Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2012/2013

Conditions/Semaphore

Autotools

Conditions

Wenn mehrere Threads synchronisiert ablaufen sollen, kann dies auch über *Conditions* geschehen, bei denen Threads blockiert werden, bis sie benachrichtigt werden, dass sie weiter machen sollen.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *abstime);
```

Hierbei muss der Thread den angegebenen Mutex aktuell besitzen. Die erste Variante wartet beliebig lange auf die Benachrichtigung, die zweite maximal die angegebene Dauer. Im Fehlerfall ist der Rückgabewert ungleich 0 (und errno gleich `ETIMEDOUT` beim Timeout der zweiten Version).
Achtung: Der Thread gibt während des Wartens den Mutex wieder frei!

Benachrichtigung der wartenden Threads

Über die Condition können einer oder alle wartenden Threads benachrichtigt werden:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Hierbei werden einer oder alle Threads aufgeweckt, wobei diese(r) dann wieder *auf den beim Wait-Befehl angegebenen Mutex wartet*.
Selbst im Fall eines Broadcasts über die Condition werden also i. A. nicht alle wartenden Threads gleichzeitig weiterarbeiten, sondern synchronsiert über den Mutex.

Erzeuger–Verbraucher über Ringpuffer

Im Folgenden sollen mehrere Erzeuger über einen Ringpuffer mehrere Verbraucher mit Daten (in diesem Fall zufällige Floats) versorgen.
Dieser Ringpuffer besteht aus einem Array und soll nach dem Prinzip FIFO arbeiten. Dafür merkt er sich jeweils die Position des ältesten Werts im Puffer und füllt diesen ringförmig, belegt also am Anfang des Array frei gewordene Plätze neu, nachdem beim Füllen das Array-Ende erreicht wurde.

Beim Einspeisen und Abholen von Werten aus dem Ringpuffer durch die Erzeuger bzw. Verbraucher werden folgende Fälle durch Conditions berücksichtigt:

- Der Ringpuffer ist leer, es stehen also keine Daten für Verbraucher zur Verfügung;
- der Ringpuffer ist voll, die Erzeuger können also keine Daten abliefern.

Teil 1: Definition des Ringpuffers

Datei ringbuffer.h:

```
#include <stddef.h>

typedef struct {
    float* buf;
    size_t capacity;
    size_t start;
    size_t count;
} rb_t;

int rb_init(rb_t *rb, size_t capacity);
int rb_destroy(rb_t *rb);
int rb_push(rb_t *rb, float value);
int rb_pop(rb_t *rb, float *value);
int rb_empty(rb_t *rb);
int rb_full(rb_t *rb);
```

Teil 2: Implementierung des Ringpuffers

Datei ringbuffer.c:

```
int rb_init(rb_t *rb, size_t capacity) {
    float *buf = calloc(capacity, sizeof(float));
    if (!buf) return(-1);
    rb->buf = buf;
    rb->capacity = capacity;
    rb->start = 0;
    rb->count = 0;
    return(0);
}

int rb_destroy(rb_t *rb) {
    if (rb->buf==NULL) return(-1);
    free(rb->buf);
    rb->buf=NULL;
    rb->capacity = 0;
    rb->start = 0;
    rb->count = 0;
    return(0);
}
```

Implementierung des Ringpuffers (Forts.)

```
int rb_push(rb_t *rb, float value) {
    if (rb->count>=rb->capacity) return(-1);
    size_t index = (rb->start + rb->count++) % rb->capacity;
    rb->buf[index] = value;
    return(0);
}

int rb_pop(rb_t *rb, float *value) {
    if (rb->count==0) return(-1);
    *value = rb->buf[rb->start++];
    rb->start %= rb->capacity;
    rb->count--;
    return(0);
}

int rb_empty(rb_t *rb) {
    return rb->count==0;
}

int rb_full(rb_t *rb) {
    return rb->count>=rb->capacity;
}
```

Logik der Conditions

Für die Erzeuger und Verbraucher werden nun folgender Mutex und folgende Conditions benutzt:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t data_wanted = PTHREAD_COND_INITIALIZER;
```

Der Mutex wird von Erzeugern und Verbrauchern gemeinsam verwendet, wobei die Erzeuger im Falle eines vollen Puffers darauf warten, dass sie über die Condition `data_wanted` von einem Verbraucher benachrichtigt werden, dass er Daten abgeholt hat und nun wieder Platz frei ist. Im Fall des leeren Puffers warten die Verbraucher auf die Condition `data_available`, über die ein Erzeuger signalisiert, dass er Daten abgeliefert hat.

