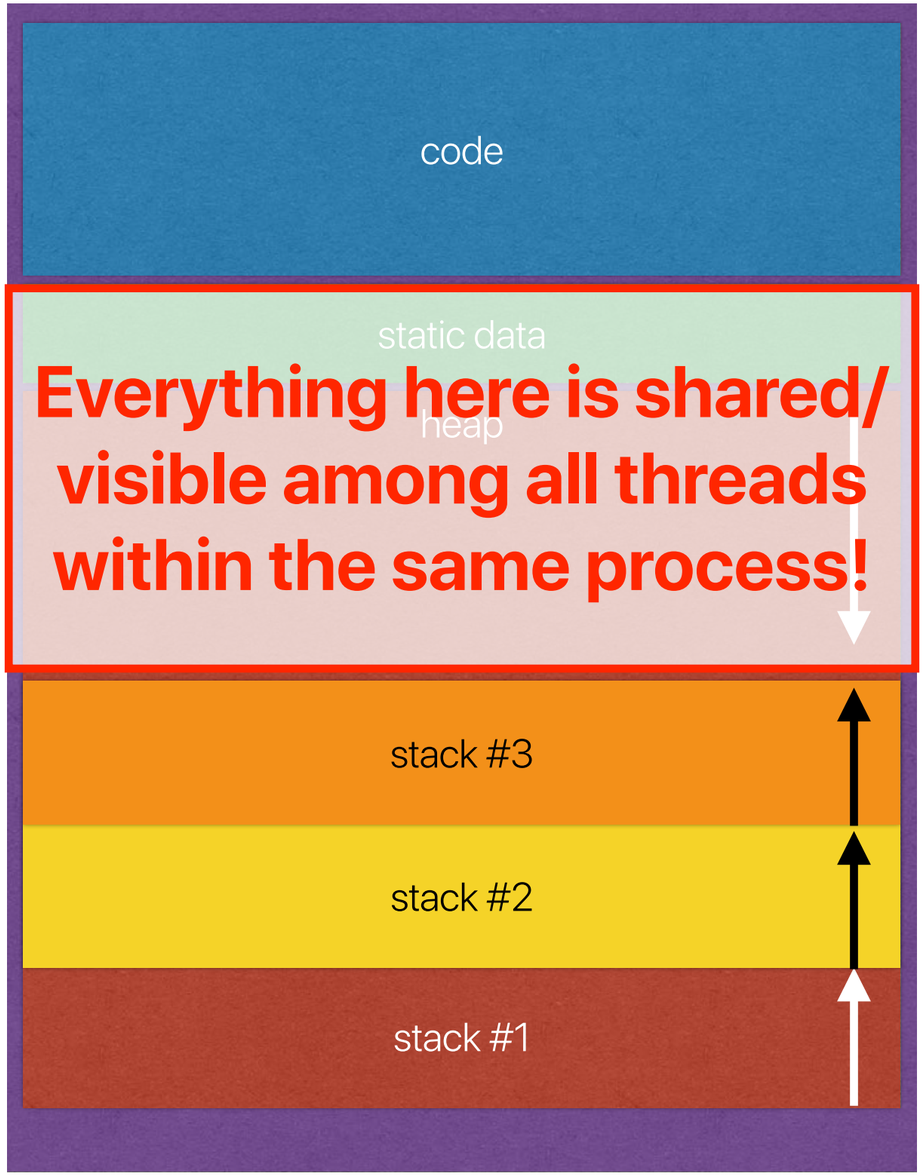
Synchronization

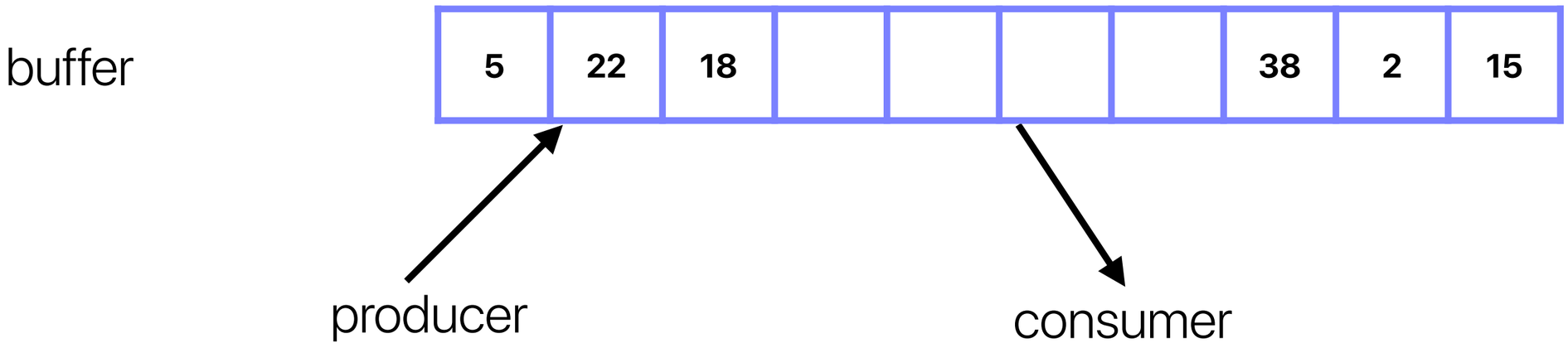
Hung-Wei Tseng

The virtual memory of multithreaded applications



Recap: Bounded-Buffer Problem

- Also referred to as “producer-consumer” problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



Recap: We need to control accesses to the buffer!

