System-Programmierung 11: Zeitmessung

Slides: tmb.gr/syspr-11

Überblick

Diese Lektion behandelt das Thema *Zeitmessung*.

Unterscheidung von realer und Prozesszeit.

Konversion zwischen Zeit-Formaten.

Sowie Timer und Schlafen.

Zeitmessung

In Programmen betrachten wir zwei Arten von Zeit:

Echtzeit (*real time*), gemessen von einem Zeitpunkt im Kalender oder einem Fixpunkt im Programm aus, ist gut für Timestamps und periodische Aktionen.

Prozesszeit (*process time*) ist die Menge an CPU-Zeit die ein Prozess konsumiert und hilft, die Performance von Algorithmen zu optimieren.

Kalender-Zeit

Unabhängig von der Zeitzone repräsentieren UNIX Systeme die Zeit seit der *Epoche*, 01.01.1970, 00:00, Universal Coordinated Time (UTC, früher Greenwich Mean Time, GMT), ~ die Geburtsstunde von UNIX.

Auf 32-bit Linux Systemen bedeutet das, *time_t*, ein *signed int*, kann Daten von 03.12.1901, 20:45:52, bis 19.01.2038, 03:14:07 repräsentieren.

Kalender-Zeit lesen mit *time()*

time() liefert Kalender-Zeit in Sekunden seit *Epoche*:

```
time_t time( // Zeit oder (time_t) -1
  time_t *t); // Sekunden seit Epoche
```

Als Argument wird typischerweise *NULL* übergeben:

```
time_t t = time(NULL);
```

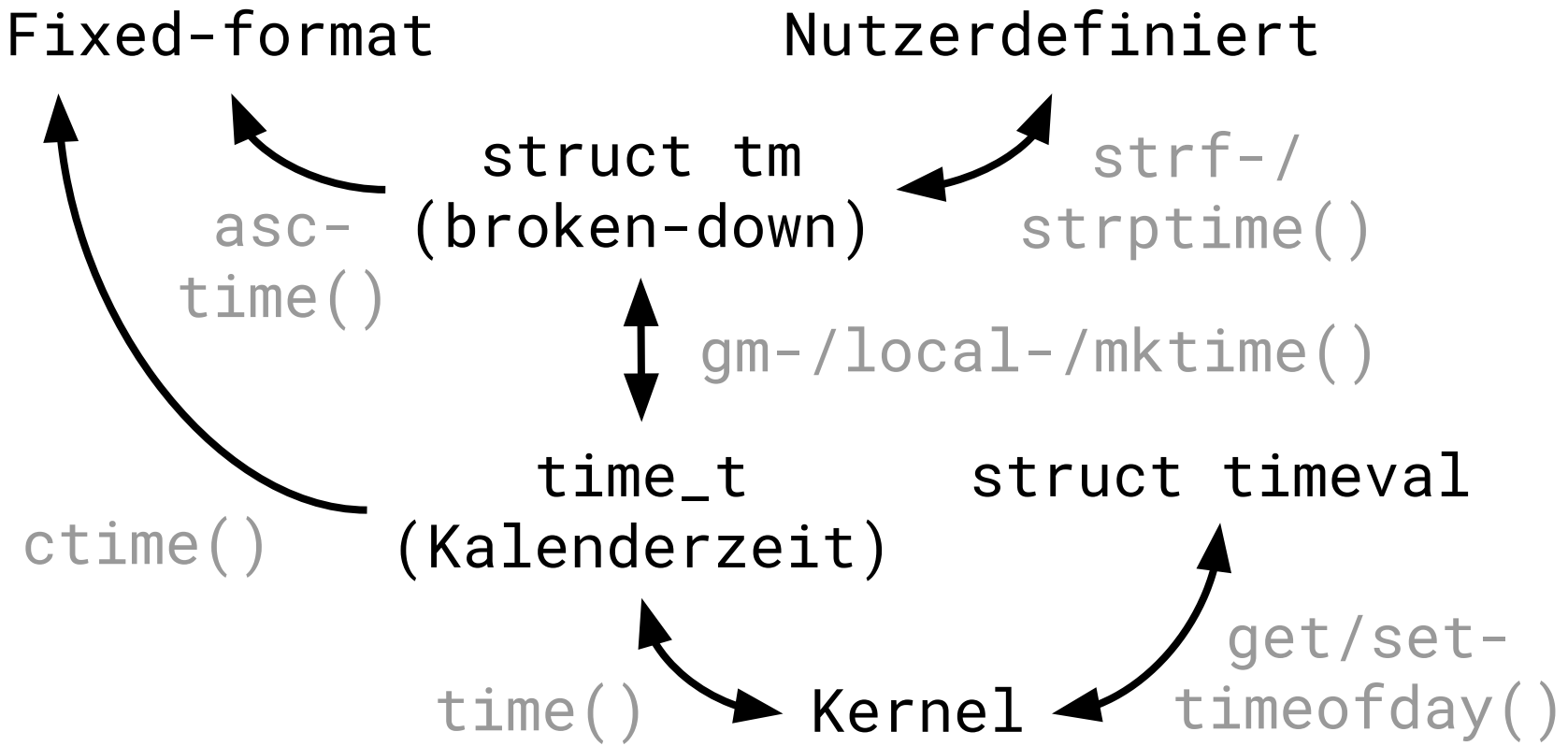
Zeit lesen/setzen mit *get/settimeofday()*

```
int gettimeofday( // 0 oder -1, errno
  struct timeval *tv,
  struct timezone *tz); // obsolet, immer NULL

int settimeofday( // 0 oder -1, errno
  struct timeval *tv,
  struct timezone *tz); // obsolet, immer NULL

struct timeval { // Zeit seit der Epoche
  time_t tv_sec;
  suseconds_t tv_usec;
};
```

