Operating Systems

Personal notes based on lecture material and assigned readings from Princeton's COS 318: Operating Systems, taught by Jaswinder P. Singh.

Table of Contents

Monolithic kernels vs. microkernels 1
Synchronization 2
CPU scheduling algorithms 4
Page table design 5
Page replacement algorithms 5
Three forms of performing I/O 7
Memory-mapped I/O 7
Direct Memory Access (DMA) 8
I/O software system layers 9
Deadlock 11

Monolithic kernels vs. microkernels

- Monolithic
  - All kernel routines together
  - System call interface
  - Examples: Linux, BSD Unix, Windows
  - Advantages
    - Shared kernel space
    - Good performance
    - Protection/security
  - Disadvantages
    - Instability
    - Inflexible – hard to maintain and extend
- Microkernel
  - OS services implemented as regular processes
  - Micro-kernel obtains services on behalf of users by messaging service processes
  - Examples: Mach, Taos, OS-X
  - Advantages
    - Flexibility

        - Fault isolation – problem in OS service will not crash kernel
    - Disadvantages
        - Inefficient – lots of boundary crossings
        - Insufficient protection
        - Inconvenient to share data between kernel and OS services

Synchronization

- Semaphores
    - Implementation
        - ```
          void semaphore_down (sema_t *s) {
               if (s->value == 0) {
                    block(&s->wait_queue);
               }
               else {
                    s->value--;
               }
          }
          ```
        - ```
          void semaphore_up (sema_t *s) {
               if (!queue_empty(&s->wait_queue)) {
                    unblock_one(&s->wait_queue);
               }
               else {
                    s->value++;
               }
          }
          ```
    - Usage
        - Can be used as a lock (i.e. for mutual exclusion) if s->value init to 1
            - First process to call semaphore_down decrements value to 0 and enters critical section
            - Any subsequent process is blocked on semaphore
            - First process exits critical section
                - If processes blocked on queue, value remains at 0 (lock remains "held") but one process is unblocked
                - Otherwise, value is restored to 1 (lock is "open")
        - Can be used to enforce event sequencing
            - In producer-consumer problem, set emptyCount = N and fullCount = 0
                - Producer only calls down on emptyCount and up on fullCount
                - Consumer only calls down on fullCount and up on emptyCount
        - In general, using for mutual exclusion not recommended
            - Easy to introduce bug, if order of downs not correct (see producer-consumer problem)

- Monitors
  - Collection of procedures, variables, and data structures grouped together in module/package
  - Processes can call procedures in monitor
    - Caveat: cannot access internal data structures directly
  - Monitor procedures are mutually exclusive
  - Example
    - ```
      Monitor ProdCons
            condition full, empty;

            procedure Enter;
            begin
                  if (buffer is full)
                        wait(full);
                  put item into buffer;
                  if (only one item)
                        signal(empty);
            end

            procedure Remove;
            begin
                  if (buffer is empty)
                        wait(empty);
                  remove an item;
                  if (buffer was full)
                        signal(full);
            end;
      ```
- Producer-consumer problem
  - Solution with locks and semaphores
    - ```
      void producer (void) (
            int item;

            while (1) {
                  item = produce_item();

                  down(&empty);
                  down(&mutex);

                  insert_item(item);

                  up(&mutex);
                  up(&full);
            }
      }
      ```
    - ```
      void consumer (void) (
            int item;
      ```
      ```

```
            while (1) {
                down(&full);
                down(&mutex);

                item = remove_item();

                up(&mutex);
                up(&empty);

                consume_item(item);
            }
        }
```

