ILLINOIS TECH

College of Computing

CS 450 Operating Systems
More Locks

Yue Duan

Recap

- We went through some software implementation of locks
    - Good concurrency practice
- Ended up with Dekker and Peterson's algorithms
    - Work under assumptions of atomic and in order memory system
    - So, they do not work in practice
        - Compiler reorders
        - memory system is not ordered
- Need hardware support for synchronization
    - Two flavors:
        - Atomic instructions that read and update a variable
            - E.g., test-and-set, xchange, ...
        - Disable interrupts

Using Interrupts

- Turn off interrupts for critical sections
  - Prevent dispatcher from running another thread
  - Code between interrupts executes atomically

```
void acquire(lock_t *l) {
        disableInterrupts();
}

void release(lock_t *l) {
        enableInterrupts();
}
```

Using Interrupts

- Disadvantages
  - Only works on uniprocessors
  - Process can keep control of CPU for arbitrary length
  - Cannot perform other necessary work

```
void acquire(lock_t *l) {
        disableInterrupts();
}

void release(lock_t *l) {
        enableInterrupts();
}
```

xchg: atomic exchange, or test-and-set

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr

int xchg(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}
```

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

xchg: atomic exchange, or test-and-set

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = ??;
}

void acquire(lock_t *lock) {
        ???
        // spin-wait (do nothing)
}

void release(lock_t *lock) {
        lock->flag = ??;
}
```

```
int xchg(int *addr, int newval)
```

xchg: atomic exchange, or test-and-set

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t *lock) {
        lock->flag = 0;
}

void acquire(lock_t *lock) {
        while (xchg(&lock->flag, 1) == 1);
        // spin-wait (do nothing)
}

void release(lock_t *lock) {
        lock->flag = 0;
}
```

Other atomic HW instructions

```
int CompareAndSwap(int *ptr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

```
void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, 0, 1) == 1) ;
    // spin-wait (do nothing)
}
```

Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:

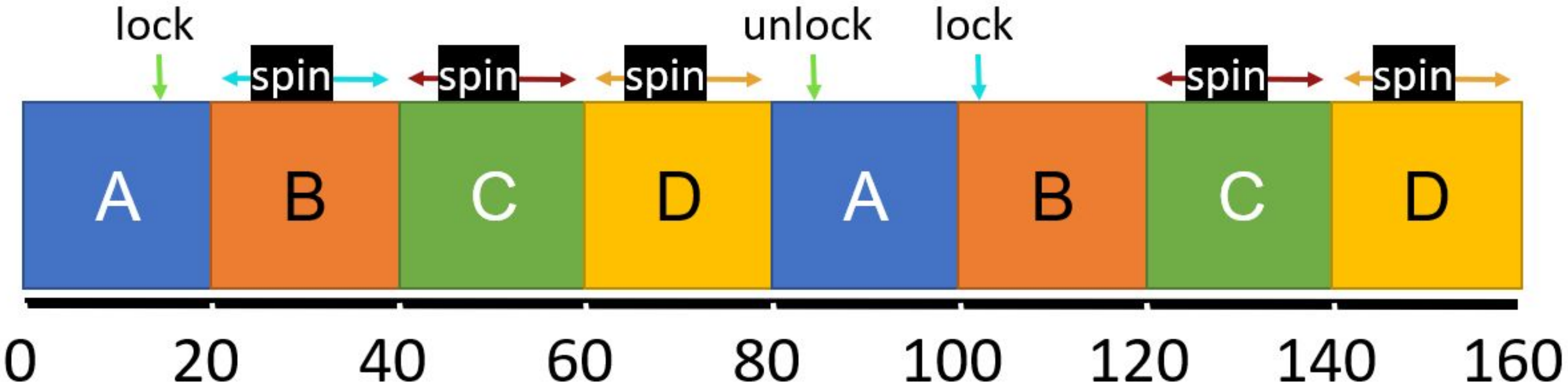
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

```
acquire(lock)
...
Critical section
...
release(lock)
```

Disabling Interrupts:

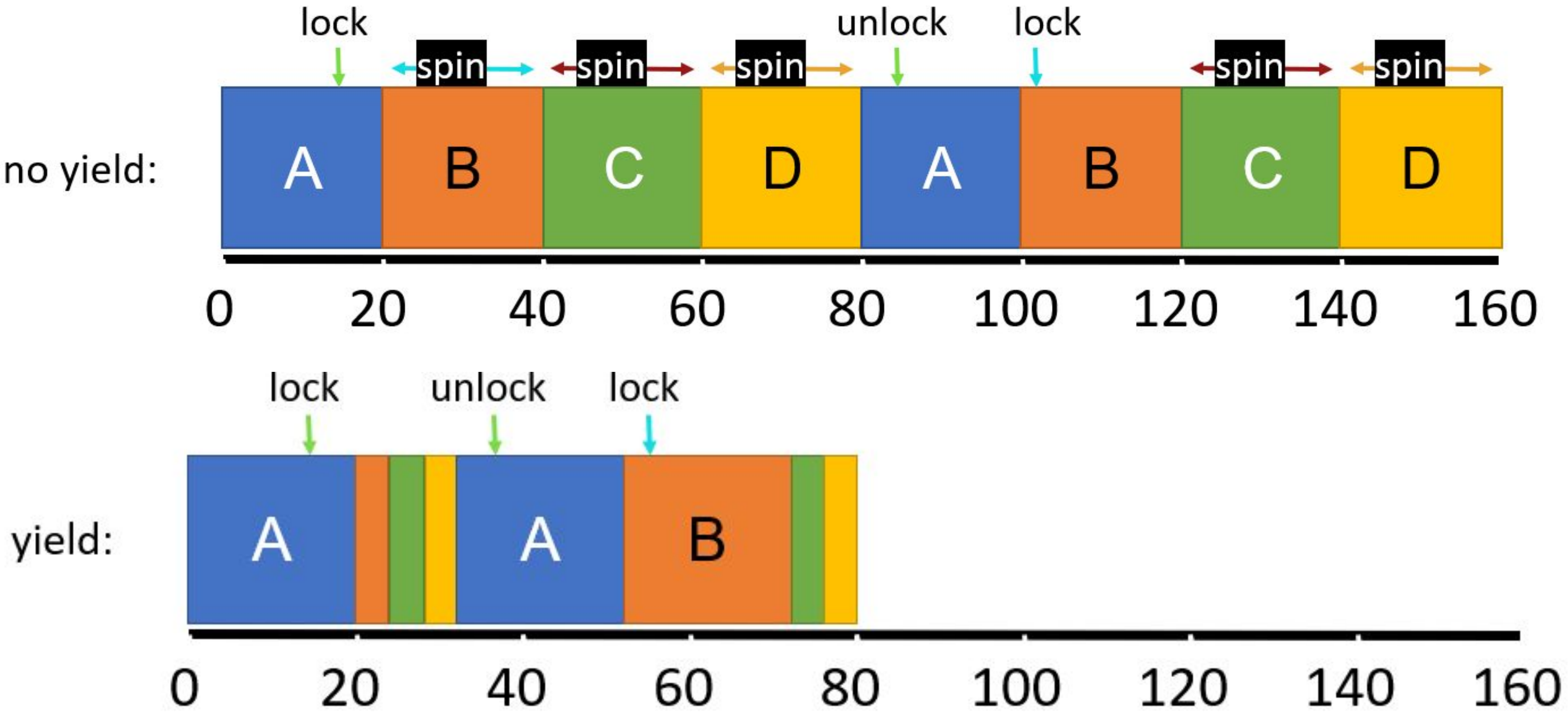
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

CPU Scheduler is Ignorant



- CPU scheduler may run B instead of A even though **B is waiting for A**

yield() Instead of spin



Spinlock Performance

- Waste...
  - Without yield: O(threads * time_slice)
  - With yield: O(threads * context_switch)
  - So even with yield, spinning is slow with high thread contention
- Next improvement:
  - Block and put thread on waiting queue instead of spinning

Lock implementation: block when waiting

- Lock implementation removes waiting threads from the ready queue
  - e.g. with park(): removes current thread from run queue
  - and unpark(threadID): returns thread tid to run queue
- Scheduler only runs ready threads
- Separates concerns
  - programmer doesn't need to deal with yield vs not/spinning issue
- **Quiz**: where should locks be implemented?

