

# EPL372

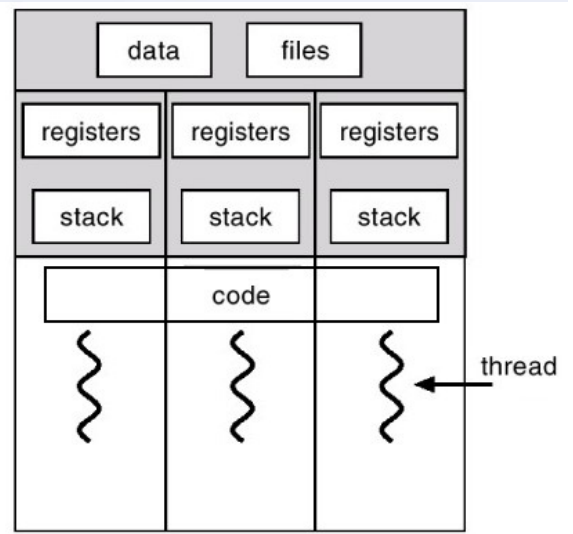
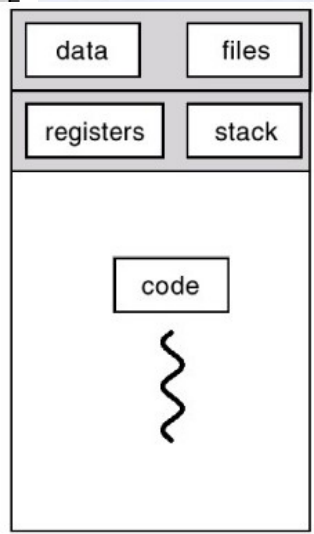
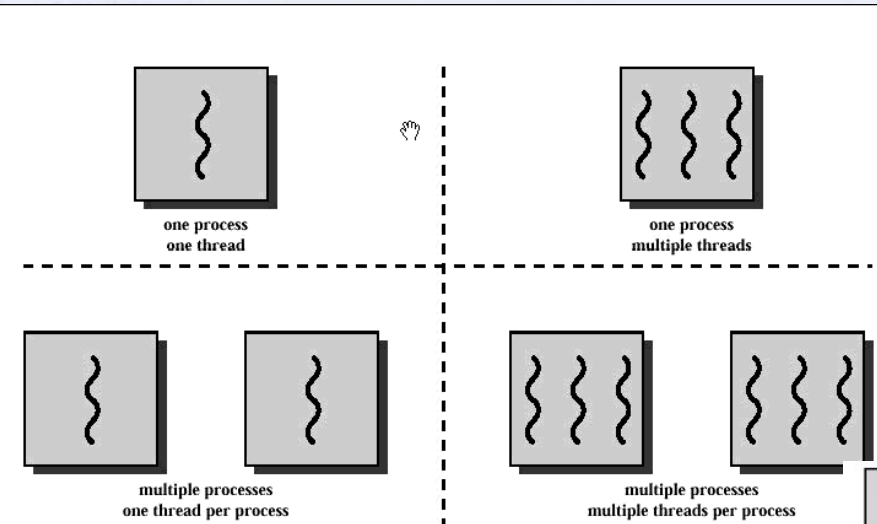
## Lab Exercise 2:

# Threads and pThreads

## Εργαστήριο 2

Πέτρος Παναγή

# Threads Vs Processes



threaded

```

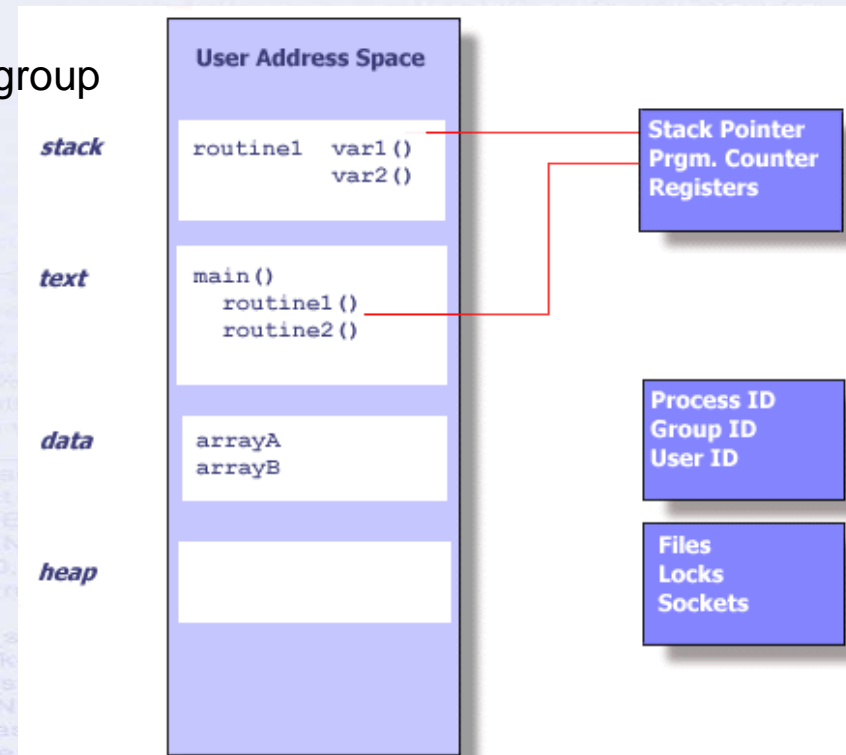
%edx,%eax\n\t"pushl %edx\n\t"call *%ebx\n\t"pushl %eax\n\t
kernel_thread(int (*fn)(void *), void * arg, unsigned long flags){
(unsigned long) fn; regs.edx = (unsigned long) arg; regs.xds =
regs.eip = (unsigned long) kernel_thread_helper; regs.xcs =
cess.. */ return do_fork(flags | CLONE_VM | CLONE_UNTRAC
structures etc.. */void exit_thread(void){ struct task_struct *tsk
bitmap... nuke it. */ if (unlikely(NULL != tsk->thread.ts_io_bitma
>thread.ts_io_bitmap = NULL;)}void flush_thread(void){ struct t
sizeof(unsigned long)*8); memset(tsk->thread.tls_array, 0, size
clear_fpu(tsk);tsk->used_math = 0;}void release_thread(struct t
bugging check if (dead_task->mm->context.size){ printk("\n
dead_task->comm, dead_task->mm->context.ldt, de
release_x86_irqs(dead_task);}/* * This gets called before we a
prepare_to_copy(struct task_struct *tsk){unlazy_fpu(tsk);}int co
unsigned long unused, struct task_struct * p, struct pt_regs * re
childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long
>eax = 0; childregs->esp = esp; p->set_child_tid = p->clear_ch
>thread.esp0 = (unsigned long) (childregs+1);p->thread.eip = (
savesegment(gs,p->thread.gs);tsk = current; if (unlikely(NULL
kmallocl(IO_BITMAP_BYTES, GFP_KERNEL); if (ip->thread.t
>thread.ts_io_bitmap, tsk->thread.ts_io_bitmap, IO_BITMAP
(clone_flags & CLONE_SETTLS) { struct desc_struct *desc; struct user_desc info;
(copy_from_user(&info, (void __user *)childregs->esi, sizeof(info))) goto out; err = -EINVAL; if (LDT_empty(&info)) goto
out; idx = info.entry_number; if (idx < GDT_ENTRY_TLS_MIN || idx > GDT_ENTRY_TLS_MAX) goto out; desc = p-
>thread.tls_array + idx - GDT_ENTRY_TLS_MIN; desc->a = LDT_entry_a(&info); desc->b = LDT_entry_b(&info); } err = 0;
out;if (err && p->thread.ts_io_bitmap) kfree(p->thread.ts_io_bitmap); return err;}/
    
```



# Process

A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory.
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

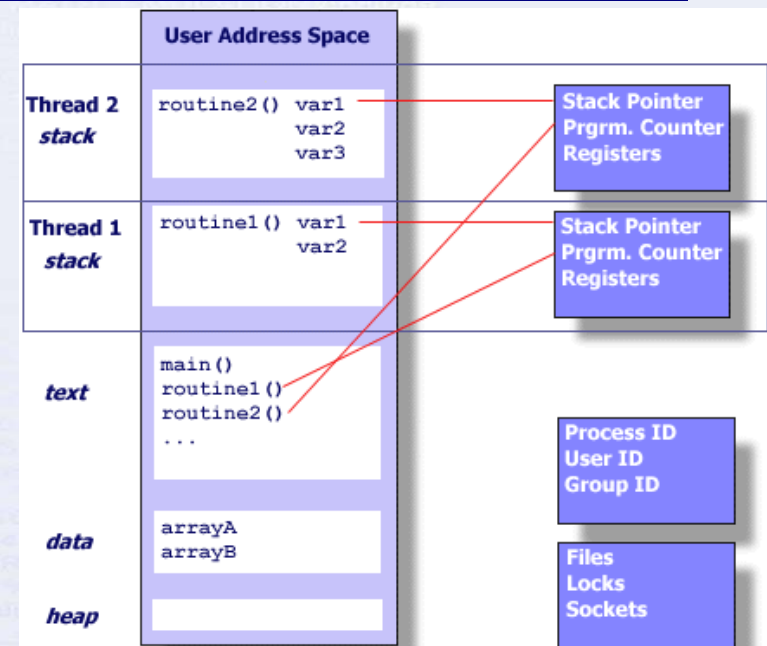


# Threads characteristics

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

Most operating systems support programs that have multiple threads of execution. Although implementations differ, they usually possess the following common characteristics:

- **Shared address space** - threads can read/write the same variables and execute the same code.
- **Private execution context** - every thread has its own set of registers
- **Private execution stack** - every thread has address space reserved for its stack
- **Thread - process association** - threads exist within and use the resources of a process. They cannot exist outside of a process



The independent flow of control is accomplished because a thread maintains its own:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.



# In Summary

- In summary, in the UNIX environment a thread:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently (and dependently)
  - Dies if the parent process dies - or something similar
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer. (This is the big Problem with threads)

# Linux POSIX threads library `pthread_create`

`pthread_create` - thread creation

## SYNOPSIS

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

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

The `pthread_create()` function is used to create a new thread, with attributes specified by `attr`, within a process. If `attr` is `NULL`, the default attributes are used. If the attributes specified by `attr` are modified later, the thread's attributes are not affected. Upon successful completion, `pthread_create()` stores the ID of the created thread in the location referenced by `thread`.

<http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>

[http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread\\_create.html](http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_create.html) 6



## pthread\_join - wait for thread termination

### SYNOPSIS

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

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

### DESCRIPTION

The `pthread_join()` function shall **suspend execution of the calling thread** until the target `thread` terminates, unless the target `thread` has already terminated. On return from a successful `pthread_join()` call with a non-NULL `value_ptr` argument, the value passed to `pthread_exit()` by the terminating thread shall be made available in the location referenced by `value_ptr`. When a `pthread_join()` returns successfully, the target thread has been terminated.

# pthread\_create: Example 1

```
1  /* Compile Using:
2  gcc -lpthread -Werror -Wall thread-create.c -o thread-create
3  */
4  #include <pthread.h>
5  #include <stdio.h>
6  #include <stdlib.h>    /* srand, rand */
7  #include <unistd.h>
8
9  int randomNumber;
10
11 void* childThread (void* threadID)
12 {
13     sleep(10-(long)threadID);
14     printf ("Child Thread %ld is called with RANDOM %d\n", (long)threadID,randomNumber++);
15     return NULL;
16 }
17 /* The main program. */
18 int main ()
19 {
20     pthread_t thread_id;
21     long threadCount;
22     srand(time(NULL));
23     randomNumber = rand()%1000;
24     /* Create a new thread. The new thread will run the print_xs function. */
25     for (threadCount = 1; threadCount< 10; threadCount++){
26         pthread_create (&thread_id, NULL, &childThread, (void *)threadCount);
27         printf ("Parent has created thread: %u \n", (unsigned int)thread_id);
28     }
29     sleep(10);
30     return 0;
31 }
```

