

# **Analisi Comparativa tra OS161 e xv6**

corso di “Programmazione di sistema” [02GRSOV]  
a.a. 2022/2023



**Politecnico  
di Torino**

realizzato dal gruppo n.40  
Jellouli Hamza s308734  
Magistro Contenta Ivan s314356  
Marinacci Giuseppe s320001

# Sezione 1

## Introduzione

xv6

Il Sistema Operativo “xv6” è un sistema operativo basato su Unix. Esso è stato progettato presso il MIT, nel 2006, a scopo didattico, cioè con l’obiettivo di insegnare i principi dei sistemi operativi. È scritto in linguaggio C ed è disponibile su GitHub. xv6 fornisce una buona base per l’apprendimento dei principi dei sistemi operativi e richiede, sempre a tale scopo, l’implementazione di alcune funzionalità.

xv6 è una reimplementazione della versione 6 di Unix scritta da Dennis Ritchie e Ken Thompson. V6 era stata usata, sempre al MIT e con il medesimo scopo, a partire dal 2002, per poi essere rimpiazzata proprio da xv6. Importante è stato anche l’apporto del “Commentario sulla Sesta Edizione di Unix” di John Lions (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)).

xv6 contiene codice estratto dalle seguenti risorse:

- JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
- Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
- FreeBSD (ioapic.c)
- NetBSD (console.c)

xv6 fa tanto uso di codice in C ed in Assembly. Ne esistono due implementazioni, le quali risultano quasi identiche:

- **x86**: anche conosciuta come “80x86” o “famiglia 8086”, è una famiglia di architetture CISC (“Complex Instruction Set Computer”) sviluppate da Intel e basate sul microprocessore Intel 8086 (e sulla sua variante 8088) a 32 bit;
- **RISC-V**: appartiene a una famiglia di architetture RISC (“Reduced Instruction Set Computer”) di 5<sup>a</sup> generazione, open-source e open-architecture, ISA da 32 bit e 64 bit per lo spazio di indirizzamento.

Vediamo la versione per processori x86.

OS161

Anche OS161 è un sistema operativo a scopo didattico, ovvero un sistema semplificato e usato per tenere corsi universitari sui sistemi operativi. Esso è di tipo BSD-like ed è più simile ad un sistema operativo “reale” rispetto a molti altri SO a scopo didattico; seppure sia eseguito su un simulatore, possiede la struttura e il design di un sistema più complesso. OS161 si basa su un’architettura MIPS con processore a 32 bit (un’architettura ampiamente adottata nell’ambito didattico).

# System Calls

## Introduzione

Una *system call* è un servizio, la cui natura può essere varia, che un Sistema Operativo offre e che può essere richiesta da un programma utente. Solitamente, per richiedere una system call è necessario scatenare un **interrupt**: si tratta di un meccanismo mediante il quale il programma in corso di esecuzione si *interrompe* al fine di effettuare la richiesta al SO.

Inoltre, tipicamente è il kernel a gestire tutti i tipi di interrupt (siano essi dovuti a syscalls o ad altro, ad esempio eccezioni o timer) anzichè lasciarli gestire ai processi, perché solo esso possiede i privilegi necessari. In caso di interrupt, il SO deve quindi essere in grado di:

- **salvare i registri** del processore, in vista di una futura ripresa trasparente una volta terminata l'esecuzione della system call;
- essere predisposto per l'esecuzione di istruzioni in **kernel mode**;
- **recuperare le informazioni** riguardanti l'evento (ad esempio, i parametri da passare alla system call);
- eseguire tutto ciò in modo **sicuro**, cioè mantenere l'isolamento tra processi utente e processi kernel.

Nella maggior parte dei SO, il modo per gestire un interrupt è avviare l'esecuzione di una nuova sequenza chiamata **interrupt handler**. Prima di avviare l'interrupt handler, il processore salva i suoi registri, in modo che il SO possa ripristinarli al ritorno dall'interrupt. Una sfida nella transizione da e verso l'interrupt handler è che il processore deve passare dalla *user mode* alla *kernel mode* e viceversa.

xv6

## Trap nelle architetture x86

Una nota sulla terminologia: sebbene il termine ufficiale in x86 sia "*interrupt*", x86 si riferisce a tutti questi eventi come "*trap*". Utilizzeremo i termini *trap* e *interrupt* in modo intercambiabile, ma è importante ricordare che le trap sono causate dal processo in esecuzione su un processore (ad esempio, il processo effettua una system call e di conseguenza genera una trap), mentre gli interrupt sono causati da dispositivi periferici e possono non essere correlati al processo in esecuzione.

In x86, gli interrupt handlers sono definiti nella *Interrupt Descriptor Table (IDT)*, che è salvata nel file *vector.S*. Esso presenta 256 voci, di cui la **64-esima** è assegnata alle system calls.

```

313    vector63:
314        pushl $0
315        pushl $63
316        jmp alltraps
317    .globl vector64
318    vector64:
319        pushl $0
320        pushl $64
321        jmp alltraps
322    .globl vector65
323    vector65:
324        pushl $0
325        pushl $65
326        jmp alltraps
327    .globl vector66

```

In x86, per generare un interrupt si usa il comando “**int n**” (la sua implementazione in xv6 verrà approfondita nella sezione “*stub routine*”). Essa ha il compito di recuperare l’n-esimo descrittore dall’IDT e, in seguito, inizia la costruzione del *trap frame*, e per farlo si avvale di *alltraps* (definita in **trapasm.S**).

```

#include "mmu.h"

# vectors.S sends all traps here.
.globl alltraps
alltraps:
    # Build trap frame.
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal

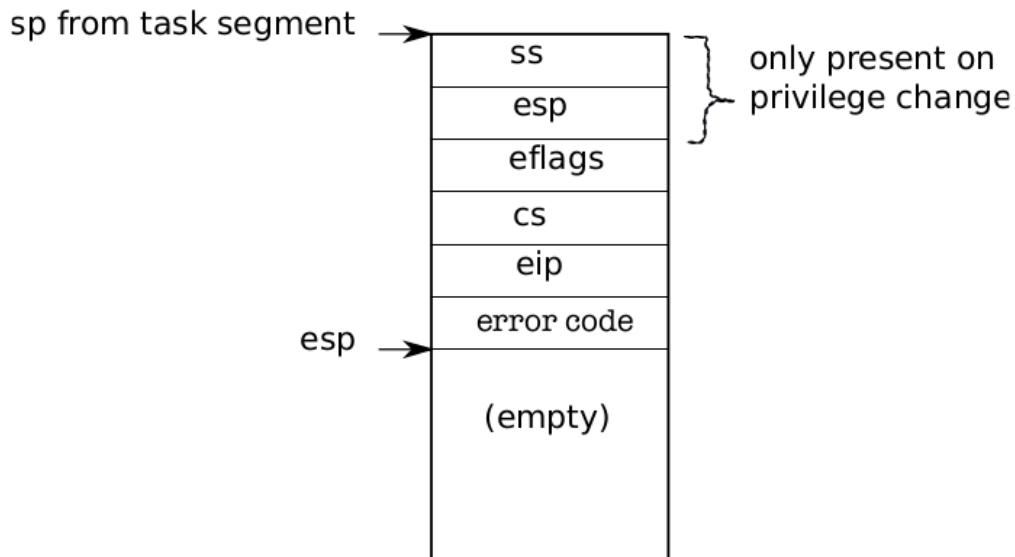
    # Set up data segments.
    movw $(SEG_KDATA<<3), %ax
    movw %ax, %ds
    movw %ax, %es

    # Call trap(tf), where tf=%esp
    pushl %esp
    call trap
    addl $4, %esp

    # Return falls through to trapret...
.globl trapret
trapret:
    popal
    popl %gs
    popl %fs
    popl %es
    popl %ds
    addl $0x8, %esp  # trapno and errcode
    iret

```

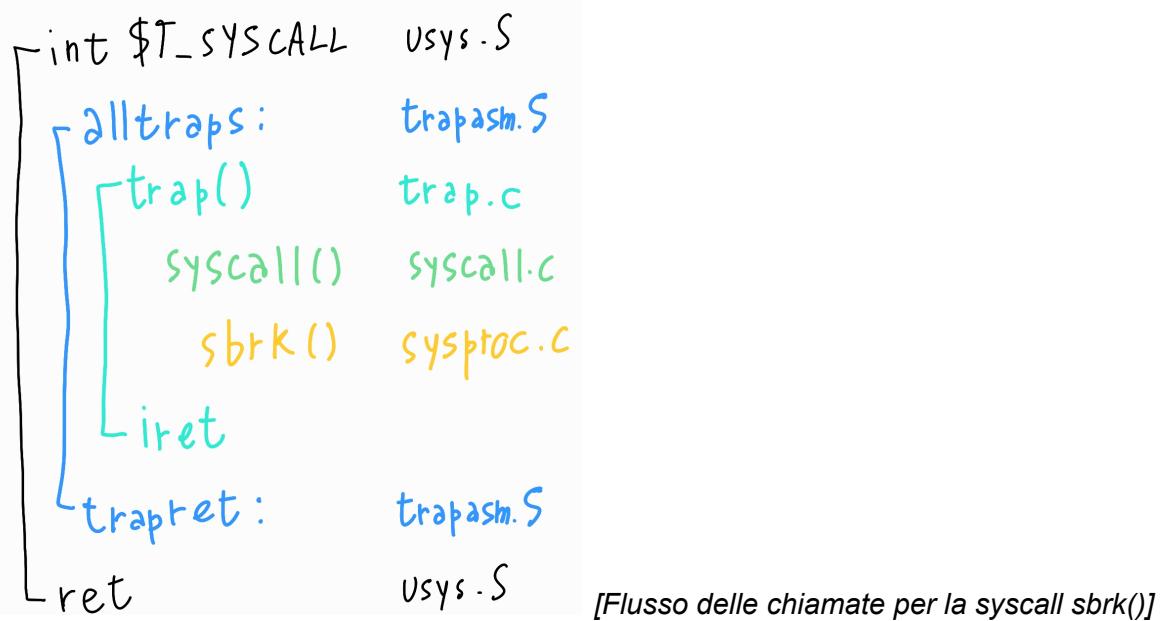
Una serie di registri verranno caricati nel **kernel stack** e verranno usati per l’esecuzione della system call da parte del kernel (si noti che in %ss ed in %esp sono salvate le informazioni per ritornare al processo utente). Prima di iniziare ad effettuare la system call vera e propria, il trap frame nel kernel stack risulterà simile a quello riportato in figura:



A questo punto, `alltraps` chiama `trap` passando l'inizio del trap frame come parametro.

