Concurrent Systems

Exercise 05 – Non-Blocking Synchronization

Stefan Reif

2018-01-15

Agenda

Non-Blocking Data Structures

Wait for a Condition

Synchronization with Futexes

Assignment 5

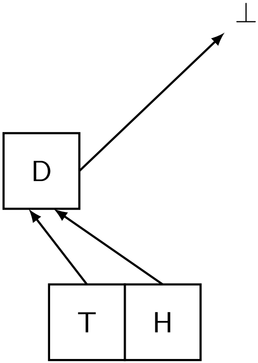
Non-blocking queue

- Many queue algorithms have been published
  - ... through scientific channels, e.g. [2, 3, 6]
  - ... through non-scientific channels, e.g. [4]

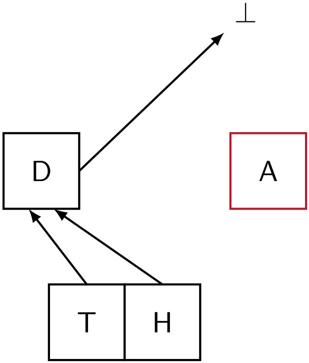
- Minimal interface
  - At least `enqueue()` and `dequeue()`
  - Sometimes `is_empty()`, `make_empty()`

- Differences between lock-free queue algorithms [5]
  - Single-producer ↔ Multi-producer
  - Single-consumer ↔ Multi-consumer
  - Lock-free ↔ Wait-free
  - Nonintrusive ↔ intrusive
  - Node-based ("Unbounded") ↔ Array-based ("bounded") ↔ Hybrid
  - Consistency guarantees
  - ...

