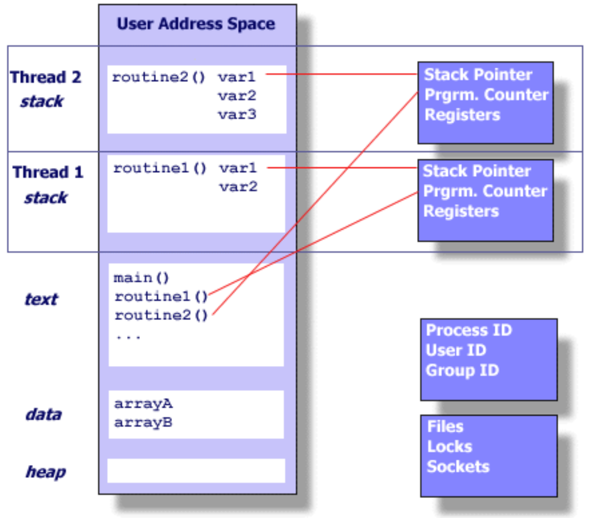
Real-Time Programming TI00AA55

Lecture 12 - 22.04.2015

Jarkko.Vuori@metropolia.fi

Thread basics

- Thread is like a process in a sense that it has its own scheduling
  - This means that threads are executed concurrently like processes
- Threads can be implemented in the kernel level or using the so called thread library (user level function library, so called jacket)
- If a computer has several processors, threads can be genuinely executed simultaneously in separate processors (real parallel processing)


User Address Space
Thread 2 stack
routine2() var1 var2 var3
Thread 1 stack
routine1() var1 var2
text
main() routine1() routine2() ...
data
arrayA arrayB
heap
Stack Pointer Prgrm. Counter Registers
Stack Pointer Prgrm. Counter Registers
Process ID User ID Group ID
Files Locks Sockets

- Thread is a ”lighter form of the process”, because
  1. Context switch between processes demands more resources than context switch between threads
  2. The communication between threads is more effective, because common global variables can be used to pass information
  3. For the same reason we can use lighter tools for thread synchronization than for process synchronization (mutex and condition variable)
- A thread is a function of the process that has it’s own scheduling
  - Every thread has it’s own stack segment (local variables)
    - But thread can asses another thread’s stack segment if he knows the right address
  - All threads of the process share the common data segment (containing global variables)
  - Threads can also share the code of function they are executing
- POSIX defines the threads in the extension POSIX:THR

Thread basics

- Thread is a “new” solution to our old problem where we had to give a quick responses to inputs from many file descriptors meaning that we cannot block the read operation from any descriptor
- We have already seen four solutions to this problems:
  1. Polling method (using non blocked mode of i/o)
  2. Separate processes
  3. Multiplexed i/o
  4. Asynchronous i/o
- The new fifth method is using different threads to read different file descriptors (there is one dedicated thread to each descriptor)
  – Then we can use blocked reading, because thread can block if there is nothing to read
  – Blocking in one thread does not prevent other file descriptors from being read in other threads, because they have their own scheduling
  – Method is closely related to the method 2 above
    - The difference is that in this case it is easier to change information between different parts of the program, because they are threads and not processes
- This method makes the whole program simple, because you don’t have to worry about blocking of read functions (or write functions)

Creation of the thread

- We have many similarities between executing a thread and executing a process: Thread is created, thread can terminate the execution, we can wait for the termination of a thread, a thread has an exit status and so on. Let's start with these three things.
- **Creation of a thread** (pthread_create)

```
<pthread.h>
int pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void*(*thread_routine)(void*),
void *arg);
```

- Function creates a thread that has it's own scheduling
- Thread is a series of instructions that are executed independently
  - This series of instructions consists of instructions of the function (thread_routine) passed as a parameter to the function `pthread_create`
- Function `pthread_create` returns and the thread, where it was called from, continues
- The function `pthread_routine` is never called and it never returns
  - It is possible to wait for it in another thread (compare to `fork` and `exec` functions)
- Function `pthread_create` stores the id of the new thread in the variable where parameter `thread` is pointing to
- The parameter `attr` can be used to give attributes for the thread
  - If this parameter is NULL, default attributes are used
  - The last (fourth) parameter is passed to the thread function when it is first time scheduled