### Implementazione software in xv6

In xv6, ogni programma in user mode che voglia chiamare una system call deve includere le librerie `user.h`, `usys.S` (che contiene il codice assembler per generare un interrupt per le system calls in user mode), e `syscall.h` (che contiene la definizione delle syscall, usate sia in user mode che in kernel mode). Di seguito si riportano i principali passaggi che avvengono durante l'esecuzione di una system call, utilizzando come esempio la system call `system_break` o `sbrk()`.



`sbrk()` è la system call utilizzata, nel contesto del sistema operativo xv6, da parte di un programma, per richiedere al kernel di incrementare (o decrementare) la dimensione dell'heap ad esso assegnato (ossia, di richiedere più o meno memoria per l'allocazione dinamica dei dati). Essa richiede un solo parametro: un intero che indichi, in bytes, la

dimensione dell'incremento (o decremento) richiesto dal processo, e ritorna 0 in caso di successo o -1 in caso di errore.

### **Implementazione di trap()**

Una trap viene lanciata ogni volta che una system call deve essere eseguita. Le trap vengono gestite dalla funzione **trap()** (in *trap.c*): quando si verifica una trap, il processore passa alla kernel mode e trasferisce il controllo alla funzione appena citata. Essa avrà il compito di determinare la causa scatenante della trap (syscall, page fault, timer interrupt, ecc.) e di invocare le opportune funzioni di gestione: nel caso di una system call, si tratta della funzione **syscall()**.

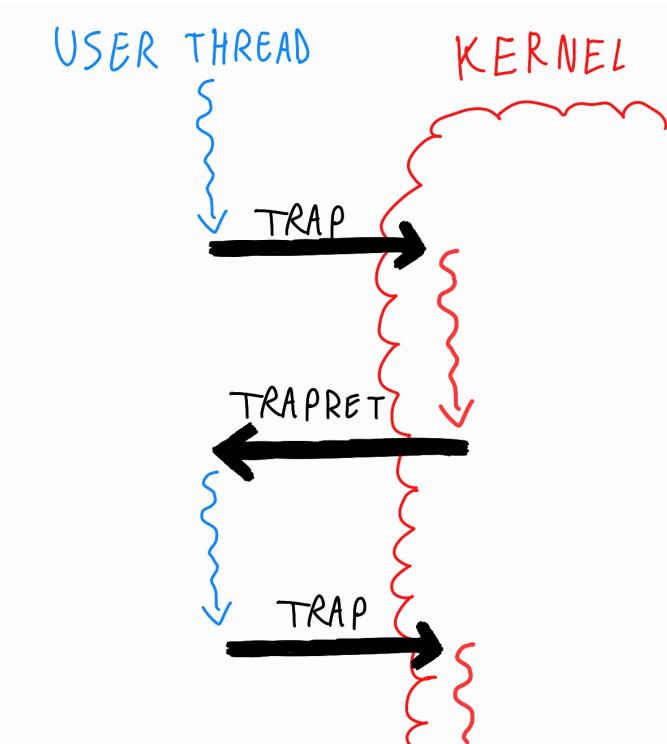
Quest'ultima, situata nel file *syscall.c*, legge il numero della system call dal registro **%eax**: se ad esso corrisponde effettivamente una syscall, farà in modo che essa venga invocata, altrimenti ritornerà un messaggio d'errore e imposterà **%eax** a -1.

A questo punto, il codice in *syscall.c* è stato in grado di identificare quale system call deve essere eseguita: il codice C da eseguire si troverà in *sysproc.c*.

A questo punto ogni system call conterrà il proprio codice. Nel caso della *sbrk()*, si utilizzano:

- La funzione **argint()**, che esegue il fetch del parametro (in questo caso, il numero di bytes di cui incrementare la dimensione dell'heap del processo, che può essere negativo) dal registro **%esp**, da usare in seguito;
- La funzione **growproc()** (definita in *proc.c*), che chiama le funzioni **allocuvm()** o **deallocuvm()** (definite in *vm.c*) che si occupano, rispettivamente, di assegnare e di sottrarre lo spazio all'heap del programma: la scelta dipende dal segno del parametro ottenuto in precedenza.

A questo punto viene invocata una funzione di ritorno mediante il codice in **trapret** (in *trapasm.S*). Esso ripristina tutti i registri salvati nel *trap frame* ed è così che il programma prosegue dopo l'esecuzione di una trap, ritornando ad eseguire il codice in user mode.



## **Stub routine**

Quando una system call viene chiamata, parte un programma in user mode chiamato "**stub routine**": in usys.S si trova una macro, che può essere pensata come il "modello" di una funzione.

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

Quello che avviene è che, in fase di pre-compilazione, il compilatore sostituisce a tale modello una funzione prima di effettuare la compilazione vera e propria. Ad esempio, al posto del placeholder *name* si utilizza il nome di una system call, come la *getpid*.

```
#define SYSCALL(getpid) \
.globl getpid; \
getpid: \
    movl $SYS_getpid, %eax; \
    int $T_SYSCALL; \
    ret
```

Come mostrato dal codice, tale procedimento è ripetuto per ciascuna delle 21 syscall. La funzione dello stub, che a questo punto avrà il nome di *getpid*, sarà quella di spostare il numero *SYS\_getpid* (definito in **syscall.h**) nel registro *%eax* e chiamare un interrupt (comando *int*) con il numero *T\_SYSCALL* (è 64 ed è definito in **traps.h**).

## **Elenco delle system call**

In totale, xv6 presenta una lista di 21 system calls già implementate, ciascuna con la propria funzione e con un intero assegnato come identificativo (le associazioni tra system call ed identificativi sono definite in **syscall.h**, mentre le implementazioni nei file sorgente **syscall.c**, **sysproc.c** e **sysfile.c**):

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 buf	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

## OS161

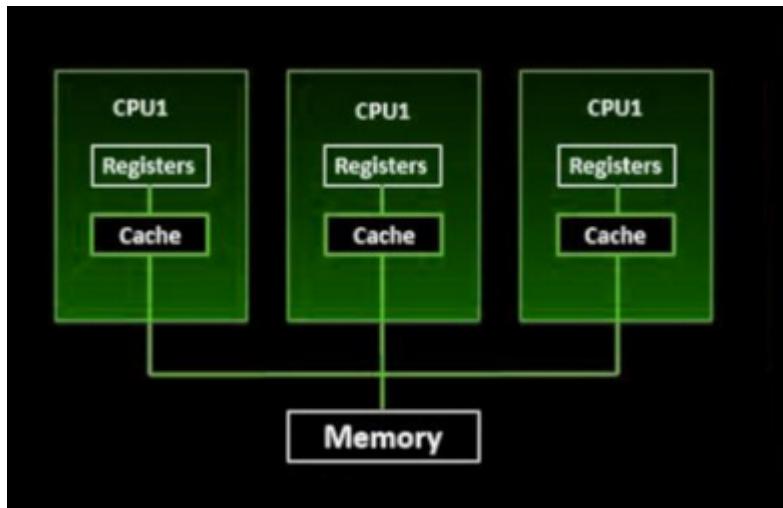
Anche in OS161 le system calls offrono ai processi un'interfaccia attraverso la quale accedere ai servizi messi a disposizione dal sistema operativo.

Il meccanismo di funzionamento è estremamente simile a quello appena descritto per xv6. Si possono infatti riconoscere, seppur con le dovute differenze implementative (ad esempio, l'utilizzo del linguaggio MIPS invece che di x86), tutti i tratti salienti dell'esecuzione di una system call:

- il primo evento che si verifica è lo scatenarsi di una **trap**, per la quale si deve per prima cosa verificare se essa sia stata causata dalla richiesta di una syscall o altro (timer, eccezioni, ecc.);
- in seguito si ha l'operazione di **context-switching**, ovvero il passaggio dalla *user mode* alla *kernel mode* (con conseguente creazione di un *trap frame*, salvato nel kernel stack, che consenta il passaggio in verso opposto alla fine dell'esecuzione senza perdere lo stato corrente della user mode);
- avviene poi l'esecuzione vera e propria della system call, che differisce a seconda di quale syscall è stata invocata;
- infine si ritorna alla *user mode* e l'esecuzione del programma riprende;

Come in xv6, anche OS161 fornisce una lista di system calls a cui è associato un ID, che è il parametro che si utilizza per selezionare la system call da eseguire.

## Meccanismi di sincronizzazione



### Introduzione

I meccanismi di sincronizzazione sono strumenti essenziali per gestire il coordinamento tra processi, attività o dispositivi che operano 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.

La sezione di codice condivisa tra più processi è detta *critical section*.

xv6

I principali meccanismi di sincronizzazione in xv6 sono:

- **Spinlock:** Sono utilizzati per ottenere l'accesso ad una risorsa condivisa in modo esclusivo ed atomico da parte di processi/threads, questo meccanismo fa busy waiting. Sono definiti con un'apposita struct chiamata "spinlock" in spinlock.h al cui interno è presente un campo che indica se lo spinlock è stato acquisito o meno, oltre ai campi nome e core della CPU a cui appartiene. Per accedere in modo esclusivo ad una determinata area di memoria, un thread deve acquisire lo spinlock corrispondente e rilasciarlo quando si esce dal codice che si vuole proteggere. Si potrebbe verificare che più thread verifichino simultaneamente la condizione per modificare il valore dello spinlock e quindi entrambi avrebbero il possesso dello spinlock: per risolverlo si utilizza un approccio basato sulla "test and set" a basso livello (assembly, facciamo riferimento alla funzione `xchg()`).

