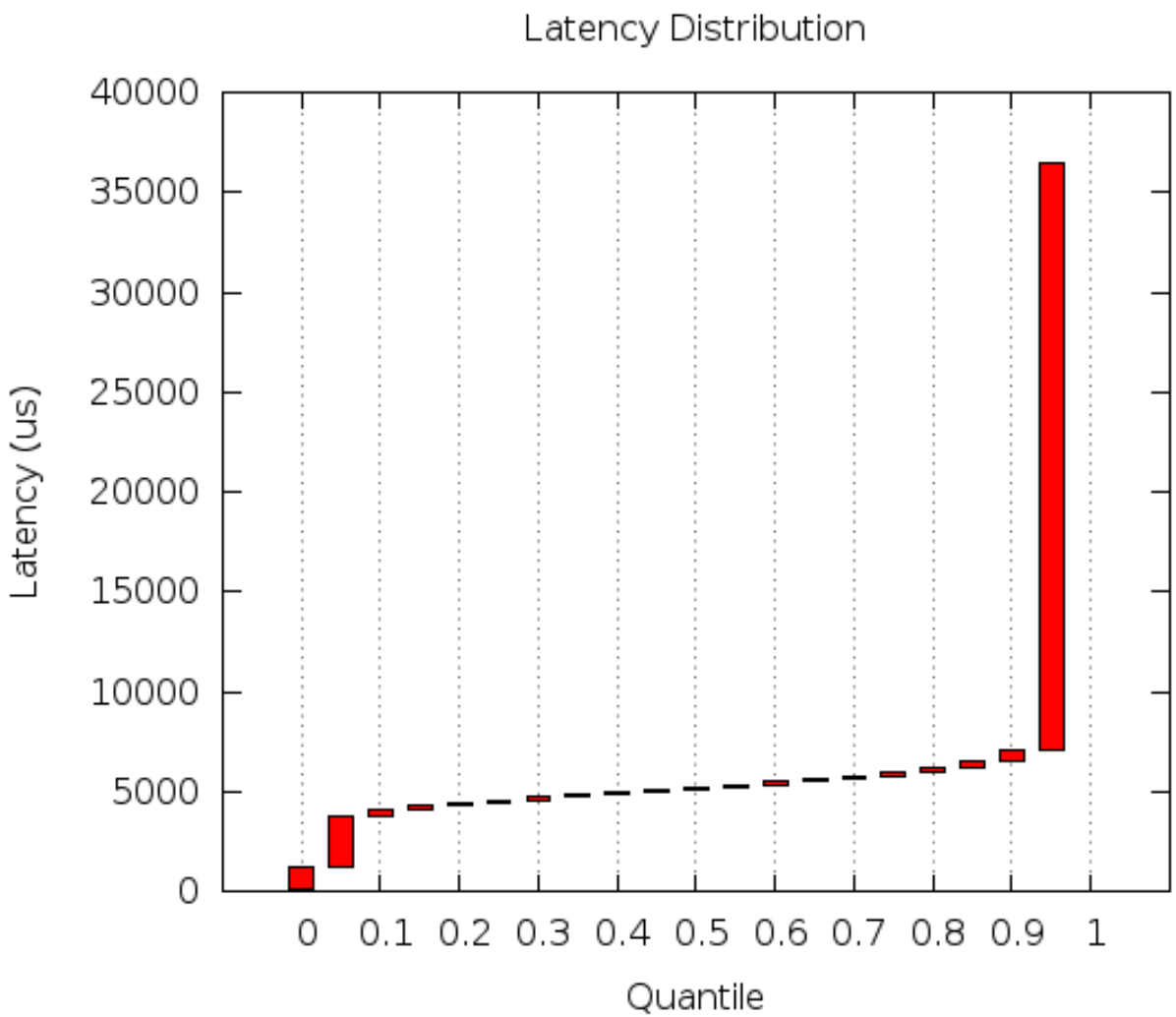
Introduction to Lock-Free Algorithms

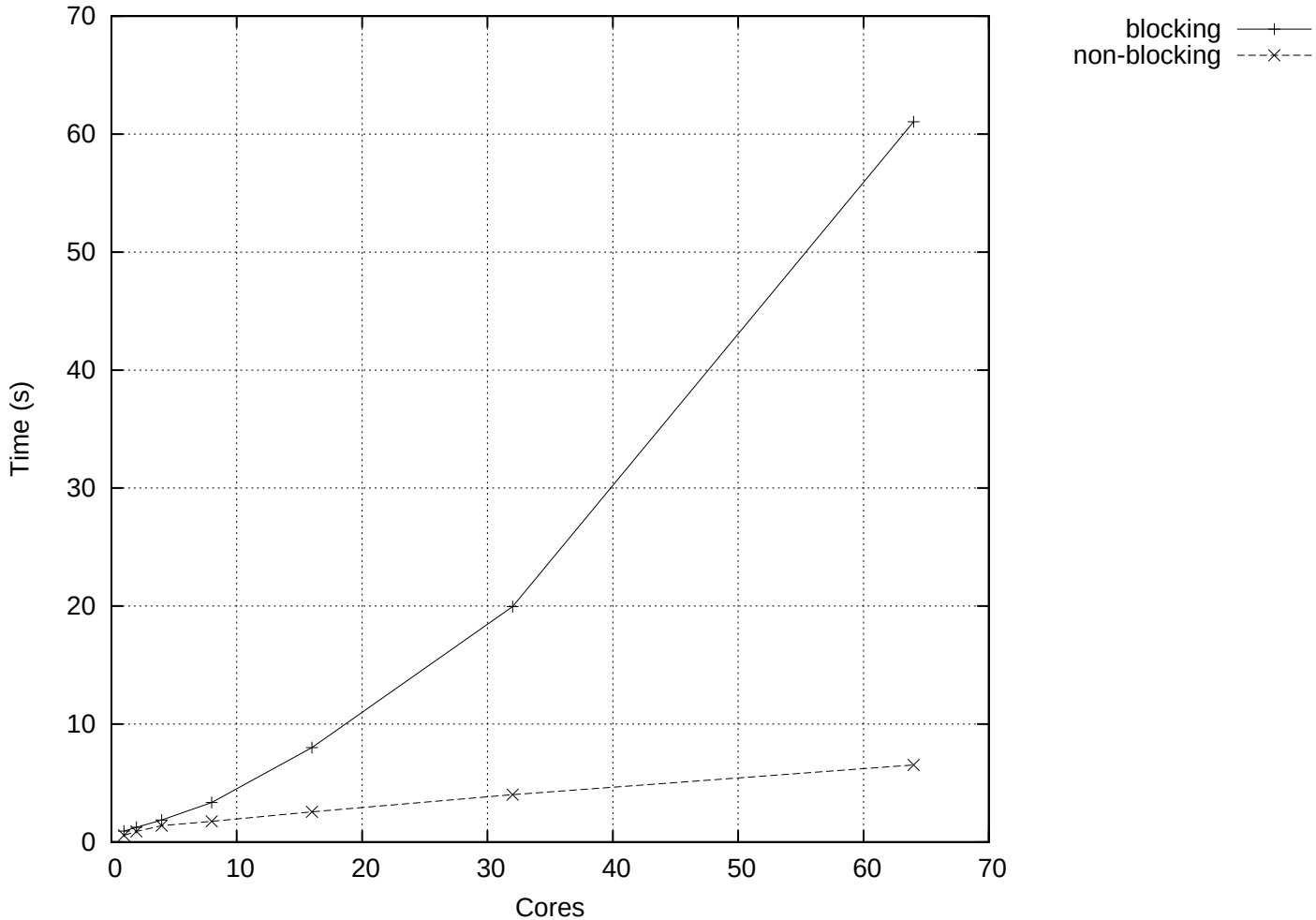
Through a case study

Samy Al Bahra
AppNexus, Inc.
September 23, 2012

Motivation


Latency Distribution
Latency (us)
40000
35000
30000
25000
20000
15000
10000
5000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Quantile