Example

Ready: {A, B, C, D}

Waiting: {}

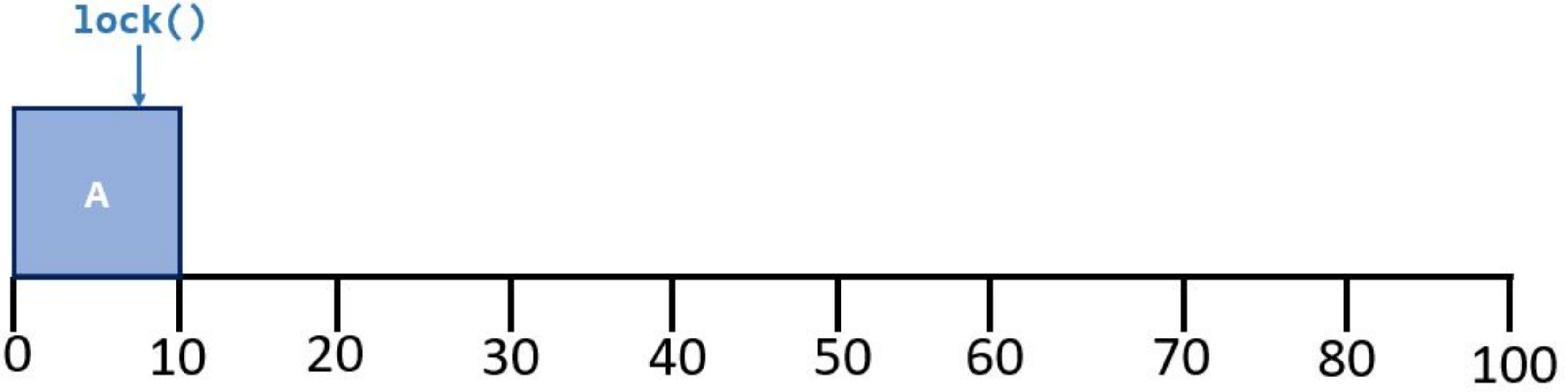
Running: {}

Example

Ready: {B, C, D}

Waiting: {}

Running: {A}

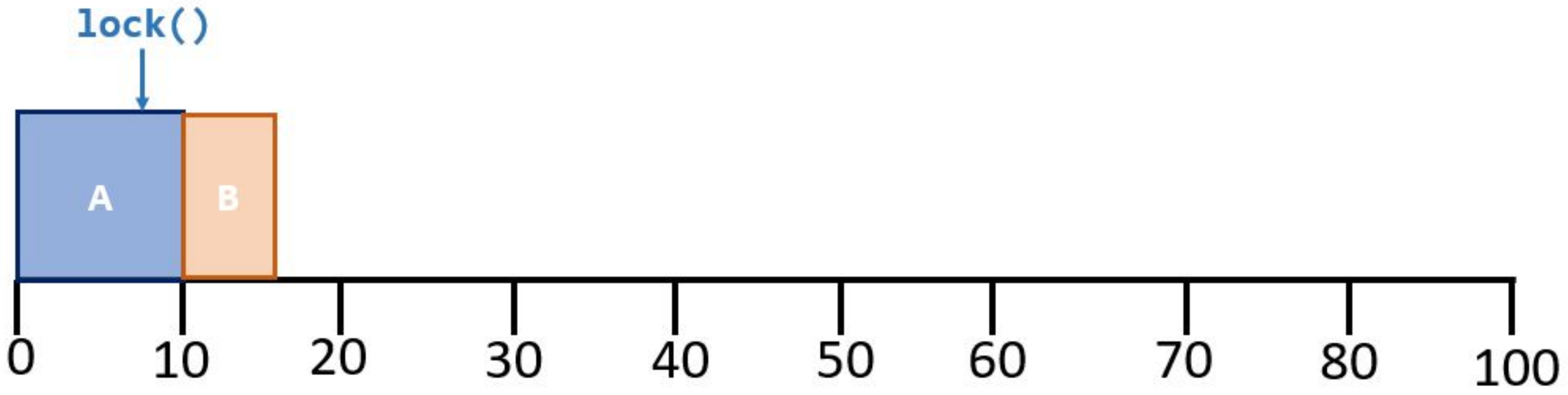


Example

Ready: {A, C, D}

Waiting: {}

Running: {B}

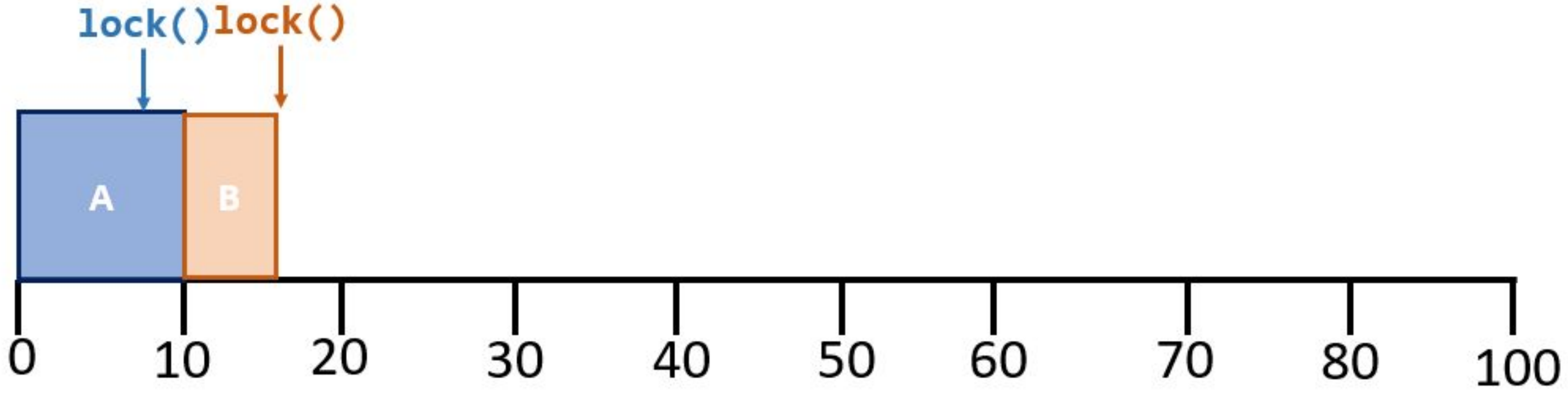


Example

Ready: {A, C, D}

Waiting: {B}

Running: {}

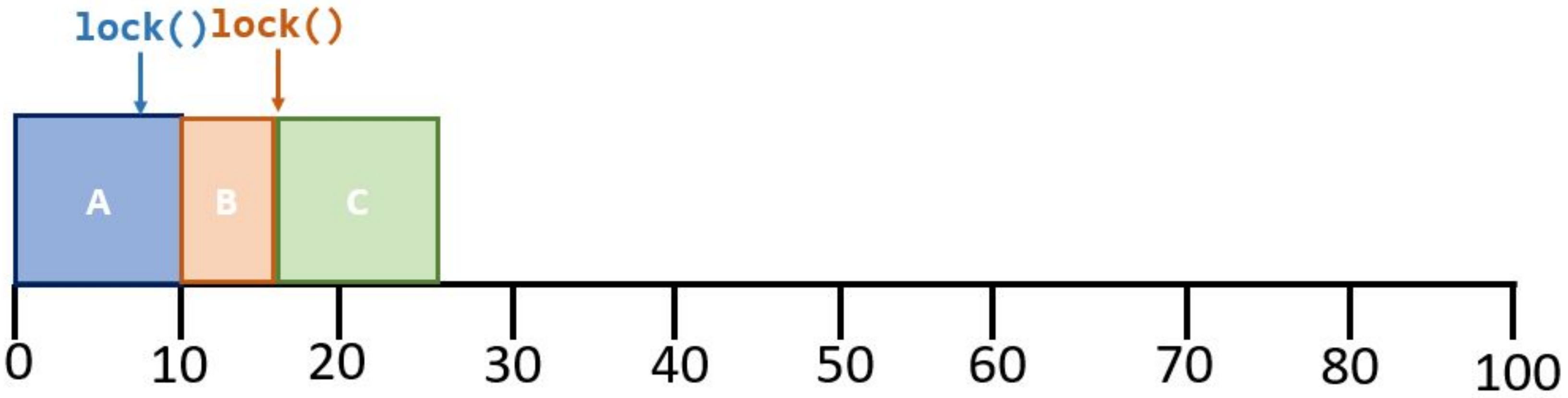


Example

Ready: {A, D}

Waiting: {B}

Running: {C}

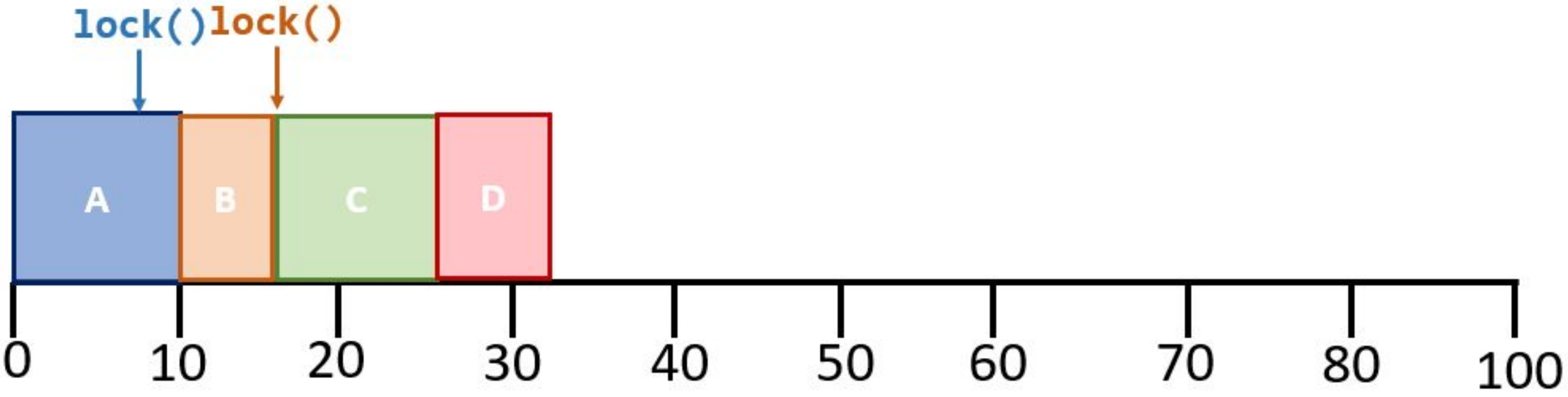


Example

Ready: {A, C}

Waiting: {B}

Running: {D}

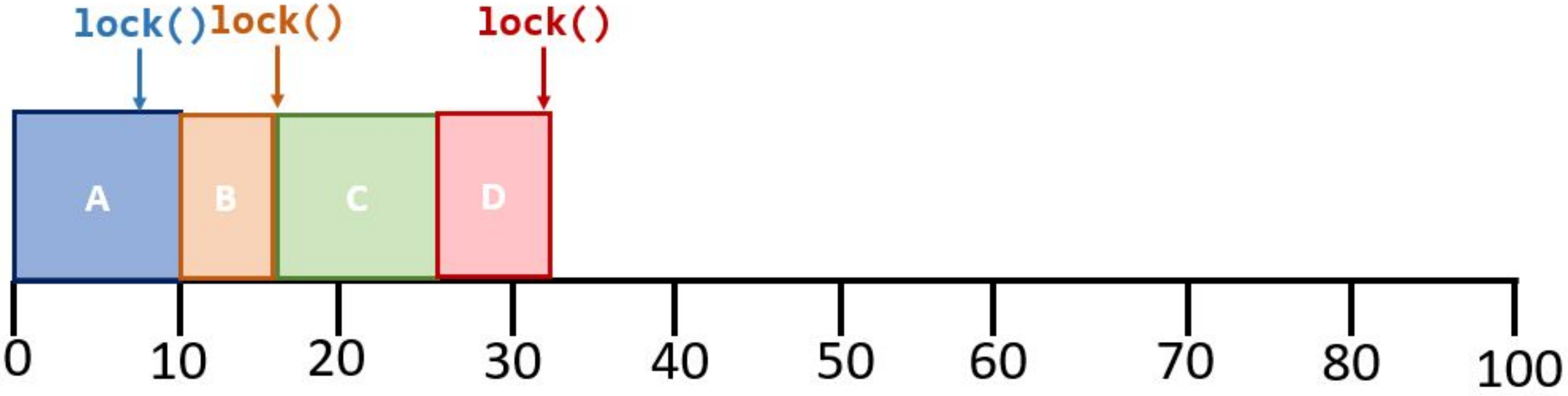


Example

Ready: {A, C}

Waiting: {B, D}

Running: {}

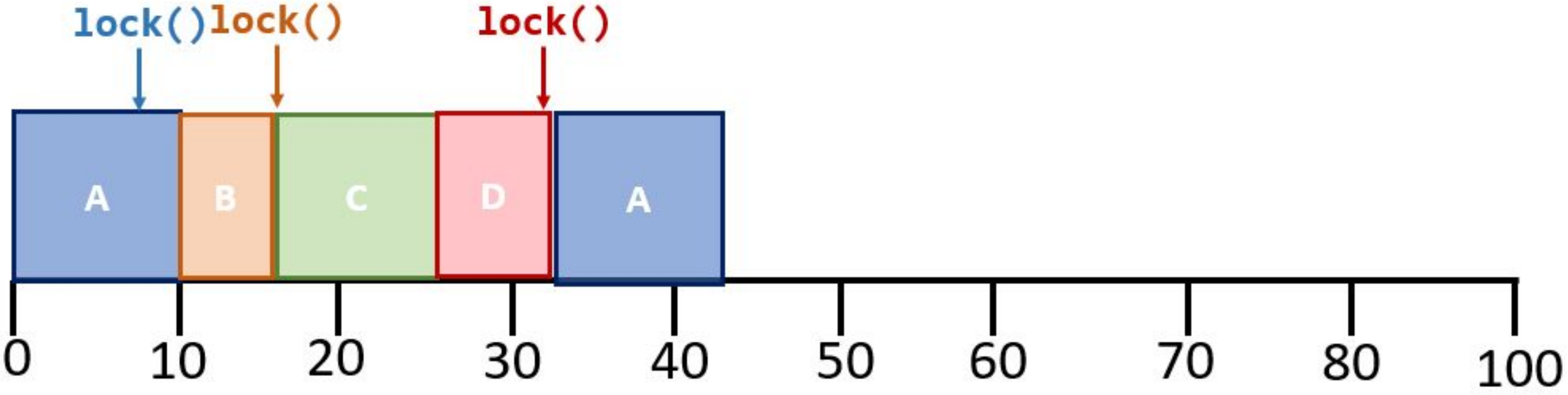


Example

Ready: {C}

Waiting: {B, D}

Running: {A}

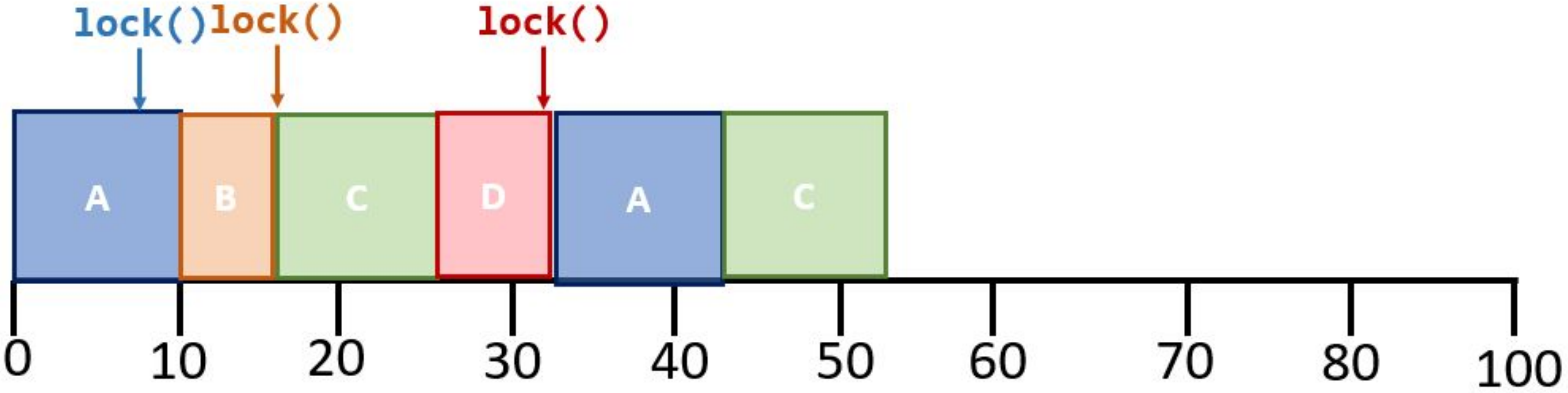


Example

Ready: {A}

Waiting: {B, D}

Running: {C}

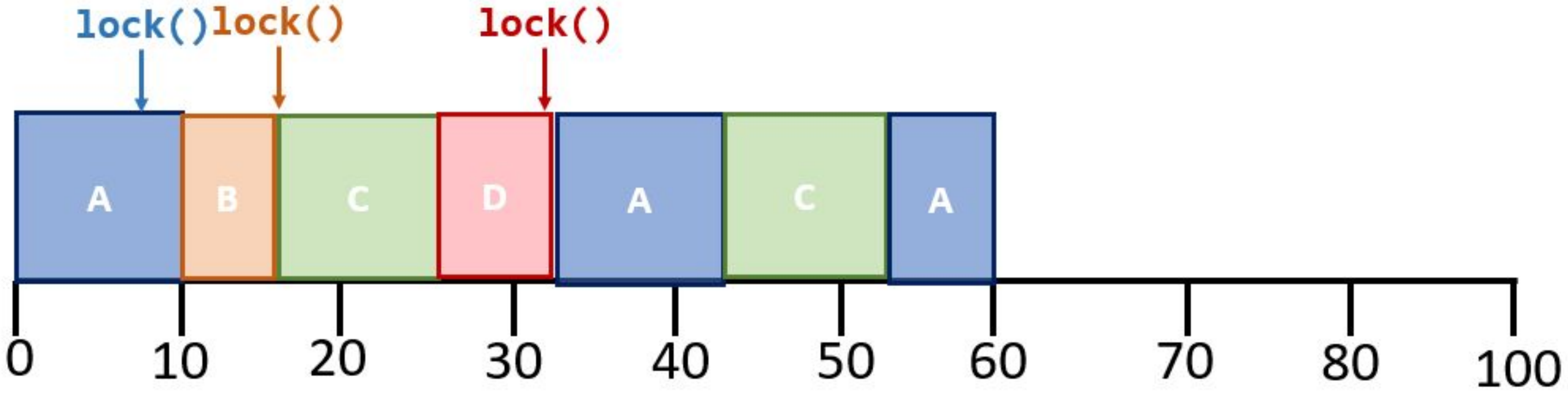


Example

Ready: {C}

