DTHREADS

Efficient Deterministic Multithreading

Tongping Li, Charlie Curtsinger, Emery D. Berger

Presented by Avidan Borisov
Intro
Background

- Everyone here can probably agree at this point that multithreading is difficult -
Background

- Everyone here can probably agree at this point that multithreading is difficult -
  - Deadlocks
Background

- Everyone here can probably agree at this point that **multithreading is difficult** -
  - Deadlocks
  - Race conditions
Background

- Everyone here can probably agree at this point that **multithreading is difficult** -
  - Deadlocks
  - Race conditions
  - Starvation
Background

- Everyone here can probably agree at this point that **multithreading is difficult** -
  - Deadlocks
  - Race conditions
  - Starvation
- But what's even more annoying is that thread executions are **non-deterministic**.
Non-Determinism

- What does the following program output?
Non-Determinism

- What does the following program output?

```c
int a = 0, b = 0;

int main()
{
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1()
{
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2()
{
  if (a == 0) {
    b = 1;
  }
}
```
Non-Determinism

• What does the following program output?

```c
int a = 0, b = 0;
int main()
{
    t1 = spawn(thread1);
    t2 = spawn(thread2);
    join(t1, t2);
    print(a, b);
}

void thread1()
{
    if (b == 0) {
        a = 1;
    }
}

void thread2()
{
    if (a == 0) {
        b = 1;
    }
}
```

What does the following program output?
What does the following program output?

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1() {
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```

Non-Determinism
Non-Determinism

• What does the following program output?

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}

void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
Non-Determinism

- What does the following program output?

```c
int a = 0, b = 0;
int main() {
    t1 = spawn(thread1);
    t2 = spawn(thread2);
    join(t1, t2);
    print(a, b);
}

void thread1() {
    if (b == 0) {
        a = 1;
    }
}

void thread2() {
    if (a == 0) {
        b = 1;
    }
}
```

What does the following program output?
Non-Determinism

• What does the following program output?

```c
int a = 0, b = 0;
int main() {
t1 = spawn(thread1);
t2 = spawn(thread2);
join(t1, t2);
print(a, b);
}
```

```c
void thread1() {
if (b == 0) {
a = 1;
}
}
```

```c
void thread2() {
if (a == 0) {
b = 1;
}
}
```

What does the following program output?
Non-Determinism

• What does the following program output?
  • (1,0)
Non-Determinism

- What does the following program output?
- (1,0)

```c
int main() {
  int a = 0, b = 0;
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1() {
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
Non-Determinism

- What does the following program output?
  - (1,0)
Non-Determinism

- What does the following program output?
- \((1,0)\)
Non-Determinism

- What does the following program output?
  - (1,0)
Non-Determinism

- What does the following program output?
- (1,0)
Non-Determinism

• What does the following program output?
  • (1,0)
Non-Determinism

- What does the following program output?
  - (1,0)
Non-Determinism

- What does the following program output?
- (1,0)

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1() {
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```

What does the following program output?
- (1,0)
Non-Determinism

• What does the following program output?
  • (1,0)  • (0,1)

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}

void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
Non-Determinism

• What does the following program output?
  • (1,0)  • (0, 1)  • (1, 1)

```
int a = 0, b = 0;

int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}

void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
  }
}

void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
Deterministic Threads

- Previous example is a classic case of a **data race**.
- Not using thread synchronization primitives causes a non-deterministic result.
Deterministic Threads

• Previous example is a classic case of a data race.
• Not using thread synchronization primitives causes a non deterministic result.
• What if we could guarantee that each run would nonetheless behave exactly the same way?
Why?

- Eliminate "Heisenbugs"
Why?

- Eliminate "Heisenbugs"
  - Even in the face of race conditions!
Why?

- Eliminate "Heisenbugs"
  - Even in the face of race conditions!
- Simplify debugging
Why?

• Eliminate "Heisenbugs"
  ▪ Even in the face of race conditions!
• Simplify debugging
• Reduce testing overhead
Why?

• Eliminate "Heisenbugs"
  ■ Even in the face of race conditions!
• Simplify debugging
• Reduce testing overhead
• Enable "record and replay" without memory operations tracking
Why?

• Eliminate "Heisenbugs"
  ▪ Even in the face of race conditions!
• Simplify debugging
• Reduce testing overhead
• Enable "record and replay" without memory operations tracking
  ▪ and more...
Introducing: DTHREADS
DTHREADS - Overview

- Fully deterministic multi-threaded system
DTHREADS - Overview

- Fully deterministic multi-threaded system
  - Now the output will always be (1,1). More on why later.

