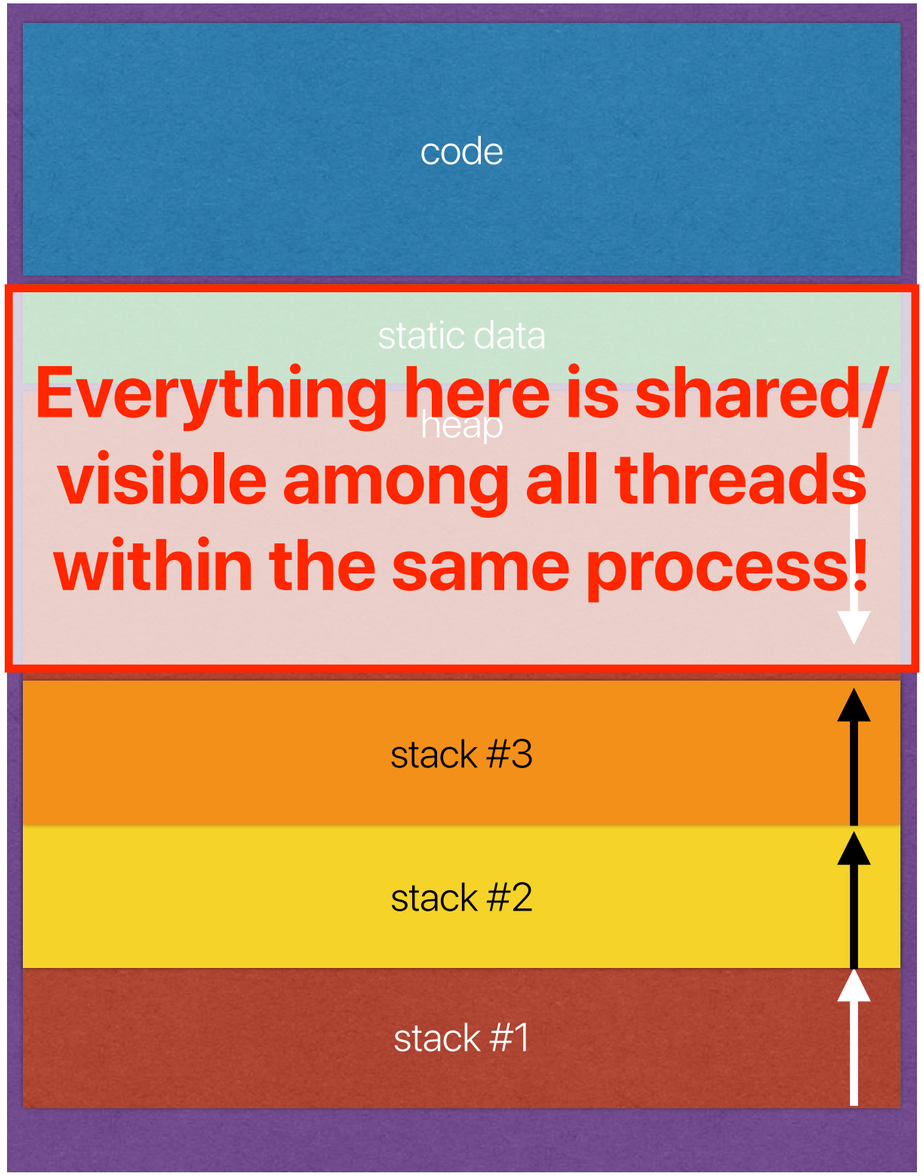
Synchronization

Hung-Wei Tseng

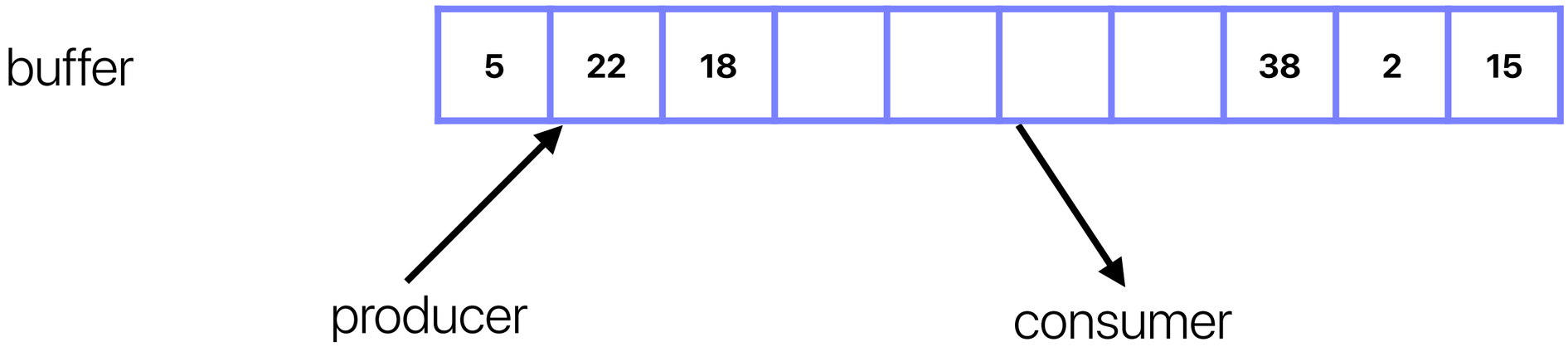
Thread programming & synchronization

The virtual memory of multithreaded applications



Bounded-Buffer Problem

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



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;
}
```

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

Use locks

