Operating Systems II

Steven Hand

Michaelmas Term 2005

8 lectures for CST IB

Course Aims

This course aims to:

- impart a detailed understanding of the algorithms and techniques used within operating systems,
- consolidate the knowledge learned in earlier courses, and
- help students appreciate the trade-offs involved in designing and implementing an operating system.

Why another operating sytems course?

- OSes are some of the largest software systems around ⇒ illustrate many s/w engineering issues.
- OSes motivate most of the problems in concurrency
- more people end up 'writing OSes' than you think
  - modifications to existing systems (e.g. linux)
  - embedded software for custom hardware
  - research operating systems lots of fun
- various subsystems not covered to date. . .

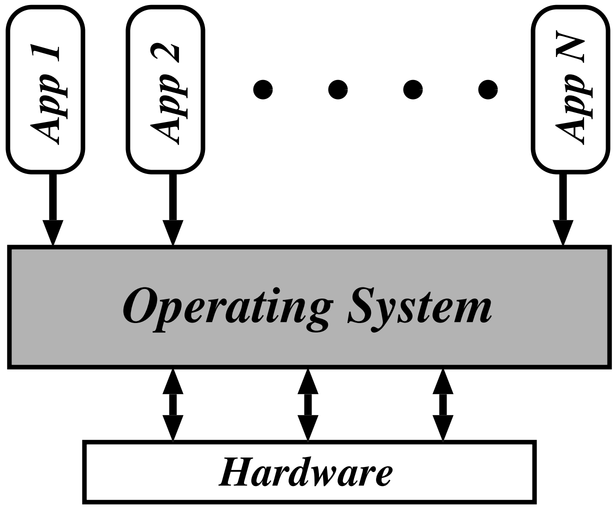
Course Outline

- Introduction and Review:
  OS functions & structures. Virtual processors (processes and threads).
- CPU Scheduling.
  Scheduling for multiprocessors. RT scheduling (RM, EDF). SRT/multimedia scheduling.
- Memory Management.
  Virtual addresses and translation schemes. Demand paging. Page replacement. Frame allocation. VM case studies. Other VM techniques.
- Storage Systems.
  Disks & disk scheduling. Caching, buffering. Filing systems. Case studies (FAT, FFS, NTFS, LFS).
- Protection.
  Subjects and objects. Authentication schemes. Capability systems.
- Conclusion.
  Course summary. Exam info.

Recommended Reading

- Bacon J [and Harris T]
  Concurrent Systems or *Operating Systems*
  Addison Wesley 1997, Addison Wesley 2003
- Tannenbaum A S
  Modern Operating Systems (2nd Ed)
  Prentice Hall 2001
- Silberschatz A, Peterson J and Galvin P
  Operating Systems Concepts (5th Ed)
  Addison Wesley 1998
- Leffler S J
  The Design and Implementation of the 4.3BSD UNIX Operating System.
  Addison Wesley 1989
- Solomon D [and Russinovich M]
  Inside Windows NT (2nd Ed) or *Inside Windows 2000 (3rd Ed)*
  Microsoft Press 1998, Microsoft Press 2000
- OS links (via course web page)
  `http://www.cl.cam.ac.uk/Teaching/current/OpSysII/`

Operating System Functions



An operating system is a collection of software which:

- *securely multiplexes resources*, i.e.
  - protects applications from each other, yet
  - shares physical resources between them.
- provides an abstract *virtual machine*, e.g.
  - time-shares CPU to provide virtual processors,
  - allocates and protects memory to provide per-process virtual address spaces,
  - presents h/w independent virtual devices.
  - divides up storage space by using filing systems.

And ideally it does all this *efficiently* and *robustly*.

Hardware Support for Operating Systems

Recall that OS should *securely* multiplex resources.
⇒ we need to ensure that an application cannot:

- compromise the operating system.
- compromise other applications.
- deny others service (e.g. abuse resources)

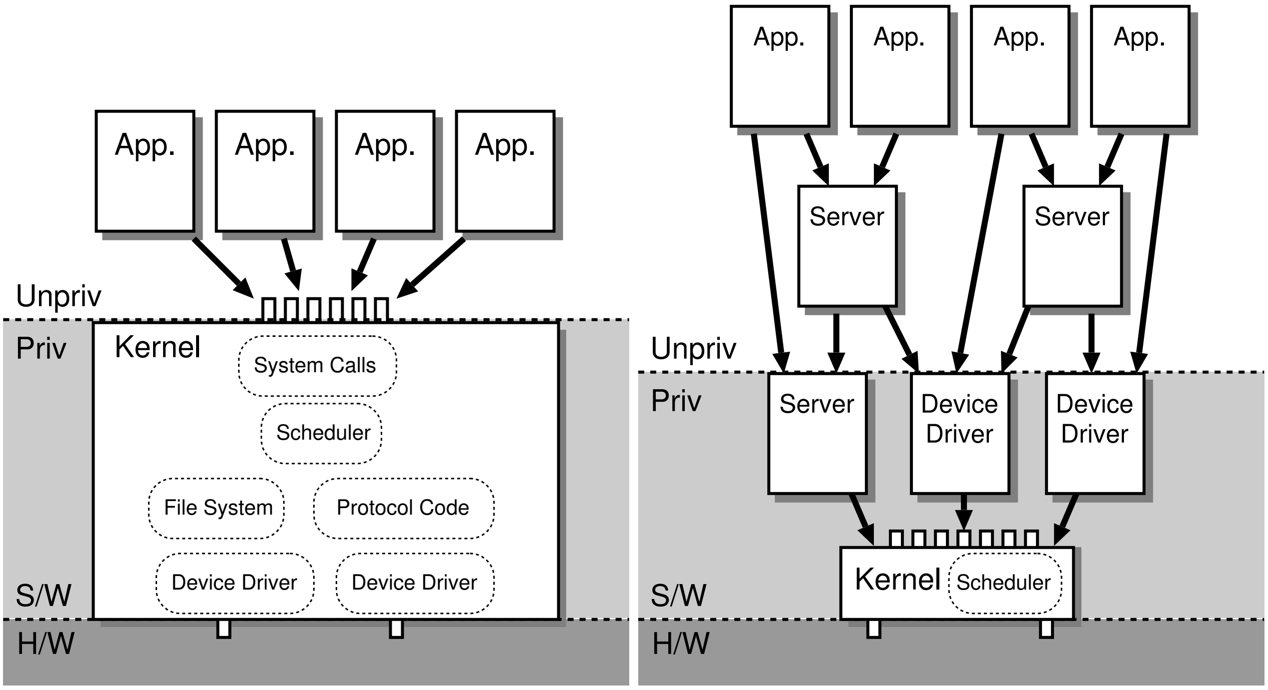
To achieve this efficiently and flexibly, we need hardware support for (at least) *dual-mode operation*.

Then we can:

- add memory protection hardware
  ⇒ applications confined to subset of memory;
- make I/O instructions privileged
  ⇒ applications cannot directly access devices;
- use a *timer* to force execution interruption
  ⇒ OS cannot be starved of CPU.

Most modern hardware provides protection using these techniques (c/f Computer Design course).

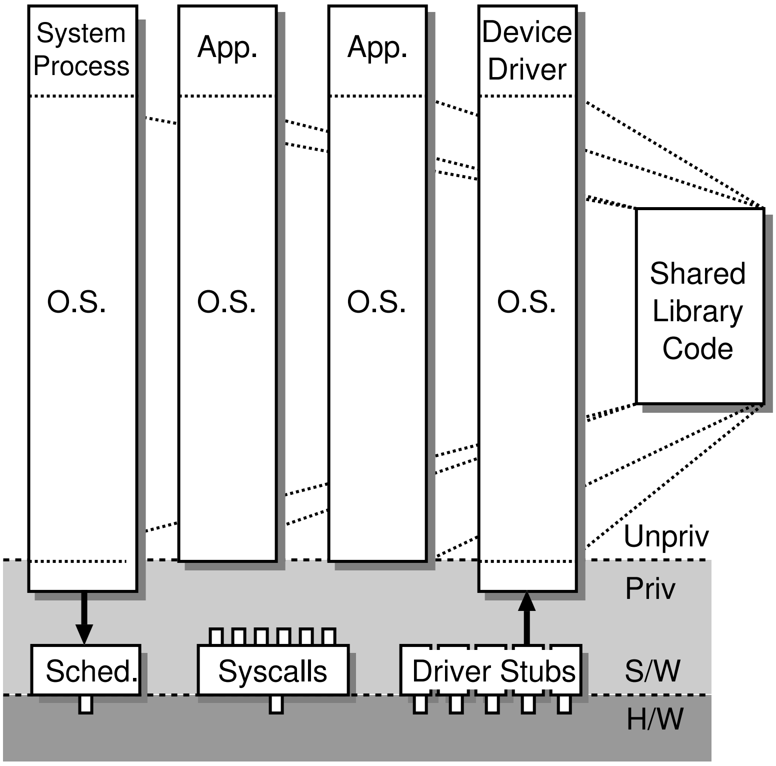
Operating System Structures



Traditionally have had two main variants:

1. Kernel-based (*lhs* above)
   - set of OS services accessible via software interrupt mechanism called *system calls*.
2. Microkernel-based (*rhs* above)
   - push various OS services into *server* processes
   - access servers via some *interprocess communication* (IPC) scheme.
   - increased modularity (decreased performance?)

Vertically Structured Operating Systems



- Consider interface people really see, e.g.
  - set of programming libraries / objects.
  - a command line interpreter / window system.
- Separate concepts of protection and abstraction ⇒ get extensibility, accountability & performance.
- Examples: Nemesis, Exokernel, Cache Kernel.

We'll see more on this next year. . .

Virtual Processors

Why virtual processors (VPs) ?

- to provide the illusion that a computer is doing more than one thing at a time;
- to increase system throughput (i.e. run a thread when another is blocked on I/O);
- to encapsulate an execution context;
- to provide a simple programming paradigm.

VPs implemented via $processes$ and $threads$:

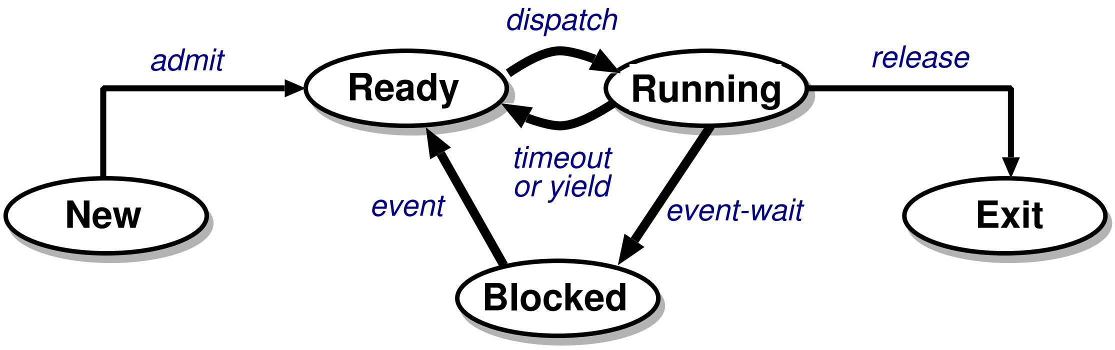
- A process (or task) is a unit of resource ownership — a process is allocated a virtual address space, and control of some resources.
- A thread (or lightweight process) is a unit of dispatching — a thread has an execution state and a set of scheduling parameters.
- OS stores information about processes and threads in process control blocks (PCBs) and thread control blocks (TCBs) respectively.
- In general, have 1 process $\leftrightarrow n$ threads, $n \geq 1$ $\Rightarrow$ PCB holds references to one or more TCBs.

Thread Architectures

- User-level threads
  - Kernel unaware of threads' existence.
  - Thread management done by application using an unpriviliged *thread library*.
  - Pros: lightweight creation/termination; fast ctxt switch (no kernel trap); application-specific scheduling; OS independence.
  - Cons: non-preemption; blocking system calls; cannot utilise multiple processors.
  - e.g. FreeBSD `pthreads`
- Kernel-level threads
  - All thread management done by kernel.
  - No thread library (but augmented API).
  - Scheduling can be two-level, or direct.
  - Pros: can utilise multiple processors; blocking system calls just block thread; preemption easy.
  - Cons: higher overhead for thread mgt and context switching; less flexible.
  - e.g. Windows NT/2K, Linux (?).

Hybrid schemes also exist. . . (see later)

Thread Scheduling Algorithms



A scheduling algorithm is used to decide which ready thread(s) should run (and for how long).

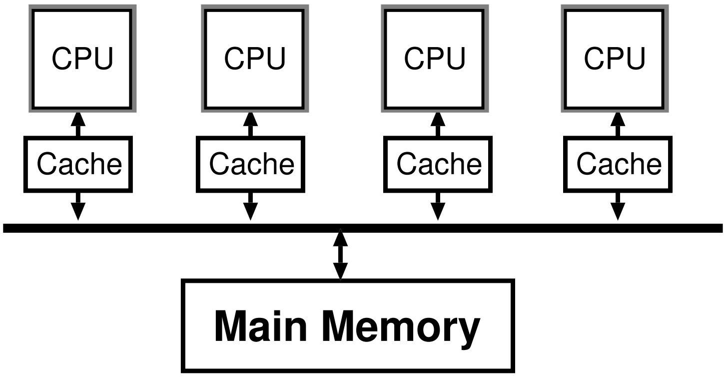
Typically use *dynamic priority scheduling*:

- each thread has an associated [integer] priority.
- to schedule: select highest priority `ready` thread(s)
- to resolve ties: use round-robin within priority
  - different quantum per priority?
  - CPU bias: per-thread quantum adaption?
- to avoid starvation: dynamically vary priorities
- e.g. BSD Unix: 128 pris, 100ms fixed quantum, load- and usage-dependent priority damping.
- e.g. Windows NT/2K: 15 dynamic pris, adaptive ∼20ms quantum; priority boosts, then decays.

Multiprocessors

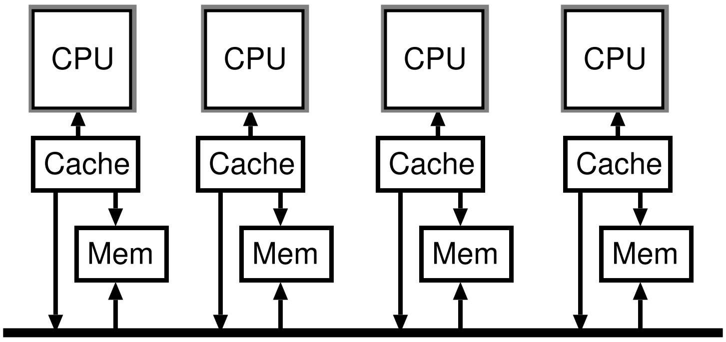
Two main kinds of [shared-memory] multiprocessor:

1. Uniform Memory Access (UMA), aka SMP.



- all (main) memory takes the same time to access
- scales only to 4, 8 processors.

2. Non-Uniform Memory Access (NUMA).



- rarer and more expensive
- can have 16, 64, 256 CPUs . . .

Whole area becoming more important. . .

Multiprocessor Operating Systems

Multiprocessor OSes may be roughly classed as either *symmetric* or *asymmetric*.

- Symmetric Operating Systems:
  - identical system image on each processor
    ⇒ convenient abstraction.
  - all resources directly shared
    ⇒ high synchronisation cost.
  - typical scheme on SMP (e.g. Linux, NT).
- Asymmetric Operating Systems:
  - partition functionality among processors.
  - better scalability (and fault tolerance?)
  - partitioning can be static or dynamic.
  - common on NUMA (e.g. Hive, Hurricane).
  - NB: asymmetric $\not\Rightarrow$ trivial "master-slave"
- Also get hybrid schemes, e.g. Disco:
  - (re-)introduce *virtual machine monitor*
  - can fake out SMP (but is this wise?)
  - can run multiple OSes simultaneously. . .

Multiprocessor Scheduling (1)

- Objectives:
  - Ensure all CPUs are kept busy.
  - Allow application-level parallelism.
- Problems:
  - Preemption within critical sections:
    - $*$ thread $\mathcal{A}$ preempted while holding spinlock.
    - $\Rightarrow$ other threads can waste many CPU cycles.
    - $*$ similar situation with producer/consumer threads (i.e. wasted schedule).
  - Cache pollution:
    - $*$ if thread from different application runs on a given CPU, lots of compulsory misses.
    - $*$ generally, scheduling a thread on a new processor is expensive.
    - $*$ (can get degradation of factor or 10 or more)
  - Frequent context switching:
    - $*$ if number of threads greatly exceeds the number of processors, get poor performance.

