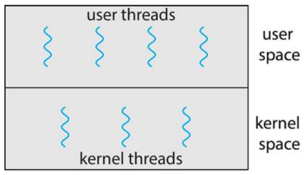
OS161 OVERVIEW

THREAD: stato di controllo di un programma in esecuzione con dei dati veri.

- Caratterizzato da **contesto/context**: ciò che viene salvato e ripristinato in caso di cambi.
  - Stato CPU: *PC, staco pointer, registri, modo di esecuzione privilegiato/non*
  - Stack, localizzato nello spazio di indirizzamento del processo
- Memoria:
  - Codice programma
  - Dati programma
  - Stack programma, contenenete registrazioni delle procedure di attivazione.

Thread si dividono in:

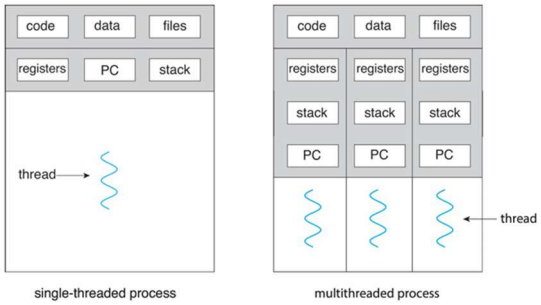
- **Thread-user**
  - Necessitano di librerie con funzioni per creare e gestire thread.
- **Thread-kernel** (noi vediamo solo questi)
  - Girano con modalità privilegiata in spazio di memoria virtuale user



Processo: programma in esecuzione che comprente:

- Codice programma → text section
- Program counter
- Stack
- Data section
- Heap (nota: in windows c'è anche la possibilità di separare)

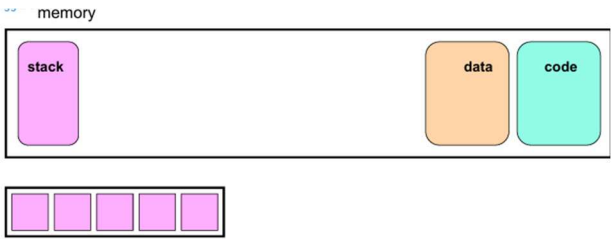
I thread in un processo condividono: codice, data e files; il resto è duplicato.



Nota: In contesti lavorativi si preferiscono multi-processi e non multi-thread. Multi-thread è più efficiente dal punto di vista della concorrenza ma meno usato in quanto: se in un processo multi-thread, si pianta un thread, si blocca tutto. Se siamo in multi-processo e si blocca un processo, gli altri continuano ad andare e faccio ripartire normalmente quello che si è piantato.

Contesto Thread:

In un processo posso avere più thread. Ogni processo ha il proprio codice, variabile globali e file che sono in comune per i suoi thread.
I thread possono girare in un multicore su cpu diverse e possono eseguire pezzi dello stesso programma.



Librerie Thread:

La libreria dei thread mette a disposizione delle **funzioni per gestire i thread**.
Una thread library può essere disponibile a livello user o a livello kernel.

- Thread sono **implementati** da una libreria di thread
- La libreria di thread salva i **contesti** dei thread quando questi non sono in esecuzione
    - **Salvati** in una struttura dati → per os161: thread structure

Os161 thread structure

Contiene:

- un puntatore ad una struttura dati di sincronizzazione
- lo stack
    - kernel level stack
    - Void * in quanto è un semplice indirizzo a memoria
- il contesto
    - Puntatore ad una struct in cui saranno salvati i registri
- il puntatore all cpu su cui gira il thread
- il puntatore al processo a cui appartiene il thread (nel caso in cui il thread appartenga al kernel, il processo sarà kernel)