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1() {
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with pthreads.
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with `pthreads`.
  - If this is how you compile with `pthreads`.

```
g++ -lpthread main.cpp
```
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with pthreads.
  - If this is how you compile with pthreads.
  - This is how you compile with dthreads.

```
g++ -ldthread main.cpp
```
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with pthreads.
- Robust to changes
DTHREADS - Overview

- Robust to changes

- As said, now the output is always (1,1)

```c
int a = 0, b = 0;
int main() {
  t1 = spawn(thread1);
  t2 = spawn(thread2);
  join(t1, t2);
  print(a, b);
}
```

```c
void thread1() {
  if (b == 0) {
    a = 1;
  }
}
```

```c
void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```
DTHREADS - Overview

- Robust to changes
  - As said, now the output is always (1,1)
  - Add some prints? The output is still always (1,1)

```c
int a = 0, b = 0;
int main() {
    t1 = spawn(thread1);
    t2 = spawn(thread2);
    join(t1, t2);
    print(a, b);
}
```

```c
void thread1() {
    if (b == 0) {
        LOG("seminar!");
        a = 1;
    }
}
```

```c
void thread2() {
    if (a == 0) {
        b = 1;
        LOG("b=1!");
    }
}
```
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with **pthreads**.
- Robust to changes
DTHREADS - Overview

- Fully deterministic multi-threaded system
- Completely compatible with pthreads.
- Robust to changes
- Efficient
How?
Synchronization Points

- Any API function used to synchronize threads - mutex_lock, mutex_unlock, cond_wait, ...
Synchronization Points

- Any **API** function used to synchronize threads - *mutex_lock, mutex_unlock, cond_wait, ...*
Synchronization Points

- Any **API** function used to synchronize threads - `mutex_lock`, `mutex_unlock`, `cond_wait`, ...

```
A

pthread_create

B
```
Synchronization Points

- Any **API** function used to synchronize threads -
  `mutex_lock`, `mutex_unlock`, `cond_wait`, ...

Diagram:
- A: `pthread_create`
- B: `mutex_lock`
Synchronization Points

- Any **API** function used to synchronize threads -
  `mutex_lock`, `mutex_unlock`, `cond_wait`, ...

---

```
A

 pthread_create

 mutex_lock

 cond_wait

 B
```
Synchronization Points

- Any **API** function used to synchronize threads - *mutex_lock, mutex_unlock, cond_wait, ...*

---

**Diagram:**

- **A**
  - pthread_create
  - cond_wait

- **B**
  - mutex_lock
  - mutex_unlock
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.

```
A
pthread_create

B
mutex_lock
cond_wait
mutex_unlock
```
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.

```
var1=5
mutex_lock
mutex_unlock
```
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.

![Diagram showing synchronization points with pthread_create, var1 == 5, cond_wait, var1=5, mutex_lock, mutex_unlock]
Synchronization Points

- **Before sync points**, a threads modifications aren't guaranteed to be visible to other threads.
Synchronization Points

- **Before sync points**, a threads modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

• **Before sync points**, a threads modifications aren't guaranteed to be visible to other threads.

• Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.

```
A  no writes to shared mem!

pthread_create

B
```
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory snapshot before and after sync point, and only write diff before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a threads modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
**Synchronization Points**

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.

![Diagram showing synchronization points with pthread_create, mutex_lock, and no writes to shared mem!](image)
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory snapshot before and after sync point, and only write diff before the next sync point. Still valid.

```plaintext
pthread_create
mutex_lock
cond_wait
write diff to shared mem
```
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

- **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
- Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
Synchronization Points

• **Before sync points**, a thread's modifications aren't guaranteed to be visible to other threads.
• Idea - take memory **snapshot** before after sync point, and only **write diff** before the next sync point. Still valid.
• How does that help us?
Parallel & Serial Phases

- Let's further expand on our previous idea.
Parallel & Serial Phases

- Let's further expand on our previous idea.
- Let's say we have three threads running in parallel.

A ➔
B ➔
C ➔
Parallel & Serial Phases

- Let's further expand on our previous idea.
- Let's say we have three threads running in parallel.
- Take snapshot at start, don't write to shared mem during run.
Parallel & Serial Phases

- Let's say we have three threads running in parallel.
- When some thread reaches a **sync point**, **wait for the others** to reach some sync point too.

A → B → C

mutex_lock
Parallel & Serial Phases

- Let's say we have three threads running in parallel.
- When some thread reaches a **sync point**, wait for the **others** to reach some sync point too.