Motivation



Motivation

- Non-blocking data structures are not a silver bullet
- Contention is always a problem

Workload	push	pop
spinlock	19	24
upmc	33	29
mpmc	33	35
mpnc	17	NA

Motivation

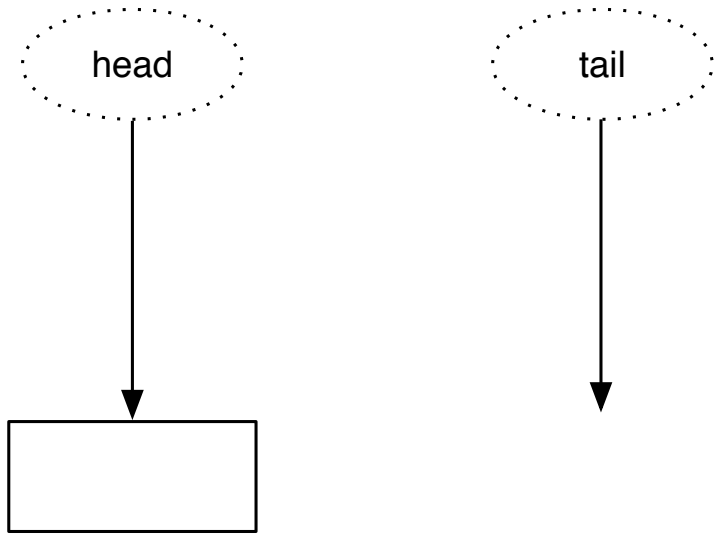
- Non-blocking data structures can provide system-wide *progress* guarantees where blocking synchronization cannot
  - Composability
  - Predictability
  - Termination-Safety
- Non-blocking data structures are heavily-dependent on underlying memory model and support for atomic operations

Case Study

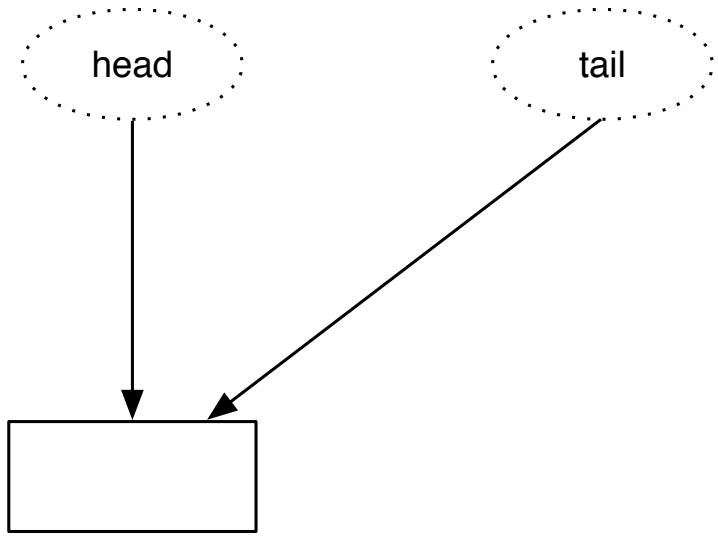
- Concurrent FIFO for single-producer/single-consumer
- Concurrent FIFO for multi-producer/multi-consumer
- Will involve exploration of:
  - Atomic operations
  - Memory models
  - Safe Memory Reclamation
  - ABA Problem
- First let us review typical FIFO algorithm.

