Pthreads and applications of mutex-abstraction

Wim H. Hesselink, Jan Eppo Jonker, 16th January 2001
Dept. of Mathematics and Computing Science, Rijksuniversiteit Groningen
P.O.Box 800, 9700 AV Groningen, The Netherlands
Email: {wim,janeppo}@cs.rug.nl
Web: http://www.cs.rug.nl/~wim

Abstract

POSIX threads are light-weight processes with mutexes and condition variables for synchronization. After an analysis of the main primitives, the paper develops a theory of mutex abstraction. This is a method for program transformation that eliminates mutexes and coarsens the grain of atomicity for the ease of verification. The method is illustrated by the construction of semaphores, a bounded buffer, a barrier, and a conditional critical region.

Keywords concurrency, pthread, mutex, condition variable, correctness proof

1 Introduction

For more than thirty years, the semaphore introduced in [4] has been one of the main synchronization primitives for co-operating sequential processes in a shared variable setting. To some extent, this was a theoretical status. In practice, many machines and operating systems only offered weaker synchronization primitives. The diversity of these weaker primitives made it hard to treat them systematically.

In recent years, the term threads has emerged for the elementary subprocesses of an application process, cf. [3, 6]. Formally speaking, a thread is a process that is not a parallel composition of processes. In 1998, the POSIX committee has specified a standard (1003.1c) for the handling of threads and their synchronization. Threads with these primitives are called POSIX threads or *pthreads*. The standard provides mutexes, condition variables and associated commands, cf. [3, 6]. The main synchronization primitives for pthreads are **wait**, **signal**, and **broadcast**. These primitives are also provided by the programming language Modula-3, cf. [5].

Semaphores are only an optional feature in POSIX, cf. [6] p. 177, whereas they are absent in Modula-3, cf. [5]. This has the effect that the efforts towards a methodology for using semaphores (e.g. [1, 7]) have to be transferred to mutexes and condition variables. It is known that semaphores can be implemented with these primitives, cf. [6]. Below we describe implementations of a plain semaphore and of a semaphore with a waiting list. Semaphores are powerful enough for all solvable synchronization and communication problems in shared variable systems, but it is often more efficient to make a new implementation with mutexes and condition variables than to use semaphores that are themselves built on these primitives.

Semaphores can be used for both mutual exclusion and synchronization. In the pthread primitives these capabilities have been separated in a nonorthogonal way. Mutexes serve for mutual exclusion. The commands **wait**, **signal**, **broadcast** serve for synchronization but cannot be used without a mutex. Yet the separation seems to be helpful. Mutexes are simpler to argue about than semaphores. Their limitations enforce a discipline in programming that makes static analysis of the code easier by allowing a coarsening of the grain of atomicity.

Formalizing some of the rules of structured locking [6], we develop a small theory of so-called mutex abstraction that enables rigorous correctness proofs. This theory is then used to implement semaphores, a barrier, and conditional critical regions.

The theory can be summarized as follows. The starting point is the mutex guarded region, the program fragment between **lock** and **unlock**. A shared variable is said to be guarded by mutex `m` if it only occurs in regions guarded by `m`. Mutex `m` is abstractable if it guards all shared variables occurring in its regions. Abstraction of an abstractable mutex `m` transforms each region of `m` into at most $W + 1$ atomic commands where W is the number of **wait** statements of the region.

In practice, the regions usually contain at most one **wait** statement. We describe two cases, repetitive initial waiting and conditional final waiting, of regions with one **wait** statement that nevertheless abstract to a single atomic command. Repetitive initial waiting is used to implement semaphores. Conditional final waiting occurs in the implementation of the barrier. Conditional critical regions rely on a variation of repetitive initial waiting.

Overview. In section 2, we pin down the semantics of the main synchronization primitives by means of atomicity brackets and **await** statements. In section 3, we define mutex-guarded regions, develop a theory of mutex abstraction, and describe the effect of mutex abstraction on the four patterns of mutex-guarded regions that occur in this paper. The two kinds of semaphores are constructed in section 4. Section 5 contains the specification, construction, and verification of a barrier. In section 6, we give an implementation of conditional critical regions, i.e., general systems of **await** statements, cf. [1]. Concluding remarks are made in section 7.

Notation. We use a programming notation like Modula-3, with assignments $x := E$, **inc** and **dec** for incrementation and decrementation, conditional statements of the form **if then** [**else**] **end**, and repetitions of the form **while do end** and **loop end**. We use type writer font for shared variables and italic for private variables. We write $x.q$ for the value of private variable x of thread q.

2 The semantics of mutexes and condition variables

If S is a command, we write $\langle S \rangle$ to indicate that command S must be executed without interference by other threads. The angled brackets $\langle\,\rangle$ are called atomicity brackets. We use the notation $\langle$ **await** B **then** S $\rangle$ to denote the atomic command to execute S if B holds and to wait otherwise, cf. [1, 2]. It is called a *simple await statement* if S is equivalent to *skip*, otherwise a *compound await statement*. Simple await statements are easily (though not efficiently) implemented by busy waiting. We do not know of any concurrent language that offers the compound await statement as a primitive. In this paper compound await statements serve as a specification mechanism and as a stage in program development.

For mutexes, we only use the POSIX primitives `pthread_mutex_lock` and `pthread_mutex_unlock`, which are abbreviated by **lock** and **unlock**. Every thread has a thread identifier, which is indicated by *self* in the code for the thread. We regard a mutex as a shared variable of type *Thread*. We say that thread p owns mutex `m` iff $\mathtt{m} = p$. Mutex `m` is said to be free iff $\mathtt{m} = \perp$; this holds initially. The value (owner) of a mutex can only be modified by the commands **lock** and **unlock**, which are specified by

(0) $\quad \textbf{lock}(\mathtt{m}): \quad \langle\, \textbf{await} \;\; \mathtt{m} = \perp \;\; \textbf{then} \;\; \mathtt{m} := \mathit{self} \,\rangle\;;$
$\quad\;\; \textbf{unlock}(\mathtt{m}): \quad \langle\, \textbf{await} \;\; \mathtt{m} = \mathit{self} \;\; \textbf{then} \;\; \mathtt{m} := \perp \,\rangle\;.$

The description of **unlock** may be unexpected. It enforces that a lock can be unlocked only by its owner. According to (0), a thread that tries to unlock a mutex it does not own, has to wait indefinitely.

