Comp 1521 Notes

Lec 1.1 – Introduction to MIPs

Concepts

It is useful to know assembly language because

- Sometimes you are required to use it
  - Writing code to interact directly with a device
- It improves your understanding of how compiled programs execute
  - Can be helpful when debugging
- It is good for performance tweaking
  - You can see how you can minimise the number of instructions to execute code faster
- You can create games in pure assembler

Modern CPUs typically have

- A set of data registers and control registers
- A control, arithmetic-logic (ALU) and floating-point (FPU) unit
- Caches
  - Caches range from L1 (fastest and smallest) to L3 (slowest and largest)
  - Sometimes there are separate caches for data and instruction
- Access to memory (RAM)
  - Address generation unit (AGU)
  - Memory management unit (MMU)
- A set of "simple" instructions
  - Transfer data between memory and registers
  - Compute values using ALU/FPU
  - Make tests and transfer control of execution

Instructions are bit patterns stored in RAM

- CPUs can fetch an instruction from memory, then decodes it to determine what to do
- The CPU then executed that instruction before moving on to fetch the next instruction
- Adding/removing instructions, and changing variable layout in memory changes bit pattern for instructions
- Executing an instruction involves:
  - Determining what the operator is
  - Determining if/which register(s) and memory location is involved
  - Carrying out the operation with the relevant operands
  - Storing result (if any) in the appropriate register/memory location

All data in a computer is represented as binary

- Computers are massive circuits, and electricity can be thought of as being off or on
  - 0 represents off and 1 represents on —> this is a base-2 system for binary

MIPS is a simple architecture

- 1521 uses MIPS32 → instructions are 32 bits long and specify an operation, and 0 or more operands
- 1521 uses simulators, not real MIPS hardware:
    - mipsy … → a command-line based emulator
    - mipsy-web … → web (WASM) GUI-based version of mipsy
- MIPS has several classes of instructions:
    - Load and store → transfer data between registers and memory
    - Computational → perform arithmetic/logical operations
    - Jump and branch → transferring control of program execution
    - Special → miscellaneous tasks
    - Coprocessor → standard interface to various co-processors, which implement floating point operations → not studied in 1521

Conventions of assembly language

- Write instructions using names rather than bit strings
- Refer to registers using either numbers or names
- Allow names (labels) associated with memory addresses
- An assembler translates this assembly language to binary CPU instructions

To run an assembly file for 1521, we use

1521 mipsy prog.s

- In terminal
- prog.s is the file written in assembly language

Code

Lec 1.2 → Intro Contd

Concepts

Assembly language

- Allows us to represent binary instructions in human-readable ways
- Allows us to add things like constants
- Gives us ways to configure other memory sections
- Gives us labels – ways to name points in memory without having to deal with addresses

Registers

- Computer RAM is fast, but not fast enough for the scale a CPU operates on so the CPU has a small amount of storage on the chip itself → we call this storage registers
- MIPS32 CPUs have 32 general-purpose registers, each of which are 32 bits (same as one int in C)
- The program counter register keeps track of which instruction (the memory address of the current instruction) to fetch and execute next → it is modified by branch/jump instructions
- Almost all computations happen between registers → if we want to compute something, we need to load each input into a separate register

In MIPS, we denote registers with a $

- Registers can be referred to using a number ($0.. %31) or by symbolic names ($zero.. $ra)
    - $zero ($0) is special → it always has the value 0 no matter what
    - $ra ($31) is also special
    - Conventions somewhat restrict what we can do with the remaining 30 registers
    - We can do whatever we want with registers $t0 to $t9 → NOTE that t0, t1, t2 are names, and are not the same as $0, $1, $2
- We can't change the names of registers since they indicate conventions

System calls are another type of instruction

- Operating systems tries to enforce reasonable limits on what programs can actually do → CPUs are also bound by these restrictions → CPUs must ask operating systems to do some things on their behalf by making a system call
- Mipsy provides some of these system calls that allow programs to interact with the outside world by printing or scanning

The system call workflow

1. We specify which system call we want in $v0
    - li $v0, x
2. We specify arguments (if any) in $a
    - li $a0, x
3. We transfer execution to the operating system
    - syscall → the OS will only fulfil our request if it looks "sane"

.text

- . signifies a directive
- .text means it is in the text part of the memory —> code
- .data means it is string literals and global variables
- .asciiz means we are storing a \0 terminated ASCII string in the current data segment

Conventions

- We write comments on the same line as the code we are commenting on, and we use # instead of //
- We don't indent to show structure → instead we:
    - Don't indent labels at all
    - Indent instructions by one step (tab)
    - Have equivalent C code as inline comments
-

We run mipsy by entering "1521 mipsy" into the terminal

- We then enter a mipsy terminal-like interface with [mipsy]
- Type help to see commands
- load prog.s → loads an assembly file
- run → runs the loaded file
- exit → exits mipsy
- NOTE: we don't need to compile
- To run programs directly from the terminal, type 1521 mipsy prog.s

It is best to simplify your C code before translating it

- Simplified C is generally written so that each line of C code maps to one MIPS instruction → make sure to compile your simplified C and check that it still works
- Translating it should be much easier now

Coded programs:

- hello_world.s
- math_simplified.c → math.s
- square_and_add_simplified.c → square_and_add.s

Code

```
        .text

main:

        …
        …

        li $v0, 0
        jr $ra

        .data
```

- Basic structure of a MIPS assembler program

tx$

- In MIPS, we denote registers with a $ → the tx is the name of the register → we can have $t0, $t1, $t2 etc.
- Registers can be referred to using a number ($0.. $31) or by symbolic names ($zero.. $ra) → NOTE that $0 and $t0 are different → t0 is a name
    - $zero ($0) is special → it always has the value 0 no matter what you do to it
    - $ra ($31) is also special → it is directly affected by two instructions we will learn later
    - We can use the remaining 30 registers however we want, but there are some conventions we have to follow → $t0 to $t9 are free to use however we want
    - $v0, $a0, $ra are needed for certain things
- Can't change register names since they indicate conventions

li $t0, x

- Stands for "load immediate"
- This instruction loads data into a register → in the above, it would load x into a register called $t0

mul $t2, $t1, $t0

- Mul is the multiply instruction → multiplies the data stored in $t1 and $t0, then stores the result in $t2

Common mipsy syscalls

Service	$v0	Arguments	Returns
printf("%d)	1	int in $a0	
fputs	4	String in $a0	
scanf("%d")	5		int in $v0
fgets	8	Line in $a0, length in $a1	
exit(0)	10		
printf("%c")	11	char in $a0	
scanf("%c")	12		char in $v0

- $a → a for arguments

la

- This is the load address instruction → used for loading fixed addresses into a register
- We use la to load labels
- Mipsy lets you use la and li interchangeably, but we shouldn't

move

- Instruction for copying values between two registers

JUST LOOK AT MIPS GUIDE FOR LITTLE INSTRUCTIONS LIKE MOVE AND ADD AND MUL

Lec 1.3 → MIPS Controls

Abiram's lec 2 → things not covered in T3 lec 1.2

Concepts

If/else statements and loops don't exist in MIPS → we have to use branching/jumping instructions to implement them instead

- There are several different branch/jump instructions, each fulfilling specific if/else/while conditions
- If a condition is true, it will jump to a different instruction
- In 1521, the j and b junmp instructions are essentially the same thing (interchangeable)
- In mipsy, we can replace the second argument, r_t with a constant (immediate value like 1, a, 2.3, etc)

In C, goto allows jumping to an arbitrary point within a program (as long as that point has been defined with a label)

- We can jump to anywhere within our program however we wish
- Labels are defined in C the same way as in MIPS
- We use goto statements to simplify if/else statements and while loops
- NOTE: programs always execute in a linear, downward fashion unless goto statements are used
- NOTE: for if statements where something == 0, we like to take the inverse since there is not branch if sth == 0 instruction, only branch if sth != 0 instruction (bnez)

Complex conditionals

- For conditions with or (||), we just use two if statements that goto the same label
- For conditions with and (&&), we just use the inverse (De Morgans law) to make it an or (||) condition:

```
if (x >= 0 && x <= 100) {
        printf("in bounds\n);
} else {
        printf("out of bounds\n");
}
```

Is equivalent to

```
if (x < 0 || x > 100) {
        printf("out of bounds\n");
} else {
        printf("in bounds\n");
}
```

Simplifying loops

- For loops should be broken down into while loops that should be broken down into if and goto

- The general structure is:
    - Loop initialise
    - Loop condition (check if we need to exit)
    - Loop body
    - Loop step (increment)
    - Loop end
- We use labels to show the structure
- We usually invert the loop condition, making it goto loop end if it activates

EXAMPLES → IN lec02 DIRECTORY:

- print_if_even_simplified.c → print_if_even.s
- count_to_10_simplified.c → count_to_10.s

Code

label:

    …code…

- Defines a label in C

goto label

- Jumps from the line goto is on to wherever the label is
- NOTE: don't actualy use goto in C programs → only use when simplifying C since it is harder to read and typically results in slower programs due to compiler difficulties

