
5         process input;
6         pass result to the next stage;
7     }
8 }
9 ...
10 stageN()
11 {
12     while(input) {
13         get next input from thread;
14         process input;
15         pass result to output;
16     }
17 }
```



## Producer–Consumer Model

- Passing data between units can be realized using a memory buffer.
  - Or just a buffer of references (pointers) to particular data units.*
  - Producer – a thread that passes data to another thread.
  - Consumer – a thread that receives data from another thread.
- Access to the buffer must be synchronized (exclusive access).



*Using the buffer does not necessarily mean the data are copied.*



# Outline

Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

**Synchronization Mechanisms**

POSIX Threads

C11 Threads

Debugging



# Synchronization Mechanisms

- Synchronization of threads uses the same principles as synchronization of processes.
  - Because threads share the memory space with the process, the main communication between the threads is through the memory and (global) variables.
  - The crucial is the control of access to the same memory.
  - **Exclusive access** to the **critical section**.
- Basic synchronization primitives are **Mutexes** and **Conditional variables**.
  - **Mutex/Locker** for exclusive access to critical sections (mutexes or spinlocks).
  - **Condition variable** synchronization of threads according to the value of the shared variable.

*A sleeping thread can be awakened by another signaling from another thread.*



## Mutex – A Locker of Critical Section

- Mutex is a shared variable accessible from particular threads.
- Basic operations that threads may perform on the mutex.
  - **Lock** the mutex (acquired the mutex to the calling thread).
    - If the mutex cannot be acquired by the thread (because another thread holds it), the thread is **blocked and waits for mutex release**.
  - **Unlock** the already acquired mutex.
    - If one or several threads are trying to acquire the mutex (by calling the lock on the mutex), one of the threads is selected for mutex acquisition.



## Example – Mutex and Critical Section

- Lock/Unlock access to the critical section via `drawingMtx` mutex

```
1 void add_drawing_event(void)
2 {
3     Tcl_MutexLock(&drawingMtx);
4     Tcl_Event * ptr = (Tcl_Event*)Tcl_Alloc(sizeof(Tcl_Event));
5     ptr->proc = MyEventProc;
6     Tcl_ThreadQueueEvent(guiThread, ptr, TCL_QUEUE_TAIL);
7     Tcl_ThreadAlert(guiThread);
8     Tcl_MutexUnlock(&drawingMtx);
9 }
```

*Example of using thread support from the TCL library.*

- Example of using a concept of `ScopedLock`

```
1 void CCanvasContainer::draw(cairo_t *cr)
2 {
3     ScopedLock lk(mtx);
4     if (drawer == 0) {
5         drawer = new CCanvasDrawer(cr);
6     } else {
7         drawer->setCairo(cr);
8     }
9     manager.execute(drawer);
10 }
```

*The ScopedLock releases (unlocks) the mutex once the local variable `lk` is destroyed at the end of the function call.*





## Generalized Models of Mutex

- Recursive – the mutex can be locked multiple times by the same thread.
- Try – the lock operation immediately returns if the mutex cannot be acquired.
- Timed – limit the time to acquire the mutex.
- *Spinlock* – the thread repeatedly checks if the lock is available for the acquisition.

*Thread is not set to blocked mode if the lock cannot be acquired.*



# Spinlock

- Under certain circumstances, it may be advantageous not to block the thread during the acquisition of the mutex (lock), e.g.,
  - Performing a simple operation on the shared data/variable on the system with true parallelism (using multi-core CPU).
  - Blocking the thread, suspending its execution, and passing the allocated CPU time to other threads may result in a significant overhead.
  - Other threads quickly perform other operations on the data, and thus, the shared resource would be quickly accessible.
- During the locking, the thread actively tests if the lock is free.

*It wastes the CPU time that can be used for productive computation elsewhere.*
- Similarly to a semaphore, such a test has to be performed by TestAndSet instruction at the CPU level.
- **Adaptive mutex** combines both approaches to use the **spinlocks** to access resources locked by the currently running thread and block/sleep if such a thread is not running.

*It does not make sense to use spinlocks on single-processor systems with pseudo-parallelism.*



# Condition Variable

- **Condition variable** allows signaling thread from other thread.
- The concept of **condition variable** allows the following synchronization operations.
  - Wait – the variable has been changed/notified.
  - Timed waiting for a signal from another thread.
  - Signaling other thread waiting for the condition variable.
  - Signaling all threads waiting for the condition variable.

*All threads are awakened, but the access to the condition variable is protected by the mutex that must be acquired, and only one thread can lock the mutex.*



## Example – Condition Variable

- Example of using condition variable with lock (mutex) to allow exclusive access to the condition variable from different threads.

