We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

make history.

THE UNIVERSITY of ADELAIDE

Operating Systems

COMP SCI 3004 / COMP SCI 7064

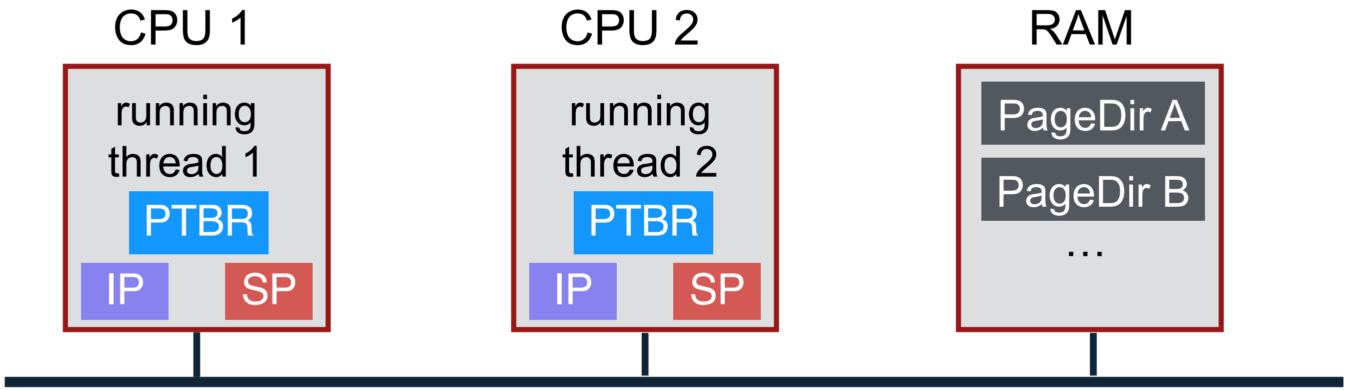
Week 6 – Concurrency: Locks

Introduction

- Locks
- Lock implementations
- & data structures
- Condition Variables

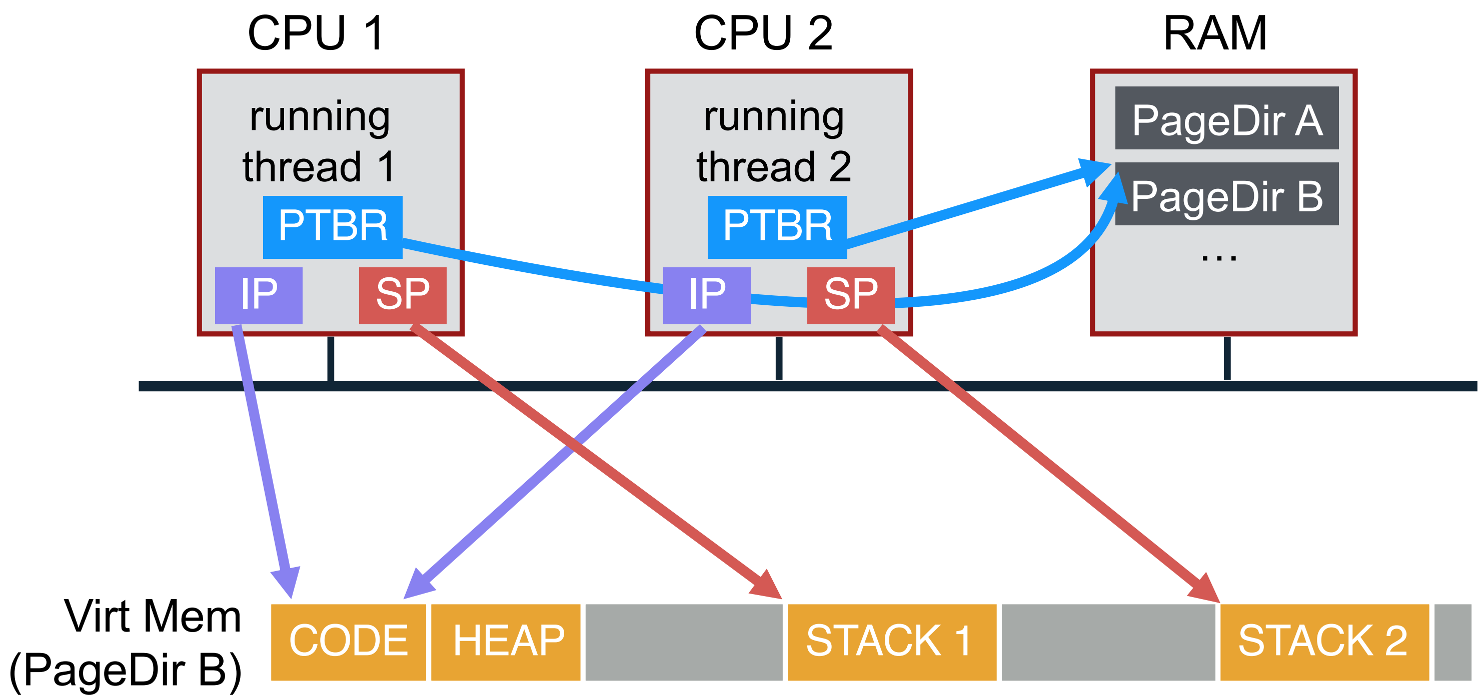
THE UNIVERSITY of ADELAIDE

Review



Virt Mem
(PageDir B)

Review:
Which registers store the same/different values across threads?


CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir B)
CODE
HEAP
STACK 1
STACK 2

Review: What is needed for CORRECTNESS?

```
balance = balance + 1;
```

Instructions accessing shared memory must execute as uninterruptable group

Need instructions to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

More general:

Need **mutual exclusion** for critical sections

- if process A is in critical section C, process B can't
  (okay if other processes do unrelated work)

THE UNIVERSITY of ADELAIDE

Locks: The Basic Idea

Ensure that any critical section executes as if it were a single atomic instruction.

- An example: the canonical update of a shared variable

```
balance = balance + 1;
```

- Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

THE UNIVERSITY of ADELAIDE

Locks: The Basic Idea

Lock variable holds the state of the lock.

- **available** (or **unlocked** or **free**)
  - No thread holds the lock.
- **acquired** (or **locked** or **held**)
  - Exactly one thread holds the lock and presumably is in a critical section.

THE UNIVERSITY of ADELAIDE

The semantics of the lock()

`lock()`

- **Try to** acquire the lock.
- If no other thread holds the lock, the thread will **acquire** the lock.
- **Enter** the *critical section*.
  - This thread is said to be the owner of the lock.
- Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

pthread Locks - mutex

The name that the POSIX library uses for a lock.

- Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&lock);
balance = balance + 1;
pthread_mutex_unlock(&lock);
```

- We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

THE UNIVERSITY of ADELAIDE

Other Examples

- **Consider multi-threaded applications that do more than increment shared balance**
- **Multi-threaded application with shared linked-list**
  - All concurrent:
    - Thread A inserting element a
    - Thread B inserting element b
    - Thread C looking up element c

Shared Linked List

```
void List_Insert(list_t *L,
                 int key) {
    node_t *new =
        malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}

int List_Lookup(list_t *L,
                 int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

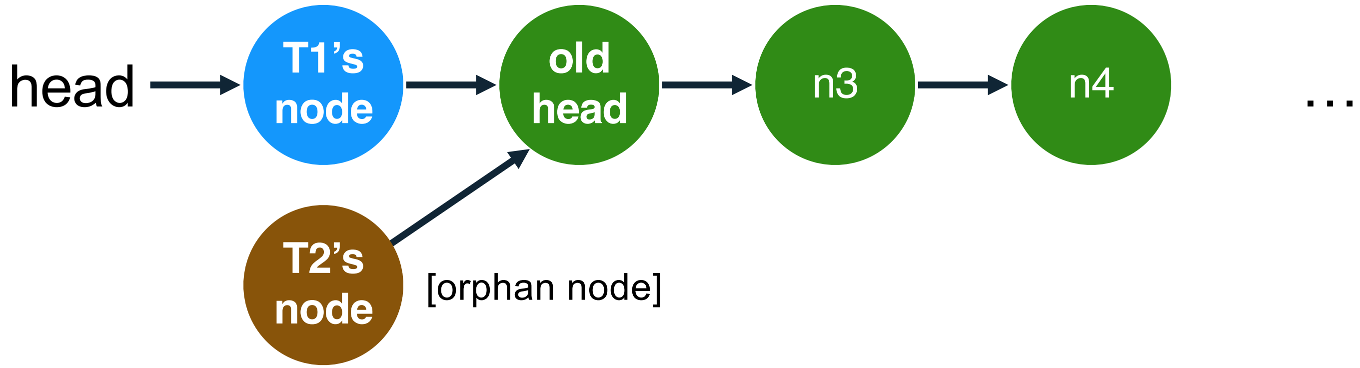
```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
}
```

What can go wrong?
Find schedule that leads to problem?

