CSCI 3120

Deadlocks and More

McAllister

Alex Safatli

Tuesday, June 25, 2013

Contents

Advanced Synchronization 3
- Nested Critical Sections . . . 3
- Deadlocks . . . 3

Memory Management 5
- Allocation . . . 5
- Virtual Memory . . . 10

File System Interface 12
- Organization . . . 12
- Directories . . . 13
- Partitions . . . 14
- File Systems . . . 14
- File Organization . . . 15
- Redundant Array of Inexpensive Disks (RAID) . . . 18

Access Control 18
- Security . . . 18
- Protection . . . 19

Review: Making a Debugger 20

Advanced Synchronization

Nested Critical Sections

How do we feel about *nesting* critical sections? Oftentimes, a critical section is associated with a piece of data being changed: these are to avoid Bernstein's Condition. Critical sections are all around protecting some manner of data. In that manner, one could have a transfer of some data from one structure to another while in the lock of a specific structure.

In the case where each process must wait for another and neither one will advance because of nested sections (see corresponding slide), we have what is known as a **deadlock**. Independently, the code is fine; together, their interaction causes a conflict.

Deadlocks

A **deadlock** is a *permanent blocking* of a set of processes that either compete for resources or communicate with each other; a deadlock happens when each process holds a resource for which another process in the set is waiting. Remember that a process in the blocked state is essentially not using resources, but this is *permanent* until the process has ended.

A solution could be to constantly switch between states in order to try and get rid of a deadlock. As an aside, a **livelock** occurs when a set of processes continue to change (not blocked) but none of their processes progress toward completing their respective tasks. This is often considered a case of deadlock but where the CPUs are consuming cycles.

What is needed for a deadlock? We have to recognize it in order to fix it. We will talk about four different things that must happen which can be broken down into two conditions (**necessary conditions**): policy and dynamic.

Policy conditions are components of the environment that are static. These include mutual exclusion, there must be a scenario that allows hold and wait (a process must be able to hold a resource while waiting for another resource to be allocated to it), and no preemption (resource preemption must not be possible). These depend on the operating system and environment.

Dynamic conditions involve what is changing and evolving while running. These include circular wait (a circular sequence of processes, each of which holds a resource needed by its predecessor and its successor). One waits on one and the other waits on the other. This can be a cycle of N processes.

NOTE A way to model this will be using an **allocation graph**. Resources are dots in the rectangle, arrows represent waiting. It is almost as if the waiting time guarantees the deadlock. However, if changed, this can be solved; one could have more than one instance of a resource.

A cycle in an allocation graph is a necessary but *not sufficient* condition for a deadlock. Complexity comes in when there are multiple instances of a resource; with just one instance of each resource, we can create a "wait for" graph instead and look for a cycle in that graph (it becomes sufficient).

To handle deadlocks, we will try to manage critical sections in such a way that an out is guaranteed and one can finish the system without a deadlock. We can call this a **safe sequence**. Dealing with deadlocks ultimately involves the same method mentioned before: prevention, avoidance, and detection/recovery.

Prevention is handled by making it impossible for a deadlock to happen. Avoidance involves setting up rules that, if followed, will not let a deadlock happen. Finally, detection and recovery means that we let deadlocks

Deadlocks continued on next page...

happen, find them, and then fix them. Like before, there is a difference in cost here.

Generally, prevention could be expensive or make little use of resources. Detection is simple to implement, low cost, but when things go wrong, there is a large mess to clean up. There is a spectrum of trade-offs.

Deadlock **prevention** can be guaranteed (e.g., making deadlocks impossible) by taking away necessary conditions and making one or more of them impossible. These generally give low use of resources and/or low levels of concurrency when resources are in high demand. Conditions include:

- *stop mutual exclusion* — probably not a good idea; defeats the purpose of a critical section,
- *stop hold and wait* — force a process to release all resources before acquiring new ones; allocate all resources to a process at one time at creation,
- *allow preemption* — let processes steal resources from another (will not work for all resources), and
- *stop circular wait* — order all resources and only allow resource requests provided in the given order. Note: This is used in the Linux kernel; locks must be acquired *in order* but this is not done on runtime – it is ensured while programming.

Deadlock **avoidance** is the idea of creating a procedure that, if followed, gives us a path through which a deadlock *will not happen*. This includes guaranteeing that there exists a schedule of processes in which a deadlock will not happen. This brings us to the idea of a **safe sequence**. It is a sequence of processes $P_1, P_2, ..., P_n$ such that the total available resources after $P_1...P_j$ have completed are sufficient to satisfy P_{j+1}'s maximum possible declared needs for $1 \leq j \leq n$. Formally, a set of n processes is safe (deadlock free) if there exists a safe sequence.

NOTE The above can be considered as an analogy: when banks give out loans, they can later call them in in order to get enough money for a large withdrawal.

The assumptions made here is that we know the maximum number of resources a process might request (pre-declare, guess, or a fixed worst-case limit e.g., descriptor/file limit) and that a process can request more than one resource at once. An operating constraint here one can make is that the OS could delay the granting of resources so that it can "batch" multiple requests together.

The idea here is to grant resource requests as long as a safe sequence still exists; at worst, the "fair" scheduler will end up running the processes in the order of the sequence and will avoid a deadlock. For new processes, the long term scheduler does not admit them to the ready queue unless the overall set of processes *is safe*. We can do this using the **Banker's Algorithm** (note that for the 2D array, if one index is given, entire row is being looked at).

NOTE In the example slide, the need matrix is the difference of allocated and maximum (these model steps 5-7). P4 can finish first so it is served first. Note also that not having a safe sequence does not mean it will result in a deadlock: only that it *could*. If an operating system refuses to allocate a resource, a programmer will typically get a `NULL`.

The running time of this algorithm is $O(mn^2)$ (m: resources, n: processes); it is expensive to run so requests are typically batched so it can be run infrequently. In reality, not all processes need to be looked at in this algorithm — only those that have needed resources less than those available and those that are non-zero. Furthermore, it may not have good apriori maximum resource estimates and few processes ever reach their maximum requests (it is a conservative and inefficient use of resources).

Deadlocks continued on next page...

