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.

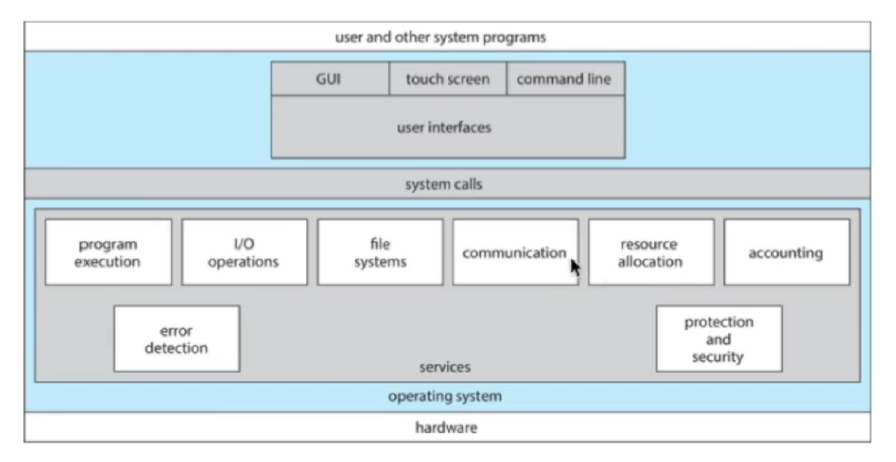
**xv6:**

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 attributi dei 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…

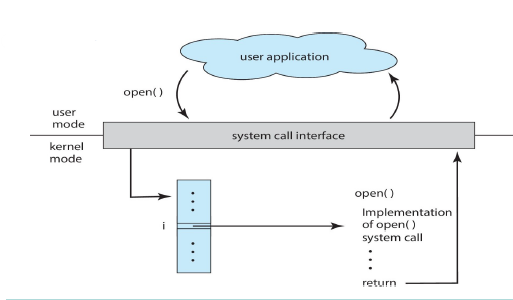
1. **Comunicazione**

* 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***

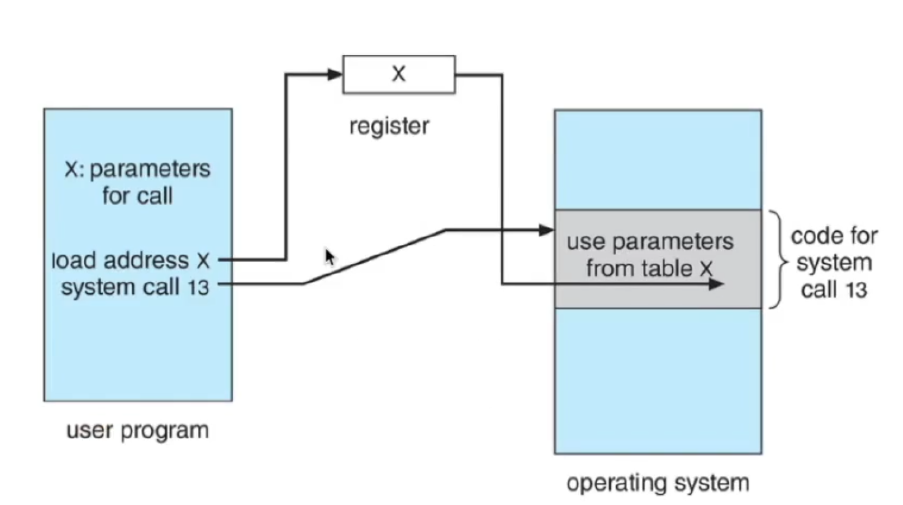
Il sistema operativo associa un numero o un ID ad ogni system call ed una tabella nello spazio del kernel che dice per ogni numero di system call, cosa deve essere fatto. Quando chiamiamo una system call tipicamente avviene uno switching da user level a kernel level, passaggio che avviene attraverso 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.