Linked-List Race

Thread 1	Thread 2
`new->key = key`	
`new->next = L->head`	
	`new->key = key`
	`new->next = L->head`
	`L->head = new`
`L->head = new`	

Resulting Linked List



Locking Linked Lists

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;


void List_Init(list_t *L) {
    L->head = NULL;
}
```

How to add locks?

`pthread_mutex_t lock;`

One lock per list

```
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock,
        NULL);
}
```

Locking Linked Lists : Approach #1

Pthread_mutex_lock(&L->lock);

Consider everything critical section
Can critical section be smaller?

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
void List_Insert(list_t *L,
                 int key) {
    node_t *new =
        malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}
int List_Lookup(list_t *L,
                int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

THE UNIVERSITY of ADELAIDE

Locking Linked Lists : Approach #2

Critical section as small as possible.

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
void List_Insert(list_t *L,
                 int key) {
    node_t *new =
        malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}
int List_Lookup(list_t *L,
                int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

THE UNIVERSITY of ADELAIDE

Locking Linked Lists : Approach #3

What about Lookup()?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

If no List_Delete() locks are not needed.

Pthread_mutex_unlock(&L->lock);

```
Void List_Insert(list_t *L,
                 int key) {
    node_t *new =
        malloc(sizeof(node_t));
    assert(new);
    new->key = key;
    new->next = L->head;
    L->head = new;
}
int List_Lookup(list_t *L,
                int key) {
    node_t *tmp = L->head;
    while (tmp) {
        if (tmp->key == key)
            return 1;
        tmp = tmp->next;
    }
    return 0;
}
```

THE UNIVERSITY of ADELAIDE

Implementing A Lock

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the hardware and the OS.

Motivation: Build them once and get them right

Lock Implementation Goals

Correctness

- Mutual exclusion
  - Only one thread in critical section at a time
- Progress (deadlock-free)
  - If several simultaneous requests, must allow one to proceed
- Bounded (starvation-free)
  - Must eventually allow each waiting thread to enter

Fairness - Each thread waits for same amount of time

Performance - CPU is not used unnecessarily (e.g., spinning)

THE UNIVERSITY
of ADELAIDE

Implementing Locks: W/ Interrupts

Disable interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void lock(lockT *l) {
    disableInterrupts();
}
```

```
void unlock(lockT *l) {
    enableInterrupts();
}
```

Disadvantages

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

THE UNIVERSITY of ADELAIDE

Implementing Locks: w/ Load+Store

First attempt: Using a flag denoting whether the lock is held or not.

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

void init(lock_t *mutex) {
        // 0 -> lock is available, 1 -> held
        mutex->flag = 0;
}

void lock(lock_t *mutex) {
        while (mutex->flag == 1)     // TEST the flag
                ;                    // spin-wait (do nothing)
        mutex->flag = 1;             // now SET it !
}

void unlock(lock_t*mutex){
        mutex->flag = 0;
}
```

Why does this not work?

THE UNIVERSITY of ADELAIDE

Race Condition with LOAD and STORE

`*flag == 0` initially

Thread 1	Thread 2
call lock()	
`while (flag == 1)`	
switch to Thread 2	
	call lock()
	`while (flag == 1)`
	`flag = 1;`
	switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

- Both threads grab lock!
- Problem: Testing lock and setting lock are not atomic

Peterson's Algorithm

- **Assume only two threads (self = 0, 1) and use just loads and stores**

```
int turn = 0;
int flag[2];

void init() {
   flag[0] = flag[1] = 0;
   turn = 0;
}

void lock() {
       flag[self] = 1;      // 'self' is the thread ID of caller
       turn = 1-self;
       while ((flag[1-self] == 1) && (turn == 1-self)) /* wait */ ;
}
void release() {
       flag[self] = 0;
}
```

THE UNIVERSITY of ADELAIDE

Peterson’s Algorithm: Intuition

- **Mutual exclusion: Enter the critical section if and only if**
  - Other thread does not want to enter
  - Other thread wants to enter, but your turn
- **Progress: Both threads cannot wait forever at while() loop**
  - Completes if other process does not want to enter
  - Other process (matching turn) will eventually finish
- **Bounded waiting**
  - Each process waits at most one critical section

Does not work on modern hardware (cache-consistency issues)

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;
}
```

THE UNIVERSITY of ADELAIDE

LOCK Implementation with XCHG

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

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    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;
}
```

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

Lock Implementation Goals

- **Correctness**
  - Mutual exclusion - Only one thread in critical section at a time
  - Progress (deadlock-free) - If several simultaneous requests, must allow one to proceed
  - Bounded (starvation-free) - Must eventually allow each waiting thread to enter
- **Fairness**
  - Each thread waits for same amount of time
- **Performance**
  - CPU is not used unnecessarily

THE UNIVERSITY *of* ADELAIDE

Compare-And-Swap

Test whether the value at the address(ptr) is equal to expected.

- If so, update the memory location pointed to by ptr with the new value.
- In either case, return the "`actual`" value at that memory location.

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

Compare-and-Swap hardware atomic instruction (C-style)

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

Spin lock with compare-and-swap

Fairness: Ticket Locks

- Idea: reserve each thread's turn to use a lock.
- Each thread spins until their turn.
- **Atomically increment a value while returning the old value at a particular address.**

```
int FetchAndAdd(int *ptr) {
        int old = *ptr;
        *ptr = old + 1;
        return old;
}
```

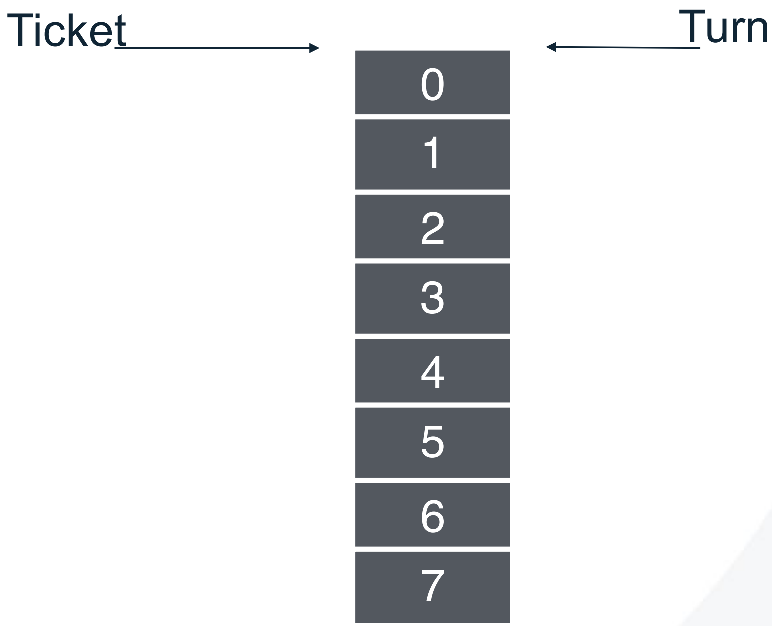
Fetch-And-Add Hardware atomic instruction (C-style)

- Acquire: Grab ticket; Spin while not thread's ticket != turn
- Release: Advance to next turn

THE UNIVERSITY of ADELAIDE

Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():



Ticket Lock Example

A lock(): gets ticket 0, spins until turn = 0 →runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs

0
1
2
3
4
5
6
7

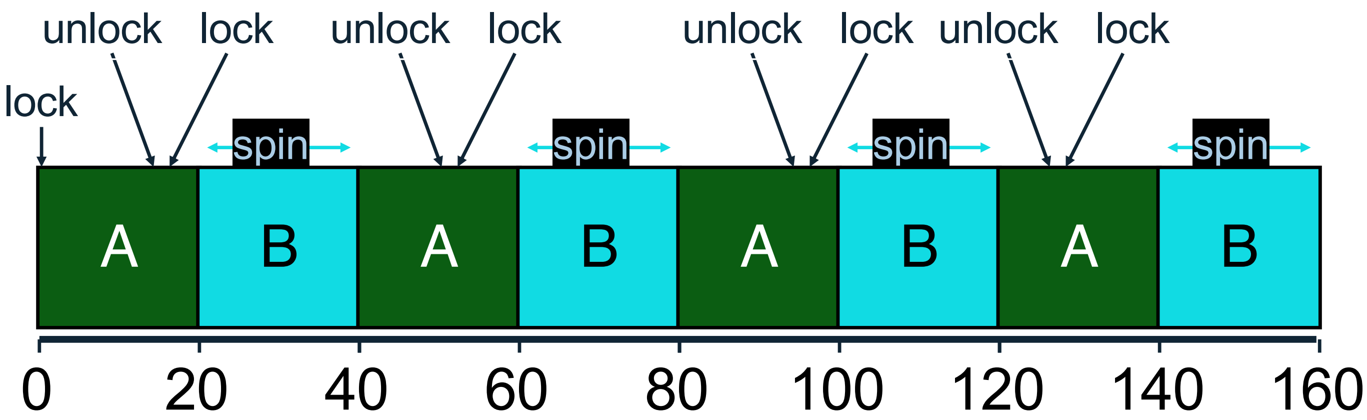
THE UNIVERSITY of ADELAIDE

Ticket Lock Implementation

```
typedef struct __lock_t {
        int ticket;
        int turn;
} lock_t;


void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
            ; // spin
}


void release (lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Basic Spinlocks are Unfair



