BS-Entwicklung mit Literate Programming

Foliensatz 10: `fork()` und `schedule()`

Hans-Georg Eßer
TH Nürnberg

v1.0, 16.12.2013

fork() und schedule()

In diesem Foliensatz:

- Prozess mit `fork()` "verdoppeln"
- Umschalten zwischen mehreren Prozessen mit `schedule()`

Zur Vorbereitung:

- Prozesszustände in Konstanten `TSTATE_*`
- und Zustandsnamen für die Prozesstabelle

Chunk: ⟨constants⟩ (1)

⟨*constants*⟩≡

```
// Thread states
#define TSTATE_READY     1   // process is ready
#define TSTATE_FORK      3   // fork() has not completed
#define TSTATE_EXIT      4   // process has called exit()
#define TSTATE_WAITFOR   5   // process has called waitpid()
#define TSTATE_ZOMBIE    6   // wait for parent to retrieve exit value
#define TSTATE_WAITKEY   7   // wait for key press event
#define TSTATE_WAITFLP   8   // wait for floppy
#define TSTATE_LOCKED    9   // wait for lock
#define TSTATE_STOPPED  10   // stopped by SIGSTOP signal
#define TSTATE_WAITHD   11   // wait for hard disk
```

Chunk: ⟨global variables⟩ (1)

- ... und die zugehörigen Namen für die Prozesstabelle:

⟨*global variables*⟩≡

```
char *state_names[12] = {
  "---",   "READY", "---",   "FORK", "EXIT",
  "WAIT4", "ZOMBY", "W_KEY",           // 0.. 7
  "W_FLP", "W_LCK", "STOPD", "W_IDE"   // 8..11
};
```

Chunk: ⟨function prototypes⟩ (1)

- Ziel: Implementation der Funktion `u_fork()`, die einen neuen Prozess als (fast) identische Kopie des aufrufenden Prozesses erzeugt
- `context_t *r`: kommt aus dem System Call Handler
- `create_thread`, `start_address`: für Threads (noch nicht impl.)

⟨*function prototypes*⟩≡

```
  int u_fork (context_t *r, boolean create_thread,↩
… void *start_address);
```

Chunk: ⟨macro definitions⟩ (1)

- Kopieren von phys. Speicherblöcken
- nutzt `PHYSICAL`-Makro und Mapping des phys. Speichers auf `0xD0000000`...

⟨*macro definitions*⟩≡

```
#define phys_memcpy(target, source, size) \
  (unsigned int)memcpy ( (void*)PHYSICAL(target), \
  (void*)PHYSICAL(source), size)
#define copy_frame(out, in) \
  phys_memcpy (out << 12, in << 12, PAGE_SIZE)
```

Chunk: ⟨function implementations⟩ (1)

⟨*function implementations*⟩≡

```
int u_fork (context_t *r, boolean create_thread, void *start_address) {
  TCB           *t_old, *t_new;       // pointers to old/new TCB
  int           old_tid, new_tid;     // thread IDs (old/new)
  int           i, j;                 // counters
  unsigned int  eip, esp, ebp;        // temp variables for register values
  addr_space_id old_as, new_as;       // old/new address spaces

  ⟨fork implementation⟩
}
```

Implementation von fork()

Nötige Schritte:

- neuen Adressraum erzeugen mit `create_new_address_space()`
- dazu neuen TCB registrieren mit `register_new_tcb()`
- TCB kopieren und richtige Werte im neuen eintragen
- Kernel Stack erzeugen (und alten kopieren)
- User-Mode-Speicher kopieren
- EIP sichern (damit neuer Prozess an richtiger Stelle startet)
- Fallunterscheidung Vater/Sohn
- Vater: Sohn in Warteschlange eintragen, erhält dessen PID zurück
- Sohn: erhält 0 zurück

Chunk: ⟨fork implementation⟩ (1)

