

# Lab 4 — waitpid



## OBIETTIVI DEL LABORATORIO

- Implementazione della system call waitpid
- (opzionale) implementazione delle system calls getpid e fork

ATTENZIONE: per questo laboratorio è richiesto il completamento dei laboratori 2 e 3, aver compreso il flow di implemementazione di una system call (in particolare, SYS\_\_exit), semafori e lock.

## WAITPID

Si vuole realizzare il support per la system call waitpid 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 solo il cambiamento di stato a processo terminato (tralasciare perciò eventuali resume connessi a signals).

Dopo il tread\_exit (vedi SYS\_exit), il processo resta in stato “zombie” fino a che un altro processo non esegue una wait/waitpid (in OS161 solo waitpid) e ne ottiene lo stato di uscita.

Su sistemi unix ci sono delle systemcall che permettono di attendere l'attesa della fine di processi figli o comunque di certi processi.

Come riconosciamo se il processo finito è quello che aspettiamo noi?

Tramite il

*pid*.

## WAITPID

Il laboratorio può essere suddiviso in più parti. Si consiglia di verificare la correttezza di ogni singola parte (tramite debugger) prima di passare alla successive.

- Attesa della terminazione di un user process con ritorno di exit status
- Distruzione della struttura dati del processo
- Assegnazione di pid al processo
- (opzionale) realizzare getpid e fork

## TERMINAZIONE DI UN PROCESSO

Si consiglia di realizzare inizialmente una funzione kernel int proc\_wait (struct proc \*p) che gestisca, mediante semaforo (aggiunti come campo alla struttura proc), l'attesa della fine (con chiamata alla SYS\_exit) di un altro processo di cui si ha il puntatore alla relativa struttura. Guardare le note del laboratorio per maggiori dettagli.

## DISTRUZIONE DELLA STRUTTURA PROCESSO

La struct proc non può essere distrutta finché un altro processo chiama la waitpid e non riceva la segnalazione con status di uscita. Si consiglia di chiamare proc\_destroy nella proc\_wait dopo l'attesa su semaforo. Questo richiede anche la modifica di sys\_exit che non deve distruggere la struttura dati ma ne segnala semplicemente la terminazione.

## ASSEGNAZIONE DEL PID

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.

## TABELLA DEI PROCESSI E WAITPID

La fine di un processo con `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.

---

L'obiettivo del laboratorio è implementare la syscall `waitpid`, che fa in modo che il kernel aspetti che un processo utente termini la sua esecuzione — o in generale che un processo aspetti il termine dell'esecuzione di un altro processo.

Dobbiamo capire che `waitpid` è una funzione `signal/wait`, che invece di aspettare la signal da parte di un altro processo/thread, aspetta la terminazione di un processo, dato un certo `pid` (process id).

Ciò significa che dobbiamo in qualche modo *generare* degli id per i nostri processi, e memorizzarli da qualche parte, per tenerne traccia.

Dobbiamo quindi implementare una

***process table***.

Attualmente, quando un processo termina la sua esecuzione, esso viene posto in stato di zombie, ma la sua struttura dati è ancora allocata, e non viene rilasciata.

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) realizzare le system call `getpid` e `fork`

