Posix Thread

References:

A.S. Tanenbaum, Cap.2 of Modern Operating Systems S.E., Prentice Hall, 2001

Tom Wagner and Don Towsley, Getting Started With POSIX Threads (available online)

Riku Saikkonen, Linux I/O port programming mini-HOWTO (available online)

Introduction

- A process is a program in execution plus
  - The current program counter;
  - Registers;
  - Variables…
- The operating system allows us to see processes as if they were executed in parallel
  - Every process has its own logical "program counter";
  - Every process is assigned the CPU for a given amount of time (it is not possible to make assumption about timing);
  - Program counter, registers, variables are saved in the memory at every context switch;
  - The difference between a process and a program is the same between a recipe (program) and its preparation (process).

Introduction

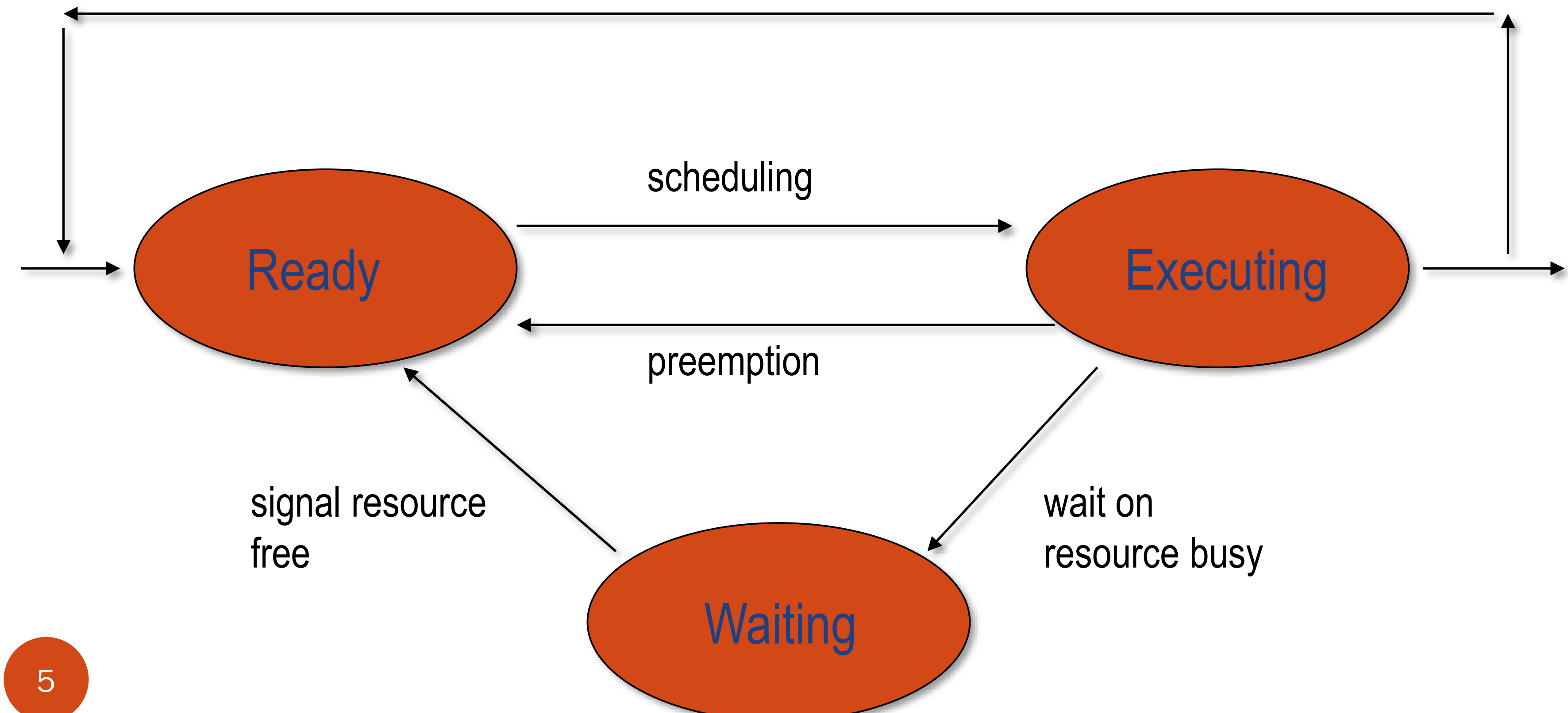
- A process has the purpose of creating an instance of a program and keeping track of its execution state.
- When are processes created?
  - System initialization (foreground & background - daemons);
  - Explicit creation by a user that executes a program;
  - Creation by another process.

Introduction

- In order to create a new process it is necessary to use a system call (e.g., "fork" in Unix, "CreateProcess" in Win32)
- In Unix
  - The memory space of the child process is initially a copy of the parent process (program counter, PSW, registers, stack, memory);
  - It is possible to change the memory image through a proper system call (execve);
  - Processes are hierarchically organized, e.g.: the process *init* forks in order to create *terminals*; after login, every *terminal* forks to execute the *shell*.
- In Windows
  - All processes have a different memory space since they are created (the program to be executed is a parameter of the "CreateProcess" system call);
  - There is not a hierarchical relation between the creator and the created process… The parent has a handle to refer to the child, but this handle can be passed to every other process.

Introduction

- States of a process
  - Executing (Running): the process uses the CPU
  - Ready: the process can execute, but the CPU is assigned to another process;
  - Waiting (Blocked): the process is waiting for en external event or resource (semaphore, mutex, access to a device)



Introduction

- The operating system keeps in memory the process table (one entry per process)
  - State of the process (Ready, Executing, Waiting)
  - Program counter
  - Open files
  - Every other information must be saved when the process switches from Executing to Ready ot to Waiting (context switch)

Process management	**Memory management**	**File management**
Resgisters Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Signals CPU time used	Pointer to text segment (code) Pointer to data segment (es. Global variables) Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Introduction

- What happens in case of an interrupt?
  - Every class of I/O devices has a memory location referred to as "interrupt vector", containing the address of the interrupt service procedure;
  - Suppose that a process is executing when an interrupt arrives: the program counter, the PSW, as well as some registers are saved on the current stack by the interrupt (at a hardware level); then the Operating System jumps to the address specified by the interrupt vector.
  - The Interrupt Service Routinge (ISR) is divided in two parts:
    - The first part– in assembler and identical for all interrupts – which saves the program counter, the PSW and registers and makes a copy of the stack in a temporary stack (this is required to restore execution);
    - The second part – in C – which executes operations that are different for every interrupts.
  - When the ISR has finished, the scheduler verifies which process should execute: it loads program counter, PSW, registers, stack, etc. Of the selected process and restore it in the Executing state;
  - Something similar happens when the interrupt is fired by the timer, and it calls immediately the scheduling routinge in order to select the next process to be executed.

