Synchronization
Cache Coherence

Chris Rossbach
CS378H Fall 2018
9/12/18
Today

• Questions?

• Administrivia
  • Lab 1 due a few minutes ago!

• Material for the day
  • Cache coherence
  • Lock implementation
  • Blocking synchronization

• Acknowledgements
  • Thanks to Gadi Taubenfield: I borrowed from some of his slides on barriers
Faux Quiz (answer any 2, 5 min)

• What is the difference between spinning/busy-wait and blocking synchronization?
• Can you write shared memory parallel applications using single-threaded processes only?
• How do you choose between spinlock/mutex on a multi-processor?
• Define the states of the MESI protocol. Is the E state necessary? Why or why not?
• What is bus locking?
• What is the difference between Mesa and Hoare monitors?
• Why recheck the condition on wakeup from a monitor wait?
• How can you build barriers with spinlocks?
• How can you build barriers with monitors?
• What is the difference between a mutex and a semaphore?
Review: correctness conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- **Bounded waiting**
  - A thread that enters the entry section enters the critical section within some **bounded number of operations**.
  - If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i’s request is granted

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

Mutex, spinlock, etc. are ways to implement these

---

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider

Implementing Locks

```cpp
int lock_value = 0;
int* lock = &lock_value;

Lock::Acquire() {
    while (*lock == 1) //spin
        *lock = 1;
}

Lock::Release() {
    *lock = 0;
}
```

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

Completely and utterly broken. How can we fix it?
HW Support for Read-Modify-Write (RMW)

Preview of Techniques:

- Bus locking
- Single Instruction ISA extensions
  - Test&Set
  - CAS: Compare & swap
  - Exchange, locked increment, locked decrement (x86)
- Multi-instruction ISA extensions:
  - LLSC: (PowerPC, Alpha, MIPS)
  - Transactional Memory (x86, PowerPC)

bool rmw(addr, value) {
    atomic {
        tmp = *addr;
        newval = modify(tmp);
        *addr = newval;
    }
}

IDEA: hardware implements something like:

Why is that hard? How can we do it?
Multiprocessor Cache Coherence

**Physics** | **Concurrency**

\[ F = ma \sim coherence \]
Cache Coherence

- P1: read X
- P2: read X
- P2: X++
- P3: read X
Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic
Cache Coherence: single-thread

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try:  load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try:  load lock, R0
    test R0
    bnz try
    store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try:  load lock, R0
    test R0
    bnz try
    store lock, 1
}

SAFE!
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
       test R0
       bnz try
       store lock, 1
}
Implementing locks requires read-modify-write operations

Required effect is:
- An atomic and isolated action
  1. read memory location **AND**
  2. write a new value to the location
- **RMW** is *very tricky* in multi-processors
- Cache coherence alone doesn’t solve it

```c
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
       test R0
       bnz try
       store lock, 1
}
```
Essence of HW-supported RMW

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:
    load lock, R0
    test R0
    bnz try
    store lock, 1
}

Make this into a single (atomic hardware instruction)
OR
A set of instructions with well-defined protocol
### HW Support for Read-Modify-Write (RMW)

<table>
<thead>
<tr>
<th>Test &amp; Set</th>
<th>CAS</th>
<th>Exchange, locked increment/decrement,</th>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
<td>x86</td>
<td>PPC, Alpha, MIPS</td>
</tr>
</tbody>
</table>

```c
int TST(addr) {
    atomic {
        ret = *addr;
        if(!*addr)
            *addr = 1;
        return ret;
    }
}

int atomic {
    if(*addr == old) {
        *addr = new;
        return true;
    }
    return false;
}
```

```c
bool cas(addr, old, new) {
    atomic {
        if(*addr == old) {
            *addr = new;
            return true;
        }
        return false;
    }
}
```

```c
int atomic {
    ret = *addr;
    if(*addr == ret) {
        *addr = val;
        return ret;
    }
    return false;
}
```

```c
void CAS_lock(lock) {
    while(CAS(&lock, 0, 1) != true);
}
```

```c
bool LLSC(addr, val) {
    ret = *
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
}
```
HW Support for RMW: LL-SC

**LLSC: load-linked store-conditional**

<table>
<thead>
<tr>
<th>PPC, Alpha, MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool LLSC(addr, val)</td>
</tr>
<tr>
<td>ret = *addr;</td>
</tr>
<tr>
<td>atomic {</td>
</tr>
<tr>
<td>if(*addr == ret) {</td>
</tr>
<tr>
<td>*addr = val;</td>
</tr>
<tr>
<td>return true;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>return false;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

```cpp
def LLSC_lock(lock):
    while True:
        old = load-linked(lock)
        if old == 0 and store-cond(lock, 1):
            return
```