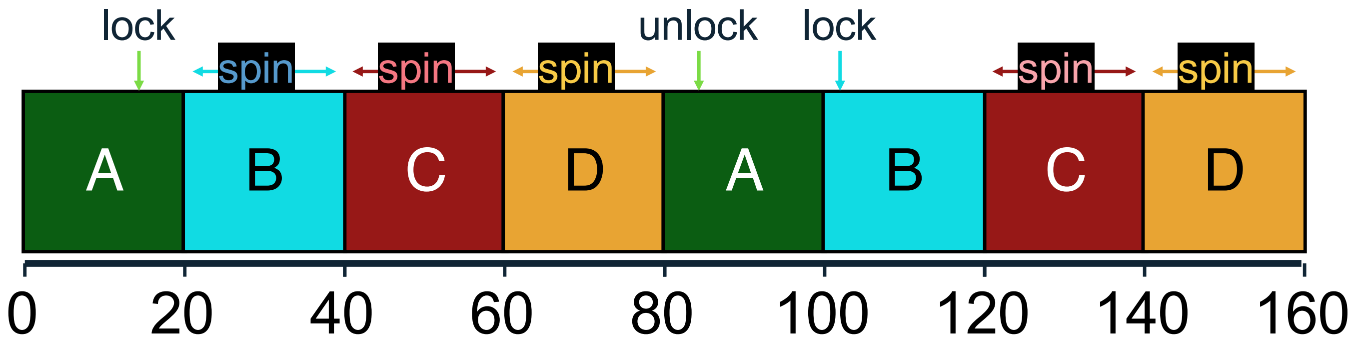
Scheduler is unaware of locks/unlocks

Spinlock Performance

- **Fast when…**
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch
- **Slow when…**
  - one CPU
  - locks held a long time
  - disadvantage: spinning is wasteful

THE UNIVERSITY of ADELAIDE

CPU Scheduler is Ignorant



CPU scheduler may run **B**, **C**, **D** instead of **A**
even though **B**, **C**, **D** is waiting for **A**