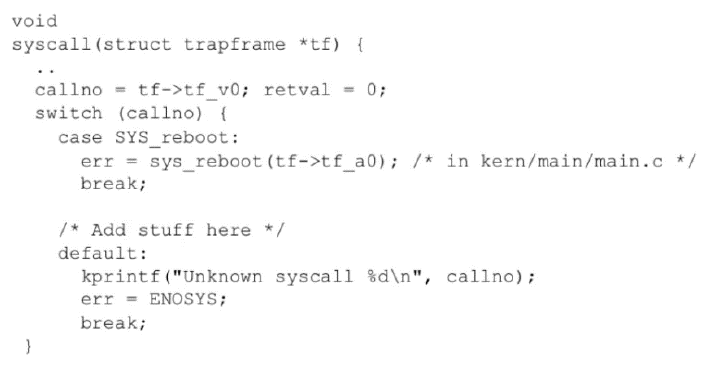
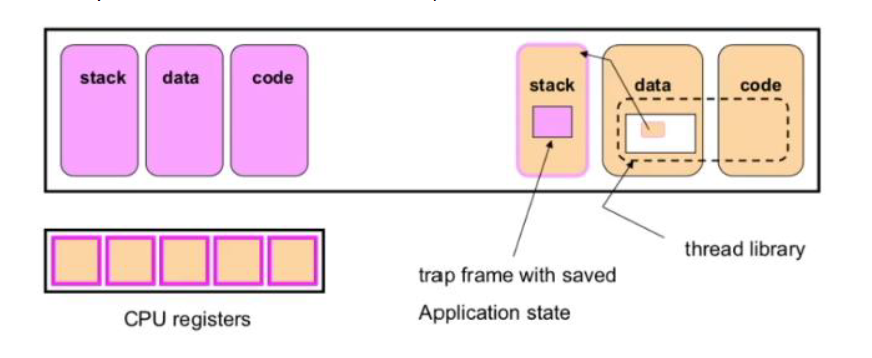
***System calls OS161***

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** (struttura che serve per salvare lo stato della CPU) sullo stack del kernel del thread e i registri della CPU sono disponibili per mantenere lo stato di esecuzione del kernel.

***OS161 MIPS System call Handler***

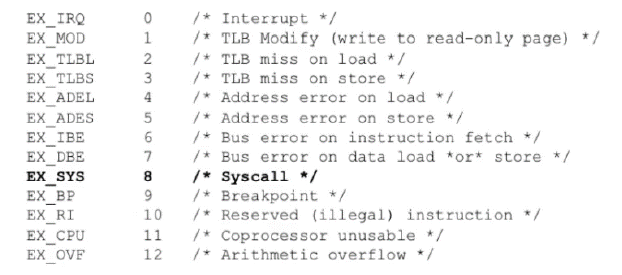
La funzione che viene attivata quando il gestore di eccezioni si accorge che si tratta di una System call è la seguente:



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

La funzione va a guardare uno dei contenuti del trapframe che il campo tf\_v0 e cioè il call number, il numero della System Call.

Il numero syscall è:



***OS161 System Calls:***

* **exit** - ***void exit(int status)*** *-* terminare il processo
* **chdir** - ***int chdir(const char \*pathname) -*** cambia la directory corrente
* **close** - ***int close(int fd) -***chiude il file
* **dup2** - ***int dup2(int oldfd, int newfd) -***clonare file descriptor
* **execv** - ***int execv(const char \*program, char \*\*args)*** *-* eseguire un programma
* **fork - *pid fork(void)*** *-* copiare il processo corrente
* **fstat** – ***int fstat (int fd, struct stat \*statbuf) -*** ottenere informazioni sullo stato del file
* [**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)** - scarica i dati del filesystem per un file specifico su disco
* **Ftruncate** – **int ftruncate(int *fd*, off\_t *filesize*)** - imposta la dimensione di un file
* **\_getcwd** - ottiene il nome della directory di lavoro corrente (backend)
* **getdirentry** – **int getdirentry(int *fd*, char \**buf*, size\_t *buflen***) - legge il nome del file dalla directory
* [**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) -** ottiene l'ID del processo
* [**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
* **Link** – **int link(const char \*oldname, const char \*newname)** - crea un collegamento fisico a un file
* **Iseek** - **off\_t lseek(int *fd*, off\_t *pos*, int *whence***) - cambia la posizione corrente nel file descriptor
* **Istat** – **int lstat(const char \*pathname, struct stat \*statbuf**) - ottenere informazioni sullo stato del file
* **mkdir** – **int mkdir(const char \**pathname*, mode\_t *mode*)** - creare una directory
* **open – int open(const char \*filename, int flags)** - aprire un file
* **pipe** – **int pipe(int \**fds*)** - creare un oggetto tubo
* **read** - **ssize\_t read(int *fd*, void \**buf*, size\_t *buflen*)** - leggere i dati dal file
* **readlink – int readlink(const char \**path*, char \**buf*, size\_t *len*)** - recuperare il contenuto del collegamento simbolico
* **reboot** – **int reboot(int *code*)** - riavviare o arrestare il sistema
* **remove** – **int remove(const char \**pathname*)** - eliminare (scollegare) un file
* **rename** – **int rename(const char \**oldname*, const char \**newname*)** - rinominare o spostare un file
* **rmdir** – **int rmdir(const char \**pathname*)** - rimuovere la directory
* **sbrk** - **void \* sbrk(intptr\_t *amount*)** - imposta l'interruzione del processo (alloca memoria)
* **stat** – **int stat(const char \**pathname*, struct stat \**statbuf*)** - ottenere informazioni sullo stato del file
* **symlink** – **int symlink(const char \**oldname*, const char \**linkname*)** - creare un collegamento simbolico
* **sync – void sync(void)** - scaricare i dati del filesystem sul disco
* **\_time** - ottenere l'ora del giorno
* **waitpid - pid\_t waitpid(pid\_t *pid*, int \**status*, int *options*)** - attendere l'uscita del processo
* **write - ssize\_t write(int *fd*, const void \**buf*, size\_t *buflen*)** - scrivere i dati su file