Case Study

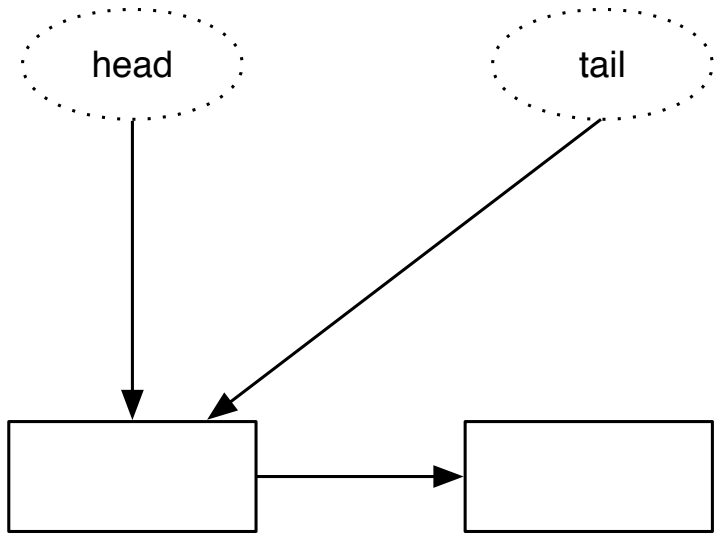
Case Study



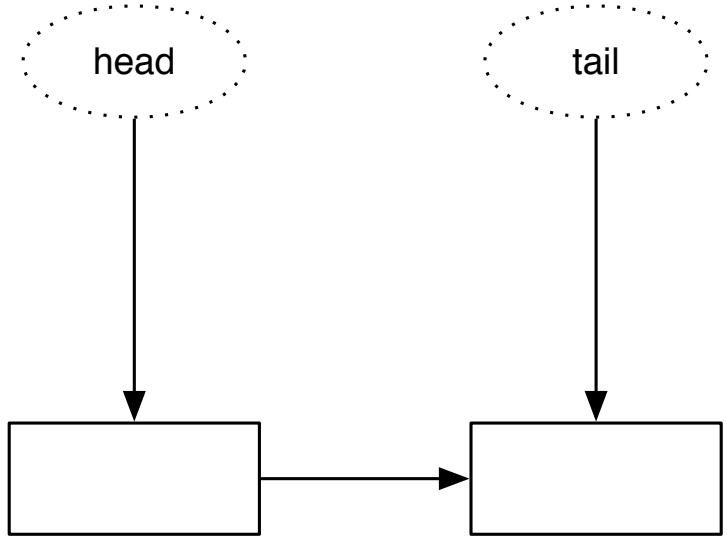
Case Study



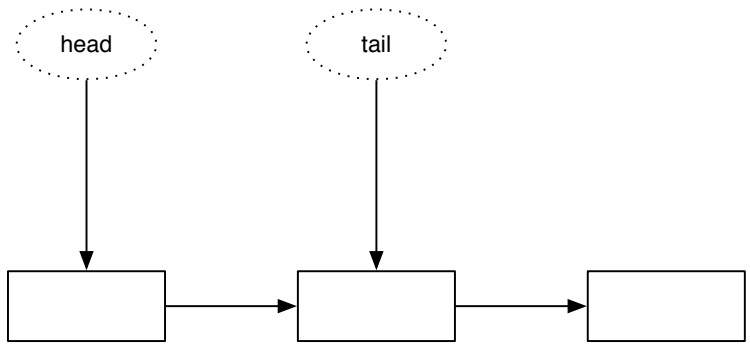
Case Study



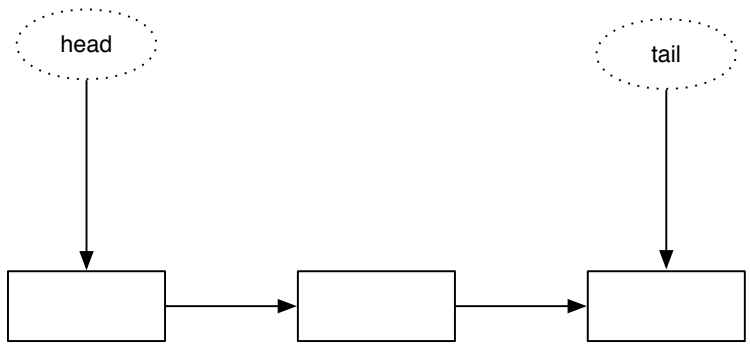
Case Study



Case Study



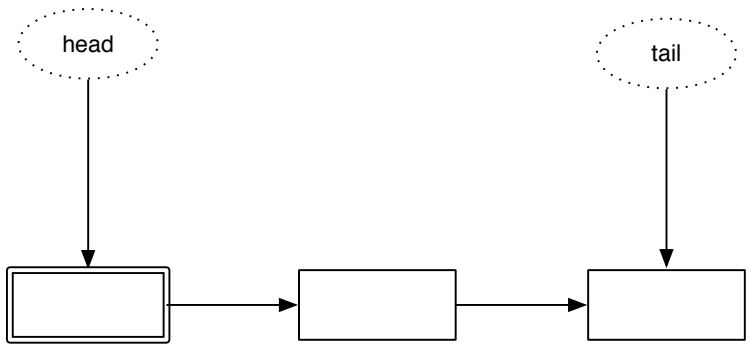
Case Study



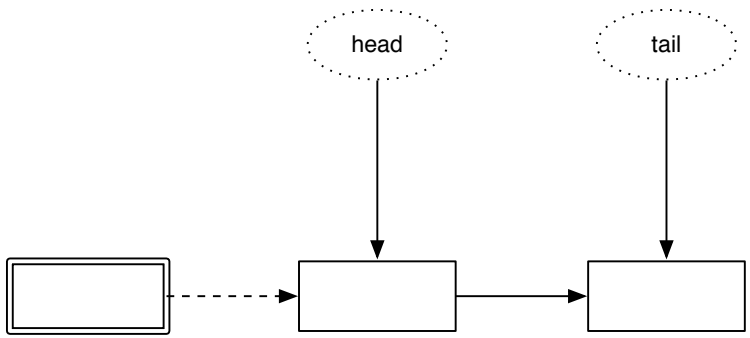
Case Study

```
struct fifo {
    struct node *head;
    struct node *tail;
};

struct node {
    void *value;
    struct node *next;
};

void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    if (f->tail == NULL) {
        f->head = n;
        f->tail = n;
    } else {
        struct node *tail = f->tail;

        tail->next = n;
        f->tail = n;
    }

    return;
}
```

