OS161

OS161 è un sistema operativo originariamente sviluppato per essere eseguito sull'architettura MIPS (Microprocessor without Interlocked Pipeline Stages). MIPS è un'architettura di processore RISC (Reduced Instruction Set Computer) nota per la sua semplicità e efficienza. Tuttavia, OS161 è stato adattato per funzionare su diverse varianti di architetture MIPS, come MIPS32 e MIPS64.

MIPS è noto per la sua semplicità e chiarezza. Le istruzioni sono di dimensioni fisse e seguono un modello regolare. L'architettura MIPS è in grado di esemplificare concetti fondamentali dei sistemi operativi, come la gestione dei processi, la gestione della memoria e la sincronizzazione tra processi.

xv6 vs xv6 RISC-V

xv6 è un sistema operativo educativo basato su Unix V6, creato presso l'Università di Berkeley, con l'obiettivo di illustrare i concetti fondamentali dei sistemi operativi. xv6 è stato inizialmente sviluppato per l'architettura x86 (Intel), ma è stato successivamente portato a diverse altre architetture, inclusa RISC-V.

Creato nel [2006](https://it.wikipedia.org/wiki/2006) a scopo educativo, nel 2020 ne è stata realizzata una [conversione](https://it.wikipedia.org/wiki/Portabilit%C3%A0) per [RISC-V](https://it.wikipedia.org/wiki/RISC-V).

Parte del codice della non più mantenuta versione per [x86](https://it.wikipedia.org/wiki/X86) è ricavato da JOS (un precedente sistema operativo Unix-like)

Principali differenze tra xv6 per x86 e xv6 per RISC-V:

***xv6 per x86:***

1. **Architettura di destinazione:**

La versione originale di xv6 è stata sviluppata per l'architettura x86 (Intel), che è ampiamente utilizzata nei computer personali e nei server.

1. **Interfaccia hardware:**

xv6 per x86 interagisce con l'hardware specifico dell'architettura x86, come le interruzioni x86, la gestione della memoria x86 e altre caratteristiche specifiche di questa architettura.

1. **Sistema di istruzioni:**

xv6 per x86 utilizza l'insieme di istruzioni x86, che è diverso dall'insieme di istruzioni di altre architetture, come RISC-V.

**xv6 per RISC-V:**

1. **Architettura di destinazione:**

La versione xv6 per RISC-V è stata adattata per funzionare sull'architettura RISC-V, che è un'architettura di processore a basso consumo energetico e open-source.

1. **Interfaccia hardware:**

xv6 per RISC-V interagisce con l'hardware specifico dell'architettura RISC-V, come le interruzioni RISC-V, la gestione della memoria RISC-V e altre caratteristiche specifiche di questa architettura.

1. **Sistema di istruzioni:**

xv6 per RISC-V utilizza l'insieme di istruzioni RISC-V, che è progettato per essere semplice ed efficiente.

In sostanza, la differenza principale tra xv6 e xv6 per RISC-V sta nell'architettura di destinazione e nei dettagli tecnici correlati all'hardware specifico.

**OS161:**

1. **Progetto educativo più completo:** OS161 è stato sviluppato come parte del corso di sistemi operativi presso l'Università di Harvard (CS161) e mira a fornire una comprensione più approfondita dei concetti dei sistemi operativi e della programmazione a basso livello.
2. **Struttura modulare:** Le chiamate di sistema in OS161 sono organizzate in moduli, ciascuno dei quali copre un aspetto specifico delle operazioni del sistema, come la gestione dei file, la gestione dei processi, la gestione della memoria, ecc.
3. **Varietà di chiamate di sistema:** OS161 include un insieme più ampio di chiamate di sistema, che coprono operazioni di gestione dei processi, sistemi di file, allocazione di memoria e così via.
4. **Maggiore complessità:** A causa del suo focus sulla fornitura di una comprensione approfondita dei sistemi operativi, OS161 può essere più complesso e impegnativo da comprendere e utilizzare.

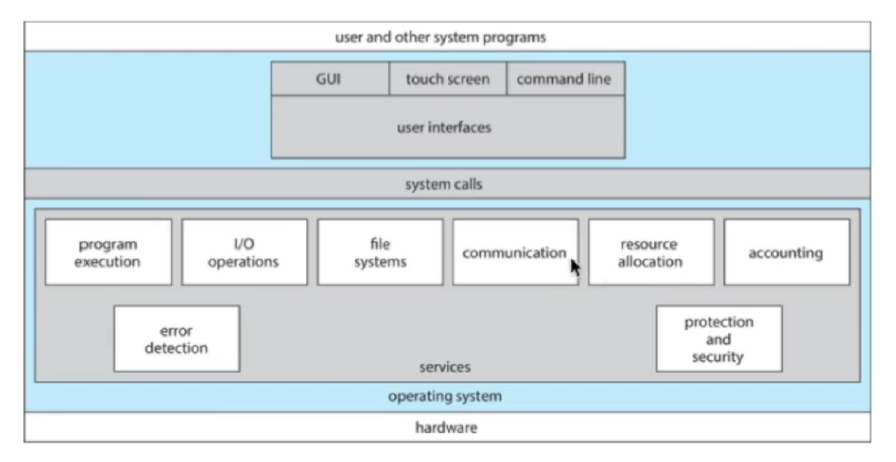
**xv6:**

1. **Semplicità focalizzata:** xv6 è stato sviluppato presso l'Università di Berkeley come una ricreazione semplificata di Unix V6, con l'obiettivo di fornire una visione più chiara e semplificata del funzionamento dei sistemi operativi.
2. **Set di chiamate di sistema di base:** Le chiamate di sistema in xv6 sono limitate a un set di base, come la manipolazione dei file, l'esecuzione dei programmi, la creazione e la gestione dei processi e l'allocazione della memoria.
3. **Eredità Unix:** xv6 è stato progettato per emulare il funzionamento di Unix V6, quindi molte delle chiamate di sistema sono simili a quelle presenti in un sistema Unix tradizionale.
4. **Facilità di comprensione:** Grazie al suo approccio più semplice, xv6 viene spesso utilizzato come introduzione più accessibile ai concetti dei sistemi operativi, rendendolo più adatto ai principianti.

