CS162 Midterm 2 Study Guide

Advanced Synchronization

How is a Condition Variable different from a semaphore?

A Condition Variable is stateless, whereas a semaphore contains a value.
Both can be used to signal a thread as a result of an action by another thread, but the semaphore can also be used for mutual exclusion.
A Conditional Variable can only be used in conjunction with a lock. Operations on the CV should only be performed when the thread is holding the lock. The lock provides mutual exclusion.
A semaphore is a more primitive, and hence more powerful and less structured, concept.

What operations are defined on Condition Variable? When can they be invoked?

Wait atomically releases the lock, relinquishes the processor, and reacquires the lock before the thread is rescheduled. Typically, the resumption from wait will be the result of a signal or broadcast, but need not be.
Signal wakes up one waiter (if any); Broadcast wakes up all.

What must the kernel do to make `cond_wait` appear to the user thread as having atomically released the lock and reacquired it upon signal?

Release the lock, block the thread, schedule other threads, and reacquire the lock before allowing the original thread to be ready for scheduling.

Why should a user thread that waits on a CV use it in a loop that checks the signal condition after resuming from cond_wait?

It may be a spurious wake up, perhaps due to broadcast or simply because the kernel decided to resume the thread.

What are the shortcomings of using interrupt disable/enable to implement locks (and semaphores)?

It disables all interrupt processing, even though the lock may have nothing to do with handling interrupts, which may cause interrupt handling to be delayed or missed entirely.
It is not available at user level, so all thread synchronization requires a full round-trip through the kernel.
It is not sufficient for multiprocessors, since another processor may access memory state associated with the lock.

What are examples of atomic read-modify-write instructions?

Test-and-set, swap, compare-and-swap. (Others we didn't discuss include fetch-and-increment and a special pair load-locked / store-conditional.)

How can atomic read-modify-write instructions be used to provide a faster implementation of locks that also works on multiprocessors.

A variable, i.e., value held in memory, is used as a guard for manipulating the lock. Test-and-set on the guard provides mutually exclusive access to the state of the lock – while contenders busy-wait on the guard. So the guard protects the lock, whereas the lock protects an arbitrary object that is manipulated by one or more threads. Access to the guard can be performed at user level, so an entry into the kernel is only necessary if the lock is busy and, hence, the acquiring thread needs to be put to sleep until the lock is released.

Virtual Addressing, Address Translation, Virtual Memory

What is the relationship between a virtual address space and a physical address space?

A thread operates in a virtual address space that contains all of its logical state – its code and data, whether that data be static, stack, or heap based – independent of where that state is held in the physical storage hierarchy. A machine provides a physical address space that contains regions of physical memory and regions of device state. An address translation mechanism translates the per-process virtual addresses referenced by threads to physical addresses in the machine in a manner that corresponds to how process state is resident in the storage hierarchy.

How are the following operating system functions implemented all or in part through the address translation mechanism?

- Protection
- Multiprogramming
- Isolation
- Memory Resource Management
- I/O efficiency
- Sharing
- Inter-process communication
- Debugging
- Demand Paging

Protection: a thread can only reference those resources that are made accessible to it through its virtual address space. In particular, it cannot in general access the state of other processes, the state of the kernel, or the state of physical devices. Exceptions to each of these restrictions are possible in a controlled fashion.

Multiprogramming: Only the active portion of each process' state (it's current working set) need to be resident in physical memory; the rest can be held in secondary storage. Upon access to a non-resident page, it can be made resident and mapped into the process virtual address space by updating the page table. The resources associated with a process or program can be switched rapidly by simply changing which page table is active.

Isolation: The protection mechanisms described above isolate processes from each other and the kernel from user processes by limiting what each can access.

Memory Resource Management: Physical memory is easily allocated because it can be treated as a collection of fixed sized blocks (i.e., frames) and any page from any process can be placed in any frame.

I/O efficiency: While I/O is being performed on behalf of one thread, other threads can be easily run. In addition, access to I/O buffers can be transferred between user and kernel by protecting or remapping pages, rather than copying data.

Sharing: processes can share storage resources by mapping the physical resource, i.e., the memory region, into their respective virtual address spaces. Similarly, kernel and user can share in a protected fashion by selectively controlling access permissions. For example, a I/O buffer may be read-only to the user and read-write to the system.

