Progettazione di Sistemi Operativi

OS/161

A cura di:
Ilaria Tessarollo
Valentina Costa
Davide Bertola

Indice

1. **Indice 2**
2. **Obiettivi 3**
   1. Assignment 1 3
   2. Assignment 4 3
3. **Fase iniziale 3**
   1. Riferimenti 3
   2. Setup Environment 3
   3. Script di supporto e modalità di debug 4
4. **Assignment 1 .... 4**
   1. Gestione degli interrupt: codice e implementazione 4
   2. Implementazione dei lock e delle Conditional Variables 5
   3. Programmi di test e output 5
5. **Assignment 2 .... 5**
   1. Memoria virtuale in OS161 5
   2. Supporto ed esecuzione dei programmi ELF 6
   3. Syscalls 7
   4. Syscall essenziali : write(), exit(), bsrk() 8
   5. Implementazione di una semplice Page Table 8
   6. Programma di test con malloc userspace 9
   7. Dimostrazione dell'uso della pagetable 9
   8. Conclusioni 11

Obiettivi

L'obiettivo di approfondire le conoscenze a livello di codice di un sistema operativo semplice ma funzionante si è concretizzato nello svolgimento di 2 assignment sulla falsa riga di quelli proposti agli studenti dell'università di waterloo:
http://www.student.cs.uwaterloo.ca/~cs350/F07/assignments/

Assignment 1

Implementazione delle primitive di sincronizzazione per quanto riguarda locks e conditional variables.

Assignment 4

Implementazione di un meccanismo di memoria virtuale paginata.

Fase iniziale

Riferimenti

OS/161 è un sistema operativo sviluppato dall'università di Harward a scopo puramente accademico. E' un sistema funzionante nelle sue componenti principali il cui codice non mira alle performance ma ad essere il più leggibile possibile. Questo sistema operativo viene utilizzato a scopo didattico da diverse università a livello mondiale. Per la nostra tesina abbiamo fatto riferimento al materiale e documentazione forniti dall'università di Waterloo

Versione navigabile del codice sorgente del kernel:
- http://www.student.cs.uwaterloo.ca/~cs350/common/os161-src-html/index.html

Setup dell'environment:
- http://www.student.cs.uwaterloo.ca/~cs350/common/Install161NonCS.html

Testo degli assignments ed eventuali hints:
- http://www.student.cs.uwaterloo.ca/~cs350/common/

Setup Environment

La fase iniziale del nostro lavoro è consistita nel setup dell'ambiente di sviluppo. Per fare ciò abbiamo seguito quasi alla lettera quanto riportato dalla guida dell'università di Waterloo con piccoli adattamenti dovuti alla distribuzione di linux da noi adottata Ubuntu 9.04.
L'accorgimento fondamentale è quello di installare la versione di gcc e g++ 3.4
Una volta installati compilatore, debugger, emulatore mips sys161, kernel os161 e modificato le variabili d'ambiente, siamo riusciti a compilare ed eseguire il kernel.

Script di supporto e modalità di debug

Il debug del kernel viene effettuato connettendo il debugger gdb alla virtual machine tramite un unix-socket con i seguenti comandi:

```
sys161 -w kernel #lancio emulatore in modalità debug
cs250-gdb kernel

dir ../os161-1.11/kern/compile/ASST0
target remote unix:.sockets/gdb
```

E' possibile specificare le ultime due direttive nel file .gdbinit in modo che vengano eseguiti automaticamente all'avvio di gdb.

Al fine di rendere più veloci le sessioni di compilazione e debug abbiamo generato degli script (cartella scripts/ allegata) per automatizzare il processo. Si noti che negli script la variabile target rappresenta il nome del file contenuto in "kern/conf/" contenente una configurazione del kernel corrispondente all'assignment desiderato, la quale viene applicata dal tool "config" presente nella stessa directory. E' stato quindi sufficiente cambiare la variabile "target" a seconda dell'assignment su cui si stava lavorando. Segnaliamo inoltre che l'ultilizzo di versioni più recenti dei tool gcc, gdb e sys161 porta a problemi di allineamento del codice in fase di debug non ancora risolti dagli sviluppatori.