Introduction

- Summary

Interrupt from I/O → Hardware: compies program counter, registers, PSW, etc. on the stack

Hardware: loads the program counter of the interrupt vector

Assembler routine: saves the registers into the process table and copies the stack

C routine: performs required operations, for instance I/O reading;

The scheduler selects a new process to be executed;

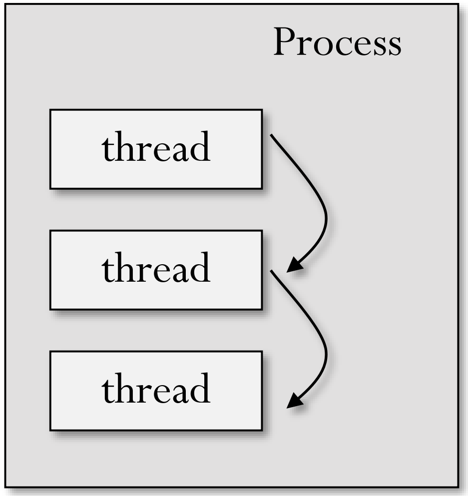
Assembler routine: executes the next process.

Threads

- Difference with processes:
  - The process are a way to group resources (e.g., memory, open file descriptors, etc.) and protect them from other processes: a thread is only a scheduling entity;
  - Thread are also known as “lightweight processes”: see multithreading.
- Characteristics of threads:
  - All threads created within the same process share the same memory, in particular:
    - Address space (data segment);
    - File descriptors;
    - Signals and signal handler;
    - Current working dir;
    - User id and group Id.

Threads

- Every thread has its own:
  - Thread ID;
  - Registers, program counter, stack pointer;
  - Stack (for local variables);
  - Signal mask;
  - Priority (staticall assigned);
  - Return value.
- A thread does not own a list of exeisting threads, and it does not know the thread that created it.



The relation between processes and threads can be depicted in this way, even if it is not completely correct.

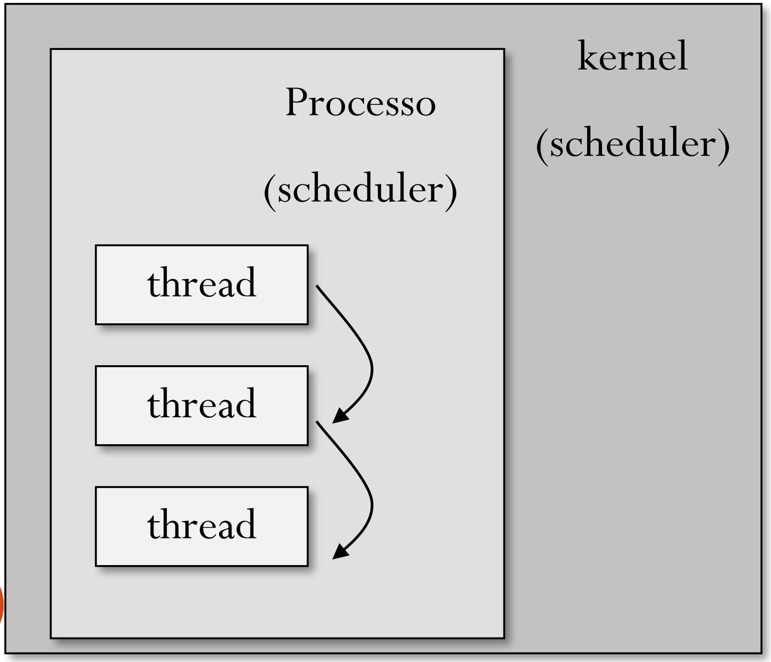
Threads

- Multithreading versus multiprocessing
- Pros ☺:
  - The creation of a new thread is usually faster than the creation of a new process, since each thread uses the same address space of the creating process. The same is true for termination time;
  - Since all threads use the same address space, the context switch is faster;
  - Communication between two threads does not require dedicated functionalities (it is sufficient to write/read from a shared memory), and – for the same reason – communication time is lower.
- Cons ☹:
  - Less protection: QNX suggests to use processes with separated (and hence protected) memory spaces;
  - Need for syncronization primitives (semaphores, mutex, conditioned variables)

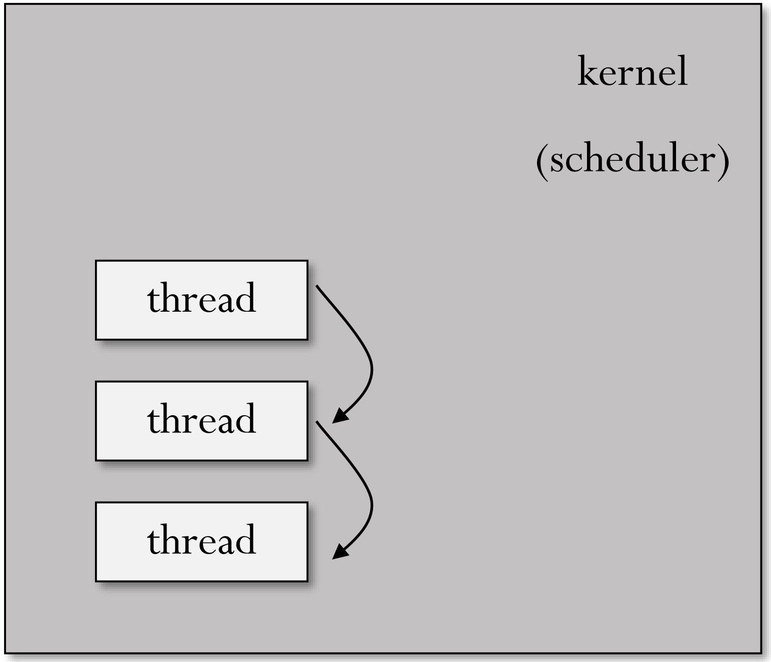
Threads

- It exists a difference between kernel threads and user threads.
  - Kernel threads are scheduled by the system scheduler, as like as they were processes with a shared memory.
  - User threads are scheduled within a process, and the system scheduler does not need to be aware of their existence: the scheduler only knows about the existence of processes.

User thread



Kernel thread



Threads

- Kernel threads versus User threads
- Pros of Kernel threads ☺:
  - Kernel threads are more efficient in I/O operations: in particular, if a thread is blocked on I/O operations, the CPU can be assigned to other threads;
  - If a kernel thread is assigned the maximum priority, I am sure that no other thread can interrupt it (as usual, interrupts can).
- Cons of Kernel threads ☹
  - Kernel threads require a Kernel that is suited to handle threads;
  - Kernrl threads are computationally less efficient concerning creation, termination, context switching.

