Concurrent Systems

Nebenläufige Systeme

XI. Guarded Sections

Wolfgang Schröder-Preikschat

January 29, 2020

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[1]
pre-/postlude sections	synchronise hardware-originated events

[1] Not to be confused with "guarded commands" [4].

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[1]

  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

[1] Not to be confused with "guarded commands" [4].

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[1]

  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* [14], for instance

[1] Not to be confused with "guarded commands" [4].

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[1]

  pre-/postlude sections
  - synchronise hardware-originated events

- similar to an explicit ("eventual values" [9, 10]) or implicit **future** [2], it is shown how to deal with "direct-result critical sections"
  - using concepts such as the **promise** [7] or promise pipelining [12]
  - functional programming meets distributed computing for synchronisation

[1] Not to be confused with "guarded commands" [4].

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[1]

  pre-/postlude sections
  - synchronise hardware-originated events

- similar to an explicit ("eventual values" [9, 10]) or implicit **future** [2], it is shown how to deal with "direct-result critical sections"

- 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].

Outline

Preface

Hardware Events
- Fundamentals
- Sequencing
- Implementation

Process Events
- Fundamentals
- Sequencing
- Implementation

Summary

Interrupt Handling

Definition (Interrupt)

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

Interrupt Handling

Definition (Interrupt)

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

- 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

Interrupt Handling

Definition (Interrupt)

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

- a **sudden upcall** (acc. [3]) performed by a processor in the middle of or between actions, depending on the processor model

- 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

Interrupt Handling

Definition (Interrupt)

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

- a **sudden upcall** (acc. [3]) performed by a processor in the middle of or between actions, depending on the processor model
- as to operating systems, usually a **trinity** of problem-specific routines is to be considered
- 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

(Ger.) *Aufgabenverteilung*

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.

Responsibility Assignment

(Ger.) *Aufgabenverteilung*

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)

Responsibility Assignment

(Ger.) *Aufgabenverteilung*

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
- identifies and activates the prelude for the given IRQ

Responsibility Assignment

(Ger.) *Aufgabenverteilung*

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):

prelude
- 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

Responsibility Assignment

(Ger.) *Aufgabenverteilung*

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):

postlude
- operates the device, if still required, and particularly the system
- starts when no more preludes are stacked and, thus, pending
- as the case may be, interacts with a process instance

Responsibility Assignment

(Ger.) *Aufgabenverteilung*

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
- identifies and activates the prelude for the given IRQ
  - ↪ in case of an **edge-triggered** IRQ, takes OS priority before it
  - ↪ in case of a **level-triggered** IRQ, takes OS priority afterwards

Relevance of Postlude

Hint (Asynchronous System Trap, AST [11, 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.

Relevance of Postlude

Hint (Asynchronous System Trap, AST [11, 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

Relevance of Postlude

Hint (Asynchronous System Trap, AST [11, 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

- 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 $\rightsquigarrow$ **synchronisation** or **reentrancy**	☺

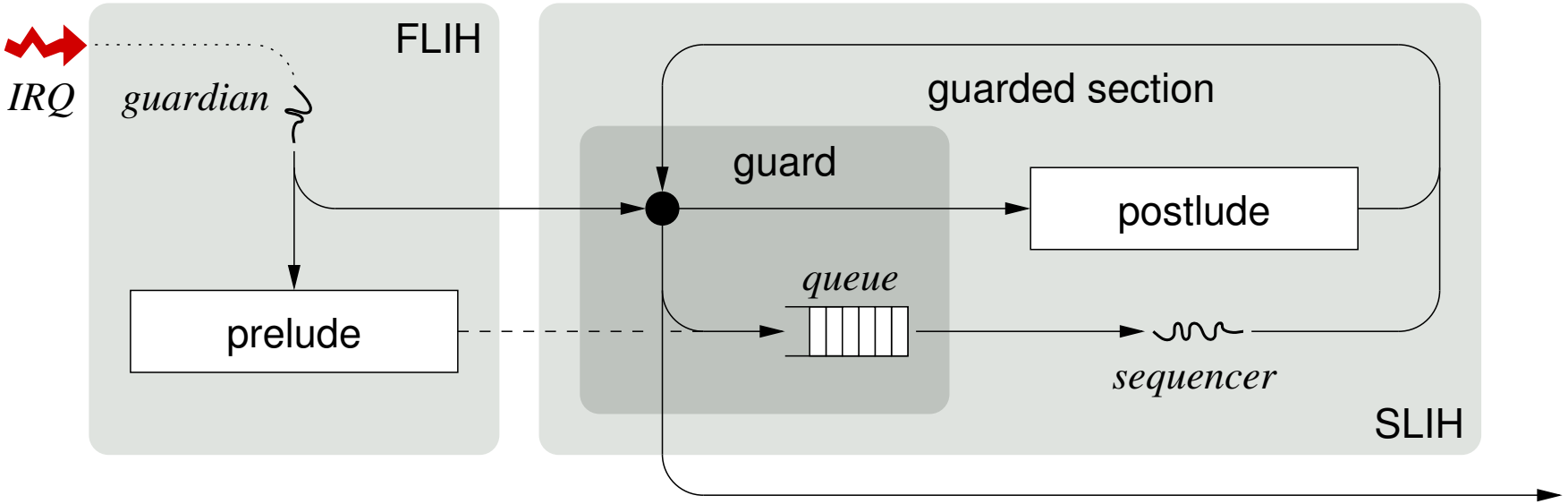
Relevance of Postlude

Hint (Asynchronous System Trap, AST [11, 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.

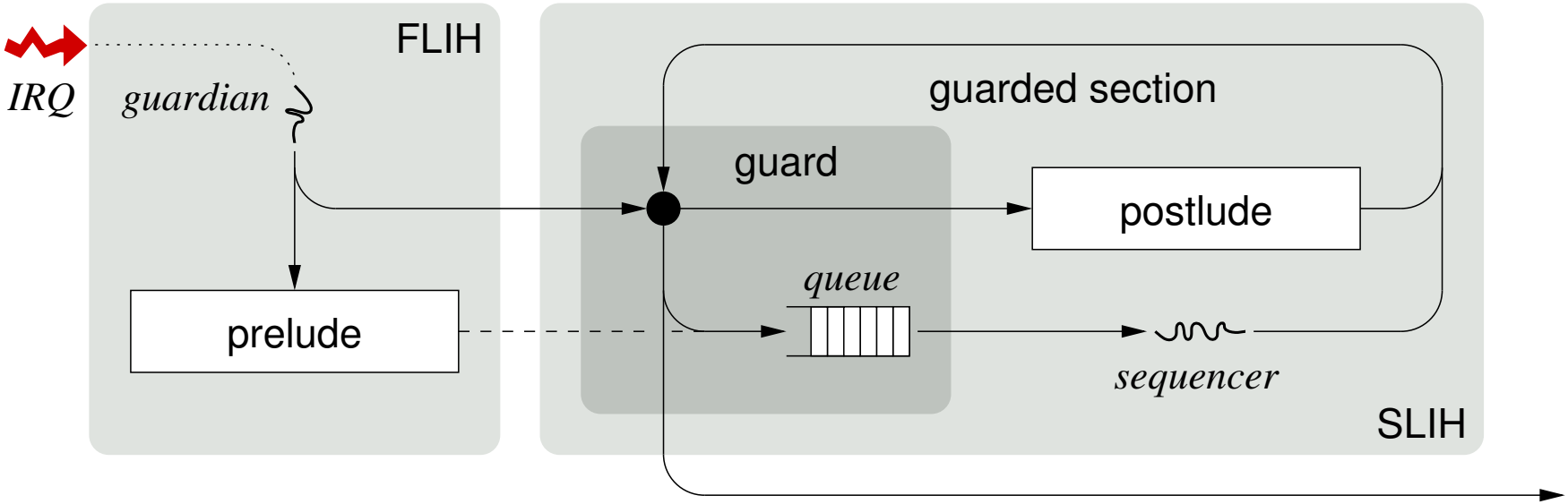
- essentially, the interrupt handler postlude equates to such an AST
- caution is advised when an **interrupt-handler control flow** expands
- purpose of the postlude is to safely allow such control-flow expansions
  - its activation is controlled similar to the control of guarded sections

