131 Exam

The CSCI131 exam is worth 30% of the overall mark in the subject. The exam will be marked out of 50 and scaled to 30%. The exam questions will mostly be taken from the following list:

Explain what you understand by "machine code programming" and assembly language programming. (2)

Explain the hardware extensions to a CPU that permitted the creation of simple "operating systems" that handle I/O in a uni-programming (single application) environment. (2)

Explain the benefits of the 1960's OS advances of "time-sharing" and "multi-programming". (2)

What do you understand by the term "I/O spooling"? (1)

"Overlays" and "virtual memory" both exploit storage on disk to allow execution of programs whose memory requirements exceed available physical memory. Explain the fundamental difference between these two technical approaches. (1)

Why is the "Algol" programming language considered important despite its proving far less popular than contemporary languages like Fortran-IV and COBOL? (1)

Provide a diagram showing the major components in a simple computer system and briefly explain their roles. (4)

Explain how data are transferred between disk and memory using "direct memory access". (2)

Explain how data are transferred to and from slow character-oriented devices such as keyboards and printers. (2)

Explain the "fetch-decode-execute" cycle of the CPU. (3)

Explain your understanding of "CPU registers". (2)

Explain the use of:

instruction register,

program counter,

stack pointer,

stack frame pointer. (2)

Instructions can be implemented by actual circuitry, or may be "micro-coded". Explain the meaning of "micro-coded". (2)

Explain how to find the largest integer in an array of positive non-zero numbers using pointers for array access. Illustrate your explanation with C code or assembly language code. (3)

Explain the terms “op-code” and “instruction mnemonic” and how they relate. (2)

What is the difference between “big-endian” and “little-endian” memory. (1)

Explain how I/O devices can be controlled using “polling” and why this approach is inefficient. (2)

Explain how “interrupt driven” I/O operations are performed. (4)

Explain “vectored interrupts” and “interrupt priority”. (2)

Explain how “memory-mapped” I/O differs from I/O using explicit input-output instructions.(2)

Identify one approach to speeding access to data in main memory. (1)

Explain how “instruction caches” and “data caches” can be used to increase processing speed. (2)

Explain your understanding of an “instruction pipeline” and how this can increase processing speed. (1)

Explain the difference between “arithmetic shift right” and “logical shift right”. (2)

Explain how signed integers are represented in both “1s complement” and “2s complement” schemes.(2)

Explain the difference between the “real numbers” of mathematics and the “floating point” numbers used in computers. (1)

Explain the representation of a floating point number. (1)

Explain the “round off” problems that occur when using floating point arithmetic. (1)

Explain why there are problems when performing “equality checks” on computed floating point values. (1)

Why was the “packed decimal” format for numbers favoured in business data processing applications?(2)

Explain how auto-increment (and auto-decrement) addressing modes can be exploited when working through an array of data values. (2)

Explain how “indexed addressing mode” can be used when accessing an array element and when accessing a field of a struct. (2)

Explain how “relative addressing” works and explain the advantage of this addressing mode. (3)

Explain how subroutines (functions) are called in a stack based architecture like the PDP11; your answer should cover how arguments and a return address are pushed on stack, how the return from subroutine works, and the tidying up steps needed. (4)

Explain how space on the stack can be used for local variables of a subroutine (function). (2)

Write the code of a recursive C function that will print the value of an integer variable. (2)

Illustrate how the stack would be used by a recursive function that prints the value of an integer variable (you should include a diagram illustrating the contents of the stack at the point where the digit '7' is about to be printed for the function initially invoked to print the value -3756892). (2)

Write the code of a non-recursive C function that will print the value of an integer variable.(2)

Sketch the assembly language version of a function that computes the address of a specific struct in an array of structs. The structs are 16-bytes in size. On entry to the function, register r0 is to hold the index value into the array (0-based array as in C/C++), while register r1 holds the base address of the array. The address of the chosen struct should be returned in register r0. (2)

Explain how a subroutine (function) can determine the addresses of argument values and local variables by reference to the value in the "stack pointer" register. (2)

Given that it is possible to determine the addresses of argument values and local variables by reference to the stack pointer, explain why it is usual to utilize one of the other registers as a "stack frame pointer". (2)

Explain how the "stack pointer" and the "stack frame pointer" are used together to allow easy access to the argument values and local variables of a subroutine (function). (4)

Explain the purpose of the "origin" directive in assembly language.(1)

Explain the concept of "macros"; illustrate your answer with a simple #define macro for C/C++ code. (2)

