Analisi comparativa tra i sistemi operativi OS161 e Xv6

Autori: Baracco Thomas s308722, Masciari Luca s317624, Valeriano Carlos s308747

Professoressa: Sarah Azimi

Programmazione di Sistema 2022/23 (02GRSOV)

Corso di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

Politecnico di Torino

Sommario

Introduzione

I sistemi operativi rivestono un ruolo cruciale nell'ambito dei sistemi informatici, in quanto gestiscono le risorse hardware e forniscono un'interfaccia fondamentale tra l'hardware stesso e le applicazioni software. La scelta di un sistema operativo adatto a un determinato scenario o progetto può avere un impatto significativo sulla performance, la sicurezza e l'efficienza del sistema. In questo contesto, il presente progetto si propone di condurre un'analisi comparativa approfondita tra due sistemi operativi didattici e open-source: OS161 e Xv6.

L'obiettivo primario di questa analisi è valutare la presenza e l'efficacia di diverse funzionalità e caratteristiche essenziali all'interno di OS161 e Xv6. Tra le funzioni da considerare rientrano le system calls, i meccanismi di sincronizzazione, nonché la gestione dei file system. Questa analisi non solo aiuterà a comprendere le capacità intrinseche di ciascun sistema operativo, ma consentirà anche di valutare la fattibilità e l'usabilità dell'implementazione, modifica o interazione di tali caratteristiche rispetto a OS161.

Nei prossimi capitoli, esamineremo in dettaglio ciascuna delle caratteristiche menzionate, evidenziando le differenze e le similitudini tra OS161 e Xv6. Inoltre, esploreremo la documentazione tecnica e le risorse disponibili per determinare la praticità di utilizzo di ciascuno di questi sistemi operativi in un contesto didattico.

Descrizione generale di OS161 e Xv6

OS161

OS161 è un sistema operativo open-source che è stato sviluppato come parte di un progetto accademico all'università di Harvard. È stato sviluppato principalmente con il linguaggio C e offre una versione semplificata di un sistema operativo basato su Unix. È stato concepito per fornire agli studenti un'opportunità pratica per imparare i concetti fondamentali della progettazione e dello sviluppo di sistemi operativi.

La sua struttura modulare consente agli studenti di esplorare e comprendere i principi fondamentali della gestione dei processi, della gestione della memoria e della comunicazione tra processi. OS161 include un emulatore del sistema, che consente di eseguire il sistema operativo in un ambiente virtuale, rendendo una base solida per l'apprendimento dei concetti chiave della progettazione dei sistemi operativi, rendendolo un'opzione ideale per scopi educativi.

Xv6

Xv6 è un altro sistema operativo open-source, ma ha una storia e un obiettivo diversi rispetto a OS161. Xv6 è stato sviluppato presso il MIT (Massachusetts Institute of Technology) ed è un sistema operativo scritto in linguaggio C. La sua origine risale al sistema operativo Unix Version 6 (da cui deriva il nome "Xv6"), e l'obiettivo principale di Xv6 è fornire una versione semplificata e moderna del sistema operativo Unix.

Xv6 è stato progettato per scopi didattici e di ricerca, ma differisce da OS161 nel senso che mira a emulare il comportamento di un sistema Unix classico. Questo lo rende una scelta ideale per coloro che desiderano imparare o condurre ricerche nel campo dei sistemi operativi Unix-like. Xv6 offre una solida base per comprendere la gestione dei processi, i file system, il meccanismo di gestione della memoria e altri aspetti critici dei sistemi operativi Unix.

Per riassumere, mentre OS161 è mirato principalmente all'educazione e fornisce una versione semplificata di un sistema operativo, Xv6 si concentra sull'imitazione di un sistema operativo Unix classico, consentendo agli studenti e ai ricercatori di esplorare gli aspetti più specifici dei sistemi Unix. Entrambi i sistemi hanno le proprie peculiarità e sono utili per scopi diversi.

Metriche di valutazione

In questo capitolo, esamineremo dettagliatamente le diverse funzionalità e caratteristiche che costituiscono la base di un sistema operativo, valutando come OS161 e Xv6 le implementano e come differiscono nel loro approccio e funzionamento.

System Calls

Uno degli aspetti chiave dei sistemi operativi è la loro capacità di fornire un'interfaccia tra le applicazioni software e l'hardware sottostante. Questa interfaccia è resa possibile grazie alle system calls, un insieme di funzioni speciali che consentono alle applicazioni di accedere ai servizi forniti dal kernel del sistema operativo. Le system calls fungono da ponte tra il livello utente e il livello kernel, consentendo l'esecuzione di operazioni fondamentali come la lettura/scrittura su disco, la gestione dei processi, la comunicazione tra processi, la gestione dei permessi e molto altro.

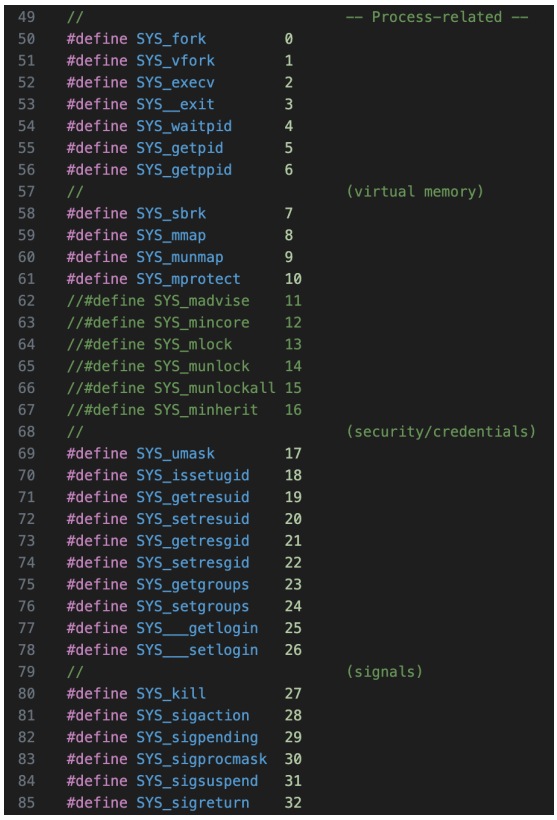
1. Funzionamento in OS161

In OS161 implementato su architettura MIPS, non ci sono differenze tra i concetti di system call, eccezioni e interrupt. Quando una di queste tre avviene, viene chiamato un determinato "exception handler" e l'hardware si occupa di aggiungere un codice per indicare la ragione per cui tale handler è stato chiamato. OS161 poi tramite il mips trap è in grado di chiamare la corretta funzione in base al codice fornito dall'hardware: l'handler delle system calls (mips syscall) nel caso di una system call.

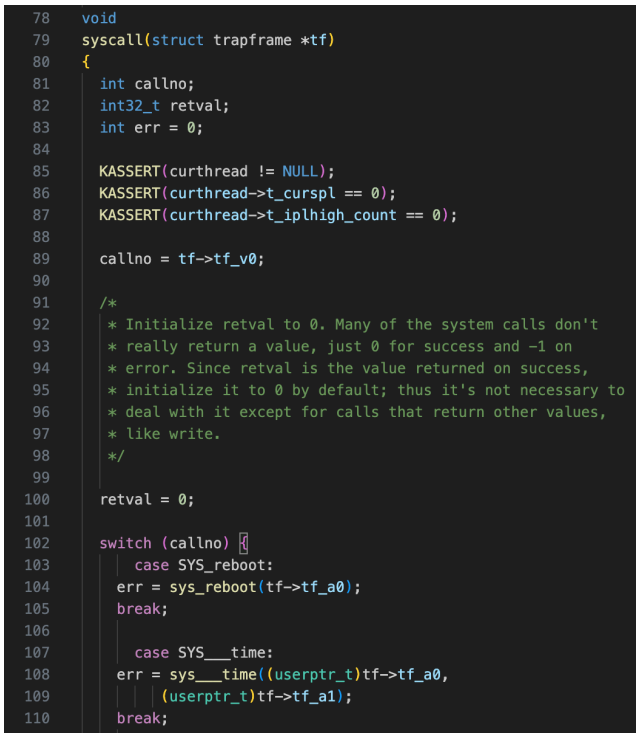
```
/* called only from assembler, so not declared in a header */
void mips_trap(struct trapframe *tf);


/* Names for trap codes */
#define NTRAPCODES 13
static const char *const trapcodenames[NTRAPCODES] = {
  "Interrupt",
  "TLB modify trap",
  "TLB miss on load",
  "TLB miss on store",
  "Address error on load",
  "Address error on store",
  "Bus error on code",
  "Bus error on data",
  "System call",
  "Break instruction",
  "Illegal instruction",
  "Coprocessor unusable",
  "Arithmetic overflow",
};
```