In sintesi, mentre OS161 offre un'esplorazione più ampia e complessa dei sistemi operativi, xv6 si concentra su una introduzione più diretta e semplificata.

***System calls***

Meccanismo usato da un processo a livello utente o applicativo per richiedere un servizio a livello kernel del sistema operativo. Le system calls forniscono dunque un’interfaccia ai servizi forniti da un sistema operativo (esecuzione di programmi, operazioni di I/0, file system ecc..) e sono generalmente scritte in C, C++.



Tipi di System Calls:

1. **Gestione dei processi/thread:** include chiamate di sistema per la creazione, l'esecuzione e la gestione dei processi/thread.

* create/terminate process
* end, abort
* load, execute
* get/info/set process attributes
* wait for time
* wait event, signal event
* allocate and free memory
* ecc..

1. **Gestione dei file e dei file system:** offre chiamate di sistema per la lettura e scrittura di file, la manipolazione delle directory e l'accesso ai file.

* create/delete file
* open/close file
  + read/write
  + get and set attributes of the file
  + ecc…

1. **Gestione dei dispositivi**

* request/release
  + read/write
  + get/set attribute dei dispositivi
  + logically attach or detach
  + ecc…

1. **Gestione delle informazioni**

* get/set time or date
* get/set system data
* get and set process, file, or device attributes
* ecc…

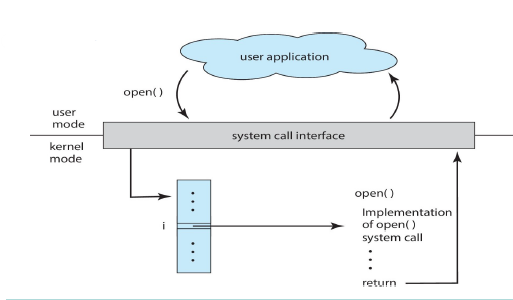
1. **Comunicazione**

* create/delete connessione
* send/receive messages
  + (From **client** to **server)**
* trasferimento di informazioni di stato
* ecc…

1. **Protezione**
   * + control access to resources
     + get/set permissions
     + allow and deny accesso utente

***Implementazione system calls***

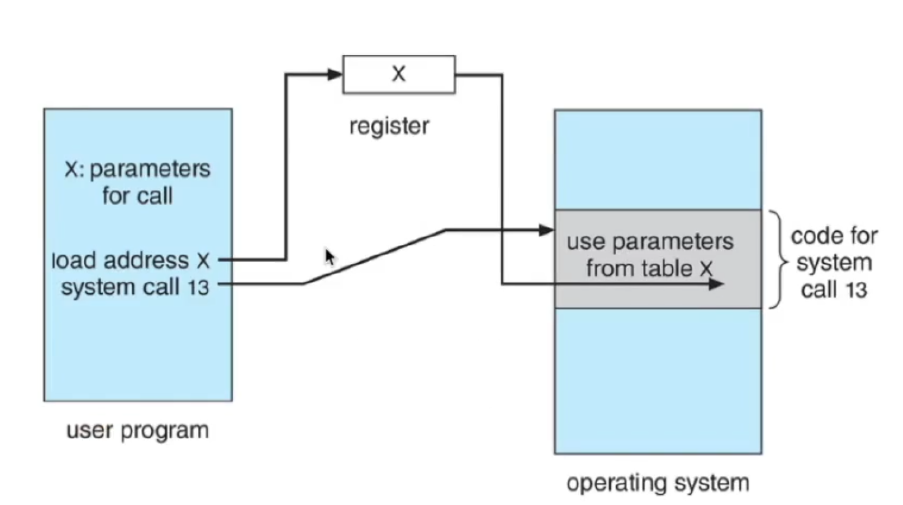
Il sistema operativo associa un numero o un ID ad ogni system call ed una tabella nello spazio del kernel che dice per ogni numero di system call, cosa deve essere fatto. Quando chiamiamo una system call tipicamente avviene uno switching da user level a kernel level, passaggio che avviene attraverso una particolare [istruzione](https://it.wikipedia.org/wiki/Istruzione_(informatica)): l’istruzione [***trap***](https://it.wikipedia.org/wiki/Eccezione_(informatica)), passando il numero associato alla system call.



Il sistema operativo va alla struttura dati del numero di chiamata di sistema ed esegue le operazioni, quando termina l'esecuzione si ritorna all'user mode.

Normalmente alle system calls vengono passati dei parametri ed il modo più semplice per passare quest’ultimi dall’user al kernel è quello di metterli nei registri e nello stack.

In pratica l’user program, che ha i parametri, va a caricare l’indirizzo di dove si trovano in un registro e quando il sistema operativo eseguirà l’istruzione trap per una determinata sys call oltre ad andare ad indentificare il codice di esecuzione corrispondente userà l’indirizzo di dove si trovano i parametri.



***System calls OS161***

In OS161 implementato su MIPS c’è un **unico gestore delle eccezioni**, un exception handler, che viene chiamato anche per le sys calls e le interruzioni.

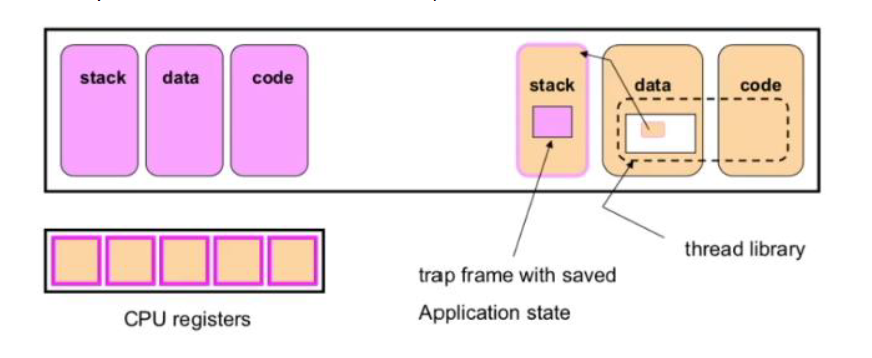
La prima cosa che farà l’exception handler è quella di riconoscere da chi è stato chiamato poiché vi è una funzione di gestione diversa per eccezioni, chiamate di sistema e interruzioni.

