**Exam 1**

1. False.
   1. Int \*Foo is a pointer to an int. int \*\*foo is a pointer to a pointer to an int.
2. False.
   1. Bar is an int. also, bar is uninitialized so its value cannot be printed. We are trying to dereference bar with \*bar, but bar is not a pointer and thus cannot be dereferenced. If bar was a pointer that stored the address of foo, then %d and dereferencing it would print the value, not the address, as dereferencing gives the value that is pointed to by the address. To print the address of foo, you would not want to dereference bar.
   2. Int foo = 0; // an int

Int \*bar = &foo; // a pointer storing the address of an int

Printf(“%p, %d”, bar, \*bar); // prints the address of foo (value of bar) and value of foo

1. False.
   1. Referencing array elements in succession is **spatial locality**, since consecutive array elements are stored close together in memory.
   2. **Temporal locality** is more like accessing the same element or set of elements in a short time period, regardless of their position in memory.
2. True.
   1. **Local linker symbols** are used within an object file or exe that are not visible outside of that file. They are used by the linker when combining multiple object files into a single exe or library but are not exported for use by other object files or exe’s. these symbols can represent functions, variables, or any other identifiers that are only needed internally within the scope of the object file or executable.
   2. local linker symbols are not program variables. **Local program variables** are variables defined in a function or block scope within a program. They are accessible only within the block or function in which they are defined and have a lifetime that is limited to the execution of that block or function.
   3. Local linker symbols can be static functions or static global variables etc, which are intended to only be used within the file or library in which they are defined.
3. Note: I assumed that dst[j][i] = src[i][j] accesses src then dst. But idk why it would access dst at all. It writes back to the cache then uses the dirty bit. Once it gets replaced it writes back to main memory (replacement policy).
   1. **Direct Mapped Cache**: each block of main memory maps to one time slot (line) in the cache. The mapping is determined by some bits in the memory address. This means if two blocks of memory map to the same cache line, one will replace the other if it gets loaded into the cache.
      1. Advantages: simple, fast access time since only one location needs to be checked to find a match.
      2. Disadvantages: can suffer from high conflict misses, especially if multiple frequently accessed memory blocks map to the same cache line.
   2. **Set Associative Cache**: compromise between direct mapped and fully associative. The cache is divided into several sets, and each block of memory can map to any line within a set. The number of lines in each set is the **associativity** of the cache. A 2-way set associative cache (E = 2) contains 2 lines per set.
      1. advantages: reduces the problem of conflict misses found in direct mapped caches since a memory block can be placed in any one of the lines in its set.
      2. Disadvantages: requires more hardware for the implementation of set selection and replacement policies. Access time is slower since multiple lines must be checked.
   3. **Fully Associative Cache**: any block of memory can be placed in any line of the cache. This allows the cache to use its entire capacity to store any data, minimizing conflict misses.
      1. Advantages: eliminate conflict misses entirely since any memory block can go into any cache line.
      2. Complex and expensive to implement. Access time is slow since the cache must check every line for a match.
4. .
   1. .
      1. A[0][0] = 0 = 00 00000000, a[1][0] = 0 + 128\*4 = 512 = 01 00000000.
      2. The array has 256 integers. The cache stores 512/4 = 128 integers, so half the array. It can be seen that array elements that are exactly half the array (128 ints) apart will map to the same spot in the cache.
      3. So each access to A[1][i] will conflict with the previous access to A[0][i], so the set will be overwritten back and forth each time. Since there are 4 ints per block and 2 ints in the array map to the same place, we have 8 ints mapping to the same set thus we have 7 conflict misses and 1 cold miss per set, totaling (7+1)\*32 = 256 total misses / 256 accesses = 100% miss rate