Threads

- Pros of user threads ☺:
  - User threads can be implemented within any operating system: the scheduler is a user-implemented routine!
  - Creation, termination, context switch are faster, since they are implemented as user procedures without a system call.
  - Different scheduling algorithms can be implemented (RM, EDF, DM).
- Cons of user threads ☹ :
  - If a process is blocked on I/O, all threads scheduled by that process are blocked. Consider a user thread using a system call (e.g., read): since the scheduler is only aware of processes (not of threads) the whole process enters a Waiting state. (The problem can be solved, but with an increased complexity).
  - Every user thread must have a "thread table" similar to the "process table", which must be implemented in user space.
  - Even if a thread has the maximum priority, I have no guarantees that it will not be interrupted by another thread. Consider a user thread which is assigned the highest priority within a process A: a lower priority thread scheduled within another process B can preempt it, if the process B has a priority higher than A.
  - The concept of "timer interrupt" does not exist: a thread must explicitly give up CPU.

Threads

- Linux threads are Kernel Threads:
  - Posix Threads allows for real-time scheduling using a static priority assignment, but they do noy provide system calls for specifying timing constraints, performing a schedulability analysis, estimating of the Worst Execution Time, etc.
  - The Linux scheduler does not make a sharp distinction between threads and processes: the scheduler has the concept of tasks, scheduling entities that can share or not share some or all the resources.
  - Consider memory:
    - When a "fork" system call is executed, a new task is executed whose memory space **is the copy** of the memory space of the creating task;
    - When a "pthread_create" system call is executed, a new task is executed whose memory space **is shared** with the creating task.

Posix Thread library

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

void *print_message_function( void *ptr );

main()
{
1   pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

   /* Create independant threads each of which will execute function */

2   iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */
```

1. Thread identifier

2. Characteristic function executed in the thread

Posix Thread library

```
3  pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   printf("Thread 1 returns: %d\n",iret1);
   printf("Thread 2 returns: %d\n",iret2);
   exit(0);
}

                                    5
4 void *print_message_function( void *ptr )
{
   char *message;
   message = (char *) ptr;
   printf("%s \n", message);
}
```

3. Waiting for a thread to terminate

4. Definition of the characteristic function executed in the thread

5. Parameter passed to the function

Posix Thread library

- In order to compile:
  - With a C compiler: "cc -lpthread pthread1.c"
  - With a C++ compiler: "g++ -lpthread pthread1.c"
- After compiling, a file named "a.out" is produced
  - The program can be executed by typing: "./a.out"
  - The output of the previous program is:
    "Thread 1 Thread 2 Thread 1 returns: 0 Thread 2 returns: 0"

- Notes:
  - In this example, the same characteristic function is used in both threads. The two threads have a different behaviour only because the argument of the function is different (obviously, it is perfectly normal to use different functions for different threads);
  - Threads always terminates by calling pthread_exit or by letting the function return.

Posix Thread library

- With processes, the usual way to create a new process is the following.

```
int runcmd(char *cmd)
{
  char* argv[MAX_ARGS];
  pid_t child_pid;
  int child_status;

  parsecmd(cmd,argv);
1 child_pid = fork();
2 if(child_pid == 0) {
    /* This is done by the child process. */

3   execvp(argv[0], argv);

    /* If execvp returns, it must have failed. */

    printf("Unknown command\n");
    exit(0);
  }
4 else {
     /* This is run by the parent.  Wait for the child
        to terminate. */

     do {
       pid_t tpid = wait(&child_status);
       if(tpid != child_pid) process_terminated(tpid);
     } while(tpid != child_pid);

     return child_status;
  }
}
```

1. A copy of the process is created

2. The child process has pid=0

3. The child executes a program

4. This is the parent process

Posix Thread library

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
  void * (*start_routine)(void *), void *arg);
  - This function is used to create threads;
  - pthread_t *thread: identifier of the thread (unsigned long int);
  - const pthread_attr_t * attr: by default set to NULL (in the following we will show how it can be set by defining the fields of the "pthread_attr_t" structure);
  - void * (*start_routine): pointer to the function to be executed;
  - void *arg: pointer to the argument of the function (to pass more argument, it is necessary to pass the pointer to a struct);
  - The function can return EAGAIN in case of error. This can happen in the case that there are not enough resources for the thread creation (or in case that the maximum number of active threads have been reached).

Posix Thread library

- int pthread_attr_init(pthread_attr_t *attr)
  - This function is fundamental for setting attributes for real-time scheduling;
  - pthread_attr_t *attr: includes the Policy (RealTime FIFO) and the priority (e.g., for RateMonotonic);
  - The struct "pthread_attr_t *attr" is first initialized by using the function above; after that, other functions are available that can be used to set the different fields of the struct.

Posix Thread library

- int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
  - This function can be used to select the scheduling policy for threads;
  - int policy: scheduling policy, which can be:
    - SCHED_OTHER (in timesharing, not suited for real-time, default value);
    - SCHED_RR (realtime, round-robin);
    - SCHED_FIFO (realtime, first-in first-out);
  - The policies SCHED_RR and SCHED_FIFO are available only for processes with superuser privileges.

Posix Thread library

- int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
  - This function contains the scheduling parameters of the thread ;
  - Const struct sched_param *param: it has different fields, among which the priority in the field "param.sched_priority":
    - The lowest priority is 0 (default value), the highest is 99;
    - The priority is ignored if the scheduling policy is SCHED_OTHER; it is used only for real-time policies SCHED_RR and SCHED_FIFO.

- int pthread_setschedparam(pthread_t target_thread, int policy, const struct sched_param *param).
  - This function allows the scheduling policy and/or the priority of the thread to be modified after creation

Posix Thread library

- int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
  - It determines if the policy and the priorities are explicitly set (PTHREAD_EXPLICIT_SCHED, default value) or inherited from the parent thread (PTHREAD_INHERIT_SCHED);
- int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
  - It determines if the thread is JOINABLE or not (i.e., if it is possible call the "pthread_join" for synchronization;
- int pthread_attr_setscope(pthread_attr_t *attr, int scope);
  - It determines if we are using user threads (PTHREAD_SCOPE_PROCESS) or kernel threads (PTHREAD_SCOPE_SYSTEM);
  - The only possibility in Linux is PTHREAD_SCOPE_SYSTEM, i.e., threads compete for CPU with processes (all of them are simply tasks!).

Or also…

- int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);

- int sched_getscheduler(pid_t pid);
  - The sched_setscheduler() system call sets both the scheduling policy and parameters for the thread whose ID is specified in pid. If pid equals zero, the scheduling policy and parameters of the calling thread will be set.

Since 2014…

- int sched_setattr(pid_t pid, struct sched_attr *attr, unsigned int flags);

- int sched_getattr(pid_t pid, struct sched_attr *attr, unsigned int size, unsigned int flags);

  - Various "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable threads are selected for execution.
  - SCHED_FIFO a first-in, first-out policy; and
  - SCHED_RR a round-robin policy.
  - **SCHED_DEADLINE** a deadline scheduling policy (we will say more later…)

Posix Thread library

- For thread synchronization the Pthread library suggests three mechanisms:
  - mutex (mutual exclusion semaphores): they block the access to shared resources;
  - join: a method to force a thread to wait for other threads to terminates;
  - condition variables: they suspend the activity of a thread until a given condition verifies.

Syncronization with Mutexes

- Mutex (and semaphores) are used to prevent from data inconsistency due to "race conditions", i.e., when the results of a program vary depend on how taks are scheduled in time:
  - Race conditions can happen when two or more threads access the same memory area: mutexes are used to serialize access;
  - Posix Threads do not implement protocols for access to shared resources, such as Priority Inheritance or Priority Ceiling. Notice that such protocols must be implemented at a kernel level, since both protocols must perform some operations when a thread tries to take or release a mutex: it is not possible to implement Priority Inheritance or Priority Ceiling at a user level;
  - In the following example it is possible that problems verify, if the mutex are not used: why?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independant threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */
```

