Realisierung von Prozessen in Betriebssystemkernen

PROF. DR. STEFFEN WENDZEL

HS Worms, Vorlesung Betriebssysteme, SoSe 2020

Agenda

1. Einführung und Motivation der Thematik
2. Erzeugen und Beenden von Userland-Prozessen
3. Verwaltungsstrukturen von Prozessen
4. Umsetzung kernelseitig erzeugter Threads
5. Limitierungen des Prozessmodells
6. Zusammenfassung

Einführung

- **Tasks** sind (Teil-)Programme in Ausführung.
  - Vollständiges Programm in Ausführung: **Prozess**,
    parallel ablaufende Funktion im Prozess: **Thread**
  - Ohne Tasks benötigen wir keine Betriebssystem.
  - Daher: Taskrealisierung elementarer Bestandteil von BS

- Im Folgenden primär Verwendung des Begriffs „**Prozess**“.
  - Begriff „Thread“ nur bei expliziter Notwendigkeit.

- Moderne BS ermöglichen das parallele Ausführen von Prozessen – **Multitasking**.
  Ohne Multitasking nur jeweils 1 laufender Prozess!
  - BS: Verwaltung sämtlicher Prozesse inkl. zugehöriger Metadaten (Speicher,
    Befehlszähler etc.).

Einführung

- Keine zwei Prozesse sind „gleich“. Sie können zwar aus derselben Programmdatei hervorgegangen sein, doch …
  - können sie unterschiedliche Daten (Programmparameter/Variablenwerte, …) verwenden;
  - können sie aktuell einen unterschiedlichen Abschnitt des Programmcodes ausführen;
  - werden sie über eine unterschiedliche Prozess-ID identifiziert;
  - können sie mit unterschiedlichen Rechten und zu unterschiedlichen Zeitpunkten ablaufen;
  - verwenden sie einen (in Teilen*) eigenen Speicherbereich.

* Bei einer Prozesserstellung wird Speicher mit redundanten Inhalten i.d.R. nicht doppelt allokiert (sog. *Copy-on-Write*).

Einführung

- Dabei werden in den meisten Fällen nicht nur zwei, drei Prozesse verwaltet.

- Typisch: nach dem Start laufen bereits 10-50 Prozesse, nach einer Anmeldung und dem Start weniger Programme können es bereits deutlich über hundert Prozesse sein.

```
$ ps -e | wc -l
186
```

  - Daher: **Effiziente Verwaltung notwendig** (gutes Scheduling, effiziente Verwaltungs-strukturen, schnelle Prozesswechsel).

Einführung

- Prozessverwaltung kennt diverse Bestandteile (Scheduling, Speicherverwaltung, Verwaltung der Interprozesskommunikation, …).

- Wir betrachten daher nur selektierte Aspekte
  - Erzeugung und Beendigung von Prozessen
  - Prozessverwaltungsstrukturen in BS-Kernen
  - Erzeugung kernelseitiger Threads
  - Limitierungen des Prozessmodells

Erzeugen und Beenden von Userland-Prozessen

Die Prozesskreierung muss zunächst ausgelöst werden [1]:

1. Beim Initialisieren des Systems
2. Benutzeranfrage (Befehlseingabe)
3. Initiierung einer Stapelverarbeitung
4. Durch einen anderen Prozess (Systemaufruf)

Direkt oder indirekt wird Systemaufruf durchgeführt:

- Linux/BSD/Unix/ULIX: Erzeugung von Prozessen wird (bis auf Ursprungsprozess) über Systemaufrufe initiiert: `(v)fork()` und `exec*()`. Alternativ über Wrapper wie `system()`.
- Windows: `CreateProcess*()` bzw. Wrapper wie `ShellExecute()`
- RIOT OS: `shell_run()`: Shell erzeugt Prozess

Anm.: Alle obigen Betriebssysteme können mit versch. Systemaufrufen auch Threads erzeugen, etwa `thread_create()` unter RIOT OS!

Letztlich muss der Kernel den Prozess erzeugen!

Erzeugen und Beenden von Userland-Prozessen: ULIX

ULIX (literate-programming BS, Uni Erlangen, sämtliche Code-Abb. i. F. aus [2]):

⟨*kernel main* 44b⟩≡
```
void main () {
  ⟨initialize kernel global variables 184d⟩
  ⟨setup serial port 345a⟩  // for debugging
  ⟨setup memory 97⟩
  ⟨setup video 337c⟩
  ⟨setup keyboard 318e⟩
  ⟨initialize system 45b⟩
  ⟨initialize syscalls 173d⟩
  ⟨initialize filesystem 45c⟩
  ⟨initialize swap 293b⟩
  initialize_module ();  // external code
  ⟨start init process 45d⟩
}
```

⟨*start* init *process* 45d⟩≡ (44b) [45d]
```
⟨enable interrupts 47b⟩
printf ("Starting five shells on tty0..tty4. Press [Ctrl-L] for de/en keyboard.\n");
start_program_from_disk ("/init");   // load flat binary of init
// never reach this line!
```
Uses printf 601a and start_program_from_disk 189.