NOTE Using an analogy, if there is a pothole that we do not want to drive in, prevention would involve paving it first, avoidance would mean finding a path that does not let us fall in, and detection would be to see when someone drives over and then to find a way around for now for all subsequent cars.

Deadlock **detection** is the idea of looking at the ”wait for” graph if all resources have a single instance (sufficient) or, in the case of some resources having more than one instance, it is hard to tell if a process is deadlocked or just waiting around. Therefore, the Banker's Algorithm can still be used: use the request array as the need array. As long as all requests can be satisfied, all of them can be granted. Note that since we're doing detection, this algorithm can be run less frequently (whenever a problem is found).

Recovery can involve a number of solutions: *crashing the system* (no happy users; could be done when nobody is using the system), *terminating a process* (must roll-back effects of process on that resource; options include killing all deadlocked process which is extreme or to remove 1 or more to eliminate it but this is hard to tell – NP-hard), or *preempting a resource* (select a ”victim”, roll them back to before a resource was allocated — this could starve the process).

NOTE Heuristics for determining which process to terminate involves stopping the process with the most resources (most resources are retrieved, but have to restart it too), stop the most recent process in the deadlock (it may have triggered the deadlock), to stop the process with the least resources, stop the oldest process, stop the process waiting for the rarest resource, stop a process at random, or use priority (lowest).

NOTE Stopping a process should not make a deadlock worse if the operating system is designed well: spare resources and processes should be available in reserve.

Options can also include, as in our database discussion, log-based recovery, checkpoints, or serializability. Realize, though, that the situation that can result in a deadlock is not always guaranteed.

Memory Management

Allocation

Note that **memory management** concerns itself with relocation, protection, sharing, logical organization, and physical organization. Note, also, that the techniques discussed in this section could be used on other media as well (can involve themselves with memory systems, backup systems, etc.).

Relocation is a question of whether or not we are able to move processes and, if so, how? **Protection** is the idea that we want to keep processes out of the memory of other processes. **Sharing** is a question of whether or not we let processes share memory. **Logical organization** is a question of how memory look like to the programmer. **Physical organization** is how it looks like to the hardware.

So, managing what we have: physical memory. There are two issues to tackle:

- When do we allocate physical memory?
- What physical memory do we allocate?

Allocating is tied to letting a program or a process know the memory address it will use; this is called **address binding**. The later the binding, the more flexible the system (compile time, assembly time, linking time, load time, runtime, etc.). Address binding them can be **absolute addressing** (compile time), **relative addressing** (link time), with **dynamic linking** (runtime).

Allocation continued on next page. . .

A **logical address** is an address seen by, used by, and generated for a process (what the programmer thinks about, an offset). A **physical address** is an address seen by the hardware; where the actual bit is stored on the memory chip. Policy choice could involve having logical and physical addresses be the same or different (and need to map between them all the time; simplest approach is through a memory base register).

Physical address space assignment is the idea of allocating physical memory in the same way that programmers understand their process memory. Programmers think of memory as one contiguous block of memory; that is how we will start assigning memory.

Contiguous Memory Allocation is where each process is given some or all of the physical memory; number of processes limited by the amount of physical memory. The limit is overcome with *swapping*; memory is copied to disk temporarily if we need currently-used physical addresses for another process (most often done with static and compile-time address binding). This is not typically done with computers but can be done with DVDs, etc.

NOTE *Swapping* can be used for suspending and resuming the five-state model for processes. The dispatcher would not perform this disk copying; this would be better suited for the short-term scheduler. The kernel and system services will remain in memory.

NOTE Consider a data transfer from a slow device into a process' memory space. Transfer options include, once the device driver has started, copying the data from the buffer space (some disks will have their own on-board buffer and so does the OS and each process, etc.) into the memory space (simple, but not efficient use of CPU time; copies data twice) or can write directly to the process' or OS' memory space — called Direct Memory Access [DMA]; efficient but driver has access to process' memory space – OS not involved – and the process cannot be swapped out).

Given a new process, how is it determined where to put the process in memory? Locations include: static partitions, dynamic partitions, and hybrid (binary buddies). The general trade-off here is that static is easier to do but can waste space; dynamic has more work to be done on the fly but can use more resources.

Static partitions is the notion of dividing memory into *fixed-size partitions*; can have all equal-sized partitions or a number of sizes available. When a process is created, it is assigned to one of these partitions. If none are available, the process is not admitted to the ready queue or an existing process is swapped to the disk. Using RAM, spreading out *v.* grouping partitions does not really have a difference — on a hard disk where memory access time between different locations is not negligible, it could.

Assigning a process to a given partitions may be a question of finding the first one that fits, or the smallest available (common) or biggest available partition (that fits). The OS then just needs to remember which partition was allocated to which process.

Advantages for this include that it is easy to manage blocks, easy to swap out a partition, and has little system overhead. Disadvantages, on the other hand, include an inefficient use of resources (can have **internal fragmentation** to occur) and that the number of partitions *limits* the number of processes.

Dynamic partitions is the idea that, when creating a new process, exactly enough space is carved for the process from some block of unallocated memory. A list of blocks of unallocated memory are kept (called the **free list** — can be stored within portions of the free blocks themselves). Note there may be some manner of a minimum size, as well, and that the block sizes will be a multiple of computer word length (2^k). As processes are allocated memory and that memory is destroyed, holes will appear: this can be fixed by moving processes but this requires a great deal of time (memory compaction).

Policies for dynamic partitioning involve:

Allocation continued on next page...

- which unallocated block do we use? (e.g., do we use a search strategy in the free list)
  - the first one that we find that fits? (fast, keep searching from where we last left off — first fit *v.* next fit algorithm; no real performance differences between the two),
  - the smallest available partition that fits? (search the whole list, hoping for optimality — best fit algorithm; could use a skip list but would have to maintain that data structure), and
  - the biggest available partition? (search the whole list — worst fit algorithm; can be considered in a situation where we want to take small pieces off the big blocks to avoid **external fragmentation** since this block can be later merged).
- where do we carve off the space in the block? at the top? bottom?

When free space is not contiguous for an arriving process, we may need to swap out a process.

