Analisi comparativa tra Os161 e Xv6

Indice

Introduzione

Questo studio si concentra principalmente sull'analisi del sistema operativo Xv6, confrontandolo con Os161. L'obiettivo di questa analisi comparativa è offrire una panoramica delle capacità di Os161 e Xv6, mettendo in evidenza le differenze e le somiglianze tra questi due sistemi operativi open source.

Xv6

Così come Os161, Xv6 è un sistema operativo nato con scopi didattici, sviluppato presso il MIT, che si propone come un sistema operativo semplice e leggero, ma allo stesso tempo completo e funzionale. É basato su UNIX

V6, da cui prende il nome, realizzato in C e assembly. La prima versione del 2006 è progettata per essere eseguibile su architetture x86, mentre nella versione del 2020 è stata realizzata una conversione per RISC-V. La versione che abbiamo scelto di analizzare è quella del 2006 per x86.

Un fatto fondamentale di xv6 è che contiene tutti i concetti fondamentali di Unix e ha una struttura simile a Unix anche se manca di alcune funzionalità che ti aspetteresti da un sistema operativo moderno. Questo è un sistema operativo leggero in cui il tempo di compilazione è molto basso e consente anche il debug remoto.

Le funzioni chiave oggetto di studio includono system calls, meccanismi di sincronizzazione, gestione della memoria virtuale e della MMU, algoritmi di scheduling, oltre a esaminare altre caratteristiche rilevanti per il corretto funzionamento di un sistema operativo.

Struttura del Kernel

Il sistema operativo ha una struttura **monolitica** (come la maggior parte dei sistemi operativi Unix): il che significa che tutte le funzionalità essenziali del sistema operativo sono implementate nel kernel stesso.

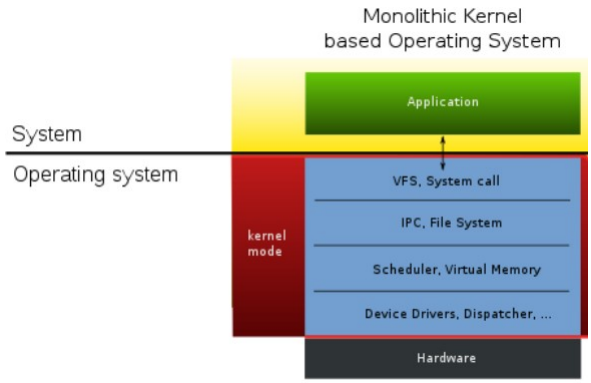

Figura 1: Kernel monolitico

In questa organizzazione, l'intero sistema operativo viene eseguito con pieno privilegio hardware. Questa organizzazione è conveniente perché il progettista del sistema operativo non deve decidere quale parte del sistema operativo non ha bisogno di pieno privilegio hardware. Inoltre, è facile per diverse parti del sistema operativo cooperare. Ad esempio, un sistema operativo potrebbe avere una cache di buffer che può essere condivisa sia dal sistema di file che dal sistema di memoria virtuale.

Un inconveniente dell'organizzazione monolitica è che le interfacce tra le diverse parti del sistema operativo sono spesso complesse, ed è quindi facile per uno sviluppatore del sistema operativo commettere un errore. In un kernel monolitico, un errore è fatale, perché spesso comporta il fallimento del kernel stesso. Se il kernel fallisce, il computer smette di funzionare e, di conseguenza, tutte le applicazioni falliscono. Il computer deve essere riavviato per ripartire.

Per ridurre il rischio di errori nel kernel, i progettisti del sistema operativo possono minimizzare la quantità di codice del sistema operativo che viene eseguita in modalità kernel ed eseguire la maggior parte del sistema operativo in modalità utente. Questa organizzazione del kernel è chiamata **microkernel**.

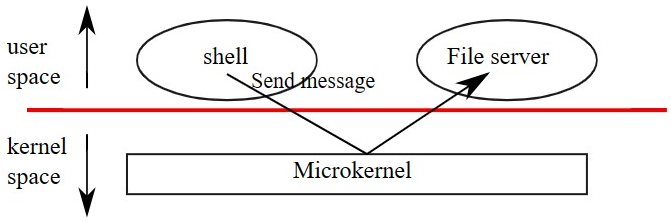

Figura 2: Microkernel

Nella Figura 2, il sistema di file viene eseguito come processo a livello utente. I servizi del sistema operativo eseguiti come processi sono chiamati server. Per consentire alle applicazioni di interagire con il file server, il kernel fornisce un meccanismo di comunicazione tra processi per inviare messaggi da un processo a livello utente a un altro.

In un microkernel, l'interfaccia del kernel consiste in alcune funzioni a basso livello per avviare applicazioni, inviare messaggi, accedere all'hardware del dispositivo, ecc. Questa organizzazione consente al kernel di essere relativamente semplice, poiché la maggior parte del sistema operativo risiede nei server a livello utente.

Poiché Xv6 non fornisce molti servizi, il suo kernel è più piccolo di alcuni microkernel.

Processi

I processi vanno, in xv6, a costituire l'unità fondamentale di *isolamento*. L'astrazione del processo fornisce l'illusione a un programma che esso abbia la sua stessa macchina privata. Un processo fornisce a un programma ciò che sembra essere un **sistema di memoria privato**, o spazio degli indirizzi, che altri processi non possono leggere o scrivere. Un processo fornisce anche al programma ciò che sembra essere la sua stessa CPU per eseguire le istruzioni del programma.

Xv6 utilizza le *page tables* (implementate dall'hardware) per dare a ogni processo il suo spazio degli indirizzi. La tabella delle pagine x86 mappa un indirizzo virtuale in un indirizzo fisico. Ogni spazio degli indirizzi di un processo mappa le istruzioni e i dati del kernel, così come la memoria del programma utente. Il kernel di Xv6 mantiene molte informazioni di stato per ciascun processo, che vengono raccolte in una struttura chiamata struct proc. Gli elementi più importanti dello stato del kernel di un processo sono la sua tabella delle pagine, il suo stack del kernel e il suo stato di esecuzione.

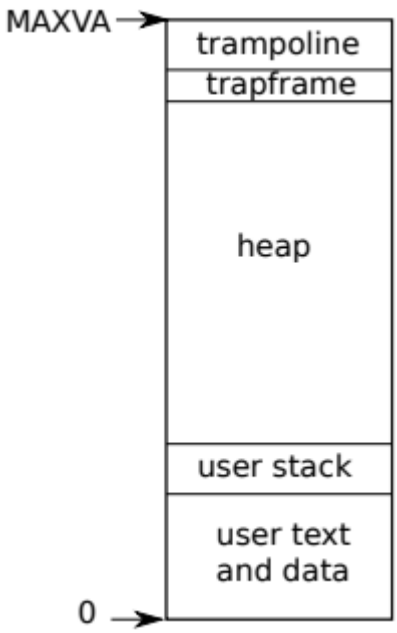

Figura 3: Layout dello spazio degli indirizzi virtuali di un processo.

Un processo è definito come: *struct proc*.

- p->state: se il processo è allocato, pronto per l'esecuzione, in esecuzione, in attesa di I/O o in uscita:

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

- p->pagetable: contiene la page table del processo.
- Altri campi sono privati per il processo e mantengono lo stato privato per ciascun processo (ad esempio registri, program counter..) o sono utilizzati dallo sheduler del processo

Quando un processo effettua una chiamata di sistema, il processore passa allo stack del kernel, aumenta il livello di privilegio hardware e inizia a eseguire le istruzioni del kernel che implementano la chiamata di sistema. Quando la chiamata di sistema si completa, il kernel torna allo spazio utente: l'hardware abbassa il livello di privilegio, passa di nuovo allo stack utente e riprende l'esecuzione delle istruzioni utente subito dopo l'istruzione di chiamata di sistema.

Page Tables

La *page table* è una struttura dati utilizzata in xv6 (come in tutti i sistemi operativi, OS161 incluso) per gestire la mappatura tra gli indirizzi virtuali e quelli fisici della memoria. La sua funzione principale è consentire a ciascun processo di operare nel proprio spazio degli indirizzi virtuale, isolato dagli altri processi. Inoltre consente di mappare gli indirizzi virtuali di un processo agli indirizzi fisici corrispondenti della memoria. Ogni processo, come visto nel paragrafo precente, ha la sua *page table*

Le istruzioni x86 (sia utente che kernel) manipolano indirizzi virtuali. La RAM della macchina, o memoria fisica, è indicizzata con indirizzi fisici.

In xv6 la memoria virtuale è realizzata tramite **paging hardware** e, a differenza di Os161, non dispone di 'tecniche complesse' come: shared memory, demand paging e COW. La pagina virtuale è divisa in due livelli:

- **Livello 0 (PGTBL0)**: contiene una tabella di 512 voci che puntano a pagine virtuali di livello 1

- **Livello 1 (PGDIR)**: contiene una tabella di 512 voci che puntano a pagine fisiche. Ogni voce ha due campi dati: uno per l'indirizzo fisico della pagina e uno per i bit di accesso alla memoria.

In xv6 le pagine vengono mappate con la seguente regola: l'indirizzo virtuale ha dimensione 32 bit ed è calcolato: `VA = PGSIZE * PTE_ADDR + offset`. Dove `PTE_ADDR` è l'indice nella tabella di livello 1 e `offset` è l'offset all'interno della pagina. Una pagina virtuale può essere mappata in più pagine fisiche, cioè una stessa pagina virtuale può avere diverse pagine fisiche associate ad essa. Questo avviene quando si esegue un programma compilato con l'opzione `-pg`, ovvero con le funzioni di profilatura attiva. In questo caso il kernel crea una copia "shadow" delle pagine del processo e le mappa in modo diverso. Le pagine shadow non sono visibili al processo, ma possono essere utilizzate dal debugger per analizzare il codice sorgente.

Modifiche alla paginazione Nel file `kernel/vm.c` troviamo le funzionalità relative alla modifica della paginazione. Ecco le principali:

- `alloc_page()`: Alloca una pagina fisica libera.
- `mapmem()`: Mappa una pagina virtuale in una pagina fisica.
- `clearpte()`: Resetta i bit di accesso di una pagina.
- `copyuvm()`: Copia le pagine virtuali di un altro processo.
- `switchuvm()`: Cambia il contesto del processo corrente.
- `loaduvm()`: Carica il contenuto di un file in una pagina virtuale.