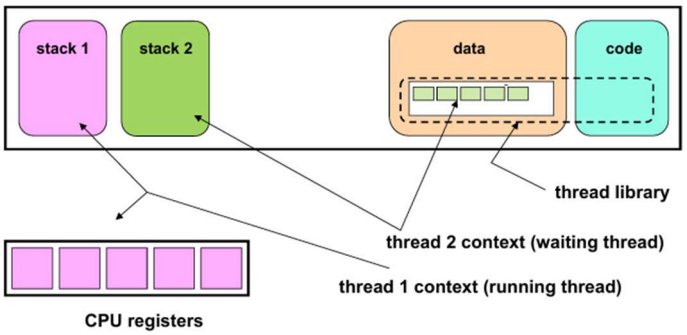
```
/* see kern/include/thread.h */

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 */
  ...
};
```

Nota: negli user ci sono anche i thread ma potrei non aver le librerie per gestirle, noi ci focalizziamo su kernel.

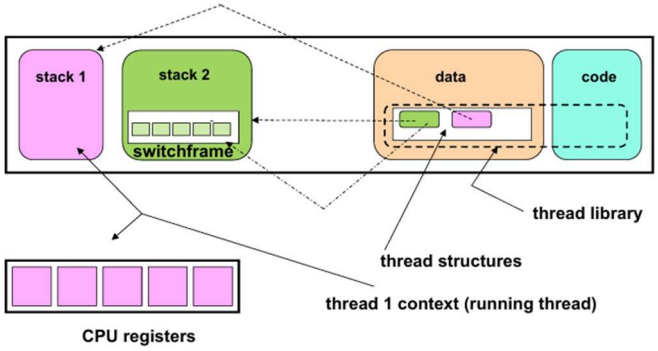
THREAD LIBRARY E 2 THREADS

Nel caso in cui si siano 2 thread e 1 CPU:

- Thread in esecuzione ha il suo stack e sta usando la CPU.
- Thead non attivo, è salvato, stack mantenuto in memoria e i suoi registri sono stati salvati.
    - Codice e dati kernel, con contesto salvato da qualche parte. (a dx)



- Dx codice thread
- Sx: struct thread
    - Ogni struct thread ha il puntatore in memoria al suo stack e al suo switch frame (che in os161 è nello stack).
    - Switchframe: contiene i registri che vanno salvati in cambio di contesto.



Thrad 1 è in esecuzione e infatti i suoi registri stanno in cpu.

Thread 2 è in pausa e i suoi registri stanno nello switchframe.

Funzioni

messe a disposizione dalla libreria sui thread:

- Funzioni per partenza/fine, usate quando servono

`thread_bootstrap`	
`thread_start_cpus`	
`thread_panic`	**Panic**→ gestire errori
`thread_shutdown`	**Shutdown**: chiude la gestione di tutti i thread

- Funzioni per gestione thread, chiamate al bootstrap

`thread_fork`	**fork** → crea thread
`thread_exit`	**exit**→ termina thread
`thread_yield`	**yield**→ termina esecuzione e fai posto a qualcun altro nella cpu, quindi cambi contesto
`thread_consider_migration`	

- Funzioni interne

`thread_create`	**create**:
`thread_destroy`	**destroy:** distrugge completamente (sarà chiamata da exit)
`thread_make_runnable`	**make_runnable**: schedula, lo mette in coda
`thread_switch`	**switch**: sarà chiamata da uield

Thread fork*:*

permette di creare un thread.

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

- Thread che chiama la thread fork:
  esegue una funzione che crea un nuovo thread e nelfrattempo lui continua a fare ciò che vuole fare.
- Per dire alla thread fork qual è **la funzione da eseguire**, si passa come parametro una funzione da eseguire passando un puntatore ad una funzione → ***entrypoint**
  La funzione che va passata alla thread_fork, deve essere consistente in nr di parametri e tipo → SONO FISSI.
- **Parametri**:
  - **Nome** thread
  - Puntatore al **processo** a cui il thread appartiene
  - **Funzione**
  - **void*** → primo parametro della funzione : solitamente puntatore ad una struct, può essere anche un puntatore ad un inizio di un vettore
  - **unsigned long** → secondo parametro funzione
- ***Nota:***
  - *In alcuni casi il thread appartiene al processo che li stanno creando ed è inutile passare il secondo parametro*
  - *In altri casi, il kernel crea una struttura dati processo e un kernel per farlo partire. In questo caso è molto inportante il seconod parametro*
- Esempio di chiamata:

