COMP1521 – Course Notes

Contents

Course Intro

- Course Convenor/Lecturer **Andrew Taylor** cs1511@cse.unsw.edu.au
- Course format:
    - Lectures (Wednesday 3 pm – 5 pm, Friday 9 am – 11 am)
    - Tutorial (Thursday 3 pm – 4 pm)
    - Lab (Thursday 4 pm – 6 pm)
- Major themes …
    - Software components of modern computer systems
    - How C programs execute (at the machine level)
    - How to write (MIPS) assembly language
    - Unix/Linux system-level programming
    - How operating systems and networks are structured
    - Introduction to concurrency, concurrent programming
- Goal: to understand execution of software in detail
- Compilers: `dcc` on CSE machines (clang or gcc elsewhere)
- Assembly language: MIPS on `QtSpim` (also `Xspim` on CSE)
- Weighting:
    - Labs (15%)
    - Weekly Tests (best 6 out of 8 – 10%)
    - Assignment 1 in Assembly Language (15%)
    - Assignment 2 in C (15%)
    - 3 hour Online Exam (must score 18+/45 in exam to pass the course - 45%)

Integers

Binary

- Interpret binary number 1011 as: $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- The base or radix is 2, digits 0 and 1
- Write number as 1011_2 (= 11_{10})

Octal Representation

- Octal (base 8) use to be popular for binary numbers
- In C, a leading 0 denotes octal as opposed to 0b for binary
- But, standard C doesn't have a way to write binary constants
- Some C compilers let you write 0b
    - Ok to use 0b in experimental code but don't use in important code

```
printf("%d", 0x2A); // prints 42
printf("%d", 052); // prints 42
printf("%d", 0b101010); // sometimes compiles and prints 42
```

Hexadecimal Representation

- Interpret hexadecimal number 3AF1 as: $3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0$
- The base or radix is 16, digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Write number as $3AF1_{16}$ (= 15089_{10})

Binary Constants

- In hexadecimal, each digit represents 4 bits

Binary	0100	1000	1111	1010	1011	1100	1001	0111
Hexadecimal	4	8	F	A	B	C	9	7

- In binary each digit represents 1 bit
  `0b01001000111110101011110010010111`
- `0b` represents the start of a binary number (in SOME compilers)
- `0` represents the start of a octal number
- `0x` represents the start of a hexadecimal number

Bits in Bytes in Words

- Values that we normally treat as atomic can be viewed as bits, e.g.
    - char = 1 byte = 8 bits (a is 01100001)
    - short = 2 bytes = 16 bits (42 is 0000000000101010)
    - int = 4 bytes = 32 bits (42 is 0000000000 … 0000101010)
    - double = 8 bytes = 64 bits

The above are common sizes and don't apply on all hardware, e.g. sizeof(int) might be 2, 4 or 8.

- `sizeof` ***variable*** – returns the number of bytes to store a variable
- `sizeof (`***type***`)` – returns the number of bytes to store a type

stdint.h

- Remembering char, short, int, double can be daunting

Type	Min	Max
char	-128	127
signed char	-128	127
unsigned char	0	255
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	-1
long	-9223372036854775808	9223372036854775807
unsigned long	0	18446744073709551615
long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

- stdint.h allows us to get guaranteed integer sizes!

```
            // range of values for type
            //              minimum                maximum
int8_t   i1; //                 -128                   127
uint8_t  i2; //                    0                   255
int16_t  i3; //               -32768                 32767
uint16_t i4; //                    0                 65535
int32_t  i5; //          -2147483648            2147483647
uint32_t i6; //                    0            4294967295
int64_t  i7; // -9223372036854775808  9223372036854775807
uint64_t i8; //                    0 18446744073709551615
```

- Signed means the values can be zero, positive and negative
- Unsigned means the values can be zero and positive

Two's Complement

- Modern computers almost always use twos complement to represent integers
- Positive integers and zero represented in obvious away
- Negative integers represented in clever way to make arithmetic in silicon fast/simpler
- For a n-bit binary the representation of `-b` is $2^n - b$
- e.g. in 8-bit two's complement `-5` is represented as $2^8 - 5 == 11111011_2$

Code example: printing all 8 bit twos complement bit patterns

```
for (int i = -128; i < 128; i++) {
      printf("%4d ", i);
      print_bits(i, 8);
      printf("\n");
}
```

- Note that: %4d, means to have a 4 character space aligned to the right as shown below

```
$ ./8_bit_twos_complement
-128 10000000
-127 10000001
-126 10000010
...
  -3 11111101
  -2 11111110
  -1 11111111
   0 00000000
   1 00000001
   2 00000010
   3 00000011
...
 125 01111101
 126 01111110
 127 01111111
```

- Notice how all the negative values begin with a 1 and positive values don't

Code example: char_bug.c

```
char c; // c should be declared int
while ((c = getchar()) != EOF) {
      putchar(c);
}
```

- stdio.h #defines EOF as -1
- Most platforms: char is signed (-128..127)
    - The loop will incorrectly exit for a byte containing 0xFF, or 255_{10} since 255 is -1 since the values loop back around
- Rare platforms: char is unsigned (0..255)
    - The loop will never exit

Bitwise Operations

Bitwise AND &

- Takes two values, and treats them as a sequence of bits
- Performs logical AND on each corresponding pair of bits
- Result contains same number of bits as inputs

Example:

```
    00100111              AND | 0  1
  & 11100011              ----|------
    --------                0 | 0  0
    00100011                1 | 0  1
```

- Can use to check for odd numbers in C

```
int isOdd(int n) {
    return n & 1;
}
```

Bitwise OR |

- Takes two values, and treats them as a sequence of bits
- Performs logical OR on each corresponding pair of bits
- Result contains same number of bits as inputs

Example:

```
    00100111               OR | 0  1
  | 11100011              ----|------
    --------                0 | 0  1
    11100111                1 | 1  1
```

Bitwise NEG ~

- Takes a single value, and treats them as a sequence of bits
- Performs logical NEG (negation) on each corresponding pair of bits
- Result contains same number of bits as inputs

Example:

```
  ~ 00100111             NEG | 0  1
    --------             ----|------
    11011000                 | 1  0
```

Bitwise XOR ^

- Takes two values, and treats them as a sequence of bits
- Performs logical XOR (exclusive or) on each corresponding pair of bits
- Result contains same number of bits as inputs

```
Example:

  00100111      XOR | 0  1
^ 11100011      ----|-----
  --------       0  | 0  1
  11000100       1  | 1  0
```

Left Shift <<

- Takes two values, and treats them as a sequence of bits and a small positive integer x
- Shifts each bit x positions to the left
- The left-end bit vanishes and the right-end bit replaced by zero
- Result contains same number of bits as inputs
- Left shift allows for multiplication by 2^n

```
Example:

00100111 << 2     00100111 << 8
--------          --------
10011100          00000000
```

Right Shift <<

- Takes two values, and treats them as a sequence of bits and a small positive integer x
- Shifts each bit x positions to the right
- The right-end bit vanishes and the left-end bit replaced by zero
- Result contains same number of bits as inputs
- Right shift allows for division by 2^n

```
Example:

00100111 >> 2       00100111 >> 8
--------            --------
00001001            00000000
```

Shifting with Negative Values

- Warning! Shifts involving negative values are not portable
- It is a common source of bugs in COMP1521 and elsewhere
- Always use unsigned values/variables to be safe/portable

shift_as_multiply.c: using shift to multiply 2^n

```
$ dcc shift_as_multiply.c print_bits.c -o shift_as_multiply
$ ./shift_as_multiply 4
2 to the power of 4 is 16
In binary it is: 00000000000000000000000000010000
$ ./shift_as_multiply 20
2 to the power of 20 is 1048576
In binary it is: 00000000000100000000000000000000
$ ./shift_as_multiply 31
2 to the power of 31 is 2147483648
In binary it is: 10000000000000000000000000000000
$
```

Code:

```
// take in the power from command line (string) and convert it into an integer
int n = strtol(argv[1], NULL, 0);
uint32_t power_of_two;

// number of bits = 8 * number of bytes of the unsigned integer
int n_bits = 8 * sizeof power_of_two;

// error checking, 1 will be shifted out of the binary
if (n >= n_bits) {
      fprintf(stderr, "n is too large\n");
      return 1;
}

// get 1
power_of_two = 1;

// shift it by the power
power_of_two = power_of_two << n;

// 2 to the power of "power" is "the shifted value"
// %u simply means an unsigned integer
printf("2 to the power of %d is %u\n", n, power_of_two);
printf("In binary it is: ");
print_bits(power_of_two, n_bits);
printf("\n");
```

set_low_bits.c: using << and – to set low n bits

```
$ dcc set_low_bits.c print_bits.c -o n_ones
$ ./set_low_bits 3
The bottom 3 bits of 7 are ones:
00000000000000000000000000000111
$ ./set_low_bits 19
The bottom 19 bits of 524287 are ones:
00000000000000111111111111111111
$ ./set_low_bits 29
The bottom 29 bits of 536870911 are ones:
00011111111111111111111111111111
```

Coding this has a neat trick!

```
// convert the argument in command line (string) into an integer
// this is how many "1"s you want
int n = strtol(argv[1], NULL, 0);

// space for 32 bits
uint32_t mask;
int n_bits = 8 * sizeof mask;

// don't use assert as a shortcut for error checking,
// but as vital information for the people who read your code
assert(n >= 0 && n < n_bits);

mask = 1;
mask = mask << n;

// e.g. 01000000 -> 00111111
mask = mask - 1;

printf("The bottom %d bits of %u are ones:\n", n, mask);
print_bits(mask, n_bits);
printf("\n");
```

set_bit_range.c: using << and – to set low n bits

```
$ dcc set_bit_range.c print_bits.c -o set_bit_range
$ ./set_bit_range 0 7
Bits 0 to 7 of 255 are ones:
00000000000000000000000011111111
$ ./set_bit_range 8 15
Bits 8 to 15 of 65280 are ones:
00000000000000001111111100000000
$ ./set_bit_range 8 23
Bits 8 to 23 of 16776960 are ones:
00000000111111111111111100000000
$ ./set_bit_range 1 30
Bits 1 to 30 of 2147483646 are ones:
01111111111111111111111111111110
```

Again we will use a masking trick!

