Analisi comparativa tra Os161 e Xv6

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.

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.

Organizzazione del Kernel

Il sistema operativo è organizzato in modo che risieda tutto all'interno del kernel in modo che le implementazioni di tutte le chiamate di sistema vengano eseguite in modalità kernel. Questa organizzazione è chiamata kernel monolitico.

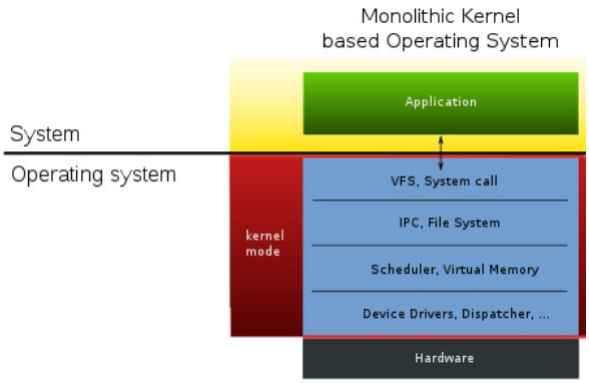

Figura 1: Kernel monolitico

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 un errore in modalità kernel spesso comporta il fallimento

del kernel. 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.

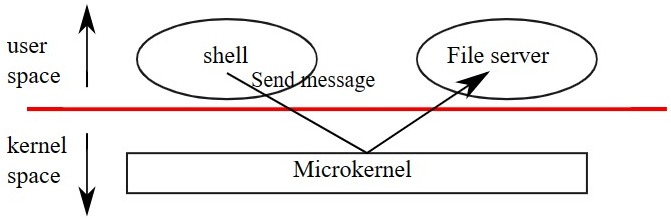

Figura 2: Microkernel

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.

Xv6 è implementato come un kernel monolitico, seguendo la maggior parte dei sistemi operativi Unix. Pertanto, in Xv6, l'interfaccia del kernel corrisponde all'interfaccia del sistema operativo e il kernel implementa l'intero sistema operativo. 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 tabelle delle pagine (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.

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.

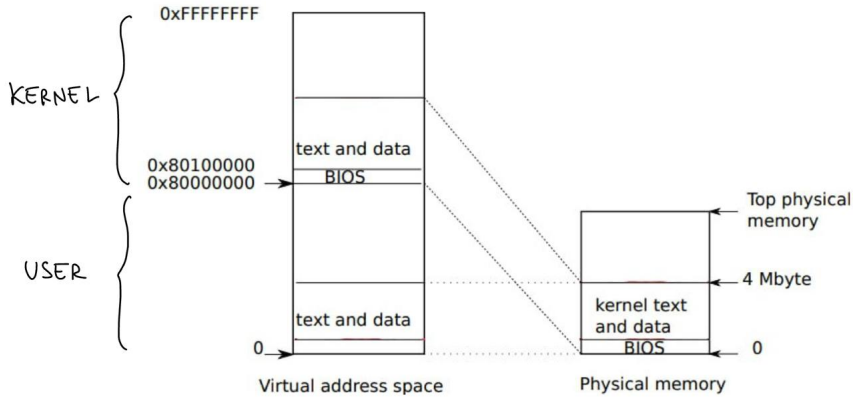

Figura 3: Layout dello spazio degli indirizzi virtuali di un processo.

Page Tables

Le tabelle delle pagine sono il meccanismo attraverso il quale il sistema operativo controlla il significato degli indirizzi di memoria. Consentono a Xv6 di multiplexare gli spazi degli indirizzi di processi diversi su una singola memoria fisica e di proteggere le memorie di processi diversi. Xv6 utilizza principalmente le tabelle delle pagine per multiplexare gli spazi degli indirizzi e proteggere la memoria. Utilizza anche alcune semplici astuzie con le tabelle delle pagine: mappare la stessa memoria (il kernel) in diversi spazi degli indirizzi, mappare la stessa memoria più di una volta in uno spazio degli indirizzi (ogni pagina utente è anche mappata nella vista fisica della memoria del kernel) e proteggere uno stack utente con una pagina non mappata.