1

1. A mutex is declared and initialized

```
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}

void *functionC()
{
 2  pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
 3  pthread_mutex_unlock( &mutex1 );
}
```

2. The thread takes the mutex

3. The thread releases the mutex

- When a thread tries to take a mutex that is already taken, the thread is blocked until the mutex is free.
- If the thread which has the mutex terminates without releasing the mutex, the mutex is blocked forever.

Syncronization with Mutexes

Without Mutex	With Mutex

int counter=0;
 	
/* Function C */
void functionC()
{
 	
   counter++
 	
}
	/* Note scope of variable and mutex are the same */
	pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
	int counter=0;
	
	/* Function C */
	void functionC()
	{
	   pthread_mutex_lock(&mutex1);
	   counter++
	   pthread_mutex_unlock(&mutex1);
	}

Possible execution sequence			
Thread 1	**Thread 2**	**Thread 1**	**Thread 2**
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable counter
			counter = 2

- The problem is that "counter++" is not an indivisible operation:
  - If the assembler operations "load" and "store" (reading and writing on registers) are executed with an unlucky timing, it is possible that both threads read before writing, with the effect that the variable is incremented only once.

Syncronization with Mutexes

- Relevant functions.
- Mutex creation:
  - pthread_mutex_init
- Taking mutexes
  - pthread_mutex_lock
  - pthread_mutex_trylock
  - pthread_mutex_timedlock
- Releasing mutexes
  - pthread mutex unlock
- Cancelling mutexes
  - pthread_mutex_destroy

Syncronization with Mutexes

int pthread_mutex_init (pthread_mutex_t *MUTEX, const pthread_mutexattr_t *MUTEXATTR)

- The function is used to initialize the mutex: all the remaining functions, if applied to a mutex that has not been initialized, return EINVAL;
- pthread mutex t *MUTEX: mutex identifier;
- const pthread mutexattr t *MUTEXATTR: attributes which can be set using one of the following values:
  - fast;
  - recursive;
  - error checking;
  - NULL (default values, corresponding to fast in the current implementation).
- The function always return 0.

- Another possibility is to initialized the mutex by assigning "pthread mutex_t *MUTEX" as equal to one of the following contsants:
  - PTHREAD MUTEX INITIALIZER
  - PTHREAD RECURSIVE MUTEX INITIALIZER NP
  - PTHREAD ADAPTIVE MUTEX INITIALIZER NP
  - PTHREAD ERRORCHECK MUTEX INITIALIZER NP

Syncronization with Mutexes

- int pthread_mutex_lock (pthread_mutex_t *mutex)
  - It sets the mutex in locked state.
    - If the previous state was unlocked, then it is switched to locked;
    - if the mutex has been previously locked by another thread, the current thread is added to a queue, waiting for its turn to take the mutex;
  - If a thread has already locked the mutex and the same thread tries to lock it again, the behaviour of the function depends on the particular mutex type:
    - Fast: the thread is blocked forever;
    - Recursive: the function succeed, but the kernel increments a counter that takes into account how many times the mutex has been locked: to unlock the mutex, it will be necessary to call the function "pthread_mutex_unlock" for an identical number of times;
    - Error checking: the function returns immediately with error code EDEADLK.

Syncronization with Mutexes

- int pthread_mutex_trylock (pthread_mutex_t *MUTEX)
  - It is different from the function "pthread_mutex_lock" because it immediately returns the error code EBUSY in the following cases:
    - The calling thread finds the mutex in the locked state;
    - The mutex has been initialized as "fast", and the calling thread is trying to lock the mutex for the second time after having locked it for the first time;
- int pthread_mutex_timedlock (pthread_mutex_t *MUTEX, const struct timespec *ABSTIME)
  - Before returning an error code, the function waits for a time defined in ABSTIME. If the thread manages to lock the mutex within such time, then the function returns 0, otherwise it returns the error code ETIMEDOUT.

Syncronization with Mutexes

- int pthread_mutex_unlock (pthread_mutex t *MUTEX)
  - the behaviour of the function depends on the particular mutex type:
    - Fast: the state switches to unlocked;
    - Recursive: the counter is decreased; when the counter equals to 0, the mutex state is set to unlocked;
    - Error checking: if the mutex has been locked by the same thread that is calling pthread_mutex_unlock, the state is simply set to unlocked. Otherwise, the error code EPERM is returned.
- int pthread_mutex_destroy (pthread_mutex t *MUTEX)
  - The function eliminates the mutex and releases all resources. The function succeeds only if the mutex is in unlocked state, and in this case the function returns 0. Otherwise, the error code EBUSY is returned.

Syncronization with Mutexes

- The struct pthread_mutex_t defined in "/usr/include/bits/pthreadtypes.h" offers many insight on how mutex are dealt with.

```
/* Mutexes (not abstract because of PTHREAD_MUTEX_INITIALIZER).   */
/* (The layout is unnatural to maintain binary compatibility
    with earlier releases of LinuxThreads.) */
typedef struct
{
  int __m_reserved;               /* Reserved for future use */
 1int __m_count;                  /* Depth of recursive locking */
 2_pthread_descr __m_owner;       /* Owner thread (if recursive or errcheck) */
  int __m_kind;                   /* Mutex kind: fast, recursive or errcheck */
  struct _pthread_fastlock __m_lock; /* Underlying fast lock */
} pthread_mutex_t;
```

1. Used for recursive mutexes

2. The thread that has locked the mutex

Syncronization with Join

- A "join" is executed when a thread needs to wait for another thread to finish its execution.
  - It is the analogous of using "wait" in multiprocessing;
  - See the following example.

