POSIX threads basics

Note: the slides do not cover thread-specific data, thread scheduling, RT threads

Last modification date: 16.11.2020

POSIX definitions

- **Process**- An address space with one or more threads executing within that address space, and the required system resources for those threads/ Many of the system resources are **shared** among all of the threads within a process. These include the process ID, the parent process ID, process group ID, session membership, real, effective, and saved set-user-ID, real, effective, and saved set-group-ID, supplementary group.
- **Thread -** A single **flow of control within a process**. Each thread has its **own** thread ID (TID), scheduling priority and policy, **`errno`** value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via **`malloc( )`**, directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.
- **Multithreaded program** – a program whose executable file was produced by compiling with c99 using the flags output by **`getconf POSIX_V7_THREADS_CFLAGS`**, and linking with c99 using the flags output by **`getconf POSIX_V7_THREADS_LDFLAGS`** and the **`-l pthread`**, or by compiling and linking using a non-standard utility with equivalent flags. Execution of a multi-threaded program initially creates a single-threaded process; the process can create additional threads using **`pthread_create()`** or SIGEV_THREAD notifications.

POSIX threads (P-threads) API

Header file	Prefixes of API symbols
<pthread.h>	pthread_ , PTHREAD_

Thread creation:

```
int pthread_create(pthread_t *thread,   // returns TID via *thread
        const pthread_attr_t *attr,     // ptr to  thread attributes structure
        void *(*start_routine)(void *), // working function
        void *arg                       // working function argument (can be NULL)
```

- `);` // returns **0 on success** non-zero on error. According to Linux **man pthreads** "**the pthreads functions do not set errno**". Yet currently errno is assigned the pthread function error code value;

The new thread starts executing by invoking `start_routine(arg)` .

Thread attributes can be set only at creation time, except detach state. A single attributes object (`struct pthread_attr_t`) can be used in multiple simultaneous calls to `pthread_create()` .

Attributes

- The attribute structure has to be initialized prior to use:

```
int pthread_attr_init(pthread_attr_t *tattr);
```

- Getting and setting individual attributes

```
int pthread_attr_getXXX(pthread_attr_t *tattr,...);
int pthread_attr_setXXX(pthread_attr_t *tattr,...);
```

Attribute (XXX)	value	meaning
detachstate	`PTHREAD_CREATE_JOINABLE`	Default, joinable state
	`PTHREAD_CREATE_DETACHED`	Non-joinable (detach) state
scope	`PTHREAD_SCOPE_PROCESS`	Thread competes for resources with same process threads
	`PTHREAD_SCOPE_SYSTEM`	… competes with all threads

Other attributes (see pthread_attr_init(3)):

- `inheritsched, schedpolicy, schedparam`
- `stack, guardsize`,

The attribute structure can be invalidated after use:

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

Passing parameters to threads (1)

- Method 1. each thread gets address of its specific parameter value. Correct (safe) technique.

  Example: passing a simple variable (here: **int**)

```
void * worker(void *argp){
        int arg=*(int *)argp; // copying argument to arg
        printf("arg=%d\n",arg);
}
int main(...){
pthread_t threads[NUM_THREADS];
int param[NUM_THREADS]; // array of thread arguments
.....
        for(n=0; n<NUM_THREADS; n++){
                param[n]=n;
                rc = pthread_create(&threads[n], NULL,
                        worker, (void *) &param[n]);
        }
        ...
}
```

Passing parameters to threads (1 – cont.)

Method 1 can be also used to pass complex arguments (structs, unions) and to receive results from the thread worker function.

```
struct mystruct{ double in; double out;}; // in/out structure
void * worker(void *argp){/
    struct mystruct *inout=*((struct mystruct *)argp);
        inout->out=sqrt(inout->in);
        printf("in=%g, out=%g\n",inout->in, inout->out);
        return NULL;
}
int main(...){
    pthread_t threads[NUM_THREADS];
    struct mystruct args[MAXPTHREADS];
        for(n=0; n<NUM_THREADS; n++){
                args[n].in=(double)(n+1);
                if (pthread_create(&threads[n], NULL,
                        worker, &args[n])){ ... };
        }
        for(n=0; n<NUM_THREADS; n++){
                if (pthread_join(threads[n],NULL)==0)
                    printf("n=%d, in=%g, out=%g\n",
                            n,args[n].in,args[n].out);
        }
.....
}
```

Passing parameters to threads (2)

- Method 2. Passing an argument **arg** via cast to **void *** . The worker has to do cast in the opposite direction into the **arg** type.