Assignment 1

Il primo assignment ha riguardato le primitive di sincronizzazione. Tutti i prototipi delle primitive di sincronizzazione sono dichiarate nel file header "synch.h" ed implementati nel file "sync.c"
L'implementazione iniziale comprendeva solo le primitive dei semafori lasciando allo studente la gestione dei locks e delle conditional variables. Per implementare le primitive mancanti è stato sufficiente seguire più fedelmente possibile quanto richiesto dai commenti disponibili sopra gli header delle funzioni.

Gestione degli interrupt: codice e implementazione

La lettura dell'implementazione pre-esistente dei semafori ci ha permesso di capire come sia possibile fare in modo che le chiamate vengano eseguite "in modo atomico". Eseguire in modo atomico in questo caso significa che nessun interrupt può fermare la parte di codice in esecuzione. Per fare in modo di disabilitare gli interrupt si usa la funzione splhigh(), mentre per riabilitarli successivamente si usa splx(). Queste funzioni disabilitano gli interrupt utilizzando il flag globale degli interrupt presente nell'architettura mips ma ignorando completamente il sistema di mascheramento prioritario presente in hardware per semplicità. L'anatomia di un segmento di codice eseguito in modo esclusivo da un solo thread senza essere interrotto è la seguente.

```
int spl;
spl = splhigh();
/*
* Codice non interrompibile
*/
splx(spl);
```

Abbiamo usato questo meccanismo in tutte le primitive da noi implementate per ottenere l'esecuzione esclusiva ed atomica.

Implementazione dei lock e delle Conditional Variables

Il funzionamento dei lock prevedere che il lock stesso abbia una memoria dell'attuale thread che è detentore del lock. Navigando il codice abbiamo notato che esiste la variabile globale "*curthread*" che punta univocamente ad una struttura che rappresenta il thread in esecuzione in un certo momento. Abbiamo quindi aggiunto alla struttura del lock una variabile "*holder*" che memorizza il thread detentore del lock, oppure NULL quando il lock risulta libero. Per gestire lo sleep e il wakeup dei thread abbiamo utilizzato le funzioni implementate in ″thread.c": "thread_sleep", "thread_wakeup". Queste 2 funzioni permettono di mettere in attesa, oppure svegliare tutti i thread in attesa in base all'indirizzo su cui questi attendono.

Per l'implementazione della funzione "cv_signal" delle conditional variables abbiamo invece dovuto implementare una nuova funzione analoga alla "thread_wakeup" che però svegliasse un solo thread (a caso) tra quelli in attesa ad un determinato indirizzo. La funzione implementata si chiama "thread_wakeup_one" ed è collocata all'interno di "thread.c". Il suo prototipo è nel rispettivo "thread.h"

Programmi di test e output

Il kernel comprende 2 test built-in accessibili dal menu principale dopo l'avvio: sy2 e sy3. Lanciando questi test e seguendo l'esecuzione col debugger gdb è stato possibile risolvere velocemente i problemi fino a passare i test stessi.

I 4 file ("synch.c", "synch.h", "thread.c", "thread.h") sono allegati alla presente relazione nella cartella rispettando la struttura delle directory che hanno all'interno del progetto.

Assignment 2

Memoria virtuale in OS161

Il sistema operativo os161 nella versione distribuita agli studenti comprende una implementazione di default della memoria virtuale. Le strutture dati sono dichiarate nei file "vm.h" ed "addrspace.h". La struttura addresspace è la struttura che tiene traccia della memoria assegnata ad un particolare thread (la struttura thread ha un puntatore al proprio

address space). L'address space di default prevede 2 segmenti. Per ogni segmento viene memorizzato un indirizzo virtuale, uno fisico, e un numero di pagine che rappresenta la dimensione del segmento.

```
struct addrspace {
 vaddr_t as_vbase1;
 paddr_t as_pbase1;
 size_t as_npages1;
 vaddr_t as_vbase2;
 paddr_t as_pbase2;
 size_t as_npages2;
 paddr_t as_stackpbase;
};
```

Quando la funzione "load_elf" (in "loadelf.c") carica un programma utente viene generato un nuovo address space associato al thread utente, e in esso vengono copiati i due segmenti del programma ELF (segmento codice e segmento dati). Per inizializzare e utilizzare gli address space vengono utilizzate le funzioni implementate nel file "dumbvm.c"
Notiamo quindi che la memoria per ogni segmento viene allocata parimenti in modo contiguo sia per quanto riguarda gli indirizzi virtuali che quelli fisici. Anche lo stack è contiguo e ha una lunghezza di default di 12 pagine.