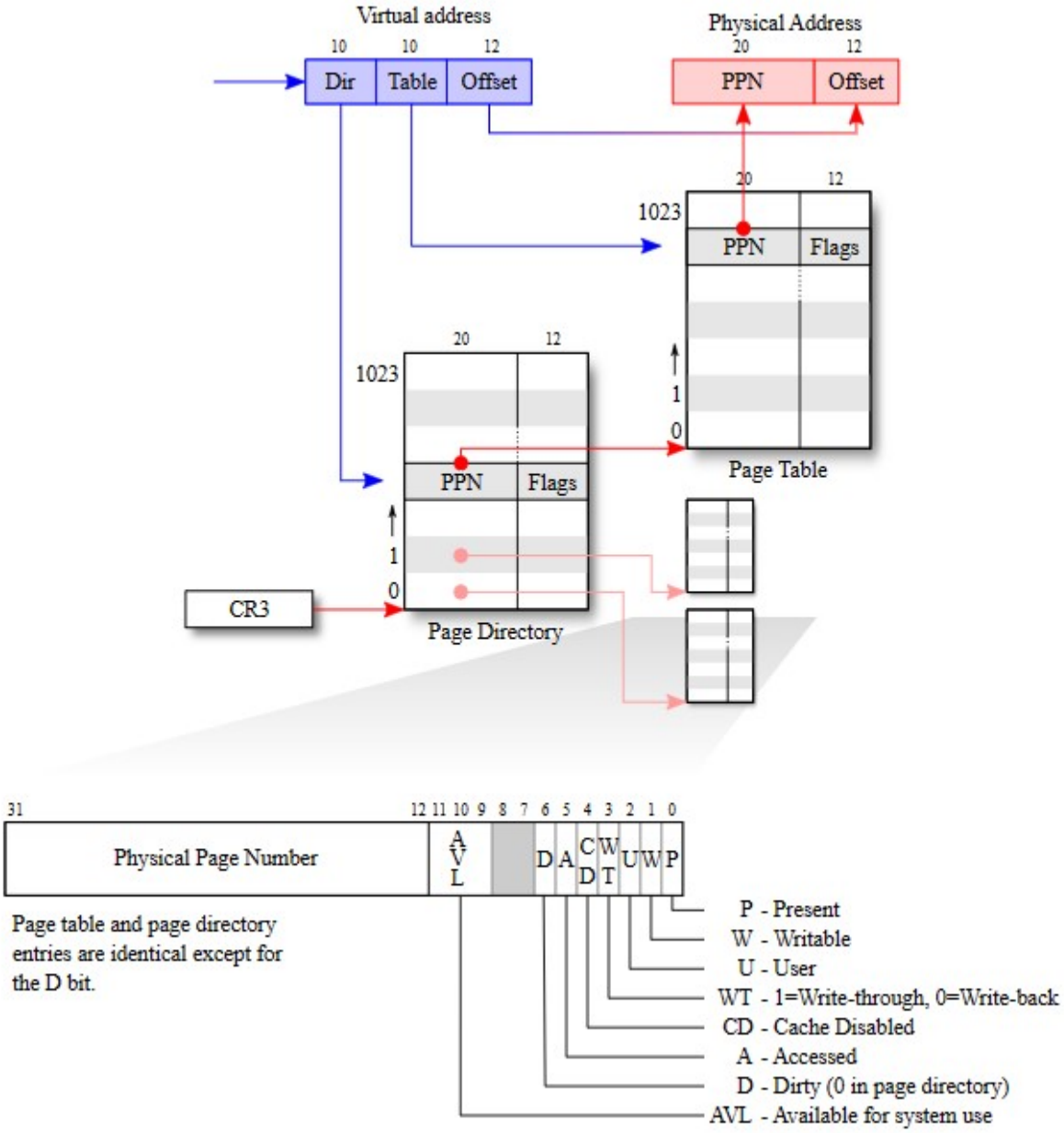

Figura 4: Page table di x86

Le istruzioni x86 (sia utente che kernel) manipolano indirizzi virtuali. La RAM della macchina, o memoria fisica, è indicizzata con indirizzi fisici. L'hardware di paginazione x86 collega questi due tipi di indirizzi, mappando ogni indirizzo virtuale in un indirizzo fisico. Una tabella delle pagine x86 è logicamente un array di 2^20 (1.048.576) page table entries (PTEs). Ogni PTE contiene un numero di physical page number (PPN) di 20 bit e alcune bandiere. L'hardware di paginazione traduce un indirizzo virtuale utilizzando i suoi primi 20 bit per indicizzare la tabella delle pagine e trovare una PTE, sostituendo i primi 20 bit dell'indirizzo con il PPN nella PTE.

La traduzione effettiva avviene in due fasi. Una tabella delle pagine è memorizzata in memoria fisica come un albero a due livelli. La radice dell'albero è un page directory di 4096 byte che contiene 1024 riferimenti simili a PTE a pagine di tabella delle pagine. Ogni pagina di tabella delle pagine è un array di 1024 PTE da 32 bit. L'hardware di paginazione 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.

System Call, Exceptions e Interrupts

Esistono tre casi in cui il controllo deve passare da un programma utente al kernel:

1. Una **System Call**: quando un programma utente richiede un servizio del sistema operativo.
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 predisporre quanto segue. Il sistema deve salvare i registri del processore per un futuro ripristino trasparente. Il sistema deve essere configurato per l'esecuzione nel kernel. Il sistema deve scegliere un punto in cui far iniziare l'esecuzione del kernel. 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.

Codice

Lo scatenarsi di una trap avviene a livello assembly con l'uso del comando *int n* dove n è il numero della trap. La prima cosa che viene fatta è il context switching, ovvero il passaggio da user mode a kernel mode. In seguito viene chiamata la funzione C trap che guarda al trap number e decide se è una system call, un'eccezione o un'interrupt. In base a questo viene chiamata la funzione corrispondente.

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.

Xv6 programma l'hardware x86 per eseguire uno switch di stack su una trap impostando un descrittore del segmento di attività attraverso il quale l'hardware carica un selettore di segmento di stack e un nuovo valore per %esp. La funzione *switchuvm* memorizza l'indirizzo della cima dello stack del kernel del processo utente nel descrittore del segmento di attività.

