OPERATING SYSTEMS:

Lesson 5:
Processes and Threads

Jesús Carretero Pérez
David Expósito Singh
José Daniel García Sánchez
Francisco Javier García Blas
Florin Isaila

BY NC SA

Content

- **Thread Concept.**
- Thread models.
- Design aspects.
- Threads in PThreads.
- Environment of a Process.
- Signals.
- Timers.
- Exceptions.

BY NC SA

Applications with concurrent processes

- A process includes a single thread of execution.
- Design of applications with multiple concurrent tasks:
  - A receiver process receives requests and launches a process per request.
  - A receiver process uses a fixed set of request handling processes.

cc BY NC SA

Overhead of process management

- Time overhead in process creation and destruction.
- Time overhead due to context switching.
- Problems with sharing resources.

CC BY NC SA

Threads

code
data
files
registers
stack
thread
single-threaded

code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

BY NC SA

Threads

- Most modern OS provide processes with multiple instruction sequences or threads of control inside them.
- Used as the basic unit of CPU usage.
- Each one consists of:
  - Thread Id.
  - Program counter.
  - Registers set.
  - Stack.
- The share with the rest of threads in the process:
  - Memory map (code region, data region, shmem).
  - Files opened.
  - Signals, semaphores, and timers.

cc BY NC SA

Benefits

- Response capacity.
  - Better response time as user workloads are separated from processing tasks (in different threads).
- Resource sharing.
  - Threads share most of the resources automatically.
- Resource economy.
  - Creating a process has a much higher overhead than creating a thread (e.g.: in Solaris the ratio is 30 to 1).
- Scalability in multiprocessor architectures.
  - Most modern operating systems use thread as unit of scheduling.
  - Higher concurrency degree allocating different threads to different processors.

cc BY NC SA

Thread support

User space
ULT – User Level Threads

- Implemented in form of functions library in user space.
- Kernel has no knowledge about them.
  - No support in any form.
- Much faster, but some problems arise.
  - Blocking system calls.
  - Effective concurrency

Kernel space
KLT – Kernel Level Threads

- Kernel in charge of creating, scheduling and destroying them.
- Slightly slower as kernel needs to participate: change of execution level.
- In blocking system calls, only the involved thread is blocked.
- In SMP, multiple threads can run concurrently.
- No support thread code in applications.
- Kernel may also use threads to perform its own tasks.

