Analisi Comparativa SO: OS161 vs xv6

Made by:
Edoardo Morello s317611
Fabio Mangani s313373
Gabriele Medica s317657

Intro

Sia os161 che xv6 sono dei sistemi operativi creati a scopo educativo, quindi con l'obiettivo di essere impiegati in ambito scolastico per permettere la comprensione del funzionamento di un sistema operativo partendo dalle basi.

Os161 è di per sè più complesso, in quanto il suo scopo è quello di emulare il più possibile i sistemi operativi odierni, mentre xv6 è più semplice e serve a capire le caratteristiche basilari che accomunano tutti i sistemi operativi.

Seppure sotto diversi punti di vista questi due sistemi operativi sono simili, in questa presentazione andremo a mettere in luce le loro differenze sostanziali.

System Calls

La progettazione delle system call ha come obiettivo il garantire un'interfaccia sicura e controllata tramite la quale il codice utente ha la possibilità di richiamare funzionalità del kernel, come ad esempio la lettura o scrittura su di un file, la creazione e gestione di processi o allocazione di memoria.

Quando un programma richiede l’esecuzione di una system call, avviene un passaggio temporaneo dalla modalità utente a quella kernel, ciò consente al sistema operativo di eseguire l’operazione richiesta.

Le system call in xv6

Quando un processo utente richiede l'esecuzione di una system call, deve far riferimento ad un numero specifico che identifica la chiamata di sistema di cui si vuole usufruire, esso viene memorizzato nel registro "a7" del processo durante l'esecuzione del codice utente. Successivamente all'esecuzione dell'istruzione "ecall" si verifica una trap, la quale passa il controllo del codice utente al kernel e viene gestita in quest'ultimo tramite la funzione "syscall".

La funzione "syscall" opera in questo modo: innanzitutto recupera il numero della system call dal registro "a7" del trapframe, il quale è una struttura dati che contiene lo stato del processo interrotto, questo numero serve per poter accedere in maniera corretta alla tabella che associa questi ultimi alle rispettive funzioni di implementazione nel kernel. I parametri della system call, come ad esempio i file descriptor o gli indirizzi di memoria, sono passati attraverso i registri, sarà poi la funzione syscall responsabile di estrarre questi parametri dal trapframe del processo interrotto.

Per l'esecuzione di determinate syscall è ovviamente richiesto l'accesso alla memoria del processo utente, tuttavia, per garantire sicurezza ed integrità, xv6 utilizza le funzioni "copyin" e "copyout", le quali verificano attentamente che gli indirizzi di memoria forniti siano validi e che il processo utente non stia cercando di accedere ad una zona non sua, ovvero del kernel o allocata per altri processi, tutto ciò ha come scopo d'impedire al codice utente di manipolare e conseguentemente danneggiare aree di memoria a cui non dovrebbe avere accesso.

Al termine dell'esecuzione della system call e al completamento di tutte le operazioni necessarie, il kernel ripristina lo stato del processo, come ad esempio i registri, come il valore di ritorno della system call memorizzato nel registro "a0", dopodichè il controllo viene restituito al processo utente, che continua l'esecuzione dal punto in cui è stata chiamata la system call. Per riassumere, le system call in xv6 garantiscono l'integrità e l'isolamento del sistema operativo, grazie ad un rigoroso controllo dei parametri e all'accesso sicuro alla memoria utente tramite apposite funzioni, xv6 assicura un'interazione affidabile con il kernel, proteggendo il sistema da errori e accessi indesiderati.

Similitudini

1. Entrambi forniscono ai processi utenti un interfacciamento alle system call simile, basato sull'attribuire ad ogni specifica operazione di chiamata di sistema un numero identificativo, poi mappato all'interno della system call table, che contiene il rispettivo puntatore a funzione, inoltre in entrambe i parametri vengono mappati nei registri.
2. Entrambi usano il concetto di user e kernel mode, infatti quando viene effettuata una chiamata di sistema avviene una transizione dalla cosiddetta user mode a quella kernel, consentendo al sistema operativo di svolgere istruzioni privilegiate per conto del programma utente.

