



Analisi comparativa tra OS161 e xv6

Jellouli Hamza s308734
Magistro Contenta Ivan s314356
Marinacci Giuseppe s320001



Sezione n.1

Analisi di xv6 e confronto con OS161



Introduzione a xv6

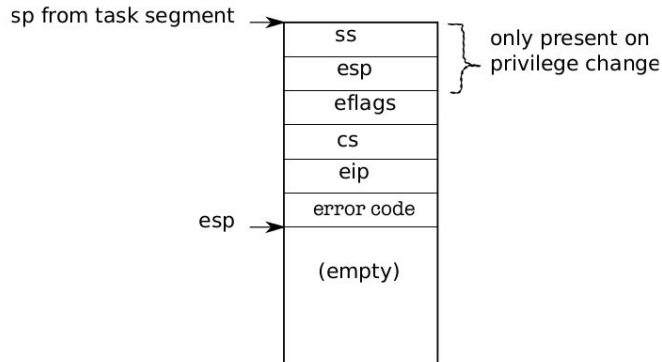
- Sistema operativo basato su UNIX V6.
- Progettato presso il MIT, nel 2006, a scopo didattico.
- Scritto quasi interamente in codice C ed Assembly.
- Architettura considerata: x86 (tipo CISC con processore a 32 bit).

System Calls

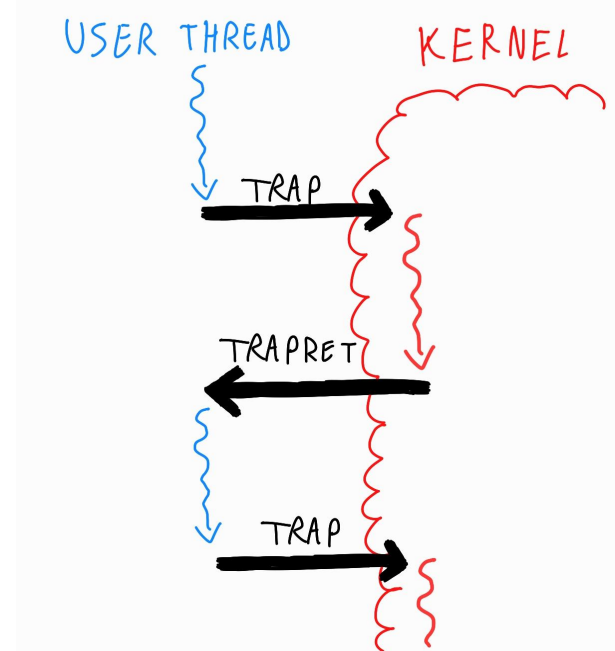
- Servizio offerto dal SO ai programmi utente.
- È necessario scatenare un **interrupt**.
- Per gestirlo, il SO deve implementare il **context switching**:
 - salvataggio di *user stack* e *kernel stack*;
 - passaggio da *user mode* a *kernel mode* (e viceversa);

Trap e context switching

- Per generare una trap si usa il comando **int n** (in *usys.S*) con $n = T_SYSCALL$, l'identificatore dell'*interrupt handler* per le system call (quest'ultimo descritto nella *Interrupt Descriptor Table*);
- **alltraps** (in *trapasm.S*), chiamata da **int**, crea il *trap frame*, poi chiama **trap** (in *trap.c*);
- Si passa da *user mode* a *kernel mode*;
- Si conservano i riferimenti utili per lo switch inverso.