   2. .
      1. We double the cache size to 1024 bits = 1024/4 ints = 256 ints = the total size of the array. This means the entire array fits in the cache, and each int maps to its own unique spot.
      2. Now the second row of the array will map to the second half of the cache, leaving the first half alone. There will be 0 conflict misses.
      3. The only misses will be cold misses. Since there are 4 ints per block, only ¼ ints will be a cold miss as we populate each block = 25% total miss rate.
5. ELF file format:
   1. .text (executable code (machine code)) – the machine code of the compiled program.
   2. .rodata (read only data) – read-only data such as the format strings in printf and jump tables for switch statements.
   3. .data (initialized data(global and static variables)) – initialized global and static C variables. Local C variables are maintained at runtime on the stack and do not appear here.
   4. .bss (uninitialized data (global and static variables)) – uninitialized (or initialized to 0) global and static C variables. They do not occupy any actual disk space.
   5. .symtab (symbol table) – a symbol table with info about functions and global variables that are defined and referenced in the program.
   6. .rel.text – a list of locations in the .text section that will need to be modified when the linker combines this object file with others. In general. These are instructions that call an external function or reference a global variable.
   7. .rel.data – relocation information for any global variables that are referenced or defined by the module.
   8. .debug – a debugging symbol table with entries for local variables and typedefs defined in the program, global variables defined and referenced in the program, and the original C source file. This is only present if the -g flag is used.
   9. .line – a napping between line numbers in the original C source program and machine code instructions in the .text section. Only present if the -g flag is used.
   10. .strtab (string table) – a string table for the symbol tables in the .symtab and .debug sections and for the section names in the section headers.
6. .
   1. 4 sets, so log2(4) = 2 bits for s (CI)
   2. 4 bytes per block, so log2(4) = 2 bits for offset (CO)
   3. 12 bit memory addresses so 12 – 2 – 2 = 8 bits for tag (CT) (there are 2^8 = 265 possible tags)
   4. 1 digit in hexadecimal = 4 digits in binary.
   5. So tag is the first 2 digits in the hexa address, and the last digit converted to binary gives the CO and CI.
   6. Go to the set, then search the line with the correct tag. If found, if valid = 1 it’s a hit, if valid = 0 it’s a cold miss so load it in.
   7. We don’t actually know the values we are loading in (the values in the table are the values of 1-byte (8 bit) words in hexadecimal form), we just know that 4 contiguous addresses will be loaded into the same line. So 834, 835, 836, 837 will all be loaded into the same line in set 1. 830-833 would map to set 0, etc.
7. Note: assuming the blank after % in the print statement is supposed to say %x.
   1. 2 strong globals in foo.c. ints so 4 bytes apart
   2. Double x in bar.c (8 bytes)
   3. X comes after y in foo.c, so y is not overwritten
   4. X is set to -0.0, so im guessing it squeezes a 0 into the 4 bytes for int x???
   5. The command allows compilation without warning
   6. %x prints a value in hexadecimal
   7. So y is still 15212, x is 0. Output is that but in hexadecimal
8. Array should be accessed in A[0][0][0], A[0][0][1], … A[0][1][0], … A[1][0][0], …. So we need for (i) for (j) for (k) { A[i][j][k] }. (or leave A[j][k][i] but change loop order to for (j) for (k) for (i).)
9. Given array[ROWS][COLS] we want to access all cols in the same row before changing rows. So we want to access A[0][0], A[0][1], … so that we traverse linearly across the contiguous memory allocated to the 2D array, rather than skipping around between blocks of memory that are given to each row. This nested loop accomplishes that with the outside loop controlling the row index, giving it good spatial locality. This is known as **stride-1 reference pattern**. It also has good temporal locality because we are repeatedly accessing sum in each iteration.