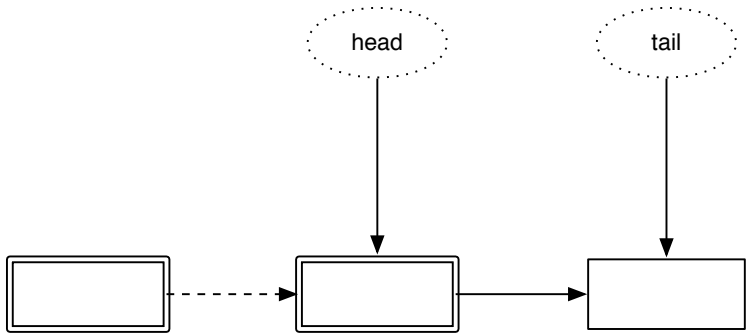
Case Study



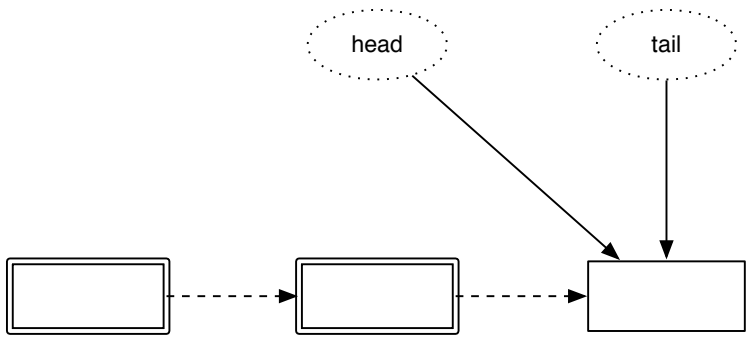
Case Study



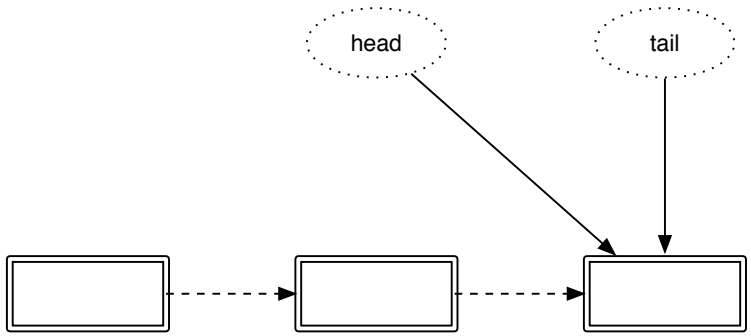
Case Study



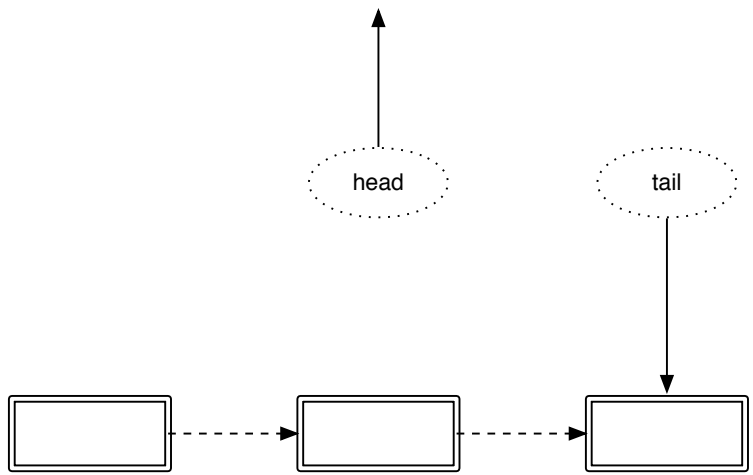
Case Study



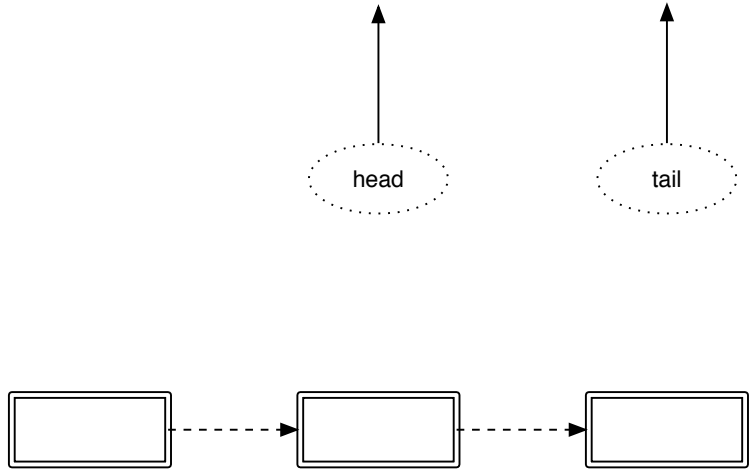
Case Study



Case Study



Case Study



Case Study

```
struct fifo {
    struct node *head;
    struct node *tail;
};

struct node {
    void *value;
    struct node *next;
};

struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *n = f->head;

    if (n == NULL)
        return NULL;

    f->head = n->next;
    if (f->head == NULL)
        f->tail = NULL;

    *value = n->value;
    return n;
}
```

Case Study

Thread A

```
struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *n = f->head;

    if (n == NULL)
        return NULL;

    f->head = n->next;
    if (f->head == NULL)
        f->tail = NULL;

    *value = n->value;
    return n;
}


/* Assume queue is empty. */
```

Thread B

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    if (f->tail == NULL) {
        f->head = n;
        f->tail = n;
    } else {
        struct node *tail = f->tail;

        tail->next = n;
        f->tail = n;
    }

    return;
}
```

Case Study

Thread A

```
struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *n = f->head;

    if (n == NULL)
        return NULL;

    f->head = n->next; <-- Assigns head to NULL
    if (f->head == NULL)
        f->tail = NULL;

    *value = n->value;
    return n;
}


/* Assume queue is empty. */
```

Thread B

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    if (f->tail == NULL) {
        f->head = n; <-- Preemption before next
        f->tail = n;
    } else {
        struct node *tail = f->tail;

        tail->next = n;
        f->tail = n;
    }

    return;
}
```

Case Study

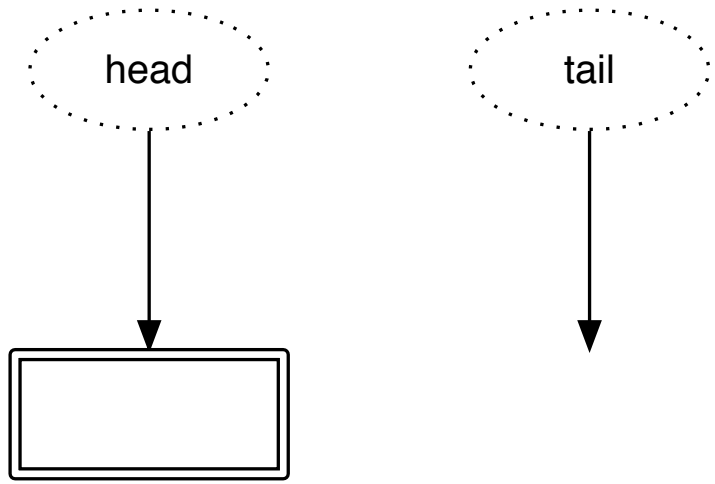
Thread A

```
struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *n = f->head;

    if (n == NULL)
        return NULL;

    f->head = n->next;
    if (f->head == NULL)
        f->tail = NULL; <-- This first

    *value = n->value;
    return n;
}


/* State: fifo.head = NULL, fifo.tail = n */
```

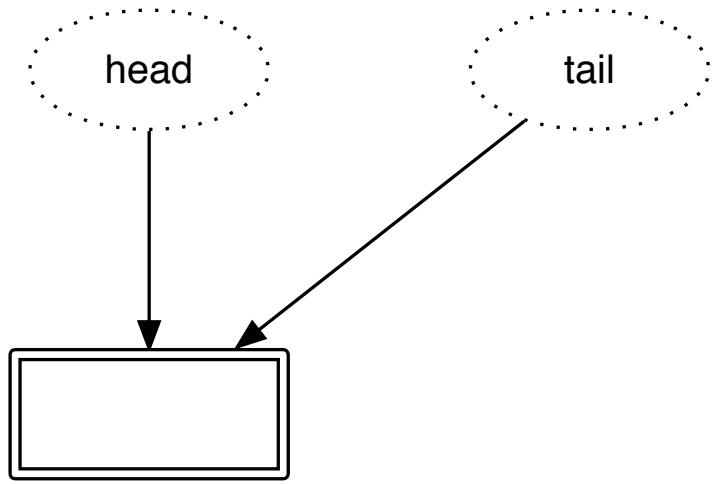
Thread B

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    if (f->tail == NULL) {
        f->head = n;
        f->tail = n; <-- Then this
    } else {
        struct node *tail = f->tail;

