國立交通大學

資訊科學與工程研究所

碩士論文

擴展檔案系統日誌功能至

非揮發性主記憶體

Extending file-system journaling to non-volatile main memory

研 究 生: 張逸康

指導教授: 張立平 教授

中華民國一〇一年八月

擴展檔案系統日誌功能至非揮發性主記憶體

Extending file-system journaling to non-volatile main memory

研 究 生: 張逸康　　Student:Yi-Kang Chang

指導教授: 張立平　　Advisor:Li-Pin Chang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2012

Hsinchu, Taiwan, Republic of China

中華民國一〇一年八月

擴展檔案系統日誌功能至非揮發性主記憶體

學生: 張逸康　　指導教授: 張立平

國立交通大學資訊科學與工程研究所碩士班

摘　　要

一般系統如果遇到斷電或是系統崩潰, 主記憶體上資料都將遺失。對於資料正確性要求高的環境, 會使用 UPS(uninterruptible power supply) 保護整個系統, 使其能夠安全關機。而 UPS 爲了維持整個系統運作, 需要不少電量, 尤其大型伺服器可能使用了上百台的主機跟儲存裝置, 若要以 UPS 保護系統則需要非常昂貴的成本。我們提出了一個方法使其能夠保護資料, 且不需要太大的成本。我們用非揮發性記憶體 (或是電池備援記憶體) 作爲主記憶體, 使其在斷電後可以保留住資料, 維持記憶體電力直到電源恢復。復電後再依照我們所設計之流程, 集合檔案系統日誌功能將資料寫回磁碟, 減少資料損失。

關鍵字: 日誌式檔案系統 (journaling file system)、非揮發性記憶體 (nvram)、電池備援記憶體 (battery backup memory)、資料回復 (data recovery)、電源中斷 (power interrupt)

Extending file-system journaling to non-volatile main memory

Student:Yi-Kang Chang Advisors:Li-Ping Chang

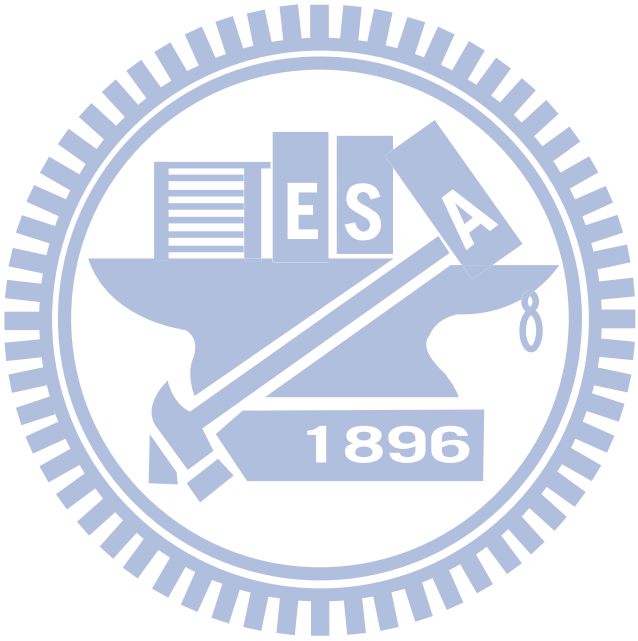
Department of Computer and Information Science
National Chiao Tung University

Abstract

In general system, all data in main memory are lost when occur power interruption. Use UPS(uninterruptible power supply) to protect entire system is a approach for reduce data losing. But it is also expensive for large system which need a high capacity UPS. We propose a new apporach:use standby power to protect main memory instead protect entire system. Maintain in-memory data until power restore and then write data to disk. We integrate characteristic of non-volatile memory and file-system journaling for reduce data losing in on power interrupt.

Keywords:journaling file system,nvram,battery backup memory, data recovery,power interrupt

目錄



圖目錄

一、簡介

傳統的檔案系統, 如果發生電源中斷或是系統崩潰, 造成檔案系統未正常卸載, 之後掛載系統可能會存在一些問題。由於檔案系統元資料 (metadata) 可能被修改到一半, 正處於不一致之狀態下遭到中斷, 造成資料結構損壞, 可能會導致檔案或是資料遺失。爲了降低災情, 會在檔案系統重新掛載之前進行檢查的指令, 如 fsck(file system check), 目的是盡可能維持檔案系統之一致性, 但這種檢查通常需要大量時間才可以完成。

爲了解決上述問題, 有學者提出了日誌式檔案系統 (Journaling File System)。日誌式檔案系統的概念, 是將欲寫入檔案系統之資料, 依照其提出寫回要求的時間適當的分成一個個集合, 作爲寫回的單位。並且在將該集合寫入檔案系統之前, 先寫一份到日誌之中, 這個動作稱爲提交。待提交完成之後, 再將資料寫入檔案系統。如果在寫入檔案系統過程中發生電源中斷或系統崩潰, 則在重新掛載時, 將資料從日誌中讀出, 再重新寫入檔案系統, 這個重新寫回的動作稱爲復原 (recovery)。同時考量檔案系統一致性, 若在資料提交過程中電源中斷, 則在復原時日誌將丟棄此不完整之資料集合, 不予以寫回。換句話說, 日誌式檔案系統的目標, 就是對於各個需寫回的資料集合, 只允許兩種結果: 完全寫回或是完全不寫。如此一來將可確保檔案系統之一致性。

雖然日誌式檔案系統可以確保檔案系統之一致性, 但是資料依然會因爲斷電而流失, 因爲作業系統核心的磁碟快取, 會將欲存入磁碟之資料暫時留在記憶體中, 以便於修改及操作。在電源中斷發生時, 記憶體中已被修改之資料若尚未提交至磁碟日誌中, 則因此而遺失。隨著科技進步, 主記憶體之容量越來越大, 磁碟快取所使用的空間也越來越高, 所以斷電造成遺失之資料量也相對提高。現今記憶體價格每 GB 約 140 元, 而磁碟每 GB 容量約 2.6 元。

對於資料安全性要求度高的環境而言, 會使用不斷電系統 (UPS) 供應電力, 讓系統能夠在失去一般電源供應後, 透過備用電源盡快將資料寫回磁碟並正常關機, 一般 UPS 要價約一、二千元。然而對於企業級的資料儲存系統, 背後往往有上百個主機在運作, 要提供大量的磁碟讀寫所需要的能量也相對高許多, 因此要建立大型的不斷電系統所需之成本也相對昂貴。

我們不樂見日誌式檔案系統放棄主記憶體及磁碟日誌中不完整的資料, 也不願以不斷電系統保護整個系統, 造成成本太過昂貴。我們需要的是一個折衷又兩全其美的方法, 使記憶體上資料能夠保住, 又不至於需要高昂的硬體成本。本篇提出一個方法, 即縮小保護的範圍僅保護主記憶體, 使系統在遭遇不預警斷電時, 記憶體上資料不會因此而消失。由於記憶體不似磁碟具有機械零件, 因此所需要之電量相對小很多。我們設計以電池作爲主記憶體之備用電源 (使用兩顆 Lithum Battery 18650的電池, 每顆電壓3.6伏特,2200毫安時), 成本僅需一、二百元, 一旦發生電源中斷, 主記憶體在失去一般電源供應後, 改由電池提供電力, 保護其資料直至電力恢復。

然而若將作業系統核心重新載入執行, 記憶體中資料就會被覆蓋, 因此我們必須搶在啓動程序 (bootloader) 底下先行分析主記憶體, 將需要寫回之資料複製到安全的區域, 然後在檔案系統掛載前寫入磁碟, 所以我們首先要面對問題就是如何分析記憶體上之資料。作業系統下資料結構之間往往都會使用指標相互指向, 不論是磁碟快取與檔案系統都使用了大量的鏈結串列 (linked-list) 結構。理論上以指標爲依據可以逐步摸索出鏈結串列中各個資料結構之位置, 但在無預警斷電下, 記憶體殘留之資料可能是作業系統運行中的任何狀態, 各資料結構之指標都有可能正被修改, 這意味著指標並不完全可信賴, 尤其鏈結串列中一旦有一個指標不正確, 後續資料結構都會找不回來, 因此要分析記憶體中資料結構並找出尚未寫回磁碟之資料, 並不能只依賴指標。當然我們可以透過掃描記憶體的方式找遍所有位址, 但是要能找出資料結構必須有識別的機制。

然而即使在分析記憶體之後能找出所有尚未寫回磁碟之資料, 也不可直接將資料寫入檔案系統, 因爲日誌中本就存在一些已完成提交, 但尚未寫入檔案系統的資料, 如果直接將記憶體中資料寫回檔案系統, 那麼在掛載檔案系統時復原過程才將日誌中資料寫入, 如此一來便破壞寫操作之順序性, 一旦發生同區塊 (block) 之舊資料覆蓋新資料的情況, 輕則視同資料遺失, 重則發生資料不一致致使檔案系統崩潰。因此對於記憶體中殘餘之資料, 必須先寫入日誌, 待掛載時復原過程依序寫回檔案系統才符合原順序性。但是若一味將資料寫入日誌, 也會在復原過程中被視爲垃圾資料而丟棄, 因爲日誌中含有特定之資料結構及管理資訊, 所以我們必須分析日誌的狀態, 才能正確的銜接日誌中的資料。

雖然透過電池保護可以將記憶體上所有資料保留下來, 再透過有效的策略分析出尚未寫回磁碟之資料, 但是將這些殘餘資料全數寫回未必是最適當的決定。因爲無預警斷電發生時, 不論是作業系統或是應用程式都是運作中遭到中斷且停止運行, 若將資料全數寫回會造成兩個問題, 第一是檔案系統一致性的問題, 對於修改中的資料若將其寫回, 可能會破壞檔案系統一致性, 因此我們配合日誌式檔案系統以交易爲寫回單位, 對於記憶體中完整的交易我們可以將其寫回, 因爲完整的交易是檔案系統確保一致性原則下的寫回單位。反之不完整的交易則捨棄。第二個問題是應用程式資料的完整性, 若是寫回不完整的修改資料, 可能反而不利於該應用程式, 因此哪些資料應該寫回須由程式自己判斷才行, 這個問題尚未有解決辦法, 需要有一個由應用程式主導的檢查點機制 (application-driven checkpointing mechanism)。礙於此極限問題之下, 我們目前只救回檔案系統認爲已經完備的資料, 即完整的交易。

本論文組織安排如下: 第二章介紹 background, 第三章說明系統概觀, 第四章講解回復程序, 第五章爲實驗結果探討分析。

二、相關研究

非揮發性記憶體可以在失去電源供應後保留住資料, 如同磁碟, 但是卻不會像磁碟讀寫時間這般長, 可以說是兼具效率與資料耐久性的儲存體, 透過這個特性, 許多學者提出了不同的學術發表來改善效率, 如 Baker,M. 提出以非揮發性記憶體作爲快取及緩衝[1]。他點出在分散式檔案系統中, 要減輕資料寫回壅塞的情況, 重點並不是需要更大的快取, 而是需要確保贓(dirty)資料之耐久性 (permanence)。這需要衡量效率與耐久性二者, 並在之間取得一個平衡點。某些經常被修改的資料, 會長時間停留在客户端的快取中, 然而爲了確保資料耐久性, 每隔一段時間都需要將資料寫回伺服器,Baker,M. 提出在客户端加入一塊非揮發性記憶體作爲快取, 使這一類資料不需要三不五時就將其寫回。另外他也提出在伺服器端增加一塊非揮發性記憶體作爲緩衝, 以減少磁碟存取之次數。這篇研究雖然改善了分散式檔案系統之存取效率 ,但是在資料耐久性考量上, 未能保護主記憶體, 一旦發生電中斷, 僅能保住非揮發性快取上之資料。

Wang, A.I.A. 設計出結合非揮發性記憶體與硬碟二者儲存裝置的檔案系統,Conquest 檔案系統[2], 其目的是爲了降低磁碟存取量並增進效率。Conquest檔案系統對於非揮發性記憶體與傳同磁碟有不同的儲存取向, 對於元資料及小型的檔案,Conquest 將其存放在非揮發性記憶體中, 而傳同磁碟僅用來存放大型檔案之內容 (大型檔案之元資料亦存放於非揮發性記憶體), 透過這樣的分工方式提高檔案系統之效率。雖然Conquest 確實發揮了非揮發性記憶體的儲存優點, 但以資料安全性角度而言, 對主記憶體上之資料仍未列入考量, 這一點與其他檔案系統並無甚大差異。Conquest 檔案系統一文中提到, 當系統故障後需透過系統備分還原資料, 此方法雖然確保系統正確性, 但依然會遺失系統中斷時記憶體中資料。