```
mutex_lock
cond_wait
```
Parallel & Serial Phases

- When all threads reach a sync point, we end the parallel phase.
Parallel & Serial Phases

- When all threads reach a sync point, we end the **parallel** phase.
- And start the **serial phase**

---

A  
```
mutex_lock
```

B  
```
mutex_lock
```

C  
```
cond_wait
```

---

leftarrow Serial Phase rightarrow
Parallel & Serial Phases

- In the serial phase, the **order of operation** between threads is always **the same**.
Parallel & Serial Phases

- In its turn, each thread will -
Parallel & Serial Phases

- In its turn, each thread will -
  - Commit its changes to the shared memory
Parallel & Serial Phases

- In its turn, each thread will -
  - Commit its changes to the shared memory
  - Perform the actual synchronization operation
Parallel & Serial Phases

- Commit its changes to the shared memory
- Perform the actual synchronization operation
- Take a new snapshot
Parallel & Serial Phases

- Perform the actual synchronization operation
- Take a new snapshot
- Trigger the next thread serial phase
Parallel & Serial Phases

- When all threads are done with their serial phases, a new **parallel phase** starts.

A

- **mutex_lock**

B

- **cond_wait**

C

- **barrier_wait**
Parallel & Serial Phases

This is the main idea behind DTHREADS.
Parallel & Serial Phases

This is the main idea behind DTHREADS.
Still many questions to be answered -
Parallel & Serial Phases

This is the main idea behind DTHREADS.
Still many questions to be answered -

1. How does **snapshotting** work?
Parallel & Serial Phases

This is the main idea behind DTHREADS.

Still many questions to be answered -

1. How does **snapshotting** work?
2. How does the transition between **parallel** and **serial** phases happen?
Parallel & Serial Phases

This is the main idea behind DTHREADS. Still many questions to be answered -

1. How does snapshotting work?
2. How does the transition between parallel and serial phases happen?
3. How is the order always preserved?
Parallel & Serial Phases

This is the main idea behind DTHREADS.
Still many questions to be answered -

1. How does snapshotting work?
2. How does the transition between parallel and serial phases happen?
3. How is the order always preserved?
4. How does each kind of sync point do its job?
Memory Snapshotting
Isolated Threads

- We want to implement a mechanism which saves each thread's memory after a sync point, and writes the changed values before the next sync point starts.
Isolated Threads

- We want to implement a mechanism which saves each thread's memory after a sync point, and writes the changed values before the next sync point starts.
  - We'll call this period a transaction.

![Diagram of isolated threads with arrows indicating sequence and direction.](image-url)
Isolated Threads

• We want to implement a mechanism which saves each thread's memory after a sync point, and writes the changed values before the next sync point starts.
  ■ We'll call this period a transaction.

• During a transaction, we can't write to the shared memory. Changes should only be visible to the local thread.
Isolated Threads

• We want to implement a mechanism which saves each thread's memory after a sync point, and writes the changed values before the next sync point starts.
  ▪ We'll call this period a transaction.

• During a transaction, we can't write to the shared memory. Changes should only be visible to the local thread.

• Ideas?
Isolated Threads? Processes!

- DTHREADS uses **processes** under-the-hood instead of **threads**.

- By using processes, memory accesses in the parallel phase work on **private copies** of the memory, and updates are **not shared**.
Isolated Threads? Processes!

- When a new thread is created, we use `clone(CLONE_FILES)` to duplicate the memory but keep the file descriptors shared across process, as with threads.
- We make `getpid()` always return the same global value.
Making Transactions

• So how do we create a transaction (take 🌟 and then 📦)?
Making Transactions

• So how do we create a transaction (take and then )?

• Lets say we have two threads (processes) that need to update some shared state.
Making Transactions

• Initially, each of these processes have this shared page mapped in their address space
Making Transactions

- Initially, each of these processes have this shared page mapped in their address space.
- When the **parallel phase** starts, we **write protect** this page, making it **read only**.
Making Transactions

- When the **parallel phase** starts, we **write protect** this page, making it **read only**.

- During the parallel phase up-to the first write, all **reads** go directly to the shared memory.
Making Transactions

- Upon the first write, a **page fault** is caught and the write goes to a copy of the page (**COW**).
Making Transactions

- Upon the first write, a **page fault** is caught and the write goes to a copy of the page (**COW**).

- Subsequent reads and writes will go to the copied page until the end of the parallel phase.
Making Transactions

- When the serial phase starts, we first copy the shared page to a twin page.
Making Transactions

- When the serial phase starts, we first **copy the shared page** to a **twin page**.
- Then each thread, in order, commits the changes by -
Making Transactions

- Then each thread, in order, commits the changes by -
  - Comparing copied page with the twin page
Making Transactions

- Then each thread, in order, commits the changes by:
  - Comparing copied page with the twin page
  - Writing the **diff** to the shared page
Making Transactions

- Then each thread, in order, commits the changes by -
  - Comparing copied page with the twin page
  - Writing the **diff** to the shared page
Making Transactions

- Then each thread, in order, commits the changes by:
  - Comparing copied page with the twin page
  - Writing the **diff** to the shared page
Making Transactions

- When the serial phase ends and we start a **new parallel phase**, we delete all copied pages and again create **read only** mappings to the shared memory.
How does it look in code?

- Let's take a look at how transactions are implemented in code.
How does it look in code?

- Let's take a look at how transactions are implemented in code.
- There are 3 main functions to implement...
How does it look in code?

- Lets take a look at how transactions are implemented in code.

- There are 3 main functions to implement
  - `atomicBegin` - starts a transaction
How does it look in code?

- Lets take a look at how transactions are implemented in code.

- There are 3 main functions to implement
  - `atomicBegin` - starts a transaction
  - `handlePageFault` - handle first write within transaction
How does it look in code?

• Lets take a look at how transactions are implemented in code.

• There are 3 main functions to implement
  ▪ `atomicBegin` - starts a transaction
  ▪ `handlePageFault` - handle first write within transaction
  ▪ `atomicEnd` - end the transaction
void atomicBegin() {
    foreach (page in modifiedPages) {
        page.writeProtect();
        page.privateCopy.free();
    }
    modifiedPages.emptyList();
}
atomicBegin

- First, all previously-written pages are write-protected
atomicBegin

- First, all previously-written pages are write-protected
- Then each previous private copy is freed and mappings are updated to point to the shared page.

