Introduction to OS161

Outline

- OS161 setup
- Understanding OS161
- Building OS161
- Running OS161

Os161

- OS 161 is a simplified operating system, (written in C) a simplified unix BSD-like OS.
- There are two supported branches of OS/161:
  - 1.x branch, first launched in 2001, provides a uniprocessor kernel programming environment.
  - 2.x branch, which debuted in 2009 and was finally fully released in 2015, moves into the multicore era by adding multiprocessor support and other more modern attributes.
- Includes both a kernel of conventional ("macrokernel") design and a simple userland, including a variety of test programs.

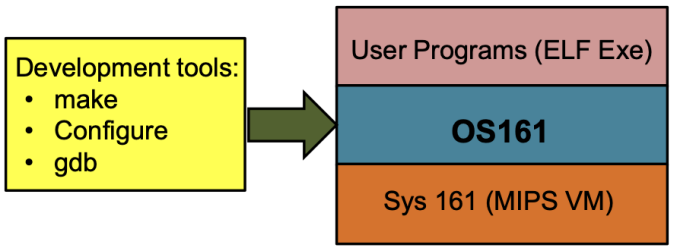
OS161 framework

OS161 includes

- The sources of the operating system (kernel), to be used for
  - Code browsing
  - Designing, implementing new/missing features
  - Running and debugging
- **A toolchain for**
  - Cross compiling (OS161 kernel for a MIPS processor)
  - Running the kernel on top of a machine simulator called sys161
  - Other tasks…

About System/161

- System/161 is a machine simulator that provides a simplified but still realistic environment for OS hacking.
- It is a 32-bit MIPS system supporting up to 32 processors.
- It was designed to support OS/161
- System/161 supports fully transparent debugging, via remote gdb into the simulator.



Os161 Support

- The base OS161 system **provides** low-level trap/interrupt, device drivers, in-kernel threads, a baseline scheduler, a minimal virtual memory system, a simple file system.

- Other things (not included) **have to be implemented**:
  - **Locks**
  - **System calls**
  - **Virtual memory -** The "dumbvm" shipped with OS161 is good enough for bootstrapping and doing the early assignments. It never reuses memory and cannot support large processes or malloc.
  - **File system**
- Many other things can be added to OS161

Os161 è un OS didattico, quindi le funzionalità di base sono implementate, mentre altre funzionalità no.

Understanding Os161 (First Lab)

- Set up OS161 development environment.
- Understand the source code structure of OS161.
- Navigate the OS/161 sources to determine where and how things are done.
- Be able to modify, build (configure, bmake) and run OS/161 kernel.
- Use GDB.

Building a kernel

Working in Os161

Directory tree (PdS Ubuntu 14.04 virtual machine):

- /home/pds: pds user directory
- os161_doc: documentation (with browsable code)
- pds-os161/root (full path: /home/pds/pds-os161/root): run/execution
  - pds-os161/root/testscripts: user program execs (from userland) to be called within os161 test menu.
- os161 (full path: /home/pds/os161): tools and os161 source/build
  - os161/tools: tools fpr compilation, make(build), debug (eg. mips-harvard.os161- gcc)
  - os161/os161-base-2.0.2: building kernel and user programs
  - os161/os161-base-2.0.2/userland: user source programs (e.g. test)
  - **os161/os161-base-2.0.2/kern: kernel source**
  - os161/os161-base-2.0.2/kern/conf: kernel configuration
  - os161/os161-base-2.0.2/kern/compile: kernel compilation/build

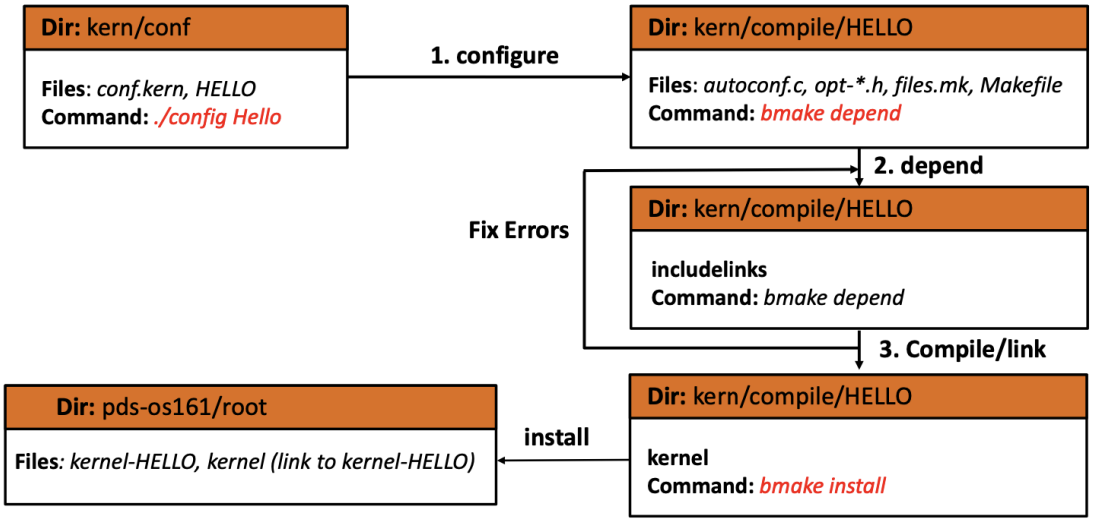
Making (building) OS161 new release: HELLO

- Three main directories:
  - Home/os161/os161-base-2.03/kern/conf
  - Home/os161/os161-base-2.03/kern/compile
  - Home/os161/root

conf lo usiamo per configurare il nuovo kernel, e ***compile*** per compilare il nuovo kernel.

root poi per buildare il tutto.

Making (building) OS161 new release: HELLO



Threads

Outline

- Kernel threads
  - Thread library
  - User processes
  - Context switch
- Syscalls & traps
- Address spaces and address translation
  - ELF files and exec load

What is a thread (process)?

- Thread or process represents the control state of an executing program
- Has an associated **context** (state)
  - Processor's **CPU state**: program counter (PC), stack pointer, other registers, execution mode (priviledged/non-priviledged)
  - **Stack**, located in the address space of the process
- Memory
  - Program code (out of context)
  - Program data (out of context)
  - Program stack containing procedure activation records (within context)

Ogni processo o thread rappresenta uno stato d'esecuzione di un programma d'esecuzione, e dispone di una certa quantità di memoria allocata, insieme ad un insieme di registri.

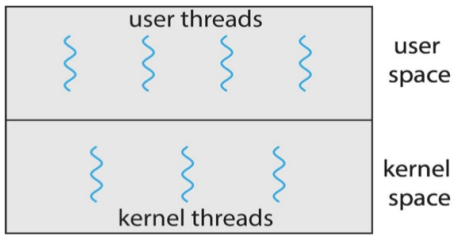
Possiamo suddividere la memoria dedicata ad un processo o thread in 3 categorie: program code, program data, e program stack.

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel Threads – Supported by the Kernel
- Examples – Virtually all general-purpose operating systems, including
  - Windows
  - **Linux**
  - Mac OS X
  - iOS
  - Andriod

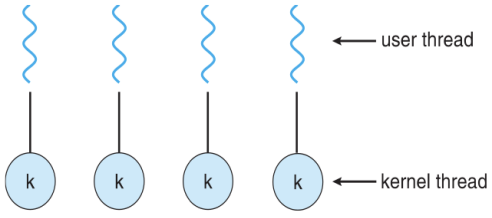
User and Kernel Threads

- User threads are supported above the kernel and are managed without kernel support.
- Kernel threads are supported and managed directly by the operating system.
- A relationship must exist between user threads and kernel threads.
- A user thread should be mapped to the kernel thread.



Multithreading Models

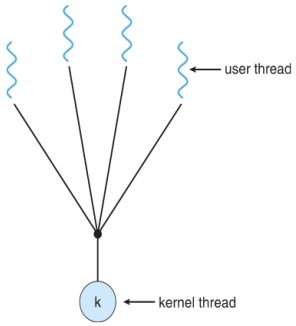
- Three common ways to establish a relationship between kernel threads and user threads:
  - **One-to-One**: Mapping each user thread to a kernel thread
    - More concurrency than many-to-one
    - The number of threads per process is sometimes restricted due to overhead
    - While allowing multiple threads to run in parallel, the drawback is that creating a user thread requires creating the corresponding kernel thread and a large number of kernel threads may burden the performance of a system.



Limitazione sul numero di thread che possiamo creare a livello kernel, ma noi non vorremmo limitarci (a livello utente) nella creazione di threads.
Ma dovendo avere un mapping 1 thread utente → 1 thread kernel le performance possono essere basse.

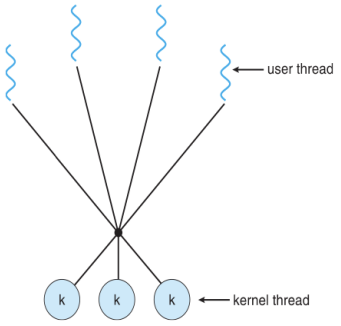
Multithreading Models

- Three common ways to establish a relationship between kernel threads and user threads:
  - **Many-to-One**: mapping many user-level threads to one kernel thread.
    - Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
    - One thread blocking causes all to block
    - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time



Multithreading Models

- Three common ways to establish a relationship between kernel threads and user threads:
  - **Many-to-Many**: mapping many user-level threads to a smaller or equal number of kernel threads
    - Allows the operating system to create a sufficient number of kernel threads
    - Developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.



Process

Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in a sequential fashion
  - The program code, also called the **text section**
  - Current activity including **program counter**, **processor registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

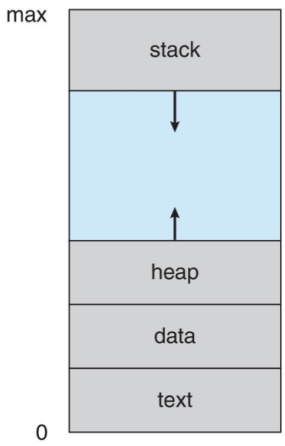
Come viene dedicata e suddivisa la memoria di un processo o thread

Process Concept

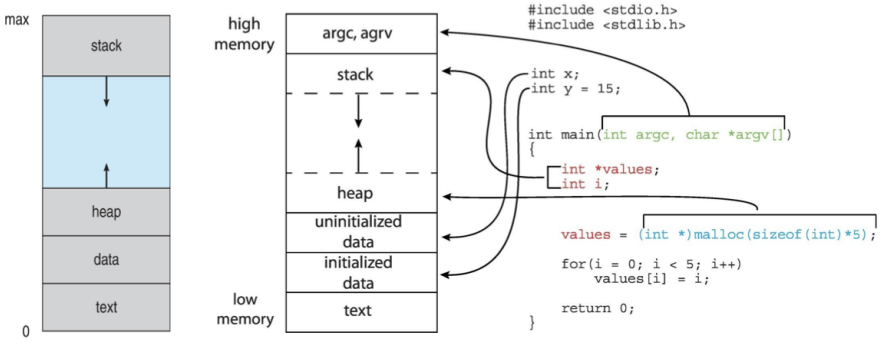
- Program is ***passive*** entity stored on disk (**executable file**); process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory

- User-side layout of the memory
- Logical memory addresses of processes are organized as:
  - **Text** where the code is
  - **Data** where the global variables are
  - **Stack** containing temporary data
  - **Heap** containing memory dynamically allocated during run time
    - **Stack** and **Heap** can increase the size



Memory Layout of a C Program

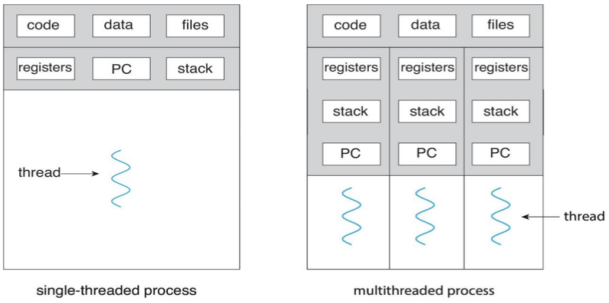


Questo per quanto riguarda un singolo processo..