- Solution with monitors (implementation above)
  - ```
    procedure Procedure
    begin
        while true do
        begin
            produce an item
            ProdCons.Enter();
        end;
    end;

    procedure Consumer
    begin
        while true do
        begin
            ProdCons.Remove()
            consume an item
        end;
    end;
    ```

CPU scheduling algorithms

- Shortest Remaining Time to Completion First (SRTCF)
  - Good response time
  - Unfair (long processes starve if short jobs keep coming in)
- FCFS vs. Round Robin
  - RR much worse (turnaround time) for jobs about the same length
    - All jobs end up having around the same, high turnaround time (the combined length of all jobs)
    - If k jobs all of length l, RR has a turnaround time of around $k(kl) = k^2 l$ while FCFS has a time of $\frac{k(k+1)}{2} l$, which is half as much
- Virtual Round Robin
  - Auxiliary queue of I/O bound processes
  - Aux queue has preference over ready queue (for better parallelism)
- Priority scheduling
    ```

    - Adjust priorities dynamically
        - I/O wait raises priority
        - Reduce priority as process runs
    - Why adjust?
        - To prevent starvation
            - E.g. Low priority task acquires lock, higher priority task blocks on lock, third task of medium priority enters -> first two processes never get to run (priority inversion)
        - Fairness
- Lottery scheduling
    - Highly responsive – if new process is granted some tickets, it has chance of winning next lottery proportional to fraction of tickets held
    - Can approximate SRTCF by giving short jobs more tickets
    - Can avoid starvation by giving each process at least 1 ticket

Page table design

- Each process needs its own page table, as each has its own virtual address space
- Just as with pages, a smaller number of page table entries get most of the references
    - Solution: a hardware cache – the TLB (Translation Lookaside Buffer)
        - Usually located inside MMU
        - Contains between 8-256 page table entries
        - If large enough (64 entries), generally managed by software
            - Frees up area on CPU chip to be used for other caches
    - Soft misses vs. hard misses
        - Soft miss – page referenced not in TLB, but in memory
        - Hard miss – page not in memory (requires disk I/O)
            - Minor page fault – page not in process' page table, but actually in memory (brought in by another process)
                - Page needs to be mapped to process' page table
            - Major page fault – page must be brought in from disk
            - Segmentation fault – invalid memory access; process killed

Page replacement algorithms

- Not Recently Used (NRU)
    - OS divides pages into 4 classes:
        - Class 0: not referenced, not modified
        - Class 1: not referenced, modified
        - Class 2: referenced, not modified
        - Class 3: referenced, modified
    - R is periodically cleared (e.g. on clock interrupt) to distinguish recently referenced pages from those not
    - NRU removes page at random from lowest-numbered, non-empty class
- FIFO With Second Chance

    - Check referenced bit R of oldest page
    - If 0, replace; else if 1, clear R, put page at end of list, and continue searching
    - Goal: find old page not referenced in current clock interval
    - Pros
        - Simple to implement
    - Cons
        - Unnecessarily inefficient – pages moved around too often
        - Worst case performance is not good
- Clock
    - Hand points to oldest page
    - If R bit is 0, page is evicted, replaced by new page, and hand is advanced
    - If R bit is 1, it is cleared and hand is advanced to next page
        - Process repeated until page found with R = 0
    - Pros
        - Pages do not have to be moved around (i.e. if R bit is 1)
- LRU
    - Good approximation of optimal
    - But…hard to implement
        - Requires keeping in-memory linked list of all pages
            - Most recently used page at front; least recently used at rear
        - Problem: list must be updated on every reference
            - Moving page is very slow operation, even in hardware
    - Alternatively (with special hardware)
        - Use 64-bit counter C, incremented after every instruction
        - After each memory reference, current value of C stored in page table entry for page just referenced
        - When page fault occurs, all counters examined to find lowest one
- Aging a.k.a. Not Frequently Used (NFU)
    - Good (software) approximation of LRU
    - Software counter (8 bits) associated with each page
    - At each clock interrupt
        - Counter is right shifted 1 bit
        - R bit is added as leftmost bit
    - On page fault, page with lowest counter is removed
    - 8 bits generally enough if clock tick around 20 msec
- Working Set
    - Set of pages in most recent K page references defined as working set
        - Alternatively: set of pages referenced in last τ msec of execution time