Differenze

1. Os161 ha 3 diversi gestori delle eccezioni che vengono chiamati rispettivamente per le syscall, per le eccezioni vere e proprie e per le interruzioni (interrupt), xv6 li gestisce diversamente a seconda che l'eccezione sia avvenuta lato kernel o lato user, nel primo caso panica, se avviene lato user il kernel si occupa di killare il processo che ha avuto un errore, a prescindere da esso (dall'errore).

Differenze

2. Os161 ha un controllo meno sofisticato per quanto riguarda l'accesso in memoria da parte di una chiamata di sistema, se in xv6 vengono usate funzioni apposite per garantire un accesso valido, in os161 qualora per esempio venisse chiamata una syscall read, il kernel accede direttamente allo spazio utente utilizzando il puntatore passato come argomento alla funzione; è il sistema operativo a garantire che questi indirizzi siano validi e appartengano allo spazio utente e più in particolare a quello assegnato al processo che ha fatto richiesta, tramite meccanismi di protezione delle pagine del sistema operativo, si tratta dunque di un approccio che favorisce una maggiore semplicità di implementazione a fronte della scelta di realizzare un sistema operativo scolastico, piuttosto ad uno più sicuro che gestisce esplicitamente gli accessi impropri.

Differenze

3. Altra piccola seppur non banale differenza consiste nei registri utilizzati per i valori di ritorno delle funzioni: in os161 esso viene restituito nel registro v0, mentre in xv6 la syscall registra tale valore nel campo a0 del trapframe di p, seguendo quindi la convenzione del linguaggio C su RISC-V. Inoltre, sempre convenzionalmente, le syscall ritornano zero e valori positivi per indicare una operazione avvenuta con successo, mentre valori negativi simboleggiano un'esecuzione che ha riscontrato un errore, in xv6 se il numero di una syscall è invalido viene ritornato -1, successivamente imposta la variabile globale "errno" con un codice di errore specifico, per descrivere cosa sia effettivamente accaduto.

Meccanismi di sincronizzazione

Ogni processo ha una propria **sezione critica**, ovvero un segmento di codice condiviso tra più processi. E’ necessario sincronizzare l’accesso dei vari processi a queste porzioni di codice per permettere una corretta condivisione dei dati.

3 requisiti base:

1. **Mutua esclusione**
2. **Progresso**
3. **Attesa limitata**

Soluzioni hardware

- **Barriere di memoria**: istruzione che forza qualsiasi modifica in memoria a essere propagata a tutti gli altri processori
- **Istruzioni hardware**: Test-and-Set oppure Compare-and-Swap, entrambe eseguite in modo atomico
- **Variabili atomiche**: basate sulle istruzioni hardware, forniscono operazioni atomiche (interrompibili) su tipi base

Queste soluzioni hardware sono però difficili da gestire per il programmatore dell'applicazione.

Soluzioni software

- **Mutex lock (spinlock)**: acquire, release. Soluzione che richiede busy waiting. No context switch quando un processo aspetta il lock.
- **Semaforo**: variabile a cui si accede con wait e signal. Coda di attesa per ogni semaforo.
- **Condition variables**: permettono sincronizzazione dei thread in base al valore dei dati.

Confronto os161-xv6

Os161 utilizza spinlocks, semafori, lock, condition variables, wait channel (che sono come condition variables che non usano semafori ma spinlock).

I lock sono una primitiva di sincronizzazione usata in os161; sono intesi per essere usati per forzare la mutua esclusione. Un lock è simile a un semaforo binario, ma con un vincolo aggiuntivo: il thread che rilascia un lock deve essere lo stesso thread che l'ha acquisito.

Anche xv6 usa molte tecniche per il controllo della concorrenza; una molto usata è il lock. Xv6 ha 2 tipi di lock: spinlock e sleeplock.

Spinlock

- xv6:

```
struct spinlock {
        uint locked;  //indice se il lock è acquisito (1) o libero (0)
        char *name; //nome del lock (per debug)
        struct cpu *cpu;     //cpu che ha acquisito il lock
        uint pcs[10];  //10 indirizzi di ritorno del chiamante, usato per tenere traccia delle
                // delle posizioni del codice che hanno eseguito il lock (per debug)
}
```

Spinlock

- Os161

  struct spinlock {

  volatile int spl; //indica se il lock è acquisito (non 0) o acquisito (0). spl significa //“software priority level” ed è usato per disabilitare/abilitare gli interrupt

  const char *name; //nome del lock (per debug)

  struct thread *thread; //thread che ha acquisito il lock

  }

Si può notare come xv6 tenga traccia della cpu che ha acquisito il lock, mentre os161 si riferisca al thread che ha acquisito il lock (la struttura thread conterrà la cpu alla quale il thread è associato).