Ma se abbiamo processi multithread, come viene gestita la memoria?

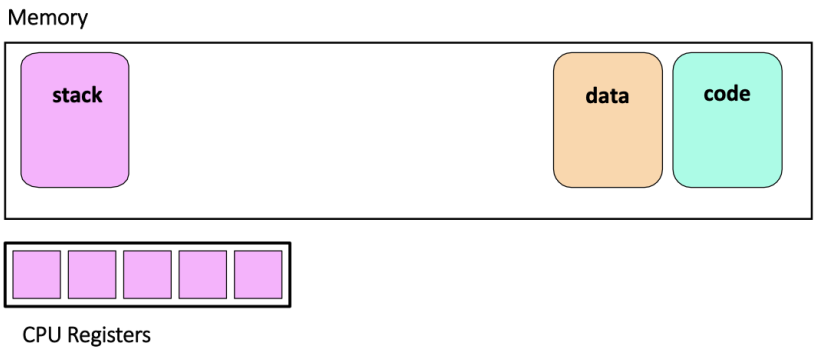
Single and Multithreaded Processes

- Single thread single process
- Single process multiple threads
- When talking about user threads, the context switch between threads is done by the thread library (POSIX).
- Thread at the kernel level, context switch which is about saving the state of registers, stack, and PC is managed by the OS (which we care about)



Vediamo che una parte (code, data e files) è condivisa tra tutti i threads, ma poi ogni thread ha il proprio stack, PC e registri..

Thread Context – OS161



Suddivisione di memoria di un singolo thread in esecuzione, per memorizzare il **contesto** di tale thread.

Implementing Threads

- Moving toward thread on OS161:
- A thread library is responsible for implementing threads.(managing threads at kernel level through library)
- Having data structure for saving thread information: the thread library stores threads' contexts (or pointers to the threads' contexts) when they are not running.
- The Data structure used by thread library to store a thread context is called **Thread Control Block**
- In the OS/161 kernel's thread implementation, thread contexts are stored in **thread structures**

La struttura dati usata per memorizzare il contesto di un thread si chiama ***Thread Control Block***.
Questa struttura dati viene usata per memorizzare il contesto di un thread, specialmente utile quando bisogna fare
context switching.

Thread Structure

The OS161 Thread Structure

All instructions are written in C, the structure that defines thread can be find in "**thread.h**"

```
/* see kern/include/thread.h */
struct thread {
  char *t_name;                    /* Name of this thread */
  const char *t_wchan_name;        /* Name of wait channel, if sleeping */
  threadstate_t t_state;           /* State this thread is in */ Every
                                      thread has a state such as
                                      running, sleeping, ready.
  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                   /* Kernel-level stack *, pointer to stack/
  struct switchframe *t_context;   /* Saved register context (on stack) *, pointer to switchframe
  struct cpu *t_cpu;               /
  struct proc *t_proc;             /* pointer to the CPU thread runs on */
  ...                              /* Process thread belongs to */
};
```

La struttura di un thread, in OS161, memorizza le informazioni dei thread che vengono eseguiti.
In particolare vi troviamo:

- Nome del thread
- Nome del wait channel, se il thread è in stato di sleep
- Stato del thread (running, sleeping, ready, zombie)
- Puntatore allo stack livello kernel (ogni thread ha uno stack in memoria user e uno stack a livello kernel)
- Puntatore allo switchframe del thread (ossia il contesto: il valore dei registri in un dato momento dell'esecuzione del thread)
- Puntatore alla CPU su cui il thread viene eseguito
- Puntatore al processo cui appartiene il thread

The OS161 Thread Structure

Data structure for saving thread information.
Each thread has name, state of execution (running, waiting, ..)

```
/* see kern/include/thread.h */
struct thread {
  char *t_name;                    /* Name of this thread */
  const char *t_wchan_name;        /* Name of wait channel, if sleeping */
  threadstate_t t_state;           /* State this thread is in */ Every
                                      thread has a state such as
                                      running, sleeping, ready.
  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                   /* Kernel-level stack */
  struct switchframe *t_context;   /* Saved register context (on stack) */
  struct cpu *t_cpu;               /* CPU thread runs on */
  struct proc *t_proc;             /* Process thread belongs to */
  ...
};
```

The OS161 Thread Structure

Each thread is associated to a stack, so there is a pointer at TCB that points to the area of stack, as part of kernel memory.

```
/* see kern/include/thread.h */

struct thread {
  char *t_name;                      /* Name of this thread */
  const char *t_wchan_name;          /* Name of wait channel, if sleeping */
  threadstate_t t_state;             /* State this thread is in */ Every
                                        thread has a state such as
                                        running, sleeping, ready.
  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                     /* Kernel-level stack */
  struct switchframe *t_context;     /* Saved register context (on stack) */
  struct cpu *t_cpu;                 /* CPU thread runs on */
  struct proc *t_proc;               /* Process thread belongs to */
  ...
};
```

The OS161 Thread Structure

Structure allows the OS to save the registers of the CPU associated to the thread. Switching from one thread to another thread, OS saves all the registers in this structure memorized as part of kernel memory.

```
/* see kern/include/thread.h */

struct thread {
  char *t_name;                      /* Name of this thread */
  const char *t_wchan_name;          /* Name of wait channel, if sleeping */
  threadstate_t t_state;             /* State this thread is in */ Every
                                        thread has a state such as
                                        running, sleeping, ready.
  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                     /* Kernel-level stack */
  struct switchframe *t_context;     /* Saved register context (on stack) */
  struct cpu *t_cpu;                 /* CPU thread runs on */
  struct proc *t_proc;               /* Process thread belongs to */
  ...
};
```

The OS161 Thread Structure

If a thread is in execution, in which CPU the thread is executed in written here. (in case of multi core processor)

```
/* see kern/include/thread.h */

struct thread {
  char *t_name;                      /* Name of this thread */
  const char *t_wchan_name;          /* Name of wait channel, if sleeping */
  threadstate_t t_state;             /* State this thread is in */ Every
                                        thread has a state such as
                                        running, sleeping, ready.

  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                     /* Kernel-level stack */
  struct switchframe *t_context;     /* Saved register context (on stack) */
  struct cpu *t_cpu;                 /* CPU thread runs on */
  struct proc *t_proc;               /* Process thread belongs to */
  ...
};
```

The OS161 Thread Structure

In os161, each thread knows its process. While thread knows its process, the process does not know its thread.

```
/* see kern/include/thread.h */

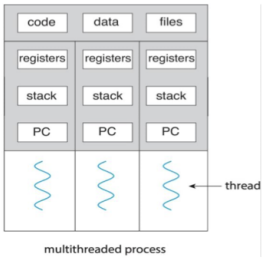
struct thread {
  char *t_name;                      /* Name of this thread */
  const char *t_wchan_name;          /* Name of wait channel, if sleeping */
  threadstate_t t_state;             /* State this thread is in */ Every
                                        thread has a state such as
                                        running, sleeping, ready.

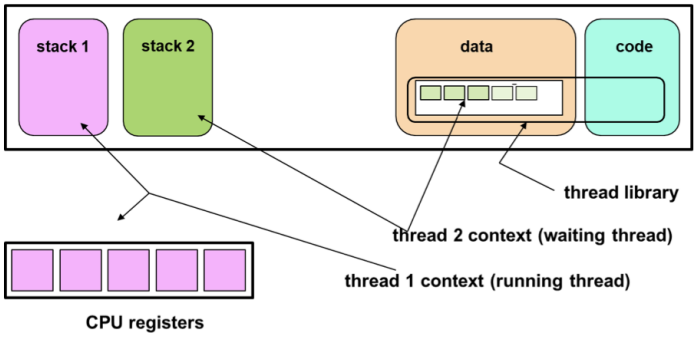
  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                     /* Kernel-level stack */
  struct switchframe *t_context;     /* Saved register context (on stack) */
  struct cpu *t_cpu;                 /* CPU thread runs on */
  struct proc *t_proc;               /* Process thread belongs to */
  ...
};
```

Il Thread conosce il processo a cui appartiene, ma non viceversa!
Quindi un processo
non può terminare un suo thread.
Questa è una peculiarità di OS161.. Potremmo implementare questa funzionalità in più.

Thread Library and Two Threads (generic)

A processor with two active threads:
1. Having two stacks for the threads
2. In the thread library, one of the two threads is active (violet one)
3. The inactive one should save its context and register in its stack
4. Switching thread in OS, recovering the context of one of the threads from stack, loading it to the CPU while saving the context of the running thread and saving it in its stack.





Thread Library

OS161 Thread Library

- Interface for bootstrap/shutdown (or panic)
  - `thread_bootstrap`
  - `thread_start_cpus`
  - `thread_panic`
  - `thread_shutdown`
- Interface for thread handling
  - `thread_fork`
  - `thread_exit`
  - `thread_yield`
  - `thread_consider_migration`
- Internal functions
  - `thread_create`
  - `thread_destroy`
  - `thread_make_runnable`
  - `thread_switch`

The functions that manage the thread in OS161

Lista di funzioni implementate in Os161 per gestire le operazioni di creazione di thread, context switch, terminare un thread etc..

Thread Structure

The OS161 Thread Structure

```
/* see kern/include/thread.h */

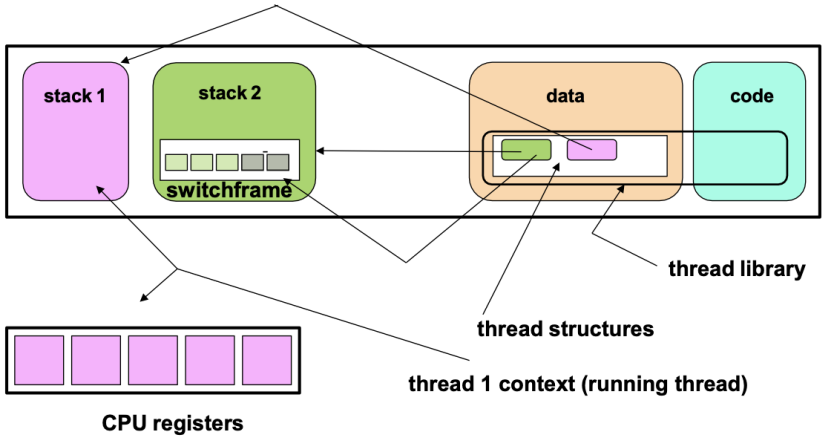
struct thread {
  char *t_name;                    /* Name of this thread */
  const char *t_wchan_name;        /* Name of wait channel, if sleeping */
  threadstate_t t_state;           /* State this thread is in */

  /* Thread subsystem internal fields. */
  struct  thread_machdep  t_machdep;
  struct threadlistnode t_listnode;
  void *t_stack;                   /* Kernel-level stack */
  struct switchframe *t_context;   /* Saved register context (on stack) */
  struct cpu *t_cpu;               /* CPU thread runs on */
  struct proc *t_proc;             /* Process thread belongs to */
  ... };
```

The two elements that allows saving the context in thread: kernel level stack and switch frame

Kernel level stack e switch frame: vengono usati per memorizzare il contesto di un thread.
Come detto prima, in generale un thread dispone di uno stack a livello utente e uno stack a livello kernel (vale a dire, uno nella memoria dedicata allo spazio d'indirizzamento utente, e uno nella memoria dedicata al kernel).
Questo perchè, ad esempio nel caso di una system call da parte di un thread utente (es. un'applicazione utente), il kernel deve avere uno stack isolato dalla memoria a livello user che non è sicura per eseguire il codice privilegiato.
D'altro canto, quindi, un kernel thread (cioè un thread di sistema che esegue di per sè delle operazioni privilegiate) usa soltanto lo stack a livello kernel, non essendo appunto un'applicazione utente.

Thread Library and Two Threads (OS161)



Quando facciamo switch di threads, dobbiamo memorizzare da qualche parte le informazioni del thread che viene messo in pausa.