Chiama o loudthread o quietthread ←

```
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);
  }
}
```

```
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);
   ...
}
```

Create:Creato nuovo thread
Switchframe: Predispongo il contesto: puntatore a thread, funzione che devo eseguire e parametri.
makeRunnable: metti in in coda alla lista della cpu per andare in esecuzione → READY.

nota: la fork mette il thread in coda ma non è detto che parti subito, partirà quando sarà chiamata la thread switch e thread sarà in testa.

Thread switch:

Funzione che **cambia il thread in esecuzione** sulla cpu:

- Riceve il nuovo stato del thread corrente (thread che deve abbandonare la cpu e lasciar posto ad un altro)
  - Nuovo stato== running → non va bene
  - Nuovo stato == ready → chiamo make_runnabble che lo manda in coda
- Estrae dalla testa per prendere il nuovo thread
  - Chiamo switchframe_switch che effettua il cambio di contesto, passando:
    - contesto del thread corrente → da salvare
    - contesto del thread nuovo → da ripristinare e quindi da usare (come puntatore by pointer)

```
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);
   ...
}
```

Nota: Thread yield chiama thread switch dicendogli che sono ready; thread switch ferma il thread in esecuzione e ne fa entrare un altro.

- *Per far ciò la thread switch deve avere*
  - *Puntatore a struct thread corrente: curthread*
  - *Puntatore a strutct thread next*

KERNEL THREAD TEST:

- tt1: call `threadtest->runthreads(1/*loud*/)` to generate `NTHREADS` (8) threads executing `loudthread`.
  8 threads mixing output of chars (120 chars each)
  (see `kern/test/threadtest.c`)
- tt2: call `threadtest2->runthreads(0/*quiet*/)` to generate `NTHREADS` (8) threads executing `quierthread`.
  8 threads doing busy wait (200000 for iterations) followed by output of 1 char (0..7)
  (see `kern/test/threadtest.c`)
- tt3: call `threadtest3->runtest3` to generate a certain number of threads doing sleep or work and synchronization
  (see `kern/test/tt3.c`)

loud: ci impiegano di più, fanno molti switch, hanno I/O

quiet: sono più veloci, fanno un input ad inizio e e un output alla fine

Come avviene lo switch: avviene il salvataggio dei registri del thread che viene sostituito e il ripristino dei registri del thread che viene ripristinato → decina di registri da 4 byte → **40 byte da salvare**.

- Salvataggio dati vecchio thread:
  - Prendi stack
  - Sposta stack pointer di 40 byte
  - Puoi usare questi 40 byte
    - Salva i registri → salva il contesto
  - Sw sp, 0(a0) → sto mettondo context = sp
- Ripristino dati nuovo thread

```
/* see kern/arch/mips/thread/switch.S */
switchframe_switch:
   /* a0/a1 point to old/new thread's switchframe (control block) */
   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -44

   /* Save the registers */
   sw ra, 36(sp)
   sw gp, 32(sp)
   sw s8, 28(sp)
   ...
   sw s1, 4(sp)
   sw s0, 0(sp)
   /* Store the old stack pointer in the old thread */
   sw sp, 0(a0) /* cur->t_context = sp; */  *a0 = sp;
```

Prendi il secondo parametro e modifica lo stackpointer → nuovo valore dello stackpointer per il nuovo context. Riprendi tutto e e inserisci nei registri.