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

Execution Sequencing of Postludes



- 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

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

Overlapping Pattern

- not unlike the guarded section as to process events described below (cf. p. 20)

- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous pre- and postludes
  - i higher priority preludes may overlap lower priority preludes
  - ii preludes may overlap postludes, but never reverse
  - iii postludes may overlap other postludes and process instances

Overlapping Pattern

- not unlike the guarded section as to process events described below (cf. p. 20)
- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous pre- and postludes
- 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

Overlapping Pattern

- not unlike the guarded section as to process events described below (cf. p. 20)
- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous pre- and postludes
- this **multiple-enqueue/single-dequeue** mode of operation eases the design of a non-blocking synchronised postlude queue

Guardian and Sequencer

From FLIH to SLIH (cf. p. 36*ff.*)

```
1  __attribute__ ((fastcall)) void guardian(long irq) {
2      static usher_t tube = { 0, {0, &tube.load.head} };
3      extern remit_t *(*flih[])(usher_t *);
4      remit_t *task;
5
6  #ifdef __FAME_INTERRUPT_EDGE_TRIGGERED__
7      pivot(&tube.busy, +1); admit(IRQ);  /* take OS priority */
8  #endif
9
10     task = (*flih[irq])(&tube);         /* activate prelude & satisfy IRQ source */
11
12 #ifdef __FAME_INTERRUPT_LEVEL_TRIGGERED__
13     pivot(&tube.busy, +1); admit(IRQ);  /* take OS priority */
14 #endif
15
16     if (tube.busy > 1) {                /* sequencer is already on duty */
17         if (task != 0) deter(&tube, task);  /* enqueue postlude & */
18         avert(IRQ);                         /* leave with CPU priority */
19     } else {                            /* bring sequencer into service */
20         if ((task != 0) && (tube.load.head.link == 0)) remit(task);
21
22         avert(IRQ);                         /* prevent lost unload */
23         while (tube.load.head.link != 0) {
24             admit(IRQ);                     /* take OS priority, again */
25             flush(&tube);                   /* forward pending postludes */
26             avert(IRQ);                     /* leave with CPU priority */
27         }
28     }
29     pivot(&tube.busy, -1);              /* leave critical section */
30 }
```

Lock-Free Synchronised Enqueue

cf. [14]

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

```
1  void enqueue_ms_lfs(queue_t *this, chain_t *item) {
2      chain_t *last;
3
4      item->link = 0;              /* terminate chain: FIFO */
5
6      last = this->tail;           /* settle insertion point */
7      this->tail = item;           /* create new partial list */
8
9      while (last->link != 0)      /* overlapping enqueue! */
10         last = last->link;       /* find end of orig. list */
11
12     last->link = item;           /* insert & combine lists */
13 }
```

Lock-Free Synchronised Enqueue

cf. [14]

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

```
1  void enqueue_ms_lfs(queue_t *this, chain_t *item) {
2      chain_t *last;
3
4      item->link = 0;              /* terminate chain: FIFO */
5
6      last = this->tail;           /* settle insertion point */
7      this->tail = item;           /* create new partial list */
8
9      while (last->link != 0)      /* overlapping enqueue! */
10         last = last->link;       /* find end of orig. list */
11
12     last->link = item;           /* insert & combine lists */
13 }
```

- 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 then may shift the **actual insertion point**

Lock-Free Synchronised Dequeue

cf. [14]

```
1  chain_t *dequeue_ms_lfs(queue_t *this) {
2      chain_t *item;
3
4      if ((item = this->head.link)  /* next item fetched */
5      &&  !(this->head.link = item->link)) {
6          this->tail = &this->head;  /* is last one, reset */
7          if (item->link != 0) {     /* overlapping enq.! */
8              chain_t *help, *lost = item->link;
9              do {                           /* recover latecomers */
10                 help = lost->link;   /* remember next & */
11                 enqueue_ms_lfs(this, lost);   /* rearrange */
12             } while ((lost = help) != 0);
13         }
14     }
15
16     return item;
17 }
```

Lock-Free Synchronised Dequeue

cf. [14]

```
1  chain_t *dequeue_ms_lfs(queue_t *this) {
2     chain_t *item;
3
4     if ((item = this->head.link)  /* next item fetched */
5     &&  !(this->head.link = item->link)) {
6        this->tail = &this->head;  /* is last one, reset */
7        if (item->link != 0) {     /* overlapping enq.! */
8           chain_t *help, *lost = item->link;
9           do {                          /* recover latecomers */
10             help = lost->link;   /* remember next & */
11             enqueue_ms_lfs(this, lost);   /* rearrange */
12          } while ((lost = help) != 0);
13       }
14    }
15
16    return item;
17 }
```

- critical is dequeuing as to the **last element** and overlapped by one or more enqueues, thus, filling up the queue again
  - one moment the fetched item was last, now latecomers must be recovered

Lock-Free Synchronised Dequeue

cf. [14]

```
1  chain_t *dequeue_ms_lfs(queue_t *this) {
2      chain_t *item;
3
4      if ((item = this->head.link)  /* next item fetched */
5      &&  !(this->head.link = item->link)) {
6          this->tail = &this->head;  /* is last one, reset */
7          if (item->link != 0) {     /* overlapping enq.! */
8              chain_t *help, *lost = item->link;
9              do {                           /* recover latecomers */
10                 help = lost->link;         /* remember next & */
11                 enqueue_ms_lfs(this, lost);   /* rearrange */
12             } while ((lost = help) != 0);
13         }
14     }
15
16     return item;
17 }
```

Hint (Lock Freedom)

Some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. [8, p. 142]

- critical is dequeuing as to the **last element** and overlapped by one or more enqueues, thus, filling up the queue again
  - one moment the fetched item was last, now latecomers must be recovered