Il *paging* della versione x86 di xv6 collega questi due tipi di indirizzi, mappando ogni indirizzo virtuale in un indirizzo fisico.

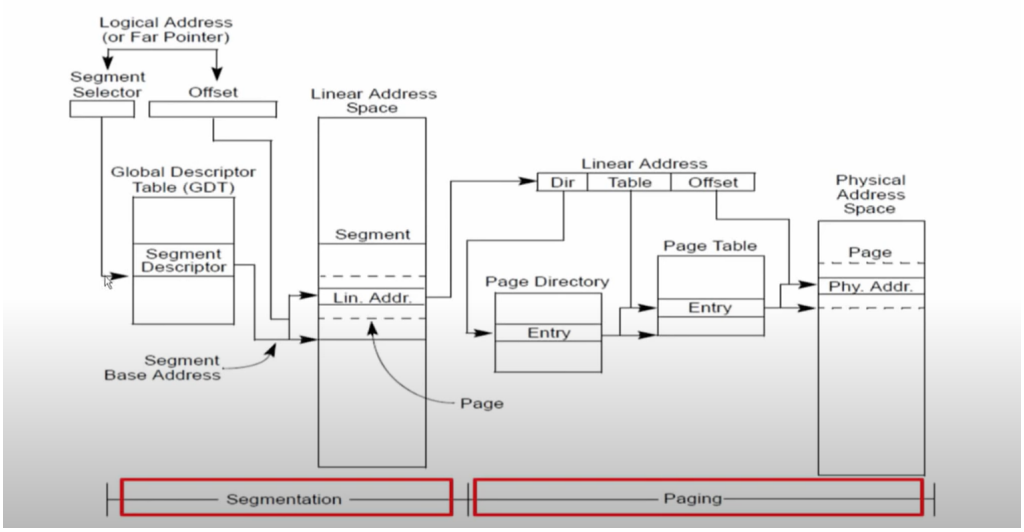

Figura 4: Schema del paging

Il *paging* è diviso in due fasi:

1. **SEGMENTAZIONE** in cui si ottiene 'l'indirizzo lineare' partendo dall'indirizzo virtuale.
2. **Page Table gerarchica** per la conversione da indirizzo lineare a fisico: Una page table x86 è logicamente un array di 2^20 (1.048.576) page table entries (PTEs). Ogni PTE è diviso in: 20 bit per il *physical page number* (PPN) e *flags*. Durante la traduzione di un virtual address utilizza i suoi primi 20 bit per indicizzare la page table e trovare una PTE, sostituendo i primi 20 bit dell'indirizzo con il PPN nella PTE.

Ciascun PTE contiene bit di flag che indicano all'hardware di paging come è consentito utilizzare l'indirizzo virtuale associato:

- PTE_P indica se la PTE è presente: se non è impostata, un riferimento alla pagina provoca un errore (cioè non è consentito).
- Controlli PTE_W se le istruzioni possono eseguire scritture sulla pagina; se non impostato, sono consentite solo letture e recuperi di istruzioni.
- PTE_U controlla se i programmi utente possono utilizzare la pagina; se deselezionato, solo al kernel è consentito utilizzare la pagina.

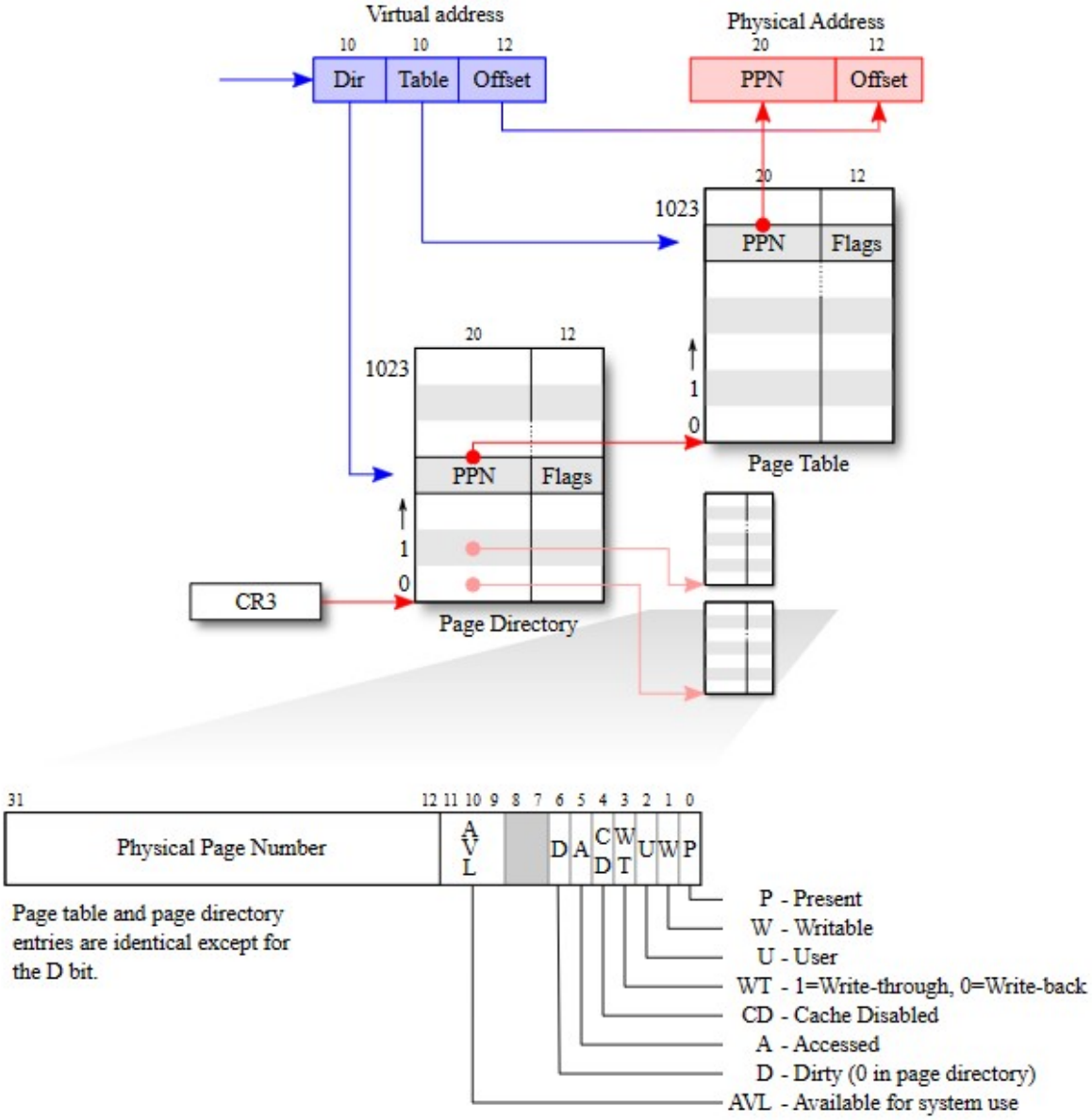

Figura 5: Page table di x86

La traduzione effettiva degl indirizzi avviene in fasi:

1. Una *page table* è memorizzata in memoria fisica come un albero a due livelli. La radice dell'albero è **CR3** che contiene l'indirizzo fisico del sistema di paginazione: il primo elemento della Page Directory formata da 4096 byte e che contiene 1024 PTE per *page table*. Ogni pagina della *page table*, quindi, è un array di 1024 PTE da 32 bit.
2. L'hardware, che gestisce le eccezioni nel caso di mancate corrispondenze nelle tabelle di paging, utilizza i primi 10 bit di un indirizzo virtuale per selezionare un ingresso del page directory. Se, sia l'ingresso del page directory che la PTE non sono presenti, l'hardware di paginazione genera una

eccezione. Questo approccio garantisce un controllo preciso sull'accesso alla memoria virtuale e impedisce l'accesso non autorizzato a regioni di memoria.
3. I primi 20 bit della Page Directory danno l'informazione su quale Page Table usare per la traduzione virtuale - fisico, e i secondi 10 bit dell'indirizzo virtuale specificano la posizione esatta in PT. I 12 bit di offset completano la traduzione.

System Call, Exceptions e Interrupts

Esistono tre casi in cui il controllo deve passare da un programma utente al kernel per accedere a servizi/privilegi limitati a quest'ultimo:

1. Una **System Call**: un modo per i programmi utente di interagire con il sistema operativo. Un programma chiama una *system call* quando effettua una richiesta al kernel del sistema operativo. Vengono utilizzate per servizi hardware, per creare o eseguire processi e per comunicare con i servizi del kernel, inclusa la pianificazione di applicazioni e processi.
2. Un **Exception**: quando un programma compie un'azione illegale. Esempi di azioni illegali includono divisione per zero, tentativo di accedere alla memoria per una voce di tabella delle pagine che non è presente, e così via.
3. Un **Interrupt**: quando un dispositivo genera un segnale per indicare che necessita dell'attenzione del sistema operativo.

In tutti e tre i casi, il design del sistema operativo deve:

1. Salvare i registri del processore per un futuro ripristino trasparente.
2. Essere configurato per l'esecuzione nel kernel scegliendo un punto in cui far iniziare l'esecuzione del kernel.
3. Il kernel deve essere in grado di recuperare informazioni sull'evento, ad esempio gli argomenti della chiamata di sistema.

Tutto deve essere fatto in modo sicuro; il sistema deve mantenere l'isolamento tra i processi utente e il kernel.Per raggiungere questo obiettivo, il sistema operativo deve essere consapevole dei dettagli su come l'hardware gestisce le chiamate di sistema, le eccezioni e le interruzioni. Nella maggior parte dei processori, questi tre eventi sono gestiti da un unico meccanismo hardware.

Ad esempio, sull'x86, un programma invoca una chiamata di sistema generando un'interruzione mediante l'istruzione `int`. Allo stesso modo, le eccezioni generano anch'esse un'interruzione. Pertanto, se il sistema operativo ha un piano per la gestione delle interruzioni, può gestire anche chiamate di sistema ed eccezioni.