Teil 4: Erzeuger

```
void* produce(void* arg) {
    rb_t *buf = (rb_t*) arg;
    printf("Thread %lu ready to create data\n", pthread_self());
    while (1) {
        pthread_mutex_lock(&lock);
        while (rb_full(buf)) pthread_cond_wait(&data_wanted, &lock);
        float data = .0001F*rand();
        printf("Producer %lu created %f\n", pthread_self(), data);
        rb_push(buf, data);
        pthread_cond_broadcast(&data_available);
        pthread_mutex_unlock(&lock);
        usleep(1000);
    }
}
```

Teil 5: Verbraucher

```
void* consume(void* arg) {
    rb_t *buf = (rb_t*) arg;
    printf("Thread %lu ready to consume data\n", pthread_self());
    while (1) {
        pthread_mutex_lock(&lock);
        while (rb_empty(buf)) pthread_cond_wait(&data_available, &lock);
        float data;
        rb_pop(buf, &data);
        printf("Consumer %lu got %f\n", pthread_self(), data);
        pthread_cond_signal(&data_wanted);
        pthread_mutex_unlock(&lock);
        usleep(2000);
    }
}
```

Hauptprogramm

```
int main() {
    srand(time(NULL));
    rb_t buf;
    rb_init(&buf, 10);
    pthread_t e1, e2, v1, v2, v3;
    pthread_create(&v1, NULL, consume, &buf);
    pthread_create(&v2, NULL, consume, &buf);
    pthread_create(&v3, NULL, consume, &buf);
    pthread_create(&e1, NULL, produce, &buf);
    pthread_create(&e2, NULL, produce, &buf);

    sleep(10);
    pthread_cancel(v1);
    ...
    pthread_cancel(e2);
    pthread_join(v1, NULL);
    ...
    pthread_join(e2, NULL);
    rb_destroy(&buf);
    return(0);
}
```

Semaphore

Über einen Mutex konnte mit `lock` und `unlock` ein kritischer Bereich so geschützt werden, dass sich immer nur *ein* Thread in diesem Bereich befinden kann.

Über eine *Semaphore* kann ein Bereich für eine bestimmte Anzahl von Threads freigegeben werden. Technisch ist eine Semaphore eine (atomare) Ganzzahlvariable, die mit einem Wert größer Null initialisiert wird.
Der geschützte Bereich wird nur betreten, wenn die Semaphore größer Null ist (ansonsten wird gewartet) und diese dann heruntergezählt. Beim Verlassen des Bereichs wird die Semaphore wieder um eins erhöht.

Eine Semaphore mit der Maximalzahl 1 (*binary semaphore*) entspricht dabei einem Mutex.

POSIX-Semaphore

```
#include <semaphore.h>

int    sem_init(sem_t *s, int pshared, unsigned max);
int    sem_destroy(sem_t *s);

int    sem_wait(sem_t *s);
int    sem_timedwait(sem_t *s, const struct timespec * abstime);
int    sem_trywait(sem_t *s);
int    sem_post(sem_t *s);
```

Bei der Initialisierung wird die Maximalzahl von Threads festgelegt.
Mit den Wait-Funktionen wird beim Betreten des Bereichs die Semaphore dekrementiert, beim Post wieder inkrementiert.
Die drei Wait-Funktionen unterscheiden sich darin, ob beliebig lange, eine endliche Zeit oder gar nicht gewartet wird, bis ein Platz frei ist.

Beispiel

Hier haben in der Semaphore maximal 3 von 10 Threads Platz:

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void* run(void *arg) {
    sem_t *s = (sem_t*) arg;
    while (1) {
        sem_wait(s);
        printf("Thread %lu in Semaphore\n", pthread_self());
        sleep(2);
        printf("Thread %lu verlaesst Semaphore\n", pthread_self());
        sem_post(s);
        sleep(1);
    }
}
```

Beispiel (Forts.)

```
int main() {
    sem_t sem;
    sem_init(&sem, 0, 3);

    pthread_t threads[10];
    int i;
    for (i=0; i<10; i++) pthread_create(&threads[i], NULL, run, &sem);
    sleep(10);
    for (i=0; i<10; i++) {
        pthread_cancel(threads[i]);
        pthread_join(threads[i], NULL);
    }
    sem_destroy(&sem);

    return(0);
}
```

Herausforderung: Schreiben portabler Software

Damit ein Projekt auf verschiedenen Unix-Systemen übersetzt werden kann, sind verschiedene Schritte nötig:

- Prüfen auf
  - Existenz des C-Compilers und weiterer notwendiger Programme,
  - Existenz benötigter Header-Dateien
- Anpassen des Makefiles an die Begebenheiten des Systems:
  - Name des C-Compilers und sonstiger Programme
  - Installationspfade

Autotools

Für die automatische Konfiguration des Makefiles an das lokale System existiert das Paket *autotools*, bestehend aus den Programmen *autoheader* und *autoconf*. Hierbei entstehen aus einem generischen Makefile (`Makefile.in`, das wiederum mit *automake* generiert werden kann) und den vorhandenen Quellen ein Skript `configure`, das das Makefile erzeugt.

Typischer Ablauf:

```
./configure --prefix=/usr
make all
make install
```

Beispiel: ringbuffer

Als Beispiel soll aus den Quellen vom *ringbuffer* aus dem ersten Teil der Vorlesung eine dynamische Bibliothek generiert werden, die dann ebenso wie die Header-Datei und die Man-Page installiert wird.
Hierzu wird folgendes `Makefile.in` benutzt:

```
top_srcdir = @top_srcdir@
srcdir = @srcdir@

CC = @CC@
CFLAGS = -I$(top_srcdir) -I$(srcdir) @CFLAGS@
LDFLAGS = @LDFLAGS@
LIBS = @LIBS@
INSTALL = @INSTALL@
INSTALL_DATA = @INSTALL_DATA@
LN_S = @LN_S@
```

Makefile.in (Forts.)

```
prefix = @prefix@
exec_prefix = @exec_prefix@
libdir = @libdir@
includedir = @includedir@
datarootdir = @datarootdir@
datadir = @datadir@
mandir = @mandir@

.PHONY: all install clean

SOVERSION = 1
RELEASE = 0

SOURCES = ringbuffer.c
HEADERS = ringbuffer.h
man3dir = $(mandir)/man3
man3_MANS = ringbuffer.3
```

Makefile.in (Forts.)

```
all: libringbuffer.so.$(SOVERSION).$(RELEASE)

libringbuffer.so.$(SOVERSION).$(RELEASE): $(SOURCES:.c=.o)
    $(CC) -shared -o $@ $(LDFLAGS) \
          -Wl,-soname=libringbuffer.so.$(SOVERSION) $+ $(LIBS)

install:
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(libdir)
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(includedir)
    $(INSTALL) libringbuffer.so.$(SOVERSION).$(RELEASE) \
               $(DESTDIR)$(libdir)
    $(INSTALL_DATA) $(HEADERS) $(DESTDIR)$(includedir)
    (cd $(DESTDIR)$(libdir); \
    rm -f libringbuffer.so.$(SOVERSION); \
    $(LN_S) libringbuffer.so.$(SOVERSION).$(RELEASE) \
            libringbuffer.so.$(SOVERSION); \
    rm -f libringbuffer.so; \
    $(LN_S) libringbuffer.so.$(SOVERSION) libringbuffer.so)
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(man3dir)
    for man in $(man3_MANS); do \
        $(INSTALL_DATA) $$man $(DESTDIR)$(man3dir); \
    done
```

Makefile.in (Forts.)

```
clean:
    rm -f libringbuffer.so.$(SOVERSION).$(RELEASE) \
          $(SOURCES:.c=.o)

%.o: %.c
    $(CC) -c $(CFLAGS) $<

%.d: %.c
    $(CC) -MM $(CFLAGS) $< | sed 's,\($*\).o[ ]*:,\1.o $@:,' > $@

ifneq ($(MAKECMDGOALS),clean)
include $(SOURCES:.c=.d)
endif
```

Generieren des configure-Skripts

In diesem einfachen Beispiel wird folgende Datei `configure.ac` benutzt:

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.68])
AC_INIT([ringbuffer], [1.0], [mail@example.com])
AC_CONFIG_SRCDIR([ringbuffer.h])

# Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL
AC_PROG_LN_S

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Hieraus wird dann mit `autoconf` das Skript `configure` erzeugt, mit dem dann das richtige Makefile automatisch generiert werden kann.

Ausblick: Prüfen auf verschiedene Header-Files

Die Stärke der *autotools* liegt darin, dass Quelldateien analysiert werden können, um Tests auf verschiedene Header-Files, Funktionen etc. zu generieren:

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif

#include <stdio.h>
#ifdef STDC_HEADERS
# include <stdlib.h>
# include <stddef.h>
#else
# ifdef HAVE_STDLIB_H
# include <stdlib.h>
# endif
#endif
#ifdef HAVE_STRING_H
# if !defined STDC_HEADERS && defined HAVE_MEMORY_H
#  include <memory.h>
# endif
# include <string.h>
#endif
```

Workflow der *autotools*

- `autoscan` analysiert den Quelltext und erzeugt `configure.scan` mit folgendem Inhalt:

```
AC_CONFIG_HEADERS([config.h])
AC_CHECK_HEADERS([memory.h stddef.h stdlib.h string.h])
```

- Nach Erzeugen von `configure.ac` anhand von `configure.scan` wird mit `autoheader` die Datei `config.h.in` generiert.
- Mit `autoconf` wird das Skript `configure` erzeugt.
- Durch Aufruf von `configure` entsteht das `config.h` mit den Informationen darüber, welche der verlangten Headerdateien auf dem System vorhanden sind.