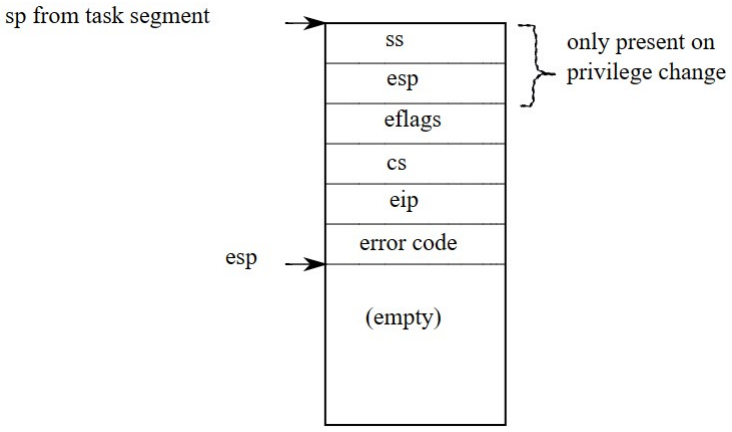

Figura 5: Stack del Kernel dopo una chiamata *int*

Quando si verifica una trap, l'hardware del processore esegue le seguenti operazioni. Se il processore stava eseguendo in modalità utente, carica %esp e %ss dal descrittore del segmento di attività, spinge il vecchio %ss utente e %esp sul nuovo stack. Se il processore stava eseguendo in modalità kernel, nulla di quanto sopra accade. Il processore poi spinge i registri %eflags, %cs e %eip. Per alcune trap (ad esempio, un page fault), il processore spinge anche una parola di errore. Il processore carica quindi %eip e %cs dall'entry corrispondente nella IDT. Xv6 utilizza uno script Perl per generare i punti di ingresso a cui puntano le voci della IDT. Ciascuna voce spinge un codice di errore se il processore non lo ha fatto, spinge il numero di interruzione e quindi salta a *alltraps*. Alltraps salva i registri del processore in modalità utente, carica il descrittore del segmento di attività del kernel e salta a trap andando così ad eseguire il codice C del kernel. In seguito il frame di trap ripristina i registri in modalità utente e quindi iret salta di nuovo nello spazio utente. Nel caso di trap verificatesi durante l'esecuzione del kernel il processore non esegue lo switch di stack.

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 indice le tabelle delle chiamate di sistema. 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 (3708). Le chiamate di sistema restituiscono convenzionalmente numeri negativi per indicare errori, numeri positivi per indicare successo.

Synchronization

Xv6 è progettato per eseguire su multiprocessori: computer con più CPU che eseguono operazioni in modo indipendente. Queste CPU condividono la RAM fisica e 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.

Lock

Una lock fornisce l'esclusione reciproca, garantendo che solo una CPU alla volta possa detenere la lock. Se una lock è associata a ciascun elemento dati condiviso e il codice tiene sempre la lock associata quando utilizza un determinato elemento, possiamo essere certi che l'elemento viene utilizzato solo da una CPU alla volta. Xv6 molto spesso utilizza i lock per evitare le race conditions.

Xv6 ha due tipi di lock: spin-lock e sleep-lock. Xv6 rappresenta uno spin-lock come una struttura chiamata struct spinlock (1501). Il campo importante nella struttura è locked, una parola che è zero quando la lock è disponibile e diversa da zero quando è detenuta. Logicamente, Xv6 dovrebbe acquisire una lock eseguendo del codice come segue:

```
void
 acquire(struct spinlock *lk)
 {
      for(;;) {
            if(!lk->locked) {
                  lk->locked = 1;
                  break;
            }
      }
 }
```

Sfortunatamente, questa implementazione non garantisce l'esclusione reciproca su un multiprocessore. Potrebbe accadere che due CPU raggiungano simultaneamente a controllare la condizione dell'if, vedano che lk->locked è zero, e quindi entrambe acquisiscano la lock eseguendo la riga successiva. A questo punto, due CPU diverse detengono la 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 la 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 la

lock. Se un processo dimentica di rilasciare una lock, queste informazioni possono aiutare a identificare il responsabile. Questi campi di debug sono protetti dalla lock e devono essere modificati solo mentre si detiene la lock.

La funzione release (1602) è l'opposto di acquire: cancella i campi di debug e quindi rilascia la 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.

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.

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

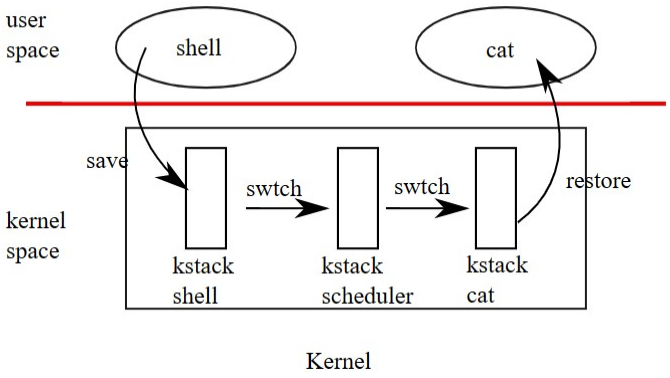

Figura 6: Switch da un processo utente ad un altro.

La Figura 6 illustra i passaggi coinvolti nel passaggio da un processo utente a un altro: una transizione utente-nucleo (chiamata di sistema o interruzione) al thread kernel del vecchio processo, un cambio di contesto al thread scheduler della CPU corrente, un cambio di contesto al thread kernel di un nuovo processo e un

ritorno alla trappola al processo a livello utente. 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.

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.

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.

Scheduling

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.

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 tabelle delle pagine 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 tabelle delle pagine 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.

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.

Sources:

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