Concurrent Systems
Nebenläufige Systeme

X. Guarded Sections

Wolfgang Schröder-Preikschat

December 11, 2014

Agenda
Preface
Hardware Events
  Fundamentals
  Sequencing
  Implementation
Process Events
  Fundamentals
  Sequencing
  Implementation
Summary

Outline
Preface
Hardware Events
  Fundamentals
  Sequencing
  Implementation
Process Events
  Fundamentals
  Sequencing
  Implementation
Summary

Subject Matter
- discussion on abstract concepts as to structural measures suited in paving the way for non-blocking synchronisation
  - guarded sections
  - synchronise process-originated events
  - pre-/postlude sections
  - synchronise hardware-originated events
- both approaches common is the fact that processes of whichever kind will never be blocked at entrance to a critical section
  - however their requests to enter and pass through may be delayed
  - an alternating sequencer takes care of retroactive request processing
  - this constrains overlapping and, thus, eases non-blocking request queues
    - per sample of interrupt-transparent synchronisation [13], for instance
- similar to an explicit (“eventual values” [8, 9]) or implicit future [2], it is shown how to deal with “direct-result critical sections”
  - using concepts such as the promise [6] or promise pipelining [11]
  - functional programming meets distributed computing for synchronisation
- one learns that guarded sections largely resemble conventional critical sections, but with a much more relaxed execution model

1 Not to be confused with “guarded commands” [4].
Interrupt Handling

**Definition (Interrupt)**

Mechanism of a (soft- or hardware) processor to prompt software to draw attention to an external process asynchronously, unpredictably, and unreproducibly.

- a **sudden upcall** (acc. [3]) performed by a processor in the middle of or between actions, depending on the processor model
- start of a simultaneous process on this very processor in **stacking** mode
- most notably, this process is characteristic of a **run-to-completion** flow
- as to operating systems, usually a **trinity** of problem-specific routines is to be considered—and assumed in the following:
  - **guardian**: interrupt-handler dispatcher running at CPU priority
  - **prelude**: first-level interrupt handler (FLIH) running at CPU/OS priority
  - **postlude**: second-level interrupt handler (SLIH) running at OS priority
- what all have in common is the **asynchronism** to the current process that was interrupted and will be delayed by their particular actions

Responsibility Assignment

**Hint (Interrupt Latency)**

In order to make **loss of interrupts** improbable, CPU priority\(^a\) must be cancelled and OS priority\(^b\) must be taken in minimum time.

\(^a\)Interrupt requests of the same and lower priority are disabled.
\(^b\)All interrupt requests are enabled.

- conceptually, prelude and postlude together constitute the interrupt handler to be dispatched due to an **interrupt request** (IRQ):
  - **guardian**: in case of an **edge-triggered** IRQ, takes OS priority before it identifies and activates the prelude for the given IRQ
  - in case of a **level-triggered** IRQ, takes OS priority afterwards
  - operates and “unloads” the device to satisfy the IRQ source
  - starts immediately if enabled by the CPU priority
  - as the case may be, releases its postlude for post-processing
- **prelude**: risk of race conditions and system-stack overflow
- **postlude**: risk of race conditions → **synchronisation** or **reentrancy**

Relevance of Postlude

**Hint (Asynchronous System Trap, AST [10, p. 414])**

On the VAX, a software-initiated interrupt to a service routine. ASTs enable a process to be notified of the occurrence of a specific event asynchronously with respect to its execution. In 4.3 BSD, ASTs are used to initiate process rescheduling.

- essentially, the interrupt handler postlude equates to such an AST
  - a mechanism that forces an interrupted process back into system mode:
    - i. when no interrupt handler prelude is pending (i.e., stacked) and
    - ii. in the moment when the interrupt handler guardian terminates (i.e., returns)
  - as if this very process performs a system call to the interrupt postlude
- caution is advised when an **interrupt-handler control flow** expands
  - **guardian**: not applicable, controls prelude and postlude (i.e., an AST)
  - **prelude**: risk of race conditions and system-stack overflow
  - **postlude**: risk of race conditions ~ synchronous reentrancy

- purpose of the postlude is to safely allow such control-flow expansions
  - its activation is controlled similar to the control of guarded sections
Execution Sequencing of Postludes

- heading for postlude execution depends on the particular prelude
  - a prelude is a function, its return value indicates the postlude to be run
  - a return value of NULL indicates that this prelude asks for no postlude
- according to the model, an interrupt indeed causes a new process but not a new process instance
  - the guardian is such a process, it operates in the name of the interrupted process instance and commands no own context
  - same applies for the sequencer, it is an optional guardian continuation and takes care for safe postlude processing

