

ANALISI COMPARATIVA TRA I SISTEMI OSIGI E XVG

Mafrici Federico, Pulignano Luana, Tartaglione Salvatore

SISTEMA OPERATIVO XV6

CARATTERISTICHE PRINCIPALI

xv6 è un sistema operativo educativo sviluppato dal MIT. Esso è stato creato per apprendere le caratteristiche fondamentali che compongono un sistema operativo.

È fortemente influenzato da Unix in quanto implementa molti dei suoi concetti chiave, visibili nella gestione dei file, dei processi e della memoria virtuale.

Il kernel è scritto in C, il che lo rende facile da comprendere e da modificare. Il codice sorgente infatti è compatto e ben strutturato in modo tale da poter estendere il sistema aggiungendo nuove funzionalità.

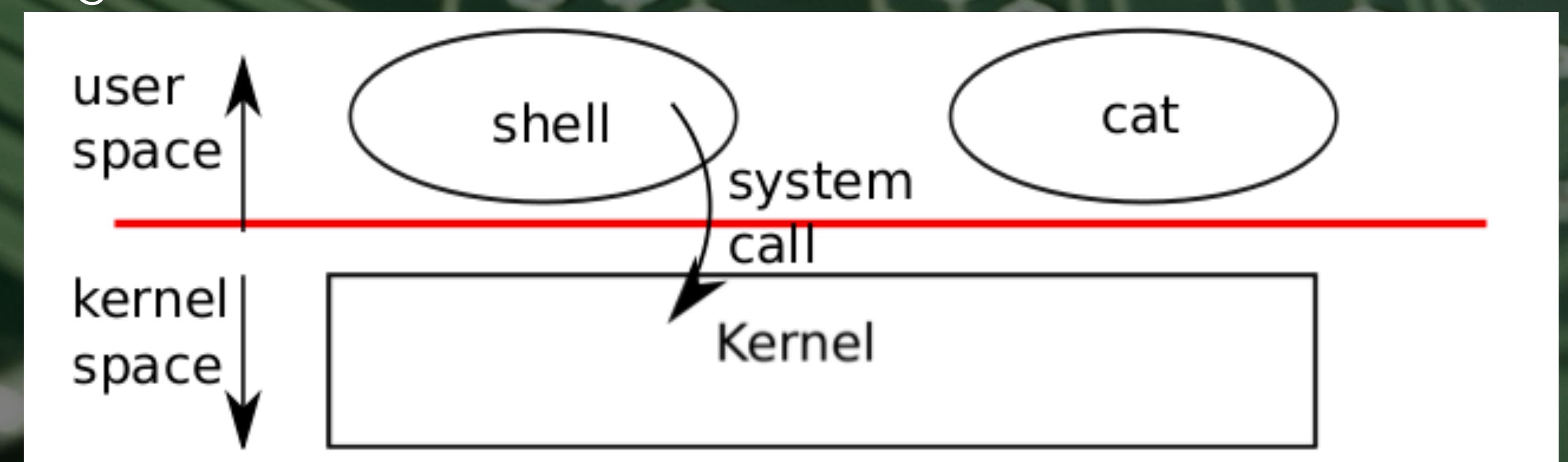
Fornisce i servizi di:

- Gestione concorrente dei processi, anche su sistemi multiprocessore
- Primitive di sincronizzazione (lock, sleep)
- Memoria virtuale

XV6: MODI DI ESECUZIONE

Per garantire un forte isolamento tra le applicazioni e il sistema operativo, xv6 utilizza tre diversi modi di esecuzione in cui poter eseguire le istruzioni:

- Machine mode: si hanno pieni privilegi per l'esecuzione ed è il modo in cui viene avviata la CPU. È utilizzato inoltre per la configurazione del SO, ma solo poche istruzioni vengono eseguite in questa modalità.
- Supervisor mode: è la modalità in cui opera il kernel per poter eseguire le istruzioni privilegiate come la disattivazione degli interrupt. Quando un'applicazione vuole eseguire un'istruzione privilegiata il kernel fa il passaggio da user mode a kernel mode per poterla attuare.
- User mode: è la modalità con cui possono eseguire le applicazioni esterne al sistema operativo e concede pochi privilegi.



PROCESSI

L'unità di esecuzione principale di xv6 è il processo. Per rafforzare l'isolamento tra processi, ad ognuno di essi è associato un proprio address space, inaccessibile da altri processi. Per fare ciò, ogni processo mantiene una sua page table. Ogni processo è mappato dalla struct *proc* che mantiene tutte le info relative. I processi in xv6 non hanno una priorità assegnata. Un processo può modificare la dimensione dell'area di memoria dinamica che gli è stata assegnata. Per fare ciò viene utilizzata la system call *sbrk*, usata sia per aumentare che per ridurre l'heap del processo.

```
// Per-process state
struct proc {
    uint sz;                                // Size of process memory (bytes)
    pde_t* pgdir;                            // Page table
    char *kstack;                            // Bottom of kernel stack for this process
    enum procstate state;                  // Process state
    int pid;                                 // Process ID
    struct proc *parent;                   // Parent process
    struct trapframe *tf;                   // Trap frame for current syscall
    struct context *context;                // swtch() here to run process
    void *chan;                             // If non-zero, sleeping on chan
    int killed;                            // If non-zero, have been killed
    struct file *ofile[NFILE];             // Open files
    struct inode *cwd;                     // Current directory
    char name[16];                          // Process name (debugging)
};

//
```

proc.h

PROCESSI

L'esecuzione di un processo utilizza due importanti system call:

- La system call *exec*, che inizializza la parte utente di un address space, comprendente il page table e lo stack. È utilizzata quando vogliamo cambiare il programma in esecuzione all'interno di uno stesso processo. La memoria allocata è dimensionata secondo l'occupazione del nuovo eseguibile.
- La system call *fork*, usata invece per la programmazione concorrente. Il suo compito è quello di creare un processo figlio duplicando lo spazio di memoria del processo padre. Essa organizza l'esecuzione dei processi basandosi sul PID associato, come illustrato di seguito:

```
int pid = fork();
if (pid == 0) {
    // Codice eseguito dal processo figlio
} else if (pid > 0) {
    // Codice eseguito dal processo genitore
} else {
    // Errore nella creazione del processo
}
```

STATI DI UN PROCESSO

In xv6 un processo può trovarsi in uno dei seguenti stati:

- UNUSED: processo attualmente non in uso.
- EMBRYO: processo in fase di creazione.
- SLEEPING: processo in attesa di un certo evento o di un intervallo di tempo specificato.
- RUNNABLE: processo pronto per essere eseguito.
- RUNNING: processo in esecuzione.
- ZOMBIE: processo figlio terminato ma che deve essere ancora liberato dal processo padre.
- RUNNINGLEAVE: stato introdotto in xv6 da alcuni sviluppatori per gestire le interruzioni in modo più efficace. È una variante di "RUNNABLE" che segnala che un processo può essere ripianificato dopo aver lasciato l'area critica del kernel.

XV6 VS OS161

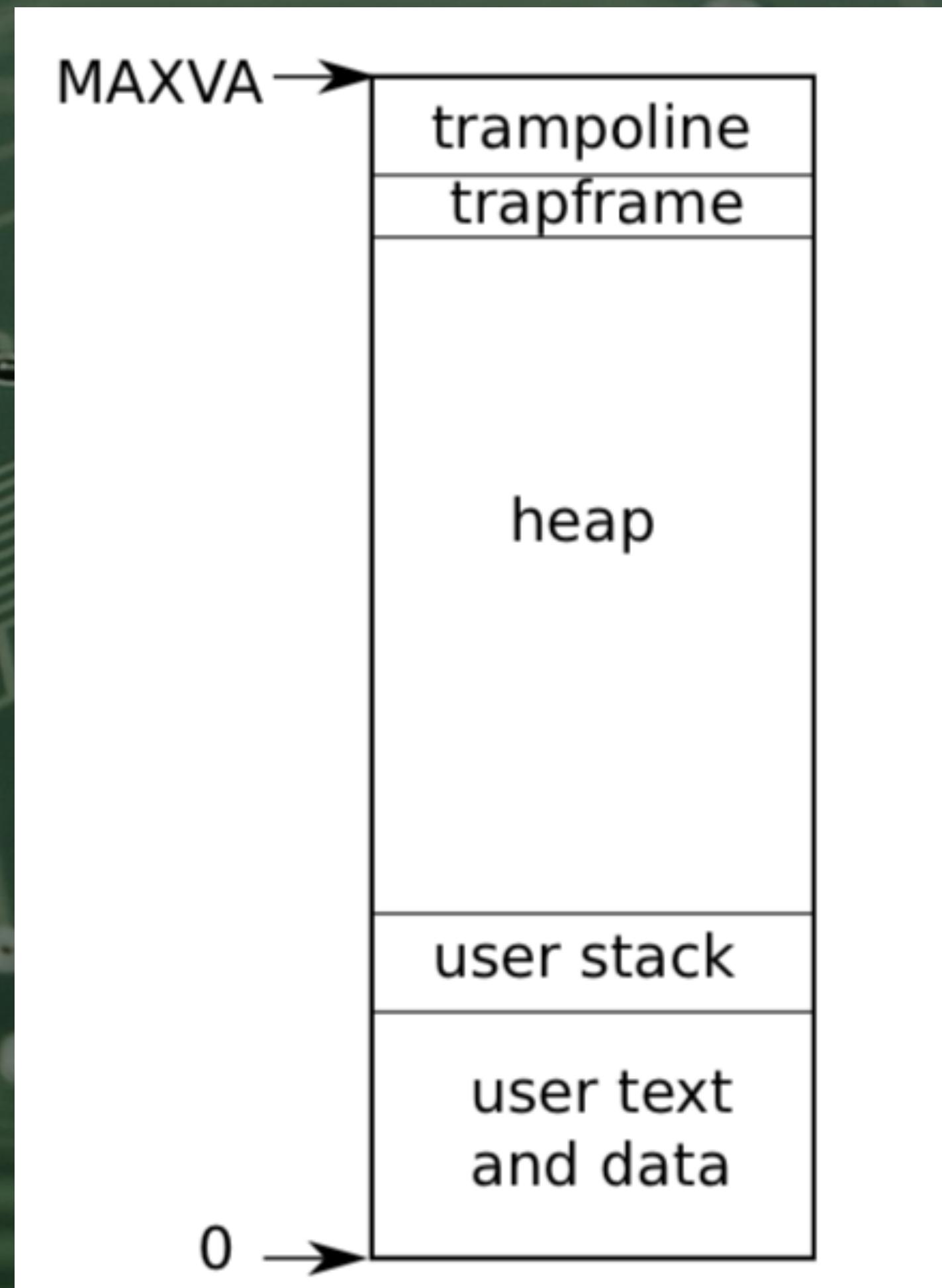
PROCESSI

- *Analogie:* La struttura del processo in entrambi i sistemi operativi è caratterizzata dalla working directory di riferimento, da un id e da una file table. Anche i processi di os161 non sono dotati di priorità, inoltre le chiamate di sistema per l'esecuzione di un programma anche in questo caso sono la execv e la fork con la stessa semantica di xv6.
- *Differenze:* In os161 come unità base d'esecuzione viene utilizzato il thread che appartiene ad un processo. I processi in xv6 mancano del campo numthread poiché un processo non può eseguire contemporaneamente più thread. Ogni processo in os161 ha solo 3 possibili stati: RUNNING, SLEEPING, READY. I primi due stati mantengono la stessa semantica di xv6 mentre lo stato READY equivale allo stato RUNNABLE di xv6.