The OS161 Thread Interface (incomplete)

The library of thread in OS161 is not complete. There are some functions to be done by US.

```
/* see kern/thread/thread.c */
```

- /* Make a new thread, which will start executing at “entrypoint". The thread will belong to the process "proc", or to the current thread's process if "proc" is null. The "data" arguments (one pointer, one *number) are passed to the function. */
- ```
  int thread_fork (const char *name, struct proc *proc,
          void (*entrypoint)(void *, unsigned long),
          void *data1, unsigned long data2);
  ```
- /* Cause the current thread to exit. Interrupts need not be disabled. */
  ```
  __DEAD void thread_exit(void);
  ```
- /* Cause the current thread to yield to the next runnable thread, but itself stay runnable. Interrupts need not be disabled. */
  ```
  void thread_yield(void);
  ```

Funzioni per creare, terminare e switchare (rispettivamente) un thread.

Thread_fork
  ```

The OS161 Thread Interface (incomplete)

```
/* see kern/thread/thread.c */
```

- /* Make a new thread, which will start executing at “entrypoint". The thread will belong to the process "proc", or to the current thread's process if "proc" is null. The "data" arguments (one pointer, one *number) are passed to the function. */

```
int thread_fork (const char *name, struct proc *proc,
        void (*entrypoint)(void *, unsigned long),
        void *data1, unsigned long data2);
```

Thread_fork, for creating a new thread. First step, defining an entry point which is a pointer to a zone of a memory (third parameter of the **thread_fork** function, a pointer to the address of a memory that contains the codes of the processor). In os161, the entry point has a fixed structure, expecting two parameters: first, a pointer to the data memory (the function to execute). The second is an integer number (a pointer to the parameters of the function).

Funzione per creare un thread

Quando creiamo un nuovo thread dobbiamo dedicare parte della memoria per il nuovo thread.
La funzione thread_fork crea un nuovo thread, prendendo come parametri:

- **nome** del nuovo thread
- **puntatore al processo** cui deve appartenere (in caso fosse Null, automaticamente verrebbe assegnato al processo del chiamante di thread_fork)
- **l’entrypoint** (cioè puntatore alla funzione che il nuovo thread deve iniziare ad eseguire)
- **i 2 parametri** dell’entrypoint (un puntatore e un number)

```
494  int
495  thread_fork(const char *name,
496          struct proc *proc,
497          void (*entrypoint)(void *data1, unsigned long data2),
498          void *data1, unsigned long data2)
499  {
500      struct thread *newthread;
501      int result;
502
503      newthread = thread_create(name);
504      if (newthread == NULL) {
505          return ENOMEM;
506      }
507
508      /* Allocate a stack */
509      newthread->t_stack = kmalloc(STACK_SIZE);
510      if (newthread->t_stack == NULL) {
511          thread_destroy(newthread);
512          return ENOMEM;
513      }
514      thread_checkstack_init(newthread);
515
516      /*
517       * Now we clone various fields from the parent thread.
518       */
519
520      /* Thread subsystem fields */
521      newthread->t_cpu = curthread->t_cpu;
522
523      /* Attach the new thread to its process */
524      if (proc == NULL) {
525          proc = curthread->t_proc;
526      }
527      result = proc_addthread(proc, newthread);
528      if (result) {
529          /* thread_destroy will clean up the stack */
530          thread_destroy(newthread);
531          return result;
532      }
533
534      /*
535       * Because new threads come out holding the cpu runqueue lock
536       * (see notes at bottom of thread_switch), we need to account
537       * for the spllower() that will be done releasing it.
538       */
539      newthread->t_iplhigh_count++;
540
541      /* Set up the switchframe so entrypoint() gets called */
542      switchframe_init(newthread, entrypoint, data1, data2);
543
544      /* Lock the current cpu's run queue and make the new thread runnable */
545      thread_make_runnable(newthread, false);
546
547      return 0;
548  }
```

Vediamo che per prima cosa la `thread_fork` chiama `thread_create` , una funzione che di fatto alloca in memoria una struttura di tipo thread, assegnandogli il nome

passato come parametro, il wait channel name “NEW” e lo stato di “Ready”.
Gli altri campi della struttura sono inizialmente assegnati a Null.
Adesso però, disponiamo di una struttura dati per il nostro thread.

```
111  /*
112   * Create a thread. This is used both to create a first thread
113   * for each CPU and to create subsequent forked threads.
114   */
115  static
116  struct thread *
117  thread_create(const char *name)
118  {
119      struct thread *thread;
120
121      DEBUGASSERT(name != NULL);
122
123      thread = kmalloc(sizeof(*thread));
124      if (thread == NULL) {
125          return NULL;
126      }
127
128      thread->t_name = kstrdup(name);
129      if (thread->t_name == NULL) {
130          kfree(thread);
131          return NULL;
132      }
133      thread->t_wchan_name = "NEW";
134      thread->t_state = S_READY;
135
136      /* Thread subsystem fields */
137      thread_machdep_init(&thread->t_machdep);
138      threadlistnode_init(&thread->t_listnode, thread);
139      thread->t_stack = NULL;
140      thread->t_context = NULL;
141      thread->t_cpu = NULL;
142      thread->t_proc = NULL;
143      HANGMAN_ACTORINIT(&thread->t_hangman, thread->t_name);
144
145      /* Interrupt state fields */
146      thread->t_in_interrupt = false;
147      thread->t_curspl = IPL_HIGH;
148      thread->t_iplhigh_count = 1; /* corresponding to t_curspl */
149
150      /* If you add to struct thread, be sure to initialize here */
151
152      return thread;
153  }
```

In seguito, la `thread_fork` alloca spazio per lo stack del thread, richiamando `kmalloc` con parametro `STACK_SIZE` , una costante globale che di default vale 4096 (4KB).
Se dopo di ciò
`newthread->t_stack == NULL` , allora la kmalloc non è andata a buon fine (spazio insufficiente per soddisfare la richiesta), dunque si procede a distruggere il nuovo thread creato tramite `thread_destroy` .
Se invece tutto va a buon fine, vediamo che

`thread_fork` procede a “popolare” i campi della struct del nuovo thread, tramite i campi del thread chiamante, assegnandogli la cpu su cui eseguire, il nome del processo cui deve appartenere.
Inoltre, chiama
`proc_addthread` che aggiorna il numero di threads appartenenti al processo, incrementandolo di 1.

```
/*
 * Add a thread to a process. Either the thread or the process might
 * or might not be current.
 *
 * Turn off interrupts on the local cpu while changing t_proc, in
 * case it's current, to protect against the as_activate call in
 * the timer interrupt context switch, and any other implicit uses
 * of "curproc".
 */
int
proc_addthread(struct proc *proc, struct thread *t)
{
    int spl;

    KASSERT(t->t_proc == NULL);

    spinlock_acquire(&proc->p_lock);
    proc->p_numthreads++;
    spinlock_release(&proc->p_lock);

    spl = splhigh();
    t->t_proc = proc;
    splx(spl);

    return 0;
}
```

Un’altra operazione importante che viene fatta, è inizializzare lo switchframe del nuovo thread tramite chiamata a `switchframe_init` che prende come parametri la struttura del nuovo thread, l’entrypoint e i parametri dell’entrypoint.

```
 * Function to initialize the switchframe of a new thread, which is
 * *not* the one that is currently running.
 *
 * The new thread should, when it is run the first time, end up calling
 * thread_startup(entrypoint, data1, data2).
 *
 * We arrange for this by creating a phony switchframe for
 * switchframe_switch() to switch to. The only trouble is that the
 * switchframe doesn't include the argument registers a0-a3. So we
 * store the arguments in the s* registers, and use a bit of asm
 * (mips_threadstart) to move them and then jump to thread_startup.
 */
void
switchframe_init(struct thread *thread,
                 void (*entrypoint)(void *data1, unsigned long data2),
                 void *data1, unsigned long data2)
{
        vaddr_t stacktop;
        struct switchframe *sf;

        /*
         * MIPS stacks grow down. t_stack is just a hunk of memory, so
         * get the other end of it. Then set up a switchframe on the
         * top of the stack.
         */
        stacktop = ((vaddr_t)thread->t_stack) + STACK_SIZE;
        sf = ((struct switchframe *) stacktop) - 1;

        /* Zero out the switchframe. */
        bzero(sf, sizeof(*sf));

        /*
         * Now set the important parts: pass through the three arguments,
         * and set the return address register to the place we want
         * execution to begin.
         *
         * Thus, when switchframe_switch does its "j ra", it will
         * actually jump to mips_threadstart, which will move the
         * arguments into the right register and jump to
         * thread_startup().
         *
         * Note that this means that when we call switchframe_switch()
         * in thread_switch(), we may not come back out the same way
         * in the next thread. (Though we will come back out the same
         * way when we later come back to the same thread again.)
         *
         * This has implications for code at the bottom of
         * thread_switch, described in thread.c.
         */
        sf->sf_s0 = (uint32_t)entrypoint;
        sf->sf_s1 = (uint32_t)data1;
        sf->sf_s2 = (uint32_t)data2;
        sf->sf_ra = (uint32_t)mips_threadstart;

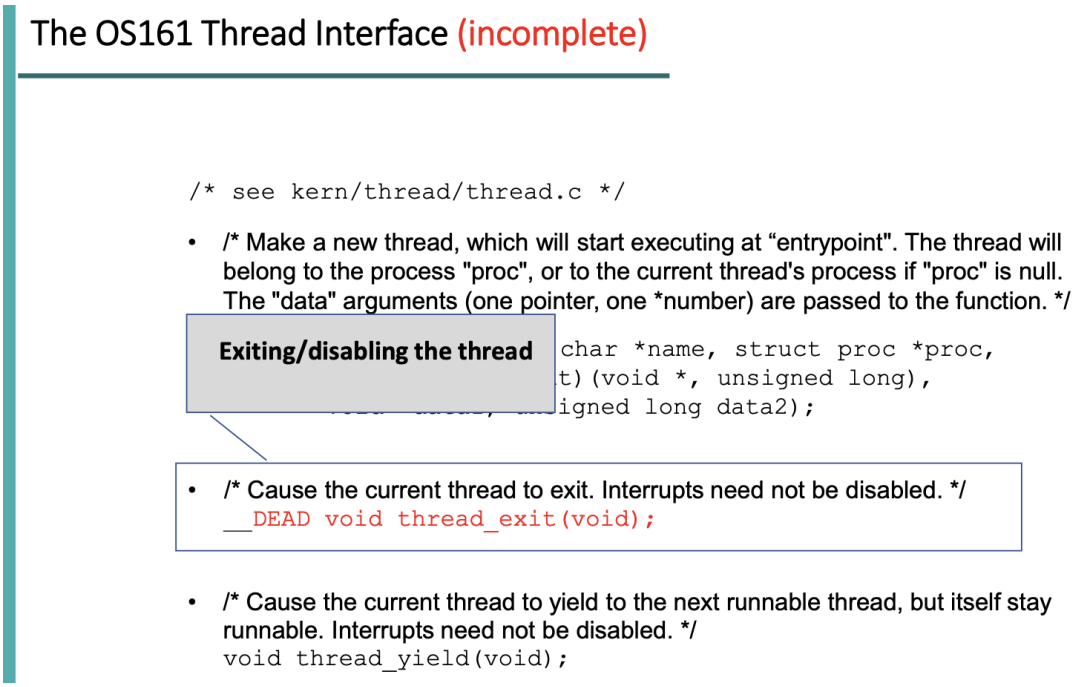
        /* Set ->t_context, and we're done. */
        thread->t_context = sf;
}
```

Senza scendere nei dettagli del funzionamento, essenzialmente questa funzione inizializza il contesto del nuovo thread: vale a dire che memorizza in appositi

campi (sf_s0 etc..) i valori che dovranno essere caricati nei registri MIPS quando il thread verrà schedulato per la prima volta.
In particolare, lo switchframe viene inizializzato memorizzando l'entrypoint del nuovo thread, coi relativi parametri, e il riferimento a `mips_threadstart` , una routine in assembly che va effettivamente a caricare i registri coi valori appena detti e fa un jump a `thread_startup` , che avvierà poi di fatto l'esecuzione del thread.
Tutto ciò avviene ovviamente quando, in un secondo momento, il thread viene schedulato per l'esecuzione.