```java
void atomicBegin() {
    foreach (page in modifiedPages) {
        page.writeProtect();
        page.privateCopy.free();
    }
    modifiedPages.emptyList();
}
```
atomicBegin

- First, all previously-written pages are write-protected.
- Then each previous private copy is freed and mappings are updated to point to the shared page.
- Finally, we clear the modified pages list.
```c
handlePageFault(
    signal(SIGSEGV, handlePageFault);

void handlePageFault(int signum, siginfo_t *siginfo) {
    void *addr = siginfo->si_addr; // address of access
    if (siginfo->si_code == SEGV_ACCERR && // write violation
        isMappedToPage(addr)) {
        handleWrite(addr);
    }
    abort();
}

void handleWrite(void* addr) {
    int pageNo = computePage(addr);
    void* pageStart = addrToPage(addr);
    PageInfo& page = getPageInfo(pageNo);
    page.copyAndDisableWriteProtection();
    atomicIncrement(&page.users);
    modifiedPages.insert(page);
}
```
Page faults are caught by handling segmentation faults.
handlePageFault

- If thrown because of memory write and its in range, handle copy-on-write.

```c
signal(SIGSEGV, handlePageFault);

void handlePageFault(int signum, siginfo_t *siginfo) {
    void *addr = siginfo->si_addr; // address of access
    if (siginfo->si_code == SEGV_ACCERR && // write violation
        isMappedToPage(addr)) {
        handleWrite(addr);
    }
    abort();
}

void handleWrite(void* addr) {
    int pageNo = computePage(addr);
    void* pageStart = addrToPage(addr);
    PageInfo& page = getPageInfo(pageNo);
    page.copyAndDisableWriteProtection();
    atomicIncrement(&page.users);
    modifiedPages.insert(page);
}
```
handlePageFault

- Calculate the page which belongs to the address

```c
void handlePageFault(int signum, siginfo_t *siginfo) {
    void *addr = siginfo->si_addr; // address of access
    if (siginfo->si_code == SEGV_ACCERR && // write violation
        isMappedToPage(addr)) {
        handleWrite(addr);
    }
    abort();
}

void handleWrite(void* addr) {
    int pageNo = computePage(addr);
    void* pageStart = addrToPage(addr);
    PageInfo& page = getPageInfo(pageNo);
    page.copyAndDisableWriteProtection();
    atomicIncrement(&page.users);
    modifiedPages.insert(page);
}
```
handlePageFault

- Map page region to private page copy and disable write protection.
handlePageFault

- Increase page writers count, and insert this page info to the modified pages list.
atomicEnd

atomicEnd commits all changes from transaction to shared memory

```java
void atomicEnd() {
    foreach (page in modifiedPages) {
        if (page.writers > 1 && !page.hasTwin()) {
            page.createTwin();
        }

        if (page.version == page.localCopy.version) {
            page.copyCommit();
        } else {
            page.diffCommit();
        }

        page.writers--;
        if (page.writers == 0 && page.hasTwin()) {
            page.twin.free();
        }

        page.version++;
    }
}
```
atomicEnd

atomicEnd commits all changes from transaction to shared memory

- For each modified page with more than one writer, we first ensure that a twin page is created.
atomicEnd commits all changes from transaction to shared memory

- If the page versions of the shared and private copy match then current thread must be first to commit, so we can just copy the entire page.
atomicEnd commits all changes from transaction to shared memory

- Otherwise, another thread already committed changes so we must use a diff commit.
atomicEnd

atomicEnd commits all changes from transaction to shared memory

- If no writers are left, free the twin page.
atomicEnd

atomicEnd commits all changes from transaction to shared memory

• After changes are committed, number of current page writers is decremented.
atomicEnd

atomicEnd commits all changes from transaction to shared memory

• Finally, increment the shared page's version.
Parallel-Serial Phase Shifts
What Do We Want?

- First we need a way to implement a synchronization mechanism which allows all threads to wait until all other threads reached a sync point.
What Do We Want?

- First we need a way to implement a synchronization mechanism which allows all threads to wait until all other threads reached a sync point.
- DTHREADS uses a variation of the common synchronization mechanism called \textit{barrier/fence}.

"Internal Fence"
(boundary between parallel and serial phases)
A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
**waitFence** is the function that a thread calls to wait for the internal fence to complete.
- **waitFence** is the function that a thread calls to wait for the internal fence to complete.

- Unlike a regular fence, it's important to note that in our fence, the **number of threads** waiting on the fence **can change** throughout execution.
Since threads are actually processes, it's important to note that the locks here are actually **cross-process** locks.
• At first, threads wait in the fence until the **arrival phase starts** (when all threads from previous fence have departed).