```
Mutex mtx; // shared variable for both threads
CondVariable cond; // shared condition variable
```

```
// Thread 1
Lock(mtx);
// Before code, wait for Thread 2
CondWait(cond, mtx); // wait for cond
... // Critical section
Unlock(mtx);
```

```
// Thread 2
Lock(mtx);
... // Critical section
// Signal on cond
CondSignal(cond, mtx);
Unlock(mtx);
```



## Parallelism and Functions

- In a parallel environment, functions can be called multiple times.
- Regarding the parallel execution, functions can be **reentrant** or **thread-safe**.
  - **Reentrant** – at a single moment, the same function can be executed multiple times simultaneously.
  - **Thread-Safe** – the function can be called by multiple threads simultaneously.
- The following needs to be satisfied for achieving the properties.
  - **Reentrant function** does not write to static data and does not work with global data.
  - **Thread-safe function** strictly access to global data using synchronization primitives.



## Main Issues with Multithreaded Applications

- The main issues/troubles with multiprocessing applications are related to synchronization.
  - **Deadlock** – a thread waits for a resource (mutex) that is currently locked by another thread that is waiting for the resource (thread) already locked by the first thread.
  - **Race condition** – access of several threads to the shared resources (memory/variables), and at least one of the threads does not use the synchronization mechanisms (e.g., critical section).

*A thread reads a value while another thread is writing the value. If Reading/writing operations are not atomic, data are not valid.*



# Outline

Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

Synchronization Mechanisms

**POSIX Threads**

C11 Threads

Debugging



## POSIX Thread Functions (pthread)

- POSIX threads library (`<pthread.h>` and `-lpthread`) is a set of functions to support multithreaded programming.
- The basic types for threads, mutexes, and condition variables are
  - `pthread_t` – type for representing a thread;
  - `pthread_mutex_t` – type for mutex;
  - `pthread_cond_t` – type for condition variable.
- The thread is created by `pthread_create()` function call, which immediately executes the new thread as a function passed as a pointer to the function.

*The thread calling the creation continues with the execution.*

- A thread may wait for another thread by `pthread_join()`.
- Particular mutex and condition variables have to be initialized using the library calls.

*Note, initialize shared variables before threads are created.*

- `pthread_mutex_init()` – initialize mutex variable.
- `pthread_cond_init()` – initialize condition variable.

*Additional attributes can be set; see documentation.*





## POSIX Threads – Example 1/10

- Create an application with three active threads for
  - Handling user input – function `input_thread()`.
    - User specifies a period output refresh by pressing dedicated keys.
  - Refresh output – function `output_thread()`.
    - Refresh output only when the user interacts with the application or the alarm is signaling the period has been passed.
  - Alarm with user defined period – function `alarm_thread()`.
    - Refresh the output or do any other action.
- For simplicity the program uses `stdin` and `stdout` with thread activity reporting to `stderr`.
- Synchronization mechanisms are demonstrated using
  - `pthread_mutex_t mtx` – for exclusive access to `data_t data`;
  - `pthread_cond_t cond` – for signaling threads.

*The shared data consists of the current period of the alarm (`alarm_period`), request to quit the application (`quit`), and number of alarm invocations (`alarm_counter`).*



## POSIX Threads – Example 2/10

- Including header files, defining data types, and declaration of global variables.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <termios.h>
5 #include <unistd.h> // for STDIN_FILENO
6 #include <pthread.h>
7
8 #define PERIOD_STEP 10
9 #define PERIOD_MAX 2000
10 #define PERIOD_MIN 10
11
12 typedef struct {
13     int alarm_period;
14     int alarm_counter;
15     bool quit;
16
17     pthread_mutex_t *mtx; // avoid global variables for mutex and
18     pthread_cond_t *cond; // conditional variable
19 } data_t; // data structure shared among the threads
```