TRAP

Una trap è un'eccezione di tipo software che si verifica quando l'applicazione utente esegue una system call, causando il passaggio da user mode a kernel mode, o a causa di un errore. In particolare, in xv6, le trap sono gestite dalla struttura dati *IDT* (Interrupt description table), un vettore in grado di instradare le routine da richiamare in caso di un'interruzione.

Quando avviene un'interruzione, è necessario salvare lo stato del processo in esecuzione in modo da essere ripristinato correttamente al ritorno dall'esecuzione della routine. Per fare ciò viene utilizzato il trapframe, una zona di memoria dove viene copiato il contenuto dei registri e che contiene un puntatore allo stack kernel riferito a tale processo. La routine per la gestione delle interruzioni è salvata nel file *trap.c*.



SYSTEM CALL

Le system call in xv6 sono implementate sia in codice assembler che in codice C. Viene utilizzata una *syscall table* per mappare ciascuna chiamata di sistema a una funzione specifica nel kernel.

Quando viene effettuata una system call, il kernel utilizza il numero della system call per individuare la funzione corrispondente nella tabella. Questo numero viene passato alla chiamata 'int' (interrupt), utilizzata ogni volta che si genera una system call. Il passaggio avviene tramite il registro eax in cui viene scritto tale numero. Tale chiamata si occupa di trasferire il controllo al kernel, il quale utlizzerà il parametro passato per discriminare la system call.

Nel file *syscall.c* è contenuta l'implementazione del dispatcher delle chiamate di sistema (*syscall()*) mentre nel file *sysproc.c* sono implementate le funzioni specifiche di ciascuna chiamata di sistema.

```
case T_SYSCALL:  
    syscall(); // Routine di gestione delle chiamate di sistema  
    break;
```

XV6 VS OS161

TRAP E SYSTEM CALL

- *Analogie:* Il meccanismo di chiamata delle system call è uguale sia in xv6 che in os161. Si utilizza sempre una struttura dati trapframe per conservare i valori necessari allo svolgimento della system call.
- *Differenze:* os161 ha un insieme di system call più vario e quindi maggiori in numero rispetto a quelle offerte da xv6.

XV6: MECCANISMI DI SINCRONIZZAZIONE

Nella sua versione nativa xv6 non è multi-thread, quindi due programmi utente non possono accedere allo stesso *userspace* contemporaneamente, ma due processi in kernel mode possono interferire se tentano di far accesso alle stesse strutture del kernel. Xv6 implementa così due semplici meccanismi di sincronizzazione:

- *spinlock*: meccanismo che permette l'accesso in mutua esclusione alla sezione critica.

```
// Mutual exclusion lock.
struct spinlock {
    uint locked;          // Is the lock held?

    // For debugging:
    char *name;           // Name of lock.
    struct cpu *cpu;      // The cpu holding the lock.
    uint pcs[10];         // The call stack (an array of program counters)
};
```

Spinlock.h

- *sleep* e *wakeup*: meccanismi usati per la sincronizzazione fra processi ,consentendo ad un processo di attendere fino a che una determinata condizione è soddisfatta o di essere svegliato quando la condizione è soddisfatta da un altro processo.

XV6: SPINLOCK

- **Acquisire un lock:** si usano le istruzioni atomiche test and set : se locked = 0 il lock è libero e può essere acquisito, se locked = 1 il lock è stato già acquisito e il processo entrerà in un busy while loop affinchè non venga rilasciato il lock. Si disabilitano le interrupt sul core in cui il processo sta chiedendo il lock per evitare che un altro processo possa interrompere il processo attuale in esecuzione (per via dello scheduler) e richiedere lo stesso lock.
- **Rilascio di un lock:** pone la variabile locked a 0.

```
// Release the lock.
void
release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    __asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

spinlock.c

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

spinlock.c

XV6: SLEEP AND WAKEUP

- **sleep:** la sleep richiede un canale nella quale il processo che chiama la sleep dorme, ed uno spinlock. Bisogna acquisire il lock della ptable per permettere di cambiare i campi *chan* e *state* della struct proc in mutua esclusione.

Successivamente si chiama la funzione **sched()** per permettere al processo di essere schedulato dalla CPU. Quando il processo si risveglia rilascia il lock della ptable e riacquisisce lo spinlock.

- **wakeup:** tale funzione chiama **wakeup1** che è protetta da un ptable.lock, quest'ultima risveglia tutti i processi che sono in sleep sul canale chan.

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = myproc();
    if(p == 0)
        panic("sleep");
    if(lk == 0)
        panic("sleep without lk");

    // Must acquire ptable.lock in order to
    // change p->state and then call sched.
    // Once we hold ptable.lock, we can be
    // guaranteed that we won't miss any wakeup
    // (wakeup runs with ptable.lock locked),
    // so it's okay to release lk.
    if(lk != &ptable.lock){ //DOC: sleeplock0
        acquire(&ptable.lock); //DOC: sleeplock1
        release(lk);
    }
    // Go to sleep.
    p->chan = chan;
    p->state = SLEEPING;

    sched();

    // Tidy up.
    p->chan = 0;

    // Reacquire original lock.
    if(lk != &ptable.lock){ //DOC: sleeplock2
        release(&ptable.lock);
        acquire(lk);
    }
}
```

```
static void
wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
```