Wait-Free Solution

```
1  void enqueue_ms_wfs(queue_t *this, chain_t *item) {
2      chain_t *last;
3      item->link = 0;       /* terminate chain: FIFO */
4      last = FAS(&this->tail, item);
5      last->link = item;    /* eventually append item */
6  }
7
8  chain_t *dequeue_ms_wfs(queue_t *this) {
9      chain_t *item = this->head.link;
10     if (item) {           /* check for last item */
11         if (item->link) /* is not, non-critical */
12             this->head.link = item->link;
13         else if (CAS(&this->tail, item, &this->head))
14                 CAS(&this->head.link, item, 0);
15     }
16     return item;
17 }
```

- with the following mapping to GCC atomic intrinsic functions:

```
1  #define FAS(ref,val) __sync_lock_test_and_set(ref, val)
2  #define CAS          __sync_bool_compare_and_swap
```

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)

Recapitulation

- in the **pre-/postlude model**, sequencer becomes that process in the context of which interrupt handling is carried out

Hint (Pro-/Epilogue [15, 14])

*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.*

Recapitulation

- in the **pre-/postlude model**, sequencer becomes that process in the context of which interrupt handling is carried out

Hint (Pro-/Epilogue [15, 14])

*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** [13, p. 10], a technique to serialise execution of conflicting threads
  - differences are due to the constrained pre-/postlude overlapping pattern
  - unless stack-based scheduling [1], any process overlapping is assumed

Recapitulation

- in the **pre-/postlude model**, sequencer becomes that process in the context of which interrupt handling is carried out

Hint (Pro-/Epilogue [15, 14])

*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** [13, p. 10], a technique to serialise execution of conflicting threads

- 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

Critical Sections Revisited

- assuming a **stack** represented as LIFO (*last in, first out*) single-linked list, whose *push*- and *pop*-operations need to be critical sections

```
1  void push(lifo_t *list, chain_t *item) {
2      acquire(&list->lock);        /* enter critical section */
3      item->link = list->link;
4      list->link = item;
5      release(&list->lock);        /* leave critical section */
6  }


7  chain_t *pull(lifo_t *list) {
8      chain_t *item;
9
10     acquire(&list->lock);        /* enter critical section */
11     if ((item = list->link) != 0)
12         list->link = item->link;
13     release(&list->lock);        /* leave critical section */
14
15     return item;
16 }
```

Critical Sections Revisited

. . . what makes the difference?

- assuming a **stack** represented as LIFO (*last in, first out*) single-linked list, whose *push*- and *pop*-operations need to be critical sections

```
1  void push(lifo_t *list, chain_t *item) {
2      acquire(&list->lock);        /* enter critical section */
3      item->link = list->link;
4      list->link = item;
5      release(&list->lock);        /* leave critical section */
6  }
```

  - processes proceed successively

```
7  chain_t *pull(lifo_t *list) {
8      chain_t *item;
9
10     acquire(&list->lock);        /* enter critical section */
11     if ((item = list->link) != 0)
12         list->link = item->link;
13     release(&list->lock);        /* leave critical section */
14
15     return item;
16 }
```

  - processes proceed successively

Critical Sections Revisited

. . . what makes the difference?

- assuming a **stack** represented as LIFO (*last in, first out*) single-linked list, whose *push*- and *pop*-operations need to be critical sections

```
1  void push(lifo_t *list, chain_t *item) {
2      acquire(&list->lock);        /* enter critical section */
3      item->link = list->link;
4      list->link = item;
5      release(&list->lock);        /* leave critical section */
6  }
```

  - processes proceed successively, neither depends on the computation result

```
7  chain_t *pull(lifo_t *list) {
8      chain_t *item;
9
10     acquire(&list->lock);        /* enter critical section */
11     if ((item = list->link) != 0)
12         list->link = item->link;
13     release(&list->lock);        /* leave critical section */
14
15     return item;
16 }
```

  - processes proceed successively, each depends on the computation result

Conditional Fire-and-Forget Pattern

- in the final analysis, critical sections are **twofold**, namely one that is *procedure*- and another one that is *function*-like
  - with the former delivering no direct result, in contrast to the latter

Conditional Fire-and-Forget Pattern

- processes heading for passing through a critical section will proceed unstopped, though simultaneous **passage requests** are serialised
  - at the end of a critical section, these requests will be processed one a time

Conditional Fire-and-Forget Pattern

- processes heading for passing through a critical section will proceed unstopped, though simultaneous **passage requests** are serialised

- accordingly, the **exit protocol** does not have to take care of blocked processes but rather intermediately incurred passage requests
  - the particular leaving process attends to handle accumulated entry calls
  - thus, critical-section execution is **asynchronous** to its requesting process

Conditional Fire-and-Forget Pattern

- processes heading for passing through a critical section will proceed unstopped, though simultaneous **passage requests** are serialised
- accordingly, the **exit protocol** does not have to take care of blocked processes but rather intermediately incurred passage requests
- in case of data dependencies as to the computation within a critical section, synchronisation on **result delivery** becomes necessary
  - thereto, computation results need to be returned and accepted **by proxy**
  - to this end, the following measures have to be provided:
    i. as additional element of the corresponding passage request, a placeholder for the computation result (*consumable resource*) and
    ii. a signalling mechanism to indicate result delivery (*logical synchronisation*)

Conditional Fire-and-Forget Pattern

- processes heading for passing through a critical section will proceed unstopped, though simultaneous **passage requests** are serialised
- accordingly, the **exit protocol** does not have to take care of blocked processes but rather intermediately incurred passage requests
- in case of data dependencies as to the computation within a critical section, synchronisation on **result delivery** becomes necessary
- in the final analysis, critical sections are **twofold**, namely one that is *procedure*- and another one that is *function*-like
  - with the former delivering no direct result, in contrast to the latter

Handling of a Critical-Section Function

- 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

Handling of a Critical-Section Function

- 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]

  promise
  - traced back to [7], a writeable, single-assignment container[2]
  - can be used to successfully complete a future with a value

[2] Refined for *promise pipelining* [12] to overcome latency in distributed systems.