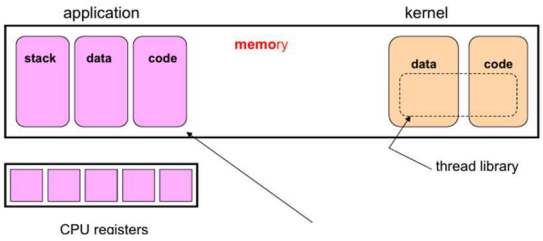
```
/* Get the new stack pointer from the new thread */
lw sp, 0(a1)  /* sp = next->t_context; */

nop /* delay slot for load */

/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s8, 28(sp)
lw gp, 32(sp)
lw ra, 36(sp)
nop /* delay slot for load */

j ra /* and return. */
addi sp, sp, 40 /* in delay slot */
.end switchframe_switch
```

nota: alcuni registri sono salvati quando avviene una chiamata alla funzione, altri no

```
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
```

```
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)
```

APPLICAZIONE E KERNEL



NOTA: kernel ha propria libreria;
processo user ha codice, dati e stack con registri (in os161 no heap tranne se qualcuno vuole implementare a mano la malloc).

Quando viene **creato un processo kernel**, ci sarà un processo user con struct proc:

```
/* 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 */
  ...
};
```

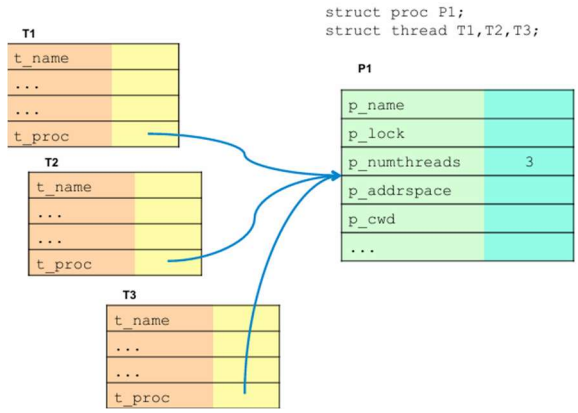
→ processo sa quanti thread ha → **num threads**

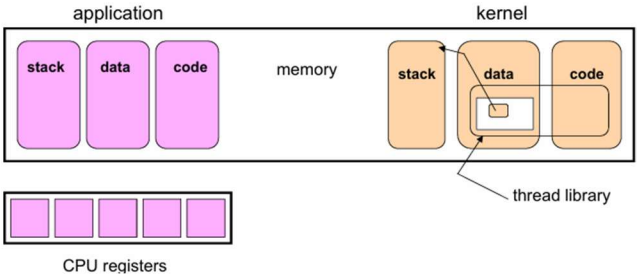
→ha il gestore della memoria virtuale→ **address space**

***Nota**: non ha un puntatore ai suoi thread di default, ma si può implementare facendo un vettore (sovradimensionato o riallocabile) o una lista linkata*

Un processo ha quindi:

- 1 struttura dati proc
- 1 struttura dati thread per ogni thread





Nota: quando un thread viene creato ha

- il proprio **stack di kernel** → dove vengono salvati i context quando si fa context switch)
- lo stack user dove ci sono variabili locali

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

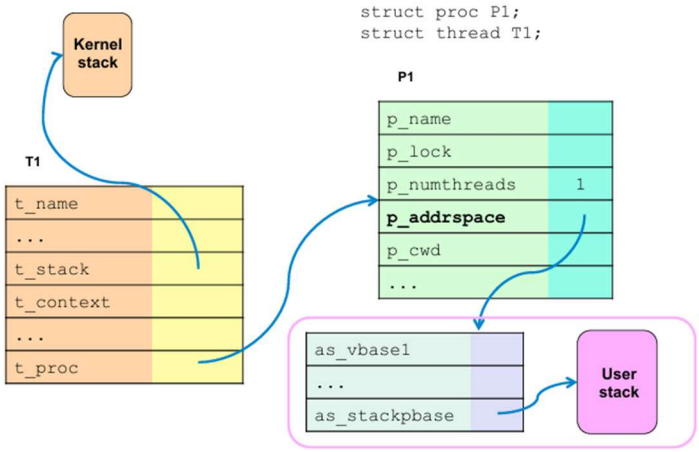
PROCESSO

File che contiene: codice, Variabili locali, strutture dati che metto in stack → eseguo processo e poi butto tutto.

