

Concurrent Systems

Nebenläufige Systeme

XI. Non-Blocking Dynamic Data Structures

Wolfgang Schröder-Preikschat

January 28, 2021



Agenda

Preface

Singly Linked List

Working Principle

Concurrent Operation

Collating Sequences

Stack

Queue

Summary



Preface

Singly Linked List

Working Principle

Concurrent Operation

Collating Sequences

Stack

Queue

Summary



Subject Matter

- discussion of the abstract concept of **non-blocking synchronisation** in the context of **dynamic data structures**



Subject Matter

- discussion of the abstract concept of **non-blocking synchronisation** in the context of **dynamic data structures**
 - namely the **singly linked list** with any access pattern of the operations
 - this is compared to structures with a defined access pattern: LIFO, FIFO
 - the **stack** and the **queue** as a linked dynamic data structure



- starting from conventional sequential solutions, semantically identical non-sequential alternatives are gradually developed
 - race conditions are revealed and tricks for avoidance are presented
 - the limits of the developed solutions are discussed



- an important aspect in the considerations and driving idea is to find solutions for the **conservative handling** of dynamic data structures
 - on the one hand, this means being able to reuse the data structures of the corresponding sequential solutions
 - coordination as a *minimal extension of system functions* [5] in its purest form
 - that is, the extension only refers to the *instructions* of a virtual machine



- an important aspect in the considerations and driving idea is to find solutions for the **conservative handling** of dynamic data structures

- on the other hand, the solutions rely on comparatively simple elementary operations such as CAS and FAS



- discussion of the abstract concept of **non-blocking synchronisation** in the context of **dynamic data structures**
- starting from conventional sequential solutions, semantically identical non-sequential alternatives are gradually developed
- an important aspect in the considerations and driving idea is to find solutions for the **conservative handling** of dynamic data structures



Outline

Preface

Singly Linked List

Working Principle

Concurrent Operation

Collating Sequences

Stack

Queue

Summary



Data Type I

- reunion with a **dynamic data structure** ([8, p.9] and [10, p.18]):
 - basic abstraction from any entity as a **list element**

```
1 typedef struct chain {  
2     struct chain *link; /* next list element */  
3 } chain_t;
```



Data Type I

- reunion with a **dynamic data structure** ([8, p. 9] and [10, p. 18]):
 - basic abstraction from any entity as a **list element**

```
1 typedef struct chain {
2     struct chain *link; /* next list element */
3 } chain_t;
```

- based on this, the following operations are defined:

```
chain_t *infix(chain_t *this, chain_t *item)
chain_t *unfix(chain_t *this)
chain_t *split(chain_t *this)
chain_t *visit(chain_t *this, chain_t *item)
chain_t *annex(chain_t *this, chain_t *item)
chain_t *erase(chain_t *this, chain_t *item)
```



Data Type I

- reunion with a **dynamic data structure** ([8, p. 9] and [10, p. 18]):
 - basic abstraction from any entity as a **list element**

```
1 typedef struct chain {  
2     struct chain *link; /* next list element */  
3 } chain_t;
```

- based on this, the following operations are defined:

```
chain_t *infix(chain_t *this, chain_t *item)  
chain_t *unfix(chain_t *this)  
chain_t *split(chain_t *this)  
chain_t *visit(chain_t *this, chain_t *item)  
chain_t *annex(chain_t *this, chain_t *item)  
chain_t *erase(chain_t *this, chain_t *item)
```

- with `this` the chain link on which the operation should work,
- `item` the list entry to be inserted, searched for or deleted and
- a chain link or list entry as a result, depending on the operation



- simple list manipulation:

```
1 inline chain_t *infix_dos(chain_t *this, chain_t *item) {
2     item->link = this->link;
3     return this->link = item;
4 }
```

Caller assured: *infix*

An entry (*item*) to be inserted is not yet on the list.



- simple list manipulation:

```
5 inline chain_t *unfix_dos(chain_t *this) {
6     chain_t *item;
7     item = this->link;
8     if (item != 0)
9         this->link = item->link;
10    return item;
11 }
```



- simple list manipulation:

```
12 inline chain_t *split_dos(chain_t *this) {
13     chain_t *tail;
14     tail = this->link;
15     this->link = 0;
16     return tail;
17 }
```



- simple list manipulation: insert and unhook entry, split list

```
1  inline chain_t *infix_dos(chain_t *this, chain_t *item) {
2      item->link = this->link;
3      return this->link = item;
4  }
5  inline chain_t *unfix_dos(chain_t *this) {
6      chain_t *item;
7      item = this->link;
8      if (item != 0)
9          this->link = item->link;
10     return item;
11 }
12 inline chain_t *split_dos(chain_t *this) {
13     chain_t *tail;
14     tail = this->link;
15     this->link = 0;
16     return tail;
17 }
```

Caller assured: infix

An entry (item) to be inserted is not yet on the list.



- more complex list manipulation: seek out

```
1 inline chain_t *visit_dos(chain_t *this, chain_t* item) {
2     chain_t *next;
3     while (this && ((next = this->link) != item))
4         this = next;
5     return this;
6 }
```



- more complex list manipulation: seek out, append

```
1 inline chain_t *visit_dos(chain_t *this, chain_t* item) {
2     chain_t *next;
3     while (this && ((next = this->link) != item))
4         this = next;
5     return this;
6 }

7 inline chain_t *annex_dos(chain_t *this, chain_t* item) {
8     return infix_dos(visit_dos(this, 0), item);
9 }
```



- more complex list manipulation: seek out, append, delete

```
1 inline chain_t *visit_dos(chain_t *this, chain_t* item) {
2     chain_t *next;
3     while (this && ((next = this->link) != item))
4         this = next;
5     return this;
6 }
7
7 inline chain_t *annex_dos(chain_t *this, chain_t* item) {
8     return infix_dos(visit_dos(this, 0), item);
9 }
10
10 inline chain_t *erase_dos(chain_t *this, chain_t *item) {
11     chain_t *hand;
12     hand = visit_dos(this, item);
13     if (hand != 0)
14         hand = unfix_dos(hand);
15     return hand;
16 }
```



Application Example

```
1 #include "chain.h"
2
3 typedef struct list {
4     chain_t head;
5 } list_t;
```



Application Example

- good for classic collating strategies
 - LIFO list \rightsquigarrow stack (cf. [10, p.19])

```
1 #include "chain.h"
2
3 typedef struct list {
4     chain_t head;
5 } list_t;
6
6 inline void push(list_t *this, chain_t *item) {
7     infix(&this->head, item);
8 }
9 inline chain_t *pull(list_t *this) {
10    return unfix(&this->head);
11 }
```