- **SIDEBAR:**
  Transactional Memory extends LLSC idea to multiple variables

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged
LLSC Lock Action Zone

P1
lock: 0

lock:

P2
lock: 1

lock:

P1
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2
lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}
LLSC Lock Action Zone II

\[
\text{lock: } \text{[SM]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } 0
\]

\[
\text{lock: } 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]

\[
\text{lock: } \text{[SIL]} 0
\]
Implementing Locks with Test&set

```cpp
int lock_value = 0;
int* lock = &lock_value;

Lock::Acquire() {
    while (test&set(lock) == 1)  //spin
        ;
}

Lock::Release() {
    *lock = 0;
}
```

What is the problem with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

(test & set  ~ CAS ~ LLSC)
Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting
What happens to lock variable’s cache line when different cpu’s contend?

Load can stall

CPU A
while (test&set (lock));
// in critical region
lock: 1
...

CPU B
while (test&set (lock));
...

Main Memory

0xF0 lock: 1
0xF4 ...

L1 L2

L1 L2

- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line “ping pongs” amongst contenders
TTS: Reducing busy wait contention

Test&Set

```cpp
Lock::Acquire() {
    while (test&set(lock) == 1);
}
```

**Busy-wait on in-memory copy**

```cpp
Lock::Release() {
    *lock = 0;
}
```

Test&Test&Set

```cpp
Lock::Acquire() {
    while(1) {
        while (*lock == 1); // spin just reading
        if (test&set(lock) == 0) break;
    }
}
```

**Busy-wait on cached copy**

```cpp
Lock::Release() {
    *lock = 0;
}
```

• What is the problem with this?
  • A. CPU usage  B. Memory usage  C. Lock::Acquire() latency
  • D. Memory bus usage  E. Does not work
Test & Test & Set with Memory Hierarchies

What happens to lock variable’s cache line when different cpu’s contend for the same lock?
Test & Test & Set with Memory Hierarchies

What happens to lock variable’s cache line when different cpu’s contend for the same lock?

Wait...why all this spinning?
How can we improve over busy-wait?

```cpp
Lock::Acquire() {
    while(1) {
        while (*lock == 1); // spin just reading
        if (test&set(lock) == 0) break;
    }
}
```
Mutex

• Same abstraction as spinlock
• But is a “blocking” primitive
  • Lock available → same behavior
  • Lock held → yield/block
• Many ways to yield
• Simplest case of semaphore

```c
void cm3_lock(u8_t* M) {
    u8_t LockedIn = 0;
    do {
        if (__LDREXB(Mutex) == 0) {
            // unlocked: try to obtain lock
            if ( __STREXB(1, Mutex) ) { // got lock
                __CLEX(); // remove __LDREXB() lock
                LockedIn = 1;
            } else task_yield(); // give away cpu
        } else task_yield(); // give away cpu
    } while(!LockedIn);
}
```
Lock Pitfalls...

A(prio-0) → lock(my_lock);
B(prio-100) → lock(my_lock);

Solution?

**Priority inheritance:** A runs at B’s priority
MARS pathfinder failure:

Other ideas?
Can you build a lock without coherence?

Dekker’s Algorithm

variables
  wants_to_enter : array of 2 booleans
  turn : integer

wants_to_enter[0] = false
wants_to_enter[1] = false
turn = 0  // or 1

p0:
  wants_to_enter[0] = true
  while wants_to_enter[1] {
    if turn = 0 {
      wants_to_enter[0] = false
      while turn = 0 {
        // busy wait
      }
      wants_to_enter[0] = true
    }
  }
  // critical section
  ...
  turn = 1
  wants_to_enter[0] = false
  // remainder section

p1:
  wants_to_enter[1] = true
  while wants_to_enter[0] {
    if turn = 1 {
      wants_to_enter[1] = false
      while turn = 1 {
        // busy wait
      }
      wants_to_enter[1] = true
    }
  }
  // critical section
  ...
  turn = 0
  wants_to_enter[1] = false
  // remainder section

Initially:  c1, c2, turn = 1, 1, 1

Process 1:
  c1 := 0

  c2 := 0?
    N
    turn := 1?
      N
      c1 := 1
      turn := 2?
        N
      turn := 1?
        N
    Y

  critical section 1
  turn := 2; c1 := 1; noncritical 1

Process 2:
  c2 := 0

  c1 := 0?
    N
    turn := 2?
      N
    turn := 1?
      N
    Y

  critical section 2
  turn := 1; c2 := 1; noncritical 2

Th.J. Dekker’s Solution
Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “consumes”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data
OK, let’s write some code for this (using locks only)

object array[N]
void enqueue(object x);
object dequeue();
Semaphore Motivation

• Problem with locks: mutual exclusion, but \textit{no ordering}
• Inefficient for producer-consumer (and lots of other things)
  • \textbf{Producer}: creates a resource
  • \textbf{Consumer}: uses a resource
  • \textbf{bounded buffer} between them
  • You need synchronization for correctness, \textit{and}...
• Scheduling order:
  • producer waits if buffer full, consumer waits if buffer empty
Semaphores

