

PdS 2023 - Laboratorio 4

Realizzare la system call waitpid (attesa fine processo)

Si vuole realizzare il supporto per la system call waitpid (in Unix/Linux esiste anche la wait, che attende un qualunque processo child), che permette a un processo di attendere il cambio di stato di un altro processo, di cui sia noto l'identificatore (pid).

Per semplicità, si chiede di gestire unicamente il cambiamento di stato a processo "terminato" (sarebbe necessario gestirne altri, quali wait/resume connessi a un signal). In sintesi, dopo thread_exit (dell'ultimo/solo thread di un dato processo) un processo resta in stato "zombie" fintanto che un altro processo non fa una wait o waitpid (in OS161 solo waitpid), e ne ottiene quindi lo stato di uscita. Il laboratorio può essere suddiviso in parti, che si consiglia di realizzare un pezzo alla volta, passando al successivo dopo aver messo a punto (compresa esecuzione/debug) il precedente:

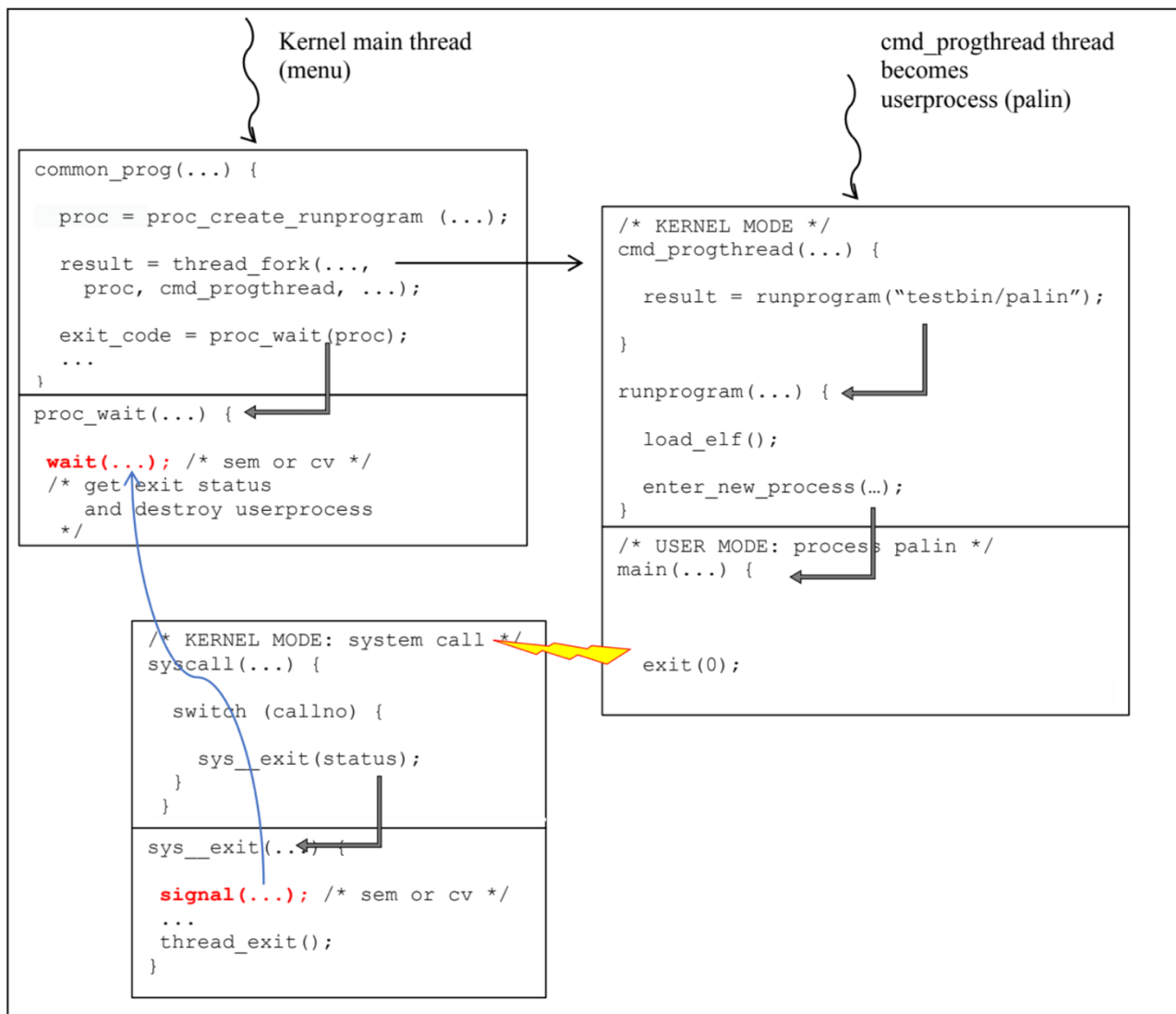
- Attesa della terminazione di un processo user, ritornandone lo stato di uscita
- Distruzione della struttura dati di un processo (struct proc)
- Assegnazione di pid a processo, gestione della tabella dei processi e waitpid
- (opzionale) realizzarele system call getpid e fork