Una nota sulla terminologia: anche se il termine ufficiale x86 è eccezione, Xv6 utilizza il termine trap, principalmente perché era il termine utilizzato dal PDP11/40 ed è quindi il termine Unix convenzionale. É importante ricordare che le trap sono causate dal processo corrente in esecuzione su un processore (ad esempio, il processo effettua una chiamata di sistema e genera di conseguenza una trap), mentre le interruzioni sono causate dai dispositivi e potrebbero non essere correlate al processo in esecuzione al momento dell'interruzione.

Il file `user/usertrap.S` contiene la traccia dei segnali di fault. Ciò permette di rispondere ai segnali di fault generati dai processi. Il kernel chiama la funzione `trap_handler()` passando come parametro lo stato del processore. La funzione *trap_handler()* determina se il segnale di fault è generato da un errore o da un segnale di interrupt. Nel primo caso chiama la funzione `fault()`, mentre nel secondo caso chiama la funzione

`intr()`. In entrambi i casi la funzione invoca la funzione corrispondente a quella generata dal segnale di fault o dall'interrupt.

Gestori di trap in Assembly

L'x86 consente 256 diverse interruzioni. Le interruzioni da 0 a 31 sono definite per eccezioni software, come errori di divisione o tentativi di accesso a indirizzi di memoria non validi. Xv6 mappa le 32 interruzioni hardware nell'intervallo da 32 a 63 e utilizza l'interruzione 64 come interruzione di chiamata di sistema.

`Tvinit`, chiamata da main, imposta le 256 voci nella tabella IDT (Interrupt Descriptor Table). L'interruzione i è gestita dal codice all'indirizzo in vectors[i]. Ogni punto di ingresso è diverso, poiché l'x86 non fornisce il numero di trap all'handler di interruzione. Utilizzare 256 handler diversi è l'unico modo per distinguere i 256 casi.

Tvinit gestisce in modo particolare T_SYSCALL, la trap di chiamata di sistema dell'utente: specifica che il gate è di tipo 'trap' passando un valore di 1 come secondo argomento. I gate di trap non cancellano il flag IF, consentendo altre interruzioni durante l'handler della chiamata di sistema.

Leggenda registri

Nome Registro	Descrizione
%cs	Imposta il livello di protezione (da 0 a 3)
%eip	Indirizzo dell'istruzione subito dopo l'istruzione int
%esp	Punta al frame trap appena costruito
%ss	Selettore

Per effettuare una chiamata di sistema sull'x86, un programma richiama l'istruzione int n, dove specifica l'indice nell'IDT. L'istruzione int esegue i seguenti passaggi:

- Recupera l'ennesimo descrittore dall'IDT, dove n è l'argomento di int.
- Verificare che CPL in %cs sia <= DPL, dove DPL è il livello di privilegio nel descrittore.
- Salva %esp e %ss in registri interni alla CPU, ma solo se PL < CPL del selettore del segmento target.
- Carica %ss e %esp da un descrittore di segmento di attività.
- Push in ordine dei registri: %ss, %esp, %flag, %cs, %eip .
- Cancella alcuni bit di %flags.
- Imposta %cs e %eip sui valori nel descrittore.

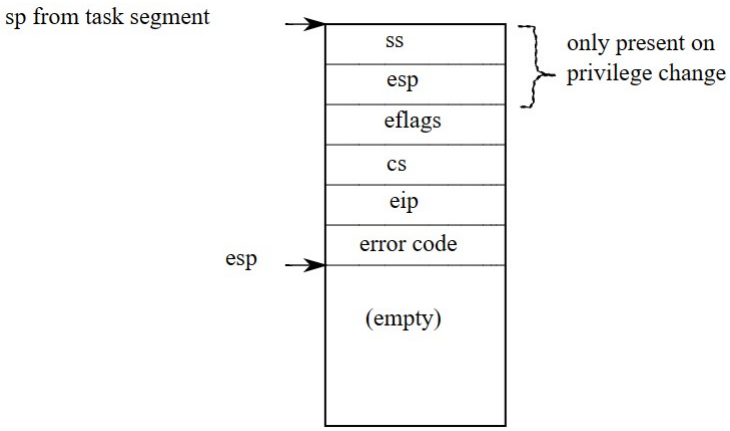

Figura 6: Stack del Kernel dopo una chiamata *int*

La Figura 6 mostra lo stack dopo il completamento di un'istruzione int e la modifica del livello di privilegio (il livello di privilegio nel descrittore è inferiore a CPL). Se l'istruzione int non richiedeva una modifica del livello di privilegio, x86 non salverà %ss e %esp. Dopo entrambi i casi, %eip punta all'indirizzo specificato nella tabella descrittore e l'istruzione a quell'indirizzo è la successiva istruzione da eseguire e la prima istruzione del gestore per l'int n. È compito del sistema operativo implementare questi gestori e di seguito vedremo cosa fa xv6.

Gestori di trap in C

Trap osserva il numero di trap dell'hardware `tf->trapno` per decidere il motivo per cui è stato chiamato e cosa deve essere fatto. Se la trap è T_SYSCALL, trap chiama il gestore di chiamate di sistema syscall.

Dopo aver verificato la chiamata di sistema, come secondo caso considerato trap cerca interruzioni hardware.

In ultimo trap assume che l'interruzione sia stata causata da un comportamento incorretto. In questo caso, se è stata causata da un programma utente Xv6 stampa i dettagli e imposta `proc->killed` per terminare il processo. Se la trap è stata causata dal kernel, Xv6 stampa un messaggio di errore e chiama panic per terminare il kernel.

Codice: System Call

Per le chiamate di sistema, trap invoca **syscall**. Syscall carica il numero di chiamata di sistema dal frame di trap, che contiene il valore salvato in %eax, e lo indicizza **system call table**. Per la prima chiamata di sistema, %eax contiene il valore *SYS_exec*, e syscall invocherà l'entrata SYS_exec-esima della tabella delle chiamate di sistema, corrispondente all'invocazione di *sys_exec*.

Quando exec ritorna, restituirà il valore restituito dal gestore di chiamate di sistema. Le system call restituiscono convenzionalmente numeri negativi per indicare errori, numeri positivi per indicare successo.

L'aggiunta di una nuova **system call** a Xv6 richiede la modifica del codice del kernel e l'aggiunta di una nuova voce alla `system call table`:

1. Definire la nuova *system call* creando una nuova funzione nel codice del kernel. La funzione deve accettare gli argomenti necessari e restituire il valore richiesto. La funzione deve anche verificare la presenza di eventuali errori e restituire un codice di errore appropriato.

2. Aggiungere il numero della *system call*: Scegliere un nuovo `numero di system call unico` e aggiungerlo alla system call table nel codice del kernel. Questa tabella è tipicamente definita in `syscall.h`.

```
#define SYS_getyear 22
```

3. Aggiungere un puntatore alla chiamata di sistema nel file `syscall.c.` Questo file contiene un array di puntatori a funzioni che utilizza i numeri (indici) sopra definiti come puntatori alle *system calls* definite in posizioni diverse. Per aggiungere una *system call* personalizzata:

```
[SYS_getyear] sys_getyear
```

Ciò significa che quando si verifica una chiamata di sistema con il numero 22, viene chiamata la funzione indicata dal puntatore di funzione sys_getyear. È quindi necessario implementare questa funzione. Si metterà solo il prototipo della funzione all'interno di questo file:

```
extern int sys_getyear(void)
```

4. Aggiungere l'interfaccia a livello utente: Aggiungete un'interfaccia a livello utente alla nuova *system call*, aggiungendo una nuova voce alla libreria delle chiamate di sistema nel file `usys.S`. Questa voce deve chiamare la chiamata di sistema con gli argomenti appropriati e restituire il risultato.

```
SYSCALL(getyear)
```

5. Implementare la **system call function** in `sysproc.c`:

```
//return the year of which
//Unix version 6 was released
int  sys_getyear(void)
{
    return 1975;
}
```

6. Per testare il corretto funzionamento della *system call* appena implementata bisogna modificare il programma a livello utente includendo il file di intestazione che definisce la nuova chiamata di sistema e chiamando la funzione della chiamata di sistema con gli argomenti appropriati.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void)
{
printf(1, "Note: Unix V6 was released in year %d\n", getyear());
    exit();
}
```

7. Modificare il `Makefile` aggiungendo la nuova *system call* alla lista di quelle già esistenti. Build e test.

```
make clean
make
```

Sincronizzazione

Xv6 è progettato per funzionare su multiprocessori: computer con più CPU che eseguono operazioni in modo indipendente. Queste CPU condividono un singolo spazio di indirizzi fisici e strutture dati; Xv6 sfrutta questa condivisione per mantenere strutture dati a cui tutte le CPU accedono in lettura e scrittura. Per evitare che le CPU si sovrappongano e corrompano le strutture dati, Xv6 utilizza meccanismi di sincronizzazione per coordinare l'accesso concorrente alle strutture dati condivise. Questi meccanismi di sincronizzazione includono `semafori` e `lock`.

Un `lock` fornisce la **mutua esclusione**, garantendo che solo una CPU alla volta possa detenere la zona di memoria di interesse. Se il *lock* è associato a ciascun elemento dati condiviso e il codice tiene sempre il *lock* associato quando utilizza un determinato elemento, possiamo essere certi che l'elemento viene utilizzato solo da una CPU per volta. Xv6 molto spesso utilizza i lock per evitare **race conditions**.

Xv6 sfrutta due tipologie di lock: `spin-lock` e `sleep-lock`.

Spin Lock

Xv6 rappresenta uno **spin-lock** come una struttura chiamata `struct spinlock`. Il campo importante nella struttura è **lk->locked**, un dato booleano che è 0 quando la lock è *disponibile*, 1 quando è *detenuta*. Logicamente, Xv6 dovrebbe acquisire una lock eseguendo il codice come segue:

`Spinlock.h` e `Spinlock.c`

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};

void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
```

```
{
  // Disable interrupts to avoid deadlock. Really
  // only necessary to protect locks that may be
  // acquired by an ISR (to avoid the deadlock
  // that would occur if you were interrupted in
  // a crit-sect by an ISR that wanted your held
  // lock and couldn't release the lock
  // until the ISR returned.) XV6 takes a
  // conservative approach and disables interrupts
  // when acquiring any spinlock.
  pushcli();
  if(holding(lk))
    panic("acquire");

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

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() introduces a guard rail for that.
  // This ensures that lock acquire/release instructions always
  // correctly book-end critical-section instructions.
  __sync_synchronize();

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

// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  __sync_synchronize();
  ...

  popcli();  // Re-enable interrupts
}
```

Sfortunatamente, questa implementazione non garantisce la mutua esclusione su un multiprocessore. Potrebbe accadere che due CPU raggiungano simultaneamente a controllare la condizione dell'if, vedano che *lk->locked = 0*, e quindi entrambe acquisiscano il lock eseguendo la riga successiva. A questo punto, due

CPU diverse detengono il *lock* violando la proprietà di esclusione reciproca. Affinchè il codice sopra sia corretto, le due righe devono essere eseguite in un passo atomico.

