Team5 Assignment4 Report2

107062313 黃寶萱　　107062117 李采蓉　　107062317 陳怡汝

◈ BufferMgr

1. 新增 AtomicBoolean 變數 hasWaitingTx 來記錄是否有 tx 正在 waiting，使用 AtomicBoolean 變數型別會比單純使用 Boolean 更加安全，因為 atomic 的特性可以確保變數的值，是一種 thread-safe 的方法，在後續的 pin()、unpin() 中可以用來判斷是否有 tx waiting，因此我們認為助教這樣的寫法更佳完善。
2. pin()、pinNew()
   A. 同樣縮小 critical section，但我們的做法是只在執行與 bufferPool 有關的操作時才 synchronized，如下方左圖黃色的部分，而助教的方法是將 bufferPool、waitingThreads 相關的區域都 synchronized，我們討論過後也認為應該 synchronized waitingThreads 才能確保 waitingThreads list 的正確性，避免同時有多個 thread 修改 waitingThreads，導致讀取到的資訊不相同，因此助教的作法較能確保系統的正確性。
   B. 新增 boolean 變數 waitOnce、更改 bufferPool.notifyAll()的執行順序
   助教的作法多用 waitOnce 來記錄該 tx 是否曾經 waiting to pin buffer，如果 waitOnce=true 代表該 tx 有 waiting 過，就需要去 notify 其他同樣在等待的 tx，我們的理解是因為如果該 tx 在 pin buffer 時需要等待，代表可能有其他的 tx 一樣 pin 不到 buffer 進入 waiting state，因此可以藉由判斷 waitOnce 的值來決定是否要 notify 其他 tx，但如果 waitOnce=false 就不用呼叫 notifyAll()，這是我們比較疑惑的地方，為什麼這樣就不用去叫醒其他的 tx。
   C. 更改 hasWaitingTx 的 value
   助教的作法在 pin()中會更改 hasWaitingTx 的 value，如果該 tx pin 不到 buffer 進入 wating 或是 waitingThreads list 不是 empty 便將該值設為 true。
   D. buffersToFlush.add(buff)
   我們沒有將 tx pin 到的 buffer 加到 buffersToFlush，這樣根本沒辦法將 dirty buffer flush 回 disk 上，我們在這裡發現我們做錯了，應該要將 pin 到的 buffer 加入 buffersToFlush 才對。

Team5	Solution

if (buff == null) {
waitingThreads.add(Thread.currentThread());
 	
while (buff == null && !waitingTooLong(timestamp)) {
synchronized (bufferPool) {
bufferPool.wait(MAX_TIME);
}
if (waitingThreads.get(0).equals(Thread.currentThread()))
buff = bufferPool.pin(blk);
}
 	
waitingThreads.remove(Thread.currentThread());
 	
// Wake up other waiting threads (after leaving this critical section)
synchronized (bufferPool) {
bufferPool.notifyAll();
}
}
	if (buff == null) {
	waitOnce = true;
	synchronized (bufferPool) {
	waitOnce = true;
	hasWaitingTx.set(true);
	waitingThreads.add(Thread.currentThread());
	
	while (buff == null && !waitingTooLong(timestamp)) {
	bufferPool.wait(MAX_TIME);
	if (waitingThreads.get(0).equals(Thread.currentThread()))
	buff = bufferPool.pin(blk);
	}
	
	waitingThreads.remove(Thread.currentThread());
	hasWaitingTx.set(!waitingThreads.isEmpty());
	}
	}

	// If it still has no buffer after a long wait,
	// release and re-pin all buffers it has
	if (buff == null) {
	repin();
	buff = pin(blk);
	} else {
	pinningBuffers.put(buff.block(), new PinningBuffer(buff));
	buffersToFlush.add(buff);
	}
	
	if (waitOnce) {
	synchronized (bufferPool) {
	bufferPool.notifyAll();
	}
	}

3. unpin()

同樣縮小 critical section 在 synchronized bufferPool，但助教多加判斷 hasWaitingTx 的 value，如果有 tx 在 waiting 才會呼叫 notifyAll()，這樣可以省去沒有 tx waiting 缺仍需要執行 notifyAll()的時間，因此助教的作法較完善。

Team5	Solution

if (pinnedBuff.pinCount == 0) {
bufferPool.unpin(buff);
pinningBuffers.remove(blk);
synchronized (bufferPool) {
bufferPool.notifyAll();
}
}
	if (pinnedBuff.pinCount == 0) {
	bufferPool.unpin(buff);
	pinningBuffers.remove(blk);
	