We regard a condition variable `v` as a shared variable of the type *Set* of *Thread*, which is initially empty. We use the instruction *enter* `v` to prescribe that the acting thread places its identifier in `v` and then waits until another thread removes that identifier from `v` and thus releases the waiting thread.

We only use the POSIX primitives `pthread_cond_wait`, `pthread_cond_broadcast`, and `pthread_cond_signal`, abbreviated by **wait**, **broadcast**, and **signal**. These primitives are expressed as follows.

(1) $\quad \textbf{wait}\ (\mathtt{v},\mathtt{m}):$
$\qquad \langle\, \textbf{unlock}\ (\mathtt{m});\ \mathit{enter}\ \mathtt{v} \,\rangle\;;$
$\qquad \textbf{lock}\ (\mathtt{m})\;.$

Note that command **wait** consists of *two* atomic commands: one to start waiting and one to lock when released. Also, note that a thread had better own the mutex to execute **wait**.

Command **broadcast**(`v`) releases all threads waiting in `v`. This means that each of them is removed from `v`, and can make progress again as soon as it acquires the mutex. This is expressed in

(2) $\quad \textbf{broadcast}\ (\mathtt{v}): \quad \langle\, \textit{release all threads from}\ \mathtt{v} \,\rangle\;.$

Command **signal**(v) is equivalent to *skip* if v is empty. Otherwise, it releases at least one waiting thread. This is expressed in

(3) **signal** (v) :
⟨ **if not** *isEmpty*(v) **then** *release some threads from* v **end** ⟩ .

According to both [5] and [6], the **signal** may wake up more than one waiting thread. It is the intention that it usually releases only one waiting thread. The requirement to wake up precisely one waiting thread is not chosen since it can be expensive to implement on multiprocessors, cf. [6] p. 35.

The text of [3] p. 296 suggests that the signal should awaken precisely one waiting thread. This specification also occurs in some Linux manual pages. We call this version the strong signal. We would prefer the standard to specify strong signals, but even if it currently does so, it is likely that older implementations only provide signals that may wake up more than one waiting thread. Below we therefore reckon with both versions. We use the formal definition

(4) **strong-signal** (v) :
⟨ **if not** *isEmpty*(v) **then** *release one thread from* v **end** ⟩ .

It is important to realize that, when some thread signals v and thus releases a waiting thread, the latter need not be able to (immediately) lock the mutex. Usually, the signalling thread owns the mutex and when it releases the mutex, some other thread may obtain the mutex first. This is the reason for the usual advice to put **wait** calls inside loops that retest the waiting condition.

3 Mutex abstraction

In this section we formalize some of the rules of structured code locking and structured data locking [6] and describe the formal consequences of these rules. We introduce mutex guarded regions to formalize code locking, abstractability of mutexes to formalize data locking, and mutex abstraction as a program transformation to ease formal verification.

The usefulness of mutexes stems from the following observations. There is always at most one thread that holds the mutex. Since such a thread must release the mutex before another thread can acquire it, the code of the thread usually only contains **lock** and **unlock** instructions in fragments of the form

(5) **lock** (m) ; S ; **unlock** (m)

where S is a command that does not contain **lock** or **unlock** instructions, but may contain **wait** instructions. The fragment from **lock** to **unlock** is called a *mutex-guarded region* of m, MG-region for short.

When a thread executes an instruction **wait**(v,m) in an MG-region of m, it releases mutex m, arrives at the so-called *re-entry* location, places its thread identifier in v, and starts waiting. For the ease of discussion, we label the re-entry location of **wait**(v,m) with $\mathtt{v}^{\#}$ where v is the condition variable used (we assume that, for every v and every thread, **wait**(v, m) occurs at most once in the code). We use the notation q **at** L to indicate that the next instruction to be executed by thread q is labelled L. Since a thread can only leave location $\mathtt{v}^{\#}$ when it has been released from v, we always have the invariant

$$q \in \mathtt{v} \quad \Rightarrow \quad q \ \mathbf{at}\ \mathtt{v}^{\#}\ .$$

It is useful also to introduce the set R(v) of the *ready* threads at the re-entry location

$$\mathtt{R(v)} = \{q \mid q\ \mathbf{at}\ \mathtt{v}^{\#}\ \wedge\ q \notin \mathtt{v}\}\ .$$

We obviously have the invariant (WL), which stands for "waiting at label":

$$\text{(WL)} \qquad q\ \mathbf{at}\ \mathtt{v}^{\#} \quad \equiv \quad q \in \mathtt{v} \quad \vee \quad q \in \mathtt{R(v)}\ .$$

We now come to the formalization of data locking. A shared variable is said to be *guarded* by mutex m, if it is referred to only in MG-regions of m. We define mutex m to be *abstractable* for the system if it guards every shared variable referred to in any MG-region of m.

We introduce *mutex abstraction* as the elimination of an abstractable mutex m together with a corresponding coarsening of the grain of atomicity. More precisely, it is a transformation of the mutex program *Pm* into a program *Pa* without mutex m and with a coarser grain of atomicity, such that *Pm* is guaranteed to implement *Pa*, i.e., if *Pa* satisfies some specification, then *Pm* satisfies the same specification. Since it is cumbersome to give a general recipe for the transformation of command *Pm* when it contains a number of **wait** instructions, possibly within conditional statements and repetitions, we only treat a number of relevant examples. In each case, abstractability of m is assumed.

The simplest case is as follows. If command S in region (5) contains no **wait** instructions, then mutex abstraction replaces region (5) by $\langle S \rangle$. The correctness of this transformation is shown as follows. If a thread is executing S, it holds mutex m and, therefore, there is no concurrent execution of an MG-region of m. If T is any concurrent command that does not contain MG-regions of m, concurrent execution of S and T is equivalent to the sequential execution $(S;T)$ (as well as to $(T;S)$), since S and T refer to disjoint sets of variables. This proves the atomicity.

We now describe mutex abstraction for an abstractable mutex m in the case of an MG-region that contains one **wait** instruction. Abstraction transforms the MG-region into a combination of one or two atomic commands (one starting at the beginning and possibly one starting at the re-entry location). We describe the transformation only in a few special cases. In each case, we give the re-entry location $\mathtt{v}^{\#}$ and we use the label $\mathtt{m}^{\#}$ to indicate the first location after the MG-region. The command enclosed between $\mathtt{v}^{\#}$ and $\mathtt{m}^{\#}$ is called the re-entry stretch.

