Leggete il diario!

http://matteo.vaccari.name/so/diario.php

Dopo ogni lezione trovate:
- gli argomenti
- le sezioni del testo corrispondenti
- i lucidi
- documentazione addizionale
- esercizi

Tutte le letture riportate nel diario **sono parte del programma di esame**, a meno che non siano esplicitamente dichiarate *facoltative*

Gestione dei processi nel kernel di Linux

Il descrittore del processo (PCB) in Linux è la `struct task_struct` in include/linux/sched.h

Lo stato è nel campo `state`

Possibili valori:
- TASK_RUNNING (running o ready)
- TASK_INTERRUPTIBLE (bloccato, *in attesa di un evento*)
- TASK_UNINTERRUPTIBLE (bloccato, *in attesa di un evento*; non può ricevere signal)
- TASK_STOPPED (bloccato indefinitamente)
- TASK_ZOMBIE

I processi sono identificati dall'indirizzo del descrittore

Il pid serve per implementare le syscall kill(2) e simili

Non è usato all'interno del kernel

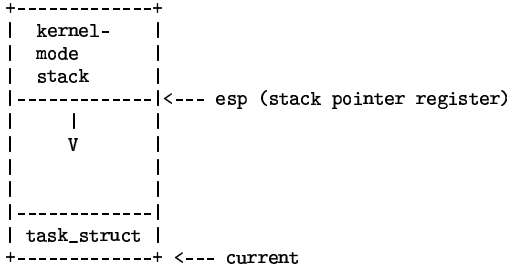
Una hash table mappa pid $\mapsto$ indirizzo descrittore

Dove sono allocati i descrittori?

I processi in kernel mode usano uno stack diverso

8KB sono allocati per il kernel mode stack

Il descrittore è allocato in fondo allo spazio libero per lo stack



La "variabile" `current`

Punta sempre al processo corrente

Non è una variabile, è una macro

- prendi il valore di `esp` (stack pointer)
- azzera i 13 bit meno significativi
- il risultato è l'indirizzo del descrittore

Es. `current->state` contiene sempre lo stato del processo
`current->p_pptr` punta al processo genitore

Esempio: implementazione di getuid(2)

Da kernel/timer.c

```
asmlinkage long sys_getuid(void)
{
        /* Only we change this so SMP safe */
        return current->uid;
}
```


Dove è implementata la *process table*?

E' implementata con una lista circolare e doppiamente linkata

```
struct task_struct {
        [ ... ]
        struct task_struct *next_task, *prev_task;
        [ ... ]
}
```

La lista `runqueue` contiene tutti i processi in stato TASK_RUNNING

Non c'è una lista per proc. in stato TASK_STOPPED o TASK_ZOMBIE;
sono reperibili tramite PID o tramite relazioni di parentela

I processi in stato TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE
sono accodati nella *wait queue* dell'evento atteso

Accodare un processo

Operazione fondamentale

Associamo una *coda* ad ogni risorsa

Wait queues

Meccanismo usato dappertutto nel kernel per bloccare un task in attesa di un evento

Definizione: in include/linux/wait.h

```
struct wait_queue {
  struct task_struct * task;
  struct wait_queue * queue;
};
```

creazione e inizializzazione:

```
wait_queue_head_t my_queue;
init_waitqueue_head (&my_queue);
```


Mettersi a dormire

Un task che vuole mettersi a dormire chiama una delle varianti di *sleep_on*

- `sleep_on(wait_queue_head_t *queue);`
- `interruptible_sleep_on(wait_queue_head_t *queue);`

Per risvegliare un task addormentato, si chiama

- `wake_up(wait_queue_head_t *queue);`

Esempio d'uso

```
DECLARE_WAIT_QUEUE_HEAD(wq);

ssize_t sleepy_read (.....)
{
  printk(KERN_DEBUG "process %i going to sleep\", current->pid);
  interruptible_sleep_on(&wq);
  printk(KERN_DEBUG "awoken %i\", current->pid);
  return 0; /* EOF */
}

ssize_t sleepy_write (.....)
{
  printk(KERN_DEBUG "process %i (%s) awakening the readers...\", current->pid, current
  wake_up_interruptible(&wq);
  return count; /* succeed */
}
```


Come funziona?

```
void simplified_sleep_on(wait_queue_head_t *queue)
{
   wait_queue_t wait;

   init_waitqueue_entry(&wait, current);
   current->state = TASK_INTERRUPTIBLE;
   add_wait_queue(queue, &wait);
   schedule();
   remove_wait_queue (queue, &wait);
}
```