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;

        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;

    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {

        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;

        // do something w/ item
    }
    return NULL;
}
```

Recap: Solving the "Critical Section Problem"

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress — a thread outside of its critical section cannot block another thread from entering its critical section
3. Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

Recap: Naive implementation

- How many of the following can the naive implementation guarantee for the producer-consumer problem?
  - ✗ At most one process/thread in its critical section
  - ✗ A thread outside of its critical section cannot block another thread from entering its critical section
  - ✗ A thread cannot be postponed indefinitely from entering its critical section
  - ✗ The solution should work regardless the speed of executing threads and the number of processors

A. 0
B. 1
C. 2
D. 3
E. 4

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
  ; // spin
    *lock = 1;          // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

Naive implementation

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

what if the thread crashes/halts here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

all threads can see lock as 0 at this point

coherence cache misses? page fault?

Poll close in 1:30

How to achieve preemptive multitasking

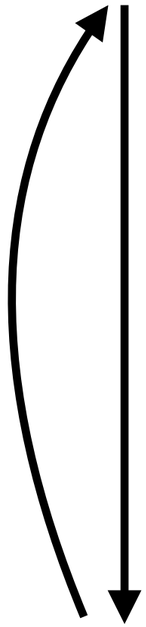
- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

Poll close in 1:30

How to achieve preemptive multitasking

- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

How preemptive multitasking works



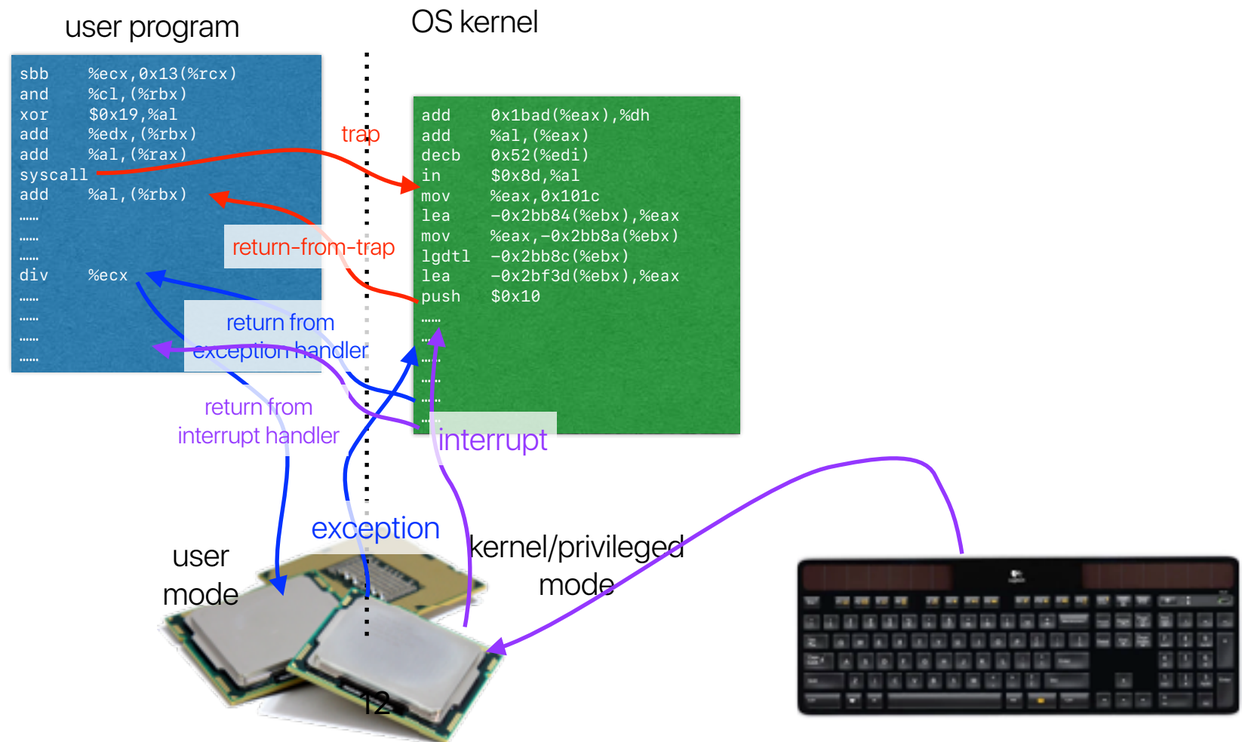
- Setup a **timer** (a hardware feature by the processor)event before the process start running
- After a certain period of time, the **timer** generates **interrupt** to force the running process transfer the control to OS kernel
- The OS kernel code decides if the system wants to continue the current process
  - If not — context switch
  - If yes, return to the process

How to achieve preemptive multitasking

- Which of the following mechanism are used to support preemptive multitasking?
  A. Exception
  B. Interrupt
  C. System calls

Three ways to invoke OS handlers

- System calls / trap instructions — raised by applications
  - Display images, play sounds
- Exceptions — raised by processor itself
  - Divided by zero, unknown memory addresses
- Interrupts — raised by hardware
  - Keystroke, network packets



Poll close in 1:30

Disable interrupts?

- How many of the following can the disable interrupts guarantee for the producer-consumer problem?
  - ① At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Disable interrupts?

- How many of the following can the disable interrupts guarantee for the producer-consumer problem?
  - ① At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ④ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Disable interrupts?

- How many of the following can the disable interrupts guarantee for the producer-consumer problem?
  - ~~①~~ At most one process/thread in its critical section
  - ② A thread outside of its critical section cannot block another thread from entering its critical section
  - ③ A thread cannot be postponed indefinitely from entering its critical section
  - ~~④~~ The solution should work regardless the speed of executing threads and the number of processors

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

— threads on other processors can still run

— now many of them can go...

— you can only disable the interrupt on the current processor

We must use atomic instructions

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

what if context switch happens here?

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}
```