Processo user in esecuzione, viene creato da un kernel thread che ha un suo stack, legge eseguibile, crea struttura dati allocando spazio di indirizzamento, carico eseguibile, thread in maniera privilegiata.

Addres space è una struct che contiene:

- ptatori in memoria a 3 intervalli contigui:
  - 1 → fisica+logica
  - 2 → fisica+ logica
  - Stack



***CREARE PROCESSO** USER IN OS161:*

- Dal menu
  - p <elf_file> {<args>}:
    - p bin/cat <filename>
    - p testbin/palin
- Menu effettua una catena di chiamate che arriva ad una **thread_fork** → thread di kernel che viene lanciato con obbiettivo **cmd_progthread** che ha un wrapper a runprogram.
  - Parte programma in esecuzione
  - Run program
    - Genera Spazio indirizzamento
    - Legge file formato elf
    - Kernel thread diventa uno user thread → nuovo processo

- Menu calls cmd_prog->common_prog
  - proc_create_runprogram: create user process
  - thread_fork:
    thread executes cmd_progthread->runprogram

runprogram

```
/* see kern/syscall/runprogram.c */
int runprogram(char *progname) {
  struct addrspace *as;
  struct vnode *v;   PUNTATORE A FILE APERTC
  vaddr_t entrypoint, stackptr;
  int result;

  /* Open the file. */   APRE FILE IN LETTURA
  result = vfs_open(progname, O_RDONLY, 0, &v);
  ...
  /* Create a new address space. */
  as = as_create(); CREA SPAZIO INDIRIZZAMENTO A PROCESSC
  ...
  /* Switch to it and activate it. */
  proc_setas(as);
  as_activate();

  /* Load the executable. */  entrypoint:PC dovrebbe partire con questo indirizzo
  result = load_elf(v, &entrypoint);
  ...     CREA SPAZIO DI INDIRIZZAMENTO: segmento, segmento, stack
  /* Done with the file now. */
  vfs_close(v);

  /* Define the user stack in the address space */
  result = as_define_stack(as, &stackptr);
  ...
  /* Warp to user mode. */
  enter_new_process(0/*argc*/, NULL/*userspace addr of argv*/,
            NULL /*userspace addr of environment*/,
            stackptr, entrypoint); THREAD diventa il nuovo processo
  /* enter_new_process does not return. */
  panic("enter_new_process returned\n");
  return EINVAL;
}
```

Trap frame: struttura dati di salvataggio e ripristino in corrispondenza di una trap o itnerrupt.

Per far **partire un programma user**, lo si mette in una situazione in cui pensa di dover fare una trapFromInterrupt → per farlo partire**, sfrutto il protocollo di ritorno da interrupt**.

Quindi per far partire un programma: **enter_new_process**

```
/* see kern/arch/mips/locore/trap.c */
void enter_new_process(int argc, userptr_t argv, userptr_t env
          vaddr_t stack, vaddr_t entry) {
    struct trapframe tf;        creo struct

    bzero(&tf, sizeof(tf));      la azzero

    tf.tf_status = CST_IRQMASK | CST_IEp | CST_KUp;
    tf.tf_epc = entry;  salvataggio del PC -> metto entry point
    tf.tf_a0 = argc;        parametri main
    tf.tf_a1 = (vaddr_t)argv;  parametri main
    tf.tf_a2 = (vaddr_t)env;   environment
    tf.tf_sp = stack;   stack

    mips_usermode(&tf);
}
void mips_usermode(struct trapframe *tf) {
   ...
    /* This actually does it. See exception-*.S. */
    asm_usermode(tf);
}
```

