Advanced Systems Topics, Section 1 of 3
– Sketch Answers for Sample Questions –

SMH

Lent Term 2004

A Caveat

These are some *answer sketches*; i.e. I'm not trying to say this is all that can be said, or what these are somehow a 'model' answer to the questions. However it should give some indication of the sort of thing I was thinking about when I set the questions.

Note that you will gain most by not looking at these sketch answers until you've attempted the questions yourself (either as "self-study" or for supervision work).

Sketch Answer 1

DSVM is nominally easier for programmers to use since it exposes the same paradigm used when writing multi-threaded applications. In terms of performance, however, DSVM is pretty much guaranteed to be worse in all cases (one reason for this is "internal fragementation" – every page partially filled by a shared data structure (or value) needs to be copied in entirety; the other main reason is the potentially multi-hop comms required to track ownership, copysets, etc). DSVM over a WAN is a pretty stupid idea; we often have multi-party communications and multiple roundtrips per logical update. This kills us in the LAN; the wide-area just makes it worse. Finally, DSVM doesn't work so well for robustness; with centralized schemes we have a central point of failure, while with decentralized schemes (multi-party tracking of ownership), the failure of any one node will likely result in the disappearance of a whole range of virtual address space. Some or all of these problems may allow solutions (e.g. using P2P and/or redundancy coding).

Message passing is a bit of a pain for programmers, although it may often be hidden under a layer of indirection (viz. RPC/RMI etc). Works better if people are using 'object' based access in the first place. Performance is pretty good since we only transfer the data required. Works in the wide area too, even though will drop in performance; need to be careful about number of round-trip times in this case (e.g. batch data for transfer, or use multicalls). Finally, we normally can survive the failure of all machines save for the two involved in a particular communication; sometime external infrastructure (e.g. registry, DNS) can cause an implcit dependence on central resources though.

Sketch Answer 2

Although Multics had an "abstract" design, many of the ideas were heavily influenced by the features supported by the GE-645 hardware. The two most notable of these are support for (i) hardware paged segments and (ii) concentric rings of privilege. The former were pretty complex (see notes page 9), with a descriptor segment (which was be paged) holding decriptors for page tables for segments. Much of the "idealized" Multics virtual memory required using these structures as 'caches' to provide the illustion of an arbitrary number of orthogonal user-named segments. This means ended up with an "active-segment table"' (AST) and a set of "known-segment tables" (KSTs) along with the descriptor tables and page tables described above. To implement all of this on e.g. the ARM would require faking out segments as page-aligned regions of virtual memory, and building all the descriptor table, AST and KST gubbins on top.

A larger problem is posed by Multics' model of concentric rings (and ring brackets). In essence this allows more flexibility than a simple user-supervisor mode split (effectively what the ARM has, fiq and irq modes notwithstanding). To implement all of this on an ARM would require a software-maintained notions of "current ring" per process or thread, ring brackets, limits and gates per (fictional) segment, *and* a protected control transfer mechanism in the "kernel" to allow transfer through gates. It'd be a considerable piece of work, although possible, and would have huge performance impact. OTOH, would probably be faster than any previous existing version of Multics :-)

Providing h/w suport to fill these gaps would almost certainly *not* be justified; experience with processors over the past 30 years demonstrates that although RISC per se may not win, putting additional complexity on chip tends to (a) slow down the critical path (b) make chips more expensive and (c) make OSes more difficult rather than less difficult to port.

Sketch Answer 3

Using "enter" capabilities (which is what e.g. the CAP called the above) allows very fine-grained protection of resources – in essence, it allows us to use the principle of least privilege (c/f Security IB) efficiently and pervasively. As an example, consider a local file system existing on a single partition. The code implementing the access methods for this may need to be able to read and write blocks anywhere within that partition, but they certainly should not be able to access other random kernel things (e.g. network, periodic timer, scheduler queues, virtual memory data structures, etc). It also does not require any access to e.g. the general data of the calling process. All it needs is: read (or read-write, depending on the operation) access to a buffer belonging to the caller, and read (or read-write) access to the underlying partition. In general, a capability system allows us to build a *component-based* OS in which each component has only those abilities it strictly requires to perform its function(s). Furthermore the use of a single thread which "tunnels" means that resource accounting is simplified, and scheduling simplified (no need to block the calling thread, or wake up a "service" thread, etc).

This is different than message passing IPC in MACH in two key ways: (1) there is a "kernel" of some kind in the MACH world which provides IPC as a basic primitive; in (hardware) capability systems, this function is provided by the hardware; as a consequence there is no need for *any* omniscient software component (i.e. there's no need for a 'kernel' of any kind), and (2) MACH IPC involves a calling thread sending a message and then becoming blocked; a service thread (in another task) which has blocked on incoming messages is then woken on receipt, and proceeds to perform whatever action; on reply, the opposite occurs; parameter passing is difficult since address spaces are different. Shared memory can be used but often requires copying; alternatively protection can be ignored (c/f L4). Neither of these seem great.