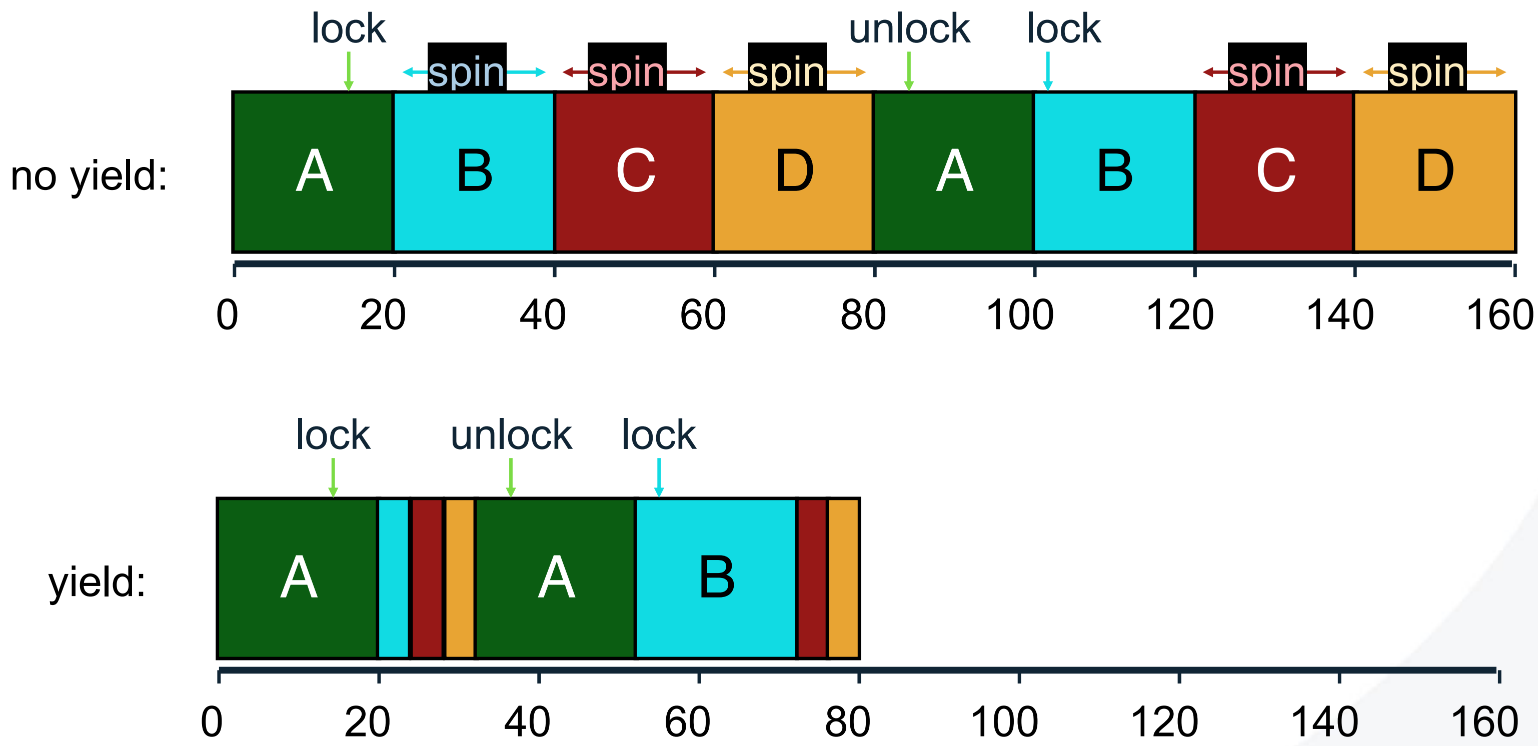
Ticket Lock with Yield()

```
typedef struct __lock_t {
        int ticket;
        int turn;
} lock_t;

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
            yield(); // give up the CPU
}

void release (lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

THE UNIVERSITY of ADELAIDE

Yield Instead of Spin



THE UNIVERSITY of ADELAIDE

Lock Evaluation

- **How to tell if a lock implementation is good?**
  - Fairness:
    - Do processes acquire lock in same order as requested?
  - Performance
    - Two scenarios:
      - low contention (fewer threads, lock usually available)
      - high contention (many threads per CPU, each contending)

Lock Implementation: Block when Waiting

Lock implementation removes waiting threads from scheduler ready queue.

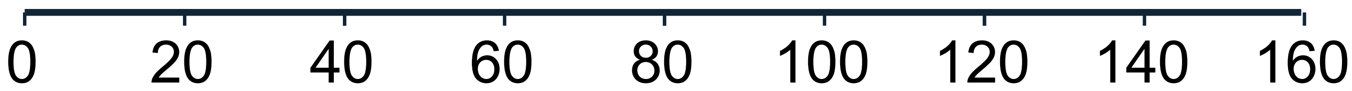
Scheduler runs any thread that is **ready**

Good separation of concerns

RUNNABLE: A, B, C, D

RUNNING: <empty>

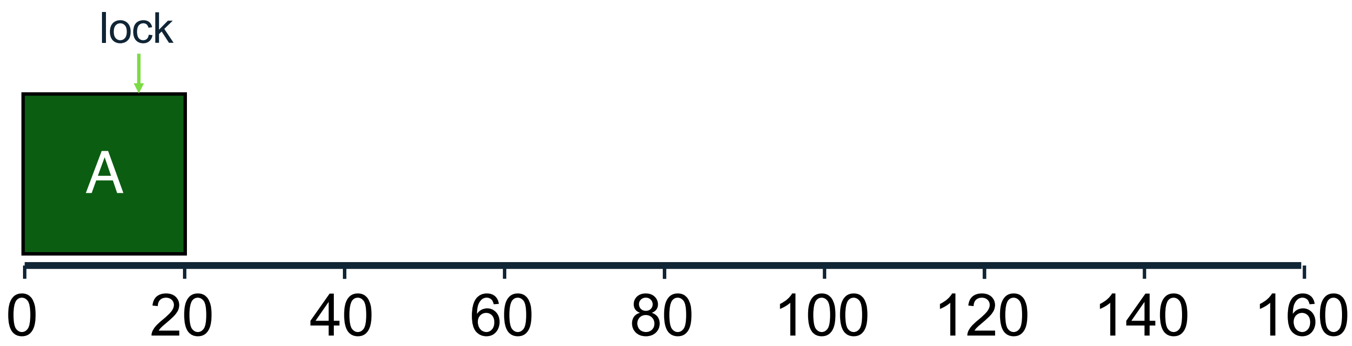
WAITING: <empty>



RUNNABLE: B, C, D

RUNNING: A

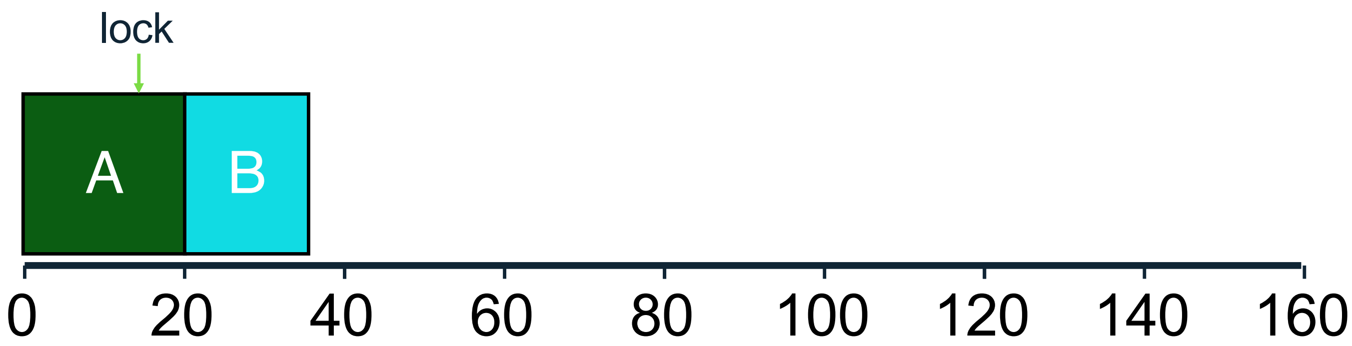
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING: B

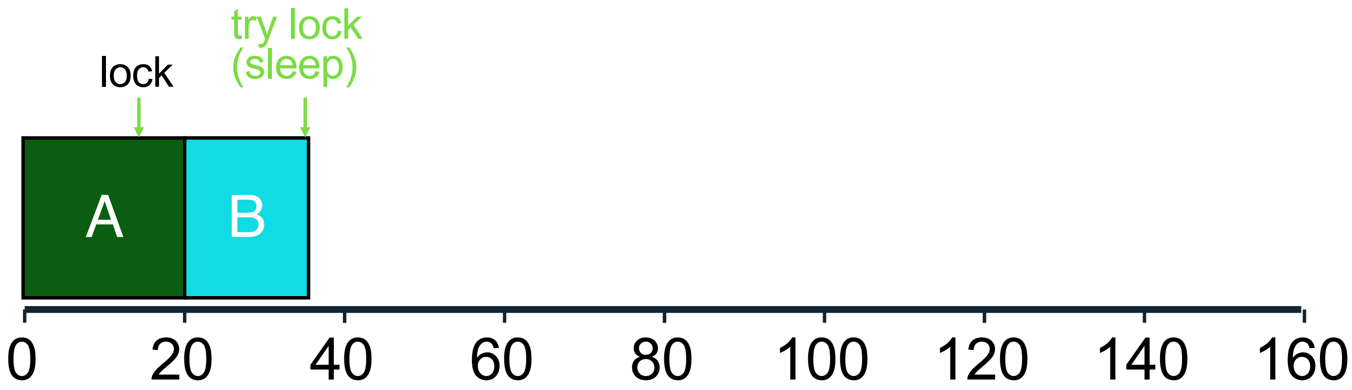