1. Move a 1 to the max_value – min_value + 1
2. Subtract -1 from that value
3. Shift it to the min_value position

```c
int low_bit = strtol(argv[1], NULL, 0);
int high_bit = strtol(argv[2], NULL, 0);

uint32_t mask;
int n_bits = 8 * sizeof mask;

// how many "1"s will be printed?
int mask_size = high_bit - low_bit + 1;

mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;

printf("Bits %d to %d of %u are ones:\n", low_bit, high_bit, mask);
print_bits(mask, n_bits);
printf("\n");
```

extract_bit_range.c: extracting a range of bits

```
$ dcc extract_bit_range.c print_bits.c -o extract_bit_range
$ ./extract_bit_range 4 7 42
Value 42 in binary is:
00000000000000000000000000101010
Bits 4 to 7 of 42 are:
0010
$ ./extract_bit_range 10 20 123456789
Value 123456789 in binary is:
00000111010110111100110100010101
Bits 10 to 20 of 123456789 are:
11011110011
```

Again we will use a masking trick!

```c
// we will need a mask! how big is the mask?
int mask_size = high_bit - low_bit + 1;

// we will get 1, shift it by the mask size, subtract one and
// move it to the low_bit, like in the previous example
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;

// now! we must & the value and the mask, to delete all irrelevant bits in
// the value and keep the ones in the mask
uint32_t extracted_bits = value & mask;

// right shift the extracted_bits so low_bit becomes bit 0
extracted_bits = extracted_bits >> low_bit;

printf("Value %u in binary is:\n", value);
print_bits(value, n_bits);
printf("\n");
printf("Bits %d to %d of %u are:\n", low_bit, high_bit, value);
print_bits(extracted_bits, mask_size);
printf("\n");
```

print_bits.c

- This involves extracting the bit value of a number an index, then printing that bit, then repeat!

```
void print_bits(uint64_t value, int how_many_bits) {

	// print bits from highest index, to lowest index (0)
	for (int i = how_many_bits - 1; i >= 0; i--) {
		int bit = get_nth_bit(value, i);
		printf("%d", bit);
	}
}

int get_nth_bit(uint64_t value, int n) {

	// shift the bit right n bits
	// this leaves the n-th bit as the least significant bit
	uint64_t shifted_value = value >> n;

	// zero all bits except the the least significant bit
	int bit = shifted_value & 1;
	return bit;
}
```

print_int_in_hex.c

- Instead of using `printf("%x", n)`, how could we print an int in hex?

```
$ dcc print_int_in_hex.c -o print_int_in_hex
$ ./print_int_in_hex
Enter a positive int: 42
42 = 0x0000002A
$ ./print_int_in_hex
Enter a positive int: 65535
65535 = 0x0000FFFF
$ ./print_int_in_hex
Enter a positive int: 3735928559
3735928559 = 0xDEADBEEF
$
```

```c
int main(void) {
	uint32_t a = 0;
	printf("Enter a positive int: ");
	scanf("%u", &a);

	printf("%u = 0x", a);

	print_hex(a);

	printf("\n");
	return 0;
}

void print_hex(uint32_t n) {

	// sizeof return number of bytes in n's representation
	// each byte is 2 hexadecimal digits
	int n_hex_digits = 2 * (sizeof n);

	// print hex digits from most significant to least significant
	for (int which_digit = n_hex_digits - 1; which_digit >= 0; which_digit--)

		// shift value across so hex digit we want
		// is in bottom 4 bits
		int bit_shift = 4 * which_digit;
		uint32_t shifted_value = n >> bit_shift;

		// mask off (zero) all bits but the bottom 4 bites
		int hex_digit = shifted_value & 0xF;

		// hex digit will be a value 0..15
		// obtain the corresponding ASCII value
		// "0123456789ABCDEF" is a char array
		// containing the appropriate ASCII values (+ a '\0')
		// this is cool! vvv
		int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
		putchar(hex_digit_ascii);
	}
}
```

int_to_hex_string.c

```
$ dcc int_to_hex_string.c -o int_to_hex_string
$ ./int_to_hex_string
$ ./int_to_hex_string
Enter a positive int: 42
42 = 0x0000002A
$ ./int_to_hex_string
Enter a positive int: 65535
65535 = 0x0000FFFF
$ ./int_to_hex_string
Enter a positive int: 3735928559
3735928559 = 0xDEADBEEF
$
```

```c
int main(void) {
	uint32_t a = 0;
	printf("Enter a positive int: ");
	scanf("%u", &a);

	char *hex_string = int_to_hex_string(a);
	// print the returned string
	printf("%u = 0x%s\n", a, hex_string);
	free(hex_string);
	return 0;
}

char *int_to_hex_string(uint32_t n) {

	// sizeof return number of bytes in n's representation
	// each byte is 2 hexadecimal digits
	int n_hex_digits = 2 * (sizeof n);

	// allocate memory to hold the hex digits + a terminating 0
	char *string = malloc(n_hex_digits + 1);

	// print hex digits from most significant to least significant
	for (int which_digit = 0; which_digit < n_hex_digits; which_digit++)
	{
		// shift value across so hex digit we want
		// is in bottom 4 bits
		int bit_shift = 4 * which_digit;
		uint32_t shifted_value = n >> bit_shift;

		// mask off (zero) all bits but the bottom 4 bites
		int hex_digit = shifted_value & 0xF;

		// hex digit will be a value 0..15
		// obtain the corresponding ASCII value
		// "0123456789ABCDEF" is a char array
		// containing the appropriate ASCII values
		int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
		string[which_digit] = hex_digit_ascii;
	}
	// 0 terminate the array
	string[n_hex_digits] = 0;
	return string;
}
```

hex_string_to_int.c

- We could use strtol to do this, but let's not!

```
$ dcc hex_string_to_int.c -o hex_string_to_int
$ dcc hex_string_to_int.c -o hex_string_to_int
$ ./hex_string_to_int 2A
2A hexadecimal is 42 base 10
$ ./hex_string_to_int FFFF
FFFF hexadecimal is 65535 base 10
$ ./hex_string_to_int DEADBEEF
DEADBEEF hexadecimal is 3735928559 base 10
$
```

```c
int main(int argc, char *argv[]) {

	// error checking, hex string not given
	if (argc != 2) {
		fprintf(stderr, "Usage: %s <hexadecimal-number>\n", argv[0]);
		return 1;
	}

	char *hex_string = argv[1];
	uint32_t u = hex_string_to_int(hex_string);
	printf("%s hexadecimal is %u base 10\n", hex_string, u);
	return 0;
}

uint32_t hex_string_to_int(char *hex_string) {

	uint32_t value = 0;

	// go through each character in the hex string
	for (int which_digit = 0; hex_string[which_digit] != 0;
	which_digit++) {

		int ascii_hex_digit = hex_string[which_digit];

		// obtain the int of the hex digit
		int digit_as_int = hex_digit_to_int(ascii_hex_digit);

		// does the most significant character first, shift it across
		4, since each hex digit is 4 bits
		value = value << 4;

		// does operation | to save it
		value = value | digit_as_int;
	}
	return value;
}

int hex_digit_to_int(int ascii_digit) {
	if (ascii_digit >= '0' && ascii_digit <= '9') {
		// the ASCII characters '0' .. '9' are contiguous
		// in other words they have consecutive values
		// so subtract the ASCII value for '0' yields the corresponding
		return ascii_digit - '0';
	}
	if (ascii_digit >= 'A' && ascii_digit <= 'F') {
		// for characters 'A' .. 'F' obtain the
		// corresponding integer for a hexadecimal digit
		return 10 + (ascii_digit - 'A');
	}

	// error checking - prints to a file
	fprintf(stderr, "Bad digit '%c'\n", ascii_digit);
	exit(1);
}
```

shift_bug.c: bugs to AVOID

```
// int16_t is a signed type (-32768..32767)
// below operations are undefined for a signed type
int16_t i;
i = -1;
i = i >> 1; // undefined - shift of a negative value
printf("%d\n", i);
i = -1;
i = i << 1; // undefined - shift of a negative value
printf("%d\n", i);
i = 32767;
i = i << 1; // undefined - left shift produces a negative value
uint64_t j;
j = 1 << 33; // undefined - constant 1 is an int
j = ((uint64_t)1) << 33; // ok
```

xor.c: fun with xor

- Imagine how encrypting works!

```
$ echo Hello Andrew|xor 42
bOFFE
kDNXO] $ echo Hello Andrew|xor 42|cat -A
bOFFE$
kDNXO] $
$ echo Hello |xor 42
bOFFE $ echo -n 'bOFFE '|xor 42
Hello
$ echo Hello|xor 123|xor 123
Hello
$
```

