

Analisi del kernel xv6

Gruppo 37

A cura di
Marco Lampis e Davide Vilella

xv6

**xv6: A Simple,
Unix-like Teaching
Operating System**

*Russ Cox, Frans Kaashoek,
Robert Morris*

Confronto tra os161 e xv6

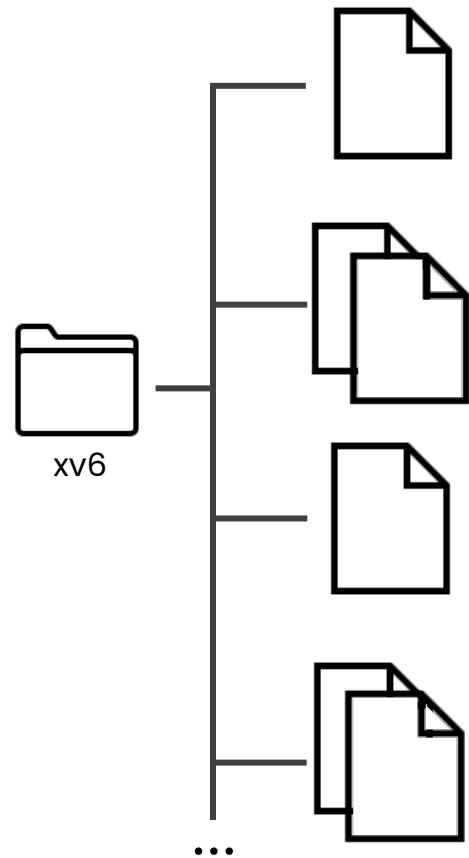
Parte 1

Introduzione a xv6

- Kernel didattico ideato e mantenuto dal **MIT** tra il 2006 e il 2018
- Architettura **x86** a **32bit** con set di istruzioni **CISC**
- Sviluppato in C ed Assembly
- Basato su UNIXv6
- Nel 2020 è stato realizzato il porting a RISC-V
- Kernel **multi-processo** ma non multi-thread



Organizzazione



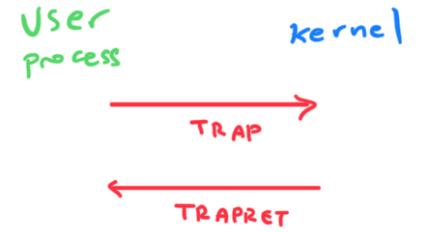
defs.h: file unico contenente le definizioni delle funzioni implementate nel kernel

syscall.c e syscall.h: contengono le definizioni e l'associazione delle system call

sysproc.c: contiene l'implementazione delle syscall e richiama le funzioni implementate nei file dedicati

ulib.c e user.h: implementazione e definizione delle funzioni di libreria disponibili per l'utente

Trap e Interrupt in x86



In x86 si distingue **trap** e **interrupt** in base alla causa della interruzione:

- Le **interruzioni** vengono generate da parte dell'hardware per segnalare il completamento o attesa di un'operazione.
- Le **trap** vengono generate da eccezioni a livello utente o per richiamare le system call nelle kernel routine.

Per generare un interrupt viene utilizzata l'istruzione `int interrupt_number`.

Sono presenti 256 interrupt handler elencati nella **Interrupt Descriptor Table**.

System Call

Il kernel mette a disposizione delle primitive di sistema per separare l'attività utente da quella del sistema.

- Viene sfruttato il meccanismo di **interrupt**.
- Ogni processo ha a disposizione una memoria dati riservata nel kernel, caricata al momento del context switching.
- I permessi sono separati per privilegio kernel e utente anche nelle modalità del processore (x86).

System Call

L'implementazione delle system call è suddivisa su più file:

- **syscall.h**: definizione delle primitive e assegnamento del numero identificativo per la chiamata.
- **syscall.c**: associazione del identificativo alla primitiva, in modo da essere richiamata tramite la primitiva `syscall()`.
- **sysproc.c**: implementazione della system call e del recupero dei parametri forniti.
- **usys.s**: definisce le primitive richiamabili per l'utente.