Ogni qualvolta una system call viene chiamata, l'esecuzione passa da "user mode" a "kernel mode" e lo stato della CPU deve essere salvato tramite il trap frame. Il trap frame in OS161 contiene informazioni relative allo stato del processo al momento dell'intercettazione di un'eccezione o di una trap. Queste informazioni includono ma non si limitano a stack pointer, program counter, valori dei registri, valore del processo. In questo modo, fintanto che il kernel gestisce una system call, lo stato della CPU è salvato nel trap frame che si trova nello stack del kernel thread. Una volta che la system call termina la sua esecuzione, lo stato viene ripristinato in modo che il processo possa riprendere l'esecuzione in maniera sicura e coerente.

Quando una system call viene chiamata, un handler chiamato syscall riceve come parametro un puntatore al trap frame che al suo interno contiene il codice della chiamata di sistema. Questo handler gestisce tale parametro con uno switch case per chiamare la corretta funzione associata.



```
49  //                          -- Process-related --
50  #define SYS_fork        0
51  #define SYS_vfork       1
52  #define SYS_execv       2
53  #define SYS__exit       3
54  #define SYS_waitpid     4
55  #define SYS_getpid      5
56  #define SYS_getppid     6
57  //                          (virtual memory)
58  #define SYS_sbrk        7
59  #define SYS_mmap        8
60  #define SYS_munmap      9
61  #define SYS_mprotect    10
62  //#define SYS_madvise   11
63  //#define SYS_mincore   12
64  //#define SYS_mlock     13
65  //#define SYS_munlock   14
66  //#define SYS_munlockall 15
67  //#define SYS_minherit  16
68  //                          (security/credentials)
69  #define SYS_umask       17
70  #define SYS_issetugid   18
71  #define SYS_getresuid   19
72  #define SYS_setresuid   20
73  #define SYS_getresgid   21
74  #define SYS_setresgid   22
75  #define SYS_getgroups   23
76  #define SYS_setgroups   24
77  #define SYS___getlogin  25
78  #define SYS___setlogin  26
79  //                          (signals)
80  #define SYS_kill        27
81  #define SYS_sigaction   28
82  #define SYS_sigpending  29
83  #define SYS_sigprocmask 30
84  #define SYS_sigsuspend  31
85  #define SYS_sigreturn   32
```



```
78  void
79  syscall(struct trapframe *tf)
80  {
81    int callno;
82    int32_t retval;
83    int err = 0;
84
85    KASSERT(curthread != NULL);
86    KASSERT(curthread->t_curspl == 0);
87    KASSERT(curthread->t_iplhigh_count == 0);
88
89    callno = tf->tf_v0;
90
91    /*
92     * Initialize retval to 0. Many of the system calls don't
93     * really return a value, just 0 for success and -1 on
94     * error. Since retval is the value returned on success,
95     * initialize it to 0 by default; thus it's not necessary to
96     * deal with it except for calls that return other values,
97     * like write.
98     */
99
100   retval = 0;
101
102   switch (callno) {
103       case SYS_reboot:
104     err = sys_reboot(tf->tf_a0);
105     break;
106
107       case SYS___time:
108     err = sys___time((userptr_t)tf->tf_a0,
109                      (userptr_t)tf->tf_a1);
110     break;
```

Quando una system call viene chiamata, un handler chiamato syscall riceve come parametro un puntatore al trap frame che al suo interno contiene il codice della chiamata di sistema. Questo handler gestisce tale parametro con uno switch case per chiamare la corretta funzione associata.

2. Implementazione in OS161

OS161 offre un insieme di system calls standard, come “read”, “write”, “exit”, fork” e molte altre, tuttavia, l’implementazione di queste non è presente in quanto lasciata agli studenti. Quando un programma utente effettua una system call, il kernel di OS161 è responsabile di gestirla. Il kernel contiene un elenco di funzioni specifiche per ciascuna system call supportata. Ad esempio, la system call “read” viene gestita da una funzione denominata sys_read. Durante l’esecuzione di una system call, i parametri necessari vengono passati dallo spazio utente allo spazio kernel tramite la seguente convenzione: come con le chiamate alle funzioni ordinarie ad architettura MIPS, i primi quattro argomenti a 32 bit sono passati nei registri degli argomenti a0-a3. Gli argomenti a 64 bit sono passati in coppie di registri allineati, ovvero a0/a1 o a2/a3. Ciò significa che se il primo argomento è a 32 bit e il secondo a 64 bit, il registro a1 rimarrà inutilizzato. Il codice della system call viene passato nel registro v0. In caso di successo della chiamata, il valore restituito viene passato nuovamente al registro v0 o nei registri v0 e v1 se a

64 bit. In caso di errore, il codice di errore viene sempre restituito tramite il registro v0 e il registro a3 viene impostato a 1 per indicare il fallimento. Al termine dell'esecuzione della system call, il program counter memorizzato nel trap frame deve essere incrementato di un'istruzione (4 byte).

```
163   if (err) {
164     /*
165      * Return the error code. This gets converted at
166      * userlevel to a return value of -1 and the error
167      * code in errno.
168      */
169     tf->tf_v0 = err;
170     tf->tf_a3 = 1;      /* signal an error */
171   }
172   else {
173     /* Success. */
174     tf->tf_v0 = retval;
175     tf->tf_a3 = 0;      /* signal no error */
176   }
177
178   /*
179    * Now, advance the program counter, to avoid restarting
180    * the syscall over and over again.
181    */
182
183   tf->tf_epc += 4;
```

Per implementare una nuova system call è sufficiente dichiarare il prototipo della funzione nel file "kern/include/syscall.h" e la sua definizione in un altro file nella cartella "kern/syscall".

3. System calls supportate in OS161

OS161 supporta una varietà di system calls (120 per l'esattezza) presenti nella maggior parte dei sistemi Unix. In questa sezione, citeremo solo le due system calls già implementate in OS161, nonché quelle che sono state implementate da noi nell'arco del corso di Programmazione di Sistema.

- `sys_reboot(int code)`: questa system call permette di riavviare il sistema operativo con il codice specificato. È spesso utilizzata per riavviare il sistema in modalità debug p per testare specifici scenari di riavvio.
- `sys___time(userptr_t user_seconds, userptr_t user_nanoseconds)`: restituisce il tempo trascorso dal boot del sistema in secondi e nanosecondi. È utile per il monitoraggio del tempo di esecuzione dei processi.
- `sys_write(int fd, userptr_t buf_ptr, size_t size)`: consente di scrivere dati da un buffer su un descrittore di file specificato tramite la funzione putch().
- `sys_read(int fd, userptr_t buf_ptr, size_t size)`: permette la lettura di dati da un descrittore di file in un buffer tramite la funzione getch().
- `sys__exit(int status)`: questa system call permette di terminare il processo corrente con uno stato di uscita specificato.

- `sys_waitpid(pid_t pid, userptr_t stats_ptr, int options)`: permette di sospendere il processo corrente fino a quando un processo figlio specifico termina.
- `sys_getpid(void)`: restituirà l'identificatore di processo (pid) del processo corrente.
- sys_fork(struct trapframe *ctf, pid_t *retval): implementata per creare un nuovo processo duplicando il processo corrente.
- `int sys_open(userptr_t path, int openflags, mode_t mode, int *errp)`: implementata per aprire un file specificato dal nome e restituirà un file descriptor.
- `int sys_close(int fd)`: implementata per chiudere un file descriptor.