- Synchronization variable
  - Integer value
    - Can’t access value directly
    - Must initialize to some value
      - sem_init(sem_t *s, int pshared, unsigned int value)
  - Two operations
    - sem_wait, or down(), P()
    - sem_post, or up(), V()

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```
Semaphore Uses

• Mutual exclusion
  • Semaphore as mutex
  • What should initial value be?
    • Binary semaphore: X=1
    • (Counting semaphore: X>1)

• Scheduling order
  • One thread waits for another
  • What should initial value be?
    // thread 0
    ... // 1st half of computation
    sem_post(s);

    // thread 1
    sem_wait(s);
    ... // 2nd half of computation

// initialize to X
sem_init(s, 0, X)
sem_wait(s);
// critical section
sem_post(s);
Producer-Consumer with semaphores

- Two semaphores
  - `sem_t full; // # of filled slots`
  - `sem_t empty; // # of empty slots`

- Problem: mutual exclusion?

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);

producer() {
    sem_wait(empty);
    ... // fill a slot
    sem_post(full);
}

consumer() {
    sem_wait(full);
    ... // empty a slot
    sem_post(empty);
}
```
Producer-Consumer with semaphores

- Three semaphores
  - `sem_t full;` // # of filled slots
  - `sem_t empty;` // # of empty slots
  - `sem_t mutex;` // mutual exclusion

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```c
producer() {
    sem_wait(empty);
    sem_wait(&mutex);
    ... // fill a slot
    sem_post(&mutex);
    sem_post(full);
}
```

```c
consumer() {
    sem_wait(full);
    sem_wait(&mutex);
    ... // empty a slot
    sem_post(&mutex);
    sem_post(empty);
}
```
Pthreads and Semaphores

- No pthread_semaphore_t!
  - Type: pthread_semaphore_t

- int pthread_semaphore_init(pthread_spinlock_t *lock);
- int pthread_semaphore_destroy(pthread_spinlock_t *lock);
...

- ?? ????
What is a monitor?

- Monitor: one big lock for set of operations/methods
- Language-level implementation of mutex
  - Entry procedure: called from outside
  - Internal procedure: called within monitor
  - Wait within monitor releases lock

Many variants...
Pthreads and conditions/monitors

- **Type** `pthread_cond_t`

```c
int pthread_cond_init(pthread_cond_t *cond,
                       const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
                       pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Java: `synchronized` keyword
- `wait() / notify() / notifyAll()`

C#: `Monitor` class
- `Enter() / Exit() / Pulse() / PulseAll()`

Why the `pthread_mutex_t` parameter for `pthread_cond_wait`?
Hoare-style Monitors
(aka blocking condition variables)

Given entrance queue ‘e’, signal queue ‘s’, condition var ‘c’

**enter:**

if(locked):
    e.push_back(thread)
else
    lock

**wait C:**

C.q.push_back(thread)
schedule // block this thread

**signal C:**

if (C.q.any())
    t = C.q.pop_front() // t → "the signaled thread"
    s.push_back(t)
    t.run
    // block this thread

**schedule:**

if s.any()
    t ← s.pop_first()
    t.run
else if e.any()
    t ← e.pop_first()
    t.run
else
    unlock // monitor unoccupied

- Leave calls schedule
- Signaler must wait, but gets priority over threads on entrance queue
- How is this different from Mesa monitors?
- Is s queue necessary?
Mesa-style monitors
(aka non-blocking condition variables)

**enter:**

```cpp
if locked:
    e.push_back(thread)
    block
else
    lock
```

**schedule:**

```cpp
if e.any()
    t ← e.pop_front
    t. run
else
    unlock
```

**notify C:**

```cpp
if C.q.any()
    t ← C.q.pop_front() // t is "notified"
    e.push_back(t)
```

**wait C:**