Lock

Una grande differenza tra i 2 so è che os161 usa una struct lock per forzare la mutua esclusione, mentre in xv6 questa struct non è presente.

-**lt_name**: nome lock (per debug)
-**lt_owner**: thread che ha acquisito il lock
-**lt_wchan**: waiting channel associato al lock, usato per mettere in attesa i thread che cercano di acquisire un lock già acquisito
-**lt_waiting**: thread che sta aspettando di acquisire il lock
-**lt_is_acquired**: lock acquisito (true) o libero (false)
-**lt_is_ordered**: indica se il lock è ordinato o meno
-**lt_recursive**: indica se il lock è ricorsivo o meno

In os161 la struct lock è definita come segue:

```
struct lock{
  const char *lt_name;
  struct thread *lt_owner;
  struct wchan *lt_wchan;
  struct thread *lt_waiting;
  bool lt_is_acquired;
  bool lt_is_ordered;
  bool lt_recursive;
}
```

Sleeplock

Xv6 possiede la struct sleeplock, ovvero una variante di lock che consente ai processi di andare in attesa (sleep) quando cercano di acquisire uno sleeplock non libero. Questo è diverso dagli spinlock, che fanno busy waiting.

```
struct sleeplock {
  uint locked; //acquisito (1) o libero (0)
  struct spinlock lk; //protegge l'accesso
  struct proc *thread; //processo con il lock
  uint pid; //pid del processo che ha il lock
}
```

Semafori

A differenza di os161, xv6 non possiede una struct semaphore per la gestione dei semafori.

Infatti si basa principalmente su spinlock e sleeplock.

Waiting channel

- xv6:

  struct wchan{

  struct spinlock lock; //per proteggere il waiting channel

  struct thread *threads[NTHREAD]; //ogni elemento dell'array è un thread in //attesa sul waiting channel

  int waiting; //numero di thread in attesa sul waiting channel

  char *name; //nome del waiting channel (per debug)

  }

Waiting channel

- os161:

  struct wchan{

  const char *wc_name; //nome del waiting channel (per debug)

  struct spinlock wc_lock; //per proteggere l'accesso

  wc_threads; coda di thread in attesa sul waiting channel

  }

La differenza tra i 2 è che in xv6 i thread in attesa sono salvati in un array dimensionato a NTHREAD e la struct wchan tiene un campo per il numero di thread in attesa; in os161 invece, i thread in attesa sono salvati in una struttura dati apposita (threadqueue), tra i cui campi ci sarà anche un numero di thread in attesa

Memoria Virtuale e
Memory Management Unit (MMU)

Un programma è memorizzato su disco come file eseguibile binario. Per essere eseguito, deve essere caricato in memoria, diventando eleggibile per l'esecuzione sulla CPU, in questo caso avremo il passaggio da memoria fisica a memoria virtuale (appunto quella della CPU).

L'associazione degli indirizzi in un sistema operativo si riferisce al processo di assegnare un indirizzo di memoria a un programma o a un processo al momento dell'esecuzione.

Il mapping tra indirizzo fisico e virtuale viene effettuata da un dispositivo hardware chiamato Memory Management Unit (MMU).

Nella gestione della memoria si possono presentare diversi problemi tra cui, qualora si utilizzasse un’allocazione contigua della memoria, la frammentazione esterna, in questo scenario lo spazio per l’allocazione è disponibile, ma non è contiguo.

Una soluzione a ciò è il paging, si divide la memoria fisica in FRAMES della stessa dimensione (in potenza di due) e la memoria logica in PAGES, anche questi aventi stessa dimensione. Quando un processo deve essere eseguito le sue pagine vengono caricate in tutti i frame disponibili anche se non sono contigui.

L’MMU per fare la traduzione degli indirizzi si appoggia alla page table, un indice che mappa per ogni numero di pagina il numero di frame corrispondente.