Lec 2.1 → MIPS Data → Pointers

JOHN SHEP SO WATCH THE TUESDAY LEC INSTEAD AND COME BACK

Concepts

The memory subsystem typically provides the capability to load or store bytes

- 1 byte = 8 bits
- Each byte has a unique address, and memory can be thought of as a gigantic array of bytes with addresses as the index
- Usually, only small group of bytes can be loaded/stored in a single operation
- General purpose computers typically have complex cache systems to improve memory performance → WE DON'T COVER AT CACHES IN 1521

MIPS32 has addresses in 32-bit

- Smaller memory than the usual 64-bit machines
- Only load/store instructions access memory on the MIPS
- 1 byte (8 bits) are loaded/stored with lb/sb
- 2 bytes (16 bits → half-word) are loaded/stored with lh/sh
- 4 bytes (32-bits → word) are loaded/stored with lw/sw
- The memory addresses used for load/store instructions are the sums of a specified register and a 16-bit constant (often 0) which is part of the instruction

Alignment

- Bytes must be stored at addresses that are multiples of 1
- Half-words must be stored at addresses that are multiples of 2
- Words must be stored at addresses that are multiples of 4
- Mipsy aligns for us when we store multiple values of different sizes in memory
-

Code

STORING → sb, sh, sw

- These commands STORE the value of a register into an address
  - MUST BE A REGISTER
- $, address

LOADING → lb, lh, lw

- These command LOAD the value at an address into an address
- $, address

array_label(index)

- Basically dereferences the indexth slot in the array
- Does *(array_label + (index * sizeof item)) → array[index] in C
- Value must be byte aligned to the type of data stored (1 for byte, 2 for half word, 4 for word

- Note that array_label is an array

```
        .data
x:      .space 4

        .text
        la      $t0, x
        li      $t1, 42
        sw      $t1, x
```

- We store pointers in the .data directive? → pointers are just labels
    - To dereference them, we need to load them into a register and then dereference the register using (brackets)
    - ```
      la      $t0, my_pointer
      lw      $a0, (my_pointer)
              Loads the word at my_pointer into $a0
      ```
- .space is the MIPS equivalent of malloc → .space 4 allocates 4 bytes of space
- Here we have x, the address of 4 uninitialized bytes in memory
- We load that address into $t0, then load 42 into $t1
- The sw instruction loads $t1 into the 4 bytes of memory at x → MIPS equivalent of creating a pointer to a malloced space, then dereferencing that pointer to change the value stored there

```
Y:      .space 32

        li      $t2, 1
        li      $t3, 'a'
        sb      $t3, y($t2)
```

- Malloced 32 bytes of memory, then int t2 = 1, char t3 = 'a'
- y($t2) is the equivalent of y[t2] = y[1] in C, so the last line means storing 'a' in the slot indexed 1 of the array → we essentially get arrays by storing smaller memory data types in a larger space of allocated memory

```
li  $t0, 42
li  $t1, 0x10000000
sb  $t0, 0($t1)
lb  $a0, 0($t1)
```

- sb $t0, 0($t1) → stores 42 at the address 0x10000000
- lb $a0, 0($t1) → loads the value stored in 0x10000000
- If we view $t1 as a pointer, 0($t1) is essentially dereferencing that pointer

```
        .data
array_label:
        .word   0:ARRAY_LEN
```
```

- Creates an array called array_label that stores ARRAY_LEN words that are initialised to 0
- Alternatively, we can just list the values we want in the array
    - .word x, y, z → array containing words x, y and z

```
    .data
Y:
    .word x
```

- Creates a global variable called Y that stores the word x

EXAMPLES

- Lab03

Lec 2.2 → MIPS Data Contd

Concepts

Global variables are variables that can be accessed anywhere within a program

- They are created by declaring a variable before main
- Unlike constants, they are variables → their value can be changed → changing the value of a global variable in a function will change that value everywhere
- Because .space allocates uninitialized bytes, we want to use .byte/.half/.word which stores an input value in succession at the next location of the current segment (the main directives: .data mainly) → we can still use .space for global variables that are declared without a value
- Since .space allocates uninitialized memory, we can use it to declare global variables that do not have a value
- However, to declare global variables with a value, we use .byte/.half/.word followed by the value to allocate memory initialized by that value

The data segment

- In MIPS, everything stored in the .data segment is stored right next to each other
- If we allocate more memory than the size of some data type, that memory could be treated as the memory for an array of that data type → e.g. .space 4 could be an array of 4 chars since char only takes 1 byte
- .byte/.half/.word can take multiple values as input and stores them next to each other → e.g. .byte a, b, c, d would allocate a byte for each char and store a, b, c, d in these 4 successive bytes → char array[4] = {a, b, c, d}
- HOWEVER, it should be noted that .space does not automatically allocate memory with the correct alignment while .word does → an int requires 4 bytes and so in MIPS needs a 4-byte alignment (the address must be divisible by 4) → this will cause errors with store instructions → explanation at T2 lec 2 26:30
-

Load vs store

- Load instructions load values into registers
- Store instructions store values from a register into memory

Arrays

- In C, array[i] is really just dereferencing the memory i * offset (sizeof data type) over from the memory of array (where the array starts → array[i] = *(array + (i * sizeof(data type))) → we use this fact to work with arrays in MIPS
- In memory, 2D arrays really just have each row of the array placed after the other, so we need a formula where the row indicator offsets the array address by one row
- The 2D array formula is thus array address + (row * number_of_cols) + col
    - array[i][j] = *(&array + (sizeof(data type) * ((i * NUM_COLS) + j)))

to define constants in MIPS, just type the constant name = value above the text segment

Structs

- To calculate the size of the structs, sum the size of each field
- The fields of a struct are allocated successively in memory, so like arrays we can access different fields by offsetting the size of previous fields → it is good to put the offset amount for each field as a constant using other constants
- It is also good to put the sizeof a struct as a constant using other constants
- E.g., for a struct with an int then a 20 char string then another int, we have

```
FIRST_FIELD OFFSET = 0
SECOND_FIELD_OFFSET = 4 + FIRST_FIELD_OFFSET
THIRD_FIELD_OFFSET = 20 + SECOND_FIELD_OFFSET
SIZEOF_STRUCT = 4 + THIRD_FIELD_OFFSET
```

Demos/Examples:

- global_increment.c → global_increment.s
- array_bytes.s
- flag.c → flag.s → 2D arrays
- struct.c → struct.s

Code

.align int

- Ensures that the next thing you store in memory (.data) is int^2 bytes-aligned by adding the necessary amount of padding (unused bytes)
- We usually use .align 2 since we work with words (4 bytes) the most

```
lb      $a0, x($t1)
```

- For lb/lh/lw
- Loads the value at the memory address x + $t1 → x is a memory address
- x cannot be another register, it has to be a label or an immediate value
- lb will compute the address of x + $t1 and then load the value at that address into $a0 (or any register) → basically, $a0 = *(x + $t1)

```
la      $t0, structX
lw      $t1, Y_OFFSET($t0)
```

- loads the word stored in Y field of structX into $t1
- Y_OFFSET($t0) → *($t0 + Y_OFFSET)
- structX is a label declared in .data that has allocated memory exactly for the fields (subvariables/attributes) of the struct

Lec 3.2 → MIPS Functions

3.1 was a short assignment overview since it was a public holiday

Concepts

The MIPS calling conventions set out ground rules on how we use functions

- The MIPS ABI lays out how different code should interact with each other
- We follow these rules to make sure that functions work nicely with each other

Functions are really just named pieces of code that can:

- take in arguments if we want it to
- Use those arguments in computations
- Return a value to a caller

Passing into and returning from functions

- $a0 to $a3 registers are used to pass arguments to a function
- The $v0 register is used to return a value <= 32 bits from a function
  (hence li $v0, 0 for return 0 at the end of main)

However, functions can also use $t registers freely, so we need to make sure values in $t registers across functions are not destroyed (changed or overwritten)

- We must assume any function called will destroy all $t registers even if we code it to specifically not → assignment autotesting will check this by destroying your values intentionally
- Clobbering → the obliterating of an existing value in a register without eventually restoring it

We preserve values between function calls by using the $s registers

- Functions cannot permanently change the value of a $s register, so we can rely on our callee functions not clobbering any values we keep there
- BUT functions can temporarily make changes to $s registers, as long as they restore them to their original values once the function is done
- We do this by:
  - Saving the $s register's original value to RAM at the start of the function
  - Restoring the $s register's original value from RAM once the function is done
- Main is also a function and must therefore also follow these rules

The stack is a region of memory we can grow and expand

- We can save the original values in $s registers by using the $sp (stack pointer) register to keep track of the top of the stack
- We can modify the stack pointer to allocate more room on the stack for us to store our values
- Each function call creates a new stack frame (grows: moves the stack pointer down) that is destroyed (shrinks: moves the stack pointer back up to where it was) when the function ends → this is how a function 'allocates' and 'deallocates' room on the stack

- We create memory on the stack to store the original value of $s registers by subtracting bytes from $sp, and so at the end of the function we must restore the value stored in those bytes then add that amount of bytes back to $sp → the amount we add is the same as the amount we add
    - The amount has to be a multiple of 4
- We subtract to get more space since the stack starts at the top and subtracting moves the stack to a lower number
- push and pop are two pseudo-instructions used to store and restore $s registers
    - With multiple pushes in one function, we need the corresponding pops to be in reverse order:

```
push    $ra
push    $s0
push    $s1

… code …

pop     $s1
pop     $s0
pop     $ra
```

    - Otherwise the values will be mismatched
    - We also need to push and pop $ra since that is the address we want to jump back to after a function ends → more below

To call other functions, we just need to jump to them, but we also need to be able to jump back

- The jal (jump and link) instruction is used to call functions
    - It jumps to the given label
    - It also sets $ra (return address) to point to the next instruction before jumping, giving us a way to return to the caller function → the jr (jump return) instruction will make it so that it jumps back to the next instruction after the function call in main once the function ends
    - However,

Prologues and epilogues

- Prologues are the start of a function
    1. Use the begin instruction
    2. Push $ra then any $s registers we want to use onto the stack
- Epilogues are the end of a function
    1. Sometimes set the return value $v0 here
    2. Pop any $s registers saved to the stack in reverse order, then pop $ra
    3. Use the end instruction
    4. Return to the caller with `jr $ra`

Leaf functions are functions that don't call any other functions

- Leaf functions don't need to preserve $ra since they don't use jal
- Leaf functions shouldn't need to use $s registers since they can use $t registers as they don't have any function calls within them

- Leaf functions thus do not need prologues and epilogues since they don't need to preserve values

Examples:

- factorial.c → factorial.s → recursive functions in MIPS and function header comments
- t_overwrite.s → shows how $t0 gets overwritten in function calls

Code

push R_t

- Allocates 4 bytes on the stack ($sp = $sp – 4) and stores the value of R_t to those 4 bytes

pop R_t

- Restores the value on the top of the stack into R_t, then deallocates the 4 bytes allocated on the stack by push ($sp = $sp + 4)

Lec 4.1 → Functions contd

Concepts

The frame pointer ($fp)

- This register points to the bottom of a given function's stack frame ($sp points to the top btw) → $fp points to the same value as $sp before a function does any pushes/pops
- The frame pointer combined with the saving of older values of $fp to the stack forms a linked list (of sorts) of stack frames → the function call stack
- Using frame pointers is optional, just useful for debugging
- Frame pointers track the original value of the stack pointer (at the start of the function), giving us a mechanism to prevent chaos if a function pushes/pops too much

The begin and end instruction → optional like frame pointers

- Begin should be the first instruction in the prologue
  - Saves the od $fp to the stack and sets $fp to the current $sp
- End should be the last instruction before jr $ra
  - Restores $sp to point to the top of the previous stack frame, and restores $fp to point to the previous value of $fp (bottom of the previous stack frame
- These pseudo-instructions are given by mipsy and make situations where you push/pop a lot much easier to debug

Examples:

- more_calls.c → more_calls.s

INTEGERS → SECOND HOUR OF LEC

Hexadecimal → base 16

- we have digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F → A = 11, B = 12, etc.
- In C, the '0x' prefix denotes a hexadecimal
- %08x is a flag specifying an 8 digit hexadecimal representation padded with 0s if there aren't enough digits → if x is lowercase, it prints lowercase, if X then uppercase
- Hexadecimals are base 16 = 2^4, so one hexadecimal digit is 4 binary digits

Col Hex vals																
0x123A	1				2				3				A			
Col binary vals	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
	0	0	0	1	0	0	1	0	0	0	1	1	1	0	1	0

- So, we just need to write each digit of the hexadecimal as a 4-bit binary representation to convert from hexadecimal to binary
- To convert from binary to hexadecimal, just find what each group of 4 bits equals when using binary col vals (8, 4, 2, 1) and that sum is each digit of the hexadecimal

Negtaive integers must be represented as 2^n – x, where n is the number of bits and x is the absolute value of the negative integer

- FUCK JAS JUST WATCH THE ANDREW TAYLOR ONES WHEN HE GETS TO IT → NEED TO LEARN SIGNED AND UNSIGNED AND NEGATIVE INTEGER SHIT
- The <stdint.h> library lets us use specific sized data types → uint_16 declares an unsigned 16-bit integer
- Normal 8-bit signed integers can represent both positive and negative integers → they have 7 bits for their value, and the leftmost 8^{th} bit is for their sign → positive integers have a 0 as their 8^{th} bit, and negative integers have 1
    - As such, 8-bit signed integers can represent a minimum value of -127 and a maximum value of 127 → 127 = 2^0 + 2^1 + … + 2^6 + 2^7
- 8-bit unsigned integers can only be positive and use all 8 bits to represent numbers
    - As such, 8-bit unsigned integers can represent a minimum value of 0, and a maximum value of 255 → 255 = 2^0 + 2^1 + … + 2^6 + 2^7
- To signify that an immediate integer value is unsigned, we put a u after it → 1 becomes 1u

Code

size_t vName = sizeof(Data Type)

- size_t is used to declare a variable vName that takes up Data Type amount of bits
- So size_t vName = sizeof(uint32_t) declares a variable vName that takes up 32 bits because the size of uint32_t is 32 bits →

Lec 4.2 → Bitwise Operations

Concepts

Modern computers use binary numbers

- Binary → base 2, digits are 0 and 1

Col values:	128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
Bit	1	0	1	0	1	0	1	0
Endian significance	Most							Least

- On cse machines, prefixing 0b to a constant of 0s and 1s denotes a binary number
- Each digit represents 1 bit

But binary is hard for humans to read, so we can also write numbers in hexadecimal

- Base 16, digits are 0 to 9, A to F (A = 10, B = 11… F = 15)
- 0x prefix denotes a hexadecimal
- Allows a 16 digit binary numbers to be represented in 4 digits
- Each digit represents 4 bits

Octal is another representation for binary numbers

- Base 8, digits are 0 to 7
- A leading 0 on a constant denotes octal → 07563
- Each digit represents 3 binary bits (8 = 2^3)
- 0b101 100 = 054 → 44 = 5*8^1 + 4*8^0

Integer types

- <stdint.h> provides us ways to work with several integer types in C
- Signed integers are those that can be positive or negative → int
  - 8 bits can range from -128 to 127
- Unsigned integers are those that can only be positive → uint
  - 8 bits can range from 0 to 255
- Integers can have different sizes (number of bits) that impact how big the numbers they can store are

Modern computers typically use two's complement to represent integers

- Positive integers and 0 is represented in the normal way
- For an n-bit binary number, -b is represented $2^n - b$:
  1. Get the absolute value of b
  2. Invert every bit → negate it (~)
  3. Add 1

	128	64	32	16	8	4	2	1
|-5|	0	0	0	0	0	1	0	1
Invert ~	1	1	1	1	1	0	1	0
Add 1	1	1	1	1	1	0	1	1
-5	1	1	1	1	1	0	1	1

    - If this were an unsigned int, this would have value 251, but an 8 bit signed int only has values -128 to 127, which is how we know it is a negative number

NOTE: chars are just smaller ints (fewer bits – typically 8 bit/1 byte) that are then hashed to ascii

Endian-ness refers to the order in which a multi-byte quantity (a sequence of bytes) is stored in your computer

- Big-endian → the most significant byte is stored at the smallest memory address, and less significant bytes are stored at larger memory addresses
- Little-endian → the least significant byte is stored at the smallest memory address, and more significant bytes are stored at larger memory addresses
- Most modern general-purpose computers use little-endian orders
- Least significant → bit representing the lowest power
- Most significant → bit representing the highest power

sizeof

- sizeof returns the number of bytes used to store a variable or data type
    - We can multiply the return of sizeof by 8 to get the number of bits
- On CSE servers:
    - char = 1 byte
    - short = 2 bytes
    - int = 4 bytes
    - double = 8 bytes

Both C and MIPS have some bitwise operations

- The C code is beneath in the Code section of notes
- The MIPS code is in the 1521 document under CPU logical instructions

NOTE: using bits and bitwise operations, we can also do things like create sets

- We declare a big integer (say 64-bits) as a set, then we can contain values from 0 to 63 → the set contains an element a if the corresponding ath bit in the set is set to 1, else it is not in the set
- Then we can do intersection pretty easily using &, and union using |, complement using ~
- There are lots of other possibilities, but we should write obvious, human-readable code

To create a binary representation with n 1s at the end, we can just do (1u <<= n) - 1

- This is equivalent to the bit representation of 2^n – 1

Examples:

- xor.c → bit flipping using xor bit operation
- odd_even.s

- pokemon.c
- bitset.c → 1:44:44 of 23T1 lec 9 → cba coding it bc its rly long

Code

THIS IS FOR C CODING

a & b → bitwise AND

- Takes two bytes a and b, and performs a logical AND between the pairs of corresponding bits → Only sets the bit in the result if both bits in a and b are set
- 10:49 has a clearer explanation
- If we do some value & 1 (00000001 in binary) and the result is 1, the value is odd, and if the result is 0, the value is even → 16:53 → we call that 1 we are &ing with a bitmask (any arbitrary constructed value we use for some purpose with bitwise operations is called a bitmask)
- We can use & with a mask of a range of 0s to unset a range of a byte
- a &= b → a = a & b

a | b → bitwise OR

- Takes two bytes a and b, and performs a logical OR between the pairs of corresponding bits → sets the bit in the result if at least one of the bits in a and b are set