Warning: the method **works correctly only if** **sizeof(arg)<=sizeof(void *)**

```
int param[NUM_THREADS]; // array of thread arguments
void * worker(void *argp){
        int arg=(int)argp; // copying argument to arg
        printf("arg=%d\n",arg);
        return NULL;
}
int main(...){
pthread_t threads[NUM_THREADS];
.....
        assert( (sizeof(int)<=sizeof(void *) );
        for(int n=0; n<NUM_THREADS; n++){
                param[n]=n;
                rc = pthread_create(&threads[n], NULL,
                        worker, (void *)n);
        }
        ...
}
```

Passing parameters to threads - bad practice

- **WARNING**: incorrect technique: a naïve passing address of a simple variable (here: **int**), that is modified within lifetime of a thread.

```
void * worker(void *argp){
        int arg=*((int *)argp); // copying argument to arg
        printf("arg=%d\n",arg);
        return NULL;
}
int main(...){
pthread_t threads[NUM_THREADS];
int arg;
.....
        for(n=0; n<NUM_THREADS; n++){
                arg=n;
                rc = pthread_create(&threads[n], NULL,
                        worker, (void *) &arg);
        }
        ...
}
```

Ending thread execution

A thread ends when:

- this thread (working) function **makes return**
- the thread function calls `pthread_exit:`

```
void pthread_exit(void *status);
```

Note: `status` should be NULL or be an address of a memory location which is accessible by a thread which is to receive the memory location via `pthread_join()` call.

- the thread was cancelled via a call to `pthread_cancel:`

```
void pthread_cancel(pthread_t thread);
```

- The whole process is terminated. This can happen when
  - the main thread (`main()` function) of a POSIX process **returns** or
  - `exit()` function is called by any thread of the process
  - a signal delivery (or `abort()` call) resulted in process termination

.

Waiting for a thread termination

- A thread of a process can wait for another thread of the same process:
  - if it knows TID of the awaited thread and
  - the awaited thread **is not detached**

```
int pthread_join(
        pthread_t tid,  // TID of the awaited thread
        void **status  // address of  the pointer to be
                       // returned (NULL → status is discarded)
);  // returns 0 on success;
// ESRCH – if the thread does not exist,
// EINVAL – when the thread is detached (so cannot be joined).
```

Making thread detached:

- Using `detachstate` attribute when creating a thread.
- Calling (by any thread of the processes):

```
int pthread_detach(
        thread_t tid  // TID of the thread to be detached
);
```

Return value of a thread function

- Thread function returns address of return value, so the address **should be valid after the function returns**. An alternative way: `pthread_exit()` call.

Correct

```
void *thread1(void *arg){
   int i=*((int *)arg), *ip;
   i++;
   ip = malloc(sizeof(int));
   if (ip) *ip = i;
   else { . . . } // failure
   return (void *)ip;
}

void *thread2(void *arg){
   static int i=*((int *)arg);
   i++;
   return (void *)&i;
}
```

Incorrect

```
void *thread3(void *arg){
   int i=*((int *)arg);
   i++;
   return (void *)&i;
}
```

- It is easy (and safe) to retrieve data produced by a thread via a global value or via a struct argument - which keeps both input as output data items. The beginners are advised to do so, and return NULL from thread function

Thread cancellation

- The thread cancellation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner.
- The **cancelability state** of a thread determines the action taken upon receipt of a cancellation. The state has two components:
  - **cancelability** (set set to **PTHREAD_CANCEL_ENABLE**, by default)
    - can be changed to **PTHREAD_CANCEL_ DISABLE** by `pthread_setcancelstate` function call. In effect cancellation requests are held pending.
  - **cancelability type** (set to **PTHREAD_CANCEL_ DEFERRED** in all newly created threads)
    - When cancelability is enabled and the cancelability type is **PTHREAD_CANCEL_ DEFERRED** cancellation requests are **held pending until a cancellation point is reached**.
    - When cancelability is enabled and the cancelability type is **PTHREAD_CANCEL_ ASYNCHRONOUS**, new or pending cancellation requests may be acted upon **at any time**. Only **async-cancel-safe functions** should be used by a thread with asynchronous cancelability.
    - Change of cancelability type: by `pthread_setcanceltype` function call.