Cha, S.K 提出的 Xmas[3]採用備用電池記憶體來增進資料庫程式之效率, 以一塊具有電池備用電源的記憶體做爲日誌緩衝, 讓交易提交時不會產生磁碟I/O, 但是其主記憶體仍是採用揮發性記憶體, 如果電源中斷或系統崩潰, 記憶體中尚未提交到日誌緩衝的資料都將遺失。

另外, 由於 DRAM 必須不停刷新以確保記憶體中資料, 所以有學者提出以非揮發性記憶體取代主記憶體之研究, 如 Qureshi,M.K 提出以相變記憶體搭配 DRAM 的主記憶體[4](以相變記憶體爲主, 在附加上一小塊DRAM 做爲緩衝) 以及 Zhou,P. 提出使用相變記憶體完全取代 DRAM 爲主記憶體[5]等研究, 都對降低耗功很有幫助, 但是對於資料安全相方面的考量, 未有展現其非揮發性主記憶體之優勢。

現今非揮發性記憶體之單位價格日趨便宜, 應用也日見廣泛, 在各種不同系統中, 利用其快速又非揮發的特性完全或部分的取代了主記憶體及傳統磁碟儲存裝置, 提升了效率。然而將其用於主記憶體的應用類型中, 對於資料維護之問題, 卻多是將其儲存體特性和檔案系統二者獨立進行, 未能使檔案系統配合其非揮發性之特性加強主記憶體中之資料復原, 對於此點仍有進一步研究空間。

三、系統概觀

3.1 電池備援

一般小型UPS, 可以提供1KVA(1千伏安) 的容量, 要價約一千至兩千元。而我們使用了Lithum Battery 18650的電池,3.6伏特的電壓,2200毫安時, 成本約一百元。實驗系統使用512MB DDRII SDRAM 2條共1GB, 在斷電後電池可支持記憶體所需電力長達72小時。

3.2 Linux 分頁快取

Linux作業系統將主記憶體中一部份用來做分頁快取, 以增進效率, 分頁快取主要功能有二: 第一是將從磁碟讀取過的資料放在快取中, 當該資料再次被存取時可以直接從記憶體取得資料, 而不必再讀取磁碟, 第二是修改 (寫) 資料時, 僅修改快取中的資料, 等到適當的時機再將資料寫回磁碟, 兩種功能都有效的減少了磁碟存取次數。

如果要修改某個檔案的部分內容,Linux 會先將該部分從磁碟載入分頁快取, 使其得以修改; 同時由於該檔案之元資料 (metadata) 也被更改, 故也須將其載入分頁快取進行修改。也就是說, 分頁快取將包含此檔案之部分使用者資料 (user data) 集元資料之最新版本。

3.2.1 分頁替換演算法

Linux分頁快取包含了兩個 LRU(Least Recently Uesd) 串列, 分別是活躍串列 (active list) 與不活躍串列 (inactive list)。而每一個分頁的旗標 (flag) 包含 PG_active 與 PG_referenced 兩個位元, 用來表示該分頁的活躍程度, 進而決定分頁如何在兩個串列之間移動。PG_active 表示該分頁是活躍的, 應該放於活躍串列中,PG_referenced 表示該分頁最近被存取過。兩個旗標位元將分頁分成四大類, 如圖3.1所示, 左圓表示不活躍串列, 右圓表示活躍串列, 串列中各分頁依照 PG_referenced 位元, 可以想像成兩個分頁子集合 (黑色橢圓), 分頁快取透過兩函數操作旗標位元, 其中紅色箭號表示 maek_page_accessed() 函數, 一但分頁被存取, 就會執行該函式。綠色箭號表示 page_referenced() 函數, 當系統要回收分頁時, 會先對活躍串列進行掃描, 並執行該函釋將 PG_referenced 清除, 接著再掃描不活躍串列, 執行該函釋將 PG_referenced 清除, 若一個屬於不活躍串列中的分頁, 其 PG_referenced 位元也沒被設置, 則會成爲替換演算法的

回收對象。

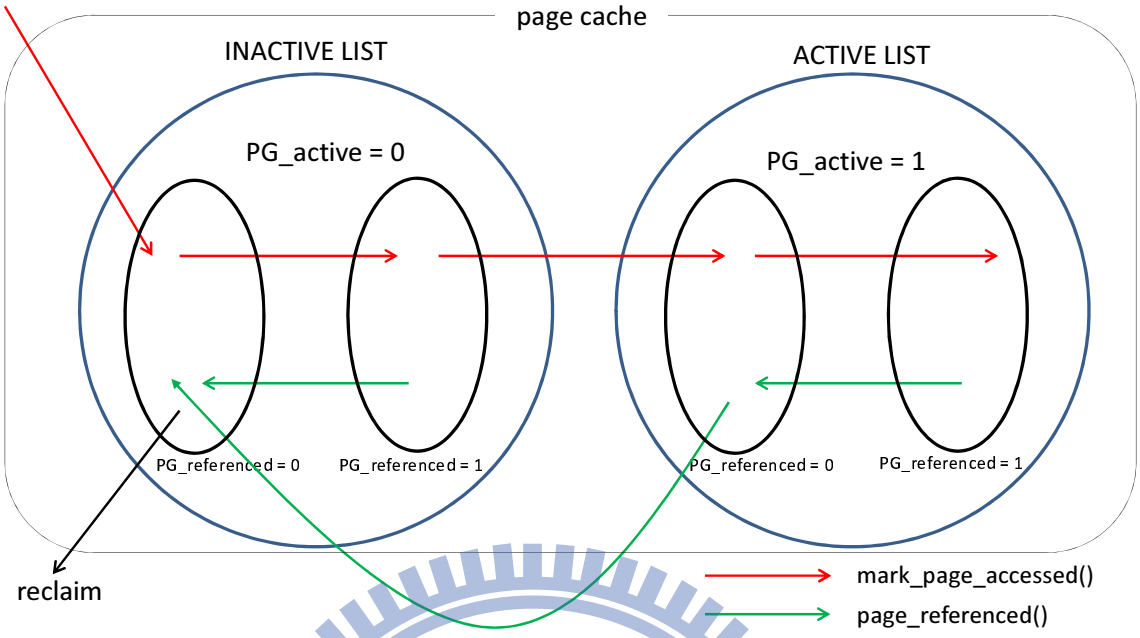


圖 3.1: Page Cache

3.2.2 分頁快取資料結構

在 Linux 中, 每一個被存取的檔案都會以一個基樹 (radix tree) 資料結構對應之, 使其能夠快速地找到各個分頁 (page), 圖3.2表示一個檔案如何透過基樹與分頁編號找到指定的分頁。由 root 開始依照分頁編號01-11-10逐層尋找節點, 最終找到其分頁位址。若最底層指標爲空指標, 代表該分頁未載入快取, 需從磁碟上取得資料。

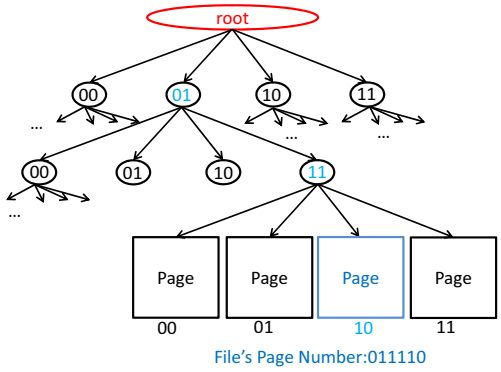


圖 3.2: Radix tree

每個分頁包含一個或數個緩衝區首部 (buffer head), 每一個緩衝區首部對應磁碟上一個區塊, 並記錄著一個指向緩衝區 (buffer) 的指標及其相關資訊。假設系統分頁爲4k byte, 磁碟區

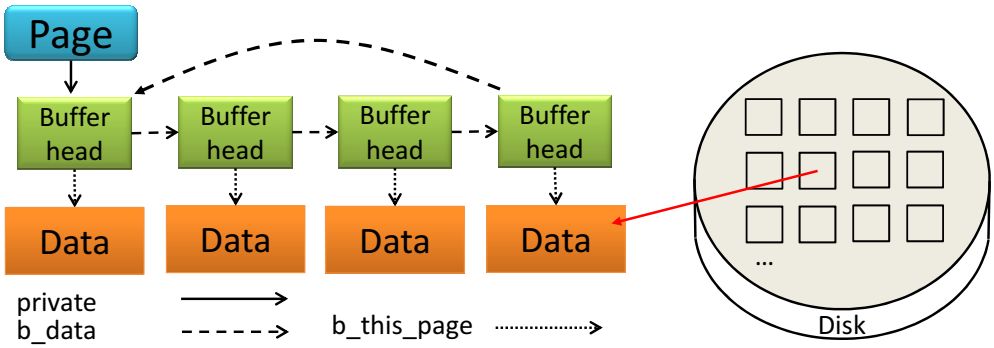


圖 3.3: Page and buffers data structure

塊爲1K byte, 則一個分頁會包含四個區塊之資料, 如圖3.3所示,Page 結構的 private 指標指向第一個緩衝區首部, 各緩衝取首部之間以 b_data 指標連成一個循環, 並各自由 b_this_page 指標指向資料所在位址。當快取中被修改過的分頁被沖回 (flush) 磁碟時, 系統會判斷該分頁所包含之緩衝是否爲髒資料 (dirty data), 並將包含髒資料之緩衝寫回對應之區塊。

Linux分頁快取透過 pdflush 執行緒周期性的將髒資料寫回磁碟, 其相關設定參數之檔案在/proc/sys/vm 底下:

1. dirty_writeback_centisecs : pdflush 執行緒喚醒周期, 預設是5秒
2. dirty_expire_centiseconds : 髒資料在分頁快取中最長保留之時間, 預設是30秒, 超過時間後作業系統將開始強制將其寫回磁碟
3. dirty_background_ratio : Linux 允許髒資料佔記憶體中的比例, 預設是10%, 一旦超過此比例, 將強制喚醒 pdflush 執行緒進行寫回的動作

3.3 日誌式檔案系統

3.3.1 日誌運作原理

更改一個檔案的內容通常, 除了檔案本身內文更改之外, 其相關之元資料也可能需要修改。這些資料的修改必須全部寫回後才能算是完成更改, 如果在執行過程中發生電源中斷或是系統崩潰, 造成部分的寫入操作沒有執行, 將使檔案系統不一致。舉例來說, 刪除 ext 檔案系統中的一個檔案會修改到這些元資料:

1. 從該檔案所屬之目錄中刪除該檔案之目錄項 (directory entry)。
2. 修改 inode bitmap, 空出該檔之索引節點 (inode) 空間。
3. 修改 block bitmap, 釋放出該檔案所佔據之區塊

如果在步驟1與2之間發生電源中斷, 由於該檔案之目錄項已被刪除, 因此已經無法從檔案系統目錄下看到該檔案, 但是該檔所佔據索引節點及區塊卻未釋放, 將永久佔據檔案系統之空間。反之如果先執行步驟3, 而在步驟1、2執行前發生電源中斷, 則該檔看似存在, 但其內容所佔據之區塊卻可能會被檔案系統分配給其他檔案而被複寫, 造成內容錯誤。故一旦發生電源中斷或系統崩潰致使檔案系統未正常卸載, 傳統檔案系統會在系統重新啓動時進行檢查的動作, 如 fsck(file system check) 檢查檔案系統不一致問題並修復之, 這往往耗費大量的時間。

3.3.2 處理與交易

日誌式檔案系統將寫操作所修該之資料依照其提出寫回要求的時間適當的分成一個個集合, 作爲寫回之單位, 稱爲交易 (tansaction)。每個交易包含許多個不可分割之寫操作, 在 ext3底下, 不可分割之寫操作稱爲處理 (handle)。以上述舉例來說, 三個步驟合在一起才完成檔案刪除的動作而不可分割, 因此這三步驟加在一起視爲一個處理。ext3檔案系統中一個處理所包含的寫操作, 必定會屬於同一筆交易。

爲了避免太頻繁存取磁碟,ext3不是以處理作爲寫入單位, 而是以交易作爲寫入單位, 一個交易的生命週期會依序經過幾個重要的狀態:

1. RUNNING: 此狀態代表該交易是此時系統中唯一可以接受寫操作資料的交易。
2. LOCKED: 此狀態表示本交易不再接受新的寫操作, 並等候本交易各處理尚未修改完之資料, 準備進行提交。
3. FLUSH: 此狀態表示正將資料提交到日誌中。

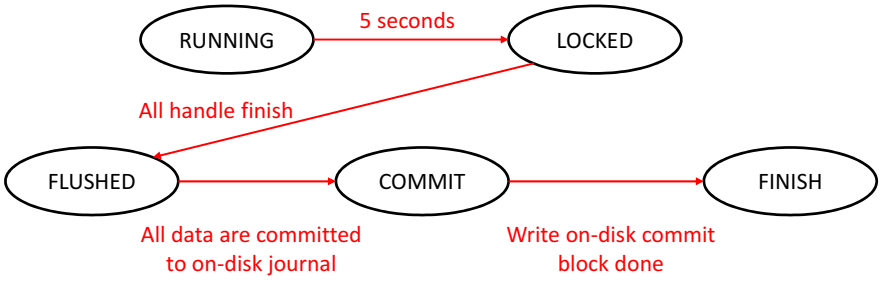


圖 3.4: Transaction State Transition

4. COMMIT: 此狀態表示該交易資料已寫入日誌, 接下來要寫一個提交區塊, 用來表示此交易完整提交。

5. FINISHED: 此狀態表示以提交完畢, 等待資料全數寫入檔案系統後即釋放。

交易各狀態轉變如圖3.4所示, 第一個狀態是 RUNNING, 運行中交易 RUNNING 狀態預設是5秒, 超過5秒系統就會要求提交運行中交易, 這件工作是由一支核心執行緒 kjournald 負責, kjournald每5秒被喚醒一次, 它會檢查當前的運行中交易是否已超過5秒, 如果超過5秒的話就會要求系統進行提交。

圖3.5表示 kjournald 執行緒之流程, 首先會判斷 journal 結構底下的 j_commit_sequence 欄位與 j_commit_request欄位之數值是否相等,j_commit_sequence 之值爲系統最近提交的交易序號, 每個交易提交完成之後都會將序號指定給 j_commit_sequence, j_commit_request之值則是系統接下來想要提交的序號, 若二者相同則代表從上一個交易提交完成到現在並無新交易欲進行提交。一般而言, 此流程初期經過此判斷時, 會得到二者相等的結果, 接下來會檢查交易計時器是否超過5秒, 若超過5秒就會將 tid 指定給 j_commit_request, 接著回到第一個判斷, 此時判斷會得到二變數不相等的結果, 於是就進行提交。

提交過程一開始會將交易狀態設爲 LOCKED, 這個時候開始, 這個階段主要是等待尚在修改資料的各處理 (即 handle, 如本節開頭所述) 結束, 它透過交易結構底下的一個變數 t_updates 代表該交易尚有多少處理未結束。一旦 t_updates 歸零, 才可以進入下一個狀態。t_updates 欄位將在3.3.5節作進一步說明。

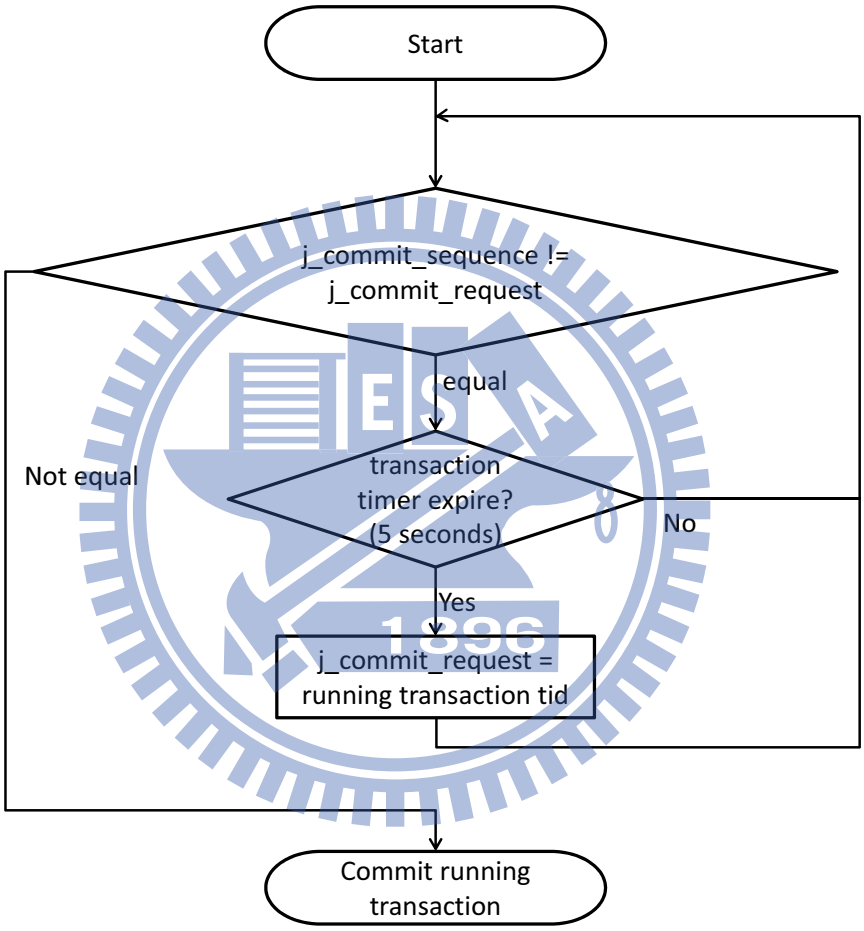


圖 3.5: kjournald

接下來會將狀態設爲 FLSUH(並將此交易由運行中交易改爲提交中交易), 接著依據日誌模式將資料寫到檔案系統或日誌。如果在此階段, 正要寫入日誌的資料恰好又被新的運行中交易修改, ext3會複製一份 read-only snapshot, 並由一個日誌首部的指標 b_frozen_data 存下此 snapshot 的位址, 這一份複製資料可供本交易提交過程寫入日誌, 而原本的資料則可以讓新的運行中交易修改。

當本交易之資料都寫入日誌後, 將狀態改爲 COMMIT 並寫一個提交區塊到日誌, 用來表示此交易資料完整無誤。如果發生電源中斷或系統崩潰, 在重新掛載檔案系統時復原程序會逐一檢查磁碟日誌中的交易, 將已完成提交之交易資料寫入檔案系統 (具有提交區塊之交易), 若發現最末一個交易尚未提交完畢就遭中斷 (未找到提交區塊), 則將之丟棄, 也就是說在復原程序進行時, 磁碟上的提交區塊存在與否決定了整個交易資料的去留。

最後一個狀態是 FINISHED, 此時資料已經寫到磁碟上的日誌, 而提交過程也到此結束。ext3會將這種已經完成提交 (寫到日誌), 但資料還未寫入檔案系統的交易加入到 journal 結構下的 j_checkpoint_transaction鏈結串列, 每當有新的交易開始提交時, 系統會順便檢查該串列上各 FINISHED 交易, 一旦發現某個交易所有資料都已寫到檔案系統, 則該交易才由此串列中刪去, 並釋放之。這個狀態的交易在記憶體中可能存在多個, 此狀態表示提交過程已經結束, 只差資料尚未完全寫到檔案系統, 即使發生電源中斷, 在檔案系統重新掛載之後也可由日誌中找回該交易之資料。

3.3.3 記憶體交易資料

同一時間系統中提交中交易及運行中交易最多只能各存在一個, 每個交易都有一個獨一無二的交易序號 (transaction id), 圖3.6表示記憶體與磁碟中各種狀態交易之情形。左邊記憶體中包含運行中交易 (交易序號101), 正在接受新的資料; 提交中交易正在進行提交 (交易序號100), 磁碟上正逐步將提交中交易之資料寫入日誌; 記憶體中也可能存在一些已經完成提交之交易 (交易序號99), 這種交易雖然已提交到日誌中, 但尚未全部寫入檔案系統, 待資料都寫入檔案系統後才可從記憶體釋放。就此時間點來看, 記憶體中的運行中交易 (序號101) 是一個不完整的交易, 而提交中交易 (序號100) 是一個完整的交易, 但由於磁碟上提交中交易之資料尚未提交完

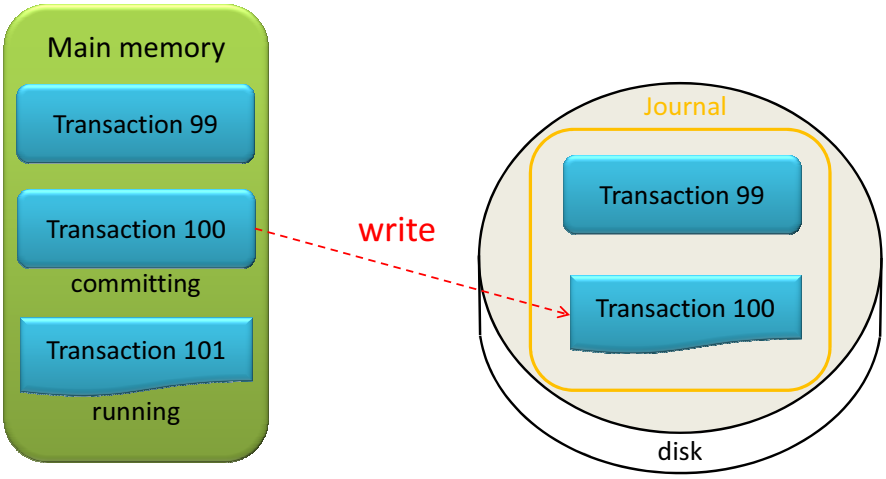


圖 3.6: Running transaction and Committing transaction

成, 故就磁碟資料來看, 提交中交易是不完整交易, 僅有更早之前已然寫入之交易 (序號99), 才是完整的交易。

如果在這個時間發生電源中斷, 在主記憶體是揮發性記憶體的情況下, 記憶體中各交易資料都將遺失, 其中已完成提交的交易 (序號99) 由於資料已經寫到磁碟於日誌中, 故可以在檔案系統復原程序中將資料從日誌寫回檔案系統, 若是使用預設模式 (即 ordered mode), 對此交易而言並不會造成資料損失。因爲日誌機制會確保先寫入使用者資料到磁碟檔案系統中, 才會開始將元資料寫入到日誌。請注意寫到磁碟日誌跟寫到磁碟檔案系統不同的, 參考圖3.7。但如果是採用 writeback 模式 (日誌操作模式之一) 由於無法保證使用者資料已經寫入磁碟, 所以這部分可能會造成資料遺失。所以嚴格上來說, 完成提交的交易資料若從記憶體中失去, 是否會造成資料遺失與日誌保護資料的方式有關, 關於日誌保護資料的模式將在下一節介紹。而提交中交易雖然在磁碟日誌中已有部分資料, 但由於提交不完整, 故會在檔案系統復原程序中被丟棄, 整個交易之資料將全部遺失; 運行中交易之資料完全不曾寫入日誌, 故該交易資料也隨電源中斷而全數消失。

3.3.4 日誌操作模式

關於日誌式檔案系統的設計, 目前已有許多研究。Ext3檔案系統便是基於 ext2檔案系統架構上附加了日誌的功能[6]。Ext3檔案系統分爲三種模式:

1. Journal 模式

2. Ordered 模式

3. Writeback 模式

Journal模式不論對於檔案資料及元資料都會先提交到日誌, 如圖3.7(a) 所示, 等待提交完畢之後再把資料寫入檔案系統, 是最完善的模式, 但耗費時間也最多。Ordered模式在提交交易時會先將檔案資料寫到檔案系統(圖3.7(b)), 待寫完之後, 才將元資料提交至日誌中, 等提交完畢之後再將元資料寫回檔案系統。這種模式僅將元資料提交到日誌, 所以速度較 Journal 模式快, 但是如果在提交元資料時發生電源中斷, 復原後日誌中最後一筆交易之元資料會被丟棄, 但是該交易的檔案資料已然寫入檔案系統, 無法回復成該交易寫入之前的狀態。Ordered 模式也是 ext3 的預設模式。Writeback 模式也僅將元資料寫入日誌 (圖3.7(c)), 它與 Ordered 模式差異在於 Writeback 不需要等到檔案資料完全寫入檔案系統之後才開始提交元資料到日誌中, 也就是說此模式可以同時寫檔案資料及提交元資料, 故此模式速度較快。