- We can use | with a mask of a range of 1s to set a range of a byte
- a |= b → a = a | b

^ → bitwise XOR (eXclusive OR)

- Takes two bytes a and b, and performs an eXclusive OR between the pairs of corresponding bits → sets the bit in the result if exactly one of the bits in a and b are set
- We can use this with any number that can be represented in 8 bits to encipher characters → xor is a symmetrical operation so putting the enciphered characters back through xor with the same number will decipher it → xor.c

~a → bitwise NOT

- Takes a byte a, and flips all the bit setting → set bits become unset and vice versa
-

a << x → bitwise left shift

- Shifts the bit settings of a byte a left x times, and the empty spaces at the right of the og byte after shifting are filled with 0 → bits that fall off disappear, and we don't want that so we need to be careful (in C, integers are 32 bits
- In a base 2 system, this effectively multiplies a by 2 x times
- DO NOT SHIFT NEGATIVE VALUES ALWAYS USE UNSIGNED VALUES
- a <<= x → a = a << x

a >> x → bitwise right shift

- Same shit as left shift

- Any bits that get pushed off the end disappear → BUT this is basically what happens when we truncate during integer division, so it is expected and not too bad
- In a base 2 system, this effectively divides a by 2 x times
- DO NOT SHIFT NEGATIVE VALUES ALWAYS USE UNSIGNED VALUES
- a >>= x → a = a >> x

With shifting

- If shifting, always use unsigned ints or there are problems
- Shifting by the same amount as there are bits will result in 0

Lec 5.1 – Floating Point Numbers

Concepts

C has 3 floating point types

- float → 32-bits
- double → 64-bits
- long double → 128 bits

Scientific notation has 3 parts

- The fraction/mantissa (1 <= number < 10 → must be expressed in this form)
- The base (usually 10)
- The exponent (the power of the base)
- 777.777 = 7.77777 * 10^2
  - Fraction/Mantissa = 7.77777
  - Base = 10
  - Exponent = 2
- 0.00777777 = 7.77777 * 10^(-3)
  - Fraction/Mantissa = 7.77777
  - Base = 10
  - Exponent = -3

Floating point notation is really just scientific notation in base 2

Col val	SIGN	4	2	1		1/2 (0.5)	1/4 (0.25)	1/8 (0.125)	1/16 (0.0625)
+7.625 =	0	1	1	1	.	1	0	1	0

- So in binary, 7.625 is just $1.111010_2 \cdot 2^2$
  - Fraction/Mantissa = 1.111010
  - Base = 2
  - Exponent = 2
- In C, we store 32-bit floating point numbers like so
  - 1 bit for sign
  - 8 bits for exponent using bias 127
  - 23 bits for the Mantissa

	Sign	Exponent (bias 127)								Mantissa																							
Col val		128	64	32	16	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$			…																
+7.265 =	0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	…														
+7.265 =	1	$2^{129-127}$								$2^0 + 2^{-1} + 2^{-3}$																							

  - 1 bit for sign

    - 8 bits for exponent using bias 127 → range of exponent is -127 to 128
    - 23 bits for mantissa (note that we start at 2^{-1} since we will always have 2^0 set to 1 due to scientific notation)
- +7.265 = $1 \cdot 2^{129-1} \cdot (2^0 + 2^{-1} + 2^{-3})$
- HOWEVER, THIS FLOATING POINT REPRESENTATION DOES NOT WORK FOR 0 AND IRRATIONALS → 13:00 Lec 10 23T2
    - Attempts to represent irrational numbers or numbers with lots of decimal places will cause a floating point error
    - Also have problems storing 0.1???
    - Shit with limits that fuck up (usually when dividing using expressions involving floating point numbers that have similar approximate values) is called a catastrophic cancellation error

To account for tiny differences when comparing floating points, we never use == or !=

- Instead we set a threshold that is very small, and if the absolute value of the difference of the floating point number and its approximate value is less than that threshold, we take it to be the same
- A good threshold is if something is within 0.000001 of a number

For 8-bit ints, we can store 0 to 255 → that is the real range

- But by subtracting a 'bias' (127) from both sides, we can get the biased range -127 to 128
- Any numbers we want to represent in this biased range we must add the bias 127 to → if we want to represent 2 in binary with the biased system, we instead represent 2 + 127 = 129 in binary
- The exponent uses this bias system

IEEE has a special way to represent 0, +/- infinity and NaN (Not a Number)

- Infinity is represented by setting all bits of the exponent to 1 and all bits of the mantissa to 0 → 0 1111 1111 0000 0000 0000… means +infinity, and if the first bit was a 1, it would be -infinity
    - 1.0 / 0.0 gives +infinity, and -1.0 / 0.0 gives -infinity
- 0.0 / 0.0 or $\sqrt{-2}$ would both produce NaN → we set all 8 bits of the exponent to 1, then have exactly 1 bit of the mantissa to be set and the rest to be 0
    - Two NaN results are not equal to each other
    - math.h has a function isnan(x) that will return 1 if x is NaN and 0 if x is not NaN
    - printf("%lf", x) where x is NaN will print nan
    - NaN can be positive or negative depending on what you set the sign bit to be
- zero is represented by setting all bits of the exponent and mantissa to 0
    - The sign bit determines whether we have +0 or -0, and +0 == -0

Code

Lec 5.2 – Files

Concepts

The operating system effectively provides a virtual machine (that is much simpler than a real machine) to each user that allows them to interact with the hardware

- The virtual machine interface can stay the same across different hardware, making it much easier for users to write portable code that works across different machines regardless of hardware
- The O/S requires hardware to provide some things
  - Privileged mode → code running in privileged mode (e.g. O/S code) can access all hardware and memory without limit
  - Non-privileged mode → code running in non-privileged mode (e.g. user code) can not access hardware directly, and has limited access to memory, but this mode provides mechanisms allowing code to make requests to the O/S (system calls), and thereby to hardware
  -

System calls transfer execution to O/S code, which is in privileged mode, and the O/S usually returns execution back to user code when the request is done

- O/S first checks that the operation is valid and permitted before executing it
- Common types of system call operations
  - Reading or writing bytes to a file
  - Requesting more memory
  - Creating and terminating a process (running and end a program)
  - Sending information via a network
- Often, library functions make system calls for us
- Different O/Ss have different system calls

Important Unix system calls → <unistd.h>

- 0 – read → reads a specified number of bytes from a file descriptor (the index of the currently opened file)
  -
- 1 – write → write some bytes to a file descriptor
- 2 – open → open a file system object, and return a file descriptor
  - Creates an entry in the file descriptor table, and returns the index of that entry (the file descriptor/file descriptor number)
- 3 – close → stop using a file descriptor
- 4 – stat → get file system metadata for a pathname
- 8 – lseek → move file descriptor to a specified offset within a file

File descriptors are small integers that index to a per-process array maintained by the O/S

- The above system calls manipulate files as a stream of bytes accessed via a file descriptor
- On Unix-like systems, files are just sequences (arrays) of 0 or more bytes

    - The bytes associated with a file have no meaning → we have 'data' but data is meaningless without context → e.g. if we have 4 bytes, we have no idea what those 4 bytes are since they could be signed or unsigned integers, or an array of 4 ASCII chars, or floating points
    - The 'file' command on the terminal can usually guess what type of file something is by looking at the individual bytes of that file
- The file descriptor table is a big array of structs that every process on a computer has
    - The structs keep track of which file we just opened, and where we are up to inside our file (our cursor)
    - The cursor keeps track of where we are reading or writing to in the file we have opened
-

There are some special files that are always already in the file descriptor table

- fd 0: stdin → where program receives input
- fd 1: stdout → where we usually write messages to → fputs("string", stdout)
- fd 2: stderr → where we write error messages to

EACH MACHINE HAS DIFFERENT SYSCALL NUMBERS, SO WE DON'T USE THE SYSCALL FUNCTION → we use the Unix system calls (<unistd.h> library functions) that will use the machine's syscalls for us

- We also have C-library functions equivalent to the Unix syscalls → open becomes fopen, write becomes fwrite, read becomes fread, and close becomes fclose → these are in man 3
- HOWEVER, the inputs may differ a bit between the C library and unistd library

NOTE: man 2 syscall_name → terminal command that tells you about syscall_name

EXAMPLES:

- cp_fgetc.c

Code

open(fileName, flag, mode)

- Opens the file called filename and does something
- The something it does is specified by the flag → e.g. read, write, create → flags can go into specifics, like whether to write over bytes in the file or at the end of the file
    - We can use multiple flags by combining them using | operator
    - The mode
- Open creates an entry in the fd table, then returns the fd (the index of that entry in the fd table)

read(fd, array, num_bytes)

- Transfers control to the O/S, asking it to read num_bytes bytes from the file indexed at fd (file descriptor number) in the file descriptor table, and to read those bytes into the array
- This moves the cursor along num_bytes, so that the next time we read from the file, we read from where we left off, not where we started