proc.c

XV6 VS OS161

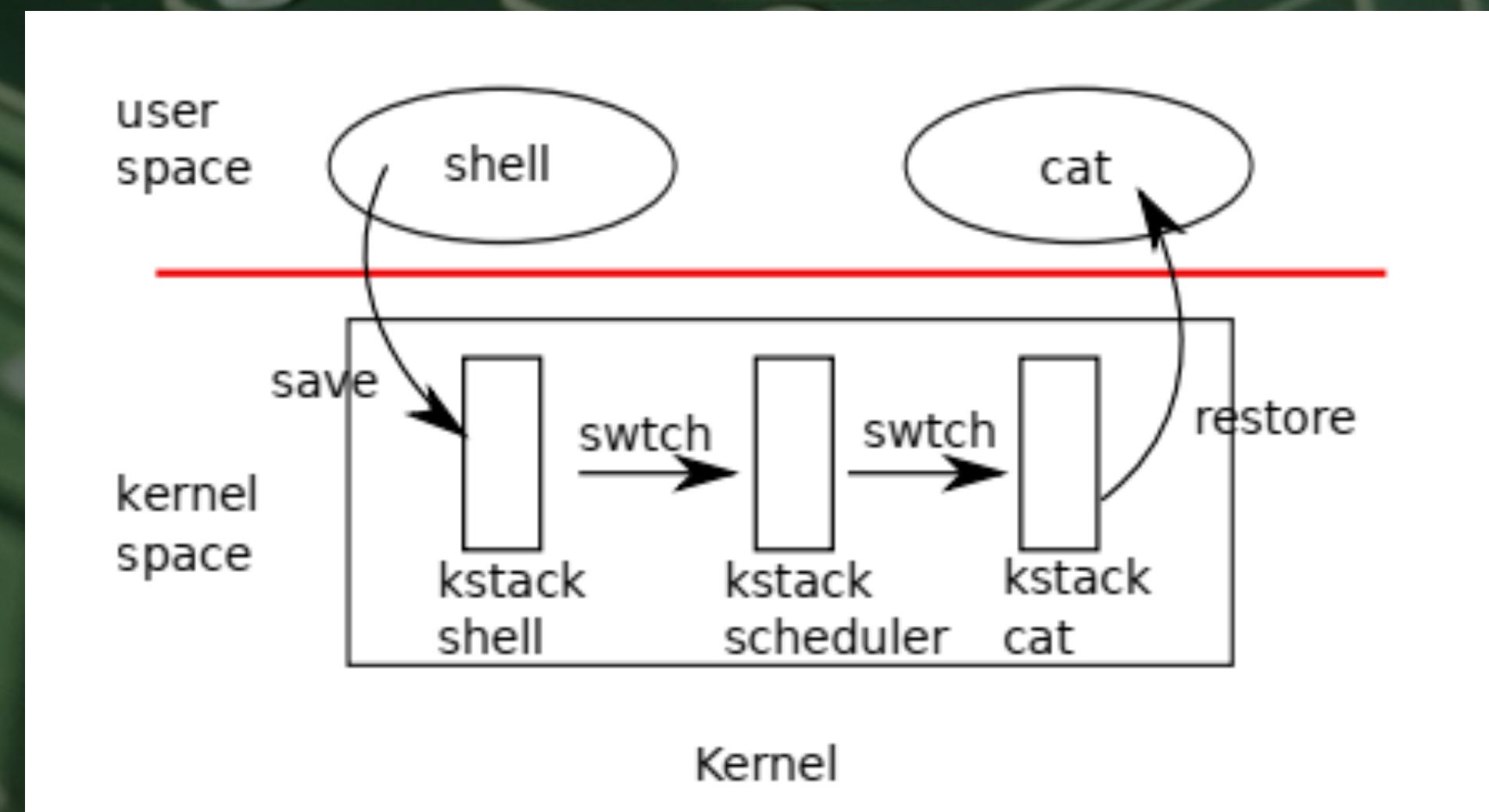
MECCANISMI DI SINCRONIZZAZIONE

- Analogie: In entrambi gli OS viene fatto uso degli *spinlock* per garantire accesso in mutua esclusione a risorse condivise. Le funzioni di *sleep* e *wakeup* sono presenti in entrambi i sistemi.
- Differenze: Poiché i processi supportano un solo thread, le variabili che caratterizzano i processi di sincronizzazione come semafori, *mutex* e *condition variabile* sono assenti in xv6. Di conseguenza, la sincronizzazione fra thread di OS161 è sostituita con una più semplice sincronizzazione fra processi.

SCHEDULING

xv6 utilizza uno scheduler di base, noto come scheduler *round robin*. In questo tipo di scheduling, ogni processo riceve una quantità fissa di tempo di CPU chiamata "quantum". Quando il quantum termina, il controllo passa al prossimo processo nello stato "RUNNABLE" (in attesa di esecuzione).

Lo scheduler di base è implementato nel file *proc.c* nella funzione *scheduler()*.



CONTEXT SWITCH

Il context switch avviene quando lo scheduler deve sostituire il processo in esecuzione con uno pronto per l'esecuzione. In xv6 il processo utente passa il controllo allo scheduler tramite la funzione *sched* mentre lo scheduler ritorna l'esecuzione all'utente tramite la funzione *scheduler*. In entrambi i casi viene utilizzata la funzione *swtch* per il cambio di contesto.

Il context switch avviene in tre casi:

- Quando il processo ha terminato
- Quando il processo deve fare una sleep
- Quando viene invocata la yield dopo l'esecuzione

CONTEXT SWITCH

Nel sistema operativo xv6, ci sono due funzioni principali coinvolte nella gestione della pianificazione dei processi: *sched* e *scheduler*. La funzione *sched* è una funzione chiamata da un processo per rilasciare volontariamente la CPU mentre la funzione *scheduler* è la parte principale del codice di *scheduling* del sistema operativo. È responsabile della scelta del prossimo processo da eseguire.

```
void
sched(void)
{
    int intena;
    struct proc *p = myproc();

    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()>ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

CONTEXT SWITCH

La struttura dati “context” è utilizzata in tale contesto per salvare lo stato dei registri ed è mantenuta da ogni processo all’interno della struct proc. La funzione context_init(struct context *c) è utilizzata per inizializzare tale struttura con valori predefiniti quando si crea un nuovo processo o si cambia il contesto di un processo esistente.