The idea of **fragmentation** is to do with a division of memory into unusable pieces. External or internal fragmentation differ based on what is making the pieces unusable. **Internal fragmentation** is where unusable pieces of memory are allocated beyond the needs of the processes (typically happens with static partitions). **External fragmentation** is where unusable pieces of memory are arranged in a manner that makes them unusable; this typically happens with dynamic partitions (can happen with static partitions when we run out of partitions of specific sizes).

Memory compaction is the idea of reorganizing dynamic partitions to move all unallocated memory into one (or a small number) of blocks. This is meant to reduce external fragmentation; this is analogous to defragmentation of a hard drive. The algorithm approaches differ here: do we move all processes to one end of memory, move the fewest number of processes or bytes (copy less files; tend to be *NP-complete*)? Will typically use a heuristic of some sort (may use the same algorithm used for dynamic partitioning). This is only done when memory is *needed immediately.* Typically, do this only to clean out a hole that is needed and no more.

The advantages of dynamic partitioning is that it has no internal fragmentation and a better use of memory. Disadvantages include that they are more complex to implement (and external fragmentation).

The **buddy system** (binary buddy system) is a hybrid system that aims to get the benefits of both static and dynamic partitioning. The idea is to choose a set of fixed-size partitions, always keep the biggest partition possible around, divide into smaller partitions and only use a best-fit piece, and recombine split partitions when both halves are free, only recombining to create valid-sized partitions. Realize you cannot have more than half a piece's worth of internal fragmentation.

NOTE Programming question on *Assignment #4*: using multiple threads is not a technique many of us have ever attempted. A place to start is to do so thinking about it as a problem solved by multiple people/friends. Think of bucket sort. Note that part of the marks will be based around time. Note the use of the utility `time` in Unix.

Implementation of Binary Buddies is done by keeping a ”free list” of each partition size. When a partition is split, the free block is moved into a smaller free list. Refinement that can be done includes using any recursive subdivision as partition sizes (e.g., Fibonacci buddies, Catalan buddies). Advantages of this system is that it remains relatively easy to manage, we are still dealing with fixed-size blocks to copy data, and it tends to still leave big blocks in memory. Disadvantages are tat it has more overhead to manage (remembering and recombining buddies).

Note that we will often see the opposite effect of this for managing space that can be grown (see `ArrayList` implementation); suppose you want a dynamically growing array in C. The array can be created using

Allocation continued on next page...

`malloc()` with a certain numbers, and it will grow by doubling (each re-allocation copies half of its data).

In reality, we talk about **non-contiguous memory allocation**. They allow physical memory for a process to be split into multiple pieces; options include **segmentation** and **paging**. Segmentation is where the programmer is aware of the non-contiguous nature of memory. Paging is where more hardware is made to help make the logical address space look contiguous. A hybrid system is also possible here.

Segmentation predates paging. It is based on a recognition that a process uses different parts of memory differently and we might want to treat that memory differently; code, stack, and heap segments. Different memory protection systems can be done for each of these segments. A variable-sized partition is used for each of these different parts. A *base register* is defined for each part and each segment is then independently available — three base addresses are available and are used as a relative location in memory. The process being forked might have an estimate for the heap and stack (code is known) but it is not guaranteed to be known.

Early computers had a fixed number of subjects (4; the above three and a user-defined segment). A finite and respective number of specialized registers were available for these segments. As a programmer, more flexibility would be beneficial. Therefore, we have moved away and have a flexible number of subjects; as soon as we do this, though, we need to pull back from the hardware a bit. We can record, in the logical address, what base register the hardware should use as an offset to get to the real data.

The register number is therefore encoded in the logical address. Compare this to **subnets**. Ultimately, the above lets us move away from a fixed set of registers to a table of base registers called the segment table; the segment ID will point to a table of base pointers (one registers in hardware will point to a table of pointers). But this table can hold more than just pointers; it can store more information about these segments (memory protection, segment length, whether or not it is in memory, was previously swapped out, or has been changed recently, for instance).

Segmentation faults are where an invalid segment ID is provided that the OS cannot deal with. Segmentation can be static or dynamic; the latter will be talked about more often. Static is where all segments are declared and set-up before executing. Dynamic is where segments can be created dynamically and changes to the segment table happen at any time. Consequences of this method, though, are important. The hardware can check for offsets that fit into the segment (thus setting off an interrupt if need be), not all logical addresses can be used. In a segmentation world, programmers have a mental model where not all addresses are the same: limitations of segmentation. Note that each segment is intended to carry a set of related data.

Advantages of segmentation is that it is more efficient than the contiguous schemes. It can also let us share segments between processes when we want to do so (e.g., running two instances of the same piece of code). It also allows us to protect some parts of memory without locking up all of memory. Furthermore, this would allow more flexible relocation than with contiguous memory allocation schemes. Finally, we can swap things to disk.

Disadvantages are that the OS must keep track of variable-sized partitions (external fragmentation), logical addressing puts a limit on the size of a segment and on the number of segments (cannot change these sizes on the fly) and that there are logical addresses that do not map to any part of memory (the programmer sees holes in his or her logical memory space, running off the end can cause programs to crash).

The aim of **paging** is to make the logical space look *contiguous* again. Fixed-sized partitions are reverted to (single partition sizes; easier to manage). Physical memory can be divided into equal-sized pieces called *frames*; logical memory can be divided into equal-sized pieces called *pages*. Pages are assigned to frames and the logical address is split into a page number and offset (with a size equal to the frame size; therefore no checks are needed).

All pointers to frames are stored in a *page table*. The page number is used to identify the appropriate row in

Allocation continued on next page. . .

the page table to find the appropriate frame. Page table entries can be augmented with bits to say whether or not the page has been assigned to a frame, or if any data has been changed recently. Paging can be static or dynamic; static is where the page table is populated before execution and dynamic is where pages are assigned dynamically. Note we have left the notion of protection (in theory, you could, but unlike in segmentation, one page does not necessarily contain one type of data structure).

This differs from segmentation in that it helps keep the page table indices *contiguous* while the physical memory may not be; a single page could be used for more than 1 process and therefore protection cannot be used.

Multi-level page tables are page tables where page tables are paged themselves; contiguous entires of a page table can be grouped into a frame and indirection tables are made to track this (empty frames do not have to be assigned). The page table index can be subdivided into two indices. More levels mean more indirection (more memory accesses) to get any datum in memory.