```
445  /*
446   * Make a thread runnable.
447   *
448   * targetcpu might be curcpu; it might not be, too.
449   */
450  static
451  void
452  thread_make_runnable(struct thread *target, bool already_have_lock)
453  {
454      struct cpu *targetcpu;
455
456      /* Lock the run queue of the target thread's cpu. */
457      targetcpu = target->t_cpu;
458
459      if (already_have_lock) {
460          /* The target thread's cpu should be already locked. */
461          KASSERT(spinlock_do_i_hold(&targetcpu->c_runqueue_lock));
462      }
463      else {
464          spinlock_acquire(&targetcpu->c_runqueue_lock);
465      }
466
467      /* Target thread is now ready to run; put it on the run queue. */
468      target->t_state = S_READY;
469      threadlist_addtail(&targetcpu->c_runqueue, target);
470
471      if (targetcpu->c_isidle && targetcpu != curcpu->c_self) {
472          /*
473           * Other processor is idle; send interrupt to make
474           * sure it unidles.
475           */
476          ipi_send(targetcpu, IPI_UNIDLE);
477      }
478
479      if (!already_have_lock) {
480          spinlock_release(&targetcpu->c_runqueue_lock);
481      }
482  }
```

In fine, vediamo una chiamata a `thread_make_runnable` : senza scendere nei dettagli, essenzialmente imposta (di nuovo?) lo stato del nuovo thread a “Ready”, e lo aggiunge alla coda di run della cpu su cui deve eseguire.



Terminare un thread

The OS161 Thread Interface (incomplete)

```
/* see kern/thread/thread.c */
```

- /* Make a new thread, which will start executing at “entrypoint". The thread will belong to the process "proc", or to the current thread's process if "proc" is null. The "data" arguments (one pointer, one *number) are passed to the function. */
- `int thread_fork (const char *name, struct proc *proc,`
- `__dead void thread_exit(void);`

Forcing the context switch. Calling the **thread_yields** puts the calling thread on pause and ask the OS to schedule the next thread.

- /* Cause the current thread to yield to the next runnable thread, but itself stay runnable. Interrupts need not be disabled. */
  `void thread_yield(void);`

Mettere in pausa un thread

Metto in coda di ready il thread e dò la possibilità di usare la CPU ad un altro thread.
Viene usata dallo stesso thread che deve mettersi in pausa.

Creating Threads Using thread_fork()

Creating “i” number of threads

```
runthreads(int doloud){ char
  name[16];
  int i, result;
  for (i=0; i<NTHREADS; i++) {
     snprintf(name, sizeof(name), "threadtest%d", i); result =
     thread_fork(name, NULL,
                 doloud ? loudthread : quietthread, NULL, i); if (result)
     {
       panic("threadtest: thread_fork failed %s)\n",
       strerror(result));
     }
  }
  for (i=0; i<NTHREADS; i++) {
    P(tsem);
  }
}
```

Creating Threads Using thread_fork()

Defining names for the threads

```
runthreads(int doloud){ char
  name[16];
  int i, result;
  for (i=0; i<NTHREADS; i++) {
     snprintf(name, sizeof(name), "threadtest%d", i);
     result = thread_fork(name, NULL,
                doloud ? loudthread : quietthread, NULL, i); if (result)
     {
       panic("threadtest: thread_fork failed %s)\n",
       strerror(result));
     }
  }
  for (i=0; i<NTHREADS; i++) {
    P(tsem);
  }
}
```

Creating Threads Using thread_fork()

Calling thread_fork, passing the name, Null (associating the thread to the current processor), pointer to the function, Null (no paramtere passed), i as the variable)

```
runthreads(int doloud){ char
  name[16];
  int i, result;
  for (i=0; i<NTHREADS; i++) {
     snprintf(name, sizeof(name), "threadtest%d", i);
     result = thread_fork(name, NULL, doloud ? loudthread:
     quietthread, NULL, i);
                if (result) {
       panic("threadtest: thread_fork failed %s)\n",
       strerror(result));
     }
  }
  for (i=0; i<NTHREADS; i++) {
    P(tsem);
  }
}
```

From thread fork() to thread execution (ready state)

```
thread_fork(..., void (*entrypoint)(void *, unsigned
            long), void *data1, unsigned long data2) {
   ...
   newthread = thread_create(...);
   ...
   switchframe_init(newthread, entrypoint, data1,
   data2); thread_make_runnable(newthread, false);
}
thread_create(...) {
   thread =
   kmalloc(sizeof(*thread)); thread-
   >... = ...;
   return thread;
}
switchframe_init(...) {
   /* setup switchframe in stack */
}
thread_make_runnable(struct thread *target) {
   ...
   target->t_state = S_READY;
   threadlist_addtail(&targetcpu->c_runqueue,
   target);
   ...
}
```

Thread_fork is a wrapper for **thread_create, switchframe_init, and thread_make_runnable.**

Thread_fork è un wrapper delle funzioni thread_create, switchframe_init e thread_make_runnable.

From thread fork() to thread execution (ready state)

```
thread_fork(..., void (*entrypoint)(void *, unsigned
            long), void *data1, unsigned long data2) {
   ...
   newthread = thread_create(...);
   ...
   switchframe_init(newthread, entrypoint, data1,
   data2); thread_make_runnable(newthread, false);
}
thread_create(...) {
   thread =
   kmalloc(sizeof(*thread)); thread-
   >... = ...;
   return thread;
}
switchframe_init(...) {
   /* setup switchframe in stack */
}
thread_make_runnable(struct thread *target) {
   ...
   target->t_state = S_READY;
   threadlist_addtail(&targetcpu->c_runqueue,
   target);
   ...
}
```

Thread_create is for creating the thread, allocating the TCB and the necessary memory for managing the thread.

thread_create alloca memoria per un nuovo thread e crea il TCB necessario per gestire il thread.

From thread fork() to thread execution (ready state)

```
thread_fork(..., void (*entrypoint)(void *, unsigned
            long), void *data1, unsigned long data2) {
   ...
   newthread = thread_create(...);
   ...
   switchframe_init(newthread, entrypoint, data1,
   data2); thread_make_runnable(newthread, false);
}
thread_create(...) {
   thread =
   kmalloc(sizeof(*thread)); thread-
   >... = ...;
   return thread;
}
switchframe_init(...) {
   /* setup switchframe in stack */
}
thread_make_runnable(struct thread *target) {
   ...
   target->t_state = S_READY;
   threadlist_addtail(&targetcpu->c_runqueue,
   target);
   ...
}
```

Creating the switch frame, allowing to memorize the values of all the registers through the **switchframe_init** to initialize the frames.

Abbiamo detto che un thread ha bisogno di una parte della memoria a lui dedicata per memorizzare informazioni dello stato dei registri, quando dobbiamo fare un context switch.
L'inizializzazione di questa parte di memoria viene fatta tramite
switchframe_init

From thread_fork() to thread execution (ready state)

```
thread_fork(..., void (*entrypoint)(void *, unsigned
            long), void *data1, unsigned long data2) {
   ...
   newthread = thread_create(...);
   ...
   switchframe_init(newthread, entrypoint, data1,
   data2); thread_make_runnable(newthread, false);
}
thread_create(...) {
   thread =
   kmalloc(sizeof(*thread)); thread-
   >... = ...;
   return thread;
}
switchframe_init(...) {
   /* setup switchframe in stack */
}
thread_make_runnable(struct thread *target) {
   ...
   target->t_state = S_READY;
   threadlist_addtail(&targetcpu->c_runqueue,
   target);
   ...
}
```

Changing the status of the thread to ready and inserting the thread in the ready list to be scheduled

Quando un thread è pronto (ha fatto thread_create e switchframe_init) mette il suo status a ready per dire alla CPU che è pronto per essere eseguito, appena ve n'è la possibilità.

From ready to execution: thread_switch

```
thread_switch(threadstate_t newstate, ...) {
   struct thread *cur, *next;

   cur = curthread;
   /* Put the thread in the right place. */
   switch (newstate) {
       case S_RUN:
       panic("Illegal S_RUN in thread_switch\n");
       case S_READY:
       thread_make_runnable(cur, true /*have lock*/);
       break;
   }
   next = threadlist_remhead(&curcpu->c_runqueue);

   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
   ...
}
```

When there are multiple threads and thread switch is required:
Thread_switch is called that change the state of the current thread, taking a new thread from the list.
Calling the **switchframe_switch** to change the context.

Quando bisogna fare un thread switch usiamo la funzione ***Thread_switch***.
Prima di tutto si cambia lo stato del thread corrente
running → ready.
Poi dobbiamo memorizzare lo stato del thread che sta andando in pausa e "caricare" sulla cpu il contesto del thread schedulato, questo viene fatto in particolare dalla funzione
switchframe_switch.

From ready to execution: thread_switch

```
thread_switch(threadstate_t newstate, ...) {
   struct thread *cur, *next;

   cur = curthread;
   /* Put the thread in the right place. */
   switch (newstate) {
       case S_RUN:
       panic("Illegal S_RUN in thread_switch\n");
       case S_READY:
       thread_make_runnable(cur, true /*have lock*/);
       break;
   }
   next = threadlist_remhead(&curcpu->c_runqueue);

   /* do the switch (in assembler in switch.S) */
   switchframe_switch(&cur->t_context, &next->t_context);
   ...
}
```

While all the functions that manage the threads are written in C, the functions that manage the context are written in assembly.
To save the context, we should know the architecture of the processor.
OS161 >> processor with MIPS architecture

Anche se tutte le funzioni che gestiscono threads sono scritte in C, le funzioni che gestiscono il contesto sono scritte in **assembly**, quindi bisogna conoscere l'architettura su cui l'OS viene eseguito per fare queste operazioni.

Review: MIPS Register Usage

```
See also:  kern/arch/mips/include/kern/regdefs.h
R0, zero = ## zero (always    returns  0)
R1, at  = ## reserved   for use by assembler
R2,  v0  =  ##  return  value  /  system  call number
R3,  v1  =  ##  return  value
R4,  a0  =  ##  1st argument   (to subroutine)
R5,  a1  =  ##  2nd argument
R6,  a2  =  ##  3rd argument
R7,  a3  =  ##  4th argument
```

The organization of registers in MIPS architecture

Review: MIPS Register Usage

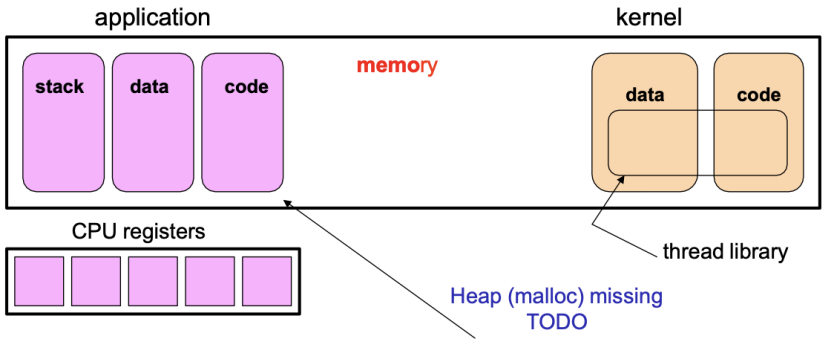
```
R08-R15, t0-t7 = ## temps (not preserved by subroutines)
R24-R25, t8-t9 = ## temps (not preserved by subroutines)
                 ## can be used without saving
R16-R23, s0-s7 = ## preserved by subroutines
                 ## save before using,
                 ## restore before return
R26-27,  k0-k1 = ## reserved for interrupt handler
R28,     gp =    ## global pointer
                 ## (for easy access to some variables)
R29,     sp =    ## stack pointer
R30,     s8/fp   ## 9th subroutine reg / frame pointer
R31,     ra =    ## return addr (used by jal)
```

In MIPS there are 31 registers, each group of registers has a meaning. Each register has two names, one is the actual name, the other one represents the characteristics of the register. When we switch the context, we save the status of these registers and then restore the new values.

Queste due slides non servono per l'esame.

User Process

Application (User Process) and Kernel



Layout the memory in OS161:
For each process, OS 161 can see the kernel and user side:
User side has stack, data and code.
The kernel side, the memory associated to the kernel.

Immaginiamo che come utente scriviamo un'applicazione.
Eseguiamo il codice e creiamo quindi un nuovo processo.
Questo nuovo processo (applicazione) ha bisogno di memoria per la parte di codice, data, stack etc..

Vediamo quindi che in memoria abbiamo il kernel con la sua zona in cui memorizza data, code, etc.. ma anche un processo utente ha bisogno di zone di memoria in cui memorizzare le sue informazioni.

Il SO deve gestire la creazione di un nuovo processo: cioè dedicare della memoria a questo processo.
Tutto ciò che vedremo ora riguarda la creazione di un processo utente.

Process structure

The OS161 proc structure

(PCB: Process Control Block)