```
struct context {
    // Registri generali
    int edi;
    int esi;
    int ebx;
    int ebp;
    int eip; // Instruction pointer (indirizzo di istruzione)
};
```

XV6 VS OS161

SCHEDULING E CONTEXT SWITCHING

- **Analogie:** Entrambi utilizzano uno scheduling round robin a priorità fissa salvando poi le variabili essenziali all'esecuzione del processo all'interno della struttura context. Anche la gestione del context switching si mantiene identica.
- **Differenze:** La struttura dati in cui viene salvato il contesto in xv6 appartiene al processo mentre in os161, essendo il thread l'unità base di esecuzione, appartiene a quest'ultimo.

XV6: MEMORIA VIRTUALE

- xv6 è un OS a 32bit, quindi lo spazio di indirizzamento virtuale per ogni processo è di 4GB
- indirizzamento virtuale paginato , con pagine di 4KB
- page table gerarchica a 2 livelli
- lo spazio di indirizzamento del processo ha: negl' indirizzi virtuali bassi(\leq 2GB) l' immagine di memoria utente, negl' indirizzi virtuali alti il codice/dati del kernel
- la memoria riservata al kernel ha parte dell' address space per codice/dati dell' OS e della memoria riservata ai device di I/O
- Il mapping tra memoria fisica e memoria virtuale è svolto dall'MMU

XV6: MEMORIA VIRTUALE

ALLOCAZIONE E RILASCIO

- Il sistema operativo utilizza una *Free List* di pagine di memoria libera. I processi, quando necessitano di memoria aggiuntiva, chiamando la *kalloc()* ottengono una pagina vuota ed il puntatore della lista si aggiorna all'elemento successivo.
- Per rilasciare le pagine viene utilizzata invece la funzione *kfree()* la quale aggiunge in testa alla *Free List* la pagina liberata .

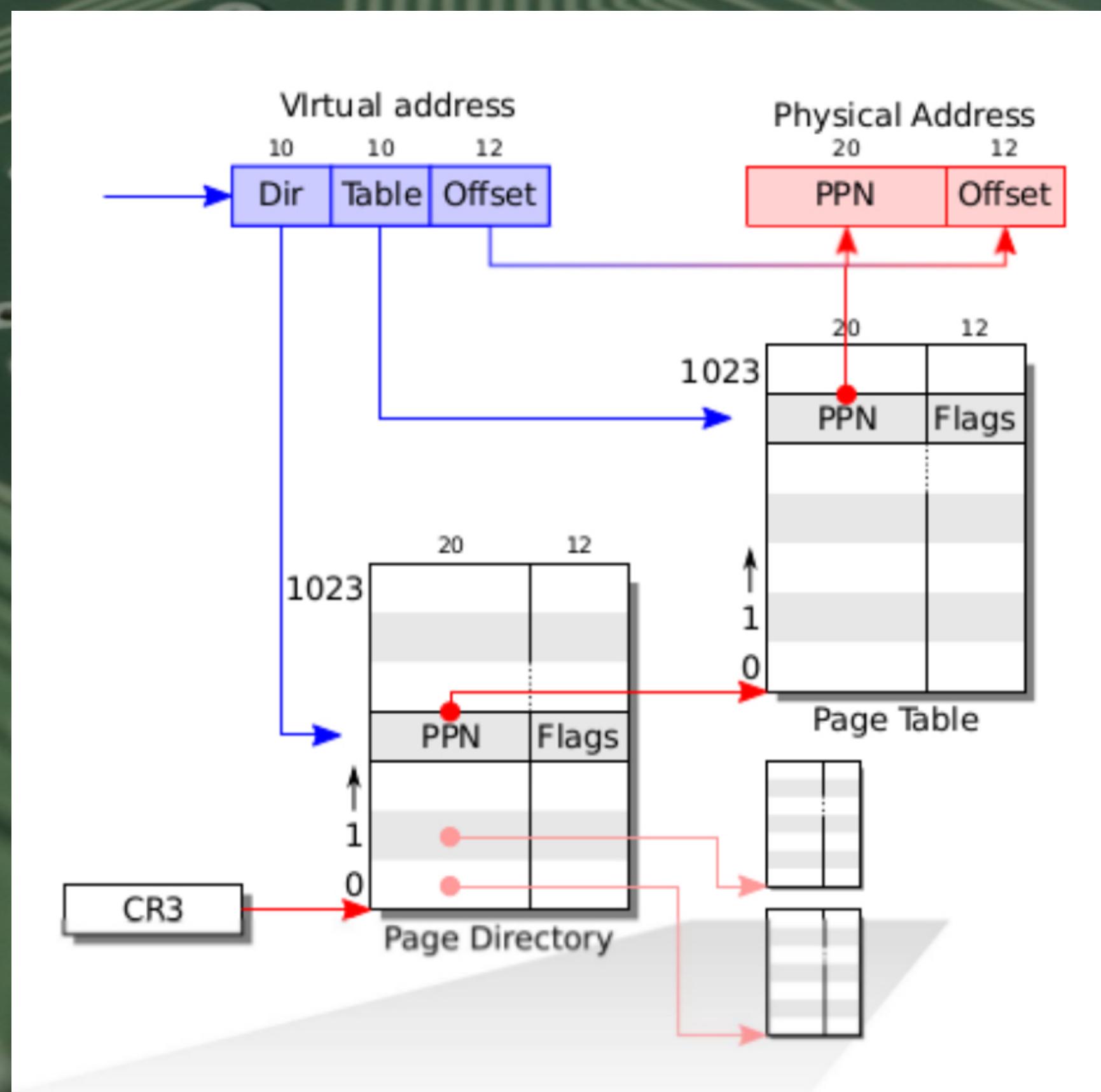
```
void  
kfree(char *v)  
{  
    struct run *r;  
  
    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)  
        panic("kfree");  
  
    // Fill with junk to catch dangling refs.  
    memset(v, 1, PGSIZE);  
  
    if(kmem.use_lock)  
        acquire(&kmem.lock);  
    r = (struct run*)v;  
    r->next = kmem.freelist;  
    kmem.freelist = r;  
    if(kmem.use_lock)  
        release(&kmem.lock);  
}
```