Handling of a Critical-Section Function

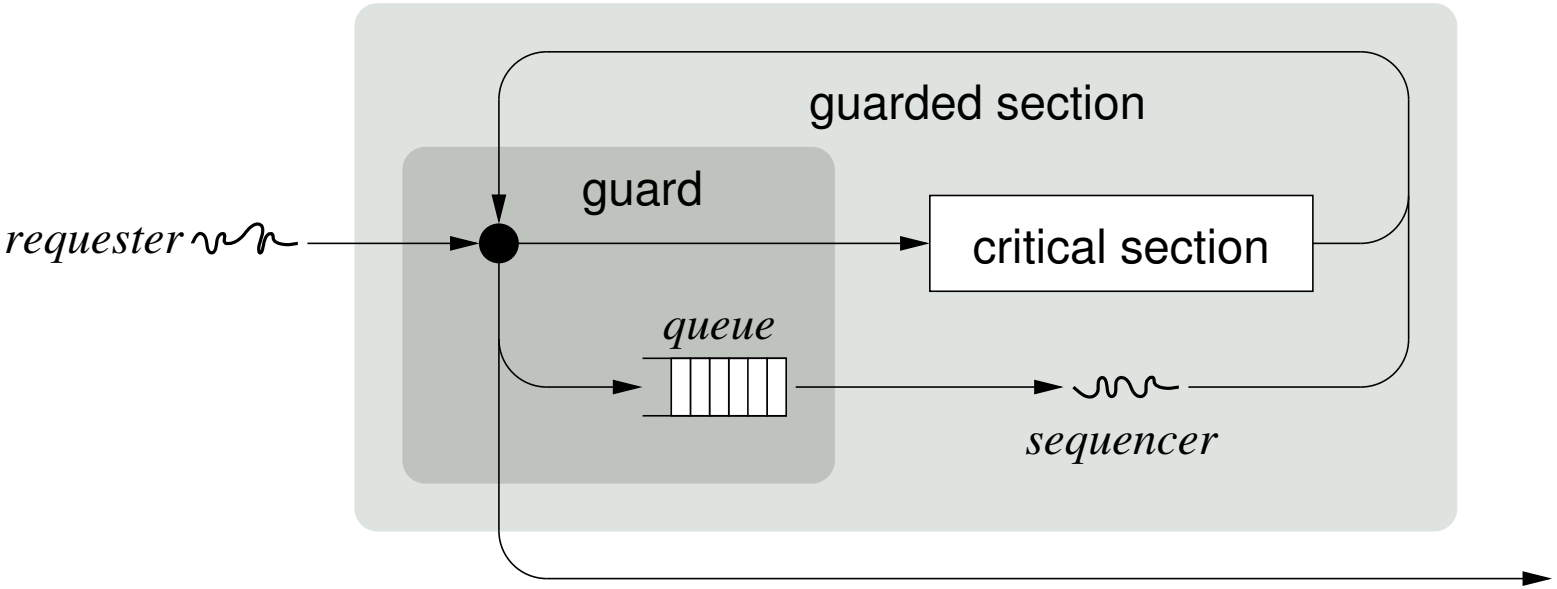
- fall back on known **linguistic concepts** in order to pattern a solution for the above-mentioned problem:
  - future: the *promise* to deliver a value at some later point in time [2]

- 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

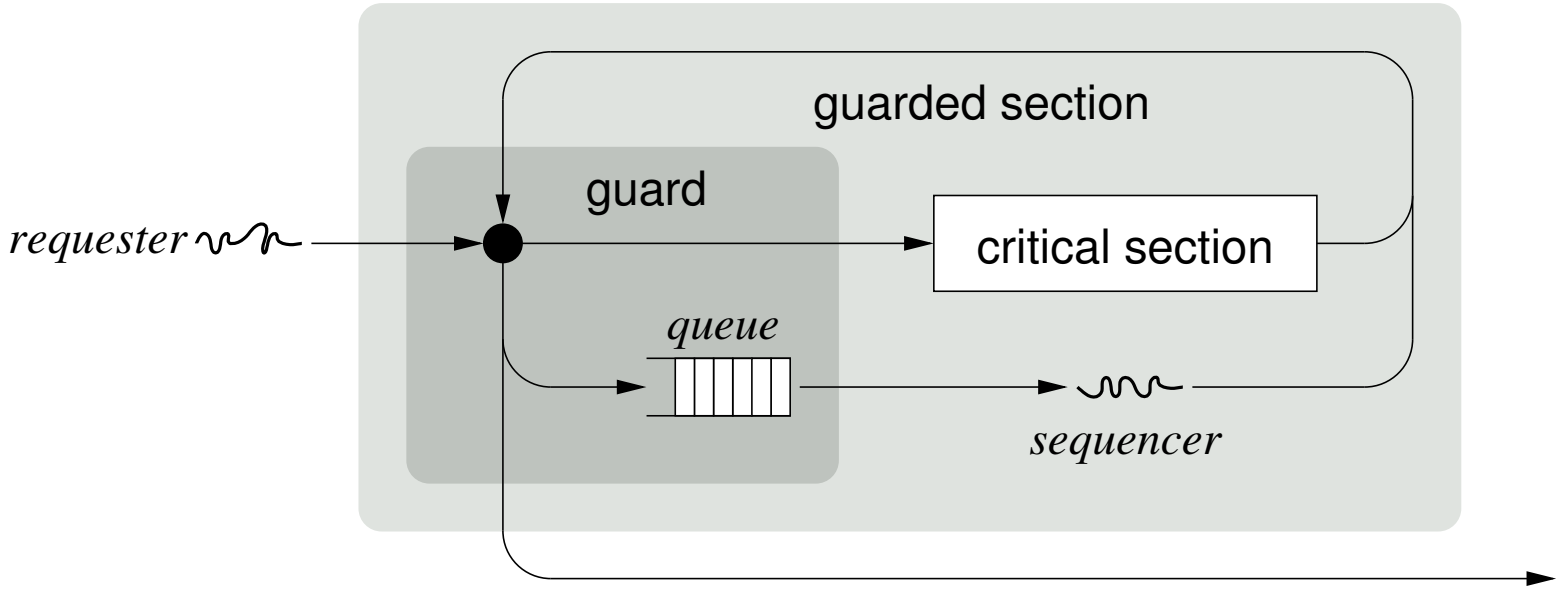
Handling of a Critical-Section Function

- 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]
- each future instance has a dedicated **resolver** taking care of (a) value assignment and (b) **promise states**
- 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 $\rightsquigarrow$ ***signalling semaphore***