4. Funzionamento in Xv6

Similmente a quanto avviene in OS161, tre sono gli eventi che richiedono alla CPU di interrompere il suo ordinario funzionamento e di passare il controllo a una determinata sezione di codice per gestire l'evento: system calls, eccezioni e interrupt. Il termine generico utilizzato per indicare uno di questi tre eventi è *trap*. Xv6 gestisce tutti i trap nel kernel e non sono mai trasferiti al segmento user. La gestione di un trap in Xv6 avviene in 4 fasi:

- azioni hardware eseguite dalla CPU RISC-V
- istruzioni assembly che preparano l'esecuzione del codice C del kernel
- funzione C (handler) che determina come gestire il trap
- esecuzione della system call o della service routine del driver di un dispositivo

Benché ci sia una certa somiglianza tra i tre tipi di trap, Xv6 separa il codice per gestire tre casi distinti: traps provenienti da user space, traps provenienti da kernel space e traps causati da timer interrupts.

Ogni CPU RISC-V ha un set di registri di controllo per dire alla CPU come gestire un trap che il kernel può leggere per avere informazioni sul trap che è avvenuto. Tra questi registri i più importanti sono:

- `stvec`: In cui il kernel scrive l'indirizzo del suo trap handler; il RISC-V salta a questo indirizzo per gestire un trap.
- `sepc`: In cui il RISC-V salva il program counter quando avviene un trap per poter riprendere l'esecuzione del programma una volta gestito il trap.
- `scause`: In cui il RISC-V salva un numero che indica la ragione che ha causato il trap.
- `sscratch`: Il trap handler utilizza questo registro come flag per evitare di sovrascrivere registri user prima di averli salvati.
- `sstatus`: Utilizza un bit chiamato SIE per controllare se gli interrupt di dispositivo sono attivi e utilizza un altro bit chiamato SPP per indicare se un trap arriva da una modalità user o supervisor.

Ogni qualvolta il RISC-V deve forzare un trap, l'hardware si occupa di svolgere i seguenti passaggi per tutti i tipi di trap a eccezione dei timer interrupts:

1. Se il trap è un device interrupt e il bit SIE è a zero, non si fa nulla di quanto segue.
2. Disabilita gli interrupt azzerando il bit SIE in `sstatus`.
3. Copia il `pc` nel registro `sepc`.
4. Salva la modalità corrente (user o supervisor) nel bit SPP in `sstatus`.
5. Imposta `scause` per indicare il motivo del trap.
6. Imposta la modalità supervisor.
7. Copia `stvec` nel `pc`.
8. Riprendi l'esecuzione dal nuovo `pc`.

Per quanto riguarda i trap generati dal codice utente, possono verificarsi durante l'esecuzione di una system call (istruzione ecall), quando si verifica un comportamento illegale o se si verifica un'interrupt da parte di un dispositivo.

Un vincolo significativo nella progettazione della gestione dei trap in Xv6 è che l'hardware RISC-V non esegue lo switch fra page tables quando avviene un trap. Questo implica che l'indirizzo del trap handler che si trova in `stvec` deve avere un valido mapping nella user page table dal momento che quella è la page table quando il codice del trap handler viene eseguito. Per soddisfare questi requisiti, Xv6 utilizza la cosiddetta *trampoline* page che contiene il codice di gestione dei trap a cui `stvec` fa riferimento. Questa pagina è mappata in ogni page table di ogni singolo processo all'indirizzo `TRAMPOLINE`, situato nella parte superiore dello spazio degli indirizzamenti virtuali. La trampoline page è anche mappata all'indirizzo `TRAMPOLINE` nella kernel page table. Poiché la trampoline page è mappata nella user page table, i trap possono iniziare a eseguire in modalità supervisor. Dal momento che la trampoline page si trova allo stesso indirizzo nello spazio degli indirizzi del kernel space, il trap handler può continuare la sua esecuzione dopo lo switch tra page tables.

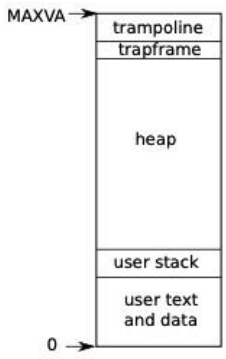


Figure 2.3: Layout of a process's virtual address space

5. Implementazione in Xv6

Il processo di esecuzione di una system call, come esemplificato nell'esecuzione della system call exec, inizia quando il codice di `initcode.S` chiama la system call exec come rappresentato in figura.

```
1   # Initial process that execs /init.
2   # This code runs in user space.
3
4   #include "syscall.h"
5
6   # exec(init, argv)
7   .globl start
8   start:
9           la a0, init
10          la a1, argv
11          li a7, SYS_exec
12          ecall
13
14  # for(;;) exit();
15  exit:
16          li a7, SYS_exit
17          ecall
18          jal exit
19
20  # char init[] = "/init\0";
21  init:
22    .string "/init\0"
23
24  # char *argv[] = { init, 0 };
25  .p2align 2
26  argv:
27    .long init
28    .long 0
```

In questa illustrazione, `initcode.S` inserisce gli argomenti per exec nei registri a0 e a1, e inserisce il numero della system call in a7. I numeri delle system call corrispondono alle voci nell'array `syscalls`, una tabella di puntatori a funzioni.

```
105 // An array mapping syscall numbers from syscall.h
106 // to the function that handles the system call.
107 static uint64 (*syscalls[])(void) = {
108 [SYS_fork]    sys_fork,
109 [SYS_exit]    sys_exit,
110 [SYS_wait]    sys_wait,
111 [SYS_pipe]    sys_pipe,
112 [SYS_read]    sys_read,
113 [SYS_kill]    sys_kill,
114 [SYS_exec]    sys_exec,
115 [SYS_fstat]   sys_fstat,
116 [SYS_chdir]   sys_chdir,
117 [SYS_dup]     sys_dup,
118 [SYS_getpid]  sys_getpid,
119 [SYS_sbrk]    sys_sbrk,
120 [SYS_sleep]   sys_sleep,
121 [SYS_uptime]  sys_uptime,
122 [SYS_open]    sys_open,
123 [SYS_write]   sys_write,
124 [SYS_mknod]   sys_mknod,
125 [SYS_unlink]  sys_unlink,
126 [SYS_link]    sys_link,
127 [SYS_mkdir]   sys_mkdir,
128 [SYS_close]   sys_close,
129 };
```

L'istruzione `ecall` in `initcode.S` provoca un trap nel kernel, dando inizio all'esecuzione di `uservec`, `usertrap` e poi `syscall`.

La funzione `syscall` recupera il numero della system call dal valore salvato in a7 nel trapframe e lo utilizza come indice per accedere a `syscalls`. Per la system call exec, a7 contiene `SYS_exec` portando alla chiamata della funzione di implementazione della system call, `sys_exec`.

```
131  void
132  syscall(void)
133  {
134    int num;
135    struct proc *p = myproc();
136
137    num = p->trapframe->a7;
138    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139      // Use num to lookup the system call function for num, call it,
140      // and store its return value in p->trapframe->a0
141      p->trapframe->a0 = syscalls[num]();
142    } else {
143      printf("%d %s: unknown sys call %d\n",
144              p->pid, p->name, num);
145      p->trapframe->a0 = -1;
146    }
147  }
148
```

Al termine dell'esecuzione di `sys_exec`, `syscall` registra il valore restituito in `p->trapframe->a0`. Ciò determinerà che la chiamata originale di `exec()` nello spazio utente restituirà tale valore, poiché la convenzione di chiamata in C su RISC-V colloca i valori di ritorno nel registro a0. Le system call restituiscono convenzionalmente numeri negativi per indicare errori e numeri maggiori o uguali a zero per indicare successo. Se il numero della system call è invalido, `syscall` stampa un errore e restituisce -1.