Guardian and Sequencer

- From FLIH to SLIH (cf. p. 38ff.)

```
__attribute__((fastcall)) void guardian(long irq) {
  static usher_t *tube = { 0, (0, &tube.load.head) ];
  extern remit_t *((flih[])(usher_t *));
  remit_t =task;
  #ifdef __FAME_INTERRUPT_EDGE_TRIGGERED__
  pivot(&tube.busy, +1); admit(irq); /* take OS priority */
  #endif
  if (task != 0) deter(&tube, task);
  ret(0) /* prevent lost unload */
  while (tube.load.head.link != 0) { /* forward pending postludes */
    admit(irq);
    flush(&tube);
    /* forward postlude */
    pivot(&tube.busy, -1); /* leave critical section */
  }

  /* prevent lost unload */
  while (tube.load.head.link != 0) { /* prevent unload */
    task = (*flih[irq])(&tube); /* activate prelude & satisfy IRQ source */
    if (task != 0) deter(&tube, task);
    /* activate prelude & satisfy IRQ source */
    ret(0) /* prevent lost unload */
    /* settle insertion point */
    if ((task != 0) && (tube.load.head.link == 0)) remit(task);
    pivot(&tube.busy, +1); admit(irq); /* take OS priority */
  }

  /* prevent lost unload */
  while (tube.load.head.link != 0) { /* forward pending postludes */
    admit(irq);
    flush(&tube);
    /* forward pending postludes */
    pivot(&tube.busy, -1); /* leave critical section */
  }
```

Overlapping Pattern

- not unlike the guarded section as to process events described below (cf. p. 20), but with the following fundamental differences:
  - simultaneous requests to run through a guarded section occur stack-wise
  - processing start as to delayed (i.e., pending) passage requests is AST-like
  - postludes are still carried out asynchronously to the interrupted process
  - notably is the implication in terms of the constructive restriction of overlappings as to simultaneous pre- and postludes
  - higher priority preludes may overlap lower priority preludes
  - preludes may overlap postludes, but never reverse
  - postludes may overlap other postludes and process instances
- regarding the whole processing chain and the involvement of guardian and sequencer process one may realise:
  - the guardian (incl. prelude) enqueues postludes possibly simultaneously, but never dequeues them
  - the sequencer dequeues postludes possibly overlapped by enqueues, but these dequeues will never overlap enqueues performed by the guardian
- this multiple-enqueue/single-dequeue mode of operation eases the design of a non-blocking synchronised postlude queue

Lock-Free Synchronised Enqueue

- assuming that simultaneous enqueues can happen only in a stacking arrangement, then the following is “thread safe”:

  ```
  void chart_ms_lfs(queue_t *this, chain_t *item) {
    chain_t *last;
    item->link = 0; /* terminate chain: FIFO */
    last = this->tail; /* settle insertion point */
    this->tail = item; /* create new partial list */
    while (last->link != 0) /* overlapping enqueue! */
      last = last->link; /* find end of orig. list */
    last->link = item; /* insert & combine lists */
  }
  ```

- idea is to create a new partial list using an atomic store and, thus, isolate the original list for later safe manipulation
  - but simultaneous enqueues may then shift the actual insertion point
Lock-Free Synchronised Dequeue cf. [13]

```c
chain_t * fetch_ms_lfs(queue_t * this) {
    chain_t * item;
    if ((item = this->head.link) /* next item fetched */
        && !(this->head.link = item->link)) {
        this->tail = &this->head; /* is last one, reset */
        if (item->link != 0) { /* overlapping enq.! */
            chain_t * help, * lost = item->link;
            do {
                help = lost->link; /* remember next & */
                chart_ms_lfs(this, lost); /* rearrange */
            } while ((lost = help) != 0);
        }
    }
    return item;
}
```

**Hint (Lock Freedom)**

Some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. [7, p.142]

---

Wait-Free Solution Special Instructions

```c
void chart_ms_wfs(queue_t *this, chain_t *item) {
    chain_t * last;
    item->link = 0; /* terminate chain: FIFO */
    last = FAS(&this->tail, item);
    last->link = item; /* eventually append item */
}

chain_t * fetch_ms_wfs(queue_t * this) {
    chain_t * item = this->head.link;
    if (item) { /* check for last item */
        if (item->link) /* is not, non-critical */
            this->head.link = item->link;
        else if (CAS(&this->tail, item, &this->head))
            CAS(&this->head.link, item, 0);
    }
    return item;
}
```