Ogni volta che viene chiamata una system call, l'esecuzione viene passata dalla modalità utente alla modalità kernel.

Mentre il kernel gestisce la chiamata di sistema, lo stato della CPU dell'applicazione viene salvato in un **trap frame** sullo stack del kernel del thread e i registri della CPU sono disponibili per mantenere lo stato di esecuzione del kernel.

***OS161 Trapframe***

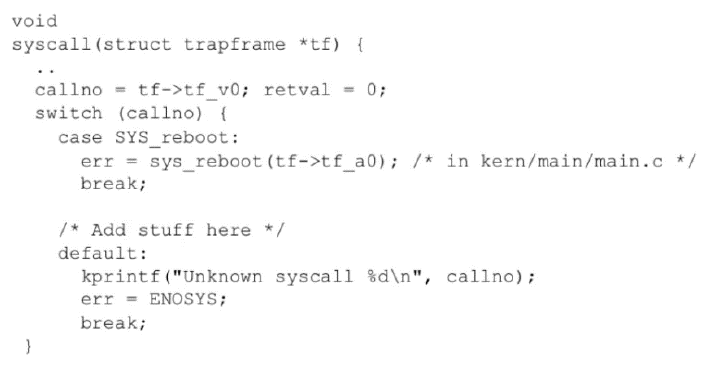
Struttura che serve per salvare lo stato della CPU cioè i registri dei task in esecuzione quando si attiva una trap.



Come si può vedere abbiamo due stack: lato utente e lato kernel.

***OS161 MIPS System call Handler***

La funzione che viene attivata quando il gestore di eccezioni si accorge che si tratta di una System call è la seguente (kern/arch/mips/syscall):



Si chiama syscall e riceve come unico parametro il puntatore al trapframe.

In sostanza è un costrutto switch case. La funzione va a guardare uno dei contenuti del trapframe, il campo tf\_v0, cioè il call number, il numero della System Call.

Il gestore delle eccezioni riconosce l’eccezione sys call e poi la syscall function riceve un argomento, dentro il trapframe, che dice quale tipo di sys call è.

Il numero syscall a livello di costanti come eccezioni è ([kern](https://github.com/NickStephens/os161/tree/ac9262955b4a203c3fe5098d48ef73e31cd59127/kern)/[arch](https://github.com/NickStephens/os161/tree/ac9262955b4a203c3fe5098d48ef73e31cd59127/kern/arch)/[mips](https://github.com/NickStephens/os161/tree/ac9262955b4a203c3fe5098d48ef73e31cd59127/kern/arch/mips)/[include](https://github.com/NickStephens/os161/tree/ac9262955b4a203c3fe5098d48ef73e31cd59127/kern/arch/mips/include)/trapframe.h):



Gli altri sono interrupt, trap per la tlb ecc..

***OS161 System Calls:***

1. **exit** - ***void exit(int status)*** *–* per terminare il processo
2. **chdir** - ***int chdir(const char \*pathname) –*** per cambiare la directory corrente
3. **close** - ***int close(int fd) –*** *per* chiudere il file
4. **dup2** - ***int dup2(int oldfd, int newfd) -*** *per* clonare file descriptor
5. **execv** - ***int execv(const char \*program, char \*\*args)*** *–* per eseguire un programma
6. **fork - *pid fork(void)*** *-* copiare il processo corrente
7. **fstat** – ***int fstat (int fd, struct stat \*statbuf) -*** ottenere informazioni sullo stato del file
8. [**fsync**](https://ops--class-org.translate.goog/man/syscall/fsync.html?_x_tr_sl=en&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp)- **int fsync(int fd)** - scaricare i dati del filesystem per un file specifico su disco
9. **ftruncate** – **int ftruncate(int *fd*, off\_t *filesize*)** - impostare la dimensione di un file
10. **\_getcwd** - ottenere il nome della directory di lavoro corrente (backend)
11. **getdirentry** – **int getdirentry(int *fd*, char \**buf*, size\_t *buflen***) - leggere il nome del file dalla directory
12. [**ge**](https://ops--class-org.translate.goog/man/syscall/getpid.html?_x_tr_sl=en&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp)**tpid** – **pid getpid(void) -** ottenere l'ID del processo
13. [**ioctl**](https://ops--class-org.translate.goog/man/syscall/ioctl.html?_x_tr_sl=en&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp) – **int ioctl(int *fd*, int *code*, void \**data*)** - varie operazioni di I/O del dispositivo
14. **link** – **int link(const char \*oldname, const char \*newname)** - creare un collegamento fisico a un file
15. **Iseek** - **off\_t lseek(int *fd*, off\_t *pos*, int *whence***) - cambiare la posizione corrente nel file descriptor
16. **Istat** – **int lstat(const char \*pathname, struct stat \*statbuf**) - ottenere informazioni sullo stato del file
17. **mkdir** – **int mkdir(const char \**pathname*, mode\_t *mode*)** - creare una directory
18. **open – int open(const char \*filename, int flags)** - aprire un file
19. **pipe** – **int pipe(int \**fds*)** - creare un oggetto tubo
20. **read** - **ssize\_t read(int *fd*, void \**buf*, size\_t *buflen*)** - leggere i dati dal file
21. **readlink – int readlink(const char \**path*, char \**buf*, size\_t *len*)** - recuperare il contenuto del collegamento simbolico
22. **reboot** – **int reboot(int *code*)** - riavviare o arrestare il sistema
23. **remove** – **int remove(const char \**pathname*)** - eliminare un file
24. **rename** – **int rename(const char \**oldname*, const char \**newname*)** - rinominare o spostare un file
25. **rmdir** – **int rmdir(const char \**pathname*)** - rimuovere la directory
26. **sbrk** - **void \* sbrk(intptr\_t *amount*)** - impostare l'interruzione del processo (alloca memoria)
27. **stat** – **int stat(const char \**pathname*, struct stat \**statbuf*)** - ottenere informazioni sullo stato del file
28. **symlink** – **int symlink(const char \**oldname*, const char \**linkname*)** - creare un collegamento simbolico
29. **sync – void sync(void)** - scaricare i dati del filesystem sul disco
30. **\_time** - ottenere l'ora del giorno
31. **waitpid - pid\_t waitpid(pid\_t *pid*, int \**status*, int *options*)** - attendere l'uscita del processo
32. **write - ssize\_t write(int *fd*, const void \**buf*, size\_t *buflen*)** - scrivere i dati su file