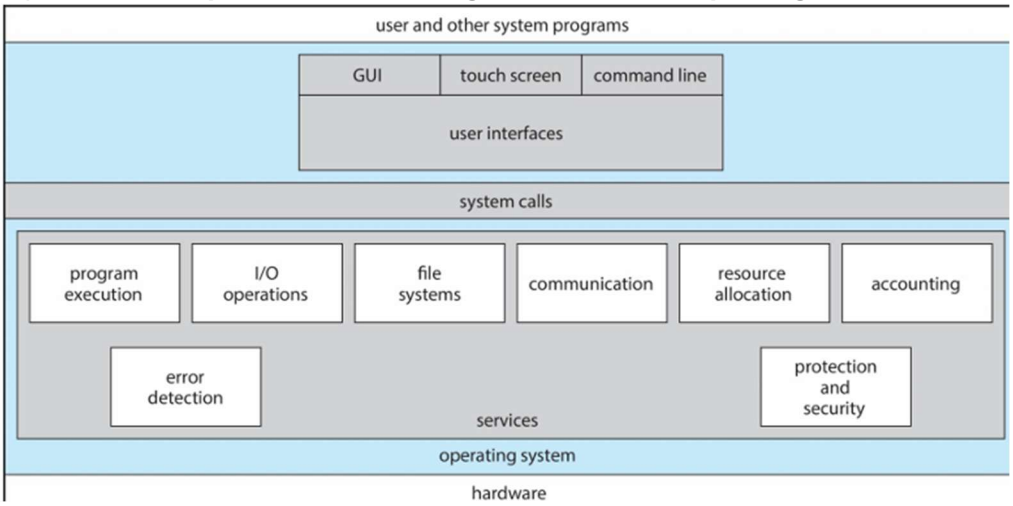
→ *tf_sp: stack pointer del processo user*

Gestendo come return for interrupt, lo faccio rornare in user mode.

Nient'altro che return from exception

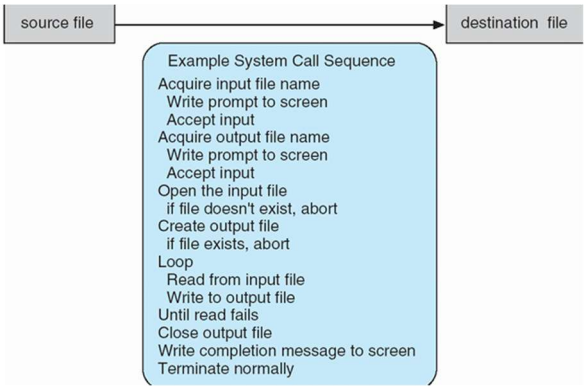
```
/* see kern/arch/mips/locore/exception-mips1.S */
asm_usermode:
   /* a0 is the address of a trapframe to use for exception "return".
    * It's allocated on our stack.
    * Move it to the stack pointer - we don't need the actual stack
    * position any more. (When we come back from usermode, cpustacks[]
    * will be used to reinitialize our stack pointer, and that was
    * set by mips_usermode.)
    * Then just jump to the exception return code above.
    */
   j exception_return          modifica stack e poi fa jump
   addiu sp, a0, -16              /* in delay slot */
   .end asm_usermode

exception_return:
   ... /* restore registers from trapframe */
   /* done */
   jr k0  /* jump (register) back */
   rfe    /* in delay slot: return from exception - resume user mode (if
   needed) */
   .end common_exception
```

Programma parte in modalità user, ma programma in modalità user non può accedere a tutto, ad esempio dipositivi IO. **Dispositivi in IO vanno gestiti in modalità privilegiata**.