#### Istruzioni:

- **acquire:** Funzione che serve per acquisire il possesso dello spinlock. Consuma cicli CPU perché continua ad eseguire l'operazione di controllo dello spinlock(ciclo while). La funzione `xchg` serve per cambiare il valore dello spinlock da 0 a 1. Se questo si trova già ad 1 allora ritorna 0. E' importante notare la funzione `pushcli()` che disabilita gli interrupt. Questo perché disabilitando e riabilitando gli interrupt, si prevengono timeslicing quando il

lock è già posseduto , prevenendo quindi il processamento dell'handler dell'interrupt avvenga. Quindi si esegue tutto velocemente senza alcuna interruzione.

```

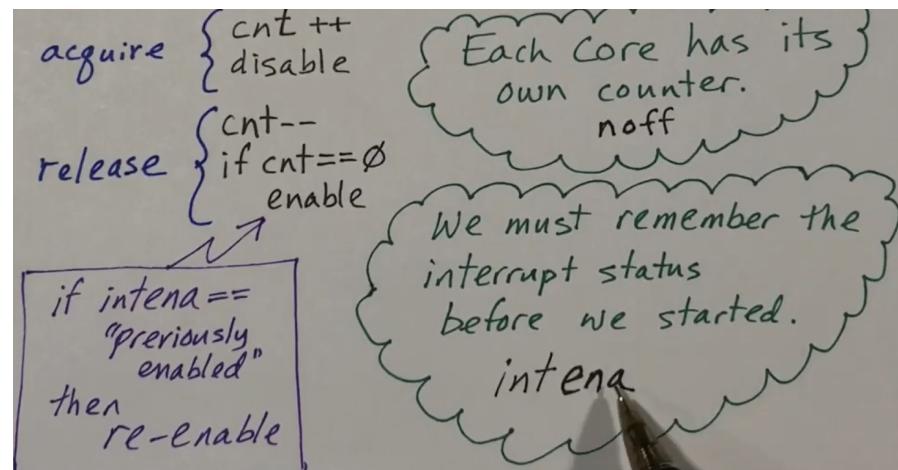
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");

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

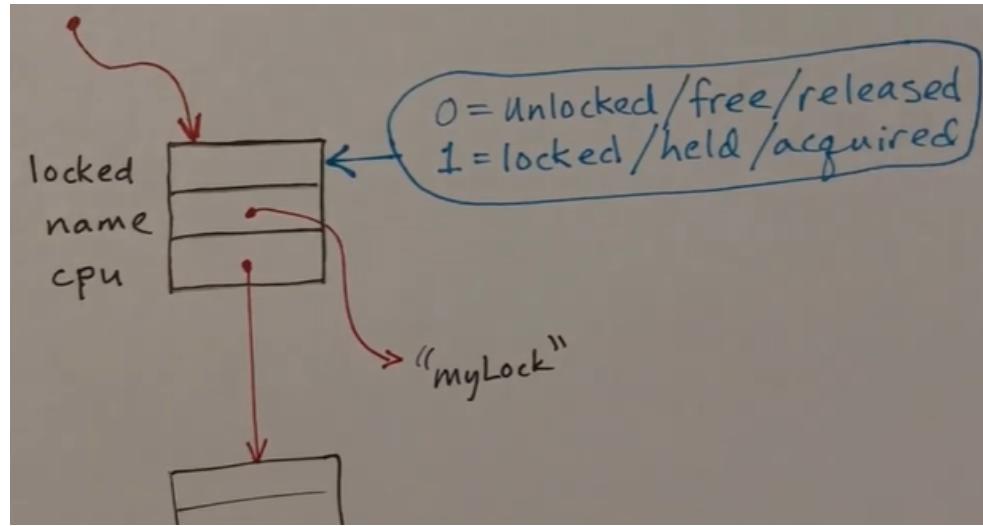
    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

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

```



- **release**: Funzione che serve per il rilascio dello spinlock.
- **initlock**: Funzione che serve per definire i parametri dello spinlock. I parametri sono: Nome, locked(per segnalare se è stato acquisito), cpu(per tenere traccia dell'id della cpu che ha acquisito lo spinlock).



- **Sleeplock:** è un meccanismo di sincronizzazione che implementa un approccio di attesa attiva (spinlock), insieme ad uno di attesa passiva (mettere il processo in stato di sleeping). Una volta acquisito lo spinlock della struttura dati si controlla se si può acquisire il possesso. Se questo è possibile si procede a impostare la variabile e a liberare lo spinlock, se invece non è possibile si mette il processo nello stato di sleeping. La caratteristica fondamentale è che i processi che vengono messi nello stato di sleeping possiedono lo spinlock, impedendo quindi race condition. I processi nello stato di sleeping vengono svegliati attraverso gli interrupt e per questo motivo hanno un utilizzo limitato. A differenza degli spinlock questo meccanismo è adatto per le lunghe attese. Le funzioni per la gestione dello sleeplock sono simili a quelle dello spinlock e si trovano nel file *sleeplock.c*.

```

1 // Long-term locks for processes
2 struct sleeplock {
3     uint locked;          // Is the lock held?
4     struct spinlock lk; // spinlock protecting this sleep lock
5
6     // For debugging:
7     char *name;           // Name of lock.
8     int pid;              // Process holding lock
9 };

```

<pre> void acquiresleep(struct sleeplock *lk) {     acquire(&amp;lk-&gt;lk);     while (lk-&gt;locked) {         sleep(lk, &amp;lk-&gt;lk);     }     lk-&gt;locked = 1;     lk-&gt;pid = myproc()-&gt;pid;     release(&amp;lk-&gt;lk); } </pre>	<pre> void releasesleep(struct sleeplock *lk) {     acquire(&amp;lk-&gt;lk);     lk-&gt;locked = 0;     lk-&gt;pid = 0;     wakeup(lk);     release(&amp;lk-&gt;lk); } </pre>
---	---

- **Sleep/Wakeup:** Sono il metodo più comune per evitare di fare busy waiting.  
**Sleep:** un processo che ha bisogno di attendere che una condizione sia soddisfatta chiama la funzione sleep. Qui il processo non fa perdere cicli macchina alla CPU e viene inserito in una coda insieme ad altri processi che attendono per la stessa condizione. Quando un altro processo che ha cognizione del fatto che la condizione è stata soddisfatta chiama la funzione Wakeup. La funzione **Wakeup** sposta il processo dalla coda di attesa (wait) a quella di ready, rendendo di fatto il processo di nuovo eseguibile. La variabile **void \*chan** rappresenta un intero ed indica il canale in cui il processo è stato messo nello stato di sleeping, quando verrà eseguita la Wakeup tutti i processi nel canale indicato come parametro che sono nello stato di sleeping vengono svegliati. Per utilizzare la sleep è comunque necessario passare per la syscall sys\_sleep definita nel file sysproc.c. E' importante notare che anche in questo caso si utilizzano gli spinlock per proteggere le sezioni critiche nelle funzioni sleep e wakeup, ma questi sono mantenuti per poco tempo per considerarla attesa attiva.

```
void sleep(void *chan, struct spinlock *lk)
{
    if(lk != &ptable.lock) {
        acquire(&ptable.lock);
        release(lk);
    }

    // Go to sleep
    proc->chan = chan;
    proc->state = SLEEPING;
    sched(); // context switch

    if(lk != &ptable.lock) {
        release(&ptable.lock);
        acquire(lk);
    }

    // Wake up all processes sleeping on chan.
    // The ptable lock must be held.
    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);
    }
}
```

```
int
sys_sleep(void)
{
    int n;
    uint ticks0;

    if(argint(0, &n) < 0)
        return -1;
    acquire(&tickslock);
    ticks0 = ticks;
    while(ticks - ticks0 < n){
        if(myproc()->killed){
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}
```

Come si può notare i meccanismi di sincronizzazione sono simili ad OS161, differiscono solo nelle loro implementazioni.

## OS161

### Lock e Spinlock

**Lock:** Meccanismo di sincronizzazione che garantisce la mutua esclusione, cioè solo un thread può accedere alla sezione critica, ed è presente un concetto di possesso (ownership). Si tratta di un meccanismo senza busy waiting rispetto a spinlock: si fa busy waiting solo per pochissime istruzioni, quindi non si parlerà di vera e propria busy waiting, è trascurabile.

**Spinlock** serve per entrare nelle *sezioni critiche*, garantire la *mutua esclusione* ed a compiere delle istruzioni in *modo atomico*. Però è *busy-waiting*, quindi consuma cicli della

cpu durante l'attesa. Si possono acquisire spinlock diversi, ma non si può acquisire più volte lo stesso spinlock se non è ancora stato rilasciato.

Differenza tra lock e spinlock: il secondo produce *busy wait*, mentre il primo no. Quindi il secondo può essere utile per operazioni molto veloci e non per operazioni molto lunghe che sprecano tempo di CPU. Infatti i lock sono implementati con gli spinlock per fare accesso a zone critiche che contengono poche istruzioni, al resto ci penseranno i lock.

**Semaphore**: Permette l'accesso concorrente da parte di processi/thread ad un numero limitato di risorse. Il programmatore decide quanti processi/thread possono entrare nell'area protetta, impostando il semaforo. Ogni volta che un processo/thread cerca di entrare nell'area protetta, chiama una funzione che decrementa il contatore se è maggiore di zero (*wait(P)*). Se è maggiore di zero il processo/thread ritorna e può procedere nell'esecuzione del codice protetto. Se invece il contatore è a zero, il processo/thread rimane in attesa fin quando questo non torna ad essere maggiore di zero. Quando un processo/thread termina di eseguire il codice protetto chiamano una funzione che incrementa il valore del contatore (*signal(V)*). *wait(P)* e *signal(V)* sono eseguiti in maniera atomica. I semafori possono essere implementati sia in modo che facciano busy waiting utilizzando gli spinlock, sia con attesa passiva usando il meccanismo Sleep/Wakeup

**Condition variable**: È il meccanismo di sincronizzazione che permette di proseguire nell'esecuzione del codice al verificarsi di una condizione, che deve essere soddisfatta da un thread diverso da quello corrente. Il thread viene messo in una coda di attesa. Il thread che soddisfa la condizione può decidere se risvegliare tutti i thread in attesa o solo uno. Sono costruite utilizzando i lock e quindi non fanno busy waiting.

**Wchannel**: molto simili alle condition variable, ma a differenza di essi sono costruite utilizzando gli spinlock (busy waiting) e sono utilizzabili solo nel kernel.

Esistono una serie di comandi per svegliare un solo thread: *cv\_signal()* per le condition variable e *wake\_one()* per i wchannel. Oppure più thread: *cv\_broadcast()* per le condition variable e la *wchan\_wakeall()* per i wchannel. Mentre per mettere in attesa dei segnali i thread si utilizzano i seguenti comandi: *wchan\_sleep()* per i wchannel e *cv\_wait()* per le condition variable.

# Memoria virtuale e MMU

## Introduzione

La **Memoria Virtuale** (Virtual Memory) consente al sistema di trasferire dati dalla memoria principale (RAM) al disco rigido e viceversa, nell'eventualità in cui la RAM debba essere utilizzata per operazioni più sensibili o che richiedano migliori prestazioni. In altre parole, rappresenta la separazione della memoria logica dalla memoria fisica.

Il **paging** è una tecnica per dividere in porzioni di dimensioni fisse la memoria virtuale e/o fisica.

La *Virtual Memory* viene implementata tramite *Demand Paging* o *Demand Segmentation*.