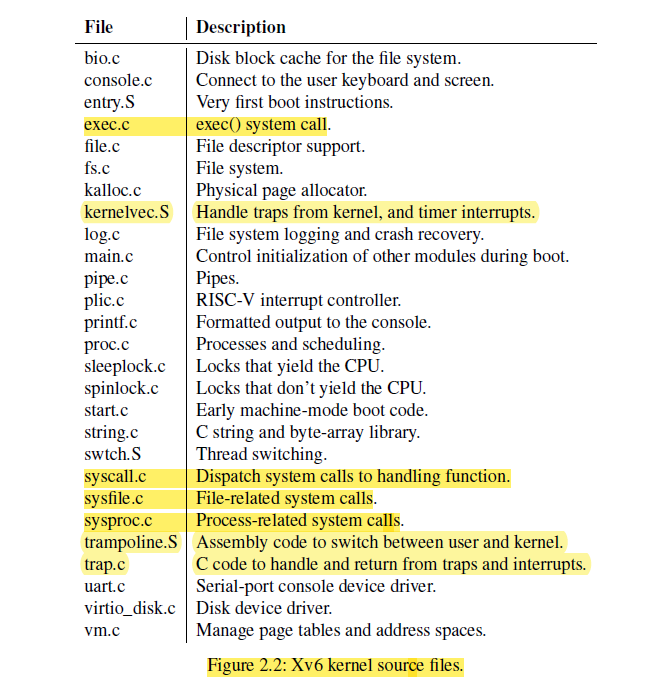
***System calls xv6*** *(Capitoli 1 2 4 del libro xv6 Risc-v)*

Il kernel xv6 fornisce un subset di servizi e system calls del tradizionale kernel Unix V6.

System call: il programma user che è in esecuzione esegue l’istruzione **ecall** per chiedere al kernel di fare qualcosa.

Un’applicazione che vuole invocare una funzione del kernel deve switchare da modalità utente a modalità kernel (o supervisore). Una volta che la CPU è passata alla modalità supervisore, il kernel può quindi convalidare gli argomenti della chiamata di sistema, decidere se l'applicazione è autorizzata a eseguire l'operazione richiesta, quindi negarla o eseguirla.

Xv6 kernel files:



Un processo può effettuare una chiamata di sistema eseguendo ***l'istruzione ecall*** RISC-V. Questa istruzione

aumenta il livello di privilegio hardware e modifica il contatore del programma in un punto di ingresso definito dal kernel. Il codice nel punto di ingresso passa a uno stack del kernel ed esegue le istruzioni del kernel che implementano la chiamata di sistema. Una volta completata la chiamata di sistema, il kernel torna allo stack utente e ritorna nello spazio utente chiamando l'istruzione sret, che riduce il livello di privilegi hardware e riprende l'esecuzione delle istruzioni dell'utente subito dopo l'istruzione della chiamata di sistema.

***Xv6 Trap Handler***

Ciascuna CPU RISC-V ha una serie di registri di controllo nei quali il kernel scrive per indicare alla CPU come gestire sys call, eccezioni o interrupts e che il kernel può leggere per sapere che tipo di trap si è verificata.

Con trap si indica genericamente una delle seguenti situazioni: system calls, interrupts o exceptions.

I registri più importanti sono (in kernel mode):

* + ***stvec (****trap vector****):*** contiene l’indirizzo del gestore della trap (handler code), al quale saltare per gestirla. Il kernel scrive questo indirizzo qui.
    - ***Kernelvec:*** handles traps che avvengono in kernel mode
    - ***Uservec:*** handles traps che avvengono in user mode
  + ***sepc:*** contiene l’indirizzo dal quale è stato effettuato il salto a stvec. Quando avviene una trap, la CPU salva il valore del PC (Program counter) qui. L’istruzione *sret (return from trap)* copia ***sepc*** in ***pc.***
  + ***scause***: contiene il numero che descrive chi ha scatenato la trap. La CPU lo scrive.
  + ***sscratch:*** il kernel inserisce qui un valore che torna utile proprio all'inizio della gestione di una trappola.
  + ***sstatus***: contiene un bit SIE(Interrupts enabled) che controlla se gli interrupt del dispositivo sono abilitati. Se la kernel cancella SIE, RISC-V posticiperà gli interrupt del dispositivo finché il kernel non imposta SIE.

La SPP(Previous Privilege Level, 0=user, 1=supervisor) bit indica se un trap proviene dalla modalità utente o dalla modalità supervisore e controlla cosa la modalità *sret* ritorna.

Quando è necessario forzare una trap, l'hardware RISC-V esegue le seguenti operazioni (delega la gestione della trap al supervisor mode) per tutti i tipi di trap eccetto gli iterrupt del timer:

* 1. se la trap è un device interrupt e il bit di stato SIE è azzerato, non viene eseguita nessuna delle operazioni seguenti
  2. disabilitazione degli interrupt, cancellando il bit di stato SIE.
  3. copia del valore contenuto in PC in sepc.
  4. salvataggio della modalità corrente (utente o supervisore) nel bit SPP in *sstatus.*
  5. impostazione di *scause* per riflettere la causa della trappola
  6. impostazione della modalità supervisore
  7. copia di *stvec* sul PC
  8. inizio dell'esecuzione sul nuovo PC

Nota che la CPU non passa alla tabella delle pagine del kernel, non passa a uno stack nel kernel e non salva nessun registro diverso dal pc. Il software del kernel deve eseguire queste attività mentre la CPU effettua un lavoro minimo al fine di avere una maggiore flessibilità con il software.

***Traps dallo spazio utente (gestione delle traps che avvengono quando siamo in user mode)***

Una trappola può verificarsi durante l'esecuzione nello spazio utente se il programma utente effettua una **chiamata di sistema (istruzione ecall),** o fa qualcosa di illegale, oppure se un dispositivo interrompe l'esecuzione.