Waiting: {B, D}

Running: {A}

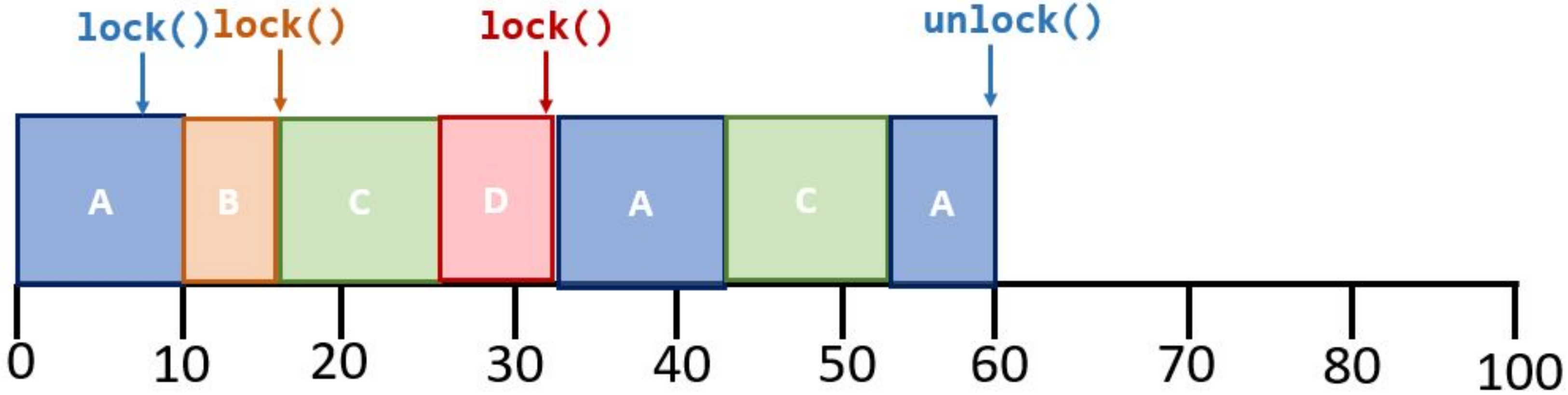


Example

Ready: {C, B, D}

Waiting: {}

Running: {A}

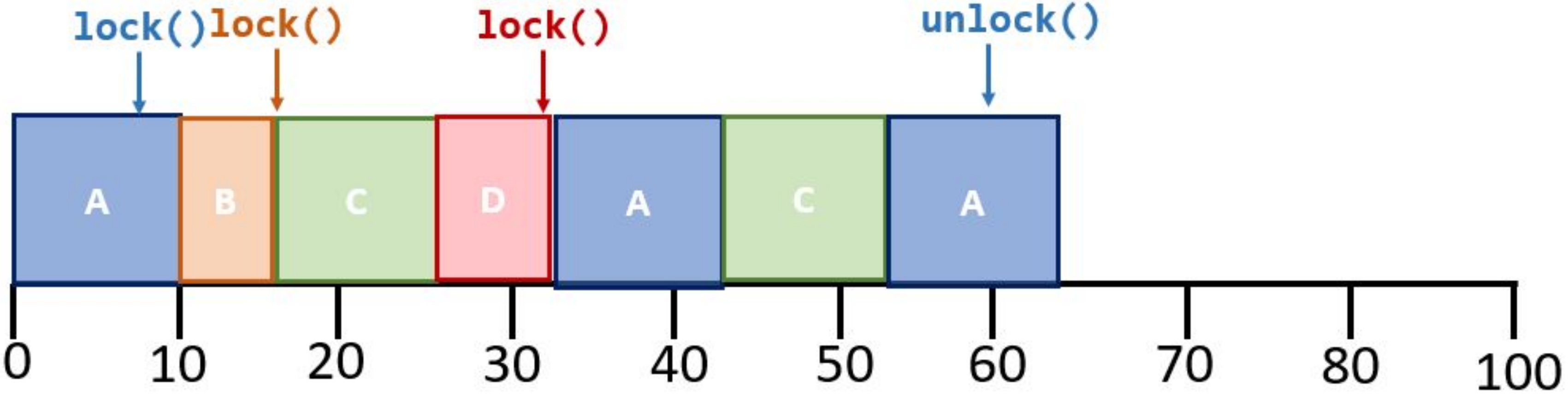


Example

Ready: {C, B, D}

Waiting: {}

Running: {A}

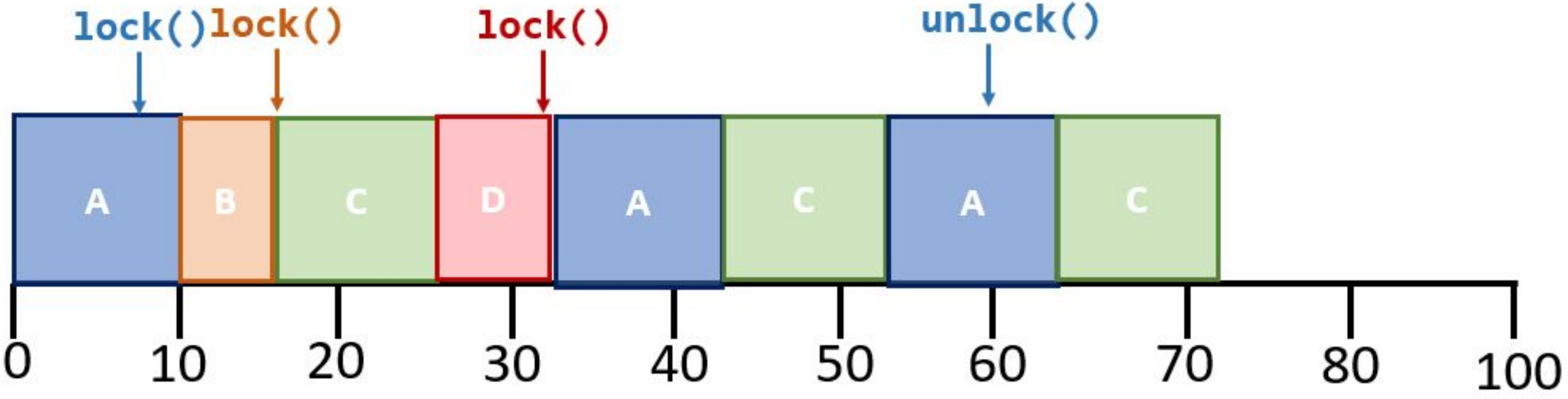


Example

Ready: {B, D, A}

Waiting: {}

Running: {C}

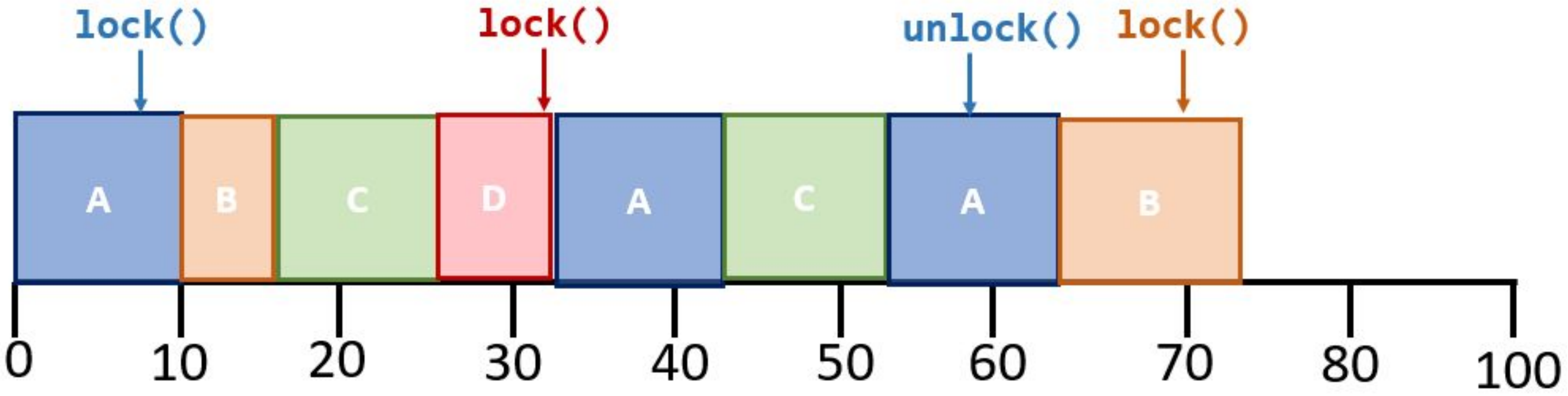


Example

Ready: {D, A, C}

Waiting: {}

Running: {B}



Benefits

- Threads aren't contending on the lock when they shouldn't be (mostly)
- **Fewer** context switches

Lock: block when waiting

```
typedef struct {
    bool lock  = false;
    bool guard = false;
    queue_t q;
} lock_t;
```

- Why do we need guard?
  - to keep queue ops from race
- Why OK to spin on guard?
  - waiting time is small
- In release(), why not set lock=false when unpark()?
- Is there a race condition?

```