Per eseguire queste due righe atomicamente, Xv6 si affida a un'istruzione x86 speciale, `xchg`. In un'operazione atomica, *xchg* scambia una parola in memoria con il contenuto di un registro. La funzione `acquire` ripete questa istruzione *xchg* in un ciclo; ogni iterazione legge atomicamente *lk->locked* e lo imposta a 1. Se il lock è già detenuta, *lk->locked* sarà già 1, quindi *xchg* restituirà 1 e il ciclo continuerà. Tuttavia, se *xchg* restituisce 0, *acquire* ha acquisito con successo la lock: locked era 0 ed è ora 1, quindi il ciclo può fermarsi. Una volta acquisita la lock, *acquire* registra, per scopi di debug, la CPU e la traccia dello stack che hanno acquisito il lock. Se un processo dimentica di rilasciare il lock, queste informazioni possono aiutare a identificare il responsabile. *Questi campi di debug sono protetti dal lock e devono essere modificati solo mentre si detiene la lock*.

La funzione `release` è l'opposto di *acquire*: cancella i campi di debug e quindi **rilascia il lock**. La funzione utilizza un'istruzione di assembly per cancellare locked, perché la cancellazione di questo campo dovrebbe essere atomica. Xv6 non può utilizzare una normale assegnazione in C, perché la specifica del linguaggio C non specifica che un'unica assegnazione è atomica. L'implementazione delle spin-lock in Xv6 è specifica per x86 e quindi Xv6 non è direttamente portabile su altri processori.

Lock in Xv6

Lock	Descrizione
bcache.lock	Protegge l'allocazione delle voci della cache del buffer del blocco
cons.lock	Serializza l'accesso all'hardware della console, evita l'output intersperso
ftable.lock	Serializza l'allocazione di una struct file nella tabella dei file
icache.lock	Protegge l'allocazione delle voci della cache dell'inode
idelock	Serializza l'accesso all'hardware del disco e alla coda del disco
kmem.lock	Serializza l'allocazione di memoria
log.lock	Serializza le operazioni sul log delle transazioni
pipe's p->lock	Serializza le operazioni su ogni pipe
ptable.lock	Serializza il cambio di contesto e le operazioni su proc->state e proctable
tickslock	Serializza le operazioni sul contatore dei ticks
inode's ip->lock	Serializza le operazioni su ogni inode e sul suo contenuto
buf's b->lock	Serializza le operazioni su ogni buffer del blocco

Esempio per l'Interrupt Handler:

Nel momento in cui uno spin-lock viene utilizzato da un gestore di interruzioni, un processore non deve mai detenere quel lock con le interruzioni abilitate. Xv6 ha un atteggiamento conservativo, infatti quando un processore entra in una sezione critica di spin-lock, Xv6 si assicura sempre che le interruzioni siano disabilitate su quel processore. Le interruzioni possono ancora verificarsi su altri processori, quindi l'acquisizione di un'interruzione può attendere che un thread rilasci uno spin-lock; solo non sullo stesso processore. Xv6

riabilita le interruzioni quando un processore non detiene spin-lock; deve fare un po' di registrazione per gestire le sezioni critiche nidificate.

Sleep Lock

Un **sleep-lock** è un tipo di lock che può essere rilasciato e acquisito da un thread diverso da quello che lo detiene. Un thread che tenta di acquisire un *sleep-lock* che è già detenuto da un altro thread viene messo in attesa. Quando il thread che detiene il *sleep-lock* lo rilascia, il thread in attesa lo acquisisce e riprende l'esecuzione.

A differenza dei *spin locks*, che continuano a girare in un loop, *sleep-locks* permettono a un thread di andare in stato di *sleep* (attesa) e di essere risvegliato quando il *lock* diventa disponibile, riducendo l'utilizzo della CPU e migliorando l'efficienza Nel contesto di xv6, un *sleep lock* ha un campo bloccato protetto da uno spin lock. Quando un thread vuole acquisire il lock, chiama la funzione `acquiresleep`, che, in modo atomico, rilascia la CPU e rilascia lo spin lock, permettendo al thread di andare in attesa. Quando il lock è disponibile, il thread viene risvegliato.

In `proc.c` è gestito quasi interamente le funzioni legate ai processi e ai thread, incluse le loro sincronizzazioni tramite lock. In `sleeplock.h` e `sleeplock.c` invece abbiamo l'implementazione dello *sleep lock* e delle sue funzioni:

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};

void
initsleeplock(struct sleeplock *lk, char *name)
{
  initlock(&lk->lk, “sleep lock”);
  lk->name = name;
  lk->locked = 0;
  lk->pid = 0;
}

void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

Gli *sleep-lock* di Xv6 supportano il rilascio del processore durante le loro sezioni critiche. Per evitare casi di deadlock, la routine di acquisizione del blocco di attesa (chiamata *acquiresleep*) rilascia il processore in modo atomico durante l'attesa e non disabilita gli interrupt.

A un livello elevato, uno *sleep-lock* ha un campo bloccato che è protetto da un spinlock, e la chiamata a sleep di *acquiresleep* cede atomicamente la CPU e rilascia lo spin-lock. Il risultato è che altri thread possono eseguire mentre *acquiresleep* aspetta. Poiché gli sleep-lock lasciano gli interrupt abilitati, non possono essere utilizzati negli interrupt. Inoltre, dato che *acquiresleep* può cedere il processore, gli *sleep-lock* non possono essere utilizzati all'interno di sezioni critiche di spin-lock

Xv6 utilizza spin-lock nella maggior parte delle situazioni, poiché hanno un basso overhead. Utilizza gli sleep-lock solo nel sistema di file, dove è conveniente poter detenere i blocchi attraverso lunghe operazioni disco.

Gli spin-lock sono più adatti per brevi sezioni critiche, poiché attendere su di essi spreca tempo CPU; gli sleep-lock funzionano bene per operazioni prolungate.

Scheduling

Ogni sistema operativo deve adottare un algoritmo di scheduling per decidere quale processo eseguire. Questo perché, anche in condizioni di muliprocessore, il numero di processi da eseguire sono maggiori del numero di processori disponibili. Xv6 utilizza un semplice algoritmo di scheduling a round-robin, che assegna a ciascun processo un intervallo di tempo fisso, chiamato quantum, durante il quale il processo può eseguire. Quando il quantum di un processo scade, Xv6 interrompe il processo e sceglie un altro processo da eseguire. É un algoritmo a priorità fissa in quanto tutti i processi hanno la stessa importanza nell'essere eseguiti.

Multiplexing

Xv6 attua il *multiplexing* passando ciascun processore da un processo a un altro in due situazioni. In primo luogo, il meccanismo di *sleep* e *wakeup* di Xv6 cambia quando un processo attende il completamento di I/O su dispositivi o pipe, o attende l'uscita di un processo figlio, o attende nella chiamata di sistema sleep. In secondo luogo, Xv6 forza periodicamente un cambio quando un processo sta eseguendo istruzioni utente. Questo multiplexing crea l'illusione che ciascun processo abbia il proprio processore, proprio come Xv6 utilizza l'allocazione di memoria e le tabelle di pagina hardware per creare l'illusione che ciascun processo abbia la propria memoria.

Context switching

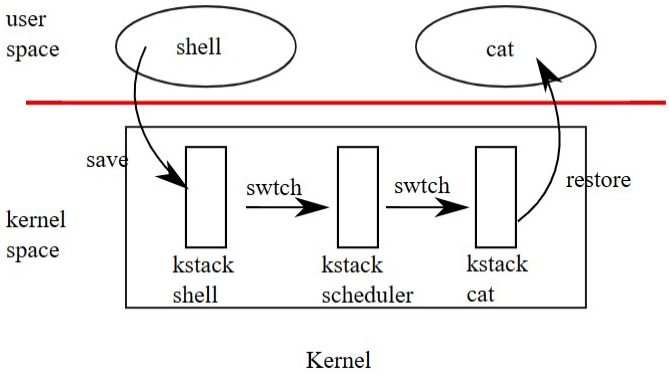

Figura 7: Switch da un processo utente ad un altro.

La Figura 7 illustra i passaggi coinvolti nel passaggio da un processo utente a un altro; xv6 esegue due tipi di *context-switching* a basso livello: dal thread del kernel di un processo al thread dello scheduler della CPU corrente e dal thread dello scheduler al thread del kernel di un processo. xv6 non passa mai direttamente da un processo dello spazio utente a un altro; ciò avviene tramite una transizione utente-kernel (*system call* o *interrupt*), un cambio di contesto allo scheduler, un cambio di contesto al thread del kernel di un nuovo processo e un ritorno di trap.

Lo scheduler di xv6 ha il suo thread (registri e stack salvati) perché talvolta non è sicuro che possa eseguire su uno stack kernel di qualsiasi processo.(Come visto nel paragrafo precedente) Il passaggio da un thread a un altro comporta il salvataggio dei registri CPU del vecchio thread e il ripristino dei registri precedentemente salvati del nuovo thread; il fatto che %esp e %eip siano salvati e ripristinati significa che la CPU cambierà gli stack e cambierà il codice che sta eseguendo.