One could implemented a software enter capability on a commodity process by implementing this as a basic primitive in a microkernel rather than a message passing IPC. This is pretty much what EROS does (albeit for x86 rather than ARM).

Sketch Answer 4

VMware-style VMM: main advantages are binary compatibility (i.e. don't need to change hosted operating system or applications); means can run multiple OSes out of the box (in theory), and can provide strong resource isolation / partitioning; disadvantages are that cost of doing this may be high on some architectures (e.g x86); and that OS arguably *must* be modified somehow to deal with fact that it's not running on the "bare metal" anymore (e.g. consider virtual hardware timer interrupts and real-time issues (e.g. TCP timeouts)).

Denali/Xen VMM: main advantages are improved performance on un-cooperative CPU architectures (e.g. x86), plus the ability to customize the hosted guest OS to handle running on a virtual machine. Keep strong isolation properties of VMWare-style VMM too. Key disadvantage is loss of (operating system) binary compatability, although Xen at least preserves application level binary compat. Requiring source modifications to an operating system may not always be possible / easy for non-technical reasons (e.g. commercial OSes).

Vserver approach: main advantage of this is that it can be *extremely* lightweight – in many cases one is simply tagging a set of processes with a common tag and hence there is very little overhead to virtuaization. Key downsides are (a) the {fault,performance}-isolation between virtual machines is pretty non-existent and (b) pretty much limited in practice to k virtual machines all of which run the same "OS" as the hosting OS.

Question 5

Note: this question is mainly "bookwork" — see notes pages 30–41 for more detail.

An application programmer might like to customize the operating system for e.g. specialized thread scheduling algorithms (worker pool implementation, parallel multiprocessor applications), customizing page-replacement or buffer-cache replacement schemes (e.g. when know what would be a 'good' thing to replace), avoiding "shadow paging" in a DBMS, etc.

An operating system might want to be able to do this to e.g. install support for a new device, filesystem or network protocol, or to adapt to current workloads (dynamic specialization – including trace-based schemes).

Techniques to allow this may broadly be classified as user-level and kernel-level; the former include microkernel approaches (user-level "servers" as part of the OS), vertical operating systems (most OS functionality as shared libraries) and hypervisor approaches (entire OSes hosted on top of a hypervisor/VMM). The latter approaches involve the transfer of code from a user-space applciation into a (privileged) kernel. While the user-space schemes mainly involve refactoring OS design, the kernel-level techniques require no such refactoring: but *do* require some method to ensure the safety of the downloaded code.

Schemes for ensuring safety include proof-carrying code (whereby we verify the behaviour of the down-loaded code is 'ok' according to some policy – we require quite a lot of baggage to make this work including a formal semantics of the language in which the extension is written) and sandboxing schemes (where we transform 'regular' code by inserting run-time safety checks). Various OSes extend this latter approach to attempt to deal with resource abuse issues (e.g. hogging cpu, holding locks, allocating lots of memory) by a combination of quotas and revocation (including transactional 'undo' where possible).

Question 6

Note: the topics covered in this question were lectured in AST last year (2003) since that class had not attended OS II. This year's class will have covered these topics in an earlier course.

A log-structured file system (LSFS) works by treating the disk as a non-volatile append-only infinite buffer. Every data and meta data update (write operation, directory modification, etc) is performed by appending an update-record to the end of the log. This avoids costly seeks (and makes disk head scheduling trivial). Reads logically proceed by finding the most recent version of all relevant blocks (directory entries, inodes and data blocks). The key potential benefit is increased performance and throughput on write operations.

The problems are multiple. Firstly, locating things (inodes, data blocks, etc) on the disk is tricky since no-idea where to begin. This can be 'solved' by occasionally writing checkpoint records at fixed (well-known) locations on disk. Checkpoints can contain pointers to *inode maps* which themselves contain pointers to inodes. Once an inode is located, the relevent file or directory blocks are located from there.

The second and more important problem is that the disk is not actually infinite. Hence the operating system must periodically compact the log by removing redundant (old) blocks. This is a major pain, since naive compacting of entire log means lots of random access reads and writes. Threading through log gives fragmentation issues and really just defers the random access death to later. 'Best' solution probably segmented logs, but even then choosing correct segments to clean is tricky.

A journal is essentially a non-volatile append-only infinite buffer as above. However unlike LSFS, this buffer takes only a small part of the disk and contains only *metadata* update records; the standard filesystem strcuture (ext2 or ntfs or whatever) is used in the remainder of the disk. Journalling is used, then, not as zero-seek optimization but rather in a way akin to a transactional log (unfortunate name overlap here).

The point of the journal, then, is to allow the efficient return of a file-system to a consistent state after a crash. This is particularly useful on modern large disks and filesystems since traditional 'scavenging' (e.g. via fsck or scandisk) typicaly takes time proportional to the size of the filesystem. Journal recovery on the other hand takes time proportional to the number of uncommitted metadata operations.