Nello stesso file "dumbvm.c" è implementata la funzione "vm_fault". Questa rappresenta l'handler dell'interrupt che viene lanciato quando si verifica un TLB miss. Il suo compito è quello di navigare le strutture dati appena descritte per determinalre l'indirizzo fisico corrispondente a quello virtuale che ha generato l'evento di TLB-miss, e quindi aggiornare la TLB di conseguenza in modo che la traduzione venga eseguita con successo.

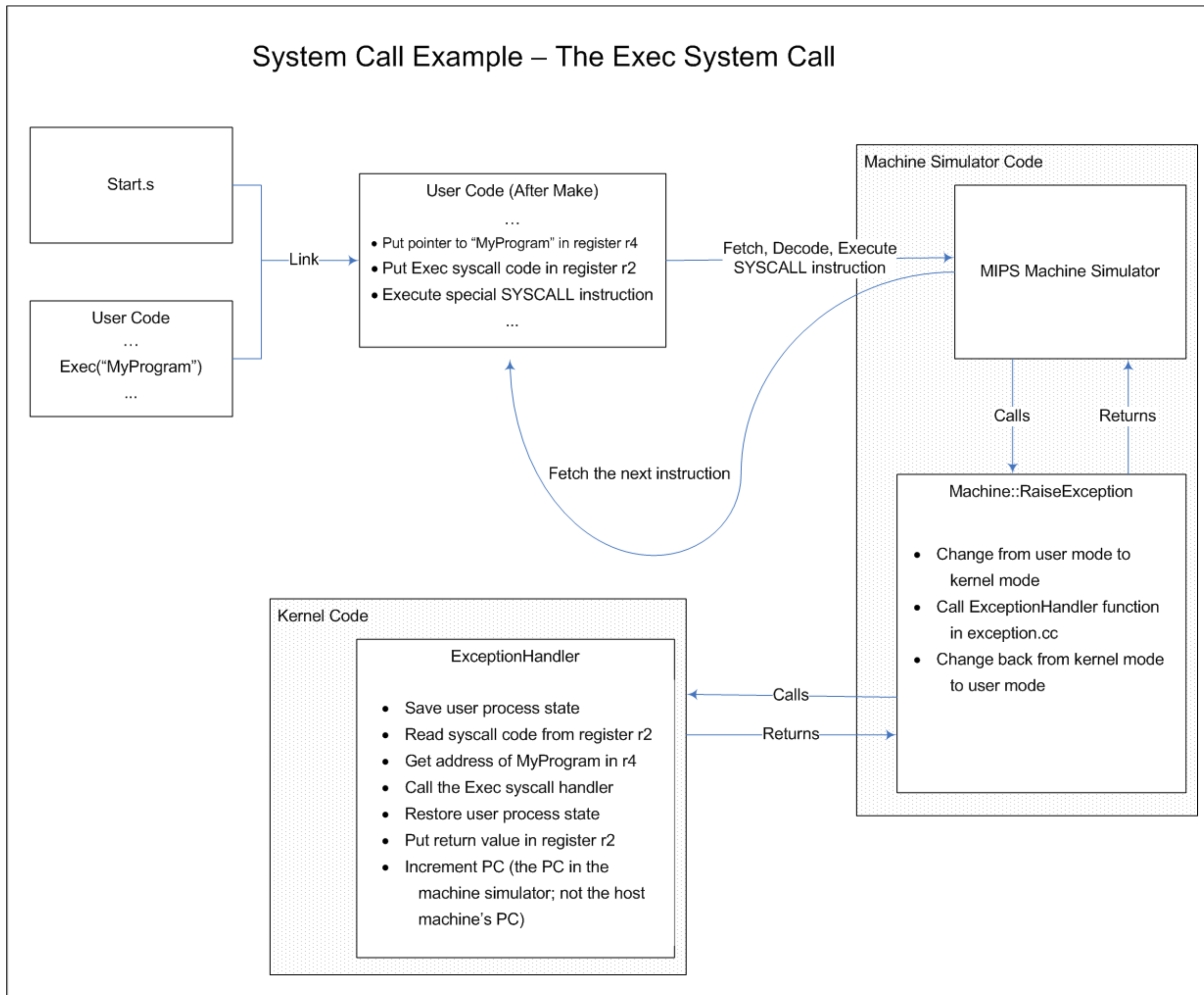
Configurando il kernel contro il target ASST3 il file "dumbvm.c" viene escluso dalla compilazione. Al suo posto viene incluso il file addresspace.c la cui scrittura viene lasciata all'utente. Il kernel così configurato non sarebbe quindi funzionante fino a quando non si raggiunga una implementazione corretta di "addresspace.c"