Nell’implementazione del paging ci sono vari algoritmi da considerare:

- un algoritmo per l’allocazione dei frame (se abbiamo tanti processi dobbiamo capire come distribuire l’allocazione di ogni frame per i processi).
- un algoritmo per il page-replacement (nel caso in cui ci sia un nuovo processo non ci fosse spazio e quindi devo selezionare delle vittime da rimpiazzare).

Tra questi ultimi troviamo algoritmi come FIFO, LRU e altri, e possono essere di tipo locale o globale (rimpiazzare uno tra tutti i frame o solo tra quelli posseduti dal processo).

Mentre per quanto riguarda i primi l’allocazione può essere Equal o Proportional (in base alla dimensione del processo).

OS161 vs XV6

Per quanto riguarda l'argomento memoria virtuale, nonostante siano entrambi dei sistemi operativi didattici, os161 risulta avere una gestione di quest'ultima più realistica e più complessa, a differenza di xv6 che ha una gestione molto più basilare.

Paginazione:

os161 può anche utilizzare una paginazione a più livelli, quindi con Page Table di primo, secondo e terzo livello come livello finale. Quella di primo livello contiene i puntatori alle TLB di secondo livello, che a loro volta contengono quelli per quelle di terzo livello, se presenti, che conterranno gli indirizzi virtuali delle pagine fisiche. Questo permette di risparmiare memoria dato che le TLB sono allocate solo quando è necessario, ed è un approccio usato nei sistemi operativi più moderni.

Xv6 invece non permette tale implementazione e si ferma quindi al singolo livello, quindi a una gestione più semplice e intuitiva, che sicuramente non dà prestazioni migliori, ma è più facile da comprendere, e ciò di sicuro è un punto a favore per sistema educativo.

Algoritmi usati :

per quanto riguarda gli algoritmi di page replacement, anche qui os161 usa qualcosa di più complesso rispetto a xv6, infatti os161 usa LRU o LFU (Least Recently Used// Least Frequently Used), mentre xv6 FIFO o Second Chance, questo permette a os161 di avere una gestione della memoria più efficiente seppur con un aumento della complessità, causato dalla gestione di algoritmi più complessi.

Xv6 usa la equal allocation , mentre os161 usa la proportional allocation.

Algoritmi di Scheduling

Gli algoritmi di scheduling sono insiemi di regole o procedure utilizzati nei sistemi operativi per determinare l'ordine in cui i processi devono essere eseguiti sulla CPU. L'obiettivo principale di questi algoritmi è massimizzare l'efficienza e l'utilizzo delle risorse di sistema, riducendo il tempo di attesa dei processi e migliorando la capacità del sistema di rispondere in modo tempestivo alle richieste degli utenti.

Algoritmi più comuni

- **First Come First Serve (FCFS):**

  Quando un processo deve essere eseguito viene messo in coda, con questo algoritmo i processi vengono eseguiti seguendo l'ordine di arrivo.

- **Round Robin (RR):**

  Ad ogni processo viene assegnato uno slot temporale all'interno della CPU, così che possano essere eseguiti ciclicamente.

- **Priority Scheduling :**

  Ad ogni processo viene assegnata una priorità, e vengono eseguiti in base ad essa.

- **Multilevel Queue Scheduling:**

  Suddivide i processi in più code, ogni coda ha la sua priorità, e ognuna con il proprio algoritmo di scheduling.

- **Shortest Job First:**

  Per ogni processo viene calcolato il tempo di esecuzione e si eseguono in ordine crescente di tempo di esecuzione.

Os161 vs xv6

Nel caso degli algoritmi di scheduling, per os161 e xv6 troviamo lo stesso algoritmo di default, ovvero il Round Robin.

Essendo che entrambi sono dei sistemi operativi educativi, sono stati fatti in maniera tale da poter supportare l'implementazione di altri algoritmi di scheduling da parte degli studenti.

La vera e propria differenza sta nella flessibilità e nella disponibilità di risorse per sviluppare tali algoritmi, infatti essendo os161 più complesso rispetto a xv6, offre una maggiore varietà di risorse e quindi permette di implementare algoritmi più complessi che in xv6 non sarebbe possibile implementare.