Le implementazioni delle system call nel kernel di Xv6 devono recuperare gli argomenti passati dal codice utente, originariamente posizionati nei registri secondo la convenzione di chiamata in C di RISC-V. Il codice del trap del kernel salva questi registri nel trap frame del processo corrente. Alcune system call passano puntatori come argomenti, presentando sfide come potenziali errori del programma utente o vulnerabilità di sicurezza.
Il kernel implementa funzioni come `argint`, `argaddr` e `argfd` per recuperare gli argomenti della struttura trap del processo. Quando si trattano puntatori, ad esempio nella system call exec, il kernel deve gestire il trasferimento sicuro dei dati tra gli spazi di memoria utente e kernel. Esempi di queste funzioni sono `fetchstr` e `copyinstr`, quest'ultima effettua la copia di stringhe dalla memoria virtuale utente alla memoria del kernel in modo sicuro, verificando anche la validità degli indirizzi forniti dall'utente.

```
22  // Fetch the nul-terminated string at addr from the current process.
23  // Returns length of string, not including nul, or -1 for error.
24  int
25  fetchstr(uint64 addr, char *buf, int max)
26  {
27    struct proc *p = myproc();
28    if(copyinstr(p->pagetable, buf, addr, max) < 0)
29      return -1;
30    return strlen(buf);
31  }
```

Inoltre, le funzioni come `copyinstr` utilizzano meccanismi come `walkaddr` per garantire la corretta manipolazione degli indirizzi fisici e virtuali, e copyout è utilizzata per copiare dati dal kernel a un indirizzo fornito dall'utente. In questo modo, Xv6 affronta in modo sicuro le sfide legate alla manipolazione degli argomenti delle system call e alla gestione dei puntatori in presenza di potenziali comportamenti dannosi o errori da parte dei programmi utente.

6. System calls supportate in Xv6

Xv6 supporta diverse system call Unix standard che trovano già un'implementazione nel codice sorgente a differenza di OS161 in cui la maggior parte delle system call deve essere progettata e sviluppata dagli studenti. Tra le principali system call fornite da Xv6 possiamo individuare quelle illustrate nell'immagine sottostante:

```
1   // System call numbers
2   #define SYS_fork    1
3   #define SYS_exit    2
4   #define SYS_wait    3
5   #define SYS_pipe    4
6   #define SYS_read    5
7   #define SYS_kill    6
8   #define SYS_exec    7
9   #define SYS_fstat   8
10  #define SYS_chdir   9
11  #define SYS_dup    10
12  #define SYS_getpid 11
13  #define SYS_sbrk   12
14  #define SYS_sleep  13
15  #define SYS_uptime 14
16  #define SYS_open   15
17  #define SYS_write  16
18  #define SYS_mknod  17
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
```

7. **Confronto tra OS161 e Xv6**

Le principali similitudini tra la gestione di system call nei due sistemi operativi analizzati sono le seguenti:

- Entrambi OS161 e Xv6 gestiscono system call, eccezioni e interrupt attraverso un meccanismo di trap.
- Quando una system call viene chiamata, l'esecuzione passa da "user mode" a "kernel mode" in entrambi i sistemi operativi.
- Entrambi utilizzano un trap frame per salvare lo stato della CPU quando si verifica un'eccezione o una trap durante l'esecuzione di una system call. Il trap frame contiene informazioni cruciali come stack pointer, program counter e valori dei registri.
- In entrambi i casi, l'implementazione delle system call prevede l'uso di un handler specifico per gestire il trap associato alla system call e viene utilizzato uno switch case per determinare quale funzione di sistema chiamare in base al codice della system call.
- Entrambi seguono una convenzione per il passaggio dei parametri delle system call, utilizzando registri specifici per i primi argomenti e stack per argomenti successivi.

Le principali differenze tra i due sistemi operativi nell'implementazione di system call sono le seguenti:

- OS161 offre un insieme di system calls standard, ma la loro implementazione è lasciata agli studenti mentre Xv6 fornisce già un set di system calls Unix standard implementate direttamente nel codice sorgente del kernel.
- Xv6 separa la gestione di trap provenienti da user space, kernel space e timer interrupts, mentre OS161 sembra gestire tutti i tipi di trap in un unico handler di eccezioni.
- Xv6 utilizza una "trampoline page" per gestire i trap, garantendo che l'indirizzo del trap handler nel kernel abbia un valido mapping nella user page table. In OS161 il kernel è mappato in una porzione di virtual address space per ogni processo.
- Xv6 affronta in modo sicuro le sfide legate alla manipolazione degli argomenti delle system call e alla gestione dei puntatori in presenza di potenziali comportamenti dannosi o errori da parte dei programmi utente, mentre non ci sono dettagli simili per OS161.

Meccanismi di sincronizzazione

Un sistema operativo, per garantire un corretto funzionamento, necessita di strumenti per gestire la concorrenza dei processi e dei thread. Infatti, in un ambiente concorrente e cooperante, le risorse sono utilizzate in modo condiviso dai processi e dai thread, il che potrebbe portare il sistema in stati inconsistenti. Problemi tipici di sincronizzazione, derivanti da una gestione assente o errata della concorrenza, includono race condition, deadlock e resource starvation.

Una gestione adeguata della concorrenza richiede la risoluzione del problema della sezione critica, definita come una porzione del programma che non può essere eseguita da più di un processo o thread alla volta, in un certo istante, a causa dell'accesso a risorse condivise.
Nei paragrafi successivi, verranno esaminati i meccanismi di sincronizzazione disponibili in OS161 e Xv6 che consentono la risoluzione del problema della sezione critica, causa dei sopracitati problemi.

1. Meccanismi in OS161

OS161 ha a disposizione una vasta gamma di meccanismi di sincronizzazione con caratteristiche diverse, ma tutte si basano su tecniche di basso livello come istruzioni hardware (Test-and-Set e Compare-and-Swap), barriere di memoria e variabili atomiche.

Il primo meccanismo di sincronizzazione, e anche il più semplice, è il mutex lock. Anche chiamato spinlock, il mutex lock permette la protezione della sezione critica e previene le race condition. Il suo funzionamento è il seguente:

- un processo deve acquisire il lock prima di entrare nella sezione critica usando l'apposita funzione di acquisizione.
- rilasciare il lock quando il processo esce dalla sezione critica tramite l'apposita funzione di rilascio.

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Lo spinlock possiede una variabile booleana che indica se il lock è disponibile o meno. Il mutex lock richiede busy waiting, che spreca cicli di CPU, quindi è una buona soluzione nei casi di attese di durata ridotta.

Un altro strumento disponibile per gestire la concorrenza è il semaforo, che offre modi più sofisticati ai processi per sincronizzare le loro attività.
Il suo funzionamento è equivalente al mutex lock, tuttavia può far entrare nella sezione critica più processi o thread. Infatti un semaforo è una variabile intera che rappresenta il numero massimo di processi o thread che possono entrare contemporaneamente nella propria sezione critica.

Le funzioni atomiche standard, oltre all'inizializzazione, che controllano il suo funzionamento sono:

- wait: Controlla che il semaforo sia maggiore di zero e lo decrementa.
- signal: Incrementa il semaforo.

Ci possono essere due tipi di semafori che differiscono dai valori che esso può assumere:

- semaforo contatore: Il valore intero può spaziare su un dominio illimitato (nei limiti dell'hardware).
- semaforo binario: Il valore intero può assumere i valori zero o uno (simile a uno spinlock).

L'implementazione deve garantire che due processi non possano eseguire wait e signal sullo stesso semaforo allo stesso tempo, quindi necessita di accortezze per gestire la condivisione. Sono possibili diverse implementazioni; alcune presentano il problema del busy waiting, mentre altre no.

OS161 dispone di un'ulteriore primitiva di sincronizzazione chiamato lock. Esso è simile a un semaforo binario con un valore iniziale di uno. Tuttavia esso impone un altro vincolo: il thread che rilascia un lock deve essere lo stesso thread che più recentemente lo ha acquisito. Questo vincolo introduce il concetto di proprietario del lock.

Un differente meccanismo presente in OS161 è la condition variable, la quale implementa la sincronizzazione sulla base del valore dei dati anziché controllare l'accesso ai dati.
Il funzionamento prevede che un thread o processo possa sospendersi su una condition variable se il valore dei dati non è quello previsto per proseguire l'elaborazione.
Quindi, il processo si sospende sulla condition variable, rilasciando il lock della sezione critica. Quando la condition variable sarà segnalata, il processo proverà ad acquisire il lock e testare il valore dei dati per continuare l'elaborazione. Insieme alla condition variable è sempre usato un lock per proteggere i dati condivisi che vengono testati e modificati.
Le due operazioni che permettono le operazioni soprastanti sono:

- wait: Permette al processo o thread di sospendersi sulla condition variable.
- signal: Permette di segnalare la condition variable, risvegliando uno o più processi in attesa su quella condition variable.