cc BY NC SA

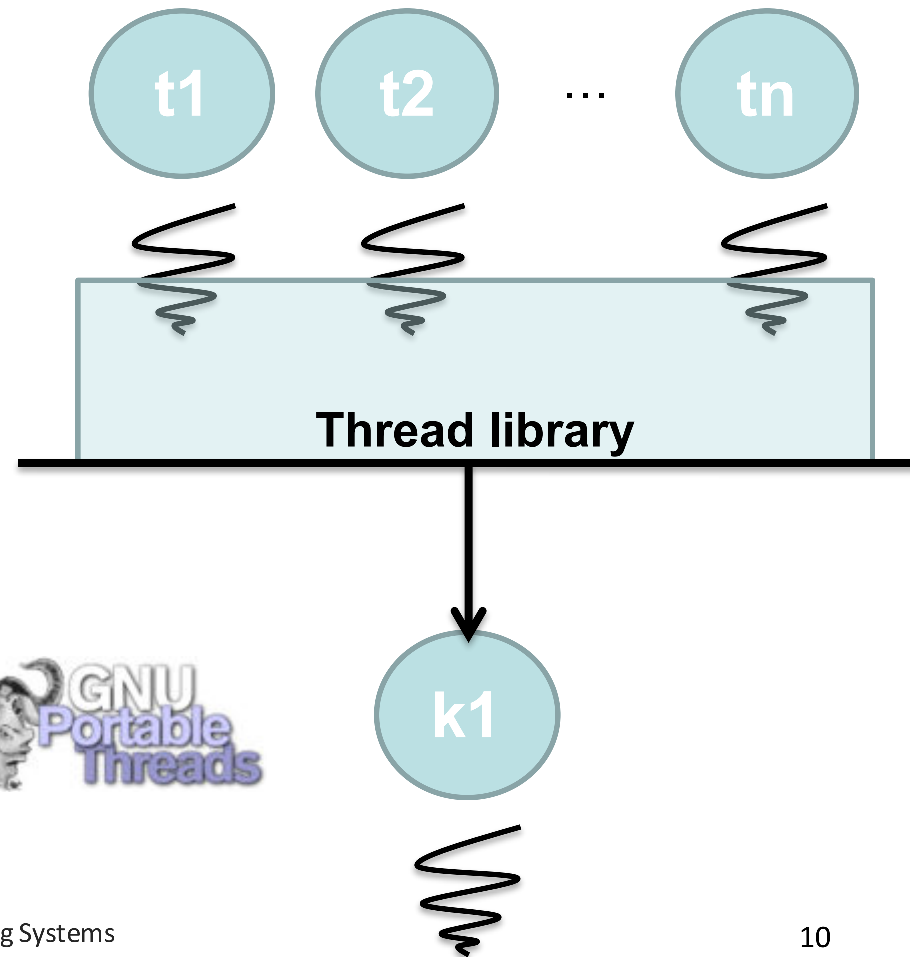
Content

- Thread Concept.
- **Thread models.**
- Design aspects.
- Threads in PThreads.
- Environment of a Process.
- Signals.
- Timers.
- Exceptions.

cc BY NC SA

Multiple threads model: Many to one

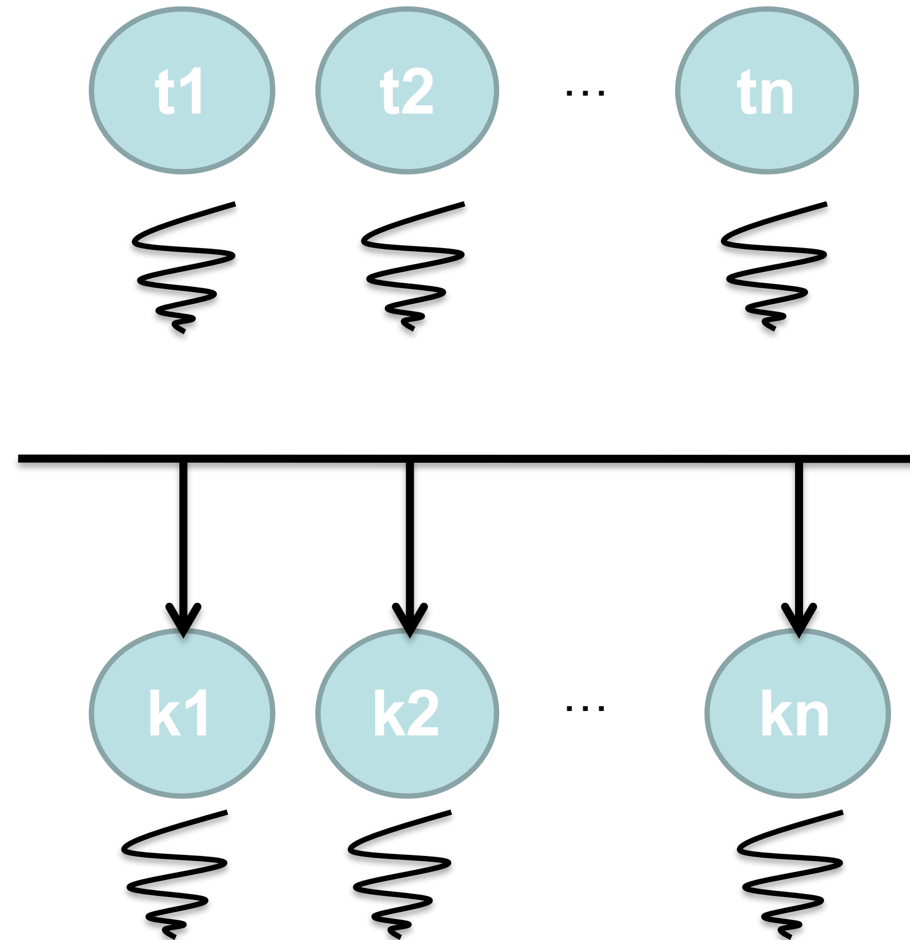
- Maps multiple user threads to a single kernel thread.
- Thread library in user space.
- Blocking call:
  - All threads are blocked.
- In multiprocessors, multiple threads cannot run at same time.