Execution Sequencing of Critical Sections

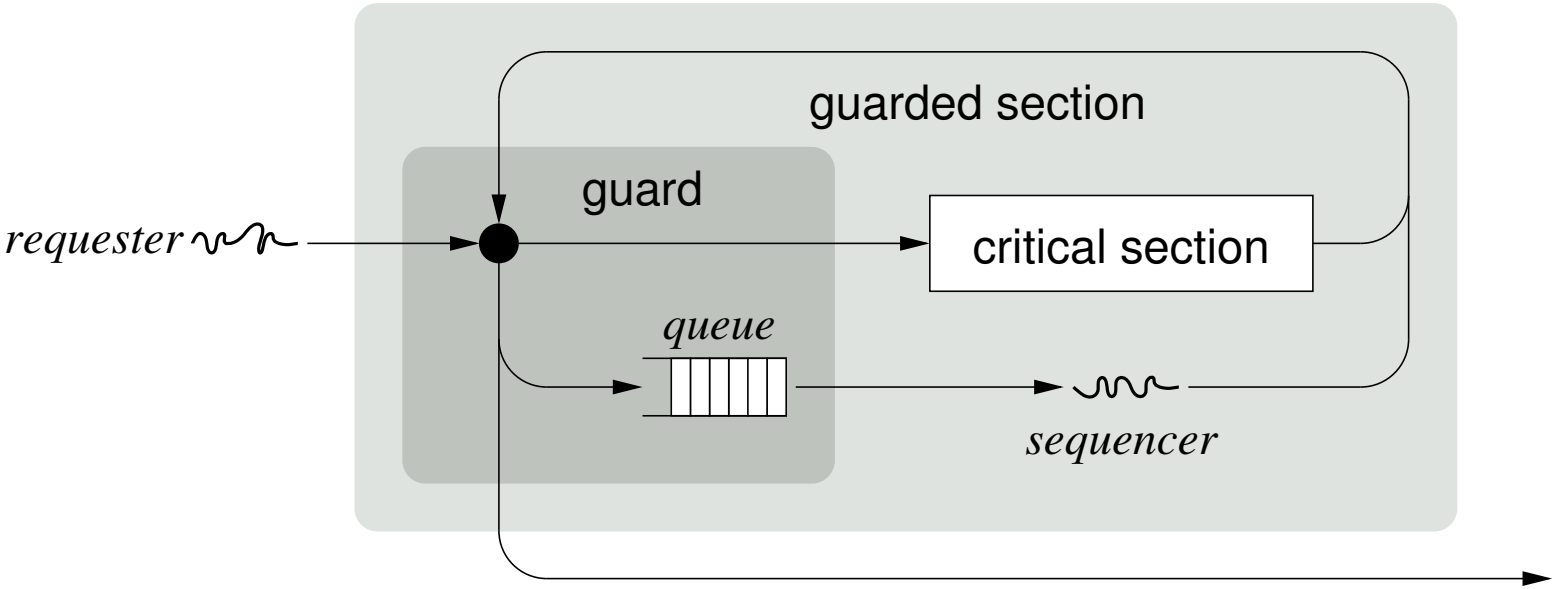


Execution Sequencing of Critical Sections



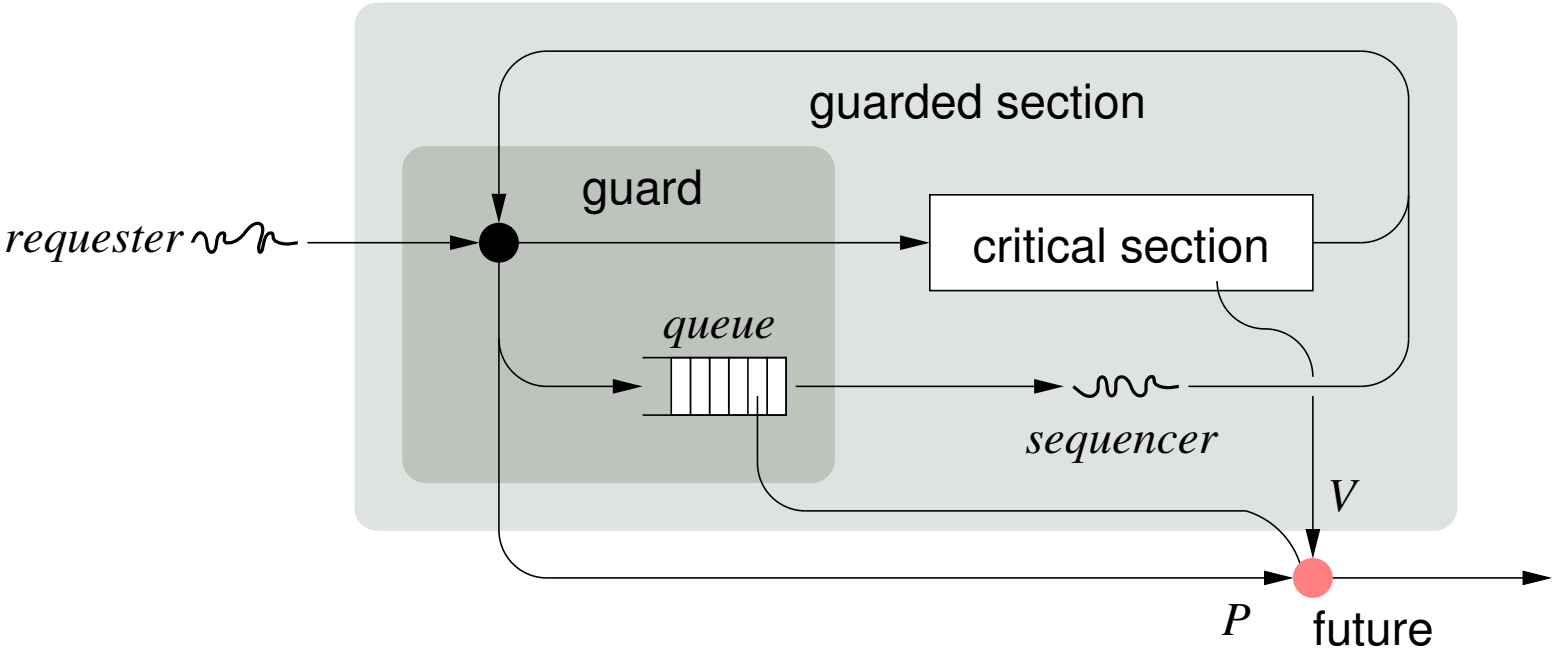
- heading for a critical section depending on the **state of occupancy**:
  - unoccupied
    - guard grants requester access to the critical section
    - the critical section becomes occupied by the requester
  - occupied
    - guard denies requester access to the critical section
    - the request gets queued and the requester bypasses