Più implementazioni sono realizzabili, come la gestione della segnalazione, ognuna con vantaggi e svantaggi.

Un'altra primitiva di sincronizzazione è il wait channel, simile alla condition variable, che permette al processo o thread di sospendersi associato a un wait channel, chiamando la funzione logica sleep, ed essere risvegliato quando viene chiamata la funzione logica wake sul wait channel associato al thread o processo. Una differenza rispetto alla condition variable è l'utilizzo degli spinlock rispetto ai lock.

2. Implementazione in OS161

La quasi totalità dei meccanismi di sincronizzazione supportati da OS161 sono già implementati e nel seguito verranno presentate le loro realizzazioni.

Lo spinlock, o mutex lock, è realizzato per mezzo di una struttura: `struct spinlock` (definita in `kern/include/spinlock.h`). Questa struttura contiene la parola di memoria su cui viene effettuato lo spin e l'informazione della CPU che mantiene quel mutex lock. Oltre alle funzioni per inizializzare e ripulire lo spinlock (`spinlock_init` e `spinlock_cleanup`), le due funzioni principali che effettuano l'acquisizione e il rilascio sono la `void spinlock_acquire(struct spinlock *lk)` e la `void spinlock_release(struct spinlock *lk)`:

- `spinlock_acquire`: Effettua l'acquisizione del mutex lock tramite l'istruzione hardware Test-and-Set all'interno di un ciclo while. Tuttavia, questa implementazione soffre di busy waiting.

```
while (1) {
    /*
     * Do test-test-and-set, that is, read first before
     * doing test-and-set, to reduce bus contention.
     *
     * Test-and-set is a machine-level atomic operation
     * that writes 1 into the lock word and returns the
     * previous value. If that value was 0, the lock was
     * previously unheld and we now own it. If it was 1,
     * we don't.
     */
    if (spinlock_data_get(&splk->splk_lock) != 0) {
        continue;
    }
    if (spinlock_data_testandset(&splk->splk_lock) != 0) {
        continue;
    }
    break;
}
```

- `spinlock_release`: Effettua il rilascio dello spinlock ripulendo i campi della struct.

Tutte le funzioni sono contenute nel file `kern/thread/spinlock.c`.

I semafori sono già implementati in OS161 tramite una struttura: `struct semaphore` (definita in `kern/include/synch.h`). Questa struttura contiene una stringa per il nome del semaforo, un wait channel per gestire la coda dei thread in attesa del semaforo, uno spinlock per proteggere la sezione critica del semaforo (quindi la sua condivisione) e il contatore.
Le due funzioni principali, oltre a quelle di creazione e distruzione del semaforo (`sem_create`, `sem_destroy`), sono:

- `void P(struct semaphore *sem)`: Effettua l'operazione logica della wait. Acquisisce lo spinlock del semaforo e controlla con un ciclo se il contatore è uguale a

zero. Se è questo il caso, il thread viene inserito nel wait channel del semaforo; altrimenti, il contatore viene decrementato e il lock rilasciato.

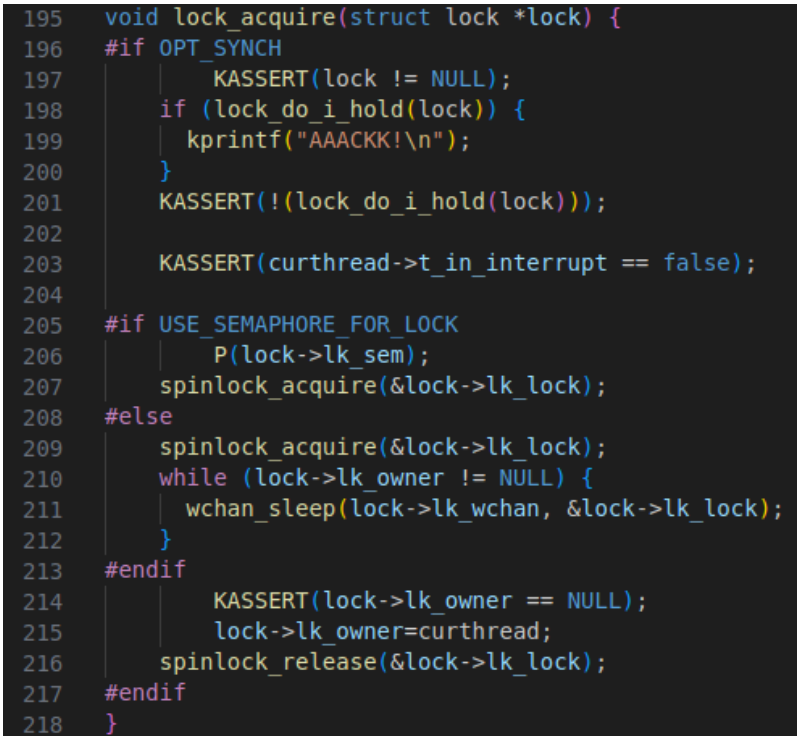
```
102     spinlock_acquire(&sem->sem_lock);
103         while (sem->sem_count == 0) {
104         wchan_sleep(sem->sem_wchan, &sem->sem_lock);
105         }
106         KASSERT(sem->sem_count > 0);
107         sem->sem_count--;
108     spinlock_release(&sem->sem_lock);
```

- `void V(struct semaphore *sem)`: Effettua l'operazione logica della signal. Dopo aver acquisito lo spinlock del semaforo, incrementa il contatore e risveglia un thread nel wait channel. Infine il lock viene rilasciato.

Questa implementazione del semaforo non soffre del busy waiting ed è funzionante in ambiente multicore.
Tutte le funzioni riguardanti il semaforo sono presenti nel file `kern/thread/synch.c`.

In OS161 il lock non è già implementato e la sua realizzazione è lasciata agli studenti. Tuttavia la sua struttura di base è già definita tramite la struttura `struct lock` (presente in `kern/include/synch.h`) e le relative funzioni sono dichiarate (`lock_create`, `lock_destroy`, `lock_acquire`, `lock_release`, `lock_do_i_hold`), ma da completare nel `kern/thread/synch.c`.
Durante il corso di Programmazione di Sistema, il lock è stato implementato in due modi alternativi: attraverso l'utilizzo dei semafori e tramite l'uso del wait channel. La `struct lock`, in entrambe le possibili realizzazioni, oltre al semaforo o al wait channel, contiene una stringa per il nome, un puntatore alla struttura di un thread (che rappresenta il proprietario del lock) e uno spinlock. Le funzioni di acquisizione e rilascio (`lock_acquire`, `lock_release`) vengono attuate tramite le funzioni specifiche del semaforo o del wait channel, con l'aggiunta del controllo sul proprietario del lock.

```
231 void lock_release(struct lock *lock) {
232 #if OPT_SYNCH
233     KASSERT(lock != NULL);
234     KASSERT(lock_do_i_hold(lock));
235     spinlock_acquire(&lock->lk_lock);
236     lock->lk_owner=NULL;
237 #if USE_SEMAPHORE_FOR_LOCK
238     V(lock->lk_sem);
239 #else
240     wchan_wakeone(lock->lk_wchan, &lock->lk_lock);
241 #endif
242     spinlock_release(&lock->lk_lock);
243 #endif
244 }
```



```
195  void lock_acquire(struct lock *lock) {
196  #if OPT_SYNCH
197          KASSERT(lock != NULL);
198      if (lock_do_i_hold(lock)) {
199        kprintf("AAACKK!\n");
200      }
201      KASSERT(!(lock_do_i_hold(lock)));
202
203      KASSERT(curthread->t_in_interrupt == false);
204
205  #if USE_SEMAPHORE_FOR_LOCK
206          P(lock->lk_sem);
207      spinlock_acquire(&lock->lk_lock);
208  #else
209      spinlock_acquire(&lock->lk_lock);
210      while (lock->lk_owner != NULL) {
211        wchan_sleep(lock->lk_wchan, &lock->lk_lock);
212      }
213  #endif
214          KASSERT(lock->lk_owner == NULL);
215          lock->lk_owner=curthread;
216      spinlock_release(&lock->lk_lock);
217  #endif
218  }
```

La condition variable è un meccanismo di sincronizzazione definito in OS161, ma senza una completa realizzazione. Durante il corso di Programmazione di Sistema, la condition variable è stata implementata tramite l'utilizzo del wait channel per gestire la coda dei thread. La struttura `struct cv` (definita in `kern/include/synch.h`) contiene una stringa per il nome della condition variable, un puntatore alla struttura di un wait channel e uno spinlock. Le funzioni principali, oltre alla creazione e alla distruzione (`cv_create`, `cv_destroy`), sono le seguenti:

- `cv_wait`: Permette di sospendersi sulla condition variable tramite la chiamata alla funzione del wait channel `wchan_sleep`.
- `cv_signal`: Permette di risvegliare un thread in attesa sulla condition variable tramite la chiamata alla funzione del wait channel `wchan_wakeone`.
- `cv_broadcast`: Permette di risvegliare tutti i thread in attesa sulla condition variable tramite la chiamata alla funzione del wait channel `wchan_wakeall`.