⟨*fork implementation*⟩≡

```
  ⟨disable interrupts⟩
  LOCK (thread_list_lock);
  old_as = current_as;  old_tid = current_task;  int ppid = old_tid;

  if (!create_thread) {
    // regular fork: clone kernel part of PD; reserve user part of memory
    new_as = create_new_address_space (
      address_spaces[old_as].memend - address_spaces[old_as].memstart,
      address_spaces[old_as].stacksize );
  } else {
    new_as = old_as;                  // creating a new thread!
    address_spaces[old_as].refcount++;
  }                                          // TO DO: ERROR CHECK!

  new_tid = register_new_tcb (new_as);       // TO DO: ERROR CHECK!
  t_old = &thread_table[old_tid]; t_new = &thread_table[new_tid];
  *t_new = *t_old;                // copy the TCB
  ⟨fork: fill new TCB⟩
  if (!create_thread) {
    // copy the memory
    ⟨fork: create new kernel stack and copy the old one⟩
    ⟨fork: copy user mode memory⟩
  }

  UNLOCK (thread_list_lock);

  eip = get_eip ();               // get current EIP
  if (current_task == ppid) { ⟨fork: parent-only tasks⟩ }
  else                      { ⟨fork: child-only tasks⟩ }
```

Chunk: ⟨fork: fill new TCB⟩ (1)

⟨*fork: fill new TCB*⟩≡

```
  // get rid of the copying; some data were already setup in register_new_tcb()
  t_new->state      = TSTATE_FORK;
  t_new->tid        = new_tid;
  t_new->ppid       = old_tid;   // set parent process ID
  t_new->addr_space = new_as;

  // copy current registers to new thread, except EBX (= return value)
  t_new->regs       = *r;
  t_new->regs.ebx   = 0;         // in the child fork() returns 0

  // copy current ESP, EBP
  asm volatile("mov %%esp, %0" : "=r"(esp));  // get current ESP
  asm volatile("mov %%ebp, %0" : "=r"(ebp));  // get current EBP
  t_new->ebp  = ebp;
  t_new->esp0 = esp;
```

Chunk: ⟨fork: parent-only tasks⟩ (1)

Vater muss:

- EIP im neuen TCB eintragen
- Sohn in Warteschlange einreihen
- Rückgabe: PID des Sohn

⟨*fork: parent-only tasks*⟩≡

```
  t_new->eip  = eip;
  add_to_ready_queue (new_tid);
  ⟨enable interrupts⟩       // must be done in parent
  return new_tid;           // in parent, fork()
↩
… returns child's PID
```

Chunk: ⟨fork: child-only tasks⟩ (1)

Sohn muss:

- Rückgabe: 0

⟨*fork: child-only tasks*⟩≡

```
if (create_thread) {
  printf ("THREAD\n");
  // set correct start address
}

return 0;      // in child, fork() returns 0
```

Neuen Kernel-Stack erzeugen

- Variante der schon in `create_new_address_space()` gesehenen Aktionen
- brauchen neue Seitentabelle (für vier Seiten unterhalb `0xC0000000`)
- zwei Schleifen von 0 bis 3 (`KERNEL_STACK_PAGES = 4`): Frame reservieren, eintragen
- und alten Kernel Stack in neuen kopieren mit
  `phys_memcpy (mmu(new_as, ...), mmu(old_as, ...), PAGE_SIZE)`

Chunk: ⟨fork: create new kernel stack and copy the old one⟩ (1)