Cancellation points (POSIX_std_1003.1-2017)

The following functions are required to be cancellation points (see pthreads(7) for other):

accept()
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl()†
fdatasync()
fsync()
getmsg()
getpmsg()
lockf()††
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat()
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()
putmsg()
putpmsg()
pwrite()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_timedwait()
sem_wait()
send()
sendmsg()
sendto()
sigsuspend()
sigtimedwait()
sigwait()
sigwaitinfo()
sleep()
tcdrain()
wait()
waitid()
waitpid()
write()
writev()

Note: function *pthread_testcancel()* is used as cancellation point only

Clean-up handlers

- Cancellation of a thread can lead to inconsistent application state (semaphores, mutexes and conditionals) or memory leaks and resource waste
- To release or free resources at cancellation or at the end of thread use stack of clean-up handlers
  - handlers are pushed on the stack with a call of

```
void pthread_cleanup_push(void (*handler)(void *), *arg)
```

  - each handler is a one parameter `(void*)` function; when invoked, it will be called with **`arg`** as its argument
- When a thread is **cancelled** or terminated via **pthread_exit** function call - the routines in the thread's list of cancellation cleanup handlers are invoked one by one in **LIFO sequence**. Note: if the thread terminates with **return** in the worker function – no cleanup handlers are invoked.
- A handler can be popped with

```
void pthread_cleanup_pop(int execute)
```

  and it is also executed if **`execute!=0`**.
- A function that uses clean-up handlers and can always be safely cancelled is called **async-cancel safe.**

Thread synchronization with pthread_join

Example 2. A single-thread program to be converted to multi-threaded

```
#include <stdio.h>
int hello( void ) { // to become thread worker fun. #1
        printf( "Hello " );
        return(0);
}
int world( void ) {// to become thread worker fun. #2
        printf( "world" );
        return( 0 );
}
int main( int argc, char *argv[] ) {
        hello();        // the first word and space ("Hello ")
        world();        // the second word ("world")
        printf( "\n" ); // end of line
        return( 0 );
}
```

Example – continuation

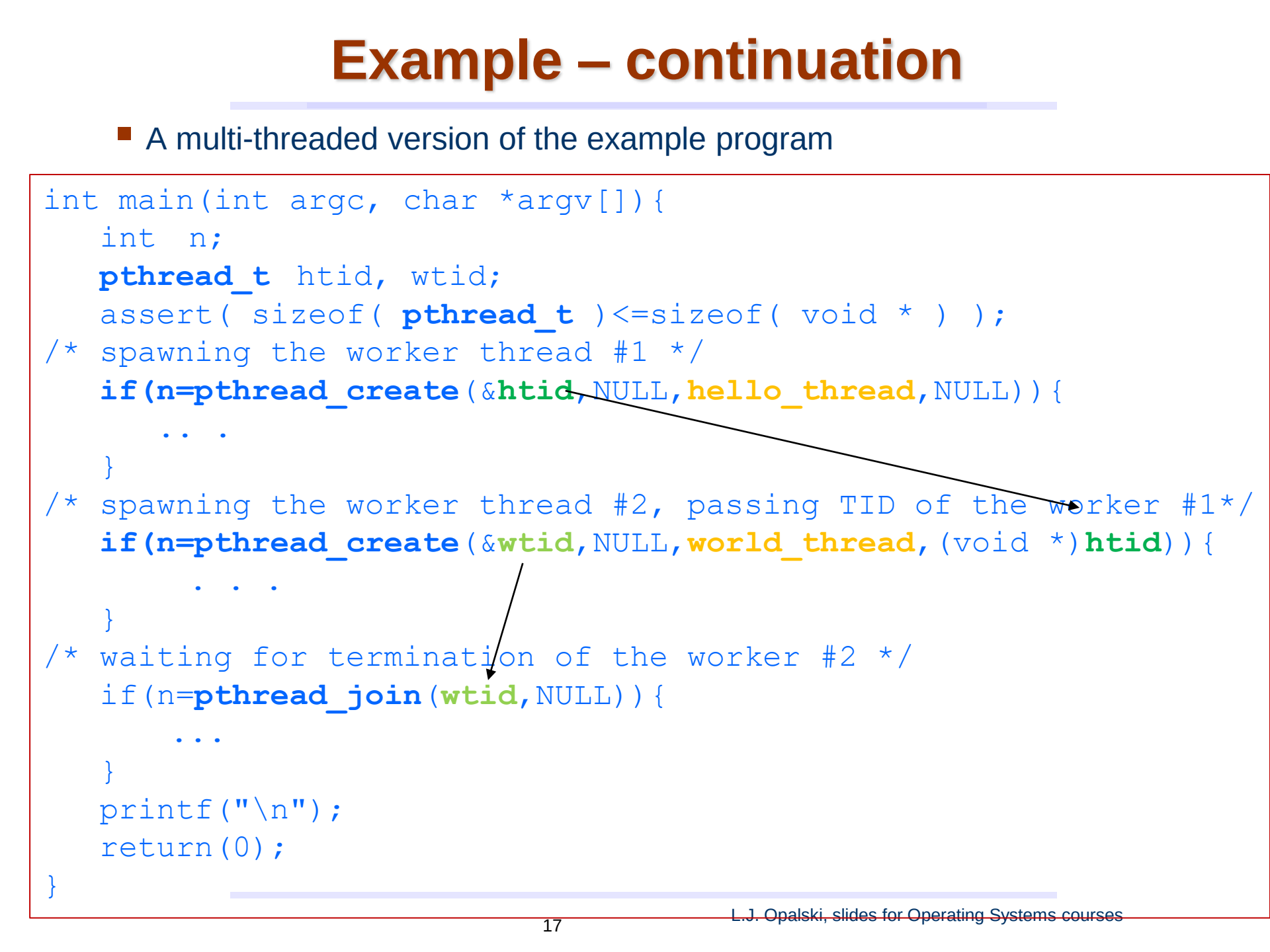
- A multi-threaded version of the example program