trap frame (nel kernel stack) dopo il context switch

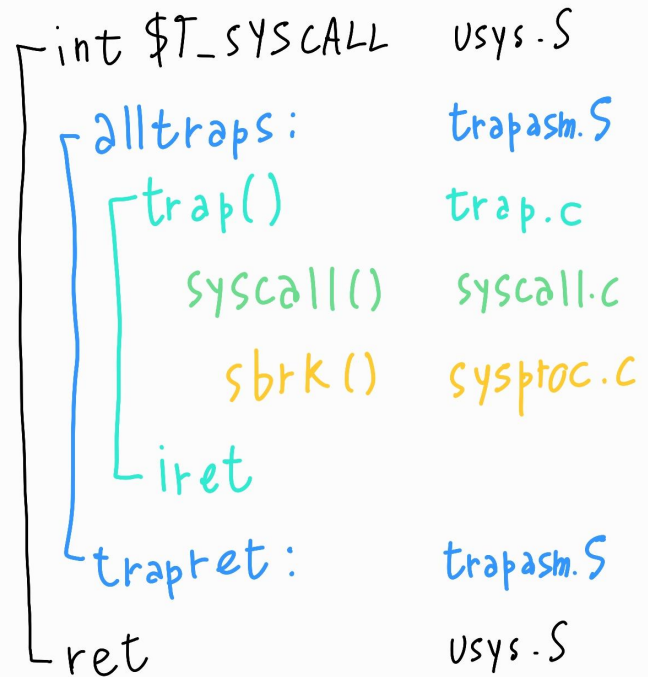


System calls in xv6

- **trap()** determina cosa ha scatenato l'interrupt;
- Nel caso di una *system call*, invoca **syscall()**;
- **syscall()** (in *syscall.c*) esegue il *fetch* dell'*ID* della *system call* specifica (in *%eax*) e la invoca;
- Infine, **trapret** (in *trapasm.S*) esegue il *context switch inverso*;

System calls implementate in xv6
(*syscall.h*)

System call	Description	ID
fork()	Create process	1
exit()	Terminate current process	2
wait()	Wait for a child process to exit	3
pipe(p)	Create a pipe and return fd's in p	4
read(fd, buf, n)	Read n bytes from an open file into <u>buf</u>	5
kill(pid)	Terminate process pid	6
exec(filename, *argv)	Load a file and execute it	7
fstat(fd)	Return info about an open file	8
chdir(dirname)	Change the current directory	9
dup(fd)	Duplicate fd	10
getpid()	Return current process's id	11
sbrk(n)	Grow process's memory by n bytes	12
sleep(n)	Sleep for n seconds	13
uptime()	Retrieve time elapsed since system boot	14
open(filename, flags)	Open a file; flags indicate read/write	15
write(fd, buf, n)	Write n bytes to an open file	16
mknod(name, major, minor)	Create a device file	17
unlink(filename)	Remove a file	18
link(f1, f2)	Create another name (f2) for the file f1	19
mkdir(dirname)	Create a new directory	20
close(fd)	Release open file fd	21



Flusso delle chiamate per la syscall *sbrk()*

Analogie e differenze con OS161

- OS161 gestisce le system calls in maniera analoga a xv6:
 - si scatena una **trap**;
 - avviene il **context switching** (con creazione del *trap frame* nel kernel stack);
 - si esegue la **system call**;
 - si ripristina lo *user stack* e si torna all'esecuzione in **user mode**.
- **Differenze di carattere implementativo** (dovute principalmente all'architettura, MIPS vs x86).

Meccanismi di sincronizzazione

I meccanismi di sincronizzazione sono strumenti essenziali per gestire il coordinamento tra processi/threads in parallelo. Questi meccanismi assicurano l'ordine e l'efficienza dell'accesso alle risorse condivise, prevenendo conflitti e garantendo un'operatività armoniosa. L'obiettivo è evitare problemi come "race condition" e "deadlock" per mantenere la coerenza e la sicurezza del sistema.

Meccanismi di sincronizzazione in OS161

OS161 mette a disposizione molteplici soluzioni per gestire la sincronizzazione che sono:



- **Semaphores:** Permettono l'accesso concorrente ai thread/processi ad un numero limitato di risorse, può essere impostato anche non in mutua esclusione
- **Lock:** Meccanismo che fa mutua esclusione, che possiede il concetto di owner
- **Spinlock:** Meccanismo che fa mutua esclusione, ma facendo busy waiting
- **Condition variable:** Meccanismo di sincronizzazione che permette l'accesso alla risorsa condivisa al verificarsi di una condizione
- **Wait channel:** Meccanismo molto simile alle condition variable, ma è busy waiting e si può utilizzare solo nel kernel

Meccanismi di sincronizzazione in XV6

Anche XV6 mette a disposizione molteplici soluzioni per gestire la sincronizzazione, ma questi sono solo per i processi dato che non viene implementato il multi threading. I meccanismi in questione sono:



Spinlock: Meccanismo che permette l'accesso in mutua esclusione alla regione protetta quando se ne ha possesso. Questo meccanismo fa busy waiting. La definizione della struttura dati si trova in `spinlock.h`, mentre i metodi per l'inizializzazione e implementare la sincronizzazione si trovano in `spinlock.c`

Sleep/Wakeup: Meccanismo di sincronizzazione che mette i processi nello stato di sleeping in attesa che una condizione venga verificata e questa dipende da un altro processo. Questo meccanismo non fa busy waiting e i metodi per utilizzarlo sono definiti in `proc.c`. La chiamata per eseguire la sleep deve necessariamente passare per la funzione `sys_sleep` contenuta nel file `sysproc.c`



Meccanismi di sincronizzazione in XV6

Sleeplock: E' un meccanismo di sincronizzazione ibrido in quanto implementa sia l'attesa attiva, sia quella passiva.