Tramite il **Demand Paging** si possono trasportare solo le pagine che ci servono.

Tramite il **Demand segmentation** si ha la divisione dei dati richiesti in piccoli segmenti per migliorare le performance.

Le **page table** rappresentano un meccanismo attraverso il quale ricavare dal *numero di pagina* (pagina logica) da consultare il corrispondente *numero di frame* (pagina fisica) e quindi sarà utile per ottenere l'indirizzo fisico finale concatenando a quest'ultimo il page offset. Queste tabelle risiedono nella memoria fisica. Le tabelle sono appunto organizzate come insieme di entry ognuna delle quali contiene la corrispondenza tra page number e frame number.

La **Virtual Address Space** è una porzione di memoria logica che il sistema operativo rende disponibile ad un processo. Il sistema operativo nasconde l'indirizzo di memoria fisica effettivo nella RAM e utilizza gli indirizzi virtuali per indirizzare pagine in RAM e su disco.

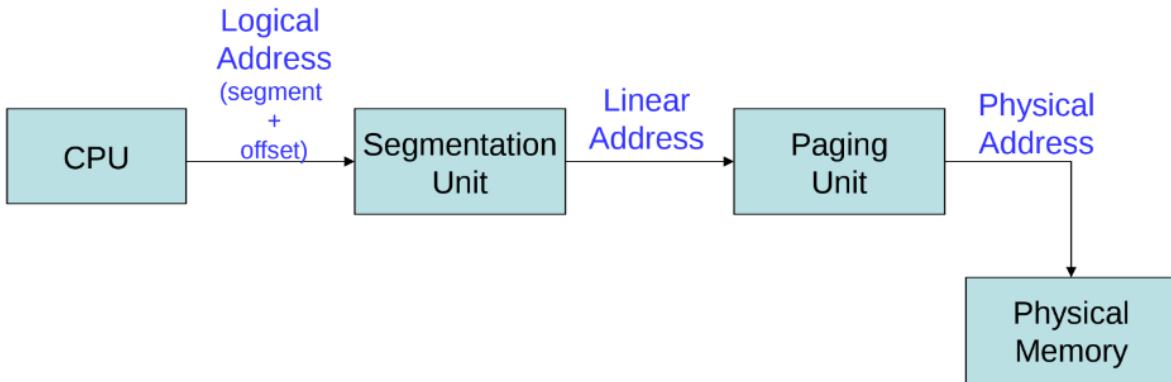
xv6

## Gestione memoria virtuale e traduzione indirizzi

Nella *versione x86* (32 bit) di xv6, la gestione della memoria virtuale è realizzata attraverso l'uso della **paginazione hardware** fornita dall'architettura stessa.

xv6 non dispone di *demand paging* dal disco, fork *copy-on-write*, *memoria condivisa* ecc.

I processori di tipo x86 supportano la traduzione degli indirizzi usando la **segmentazione** (strategia per la quale l'intero spazio di memoria è diviso in segmenti ognuno dei quali è assegnato ad un processo). A partire dall'indirizzo logico (contenente il Segment Selector), si consulta la *Descriptor Table*: si tratta di una tabella tipicamente disponibile a livello globale della quale si conosce il puntatore a tale tabella; contiene *Segment Descriptor* ai vari segmenti, dai quali si ottiene l'indirizzo base al quale aggiungere l'offset per ottenere l'indirizzo lineare da 32 bit (4 B).



Le **page table** sono di tipo *gerarchico* a 2 livelli, utili per risolvere il problema della memoria contigua; quindi i 32 bit dell'*indirizzo lineare* vengono suddivisi in tre porzioni, due per il page number ed una per il page offset: dei 20 bit alti per ottenere il frame number, i 10 bit più alti servono per individuare in quale delle tabelle di pagine di secondo livello si va a cercare l'entry desiderata, mentre i restanti 10 bit per ottenere il numero di frame da consultare, codificato su 20 bit di frame number. Tolti i 20 bit alti, rimangono 12 bit bassi, dedicati al page offset, i quali identificano il byte della pagina da consultare la cui dimensione è pari a  $2^{12}$  B = 4 KB; quindi si aggiungono questi 12 bit dopo il frame number per ottenere l'*indirizzo fisico* da consultare.

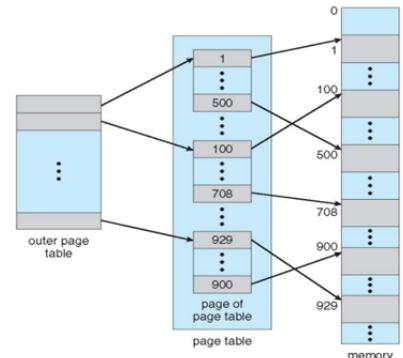
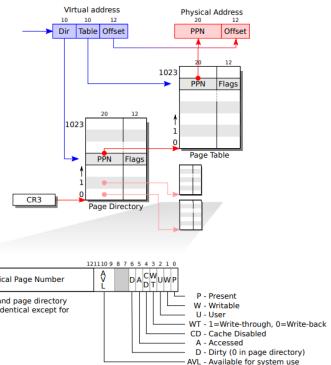
In definitiva si può vedere il tutto come un *albero a 2 livelli*, dove la page table alla radice (outer page table) contiene  $2^{10} = 1024$  entry che indirizzano ad altrettante page table di secondo livello (inner page table) ognuna delle quali possiede  $2^{10} = 1024$  entry contenenti l'associazione page number - frame number, grazie alla quale arriviamo al nodo foglia corrispondente alla pagina fisica da consultare.

Si ha una struttura ad albero che consente di avere una *gerarchia di indirizzamento virtuale*.

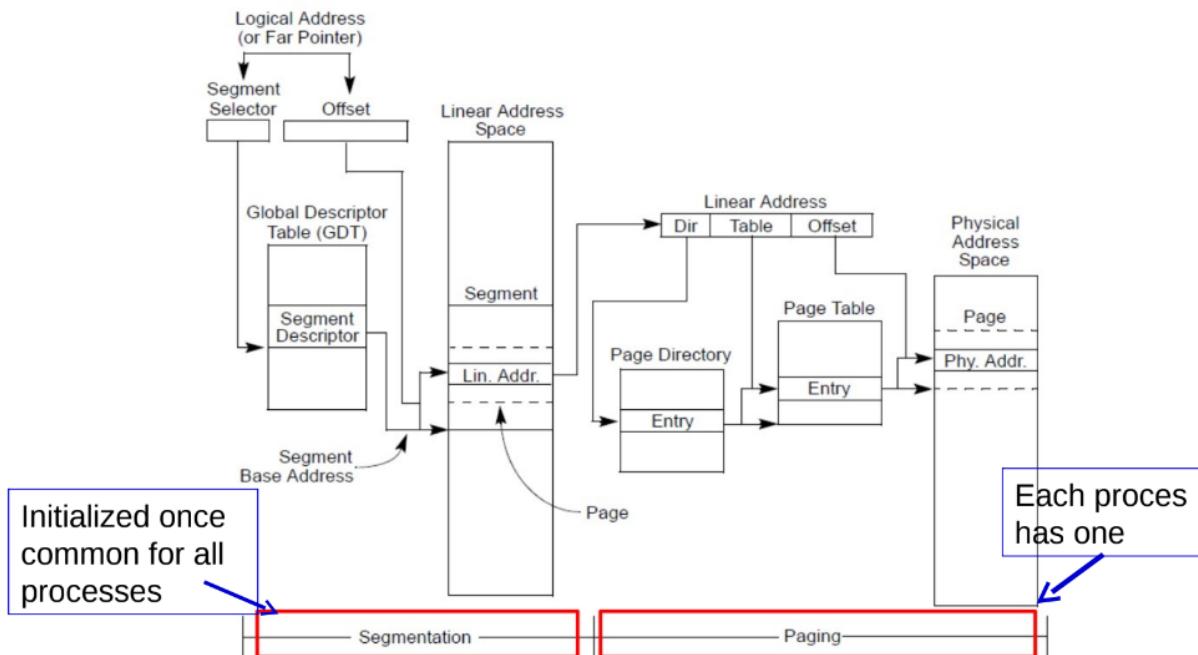
Inoltre ogni entry della page table è caratterizzata da *bit di protezione* per garantire sicurezza alla memoria del processo.

In xv6 ogni *processo* ha uno *spazio di indirizzamento* separato da quello degli altri processi e di conseguenza ognuno di loro ha una propria *page table* che indirizza alla singola memoria fisica. Il *registro CR3* contiene il puntatore alla page directory (od outer page table) del processo in esecuzione.

La traduzione da indirizzo virtuale ad indirizzo fisico viene gestita dal MMU (Memory Management Unit) il quale compie il mapping run-time tra questi due indirizzi. L'architettura x86 utilizza un TLB (Translation Lookaside Buffer) per memorizzare



temporaneamente le traduzioni di pagine frequentemente utilizzate. Questo accelera l'accesso alla memoria virtuale riducendo la necessità di accedere alle tabelle delle pagine in memoria.



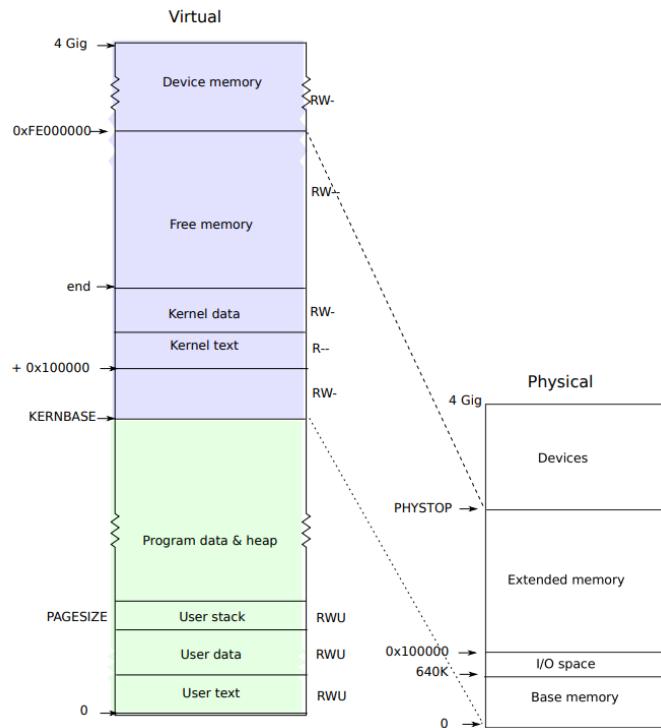
I processi utente sono creati grazie alla system call *fork*. Una volta che il processo figlio è allocato, la sua immagine di memoria viene impostata come *copia completa* di quella del processo padre grazie alla chiamata a *copyuvvm()*. Quindi NON si fa ricorso alla **tecnica Copy-on-write (COW)** la quale ricorre alla *condivisione delle pagine tra più processi*.