Application Example

```
1 #include "chain.h"
2
3 typedef struct list {
4     chain_t head;
5 } list_t;
```

- good for classic collating strategies
- FIFO list \leadsto queue

```
12 inline void enlist(list_t *this, chain_t *item) {
13     annex(&this->head, item);
14 }
15 inline chain_t *delist(list_t *this) {
16     return unfix(&this->head);
17 }
```



```
1 #include "chain.h"
```

```
2
3 typedef struct list {
4     chain_t head;
5 } list_t;
```

■ good for classic collating strategies

■ LIFO list \rightsquigarrow stack (cf. [10, p.19])

■ FIFO list \rightsquigarrow queue

```
6 inline void push(list_t *this, chain_t *item) {
7     infix(&this->head, item);
8 }
```

```
9 inline chain_t *pull(list_t *this) {
10    return unfix(&this->head);
11 }
```

```
12 inline void enlist(list_t *this, chain_t *item) {
13    annex(&this->head, item);
14 }
```

```
15 inline chain_t *delist(list_t *this) {
16    return unfix(&this->head);
17 }
```



Non-Sequential Execution

- all chain operations are prone to **race conditions**:
 - `infix` ■ new entries can disappear
 - `unfix` ■ the same entry can be extracted several times
 - `split` ■ list entries at the front end can disappear
 - `visit` ■ incorrect/invalid list section can be run through



Non-Sequential Execution

- all chain operations are prone to **race conditions**:
 - `infix` ■ new entries can disappear
 - `unfix` ■ the same entry can be extracted several times
 - `split` ■ list entries at the front end can disappear
 - `visit` ■ incorrect/invalid list section can be run through
 - `annex` ■ an entry on a wrong list can be made (inherited error)
 - `erase` ■ an entry was wrongly not deleted (inherited error)



Non-Sequential Execution

- all chain operations are prone to **race conditions**

- they are **not thread safe** and need to be synchronised
 - blocking synchronisation can be child's play here: use a **chain monitor**
 - very simple is a medium-grain approach, with a lock on the entire list
 - more difficult is a fine-grained solution, with one lock per list item



Non-Sequential Execution

- all chain operations are prone to **race conditions**

- they are **not thread safe** and need to be synchronised
 - blocking synchronisation can be child's play here: use a **chain monitor**
 - very simple is a medium-grain approach, with a lock on the entire list
 - more difficult is a fine-grained solution, with one lock per list item
 - ↪ in both cases, the locking effort is high compared to the actual operation



Non-Sequential Execution

- all chain operations are prone to **race conditions**

- they are **not thread safe** and need to be synchronised
 - non-blocking synchronisation is basically a fine-grained technique
 - thus also of a similar complexity as the corresponding blocking variant
 - because protecting a single list element is usually not enough



Non-Sequential Execution

- all chain operations are prone to **race conditions**

- they are **not thread safe** and need to be synchronised
 - non-blocking synchronisation is basically a fine-grained technique
 - thus also of a similar complexity as the corresponding blocking variant
 - because protecting a single list element is usually not enough
 - ↔ simultaneous processes to the left and right of it must also be considered



Non-Sequential Execution

- short blocking times are asked what motivates an optimistic approach
- non-blocking is no more difficult than fine-grain blocking



- attach an entry to the hook:

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do item->link = this->link;
3     while (!CAS(&this->link, item->link, item));
4     return item;
5 }
```



- attach an entry to the hook:

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do item->link = this->link;
3     while (!CAS(&this->link, item->link, item));
4     return item;
5 }
```

- a certain collating strategy was wrongly assumed here: LIFO
 - this only manipulates the list on one side: at the top (`this`) of the stack
 - but now every other position on the list can be changed at the same time



- attach an entry to the hook:

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do item->link = this->link;
3     while (!CAS(&this->link, item->link, item));
4     return item;
5 }
```

- note that the insertion point here (`*this`) can be any list element
 - this list item could be deleted while a new item is being appended to it
 - thus, the new list item disappears immediately with the deleted element



- attach an entry to the hook:

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do item->link = this->link;
3     while (!CAS(&this->link, item->link, item));
4     return item;
5 }
```

- that this entry (`*this`) is currently being deleted must be indicated
 - the link pointer of the element concerned is to be regarded as frail
 - it may not be used for chaining...



- attach an entry to the hook: almost too good to be true...

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do item->link = this->link;
3     while (!CAS(&this->link, item->link, item));
4     return item;
5 }
```

- that this entry (`*this`) is currently being deleted must be indicated
 - the link pointer of the element concerned is to be regarded as frail
 - it may not be used for chaining...



- unhook the attached element and attach its successor

```
1 inline chain_t *unfix_lfs(chain_t *this) {
2     chain_t *item;
3     do if ((item = pure(this->link)) == 0) break;
4     while (!purge(this, item, pure(item->link)));
5     return item;
6 }
```



- unhook the attached element and attach its successor

```
1 inline chain_t *unfix_lfs(chain_t *this) {
2     chain_t *item;
3     do if ((item = pure(this->link)) == 0) break;
4     while (!purge(this, item, pure(item->link)));
5     return item;
6 }
```

Delete mark
a pointer tag

- the deletion indicator is based on the tagging of chain attributes (`link`)
 - when deleting (`purge`), the `link` pointer attribute becomes frail (l. 4)
 - when using such a pointer attribute, attention must be paid to purity (l. 3)



- unhook the attached element and attach its successor

```
1 inline chain_t *unfix_lfs(chain_t *this) {
2     chain_t *item;
3     do if ((item = pure(this->link)) == 0) break;
4     while (!purge(this, item, pure(item->link)));
5     return item;
6 }
```

Delete mark
a pointer tag

- the actual deletion is done using the following auxiliary function

```
7 inline bool purge(chain_t *this, chain_t *item, chain_t *next) {
8     bool done;
9     CAS(&item->link, next, mark(next, FRAIL));
10    if (!(done = CAS(&this->link, item, next)))
11        CAS(&item->link, mark(next, FRAIL), next);
12    return done;
13 }
```



- unhook the attached element and attach its successor

```
1 inline chain_t *unfix_lfs(chain_t *this) {
2     chain_t *item;
3     do if ((item = pure(this->link)) == 0) break;
4     while (!purge(this, item, pure(item->link)));
5     return item;
6 }
```

Delete mark
a pointer tag

- the actual deletion is done using the following auxiliary function

```
7 inline bool purge(chain_t *this, chain_t *item, chain_t *next) {
8     bool done;
9     CAS(&item->link, next, mark(next, FRAIL));
10    if (!(done = CAS(&this->link, item, next)))
11        CAS(&item->link, mark(next, FRAIL), next);
12    return done;
13 }
```

- after successful deletion (CAS succeeds, l. 10), the pointer remains frail
- regular reuse makes the pointer stable again



- unhook the attached element and attach its successor