```
char*  
kalloc(void)  
{  
    struct run *r;  
  
    if(kmem.use_lock)  
        acquire(&kmem.lock);  
    r = kmem.freelist;  
    if(r)  
        kmem.freelist = r->next;  
    if(kmem.use_lock)  
        release(&kmem.lock);  
    return (char*)r;  
}
```

XV6: PAGE TABLE

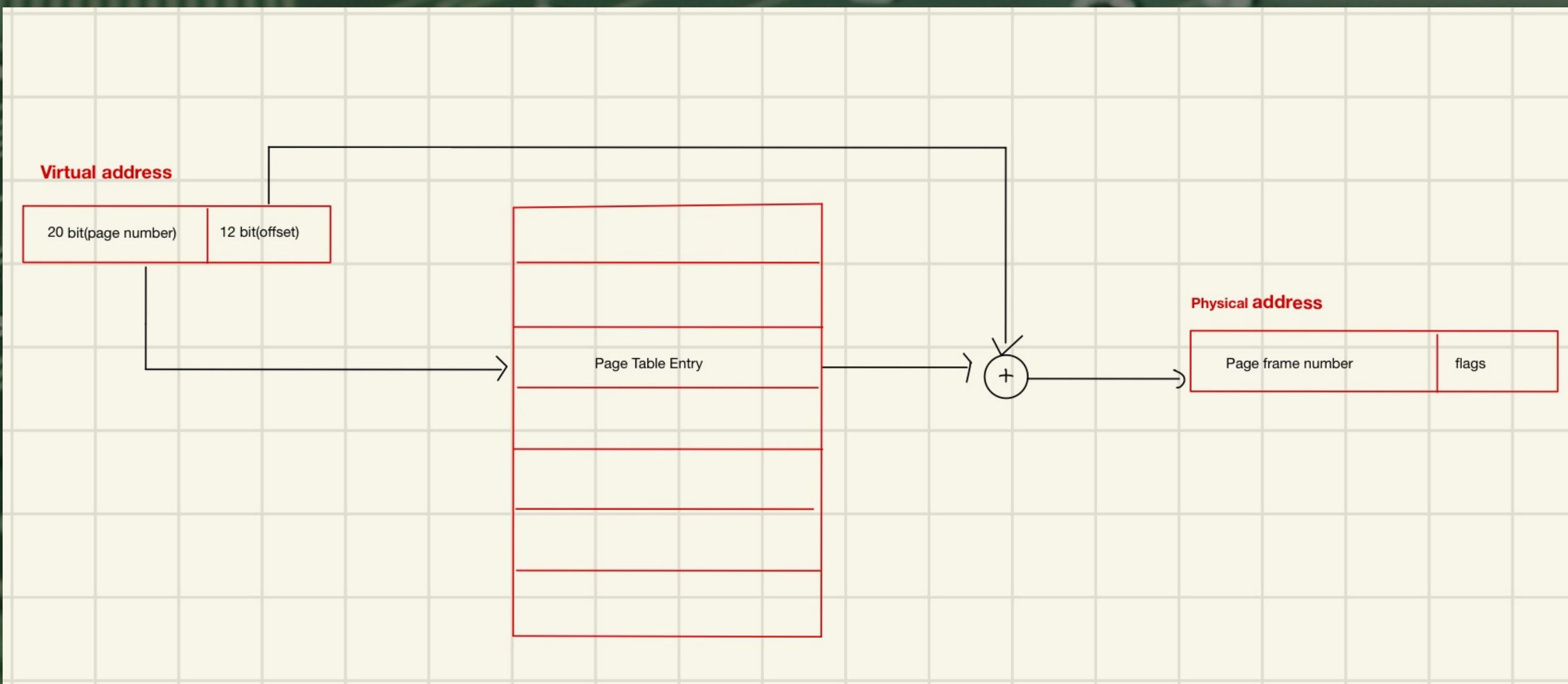
La page table di xv6 ha due livelli, il virtual address di 32 bit è così organizzato:

- 10 bits index nella outer page table (chiamata Page Directory Index)
- 10 bits per la inner page table
- 12 bits di offset



XV6: PAGE TABLE

- Spazio di indirizzamento virtuale= 2^{32} bytes
- Grandezza pagina = 4KB = 2^{12} bytes, quindi 12 bit di offset
- La page table è un array di 2^{20} page table entries (**PTE**), ogni **PTE** ha 20 bit per indicizzare il frame fisico e alcuni flags:
- PTE_P: pagina presente (se non settata, scatena un page fault)
- PTE_W:pagina scrivibile (se non settata, la pagina è accessibile in sola lettura)
- PTE_U:indicante che è una pagina utente(se non settata, solo il kernel può accedervi)



XV6 VS OS161

MEMORIA VIRTUALE E PAGE TABLE

- Analogie: Lo spazio di indirizzamento virtuale è della stessa dimensione (4GB) così come quella delle pagine(4KB).
- Differenze: in os161 la gestione della memoria libera è fatta tramite un vettore nominato FreeRamFrames necessario per effettuare il mapping tra indirizzo fisico e virtuale. In aggiunta si utilizza un vettore allocSize che indica l'offset delle pagine occupate. In xv6 c'è invece una freeList di pagine libere.

freeRamFrames	
0	0
0	0
0	0
0	2
0	0
1	0
1	0
1	0
0	1
1	0
0	1

allocSize

CONFRONTO TRA XV6 E OS16I

Caratteristiche	xv6	Os16I
Linguaggi	C, Assembly	C, Assembly
Architettura	Kernel Monolitico	Macro Kernel
Scheduler	Round Robin	Round Robin
Inter process communication	Lock	Semafori
Virtual memory	Si	Si
Task States	Running, Unused, Runnable, Embryo, Sleeping, Zombie, Runningleave	Running, Waiting, Sleep
File system	Si	Si

INSTALLAZIONE DI XV6

- Per l'installazione di xv6 è necessario scaricare un sottosistema Linux per Windows noto come WSL fare riferimento alla seguente guida: <https://learn.microsoft.com/it-it/windows/wsl/install>
 - È anche necessario installare qemu e ciò si può fare direttamente da terminale tramite i comandi:
 1. sudo apt update
 2. sudo apt upgrade
 3. sudo apt install git build-essential qemu-system
 - Fatto ciò scaricare direttamente dal seguente link il codice sorgente del SO: <https://github.com/mit-pdos/xv6-public>
 - Dopo essersi spostati nel direttorio del sistema operativo, avviarlo una prima compilazione eseguendo i comandi:
 1. make qemu-nox
- Se la compilazione ha successo comparirà una nuova finestra con l'esecuzione di xv6.

