Concurrent Systems

Nebenläufige Systeme

VIII. Monitor

Wolfgang Schröder-Preikschat

— Selbststudium —

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
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:
  - i. check that only these procedures are carried out on that variable
  - ii. ensure 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

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]

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]

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 ~ lockout simultaneous processes
    - on behalf of the occupying process, at return the procedure epilogue releases the monitor again ~ 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

1Thinking of a multi-language system.
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 $\sim$ lockout simultaneous processes
  - on behalf of the occupying process, at return the procedure epilogue releases the monitor again $\sim$ 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
- in logical respect, deadlocks due to programmed absence of unblocking of critical sections are impossible

1Thinking of a multi-language system.

### Intentional Process Delay

- multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

Hansen  
- proposed to attach a shared variable to an event [6, p.577]  
  - with cause and await as intrinsic functions for event signalling
multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

- proposed a non-attached **condition variable** [12, p. 550]
  - with **wait** and **signal** as intrinsic functions for condition handling

in operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition
- sticking point is how the event queue is being acted upon:
  - all processes can be transferred to the monitor waitlist (**cause**)
  - suggests that the former take priority over the latter [7, p. 118]
**Intentional Process Delay**

- Multilateral (blocking) synchronisation is implicit basis of a monitor, but **unilateral synchronisation** needs to be made explicit

**Hansen**
- proposed to attach a shared variable to an event [6, p. 577]
  - with **cause** and **await** as intrinsic functions for event signalling

**Hoare**
- proposed a non-attached **condition variable** [12, p. 550]
  - with **wait** and **signal** as intrinsic functions for condition handling

- In operating-system terms, per variable an **event queue** of processes waiting by reason of a certain condition
- Sticking point is how the event queue is being acted upon:
  - **Hansen**
    - **all processes** can be transferred to the monitor waitlist (**cause**)
      - suggests that the former take priority over the latter [7, p. 118]
      - remodels his idea to a **single-process waitlist** [8, 9]: all \(\equiv one\)
  - **Hoare**
    - **exactly one** out of the waiting processes is selected (**signal**)
      - decrees that the chosen one is immediately resumed [12, p. 550]
      - but signalling is non-effective (void) if no process would be waiting on it

**Signalling Conventions**

cf. [13]

- Explicit signal operation assumed
Signalling Conventions  

- explicit signal operation assumed, \textit{signal-and-ϕ}, with ϕ indicating the behaviour of the signalling process as follows:
  - **wait**
    - join monitor \textit{entrance queue} and leave the monitor
    - resume all signalled processes (one at a time)
    - re-enter the monitor, compete against all processes
  - **urgent wait**
    - join \textit{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
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)
  - re-enter the monitor, compete against all processes
  - 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)

  - **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
    - carry on holding the monitor, keep inside the procedure
    - resume all signalled processes (one at a time) at return

  - **return**: leave the monitor and resume the single signalled process
    - if applicable, the order of process resumption is undefined
    - if so, resume no longer waiting processes (one at a time)

  - **continue**: carry on holding the monitor, keep inside the procedure
    - resume all signalled processes (one at a time)
    - re-enter the monitor, enjoy priority over entrant processes

- 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.

Signalling Conventions

- 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

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

Atomicity of Control Transfer

- consequence for the **ownership structure** of monitor and signaller:
  - **change**: signal and wait, urgent wait, or return
  - **keep**: signal and continue or automatic signalling
Atomicity of Control Transfer

- consequence for the **ownership structure** of monitor and signaller:
  - **change**
    - signal and wait, urgent wait, or return
  - **keep**
    - signal and continue or automatic signalling
- with an **indivisible change** in ownership a signallee has guarantee on the still effective invalidation of its waiting condition

- **urgent wait**
  - exactly for the single signalled process
  - by definition, the process to be resumed is predetermined
  - no other process can re-establish the waiting condition
  - makes re-evaluation of the waiting condition unnecessary
    - if (!condition), wait: intolerant to false signalisation
- **return**
  - ditto

...
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;

typedef struct condition {
    monitor_t *guard; /* enclosing monitor */
    lineup_t event; /* signal - awaiting processes */
} condition_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

Functional Interface

```c
extern void lockout(monitor_t*); /* enter monitor */
extern void proceed(monitor_t*); /* leave monitor */
extern void watch(condition_t*); /* wait on signal */
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.
Monitor Procedures  

**Functional Interface**

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:

1. `#include "monitor.h"
2. `
3. `#define BUF_SIZE 80`
4. `
5. `typedef struct buffer {
6.   condition_t space; /* control of reusables */
7.   condition_t data; /* control of consumables */
8.   char store[BUF_SIZE]; /* reusable resource */
9.   unsigned in, out; /* store housekeeping */
10.  unsigned count; /* wait/signal condition */
11. } buffer_t;`

Instantiation of the necessary monitor and condition variables:

1. `static monitor_t storehouse = {1}; /* monitor is free */`
2. `static buffer_t buffer = { /* actual buffer */
3.   {&storehouse}, {&storehouse} /* link to monitor */
4. };`

Consolidating Example II  

**Bounded-Buffer Fill**

Handmade monitor procedure to put one item into the buffer:

1. `void put(char item) {
2.   lockout(&storehouse); /* procedure prologue */
3.   {
4.     while (buffer.count == BUF_SIZE)
5.       watch(&buffer.space);
6.     buffer.store[buffer.in] = item;
7.     buffer.in = (buffer.in + 1) % BUF_SIZE;
8.     buffer.count += 1;
9.   }
10.  spark(&buffer.data);
11. } proceed(&storehouse); /* procedure epilogue */
12. }

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;
}
```

