Pthread Library

Outline

1. General descriptions
2. Thread management
3. Scheduler(s) in Linux
4. Time management
5. Handling periodic threads
6. Mutual exclusion
7. Examples

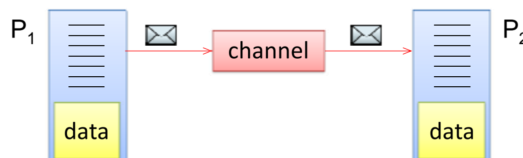
Process

A process is the main execution entity managed by the operating system for executing a program on a specific set of data.

- The following things are needed by the OS to manage a process:
  - code: compiled set of instructions;
  - data: memory space for global variables;
  - stack: memory space for local variables;
  - context: status of CPU registers;
  - parameters: type, priority, arguments.

Process

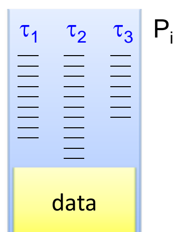
- Processes do not share memory!
- Different processes
  - execute different code;
  - access distinct memory spaces.
- Processes can communicate through a message passing mechanism:



Process and threads

- A process can include several concurrent sub-processes, called threads.
- Threads belonging to the same process:
  - share the same memory space;
  - have distinct stacks;
  - can execute the same code.

Example of a process composed by 3 threads:



Pthread Library

It includes a set of functions for thread management:

- **pthread_create:** create a thread
- **pthread_exit:** terminate the calling thread
- **pthread_cancel:** terminate another thread
- **pthread_join:** wait for the termination of a thread

Compiling

To use the Pthread library, insert **#include** <pthread.h>

```
gcc try.c -o try -lpthread -lrt
```

Thread creation

```
int  pthread_create(thread_t        *id,
                    pthread_attr_t  *attr,
                    void            *(*body)(void *),
                    void            *arg);
```

Creates a thread and returns 0 on success, or an error code, otherwise:

- id — contains the identifier of the created thread.
- attr — pointer to a structure that contains the thread attributes (if NULL uses default values).
- body — pointer to the function defining the thread code.
- arg — pointer to a single argument of the thread. To have more arguments, pass a pointer to a structure.

Thread creation

Example:

Thread ID · use default attributes · no argument

```
pthread_t tid;
    pthread_create(&tid, NULL, task, NULL);
```

```
void  *task()
{
      printf("I am a simple thread.\n");
}
```

Thread creation

Main thread:

```
pthread_t  th1;
pthread_create(&th1,…
```

thread th1

A create thread may not be immediately active. After a create one of the following things is true:

- the thread is not active yet;
- the thread in the ready queue (thus active);
- the thread already terminated.

Thread termination

A thread can terminate for different causes:

- when it executes the last instruction of the associated function (normal termination);
- when it calls the **pthread_exit** function;
- when another thread executes **pthread_cancel**.
- when its father process (e.g., the main() process) terminates for a normal termination or for a call to the **exit** function.

Thread termination

```
void  pthread_exit(void  *retval);
```

Terminate the calling thread and put in *retval the value returned by the terminated thread.

```
int  pthread_cancel(pthread_t  th);
```

Send a cancelation request for thread th. The actual termination depends on two attributes of the thread:

state: *enabled* (default) or *disabled*

type: *asynchronous* or *deferred* (default)

that can be set by the following functions

pthread_setcancelstate and **pthread_setcanceltype**

Thread joining

It is possible to wait for the termination of a thread through the function **pthread_join**:

```
int  pthread_join(pthread_t  th, void  **retval);
```

Wait for the termination of thread th.

- If retval $\neq$ NULL, the return value of the terminated thread is copied in *retval.
- If the thread is already terminated, *retval gets the value PTHREAD_CANCELED.
- It returns 0 in case of success, or an error code.

Example - create

```
#include <pthread.h>
void        *task(void *p);
int  main()
{
pthread_t  tid1, tid2;
int        tret1, tret2;
int        a = 1, b = 2;

    tret1 = pthread_create(&tid1, NULL, task, (void*)&a);
    tret2 = pthread_create(&tid2, NULL, task, (void*)&b);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("Thread1 returns %d\n", tret1);
    printf("Thread2 returns %d\n", tret2);
    return 0;
}
```

The 2 threads have the same code, but they can be made different through the passed parameter.