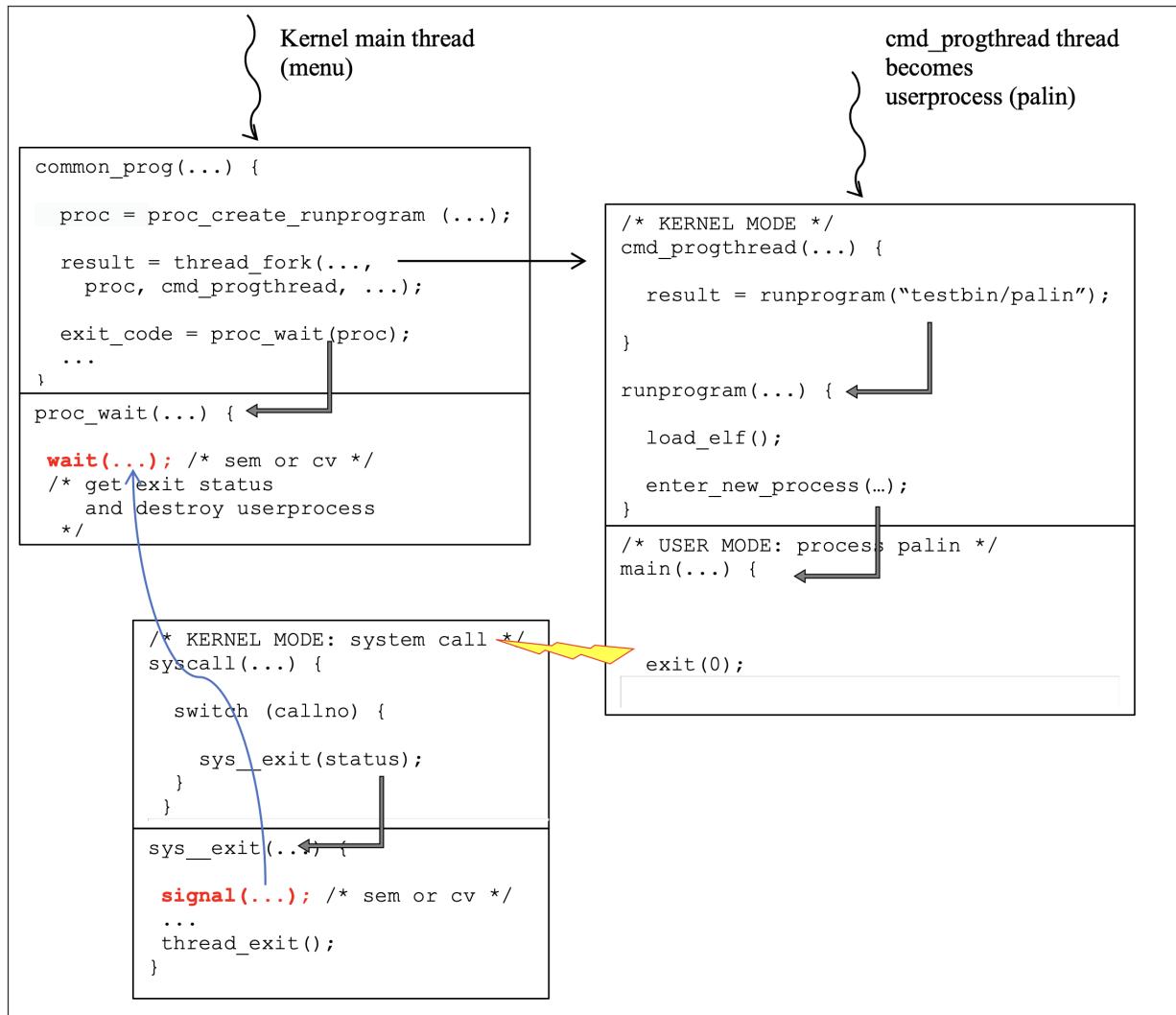
### Attesa di terminazione processo

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 `_exit - 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)



Vediamo la soluzione del prof.

## processTable

```

55  #define MAX_PROC 100
56  static struct _processTable {
57      int active;           /* initial value 0 */
58      struct proc *proc[MAX_PROC+1]; /* [0] not used. pids are >= 1 */
59      int last_i;            /* index of last allocated pid */
60      struct spinlock lk;   /* Lock for this table */
61  } processTable;

```

Notiamo l'implementazione della suddetta process table.

All'interno troviamo:

- flag per capire se impostarla ad attiva o meno
- array di 100 puntatori a struttura di processo (in realtà 101, ma 0 non si usa)
- indice dell'ultimo pid allocato
- spinlock per la table

Notiamo inoltre che `processTable` è una variabile globale, quindi sempre disponibile.

## proc\_search\_pid

```
69  /*
70  * G.Cabodi - 2019
71  * Initialize support for pid/waitpid.
72  */
73  struct proc *
74  proc_search_pid(pid_t pid) {
75  #if OPT_WAITPID
76      struct proc *p;
77      KASSERT(pid>=0&&pid<MAX_PROC);
78      p = processTable.proc[pid];
79      KASSERT(p->p_pid==pid);
80      return p;
81  #else
82      (void)pid;
83      return NULL;
84  #endif
85 }
```

Restituisce un puntatore a `proc`, dato un pid.

In questa soluzione pid è l'index della processTable, quindi torniamo semplicemente

`processTable.proc[pid]`.

Notiamo che lo facciamo  
senza mutual exclusion.