## POSIX Threads – Example 3/10

- Functions prototypes and initialize of variables and structures.

```
21 void call_termios(int reset); // switch terminal to raw mode
22 void* input_thread(void*);
23 void* output_thread(void*);
24 void* alarm_thread(void*);
25
26 // - main function -----
27 int main(int argc, char *argv[])
28 {
29     data_t data = { .alarm_period = 100, .alarm_counter = 0, .quit = false };
30     enum { INPUT, OUTPUT, ALARM, NUM_THREADS }; // named ints for the threads
31     const char *threads_names[] = { "Input", "Output", "Alarm" };
32     void* (*thr_functions[])(void*) = {
33         input_thread, output_thread, alarm_thread // array of thread functions
34     };
35
36     pthread_t threads[NUM_THREADS]; // array for references to created threads
37     pthread_mutex_t mtx;
38     pthread_cond_t cond;
39     pthread_mutex_init(&mtx, NULL); // init. mutex with default attributes
40     pthread_cond_init(&cond, NULL); // init. condition variable with default attributes
41     data.mtx = &mtx; // make mtx accessible from the shared data structure
42     data.cond = &cond; // make cond accessible from the shared data structure
```



## POSIX Threads – Example 4/10

- Create threads and wait for terminations of all threads.

```
43  call_termios(0); // switch terminal to raw mode
44  for (int i = 0; i < NUM_THREADS; ++i) {
45      int r = pthread_create(&threads[i], NULL, thr_functions[i], &data);
46      printf("Create thread '%s' %s\r\n", threads_names[i], ( r == 0 ? "OK" :
47          "FAIL" ) );
48  }
49  int *ex;
50  for (int i = 0; i < NUM_THREADS; ++i) {
51      printf("Call join to the thread %s\r\n", threads_names[i]);
52      int r = pthread_join(threads[i], (void*)&ex);
53      printf("Joining the thread %s has been %s - exit value %i\r\n",
54          threads_names[i], (r == 0 ? "OK" : "FAIL"), *ex);
55  }
56  call_termios(1); // restore terminal settings
57  return EXIT_SUCCESS;
58 }
```



## POSIX Threads – Example 5/10 (Terminal Raw Mode)

- Switch terminal to raw mode.

```
59 void call_termios(int reset)
60 {
61     static struct termios tio, tioOld; // use static to preserve the initial
        settings
62     tcgetattr(STDIN_FILENO, &tio);
63     if (reset) {
64         tcsetattr(STDIN_FILENO, TCSANOW, &tioOld);
65     } else {
66         tioOld = tio; //backup
67         cfmakeraw(&tio);
68         tcsetattr(STDIN_FILENO, TCSANOW, &tio);
69     }
70 }
```

*The caller is responsible for appropriately calling the function, e.g., to preserve the original settings, the function must be called with the argument zero only once.*



## POSIX Threads – Example 6/10 (Input Thread 1/2)

```
72 void* input_thread(void* d)
73 {
74     data_t *data = (data_t*)d;
75     static int r = 0;
76     int c;
77     while (( c = getchar()) != 'q') {
78         pthread_mutex_lock(data->mtx);
79         int period = data->alarm_period; // save the current period
80         // handle the pressed key detailed in the next slide
81     }
82     ...
83
84     if (data->alarm_period != period) { // the period has been changed
85         pthread_cond_signal(data->cond); // signal the output thread to refresh
86     }
87     data->alarm_period = period;
88     pthread_mutex_unlock(data->mtx);
89 }
90
91 r = 1;
92 pthread_mutex_lock(data->mtx);
93 data->quit = true;
94 pthread_cond_broadcast(data->cond);
95 pthread_mutex_unlock(data->mtx);
96 fprintf(stderr, "Exit input thread %lu\r\n", pthread_self());
97 return &r;
98 }
```