In all cases we use S, T, U to stand for arbitrary commands that contain no **lock**, **unlock**, or **wait** instructions. To give the flavor of it, we first describe the simple and not particularly useful case of an MG-region that only contains one unconditional **wait** instruction:

(6)
$$
\begin{array}{rl}
 & \mathbf{lock}(\mathtt{m}) \ ; \\
 & S \ ; \\
 & \mathbf{wait}(\mathtt{v},\mathtt{m}) \ ; \\
\mathtt{v}^{\#}\!: & T \ ; \\
 & \mathbf{unlock}(\mathtt{m})
\end{array}
$$

Mutex abstraction transforms region (6) into the sequence of two atomic commands

$$
\begin{array}{rl}
 & \langle \;\; S \ ; \ \mathit{enter}\ \mathtt{v} \ \rangle \ ; \\
\mathtt{v}^{\#}\!: & \langle \;\; T \quad\;\; \rangle \ ; \ \mathtt{m}^{\#} :
\end{array}
$$

More useful is the MG-region with *repetitive initial waiting*

(RIW)
$$
\begin{array}{rl}
 & \mathbf{lock}(\mathtt{m}) \ ; \\
\mathtt{v}^{\#}\!: & \mathbf{while} \;\; B \;\; \mathbf{do} \;\; \mathbf{wait}(\mathtt{v},\mathtt{m}) \;\; \mathbf{end} \ ; \\
 & S \ ; \\
 & \mathbf{unlock}(\mathtt{m})
\end{array}
$$

When the thread has been awakened, its next command is to retest the guard. We therefore place the re-entry label at **while**. Mutex abstraction transforms region (RIW) into the atomic command

(RIWt)
$$
\begin{array}{rl}
\mathtt{v}^{\#}\!: & \langle \;\; \mathbf{if} \;\; B \;\; \mathbf{then} \;\; \mathit{enter}\ \mathtt{v}\ \mathit{return\ at}\ \mathtt{v}^{\#} \\
 & \quad\;\; \mathbf{else} \;\; S \;\; \mathbf{end} \ \rangle \ ; \ \mathtt{m}^{\#} :
\end{array}
$$

where the phrase *enter* v *return at* $\mathtt{v}^{\#}$ expresses that the suspended thread resides at location $\mathtt{v}^{\#}$ and may awaken there. Indeed, a thread that tries to execute (RIW) while B holds places its identifier in v and starts waiting; when it is released it has to start all over again. On the other hand, if B is false, the thread executes S and thus concludes (RIW).

The next case is the MG-region with *conditional final waiting*:

(CFW)
$$
\begin{array}{rl}
 & \mathbf{lock}(\mathtt{m}) \ ; \\
 & S \ ; \\
 & \mathbf{if} \;\; B \;\; \mathbf{then} \;\; \mathbf{wait}(\mathtt{v},\mathtt{m}) \\
 & \mathbf{else} \;\; T \;\; \mathbf{end} \ ; \\
\mathtt{v}^{\#}\!: & \mathbf{unlock}(\mathtt{m})
\end{array}
$$

In this case, a re-entrant thread needs to re-acquire the mutex and can then immediately release it. After abstraction, the re-entry location $\mathtt{v}^{\#}$ coincides with $\mathtt{m}^{\#}$. Mutex abstraction transforms region (CFW) into the atomic command

(CFWt)
$$
\begin{array}{l}
\langle \;\; S \ ; \\
\quad\;\; \mathbf{if} \;\; B \;\; \mathbf{then} \;\; \mathit{enter}\ \mathtt{v} \\
\quad\;\; \mathbf{else} \;\; T \;\; \mathbf{end} \ \rangle \ ; \ \mathtt{v}^{\#} : \ \mathtt{m}^{\#} :
\end{array}
$$

The final example is a variation of repetitive initial waiting

(VIW) $\quad$ **lock**(m) ;
if $B0$ **then**
S ;
repeat **wait**(v,m)
v$^{\#}$: **until** $B1$;
T ;
end ;
U ;
unlock(m)

Mutex abstraction transforms region (VIW) into the combination of two atomic commands

(VIWt) $\quad$ ⟨ **if** $B0$ **then** S ; *enter* v
else U ; **goto** m$^{\#}$ **end** ⟩ ;
v$^{\#}$: ⟨ **if** $B1$ **then** T ; U
else *enter* v *return at* v$^{\#}$ **end** ⟩ ; m$^{\#}$:

It is clear that MG-regions of other forms, possibly with more than one **wait** instruction, can be treated analogously. If the MG-region contains W **wait** statements, it is transformed into at most $1 + W$ atomic commands, one for entry at the beginning and one for each re-entry location, but some of these may coincide or be empty.

When arguing about a program with mutexes we use mutex abstraction implicitly to coarsen the grain of atomicity. In other words, the transformation is not made explicit, but invariants are only required to hold when no thread owns the mutex. So, in the relevant states, all threads are outside MG-regions or at re-entry locations. In particular, threads in R(v) are at v$^{\#}$ and are waiting to acquire the mutex.

Remarks. (a) If an MG-region contains **wait**(v,m), **signal**(v), or **broadcast**(v), the waiting list v is a shared variable referred to in that region. Therefore, v is guarded by mutex m only if all commands **wait**(v,m), **signal**(v), and **broadcast**(v) are contained in MG-regions of m. It follows that, if **wait**(v,m) occurs in an MG-region of m, abstractability of m requires that **signal**(v) and **broadcast**(v) are also contained in MG-regions of m.

(b) A shared variable modified only by thread p is called an output variable of p. For an output variable of p to be guarded by m, it suffices that its modifications by p and its inspections by other threads are contained in MG-regions of m; inspections by p need not be guarded since p may keep a local copy.

4 Implementing semaphores

In this section, we illustrate the use of the pthread primitives and the theory of mutex abstraction by giving two implementations of semaphores.