— the lock must be updated atomically

```
void Pthread_mutex_unlock(volatile unsigned int *lock)
{
    *lock = 0;
}
```

We must use atomic instructions

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
```

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}
```

exchange the content in %0 and %1

a prefix to xchgl that locks the whole cache line

```
void Pthread_mutex_lock(volatile unsigned int *lock) {
    // what code should go here?
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    // what code should go here?
}
```

We must use atomic instructions

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
```

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void Pthread_mutex_lock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1);
}

void Pthread_mutex_unlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Semaphores

Semaphores

- A synchronization variable
- Has an integer value — current value dictates if thread/process can proceed
- Access granted if val > 0, blocked if val == 0
- Maintain a list of waiting processes

Semaphore Operations

- `sem_wait(S)`
  - if S > 0, thread/process proceeds and decrement S
  - if S == 0, thread goes into “waiting” state and placed in a special queue
- `sem_post(S)`
  - if no one waiting for entry (i.e. waiting queue is empty), increment S
  - otherwise, allow one thread in queue to proceed

Semaphore Op Implementations

```
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}
```

```
sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}
```

```
sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}
```

Atomicity in Semaphore Ops

- Semaphore operations must operate atomically
  - Requires lower-level synchronization methods requires (test-and-set, etc.)
  - Most implementations still require on busy waiting in spinlocks
- What did we gain by using semaphores?
  - Easier for programmers
  - Busy waiting time is limited

Poll close in 1:30

Adding Synchronization?

- What variables to use for this problem?

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    Sem_init(&filled, 0);
    Sem_init(&empty, BUFF_SIZE);
    while(TRUE) {
        int item = …;
        Sem_wait(&W);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Sem_post(&X);
    }
    printf("parent: end\n");
    return 0;
}
```

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Sem_wait(&Y);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        // do something w/ item
        Sem_post(&Z);
    }
    return NULL;
}
```

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Poll close in 1:30

Adding Synchronization?

- What variables to use for this problem?

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    Sem_init(&filled, 0);
    Sem_init(&empty, BUFF_SIZE);
    while(TRUE) {
        int item = …;
        Sem_wait(&W);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Sem_post(&X);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Sem_wait(&Y);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        // do something w/ item
        Sem_post(&Z);
    }
    return NULL;
}
```

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Adding Synchronization?

- What variables to use for this problem?

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    Sem_init(&filled, 0);
    Sem_init(&empty, BUFF_SIZE);
    while(TRUE) {
        int item = …;
        Sem_wait(&W);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Sem_post(&X);
    }
    printf("parent: end\n");
    return 0;
}
```