#### Virtual Address Space

In xv6 lo **spazio di indirizzamento virtuale** (virtual address space) di ogni processo è di 4 GB ed è diviso in due porzioni, una dedicata allo *user* e l'altra dedicata al *kernel*, ognuna delle quali occupa 2 GB di memoria: lo **user space** occupa uno spazio di memoria che va dall'indirizzo virtuale zero 0x00000000 fino all'indirizzo definito da *KERNBASE* (come anticipato pari a 2 GB, 0x80000000), mentre il **kernel space** occupa uno spazio di memoria che va da *KERNBASE* a 0xFFFFFFFF (4 GB).

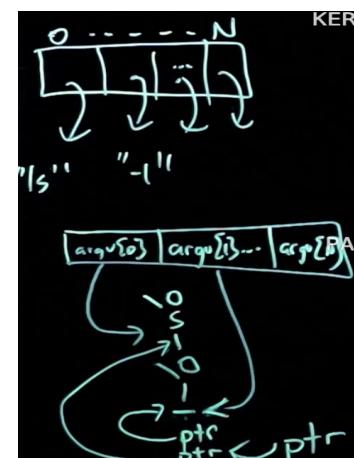
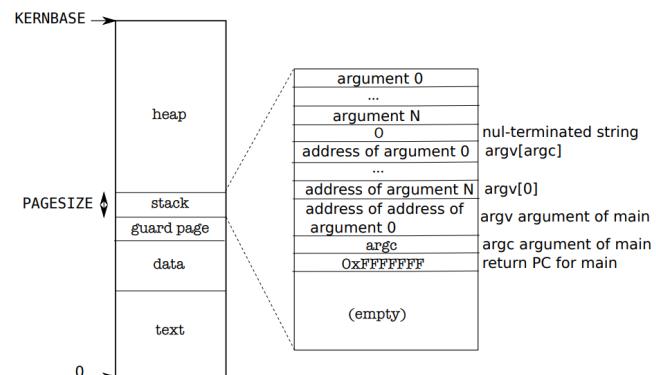
Lo spazio di indirizzamento virtuale di un processo *kernel*, contenente codice e dati, inizia dall'indirizzo di memoria definito da *KERNBASE* ed occupa uno spazio di memoria massimo definito da *PHYSTOP*, il cui valore è di 2 GB (0xE000000, definito in *memlayout.h*). Quindi lo spazio di indirizzamento virtuale di tale processo assume il range [*KERNBASE*, *KERNBASE+PHYSTOP*] ed è mappato nella memoria fisica nel range [0, *PHYSTOP*].

È possibile ridefinire la dimensione delle due porzioni cambiando tali costanti in *memlayout.h*.



### Processo:

- **Kernel mode:** il kernel include il BIOS; viene caricato lo spazio di indirizzamento fisico nella parte superiore dello spazio di indirizzamento virtuale, utile per il mapping tra dispositivi, l'allocazione di memoria libera (extended memory). Il kernel è posizionato alla stessa posizione per ogni processo: ciò permette di facilitare scrittura e lettura dalla memoria fisica.
- **User mode:** 4 KB di stack e memoria dinamica per dati e heap dei programmi. Lo stack ha una guard page per prevenire Segmentation Fault, di scrivere su pagine protette. Nello stack si ha un vettore di argomenti contenente ciò che viene scritto su linea di comando (ogni parola, divisa dalle altre da uno spazio, viene messa in una cella del vettore): di questo viene allocato l'indirizzo ed il contenuto di ogni cella.



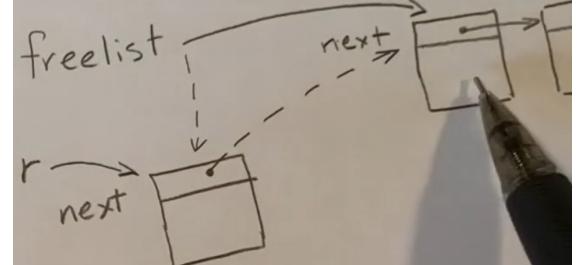
[Esempio di argomenti in stack con valore e puntatore]

## Implementazione

### Page Table e MMU

Per gestire l'**albero delle page table** e l'**allocazione/deallocazione di una pagina**, si ricorre alle seguenti funzioni:

- in *vm.c*:
  - *walkpgdir(pgd, va, alloc)*: attraversa l'albero delle page table (passando il parametro *pgd* la quale indica la page table all'indirizzo virtuale *va*) ritornando la corrispondente entry, altrimenti un puntatore NULL. È possibile creare page table indicandolo nel valore da passare come terzo parametro della funzione. Restituisce la traduzione dell'indirizzo che fa l'MMU.
  - *mappages(pgd, va, size, pa, perms)*: aggiunge entry in una page table di un processo o del kernel; per compiere questa operazione richiama la funzione *walkpgdir()*. Per il mapping bisogna dare i permessi necessari, come quelli di lettura, scrittura, esecuzione ed accessibilità da parte dell'utente.
  - *kvmalloc()*: utile per allocare la page table per il kernel address space.
  - *freevm()*: libera tutte le pagine di una page table ed i corrispondenti data page. La cancellazione delle pagine avviene in maniera ricorsiva, chiamando la funzione *deallocuvm(pgd, KERNBASE, 0)* e ricorrendo anche alla funzione *walkpgdir()*.
- in *kalloc.c*:
  - è presente una *struct kmem* contenente una lista di pagine libere (lista linkata) ed uno spinlock; in *end[]* è memorizzato il primo indirizzo usabile.
  - *kalloc()*: utile per allocare una pagina. La funzione acquisisce lo spinlock della *struct kmem* (*kmem.lock*) e ritorna il puntatore alla lista di pagine libere (*struct run \*r; r = kmem.freelist;*). Se il puntatore non è nullo (*if(r)*) allora si assegna al puntatore della lista delle pagine libere l'indirizzo della pagina successiva a quella appena allocata (*kmem.freelist = r->next;*). Alla fine lo spinlock viene rilasciato. Funzione tipicamente invocata da *allocuvm()* (in *vm.c*) per allocare pagine ad uno spazio di indirizzamento virtuale esistente.
  - *kfree(p)*: accetta come argomento la pagina da liberare; si riempie quest'ultima con valori *junk* grazie alla funzione *memset()*. Quindi si aggiunge tale pagina alla lista delle pagine libere.



## OS161

### Virtual Memory: Allocazione e deallocazione pagine

OS161 supporta il **paging** ed il **Demand Paging**. La dimensione della pagina è specificata dalla costante *PAGE\_SIZE* definita in *vm.h* e tipicamente ha valore pari a 4 KB, come in xv6.

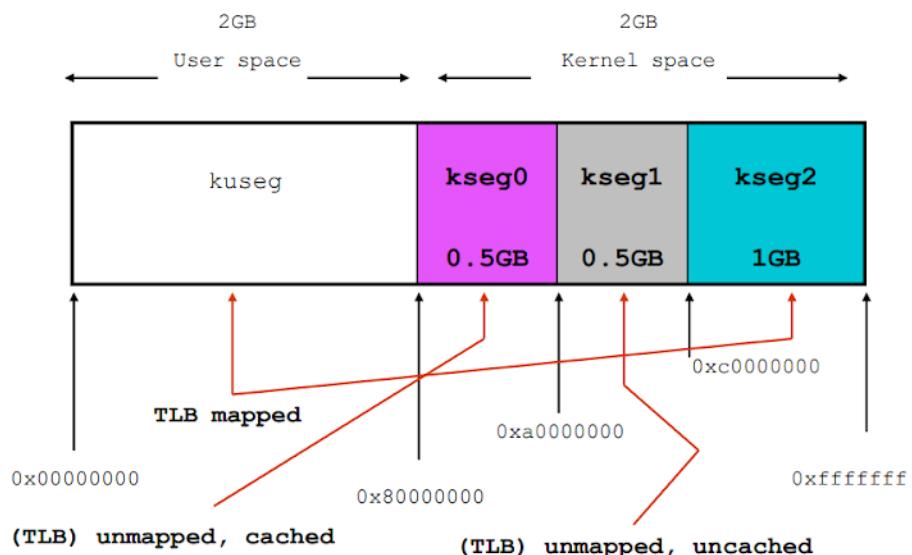
OS161, nella *versione base* contiene un gestore di memoria virtuale che effettua unicamente *allocazione contigua* di memoria reale, senza mai rilasciarla (vedi kern/arch/mips/vm/dumbvm.c).

Per l'allocazione di pagine sia uno *user thread* che un *kernel thread* fanno riferimento alla funzione **getppages()** la quale invoca la funzione **ram\_stalmem()** (entrambe in kern/arch/mips/vm/ram.c) per ottenere il primo indirizzo utile per un determinato numero di pagine libere, quest'ultimo passato come parametro.

Logical addr. (KSEG0)	Physical addr.
0x80000000	0x0
0x80000200	0x200
0x80039d54 (_end)	0x39d54
0x8003a000 (P)	0x3a000
0x8003b000 (firstfree)	0x3b000 <b>(firstpaddr)</b>
0x80100000	0x100000

Virtual address space

Lo spazio di indirizzamento virtuale è diviso in **kernel space** e in **user space** (kuseg). Il **kernel space** è diviso in tre porzioni: **kseg0** (0.5 GB non memorizzati nel TLB, ma vengono messi nella cache, utile per *memorizzare informazioni sul kernel*), **kseg1** (0.5 GB, non memorizzati nel TLB né nella cache, utile per il *mapping I/O dei dispositivi*) e **kseg2** (1 GB, non usato in OS161).



Struttura simile a quella di xv6 in quanto lo user space è identico ed i segmenti del kernel space sono simili in quanto rivestono gli stessi ruoli: contengono mapping con dispositivi I/O, informazioni su dati e codice del kernel e memoria disponibile per estendere quella già a disposizione.

Dato un indirizzo logico *addr* il corrispondente indirizzo fisico è dato dalla sottrazione tra l'indirizzo logico e l'indirizzo di KSEG0.

