PdS 2023 - Laboratorio 2

Rendere eseguibile un programma utente

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 calls read e write, exit e gli argomenti al main.
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:
Per capire quali programmi user siano disponibili, si vada (nel direttorio os161/os161-base-2.0.3) 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 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 (o altro): es. gcc –g –o palin palin.c) e gdb (invece di usare mips-harvard-os161-gdb).

System calls write, read e exit

Si consiglia di realizzare due funzioni sys_write() e sys_read() (con parametri che ricalchino i prototipi[1] 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()).
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().

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.