```
int buffer[BUFF_SIZE]; // shared global
sem_t filled, empty;
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Sem_wait(&Y);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        // do something w/ item
        Sem_post(&Z);
    }
    return NULL;
}
```

	W	X	Y	Z
A	empty	empty	filled	filled
B	empty	filled	filled	empty
C	filled	empty	empty	filled

Poll close in 1:30

Are semaphores good enough?

- How many of the following statements are correct regarding semaphores implemented through atomic instructions?
  - ① Semaphores can only support limited amount of concurrency/threads
  - ② Semaphores can work correctly if one of the threads go into a faulty state
  - ③ Semaphores do not prevent deadlock situations
  - ④ A thread entering its critical section protected by (a) semaphore(s) may not be able to make meaningful progress during a scheduling quanta
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Are semaphores good enough?

- How many of the following statements are correct regarding semaphores implemented through atomic instructions?
  - ① Semaphores can only support limited amount of concurrency/threads
  - ② Semaphores can work correctly if one of the threads go into a faulty state
  - ③ Semaphores do not prevent deadlock situations
  - ④ A thread entering its critical section protected by (a) semaphore(s) may not be able to make meaningful progress during a scheduling quanta
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Are semaphores good enough?

- How many of the following statements are correct regarding semaphores implemented through atomic instructions?
  - ① Semaphores can only support limited amount of concurrency/threads
  - ② Semaphores can work correctly if one of the threads go into a faulty state
  - ③ Semaphores do not prevent deadlock situations
  - ④ A thread entering its critical section protected by (a) semaphore(s) may not be able to make meaningful progress during a scheduling quanta

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

RCU Usage In the Linux Kernel: Eighteen Years Later

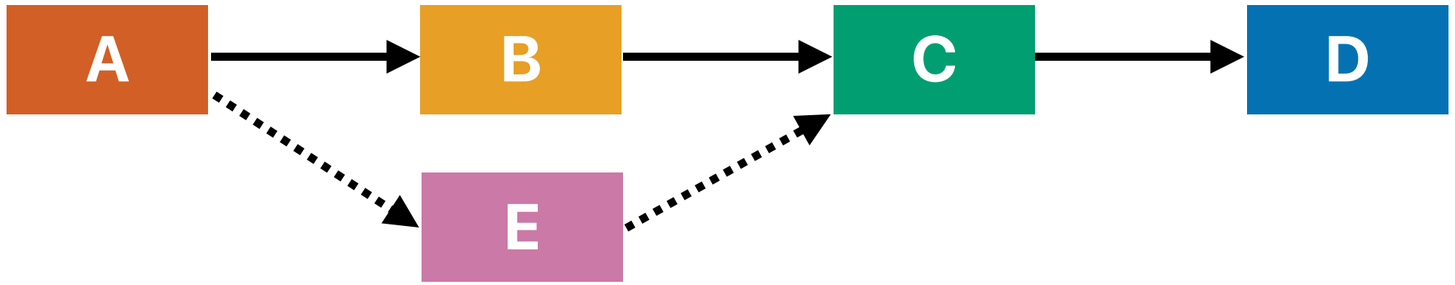
Paul E. McKenney (Facebook), Joel Fernandes (Google), and Silas Boyd-Wickize (MIT CSAIL)

ACM SIGOPS Operating Systems Review Vol. 54, No. 1, August 2020, pp. 47–63.

Poll close in 1:30

RCU: Read-copy-update

- Consider the following linked-list structure



How many of the following statements are true (or can be done) if we use RCU to traverse/update the data structure appropriately?

① Any running thread can traverse the linked list without waiting for a lock

② RCU can only allow as many concurrent reading threads as the number of hardware threads (i.e., number of processor threads).

③ If a thread is removing B from the list and replacing B with a new node E, B can only be physically removed if all preceding threads traversing the linked list have completed

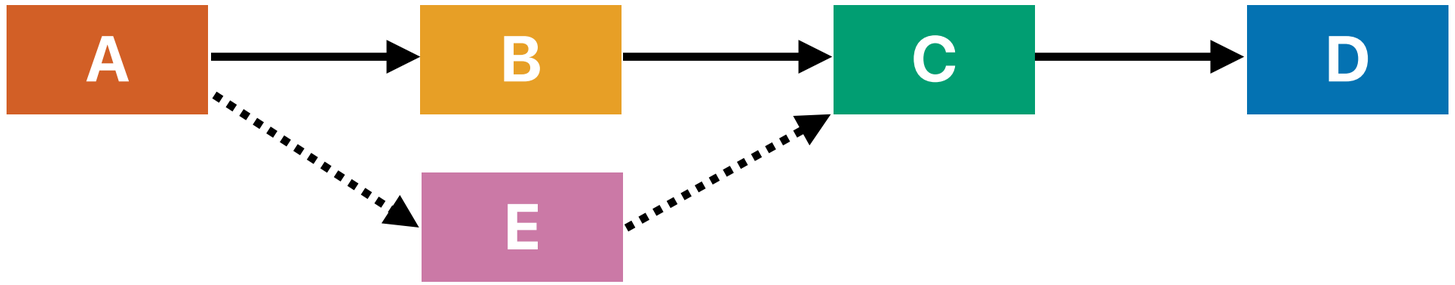
④ RCU is an implementation of wait-free synchronization

A. 0
B. 1
C. 2
D. 3
E. 4

Poll close in 1:30

RCU: Read-copy-update

- Consider the following linked-list structure