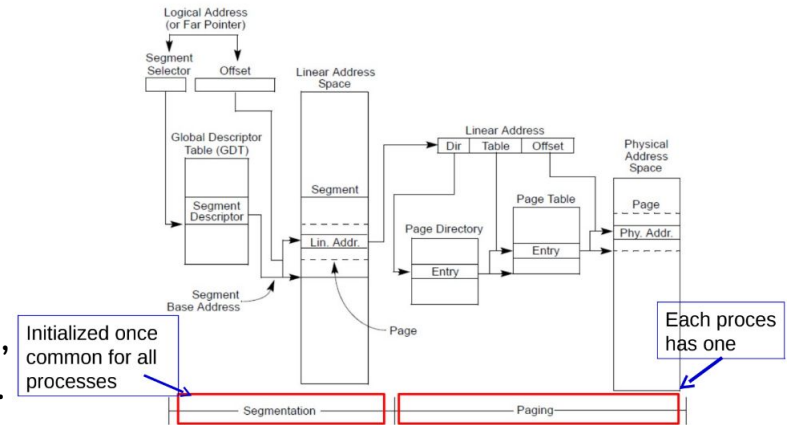
L'attesa attiva è implementata per l'accesso alla struttura dati definita in sleeplock.h, mentre quella passiva è implementata solo nella funzione acquire che insieme alle altre funzioni per la gestione degli sleeplock è presente in sleeplock.c .

L'attesa passiva è utilizzata dopo aver possesso dello spinlock ed è utilizzata nel caso risultasse che lo sleeplock sia già posseduto. L'attesa attiva è implementato mettendo nello stato di sleeping il processo. Quando il processo viene messo nello stato di sleeping continua a possedere lo spinlock della struttura dati, evitando quindi le race condition.



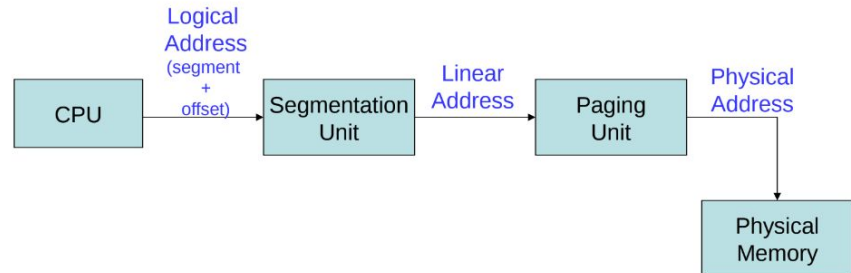
Memoria virtuale

- Gestione della memoria virtuale:
 - Memoria virtuale utile per trasferire dati dalla memoria RAM al disco e viceversa.
 - in xv6
 - Realizzata con la ***paginazione hardware***.
 - Non dispone di *demand paging*, fork COW e *memoria condivisa*.



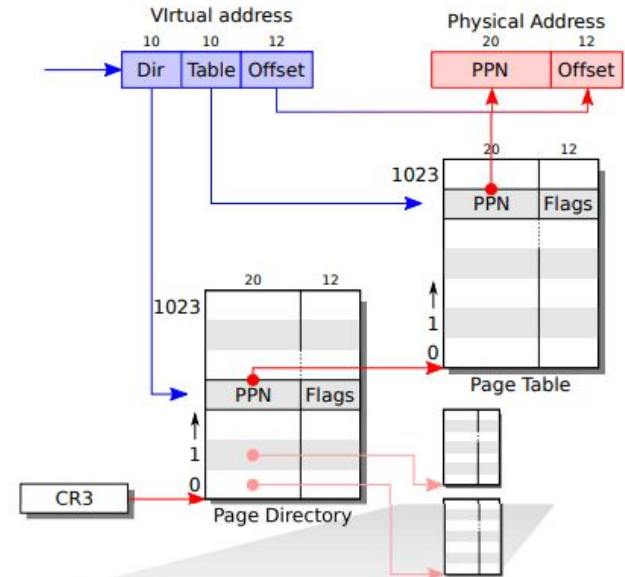
Memoria virtuale

- Traduzione degli indirizzi:
 - Indirizzo logico → indirizzo lineare: **Segmentazione**.
 - Indirizzo lineare → indirizzo fisico: **Page table gerarchica**.
 - gestita da **MMU**
 - **TLB** memorizza temporaneamente delle traduzioni