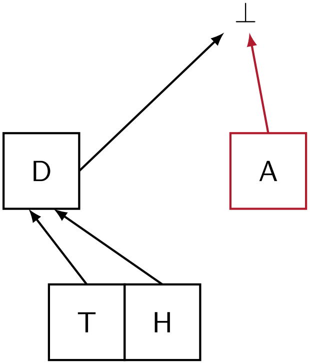
Queue implementation



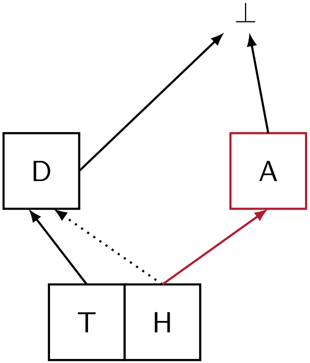
Queue implementation



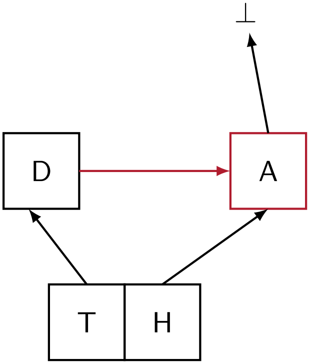
Queue implementation



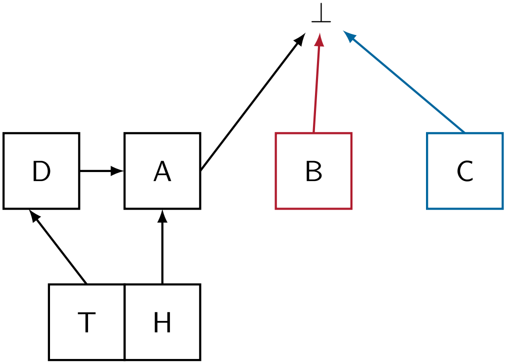
Queue implementation



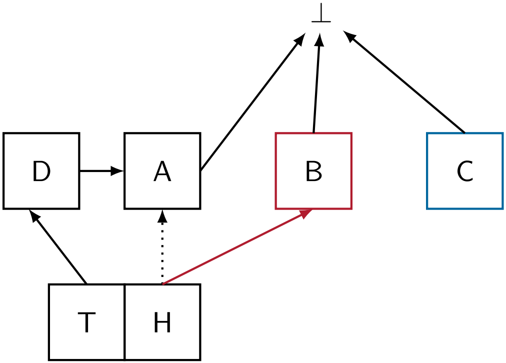
Queue implementation



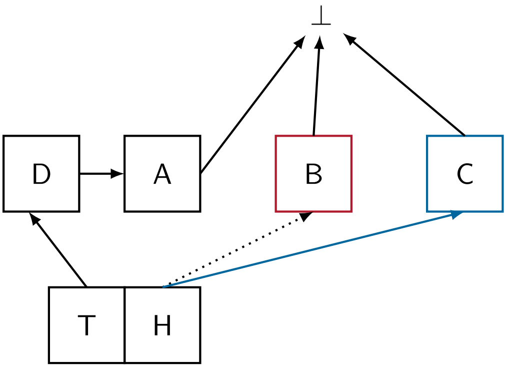
Queue implementation



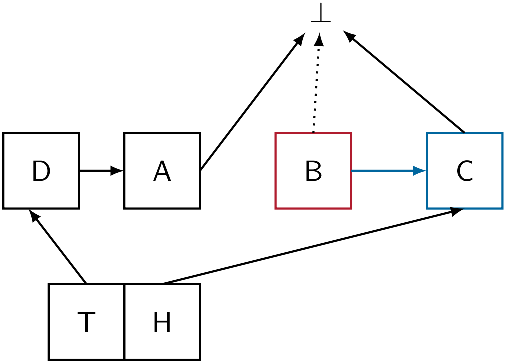
Queue implementation



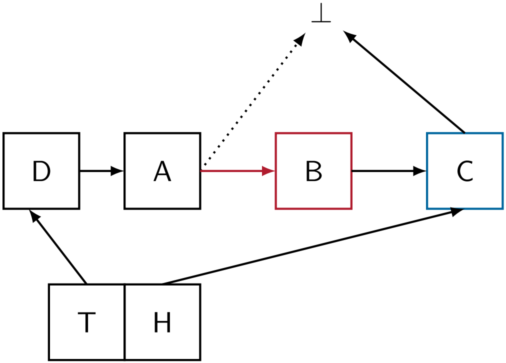
Queue implementation



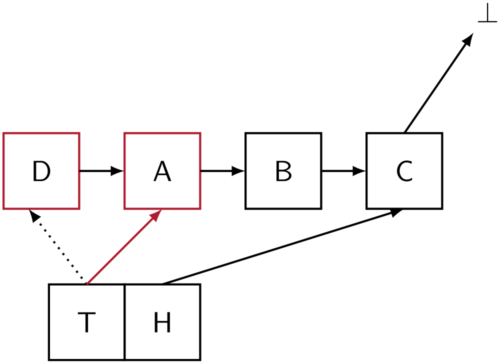
Queue implementation



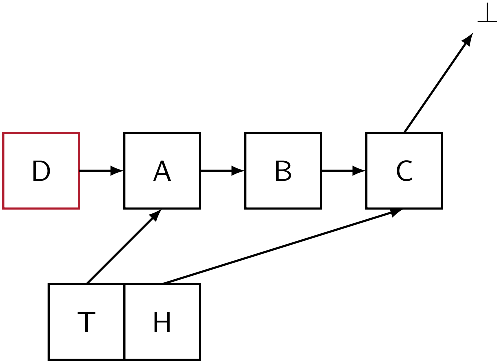
Queue implementation



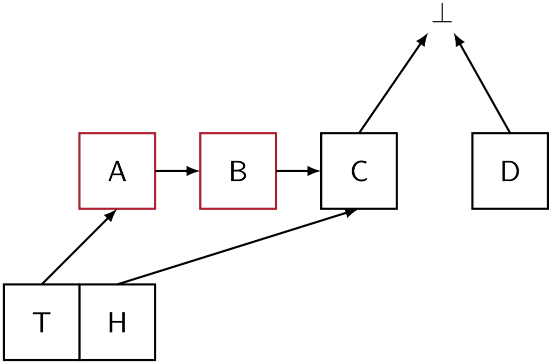
Queue implementation



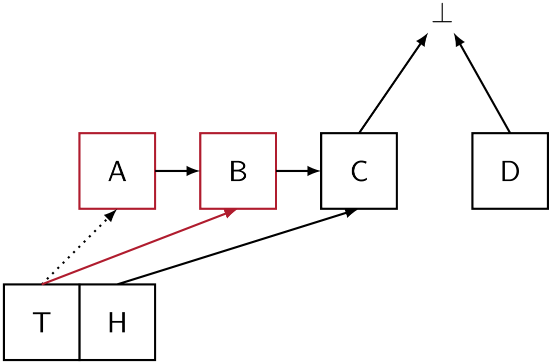
Queue implementation



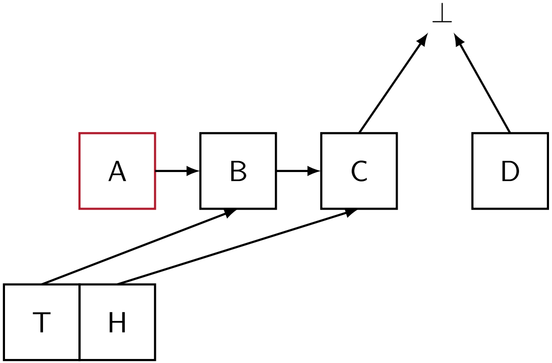
Queue implementation



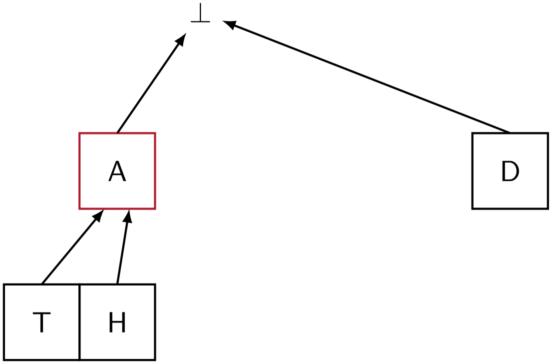
Queue implementation



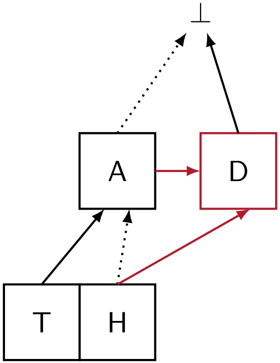
Queue implementation



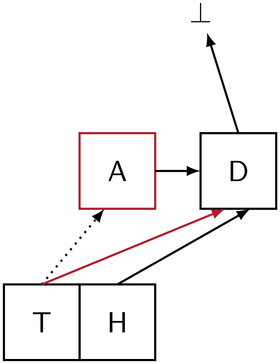
Queue implementation



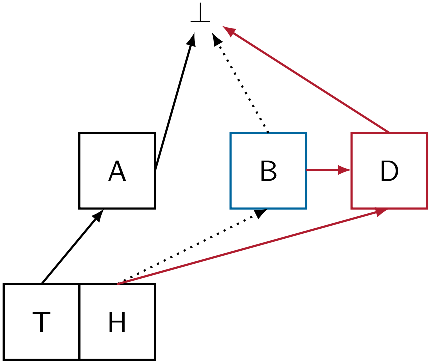
Queue implementation



Queue implementation



Queue implementation



Wait for a condition

- Actively or Inactively ("passive")
  - Does the thread release the processor core?
  - Different performance, energy characteristics, ...
- Active waiting ...
  - Is usually fast, except for cache contention
  - Causes relatively high energy consumption
  - Probably wastes CPU time
- Passive waiting ...
  - Tends to be more energy-efficient
  - Supports many-threaded system

  $\rightarrow$ More threads than cores

Eager event

- No thread management data

```c
typedef struct event {
  atomic_int state;
} event;
```

- Eager check for condition

```c
void event_await(event *e) {
  while (PENDING == atomic_load(&e->state)); ← acquire
}
```

- Notification by assignment

```c
void event_signal(event *e) {
  atomic_store(&e->state, PAST); ← release
}
```

Passive waiting done wrong

- How not to wait for a condition

```
void event_await(event *e) {
  while (PENDING == atomic_load(&e->state))
    sleep();
}
```

✗

- Lost-wakeup problem
  - Critical section between `load()` and `sleep()`
  - Concurrent notification is lost
  - Need for atomic `test_and_sleep()` operation

Passive waiting

How to wait for a condition passively?