How many of the following statements are true (or can be done) if we use RCU to traverse/update the data structure appropriately?

① Any running thread can traverse the linked list without waiting for a lock

② RCU can only allow as many concurrent reading threads as the number of hardware threads (i.e., number of processor threads).

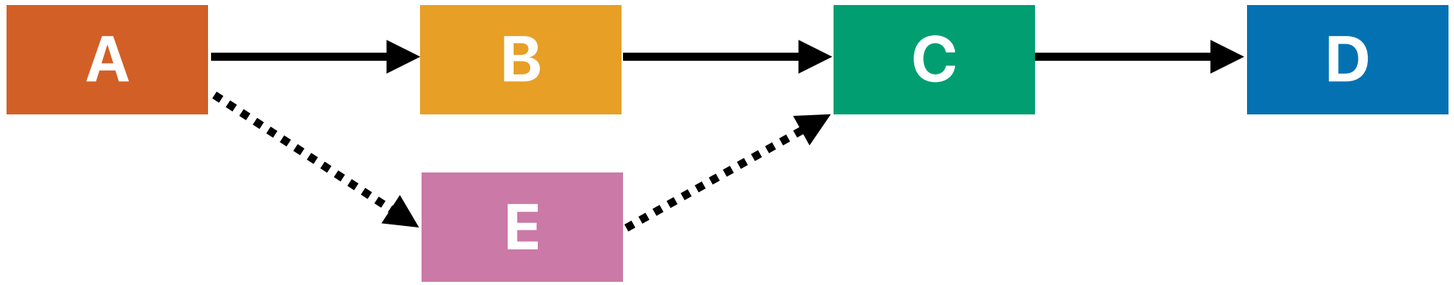
③ If a thread is removing B from the list and replacing B with a new node E, B can only be physically removed if all preceding threads traversing the linked list have completed

④ RCU is an implementation of wait-free synchronization

A. 0
B. 1
C. 2
D. 3
E. 4

RCU: Read-copy-update

- Consider the following linked-list structure



How many of the following statements are true (or can be done) if we use RCU to traverse/update the data structure appropriately?

① Any running thread can traverse the linked list without waiting for a lock

② RCU can only allow as many concurrent reading threads as the number of hardware threads (i.e., number of processor threads).

③ If a thread is removing B from the list and replacing B with a new node E, B can only be physically removed if all preceding threads traversing the linked list have completed

④ RCU is an implementation of wait-free synchronization

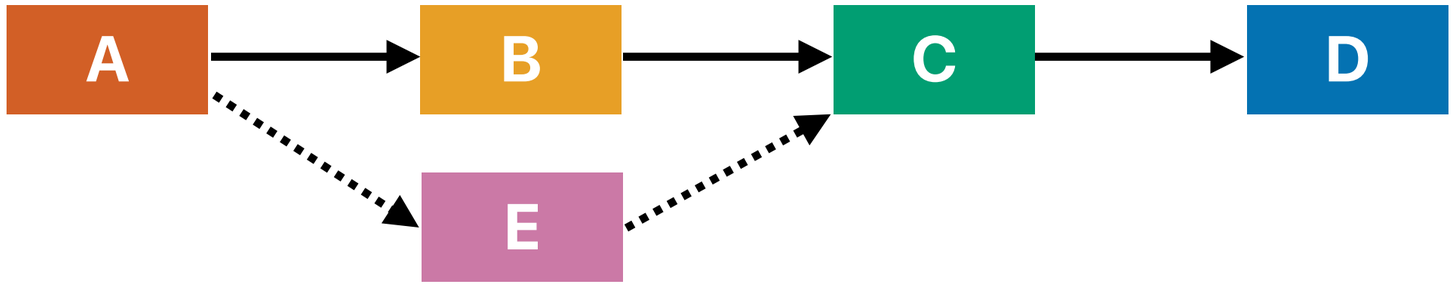
A. 0
B. 1
C. 2
D. 3
E. 4

RCU API

API Name	C Equivalent
rcu_read_lock() = rcu_read_unlock()	Simply disable/re-enable interrupts
rcu_assign_pointer(p, x)	p = x
rcu_dereference(p)	*p
synchronize_rcu()	Wait for **existing** RCU critical sections to complete

RCU: Read-copy-update

- Consider the following linked-list structure



How many of the following statements are true (or can be done) if we use RCU to traverse/update the data structure appropriately?

① Any running thread can traverse the linked list without waiting for a lock — Yes — just disable interrupt, deterministic operations

② RCU can only allow as many concurrent reading threads as the number of hardware threads (i.e., number of processor threads). — Yes, because there are only these many processor available & all are running since interrupt are disabled

③ If a thread is removing B from the list and replacing B with a new node E, B can only be physically removed if all preceding threads traversing the linked list have completed

④ RCU is an implementation of wait-free synchronization

A. 0
B. 1
C. 2
D. 3
E. 4

Poll close in 1:30

Why disabling interrupts

- How many of the following statements describing the reason why rcu_read_lock disable interrupts
  - ① Guarantee mutual exclusions for reads
  - ② Guarantee mutual exclusions for updates
  - ③ Guarantee all readers can run to finish without being context switched out
  - ④ Simplifies the implementation of updates
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Poll close in 1:30

Why disabling interrupts

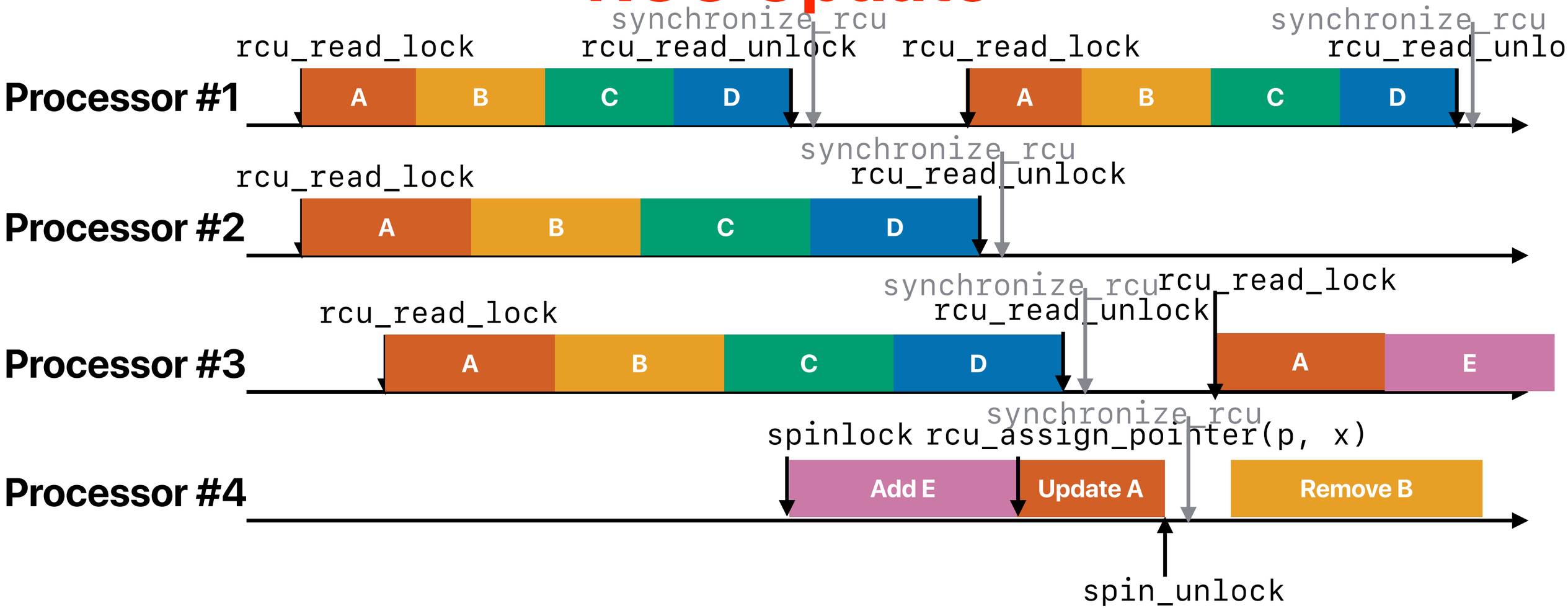
- How many of the following statements describing the reason why rcu_read_lock disable interrupts
  - ① Guarantee mutual exclusions for reads
  - ② Guarantee mutual exclusions for updates
  - ③ Guarantee all readers can run to finish without being context switched out
  - ④ Simplifies the implementation of updates