In `swtch.S` troviamo il codice assembly per il context switching a livello kernel:

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)  # Write stack pointer to *old
  movl %edx, %esp  # Switch %esp to `new`

  # Load new callee-save registers.
  # Format is exact same as saved above,
  # as these were saved by a previous
  # call to `swtch`. %eip was saved
  # implicitly as return address by the
  # `call` instruction that invoked that
  # previous `swtch` execution--`ret`
  # jumps to it. So this completely
```

```
  # encapsulates switching b/w threads
  # of execution. Combine with `switchuvm`
  # to switch b/w procs.

  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Per semplicità possiamo vedere quanto scritto in assembly anche scritto in C per un eventuale studio del comportamento:

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};

void swtch(struct context **old, struct context *new);
```

La funzione *swtch* esegue il salvataggio e il ripristino per un cambio di thread. Ogni contesto è rappresentato da un struct context*, un puntatore a una struttura memorizzata nello stack kernel coinvolto. Swtch prende due argomenti: struct context **old e struct context *new. Inserisce i registri correnti nello stack e salva il puntatore dello stack in *old. Quindi swtch copia new in %esp, estrae i registri precedentemente salvati e restituisce.

Scheduler

Un processo che desidera cedere la CPU deve acquisire il lock della tabella dei processi **ptable.lock**, rilasciare eventuali altri lock che sta tenendo, aggiornare il proprio stato (*proc->state*) e quindi chiamare *sched*. Quest'ultimo verifica le condizioni necessarie, tra cui l'assenza di altri lock e la disabilitazione degli interrupt, e quindi chiama *swtch* per passare al contesto dello scheduler. Lì, avviene la selezione di un nuovo processo eseguibile, seguito da un nuovo passaggio di contesto attraverso swtch, che porta l'esecuzione al nuovo processo.

Una caratteristica unica è l'uso di ptable.lock attraverso le chiamate a swtch, anche se questo va contro la convenzione di rilasciare il lock nel thread che l'ha acquisito. Tuttavia, questo approccio è necessario per proteggere invarianti critiche sui campi di stato e contesto del processo durante swtch.

Il codice della pianificazione acquisisce e rilascia ptable.lock per mantenere la coerenza degli stati dei processi. Inoltre, la sua struttura è progettata per far rispettare invarianti cruciali su ciascun processo. L'acquisizione e il rilascio del lock avvengono in thread diversi per garantire che gli invarianti siano rispettati durante le transizioni di stato.

L'ultima sezione ha esaminato i dettagli di basso livello di swtch; ora prendiamo swtch come dato ed esaminiamo le convenzioni coinvolte nel passaggio dal processo allo scheduler e di nuovo al processo.

Ogni sistema operativo deve adottare un **algoritmo di scheduling** per decidere quale processo eseguire. Questo perché, anche in condizioni di muliprocessore, il numero di processi da eseguire sono maggiori del numero di processori disponibili. Xv6 utilizza un semplice algoritmo di scheduling denominato `round-robin` basato sul *time-sharing*. Le sue caratteristiche principali includono:

- *Rotazione dei Processi*: I processi vengono eseguiti in cicli circolari, assegnando a ciascun processo un intervallo di tempo noto come **quantum** o **time slice**.
- *FIFO*: I processi vengono accodati in una coda e vengono eseguiti in ordine di arrivo. Quando un processo termina il suo quantum, viene messo in fondo alla coda, consentendo agli altri processi di essere eseguiti. É un algoritmo a *priorità fissa* in quanto tutti i processi hanno la stessa importanza nell'essere eseguiti.
- Nessun processo rimane in esecuzione indefinitamente.
- *Implementazione Semplice*.
- *Overhead di Cambio di Contesto*: Può generare un overhead di cambio di contesto, specialmente con processi che richiedono frequenti switch.

Una volta che il kernel ha finito di configurarsi (*swtch.S*), inizializzare tutti i dispositivi e i driver, ecc, in `proc.c` l'ultima funzione che `main()` chiama è `scheduler()`. Gli interrupt sono stati disabilitati nel boot loader e non sono stati ancora abilitati, quindi è anche compito dello scheduler abilitarli per la prima volta in xv6.

scheduler() non ritorna mai; è un ciclo infinito che continua a cercare un processo nella tabella dei processi con stato **RUNNABLE**, quindi lo esegue. Da quel momento in poi, ad eccezione degli interrupt e delle chiamate di sistema, il kernel farà sempre e solo una cosa: pianificare l'esecuzione dei processi.

```
void scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor. This, (and
    // the release() call below) is important for when
    // the CPU is idle (can find no RUNNABLE proc) and
    // loops continuously. If it held the lock and looped,
    // no other CPU can do any proc-related work, such
    // as marking a proc RUNNABLE so as to de-idle this
    // CPU. Further, procs may be waiting for I/O, in
    // which case interrupts had better be on.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process. Important: It is
      // the process's job to release ptable.lock
```

```
      // and then reacquire it before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
      // sched()'s `swtch` usually enters here
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

Sleep e Wakeup

Meccanismo di sincronizzazione che permette i procesi di interagire tra loro. Sleep e wakeup consentono ai processi nello stato di sleeping di dormire in attesa di un evento e ad un altro processo di svegliarlo una volta che l'evento è avvenuto. Questi meccanismi di coordinamento sono anche chiamati **meccanismi di coordinamento sequenziale**.

Un esempio di implementazione è la seguente:

```
struct q {
     struct spinlock lock;
     void *ptr;
 };

 void* send(struct q *q, void *p)
 {
    acquire(&q->lock);
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);
    release(&q->lock);
 }

 void* recv(struct q *q)
 {
  void *p;

    acquire(&q->lock);
    while((p = q->ptr) == 0)
        sleep(q, &q->lock);
    q->ptr = 0;
    release(&q->lock);
```

```
    return p;
}
```

Questa implementazione evita il problema del *lost wake-up* ed impedisce la generazione di un eventuale *deadlock*.

Questo meccanismo non fa *busy waiting* e i metodi per utilizzarlo sono definiti in `proc.c`. La chiamata per eseguire la sleep deve necessariamente passare per la funzione `sys_sleep` contenuta nel file `sysproc.c`.

File System

Lo scopo di un file system è organizzare e archiviare i dati. I file system supportano tipicamente la condivisione di dati tra utenti e applicazioni, oltre alla persistenza in modo che i dati siano ancora disponibili dopo un riavvio. Il file system di xv6 fornisce file, directory e percorsi simili a Unix e archivia i suoi dati su un disco IDE per garantire la persistenza.

Attualmente i file xv6 sono limitati a 268 blocchi o 268*BSIZE byte (BSIZE è 1024 in xv6, `fs.h`). Questo limite deriva dal fatto che un inode xv6 contiene 12 numeri di blocco “diretti” e un numero di blocco “singolarmente indiretto”, che si riferisce a un blocco che contiene fino a 256 numeri di blocco in più, per un totale di 12 256=268 blocchi.

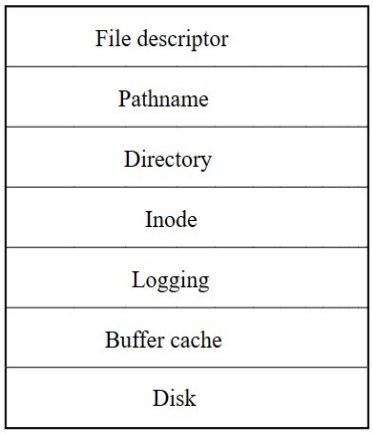

Figura 8: Livelli del file system di xv6.

L'implementazione del sistema di file di xv6 è organizzata in sette livelli, come mostrato nella Figura 8:

- Il livello **Disk**: legge e scrive blocchi su un hard disk IDE.
- Il livello **Buffer cache** ha due funzioni fondamentali:

    1. sincronizzare l'accesso ai blocchi del disco per garantire che solo una copia di un blocco sia in memoria e che solo un thread del kernel alla volta utilizzi quella copia.

       Se un *kernel thread* ha ottenuto un riferimento a un buffer ma non lo ha ancora rilasciato, le chiamate degli altri thread per lo stesso blocco rimarranno in attesa. I livelli più alti del file system si affidano alla sincronizzazione dei *buffer cache*

    2. memorizzare nella cache i blocchi più popolari in modo che non vengano riletti dal disco lento. Il codice è in `bio.c`.

Buffer cache ha un numero fisso di buffer per contenere i blocchi del disco, il che significa che se il file system richiede un blocco che non è già nella cache, la cache del buffer deve riciclare un buffer che attualmente contiene qualche altro blocco. La *buffer cache* ricicla il buffer utilizzato meno di recente per il nuovo blocco. Il presupposto è che: *il buffer utilizzato meno di recente sia quello con meno probabilità che venga utilizzato nuovamente a breve*.

- Il livello **Logging**: consente ai livelli superiori di incapsulare gli aggiornamenti a diversi blocchi in una transazione e garantisce che i blocchi vengano aggiornati atomicamente in caso di arresti anomali (cioè, tutti vengono aggiornati o nessuno).

  Xv6 risolve il problema dei crash durante le operazioni del file system con una semplice versione di *logging*. Una chiamata di sistema xv6 non scrive direttamente le strutture dati del file system su disco. Invece, inserisce una descrizione di tutte le scritture che desidera effettuare in un log sul disco. Una volta che la chiamata di sistema ha registrato tutte le sue scritture, scrive uno speciale record di commit sul disco indicando che il log contiene un'operazione completa. A quel punto la chiamata di sistema copia le scritture nelle strutture dati del file system su disco. Una volta completate tali scritture, la chiamata di sistema cancella il registro su disco.

  Se il sistema dovesse bloccarsi e riavviarsi, il codice del file system viene ripristinato dal crash come segue, prima di eseguire qualsiasi processo. Se il registro è contrassegnato come contenente un'operazione completa, il codice di ripristino copia le scritture nella posizione in cui appartengono nel file system su disco. Se il registro non è contrassegnato come contenente un'operazione completa, il codice di ripristino ignora il registro. Il codice di ripristino termina cancellando il registro.

  Un esempio di log si ha in *filewrite*:

  ```
  begin_trans();
  ilock(f->ip);
  r = writei(f->ip, ...);
  iunlock(f->ip);
  commit_trans();
  ```

- Il livello **Inode e Block Allocator**: fornisce file senza nome, ciascuno rappresentato utilizzando un `inode` (Figura 9) e una sequenza di blocchi che contengono i dati del file.