System calls

fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return the current process's pid
sleep(n)	Sleep for n clock ticks
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, maj, min)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Processi

xv6 mette a disposizione la struttura dati proc per l'astrazione di un processo. Il multi threading non è supportato, ogni processo è dotato di un solo thread principale.

Context è un puntatore allo stack del contesto dove vengono memorizzate le informazioni nel context switching.

In state è indicato lo stato del processo, che può essere: *SLEEPING*, *RUNNABLE*, *RUNNING*.

```
// proc.c
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};
```

Tabella dei processi

Tutti i processi creati sono indirizzati nel vettore di processi proc[] all'interno della struttura dati **ptable**.

Per operare sulla struttura dati è necessario richiedere e rilasciare lo spinlock **ptable.lock**.

Per mettere un processo in esecuzione è necessario:

- Mettere p->state = RUNNING
- Richiamare la funzione swtch()

```
// proc.c

struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

Context switching

xv6 implementa l'**isolamento forte** in modo da impedire ai processi di interferire tra di loro e accedere alla memoria riservata, suddivisa in **user stack** e **kernel stack**.

Viene richiamato all'interno di trapasm.s la funzione alltraps mediante interrupt, creando il trapframe per poi richiamare trap() da trap.c

Trap determina la natura della interruzione identificando la system call tramite syscall() effettuando il fetch dell'id.

Il context switch è ultimato dalla funzione trapret.

xv6 vs os161

Processi e context switching

- xv6 è single-thread, mentre os161 è multi-thread.
- Funzionamento tramite interrupt e gestione delle trap analogo, lievi variazioni sul piano implementativo dovuto alla differenza di architettura.
- I processi in xv6 non implementano il livello di priorità.

Meccanismi di sincronizzazione

