# Threads

# Thread

Threading is the creation and management of a variety of executable elements within a single process

A process contains one or more threads

Exist in the same process address space

# The advantages of threads

- Parallelism
- Context switch
- Memory saving

# What to choose?

Processes are generally **more reliable**

- processes interacting at different times
- distributed application
- data security

Threads - generally **faster**

- easier IPC

# Types of implementations

- User threads (Linux 2.4)
- Kernel thread (Linux version >= 2.6, NPTL)
- Hybrid implementation

# User threads

**The benefits of custom threads are as follows:**

- thread switching does not require kernel participation - no switching from task mode to kernel mode;
- planning can be determined by the application-the best algorithm is chosen;
- user threads can be used on any OS - only a compatible thread library is required.

**Disadvantages of user threads:**

- most system calls are blocking and the kernel blocks processes-including all threads within the process;
- the kernel can only schedule processes to processors - two threads within the same process cannot run simultaneously on two different processors.

# Kernel thread

**Benefits of kernel-level threads:**

- the kernel can schedule multiple threads of the same process to run on multiple processors at the same time, blocking is performed at the thread level;
- kernel procedures can be multithreaded.

**Disadvantages:**

- switching threads within a single process requires the participation of the kernel.

# POSIX Threads

There are many implementations of threads:

**Android, Apache, GNOME, Mozilla**

Unified standard of UNIX-threads

**POSIX-Threads**

# API

<pthread.h>

**About 100 system calls:**

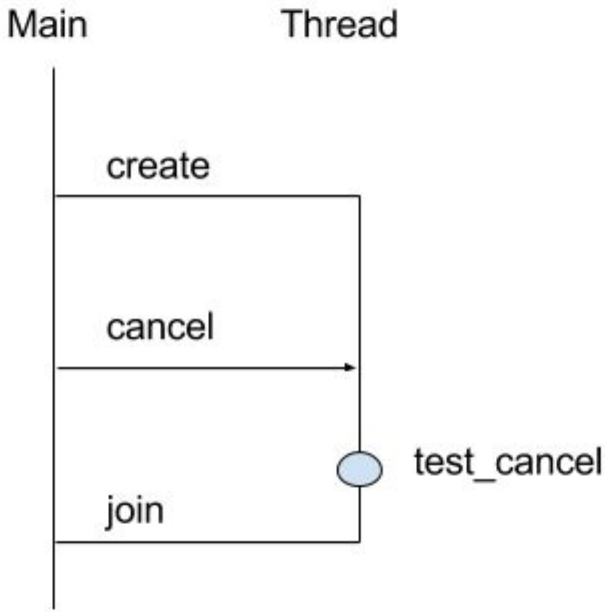
- thread control-functions for creating, destroying, joining, and detaching threads
- synchronization - functions to control thread synchronization

# Linking

Thread library: **libpthread**

gcc **-pthread** example.c

# Thread life cycle




cleanup_push(*)
cleanup_pop()

# libc: pthread_cancel.c

```c
  /* Mark the thread as canceled.  This has to be done
     atomically since other bits could be modified as well.  */
  while (atomic_compare_and_exchange_bool_acq (&pd->cancelhandling, newval,
                                               oldval));

  return result;
}
```

https://github.com/lattera/glibc/blob/a2f34833b1042d5d8eeb263b4cf4caaea138c4ad/nptl/pthread_cancel.c

# Creating threads

```
#include <pthread.h>

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

# Return result

0 - on success

otherwise, an error number (without the use of errno!)

| | |
|---|---|
| EAGAIN | the calling process is severely under-resourced to create a new thread; this is usually because the process has reached the thread count limit for each user or for the entire system; |
| EINVAL | the pthread_attr_t object, specified via the attr has an invalid attributes; |
| EPERM | the calling process does not have permission to set some attributes of the pthread_attr_t object specified through attr |

# Example

```
pthread_t thread;
int ret;
ret = pthread_create (&thread, NULL, start_routine, NULL);
if (!ret) {
    errno = ret;
    perror("pthread_create");
    return -1;
}
```

# Thread ID

Unlike **PID-a** is assigned by the library

Although there is **gettid() -** use it does not make sense

```
#include <pthread.h>
pthread_t pthread_self (void);
```

# Comparison of identifiers

pthread_t  - thread ID

```
#include <pthread.h>
int pthread_equal (pthread_t t1, pthread_t t2);
```

0 - if equal, otherwise not equal

# The end of the stream

- if a thread returns from the start procedure, it is interrupted; this is the equivalent of" out of bounds " in main();

- if the thread invokes the function **pthread_exit()**, it terminates; this is the equivalent of calling exit();

- if a thread is canceled by another thread through the **pthread_cancel ()** function, it terminates; this is analogous to sending a SIGKILL signal through kill().

# Exit