Some remarks of thread creation

- Remark 1. Thread is automatically in “ready to run state” after successful creation. This means that it is started automatically when scheduler gives a turn to run
- Remark 2. It would be possible to call the thread function in the usual way and not using `pthread_create`. The function would then be run in the thread of the caller

Thread termination and waiting

- Thread termination (pthread_exit)
  `void pthread_exit(void *value);`
- A thread terminates, when it calls function `pthread_exit`. The function takes one parameter. This parameter is a **pointer**
  - The pointer points to the exit status of the thread
  - A thread can wait for the termination of another thread
  - The waiting process gets the pointer to the exit status so that it can examine the exit status of the terminated thread

- **Waiting** for a thread (pthread_join)
  ```
  <pthread.h>
  int pthread_join(pthread_t thread,  void
  **value_ptr);
  ```
- A thread can wait for another thread (what ever thread can wait for what ever another thread) by calling `pthread_join`
- The parameter thread indicates the thread we are waiting for
  - The function blocks if the thread is not terminated yet
  - If the thread has been terminated before `pthtread_join` function is called, `pthread_join` returns immediately (still receives the pointer)
- If we don’t want to wait for a thread and want to save system resources that are needed to maintain the exit status, it is possible to **detach** the thread (from the waiters) with the function pthread_detach
  `int pthread_detach(pthread_t thread);`
- Example. A thread can detach itself by calling
  ```
          pthread_detach(pthread_self());
  ```

Example: create, join, exit

- How to create, wait (join) and exit a thread:

```
void* tf(void*);
int status;

int main(void) {
  pthread_t id;
  int      *pstat;

  pthread_create(&id, NULL, tf, NULL);
  //…
  pthread_join(id, (void**)&pstat);
  printf("Status is %d\n", *pstat);
}

void* tf(void* p) {
  //…
  status = 10;
  pthread_exit((void*) &status);
  // or return &status;
}
```

- Remark. The example above demonstrates in a simple way
  1. how a thread is created,
  2. how a thread can be waited for and
  3. how a thread terminates passing a termination status to the thread that is waiting for it
- In this example the thread stores a value in a global variable and passes it's pointer to the waiting thread
  – The waiting thread could of course access the global variable directly

Process and thread comparison

Create and wait a
child process

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

int main(void){
    pid_t pid;
    int   status;