Tutte le funzioni riguardanti la condition variable sono nel file `kern/thread/synch.c`.

Il wait channel in OS161 è già implementato per mezzo di una struttura `struct wchan` (definita in `kern/thread/thread.c`). Questa struttura contiene una stringa per il nome del wait channel e una struttura `struct threadlist` per salvare la lista di thread in attesa nel wait channel. Oltre alle funzioni di creazione e distruzione (`wchan_create`, `wchan_destroy`), sono presenti le seguenti funzioni:

- `wchan_sleep`: Effettua la sospensione del thread sul wait channel tramite la funzione `thread_switch`, impostando lo stato del thread su sleep sul wait channel.
- `wchan_wakeone`: Effettua il risveglio di un thread in attesa, se esiste, tramite le funzioni `threadlist_remhead` (per recuperare un thread in attesa) e `thread_make_runnable` (per impostare lo stato del thread in ready).

```
1019  void
1020  wchan_wakeone(struct wchan *wc, struct spinlock *lk)
1021  {
1022      struct thread *target;
1023
1024      KASSERT(spinlock_do_i_hold(lk));
1025
1026      /* Grab a thread from the channel */
1027      target = threadlist_remhead(&wc->wc_threads);
1028
1029      if (target == NULL) {
1030          /* Nobody was sleeping. */
1031          return;
1032      }
1033
1034      /*
1035       * Note that thread_make_runnable acquires a runqueue lock
1036       * while we're holding LK. This is ok; all spinlocks
1037       * associated with wchans must come before the runqueue locks,
1038       * as we also bridge from the wchan lock to the runqueue lock
1039       * in thread_switch.
1040       */
1041
1042      thread_make_runnable(target, false);
1043  }
```

- `wchan_wakeall`: Effettua il risveglio di tutti i thread in attesa tramite un ciclo, chiamando le funzioni `threadlist_remhead` e `thread_make_runnable`.

Tutte le funzioni riguardanti il wait channel sono nel file `kern/thread/thread.c`.

3. Meccanismi in Xv6

Xv6 ha a disposizione, come OS161, una vasta gamma di meccanismi di sincronizzazione, a partire dai lock ai wait channel. Come in OS161, questi strumenti si basano su tecniche di basso livello, ma sono caratterizzati da proprietà differenti l'uno dall'altro.

Il primo meccanismo supportato da Xv6 è il mutex lock, anche chiamato spinlock. Come in OS161, permette di proteggere la sezione critica e il suo funzionamento è equivalente allo spinlock descritto per OS161, con le funzioni logiche di acquire e release.
Lo spinlock possiede una variabile intera che indica se il lock è disponibile o meno, e anch'esso, come in OS161, richiede busy waiting.

Un altro strumento disponibile in Xv6 per gestire la concorrenza è lo sleeplock. Tale meccanismo permette di proteggere la sezione critica e il suo funzionamento è equivalente a quello dello spinlock.
Lo sleeplock possiede una variabile intera che indica se il lock è disponibile o meno. Inoltre esso possiede una variabile intera che indica il proprietario del lock. A differenza dello spinlock, non soffre di busy waiting: quando viene chiamata la funzione acquire e il lock è occupato, il thread viene sospeso nello stato di sleep senza occupare la CPU.
Inoltre, la variabile intera deve essere protetta da accessi concorrenti tramite uno spinlock.

Il meccanismo di sincronizzazione wait channel in Xv6 non è propriamente realizzato come uno strumento indipendente, come in OS161. Difatti il wait channel è utilizzato come un canale su cui un processo può rimanere in attesa, diventando una sua caratteristica che indica se tale processo è in attesa su un canale, e quale, oppure no.
Le funzioni logiche che gestiscono il suo funzionamento sono la sleep, che permette di far sospendere il processo su un wait channel, e la wake, che permette di risvegliare i processi su un wait channel specifico.
Essendo il wait channel una proprietà del processo, occorre uno spinlock per gestire l'accesso condiviso ad esso.

4. Implementazione in Xv6

La totalità dei meccanismi in Xv6, illustrati nel paragrafo precedente, è già implementata e nel seguito verranno presentate le loro realizzazioni.

Lo spinlock è realizzato per mezzo di una struttura: `struct spinlock` (definita in `kernel/spinlock.h`). Questa struttura contiene una variabile intera, la quale indica se il lock è disponibile e su cui viene effettuato lo spin. Inoltre, contiene una stringa per il nome dello spinlock e un puntatore alla struttura che definisce una CPU (`struct cpu`), per tenere traccia della CPU che mantiene il lock.
Le funzioni principali, oltre a quella di inizializzazione (`initlock`), sono le seguenti:

- `void acquire(struct spinlock *lk)`: Effettua l'acquisizione del lock mediante la funzione `__sync_lock_test_and_set,` contenuta all'interno di un ciclo while. La funzione `__sync_lock_test_and_set` richiama l'istruzione hardware atomica (`amoswap`), la quale effettua un'operazione simile alla Compare-and-Swap. La funzione `__sync_synchronize` richiama una barriera di memoria, per impedire il riordino delle istruzioni, e la funzione `push_off` disattiva gli interrupt.

```
21  void
22  acquire(struct spinlock *lk)
23  {
24    push_off(); // disable interrupts to avoid deadlock.
25    if(holding(lk))
26      panic("acquire");
27
28    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
29    ;
30    __sync_synchronize();
31
32    lk->cpu = mycpu();
33  }
```

- `void release(struct spinlock *lk)`: Effettua il rilascio del lock tramite la funzione `__sync_lock_release`, la quale effettua un'assegnazione atomica. La funzione `pop_off` attiva gli interrupt.

Tutte le funzioni riguardanti lo spinlock sono nel file `kernel/spinlock.c`.

In Xv6, lo sleeplock è implementato per mezzo di una struttura: `struct sleeplock` (definita in `kernel/sleeplock.h`). Questa struttura contiene una variabile intera che indica se il lock è disponibile, uno spinlock per proteggere i campi della struttura, una stringa per il nome dello sleeplock e una variabile intera contenente il pid del processo che mantiene il lock. Le funzioni principali dello spleeplock (definite in `kernel/sleeplock.c`), oltre a quella di inizializzazione, sono:

- `void acquiresleep(struct sleeplock *lk)`: Effettua l'acquisizione del lock, testando la variabile intera dopo aver acquisito lo spinlock. Se il lock non è disponibile, il processo viene sospeso mediante la funzione `sleep`. Altrimenti, il lock viene acquisito impostando la variabile intera a uno e il pid con quello del processo.

```
21  void
22  acquiresleep(struct sleeplock *lk)
23  {
24    acquire(&lk->lk);
25    while (lk->locked) {
26      sleep(lk, &lk->lk);
27    }
28    lk->locked = 1;
29    lk->pid = myproc()->pid;
30    release(&lk->lk);
31  }
```

- `void releasesleep(struct sleeplock *lk)`: Effettua il rilascio del lock, pulendo i campi della variabile intera e del pid, e risveglia i processi sospesi sul lock tramite la funzione `wakeup`.