Example - create

```
void   *task(void *p)
{
int    *pi;

       pi = (int *)p;
       printf("This is TASK %d\n", *pi);
}
```

Thread attributes

Specify the characteristics of a thread:

- **stack size** dimension of the stack memory.
- **state** type (joinable or detached).
- **priority** thread priority level.
- **scheduler** algorithm to be used for scheduling the thread.

Such attributes must be initialized and destroyed:

```
int   pthread_attr_init(pthread_attr_t  *attr);
int   pthread_attr_destroy(pthread_attr_t  *attr);
```

Thread stack

The size of the stack of a created thread can be set by **pthread_attr_setstacksize()** and can retrieved by function **pthread_attr_getstacksize()** (the default value is 2 MB):

```
#define          STK_SIZE    40960     // 4 MB
pthread_t        tid;
pthread_attr_t   attr;
void             *stack;

    pthread_attr_init(&attr);

    pthread_attr_setstacksize(&attr,
                     &stack, STK_SIZE);

    pthread_create(&tid, &attr, task, NULL);
```

Thread priority

The priority of a thread is specified through a structure containing a single field:

```
struct sched_param {
    int sched_priority;
};
```

The priority must be first assigned through a local structure and then inserted in the thread attributes by a proper function:

```
struct sched_param  mypar;
   mypar.sched_priority = 23;
   pthread_attr_setschedparam(&myatt, &mypar);
```

Linux Schedulers

One of the following schedulers can be used for a thread:

- **SCHED_OTHER** Round Robin scheduler with aging mechanism (default policy).
- **SCHED_FIFO** Priority scheduler: threads with same priority are managed by FIFO.
- **SCHED_RR** Priority scheduler: threads with same priority are managed by Round-Robin.
- **SCHED_DEADLINE** Reservation scheduler implemented as a set of CBS running on top of EDF.

SCHED_FIFO and SCHED_RR and SCHED_DEADLINE are called real-time policies and can only be used from root.

Real-Time policies

SCHED_FIFO (Fixed-Priority Scheduling + FIFO)

Priority scheduler: threads with the same priority are handled by a FIFO policy. A thread is executed until termination, cancellation, blocking, or preemption.

SCHED_RR (Fixed-Priority Scheduling + RR)

Priority scheduler: threads with the same priority are handled by a Round-Robin policy. A thread is executed until termination, cancellation, blocking, preemption, or exhaustion of the time quantum.

The time quantum depends on the system and cannot be defined by the user, but it can be retrieved by calling the function sched_rr_get_interval().

Round-Robin quantum

int sched_rr_get_interval(pid_t *pid*, struct timespec **tp*);

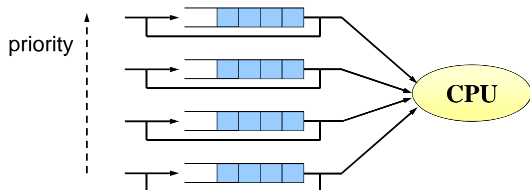
The value of the time quantum used by the Round-Robin scheduler for the process identified by *pid* is copied in the structure pointed by *tp*.

If *pid* = 0, it returns the time quantum used for the calling process:

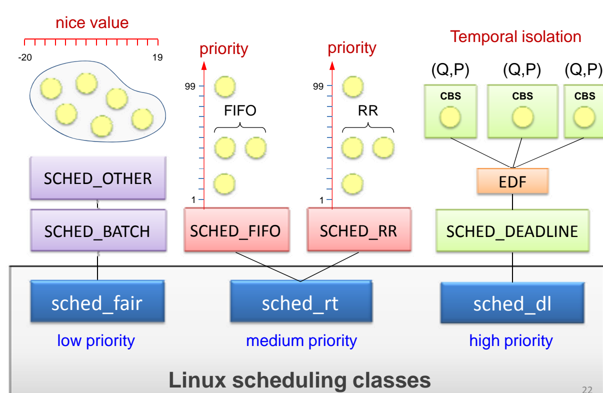
```
#include <sched.h>
struct timespec  q;
   sched_rr_get_interval(0, &q);
   printf("Q: %ld s, %ld ns\n",
                   q.tv_sec, q.tv_nsec);
```

Priority Schedulers

- Linux allows 99 priority levels: 1 (low), 99 (high). However, the POSIX standard requires only 32 levels.
- For each priority level there is a queue, in which all threads with the same priority are inserted.
- The thread at the head of the highest priority queue is selected as a running task:



Linux Schedulers



Setting the scheduler

The scheduler is also specified through the attributes, by calling the function **pthread_attr_setschedpolicy**.