A *semaphore* s is a shared integer variable that determines the number of times threads may execute the synchronization action P(s). So, a thread may execute $P(\mathtt{s})$ only if $\mathtt{s} > 0$. Whenever a thread executes $P(\mathtt{s})$, the value of $\mathtt{s}$ is decremented by 1. The other semaphore action is $V(\mathtt{s})$, which increments $\mathtt{s}$ by 1. The initial value of $\mathtt{s}$ is given at the point of declaration or initialization. The value of $\mathtt{s}$ cannot be modified or inspected by other means. Formally, we have that $P(\mathtt{s})$ and $V(\mathtt{s})$ are atomic commands given by

$$(7) \qquad \begin{array}{ll} P(\mathtt{s}): & \langle\ \mathbf{await}\ \ \mathtt{s} > 0\ \ \mathbf{then}\ \ \mathbf{dec}\ (\mathtt{s})\ \rangle\ ; \\ V(\mathtt{s}): & \langle\ \mathbf{inc}\ (\mathtt{s})\ \rangle\ . \end{array}$$

The semaphore thus described is the plain semaphore, cf. [2].

A slightly stronger primitive is the semaphore of [7] with a set `Q(s)` of waiting threads, which are enabled again in arbitrary order. Then $P(\mathtt{s})$ and $V(\mathtt{s})$ are given by

$$(8) \qquad \begin{array}{ll} P(\mathtt{s}): & \langle\ \mathbf{if}\ \ \mathtt{s} > 0\ \ \mathbf{then}\ \ \mathbf{dec}\ (\mathtt{s}) \\ & \quad \mathbf{else}\ \ \mathit{enter}\ \mathtt{Q(s)}\ \ \mathbf{end}\ \rangle\ ; \\ V(\mathtt{s}): & \langle\ \mathbf{if}\ \ \mathit{isEmpty}\ \ (\mathtt{Q(s)})\ \ \mathbf{then}\ \ \mathbf{inc}\ (\mathtt{s}) \\ & \quad \mathbf{else}\ \ \mathit{release\ one\ thread\ from}\ \mathtt{Q(s)}\ \ \mathbf{end}\ \rangle\ . \end{array}$$

As before, threads in `Q(s)` are disabled, and releasing a thread enables it again.

Remark. To see that the semaphore of (8) is stronger than the one of (7), consider two threads, both in a mutual exclusion loop of the form

$$\mathbf{loop}\ \ \mathit{NCS}\ ;\ P(\mathtt{s})\ ;\ \mathit{CS}\ ;\ V(\mathtt{s})\ \ \mathbf{end}$$

where *NCS* is a noncritical section and *CS* is a critical section. If one thread executes *CS*, while the other thread is waiting, the V action of the former enables the latter. In the case of (7), the first thread could repeatedly (even infinitely often) pass the semaphore, while the other thread is waiting indefinitely. This is not possible with (8).

4.1 Implementing a plain semaphore

We now show how the thread primitives of Section 2 can be used to implement the plain semaphore. For this purpose we introduce a record type with a mutex, a condition variable and a number:

$$\begin{array}{l} \mathbf{type}\ \ \mathit{Sem} = \mathbf{record} \\ \quad \mathtt{m} : \mathit{Mutex}\ ; \\ \quad \mathtt{v} : \mathit{ConditionVar}\ ; \\ \quad \mathtt{val} : \mathit{integer} \\ \mathbf{end}\ . \end{array}$$

The field `s.val` represents the value of the abstract semaphore `s`. The operation $P(\mathtt{s})$ is implemented by

$$
\begin{array}{l}
\quad\;\; \textbf{procedure } \mathit{PSem}\ (\textbf{var } \mathtt{s} : \mathit{Sem}) = \\
\qquad\quad \textbf{lock}\ (\mathtt{s.m})\ ; \\
\mathtt{s.v}^{\#}\!:\ \textbf{while}\ \ \mathtt{s.val} = 0\ \ \textbf{do} \\
\qquad\qquad \textbf{wait}\ (\mathtt{s.v}, \mathtt{s.m})\ \ \textbf{end}\ ; \\
\qquad\quad \textbf{dec}\ (\mathtt{s.val})\ ; \\
\qquad\quad \textbf{unlock}\ (\mathtt{s.m}) \\
\quad\;\; \textbf{end}\ .
\end{array}
$$

The body of *PSem* is an MG-region of mutex `s.m` with repetitive initial waiting (RIW) and with re-entry location $\mathtt{s.v}^{\#}$ at **while**. The operation $V(\mathtt{s})$ is implemented by

$$
\begin{array}{l}
\textbf{procedure } \mathit{VSem}\ (\textbf{var } \mathtt{s} : \mathit{Sem}) = \\
\quad \textbf{lock}\ (\mathtt{s.m})\ ; \\
\quad \textbf{inc}\ (\mathtt{s.val})\ ; \\
\quad \textbf{signal}\ (\mathtt{s.v})\ ; \\
\quad \textbf{unlock}\ (\mathtt{s.m}) \\
\textbf{end}\ \textbf{.}
\end{array}
$$

The only shared variables referred to in *PSem* and *VSem* are `s.val` and `s.v`. These variables only occur in the MG-regions of *PSem* and *VSem* and are therefore guarded by mutex `s.m`. Therefore, mutex `s.m` is abstractable and we may apply mutex abstraction.

Mutex abstraction of *PSem* yields one atomic command of the form (RIWt), which decrements `s.val` with 1 under precondition $\mathtt{s.val} \neq 0$; otherwise `s.val` is not changed. It follows that *PSem* preserves the invariant

(I0) $\qquad \mathtt{s.val} \geq 0\ .$

The body of *VSem* contains no **wait** instructions, so its abstraction is a single atomic command that sends a signal and increments `s.val`. It thus also preserves (I0).

This does not yet imply correctness of the implementation. We also have to prove that threads are not waiting unnecessarily. For this purpose, we prove that, whenever some thread is enlisted in `s.v` while $\mathtt{s.val} > 0$, there are threads ready to pass the semaphore, i.e. the set of re-entrant threads `R(s.v)` is nonempty. By contraposition, this is the same as the condition that, if `R(s.v)` is empty, $\mathtt{s.val} = 0$ or `s.v` is empty. It therefore suffices to prove that after mutex abstraction the system satisfies the invariant

(EWs) $\qquad \mathtt{R(s.v)} = \emptyset \ \ \Rightarrow\ \ \mathtt{s.val} = 0 \ \ \vee\ \ \mathtt{s.v} = \emptyset\ .$

In order to prove that (EWs) is an invariant, we strengthen it to the invariant

(I1) $\qquad \mathtt{s.val} \leq \#\mathtt{R(s.v)} \ \ \vee\ \ \mathtt{s.v} = \emptyset\ .$

Indeed, it is clear that (I1) implies (EWs). Invariance of (I1) is shown as follows. Whenever a thread enters `s.v`, we have $\mathtt{s.val} = 0$. Whenever `s.val` is incremented while `s.v` is nonempty, the set `R(s.v)` is extended by the signal in *VSem*. Whenever `R(s.v)` shrinks when `s.val` > 0, the field `s.val` is decremented.