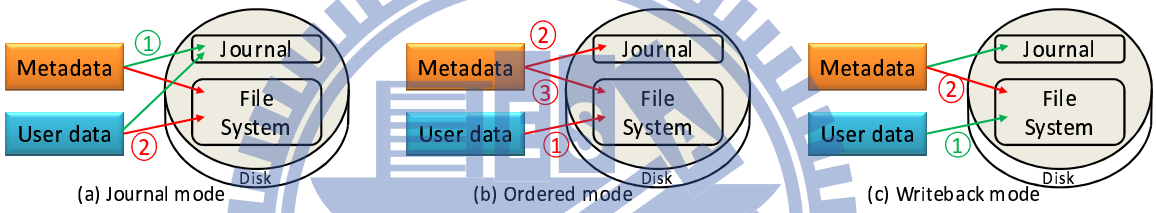


圖 3.7: Journaling Modes

日誌式檔案系統再將資料寫回磁碟時, 是以交易做爲寫回單位, 每隔一段時間將快取中部分被修改之資料一起寫入磁碟的日誌, 之後再將資料寫入檔案系統。所謂的交易是指依序列對檔案系統的寫入操作, 交易必須符合 ACID 法則:

1. 原子性 (Atomic)

   交易作爲一個整體被執行, 包含在其中的對檔案系統的操作要麼全部被執行, 要麼都不執行

2. 一致性 (Consistency)

   交易應確保檔案系統的狀態從一個一致狀態轉變爲另一個一致狀態。一致狀態的含義是檔案系統中的資料應滿足完整性

3. 隔離性 (Isolation)

   不同交易之間不互相影響, 提交中的交易內容不應該被運行中交易所改變

4. 持久性 (Durability)

已完成提交之交易對檔案系統之修改應該永久保存在檔案系統中

滿足上述四項法則之設計, 一旦發生無預警斷電, 系統重開後進行復原程序時, 日誌中之交易只允許有兩種結果:1. 資料全部寫入檔案系統中。2. 丟棄, 全部不寫入檔案系統 (原子性)。如此一來可以確保檔案系統處於一致的狀態 (一致性)。

3.3.5 主記憶體日誌資料結構

不同的日誌式檔案系統有不一樣演算方法及資料結構, 我們以 ext3檔案系統爲研究對象, 它是以 ext2爲基礎架構, 再增加日誌功能。圖3.8表示 ext3檔案系統在記憶體中幾個重要結構之間關係,ext3中有個核心結構:Struct journal_s, 該結構記錄 ext3日誌功能運作所需要的相關訊息, 其中兩個重要的指標:j_running_transaction 及 j_committing_trans-action, 用來指向系統目前「運行中交易」(running transaction) 以及「提交中交易」(com-mitting transaction)。其中運行中交易是系統當下可接受應用程式提出寫回操作之交易, 當系統執行要一個操作時, 系統會分配一個處理資資料結構 (handle_t), 並且將運行中交易的 t_updates 加一, 接著對於每個被修改的緩衝區, 會分配一個對應的日誌首部 (journal head) 並依各緩衝區資料性質, 加入對應的雙向鏈結串列。如圖3.8表示 Ordered mode 底下, 對於使用者資料的日誌首部會加入 t_sync_datalist 串列, 而元資料則會放到 t_buffers 串列, 待所有日誌首部接加入運行中交易對應的串列後, 該處裡才算結束, 此時 t_updates 則減一。從一個處理的開始到結束 (t_buffers 加一到減一) 僅只對記憶體中資料修改並且將日誌首部插入提交中交易而已, 尚未將任何資料寫入磁碟 (寫入磁碟是提交階段才會執行的工作)。

等到一定時間之後 (預設是5秒) 會開始進行提交, 成爲提交中交易。提交中交易表示已經進入了提交階段, 不再接受寫操作。當一個分頁快取緩衝區之資料要寫回磁碟時,ext3會分配該緩衝區首部一個對應的日誌首部 (journal head), 並且將日誌首部加入當前運行中交易底下的鏈結串列中。如圖3.8所示, 在 Ordered mode 底下, 運行中交易具有兩個不同的鏈結串列, 分別用來串連使用者資料之日誌首部, 以及元資料之日誌首部。日誌首部與緩衝區首部兩結構間是一對一關係, 故一個日誌首部即代表一個磁碟區塊的寫操作。

每一個交易都有一個獨一無二交易序號 (tid - transaction id), 系統中依遞增順序賦予各

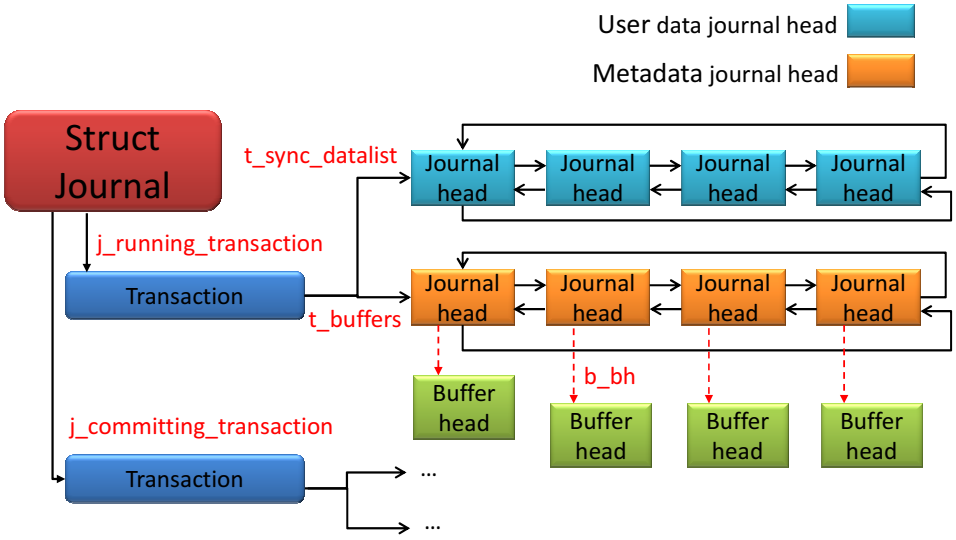


圖 3.8: The relation between journal,transaction and journal head

交易序號, 因此序號大小代表著時間先後之意義。圖3.9表示各交易之間順序關係, 橫軸是時間軸, 右邊是目前時間點。tid 97、98兩個交易已經完成提交, 是比較老的交易;tid 99是提交中交易, 較前者爲新; tid 100是運行中交易, 也是目前可以接受寫操作之交易, 是最新的交易。ext3任一個時間點僅允許 (最多) 一個運行中交易以及提交中交易, 故二者間序號差值必爲1。

以 order mode 爲例, 提交中交易會依序進行三個主要動作 (圖3.10):

1. 將使用者資料寫入檔案系統
2. 將元資料提交至日誌
3. 將元資料寫入檔案系統

在 order mode 下, 三個動作不可並行, 換言之提交元資料到日誌中必須要等所有使用者資料都寫入檔案系統, 其等待方法如下所述: 使用者資料在交易 RUNNING 狀態時, 原是加入到

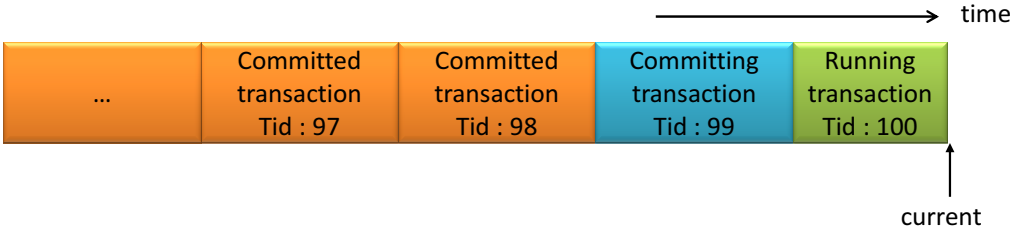


圖 3.9: Transaction timeline

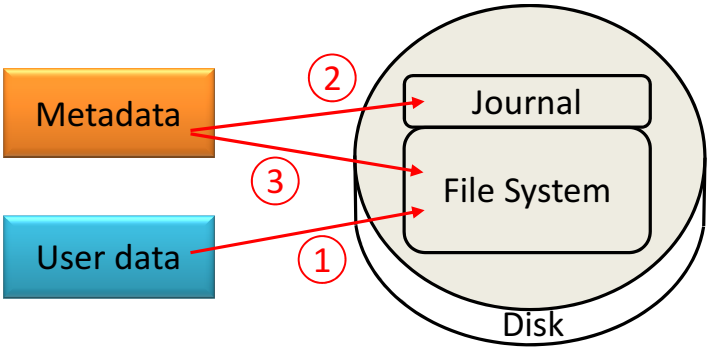


圖 3.10: Ordered Mode

t_sync_datalist 串列中, 當交易進入提交階段的 FLUSHED 狀態後, 對於每個日誌首部對應的資料都送出寫回磁碟的請求, 並將日誌首部一一移到另一個雙向鏈結串列 (t_locked_list), 接著透過 while 迴圈不停的檢查該串列第一個日誌首部之資料是否已經寫回磁碟, 若日誌首部所對應的資料已經寫入磁碟, 則將該日誌首部 (第一個) 釋放, 下一次迴圈則會開始檢查串列中下一個日誌首部 (原第二個, 由於第一個日誌首部已被移除釋放, 故成爲第一個)。若尚未寫回磁碟, 則提交程序會一直滯留在 while 迴圈, 直到所有使用者資料的 I/O 都完成。提交程序才會進到下一個階段, 把元資料寫到日誌中。同樣的等到元資料都提交到日誌之後, 才可以將元資料寫入檔案系統。上述第1、2兩步驟完成後才算結束提交階段, 交易狀態狀態改爲 FINISHED, 並把交易加入 j_checkpoint_transaction 串列中, 第3步驟的磁碟寫入並不算在提交過程之內, 也不使用 while 迴圈等候其 I/O 完成, 提交過程最後部分僅送出寫回磁碟的請求, 接著即將狀態改爲 FINISHED, 待其自然完成。不論是哪一種日誌模式, 在 j_checkpoint_transaction 串列底下的交易, 都只有在該交易資料全數寫回磁碟之後, 才會從串列中移除, 並釋放之。

3.3.6 磁碟日誌資料結構

ext3將日誌隱藏起來, 它不存在於任何一個目錄 (directory) 底下, 無法透過目錄樹狀結構描繪出它的位置, 是一個特別的檔案。要找到日誌檔案必須直接從該磁區之超級區塊 (superblock) 中取得其索引節點編號 (inode number), 再透過 ext3的 (組描述符)Group Descriptor 找到包含此索引節點的索引節點表 (inode table), 最後從索引節點表中讀出日誌的索引節點, 即可掌握日誌檔案在磁碟上的各個資料區塊 (更詳細的流程將會在下一章介紹)。索引節點資料結構紀錄了一個檔案各區塊之磁區邏輯區塊號碼 (logical block number)。

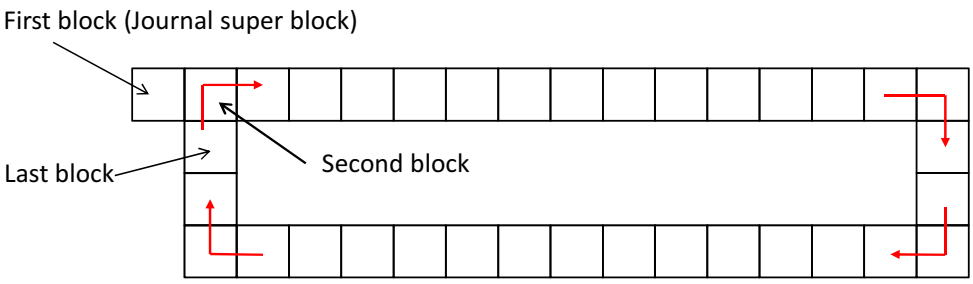


圖 3.11: Ring structure journal file

日誌檔案的第一個區塊爲日誌超級區塊 (journal super block), 記錄著日誌所有的狀態資訊, 其餘的區塊則用來存放已提交交易的資料。如圖3.11所示, 日誌檔案從邏輯邏輯角度來看可將之視爲一個環形的資料儲存結構,ext3從第 2 個區塊開始填入資料, 依序使用區塊直到最後一個區塊填入資料後, 再次從第 2 個區塊開始重複使用。

