PdS 2020 - Laboratorio di OS161 - 2

Rendere eseguibile un programma utente in OS161

I programmi utente OS161, richiamabili mediante il comando "`p programma`", ad esempio "`p testbin/palin`", "`p sbin/reboot`", non sono eseguibili in modo corretto, in quanto, nella versione base di OS161, manca il supporto per

- le system call `read` e `write`,
- la system call `exit` (`_exit`),
- la gestione della memoria virtuale con restituzione della memoria allocata ai processi,
- gli argomenti al *main* (`argc, argv`).

Le system call `read` e `write`, pur se non strettamente necessarie per attivare un processo user, lo diventano nel caso in cui tale processo effettui I/O (succede in quasi tutti i programmi). I programmi di test proposti effettuano in genere output, raramente input, per cui risulta più importante il supporto per la `write`.

Al termine di un programma (fine del *main*) si chiama (implicitamente, quando non fatto in modo esplicito) la system call `exit`, il che provoca un crash del sistema, in quanto non esiste supporto per tale operazione.

Si chiede di realizzare il supporto per i punti precedenti (argomenti al *main* esclusi), ***in modo parziale***, tale da consentire l'esecuzione di un programma utente con semplice I/O e rilascio della memoria allocata per il processo. NON essendo richiesto il supporto per gli argomenti al main, ***non è detto quindi che tutti i programmi funzionino correttamente, in particolare quelli che prevedono argomenti al main***.

NOTE:

1) per capire quali programmi user siano disponibili, si vada (nel direttorio `os161/os161-base-2.0.2`) in `userland`: in `userland/testbin`, `userland/bin`, `userland/sbin` (e direttori sottostanti) si trovano i programmi user di test e altro. I programmi potrebbero essere (oltre che letti/capiti) modificati e ricompilati (con `bmake` e `bmake install`, che li rende disponibili all'esecuzione in `pds-os161/root/testbin` (e altri direttori simili). Siccome non si può usare il debugger all'interno di un programma user (`mips-harvard-os161-gdb` fa debug lato kernel), se si volesse fare debug di eventuali modifiche, oppure della versione esistente, si potrebbe compilare un programma (es. `testbin/palin/palin.c`) con `gcc` (per il sistema Ubuntu e il processore Intel/AMD host: es. `gcc -g -o palin palin.c`) e `gdb` (invece di usare `mips-harvard-os161-gdb`).

2) Si consideri la modalità di attivazione di un nuovo thread/processo da parte della `menu_execute()`, alla chiamata di `cmd_dispatch()`, per rispondere al comando "`p …`". Questa attiva un nuovo kernel thread mediante `thread_fork()`, nella funzione `common_prog()`. Il thread principale (il parent) esce SENZA attendere il termine del thread generato. Il nuovo thread esegue la funzione `cmd_progthread()` (che a sua volta chiama `runprogram()`). La funzione `common_prog()` crea inoltre un descrittore di processo, mediante `proc_create_runprogram()`. Nel frattempo il thread principale è già tornato al menu, in attesa di un nuovo comando in input.

System calls write e read

Si consiglia di realizzare due funzioni `sys_write()` e `sys_read()` (con parametri che ricalchino i prototipi di `read` e `write`). Le funzioni potrebbero, ad esempio, essere realizzare in un file `kern/syscall/file_syscalls.c`.

Si suggerisce di limitare la realizzazione al caso di IO su *stdout* e *stdin* (eventualmente *stderr*), evitando la gestione esplicita dei file (proposta in un successivo laboratorio) e ricorrendo invece (in quanto si tratta di file testo) alle funzioni `putch()` e `getch()` (si veda ad esempio il loro utilizzo in `kprintf()` e `kgets()`).

System call _exit

Il problema di os161 nella versione base, in relazione alla fine di un processo, è legato a due aspetti:

- il processo, che termina con `exit()`, non ancora realizzata, entra nella funzione `syscall()`, priva di supporto per `SYS__exit`, il che causa la chiamata a "*panic*";
- l'aggiunta di un "`case SYS__exit:`" in `syscall()` richiede un supporto minimale per l'azione di terminazione di un processo.

Il supporto per la `_exit()` (chiamata dalla `exit()` C, eventualmente in modo implicito al termine del `main`) si ottiene lavorando in modo simile alle system call `read()` e `write()`. La funzione C `exit` (file `userland/lib/libc/stdlib/exit.c`) riceve un unico parametro intero (il codice di errore/successo).

Il suo compito è quello di:

- mettere questo codice nel campo previsto (va aggiunto !) del descrittore di processo/thread (`struct thread`, si veda `kern/include/thread.h` e/o `struct proc` in `kern/include/proc.h`). Tale codice dovrà eventualmente essere letto da un altro thread.
- Far terminare il thread (e il processo) che chiama la `exit()`, chiamando la funzione `thread_exit()` (file `thread.c`), la quale pulisce la struttura dati del thread chiamante e lo manda in stato di *zombie*. Il descrittore del thread non viene distrutto (è quindi ancora leggibile da un altro thread). La distruzione vera e propria sarà effettuata dalla `thread_destroy()`, chiamata dalla `exorcise()` alla successiva `thread_switch()`.