Here is an example of calculating the memory necessary for a certain page table scenario:

```
memory: 32 bit addresses
frame size: 2^10 bytes (1 KB)
page table entry size: pointer (4 bytes), status bit (4 bytes, 1 word) --> 8 bytes
page table entries in a frame: (2^10)/(2^3) = 2^7
where the numerator is frame size and denominator is bytes per entry

logical address:

<----------- 32 ------------>
| |   |   |   |   offset    |
<1>< 7>< 7>< 7><---- 10 ---->

this is a 4-level page table.
```

Large page tables can be dealt with with solutions including **translation look-aside buffers** (TLB) and **hashed page tables**. The former deals with associative memory, like a cache, can store mappings from logical pages to physical frames. This does not save on space (associative memory is expensive) but saves on number of memory accesses to get to one datum. The latter involves putting the page table rows into a hash table; pages are located by hash table look-up — requires a bit more computation but it is one-stop shopping.

Another solution is to use an **inverted page table**. This comes out of an observation that each frame can be allocated to only one process and that each process might be possibly tracking it. Therefore, a table can be created with one entry per frame: each table row contains the process ID and the page number in that process whose data is in the corresponding frame. When a process X wants to access page Y, the page table is searched for X,Y and we can get back the corresponding frame if there; this could be stored in a hash table or in associative memory for faster searches. Storage space for this is *independent* of the number of processes, therefore saving on space.

Paged segmentation is an aim to get the benefits of both of these memory systems: related data is grouped into **segments** (few but big segments, allows for better protection and sharing), and applying paging to the segment space; the segment memory does not need to be contiguous and does not need to be completely allocated.

Memory Management continued on next page. . .

Virtual Memory

Virtual memory is a technique that allows us to execute a process not completely in memory (because we have not loaded all of the processes yet, we do not want to load all of the process, keeping error handling code on disk until needed, or because the process is bigger than the available storage where the OS is limiting how much physical memory is assigned to a process or because of hardware limitations). We focus on a *paging view of virtual memory*; this can be done with segmentation or paged segmentation, too.

There is some terminology necessary for this concept. **Locality of reference** is the *tendency* of a process to access *nearby memory addresses*: when running through an array of values, linearly travelling through the array. The **working set** is a reference to the set of pages of a process in which the program is executing within some (short) time interval. The **resident set** is the set of pages of a process in physical memory. **Paging** refers to the idea of copying a page to or from the disk (think swapping). A **page fault** is an error condition that happens when a process accesses a page for which the frame is not in memory (no memory assigned, or the page was copied to disk). Finally, **thrashing** refers to excessive paging: more time paging than executing.

Ideally, the working set size is smaller than the resident set size; the working set also changes over time and there is a need to adapt the size and content of the resident set to match. When the sizes are not well-matched, the working set is much smaller than the resident set means frames allocated could be used for another process (this is okay); working set larger than resident set implies thrashing.

The *medium term scheduler* detects thrashing; a process is suspended until that process' resident set size can be much larger.

To facilitate this mechanism, each page table entry is augmented with tracking bits: a **present bit**, a **modified bit**, a **read or read/write bit**, a **shared bit**, and a **locked bit**. When a memory access is made to page X, it is located in the page table for the process; if in memory, it is accessed. Otherwise, it is located by frame in memory (resident set), locate the page on disk and copy it back into the frame, modify the page table entry, and restart the access instruction.

There are a plethora of issues here. Where do the frames go on disk? What is the fetch policy? What is the placement policy in memory? What is the replacement policy? What is the resident set management policy? What is the cleaning policy?

Frames are stored in **swap space** or a **swap file**. Swap space is a partition on the disk specifically pre-formatted to store frames (typically, the primary swap alternative). A swap file is a regular file meant to hold frames (stored among other varying-sized files); size usually pre-allocated and fixed but this is typically the secondary swap alternative. These both have their own index structure to make finding frames efficient.

NOTE As RAM gets bigger, doubling the RAM is no longer wise advice for a swap space. For any process that is running, only a third of its memory is being used at any given time. Imagine, though: if this amount of swap space was being used, your computer is going to be in trouble (will lead into thrashing). Sort of converging on equal the RAM now.

The fetch policy has two different techniques: proactive or reactive. In general, being reactive is to **demand paging** (retrieve pages when a process requests a logical address not in memory; reactive to page faults). The alternative is **prepaging**: predicting future page references and retrieving the pages proactively: like pipelining in instruction execution, aiming to exploit locality of reference; fast page access when prediction is right but wasted disk accesses when prediction is wrong. This is not shown to be generally more effective due to the difficulty of this (and the great deal of time being put in).

Virtual Memory continued on next page. . .

NOTE Think pipelining for prepaging. When a branch point is hit, does not work so well.

NOTE The thread pool question in *Assignment 3* can be helpful on *Assignment 4* with the puzzle programming question.

NOTE A process excessively thrashing would hurt other processes because of more I/O.

Resident management policy involves having a set size of a resident for a process (fixed or variable – unbounded or bounded variable). If thrashing occurs, there is nothing that will be done. Another technique is to use replacement scope: on a local scale, when looking for a frame, only look to other frames owned by the process. On a global scale, look at all frames in memory and consider for replacement.

The cleaning policy involves considering when pages are written to disk. **Demand cleaning** is where pages are written to disk when a page fault happens and a page needs to be swapped to the disk. **Precleaning** is for dirt pages to be copied to the disk in anticipation of needing a frame later during idle times.

Replacement policy is incredibly important; this involves computation and is not merely a policy decision. When a new frame is needed:

1. Use a frame currently not allocated to any process,
2. Use a clean frame,
3. If all frames are used and dirty, then select a frame to swap to the disk (conforming with replacement scope) using an algorithm to select the frame: optimal, FIFO, second chance cyclic, Least Recently Used (LRU), not recently used, and counting-based page replacement.

Page reference sequence is the sequences of pages accessed by a process as it executes. With 4-bit page numbers, the page reference sequence for logical address accesses 0x5abe, 0x4402, 0x77c2, 0x77c6, and 0x0100 is 5, 4, 7, 7, 0.