## Algoritmi di scheduling

Lo **scheduling** rappresenta la gestione dei processi in attesa di esecuzione su un calcolatore ad opera di un componente del sistema operativo, detto **scheduler**. Ci si basa su numero e disponibilità di CPU/core e sulla politica che viene adottata per pescare il processo da eseguire. Utile solo per il parallelismo computazionale e non di dato. L'obiettivo è quello di massimizzare l'utilizzo delle risorse, in particolare della CPU.

Il **context-switching** indica una particolare operazione del sistema operativo che conserva lo stato del processo o thread, in modo da poter essere ripreso in un altro momento.

I **processi** rappresentano programmi completi ed indipendenti, mentre i **thread** rappresentano differenti parti dello stesso programma. Più *thread* dello stesso *processo* condividono codice, variabili (non locali), risorse del SO.

Quando avviene uno scambio tra il thread in esecuzione ed un altro, si ha la necessità di memorizzare i registri ed i dati del thread in esecuzione e di caricare i dati del thread da eseguire dal suo stack. Quando si mette un thread in pausa e si riprende l'esecuzione di un altro si ha il **context-switching**, il quale cambia il contesto (memoria e stato del thread/processo al momento del "cambio") da un processo ad un altro e segue la politica adottata dallo scheduler.

Dati ed informazioni dei registri vengono salvati in particolari strutture dati, tipicamente nelle **switchframe**, utili per operazioni di **context-switching** (a differenza dei **trapframe** utili per chiamate a system call e che necessitano anche di variabili temporanee). Quando si riprende l'esecuzione del thread sospeso, si ripristina il suo context dallo switchframe.

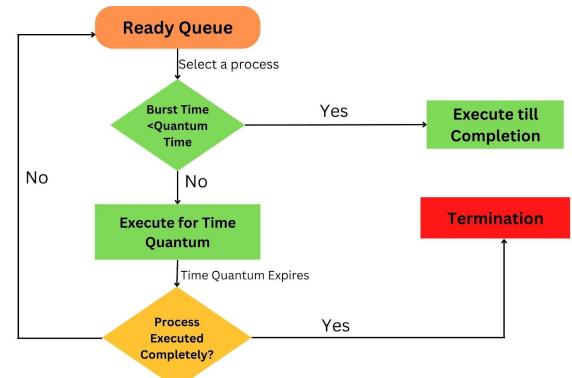
xv6

### Policy

In xv6 i processi vengono schedulati secondo lo **Strawman Scheduling** il quale rappresenta una variante del *Round-Robin*, cioè una policy per la quale viene dedicato ad ogni processo un *intervallo di tempo* di esecuzione (time slice) uguale per tutti. Per questo approccio i processi sono organizzati in una lista, condivisa tra tutti i core e gestita con modalità FIFO; ogni volta si cerca il primo processo in stato di **RUNNABLE** ed appena lo si trova questo viene mandato in esecuzione dallo scheduler, mentre il task sospeso viene inserito alla fine della lista. Questo approccio non supporta e non segue particolari ordini di priorità.

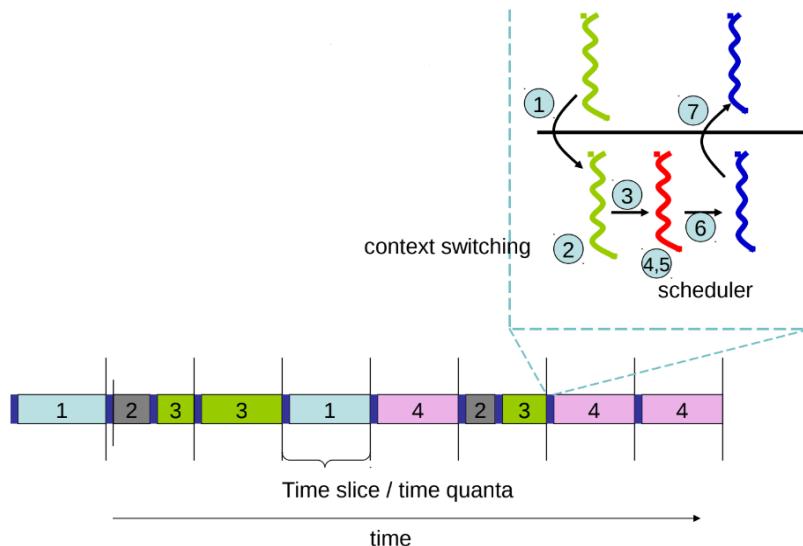
L'*intervallo di tempo* si aggira tipicamente su 1M di colpi di clock (circa 100 ms). xv6 usa **timer interrupt** per forzare l'arresto del processo dopo tale intervallo di tempo. In questa maniera si ha l'illusione che ogni processo sia associato ad un core ed ad una porzione di memoria.

Un aspetto da tenere d'occhio è il **context-switch time** il quale rappresenta il tempo speso per lo scambio tra due processi, intervallo di tempo per mandarne uno in esecuzione e l'altro in attesa. Durante questo intervallo di tempo si ha uno scambio di dati tra i processi ed avviene lo scambio tra il processo in esecuzione ed il processo in attesa.



Pregi: tutti i processi hanno la stessa probabilità di essere eseguiti sulla CPU; si ha un tempo di risposta breve.

Difetti: il tempo di attesa medio per l'esecuzione di un processo è relativamente lungo; le prestazioni sono inversamente proporzionali al quanto di tempo definito (al diminuire del quanto di tempo si ha un aumento delle prestazioni e vale anche il contrario); si ha un incremento dei context-switching e di conseguenza si spende più tempo nel cambio di contesti e di processi e ciò comporta un overhead.



### Implementazione

Il file sorgente **proc.c** contiene lo scheduler che esegue lo *scheduling* e il *context switching* tra i processi.

Sostanzialmente nella funzione ***scheduler()*** (in proc.c) si legge in maniera ciclica (costrutto **for** interno) la lista dei processi contenuta nella **struct ptable**, ***ptable.proc[]***, ed appena si incontra un processo in stato **RUNNABLE** lo si esegue, portandolo nello stato **RUNNING** e si esegue la funzione ***swtch()*** dallo scheduler al processo user. In poche parole avviene lo "scambio" tra lo scheduler e il kernel thread associato al processo ed il viceversa al ritorno.

La ***ptable.proc[]*** è condivisa tra tutte le CPU. I multi-scheduler threads necessitano di avere un lock per poter leggere tale tabella, quindi si acquisisce lo spinlock ***ptable.lock*** con ***acquire(&ptable.lock)*** e lo si rilascia con ***release(&ptable.lock)*** alla fine del ciclo sulla lista dei processi; la funzione ***swtch()*** necessita l'acquisizione del lock da tale processo.

***scheduler()*** abilita le *interruzioni* con ***sti()***, mentre le disabilita con la funzione di acquisizione del lock di ***ptable.lock*** e le riabilita con la funzione di rilascio dello stesso spinlock; in questa maniera lo scheduler "accetta" le interruzioni per un breve intervallo di tempo, permettendo che i processi possano andare dallo stato di **SLEEPING** allo stato di **RUNNABLE** prima di essere analizzate dal ciclo **for** interno.