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

#define NTHREADS 10
void *thread_function();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, &thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
1     pthread_join( thread_id[j], NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the thre ls */
   /* have been completed.                                                 */

   printf("Final counter value: %d\n", counter);
}
```

1. The thread is suspended until the othere thread has finished

2. The thread gets its own id

```
void *thread_function()
{
2   printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}
                    3
```

3. The usage of pthread_exist would be required to pass a return value.

- Compile: “cc -lpthread join1.c”
- Run: “./a.out”

- Results:

Thread number 1026

Thread number 2051

Thread number 3076

Thread number 4101

Thread number 5126

Syncronization with Join

- Relevant functions.
- Termination:
  - pthread_exit;
  - pthread_cancel.
- Synchronization:
  - pthread_join.

Syncronization with Join

- void pthread_exit(void *retval);
  - This function kills the thread, and therefore the function itself has no return value;
  - void *retval: return value of thread; if the thread is not in detached state, the return value can be examined by another thread by calling the function "pthread_join";
  - Notice that the return value should not have a local scope, otherwise it ceases to exist in the very moment when the thread exits.
- int pthread cancel (pthread_t THREAD)
  - It sends a cancel request for a thread;
  - pthread_t THREAD: the identifier of the thread to be cancelled.

Syncronization with Join

- int pthread_join(pthread_t THREAD, void **value_ptr);
  - The function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.
  - pthread_t THREAD: the identifier of the thread to be waited for;
  - void **value_ptr: the return value of the thread.
  - Return values
    - 0 in case of success;
    - EINVAL if the thread THREAD has already terminated, or if another thread is waiting for its termination;
    - ESRCH if the thread THREAD does not exist;
    - DEADLK if the argument THREAD refers to the calling thread;

Syncronization with Conditions

- Condition variables are used to suspend a thread until an event happens:
  - The thread suspends its own execution on the condition variable until another threads signals that the condition has verified;
  - Mutexes cannot be used to implement a similar behavior, and for this reason the Posix standard implements a specific mechanism;
  - A conditioned variable must always be associated with a mutex to avoid race conditions: wating on / signalling a conditioned variable is not an indivisible operation!

Syncronization with Conditions

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}
```

1 2 3

1. Two mutexes are declared and initialized

2. Conditioned variable

3. Two threads with different functions

Syncronization with Conditions

4. Mutex lock

```
void *functionCount1()
{
    for(;;)
    {
     4 pthread_mutex_lock( &condition_mutex );

        5  pthread_cond_wait( &condition_cond, &condition_mutex );

        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

     }
}
```

5. Waiting for a condition

Syncronization with Conditions

```
void *functionCount2()
{
    for(;;)
    {
       pthread_mutex_lock( &condition_mutex );


6         pthread_cond_signal( &condition_cond );

       pthread_mutex_unlock( &condition_mutex );

       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );


    }

}
```

6. Signalling a condition. Chi esegue?

Libreria Posix Thread

- Compile: “cc -lpthread cond1.c”
- Run: “./a.out”
- A possible result:

Counter value functionCount1: 1
Counter value functionCount2: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount1: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
…

It is possible to have two subsequent executions of functionCount2, but not two subsequent executions of functionCount1!

Syncronization with Conditions

- Relevant functions.
- Creation / Destruction:
  - pthread_cond_init;
  - pthread_cond_destroy;
- Waiting:
  - pthread_cond_wait;
  - pthread_cond_timedwait;
- Waking up
  - pthread_cond_signal;
  - pthread_cond_broadcast.

Syncronization with Conditions

- int pthread_cond_init (pthread_cond_t *COND, pthread_condattr_t *cond_ATTR)
  - The function initializes a conditioned variable;
  - pthread_cond_t *COND: identifier of the conditioned variable;
  - Pthread_condattr_t* cond_ATTR: currently not supported (NULL).
  - The function always returns 0.

- A conditioned variable pthread_cond_t *COND can also be statically initialized by setting it as equal to a constant PTHREAD_COND_INITIALIZER.

Syncronization with Conditions

- int pthread_cond_signal (pthread_cond_t *COND)
  - The function restore the execution of a thread waiting for the condition specified by COND.
  - If there is no thread waiting for the condition, the function has no effect;
  - If there are more threads waiting for the condition, only one thread wakes up, and it is not possible to specify which one;
  - The function always returns 0.
- int pthread_cond_broadcast (pthread_cond_t *COND)
  - All threads waiting for the condition COND wake up;
  - The function always returns 0.

Syncronization with Conditions

- int pthread_cond_wait (pthread_cond_t *COND, pthread_mutex_t *MUTEX)
  - The function suspends the calling thread;
  - pthread_cond_t *COND: identifier of the conditioned variable;
  - pthread_mutex_t *MUTEX: the mutex is important to avoid that the condition is signalled before the thread is suspended; otherwise the signal could be lost forever.

Syncronization with Conditions

- About the usage of mutexes with condition variable (see the example at the beginning of this Section):
  - The thread1 wants to suspend on a condition variable: however, the call to pthread_cond_wait is not an indivisible operation and therefore it must be protected.
  - Something similar is true for thread2: the call to pthread_cond_signal is not an indivisible operation.
  - The problem is that, after taking the mutex, thread1 is suspended by calling pthread_cond_wait: then, how is it possible for thread2 to enter the critical section by taking the mutex and signalling the condition?
- Solution:
  - The function pthread_cond_wait called by thread1 unlocks the mutex atomically and waits for the conditioned variable to be signaled;
  - When the condition is signaled by thread2, pthread_cond_wait takes back the mutex before returning, whereas thread2 is blocked within the function pthread_cond_signal;
  - The mutex will be finally unlocked by thread 1, allowing thread 2 to continue.

Syncronization with Conditions

- int pthread_cond_timedwait (pthread_cond_t *COND, pthread_mutex_t *MUTEX, const struct timespec *ABSTIME)
  - The function suspends the calling thread: however, waiting time has a limited duration.
  - const struct timespec *ABSTIME: the waiting time before returning if the condition has not been signaled.
  - If the condition has not been signaled before the time expires, the mutex gets back to the locked state, and the function returns the error code ETIMEDOUT.
- int pthread_cond_destroy (pthread_cond_t *COND)
  - The function eliminates the condition variable, by freeing allocated resources.
  - The function returns the error code EBUSY if there is still a thread waiting on that condition variable, otherwise it returns 0.