    pid = fork();
    if (pid > 0) {
        // task of parent
        ----
        wait(&status);
        exit(0);
    }
    if (pid == 0) {
        // task of child
        ----
        exit(0);
    }
}
```

Create and wait (join)
a thread

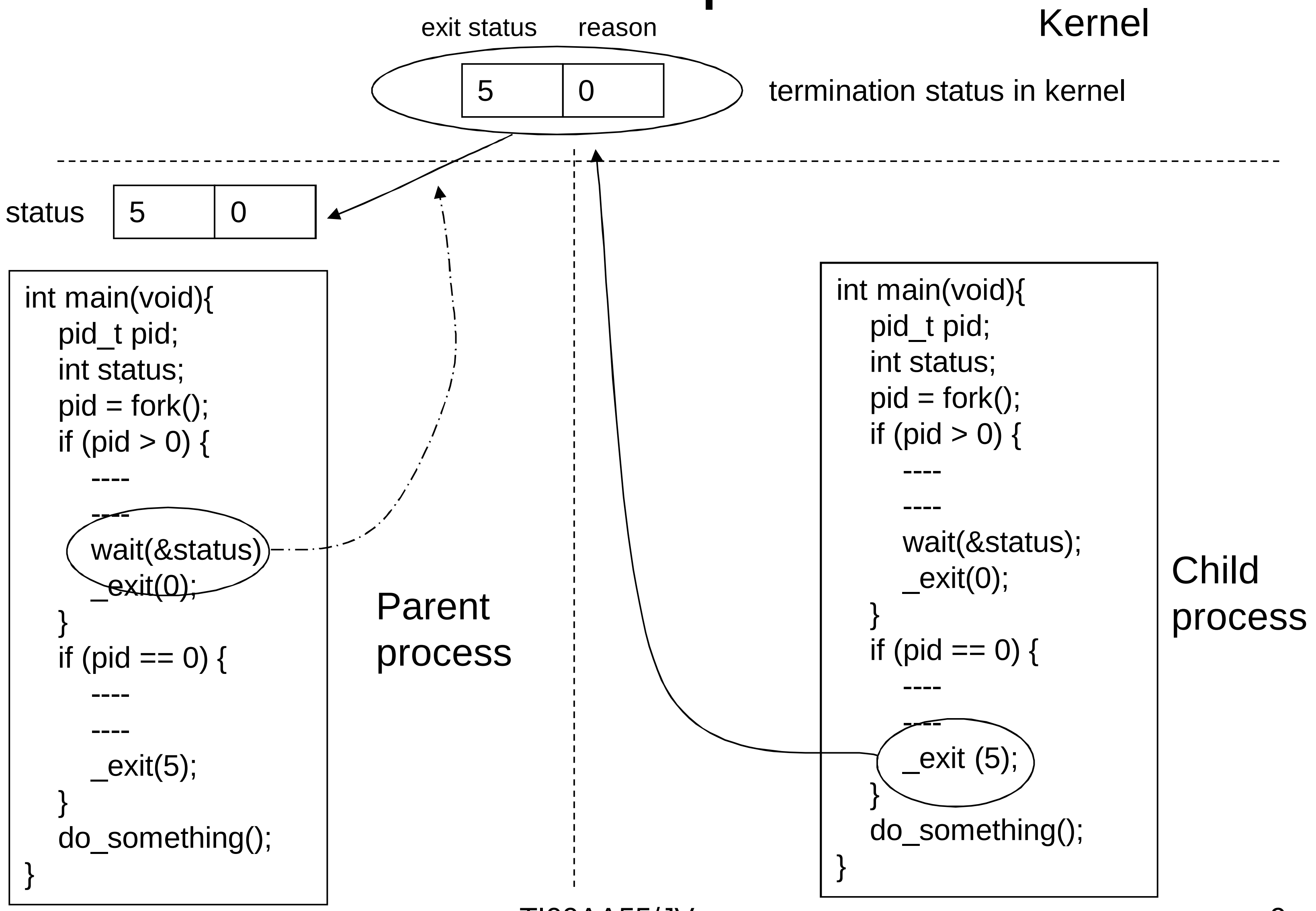
```
#include <pthread.h>
#include <stdlib.h>
void* tf(void* par);

int main(void){
    pthread_t tid;
    int       status;

    pthread_create(&tid, NULL, tf, &status);
    //task of main thread
    ----
    pthread_join(tid, NULL);
    exit(0);
}

void* tf(void* par) {
       // task of another thread
       ----
       *(int*)par = 0;
       pthread_exit(par);
}
```

Passing termination status from the child to the parent



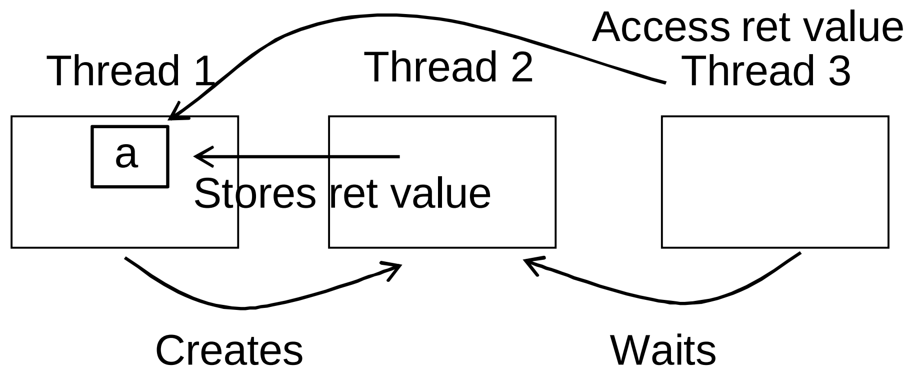
```
int main(void){
    pid_t pid;
    int status;
    pid = fork();
    if (pid > 0) {
        ----
        ----
        wait(&status);
        _exit(0);
    }
    if (pid == 0) {
        ----
        ----
        _exit(5);
    }
    do_something();
}
```