- By default, a thread is scheduled with the same policy used for the father (SCHED_OTHER for the main).
- The use of different policies must be notified to the kernel by the function **pthread_attr_setinheritsched**:

```
pthread_attr_t   myatt;
   pthread_attr_init(&myatt);
   pthread_attr_setinheritsched(&myatt, PTHREAD_EXPLICIT_SCHED);
   pthread_attr_setschedpolicy(&myatt, SCHED_FIFO);
```

Example - create

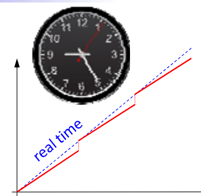
```
int   main()
{
pthread_attr_t      myatt;    /* attribute structure   */
struct sched_param  mypar;    /* priority structure    */
pthread_t           tid;      /* thread id             */

  pthread_attr_init(&myatt);

  pthread_attr_setinheritsched(&myatt, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&myatt, SCHED_FIFO);

  mypar.sched_priority = 23;
  pthread_attr_setschedparam(&myatt, &mypar);

  err = pthread_create(&tid, &myatt, task, NULL);

  pthread_join(tid, NULL);
  pthread_attr_destroy(&myatt);
}
```

Time management

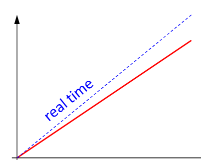
Types of clocks

Linux supports the following clocks:

CLOCK_REALTIME: gives the closest value to the absolute time. However, it can be discontinuous due to small adjustments.



CLOCK_MONOTONIC: represents the time elapsed from a given undefined instant of time. It is not affected by adjustments, hence it is the best solution to measure the time elapsed between two events.



Time representation

In the POSIX standard, time is represented through the following structure, defined in <time.h>:

```
struct timespec {
     time_t   tv_sec;     /* seconds */
     long     tv_nsec;    /* nanoseconds */
}
```

- The type **time_t** depends on the implementation, but usually it is a 32-bit integer.
- Unfortunately, the standard library does not provide functions to perform operations on time variables, thus it is necessary to define auxiliary functions.

Time copy

This function copies a source time variable ts in a destination variable pointed by td:

```
void time_copy(struct timespec *td,
               struct timespec  ts)
{
    td->tv_sec  = ts.tv_sec;
    td->tv_nsec = ts.tv_nsec;
}
```

Addition of milliseconds

This function adds a value ms expressed in milliseconds to the time variable pointed by t:

```
void time_add_ms(struct timespec *t, int ms)
{
    t->tv_sec += ms/1000;
    t->tv_nsec += (ms%1000)*1000000;

    if (t->tv_nsec > 1000000000) {
        t->tv_nsec -= 1000000000;
        t->tv_sec += 1;
    }
}
```

Time comparison

This function compares two time variables t1 and t2 and returns **0** if they are equal, **1** if t1 > t2, **-1** if t1 < t2:

```
int time_cmp(
    struct timespec t1,
    struct timespec t2)
{
    if (t1.tv_sec > t2.tv_sec) return 1;
    if (t1.tv_sec < t2.tv_sec) return -1;
    if (t1.tv_nsec > t2.tv_nsec) return 1;
    if (t1.tv_nsec < t2.tv_nsec) return -1;
    return 0;
}
```

Available functions

`int clock_getres(clockid_t clk_id, struct timespec *res);`

If *res* ≠ *NULL*, it copies in **res* the resolution of the clock specified by *clk_id*. Such a resolution depends on the specific implementation and cannot be changed.

`int clock_gettime(clockid_t clk_id, struct timespec *t);`

It copies in **t* the value of the clock specified by *clk_id*.

`int clock_settime(clockid_t clk_id, struct timespec *t);`

Sets the clock specified by *clk_id* to the value pointed by *t*. The value is truncated if it is not multiple of the resolution.

Available functions

```
int clock_nanosleep(clockid_t clk_id, int flag,
                    const struct timespec *t, struct timespec *rem);
```

Suspends the execution of the calling thread until the clock *clk_id* reaches the time specified by *t*.

- If *flag = 0*, time *t* is interpreted as relative to the current time;
- If *flag* = TIMER_ABSTIME, time t is interpreted as absolute.
- If the thread is awaken before the requested time, the remaining time is stored in *rem*.