Attesa di terminazione processo

Si consiglia di realizzare dapprima una funzione int proc_wait (struct proc *p), che gestisca (senza bisogno di gestire pid e di supportare la system call waitpid), mediante semaforo o condition variable (aggiunti come campo alla struct proc), l'attesa della fine (con chiamata alla syscall __exit()) di un altro processo, di cui si ha il puntatore alla relativa struct proc. Si tratta quindi di una funzione del kernel, utilizzabile solo all'interno di questo (perché sfrutta il puntatore a struct proc). Tale funzione potrebbe essere realizzata in kern/proc/proc.c e provata (chiamata) all'interno di common_prog, dopo che questa ha chiamato thread_fork con successo, per aspettare la fine del processo attivato (e non tornare immediatamente al menu che richiederebbe subito un altro comando). La common_prog potrebbe quindi aspettare la fine del processo child mediante:

```
exit_code = proc_wait(proc);
```

e stampare su console il codice di ritorno (quello ricevuto dalla sys__exit e salvato nella struct proc) prima di ritornare al programma chiamante. La figura rappresenta lo schema delle chiamate e della sincronizzazione. Si noti che le chiamate a wait() e signal() vanno sostituite da opportune funzioni, che dipendono dalle scelte implementative fatte (semaforo, condition variable, funzione wrapper o chiamata diretta).



Distruzione della struct proc

La struct `proc` di un processo non può essere distrutta (durante la `__exit`) finché un altro processo che chiami `wait/waitpid` non ne riceva la segnalazione (con lo stato di uscita). Tra le varie soluzioni possibili per liberare quindi una struct `proc` mediante la `proc_destroy`, si consiglia di chiamare quest'ultima all'interno della `proc_wait`, dopo l'attesa su semaforo o condition variable (cioè la struct non viene distrutta quando termina il processo ma all'interno di una `proc_wait`, fatta da un altro processo, eventualmente il kernel). Questo implica, probabilmente, una modifica alla realizzazione precedente della `sys__exit`, che ora non necessiterebbe più di rilasciare l'address space, ma unicamente di segnalare (su semaforo o condition variable) la fine del processo, prima di chiamare `thread_exit`. In altri termini, la `sys__exit` termina il thread, non distrugge la struttura dati del processo, ma ne segnala semplicemente la terminazione. La distruzione completa di un processo viene effettuata dalla `proc_wait` dopo che ne ha ricevuto la segnalazione di fine. La `proc_wait` gestisce inoltre il ritorno dello stato di uscita del processo.

