

# ANALISI COMPARATIVA tra OS161e Xv6

Realizzato da:

Scaturro Andrea Pio: [s319480@studenti.polito.it](mailto:s319480@studenti.polito.it)

Interrante Bonadia Alekos: [s319849@studenti.polito.it](mailto:s319849@studenti.polito.it)

# XV6

Xv6 è un S.O. basato su UNIX V6, progettato dal MIT a scopo didattico e scritto quasi interamente in C.

Architettura: RISC-V

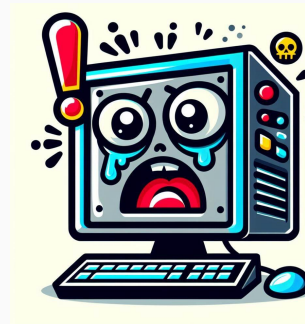


# Kernel

- **xv6** è un sistema operativo basato su un kernel monolitico

L'intero sistema opera con privilegi hardware completi, semplificando la cooperazione tra le sue parti.

Questa struttura può portare a complessità nelle interfacce, aumentando il rischio di errori che portano al blocco del sistema.



# Kernel

Al contrario, i microkernel come **os161** minimizzano le operazioni in modalità supervisore

Mantenendo la maggior parte del sistema operativo a livello utente per una maggiore semplicità e riduzione degli errori.

Sebbene xv6 sia concettualmente monolitico, è più piccolo di molti microkernel a causa dei pochi servizi offerti.



xv6

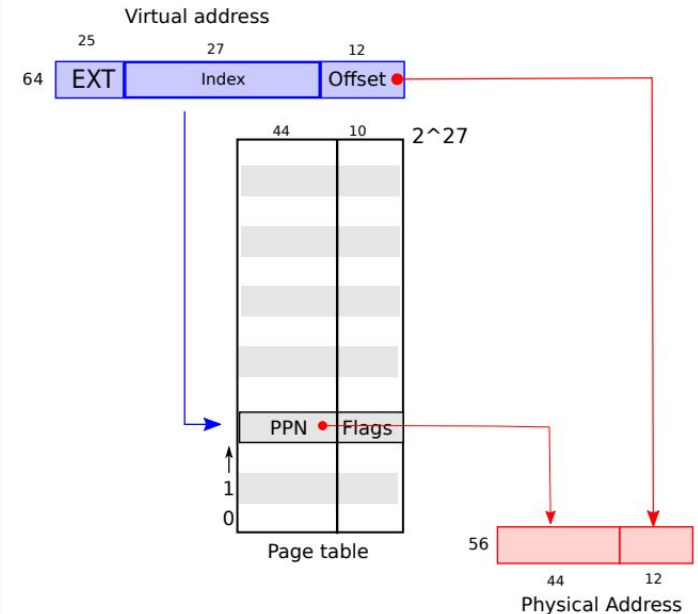


OS161

# Gestione della memoria

## Gestione della Memoria su Xv6 con RISC-V

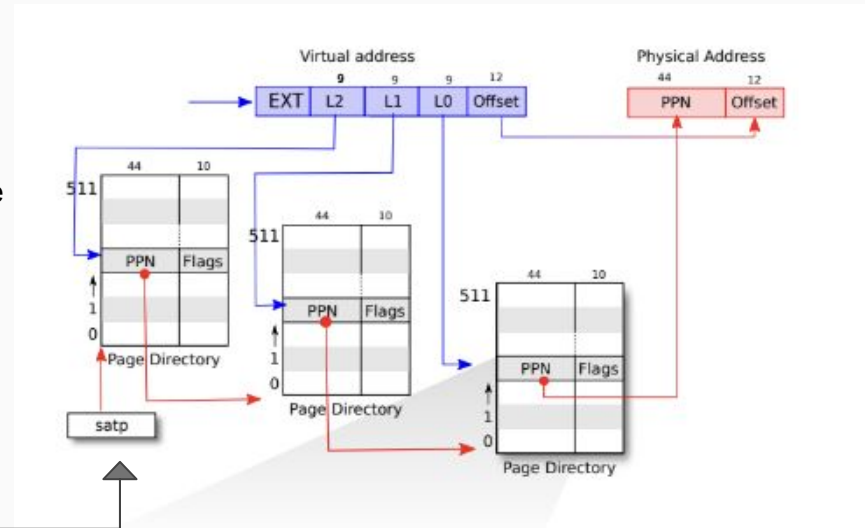
- Le istruzioni RISC-V manipolano indirizzi virtuali, mentre la RAM è indicizzata con indirizzi fisici.
- Tabella delle pagine RISC-V collega indirizzi virtuali e fisici, mappando ogni indirizzo virtuale su uno fisico.
- Configurazione Sv39:**
  - Xv6 utilizza Sv39 RISC-V, con gli ultimi 39 bit di un indirizzo virtuale a 64 bit in uso.
  - Sv39 impiega una tabella delle pagine con  $2^{27}$  entry, ciascuna con un PPN a 44 bit e flag.
- Processo di Traduzione:**
  - L'hardware di paging traduce un indirizzo virtuale utilizzando i primi 27 bit.
  - La Figura illustra il processo con una vista logica della tabella delle pagine come un array di PTE.