- `int holdingsleep(struct sleeplock *lk)`: Restituisce uno se lo sleeplock è occupato e il chiamante della funzione è il proprietario del lock, altrimenti zero.

```
43  int
44  holdingsleep(struct sleeplock *lk)
45  {
46    int r;
47
48    acquire(&lk->lk);
49    r = lk->locked && (lk->pid == myproc()->pid);
50    release(&lk->lk);
51    return r;
52  }
```

Il wait channel, come descritto nella sezione precedente, è una proprietà del processo. Infatti, nella struttura rappresentante il processo (`struct proc`, definita in `kernel/proc.h`), è presente un puntatore opaco (`void *chan`), il quale indica il canale su cui il processo è o non è in attesa.
La gestione del wait channel è effettuata tramite le seguenti funzioni:

- `void sleep(void *chan, struct spinlock *lk)`: Effettua la sospensione del processo che chiama la funzione. Acquisisce lo spinlock del processo per modificare il puntatore opaco del wait channel e imposta lo stato del processo a `SLEEPING`. Chiamando la funzione `sched`, il processo rilascia la CPU, effettuando il cambio di contesto. Le istruzioni successive vengono eseguite al risveglio del processo.

```
535  void
536  sleep(void *chan, struct spinlock *lk)
537  {
538    struct proc *p = myproc();
539    acquire(&p->lock);
540    release(lk);
541
542    p->chan = chan;
543    p->state = SLEEPING;
544
545    sched();
546
547    p->chan = 0;
548
549    release(&p->lock);
550    acquire(lk);
551  }
```

- `void wakeup(void *chan)`: Effettua il risveglio di tutti i processi in attesa sul wait channel mediante un ciclo for che scorre l'array di `struct proc` (rappresentante la lista dei processi presenti nel kernel) impostando lo stato dei processi in attesa sul wait channel a `RUNNABLE`.

```
566  void
567  wakeup(void *chan)
568  {
569    struct proc *p;
570
571    for(p = proc; p < &proc[NPROC]; p++) {
572      if(p != myproc()){
573        acquire(&p->lock);
574        if(p->state == SLEEPING && p->chan == chan) {
575          p->state = RUNNABLE;
576        }
577        release(&p->lock);
578      }
579    }
580  }
```

5. Confronto tra OS161 e Xv6

Entrambi i sistemi operativi offrono una varietà di meccanismi di sincronizzazione per gestire la concorrenza e la cooperazione dei processi in ambienti multicore. Nel seguito, per ogni meccanismo analizzato nelle sezioni precedenti, si esamineranno le similitudini e le differenze delle implementazioni nei due sistemi operativi, OS161 e Xv6:

- spinlock: Implementato sia in OS161 che in Xv6 in modo equivalente con la caratteristica del busy waiting.
- lock: Definito, ma da completare, in OS161 con il concetto di proprietario del lock. In Xv6 non è presente un meccanismo analogo, ma lo sleeplock ha il concetto di proprietario del lock in senso non stretto (qualunque processo può rilasciare il lock, non solo il proprietario).
- sleeplock: Implementato in Xv6 come complementare allo spinlock che non soffre di busy waiting. In OS161 non è presente uno strumento semplice con la medesima logica.
- semaforo: Implementato in OS161, ma non presente in Xv6.
- condition variable: Definito, ma da completare, in OS161 senza un corrispondente in Xv6.
- wait channel: Implementato in OS161 con un corrispondente più semplice in Xv6.

In generale, OS161 ha una gamma più ampia di meccanismi di sincronizzazione e di complessità differenti. Inoltre, OS161 predispone la realizzazione degli strumenti non completi definendo le basi di esso (struttura e funzioni). Invece, Xv6 possiede un numero più ridotto di meccanismi di sincronizzazione di complessità ridotta e non predispone la realizzazione degli strumenti non presenti, lasciando completa libertà allo sviluppatore.

File-System

Affinché un sistema operativo funzioni correttamente, deve presentare una struttura dati che organizza in modo specifico le operazioni di scrittura, ricerca, lettura, memorizzazione, modifica ed eliminazione dei file. La struttura dei file deve assicurare un'identificazione rapida e priva di errori, nonché un accesso il più veloce possibile da parte degli utenti.

I file system, tipicamente, supportano la condivisione di dati tra gli utenti e le applicazioni, nonché persistenza, cosicché i dati siano disponibili anche dopo il reboot.

Un file system affronta diverse sfide:

- Il file system ha bisogno di strutture dati sul disco per rappresentare l'albero dei nomi di file e direttori, per tenere traccia delle identità dei blocchi che contengono al loro interno i contenuti di ogni file e per tenere traccia di quali aree sul disco sono libere.
- Il file system deve supportare il ripristino dagli arresti anomali. Ovvero, se avviene un *crash* (es. interruzione di corrente), il file system deve continuare a lavorare correttamente anche dopo il restart del sistema. Il rischio è che un arresto anomalo può interrompere una sequenza di aggiornamenti e lasciare strutture dati inconsistenti su disco.
- Processi differenti possono operare sul file system allo stesso momento. È compito quindi di quest'ultimo di coordinarsi per mantenere gli invarianti.
- Fare l'accesso su disco è vari ordini di grandezza più lento che accedere alla memoria. Il file system deve quindi mantenere una cache dei blocchi più usati.

I prossimi capitoli indicheranno le caratteristiche dei file system implementati in OS161 e Xv6 e le loro differenze.

1. File-System in OS161

Il File-System nativo in OS161 è chiamato SFS (Simple File System).

SFS è un file system semplice e limitato, ma è sufficiente per le esigenze di un sistema operativo didattico come OS161. I vantaggi di questo formato sono:

- La navigazione degli eventi è possibile utilizzando semplici funzioni di file system indipendenti dal contenuto
- Spese generali molto basse. Nessuna perdita dovuta alla granularità delle dimensioni del blocco
- È possibile creare l'intero file system valido aggiungendo contenuto

D'altra parte, la navigazione nelle directory ad accesso casuale è piuttosto lenta perché non esiste un'indicizzazione incorporata o una gerarchia di directory. Per un file system da 500 MB contenente file da 5 k byte, ciò rappresenta un sovraccarico di ricerca iniziale di ~ 1-2 secondi (~ 100.000 ricerche).

Inoltre, SFS non è resistente agli arresti anomali. Tutte le scritture vengono memorizzate nel buffer e vengono scritte sul disco in un momento e in un ordine arbitrari. Se OS161 dovesse bloccarsi inaspettatamente nel bel mezzo delle operazioni, i dati su disco potrebbero facilmente essere lasciati in uno stato incoerente, rendendo il filesystem inutilizzabile e con quantità arbitrarie di dati persi

Struttura del SFS

La struttura di un file SFS è la seguente:

<table>
<tr><td colspan="2">VolumeSpec</td></tr>
<tr><td colspan="2">Head</td></tr>
<tr><td>File1</td><td>File1 Binary Data</td></tr>
<tr><td>File2</td><td>File2 Binary Data</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td colspan="2">Tail</td></tr>
</table>

- **VolumeSpec:** questa è semplicemente una stringa di caratteri di 12 byte che rappresenta la versione del filesystem. Ad esempio: “SFS V00.01”
- **Head:** questo è un breve record di intestazione. L'ordine dei byte si applica solo al campo *time* di questo record.

<table>
<tr><td>type = “HEAD”</td></tr>
<tr><td>byte_order = 0x04030201</td></tr>
<tr><td>time</td></tr>
</table>

- **File:** i record di file sono record di lunghezza variabile contenenti informazioni su un file.

type = “FILE”
byte_order = 0x04030201
Sz

head_sz	attr	reserved
name....		
name (continued)....		

  - *byteorder:* corrisponde solo a questa intestazione. La fine dei dati non è definita da SFS
  - *sz:* corrisponde alla dimensione del file di dati. Può essere un numero qualsiasi, ma il file stesso verrà riempito per occupare un multiplo di 4 byte
  - *head_sz:* multiplo di 4
  - *attr:*
    - *SFS_ATTR_INVALID:* file eliminato
    - *SFS_ATTR_PUSHDIR:* invia il percorso corrente allo stack dei percorsi
    - *SFS_ATTR_POPDIR:* estrae il percorso corrente dallo stack dei percorsi
    - *SFS_ATTR_NOCD:* questo record non reimposta il basedir
  - *name:* il nome del file
- **Tail:** il record di coda ha lo stesso formato del record “HEAD”, ma di tipo=”TAIL”. Questo record non è necessario, ma può essere presente per indicare che il file è stato chiuso correttamente.