**Hint (Lock Freedom)**

Some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. [7, p.142]

---

Recapitulation

- in the **pre-/postlude model**, sequencer becomes that process in the context of which interrupt handling is carried out
- more precisely, the process at the bottom of an interrupt-handler stack
- put differently, the interrupted process that “activated” the guard (p. 9)

**Hint (Pro-/Epilogue [14, 13])**

At first glance, interrupt handler pre-/postludes seemingly resemble the pro-/epilogue model. While this is quite true for preludes, it does not hold for postludes. Epilogue execution is a **synchronous event** as to the interrupted kernel-level process, in contrast to postludes.

- postlude guide through is not unlike **procedure chaining** [12, p.10], a technique to serialize execution of conflicting threads
- differences are due to the constrained pre-/postlude overlapping pattern
- unless stack-based scheduling [1], any process overlapping is assumed
- this similarity gives reason to think about a **generalisation** of the pre-/postlude model to synchronise **process-instance** events

Outline

- Preface
- Hardware Events
  - Fundamentals
  - Sequencing
  - Implementation
- Process Events
  - Fundamentals
  - Sequencing
  - Implementation
- Summary
Handling of a Critical-Section Function

Conditioned Fire-and-Forget Pattern

Execution Sequencing of Critical Sections

in structural respect not unlike conventional critical sections, but as to its flow model very different and non-blocking

```c
future tobe;    /* value container & promise */
guarded(tobe) *task = {    /* future as parameter */
    /* compute promised value 'item', part 1 */
    if (!phase(ewd)) break;    /* conditional synchr. */
    /* compute promised value 'item', part 2 */
    task->tobe.prove(item);    /* fulfil promise */
}
tobe.exact();    /* await fulfilment of promise */
```

key aspect is that a process never blocks incoming a guarded section, but its request to pass through that section may be delayed

fall back on known linguistic concepts in order to pattern a solution for the above-mentioned problem:

- **future** the promise to deliver a value at some later point in time [2]
  - read-only placeholder object created for a not yet existing result
  - the result is computed concurrently and can be later collected
- **promise** traced back to [6], a writeable, single-assignment container
  - can be used to successfully complete a future with a value

each future instance has a dedicated resolver taking care of (a) value assignment and (b) promise states:

- **kept** value computed, assignment took place
- **broken** computation aborted, assignment ceases to take place
- **pending** process in progress, assignment did not just yet take place

based on these states, a process is able to synchronise on the event that the promise to deliver a value was either kept or broken

- the resolver (process inside the critical section) acts as producer
- the future using process acts as consumer  signaling semaphore

heading for a critical section depending on the state of occupancy:

- **unoccupied** guard grants requester access to the critical section
  - the critical section becomes occupied by the requester
- **occupied** guard denies requester access to the critical section
  - the request gets queued and the requester bypasses

leaving a critical section depending on the request-queue state:

- **empty** critical section becomes unoccupied, the process continues
  - the actual leaving process becomes sequencer and re-enters the critical section for each queued request
- **full**
Synchronisation of Direct-Result Critical Sections

- A passage request may refer to a multi-elementary future object:
  - i. a promise indicator (kept, broken, pending)
  - ii. a placeholder of problem-specific type as to the critical section
  - iii. a binary semaphore that is used in producer/consumer mode
    - i.e., a signalling semaphore applicable by different processes
- In case of a direct-result critical section, the sequencer takes the part of a resolver that also have to signal the “kept” or “broken” state
  - V does the signalling and by means of P the signal can be consumed

Run-to-Stopover for Peer Processes

- Critical sections controlled by processes waiting on events caused by external processes can be handled straightforwardly
  - As the external process, in order to making progress, does not depend on any internal process or state of any critical section
  - Thus, interaction between external and internal processes is non-critical
- Unlike internal processes, provided that they have to interact with their peers using shared resources inside a critical section
  - Relevant at this point is the producer/consumer style of interaction, only
    - If the consumer needs to wait on the producer inside a critical section
    - Then the critical section must be unoccupied by the consumer while waiting
  - Other “critical interaction” is implicit subject matter of any critical section
- As a consequence, precautions must be taken for interacting internal processes—similar to signalling inside monitors [15, p. 9]
  - Without clearing the guarded section, a stopover process may deadlock

Execution Characteristics of the Critical Section

- Critical sections controlled by processes in a run-to-completion style can be handled straightforwardly

Definition (Run to Completion (Process))

A potentially preemptive process free from self-induced wait states as to the possible non-availability of reusable or consumable resources.