- xor by the same value, gives the original message back!

```
int xor_value = strtol(argv[1], NULL, 0);

// error checking
if (xor_value < 0 || xor_value > 255) {
      fprintf(stderr, "Usage: %s <xor-value>\n", argv[0]);
      return 1;
}

int c;

while ((c = getchar()) != EOF) {
      // exclusive-or
      // ^ | 0 1
      // --|----
      // 0 | 0 1
      // 1 | 1 0
      int xor_c = c ^ xor_value;
      putchar(xor_c);
}
```

pokemon.c: using an int to represent a set of values

```
#define FIRE_TYPE       0x0001 -> 0000000000000001
#define FIGHTING_TYPE   0x0002 -> 0000000000000010
#define WATER_TYPE      0x0004 -> 0000000000000100
#define FLYING_TYPE     0x0008 -> 0000000000001000
#define POISON_TYPE     0x0010 -> 0000000000010000
#define ELECTRIC_TYPE   0x0020 -> 0000000000100000
#define GROUND_TYPE     0x0040 -> 0000000001000000
#define PSYCHIC_TYPE    0x0080 -> 0000000010000000
#define ROCK_TYPE       0x0100 -> 0000000100000000
#define ICE_TYPE        0x0200 -> 0000001000000000
#define BUG_TYPE        0x0400 -> 0000010000000000
#define DRAGON_TYPE     0x0800 -> 0000100000000000
#define GHOST_TYPE      0x1000 -> 0001000000000000
#define DARK_TYPE       0x2000 -> 0010000000000000
#define STEEL_TYPE      0x4000 -> 0100000000000000
#define FAIRY_TYPE      0x8000 -> 1000000000000000
```

- This is a simple example of a single integer specifying a set of values
- Interacting with hardware often involves this sort of code

```
// the | operator, sort of combines all the ones
uint16_t our_pokemon = BUG_TYPE | POISON_TYPE | FAIRY_TYPE;

// but the & operator, sort of only outputs the one at the point
// example code to check if a pokemon is of a type:
if (our_pokemon & POISON_TYPE) {
      printf("Poisonous\n"); // prints
}
if (our_pokemon & GHOST_TYPE) {
      printf("Scary\n"); // does not print
}

// example code to add a type to a pokemon
our_pokemon |= GHOST_TYPE;

// example code to remove a type from a pokemon
our_pokemon &= ~ POISON_TYPE;

printf(" our_pokemon type (2)\n");
if (our_pokemon & POISON_TYPE) {
      printf("Poisonous\n"); // does not print
}

if (our_pokemon & GHOST_TYPE) {
      printf("Scary\n"); // prints
}
```

bitset.c: a harder example

```
$ dcc bitset.c print_bits.c -o bitset
$ ./bitset
Set members can be 0-63, negative number to finish
Enter set a: 1 2 4 8 16 32 -1
Enter set b: 5 4 3 33 -1
a = 0000000000000000000000000000000100000000000000010000000100010110
b = 0000000000000000000000000000001000000000000000000000000000111000
a = {1,2,4,8,16,32}
b = {3,4,5,33}
a union b = {1,2,3,4,5,8,16,32,33}
a intersection b = {4}
cardinality(a) = 6
is_member(42, a) = 0
```

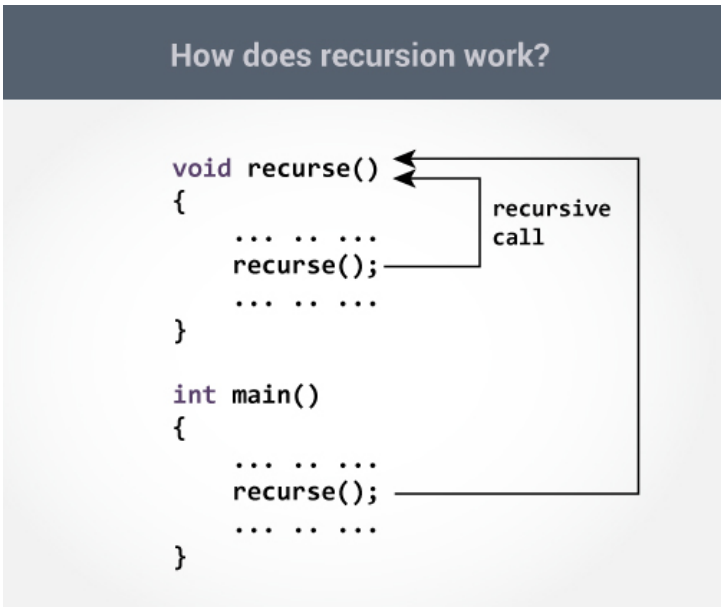
```c
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include "print_bits.h"

typedef uint64_t set;

#define MAX_SET_MEMBER ((int)(8 * sizeof(set) - 1))
#define EMPTY_SET 0

set set_add(int x, set a);
set set_union(set a, set b);
set set_intersection(set a, set b);
set set_member(int x, set a);
int set_cardinality(set a);
set set_read(char *prompt);
void set_print(char *description, set a);

void print_bits_hex(char *description, set n);

int main(void) {
    printf("Set members can be 0-%d, negative number to finish\n",
           MAX_SET_MEMBER);
    set a = set_read("Enter set a: ");
    set b = set_read("Enter set b: ");

    print_bits_hex("a = ", a);
    print_bits_hex("b = ", b);
    set_print("a = ", a);
    set_print("b = ", b);
    set_print("a union b = ", set_union(a, b));
    set_print("a intersection b = ", set_intersection(a, b));
    printf("cardinality(a) = %d\n", set_cardinality(a));
    printf("is_member(42, a) = %d\n", (int)set_member(42, a));

    return 0;
}
```

```
set set_add(int x, set a) {
    return a | ((set)1 << x);
}

set set_union(set a, set b) {
    return a | b;
}

set set_intersection(set a, set b) {
    return a & b;
}

// return a non-zero value iff x is a member of a
set set_member(int x, set a) {
    assert(x >= 0 && x < MAX_SET_MEMBER);
    return a & ((set)1 << x);
}

// return size of set
int set_cardinality(set a) {
    int n_members = 0;
    while (a != 0) {
        n_members += a & 1;
        a >>= 1;
    }
    return n_members;
}

set set_read(char *prompt) {
    printf("%s", prompt);
    set a = EMPTY_SET;
    int x;
    while (scanf("%d", &x) == 1 && x >= 0) {
        a = set_add(x, a);
    }
    return a;
}

// print out member of the set in increasing order
// for example {5,11,56}
void set_print(char *description, set a) {
    printf("%s", description);
    printf("{");
    int n_printed = 0;
    for (int i = 0; i < MAX_SET_MEMBER; i++) {
        if (set_member(i, a)) {
            if (n_printed > 0) {
                printf(",");
            }
            printf("%d", i);
            n_printed++;
        }
    }
    printf("}\n");
}
```

```c
// print description then binary, hex and decimal representation of value
void print_bits_hex(char *description, set value) {
    printf("%s", description);
    print_bits(value, 8 * sizeof value);
    printf(" = 0x%08jx = %jd\n", (intmax_t)value, (intmax_t)value);
}
```

Recursion

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.



Other Notes

- `%x` will print lowercase hexadecimal
- `%X` will print uppercase hexadecimal
- `int8_t` has 8 bits, `int16_t` has 16 bits, `int32_t` has 32 bits
- `atoi` converts a string to an integer and has no error checking
- `strtol` converts a string to an integer and can convert to hexadecimal, decimal, etc

Floating Point

Floating Point Numbers

- C has three floating point types:
  - `float` . . . typically 32-bit (lower precision, narrower range)
  - `double` . . . typically 64-bit (higher precision, wider range)
  - `long double` . . . typically 128-bits (but only 80 used, we rarely use this)
- Floating point constants, e.g. 3.14159 or 1.9e-9 are `double`
- Most of the time we'll be using doubles. Float is more used when saving space is crucial

- REMEMBER: Division of 2 ints in C yields an int, but division of a double and int in C yields a double!
- This allows you to turn an int into a double. Or another way, for example for 1, is using: ((uint32_t)1), which turns the 1 into an unsigned 32 bit int type with stdint.h

```
double d = 4/7.0;

// prints in decimal with (default) 6 decimal places
printf("%lf\n", d);      // prints 0.571429

// prints in scientific notation
printf("%le\n", d);      // prints 5.714286e-01

// picks best of decimal and scientific notation
printf("%lg\n", d);      // prints 0.571429

// prints in decimal with 9 decimal places
printf("%.9lf\n", d);     // prints 0.571428571

// prints in decimal with 1 decimal place and field width of 10 (aligned to
the right, place a negative after the % for left justification/alignment)
printf("%10.1lf\n", d);  // prints 0.6
```

We cannot represent all Reals with Floating Point Numbers

- Remember that we have a fixed number of bits.
- So there is a finite number of bit patterns.
- This means that only a finite subset of reals can be represented!
- Almost all real values have no EXACT representation!!!!! So floating point numbers represent the closest.
- This results in an error in our calculations that may or may not be disastrous.