⟨*fork: create new kernel stack and copy the old one*⟩≡

```
// create new kernel stack and copy the old one
page_table* stackpgtable = (page_table*)request_new_page();
  // will be removed in destroy_address_space()
address_spaces[new_as].kstack_pt = (unsigned int)stackpgtable;
memset (stackpgtable, 0, sizeof(page_table));
page_directory *tmp_pd;
tmp_pd = address_spaces[new_as].pd;
KMAPD ( &tmp_pd->ptds[767], mmu (0, (uint)stackpgtable) );

uint framenos[KERNEL_STACK_PAGES];   // frame numbers of kernel stack pages

for (i = 0;  i < KERNEL_STACK_PAGES;  i++)
  framenos[i] = request_new_frame();
  //will be removed in destroy_address_space()

for (i=0; i<KERNEL_STACK_PAGES; i++)
  as_map_page_to_frame (new_as, 0xbffff - i, framenos[i]);

// copy each page separately: they need not be physically connected or in order
unsigned int base = 0xc0000000-KERNEL_STACK_SIZE;
for (i = 0;  i < KERNEL_STACK_PAGES;  i++)
  phys_memcpy ( mmu(new_as, base + i*PAGE_SIZE),
                mmu(old_as, base + i*PAGE_SIZE), PAGE_SIZE );
```

Chunk: ⟨function prototypes⟩ (2)

- Funktion `get_eip()` in Assembler-Datei definiert

⟨*function prototypes*⟩+≡

```
extern unsigned int get_eip ();
```

Chunk: ⟨start.asm⟩ (1)

- Trick: Beim Aufruf der Funktion liegt Rücksprungadresse oben auf dem Stack
- mit `pop eax` in Register EAX holen (und mit `push` zurück schreiben)
- EAX enthält Rückgabewert der Funktion (Aufrufkonvention)

⟨*start.asm*⟩≡
```
global get_eip
get_eip:
  pop eax
  push eax
  ret
```

Chunk: ⟨fork: copy user mode memory⟩ (1)

⟨*fork: copy user mode memory*⟩≡
```
    // clone first 3 GB (minus last directory entry) of address space
    page_directory *old_pd, *new_pd;
    page_table     *old_pt, *new_pt;
    old_pd = address_spaces[old_as].pd;
    new_pd = address_spaces[new_as].pd;

    for (i = 0;  i < 767;  i++) {          // only 0..766, not 767 (= kstack)
      if (old_pd->ptds[i].present) {
        // walk through the entries of the page table
        old_pt = (page_table*)PHYSICAL(old_pd->ptds[i].frame_addr << 12);
        new_pt = (page_table*)PHYSICAL(new_pd->ptds[i].frame_addr << 12);
        for (j = 0;  j < 1024;  j++)
          if (old_pt->pds[j].present)
            copy_frame ( new_pt->pds[j].frame_addr, old_pt->pds[j].frame_addr );
      };
    };
```

Chunk: ⟨syscall prototypes⟩ (1)

- `fork()` im User Mode über Syscall verfügbar

⟨*syscall prototypes*⟩≡
```
void syscall_fork (context_t *r);
```

Chunk: ⟨syscall functions⟩ (1)

⟨*syscall functions*⟩≡
```
  void syscall_fork (context_t *r) {
    r-
>ebx = (unsigned int) u_fork(r, false, NULL);   // false: no thread, starts at 0
    return;
  };
```

⟨*initialize syscalls*⟩≡
```
  install_syscall_handler (__NR_fork, syscall_fork);
```

Chunk: ⟨function prototypes⟩ (3)

- Aufruf des Schedulers aus Timer-Interrupt-Handler heraus
- `scheduler()` wählt nächsten Prozess aus und erledigt den Context Switch

⟨*function prototypes*⟩+≡
```
void scheduler (context_t *r, int source);
```

Chunk: ⟨enable scheduler⟩ (1)

- Scheduler läuft nur, wenn `scheduler_is_active` auf `true` steht

⟨*global variables*⟩+≡
```
int scheduler_is_active = false;
```

⟨*enable scheduler*⟩≡
```
scheduler_is_active = true;
```

⟨*disable scheduler*⟩≡
```
scheduler_is_active = false;
```

Chunk: ⟨global variables⟩ (2)

- Wir merken uns beim Umschalten in `t_old` und `t_new`, wo die TCBs vom alten und neuen Prozess liegen
- Warum globale Variablen? Die lokalen gehen beim Wechsel des Adressraums verloren (liegen auf dem Stack)

⟨*global variables*⟩+≡
```
TCB *t_old, *t_new;
```

Chunk: ⟨function implementations⟩ (2)

- Es folgt die Implementation des Schedulers...