I meccanismi di sincronizzazione sono strumenti utilizzati nella programmazione concorrente per coordinare l'esecuzione di più thread o processi al fine di evitare problemi come la competizione per risorse condivise (race conditions), accessi concorrenti imprevedibili e errori di sincronizzazione.



Meccanismi di sincronizzazione

os161

-  **Lock:** Permette l'accesso in mutua esclusione ad una regione protetta, senza l'utilizzo di busy waiting.
-  **Spinlock:** Permette l'accesso in mutua esclusione ad una risorsa ma con busy waiting, ovvero consumando cicli di CPU durante l'attesa.
-  **Semaphores:** implementano un contatore per gestire il numero di thread/processi che fanno accesso alla regione protetta, permettono l'ingresso di più entità contemporaneamente.

Meccanismi di sincronizzazione

os161

-  **Condition variable:** permettono l'accesso a un'area protetta solo dopo il verificarsi di un'azione eseguita da un'altra entità. Sono spesso utilizzate in combinazione con locks per implementare la sincronizzazione condizionale. Non fanno busy waiting.
-  **Wait channel:** spesso utilizzati insieme a locks e condition variable, rappresentano un punto di sincronizzazione in cui i thread/processi possono aspettare. Quando un thread attende su un wait channel, rilascia il lock associato, il che consente ad altri thread di accedere alla risorsa, e si mette in uno stato di attesa finché non viene svegliato.

Meccanismi di sincronizzazione

xv6

Anche in xv6 sono implementati diversi meccanismi di sincronizzazione, a differenza di os161 sono pensati solo per processi in quanto xv6 non supporta il multi threading

- **Spinlock:** garantiscono l'accesso esclusivo a una risorsa condivisa da parte dei processi, effettuano attesa attiva. La struttura dati è definita in "spinlock.h" mentre i metodi per inizializzazione e utilizzo sono implementati in "spinlock.c".



Meccanismi di sincronizzazione

xv6