The **optimal replacement strategy** keeps in memory pages needed in the *immediate future* to avoid having to page fault for that page; replace the page whose next use is farthest in the future. The problem here is that we need to know the *whole* reference string to apply the strategy (need to predict the future); this is generally only available after the program has run. Why bother with this strategy, then? It is used as a comparator for other strategies to see how far off from ”the best” we are.

For the **First In First Out (FIFO)** strategy, frames are replaced in the order in which they are allocated; cycling around the list of frames. Sometimes referred to as a cyclic strategy. Intuitively, having more frames gives more space for a process, so should have fewer page faults. **Belady's anomaly** is where there are page reference sequences for which the number of page fault increases if we use more frames. This is simple to implement but does not take into consideration the flow or use pattern of the process; this can be really bad for loops that need one more frame than fits into memory.

Second chance cyclic strategy is like FIFO except frames are skipped over that have been used since the last time that the frame was considered as a replacement frame. A reference bit is added to each frame: when accessed, its reference bit its set. When looking for a frame to replace, use the first frame whose reference bit is 0. If we consider a frame for replacement and skip it because it is 1, set its bit to 0. This is still simple, takes memory accesses into consideration (somewhat) but degenerates to FIFO so could still be bad.

Least recently used (LRU) approximates the optimal strategy; replace the frame not referenced for the longest period of time – assumes a page not referenced recently is unlikely to be referenced soon in the future (use the past to predict the future). Used very commonly. In practice, on each memory access we need to update the time of access for a corresponding frame; when a fault occurs, you need to search all page table entries for the oldest timestamp. This works well in practice but has the problem of tracking the use time.

Not recently used strategy combines the idea of second chance cyclic and LRU. LRU does not need the exact reference time, just a rough estimate of how old the access was. Second chance cyclic used a bit. The idea here is to age out accesses to pages by using multiple reference bits; when combined, the set of bits mimic a coarse timestamp for LRU. Note that, here, we still need to search for the smallest value when we need to do a replacement. Furthermore, bit-based operations are easier to manage by hardware.

NOTE The programming question in *Assignment 4* could have been managed by going cell-by-cell, row-by-row, managing threads by splitting sets into subsets, or going into several Tron-like spirals. Efficiency was best managed by changing the data structure for the pieces.

A **counting based** strategy bases the selection of a replacement page on how often it is referenced (not when it is referenced). So, for the question of which frame we replace, we can replace:

- the frame with the fewest references
  - assumes that the page is hardly used
  - penalizes new pages just entering the working set
- the frame with the most references
  - assumes lots of references means it is important
  - older pages can accumulate as time progresses and remain in memory even though they are not being used anymore

A way to fix this dilemma is to do some waiting: anything with less than a certain number of accesses get a free pass or anything that is extremely large could be replaced automatically (this would *amortize* the cost; that one access costed a lot but over a long period of time, this would be a greater payoff on average). Another way to fix this would involve *aging* access counts: anything that is really old will decrease its size geometrically.

File System Interface

Organization

A **file system** has many components: a field, record, file, directory, and partition: aggregating different sizes of information. A **field** is a basic element of data, a **record** is a collection of related fields, a **file** is a named collection of related information or logical records, a **directory** is a collection of files and their attributes, and a **partition** is a separation of a disk into logically independent spaces. Think of fields as analogous to a character in a String.

Rules could be enforced about what is placed in a directory, but this is not enforced by the operating system; this is merely a decision of the user. For example, the `bin` directory typically only contains executables.

File attributes are the first contact point in a file system: common attributes include name, size, location (device, place on the drive), its creator, access control, and file type. Depending on the file system being used, these can vary (e.g., files can be links, directories).

Organization continued on next page. . .

Modes of access can be **sequential** (continually move forward, e.g. tape drive), **direct or relative** (random access — needs some manner of addressing, e.g. disk), or **indexed** (random access and separate index file to find items quickly, e.g. database). This could influence hardware configurations or programming interfaces.

NOTE Cassette tapes are usually used for backups of systems; this is because they are infrequently used and when they are accessed, this is done sequentially (everything has to be accessed) anyway.

An **open file table** is a cached structure in memory of a process' access to files where information about the files are stored. This could be one large structure for an entire OS or separate tables. Your file pointer is typically some reference into the open file table; e.g., a table row number or a pointer to the table entry. The open file table contains information like the current logical location within the file, the mode of access, and location information for the device driver.

The organization of fields, records, and files was done in other courses. The focus here will be on choice of directories.

Directories

Directory structures could involve single level directories for all users (very basic functionality) — a **single shared directory**. This does not please the users very much (coexistence of files) but it is extremely simple to implement. This is actually used today: webpages are entirely under one top-level, shared, web directory.

A **two level directory** introduces an absolute path and search path: have a root directory and then a level where we can differentiate between system files and users. We as users only have a single directory, but we are not mixing them with other user files. Even if we have directories in our userspace, we would have to program our own shell to unpack those directories; the OS would not be able to manage those subdirectories. Note that the entire control of this structure is entirely with the OS; anyone but the OS would not be able to create a new subdirectory for some random user.

A **tree structure** allows any user to create a subdirectory. Each user can create directory structures of their own (control the directory structure) and poke at the data structures that are the file system and can create and destroy items. This means access control is needed for each particular user; also, relative paths become incredibly important here (above, we could just use the name of a user as an absolute path). These are convenient but still not as powerful as what we have.

NOTE Think: memory management structures and methods of addressing.

An **acyclic graph directory** is where multiple paths now exist to files with consequences on programs that traverse the file system. Directory links (hard and soft — symbolic, symlinks, shortcuts, ...) are introduced here. This is what we think about, but the actual structure is below.

A **general graph directory** is where directory cycles are now possible; even more consequences for programs that verse the graph structure, including the ability to isolate some files without deleting them. Garbage collection gets introduced (think of programming constructs: when no path is available, have to reclaim space). This can be a problem for depth-first searches, for instance, because of the potential for loops (even acyclic had problems of repetition and so needed better code to take that into account).

NOTE Files able to point to themselves is useful for addressing: `/local/bin/bin` vs. `local/bin` on path.

Directories continued on next page...

NOTE As part of the filesystem check process, could involve looking at what is no longer reclaimable but may not have completely been removed; put them in `/lost+found`.