Multiprocessor Scheduling (2)

Consider basic ways in which one could adapt uniprocessor scheduling techniques:

- Central Queue:
  - ✔ simple extension of uniprocessor case.
  - ✔ load-balancing performed automatically.
  - ✘ n-way mutual exclusion on queue.
  - ✘ inefficient use of caches.
  - ✘ no support for application-level parallelism.
- Dedicated Assignment:
  - ✔ contention reduced to thread creation/exit.
  - ✔ better cache locality.
  - ✘ lose strict priority semantics.
  - ✘ can lead to load imbalance.

Are there better ways?

Multiprocessor Scheduling (3)

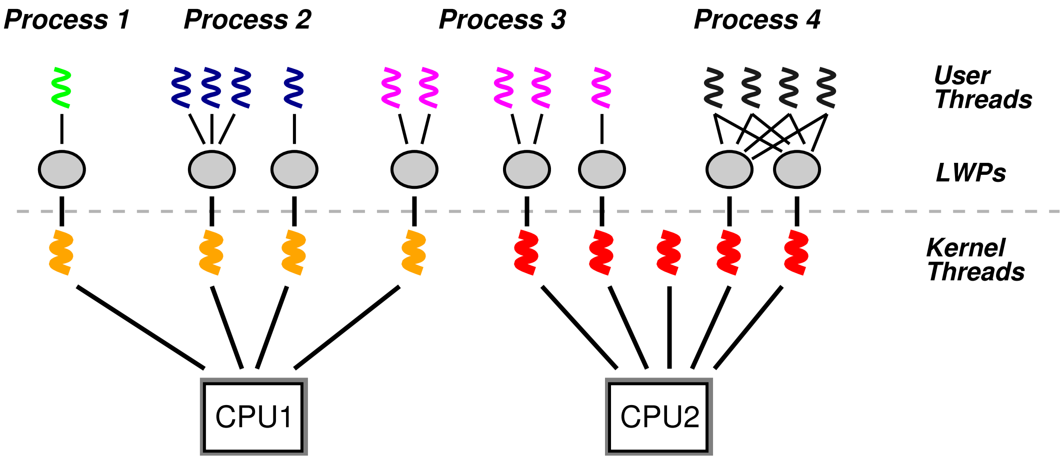
- Processor Affinity:
  - modification of central queue.
  - threads have *affinity* for a certain processor ⇒ can reduce cache problems.
  - but: load balance problem again.
  - make dynamic? (cache affinity?)
- ‘Take’ Scheduling:
  - pseudo-dedicated assignment: idle CPU “takes” task from most loaded.
  - can be implemented cheaply.
  - nice trade-off: load high ⇒ no migration.
- Co-scheduling / Gang Scheduling:
  - Simultaneously schedule “related” threads.

  ⇒ can reduce wasted context switches.
  - Q: how to choose members of gang?
  - Q: what about cache performance?

Example: Mach

- Basic model: dynamic priority with central queue.
- Processors grouped into disjoint *processor sets*:
  - Each processor set has 32 shared ready queues (one for each priority level).
  - Each processor has own local ready queue: absolute priority over global threads.
- Increase quantum when number of threads is small
  - 'small' means #threads $<$ $(2\times$ #CPUs)
  - idea is to have a sensible *effective quantum*
  - e.g. 10 processors, 11 threads
    - ∗ if use default 100ms quantum, each thread spends an expected 10ms on runqueue
    - ∗ instead stretch quantum to 1s ⇒ effective quantum is now 100ms.
- Applications provide *hints* to improve scheduling:
  1. discouragement hints: mild, strong and absolute
  2. handoff hints (aka "yield to") — can improve producer-consumer synchronization
- Simple gang scheduling used for allocation.

MP Thread Architectures



Want benefits of both user and kernel threads without any of the drawbacks ⇒ use hybrid scheme.

E.g. Solaris 2 uses *three-level scheduling*:

- 1 kernel thread ↔ 1 LWP ↔ n user threads
- user-level thread scheduler ⇒ lightweight & flexible
- LWPs allow potential multiprocessor benefit:
  - more LWPs ⇒ more scope for true parallelism
  - LWPs can be *bound* to individual processors ⇒ could in theory have user-level MP scheduler
  - kernel scheduler is relatively cache agnostic (although have processor sets (≠ Mach's)),

Overall: either *first-class threads* (Psyche) or *scheduler activations* probably better for MP.

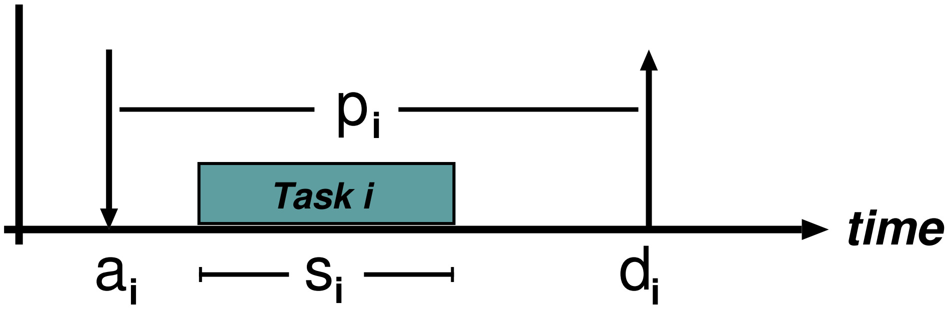
Real-Time Systems

- Need to both produce correct results **and** meet predefined deadlines.
- "Correctness" of output related to time delay it requires to be produced, e.g.
  - nuclear reactor safety system
  - JIT manufacturing
  - video on demand
- Typically distinguish hard real-time (HRT) and soft real-time (SRT):

  HRT — output value = 100% before the deadline, 0 (or less) after the deadline.

  SRT — output value = 100% before the deadline, (100 - $f(t)$)% if t seconds late.
- Building such systems is all about *predictability*.
- It is *not* about speed.

Real-Time Scheduling



- Basic model:
  - consider set of tasks T_i, each of which arrives at time a_i and requires s_i units of CPU time before a (real-time) deadline of d_i.
  - often extended to cope with *periodic* tasks: require s_i units every p_i units.
- Best-effort techniques give no predictability
  - in general priority specifies *what* to schedule but not *when* or *how much*.
  - i.e. CPU allocation for thread t_i, priority p_i depends on all other threads at t_j s.t. $p_j \geq p_i$.
  - with dynamic priority adjustment becomes even more difficult.

⇒ need something different.

Static Offline Scheduling

Advantages:

- Low run-time overhead.
- Deterministic behaviour.
- System-wide optimization.
- Resolve dependencies early.
- Can prove system properties.

Disadvantages:

- Inflexibility.
- Low utilisation.
- Potentially large schedule.
- Computationally intensive.

In general, offline scheduling only used when determinism is the overriding factor, e.g. MARS.

Static Priority Algorithms

Most common is Rate Monotonic (RM)

- Assign static priorities to tasks off-line (or at 'connection setup'), high-frequency tasks receiving high priorities.
- Tasks then processed with no further rearrangement of priorities required ($\Rightarrow$ reduces scheduling overhead).
- Optimal, static, priority-driven alg. for preemptive, periodic jobs: i.e. no other static algorithm can schedule a task set that RM cannot schedule.
- Admission control: the schedule calculated by RM is always feasible if the total utilisation of the processor is less than $ln2$
- For many task sets RM produces a feasible schedule for higher utilisation (up to $\sim 88\%$); if periods harmonic, can get 100%.
- Predictable operation during transient overload.

Dynamic Priority Algorithms

Most popular is Earliest Deadline First (EDF):

- Scheduling pretty simple:
  - keep queue of tasks ordered by deadline
  - dispatch the one at the head of the queue.
- EDF is an optimal, dynamic algorithm:
  - it may reschedule periodic tasks in each period
  - if a task set can be scheduled by any priority assignment, it can be scheduled by EDF
- Admission control: EDF produces a feasible schedule whenever processor utilisation is $\leq$ 100%.
- Problem: scheduling overhead can be large.
- Problem: if system overloaded, all bets are off.

Notes:

1. Also get least slack-time first (LSTF):
   - similar, but not identical
   - e.g. A: 2ms every 7ms; B: 4ms every 8ms
2. RM, EDF, LSTF all *preemptive* (c/f P3Q7, 2000).

Priority Inversion

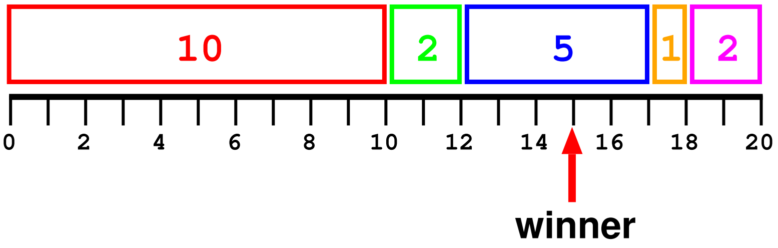
- All priority-based schemes can potentially suffer from *priority inversion*:
- e.g. consider low, medium and high priority processes called P_l, P_m and P_h respectively.
  1. first P_l admitted, and locks a semaphore $\mathcal{S}$.
  2. then other two processes enter.
  3. P_h runs since highest priority, tries to lock $\mathcal{S}$ and blocks.
  4. then P_m gets to run, thus preventing P_l from releasing $\mathcal{S}$, and hence P_h from running.
- Usual solution is *priority inheritence*:
  - associate with every semaphore $\mathcal{S}$ the priority P of the highest priority process waiting for it.
  - then temporarily boost priority of *holder* of semaphore up to P.
  - can use handoff scheduling to implement.
- NT "solution": priority boost for CPU starvation
  - checks if $\exists$ ready thread not run $\geq$ 300 ticks.
  - if so, doubles quantum & boosts priority to 15

Multimedia Scheduling

- Increasing interest in multimedia applications (e.g. video conferencing, mp3 player, 3D games).
- Challenges OS since require presentation (or processing) of data in a timely manner.
- OS needs to provide sufficient *control* so that apps behave well under contention.
- Main technique: exploit SRT scheduling.
- Effective since:
  - – the value of multimedia data depends on the timeliness with which it is presented/processed.
  - ⇒ real-time scheduling allows apps to receive sufficient and timely resource allocation to handle their needs even when the system is under heavy load.
  - – multimedia data streams are often somewhat tolerant of information loss.
  - ⇒ informing applications and providing *soft* guarantees on resources are sufficient.
- Still ongoing research area. . .

Example: Lottery Scheduling (MIT)

total = 20
random [1..20] = 15

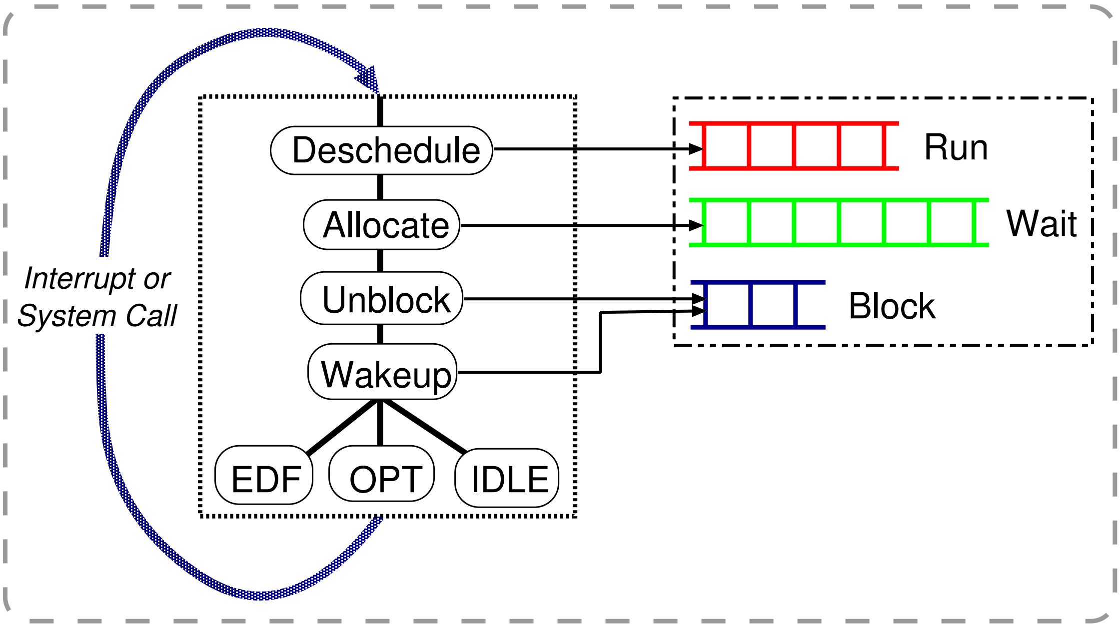


- Basic idea:
  - distribute set of *tickets* between processes.
  - to schedule: select a ticket [pseudo-]randomly, and allocate resource (e.g. CPU) to the winner.
  - approximates *proportional share*
- Why would we do this?
  - simple uniform abstraction for resource management (e.g. use tickets for I/O also)
  - “solve” priority inversion with ticket transfer.
- How well does it work?
  - $\sigma^2 = np(1-p) \Rightarrow$ accuracy improves with $\sqrt{n}$
  - i.e. “asymptotically fair”
- Stride scheduling much better. . .

Example: BVT (Stanford)

- Lottery/stride scheduling don't explicitly support *latency sensitive* threads:
  - some applications don't require much CPU but need it at the right time
  - e.g. MPEG player application requires CPU every frame time.
- *Borrowed Virtual Time* (BVT) uses a number of techniques to try to address this.
  - execution of threads measured in *virtual time*
  - for each thread t_i maintain its *actual virtual time* A_i and *effective virtual time* E_i
  - $E_i = A_i - (warpOn?W_i : 0)$, where W_i is the *virtual time warp* for t_i
  - always run thread with lowest E_i
  - after thread runs for Δ time units, update $A_i \leftarrow A_i + (\Delta \times m_i)$
  - (m_i is an inversely proportional weight)
  - also have *warp time limit* L_i and *unwarp time requirement* U_i to prevent pathological cases
- Overall: lots of confusing parameters to set. . .

Example: Atropos (CUCL)



- Basic idea:
  - use EDF with implicit deadlines to effect proportional share over explicit timescales
  - if no EDF tasks runnable, schedule best-effort
- Scheduling parameters are (s, p, x) :
  - requests s milliseconds per p milliseconds
  - x means “eligible for slack time”
- Uses explicit admission control
- Actual scheduling is easy (∼200 lines C)

Virtual Memory Management

- Limited physical memory (DRAM), need space for:
  - operating system image
  - processes (text, data, heap, stack, . . .)
  - I/O buffers
- Memory management subsystem deals with:
  - Support for address binding (i.e. loading, dynamic linking).
  - Allocation of limited physical resources.
  - Protection & sharing of ‘components’.
  - Providing convenient abstractions.
- Quite complex to implement:
  - processor-, motherboard-specific.
  - trade-offs keep shifting.
- Coming up in this section:
  - virtual addresses and address translation,
  - demand paged virtual memory management,
  - 2 case studies (Unix and VMS), and
  - a few other VM-related issues.

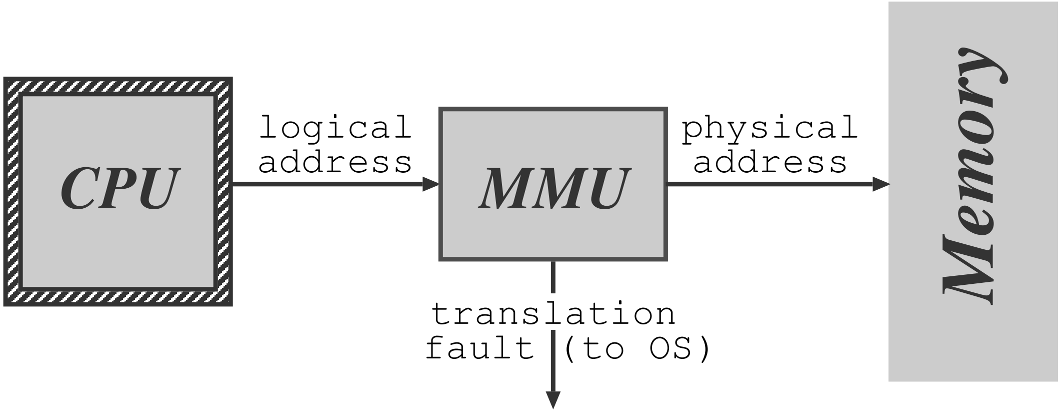
Logical vs Physical Addresses (1)

Old systems directly accessed [physical] memory, which caused some problems, e.g.

- Contiguous allocation:
  - need large lump of memory for process
  - with time, get [external] fragmentation
  - ⇒ require expensive compaction
- Address binding (i.e. dealing with *absolute* addressing):
  - "`int x; x = 5;`" → "`movl $0x5, ????`"
  - compile time ⇒ must know load address.
  - load time ⇒ work every time.
  - what about swapping?
- Portability:
  - how much memory should we assume a "standard" machine will have?
  - what happens if it has less? or more?