One may notice that predicate (I1) is not an invariant in the setting without mutex abstraction: when a thread leaves `R(s.v)` while `s.val` > 0, it may momentarily falsify (I1) before it re-establishes (I1) by decrementing `s.val`.

In [6] p. 181, a semaphore implementation is presented that is essentially equivalent to this one; it only differs in that it uses an additional counter to avoid giving a signal when there is no waiting thread.

Notice that, if a thread performs a V operation while another thread is waiting, and then immediately performs a P operation, it may acquire the mutex first and thus pass the semaphore instead of the waiting thread. So, this indeed only implements the plain semaphore (7), not the semaphore with a waiting list (8).

4.2 Implementing a semaphore with waiting list

The stronger signal of (4) can be used to implement the stronger kind of semaphore (8). We use the same type *Sem* as before, but we now allow `s.val` to be negative. The operation $P(\mathtt{s})$ is implemented by

$$\begin{array}{rl}
 & \textbf{procedure } \mathit{PSema}\ (\textbf{var}\ \mathtt{s} : \mathit{Sem}) = \\
 & \quad \textbf{lock}\ (\mathtt{s.m})\ ; \\
 & \quad \textbf{dec}\ (\mathtt{s.val})\ ; \\
 & \quad \textbf{if}\ \mathtt{s.val} < 0\ \ \textbf{then wait}\ (\mathtt{s.v}, \mathtt{s.m})\ \textbf{end}\ ; \\
\mathtt{s.v}^{\#}\!: & \quad \textbf{unlock}\ (\mathtt{s.m}) \\
 & \textbf{end}\ .
\end{array}$$

This is a case of conditional final waiting (CFW) with the re-entry location at **unlock**. The value of semaphore (8) is represented by the expression

$$\sigma = (\mathtt{s.val})\ \textbf{max}\ 0\ .$$

We obviously have the invariant $\sigma \geq 0$. Moreover, a call of *PSema* with precondition $\sigma > 0$, or equivalently `s.val` > 0, decrements σ with 1 and terminates. A call of *PSema* with precondition $\sigma = 0$ leaves σ unchanged and leads to waiting. An execution of the re-entry stretch does not modify σ.

The operation $V(\mathtt{s})$ is implemented by

$$\begin{array}{l}
\textbf{procedure } \mathit{VSema}\ (\textbf{var}\ \mathtt{s} : \mathit{Sem}) = \\
\quad \textbf{lock}\ (\mathtt{s.m})\ ; \\
\quad \textbf{if}\ \mathtt{s.val} < 0\ \ \textbf{then strong-signal}\ (\mathtt{s.v})\ \textbf{end}\ ; \\
\quad \textbf{inc}\ (\mathtt{s.val})\ ; \\
\quad \textbf{unlock}\ (\mathtt{s.m}) \\
\textbf{end}\ .
\end{array}$$

One can easily verify that, again, mutex `s.m` is abstractable.

A call of *VSema* with precondition `s.val` ≥ 0 increments σ and a call with `s.val` < 0 leaves σ unchanged and releases one thread from the waiting list `s.v`.

To prove that this semaphore implements (8), we thus have to prove that `s.v` is empty if and only if $\texttt{s.val} \geq 0$. It therefore suffices to prove the invariant

$$\text{(J0)} \qquad \#\texttt{s.v} = (-\texttt{s.val}) \ \mathbf{max}\ 0 \ .$$

Preservation of (J0) is shown as follows. Every complete execution of *PSema* has precondition $\texttt{s.val} > 0$ and hence does not modify either side of (J0). An incomplete execution of *PSema* has precondition $\texttt{s.val} \leq 0$ and, hence, increments both sides of (J0) with 1. The re-entry stretch clearly preserves (J0). Execution of *VSema* with precondition $\texttt{s.val} \geq 0$ does not modify either side of (J0). Execution of *VSema* with precondition $\texttt{s.val} < 0$ preserves (J0) since `s.val` is incremented and the strong signal releases one thread from `s.v`, which is nonempty since (J0) holds in the precondition.

Note that it is incorrect to move the incrementation of `s.val` from *VSema* to the position directly after the call of **wait** in *PSema*, since then another call of *VSema* may pass unnoticed. Also, notice that, if the list of $s.v$ is a FIFO queue, the threads waiting at the semaphore are released in FIFO order.

5 Barrier synchronization

When a task is distributed over several threads, it is often the case that, at certain points in the computation, the threads must wait for all other threads before they can proceed with the next part of the computation. This is called *barrier synchronization*, cf. [1]. We model the problem by letting each thread execute the infinite loop

$$\mathbf{loop}\ \ \mathit{TNS}\ ;\ \mathit{Barrier}\ \ \mathbf{end}\ \ .$$

Here, *TNS* stands for a terminating noncritical section, a program fragment that necessarily terminates but that in all other aspects is irrelevant for the problem at hand.

The threads may only pass the barrier when all of them have completed the last noncritical section. On the other hand, when all of them have completed the last noncritical section, they all must pass the barrier and start the next noncritical section.

Thus, if we count the noncritical sections each thread has executed, the safety requirement is that the counters of the threads are always almost equal. In order to make this more precise, we provide each thread with a private ghost variable *cnt*, initially 0. The private variables *cnt* must not be modified by *TNS* and must be incremented by *Barrier* according to the Hoare triple

$$\text{(H)} \qquad \{\ \mathit{cnt.self} = C\ \}\ \ \mathit{Barrier}\ \{\ \mathit{cnt.self} = C+1\ \}\ \ .$$

One could propose to specify the barrier by the requirement that $\mathit{cnt}.q = \mathit{cnt}.r$ whenever both q and r are in *TNS*. This condition, however, would also be satisfied by disallowing different threads in *TNS* at the same time (i.e., by forcing mutual exclusion) and, then, no relationship between the counters of different threads would remain.