```
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00

Booting from Hard Disk...xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
```

IMPLEMENTAZIONE DI UNA SYSTEM CALL

1. -Per prima cosa si deve andare ad inserire la firma della system call all'interno dei file "syscall.h"
2. -Nel file "syscall.c" si aggiunge invece la firma delle funzioni da implementare e poi si aggiunge alla tabella delle system call l'associazione [nome systemcall] => sys_da chiamare.
3. Successivamente si procede ad inserire il prototipo all'interno dei file "defs.h, user.h" (nel nostro caso ad esempio int cps(void))
4. Nel file "sys_proc.c" inserire la routine eseguita dal system call.
5. Inserire poi la funzione che verrà eseguita dalla routine della system call all'interno del file "proc.c"
6. A questo punto in "usys.s" definire la voce della system call implementata.

1 Per prima cosa si deve andare ad inserire la firma della system call all'interno dei file "syscall.h"

2 Nel file "syscall.c" si aggiunge invece la firma delle funzioni da implementare e poi si aggiunge alla tabella delle system call l'associazione [nome systemcall] => sys_da chiamare.

```
109 static int (*syscalls[])(void) = {  
110 [SYS_fork] sys_fork,  
111 [SYS_exit] sys_exit,  
112 [SYS_wait] sys_wait,  
113 [SYS_pipe] sys_pipe,  
114 [SYS_read] sys_read,  
115 [SYS_kill] sys_kill,  
116 [SYS_exec] sys_exec,  
117 [SYS_fstat] sys_fstat,  
118 [SYS_chdir] sys_chdir,  
119 [SYS_dup] sys_dup,  
120 [SYS_getpid] sys_getpid,  
121 [SYS_sbrk] sys_sbrk,  
122 [SYS_sleep] sys_sleep,  
123 [SYS_uptime] sys_uptime,  
124 [SYS_open] sys_open,  
125 [SYS_write] sys_write,  
126 [SYS_mknod] sys_mknod,  
127 [SYS_unlink] sys_unlink,  
128 [SYS_link] sys_link,  
129 [SYS_mkdir] sys_mkdir,  
130 [SYS_close] sys_close,  
131 [SYS_cps] sys_cps,  
132 [SYS_chpr] sys_chpr,  
133 };
```

2.1 syscall.c

```
extern int sys_chdir(void);  
extern int sys_close(void);  
extern int sys_dup(void);  
extern int sys_exec(void);  
extern int sys_exit(void);  
extern int sys_fork(void);  
extern int sys_fstat(void);  
extern int sys_getpid(void);  
extern int sys_kill(void);  
extern int sys_link(void);  
extern int sys_mkdir(void);  
extern int sys_mknod(void);  
extern int sys_open(void);  
extern int sys_pipe(void);  
extern int sys_read(void);  
extern int sys_sbrk(void);  
extern int sys_sleep(void);  
extern int sys_unlink(void);  
extern int sys_wait(void);  
extern int sys_write(void);  
extern int sys_uptime(void);  
extern int sys_cps(void);  
extern int sys_chpr(void);
```

2.2 syscall.c

```
1 // System call numbers  
2 #define SYS_fork 1  
3 #define SYS_exit 2  
4 #define SYS_wait 3  
5 #define SYS_pipe 4  
6 #define SYS_read 5  
7 #define SYS_kill 6  
8 #define SYS_exec 7  
9 #define SYS_fstat 8  
10 #define SYS_chdir 9  
11 #define SYS_dup 10  
12 #define SYS_getpid 11  
13 #define SYS_sbrk 12  
14 #define SYS_sleep 13  
15 #define SYS_uptime 14  
16 #define SYS_open 15  
17 #define SYS_write 16  
18 #define SYS_mknod 17  
19 #define SYS_unlink 18  
20 #define SYS_link 19  
21 #define SYS_mkdir 20  
22 #define SYS_close 21  
23 #define SYS_cps 22  
24 #define SYS_chpr 23
```

1 syscall.h

3. Successivamente si procede ad inserire il prototipo all'interno dei file "defs.h, user.h"

4. Nel file "sys_proc.c" inserire la routine eseguita dal system call.

```
105 // proc.c
106 int cpuid(void);
107 void exit(void);
108 int fork(void);
109 int growproc(int);
110 int kill(int);
111 struct cpu* mycpu(void);
112 struct proc* myproc();
113 void pinit(void);
114 void procdump(void);
115 void scheduler(void) __attribute__((noreturn));
116 void sched(void);
117 void setproc(struct proc*);
118 void sleep(void*, struct spinlock*);
119 void userinit(void);
120 int wait(void);
121 void wakeup(void*);
122 void yield(void);
123 int cps(void);
124 int chpr(int pid,int priority);
125
126
127
```

3 defs.h

```
4 // system calls
5 int fork(void);
6 int exit(void) __attribute__((noreturn));
7 int wait(void);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
17 int fstat(int fd, struct stat*);
18 int link(const char*, const char*);
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int cps(void);
27 int chpr(int,int);
28
29
30
31
32
33
```

3 user.h

```
93 int sys(cps(void){
94     return cps();
95 }
96
97 int sys_chpr(void){
98     int pid,pr;
99     if(argint(0,&pid)<0)
100         return -1;
101     if(argint(1,&pr)<0)
102         return -1;
103
104     return chpr (pid,pr);
105 }
```

4 sys_proc.c

5. Inserire poi la funzione che verrà eseguita dalla routine della system call all'interno del file "proc.c"

6.A questo punto in "usys.s" definire la voce della system call implementata.