```
/* see kern/include/proc.h */

 struct proc {
   char *p_name;                    /* Name of this process */
   struct spinlock p_lock;          /* Lock for this structure */
   unsigned p_numthreads;           /* Number of threads in this process */
   /* VM */
   struct addrspace *p_addrspace;   /* virtual address space */
   /* VFS */
   struct vnode *p_cwd;             /* current working directory */

   /* add more material here as needed */
```

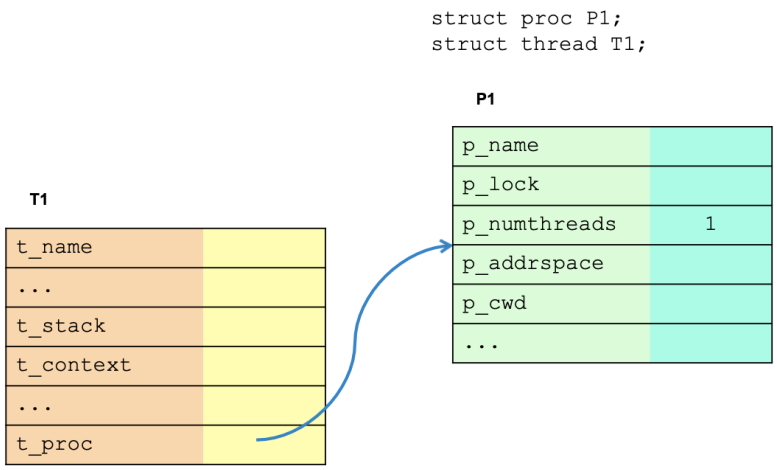
Every time a program is launched, a process is created >> data structure (PCB) is created. The data structure includes:
Process name, number of active thread (not knowing which threads), synchronization (spinlock), pointer to the virtual address space of the process (where the process is mapped in virtual address space, files)

Quando un utente chiede di eseguire un'applicazione, chiede quindi di creare un nuovo processo.
A livello di SO, quando viene creato un processo, il SO deve creare una struttura dati in memoria per memorizzare delle informazioni.

Se vogliamo vedere in OS161, in proc.h vediamo una *data structure* che viene usata per memorizzare informazioni per un processo.

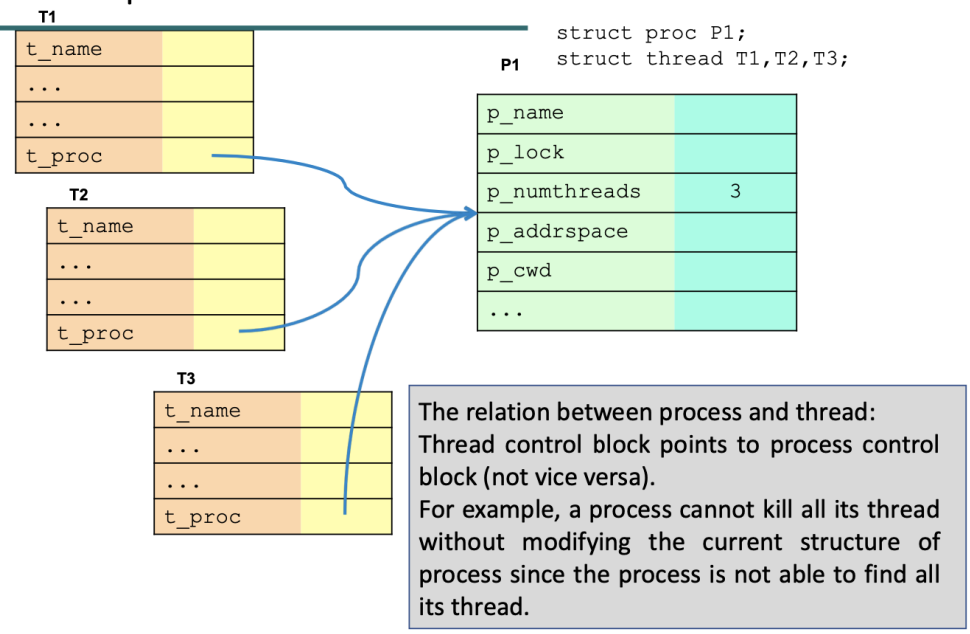
Ricordiamo che un thread sa a quale processo appartiene, ma un processo **non sa** quali siano i threads che gli appartengono.

Single threaded Process

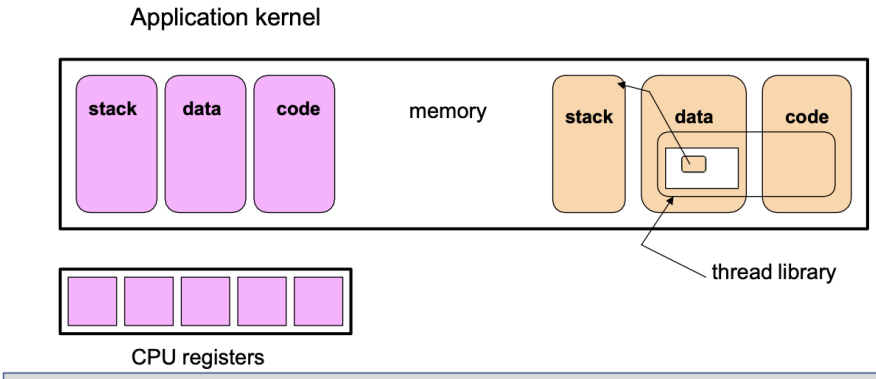


Vediamo quindi che nella struttura dati di un thread viene memorizzato il riferimento al processo cui appartiene.

Multi threaded process



OS161 User and Kernel Thread Stacks



Each OS161 thread has two stacks, one (user side) that is used while the thread is executing unprivileged (user) application code, and another (kernel side) that is used while the thread is executing privileged kernel code.

Ogni thread ha 2 stack:

- User level stack
  Usato per eseguire
  unprivileged application code
- Kernel level stack
  Usato per eseguire
  privileged kernel code

Running a user program

(GDB: set breakpoint on load_elf) from os161 menu

`p <elf_file> {<args>}`:

- p bin/cat <filename>
- p testbin/palin

- Menu calls `cmd_prog->common_prog`
  - `proc_create_runprogram`: create user process
  - `thread_fork`:
    thread executes `cmd_progthread->runprogram`
    - Generate address space
    - Read ELF file
    - Enter new process (kernel thread becomes USER thread)

What happens when we execute a program(p file name (.elf):
1. OS calls **proc_create_runprogram** to create a user process (create PCB and allocating memory)
2. Each process should have at least one thread, therefore, calling **thread_fork**.
3. **run_program** execution: generating the address space, loading the elf file into memory, starting the execution of the thread.

proc_create_runprogram viene usato per creare un processo utente.
Quando eseguiamo un'applicazione creiamo un processo e abbiamo bisogno di almeno 1 thread per eseguire tale processo.
Quindi la prima cosa che viene fatta, dopo aver creato il processo tramite *proc_create_runprogram*, è una chiamata a *thread_fork* per la creazione di tale thread.
Poi dobbiamo eseguire il processo e quindi si fa una chiamata a *run_program*

Vediamo nel dettaglio:

```
144  /*
145   * Command for running an arbitrary userlevel program.
146   */
147  static
148  int
149  cmd_prog(int nargs, char **args)
150  {
151      if (nargs < 2) {
152          kprintf("Usage: p program [arguments]\n");
153          return EINVAL;
154      }
155
156      /* drop the leading "p" */
157      args++;
158      nargs--;
159
160      return common_prog(nargs, args);
161  }
```

Quando eseguiamo un programma utente, ad esempio tramite comando `p testbin/palin` , viene chiamata la funzione `cmd_prog` , la quale a sua volta chiama `common_prog` , passandole fondamentalmente il nome del programma da eseguire.

```c
/*
 * Common code for cmd_prog and cmd_shell.
 *
 * Note that this does not wait for the subprogram to finish, but
 * returns immediately to the menu. This is usually not what you want,
 * so you should have it call your system-calls-assignment waitpid
 * code after forking.
 *
 * Also note that because the subprogram's thread uses the "args"
 * array and strings, until you do this a race condition exists
 * between that code and the menu input code.
 */
static
int
common_prog(int nargs, char **args)
{
    struct proc *proc;
    int result;

    /* Create a process for the new program to run in. */
    proc = proc_create_runprogram(args[0] /* name */);
    if (proc == NULL) {
        return ENOMEM;
    }

    result = thread_fork(args[0] /* thread name */,
            proc /* new process */,
            cmd_progthread /* thread function */,
            args /* thread arg */, nargs /* thread arg */);
    if (result) {
        kprintf("thread_fork failed: %s\n", strerror(result));
        proc_destroy(proc);
        return result;
    }

    /*
     * The new process will be destroyed when the program exits...
     * once you write the code for handling that.
     */

    return 0;
}
```

La funzione `common_prog` si occupa di chiamare `proc_create_runprogram`, la quale a sua volta chiamando `proc_create` e facendo altre operazioni, di base crea la struttura dati per il nuovo processo da eseguire.

```
187  /*
188   * Create a fresh proc for use by runprogram.
189   *
190   * It will have no address space and will inherit the current
191   * process's (that is, the kernel menu's) current directory.
192   */
193  struct proc *
194  proc_create_runprogram(const char *name)
195  {
196      struct proc *newproc;
197
198      newproc = proc_create(name);
199      if (newproc == NULL) {
200          return NULL;
201      }
202
203      /* VM fields */
204
205      newproc->p_addrspace = NULL;
206
207      /* VFS fields */
208
209      /*
210       * Lock the current process to copy its current directory.
211       * (We don't need to lock the new process, though, as we have
212       * the only reference to it.)
213       */
214      spinlock_acquire(&curproc->p_lock);
215      if (curproc->p_cwd != NULL) {
216          VOP_INCREF(curproc->p_cwd);
217          newproc->p_cwd = curproc->p_cwd;
218      }
219      spinlock_release(&curproc->p_lock);
220
221      return newproc;
222  }
```

Come abbiamo detto prima, poi, un processo ha bisogno di almeno un thread per eseguire, e quindi la funzione `common_prog` prosegue chiamando `thread_fork` , passandole il nome del processo (che quindi sarà anche il nome del thread), puntatore alla struttura del processo appena creata, e la funzione `cmd_progthread` come entrypoint..

Cosa è `cmd_progthread`?

```c
/*
 * Function for a thread that runs an arbitrary userlevel program by
 * name.
 *
 * Note: this cannot pass arguments to the program. You may wish to
 * change it so it can, because that will make testing much easier
 * in the future.
 *
 * It copies the program name because runprogram destroys the copy
 * it gets by passing it to vfs_open().
 */
static
void
cmd_progthread(void *ptr, unsigned long nargs)
{
    char **args = ptr;
    char progname[128];
    int result;

    KASSERT(nargs >= 1);

    if (nargs > 2) {
        kprintf("Warning: argument passing from menu not supported\n");
    }

    /* Hope we fit. */
    KASSERT(strlen(args[0]) < sizeof(progname));

    strcpy(progname, args[0]);

    result = runprogram(progname);
    if (result) {
        kprintf("Running program %s failed: %s\n", args[0],
            strerror(result));
        return;
    }

    /* NOTREACHED: runprogram only returns on error. */
}
```

Come detto prima, l'entrypoint è il codice che il thread esegue non appena viene schedulato per la prima volta.
Stiamo cercando di eseguire un programma utente, quindi come entrypoint passiamo la funzione
`cmd_progthread` , che si occupa appunto di avviare l'esecuzione di un programma utente.
Vediamo che è un wrapper della funzione
`runprogram` , che riceve in input il nome del programma da eseguire.