        tail->next = n;
        f->tail = n;
    }

    return;
}
```

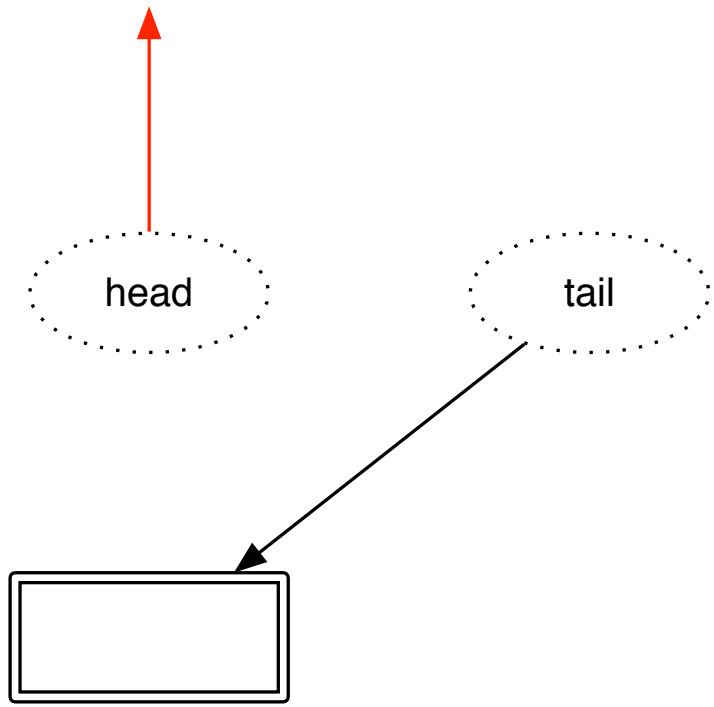
Case Study



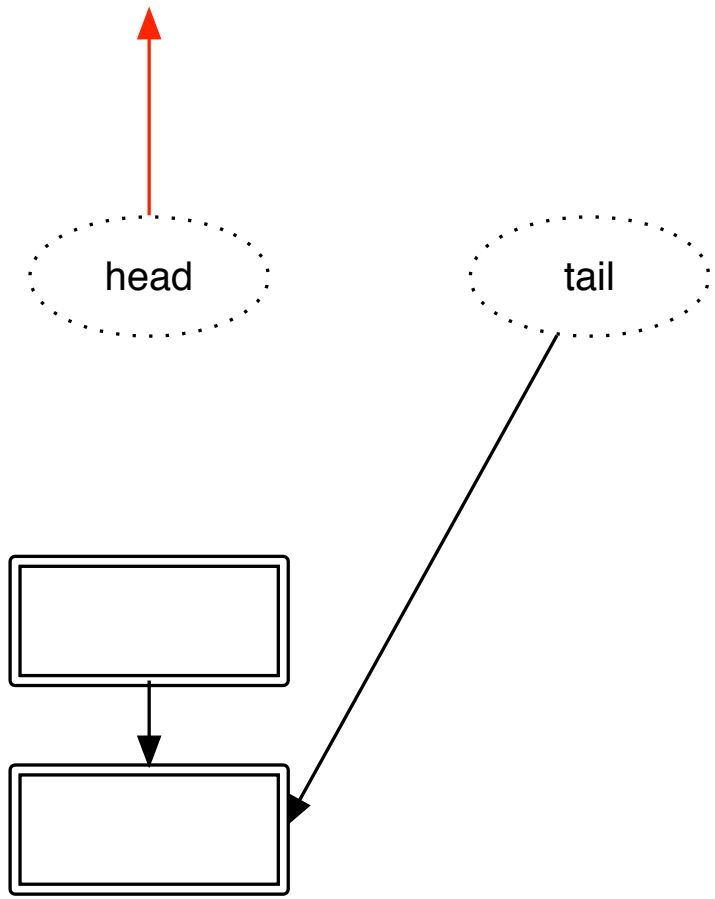
Case Study



Case Study



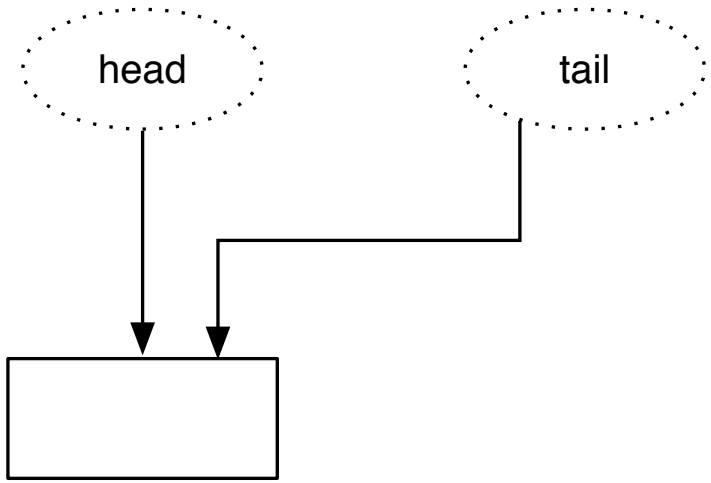
Case Study



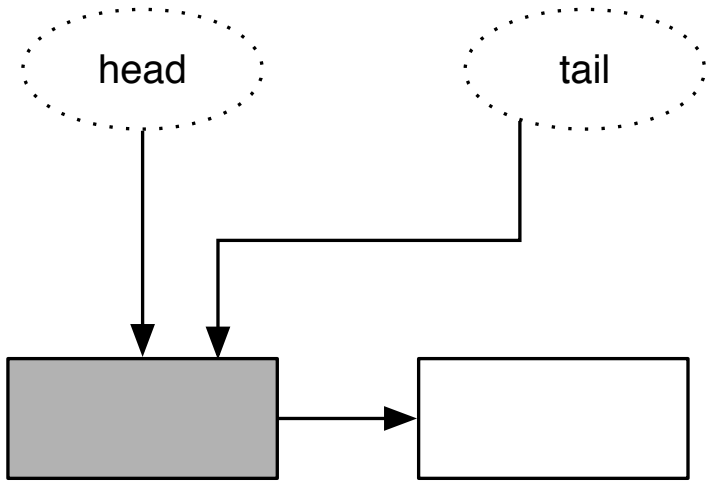
Case Study

- Modification of tail and head pointer must be atomic in the case of an empty queue.
- Non-empty queue can be retrofitted so that consumer only touches head and producer only touches tail.

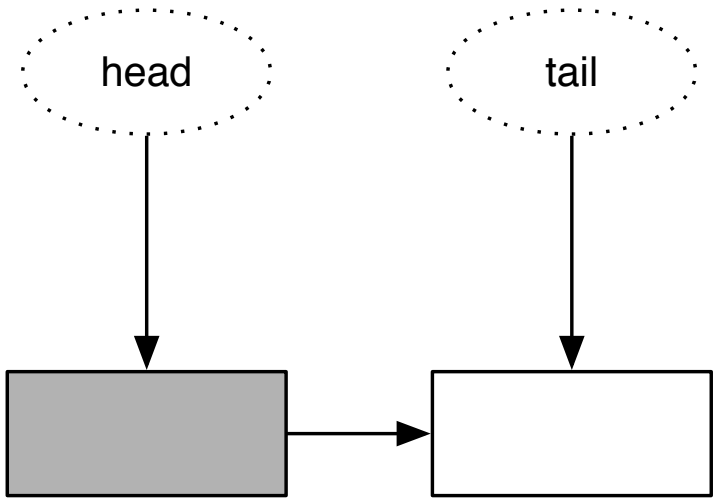
Case Study



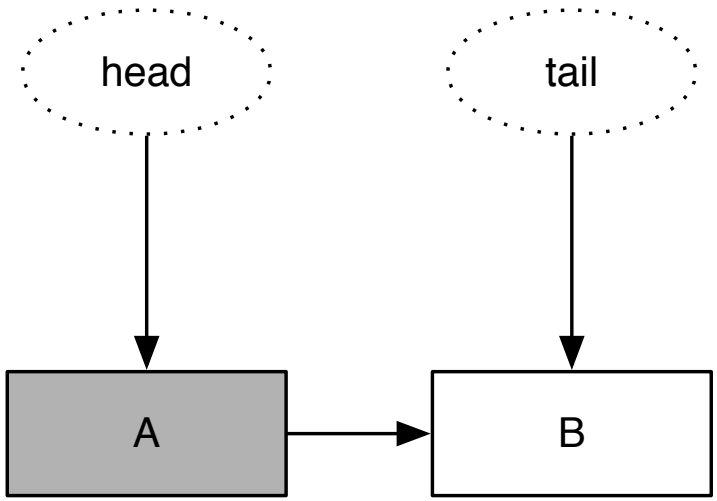
Case Study



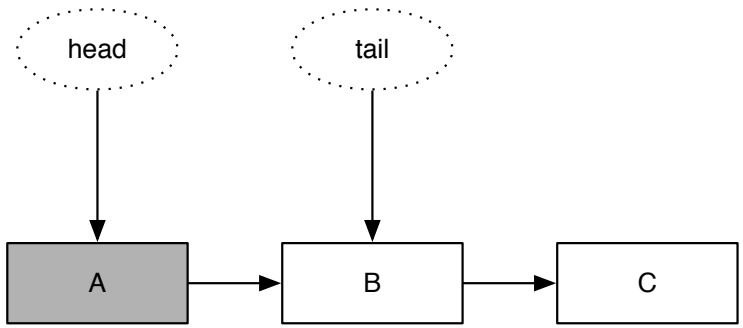
Case Study



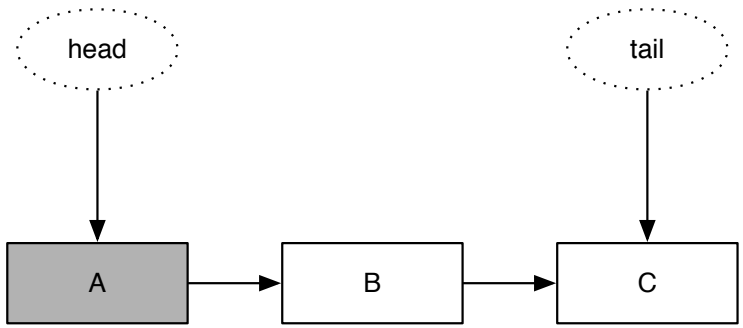
Case Study



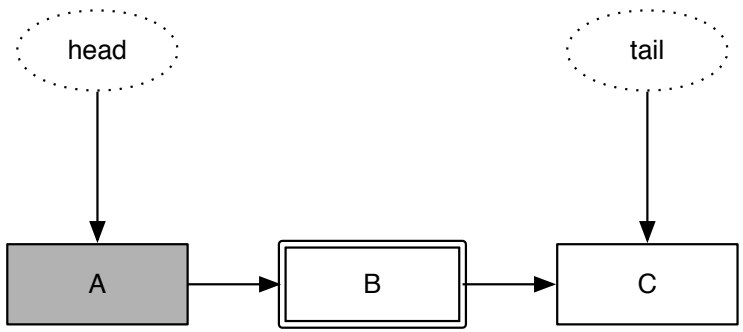
Case Study



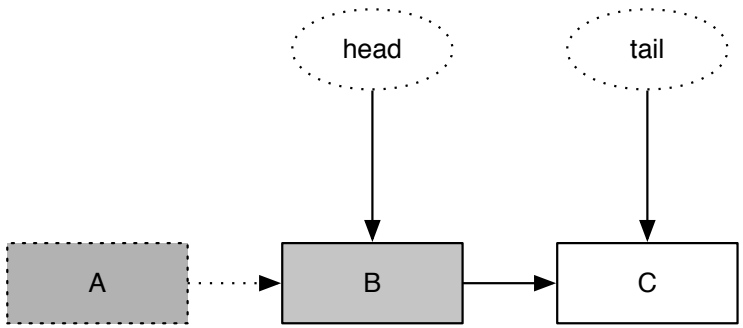
Case Study



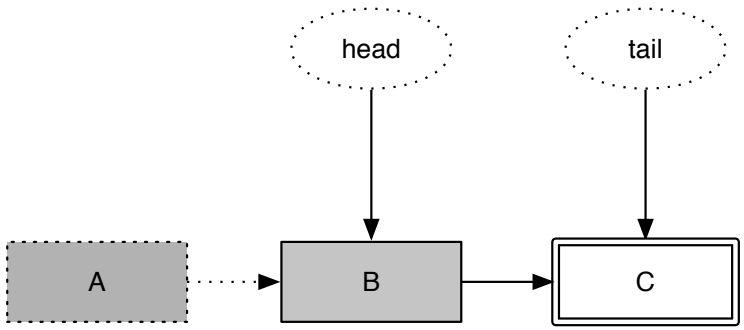
Case Study



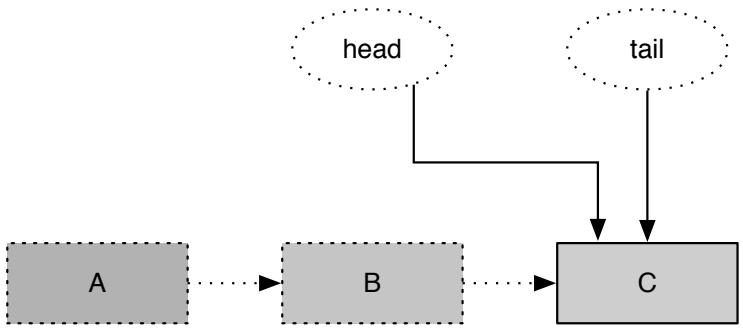
Case Study



Case Study



Case Study



Case Study