一個提交中交易寫入日誌中會包含數種區塊, 分別是取消區塊 (revoke block)、描述區塊 (descriptor block)、資料區塊 (data block) 以及提交區塊 (commit block)。取消區塊紀錄該交易中被刪除的區塊號, 當檔案系統進行復原程序時, 會先依照個交易之取消區塊建立 hash table, 當某個區塊在序號 k 的交易中被刪除 ,代表所有序號小於 k 之交易都不需要對該區塊寫回資料, 因爲復原程序最後也會將之刪除, 如此一來就可以省下時間。取消區塊僅是爲了加速復原程序之速度, 一個交易提交至日誌不一定會有取消區塊。資料區塊之內容即爲緩衝區之資料, 也就是欲寫回檔案系統之資料內容。描述區塊中包含許多個日誌區塊標籤 (journal block tag), 每個標籤記錄著一個資料區塊所對應的磁區區塊號碼, 依照其順序, 第一個標籤所紀錄之區塊號碼爲該描述區塊往後數起第一個資料區塊的磁區區塊號碼。提交區塊代表此交易到此區塊結束, 作爲識別交易是否完整提交的判斷依據。

磁碟中有一緩衝, 當資料從作業系統發送給磁碟後, 會先暫存在緩衝中, 之後磁碟會配合其物理特性將資料重新排序 (reorder), 以求能更快地讀寫。然而日誌式檔案系統需要透過提交紀錄 (commit record) 來判斷交易是否完整 (以 ext3而言提交紀錄就是日誌中交易最末端的提交區塊)。理論上而言, 提交紀錄會是一個交易提交時最後一筆寫入的資料 (最後寫入的意義在於必須保證前面的資料都寫已畢, 本交易資料都已完整地寫入日誌), 如果在重新排序之後, 一個交

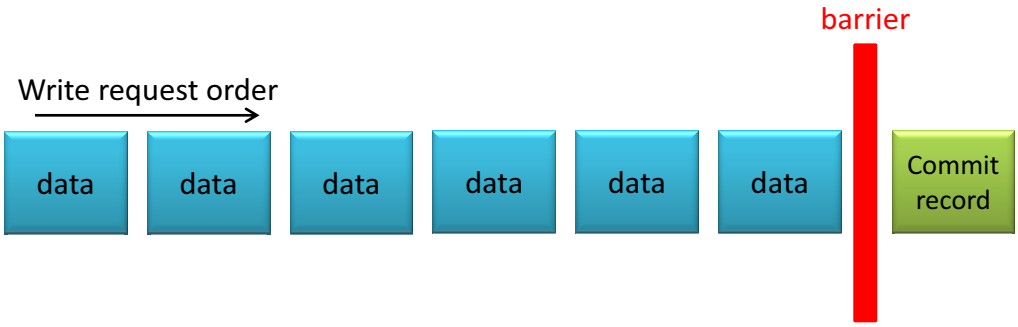


圖 3.12: Add barrier before commit record

易的提交區塊被安排在該交易的某些資料前寫入磁碟日誌, 此時一旦發生斷電, 待系統重開進行復原程序時就會發生問題, 因爲該交易之資料尚未寫完, 但是卻先寫了提交區塊, 造成復原程序誤判資料是完整的, 而將其寫至檔案系統, 如此一來就違背了交易的原子性, 同時也可能造成檔案系統不一致。

爲了解決這個問題, 有學者提出在寫回資料之間插入一些屏障 (barrier), 並要求資料寫回時必須先將屏障之前資料寫完, 才能寫屏障後的資料。最基本的實作方式就是系統先要求磁碟將緩衝資料沖到磁碟中, 接著系統才送出屏障之後的寫回資料給磁碟。透過這樣設計, 將屏障插入在提交紀錄之前 (如圖3.12所示), 如此一來雖然磁碟會進行重新排序, 但是緩衝區之中只有資料區塊可以進行重排, 之後系統要求沖下緩衝區的資料, 才將提交紀錄發送給磁碟, 如此一來提交紀錄依然會成爲交易最後一個寫入磁碟的區塊, 這樣就可以確實透過提交紀錄來判斷該交易是否完整寫入磁碟。

3.4 分頁取代與交易提交

本章到上一節爲止, 已介紹 ext3檔案系統的日誌功能運作原理以及實作所使用的資料結構,ext3檔案系統是建立在 ext2的架構上再加入日誌功能與 ext2相結合, 而有關 ext3的日誌功能被抽象成一個核心層 (kernel layer) 稱爲 JBD(journal block device), 它並不是一個眞正的裝置, 在 Linux 底下並沒有對應的裝置檔案 (device file)。本章所敘述的日誌運作原理及所各種資料結構的操作都包含在這一層裡面, 例如交易、處理、日誌首部等資料結構, 交易運行、提交等處理流程都算是在 JBD 一層中, 我們可以說 ext3 = ext2 + JBD。

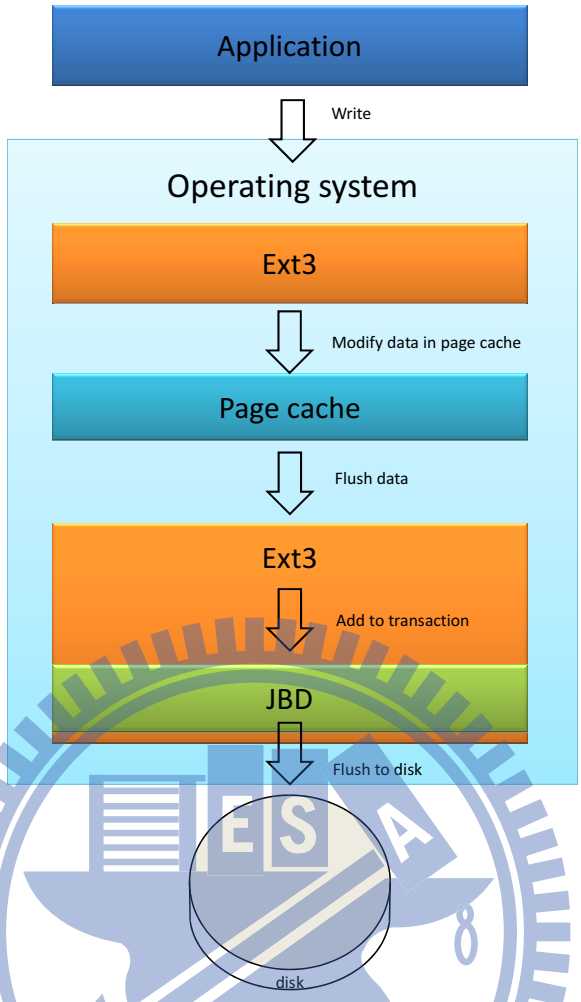


圖 3.13: page cache, ext3 and JBD kernel layers

Linux的分頁快取與檔案系統是分開的兩層核心層, 當分頁快取之資料要寫回磁碟時, 會呼叫該磁區掛載的檔案系統之對應函式 write_ page, 這是一個函式指標, 該指標會指向掛載的檔案系統其相對應函式, 由此可見, 分頁快取並不干涉檔案系統如何操作這些寫回的資料, 如果掛載的是日誌式檔案系統, 那麼會對資料採取特殊的寫回機制以實現日誌功能, 分頁快取也跟該寫回機制無關。若掛載的是 ext3檔案系統的 ordered 模式, 則 write_ page 指標會指向 ext3_ordered_writepage, 由 JBD 相關函式處理後續的流程。如圖3.13所示, 當使用者在修改資料時, 作業系統會根據要修改的資料, 讓檔案系統把資料找出來, 並將資料暫時存放在分頁快取中, 以便使用者修改。當分頁快取中的資料要衝回磁碟, 會呼叫 ext3對應的函式, 而 ext3則會將資料將資料分配日誌首部並加日運行中交易, 這些日誌功能的相關動作是由 ext3的 JBD 完成, 最後資料由 JBD 衝回磁碟, 完成寫回程序。

當檔案系統的架構改變時,ext3會用一個處理 (handle) 保護這一連串對元資料修改之原子性, 使這些修改的元資料資料在一個處理開始到結束這段期間, 逐一交由 JBD 接手, 做之後寫入磁碟的之相關準備。改變系統架構的時機有兩個:

1. 執行系統呼叫造成檔案系統架構之改變 (例如: 刪除一個檔案)
2. 將使用者資料寫回磁碟時造成檔案系統架構上的改變 (例如: 分配一個新的磁碟區塊給某個檔案)

以下對於這兩種時機分別作說明。

第一個時機點是在執行系統呼叫時, 可能會對應該呼叫改變檔案系統之架構, 而遭到修改的元資料則會交由 JBD 接手, 並且 JBD 會在一個處理期間 (處理開始到結束) 把這些資料加入到運行中交易。這裡檔案系統架構的改變對於該系統呼叫是即時性的, 也就是說, 隨著該系統呼叫的過程中, 這些元資料就被加入到運行中交易。比如說 ext3 刪除一個檔案時, 會呼叫 ext3_dirty_inode 函式, 並開始一個處理。接著檔案系統會完成下述步驟:

1. 在釋放檔案各個區塊, 透過索引節點的 direct, indirect, double indirect, triple indriect 等等區塊指標, 將所有區塊都釋放 (實際上就是將這個檔案占用的區塊所對應的區塊位圖 (block bitmap) 中的位元設爲 0)。
2. 將上述修改的區塊位圖加入到運行中交易
3. 把索引節點的 direct, indirect, double indirect, triple indriect 設爲 0
4. 將索引節點部分欄位清除, 並寫下刪除時間
5. 將索引節點表中 (inode table), 包含此索引點之區塊的緩衝區加入運行中交易
6. 釋放索引節點, 也就是修改索引節點位圖 (inode bitmap), 然後將此索引節點位圖之區塊緩衝區也加入運行中交易

最後在 ext3_dirty_inode 函式返回前, 呼叫 ext3_journal_stop 結束該處理。這些修改在操作執行的同時, 就被逐步加入運行中交易, 等到交易可以提交時, 資料就會被寫到磁碟。

第二個時機點是在使用者資料寫回磁碟時, 也可能會因此改變檔案系統之架構。舉個例子來說, 我們開啓了一個檔案, 並新增寫入了一個磁碟區塊大小的內容。這裡有一點要注意的是, 在我們增新檔案內容的時候,Linux 並不會即時把資料寫到磁碟, 而是先將資料放在分頁快取中, 因此對於新增的資料, 檔案系統尚未分配磁碟區塊給予該資料。假設經過一段時間後, 分頁快取演算法選出一個分頁要進行寫回, 而這個分頁恰是我們所增新的檔案內容。對於被選出寫回磁碟的分頁, 系統會呼叫 ext3的 ext3_ordered_writepage, 該函式首先會開始一個處理, 並將寫回之資料加入運行中交易, 由於這個新增的資料並沒有對應的磁碟區塊, 因此系統還必須分配磁碟上的一個區塊, 以便之後存放該分頁的資料。如此一來, 檔案系統的架構受到改變, 對於這個改變所需要修改的各個元資料, 也必須在同一個處理中加入運行中交易。這裡修改元資料包含了檔案系統的一個區塊位圖 (block bitmap), 該檔案的索引節點, 以及新增區塊如果不是索引節點 direct 可以直接存下區塊位址 (檔案已經超過 12 個區塊), 而需要透過 indirect(也有可能是 double indirect 或 triple indirect) 存放的話, 那在該分支最後一個區塊也要增新寫入一個區塊的號碼, 也就是說這個區塊也被修改, 這些種種元資料都要加入到運行中交易。最後才會將這個處理結束 (由此可見, 這些有修改的元資料以及寫回的使用者資料被視爲同一個處理之內)。

上述舉例的情況, 檔案系統架構的改變是非即時性的, 因爲新增的使用者資料, 被暫時存放在分頁快取中, 等到該分頁資料被寫回時系統才會分配磁碟區塊, 也就是說等到分頁被寫回時檔案系統的架構才改變。