La "magica" funzione `schedule()`

```
void schedule_simplified(void)
{
    struct task_struct* runnable;

    /* Get next thread to run from the run queue */
    runnable = get_next_runnable();

    /*
     * Activate the new thread, saving the context of the current thread.
     * Eventually, this thread will get re-activated and switch_to_thread()
     * will "return", and then schedule() will return to wherever
     * it was called from.
     */
    switch_to_thread(runnable);
}
```


Un esempio di implementazione di semafori (Xinu)

```
#ifndef NSEM
#define NSEM            50      /* number of semaphores, if not defined */
#endif

#define SFREE   1               /* this semaphore is free               */
#define SUSED   2               /* this semaphore is used               */

struct  sentry  {               /* semaphore table entry                */
        char    sstate;         /* the state SFREE or SUSED             */
        int     semcnt;         /* count for this semaphore             */
        int     sqhead;         /* q index of head of list              */
        int     sqtail;         /* q index of tail of list              */
};
extern  struct  sentry  semaph[];
extern  int     nextsem;

#define isbadsem(s)     (s<0 || s>=NSEM)
```

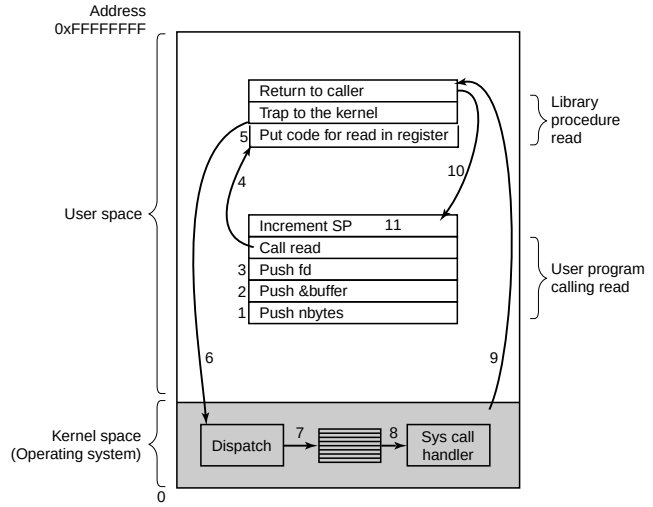

L'implementazione di *down* (wait)

```
SYSCALL wait(sem)
        int     sem;
{
        STATWORD ps;
        struct  sentry  *sptr;
        struct  pentry  *pptr;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if (--(sptr->semcnt) < 0) {
                pptr = &proctab[currpid];
                pptr->pstate = PRWAIT;
                pptr->psem = sem;
                enqueue(currpid,sptr->sqtail);
                resched();
        }
        restore(ps);
        return(OK);
}
```


L'implementazione di *up* (signal)

```
/*------------------------------------------------------------------------
 * signal  --  signal a semaphore, releasing one waiting process
 *------------------------------------------------------------------------
 */
SYSCALL signal(sem)
        int     sem;
{
        STATWORD ps;
        register struct sentry  *sptr;

        disable(ps);
        if (isbadsem(sem) || (sptr= &semaph[sem])->sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ((sptr->semcnt++) < 0)
                ready(getfirst(sptr->sqhead), RESCHYES);
        restore(ps);
        return(OK);
}
```

Torniamo sul meccanismo delle chiamate di sistema...



Come funzionano le system call? (Su architettura Intel)

0. mettere il *numero* della syscall nel registro EAX

1. eseguire la *trap* INT 0x80

L'implementazione di una semplicissima system call

```
asmlinkage long sys_getpid(void)
{
        /* This is SMP safe - current->pid doesn't change */
        return current->tgid;
}
```


Un'altra system call

```
asmlinkage long sys_nice(int increment) {
        long newprio;

        if (increment < 0) {
                if (!capable(CAP_SYS_NICE))
                        return -EPERM;
                if (increment < -40)
                        increment = -40;
        }
        if (increment > 40)
                increment = 40;

        newprio = current->nice + increment;
        if (newprio < -20)
                newprio = -20;
        if (newprio > 19)
                newprio = 19;
        current->nice = newprio;
        return 0;
}
```

Aggiungiamo una nuova system call

Creiamo un file kernel/matteo.c

```
#include <linux/matteo.h>

asmlinkage int sys_matteo() {
        return 42;
}
```


Dobbiamo fabbricare un *wrapper*

Quando scriviamo `getpid();` in un programma C, chiamiamo un wrapper

Che è implementato nella libc

Aggiungiamo una nuova system call

Il corrispondente include/linux/matteo.h

```
#include <linux/linkage.h>
#include <linux/unistd.h>

_syscall0(int, matteo)
```