Il percorso di alto livello di una trap dello spazio utente è:

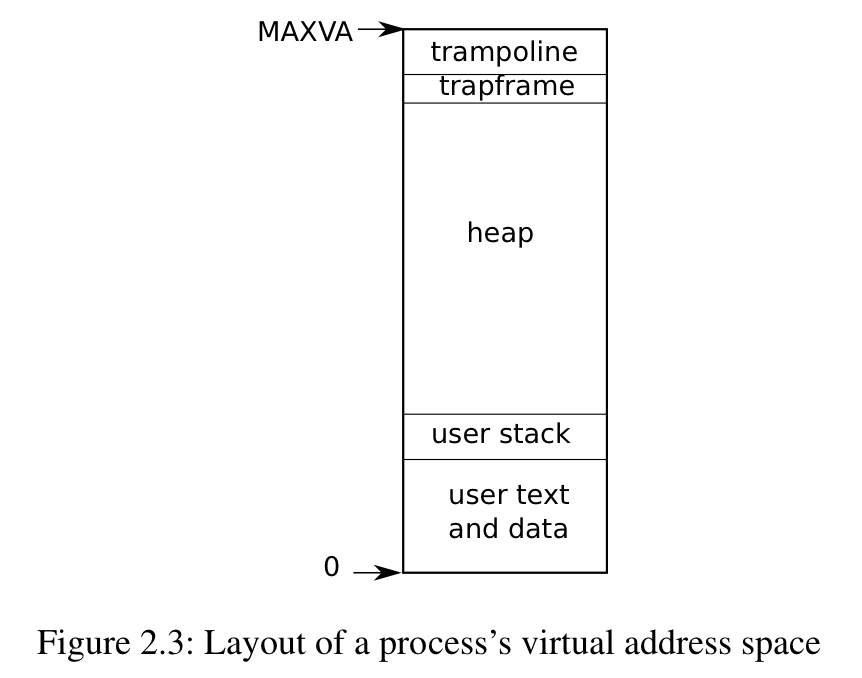
1. ***uservec*** (kernel/**trampoline.S**:16) ← questo è l’indirizzo salvato in ***stvec***
2. ***usertrap*** (kernel/**trap.c**:37);
3. e al ritorno, ***usertrapret*** (kernel/**trap.c**:90)
4. ***userret*** (kernel/**trampoline.S**:16).

***Uservec***

Poiché l'hardware RISC-V non cambia le page table durante una trap (il registro satp punta alla page table corrente in memoria), la page table utente deve includere una mappatura per *uservec* (il vettore delle istruzioni delle trappole a cui *stvec* punta). *Uservec* cambia *satp* per puntare alla page table del kernel. Questo cambio non deve causare problemi, così la page table del kernel deve anche contenere una mappatura a uservec. Dunque, uservec deve essere mappato allo stesso indirizzo nella tabella delle pagine del kernel e nella tabella delle pagine utente per continuare l'esecuzione delle istruzioni dopo il cambio.

Xv6 usa una pagina ***trampoline (*trampolino**), che contiene *uservec*, mappata allo stesso indirizzo virtuale (TRAMPOLINE) nella tabella delle pagine del kernel e in ogni tabella delle pagine utente.

Il contenuto della trappola è impostato in trampoline.S, e nell'esecuzione del codice utente stvec è impostato su uservec.

[](https://timothya.com/learning/Pasted%20image%2020210910190853.png)

* Quando uservec inizia l’esecuzione, tutti e 32 i registri contengono valori posseduti dal codice interrotto. L'istruzione csrrw all'inizio ***di uservec scambia i contenuti del primo registro utente a0 con sscratch (registro).***
* Il compito successivo di uservec è ***salvare i registri utente:***
* prima di entrare in user mode, il kernel ha precedentemente impostato sscratch per puntare a un trapframe specifico per processo che (tra le altre cose) ha spazio per salvare tutti i registri utente (kernel/proc.h:44).
* satp si riferisce ancora alla tabella delle pagine utente e uservec ha bisogno che il trapframe sia mappato nello spazio degli indirizzi utente. Allora quando si crea ogni processo, xv6 assegna una pagina per il trapframe del processo, e si assicura che sia sempre mappato all'indirizzo virtuale utente TRAPFRAME, che è appena sotto TRAMPOLINE. Il p->trapframe punta anche al trapframe, anche se al suo indirizzo fisico, in modo che il kernel possa usarlo attraverso la tabella delle pagine del kernel.
* dopo lo scambio di a0 e sscratch, a0 contiene un puntatore al trapframe del processo corrente. uservec salva ora tutti i registri utente lì, compreso il a0 dell'utente, letto da sscratch.
  + Il trapframe contiene puntatori allo stack kernel del processo corrente, all'hartid della CPU corrente, all'indirizzo di usertrap e all'indirizzo della tabella delle pagine del kernel. uservec recupera questi valori, cambia satp alla tabella delle pagine del kernel, e ***chiama usertrap.***

***Usertrap()***

Il compito di usertrap è determinare la causa della trappola (eccezione, device interrupt, syscall, timer interrupt), elaborarla e ritornare (kernel/trap.c:37):

* + aggiorna stvec in kernelvec, (perché gli interrupt utilizzano gestori diversi mentre sono in modalità kernel), in modo che una trappola mentre è nel kernel venga gestita da kernelvec.
  + salva il sepc (il contatore del programma utente salvato) ancora perché potrebbe esserci un cambio di processo in usertrap che potrebbe causare sovrascrittura di sepc.

Se la trappola è una chiamata di sistema, **syscall()** la gestisce; se è un'interruzione del dispositivo, **devintr()**; altrimenti è un'eccezione, e il kernel termina il processo che ha causato il guasto. Il percorso della chiamata di sistema aggiunge quattro al PC utente salvato perché RISC-V, nel caso di una chiamata di sistema, lascia il puntatore del programma puntato all'istruzione ecall. All'uscita, usertrap controlla se il processo è stato terminato o se deve cedere la CPU (se questa trappola è un'interruzione del timer).