```
int main(void){
    pid_t pid;
    int status;
    pid = fork();
    if (pid > 0) {
        ----
        ----
        wait(&status);
        _exit(0);
    }
    if (pid == 0) {
        ----
        ----
        _exit (5);
    }
    do_something();
}
```

Exit status of thread

- The type of the return value of a thread is void *. This can be used for example in the following ways:
  1. Int value can be converted to the pointer, so that this integer itself indicates the termination status (this is not recommended way)
  2. The terminating thread allocates memory from the dynamic area and stores return information there and then returns the address to this memory area. The problem in this solution is that the waiting thread should free the memory space to avoid memory leaks
  3. The thread that creates the thread (and possibly but not necessarily waits for it) passes the address of the variable of it's own to the new thread. The terminating thread stores return value to this variable and returns it's address to the waiting process

- Remark. The third way works even when the waiting thread is different from the thread that created the terminating thread provided that the original thread is living longer than the two others



Termination of thread vs. process

- Comparing terminations of a thread and a process: A process can consists of several threads. If a process is terminated, it means that all of it’s threads are terminated. A process terminates if
    1. One of it’s threads calls exit
    2. Main thread (main) returns
    3. Last thread of the process terminates (pthread_exit)
- Only the thread terminates if
    1. Thread calls function pthread_exit
    2. Thread returns

More functions for thread handling

- A thread can ask it’s own thread id `pthread_t pthread_self(void);`
- Function returns the thread id of the thread where it was called
- Comparing thread id:s `int pthread_equal(pthread_t t1, pthread_t t2);`
  - If you have two thread id:s, you can find out, if the threads they represent are same or not using the function `pthread_equal`
  - Remark. Note that you should not use the comparison operator ==, because it is not guaranteed that it works
- A thread can terminate (cancel) another thread, if the latter is in the state PTHREAD_CANCEL_ENABLE (that is default state of a thread). Cancelling can be done with the function pthread_cancel `int pthread_cancel(pthread_t thread);`
- A thread can terminate itself using the function `pthread_exit` as we have seen before

Errors of thread functions

- Thread functions return error code directly
  - They do not use global variable (errno)
  - This means that you cannot use function `perror` after calling pthread functions
  - But you still can use function `strerror`
    - Passing error code returned by the pthtread function you can ”convert” the numerical error code into plain text
- Remark. If it is possible that pthread function can fail, then there are only two categories for return values: Only zero means success. All other return values mean error. Negative return values are not usually used at all.
- Example

```
int error;
pthread_t tid;
if(error = pthread_create(&tid, NULL, t_func, NULL))
  fprintf(stderr, "Error message is %s", strerror(error));
```

Thread parameter

- It is possible to specify, what parameter will be (later) passed to the thread function
  - This is done when the thread is created (the last parameter of function `pthread_create`)
- A thread can take one parameter
  - The type of this parameter is void *
  - The parameter is passed to the thread function, when the thread is scheduled first time
- What ever data can be passed to the thread using this parameter, because we can pass an address of simple variable, an address of an array or an address of a structure for example
- When parameters are passed to the thread function, we must be careful that the thread really gets the data, that we want it to get
  - The problem can arise, because we cannot know the moment when the thread really starts to execute
  - It is possible for example that the variable whose address we pass to the thread, does not contain no longer the same value at the moment when the thread starts as it was, when we called `pthread_create` function
- The example on the following page shows, how a thread can pass an address of status variable to the new thread, so that the new thread can store the return value in the variable of the creating thread

Example: thread parameter

- How to pass a parameter to the new thread:

```
void* tf(void*);
int main(void) {
  pthread_t id;
  int      *pstat;
  int       status;

  pthread_create(&id, NULL, tf, &status);
  //…
  pthread_join(id, (void**)&pstat);
  printf("Status is %d\n", *pstat);
}