cc BY NC SA

Multiple threads model: One to one

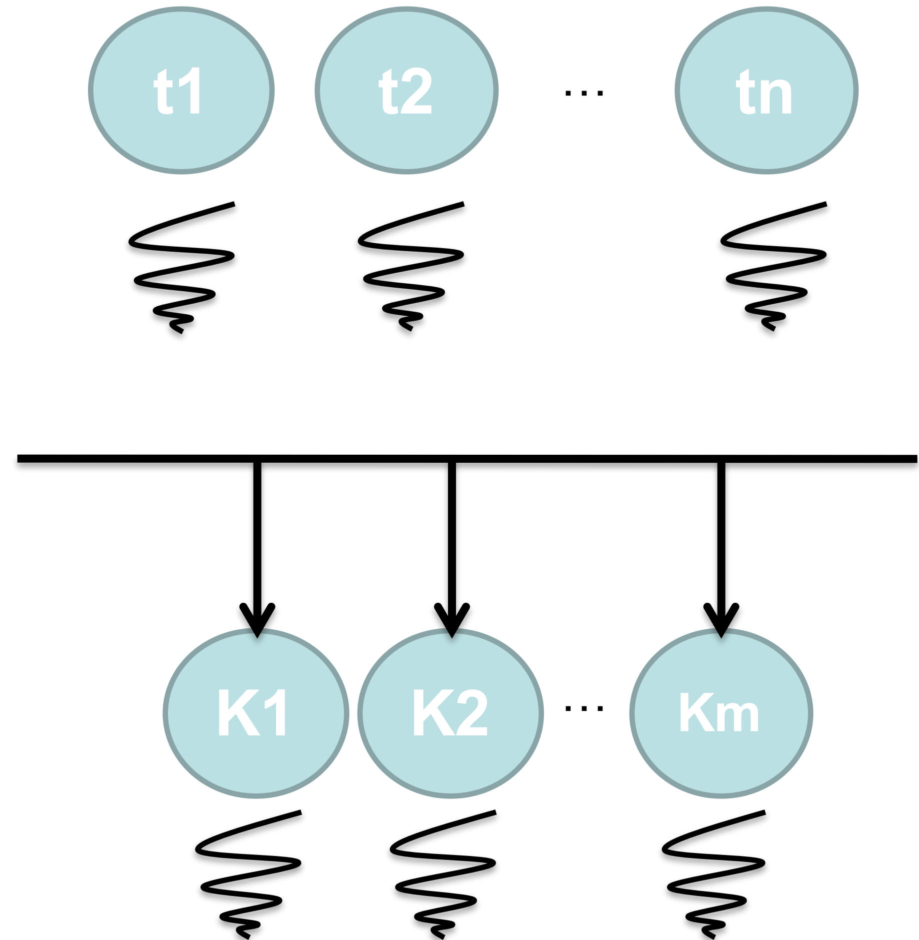
- Maps a kernel thread to every user thread.
- Most implementations restrict the number of threads that can be created.
- Examples:
  - Linux 2.6.
  - Windows.
  - Solaris 9.



cc BY NC SA

Multiple threads model: Many to many

- This model multiplexes user threads into a fixed number of kernel threads.
- Operating System threads gets much more complex.
- Examples:
  - Solaris (before version 9).
  - HP-UX.
  - IRIX.



CC BY NC SA

Content

- Thread Concept.
- Thread models.
- **Design aspects.**
- Threads in PThreads.
- Environment of a Process.
- Signals.
- Timers.
- Exceptions.

