9. Projekt: Thread-Bibliothek

Programmier-Projekt

- Implementierung einer User-Level-Thread-Bibliothek (ähnlich POSIX Threads)
  - Threads erzeugen mit `pthread_create()`
  - Threads beenden mit `pthread_exit()`
  - auf Thread warten mit `pthread_join()`
  - Thread-ID abfragen mit `pthread_self()`
- Zur Implementation gehört:
  - Round-Robin-Scheduler für Threads
  - Timer (für unterbrechenden Scheduler)

Projekt: Grundlagen

Übersicht

- Datenstrukturen für Threads
- Context Switch und Stacks
- Timer (`alarm`, `setitimer`) unter Linux
- Signal Handler

Datenstrukturen

- Für jeden Thread einen Thread Control Block (TCB) verwalten
  - Zustandsinformationen zum Thread (blockiert, bereit etc.)
  - Thread-Context
    - Register
    - Stack, Stack Pointer, Base Pointer
  - Verwaltungsdaten (z. B. Priorität)
- `pthread_create` muss einen neuen TCB erzeugen

Stack (1)

- Programme nutzen den Stack über zwei Register:
  - **esp**, zeigt immer auf aktuelles Ende des Stacks (Stack „wächst“ nach unten)
  - **ebp**, zeigt immer auf aktuellen Stack Frame; erlaubt einer Funktion schnellen Zugriff auf
    - Aufruf-Argumente (oberhalb ebp) und
    - lokale Variablen (unterhalb ebp)

Stack (2)

- Bei Funktionsaufrufen wird der Stack benutzt:
  - Aufruf `func (a,b,c)` schreibt auf den Stack:
    - `c`, `b`, `a` (umgekehrte Reihenfolge)
    - Rücksprungadresse (Befehl nach `func(a,b,c)`)
  - `func` sichert alten **ebp** auf Stack
  - `func` reserviert Platz für lokale Variablen auf Stack, und zwar durch Anpassen von **esp**
  - in `func` Zugriff auf
    - Argumente: z. B. +4(**ebp**), +8(**ebp**), …
    - lokale Variablen: z. B. -4(**ebp**), -8(**ebp**), …

Stack (3)

```
// ebp-test.c

#include <stdio.h>   // printf

#define get_ebp(v) \
  __asm__ ("mov %%ebp, %0" : "=r"(v))

int f (int x, int y) {
  unsigned a, b;

  get_ebp (a);
  printf ("&a  : %8p \n",      &a);
  printf ("&b  : %8p \n\n",    &b);
  printf ("ebp : 0x%08x \n\n", a);
  printf ("&x  : %8p \n",      &x);
  printf ("&y  : %8p \n",      &y);
  return 0;
}

int main () {
  f (12, 24);
}
```

```
$ ./ebp-test
&a  : 0xbfad6248
&b  : 0xbfad624c

ebp : 0xbfad6258

&x  : 0xbfad6260
&y  : 0xbfad6264
```


Stack- und Base-Pointer

```
int f (int x) {
  int y = 2;
  return x+y;
}

int main () {
  int a = f (42);
  return a;
}
```

```
leave =

movl   %ebp, %esp
popl   %ebp
```

```
f:
   pushl   %ebp              // ebp sichern
   movl    %esp, %ebp        // ebp = esp
   subl    $16, %esp         // esp -= 16
   movl    $2, -4(%ebp)      // y = 2
   movl    -4(%ebp), %eax    // eax = y
   movl    8(%ebp), %edx     // edx = x
   addl    %edx, %eax        // eax += edx
   leave
   ret

main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $20, %esp
   movl    $42, (%esp)       // 42 auf Stack
   call    f                 // f(42) in eax
   movl    %eax, -4(%ebp)    // a = eax
   movl    -4(%ebp), %eax    // eax = a
   leave
   ret
```

Threads und Stacks

- Jeder Thread braucht seinen eigenen Stack (sonst: Verwirrung beim Umschalten auf einen anderen Thread)
- Der „erste“ Stack kommt automatisch; für jeden neuen Thread zusätzlichen Stack-Speicher allozieren (`malloc`)
- Beim Wechsel von einem Thread zum anderen auch zum richtigen Stack umschalten
- In der Übung: Stacks bei `pthread`-Implem.

Timer und Signal-Handler

- Umschalten zwischen mehreren Threads über Timer
- ähnlich wie bei Prozess-Scheduler: Der wird über Timer-Interrupts realisiert
- für User-Level-Threads geht es nicht mit Timer-Interrupts (Kernel kennt die Threads nicht), Alternative: Alarm-Signal und Timer:
  - Signal-Handler für das Signal `SIGALRM`
  - Regelmäßig dieses Signal erzeugen

Signal-Handler

- Über `alarm()` kann jeder Prozess Signal-Handler für verschiedene Signale eintragen
- Uns interessiert das Signal `SIGALRM` (14)
- Eintragen des Handlers mit:

```
#include <signal.h>   // signal (), SIGALRM
signal (SIGALRM, alarm_handler)
```

- Handler-Funktion implementieren:

```
void alarm_handler (int sig) {
  ...
}
```


Timer erzeugen: alarm()

- Der Aufruf `alarm(`*sec*`)` erzeugt einen (einmaligen) Timer, der nach *sec* Sekunden auslöst
- Damit der Timer regelmäßig auftritt, muss der Signal-Handler den Alarm erneuern
- `alarm()` erlaubt nur sekundengenaue Angaben, besser für Scheduling ist die Funktion `setitimer()`

Timer erzeugen: setitimer()

- `setitimer()` erwartet (u. a.) ein Argument vom Typ `struct itimerval`:

```
struct itimerval {
  struct timeval it_interval; /* next value */
  struct timeval it_value;    /* current value */
};

struct timeval {
  long tv_sec;                /* seconds */
  long tv_usec;               /* microseconds */
};
```

- Das erlaubt mikrosekunden-genaue Aufrufe

setitimer-Beispiel

- Timer zählt Echtzeit (`ITIMER_REAL`) oder nur die Zeit, in der der Prozess läuft (`ITIMER_VIRTUAL`) – für Scheduler ist die Prozesszeit interessanter
- Timer nicht nur einmalig, sondern mit Intervall

```
#include <sys/time.h>

struct itimerval timer;
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 250000;    // Alarm nach 250 msec
timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = 100000; // dann alle 100 msec
setitimer (ITIMER_VIRTUAL, &timer, NULL);
```


Thread-Scheduler

- Idee: `setitimer()` erzeugt regelmäßig ein `SIGALRM`-Signal
- Handler-Funktion für dieses Signal ist unser Scheduler
- Scheduler wählt neuen Thread aus und sorgt dafür, dass Prozess (nach Verlassen des Handlers) nicht zu aktuellem Thread, sondern zu neuem zurück kehrt
- Problem: Wie kriegt man das hin? → Genaues Verständnis des Stack-Aufbaus nötig