## proc\_init\_waitpid

```

89  * Initialize support for pid/waitpid.
90  */
91 static void
92 proc_init_waitpid(struct proc *proc, const char *name) {
93 #if OPT_WAITPID
94     /* search a free index in table using a circular strategy */
95     int i;
96     spinlock_acquire(&processTable.lk);
97     i = processTable.last_i+1;
98     proc->p_pid = 0;
99     if (i>MAX_PROC) i=1;
100    while (i!=processTable.last_i) {
101        if (processTable.proc[i] == NULL) {
102            processTable.proc[i] = proc;
103            processTable.last_i = i;
104            proc->p_pid = i;
105            break;
106        }
107        i++;
108        if (i>MAX_PROC) i=1;
109    }
110    spinlock_release(&processTable.lk);
111    if (proc->p_pid==0) {
112        panic("too many processes. proc table is full\n");
113    }
114    proc->p_status = 0;
115 #if USE_SEMAPHORE_FOR_WAITPID
116    proc->p_sem = sem_create(name, 0);
117 #else
118    proc->p_cv = cv_create(name);
119    proc->p_lock = lock_create(name);
120 #endif
121 #else
122     (void)proc;
123     (void)name;
124 #endif
125 }

```

La funzione riceve una struct proc, un nome e, in mutua esclusione (`spinlock_acquire(&processTable.lk)`), fa una serie di cose:

- legge il primo indice “disponibile” nell’array dei processi (ultimo inserito+1)
- entra in un while che cerca per una posizione libera nell’array, e appena la trova vi assegna il processo, aggiorna `last_i`, e assegna l’indice `i` come pid del processo.

Questo while fa “un giro” intero dell’array, fino a tornare all’indice `last_i`, nel caso in cui non trovasse prima una posizione libera nell’array

Dopo di che, se tutto è andato bene, dobbiamo gestire la sincronizzazione, dunque crea un semaforo (o una cv) sul processo.  
Questo è un modo in cui possiamo fondamentalmente stabilire un collegamento tra un processo e un id numerico.

## proc\_end\_waitpid

```
127  /*
128   * G.Cabodi - 2019
129   * Terminate support for pid/waitpid.
130   */
131 static void
132 proc_end_waitpid(struct proc *proc) {
133 #if OPT_WAITPID
134     /* remove the process from the table */
135     int i;
136     spinlock_acquire(&processTable.lk);
137     i = proc->p_pid;
138     KASSERT(i>0 && i<=MAX_PROC);
139     processTable.proc[i] = NULL;
140     spinlock_release(&processTable.lk);
141
142 #if USE_SEMAPHORE_FOR_WAITPID
143     sem_destroy(proc->p_sem);
144 #else
145     cv_destroy(proc->p_cv);
146     lock_destroy(proc->p_lock);
147 #endif
148 #else
149     (void)proc;
150 #endif
151 }
```

La funzione semplicemente elimina dall'array il processo ricevuto come parametro, e distrugge il semaforo (o la cv).

## proc\_create

```

153  /*
154   * Create a proc structure.
155   */
156 static
157 struct proc *
158 proc_create(const char *name)
159 {
160     struct proc *proc;
161
162     proc = kmalloc(sizeof(*proc));
163     if (proc == NULL) {
164         return NULL;
165     }
166     proc->p_name = kstrdup(name);
167     if (proc->p_name == NULL) {
168         kfree(proc);
169         return NULL;
170     }
171
172     proc->p_numthreads = 0;
173     spinlock_init(&proc->p_lock);
174
175     /* VM fields */
176     proc->p_addrspace = NULL;
177
178     /* VFS fields */
179     proc->p_cwd = NULL;
180
181     proc_init_waitpid(proc, name);
182
183     return proc;
184 }
```

Notiamo che nella funzione `proc_create`, già esistente, abbiamo ora inserito una chiamata a `proc_init_waitpid`.

## proc\_destroy

```

187 * Destroy a proc structure.
188 *
189 * Note: nothing currently calls this. Your wait/exit code will
190 * probably want to do so.
191 */
192 void
193 proc_destroy(struct proc *proc)
194 {
195     /*
196     * You probably want to destroy and null out much of the
197     * process (particularly the address space) at exit time if
198     * your wait/exit design calls for the process structure to
199     * hang around beyond process exit. Some wait/exit designs
200     * do, some don't.
201     */
202
203     KASSERT(proc != NULL);
204     KASSERT(proc != kproc);
205
206     /*
207     * We don't take p_lock in here because we must have the only
208     * reference to this structure. (Otherwise it would be
209     * incorrect to destroy it.)
210     */
211
212     /* VFS fields */
213     if (proc->p_cwd) {
214         VOP_DECREF(proc->p_cwd);
215         proc->p_cwd = NULL;
216     }
217
218     /* VM fields */
219     if (proc->p_addrspace) {
220         /*
221         * If p is the current process, remove it safely from
222         * p_addrspace before destroying it. This makes sure
223         * we don't try to activate the address space while
224         * it's being destroyed.
225         *
226         * Also explicitly deactivate, because setting the
227         * address space to NULL won't necessarily do that.
228         *
229         * (When the address space is NULL, it means the
230         * process is kernel-only; in that case it is normally
231         * ok if the MMU and MMU-related data structures
232         * still refer to the address space of the last
233         * process that had one. Then you save work if that
234         * process is the next one to run, which isn't
235         * uncommon. However, here we're going to destroy the