BY NC SA

Fork and exec calls

- exec call: replaces the current process by a new code and data
  - All the related threads are destroyed
- In UNIX type systems. What to do if fork is called from a thread?
  - Duplicate process with all the threads.
    - Appropriate if exec is not going to be called immediately after to change process image.
  - Duplicate process with only the calling thread.
    - More efficient if exec is going to be called and all threads would be cancelled in any case.
- Linux solution: Two versions of fork.

CC BY NC SA

Threads cancellation

- Situation when a thread notifies to others that they must terminate.
- Options:
  - Asynchronous cancellation: Forces immediate termination of the thread.
    - Problems to deallocate the thread resources
  - Deferred cancellation: Thread checks periodically whether it should terminate.
    - Better approach.

cc BY NC SA

Threads and request processing

- Applications receiving requests and processing them may make use of threads.

- But:
  - Thread creation/destruction time is a delay (although lower than process creation/destruction).
  - There is a limit in the number of concurrent threads.
  - When a request burst happens, resources may be exhausted.

cc BY NC SA

Thread Pools

- A *Thread Pool* is created and threads wait until requests arrive.

- Advantages:
  - Delay minimization: Thread already exists.
  - There is a established limit over the number of concurrent threads.

CC BY NC SA

Content

- Thread Concept.
- Thread models.
- Design aspects.
- **Threads in PThreads.**
- Environment of a Process.
- Signals.
- Timers.
- Exceptions.

cc BY NC SA

Thread creation

- ```
  int pthread_create(pthread_t *thread,
    const pthread_attr_t  *attr,
    void *(*func)(void *),
    void *arg)
  ```
  - Creates a thread and starts its execution.
  - **thread**: address of a variable of type **pthread_t** used as handle.
  - **attr**: address of a structure with attributes. NULL may be passed to specify default attributes.
  - **func**: Function with the thread-related execution code.
  - **arg**: Pointer to thread parameter. Only one parameter can be passed.
- `pthread_t pthread_self(void)`
  - Returns thread identifier for the calling thread.

CC BY NC SA
  ```

Waiting and termination

- `int pthread_join(pthread_t thread, void **value)`
  - Invoking thread waits until the thread identified by the handle has terminated.
  - **thread**: Handle to thread that must be waited to terminate.
  - **value**: Thread termination value

- `int pthread_exit(void *value)`
  - Allows a thread to terminate its execution, stating its termination status.
  - Termination status cannot be a pointer to a local variable.

CC BY NC SA

Universidad Carlos III de Madrid
www.uc3m.es

Example

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

struct addparam{
  int n, m, r;
};
typedef struct addparam
   addparam_t;

void add(addparam_t * par)
   {
  int i;
  int sum=0;
  for (i=par->n;i<=par-
   >m;i++) {
    sum+=i;
  }
  par->r=sum;
}
```

```
int main() {
  pthread_t th1, th2;
  addparam_t s1 = {1,50,0};
  addparam_t s2 = {51,100,0};

  pthread_create(&th1, NULL,
   (void*)add, (void*)&s1);
  pthread_create(&th2, NULL,
   (void*)add, (void*)&s2);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  printf("Total sum=%d\n",
    s1.r+s2.r);
}
```

cc BY NC SA

Thread attribute

- Each thread has a set of attributes associated to it.
- Attributes represented by a variable of **pthread_attr_t** type.

- Attributes control:
  - Whether a thread is detached or joinable.
  - The size of the thread private stack.
  - The location of the thread stack.
  - The scheduling policy.

CC BY NC SA

Attributes

- `int pthread_attr_init(pthread_attr_t * attr);`
  - Initializes a thread attribute structure.

- `int pthread_attr_destroy(pthread_attr_t * attr);`
  - Destroys a thread attribute structure.

- `int pthread_attr_setstacksize(pthread_attr_t * attr, int stacksize);`
  - Defines the stack size for a thread.

- `int pthread_attr_getstacksize(pthread_attr_t * attr, int *stacksize);`
  - Allows to obtain the size of the thread stack.