**gcc -lpthread -Werror -Wall thread-create.c -o thread-create**



# pthread\_create: Example 2

Download and Run the Second Example:  
Do the Changes in the code and explain why the results differ.

```
...
#include <linux/errno.h>#include <linux/sched.h>#include
...
pthread_create: Example 2
...
asm volatile(
>flags); clear_thread_flag(TIF_POLLING_NRFLAG); /* Don't allow local interrupts
work to be done, so just try to conserve power and have a chance to say that
they'd like to reschedule */void cpu_idle(void) { __asm__ __volatile__(
pm_idle; if (!idle) idle = default_idle; irq_stat[cpu_processor_id] = IDLE;
idle(); schedule();)static int __init idle_setup(char *str, int *args, void *data)
pm_idle = poll_idle; return 1; __setup("idle=", idle_setup); show_regs(struct pt_regs *
0L, cr3 = 0L, cr4 = 0L; printk("Pid: %d, comm: %20s\n", current->pid, current->comm);
CPU: %d\n", 0xffff & regs->xcs, regs->eip, smp_processor_id()); print_xferd(0, 0, 0);
printk(" ESP: %04x:%08lx", 0xffff & regs->xss, regs->esp); printk(" EFLA: %04x:%08lx",
printk(" EAX: %08lx EBX: %08lx ECX: %08lx EDX: %08lx\n", regs->eax, regs->ebx, regs->ecx,
EDI: %08lx EBP: %08lx", regs->edi, regs->ebp); printk(" DS: %04x:%08lx", regs->xds,
>xes); __asm__ ("movl %%cr0, %0": "=r" (cr0)); __asm__ ("movl %%cr2, %0": "=r" (cr2));
(cr3)); /* This could fault if %cr4 does not exist */ __asm__ ("1: movl %%cr4, %0\n"
\n" ".long 1b,2b\n" ".previous\n" "=r" (cr4); "0":); printk("CR0: %08lx CR2: %08lx CR3: %08lx\n",
cr0, cr2, cr3, cr4); show_trace(NULL, &regs->esp);}/* This gets run with %ebx containing the
aining " the "args", */extern void kernel_thread_helper(void); __asm__ ("align 4\n" "kernel_thread_helper
%edx,%eax\n\t" "pushl %edx\n\t" "call %ebx\n\t" "pushl %eax\n\t" "call do_exit");/* Creates a kernel thread
kernel_thread(int (*fn)(void *), void * arg, unsigned long flags){ struct pt_regs regs; memset(&regs, 0,
(unsigned long) fn; regs.edx = (unsigned long) arg; regs.xds = __USER_DS; regs.xes = __USER_DS; regs.eip =
regs.eip = (unsigned long) kernel_thread_helper; regs.xcs = __KERNEL_CS; regs.eflags = 0x280; /* OK, create
cess.. */ return do_fork(flags | CLONE_VM | CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);}/* Proc owned
structures etc.. */void exit_thread(void){ struct task_struct *tsk = current; /* The process may have allocated
bitmap... nuke it. */ if (unlikely(NULL != tsk->thread.ts_io_bitmap)) { kfree(tsk->thread.ts_io_bitmap); tsk-
>thread.ts_io_bitmap = NULL;}}void flush_thread(void){ struct task_struct *tsk = current; memset(tsk->thread.ts_debugreg, 0,
sizeof(unsigned long)*8); memset(tsk->thread.tls_array, 0, sizeof(tsk->thread.tls_array)); /* * Forget coprocessor state. */
clear_fpu(tsk); tsk->used_math = 0; void release_thread(struct task_struct *dead_task){ if (dead_task->mm){ /* Temporary de-
bugging check if (dead_task->mm->context.size){ printk("WARNING: dead process %Bs still has LDT? %=p,%d\n",
dead_task->comm, dead_task->mm->context.ltd, dead_task->mm->context.size); BUG(); }}
release_x86_irqs(dead_task);}/* * This gets called before we allocate a new thread and copy * the current task into it */void
prepare_to_copy(struct task_struct *tsk){unlazy_fpu(tsk);}int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
unsigned long unused, struct task_struct * p, struct pt_regs * regs){ struct pt_regs * childregs; struct task_struct *tsk; int err;
childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long) p->thread.info)) - 1; struct cpy(childregs, regs); childregs-
>eax = 0; childregs->esp = esp; p->set_child_tid = p->clear_child_tid = NULL; p->thread.esp = (unsigned long) childregs; p-
>thread.esp0 = (unsigned long) (childregs+1); p->thread.eip = (unsigned long) ret_from_fork; savesegment(fs, p->thread.fs),
savesegment(gs, p->thread.gs); tsk = current; if (unlikely(NULL != tsk->thread.ts_io_bitmap)) { p->thread.ts_io_bitmap =
kmallocc(IO_BITMAP_BYTES, GFP_KERNEL); if (ip->thread.ts_io_bitmap) return -ENOMEM; memcpy(p-
>thread.ts_io_bitmap, tsk->thread.ts_io_bitmap, IO_BITMAP_BYTES);}/* * Set a new TLS for the child thread? */if
(clone_flags & CLONE_SETTLS) { struct desc_struct *desc; struct user_desc info; int idx; err = -EFAULT; if
(copy_from_user(&info, (void __user *) childregs->esi, sizeof(info))) goto out; err = -EINVAL; if (LDT_empty(&info)) goto
out; idx = info.entry_number; if (idx < GDT_ENTRY_TLS_MIN || idx > GDT_ENTRY_TLS_MAX) goto out; desc = p-
>thread.tls_array + idx - GDT_ENTRY_TLS_MIN; desc->a = LDT_entry_a(&info); desc->b = LDT_entry_b(&info); } err = 0;
out: if (err && p->thread.ts_io_bitmap) kfree(p->thread.ts_io_bitmap); return err;}/* * fill in the user structure for a core dump...
```