Can avoid lots of problems by separating concept of *logical* (or virtual) and *physical* addresses.

Logical vs Physical Addresses (2)



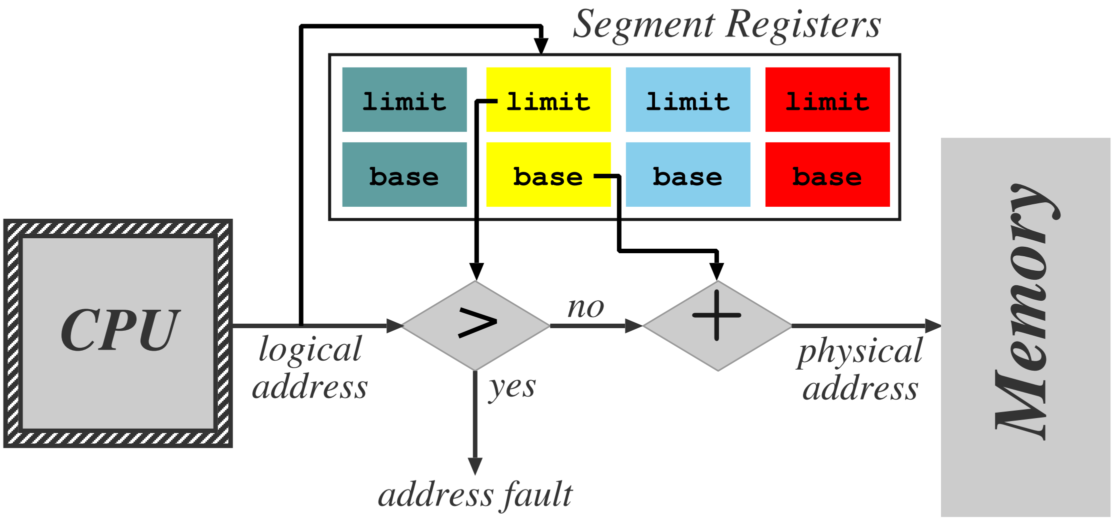
Run time mapping from logical to physical addresses performed by special hardware (the MMU).

If we make this mapping a *per process* thing then:

- Each process has own *address space*.
- Allocation problem split:
  - virtual address allocation easy.
  - allocate physical memory ‘behind the scenes’.
- Address binding solved:
  - bind to logical addresses at compile-time.
  - bind to real addresses at load time/run time.

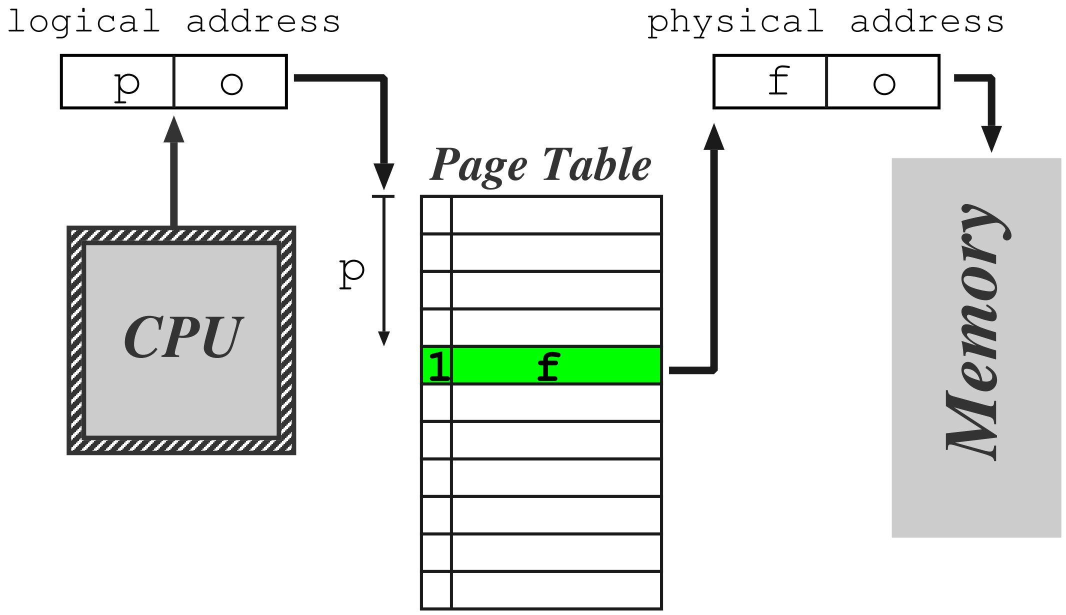
Two variants: segmentation and paging.

Segmentation



- MMU has a set (≥ 1) of segment registers (e.g. orig x86 had four: `cs`, `ds`, `es` and `ss`).
- CPU issues tuple (s, o):
  1. MMU selects segment s.
  2. Checks $o \leq$ `limit`.
  3. If ok, forwards `base`$+o$ to memory controller.
- Typically augment translation information with *protection bits* (e.g. read, write, execute, etc.)
- Overall, nice logical view (protection & sharing)
- Problem: still have [external] fragmentation.

Paging



1. Physical memory: f frames each 2^s bytes.
2. Virtual memory: p pages each 2^s bytes.
3. *Page table* maps $\{0, \ldots, p-1\} \rightarrow \{0, \ldots, f-1\}$
4. Allocation problem has gone away!

Typically have $p \gg f \Rightarrow$ add *valid* bit to say if a given page is represented in physical memory.

- Problem: now have *internal* fragmentation.
- Problem: protection/sharing now per page.

Segmentation versus Paging

	logical view	allocation
Segmentation	✓	✗
Paging	✗	✓

⇒ try combined scheme.

- E.g. *paged segments* (Multics, OS/2)
  - divide each segment s_i into $k = \lceil l_i / 2^n \rceil$ pages, where l_i is the limit (length) of the segment.
  - have page table per segment.

  ✗ high hardware cost / complexity.

  ✗ not very portable.

- E.g. *software segments* (most modern OSs)
  - consider pages $[m, \ldots, m+l]$ to be a segment.
  - OS must ensure protection / sharing kept consistent over region.

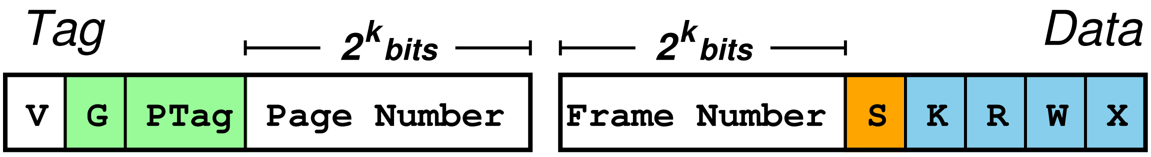
  ✗ loss in granularity.

  ✓ relatively simple / portable.

Translation Lookaside Buffers

Typically #pages large ⇒ page table lives in memory. Add *TLB*, a fully associative cache for mapping info:

- Check each memory reference in TLB first.
- If miss ⇒ need to load info from page table:
  - may be done in h/w or s/w (by OS).
  - if full, replace entry (usually h/w)
- Include protection info ⇒ can perform access check in parallel with translation.
- Context switch requires [expensive] flush:
  - can add process tags to improve performance.
  - "global" bit useful for wide sharing.
- Use *superpages* for large regions.
- So TLB contains n entries something like:

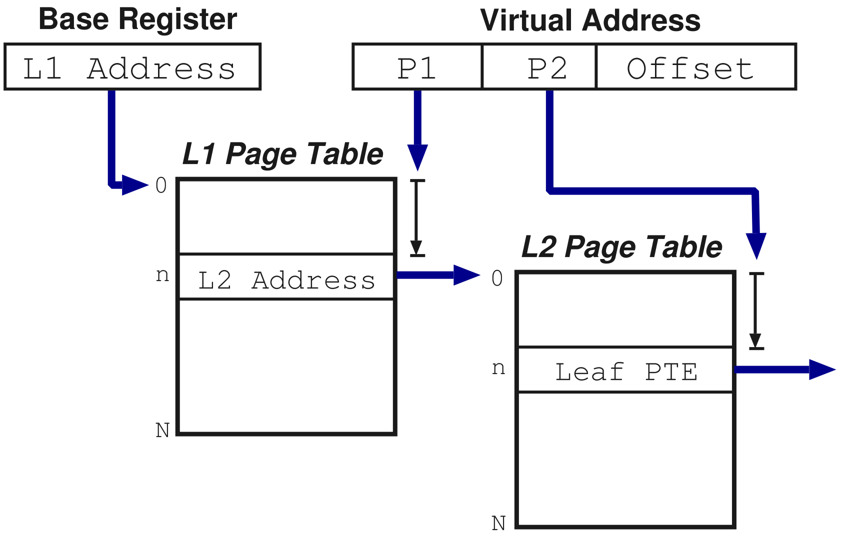


Most parts also present in *page table entries* (PTEs).

Page Table Design Issues

- A page table is a data-structure to translate pages $p \in P$ to frames $f \in \{F \cup \perp\}$
- Various concerns need to be addressed:
  - *time-efficiency*: we need to do a page-table lookup every time we get a TLB miss $\Rightarrow$ we wish to minimize the time taken for each one.
  - *space-efficiency*: since we typically have a page table for every process, we wish to minimize the memory required for each one.
  - *superpage support*: ideally should support any size of superpage the TLB understands.
  - *sharing*: given that many processes will have similar address spaces (`fork`, shared libraries), would like easy sharing of address regions.
  - *sparsity*: particularly for 64-bit address spaces, $|P| \gg |F|$ – best if we can have page table size proportional to $|F|$ rather than $|P|$.
  - *cache friendly*: PTEs for close together $p1, p2$ should themselves be close together in memory.
  - *simplicity*: all other things being equal, the simpler the solution the better.

Multi-Level Page Tables



- Modern systems have 2^{32} or 2^{64} byte VAS $\Rightarrow$ have between 2^{22} and 2^{42} pages (and hence PTEs).
- Solution: use N-ary tree (N large, 256–4096)
- Keep PTBR per process and context switch.
- Advantages: easy to implement; cache friendly.
- Disadvantages:
  - Potentially poor space overhead.
  - Inflexibility: superpages, residency.
  - Require $d \geq 2$ memory references.

MPT Pseudo Code

Assuming 32-bit virtual addresses, 4K pages, 2-level MPT (so $|P1| = |P2| =$ 10bits, and $|\text{offset}| =$ 12bits)

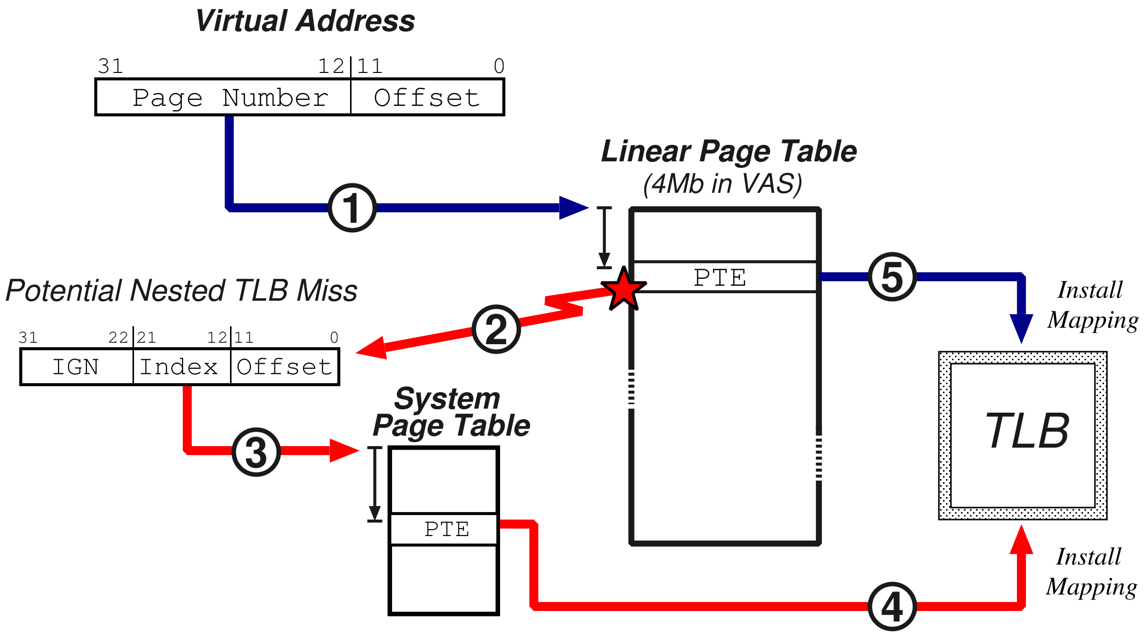
```
.entry TLBMiss:
  mov r0 <- Ptbr          // load page table base
  mov r1 <- FaultingVA    // get the faulting VA
  shr r1 <- r1, #12       // clear offset bits

  shr r2 <- r1, #10       // get L1 index
  shl r2 <- r2, #2        // scale to sizeof(PDE)
  pld r0 <- [r0, r2]      // load[phys] PDE

  shr r0 <- r0, #12       // clear prot bits
  shl r0 <- r0, #12       // get L2 address
  shl r1 <- #22, r1       // zap L1 index
  shr r1 <- #20, r1       // scale to sizeof(PTE)
  pld r0 <- [r0, r1]      // load[phys] PTE

  mov Tlb <- r0           // install TLB entry
  iret                    // return
```

Linear Page Tables



- Modification of MPTs:
  - typically implemented in software
  - pages of LPT translated on demand
  - i.e. stages ②, ③ and ④ not always needed.
- Advantages:
  - can require just 1 memory reference.
  - (initial) miss handler simple.
- But doesn't fix sparsity / superpages.
- *Guarded page tables* ($\approx$ tries) claim to fix these.

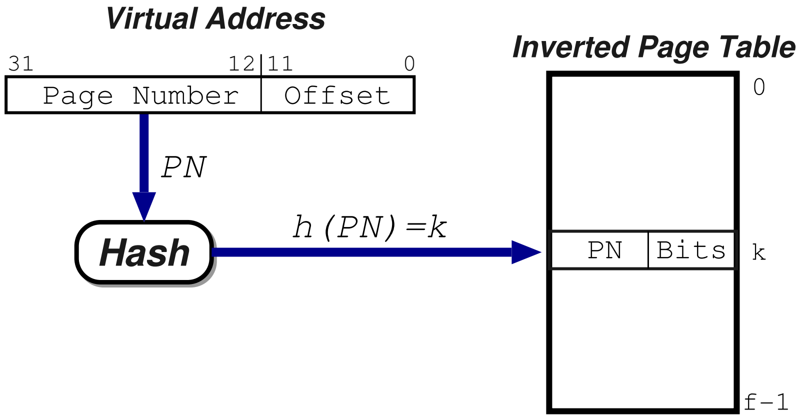
LPT Pseudo Code

Assuming 32-bit virtual addresses, 4K pages (so $|\text{pagenumber}| = 20\text{bits}$, and $|\text{offset}| = 12\text{bits}$)

```
.entry TLBMiss:
  mov r0 <- Ptbr          // load page table base
  mov r1 <- FaultingVA    // get the faulting VA
  shr r1 <- r1, #12       // clear offset bits
  shl r1 <- r1,  #2       // scale to sizeof(PTE)
  vld r0 <- [r0, r1]      // virtual load PTE:
                          // - this can fault!
  mov Tlb <- r0           // install TLB entry
  iret                    // return

.entry DoubleTLBMiss:
  add r2 <- r0, r1        // copy dble fault VA
  mov r3 <- SysPtbr       // get system PTBR
  shr r2 <- r2, #12       // clear offset bits
  shl r2 <- r2, #22       // zap "L1" index
  shr r2 <- r2, #20       // scale to sizeof(PTE)
  pld r3 <- [r3, r2]      // phys load LPT PTE
  mov Tlb <- r3           // install TLB entry
  iret                    // retry vld above
```

Inverted Page Tables



- Recall $f \ll p \rightarrow$ keep entry per *frame*.
- Then table size bounded by physical memory!
- IPT: frame number is `h(pn)`

✔ only one memory reference to translate.

✔ no problems with sparsity.

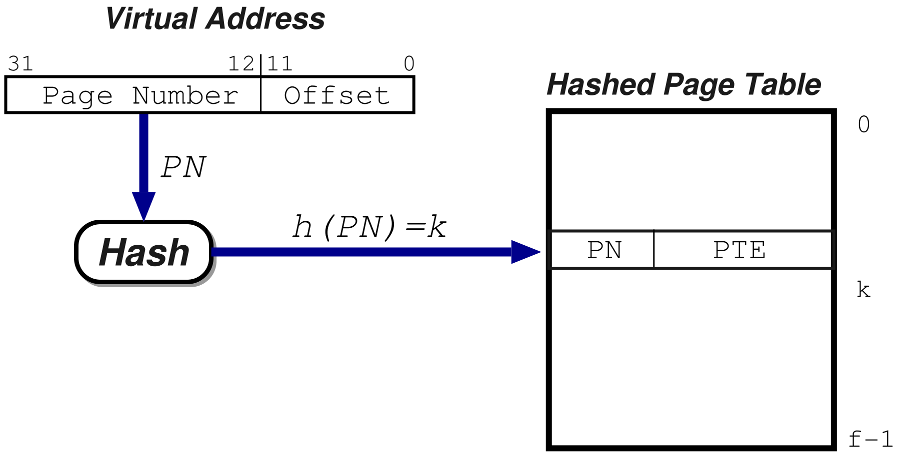
✔ can easily augment with process tag.