A **per-application directory** is where each application now only sees its own subdirectory; the overarching file system, including the area for the OS, is not visible to users and applications (and applications cannot share files). This was previously available through `chroot` command (change a process' root) but here is made mandatory. This is the idea of **sandboxing** present in mobile devices. The root directory is still present but reserved to the OS; it might even exist only in memory.

Linking is where files can be identified as links in a directory to a file location on disk. A **soft link** or symbolic link is where the directory information stores the path to another directory entry that has the file location information (requires more accesses to get to the file; can span partitions). A **hard link** is where the directory information refers to the location on disk for the file (direct access to a natural address on the device; only allowed to link to the same partition, generally).

Every file has at least one hard link; a file is kept on disk as long as there is at least one hard link to it. Therefore, we must track the number of hard links to a file to know whether it can be deleted.

Other directory structures include **journaled file systems** (transactions, place them into a log), **versioning file systems** (source control built into the file system; a persistent file system — useful in critical file systems), **database file systems** (SQL queries, indices, etc.), **transactional file systems** (do not want transactions to be half-done; atomic, reliable environments) and **network file systems** (no longer on a small number of disks; goes across network devices — e.g., off in the cloud).

Partitions

Disk partitions are subdivisions of a disk into a logical storage unit; similar to static memory segmentation but for disks. **Physical partitions** and **logical partitions** can differentiate between two partition types. Physical partitions is an idea of carving out and isolation of a portion of a disk. Could be picked up by the hardware but limits the number we can have. On the other hand, logical partitions is done through the file system; consideration of segregation of units on a higher level.

Each partition has its own file system; different partitions on the same disk or computer can hold different file systems. Furthermore, each partition has its own root directory; the rest is up the file system to build. Each disk has one **primary partition**: the root for the disk; each bootable disk has a Master Boot Record (MBR) that tells an OS to find the starting block of code to load that will ultimately load the kernel into memory.

How do we access partitions? Present a single file system; create locations in one file system where we expect to transition into other file systems (called **mount points**). The root directory of partition X is seen as directory Y in the overall file system.

File Systems

It is not uncommon to use the file system to model other aspects of the operating system: lists of processes running, kernel information, semaphores, and named pipes. No system calls are needed to give this information to the user; it is available in a *read-only format* and therefore far less system calls have to be done. Different "device drivers" can go through part of the OS kernel and give the information that it provides at that **mount point**. It still runs under the same mechanism as the above.

File Systems continued on next page...

Furthermore, the file system can sometimes be used through the memory system: **memory mapped I/O**. Files can be accessed as if it was RAM or memory: even if a file is being used in a random-access manner, it does not have to be accessed serially (e.g., do not have to use various system calls to move through the file).

So, how does the system *start up*? We will come back to this.

A file system's implementation can be discussed by posing the question: ”how do we map the logical file structure into physical disk organization and structure?” In a drive, a platter is in contact with a read/write head present on a read arm. A physical disk is typically organized by dividing a disk into a set of **concentric tracks**. Each track is divided into equal-sized sectors (typically are 512 bytes, 1KB, 2KB, and 4KB).

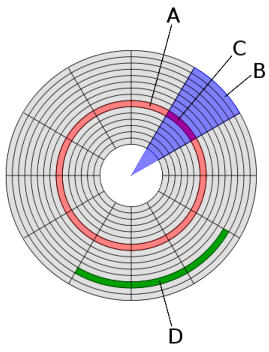

Figure 1: *A hypothetical disk where A is the track, B is a sector, C is a track sector, and D is a cluster.*

How much **internal fragmentation** are we willing to live with? Internal fragmentation is inevitable. Average file size could be a concern, and so can the size of the disk. As in the buddy system, this does not have a nice division. All the files on the disk could be tiny: small script files — therefore, we can use small blocks. Note that larger blocks mean we will have less I/O operations (moving the arm) but possibly a lot more internal fragmentation; the opposite is true for smaller sizes. With such a large pool of memory today, fewer I/O operations tends to be the choice.

Responsibility of mapping is delegated to different parts of the OS. A directory path to a file is understood by the logical file system. A simple logical file involves itself in the file organization module and free space management. A physical file address is available in a basic file system (interface into the file system; knowledge of the disk itself). A request to read/write a block is done by the I/O controller. Finally, driver commands are done by the device driver.

We have talked about the logical file system already. We are now moving on to file organization.

File Organization

Constraints on file organization include:

- seeking to a track on a disk can be slow,
- when reading a sector for a byte, it is just as efficient to read in the entire sector,
- files have arbitrary size (we generally append to files and run through them from front-to-back,
- and the assumption here is that all sectors are equal — in reality, some sectors are more equal than others.

File Organization continued on next page. . .

Ideally, memory should be contiguous: moving to the rest of a file means moving from one track to the next. Note that closer to the centre there is a smaller amount of space. Two different variations of disks come about: variable rate (slow down at the edge and speed up at the centre to keep a constant rate of bytes coming in) and constant rate.

Therefore, how would one *build a file organization module*? We have to determine where and how files are placed on disk, and manage the free space (know what is in every sector). Options include:

1. **contiguous storage** (not ideal for hard drives, but moreso for CDs, etc.),
2. **linked list storage** (e.g., FAT),
3. **indexed storage** (inodes; very commonly used), and
4. **combined index storage** (inodes; very commonly used).

Contiguous storage is where each file is stored in a contiguous sequence of sectors on one (or adjacent) tracks. Treat the disk space as a linear sequence of sectors and apply contiguous allocation schemes from memory management. Directory entries just store the starting track and sector of a file and the size or ending track and sector.

NOTE Think of a inward-directed spiral on the disk (but realize the head cannot move instantaneously). What happens is that there is a lag (*head reposition time*): have to establish a new point to reposition the head for the next track.

Advantages of contiguous storage is that it is good for sequential or random access to a file. Disadvantages are that we must predict the size of the file, accept external fragmentation, or be ready to copy files when they grow (leaves a hole; need to defragment it later). Examples are CDs or DVDs (generally read-only ones).

Linked list storage is a *non-contiguous storage of files.* It is a disk-based version of **paging**: the disk is divided into fixed-size sectors, the file is stored in a set of sectors, and we have a mapping from one sector to the next sector in the file. In a way, sectors are stored like a linked list with one sector pointing to the next one for the file. It is more efficient on the hard drive size if we can preplan by assuming fixed sizes of sectors; otherwise, bytes have to be indexed.