void* tf(void* p) {
  //…
  *(int*)p = 10;
  pthread_exit(p);
  // or return p;
}
```

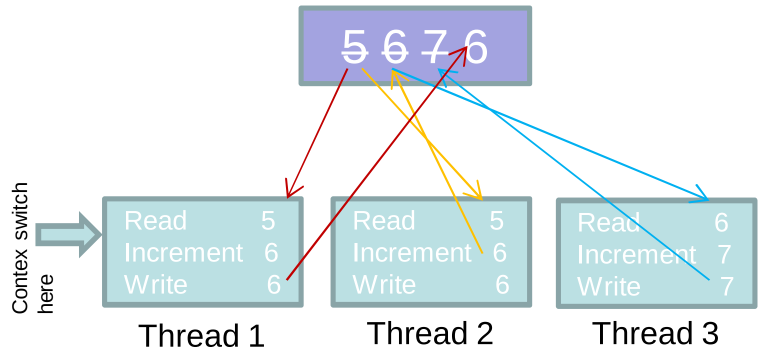
- Remark 1. It would be possible for the main thread to access directly the status variable in it’s own memory area, but here the sub thread passes back the address it has received
- Remark 2. It is worth noting that it would be possible to let a third thread to wait for this new thread and let it to access termination status from the memory area of the main thread (see the figure on the page 10)

Thread synchronization tools (mutex)

- Mutexes are developed for mutual exclusion for critical sections of code
  - They are intended for a protection of critical sections (short time access)
- Example 1
  - Protection of the counter, that is used by several threads (read, increment or decrement, write). See the following page why protection is needed
- Example 2
  - Protection of non thread safe functions
- Example 3
  - Protecting synchronization flags. For example situation where what ever thread can set a stop flag
  - After that it is not allowed for any thread to continue to do a certain thing
- Example 4
  - Protecting data structures like a linked list that is used by several threads
- Mutex handling functions
  `<pthread.h>`
  Initialization macro
```
PTHREAD_MUTEX_INITIALIZER
int pthread_mutex_lock(pthread_mutex_t
*mutex);
int pthread_mutex_trylock(pthread_mutex_t
*mutex);
int pthread_mutex_unlock(pthread_mutex_t
*mutex);
```
- Remark. When mutex has been initialized with the macro macro PTHREAD_MUTEX_INITIALIZER it has default attributes
```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

Example of the problem

- Lets examine more carefully the problem of the example 1 on the previous page
  - Three threads increment the same counter
  - In the beginning the counter value is 5
- The following chain of events is possible:
  - Thread 1 reads the counter (5)
  - Context switch happens
  - Thread 2 increments (reads, increments and writes back) the counter (6)
  - Thread 3 increments the counter (7)
  - Turn to run is returned back to the thread1
    - It increments the value 5, it has read already earlier and writes back the value 6



- This means that we loose the effect of the additions in threads 2 and 3 !!
- We need mutex to guarantee that the three steps needed in the additions are done atomically:

```
pthread_mutex_lock(&mutex);
counter++;
pthread_mutex_unlock(&mutex);
```

Basic idea of mutex

- Mutex is a variable that has two possible values:
  - 0 and 1
  - Zero means that the resource (we want to protect) is free and can be accessed
  - One means that the resource is reserved
- If a thread wants to access the resource, it calls `pthread_mutex_lock`
- If the resource is free, the function increments the mutex value (so that it indicates that the resource is reserved) and returns immediately
  - Increment (read, increment, write) of the mutex value is done atomically
- If the resource is reserved, the function blocks and waits until the mutex is freed
- When the thread has done the critical task, it must unlock the mutex by calling the function `pthread_mutex_unlock`
  - This function decrements the mutex value (so that it indicates that resource is free again)
  - Decrement (read, decrement, write) of the mutex value is done atomically
- Remark. According POSIX, the thread that has locked the mutex should also unlock it
  - In other words, a thread must not unlock the mutex that was locked by another thread

Condition variables and events

- Condition variables can be used to wait for the event when some condition becomes true (instead of using a so called busy loop)
- Condition variables are always used with a mutex
- The type of the condition variable is `pthread_cond_t`
- The most important functions for handling condition variables: Initialization macro

```
PTHREAD_COND_INITIALIZER
int pthread_cond_wait(pthread_cond_t
*cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t
*cond);
```

- The basic idea of function `pthread_cond_wait` is that it suspends the thread and unlocks the mutex atomically
  - The function returns when the condition variable is set to signalled state with the function `pthread_cond_signal`
  - This is done in some other thread
  - When function `pthread_cond_wait` returns the mutex is automatically locked again

Example of using condition variable

- Let’s consider the situation where a sub-thread increments the counter and the main-thread waits until the counter reaches a given value
- Thread that tests a condition:

```
pthread_mutex_lock(&mutex);
  while (counter < 100)
    pthread_cond_wait(&cond, &mutex);
// use counter
pthread_mutex_unlock(&mutex);
```