Convenzione sui nomi dei file SFS

Poiché il file SFS contiene semplicemente un elenco di descrittori di file e dati binari, i percorsi possono essere considerati come creazioni ad hoc del codice del lettore SFS. Tuttavia vengono applicate le seguenti convenzioni in modo che sia possibile ricostruire una normale struttura di directory da un file SFS.

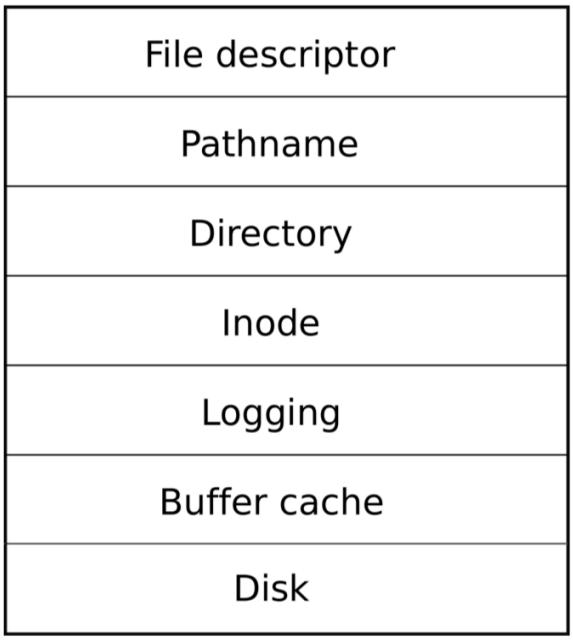
/xxx/xxx/yyy	/xxx/xxx/ is the “directory part” yyy is the “file” part
/xxx	absolute path
xxx or xxx/xxx	path is relative to the directory part of the previous entry
xxx/	Directory
0 length	Zero length files are used to push/pop the current base path from a stack. The files attr field should be set to SFS_ATTR_PUSHDIR or SFS_ATTR_POPDIR

Poiché non sono necessarie voci di directory esplicite, una directory è definita esistente se fa parte del percorso di un file esistente.

È perfettamente legale avere una “directory” che contiene anche dati.

2. **File-System in Xv6**

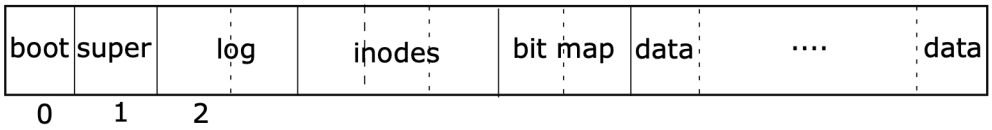
   Il file system in Xv6 è organizzato in sette livelli, indicati nella figura di seguito.



Qui spiegheremo le caratteristiche dei vari livelli:

- **Livello di disco**: legge e scrive i blocchi su un hard drive IDE.
- **Livello di buffer cache**: salva nella cache i blocchi del disco e sincronizza l'accesso ad essi, facendo attenzione che un solo processo kernel alla volta possa modificare i dati salvati nei blocchi.
- **Livello di logging**: consente ai livelli più alti di racchiudere gli aggiornamenti su diversi blocchi in una transazione e garantisce che i blocchi vengano aggiornati atomicamente in caso di arresti anomali (ovvero, tutti vengono aggiornati o nessuno).
- **Livello di inode**: fornisce singoli file, ciascuno rappresentato come un inode con un numero *i* univoco e alcuni blocchi che contengono i dati del file.
- **Livello di directory**: implementa ciascuna directory come un tipo speciale di inode il cui contenuto è una sequenza di voci di directory, ciascuna delle quali contiene il nome di un file e il numero *i*.
- **Livello di pathname**: fornisce nomi di percorso gerarchici come `/usr/rtm/xv6/fs.c` e li risolve con una ricerca ricorsiva.
- **Livello file descriptor**: astrae molte risorse Unix (ad esempio pipe, dispositivi, file, ecc.) utilizzando l'interfaccia del file system, semplificando la vita dei programmatori di applicazioni.

Il file system ha bisogno di un piano per dove salvare gli inode e i blocchi contenuto sul disco. Per fare ciò, Xv6 divide il disco in diverse sezioni, mostrate nella figura sottostante.



Il file system non utilizza il blocco 0 (mantiene i dati del boot). Il blocco 1 è chiamato *superblock*, che contiene i metadati del file system (la dimensione in blocchi, il numero di blocchi, il numero di inode e i numeri di blocchi nel log). I blocchi a partire dal 2 contengono i log. Dopo i log ci sono gli inode, con diversi inode per blocco. Dopo di essi arrivano i blocchi bitmap che tracciano quali blocchi di dati sono in uso. I blocchi rimanenti sono blocchi dati; ciascuno è contrassegnato come libero nel blocco bitmap o contiene il contenuto di un file o di una directory. Il superblocco viene riempito da un programma separato, chiamato `mfks`, che costruisce un file system iniziale.

3. Confronto tra OS161 e Xv6

Di seguito vengono elencate le similitudini e le differenze tra i file system implementati nei diversi sistemi operativi analizzati:

Similitudini:

- Entrambi i file system utilizzano un albero di directory per rappresentare la struttura dei file sul disco.
- Entrambi i file system utilizzano blocchi per memorizzare i dati dei file.
- Entrambi i file system supportano le operazioni di lettura, scrittura, apertura, chiusura, creazione e cancellazione dei file.

Differenze:

- SFS è un file system più semplice e limitato rispetto a Xv6.
    - SFS non utilizza una gerarchia di directory incorporata, il che rende lenta la navigazione nelle directory ad accesso casuale.
    - SFS non è resistente agli arresti anomali, poiché tutte le scritture vengono memorizzate nel buffer e vengono scritte sul disco in un momento e in un ordine arbitrari.
- Xv6 è un file system più complesso e robusto rispetto a SFS.
    - Xv6 utilizza una gerarchia di directory incorporata, che migliora le prestazioni di navigazione nelle directory ad accesso casuale.
    - Xv6 è resistente agli arresti anomali, poiché utilizza un meccanismo di logging per garantire che le modifiche ai blocchi di dati vengano eseguite atomicamente in caso di arresto anomalo.

Conclusioni

In conclusione, l'analisi comparativa tra i sistemi operativi didattici OS161 e Xv6 ha fornito un quadro dettagliato delle loro caratteristiche, prestazioni e strutture interne in termini di system calls, meccanismi di sincronizzazione e di gestione del file system. Entrambi i sistemi operativi offrono un ambiente di apprendimento robusto per comprendere i concetti fondamentali dei sistemi operativi, ma presentano differenze significative nelle loro implementazioni.
OS161, con la sua struttura modulare, fornisce un approccio dettagliato e altamente configurabile per l'apprendimento. Tuttavia, la complessità della sua configurazione e l'intricata struttura dei file al suo interno potrebbero rappresentare una sfida per gli studenti meno esperti. D'altra parte, Xv6 si distingue per la sua semplicità e chiarezza, offrendo un ambiente più accessibile per gli studenti alle prime armi. La sua implementazione elegante e snella rende più agevole l'apprendimento dei concetti chiave dei sistemi operativi, ma potrebbe mancare di profondità in confronto a OS161.

Entrambi i sistemi operativi presentano un set di system call essenziali e forniscono un contesto pratico per esplorare concetti come la sincronizzazione dei processi e la gestione del filesystem. La scelta tra OS161 e Xv6 dipenderà dalle esigenze specifiche dell'insegnamento e dalla familiarità degli studenti con i concetti di sistemi operativi.

In definitiva, entrambi i sistemi operativi offrono un approccio educativo valido, contribuendo a formare una comprensione approfondita dei principi dei sistemi operativi, e la selezione tra i due dipenderà dalle preferenze e dalle esigenze specifiche del corso didattico.

Riferimenti

- Libro ufficiale di Xv6: xv6: a simple, Unix-like teaching operating system
- https://www.eecs.harvard.edu/~cs161/assignments/a4.html
- https://www.star.bnl.gov/public/daq/DAQ1000/sfs.pdf