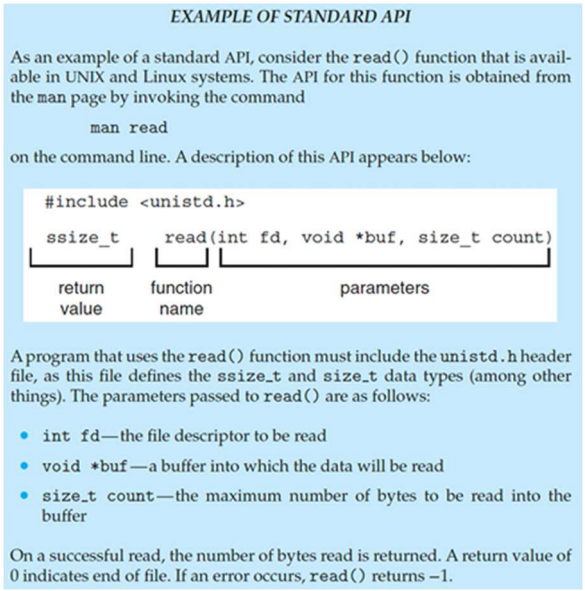


Quando c'è un programma user in esecuzione, il kernel fornisce una serie di servizi e ci sono le cosidette **system call** → interfaccia che il sistema operativo fornisce al programma utente per chiedere dei servizi.

Nota: se il nome del file è acquisito da tastiera:



Slide 46 → esempio



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

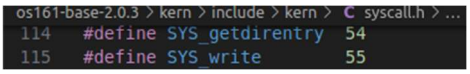
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Syscall:

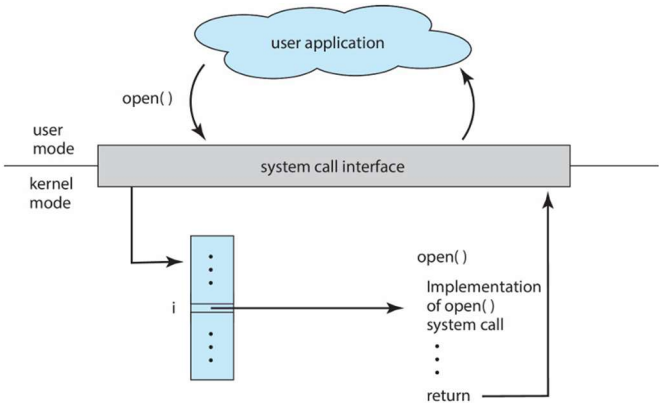
- Riceve 55 (vedendo il file syscall.h, vedi che è una SysWrite)
- In syscall.c ho i capture degli



SystemCall

Chiamate di sistema che reagiscono alle chiamate di un programma quando ha bisogno di un servizio dal sistema operativo.

Un programma user, per chiamare una funzione di sistema, deve chiamare delle funzioni privilegiate (non possono essere parte dell'eseguibile, programma user per usarle bisognerebbe fare una call a quel pezzo di kernel).
→ Bisogna fare interrupt via sw (*es: quello che si attiva in caso di divisioni per 0 o provando ad usare un puntatore con valore 0 → TRAP: interrupt sw → stai chiamando qualcosa che non è un pezzo del tuo eseguibile*). Se l'applicazione user chiama open in user mode → in **kernel mode**: si consulta tabella in cui si comprende se si tratta di interrupt, trap o eccezione e poi si accede ad altra tabella che mi dice quale **funzione eseguire**:



Dunque, possiamo dire che una call alla open non è altro che una **chiamata ad una trap**.

Come i metodi **passano parametri al sistemo operativo**?

- Semplice: passare i parametri ai **registri**
  - In alcuni casi ci possono essere più parametri di registri
- Parametri salvati in un **blocco o tabelle in memoria** e indirizzo del blocco passato come parametro del registro.

TRAP FRAME: x salvataggio e ritorno da una trap.
Anche il trap frame viene salvato nello stack, quando viene chiamata una sistem call:

- chiama sys_call
- gestore delle trap che corrisponde alla system call
- arriva un trap
- syscall vede un puntatore al trapfram (vede i registri salvati).
  Questi registri salvati vedono le cose che ci interessano