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

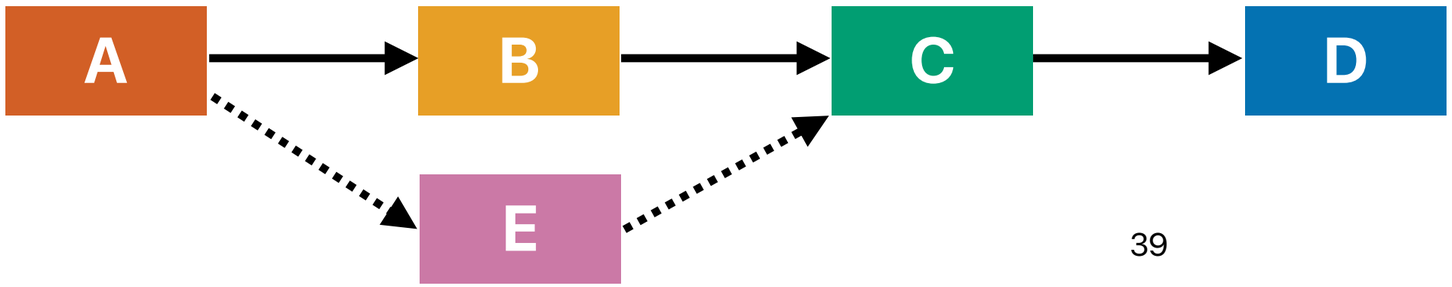
Why disabling interrupts

- How many of the following statements describing the reason why `rcu_read_lock` disable interrupts
  - ~~①~~ Guarantee mutual exclusions for reads — (1) Does not help & (2) It's never a goal of RCU
  - ~~②~~ Guarantee mutual exclusions for updates — (1) Does not help & (2) You need locks
  - ③ Guarantee all readers can run to finish without being context switched out — Yes — but why we need to guarantee this?
  - ④ Simplifies the implementation of updates — Here is the reason!!!

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

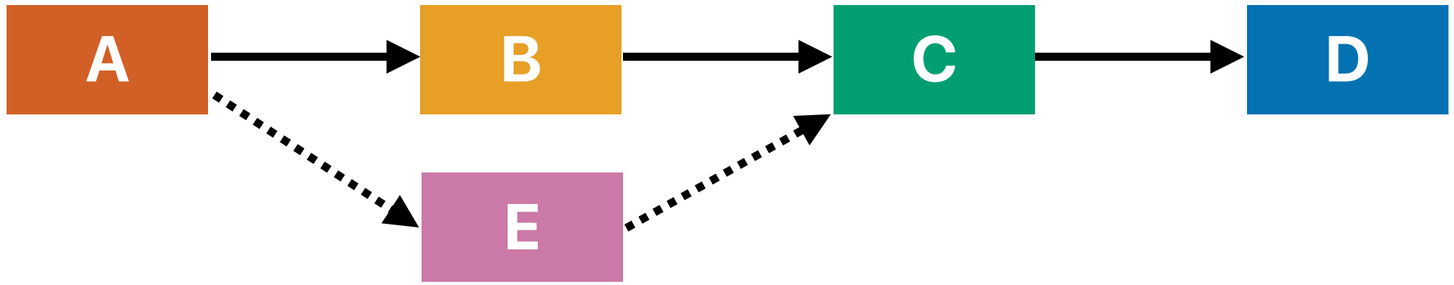
RCU Update


synchronize_rcu
rcu_read_lock
rcu_read_unlock
rcu_read_lock
synchronize_rcu
rcu_read_unloc
Processor #1
A
B
C
D
A
B
C
D
synchronize_rcu
rcu_read_lock
rcu_read_unlock
Processor #2
A
B
C
D
synchronize_rcu
rcu_read_lock
rcu_read_lock
rcu_read_unlock
Processor #3
A
B
C
D
A
E
synchronize_rcu
spinlock
rcu_assign_pointer(p, x)
Processor #4
Add E
Update A
Remove B
spin_unlock


A
B
C
D
E

RCU: Read-copy-update

- Consider the following linked-list structure



How many of the following statements are true (or can be done) if we use RCU to traverse/update the data structure appropriately?

① Any running thread can traverse the linked list without waiting for a lock — Yes — just disable interrupt, deterministic operations

② RCU can only allow as many concurrent reading threads as the number of hardware threads (i.e., number of processor threads). — Yes, because there are only these many processor available & all are running since interrupt are disabled

③ If a thread is removing B from the list and replacing B with a new node E, B can only be physically removed if all preceding threads traversing the linked list have completed — This the magic of RCU — allowing threads to continue without being affected by the update

④ RCU is an implementation of wait-free synchronization

A. 0
B. 1
C. 2
D. 3
E. 4

Wait-free Synchronization

Maurice Herlihy
Brown University
ACM Transactions on Programming Languages and Systems (TOPLAS), 1991

Poll close in 1:30

Wait-free synchronization

- How many of the following statements fulfill the requirements of wait-free synchronization?
  - ① Any operation from a process accessing the shared data structure must be completed within a finite number of steps
  - ② No process can be prevented from completing its operation by failures from other processes
  - ③ The implementation of wait-free synchronization cannot depend on hardware support
  - ④ The implementation of wait-free synchronization should work regardless of the processing speed

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Poll close in 1:30

Wait-free synchronization

- How many of the following statements fulfill the requirements of wait-free synchronization?
  - ① Any operation from a process accessing the shared data structure must be completed within a finite number of steps
  - ② No process can be prevented from completing its operation by failures from other processes