write(fd, *string, num_bytes)

- Asks O/S to write num_bytes bytes from the start of the string *string points to into the file referred to by (indexed at) fd
- NOTE: we don't need to write '\0' to files, so num_bytes can just be the number of visible chars and whitespace chars

close(fd)

- Removes the entry referred to by fd from the file descriptor table, closing it
- Just like with malloc and free, every open should have a close

perror("string")

- Prints the input string, then the appropriate error message depending on errno

Lec 7.1 – Library Functions

Concepts

<stdio.h> is a part of the standard C library that provides a higher level API (application programming interface) to manipulate files

- They are available in every C implementation that can do I/O (input/output)
- <stdio.h> functions are portable, convenient and efficient, and should be used unless there is a good reason not to → e.g. writing programs with special I/O requirements like a database implementation
- On Unix-like systems, stdio.h functions will call open, read and write but with buffering for efficiency

stdio buffering

- We assume stdio buffering size (BUFSIZ) is 4096
- Buffering makes programs/functions more efficient as multiple syscalls are somewhat costly in terms of performance and efficiency
- For reading:
    - The first fgetc we call requests 4096 bytes by using the read syscall for us, and stores those bytes inside the input buffer array FILE * points to
    - The next 4095 fgetc calls return a byte from the input buffer instead of doing a syscall again
    - Then it repeats on the 4097th fgetc call which requests another 4096 bytes
- For writing:
    - The first 4095 fputc calls put bytes into the output buffer array
    - The 4096th fputc call calls write to write (flush) all 4096 bytes in the output buffer to the output file/stream
    - Then it repeats
    - NOTE: the fflush() function call force empties the output buffer array, and the newline char '\n', scanf function call and exit/return actually flush the output buffer for us
- We need to make sure output is flushed so that content is written to the file before the program terminates
- Basically, we allocate large amounts of memory at once then read/write to that memory until we run out
    - If we run out while reading, we ask for more memory
    - If we run out while writing, we flush the allocated memory then ask for more?

File systems manage persistent stored data (e.g. data on a magnetic disk or SSD)

- On unix-like systems, directories are objects containing zero or more files or directories → they are sets of files or directories
- File systems also maintain metadata (other data about the file besides the data in the file) for files and directories → e.g. permissions, last modified time, file type, file size
- Filenames are sequences of 1 or more bytes

- Can contain any byte except 0x00 (ASCII '\0') and 0x2F (ASCII '/' used to separate components of pathnames)
- Maximum filename length is usually 255 bytes, but depends on file system
- Also can't name files . (current directory) and .. (parent directory)
- Some programs (shell, ls) treat filenames starting with . specially

./

- ./prog tells our computer to access the file in our current directory called prog
- If we just do /directory/prog, our computer knows to access the file called prog even without the ./ since it is following the path to directory

Files and directories can be accessed via pathnames

- Absolute pathnames start with a leading / and give a full path from root
  - /cs1521/public_html/, and /cs1521/lecs/lec07
- Every process (running program) has a current working directory (CWD)
  - The pwd (print working directory) terminal command gives the absolute path to the current working directory
- Relative pathnames do not start with a leading /
  - They are appended to the CWD
  - E.g. if my CWD is /cs1521 and I enter cd lecs/lec07, lecs/lec07 is the relative pathname and the cd command will append that to my CWD and change into the /cs1521/lecs/lec07 CWD
- Unix-like file systems are actually tree-like, and the absolute pathname is the path from the root of the tree to the CWD → BUT they are not trees because of symbolic links (
  - Symbolic links are shortcuts that let you jump from one path to another without backtracking → makes it a graph, not a tree
  - Files are leaf nodes

Symbolic links are shortcuts between 'branches' of file systems that let you jump from one branch to another without backtracking

- This makes unix-like file systems only tree-like and technically graphs to some degree

ln -s relative/path/name portal

- Makes a symbolic link called portal from the CWD to CWD/relative/path/name

ls – l

- Prints out all files/directories in the CWD with additional metadata
  - Permissions (the weird letter combo)
    - x → execute
    - r → read
    - w → write
    - - → does not have the permission that should be in that spot
    - The first char tells us whether it is a directory (d) or a file (-) or a character device (c) or a symbolic link (l), or etc.
    - First set of rwx (three chars) after the first char is the owner's permissions

        - Second set of rwx (three chars) is the group's permissions
        - Third set of rwx is the public (everyone else's) permissions
    - User ownership
    - Group associated with file
    - File size
    - Last modified time and date
    - Location in computer???

The echo terminal command reads in input then spits it straight out to stdout by default

- echo fuck > file will read in fuck as input then output it to file instead of stdout

File systems are used to access:

- Files
- Directories (folders)
  storage devices (disks, SSDs)
- Peripherals (keyboard, mouse, USB)
- System information
- Inter-process communication
- Network
- A whole bunch more
-
- NOTE: all of these things are considered files by Unix-like O/Ss

NOTE: running ./prog blah 2> error.log where there should be a file instead of blah will redirect any outputs to stderr stream into a new file called error.log

- Just using > without 2 will redirect stdout to some file we specify instead of error.log

EXAMPLES:

- cp_stdio.c
- hello_stdio.c
- buffering.c
- last_byte.c
-

Code

FILE *fopen(const char *pathname, const char *mode)

- fopen is the stdio.h equivalent of open()
- pathname is the name of the file we want to open
- mode is a string of 1 or more characters including
    - 'r' → open a text file for reading
    - 'w' → open a text file for writing
        - Truncates a file to zero length (discards its contents to make it an empty file) if it exists and then writes into it
        - Creates a file and writes into it if pathname doesn't exist
    - 'a' → open a text file for writing

        - Appends what is written to the end of pathname if pathname exists
        - Creates a file and writes into that file if pathname doesn't exist
- fopen returns a FILE * pointer

fgetc(FILE *stream)
- Reads a byte from stream file
- getchar() is just fgetc(stdin)

fputc(int c, FILE *stream)
- Writes a byte c to stream file
- putchar(int c) is just fputs(c, stdout)

fputs and fgets
- Writes and reads strings
- Knows it is writing/reading a char array (string) and will NULL-terminate for us
- fputs doesn't need to know the size it is writing out to stdout since strings are null-terminated
- puts(char *s) is just fputs(s, stdout)

fscanf(FILE *stream, const char *format, …) and fprintf(FILE *stream, const char *format)
- Reads/writes formatted input into stream
- *format is the string ("I wanna sleep for %d hours", num_hours) we want to write or string of format specifiers ("%d %c %lf, etc.") telling us what to read in
- … is for the extra shit → for fscanf it is where we want to read into (&num_hours), and for fprintf it is what we want to put into the format specifiers ("", num_hours)
- scanf(char *format, …) is just fscanf(stdin, format, …)
- printf(char *format, …) is just fprintf(stdout, format, …)

fprintf(FILE *stream, const char *format, …)
- Writes formatted output to stream

fread (void *ptr, size_t size, size_t nmemb, FILE *stream) and fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
- Read/write an array of bytes into/to stream
- Usually better to use fgetc/fputc in a loop
- Void *ptr is a pointer to an array containing nmemb elements of size size
- We use fwrite/fread (or fgetc/fputc in a loop) for binary data, since binary data may contain zero bytes and cause issues with fputs/fgets and fscanf/fprintf → e.g. when reading or writing a jpg

fflush(FILE *stream)
- Flushes everything currently in the output buffer out to stream if stream is an output stream → writes all data in the output buffer
- Will rarely ever need to use fflush besides error handling

fopen(FILE *pathname)

- Opens a file for us → uses malloc to store a struct FILE that contains the location of pathname's entry in the fd table and the position of our cursor in pathname

fclose(FILE *pathname)

- Closes pathname for us
- Calls free for us to free the memory allocated in fopen (which uses malloc for us to create an entry in the fd table)
- Also flushes the output buffer for us when called

lseek(int fd, off_t offset, int whence)

- fd is the file descriptor number
- offset is the offset of the pointer (cursor) in the file
- whence is a directive
    - SEEK_SET → the file offset (cursor) is set to 'offset' bytes
    - SEEK_CUR → the file offset is set to its current location plus 'offset' bytes
    - SEEK_END → the file offset is set to the size of the file plus 'offset' bytes
- Much more efficient than traversing through each byte in a file
- THIS IS A SYSCALL → needs <unistd.h>