```
#include <pthread.h>

void pthread_exit (void *retval);
```

# Join threads

#include <pthread.h>

int pthread_join (pthread_t thread, void **retval);

Some analog of call **wait()**

retval - returned value

Each thread can join any other

# Possible errors

EDEADLK — a deadlock has occurred — thread is already waiting to join the calling thread or is itself a calling thread;

EINVAL — it is impossible to attach a thread, specified through the **thread**

ESRCH — he value of the **thread** is unacceptable.

# Example

Create threads, attach

pthread1.c

pthread1-2.c

pthread1-3.c

pthread1-4.c

# Exercise 4a

Create program that create two threads.

First thread should output: “Hello”

Second thread should output: “World”

Main thread should wait (*use phread_join*) while all threads are finished

**templates: pthread1-*.c**

# Example

PI computation without race conditions

phread_pi_race.c

phread_pi.c

# Initializing mutexes

```
#include <pthread.h>

pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

# Locking mutexes

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

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

A successful call will block the calling thread until the mutex specified as mutex is available

# Errors

EDEADLK — the calling thread already owns the requested mutex

EINVAL — mutex value is not valid.

# Unlocking mutexes

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

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

A successful call releases the mutex specified by mutex and return 0

# Errors

EINVAL — mutex value is not valid

EPERM — the calling process does not own the mutex specified as mutex; attempting to release a mutex that you do not own is an error.

# Example

PI computation with single shared memory and race conditions

pthread_pi_lock_1.c - race

pthread_pi_lock_2.c - lock

pthread_pi_lock_3.c - smart lock

pthread_pi_lock_4.c - spin lock

pthread_pi_lock_5.c - spin lock 2

# Semaphores

```
// Init
int sem_init(sem_t *sem, int pshared, unsigned int value);

// Increase by 1
int sem_post(sem_t *sem);

// Decrease by 1
int sem_wait(sem_t *sem);
```

# Example

Basic semaphore usage

phread_sems.c

# Exercise 4b

Create parallel version of program for vectors multiplication based on PThreads.

Questions:

1. Compare execution time of the two versions of the program:
   a. based on **fork**s
   b. based on **pthread**s.

**Template: vector-multiplication-fork.c**

# Condition Variables

Condition variables provide yet another way for threads to synchronize.

While mutexes implement synchronization by controlling **thread access to data**, condition variables allow threads to synchronize **based upon the actual value of data**.

# Condition Variables

Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

A condition variable is always used in conjunction with a mutex lock.

# Condition Variables

**Thread A**

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call **pthread_cond_wait()** to perform a blocking wait for signal from Thread-B. Note that a call to **pthread_cond_wait()** automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

**Thread B**

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

# Condition Variables

Condition variables must be declared with type **pthread_cond_t**, and must be initialized before they can be used. There are two ways to initialize a condition variable:

1. Statically, when it is declared. For example:
   **pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;**
2. Dynamically, with the **pthread_cond_init()** routine.

# Waiting and Signaling on Condition Variables

**pthread_cond_wait()** blocks the calling thread until the specified *condition* is signalled.

This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits.

After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread.

The programmer is then responsible for unlocking *mutex* when the thread is finished with it.

# Example

condition_var.c

# Example

Condition Variables and Prime numbers

find_prime_number_seq_0.c

# Exercise 4c

Create parallel version of program for finding prime number under certain position. Use at least **two threads**.

**template: find_prime_number_seq_4.c**

# Completing other threads

#include <pthread.h>

int pthread_cancel (pthread_t thread);

0 - if successful, otherwise the parameter **thread** is incorrect

# Cancellation policy

The cancellation status of the thread (enabled by default)

#include <pthread.h>

int pthread_setcancelstate (int state, int *oldstate);

state: **PTHREAD_CANCEL_ENABLE** или
**PTHREAD_CANCEL_DISABLE**

# Type abort the thread

- Asynchronous:

At any time

- Deferred:

Only at certain moments

# Change the type of cancellation

#include <pthread.h>

int pthread_setcanceltype (int type, int *oldtype);

type: **PTHREAD_CANCEL_ASYNCHRONOUS** или **PTHREAD_CANCEL_DEFERRED**

# Check point for delayed thread cancellation

```
#include <pthread.h>

void pthread_testcancel(void);
```

**Release resources before completion**

pthread_cleanup_push( void *func, void *arg)

Run when:

- Thread cancel
- Exit via pthread_exit
- pthread_cleanup_pop

# Example

Delayed thread stop

pthread2.c

# Example streams from Java

```java
Thread t;

t.start();
t.interrupt() // isInterrupted
t.join();
```

Special method t.interrupted()

```java
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

# Release resources

```
while( true ) {
  try {
    // do something
    // sleep(1)
  }
  catch( InterruptionException e ) {

  }
}
```

Analogue **pthread_cleanup_push**