void acquire(lock_t *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        enqueue(l->q, tid);
        l->guard = false;
        park(); // blocked
    } else {
        l->lock  = true;
        l->guard = false;
    }
}

void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
```

Race Condition

Thread 1

```
void acquire(lock_t *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        enqueue(l->q, tid);
        l->guard = false;

        park(); // blocked
    } else {
        l->lock  = true;
        l->guard = false;
    }
```

Thread 2

```
void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
```

false!

bad!

time

Lock: the fix

```
typedef struct {
    bool lock  = false;
    bool guard = false;
    queue_t q;
} lock_t;
```

- setpark(): informs the OS of my plan to park() myself
- If there is an unpark() between my setpark() and park(), park() will return immediately (no blocking)
- **Why does this fix the race?**

```
void acquire(lock_t *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        enqueue(l->q, tid);
        setpark(); // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock  = true;
        l->guard = false;
    }
}
void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
```

Spinning vs. Blocking

- Each approach is better under different circumstances
- **Uniprocessor**
  - Waiting process is scheduled ➜ Process holding lock isn’t
  - Therefore, waiting process should always relinquish processor
  - Associate queue of waiters with each lock (as in previous implementation)
- **Multiprocessor**
  - Waiting process is scheduled ➜ Process holding lock **might be**
  - Spin or block depends on **how long before lock is released**
  - Lock is going to be released quickly ➜ Spin-wait
  - Lock released slowly ➜ Block