## POSIX Threads – Example 7/10 (Input Thread 2/2)

- `input_thread()` – handle the user request to change period.

```
81  switch(c) {
82      case 'r':
83          period -= PERIOD_STEP;
84          if (period < PERIOD_MIN) {
85              period = PERIOD_MIN;
86          }
87          break;
88      case 'p':
89          period += PERIOD_STEP;
90          if (period > PERIOD_MAX) {
91              period = PERIOD_MAX;
92          }
93          break;
94  }
```



## POSIX Threads – Example 8/10 (Output Thread)

```
96 void* output_thread(void* d)
97 {
98     data_t *data = (data_t*)d;
99     static int r = 0;
100    bool q = false;
101    pthread_mutex_lock(data->mtx);
102    while (!q) {
103        pthread_cond_wait(data->cond, data->mtx); // wait for next event
104        q = data->quit;
105        printf("\rAlarm time: %10i    Alarm counter: %10i", data->alarm_period,
106            data->alarm_counter);
107        fflush(stdout);
108    }
109    pthread_mutex_unlock(data->mtx);
110    fprintf(stderr, "Exit output thread %lu\r\n", (unsigned long)pthread_self());
111    return &r;
112 }
```





## POSIX Threads – Example 9/10 (Alarm Thread)

```
113 void* alarm_thread(void* d)
114 {
115     data_t *data = (data_t*)d;
116     static int r = 0;
117     pthread_mutex_lock(data->mtx);
118     bool q = data->quit;
119     useconds_t period = data->alarm_period * 1000; // alarm_period is in ms
120     pthread_mutex_unlock(data->mtx);
121     while (!q) {
122         usleep(period);
123         pthread_mutex_lock(data->mtx);
124         q = data->quit;
125         data->alarm_counter += 1;
126         period = data->alarm_period * 1000; // update the period if it has been changed
127         pthread_cond_broadcast(data->cond);
128         pthread_mutex_unlock(data->mtx);
129     }
130     fprintf(stderr, "Exit alarm thread %lu\r\n", pthread_self());
131     return &r;
132 }
133 }
```



## POSIX Threads – Example 10/10

- The example program `lec08/threads.c` can be compiled and run.

```
clang -c threads.c -std=gnu99 -O2 -pedantic -Wall -o threads.o
clang threads.o -lpthread -o threads
```

- The period can be changed by 'r' and 'p' keys.
- The application is terminated after pressing 'q'.

```
./threads
Create thread 'Input' OK
Create thread 'Output' OK
Create thread 'Alarm' OK
Call join to the thread Input
Alarm time:          110   Alarm counter:          20Exit input thread 750871808
Alarm time:          110   Alarm counter:          20Exit output thread 750873088
Joining the thread Input has been OK - exit value 1
Call join to the thread Output
Joining the thread Output has been OK - exit value 0
Call join to the thread Alarm
Exit alarm thread 750874368
Joining the thread Alarm has been OK - exit value 0
```

`lec08/threads.c`



# Outline

Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

Synchronization Mechanisms

POSIX Threads

**C11 Threads**

Debugging



# C11 Threads

- C11 provides a “wrapper” for the POSIX threads.

*E.g., see <http://en.cppreference.com/w/c/thread>*

- The library is `<threads.h>` and `-lstdthreads`.

- Basic types

- `thrd_t` – type for representing a thread;
- `mtx_t` – type for mutex;
- `cnd_t` – type for condition variable.

- Creation of the thread is `thrd_create()` and the thread body function has to return an `int` value.

- `thrd_join()` is used to wait for a thread termination.