Characteristics of Floating Point Types

```c
#include <stdio.h>
#include <float.h>

int main(void) {

    float f;
    double d;
    long double l;
    printf("float       %2lu bytes  min=%-12g  max=%g\n", sizeof f, FLT_MIN, FLT_MAX);
    printf("double      %2lu bytes  min=%-12g  max=%g\n", sizeof d, DBL_MIN, DBL_MAX);
    printf("long double %2lu bytes  min=%-12Lg  max=%Lg\n", sizeof l, LDBL_MIN, LDBL_MAX);

    return 0;
}
```

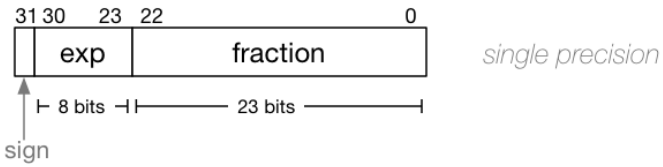
```
$ ./floating_types
float        4 bytes  min=1.17549e-38   max=3.40282e+38
double       8 bytes  min=2.22507e-308  max=1.79769e+308
long double 16 bytes  min=3.3621e-4932  max=1.18973e+4932
```

How do we represent Floating Point Numbers? Introducing the IEEE 754 standard!

- C floats almost always IEEE 754 single precision (binary32)
- C double almost always IEEE 754 double precision (binary64)
- C long double might be IEEE 754 (binary128)
- IEEE 754 representation has 3 parts: sign, fraction and exponent

IEEE 754 single precision

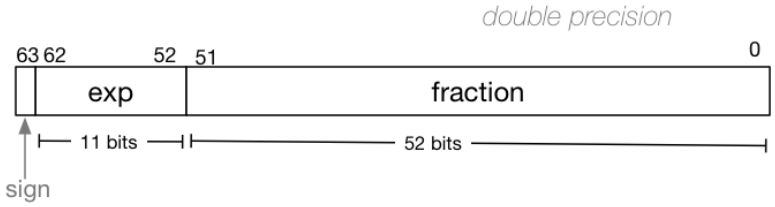
- Numbers are calculated by: $(-1)^{sign} * (1 + fraction) * 2^{power}$
- Are represented by in IEEE 754 single precision as:



- **sign** is simply if the number is positive or negative
  - Positive means sign = 0
  - Negative means sign = 1
- **fraction** always has one digit before the decimal point (normalised)
  - Which is a **1**, this is not stored
- **exponent** is stored as a positive number, which is a number between 0 and 255.
- To get the **power**, for a 32-bit float, subtract $2^{8-1} - 1 = 127$ (**bias**).
  Hence, **bias** + **power** = **exponent**.

IEEE 754 double precision

- Numbers are calculated by: $(-1)^{sign} * (1 + fraction) * 2^{power}$
- Are represented by in IEEE 754 double precision as:



- **sign** is simply if the number is positive or negative
  - Positive means sign = 0
  - Negative means sign = 1
- **fraction** always has one digit before the decimal point (normalised)
  - Which is a **1**, this is not stored
- **exponent** is stored as a positive number, which is a number between 0 and 2047.
- To get the **power**, for a 32-bit float, subtract $2^{11-1} - 1 = 1023$ (**bias**).
  Hence, **bias** + **power** = **exponent**.

Examples

- 1010.1011 is normalised as 1.0101011×2^{011}
- $1010.1011 = 10 + 11/16 = 10.6875$
- $1.0101011 \times 2^{011} = (1 + 43/128) \times 2^3 = 1.3359375 \times 8 = 10.6875$

```
0.15625 is represented in IEEE-754 single-precision by these bits:
00111110001000000000000000000000

sign | exponent | fraction
   0 | 01111100 | 01000000000000000000000

sign bit = 0
sign = +

exponent        = 01111100 binary
                = 124 decimal
power           = 124 - exponent_bias
                = 124 - 127
                = -3
number = +1.01000000000000000000000 binary * 2**-3
       = 1.25 decimal * 2**-3
       = 1.25 * 0.125
       = 0.15625
```

```
$ ./explain_float_representation -0.125
-0.125 is represented as a float (IEEE-754 single-precision) by these
10111110000000000000000000000000

sign | exponent | fraction
   1 | 01111100 | 00000000000000000000000

sign bit = 1
sign = -

exponent = 01111100 binary
         = 124 decimal
power    = 124 - exponent_bias
         = 124 - 127
         = -3

number = -1.00000000000000000000000 binary * 2**-3
       = -1 decimal * 2**-3
       = -1 * 0.125
       = -0.125

$ ./explain_float_representation 150.75
150.75 is represented in IEEE-754 single-precision by these bits:
01000011000101101100000000000000

sign | exponent | fraction
   0 | 10000110 | 00101101100000000000000

sign bit = 0
sign = +

exponent = 10000110 binary
         = 134 decimal
power    = 134 - exponent_bias
         = 134 - 127
         = 7

number = +1.00101101100000000000000 binary * 2**7
       = 1.17773 decimal * 2**7
       = 1.17773 * 128
       = 150.75
```

C (IEEE 754) has representation for +/- infinity, NaN and zero

- These values exist in math.h, and propagates sensibly through calculations

```c
#include <stdio.h>
#include <math.h>

int main(void) {
    double x = 1.0/0.0;
    printf("%lf\n", x); //prints inf
    printf("%lf\n", -x); //prints -inf
    printf("%lf\n", x - 1); // prints inf
    printf("%lf\n", 2 * atan(x)); // prints 3.141593
    printf("%d\n", 42 < x); // prints 1 (true)
    printf("%d\n", x == INFINITY); // prints 1 (true)
    return 0;
}
```

Infinity Representation

- Conditions (all must be met):
    - **sign** is 0 or 1
        - if 0 -> positive infinity
        - if 1 -> negative infinity
    - **exponent** is ALL 1
    - **fraction** is ALL 0

Example:

```
$ ./explain_float_representation inf
inf is represented in IEEE-754 single-precision by these bits:
01111111100000000000000000000000

sign | exponent | fraction
   0 | 11111111 | 00000000000000000000000

sign bit = 0
sign = +

raw exponent = 11111111 binary
             = 255 decimal

number = +inf
```

NaN (Not a Number) Representation

- Negative NaN doesn't exist. It may print it, but has no meaning!
- Conditions (all must be met):
    - **exponent** is ALL 1
    - **fraction** is NOT 0

Example:

```
$ ./explain_float_representation 01111111110000000000000000000000

sign | exponent | fraction
   0 | 11111111 | 10000000000000000000000

sign bit = 0
sign = +

raw exponent = 11111111 binary
             = 255 decimal

number = NaN
```

Zero Representation

- 0 and -0 exists!
- Conditions (all must be met):
    - **sign** is 0 or 1
        - if 0 -> positive infinity
        - if 1 -> negative infinity
    - **exponent** is ALL 0
    - **fraction** is ALL 0

Example:

```
$ ./explain_float_representation -0
inf is represented in IEEE-754 single-precision by these bits:
00000000000000000000000000000000

sign | exponent | fraction
   1 | 00000000 | 00000000000000000000000

sign bit = 1
sign = -

raw exponent = 00000000 binary
             = 0 decimal

number = -0
```

Consequence of most reals not having exact representation

```c
#include <stdio.h>

int main(void) {
    double a, b;

    a = 0.1;
    b = 1 - (a + a + a + a + a + a + a + a + a + a);

    if (b != 0) {  // better would be fabs(b) > 0.000001
                   // fabs() takes in a double
        printf("1 != 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1\n");
    }

    printf("b = %g\n", b); // prints 1.11022e-16

    return 0;
}
```

- Why did this happen?
- a is not represented as 0.1, but more like 0.10...55
- This gets weird results
- So: DO NOT USE == and != with floating point values
- Instead check if values are close using: fabs(b) > 0.00001

Catastrophic Cancellations

```c
#include <stdio.h>
#include <math.h>

int main(void) {

    double x = 0.000000011;
    double y = (1 - cos(x)) / (x * x);

    // correct answer y = ~0.5
    // prints y = 0.917540
    printf("y = %lf\n", y);

    // division of similar approximate value
    // produces large error
    // sometimes called catastrophic cancellation
    printf("%g\n", 1 - cos(x)); // prints  1.11022e-16
    printf("%g\n", x * x); // prints 1.21e-16
    return 0;
}
```

- Demonstrate approximate representation of reals producing error. sometimes if we subtract or divide two approximations which are very close together we can can get a large relative error correct answer if x == 0.000000011 (1 - cos(x)) / (x * x) is very close to 0.5 code prints 0.917540 which is wrong by a factor of almost two

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/*```
$ dcc double_not_always.c -o double_not_always
$ ./double_not_always 42.3
d = 42.3
d == d is true
d == d + 1 is false
$  ./double_not_always 4200000000000000000
d = 4.2e+18
d == d is true
d == d + 1 is true
$ ./double_not_always NaN
d = nan
d == d is not true
d == d + 1 is false
````*/

int main(int argc, char *argv[]) {
    assert(argc == 2);

    double d = strtod(argv[1], NULL);

    printf("d = %g\n", d);

    if (d == d) {
        printf("d == d is true\n");
```
```