```
1 inline chain_t *unfix_lfs(chain_t *this) {
2     chain_t *item;
3     do if ((item = pure(this->link)) == 0) break;
4     while (!purge(this, item, pure(item->link)));
5     return item;
6 }
```

Delete mark
a pointer tag

- the actual deletion is done using the following auxiliary function

```
7 inline bool purge(chain_t *this, chain_t *item, chain_t *next) {
8     bool done;
9     CAS(&item->link, next, mark(next, FRAIL));
10    if (!(done = CAS(&this->link, item, next)))
11        CAS(&item->link, mark(next, FRAIL), next);
12    return done;
13 }
```

- after successful deletion (CAS succeeds, l. 10), the pointer remains frail
- regular reuse makes the pointer stable again \rightsquigarrow potential **race condition**



Tagging of Pointer Attributes

- misappropriation of bits of a pointer that are free due to alignment

```
1 typedef enum flag {  
2     FRAIL=(1<<0),    /* pointer should not be used */  
3     DODGY=(1<<1)    /* reusing the pointer is tricky */  
4 } flag_t;
```



Tagging of Pointer Attributes

- misappropriation of bits of a pointer that are free due to alignment

```
1 typedef enum flag {
2     FRAIL=(1<<0),    /* pointer should not be used */
3     DODGY=(1<<1)    /* reusing the pointer is tricky */
4 } flag_t;
```

- simple bit operations based on this for setting, querying and cleaning

```
5 inline long mark(long item, flag_t flag) {
6     return item | flag;
7 }
8
9 inline bool just(long item, flag_t flag) {
10    return item & flag;
11 }
12
13 inline long pure(long item) {
14    return item & ~(FRAIL|DODGY);
15 }
```



- attach an entry to the hook: feasible only if not frail...

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do {
3         item->link = this->link;
4         if (just(item->link, FRAIL))
5             return 0;
6     } while (!CAS(&this->link, item->link, item));
7     return item;
8 }
```

- if the hook entry is just being deleted, it is unclear where else to attach
- in this case, the caller must determine the new insertion point \rightsquigarrow fail



- attach an entry to the hook: feasible only if not frail...

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do {
3         item->link = this->link;
4         if (just(item->link, FRAIL))
5             return 0;
6     } while (!CAS(&this->link, item->link, item));
7     return item;
8 }
```

- overlapping deletion (purge) of the link (*this) leaves a frail pointer
 - then CAS fails because this link pointer has been changed
 - no longer equals its original value (cf. l.3)
 - the operation is retried but then detects the conflict (l.3) and aborts (l.4)



- attach an entry to the hook: feasible only if not frail...

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do {
3         item->link = this->link;
4         if (just(item->link, FRAIL))
5             return 0;
6     } while (!CAS(&this->link, item->link, item));
7     return item;
8 }
```

- overlapping deletion (purge) of the link (*this) leaves a frail pointer

- however, reusing the deleted list item (*this) presents a problem

- the link pointer can soon be purified again



- attach an entry to the hook: feasible only if not frail...

```
1 inline chain_t *infix_lfs(chain_t *this, chain_t *item) {
2     do {
3         item->link = this->link;
4         if (just(item->link, FRAIL))
5             return 0;
6     } while (!CAS(&this->link, item->link, item));
7     return item;
8 }
```

- overlapping deletion (purge) of the link (*this) leaves a frail pointer

- however, reusing the deleted list item (*this) presents a problem

- the link pointer can soon be purified again \rightsquigarrow potential **race condition**



Safe Reuse of a Purged List Entry

- which side (of `unfix`) has to take care of this is quite controversial
 - `caller` only here is it known whether the list item will be reused at all
 - `callee` whether an entry is made in a reused list item is undecidable here



Safe Reuse of a Purged List Entry

- which side (of `unfix`) has to take care of this is quite controversial

callee ■ whether an entry is made in a reused list item is undecidable here

- assume the following sequence of instructions:

```
1 chain_t etc = { 0 };
2 hook = infix(&etc, a); /* etc->a */
3 took = infix(hook, b); /* etc->a->b */
4 node = unfix(hook); /* etc->a, node == b */
5 if (node != 0) /* true: typo "took" */
6     infix(&etc, node); /* etc->b->a */
```



Safe Reuse of a Purged List Entry

- which side (of unfix) has to take care of this is quite controversial

callee ■ whether an entry is made in a reused list item is undecidable here

- assume the following sequence of instructions:

```
1 chain_t etc = { 0 };
2 hook = infix(&etc, a); /* etc->a */
3 took = infix(hook, b); /* etc->a->b */
4 node = unfix(hook); /* etc->a, node == b */
5 if (node != 0) /* true: typo "took" */
6     infix(&etc, node); /* etc->b->a */
```

- as a programming error that has the same effect (from l.2) as:

```
infix(&etc, unfix(&etc)) || infix(&etc, b)
```

– here unfixed list item “a” is reused, put back on the list

- executed in parallel with a potential race condition (cf. p. 14)



Safe Reuse of a Purged List Entry

- which side (of unfix) has to take care of this is quite controversial

callee ■ whether an entry is made in a reused list item is undecidable here

- assume the following sequence of instructions:

```
1 chain_t etc = { 0 };
2 hook = infix(&etc, a); /* etc->a */
3 took = infix(hook, b); /* etc->a->b */
4 node = unfix(hook); /* etc->a, node == b */
5 if (node != 0) /* true: typo "took" */
6     infix(&etc, node); /* etc->b->a */
```

- as a programming error that has the same effect (from l.2) as:

```
infix(&etc, unfix(&etc)) || infix(&etc, b)
```

– here unfixed list item “a” is reused, put back on the list

- executed in parallel with a potential race condition (cf. p. 14)

↪ a solution on the callee side does not seem to help very much...