Where are the lists stored? This is more overhead to manage. Cannot be stored in memory: would disappear after rebooting. This has to be stored on the disk itself, then. The pointers could be stored in each sector (at the start, typically): this makes files hard to access randomly (have to traverse the list) and has low reliability if a sector of a file is corrupted.

The pointers could be similarly stored in the equivalent of an inverted page table. All the sequence of pointers can be gathered into one table (called the *File Allocation Table* where the OS caches the table in memory for fast access). The table can therefore be stored in a protected area of the disk and a duplicate or backup table can be made in case of corruption. This leads to the FAT file systems of the early DOS years. Typically held somewhere in the centre; always half a disk away from this table.

NOTE With copies, we have a problem of consistency. Could have a master table and opportunistically update shallow copies at different checkpoints on the disk. The larger the number in the FAT label refers to a larger number of bits allocated for the pointer going into the file system.

To find unused structures, one could traverse the table linearly (done in DOS), keep track of free sectors in another linked list, refer to a bit vector, or, less commonly, place pointers to all free blocks in any given free block.

File Organization continued on next page. . .

Flags can be present in this table for the end of a file, bad sectors, and unused sectors.

Advantages of this technique are that there are no external fragmentation and that it is good for sequential access. Disadvantages are that it is bad for random access, has low reliability if a sector fails, storing the entire FAT in memory is not ideal for today's storage, and that there is lots of space allocated to track unused structures.

NOTE Furthermore, think of formatting of the disk: is the table being removed ("fast format"), or are 0s being written to the entire drive? The same can be said of removing individual files. Note there are certain portions of the drive (bad blocks) that should not be touched; system files, etc.

Indexed storage is the idea of gathering all linked-list-style pointers for one file into an *index* for that file; these index files are stored in pre-defined structures known as **inodes** (sort of like having a single page table for each process). Thus, part of the process of formatting a disk is creating a set of inodes, distributed in a few locations on the disk (marking off tracks for this use). This is a limited resource on the disk and is pre-formatted; can have them all be contiguous and be numbered.

An inode can contain file size, device ID, user ID of owner, group ID of owner, file mode, time stamps, a link counter, and, most importantly, pointers to the file sectors/tracks (number of which depend on the file size). Notice that the filename is not placed in the inode. More than one hard links could point to the same inode; having information like the permissions, size, etc. means that the same file is being pointed to, but having different filenames mean it can be referred to by different names.

The inodes are a fixed number of pointers; *how are big files handled*? The last pointer of the inode could point to another inode (a linked list of inodes). Leaving header information empty would create internal fragmentation. Or a new definition of an inode can be used that uses this header space as more space for pointers. Finally, a multi-level index could be created where entries point to lists of pointers: handy for finding more random access to our files. The former is not great for random access. The latter still limits the file size (unless we plan for arbitrary levels) and can add more indirection than we want for small files.

NOTE What is the limit of a file size here? Given an inode with x pointers, an extension inode with y pointers, a block size of b bytes, a single inode would be limited to xb bytes, a 2-level inode could hold xyb bytes, a 3-level inode could hold xy^2b bytes, and so on. Note it is not entirely unreasonable to have a multiple file size for a file system; the more redirection is needed to fit very large files that may not be often accessed could become costly and inefficient.

NOTE This brings us about to an idea of **union data structures**. An infrequently used part of C is something known as a `union` (rather than `struct`) where the largest amount of space needed for any one variable is established but only one spot of memory is present for any of the variables.

A **combined index storage** system expands on indexed storage to accommodate both small *and* big files (see note box above). The first $n-2$ entries of an inode point directly to disk, entry $n-1$ points to a table of indices with one level of indirection, and n to one with two levels. Therefore: small files only need the first inode; large files can expand into more indrection. This is often place found in today's file systems.

Throughout all of these techniques, free sectors have to be stored somewhere. Alternatives for this include:

- bit vector (one bit for each sector),
- compressed representation (run-length encoding; think Huffman encoding),
- linked list: store in the FAT, and

- grouping: take one inode or one sector and fill it with references to unused sectors; chain the inodes or sectors into a linked list.

Note that in disks, we can have a similar situation as we have for processes: we can discuss scheduling. Practices evolve with file systems, but we typically choose nonpreemptive systems: random priority, FIFO, LIFO, shortest seek time first (could starve), scan (cscan, fscan; think of a bus — will move around and handle jobs as it hits a track where a request is had), look, ...

Furthermore, **non-local storage** includes **Network-Attached Storage** (NAS) where access to storage is through a regular network and **Storage Area Networks** (SAN) where a separate, private network between servers provides access to disk devices.

Redundant Array of Inexpensive Disks (RAID)

What happens when part of an inode fails, or when a sector fails? If no backups are being had, one could set up a disk organization that places data on multiple disks to provide *fault tolerance* and *parallelism.* Disk space for relatively unreliable drives is very inexpensive: pools of disks can be bought that could fail. Different levels of this system exist with different performance vs. cost cutoffs. Higher levels ensure greater protection in case of total disk failure.

NOTE We are not expected to know what each level does for evaluations.

Non-redundant striping means to take data and have file blocks divided among disks (level 0). This is also very fast (parallel). **Mirrored disks** means to automatically copy files to backup disks (level 1). **Memory-style error-correcting codes** means to have parity or error connecting bits stored on separate disks (level 2). **Bit-interleaved parity** is like level 2 but a combination of parity bits and sector checks allow for bit correction (level 3). **Block-interleaved parity** has correction on a block level (level 4). **Block-interleaved distributed parity** inter-disperses this correction amongst real data (level 5). Finally, **P and Q redundancy** is like level 5 but with redundant copies of the parity, error correcting data (level 6).

Access Control

Security

Confidentiality, **integrity**, and **availability** of resources are each a different sort of factor. The first involves having data and assets only accessible to authorized users. Integrity means they can only be changed by authorized users. Finally, availability means they are available to authorized users. The goal of **access control** is to protect against both accidental (easier to fix) and deliberate transgressions of users and processes. Furthermore, the principle of **least privilege** is attempted to be enforced here: only allow a level of access a user needs to do his or her work and nothing more (partly what a lot of users saw as a headache when they went into Windows Vista; XP was *default allow* and Vista was *default deny*).