```c
/*
 * Load program "progname" and start running it in usermode.
 * Does not return except on error.
 *
 * Calls vfs_open on progname and thus may destroy it.
 */
int
runprogram(char *progname)
{
	struct addrspace *as;
	struct vnode *v;
	vaddr_t entrypoint, stackptr;
	int result;

	/* Open the file. */
	result = vfs_open(progname, O_RDONLY, 0, &v);
	if (result) {
		return result;
	}

	/* We should be a new process. */
	KASSERT(proc_getas() == NULL);

	/* Create a new address space. */
	as = as_create();
	if (as == NULL) {
		vfs_close(v);
		return ENOMEM;
	}

	/* Switch to it and activate it. */
	proc_setas(as);
	as_activate();

	/* Load the executable. */
	result = load_elf(v, &entrypoint);
	if (result) {
		/* p_addrspace will go away when curproc is destroyed */
		vfs_close(v);
		return result;
	}

	/* Done with the file now. */
	vfs_close(v);

	/* Define the user stack in the address space */
	result = as_define_stack(as, &stackptr);
	if (result) {
		/* p_addrspace will go away when curproc is destroyed */
		return result;
	}

	/* Warp to user mode. */
	enter_new_process(0 /*argc*/, NULL /*userspace addr of argv*/,
			  NULL /*userspace addr of environment*/,
			  stackptr, entrypoint);

	/* enter_new_process does not return. */
	panic("enter_new_process returned\n");
	return EINVAL;
}
```

La funzione `runprogram` si occupa dunque di:

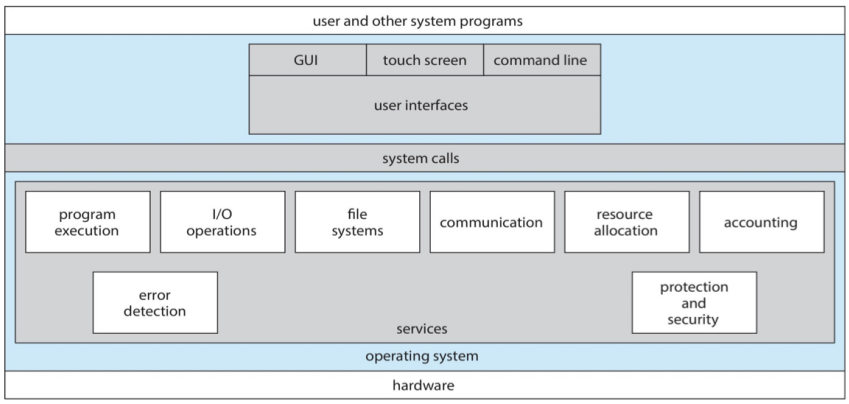
- Creare l'address space (cioè allocare una struttura di tipo addrspace tramite `as_create`) del processo

- Assegnarlo al processo stesso (tramite `proc_setas`) e attivarlo (?)
- Caricare l'eseguibile, assegnandolo all'entrypoint del processo
- Assegnare, tramite `as_define_stack`, al processo l'indirizzo dello stack utente (USERSTACK = 0x80000000, essendo uno stack descending)
- Finalmente tramite `enter_new_process` , senza scendere nei dettagli di quest'ultima, si inizia ad eseguire il programma utente

System Calls

A view of Operating System Services

- An operating system provides an environment for the execution of programs.
- It makes certain services available to programs and users of those programs.
- System calls provide an interface to the services made available by an operating system.
- Typically written in a high-level language (C or C++)



Per il secondo lab dobbiamo implementare delle syscalls.

In questa seconda parte vediamo la gestione di syscalls.

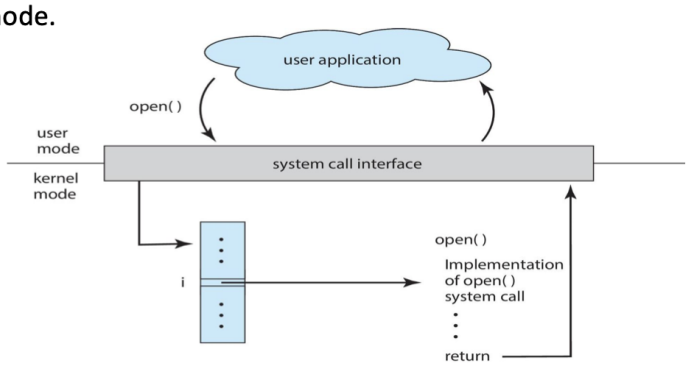
Cosa è una syscall?
Come utente noi non possiamo parlare mai direttamente con il SO.
Non possiamo chiedere direttamente al SO di fare delle operazioni.
Se dobbiamo fare un'operazione di I/O, ad esempio, chiediamo al kernel tramite syscall che dobbiamo fare un'operazione, e il kernel informa il SO che deve essere eseguita tale operazione.

System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship

- The OS associate a number or ID to each system call, also a table at the kernel space that says for each number of system call, what should be done.
- Every time that the program execute a system call (indecently from the system call), context switch from user mode to kernel mode is done, while passing the number associated to the system call.
- OS goes to the data structure of the system call number and executes the operations.
- When done, passing to the user mode.



Una interrupt che causa uno switch da user mode a kernel mode.
Informiamo il kernel di quale syscall stiamo usando.
A ogni sycall è associato un identificatore, quindi come utente quando scriviamo un'applicazione in cui ad esempio dobbiamo leggere un file, quando eseguiamo l'applicazione e arriviamo alla syscall sul file, facciamo uno switch da user mode a kernel mode perchè informiamo il kernel che abbiamo bisogno di gestire un file.

Il kernel prende l'id della syscall, va nella sua memoria e cerca il codice associato a questo identificatore, e comunica al SO che vorrebbe eseguire questa operazione richiesta dall'utente.
Si esegue quindi la syscall e alla fine si fa un altro switch da kernel mode a user mode, comunicando all'utente che la syscall è stata eseguita, restituendo un output che l'utente poi userà.

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

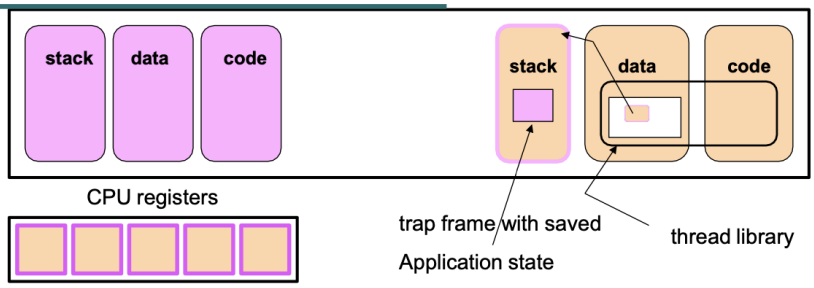
Mips trap: Handling System Calls, Exceptions, and Interrupts

- On the MIPS, the same exception handler is invoked to handle system calls, exceptions and interrupt
- The hardware sets a code to indicate the reason (system call, exception, or interrupt) that the exception handler has been invoked
- OS161 has a handler function corresponding to each of these reasons. The mips trap function tests the reason code and calls the appropriate function: the system call handler (mips syscall) in the case of a system call.
- Mips trap can be found in *kern/arch/mips/locore/trap.c*.

In OS161 implemented on MIPS, there is not difference between system calls, exceptions and interrupts.
In case each one of these three happens, the process is going to execute a particular code (handler).
A specific code is implemented to specify the reason of the exceptions. Therefore, three handler is defined for the three mentioned events.

In OS161, quando facciamo una syscall o un'eccezione o un'interruzione, praticamente è la stessa cosa (sono equivalenti).
Praticamente avviene un'interruzione in tutti e tre i casi, e questa interruzione va gestita tramite un handler.

OS161 Trap Frame

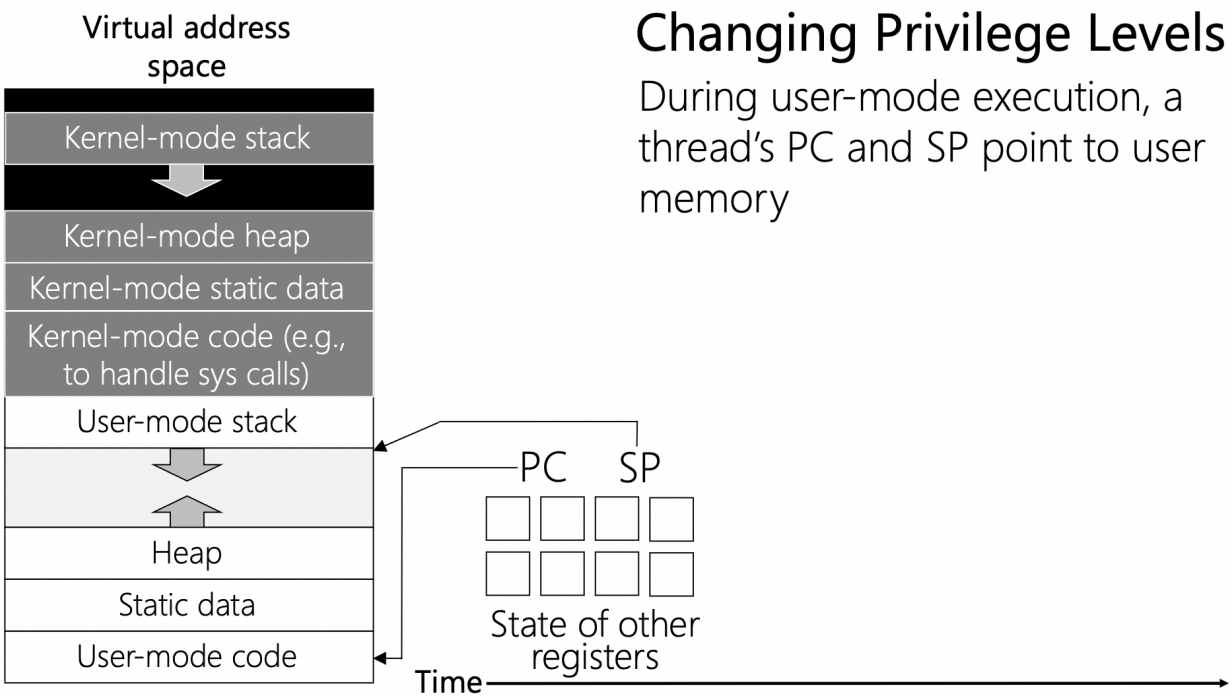


Every time a system call is called, the execution is passed from user mode to kernel mode. Therefore, the status of CPU should be saved through trap frame.
Trap frame will be saved at the stack of kernel thread.
So, while a thread is being executed, it is possible that the execution is interrupted (by system call, exception, interrupt).
Therefore, processors state should be saved for the execution of the system call and reloading it after.

While the kernel handles the system call, the application's CPU state is saved in a trap frame on the thread's kernel stack, and the CPU registers are available to hold kernel execution state.

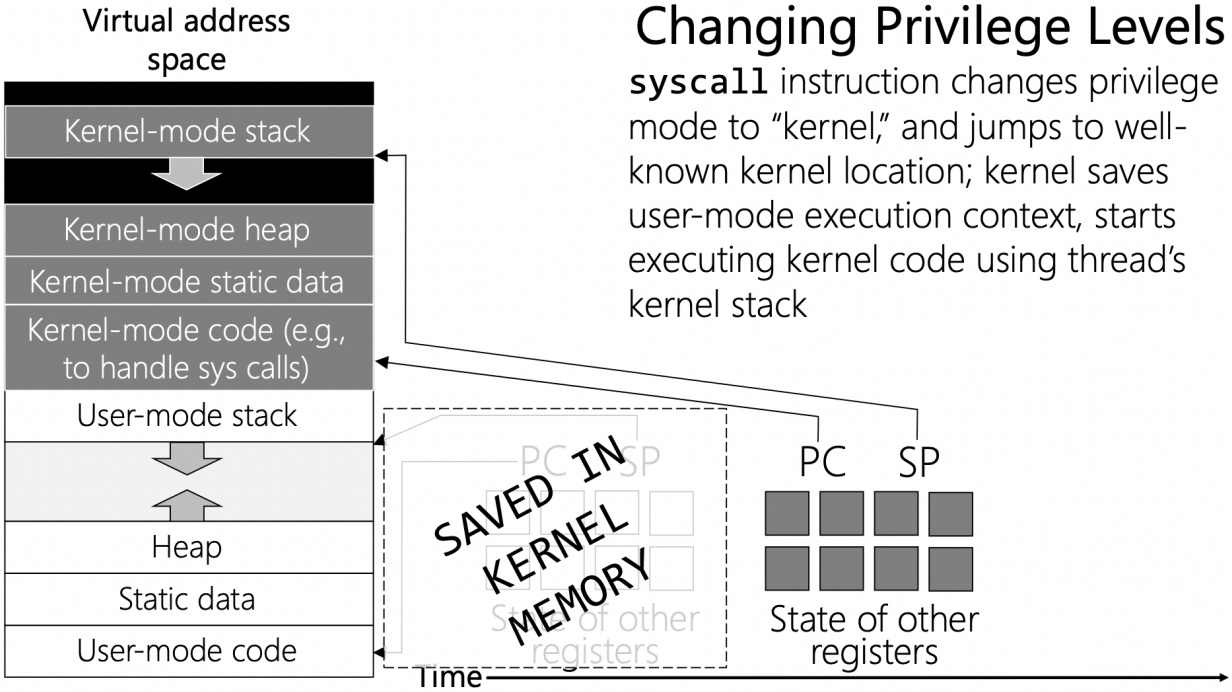
Changing Privilege Levels

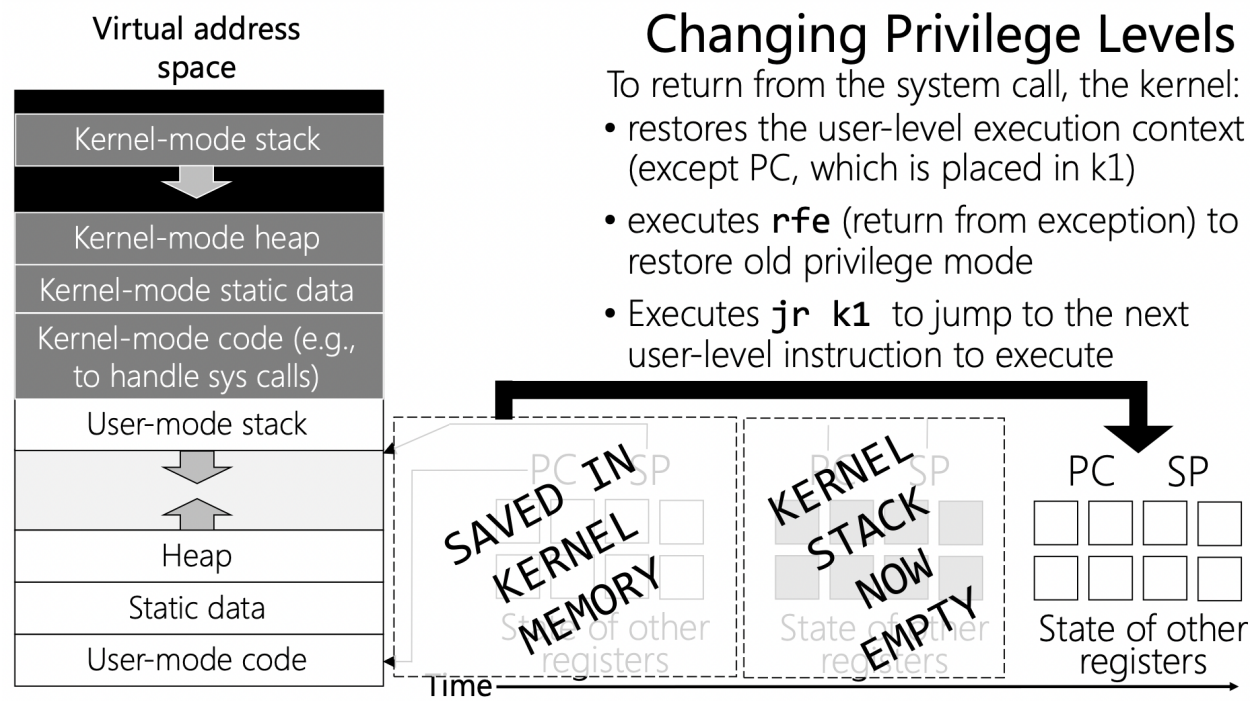
During user-mode execution, a thread's PC and SP point to user memory



Changing Privilege Levels

`syscall` instruction changes privilege mode to "kernel," and jumps to well-known kernel location; kernel saves user-mode execution context, starts executing kernel code using thread's kernel stack





OS161 MIPS System Call Handler

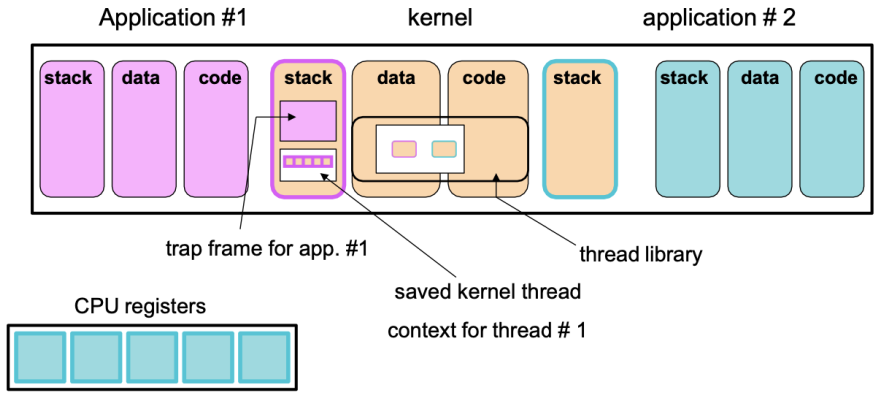
```
Void syscall(struct trapframe *tf) {
  ..
  callno = tf->tf_v0; retval = 0;
  switch (callno) {
    case SYS_reboot:
      err = sys_reboot(tf->tf_a0); /* in kern/main/main.c */
      break;