Safe Reuse of a Purged List Entry

- which side (of `unfix`) has to take care of this is quite controversial
 - `caller` only here is it known whether the list item will be reused at all
 - `callee` whether an entry is made in a reused list item is undecidable here

- each side should regulate their race conditions for themselves
 - this also applies to blocking synchronisation: *separation of concerns* [1]





■ lock-free variant

```
1 inline chain_t *split_lfs(chain_t *this) {
2     chain_t *tail;
3     do tail = this->link;
4     while (!CAS(&this->link, tail, 0));
5     return pure(tail);
6 }
```



- wait-free variant

```
1 inline chain_t *split_wfs(chain_t *this) {  
2     return pure(FAS(&this->link, 0));  
3 }
```



■ lock-free variant

```
1 inline chain_t *split_lfs(chain_t *this) {
2     chain_t *tail;
3     do tail = this->link;
4     while (!CAS(&this->link, tail, 0));
5     return pure(tail);
6 }
```

■ wait-free variant

```
1 inline chain_t *split_wfs(chain_t *this) {
2     return pure(FAS(&this->link, 0));
3 }
```

- both variants are compatible with `infix` (p. 14) and `unfix` (p. 12)
 - no conflicts like when both operations are carried out simultaneously



- a list entry with a frail link is problematic, following it is questionable
 - this entry was or will be deleted soon, its link is therefore undefined



- a list entry with a frail link is problematic, following it is questionable
 - this entry was or will be deleted soon, its link is therefore undefined
- if a frail link is encountered while searching:
 - either start over again and retry (l. 5) or abort with error code (l. 7–8)

```
1 inline chain_t *visit_lfs(chain_t *this, chain_t *item) {
2     chain_t *next, *root = this;
3     while (this && ((next = this->link) != item))
4 #ifdef __FAME_CHAIN_VISIT_RETRY__
5         this = just(next, FRAIL) ? root : next;
6 #else
7         if (just(next, FRAIL)) return -1;
8         else this = next;
9 #endif
10    return this;
11 }
```



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t *item) {  
3     return infix_lfs(visit_lfs(this, 0), item);  
4 }
```



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t *item) {  
3     return infix_lfs(visit_lfs(this, 0), item);  
4 }
```

- delete an entry from the linked list:

```
5 inline chain_t *erase_lfs(chain_t *this, chain_t *item) {  
6     chain_t *hand = visit_lfs(this, item);  
7     if (hand != 0)  
8         hand = purge(hand, item, pure(item->link));  
9     return hand;  
10 }
```



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t *item) {  
3     return infix_lfs(visit_lfs(this, 0), item);  
4 }
```

- delete an entry from the linked list:

```
5 inline chain_t *erase_lfs(chain_t *this, chain_t *item) {  
6     chain_t *hand = visit_lfs(this, item);  
7     if (hand != 0)  
8         hand = purge(hand, item, pure(item->link));  
9     return hand;  
10 }
```

- but that would be too good to be true, it is not that easy
 - the composite operations consist of two complex individual steps:
 1. localisation of the list element that contains the link pointer sought
 2. application of the respective operation to the localised list element
 - both steps happen one after the other



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t *item) {  
3     return infix_lfs(visit_lfs(this, 0), item);  
4 }
```

- delete an entry from the linked list:

```
5 inline chain_t *erase_lfs(chain_t *this, chain_t *item) {  
6     chain_t *hand = visit_lfs(this, item);  
7     if (hand != 0)  
8         hand = purge(hand, item, pure(item->link));  
9     return hand;  
10 }
```

- but that would be too good to be true, it is not that easy
 - the composite operations consist of two complex individual steps:
 1. localisation of the list element that contains the link pointer sought
 2. application of the respective operation to the localised list element
 - both steps happen one after the other \rightsquigarrow prone to **race condition**



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t* item) {  
3     item->link = 0;  
4     do this = visit(this, 0);  
5     while (!CAS(&this->link, 0, item));  
6     return item;  
7 }
```

- after finding the end of the list, it may have been deleted or moved



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- delete an entry from the linked list:

```
8 inline chain_t *erase_lfs(chain_t *this, chain_t *item) {
9     chain_t *hand;
10    do if ((hand = visit(this, item)) == 0) break;
11    while (!purge(hand, item, pure(item->link)));
12    return hand;
13 }
```

- after finding the entry on the list, it may have been deleted



```
1 #define __FAME_CHAIN_VISIT_RETRY__
```

- append an element to the apparent end of the linked list:

```
2 inline chain_t *annex_lfs(chain_t *this, chain_t* item) {  
3     item->link = 0;  
4     do this = visit(this, 0);  
5     while (!CAS(&this->link, 0, item));  
6     return item;  
7 }
```

- delete an entry from the linked list:

```
8 inline chain_t *erase_lfs(chain_t *this, chain_t *item) {  
9     chain_t *hand;  
10    do if ((hand = visit(this, item)) == 0) break;  
11    while (!purge(hand, item, pure(item->link)));  
12    return hand;  
13 }
```



Outline

Preface

Singly Linked List

Working Principle

Concurrent Operation

Collating Sequences

Stack

Queue

Summary



Preliminary Remark

- the access method has a great influence on possible race conditions



Preliminary Remark

- the access method has a great influence on possible race conditions
 - ■ without a specific access pattern; a chain-like approach as before
 - shared variables are all link pointers in the list



Preliminary Remark

- the access method has a great influence on possible race conditions
 - ■ without a specific access pattern; a chain-like approach as before
 - shared variables are all link pointers in the list
 - ↪ when handling the link of any list element, attention must be paid to concurrent processes to the left and right of it



Preliminary Remark

- the access method has a great influence on possible race conditions

LIFO ■ *last in, first out*; a stack-like approach

- shared variable is only the pointer to the list head



Preliminary Remark

- the access method has a great influence on possible race conditions

- LIFO
- *last in, first out*; a stack-like approach
 - shared variable is only the pointer to the list head
 - ↪ attention must be paid only to concurrent processes when handling the link to the list head, all other links in the list are not critical



Preliminary Remark

- the access method has a great influence on possible race conditions

- FIFO
- *first in, first out*; a queue-like approach
 - shared variables are the pointers to the list head and tail, only



- the access method has a great influence on possible race conditions

- FIFO
- *first in, first out*; a queue-like approach
 - shared variables are the pointers to the list head and tail, only
 - ↪ quite similar to LIFO, except that handling the links to both the list head and the list tail has to be considered together



Preliminary Remark

- the access method has a great influence on possible race conditions

LIFO ■ *last in, first out*; a stack-like approach

FIFO ■ *first in, first out*; a queue-like approach

- following topic is singly linked lists with LIFO and FIFO semantics



- a singly-linked list with collating sequence *last in, first out* (LIFO):

```
1 #include "chain.h"
2
3 /* A stack data type, here represented
4  * in terms of a singly-linked list.
5  */
6 typedef struct stack {
7     chain_t head;      /* top of stack: list head */
8 } stack_t;
```



- a singly-linked list with collating sequence *last in, first out* (LIFO):

```
1 #include "chain.h"
2
3 /* A stack data type, here represented
4  * in terms of a singly-linked list.
5  */
6 typedef struct stack {
7     chain_t head;      /* top of stack: list head */
8 } stack_t;
```

- based on this, the following operations are defined:

```
void push(stack_t *this, chain_t *item)
chain_t *pull(stack_t *this)
chain_t *peek(stack_t *this)
```



- a singly-linked list with collating sequence *last in, first out* (LIFO):

```
1 #include "chain.h"
2
3 /* A stack data type, here represented
4  * in terms of a singly-linked list.
5  */
6 typedef struct stack {
7     chain_t head;      /* top of stack: list head */
8 } stack_t;
```

- based on this, the following operations are defined:

```
void push(stack_t *this, chain_t *item)
chain_t *pull(stack_t *this)
chain_t *peek(stack_t *this)
```

- with `this` as the list head, that is, the *stack pointer*,
- `item` as the list entry to be stacked and
- a chain link or list entry as a result, depending on the operation



- simply map onto the lock-free synchronised chaining operations:

```
1 inline void push(stack_t *this, chain_t *item) {
2     infix_lfs(&this->head, item);
3 }
4
5 inline chain_t *pull(stack_t *this) {
6     return unfix_lfs(&this->head);
7 }
```



- simply map onto the lock-free synchronised chaining operations:

```
1 inline void push(stack_t *this, chain_t *item) {
2     infix_lfs(&this->head, item);
3 }
4
5 inline chain_t *pull(stack_t *this) {
6     return unfix_lfs(&this->head);
7 }
```

- however, this mapping does not benefit from the LIFO access method
 - overhead due to functionality that is not required is carried along:
 - tagging of pointer attributes
 - pointer tagging and untagging in `unfix`
 - checking for a frail pointer in `infix`
 - additional work, as no specific access pattern could be assumed



- simply map onto the lock-free synchronised chaining operations:

```
1 inline void push(stack_t *this, chain_t *item) {
2     infix_lfs(&this->head, item);
3 }
4
5 inline chain_t *pull(stack_t *this) {
6     return unfix_lfs(&this->head);
7 }
```

- this kind of **black-box reuse** without being able to derive a benefit from it in non-functional terms is a matter of dispute





- wait-free (synchronised) function

```
1 inline chain_t *peek(stack_t *this) {  
2     return this->head.link;  
3 }
```



■ wait-free (synchronised) function

```
1 inline chain_t *peek(stack_t *this) {
2     return this->head.link;
3 }
```

■ lock-free synchronised functions

```
4 inline void push_lfs(stack_t *this, chain_t *item) {
5     do item->link = peek(this);
6     while (!CAS(&this->head.link, item->link, item));
7 }
```

```
8 inline chain_t *pull_lfs(stack_t *this) {
9     chain_t *item;
10    do if ((item = peek(this)) == 0) break;
11    while (!CAS(&this->head.link, item, item->link));
12    return item;
13 }
```



- wait-free (synchronised) function

```
1 inline chain_t *peek(stack_t *this) {
2     return this->head.link;
3 }
```

- lock-free synchronised functions \rightsquigarrow **conceptual reuse**

```
4 inline void push_lfs(stack_t *this, chain_t *item) {
5     do item->link = peek(this);
6     while (!CAS(&this->head.link, item->link, item));
7 }
```

```
8 inline chain_t *pull_lfs(stack_t *this) {
9     chain_t *item;
10    do if ((item = peek(this)) == 0) break;
11    while (!CAS(&this->head.link, item, item->link));
12    return item;
13 }
```



Data Type III

- a much more complex object of investigation, at a second glance:

```
1 typedef struct queue {
2     chain_t  head;           /* first item */
3     chain_t *tail;         /* insertion point */
4 } queue_t;
```

- the tail pointer addresses the linkage element of a next item to be queued
- it does not directly address the last element in the queue, but indirectly



Data Type III

- a much more complex object of investigation, at a second glance:

```
1 typedef struct queue {
2     chain_t  head;           /* first item */
3     chain_t *tail;         /* insertion point */
4 } queue_t;
```

- consequence is that even an empty queue shows a valid tail pointer:

```
1 inline chain_t *deplete(queue_t *this) {
2     chain_t *list = this->head.link;
3
4     this->head.link = 0;     /* null item */
5     this->tail = &this->head; /* linkage item */
6
7     return list;
8 }
```

- used to reset a queue and at the same time return all its list members



- same **precondition** as before: an item to be queued is not yet queued
 - a simple **first-in, first-out method** (FIFO) is implemented

```
1 inline void enqueue_dos(queue_t *this, chain_t *item) {
2     item->link = 0;           /* finalise chain */
3     this->tail->link = item;  /* append item */
4     this->tail = item;       /* set insertion point */
5 }
```

- 3 ■ the queue head pointer gets set to the first item implicitly



- same **precondition** as before: an item to be queued is not yet queued
 - a simple **first-in, first-out method** (FIFO) is implemented

```
6 inline chain_t *dequeue_dos(queue_t *this) {
7     chain_t *node;
8     if ((node = this->head.link)           /* filled? */
9         && !(this->head.link = node->link)) /* last item? */
10         this->tail = &this->head;        /* reset */
11     return node;
12 }
```

10 ■ the tail pointer must always be valid, even in case of an empty queue



- inspired by the lock-free solution using atomic load/store [9, p. 28]:

```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last;
3
4     item->link = 0;
5
6     do last = this->tail;
7     while (!CAS(&this->tail, last, item));
8
9     last->link = item;
10 }
```



- inspired by the lock-free solution using atomic load/store [9, p. 28]:

```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last;
3
4     item->link = 0;
5
6     do last = this->tail;
7     while (!CAS(&this->tail, last, item));
8
9     last->link = item;
10 }
```

- a **plausibility check** shows correctness as to this overlap pattern:
 - 6 ■ critical shared data is the tail pointer, a local copy is read
 - each overlapping enqueue holds its own copy of the tail pointer
 - 7 ■ validate and, if applicable, write to update the tail pointer
 - the item becomes new fastener for subsequent enqueue operations
 - 9 ■ eventually, the item gets inserted and becomes queue member
 - the assignment operator works on local operands, only



- inspired by the lock-free solution using atomic load/store [9, p. 28]:

```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last;
3
4     item->link = 0;
5
6     do last = this->tail;
7     while (!CAS(&this->tail, last, item));
8
9     last->link = item;
10 }
```

Hint (Onefold Update)

Only a single shared variable needs to be updated in this scenario.

- a **plausibility check** shows correctness as to this overlap pattern:
 - 6 ■ critical shared data is the tail pointer, a local copy is read
 - each overlapping enqueue holds its own copy of the tail pointer
 - 7 ■ validate and, if applicable, write to update the tail pointer
 - the item becomes new fastener for subsequent enqueue operations
 - 9 ■ eventually, the item gets inserted and becomes queue member
 - the assignment operator works on local operands, only



- inspired by the lock-free solution for a stack pull operation (p. 24):