Supporto ed esecuzione dei programmi ELF

La memoria virtuale viene utilizzata solo ed esclusivamente per i programmi lanciati a livello utente e caricati tramite la funzione "load_elf". Per caricare un programma esterno è possibile usare il comando "p" dal menu principale del kernel. L'esecuzione di programmi a livello utente non è completamente supportata in quanto os161 non ha implementazione delle primitive fondamentali che permettono ai programmi di interagire col sistema. Per poter eseguire dei programmi di test abbiamo quindi dovuto capire il funzionamento e scrivere una semplice versione delle più importanti chimate di sistema (parte dell'assignment 2).

Syscalls

Le chiamate di sistema vengono viste a livello utente attraverso la standar library del C. Questa espone un prototipo al programma utente ma non dispone di una vera e propria implementazione del corpo delle funzioni. Al posto dell'implementazione viene usata una routine scritta in assembler responsabile di impostare l'hardware in modo da lanciare una TRAP (con un opportuno codice corrispondente alla syscall desiderata). Sarà successivamente l'handler a livello kernel che gestirà l'eccezione eseguendo l'opportuna routine corrispondente alla chiamata di sistema desiderata.



La parte di codice a livello kernel responsabile del dispatch di ogni interrupt al rispettivo handler nel caso delle syscall è la funzione "mips_syscall" nel file "syscall.c"
A questa funzione è stato necessario aggiungere gli handler successivamente da noi creati per ogni chiamata di sistema implementata. Notiamo che i parametri non vengono passati dalle funzioni in modo usuale, ma devono essere recuperati nella struttura trapframe a seguito dell'interrupt.

Syscall essenziali : write(), exit(), bsrk()

Le syscall di base mancanti che abbiamo dovuto implementare sono write, exit, e bsrk. La documentazione che specifica il prototipo e i requisiti per ogni syscall è disponibile al seguente indirizzo:

- http://www.cs.toronto.edu/~demke/369S.07/OS161_man/syscall/

Le prime due syscall sono state implementate in modo più superficiale senza tenere conto delle complicazioni che deriverebbero da un vero ambiente multithread. La loro implementazione ci è servita solo per far funzionare i nostri programmi di test uno alla volta pertanto non rispondono a tutti i requisiti specificati nella documentazione. Il loro codice è presente nel file "syscall.c"

Per quanto riguarda "bsrk" (chiamata sys_bsrk a livello kernel) la sua implementazione è completa. La sua funzione è quella di

- "allocare" N pagine di memoria fisica a seconda della dimensione richiesta dall'utente
- trovare uno spazio contiguo di N pagine nello spazio di indirizzamento virtuale
- inserire le corrispondenze tra indirizzo virtuale e fisico nella page table per ogni pagina allocata
- restituire l'indirizzo virtuale dell'inizio della memoria allocata

La complessità dell'algoritmo di ricerca di spazio virtuale contiguo è lineare.
Il codice è compreso nel file "addrspace.c"