```c
void waitFence(void) {
    lock();
    while (!isArrivalPhase()) {
        CondWait();
    }

    waiting_threads++;
    if (waiting_threads < live_threads) {
        while (!isDeparturePhase()) {
            CondWait();
        }
    } else {
        setDeparturePhase();
        CondBroadcast();
    }

    waiting_threads--;
    if (waiting_threads == 0) {
        setArrivalPhase();
        CondBroadcast();
    }
    unlock();
}
```
• At first, threads wait in the fence until the **arrival phase starts** (when all threads from **previous fence have departed**).

• When a thread enters the **arrival phase**, it increases the number of waiting threads and waits for the **departure phase** to start.
• If the thread is the **last to enter** the fence, it sets the departure flag and wakes the waiting threads.
• If the thread is the **last to enter** the fence, it sets the departure flag and wakes the waiting threads.

• As threads leave the fence, they **decrease the waiting threads count**.
• Last thread that leaves sets the **arrival flag** and wakes all threads that enter the next fence cycle.

```c
void waitFence(void) {
    lock();
    while (!isArrivalPhase()) {
        CondWait();
    }
    waiting_threads++;
    if (waiting_threads < live_threads) {
        while (!isDeparturePhase()) {
            CondWait();
        }
    } else {
        setDeparturePhase();
        CondBroadcast();
    }
    waiting_threads--;
    if (waiting_threads == 0) {
        setArrivalPhase();
        CondBroadcast();
    }
    unlock();
}
```
Deterministic Serial Phase

When all threads leave the internal fence, it's time to execute the serial phase one-by-one in deterministic order.
Deterministic Serial Phase

When all threads leave the internal fence, it's time to execute the serial phase one-by-one in deterministic order.

This mechanism is achieved by using token passing.
Deterministic Serial Phase

We have a **global token** which whoever holds is the **only thread** that can execute in the serial phase.
Deterministic Serial Phase

While the thread with the token executes, other threads are waiting for their turn to get the token.

Parallel Phase → Serial Phase:

A → barrier_wait → Parallel Phase

B → mutex_lock → wait for token → Serial Phase

C → cond_wait → wait for token → Serial Phase
Deterministic Serial Phase

When the thread finishes, it passes the token to the next thread in the circular token queue.
Deterministic Serial Phase

When the thread finishes, it passes the token to the next thread in the circular token queue.
Deterministic Serial Phase

Threads can come and go from the queue, but the **order** between them is always **kept the same**.

A

B

C

mutex_lock

cond_wait

barrier_wait

A passes to

B passes to

C
Deterministic Serial Phase

The order is based on a deterministic thread index set on when each thread is created.

\[A\]oblin \hspace{1cm} \text{mutex_lock} \hspace{1cm} [B\]arrier_wait \hspace{1cm} [C]\]ond_wait

\[A\] passes to \hspace{1cm} \text{passes to} \hspace{1cm} [C]\]
Implementation

A global shared `token_holder` variable holds the thread id which currently has the token.

A → `mutex_lock` → barrier_wait → B

B → `token_holder` →

C → `cond_wait` →
Implementation

Two functions for token management -

- **waitToken** - block until current thread gets the token
- **putToken** - pass the token to the next waiting thread
Implementation

waitToken is called at the parallel phase (by the sync point function), and ends in the serial phase, with current thread holding the token.

```c
void waitToken() {
    waitFence();
    while (token_holder != current_thread()) {
        yield();
    }
}
```
Implementation

- It first waits at the **internal fence**, to shift from the parallel to serial phase.

```c
void waitToken() {
    waitFence();
    while (token_holder != current_thread()) {
        yield();
    }
}
```
Implementation

- Then it yields the CPU until the `token_holder` is the current thread.

```c
void waitToken() {
  waitFence();
  while (token_holder != current_thread()) {
    yield();
  }
}
```
Implementation

When current thread is done, it calls `putToken` which takes next thread from the token queue, and overwrites `token_holder`.

