The pthread library

- In Linux, when a new process is created, it already contains a thread, used to execute the main() function Additional threads can be created using the pthread library, which is part of the C library.
- Of course all threads inside a given process will share the same address space, the same set of open files, etc.
- The pthread library also provide thread synchronization primitives: mutexes and conditions.
- This pthread library has its own header : pthread.h
- Applications using pthread function calls should be explicitly linked with the pthread library: gcc o app app.c lpthread

Creating a new thread:

The function to create a new thread is pthread_create()

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

- *thread is a pointer to a pthread_t structure that will be initialized by the function. Later, this structure can be used to reference the thread.*
- *Attr is a pointer to an optional structure pthread_attr_t. This structure can be manipulated using pthread_attr_*() functions. It can be used to set various attributes of the threads (detach policy, scheduling policy, etc.)*
- *start_routine is the function that will be executed by the thread*
- *arg is the private data passed as argument to the start_routine function.*

Thread creation, code sample:

```
#include <pthread.h>
void *thread(void *data)
{
        while(1) {
                printf(« Hello world from thread »);
        }
}

int main(void) {
        pthread_t th;
        pthread_create(& th, NULL, thread, NULL);
        return 0;
}
```

Joinable and detached threads

- When the main() function exits, all threads of the application are destroyed.
- The pthread_join() function call can be used to suspend the execution of a thread until another thread terminates. This function must be called in order to release the ressources used by the thread, otherwise it remains as zombie.
- Threads can also be detached, in which case they become independent. This can be achieved using.
    - Thread attributes at thread creation, using:
      `pthread_attr_setdetachstate(& attr, PTHREAD_CREATE_DETACHED);`
    - pthread_detach(), passing the pthread_t structure as argument.

Thread join, code sample

```
#include <pthread.h>
void *thread(void *data)
{
        int i;
        for (i = 0; i < 100; i++) {
        printf(« Hello world from thread »);
        }
}

int main(void) {
        pthread_t th;
        pthread_create(& th, NULL, thread, NULL);
        pthread_join(& th, NULL);
        return 0;
}
```

Thread cancelation

- It is also possible to cancel a thread from another thread using the pthread_cancel() function, passing the pthread_t structure of the thread to cancel.

```
#include <pthread.h>
void *thread(void *data)
{
	while(1) {
		printf(« Hello world from thread »);
	}
}
int main(void) {
	pthread_t th;
	pthread_create(& th, NULL, thread, NULL);
	sleep(1);
	pthread_cancel(& th);
	pthread_join(& th, NULL);
	return 0;
}
```

pthread mutexes

- The pthread library provides a mutual exclusion primitive, the pthread_mutex.

- Declaration and initialization of a pthread mutex.

Solution 1: at definition time: pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Solution 2: at runtime: pthread_mutex_t lock;

```
...
pthread_mutex_init(& lock, NULL);
...
pthread_mutex_destroy(& lock);
```

- The second argument to pthread_mutex_init() is a set of mutexspecific attributes, in the form of a pthread_mutexattr_t structure that can be initialized and manipulated using pthread_mutexattr_*() functions.

- Take the mutex: ret = pthread_mutex_lock(& lock);

- If the mutex is already taken by the calling threads, three possible behaviours depending on the mutex type (defined at creation time)

- ***Normal (« fast ») mutex**: the function doesn't return, deadlock*
- ***« Error checking » mutex**: the function return with the EDEADLK error*
- ***« Recursive mutex »**: the function returns with success*

- Release the mutex: ret = pthread_mutex_unlock(& lock);

- Try to take the mutex: ret = pthread_mutex_trylock(& lock);

pthread conditions

- Conditions can be used to suspend a thread until a condition becomes true, as signaled by another thread.

- Initialization, static or dynamic

- *pthread_cond_t cond = PTHREAD_COND_INITIALIZER;*
- *pthread_cond_t cond;*
- *pthread_cond_init(& cond, NULL);*

- Wait for the condition: pthread_cond_wait(& cond, & mutex): The mutex will be released before waiting and taken again after the wait.