Interprocess communication: processes can communicate with one another through message exchanges, e.g., sockets or RPC, or through shared address space as described above. Even if they are using message-based communication the buffers underlying it may be implemented using shared memory. Or processes may communicate through writing and reading files, and files may be mmap'd into the virtual address space.

Debugging: A debugging process can be used to debug another process by "attaching" to that processes address space.

Demand Paging: When a thread accesses a non-resident page in its process' virtual address space a page-fault occurs, suspending the current instruction and allowing the operating system page fault handler to load the corresponding page from secondary storage to physical memory and map it into the process address space before retrying the instruction.

How does a virtual address space eliminate the need for relocation upon loading an executable program?

The process can execute in a virtual address space as if its addressing starts at zero, thus the references within the code and data sections of the file can be used directly without any offset adjustments.

What does an operating system need to do to "exec" a program in a VAS?

Create a process for it, i.e., allocate and initialize a data structure (such as a Process Control Block) to hold process-specific information, create and initialize a page table (PT) for it, record the PTs location in the PCB, initialize file descriptors, set up code and static data regions of the VAS as backed by executable file, create an initial set of registers and transfer control to it. The stack and heap regions may be set up as well, or the language run time library may set those when the program first executes, i.e., before calling main.

How is paging different from segmentation?

Paging breaks up the virtual address space into fixed size blocks, regardless of logical structure. Segmentation divides the address space into variable sized extents that correspond to logical units in the program.

What is fragmentation?

Fragmentation is when unusable regions of storage emerge because of how resources are allocated. Iterative allocation and de-allocation of variable length blocks, such as segments in memory, can give rise to *external fragmentation*. There may be many small, unused chunks of storage between used ones, and the aggregate unused space may be enough to service further requests, but no single block is big enough to service the requests. Fixed size allocation, such as paging, eliminates external fragmentation but introduces *internal fragmentation*, because requests must be rounded up to an integral multiple of the page size.

How is an N-bit virtual address translated into a physical address using a one level page table for pages of 2^K bytes?

The upper N-K bits form the page number (VPN) within the virtual address space, while the lower K bits form the offset within a page. The N-K bit VPN is used as an index into an array, the page table, to access entry PT[VPN]. The page table entry contains flag bits that indicate if the corresponding page is resident in memory and if so, its frame number. Thus,

```
vpn = VA >> k
off = VA & (2^k – 1)
if PT[vpn].valid
     PAS = PT[vpn].frame << k + off
else
     Page_fault
```

How are program segments typically represented in a large, flat virtual address space?

Dynamic segments, such as stack, heap, and dynamically linked library code, are allocated regions in the address space with ample room between them to allow them to grow. The operating system may recognize that certain regions should grow automatically, such as the stack, while others should require an explicit request to grow, e.g., sbrk to expand the heap.

How large is a single level page table or a N-bit address space with pages of 2^K bytes and an M-bit physical memory space, assuming that all of the VAS is represented? For concreteness, what does this mean for a 32-bit address, 8 KB pages, and 32 MB of memory? For a 48-bit address, 8 KB pages, and 1 GB of memory?

Each entry must be big enough to hold a M-K bit frame number field. This requires 32-13 = 19 bits for the small case and 48-13 = 35 bits in the large case. Thus, in the small case it fits easily in a 32-bit word, whereas in the large one it does not. However, such a large address space is almost always found on machines with 64-bit words.

The page table has 2^{N-K} entries of at least M-K bits each. This means 512K entries or 2 MB in the small case and 32G entries or 256 GB (with 8B entries) in the large case. And this is per process. Thus, the address translation mechanism would occupy an excessive amount of physical memory.

Consider a 48-bit virtual address translated into a physical address using a three level page table for pages of 2^K bytes with page table entries 8 bytes in size. What would be a natural page size for such a system? What is the memory footprint for the translation to map four dense regions of, say, 100 MBs each, i.e., segments?

A page-size chunk of page table entries contains 2^{K-3} entries. For 8KB pages and 8 byte entries, this would be 1024 entries per chunk. The offset portion utilizes the lower K bits of the address, thus we have 48-K bits to translate. Each level naturally takes care of K-3 bits. So we need $3(K-3) + K \sim 48$. So a natural choice of K would be 14 or 15.

