SE 350: Operating Systems

Michael Noukhovitch

Winter 2015, University of Waterloo

Notes written from Thomas Reidemeister's lectures.

Contents

1 Introduction

1.1 Definitions

Operating Systems: a standardized abstraction from hardware that:

- manages resources
- provides set of services
- consumes resources

Instruction execution:

1. fetches instruction into IR
2. executes instruction

In reality, it is a bit more complicated: there is a pipeline, out of order execution...

1.2 Interrupts

1.2.1 Type of interrupts

- **program** result of instruction execution e.g. arithmetic overflow
- **timer** timer within process, allows OS to perform regular functions
- **I/O** generated by I/O controller
- **hardware failure** power failure or memory parity failure

1.2.2 How interrupts work

Hardware:

1. interrupt issued
2. processor finishes current instruction
3. acknowledge interrupt
4. push PSW and PC onto control stack
5. load new PC

Software:

6. save remainder of process state
7. interrupt
8. restore process state information
9. restore PSW and PC

1.3 Multiple interrupts

Two ways of handling an interrupt during an interrupt:

1.3.1 Sequential

Ignore any interrupts when you are in an interrupt and when done, check for interrupts that occurred.

1.3.2 Nested

If the second interrupt is of higher priority, recurse into it. Otherwise, wait until interrupt is finished.

1.4 Peripheral interrupts

Peripherals such as hard drives take a while to complete their action. As opposed to waiting and wasting that time, we use an interrupt to execute other instructions while our I/O process runs.

1.4.1 Programmed I/O

No interrupts occur, just wait until the I/O is complete

1.4.2 Interrupt-driven I/O

Processor interrupted when I/O is ready and all writes/reads are passed through CPU into memory. This is faster, since there is no waiting.

1.4.3 Direct memory access

Transfers a block of data directly into memory and interrupt is sent when process is complete. This is more efficient because data does not need to go through CPU. Not always available (e.g. external peripheral)

1.5 Memory hierarchy

Major constraints in memory:

- size
- speed
- cost

1.5.1 Hierarchy

Top: Inboard memory
Middle: Outboard storage
Bottom: Offline storage

1.5.2 Cache

Cache: small, fast memory, invisible to the OS, that speeds up accesses exploiting the principle of locality

2 Operating Systems Overview

2.1 Definition

Operating System a program that controls the execution of applications and is a standard interface between hardware and software

- **convenience**: need no knowledge of hardware
- **efficiency**: move optimization from devs to tools
- **ability to evolve**: can replace internals

Kernel: portion of the OS in main memory; a nucleus that contains frequently used functions **OS services**

- program development and execution
- access & control of access to I/O
- system access control
- error detection and response
- accounting

2.2 OS Innovations

2.2.1 Hardware Features

- **Memory protection**: do not allow memory containing monitor to be altered
- **Timer**: prevents a job from monopolizing system
- **Privileged instruction**: certain instructions can only be executed by the monitor (e.g. I/O)
- **Interrupts**

2.2.2 Modes of operation

To protect users from each other (and the kernel from user), we have two modes:

- **User mode**: not privileged
- **Kernel mode**: privileged and access to protected memory

2.2.3 Multiprogramming

When one job needs to wait for I/O, the processor can switch to another job. The timer is also used to switch processes and stop monopolization.

- maximize processor use
- use job control language

2.2.4 Time Sharing

Processor time is shared by multiple users

- minimize response time

2.3 Major Achievements

2.3.1 Processes

Process: a program in execution

- a program
- associated data
- execution content (needed by OS)

2.3.2 Memory Management

- process isolation
- automatic allocation
  - virtual memory: allows programmer to address memory without regard to physical addressing
  - paging: allows processes to be comprised of fixed-size blocks (pages)
- swap program code
- shared memory
- long term storage

2.3.3 Information Security

- availability: protecting system against interruption (downtime)
- confidentiality: authorizing data (chmod)
- data integrity: protect from modification
- authenticity: verifying identity of users

2.3.4 Scheduling and Resource Management

- fairness: give equal access to resources
- differential responsiveness: discriminate by class of jobs
- efficiency: maximize throughput

2.3.5 System Structure

The system as a hierarchical structure with each level relying on lower levels for its functions.