ATTENZIONE: la funzione `proc_destroy()` richiede che il processo che si sta distruggendo (struttura dati) non abbia più thread attivi (si veda il codice della `proc_destroy`, che contiene l'asserzione `KASSERT (proc->p_numthreads== 0);`). Siccome la `sys__exit()` segnala la fine del processo prima di chiamare `thread_exit()`, è possibile che il kernel in attesa (nella `common_prog()`) venga svegliato e chiami la `proc_destroy()` prima che `thread_exit()` “stacchi” il thread dal processo (si veda il codice della `thread_exit()`, in particolare la chiamata a `proc_remthread()`). La soluzione per evitare questa corsa critica (race) non è univoca. Si suggerisce, come possibile opzione, di chiamare la `proc_remthread()` in modo esplicito nella `sys__exit()`, “prima” di segnalare la fine del processo, e modificare la `thread_exit()` in modo che accetti un thread già staccato dal processo (attenzione però a non OBBLIGARE la `thread_exit()` a vedere SEMPRE un thread “staccato”: la `thread_exit()` viene chiamata anche in altri contesti, fuori dalla `sys__exit()`).

Distruzione della struct proc

La `proc_wait` non realizza completamente il lavoro richiesto alla `waitpid`, in quanto parte da un puntatore a processo (invece che da un pid, un valore intero). Per l'attribuzione di un pid a un processo, occorre tener conto che si tratta di un intero unico (tipo `pid_t`), di valore compreso tra `PID_MIN` e `PID_MAX` (`kern/include/limits.h`), definiti in base a `__PID_MIN` e `__PID_MAX` (`kern/include/kern/limits.h`). Per l'attribuzione del pid e i passaggi da processo (puntatore a struct `proc`) a pid e viceversa, occorre realizzare una tabella. Per semplicità, si consiglia di realizzare un vettore di puntatori a struct `proc`, in cui l'indice corrisponda al pid (in tal caso si consiglia di utilizzare come pid massimo un numero accettabile, ad es. 100), oppure un vettore di coppie (pid, puntatore a processo). Un opportuno campo pid nella struct `proc` può invece consentire reperire il pid a partire dal puntatore. Ogni nuovo processo creato va aggiunto alla tabella, generandone il pid, ogni processo distrutto va rimosso dalla tabella (una volta completata una `wait/waitpid`). La tabella può, ad esempio, essere realizzata come variabile globale in `kern/proc/proc.c`. Occorre poi realizzare una eventuale funzione `sys_waitpid` da chiamare in `syscall()` (si consiglia di utilizzare lo stesso file, es. `proc_syscalls.c`, già usato per la `sys__exit`).

Tabella dei processi e waitpid

La `common_prog`, funzione interna al kernel, non ha bisogno di gestire il pid di un processo creato (ne ha già il puntatore), quindi non necessita, per attendere la fine del processo creato, del supporto per la `waitpid` (che richiede di gestire la tabella dei processi). In sostanza, la fine di un processo con `_exit` (e system call `sys__exit`) non necessita, se ad aspettare è il kernel che ne ha il puntatore, della `waitpid` (con processo identificato da pid), ma è sufficiente la `proc_wait` (processo identificato da puntatore). La `waitpid` invece è necessaria per gestione di processi da parte di programmi user. Ad esempio, `testbin/forktest` permetterebbe di verificare il funzionamento di `waitpid`. Tuttavia, per il funzionamento di tale programma di

test occorre realizzare la `getpid`, che ottiene il pid del processo corrente (puntato da `curproc`), e la `fork`, che permette di generare un processo child a livello user. Questa parte (realizzare `getpide fork`) può essere considerata opzionale (si consiglia di provare a realizzarla solo una volta completato con successo il resto del laboratorio): realizzare la `getpid` è semplice, la `fork` meno, in quanto si tratta di clonare (duplicare, l'intero address space di un processo (il child è una copia del padre) e di farlo partire correttamente). Un modo semplice per testare (senza bisogno della `fork`), non direttamente la `waitpid`, ma l'eventuale `sys_waitpid` chiamata in `syscall` per supportarla, è di ottenere in `common_prog` il pid del processo child (mediante `sys_getpid()` o altra strategia), e successivamente attendere con `sys_waitpid` anziché `proc_wait`. Si rappresenta, nella figura che segue, un possibile scenario per verificare la correttezza della catena `getpid/waitpide` gestione della tabella dei processi, non direttamente, ma indirettamente tramite le funzioni `sys_getpid()` e `sys_waitpid()`.