✘ no option on which frame to allocate

✘ dealing with collisions.

✘ cache unfriendly.

Hashed Page Tables



- HPT: simply extend IPT into proper hash table.
- i.e. make frame number explicit.
  - ✓ can map to any frame.
  - ✓ can choose table size.
  - ✗ table now bigger.
  - ✗ sharing still hard.
  - ✗ still cache unfriendly, no superpages.
- Can solve these last with *clustered page tables*.

Virtual Memory

- Virtual addressing allows us to introduce the idea of *virtual memory*:
  - already have valid or invalid page translations; introduce new "non-resident" designation
  - such pages live on a non-volatile *backing store*
  - processes access non-resident memory just as if it were 'the real thing'.
- Virtual memory (VM) has a number of benefits:
  - *portability*: programs work regardless of how much actual memory present
  - *convenience*: programmer can use e.g. large sparse data structures with impunity
  - *efficiency*: no need to waste (real) memory on code or data which isn't used.
- VM typically implemented via *demand paging*:
  - programs (executables) reside on disk
  - to execute a process we load pages in *on demand*; i.e. as and when they are referenced.
- Also get *demand segmentation*, but rare.

Demand Paging Details

When loading a new process for execution:

- create its address space (e.g. page tables, etc)
- mark PTEs as either "invalid or "non-resident"
- add PCB to scheduler.

Then whenever we receive a *page fault*:

1. check PTE to determine if "invalid" or not
2. if an invalid reference ⇒ kill process;
3. otherwise 'page in' the desired page:
    - find a free frame in memory
    - initiate disk I/O to read in the desired page
    - when I/O is finished modify the PTE for this page to show that it is now valid
    - restart the process at the faulting instruction

Scheme described above is *pure* demand paging:

- never brings in a page until required ⇒ get lots of page faults and I/O when process begins.
- hence many real systems explicitly load some core parts of the process first

Page Replacement

- When paging in from disk, we need a free frame of physical memory to hold the data we're reading in.
- In reality, size of physical memory is limited ⇒
  - need to discard unused pages if total demand for pages exceeds physical memory size
  - (alternatively could swap out a whole process to free some frames)
- Modified algorithm: on a page fault we
  1. locate the desired replacement page on disk
  2. to select a free frame for the incoming page:
     (a) if there is a free frame use it
     (b) otherwise select a *victim page* to free,
     (c) write the victim page back to disk, and
     (d) mark it as invalid in its process page tables
  3. read desired page into freed frame
  4. restart the faulting process
- Can reduce overhead by adding a 'dirty' bit to PTEs (can potentially omit step 2c above)
- Question: how do we choose our victim page?

Page Replacement Algorithms

- First-In First-Out (FIFO)
  - keep a queue of pages, discard from head
  - performance difficult to predict: no idea whether page replaced will be used again or not
  - discard is independent of page use frequency
  - in general: pretty bad, although very simple.
- Optimal Algorithm (OPT)
  - replace the page which will not be used again for longest period of time
  - can only be done with an oracle, or in hindsight
  - serves as a good comparison for other algorithms
- Least Recently Used (LRU)
  - LRU replaces the page which has not been used for the longest amount of time
  - (i.e. LRU is OPT with -ve time)
  - assumes past is a good predictor of the future
  - Q: how do we determine the LRU ordering?

Implementing LRU

- Could try using *counters*
  - give each page table entry a time-of-use field and give CPU a logical clock (counter)
  - whenever a page is referenced, its PTE is updated to clock value
  - replace page with smallest time value
  - problem: requires a search to find min value
  - problem: adds a write to memory (PTE) on every memory reference
  - problem: clock overflow
- Or a *page stack*:
  - maintain a stack of pages (doubly linked list) with most-recently used (MRU) page on top
  - discard from bottom of stack
  - requires changing 6 pointers per [new] reference
  - very slow without extensive hardware support
- Neither scheme seems practical on a standard processor ⇒ need another way.

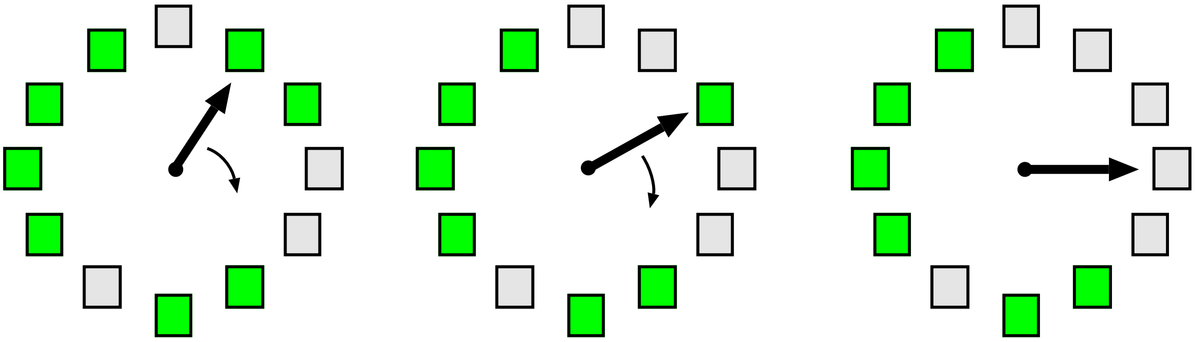
Approximating LRU (1)

- Many systems have a *reference bit* in the PTE which is set by h/w whenever the page is touched
- This allows *not recently used* (NRU) replacement:
  - periodically (e.g. 20ms) clear all reference bits
  - when choosing a victim to replace, prefer pages with clear reference bits
  - if also have a *modified bit* (or *dirty bit*) in the PTE, can extend MRU to use that too:

Ref?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

- Or can extend by maintaining more history, e.g.
  - for each page, the operating system maintains an 8-bit value, initialized to zero
  - periodically (e.g. 20ms) shift reference bit onto high order bit of the byte, and clear reference bit
  - select lowest value page (or one of) to replace

Approximating LRU (2)

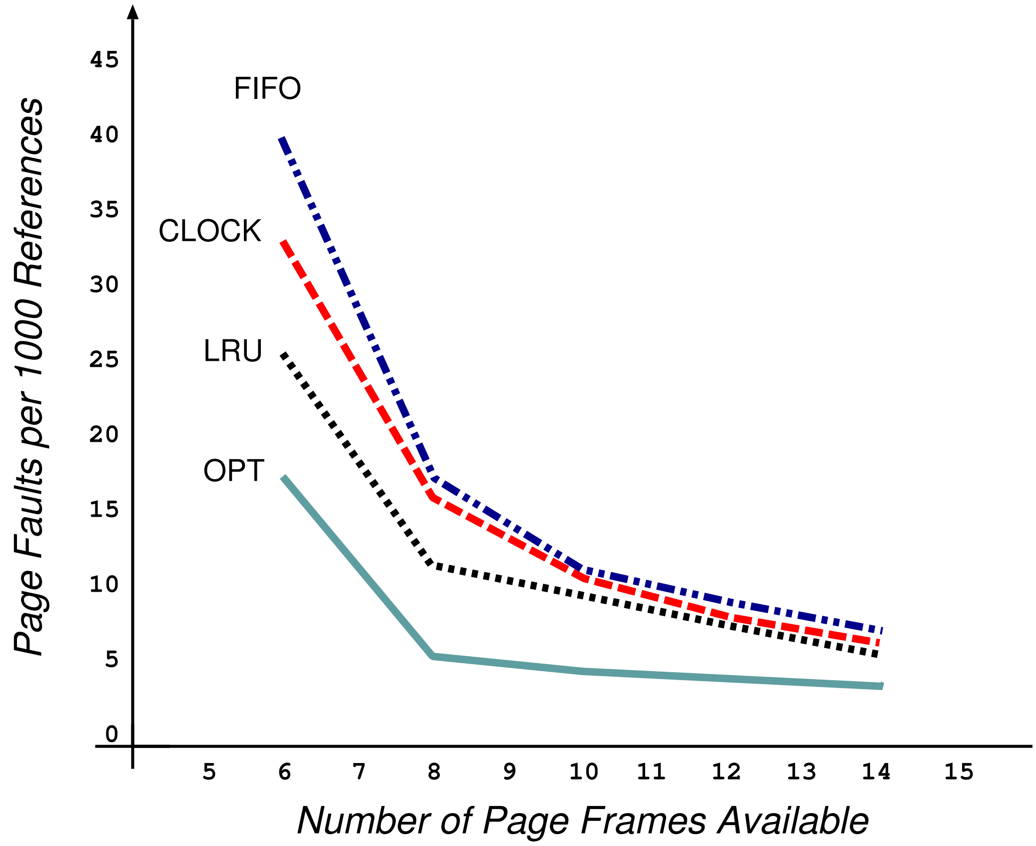


- Popular NRU scheme: *second-chance FIFO*
  - store pages in queue as per FIFO
  - before discarding head, check its reference bit
  - if reference bit is 0, discard, otherwise:
    - reset reference bit, and
    - add page to tail of queue
    - i.e. give it "a second chance"
- Often implemented with a circular queue and a current pointer; in this case usually called *clock*.
- If no h/w provided reference bit can emulate:
  - to clear "reference bit", mark page no access
  - if referenced ⇒ trap, update PTE, and resume
  - to check if referenced, check permissions
  - can use similar scheme to emulate modified bit

Other Replacement Schemes

- Counting Algorithms: keep a count of the number of references to each page
  - LFU: replace page with smallest count
  - MFU: replace highest count because low count $\Rightarrow$ most recently brought in.
- Page Buffering Algorithms:
  - keep a min. number of victims in a free pool
  - new page read in before writing out victim.
- (Pseudo) MRU:
  - consider access of e.g. large array.
  - page to replace is one application has *just finished with*, i.e. most recently used.
  - e.g. track page faults and look for sequences.
  - discard the k^{th} in victim sequence.
- Application-specific:
  - stop trying to second guess what's going on.
  - provide hook for app. to suggest replacement.
  - must be careful with denial of service. . .

Performance Comparison



Graph plots page-fault rate against number of physical frames for a pseudo-local reference string.

- want to minimise area under curve
- FIFO can exhibit Belady's anomaly (although it doesn't in this case)
- getting frame allocation right has major impact. . .

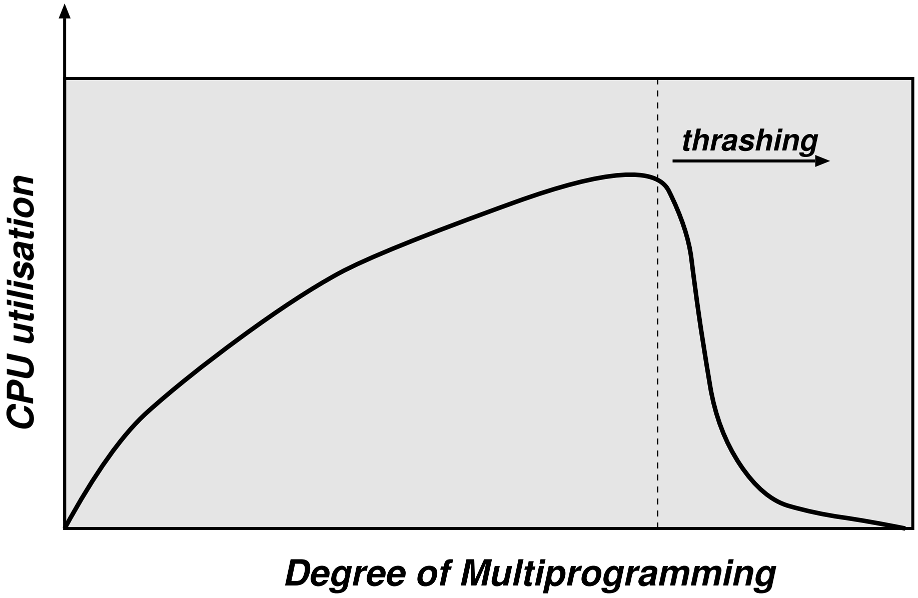
Frame Allocation

- A certain fraction of physical memory is reserved per-process and for core OS code and data.
- Need an *allocation policy* to determine how to distribute the remaining frames.
- Objectives:
  - Fairness (or proportional fairness)?
    - ∗ e.g. divide m frames between n processes as m/n, with remainder in the free pool
    - ∗ e.g. divide frames in proportion to size of process (i.e. number of pages used)
  - Minimize system-wide page-fault rate? (e.g. allocate all memory to few processes)
  - Maximize level of multiprogramming? (e.g. allocate min memory to many processes)
- Most page replacement schemes are *global*: all pages considered for replacement.

⇒ allocation policy implicitly enforced during page-in:

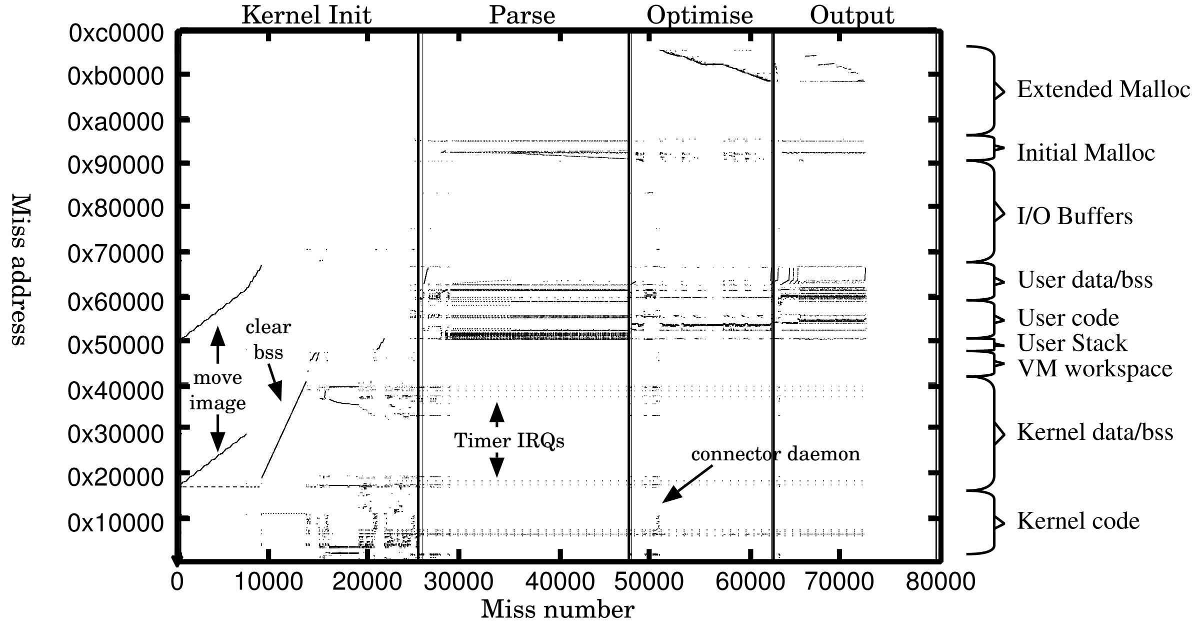
- allocation succeeds iff policy agrees
- 'free frames' often in use ⇒ steal them!

The Risk of Thrashing



- As more processes enter the system, the frames-per-process value can get very small.
- At some point we hit a wall:
  - a process needs more frames, so steals them
  - but the other processes need those pages, so they fault to bring them back in
  - number of runnable processes plunges
- To avoid thrashing we must give processes as many frames as they "need"
- If we can't, we need to reduce the MPL (a better page-replacement algorithm will not help)

Locality of Reference



Locality of reference: in a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space.

- procedure being executed
- . . . sub-procedures
- . . . data access
- . . . stack variables

Note: have locality in both space and time.

Avoiding Thrashing

We can use the locality of reference principle to help determine how many frames a process needs:

- define the *Working Set* (Denning, 1967)
  - set of pages that a process needs in store at "the same time" to make any progress
  - varies between processes and during execution
  - assume process moves through $phases$
  - in each phase, get (spatial) locality of reference
  - from time to time get $phase\ shift$
- Then OS can try to prevent thrashing by maintaining sufficient pages for current phase:
  - sample page reference bits every e.g. 10ms
  - if a page is "in use", say it's in the working set
  - sum working set sizes to get total demand D
  - if $D > m$ we are in danger of thrashing $\Rightarrow$ suspend a process
- Alternatively use page fault frequency (PFF):
  - monitor per-process page fault rate
  - if too high, allocate more frames to process

Other Performance Issues

Various other factors influence VM performance, e.g.