1. circuits: registers, gates, buffers
2. instructions: add, subtract, load, store
3. procedures: call stack, subroutine
4. interrupts
5. processes: suspend, wait, resume
6. local store: blocks of data, allocate
7. virtual memory: segments, pages
8. communications: pipes
9. file system: files
10. external devices
11. directories
12. user process
13. shell

2.4 Modern Operating System

Developments leading to modern operating systems:

- **Microkernel architecture**: only essentials to kernel, everything else in user space (e.g. QNX)
- **Multithreading**: process divided into concurrent threads
- **Symmetric multiprocessing**: multiple processors share main memory and I/O
- **Distributed OS**: illusion of a single main and secondary memory
- **Asymmetric multiprocessing**: one big processor controls many small ones
- **Object oriented design**: customize OS without disrupting system

3 Processes

3.1 Process Elements

Process Control Block (PCB): data structure that contains process elements

- **identifier**
- **state**
- **priority**
- **memory pointers**
- **context data**: registers, PSW, PC
- **I/O status information**
- **accounting information**: processor time, time limits, threads

3.2 Process Model

3.2.1 Five State Model

- **running**: currently executing
- **ready**: can be executed
- **waiting**: can't execute, blocked by something
- **new**
- **exit**: halted or aborted

Since the processor is faster than I/O, we may sometimes want to admit more processes even after we are out of memory. To do this we **swap** out processes to disk that are waiting and we get two new states:

- **blocked/suspend**
- **ready/suspend**

3.2.2 Process Creation

- new batch job
- interactive logon
- created by OS for a service
- created by existing process

3.2.3 Process Termination

- normal completion
- time limit exceeded
- time overrun
- memory unavailable
- bounds error (segfault)
- protection error
- arithmetic error
- I/O error
- invalid instruction
- privileged instruction
- data misuse
- OS intervention
- parent termination
- parent request

3.2.4 Process Suspension

- swapping
- other OS reason
- user request
- timing
- parent request

3.2.5 Process Switching

clock interrupt : maximum allowed time surpasses

I/O interrupt : I/O completed

memory fault : memory address is in virtual memory, bring into main memory

trap : error or exception, used for debugging

supervisor call : switch to kernel process

3.3 OS Control Structures

3.3.1 Memory Tables

- keeps track of main and secondary memory
- protection for access to shared memory
- information to manage virtual memory

3.3.2 I/O Tables

- manages I/O devices (status and availability)
- location in main memory for transferring I/O

3.3.3 File Tables

- manages existance and location of files
- attributes and status of files (e.g. rwxr...)

3.3.4 Process Table

- where process is located in memory as a **process image**:
  - program
  - data
  - system stack
  - PCB

4 Threads, SMP, Microkernels

4.1 Threads

Thread: execution entity under a process **Multithreading**: multiple threads of execution within a single process

4.1.1 Benefits

- less time to create/terminate (no kernel resource allocation)
- less time to switch between threads v. processes
- sharing memory between threads

4.1.2 States

- spawn
- block
- unblock
- finish

4.1.3 Approaches

User-level threads:

- thread management by application
- less switching overhead
- scheduling is app-specific
- can run on any OS

Kernel-level threads:

- thread management by kernel
- can schedule threads on multiple processors
- OS calls blocking only the thread

Combined approach:

- threads can be grouped to kernel threads
- kernel knows/schedules threads and processes

4.2 Microkernel

Microkernel: small operating system that only contains essential core functions

4.2.1 Benefits

- uniform interface on request by process
- extensibility
- flexibility
- portability
- reliability
- distributed systems support
- object-oriented OS