- Thread that modifies the condition:

```
pthread_mutex_lock(&mutex);
counter++;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

- In this case it is guaranteed that, when the thread that tests the condition uses the counter, it’s value is still the same than it was when it came out from the while loop (greater that or equal than 100)

Threads and signals

- Signals used by the system (for example SIGCHLD, SIGALRM, SIGIO) are delivered to the process
  - In similar way, signals that are sent by the functions `kill`, `raise` or `sigqueue` are delivered to the process
  - They cannot be delivered to a specific thread
- On the other hand, only one thread receives the signal
  - According the POSIX, the thread that receives the signal can be what ever of the threads of the process
- This means for example, that a signal can interrupt a system call in what ever thread
  - This can be a problem because threads are usually used to let system calls to block as we saw in the beginning!
- Because of the things mentioned above we need new features for signal handling in threads
  - Thread library contains functions that can be used for example to send a signal to the specific thread and to set a signal mask for a specific thread
- Most important signal related thread library functions are

```
int pthread_kill(pthread_t thread, int
sig);
int pthread_sigqueue(pthread_t
*thread, int sig, const union sigval
value);
int pthread_sigmask(int how, const
sigset_t *set, sigset_t *oldset);
```

Signal handling thread

- We have seen that signals in general can cause many kind of problems because of their asynchronous nature
- It is possible to avoid these problems by handling all signals in one thread dedicated to this purpose
  - Signals are handled synchronously (not anymore asynchronously)
- Basic idea is that we block all signals in other threads and use `sigwait` function to wait for all signals in a signal handling thread
- This makes it very easy to handle signals, because we do not need to write signal handler functions at all
  - We can use the function `sigwait` and let it block in this special thread
  - Function `sigwait` returns when one of the signals indicated in the first parameter is delivered
  - The second parameter tells what signal caused the `sigwait` function to return
- See the example program signal_handler_thread_1.c
- Remark. Signals do not interrupt any pthread function
  - This is mentioned specifically in the standard (for example, waiting another thread with a function `pthread_join`, is not interrupted)

Thread attributes 1

- A thread has attributes that specifies certain properties of the thread
  - Attributes are given when the thread is created with the function `pthread_create`
  - The second parameter of this function specifies attributes
  - The type of the parameter is `const pthread_attr_t *`
- If this parameter is NULL, default attributes are used
- Type `pthread_attr_t` is a structure having members that determine different properties of the thread, like
  - detach properties of the thread (state)
  - the size of the stack
  - scheduling properties of the thread
  - Etc.
- Attribute is an abstract data type
  - It has several functions that are used to set and get the properties
- If you need to set a certain properties in the creation of the thread the procedure is as follows:
  - Initialize the attribute variable with the function `pthread_attr_init(pthread_attr_t *attr);`
  - Apply different function to set different properties to the attribute variable
  - Pass the address of the attribute variable to the function `pthread_create`
    - The properties you have set to the attribute variable are the properties of the thread

Thread attributes 2

- Functions that are used to set properties to the attribute are all of the following format:
```
int
pthread_attr_setXXXX(pthread_attr_t
*attr, int feature);
```
- Remark 1. It is also possible to ask the properties that are currently set in the attribute variable. For this purpose there corresponding functions of the format:
```
int pthread_attr_getXXXX(const
pthread_attr_t *attr, int *feature);
```
- The following two ways to create a thread lead to the same result so that in both cases the threads have same properties.
- Way 1
```
pthread_t id;
pthread_create(&id, NULL, threadFunc,
NULL);
```
- Way 2
```
pthread_t id;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_create(&id, &attr, threadFunc,
NULL);
```

Example: Using thread attributes

```
int main(void) {
  pthread_t id;
  int r, detachstate;
  pthread_attr_t attr;
  size_t stacksize;

  pthread_attr_init(&attr);
  /* Get and set thread detached attribute */
  pthread_attr_getdetachstate(&attr, &detachstate);
  if (detachstate == PTHREAD_CREATE_JOINABLE)
    printf("Detachstate: PTHREAD_CREATE_JOINABLE");
  else if (detachstate == PTHREAD_CREATE_DETACHED)
    printf("Detachstate: PTHREAD_CREATE_DETACHED");
  else
    printf("Detachstate is not what is should be\n");
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  //Options:PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE

  /* Get and set thread stack size attribute */
  pthread_attr_getstacksize(&attr, &stacksize);
  printf("Default stack size = %ld\n", stacksize);
  stacksize = 20000000;
  pthread_attr_setstacksize(&attr, stacksize);

  pthread_create(&id, &attr, threadFunction, NULL);
  /* Free attribute and wait for the thread */
  pthread_attr_destroy(&attr);
  r = pthread_join(&id, NULL); // causes an error if detached
  if (r != 0)
    printf("Join error: %s\n", strerror(r));  // Invalid argument
  else
    printf("Join succeeded\n");
  return 0;
}
```

Thread scheduling (contention scope)

- Threads compete for the processor time (for turns to run)
  - The competition can be system wide or process wide. This is also indicated in the thread attribute
  - This property can be set to the attribute with the function
    ```
    int
    pthread_attr_setscope(pthread_att
    r_t *attr, int contentionscope);
    ```
  - This property can be asked from the attribute with the function
    ```
    int pthread_attr_getscope(const
    pthread_attr_t *attr, int
    *contentionscope);
    ```
- The possible values for the parameter contentionscope according POSIX are
  - PTHREAD_SCOPE_SYSTEM or
  - PTHREAD_SCOPE_PROCESS
- In Linux, only PTHREAD_SCOPE_SYSTEM is used
  - This means that each thread is scheduled like a process
  - In fact, in Linux a thread has been implemented in very similar way than a process
  - Both of them use a lower level system call clone
- The consequence is that the effect is the same, weather you use scheduling functions of the process or scheduling functions of the thread for the main thread in Linux
- Remark. Contention scope can be asked and set also directly from/to the thread (see page 25).

Thread scheduling (scheduling policy)

- The kernel uses different policies in the scheduling
  - Scheduling policy is also indicated in the thread attribute
- The following function is used to set the scheduling policy to the thread attribute

```
pthread_attr_setschedpolicy(
pthread_attr_t *attr, int
policy);
```

- The following function is used to get the scheduling policy from the thread attribute

```
int
pthread_attr_getschedpolicy(pthre
ad_attr_t *attr, int *policy);
```

- The possible values for the parameter policy (the possible scheduling policies) are:
  - SCHED_OTHER (Default policy in Linux))
  - SCHED_FIFO (First In First Out)
  - SCHED_RR (Round Robin)
- The default scheduling method in Linux is SCHED_OTHER
- The latter two policies are actual real time scheduling methods

Thread scheduling (scheduling priority)

- The scheduling policies FIFO and RR have priority
- They are set to and get from the attribute with functions
```
pthread_attr_setschedparam(
pthread_attr_t *attr, const struct
sched_param *param);
int
pthread_attr_getschedparam(pthread_att
r_t *attr, struct sched_param *param);
```
- These functions have address of the structure struct sched_param as a parameter
  - This structure has a member `sched_priority` that tells the priority
- The struct sched_param is defined as follows:
```
struct sched_param {
      ...
  int sched_priority;
      ...
};
```
- The priority is between 1 – 99 for FIFO and RR policies.

- Now we have seen how to set and get scheduling policy and scheduling priority to/from the thread attribute.
- These things can also be set to and get directly from the thread without the attribute variable
- These functions are:
```
pthread_setschedparam(pthread_t
thread, int policy, const struct
sched_param *param);
pthread_getschedparam(pthread_t
thread, int *policy, struct
sched_param *param);
```

Process scheduling

- In Linux threads are implemented as light weight processes
  - Threads in one process are equal in the scheduling to the threads in another process even if processes can have a different number of threads
    - If there is only one thread in the process, you can set and get scheduling parameters using corresponding functions of the process
- The scheduling policy of the process can be asked with the function
  `int sched_getscheduler(pid_t pid);`
- The function returns scheduling policy
- The scheduling policy of the process and the priority can be set with the function
  `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);`

Scheduling functions examples

```
void set_scheduling(int policy, int priority) {
  struct sched_param param;
  int policy_asked;

  param.sched_priority = priority;
  //sched_setscheduler(0, SCHED_RR, &param);
  if (sched_setscheduler(0, policy, &param) < 0 ) {
    perror("sched_setscheduler ");
    exit(0);
  }
}
```

```
void change_priority(int new_priority) {
  struct sched_param param;

  sched_getparam(0, &param);
  param.sched_priority = new_priority;
  if (sched_setparam(0, &param) < 0) {
    perror("sched_setparam ");
    exit(0);
  }
}
```