Real-time scheduling with threads

- An example is shown on how to use threads to schedule different tasks with Rate Monotonic:
  - Only periodic tasks (3 tasks in this example);
  - Many important aspects related, e.g., to the analysis of the Worst Case Execution time are ignored or underestimated;
  - The code to be executed by each task T_i is contained in a function "void task*i*_code()".

Real-time scheduling with threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>
#include <sys/types.h>
#include <sys/types.h>

void task1_code( );
void task2_code( );
void task3_code( );
void *task1( void *);
void *task2( void *);
void *task3( void *);

#define NTASKS 3
long int periods[NTASKS];
struct timeval next_ready[NTASKS];
long int WCET[NTASKS];
pthread_attr_t attributes[NTASKS];
pthread_t thread_id[NTASKS];
struct sched_param parameters[NTASKS];
int deadline_missed[NTASKS];
```

1. Required headers

2. The code to be executed by each task (application dependent)

3. The characteristic function of each thread (used for temporization, application independent)

4. Additional information required for each task

Real-time scheduling with threads

5. Set periods

6. Read maximum and minimum priorities

```
main()
{
  //set task periods
5 periods[0]= 100000000; //in nanoseconds
  periods[1]= 200000000; //in nanoseconds
  periods[2]= 400000000; //in nanoseconds

  struct sched_param priomax;
6 priomax.sched_priority=sched_get_priority_max(SCHED_FIFO);
  struct sched_param priomin;
  priomin.sched_priority=sched_get_priority_min(SCHED_FIFO);
```

Real-time scheduling with threads

```
 // set the maximum priority for the current thread
  if (getuid() == 0)
7  pthread_setschedparam(pthread_self(),SCHED_FIFO,&priomax);

  int i;
  for (i =0; i < NTASKS; i++)
    {
      struct timeval timeval1;
      struct timezone timezone1;
      struct timeval timeval2;
      struct timezone timezone2;

   gettimeofday(&timeval1, &timezone1);
      //execute the tasks to estimate their WCET
      if (i==0)  task1_code();
8     if (i==1)  task2_code();
      if (i==2)  task3_code();
      gettimeofday(&timeval2, &timezone2);

      WCET[i]= 1000*((timeval2.tv_sec - timeval1.tv_sec)*1000000
   +(timeval2.tv_usec-timeval1.tv_usec));
      printf("\nWorst case computation time %d=%d \n", i, WCET[i]);
    }
```

7. The current thread must have the highest priority.

8. Execute the code of each task and estimate its temporal duration.

Real-time scheduling with threads

9. Compute U and check for scehedulability

```
//compute Ulub
   double U = WCET[0]/periods[0]+WCET[1]/periods[1]+WCET[2]/periods[2];
  //compute U
   double Ulub = 3*(2^(1/3)-1);
9 if (U > Ulub)
    {
      printf("\n U=%lf Ulub=%lf Task set not schedulable", U, Ulub);
      return(-1);
    }
  printf("\n U=%lf Ulub=%lf Task set schedulable", U, Ulub);
  fflush(stdout);
  sleep(5);

10 if (getuid() == 0)
    pthread_setschedparam(pthread_self(),SCHED_FIFO,&priomin);
```

10. Restore the lowest priority for the current thread

Real-time scheduling with threads

```
for (i =0; i < NTASKS; i++)
  {
11    pthread_attr_init(&(attributes[i]));

pthread_attr_setinheritsched(&(attributes[i]),
PTHREAD_EXPLICIT_SCHED);

//set real-time fifo policy
12    pthread_attr_setschedpolicy(&(attributes[i]), SCHED_FIFO);

    //set priority
 13parameters[i].sched_priority = priomin.sched_priority+NTASKS - i;
 14pthread_attr_setschedparam(&(attributes[i]), &(parameters[i]));
  }
```

11. Initialize attributes for every thread

12. Set scheduling policy

13. Set priority

14. Set attributes for every thread

Real-time scheduling with threads

```
  int iret[NTASKS];
  struct timeval ora;
  struct timezone zona;
  gettimeofday(&ora, &zona);

  //compute the beginning of the subsequent period,
  //which will be used to suspend the task after execution
  for (i = 0; i < NTASKS; i++)
    {
      long int periods_micro = periods[i]/1000;
15    next_ready[i].tv_sec = ora.tv_sec + periods_micro/1000000;
      next_ready[i].tv_usec = ora.tv_usec + periods_micro%1000000;
      deadline_missed[i] = 0;
    }

  iret[0] = pthread_create( &(thread_id[0]), &(attributes[0]), task1, NULL);
16iret[1] = pthread_create( &(thread_id[1]), &(attributes[1]), task2, NULL);
  iret[2] = pthread_create( &(thread_id[2]), &(attributes[2]), task3, NULL);

  pthread_join( thread_id[0], NULL);
  pthread_join( thread_id[1], NULL);
  pthread_join( thread_id[2], NULL);
  exit(0);
}
                                   17
```

15. By assuming that the current time is the beginning of the first period, compute the beginning of the second period

16. Create threads

17. End of main

Real-time scheduling with threads

```
void task1_code()
{
  int i,j;
  for (i = 0; i < 10; i++)
    {
      for (j = 0; j < 10000; j++)
   {
        double uno = rand()*rand();
        }
  }
  printf("1");
  fflush(stdout);
}
```

17

17. Example of application dependent task code (in this case it just performs random operations)

Real-time scheduling with threads

```
void *task1( void *ptr)
{
  int i=0;
  struct timespec waittime;
  waittime.tv_sec=0; /* seconds */
  waittime.tv_nsec = periods[0]; /* nanoseconds */

  //it executes 100 times ... it could be an infinite loop.

18  for (i=0; i < 100; i++)
     {
19     task1_code();

       struct timeval ora;
       struct timezone zona;
20     gettimeofday(&ora, &zona);

  //after execution, it computes the time before the next period starts.
21     long int timetowait= 1000*((next_ready[0].tv_sec - ora.tv_sec)*1000000
                            +(next_ready[0].tv_usec-ora.tv_usec));
       waittime.tv_sec = timetowait/1000000000;
       waittime.tv_nsec = timetowait%1000000000;
```

18. This could be an infinite loop, since periodic tasks are virtually executed «forever».

19. It executes the application dependent code

20. It gets the time of the day

21. It computes the remaining time up to the next period

Real-time scheduling with threads

```
22  if (timetowait <0)  deadline_missed[0]++;

    //suspend the task until the beginning of the next period.
23  nanosleep(&waittime, NULL);

 //compute the beginning of the subsequent period,
 //which will be used to suspend the task after the next e

    long int periods_micro=periods[0]/1000;
24  next_ready[0].tv_sec = next_ready[0].tv_sec +
                                periods_micro/1000000;
    next_ready[0].tv_usec = next_ready[0].tv_usec +
                                periods_micro%1000000;
   }
}
```

22. This means that a deadline has been missed

23. The task is suspended until the next period starts PROBLEM!!

24. Compute the beginning of the second next period, which will be used to suspend the task after the next execution

The code below can create problems… why (see 21, 23 in the previous slides)?