Example

```
struct timespec t;      /* absolute time */
struct timespec dt;     /* time interval */
int    delta = 500;     /* delay in milliseconds */

       clock_gettime(CLOCK_MONOTONIC, &t);
       time_add_ms(&t, delta);

       /* suspend until absolute time t */
       clock_nanosleep(CLOCK_MONOTONIC,
                       TIMER_ABSTIME, &t, NULL);

       /* suspend for a relative interval of 500 ms */
       dt.tv_sec = 0;
       dt.tv_nsec = delta*1000000;
       clock_nanosleep(CLOCK_MONOTONIC, 0, &dt, NULL);
```

A wrong periodic thread

```
void   *task(void *arg)
{
struct timespec t;
int    period = 100;      /* period in milliseconds */

       while (1) {
              /* do useful work */
              clock_gettime(CLOCK_MONOTONIC, &t);
              time_add_ms(&t, period);
              clock_nanosleep(CLOCK_MONOTONIC,
                              TIMER_ABSTIME, &t, NULL);
       }
}
```

The task has a variable period equal to 100 ms + the response time (variable from job to job).

A periodic thread

```
void   *task(void *arg)
{
struct timespec t;
int    period = 100;      /* period in milliseconds */

       clock_gettime(CLOCK_MONOTONIC, &t);
       time_add_ms(&t, period);

       while (1) {
              /* do useful work */
              clock_nanosleep(CLOCK_MONOTONIC,
                              TIMER_ABSTIME, &t, NULL);
              time_add_ms(&t, period);
       }
}
```

Checking for deadline miss

```
void   *task(void *arg)
{
struct timespec t, now;
int    period = 100;      /* period in milliseconds */

       clock_gettime(CLOCK_MONOTONIC, &t);
       time_add_ms(&t, period);

       while (1) {
              /* do useful work */
              clock_gettime(CLOCK_MONOTONIC, &now);
              if (time_cmp(now, t) > 0) exit(-1);
              clock_nanosleep(CLOCK_MONOTONIC,
                              TIMER_ABSTIME, &t, NULL);
              time_add_ms(&t, period);
       }
}
```

Scheme of a periodic thread

Hence, a periodic thread has the following scheme:

```
void  *task(void *p)
{
<local state variables>

      set_period();

      while (1) {

          <thread body>

          if (deadline_miss()) <do action>;
          wait_for_period();
      }
}
```

Real-Time parameters

They can be stored in a dedicated structure:

```
struct      task_par {
   int      arg;           /* task argument    */
   long     wcet;          /* in microseconds  */
   int      period;        /* in milliseconds  */
   int      deadline;      /* relative (ms)    */
   int      priority;      /* in [0,99]        */
   int      dmiss;         /* no. of misses    */
   struct timespec at;     /* next activ. time */
   struct timespec dl;     /* abs. deadline    */
};
```

- Such a structure must be initialized before calling the thread_create and passed as a thread argument.
- A structure for each thread is required.

Example: main

```
struct sched_param mypar;
struct task_par tp[NT];
pthread_attr_t  att[NT];
pthread_t       tid[NT];
 for (i=0; i<NT; i++) {
   tp[i].arg = i;
   tp[i].period = 100;
   tp[i].deadline = 80;
   tp[i].priority = 20;
   tp[i].dmiss = 0;

   pthread_attr_init(&att[i]);
   pthread_attr_setinheritsched(&att[i], PTHREAD_EXPLICIT_SCHED);
   pthread_attr_setschedpolicy(&att[i], SCHED_FIFO);
   mypar.sched_priority = tp[i].priority;
   pthread_attr_setschedparam(&att[i], &mypar);
   pthread_create(&tid[i], &att[i], task, &tp[i]);
 }
```

arrays of variables for all the threads

- If known, the WCET can be used for the guarantee test.
- The absolute deadline is set online as soon as the activation time is known.

Example: thread

```
void *task(void *arg)
{
<local state variables>
struct     task_par    *tp;

     tp = (struct task_par *)arg;
     i = tp->arg;

     set_period(tp);

     while (1) {

          <thread body>

          if (deadline_miss(tp))
               printf("!");
          wait_for_period(tp);
     }
}
```

retrieves the pointer to **task_par**

retrieves the argument

set_period()

```
void set_period(struct task_par *tp)
{
struct timespec t;

     clock_gettime(CLOCK_MONOTONIC, &t);
     time_copy(&(tp->at), t);
     time_copy(&(tp->dl), t);
     time_add_ms(&(tp->at), tp->period);
     time_add_ms(&(tp->dl), tp->deadline);
}
```

Reads the current time and computes the next activation time and the absolute deadline of the task.