***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 xv6 1 2 4 8***

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

(La shell è un programma user che ci illustra il potere dell’interfaccia di system call andando a leggere i comandi forniti dall’utente ed eseguendoli.)

Le chiamate di sistema in xv6 sono semplificate e si ispirano al design di Unix V6.



AGGIUNGERE “come vengono chiamate le sys call su xv6”, file syscall.c e syscall.h + pag 45 book

+ trap, file kernel/proc.h

+ file kernel/trampoline.S (assembly)

+ (assembly) anche di OS161

***Xv6 system calls:***

* **fork** - ***int fork()*** *-*  creazione processo, ritorna il PID del figlio
* **exit** - ***int exit(int status) -***terminare il processo corrente, nessuno tipo di ritorno
* **wait** –***int wait(int \*status) –***attende l’uscita del processo figlio, ritorna il PID del figlio
* **Kill -** –***int kill(int pid) –***termina il processo, ritorna 0 o -1 per errori
* **getpid** –***int getpid() –***ritorna l’id del processo corrente
* **sleep** –***int wait(int \*status) –***pausa per n clock ticks
* **exec** –***int exec(char \*file, char argv[]) –***carica un file e lo esegue con gli argomenti, vi è un ritorno solo in caso di errore
* **sbrk** – ***char \*sbrk(int n) –***aumenta la memoria del processo di n byte e ritorna l’inizio della nuova memoria
* **open** –***int open(char \*file, int flags) –***apre un file, i flag indicano lettura o scrittura, ritorna un fd(file descriptor)
* **write** –***int write(int fd, char \*buf, int n) –***scrive n bytes da *buf* nel file descriptor fd, ritorna n
* **read** –***int read(int fd, char \*buf, int n) –*** legge n bytes in *buf,* ritorna numero read o se è la fine del file
* **close** –***int close(int fd) –***rilascia il file aperto fd
* **dup** –***int dup(int fd) –***ritorna un nuovo file descriptor
* **pipe** –***int pipe(int p[]) –***crea una pipe e inserisce read/write file descriptors in p[0] e p[1]
* **chdir** –***int chdir(char \*dir) –***cambia la directory corrente
* **mkdir** –***int mkdir(char \*dir) –***crea una nuova directory
* **mknod** –***int mknod(char \*file, int, int) –***crea un device file
* **fstat** –***int fstat(int fd, struct stat \*st) –***memorizza le info su un file aperto in \*st
* **stat** –***int stat(int fd, struct stat \*st) –***memorizza le info su uno specifico file in \*st
* **link** –***int link(char \*file1, char \*file2) –***crea un nuovo nome per il file 1
* **unlink** –***int unlink(char \*file) –***rimuove 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***

**exit():** La chiamata di sistema **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():** La chiamata di sistema **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():** La chiamata di sistema **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):** La chiamata di sistema **fork()** 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:** la funzione forza la scrittura su disco di tutti i buffer del filesystem sporco e dello stato del filesystem sporco.

**time:** è una funzionalità che 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.*

In sintesi, mentre OS161 si concentra su un set più ampio e complesso di chiamate di sistema per fornire una comprensione dettagliata dei sistemi operativi, xv6 adotta un approccio più semplificato, concentrandosi su concetti fondamentali e seguendo da vicino il design di Unix V6.