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
void *hello_thread( void *arg ) {/* worker fun. #1 */
      printf("Hello ");/* the first worker doesn't wait */
      return NULL;
}
void *world_thread( void *arg ) {/* worker fun. #1 */
      int           n;
      pthread_t     tid    = (pthread_t) arg;
/* waiting for the thread with given TID */
      if ( n=pthread_join( tid, NULL ) ) { ... }
      printf("world");
      pthread_exit(NULL); // or just return NULL;
}
```

Example – continuation

- A multi-threaded version of the example program

```
int main(int argc, char *argv[]){
   int  n;
   pthread_t htid, wtid;
   assert( sizeof( pthread_t )<=sizeof( void * ) );
/* spawning the worker thread #1 */
   if(n=pthread_create(&htid,NULL,hello_thread,NULL)){
      .. .
   }
/* spawning the worker thread #2, passing TID of the worker #1*/
   if(n=pthread_create(&wtid,NULL,world_thread,(void *)htid)){
      . . .
   }
/* waiting for termination of the worker #2 */
   if(n=pthread_join(wtid,NULL)){
      ...
   }
   printf("\n");
   return(0);
}
```



Preventing concurrent memory access

- **The example program #2** below is naively meant to increment and decrement **cnt** variable **n** times; in fact final **cnt** value **cannot be predicted**
- **Failure** results from concurrent non-atomic modification of **cnt** variable

```
int cnt, n=10000;
void *worker(void *arg){
int i, v;
   for(i=0; i<n; i++){ // cnt variable is modified by *((int *)arg) , i.e. 1 or -1
        v=cnt;           // in each turn of the loop
        printf("TID %d: %d\n",pthread_self(),v+((int *)arg)[0]);
        cnt=v+((int *)arg)[0];
   }
   return NULL;
}
int main(int argc, char *argv[]){
int inc=1, dec=-1;
pthread_t tid1, tid2;
        if(pthread_create(&tid1,NULL,worker,&inc)){…}  // do cnt+=n
        if(pthread_create(&tid2,NULL,worker,&dec)){…}  // do cnt-=n
        if(pthread_join(tid1,NULL)) {…}
        if(pthread_join(tid2,NULL)) {…}
        fprintf(stderr,"n=%d, cnt=%d\n",n,cnt); // show final cnt value
        return 0;
}
```

Mutex

- **Mutex** - a synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides; namely, mutual-exclusion. **The thread that has locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex**.
- Basic mutex operations:
  - Locking access to **critical section**

```
int pthread_mutex_lock(pthread_mutex_t *mp);// blocking
int pthread_mutex_trylock(pthread_mutex_t *mp);// non-bl.
```

  - Unlocking access to critical section

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

Pattern of use:

```
lock
// critical section: code that should access a shared variable in exclusive fashion
.....
unlock
```

Mutex attributes

- Initialization of a mutex