  - ③ The implementation of wait-free synchronization cannot depend on hardware support
  - ④ The implementation of wait-free synchronization should work regardless of the processing speed

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Wait-free synchronization

A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds on the other processes. The wait-free condition provides fault-tolerance: no process can be prevented from completing an operation by undetected halting failures of other processes, or by arbitrary variations in their speed. The fundamental problem of wait-free synchronization can be phrased as follows:

> Given two concurrent objects X and Y, does there exist a wait-free implementation of X by Y?

Wait-free synchronization

- How many of the following statements fulfill the requirements of wait-free synchronization?
  - ① Any operation from a process accessing the shared data structure must be completed within a finite number of steps
  - ② No process can be prevented from completing its operation by failures from other processes
  - ③ The implementation of wait-free synchronization cannot depend on hardware support
  - ④ The implementation of wait-free synchronization should work regardless of the processing speed

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Wait-free synchronization

- How many of the following statements fulfill the requirements of wait-free synchronization?
  - ① Any operation from a process accessing the shared data structure must be completed within a finite number of steps — **Only true for RCU reads**
  - ② No process can be prevented from completing its operation by failures from other processes — **Only true for RCU reads**
  - ~~③~~ The implementation of wait-free synchronization cannot depend on hardware support
  - ④ The implementation of wait-free synchronization should work regardless of the processing speed — **Only true for RCU reads**
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Wait-Free Queues With Multiple Enqueuers and Dequeuers *

Alex Kogan
Department of Computer Science
Technion, Israel
sakogan@cs.technion.ac.il

Erez Petrank
Department of Computer Science
Technion, Israel
erez@cs.technion.ac.il

Abstract

The queue data structure is fundamental and ubiquitous. Lock-free versions of the queue are well known. However, an important open question is whether practical wait-free queues exist. Until now, only versions with limited concurrency were proposed. In this paper we provide a design for a practical wait-free queue. Our construction is based on the highly efficient lock-free queue of Michael and Scott. To achieve wait-freedom, we employ a priority-based helping scheme in which faster threads help the slower peers to complete their pending operations. We have implemented our scheme on multicore machines and present performance measurements comparing our implementation with that of Michael and Scott in several system configurations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures; E.1 [*Data structures*]: Lists, stacks, and queues

General Terms Algorithms, Performance

Keywords concurrent queues, wait-free algorithms

1. Introduction