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING:

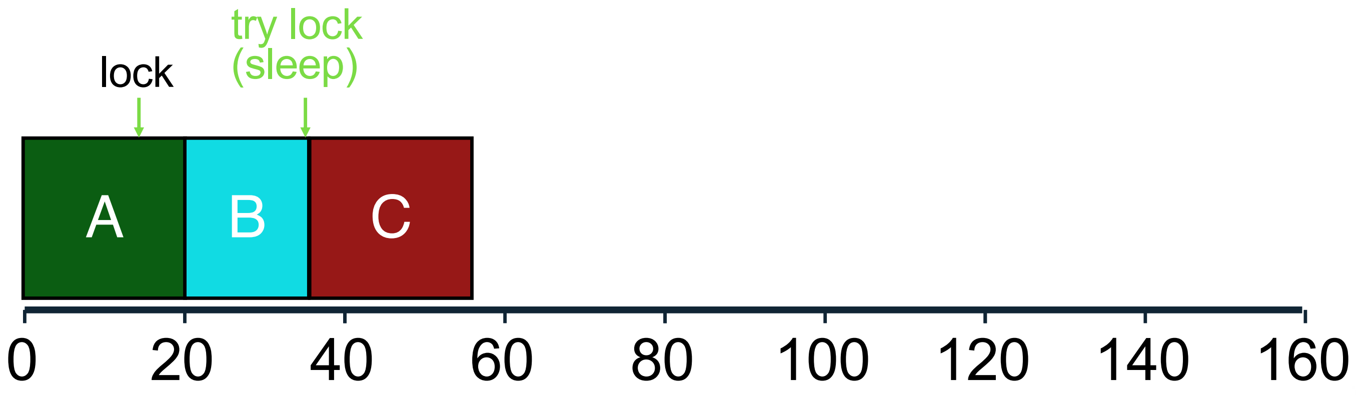
WAITING: B



RUNNABLE: D, A

RUNNING: C

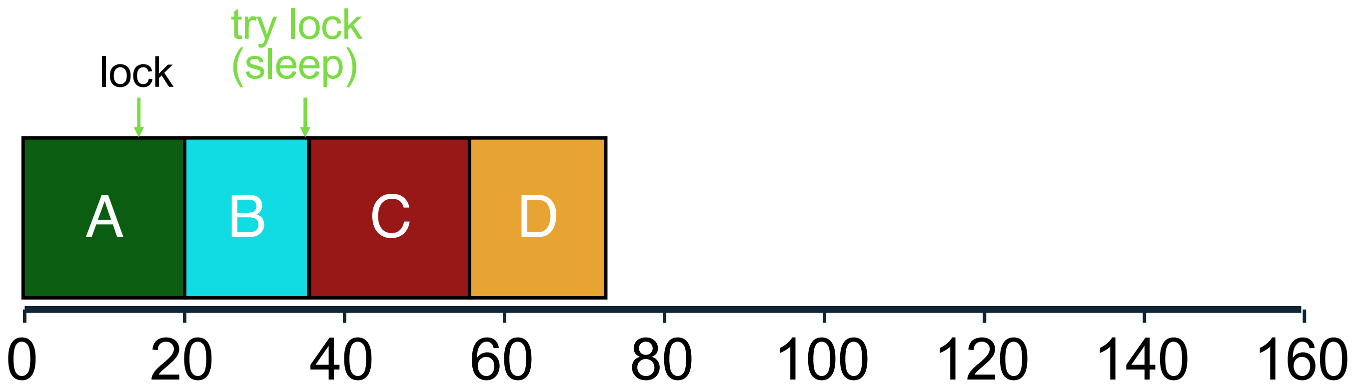
WAITING: B



RUNNABLE: A, C

RUNNING: D

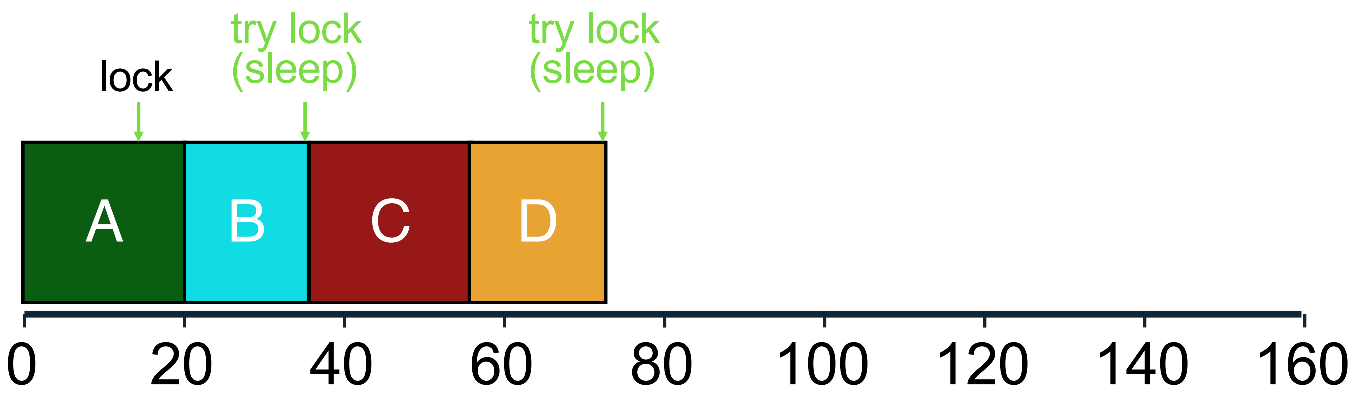
WAITING: B



RUNNABLE: A, C

RUNNING:

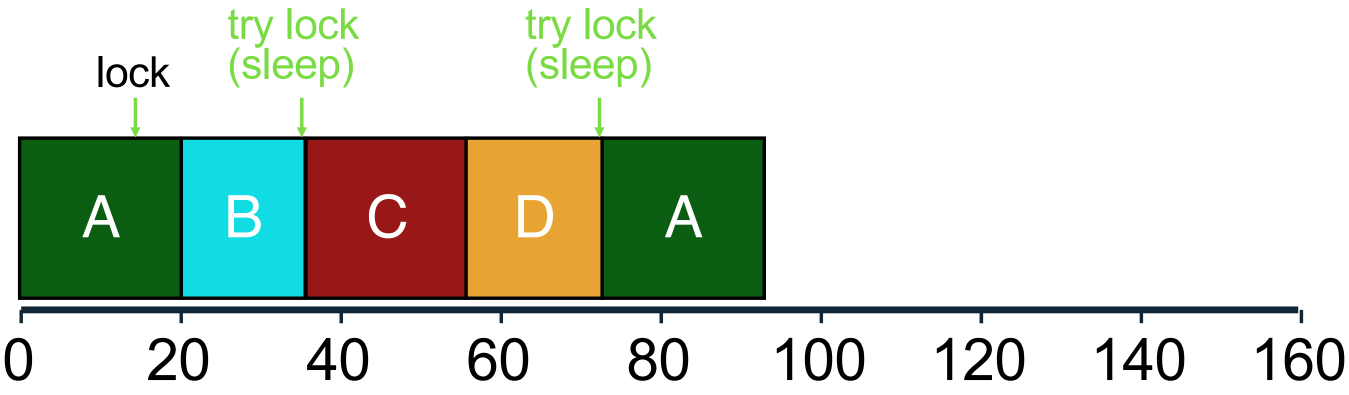
WAITING: B, D



RUNNABLE: C

RUNNING: A

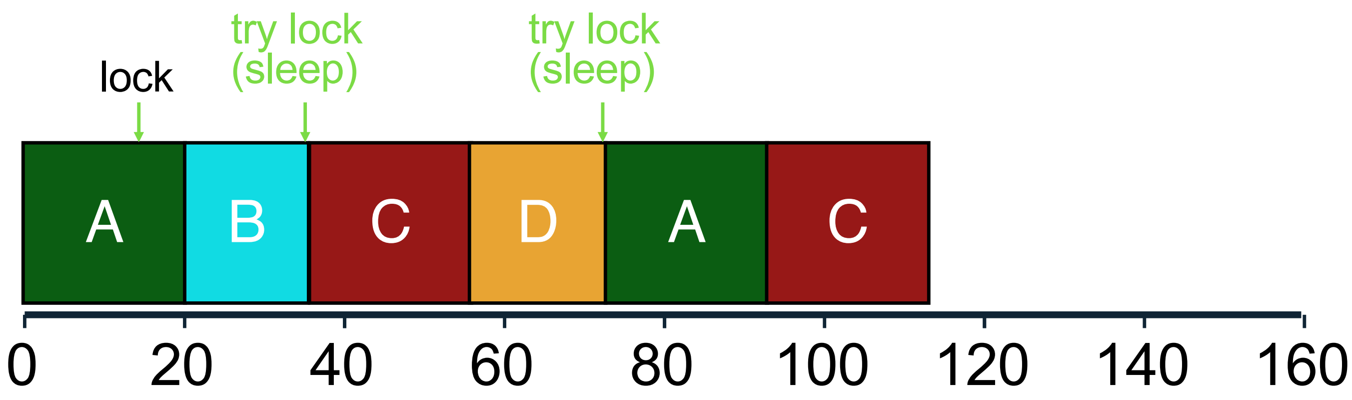
WAITING: B, D



RUNNABLE: A

RUNNING: C

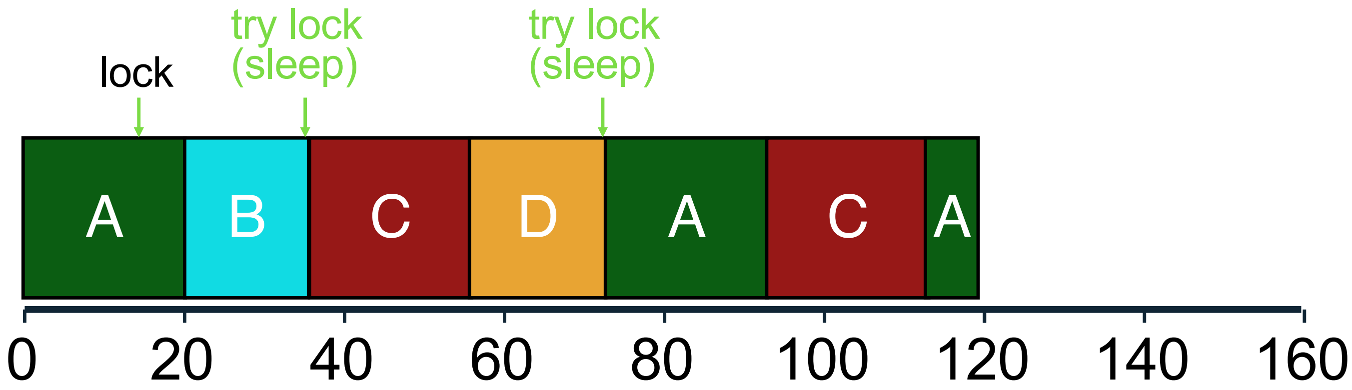
WAITING: B, D



RUNNABLE: C

RUNNING: A

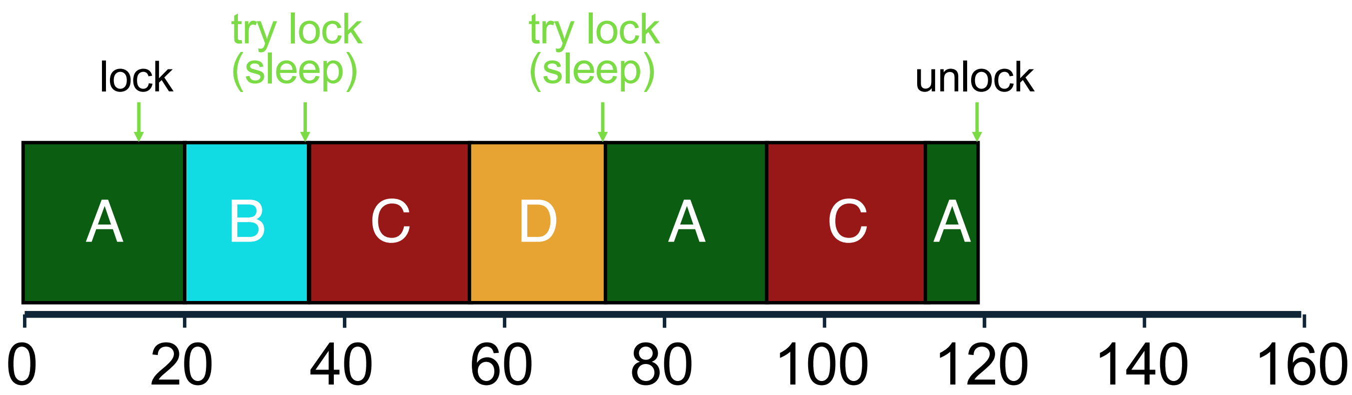
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

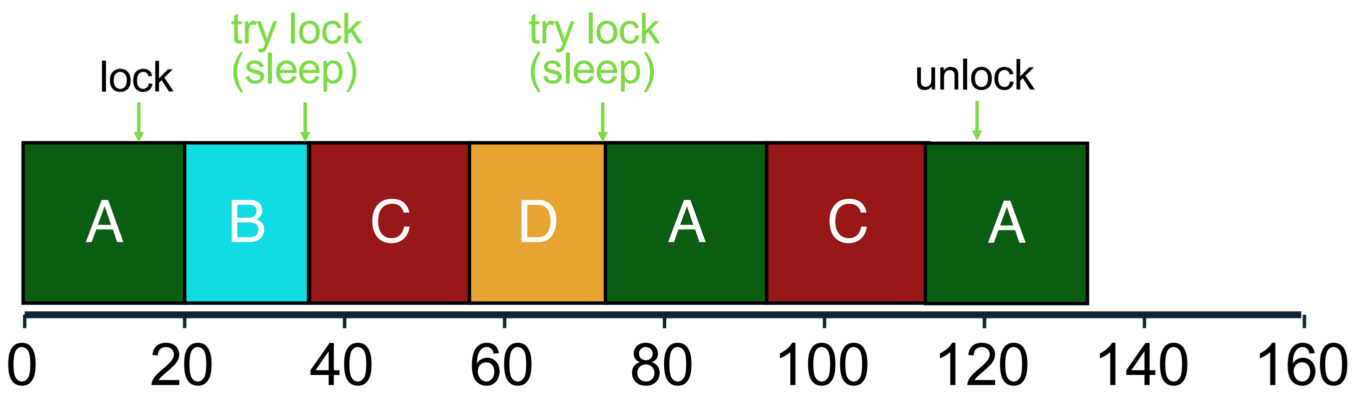
WAITING: D



RUNNABLE: B, C

RUNNING: A

WAITING: D

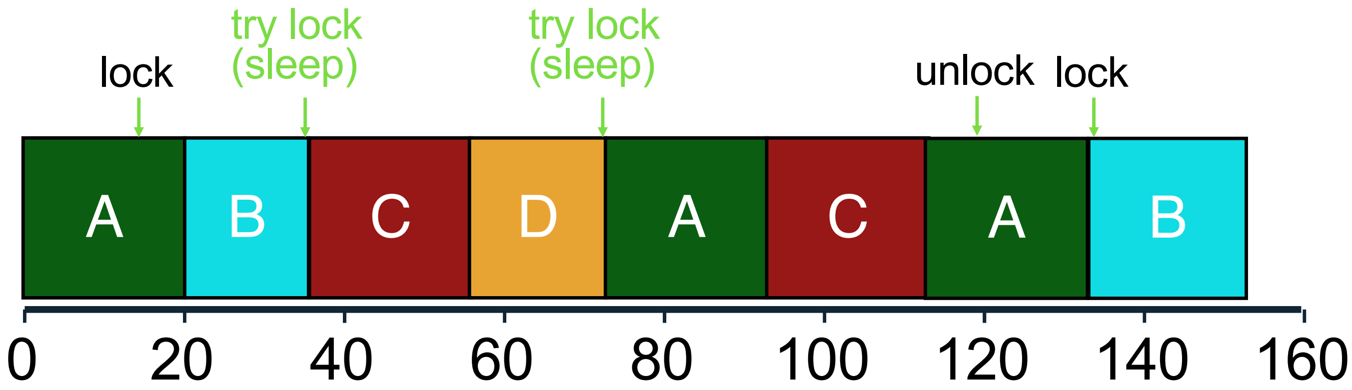


THE UNIVERSITY of ADELAIDE

RUNNABLE: C, A

RUNNING: B

WAITING: D



THE UNIVERSITY of ADELAIDE

Spinlock Performance

- **Waste of CPU cycles?**
  - 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

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

(a) Why is **guard** used?
(b) Why okay to **spin** on guard?
(c) In release(), why not set flag=0 when unpark?
(d) What is the race condition?