Zeit-Konversion



Zeit-Konversion zu String mit *ctime()*

Zeit-Wert von *time_t* zu 26 Byte String konvertieren:

```
char *ctime(const time_t *t);
```

Das Resultat wird entsprechend der lokalen Zeitzone und DST dargestellt, z.B. *Wed Jun 8 14:22:34 2011*

Der String enthält ein `'\n'` und ist `'\0'`-terminiert, sowie statisch alloziert, bis zum nächsten Aufruf.

Zeit-Konversion mit *gm- / localtime()*

Zeit von *time_t* zu "broken-down" UTC konvertieren:

```
struct tm *gmtime(const time_t *t);
```

Zeit von *time_t* zu "broken-down" Lokalzeit:

```
struct tm *localtime(const time_t *t);
```

Bei Fehlern wird *NULL* retourniert und *errno* gesetzt.

Zeit im "broken-down" Format *struct tm*

```
struct tm {
  int tm_sec; // Sekunden [0..60]
  int tm_min; // Minuten [0..59]
  int tm_hour; // Stunde [0..23]
  int tm_mday; // Tag im Monat [1..31]
  int tm_mon; // Monat im Jahr [0..11]
  int tm_year; // Jahre seit 1900
  int tm_wday; // Wochentag [0..6], So = 0
  int tm_yday; // Tag im Jahr [0..365]
  int tm_isdst; // Daylight Saving Time Flag
}
```

Zeit-Konversion zu *time_t* mit *mktime()*

Zeit von "broken-down" zu *time_t* konvertieren:

```
time_t mktime(struct tm *t);
```

Bei Fehler resultiert *(time_t) -1* und *errno* ist gesetzt.

Die *_sec*, *_min*, *_hour*, *_mday*, *_month* und *_year* Werte werden beim Überlauf eines Werts angepasst.

Die Funktion ignoriert *tm_wday* und *tm_yday*, bzw. setzt gültige Werte dafür im *struct tm* Argument *t*.

Zeit-Konversion zu String mit *asctime()*

Zeit-Wert von *struct tm* zu String konvertieren:

```
char *asctime(const struct tm *t);
```

Das Resultat wird ohne Änderung der Zeitzone oder DST dargestellt, z.B. *Wed Jun 8 14:22:34 2011*

Der String enthält ein `'\n'` und ist `'\0'`-terminiert, sowie statisch alloziert, bis zum nächsten Aufruf.

Hands-on, 5': Kalender-Zeit

Lesen Sie das folgenden [TLPI] Beispiel Programm:
`calendar_time.c`

Vergleichen Sie den Output der Kommandos:

```
$ date
$ ./calendar_time
```

Zeit-Konversion zu String mit *strftime()*

Zeit von *struct tm* zu String konvertieren, formatiert:

```
size_t strftime( // Länge von s ohne \0, od. 0
  char *s, // Zeit als String, gemäss Format
  size_t max, // Max. erwartete String-Länge
  const char *format, // Format String
  const struct tm *t); // Zeit (broken-down)
```

z.B. ISO Datum und Zeit: `"%Y-%m-%dT%H:%M:%SZ"`

```
2018-12-29T12:17:25Z // Z nur, falls UTC Zeit
```

Zeit String parsen mit *strptime()*

Datum und Zeit *t* aus String *s* parsen mit *strptime()*:

```
char *strptime( // nächstes Zeichen in s
  const char *s, // String mit Datum
  const char *format, // Format String
  struct tm *t); // Resultat (broken-down)
```

Falls Parsen nicht erfolgreich, kommt *NULL* zurück.