```
//after execution, it computes the time before the next period
  starts.

long int timetowait= 1000*((next_ready[0].tv_sec -
  ora.tv_sec)*1000000+(next_ready[0].tv_usec-ora.tv_usec));
      waittime.tv_sec = timetowait/1000000000;
      waittime.tv_nsec = timetowait%1000000000;
```

What happens if another task preempts this one here?

```
  //suspend the task until the beginning of the next period.

nanosleep(&waittime, NULL);
```

Solution: user clock_nanosleep

- int clock_nanosleep(clockid_t clock_id, int flags,
  const struct timespec *request,
  struct timespec *remain);

Like nanosleep, clock_nanosleep() allows the calling thread to sleep for an interval specified with nanosecond precision. It differs in allowing the caller to select the clock against which the sleep interval is to be measured, and in allowing the sleep interval to be specified as either an absolute or a relative value.

Aperiodic tasks in background

```
…
// application specific code
void task1_code( );
void task2_code( );
void task3_code( );
void task4_code( );
void task5_code( );

// thread functions
void *task1( void *);
void *task2( void *);
void *task3( void *);
void *task4( void *);
void *task5( void *);

// initialization of mutexes and conditions
pthread_mutex_t mutex_task_4 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_task_5 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_task_4 = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_task_5 = PTHREAD_COND_INITIALIZER;
```

1. The code to be executed by each aperiodic task (application dependent)
2. The characteristic function of each aperiodic thread (application independent)
3. We need two conditions since we have two aperiodic tasks

Aperiodic tasks in background

```
main()
{
  periods[0]= 100000000;
  periods[1]= 200000000;
  periods[2]= 400000000;

  //for aperiodic tasks we set the period equals to 0
4 periods[3]= 0;
  periods[4]= 0;

  sched_param priomax;
  priomax.sched_priority=sched_get_priority_max(SCHED_FIFO);
  sched_param priomin;
  priomin.sched_priority=sched_get_priority_min(SCHED_FIFO);
```

4. Set periods of aperiodic tasks

Aperiodic tasks in background

```
for (int i =0; i < NTASKS; i++)
    {
      struct timeval timeval1;
      struct timezone timezone1;
      struct timeval timeval2;
      struct timezone timezone2;
      gettimeofday(&timeval1, &timezone1);

      if (i==0)   task1_code();
      if (i==1)   task2_code();
      if (i==2)   task3_code();

      //aperiodic tasks
   5  if (i==3)   task4_code();
      if (i==4)   task5_code();

      gettimeofday(&timeval2, &timezone2);

      WCET[i]= 1000*((timeval2.tv_sec - timeval1.tv_sec)*1000000
                         +(timeval2.tv_usec-timeval1.tv_usec));
      printf("\nWorst Case Execution Time %d=%d \n", i, WCET[i]);
    }
```

5. Execute the code of each task and estimate its temporal duration.

Aperiodic tasks in background

```
for (int i =0; i < NPERIODICTASKS; i++)
  {
    pthread_attr_init(&(attributes[i]));
    pthread_attr_setinheritsched(&(attributes[i]),
PTHREAD_EXPLICIT_SCHED);

    pthread_attr_setschedpolicy(&(attributes[i]), SCHED_FIFO);
    parameters[i].sched_priority = priomax.sched_priority - i;
    pthread_attr_setschedparam(&(attributes[i]), &(parameters[i]));
  }

// aperiodic tasks
for (int i =NPERIODICTASKS; i < NTASKS; i++)
  {
    pthread_attr_init(&(attributes[i]));
    pthread_attr_setschedpolicy(&(attributes[i]), SCHED_FIFO);

    //set minimum priority (background scheduling)

  6 parameters[i].sched_priority = 0;
    pthread_attr_setschedparam(&(attributes[i]), &(parameter
  }
```

6. Minimum priority: background scheduling

Aperiodic tasks in background

```
struct timeval ora;
  struct timezone zona;
  gettimeofday(&ora, &zona);

  for (int i = 0; i < NPERIODICTASKS; i++)
    {
      long int periods_micro = periods[i]/1000;
      next_arrival_time[i].tv_sec = ora.tv_sec + periods_micro/1000000;
      next_arrival_time[i].tv_usec = ora.tv_usec + periods_micro%1000000;
      missed_deadlines[i] = 0;
    }

  // thread creation
  iret[0] = pthread_create( &(thread_id[0]), &(attributes[0]), task1, NULL);
  iret[1] = pthread_create( &(thread_id[1]), &(attributes[1]), task2, NULL);
  iret[2] = pthread_create( &(thread_id[2]), &(attributes[2]), task3, NULL);
  iret[3] = pthread_create( &(thread_id[3]), &(attributes[3]), task4, NULL);
7 iret[4] = pthread_create( &(thread_id[4]), &(attributes[4]), task5, NULL);

  pthread_join( thread_id[0], NULL);
8 pthread_join( thread_id[1], NULL);
  pthread_join( thread_id[2], NULL);

  exit(0);
}
```

7. Create all threads

8. I cannot join the aperiodic threads, as they are suspended

Aperiodic tasks in background

```
void task1_code()
{
  // the task does something...
  double ap;
  for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 1000; j++)
9       ap = rand()*rand()%10;
    }

  // when the random variable ap =0, aperiodic task is executed
  if (ap == 0)
    {
      printf(":Execute (4)");fflush(stdout);
      pthread_mutex_lock(&mutex_task_4);
10    pthread_cond_signal(&cond_task_4);
      pthread_mutex_unlock(&mutex_task_4);
    }
  // when the random variable ap =1, aperiodic task is executed
  if (ap == 1)
    {
      printf(":Execute (5)");fflush(stdout);
      pthread_mutex_lock(&mutex_task_5);
      pthread_cond_signal(&cond_task_5);
11    pthread_mutex_unlock(&mutex_task_5);
    }
}
```

9. Decide when aperiodic tasks shall be executed

10. Signal to wake up task 4

11. Signal to wake up task 5

Aperiodic tasks in background