```
void acquire(lock_t *l) {
   while (TestAndSet(&l->guard, 1) == 1);
   if (l->flag) {
         queue_add(l->q, gettid());
         l->guard = 0;
         park();       // blocked
   } else {
         l->lock = 1;  //lock is acquired
         l->guard =0;
   }
}

void release(lock_t *l) {
   while (TestAndSet(&l->guard, 1) == 1);
   if (queue_empty(l->q)) l->flag=0;
   else unpark(queue_remove(l->q));
   l->guard = 0;
}
```

THE UNIVERSITY of ADELAIDE

Race Condition

Thread 1 (in "**acquire**")

```
if (l->flag) {
    queue_add(l->q, gettid());
     l->guard = 0;



     park();     // block
```

Thread 2 (in "**release**")

```
while (TestAndSet(&l->guard, 1) == 1);
   if (queue_empty(l->q)) l->flag=0;
   else unpark(queue_remove(l->q));
   l->guard = 0;
```

Problem: Guard not held when call park()
Unlocking thread may unpark() before other park()

THE UNIVERSITY of ADELAIDE

Lock Implementation: Block when Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
     m->flag  = 0;
     m->guard = 0;
     queue_init(m->q);
}
```

```
void acquire(lock_t *l) {
   while (TestAndSet(&l->guard, 1) == 1);
   if (l->flag) {
         queue_add(l->q, gettid());
         setpark(); // notify of plan
         l->guard = 0;
         park();     // blocked
   } else {
         l->lock = 1;  //lock is acquired
         l->guard =0;
   }
}

void release(lock_t *l) {
   while (TestAndSet(&l->guard, 1) == 1);
   if (queue_empty(l->q)) l->flag=0;
   else unpark(queue_remove(l->q));
   l->guard = 0;
}
```

setpark() fixes race condition

Spin-Waiting vs Blocking

- **Each approach is better under different circumstances**
  - Uniprocessor
    - Waiting process is scheduled --> Process holding lock isn’t
    - Waiting process should always relinquish processor
    - Associate queue of waiters with each lock
  - Multiprocessor
    - Waiting process is scheduled --> Process holding lock might be
    - Spin or block depends on how long, t, before lock is released
    - Lock released quickly --> Spin-wait
    - Lock released slowly --> Block
    - Quick and slow are relative to context-switch cost, C

THE UNIVERSITY of ADELAIDE

When to Spin-Wait? When to Block?

- Knowing how long, **t,** before the lock is released can determine optimal behaviour
- How much CPU time is wasted when spin-waiting? t
- How much is wasted when blocking? C
- What is the best action when $t<C$? spin-wait
- When $t>C$? block

- Problem: Requires knowledge of the future; too much overhead to make any special prediction

THE UNIVERSITY *of* ADELAIDE

Futex

Linux provides a futex. More functionality goes into the kernel (i.e. it is a system call)

- `futex_wait(address, expected)`
  - Put the calling thread to sleep
  - If the value at address is not equal to expected, the call returns immediately.
- `futex_wake(address)`
  - Wake one thread that is waiting on the queue.

Condition Variables

Concurrency Objectives

- **Mutual exclusion (e.g., A and B don’t run at same time)**
  - solved with locks
- **Ordering (e.g., B runs after A does something)**
  - solved with condition variables and semaphores

Condition Variables

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

Condition Variables

- **wait(**`cond_t *cv, mutex_t *lock`**)**
  - assumes the lock is held when wait() is called
  - 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, just return, doing nothing

THE UNIVERSITY of ADELAIDE

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

THE UNIVERSITY of ADELAIDE

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);     // x
        Cond_wait(&c, &m);  // y
        Mutex_unlock(&m);   // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);     // a
        Cond_signal(&c);    // b
        Mutex_unlock(&m);   // c
}
```

Example schedule:

Parent:	x	y				z
Child:			a	b	c	

Works!

Join Implementation: Attempt 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);     // x
        Cond_wait(&c, &m);  // y
        Mutex_unlock(&m);   // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);     // a
        Cond_signal(&c);    // b
        Mutex_unlock(&m);   // c
}
```

Example broken schedule:

Parent:				x	y
Child:	a	b	c		

Keep State

- **Keep state** in addition to CV's!
- CV's are used to signal threads when state changes
- If state is already as needed, thread doesn't wait for a signal!

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m);// y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        done = 1;             // a
        Cond_signal(&c);      // b
}
```

Fixes previously broken ordering:

Parent:			w	x	y	z
Child:	a	b				

THE UNIVERSITY of ADELAIDE

Join Implementation: Attempt 2

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m);// y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        done = 1;           // a
        Cond_signal(&c);    // b
}
```

But you can construct ordering that does not work:

Parent:	w	x			y	… sleep forever
Child:			a	b		

Join Implementation: Correct

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m);// y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);       // a
        done = 1;             // b
        Cond_signal(&c);      // c
        Mutex_unlock(&m);     // d
}
```

Parent:	w	x	y				z
Child:				a	b	c	

Use mutex to ensure no race condition

THE UNIVERSITY of ADELAIDE

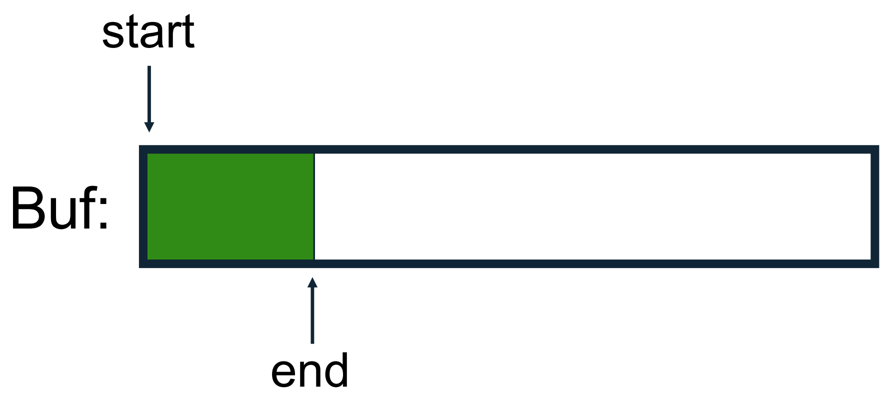
Producer/Consumer Problem

Example: UNIX Pipes

- **A pipe may have many writers and readers**
- **Internally, there is a finite-sized buffer**
- **Writers add data to the buffer**
  - Writers have to wait if buffer is full
- **Readers remove data from the buffer**
  - Readers have to wait if buffer is empty

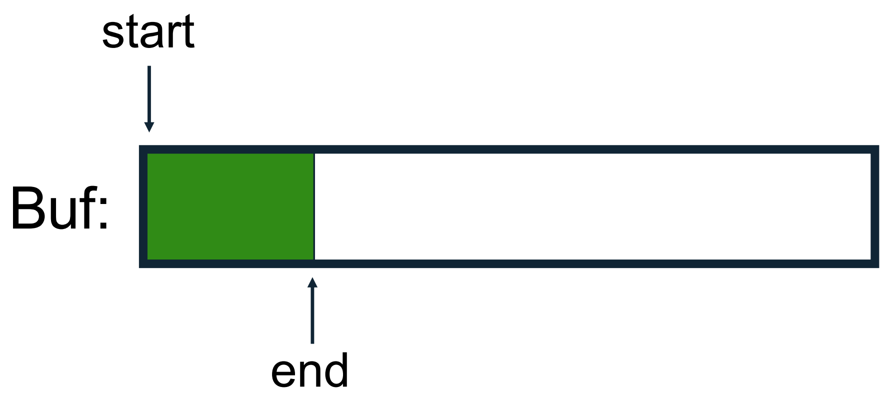
Example: UNIX Pipes

Example: UNIX Pipes

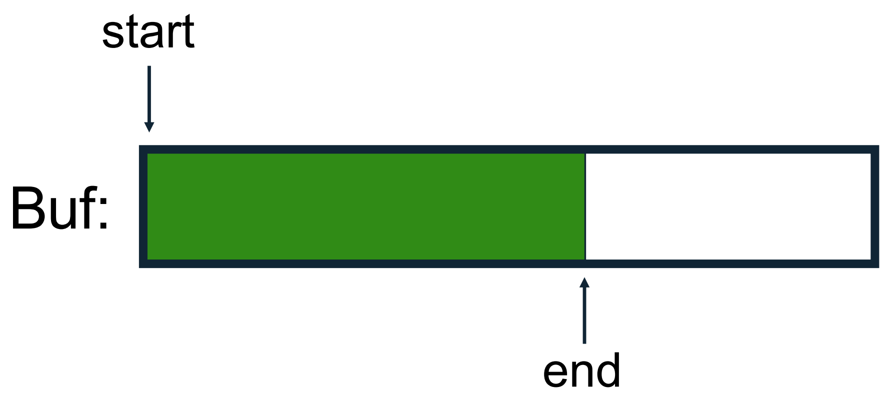


write!

Example: UNIX Pipes

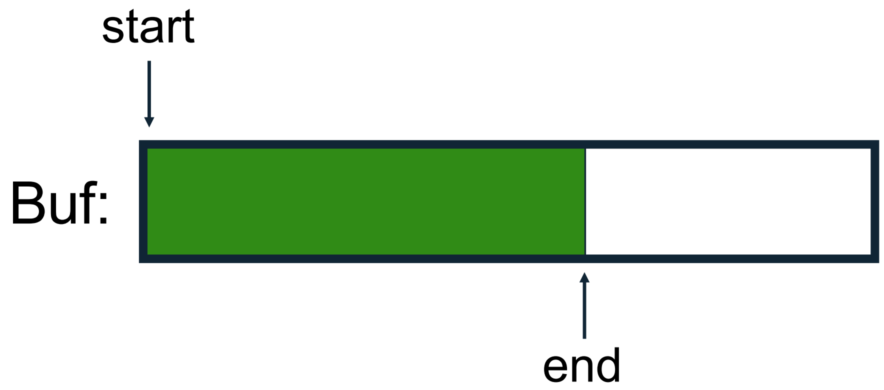


Example: UNIX Pipes

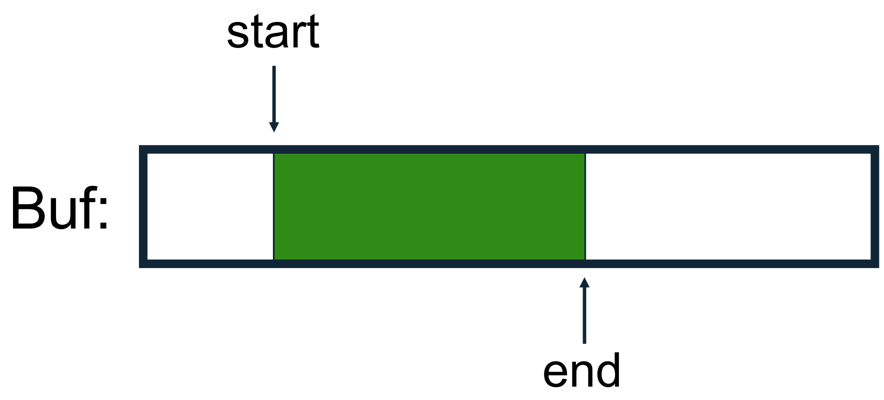


write!

Example: UNIX Pipes

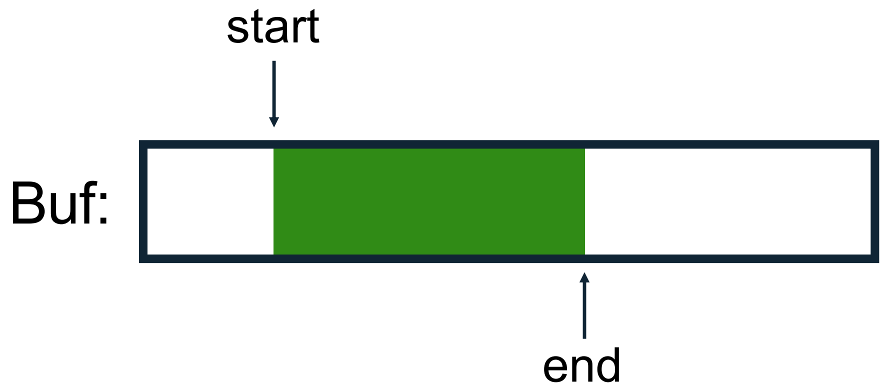


Example: UNIX Pipes



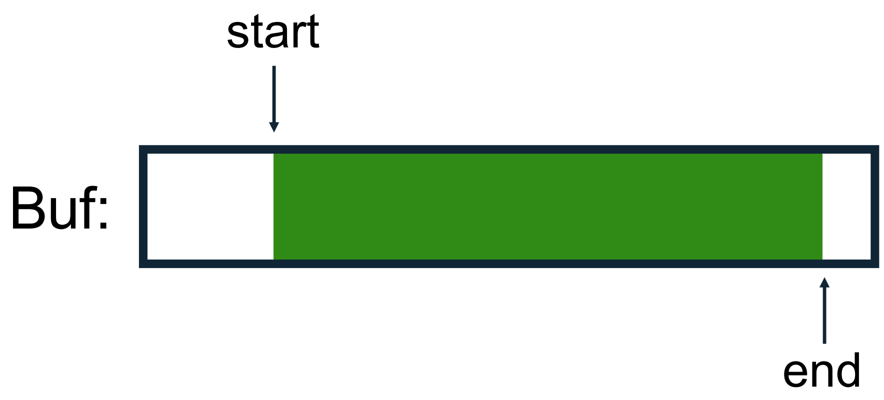
read!

Example: UNIX Pipes



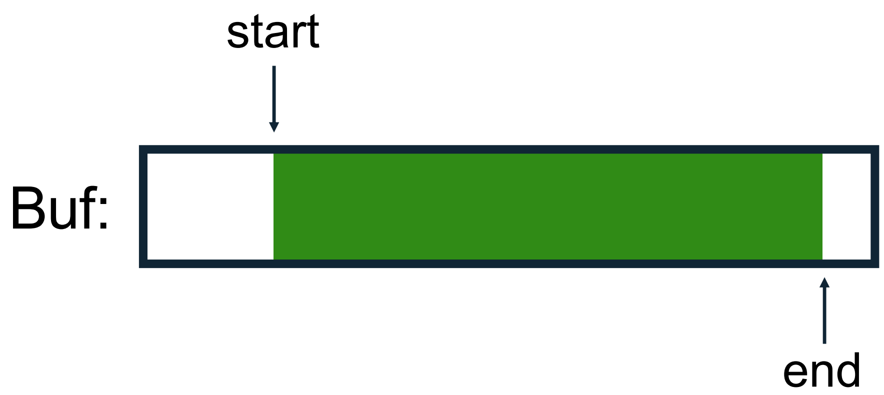
THE UNIVERSITY of ADELAIDE

Example: UNIX Pipes

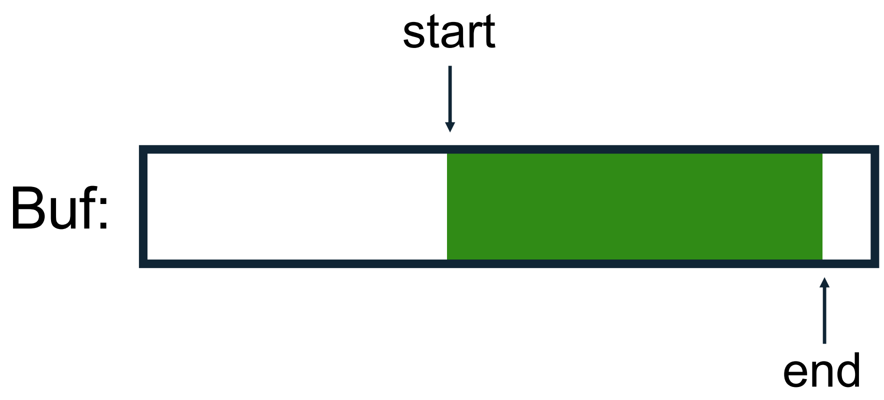


write!

Example: UNIX Pipes

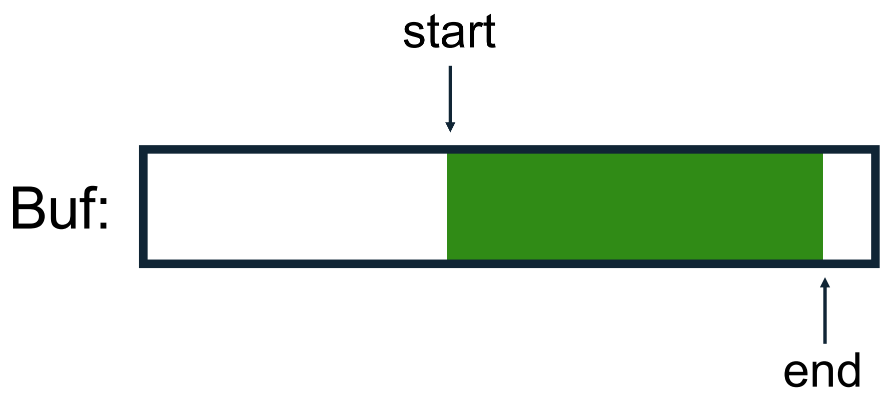


Example: UNIX Pipes

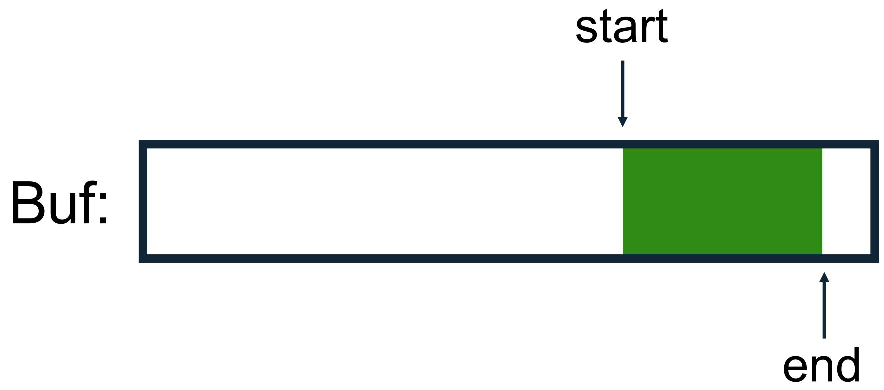


read!

Example: UNIX Pipes

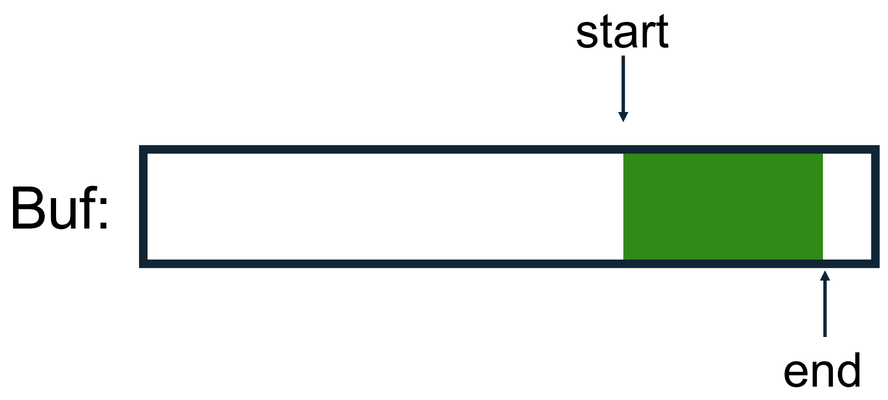


Example: UNIX Pipes



read!

Example: UNIX Pipes



Example: UNIX Pipes

read!

Example: UNIX Pipes

read!

note: readers must wait

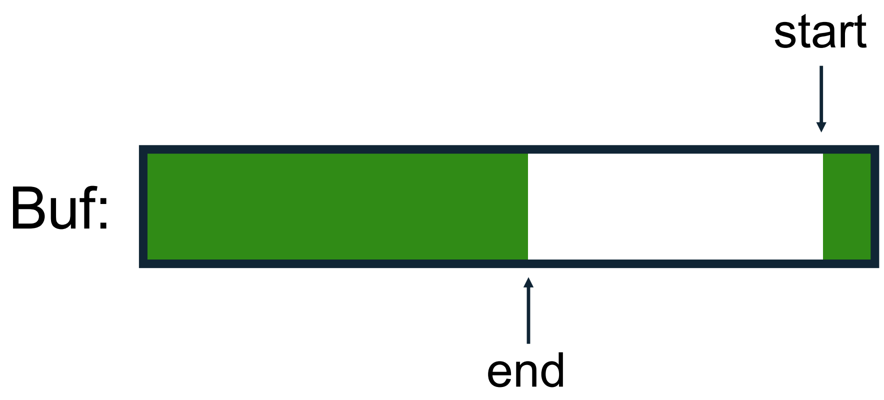
Example: UNIX Pipes

Example: UNIX Pipes

write!

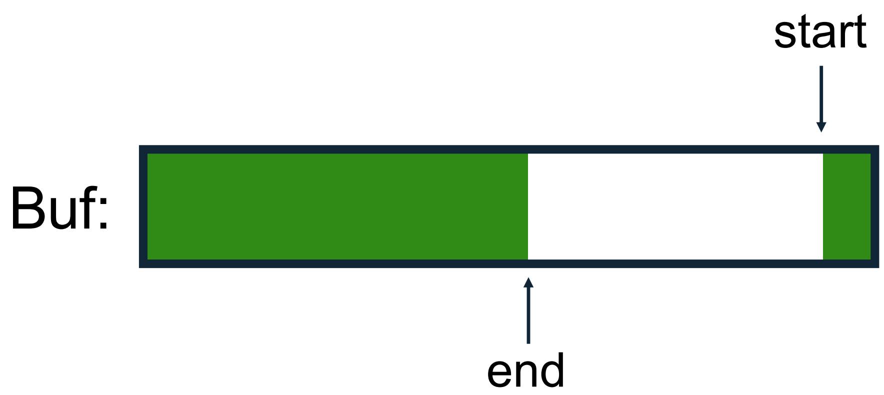
Example: UNIX Pipes

Example: UNIX Pipes



write!

Example: UNIX Pipes



Example: UNIX Pipes

write!

Example: UNIX Pipes

write!

note: writers must wait

Example: UNIX Pipes

Example: UNIX Pipes

read!

Example: UNIX Pipes

- **Implementation**
  - reads/writes to buffer require locking
  - when buffers are full, writers must wait
  - when buffers are empty, readers must wait

THE UNIVERSITY of ADELAIDE

Producer/Consumer Problem

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems (e.g., web servers)

General strategy use condition variables to:

- make producers wait when buffers are full
- make consumers wait when there is nothing to consume

Produce/Consumer Example

- **Start with easy case:**
  - 1 producer thread
  - 1 consumer thread
  - 1 shared buffer to fill/consume (max = 1)

- Numfill = number of buffers currently filled
- Examine slightly broken code to begin…

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY *of* ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[SLEEPING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =1

[SLEEPING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =1

[SLEEPING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[SLEEPING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNABLE]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =0

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[SLEEPING]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

THE UNIVERSITY of ADELAIDE

numfull =1

[RUNNING]

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

What about 2 consumers?

Can you find a problematic timeline with 2 consumers (still 1 producer)?

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);              //p1
        while(numfull == max)        //p2
            Cond_wait(&cond, &m);//p3
        do_fill(i);                  //p4
        Cond_signal(&cond);          //p5
        Mutex_unlock(&m);            //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);              //c1
        while(numfull == 0)          //c2
            Cond_wait(&cond, &m);//c3
        int tmp = do_get();          //c4
        Cond_signal(&cond);          //c5
        Mutex_unlock(&m);            //c6
        printf("%d\n", tmp);         //c7
    }
}
```