4.2.2 Design

- low-level memory management: map each virtual page to a physical page
- interprocess communication: copying messages
- I/O and interrupt management: interrupts as messages, no knowledge of IRQ handlers

5 Concurrency

5.1 Terms

critical section : section of code that may have concurrency issues (shared memory ...)

deadlock : processes are locked because each is waiting for the other

livelock : two processes cycle uselessly because of the others' cycling

mutual exclusion : shared resources may not be modified concurrently

race condition : multiple threads/processes read and write shared memory concurrently

starvation : a runnable process is overlooked indefinitely by the scheduler

For mutual exclusion requires:

- 1 process at a time (per resource) in the critical section
- process halting in non-critical section must not interfere with other processes
- no deadlock or starvation
- no delay to critical section if it is unblocked
- no assumptions about relative process speeds or quantities
- process remains inside critical section for finite number of time

and can be achieved by any of the following options:

5.2 Hardware Support

5.2.1 Interrupt Disabling

- guarantees mutual exclusion on uniprocessor
- no guarantee for mutliprocessor
- a process runs until interrupt or it invokes OS service

5.2.2 Machine Instructions

- performed in single instruction cycle
- access to memory is blocked for other instructions

Advantages:

- applicable to any number of processes (single processor or multiprocessor)
- simple and easy to verify
- can support multiple critical sections

Disadvantages:

- busy-waiting consumes time
- starvation possible (one proc leaves critical section and multiple waiting)
- deadlock possible (low proiority holds critical section but OS switched to high proirity)
- pipleline stalls

5.3 Semaphore

Semaphore: variable with integer value used for signaling

5.3.1 Operations

- initialize with non-negative value
- **wait** decrements value
- **signal** increments value
- only the process with the lock can release the lock

Example 5.1. Semaphore primitive

```
struct semaphore {
    int count;
    queueType queue;
}

void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        s.queue.push(thisProcess);
        block(thisProcess);
    }
}
void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        nextProcess = s.queue.pop();
        addToReady(nextProcess);
    }
}
```

5.3.2 Producer-Consumer

Concurrency problem where a producer adds elements to a buffer and a consumer takes them out, we can use extra semaphores to keep track of other critical values

Example 5.2. Infinite-buffer producer/consumer

```
semaphore n = 0;
semaphore s = 1;
void producer() {
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer() {
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
```

5.3.3 Reader-Writer

Concurrency problem where many readers can read from a file, but only one writer may write to it and all readers must wait for the writer

Example 5.3. Readers/Writers problem with Reader priority

```
int readCount;
semaphore x = 0, wsem = 1;
void reader() {
    while (true) {
        semWait(x);
        readCount++;
        if (readCount == 1)
            semWait(wsem);
        semSignal(x);
        READUNIT();
        semWait(x);
        readCount--;
        if (readCount == 0)
            semSignal(wsem);
        semSignal(x);
    }
}
void writer() {
    while (true) {
        semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);
    }
}
```

5.4 Monitor

Monitor: a software module that controls entry to data using a mutex

- uses condition variables for signaling
- unused signals are lost *(diff from semaphore)*

5.5 Message Passing

Enforce mutual exclusion by information exchange

5.5.1 Synchronization

Blocking : sender and receiver are blocked until messaged delivered (rendevous)

Non-blocking Send : only receiver blocked until message arrives

Non-blocking : no waiting

5.5.2 Addressing

Direct Addressing:

- send has a specfic identifier of destination process
- receive knows which process to expect OR
- receieve uses source parameter to return value after message received

Indirect Addressing:

- messaged sent to shared data structure made of queues (**mailboxes**)
- processes send messages to a mailbox for others to pick up

Example 5.4. Indirect messaging

```
const int n = /* number of processes */
create_mailbox(mutex);
void P (int i) {
    message msg;
    while (true) {
        receieve(mutex, msg);
        /* critical section */
        send(mutex, msg);
        /* remainder */
    }
}
```