- Signaling the condition

- *To one thread waiting: pthread_cond_signal(& cond);*
- *To all threads waiting: pthread_cond_broadcast(& cond);*

pthread conditions example:

Receiver side:

```
pthread_mutex_lock(& lock);
while(is_queue_empty())
	pthread_cond_wait(& cond, & lock);
/* Something in the queue, and we have the mutex ! */
pthread_mutex_unlock(& lock);
```

Sender side:
pthread_mutex_lock(& lock);
/ Add something to the queue */*
pthread_mutex_unlock(& lock);
pthread_cond_signal(& cond);

POSIX shared memory

- A great way to communicate between processes without going through expensive system calls.
- Open a shared memory object:

- *shm_fd = shm_open("acme", O_CREAT | O_RDWR, 0666);*
  A zero size /dev/shm/acme file appears.

- Set the shared memory object size:

- *ftruncate(shm_fd, SHM_SIZE);*
  /dev/shm/acme is now listed with the specified size.

- If the object has already been sized by another process, you can get its size with the fstat function.
- Map the shared memory in process address space:

- *addr = mmap (0, SHM_SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);*
  Now we have a memory area we can use!

- Lock the shared memory in RAM (best for realtime tasks):

- *mlock(addr, SHM_SIZE);*

- Use the shared memory object! Other processes can use it too.

Exiting:

- Unmap the shared memory object:

- *munmap (addr, SHM_SIZE);*
  This automatically unlocks it too.

- Close it:

- *close (shm_fd);*

- Remove the shared memory object:

- *shm_unlink ("acme");*
  The object is effectively deleted after the last call to shm_unlink.

POSIX message queues:

Deterministic and efficient IPC. See man mqueue.h. Advantages for realtime applications:
- Preallocated message buffers
- Messages with priority. A message with a higher priority is always received first.
- Send and receive functions are synchronous by default. Possibility to set a wait timeout to avoid nondeterminism.
- Support asynchronous delivery notifications.

Creating and opening a message queue

- Declare queue attributes:

- *queue_attr.mq_maxmsg = 16; /* max number of messages in queue */*
- *queue_attr.mq_msgsize = 128; /* max message size */*

- Open a queue:

- *qd = mq_open("/msg_queue", OCREAT | O_RDWR, 0600, &queue_attr);*

- Posting a message:

- *#define PRIORITY 3*
  char msg[] = "Goodbye Bill";
  mqsend(qd, msg, strlen(msg), PRIORITY);

- Closing the queue:

- *mq_close(qd);*

Receiving a message:

- Opening the shared message queue:

- *qd = mq_open("/msg_queue", O_RDWR, 0600, NULL);*

- Waiting for a message:

- *mq_receive(qd, text, buf, buf_size, &prio);*

- Close the queue:

- *mq_close(qd);*

- Destroy the queue:

- *mq_unlink("/msg_queue");*

POSIX semaphores:

Resources for sharing resources between threads or processes. See man semaphore.h.

- Named semaphores: can be used between unrelated processes.
- Unnamed semaphores: can be used between threads from the same process, or by related processes (parent / child).

- Open and / or create a named semaphore:
    - sem_open();

- Close a named semaphore:
    - sem_close();

- Destroy a named semaphore:
    - sem_unlink();

- Initialize an unnamed semaphore:
    - sem_init();

- Destroy an unnamed semaphore:
    - sem_destroy();

- Get current semaphore count:
    - sem_getvalue();

-Try to lock the semaphore. Wait otherwise:
    - sem_wait();

- Just tries to lock the semaphore, but gives up if the semaphore is already locked:
    - sem_trywait();

- Release the semaphore:
    - sem_post();

POSIX signals:

- Signals are a mechanism to notify a process that an event occured : expiration of a timer, completion of an asynchronous I/O operation, or any kind of event specific to your application
- Signals are also used internally by the system to tell a process that it must be suspended, restarted, stopped, that is has done an invalid memory reference, etc.
- Each signal is identified by a number : SIGSEGV, SIGKILL, SIGUSR1, etc.
- An API is available to catch signals, wait for signals, mask signals, etc.