Implementazione di una semplice Page Table

La pagetable da noi implementata è la più semplice possibile. Abbiamo fatto un array di dimensione pari al numero di pagine indirizzabili dalla memoria virtuale. La memoria virtuale di default è di 2Gb, mentre la dimensione di una pagina è di 4Kb. Tale array viene allocato in modo dinamico alla creazione di un nuovo address space. Abbiamo pertanto modificato la definizione di address space nel file "addrspace.h", e le funzioni di supporto alla gestione nel file "addrspace.c"
Sempre nello stesso file abbiamo inoltre reimplementato la funzione "vm_fault" in modo che eseguisse la traduzione da indirizzo virtuale a fisico utilizzando la nostra pagetable ed aggiornando la TLB con la traduzione corretta.

Quando viene inserita oppure ricercata una pagina di memoria fisica a partire da un indirizzo virtuale in pratica si indirizza l'array per displacement. La complessità dell'algoritmo sia in inserimento che ricerca è quindi O(1). La principale pecca di una page table indirizzata per indirizzo virtuale è l'overhead di memoria in quanto vengono allocate molte più entry di quelle che saranno usate dal programma (nel caso di memoria virtuale più grande della memoria fisica) e nel caso di multiprogrammazione viene allogata una page table per ogni processo. Usare una inverted page table sarebbe più efficiente sotto questo punto di vista.

Sottolineamo inoltre che non sono state cambiate le strutture pre esistenti perché le routine di caricamento degli eseguibili ELF sono troppo dipendenti da esse. Abbiamo quindi fatto in modo che la nostra implementazione di page table convivesse con la precedente struttura a segmenti contigui.

Notiamo infine che nella page table tutte le pagine vengono trattate allo stesso modo senza tenere memoria dell'appartenenza a un determinato segmento oppure allo stack, col vantaggio di una maggiore semplicità di gestione e il rilassamento del vincolo di contiguità.

Programma di test con malloc userspace

L'allocazione di memoria viene esposta a livello utente tramite la già discussa chiamata di sistema "bsrk". Questa è in grado di allocare memoria con una granularità pari alla dimensione di una pagina. Nella stragrande maggioranza dei casi la memoria dinamica a livello utente viene usata per allocazioni di dati molto più piccoli di una pagina. Per questo motivo a livello utente viene generalmente resa disponibile dalla libreria standard del C la funzione "malloc" (o varianti calloc, alloc, ..). Questa funzione è responsabile di richiedere al sistema un certo quantitativo di memoria detto "chunk", (tramite la syscall "bsrk") e di tenere traccia dell'utilizzo interno a questa memoria, tramite una struttura dati che indicizza le variabili allocate e il loro spazio occupato. La stessa struttura viene usata dalla funzione "free" per poter cancellare le variabili allocate.

Non essendo presente una implementazione di malloc nelle librerie fornite abbiamo deciso di implementarne una nello stesso codice del programma di test senza però gestire la complicazione di supportare un'eventuale chiamata free per liberare la memoria occupata. La struttura indice per il chunk è stata implementata per un massimo di 100 variabili memorizzando per ognuna di esse indirizzo di inizio e dimensione occupata all'interno del chunk.

```
struct alloc_entry {
   void *start;
   size_t size;
};
struct alloc_entry vartable[100];
```

Notiamo inoltre che, per come è scritto il programma, la memoria contenente l'indice stesso fa parte del segmento di codice, ed è quindi allocata dalle procedure di caricamento del file ELF.

Il programa di test è contenuto nella directory "testbin/memtest" e dispone di makefile correttamente scritto per poter essere compilato e installato nel kernel assieme agli altri esempi pre-esistenti (è sufficiente digitare "make; make install" dalla directory "testbin/").