6 Deadlocks

6.1 Introduction

Deadlock : permanent blocking of a set of process over shared resources

There are two types of resources for the purpose of deadlocks:

- **reusable**: deadlock occurs if processes hold the resource the other requests
- **consumable**: deadlock occurs if a receive() message is blocking

Conditions for a deadlock:

- mutual exclusion: necessary
- no preemption: necessary
- hold and wait: necessary
- circular wait: sufficient

6.2 Prevention

Deadlock Prevention: design the system to prevent at least one of the conditions:

- mutual exlcusion
  - must be OS supported
  - often times, impossible to remove
- no preemption
  - if request denied, release all resources and re-request
  - use preemption to release all resources by lower priority
- hold and wait
  - request all necessary resources at once
- circular wait
  - use a linear ordering of resources, locking in order

6.3 Avoidance

Deadlock Avoidance: dynamically decide whether a resource allocation could lead to a deadlock

- maximum resource requirements known
- processes are indepenedent
- no process may exit holding resources

Two strategies

1. don't start a process if the sum of all its requests could lead to deadlock
2. don't grant incremental resource request if it could lead to deadlock

Banker's Algorithm: maintain a safe state where, with the current allocation scheme, there is at least one sequence that does not lead to deadlock

6.4 Detection

Deadlock Detection: periodically check for a deadlock and choose a strategy to undo the deadlock
Strategies:

- abort all deadlocked processes
- backup processes to some checkpoint
- iteratively abort deadlocked processes until deadlock is gone
- iteratively preempt resources until deadlock is gone

7 Memory Managementt

7.1 Requirements

- relocation: swapping, logical -> physical addressing
- protection: no access to memory of other processes (run-time checks)
- sharing: allow process to share memory
- logical organization: write and compile code modules separately
- physical organization: amount of RAM should not stop program from running

7.2 Fixed Partitioning

Fixed partitioning: all sizes are fixed from beginning

- equal size: simple but problematic
- unequal size: assignment is better but slower

7.3 Dynamic Partitioning

Dynamic partitioning: partitions vary in length and number, can cause **external fragmentation** (holes between blocks of memory)

Algorithms to fit a block into memory:

- best-fit algorithm: choose block closest in size (slow)
- first-fit: fastest
- next-fit: first fit from previous allocation
- buddy system: split a larger block into 2, use a BST to keep track

7.4 Paging

Paging: partition memory into small fixed-size chunks

- **pages**: chunks of a process
- **frames**: chunks of memory

logical address: (page, offset)
physical address: (frame, offset)

7.5 Segmentation

Segmentation: variable size pages, enables no internal fragmentation
logical address: (segment number, offset)

7.6 Paging + Segmentation

One segment table per process and one page table per segment

8 Virtual Memory

8.1 Introduction

Virtual Memory : load the program's code and data on demand, reading and storing to disk

Resident Set : portion of the process that is in main memory

Page Fault : interrupt triggered when OS accesses non-resident data

Thrashing : constant paging degrading performance

Internal Fragmentaion : fragmentation inside a block

External Fragmentation : fragmentation between blocks

8.2 Page Tables

Each process has a **page table** that stores:

- page number
- present bit
- modified bit
- other control bits
- frame number

These tables can end up being too big so we use:

- **Two-Level Hierarchical Page Table**: use a table to page the paging table
- **Inverted Page Table**: hash on page numbers to get pointer to page table entries, total memory proportional to real address space

Translation Lookaside Buffer: a cache for page table that uses the principle of (temporal) locality to store the recent pages accessed

8.3 Segment Tables

Each entry contains:

- length of the segment
- present bit
- modified bit
- other control bits
- segment base address

8.4 Required Algorithms

OS design must support:

- **fetch policy**: when a page should be brought into memory
  - demand paging: bring pages in on demand
  - prepaging: bring in more pages than needed
- **placement policy**: where a process piece should reside (minimize fragmentation)
- **replacement policy**: which page should be replaced, local vs global scope
  - optimal: replace page where time to next reference is longest
  - least recently used (LRU)
  - first-in first-out (FIFO): replace page that has been in memory longest
  - clock policy: replace first page with "use" bit 0 (approximates LRU)
  - page buffering: cache and revive pages to two lists: modified and unmodified
  - working set: remove pages that haven't been referenced in past t time
- **resident set management**: good size for resident set

  fixed allocation : decide how much memory to give ahead of time
  variable allocation : number of pages allocated varies over time
  local replacement scope : replace only pages from same process
  global replacement scope : can replace any page

  - fixed, local: difficult to predict correct size
  - variable, global: easy to implement, risk of reducing process' resident set
  - variable, local: re-evaluate allocation periodically
- **cleaning policy**: when to write pages back to memory
  - demand cleaning: write out a page only when replaced
  - pre-cleaning: write out in batches
- **load control**: control number of processes resident in main memory (avoid thrashing) figure out which process to remove: **process suspension**:
  - lowest priority process
  - faulting process
  - last process activated
  - process with smallest resident set (easiest to reload)
  - largest process (most memory freed)

9 Uniprocessor Scheduling

9.1 Scheduling Types

Long-term : whether to add to pool of processes

Medium-term : whether to add process to main memory

Short-term : which process to execute (aka dispatcher)

- invoked during an interrupt

I/O Scheduling : which process' I/O request to handle

9.2 Criteria

User-oriented, Performance

Turnaround Time : time between submission and completion of process

Response Time : time between submission and first response

Deadline : most important criteria

User-oriented, Other

Predictability : a program should run about the same regardless of load

System-oriented, Performance

Throughput : number of processes completed per unit time

Process Utilization : percentage of time processor is busy

System-oriented, Other

Fairness : without guidance, all processes should be treated the same

Enforcing Priorities : favour higher priorities

Balancing Resources : keep resources busy, favour processes that don't utilize stressed resources

9.3 Scheduling Algorithms

Decision mode can be:

- preemptive
- non-preemptive
- cooperative: developer programs the context switch

9.3.1 First-Come First-Served

Each process joins the ready queue and runs in FIFO

- good for computationally-intensive
- favours CPU-bound processes

9.3.2 Round Robin

User preemption based on periodic interrupts every quantum q

- favours CPU-bound processes

Improve it with **Virtual Round Robin** which maintains specific I/O queues to even out the playing field

9.3.3 Shortest Process Next

Non-preemptive strategy with shortest expected time

- possible starvation of long processes
- must know service time

9.3.4 Shortest Remaining Time

Preemptive version of SPN

- no additional interrupt (as in RR)
- better turnaround time than SPN
- elapsed service time must be recorded

9.3.5 Highest Response Ratio Next

Choose process with greatest Ratio $= \frac{\text{time waiting+service time}}{\text{service time}}$

- need to know service time
- no starvation

9.3.6 Feedback

Use queues of different priorities, penalizing processes that take long by moving a process down a priority after it finishes running for q time

- don't need to know service time
- may favour I/O-bound processes
- starvation of long running processes possible

9.3.7 Fair Share

Make decisions on the basis of process sets (*e.g.* threads that make up applications). Assign weights to each set and allocate resources according to weights

10 Multiprocessor Scheduling

10.1 Introduction

10.1.1 Classifications

- loosely coupled/distributed: work together but each has its own memory and I/O
- functionally-specialized: controlled by a master processor
- tightly coupled: share main memory, controlled by OS

10.1.2 Granularity

- independent: multiple unrelated processes
- coarse/very coarse: distributed processing across network nodes
- medium: parallel processing within an application (*e.g* threads)
- fine: parallelism inherent in single instruction stream

10.2 Scheduling Design

Assignment of processes to processors:

1. multiprocessor is uniform
   (a) static assignment: assign processes initially and no change
   (b) process migration: migrate processes dynamically
   (c) dynamic load balancing: start with static assignment but migrate as needed
2. multiprocessor is heterogenous: need special software

<++>