Execution Sequencing of Critical Sections

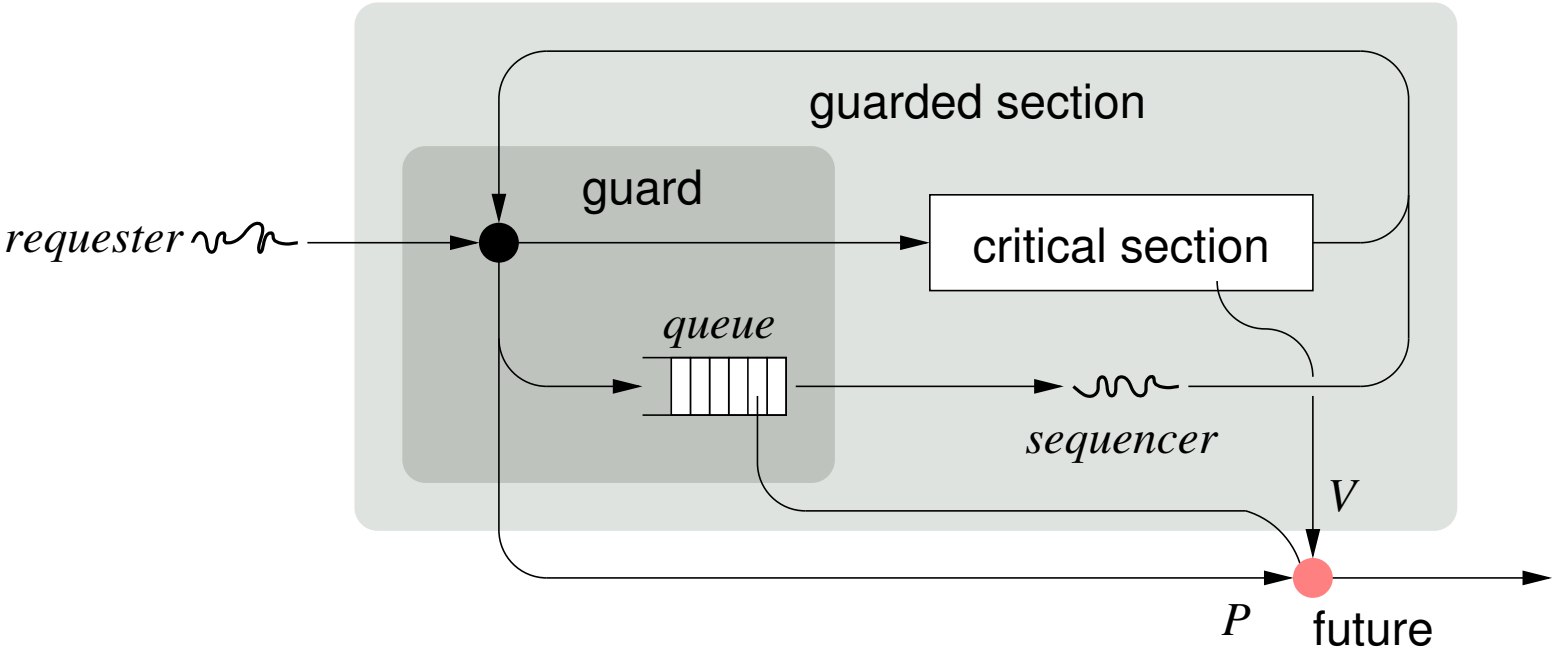


- leaving a critical section depending on the **request-queue state**:
  - empty
    - critical section becomes unoccupied, the process continues
  - full
    - the actual leaving process becomes sequencer and re-enters the critical section for each queued request

Synchronisation of Direct-Result Critical Sections

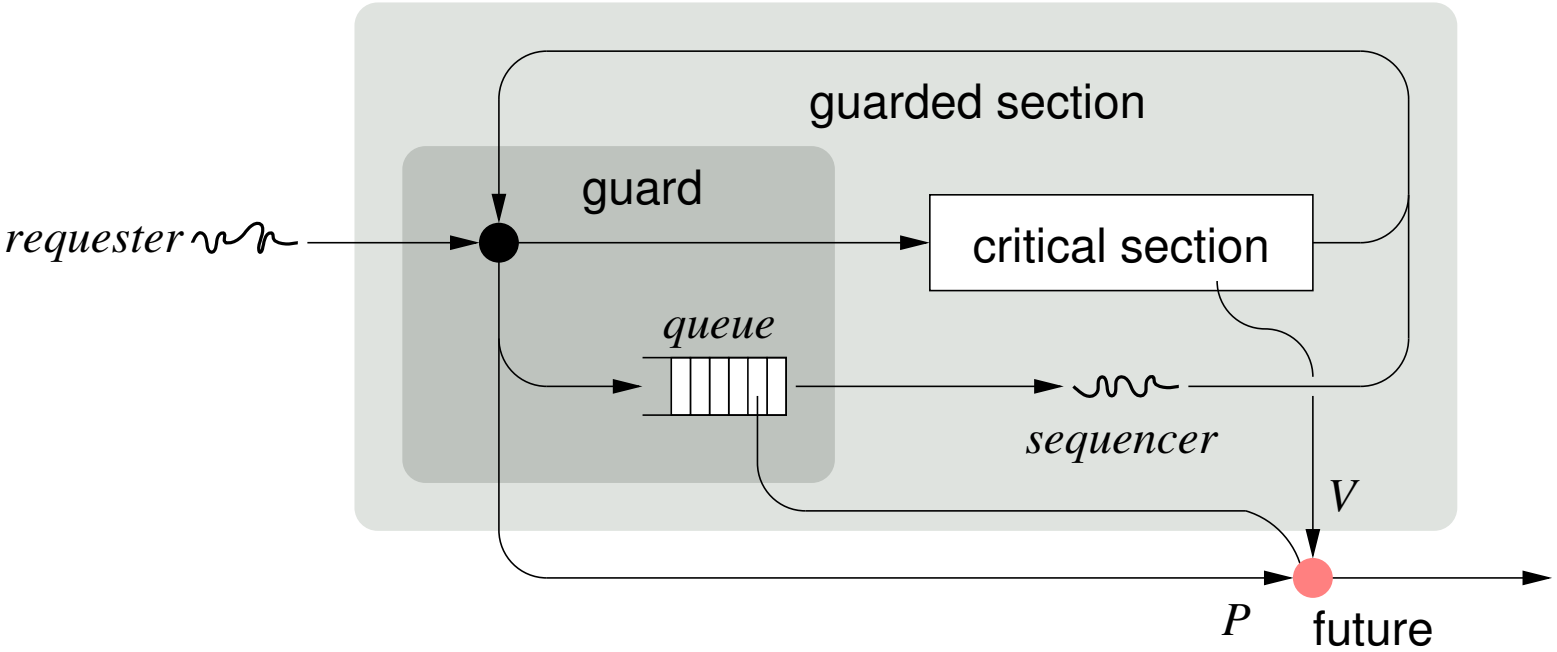


Synchronisation of Direct-Result Critical Sections



- 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

Synchronisation of Direct-Result Critical Sections



- 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

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.

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

Execution Characteristics of the Critical Section

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

- 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.

Execution Characteristics of the Critical Section

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

- 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.

i processes waiting on events caused by an **external process** (e.g., I/O)
ii processes interacting with an **internal process** due to *resource sharing*

Execution Characteristics of the Critical Section

- critical sections controlled by processes in a **run-to-completion style** can be handled straightforwardly
- control of a **run-to-stopover style** of execution of a critical section depends on the locality of peer processes
- 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

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**[3]

[3] Have peripherals (i.e., I/O devices) in mind to understand external processes. Production of input data using a keyboard, mouse, network card, disk, or sensor, for example, is not caused by an OS-controlled **producer-process instance**.

Run-to-Stopover for Peer Processes

- critical sections controlled by processes waiting on events caused by **external processes** can be handled straightforwardly

- unlike **internal processes**, provided that they have to interact with their peers using ***shared resources*** inside a critical section

Run-to-Stopover for Peer Processes

- critical sections controlled by processes waiting on events caused by **external processes** can be handled straightforwardly

- 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

Run-to-Stopover for Peer Processes

- critical sections controlled by processes waiting on events caused by **external processes** can be handled straightforwardly

- 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

Run-to-Stopover for Peer Processes

- critical sections controlled by processes waiting on events caused by **external processes** can be handled straightforwardly
- unlike **internal processes**, provided that they have to interact with their peers using ***shared resources*** inside a critical section
- as a consequence, **precautions** must be taken for interacting internal processes—similar to signalling inside monitors [16, p. 9]
  - without clearing the guarded section, a **stopover process** may deadlock

Overlapping Pattern

- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous requester and sequencer processes

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

Overlapping Pattern

- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous requester and sequencer processes

- 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

Overlapping Pattern

- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous requester and sequencer processes

- 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** [6, 5]

Overlapping Pattern

- notably is the implication in terms of the **constructive restriction** of overlappings as to simultaneous requester and sequencer processes