```
int pthread_mutex_init (
  pthread_mutex_t *mp,// ptr to mutex
  const pthread_mutexattr_t *mattr);// ptr to attributes
```

- Initialization of an attributes structure with default values

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr) ;
```

For setting/getting individual attributes see `man pthread_mutexattr_destroy.`

- Available attributes are implementation specific. Linux mutex attribute `pshared` determines if the mutex can be shared by processes or not. Default:**NO**

Funs: `pthread_mutexattr_getpshared`, `pthread_mutexattr_setpshared`

Mutex types: `NORMAL` mutex does not detect deadlock when locking locked

`ERRORCHECK` operations checked for validity (e.g. locking locked)

`RECURSIVE` multiple locking possible but require multiple unlocking

Example 2 – a rework

- Re-work of example 2: modifications of shared variable **cnt** are mutex protected now

```
int cnt, n;
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;// static
initializ.
void *worker(void *arg){
int i, v;
   for(i=0; i<n; i++){
       if(pthread_mutex_lock(&mtx)){. . .}
          v=cnt;
          printf("TID %d: %d\n",pthread_self(),v+((int
*)arg)[0]);
                cnt=v+((int *)arg)[0];
                if(pthread_mutex_unlock(&blokada)){. . .}
       }
       return NULL;
}
```

Reentrant and thread-safe functions

- **Reentrant function** - A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.
- **Thread-Safe -** A function that may be safely invoked concurrently by multiple threads. Each function defined in the System Interfaces volume of IEEE Std 1003.1-2001 is thread-safe unless explicitly stated otherwise (see **man pthreads(7)** for the list). Examples are any ‘‘pure’’ function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

Threads and signals

- Signals originating outside a process are delivered to a process, so the action taken for each signal is defined per process.
- For multi-thread process each signal handling is performed in a context of some specific thread.
- For signals that resulted from a thread activity related exception (e.g. division by 0, illegal memory access) the signal shall be generated for the thread that caused the signal to be generated . For other signals causes the target thread is selected among those which do not currently block this signal.
- `pthread_kill()` call requests that a specified signal be delivered to the specified thread (i.e. it makes the signal to be handled in the context of the given thread

```
int pthread_kill(pthread_t thread, int sig);
```

- Each thread has a “signal mask” that defines the set of signals currently blocked from delivery to it. The signal mask for a thread shall be initialized from that of its parent or creating thread, or from the corresponding thread in the parent process if the thread was created as the result of a call to `fork()`.

Setting/getting thread signal mask

```
int pthread_sigmask(int how, const sigset_t *new,
                    sigset_t *old);
how==SIG_SETMASK - to set the whole mask
how==SIG_BLOCK   - to add selected signals to the thread
                    signal blocking mask
how==SIG_UNBLOCK - to remove selected signals from
                    the thread signal blocking mask
```

Remarks

- Signal masks of threads of one process can differ, but signal dispositions/handlers are shared. Therefore masks determine which thread can be interrupted to handle signal and which threads will not be interrupted (with related side-effects).
- Signals related to hardware exceptions (SIGSEGV, SIGILL, SIGFPE) interrupt threads, which triggered the signal delivery. Other signals can interrupt any thread, which does not block that signal.

Warning:

Use of `sigprocmask()` (which is used to set a mask in a single-thread program) in a multi-thread program causes unspecified behavior

Other useful thread-related functions

```
pthread_t pthread_self(void);  // returns caller TID

int pthread_equal(pthread_t tid1, pthread_t tid2);
                    // compares 2 TIDs


int pthread_once(
        pthread_once_t *once_control,
        void (*init_routine)(void)
);
// The first call to pthread_once() by any thread in a process, with a given
//  once_control, shall call the init_routine() with no arguments.
// Subsequent calls of pthread_once() with the same once_control  shall
// not call the init_routine . The datum pointed at by once_control
// should be initialized statically before pthread_once()  use:

pthread_once_t once_control_dat = PTHREAD_ONCE_INIT;
once_control=&once_control_dat;
```

Atomicity of operations

- Thread safe function or code can be executed simultaneously in two or more threads without spurious results
- Thread must treat not safe functions and code as critical section and synchronize access explicitly.
  - functions that:
    - write to static buffers
    - modify process specific resources (like CWD)

    are not thread safe
  - access to shared (not private) data in program is not thread safe
  - Most of library functions that are not thread safe have thread safe alternative (`_r`)

- Operations on streams are atomic i.e. many threads do not mix data sent or received from the stream in single function call (e.g. `printf`)