- **Sleeplock:** permettono di gestire situazioni in cui un processo potrebbe dover attendere più a lungo per l'accesso a una risorsa condivisa. Consentono ai processi in attesa di "addormentarsi", evitando il busy waiting. Non possono essere utilizzati negli "interrupt handler" in quanto lasciano gli interrupts attivati. A seconda delle necessità può convenire utilizzare gli spinlock in quanto riducono l'overhead.



Memoria virtuale

La memoria virtuale permette di estendere lo spazio di indirizzamento dei processi, in modo da indirizzare più della reale dimensione della RAM.

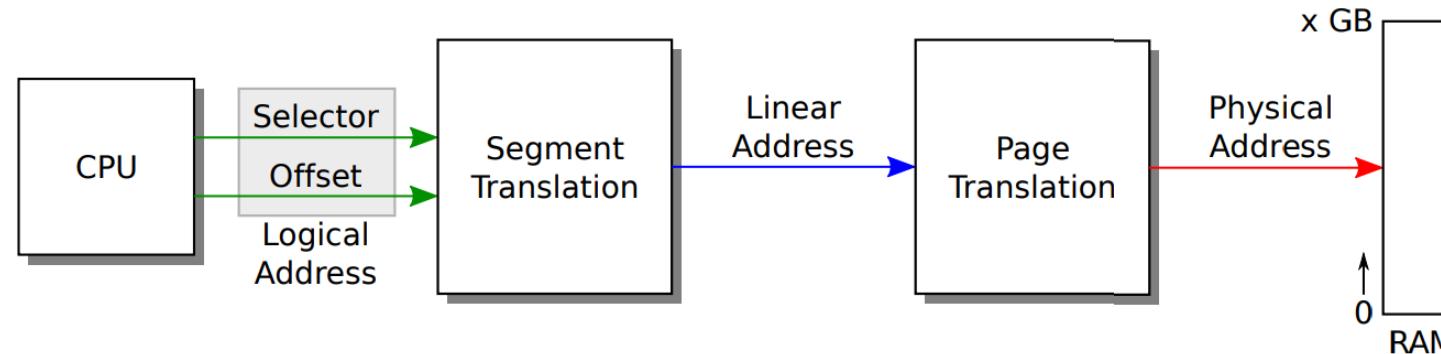
La traduzione degli indirizzi in xv6 è delegata alla MMU (Memory Management Unit), che si occupa di tradurre gli indirizzi virtuali utilizzati dai programmi in indirizzi fisici della memoria principale (RAM).



Traduzione a indirizzo lineare

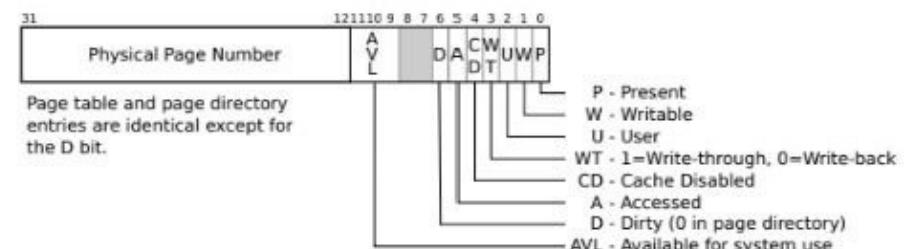
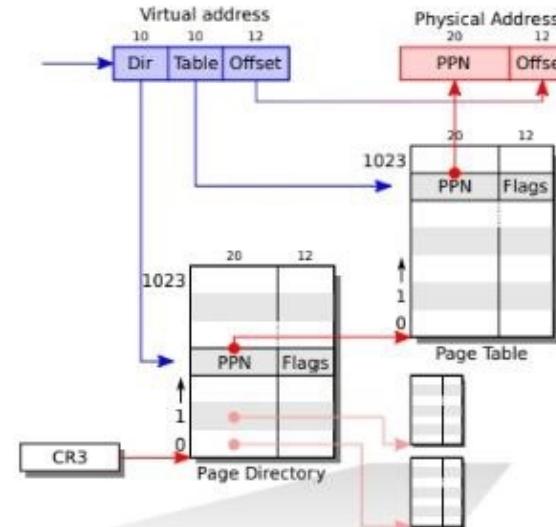
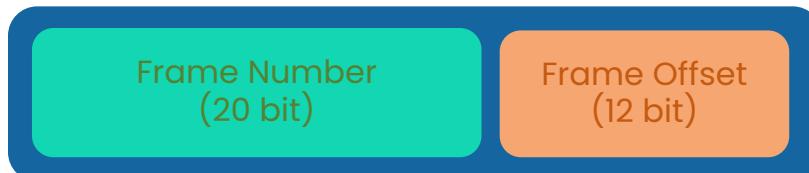
Traduzione dell'indirizzo virtuale

- L'indirizzo virtuale è formato da Segment Selector e Offset
- Si ricava Segment Descriptor dalla DT
- Si aggiunge l'Offset e si ricava l'indirizzo lineare



Traduzione a indirizzo fisico

- Page Table: 2 livelli di gerarchia, 1024 entry, ogni entry 1 bit protezione
 - Indirizzo 20 bit (alti) = 10 (p1) + 10 (p2)
 - Kernel: una page table
 - User: una page table per ogni processo
 - Registro cr3: puntatore alla page table in uso dal processore

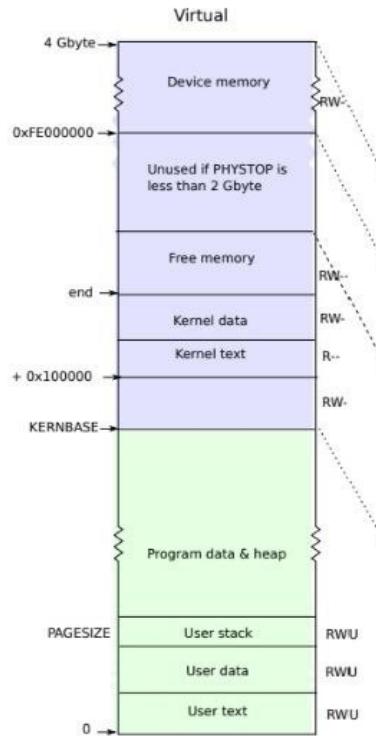


Funzioni per la memoria virtuale

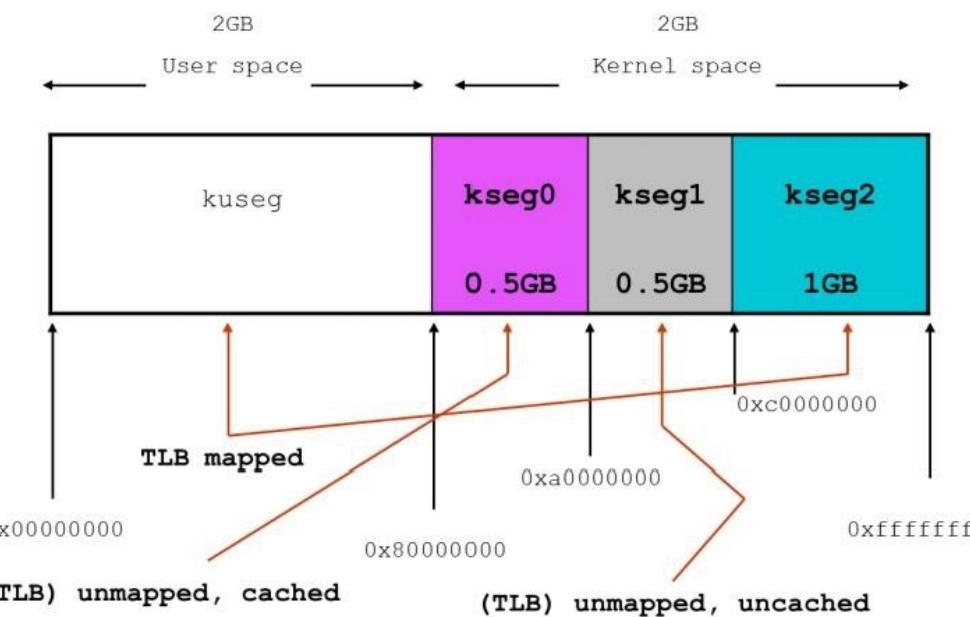
- **kvmalloc()**: alloca la kernel page table
- **freevm(pgd)**: libera le pagine di una page table
- **kalloc()**: alloca interamente una pagina di 4kb
- **kfree(p)**: libera la pagina passata come parametro
- **walkpgdir(pgd, va, alloc)**: attraversa la page table e restituisce un puntatore alla Page Table Entry (PTE) corrispondente all'indirizzo virtuale specificato
- **mappages(pgd, va, size, pa, perm)**: aggiunge entry in una page table

Spazio di indirizzamento virtuale

xv6



os161



xv6 vs os161

Memoria virtuale

- Entrambi dispongono di pagine da 4KB
- os161 supporta Demand Paging, xv6 solamente il Paging
- xv6 implementa la `kalloc()` per allocare una sola pagina