- Data race between condition check and sleep instruction
- Atomic operation required
  - x86: `monitor`, `mwait` (privileged [1])
  - Linux: `futex()` system call

Linux `futex` interface

```
syscall(SYS_futex, addr, FUTEX_WAIT, exp, &timeout);
syscall(SYS_futex, addr, FUTEX_WAIT, exp, NULL);
syscall(SYS_futex, addr, FUTEX_WAKE, num_threads);
...
```

Problem of spurious wakeup

- Threads can wake up despite not being signalled
- Need for re-evaluation of waiting conditions

Linux futex semantics

- Wait until a variable changes

```
void futex_wait(int *addr, int exp) {
  if (*addr == exp)
    sleep_on(addr);
}
```

- Address-dependent wakeup function

```
void futex_wake(int *addr) {
  for (Thread *t ← Threads)
    if (t->waiting_on == addr)
      wakeup(t);
}
```

Linux futex semantics

- Wait until a variable changes

```
void futex_wait(int *addr, int exp) {
  if (*addr == exp)        ] atomic
    sleep_on(addr);        ]
}
```

- Address-dependent wakeup function

```
void futex_wake(int *addr) {
  for (Thread *t ← Threads)
    if (t->waiting_on == addr)
      wakeup(t);
}
```

Linux futex usage pattern

- futex usage pattern

```
int i = atomic_load(&v);
futex_wait(&v, i);
```

```
atomic_store(&v, j);
futex_wake(&v);
```

- Critical Section
  - Between `atomic_load()` and `futex_wait()`
  - Concurrent signal must be caught

Linux futex usage pattern

- futex usage pattern

```
int i = atomic_load(&v);
futex_wait(&v, i);
```

```
atomic_store(&v, j);
futex_wake(&v);
```

- Critical Section
  - Between `atomic_load()` and `futex_wait()`
  - Concurrent signal must be caught

- futex overlapping patterns

```
int i = atomic_load(&v);
futex_wait(&v, i);
```

```


atomic_store(&v, j);
futex_wake(&v);
```

Linux futex usage pattern

- futex usage pattern

```
int i = atomic_load(&v);
futex_wait(&v, i);
```

```
atomic_store(&v, j);
futex_wake(&v);
```

- Critical Section
  - Between atomic_load() and futex_wait()
  - Concurrent signal must be caught

- futex overlapping patterns

```
int i = atomic_load(&v);

futex_wait(&v, i);
```

```

atomic_store(&v, j);

futex_wake(&v);
```

Linux futex usage pattern

- futex usage pattern

```
int i = atomic_load(&v);
futex_wait(&v, i);
```

```
atomic_store(&v, j);
futex_wake(&v);
```

- Critical Section
  - Between atomic_load() and futex_wait()
  - Concurrent signal must be caught

- futex overlapping patterns

```
int i = atomic_load(&v);


futex_wait(&v, i);
```

```

atomic_store(&v, j);
futex_wake(&v);

```

Simple futex-based event

- Glibc has no direct SYS_futex wrapper

```c
#define futex(...) syscall(SYS_futex, __VA_ARGS__)
```

- wait-operation using a futex

```c
void event_await(event *e) {
  while (PENDING == atomic_load(&e->state))
    futex((int *) &e->state,
          FUTEX_WAIT, PENDING, NULL);
}
```

- signal-operation on a futex

```c
void event_signal(event *e) {
  atomic_store(&e->state, PAST);
  futex((int *) &e->state, FUTEX_WAKE, INT_MAX);
}
```

Simple futex-based event

- Glibc has no direct SYS_futex wrapper

```
#define futex(...) syscall(SYS_futex, __VA_ARGS__)
```

- wait-operation using a futex

```
void event_await(event *e) {
  while (PENDING == atomic_load(&e->state))
    futex((int *) &e->state,
          FUTEX_WAIT, PENDING, NULL);
}
```

- signal-operation on a futex

```
void event_signal(event *e) {
  atomic_store(&e->state, PAST);
  futex((int *) &e->state, FUTEX_WAKE, INT_MAX);
}
```

- ABA-Problem?

→ reset() operation available?