	// Optimization: If there are no txs waiting for pinning buffers,
	// skip notifying.
	if (hasWaitingTx.get()) {
	synchronized (bufferPool) {
	bufferPool.notifyAll();
	}
	}
	}

4. flushAll()、flushAllMyBuffers()、available()

同樣拿掉 synchronized，並無相異處。

5. unpinAll()

同樣縮小 critical section，改成只對 bufferPool 做 synchronized。

6. repin()

同樣縮小 critical section，只對 bufferPool 做 synchronized，但助教的作法新增 Map<BlockId, Integer> pinCounts 來記錄 pinning buffer 對應的 blockId 以及 pinCount value，但不太確定這樣做法的實際效果是什麼，對系統會造成甚麼影響。

Team5	Solution

`// Unpin all buffers it has`
`for (Entry<BlockId, PinningBuffer> entry : pinningBuffers.entrySet()) {`
`  blksToBeRepinned.add(entry.getKey());`
`  buffersToBeUnpinned.add(entry.getValue().buffer);`
`}`
 	
`// Un-pin all buffers it has`
`for (Buffer buf : buffersToBeUnpinned)`
`  unpin(buf);`
 	
`// Wait other threads pinning blocks`
`synchronized (bufferPool) {`
`  bufferPool.wait(MAX_TIME);`
`}`
	`// Unpin all buffers it has`
	`for (Entry<BlockId, PinningBuffer> entry : pinningBuffers.entrySet()) {`
	`  blksToBeRepinned.add(entry.getKey());`
	`  pinCounts.put(entry.getKey(), entry.getValue().pinCount);`
	`  buffersToBeUnpinned.add(entry.getValue().buffer);`
	`}`
	
	`// Un-pin all buffers it has`
	`for (Buffer buf : buffersToBeUnpinned)`
	`  unpin(buf);`
	
	`// Wait other threads pinning blocks`
	`synchronized (bufferPool) {`
	`  bufferPool.wait(MAX_TIME);`
	`}`

◈ BufferPoolMgr

我們的做法只移除 flushAll()的 synchronized，以下修改皆為助教的作法：

1. 更改 numAvailable 的 data type

   將 numAvailable 從原本的 int 改為 AtomicInteger，能確保一次只能有一個 thread 修改 numAvailable 的 value，避免系統出現錯誤，因此這樣的修改是有其必須性的。

2. 更改 lastReplacedBuff

   將 lastReplacedBuff 加上 violatile，這樣每當修改 lastReplacedBuff 時會馬上將 update 到 main memory，是一個能確保資料正確且更方便直接的做法。

3. 新增 ReentrantLock[] fileLocks、ReentrantLock[] blockLocks

   當每次要 pin buffer 時，需要先拿到先拿到該 block 的 lock，即存在 blockLocks

4. pin()

   A. Lock the specific block：首先要先拿到該 block 的 lock，避免在 pin buffer 的過程中有其他的 tx 更改 block 的 data。

   B. Lock the buffer：在找 unpinned buffer 的過程中，每個 iteration 都會先嘗試去拿該 buffer 的 lock，如果成功拿到 lock 再去看能不能 pin 這個 buffer，因為如果沒有 tx pin 這個 buffer，buffer 裡面的資料就可以 swap 成另一個 block data，為了避免在 swap data 的過程中有其他 tx 改動 buffer 裡面的資料，因此需要先 lock buffer，才可以 pin buffer。

C. Block lock 結合 buffer lock 的做法會比我們 synchronized 整個 pin()有更好的 concurrency，因為只有 access 同一個 block 的 tx 需要 waiting，如果是要 pin 不同的 block 就可以繼續執行，並不會互相影響。

Team5	Solution

synchronized Buffer pin(BlockId blk) {
Buffer buff = findExistingBuffer(blk);
if (buff == null) {
buff = chooseUnpinnedBuffer();
if (buff == null)
return null;
BlockId oldBlk = buff.block();
if (oldBlk != null)
blockMap.remove(oldBlk);
buff.assignToBlock(blk);
blockMap.put(blk, buff);
}
if (!buff.isPinned())
numAvailable--;
buff.pin();
return buff;
}
	Buffer pin(BlockId blk) {
	// The blockLock prevents race condition.
	// Only one tx can trigger the swapping action for the same block.
	ReentrantLock blockLock = prepareBlockLock(blk);
	blockLock.lock();
	try {
	// Find existing buffer
	Buffer buff = findExistingBuffer(blk);
	// If there is no such buffer
	if (buff == null) {
	// Choose Unpinned Buffer
	int lastReplacedBuff = this.lastReplacedBuff;
	int currBlk = (lastReplacedBuff + 1) % bufferPool.length;
	// Note: this check will fail if there is only one buffer
	while (currBlk != lastReplacedBuff) {
	buff = bufferPool[currBlk];
	// Get the lock of buffer if it is free
	if (buff.getSwapLock().tryLock()) {
	try {
	// Check if there is no one use it
	if (!buff.isPinned() && !buff.checkRecentlyPinnedAndReset()) {
	this.lastReplacedBuff = currBlk;
	// Swap
	BlockId oldBlk = buff.block();
	if (oldBlk != null)
	blockMap.remove(oldBlk);
	buff.assignToBlock(blk);
	blockMap.put(blk, buff);
	if (!buff.isPinned())
	numAvailable.decrementAndGet();
	// Pin this buffer
	buff.pin();
	return buff;
	}
	} finally {
	// Release the lock of buffer
	buff.getSwapLock().unlock();
	}
	}