- Mutex and condition variable are initialized (without attributes)

- `mtx_init()` – initialize mutex variable;
- `cnd_init()` – initialize condition variable.



## C11 Threads Example

- The previous example `lec08/threads.c` implemented with C11 threads is in `lec08/threads-c11.c`.

```
clang -std=c11 threads-c11.c -lstdthreads -o threads-c11
./threads-c11
```

- Basically, the function calls are similar, with different names and minor modifications.
  - `pthread_mutex_*`() → `mxt_*`().
  - `pthread_cond_*`() → `cnd_*`().
  - `pthread_*`() → `thrd_*`().
  - Thread body functions return int value.
  - There is not `pthread_self()` equivalent.
  - `thrd_t` is implementation dependent
  - Threads, mutexes, and condition variables are created/initialized without specification of particular attributes.
  - The program is linked with the `-lstdthreads` library.

*Simplified interface.*

`lec08/threads-c11.c`



# Outline

Introduction

Multithreaded applications and operating system

Models of Multi-Thread Applications

Synchronization Mechanisms

POSIX Threads

C11 Threads

**Debugging**



# How to Debug Multi-Thread Applications

- The best tool to debug a multi-thread application is  
to do not need to debug it.
- It can be achieved by discipline and a prudent approach to shared variables.
- Otherwise a debugger with a minimal set of features can be utilized.



# How to Debug Multi-Thread Applications

- The best tool to debug a multi-thread application is  
to do not need to debug it.
- It can be achieved by discipline and a prudent approach to shared variables.
- Otherwise a debugger with a minimal set of features can be utilized.





# How to Debug Multi-Thread Applications

- The best tool to debug a multi-thread application is  
to do not need to debug it.
- It can be achieved by discipline and a prudent approach to shared variables.
- Otherwise a debugger with a minimal set of features can be utilized.



# Debugging Support

- Desired features of the debugger.
  - List of running threads.
  - Status of the synchronization primitives.
  - Access to thread variables.
  - Breakpoints in particular threads.

lldb – <http://lldb.llvm.org>; gdb – <https://www.sourceware.org/gdb>  
cgdb, ddd, kgdb, Code::Blocks or Eclipse, Kdevelop, Netbeans, CLion

SlickEdit – <https://www.slickedit.com>; TotalView – <http://www.roguewave.com/products-services/totalview>

- **Logging** can be more efficient for debugging a program than manual debugging with manually set breakpoints.
  - Deadlock is mostly related to the order of locking.
  - Logging and analyzing access to the lockers (mutex) can help to find the wrong order of the thread synchronizing operations.



## Comments – Race Condition

- Race condition is typically caused by a lack of synchronization.
- It is worth remembering the following.
  - **Threads are asynchronous!**

*Do not rely that a code execution is synchronous on a single processor system.*

- **When writing multi-threaded applications, assume that the thread can be interrupted or executed at any time!**

*Parts of the code that require a particular execution order of the threads need synchronization.*

- **Never assume that a thread waits after it is created!**

*It can be started very soon and usually much sooner than you can expect.*

- **Unless you specify the order of the thread execution, there is no such order!**

*"Threads are running in the worst possible order". Bill Gallmeister"*



## Comments – Deadlock

- Deadlocks are related to the mechanisms of synchronization.
  - Deadlock is much easier to debug than the race condition.
  - Deadlock is often the *mutex deadlock* caused by order of multiple mutex locking.
  - **Mutex deadlock can not occur** if, at any moment, each thread has (or it is trying to acquire) **at most a single mutex**.
  - It is **not recommended to call functions with a locked mutex**, especially if the function is attempting to lock another mutex.
  - **It is recommended to lock the mutex for the shortest possible time.**



# Summary of the Lecture



# Topics Discussed

- Multithreaded programming
  - Terminology, concepts, and motivations for multithreaded programming
  - Models of multi-threaded applications
  - Synchronization mechanisms
  - POSIX and C11 thread libraries

*Example of an application*

- Comments on debugging and multi-thread issues with the race condition and deadlock