    - Prepaging
        - Load processes' working set before letting it run
    - Page table scanned
        - If R bit is 1, current virtual time written into TOLU (time of last use) field in page table entry
        - If 0, page is candidate for removal
            - Process age is computed (current virtual time – TOLU)

                - If age greater than τ, process evicted
        - If entire table scanned without eviction candidate, all pages in working set
            - If one or more pages with R = 0 found, oldest on evicted
            - Otherwise, page randomly chosen and removed
- WSClock
    - Combines working set with clock algorithm
    - Used widely in practice
    - Differences
        - If R bit of page is 0 and age is greater than τ
            - If page is clean, it is evicted and replaced
            - If page is dirty, disk write is scheduled
                - Clock hand advances
        - If clock hand returns to starting point…
            - If writes scheduled – hand continues moving forward to find first, now-clean page
            - If no writes scheduled, first clean page is replaced
                - Assumption: all pages in working set
    - Pros
        - Relatively simple to implement
        - Good performance

Three forms of performing I/O

- Programmed I/O
    - OS continually checks to see if printer is available/ready to accept character (polling/busy waiting)
    - Simple, but is slow and wastes CPU cycles
        - Inefficient in more complex systems
- Interrupt-driven I/O
    - Interrupts used to notify CPU when printer is ready, so it doesn't have to wait
    - Interrupts occur on every character – wastes CPU time
- DMA-based I/O
    - DMA controller feeds characters to printer, freeing up CPU to do other work
        - Number of interrupts reduced to 1 per buffer printed, instead of 1 per character
    - DMA controller much slower than CPU, so if CPU is idle, then not worth it

Memory-mapped I/O

- What is it?
    - Way of performing input/output between CPU and peripheral devices
    - Means of communication between CPU and control registers and device data buffers
        - Each controller has few registers used for communication with CPU
- Approaches

    - 1 – Each control register assigned to an I/O port (8- or 16- bit register). The I/O port space forms a separate, protected address space from that of user programs. Used by early computers, such as IBM 360
    - 2 – Memory-mapped I/O. Map all control registers into memory space. Specifically, all control registers assigned unique memory addresses at/near top of address space
- Advantages
    - Device control registers become just variables in memory and can be addressed in C just like any other variables (abstraction)
        - I/O device driver can be written entirely in C
    - No special protection mechanism needed to prevent user processes from performing I/O
        - OS must just map control registers to addresses outside of process' virtual address space
        - Control registers for different devices can be assigned to different pages of address space, allowing OS to give fine grained control to user (simply by including certain pages in page table, and excluding others)
    - Every instruction that can reference memory can now also reference control registers
        - Improves performance; otherwise, extra instructions needed to read control register into CPU, etc.
- Disadvantages
    - Caching has to be selectively disabled for device control registers
        - Control register values are not variables in the same sense as regular registers – their values can change independently of what the code does/does not do, and thus cannot be cached
    - Separate memory bus on memory-mapped machines means I/O devices have no way of seeing memory address/responding to them
        - Possible solutions require additional hardware complexity
            - First send all memory references to memory; if memory fails to respond, CPU tries other buses
            - Put snooping device on memory bus to pass all addresses presented to potentially interested I/O devices
                - I/O devices may not be able to process requests at speed memory can
            - Filter addresses in memory controller; address in nonmemory range (premarked) forwarded to devices instead of memory
                - Requires using special range denoting registers

Direct Memory Access (DMA)

- Efficient mechanism for CPU to exchange data with I/O controller
- Requires special DMA controller in hardware (or separate DMA controller for each I/O device)

- Normal disk read
  - Disk controller reads block from disk drive bit by bit into internal buffer
  - Disk controller computes checksum to verify that no read errors have occurred
  - Disk controller causes interrupt
  - On invocation, OS reads disk block from controller's buffer a byte/word at a time into main memory
- DMA procedure
  - CPU programs DMA controller by setting its registers