```java
1  void putToken() {
2    token_holder = token_queue.nextThread();
3 }
```

A → `mutex_lock` → barrier_wait
B → `mutex_lock` → `cond_wait`
C → `mutex_lock` → `cond_wait`
Putting It All Together
pthreads API

pthread_attr_destroy
pthread_attr_getdetachstate
pthread_attr_getguardsize
pthread_attr_getinheritsched
pthread_attr_getnamework_np
pthread_attr_getscope
pthread_attr_getsysname_np
pthread_attr_init
pthread_attr_setdetachstate
pthread_attr_setguardsize
pthread_attr_setinheritsched
pthread_attr_setnamework_np
pthread_attr_setscope

pthread_cancel
pthread_cleanup_peek_np
pthread_cleanup_pop
pthread_cleanup_push
pthread_clear_exit_np
pthread_cond_broadcast
pthread_cond_destroy
pthread_cond_getname_np
pthread_cond_init
pthread_cond_setname_np
pthread_cond_signal
pthread_cond_timedwait
pthread_cond_wait

pthread_mutex_destroy
pthread_mutex_getname_np
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_setname_np
pthread_mutex_setprioceiling
pthread_mutex_timedlock_np
pthread_mutex_unlock

pthread_rwlock_destroy
pthread_rwlock_getname_np
pthread_rwlock_init
pthread_rwlock_rdlock
pthread_rwlock_setname_np
pthread_rwlock_timedrdlock_np
pthread_rwlock_timedwrlock_np
pthread_rwlock_tryrdlock
pthread_rwlock_trywrlock
pthread_rwlock_unlock

pthread_self
pthread_set_mutexattr_default_np
pthread_setcancelstate
pthread_setcanceltype
pthread_setconcurrency
pthread_setnamework_np
pthread_setnameworkoption_np
pthread_setschedparam
pthread_setspecific
pthread_signal_to_cancel_np
pthread_test_exif_np
pthread_testcancel
pthread_trace_init_np

sched_yield
pthreads API

- **pthreads does a lot**
pthreads API

- pthreads does a lot
- each function is a sync point which 
dthreads re-implements
• **pthreads does a lot**
  • each function is a sync point which
dthreads re-implements
• we'll take a look at how **locks** and
**condition variables** are implemented
Most of DTHREADS implementations are variations on the following template:

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Generic Template

Most of DTHREADS implementations are variations on the following template:

- Thread reaches sync point

```
void sync_point() {
    waitToken();
    atomicEnd(); // do actual sync operation
    putToken();
    atomicBegin();
}
```
Most of DTHREADS implementations are variations on the following template:

- It waits for the token

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Most of DTHREADS implementations are variations on the following template

- The serial phase starts

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Most of DTHREADS implementations are variations on the following template

- Changes are committed

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Generic Template

Most of DTHREADS implementations are variations on the following template

• Some synchronization is performed, might call the real pthreads function

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Most of DTHREADS implementations are variations on the following template

- Token is passed to next thread
Generic Template

Most of DTHREADS implementations are variations on the following template

- New transaction begins

```c
void sync_point() {
    waitToken();
    atomicEnd();
    // do actual sync operation
    putToken();
    atomicBegin();
}
```
Locks

- Under DTHREADS, all locks are turned into one global lock.
  - The *global token* serves as the lock.

- Has potential to hurt performance (we'll see it doesn't most of the time), but necessary for *deterministic execution*. 
Locks

Let's have a look at the lock operation.

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
    }
    lock_count++;
}
```
Locks

Let's have a look at the lock operation.

Note that it effectively ignores `<m>` and only depends on previously described mechanisms - doesn't differentiate different locks.
Locks

Different locks can be locked simultaneously, but we've effectively merged all locks into one lock.

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
    }
    lock_count++;
}
```
Different locks can be locked simultaneously, but we've effectively merged all locks into one lock.

That's why we keep a thread-local counter of the number of currently owned locks.
Locks

Different locks can be locked simultaneously, but we've effectively merged all locks into one lock.

That's why we keep a thread-local counter of the number of currently owned locks.

And only actually lock when the thread isn't holding yet any locks.
Locks

Different locks can be **locked simultaneously**, but we've effectively merged all locks into **one lock**.

That's why we keep a **thread-local counter** of the number of currently owned locks.

And only actually lock when the thread isn't holding yet any locks.

No lock guarding lock_count? 😐
Locks

Different locks can be **locked simultaneously**, but we've effectively merged all locks into **one lock**.

That's why we keep a **thread-local counter** of the number of currently owned locks.

And only actually lock when the thread isn't holding yet any locks.

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
    }
    lock_count++;
}
```

No lock guarding lock_count? 😟
No problem, it's thread local
Locks

If the thread isn't holding any locks we -

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
    }
    lock_count++;
}
```
Locks

If the thread isn't holding any locks we -

- Wait for the token

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
        lock_count++;
    }
}
```
Locks

If the thread isn't holding any locks we -

- Wait for the token
- End current transaction (commit)
If the thread isn't holding any locks we -

- Wait for the token
- End current transaction (commit)
- Start a new transaction

```c
void mutex_lock(mutex_t* m) {
    if (lock_count == 0) {
        waitToken();
        atomicEnd();
        atomicBegin();
    }
    lock_count++;
}
```
Locks

If the thread isn't holding any locks we -

- Wait for the token
- End current transaction (commit)
- Start a new transaction

Where's putToken?
Locks

If the thread isn't holding any locks we -

- Wait for the token
- End current transaction (commit)
- Start a new transaction

Where's `putToken`?