```

```

236     * address space, so we need to make sure that nothing
237     * in the VM system still refers to it.)
238     *
239     * The call to as_deactivate() must come after we
240     * clear the address space, or a timer interrupt might
241     * reactivate the old address space again behind our
242     * back.
243     *
244     * If p is not the current process, still remove it
245     * from p_addrspace before destroying it as a
246     * precaution. Note that if p is not the current
247     * process, in order to be here p must either have
248     * never run (e.g. cleaning up after fork failed) or
249     * have finished running and exited. It is quite
250     * incorrect to destroy the proc structure of some
251     * random other process while it's still running...
252     */
253     struct addrspace *as;
254
255     if (proc == curproc) {
256         as = proc_setas(NULL);
257         as_deactivate();
258     }
259     else {
260         as = proc->p_addrspace;
261         proc->p_addrspace = NULL;
262     }
263     as_destroy(as);
264 }
265
266 KASSERT(proc->p_numthreads == 0);
267 spinlock_cleanup(&proc->p_lock);
268
269 proc_end_waitpid(proc);
270
271 kfree(proc->p_name);
272 kfree(proc);
273 }
```

Analogamente, nella `proc_destroy`, abbiamo inserito una chiamata a

`proc_end_waitpid`.

## syscall.c

```

129     case SYS_exit:
130         /* TODO: just avoid crash */
131         sys_exit((int)tf->tf_a0);
132         break;
133     case SYS_waitpid:
134         retval = sys_waitpid((pid_t)tf->tf_a0,
135                               (userptr_t)tf->tf_a1,
136                               (int)tf->tf_a2);
137         if (retval<0) err = ENOSYS;
138     else err = 0;
139         break;
140     case SYS_getpid:
141         retval = sys_getpid();
142         if (retval<0) err = ENOSYS;
143     else err = 0;
144         break;
145
146 #if OPT_FORK
147     case SYS_fork:
148         err = sys_fork(tf,&retval);
149         break;
150#endif

```

Notiamo che abbiamo modificato il file `syscall.c`, aggiungendo dunque il supporto per nuove syscalls, in particolare:

- `SYS_waitpid`  
Prende in input un pid, un puntatore in user memory e un int
- `SYS_getpid`

Esaminiamo quindi tali syscalls.

## sys\_getpid

```

70 pid_t
71 sys_getpid(void)
72 {
73 #if OPT_WAITPID
74 | KASSERT(curproc != NULL);
75 | return curproc->p_pid;
76 #else
77 | return -1;
78 #endif
79 }

```

La funzione restituisce il pid del *current process*.

Infatti ricordiamo che quando creiamo un processo, tramite

`proc_create`, chiamiamo `proc_init_waitpid`, che al suo interno assegna un pid al processo creato.

Quindi è poi possibile leggerlo.

## sys\_waitpid

```

50 int
51 sys_waitpid(pid_t pid, userptr_t statusp, int options)
52 {
53 #if OPT_WAITPID
54 | struct proc *p = proc_search_pid(pid);
55 | int s;
56 | (void)options; /* not handled */
57 | if (p==NULL) return -1;
58 | s = proc_wait(p);
59 | if (statusp!=NULL)
60 | | *(int*)statusp = s;
61 | return pid;
62 #else
63 | (void)options; /* not handled */
64 | (void)pid;
65 | (void)statusp;
66 | return -1;
67 #endif
68 }

```

Premessa: **non gestiamo le options**.

La funzione recupera il processo tramite il pid ricevuto come parametro

(chiamando `proc_search_pid`) e chiama una funzione `proc_wait`.

Tale funzione semplicemente attende la terminazione di un processo, dato il puntatore a tale processo.

Dopo di che, recupera il codice di stato d'uscita del processo, ne distrugge la struttura (tramite `proc_destroy`) e ritorna il codice di stato.