- Program structure: consider for example

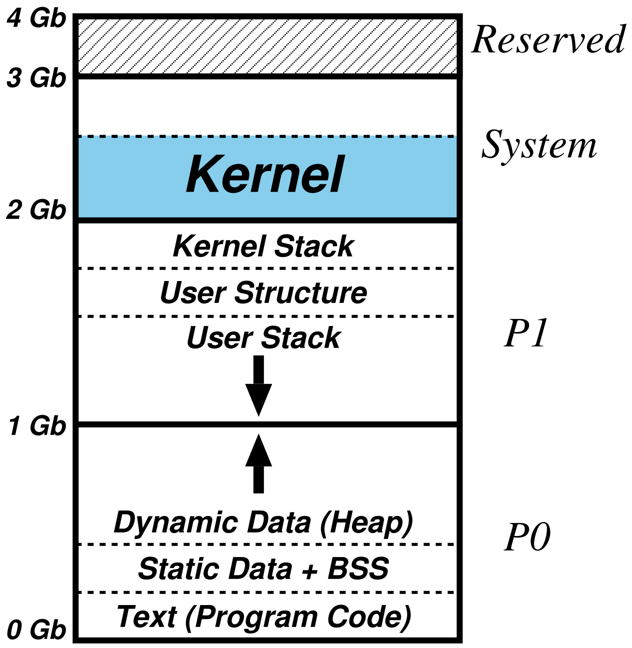
```
for(j=0; j<1024; j++)
    for(i=0; i<1024; i++)
        array[i][j] = 0;
```

  - a killer on a system with 1024 word pages; either have all pages resident, or get 2^{20} page faults.
  - if reverse order of iteration (i,j) then works fine
- Language choice:
  - ML, lisp: use lots of pointers, tend to randomise memory access $\Rightarrow$ kills spatial locality
  - Fortran, C, Java: relatively few pointer refs
- Pre-paging:
  - avoid problem with pure demand paging
  - can use WS technique to work out what to load
- Real-time systems:
  - no paging in hard RT (must lock all pages)
  - for SRT, trade-offs may be available

Case Study 1: Unix

- Swapping allowed from very early on.
- Kernel Per-process info. split into two kinds:
  - `proc` and `text` structures always resident.
  - page tables, `user` structure and kernel stack could be swapped out.
- Swapping performed by special process: the *swapper* (usually process 0).
  - periodically awaken and inspect processes on disk.
  - choose one waiting longest time and prepare to swap in.
  - victim chosen by looking at scheduler queues: try to find process blocked on I/O.
  - other metrics: priority, overall time resident, time since last swap in (for stability).
- From 3BSD / SVR2 onwards, implemented demand paging.
- Today swapping only used when dire shortage of physical memory.

Unix: Address Space



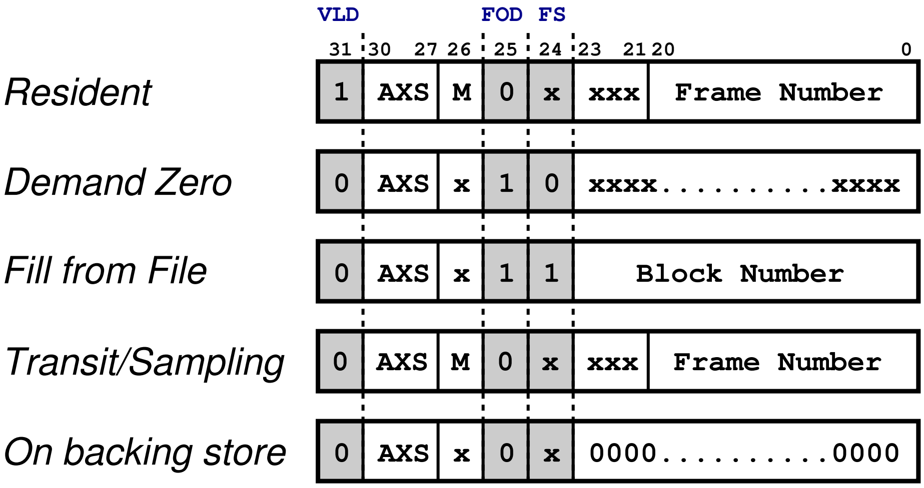
4.3 BSD UNIX address space borrows from VAX:

- 0Gb–1Gb: segment $P0$ (text/data, grow upward)
- 1Gb–2Gb: segment $P1$ (stack, grows downward)
- 2Gb–3Gb: *system* segment (for kernel).

Address translation done in hardware LPT:

- System page table always resident.
- $P0$, $P1$ page tables in system segment.
- Segments have page-aligned length.

Unix: Page Table Entries



	VLD 31	30 27	26	FOD 25	FS 24	23 21	20 0
Resident	1	AXS	M	0	x	xxx	Frame Number
Demand Zero	0	AXS	x	1	0	xxxx..........xxxx	
Fill from File	0	AXS	x	1	1	Block Number	
Transit/Sampling	0	AXS	M	0	x	xxx	Frame Number
On backing store	0	AXS	x	0	x	0000..........0000	

- PTEs for valid pages determined by h/w.
- If valid bit not set ⇒ use up to OS.
- BSD uses *FOD* bit, *FS* bit and the *block number*.
- First pair are "fill on demand":
  - DZ used for BSS, and growing stack.
  - FFF used for executables (text & data).
  - Simple pre-paging implemented via *klusters*.
- Sampling used to simulate reference bit.
- Backing store pages located via *swap map*(s).

Unix: Paging Dynamics

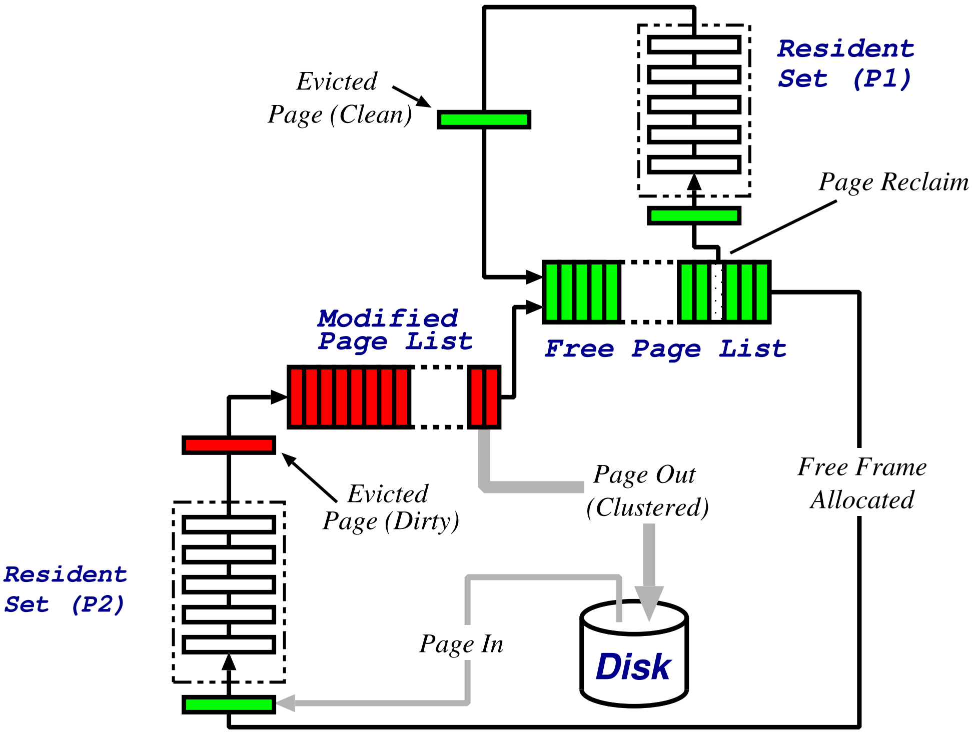
- Physical memory managed by *core map*:
  - array of structures, one per *cluster*.
  - records if free or in use and [potentially] the associated page number and disk block.
  - *free list* threads through core map.
- Page replacement carried out by the *page dæmon*.
  - every 250ms, OS checks if "enough' (viz. `lotsfree`) physical memory free.
  - if not ⇒ wake up page dæmon.
- Basic algorithm: global [two-handed] clock:
  - hands point to different entries in core map
  - first check if can replace front cluster; if not, clear its "reference bit" (viz. mark invalid).
  - then check if back cluster referenced (viz. marked valid); if so given second chance.
  - else flush to disk (if necessary), and put cluster onto end of free list.
  - move hands forward and repeat. . .
- System V Unix uses an almost identical scheme. . .

Case Study 2: VMS

- VMS released in 1978 to run on the VAX-11/780.
- Aimed to support a wide range of hardware, and a job mix of real-time, timeshared and batch tasks.
- This led to a design with:
  - A *local* page replacement scheme,
  - A *quota* scheme for physical memory, and
  - An aggressive *page clustering* policy.
- First two based around idea of *resident set*:
  - simply the set of pages which a given process currently has in memory.
  - each process also has a *resident-set limit*.
- Then during execution:
  - pages faulted in by pager on demand.
  - once hit limit, choose victim from resident set.

  ⇒ minimises impact on others.
- Also have swapper for extreme cases.

VMS: Paging Dynamics



- Basic algorithm: simple [local] FIFO.
- Suckful ⇒ augment with software "victim cache":
  - Victim pages placed on tail of FPL/MPL.
  - On fault, search lists before do I/O.
- Lists also allow aggressive *page clustering*:
  - if $|\text{MPL}| \geq$ `hi`, write $(|\text{MPL}| -$ `lo`$)$ pages.
  - Get ∼100 pages per write on average.

VMS: Other Issues

- Modified page replacement:
  - introduce *callback* for privileged processes.
  - prefer to retain pages with TLB entries.
- Automatic resident set limit adjustment:
  - system counts #page faults per process.
  - at quantum end, check if rate > `PFRATH`.
  - if so and if "enough" memory ⇒ increase RSL.
  - *swapper trimming* used to reduce RSLs again.
  - NB: real-time processes are exempt.
- Other system services:
  - `$SETSWM`: disable process swapping.
  - `$LCKPAG`: lock pages into memory.
  - `$LKWSET`: lock pages into resident set.
- VMS still alive:
  - recent versions updated to support 64-bit address space
  - son-of-VMS aka Win2K/XP also going strong.

Other VM Techniques

Once have MMU, can (ab)use for other reasons

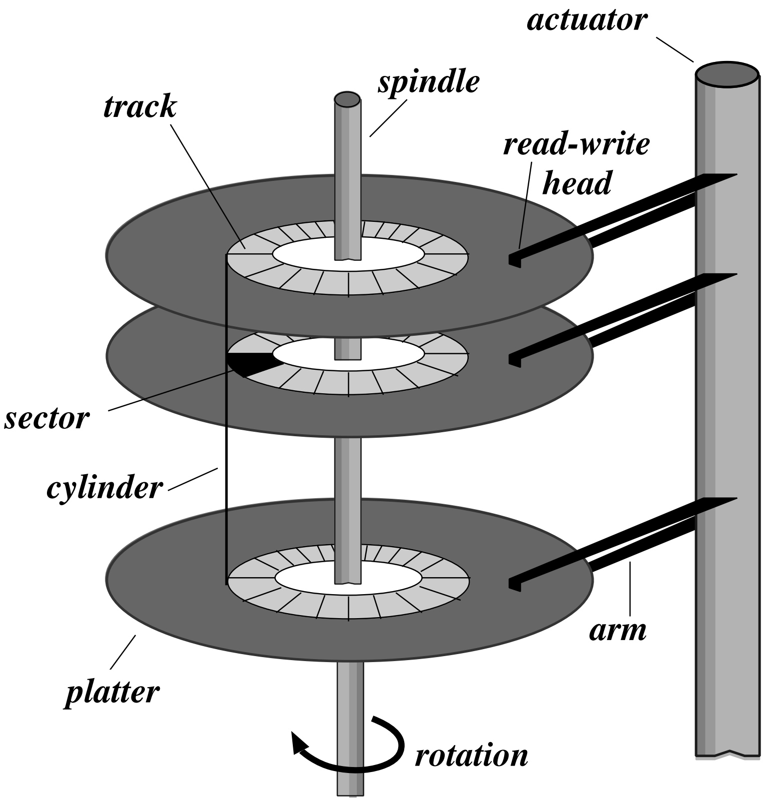
- Assume OS provides:
  - system calls to change memory protections.
  - some way to "catch" memory exceptions.
- This enables a large number of applications.
- e.g. concurrent garbage collection:
  - mark unscanned areas of heap as no-access.
  - if mutator thread accesses these, trap.
  - on trap, collector scans page(s), copying and forwarding as necessary.
  - finally, resume mutator thread.
- e.g. incremental checkpointing:
  - at time t atomically mark address space read-only.
  - on each trap, copy page, mark r/w and resume.
  - ✓ no significant interruption.
  - ✓ more space efficient

Single Address Space Operating Systems

- Emerging large (64-bit) address spaces mean that having a SVAS is plausible once more.
- Separate concerns of “what we can see” and “what we are allowed to access”.
- Advantages: easy sharing (unified addressing).
- Problems:
  - address binding issues return.
  - cache/TLB setup for MVAS model.
- Distributed shared virtual memory:
  - turn a NOW into a SMP.
  - how seamless do you think this is?
- Persistent object stores:
  - support for pickling & compression?
  - garbage collection?
- Sensible use requires restraint. . .

Disk I/O

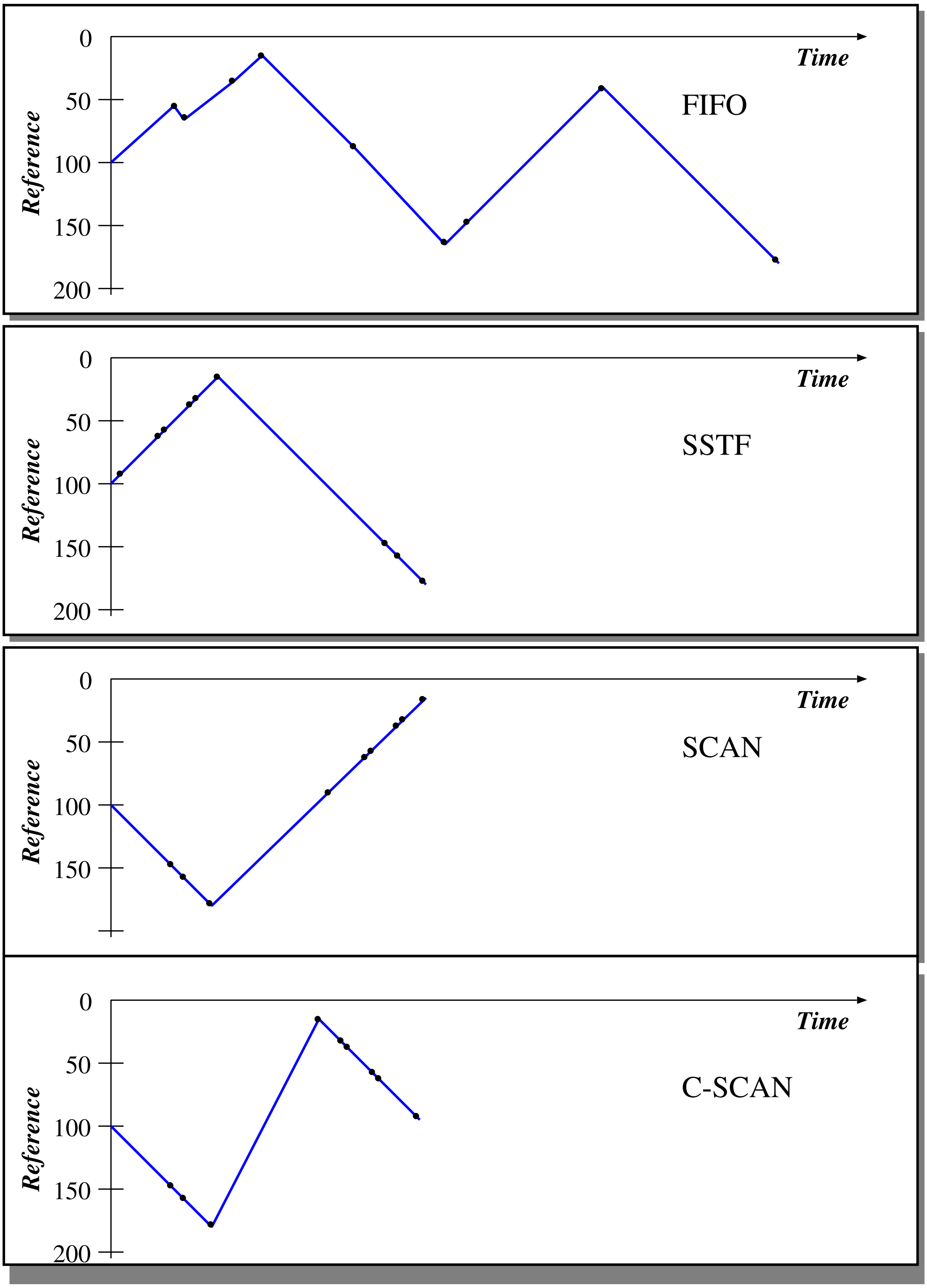
- Performance of disk I/O is crucial to swapping/paging and file system operation
- Key parameters:
  1. wait for controller and disk.
  2. seek to appropriate disk cylinder
  3. wait for desired block to come under the head
  4. transfer data to/from disk
- Performance depends on *how the disk is organised*