- os161 `ram_stalmem(npages)` alloca `npages` pagine
- xv6 non mette a disposizione `kmalloc` per l'allocazione dinamica della memoria nel kernel
- xv6 non mette a disposizione la memoria condivisa tra processi

xv6 vs os161

Spazio di indirizzamento virtuale

- Limite massimo dello spazio di indirizzamento virtuale di 4 GB.
- Suddivisione secondo il modello di segmentazione classico: segmento codice, segmento dei dati e segmento dedicato allo stack.

Lo spazio di indirizzamento è diviso in:

- User space → porzione inferiore [0x0, KERNBASE]
- Kernel space → porzione superiore [KERNBASE, 0xFFFFFFFF]

Conclusione: Non ci sono particolari differenze fra i due sistemi operativi.

Algoritmi di scheduling

Lo scheduler di xv6 implementa un algoritmo di scheduling a priorità fissa basato su round-robin:

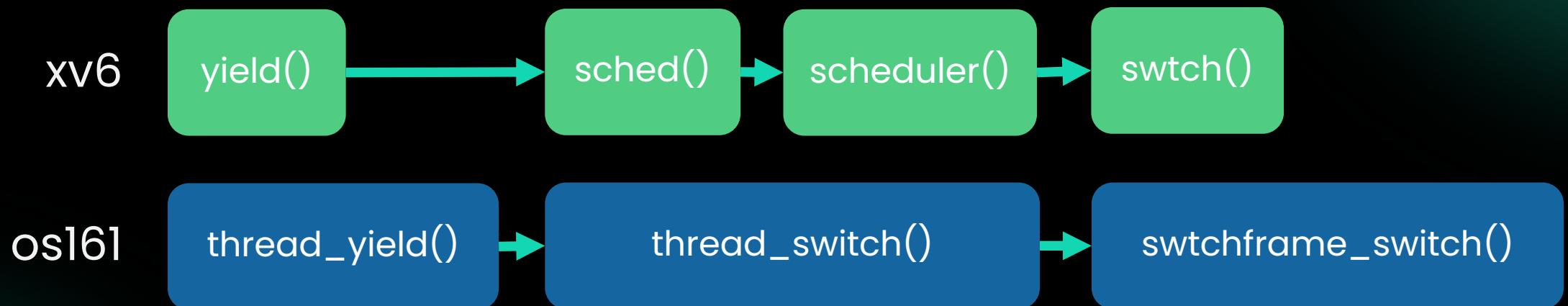
- I processi non hanno un valore di priorità associato, e l'ordine di esecuzione è determinato da un ciclo di round-robin.
- Il tempo di risposta è breve
- Il tempo medio di attesa è lungo

In os161 è implementato il Pre-emptive Round robin:

- Assegna a ciascun processo un quantum di tempo fisso per l'esecuzione sulla CPU
- Approccio che cerca di garantire un trattamento equo tra i processi.

xv6 vs os161

Algoritmi di scheduling



In xv6 lo `scheduler` si occupa dell'ordinamento di processi (in quanto mono-thread) mentre in os161 permette lo scheduling di thread.

Sviluppo delle implementazioni mancanti

Parte 2

Installazione ed esecuzione XV6

E' possibile emulare xv6 nativamente in un ambiente linux con architettura x86, mentre su altri sistemi operativi è necessario usare un ulteriore layer di compatibilità.

- Su distribuzione **linux** x86 è possibile emulare con **qemu**
- Su **Windows** è necessario utilizzare WSL, da cui poi è disponibile **qemu**
- Su **MacOS** con chip arm M1 o superiori è necessario emulare una distribuzione linux x86 mediante UTM e solo dopo avviare l'umulazione tramite **qemu**

Installazione ed esecuzione XV6

Istruzioni per l'installazione:

```
● ● ●  
sudo apt-get install qemu  
  
sudo apt install qemu-system-x86  
  
sudo apt-get install lib6c-dev:i386  
  
git clone https://github.com/mit-pdos/xv6-public.git xv6  
  
chmod 700 -R xv6  
  
cd xv6
```

Istruzioni per l'esecuzione:

```
● ● ●  
make clean  
  
make  
  
make qemu-nox
```

Implementazioni

Ci siamo occupati di implementare le seguenti funzionalità mancanti in xv6

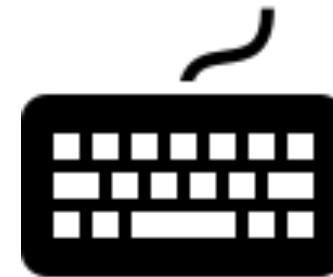
Semaphore



Barrier



Input



Semaphore



Strutture dati

```
// semaphore.h

#define MAX_PROC 32
#define MAX_SEM 32

struct semaphore {
    const char *name;
    int count;
    struct spinlock lock;
    void* queue[MAX_PROC];
    unsigned int tail;
    unsigned int current;
};

struct semaphore semaphores[MAX_SEM];
```

Semaphore



Funzioni

- **sem_create**
inizializza il semaforo con indice s, inizializza il contatore al valore fornito count indicando il numero di processi che possono accedere alla zona critica.
- **sem_wait**
mette il processo chiamante in attesa sul semaforo s.
- **sem_signal**
sveglia tutti i processi in attesa sul semaforo s.

Semaphore



```
●○●
void sem_wait(int s){          // semaphore.c
    struct semaphore* sem = &semaphores[s];

    acquire(&sem->lock);
    sem->queue[sem->tail] = myproc();
    sem->tail = (sem->tail + 1) % MAX_PROC;

    while(sem->count == 0)
        sleep(sem, &sem->lock);
    sem->count = sem->count - 1;
    release(&sem->lock);
}
```

WAIT

```
●○●
void sem_signal(int s){         // semaphore.c
    struct semaphore* sem = &semaphores[s];

    acquire(&sem->lock);
    sem->count = sem->count + 1;
    sem->current = (sem->current + 1)%MAX_PROC;
    wakeup(sem);

    release(&sem->lock);
}
```

SIGNAL

Barrier



Strutture dati

```
// barrier.h

#define MAX_BARRIERS 32

struct barrier {
    uint nproc;
    char* name;
    struct spinlock lock;
};

struct barrier barriers[MAX_BARRIERS];
```

Barrier



Funzioni

- **barrier_create**
inizializza la barrier con indice n, nproc è il numero di processi che devono mettersi in attesa per sbloccarsi.
- **barrier_wait**
mette il processo chiamante in attesa sulla n-esima barrier messa a disposizione dal sistema.

Barrier



```
●●●  
int barrier_create(int n, int nproc, char *name){  
    if (n <= 0 || n >= MAX_BARRIERS){ return -1; }  
    if (nproc <= 0){ return -1; }  
  
    struct barrier *b = &barriers[n];  
    b->nproc = nproc;  
    b->name = name;  
  
    initlock(&b->lock, "barrier");  
    return 0;  
} // barrier.c
```

CREATE

```
●●●  
void barrier_wait(int n){  
    struct barrier* b = &barriers[n];  
  
    acquire(&b->lock);  
    b->nproc--;  
    if (b->nproc == 0){  
        wakeup(b);  
    } else {  
        sleep(b, &b->lock);  
    }  
  
    release(&b->lock);  
    return 0;  
} // barrier.c
```

WAIT

Input



```
int getint(){
    char buffer [256]; // buffer di input utente
    int segno = 1;     // segno del risultato
    int risultato = 0; // risultato finale
    int cont = 0;