⟨*function implementations* 100b⟩+≡
```
void start_program_from_disk (char *progname) {
  ⟨start program from disk: prepare address space and TCB entry 190a⟩
  ⟨start program from disk: load binary 190c⟩
  ⟨start program from disk: create kernel stack 192a⟩
  ⟨start program from disk: set uid, gid, euid, egid 573b⟩
  ⟨start program from disk: activate the new process 192d⟩
};
```
Defines:
start_program_from_disk, used in chunk 45d.

Erzeugen und Beenden von Userland-Prozessen: ULIX

```
⟨function implementations 100b⟩+≡
  void start_program_from_disk (char *progname) {
    ⟨start program from disk: prepare address space and TCB entry 190a⟩
    ⟨start program from disk
    ⟨start program from disk
    ⟨start program from disk
    ⟨start program from disk
  };
Defines:
  start_program_from_disk, used in
```

[190a] ⟨start program from disk: prepare address space and TCB entry

```
// create new address space (64 KB + 4 KB stack) and reg
addr_space_id as   = create_new_address_space (64*1024, 4096);
thread_id     tid  = register_new_tcb (as);
TCB           *tcb = &thread_table[tid];

// fill TCB structure
tcb->tid = tcb->pid = tid;                    // identical thread/process ID
tcb->ppid = 0;                                // parent: 0 (none)
tcb->terminal = 0;                            // default terminal: 0
memcpy (tcb->cwd,     "/",   2);             // set current directory
memcpy (tcb->cmdline, "new", 4);             // set temporary command line
thread_table[tid].files[0] = DEV_STDIN;       // initialize standard I/O
thread_table[tid].files[1] = DEV_STDOUT;      // file descriptors
thread_table[tid].files[2] = DEV_STDERR;
for (int i = 3;  i < MAX_PFD;  i++) tcb->files[i] = -1;
activate_address_space (as);                  // activate the new address space
```

Sucht und allokiert freien Speicher für den Prozess. Markiert den Speicher als "AS_USED", setzt Startadresse, Endadresse, bereitet Stack vor, setzt Referenzcounter=1 („used by one process").

Durchsuche TCB (Thread Control Block)-Liste nach freiem Eintrag; falls gefunden: markiere als „used" und setze Zeiger für den Adress Space auf den zuvor reservierten.

Prozesse haben in ULIX bis zu 16 offene Files; da 0-2 bereits belegt sind, werden 3-15 inaktiv gesetzt.

Erzeugen und Beenden von Userland-Prozessen: ULIX

```
⟨function implementations 100b⟩+≡
  void start_program_from_disk (char *progname) {
    ⟨start program from disk: prepare address space and TCB entry 190a⟩
    ⟨start program from disk: load binary 190c⟩
    ⟨start program from disk: create kernel stack 192a⟩
    ⟨start program from disk: set uid, gid, euid, egid 573b⟩
    ⟨start program from disk: activate the new process 192
  };
Defines:
  start_program_from_disk, used in chunk 45d.
```

4 Pages (jeweils 4K) werden unter ULIX für jeden Prozess reserviert.

```
KERNEL_STACK_SIZE = KERNEL_STACK_PAGES x PAGE_SIZE
```

Register auf Ende von Stack initialisieren. Am Anfang ist ESP=EBP, später wächst ESP mit Stack.

```
⟨start program from disk: load binary 190c⟩≡
  int fd = u_open (progname, O_RDONLY, 0);
  u_read (fd, (char*)BINARY_LOAD_ADDRESS, PROGSIZE);
  u_close (fd);
```

```
⟨start program from disk: create kernel stack 192a⟩≡                (189) 192b ▷
unsigned int framenos[KERNEL_STACK_PAGES];        // frame numbers of kernel stack
for (int i = 0;  i < KERNEL_STACK_PAGES;  i++) {  // pages
  framenos[i] = request_new_frame ();
  as_map_page_to_frame (current_as, 0xbffff - i, framenos[i]);
}
```