Al momento attuale non si chiede di realizzare il salvataggio dello stato, quindi si consiglia di implementare una versione ridotta della `exit()`. Tale funzione deve essere in grado quanto meno di attivare la `as_destroy()` e la chiusura del thread: si suggerisce a tale scopo di realizzare una funzione `sys__exit(int status)` (creando ad esempio un file `kern/syscall/proc_syscalls.c` che la contenga). La funzione effettui quindi le chiamate ad `as_destroy()` e `thread_exit()`.

Il motivo principale di questa realizzazione parziale e provvisoria è quello di riuscire ad attivare e chiudere un processo utente senza errori distruttivi, facendo in modo di poter testare l'allocazione e (soprattutto) la de.allocazione della memoria user. Per "terminare" la system call exit serve un supporto per la sincronizzazione (syscall waitpid), per read e write serve un supporto più adeguato in termini di file system: entrambi gli argomenti saranno affrontati in successivi laboratori.

Gestione della memoria virtuale

Os161, nella versione base contiene un gestore di memoria virtuale che effettua unicamente allocazione contigua di memoria reale, senza mai rilasciarla (si veda il file `kern/arch/mips/vm/dumbvm.c`). Ad ogni chiamata a `getppages()` / `ram_stealmem()` viene allocato un nuovo intervallo nella RAM fisica, che non verrà più rilasciato.

Si chiede di realizzare un allocatore di memoria contigua, che, anziché utilizzare unicamente le funzioni `getppages()`/`ram_stealmem()` nella versione attuale, le modifichi, mantenendo traccia delle pagine (o frame) di RAM occupate e libere.

Si consiglia, in questa prima versione, di realizzare una "*bitmap*" o più semplicemente un vettore di flag (interi o char). Ciò richiede di modificare la ricerca di RAM libera, da parte di `as_prepare_load()` e `alloc_kpages()` (per inizializzare le struttura dati) e la restituzione di memoria da parte de `is_destroy()` e `free_kpages()`. Si consiglia di modificare la `getppages()` e/o la `ram_stealmem()` (per allocare memoria fisica contigua). ATTENZIONE: `getppages()` viene chiamata sia da `kmalloc()` -> `alloc_kpages()` (per allocazione dinamica al kernel) che da `as_prepare_load()` per allocare memoria ai processi user.

SOLUZIONE PROPOSTA

Si forniscono in allegato alcuni file (`dumbvm.c`, `syscall.c`, `file_syscalls.c`, `proc_syscalls.c`, `syscall.h`) che realizzano in parte il lavoro proposto. I file prevedono l'aggiunta dell'opzione "*syscalls*" in `conf.kern`, che rende opzionali i file `proc_syscalls.c` e `file_syscalls.c`.

Si tratta di realizzazioni parziali, per le quali sono possibili migliorie e/o alternative. **ATTENZIONE: Per un eventuale utilizzo occorre collocare i file nelle cartelle corrette (è necessario aver fatto il precedente laboratorio e aver capito come gestire le opzioni e i .h!), oltre a definire e gestire correttamente l'opzione *syscalls* in `conf.kern` *(*come spiegato in precedenza*).***

Per fare test/debug del programma sviluppato, si suggerisce di eseguire (con debug e breakpoint sulle funzioni di interesse) funzioni che allochino e deallochino memoria, ad esempio i test dei thread (*tt1*, *tt2*, *tt3*), per `kmalloc()` / `kfree()`, o i programmi user per generare e liberare l'address space di processi.

Si suggeriscono, in particolare, alcuni possibili lavori in relazione all'allocatore di memoria (***ATTENZIONE: le domande successive sono facoltative, non tutte di semplice soluzione; sta quindi allo studente valutare, in funzione della sua capacità e preparazione, se ed eventualmente cosa realizzare***):

1. Supporto completo per `kfree()`: la versione proposta non permette il rilascio della memoria allocata con `kfree()` prima che `vm_bootstrap()` attivi le tabelle di allocazione. Si studi una soluzione tale da ovviare a questo problema.

2. Realizzare varianti dello schema proposto, ad esempio ottenere tutta la RAM disponibile al bootstrap, per poi gestire sempre allocazione e deallocazione mediante le tabelle.

3. Utilizzo di una bitmap vera, anziché un vettore di char, per la tabella `freeRamFrames`. Una bitmap come vettore di bit viene solitamente realizzata come vettore di `unsigned int` o `unsigned long int`, e richiede l'utilizzo di opportuni operatori per accedere ai singoli bit (shift e o maschere con operatori di bitwise or/and). Una semplice spiegazione su come realizzare

vettori di bit in C si trova qui: http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html .

4. Realizzare una funzione che stampi statistiche sulla memoria allocata e libera. La funzione venga chiamata mediante un nuovo comando (*memstats*) aggiunto al menu di OS161.

5. Migliorare il tempo di ricerca di un intervallo libero e/o alleggerire i requisiti di mutua esclusione. La versione proposta ha un costo lineare di ricerca, in cui si lavora in mutua esclusione. Inoltre, gli accessi alla tabella sono effettuati in mutua esclusione. Si consideri la possibilità di evitare la mutua esclusione nella funzione `isTableActive()`. Si valuti l'opportunità di ridurre i requisiti di mutua esclusione per l'accesso alle tabelle.