**Exam 2**

1. The **multi level feedback queue (MLFQ)** scheduler is designed to optimize various aspects of computing workloads by adjusting to the behavior of the system and its processes.
   1. A. MLFQ learns things about running jobs. It adjusts the priority of jobs based on their observed behavior, such as whether they are using I/O operations.
   2. B. MLFQ does not starve long running jobs. It does prioritize shorter running jobs, but after a certain amount of time, long jobs that have been at the bottom of the queue get cycled back up to the top, giving them opportunity to run for a bit at given intervals, so they will eventually finish.
   3. A. MLFQ assigns different length time slices to jobs at different priority levels. Higher priority jobs get shorter time slices which helps with response time and lower priority jobs get longer time slices.
   4. A. within each queue of the MLFQ, a round-robin scheduling algorithm is often used to ensure that all jobs at the same priority level receive equal CPU time.
   5. A. MLFQ may reset the priorities of jobs under certain conditions to prevent a job from gaming the system. This also helps with jobs that change their behavior over time, like a CPU bound job that starts making I/O requests.
2. C. A runs for 100ms, B runs for 100ms then finishes, A funs for 400ms and finishes.
   1. A runs for 100ms until B arrives. At this point A has 400ms remaining and B only has 100ms, so B takes over. Then B finishes and A resumes.
3. SJF and STCF starve longer running jobs, miss out on CPU when the current process is waiting on an I/O request, and potentially have worse average response time (long job may sit for awhile before first response, whereas in MLFQ all jobs get a response shortly after first arrival.
   1. A,D just A. idk why its not d
4. Jobs are taking turns running for a set time interval. This looks like Round Robin.
   1. **First in, First out (FIFO)**: new jobs are placed at the back of the line. The current job runs to completion, then the next job in the queue starts.
      1. B. FIFO would run each job to completion.
   2. **Round Robin (RR)**: new jobs are placed into a queue. Run the current job for a specific time slice then move to the next job in the queue. Repeat until all jobs are completed. If the RR is **preemptive**, it means the scheduler has the ability to interrupt a running process and reassign the CPU to a higher priority process.
      1. A. switch to next job in RR order every 10 time units.
   3. **Shortest Time to Completion First (STCF, not SJF)**: run whatever job is waiting that has the shortest total run time. Run the job from start to finish, or until a shorter job arrives. If a shorter job arrives, pause the current job to run the shorter job.
      1. B. this one could have worked, but when C finishes, it would have gone back to B, since B only has 5s remaining and A has 10.
   4. **Multi-Level Feedback Queue (MLFQ)**: multiple levels of Round Robin, learns from the past, incorporates I/O to run other jobs when the current job is waiting on an I/O. To prevent gaming, each job gets a total amount of time before dropping a level, so if you don’t finish your time slot, your next time slot will start with how much of your allowance is left. Lower priority levels get longer time slices. After some time, reset all jobs to the top queue.
      1. A. This one could work for this example bc lets say A,B,C arrive. A runs for its time slice, then drops a level. B runs for its time slice, then drops a level. C runs then finishes. Then the top level is empty, so check the second level. A runs for its time slice and finishes, then B runs for its time slice and finishes.
5. The MLFQ does not have priority boosting or differing time slice lengths.
   1. Priority 2. P1 is in the middle of its time slice in queue 2, as it didn’t use up time during its I/O operation.
   2. E.
   3. B. CPU-heavy jobs may sink to the bottom and get starved.
6. .

**In class review**

Locality

* Principle of locality: programs tend to use data and instructions with addresses near or equal to those they have used recently
* **Temporal locality**: recently referenced items are likely to be referenced again soon.
* **Spatial locality:** close items in memory tend to be referenced close together in time.
* Example slide 5: Does this function have a good locality with respect to array a?
  + It has good spatial locality bc were doing rows on the outer loop, and good temporal locality because were accessing sum repeatedly
* Example slide 6:
  + Does not have good spatial locality since we are accessing a different row each inner iteration, so we are jumping around in memory. Good temporal locality though because we are repeatedly accessing sum.

Cache memories

* Cache memories are small, fast SRAM-based memories managed automatically in hardware. They hold frequently accessed blocks of main memory.
* CPU looks first for data in the cache
* Typical system structure: (check slide 7)

Cache structure

* S = number of sets, E = number of lines per set, B = number of bytes per block
* First bit in line = valid bit, which says if data is in the block
* Next is tag
* Then block which stores the actual data
* Locate set. Check if any line in set has matching tag, check if valid bit of line = 1. If yes then it’s a hit. Go to block offset to find data.
* Address of word = t bits for tag, then s bits for set, then b bits for block offset

Cache simulation (direct mapped)

* We have 4 sets, so we need two bits in the memory address for set, block has two bytes per block, so we need 1 bit for offset, then tag gets all the remaining bits.

Cache simulation (2-way set associative cache)

* 2 sets, 2 lines per set (E = 2), 2 bytes per block
* If the last one was 0100 instead of 0000 we would get a miss, then which one do we replace? It depends on the cache and we don’t need to know.

Static linking

* Programs are translated and linked using a compiler driver:
  + Gcc -o prog main.c sum.c
  + ./prog
* Source files ->[translators]-> separately compiled relocatable object files ->[linker]-> fully linked executable object file (contains all source code and data)

ELF (Executable and Linkable Format)

* Standard binary format for object files
* ELF header, object file sections (.stuff), Section header table (describes object file sections like the locations and sizes of the sections)
* Elf format supports dynamic linking

Linker symbols

* **Global symbols**: symbols defined by module m that can be referenced by other modules. Ex: non static C functions and non static global variables
* **External symbols**: global symbols that are referenced by module m but defined by some other module
* **Local symbols**: defined and referenced exclusively by module m. ex: C functions and global variables defined with the **static** attribute.