```
1 chain_t *dequeue_lfs(queue_t *this) {
2     chain_t *node;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node, node->link));
6
7     if (node->link == 0)
8         this->tail = &this->head;
9
10    return node;
11 }
```



- inspired by the lock-free solution for a stack pull operation (p. 24):

```
1 chain_t *dequeue_lfs(queue_t *this) {
2     chain_t *node;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node, node->link));
6
7     if (node->link == 0)
8         this->tail = &this->head;
9
10    return node;
11 }
```

- a **plausibility check** shows correctness as to this overlap pattern:
 - 4 ■ critical shared data is the head pointer, a local copy is read
 - each overlapping dequeue holds its own copy of the head element
 - 5 ■ validate and, if applicable, write to update the head pointer
 - 7 ■ each dequeued item is unique, only of them was last in the queue
 - 8 ■ the tail pointer must always be valid, even in case of an empty queue



- inspired by the lock-free solution for a stack pull operation (p. 24):

```
1 chain_t *dequeue_lfs(queue_t *this) {
2     chain_t *node;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node, node->link));
6
7     if (node->link == 0)
8         this->tail = &this->head;
9
10    return node;
11 }
```

Hint (Onefold Update)

Only a single shared variable needs to be updated in this scenario.

- a **plausibility check** shows correctness as to this overlap pattern:
 - 4 ■ critical shared data is the head pointer, a local copy is read
 - each overlapping dequeue holds its own copy of the head element
 - 5 ■ validate and, if applicable, write to update the head pointer
 - 7 ■ each dequeued item is unique, only of them was last in the queue
 - 8 ■ the tail pointer must always be valid, even in case of an empty queue



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element:
enq || *deq*
 - enqueue memorised the chain link of that element
 - dequeue removed that element including the chain link
 - enqueue links the new element using an invalid chain link



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element:
enq || *deq*
 - enqueue memorised the chain link of that element
 - dequeue removed that element including the chain link
 - enqueue links the new element using an invalid chain link↔ **lost enqueue**: linking depends on dequeue progression



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element:

deq || *enq*

- dequeue removed that element and notices “vacancy”
- enqueue appends an element to the one just removed
- dequeue assumes “vacancy” and resets the tail pointer



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element:

deq || *enq*

- dequeue removed that element and notices “vacancy”
- enqueue appends an element to the one just removed
- dequeue assumes “vacancy” and resets the tail pointer

↪ **lost enqueue**: resetting depends on enqueue progression



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element

- enqueue and dequeue must assist each other to solve the problem

- assist without special auxiliary nodes but preferably with simultaneous consideration of **conservative data-structure handling**



- idea is to use the chain-link of a queue element as **auxiliary means** for the interaction between enqueue and dequeue [6]
 - let $last$ be the pointer to the chain link of the queue end tail and
 - let $link_{last}$ be the chain link pointed to by $last$, then:

$$link_{last} = \begin{cases} last, & \text{chain link is valid, was not deleted} \\ 0, & \text{chain link is invalid, was deleted} \\ \text{else,} & \text{chain link points to successor element} \end{cases}$$



- idea is to use the chain-link of a queue element as **auxiliary means** for the interaction between enqueue and dequeue [6]

- $link_{last}$ set to 0 models the per-element “deleted bit” as proposed in [2]
- for a FIFO queue, only the end-tail element needs to carry that “bit”



- idea is to use the chain-link of a queue element as **auxiliary means** for the interaction between enqueue and dequeue [6]

- in contrast to [2], advanced idea is to do without a garbage-collection mechanism to dispose of the “deleted” queue end-tail element
 - purpose is to signal unavailability of the end-tail chain link to enqueue
 - thus, when dequeue is going to remove *last* it attempts to zero $link_{last}$
 - contrariwise, enqueue appends to *last* only if $link_{last}$ still equals *last*



- idea is to use the chain-link of a queue element as **auxiliary means** for the interaction between enqueue and dequeue [6]

- in contrast to [2], advanced idea is to do without a garbage-collection mechanism to dispose of the “deleted” queue end-tail element

- signalling as well as validation can be easily achieved using CAS
 - algorithmic construction versus CDS [3, p. 124] or DCAS [4, p. 4-66]...



```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last, *hook;
3
4     item->link = item;           /* self-reference: hook */
5
6     do hook = (last = this->tail)->link; /* tail end */
7     while (!CAS(&this->tail, last, item));
8
9     if (!CAS(&last->link, hook, item)) /* endpiece? */
10        this->head.link = item;      /* no longer! */
11 }
```



```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last, *hook;
3
4     item->link = item;           /* self-reference: hook */
5
6     do hook = (last = this->tail)->link; /* tail end */
7     while (!CAS(&this->tail, last, item));
8
9     if (!CAS(&last->link, hook, item)) /* endpiece? */
10        this->head.link = item;      /* no longer! */
11 }
```

- validate availability of the ending and potential **volatile chain link**:
 - 9 ■ CAS succeeds only if the last chain link is still a self-reference
 - in that case, the embracing last element was not dequeued
 - 10 ■ CAS fails if the last chain link is no more a self-reference
 - in that case, the embracing last element was dequeued
- ↪ the item to be queued must be head element of the queue, because further enqueues use this very item as leading chain link (l.7)



```
1 chain_t *dequeue_lfs(queue_t *this) {
2     chain_t *node, *next;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node,
6         ((next = node->link) == node ? 0 : next)));
7
8     if (next == node) {          /* self-reference, is last */
9         if (!CAS(&node->link, next, 0)) /* try to help */
10            this->head.link = node->link; /* filled */
11        else CAS(&this->tail, node, &this->head);
12    }
13
14    return node;
15 }
```