- We intentionally **don't pass** the token **until we unlock** the lock!
Similarly, the **unlock** operation will -

```c
void mutex_unlock(mutex_t* m) {
    lock_count--; 
    if (lock_count == 0) {
        atomicEnd(); 
        putToken(); 
        atomicBegin(); 
        waitFence(); 
    } 
}
```
Locks

Similarly, the **unlock** operation will:

- Decrease the thread lock count

```c
void mutex_unlock(mutex_t* m)
{
    lock_count--;
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Locks

Similarly, the **unlock** operation will -

- Decrease the thread lock count
- If thread has no more locks held

```c
void mutex_unlock(mutex_t * m) {
    lock_count--;
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Locks

Similarly, the **unlock** operation will -

- Decrease the thread lock count
- If thread has no more locks held
  - End transaction

```c
void mutex_unlock(mutex_t* m) {
    lock_count--;  
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Locks

Similarly, the **unlock** operation will -

- Decrease the thread lock count
- If thread has no more locks held
  - End transaction
  - **Release the token**

```c
void mutex_unlock(mutex_t* m) {
    lock_count--;
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Locks

Similarly, the **unlock** operation will -

- Decrease the thread lock count
- If thread has no more locks held
  - End transaction
  - **Release the token**
  - Start new transaction

```c
void mutex_unlock(mutex_t* m) {
    lock_count--;
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Locks

Similarly, the **unlock** operation will -

- Decrease the thread lock count
- If thread has no more locks held
  - End transaction
  - **Release the token**
  - Start new transaction
  - Wait on the internal fence until parallel phase starts.

```c
void mutex_unlock(mutex_t* m) {
    lock_count--;
    if (lock_count == 0) {
        atomicEnd();
        putToken();
        atomicBegin();
        waitFence();
    }
}
```
Condition Variables

- Determinism for condition variables is tricky, since the OS has **no guarantees on the order threads wake up** on the condition.

- DTHREADS guarantees threads **wake up in the order they waited** (first to wait = first to wake).
Condition Variables

- To enforce this order, each condition variable holds a queue of waiting threads, each with an is_ready bit.
- `cond_wait` appends current thread to queue
- `cond_signal` pops next thread from queue, and sets its ready bit
Condition Variables

Let's look at `cond_wait`.

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--; // Decrease the number of live threads.
    cond->queue.addThread(current_thread());
    putToken();

    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }

    while (token_holder != current_thread()) {
        yield();
    }

    atomicBegin();
}
```
Condition Variables

Let's look at `cond_wait`

- Wait for token and commit changes

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();
    while (!cond->queue.isReady(current_thread())) { 
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) { 
        yield();
    }
    atomicBegin();
}
```
Condition Variables

Lets look at cond_wait

- Wait for token and commit changes
- We want the current thread **out of the serial phase**, until it wakes up on the condition.
Condition Variables

Lets look at cond_wait

- Wait for token and commit changes
- **Remove** current thread from **token queue**
Condition Variables

Let's look at `cond_wait`

- Wait for token and commit changes
- **Remove** current thread from token queue

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();
    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
Condition Variables

Let's look at cond_wait

• Wait for token and commit changes

• **Remove** current thread from **token queue**

• **Decrease** number of threads for the fence

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();

    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }

    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
Condition Variables

- Add current thread to the waiting threads queue

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();
    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
Condition Variables

- Add current thread to the waiting threads queue
- Pass the token

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();

    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
The thread is then suspended on the actual pthread condition variable, until it's woken up and marked ready in the queue.

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();
    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
Condition Variables

- The thread then waits back for the token. Why?

```c
void cond_wait(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    token_queue.removeThread(current_thread());
    live_threads--;
    cond->queue.addThread(current_thread());
    putToken();
    while (!cond->queue.isReady(current_thread())) {
        pthread_cond_wait(cond, m);
    }
    while (token_holder != current_thread()) {
        yield();
    }
    atomicBegin();
}
```
The thread then waits back for the token. Why?

- *cond_wait* can be called within a locked mutex.
Condition Variables

- The thread then waits back for the token. Why?
  - `cond_wait` can be called within a locked mutex.

- Finally, it starts a new transaction
Now let's look at **cond_signal**

```c
void cond_signal(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();

    if (cond->queue.length != 0) {
        lock();
        thread = cond->queue.removeNext();
        token_queue.insert(thread);
        live_threads++;
        thread.setReady(true);
        pthread_cond_signal(cond, m);
        unlock();
    }

    atomicBegin();
}
```
Now let's look at `cond_signal`

- Wait for token and commit changes
Condition Variables

Now lets look at `cond_signal`

- Wait for token and commit changes
- Nothing to do if the waiting queue is empty

```c
void cond_signal(cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    if (cond->queue.length != 0) {
        lock();
        thread = cond->queue.removeNext();
        token_queue.insert(thread);
        live_threads++;
        thread.setReady(true);
        pthread_cond_signal(cond, m);
        unlock();
    }
    atomicBegin();
}
```
Now let's look at `cond_signal`

- Wait for token and commit changes
- Nothing to do if the waiting queue is empty
- Cross-process locking
**Condition Variables**

- Pop next waiting thread from the queue
Condition Variables

- **Pop next** waiting thread from the queue
- **Insert it** back to the token queue

```c
void cond_signal(cond_t* cond, mutex_t* m) {
  waitToken();
  atomicEnd();
  if (cond->queue.length != 0) {
    lock();
    thread = cond->queue.removeNext();
    token_queue.insert(thread);
    live_threads++;
    thread.setReady(true);
    pthread_cond_signal(cond, m);
    unlock();
  }
  atomicBegin();
}
```
Condition Variables

- **Pop next** waiting thread from the queue
- **Insert it** back to the token queue
- Update number of live threads for the fence.