wait() wait() signal() wait() signal()

Producer: p1 p2 p4 p5 p6 p1 p2 p3
Consumer1: c1 c2 c3
Consumer2: c1 c2 c3 c2. c4 c5

does last signal wake producer or consumer2?

How to wake the right thread?

- **One solution:**
  - Wake all the threads!

Waking All Waiting Threads

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- 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, just return, doing nothing

broadcast(cond_t *cv)
- wake all waiting threads (if >= 1 thread is waiting)
- if there are no waiting thread, just return, doing nothing

any disadvantage?

THE UNIVERSITY of ADELAIDE

Example Need for Broadcast

```
void *allocate(int size) {
        mutex_lock(&m);
        while (bytesLeft < size)
                cond_wait(&c);
        …
}
```

```
void free(void *ptr, int
size) {
    …
    cond_broadcast(&c)
    …
}
```

How to wake the right thread?

- **One solution:**
  - Wake all the threads!

- Better solution (usually):

use two condition variables

THE UNIVERSITY of ADELAIDE

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);                //p1
        while(numfull == max)          //p2
            Cond_wait(&empty, &m);//p3
        do_fill(i);                    //p4
        Cond_signal(&fill);            //p5
        Mutex_unlock(&m);              //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);                //c1
        while(numfull == 0)            //c2
            Cond_wait(&fill, &m);//c3
        int tmp = do_get();            //c4
        Cond_signal(&empty);           //c5
        Mutex_unlock(&m);              //c6
        printf("%d\n", tmp);           //c7
    }
}
```

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

THE UNIVERSITY of ADELAIDE

Producer/Consumer: Two CVs and WHILE

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);               //p1
        while(numfull == max)         //p2
            Cond_wait(&empty, &m);//p3
        do_fill(i);                   //p4
        Cond_signal(&fill);           //p5
        Mutex_unlock(&m);             //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);               //c1
        while(numfull == 0)           //c2
            Cond_wait(&fill, &m);//c3
        int tmp = do_get();           //c4
        Cond_signal(&empty);          //c5
        Mutex_unlock(&m);             //c6
        printf("%d\n", tmp);          //c7
    }
}
```

Is this correct?

Correct!

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every `do_fill()`
- a producer will get to run after every `do_get()`

THE UNIVERSITY of ADELAIDE

Recheck Assumptions

- Whenever a lock is acquired, recheck assumptions about state!
- Possible for another thread to grab lock in between signal and wakeup from wait
- Note that some libraries also have “spurious wakeups” (may wake multiple waiting threads at signal or at any time)

Summary: Rules for CVs

- Keep state in addition to CV's
- Always do wait/signal with lock held
- Whenever thread wakes from waiting, recheck state