ELF example

* Buf:
  + An array of 2 integers.
  + Defined globally in m.o (not m.c)
  + It is an initialized, non-static global variable, but it is declared as extern in swap.c, which is called before it is defined in m.c. so it is type extern.
  + Goes in .data
* Bufp0:
  + Pointer to an integer, defined in swap.o
  + Type global and goes in .data since it is initialized
* Bufp1:
  + Pointer to an int, defined in swap.o
  + Uninitialized global variable, goes in .bss
* Swap:
  + Function defined in swap.o
  + Non static, so type global
  + It is a function so it goes in machine code .text

How linker resolves duplicate symbol definitions

* **Program symbols** are either strong or weak
  + **Strong**: procedures and initialized globals
  + **Weak**: uninitialized globals
  + Ex:
    - Int foo = 5; strong
    - Int foo; weak
    - P1() {} strong

**Linker’s symbol rules** (on the test)

* Rule 1: multiple strong symbols are not allowed
  + Each item can only be defined once. Otherwise we get a linker error
* Rule 2: given a strong symbol and multiple weak symbols, choose the strong symbol.
  + References to the weak symbol resolve to the strong symbol
* Rule 3: if there are multiple weak symbols, either pick an arbitrary one or throw an error.
  + Modern gcc throws an error by default, but you can override this with gcc -fno-common

Linker puzzles

1. We have a weak symbol x and two strong symbols p1(). Since there are two strong symbols with the same name p1, we get an error by rule 1.
2. Two uninitialized globals x (both weak). Either pick an arbitrary one or throw an error by rule 3.
3. Two weak symbols x, but of differing types. Compiler may pick one or throw error. If the compiler picks double x, writes to it will overwrite y since y comes right after x in memory.
4. A strong symbol x and weak symbol x. references to weak symbol will go to strong symbol. Since weak x is a double, writing to it will put a double amount of memory to int x and overwrite y.
5. A strong and weak x. writes to weak x will go to strong x. P2 is a library which overwrites x when called in p1. To fix it make one static

Library interpositioning

* Powerful linking technique that allows programmers to intercept calls to arbitrary functions.
* Useful for debugging, monitoring and profiling, security
* Can occur at:
  + compile time: when the source code is compiled
  + link time: when the relocatable object files are statically linked to form an executable object file
  + load / run time: when an executable object file is loaded into memory, dynamically linked, and then executed.

Compile time interpositioning: makes no sense

* you’re basically just overwriting a stock function to perform custom behavior. So the example on slide 31 when you include malloc.h and call malloc(size) you’re actually calling mymalloc(size) which still calls malloc and gives you a pointer to memory but also prints the details of the allocated memory while its at it.
* You need the -I flag for it to work

Process scheduling

* Process: an instance of a running program
  + Provides each program with two key abstractions
    - Logical control flow: each program seems to have exclusive use of the CPU
    - Private address space: each program seems to have exclusive use of main memory
  + Process states: Running, ready, blocked
* **Turnaround time:** time of completion – time of arrival
* **Response time**: time of first execution – time of arrival
* Fairness: all processes get the same amount of CPU over time
* FIFO
  + Con: Long running jobs delay short jobs. Turnaround and response time suffer
* SJF: choose the waiting job with the shortest time. Non preemptive.
  + Shortest jobs don’t get blocked up by long jobs
  + But if a longer job is already running you’re fucked
  + Con: long jobs cannot be interrupted and delay short jobs. Response and turnaround time suffer.
* STCF: SJF, but with preemption
  + If a long running job is going and a short job comes in, it takes over
  + Con: This and SJF are not fair to longer jobs
* RR: run each job for a time quantum, then switch to next job in the queue
  + Good for response time
  + More fair to all processes
  + Not as good for average turnaround time
* MLFQ
  + Multiple levels of RR
  + This and RR are the only ones that really work when we don’t know the length of the jobs ahead of time
  + Rules
  + Problems with basic MLFQ
    - Starvation
    - Game the scheduler: issue an I/O operation after 99% of your time slice to avoid getting bumped down a level
    - A program may change its behavior over time: CPU bound process turns into I/O bound process
  + For the exam, he will clarify what variant of MLFQ he is talking about in the specific problem