Operating System : Three Easy Pieces

[1] Virtualization

* CPU Virtualization

1. Time-Sharing : Process

2. Limited Direct Execution

– To take better speed, and to prevent bad prejudice. There two modes: user mode, Kernel mode.

3. Scheduling

1) First In First Out → Shortest Job First → Shortest Time to Completion First (Turnaround time)

2) Round-Robin (Response time) → Better Turnaround time, Worse Response time (trade-off)

3) Multi-Level Feedback Queue (Cannot preview execution time)

- How to give priority (Law)

(1) Priority, if same Round-Robin

(2) Once in, get priority

(3) If given-time exhausted, drop priority

(4) After certain time S, all process get priority

4) Lottery Scheduling (Proportional Share) → used in Virtual machine (easy to know execution time)

4. Multiprocessor Scheduling

1) problem : Cache coherence, Synchronization, Cache affinity.

2) Single Queue Multiprocessor Scheduling – Good Workload, Easy implementation

3) Multi Queue Multiprocessor Scheduling – Good Scalability, Good Cache affinity

*Memory Virtualization

1. base and bound (Dynamic relocation)

– Easy Implementation, Protection.

- Not flexibility, Internal fragmentation; too wasteful

2. Segmentation

– Each segment has their own base and bound, Flexible.

- Not enough flexibility, External fragmentation; free space management, compact needed.

3. Paging

- Keep translation information by array (index is mapped to entry), Page Table is in the Memory.

→ Obviously each processes' page table is in OS area in memory.

- It also require too many memory space and Too late

→ It cause overhead to read information in memory.

4. TLB (Translation Lookaside Buffer) – a part of MMU(Memory Management Unit)

- TLB is a small cache, miss occur efficiency down.

- TLB miss can be reduced by Spatial locality and Temporal locality.

- There are two TLB miss handling ways: Hardware(CISC), Software(RISC).

→ Software TLBMH(TLB Miss Handler) is flexible, simple but it has two problems.

1) Trap handler handle two case (TLB Miss, and others).

2) TLB miss occur when accessing TLB miss handler.

→ Locate TLBMH in physical Memory, or in portion of TLB forever.

- In Context transportation, using LRU or random policy.

- Simple linear page table has too many memory space.

1) Segment-page hybrid: reducing waste, but non-flexible, cause external fragmentation.

2) Multi-Level Page Table: change linear table to Tree, Page Directory.

- Page Directory Entries has Valid bit, Page Frame Number.

- MLPT allocate PT space according to using address space, so can use smaller page table.
- Good Memory Management by dividing Page Table by page size. Flexible.
- But in TLB miss, there are two times Memory load. And complex (deeper level, good efficiency in hit but worse efficiency in miss).

3) Inverted Page Table(IPT)
- Using more space, using less time. (trade-off)
- Swap space in Disk decide total number of Memory Page which system uses.

→ present bit presents whether Page exist. If no and access that, Page fault is occurred.

→ When page fault occurred, page fault handler is operating in background(Daemon). It can take Page at disk with page table information which directs their swap space.

→ Processor can use more memory than real by using swap

* kind of Cache miss

(1) Compulsory miss : Empty Cache, refer that entry at first.

(2) Capacity miss : Full Cache, Evict any entries to get new entry.

(3) Conflict miss : Because of Set-associativity.

5. Page Replacement Policy

- Compare standard: The Optimal Replacement Policy (Unreal, Hard to implement, just standard)

→ Replace the last accessed page is the best way, and occurs the least miss.

1) First In First Out: Easy implementation, worse efficiency.

2) Random: Only follow luck, There is no corner case.

3) Least Recently Used, Least Frequently Used: Good efficiency.

→ Perfect implementation is hard, clock algorithm and use bit can make LRU approximately.

→ The way like Clock Algorithm, Use bit called **Scan resistance**. It is important in algorithm.

4) Dirty Page: To evict Dirty Page, CPU should write it on Disk. It prefers to evict Clean Page.

5) Prefetching: Using Spatial locality.

6. Thrashing

- occurred when Processes require more memory than system have. Several version of Linux solve it by out-of-memory killer.

[2] Concurrency

1. Thread

- A processing flow in Process sharing address space. Multi-thread program has more processing points than one. In thread context switch, thread uses address space which before one used.
- Each Threads are allocated one stack space(called thread local storage) in address space. Its size don’t have to be big, there are no problems in almost case. (except program contains many Recursive calls.)
- Sharing Data between thread can have problems, and it will be solved by mutual exclusion.
- Critical section indicates a part of code which accesses to sharing resources like variable, data.
- Race condition indicates the condition which threads try to access Critical section simultaneously.
- Indeterminate program is a program which its processing result will be change every process because of the dependency of multi-thread.
- Programmer creates Thread, Operating System controls it.

2. Lock

- Lock covers Critical section of code, then critical section is proceeded like one atomic process.
- Lock variable(mutex) has two states: available, acquired. In lock() operation, if any thread does not take lock, then that thread get lock and enter the critical section and lock is changed to acquired. In unlock() operation, lock is changed to available.
- Through Lock, programmer has a portion of control of Thread.
- Implementation of Lock

(1) Control Interrupt: Easy implement, but should allow process operates privileged operation, cannot be applied on multi-thread, and Disable interrupt in long time can lose important interrupts.
(2) Test-And-Set (Atomic Exchange) : fail to implement lock, and spin-wait is too unefficiency.
(3) Spin Lock : can supply Mutual Exclusion, but unfair and not effective in single processor. If number of thread is almost same with number of CPU (multi-processor), then effective.
(4) Compare-And-Swap(Compare-And-Exchange): more powerful than Test-And-Set
(5) Fetch-And-Add : On lock() each thread allocates their ticket, and scheduled their turn. Fair.
- Each Lock implementation has too much spin and waste CPU time for it.
→ Solve: If a thread reaches to spin, then Unconditionally yield CPU to others.
→ But there are too many Context switches, and does not solve Starving problem at all. (Any thread does only yield forever.)
- It is solved by OS support and management for waiting thread with Queue.
- Using Wake up / Waiting race reduces waiting time occurred by spin, and waiting queue solves starving problem by controlling next lock owner.
- Two level Lock : In First level, wait with spinning and expect it can be lock owner soon. If it didn't be lock owner, enter Second level and sleep until waked-up.

3. Lock based Concurrency Data Structure

- Thread Safe: Lock added Data Structure. Data Structure can be used by thread with safety.
→ Just adding lock simply, there will be performance problem. Should add others.
- Perfect Scaling: Despite of more workload, each work is proceeded concurrently so there are no extension of completion time.
- Concurrency Counter; Sloppy Counter: local counter and total counter. Local value transfer to total counter for certain frequency S. Lower S, better precise lower speed. (trade-off)
- Concurrency Linked List; Hand-over-hand locking(lock coupling) : Not total lock, but each node lock. But changing each node lock between acquired and available occur much overheads.
→ Regardless of better concurrency, performance can be worse. (lock,unlock occur overhead…)
- **Using one big lock is standard of concurrency data structure. Be Careful in Code flow when using lock(), and unlock().**
- Concurrency Queue: one lock in head, the other lock in tail. Insert routine access only tail, and extraction routine access only head.
- Concurrency Hash Table: Use lock on each Hash bucket and Use Concurrency Linked List.
- When making Concurrency Data Structure, Using one Big Lock (the simplest way). And only when performance decrease problem occurred, improve that. Even Linux OS used one Big Kernel Lock. But as multi-processor commercialized, BKL is removed.

4. Condition Variable

- Condition Variable is as a queue, thread wait until process state is same as wanted condition.
- When call wait(), assume that mutex is locked. Wait() do unlock and load with calling thread. If other thread send wake-up signal, then thread get lock again before wait() return. (**IMPORTANT**)
- When signal send to waiting process and it waked-up, cannot assure that the status is maintained until that process restart. (Producer/Consumer, bounded buffer problem)
→ Using while statement for condition checking, extending buffer size for concurrency.
- Hoare Semantic is the opposite concept of Mesa Semantic, as soon as waked-up thread restart immediately.
- In Mesa semantic, always use **While** statement.

5. Semaphore

- Semaphore has two function: sem_wait(), sem_post()
- sem_wait() make caller wait until Semaphore value is better than one, and decrement value one.
- sem_post() add Semaphore value and wake up one of waiting threads.
- If Semaphore value is negative, that value indicates the number of waiting threads.

- Binary Semaphore: Initial value is 1 and only have two value: 0, 1 (available, acquired). It can be used for lock.
- Condition Semaphore: initial value is 0 and it changed -1 to 0 to 1.
- Semaphore is a The Old Solution for Deadlock.

6. Concurrency Error

- non Deadlock error

(1) Atomicity violation

- Expected serializability(keeping sequentially) between many memory reference operations is not violated. To solve it, Add proc_info_lock for every accessing proc_info part.

(2) Order violation

- Sequence between two memory reference is changed. To solve it, Using Condition variable for forcing sequence.

- Deadlock Error
- Because of Dependency between code, and also encapsulation, Deadlock is occurred. Conditions.

(1) Mutual Exclusion: Thread requires its own control for its requirements.

(2) Hold-and-Wait: Thread waits with holding resource allocated to it.

(3) No preemption: Cannot take resource(lock) from the thread that occupying it.

(4) Circular wait: Circular loop of threads which take one or more what next thread requires.

- Circular wait can solve with total ordering. If hard to make total order, then make partial ordering.
- Hold-and-wait can solve with taking all locks at once. But it decrease concurrency.
- No Preemption can solve with trylock(). But it occur livelock. Livelock can solve adding delay for each loop.
- Mutual Exclusion can solve with remove mutual exclusion, but it’s hard. If can using Wait-free data structure did not need explicit locks.
- Deadlock can avoid with Scheduling. But it occur performance decrease.
- If Deadlock occur only once in a while, then Discovery and recovery can be good selection.

7. Event-Based Concurrency

- Event loop handle Event with Event handler. Event handle is the only work of system, Decision of next handled event is same effect with scheduling.
- Blocking Interface do all its own works before return to caller. But Non-blocking interface return immediately, so its works are completed at the background. Event-Based Programming should take Non-Blocking Interface.
- Do not Block in Event-Based server. Anything of executions of caller should not be blocked.
- There are Blocking System Call problems. System call of event can be blocked.

→ Solved with asynchronous I/O.

- And Event-Based Programming is Complex. Manual stack management.
- In multi-processor, Event-Based programming should take lock again. And it is not fitted with any system like paging. Also routine is changing, so routine management becomes difficult on an event-based. At last, some system did not adopt asynchronous I/O yet.
- Map-Reduce implement concurrency without lock or condition variable.

[3] Persistence

-