四、復原程序

當電源中斷時, 透過電池來保護主記憶體上資料, 使資料不會因斷電而遺失, 等待電源恢復後, 透過適當的復原程序將資料寫回磁碟。圖4.1說明資料復原的流程, 分爲五個步驟: 1. 系統復電時, 在 bootloader 底下進行記憶體掃描, 找出需要寫回磁碟之資料。2. 將資料及其相關資訊複製到保留區域。3.掃描磁碟上的日誌檔案, 分析斷電時的狀態。4. 將保留區資料依據各筆資料之資訊, 寫入日誌或檔案系統。5. 掛載 ext3, 透過 ext3的復原程序將日誌中資料寫至檔案系統。

4.1 啓動程序與記憶體解讀

在電源恢復之後, 如果直接將作業系統核心載入主記憶體執行, 則會覆蓋原記憶體上之資料, 因此我們必須搶在載入核心之前就將資料複製到不會被覆蓋的保留區域。啓動程序是一支在作業系統載入前執行的程式, 通常被用來供使用者選擇欲載入之核心, 所以我們可以在啓動程序過程中執行資料搬移的動作 (實作上我們所修改了 Redboot 這支程式) , 然而主記憶體上之資料並非全是需要寫回磁碟的資料, 如果複製整個系統核心資料到保留區域, 那麼則需要準備與作業系統核心佔用量一樣大的記憶體空間。我們在複製資料前就分析記憶體中那些資料需要寫回, 接著將需寫回磁碟之資料複製到保留空間, 複製完畢之後才可以載入作業系統核心並執行之。實驗系統使用了1GB 的主記憶體, 在分析記憶體之後, 我們僅需額外的 128MB 的保留空間, 但這不是保證正確的設定, 最保險的作法是預留一比一的記憶體保留空間 (也就是1GB 保留空間)。

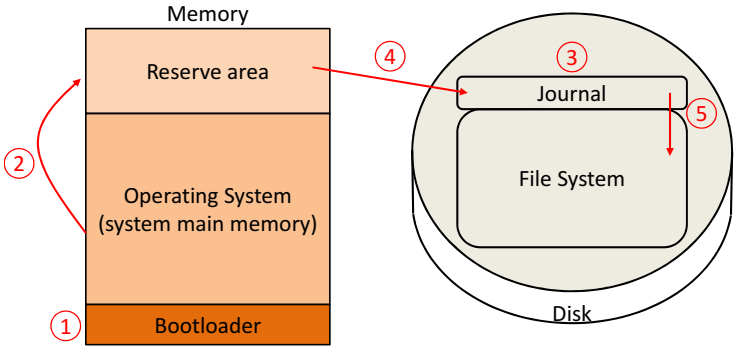


圖 4.1: Recovery procedure

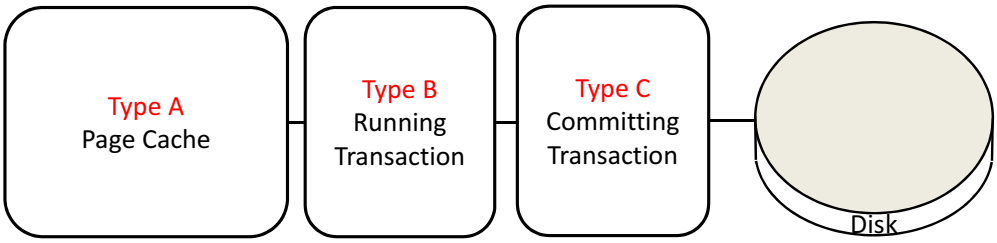


圖 4.2: Three types of dirty data in main memory

記憶體中的髒資料可分爲三種 (如圖4.2), 第一種 (type A) 是在分頁快取中被修改過, 但是系統並未提出寫回的資料。第二種 (type B) 是在分頁快取中被修改, 而且系統已提出寫回需求 (分配給予一個對應的日誌首部), 並加入運行中交易的資料。第三種 (type C) 同樣是被修改, 但是較早之前就已經提出寫回需求, 所屬的交易已經成爲提交中交易之資料。從時間順序上來看, 如果 Type A 的資料中, 有一個分頁被分頁快取替換演算法選爲寫回磁碟的分頁,ext3會將資料加入運行中交易之後, 我們就將資料分類爲 Type B(相關細節請參考3.4一節)。當這個交易開始提交後, 資料則分類爲 Type C。對於 Type A 及 Type B 的資料, 因爲在斷電時尚處於可以修改之狀態, 如果資料在更改中遭到斷電, 將其寫回檔案系統可能會破壞一致性原則, 所以我們的目標是找回第三種資料。

我們可以透過各日誌首部找到所屬之交易, 並由該交易的 tid 辨別是否爲提交中交易。爲了找出哪些緩衝區是上述的第三種資料, 我們必須解讀記憶體遺留的資訊進而找出提交中交易的各個日誌首部。解讀的方法分爲兩種, 第一種是記憶體分析 (memory parsing), 先透過掃描的方式找到記憶體中交易資料結構的雙向鏈結串列指標, 透過指標找到串列中第一個日誌首部作爲起點, 接著依靠各日誌首部的雙向指標一步一步找出全部的日誌首部。

第二種方法是記憶體掃描 (memory scanning)。我們修改了 ext3檔案系統日誌首部之資料結構, 加入了一個4 byte 的欄位作爲簽章 (signature), 並在系統呼叫 journal_add_journal_head 時賦予初值 (當一個分頁快取中的緩衝區資料欲寫回磁碟, 系統就會呼叫此函式, 此函式會分配一個對應的日誌首部並將其加入運行中交易, 我們在分配日誌首部後賦予其簽章值0x55775500)。解讀記憶體方法就是進行記憶體掃描, 找出所有符合0x55775500值之記憶體位址。

就理論上來說, 方法一 (記憶體分析) 透過這些指標就可以一步一步找出提交中交易的各個日誌首部。然而不論是提交中交易或是運行中交易, 這些指標都會經常被更動, 因爲鏈結串列要刪除或插入元素時, 必須修改前後元素指標。當系統無預警斷電時, 無法保證各指標是否處於一個穩定而正確的狀態, 一旦串列中某一個日誌的指標錯誤, 在分析記憶體時就會導致該指標之後續日誌因此而找不到 (記憶體分析之方法在本文第五章實驗比對中可發現, 此方法相較於記憶體掃描 (第二種方法) 約少了0.24%的日誌首部)。因此我們決定採用第二種方法, 記憶體掃描 (memory scanning)。我們透過掃描找出記憶體中所有符合0x55775500的位址 (以4 byte 爲單位, 由低位止掃到高位址), 再掃描過程中一旦發現0x55775500, 即檢查該位址是否爲日誌首部的簽章位址 (因爲記憶體中可能剛好某些位址其數值恰爲簽章之值, 所以我們必須確認該位址的確是日誌首部)。

日誌首部檢查方式並不複雜, 我們檢查其指標是否指向對應緩衝區首部, 再藉由緩衝區首部之指標指回此日誌首部。如圖4.3所示, 我們先從掃描記憶體找到的日誌首部的簽章位址推算出 b_bh 指標的位址 ,其內容即爲緩衝區首部的位址, 找到緩衝區首部後再推算出 b_private 指標的位址, 並依其內容值找回日誌首部, 最後再由日誌首部位址加上簽章欄位的偏移量算出簽章的位址, 檢查是否與一開始簽章位址相同。

有一點要注意的是, 上述兩個指標的內容值是作業系統底下的虛擬記憶體位址, 必須要根據作業系統虛擬記憶體位址與實體記憶體位址的對應關係算出實體位址才找到對應的資料結構, 在本研究使用的開發平台, 其虛擬位址與實體位址之差值恰爲1GB, 但這虛擬到實體位址的對應關係可能會因爲作業系統與硬體架構不同而有所差異, 必要時須查詢作業系統的分頁表 (page table)。

4.2 資料複製

在啓動程式中可能無法直接讀寫外部儲存裝置 (例如磁碟), 因此我們將資料先複製到記憶體保留區, 接著才載入作業系統核心。一般而言, 緩衝區首部的 b_data 指標就是資料所在位置, 但是如果在交易提交過程中, 緩衝區的資料又被修改, 此時 ext3會即時複製一份 read-only

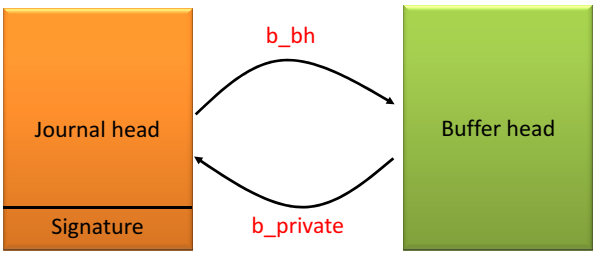


圖 4.3: Check journal head structure

snapshot, 以供提交所需。這一塊資料位址由日誌首部的 b_forzen_data 指標記下, 並在提交過程中將該資料寫入日誌, 而原本緩衝區的資料則可由後續修改需求進行修改。因此我們在復原程序複製資料到保留區域的時候, 若找到該日誌首部具有 b_frozen_data, 那麼我們應該複製這一塊資料至保留區, 而不是原本的緩衝區資料。

由於本研究的目標是將提交中交易的資料 (Type C) 救回, 故僅只將複製提交中交易的資料到保留區。我們可以透過日誌首部的 b_transaction 指標看出該日誌首部所屬之交易資料結構的位址, 並與 journal 結構中 j_committing_transaction 指標 (如圖3.8所示) 比較, 檢查二者是否相同, 若相同則代表該日誌首部之資料爲提交中交易之資料。若欲將所有贓資料都寫回, 可以透過掃描的方式找出所有緩衝區首部並檢查資料是否爲髒資料 (檢查緩衝區首部的 b_state 欄位, 這是一個 flag, 其中 BH_Dirty 位 (第二位) 若被設置則表示該資料爲髒資料), 若是髒資料且緩衝區首部 b_private 欄位爲空指標則屬於 Type A, 若緩衝區首部 b_private 欄位非空指標 (指向某個日誌首部), 需再由日誌首部判斷該資料屬於何交易, 判斷方式即上述 Type C 之判斷方式。將資料分成三類之後, 再將其複製到保留區域。要注意的是, 強制將 Type A 與 Type B 資料寫回磁碟可能導致檔案系統損壞 (違反檔案系統一致性)。

總結上述所得, 我們透過日誌首部簽章找到日誌首部, 再透過指標找到緩衝區首部, 再找到緩衝區 (即資料), 最後將緩衝區複製到保留的區域, 除此之外還必須複製一些相關資訊供資料寫回程序所需, 如圖4.4所示, 除了緩衝區之外我們要須將裝置代碼 (磁區的代碼)、磁區區塊號碼 (代表資料要寫回磁區的哪一個區塊)、緩衝區資料性質 (使用者資料或是元資料)、日誌首部所屬之交易序號以及資料複製所在的保留區記憶體位址等資訊, 也對應各緩衝區資料一併寫入記憶體保留區。

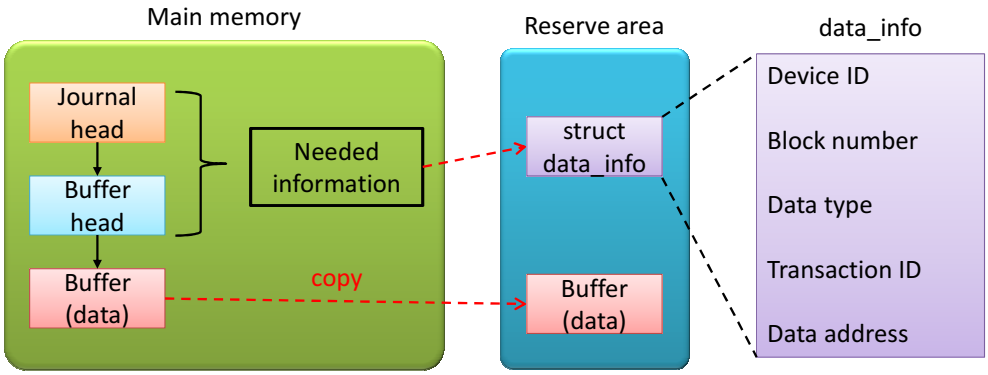


圖 4.4: Copy buffer and corresponding information to reserve area

到目前爲止包含記憶體掃描以及資料複製的種種行爲, 都是在啓動程式底下完成, 透過啓動程式所存取的記憶體位址爲實體位址。我們將資料寫在指定的位址, 之後復原時透過使用者程式去將資料讀出, 然而使用者程式僅能看到使用者空間 (user space) 的虛擬位址, 所以要存取這些實體位址資料須透過記憶體映射, 將這一段實體記憶體映射到使用者空間方可存取, 關於這點將在下一節說明。