Memoria virtuale

- Traduzione ad indirizzo fisico:
 - Page table
 - gerarchica a 2 livelli
 - page number:
 - 20 bit (alti) = 10 (p1) + 10 (p2)
 - 1024 entry ciascuna per page table
 - *registro CR3*:
 - puntatore alla page directory del processo corrente
 - un bit di protezione per entry
 - kernel: 1 page table
 - user: 1 page table per processo utente
 - Dimensione pagina:
 - 12 bit (bassi) offset → 4 KB
 - Indirizzo fisico da 32 bit:
 - 20 bit frame number
 - 12 bit frame offset



Memoria virtuale

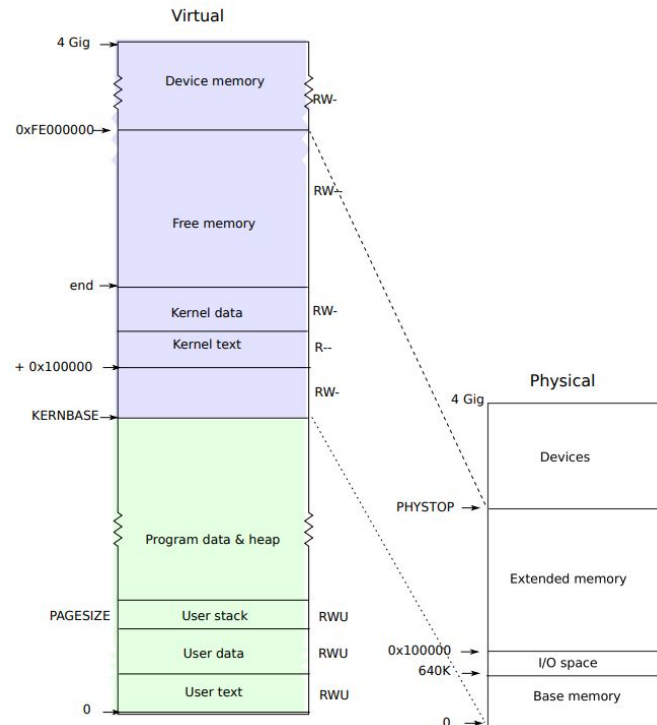
xv6 vs OS161

- **OS161** supporta il *paging* ed il *Demand Paging*, **xv6** solo il *paging*
- Dimensione della pagina: 4 KB
- Allocazione pagine:
 - in **xv6** *kalloc()* alloca solo una pagina
 - in **OS161** *ram_stealmem(npages)* alloca *npages* pagine.

Spazio di indirizzamento virtuale

- Dimensione: 4 GB
- Due porzioni:
 - user space
[0x0, KERNBASE]
 - kernel space
[KERNBASE, 0xFFFFFFFF]

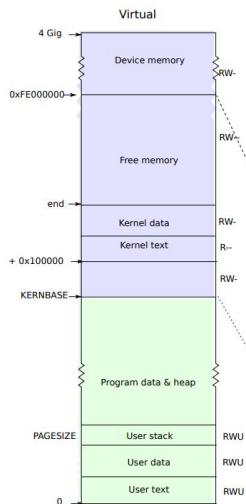
- costanti KERNBASE e
PHYSTOP definite in *memlayout.h*



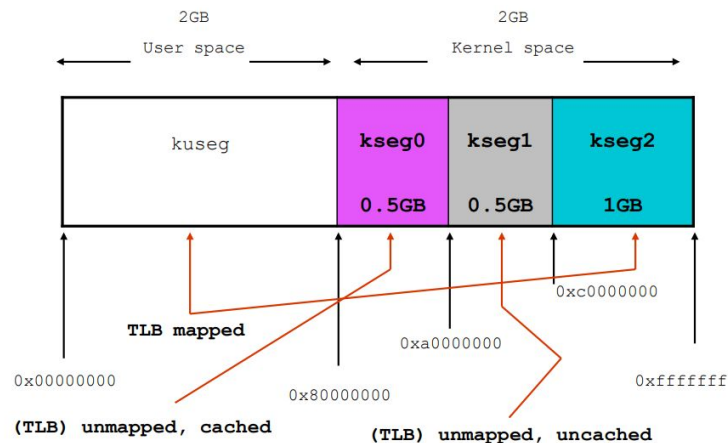
Spazio di indirizzamento virtuale xv6 vs OS161