Dimostrazione dell'uso della pagetable

Utilizzando il programma di test completo di funzione malloc abbiamo aggiunto delle "kprintf" all'interno delle funzioni "pt_insert" e "sys_bsrk". La prima è responsabile di inserire una corrispondenza nella page table tra memoria virtuale e memoria fisica, la seconda invece è la funzione che ritaglia uno spazio di heap richiesto dalla funzione malloc. Lo spazio allocato dalla malloc è fissato a 3 pagine (più che sufficienti per il programma di

test). Il programma di test eseguito per questa dimostrazione è il seguente

```
int
main(int argc, char *argv[])
{
  printf("--- User Program Loaded ---\n");
  char *p = malloc(4000*sizeof(char));
  strcpy(p,"Hello World 1");strcpy(p,"Hello World 1");
  printf("\"%s\" at %p\n", p, p);

  p = malloc(4000*sizeof(char));
  strcpy(p,"Hello World 2");
  printf( "\"%s\" at %p\n", p, p);
}
```

Vengono allocati 2 array di caratteri e copiate al loro interno due stringhe di testo. Per ogni array viene stampata a video la stringa contenuta e l'indirizzo di partenza. In questo modo possiamo verificare l'effettiva allocazione di memoria ed il suo accesso sia in scrittura che in lettura.

Eseguendo il programma otteniamo il seguente output.

```
OS/161 kernel: p testbin/memtest
pt_insert : virtual index 0x400000 => fisical index 0x22f000
pt_insert : virtual index 0x401000 => fisical index 0x230000
pt_insert : virtual index 0x10000000 => fisical index 0x231000
pt_insert : virtual index 0x80000000 => fisical index 0x232000
pt_insert : virtual index 0x7ffff000 => fisical index 0x233000
pt_insert : virtual index 0x7fffe000 => fisical index 0x234000
pt_insert : virtual index 0x7fffd000 => fisical index 0x235000
pt_insert : virtual index 0x7fffc000 => fisical index 0x236000
pt_insert : virtual index 0x7fffb000 => fisical index 0x237000
pt_insert : virtual index 0x7fffa000 => fisical index 0x238000
pt_insert : virtual index 0x7fff9000 => fisical index 0x239000
pt_insert : virtual index 0x7fff8000 => fisical index 0x23a000
pt_insert : virtual index 0x7fff7000 => fisical index 0x23b000
pt_insert : virtual index 0x7fff6000 => fisical index 0x23c000
pt_insert : virtual index 0x7fff5000 => fisical index 0x23d000
--- User Program Loaded ---
sys_brk : needed pages 3
pt_insert : virtual index 0x1000 => fisical index 0x23e000
pt_insert : virtual index 0x2000 => fisical index 0x23f000
pt_insert : virtual index 0x3000 => fisical index 0x240000
"Hello World 1" at 0x1000
"Hello World 2" at 0x1fa1
Operation took 0.929517240 seconds
OS/161 kernel [? for menu]:
```

Possiamo distinguere nel tracciato in ordine cronologico le seguenti fasi

- Al lancio il programma c'è una prima fase di caricamento dove vengono allocate una serie di pagine dalla funzione "pt_insert".
    - Inizialmente vengono allocate 4 pagine corrispondenti ai 2 segmenti contenuti nel file ELF nei quali viene copiato il codice e i dati del programma.
    - Successivamente dall'indirizzo virtuale 0x7ffff000 a 0x7fff5000 vengono allocate le pagine dello stack.
- Finito il caricamento inizia l'esecuzione del programma che stampa a video: "--- User Program Loaded ---"
- La funzione malloc richiede uno spazio di heap corrispondente a 3 pagine
- Le 3 pagine richieste vengono allocate ad indirizzi virtuali contigui
- Il programma crea e stampa i due array correttamente

Si noti che gli indirizzi sono espressi in esadecimale quindi la dimensione di una pagina che è 4096 byte diventa 0x1000. Il secondo array risulta quindi a cavallo tra la prima e la seconda pagina perché parte da 0x1fa1

Conclusioni

L'esperienza condotta si è dimostrata positiva. Siamo riusciti a comprendere il funzionamento e il codice di un sistema operativo funzionante e di implementarne alcune parti senza grossi problemi. Riteniamo inoltre il presente documento abbastanza argomentato da poter servire come base di conoscenza per futuri lavori da parte di altri studenti.