The proliferation of multicore systems motivates the research for efficient concurrent data structures. Being a fundamental and commonly used structure, first-in first-out (FIFO) queues[1] have been studied extensively, resulting in many highly concurrent algorithms. strict deadlines for operation completion exist, e.g., in real-time applications or when operating under a service level agreement (SLA), or in heterogenous execution environments where some of the threads may perform much faster or slower than others. Yet, most previous queue implementations (e.g., [14, 17, 19, 21, 24, 25]) provide the weaker *lock-free* property; lock-freedom ensures that among all processes accessing a queue, at least one will succeed to finish its operation. Although such non-blocking implementations guarantee global progress, they allow scenarios in which all but one thread starve while trying to execute an operation on the queue. The few wait-free queue constructions that exist, as discussed later, are either stem from general transformations on sequential objects and are impractical due to significant performance drawbacks, or severely limit the number of threads that may perform one or both of the queue operations concurrently.

In fact, when considering concurrent data structures in general, one realizes that with only a few exceptions (e.g., [5, 22]), wait-free constructions are very rare in practice. A possible reason for this situation is that such constructions are hard to design in an efficient and practical way. This paper presents the first practical design of wait-free queues, which supports multiple concurrent dequeuers and enqueuers. Our idea is based on the lock-free queue implementation by Michael and Scott [19], considered to be one of the most efficient and scalable non-blocking algorithms in the literature [11, 14, 24]. Our design employs an efficient helping mechanism, which ensures that each operation is applied exactly once and in a bounded time. We achieve wait-freedom by assigning each

Announcement

- Regarding in-person instructions starting from February
  - We will have live sessions in Material Science Building 103 starting **next Tuesday** 2p!
  - Online options remain open
  - Please setup "Poll everywhere" before attending lectures — both in-person and online — by next Tuesday as well
- Midterm
  - Will release on 2/10/2021 0:00am and due on 2/11/2021 11:59:00pm
  - You will have to find a consecutive, non-stop 80-minute slot with this period
  - One time, cannot reinitiate — please make sure you have a stable system and network
  - **No late submission is allowed**
- Project released — https://github.com/hungweitseng/CS202-MMA
  - Groups in 2. 3 is acceptable, but not recommended
  - **Pull the latest version — had some changes for later kernel versions**
  - **Install an Ubuntu Linux 20.04 VM as soon as you can!**
  - **Please do not use a real machine — you may not be able to reboot again**
  - Need help? Check for office hours — https://calendar.google.com/calendar/u/0/r?cid=ucr.edu_b8u6dvkretn6kq6igunlc6bldg@group.calendar.google.com
- Reading quizzes due next Tuesday

Computer
Science &
Engineering

202

つづく