fseek(FILE *pathname, long offset, int whence)

- C library equivalent of lseek
- Whence is the same directives as those for lseek
- Changes your position to an offset amount from whence

long ftell(FILE *stream)

- Returns a long telling you how many bytes into a file you currently are

Lec 7.2 – File Metadata

Concepts

Metadata for file system objects is stored in inodes, which hold:

- The location of file contents in file systems
- File type
- File size in bytes
- File ownership
- File access permissions
- Timestamps → file creation, and last modification and access times

ls -l displays the metadata of files in your CWD

unix-like file systems effectively have a large array of inodes that contain your files metadata

- An inode's index in this array is its inode-number (i-number), which is unique to a file in a filesystem
- Directories are effectively a list of (name, inumber) pairs
- ls -i prints inode-numbers of each file
- NOTE: sometimes very small files are stored in the inode to improve performance

Accessing files by name (roughly):

1. Open directory and scan for name
2. If not found, "No such file or directory"
3. If found as (name, inumber), access inode table inodes[inumber]
4. Collect file metadata and
    a. Check file access permissions given to the current user/group → if insufficient permissions, "permission denied"
    b. Collect information about file's location and size
    c. Update last access timestamp
5. Use data in inode to access file contents

Permissions rwx

- rwx → read, write, execute → a – in the position of a permission means they don't have that permission
    - Behind the scenes, if you have a permission there is a 1, and if you don't there is a 0 → it is just a bitfield

	User			Group			Other		
	r	w	x	r	w	x	r	w	x
	4	2	1	4	2	1	4	2	1
	1	1	1	1	0	1	0	0	0
Octal number	7			5			0		

    - In base 2, the largest number we can create from 3 bits is 7 → we can represent each of the rwx groups using an octal number

    - chmod 750 prog → terminal command that changes the permissions bitfield to fit 750, where each digit is the octal number of the rwx groupings

- ls – l first column of info is the file type followed by permissions for the owner, group and public
- Directories can have execute permissions which just mean those with the execute permission can enter into the directory
- Non-compiled files (prog.c) do not have execute permissions because you cannot execute them
- The st_mode field of the stat struct is a bitfield composed using a bitwise-or | of defined bitmasks → 3 bits for file type, then an empty bit, then 3 bits for permissions
    - Bitmasks at 20:56 of lec 11a → man 7 inode

NOTE: Every directory has an entry for the current directory (.) and for the parent directory (..)

- If traversing directories recursivel, make sure you don't recurse through the parent or current directories or you will encounter an infinite recursion

NOTE: any file path (pathname) input into library functions are evaluated as relative paths to the CWD

Examples:

- stat.c
- chmod.c
- chang_mode_to_640.c
- mkdir.c
- getcwd.c → getcwd and chdir implemented

Code

stat(const char *pathname, struct stat *statbuff)

- Returns metadata associated with pathname in statbuf
    - inode number
    - File type (file, directory, symbolic link, device)
    - Size of file in bytes (if it is a file)
    - Permissions (rwx)
    - Times of last access/modification/status-change
    - Lots more, but most of the other stuff is not relevant to us 7:59 of lec 11a 23T2 shows the actual struct
- Returns -1 and sets errno if metadata is not accessible

fstat(int fd, struct stat *statbuf

- Same as stat, but gets data through an open file descriptor

lstat(const char *pathname, struct stat *statbuf)

- Same as stat, but doesn't follow symbolic links

chmod (const char *pathname, mode_t mode)

- Needs <sys/stat.h>
- mode is the three digit octal code we want to change the permissions for pathname to match

opendir(const char *name)

- Needs <sys/types.h> and <dirent.h>
- Opens a directory given a path (name) to it, and returns a 'directory stream' DIR, which lets us read and see what is in the directory

DIR *readir(DIR *dirp)

- Needs <dirent.h>
- dirp is a pointer to a directory entry (file in a directory) in the directory stream
- Returns a pointer to the next entry (file) in the directory dirp → to a dirent struct in dirp that represents the next entry
- Returns a pointer to the next entry (file) in directory dirp on success, otherwise returns NULL if there is no next entry (end of directory) or if an error occurred

closedir(const char *name)

- Closes the directory, freeing shit malloced by opendir

mkdir (const char *pathname, mode_t mode)

- Creates a new directory called pathname with permissions mode
- If pathname is a/b/c/d, directories a, b and c must exist, you must have write permissions for c, and d must not already exist
    - We don't need to actually input pathname as a/b/c/d, just entering a string will append it to the CWD
    - Needs a, b and c to exist since we can only create one directory at a time
- The new directory must contain two initial entries
    - . → a reference to itself
    - .. → a reference to its parent directory
- Returns 0 if successful, otherwise it returns -1 and sets errno

unlink (char *oldpath)

- Removes a file/directory
- Rarely used in 1521

rename(char *oldpath, char *newpath)

- Renames a file/directory oldpath to newpath
- Unlikely to use in 1521

int chdir(char *path)

- Changes the current working directory to path
- Returns 0 on success, otherwise sets errno and returns -1

char *getcwd(char *buf, size_t size)

- Gets the current working directory and stores it in the string buf

link (char *oldpath, char *newpath)

- Creates a hard link to a file
- Not used in 1521

symlink(char *target, char *linkpath)

- Creates a symbolic link from target to linkpath
- Unlikely to use in 1521

Lec 7.3 – Text Encoding

Concepts

In C, text refers to strings, which are just arrays of characters

- Once we can encode characters, we can create strings to represent text
- Modern computers use UNICODE to represent text
- Text encoding is done by a lookup table, NOT a mathematical expression

ASCII

- All common 'American' symbols can be represented as 7-bit encodings → 127 possible symbols and control sequences (for teletypewriters → NUL, LF)
- Columns in ASCII table are called sticks, and symbols in each stick serve some common purpose

UNICODE

- MAIN FOCUS IN 1521 TEXT ENCODING
- The goal of Unicode is to create a single unified encoding that can represent all characters in the world → currently has 149,251 char representations
- Unicode is very large, so it needs a very structured layout → the Unicode Standard defines a codespace (the encoding) that ranges from 0x0000 to 0x10FFFF where each hex value represents a code point (character/symbol)
- Unicode codespace has 1,114,112 code points, and only 293,168 are assigned
- These code points are split into 17 planes, and within each plane the code points are split into blocks → 41:12 lecture 12 → planes 4-13 are unassigned (unused)
    - Blocks come in multiples of 16 (usually 128) and are used to roughly group characters by their purpose → 42:50
    - First block of 0^{th} plane is just ASCII

UNICODE characters also have a major and minor category

- Major → letters, mark, number, punctuation, symbol, separator, other
    - 90% of Unicode chars are classified as other
-

Storing UNICODE characters → 1:00:14

- Code points range from 0x0000 to 0x10FFFF, so we need at least 21 bits → UTF-32 is a fixed width encoding that uses 32 bits to represent each character
- A UNICODE code point is simply stored in 32 bits (convert hex code point to binary and store)
- U+ prefix denotes the representation of a raw UNICODE code point, which is always at least 4 hex digits
    - A last hex digit represents the plane number of the code point → no last plane digit if in 0^{th} plane
    - We add leading 0s in the UTF-32

UTF-8 → 1:02:00

- UTF-32 is very wasteful with the leading 0s → memory expensive
- Most characters used commonly are in the first block of the 0th plane (ASCII), which only need 7 bits and are near the beginning
- Thus, we usually use UTF-8 when transferring data and not UTF-32
- C uses UTF-8
- UTF-8 is a variable-width encoding → a single UTF-8 character can be anywhere from 1 to 4 bytes long
    - All ASCII characters can be represented in 1 byte without any wasted 0 bits
    - All of BMP (0th plane) can be represented in 3 bytes → 8 bits more efficient than UTF-32
    - Worst case, all 4 bytes are used making it the same as UTF-32
- The first bits of the first byte tell us how many bytes are used to store the UTF-8 character
    - All other bytes start with a 10 to signify that they are not the first byte

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx	-	-	-
2	11	110xxxxx	10xxxxxx	-	-
3	16	1110xxxx	10xxxxxx	10xxxxxx	-
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Converting UTF-32 to UTF-8

- Example at 1:04:40

1. Get the UTF-32 representation and convert it to 32 bit binary
2. Get rid of all leading 0s in the 32 bit binary
3. Figure out how many bytes to use based on the length of the binary
4. Align the binary with the UTF-8 layout's Xs, filling any remaining Xs with leading 0s → using bitwise OR |
5. Convert that binary into hex to get the UTF-8 encoding

NOTE: on our terminal, unicode -s "string" --max 0 will tell us all the Unicode characters in string

Code

EXAMPLES → cs1521/lecs/lec07:

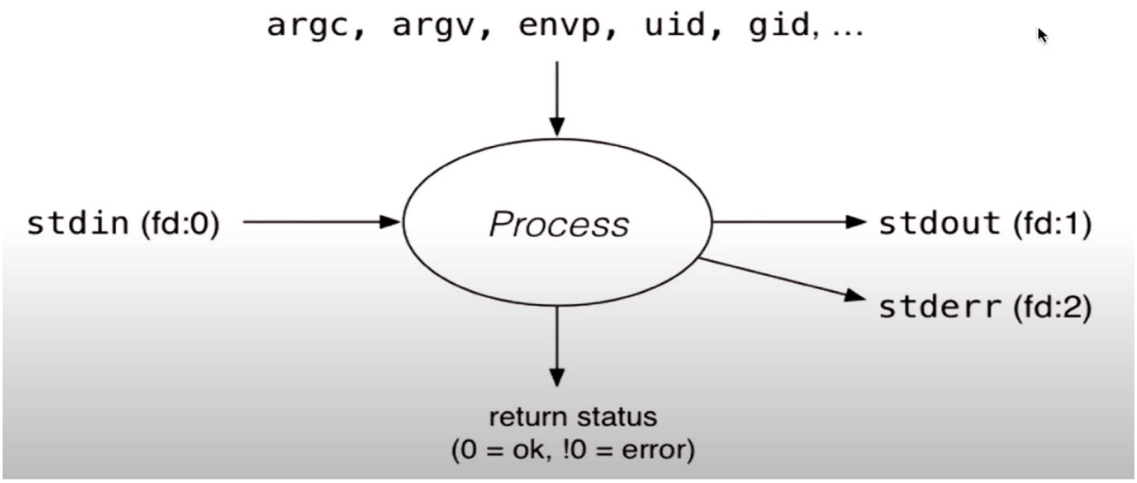
- count_UNICODE.c

Lec 8.1 – Processes

Concepts

A process is a program executing in an environment

- The OS manages processes
- On Unix/Linux systems, processes have (IDFK)