NOTE: the timer is not set to interrupt.

wait_for_period()

```
void  wait_for_period(struct task_par *tp)
{
      clock_nanosleep(CLOCK_MONOTONIC,
              TIMER_ABSTIME, &(tp->at), NULL);

      time_add_ms(&(tp->at), tp->period);
      time_add_ms(&(tp->dl), tp->period);
}
```

Suspends the calling thread until the next activation and, when awaken, updates activation time and deadline.

NOTE: Even though the thread calls **time_add_ms()** after the wake-up time, the computation is correct.

deadline_miss()

```
int  deadline_miss(struct task_par *tp)
{
struct timespec now;

      clock_gettime(CLOCK_MONOTONIC, &now);

      if (time_cmp(now, tp->dl) > 0) {
          tp->dmiss++;
          return 1;
      }
      return 0;
}
```

If the thread is still in execution when re-activated, it increments the value of **dmiss** and returns **1**, otherwise returns **0**.

Thread synchronization

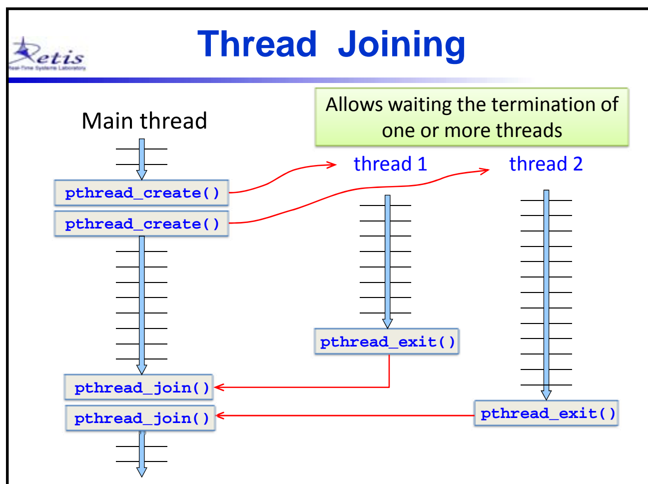
Linux threads can synchronize by the following mechanisms:

Joining allows waiting the termination of one or more threads.

Semaphore allows mutual exclusion and synchronization.

Mutex allows safer and more advanced mutual exclusion.

Thread Joining



Semaphores

General semaphores are not part of the **pthread** library, since are defined in the **POSIX** standard.

- To use them, you need to include <semaphore.h>.
- A semaphore is a variable of type **sem_t**.

Main functions:

- **sem_init:** initializes a semaphore.
- **sem_destroy:** reclaims the memory needed for a semaphore that is not used.
- **sem_wait:** wait on a semaphore.
- **sem_post:** signal on a semaphore.
- **sem_getvalue:** returns the value of a semaphore.

Initializing semaphores

int sem_init (sem_t **sem*, **int** *pshared*, **unsigned int** *v*);

Initializes a semaphore:

sem pointer to a semaphore;

pshared if 0, specifies that sem is shared among threads (hence it must be declared as global); if ≠ 0, specifies that sem is shared among processes.

v initial value of the semaphore;

Returns **0** in case of success, **-1** in case of error.

Using semaphores

int sem_wait (sem_t **sem*);

If the current value of the semaphore is 0, it blocks the calling thread until a **sem_post** is executed, otherwise it decreases the semaphore and exits.

int sem_post (sem_t **sem*);

If there are threads blocked on the semaphore, it awakes the one with the highest priority, otherwise it increments the semaphore and exits.

Both functions return **0** on success and **-1** on error.

Using semaphores

int sem_destroy (sem_t *sem);

It reclaims the memory used for the semaphore.

int sem_getvalue (sem_t *sem, int *pval);

It reads the value of the semaphore and writes it in the variable pointed by *pval*.

Both functions return **0** on success and **-1** on error.