After that we need to store the information about the process kernel stack into two TCB fields `esp0` and `ebp`.

```
⟨start program from disk: create kernel stack 192a⟩+≡                (189) ◁192a
char *kstack = (char*) (TOP_OF_KERNEL_MODE_STACK-KERNEL_STACK_SIZE);
memaddress adr = (memaddress)kstack;              // one page for kernel stack
```

```
tcb->esp0 = (uint)kstack + KERNEL_STACK_SIZE;    // initialize top-of-stack and
tcb->ebp  = (uint)kstack + KERNEL_STACK_SIZE;    // ebp (base pointer) values
```

Erzeugen und Beenden von Userland-Prozessen: ULIX

⟨*function implementations* 100b⟩+≡

```
void start_program_from_disk (char *progname) {
  ⟨start program from disk: prepare address space and TCB e
  ⟨start program from disk: load binary 190c⟩
  ⟨start program from disk: create kernel stack 192a⟩
  ⟨start program from disk: set uid, gid, euid, egid 573b⟩
  ⟨start program from disk: activate the new process 192d⟩
};
```

Defines:
`start_program_from_disk`, used in chunk 45d.

⟨*start program from disk: set uid, gid, euid, egid* 573b⟩≡

```
thread_table[tid].uid  = 0;  thread_table[tid].gid  = 0;
thread_table[tid].euid = 0;  thread_table[tid].egid = 0;
thread_table[tid].ruid = 0;  thread_table[tid].rgid = 0;
```

`ruid/rgid` (real user/grp. ID) dienen nur als Backup der ursprüngl. Werte bei Userwechsel zur Laufzeit.

Erzeugen und Beenden von Userland-Prozessen: ULIX

```
⟨function implementations 100b⟩+≡
  void start_program_from_disk (char *progname) {
    ⟨start program from disk: prepare address space and TCB entry 190a⟩
    ⟨start program from disk: load binary 190c⟩
    ⟨start program from disk: create kernel stack 192a⟩
    ⟨start program from disk: set uid, gid, euid, egid 573b⟩
    ⟨start program from disk: activate the new process 192d⟩
  };
Defines:
  start_program_from_di
```

```
⟨start program from disk: activate the new process 192d⟩≡
  current_task = tid;                    // make this the current task
  add_to_ready_queue (tid);              // add process to ready queue
  ⟨enable scheduler 276a⟩
  cpu_usermode (BINARY_LOAD_ADDRESS,
                TOP_OF_USER_MODE_STACK); // jump to user mode
```

Task wird an den Anfang einer doppelt verketten Liste rechenbereiter Tasks gesetzt

`scheduler_is_active = true;` (weil dies der erste Prozess ist)

Wechsel von Kernelmode (Ring 0) zu Usermode (Ring 3); danach Ausführung des Prozesscodes

Beenden von Userland-Prozessen

Die Prozessbeendigung muss zunächst ausgelöst werden [1]:

- Unfreiwillig durch schweren Fehler (etwa Division durch Null) → Kernel beendet Prozess
- Unfreiwillig durch anderen Prozess (bspw. `kill`-Signal (`SIGKILL`) unter Linux oder `TerminateProcess()` unter Windows)
- Freiwillig durch direkten Systemaufruf wie `exit()` (Linux) oder `ExitProcess()` (Windows) bzw. indirekt durch `return` der `main`-Funktion)

Beenden von Userland-Prozessen: ULIX

Szenario 1: exit()-Syscall

```
void syscall_exit (context_t *r) {
  // exit code is in ebx register:
  ⟨begin critical section in kernel 380a⟩  // access the thread table
  // close open files
  thread_id pid = thread_table[current_task].pid;
  int gfd;
  for (int pfd = 0;  pfd < MAX_PFD;  pfd++) {
    if ((gfd = thread_table[pid].files[pfd]) != -1)  u_close (gfd);
  }

  // modify thread table
  thread_table[current_task].exitcode = r->ebx;   // store exit code
  thread_table[current_task].state = TSTATE_EXIT; // mark process as finished
  remove_from_ready_queue (current_task);         // remove it from ready queue
  wake_waiting_parent_process (current_task);     // wake parent
  destroy_address_space (current_as);             // return the memory
  ⟨remove childrens link to parent 217b⟩          // notify children

  // finally: call scheduler to pick a different task
  ⟨end critical section in kernel 380b⟩
  scheduler (r, SCHED_SRC_RESIGN);
};
```