Disk Scheduling

- In a typical multiprogramming environment have multiple users queueing for access to disk
- Also have VM system requests to load/swap/page processes/pages
- We want to provide best performance to all users — specifically reducing seek time component
- Several policies for scheduling a set of disk requests onto the device, e.g.
  1. FIFO: perform requests in their arrival order
  2. SSTF: if the disk controller knows where the head is (hope so!) then it can schedule the request with the shortest seek from the current position
  3. SCAN ("elevator algorithm"): relieves problem that an unlucky request could receive bad performance due to queue position
  4. C-SCAN: scan in one direction only
  5. N-step-SCAN and FSCAN: ensure that the disk head always moves

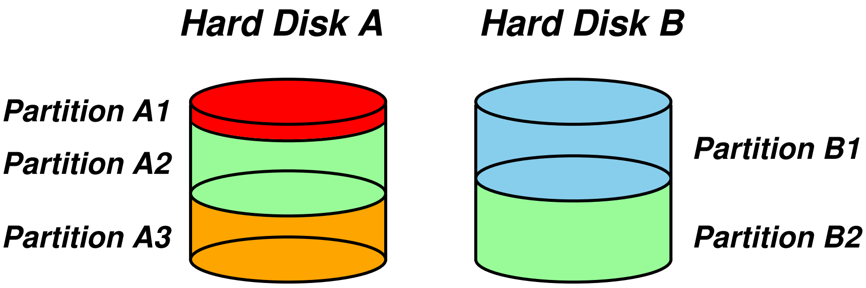
Reference String **=** `55,58,39,18,90,160,150,38,184`



Other Disk Scheduling Issues

- Priority: usually beyond disk controller's control.
  - system decides to prioritise, for example by ensuring that swaps get done before I/O.
  - alternatively interactive processes might get greater priority over batch processes.
  - or perhaps short requests given preference over larger ones (avoid "convoy effect")
- SRT disk scheduling (e.g. Cello, USD):
  - per client/process scheduling parameters.
  - two stage: admission, then queue.
  - problem: overall performance?
- 2-D Scheduling (e.g. SPTF).
  - try to reduce rotational latency.
  - typically require h/w support.
- Bad blocks remapping:
  - typically transparent ⇒ can potentially undo scheduling benefits.
  - some SCSI disks let OS into bad-block story

Logical Volumes



Modern OSs tend to abstract away from physical disk; instead use *logical volume* concept.

- Partitions first step.
- Augment with "soft partitions":
  - allow v. large number of partitions on one disk.
  - can customize, e.g. "real-time" volume.
  - aggregation: can make use of v. small partitions.
- Overall gives far more flexibility:
  - e.g. dynamic resizing of partitions
  - e.g. *striping* for performance.
- E.g. IRIX `xlm`, OSF/1 `lvm`, NT `FtDisk`.
- Other big opportunity is *reliability*. . .

RAID

RAID $=$ **R**edundant **A**rray of **I**nexpensive **D**isks:

- Uses various combinations of striping and $mirroring$ to increase performance.
- Can implement (some levels) in h/w or s/w
- Many levels exist:
  - RAID0: striping over n disks (so actually !**R**)
  - RAID1: simple mirroring, i.e. write n copies of data to n disks (where n is 2 ;-).
  - RAID2: hamming ECC (for disks with no built-in error detection)
  - RAID3: stripe data on multiple disks and keep parity on a dedicated disk. Done at byte level $\Rightarrow$ need spindle-synchronised disks.
  - RAID4: same as RAID3, but block level.
  - RAID5: same as RAID4, but no dedicated parity disk (round robin instead).
- AutoRAID trades off RAIDs 1 and 5.
- Even funkier stuff emerging. . .

Disk Caching

- Cache holds copy of some of disk sectors.
- Can reduce access time by applications if the required data follows the locality principle
- Design issues:
  - transfer data by DMA or by shared memory?
  - replacement strategy: LRU, LFU, etc.
  - reading ahead: e.g. track based.
  - write through or write back?
  - partitioning? (USD. . .)
- Typically O/S also provides a cache in s/w:
  - may be done per volume, or overall.
  - also get *unified* caches — treat VM and FS caching as part of the same thing.
- Software caching issues:
  - should we treat all filesystems the same?
  - do applications know better?

4.3 BSD Unix Buffer Cache

- Name? Well *buffers* data to/from disk, and *caches* recently used information.
- Modern Unix deals with *logical* blocks, i.e. FS block within a given partition / logical volume.
- “Typically” prevents 85% of implied disk transfers.
- Implemented as a hash table:
  - Hash on `(devno, blockno)` to see if present.
  - Linked list used for collisions.
- Also have **LRU** list (for replacement).
- Internal interface:
  - `bread()`: get data & lock buffer.
  - `brelse()`: unlock buffer (clean).
  - `bdwrite()`: mark buffer dirty (lazy write).
  - `bawrite()`: asynchronous write.
  - `bwrite()`: synchronous write.
- Uses `sync` every 30 secs for consistency.
- Limited prefetching (read-ahead).

NT/2K Cache Manager

- Cache Manager caches “virtual blocks”:
  - viz. keeps track of cache “lines” as offsets within a *file* rather than a volume.
  - disk layout & volume concept abstracted away.
  - ⇒ no translation required for cache hit.
  - ⇒ can get more intelligent prefetching
- Completely unified cache:
  - cache “lines” all in virtual address space.
  - decouples physical & virtual cache systems: e.g.
    - ∗ virtually cache in 256K blocks,
    - ∗ physically *cluster* up to 64K.
  - NT virtual memory manager responsible for actually doing the I/O.
  - allows lots of FS cache when VM system lightly loaded, little when system is thrashing.
- NT/2K also provides some user control:
  - if specify `temporary` attrib when creating file ⇒ will never be flushed to disk unless necessary.
  - if specify `write_through` attrib when opening a file ⇒ all writes will synchronously complete.

File systems

What is a filing system?

Normally consider it to comprise two parts:

1. Directory Service: this provides
    - naming mechanism
    - access control
    - existence control
    - concurrency control
2. Storage Service: this provides
    - integrity, data needs to survive:
        - hardware errors
        - OS errors
        - user errors
    - archiving
    - mechanism to implement directory service

What is a file?

- an ordered sequence of bytes (UNIX)
- an ordered sequence of records (ISO FTAM)

File Mapping Algorithms

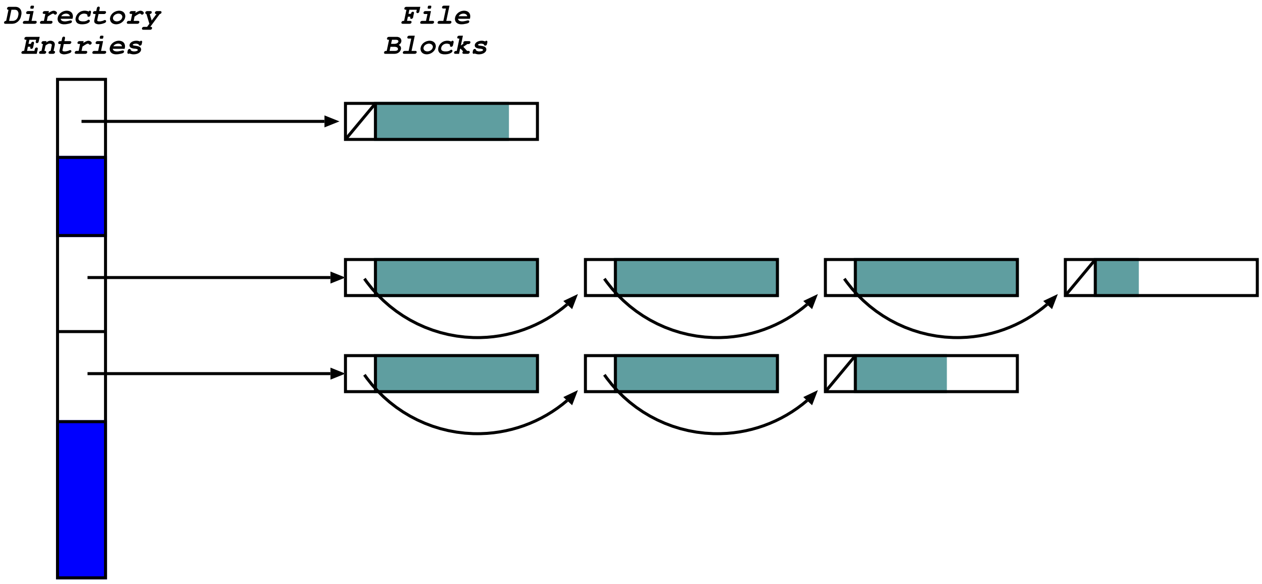
Need to be able to work out which disk blocks belong to which files ⇒ need a file-mapping algorithm, e.g.

1. chaining in the material
2. chaining in a map
3. table of pointers to blocks
4. extents

Aspects to consider:

- integrity checking after crash
- automatic recovery after crash
- efficiency for different access patterns
  - of data structure itself
  - of I/O operations to access it
- ability to extend files
- efficiency at high utilization of disk capacity

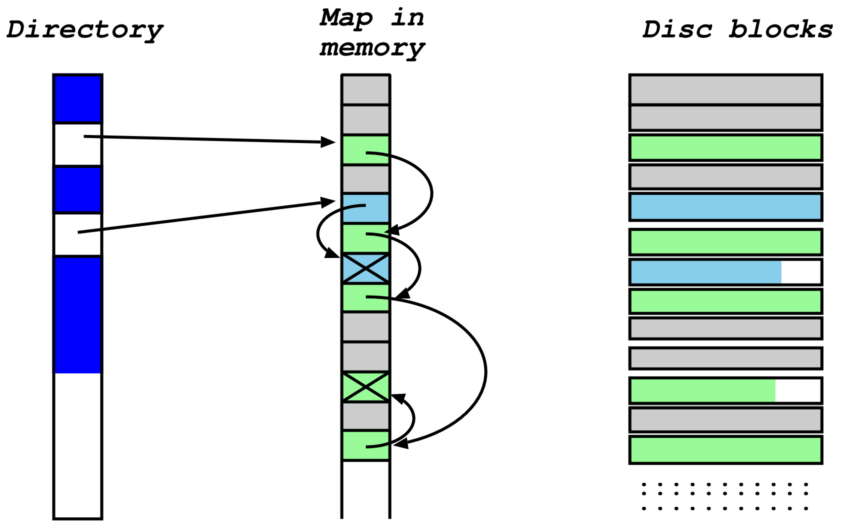
Chaining in the Media



Each disk block has pointer to next block in file.
Can also chain free blocks.

- OK for sequential access – poor for random access
- cost to find disk address of block n in a file:
  - best case: n disk reads
  - worst case: n disk reads
- Some problems:
  - not all of file block is file info
  - integrity check tedious. . .

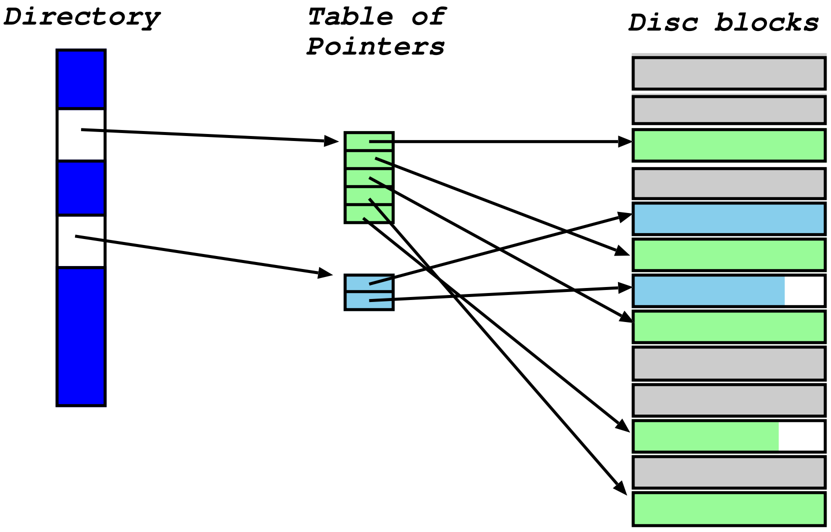
Chaining in a map



Maintain the chains of pointers in a map (in memory), mirroring disk structure.

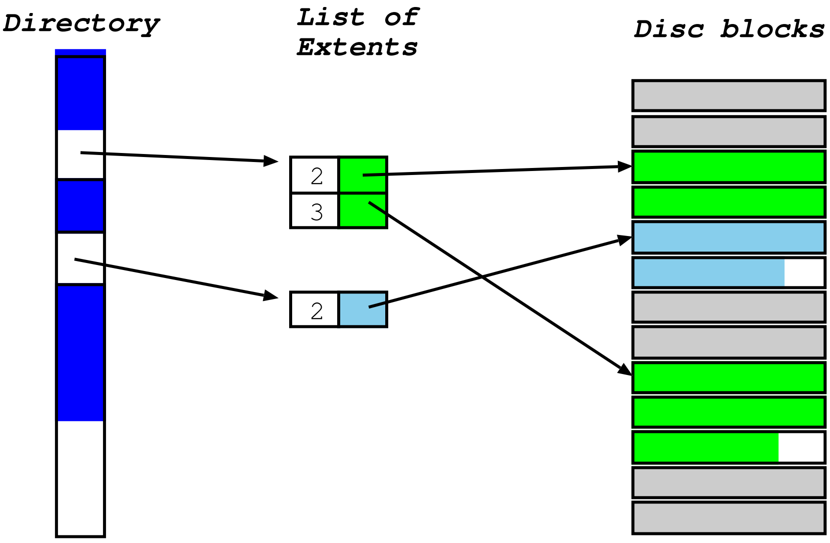
- disk blocks only contain file information
- integrity check easy: only need to check map
- handling of map is critical for
  - performance: must cache bulk of it.
  - reliability: must replicate on disk.
- cost to find disk address of block n in a file:
  - best case: n memory reads
  - worst case: n disk reads

Table of pointers



- access cost to find block n in a file
  - best case: 1 memory read
  - worst case: 1 disk read
- i.e. good for random access
- integrity check easy: only need to check tables
- free blocks managed independently (e.g. bitmap)
- table may get large ⇒ must chain tables, or build a tree of tables (e.g. UNIX inode)
- access cost for chain of tables? for hierarchy?

Extent lists



Using contiguous blocks can increase performance. . .

- list of disk addresses and lengths (extents)
- access cost: [perhaps] a disk read and then a searching problem, O(log(number of extents))
- can use bitmap to manage free space (e.g. QNX)
- system may have some maximum #extents
  - new error *can't extend file*
  - could copy file (i.e. compact into one extent)
  - or could chain tables or use a hierarchy as for table of pointers.

File Meta-Data (1)

What information is held about a file?

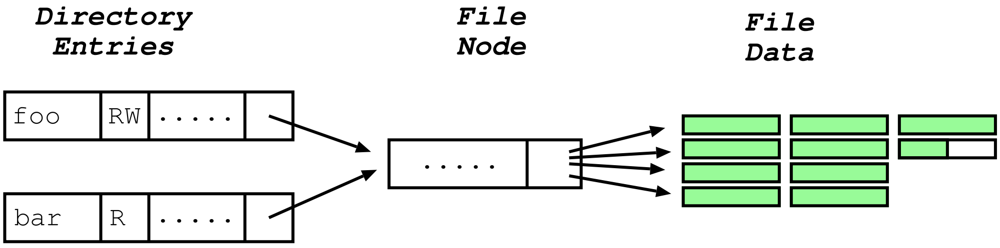
- times: creation, access and change?
- access control: who and how
- location of file data (see above)
- backup or archive information
- concurrency control

What is the name of a file?

- simple system: only name for file is human comprehensible text name
- perhaps want multiple text names for file
  - soft (symbolic) link: text name → text name
  - hard link: text name → file id
  - if we have hard links, must have reference counts on files

Together with the data structure describing the disk blocks, this information is known as the file *meta-data*.

File Meta-Data (2)



Where is file information kept?