    - CPU issues command to disk controller instructing it to read data from disk into internal buffer and verify the checksum
  - DMA controller initiates transfer by issuing read request over bus to disk controller
    - Disk controller does not know/care whether read request came from CPU or DMA controller
  - Disk controller writes to memory (fly-by mode)
    - Memory address placed on bus' address lines
  - DMA controller then increments memory address to use and decrements byte count
    - If byte count >= 0, steps above are repeated until count reaches 0
  - DMA controller interrupts CPU to let it know that transfer is complete
- Disadvantages
  - Main CPU often much faster than DMA controller
    - Can do job faster when limiting factor is not speed of I/O device
  - Having CPU do all work in software can be cheaper

I/O software system layers

- Top to bottom
  - User-level I/O software
  - Device-independent operating system software
  - Device drivers
  - Interrupt handlers
  - Hardware
- Interrupt handlers
  - Handling steps
    - Save registers (not already saved by interrupt hardware)
    - Set up context for interrupt-service procedure
      - Set up TLB, MMU, page table
      - Set up stack for interrupt-service procedure
    - Acknowledge interrupt controller
      - If no centralized interrupt controller, re-enable interrupts
    - Copy registers from where saved to process table
    - Run interrupt-service procedure
    - Invoke scheduler to choose which process to run next

        - Set up MMU context for process to run next
        - Load new process' registers
        - Start running new process
- Device drivers
    - Device-specific code for controlling an I/O device attached to a computer
    - Normally part of kernel (in current architectures)
        - Needs to access device's hardware
        - Moving device drivers to user space (e.g. as in MINIX 3) would eliminate major source of system crashes – buggy drivers that interfere with kernel function
    - Installation of new drivers expected
        - Driver functionality and interface with OS must be well-defined
            - Standard interfaces defined for block devices, and for character devices (see below)
        - Drivers normally positioned below rest of OS (device-independent software), but above interrupt handlers
    - Categories
        - Block devices – disks
        - Character devices – keyboards, printers
- Device-independent I/O software
    - Boundary between drivers and device-independent software is system dependent
    - Typical device-independent functionality
        - Uniform device driver interfacing
        - Buffering
        - Error reporting
        - Allocating/releasing dedicated devices
        - Providing device-independent block size
    - Uniform interfacing for device drivers
        - OS defines a list of functions (i.e. an interface) that device driver must supply
            - Device driver provides table of function pointers to OS
        - I/O device naming
            - Device-independent software maps symbolic device names (i.e. /dev/disk0) to *major device number* (contained in i-node)
        - Protection
            - Devices appear in file system as named objects, so file protection rules apply and system can set proper permissions
    - Buffering
        - Buffer created inside kernel to hold (buffer) input from/output to block and character devices
            - Double buffering and circular buffer schemes
        - Necessary sequential buffering comes with a performance cost
    - Errors
        - Framework for error handling is device-independent

        - Classes
            - Programming errors – e.g. writing to input device, providing invalid buffer address
                - Solution – report back error code to caller
            - I/O errors – e.g. writing to damaged disk block
                - Driver attempts to handle situation
                - If it can't, passes problem up to device-independent software
                - Can ask user what to do – retry, ignore, kill calling process
                - If no user available, can fail system call with error code
    - Allocating/releasing dedicated devices
        - Some devices (i.e. printers) can only be used by single process
        - OS must manage requests for device usage
            - One solution – require processes to call open() and close(), returning failure if device unavailable
            - Alternative – block caller, instead of failing, on a device request queue
    - Device-independent block size
        - Device-independent software hides fact that underlying devices (i.e. disks) have different block (i.e. sector) sizes

Deadlock

- Four necessary conditions
    - Mutual exclusion condition
        - Resource can be assigned to exactly one process at a time
    - Hold and wait
        - Processes holding resources can request new resources
    - No preemption
        - Resources cannot be taken away from processes
    - Circular chain of requests