At K=14, we have 16 KB pages and 2K PTEs per page sized chunks (11 bits). Working from the bottom up, each level 3 PTE chunk maps 32 MBs (25 bits), each level 2 chunk maps up to 32 GB (36 bits), and two level 1 pages are needed to map the top 12 bits.

Each region would require four level 3 PTE chunks, each of these will require one level 2 chunk, and there are two level 1 chunks, so a total of 18 pages x 16 KB, so less than half a MB to map these 400 MBs of data pages.

What would the TLB hold in such a situation as the one above?

The TLB caches overall translations for pages, crossing over multiple levels of page table translations. Thus, each TLB entry would hold the 48-14 = 34 bit vpn as a tag and the physical frame number at which that page is resident, as well as tag bits.

How many page faults might occur in accessing an address in the worst case? In a typical case?

Three. The root PTE pages must always be resident. However, the level 2 PT may be paged, so a page of PTEs might need to be made resident. The level 3 PT page it references may need to be made resident. And then the actual page of process address space may need to be paged in.

On the other hand, if we consider the four large regions scenario described above, a region has to grow substantially before an additional level 3 PTE page will need to be paged in and a new level 2 would only be needed if a new region started to be used. Thus, in practice it is rare that anything more than the actual user page will need to be paged in.

Using paging, what is the unit of storage that can be shared between processes? What about with segmentation? What needs to happen if some portion of the shared region was swapped out to secondary storage?

With paging, an integral number of pages must be shared. So the smallest unit of sharing is a page. With segmentation, a variable length extent can be shared.

When using paging to support sharing, if a shared page is swapped out all page tables that reference the page need to be updated and any TLB entries that refer to it must be invalidated. With segmentation, the entire segment must be swapped and the segment tables of all processes that share the segment must be updated.

Why is it important to have a high degree of associativity in the TLB?

Each TLB entry maps an entire page. A single instruction will typically refer to at least two pages, the one holding the instruction itself and, if the instruction does a load or store, the one holding the data. More complex instructions, such as those in the x86 architecture, may access multiple memory locations.

When does the operating system need to flush or invalidate TLB entries?

Unlike memory caches, TLB consistency is not typically handled in hardware. Thus, whenever the OS switches processes or updates a PTE, it may need to invalidate entries in the TLB. In some machines, invalidation on process switch is avoided by including the process identifier (PID) in the TLB entry? This does raise a complication that if two processes share a page, it will appear in multiple entries in the TLB.

What happens on a TLB miss? With a two-level page table, how many memory accesses are required? What should we expect the TLB miss time to be compared to a memory access time?

On a TLB miss, the memory management unit must traverse the address translation structure to obtain the physical address and enter the VPN->PFN (physical frame number) translation. On a two-level page table, this involves two memory accesses (and possibly one page fault) to access the two relevant PTEs. However, both PTEs are likely to be cached (and resident) so the TLB miss penalty can be much smaller than two memory accesses.

What is the difference between a memory cache and a TLB?

A cache transparently holds the data resident in particular physical memory addresses. Portions of the physical address form the cache tag. Misses are handled transparently by the cache controller. The cache is transparent to operating system and user software, some memory references just take less time than other. The TLB is a specialized cache for page table entries. It is updated upon virtual to physical address translation when the memory management unit accesses page tables. It is transparent to user software, but requires management by OS software.

What is the average access time for any level of the storage hierarchy?

Ave Access Time = Hit Time + Miss Penalty * (1 – Hit Rate)

What are the degrees of freedom in cache design?

Cache size, Block Size, Associativity, Write Policy, Replacement Policy

How is demand-paged virtual memory similar to and different from caching?

Both provide automatic management of the storage hierarchy taking advantage on temporal and spatial locality. Caches are handled entirely in hardware, transparent to software, and have a fixed organization. Virtual memory involves a collaboration of hardware and software, where substantial operating system software is require to handle a miss, i.e., a page fault. For a memory cache, the miss penalty may be 10 to 100 times the hit time, whereas in virtual memory management it is on the order of a million times the hit time. Paging treats memory essentially as a fully associative cache for the virtual address space that is backed by disk.

Where does caching arise in operating systems?

- Operating system data structures can be designed to improve memory cache performance (or not).