Oracle

- Suppose we know how long **(t) a lock will be held**
- And suppose we make our decision to block or spin based on **C, the cost of a context switch**

$$action = \begin{cases} t \leq C & \rightarrow spin \\ t > C & \rightarrow block \end{cases}$$

But we need to know the future!

Two-Phase Locking

- A hybrid approach that combines best of spinning and blocking
- Phase 1:
    - spin for a short time, hoping the lock becomes available soon
- Phase 2:
    - if lock not released after a short while, then block
- Question: **how long to spin for?**
    - **There's a nice theory which is in practice hard to implement, so just spin for a few iterations**

Concurrency Goals

- Mutual Exclusion
  - Keep two threads from executing in a critical section concurrently
  - We solved this with **locks**
- Dependent Events
  - We want a thread to wait until some particular event has occurred
  - Or some condition has been met
  - Solved with **condition variables and semaphores**

Example: join()

```
pthread_t p1, p2;
pthread_create(&p1, NULL, mythread, “A”);
pthread_create(&p2, NULL, mythread, “B”);
// join waits for the threads to finish
pthread_join(p1, NULL);
pthread_join(p2, NULL);
printf(“Main: done\n [balance: %d]\n”, balance);
return 0;
```

join(): parent must wait for child thread to finish

Condition Variables

- CV:
    - queue of waiting threads
- **B** waits for a signal on CV before running
    - wait(CV, …); •
- **A** sends signal() on CV when time for **B** to run
    - signal(CV, …);

API

- **wait**(cond_t * cv, mutex_t * lock)
  - assumes lock is held when wait() is called (why?)
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- **signal**(cond_t * cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, NOP

THANK YOU!