Concurrent Systems

Nebenläufige Systeme

VIII. Monitor

Wolfgang Schröder-Preikschat

November 27, 2014

Outline

Preface

Fundamentals
  Mutual Exclusion
  Condition Variable
  Signalling Semantics

Implementation
  Data Structures
  Use Case
  Operations

Summary

Subject Matter

- discussion on abstract concepts as to “a shared variable and the set of meaningful operations on it” [7, p. 121]:
  - monitor
    - a language notation, initially denoted by critical region [6, 7]
    - associates a set of procedures with a shared variable
    - enables a compiler to:
      - ensure that only these procedures are carried out on that variable
      - check that the respective operations exclude each other in time
  - condition
    - one or more special variables that do “not have any stored value accessible to the program” [12, p. 550]
    - used to indicate and control a particular wait mode
    - for the respective process inside the monitor

- in functional terms, get to know “monitor” as fundamental means of synchronisation independent of linguistic features
  - explanation of various styles: Hansen, Hoare, Concurrent Pascal, Mesa
  - according to this, schematic representation of implementation variants
  - demonstrate basic functions of a fictitious (language) run-time system
Motivation

Semaphore Considered Harmful

for all advantages, semaphores are to be approached with caution:
- too low level, programmers must keep track of all calls to P and V
- although different, used for both uni- and multilateral synchronisation
- out of it, various design and languages concepts originated:
  - idea for structuring control of sharing [5, p. 135–136]
  - mutual exclusive use of a shared variable [6]
  - a shared variable associated with an event queue [6]
  - synchronisation rules within type definitions [2]
  - class-like synchronised data type [7, 12, 14]
  - inspired by SIMULA 67 [4, 3]
  - first implemented in Concurrent Pascal [9]
  - comes in a characteristic of many kinds [1, 10]
- however, the concept is beyond a programming-language construct
  - it is fundamental for system programming and system-level operation

Hint (Monitor [7, p. 121])
The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy.

Class Concept Expanded by Coordination

key aspect is to facilitate solely indirect access to shared variables by means of monitor procedures
- by definition, these procedures have to execute by mutual exclusion
  - on behalf of the calling process, the procedure prologue applies for exclusive occupation of the monitor \( \leadsto \text{lockout} \) simultaneous processes
  - on behalf of the occupying process, at return the procedure epilogue releases the monitor again \( \leadsto \text{proceed} \) locked processes, if any
- usually, a compiler is in charge of ejecting the procedure pro- and epilogue
  - only infinite loops or hardware failures may prevent epilogue execution
  - only constructs beyond the frame of reference may force abnormality\(^1\)
- in logical respect, deadlocks due to programmed absence of unblocking of critical sections are impossible
- accordingly, instructions for synchronisation are cross-cutting concern of the monitor and no longer of the whole non-sequential program
- particularly, instructions to protect critical sections are not made explicit
  - given that foreign-language synchronisation primitives cannot be used\(^1\)

\(^1\)Thinking of a multi-language system.
Signalling Conventions

- explicit signal operation assumed, *signal-and-φ*, with φ indicating the behaviour of the signalling process as follows:
  - **wait**: join monitor entrance queue and leave the monitor
    - resume all signalled processes (one at a time)
  - **urgent wait**: join preferential queue and leave the monitor
    - resume one signalled process (first come, first served)
    - re-enter the monitor, enjoy priority over entrant processes
  - **return**: leave the monitor and resume the single signalled process
  - **continue**: carry on holding the monitor, keep inside the procedure
    - resume all signalled processes (one at a time) at return
  - **automatic**: leave the monitor and re-evaluate waiting conditions
    - if so, resume no longer waiting processes (one at a time)

- in case of absence of a signal primitive, signalling may still happen:
  - **automatic**: leave the monitor and re-evaluate waiting conditions
    - if so, resume no longer waiting processes (one at a time)

-A main issue is the control transfer between signaller and signallee.