Vor dem Aufruf, *t* initialisieren mit *memset()*:

```
memset(&t, 0, sizeof(struct tm));
```

Hands-on, 5': Zeit parsen / formatieren

Lesen Sie das folgenden [TLPI] Beispiel Programm:
`strtime.c`

Vergleichen Sie den Output der Kommandos:

```
$ ./strtime "9:39:46pm 1 Feb 2011"\
 "%I:%M:%S%p %d %b %Y"
$ ./strtime "9:39:46pm 1 Feb 2011"\
 "%I:%M:%S%p %d %b %Y" "%F %T"
```

Geben Sie das Datum im ISO 8601 Format aus.

Zeitzonen

Zeitzonen bestimmen, welche Zeit in einer Region gilt.

Zeitzone übergeben via Umgebungsvariable *TZ*, z.B.

```
$ export TZ=':UTC'
$ date

$ export TZ=':US/Hawaii'
$ date

$ ls /usr/share/zoneinfo # weitere Zeitzonen
```

Zeitzone initialisieren mit *tzset()*

Zeitzone bzw. -Variablen initialisieren mit *tzset()*:

```
void tzset(void);
```

Der Call *tzset()* liest die Zeitzone aus der Umgebungsvariable *TZ* und setzt die folgende globalen Variablen:

```
extern char *tzname[2]; // Zone und DST Zone
extern long timezone; // Differenz zu UTC in s
extern int daylight; // Nicht-Null, falls DST
```

Locale

Die *Locale* (wörtlich "der Schauplatz") bestimmt, wie Zahlen, Beträge, Datum und Zeit dargestellt werden, für Internationalisierung bzw. Lokalisierung.

Locales sind in Dateien abgelegt:

```
$ ls /usr/share/locale
```

z.B. in der Schweiz:

`de_CH.UTF-8, fr_CH.UTF-8 und it_CH.UTF-8`

Locale lesen und setzen mit *setlocale()*

Locale für Kategorie lesen oder setzen mit *setlocale()*:

```
char *setlocale( // Locale String, oder NULL
  int category, // Teil der Locale, od. LC_ALL
  const char *locale); // "" => Env. Variablen
```

Locale Kategorien sind z.B.

`LC_TIME, LC_NUMERIC, LC_MONETARY, LC_PAPER, …`

Aktuelle Locale lesen: `setlocale(LC_ALL, NULL);`

Prozesszeit

Prozesszeit ist die *CPU Zeit*, die ein Prozess seit seiner Kreation verbraucht hat, mit folgenden Komponenten:

User CPU Zeit ist die im User Mode verbrachte Zeit, die auch als *virtuelle Zeit* bezeichnet wird.

System CPU Zeit ist im Kernel Mode verbrachte Zeit, während System Calls oder z.B. beim Paging.

CPU und reale Zeit messen mit *times()*

Zeitmessung mit *times()*, analog zu *time* Kommando:

```
clock_t times( // Ticks seit t0, fix, arbiträr*
  struct tms *t); // User & System Zeit, Ticks

struct tms { // *) nur Delta messen macht Sinn
  clock_t tms_utime; // User Zeit
  clock_t tms_stime; // System Zeit
  clock_t tms_cutime; // User Zeit der Kinder
  clock_t tms_cstime; // System Zeit der Kinder
}; // sysconf(_SC_CLK_TCK) => Ticks/Sekunde
```

CPU Zeit messen mit *clock()*

CPU Zeit messen mit *clock()*, total, User und Kernel:

```
clock_t clock(void); // oder (clock_t) -1
```

Dieser Call verwendet eine andere Clock Auflösung:

```
CLOCKS_PER_SEC; // clock() spezifische Ticks/s
```

Auf Linux umfasst das Total nur Parent Prozess Zeit, auf anderen Plattformen auch Child Prozess Zeit.

Hands-on, 15': Zeitmessung

time.!c

Schreiben Sie ein eigenes *time* Programm, *my_time.c*

Das zu messende Programm soll aus *argv[1]* gelesen und mit *fork() und execve()* gestartet werden.

Der Parent Prozess wartet mit *wait()*, und bestimmt die Laufzeit, real und CPU Zeit, des Child Prozesses.

Die Ausgabe soll derjenigen von *time* entsprechen.