Si noti come questa sia la prima volta che chiamiamo la funzione `proc_destroy`, mentre prima la struttura dati di un processo rimaneva allocata anche dopo la sua terminazione!

```
431 int
432 proc_wait(struct proc *proc)
433 {
434 #if OPT_WAITPID
435     int return_status;
436     /* NULL and kernel proc forbidden */
437     KASSERT(proc != NULL);
438     KASSERT(proc != kproc);
439
440     /* wait on semaphore or condition variable */
441 #if USE_SEMAPHORE_FOR_WAITPID
442     P(proc->p_sem);
443 #else
444     lock_acquire(proc->p_lock);
445     cv_wait(proc->p_cv);
446     lock_release(proc->p_lock);
447 #endif
448     return_status = proc->p_status;
449     proc_destroy(proc);
450     return return_status;
451 #else
452     /* this doesn't synchronize */
453     (void)proc;
454     return 0;
455 #endif
456 }
```

## Attenzione!

Vediamo come, dopo aver atteso la terminazione del processo, recuperiamo il codice di stato tramite `proc->p_status`, il quale viene impostato dalla `sys_exit`! Esaminiamo dunque la

`sys_exit`.

## sys\_exit

```

22  /*
23  * system calls for process management
24  */
25 void
26 sys_exit(int status)
27 {
28 #if OPT_WAITPID
29     struct proc *p = curproc;
30     p->p_status = status & 0xff; /* just lower 8 bits returned */
31     proc_remlist(proc);
32 #if USE_SEMAPHORE_FOR_WAITPID
33     V(p->p_sem);
34 #else
35     lock_acquire(p->p_lock);
36     cv_signal(p->p_cv);
37     lock_release(p->p_lock);
38 #endif
39 #else
40     /* get address space of current process and destroy */
41     struct addrspace *as = proc_getas();
42     as_destroy(as);
43 #endif
44     thread_exit();
45
46     panic("thread_exit returned (should not happen)\n");
47     (void) status; // TODO: status handling
48 }

```

Vediamo che adesso è più complessa.

La funzione riceve lo status del processo.

Tale status viene mascherato con

`0xff`, cioè ne prendiamo solo il byte meno significativo, e lo salviamo nella struttura dati del processo, nel campo `p->p_status`.

Dopo di ciò facciamo una signal per segnalare, a chi sta aspettando per la terminazione di questo processo, che lo stiamo terminando.

Dopo di che chiamiamo

`thread_exit` che si occuperà di:

- “staccare” il thread dal processo  
(tramite `proc_remlist`, che semplicemente decrementa il counter dei threads appartenenti al processo)

- chiamare `thread_switch`  
che a sua volta si occupa di  
**cambiare lo stato** del thread corrente (mettendolo a zombie), **distruggere lo stack kernel level** del thread e **la struttura dati del thread** stesso, per poi **fare uno switch** ad un altro thread.

### Nota!

Questa volta, al contrario di come facevamo prima di implementare la corretta terminazione di un processo, **non distruggiamo** l'address space del processo nella `sys_exit`.

Infatti, rimandiamo questa operazione nella

`proc_wait`, che si occupa poi di distruggere il processo, insieme al suo address space richiamando `proc_destroy`, la quale appunto finora non si era utilizzata, e possiamo vedere che all'interno richiama `as_destroy`.

Possiamo ora capire meglio la funzione `proc_wait`

```

431 int
432 proc_wait(struct proc *proc)
433 {
434 #if OPT_WAITPID
435     int return_status;
436     /* NULL and kernel proc forbidden */
437     KASSERT(proc != NULL);
438     KASSERT(proc != kproc);
439
440     /* wait on semaphore or condition variable */
441 #if USE_SEMAPHORE_FOR_WAITPID
442     P(proc->p_sem);
443 #else
444     lock_acquire(proc->p_lock);
445     cv_wait(proc->p_cv);
446     lock_release(proc->p_lock);
447 #endif
448     return_status = proc->p_status;
449     proc_destroy(proc);
450     return return_status;
451 #else
452     /* this doesn't synchronize */
453     (void)proc;
454     return 0;
455 #endif
456 }

```

Vediamo che quindi, una volta che è stata fatta la signal, usciremo dalla wait e proseguiremo nell'esecuzione, leggendo lo status del processo (precedentemente salvato nella struttura dati del processo dalla `sys_exit`) e distruggendo poi la struttura dati del processo.

Concludiamo restituendo lo status del processo terminato, al processo che ne attendeva la terminazione.

## Problema!

Notiamo che nella nuova `sys_exit`, prima di fare la signal, chiamiamo `proc_remthread`, ma tale funzione è anche chiamata all'interno della `thread_exit`, che viene anch'essa chiamata dopo.

Questo è un problema, perchè ci troveremmo nel caso in cui `proc->p_numthreads` sarebbe minore di 0 (avendo chiamato `proc_remthread` 2 volte), facendo panicare il kernel.