```
void show_scheduling ( void )  {
  struct sched_param param;
  int policy_asked;
  policy_asked = sched_getscheduler(0);
  if (policy_asked == SCHED_OTHER)
    printf("SCHED_OTHER\n");
  else if (policy_asked == SCHED_FIFO)
    printf("SCHED_FIFO\n");
  else if (policy_asked == SCHED_RR)
    printf("SCHED_RR\n");
  else
    printf("Something that should not be possible\n");
  sched_getparam(0, &param);
  printf("Scheduling priority is %d\n", param.sched_priority);
}
```

More process scheduling functions

- The scheduling parameters of the process can be asked with the function
  ```
  int sched_getparam(pid_t pid, struct sched_param *param);
  ```
- The scheduling parameters of the process can be set with the function
  ```
  int sched_setparam(pid_t pid, const struct sched_param *param);
  ```
- The last parameter is again the same structure containing the member sched_priority
- Time quantum that is used in Round Robin real time scheduling can be determined with the function sched_rr_get_interval. The prototype is
  ```
  int sched_rr_get_interval(pid_t pid, struct timespec *interval);
  ```

Mutex attributes 1

- We have used initialization macro `PTHREAD_MUTEX_INITIALIZER` to initialize a mutex. It gives a mutex default attributes
- If default attributes are not suitable in the application, you can use function `pthread_mutex_init` to initialize a mutex
  - It takes the pointer parameter to the variable of type `pthread_mutexattr_t` that contains mutex attributes
- If mutex has been initialized this way, you also need to remember to call function `pthread_mutex_destroy`, when you do not anymore need the attribute
- Prototypes of these functions are:

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

- Mutex attributes are priority ceiling, protocol, process-shared and mutex type
- The first two are used for “fine tuning” the usage of mutexes in FIFO and RR scheduling

Mutex attributes 2

- Process-shared attribute has two possible values
  - PTHREAD_PROCESS_PRIVATE (default) and
  - PTHREAD_PROCESS_SHARED
- The latter option means that the mutex can be used between threads in separate process
- By default mutexes can only be used between threads in the same process

- Mutex type has four possible values
  - PTHREAD_MUTEX_NORMAL,
  - PTHREAD_MUTEX_ERRORCHECK,
  - PTHREAD_MUTEX_RECURSIVE,
  - PTHREAD_MUTEX_DEFAULT (default).
- The default option means that:
  - Attempting to recursively lock a mutex of this type results in undefined behavior
  - Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior
  - Attempting to unlock a mutex of this type which is not locked results in undefined behavior

Thread cancellation 1

- Threads have a so called **cancellation state.** Options are
  - PTHREAD_CANCEL_ENABLE (default) or
  - PTHREAD_CANCEL_DISABLE
- A thread can cancel another thread if the thread to be cancelled is in the state PTHREAD_CANCEL_ENABLE
- Cancellation is done with the function pthread_cancel. The prototype is
  `int pthread_cancel(pthread_t thread);`
- A thread can change it's own cancellation state with the function pthread_setcancelstate. The prototype is
  `int pthread_setcancelstate(int state, int *oldstate);`
- There is also another feature regarding cancellation: **cancellation type**. Cancellation type is either
  - PTHREAD_CANCEL_DEFERRED (default)
  - or PTHREAD_CANCEL_ASYNCHRONOUS
- A thread can change it's own cancellation type with the function `pthread_setcanceltype`. The prototype is
  `int pthread_setcanceltype(int type, int *oldtype);`
- If the cancellation type is
  A. THREAD_CANCEL_DEFERRED, the thread does not terminate immediately
     - It terminates when it reaches the next cancellation point

Thread cancellation 2

- It is possible that a thread has no cancellation point. In that case the thread does not terminate at all even if cancellation request has been done
- Cancellation point are for example
  - pthread_join
  - pthread_cond_wait
  - pthread_cond_timedwait
  - pthread_testcancel
  - sem_wait
  - sigwait
- Note the function `pthread_testcancel` in the list
  - Function tests weather a cancellation request has been done
  - If there is a cancellation request, the thread terminates
  - This way it is possible to affect where termination happens.
- Remark 1. Slow blocking system calls are also cancellation points.
- Remark 2. If a thread is waiting for a mutex (`pthread_mutex_lock`), it is not a cancellation point!
- If the cancellation type is
  - B. THREAD_CANCEL_ASYNCHRONOUS, termination happens in any situation immediately
- Mention `pthread_cleanup_push`