⟨*function implementations*⟩+≡
```
void scheduler (context_t *r, int source) {
  debug_printf ("*");
  ⟨scheduler implementation⟩
}
```

Chunk: ⟨scheduler implementation⟩ (1)

- Zombie-Check *) - nicht in der Vorlesung
- (siehe Skript)
- `scheduler()` direkt beenden, wenn Scheduler deaktiviert ist oder es nur einen Prozess gibt

⟨*scheduler implementation*⟩≡
⟨*scheduler: check for zombies*⟩

```
  // check if we want to run the scheduler
  if (!scheduler_is_active)  return;
  if (!thread_table[2].used) return;   // are ther↩
…e already two threads?
```

*) Zombie: Prozess, der bereits beendet ist und dessen Vaterprozess noch nicht seinen Rückgabewert auslesen konnte

Chunk: ⟨scheduler implementation⟩ (2)

⟨scheduler implementation⟩+≡

```
t_old = &thread_table[current_task];
⟨scheduler: find next process and set t_new⟩
if (t_new != t_old) { ⟨scheduler: context switch⟩ }
⟨scheduler: check pending signals⟩ // see chapter on signals
⟨scheduler: free old kernel stacks⟩  // if there are any
return;
```

- Finde nächsten Prozess (Round Robin)
- Context Switch nur, wenn es was zu wechseln gibt (`t_new != t_old`)
- Hat der neue Prozess Signale erhalten? (Signale: nicht in der Vorlesung)
- alte Kernel-Stacks aufräumen (siehe letzte Vorlesung, `kstack_delete_list`)

Chunk: ⟨scheduler: find next process and set t_new⟩ (1)

⟨scheduler: find next process and set `t_new`*⟩≡*

```
int tid;
search:
if (source == SCHED_SRC_WAITFOR) {
  // we cannot use the ->next pointer!
  tid = thread_table[1].next;   // ignore idle process
} else {
  tid = t_old->next;
}
if (tid == 0)  // end of queue reached
  tid = thread_table[1].next;   // ignore idle process
if (tid == 0)  // still 0? run idle task
  tid = 1; // idle
t_new = &thread_table[tid];
if (t_new->addr_space == 0 || t_new->state != TSTATE_READY)
  goto search; // continue searching
// found it!
```

Context Switch

- Für den eigentlichen Context Switch definieren wir zunächst Makros, die den Zugriff auf ESP und EBP erleichtern
- Switch ist dann i. W.:
  - Sichern aller aktuellen Register im alten TCB
  - Umschalten auf neuen Adressraum
  - Rücksichern alle Register aus dem neuen TCB

Chunk: ⟨macro definitions⟩ (2)

⟨macro definitions⟩+≡

```
#define COPY_VAR_TO_ESP(x)  asm volatile ("mov %0, %%esp" :         : "r"(x) )
#define COPY_VAR_TO_EBP(x)  asm volatile ("mov %0, %%ebp" :         : "r"(x) )
#define COPY_ESP_TO_VAR(x)  asm volatile ("mov %%esp, %0" : "=r"(x)          )
#define COPY_EBP_TO_VAR(x)  asm volatile ("mov %%ebp, %0" : "=r"(x)          )
#define WRITE_CR3(x)        asm volatile ("mov %0, %%cr3" :         : "r"(x) )
```

Chunk: ⟨scheduler: context switch⟩ (1)

⟨scheduler: context switch⟩≡

```
t_old->regs = *r;                // store old:   registers
COPY_ESP_TO_VAR (t_old->esp0);   //              esp (kernel)
COPY_EBP_TO_VAR (t_old->ebp);    //              ebp

current_task = t_new->tid;
current_as   = t_new->addr_space;
current_pd   = address_spaces[t_new->addr_space].pd;
WRITE_CR3 ( mmu (0, (unsigned int)current_pd) );   // activate address space

COPY_VAR_TO_ESP (t_new->esp0);   // restore new: esp
COPY_VAR_TO_EBP (t_new->ebp);    //              ebp
*r = t_new->regs;                //              registers
```