Non ci sono particolari differenze tra gli spazi di indirizzamento virtuale dei due OS.

xv6

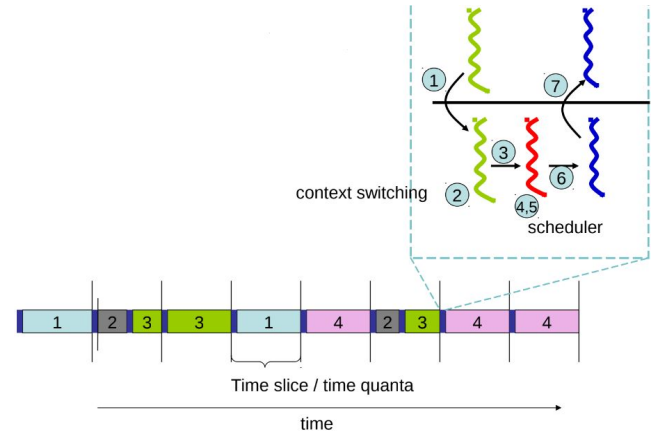


OS161



Algoritmi di scheduling

- Politica di scheduling: **Strawman Scheduling**
 - Variante del *Round-Robin*
 - Esecuzione di ogni processo per un dato intervallo di tempo.
 - Intervallo di tempo sui 100 ms
 - Timer interrupt
 - Pregi:
 - tutti i processi hanno la stessa probabilità di essere eseguiti dalla CPU
 - breve tempo di risposta
 - Difetti:
 - lungo tempo di attesa medio
 - overhead da context-switching



Algoritmi di scheduling

- Implementazione

- Data la lista di processi ***ptable.proc[]*** ed acquisito lo spinlock ***ptable.lock***, si individua il primo processo in stato RUNNABLE
- Lo si esegue (si pone *p->state = RUNNING*) e si esegue la funzione ***swtch()*** per passare dallo scheduler al processo user
- Interruzioni:
 - *sti()* abilita
 - *acquire(&ptable.lock)* disabilita
 - *release(&ptable.lock)* riabilita
 - Tra *sti()* ed *acquire()* si può rendere eseguibile un processo con un'interruzione

```
proc.c
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;

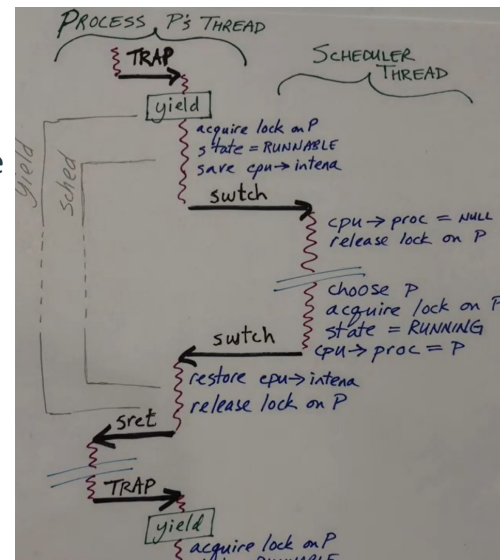
            // Switch to chosen process. It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

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

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

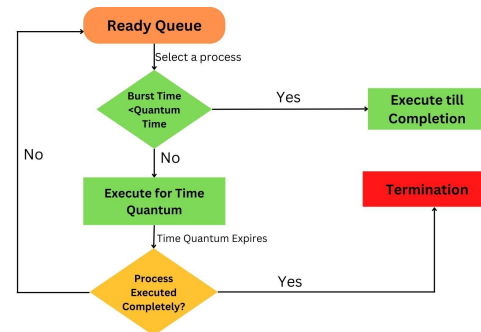
Context-switching

- Operazione di scambio tra processo in esecuzione (RUNNING) e processo eseguibile (RUNNABLE), salvando il contesto del primo e caricando i registri del secondo.
- Il passaggio da user a kernel mode avviene attraverso una **trap**.
 - Le *trap* sono temporizzate, determinate da timer interrupt.
- Il kernel thread chiama la funzione **yield()** (in proc.c) che forza il processore a rilasciare il controllo del thread in esecuzione.
- Il passaggio da kernel mode del thread a scheduler thread e viceversa viene gestita dalla funzione **sched()** (in proc.c).
 - Lo *scheduler thread* è gestito dalla funzione **scheduler()** (in proc.c).
- Il passaggio da kernel a user mode avviene attraverso una **trapret**.
 - Funzione svolta da **trapret()** (in trapasm.S).
- La funzione **swtch()** (in swtch.S) salva i registri correnti e carica i registri salvati nel thread.



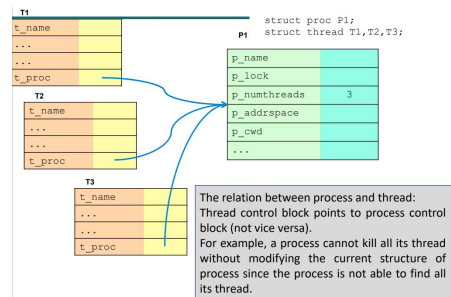
Scheduling xv6 vs OS161

- Per lo scheduling sono state adottate politiche simili
 - in **xv6** si segue una politica di **Strawman Scheduling**
 - in **OS161** si segue una politica di **Pre-emptive Round Robin**
- Per il context switching si hanno flussi di chiamate simili
 - in **xv6** `yield()` → `sched()` → `scheduler()` → `swtch()`
 - in **OS161** `thread_yield()` → `thread_switch()` → `switchframe_switch()`
 - La differenza tra i due sistemi operativi è che per **xv6** il context-switching avviene tra main-thread di processi diversi, mentre per **OS161** può avvenire tra thread dello stesso processo.




Processi e thread

- Su **OS161** è supportato il multi-threading: un processo può avere uno o più thread. Unità minima di elaborazione: thread.
- Su **xv6** è supportato solo il single-threading: i processi sono costituiti da un solo thread (main thread), non è implementato e supportato il multi-threading: non si ha un'implementazione per i thread e non ci sono campi appositi nella struct proc. Unità minima di elaborazione: processo.