***Usertrapret e userret***

Una volta che la trap è stata processata, il primo passo nel ritorno allo spazio utente è la chiamata a usertrapret (kernel/trap.c:90). Questa funzione configura i registri di controllo RISC-V per prepararsi a una futura trappola dallo spazio utente. Questo comporta il cambio di stvec per fare riferimento a uservec, la preparazione dei campi trapframe di cui uservec ha bisogno, e l'impostazione di sepc al contatore del programma utente precedentemente salvato. Alla fine, usertrapret chiama userret sulla pagina trampolino che è mappata sia nelle tabelle delle pagine utente che in quelle del kernel; il motivo è che il codice assembly in userret cambierà le tabelle delle pagine.

La chiamata di usertrapret a userret passa un puntatore alla tabella delle pagine utente del processo in a0 e TRAPFRAME in a1 (kernel/trampoline.S:88).

userret cambia satp alla tabella delle pagine utente del processo. Ricorda che la tabella delle pagine utente mappa sia la pagina trampolino che TRAPFRAME, ma nient'altro dal kernel. Ancora una volta, il fatto che la pagina trampolino sia mappata allo stesso indirizzo virtuale nelle tabelle delle pagine utente e del kernel è ciò che consente a uservec di continuare l'esecuzione dopo il cambio di satp.

userret copia l'utente del trapframe a0 in sscratch in preparazione per un successivo scambio con

TRAPFRAME. Da questo momento in poi gli unici dati che userret potrà utilizzare saranno il contenuto del registro e il contenuto del trapframe. Il prossimo userret ripristina i registri utente salvati dal trapframe, fa uno scambio finale di a0 e sscratch per ripristinare l'utente a0 e salvare TRAPFRAME per la trappola successiva e usa sret per tornare allo spazio utente.

Usertrapret e userret svolgono rispettivamente le azioni di usertrap e uservec.

***Traps dallo spazio kernel (gestione delle traps che avvengono quando siamo in kernel mode)***

Xv6 configura i registri della trap della CPU in modo leggermente diverso a seconda che il codice utente o del kernel stia eseguendo.

Quando il kernel è in esecuzione su una CPU, il kernel imposta stvec affinché punti al codice assembly in **kernelvec** (kernel/kernelvec.S:10). Poiché xv6 è già nel kernel, kernelvec può fare affidamento su satp impostato sulla page table del kernel e sul puntatore allo stack che fa riferimento ad uno stack del kernel valido. kernelvec salva tutti i registri in modo che il codice interrotto possa riprendere eventualmente senza disturbi.

1. kernelvec salva i registri nello stack del thread del kernel interrotto.

Questo è particolarmente importante se la trappola causa un passaggio a un thread diverso: in questo caso, la trappola tornerà effettivamente nello stack del nuovo thread, lasciando i registri del thread interrotto salvati in modo sicuro sul suo stack.

1. kernelvec salta a kerneltrap (kernel/trap.c:134).
2. kerneltrap è preparato per due tipi di trappole:
   1. interruzioni di dispositivo
   2. eccezioni.

* Chiama devintr() (kernel/trap.c:177) per controllare e gestire le prime. Se la trappola non è un'interruzione di dispositivo, deve essere un'eccezione e questa è sempre un errore fatale se si verifica nel kernel di xv6; il kernel chiama panic e smette di eseguire.
* Se kerneltrap è stata chiamata a causa di un'interruzione del timer e un thread del kernel di un processo è in esecuzione (piuttosto che un thread dello scheduler), kerneltrap chiama yield() per dare la possibilità ad altri thread di essere eseguiti. Ad un certo punto, uno di quei thread farà yield e permetterà al nostro thread e al suo kerneltrap di riprendere di nuovo.

1. Quando il lavoro di kerneltrap è completo, deve tornare al codice interrotto dalla trap, ripristina i registri di controllo sepc e sstatus e ritorna a kernelvec (kernel/kernelvec.S:48).
2. kernelvec estrae i registri salvati dallo stack ed esegue sret, che copia sepc in pc e riprende il codice del kernel interrotto.

Xv6 imposta stvec di una CPU su kernelvec quando quella CPU entra nel kernel dallo spazio utente; è possibile vedere ciò in usertrap (kernel/trap.c:29). C'è una finestra di tempo in cui il kernel sta eseguendo ma stvec è impostato su uservec, ed è cruciale che le interruzioni del dispositivo siano disabilitate durante quella finestra. Fortunatamente, il RISC-V disabilita sempre le interruzioni quando inizia a prendere una trappola, e xv6 non le abilita nuovamente fino dopo aver impostato stvec.

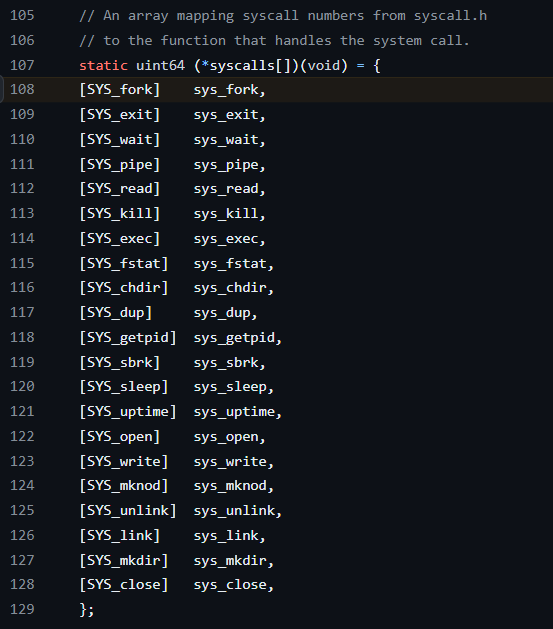
***CODICE***

***Chiamare le system calls***

Vediamo come la chiamata dell'utente raggiunge l'implementazione della chiamata di sistema nel kernel.

* Il codice utente inserisce gli argomenti per la chiamata di sistema nei registri a0 e a1, e inserisce il numero di chiamata di sistema in a7.