- no hard links: keep it in the directory structure.
- if have hard links, keep file info separate from directory entries
  - file info in a block: OK if blocks small (e.g. TOPS10)
  - or in a table (UNIX i-node / v-node table)
- on OPEN, (after access check) copy info into memory for fast access
- on CLOSE, write updated file data and meta-data to disk

How do we handle *caching* meta-data?

Directory Name Space

- simplest — flat name space (e.g. Univac Exec 8)
- two level (e.g. CTSS, TOPS10)
- general hierarchy
  - a tree,
  - a directed (acyclic) graph (DAG)
- structure of name space often reflects data structures used to implement it
  - e.g. hierarchical name space ↔ hierarchical data structures
  - but, could have hierarchical name space and huge hash table!

General hierarchy:

- reflects structure of organisation, users' files etc.
- name is full path name, but can get rather long: e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  - offer relative naming
  - login directory
  - current working directory

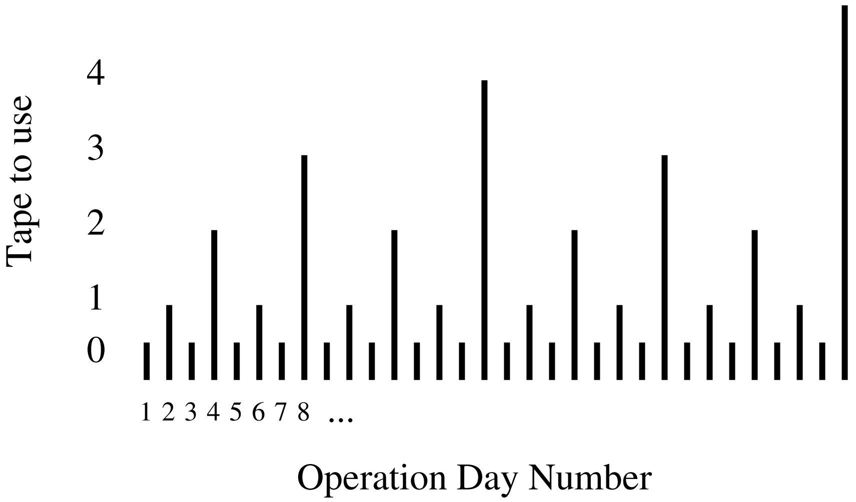
Directory Implementation

- Directories often don't get very large (especially if access control is at the directory level rather than on individual files)
  - ✓ often quick look up
  - ✗ directories may be small compared to underlying allocation unit
- But: assuming small dirs means lookup is naïve ⇒ trouble if get big dirs:
  - optimise for iteration.
  - keep entries sorted (e.g. use a B-Tree).
- Query based access:
  - split filespace into system and user.
  - system wants fast lookup but doesn't much care about friendly names (or may actively dislike)
  - user wishes 'easy' retrieval ⇒ build content index and make searching the default lookup.
  - Q: how do we keep index up-to-date?
  - Q: what about access control?

File System Backups

- Backup: keep (recent) copy of whole file system to allow recovery from:
  - CPU software crash
  - bad blocks on disk
  - disk head crash
  - fire, war, famine, pestilence
- What is a *recent* copy?
  - in real time systems recent means mirrored disks
  - daily usually sufficient for 'normal' machines
- Can use *incremental* technique, e.g.
  - full dump performed daily or weekly
    - copy whole file system to another disk or tape
    - ideally can do it while file system live. . .
  - incremental dump performed hourly or daily
    - only copy files and directories which have changed since the last time.
    - e.g. use the file 'last modifed' time
  - to recover: first restore full dump, then sucessively add in incrementals

Ruler Function



- Want to minimise #tapes needed, time to backup
- Want to maximise the time a file is held on backup
  - number days starting at 1
  - on day n use tape t such that 2^t is highest power of 2 which divides n
  - whenever we use tape t, dump all files modified since we last used that tape, or any tape with a higher number
- If file is created on day c and deleted on day d a dump version will be saved substantially after d
- the length of time it is saved depends on $d - c$ and the exact values of c, d

Crash Recovery

- Most failures affect only files being modified
- At start up after a crash run a *disk scavenger*:
  - try to recover data structures from memory (bring back core memory!)
  - get current state of data structures from disk
  - identify inconsistencies (may require operator intervention)
  - isolate suspect files and reload from backup
  - correct data structures and update disk
- Usually much faster and better (i.e. more recent) than recovery from backup.
- Can make scavenger's job simpler:
  - replicate vital data structures
  - spread replicas around the disk
- Even better: use *journal* file to assist recovery.
  - record all meta-data operations in an append-only [infinite] file.
  - ensure log records written before performing actual modification.

Immutable files

- Avoid concurrency problems: use write-once files.
  - multiple version numbers: foo!11, foo!12
  - invent new version number on close
- Problems:
  - disk space is not infinite
    - only keep last k versions (archive rest?)
    - have a explicit *keep* call
    - share disk blocks between different versions — complicated file system structures
  - what does name without version mean?
  - and the killer. . . directories aren't immutable!
- But:
  - only need concurrency control on version number
  - could be used (for files) on unusual media types
    - write once optical disks
    - remote servers (e.g. Cedar FS)
  - provides an audit trail
    - required by the spooks
    - often implemented on top of normal file system; e.g. UNIX RCS

Multi-level stores

Archiving (c.f. backup): keep frequently used files on fast media, migrate others to slower (cheaper) media.

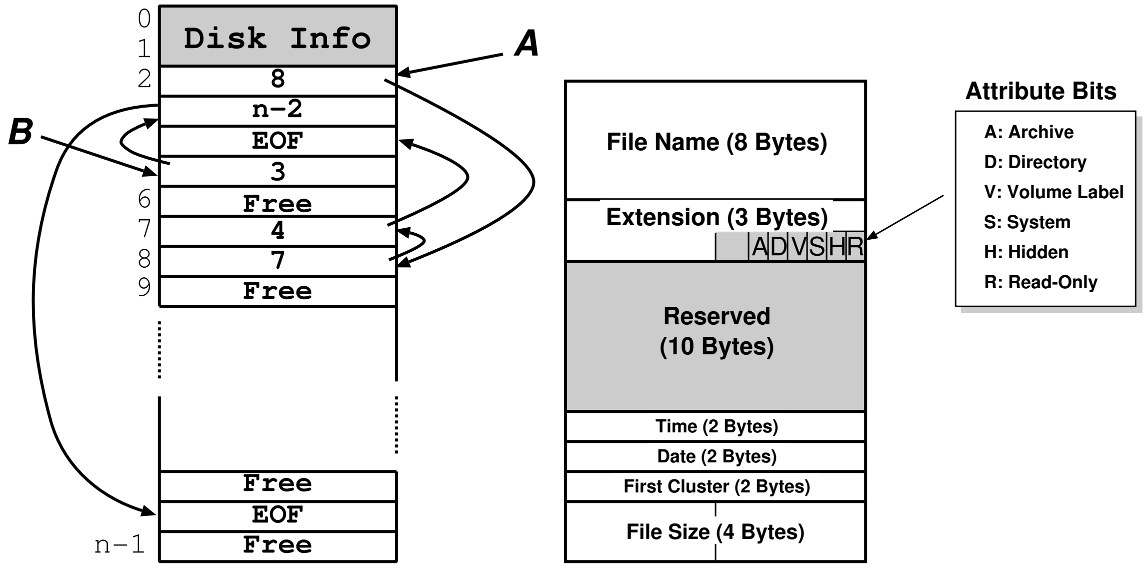
Can be done by:

- user — *encouraged* by accounting penalties
- system — migrate files by periodically looking at time of last use
- can provide transparent naming but not performance, e.g. HSM (Windows 2000)

Can integrate multi-level store with ideas from immutable files, e.g. Plan-9:

- file servers with fast disks
- write once optical juke box
- every night, mark all files immutable
- start migration of files which changed the previous day to optical disk
- access to archive explicit
  e.g. /archive/08Jan2005/users/smh/. . .

Case Study 1: FAT16/32



- A file is a linked list of *clusters*: a cluster is a set of 2^n contiguous disk blocks, $n \geq 0$.
- Each entry in the `FAT` contains either:
  - the index of another entry within the `FAT`, or
  - a special value `EOF` meaning "end of file", or
  - a special value `Free` meaning "free".
- Directory entries contain index into the `FAT`
- `FAT16` could only handle partitions up to $(2^{16} \times c)$ bytes ⇒ max 2Gb partition with 32K clusters.
- (and big cluster size is *bad*)

Extending FAT16 to FAT32

- Obvious extetension: instead of using 2 bytes per entry, `FAT32` uses 4 bytes per entry

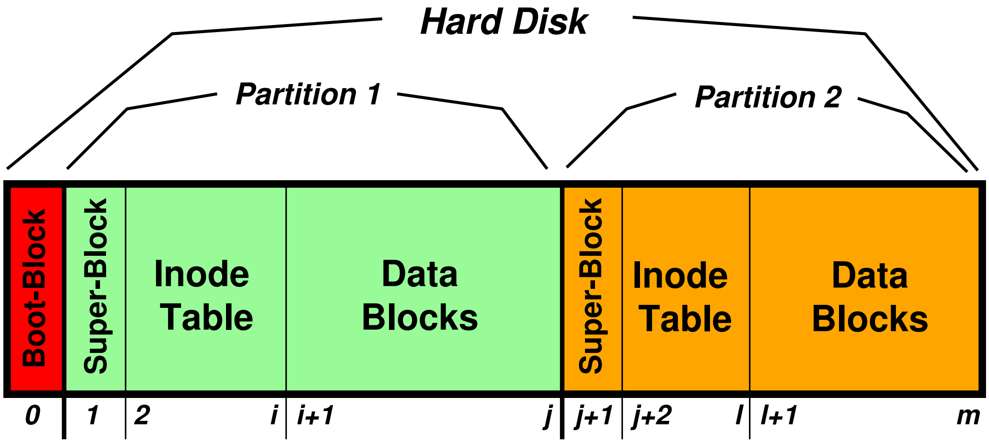
⇒ can support e.g. 8Gb partition with 4K clusters

- Further enhancements with `FAT32` include:
  - can locate the root directory anywhere on the partition (in `FAT16`, the root directory had to immediately follow the `FAT`(s)).
  - can use the backup copy of the `FAT` instead of the default (more fault tolerant)
  - improved support for demand paged executables (consider the 4K default cluster size . . .).
- `VFAT` on top of `FAT32` does long name support: unicode strings of up to 256 characters.
  - want to keep same directory entry structure for compatibility with e.g. DOS

  ⇒ use *multiple* directory entries to contain successive parts of name.
  - abuse `V` attribute to avoid listing these

Case Study 2: BSD FFS

The original Unix file system: simple, elegant, slow.



The *fast file-system* (FFS) was develped in the hope of overcoming the following shortcomings:

1. Poor data/metadata layout:
    - widely separating data and metadata ⇒ almost guaranteed long seeks
    - head crash near start of partition disastrous.
    - consecutive file blocks not close together.
2. Data blocks too small:
    - 512 byte allocation size good to reduce internal fragementation (median file size ~ 2K)
    - but poor performance for somewhat larger files.

FFS: Improving Performance

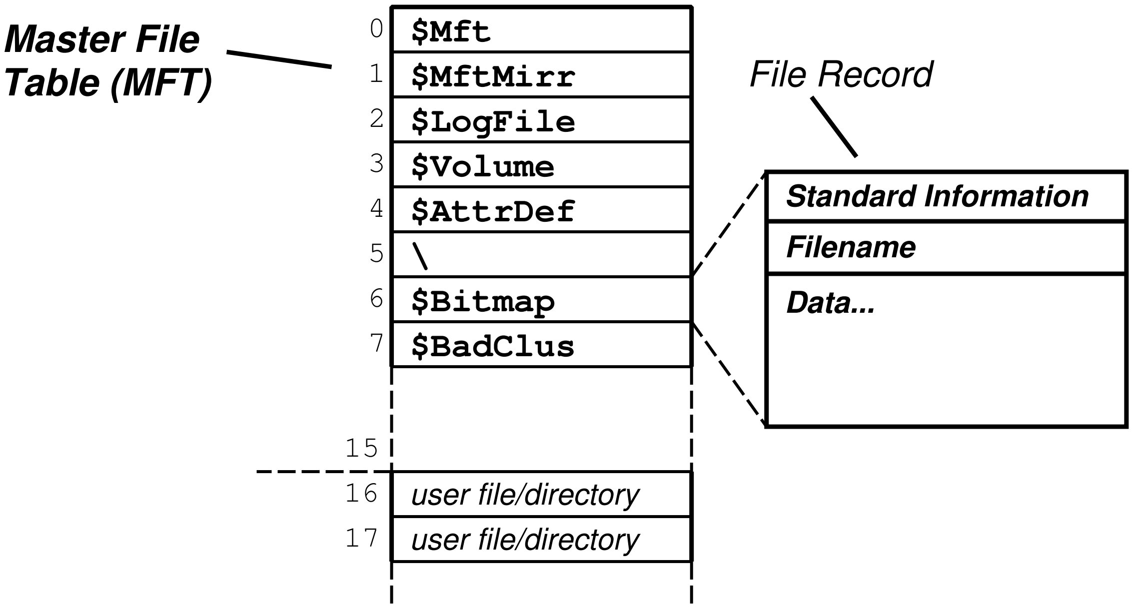
The FFS set out to address these issues:

- Block size problem:
  - use larger block size (e.g. 4096 or 8192 bytes)
  - but: last block in a file *may* be split into fragments of e.g. 512 bytes.
- Random allocation problem:
  - ditch free list in favour of bitmap ⇒ since easier to find contiguous free blocks (e.g. `01110000011101`)
  - divide disk into *cylinder groups* containing:
    - ∗ a superblock (replica),
    - ∗ a set of inodes,
    - ∗ a bitmap of free blocks, and
    - ∗ usage summary information.
  - (cylinder group $\simeq$ little Unix file system)
- Cylinder groups used to:
  - keep inodes near their data blocks
  - keep inodes of a directory together
  - increase fault tolerance

FFS: Locality and Allocation

- Locality key to achieving high performance
- To achieve locality:
  1. don't let disk fill up ⇒ can find space nearby
  2. spread unrelated things far apart.
- e.g. the BSD allocator tries to keep files in a directory in the same cylinder group, but spread directories out among the various cylinder groups
- similarly allocates runs of blocks within a cylinder group, but switches to a different one after 48K
- So does all this make any difference?
  - yes! about 10x–20x original FS performance
  - get up to 40% of disk bandwidth on large files
  - and *much* beter small file performance.
- Problems?
  - block-based scheme limits throughput ⇒ need decent clustering, or skip-sector allocation
  - crash recovery not particularly fast
  - rather tied to disk geometry. . .

Case Study 3: NTFS



- Fundamental structure of NTFS is a *volume*:
  - based on a logical disk partition
  - may occupy a portion of a disk, and entire disk, or span across several disks.
- An array of file records is stored in a special file called the Master File Table (MFT).
- The MFT is indexed by a *file reference* (a 64-bit unique identifier for a file)
- A file itself is a structured object consisting of set of attribute/value pairs of variable length. . .

NTFS: Recovery

- To aid recovery, all file system data structure updates are performed inside *transactions*:
  - before a data structure is altered, the transaction writes a log record that contains redo and undo information.
  - after the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
  - after a crash, the file system can be restored to a consistent state by processing the log records.
- Does not guarantee that all the user file data can be recovered after a crash — just that metadata files will reflect some prior consistent state.
- The log is stored in the third metadata file at the beginning of the volume (`$Logfile`) :
- Logging functionality not part of NTFS itself:
  - NT has a generic *log file service*

  ⇒ could in principle be used by e.g. database
- Overall makes for far quicker recovery after crash

NTFS: Other Features

- Security:
  - each file object has a *security descriptor attribute* stored in its MFT record.
  - this atrribute contains the access token of the owner of the file plus an access control list
- Fault Tolerance:
  - `FtDisk` driver allows multiple partitions be combined into a logical volume (RAID 0, 1, 5)
  - `FtDisk` can also handle *sector sparing* where the underlying SCSI disk supports it
  - NTFS supports software *cluster remapping*.
- Compression:
  - NTFS can divide a file's data into *compression units* (blocks of 16 contiguous clusters)
  - NTFS also has support for *sparse files*
    - ∗ clusters with all zeros not allocated or stored
    - ∗ instead, gaps are left in the sequences of VCNs kept in the file record
    - ∗ when reading a file, gaps cause NTFS to zero-fill that portion of the caller's buffer.

Case Study 4: LFS (Sprite)

LFS is a *log-structured file system* — a radically different file system design:

- Premise 1: CPUs getting faster faster than disks.
- Premise 2: memory cheap ⇒ large disk caches
- Premise 3: large cache ⇒ most disk reads "free".

⇒ performance bottleneck is writing & seeking.