Waiting inside a monitor

Without leaving the monitor, another process is unable to signal.

---

Fundamental Data Types

```c
typedef struct monitor {
    semaphore_t mutex; /* initial {1} */
    #ifdef __FAME_MONITOR_SIGNAL_URGENT_WAIT__
    lineup_t urgent; /* urgent waiting signallers */
    #endif
} monitor_t;
```

Data type used for keeping track of waiting processes (cf. p.18):

```c
typedef struct lineup {
    int count; /* number of waiting processes */
    event_t crowd; /* wait-for event */
} lineup_t;
```
Monitor Procedures

1 extern void lockout (monitor_t *); /* enter monitor */
2 extern void proceed (monitor_t *); /* leave monitor */
3 extern void watch (condition_t *); /* wait on signal */
4 extern void spark (condition_t *); /* signal condition */

Consider these operations an additional run-time system element for a compiler of a “concurrent C-like” programming language.

Calls to lockout and proceed will be automatically generated as part of the pro- and epilogue of the respective monitor procedure.

Similarly, calls to watch and spark will be generated for the corresponding applications of condition variables.

In addition, instances of type monitor and condition will be automatically ejected, too, by the code generation process of such a compiler.

Further improvements [12, p. 551] are imaginable to also better reflect the different signalling semantics.

Consolidating Example I

Bounded Buffer

A bounded buffer is controlled by a pair of condition variables:

```c
#include "monitor.h"
#define BUF_SIZE 80
typedef struct buffer {
  condition_t space;  /* control of reusables */
  condition_t data;   /* control of consumables */
  char store[BUF_SIZE]; /* reusable resource */
  unsigned in, out;   /* store housekeeping */
  unsigned count;      /* wait/signal condition */
} buffer_t;
```

Instantiation of the necessary monitor and condition variables:

```c
static monitor_t storehouse = {1}; /* monitor is free */
static buffer_t buffer = {
  {& storehouse }, {& storehouse } /* link to monitor */
};
```

Consolidating Example II

Bounded-Buffer Fill

Handmade monitor procedure to put one item into the buffer:

```c
void put (char item) {
  lockout (& storehouse ); /* procedure prologue */
  {
    while (buffer.count == BUF_SIZE)
      watch (& buffer.space);
    buffer.store [buffer.in ] = item;
    buffer.in = (buffer.in + 1) % BUF_SIZE;
    buffer.count += 1;
    spark (& buffer.data);
  }
  proceed (& storehouse ); /* procedure epilogue */
}
```

2–3 • monitor entrance, usually to be generated by a compiler
4–11 • body of monitor procedure, to be programmed by a human
12–13 • monitor exit, usually to be generated by a compiler

Consolidating Example III

Bounded-Buffer Empty

Handmade monitor procedure to get one item out of the buffer:

```c
char get () {
  char item;
  lockout (& storehouse ); /* procedure prologue */
  {
    while (buffer.count == 0) watch (& buffer.data);
    item = buffer.store[buffer.out ];
    buffer.out = (buffer.out + 1) % BUF_SIZE;
    buffer.count -= 1;
    spark (& buffer.space);
  }
  proceed (& storehouse ); /* procedure epilogue */
  return item;
}
```
control of unblocking

---

**Event Queue**

- A classic monitor implementation on event queue basis is considered:

  ```
  typedef struct event { } event_t;
  ```

  - **catch** makes the process unsusceptible against lost wakeup:
    - i) non-effective in case of cooperative scheduling, otherwise
    - ii) inhibits preemption or dispatching (SMP), resp., or
    - iii) notifies event sensibility to potential signalers (cause)
  - **coast** if the process was not yet detected by cause, blocks on the event
  - otherwise, clears the catch state and keeps the process running
  - **await** blocks the process on the specified event (i.e., signalled by cause)
  - **cause** unblocks processes (tentatively) waiting on the specified event
  - based on this abstraction, waitlist operations can be composed next

