Spring 2019 Advanced Operating System

Midterm

DUE DATE: 4/26/2019, 14:20

學號：R07944058　系級：網媒碩一　姓名：陳鵬宇

Problem 1. (20 %)

Lazy consistency model

- pros：
  1. 更新資料會延遲到下一次 acquire，只有在某個 process acquire 某個 data 的 lock 時，才需要向持有 lock 的 process "pull"，overhead 較小。
  2. 如果相同的 process 需要同一份 lock，不需要執行任何動作。
  3. 有利於循環中的 acquire-release pairs。
- cons：如果副本在將更新發送到其他副本之前崩潰，則可能會丟失更新。延遲複製也不適用於需要嚴格同步（例如：視訊會議）的情況。
- 假設共有 k 個 processes：$P_1, \ldots, P_k$，彼此共享了 n 個變數，若其中 P_i $(1 \leq i \leq k)$ 要訪問其中一個變數，他會向持有該變數 lock 的 P_j $(1 \leq j \leq k, j \neq i)$ 給一個 acquire 的 request，P_j 再 release lock 給 P_i，所以 # of messages 取決於有幾個 processes 要訪問幾次變數，每訪問一次為一個 acquire-release pair，需要傳送 2 次 messages。
- 因為 lazy consistency model 需要有 pull 的動作，而不是一個 process 訪問完變數後就固定 release lock 給其它所有人，故 programmability 較低，相對較難實作。
- # of synchronization 取決於 # of acquire requests，每當一個 process acquires 時，就會與持有該 lock 的 process 同步，達成 cache coherence。

Release consistency model

- pros：
  1. 不需 block process 以便完成 coherence actions，我們只需要在 release point 時 block 住，確保在 release 之前的所有操作到這個時間點都已經完成。簡單來說：退出 critical region 時，共享記憶體將保持一致。
  2. 盡可能地隱藏越多寫入延遲來提高性能。
- cons：
  1. 一但某個 process，acquire lock → data access → release lock 後，他必需要對所有其他人廣播一次「我已經 release 某變數的 lock 了」，然而有些 process 或許根本沒有 access 該變數的需求，就變成多餘的浪費了。
  2. 硬體複雜性增加和更複雜的程式模型。
- 假設共有 k 個 processes：$P_1, \ldots, P_k$，彼此共享了 n 個變數，若其中 P_i $(1 \leq i \leq k)$ 要訪問其中一個變數，他會：

1. acquire lock → data access → release lock
2. 在 release lock 時，他要跟其它 $k-1$ 個人傳送「我已經 released lock」的 message，所以 # of messages 取決於總共 release 了幾次 lock，每 release 一次，需要 $k-1$ messages。

- 在 critical section 時，寫和讀都不需要按照順序，並且只要在 release 時，廣播更新其它人手中的 copy，programmability 較 lazy consistency model 容易。
- 每當有 process release 某變數的 lock，他就會 push 給大家確保 cache coherence，所以 # of synchronization 等於 releases 的次數。

Weak consistency model

- pros：

1. Local cache 僅在任務執行的某些點上保持 coherency，從而減少了開銷。
2. Local cache 可以平行地保持 coherency。
3. process 不用每寫一次就將結果廣播出去，可以累積一陣子後，再一次廣播，這在其它 processes 還不需要那麼立即知道我們的更新時，能提升不少性能。
4. Weakly consistent invalidating protocol 也可以平行地使遠程拷貝無效，這些優化可以進一步提高性能。

- cons：

1. 因為我們是一次廣播「一定的量」給其它 processes，當某個 process 需要的資訊只有其中一部分時，他仍然必需等到其它資訊都被同步完成，這樣一來還不如事先同步那些資訊，兩者必需取得平衡。
2. 當 process 中有某個演算法需要 explicit commands，以便在能夠在適當的點執行任務，這時 weak consistency model 可能無法滿足。

- 同步變數（s）用於將所有 writes 廣播給所有人，並針對分散式系統中其他位置發生的 global data 更新執行本地更新。# of messages 取決於共有幾個 s，去告訴大家：「我們要同步了」，假設共有 k 個 processes：$P_1, \ldots, P_k$，在每一個 critical section（s 與 s 之間）寫的值就必跟其它人做更新，最多會有 k 個人都做改動，每個人都寫到 global（k 次 messages），大家再根據 global 把值存下來（k 次 messages），共需 $2k$ messages。
- programmablitiy 最容易，在執行所有先前對 synchronization variables 的訪問前，不允許執行 data access 訪問（讀或寫）。
- # of synchronization = # of synchronization variables。

Problem 2. (20%)

Micro kernel

Micro kernel 小但高度靈活，且可擴性、客制性、容錯性等都不錯。每個 kernel 都掌管了某件事，因此 customization 維度滿高。每個 kernel（服務）彼此獨立，可以減少系統之間的耦合度，也較容易實作和除錯。當一個 kernel 崩潰時，也不會影響到其它 kernel 的運作。我們只要重啓此 kernel，就能讓 OS 恢復運作，同時 OS 也可以新增、移除某些 kernel，靈活性高。但需要有 IPC 的機制，耗費的資源比簡單的 call function 多，又會涉及到 kernel space 和 user space 的 context switch，故效能可能差了點。