CC BY NC SA

Detached and joinable threads

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
  - Sets the termination state for a thread.
  - If **detachstate == PTHREAD_CREATE_DETACHED**
    - Thread releases its resources upon thread termination.
  - if **detachstate = PTHREAD_CREATE_JOINABLE**
    - Resources are not released automatically.
    - A call to **pthread_join()** is needed.

- `int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate)`
  - Allows to get the termination status.

CC BY NC SA

Example: Detached threads

```
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10
void func(void)  {
  printf("Thread %d \n", pthread_self());
  pthread_exit(0);
}

int main() {
  int j;
  pthread_attr_t attr;
  pthread_t thid[MAX_THREADS];
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
  for(j = 0; j < MAX_THREADS; j ++)
    pthread_create(&thid[j], &attr,  func, NULL);
  sleep(5);
}
```

CC BY NC SA

Content

- Thread Concept.
- Thread models.
- Design aspects.
- Threads in PThreads.
- **Environment of a Process.**
- Signals.
- Timers.
- Exceptions.

CC BY NC SA

Environment of a process

- Mechanism to pass information to a process.
- Set of **<name,value>** pairs.

- Example

```
PATH=/usr/bin:/home/joe/bin
TERM=vt100
HOME=/home/joe
PWD=/home/joe/books/second
TIMEZONE=MET
```

CC BY NC SA

Environment of a process

- The process environment is placed in the process stack at initiation.

- Access:
  - Operating system places some default values (e.g.: PATH).
  - Access through commands (set, export).
  - Access through OS API (putenv, getenv).

cc BY NC SA

Environment of a process

- A process gets third argument to main:
  - Address of a table with environment variables.

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

int main(int argc, char** argv, char** envp) {
  for (int i=0;envp[i]!=NULL;i++) {
    printf("%s\n",envp[i]);
  }
  return 0;
}
```

CC BY NC SA

Environment of a process

- `char * getenv(const char * var);`
  - Get the value of an environment variable.

- `int setenv(const char * var, const char * val, int overwrite);`
  - Modifies or adds an environment variable

- `int putenv(const char * par);`
  - Modifies or adds a pair **var=value**.

CC BY NC SA

Content

- Thread Concept.
- Thread models.
- Design aspects.
- Threads in PThreads.
- Environment of a Process.
- **Signals.**
- Timers.
- Excepetions.

CC BY NC SA

Signals

- Mechanisms to allow notifying a process of an event occurrence.

- Examples:
  - A parent process receives signal SIGCHLD when a child process finishes.
  - A process receives a signal SIGILL when it tries to execute an illegal machine instruction.

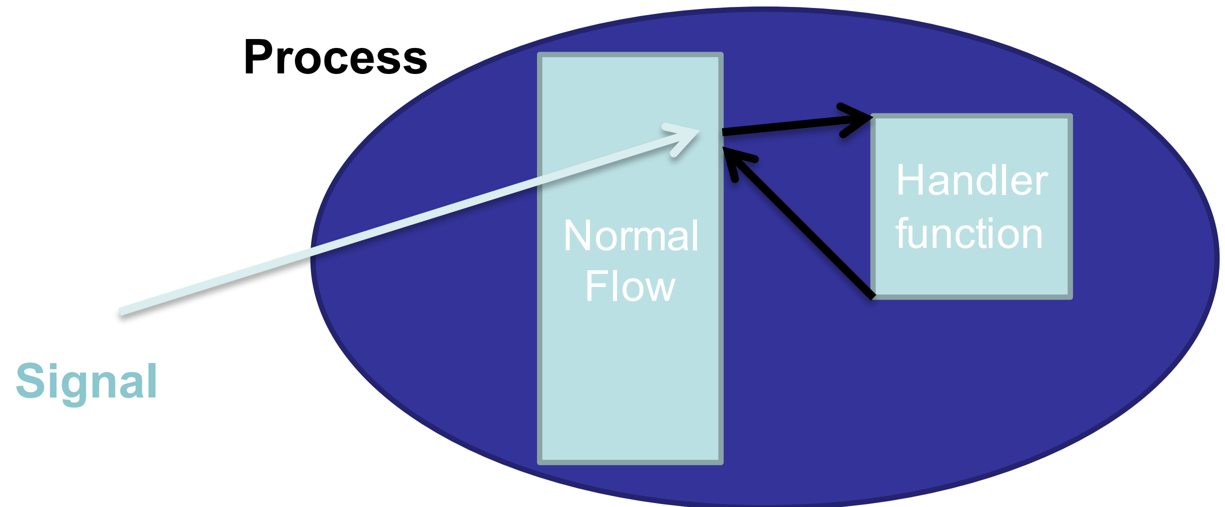
Mechanism of UNIX-like OS

cc BY NC SA

Signals

- Signals are interrupts sent to the process.

- Send or generation
  - Process -> Process (within the group) with kill command.
  - OS -> Process.



CC BY NC SA

Signals

- Some signals
  - **SIGILL**: Illegal instruction.
  - **SIGALRM**: timer elapsed.
  - **SIGKILL**: kill process.

- When a process receives a signal:
  - If it is running: Stops running the current machine instruction.
  - If there is a handler routine for the signal: Branch to run the handler.
  - If the handler does not finish the process: Return to the point where the signal was received.

cc BY NC SA

Signal

- OS transmits signal to process:
  - Process must be ready to receive it.
    - Specifying a signal procedure with **sigaction**.
    - Masking the signal with **sigprogmask**
  - If it is not ready, it performs the default action:
    - Generally process dies.
    - Some signals are ignored or have another effect.

CC BY NC SA

POSIX services for handling signals

- `int kill(pid_t pid, int sig)`
  - Send signal sig to process **pid**.
  - Special cases:
    - **pid==0**
      - Signal to process with **gid** equal to process **gid**.
    - **pid==-1**
      - Signal to all processes (except system processes)
    - **pid <-1**
      - Signal to all processes with **gid** equal to absolute value of **pid**