- Processes will not await external events from inside the critical section
- Control of a run-to-stopover style of execution of a critical section depends on the locality of peer processes:

Definition (Run to Stopover (Process))

A potentially preemptive process possibly being subject to wait states.

- Processes waiting on events caused by an external process (e.g., I/O)
- Processes interacting with an internal process due to resource sharing
- Both styles of execution concern the period of a critical section, only
  - But at large, a process may be classified run to completion and stopover

Overlapping Pattern

- Notably is the implication in terms of the constructive restriction of overlappings as to simultaneous requester and sequencer processes
  - I. Requesters of any guarded section may overlap each other
  - II. Self-overlapping of a sequencer is impossible
  - III. Only sequencers of different guarded sections may overlap each other
- Regarding the whole request processing chain and the involvement of requester and sequencer process one may realise:
  - Multiple requester may enqueue passage requests possibly simultaneously, but they will never dequeue these
  - A single sequencer only dequeues passage requests, but this may happen simultaneously to enqueues of one or more requesters
- This multiple-enqueue/single-dequeue mode of operation eases the design of a non-blocking synchronised passage-request queue
  - Furthermore, synchronisation then happens to be even wait-free [5]

Hint (Wait Freedom)

Any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. [7, p. 124]
Data Type I

typedef struct guard {
  bool busy; /* state: initial false */
  queue_t load; /* pending passage requests */
  indicator_t * hint; /* sequencer blocked-on event */
} guard_t;

locking (clasp) and unlocking (loose) of a guarded section:
inline bool clasp (guard_t * this) {
  return this -> busy || TAS(& this -> busy);
}
inline void loose (guard_t * this) {
  this -> busy = false;
}

clasp is a test-and-test-and-set (TATAS) to mitigate bus lock bursts
note that the TAS-part should be mapped to CAS or LL/SC, resp.
- the former or a CISC- and the latter for a RISC-type of processor

Claiming and Clearing

first-come, any serve (FCAS)
inline bool vouch (guard_t * this, order_t * work) {
  bool busy = clasp (this);
  if (busy)
    chart(&this->load, &work->next);
  return busy == false;
}
inline order_t * clear (guard_t * this) {
  order_t * next;
  if ((next = (order_t *) fetch(&this->load)) == 0)
    loose (this);
  return next;
}
clearing, as shown here, is prone to the lost update problem:
- the process just leaving the critical section finds no pending passage request and will loose the guarded section next
- but before the guarded sections is loosened, a simultaneous process comes in and attempts claiming it
- another passing request is generated—and may get lost forever

Waiting

Watch for External Processes

using an indicator for the "art of waiting" (p.42) of the sequencer
inline bool phase (guard_t * this, indicator_t * hint) {
  enroll (hint); /* expect external event */
  loose (this); /* leave guarded section */
  repose (hint); /* receive external event */
  return clasp (this); /* retry to enter... */
}
- the sequencer indicates interest in receiving an external signal,
- leaves the guarded section—but pending requests still may exist,
- will block (if signal is pending) or continue (if signal occurred),
- and, finally, reapplies for passing through the guarded section
- further action is context-dependent and better up to the phase caller

Trailing Conditional Wait

Watch for Ex-/Internal Processes

inline void trail (guard_t * this, indicator_t * hint) {
  enroll (this->hint = hint);
}
inline order_t * clear (guard_t * this) {
  order_t * next;
  if ((next = (order_t *) fetch(&this->load)) == 0) {
    indicator_t * hint;
    if ((hint = this->hint)) this->hint = 0;
    loose (this);
    if (hint) repose (hint); /* block or continue */
  }
  return next;
}
deadlock hazard due to potential of a lost update (cf. p.26):
- fetch and reset wait indicator, as an indivisible "guarded" operation
- if applicable, await event—that, however, possibly fails to appear
- if reapplication fails, the current process is no longer sequencer
- during the waiting period, another process entered the guarded section
- thus, the current process must leave the guarded section or synchronise
- as requests may be and remain still pending, waiting on respective internal processes is problematic → trailing conditional wait
typedef struct order {
    chain_t next;  /* passage-request chaining */
    item_t post;   /* argument placeholder */
} order_t;