ABA-safe `futex`-based latch

- Use larger state space

```
#define is_pending(x) (!((x)&1))
```

- Conditionally wait on variable change

```
void event_await(event *e) {
  int val = atomic_load(&e->state);
  if (!is_pending(val))
    return;
  do {
    futex((int *) &e->state, FUTEX_WAIT, val, NULL);
  } while (atomic_load(&e->state) == val);
}
```

ABA-safe `futex`-based latch (2)

- Conditionally increment state variable

```
void event_signal(event *e) {
  int val = atomic_load(&e->state);
  if (!is_pending(val)) return;
  if (atomic_compare_exchange_strong(&e->state,
                                      &val, val + 1))
    futex((int *) &e->state, FUTEX_WAKE, INT_MAX);
}
void event_reset(event *e) {
  int val = atomic_load(&e->state);
  if (is_pending(val)) return;
  atomic_compare_exchange_strong(&e->state,
                                  &val, val + 1);
}
```

Worker thread waiting condition

- Similar to event implementation
  - Unilateral synchronization
  - Correctness depends on job queue consistency

- Implementation variants
  - `sensitivity` flag
    $\rightarrow$ Wait while `sensitive` $\neq 0$
  - `load` counter
    $\rightarrow$ Wait for `load` $\neq 0$
  - `generation` counter
    $\rightarrow$ Wait for `generation` increment
  - ...

Worker thread waiting condition – sensitivity flag

- Get next Job

```
job *next() {
  atomic_store(&sensitive, 1); // single consumer!
  job *n = dequeue();
  if (n)
    return atomic_store(&sensitive, 0), n;
  futex((int *) &sensitive, FUTEX_WAIT, 1, NULL);
  return next();
}
```

- Submit a job

```
void submit(job *j) {
  enqueue(j); int old = 1;
  if (atomic_compare_exchange_strong(&sensitive,
                                     &old, 0))
    futex((int *) &sensitive, FUTEX_WAKE, 1);
}
```

Worker thread waiting condition – load counter

- Get next Job

```c
job *next(){
  while (!atomic_load(&load))
    futex((int *) &load, FUTEX_WAIT, 0, NULL);
  atomic_fetch_sub(&load, 1); // single consumer!
  return dequeue();
}
```

- Submit a job

```c
void submit(job *j) {
  enqueue(j);
  if (!atomic_fetch_add(&load, 1))
    futex((int *) &sensitive, FUTEX_WAKE, 1);
}
```

Worker thread waiting condition – generation counter

- Get next Job

```
job *next() {
  int now = atomic_load(&generation);
  job *n = dequeue();
  if (n)
    return n;
  futex((int *) &generation, FUTEX_WAIT, now, NULL);
  return next();
}
```

- Submit a job

```
void submit(job *j) {
  enqueue(j);
  atomic_fetch_add(&generation, 1);
  futex((int *) &generation, FUTEX_WAKE, INT_MAX);
}
```

Assignment 5

- Rewrite LWT without locks
  - Remove all scheduler locks ...
  - ... but still provide high-level synchronization for LWT threads
- Tackle the ABA problem
  - Use version counters where needed
- All LWT synchronization mechanisms use futexes
  - Internal `lwt_futex` data structure
  - Used by mutexes, condition variables, semaphore, `join()`, ...
  - Only one mechanism for passive waiting / thread blocking
- Worker threads use the `futex` system call for passive waiting
  - Sleep when no `lwt_thread` is ready

Reference List I

[1] INTEL CORPORATION (Hrsg.):
Intel Architecture Software Developer's Manual.
Santa Clara, California, USA: Intel Corporation, 2016

[2] MICHAEL, M. M. ; SCOTT, M. L.:
Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms.
In: *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, 1996, S. 267–275

[3] MORRISON, A. ; AFEK, Y. :
Fast Concurrent Queues for x86 Processors.
In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM Press, 2013, S. 103–112

[4] VYUKOV, D. :
Intrusive MPSC node-based queue.
`http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue`,

[5] VYUKOV, D. :
Producer-Consumer Queues.
`http://www.1024cores.net/home/lock-free-algorithms/queues`,

Reference List II

[6] YANG, C. ; MELLOR-CRUMMEY, J. :
A Wait-free Queue As Fast As Fetch-and-add.
In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM Press, 2016, S. 16:1–16:13