```
1 chain_t *dequeue_lfs(queue_t *this) {
2     chain_t *node, *next;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node,
6         ((next = node->link) == node ? 0 : next)));
7
8     if (next == node) {          /* self-reference, is last */
9         if (!CAS(&node->link, next, 0)) /* try to help */
10            this->head.link = node->link; /* filled */
11        else CAS(&this->tail, node, &this->head);
12    }
13
14    return node;
15 }
```

- validate **tail-end invariance** of a one-element queue ($head = tail$):

- 9 ■ CAS fails if the node dequeued no more contains a self-reference
- 10 ■ thus, enqueue happened and left at least one more element queued
- 11 ■ enqueue was assisted and the dequeued node could be last, really



Outline

Preface

Singly Linked List

Working Principle

Concurrent Operation

Collating Sequences

Stack

Queue

Summary



Résumé



- non-blocking synchronisation of **dynamic data structures**, without being able to benefit from well-defined access patterns, is tricky
 - even the singly-linked list presents some difficulties
 - it is not enough to look at individual list elements in isolation
 - rather, entries in the immediate vicinity must also be taken into account
 - for a doubly-linked list, the solutions look quite different



- however, knowledge of **access patterns** is by no means a guarantee for simple and easy-to-use solutions
 - LIFO is simple as there is only a single shared variable in the list
 - the link to the beginning of the list, the “stack pointer”
 - in contrast to FIFO, where two such variables are to be kept consistent
 - the links to the beginning and the ending of the list, the queue pointers
 - but both links only form a critical case if the queue contains a single item



- driving force in the development of the solutions was a **conservative handling** of dynamic data structures
 - reuse of the data structures given with the original sequential variants
 - use of one-word atomic processor instructions such as CAS and FAS



- non-blocking synchronisation of **dynamic data structures**, without being able to benefit from well-defined access patterns, is tricky

- however, knowledge of **access patterns** is by no means a guarantee for simple and easy-to-use solutions

- driving force in the development of the solutions was a **conservative handling** of dynamic data structures



Reference List I

- [1] DIJKSTRA, E. W.:
On the Role of Scientific Thought.
<http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>, Aug. 1974
- [2] HARRIS, T. L.:
A Pragmatic Implementation of Non-blocking Linked-Lists.
In: WELCH, J. (Hrsg.): *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)* Bd. 2180, Springer-Verlag, 2001 (Lecture Notes in Computer Science). –
ISBN 3-540-42605-1, S. 300-314
- [3] IBM CORPORATION (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.
(GA22-7000-4, File No. S/370-01)
- [4] MOTOROLA INC. (Hrsg.):
Motorola M68000 Family Programmer's Reference Manual.
Rev. 1.
Schaumburg, IL, USA: Motorola Inc., 1992.
(M68000PM/AD)



Reference List II

- [5] PARNAS, D. L.:
Designing software for ease of extension and contraction.
In: *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*, ACM, 1978, S. 264–277

- [6] SCHÖN, F. ; SCHRÖDER-PREIKSCHAT, W. :
Lock-Free FIFO Queue Using CAS With Simultaneous Consideration of Conservative Data-Structure Handling.
Febr./März 2009. –
Discourse

- [7] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Concurrent Systems.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides)

- [8] SCHRÖDER-PREIKSCHAT, W. :
Critical Sections.
In: [7], Kapitel 4

- [9] SCHRÖDER-PREIKSCHAT, W. :
"Guarded Sections".
In: [7], Kapitel 10



- [10] SCHRÖDER-PREIKSCHAT, W. :
Non-Blocking Synchronisation.
In: [7], Kapitel 11



```
1 inline chain_t *deplete_wfs(queue_t *this) {
2     chain_t *list = &this->head.link;
3     if (list != 0) {                               /* empty queue */
4         if (!CAS(&this->head.link, list, 0))
5             return 0;
6         this->tail = &this->head;   /* reset queue */
7     }
8     return list;
9 }
```



```
1 inline chain_t *deplete_wfs(queue_t *this) {
2     chain_t *list = &this->head.link;
3     if (list != 0) {                               /* empty queue */
4         if (!CAS(&this->head.link, list, 0))
5             return 0;
6         this->tail = &this->head;   /* reset queue */
7     }
8     return list;
9 }
```

- depletion (dt. Entleerung) of a queue delivers all entries at once, as a singly linked list, and defines the initial state for adding new entries
 - 3 ■ the queue is full, try to take the listed entries as a whole
 - 4 ■ make sure that a filled queue is only emptied once
 - 5 ■ a simultaneous deplete or dequeue causes the operation to end here!
 - 6 ■ define the initial state of an empty queue



```
1 inline chain_t *deplete_wfs(queue_t *this) {
2     chain_t *list = &this->head.link;
3     if (list != 0) {                               /* empty queue */
4         if (!CAS(&this->head.link, list, 0))
5             return 0;
6         this->tail = &this->head;   /* reset queue */
7     }
8     return list;
9 }
```

- depletion (dt. Entleerung) of a queue delivers all entries at once, as a singly linked list, and defines the initial state for adding new entries
 - 3 ■ the queue is full, try to take the listed entries as a whole
 - 4 ■ make sure that a filled queue is only emptied once
 - 5 ■ a simultaneous deplete or dequeue causes the operation to end here!
 - 6 ■ define the initial state of an empty queue
- **line 6** defines a “turning point” for simultaneous enqueue operations
 - entries arriving *beforehand* are still added to the list to be delivered
 - *then* incoming entries are collected in the newly created queue



Epoch-wise Queue Processing

- the problem with the queue operations shown is to ensure that two pointers are changed consistently



Epoch-wise Queue Processing

- the problem with the queue operations shown is to ensure that two pointers are changed consistently
 - both tail and head pointer change element by element per en-/dequeue
 - each enqueue creates an entry for the next “epoch” of queue processing
 - each dequeue removes an entry for processing in the current “epoch”
 - complex list manipulations per operation that have to be synchronised



- assuming only the tail pointer is read and written as usual, while the head pointer is only read and what it points to is only zeroed
 - both epochs use their own list and are isolated from each other
 - the work epoch starts with all the entries listed in the collective epoch



- these two dedicated epochs motivate the following data structure:

```
1 typedef struct epoch {
2     chain_t *work; /* currently processed entries */
3     queue_t next; /* entries for the next period */
4 } epoch_t;
```

- if the `work` list runs empty, it takes over the next queued entries
- the queue is then completely emptied at once, reset to its initial state



Epoch Operations

- in the **collection epoch**, entries are placed in the queue as usual

- the previously collected entries are processed in the **work epoch**



Epoch Operations

- in the **collection epoch**, entries are placed in the queue as usual

```
1 void collate_dos(epoch_t *this, chain_t *item) {  
2     enqueue_dos(&this->next, item);  
3 }
```



- the previously collected entries are processed in the **work epoch**

```
4 chain_t *discard_dos(epoch_t *this) {
5     chain_t *item;
6     if ((item = this->work) != 0)
7         if ((this->work = item->link) == 0)
8             this->work = deplete_dos(&this->next);
9     return item;
10 }
```



- the previously collected entries are processed in the **work epoch**

```
4 chain_t *discard_dos(epoch_t *this) {
5     chain_t *item;
6     if ((item = this->work) != 0)
7         if ((this->work = item->link) == 0)
8             this->work = deplete_dos(&this->next);
9     return item;
10 }
```

- If there is no more work to be done, the queue of entries that have been collected in the meantime is emptied (cf. `deplete`, p. 25)
- that is, all queued entries are placed on the work list and the queue is reset for the next collection epoch



- in the **collection epoch**, entries are placed in the queue as usual