Registering a signal handler

- A signal handler can be registered using: sighandler_t signal(int signum, sighandler_t handler);

⇒ The handler has the following prototype : void handler(int signum)

- int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

⇒ The sigaction structure contains the reference to the handler

⇒ The handler can have two different prototypes:

    -void handler(int signum)
    -void handler(int signum, siginfo_t *info, void *data)

- Inside the handler code, only some functions can be used : only the asyncsignalsafe functions.

Signal registration example:

```
#include <signal.h>
#include <assert.h>
#include <unistd.h>
#include <stdio.h>

void myhandler(int signum){

        printf("Signal catched!\n");

}

int main(void){
        int ret;
        struct sigaction action = {
                .sa_handler = myhandler,
        };
        ret = sigaction(SIGUSR1, & action, NULL);
```

```
        assert(ret == 0);
        while(1);
        return 0;
}
```

⇒ From the command line, the signal can then be sent using kill USR1 PID

Sending a signal:

- From the command line, with the famous kill command, specifying the PID of the process to which the signal should be sent. By default, kill will send SIGTERM. Another signal can be sent using kill USR1.
- POSIX provides a function to send a signal to a process: `int kill(pid_t pid, int sig);`

⇒ In a multithread program, the signal will be delivered to an arbitrary thread. Use tkill() to send the signal to a specific thread.

Signal sets and their usage

- A type sigset_t is defined by POSIX, to hold a set of signals. This type is manipulated through different functions:
    - sigemptyset() to empty the set of signals
    - sigaddset() to add a signal to a set
    - sigdelset() to remove a signal from a set
    - sigfillset() to fill the set of signals with all signals
- Signals can then be blocked or unblocked using: `sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
- sigset_t are also used in many other functions:
    - sigaction() to give the list of signals that must be blocked during execution of the handler.
    - sigpending() to get the list of pending signals

Waiting for signals:

2 ways of waiting for signals:

- sigwaitinfo() and sigtimedwait() to wait for blocked signals (signals which remain pending until they are processed by a thread waiting for them.)
- sigsuspend() to register a signal handler and suspend the thread until the delivery of an unblocked signal (which are delivered without waiting for a thread to wait for them).

POSIX clocks and timers

Compared to standard (BSD) timers in Linux

- -Possibility to have more than 1 timer per process.
- -Increased precision, up to nanosecond accuracy
- -Timer expiration can be notified either with a signal or with a thread.
- -Several clocks available.

Available POSIX clocks:

Defined in /usr/include/linux/time.h

- CLOCK_REALTIME: Systemwide clock measuring the time in seconds and nanoseconds since Jan 1, 1970, 00:00. Can be modified. Accuracy: 1/HZ (1 to 10 ms)
- CLOCK_MONOTONIC: Systemwide clock measuring the time in seconds and nanoseconds since system boot. Cannot be modified, so can be used for accurate time measurement. Accuracy: 1/HZ
- CLOCK_PROCESS_CPUTIME_ID: Measures process uptime. 1/HZ accuracy. Can be changed.
- CLOCK_THREAD_CPUTIME_ID: Same, but only for the current thread.

Time management:

Functions defined in time.h

- clock_settime: Set the specified clock to a value
- clock_gettime: Read the value of a given clock
- clock_getres: Get the resolution of a given clock.

Using timers

Functions also defined in time.h

- clock_nanosleep: Suspend the current thread for the specified time, using a specified clock.
- nanosleep: Same as clock_nanosleep, using the CLOCK_REALTIME clock.
- timer_create: Create a timer based on a given clock.
- timer_delete: Delete a timer
- timer_settime: Arm a timer.
- timer_gettime: Access the current value of a timer.

Asynchronous I/O:

- Helpful to implement nonblocking I/O. Allows to overlap compute tasks with I/O processing, to increase determinism.
- Supported functionality:

-Send multiple I/O requests at once from different sources

-Cancel ongoing I/O requests

-Wait for request completion

-Inquire the status of a request: completed, failed, or in progress.

Compiling instructions:

- Includes: nothing special to do. Available in the standard path.

- Libraries: link with librt. Example: gcc lrt o rttest rttest.c