1 OS notes

1.1 History of computers

TL;DR

1.2 Computer Architecture

1.2.1 The Von Neumann Architecture

1.2.2 Levels of programming languages

1. Executable File (“Machine code”)
2. Object File linked with other Object Files (“Libraries”) into 1.
3. ASM Source which is assembled into 2.
4. C/C++ Source code which is compiled into 3.
5. ML/Java Byte-code which is interpreted

This is analogous to operation of the computer.

1.2.3 Layered Virtual Machines

Think of a virtual machine in each layer built on the lower VM; machine in one level understands language of that level

0. Digital Logic Level
1. **Conventional Machine Level**
2. **Operating System Level**
3. Assembly Language Level
4. Compiled Language Level
5. Meta-Language Level

1.2.4 Registers

- Very fast on-chip memory
- Typically 32 or 64 bits
- 8 to 128 registers is usual
- Data is loaded onto registers before being operated on
- Registers may not be visible to the programmer
- Most processors have *data* and *control* (special meaning to CPU) registers

1.2.5 Memory Hierarchy

1. CPU
2. Cache
    i. fast, expensive
    ii. several levels of cache
3. Main Memory
4. DISK I/O
    i. I/O devices usually connected via a bus
    ii. very slow and cheap

1.2.6 Fetch-Execute Cycle

PC initialised to fixed value on CPU reset. Then repeat until halt:

1. instruction *fetched* from memory address in PC into instruction buffer
2. Control Unit *decodes* the instruction
3. Execution Unit *executes* the instruction
4. PC is updated either explicitly by a jump or implicitly

1.2.7 Buses

A bus is a group of 'wires' shared by several devices. Buses are cheap and versatile but can become a bottleneck on performance. A bus typically has:

- address lines
- data lines
- control lines

A bus is operated in a master-slave protocol: e.g. to read data from memory, CPU puts address on a bus and asserts 'read'; memory retrieves data, puts data on bus; PC reads from bus. In some cases an initialisation protocol is needed to decide which device is the bus master.

1.2.8 Bus Hierarchy

1.2.9 Interrupts

There are devices much slower than the CPU. We can't have CPU wait for these devices. Also, external events may occur.

Interrupts provide a suitable mechanism. Interrupt is a signal line into CPU. When asserted, CPU jumps to a particular location (e.g. on x86 on interrupt the CPU jumps to address stored in relevant entry of table pointed to by IDTR control register). The jump saves state; when the interrupt handler finishes, it uses a special return instruction to restore control to original program.

1.2.10 Direct Memory Access

DMA means allowing devices to write directly (via a bus) into main memory, e.g. CPU tells device 'write next block of data into address x' and gets an interrupt when done.

PCs have a basic DMA; IBM main-frames have I/O channels which are a sophisticated extension to DMA.

1.3 Operating System function

1.3.1 What is an Operating System for?

- handles relations between CPU, memory and devices
- handles allocation of memory
- handles sharing of memory and CPU between different logical tasks
- handles file management
- (in Windows) handles the UI graphics

Kernel - single (logical) program that is loaded at boot time and has primary control of the computer

1.3.2 Early batch systems

In the beginning, OS simply transferred programs from punch cards into memory. Operator had to set up entire job, programmatically.

Monitor is a simple resident OS that reads jobs, transfers control to programs, receives control. Monitor is permanently resident, programs must be loaded into a different area of the memory.

Batches of jobs can be put onto one tape and read in turn by the monitor - reduces human intervention.

Protecting the monitor from the users, which should not be able to:

- **memory protection**: write to monitor memory
- **timer control**: run forever
- **privileged instructions**: directly access I/O or certain other machine functions
- **interrupts**: delay the monitor's response to external events

1.3.3 Multiprogramming

Jobs would waste 75% CPU cycles waiting on I/O. *Multiprogramming* was introduced to tackle this. Monitor loaded several user programs when one is waiting for I/O, run another.

Multiprogramming, means the monitor must:

- manage memory among the various tasks
- schedule execution of the tasks

1.3.4 Time-sharing

Allow interactive access to computer with many users sharing. Early system gave each user 0.2s of CPU time; then save state and load state of next scheduled user.

1.3.5 Virtual Memory

Multitasking and time sharing are much easier if all tasks are resident rather than being swapped in and out of memory.

Virtual memory - decouples memory as seen by the user task from physical memory. Task sees virtual memory which may be anywhere in real memory or paged out to a disk.

1.3.6 The Process Concept

With virtual memory, it becomes natural to give different tasks their own independent *address space* or view of memory. Monitor then schedules *processes* appropriately and does all the *context-switching* transparently to user process.

1.3.7 Modes of CPU operation

To protect OS from users, all modern CPUs operate in more than one *privilege level*:

- IBM has **supervisor** and **problem** states
- x86 has rings 0, 1, 2, 3

Transitions to higher privilege levels are only possible through tightly controlled mechanisms. IBM **SVC** or Intel **INT** are like software interrupts that change to supervisor mode and jump to pre-determined address.

1.3.8 Memory Protection

Virtual memory allows user's memory to be isolated from kernel memory and other users' memory:

- A frame or page may be read or write accessible only to a processor in high privilege level
- In S/370 each frame of memory has a 4-bit storage key and each task runs with a particular key
- The virtual memory mechanism can be extended with permission bits; frames can then be shared
- Combination of all of the above may be used

1.3.9 OS Structure

- **Traditional (Monolithic)**
  - All OS functions sit in the kernel a single function can crash the whole system
- **Micro-kernel**
  - Small core which talks to (maybe privileged) components in separate servers
  - Increase modularity
  - Increase extensibility
  - More overhead
  - Difficult to implement
  - Keep multiple copies of OS data structures

Modern OSes are hybrid: - Linux is monolithic but has (un)loadable modules - Windows started micro-kernel but for performance changed

1.4 Processes

Process is a program in execution. It may have own view of memory, sees one processor although it's sharing it with other processes - **virtual processor**. To switch between processes we need to track:

- its memory, including stack and heap
- contents of registers
- PC
- state

1.4.1 Process States

- **New**: process being created
- **Running**: process being executed on CPU
- **Ready**: not on CPU but ready to be
- **Blocked**: waiting for an event (I/O)
- **Exit**: process finished, awaiting clean-up

- **admit**: process control set up, move to run queue
- **dispatch**: scheduler gives CPU to runnable process
- **time-out/yield**: running process forced to/volunteers to give up CPU
- **event-wait**: waiting for an event (I/O)
- **event**: event occurs - wake up process and tell it
- **release**: process terminates, release resources

1.4.2 Process Control Block (PCB)

- unique process ID
- process state
- PC and other registers
- memory management
- scheduling and memory management info
- list of open files, name of executable, owner, CPU time used so far, devices

1.4.3 Kernel Context

Kernel executes:

- Older OSes: single program in real memory
- Modern OSes: may execute in context of a user process, parts of OSes may be processes (e.g. I/O in Unix and IBM)

1.4.4 Creating Processes

- By the OS when a job is submitted or a user logs on
- By the OS to perform background service for a user (e.g. printing)
- By explicit request from user program

When a process is created, OS must:

- assign unique identifier
- allocate memory space, kernel and user memory
- initialise PCB and memory management tables
- link PCB into OS data structures
- initialise remaining control structures
- WinNT, IBM: load program
- Unix: make child process copy of parent (on write)

1.4.5 Ending Processes

- Terminate voluntarily (e.g. exit())
- Perform illegal operation
- Be killed by user (e.g. kill()) or OS
  - allocated resources exceeded
  - task functionality no longer needed
  - parent terminating

On termination, OS must:

- deal with pending output, etc.
- release all system resources held by the process
- unlink PCB from OS data structures
- reclaim all user and kernel memory

1.4.6 Threads

Processes

- own resources such as address space, I/O devices and files
- are units of scheduling and execution

Threads, however, are allowed to be executed concurrently in one process. Everything said about scheduling applies to threads as well but process-level context is shared by thread contexts.

- creating threads is quick (cca. 10 times faster than process)
- ending threads is quick
- switching threads within process is quick
- inter-thread communication is quick and easy (shared memory)

1.4.7 Thread operations

- **create**: thread spawns a new thread, specifying instruction pointer or routine to call, OS sets up everything
- **block**: thread waits for event - other threads may execute
- **unblock**: event occurs, thread becomes ready
- **finish**: thread completes, context reclaimed

1.4.8 Thread libraries

- thread library implements mini-process scheduler (in user space)
- context of thread is PC, registers, stacks, etc. (in user space)
- thread control block (in user process's memory)
- switching between threads voluntary or on time-out

Advantages

- context-switching is fast (no OS)
- scheduling can be tailored to the application
- library can be OS-independent

Disadvantages

- if thread makes blocking system call, entire process is blocked (there are ways around this)
- user-space threads don't execute concurrently on multi-processor systems

1.5 Multi-processing

Several processors used together

- **Single Instruction Single Data Stream (SISD)**: normal set-up, one processor, one instruction stream, one memory
- **Single Instruction Multiple Data Stream (SIMD)**: a single program executes in lock-step on several processors (large scientific apps)
- **Multiple Instruction Single Data Stream (MISD)**: not used
- **Multiple Instruction Multiple Data Stream (MIMD)**: many processors each execution different programs on different data

Within MIMD, processors could be *loosely coupled* (e.g. network of PCs with communication links) or *tightly coupled* (e.g. processors connected via a single bus).

1.5.1 Symmetric Multi-processing (SMP)

Where does the OS run when multiple processors?

- **master-slave**: kernel runs on one CPU, and dispatches processes to others. All I/O is done by request on kernel CPU. Easy but inefficient and failure prone.
- **symmetric**: the kernel executes on any CPU. Kernel may be multi-process or multi-threaded. Each processor may have its own scheduler. More flexible and efficient - more complex.

SMP OS design considerations

- **cache coherence**: several CPUs, one shared memory. Each CPU has its own cache. Usually solved by hardware designers.
- **re-entrancy**: several CPUs may call kernel simultaneously. Kernel code must be written to handle this.
- **scheduling**: genuine concurrency between threads and kernel threads.
- **memory**: must maintain virtual memory consistency between processors.
- **fault tolerance**: single CPU failure should not influence others.

1.5.2 Scheduling

Happens over several time-scales and at several levels:

- **batch scheduling (long-term)**: which jobs should be started
- **medium-term**: some OSes *suspend* or *swap out* processes to ameliorate resource contention
- **process scheduling (short-term)**: which process gets CPU next, how long

Criteria for scheduling

- good utilisation: minimise amount of CPU idle time and job throughput
- fairness: all jobs should get a 'fair' share of the CPU
- priority: high-priority jobs get larger share
- response time: fast response to interactive input
- real-time: hard deadlines, e.g. chemical plant control
- predictability: avoid wild variations in user-visible performance

Balance is dependent on the system. On a PC, response time is important. On a main-frame throughput is important.

1.5.3 Non-pre-emptive Policies

Once a job gets CPU it keeps it until it yields it or needs e.g. I/O. Suitable for long-term policies, not used for short-term.

- **first-come-first-served (FCFS)**: Favours long and CPU-bound processes over short or I/O bound processes. Used as sub-component of priority systems.
- **shortest-process-next (SPN)**: Dispatch process with shortest expected processing time. Improves overall performance time. Favours short jobs and has poor predictability. To estimate expected time - user can estimate (long-term), build up CPU residency over time (short-term)

1.5.4 Pre-emptive Policies

Processes could be interrupted after some time - **quantum**.

- **round-robin**: When quantum expires, running process sent to the back of the queue. Favours CPU-bound processes - can be refined to avoid. Quantum should be slightly greater than average interaction time (Unix - 50 ms).
- **shortest-remaining-time (SRT)**: Pre-emptive version of SPN. On quantum expiry, dispatch process with shortest expected running time. Tends to starve long CPU-bound processes.
- **feedback**: use dynamically assigned priorities.
  - New process starts in queue with priority 0 (highest).
  - Each time it is pre-empted, goes back to next lower priority queue.
  - Dispatch first process in highest occupied queue.
  - Tends to starve long jobs, possible solutions:
    - Increase quantum for lower priority processes
    - Raise priority for processes that are starved

1.5.5 Multi-processor Scheduling

- Assigning processes to processors
  - static assignment - may have idle CPUs
  - dynamic assignment - complexity increased
- Deciding on multi-programming on each CPU
  - If many CPUs and app parallel at thread-level then maybe don't.
- Dispatching processes

1.5.6 SMP Scheduling

For *process scheduling*, performance analysis and simulation indicate that the differences between scheduling algorithms are reduced in SMP - no need to use complex systems, FCFS or a variant may suffice. However, FCFS has disadvantages:

- single pool of TCBs (like PCB for threads) must be accessed with mutual exclusion - may be a bottleneck
- pre-empted threads are unlikely to be re-scheduled to the same CPU - loses benefit of a CPU cache
- unlikely for program to get its threads running at the same time, opposite could impact performance

For *thread scheduling*, situation more complex - unlike processes, threads often interact. Main approaches are:

- **load sharing**: idle processors selects ready thread from whole pool
  - simplest and most like uni-processing environment
- **gang scheduling**: a gang of related threads are simultaneously dispatched to a set of CPUs
- **dedicated CPUs**: static assignments of threads to CPUs
- **dynamic scheduling**: involve the application in changing number of thread; OS shares CPUs among apps 'fairly'

Most systems use load sharing (with tweaks), some scientific systems use gang scheduling.

1.5.7 Real-Time Scheduling

Real-time systems have deadlines. Theses may be *hard* (necessary for success of a task) or *soft* (if not met, still worth running the task).

Requirements for RT systems:

- **determinism**: need to acknowledge events within pre-determined time
- **responsiveness**: take appropriate action quickly enough
- **user control**: hardness of deadlines and priorities is a matter for user
- **reliability**: systems must fail softly, no panicking, ideally no fails

1.6 Concurrency

Control access to a shared variable: protect each read-write sequence by a *lock* which ensures *mutual exclusion.*

1.6.1 Mutual Exclusion

Allow process to identify *critical sections* where they have exclusive access to a resource.

Requirements:

- mutual exclusion must be enforced
- processes blocking in non-critical section must not interfere with others
- processes wishing to enter critical section must eventually be allowed to do so
- entry to critical section should not be delayed without a cause
- there can be no assumptions about speed or number of processors

Implementation:

- via hardware: special machine instructions
- via OS support: OS provides primitives to call
- via software: entirely by user code

We assume that mutual exclusion exists in hardware, memory access is atomic - only one write/read at a time.

Dead lock - both processes loop forever waiting for the other to finish.

Live lock - both processes run in exact synchrony and keep deferring to each other.

1.6.2 Mutex - Dekker's algorithm

Ensure that one process has priority, so will not defer and give other process priority after performing own critical section.

```
/* using j instead of i hat */
flag[i] = true;
while (flag[j]) {
    if (turn == j) {
        flag[i] = false;
        while (turn == j) {}
        flag[i] = true;
    }
}
/* critical section */
turn = j;
flag[i] = false;
```

1.6.3 Mutex - Peterson's algorithm

Peterson found a more simple and elegant algorithm.

```
/* using j instead of i hat */
flag[i] = true;
turn = j;
while (flag[j] && turn == j) {}
/* critical section */
flag[i] = false;
```

1.6.4 Mutex - Using hardware support

- Uniprocessor: mutex achieved by disabling processes from being interrupted. Used extensively in many OSes. Forbidden to user programs.
- SMP systems: special instruction, e.g. IBM has `TEST AND SET` which reads a bit of memory and then sets it to 1, atomically. Easy mutex, have a variable `token` and process grabs `token` using test-and-set:

  ```
  while (test-and-set(token) == 1) {}
  /* critical section */
  token = 0;
  ```

  This is still busy-waiting, deadlock is possible and if low priority grabs the token, high priority pre-empts and can wait forever.

1.6.5 Semaphores

A semaphore is a special (integer) variable which can be accessed only by the following operations:

- `init(s, n)`: create semaphore and initialise it to non-negative value `n`
- `wait(s)`: semaphore value decremented; if value negative calling process is blocked
- `signal(s)`: semaphore is incremented; if value non-positive one process blocked on `wait` is unblocked.

Traditionally, `P` and `V` are used for `wait` and `signal`.

Semaphore could be:

- strong: waiting process are released FIFO; more useful, generally provided
- weak: no guarantee about the order; not used here

Binary semaphore - takes on values 0 and 1. `wait` decrements 1 to 0 or blocks if 0 already. `signal` unblocks, or increments from 0 to 1 if no blocked processes.

Advantage of semaphores - The mutex problem are confined inside just two system calls. User programs do not need to busy-wait; only the OS busy-waits (for a short time).

Using semaphores

```
wait(s);
/* critical section */
signal(s);
```

When semaphore is initialised to `m` instead of `1` then `m` processes run at the same time.

1.6.6 The Producer-Consumer problem

A *producer* repeatedly puts items into a buffer and a *consumer* takes them out. The problem is to make this work without delaying either party. Solution using two semaphores `init(n, 0)` (tracks number of items in buffer) and `init(s, 1)` (used to lock the buffer):

Producer loop

```
datum = produce();
wait(s); // wait until can add to buffer
/* critical section */
append(buffer, datum);
signal(s); // done with buffer
signal(n); // added another item to be consumed
```

Consumer loop

```
wait(n); // wait for items in buffer
wait(s); // wait until can extract from buffer
/* critical section */
datum = extract(buffer);
signal(s); // done with buffer
consume(datum);
```

1.6.7 Monitor

Semaphores have `wait` and `signal` separated in code - hard to understand. A *monitor* is an object which provides some methods (protected by mutex) so only one process can be 'in the monitor' at a time. Monitor variables are only accessible from monitor methods.

- `cwait(c)`: where `c` is a *condition variable* confined to monitor; process is suspended and the monitor released for another process.
- `csignal(c)`: some process suspended on `c` is released and takes the monitor.

Unlike semaphores, `csignal` does nothing if no process is waiting.

Advantage of monitors - monitors enforce mutex and all the synchronisation is inside the monitor methods where it's easier to find and check.

1.6.8 The Readers/Writers Problem

Resource which can be read by many processes at one but any read must block a write. It can be written by only one process at once, blocking everything else. Can be solved using semaphores. Who has priority?

- Unix file locks
- OS/390 ENQ syscall provides general purpose read/write locks
- Linux kernel uses read/write semaphores internally

1.6.9 Message Passing

Many systems provide message passing services. Processes may `send` and `receive` messages from each other. `send` and `receive` may be blocking or non-blocking when there is no receiver waiting or no message to receive. Most usual is non-blocking `send` and blocking `receive`.

Can be used for mutex and synchronisation:

- simple mutex by using a single message as a *token*
- producer/consumer: producer sends data as messages to consumer; consumer sends null messages to acknowledge them

1.6.10 Deadlock

Permanent blocking of two or more processes in a situation where each holds a resource the other needs but will not release it until after obtaining the other's resource.

Process P

```
acquire(A);
acquire(B);
release(A);
release(B);
```

Process Q

```
acquire(B);
acquire(A);
release(B);
release(A);
```

Another instance is when two processes are each waiting for the other to send a message.

Preventing a deadlock

3 facts need to be true for deadlock to happen

- resources are held by only one process at a time
- a resource can be held while waiting for another
- processes do not unwillingly lose resources

If any of these does not hold, deadlock does not happen. If they are true, deadlock may happen if

- a circular dependency arises between resource requests

The first three can be prevented from holding but not practically. The fourth can be prevented by ordering resources and requiring processes to acquire resources in increasing order.

Avoiding a deadlock

A more refined approach is to deny resource requests that might lead to a deadlock. This requires processes to declare in advance the maximum resource they might need. Then, when a process requests a resource, *analyse* whether granting the request might result in a deadlock.

The analysis is done as follows: if we grant the request, is there sufficient resource to allow one process to run to completion? And when it finishes can we run another? And so on.. If not, we should deny the request.

Deadlock detection

There are techniques to detect whether a deadlock exists. Then we can:

- kill all deadlocked processes
- selectively kill deadlocked processes
- forcibly remove resources from some processes
- if checkpoint-restart is available, roll back to pre-deadlock point and hope it doesn't happen again

1.7 Memory Management