# Gestione della memoria

## Struttura a Tre Livelli

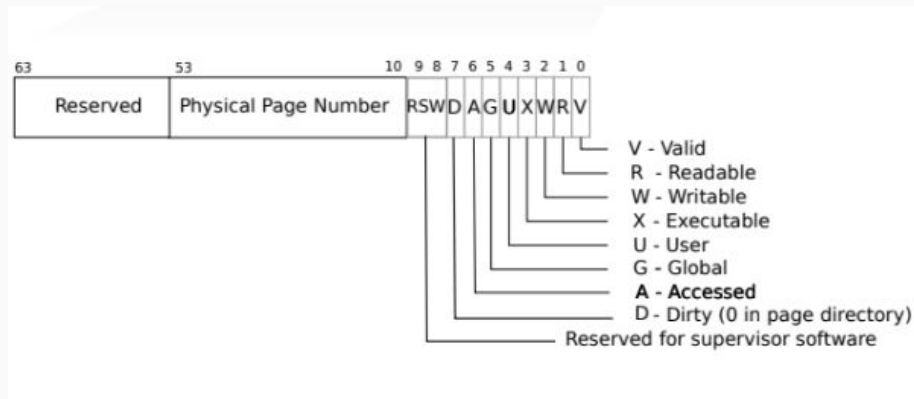
- **Memoria Efficientemente Organizzata:**
  - La tabella delle pagine è archiviata come un albero a tre livelli.
  - Ogni livello contiene 512 PTE, con l'hardware di paging che utilizza i primi 9 bit per selezionare PTE in ciascun livello.
- **Vantaggi della Struttura a Tre Livelli:**
  - Efficienza nella registrazione di PTE, salvando spazio quando ampi intervalli di indirizzi non sono mappati.
  - Per dire alla CPU di utilizzare una tabella delle pagine, il kernel deve scrivere l'indirizzo fisico della tabella delle pagine root nel registro satp.(Supervisor Address Translation and Protection)



# Gestione della memoria

- **Flag Contenuti in PTE:**

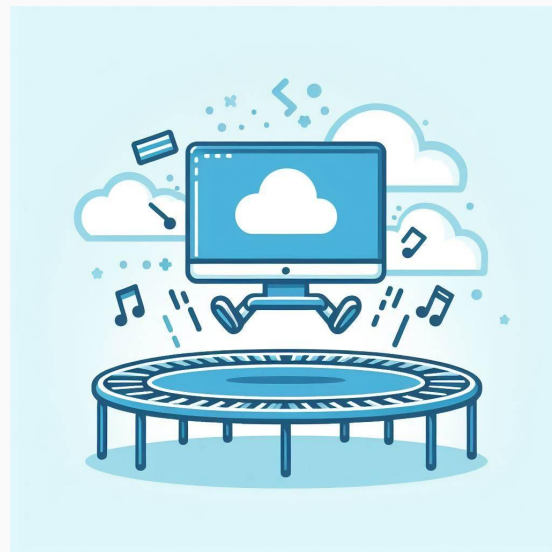
- **PTE\_V:** Indica la presenza della PTE.
- **PTE\_R:** Controllo lettura - determina se le istruzioni possono leggere la pagina.
- **PTE\_W:** Controllo scrittura - determina se le istruzioni possono scrivere nella pagina.
- **PTE\_X:** Controllo esecuzione - determina se la CPU può eseguire il contenuto della pagina come istruzioni.
- **PTE\_U:** Controllo modalità utente - determina se le istruzioni in modalità utente possono accedere alla pagina.