    /* Add stuff here */
    default:
      kprintf("Unknown syscall %d\n", callno);
      err = ENOSYS;
      break;
 }
```

Handler for system call in MIPS through switch case:
When a system call is called, the handler is executed which save the context of the thread in a trap frame, executing the code related to the system call (#),

In OS161 non tutte le syscall sono implementate (es syscall di readfile non è implementata).

Two Processes in OS161



More than one program in execution:
1. Two user memory address for the two program (application).
2. Data and code section.
3. Two kernel for the two threads for storing different data structure such as switch frame or trap frame.

Gestione della memoria completa in OS161.

Ogni applicazione ha il suo stack di livello kernel, perchè quando dobbiamo fare context switch il contesto dell'applicazione deve essere memorizzato, e viene usato lo stack del thread per memorizzare lo stato.

OS161 Memory Management

- Kernel/User address spaces
- Mips logical addresses
- Mips TLB
- DUMBVM virtual memory management
- Loading an ELF file into a (process) address space

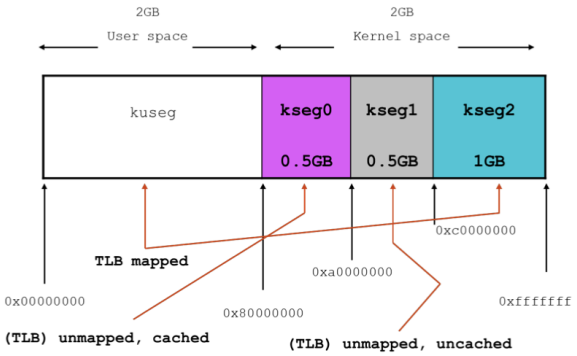
How the memory is organized in OS161:
The memory is divided into kernel and user space.
In Mips, a small MMU and TLB is provided.
The virtual memory management is organized by DUMBVM that performs pagination on contiguous allocation, but not allocating memory (lab 02).

Abbiamo visto che la memoria è divisa in due parti:

- Utente
- Kernel

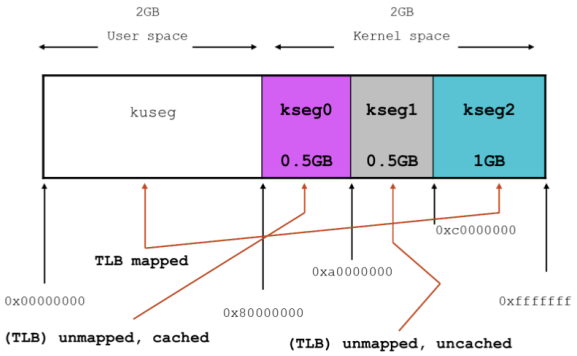
Quindi in totale in OS161 abbiamo in totale 4GB di memoria, e 2GB sono dedicato allo user space, e 2GB sono dedicati al kernel space.

Address Translation on the MIPS R3000



OS161 is running on Mips, a 32 bits processor and addresses at 32 bits. Therefore, 4GB of addresses.

Address Translation on the MIPS R3000



Dividing the memory to fix sections, using some sections for user and some sections for kernel, 2 GB user space, and 2 GB kernel space.
2GB kernel space is divided to subsections.
Kseg0, 0.5GB are not passed through TLB but cache (direct access to physical address).
Kseg1, 0.5 GB, not passed through TLB not cache, for mapping I/O devices.
Kseg2, 1 GB, not used in OS161.

Il kernel space è a sua volta diviso in 3 parti.

OS161 Address Spaces: dumbvm

- OS161 starts with a very simple virtual memory implementation
- virtual address spaces are described by addrspace objects, which record the mappings from virtual to physical addresses

```
struct addrspace { #if OPT_DUMBVM
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
#else
  /* Put stuff here for your VM system */ #endif
};
```

This amounts to a slightly generalized version of simple dynamic relocation, with three bases rather than one.

See kern/include/addrspace.h

Loading a Program into an address space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program's code and data into that address space
- A program's code and data is described in an *executable file*, which is created when the program is compiled and linked
- OS161 (and other operating systems) expect executable files to be in ELF(**E**xecutable and **L**inking **F**ormat)format
- the OS161 execv system call, which re-initialize the adress space of a process

```
#include <unistd.h>
int execv(const char *program, char **args);
```

- The program parameter of the execv system call should be the name of the ELF executable file for the program that is to be loaded into the address space.

ELF File

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space
- the ELF file identifies the (virtual) address of the program's first instruction
- the ELF file also contains lots of other information (e.g., section descriptors,symbol tables) that is useful to compliers, linkers, debuggers, loaders and other tools used to build programs.

Address Space Segments in ELF Files

- Each ELF segment describes a contiguous region of the virtual address space.
- For each segment, the ELF file includes a segment *image* and a header, which describes:

- the virtual address of the start of the segment
- the length of the segment in the virtual address space
- the location of the start of the image in the ELF file
- the length of the image in the ELF file

- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

To initialize an address space, the kernel copies images from the ELF file to the specifed portions of the virtual address space

ELF Files and OS161

- OS161's dumbvm implementation assumes that an ELF file contains two segments:
  - a *text segment*, containing the program code and any read- only data
  - a *data segment*, containing any other global program data
- the ELF file does not describethe stack (why not?)
- dumbvm creates a *stack segment* for each process. It is
- 12 pages long, ending at virtual address 0x7fffffff
- Look at kern/syscall/loadelf.c to see how OS161 loads segments from ELF files

sys__exit

Esaminiamo il funzionamento di un'importante syscall: sys__exit.