```
    } else {
        // will be executed if d is a NaN
        printf("d == d is not true\n");
    }

    if (d == d + 1) {
        // may be executed if d is large
        // because closest possible representation for d + 1
        // is also closest possible representation for d
        printf("d == d + 1 is true\n");
    } else {
        printf("d == d + 1 is false\n");
    }

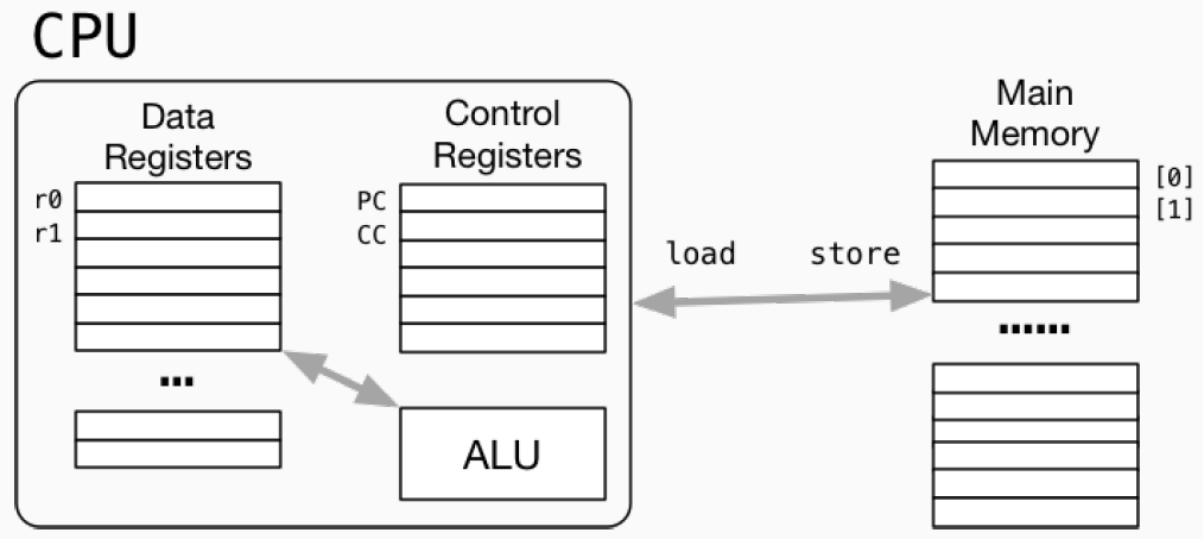
    return 0;
}
```

- 9007199254740993 is 253 + 1
  it is smallest integer which can not be represented exactly as a double
- The closest double to 9007199254740993 is 9007199254740992.0
- aside: 9007199254740993 can not be represented by a int32_t
  it can be represented by int64_t

MIPS Basics

CPU Components

- A typical modern CPU has
  - a set of data registers
  - a set of control registers (incl PC)
  - an arithmetic-logic unit (ALU)
  - access to memory (RAM)
  - a set of simple instructions
    - transfer data between memory and registers
    - push values through the ALU to compute results
    - make tests and transfer control of execution
- Different types of processors have different configurations of the above



How a CPU (1 core) works: Fetch-Execution Cycle

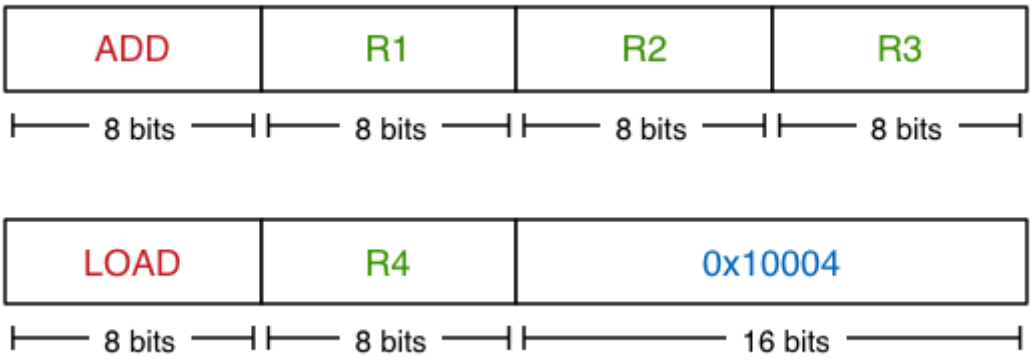
- Typical CPU program execution pseudo-code:

```
uint32_t pc = STARTING_ADDRESS;
while (1) {
	uint32_t instruction = memory[pc]; // fetch the instruction
	pc++; // move to next instruction
	if (instruction == HALT) {
		break;
	} else {
		execute(instruction); // decode the 32-bit instruction + execute it
	}
}
```

- `pc` = program counter, a CPU register which tracks execution

How do we execute an instruction?

- Executing an instruction involves:
  - Determine what the OPERATOR is
  - Determine which REGISTERS, if any, are involved
  - Determine which MEMORY LOCATION, if any, is involved
  - PERFORM the operation with the relevant operands
  - STORE result, if any, in appropriate register

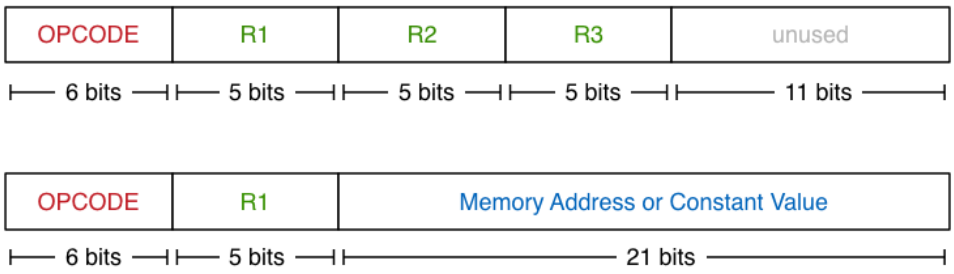


MIPS Architecture

- MIPS is a well-known and simple architecture
  - Historically used everywhere
  - Still used in some embedded fields e.g. modems, TVs
  - But being out-competed by ARM (in phones, etc)
- We consider the MIPS32 version of the MIPS family
  - qtspim: provides a GUI front-end, useful for debugging
  - spim: command-line based version, useful for testing
  - xspim: GUI front-end, useful for debugging, only in CSE labs

MIPS Instructions

- MIPS has several classes of its own instructions:
  - load and store: transfer data between registers and memory
  - computational: perform arithmetic/logical operations
  - jump and branch: transfer control of program execution (allows for loops)
  - coprocessor: standard interface to various co-processors
  - special: miscellaneous tasks (e.g. syscall) – making requests to the OS
- Every MIPS instruction are 32-bits long, and specify …
  - An OPERATION (e.g. load, store, add, branch, …)
  - One or more operands (e.g. registers, memory addresses, constants)
- The instruction formation varies, some possible formats:



- So MIPS instructions are simply 32-bit patterns
- We could write those instructions using a sequence of hex digits such as 0x3c041001, 0x34020004, 0x0000000c or 0x03e00008. But this is unreadable and difficult to maintain!
- Adding/removing instructions changes the bit pattern for other instructions
- Changing variable layout in memory, changes bit patterns for instructions
- SOLUTION: assembly language – a symbolic way of specifying machine code
    - Write instructions using names rather than bit strings
    - Refer to instructions using either numbers or names
    - Allow names (labels) to be associated with memory addresses
- Examples:

```
lw     $t1,address         # reg[t1] = memory[address]
sw     $t3,address         # memory[address] = reg[t3]
                           # address must be 4-byte aligned
la     $t1,address         # reg[t1] = address
lui    $t2,const           # reg[t2] = const <<< 16
and    $t0,$t1,$t2         # reg[t0] = reg[t1] & reg[t2]
add    $t0,$t1,$t2         # reg[t0] = reg[t1] + reg[t2]
                           # add signed 2's complement ints
addi   $t2,$t3, 5          # reg[t2] = reg[t3] + 5
                           # add immediate, no sub immediate
mult   $t3,$t4             # (Hi,Lo) = reg[t3] * reg[t4]
                           # store 64-bit result in
                           # registers Hi,Lo
seq    $t7,$t1,$t2         # reg[t7] = (reg[t1] == reg[t2])
j      label               # PC = label
beq    $t1,$t2,label       # PC = label if reg[t1]==reg[t2]
nop                        # do nothing
```

- MIPS CPU has:
    - 32 general purpose registers (32-bit)
    - 16/32 floating-point registers (for float/double)
        - If you do operations involving floating point numbers
        - Varies in different architectures and no useful to learn in COMP1521
    - PC: 32-bit register (always aligned on 4-byte boundary)
        - A special register used by the branch and jump operator
        - It indicates where a computer is in its program sequence
    - HI, LO: for storing results of multiplication and division
- CPU accesses registers faster than accessing from memory

- We can refer to registers as $0 .. $31 or by symbolic names
    - Register $0 always has value 0, cannot be written
    - Registers $1, $26, $27 reserved for use by system
    - There is no difference between registers 1..30 in the silicon
- In OUR code, we will use their symbolic names as it allows compiled code from different sources to be combined (linked)
- Generally, use registers 8 to 25 ($t0 .. $t9 $s0 .. $s7)
- Use other registers only for conventional purpose
    - E.g. use $a0..$a3 for arguments (e.g. stuff that you want to print)
- Never use registers $1, $26, $27

- Integer registers:

Number	Names	Conventional Usage
0	$zero	Constant 0
1	$at	Reserved for assembler
2,3	$v0,$v1	Expression evaluation and results of a function
4..7	$a0..$a3	Arguments 1-4
8..16	$t0..$t7	Temporary (not preserved across function calls)
16..23	$s0..$s7	Saved temporary (preserved across function calls)
24,25	$t8,$t9	Temporary (preserved across function calls)
26,27	$k0,$k1	Reserved for OS kernel
28	$gp	Pointer to global area
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address (used by function call instruction)

Data and Addresses

- All operations refer to data, either
  - in a register
  - in memory
  - a constant which I embedded in the instruction itself
    - the syntax for a constant is the same like C, e.g: 1, -3, -1, “a string”, ‘a’, ‘\n’
- Computation operations refer to registers or constants
- Only load/store instructions refer to memory

Integer Arithmetic Instructions

assembly	meaning	bit pattern
`add` r_d, r_s, r_t	$r_d = r_s + r_t$	000000ssssstttttddddd00000100000
`sub` r_d, r_s, r_t	$r_d = r_s$ - r_t	000000ssssstttttddddd00000100010
`mul` r_d, r_s, r_t	$r_d = r_s$ * r_t	011100ssssstttttddddd00000000010
`rem` r_d, r_s, r_t	$r_d = r_s$ % r_t	pseudo-instruction
`div` r_d, r_s, r_t	$r_d = r_s$ / r_t	pseudo-instruction
`addi` r_t, r_s, I	$r_t = r_s$ + I	001000ssssstttttIIIIIIIIIIIIIIII

- Integer arithmetic is 2’s-complement
- There are equivalent instructions such as `addu`, `subu`, `mulu`, `addiu`: instructions which do not stop execution on overflow
- `spim` allows second operand (r_t) to be replaced by a constant and will generate appropriate real MIPS instruction(s)

assembly	meaning	bit pattern
`div` r_s,r_t	`hi` = r_s % r_t; `lo` = r_s / r_t	000000ssssstttttt0000000000011010
`mult` r_s,r_t	`hi` = (r_s * r_t) >> 32 `lo` = (r_s * r_t) & 0xffffffff	000000ssssstttttt0000000000011000
`mflo` r_d	r_d = `lo`	0000000000000000ddddd000000001010
`mfhi` r_d	r_d = `hi`	0000000000000000ddddd000000001001

- ^ Little use of these except in challenge exercises
- `mult` provides multiply with 64-bit result instead of the usual `mul`
- also for `div`, but has a different number of operands
- pseudo-instruction `rem` r_d, r_s, r_t translated to `div` r_s, r_t plus `mfhi` r_d
- pseudo-instruction `div` r_d, r_s, r_t translated to `div` r_s, r_t plus `mflo` r_d
- `divu` and `multu` are unsigned equivalents of `div` and `mult`

Bit Manipulation Instructions

assembly	meaning	bit pattern
`and` r_d, r_s, r_t	r_d = r_s & r_t	000000ssssstttttddddd00000100100
`or` r_d, r_s, r_t	r_d = r_s | r_t	000000ssssstttttddddd00000100101
`xor` r_d, r_s, r_t	r_d = r_s ^ r_t	000000ssssstttttddddd00000100110
`nor` r_d, r_s, r_t	r_d = ~ (r_s | r_t)	000000ssssstttttddddd00000100111
`andi` r_t, r_s, I	r_t = r_s & I	001100ssssstttttIIIIIIIIIIIIIIII
`ori` r_t, r_s, I	r_t = r_s | I	001101ssssstttttIIIIIIIIIIIIIIII
`xori` r_t, r_s, I	r_t = r_s ^ I	001110ssssstttttIIIIIIIIIIIIIIII
`not` r_d, r_s	r_d = ~ r_s	pseudo-instruction

- spim translates `not` r_d, r_s to `nor` r_d, r_s, $0

Shift Instructions

assembly	meaning	bit pattern
`sllv` r_d, r_t, r_s	$r_d = r_t << r_s$	000000ssssstttttddddd00000000100
`srlv` r_d, r_t, r_s	$r_d = r_t >> r_s$	000000ssssstttttddddd00000000110
`srav` r_d, r_t, r_s	$r_d = r_t >> r_s$	000000ssssstttttddddd00000000111
`sll` r_d, r_t, I	$r_d = r_t << I$	00000000000tttttddddd IIIII000000
`srl` r_d, r_t, I	$r_d = r_t >> I$	00000000000tttttddddd IIIII000010
`sra` r_d, r_t, I	$r_d = r_t >> I$	00000000000tttttddddd IIIII000011

- `sll` (shift left logical): shifts to the left and fills in with 0s
- `srl` (shift right logical): shifts to the right and fills with 0s
- `sra` (shift right arithmetic): shifts to the right and fills in with whatever was the most significant digit, useful for ensuring that shifting a negative number divides by two

  1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  1111 1101 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

  0011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  0000 0001 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
- `sllv`, `srlv`, `srav`: the third operand is a register
- spim provides `rol` and `ror`: pseudo instructions which rotate bits (no simple C equivalent)

Miscellaneous Instructions

assembly	meaning	bit pattern
`li` R_d, *value*	$R_d = value$	psuedo-instruction
`la` R_d, *label*	$R_d = label$	psuedo-instruction
`move` R_d, R_s	$R_d = R_s$	psuedo-instruction
`slt` R_d, R_s, R_t	$R_d = R_s < R_t$	000000ssssstttttddddd00000101010
`slti` R_t, R_s, I	$R_t = R_s < I$	001010ssssstttttIIIIIIIIIIIIIIII
`lui` R_t, I	$R_t = I << 16$	00111100000tttttIIIIIIIIIIIIIIII
`syscall`	system call	00000000000000000000000000001100

- `slt` and `slt`, produce same results as C (1 for true, 0 for false)
- `lui` can help put the upper 16-bits of a value into a register

```
# examples of miscellaneous instructions
start:
      li    $8,   42     # $8 = 42
      li    $24,  0x2a   # $24 = 42
      li    $15,  '*'    # $15 = 42
      move  $8,   $9     # $8 = $9
      la    $8,   start  # $8 = address corresponding to start
```

Example Translation of Pseudo Instructions

Psuedo-Instructions	Real Instructions
`move $a1, $v0`	`addu $a1, $0, $v0`
`li $t5, 42`	`ori $t5, $0, 42`
`li $s1, 0xdeadbeef`	`lui $at, 0xdead  // spim split it up!` `ori $s1, $at, 0xbeef`
`la $t3, label`	`lui $at, label[31..16]` `ori $t3, $at, label[15..0]`

Examples of Instructions in Action

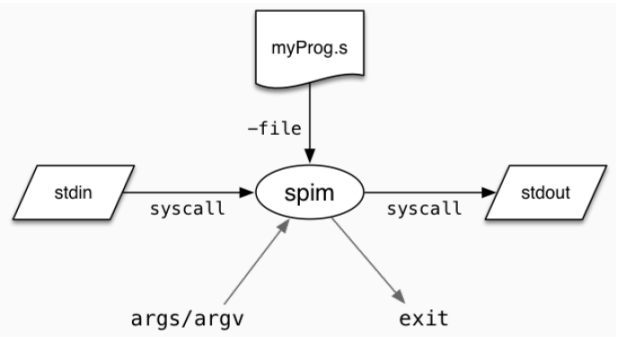
Assembler	Encoding
`add $a3, $t0, $zero`	
`add $d, $s, $t`	000000 sssss ttttt ddddd00000100000
`add $7, $8, $0`	000000 00111 01000 0000000000100000
	0x01e80020 (decimal 31981600)
`sub $a1, $at, $v1`	
`sub $d, $s, $t`	000000 sssss ttttt ddddd00000100010
`sub $5, $1, $3`	000000 00001 00011 0010100000100010
	0x00232822 (decimal 2304034)
`addi $v0, $v0, 1`	
`addi $d, $s, C`	001000 sssss ddddd CCCCCCCCCCCCCCCC
`addi $2, $2, 1`	001000 00010 00010 0000000000000001
	0x20420001 (decimal 541196289)

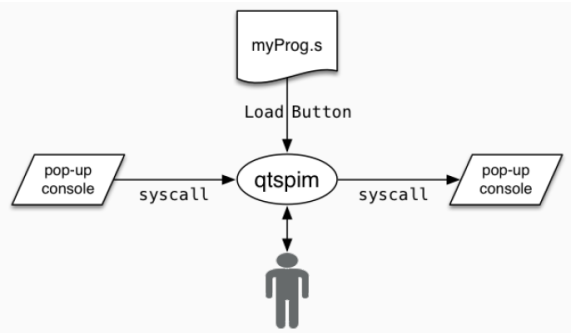
MIPS vs SPIM

- MIPS is a machine architecture, including instruction set
- SPIM is an emulator for the MIPS instruction set
- SPIM can:
  - read text files containing instruction + directives
  - converst to machine code and loads into “memory”
  - provides debugging capabilities
    - single step, breakpoints, view registers/memory etc
  - provides mechanism to interact with operating system (syscall)
- Also provides extra instructions which are more convenient/mnemonic
  - e.g. move $s0, $v0 rather than addu $s0, $0, $v0

Using SPIM

- How to execute MIPS code with SPIM
  - spim: command line tool
    - load programs using -file option
    - interact using stdin/stdout via login terminal
  - qtspim: GUI environment
    - load programs via a load button
    - interact via a pop-up stdin/stdout terminal
  - xspim: GUI environment
    - similar to qtspim, but not as pretty
    - requires X-windows server





Key System Calls

- SPIM provides I/O and memory allocation via the `syscall` instruction
- Value `$v0` specifies which system call

Service	$v0	Arguments	Returns
`printf("%d")`	1	int in $a0	
`printf("%s")`	4	string in $a0	
`scanf("%d")`	5	none	int in $v0
`fgets`	8	buffer address in $a0 length in $a1	
`exit(0)`	10	status in $a0	
`printf("%c")`	11	char in ‘$a0	
`scanf("%c")`	12	none	char in $v0

- Not usually used in COMP1521:

Service	$v0	Arguments	Returns
`printf("%f")`	2	float in $f12	
`printf("%lf")`	3	double in $f12	
`scanf("%f")`	6	none	float in $f0
`scanf("%lf")`	7	none	double in $f0
`sbrk`	9	nbytes in $a0	address in $v0
`exit(status)`	17	status in $a0	

MIPS Assembly Language

- MIPS assembly language programs contain
  - comments ... introduced by #
  - labels ... appended with :
  - directives ... symbol beginning with .
  - assembly language instructions
- Programmers need to specify
  - data objects that live in the data region
  - functions (instruction sequences) that live in the code/text region
- Each instruction or directive appears on its own line

Our First MIPS Program

- In C:

```
int main(void) {
    printf("I love MIPS\n");
    return 0;
}
```

- In MIPS:

```
main:
    la      $a0, string     # pass address of string as argument
    li      $v0, 4          # 4 is printf "%s" syscall number
    syscall
    li      $v0, 0          # return 0
    jr      $ra
    .data
string:
.asciiz "I love MIPS\n"
```

MIPS Control

Breaking up C code to translate to MIPS

- To translate C to MIPs, it is always easier to break down our C code and make it simpler. For example:

```c
#include <stdio.h>

int main(void) {
    int x = 17;
    int y = 25;
    printf("%d\n", x + y);

    return 0;
}
```

Can be written as:

```c
#include <stdio.h>

int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("\n");
    return 0;
}
```

This can be translated to MIPS easily!

```
main:                   # x,y,z in $t0,$t1,$t2,
    li   $t0, 17        # x = 17;

    li   $t1, 25        # y = 25;

    add  $t2, $t1, $t0 # z = x + y
                        # we can replace $t2 with $a0, but by following
                        # simple patterns, we write correct code.
                        # the goal is not optimisation (or minimising instructions)

    move $a0, $t2       # printf("%d", z);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0         # return 0
    jr   $ra
```

Jump Instructions

assem.	meaning	bit pattern
`j` *label*	pc = pc & 0xF0000000 | (X<<2)	000010XXXXXXXXXXXXXXXXXXXXXXXX
`jal` *label*	r_{31} = pc + 4; pc = pc & 0xF0000000 | (X<<2)	000011XXXXXXXXXXXXXXXXXXXXXXXX
`jr` r_s	pc = r_s	000000sssss00000000000000001000
`jalr` r_s	r_{31} = pc + 4; pc = r_s	000000sssss00000000000000001001

- Jump instructions UNCONDITIONALLY transfer execution to a new location
- `j` (jump): jump to specified target
- `jr` (jump register): jump to the address stored in r_s
- `jal` (jump and link): go to the target address and save in `$ra`
- `jalr` (jump and link register): go to the address stored in r_s and save in `$ra`

Branch Instructions

assembler	meaning	bit pattern
`b` *label*	pc += I<<2	pseudo-instruction
`beq` r_s,r_t,*label*	if (r_s == r_t) pc += I<<2	000100sssssttttttIIIIIIIIIIIIIIII
`bne` r_s,r_t,*label*	if (r_s != r_t) pc += I<<2	000101sssssttttttIIIIIIIIIIIIIIII
`ble` r_s,r_t,*label*	if (r_s <= r_t) pc += I<<2	pseudo-instruction
`bgt` r_s,r_t,*label*	if (r_s > r_t) pc += I<<2	pseudo-instruction
`blt` r_s,r_t*label*	if (r_s < r_t) pc += I<<2	pseudo-instruction
`bge` r_s,r_t*label*	if (r_s >= r_t) pc += I<<2	pseudo-instruction
`blez` r_s,*label*	if (r_s <= 0) pc += I<<2	000110sssss00000IIIIIIIIIIIIIIII
`bgtz` r_s,*label*	if (r_s > 0) pc += I<<2	000111sssss00000IIIIIIIIIIIIIIII
`bltz` r_s,*label*	if (r_s < 0) pc += I<<2	000001sssss00000IIIIIIIIIIIIIIII
`bgez` r_s,*label*	if (r_s >= 0) pc += I<<2	000001sssss00001IIIIIIIIIIIIIIII

- Branch instructions CONDITIONALLY transfer execution to a new location
- If the if statement is true, it transfers execution to the label
- `spim` will calculate correct value for I from location of label in code
- `spim` allows second operand (r_t) to be replaced by a constant

goto in C

- The goto statement allows transfer of control to any labelled point in a function
- We can put labels in C! (Remember: Labels in C or MIPs cannot be any keywords in C or MIPS)
- For example, this code:

```
for (int i = 1; i <= 10; i++) {
      printf("%d\n", i);
}
```

can be written as:

```
   int i = 1;
loop:
    if (i > 10) goto end;
      i++;
      printf("%d", i);
      printf("\n");
    goto loop;
end:
```

In MIPS:

```
   li    $t0, 0       # i in $t0
   li    $t1, 0       # n in $t1
loop:
   bge   $t0, 5, end
   add   $t1, $t1, $t0
   addi  $t0, $t0, 1
   j loop
end:
```

- We do not use goto in C normally!
- goto results in slower programs
- Using goto is considered bad programming style
- Do not ever use it
- Kernel and embedded programmers sometimes use goto (for error handling)

If Translation

```
if (i < 0) {
    n = n - i;
} else {
    n = n + i;
}
```

Turns into:

```
if (i >= 0) goto else1;
    n = n - i;
    goto end1;
else1:
    n = n + i;
end1:
```

- Note that `else` can't be used as a label in C

In MIPS:

```
# assume i in $t0
# assume n in $t1
    bge $t0, 0, else1
    sub $t1, $t1, $t0
    j end1
else1:
    add $t1, $t1, $t0
end1:
```

If/and Translation

```
if (i < 0 && n >= 42) {
    n = n - i;
} else {
    n = n + i;
}
```

Turns into:

```
if (i >= 0) goto else1;
if (n < 42) goto else1;
    n = n - i;
    goto end1;
else1:
    n = n + i;
end1:
```

In MIPS:

```
# assume i in $t0
# assume n in $t1
    bge $t0, 0, else1
    blt $t1, 42, else1
    sub $t1, $t1, $t0
    j end1
else1:
    add $t1, $t1, $t0
end1:
```

Odd Even Translation

```
if (i < 0 || n >= 42) {
    n = n - i;
} else {
    n = n + i;
}
```

Turns into:

```
if (i < 0) goto then1;
if (n >= 42) goto then1;
goto else1;
then1:
    n = n - i;
    goto end1;
else1:
    n = n + i;
end1:
```

In MIPS:

```
# assume i in $t0
# assume n in $t1
    blt $t0, 0, then1
    blt $t1, 42, then1
    j else1
then1:
    sub $t1, $t1, $t0
    j end1
else1:
    add $t1, $t1, $t0
end1:
```

Printing First 10 Integers Translation

```
int main(void) {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Turns into:

```
int main(void) {
    int i;
    i = 1;
loop:
    if (i > 10) goto end;
        i++;
        printf("%d", i);
        printf("\n");
    goto loop;
end:
    return 0;
}
```

In MIPS:

```
main:                   # int main(void) {
                        # int i;  // in register $t0

    li    $t0, 1        # i = 1;

loop:                   # loop:
    bgt  $t0, 10, end   # if (i > 10) goto end;

    move $a0, $t0       #   printf("%d" i);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1    #   i++;

    j    loop           # goto loop;

end:
    li   $v0, 0         # return 0
    jr   $ra
```

Odd or Even with scanf

```c
int main(void) {
    int x;

    printf("Enter a number: ");
    scanf("%d", &x);

    if ((x & 1) == 0) {
        printf("Even\n");
    } else {
        printf("Odd\n");
    }

    return 0;
}
```

Turns into:

```c
int main(void) {
    int x, v0;

    printf("Enter a number: ");
    scanf("%d", &x);

    v0 = x & 1;
    if (v0 == 1) goto odd;
        printf("Even\n");
    goto end;
odd:
        printf("Odd\n");
end:
    return 0;
}
```