Zusammenfassung (1)

- Booten: Multiboot-Header, Protected Mode
- Speicher: Segmentierung, Paging, Deskriptor-Tabellen
- Speicher-Init.: Trick-GDT, Page Directory + Page Tables, richtige GDT, Video-Speicher, Mapping des phys. Speichers
- Speicher-Verw.: Frame-Tabelle (Bitmap), `request_new_frame`, `release_frame`, `request_new_page(s)`, `release_page`
- Einträge in Page Dir./Table schnell füllen mit `KMAPD`, `KMAP`, `UMAPD`, `UMAP`
- Zugriff auf phys. Speicher: `PHYSICAL`
- MMU-Simulation mit `mmu`, `mmu_p`, `pageno_to_frameno` (für Address Space 0)

Zusammenfassung (2)

- Interrupts: IDT, generischer Assembler-Code für `irq0` ... `irq15`
- Aufruf von `irq_handler` aus den Assembler-Routinen
- `irq_handler` ruft (falls vorhanden) spezifischen Interrupt-Handler für IRQ auf; Beispiel: Tastatur
- Fault Handler: analog zu Interrupt-Handlern; `isr0` ... `isr31`
- System Calls: Interface User Mode → Kernel-Funktionen, über `int 0x80`
- Eintrag neuer Syscalls mit `install_syscall_handler`
- Beispiele: Syscalls für `printf`, `readline`
- braucht wieder Assembler-Code (ähnlich wie bei Interrupts und Faults)

Zusammenfassung (3)

- bei allen Handler-Typen immer Registerinhalte in `context_t *r` (bzw. `struct regs *r`)
- Syscall-Nummern `__NR_*`
- Adressräume: separate Page Directories (und Page Tables) für Prozesse, `create_new_address_space`
- TCB (Thread Control Block), `register_new_tcb`
- Speicher-Layout eines Prozesses (User Mode Memory, Heap, User Mode Stack, Kernel Mode Stack; Kernel Code/Data)
- Zuordnung Seite → Rahmen für Adressraum eintragen: `as_map_page_to_frame`
- Adressraum zerstören; `kstack_delete_list`

Zusammenfassung (4)

- Adressraum aktivieren: direkt im Scheduler, sonst Problem bei Umschaltung
- Prozessspeicher erweitern: `sbrk`
- TCB-Warteschlangen: ready queue, blocked queues (doppelt verkettete Listen)
- `thread_table[0]`: `next` als Anfang der ready queue
- Funktionen für Verwaltung der Queues
- wichtige globale Variablen: `current_task`, `current_as`, `current_pd`

Zusammenfassung (5)

- Erster Prozessstart: `start_program_from_disk` (in Übung: `start_program_from_ram`)
- GDT-Einträge für User Mode, TSS
- Umschalten in User Mode: `cpu_usermode`, Trick: Manipulation der Segmentregister, RPL 3 setzen
- Prozess verdoppeln mit `fork`
- Scheduler: Aufruf über Timer-Handler, wählt nächsten Prozess, macht Context Switch
- Context Switch: i. W. Register in TCB sichern und dann Register aus neuem TCB laden

Was fehlt?

- Timer (Interrupt-Handler) → heutige Übung
- Dateisysteme, Floppy/Festplatte → Lektüre (Dezember/Januar)
- Synchronisation (Mutexe) → Januar
- Prozesse: `exec`, `exit`, `waitpid` → Januar
- Mehr System Calls (und zugehörige Library-Funktionen für User-Mode-Programme) → Projekt (Januar/Februar)
- Netzwerk, grafische Oberfläche, Sound, SMP, ...
  → 2049 (oder Bachelor-Arbeiten)