Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Part 2: Process management

Seminar 5

POSIX threads programming

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

- **Goals**
  - Using **POSIX calls** related to **threads creation and basic management**
  - Experimenting **race condition** and the trouble it introduces in concurrent (multithread) programming

- **Bilbiogaphy**
  - "UNIX System Programming", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4 . Chapter 12

- **Introduction**
- Creation
- Ending and waiting
- Identification
- Race condition

- **POSIX process**
  - It creates an initial thread that executes **main()** function
    - Every thread can create other threads to perform other functions inside the process address space
  - **All threads** inside a process are at the **same level**
    - They are "**brothers**" instead of the "parent-children" relationship in processes
  - All **threads inside a process share global process variables and resources** (files, signal handlers, etc.)
    - Furthermore every thread has a **private copy of its own parameters and local varibles** related to the function it executes

- Basic thread management functions
  - `pthread` library (`#include <pthread.h>`)

Function name	Operation
pthread_create	Creates a thread that executes an specified function
pthread_attr_init	Intitializes a thread attribute object to its default values
pthread_attr_destroy	Frees a thread attribute object
pthread_join	Waits for the specified thread to end
pthread_exit	Ends the calling thread
pthread_self	Returns the calling thread ID
pthread_attr_setdetachstate	Changes the detached state attribute
pthread_attr_getdetachstate	Checks the detached state attribute

You have to compile with `-lpthread` option

- Introduction
- **Creation**
- Ending and waiting
- Identification
- Race condition

Creation

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

- `pthread_create ()`
  - It creates a **new thread in ready state**
  - The creator and created threads **compite for the CPU** according to the system scheduling policy (runtime and/or OS)
  - Any thread can call it, not only the main thread
- Arguments
  - `attr`: attribute that features the new thread
  - `start_routine`: function that contains the thread code
  - `arg`: pointer to thread parameters
  - `thread`: is an output argument that is the new thread ID

Creation

- Attributes for thread creation:

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

where

  – `attr` is the attribute to be created/destroyed

- These functions do:
  – Create/destroy a thread creation attribute
  – attr_`init` initializes `attr` to default values
    - Specific functions allow changing attributes values later
  – The same attribute variable can be reused to create several threads

Creation

- Changing/checking thread attributes:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

- where
  - `detachstate` indicates if another thread will be able to wait for the actual thread ending using `pthread_join`
  - Its possible values are:
    - PTHREAD_CREATE_JOINABLE
    - PTHREAD_CREATE_DETACHED

- Example: Hello World

```
//file: pthread_hello.c
//compile: gcc pthread_hello.c -o pthread_hello -lpthread
//see threads in execution: ps -lT
```

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

void *My_Print(void *ptr ) {
   char *message;

   message = (char *) ptr;
   write(1,message,strlen(message));
}

int main() {
   pthread_t thread1, thread2;
   pthread_attr_t attr;

   pthread_attr_init(&attr);
   pthread_create(&thread1, &attr, My_Print, "Hello ");
   pthread_create(&thread2, &attr, My_Print, " World\n");

   return 0;
}
```

- Introduction
- Creation
- **Ending and waiting**
- Identification
- Race condition

- **Ending POSIX threads**
  - A thread ends execution by its own when:
    - The thread function ends
    - The thread calls `pthread_exit`

```
int pthread_exit(void *exit_status);
```

where

- `exit_status` is a pointer to a variable by means of which a thread that ends calling `pthread_exit` communicates a ending condition value to another thread waiting to it with `pthread_join`

  - A process ends when its last thread ends

Ending and waiting

- A thread can wait for another to end
  - If the waited thread has been created with the attribute `PTHREAD_CREATE_JOINABLE`
  - To waiting thread should call `pthread_join`

```
int pthread_join(pthread_t thread, void **exit_status);
```

  - `exit_status` is the waited thread returning value through `pthread_exit`

Ending and waiting

- Example

```
...
void *function(void *p) {
   printf("I am a happy brother!\n");
   sleep(10);
}
...
int main( void ) {
   pthread_t        id_thread;
   pthread_attr_t   attributes;

   printf("Main thread: start\n");
   pthread_attr_init(&attributes);
   pthread_create(&id_thread, &attributes, function, NULL);
   printf("Main thread: I have create a brother\n");
   pthread_join(id_thread, NULL);
   printf("Main thread: That´s all folks!");
}
```

¡Warning! We have to declare a variable of thread type for every thread to be created

¿What would be the execution result if we remove pthread_join?

- Introduction
- Creation
- Ending and waiting
- **Identification**
- Race condition

Identification

- Thread identification:

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t th1, pthread_t th2);
```

where

- **`pthread_self`** returns the own thread ID of the calling thread
- **`pthread_equal`** compares two thread ID
  - Thread IDs implementation is unknown
  - It returns **cero** (0) if they are NOT equal and another value if they are equal

Identification

Example: Periodic thread creation

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

void *func_period (void *arg) {
    int period, i;
    period= *((int *)arg);
    for (i=0; i<10; i++) {
        printf("Pthread(period %d):", period);
        printf(" %ld\n", (long) pthread_self());
        sleep (period);
    }
}
```

```c
int main () {
    pthread_t t1,t2;
    pthread_attr_t attr;
    int period1=1, period2=2;

    if (pthread_attr_init(&attr) != 0) {
        printf("Error: atributtes\n");
        exit(1);
    }
    if (pthread_create(&t1, &attr, func_period, &period1) != 0) {
        printf("Error: creating first pthread\n");
        exit(1);
    }
    if (pthread_create(&t2, &attr, func_period, &period2) != 0) {
        printf("Error: creating second pthread\n");
        exit(1);
    }
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

```c
//file: th_periodic.c
//compile: gcc th_periodic.c -o th_periodic -lpthread
//see threads in execution: ps -lT
```

- Introduction
- Threads creation
- Threads ending
- Waiting
- Thread identification
- **Race condition**

Example "globalvar.c" sequential

- It increments 40.000.000 times a global variable
  - Then the final result must be 40.000.000

```
//file: globalvar.c
//compile: gcc globalvar.c -o globalvar
```

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

int Globalvariable;

int main() {
    int i;
    long iterations = 40000000;
    for (i=0; i<(iterations); i++) {
        variableGlobal ++;
    }
    printf("Globalvariable= %d\n",Globalvariable);

    return 0;
}
```

Execution result??

Race condition

- **Example** race_condition.c
  - "globalvar.c" concurrent
  - Two threads cooperate in incrementing the global variable
  - Every thread does 20.000.000 operations
  - At the end:

    Gobalvariable = 40000000

```
//file: race condition.c
//compile: gcc race_condition.c -o race_condition -lpthread
```

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

int Globalvariable;

void *Addition(void *ptr) {
    int i, aux_variable;
    int *iter = (int *)ptr;
    for (i=0; i<*iter; i++){
        aux_variable = Globalvariable;
        aux_variable++;
        Globalvariable = aux_variable;
    }
}
```

```c
int main() {
    long iterations = 20000000;
    pthread_t t1, t2;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_create(&t1, &attr, Addition, &iterations);
    pthread_create(&t2, &attr, Addition, &iterations);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Globalvariable= %d\n", Globalvariable);
    return 0;
}
```

Is ALWAYS execution result 40.000.000??