COMP3231/9201/3891/9283 Operating Systems 2021/T1

UNSW

Tutorial Week 10

Questions and Answers

Virtual Memory

1. What effect does increasing the page size have?

    - Increases internal fragmentation, and the unit of allocation is now greater, hence greater potential for un-used memory when allocation is rounded up. Practically speaking, this is not usually an issue, and round up an application to the next page boundary usually results in a relative small amount wasted compared to the application size itself.
    - Decreases number of pages. For a given amount of memory, larger pages result in fewer page table entries, thus smaller (and potentially faster to lookup) page table.
    - Increases TLB coverage as the TLB has a fixed number of entries, thus larger entries cover more memory, reducing miss rate.
    - Increases page fault latency as a page fault must load the whole page into memory before allowing the process to continue. Large pages have to wait for a longer loading time per fault.
    - Increases swapping I/O throughput. Given a certain amount of memory, larger pages have higher I/O throughput as the content of a page is stored contiguously on disk, giving sequential access.
    - Increases the working set size. Work set is defined as the size of memory associates with all pages accessed within a window of time. With large the pages, the more potential for pages to include significant amounts of unused data, thus working set size generally increases with page size.

---

2. Why is demand paging generally more prevalent than pre-paging?

    Pre-paging requires predicting the future (which is hard) and the penalty for making a mistake may be expensive (may page out a needed page for an unneeded page).

---

3. Describe four replacement policies and compare them.

    - Optimal
        - Toss the page that won't be used for the longest time
        - Impossible to implement
        - Only good as a theoretic reference point: The closer a practical algorithm gets to optimal, the better
    - FIFO
        - First-in, first-out: Toss the oldest page
        - Easy to implement
        - Age of a page is isn't necessarily related to usage
    - LRU
        - Toss the least recently used page

- Assumes that page that has not been referenced for a long time is unlikely to be referenced in the near future
- Will work if locality holds
- Implementation requires a time stamp to be kept for each page, updated on every reference
- Impossible to implement efficiently

- Clock
  - Employs a usage or reference bit in the frame table.
  - Set to one when page is used
  - While scanning for a victim, reset all the reference bits
  - Toss the first page with a zero reference bit.

---

4. What is thrashing? How can it be detected? What can be done to combat it?

   Thrashing is where the sum of the working set sizes of all processes exceeds available physical memory and the computer spends more and more time waiting for pages to transfer in and out.

   It can be detected by monitoring page fault frequency.

   Some processes can be suspended and swapped out to relieve pressure on memory.

Multi-processors

5. What are the advantages and disadvantages of using a global scheduling queue over per-CPU queues? Under which circumstances would you use the one or the other? What features of a system would influence this decision?

   Global queue is simple and provides automatic load balancing, but it can suffer from contention on the global scheduling queue in the presence of many scheduling events or many CPUs or both. Another disadvantage is that all CPUs are treated equally, so it does not take advantage of hot caches present on the CPU the process was last scheduled on.

   Per-CPU queues provide CPU affinity, avoid contention on the scheduling queue, however they require more complex schemes to implement load balancing.

---

6. When does spinning on a lock (busy waiting, as opposed to blocking on the lock, and being woken up when it's free) make sense in a multiprocessor environment?

   Spinning makes sense when the average spin time spent spinning is less than the time spent blocking the lock requester, switching to another process, switching back, and unblocking the lock requester.

---

7. Why is preemption an issue with spinlocks?

   Spinning wastes CPU time and indirectly consumes bus bandwidth. When acquiring a lock, an overall system design should minimise time spent spinning, which implies minimising the time a lock holder holds the lock. Preemption of the lock holder extends lock holding time across potentially many time slices, increasing the spin time of lock aquirers.

---

8. How does a read-before-test-and-set lock work and why does it improve scalability?

See the lecture notes for code. It improves scalability as the spinning is on a read instruction (not a test-and-set instruction), which allows the spinning to occur on a local-to-the-CPU cached copy of the lock's memory location. Only when the lock changes state is cache coherency traffic required across the bus as a result of invalidating the local copy of the lock's memory location, and the test-and-set instruction which require exclusive access to the memory across all CPUs.

Scheduling

9. What do the terms *I/O bound* and *CPU bound* mean when used to describe a process (or thread)?

   The time to completion of a *CPU-bound* process is largely determined by the amount of CPU time it receives.

   The time to completion of a *I/O-bound* process is largely determined by the time taken to service its I/O requests. CPU time plays little part in the completion time of I/O-bound processes.

---

10. What is the difference between cooperative and pre-emptive multitasking?

    Cooperative the thread specifically release the CPU, pre-emptive the thread has no choice.

---

11. Consider the multilevel feedback queue scheduling algorithm used in traditional Unix systems. It is designed to favour IO bound over CPU bound processes. How is this achieved? How does it make sure that low priority, CPU bound background jobs do not suffer starvation?

    Note: Unix uses low values to denote high priority and vice versa, 'high' and 'low' in the above text does not refer to the Unix priority value.

    I/O bound processes typically have many short CPU bursts. If such a process is scheduled, it will typically not use up it's time slice. Priorities are recomputed taking into account the consumed CPU, and hence an I/O-bound process will end up having a higher priority than a process that started at the same priority level and which used more CPU cycles.

    Even a process with low priority will be scheduled eventually, since the priority of processes that are continually scheduled eventually also receive a low or lower priority.

    Also note that the recorded amount of CPU consumed (that is used to calculate priority) is aged (reduced) over time, and hence CPU-bound processes also increase in priority is they are not scheduled.

---

12. Why would a hypothetical OS always schedule a thread in the same address space over a thread in a different address space? Is this a good idea?

    Context switch is faster. Better locality. If done too often (always) it starves other tasks.

---

13. Why would a round robin scheduler NOT use a very short time slice to provide good responsive application behaviour?

    CPU is consumed when switching from one task to another. This switching does not contribute to application progress. If done very frequently (a very short time slice), a significant portion of available CPU will be wasted on scheduler overhead.

I/O

14. Describe *programmed I/O* and *interrupt-driven I/O* in the case of receiving input (e.g. from a serial port). Which technique normally leads to more efficient use of the CPU? Describe a scenario where the alternative technique is more efficient.

    For programmed I/O, the CPU waits for input by continuously reading a status register until it indicates input data is ready, afterwhich, the CPU can read the incoming data, and then return to reading the status register wainting for further input.

    For interrupt-driven I/O, the serial port device sends an interrupt to the CPU when data is ready to be read. The interrupt handler is then invoked, which acknowledges receiving the interrupt, reads the data from the device, and returns from the interrupt. The CPU is free for other activities while not interrupt processing.

    Programmed I/O is normally less efficient as the CPU is *busy waiting* for input when input is unavailable.

    Programmed I/O can be more efficient in circumstances where input it frequent enough such that the overhead of interrupts (getting into and out of the interrupt handler) is more than the average time spent busy waiting. A realistic example is fast networking where packets are nearly always available.

---

15. A device driver routine (e.g. `read_block()` from disk) is invoked by the file system code. The data for the filesystem is requested from the disk, but is not yet available. What do device drivers generally do in this scenario?

    They block on a synch primitive that is later woken up by the disk interrupt handler when the block is available.

---

16. Describe how I/O buffering can be formulated as a *bounded-buffer producer-consumer problem*.

    Take the a file system buffer cache as an example. File system writes are *produced* by application requests. The OS must select an available entry in the buffer cache (or re-use an existing one), or block until a free slot is available in the *bounded-size* buffer cache.

    The interrupt handler can be considered a consumer as after the write to disk completes, it marks the buffer cache entry as *clean* (consumed), freeing it up for further writes.

---

17. An example operating system runs its interrupt handlers on the kernel stack of the currently running application. What restriction does this place on the interrupt handler code? Why is the restriction required?

    The interrupt handler must not block waiting for a resource, as it indirectly blocks the application that was running.

    If the application has the resource the interrupt handler requires (e.g. memory buffers), the system is deadlocked.

Page last modified: 9:13am on Tuesday, 9th of February, 2021

Screen Version

CRICOS Provider Number: 00098G