```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
```



Sezione n.2

Implementazione delle funzionalità mancanti in xv6



Implementazione di system call

- Sviluppo di due system calls
 - mostrano informazioni sui processi attivi
 - **ps** fornisce informazioni sui processi attivi (PID, nome del processo, dimensione del processo in memoria)
 - **pstree** mostra le relazioni di parentela (padre-figlio) tra i processi

Implementazione di system call

Realizzazione:

1. Aggiungere una nuova mappatura in **syscall.h**: `#define SYS_name number`;
2. In **syscall.c** associare la costante con la funzione: `[SYS_name] sys_name`;
Rendere la funzione disponibile all'esterno: `extern int sys_name(void)`;
3. In **sysproc.c** implementare la funzione definita prima;
4. L'implementazione potrebbe continuare in **proc.c** in una nuova funzione da mettere nei file header **defs.h** e **user.h**;
5. In **usys.S** bisogna aggiungere la voce `SYSCALL(name)`;
6. Implementare comando da shell per chiamarla in un file sorgente apposito;
7. Includere il file sorgente del punto 6 nel **Makefile**;
8. Compilazione del sistema operativo per rendere disponibile il comando nella shell;

Implementazione di system call

ps

- Serve per elencare i processi attivi con le informazioni rilevanti, quali PID, nome, dimensione in memoria
- Funzionalità:
 - senza argomenti: stampa tutti i processi attivi
 - con un argomento intero: stampa il processo attivo con il PID uguale all'argomento, altrimenti restituisce un errore
- **ps** → `sys_getprocinfo()` → `getprocinfo()`
 - torna la lista dei processi attivi con le caratteristiche significative al chiamante
 - `struct pstat`: PID, nome, dimensione del processo in memoria, numero di processi attivi (in `processInfo.h`)

```
$ ps
PID      |      Name      |      Size
1         |      init       |    12288
2         |      sh         |   16384
6         |      ps         |    12288
```