5. pinNew()

A. 與 pin()相同在 pin 之前都需要先拿到 buffer 的 lock。

B. Lock the specific file：如果要在一個 file 裡面新增一個 block，需要先拿到該 file 的 lock 而不是 block lock，一次只能有一個 tx 新增該 file 的 block，最後不論是否成功 pin buffer 都要 release file lock。相較於我們直接對這整個 pinNew()做 synchronized，file lock 的做法可以有更好的 concurrency，因為只有 access 同一個 file 的 tx 需要 waiting，並不是每個呼叫 pinNew()的 tx 都需要等待。

6. unpin()

利用 buffer lock 在 unpin 時 lock 住該 buffer，相較於 lock 住整個 unpin()容易導致多個 tx 進入 waiting state，lock buffer 的方式可以使當要 unpin 同一個 buffer 時才需要 wait，能使系統有更好的 performance。

Team5	Solution

synchronized void unpin(Buffer... buffs) {
for (Buffer buff : buffs) {
buff.unpin();
if (!buff.isPinned())
numAvailable++;
}
}
	void unpin(Buffer... buffs) {
	for (Buffer buff : buffs) {
	try {
	// Get the lock of buffer
	buff.getSwapLock().lock();
	buff.unpin();
	if (!buff.isPinned())
	numAvailable.incrementAndGet();
	} finally {
	// Release the lock of buffer
	buff.getSwapLock().unlock();
	}
	}
	}

7. flushAll()

同樣移除掉 synchroized，但我們的做法是直接呼叫 buff.flush()，而助教的作法是要先拿到 buffer lock 才可以將資料 flush 回 disk，flush 完便 unlock，因此助教的作法較能確保系統資料正確。

Team5	Solution

void flushAll() {
for (Buffer buff : bufferPool)
buff.flush();
}
	void flushAll() {
	for (Buffer buff : bufferPool) {
	try {
	buff.getSwapLock().lock();
	buff.flush();
	} finally {
	buff.getSwapLock().unlock();
	}
	}
	}

◈ Buffer

我們和助教的 project 皆有使用 ReentrantReadWriteLock()藉以允許同時多個 thread 讀取以及同時一個 thread 寫入。針對 ReentrantReadWriteLock()的部份列出以下的不同，(+)表示我們新增 readLock 或 writeLock，(-)表示助教新增 readLock 或 writeLock：

1. ReadLock
   A. public BlockId block() (+)
   B. boolean isPinned() (+)
   C. boolean isModifiedBy(long txNum) (+)
   D. Page getUnderlyingPage() (+)
2. WriteLock
   A. void close() (-)
   B. void pin() (+)

C. void unpin() (+)

D. boolean isModified() (-)

E. void assignToBlock(BlockId blk) (+)

F. void assignToNew(String fileName, PageFormatter fmtr) (+)

除此之外，助教有將 pins 變數從 int 更改成 AtomicInteger 型態、新增型態為 AtomicBoolean 的 isRecentlyPinned 變數、新增型態為 boolean 的 isModified 取代 modifiedBy、新增變數為 lock 的 swapLock 以及新增兩個 functions：getSwapLock()、checkRecentlyPinnedAndReset()。這部份我們從助教新增的內容分成四個部分說明：

1. isRecentlyPinned

   多了 checkRecentlyPinnedAndReset()，用來確保 isRecentlyPinned 是否為 true 並且 reset 為 false。在 pin()呼叫時會將 isRecentlyPinned 設為 true，作為最近有被 pinned 的象徵。助教的方法是多新增一個 AtomicBoolean 變數來查看是否有最近 pinned 過，如此除了可以保持正確性還可以加快速度，所以我們認為助教的方法比較好。