Authentication is a principle that begs to be described here. This is usually some combination or subset of:

- what you know (password, challenge question; relies on you not divulging this information or be reasonably obscure),
- what you have (access card, RSA card), and

Security continued on next page...

- who you are (biometrics).

For example, debit cards require a PIN as well as the card itself.

Different threats exist here: an **interruption** (attacks availability), **modification** (attacking integrity), **intereption** (seeing what someone is doing; could replay or duplicate), and **fabrication** (makes a transaction on one's behalf).

Policy and **mechanism** need a distinction here. Policy is a statement on the way we want the system to work or behave. Mechanism are the techniques that we can apply in a system to enforce or exclude certain behaviour. *Policy is implemented through mechanisms*; we do not want mechanisms to dictate the policy.

Help can be acquired from the hardware: access to the hardware, though, makes attacks more effective. One could boot into a new OS (dangerous because could be on a network and be considered a trusted computer), enter "single-user" mode, trigger non-maskable interrupts, change chips, have direct access to bus information, and so on. There is a reason server rooms are kept secure. Physical security is not a concern of this course, and neither is social engineering.

Protection

It is important to talk about **protection domains** here. These are the set of objects available to a process together with the set of operations that the process may perform on the objects. **Access rights** are abilities to execute an operation on an object.

Access control matrices have two typical ways to store such a sparse matrix: focusing on compressing by columns (called an access control list) or by compressing by row (called a capability list; aka tickets). Alternative include **centralized storage** (lightweight directory access protocol; LDAP), **lock-key mechanisms**, and **role-based access control** (RBAC).

An **access control list** is when there is a list of all protection domains and their access rights for every given object. It is stored with the object being protected and is easy to add or remove permissions for any one object. Consequently, it is therefore hard to track down all of the permissions for one protection domain. The UNIX file system has coarse and fine ACL (see `getfactl` and `setfactl`).

Capability lists are when the list of all objects and access rights that belong to any given domain of protection are collected. it is stored with the user or domain of protection themselves; it is easy for a user to know what they can or cannot do. When a user wants to access an object, they present the list and an operation — the object checks that the list is valid and the list allows the operation on the object. These lists *must not be forgeable* and one must be careful about who can issue these lists. Also, it should be determined whether or not the lists should be transferable; how do you prevent a transfer?

And how do we control how access rights change? Each domain of protection is treated like another object to protect; new operations can be included to managae domains (owner, control, transfer, copy, limited copy, switch, ...).

Access right issues include: **revocation** (removing an access right; depends on mechanism used to convey rights), **limited propagation** (how far can a permission go to one domain), and **confinement** (ensure that data obtained in one domain does not leak into another).

Review: Making a Debugger

What is involved in making a **debugger**? Consider that you have an executable you wish to debug, and then a program which is known as a debugger. There are two heavyweight processes that are theoretically involved here: a debugger and the executable.

Note that, here, we must have separate memory spaces. The memory space within the executable has to be get at. Furthermore, they have to be able to communicate: commands, the variables and what are in memory, state (registers), and flow of control (which will involve interrupts as a mechanism).

The executable here is a child to the parent who forked that process. Being a thread here would not be very advantageous if there are going to be crashes in the executable (it will not affect the debugger). Commands and, furthermore, signal calls and interrupts must be sent to the child. Default interrupt handlers can also be created here.

Now let us take this further; listing variables, etc. must be acquired from the source files (which are on the file system). Note that the executable file is also present there. System calls can be made to get the **symbol table**: functions, variable names (and where they are located in the stack), etc. That metadata can be acquired and information can be obtained about the lines.

NOTE Recall that interrupts, system calls, and traps differ. Interrupts come from the hardware, system calls is software making a request from the operating system, and traps which allows users to issue a software command from user level mode to spring an interrupt (controlled reclamation of control by the operating system).

Anything that is needed to know is present in the **process control block** (such as state, instruction register). Getting access to this block requires **root access** to run; debuggers these days do not necessarily require that but are more complicated.

What have we managed to get done so far?

1. start and stop the executable,
2. list where we are,
3. print registers,
4. print variables (mapped to registers – easy, on the stack, in the heap; these latter ones are caught in the memory structure – segmentation vs. paging because segmentation is individual to data structures; access segment table),
5. (conditional) breakpoints (see below),
6. execute a different function, and
7. debugging at a crash.

Note that if we are using **segmentation**, both processes (e.g., the operating systems) are using this memory architecture. Our basic notion of heavyweight resources are that they possess different systems of resources. The rules can be bent slightly here: parts of the segment table of the debugger can be mapped to parts of the segment table of the executable and looked at directly.

This brings us to **address binding** again. Different states are present for when values are assigned to physical addresses. Logical addresses in memory space here are broken into two part: **segment ID** and **offset**. The consequence here is that memory can also be changed here. The same can be said of registers (e.g., even the **instruction register**).

Review: Making a Debugger continued on next page. . .

Breakpoints are triggered by **traps**. There are more elegant ways of doing this today, but what can be done is to interrupt the parent at a line number corresponding to this breakpoint by a trap call. This will also pause the child (causing an interrupt to happen and go into an interrupt handler). How do we handle this and make the executable do this?

A ”slimy” way of doing this is rewriting the code at those lines through the symbol table to call the trap. Another way of doing this is to give space to do this when compiling the code (place garbage instructions at each line). No-ops will not slow the execution very much and can be replaced with trap calls; the memory address to return to is on the stack when in the interrupt handler.

What about **conditional breakpoints**? These conditionals can be done in the debugger before returning to the user/executable.

Furthermore, sometimes executing different functions can be useful. An example of this is to execute functions that print out data structures. One could change the instruction register or even create a new stack frame (put parameters on the stack, etc.). The return address would be another trap to return to the debugger.

When a crash occurs, the process goes into the exit state (because it is done) so it can be cleaned up. Because there is a debugger attached, the debugger can delay this process and poke around; if it can be fixed, it can be put back into the ready state. If the operating system is helping catch memory leaks, the exit state is also a place where this can be done.