4.3 資料寫回

4.3.1 記憶體映射

保留區記憶體上的資料必須在檔案系統掛載之前寫回, 以避免掛載後有新資料寫入, 因此我們必須在作業系統將驅動程式運行之後, 掛載檔案系統之前進行資料寫回的程序。我們寫了一支使用者程式來將資料寫回磁碟, 但由於資料暫存在保留區之中, 因此我們需要透過記憶體映射 (memory mapping) 函式:mmap 將保留區映至使用者空間。首先呼叫 open 函式開啓記憶體裝置/dev/mem(在 Linux 底下每一個裝置都被視爲一個檔案, 在 dev 資料夾底下)。接著呼叫 mmap 函式, 參數傳入欲映射之實體記憶體起始位址以及映射長度, 該函式回傳該程式使用者空間的一個虛擬位址對齊映射的起點。如圖4.5所示, 假設我們將實體記憶體保留區域0x50000000到0x7FFFFFFF 這一段記憶體, 透過 mmap 函式映射到使用者空間中, 並得到回傳虛擬位置爲0xC0000000, 則映射後我們可以透過虛擬位址 0xC0000000 到0xEFFFFFFF 這一段空間操作實體記憶體保留區域。

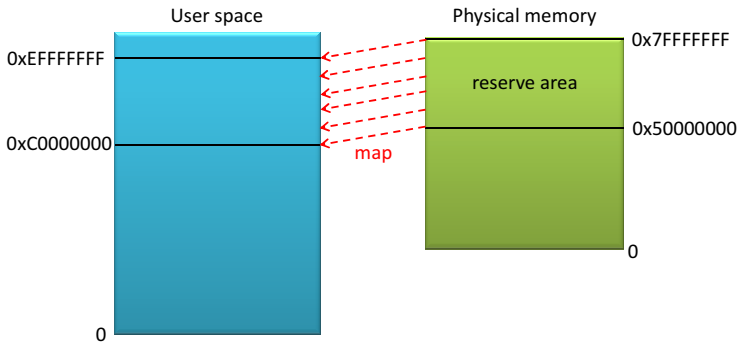


圖 4.5: Memory mapping

4.3.2 日誌檔案讀取

如果直接將保留區資料寫入檔案系統會造成一些問題, 以 ext3 ordered mode 爲例, 元資料在提交時會先寫入日誌, 提交完畢之後才寫入檔案系統, 如果我們直接將保留區上面的資料寫入檔案系統, 則在掛載檔案系統時,ext3復原程序會將日誌中完成提交動作之交易資料寫入檔案系統, 可能會造成資料覆蓋的情況。然而記憶體保留區所保留之資料是斷電當下的記憶體資料, 是提交中交易底下所串連之資料 (較新的資料), 而日誌中是已經完成提交的交易資料, 這些資料是比較舊的版本。一旦發生覆蓋的情況, 則最新資料即會被舊資料所覆蓋。爲了避免最新資料被覆蓋, 我們必須將保留區中的元資料寫入磁碟日誌, 銜接在舊的交易之後, 使檔案系統掛載時, 復原程序會依照交易提交順序將元資料寫入檔案系統。而保留區中使用者資料則可以直接寫回檔案系統 (因爲在 ordered mode 底下, 使用者資料並不會寫入日誌, 故不會於 ext3復原程序進行時發生使用者資料覆蓋的問題)。

爲了能夠正確無誤的銜接日誌中舊有資料, 我們必須能夠分析日誌目前狀態。雖然檔案系統尚未掛載, 但我們依然可以透過存取區塊裝置 (block device) 的方式存取整個磁區, 首先呼叫 open 函式開啓欲存取的磁區 (位於/dev 之下), 接者透過 seek64函式跳到要存取的位置, 然後透過 read 函式讀取指定的長度。要注意的是, 這些函式的參數都是以位元組爲單位, 如果想要讀某一個區塊, 必須將區塊號碼及區塊大小二數相乘, 將資料由區塊單位換算成位元組單位, 才能進行讀取。

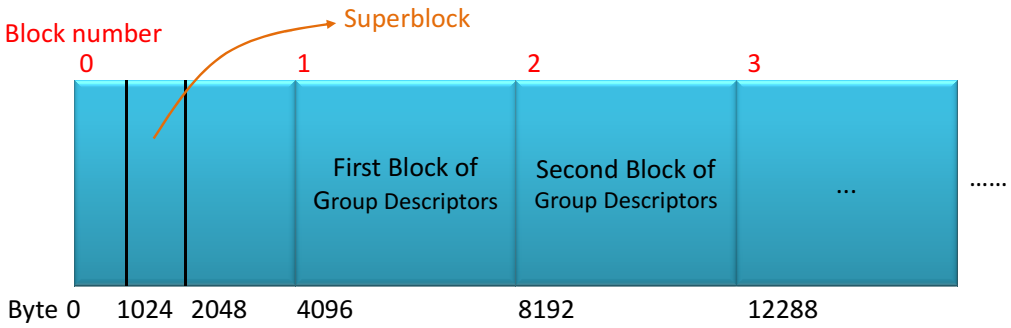


圖 4.6: The position of superblock and group desciptors in first block group

在 ext3底下, 所謂的日誌其實是一個特殊的檔案, 它不屬於任何一個目錄之下, 也不具有目錄項 (dentry) 要找到 ext3的日誌必須先從磁區的超級區塊 (superblock) 中讀出日誌的索引節點號碼 (inode number), 再透過 ext3的 (組描述符)Group Descriptor 找到索引節點表 (inode table)。從索引節點表中讀出日誌的索引節點後, 即可掌握日誌檔案在磁碟上的各個資料區塊, 詳細說明如下:

Ext3檔案系統將整個磁區分成許多個區塊組 (block group), 每一個區塊組都含有超級區塊跟全部的組描述符, 我們只需要讀出第一個區塊組的超級區塊及組描述符即可。圖4.6表示在區塊大小爲4KB 的磁區底下, 超級區塊與組描述符的位址, 上標表示區塊號碼, 下標則表示位元組位址。第一個區塊組的超級區塊位於該磁區第1024byte 到2047byte, 大小恰爲1KB, 而 Group descriptor 則是從超級區塊後的下一個區塊開始。一個磁區的區塊大小可由超級塊的 s_log_block_size 欄位看出來, 這個數值表示2的冪次方, 並以10爲基準。如果該值爲0, 則區塊大小爲2^{10+0}, 故爲1024位元組, 若該值爲2, 則區塊大小爲2^{10+2}, 即爲4096位元組。算出區塊大小後就可以找出組描述符的起始位元組, 圖4.6是描述4KB 區塊大小的磁區之分布情形 ,由於組描述符是從超級區塊所在位置的下一個區塊開始, 所以從2048到4095位元組不會被使用。找到組描述符之後, 再根據超級塊所記錄的日誌索引節點號碼即可以找出索引節點, 計算方式如下: 先從超級塊讀出一個區塊群包含索引節點之數量 (s_inodes_per_group), 接著將日誌索引節點號碼除以該數量, 其商數即爲該節點所在之區塊組, 餘數即代表該組中第幾個索引節點。例如日誌索引節點編號爲1057, 且每個區塊組具有100個索引節點, 則將1057除以100得其商爲10, 餘爲57, 代表此索引節點位於第10區塊組的索引節點表中第57個索引節點。我們透過組描述符得到第10個區塊組的索引節點表之區塊號碼, 再讀出該表中第57個索引節點即可。

4.3.3 日誌檔案分析與資料寫入

掌握了日誌索引節點後即可進行日誌分析, 首要步驟是讀出日誌超級區塊 (日誌第一個區塊), 這裡有一點值得注意的是, 日誌中所有訊息都是以 big endian 寫入, 若是使用 little endian 的處理器讀取, 需要進行數值轉換。從日誌超級區塊提供的資訊找出目前日誌中有效資料 (尚未寫回檔案系統之交易) 是從哪個區塊開始 (日誌超級區塊的 s_start 欄位記錄著起始區塊的邏輯號碼, 若此值爲0表示日誌無有效資料, 不需要進行復原程序)。接著從該區塊開始依序掃描日誌中各交易, 直到最後一個完整的交易爲止, 然後我們在完整交易後接上保留區的資料。

圖4.7(a1)、(a2) 表示原本日誌中的情形, 圖 (a1) 中日誌最後一個交易是一個完整的交易 (具有提交區塊), 這種情況代表提交中交易之元資料尚未寫入日誌就遭斷電, 日誌中看不到任何提交中交易的資料。圖 (a2) 中最後一個交易並不完整 (沒有提交區塊), 這個情況代表提交中交易已經開始將元資料寫入日誌, 但由於尚未完成提交過程就遭斷電, 所以不具有提交區塊。不論是上述哪一種情形, 該提交中交易都還在提交階段, 因此各緩衝區資料都會保留在記憶體內, 也就是說這些資料可以在啓動程式底下被我們找到且複製到保留區域。所以如果日誌中是如圖 (a2) 的情形, 我們在銜接資料時後就直接覆蓋不完整的交易所占用的區塊, 將資料銜接在最後一個完整交易之後。

如圖4.7所示, 我們從 s_start 開始掃描各交易, 找出最後的完整交易位置。接著我們仿造交易提交時寫入日誌的各種資料區塊 (描述區塊、提交區塊等等), 加入一個僞造的交易到日誌中, 該僞造交易包含提交中交易欲寫入日誌之各緩衝區資料 (即保留區中的元資料區塊), 這一個新加入的僞造交易銜接在最後一個「完整」的交易之後, 如圖 (b)。如此一來,ext3掛載時進行的復原程序將認爲我們仿造的交易也是斷電前提交完成的交易之一, 而將其寫回檔案系統, 如此一來就可以維持正確的資料寫入順序, 避免同一區塊之新資料被舊資料覆蓋的問題。

最後一步我們掛載 ext3檔案系統, 此時 ext3會判斷是否需要進行日誌資料的復原程序。若日誌超級區塊的 s_start 值爲0, 表示日誌中並無有效資料, 不需要進行復原程序; 若其值非0, 則會從該值所記錄的區塊開始, 進行交易資料復原程序。要注意的就是, 如果斷電當下日誌中恰無資料需要回復 (s_start 爲0), 而在我們加入僞造交易到日誌中之後必須將該值改爲新加入的

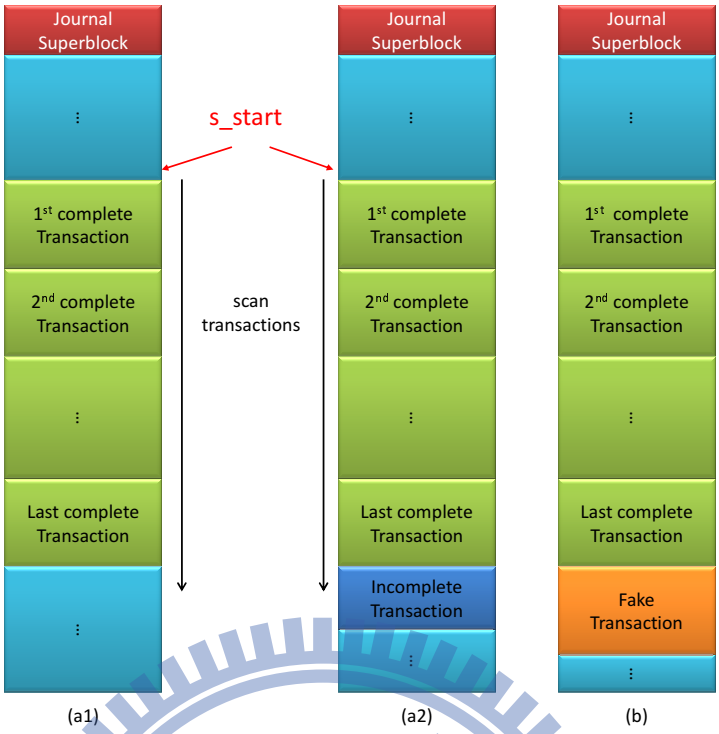


圖 4.7: Manipulate the in-disk journal

交易起始區塊號碼 (新加入交易起始區塊號碼應爲 1), 如此 ext3 才會進行復原程序。

五、實驗

本研究設計了三個實驗, 第一個實驗是在虛擬機器 (virtual machine) 上執行, 我們複製了一些 Linux 的程式碼檔案到指定目標資料夾, 並在複製指令後將虛擬機器強制關機, 事後觀察目標資料夾檔案的情況。這個實驗是本研究的前置實驗, 目的是爲了觀察斷電對檔案造成的影響。第二個實驗是有關記憶體解讀的實驗, 我們以本實驗比較記憶體分析與記憶體掃描兩種方法的效果。第三個實驗是本研究的成效比對實驗, 我們比較原 ext3復原程序與加入我們設計方法的復原程序兩者之間效果差異。