2. isModified

   isModified 是用來取代 modifiedBy，不再用一個 set 去維護所有被 modified 的 txNum，而只是將 isModified 設為 true，所以在 modifiedBy.clear()時也只是將 isModified 設為 false。因此助教的 project 多了回傳 isModified 變數的 isModified()，少了回傳整個 set 的 isModifiedBy(long txNum)。只用 boolean 變數判斷就可以達到 synchronized 的效果並且可以減少紀錄每個 txNum 的時間，所以我們認為助教的方法比較好。

3. pins / swapLock

   多了 getSwapLock()，用來得到新增的 swapLock，可以確保在 pin()、unpin()、isPinned()前得到 swapLock，因此不須像我們的做法一樣再對這三個 function 做 readLock()和 writeLock()的保護。

   另外，修改 pins 變數時由於是 AtomicBoolean 的型態，可以維持其原子性確保正確，所以在 assignToBlock(BlockId blk)和 assignToNew(String fileName, PageFormatter fmtr)這兩個 functions 中不再需要用 writeLock()保護，而只需要判斷 pins 的數量大於 1 就丟 exception。這個方法是妥善運用 AtomicBoolean 的特性藉以提升效能，所以助教的方法比較好。

4. others

原本的 block()是有 sychronized，我們認為需要保持 synchronized 的特性所以有用 writeLock()確保同步，所以我們認為我們的方法比較好。

原本的 close()是有 sychronized，我們認為一個 buffer 的 close()不需要再 synchronized 因此沒有用 writeLock()確保同步，所以助教的方法比較好，是我們考慮的不夠全面。

雖然 getUnderlyingPage()只是用來 debug，但我們認為多新增 ReadLock 可以確保不會出錯，因此我們的方法更好。

◈ Page

助教的 Page.java 沒有做任何的更改，而我們 Page.java 有將除了 getVal(int offset, Type type) 和 setVal(int offset, Constant val)以外的 function 刪減了 synchronized。因為我們認為一個 page 對應到一個 buffer，而且 buffer 有做了不少同步的資料維護，所以有沒有 synchronized 並不會影響太多，但多加 synchronized 確實會比較安全，所以助教原先沒有更改任何部份的方法比較好，是我們考慮的不夠全面。

◈ FileMgr

我們的作法是因為 JavaNio 本身就有支援 multi-thread 的操作，因此在 FileMgr 的部分我們將所有 synchronized 移除，這部分與助教做法大致相同。而助教在優化上有另外對 file 的狀態進行 cache 的動作，透過 fileNotEmptyCache，在 file 不是 empty 或沒有紀錄在 fileNotEmptyCache 裡的時候加入或更新 file 的狀態，相較於原本的作法可以減少在 file 不是 empty 的時候重複計算 file size 的操作，這部分也是我們沒有特別考慮到的地方。

◈ BlockId

在 blockId 裡面我們在 Constructor 裡面對 hashCode 預先進行運算，以省下每次呼叫 hashCode()時的都必須重複計算 hashCode 的運算量，這部分與助教做法相同。而助教有另外優化 toString()，在一開始就先算好 MyString 避免每次呼叫 toString 都必須重複計算，這是我們沒有特別注意到可以優化的地方。

◈ JavaNioFileChannel

由於我們起初並未考慮到 javanio 資料夾的內容是可以被更動的，所以沒有對 JavaNioFileChannel.java 做任何更改，而助教的 JavaNioFileChannel.java 有使用 ReentrantReadWriteLock()藉以允許同時多個 thread 讀取以及同時一個 thread 寫入。因為 Java NIO 中的 FileChannel 提供了一種通過通道來訪問檔案的方式，可以通過這個文件通

道進行文件的讀寫，所以助教在此檔案針對讀寫做更進一步的優化，我們認為助教在這方面的做法比較好。

另外，助教還多新增維護 file 大小的變數，可以減少其中 size()的呼叫，藉以加速運行速度，這個做法我們沒有想過，可以藉此學習。

◈ JaydioDirectIoChannel

由於我們起初並未考慮到 jaydio 資料夾的內容是可以被更動的，所以沒有對 JaydioDirectIoChannel.java 做任何更改，而助教的 JaydioDirectIoChannel.java 有使用 ReentrantReadWriteLock()藉以允許同時多個 thread 讀取以及同時一個 thread 寫入。JavaNioFileChannel.java 與 JaydioDirectIoChannel.java 有相似的作用，差別在於後者有直接操作 Direct IO 的接口，因此助教是以相同方式處理兩個檔案，也有新增維護 file 大小的變數，這部份助教的 project 不僅讓我們更加認識 JavaNIO 和 Jaydio，也更加熟悉對於讀取和寫入 function 的判別。