# Gestione della memoria

## Gestione della Pagina Trampolino

- Ruolo Chiave del Trampolino:
  - Mappato in tutte le tabelle delle pagine dei processi e nella tabella delle pagine del kernel a un indirizzo noto, chiamato TRAMPOLINE.
- Accesso in Modalità Supervisore:
  - La pagina del trampolino è accessibile solo in modalità supervisore, facilitando la gestione delle trap.
- Ponte tra Utente e Kernel:
  - Garantisce transizioni fluide tra i livelli di privilegio durante le trap.





# Gestione della memoria

## Allocazione e Liberazione della Memoria Fisica

- **Metodo di Assegnazione:**
  - xv6 assegna e libera intere pagine di dimensioni 4096 byte. Utilizzo di pagine di dimensioni fisse semplifica la gestione della memoria.
- **Tracciamento delle Pagine Libere:**
  - Utilizzo di una "linked list":
    - Pagine libere sono tenute in una lista concatenata
    - L'allocazione rimuove una pagina da questa lista.
    - La liberazione aggiunge la pagina alla lista.

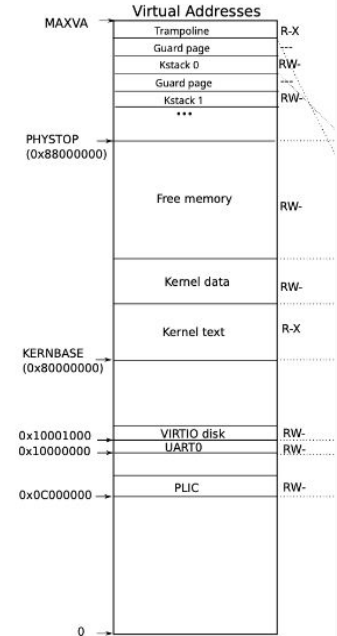
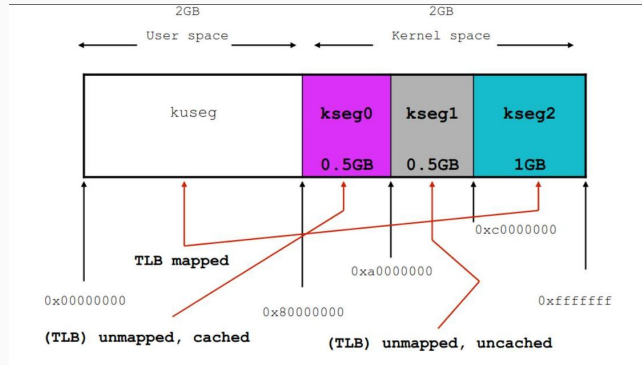
# Differenze

## OS161

- Supporta paging e Demand paging
- Dimensione pagina 4KB
- Allocazione pagine: allocazione di n pagine con `ram_stealmem(npages)`

## Xv6

- Supporta il paging
- Dimensione pagina 4KB
- Allocazione pagine: una alla volta con `kalloc()`



# System Call

Tre tipi di eventi provocano un'interruzione dell'esecuzione normale delle istruzioni:

- **Chiamata di Sistema (System Call):**
  - Il programma utente richiede al kernel di eseguire un'operazione specifica.
- **Eccezione:**
  - Un'istruzione (utente o kernel) compie un'azione illegale.
- **Interruzione del Dispositivo:**
  - Un dispositivo richiede attenzione, ad esempio, completamento di un'operazione di I/O.

# System Call

## **Gestione delle Trap in xv6:**

- Tutte le trap sono gestite nel kernel, impedendo al codice utente di gestirle direttamente.
- Ragioni: Chiamate di sistema sono gestite nel kernel; isolamento richiesto per interrupt da dispositivi, eccezioni vengono gestite nel kernel.

## **Fasi della Gestione delle Trap in xv6:**

- Rilevamento dell'interruzione o eccezione hardware della CPU.
- Salvataggio dello stato del processo.
- Esecuzione dell'azione appropriata nel kernel, come una chiamata di sistema.

## **Differenziazione nella Gestione delle Trap in xv6:**

- Utilizzo di codice separato per gestire trap utente, trap kernel e interruzioni del timer.
- Maggiore chiarezza e controllo nella gestione di ciascun tipo di trap.

# System Call

## **Progettazione delle Trap in xv6:**

- L'hardware RISC-V non modifica le tabelle delle pagine durante una trap.
- Necessità di mapping valido del gestore trap sia nella tabella delle pagine utente che in quella del kernel.