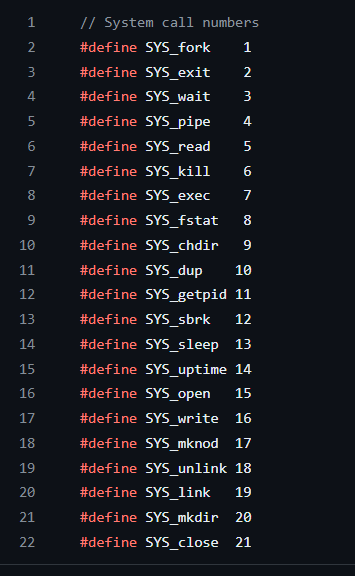
I numeri di chiamata di sistema corrispondono alle voci nell'array syscalls, una tabella di puntatori a funzione (kernel/syscall.c:108).



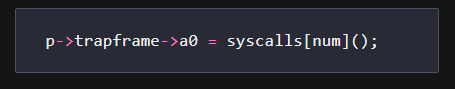
* L'istruzione ecall fa scattare una trappola nel kernel ed esegue uservec, usertrap e poi syscall, come visto in precedenza.
* syscall (kernel/syscall.c:133) recupera il numero di chiamata di sistema dal registro a7 salvato nel trapframe e lo usa per indicizzare syscalls.



* a7 contiene per esempio SYS\_exec (kernel/syscall.h:8), risultando in una chiamata alla funzione di implementazione della chiamata di sistema sys\_exec.



* Quando la funzione di implementazione della chiamata di sistema restituisce, syscall registra il suo valore di ritorno in p->trapframe->a0.



La convenzione di chiamata in linguaggio C su RISC-V inserisce i valori di ritorno in a0.

Le chiamate di sistema convenzionalmente restituiscono numeri negativi per indicare errori e numeri zero o positivi per indicare il successo. Se il numero di chiamata di sistema è invalido, syscall stampa un errore e restituisce -1.

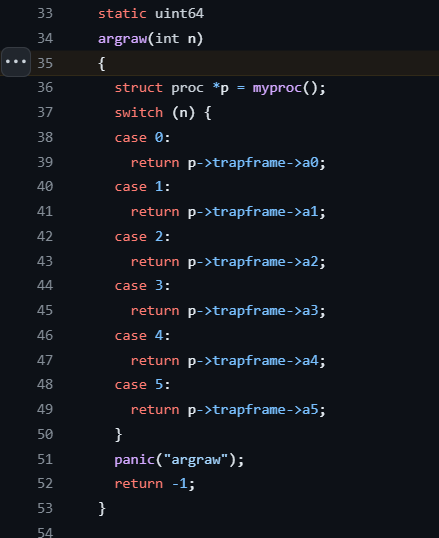
***Argomenti delle system calls***

Le implementazioni delle chiamate di sistema nel kernel devono trovare gli argomenti passati dal codice utente. Gli argomenti sono inizialmente nei registri.

Il codice della trap del kernel salva i registri utente nel trap frame del processo corrente.

Le funzioni argint, argaddr e argfd recuperano l'n-esimo argomento della chiamata di sistema dal trap frame come un intero, un puntatore o un descrittore di file.

Tutte chiamano argraw per recuperare il registro utente salvato appropriato (kernel/syscall.c:35):



Alcune chiamate di sistema passano puntatori come argomenti, e il kernel deve usare questi puntatori per leggere o scrivere nella memoria utente. Questi puntatori presentano due sfide:

* In primo luogo, il programma utente potrebbe essere difettoso o malevolo e potrebbe passare al kernel un puntatore non valido o un puntatore destinato a ingannare il kernel affinché acceda alla memoria del kernel invece della memoria utente.
* In secondo luogo, i mapping della page table del kernel xv6 non sono gli stessi della page table utente, quindi il kernel non può utilizzare istruzioni ordinarie per caricare o memorizzare da indirizzi forniti dall'utente.

***Xv6 system calls:***

1. **fork** - ***int fork()*** *-*  creazione processo, ritorna il PID del figlio
2. **exit** - ***int exit(int status) -***terminare il processo corrente, nessuno tipo di ritorno
3. **wait** –***int wait(int \*status) –***attendere l’uscita del processo figlio, ritorna il PID del figlio
4. **Kill -** –***int kill(int pid) –***terminare il processo, ritorna 0 o -1 per errori
5. **getpid** –***int getpid() –***ritornare l’id del processo corrente
6. **sleep** –***int wait(int \*status) –***pausa per n clock ticks
7. **exec** –***int exec(char \*file, char argv[]) –***caricare un file e lo esegue con gli argomenti, vi è un ritorno solo in caso di errore
8. **sbrk** – ***char \*sbrk(int n) –***aumentare la memoria del processo di n byte e ritorna l’inizio della nuova memoria
9. **open** –***int open(char \*file, int flags) –***aprire un file, i flag indicano lettura o scrittura, ritorna un fd(file descriptor)
10. **write** –***int write(int fd, char \*buf, int n) –***scrivere n bytes da *buf* nel file descriptor fd, ritorna n
11. **read** –***int read(int fd, char \*buf, int n) –*** leggere n bytes in *buf,* ritorna numero read o se è la fine del file
12. **close** –***int close(int fd) –***rilasciare il file aperto fd
13. **dup** –***int dup(int fd) –***ritornare un nuovo file descriptor
14. **pipe** –***int pipe(int p[]) –***creare una pipe e inserisce read/write file descriptors in p[0] e p[1]
15. **chdir** –***int chdir(char \*dir) –***cambiare la directory corrente
16. **mkdir** –***int mkdir(char \*dir) –***creare una nuova directory
17. **mknod** –***int mknod(char \*file, int, int) –***creare un device file
18. **fstat** –***int fstat(int fd, struct stat \*st) –***memorizzare le info su un file aperto in \*st
19. **stat** –***int stat(int fd, struct stat \*st) –***memorizzare le info su uno specifico file in \*st
20. **link** –***int link(char \*file1, char \*file2) –***creare un nuovo nome per il file 1
21. **unlink** –***int unlink(char \*file) –***rimuovere un file

A meno di variazioni tutte le system calls citate ritornano -1 nel caso ci sia un errore altrimenti ritornano 0.