Basic idea: solve write/seek problems by using a *log*:

- log is [logically] an append-only piece of storage comprising a set of *records*.
- all data & meta-data updates written to log.
- periodically flush entire log to disk in a single contiguous transfer:
  - high bandwidth transfer.
  - can make blocks of a file contiguous on disk.
- have two logs ⇒ one in use, one being written.

What are the problems here?

LFS: Implementation Issues

1. How do we find data in the log?
   - can keep basic UNIX structure (directories, inodes, indirect blocks, etc)
   - then just need to find inodes ⇒ use *inode map*
   - find inode maps by looking at a checkpoint
   - checkpoints live in fixed region on disk.
2. What do we do when the disk is full?
   - need asynchronous *scavenger* to run over old logs and free up some space.
   - two basic alternatives:
     1. compact live information to free up space.
     2. thread log through free space.
   - neither great ⇒ use *segmented log*:
     - divide disk into large fixed-size segments.
     - compact within a segment, thread between segments.
     - when writing use only clean segments
     - occasionally clean segments
     - choosing segments to clean is hard. . .

Subject of ongoing debate in the OS community. . .

Protection

Require protection against unauthorised:

- release of information
  - reading or leaking data
  - violating privacy legislation
  - using proprietary software
  - covert channels
- modification of information
  - changing access rights
  - can do sabotage without reading information
- denial of service
  - causing a crash
  - causing high load (e.g. processes or packets)
  - changing access rights

Also wish to protect against the effects of errors:

- isolate for debugging
- isolate for damage control

Protection mechanisms impose controls on access by *subjects* (e.g. users) on *objects* (e.g. files).

Protection and Sharing

If we have a single user machine with no network connection in a locked room then protection is easy.

But we want to:

- share facilities (for economic reasons)
- share and exchange data (application requirement)

Some mechanisms we have already come across:

- user and supervisor levels
  - usually one of each
  - could have several (e.g. MULTICS rings)
- memory management hardware
  - protection keys
  - relocation hardware
  - bounds checking
  - separate address spaces
- files
  - access control list
  - groups etc

Design of Protection System

- Some other protection mechanisms:
  - lock the computer room (prevent people from tampering with the hardware)
  - restrict access to system software
  - de-skill systems operating staff
  - keep designers away from final system!
  - use passwords (in general challenge/response)
  - use encryption
  - legislate
- ref: Saltzer + Schroeder Proc. IEEE Sept 75
  - design should be public
  - default should be no access
  - check for current authority
  - give each process minimum possible authority
  - mechanisms should be simple, uniform and built in to lowest layers
  - should be psychologically acceptable
  - cost of circumvention should be high
  - minimize shared access

Authentication of User to System (1)

Passwords currently widely used:

- want a long sequence of random characters issued by system, but user would write it down
- if allow user selection, they will use dictionary words, car registration, their name, etc.
- best bet probably is to encourage the use of an algorithm to remember password
- other top tips:
  - don't reflect on terminal, or overprint
  - add delay after failed attempt
  - use encryption if line suspect
- what about security of password file?
  - only accessible to login program (CAP, TITAN)
  - hold scrambled, e.g. UNIX
    - only need to write protect file
    - need irreversible function (without password)
    - maybe 'one-way' function
    - however, off line attack possible
      $\Rightarrow$ really should use *shadow passwords*

Authentication of User to System (2)

E.g. passwords in UNIX:

- simple for user to remember

  `arachnid`

- sensible user applies an algorithm

  `!r!chn#d`

- password is DES-encrypted 25 times using a 2-byte per-user ‘salt’ to produce a 11 byte string
- salt followed by these 11 bytes are then stored

  `IML.DVMcz6Sh2`

Really require unforgeable evidence of identity that system can check:

- enhanced password: challenge-response.
- id card inserted into slot
- fingerprint, voiceprint, face recognition
- smart cards

Authentication of System to User

User wants to avoid:

- talking to wrong computer
- right computer, but not the login program

Partial solution in old days for directly wired terminals:

- make login character same as terminal attention, or
- always do a terminal attention before trying login

But, today micros used as terminals ⇒

- local software may have been changed
- so carry your own copy of the terminal program
- but hardware / firmware in public machine may have been modified

Anyway, still have the problem of comms lines:

- wiretapping is easy
- workstation can often see all packets on network

⇒ must use encryption of some kind, and must trust encryption device (e.g. a smart card)

Mutual suspicion

- We need to encourage lots and lots of suspicion:
  - system of user
  - users of each other
  - user of system
- Called programs should be suspicious of caller (e.g. OS calls always need to check parameters)
- Caller should be suspicious of called program
- e.g. Trojan horse:
  - a ‘useful’ looking program — a game perhaps
  - when called by user (in many systems) inherits all of the user’s privileges
  - it can then copy files, modify files, change password, send mail, etc. . .
  - e.g. Multics editor trojan horse, copied files as well as edited.
- e.g. Virus:
  - often starts off as Trojan horse
  - self-replicating (e.g. ILOVEYOU)

Access matrix

Access matrix is a matrix of subjects against objects.

Subject (or principal) might be:

- users e.g. by uid
- executing process in a protection domain
- sets of users or processes

Objects are things like:

- files
- devices
- domains / processes
- message ports (in microkernels)

Matrix is large and sparse ⇒ don't want to store it all.

Two common representations:

1. by object: store list of subjects and rights with each object ⇒ *access control list*
2. by subject: store list of objects and rights with each subject ⇒ *capabilities*

Access Control Lists

Often used in storage systems:

- system naming scheme provides for ACL to be inserted in naming path, e.g. files
- if ACLs stored on disk, check is made in software $\Rightarrow$ must only use on low duty cycle
- for higher duty cycle must cache results of check
- e.g. Multics: open file = memory segment.
  On first reference to segment:
  1. interrupt (segment fault)
  2. check ACL
  3. set up segment descriptor in segment table
- most systems check ACL
  - when file opened for read or write
  - when code file is to be executed
- access control by program, e.g. Unix
  - exam prog, RWX by examiner, X by student
  - data file, A by exam program, RW by examiner
- allows arbitrary policies. . .

Capabilities

Capabilities associated with active subjects, so:

- store in address space of subject
- must make sure subject can't forge capabilities
- easily accessible to hardware
- can be used with high duty cycle
  e.g. as part of addressing hardware
  - Plessey PP250
  - CAP I, II, III
  - IBM system/38
  - Intel iAPX432
- have special machine instructions to modify (restrict) capabilities
- support passing of capabilities on procedure (program) call

Can also use *software* capabilities:

- checked by encryption
- nice for distributed systems

Password Capabilities

- Capabilities nice for distributed systems but:
  - messy for application, and
  - revocation is tricky.
- Could use timeouts (e.g. Amoeba).
- Alternatively: combine passwords and capabilities.
- Store ACL with object, but key it on capability (not implicit concept of “principal” from OS).
- Advantages:
  - revocation possible
  - multiple “roles” available.
- Disadvantages:
  - still messy (use ‘implicit’ cache?).

Covert channels

Information leakage by side-effects: lots of fun!

At the hardware level:

- wire tapping
- monitor signals in machine
- modification to hardware
- electromagnetic radiation of devices

By software:

- leak a bit stream as:

file exists	page fault	compute a while	1
no file	no page fault	sleep for a while	0

- system may provide statistics
  e.g. TENEX password cracker using system provided count of page faults

In general, guarding against covert channels is prohibitively expensive.

(only usually a consideration for military types)

Summary & Outlook

- An operating system must:
  1. securely multiplex resources.
  2. provide / allow abstractions.
- Major aspect of OS design is choosing trade-offs.
  - e.g. protection vs. performance vs. portability
  - e.g. prettiness vs. power.
- Future systems bring new challenges:
  - scalability (multi-processing/computing)
  - reliability (computing infrastructure)
  - ubiquity (heterogeneity/security)
- Lots of work remains. . .

Exams 2005:

- 2Qs, one each in Papers 3 & 4
- (not mandatory: P3 and P4 both x-of-y)
- Qs generally book-work plus ‘decider’

1999: Paper 3 Question 8

FIFO, LRU, and CLOCK are three page replacement algorithms.

a) Briefly describe the operation of each algorithm. **[6 marks]**

b) The CLOCK strategy assumes some hardware support. What could you do to allow the use of CLOCK if this hardware support were not present? **[2 marks]**

c) Assuming good temporal locality of reference, which of the above three algorithms would you choose to use within an operating system? Why would you not use the other schemes? **[2 marks]**

What is a *buffer cache*? Explain why one is used, and how it works. **[6 marks]**

Which buffer cache replacement strategy would you choose to use within an operating system? Justify your answer. **[2 marks]**

Give *two* reasons why the buffering requirements for network data are different from those for file systems. **[2 marks]**

1999: Paper 4 Question 7

The following are three ways which a file system may use to determine which disk blocks make up a given file.

a) chaining in a map
b) tables of pointers
c) extent lists

Briefly describe how each scheme works. [**3 marks each**]

Describe the benefits and drawbacks of using scheme (c). [**6 marks**]

You are part of a team designing a distributed filing system which replicates files for performance and fault-tolerance reasons. It is required that rights to a given file can be revoked within T milliseconds ($T \geq 0$). Describe how you would achieve this, commenting on how the value of T would influence your decision. [**5 marks**]

2000: Paper 3 Question 7

Why are the scheduling algorithms used in general purpose operating systems such as Unix and Windows NT not suitable for real-time systems? [**4 marks**]

Rate monotonic (RM) and earliest deadling first (EDF) are two popular scheduling algorithms for real-time systems. Describe these algorithms, illustrating your answer by showing how each of them would schedule the following task set.

Task	*Requires Exactly*	*Every*
$\mathcal{A}$	2ms	10ms
$\mathcal{B}$	1ms	4ms
$\mathcal{C}$	1ms	5ms

You may assume that context switches are instantaneous.

[**8 marks**]

Exhibit a task set which is schedulable under EDF but not under RM. You should demonstrate that this is the case, and explain why.
*H*int: consider the relationship between task periods.

[**8 marks**]

2000: Paper 4 Question 7

Why is it important for an operating system to schedule disk requests? [**4 marks**]

Briefly describe each of the SSTF, SCAN and C-SCAN disk scheduling algorithms. Which problem with SSTF does SCAN seek to overcome? Which problem with SCAN does C-SCAN seek to overcome? [**5 marks**]

Consider a Winchester-style hard disk with 100 cylinders, 4 double-sided platters and 25 sectors per track. The following is the (time-ordered) sequence of requests for disk sectors:

> { 3518, 1846, 8924, 6672, 1590, 4126, 107, 9750, 158, 6621, 446, 11 }.

The disk arm is currently at cylinder 10, moving towards 100. For each of SSTF, SCAN and C-SCAN, give the order in which the above requests would be serviced. [**3 marks**]

Which factors do the above disk arm scheduling algorithms ignore? How could these be taken into account? [**4 marks**]

Discuss ways in which an operating system can construct logical volumes which are (a) more reliable and (b) higher performance than the underlying hardware. [**4 marks**]

2001: Paper 3 Question 7

What are the key issues with scheduling for shared-memory multiprocessors? [**3 marks**]

Processor affinity, *take scheduling* and *gang scheduling* are three techniques used within multiprocessor operating systems.

a) Briefly describe the operation of each. [**6 marks**]

b) Which problem does the processor affinity technique seek to overcome? [**2 marks**]

c) What problem does the processor affinity technique suffer from, and how could this problem in turn be overcome? [**2 marks**]

d) In which circumstances is a gang scheduling approach most appropriate? [**2 marks**]

What additional issues does the virtual memory management system have to address when dealing with shared-memory multiprocessor systems? [**5 marks**]

2001: Paper 4 Question 7

In the context of virtual memory management:

a) What is *demand paging* ? How is it implemented ? **[4 marks]**

b) What is meant by *temporal locality of reference* ? **[2 marks]**

c) How does the assumption of temporal locality of reference influence page replacement decisions? Illustrate your answer by briefly describing an appropriate page replacement algorithm or algorithms. **[3 marks]**

d) What is meant by *spatial locality of reference* ? **[2 marks]**

e) In what ways does the assumption of spatial locality of reference influence the design of the virtual memory system ? **[3 marks]**

A student suggests that the virtual memory system should really deal with "objects" or "procedures" rather than with pages. Make arguments both for and against this suggestion. **[4 and 2 marks respectively]**

.

2003: Paper 3 Question 5

Modern server-class machines often use a Redundant Array of Inexpensive Disks (RAID) to provide non-volatile storage.

a) What is the basic motivation behind this? [**2 marks**]

b) Describe RAID level 0. What are the benefits and drawbacks of this scheme? [**3 marks**]

c) Describe RAID level 1. What are the benefits and drawbacks of this scheme? [**3 marks**]

d) Compare and contrast RAID levels 3, 4 & 5. What problem(s) with the former pair does the last hope to avoid? [**6 marks**]

A server machine has k identical high-performance IDE disks attached to independent IDE controllers. You are asked to write operating system software to treat these disk as a RAID level 5 array containing a single file-system. Your software will include routines to read filesytem data, write filesystem data, and to schedule these read and write requests. What difficulties arise here and how may they be addressed?

[**6 marks**]

2003: Paper 4 Question 5

Describe the basic operation of a *log-structured file system*. What are the potential benefits? What are the problems? [**8 marks**]

Several modern file systems make use of *journalling*. Describe how a journal is used by the file system, and the situations in which it is beneficial. [**6 marks**]

You are assigned the task of designing a tape backup strategy for an important file server. The goal is to maximize the time any file is held on backup while minimizing the number of tapes required. Sketch your strategy, commenting on the support you require from the file system, and justifying your design decisions. [**6 marks**]

2004: Paper 3 Question 5

(*a*) What problem do real-time schedling algorithms try to solve? [**2 marks**]

(*b*) Describe one *static priority* and one *dynamic priorty* real-time scheduling algorithm. You should discuss the issue of admission control, and comment on the data structures that an implementation would need to maintain and on how these would be used to make scheduling decisions. [**8 marks**]

(*c*) A designer of a real-time system wishes to have concurrently executing tasks share a data structure protected by a mutual exclusion lock.

(i) What scheduling problem could arise here? [**2 marks**]

(ii) How could this problem be overcome? [**2 marks**]

(*d*) The designer also wishes the real-time system to use demand paged virtual memory for efficiency. What problems could arise here, and how could they be overcome? [**6 marks**]

2004: Paper 4 Question 5

(a) Most conventional hardware translates virtual addresses to physical addresses using *multi-level page tables* (MPTs):

(i) Describe with the aid of a diagram how translation is performed when using MPTs. [**3 marks**]

(ii) What problem(s) with MPTs do *linear page tables* attempt to overcome? How is this achieved? [**3 marks**]

(iii) What problems(s) with MPTs do *inverted page tables* (IPTs) attempt to overcome? How is this achieved? [**3 marks**]

(iv) What problems(s) with IPTs do *hashed page tables* attempt to overcome? How is this achieved? [**3 marks**]

(b) Operating systems often cache part of the contents of the disk(s) in main memory to speed up access. Compare and contrast the way in which this is achieved in (i) 4.3 BSD Unix and (ii) Windows 2000. [**8 marks**]

2005: Paper 3 Question 5

(a) Modern operating systems typically support both *threads* and *processes*. What is the basic difference between a *thread* and a *process*? Why do operating systems support both concepts? [**2 marks**]

(b) You get a summer job with a company which has an in-house operating system called sOs. sOs uses static priority scheduling, supports at most 32 concurrently-executing processes, and only works on uniprocessor machines. Describe with justification how you would modify sOs in order to:

(i) support up to 65536 concurrently executing processes. [**2 marks**]

(ii) reduce or eliminate the possibility of starvation. [**3 marks**]

(iii) efficiently schedule processes on an 8 CPU symmetric multiprocessor (SMP) machines. [**5 marks**]

(iv) support threads in addition to processes on SMP machines. [**3 marks**]

(c) How would you go about reducing the time taken to boot a modern operating system? [**5 marks**]

2005: Paper 4 Question 5

(*a*) Scheduling disk requests is important to reduce the average service time. Describe briefly how the SSTF, SCAN and C-SCAN scheduling algorithms work.

[3 marks]

(*b*) Recently there have been proposals that *2-D disk scheduling* algorithms should be used. What is the basic idea behind 2-D disk scheduling? **[2 marks]**

(*c*) You are asked to develop support for 2-D disk scheduling in a commodity operating system.

(i) What would be the major difficulty that you face?

[2 marks]

(ii) Sketch a design for how you would go about overcoming this difficulty, and comment on how well you think the resulting system would work. **[8 marks]**

(*d*) Several modern file-systems and databases make use of a *journal* to aid in crash recovery.

(i) Briefly describe how journalling helps crash recovery.

[2 marks]

(ii) A researcher suggests adding 128 MB of NVRAM to a disk drive and using this to store the journal. Discuss the advantages and disadvantages of this approach.

[3 marks]