- ```
  int sigaction(int sig,
    struct sigaction *act,
    struct sigaction *oact)
  ```
  - Permits to specify actions to the signal **sig**.
  - Old action can be stored in **oact**.

CC BY NC SA
  ```

Sigaction struct definition

```
struct sigaction {
  void (*sa_handler)(); /* handlers*/
  sigset_t sa_mask; /* blocked signals */
  int sa_flags;     /* options */
};
```

- Handler:
  - **SIG_DFL**: Default action (usually terminates process).
  - **SIG_IGN**: Ignores signal.
  - Address of handler function.

- Mask of signals to block during handler.
- Options usually to zero.

CC BY NC SA

Signals sets

- `int sigemptyset(sigset_t * set);`
  - Creates an empty signal set.
- `int sigfillset(sigset_t * set);`
  - Creates a full set of all possible signals.
- `int sigaddset(sigset_t * set, int signo);`
  - Adds a signal to a set of signals.
- `int sigdelset(sigset_t * set, int signo);`
  - Removes a signal from a signal set.
- `int sigismember(sigset_t * set, int signo);`
  - Checks whether a signal belongs to a signal set.
  - Checks if one signal is part of a signal set.

cc BY NC SA

Example

- Ignore signal **SIGINT**
  - Produced when **Ctrl+C** keys are pressed.

```
struct sigaction act;
act.sa_handler = SIG_IGN;
act.flags = 0;
sigemptyset(&act.sa_mask);
Sigaction(SIGINT, &act, NULL);
```

CC BY NC SA

POSIX services for signals

- **`int pause(void)`**
  - Blocks a process until signal reception.
  - Does not specify a timeout.
  - Does not allow to select type of signal awaited.
  - Does not unblock process upon ignored signals.

- **`int sleep(unsigned int sec)`**
  - Suspends a process until a timeout elapses or a signal is received.

CC BY NC SA

Content

- Thread Concept.
- Thread models.
- Design aspects.
- Threads in PThreads.
- Environment of a Process.
- Signals.
- **Timers.**
- Exceptions.