**Waitlist Operations I**

```c
inline int avail(lineup_t *this)
{
    if (this->count > 0) /* any delayed? */
        cause(&this->crowd); /* yes, unblock */
    return this->count;
}
```

```c
inline void brace(lineup_t *this)
{
    this->count ++; /* one more delaying */
    void (*evoke)(lineup_t *this) {
        int count = this->count; /* save state */
        if (count > 0) /* any delayed? */
            admit(elect(&this->crowd)); /* yes, seize CPU */
        return count;
}
```

**Waitlist Operations II**

```c
inline void catch(event_t*);
inline void await(event_t*);
inline void cause(event_t*);
```

- **catch** marks the process susceptible against lost wakeup:
  - i) non-effective in case of cooperative scheduling, otherwise
  - ii) inhibits preemption or dispatching (SMP), resp., or
  - iii) notifies event sensibility to potential signalers (cause)

- **coast** ensures that a process in running state is detectable by cause

- **await** if the process was not yet detected by cause, blocks on the event

- otherwise, clears the catch state and keeps the process running

- **cause** blocks the process on the specified event (i.e., signalled by cause)

- **cause** unblocks processes (tentatively) waiting on the specified event

- based on this abstraction, waitlist operations can be composed next

**Signalling Semantics**

- As has been foreshadowed by a configuration option (cf. p.12):
  - signal and continue Mesa-style
  - signal and return signal and urgent Mesa-style
  - Hansen-style as to Concurrent Pascal [8, 9]
  - Hansen-style as originally proposed [7]

- Some reflect improvements as proposed by Hoare [12, p.551, 1–4.]

- Starting point was the strict approach of signal and urgent wait monitor

- Here, the discussion is in the order as to increasing complexity/overhead

- As indicated by the data type (cf. p.12), the designs presented next
  are typical for an approach using event queues

- Note that signalling is non-effective if no process is waiting on it (cf. p.8)

- This excludes the use of semaphores, as V leaves a signal trace

  - V always has an effect: at least it increases the semaphore value

- Lightweight and efficient monitor operation benefits from cross-layer optimisation in constructive means

  - From language- to system-level run-time system to operating system

---

© wosch CS (WS 2014, LEC 8) Implementation – Operations 17–33

© wosch CS (WS 2014, LEC 8) Implementation – Operations 18–33
### Signal and Continue

```c
void lockout(monitor_t *this) { P(&this->mutex); }
void proceed(monitor_t *this) { V(&this->mutex); }
void watch(condition_t *this) {
  brace(&this->event);    /* prepare to release */
  proceed(&this->guard);  /* release monitor */
  shift(&this->event);    /* release processor */
}
void spark(condition_t *this) {
  avail(&this->event);    /* try signal process */
}
```

- as `watch` needs to release the monitor before releasing the processor, a potential race condition must be prevented.
- `brace` notifies upcoming blocking of the current process to the system.
- this is to assure the current process of progress guarantee as soon as the monitor was released and another process is enabled to `spark` a signal.

### Signal and Return

```c
void lockout(monitor_t *this) { P(&this->mutex); }
void proceed(monitor_t *this) { V(&this->mutex); }
void watch(condition_t *this) {
  brace(&this->event);    /* prepare to release */
  proceed(&this->guard);  /* release monitor */
  shift(&this->event);    /* release processor */
}
void spark(condition_t *this) {
  if (!avail(&this->event)) /* no watcher waiting? */
    proceed(&this->guard);  /* release monitor */
}
```

- calling `spark` must be the final action within a monitor procedure.
- similar to the `continue` statement of Concurrent Pascal [9, p.205]
- otherwise, the signaller could proceed inside an unlocked monitor if no signallee was detected.