Timer und Schlafen

Ein *Timer* erlaubt es einem Prozess, Notifikationen für sich einzuplanen, auf einen späteren Zeitpunkt.

Schlafen (sleeping) suspendiert einen Prozess oder Thread für eine zuvor festgelegte Zeitdauer.

Neben dem klassischen UNIX API gibt es POSIX und Linux spezifische APIs um Timer zu erstellen.

Intervall Timer setzen mit *setitimer()*

Intervall Timer setzen mit *setitimer()*:

```
int setitimer( // 0 oder -1, errno
  int which, // ITIMER_REAL|VIRTUAL|PROF
  const struct itimerval *new_value, // ist neu
  struct itimerval *old_value); // wird ersetzt
```

Wenn der Timer ausläuft, wird ein Signal verschickt:

```
ITIMER_REAL    => SIGALRM
ITIMER_VIRTUAL => SIGVTALRM
ITIMER_PROF    => SIGPROF
```

Struct *itimerval*

Für *new_value* und *old_value* von *setitimer()*:

```
struct itimerval {
  struct timeval it_interval; // Timer Periode
  struct timeval it_value; // Zeit bis Signal
}; // it_interval = {0,0} => einmaliger Timer

struct timeval {
  time_t tv_sec; // Sekunden
  suseconds_t tv_usec; // Mikrosekunden
};
```

Intervall Timer lesen mit *getitimer()*

Intervall Timer lesen mit *getitimer()*:

```
int getitimer( // 0 oder -1, errno
  int which, // ITIMER_REAL|VIRTUAL|PROF
  struct itimerval *curr_value);
```

Verbleibende Zeit bis zum nächsten Timer Signal:

```
struct timeval t = curr_value.it_value;
```

Die verbleibende Zeit wird kürzer bis zum Signal, und beginnt dann wieder bei *curr_value.it_interval*.

Hands-on, 5': Timer

Lesen Sie das folgenden [TLPI] Beispiel Programm:
`real_timer.c`

Testen Sie den Timer, z.B. mit den Kommandos:

```
$ ./real_timer 1 800000 1 0 # 1.8s, 1s Periode
$ ./real_timer 3 0 # einmaliger Timer, nach 3s
```

Timer setzen mit *alarm()*

Einmalig auftretenden Timer setzen mit *alarm()*:

```
unsigned int alarm( // verbleibende Sekunden
  unsigned int seconds); // nächstes Signal
```

Dieser Aufruf kann nie zu einem Fehler führen.

Ablaufen des Timers löst das *SIGALRM* Signal aus.

Ein existierender Timer kann gelöscht werden mit:

```
alarm(0);
```

Timer Genauigkeit

Je nach Prozessorlast kann es sein, dass ein Prozess erst kurz nach Ablauf eines Timers wieder läuft.

Dies hat aber keinen Einfluss auf das nächste Signal, Intervalle werden genau eingehalten, ohne Drift.

Die Genauigkeit eines Timers ist auf modernen Linux Systemen durch die Frequenz der Hardware Clock beschränkt, und erreicht ca. eine Mikrosekunde.

Prozess suspendieren mit *sleep()*

Prozess suspendieren für fixe Zeitspanne mit *sleep()*:

```
unsigned int sleep( // 0 oder verbleibende s
  unsigned int seconds);
```

Auf Linux ist der Call mit *nanosleep()* implementiert:

```
int nanosleep( // 0 oder -1, errno
  const struct timespec *requested,
  struct timespec *remaining);
struct timespec {
  time_t tv_sec; long tv_nsec;
};
```

Selbststudium: Vorbereitung Assessment

Beginnen Sie damit, den behandelten Stoff zu wiederholen, für das kommende Assessment.

Feedback oder Fragen?

Gerne im Slack https://fhnw-syspr.slack.com/

Oder per Email an thomas.amberg@fhnw.ch

Danke für Ihre Zeit.


kim on Twitter: "It's 2019.
https://twitter.com/grufwub/status/1079893186573672449
Home
Notifications
Messages
Search Twitter
Tweet
kim
@grufwub
Follow
It's 2019.
We're now exactly halfway between y2k and the 32-bit Unix time overflow.
1:12 AM - 1 Jan 2019
2,193 Retweets 5,691 Likes
55
2.2K
5.7K
Tweet your reply
DrTune @drtune · 13h
Replying to @grufwub
the 32-bit time overflow is going to be huge; IMO a much, much bigger problem than Y2K. Vast amounts of code use uint32_t time
kim
@grufwub
Ex-physicist, dev
hacker / infosec-a
and hobbyist pho
A.I. generated. Tra