```
11  void sys__exit(int status) {
12      struct addrspace *as = proc_getas();
13      as_destroy(as);
14      thread_exit();
15      panic("thread_exit returned (this should not happen)\n");
16      (void) status; // TODO: status handling
17  }
```

Essa recupera l'address space del processo in esecuzione, tramite `proc_getas` . In seguito chiama `as_destroy` per distruggere tale address space (dellocare lo spazio dedicato al processo a livello user)

(Nella versione base vi è solo `kfree(as)` *, qui si riporta la versione di as_destroy completa dopo il lab sulla gestione della memoria)*

```
362  void as_destroy(struct addrspace *as){
363    dumbvm_can_sleep();
364    freeppages(as->as_pbase1, as->as_npages1);
365    freeppages(as->as_pbase2, as->as_npages2);
366    freeppages(as->as_stackpbase, DUMBVM_STACKPAGES);
367    kfree(as);
368  }
```

Dopo di che si chiama la funzione `thread_exit` .
Dobbiamo tenere a mente che in questo momento dell'esecuzione, ad eseguire è un kernel thread.
Infatti, anche se stavamo eseguendo un programma utente, nel momento in cui si fa una syscall, è un kernel thread a dover eseguire le operazioni privilegiate.
In questo caso stiamo eseguendo una syscall che si occupa di terminare il processo.

thread_exit

```
769  /*
770   * Cause the current thread to exit.
771   *
772   * The parts of the thread structure we don't actually need to run
773   * should be cleaned up right away. The rest has to wait until
774   * thread_destroy is called from exorcise().
775   *
776   * Does not return.
777   */
778  void
779  thread_exit(void)
780  {
781      struct thread *cur;
782
783      cur = curthread;
784
785      /*
786       * Detach from our process. You might need to move this action
787       * around, depending on how your wait/exit works.
788       */
789      proc_remthread(cur);
790
791      /* Make sure we *are* detached (move this only if you're sure!) */
792      KASSERT(cur->t_proc == NULL);
793
794      /* Check the stack guard band. */
795      thread_checkstack(cur);
796
797      /* Interrupts off on this processor */
798          splhigh();
799      thread_switch(S_ZOMBIE, NULL, NULL);
800      panic("braaaaaaaiiiiiiiiiiinssssss\n");
801  }
```

La funzione `thread_exit`

- **rimuove** il corrente thread dal processo cui appartiene
  (
  `proc_remthread` essenzialmente decrementa il numero di threads del processo cui appartiene il thread corrente e imposta a NULL il riferimento al processo del thread corrente)
- **chiama** `thread_switch`

thread_switch

```
550  /*
551   * High level, machine-independent context switch code.
552   *
553   * The current thread is queued appropriately and its state is changed
554   * to NEWSTATE; another thread to run is selected and switched to.
555   *
556   * If NEWSTATE is S_SLEEP, the thread is queued on the wait channel
557   * WC, protected by the spinlock LK. Otherwise WC and Lk should be
558   * NULL.
559   */
560  static
561  void
562  thread_switch(threadstate_t newstate, struct wchan *wc, struct spinlock *lk)
563  {
564      struct thread *cur, *next;
565      int spl;
566
567      DEBUGASSERT(curcpu->c_curthread == curthread);
568      DEBUGASSERT(curthread->t_cpu == curcpu->c_self);
569
570      /* Explicitly disable interrupts on this processor */
571      spl = splhigh();
572
573      cur = curthread;
574
575      /*
576       * If we're idle, return without doing anything. This happens
577       * when the timer interrupt interrupts the idle loop.
578       */
579      if (curcpu->c_isidle) {
580          splx(spl);
581          return;
582      }
583
584      /* Check the stack guard band. */
585      thread_checkstack(cur);
586
587      /* Lock the run queue. */
588      spinlock_acquire(&curcpu->c_runqueue_lock);
589
590      /* Micro-optimization: if nothing to do, just return */
591      if (newstate == S_READY && threadlist_isempty(&curcpu->c_runqueue)) {
592          spinlock_release(&curcpu->c_runqueue_lock);
593          splx(spl);
594          return;
595      }
596
597      /* Put the thread in the right place. */
598      switch (newstate) {
599          case S_RUN:
600          panic("Illegal S_RUN in thread_switch\n");
601          case S_READY:
602          thread_make_runnable(cur, true /*have lock*/);
603          break;
604          case S_SLEEP:
605          cur->t_wchan_name = wc->wc_name;
```

```
606             /*
607              * Add the thread to the list in the wait channel, and
608              * unlock same. To avoid a race with someone else
609              * calling wchan_wake*, we must keep the wchan's
610              * associated spinlock locked from the point the
611              * caller of wchan_sleep locked it until the thread is
612              * on the list.
613              */
614             threadlist_addtail(&wc->wc_threads, cur);
615             spinlock_release(lk);
616             break;
617             case S_ZOMBIE:
618             cur->t_wchan_name = "ZOMBIE";
619             threadlist_addtail(&curcpu->c_zombies, cur);
620             break;
621         }
622         cur->t_state = newstate;
623 
624         /*
625          * Get the next thread. While there isn't one, call cpu_idle().
626          * curcpu->c_isidle must be true when cpu_idle is
627          * called. Unlock the runqueue while idling too, to make sure
628          * things can be added to it.
629          *
630          * Note that we don't need to unlock the runqueue atomically
631          * with idling; becoming unidle requires receiving an
632          * interrupt (either a hardware interrupt or an interprocessor
633          * interrupt from another cpu posting a wakeup) and idling
634          * *is* atomic with respect to re-enabling interrupts.
635          *
636          * Note that c_isidle becomes true briefly even if we don't go
637          * idle. However, because one is supposed to hold the runqueue
638          * lock to look at it, this should not be visible or matter.
639          */
640 
641         /* The current cpu is now idle. */
642         curcpu->c_isidle = true;
643         do {
644             next = threadlist_remhead(&curcpu->c_runqueue);
645             if (next == NULL) {
646                 spinlock_release(&curcpu->c_runqueue_lock);
647                 cpu_idle();
648                 spinlock_acquire(&curcpu->c_runqueue_lock);
649             }
650         } while (next == NULL);
651         curcpu->c_isidle = false;
652 
653         /*
654          * Note that curcpu->c_curthread may be the same variable as
655          * curthread and it may not be, depending on how curthread and
656          * curcpu are defined by the MD code. We'll assign both and
657          * assume the compiler will optimize one away if they're the
658          * same.
659          */
660         curcpu->c_curthread = next;
661         curthread = next;
662 
663         /* do the switch (in assembler in switch.S) */
664         switchframe_switch(&cur->t_context, &next->t_context);
```

```
666     /*
667      * When we get to this point we are either running in the next
668      * thread, or have come back to the same thread again,
669      * depending on how you look at it. That is,
670      * switchframe_switch returns immediately in another thread
671      * context, which in general will be executing here with a
672      * different stack and different values in the local
673      * variables. (Although new threads go to thread_startup
674      * instead.) But, later on when the processor, or some
675      * processor, comes back to the previous thread, it's also
676      * executing here with the *same* value in the local
677      * variables.
678      *
679      * The upshot, however, is as follows:
680      *
681      *    - The thread now currently running is "cur", not "next",
682      *      because when we return from switchrame_switch on the
683      *      same stack, we're back to the thread that
684      *      switchframe_switch call switched away from, which is
685      *      "cur".
686      *
687      *    - "cur" is _not_ the thread that just *called*
688      *      switchframe_switch.
689      *
690      *    - If newstate is S_ZOMB we never get back here in that
691      *      context at all.
692      *
693      *    - If the thread just chosen to run ("next") was a new
694      *      thread, we don't get to this code again until
695      *      *another* context switch happens, because when new
696      *      threads return from switchframe_switch they teleport
697      *      to thread_startup.
698      *
699      *    - At this point the thread whose stack we're now on may
700      *      have been migrated to another cpu since it last ran.
701      *
702      * The above is inherently confusing and will probably take a
703      * while to get used to.
704      *
705      * However, the important part is that code placed here, after
706      * the call to switchframe_switch, does not necessarily run on
707      * every context switch. Thus any such code must be either
708      * skippable on some switches or also called from
709      * thread_startup.
710      */
711
712
713     /* Clear the wait channel and set the thread state. */
714     cur->t_wchan_name = NULL;
715     cur->t_state = S_RUN;
716
717     /* Unlock the run queue. */
718     spinlock_release(&curcpu->c_runqueue_lock);
719
720     /* Activate our address space in the MMU. */
721     as_activate();
722
723     /* Clean up dead threads. */
724     exorcise();
```

```
725
726     /* Turn interrupts back on. */
727     splx(spl);
728 }
```

La `thread_switch`, in questo caso, imposta lo stato del thread corrente a “Zombie” e lo aggiunge alla coda di threads zombie.
Ciò vuol dire che il thread ha terminato di eseguire, ma le sue strutture dati sono ancora allocate e devono essere rilasciate.
Dopo di che, la funzione prende un altro thread dalla coda di running e lo attiva per l’esecuzione..

La thread_switch fa anche tante altre cose, ma non scendiamo nei dettagli.
Notiamo solo che, ad un certo punto, chiama la funzione
`exorcise()`

exorcise

```
562   thread_switch(threadstate_t newstate, struct wchan *wc, struct spinlock *lk)
717       /* Unlock the run queue. */
718       spinlock_release(&curcpu->c_runqueue_lock);
719
720       /* Activate our address space in the MMU. */
721       as_activate();
722
723       /* Clean up dead threads. */
724       exorcise();
725
726       /* Turn interrupts back on. */
727       splx(spl);
728   }
```

```
290   /*
291    * Clean up zombies. (Zombies are threads that have exited but still
292    * need to have thread_destroy called on them.)
293    *
294    * The list of zombies is per-cpu.
295    */
296   static
297   void
298   exorcise(void)
299   {
300       struct thread *z;
301
302       while ((z = threadlist_remhead(&curcpu->c_zombies)) != NULL) {
303           KASSERT(z != curthread);
304           KASSERT(z->t_state == S_ZOMBIE);
305           thread_destroy(z);
306       }
307   }
```

La funzione `exorcise()` non fa altro che chiamare `thread_destroy` su tutti i threads nella coda di threads in stato di **zombie**.

thread_destroy

```
255  /*
256   * Destroy a thread.
257   *
258   * This function cannot be called in the victim thread's own context.
259   * Nor can it be called on a running thread.
260   *
261   * (Freeing the stack you're actually using to run is ... inadvisable.)
262   */
263  static
264  void
265  thread_destroy(struct thread *thread)
266  {
267      KASSERT(thread != curthread);
268      KASSERT(thread->t_state != S_RUN);
269
270      /*
271       * If you add things to struct thread, be sure to clean them up
272       * either here or in thread_exit(). (And not both...)
273       */
274
275      /* Thread subsystem fields */
276      KASSERT(thread->t_proc == NULL);
277      if (thread->t_stack != NULL) {
278          kfree(thread->t_stack);
279      }
280      threadlistnode_cleanup(&thread->t_listnode);
281      thread_machdep_cleanup(&thread->t_machdep);
282
283      /* sheer paranoia */
284      thread->t_wchan_name = "DESTROYED";
285
286      kfree(thread->t_name);
287      kfree(thread);
288  }
```

La `thread_destroy` rilascia la memoria associata al thread: rilascia lo stack a livello kernel allocato per il thread, e la struttura thread di per sè.

Notiamo che finora, non vi è alcuna operazione fatta per distruggere la struttura del processo, che rimane invece allocata anche se tutti i suoi threads sono stati distrutti!

È utile tenere a mente questa cosa perchè ci torneremo su nell'ultimo laboratorio, quello per l'implementazione della sycall `waitpid` , che consente di terminare correttamente l'esecuzione di un processo.