```
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;
}
```

```
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;
}
```

How to implement lock/unlock

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;
}
```

```
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;
}
```

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

RCU Usage In the Linux Kernel: Eighteen Years Later

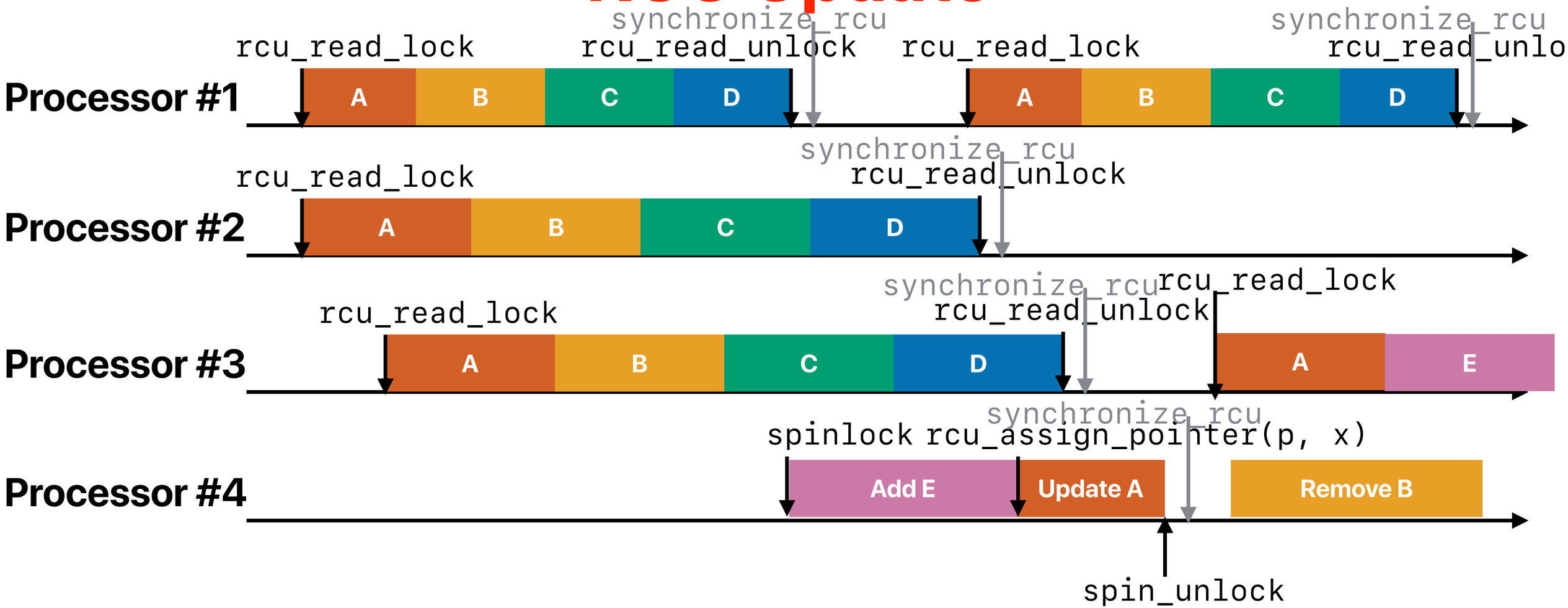
Paul E. McKenney (Facebook), Joel Fernandes (Google), and Silas Boyd-Wickize (MIT CSAIL)

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

RCU API

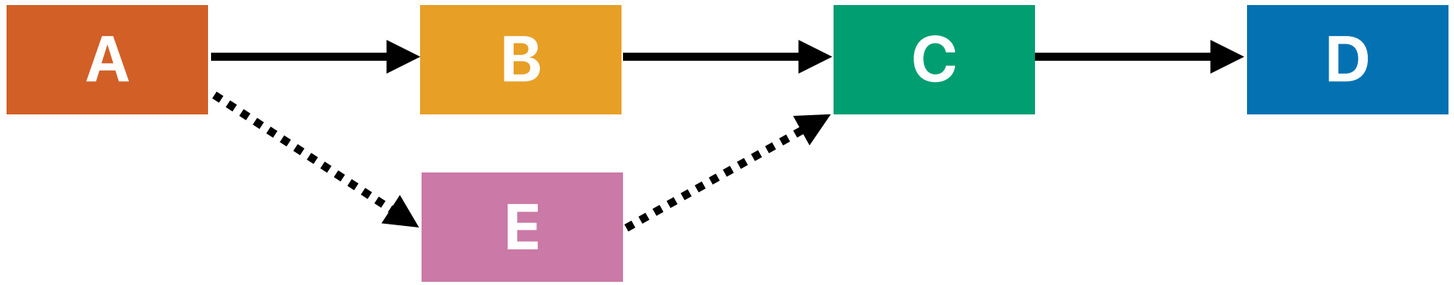
API Name	C Equivalent
rcu_read_lock() = rcu_read_unlock()	Simply disable/re-enable interrupts
rcu_assign_pointer(p, x)	p = x
rcu_dereference(p)	*p
synchronize_rcu()	Wait for **existing** RCU critical sections to complete

RCU Update


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

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