### Signal and Wait

#### Combined Monitor Waitlist

```c
void lockout(monitor_t *this) { P(&this->mutex); }
void proceed(monitor_t *this) { V(&this->mutex); }
void watch(condition_t *this) {
  brace(&this->event);    /* prepare to release */
  proceed(&this->guard);  /* release monitor */
  shift(&this->event);    /* release processor */
}
void spark(condition_t *this) {
  if (evoke(&this->event)) /* signallee done! */
    lockout(this->guard);  /* re-enter monitor */
}
```

- as the case may be, the signaller blocks on a condition variable.
- in case of a pending signallee, the signaller interrupts execution.
  - a process switch inside the looked monitor takes place (cf. p.19)
  - in the further course, another process unlocks/releases the monitor.
- accordingly, the signaller must make sure to **relock** the monitor.

### Signal and Urgent Wait I

#### Monitor Entrance/Exit

```c
void lockout(monitor_t *this) { P(&this->mutex); }
void proceed(monitor_t *this) {
  if (!avail(&this->urgent)) /* no urgent waiting */
    V(&this->mutex);    /* release monitor */
}
```

- in contrast to the solutions discussed before, **exit** from the monitor needs to check two waitlists for pending processes:
  - the re-entrance waitlist (**urgent**), but only in case of urgent processes
  - the entrance waitlist (**mutex**), else
- by definition, urgent processes interrupted own operation in favour of processes pending for **event** handling.
- urgent processes caused events, recently, and want be resumed, expressly.
- indicator of urgent waiting processes is a counter by means of which the number of process blockings is registered.

© wosch  CS (WS 2014, LEC 8)  Implementation – Operations 21–33
void watch(condition_t *this) {
    brace(&this->event);  /* prepare to release */
    proceed(this->guard);  /* release monitor */
    shift(&this->event);  /* release processor */
}

void spark(condition_t *this) {
    if (avail(&this->event))  /* watcher waiting? */
        defer(&this->guard->urgent);  /* urgent wait */
}

as the case may be, spark makes the current process urgent waiting

a preferential queue (Ger. Vorzugswarteschlange) is used to this end

defer results in a process switch from line 9 to line 4, back and forth
from spark to shift, out of watch, and back to spark at monitor exit

urgent waiting processes keep proceed off from unlocking the monitor
when the monitor owner returns or blocks, an urgent process resumes
as a consequence, the monitor should not be protected by a mutex

in linguistic terms, a monitor is a language notation for a critical
region and one or more associated shared variables

a shared class [7, p.226–232], inspired by SIMULA 67 [3]
linked with event queues [6] or condition variables [12], resp.
differentiated by several signalling semantics and conventions [13]

in operating-system terms, a monitor is a means of control of the
scheduling of resources among interacting processes

mutual-exclusive use of non-preemptable reusable resources
coordinated use of consumable resources according to a causal chain

in system-programming terms, a monitor can be readily implemented
by a binary semaphore and event queues

note that a mutex is to be rejected for the signal and wait variants

Hansen
In practice, monitors would, of course, be implemented by
uninterruptible operations in assembly language. [11, p.31]
Reference List I


Forschungsbericht

In: Communications of the ACM 9 (1966), Sept., Nr. 9, S. 671–678

In: Acta Informatica 1 (1971), S. 115–138

Reference List II

In: Communications of the ACM 15 (1972), Jul., Nr. 7, S. 574–578


Reference List III


Reference List IV


In: C++ Report 11 (1999), Sept., Nr. 9, S. 1–9


In: Lehrstuhl INFORMATIK 4 (Hrsg.): Concurrent Systems. FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 3
handmade monitor procedures are prone to absence of unblocking the
monitor before return: `proceed` is missing or will never be executed

- object constructors/destructors find a remedy [16, p. 220, Sec. 6.1.4]

```cpp
class atomic {
  static monitor_t sluice;
  public:
    atomic() { lockout(&sluice); );
    ~atomic() { proceed(&sluice); );
};
```

- exit from the scope of an `atomic` instance implicitly performs `proceed`:

```cpp
int64_t inc64(int64_t *i) {
  atomic inc; return *i + 1;
}
```

- a technique that is also known as the scoped locking pattern [15]