```
struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = p->next;

    if (n == NULL)
        return NULL;

    *value = n->value;
    f->head = n;
    return p;
}
```

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    f->tail->next = n;
    f->tail = n;
    return;
}
```

Case Study

- C99 specification does not define a concurrent memory consistency model for compliant execution environments
- C11 provides this through the stdatomic interface, but is still not inclusive of all memory ordering semantics

```
#include <stdio.h>
#include <pthread.h>

static unsigned int signal;

static void *
wait_for_signal(void *unused)
{

        signal = 1;

        puts("[T] Testing signal.");

        while (signal == 1);

        puts("[T] We have received the
      signal.");
        return (NULL);
}
```

```
int
main(int argc, char *argv[])
{
        pthread_t a;
        unsigned int i;

        pthread_create(&a, NULL,
      wait_for_signal, NULL);

        for (i = 0; i < 10; i++) {
                puts("[M] Setting signal...");
                sleep(1);
                signal = 0;
        }
        pthread_join(a, NULL);

        return (0);
}
```

Case Study

```
(gdb) disas wait_for_signal
Dump of assembler code for function wait_for_signal:
0x0000000100000e30 <wait_for_signal+0>: push   %rbp
0x0000000100000e31 <wait_for_signal+1>: mov    %rsp,%rbp
0x0000000100000e34 <wait_for_signal+4>: movb   $0x1,0x245(%rip)        # 0x100001080 <signal.b>
0x0000000100000e3b <wait_for_signal+11>: lea    0x96(%rip),%rdi        # 0x100000ed8
0x0000000100000e42 <wait_for_signal+18>: callq  0x100000e84 <dyld_stub_puts>
0x0000000100000e47 <wait_for_signal+23>: mov    0x233(%rip),%al        # 0x100001080 <signal.b>
0x0000000100000e4d <wait_for_signal+29>: cmp    $0x1,%al
0x0000000100000e4f <wait_for_signal+31>: jne    0x100000e62 <wait_for_signal+50>
0x0000000100000e51 <wait_for_signal+33>: nopl   0x0(%rax)
0x0000000100000e58 <wait_for_signal+40>: nopl   0x0(%rax,%rax,1)
0x0000000100000e60 <wait_for_signal+48>: jmp    0x100000e60 <wait_for_signal+48>
0x0000000100000e62 <wait_for_signal+50>: lea    0x87(%rip),%rdi        # 0x100000ef0
0x0000000100000e69 <wait_for_signal+57>: callq  0x100000e84 <dyld_stub_puts>
0x0000000100000e6e <wait_for_signal+62>: xor    %eax,%eax
0x0000000100000e70 <wait_for_signal+64>: pop    %rbp
0x0000000100000e71 <wait_for_signal+65>: retq
End of assembler dump.
```

Case Study

Volatile Semantics

"An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment ... The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type."