Tut nichts, nur Placeholder, da ULIX-Kernelmode Interrupts deaktiviert

Alle offenen Dateideskriptoren schließen

⟨*remove childrens link to parent* 217b⟩≡

```
for (int pid = 0;  pid < MAX_THREADS;  pid++)
  if (thread_table[pid].ppid == current_task)
    thread_table[pid].ppid = 1;  // set parent to idle process
```

Scheduler aktivieren, um nächsten Prozess Rechenzeit zu geben

Beenden von Userland-Prozessen: ULIX

Szenario 2: SIGKILL

- The SIGKILL$_{562a}$ signal:

⟨u_kill: *special cases* 563a⟩+≡

```
case SIGKILL:
```

⟨u_kill: *remove thread from queue* 564c⟩

```
tcb->used = false;
tcb->state = TSTATE_EXIT;
```

⟨u_kill: *write kill message* 564b⟩

```
wake_waiting_parent_process (pid);
```

⟨*enable scheduler* 276a⟩

```
if (pid == current_task) {
  ⟨resign 221d⟩  // enter scheduler
}
return 0;
```

Task aus aktueller Warteschlange entfernen (bspw. Ready- oder Harddisk-Blocked-Queue)

Thread Control Block aufräumen

Falls Parent mit `waitpid()` auf das Beenden des Childs wartet: Parent aus WaitPID-Queue in Ready-Queue bringen. Prozesse in der WaitPID-Queue werden vom Scheduler nicht aufgerufen, da wartend.

Scheduler aktivieren (falls er ausgesetzt wurde, bspw. durch einen Interrupt). Falls der Prozess (`pid`) sich selbst SIGKILL gesendet hat: Rechenzeit wieder abgeben, sodass anderer Prozess bearbeitet werden kann.

Verwaltung von Prozessen

- Warum wichtig? Was muss verwaltet werden?
  - Erzeugung und Beendigung kennen wir ja bereits
  - Speicher/Stack, Fehler (Division durch Null etc.), Interrupts, Berechtigungen, …
- Notwendig ist hierzu letztlich die Verwaltung der Kernel-internen Datenstrukturen.

Beispiel: ULIX Thread Control Blocks

```
typedef struct {
      thread_id  pid;          // process id
      thread_id  tid;          // thread id
      thread_id  ppid;         // parent process
      int        state;        // state of the process
      context_t  regs;         // enthält Usermode-Registerwerte (x86)
      memaddress esp0;         // kernel stack pointer
      memaddress eip;          // program counter
      memaddress ebp;          // kernel stack base pointer
      addr_space_id addr_space;//Index für Address Space Table
      thread_id  next;         // id of the ``next'' thread
      thread_id  prev;         // id of the ``previous'' thread
      boolean    used;         // setzt bspw. SIGKILL auf 'false'
      int        error;        // für Syscall-Fehler (z.B. EINVAL)
      int        exitcode;
      int        waitfor;      // pid of child this process waits for
      ...
```

```
      ...
      char       cmdline[CMDLINE_LENGTH];
      boolean    new;              // is this thread new?
      // extra kernel stack for this thread
      void       *top_of_thread_kstack; // falls Initialgröße zu klein
      int        terminal;
      int        files[MAX_PFD];
      char       cwd[256];
      sighandler_t sighandlers[32];
      unsigned long sig_pending;   // 32 Bits, um zu behandelnde und …
      unsigned long sig_blocked;   // … geblockte Signale zu markieren
      word       uid;              // user ID
      word       gid;              // group ID
      word       euid;             // effective user ID
      word       egid;             // effective group ID
      word       ruid;             // real user ID
      word       rgid;             // real group ID
} TCB;
```

Kernelseitige Threads

- Tasks (Prozesse bzw. Threads) existieren nicht nur im Userspace. Auch der Kernel kann von einer internen Parallelisierung profitieren → **Kernel-Threads**.
- Einsatz bspw. für Handling von Interrupts und für die Überwachung von USB-Hotplugging.
- Beispiel Linux:
  - Repräsentiert durch Task-Struktur (wie Userland-Tasks)
  - Scheduler ist auch für Scheduling der Kernel-Threads zuständig
  - `kernel_thread()` erzeugt neuen Kernel-Thread, im Hintergrund läuft aber klassisches `clone()` ab, das auch Userland-Threads erzeugt (nur mit Übergabe des Flags `CLONE_KERNEL`).
  - Kein eigener Adressraum (Kernelspace für alle Kernel Threads zugänglich); Zeiger auf Adressbereich == NULL
  - Kein Context-Switch in Userspace notwendig
  - Synchronisierung notwendig (Locks/Semaphore)

Limitierungen des Prozessmodells

- Entscheidungen bei Implementierung und Limits der Hardware bedingen verschiedene Grenzen für Prozesse.
- Beispielsweise die maximale Anzahl …
  - parallel ablaufender Tasks/Threads,
  - offener Dateien pro Prozess,
  - Verfügbarer Signale,
  - Anzahl an Locks pro Prozess oder für das gesamte Betriebssystem,
  - Größe des Adressraums für Prozesse,
  - …

Beispiel: ULIX-Prozesslimits

Änderungen einfach möglich (bspw. Heraufsetzen von `MAX_LOCKS`).
Rückwärtskompatibilität ggf. problematisch (bspw. Verlängerung `cwd`)!