## **Gestione delle Trap in User Space e Kernel Space - Punti Comuni:**

- Entrambe coinvolgono l'uso di una "trampoline page" mappata sia nello spazio utente che nel kernel.
- Salvataggio dei registri necessari in un trapframe prima di gestire la trap.

# Differenze

**OS161 gestisce le system calls in maniera analoga a xv6:**

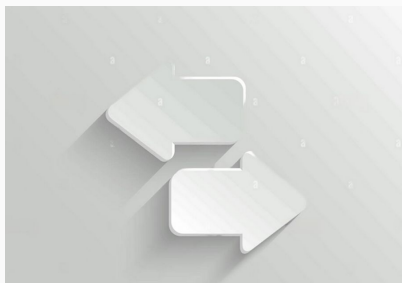
- si scatena una trap;
- avviene il context switching (con creazione del trap frame nel kernel stack);
- si esegue la system call;
- si ripristina lo user stack e si torna all'esecuzione in user mode.

Differenze di carattere implementativo (dovute principalmente all'architettura, RISC-V MIPS).

# Sincronizzazione

## Introduzione ai Meccanismi di Sincronizzazione:

- Fondamentali per il coordinamento tra processi, attività o dispositivi in esecuzione parallela.
- Obiettivo: Evitare "race condition" e "deadlock" per garantire coerenza e sicurezza del sistema.
- **Critical Section:** Sezione di codice condivisa tra più processi.



# Sincronizzazione

- **Spinlock:**
  - Accesso esclusivo e atomico a risorse condivise.
  - Utilizzo di "busy waiting" (attesa attiva) per il rilascio della risorsa.
  - Struttura "spinlock" con informazioni come stato, nome, e core CPU associato.
  - Uso di "test and set" a basso livello per evitare acquisizioni simultanee.
  - Operazioni: acquire, release, initlock.
  - Disabilitazione/riabilitazione interrupt per prevenire timeslicing quando il lock è posseduto.



# Sincronizzazione

- **Sleeplock:**

- Combina attesa attiva (spinlock) con attesa passiva (mettere in "sleeping").
- Garantisce che i processi "addormentati" possiedano lo spinlock per evitare race condition.
- Processi "sleeping" risvegliati tramite interrupt.
- Adatto per attese più lunghe.

# Differenze

## OS161

- Semafori
- Lock
- Condition Variable
- Wait Channel
- Spinlock

## Xv6

- Spinlock
- Sleeplock

# IMPLEMENTAZIONI NUOVE FUNZIONALITA'



# System Call: ps

## Implementazione Syscall:

- 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.
- 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.
- 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

# System Call: ps

- 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.
- `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`.
- 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`.
- 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.
- 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.

# System Call: ps

- la syscall restituisce la lista di processi attivi con relativi pid, la syscall è stata implementata esportando la tabella dei processi.

```
xv6 kernel is booting  
  
hart 1 starting  
hart 2 starting  
init: starting sh  
$ ps  
process list  
name      pid      state  
init      1        SLEEPING  
sh         2        SLEEPING  
ps         3        RUNNING  
$
```

# Implementazione doppia indirezione

- La seconda funzionalità implementata è l'incremento della dimensione massima di un file all'interno del sistema operativo.
- Inizialmente i file in xv6 sono limitati a 268 blocchi
- Questo limite deriva dal fatto che un inode xv6 contiene 12 blocchi diretti e un numero di blocco indiretto singolo, che punta a un blocco che contiene fino a 256 blocchi, per un totale di  $12+256=268$
- Lo scopo di questa implementazione è quello di incrementare il limite appena citato andando ad inserire un blocco indiretto doppio, in questo modo un inode potrà contenere:  $11+256 + 2562 = 65803$  blocchi
- Andando a selezionare il test writebig attraverso il comando “writebigfile”, oppure tramite “usertests writebig” viene creato il file più grande possibile e segnala tale dimensione.

```
xv6 kernel is booting
```

```
hart 1 starting  
hart 2 starting  
init: starting sh  
$ writebigfile
```

```
MAXFILE: 65803
```

```
[=====] 100%  
wrote: 65803
```

```
xv6 kernel is booting
```

```
hart 1 starting  
hart 2 starting  
init: starting sh  
$ usertests writebig  
(MAXFILE: 65803 ) test :writebig passed  
$
```

FINE