- There are no visibility or atomicity guarantees provided by volatile
- Rely on atomic primitives library (C11 has a standardized atomics interface)

Case Study

```
#include <ck_pr.h>
#include <stdio.h>
#include <pthread.h>

static unsigned int signal;

static void *
wait_for_signal(void *unused)
{

        ck_pr_store_uint(&signal, 1);

        puts("[T] Testing signal.");

        while (ck_pr_load_uint(&signal) == 1);

        puts("[T] We have received the
      signal.");
        return (NULL);
}
```

```
int
main(int argc, char *argv[])
{
        pthread_t a;
        unsigned int i;

        pthread_create(&a, NULL,
      wait_for_signal, NULL);

        for (i = 0; i < 10; i++) {
                puts("[M] Setting signal...");
                sleep(1);
                ck_pr_store_uint(&signal, 0);
        }
        pthread_join(a, NULL);

        return (0);
}
```

Case Study

```
(gdb) disas wait_for_signal
Dump of assembler code for function wait_for_signal:
0x0000000100000e30 <wait_for_signal+0>: push   %rbp
0x0000000100000e31 <wait_for_signal+1>: mov    %rsp,%rbp
0x0000000100000e34 <wait_for_signal+4>: movl   $0x1,0x242(%rip)        # 0x100001080 <signal>
0x0000000100000e3e <wait_for_signal+14>: lea    0x8b(%rip),%rdi        # 0x100000ed0
0x0000000100000e45 <wait_for_signal+21>: callq  0x100000e7e <dyld_stub_puts>
0x0000000100000e4a <wait_for_signal+26>: nopw   0x0(%rax,%rax,1)
0x0000000100000e50 <wait_for_signal+32>: mov    0x22a(%rip),%eax        # 0x100001080 <signal>
0x0000000100000e56 <wait_for_signal+38>: cmp    $0x1,%eax
0x0000000100000e59 <wait_for_signal+41>: je     0x100000e50 <wait_for_signal+32>
0x0000000100000e5b <wait_for_signal+43>: lea    0x86(%rip),%rdi        # 0x100000ee8
0x0000000100000e62 <wait_for_signal+50>: callq  0x100000e7e <dyld_stub_puts>
0x0000000100000e67 <wait_for_signal+55>: xor    %eax,%eax
0x0000000100000e69 <wait_for_signal+57>: pop    %rbp
0x0000000100000e6a <wait_for_signal+58>: retq
End of assembler dump.
```

Case Study

```
struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

    *value = ck_pr_load_ptr(&n->value);
    f->head = n;
    return p;
}
```

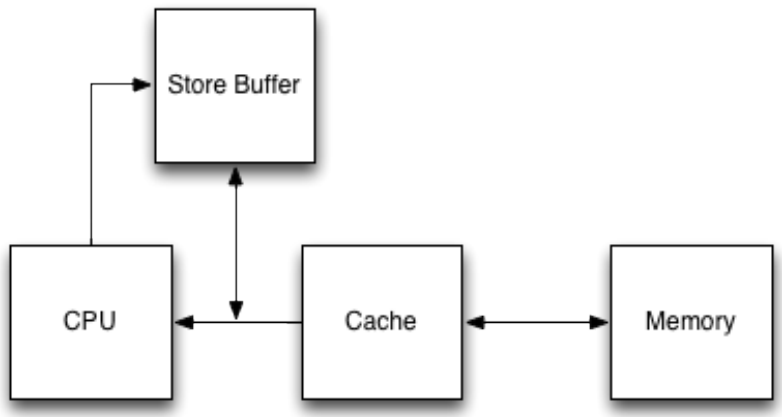
```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    ck_pr_store_ptr(&f->tail->next, n);
    f->tail = n;
    return;
}
```

Memory Ordering

- Defines the semantics of memory operation visibility and ordering.
- Modern processors with intruction-level parallelism are free to re-order operations according to the underlying model.

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

1:    n->value = v;
2:    n->next = NULL;
3:    ck_pr_store_ptr(&f->tail->next, n);
4:    f->tail = n;
      return;
}
```

Memory Ordering



Memory Ordering

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

4:    n->value = v;
3:    n->next = NULL;
1:    ck_pr_store_ptr(&f->tail->next, n);
2:    f->tail = n;
      return;
}

struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

    *value = ck_pr_load_ptr(&n->value);
    f->head = n;
    return p;
}
```

Memory Ordering

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

1:    n->value = v;
2:    n->next = NULL;
->    ck_pr_fence_store();
3:    ck_pr_store_ptr(&f->tail->next, n);
4:    f->tail = n;
      return;
}
```

Memory Ordering

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

2:    n->value = v;
1:    n->next = NULL;
->    ck_pr_fence_store();
3:    ck_pr_store_ptr(&f->tail->next, n);
4:    f->tail = n;
      return;
}
```

Memory Ordering

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

2:    n->value = v;
1:    n->next = NULL;
->    ck_pr_fence_store();
4:    ck_pr_store_ptr(&f->tail->next, n);
3:    f->tail = n;
      return;
}
```

Memory Ordering

- A store fence guarantees partial ordering with respect to stores before and after the fence
- A store fence may imply a store buffer flush
- A store fence is also referred to as a write barrier

$$(A, B, C) <_t (D, E, F)$$

```
A
B
C
ck_pr_fence_store();
D
E
F
```

Memory Ordering

- Store barriers are typically paired with load barriers.
- Load fence may imply an invalidation queue flush
- Load barriers provide a partial ordering with respect to loads before and after the load fence.

$$(A, B, C) <_t (D, E, F)$$

```
A
B
C
ck_pr_fence_load();
D
E
F
```

Memory Ordering

```
struct node *
dequeue(struct fifo *f, void **value)
{
1:  struct node *p = f->head;
2:  struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

3:   ck_pr_fence_load();
4:   *value = ck_pr_load_ptr(&n->value);
N:   f->head = n;
     return p;
}
```

Memory Ordering

- Store fence provides ordering guarantees on store operations.
- Load fence provides ordering guarantees on load operations.
- A "normal" or "full" fence provides ordering guarantees on all memory operations.

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    ck_pr_fence_store();
    ck_pr_store_ptr(&f->tail->next, n);
    f->tail = n;
    return;
}

struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

    ck_pr_fence_load();
    *value = ck_pr_load_ptr(&n->value);
    f->head = n;
    return p;
}
```

Memory Ordering

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    ck_pr_fence_memory();
    ck_pr_store_ptr(&f->tail->next, n);
    f->tail = n;
    return;
}

struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

    ck_pr_fence_memory();
    *value = ck_pr_load_ptr(&n->value);
    f->head = n;
    return p;
}
```

Memory Ordering

Memory models in the real world.

Memory Model	load|load	load|store	store|store	store|load	atomic
TSO	no	no	no	yes	no
PSO	no	yes	yes	yes	stores
RMO	yes	yes	yes	yes	stores/loads

Architecture	TSO	PSO	RMO
ARM	no	no	yes
x86	yes	no	yes*
POWER	no	no	yes
SPARCv9	yes*	yes	yes

Memory Ordering

Memory Model	load|load	load|store	store|store	store|load	atomic
TSO	no	no	no	yes	no
PSO	no	yes	yes	yes	stores
RMO	yes	yes	yes	yes	stores/loads

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{

    n->value = v;
    n->next = NULL;
    ck_pr_fence_store();
    ck_pr_store_ptr(&f->tail->next, n);
    f->tail = n;
    return;
}

struct node *
dequeue(struct fifo *f, void **value)
{
    struct node *p = f->head;
    struct node *n = ck_pr_load_ptr(&p->next);

    if (n == NULL)
        return NULL;

    ck_pr_fence_load(); <- Is this too heavy-weight?
    *value = ck_pr_load_ptr(&n->value);
    f->head = n;
    return p;
}
```

Atomic Operations

- What progress guarantees do we provide?
- If we wanted to generalize this to multiple-producers and multiple-consumers, what atomic instructions should we probably use?
- Atomic operation placement in consensus hierarchy determines which N-process consensus problem the atomic operation can solve.