typedef union item {
    long (*lump)[];    /* argument vector (N > 1) */
    long sole;         /* single argument (N = 1) */
} item_t;

depending on the number of parameters, the structure describes a multi- or uni-element argument vector

in the multi-element case, the argument vector is placed adjacent to its item or order, resp., instance (cf. p.43)

in addition, this vector also serves as placeholder for a future value

fore editing of passage-request parameters, optional:

order_t *task = order(2);  /* two parameters */
(*task->post.lump)[0] = (long)index;
(*task->post.lump)[1] = value;

entry protocol, agreement on the sequencer process:

extern guard_t gate;
if (vouch(&gate, task)) do  /* enter section */

midsection (i.e., actual critical section) flow control, optional:

extern indicator_t hint = { 0 };  /* initial: block */
trail(&gate, &hint);
/* block in clear */
/* alternatively or combined, depending on what suits */
if (!phase(&gate, &hint)) break;  /* block in-place */

exit protocol, processing of pending passage requests:

while ((task = clear(&gate)));
/* leave section */

besides logical synchronisation in the midsection, any other programming statements are doable as well—like in conventional critical sections

guarding of critical sections at operating-system as well as instruction set architecture level and in a non-blocking manner

processes are never delayed at entrance of an already occupied critical section, however their requests to pass through

not unlike procedure chaining, but also supporting in-line functions

at both levels, overlappings as to simultaneous processes result in a multiple-enqueue/single-dequeue model of request handling

the sequencer will be the only process being in charge of dequeuing

that is, the continuation of a requester (lev. 3) or the guardian (lev. 2)⁴

whereby this continuation is commander-in-chief of a critical section

when a requester process requires a direct result from the sequencer process, interaction in a consumer/producer-style takes place

in such a case, the respective request is associated with a future object

it carries the promise of the sequencer to deliver a result to the requester

a future-specific signalling semaphore then indicates result availability

besides supporting conventional critical sections, this approach eases design of non-blocking synchronised non-sequential programs

⁴Operating-sytem machine or instruction set architecture level, respectively.
Reference List I


Reference List II


Reference List III


Reference List IV


Reference List V

In: Lehrstuhl Informatik 4 (Hrsg.): Concurrent Systems. FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 8

Guardian Insulating and Invoking

```plaintext
Joint:

```pushl %ecx  # save volatile register
movl $0, %ecx  # pass IRQ number
JointN:

```pushl %edx  # save another volatile register
```pushl %eax  # ditto
```call _guardian  # fastcall to guardian
```popl %eax  # restore volatile register
```popl %edx  # ditto
```popl %ecx  # ditto
```iret  # resume interrupted process

each IRQ entry in the CPU exception vector is associated with a joint

```Joint42:

```pushl %ecx  # save volatile register
movl $42, %ecx  # pass IRQ number
```jmp _jointN  # switch to common joint section...

```

Simple Interrupt Handler

- First-level interrupt handler (FLIH), at CPU/OS priority (p.11, l.7)
- Second-level interrupt handler (SLIH), at OS priority (p.11, l.7/13)

```c
remit_t * prelude(usher_t *tube) {
    static remit_t task = { {}, postlude);
    /* Come here for device pre-processing &
    * device-related IRQ acknowledgement. */
    deter(tube, &task); /* force postlude to queue */
    return 0; /* don't request shortcut */
}
```

without l.5, postlude shortcut (p.11, l.20) goes with return &task

```c
void postlude(order_t *todo) {
    /* Come here for device post-processing &
    * any asynchronous system interaction. */
    V((semaphore_t *)todo->post.sole);
}
```

system interaction means: to vouch for guarded sections (cf. p.30)

```c
inline remit_t * untie(usher_t *tube) {
    return (remit_t *) fetch(&tube->load);
}
inline void flush(usher_t *tube) {
    remit_t *next;
    do if ((next = untie(tube))) remit(next);
    while (next != 0);
}
```

with queue synchronisation style: #define __FAME_SYNC_ITS__ resulting in "{chart,fetch}_ms_lfs" or "_wfs", resp.

Interrupt-Handler Guard

- A doorman (Ger. Pförtner) for guarded sections at the low level of handling asynchronous program interrupts, a specialised guard:

```c
typedef guard_t usher_t;
inline void deter(usher_t *tube, remit_t *task) {
    chart(&tube->load, &task->data.next);
}
inline remit_t * untie(usher_t *tube) {
    return (remit_t *) fetch(&tube->load);
}
inline void flush(usher_t *tube) {
    remit_t *next;
    do if ((next = untie(tube))) remit(next);
    while (next != 0);
}
```
Job Definition and Start

- a SLIH or an interrupt-handler postlude, resp., is a passage request (cf. p. 29) attended by a procedure address
- that is to say, a request object with implicit processing method

```c
typedef struct remit {
  order_t data;       /* parameter set */
  void (*code)(order_t *);  /* procedure address */
} remit_t;

inline void remit(remit_t *this) {
  (*this->code)(&this->data); /* run that job */
}
```

- at process-event level, this structure specifies different parameterised critical sections associated with the same guarded section
- it allows for procedure chaining similar to that of Synthesis [12, p. 10]

Art of Waiting

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- note that a semaphore has memory semantics with regard to signals
- thus, awaiting a signal by means of $P$ once a sequencer process released the guarded section is free of the lost-wakeup problem

- another option is falling back on the event queue [15, p. 17]:
  - just if one wants to implement $P$ and $V$ as a guarded section, for example

```c
typedef event_t indicator_t;
inline void enroll(indicator_t *hint) { catch(hint); }
inline void repose(indicator_t *hint) { coast(); }
inline void arouse(indicator_t *hint) { cause(hint); }
```

Guarded-Section Sample

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- note that a semaphore has memory semantics with regard to signals
- thus, awaiting a signal by means of $P$ once a sequencer process released the guarded section is free of the lost-wakeup problem

- another option is falling back on the event queue [15, p. 17]:
  - just if one wants to implement $P$ and $V$ as a guarded section, for example

```c
typedef event_t indicator_t;
inline void enroll(indicator_t *hint) { catch(hint); }
inline void repose(indicator_t *hint) { coast(); }
inline void arouse(indicator_t *hint) { cause(hint); }
```

Order Allocation/Deallocation

- in order to decrease latency and lower overhead, specialisation towards the use of an order pool is recommended

```c
inline order_t *order(unsigned long n) {
  order_t *item;
  if (n < 2)
    item = (order_t *)malloc(sizeof(order_t));
  else {
    item = (order_t *)
      malloc(sizeof(order_t) + n * sizeof(long));
    if (item)
      item->post.lump = (void *)
        ((long)item + sizeof(*item));
  }
  return item;
}
```

- in order to decrease latency and lower overhead, specialisation towards the use of an order pool is recommended

```c
inline void ditch(order_t *item) {
  free(item);
}
```

Fictitious Alarm-Message Handling

- straightforward is the use of a signalling semaphore:

```c
typedef semaphore_t indicator_t;
inline void enroll(indicator_t *hint) { }
inline void repose(indicator_t *hint) { P(hint); }
inline void arouse(indicator_t *hint) { V(hint); }
```

- note that a semaphore has memory semantics with regard to signals
- thus, awaiting a signal by means of $P$ once a sequencer process released the guarded section is free of the lost-wakeup problem

- another option is falling back on the event queue [15, p. 17]:
  - just if one wants to implement $P$ and $V$ as a guarded section, for example

```c
typedef event_t indicator_t;
inline void enroll(indicator_t *hint) { catch(hint); }
inline void repose(indicator_t *hint) { coast(); }
inline void arouse(indicator_t *hint) { cause(hint); }
```

5A binary semaphore used in a producer/consumer style of interaction.
typedef struct future {
    promise_t data; /* prospective value */
    indicator_t gate; /* signalling element */
} future_t; /* prospective value */

A future object is the promise—of a guarded section, here—to deliver a result at some later point in time:

typedef enum status {
    PENDING, KEPT, BROKEN
} status_t;

typedef struct promise {
    status_t bond; /* processing state */
    item_t item; /* future-value placeholder */
} promise_t; /* prospective value */

whereby the promise is a result placeholder, on the one hand, and keeps track of the status of result delivery, on the other hand.

**Simple Future Implementation**

```c
inline status_t probe(future_t *this) {
    return this->data.bond;
}

inline void trust(future_t *this) { enroll(&this->gate); }

inline item_t *exact(future_t *this) {
    repose(&this->gate);
    return probe(this) == KEPT ? &this->data.item : 0;
}

inline void bring(future_t *this, status_t bond) {
    this->data.bond = bond;
    arouse(&this->gate);
}

inline void prove(future_t *this, item_t *item) {
    this->data.item = *item;
    bring(this, KEPT);
}

inline void abort(future_t *this) { bring(this, BROKEN); }
```