***Descrizione più dettagliata delle system calls***

**exit():** viene utilizzata per terminare il processo chiamante. Essa restituisce uno stato di uscita (exitcode) al processo padre. Il sistema operativo rilascia tutte le risorse associate al processo in uscita.

**chdir():** viene utilizzata per impostare la directory corrente del processo corrente sulla directory denominata dal pathname.

**close():** chiude il file identificato da `filehandle`, rilasciando tutte le risorse associate.

**dup2():** duplica il descrittore di file `oldfilehandle` nel descrittore di file `newfilehandle`.

**execv():** viene utilizzata per rinizializzare lo spazio di indirizzamento di un processo.

Essa permette ad un processo utente di generare un altro processo utente che esegue un altro programma.

A differenza della fork() crea un nuovo eseguibile.

I parametri che riceve devono essere: *il nome del file ELF eseguibile* che viene caricato nello spazio degli indirizzi e gli argomenti da passare a questo nuovo programma*.*

**fork(void):** viene utilizzata per creare un nuovo processo figlio identico al processo chiamante (keeping same executable). Essa consente ad un processo user di generare un altro processo user che svolga lo stesso eseguibile.

Il valore di ritorno al processo chiamante è il nuovo process ID (PID), mentre al nuovo processo (figlio) ritorna 0.

**fstat(int fd, struct stat \*statbuf):**recupera le informazioni sullo stato del file a cui fa riferimento l'handle del file fd e le memorizza nella struttura stat puntata da statbuf.

**fsync():**forza la scrittura su disco dei buffer sporchi del filesystem e di altri stati sporchi del filesystem associati all'oggetto a cui fa riferimento fd.

**ftruncate():** imposta forzatamente la dimensione del file a cui fa riferimento fd su filesize. Se questo espande il file, i nuovi dati appaiono come se fossero riempiti con zero. (Sui file system che supportano file sparsi, non è necessario allocare fisicamente il nuovo spazio.) Se l'azione riduce il file, i dati in eccesso vengono eliminati.

**getcwd():** è utilizzata per ottenere il percorso assoluto della directory di lavoro corrente di un processo.

**getdirentry():** recupera il nome file successivo da una directory a cui fa riferimento l'handle del file filehandle. Il nome è memorizzato in buf, un'area di dimensioni buflen. Viene restituita la lunghezza del nome effettivamente trovato.

**getpid():** ritorna l’id del processo corrente (chiamante)

**ioctl():** esegue un operazione specifica sull’oggetto a cui fa riferimento il file handle fd in input

**link():** crea un nuovo nome per il file a cui fa riferimento il vecchio nome

**iseek:** sposta l'offset del file identificato da `filehandle` secondo i parametri specificati da `pos` e `whence`.

**istat:** recupera le informazioni sullo stato del file a cui fa riferimento il *pathname* e le memorizza nella struttura stat a cui fa riferimento *statbuf.*

**mkdir:** crea una directory

**open:** apre un file specificato dal percorso `filename` con le opzioni specificate dai `flags`. La modalità (`mode`) specifica i permessi quando il file viene creato.

**pipe:** la chiamata pipe crea un oggetto pipe anonimo nel sistema e lo associa a due handle di file nel processo corrente, uno per la fine di lettura e uno per la fine di scrittura.

**read:** legge *buflen* byte dal file specificato da *fd*, nella posizione nel file specificata dalla posizione di ricerca corrente del file, e li memorizza nello spazio puntato da *buf*.

**readlink:** recupera il contenuto del collegamento simbolico denominato dal *path* e lo inserisce nel buffer *buf.* Vengono scritti al massimo *len* byte.

**reboot:** riavvia o spegne il sistema. L'azione specifica dipende dal codice passato

**remove:** il nome del file a cui fa riferimento il *pathname* viene rimosso dal filesystem. Il file vero e proprio non viene rimosso finché non esistono più riferimenti ad esso, indipendentemente dal fatto che tali riferimenti siano sul disco o in memoria.

**rename:** al file (o altro oggetto) a cui fa riferimento *oldname* viene assegnato il nome *newname* e il nome oldname viene rimosso. Se newname esiste già, viene rimosso anche questo.

**rmdir:** rimuove la directory denominata dal *pathname.*

**sbrk:** incrementa (o diminuisce) la dimensione del segmento dei dati del processo chiamante. È spesso utilizzata per richiedere o rilasciare memoria dinamicamente.

**stat:** recupera le informazioni sullo stato del file a cui fa riferimento il percorso e le memorizza nella struttura stat a cui fa riferimento statbuf

**symlink:** crea un collegamento simbolico. Il collegamento simbolico stesso si chiama linkname e punta al vecchio nome.

**sync:** forza la scrittura su disco di tutti i buffer del filesystem sporco e dello stato del filesystem sporco.

**time:** consente a un processo di ottenere informazioni sul tempo corrente o di misurare il tempo di esecuzione di una porzione di codice.

**waitpid:** sospende il processo chiamante fino a quando uno specifico processo figlio identificato da `pid` termina. Restituisce l'ID del processo figlio terminato e imposta lo stato di uscita nel puntatore `status`.

**wait:** sospende il processo chiamante finché uno qualsiasi dei suoi processi figli termina. Restituisce l'ID del processo figlio terminato e imposta lo stato di uscita nel puntatore `status`.

**write:** scrive *buflen* byte nel file specificato da *fd*, nella posizione nel file specificata dalla posizione di ricerca corrente del file, prendendo i dati dallo spazio puntato da *buf.*

***System calls implementate nella versione base di OS161***

* **gestione dei processi**:
  + Creazione: fork, execv
  + Distruzione: \_exit
  + Sincronizzazione: waitpid
  + Attribute Mgmt: getpid

(non ho certezze, dal codice sembra che non sia implementato nulla)

*Le System calls NON implementate o implementate in versioni ridotte (implementate in laboratorio) nella versione base di OS161:*

* **gestione dei processi**:
  + read
  + write
  + exit
  + waitpid
  + getpid
  + fork
* **gestione dei file** (PACCO DI SLIDE os161-file, Cabodi):
  + open()
  + close()
  + read()
  + write()
  + lseek()
  + dup2()

***System calls implementate in xv6 (****quelle viste sopra****)***