Nel **caso particolare** in cui avessimo un singolo thread **RUNNABLE** nella CPU si compie lo switch da questo allo scheduler e poi dallo scheduler a questo processo, non trovando altri processi in stato **RUNNABLE**.

```

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;
            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-switching

Per compiere uno *switch* tra processi, xv6 attua due tipologie di ***context-switching*** a basso livello: dal kernel thread del processo utente allo scheduler della CPU corrente e viceversa.

Il passaggio da user mode a kernel mode del thread avviene come risultato di una **trap**: questa potrebbe essere un'interruzione o una system call dallo user thread. La **trapret** o *trap return* compie il percorso inverso, cioè dal kernel mode allo user mode.

Lo user thread presenta delle *trap temporizzate* determinate da timer interrupt che si presentano dopo che è trascorso un quanto di tempo. Il tutto è gestito dallo scheduler grazie alla funzione *sched()*.

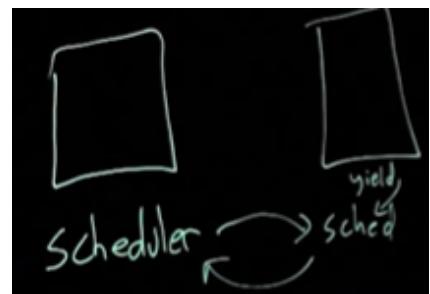
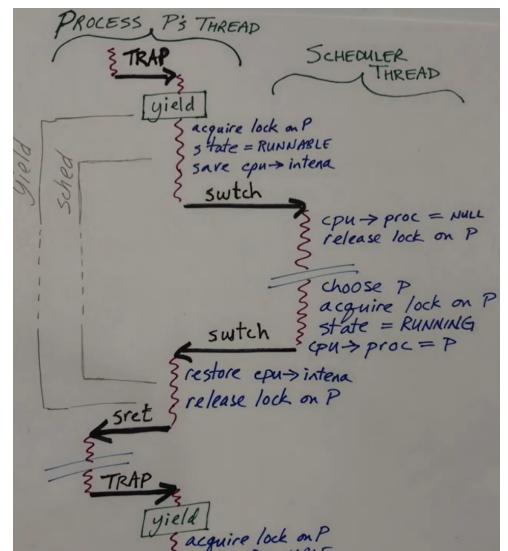
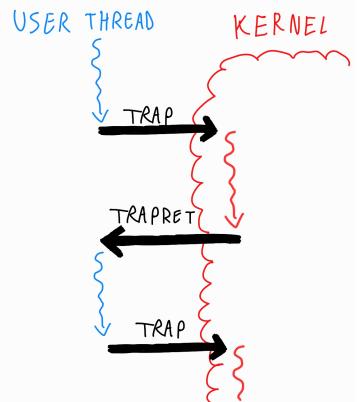
La **trap** viene analizzata dalla funzione *trap()* in trap.c, mentre **trapret** viene svolta da *trapret()* in traspasm.S.

Quando si presenta una **trap** si passa dallo user thread al kernel thread associato al processo user, il quale chiama la funzione *yield()* (in proc.c) la quale forza il processore a rilasciare il controllo del thread in esecuzione e di mandarlo alla fine della "Ready Queue", lista dei processi/thread da eseguire, dove ognuno di essi ha la stessa priorità di schedulazione. Questa funzione necessita l'acquisizione del **ptable.lock** per poter cambiare lo stato del processo corrente in RUNNABLE e chiamare la funzione *sched()* (in proc.c) affinché avvenga il passaggio dal kernel thread del processo utente appena sospeso allo scheduler thread, invocando la funzione *swtch()*. Lo *scheduler()* (in proc.c) acquisisce lo spinlock e, come detto, mette in esecuzione un processo in stato RUNNABLE preso dalla lista dei processi e poi viene invocata la *swtch()* affinché avvenga lo scambio tra scheduler thread e kernel thread del processo messo in esecuzione. Al ritorno da questa funzione vengono completate le ultime istruzioni della *sched()* e si ritorna allo user mode del processo con una **trapret**, ovvero *trap return*.

*scheduler()* e *sched()* sono co-routines e si ha quindi una *cooperazione* tra di loro: una volta che *sched()* acquisisce il lock, grazie a *yield()*, si ha lo switch a *scheduler()* e quest'ultimo acquisisce il lock e compie lo switch a *sched()* che lo rilascia tramite *yield()*.

La funzione *swtch()* (in swtch.S) salva i registri correnti e carica i registri salvati nel kernel thread del processo/scheduler (proc->context o mycpu()->scheduler, rappresenta lo *switchframe*) nei registri hardware del x86, inclusi stack pointer ed instruction pointer e carica i registri dello scheduler/processo. *swtch()* accetta due parametri: struct context \*\*old e struct context \*new; del primo si salva il contesto, mentre del secondo si caricano i suoi registri. La funzione viene chiamata due volte: nelle funzioni *sched()* e *scheduler()*, rispettivamente si passa dal kernel thread del processo utente a quello dello scheduler (*swtch(&p->context,mycpu()->scheduler)*) ed il viceversa (*swtch(&(c->scheduler),p->context)*).

Regola: non si possono avere lock quando si attua l'operazione di *yield* sulla CPU, ad eccezione del **ptable.lock**.



## OS161

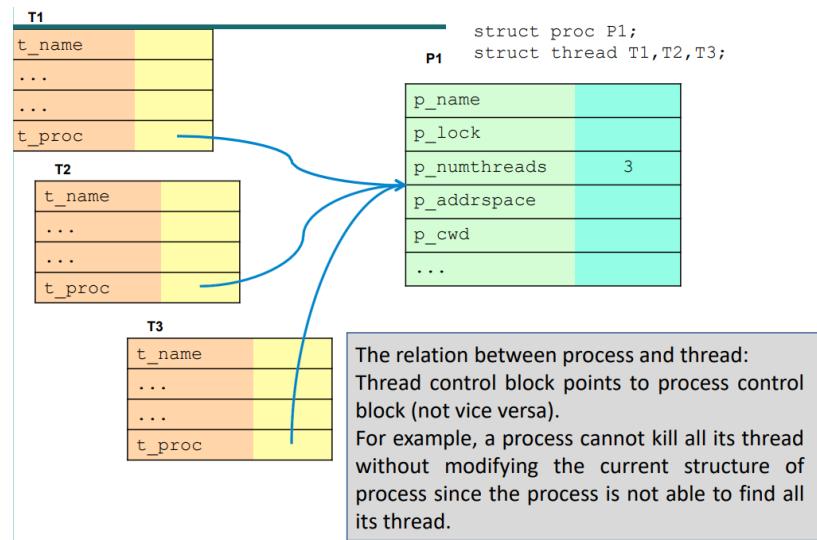
La politica di scheduling adottata da OS161 è quella del **Preemptive Round Robin**, ovvero di una politica di scheduling con prelazione (lo scheduler interrompe temporaneamente l'esecuzione del task per eseguirne un altro) per la quale ogni processo viene eseguito per un determinato intervallo di tempo dalla CPU: tipicamente tale intervallo di esecuzione viene detto scheduling quantum e rappresenta un multiplo del tick del processo, dura all'incirca 1 ms (mentre i context-switching che occorrono per cambiare processo impiegano 20 ms).

A differenza di xv6, OS161 supporta il multi-threading, quindi può compiere scambi tra un thread in esecuzione ed un altro thread dello stesso processo. In xv6 non si ha un'implementazione per i thread e non ci sono campi appositi nella struct proc. È supportato solo il single-threading.

Analogamente a quanto accade in xv6, si ha una funzione di *yield* (*thread\_yield()*) che invoca la funzione di schedulazione (*thread\_switch*) che ne chiama una a basso livello (*switchframe\_switch()*) la quale salva il *thread context*, contenenti valori di registri ed altro.

## Processi e thread

Su **OS161** è supportato il *multi-threading*: un processo può avere uno o più thread.  
Unità minima di elaborazione: thread.



Mentre in **xv6** 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. È supportato solo il *single-threading*.

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)
};
```

Per questo motivo l'allocazione di stack e l'assegnazione di campi avviene su due livelli diversi nei due sistemi operativi: a livello di processo su **xv6** ed a livello di thread su **OS161**.

## Sezione 2

### Introduzione

#### Installazione del sistema operativo

In generale è necessario emulare il sistema operativo su uno di tipo Linux, quindi se si lavora su Ubuntu si può lavorare richiamando direttamente l'emulatore QEMU, altrimenti su Windows e su MacOS bisogna usare un layer di compatibilità basato su un sistema operativo Linux per poter emulare xv6. Su Windows si utilizza WSL e per l'installazione e l'emulazione del sistema operativo si usano le stesse istruzioni e lo stesso emulatore QEMU. Non è possibile creare una macchina virtuale Ubuntu e poi utilizzare l'emulatore per usare il sistema operativo xv6, poiché si tratterebbe di virtualizzare più volte un sistema operativo su un computer. In sostanza l'emulazione consente ad un sistema operativo di imitarne un altro.

**QEMU:** un software che implementa un particolare sistema di emulazione che permette di ottenere un'architettura informatica nuova e disgiunta in un'altra che si occuperà di ospitarla.

Una volta eseguite le istruzioni riportate nel seguente paragrafo, apparirà una finestra separata contenente la visualizzazione della macchina virtuale. Dopo qualche secondo, il BIOS virtuale di QEMU caricherà il boot loader di xv6 da un'immagine virtuale del disco rigido contenuta nel file xv6.img, che a sua volta caricherà ed eseguirà il kernel xv6. Dopo che tutto è stato caricato, si dovrebbe ottenere un prompt "\$" nella finestra di visualizzazione di xv6 e si dovrebbe essere in grado di inserire comandi nella shell di xv6.

#### Istruzioni

```
sudo apt update  
sudo apt upgrade  
sudo apt-get install qemu  
sudo apt install qemu-system-x86  
sudo apt-get install libc6-dev-i386  
git clone https://github.com/mit-pdos/xv6-public.git xv6-public  
chmod 700 -R xv6-public  
cd xv6-public  
make clean (compilazione e pulizia del sistema operativo per rimuovere file non creati correttamente)  
make (compilazione sistema operativo)  
make qemu (altrimenti, se si vuole operare da linea di comando sulla stessa shell: make qemu-nox; compilazione ed esecuzione sistema operativo)
```

#### Esecuzione codice e debug

Il modo più semplice per eseguire il debug di un programma in QEMU è utilizzare la funzione di debug remoto di **GDB** e lo stub di debug GDB remoto di QEMU. Il debug remoto è una tecnica molto importante: l'idea di base è che il debugger principale (GDB in questo caso) venga eseguito separatamente dal programma da debuggere (potrebbero trovarsi su macchine completamente separate); funziona in modalità di ascolto su porta TCP. In questo

caso, un piccolo stub di debug remoto è tipicamente incorporato nel programma in fase di debug; lo stub di debug remoto implementa un semplice linguaggio di comandi che il debugger principale utilizza per ispezionare e modificare la memoria del programma di destinazione, impostare punti di interruzione, avviare e interrompere l'esecuzione, ecc. Quando si esegue il debug remoto, bisogna assicurarsi sempre che l'immagine del programma fornita a GDB sia esattamente la stessa dell'immagine del programma in esecuzione sul target di debug: se non sono sincronizzate per qualsiasi motivo (ad esempio, perché si è modificato e ricompilato il programma e si è riavviato QEMU senza riavviare anche GDB con la nuova immagine), gli indirizzi dei simboli, i numeri di riga e altre informazioni fornite da GDB non avranno alcun senso. Una volta che GDB si è connesso con successo allo stub di debug remoto di QEMU, recupera e visualizza le informazioni sul punto in cui il programma remoto si è fermato. Come già detto, lo stub di debug remoto di QEMU arresta la macchina virtuale prima che esegua la prima istruzione.

### Istruzioni

#### *Prima shell*

*make qemu-nox-gdb* (o *make qemu-gdb*)

#### *Seconda shell*

*gdb* (se già installato)

*target remote :TCP-Port* (o *target remote localhost:TCP-Port*; tipicamente si usa la porta 26000)

```
ivan@IVAN-PC:~/xv6/xv6$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tpl > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S
-gdb tcp::26000
xv6...
```

*file kernel* (utile per caricare correttamente l'immagine dell'OS)

I comandi utili ad eseguire il programma in modalità debug sono:

- *break (b) x* (breakpoint alla funzione x)
  - *b x:y* (breakpoint alla funzione x linea y)
- *disable breakpoint x* (disabilita il breakpoint con identificativo x, numero intero da 1 in su)
- *step (s)* (utile per entrare dentro la funzione nella sua esecuzione)
- *next (n)* (procedere all'istruzione successiva)
- *continue (c)* (continua fino alla fine o fino ad una richiesta di input)
- *CTRL + C* (se si sta continuando con l'esecuzione del sistema operativo, con questo comando il debug si ferma alla linea di codice corrente; torna utile per rimettere dei breakpoint e visualizzare l'andamento).

Character	Command
b	breakpoint set a place where GDB stops executing your program
c	continue program execution
l	list source code
n	execute the next line of code, then stop
P	print the value of a variable or expression
q	quit GDB
r	run your program from the beginning
s	step to the next line of code
x	examine memory

## Implementazione di System Call

Per questa parte del progetto abbiamo deciso di sviluppare due system calls, le quali mostrano informazioni sui processi attivi. Ci siamo ispirati a due comandi usati su terminali Linux, ovvero a **ps** ed a **pstree**: il primo fornisce informazioni sui processi attivi (PID, nome del processo, dimensione del processo in memoria) ed il secondo mostra le relazioni di parentela tra i processi.