A process is an instance of an executing program

- Each process has an execution state defined by
  - Current values of CPU registers
  - Current contents of its memory
  - Information about open files (and other results of syscalls)
- On Unix/Linux
  - Each process has a unique process ID (PID), which is a positive integer of type pid_t → defined in unistd.h
  - Low numbered processes are USUALLY system-related and started at boot
  - Some parts of the OS may appear to run as processes

Each process has a parent process

- This is the process that created the process
- If a process' parent terminates, its parent becomes init (PID 1)
- Unix provides some commands to manipulate processes
  - sh → creating processes via object-file name
  - ps → showing process information
  - w → showing per-user process information
  - top → showing high-cpu-usage process information
  - kill → sending a signal to a process
- Normally a parent process waits for the child process to finish before doing shit like terminating, but if it doesn't, we need to use waitpid

Preemption is the act of temporarily interrupting the execution of a task with the intention of resuming it later

- On pre-emption a context switch occurs → the process's entire state is saved and the new process's state is restored → very expensive
- Processes will be pre-empted if
    - It ran long enough and the OS replaces it by a waiting process
    - It needs to wait for input, output or some other operation
- The 'scheduler' decides which process to run next → it attempts to
    - Fairly share the CPU among competing processes
    - Minimise response delays (lag) for interactive users
    - Meet other real-time requirements
    - Minimise the number of expensive context switches

Making processes

- The old way was to use fork() then exec()
    - fork() → duplicates the current parent process to make an identical child process
    - exec() → overwrites the current process (run by child process)???
    - BUT this is very bug prone and is low security
- The new standard way is to use posix_spawn()
    -

Environment variables are strings that have the form name=value, and are stored in an array

- An environment variable is one string that can be thought of as two strings separated by an '=' → name and value pairs
- The final element of the array is a NULL pointer
- The global variable environ is how we access environment variables
    - Though we can also pass it in as another parameter to main → char *env[]
- SHOULD use getenv and setenv to access environment variables instead of indexing into the array ourselves

- Almost all programs pass the environment variables they are given to any programs they run
- Environment variables also provide us with a simple mechanism to pass settings to all programs → e.g. setting stuff like the timezone (TZ) or preferred language (LANG)

Pipes are one-directional byte streams provided by the OS

- All pipes have a read end and a write end
    - Bytes written to the write end can be read from the read end
- 50:45 LEC 14
- SORT OF EXTRA SHIT FOR 1521

FUCK IDK:

- 35:25 lec 13 → searching environment variable and shit
- echo $envName → terminal command prints the value of an anvironment variable called envName
- 50:19 → get_status code → posix error checking n some other shit

- /cs1521/lecs/lec07/ls_spawn.c → code using posix_spawn and waitpid with the error checking necessary
- /cs1521/lecs/lec07/date_with_pipe.c → code using a pipe
- /cs1521/lecs/lec07/date_with_popen.c → code using popen
- date_with_spawn.c → EXTENSION, DW ABT IT

Code

posix_spawn(pid_t *pid, const char *path, const posix_spawn_file_actions_t *file_actions, const posix_spawnattr_t *attrp, char *const argv[], char *const envp[])

- pid → a pointer to a process id → use this to get the process id of a new program after calling posix_spawn?
- path → path to the program we want to run
- file_actions → specifies actions to be performed on files before running program → e.g. redirecting stdin/stdout to a file?
- attrp → specifies what attributes for new process → NOT IN CS1521
- argv[] → array of arguments to pass to a new program
- envp[] → array of environment variables to pass to a new program

- posix_spawn() creates a new child process that executes a program specified by path
  - Requires <spawn.h>
  - If the new process was successfully made, posix_spawn returns 0 and sets pid to point to the process id of the new child process → otherwise an error number is returned and pid is set to NULL
    - Need to set errno = posix_spawn_ret
- If we want the usual file actions and attributes, just input NULL for both
  - posix_spawn(pid, path, NULL, NULL, argv[], envp[])

char *getenv(const char *name)

- Searches the environment for a variable called name, and returns a pointer to the corresponding value string
- Returns NULL if it can't find the variable called name

int setenv(const char *name, const char *value, int overwrite)

- Searches the environment for a variable called name
- If a variable called name is found and overwrite != 0, the value of name is changed to the input value → if overwrite == 0, nothing happens
- If no variable called name is found, create a new environment variable called name with the input value
- Success returns 0, an error returns -1 and sets errno

pid_t getpid()

- Gets and returns the process ID of the process that calls it
- Getppid does this for the parent of the process that calls it, and getpgid does this for the process group (processes live in groups)

FORK AND EXEC FUNCTIONS IDFK

- OBSOLETE SO DON'T USE THIS SHIT???

int execvp(const char *file, char *const argv[])

- Runs another program in place of the current process
- file → an executable program?
- argv[] → arguments to pass into the new program
- Most of the current process is re-initialised → a new address space is created and all variables from the current process are lost → open file descriptors remain open and unchanged
- PID is unchanged since we replace stuff???
- If there is an error, -1 is returned and errno is set, else it doesn't return??

exit(int status)

- Terminates the current process after cleaning memory and flushing buffers
- Stdlib.h provides the EXIT_FAILURE = 1 and EXIT_SUCCESS = 0, which we can pass in to say if a program/process was successful or not

pid_t waitpid(pid_t pid, int *wstatus, int options)

- Pauses the current parent process until the child process with PID pid changes state → state changes include finishing, stopping, re-starting
- Ensures that child resources are released on exit
- Special pid values → return values?
  - pid == -1 → wait on any child process
  - pid == 0 → wait on any child in process group
  - pid > 0 → wait on specified process to change state
- wstatus is a pointer to an int that gives us information about what actually happens with the process → e.g. stores its exit status, return value, if and how the process crashed
- options provide variations in waitpid() behaviour
  - 0 just waits for the child process to finish → default
  - WNOHANG → returns immediately if no child has exited
  - WCONTINUED → return if a stopped child has been restarted
  - WNOHANG and WCONTINUED are constants
- Requires <sys/wait.h>
  - Use WIFEXITED and WEXITSTATUS macros to get information from wstatus

int system(const char *command)

- Uses fork to create a child process that executes the shell command specified using exec in /bin/sh, then returns
  - Returns -1 and sets errno if there was an error
- ONLY USE FOR DEBUGGING
  - Prone to security exploits and can break easily → don't use in code that handles untrusted input or needs to be reliable

int pipe(int array[2])

- Places a file descriptor into each slot of the array to create a pipe

    - [0] is read end
    - [1] is write end
- Returns 0 if successful, -1 otherwise

FILE *popen(const char *command, const char *type)

- command is a shell command (ls, date, etc)
- type is “r” or “w” specifying if it is the read or write end of the pipe
    - if “r”, we read the output of command and write it to the write end of the parent process (e.g. stdout)
    - if “w”, we write the output of command to the read end of the parent process (e.g. stdin)
- Opens a process by creating a pipe
- The FILE opened with popen must be closed with pclose

Lec 8.2 – Concurrency, Parallelism, Threads

Concepts

Concurrency → illusion that our computer is multi-tasking

- The CPU executes processes 'concurrently' by rapidly switching between multiple processes → in actuality, a CPU can only run one process at a given instance in time, but just switches so quickly it looks like its doing multiple things at once
- CPUs will run one process for a very short instance (milli or micro seconds) then switch to run another and switch back or to another
    - NOTE: switching between processes also takes a very short amount of time → context switches?
- Only one core in CPU → cores are needed to execute instructions

Parallelism → computer actually multi-tasking

- Multiple cores in CPU, so multiple instructions can be run at once
- Divides the processes that need to be run across different cores so that multiple processes can truly be executed simultaneously
- Sort of like a subset of concurrency???
- Parallelism can also occur between multiple computers → allows division of tasks to reduce time taken

Flynn's Taxonomy → common classifications of types of parallelism

- SISD → Single Instruction, Single Data → No parallelism
    - E.g. mips code → computer runs one instruction at a time, line by line
- *SIMD* → Single Instruction, Multiple Data → Vector processing
    - Multiple cores of a CPU executing parts of the same instruction