- Thread scheduling can impact memory cache performance. Short time quanta and frequent switching can reduce cache performance.
- TLBs are a cache of page tables.
- File systems cache disk blocks, including blocks containing part of the file system structure.
- Virtual memory caches pages of the virtual address space in physical memory.
- Domain name system is a cache of hostname to IP address translations. And there are many more.

When a page fault handler needs to evict a page from memory, how does it find the corresponding page on disk to swap it into?

Typically, the OS maintains a table with an entry for every memory frame that records the VPN for the page that is resident in the frame. Otherwise, it would have to search the entire page table to find the page that occupies the frame.

When a page fault handler is swapping in a page, how does it pick of memory frame?

It maintains a freel-list of unused frames. This might be a bit vector of frames or it might use the frame memory to hold the list. It seeks to push unused pages to disk so that there will be some free frames should it need one.

What actions and precautions must a system call take when a user virtual address is passed as an argument to the call, e.g., a pointer to a data buffer?

First it must translate the user virtual address to something that the OS can access. The OS typically does not operate through the user page table. It may even operate in a "physical address mode." Thus, it may need to consult the user page table explicitly to locate the desired physical address. In addition, it cannot trust the user pointer. The errant or malicious user may attempt to pass a pointer that would be invalid for it to access, but which the OS could access, such as a pointer into kernel memory. Or, it may try to crash the OS by passing a bad pointer. So the syscall must safety check all user pointers.

File Systems

What are the stages in operating system processing of a file read or write system call?

- Dispatch to the filesystem syscall handler
- Validate the user syscall args (handle, buffer, length)
- Consult the file descriptor for the handle passed to the call to obtain file number (or inode number) and position.

- Consult the file index structure to determine the driver and disk blocks for the I/O transfer
- Invoke the device driver to perform the transfer.
- Driver issues commands to the I/O controller for the disk, SSD, or other physical device containing the block.
- It issues commands to transfer data between memory and the controller.
- Detect completion of the I/O transfer and complete the file system operation.
- Complete the syscall.

What are the physical components of the I/O system?

- A wide range of physical devices, including storage, displays, networks, audio, keys, pointers, printers, etc.
- A hierarchy of busses: memory bus => high speed interconnection bus (e.g., PCI express) => lower speed I/O busses (SCSI, IDE, Serial, USB, …). The bandwidths of these vary by over 12 orders of magnitude.
- I/O controllers associated with busses and with devices.

How does operating system software on the processor interact with I/O controllers?

- Memory mapped I/O (reading and writing controller registers)
- Direct Memory Access (block transfer between memory and controller)
- Interrupts (asynchronous notification from the controller)

What determines the bandwidth achieved on an I/O operation?

The peak data transfer is determined by the transfer rate of the bus, the controller, or the device – whichever is least forms the bottleneck.
The effective bandwidth is determined both by the peak transfer bandwidth and the delays associated with initiating and completing the transfer, i.e., the startup costs. These are often fixed costs, independent of the transfer length.
Thus, the time to perform a I/O operation of size n is of the form $T(n) = S + n/B$, where S is the startup cost and B is the bandwidth.
The effective bandwidth is $BW(n) = n/T(n) = n / (S+n/B) = B * (1/(1+ SB/n))$.

What are figures of merit for the performance of I/O operations?

Response time (latency) and throughput (bandwidth)

What software factors can impact I/O performance?

- Bursts of requests may create queuing delays, further increasing the startup time and decreasing effective bandwidth.

- The sequence of requests may be well-suited to the underlying device (e.g., sequential reads of disk blocks) or not.

- If there are multiple outstanding requests, driver software may be able to re-order and schedule operations to make them better suited to the device, e.g., C-SCAN disk transfer scheduling.

If a system processes 1,000 tasks per second and on average tasks take 2.5 seconds, how many tasks are expected to be on-going within the system at any time?

2,500. Little's law. And "the system" could be a web server, a network, a cafeteria or just about anything.

How does the RPM of a disk influence its performance? What performance factors are independent of RPM?

The *transfer rate* is determined by the velocity of the surface under the head x the recording density of the information on the surface. Of course, this is not uniform since the outer parts of the disk travel faster.

The rotational latency is the time for the desired sector to come under the head. On average, this would be half a rotation. For example, 7200 RPM is 8.3 ms per revolution. 10000 RPM drops this to 6 ms.

The other factor is the time to position the head to the cylinder containing the desired sector.