```
1 void collate_dos(epoch_t *this, chain_t *item) {
2     enqueue_dos(&this->next, item);
3 }
```

- the previously collected entries are processed in the **work epoch**

```
4 chain_t *discard_dos(epoch_t *this) {
5     chain_t *item;
6     if ((item = this->work) != 0)
7         if ((this->work = item->link) == 0)
8             this->work = deplete_dos(&this->next);
9     return item;
10 }
```



Collection Epoch

- collation of entries on a list that will be processed in the next epoch
 - it notes the next entries that are received while processing earlier entries
 - the queue (next) provided for this is a separate list from the work list

```
1 void collate_lfs(epoch_t *this, chain_t *item) {
2     chain_t *last;
3
4     do last = this->next.tail;
5     while (!CAS(&last->link, 0, item));
6
7     CAS(&this->next.tail, last, item);
8 }
```



Collection Epoch

- collation of entries on a list that will be processed in the next epoch
 - it notes the next entries that are received while processing earlier entries
 - the queue (next) provided for this is a separate list from the work list

```
1 void collate_lfs(epoch_t *this, chain_t *item) {
2     chain_t *last;
3
4     do last = this->next.tail;
5     while (!CAS(&last->link, 0, item));
6
7     CAS(&this->next.tail, last, item);
8 }
```

- 4–5 ■ make sure that the next entry (item) is queued at the end: FIFO
- try again in case of a simultaneous collate() or discard()
 - do this until the new last entry has been committed¹

¹A *spin on read* mode of operation makes sense if contention is too high.



Collection Epoch

- collation of entries on a list that will be processed in the next epoch
 - it notes the next entries that are received while processing earlier entries
 - the queue (`next`) provided for this is a separate list from the work list

```
1 void collate_lfs(epoch_t *this, chain_t *item) {
2     chain_t *last;
3
4     do last = this->next.tail;
5     while (!CAS(&last->link, 0, item));
6
7     CAS(&this->next.tail, last, item);
8 }
```

- 7
- commit the new last entry of the queue if there is no conflict
 - this action releases any polling process from the loop (l. 4–5)
 - which also applies when a new work epoch starts (cf. l. 6, p.38)



- collation of entries on a list that will be processed in the next epoch
 - it notes the next entries that are received while processing earlier entries
 - the queue (next) provided for this is a separate list from the work list

```
1 void collate_lfs(epoch_t *this, chain_t *item) {
2     chain_t *last;
3
4     do last = this->next.tail;
5     while (!CAS(&last->link, 0, item));
6
7     CAS(&this->next.tail, last, item);
8 }
```

- 4–5 ■ make sure that the next entry (item) is queued at the end: FIFO
 - try again in case of a simultaneous collate() or discard()
 - do this until the new last entry has been committed¹
- 7 ■ commit the new last entry of the queue if there is no conflict
 - this action releases any polling process from the loop (l. 4–5)
 - which also applies when a new work epoch starts (cf. l. 6, p.38)

¹A *spin on read* mode of operation makes sense if contention is too high.



- processing of entries on a work list collated in the previous epoch
 - it takes the queue entries to be processed from its own, separated list
 - this list will only be shortened, but not extended

```
1 chain_t *discard_wfs(epoch_t *this) {
2     chain_t *item;
3     if ((item = this->work) != 0)
4         if (CAS(&this->work, item, item->link))
5             if (CAS(&item->link, 0, item)) /* ran empty */
6                 this->work = deplete_wfs(&this->next);
7             else CAS(&this->work, 0, item->link);
8     return item;
9 }
```



Work Epoch

- processing of entries on a work list collated in the previous epoch
 - it takes the queue entries to be processed from its own, separated list
 - this list will only be shortened, but not extended

```
1 chain_t *discard_wfs(epoch_t *this) {
2     chain_t *item;
3     if ((item = this->work) != 0)
4         if (CAS(&this->work, item, item->link))
5             if (CAS(&item->link, 0, item)) /* ran empty */
6                 this->work = deplete_wfs(&this->next);
7             else CAS(&this->work, 0, item->link);
8     return item;
9 }
```

- 3 ■ there is at least one entry on the work list
- 4 ■ delete the entry from the work list if it is still there
- 5-6 ■ try to get new entries when the work list has been emptied
 - make sure that the last entry cannot be followed by another entry
 - end the current collection epoch, accept all entries at once
- 7 ■ the last entry suddenly got a successor (l. 5, p. 41), note it
- 8 ■ deliver the next entry from the work list, if there is still one



Work Epoch

- processing of entries on a work list collated in the previous epoch
 - it takes the queue entries to be processed from its own, separated list
 - this list will only be shortened, but not extended—with one exception

```
1 chain_t *discard_wfs(epoch_t *this) {
2     chain_t *item;
3     if ((item = this->work) != 0)
4         if (CAS(&this->work, item, item->link))
5             if (CAS(&item->link, 0, item)) /* ran empty */
6                 this->work = deplete_wfs(&this->next);
7             else CAS(&this->work, 0, item->link);
8     return item;
9 }
```

- 3 ■ there is at least one entry on the work list
- 4 ■ delete the entry from the work list if it is still there
- 5–6 ■ try to get new entries when the work list has been emptied
 - make sure that the last entry cannot be followed by another entry
 - end the current collection epoch, accept all entries at once
- 7 ■ the last entry suddenly got a successor (l. 5, p. 41), note it !
- 8 ■ deliver the next entry from the work list, if there is still one



- processing of entries on a work list collated in the previous epoch
 - it takes the queue entries to be processed from its own, separated list
 - this list will only be shortened, but not extended—**with one exception**

```
1 chain_t *discard_wfs(epoch_t *this) {
2     chain_t *item;
3     if ((item = this->work) != 0)
4         if (CAS(&this->work, item, item->link))
5             if (CAS(&item->link, 0, item)) /* ran empty */
6                 this->work = deplete_wfs(&this->next);
7             else CAS(&this->work, 0, item->link);
8     return item;
9 }
```

- 3 ■ there is at least one entry on the work list
- 4 ■ delete the entry from the work list if it is still there
- 5–6 ■ try to get new entries when the work list has been emptied
 - make sure that the last entry cannot be followed by another entry
 - end the current collection epoch, accept all entries at once
- 7 ■ the last entry suddenly got a successor (l. 5, p. 41), note it !
- 8 ■ deliver the next entry from the work list, if there is still one