We therefore require the barrier condition that, for all threads q and r, (BC) $\quad q$ **in** $\mathit{TNS} \quad \Rightarrow \quad cnt.q \leq cnt.r$.

This invariant expresses that a thread that runs ahead must not execute *TNS* but wait. It is easy to see that (BC) implies that $cnt.q = cnt.r$ whenever both q and r are in *TNS*. Note that the ghost variables $cnt.q$ only serve in the formal specification, but that they have no role in the computation.

We now turn to a solution. Let N be the number of threads. It is clear that $N-1$ threads must wait at the barrier until the last one arrives. We therefore introduce a shared variable `atbar` for the number of threads waiting at the barrier. Clearly, `atbar` $= 0$ initially. We introduce a mutex `m` and a condition variable `v` to enforce waiting. The last thread to arrive at the barrier resets `atbar` to 0 and releases all waiting threads. We thus get

(9)
```
        loop
            TNS ;
            lock (m) ;
            inc (cnt) ; inc (atbar) ;
            if atbar = N  then
                atbar := 0 ;
                broadcast (v) ;
            else  wait (v,m)  end  ;
    v#:     unlock (m)
        end .
```

The MG-region represents the barrier. This is a case of conditional final waiting with re-entry at **unlock**. We have swapped the two branches of (CFW) for the sake of lay-out. It is clear that the MG-region satisfies Hoare triple (H). Invariant validity of (BC) is less obvious. To verify this, we first observe that the only shared variables are `atbar` and the waiting list `v` and that they are both guarded by mutex `m`. It follows that `m` is abstractable, cf. section 3. We can therefore apply mutex abstraction which transforms the barrier into a single atomic command of the form (CFWt).

Since a thread in *TNS* is not at the re-entry location $\mathtt{v}^{\#}$, predicate (BC) easily follows from (WL) together with the invariant

(K0) $\quad q \notin \mathtt{v} \quad \Rightarrow \quad cnt.q \leq cnt.r$.

Predicate (K0) is threatened by the incrementation of $cnt.q$ when q does not enter `v` and by the broadcast that releases all threads from `v`. It is therefore threatened only by a thread p that executes the MG-region under the precondition `atbar` $= N-1$. We now need the following invariants

(K1) $\quad \mathtt{atbar} = \#\mathtt{v}$;
(K2) $\quad q \in \mathtt{v} \quad \Rightarrow \quad cnt.q = (\mathtt{MIN}\ r :: cnt.r) + 1$.

Indeed, this implies that when thread p starts the MG-region with precondition `atbar` $= N-1$, all other threads are in `v` and have $cnt.q = cnt.p + 1$. Execution of the MG-region therefore makes all values *cnt* equal and thus validates the consequent of (K0).

Predicate (K1) is an invariant, since an acting thread either enters `v` and increments `atbar`, or makes `v` empty and `atbar` equal to 0. Since the broadcast makes `v` empty, predicate (K2) is threatened only when the acting thread starts waiting. In that case, (K1) implies the existence of some other threads outside `v`. Now (K0) implies that all threads outside `v` have the minimal value for *cnt*. Therefore, when thread q enters `v`, it validates the consequent of (K2) for q and preserves the value of the minimum. This shows that (K2) is an invariant.

Note that the invariants are no invariants in the setting without mutex abstraction. This is most obvious for (K1), since the modifications of `atbar` are distinct from the modifications of `v`.

Remark. Barriers are often needed to allow a single thread, say p_0, to perform a certain action, say *Snap*, like taking a snapshot of part of the global state, after which action all threads may proceed again. In that case, one would need to give all threads programs of the form

(10) **loop**
  TNS ;
  Barrier ;
  if $\mathit{self} = p_0$ **then** *Snap* **end** ;
  Barrier
end .

Usually, however, there is a better option, since most often the choice of thread p_0 is irrelevant. In that case, one can decide to let *Snap* be executed by the last thread at the barrier. This means that command *Snap* is included in the **then** branch of program (9), say after the assignment `atbar` := 0. In this way, a second *Barrier* as in (10) is superfluous. This possibility was proposed by Arnold Meijster.

6 Passing the baton

In our favorite design method for concurrent algorithms, at some point the threads are synchronized by means of atomic synchronization statements that have either of two forms:

$$\langle\ S(i)\ \rangle \quad \text{or} \quad \langle\ \textbf{await}\ B(i)\ \textbf{then}\ S(i)\ \rangle\ .$$

If nothing is known about the way in which the guards $B(\_)$ are truthified, the only conceivable implementations are forms of busy waiting, i.e., repeated guard evaluation without triggers from the environment. In general this entails much unproductive activity. We now assume that all statements that may modify the variables in the guards $B(\_)$ are among the statements $S(\_)$ mentioned. In other words, the guards $B(\_)$ are only affected by the statements $S(\_)$. It follows that a thread that finds a guard $B(\_)$ false, can start waiting, provided that every thread that executes some statement $S(\_)$ awakens sufficiently many waiting threads.

Andrews [1], p. 199, presents a method to implement such a system of related **await** statements by means of a split binary semaphore. The method is called "the technique of passing the baton". In this section, we present a similar method based on pthread primitives. As was pointed out to us by W.H.J. Feijen, this may be regarded as an implementation of a conditional critical region (CCR), e.g., cf. [1].

For the sake of uniformity, we regard the bare commands $\langle\, S(i) \,\rangle$ as **await** statements with guards *true*. We thus restrict our attention to a system of atomic synchronization statements of the form

(11) $\langle$ **await** $B(i)$ **then** $S(i)$ $\rangle$,

where some of the guards $B(\_)$ may be identically *true*.

Remark. We are mainly interested in cases where index i ranges over a small set, say with K elements, and where the number of threads is (much) bigger than K. For example, in a traffic simulation, the threads may represent cars that have to pass a crossing. Cars from the same direction are allowed to cross concurrently, but cars from different directions must wait for each other. We can use a guard $B(i)$ to indicate that direction i is allowed to cross. Then we need one additional atomic statement without waiting condition for cars that leave the crossing. In this way, the crossing is implemented by a family of related **await** statements where K is one plus the number of directions.