`_syscall0` è una *macro* che genera il wrapper

definita in `<linux/unistd.h>`

usiamo pesantemente il *preprocessor* del linguaggio C

(per esaminare il risultato: `gcc -E`)

Posso esaminare il codice preprocessato con gcc -E

output di `gcc -E` (leggermente ripulito)

```
int matteo (void) {
  long __res;
  __asm__ volatile ("int $0x80" : "=a" (__res) : "0" (__NR_matteo ));
  return __res;
}
```

Ci occorre un *numero* per la system call

Esaminiamo include/asm-i386/unistd.h

```
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
[...]
```

aggiungiamo in fondo

```
/* MV */
#define __NR_matteo             243
```

stiamo modificando il kernel; aggiungiamo una sigla (MV) per ritrovare (con grep) tutti i punti che abbiamo toccato

System call table

Dobbiamo modificare la tabella delle system call
file arch/i386/kernel/entry.S

```
.data
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
[...]
```

aggiungiamo in fondo

```
        .long SYMBOL_NAME(sys_matteo)           /* MV */
```


Modifichiamo il Makefile

file kernel/Makefile

troviamo le righe

```
obj-y     = sched.o dma.o fork.o exec_domain.o panic.o printk.o \
            module.o exit.o itimer.o info.o time.o softirq.o resource.o \
            sysctl.o acct.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o context.o
```

e aggiungiamo il nostro file

```
obj-y     = sched.o dma.o fork.o exec_domain.o panic.o printk.o \
            module.o exit.o itimer.o info.o time.o softirq.o resource.o \
            sysctl.o acct.o capability.o ptrace.o timer.o user.o \
            signal.o sys.o kmod.o context.o matteo.o
```

e ricompiliamo il kernel!

Riassumendo...

0. aggiungiamo i file kernel/matteo.c e include/linux/matteo.h
1. rubiamo un numero di syscall e lo #definiamo in include/asm-i386/unistd.h
2. aggiungiamo una riga alla tabella delle syscall in arch/i386/uni-std
3. modifichiamo il kernel/Makefile

Esercitiamo la nuova syscall

compiliamo con gcc -I/home/matteo/linux-2.4.19/include

```
#include <linux/matteo.h>
void main() {
  printf("Il valore restituito da matteo() è %d\n'', matteo());
}
```


Esercizio

Realizzare quanto visto nelle slide precedenti

- scaricare i sorgenti di linux (quanto visto sopra è per Linux 2.4)
- applicare le modifiche
- compilare il kernel
- fare il boot con il nuovo kernel (in alternativa, usare User Mode Linux)
- esercitare la nuova syscall

Esercizio, parte A

Implementare un nuovo meccanismo di sincronizzazione che permette a più di un processo di bloccarsi in attesa di un *evento*, fino a che un altro processo non *segnala* l'evento.

Quando l'evento viene segnalato, tutti i processi bloccati si sbloccano

Se non ci sono processi in attesa, la segnalazione non ha effetto

Occorre implementare le seguenti syscall:

- int event_open(int n);
  se n = 0, crea un nuovo evento e ne restituisce un ID
  se n > 0, informa il kernel che il processo vuole usare un evento esistente
- int event_close(int n);
  distrugge un evento
- int event_wait(int n);
  blocca fino a che l'evento non è segnalato
- int event_signal(int n);
  segnala l'evento

In ogni caso un risultato negativo indica fallimento

Esercizio, parte B

Scrivere un programma applicativo per testare le nuove syscall

Il programma deve testare il caso normale (es. un processo in attesa) e i casi di confine:
- chiamare event_signal() quando non ci sono processi in attesa
- chiamare event_signal() con due o più processi in attesa
- funzionamento con più eventi allocati contemporaneamente
- processi in attesa quando viene chiamata event_close()

In kernel mode non abbiamo le librerie

Si usa la funzione printk() per stampare sulla console
- include/linux/kernel.h
- kernel/printk.c

Si usano kmalloc() e kfree() per allocare memoria
- include/linux/slab.h
- mm/slab.c

Problema: reference counting degli eventi

Se vogliamo implementare correttamente la event_close(), dobbiamo contare il numero di processi che hanno aperto un evento.

Occorre modificare:
- task_struct
- fork
- exit

In prima approssimazione, possiamo evitare di implementare la event_close()

Per capire il kernel, occorre leggere il codice

`http://lxr.linux.no/source/`

contiene il codice del kernel in forma di ipertesto

Per approfondimenti:

Linux Device Drivers, 2nd Edition
Alessandro Rubini & Jonathan Corbet
`http://www.xml.com/ldd/chapter/book/`