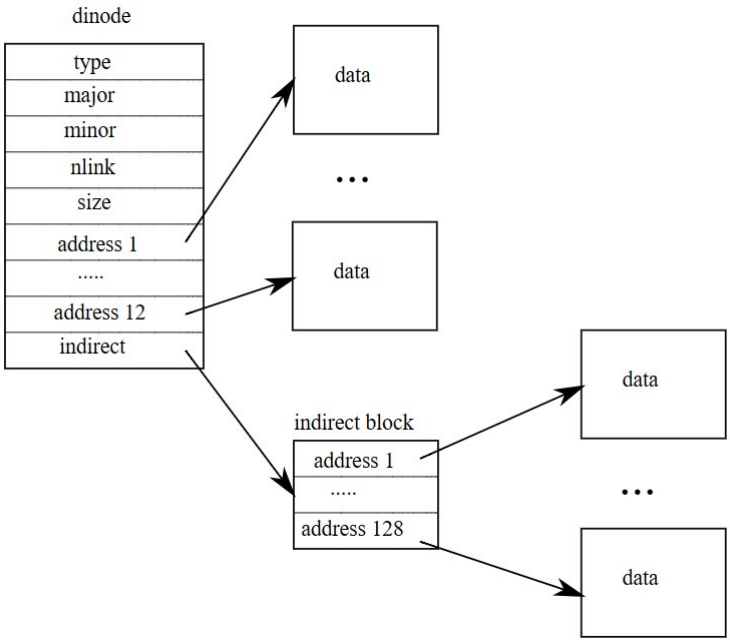

Figura 9: inode.

Il file system deve disporre di un piano per la posizione in cui archiviare gli inode e i blocchi di contenuto sul disco. Per fare ciò, xv6 divide il disco in diverse sezioni, come mostrato nella Figura 9. Il file system non utilizza il blocco 0 (contiene il settore di avvio). Il blocco 1 è chiamato *superblocco*; contiene metadati sul file system (la dimensione del file system in blocchi, il numero di blocchi di dati, il numero di inode e il numero di blocchi nel log). I blocchi che iniziano da 2 contengono inode, con più inode per blocco. Successivamente vengono visualizzati i blocchi bitmap che tengono traccia dei blocchi di dati in uso. La maggior parte dei blocchi rimanenti sono blocchi di dati che contengono il contenuto di file e directory.

Il contenuto di file e directory viene archiviato in blocchi di dischi, che devono essere allocati da un pool libero. Il *block allocator* di xv6 mantiene una bitmap libera sul disco, con un bit per blocco. Un bit 0 indica che il blocco corrispondente è libero; un bit 1 indica che è in uso. I bit corrispondenti al settore di avvio, al superblocco, ai blocchi di inode e ai blocchi di bitmap sono sempre impostati.

Il termine **inode** può avere uno dei due significati correlati:

- potrebbe fare riferimento alla struttura dei dati su disco contenente la dimensione di un file e l'elenco dei numeri dei blocchi di dati.
- potrebbe riferirsi a un inode in memoria, che contiene una copia dell'inode su disco oltre a informazioni aggiuntive necessarie all'interno del kernel.

Tutti gli inode su disco sono raggruppati in *un'area contigua* del disco chiamata *inode's block*. Ogni inode ha la stessa dimensione, quindi è facile, dato un numero n (detto *inode number*), trovare l'ennesimo inode sul disco.

L'inode su **disco** è definito da una *struct dinode*. Il campo $tipo$ distingue tra file, directory e file speciali (dispositivi). Un tipo zero indica che un inode su disco è libero. Il campo $nlink$ conta il numero di voci della directory che fanno riferimento a questo inode, per riconoscere quando l'inode dovrebbe essere liberato. Il campo $dimension$ registra il numero di byte di contenuto nel file. L'array degli $indirizzi$ registra i numeri dei blocchi del disco che contengono il contenuto del file.

Il **kernel** mantiene in memoria l'insieme degli inode attivi; *struct inode* è la copia in memoria di una *struct dinode* (`file.h`) su disco. Il kernel memorizza un inode in memoria solo se ci sono puntatori C che fanno riferimento a quell'inode. Il campo ref conta il numero di puntatori C che si riferiscono all'inode in memoria e il kernel scarta l'inode dalla memoria se il conteggio dei riferimenti scende a zero. Le funzioni `iget` e `iput` acquisiscono e rilasciano puntatori a un inode, modificando il conteggio dei riferimenti.

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

- Il livello **Directory inodes**: implementa ogni directory come un tipo speciale di inode il cui contenuto è una sequenza di voci di directory, ciascuna delle quali contiene un nome e un riferimento i all'inode del file indicato.
- Il livello **Pathname**: fornisce nomi di path gerarchici come `/usr/rtm/xv6/fs.c` e li risolve con una ricerca ricorsiva.
- Il livello **File Descriptor**: astrae molte risorse Unix (ad esempio, pipe, dispositivi, file, ecc.) utilizzando l'interfaccia del sistema di file, semplificando la vita degli sviluppatori.

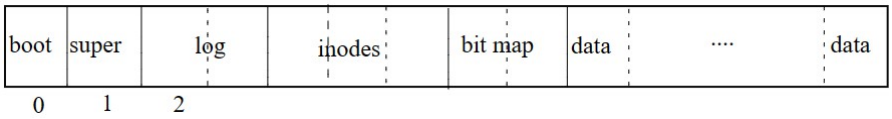

Figura 10: Struttura del file system di xv6.

Un aspetto dell'interfaccia Unix è che la maggior parte delle risorse in Unix sono rappresentate come file, inclusi dispositivi come la console, le pipe e, naturalmente, i file reali. Il livello *file descriptor* è il livello che raggiunge questa uniformità.

Xv6 fornisce a ogni processo la propria *tabella di file aperti*, o descrittori di file. Ogni file aperto è rappresentato da un file struct, che è un wrapper attorno a un inode o a una pipe, più un i/o compensare. Ogni chiamata a open crea un file newopen (un nuovo file struct): se più processi aprono lo stesso file in modo indipendente, le diverse istanze avranno offset di i/o diversi. D'altra parte, un singolo file aperto (lo stesso file struct) può apparire più volte nella tabella dei file di un processo e anche nelle tabelle dei file di più processi. Ciò accadrebbe se un processo utilizzasse open per aprire il file e poi creasse alias utilizzando dup o lo condividesse con un bambino utilizzando fork. Un conteggio dei riferimenti tiene traccia del numero di

riferimenti a un particolare file aperto. Un file può essere aperto in lettura o in scrittura o in entrambi. I campi leggibili e scrivibili tengono traccia di ciò.

Tutti i file aperti nel sistema sono conservati in una tabella di file globale, la `ftable` che ha funzioni per: allocare un file (`filealloc`), creare un riferimento duplicato (`filedup`), rilasciare un riferimento (`fileclose`) e leggere e scrivere dati (`fileread` e `filewrite`).

Os161

Os161 è un sistema operativo didattico progettato per insegnare i principi dei sistemi operativi moderni. È simile a Unix e basato su BSD e include un kernel, un simulatore di macchina MIPS e un insieme di programmi di sistema e di utente, insieme a una serie di esercizi di programmazione. Os161 è stato sviluppato presso l'Università di Harvard e l'Università di Toronto.

Struttura del Kernel

Os161 è basato su un'architettura a microkernel, il che significa che il kernel è **modulare**(diverse funzionalità sono suddivise in moduli separati. Questi moduli possono essere sviluppati, testati e sostituiti in modo indipendente) e divide le funzionalità del sistema operativo in componenti distinti. Questi componenti possono comunicare tra loro attraverso meccanismi di messaggistica.

La *memoria* nel kernel di 0s161 è solitamente suddivisa in tre parti principali:

1. *Kernel Text*: Questa è l'area di memoria in cui risiede il codice eseguibile del kernel. Contiene le istruzioni necessarie per eseguire le operazioni di base del sistema operativo.
2. *Kernel Data*: Questa area memorizza le variabili globali e i dati utilizzati dal kernel. Include strutture dati come tabelle di processo, informazioni sulla gestione della memoria e altre variabili di stato del kernel.
3. *Heap e Stack Kernel*: Queste aree sono utilizzate per la gestione dinamica della memoria all'interno del kernel. Lo heap viene utilizzato per l'allocazione dinamica di strutture dati, mentre lo stack gestisce le chiamate alle funzioni e le variabili locali.

Processi

Utilizza una struttura di processo simile, ma l'implementazione può variare a seconda dell'architettura di destinazione.macchina privata con memoria e CPU dedicate. Di solito è necessario che l'intero processo debba essere interamente in memoria prima della sua esecuzione, in Os161 la *virtual memory* permette che il processo sia solo parzialmente in memoria. Le istruzioni invece devono essere in memoria fisica.

La *virtual memory* è la separazione tra *userl logical memory* dalla *memoria fisica*. Solo la parte di programma da eseguire deve essere in memoria e lo spazio degli indirizzi logici puo' anche essere più grande dello spazio degli indirizzi fisici. Questo permette a più programmi di funzionare contemporaneamente, maggiori performance e velocità. Può essere implementata tramite *demand paging* oppure *demand segmentation*

Lo *spazio di indirizzamento virtuale* fa riferimento alla vista logica di come i processi sono salvati in memoria: partono dall'indirizzo 0 e con indirizzi contigui fino a fine spazio:

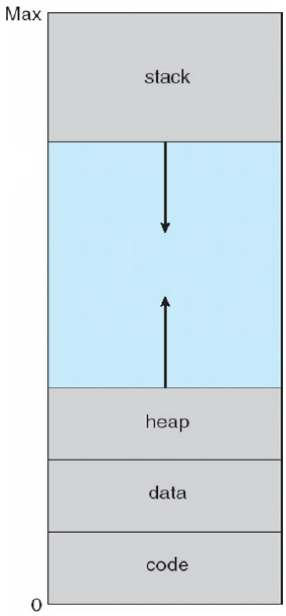

Figura 11: Stack del Kernel dopo una chiamata

Nel kernel OS/161, i processi sono organizzati attraverso una struttura dati denominata **coda di esecuzione dei processi**. La coda è implementata come una lista doppiamente concatenata, e il suo header si trova nel file `kern/include/threadlist.h`. Questa lista contiene i processi attualmente in esecuzione sul sistema.

```
struct threadlistnode {
  struct threadlistnode *tln_prev;
  struct threadlistnode *tln_next;
  struct thread *tln_self;
};

struct threadlist {
  struct threadlistnode tl_head;
  struct threadlistnode tl_tail;
  unsigned tl_count;
};
```

Ogni CPU ha la sua coda di esecuzione dei processi, e la lista contiene le informazioni essenziali sui processi in esecuzione, come il **PID** (Process ID) e altre proprietà associate al processo. L'utilizzo di una lista doppiamente concatenata consente un accesso efficiente e una gestione flessibile dei processi.