For scheduling purposes, we additionally provide a function *count* such that *count* (i) is a lower bound of the number of threads about to execute the **await** statement with parameter i. This function may be called in the commands $S(\_)$, but must not occur in the guards $B(\_)$.

The easiest implementation is to let a thread that executes some $S(\_)$ awaken all waiting threads. In this solution, we need one mutex and one condition variable, say `m` and `v`. The family of commands (11) is implemented by

(12)
```
procedure SynchAct (i) =
    lock (m) ;
    while ¬B(i) do wait (v,m) end ;
    S(i) ;   broadcast (v) ;
    unlock (m)
end .
```

Since **broadcast**(`v`) awakens all threads waiting in `v`, this solution may require much communication and superfluous testing. We therefore prefer to let the testing be done by the thread that has modified the state.

For this purpose, we first use the signal according to (3) of section 2 . We use one mutex `m` as before. Let us assume that the indices i range from 0 to $K-1$. The idea of the solution is to associate a condition variable to each of the guards and to signal it when the guard is truthified. We thus introduce an array `v` of condition variables of length K and a shared array `cnt` of integers of length K. We use an additional boolean shared variable `busy` to give priority to the threads that are currently being awakened. We postulate the initial values $\mathtt{cnt}[i] = 0$ and $\mathtt{busy} = \mathit{false}$.

Function *count* is implemented by $\mathit{count}(i) = \mathtt{cnt}[i]$. Indeed, we use $\mathtt{cnt}[i]$ for the number of threads waiting at index i. The family of commands (11) is implemented by

(13)
```
        procedure SynchAct (i) =
            lock (m) ;
            if busy ∨ ¬B(i) then
                cnt[i] ++ ;
                repeat wait (v[i],m)
        v[i]#:  until B(i) ;
                cnt[i] --
            end ;
            S(i) ;
            if some j ∈ [0 . . K − 1]
            has cnt[j] > 0 ∧ B(j) then
                busy := true ;
                signal (v[j])
            else busy := false end ;
            unlock (m)
        end .
```

The whole procedure body is one MG-region of the type (VIW) with re-entry location at the **until**. We can therefore apply mutex abstraction and transform the MG-region into a combination of two atomic commands as in (VIWt).

Safety of the solution requires that, whenever a thread executes $S(k)$, it does so with precondition $B(k)$. This follows from the observation that every thread that executes $S(k)$ first tests $B(k)$ in the same atomic command and that the possible modification of `cnt` in between does not modify $B(k)$.

In the second **if** statement, which is called the *junction*, an arbitrary index j is chosen for which $\mathtt{cnt}[j] > 0$ and $B(j)$ holds, if one exists. In other cases, no signal needs to be given. Of course, in practical applications, the programmer may refine (remove) the nondeterminacy by imposing a preferred order for the search of j. The nondeterminacy here serves to provide maximal applicability.

The progress property we want to prove is that no thread is ever waiting unnecessarily. This will be proved in Section 6.1 below.

In general, the test of $B(i)$ at label $\mathtt{v}[i]^{\#}$ cannot be omitted. For example, consider two threads that call (13) with the same argument i. Assume that $B(i)$ is false so that both threads start waiting. Then some other thread makes $B(i)$ true and signals $\mathtt{v}[i]$. Since the signal is weak, it may release both threads from $\mathtt{v}[i]$. One of these threads executes $S(i)$ and may in this way falsify $B(i)$. Therefore the other thread must retest guard $B(i)$.

As the scenario of the previous paragraph suggests, there are circumstances where retesting is superfluous. Indeed, if the signal in (13) is replaced by a strong signal, the **repeat until** clause can be simplified to **wait**(v[i], m). The correctness of this simplification is proved in Section 6.2 below.

If there is always at most one thread waiting in v[j], there is no difference between the signal and the strong signal. So, also in that case, the **repeat until** clause can be simplified to **wait**(v[i], m). This is applicable e.g. when the indices i are thread identifiers and thread i is the only one to execute $\langle$**await** $B(i)$ **then** $S(i)\rangle$.

Of course, the keyword **repeat** and the clause **until** $B(i)$ can also be omitted if guard $B(i)$ is identically true. Waiting is still required to give priority to the awakened thread(s).

Remarks. (a) If some application has other commands that have to use the variables that occur in $B(k)$ and $S(k)$ under mutual exclusion, but that do not modify the guards $B(k)$, then such a command can be simply enclosed by a **lock** and **unlock** of the mutex.

(b) The junction can sometimes be simplified. For example, if command $S(i0)$ has the postconditions cnt[$i1$] > 0 and $B(i1)$, it is possible to choose $j := i1$. There are also cases where the search for j had better follow a specific order for fairness considerations.

6.1 Verification of progress

We assume that the number of threads is bounded. We cannot prove progress unconditionally, since family (11), which we are implementing, is not guaranteed to make progress. The progress property that we shall prove is therefore phrased conditionally. In fact, we prove that, whenever a thread is waiting unnecessarily, within a bounded number of steps some waiting thread will complete its call of (13).

In order to prove this, we must first formalize what it means to be waiting unnecessarily. After mutex abstraction, we need only consider states where all threads are outside the MG-region or at a re-entry location v[k]$^{\#}$. In the former case, the thread is said to be outside. In the latter case, the thread is said to *wait at* index k. A thread is said to be *waiting unnecessarily* if it waits at some index k while $B(k)$ holds.

Since we introduced mutex m, array v with its waiting lists v[k], and the boolean busy to implement (11), it is clear that these variables do not occur in the guards $B(i)$ and the commands $S(i)$. We use this below implicitly.

It is easy to see that cnt[k] is the number of threads waiting at k, as formalized in the invariant

$$\text{(L0)} \qquad \mathtt{cnt}[k] \;=\; (\#\, q :: q \ \mathbf{at}\ \mathtt{v}[k]^{\#})\ .$$

The invariance of (L0) follows by inspection of the code of *SynchAct*. Indeed, when a thread increments cnt[k], it arrives at v[k]$^{\#}$. When a thread leaves v[k]$^{\#}$, it decrements cnt[k]. Moreover, cnt is not modified by other commands.

We claim that busy holds whenever a thread is waiting at some index while its guard holds; this is expressed in

(L1) $\quad q$ **at** $\texttt{v}[k]^{\#} \;\wedge\; B(k) \;\Rightarrow\; \texttt{busy}$.