    - BEYOND 1521!
- MISD → Multiple Instruction, Single Data → Pipelining
    - Data flows through multiple instructions?
    - Very rare in the real world
- *MIMD* → Multiple Instruction, Multiple Data → Multi-processing
    - Multiple cores of a CPU executing different instructions

NOTE: both parallelism and concurrency need to deal with synchronisation, which is

Parallelism across processes

- One method is to create multiple processes that each do part of a job
    - Child executes concurrently with parent in its own address space
    - Child inherits some state information from parent (e.g. open fds)
- Disadvantage of this method
    - Multiple processes created
        - Switching between them all can be very expensive
        - Requires a lot of memory
    - Communication between processes may be limited and/or slow
- BUT, having separate processes in separate address spaces makes the processes more robust (cybersecurity safe stuff and won't overwrite data for other processes)

Lec 9.1 – Concurrency, Parallelism, Threads Cont'd

Concepts

Threads allow parallelism within a process

- Each thread has its own execution state (Thread Control Block → TCB)
- Threads within a process share address space:
  - They share code → functions
  - They share global/static variables
  - They share the heap (mallocs and frees)
  - BUT, they do not share the stack, meaning local variables are not shared
  - This makes it easy for threads to communicate, while also not worrying about local variables colliding and clobbering
- Threads within a process share file descriptors and signals
- Threads allow simultaneous (concurrent) execution
  - Using multiple threads for a single task can reduce the time taken at the cost of CPU usage

NOTE: the CPU scheduler decides which thread to execute at any given time

- This is non-deterministic → we have no clue how it chooses, and it doesn't always choose the same way
- Operations that only require one instruction (e.g. adding) are called atomic operations → non-atomic operations require multiple instructions causing potential problems with CPU switching → example at 1:15:00 lec 15
  - Race condition → when multiple threads try to access the exact same resources causing a loss of data? → e.g. two threads trying to access one global variable → Mutexes solve this
- We can create 'critical sections' to avoid this problem → critical sections are sections of code that must be executed together → tells CPU not to switch while executing code/instruction in this section
  - We want critical sections to be as small as possible since entering a critical section pauses all other threads that need to access the same resource, slowing down a program

Another way to avoid race condition problems is to use some of the provided atomic instructions in <stdatomic.h> on atomic data types

- atomic_int is an atomic integer data type we can use atomic instructions on
- If we increment or decrement or do whatever to a global variable that is an atomic type using normal C (e.g. global++), C knows it is atomic type and does an atomic operation
- HOWEVER, atomic instructions require specialised hardware → compiling a program with stdatomic will warn you if you don't have this hardware
- Although atomic instructions are faster and simpler than traditional locking, there is a performance penalty with using them → increases program complexity

Other issues with thread concurrency

- Data races → thread behaviour depends on unpredictable ordering → can potentially produce difficult bugs or security vulnerabilities
- Deadlock → multiple threads stopped because they are all waiting on each other to unlock a resource
    - Explanation at 1:30:00 lec 15 → all locks for the same resources should be done in the same order, and unlocks should be done in reverse order like with popping and pushing → global ordering of lock acquisition
- Livelock → threads running without making any progress
    - Not rly 1521
- Starvation → some threads never getting to run
- MORE IN 3231

EXAMPLES → /cs1521/lecs/lec09

- five_calls.c → function pointers in C
- two_threads.c → pthread_create and pthread_join → shows us how the CPU will choose which thread it wants to execute at any given time
- thread_sum.c → demonstrates how usage of multiple threads allows for parallel processing that can speed up a task → note some inaccuracy from using doubles

Code

NOTE: The following code requires the <pthread.h> library

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(thread_main)(void *), void *arg)

- thread is a pointer to the address of the variable we want to store the spawned thread's 'ID' in
- attr we just put NULL in 1521
- the thread_main thing is a function pointer → allows us to pass functions into other functions → sort of like passing in a label to the start of a function in MIPS
    - void * is the return type of the function the function pointer points to
    - thread_main is the name of the function pointer
    - (void *) indicates the arguments we need to pass in to the function
    - void *arg is where we pass in the parameters we want to input into the function pointed to
-
- Returns 0 on success, otherwise returns -1 and sets errno

int pthread_join(pthread_t thread, void **retval)

- Basically waitpid but for threads → waits until thread terminates before main terminates
- retval is the return/exit value of thread

void pthread_exit(void *retval)

- Basically exit →terminates the execution of the current thread and frees its resources
- retval is the return/exit value of the thread terminated

int pthread_mutex_lock (pthread_mutex_t *mutex)

- Locks an unlocked mutex given a pointer to that mutex
- mutex is just some variable we create to say if something is locked or not → we typically name it using the name of the resource we want to lock followed by lock
- For a particular mutex, only one thread can be running between a mutex lock and unlock → other threads attempting to mutex lock will have to wait until the first thread executes mutex unlock
- Mutex is a global variable also. We initialise it to PTHREAD_MUTEX_INITIALIZER, which is defined in <pthread.h> → ensures mutex begins as unlocked

int pthread_mutex_unlock (pthread_mutex_t *mutex)

- Unlocks a locked mutex

<stdatomic.h>

- The following code does an atomic operation on a global variable given a pointer to that global variable → parameters are (atomic data *data, int value)
- fetch_add → data += value
- fetch_sub → data -= value
- fetch_and → data &= value
- fetch_or → data |= value
- fetch_xor → data ^= value

Lec 9.2 – Concurrency, Parallelism, Threads Cont'd

Concepts

Another issue with threads is data lifetime

- When sharing data with a thread, we can only pass the address of our data → but what if by the time the thread reads the data, that data no longer exists?
- This may occur when we have a thread that creates a thread which uses a local variable declared in the first thread, but then the stack frame for the first thread closes, destroying the local variable → EXPLANATION AT 12:20 lec 16
- Basically, if the lifetime of a variable is shorter than the lifetime of a thread using that variable, we have a problem
    - A solution is to use malloc to put the local variable on the heap, so that it won't be destroyed when the stack frame closes, but we need to be able to free it once we are done with it to avoid memory leaks
    - Another solution is to force both the calling thread and the newly created thread to wait for each other
        - The calling thread shouldn't proceed until the new thread has read the data, and the new thread shouldn't proceed too far before letting the calling thread keep moving (else it could stall performance)
        - We can implement this cross-thread waiting with barriers

Barriers → ADVANCED TOPIC

- 22:18 lec 16
- We create a barrier, then pass that barrier into the new thread → when one thread reaches the barrier, it stops until the other thread also reaches the barrier, at which point they both proceed
    - We make the child thread create copy of whatever data might die before the child thread can execute then we reach the barrier, allowing the calling thread to terminate and close its stack frame
- Synchronises threads until all threads have a copy of the necessary data, then all thread proceed in 'parallel'

Semaphore → ADVANCED TOPIC → NON-ACCESSIBLE

- 29:00 lec 16
- Semaphores are a more general synchronisation mechanism than mutexes
- We can allow a specified number of threads to access a resource at any given time

Code

pthread_barrier_init(barrier_t *barrier, const pthread_barrierattr_t *restrict attr, int count)

- Creates a barrier that only releases after count threads wait on the barrier
- For attr we jus tput NULL

pthread_barrier_wait(barrier_t *barrier)

- Waits at barrier until the required number of threads are waiting

int sem_init(sem_t *sem, int pshared, int value)

- sem is the address of the semaphore
- pshared is an option setting
    - 0 → semaphore is to be shared between the threads of a process
    - Non-zero → semaphore is to be shared between processes
- value is the number of threads we want to allow access to the resource at any given time
- returns 0 on success, otherwise returns -1 and sets errno

int sem_wait(sem_t *sem)

- AKA P
- If sem > 0, decrements sem by 1 then proceeds, otherwise waits until sem > 0
- returns 0 on success, otherwise returns -1 and sets errno

int sem_post(sem_t *sem)

- AKA V
- increments sem by 1
- Every sem_post should follow a sem_wait
- returns 0 on success, otherwise returns -1 and sets errno

Lec 9.2 + 10.1 – Virtual Memory

Concepts

Code

Lec x.y

Concepts

https://cgi.cse.unsw.edu.au/~cs1521/23T2/lectures/
Code