Example - semaphores

```
#include <semaphore.h>

sem_t  sem1, sem2, sem3;        /* defines 3 semaphores    */

int    main()
{

       /* initializes sem1 for synchronization        */
       sem_init(&sem1, 0, 0);

       /* initializes sem2 for mutual exclusion       */
       sem_init(&sem2, 0, 1);

       /* initializes sem3 for simultaneous access    */
       /* of max 4 threads to a 4-units resource      */
       sem_init(&sem3, 0, 4);
```

Synchronizing on an event

```
sem_t  event;
sem_init(&event, 0, 0);
```


Mutual exclusion

```
struct point {
     int x;
     int y;
} p;

void   *writer();
void   *reader();
sem_t    s;

int   main()
{
pthread_t    tid1, tid2;

      sem_init(&s, 0, 1);

      pthread_create(&tid1, NULL, writer, NULL);
      pthread_create(&tid2, NULL, reader, NULL);

      /* do some useful work     */
}
```


Mutual exclusion

```
void   *writer()
{
       sem_wait(&s);
       p.x++;
       p.y++;
       sem_post(&s);
}
```

```
void   *reader()
{
       sem_wait(&s);
       printf("(%d,%d)\n", p.x, p.y);
       sem_post(&s);
}
```

Mutex semaphores

A MUTEX is a special type of binary semaphore with some restrictions aimed at containing programming errors:

- it can only be used for mutual exclusion, not for synchronization;
- a mutex locked by a thread can be unlocked only by the same thread.

Main functions:

- **pthread_mutex_init:** initializes a mutex.
- **pthread_mutex_destroy:** reclaims the memory of a mutex that is no more used.
- **pthread_mutex_lock:** wait on a mutex.
- **pthread_mutex_unlock:** signal on a mutex.

Initializing a mutex

A mutex must be declared and initialized before using it. There are 3 ways for initializing a mutex:

```
pthread_mutex_t      mux;          /* define a mutex       */
pthread_mutexattr_t  matt;         /* define attributes    */

/* Way 1: direct initialization */
pthread_mutex_t     mux = PTHREAD_MUTEX_INITIALIZER;

/* Way 2: initialization with default values */
pthread_mutex_init(&mux, NULL);

/* Way 3: initialization with attributes */
pthread_mutexattr_init(&matt);
pthread_mutex_init(&mux, &matt);
```

In the reported example, the three ways are equivalent and initialize the mutex with the default values.

Example - Mutex

```
struct point {
    int x;
    int y;
} p;

void  *writer();
void  *reader();
pthread_mutex_t   mux = PTHREAD_MUTEX_INITIALIZER;

int   main()
{
pthread_t    tid1, tid2;

    pthread_create(&tid1, NULL, writer, NULL);
    pthread_create(&tid2, NULL, reader, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

Global data structure shared by threads in mutual exclusion.

Definition and initialization of a mutex semaphore.

Example - Mutex

```
void  *writer()
{
      pthread_mutex_lock(&mux);
      p.x++;
      p.y++;
      pthread_mutex_unlock(&mux);
}
```

```
void  *reader()
{
      pthread_mutex_lock(&mux);
      printf("(%d,%d)\n", p.x, p.y);
      pthread_mutex_unlock(&mux);
}
```

Mutex protocols

Three protocols can be defined on a mutex:

PTHREAD_PRIO_NONE

No protocol (classic semaphore, default value).

PTHREAD_PRIO_INHERIT

Priority Inheritance protocol.

PTHREAD_PRIO_PROTECT

Immediate Priority Ceiling protocol
(also known as **Highest Locker Priority**).

NOTE: these are not supported by POSIX semaphores.

Priority Inheritance

```
#define _GNU_SOURCE

pthread_mutex_t      mux1, mux2;   /* define 2 mutexes     */
pthread_mutexattr_t  matt;         /* define mutex attrib. */

pthread_mutexattr_init(&matt);     /* initialize attributes */
pthread_mutexattr_setprotocol(&matt, PTHREAD_PRIO_INHERIT);

pthread_mutex_init(&mux1, &matt);
pthread_mutex_init(&mux2, &matt);

pthread_mutexattr_destroy(&matt); /* destroy attributes    */
```

- You need to insert: `#define _GNU_SOURCE`
- Each mutex can use a different protocol.
- The same variable matt can be used to initialize different semaphores.

Immediate Priority Ceiling

```
#define _GNU_SOURCE

pthread_mutex_t      mux1, mux2;   /* define 2 mutexes     */
pthread_mutexattr_t  matt;         /* define mutex attrib. */

pthread_mutexattr_init(&matt);
pthread_mutexattr_setprotocol(&matt, PTHREAD_PRIO_PROTECT);

pthread_mutexattr_setprioceiling(&matt, 10);
pthread_mutex_init(&mux1, &matt);

pthread_mutexattr_setprioceiling(&matt, 15);
pthread_mutex_init(&mux2, &matt);

pthread_mutexattr_destroy(&matt);
```

- The ceiling of a mutex must be set equal to the highest priority among the threads using it.