```cpp
C.q.push_back(thread)
schedule
block
```

- (Leave calls schedule)
- Can be extended with extra queues for priority
- What are the differences?
Example: anyone see a bug?

Storage.Allocator: MONITOR = BEGIN
    availableStorage: INTEGER,
    moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
    RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
            DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
    END;

    <put back chunk of size words & update availableStorage>;
    NOTIFY moreAvailable END;

    pNew ← Allocate[size];
    <copy contents from old block to new block>;
    Free[pOld] END;

END.
Barriers
Prefix Sum

begin

\begin{array}{cccccc}
  a & b & c & d & e & f \\
\end{array}

end

\begin{array}{cccccc}
  a & a+b & a+b+c & a+b+c+d & a+b+c+d+e & a+b+c+d+e+f \\
\end{array}

\text{time}
Prefix Sum

begin

a

b

a+b

c

da

e

e+f

d

e+f

d

e+f

d

e+f

d

e+f

e+f

end

a

a+b

a+b+c

a+b+c+d

a+b+c+d+e

a+b+c+d+e+f

d

e+f

d

e+f

d

e+f

d

e+f

d

e+f

d

e+f

d

e+f

e+f

\text{time}
Parallel Prefix Sum

begin

a
b
a+b
b+c
b+c+d
a+b+c+d
b+c+d+e
b+c+d+e+f

end

a
a+b
a+b+c
a+b+c+d
a+b+c+d+e
a+b+c+d+e+f

Chapter 5  Synchronization Algorithms and Concurrent Programming  Gadi Taubenfeld © 2014
Pthreads Parallel Prefix Sum

```c
int g_values[N] = { a, b, c, d, e, f };  

void prefix_sum_thread(void * param) {
    int i;
    int id = *((int*)param);
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<=1) {
        g_values[id+stride] += g_values[id];
    }
}

Will this work?
Pthreads Parallel Prefix Sum

```c
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};
int g_values[N] = { a, b, c, d, e, f }

void prefix_sum_thread(void * param) {
    int i;
    int id = *((int*)param);
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<=1) {
        pthread_mutex_lock(&g_locks[id]);
        pthread_mutex_lock(&g_locks[id+stride]);
        g_values[id+stride] += g_values[id];
        pthread_mutex_unlock(&g_locks[id]);
        pthread_mutex_unlock(&g_locks[id+stride]);
    }
}
```
Parallel Prefix Sum

\begin{align*}
\text{begin} & \quad a & b & c & d & e & f \\
\text{barrier} & \quad a & a+b & b+c & c+d & d+e & e+f \\
\text{barrier} & \quad a & a+b & a+b+c & a+b+c+d & b+c+d+e & c+d+e+f \\
\text{end} & \quad a & a+b & a+b+c & a+b+c+d & a+b+c+d+e & a+b+c+d+e+f \\
\end{align*}
What is a Barrier?

- Coordination mechanism (algorithm)
- forces processes/threads to wait until each one of them has reached a certain point.
- Once all the processes/threads reach barrier, they all can pass the barrier.
Pthreads and barriers

- **Type** `pthread_barrier_t`

```c
int pthread_barrier_init(pthread_barrier_t *barrier,
                        const pthread_barrierattr_t *attr,
                        unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```
Pthreads Parallel Prefix Sum

```c
pthread_barrier_t g_barrier;
pthread_mutex_t g_locks[N];
int g_values[N] = { a, b, c, d, e, f };,

void init_stuff() {
    ...
    pthread_barrier_init(&g_barrier, NULL, N-1);
}

void prefix_sum_thread(void * param) {
    int i;
    int id = *((int*)param);
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<=1) {
        pthread_mutex_lock(&g_locks[id]);
        pthread_mutex_lock(&g_locks[id+stride]);
        g_values[id+stride] += g_values[id];
        pthread_mutex_unlock(&g_locks[id]);
        pthread_mutex_unlock(&g_locks[id+stride]);
    }
    pthread_barrier_wait(&g_barrier);
}
```
Barrier Goals

Ideal barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Low shared memory references per process
- No need for shared memory initialization
- Symmetric-ness (same amount of work for all processes)
- Algorithm simplicity
- Simple basic primitive
- Minimal propagation time
- Reusability of the barrier (must!)
Barrier Building Blocks

• Semaphores
• Atomic Bit
• Atomic Register
• Fetch-and-increment register
• Test and set bits
• Read-Modify-Write register
Barrier with Semaphores
Barrier using Semaphores
Algorithm for n processes

```
shared arrival: binary semaphore, initially 1
    departure: binary semaphore, initially 0
    counter: atomic register ranges over {0, ..., n}, initially 0

1  sem_wait(arrival)
2  counter := counter + 1 // atomic register
3  if counter < n then sem_post(arrival) else sem_post(departure)
4  sem_wait(departure)
5  counter := counter - 1
6  if counter > 0 then sem_post(departure) else sem_post(arrival)
```

*Question:* Would this barrier be correct if the shared counter won’t be an *atomic* register?
Barrier using Semaphores

Properties

• **Pros:**
  • Very Simple
  • Space complexity $O(1)$
  • Symmetric

• **Cons:**
  • Required a strong object
    • Requires some central manager
    • High contention on the semaphores
  • Propagation delay $O(n)$
Questions?