Inoltre in realtà, alla seconda chiamata di

`proc_remthread`, già la prima assert fallirebbe (`KASSERT(proc != NULL);`), in quanto la prima volta che si è chiamata avrebbe posto a NULL il riferimento al processo del thread.

Dobbiamo trovare un modo per

*togliere* quella chiamata a `proc_remthread` nella `sys_exit`, oppure toglierla dalla `thread_exit`.

Ma innanzitutto:

*perchè ci serve la chiamata a `proc_remthread` nella `sys_exit`?*

```
357  /*
358   * Remove a thread from its process. Either the thread or the process
359   * might or might not be current.
360   *
361   * Turn off interrupts on the local cpu while changing t_proc, in
362   * case it's current, to protect against the as_activate call in
363   * the timer interrupt context switch, and any other implicit uses
364   * of "curproc".
365   */
366 void
367 proc_remthread(struct thread *t)
368 {
369     struct proc *proc;
370     int spl;
371
372     proc = t->t_proc;
373     KASSERT(proc != NULL);
374
375     spinlock_acquire(&proc->p_lock);
376     KASSERT(proc->p_numthreads > 0);
377     proc->p_numthreads--;
378     spinlock_release(&proc->p_lock);
379
380     spl = splhigh();
381     t->t_proc = NULL;
382     splx(spl);
383 }
```

Supponiamo di levare la chiamata a `proc_remthread` nella `sys_exit`..

Dopo aver fatto la signal, quindi, chiameremo

`thread_exit`, la quale internamente farà la sua chiamata a `proc_remthread`.

Ma

**potrebbe** succedere che la `thread_exit` non sia abbastanza veloce da staccare il thread dal processo dopo aver fatto la signal.

Cosa succederebbe quindi?

```
431 int
432 proc_wait(struct proc *proc)
433 {
434 #if OPT_WAITPID
435     int return_status;
436     /* NULL and kernel proc forbidden */
437     KASSERT(proc != NULL);
438     KASSERT(proc != kproc);
439
440     /* wait on semaphore or condition variable */
441 #if USE_SEMAPHORE_FOR_WAITPID
442     P(proc->p_sem);
443 #else
444     lock_acquire(proc->p_lock);
445     cv_wait(proc->p_cv);
446     lock_release(proc->p_lock);
447 #endif
448     return_status = proc->p_status;
449     proc_destroy(proc);
450     return return_status;
451 #else
452     /* this doesn't synchronize */
453     (void)proc;
454     return 0;
455 #endif
456 }
```

Succederebbe che il processo risvegliato, in attesa della terminazione del processo che aspettava, proseguirebbe nell'esecuzione, chiamando `proc_destroy`.

```
193 proc_destroy(struct proc *proc)
263     as_destroy(as);
264 }
265
266 KASSERT(proc->p_numthreads == 0);
267 spinlock_cleanup(&proc->p_lock);
268
269 proc_end_waitpid(proc);
270
271 kfree(proc->p_name);
272 kfree(proc);
273 }
```

In `proc_destroy` ad un certo punto succede questo: un'assert su `p_numthreads == 0`.

Ma se la

`thread_exit` non è abbastanza veloce a rimuovere il thread dal processo *prima* che ciò accada, la `proc_destroy` tenterebbe comunque a distruggere il processo (che ha ancora un thread attaccato), l'assert fallirebbe e il kernel panicherebbe.

Per questo motivo si era inserita quella chiamata a `proc_remthread` *prima* della `thread_exit`.

## Soluzione!

```

778 void
779 thread_exit(void)
780 {
781     struct thread *cur;
782
783     cur = curthread;
784
785     /*
786      * Detach from our process. You might need to move this action
787      * around, depending on how your wait/exit works.
788      */
789     if (cur->t_proc != NULL) {
790         proc_remlist(cur);
791     }
792
793     /* Make sure we *are* detached (move this only if you're sure!) */
794     KASSERT(cur->t_proc == NULL);
795
796     /* Check the stack guard band. */
797     thread_checkstack(cur);
798
799     /* Interrupts off on this processor */
800     splhigh();
801     thread_switch(S_ZOMBIE, NULL, NULL);
802     panic("braaaaaaiiiiiiiinssssss\n");
803 }

```

Modifichiamo la `thread_exit`: se quando chiamiamo la `thread_exit`, il thread corrente ha ancora un riferimento al processo, allora chiamiamo `proc_remlist` e li disaccoppiamo.