- Maximale Anzahl parallel ablaufender Prozesse (`MAX_THREADS`) liegt bei 1024
  - Damit ergibt sich auch `MAX_ADDR_SPACES = 1024`: Threads können sich zwar den Adressraum teilen, aber auch nicht mehr als einen Adressraum nutzen [2].
  - Vergleich: Linux typischerweise 32.768
- TCB-Struktur:
  - `int files[MAX_PFD];  // MAX_PFD ist auf 16 gesetzt`
    - Linux: typischerweise >=300.000 (jedoch konfigurierbar!)
    - OpenBSD: `5*(NPROCESS + MAXUSERS) + 80`, wobei `NPROCESS = 30 + 16 * MAXUSERS` und `MAXUSERS=80`, somit: 5*((30+16*80)+80)+80) = 7030
  - `char cwd[256];` (maximal 256 Zeichen langer Pfadname)
  - `sighandler_t sighandlers[32];` (maximal 32 Signalhandler)
- Maximale Anzahl an Locks (`MAX_LOCKS`): 1024; gilt insg. für alle Prozesse des Systems

Zusammenfassung

- Tasks sind elementarer Bestandteil von Betriebssystemen
  - Notwendig für Multitasking
- Erzeugung und Beendigung von Prozessen kann verschiedene Auslöser haben
- Diverse Schritte zur Verwaltung von Prozessen bei Erzeugung und Beendigung notwendig
  - insb. Modifikation der Prozesstabelle inkl. Verweise auf zu (de)allokierende Ressourcen wie virtuellen Speicher, Dateideskriptoren usw.
- Parallelisierung auch im Kernel möglich
- Limits von Prozessumgebungen sind teils hart und teils konfigurierbar

Quellen

[1] A. S. Tanenbaum, H. Bos: Moderne Betriebssysteme, Pearson Studium, 4. Aufl., 2016.

[2] H.-G. Eßer, F. C. Freiling: The Design and Implementation of the ULIX Operating System, Universität Erlangen-Nürnberg, 2015.

[3] `fork(2)` Linux Manual Page, 2017.

[4] `vfork(2)` Linux Manual Page, 2012.

Anhang

Ulix Lock-Placeholders (disabled Interrupts [2])

⟨*begin critical section in kernel* 380a⟩≡

(168d 169a 184–87 209c 216b 219c 221a 255a 260a 276d 361c 362 366 391 392 416b 521a 530 539c 540c 545b 548b 551 580c 581)

```
// do nothing
```

⟨*end critical section in kernel* 380b⟩≡

(168d 169a 184–87 212 216b 219c 221a 255a 260a 276d 277a 280b 361c 362 366 391 392 416b 521b 531a 545b 580c 581)

```
// do nothing
```

The interested reader might ask: Why didn't we omit these markers from the beginning if they are empty anyway? There are two answers to this question:

1. Omitting these markers would have avoided the discussion (and identification) of critical sections, which would have avoided some nice intellectual challenges.
2. Keeping these markers allows for a future evolution of Ulix into an interruptible kernel.

In such a future Ulix version with an interruptible kernel these chunks would be implemented using ⟨*disable interrupts* 47a⟩ and ⟨*enable interrupts* 47b⟩, or with the nestable versions.

fork(2) und vfork(2)

"Under Linux, fork(2) is implemented using copy-on-write pages, so the only penalty incurred by fork(2) is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. However, in the bad old days a fork(2) would require making a complete copy of the caller's data space, often needlessly, since usually immediately afterward an exec(3) is done. Thus, for greater efficiency, BSD introduced the **vfork()** system call, which did not fully copy the address space of the parent process, but borrowed the parent's memory and thread of control until a call to execve(2) or an exit occurred. The parent process was suspended while the child was using its resources. The use of vfork() was tricky: for example, not modifying data in the parent process depended on knowing which variables were held in a register."

"**vfork()** is a special case of clone(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an execve(2)."