- monitor entrance and exit and body of monitor procedure as before

Event Queue

Classical Technique for Monitor Implementation

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

```c
typedef struct event { } event_t;;
extern void catch(event_t*); /* expect event */
extern int coast(); /* wait for event */
extern int await(event_t*); /* catch & coast */
extern int cause(event_t*); /* signal event */
```

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

© wosch CS (— Selbststudium —) Implementation–Use Case 16

© wosch CS (— Selbststudium —) Implementation–Operations 17

© wosch CS (— Selbststudium —) Implementation–Operations 17
Event Queue

Classical Technique for Monitor Implementation

- a classic monitor implementation on **event queue** basis is considered:

```c
typedef struct event { } event_t;

extern void catch(event_t*); /* expect event */
extern int coast(); /* wait for event */
extern int await(event_t*); /* catch & coast */
extern int cause(event_t*); /* signal event */
```

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

**await** blocks the process on the specified event (i.e., signalled by **cause**) based on this abstraction, **waitlist operations** can be composed next.

---

Waitlist Operations I

Control of Blocking

```c
inline void brace(lineup_t *this) {
  this->count++; /* one more delaying */
  catch(&this->crowd); /* ready to block/continue */
}

inline void shift(lineup_t *this) {
  coast(); /* conditionally block */
  this->count--; /* one less delaying */
}

inline void defer(lineup_t *this) {
  this->count++; /* one more delaying */
  await(&this->crowd); /* unconditionally block */
  this->count--; /* one less delaying */
}

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

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

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

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

note that *evoke* forces a process switch within a still locked monitor
- as the case may be, the resuming process then unlocks the monitor
- consequently, the monitor should not be protected by a *mutex* object

- as has been foreshadowed by a *configuration option* (cf. p. 12):
  - Mesa-style [14]
  - Hansen-style as to Concurrent Pascal [8, 9]
  - Hansen-style as originally proposed [7]
  - Hoare-style [12]
Signalling Semantics

as has been foreshadowed by a **configuration option** (cf. p. 12):
- signal and continue  
  - Mesa-style [14]
- signal and return  
  - Hansen-style as to Concurrent Pascal [8, 9]
- signal and wait  
  - Hansen-style as originally proposed [7]
- signal and urgent wait  
  - Hoare-style [12]

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

---

**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 */
}
```
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) {
    if (!avail(&this->event)) /* no watcher waiting? */
        proceed(this->guard); /* release monitor */
}
```

- 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

1. `void lockout(monitor_t *this) { P(&this->mutex); }
2. void proceed(monitor_t *this) { V(&this->mutex); }
3. void watch(condition_t *this) {
   brac(&this->event); /* prepare to release */
   proceed(this->guard); /* release monitor */
   shift(&this->event); /* release processor */
}
4. 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

1. `void lockout(monitor_t *this) { P(&this->mutex); }
2. void proceed(monitor_t *this) { /* no urgent waiting */
   V(&this->mutex); /* release monitor */
}
3. void watch(condition_t *this) {
   if (!avail(&this->urgent)) /* no urgent waiting */
   V(&this->mutex); /* release monitor */
}
4. void spark(condition_t *this) {
   if (evoke(&this->event)) /* signallee done! */
   lockout(this->guard); /* re-enter 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

Signal and Urgent Wait I

Monitor Entrance/Exit

1. `void lockout(monitor_t *this) { P(&this->mutex); }
2. void proceed(monitor_t *this) { /* no urgent waiting */
   V(&this->mutex); /* release monitor */
}
3. void watch(condition_t *this) {
   if (!avail(&this->urgent)) /* no urgent waiting */
   V(&this->mutex); /* release monitor */
}
4. void spark(condition_t *this) {
   if (evoke(&this->event)) /* signallee done! */
   lockout(this->guard); /* re-enter 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:

i. the re-entrance waitlist (urgent), but only in case of urgent processes
ii. 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

Signal and Urgent Wait II

Condition Handling

```c
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

i. a **preferential queue** (Ger. Vorzugswarteschlange) is used to this end
ii. **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
Signal and Urgent Wait II

Condition Handling

```c
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

Process States and State Transitions
cf. [17, p.27]
Résumé

- 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]
Résumé

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


Reference List II

Monitor Entry/Exit Revisited

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]

```
1 class atomic {
2     static monitor_t sluice;
3 public:
4     atomic() { lockout(&sluice); }; // lockout
5     ~atomic() { proceed(&sluice); }; // proceed
6 };
```

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

```
1 int64_t inc64(int64_t *i) {
2     atomic inc; return *i + 1;
3 }
```

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