Operation	Consensus Number
compare_and_swap,ll/sc	∞
fetch_and_ϕ	2
load/store	1

Atomic Operations

- Since we would like to support any number of processes for our data structures, then compare_and_swap is a good first choice.

Compare and Swap

```
bool
ck_pr_cas_ptr(void **target, void *compare, void *update)
{
    atomically {
        if (*target == compare) {
            *target = update;
            return true;
        }
    }

    return false;
}
```

Case Study

```
void
enqueue(struct fifo *f, struct node *n, void *v)
{
    struct node *tail, *next;

    n->v = v;
    n->next = NULL;
    ck_pr_fence_store();

    for (;;) {
        tail = ck_pr_load_ptr(&f->tail);
        ck_pr_fence_load();
        next = ck_pr_load_ptr(&tail->next);

        if (next == NULL) {
            if (ck_pr_cas_ptr(&tail->next, NULL, n) == true)
                break;
        } else {
            ck_pr_cas_ptr(&f->tail, tail, next);
        }
    }

    ck_pr_fence_store();
    ck_pr_cas_ptr(&f->tail, tail, n);
    return;
}
```

Case Study

```
struct node *
dequeue(struct fifo *f, void **v)
{
    struct node *head, *tail, *next;

    for (;;) {
        head = ck_pr_load_ptr(&f->head);
        tail = ck_pr_load_ptr(&f->tail);
        next = ck_pr_load_ptr(&head->next);

        if (head == tail) {
            if (next == NULL)
                return NULL;

            ck_pr_cas_ptr(&f->tail, tail, next);
        } else {
            *v = ck_pr_load_ptr(&next->value);
            if (ck_pr_cas_ptr(&fifo->head, head, next) == true)
                break;
        }
    }

    return head;
}
```

Non-Blocking Hierarchy

In broad strokes...

- Wait-freedom: Per-operation progress guarantees. Every operation is guaranteed to complete in a finite number of steps.
- Lock-freedom: Per-object progress guarantees. It is guaranteed that some operation which has already started will always complete successfully in a finite number of steps.
- Obstruction-freedom: A single thread is guaranteed to make progress if other threads are suspended. Partially completed operations must be aborted.

$$WF \subset LF \subset OF$$

Safe Memory Reclamation

```
struct fifo fifo;

void
reader(void)
{
    struct node *garbage;
    void *value;

    for (;;) {
        garbage = dequeue(&fifo, &value);
        if (garbage == NULL)
            continue;

        free(garbage);
    }

    return;
}

void
writer(void)
{
    struct node *m;
    uintptr_t i = 0;

    for (;;) {
        m = malloc(sizeof *m);
        enqueue(&fifo, m, (void *)i++);
    }

    return;
}
```

Safe Memory Reclamation

```
struct fifo fifo;
unsigned int ref;

void
reader(void)
{
    struct node *garbage;
    void *value;

    for (;;) {
        garbage = dequeue(&fifo, &value);
        if (garbage == NULL)
            continue;

        while (ck_pr_load_uint(&ref) != 0)
            /* Let's wait until reference count is 0. */;

        free(garbage);
    }

    return;
}
```

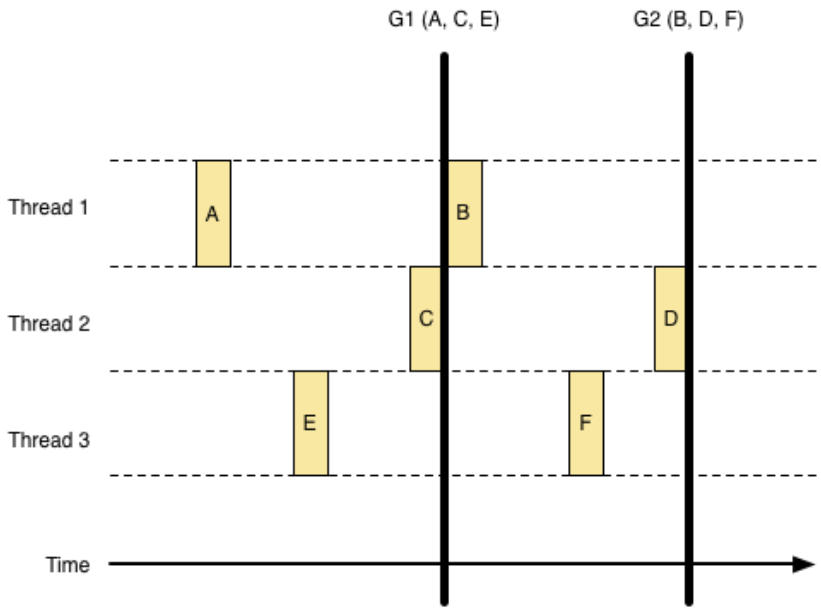
```
void
writer(void)
{
    struct node *m;
    uintptr_t i = 0;

    for (;;) {
        m = malloc(sizeof *m);
        ck_pr_inc_uint(&ref);
        enqueue(&fifo, m, (void *)i++);
        ck_pr_dec_uint(&ref);
    }

    return;
}
```

Safe Memory Reclamation

```
struct fifo fifo;
unsigned int timestamp[NTHREADS];

void
reader(void)
{
    struct node *garbage;
    void *value;
    int i;
    unsigned int s, sn[NTHREADS];

    for (;;) {
        garbage = dequeue(&fifo, &value);
        if (garbage == NULL)
            continue;

        for (i = 0; i < NTHREADS; i++) {
            sn[i] = ck_pr_load_uint(&timestamp[i]);
        }

        for (i = 0; i < NTHREADS; i++) {
            s = ck_pr_load_uint(&timestamp[i]);
            while (s == sn[i])
                /* Wait for writer quiescent state. */;
        }

        free(garbage);
    }

    return;
}
```

```
void
writer(void)
{
    struct node *m;
    uintptr_t i = 0;

    for (;;) {
        m = malloc(sizeof *m);
        enqueue(&fifo, m, (void *)i++);
        ck_pr_inc_uint(&timestamp[MYTHREAD]);
    }

    return;
}
```

Safe Memory Reclamation



Next Steps

- Books
  - "Is Parallel Programming Hard, And, If So, What Can You Do About it?" (McKenney)
  - "The Art of Multiprocessor Programming" (Herlihy, Shavit)
  - "UNIX Systems for Modern Architectures" (Schimmel)
- Libraries
  - Amino CBBS
  - Concurrency Kit
  - URCU
- Websites
  - http://www.1024cores.net
  - http://reddit.com/r/systems
  - http://www.rdrop.com/users/paulmck/RCU/
  - http://concurrencykit.org/