In OS161 esiste una struttura dati chiamata `Process Table`. Questa tabella dei processi mappa un PID (Process ID) ad una struttura dati associata a un processo. Il suo scopo principale è tenere traccia dei processi in esecuzione nel sistema. Ogni processo ha un PID univoco assegnato dalla tabella.

La struttura del processo in Os161 è descritta in `process.h` e tutte le funzioni annesse sono sviluppate in `process.c`:

```
/* Process Structure */
struct process {
  char *p_name;
  pid_t p_id;
```

```
  /* Internal Stuff */
  struct processlistnode p_listnode; /* Link for run/sleep/zombie lists */

  /*For waitpid()*/
  struct semaphore *p_waitsem;
  // struct cv *p_waitcv;
  // struct lock *p_waitlock;
  //struct processlist p_waiters;

  pid_t p_parentpid;
  //struct processlist p_children;
  struct thread *p_thread;

  /* For fork() */
  // struct semaphore *p_forksem;

  // Array of file handle pointers; initialize to NULL pointers on process
creation.
  struct file_handle* p_fd_table[FD_MAX];
};

/* States a process can be in. */
typedef enum {
  P_FREE,       /* available (no process) */
  P_USED,       /* unavalable (process running) */
  P_ZOMBIE, /* zombie (process exited) */
  P_INVALID /* Invalid PID */
} pidstate_t;

struct process *init_process_create(const char*);

int process_create(const char*, pid_t parent, struct process**);
void process_exit(pid_t pid, int exitcode);
void process_destroy(pid_t pid);
int process_wait(pid_t pidToWait, pid_t pidToWaitFor);
/* Call once during system startup to allocate data structures */
void processtable_bootstrap(void);
void processtable_biglock_acquire(void);
void processtable_biglock_release(void);
bool processtable_biglock_do_i_hold(void);
int allocate_pid(pid_t* allocated_pid);
void release_pid(int);
struct process* get_process(pid_t);
pidstate_t get_pid_state(pid_t);
pid_t get_process_parent(pid_t);
int get_free_file_descriptor(pid_t);                // Given a process id, returns
a file descriptor that is free.
void release_file_descriptor(pid_t, int fd);        // Closing file, so release fd
struct file_handle* get_file_handle(pid_t, int fd); // Given a file descriptor,
returns the pointer to the associated file handle.
int get_process_exitcode(pid_t);
void set_process_parent(pid_t,pid_t);
void abandon_children(pid_t);
```

```
void collect_children(void);
void process_cleanup(void);
```

Inoltre vi è una seconda struttura chiamata `processlist` che è parte dell'implementazione del kernel e si occupa di gestire la lista dei processi in esecuzione nel sistema operativo. Essa è essenziale per tenere traccia di tutti i processi attivi, ciascuno identificato da un PID (Process ID). La struttura dati associata a questa lista contiene informazioni cruciali su ogni processo, consentendo al kernel di monitorare e gestire correttamente l'esecuzione dei processi.

Gestione della Memoria

La *gestione della memoria* in Os161 è fondamentale per garantire un ambiente operativo stabile ed efficiente. Come in xv6 adotta la *virtual memory* tramite l'utilizzo di *page table*. Le page tables sono un componente chiave nella gestione della memoria di Os161, svolgendo un ruolo fondamentale nell'ottimizzazione delle prestazioni e nella garanzia di sicurezza e isolamento tra i processi. Inoltre, Os161 implementa varie tecniche avanzate per ottimizzare l'utilizzo della memoria, tra cui l'utilizzo di memoria condivisa (shared memory), il demand paging e la condivisione in lettura/scrittura (Copy-On-Write, COW).

La **memoria condivisa** consente a più processi di accedere e condividere una stessa area di memoria, facilitando la comunicazione e lo scambio di dati tra di essi. Questa tecnica è utile in scenari in cui processi diversi necessitano di collaborare o condividere informazioni senza dover ricorrere a complesse operazioni di copia.

Il **demand paging** è una strategia che consente di caricare in memoria solo le pagine necessarie per l'esecuzione di un processo, ritardando il caricamento delle pagine non ancora utilizzate. Ciò ottimizza l'utilizzo della memoria, riducendo il carico iniziale e consentendo una gestione più efficiente delle risorse di sistema.

La **condivisione in lettura/scrittura** (**Copy-On-Write**, **COW**) è una tecnica che consente a più processi di condividere la stessa area di memoria in modalità di sola lettura. Quando un processo tenta di scrivere in una pagina condivisa, la pagina viene duplicata in modo trasparente, garantendo che ciascun processo abbia la propria copia modificabile. Questo approccio riduce il consumo di memoria e migliora le prestazioni.

System Call, Exceptions e Interrupts

Simile a xv6, gestisce system calls, interrupts ed exceptions in modo da facilitare l'esecuzione del sistema operativo.

In OS/161, le *system call* vengono gestite in modalità kernel. Quando un'applicazione utente esegue una system call, il controllo passa al kernel. Quando avviene un **trap** (o un **interrupt**) per la chiamata di una *system call*, OS/161 (come xv6) passa dalla modalità utente alla modalità kernel, nota anche come "modalità privilegiata". Questo è reso possibile grazie alle istruzioni specifiche delle architetture dei processori moderni. Quando il kernel gestisce una *system call*, salva il contesto dell'applicazione utente (registri, puntatori di stack, ecc.), esegue il codice della *system call* e poi ripristina il contesto dell'applicazione utente prima di restituire il controllo all'applicazione.

Os161 supporta una gamma più ampia di system call che riflettono meglio le funzionalità di un sistema operativo completo. Include operazioni di gestione dei processi, gestione dei file, comunicazione tra processi, sincronizzazione e altro.

- *Astrazione e modularità*: OS161 promuove una maggiore astrazione e modularità rispetto a xv6. Le system call sono implementate come chiamate remote attraverso messaggi, il che consente una maggiore separazione tra il codice utente e il codice kernel.
- *Isolamento e protezione*: Grazie alla separazione dei servizi e dei componenti nel microkernel di OS161, è più probabile che il sistema operativo offra un maggiore isolamento e protezione tra le diverse componenti. Questo è vantaggioso in termini di sicurezza e stabilità

In Os161 le *system call* sono gestite in questo modo:

- **Interruzioni (Trappole)**: Quando un'applicazione utente richiede un servizio del kernel attraverso una chiamata di sistema, il controllo passa dalla modalità utente alla modalità kernel. Questo passaggio avviene attraverso una trappola o un'interruzione.
- **Interrupt Handler del Kernel**: Una volta che l'interruzione o la trappola viene generata, il controllo passa all'interrupt handler del kernel: una porzione di codice del kernel che gestisce gli eventi generati dalle interruzioni e dalle trappole.
- **Gestione delle Chiamate di Sistema**: Il kernel di OS/161 ha una tabella delle chiamate di sistema (*system call table*) che associa i numeri delle chiamate di sistema alle funzioni del kernel corrispondenti. Ad esempio, il numero 0 potrebbe essere associato a una chiamata di sistema per terminare un processo, il numero 1 per scrivere su un file, il numero 2 per leggere da un file, e così via.

Esempio di definizione di una tabella delle chiamate di sistema in OS/161:

```
// Definizione della tabella delle chiamate di sistema in OS/161
typedef int (*syscall_function_t)(void);
syscall_function_t syscall_table[SYSCALL_COUNT];
```

- **Esecuzione delle Chiamate di Sistema**: Una volta individuata la chiamata di sistema richiesta e associata alla sua funzione corrispondente, il kernel esegue il codice della funzione di chiamata di sistema.
- **Restituzione dei Risultati**: Dopo l'esecuzione della chiamata di sistema, il controllo ritorna all'applicazione utente, e il risultato della chiamata di sistema (ad esempio, il valore restituito da una funzione di lettura/scrittura) può essere restituito all'applicazione.
- **Ripristino del Contesto**: Durante il passaggio dalla modalità utente alla modalità kernel, e viceversa, il kernel salva e ripristina il contesto del processo corrente, compresi i registri, lo stack e altre informazioni importanti. Questo è necessario per garantire che l'esecuzione possa riprendere correttamente dopo una chiamata di sistema.