Altrimenti procediamo assumendo che appunto il processo non ha più threads attaccati.

```

113 static
114 int
115 common_prog(int nargs, char **args)
116 {
117     struct proc *proc;
118     int result;
119
120     /* Create a process for the new program to run in. */
121     proc = proc_create_runprogram(args[0] /* name */);
122     if (proc == NULL) {
123         return ENOMEM;
124     }
125
126     result = thread_fork(args[0] /* thread name */,
127                           proc /* new process */,
128                           cmd_progthread /* thread function */,
129                           args /* thread arg */, nargs /* thread arg */);
130     if (result) {
131         kprintf("thread_fork failed: %s\n", strerror(result));
132         proc_destroy(proc);
133         return result;
134     }
135
136     int exit_code = proc_wait(proc);
137
138     return exit_code;
139
140     /*
141      * The new process will be destroyed when the program exits...
142      * once you write the code for handling that.
143      */
144
145     // return 0;
146 }

```

Inoltre modifichiamo la funzione `common_prog`, la quale si occupa di avviare un processo utente, aspettandone ora l'esecuzione e terminazione, leggendone il codice di stato e ritornandolo.

Possiamo ora vedere che torniamo al menu di OS161 solo **dopo** aver finito l'esecuzione di un programma utente, ad esempio `testbin/palin`

```

.....
Operation took 15.346716815 seconds
IS a palindrome
OS/161 kernel [? for menu]: 
```

## fork (EXTRA)

```
91 int sys_fork(struct trapframe *ctf, pid_t *retval) {
92     struct trapframe *tf_child;
93     struct proc *newp;
94     int result;
95
96     KASSERT(curproc != NULL);
97
98     newp = proc_create_runprogram(curproc->p_name);
99     if (newp == NULL) {
100         return ENOMEM;
101     }
102
103     /* done here as we need to duplicate the address space
104      | of the current process */
105     as_copy(curproc->p_addrspace, &(newp->p_addrspace));
106     if(newp->p_addrspace == NULL){
107         proc_destroy(newp);
108         return ENOMEM;
109     }
110
111     /* we need a copy of the parent's trapframe */
112     tf_child = kmalloc(sizeof(struct trapframe));
113     if(tf_child == NULL){
114         proc_destroy(newp);
115         return ENOMEM;
116     }
117     memcpy(tf_child, ctf, sizeof(struct trapframe));
118
119     /* TO BE DONE: linking parent/child, so that child terminated |
120      | on parent exit */
121
122     result = thread_fork(
123         curthread->t_name, newp,
124         call_enter_forked_process,
125         (void *)tf_child, (unsigned long)0/*unused*/);
126
127     if (result){
128         proc_destroy(newp);
129         kfree(tf_child);
130         return ENOMEM;
131     }
132
133     *retval = newp->p_pid;
134
135     return 0;
136 }
137 #endif
```

Vediamo la soluzione proposta per la syscall `sys_fork` del prof.

Questa syscall consente di generare un processo a partire da un altro processo.  
Essenzialmente crea un programma utente tramite

`proc_create_runprogram`, la quale a sua volta crea la struttura dati del processo tramite `proc_create` e la inizializza.

Poi, copia l'address space del processo padre, e lo assegna al nuovo processo. Copia anche la trapframe del processo padre, e la assegna al nuovo processo.

Poi esegue la

`thread_fork`, però al contrario di quanto accade in `common_prog` quando eseguiamo un nuovo programma, in cui, tramite `cmd_proghread` e poi `runprogram`, carichiamo un file eseguibile dato che creiamo un processo nuovo da zero, qui abbiamo clonato un altro processo e quindi la funzione che il nuovo thread esegue è

`call_enter_forked_process`.

```
82 static void
83 call_enter_forked_process(void *tfv, unsigned long dummy) {
84     struct trapframe *tf = (struct trapframe *)tfv;
85     (void)dummy;
86     enter_forked_process(tf);
87
88     panic("enter_forked_process returned (should not happen)\n");
89 }
```

La funzione semplicemente fa un cast a `(trapframe *)` (che era stato passato come `(void *)` nella `thread_fork`) e chiama `enter_forked_process` passandole il trapframe.

```

189  /*
190   * Enter user mode for a newly forked process.
191   *
192   * This function is provided as a reminder. You need to write
193   * both it and the code that calls it.
194   *
195   * Thus, you can trash it and do things another way if you prefer.
196   */
197 void
198 enter_forked_process(struct trapframe *tf)
199 [
200 #if OPT_FORK
201     // Duplicate frame so it's on stack
202     struct trapframe forkedTf = *tf; // copy trap frame onto kernel stack
203
204     forkedTf.tf_v0 = 0; // return value is 0
205     forkedTf.tf_a3 = 0; // return with success
206
207     forkedTf.tf_epc += 4; // return to next instruction
208
209     as_activate();
210
211     mips_usermode(&forkedTf);
212 #else
213     (void)tf;
214 #endif
215 ]

```

Tale funzione copia il trapframe nel kernel stack del nuovo thread, e poi avvia effettivamente il thread.