- this **multiple-enqueue/single-dequeue** mode of operation eases the design of a non-blocking synchronised passage-request queue

Hint (Wait Freedom)

Any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. [8, p. 124]

Data Type I

```
1  typedef struct guard {
2      int book;              /* # of concurrent requests */
3      queue_t load;          /* pending passage requests */
4  #ifdef __FAME_GUARD_ADVANCED__
5      ...
6  #endif
7  } guard_t;
```

Data Type I

```
1  typedef struct guard {
2      int book;              /* # of concurrent requests */
3      queue_t load;          /* pending passage requests */
4  #ifdef __FAME_GUARD_ADVANCED__
5      ...
6  #endif
7  } guard_t;
```

- invariably, a **chain-like queue** of registered "passage requests"
  - mandatory, sufficient for elementary guarded sections
  - with a twofold meaning of the *book* attribute depending on its value
    - i the actual number of passage requests pending for processing
    - ii the state of occupancy (cf. p. 20): occupied if $book > 0$, unoccupied else

Data Type I

```
1  typedef struct guard {
2      int book;              /* # of concurrent requests */
3      queue_t load;          /* pending passage requests */
4  #ifdef __FAME_GUARD_ADVANCED__
5      ...
6  #endif
7  } guard_t;
```

- invariably, a **chain-like queue** of registered "passage requests"

- variably, additional stuff for advanced control of guarded sections:
  - some **timeout** that ensures progress for the actual **major sequencer**
  - a **minor sequencer** to replace the major sequencer at timeout
  - any management data to prevent **priority inversion**, if applicable
  - ...

Claiming

- vouch for sequential execution of a guarded critical section:

```
1 inline order_t *vouch(guard_t *this, order_t *work) {
2     enqueue(&this->load, work);
3     if (FAA(&this->book, 1) == 0)
4         return dequeue(&this->load);
5     return 0;
6 }
```

- 2 remember this passage request
- 3 check state of occupancy and book passage request
- 4 was unoccupied, became sequencer, accept first passage request
- could be a request different from the one that was just remembered

Claiming and Clearing

- vouch for sequential execution of a guarded critical section:

```
1  inline order_t *vouch(guard_t *this, order_t *work) {
2      enqueue(&this->load, work);
3      if (FAA(&this->book, 1) == 0)
4          return dequeue(&this->load);
5      return 0;
6  }
```

- clear the next passage request, if any, pending for processing:

```
7  inline order_t *clear(guard_t *this) {
8      if (FAA(&this->book, -1) > 1)
9          return dequeue(&this->load);
10     return 0;
11 }
```

8 ■ count completion and check for further pending requests

9 ■ remove next passage request, if any available

Data Type II


```
1 typedef struct order {
2     chain_t next;          /* passage-request chaining */
3     item_t post;           /* argument placeholder */
4 } order_t;
```

Data Type II

```
1  typedef struct order {
2      chain_t next;          /* passage-request chaining */
3      item_t post;           /* argument placeholder */
4  } order_t;
```

- layout of an **argument vector** for passage-request parameters:

```
1  typedef union item {
2      long (*lump)[];        /* argument vector (N > 1) */
3      long sole;             /* single argument (N = 1) */
4  } 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. 41)

Data Type II

```
1  typedef struct order {
2      chain_t next;          /* passage-request chaining */
3      item_t post;           /* argument placeholder */
4  } order_t;
```

- layout of an **argument vector** for passage-request parameters:

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

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

Piece the Puzzle Together

Piece the Puzzle Together

- fore **editing** of passage-request parameters, optional:

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

Piece the Puzzle Together

- **entry protocol**, agreement on the sequencer process:

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

Piece the Puzzle Together

- midsection (i.e., actual critical section), **solo attempt**:

```
6 /* Several Species of Small Furry Animals
7  * Gathered Together in a Cave and
8  * Grooving with a Pict */
```

Piece the Puzzle Together

- **exit protocol**, processing of pending passage requests:

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

Piece the Puzzle Together

- fore **editing** of passage-request parameters, optional:

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

- **entry protocol**, agreement on the sequencer process:

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

- midsection (i.e., actual critical section), **solo attempt**:

```
6 /* Several Species of Small Furry Animals
7  * Gathered Together in a Cave and
8  * Grooving with a Pict */
```

- **exit protocol**, processing of pending passage requests:

```
9 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

Outline

Preface

Hardware Events
- Fundamentals
- Sequencing
- Implementation

Process Events
- Fundamentals
- Sequencing
- Implementation

Summary

Résumé

- 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

[4] Operating-system machine or instruction set architecture level, respectively.

Résumé

- 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

[4] Operating-system machine or instruction set architecture level, respectively.

Résumé

- 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

[4] Operating-system machine or instruction set architecture level, respectively.

Résumé

- 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

[4] Operating-system machine or instruction set architecture level, respectively.

Résumé

- 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

[4] Operating-system machine or instruction set architecture level, respectively.

Reference List I

[1] Baker, T. P.:
Stack-Based Scheduling of Realtime Processes.
In: *The Journal of Real-Time Systems* 3 (1991), März, Nr. 1, S. 67–99

[2] Baker, Jr., H. C. ; Hewitt, C. :
The Incremental Garbage Collection of Processes.
In: Low, J. (Hrsg.): *Proceedings of the 1977 ACM Symposium on Artificial Intelligence and Programming Languages*.
New York, NY, USA : ACM, 1977, S. 55–59

[3] Clark, D. D.:
The Structuring of Systems Using Upcalls.
In: Baskett, F. (Hrsg.) ; Birrell, A. (Hrsg.) ; Cheriton, D. (Hrsg.): *Proceedings of the Tenth ACM Symposium on Operating System Principles (SOSP '85)*.
New York, NY, USA : ACM, 1985. –
ISBN 0–89791–174–1, S. 171–180

[4] Dijkstra, E. W.:
Guarded Commands, Nondeterminacy and Formal Derivation of Programs.
In: *Communications of the ACM* 18 (1975), Aug., Nr. 8, S. 453–457

Reference List II

[5] Drescher, G. ; Schröder-Preikschat, W. :
An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections / Friedrich-Alexander-Universität Erlangen-Nürnberg, Department of Computer Science.
2015 (CS-2015-01). –
Forschungsbericht. –
ISSN 2191–5008

[6] Drescher, G. ; Schröder-Preikschat, W. :
Guarded Sections: Structuring Aid for Wait-Free Synchronisation.
In: *Proceedings of the 18th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2015)*, IEEE Computer Society, 2015. – ISBN 978–1–4799–8781–8, S. 280–283

[7] Friedman, D. P. ; Wise, D. S.:
The Impact of Applicative Programming on Multiprocessing.
In: *Proceedings of the International Conference on Parallel Processing (ICPP 1976)*. Piscataway, NJ, USA : IEEE Computer Society, 1976, S. 263–272

[8] Herlihy, M. :
Wait-Free Synchronization.
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124–149