### Realizzazione

Per realizzare una system call in **xv6**:

1. syscall.h contiene una mappatura tra il nome della system call e il numero con cui viene identificata. È necessario aggiungere a queste mappature la nuova system call, assegnando a questa (*SYS\_name*) un intero positivo diverso da quelli già usati.
2. syscall.c contiene funzioni di aiuto per analizzare gli argomenti delle system call e puntatori alle implementazioni delle system call. In corrispondenza di (\**syscalls*[]), bisogna associare alla costante definita nel punto precedente la funzione che la esegue, denominata *sys\_name*.  
Inoltre bisogna renderla disponibile all'esterno di tale file sorgente, quindi prima la si riscrive anticipando a questa la voce “extern” e scrivendo il tipo di parametri ed il tipo di ritorno.
3. sysproc.c contiene le implementazioni delle system call relative ai processi. Qui si aggiungerà il codice delle system call *sys\_name()* che ritorna un intero e che di solito contiene una funzione *name()* che deve risiedere nel file *proc.c*.
4. in proc.c, definire nel file sorgente citato nel punto precedente la funzione *name()* e riportare il suo prototipo nei file header defs.h (file header al quale si riferisce *proc.c*) e user.h (file header al quale si riferisce *sysproc.c*) dove sono definite le altre funzioni per le system call.  
Non si può avere un unico file header perché *defs.h* e *user.h* condividono funzioni che hanno parametri o tipi di ritorno diversi.
5. usys.S contiene un elenco di system call esportate dal kernel. Al suo interno bisogna aggiungere una voce *SYSCALL(name)*, dove al posto di *name* si riporta il nome della funzione precedentemente definita in *proc.c*.
6. affinché tale system call possa essere invocata da linea di comando, bisogna creare un apposito file sorgente name.c contenente la funzione main e che invochi tale funzione. Affinché *name.c* possa invocarla questo deve includere la libreria *user.h*, dove abbiamo aggiunto l'intestazione della funzione *name()*; oltre a questa libreria si aggiungono le librerie *types.h* e *stat.h* e poi quelle necessarie al file.
7. in Makefile, si riporta il nome del file sorgente affinché possa essere richiamato da linea di comando: si aggiunge il suo nome alla voce *UPROGS*, formattata come gli altri file, ed alla voce *EXTRA*, formattata come gli altri file.
8. a questo punto si compila e si esegue il kernel con i soliti comandi (make clean - make qemu): si può notare, digitando il comando “ls”, che nella lista dei comandi/eseguibili è presente la funzione implementata.

## ps

Il comando **ps** (abbreviazione di “Process Status”) serve per elencare i processi attivi con le informazioni rilevanti, quali PID, nome, dimensione in memoria, ecc. Nella nostra implementazione abbiamo usato solo i primi tre campi citati.

Questa system call possiede due funzionalità:

- se non vengono passati argomenti oltre al comando *ps*, allora questa system call stamperà a video le informazioni relative a tutti i processi attivi;
- se viene passato come argomento aggiuntivo un intero, allora stamperà a video le informazioni relative al processo avente un PID uguale al parametro passato. Se non esiste un processo attivo avente come PID il valore di tale argomento, allora stamperà un errore.

Per entrambe le implementazioni si fa riferimento alla funzione *getprocinfo()* implementata in *proc.c* la quale restituisce alla funzione chiamante (funzione *main()* in *ps.c*) l’elenco dei processi attivi con le informazioni significative (registerate nella struttura dati *struct pstat* in *processInfo.h*, passata alla funzione per riferimento). In base al numero di argomenti si decide cosa fare con tale lista:

- in assenza di argomenti si stampano i processi attivi con le loro caratteristiche (PID, nome, dimensione in Byte).

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

- in presenza di un argomento si verifica se il numero passato come argomento rappresenta il PID di un processo attivo:
  - se è presente tale match allora si stampa il processo con le sue caratteristiche.

\$ ps 1			
PID	Name	Size	
1	init	12288	

- se non è presente allora viene stampato un errore.

\$ ps			
PID	Name	Size	
1	init	12288	
2	sh	16384	
3	ps	12288	

\$ ps 3			
Error: No Process for this PID			

\$ ps			
PID	Name	Size	
1	init	12288	
2	sh	16384	
5	ps	12288	

## pstree

Il comando **pstree** serve per costruire l’albero dei processi, in base alla parentela tra di essi. Come informazioni riportiamo il PID ed il nome di ogni processo.

Per implementare questa funzione facciamo riferimento al campo *parent* della struct proc la quale ci restituisce la struttura dati processo padre grazie alla quale risalire al suo PID. Per costruire l'albero ricorriamo ad una funzione *ricorsiva* che prende come parametri il vettore dei processi, il PID padre che si sta considerando in quell'istante, un vettore dei processi per memorizzare i processi intermedi di quel ramo e l'indice di quest'ultimo; grazie a questa funzione si riesce a compiere correttamente la stampa dell'albero.

La funzione da chiamare è *getproctree()* e la funzione ricorsiva che viene chiamata è la *walk()*. Il problema di questa soluzione è che questo comando porta ad un esaurimento della memoria dinamica per l'esecuzione dei comandi successivi.

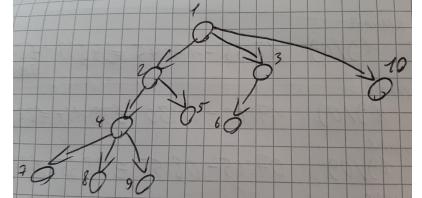
```
init: starting sh
$ sh ; ps
$ ptree
[PID: 1 Name: init]    ->      [PID: 2 Name: sh]    ->      [PID: 3 Name: sh]    ->      [PID: 4 Name: sh]    ->      [PID: 5 Name: ptree]
$ sh ; ptree
allocvum out of memory
lapicid 0: panic: kfree
80102555 80106bee 801070ca 80103adf 80104de9 80105e6d 80105c44 0 0
```

Un'altra soluzione valida è di tipo *iterativo*: si calcola prima il vettore dei padri, scandendo da destra verso sinistra la lista dei processi della page table (si può notare che se pid1<pid2 allora il processo pid2 non può essere padre del processo pid1), e poi, sulla base di questo, si stampano tutti i percorsi, non soltanto quelli dal nodo foglia.

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

Purtroppo con xv6 non abbiamo avuto modo di vedere come funziona la system call nel caso in cui un processo dovesse avere più figli, ma sosteniamo che entrambi gli approcci siano giusti (di quello *ricorsivo* ci sono esempi nel codice effettuati su vettore di interi, rappresentante il vettore dei padri; chiama la funzione *walk\_prova()*).

```
init: starting sh
$ ptree
[PID: 1]    ->      [PID: 2]    ->      [PID: 4]    ->      [PID: 7]
[PID: 1]    ->      [PID: 2]    ->      [PID: 4]    ->      [PID: 8]
[PID: 1]    ->      [PID: 2]    ->      [PID: 4]    ->      [PID: 9]
[PID: 1]    ->      [PID: 2]    ->      [PID: 5]    ->
[PID: 1]    ->      [PID: 3]    ->      [PID: 6]
[PID: 1]    ->      [PID: 10]
```



## Debug della system call

Durante il debug delle system call abbiamo potuto constatare quanto affermato nella sezione n.1, ovvero che alla chiamata di una system call viene generato un interrupt *trap* dove viene memorizzata l'azione richiesta: tale informazione è salvata nel *trapframe*, in particolare in *tf->trapno*, dove bisogna verificare se fa riferimento ad una system call o meno. Come si può notare dallo screenshot sottostante si ha a sinistra una shell che emula in modalità debug il sistema operativo ed a destra una shell che compie il debug del sistema operativo tramite il compilatore *gdb*. Alla chiamata della system call “*ps 1*” si può notare, oltre alle trap temporizzate (riconoscibili dal fatto che in *trap.c* saltano il costrutto *if(tf->trapno == T\_SYSCALL)*) ed entrano nel costrutto *switch(tf->trapno)*), una trap che entra nel costrutto *if(tf->trapno == T\_SYSCALL)* ed esegue le varie istruzioni fino ad entrare ed eseguire la funzione *syscall()* in *syscall.c*, all'interno della quale si controlla l'identificativo della system call e si esegue quella invocata.

```

ivan@IVAN-PC:~/xv6/xv6$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmp > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,for-
mat=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S
-gdb tcp::26000
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
start 58
init: starting sh
$ ps 1
|
```

```

49      switch(tf->trapno){
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, trap (tf=0x80115418 <stack+3912>) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) n
49      switch(tf->trapno){
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, trap (tf=0x80115418 <stack+3912>) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) Quit
(gdb) Quit
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, trap (tf=0x80115418 <stack+3912>) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, trap (tf=0x8dffefb4) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) n
40          if(myproc()>killed)
(gdb) n
42              myproc()>tf = tf;
(gdb) n
43                  syscall();
(gdb) s
0x80104bc:    jne    syscall () at syscall.c:138
139      struct proc *curproc = myproc();
(gdb) n
141      num = curproc->tf->eax;
(gdb) n
142      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) n
|
```

Inoltre possiamo verificare nel prossimo screenshot che, dopo aver rilevato l'interrupt di tipo system call nella funzione *trap()* in *trap.c* e dopo alcune trap temporizzate, si entra prima nella funzione della system call in *sysproc.c* e poi nella funzione invocata da quest'ultima, la quale risiede in *proc.c*: nell'esempio si vede che il comando *ps* invoca la *sys\_getprocinfo()* (in *sysproc.c*) la quale chiama la funzione *getprocinfo()* (in *proc.c*).

```

ivan@IVAN-PC:~/xv6/xv6$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmp > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,for-
mat=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S
-gdb tcp::26000
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
start 58
init: starting sh
$ ps 1
|
```

```

Continuing.

Thread 1 hit Breakpoint 3, trap (tf=0x8df23f28) at trap.c:39
39      if(tf->trapno == T_SYSCALL){
(gdb) 
Continuing.

Thread 1 hit Breakpoint 4, sys_getprocinfo() at sysproc.c:98
98      if (argptr[0], (char **)d, sizeof(struct pstat)) < 0
(gdb) l
93
94      int
95      sys_getprocinfo(void)
96      {
97          struct pstat *d;
98          if (argptr[0], (char **)d, sizeof(struct pstat)) < 0)
99              return -1;
100          getprocinfo(d);
101          return 0;
102      }
(gdb) c
Continuing.

Thread 1 hit Breakpoint 5, getprocinfo (ps=0x29bc) at proc.c:506
506      acquire(&ptable.lock);
(gdb) L
501
502      int
503          getprocinfo(struct pstat* ps){
504              int i, j;
505              struct proc *p;
506
507              acquire(&ptable.lock);
508              for(i=0, j=0, p = ptable.proc; p < &ptable.proc[NPROC]; p++, i++) // p
509                  if(p->state != UNUSED)
510                      ps->pid[i] = p->pid;
(gdb)
|
```