Si riportano le funzioni chiamate da `common_prog` per comparazione.

```
61  /*
62   * Function for a thread that runs an arbitrary userlevel program by
63   * name.
64   *
65   * Note: this cannot pass arguments to the program. You may wish to
66   * change it so it can, because that will make testing much easier
67   * in the future.
68   *
69   * It copies the program name because runprogram destroys the copy
70   * it gets by passing it to vfs_open().
71   */
72 static
73 void
74 cmd_progthread(void *ptr, unsigned long nargs)
75 {
76     char **args = ptr;
77     char progname[128];
78     int result;
79
80     KASSERT(nargs >= 1);
81
82     if (nargs > 2) {
83         kprintf("Warning: argument passing from menu not supported\n");
84     }
85
86     /* Hope we fit. */
87     KASSERT(strlen(args[0]) < sizeof(progname));
88
89     strcpy(progname, args[0]);
90
91     result = runprogram(progname);
92     if (result) {
93         kprintf("Running program %s failed: %s\n", args[0],
94                strerror(result));
95         return;
96     }
97
98     /* NOTREACHED: runprogram only returns on error. */
99 }
```

```

48  /*
49   * Load program "progname" and start running it in usermode.
50   * Does not return except on error.
51   *
52   * Calls vfs_open on progname and thus may destroy it.
53   */
54 int
55 runprogram(char *progname)
56 {
57     struct addrspace *as;
58     struct vnode *v;
59     vaddr_t entrypoint, stackptr;
60     int result;
61
62     /* Open the file. */
63     result = vfs_open(progname, O_RDONLY, 0, &v);
64     if (result) {
65         return result;
66     }
67
68     /* We should be a new process. */
69     KASSERT(proc_getas() == NULL);
70
71     /* Create a new address space. */
72     as = as_create();
73     if (as == NULL) {
74         vfs_close(v);
75         return ENOMEM;
76     }
77
78     /* Switch to it and activate it. */
79     proc_setas(as);
80     as_activate();
81
82     /* Load the executable. */
83     result = load_elf(v, &entrypoint);
84     if (result) {
85         /* p_addrspace will go away when curproc is destroyed */
86         vfs_close(v);
87         return result;
88     }
89
90     /* Done with the file now. */
91     vfs_close(v);
92
93     /* Define the user stack in the address space */
94     result = as_define_stack(as, &stackptr);
95     if (result) {
96         /* p_addrspace will go away when curproc is destroyed */
97         return result;
98     }
99
100    /* Warp to user mode. */
101    enter_new_process(0 /*argc*/,
102                      NULL /*userspace addr of argv*/,
103                      NULL /*userspace addr of environment*/,
104                      stackptr, entrypoint);
105
106    /* enter_new_process does not return. */
107    panic("enter_new_process returned\n");
108    return EINVAL;
109 }
```

```

406  /*
407   * enter_new_process: go to user mode after loading an executable.
408   *
409   * Performs the necessary initialization so that the user program will
410   * get the arguments supplied in argc/argv (note that argv must be a
411   * user-level address) and the environment pointer env (ditto), and
412   * begin executing at the specified entry point. The stack pointer is
413   * initialized from the stackptr argument. Note that passing argc/argv
414   * may use additional stack space on some other platforms (but not on
415   * mips).
416   *
417   * Unless you implement execve() that passes environments around, just
418   * pass NULL for the environment.
419   *
420   * Works by creating an ersatz trapframe.
421   */
422 void
423 enter_new_process(int argc, userptr_t argv, userptr_t env,
424 |   |   vaddr_t stack, vaddr_t entry)
425 {
426     struct trapframe tf;
427
428     bzero(&tf, sizeof(tf));
429
430     tf.tf_status = CST_IRQMASK | CST_IEp | CST_KUp;
431     tf.tf_epc = entry;
432     tf.tf_a0 = argc;
433     tf.tf_a1 = (vaddr_t)argv;
434     tf.tf_a2 = (vaddr_t)env;
435     tf.tf_sp = stack;
436
437     mips_usermode(&tf);
438 }
```