Ogni numero di chiamata di sistema è associato a una funzione specifica nel kernel. Ecco un esempio semplificato di come potrebbe apparire la definizione e l'inizializzazione della tabella delle chiamate di sistema in OS/161:

```
// Definizione delle chiamate di sistema in OS/161
#define SYS_HALT 0
#define SYS_READ 2
// ... altre chiamate di sistema ...
// Inizializzazione della tabella delle chiamate di sistema
syscall_table[SYS_HALT] = sys_halt;
```

```
syscall_table[SYS_READ] = sys_read;
// ... inizializzazione di altre chiamate di sistema ...
```

Una volta che una chiamata di sistema è stata richiesta dall'applicazione utente, il kernel esegue la funzione associata all'interno della tabella delle chiamate di sistema.

```
int sys_read(int filehandle, char *buf, size_t size) {
    // Implementazione della lettura dal filehandle nel buffer
    // Restituzione del numero di byte letti o di un valore di errore }
```

Sincronizzazione

Nella versione più aggiornata vi è la possibilità di lavorare su dispositivi multiprocessori, quindi la gestione e l'accesso alle *sezioni critiche* è fondamentale. In *Os161* vengono riproposti i `mutex lock` per proteggere le *sezioni critiche* e evitare *race conditions*:

1. Un processo acquisisce il *lock* prima di entrare nella *sezione critica* usando la funzione `acquire()`
2. Rilascia il *lock* quando esce dalla sezione di interesse del codice grazie alla funzione `release()`

```
while (true){
  #acquire lock
  critical section
  release lock
  remainder section
}
```

In Os161 troviamo un lock in particolare: `Spinlock` (*kern/thread/spinlock.c*):

```
...
while (1) {
  if (spinlock_data_get(&splk -> splk_lock) != 0)
      continue;   # verifica che il lock non sia già stato acquisito da un altro
thread - 0: da acquisire / 1: acquisito
  if (spinlock_data_testandset(&splk -> splk_lock) != 0)
      continue;  # acquisisce il lock
  break;
}
/*La funzione `spinlock_data_testandset()` è una funzione atomica che imposta la
variabile su 1 solo se era 0, al tramonto dell'istruzione restituisce il valore
precedente. Se non era 0, l'istruzione viene ripetuta.*/
...
```

Os161 però utilizza altre tecniche di sincronizzazione più sofisticate; tra queste ci sono i `semafori`. Un semaforo **S** è un valore intero che permette l'accesso a due operazioni **atomiche**: `signal` e `wait()`

```
wait(S) {
  while(S<=0);
    //busy wait
    S--;
}
Decrementa il contatore. Se il contatore diventa negativo, il processo o thread
viene messo in attesa.

signal(S){
  S++;
}
Incrementa il contatore. Se ci sono processi o thread in attesa, uno di essi viene
risvegliato.
```

In Os161 il codice relativo ai semafori: creazione,gestione ecc può essere letto in `kern/thread/synch.c` e `kern/thread/synch.h`

```
struct semaphore{
    int count;
    char *name;
};
```

A differenza di *xv6*, in Os161 i processi hanno un campo extra che ne definisce la `priorità`. In ambito di sincronizzazione è un fattore di cui tener conto: si disabilitano gli *interrupts* del timer prima dell'ingresso nella *sezione critica* e si riabilitano all'uscita. Questa tecnica si basa sul fattocceh i thread con priorità più bassa **non possono interrompere** l'esecuzione di un thread con priorità superiore. In altre parole, i thread con priorità superiore devono attendere che un thread con priorità superiore abbandoni la *sezione critica* prima di poter essere eseguiti. Una semplice interfaccia per la gestione dell'interrupt è nel file: `kern/arch/mips/include/spl.h`.

Os161 usa anche i `lock`:

```
struct lock *mylock = lock create("LockName");
lock_aquire(mylock);
// critical section
lock_release(mylock);
```

I lock sono molto simili a dei *semafori binari* conun valore iniziale $S = 1$ ma con un ulteriore imposizione: *il thread che rilascia il lock deve essere lo stesso che lo ha acquisito*

Vi sono altre tecniche che permettono di ottimizzare la sincronizzazione di thread e/o processi, tra queste vi sono le `condition variables` che permettono ai thread di sospendersi quando non sono in grado di procedere a causa di una condizione specifica. Questa *variabile* funge da meccanismo di notifica, consentendo ad altri thread di segnalare al thread in attesa che la condizione desiderata è stat soddisfatta. Solo quando il thread viene risvegliato e riacquisisce il lock, può procedere nell'esecuzione.

```
lock_acquire(lock);
while (condition not true)
cv_wait(cond,lock);
... // do stuff
lock_release(lock);

lock_acquire(lock);
... // modify condition
cv_signal(cond);
lock_release(lock);
```

Una variante di queste *condition variables* ma che usano **spinlocks** (al posto dei semafori) sono i `wait channels`, usati per le sincronizzazioni a livello kernel.

Scheduling

OS161 fornisce per impostazione predefinita una semplice **coda unica round-robin**. Funziona così:

1. `hardclock` from `kern/thread/clock.c` verrà chiamato periodicamente (dal gestore dell'interruzione dell'orologio hardware)
2. A seconda di come impostato lo `schedule`, si definisce l'ordine
3. Quindi chiamerà `thread_yield` per far sì che il $thread_1$ corrente ceda la CPU al $thread_2$. $thread_1$ va in sleep

A differenza di *xv6*, posso *giocare* con le `priorità` in modo da assegnare ad ogni thread un' *importanza* più o meno alta a seconda della funzionalità o contesto. Usiamo la funzione `schedule` per dare ai *thread interattivi* una priorità più alta. Ci sono due ragioni:

- *Il tuo tempo è più prezioso di quello del computer*. Quindi, in generale, dovremmo servire prima quei thread che interagiscono con te. Ad esempio, non vuoi aspettare che il computer sia in una shell mentre è impegnato a eseguire il backup, giusto?
- *I thread interattivi tendono ad essere vincolati all'I/O, il che significa che spesso rimangono bloccati in attesa di input o output*. Quindi normalmente non riescono a consumare il tempo concesso loro. In questo modo possiamo passare ai thread vincolati al calcolo quando si bloccano e aumentare l'utilizzo del computer.

Se *state* di un thread è **S_READY**, significa che il thread corrente ha consumato tutto il suo intervallo di tempo ed è costretto a cedere a un altro thread (tramite il timer haerdware). Quindi possiamo supporre che non sia interattivo o che richieda molti calcoli. Tuttavia, se *newstate* è **S_SLEEP**, significa che il thread corrente si offre di cedere a un altro thread , magari in attesa di I/O o di un mutex. Quindi possiamo supporre che questo thread sia più interattivo o che richieda I/O.

File System

Il file system nativo di OS/161 è chiamato SFS (Simple File System). Sebbene sia un'implementazione relativamente ingenua, SFS vanta la maggior parte delle funzionalità di base che ti aspetteresti da un

filesystem:

- Directory gerarchiche
- Supporto di file di grandi dimensioni tramite blocchi indiretti multilivello
- Blocco a grana fine per una maggiore concorrenza
- Una cache buffer in memoria di dati e metadati del file system

Manca il supporto per il file system, inteso comme insieme di system calls che forniscono alcune operazioni sui files. Tale supporto, oltre alla realizzazione delle singole funzioni, necessita di opportune strutture dati che permettono la ricerca dei files e la loro identificazione. Il **VFS** - *Virtual file system*- è in `kern/vfs.c` e `kern/include/vfs.h`, dove la *struct vnode* rappresenta un file e vi sono alcune funzionalità di base come `vfs_open` e `vfs_close`.

```
struct vnode {
char *path; //percorso del file/directory
unsigned int permissions; //Permessi di accesso al file/directory
size_t size; //dimensione del file (solo per i file)
//altro...
}
```

Confronto tra Os161 e Xv6

- **Struttura Semplice**:
    - **Xv6**: É progettato con un'architettura relativamente semplice e compatta, facilitando la comprensione e lo studio dei principi fondamentali dei sistemi operativi.
    - **Os161**: Analogamente, Os161 è progettato per l'istruzione e sviluppato presso l'Università di Harvard. Si concentra sull'apprendimento dei principi dei sistemi operativi.
- **Architettura di Destinazione**:
    - **Xv6**: Inizialmente progettato per l'architettura x86. Successivamente, è stata sviluppata una versione per RISC-V.
    - **Os161**: Supporta diverse architetture, inclusi MIPS, ARM e Intel.
- **Organizzazione del Kernel**:
    - **Xv6**: É un kernel monolitico, il che significa che l'intero sistema operativo opera in modalità kernel. Questo design semplifica la cooperazione tra diverse parti del sistema.
    - **Os161**: Utilizza un'organizzazione a microkernel, dove solo le funzioni essenziali risiedono nel kernel, mentre i servizi aggiuntivi sono implementati come server a livello utente.
- **Gestione della Memoria**:
    - **Xv6**: Implementa un sistema di gestione della memoria basato su paging. Le page tables consentono il controllo degli indirizzi di memoria, permettendo a Xv6 di multiplexare gli spazi degli indirizzi di processi diversi su una singola memoria fisica e proteggere le memorie di processi diversi.
    - **Os161**: Anche Os161 utilizza page tables e presenta concetti di gestione della memoria simili.

- **Gestione dei Processi**:
    - **Xv6**: I processi costituiscono l'unità fondamentale di isolamento. Ogni processo ha il proprio spazio degli indirizzi virtuale.
    - **Os161**: Utilizza una struttura di processo simile, ma l'implementazione può variare a seconda dell'architettura di destinazione.macchina privata con memoria e CPU dedicate.
- **System Call, Exceptions e Interrupts**:
    - **Xv6**: Gestisce system call, eccezioni e interrupts attraverso un meccanismo comune. Le interfacce di sistema sono implementate tramite trap (o interrupt), e il kernel decide come gestire l'evento in base al numero di trap.
    - **Os161**: Simile a xv6, gestisce system calls, interrupts ed exceptions in modo da facilitare l'esecuzione del sistema operativo.
- **Sincronizzazione**:
    - **Xv6**: Utilizza spin-lock e sleep-lock per garantire l'accesso esclusivo alle risorse condivise tra i processi. Gli spin-lock sono utilizzati in sezioni critiche di breve durata, mentre gli sleep-lock supportano il rilascio temporaneo del processore.
    - **Os161**: Implementa meccanismi di sincronizzazione simili, come semafori e lock.
- **Scheduling**:
    - **Xv6**: Utilizza un algoritmo di scheduling a round-robin, assegnando a ciascun processo un quantum di tempo.
    - **Os161**: Utilizza un algoritmo di scheduling simile, come round-robin o altri algoritmi in base alla configurazione.
- **File System**:
    - **Xv6**: Fornisce un sistema di file semplificato, ispirato a quello presente in Unix V6. Include concetti come la gestione di directory, la lettura e scrittura di file, e altre operazioni di base di un sistema di file.
    - **Os161**: Implementa un file system simile, che supporta file, directory e percorsi simili a Unix.

Conclusioni

In generale, entrambi xv6 e OS/161 forniscono un'esperienza pratica nell'implementazione dei concetti chiave dei sistemi operativi, ma differiscono nell'approccio e nell'architettura di destinazione.

Fonti

https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf