1 History of computers

TL;DR

2 Computer Architecture

2.1 The Von Neumann Architecture

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.

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

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

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

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

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.

2.7.1 Bus Hierarchy

2.8 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.

2.9 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.

3 Operating System function

- 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

3.1 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

3.2 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

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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

3.3 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

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

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

4.1.1 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

4.2 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)

4.3 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)

4.4 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

4.5 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)

4.5.1 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

4.5.2 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

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).

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.

5.1.1 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.

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

5.2.1 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.

5.2.2 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)

5.2.3 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

5.2.4 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

5.2.5 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.

5.2.6 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

6 Concurrency

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

6.1 Mutual Exclusion

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

6.1.1 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

6.1.2 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.

6.1.3 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;
```

6.1.4 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;
```

6.1.5 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.

6.2 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.

6.2.1 Advantage of semaphores

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

6.2.2 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.

6.2.3 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);
```

6.3 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.

6.3.1 Advantage of monitors

Monitors enforce mutex and all the synchronisation is inside the monitor methods where it’s easier to find and check.

6.3.2 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

6.4 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

6.5 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.

6.5.1 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.

6.5.2 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.

6.5.3 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

7 Memory Management

Each process needs memory for

- code: program itself
- static data: compiled into program
- dynamic data: stack, heap

Key requirements

- relocation: moving programs in memory
- allocation: assigning memory for processes
- protection: preventing access to other processes' memory
- sharing: prevent, except for where appropriate
- logical organisation: how memory is seen by process
- physical organisation: how memory is arranged in HW

Relocation problem

When we load contents of a static variable into a register where is variable in memory? When branch, where branch?

Compiler can tag all memory references and make them relative to start of program. Then *relocating header* loads program at location `X` and adds `X` to all memory addresses in the program. Expensive and what if program swapped out and brought in elsewhere.

We could provide hardware instructions that access memory relative to a *base register* and have programmer use these. Program then sets base register but nothing else. E.g. in S/390 a typical instruction is:

```
L R13, 568(R12)
```

which loads register 13 with value in address contained in register 12 offset by 568. This could be done by hardware and OS instead of the programmer.

7.1 Segmentation

A segment is a portion of memory starting at an address given in a *base register* `B`. The OS loads a value `b` into `B`. When the program refers to a memory address `x`, hardware transparently translates it into `x + b`.

To achieve protection, we can add *limit register* `L`. OS loads `L` with length of segment `l`. If `x > l` then raise *address fault* (exception SEGFAULT).

7.1.1 Partitioning

Segmentation allows programs to be put into any available chunk of memory. How do we partition memory between various processes?

- **fixed partitioning**: divide memory into fixed chunks. Disadvantage: small process in large chunk is wasteful
- **dynamic partitioning**: load process into a suitable chunk; when exits, free chunk and merge with neighbouring free chunks. Disadvantage: *(external) fragmentation* - memory tends to get split into small chunks. May need to *swap out* running process to make room for higher priority new process. How do we choose chunks?
  - first fit: choose first big enough chunk (generally best)
  - next fit: choose first big enough chunk after last allocated chunk (fragments more)
  - best fit: choose chunk with least waste (fragments a lot)

7.1.2 Partitioning - The Buddy System

Compromise between static and dynamic.

- Memory is maintained as a binary tree of blocks of size 2^k for $L \leq k \leq U$ suitable lower and upper bounds
- When process of size s, $2^{i-1} < s \leq 2^i$ comes in, look for free block of size 2^i. If none, find (recursively) block of size 2^{i+1} and split it in two.
- When blocks are freed, merge free sibling nodes ("buddies") to re-create bigger blocks.

Variants on the buddy system are still used, e.g. in allocating memory within the Linux kernel.

7.1.3 Multiple Segments

We can extend segmentation to have multiple segments for a program:

- hardware/OS provide different segments for different types of data e.g. code, static data and dynamic data.
- hardware/OS provides multiple segments at user request
  - logical memory address viewed as a pair `(s, o)`
  - process has *segment table*: look up entry `s` in table to get base and limit by b and l
  - translate as normal to `o + b` or raise fault if `o + b > l`

7.1.4 Advantages of Segmentation

- may correspond to user view of memory
- protection can be done per segment, each segment can be protected agains, e.g. read, write, execute
- sharing of code/data easy but it is better to have a single list of segment descriptors and have process segment tables point into that than to duplicate info between processes

7.1.5 Disadvantages of Segmentation

- variable size segments lead to external fragmentation, again
- may need to *compact* memory due to fragmentation
- small segments tend to minimise fragmentation but annoy the programmer

7.2 Paging

Small segments reduce fragmentation; variable size segments introduce problems.

Paging - have many small fixed-size segments always provided (invisibly) to the programmer.

Virtual storage is divided in *pages* of fixed size (typically 4KB). Each *page* is mapped to a *frame* of real storage by means of a *page table*.

- **Page**: virtual memory unit
- **Frame**: real memory unit
- **Page table**: translates pages to frames
  - includes *valid bit* since not all pages may have frames
  - start and length of a page table are held in control registers like in segmentation
  - may include *protection bit(s)* in page table entry for read, write, execute

7.2.1 Translation Lookaside Buffer (TLB)

With paging (or segmentation) each logical memory reference needs two physical memory references. A TLB is a special associative cache for keeping recently used paging information - it maps pages into frames, bypassing the page table. Like all caches, TLB has coherency problem:

- when process context-switches, active page table changes must flush TLB
- when page is freed must invalidate entry in TLB
- changes in protection bits must also invalidate TLB entry

7.2.2 Multi-level Paging

Modern systems have address space of at least $2^3$1 bytes or $2^1$9 4K pages. Modern systems have two or more levels of page table

7.2.3 Sharing Pages

Memory can be shared by having different pages map to the same frame. For code we need re-entrant code (not self-modifying). Otherwise, use copy on write:

- mark the pages as read-only in each process (using protection bits)
- when process writes, generate protection exception
- OS handles exception by allocating new frame, copying shared frame, and updating process's page table

7.2.4 Virtual Memory

Pages don't have to be in real memory all the time. Store them on disk when not needed!

- initialise process's page table with invalid entries
- on first reference to page, get exception: handle it, allocate frame, update page table entry
- when real memory gets tight, choose some pages, write them to disk, invalidate them and free frames for use elsewhere
- when process refers to page on disk, get exception: handle by reading from disk (if necessary paging out some other page)

OSes often use frame-address portion of invalid page table entry to keep its location on disk.

Hardware support for VM usually include:

- modified bit: no need to write out page if not changed since last read in
- reference bit or counter: unreferenced pages are candidates for freeing

Architectures differ on where this happens:

- Intel: modified and reference bits are part of page table entry
- S/390: they are part of storage key associated with each real frame

7.2.5 Combined Paging and Segmentation

- S/390
  - Intertwined, can be seen as 2-level paging system
  - Logical address is 31 bits
  - First 11 bits index into current segment table
  - Next 8 bits index into page table
  - Remaining bits are offset

  Page tables can be paged out, by marking their entries invalid in the segment table (one segment table per process).
- Intel
  - Full-blown segmentation and independent paging
  - Logical address is 16-bit segment id and 32-bit offset
  - Segment id portion of logical address is found via a segment register which is usually implicit in access type (`CS` for instruction, `DS` for data, `SS` for stack, `ES` for string data) but can be specified to be in any of six segment registers
  - Segment registers are part of task context
  - There may be single global segment table but also task-specific tables

  The result of segment translation is 32-bit linear address which completely independently goes through a 2-level paging system.

  - Segment related info (e.g. segment tables) can be paged out; so can 2nd-level page tables
  - There is no link between pages and segments: segments don't need to lie on page boundaries
  - Pages can be 4KB or 4MB
  - Page table register is part of task context, stored in task segment.

7.2.6 Paging Policies

In a virtual memory system, OS needs to decide what to page in and out

- minimise the number of *page faults*: avoid paging out pages that will be needed
- minimise disk I/O: avoid *reclaiming* dirty pages

Fetch Policies

When should a page be brought into main memory from disk?

- demand paging: when referenced - locality principle suggests this should work well after an initial burst of activity
- pre-paging: try to bring in pages ahead of demand, exploiting characteristics of disk to improve efficiency
  - not shown to be effective
  - used little, if at all
  - few years ago it became a live issue again with a study suggesting it could be useful now

Replacement Policy

When memory runs out and a page is brought in, who do we throw out? Aim: page out the page with the longest time until next reference.

- LRU: choose the page with the longest time since last reference. This is almost optimal - high overhead.
- FIFO: simple but pages out heavily-used pages
- clock policy: attempts to get some of the performance of LRU without the overhead
  - Makes use of the *accessed bit*
  - Put frames in a circular list 0, . . . , `n` - 1 with index `i`
  - When looking for a page replace, do

    ```
    increment i;
    while (frame i used) {
        clear use bit on frame i;
        increment i;
    }
    return i;
    ```

  - Doesn't choose a page unless it has been unreferenced for one complete pass through storage. Performs reasonably well, 25% worse than LRU.
  - Enhancement to reduce I/O: scan only unmodified frames, without clearing use bit. If this fails, scan modified frames, clearing use bit. If this fails, start from beginning.

7.2.7 Page Caching

Many OSes use the clock policy with caches and buffers:

- When a page is replaced, it's added to the end of the *free page list* if clear, or the `modified page list` if dirty.
- The actual frame used for the paged-in page is the head of the free page list.
- If no free pages, or when modified list gets beyond certain size, write out modified pages and move to free list.

This means that:

- pages in the cache can be instantly restored if referenced again
- I/O is batched and therefore more efficient

Linux allows you to tune various parameters of the paging caches. It also has a background kernel thread that handles actual I/O this 'trickles out' pages to keep a certain amount of memory free most of the time, to make allocation fast.

7.2.8 Resident Set Management

In the previous schemes, when a process *page faults*, some other process's page may be thrown out. Alternatively, we could have a *resident set* for each process.

- allocate a certain number of frames to each process
- after a process reaches its allocation, if a page faults, choose some page of that process to reclaim
- re-evaluate resident set size (RSS) from time to time

How to choose RSS

The working set of a process over time Δ is the set of pages referenced in the last Δ time units. aim to keep the working set in memory. Working sets tend to be stable for some time (locality) and change to a new stable set every so often (inter-locality transitions).

Actually tracking the working set is expensive. Some approximations:

- **page fault frequency**: choose threshold frequency `f`. On page fault:
  - if virtual time since last fault is $<$ 1/`f`, add one page to RSS
  - otherwise, discard unreferenced pages and shrink RSS; clear use bits on other pages
  - works quite well but poor performance in inter-locality transitions
- **variable-interval sampled working set**: at intervals,
  - evaluate working set (clear use bits at start, check at end)
  - make this the initial resident set for next interval
  - add any faulted-in pages (shrink RSS between intervals)
  - the interval is every `Q` page faults subject to lower and upper time bounds `L` and `U`
  - Tune `Q`, `L` and `U` according to experience

8 Input/Output