Explain how "race conditions" can arise and cause errors in interrupt driven systems. (2)

Explain the concept of a "critical region". (2)

Why did computer architects incorporate specialized "supervisor call" (trap) instructions instead of simply allowing subroutine calls to OS routines? (2)

Explain how a simple OS could be constructed using "supervisor call" (trap) instructions to offer services needed by application programs. (3)

Why did many simple single user operating systems use wait-loops for control of I/O instead of interrupts? (2)

When DEC's computer architects decided to extend the PDP11 architecture to support multi-tasking operating systems, they added a second stack pointer as well as memory-management hardware. Why was the second stack pointer needed? (1)

Identify four C++ features that are missing from C. (2)

C's use of "header" files in the context of separately compiled pieces of code represented a great advance over earlier languages that had allowed for separate compilation (languages such as FORTRAN II/IV). Explain the benefit of using headers. (2)

Explain the relationship between the GNU project and the Linux project. (1)

Explain Stallman's argument that the Linux system should really be known as **GNU**/Linux. (1)

Explain what you understand by a "Linux distribution" (Linux/distro). (1)

The kernel – software that manages and allocates computer resources. Identify four principal resources managed by the kernel code. (2)

Explain why there will be numerous processes running as "root" on a Linux system (or "system" on Windows). (2)

Explain what you understand by a "daemon process". (1)

Explain "orphan process" and "zombie process" as these terms are used for Linux. (2)

Processes on Linux have 5 major areas of memory. Explain the different usage of each of these areas. (2)

A simplified version of the Linux virtual memory model has the user memory areas as one large virtual address space. Diagram this memory model. (2)

The OS maintains several data structures for each process. Identify four of the types of data maintained by the OS. (2)

Explain the usage of Linux's "fork" system call. (2)

The combination of "fork" and "exec" is more common than a simple process fork. Explain what happens with "fork" and what subsequently happens with an "exec" system call. (2)

If a process creates a child process with "fork", it is expected to wait for the child process to finish through use of the waitpid() system call. Why should it wait? What happens if it doesn't? (2)

Explain what the "brk" system call does. (1)

Explain the use of the "malloc" and "free" functions in the standard library, and outline how storage space in the heap gets managed. (4)

The core parts of the Unix/Linux OS were created long before "exceptions" were conceived. So, how does the OS report errors that occurred when handling system calls, and how does this affect the way that programs should be written. (2)

Identify four of the main groups of system calls. (2)

One of the aspects that made Unix superior to contemporary operating systems was its approach to handling I/O devices. Explain the Unix approach. (2)

The original implementation of many of the system calls often returned pointers to static data structures using code along the following lines:

```
struct demo *syscall(arg1, arg2, arg3) {
        static struct demo results;
```

```
    …
    return &results;
}
```

Explain the problems that can arise with this usage and identify how modern versions of the same system calls avoid these problems. (2)

Explain Unix/Linux's "owner, group, others" permissions on files. (1)

Explain what you understand by "root privilege" and "sudo privilege". (2)

Detail the data that are held in /etc/passwd and explain the use of "shadow passwords". (2)

Three different types of disk partition were described in the lectures. Identify and explain the usage of these different types of disk partition. (3)

Explain how a file system partition on disk incorporates four main elements and briefly explain their usage. (4)

Explain the advantages and disadvantages of a file-system using "contiguous allocation". (2)

Explain the advantages and disadvantages of a file-system using linked-list allocation. (2)

Explain how the Unix/Linux i-node based mechanism for recording files allows for unrestricted size and fast random access to any block of the file. (3)

Linux's ext2 file system used several "blockgroups", each with its own boot-block, i-node table and data blocks, in a disk partition. What was the justification for this extra complexity? (2)

Explain the most fundamental difference between the Unix/Linux file-system hierarchy and that of Windows. (1)

When installing a new service on a Linux system, you will need to create configuration data and logging directories. Where are these elements typically created? (1)

Linux directories can contain ~4 different types of files - "directories", "normal files", "links", and "devices". Explain each of these types. (2)

What is a "chroot jail" and why would you want one? (1)

Increasingly, storage on rotating disk media is being replaced by storage on solid-state memory. Why is this likely to lead to changes in the operating system and the re-working of approaches to database access? (1)

In a multi-threaded program, each thread has its own sequence of nested function calls and therefore needs its own stack. Explain how the Linux virtual memory model for a process can adapt to handle multiple threads and their stacks. (2)