BY NC SA

Timers

- Operating system keeps a timer per process (UNIX).
  - Kept in process **PCB** a counter with remaining time for the timer to elapse.
  - Operating system routine updates all timers.
  - If a timer reaches zero, it runs the handler function.

- In UNIX, the operating system sends a signal **SIGALRM** to process when its timer elapses.

cc BY NC SA

Universidad Carlos III de Madrid
www.uc3m.es

POSIX services for timers

- `int alarm(unsigned int sec)`
  - Sets a timer.
  - If argument is zero, deactivates timer.

CC BY NC SA

Example: Print message every 10 seconds

```
#include <signal.h>
#include <stdio.h>
void handle_alarm(void) {
  printf("Activated \n");
}

int main()  {
  struct sigaction act;

  /* Setis handler for SIGALRM */
  act.sa_handler = handle_alarm;
  act.sa_flags = 0;
  sigaction(SIGALRM, &act, NULL);

  act.sa_handler = SIG_IGN;            /* ignore SIGINT */
  sigaction(SIGINT, &act, NULL);
  for(;;){      /* SIGALRM every 10 secons */
    alarm(10);
    pause();
  }
}
```

Universidad Carlos III de Madrid
www.uc3m.es

Timed termination

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

pid_t pid;
void handle_alarm(void) {
  kill(pid, SIGKILL);
}

main(int argc, char **argv) {
  int status;
  char **args;
  struct sigaction act;
  args = &argv[1];
  pid = fork();
  switch(pid) {
    case -1:   /* error in fork() */
      perror ("fork");
    exit(-1);
    case 0:    /* child */
      execvp(args[0], args);
      perror("exec");
      exit(-1);
    default:   /* parent */
      /* set handler*/
      act.sa_handler = handle_alarm;
      act.sa_flags = 0;
      sigaction(SIGALRM, &act, NULL);
      alarm(5);
      wait(&status);
  }
  exit(0);
}
```

CC BY NC SA

Content

- Thread Concept.
- Thread models.
- Design aspects.
- Threads in PThreads.
- Environment of a Process.
- Signals.
- Timers.
- **Exceptions.**

CC BY NC SA

Exceptions

- Win32 offers structured exceptions management.
- Extension to C language for structured exceptions management.
  - It is not part of ISO C.

```
__try {
   /* Main code*/
}
__except (expr) {
   /* Exception
   handling*/
}
```

- Expression in __except must evaluate to:

- **EXCEPTION_EXECUTE_HANDLER**
  - Get into block.
- **EXCEPTION_CONTINUE_SEARCH**
  - Propagate exception.
- **EXCEPTION_CONTINUE_EXECUTION**
  - Ignore exception.

CC BY NC SA

Exception code

- **DWORD GetExceptionCode()**
  - Not a system call: just a macro.
  - Only can be used within exception handling.
  - A long list of values can be obtained:
    - `EXCEPTION_ACCESS_VIOLATION`
    - `EXCEPTION_ILLEGAL_INSTRUCTION`
    - `EXCEPTION_PRIV_INSTRUCTION`
    - `...`

CC BY NC SA

Safe string copy

```
LPTSTR SafeCopy(LPTSTR s1, LPTSTR s2) {
  __try {
    return strcpy(s1,s2);
  }
  __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION?
   EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
   {
    return NULL;
  }
}
```

Summary

- A process may have several threads of execution.
- A multi-threaded application consumes less resources than a multi-process application.
- Each system has a thread support mode:
  - ULT versus KLT.
- PThreads is a user library for threading.
- Win32 offers kernel threads with support for Thread Pools.

CC BY NC SA

Summary

- Environment variables allow to pass information to processes
- POSIX signals can be ignored or handled.
- Timers have different resolution in POSIX and Win32.
- Structured exception handling allows to handle anomalous situations through an extension to C language.

CC BY NC SA

OPERATING SYSTEMS:

Lesson 5:
Processes and Threads

Jesús Carretero Pérez
David Expósito Singh
José Daniel García Sánchez
Francisco Javier García Blas
Florin Isaila

cc BY NC SA