```
int cps(){
    struct proc * p;
    sti();
    acquire(&ptable.lock);
    cprintf("nome \t pid \t stato \t \t priorità \n");
    for(p= ptable.proc; p< &ptable.proc[NPROC]; p++){
        if (p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t %d \n ",p->name,p->pid,p->priority);
        else if (p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t %d \n ",p->name,p->pid,p->priority);
        else if (p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t %d \n ",p->name,p->pid,p->priority);
    }
    release(&ptable.lock);
    return 22;
}

int chpr(int pid,int priority){
    struct proc *p;
    acquire(&ptable.lock);
    for(p= ptable.proc; p< &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->priority = priority;
            break;
        }
    }
    release(&ptable.lock);

    return pid;
}
```

5.1 proc.c

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(cps)
SYSCALL(chpr)
```

6.1 usys.s

IMPLEMENTAZIONE TEST SYSTEM CALL

Fatto ciò abbiamo implementato la system call, tuttavia c'è ancora bisogno di realizzare un comando che genera la system call per verificare che questa sia funzionante. Vediamo come fare:

- Si crea un file con nome a piacere e si aggiunge il [nome_file] nel Makefile per permettere la compilazione.

```
EXTRA=\
mkfs.c 
```

```
# that disk image changes after first build are persistent until clean. More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _ps\
    _test\
    _priority\
    _zombie\

fs.img: mkfs README $(UPROGS)
    ./mkfs fs.img README $(UPROGS)

-.include *.d

clean:
    rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
    *.o *.d *.asm *.sym vectors.S bootblock entryother \
    initcode initcode.out kernel xv6.img fs.img kernelmemfs \
    xv6memfs.img mkfs .gdbinit \
    $(UPROGS)

# make a printout
FILES = $(shell grep -v '^#' runoff.list)
PRINT = runoff.list runoff.spec README toc.hdr toc.ftr $(FILES)

TINI = runoff.list runoff.spec README toc.hdr toc.ftr $(FILES)
EIGER = $(shell grep -v '^#', runoff.list)

# make a printout
```

SYSTEM CALL IMPLEMENTATE:

PS E CPS

È necessario osservare che, per inserire il concetto di priorità abbiamo aggiunto alla struct proc una variabile che tiene traccia della priorità e, abbiamo modificato la funzione alloc_proc affinché ogni processo riceva una priorità di default pari a 10. Il meccanismo di priorità è crescente(valore più basso == priorità maggiore)

Andiamo ora a vedere i servizi offerti dalle due system call:

PS : La ps stampa gli attuali processi in esecuzione indicandone il nome, il pid, lo stato e la priorità.

CPSH: permette di modificare la priorità del processo identificato dal pid fornito con il valore desiderato.

```
37 // Per-process state
38 struct proc {
39     uint sz;                                // Size of process memory (bytes)
40     pde_t* pgdir;                           // Page table
41     char *kstack;                            // Bottom of kernel stack for this process
42     enum procstate state;                  // Process state
43     int pid;                                // Process ID
44     struct proc *parent;                   // Parent process
45     struct trapframe *tf;                  // Trap frame for current syscall
46     struct context *context;                // swtch() here to run process
47     void *chan;                             // If non-zero, sleeping on chan
48     int killed;                            // If non-zero, have been killed
49     struct file *ofile[NOFILE];            // Open files
50     struct inode *cwd;                     // Current directory
51     char name[16];                          // Process name (debugging)
52     int priority; |                         // Priority
53 };
54
55 };
56
57 }
```

proc.h

FUNZIONAMENTO DI SYSTEM CALL

- Quando da shell invochiamo il comando cps viene eseguita la funzione cps. L'esecuzione genera una trap gestita nel file trap.c.
- All'interno di trap.c la funzione trap verifica se c'è stata una system call oppure un evento di altro genere.
- Nel caso di system call si esegue la funzione syscall in syscall.c e si cerca il SYSCALL_ID.
- Nel nostro caso la syscall sarebbe sys_cps che si trova in sysproc.c (vedere foto)

```
Thread 1 hit Breakpoint 2, trap (tf=0x8dfd6fb4) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:139
139      struct proc *curproc = myproc();
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, trap (tf=0x8dfd6f18) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) c
Continuing.

Thread 1 hit Breakpoint 4, sys_cps () at sysproc.c:94
94      return cps();
(gdb) c
Continuing.
```

IMPLEMENTAZIONE SHUTDOWN

- Per implementare tale system call mancante in xv6, abbiamo seguito gli stessi passi precedenti per l' implementazione della cps e della chpr , con l' unica peculiarità che risulta essere la chiamata outw la quale si interfaccia con l' hardware sottostante per inviare il messaggio di shutdown. In particolare tale codice di shutdown dipende dall' hardware sottostante.Dato che xv6 è stato emulato sulla versione più recente di Qemu, il segnale da inviare all' hardware è specificato nella documentazione.

Emulator-specific methods

In some cases (such as testing), you may want a poweroff method, but not need it to work on real hardware.

In Bochs, and older versions of QEMU(than 2.0), you can do the following:

```
outw(0xB004, 0x2000);
```

In newer versions of QEMU, you can do shutdown with:

```
outw(0x604, 0x2000);
```

Documentazione

```
int sys_shutdown(void){  
    outw(0x604,0x2000);  
    return 0;  
}
```

sysproc.c