Reference List III

[9] Hibbard, P. :
Parallel Processing Facilities.
In: Schuman, S. A. (Hrsg.): *Proceedings of the Second Conference on New Directions in Algorithmic Languages*, Institut de Recherche d'Informatique et d'Automatique (IRIA), Rocquencourt, 1976. –
ISBN 2726101240, S. 1–7

[10] Hibbard, P. ; Hisgen, A. ; Rodeheffer, T. :
A Language Implementation Design for a Multiprocessor Computer System.
In: *Proceedings of the 5th International Symposium on Computer Architecture (ISCA '78)*.
New York, NY, USA : ACM, 1978, S. 66–72

[11] Leffler, S. J. ; McKusick, M. K. ; Karels, M. J. ; Quarterman, J. S.:
The Design and Implementation of the 4.3 BSD UNIX Operating System.
Addison-Wesley, 1989. –
ISBN 0–201–06196–1

Reference List IV

[12] Liskov, B. J. H. ; Shrira, L. :
Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.
In: *Proceedings of the ACM SIGPLAN 1988 International Conference on Programming Language Design and Implementation (PLDI '88)*.
New York, NY, USA : ACM, 1988. –
ISBN 0–89791–269–1, S. 260–267

[13] Pu, C. ; Massalin, H. :
An Overview of The Synthesis Operating System / Department of Computer Science, Columbia University.
New York, NY, USA, 1989 (CUCS-470-89). –
Forschungsbericht

[14] Schön, F. ; Schröder-Preikschat, W. ; Spinczyk, O. ; Spinczyk, U. :
On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System.
In: Lee, I. (Hrsg.) ; Kaiser, J. (Hrsg.) ; Kikuno, T. (Hrsg.) ; Selic, B. (Hrsg.): *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*.
Washington, DC, USA : IEEE Computer Society, 2000. –
ISBN 0–7695–0607–0, S. 270–277

Reference List V

[15] Schröder-Preikschat, W. :
The Logical Design of Parallel Operating Systems.
Upper Saddle River, NJ, USA : Prentice Hall International, 1994. –
ISBN 0–13–183369–3

[16] Schröder-Preikschat, W. :
Monitor.
In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems*.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 8

Guardian Insulating and Invoking

```
1  _joint:
2      pushl %ecx          # save volatile register
3      movl  $0, %ecx      # pass IRQ number
4  _jointN:                # come here for IRQ number N > 0
5      pushl %edx          # save another volatile register
6      pushl %eax          # ditto
7      call _guardian      # fastcall to guardian
8      popl  %eax          # restore volatile register
9      popl  %edx          # ditto
10     popl  %ecx          # ditto
11     iret                # resume interrupted process
```

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

```
1  _joint42:
2      pushl %ecx          # save volatile register
3      movl  $42, %ecx     # pass IRQ number
4      jmp   _jointN       # switch to common joint section...
```

S{a,i}mple Interrupt Handler

- first-level interrupt handler (FLIH), at CPU/OS priority (p. 11, l. 7)

```
1  remit_t *prelude(/*optional*/usher_t *tube) {
2      static remit_t task = { {}, postlude};
3      /* Come here for device pre-processing &
4       * device-related IRQ acknowledgement. */
5      deter(tube, &task); /* force postlude to queue */
6      return 0;           /* don't request shortcut */
7  }
```

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

- second-level interrupt handler (SLIH), at OS priority (p. 11, l. 7/13)

```
1  void postlude(/*optional*/order_t *todo) {
2      /* Come here for device post-processing &
3       * any asynchronous system interaction. */
4      V((semaphore_t *)todo->post.sole);
5  }
```

  - system interaction means: to *vouch* for guarded sections (cf. p. 28)

Interrupt-Handler Guard

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

```
1  typedef guard_t usher_t;
2
3  inline void deter(usher_t *tube, remit_t *task) {
4      enqueue(&tube->load, &task->data.next);
5  }
6
7  inline remit_t *untie(usher_t *tube) {
8      return (remit_t *)dequeue(&tube->load);
9  }
10
11 inline void flush(usher_t * tube) {
12     remit_t *next;
13     do if ((next = untie(tube))) remit(next);
14     while (next != 0);
15 }
```

- with queue synchronisation style: `#define __FAME_SYNC_ITS__`
  - resulting in "`{en,de}queue_ms_lfs`" or -"`_wfs`", resp.

Job Definition and Start

- a SLIH or an interrupt-handler postlude, resp., is a **passage request** (cf. p. 27) attended by a procedure address
  - that is to say, a request object with implicit processing method

```
1  typedef struct remit {
2      order_t data;                    /* parameter set */
3      void (*code)(order_t *);         /* procedure address */
4  } remit_t;
5
6  inline void remit(remit_t *this) {
7      (*this->code)(&this->data); /* run that job */
8  }
```

- 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 [13, p. 10]

Art of Waiting

- straightforward is the use of a **signalling semaphore**[5]:

```
1 typedef semaphore_t indicator_t;
2 inline void enroll(indicator_t *hint) { }
3 inline void repose(indicator_t *hint) { P(hint); }
4 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
  - a V saves the signalling event in the semaphore, causing P to continue
- another option is falling back on the **event queue** [16, p. 17]:
  - just if one wants to implement P and V as a guarded section, for example

```
1 typedef event_t indicator_t;
2 inline void enroll(indicator_t *hint) { catch(hint); }
3 inline void repose(indicator_t *hint) { coast(); }
4 inline void arouse(indicator_t *hint) { cause(hint); }
```

[5] A **binary semaphore** used in a producer/consumer style of interaction.

Order Allocation/Deallocation

```
1  inline order_t *order(unsigned long n) {
2      order_t *item;
3      if (n < 2)
4          item = (order_t *)malloc(sizeof(order_t));
5      else {
6          item = (order_t *)
7              malloc(sizeof(order_t) + n * sizeof(long));
8          if (item)
9              item->post.lump = (void *)
10                 ((long)item + sizeof(*item));
11     }
12     return item;
13 }
14
15 inline void ditch(order_t *item) {
16     free(item);
17 }
```

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

Data Type III

```
1  typedef struct future {
2      promise_t data;      /* prospective value */
3      indicator_t gate;    /* signalling element */
4  } future_t;
```

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

```
1  typedef enum status {
2      PENDING, KEPT, BROKEN
3  } status_t;
4
5  typedef struct promise {
6      status_t bond;       /* processing state */
7      item_t item;         /* future-value placeholder */
8  } promise_t;
```

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

S{a,i}mple Future Implementation

```
1  inline status_t probe(future_t *this) {
2      return this->data.bond;
3  }
4
5  inline void trust(future_t *this) { enroll(&this->gate); }
6
7  inline item_t *exact(future_t *this) {
8      repose(&this->gate);
9      return probe(this) == KEPT ? &this->data.item : 0;
10 }
11
12 inline void bring(future_t *this, status_t bond) {
13     this->data.bond = bond;
14     arouse(&this->gate);
15 }
16
17 inline void prove(future_t *this, item_t *item) {
18     this->data.item = *item;
19     bring(this, KEPT);
20 }
21
22 inline void abort(future_t *this) { bring(this, BROKEN); }
```