```c
void cond_signal (cond_t* cond, mutex_t* m) {
    waitToken();
    atomicEnd();
    if (cond->queue.length != 0) {
        lock();
        thread = cond->queue.removeNext();
        token_queue.insert(thread);
        live_threads++;
        thread.setReady(true);
        pthread_cond_signal(cond, m);
        unlock();
    }
    atomicBegin();
}
```
Mark next thread as ready.
Condition Variables

- Mark next thread as ready.
- Use pthreads original cond_signal to actually wake up the new thread.
Condition Variables

- Mark next thread as ready.
- Use pthreads original cond_signal to actually wake up the new thread
- Start new transaction

```c
void cond_signal (cond_t * cond, mutex_t * m) {
    waitToken();
    atomicEnd();
    if (cond->queue.length != 0) {
        lock();
        thread = cond->queue.removeNext();
        token_queue.insert(thread);
        live_threads++;
        thread.setReady(true);
        pthread_cond_signal(cond, m);
        unlock();
    }
    atomicBegin();
    atomicBegin();
}
```
Evaluation
Concerns

- Looks like we have a price to pay for our deterministic execution
Concerns

- Looks like we have a **price to pay** for our deterministic execution
  - processes vs threads
Concerns

• Looks like we have a **price to pay** for our deterministic execution
  - processes vs threads
  - page fault handling
Concerns

- Looks like we have a **price to pay** for our deterministic execution
  - processes vs threads
  - page fault handling
  - copying and diffing memory
Concerns

• Looks like we have a **price to pay** for our deterministic execution
  - processes vs threads
  - page fault handling
  - copying and diffing memory
  - stopping at synchronization points
Concerns

- Looks like we have a **price to pay** for our deterministic execution
  - processes vs threads
  - page fault handling
  - copying and diffing memory
  - stopping at synchronization points
- How can DTHREADS **match and even exceed** the performance of pthreads?
Concerns

- Most of the costs are **amortized** -
  - Process creation costlier than threads but happens once.
  - Page fault is only handled on the first write.
  - We diff/copy memory only at synchronization points.
Concerns

- Most of the costs are amortized -
  - Process creation costlier than threads but happens once.
  - Page fault is only handled on the first write.
  - We diff/copy memory only at synchronization points.

- Several optimizations try to reduce the amount of work -
  - Lazy commits, diffs and twin page creation
  - Exclusive/First page writers
The False Sharing Problem

- How can DTHREAD exceed the performance of pthreads?
- Threads often suffer from false-sharing, the effect of threads on different cores often invalidating each other's local caches, since they work on shared memory.
- Since in DTHREADS threads are processes, they operate on private address spaces, therefore they don't suffer from this issue, and might have better performance.
Figure 15. Normalized execution time with respect to pthreads (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than pthreads, while providing deterministic behavior.
Evaluation

Figure 15. Normalized execution time with respect to pthreads (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than pthreads, while providing deterministic behavior.
Evaluation

Figure 15. Normalized execution time with respect to pthreads (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than pthreads, while providing deterministic behavior.
Disclaimers and Summary
Disclaimers
Disclaimers

- DTHREADS doesn't support programs with ad-hoc synchronization that don't depend on pthreads, such as atomic instructions.
Disclaimers

- DTHREADS doesn't support programs with ad-hoc synchronization that don't depend on pthreads, such as atomic instructions.

- DTHREADS only guarantees the same deterministic run out of the set of possible runs assuming they don't change - there are no guarantees if there's external non-determinism.
Disclaimers

- DTHREADS usually **matches or exceeds performance** of pthreads. But **extensive use of locking** might degrade performance, as it might effectively serialize execution.
Disclaimers

- DTHREADS usually *matches or exceeds performance* of pthreads. But *extensive use of locking* might degrade performance, as it might effectively serialize execution.

- DTHREADS can *increase* the program's *memory footprint*, since it stores per-process copies of pages (but they are recycles between transactions).
Summary

- DTHREADS is a deterministic replacement for pthreads.
Summary

- DTHREADS is a **deterministic replacement** for pthreads.
- It's easy to use, requires **no source code modification**.
Summary

- DTHREADS is a **deterministic replacement** for pthreads.
- It's easy to use, requires **no source code modification**.
- By **converting threads into processes**, DTHREADS leverages MMU protection to track and isolate concurrent memory updates.
Summary

- DTHREADS ensures determinism by having a deterministic commit protocol when threads reach synchronization points, and complete isolation of thread updates in any other case.
Summary

- DTHREADS ensures determinism by having a **deterministic commit protocol** when threads reach synchronization points, and complete isolation of thread updates in any other case.

- Deterministic thread execution has many benefits.
Questions?