```
$ ps 1
PID      |      Name      |      Size
1         |      init       |    12288
```

Implementazione di system call pstree

- Serve per costruire l'albero dei processi, in base alla parentela tra di essi.
- Informazioni utili: PID e nome del processo.
- Implementazione:
 - ricorsiva: visita in profondità dell'albero e stampa dei nodi/processi, grazie alla funzione `walk()`
 - **Parametri**: vettore dei processi, PID padre, vettore di memorizzazione nodi, indice di quest'ultimo.
 - **Criticità**: causa l'esaurimento di memoria dinamica
 - iterativa: si legge la lista dei processi da destra a sinistra, si calcola il vettore dei padri e si stampa per ogni processo l'intero ramo dell'albero
 - **Criticità**: stampe multiple dello stesso ramo senza qualche nodo
- **`pstree`** → `sys_getproctree()` → `getproctree()`
- Non è stato provato il caso in cui un processo ha più figli per mancata implementazione da parte del sistema operativo

```
init: starting sh
$ sh ; ps
$ pstree
[PID: 1 Name: init] -> [PID: 2 Name: sh] -> [PID: 3 Name: sh] -> [PID: 4 Name: sh] -> [PID: 5 Name: pstree]
$ sh ; pstree
allocation out of memory
lapicid 0: panic: kfree
80102555 80106bee 801070ca 80103adf 80105af6 80104de9 80105e6d 80105c04 0 0
```

```
$ pstree
[PID: 3 Name: pstree] <- [PID: 2 Name: sh] <- [PID: 1 Name: init]
[PID: 2 Name: sh] <- [PID: 1 Name: init]
$
```

Debug della system call

- Quando viene chiamata la system call viene generato un interrupt
- **trap()** in *trap.c* verifica di che tipologia è l'interrupt
 - se è di tipo T_SYSCALL allora viene chiamata la funzione **syscall()**
 - di conseguenza viene invocata la funzione che la implementa per l'esecuzione del codice

```
hadoop@PC:~/hadoop$ cd ..&& make qemu-gdb
sed "s/localhost:1238/localhost:26880/" < .gdbinit.tpl > .gdbinit
** Now run "gdb"
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -smp 2 -m 512 -S
gdb tcp://26880
v6.
qemu starting 0
sh: size 1080 mblocks 941 minodes 280 nlog 30 logstart 2 inodestart 32 heap
start 58
just starting sh
i ps 1

hadoop@PC:~/hadoop$ cd ..&& make qemu-gdb
sed "s/localhost:1238/localhost:26880/" < .gdbinit.tpl > .gdbinit
** Now run "gdb"
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -smp 2 -m 512 -S
gdb tcp://26880
v6.
qemu starting 0
sh: size 1080 mblocks 941 minodes 280 nlog 30 logstart 2 inodestart 32 heap
start 58
just starting sh
i ps 1
```



```

root@BTVM-PC:~/xv6/xv6# make qemu-gdb
sed -i 's/localhost:1234/localhost:20000/' <.gdbinit.tmpl> .gdbinit
$ cd
$ ./xv6
$ qemu-system-i386 -serial mon:stdio -drive file:fs.img,index=1,media=disk,format=raw,if=ata,raid=0 -drive file:svs.img,index=0,media=disk,format=raw -smp 2 -m 512 -S
$ gdb top:20000
$ v
$ qemu: starting 0
$ b: size 10880 nblocks 960 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
start 58
init: starting sh
$ ps 1

```

```

Continuing.

Thread 1 hit breakpoint 3, trap (tf=0x80df2f28) at trap.c:139
93 if (tf & trapno == T_SYSCALL)
94 {
Continuing.

Thread 1 hit breakpoint 4, sys_getprocinfo (i) at sysproc.c:98
98 if (argptr < 0, (char **)id, sizeof(struct pstab)) < 0)
99 {
100 {
101 int
102 sys_getprocinfo(void)
103 {
104 {
105 struct pstab *d;
106 if (argptr < 0, (char **)id, sizeof(struct pstab)) < 0)
107 return -1;
108 getprocinfo(d);
109 return 0;
110 }
111 }
112 }
113 }
Continuing.

Thread 1 hit breakpoint 5, getprocinfo (ps&0x290c) at sys.c:586
586 acquire(&table lock);
587 {
588 {
589 int
590 getprocinfo(struct pstab* ps)
591 {
592 int i;
593 struct proc *p;
594 acquire(&table lock);
595 for (i = 0; p = &table proc; p = &table proc; i++)
596 if (p->struct proc (i,0) < sizeof(struct proc) -> p (struct proc NEW)
597 if (p->state != UNUSED)
598 if (p->nid[i] == p->pid;
599 }
600 }

```



Fine