5.1 實驗一

本實驗在 VMware 上執行, 安裝的作業系統是 Ubuntu Linux 2.6.37版本。我們用 copy 指令複製了91MB 的檔案資料 (共計14856個檔案及585個資料夾), 並在提示符號返回之後將 VM 強制關機。這個實驗是本研究的前置實驗, 目的是觀察無預警斷電對檔案系統資料寫入會造成什麼樣的影響。我們在提示符號返回後每個五秒做一次中斷實驗, 並在系統重開之後統計目標資料夾底下檔案的情形。並觀察目標資料夾下檔案的狀況。其結果如圖5.1所示, 縱座標表示檔案的數量, 橫坐標表示時間 (從提示符號返回開始), 單位是秒。我們將 copy 指令所複製的檔案分三種類型統計:

1. 遺失 (lost) 的檔案

   若一檔案屬於來源資料夾內, 卻未被寫入目標資料夾下, 則被歸類爲遺失檔案

2. 損壞 (corruption) 的檔案

   檔案已由 copy 指令複製至目標資料夾下, 但其內容跟來源資料夾之檔案相異, 則歸類爲損壞的檔案

3. 正確 (correct) 的檔案

   目標資料夾下之檔案與來源資料夾之檔案內容完全相同

檔案遺失原因是該檔案之元資料尚未寫入檔案系統, 因此無法從資料夾底下看到檔案。檔案損壞的原因是元資料已經寫入檔案系統, 但是資料尚未寫完, 因此檔案內容與來源並不相同。

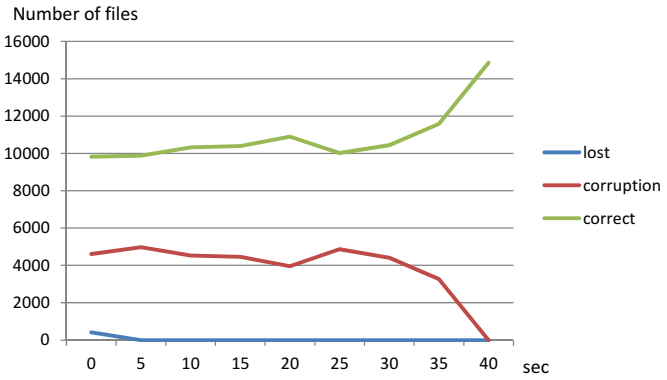


圖 5.1: Experimental result 1

由實驗結果可以發現, 若在提示符號返回+5秒內強制關機, 某些檔案無法在目標資料夾下看到 (遺失), 也就是說這些檔案之元資料並未寫入檔案系統。而+5秒之後才強制關機的話, 所以檔案皆可在目標資料夾下看到, 但是有三、四千個檔案的內容並不正確, 這個情況直到+40秒之後才結束, 也就是說 copy 指令直到+40秒之後才將所有資料寫回。

5.2 實驗二

本實驗是關於記憶體解讀的實驗, 目的是比較兩種不同的記憶體解讀方式何者較適合。我們在開發機 VessRAID 1841i 上進行本實驗, 這是一台網路硬碟 (NAS), 使用 redboot 啓動程式, 作業系統是 Linux 2.6.17的版本, 檔案系統爲 Ext3。開發機使用電池備援記憶體, 內含 2GB 的 SDRAM 以及2顆18650的鋰電池, 每顆電壓3.6V,2200mAh。我們在 Linux 底下執行 copy 指令, 並於指令進行中關掉電源, 當重新開機時, 於啓動程式 (redboot) 底下進行記憶體解讀。

我們用兩種方法解讀記憶體, 試圖找出交易結構底下的日誌首部資料結構, 觀察兩個方法各找出日誌首部之數量。第一種方法是記憶體分析, 我們透過交易結構之指標找到日誌首部雙向鏈結串列之位址, 並分別透過兩個方向指標依序尋找日誌首部, 我們將找到的日誌首部各自累加, 直到返回第一個日誌首部或是指標指到非日誌首部結構的位址即終止。第二種方法是記憶體掃描, 我們在日誌首部中增加了一個4byte 的簽章 (signature), 並於記憶體中掃描個簽章 (需透過前述檢查步驟確認該位址確爲日誌首部), 藉此統計日誌首部之數量。

本實驗結果, 透過記憶體分析的方法, 沿 next 指標之方向搜尋雙向鏈結串列共得18608個日誌首部; 沿 prev 指標搜尋結果得18540個日誌首部; 採用記憶體分析之方法所得之日誌首部18619個日誌首部。

由實驗結果我們可以發現, 記憶體分析的方法並不可靠, 因爲一個完美的雙向鏈結串列結構理論上而言, 不管從 next 或是 prev 方向尋找, 應該會經過一樣多的日誌首部數量返回原點, 但實驗結果統計出來卻有出入, 且兩個數量皆小於記憶體掃描所得之日誌首部數量, 以平均來看約少了0.24

5.3 實驗三

此實驗是本研究之成效比對實驗, 目的是比較我們提出的復原程序與原 ext3復原程序可以多救回多少資料。我們在 VessRAID 1841i 上進行實驗, 該機器內含 Intel 81348處理器,2GB 的 SDRAM, 其中 1GB 作爲作業系統所使用, 並以兩顆鋰電池做記憶體備用電源。軟體方面使用 redboot 啓動程式,Linux 2.6.17作業系統,Ext3檔案系統 ordered 模式。

本實驗僅復原提交中交易所含之資料, 對於運行中交易以及分頁快取中贓資料不予以寫回, 因爲這些資料在斷電當下可能包含某些操作之部分已修改資料, 若將其寫回會違反原子性原則, 寫回後會影響檔案系統一致性。我們觀察復原了提交中交易可以多救回多少資料。

實驗流程如下:

1. 在 Linux 底下進行檔案 copy 的指令, 並於提示符號返回後, 下一個交易提交時重啓電源
2. 在 redboot 底下進行記憶體掃描
3. 將提交中交易之資料及其相關資訊複製到保留區域
4. 載入作業系統核心

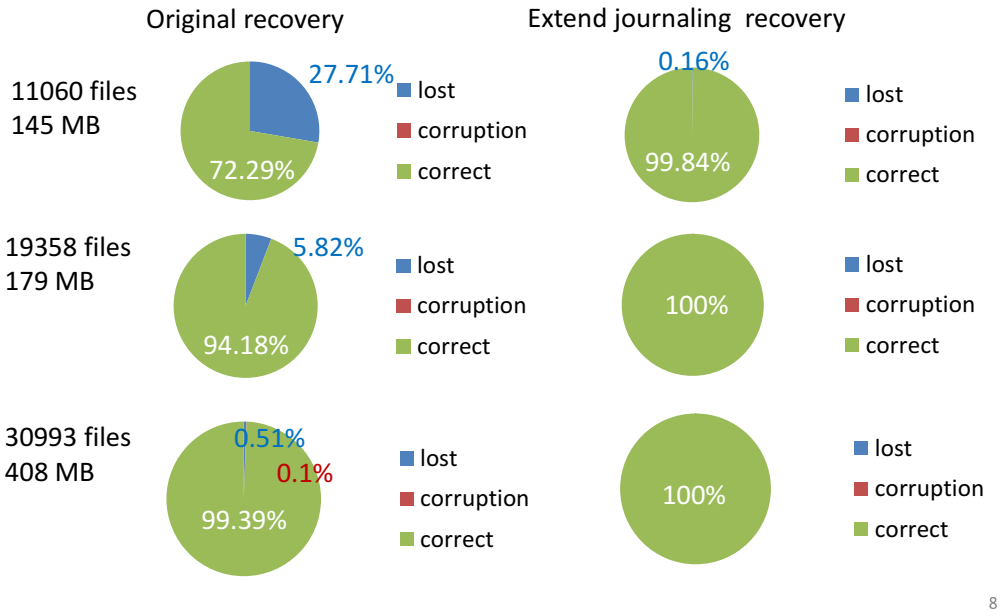


圖 5.2: Experimental result 3

5. 將記憶體保留區之使用者資料寫到檔案系統, 並將元資料包裝成一個交易提交至檔案系統

6. 掛載檔案系統 Ext3, 透過 Ext3復原程序將元數據寫到檔案系統

7. 統計目標資料夾底下檔案情況

我們進行了三組實驗, 分別複製了 11060,19358以及30993個檔案, 實驗結果如圖5.2所示, 左邊爲原 ext3復原程序執行後, 由目標資料夾下統計檔案之數量; 右邊爲本研究所提出之復原程序執行後, 目標資料夾底下檔案室數量。其中三種檔案統計類型與實驗一定義相同。由實驗結果可以看出, 本研究所提出之方法確實有效減少了檔案的損失, 不過其效果超出預期, 與實驗一所得結論並不相符。實驗一我們得知 copy 指令待提示符號返回+40秒後才將所有資料寫完, 然而本實驗在提示符號返回後第一個交易時, 就幾乎將所有資料寫完 (運行中交易預設5秒會進行提交), 兩實驗表示了不同的資料寫回行爲。我們認爲此部分差異是因爲 VessRIAD 之作業系統核心已由該產品工程師修改, 其詳細情形尚需洽詢工程師。

六、未來研究

本研究到目前爲止, 可以將主記憶體中提交中交易的資料復原, 我們最終目標是對記憶體中運行中交易及分頁快取中髒資料也能夠復原。由於斷電當下這兩部分資料可能正被修改, 如果直接將資料寫回, 可能會導致檔案系統不一致。比如說在 ext3檔案系統下進行檔案刪除的操作, 這是一個原子操作 (Atomic operation),Ext3本身要進行幾個底等級的操作, 包含刪除該黨之 dentry、修改 inode 位元圖、修改區塊位元圖等等。這些動作都完成修改之後, 才算是正確的刪除了該檔案, 如果只修改其中一部分資料就遭到斷電, 而回復程序將這部分資料寫入檔案系統, 會造成檔案系統不一致。因此, 若要將這兩部分資料復原須考慮以原子操作爲單位, 也就是說對於不完整的原子操對應之資料, 不應該將其復原, 如此才能盡可能的救回資料且不影響檔案系統一致性。

七、結論

傳統檔案系統遇到斷電時, 會對資料造成影響, 除了遺失記憶體中未寫回資料外, 可能還會影響到檔案系統一致性。使用 UPS 保護整個系統可以確保資料寫回並安全關機, 但是對於大型儲存設備, USP所需要成本過高。我們提出一個方法, 在斷電時透過電池作備用電源保護主記憶體上之資料 (或是使用非揮發性主記憶體), 等待電源恢復, 並透過特定之流程, 配合檔案系統特性將資料寫回磁碟, 同時維持檔案系統一致性以及寫操作之順序性。目前爲止, 我們正確地寫回了提交中交易之資料。未來我們希望也能復原運行中交易以及其他分頁快取中所含之贓資料, 對於這兩種資料寫回, 必須更進一步分析各原子操作所修改之資料才能在不影響檔案系統一致性的前提下, 盡量將資料復原。

參考文獻

[1] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. *ACM SIGPLAN Notices*, 27(9):10–22, 1992.

[2] A.I.A. Wang, G. Kuenning, P. Reiher, and G. Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *ACM Transactions on Storage (TOS)*, 2(3):309–348, 2006.

[3] S.K. Cha, J.H. Park, and B.D. Park. Xmas: an extensible main-memory storage system. In *Proceedings of the sixth international conference on Information and knowledge management*, pages 356–362. ACM, 1997.

[4] M.K. Qureshi, V. Srinivasan, and J.A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 24–33. ACM, 2009.

[5] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 14–23. ACM, 2009.

[6] S.C. Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.