Unikernel

Unikernel 沒有了 user space 和 kernel space 分別，只有一連續地址，因此 Unikernel 一次只能執行一個應用程式，所有軟體、硬體的邏輯，都在一個空間之中，這樣的好處是能達到非常高度的 customization，同時 image 也非常小，一但 image 小，啓動的速度就很快，都在毫秒級別，比起傳統 kernel 提升的速度是很驚人的。假設我只需要 DNS 的服務，那我就只需要某個 DNS 的 image，不僅輕量化、customization 也好。Unikernel 最後產物為 kernel image，可以在 hypervisor 等環境上執行，傳統方式發佈的可能是一整個應用程式，再好一點的話為一個 container image 而 Unikernel 則是一高度客製化的 kernel。

Scenario

Micro kernel 只提供最簡單的服務，例如：IPC、scheduling，剩下的都交給 server 去處理，server 類似於傳統的 monolithic kernel，例如：Linux。以前 server 都是直接跟硬體溝通，現在多加了一層 kernel (L4)，它處理所有的 hardware interrupt，然後將此 message 傳給 server，server 再傳給 software。可能發生以下場景：

- 瀏覽器 process 需要一個 network packet。收到 packet 以後，L4 作出 interrupt，傳給 server，server 再與 L4 通過 IPC 請求 packet 內容，L4 會 map 到 L4Linux 的 address space，server 再傳給 software (e.g. Firefox)。
- 由於記憶體容量有限，有些內容會暫時 swap 到硬碟，application 想用的時候發現不在了，就產生 page fault，首先處理 interrupt 的是 kernel，kernel sends IPC 給 L4Linux，L4Linux 從自己管理的 physical memory 裡面分一個 page 出來給 application，並更新自己的 shadow page table (不是真正的 page table)。L4Linux 產生一個 return value 告訴 L4 已經完成。L4 再更新他真實的 page table。

Unikernel 在開發過程中，開發者可以假設自己在傳統的 OS 上進行開發，而所有內核相關的功能，暫且由開發者的 OS 提供。在測試環境中，大部分 Unikernel 的實作方式是：

- 將應用程式的程式碼與需要的內核模組建成 Unikernel 後，再將其跑在一個傳統的 OS 上，利用傳統 OS 上的工具來測試 Unikernel。以 Rumprun 為例，它可以通過 KVM/QEMU 來運行一個 Rumprun Unikernel VM，隨後用 Host OS 上的 GDB 來對其進行測試。

Problem 3. (30%)

假設共有 N 個 nodes，首先初始化 $node_0, node_1, \ldots, node_{N-1}$ 的 `PageTable`，如果自己是該 page 的 owner (`isOwner = true`)，那 `probOwner` 就設成自己；否則，就指向下一個 node ($node_{i+1 \mod N}$)，以下是 $node_i$ 的 `PageTable` 示意圖：

#page	probOwner	copyset	access	lock	isOwner
0	$node_{(i+1) \mod N}$	{}	NIL	false	false
1	$node_i$	{}	NIL	false	true
⋮	⋮	⋮	⋮	⋮	⋮
PAGETABLESIZE − 1					

每個 server 在執行前都必需 `lock`，執行結束時 release lock (`unlock`)，先給定 `lock`、`unlock` 和 `invalidate` 的 pseudo code，在這裡 `lock` 的參數指的是將 `PageTable` 第 p 個 page 的 `lock` 屬性設成 `true`，這樣若其它 process 也同時要 access 同一個 page 時，就會被加到 queue 中。

```
lock(PageTable[p].lock)
    while (true)
        if (testAndSet the lock bit to true) break
        if (fail) queue this process

unlock(PageTable[p].lock)
    PageTable[p].lock = false
    if (there is a process waiting on the lock) resume that process

invalidate(p, copyset)
    for node in copyset
        send an invalidation request "your p is invalid" to node
```

Read Fault Handler

當有某 node 對 PageTable read request 時，產生 page fault，我們首先對該 page p 可能的 owner 拿 request，並將 probOwner 設為該 node、access 設成 read 屬性。

```
lock(PageTable[p].lock)
ask PageTable[p].probOwner for read access to p
PageTable[p].probOwner = replyNode
PageTable[p].access = read
unlock(PageTable[p].lock)
```

Read Server

當確定沒有 node 也競爭同一 page 時，就將新的 requestNode 加入 copyset，將 page p 傳給 requestNode；否則，就告訴 requestNode，可能擁有此 page p 的 node 是誰。

```
lock(PageTable[p].lock)
if PageTable[p].access != NIL
    PageTable[p].copyset = PageTable[p].copyset + { requestNode }
    PageTable[p].access = read
    send p to requestNode
else
    forward request to PageTable[p].probOwner
    PageTable[p].probOwner = requestNode
unlock(PageTable[p].lock)
```

Write Fault Handler

和 Read Fault Handler 很像，都是要權限，這回是要 write 的權限，這裡多了一個步驟 invalidate()，一但這裡的值被寫的話，其它人手上的 copy 就都會是 invalid 的。

```
lock(PageTable[p].lock)
ask PageTable[p].probOwner for write access to p
invalidate(p, PageTable[p].copyset)
PageTable[p].probOwner = self
PageTable[p].access = write
```

```
PageTable[p].copyset = {}
unlock(PageTable[p].lock)
```

Write Server

當 node 訪問的該 PageTable 終於是 owner 了，我們就將 acesss 設成 NIL，並且將 p 和 copyset 的資訊轉交給 requestNode，最後再將 probOwner 設成 requestNode。

否則，就告訴 requestNode，可能擁有此 page p 的 node 是誰，由於 requestNode 最後會變成 page p 的 owner，所以我們在這裡將 probOwner 設為 requestNode。

```
lock(PageTable[p].lock)
if (PageTable[p].isOwner)
    PageTable[p].access = NIL
    send p and PageTable[p].copyset to requestNode
    PageTable[p].probOwner = requestNode
else
    forward request to PageTable[p].probOwner
    PageTable[p].probOwner = requestNode
unlock(PageTable[p].lock)
```

Invalidate Server

```
if PageTable[p].access != NIL
    invalidate(p, copyset)
    PageTable[p].access = NIL
    PageTable[p].probOwner = requestNode
    PageTable[p].copyset = {}
```

Fail Recovery

假設 node[j] 掛掉，我們可以運用其它 nodes 的 PageTable 找回 node[j] 的 PageTable，一律從 node[0] || node[1] 開始像 linked list 那樣往下找，不停詢問 currNode 是否為該 page p 的 owner。迴圈結束後，看 visited 是不是除了 visited[j] 外都被 label 成 true 了，如果是的話，node[j] 為該 page p 的 owner，最後再 assign 值到正確 field。

```
for p in PAGETABLESIZE
    let visited[0..N - 1] be an array with all falses
    if (j == 0) let currIndex = 1
    else let currIndex = 0
    currNode = node[currIndex]
    copyset = {}
    while (!currNode.PageTable[p].isOwner)
        visited[currIndex] = true
        currNode = currNode.PageTable[p].probOwner
        if (currNode == node[j])
            currIndex = the smallest index k that is not visited
            currNode = node[currIndex]
        else currIndex = currNode.index
    if visited[0..N - 1] are labeled true except visited[j]
```

```
    node[j].PageTable[p].isOwner = true
else node[j].PageTable[p].isOwner = fasle
currNode.PageTable[p].probOwner = currNode
currNode.PageTable[p].copyset = copyset
```

Problem 4. (20%)

通常記憶體搬動的 overhead 都滿大的，而像 VNF 也是希望將 core 的一些 services 回歸到 edge 端上計算，這樣就不用在 user 和 server 之間不停搬動記憶體，造成 overhead。

Active message，Paper 中提到一個例子為 NIC，NIC 網絡接口包含 5 個輸入和 5 個輸出暫存器，用於 set-up 和 consume messages。發送 message 後，輸出暫存器不馬上清掉他的 buffer，他仍然保持其值，當他又需要發送相同的 message 時，就可以直接用 register 的值。data 可以從輸入移動到輸出暫存器，以便在回覆或轉發 message 時重用 data，這樣可以降低搬動 memory 的 overhead。

Active Message 的低開銷使小 message 更具吸引力，這簡化了開發並減少了網路擁塞，對於小 message，網絡本身的緩衝通常是足夠的。另外還有一個做法叫做 Message coalescing，他將一小塊一小塊 messages 合併成一大塊 message，這種作法能有效地發送相同數量的資料，同時減少每份 message 在 service providers 和 users 間搬動的開銷。

Problem 5. (10%)

(a) 三個 nodes 可能因果相關的 writes 對所有 processes 順序一致，且看到因果關係上的順序是一樣的，所以是 causal consistency。

(b) 要達成 sequential consistency，所有 processes 看到的順序要是一樣的，但不一定要是真實的順序，以下舉其中一例：

Node N_1	Node N_2	Node N_3
$r_1(d)$	$r_1(d)$	$r_1(d)$
$w_3(e)$	$w_3(e)$	$w_3(e)$
$w_3(f)$	$w_3(f)$	$w_3(f)$
$w_2(a)$	$w_2(a)$	$w_2(a)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$r_1(a)$	$r_1(a)$	$r_1(a)$
$r_1(b)$	$r_1(b)$	$r_1(b)$
$w_1(c)$	$w_1(c)$	$w_1(c)$
$w_1(a)$	$w_1(a)$	$w_1(a)$
$r_2(a)$	$r_2(a)$	$r_2(a)$

References

[1] Lazy Release Consistency for Software Distributed Shared Memory
[2] Replication and Consistency Models
[3] 淺談 Microkernel 設計和真實世界中的應用
[4] Memory Coherence in Shared Virtual Memory Systems
[5] Active Messages: a Mechanism for Integrated Communication and Computation
[6] VNF Placement Optimization at the Edge and Cloud
[7] 上課投影片、維基百科等