Which is 'better'? Tricky to give a precise answer, but based on the decriptions above it's fairly clear that I believe the journalling scheme is by far the more sensible...

Question 7

Sketch structure of B+-tree: c/f notes page 48. Lookup for a key k takes place by performing a binary search at each level (starting at the root) to locate (e.g.) the smallest key p greater than k. We then follow the pointer to the r.h.s. of p and continue until we hit the leaf. All valid keys are present in some leaf node $\Rightarrow$ we can detect success or failure only by traversing the entire depth of the tree.

Insertion of a record with key k proceeds by first doing a look up for k and then (assuming it doesn't exist) inserting an entry for k in the leaf node we ended up at. If this leaf is full, we need to split it; this means inserting a new entry in the parent. If the parent is full, recurse upward; if we hit the root, we add a new level to the tree.

Choice of node size n is generally influenced by practical limitations and performance trade-offs; as an example of the former, we may find that multiples of "sector size" are sensible in order to avoid wastage on disk – this effectively quantizes our choices. To select a particular solution, we need to consider the trade-off between the binary search time at each level ($= O(\log(n))$) and the number of disk reads ($= \log_n N$, where N is the number of records in the tree). Empirical evidence on a particular machine (and incorporating assumptions about node cache size and replacement policy) can be used to produce a quantitative answer.

B+-trees are nice for DB systems since (a) they allow us to perform equality matches extremely efficiently (b) they allow us to perform *range* queries fairly efficiently (depends on n, exact query, and actual data), and (c) every node is identical in size. They are also relatively simple if we ignore balancing after deletion.

Note: the 'answer' to the last bit of this question wasn't lectured; instead wanted to see what people would come up with.

Building distributed/parallel databases is a research topic, as is building efficient indices for them. Two extreme approaches to distributing data are (1) put different tables on each machine (e.g. in simple case, one table per machine) which allows parallel selects and (some kinds of) efficient joins, and (2) stripe every table every machine (so e.g. in simple case each machine has an N^{th} of the entire database), allowing parallel joins (w/ subsequent merge).

In the former case, an index for a particular table should be built on that machine and (perhaps) replicated to other machines, or to a (set of) front-end query processor(s). Updates to a given table are localized to a given machine, and so rebuilding the index is easy – if replicas have been made, an "invalidate" or "update" will be required after the new index comes into existence. Using sparse indexes can avoid even this in many cases, allowing lazy update. Performing queries in this world really requires a query processor or other front-end which knows about (the indexes of) all tables.

In the latter case, each node can build a *partial index* for those portions of each table it owns. We can then consider these partial indexes to be second-level nodes in a distributed B+-tree, and replicate the root widely. This allows query processing to proceed in parallel (i.e. any machine can perform it, redirecting index or data requests to relevant other machines as appropriate). However updates become more difficult; in the simple case, a distributed locking scheme is required for any updates (since they may involve splitting the root). Various optimizations may be applied, including reducing the strictness of balance (e.g. pi-trees) and or building new indexes in parallel with using old ones, and atomically replacing the root periodically (e.g. altavista style).

Question 8

NFS is based on a client-server model; various clients 'mount' a particular file-system via the mount protocol, and then access files, directories, etc, via Sun RPC. NFS V2 is stateless; no record of clients or open files, no implicit arguments to requests, no write-back caching on server. This means performance can suck somewhat. NFS V3 has a bunch of optimizations (including larger transfer size and 'asynchronous' writes) which make performance a lot better; NFS V4 is a funky new thing which doesn't really exist yet. See notes pages 57–58.

All NFS implementations involve a (at least logically) central server; this can become a bottleneck, since all requests need to pass through this (not just control stuff, but also all data reads, writes, etc).

xFS, by contrast, avoids a single central server. Instead a collection of nodes cooperate to provide a distributed storage system. Different machines are nominated as 'manager' for a given file / set of files, and this information is propagated in a globally replicated manager map. Using this means that a client knows who to contact in the first instance to obtain access to a file. However actual storage is also replicated, meaning that data transfers (e.g. reads and writes) are effectively "striped" across a bunch of machines. Fairly trivial to see for reads, but for writes could cause problems (read-modify-write!). Hence xFS uses a log-structure to ensure that all writes can be stripe size (and can be indept of other updates).

The systems as described are clearly different, both in terms of basic operation and in terms of achieved performance (xFS does a lot better than NFS v? in various benchmarks). However on the flip side, xFS is considerably more complex, not widely deployed, and brings with it the joy of a *distributed* log cleaning problem. Meanwhile NFS although requiring a logically centralized server can in practice use multiple servers for e.g. different parts of the name space (c/f amd, autofs, etc) or even have a more general partitioning of a distributed store (c/f "slice uProxy" paper, aka 'Interposed Request Routing for Scalable Network Storage', Anderson et al, OSDI 2000, available from course web page).