```
void task4_code()
{
  for (int i = 0; i < 10; i++)
    {
      for (int j = 0; j < 1000; j++)
12        double uno = rand()*rand();
    }
  printf(" -aperiodic 4- ");
  fflush(stdout);
}

void *task4( void *)
{
13while (1)
    {
      // waiting for task 1 to signal the condition
      pthread_mutex_lock(&mutex_task_4);
      pthread_cond_wait(&cond_task_4, &mutex_task_4);
      pthread_mutex_unlock(&mutex_task_4);
14    task4_code();
    }
}
```

12. This is the activity performed by the aperiodic task

13. Remember: aperiodic tasks are «infinite» instances of…

14. Signal from Task 1

Aperiodic scheduling with polling server

```
1  double ap;
   void task1_code()
   {
     // the task does something...
   for (int i = 0; i < 10; i++)
        {
             for (int j = 0; j < 1000; j++)
         2   ap = rand()*rand()%10;
        }
   }

3  void *polling_server( void *ptr)
   {
      //synchronization to make it periodic
     for (i=0; i < 100; i++)
        {
          if (ap==0) task4_code();
      4  if (ap==1) task5_code();

   //synchronization to make it periodic
         }
```

1. This is simply a global variable

2. In principle, it should be protected with a semaphore

3. The polling server is nothing more than a periodic task

4. The polling server executes the code of other tasks

4. We are not checking if there is enough «capacity»: it should be improved.

RT Communication

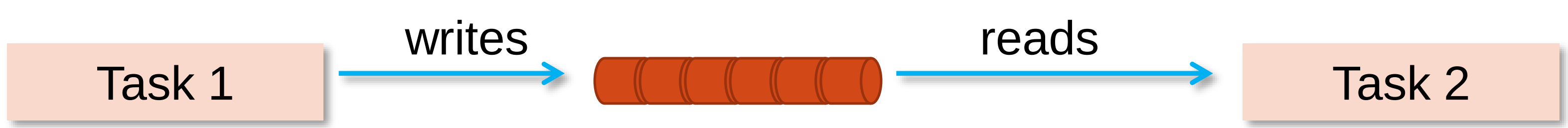
- The IPC (Inter-process communication) available for Linux provide tools for allowing processes or threads to communicate with each other.
  - Shared memory and mutexes (pthread library)
  - Half-duplex UNIX pipes
  - **FIFO - named pipes**
  - System V message queues
  - System V semaphores
  - System V shared memory
  - Berkeley style sockets
- FIFO queues and message queues are not typical of the libpthread. However, they can be used with threads and they are suited for real-time scheduling.

RT Communication

- **UNIX System V**, commonly abbreviated **SysV** (and usually pronounced—though rarely written—as "System Five"), is one of the first commercial versions of the Unix operating system.
  - It was originally developed by American Telephone & Telegraph (AT&T) and first released in 1983.
- **Berkeley Software Distribution** (**BSD**, sometimes called **Berkeley Unix**) is a Unix operating system derivative developed and distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, from 1977 to 1995.
  - Today the term "BSD" is often used non-specifically to refer to any of the BSD descendants which together form a branch of the family of Unix-like operating systems. Operating systems derived from the original BSD code remain actively developed and widely used.

RT Communication

- A “named pipe” works as a pipe, with some fundamental differences:
  - Named pipes exist in the file system as special device files;
  - Processes or threads with different parents can share data through a named pipes;
  - When tasks have finished to communicate, the named pipe remains in the file system for a following usage.



RT Communication

- There are many ways to create a pipe.
- Directly from the command shell:

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

  - With mknod, it is necessary a call to chmod to change permissions
  - With mkfifo, it can be done in a single shot.
- FIFO files can be immediately recognized in the file system by the symbol "p" (by executing ls –l to visualize the extended list of the directory content) :

```
ls -l MYFIFO
prw-r--r-- 1 root root 0 Dec 14 22:15 MYFIFO|
```

  - The vertical bar is the pipe symbol.

RT Communication

- In order to create a FIFO within a C program, we can use the system call mknod() :
- int mknod(char *pathname, mode_t mode, dev_t dev)
  - Return value: 0 in case of success, -1 in case of error by setting errno = EFAULT (non valid path) EACCES (non valid permissions), etc.
  - It creates a node in the file system (file, device file, or FIFO)
- Example:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

  - The file ``/tmp/MYFIFO" is created as a FIFO file;
  - The permissions are "0666", which are then influenced by the current value of umask:
    - final_permissions= requested_permissions & ~original_umask
    - A trick is calling umask(0) before creating the node.
  - The third argument is ignored unless we are creating a device file (for drivers). In that case, it would specify the major and minor number of the device file (it is used to associate the right driver to the device file).

RT Communication

- System calls to handle pipes
- int open(const char *pathname, int flags);
  - Return value: file descriptor;
  - Arguments: path and flags
    - Flags must contain one of the following: O_RDONLY, O_WRONLY or O_RDWR, put in OR with other options that are required for handling files;
    - For queues we have the options FIFO O_NONBLOCK (which opens in non blocking mode) or O_NDELAY.

RT Communication

- ssize_t write(int fd, const void *buf, size_t count);
  - Return value: the number of written bytes in case of success, -1 in case of error. It is possible to check errno for the corresponding error code:
    - EPIPE if fd is connected to a pipe which has not been opened for reading by any process.
    - EAGAIN when the pipe is non blocking, and we are in a situation that would produce blocking.
  - Arguments: fd is the file descriptors, buf contains the message to be sent, count the number of bytes of the message

RT Communication

- ssize_t read(int fd, void *buf, size_t count);
  - Return value: the number of read bytes in case of success (which can be smaller than count), -1 in case of error. It is possible to check errno for the corresponding error code:
    - EPIPE if fd is connected to a blocking pipe, and not all data are available as a consequence of a pipe whose writing side has been closed.
  - Arguments: fd is the file descriptor, buf will contain the received bytes, count the number of bytes that the read function wants to receive.

- int close(int fd);
  - Return value: 0 in case of success; -1 in case of error;
  - Arguments: fd is the file descriptors.

RT Communication

- Instead of open/close/read/write it is possible to use specific functions for file manipulation
  - FILE *fopen(const char *path, const char *mode);
  - int fgetc(FILE *stream);
  - char *fgets(char *s, int size, FILE *stream);
  - int fscanf(FILE *stream, const char *format, ...);
  - int fclose(FILE *stream);
  - etc.
- In general, we refer to open/read/write/close system calls that handles non formatted data streams and hence are more general (that is, they are suited for any device that accepts byte streams).

Server

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
    1   mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
            2   fp = fopen(FIFO_FILE, "r");
            3   fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
            4   fclose(fp);
        }

        return(0);
```

Create the pipe

Open the pipe

Get a string of maximum 80 characters

Close the pipe

Client

```c
#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

    1   if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

    2   fputs(argv[1], fp);

    3   fclose(fp);
        return(0);
}
```

Open the pipe

Send a string of characters

Close the pipe