Invariance of (L1) is seen as follows. Thread q can arrive at $\texttt{v}[k]^{\#}$ only when `busy` or $\neg B(k)$ holds. If a thread, say p, truthifies $B(k)$ in $S(\_)$ while q is at $\texttt{v}[k]^{\#}$, the next action of p is to truthify `busy` as well, because of (L0). Also because of (L0), no thread falsifies `busy` while the antecedent of (L1) holds.

According to (WL), a thread q waits at index k if and only if it is a member of the waiting list $\texttt{v}[k]$ or of the set of ready threads $\texttt{R}(\texttt{v}[k])$. Let us write $\texttt{R}[k]$ for $\texttt{R}(\texttt{v}[k])$.

We claim that, when `busy` holds, there is a thread ready to acquire the mutex while its guard also holds:

(L2) $\quad \texttt{busy} \;\Rightarrow\; (\exists\, k :: B(k) \;\wedge\; \texttt{R}[k] \neq \emptyset)$.

The invariance of (L2) is shown as follows. Whenever a thread executes the junction, it either sets `busy` to false or it sends a signal to $\texttt{v}[k]$ for some k while $\texttt{cnt}[k] > 0$ and $B(k)$ holds. In the former case, (L2) is established. In the latter case, it follows from (WL) and (L0) that $\texttt{v}[k]$ or $\texttt{R}[k]$ is nonempty so that, afterwards, $\texttt{R}[k]$ is nonempty. So, again, (L2) is established. Incomplete executions of the MG-region do not affect (L2).

As announced above, we finally show that, whenever a thread is waiting unnecessarily, within a bounded number of steps some waiting thread completes its call of *SynchAct*. Let WU be the set of indices where some thread is waiting unnecessarily. We want to prove that, if WU is nonempty, within a bounded number of steps $S(k)$ is executed for some index $k \in WU$.

So, assume that WU is nonempty. Then (L1) implies that `busy` holds. Since `busy` holds, outside threads can only enter some waiting queue. Any thread q waiting at some $k \notin WU$, is in $\texttt{v}[k]$ or in $\texttt{R}[k]$, but the only thing it can do is to enter $\texttt{v}[k]$. This implies that WU does not change before $S(k)$ is executed for some index $k \in WU$.

On the other hand, since `busy` holds, (L2) implies the existence of an index k such that $B(k)$ holds and that $\texttt{R}[k]$ is nonempty. Any such index k is an element of WU. Since the total number of threads is bounded, it follows that, after a bounded number of steps, some thread $q \in \texttt{R}[k]$ for some $k \in WU$ executes $S(k)$.

6.2 The simplification for strong signals

In this section we assume that the **signal** in (13) is replaced by a **strong-signal**. We then prove that the key word **repeat** and the clause **until** $B(i)$ can be removed. For this purpose it suffices to show that, for program (13) with a **strong-signal**, we have the invariant that, for every ready thread at the re-entry location, the waiting guard holds:

(M0) $\quad \texttt{R}[k] \neq \emptyset \;\Rightarrow\; B(k)$.

Predicate (M0) can only be falsified when another thread enables a thread q in some waiting list or falsifies $B(k)$. When another thread releases q from list

$\mathtt{v}[k]$, it does so only if $B(k)$ holds. Predicate (M0) is therefore preserved under releases. In order to show that (M0) is preserved under modification of $B(k)$, we postulate

(M1) $\quad \mathtt{R}[k] \neq \emptyset \quad \Rightarrow \quad \mathtt{busy}$,
(M2) $\quad q \in \mathtt{R}[k] \quad \wedge \quad r \in \mathtt{R}[m] \quad \Rightarrow \quad q = r$.

Indeed, $B(k)$ can only be modified by execution of $S(\_)$. Predicate (M1) implies that no outside thread can enter while $\mathtt{R}[k]$ is nonempty. Predicate (M2) implies that there is at most one ready thread waiting. It follows that, if a thread executes $S(\_)$, it was the only ready waiting thread. Therefore, afterwards, $\mathtt{R}[k]$ is empty.

Preservation of (M1) and (M2) is shown as follows. The **strong signal** always makes at most one waiting thread ready. It is only given when `busy` is set to true. Once some waiting thread q is ready, the predicates (M1) and (M2) imply that thread q is the only thread that can execute the junction, in which case q leaves $\mathtt{R}[k]$, while possibly inducing another thread to enter it.

One may note that the uniqueness of the inside thread implies that the decrementation of `cnt[_]` can be moved to the position immediately after the call of the strong signal.

By replacing the signal in (13) by a strong signal and removing the key word **repeat** and the clause **until** $B(i)$, one avoids that guard $B(i)$ is tested twice for every enabling. This makes the program more efficient. On the other hand, if the signals in (13) are indeed weak in the sense that they often release more than one waiting thread, it is likely that waiting threads are released that must enter again after testing the guard. The weak signals, therefore, may lead to unproductive context switches. This also hampers performance.

7 Conclusion

We have developed a concept of mutex abstraction to ease the formal treatment of programs for pthreads with mutexes and condition variables. It is based on the ideas of structured code locking and structured data locking, well-known among pthread programmers. Our contributions are the formalization in the form of mutex guarded regions, the formal introduction of waiting queues and re-entry locations, mutex abstraction as a form of program transformation, and its use in rigorous correctness proofs. For the latter purpose, we developed a number of example programs, which, apart from the plain semaphore, seem to be new.

References

[1] Andrews, G.R.: Concurrent Programming, principles and practice. Addison-Wesley 1991.

[2] Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer V. 1991.

[3] Bacon, J.: Concurrent systems. Addison Wesley Longman Ltd 1998. ISBN 0-201-17767-6.

[4] E.W. Dijkstra: Co-operating sequential processes. In: F. Genuys (ed.): Programming Languages (NATO Advanced Study Institute). Academic Press, London etc. 1968, pp. 43–112.

[5] Harbison, S.P.: Modula–3, Prentice Hall 1992

[6] Kleiman, S., Shah, D., Smaalders, B.: Programming with Threads. SunSoft Press (Prentice Hall 1996), ISBN 0-13-172389-8.

[7] Martin, A.J., Snepscheut, J.L.A. van de: Design of synchronization algorithms. In: M. Broy (ed.): Constructive Methods in Computing Science. Springer V. 1989, pp. 445–478.