What are the benefits and drawbacks of the following disk scheduling algorithms: FIFO, Shortest-seek-first, SCAN, C-SCAN?

FIFO may experience unnecessary seek delays, but respects the request order and thus provides stronger guarantees of what happens before what?

Shortest seek time first is a simple greedy schedule, but does not respect order and is not optimal and can lead to starvation.

SCAN and C-SCAN are extremely simple scheduling algorithms that retain fairness. They do not respect issue order. C-Span, with it unidirectional service, tends to accumulate clusters of transfers in areas of the disks while servicing other.

When are the effects of sophisticated scheduling algorithms most pronounced?

At intermediate to high load levels. At low load there will seldom be outstanding request in the queue, so little opportunity for optimization. It reduces to FIFO. At very high levels, there will be such a large backlog that delays will be high regardless of level, but seek avoidance will have dramatic effects and fairness most important. This is where C-SCAN provides both performance and resilience.

What are the major components of a file system?

- User level I/O library that issues system calls and provides the API.
- System level file descriptors that maintain information about open files for the process.
- Directory structure that is used to translate a file pathname to a file number (or inode number) and to provide access control.
- File index structure that is used to translate file number plus position into disk block.
- Block storage which holds file data blocks, the file index structure, and directory structures (in special files).
- Free list, which records which blocks in a volume are used or free.

Most operating systems separate open/close file operations from read/write operations. What happens on open in Unix-like systems?

- The directory structure is consulted to obtain the inode number, if the exists, and validate user permissions to access the file. If not, a directory entry and an inode is created.
- A file descriptor is created and associated with the process, which contains either the inode index or a copy of the inode itself.
- A handle on this file descriptor is returned to the user.

How does the inode structure used in FFS reflect the observed properties of how applications use files?

- The vast majority of files are small, but most of the storage space is occupied by very large files.
- Inodes have several (12) pointers direct to disk blocks for the common case of small files.
- For larger files, it has a indirect pointer to a block of pointers to disk block, a double indirect pointer, and a triple indirect pointer.

In what ways does NTFS go even further in optimizing the observed properties of files?
The data for very small files can be stored in the index entry itself, i.e., in the Master File Table.

How can a file system design enable efficient use of the underlying disk hardware?

- Allocate data blocks such that if the file grows it will tend to be laid out sequentially on disk, rather than being scattered around. This may involve leaving free space at the end of the file for it to grow into, or by copying the file from time to time into contiguous groups of blocks.

- Co-locate parts of the file system that might be accessed closely together in time. Rather than separating the directory, index, blocks and freelist into different regions, group parts of them together into block groups.

- Co-locate files that are in the same directory and try to keep them close to the directory entries as well.

How might these change for SSD storage?
On an SSD it is no longer as important to co-locate objects to reduce seeks and rotational latency, but it is important to be able to group transfers into large contiguous blocks, especially for writes, and to avoid wearing out particular regions, writes should be spread over the entire storage extent. Thus, copy-on-write file systems become especially attractive.

When can the inode and storage blocks of a file be reclaimed?

After the last directory entry, i.e., hard link, for the file has been deleted.

How is the file protection model different in FAT, FFS, and NTFS?

FAT has no protection. FFS recognizes R|W|X permissions or each is owner, group, and other. NTFS has access control lists that can specify permissions on a per-user basis.

What is a journaling file system?

One that uses transactions to provide ACID properties on internal operations, such as updating the directory, index, and free list consistently.

What is the basic structure of a transaction?

- Operations are performed to prepare for a transactional update to a data structure, which may require obtaining multiple locks, reading data objects, and computing updates that will be made to the structures.
- A transaction is opened, obtaining a transaction id.
- A sequence of idempotent updates are recorded into a non-volatile log.
- The transaction is committed by performing an atomic operation on the log.
- Locks are released and future operations treat the log as taking precedence over the state of the structures that are modified by the transaction.
- In the background, the log is "redone" onto the permanent structure and completed transactions are garbage collected.
- If a crash occurs, updates to the structure are replayed from committed transactions in the log. Uncommitted transactions are discarded.

How does copy-on-write help support transactions for journaling?

Instead of updating in place, a new version of the file is created with a new index, new indirection blocks, and new data blocks. Making the new version THE version, essentially the rename, can be the atomic operation.