    // input dell'utente con la chiamata di sistema
    gets(&buffer, 256);

    // tengo traccia del segno
    if (buffer[0] == '-'){
        segno = -1;
        cont++;
    }

    while(buffer[cont] != '\0'){
        char cifra = buffer[cont] - '0';

        if(cifra < 0 || cifra > 9) break ;

        risultato = risultato*10 + cifra;
        cont = cont + 1;
    }
    risultato = risultato * segno;
    return risultato;
}                                            // input.c
```

Implementazione system call

1. Aggiungere nuova mappatura in **syscall.h**: `#define SYS_name N`
2. In **syscall.c** associare la funzione con la costante e rendere la funzione disponibile dall'esterno:
 - `[SYS_name] sys_name`
 - `Extern int sys_name(void)`
3. Implementare la funzione definita in **sysproc.c**
4. Se necessario, completare la funzione in **proc.c**
5. Aggiungere in **usys.s** la voce `SYSCALL(name)`
6. Compilare il kernel per rendere il comando disponibile da shell

System call implementate

Abbiamo implementato le seguenti system call in modo da poter usare semaphore and barrier dal lato utente:

```
//syscall.h
#define SYS_sem_create 23
#define SYS_sem_wait 24
#define SYS_sem_signal 25

#define SYS_barrier_create 26
#define SYS_barrier_signal 27
```

Testing

Per verificare il corretto funzionamento delle funzionalità implementate, sono stati compiuti i seguenti passaggi:

- Realizzazione di un programma utente in grado di richiamare le system call
- Aggiunta del programma nel makefile, in modo da poterlo eseguire
- Esecuzione del programma di testing con valutazione dei risultati



Test semaphore

- Processo padre inizializza un semaphore tra quelli disponibili e genera n_proc processi figli
- I figli si mettono in attesa sul semaforo, solo proc_count processi entrano nell'area critica
- Quando un processo finisce libera posto ad un processo in attesa
- Il padre aspetta che tutti i figli concludano le loro operazioni

```
*****  
TEST SEMAPHORE  
*****  
  
padre: generato figlio 0  
padre: generato figlio 1  
padre: generato figlio 2  
padre: generato figlio 3  
figlio 0: risvegliato  
figlio 1: risvegliato  
figlio 0: ho rilasciato  
figlio 2: risvegliato  
figlio 1: ho rilasciato  
figlio 3: risvegliato  
figlio 2: ho rilasciato  
figlio 3: ho  
rilasciato  
  
*****  
TEST SEMAPHORE COMPLETED  
*****
```

Test barrier

- Processo padre inizializza una barrier tra quelle disponibili generando `n_proc` processi figli
- I figli svolgono la loro elaborazione e si mettono in attesa sulla barrier
- Quando `n_proc` processi saranno in attesa, vengono svegliati e completano l'elaborazione
- Il padre aspetta che tutti i figli concludano le loro operazioni

```
●○●
*****
TEST BARRIER
*****
padre: generato figlio 0
padre: generato figlio 1
padre: generato figlio 2
padre: generato figlio 3
padre: generato figlio 4
figlio 0: in attesa
figlio 1: in attesa
figlio 2: in attesa
figlio 3: in attesa
figlio 4: in attesa
figlio 4: risvegliato
figlio 0: risvegliato
figlio 1: risvegliato
figlio 2: risvegliato
figlio 3:
risvegliato
*****
TEST SEMAPHORE BARRIER
*****
```

Test input

- Viene chiesto di inserire un numero in input
- Viene chiesto di inserire un secondo numero in input
- Viene effettuata la somma dei due valori interi
- Viene inserito un carattere in ingresso

```
*****  
TEST INPUT  
*****  
  
Inserisci un numero: 123  
Inserisci un secondo numero: 400  
La somma dei due numeri vale: 523  
  
Inserisci un carattere: a  
Il carattere inserito e': a  
  
*****  
TEST INPUT COMPLETED  
*****
```

Fine della presentazione

Grazie per l'attenzione