In what circumstances might you decide to make use of SIGUSR1 and SIGUSR2 signals. (1)

Explain how “pipes” can be used for inter-process communication. (2)

Explain how Linux processes can utilize “shared memory”. (2)

Explain how the “domain name system” works to support communication between processes running on computers attached to the Internet. (4)

Explain the concepts of “port” and “socket”. (2)

Explain the primary difference between UDP/IP and TCP/IP. (1)

Explain the role of the “inetd” process. (1)

Explain what a “page fault interrupt” is and how the OS would handle a page fault. (3)

Explain why the more complex “upstart” system has largely replaced Unix/Linux’s traditional “init” approach. (2)

Explain why C++ has to rename all functions prior to generating linkage data. (2)

Explain what a “target” is in a makefile, and illustrate a target definition for an executable comprising several compiled project files and code from libraries. (2)

Explain the concept of “static linking”. (2)

When using static linking, the order in which libraries are listed is significant. Explain why. (2)

Draw diagrams showing the different Linux virtual memory maps for programs built using static linking and programs using shared objects. (2)

What are the advantages and costs of building a “shared object” version of a library? (2)

Both the compilation and linkage steps in the program build process may need to be configured to use libraries. Explain when special configuration is required and how it is performed for compiler and linker. (3)

You imported a non-standard specialist library that came with both .a and .so versions of the compiled library code. You found that you could get your program to work with the .a library but not with the .so version. Explain why this might happen. (2)

Explain the use of the environment variable LD_LIBRARY_PATH. (2)

Explain how pipelines work in standard Unix/Linux shells such as sh, bash, or csh; write a pipeline that will list the disk space consumed by all files and directories with the output ordered to show the largest first. (2)

Why did touch screen command shells finally become popular? (1)

Explain how a two-pass absolute assembler works. (4)

Explain the form of a file generated by an assembler for relocatable code that is to be processed by a linker. (3)

Explain how a linker can build an executable program image on disk and how a loader would map this into memory. (2)

Explain the work performed by the “cpp” pre-processor in the standard gcc/g++ compilation chain. (2)

Explain the role of the “lexical analyser”. (3)

The typical compilation process involves the following steps:

Lexical analysis

Syntax analysis

Semantic analysis

Intermediate code generation

Machine independent code optimization

Code generation

Machine dependent code optimization.

Explain each of these steps. (7)

Explain “parse tree”, “abstract syntax tree”, “attributed syntax tree”. (3)

Describe two optimisation steps that are potentially applicable to “intermediate code”. (2)

Explain “peep hole” optimisation and provide an example of its use when optimising machine specific code. (2)

Assuming the common definitions of expression, term, factor etc, explain how a shift reduce parser would parse the expression: U*T + 0.5*ALPHA*T*T (expression for distance travelled in time T with starting velocity U and acceleration ALPHA). (4)

Knuth developed his theory of LR parsing in 1965, but it was several years before there were any practical implementations and these implementations were simplified. What is the problem with Knuth’s LR scheme that makes it difficult to implement? (1)

Explain what you understand by “a recursive descent parser”. (2)

Explain how the “flex” and “bison” tools are used to build a compiler. (4)

Explain the construction of the “gcc” compilation covering why essentially the same compiler code can handle multiple languages (C, C++, Java, Fortran, …) and generate code for multiple target machines (ARM, IA-32, IA-64, ..).

Explain why C and C++ always pass arrays by reference (1)

Explain how C/C++ handle multi-dimensional arrays in function calls. (1)

Explain how the code generated for C++ functions is made much more complex by the need to handle exceptions. (2)

Explain how the C++ compiler handles generic (template based) function calls, illustrating your explanation with a template function that returns the largest value in an array – a function that may get used with short[], long[], double[]. (2)

A C++ class defines a data structure and the functions used to manipulate that structure. You can have a class Point with [x,y] data members and methods like double radius() and double theta() (converting from cartesian to polar coordinates). Code using instances reads like: Point p1(3.0,5.0); …; double r = p1.radius(). Explain how the compiler generates code that gets the radius() function use the correct [x,y] data. (2)

Explain the use of "virtual tables" for classes that form part of a class hierarchy. (3)

Explain "slicing" bugs that can occur with carelessly written code using classes from a class hierarchy. (2)

Explain the "mark and sweep" approach to garbage collection. (2)

Why do languages like Java and C# use double indirection pointers for "object reference variables". (2)