In MIPS:

```
main:
    la   $a0, string0    # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5          # scanf("%d", x);
    syscall

    and  $t0, $v0, 1     # if (x & 1 == 0) {
    beq  $t0, 1, odd

    la   $a0, string1    # printf("Even\n");
    li   $v0, 4
    syscall

    j    end

odd:                     # else
    la   $a0, string2    # printf("Odd\n");
    li   $v0, 4
    syscall

end:
    li   $v0, 0          # return 0
    jr   $ra

    .data
string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Even\n"
string2:
    .asciiz "Odd\n"
```

Sum 100 Squares

```c
int main(void) {
    int sum = 0;

    for (int i = 0; i <= 100; i++) {
        sum += i * i;
    }

    printf("%d\n", sum);
    return 0;
}
```

Turns into:

```c
int main(void) {
    int i, sum, square;

    sum = 0;
    i = 0;
    loop:
       if (i > 100) goto end;
       square = i * i;
       sum = sum + square;
       i = i + 1;
    goto loop;

end:
    printf("%d", sum);
    printf("\n");

    return 0;
}
```

In MIPS:

```
main:
    li   $t0, 0          # sum = 0;
    li   $t1, 0          # i = 0

loop:
    bgt  $t1, 100, end   # if (i > 100) goto end;
    mul  $t2, $t1, $t1   # square = i * i;
    add  $t0, $t0, $t2   # sum = sum + square;

    addi $t1, $t1, 1     # i = i + 1;
    j    loop

end:
    move $a0, $t0        # printf("%d", sum);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0          # return 0
    jr   $ra
```

MIPS Data

The Memory Subsystem

- We can load and store bytes in memory, or RAM
- You can think memory as a huge array of bytes, each with a unique address
- The addresses on the MIPS CPU we are using are 32 bits, hence a 32 bit operating system

Accessing Memory on MIPS

- There are only 2 instructions, load and store, that can access memory on the MIPS
- Load operators: Load to the CPU

operator	bytes/bits
`lb` (load byte)	1 byte or 8 bits
`lh` (load half-word)	2 bytes or 16 bits
`lw` (load word)	4 bytes or 32 bits

- Store operators: Store to the memory

operator	bytes/bits
`sb` (store byte)	1 byte or 8 bits
`sh` (store half-word)	2 bytes or 16 bits
`sw` (store word)	4 bytes or 32 bits

- For `sb` and `sh` operators, low (lease significant) bits of source register are used
- For `lb` and `lh` operators, assume the byte or half-word respectively, contains a 8-bit/16-bit signed integer, so… the high 24/16-bits of the destination register are set to 1 if 8-bit/16-bit integer is negative – in order to preserve the number
- There is a unsigned equivalent, `lbu` and `lhu` which assume the integer is unsigned, soo.. the high 24/16-bits of destination register is set to 0

MIPS Load/Store Instructions

assembly	meaning	bit pattern
`lb` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`]	`100000ssssstttttIIIIIIIIIIIIIIII`
`lh` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`] |	`100001ssssstttttIIIIIIIIIIIIIIII`
	`mem`[r_s+`I`+1] << 8	
`lw` r_t, `I`(r_s)	r_t = `mem`[r_s+`I`] |	`100011ssssstttttIIIIIIIIIIIIIIII`
	`mem`[r_s+`I`+1] << 8 |	
	`mem`[r_s+`I`+2] << 16 |	
	`mem`[r_s+`I`+3] << 24	
`sb` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t & 0xff	`101000ssssstttttIIIIIIIIIIIIIIII`
`sh` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t & 0xff	`101001ssssstttttIIIIIIIIIIIIIIII`
	`mem`[r_s+`I`+1] = r_t >> 8 & 0xff	
`sw` r_t, `I`(r_s)	`mem`[r_s+`I`] = r_t & 0xff	`101011ssssstttttIIIIIIIIIIIIIIII`
	`mem`[r_s+`I`+1] = r_t >> 8 & 0xff	
	`mem`[r_s+`I`+2] = r_t >> 16 & 0xff	
	`mem`[r_s+`I`+3] = r_t >> 24 & 0xff	

- The memory address used for load/store instructions is the sum of a register and a 16-bit constant (often 0) which is part of the instruction
- This constant is called the offset, allowing us to move across memory in bytes.

Code example: storing and loading a value (no labels)

```
# simple example of load & storing a byte
# we normally use directives and labels
main:
    li        $t0, 42
    li        $t1, 0x10000000    # vvv the brackets mean referring to an address
    sb        $t0, 0($t1)        # store 42 in byte at address 0x10000000
    lb        $a0, 0($t1)        # load $a0 from same address
    li        $v0, 1             # print $a0
    syscall
    li        $a0, '\n'          # print '\n'
    li        $v0, 11
    syscall
    li        $v0, 0             # return 0
    jr        $ra
```

Assembler Directives

```
       .text              # following instructions placed in text
       .data              # following objects placed in data
       .globl             # make symbol available globally
a:     .space 18          # int8_t a[18];
       .align 2           # align next object on 4-byte address
i:     .word 2            # int32_t i = 2;
v:     .word 1,3,5        # int32_t v[3] = {1,3,5};
h:     .half 2,4,6        # int16_t h[3] = {2,4,6};
b:     .byte 7:5          # int8_t b[5] = {7,7,7,7,7};
f:     .float 3.14        # float f = 3.14;
s:     .asciiz "abc"      # char s[4] {'a','b','c','\0'};
t:    .ascii "abc"        # char s[3] {'a','b','c'};
```

- `.text` and `.data` can be mixed throughout your `.s` file, but it's not really good style

SPIM Memory Layout

Region	Address	Notes
text	0x00400000	instructions only; read-only; cannot expand
data	0x10000000	data objects; read/write; can be expanded
stack	0x7fffefff	grows down from that address; read/write
k_text	0x80000000	kernel code; read-only
		only accessible in kernel mode
k_data	0x90000000	kernel data'
		only accessible in kernel mode

Code example: storing and loading a value (with labels)

```
main:
    li        $t0, 42
    li        $t1, x
    sb        $t0, 0($t1)        # store 42 in byte at address 0x10000000
    lb        $a0, 0($t1)        # load $a0 from same address
    li        $v0, 1             # print $a0
    syscall
    li        $a0, '\n'          # print '\n'
    li        $v0, 11
    syscall
    li        $v0, 0             # return 0
    jr        $ra
.data
x: .space     1                   # set aside 1 byte and associate label x with
```

```
                                    # its address
```

Testing Endian-ness

In C:

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint8_t b;
    uint32_t u;

    u = 0x03040506;
    // load first byte of u
    // this is a dereferenced pointer, that points at an 8-bit value,
    // which is the address of u
    b = *(uint8_t *)&u;
    // prints 6 if little-endian
    // and 3 if big-endian
    printf("%d\n", b);
}
```

In MIPS:

```
main:
    li   $t0, 0x03040506
    la   $t1, u
    sw   $t0, 0($t1) # u = 0x03040506; (store $t0, add the address in memory
                                         of the value at $t1)

    lb   $a0, 0($t1) # b = *(uint8_t *)&u; (treat the value of $t1 as an
                                            address and load the value at that
                                            address in memory to $a0)

    li   $v0, 1      # printf("%d", a0);

    syscall

    li   $a0, '\n'   # printf("%c", '\n');
    li   $v0, 11
    syscall


    li   $v0, 0     # return 0
    jr   $ra

    .data
u:
    .space 4
```

Setting a Register to an Address

- The la (load address) instruction is used to set a register to a labelled memory address
  ```
  la $t8, start
  ```
- The memory address will be fixed before the program runs
- For example, if `vec` is the label for memory address `0x10000100` then these two instructions are equivalent:
  ```
  la $t7, vec
  li $t7, 0x10000100
  ```
- In both cases the constant is encoded as part of the instruction(s)
- Neither `la` or `li` access memory – they are very different to the `lw` instruction

Specifying Addresses – some SPIM short cuts

- You can omit the offset 0, in MIPS code:

  ```
  sb $t0, 0($t1)        # store $t0 in byte at address in $t1
  sb $t0, ($t1)         # same
  ```

- For convenience, SPIM allows addresses to be specified in a few other ways and will generate appropriate real MIPS instructions

  ```
  sb $t0, x             # store $t0 in byte at address labelled x
  sb $t1, x+15          # store $t1 15 bytes past address labelled x
  sb $t2, x($t3)        # store $t2 $t3 bytes past address labelled x
  ```

- These are pseudo-instructions
- You can use these short-cuts but they may not help you much
- Most assemblers have similar short cuts for convenience

Global/Static Variables

- Global variables can be accessed by multiple C files
- Global static variables can be accessed by all the functions of a C file
- Static variables is alive only when the program is being run

- Global/static variables need appropriate number of bytes allocated in data segment in .space:

  C:
  ```
  double val;
  char str[20];
  int vec[20];
  ```
  MIPS:
  ```
  val: .space 8
  str: .space 20
  vec: .space 80
  ```

  initialised to 0 by default, other directives allow initialisation to other values

  C:
  ```
  int val = 5;
  int arr[4] = {9,8,7,6};
  char msg[7] = "Hello\n";
  ```
  MIPS:
  ```
  val: .word 5
  arr: .word 9, 8, 7, 6
  msg: .asciiz "Hello\n"
  ```

Add Variables in Memory (uninitialized)

In C:

```c
#include <stdio.h>

int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("\n");
    return 0;
}
```

In MIPS:

```
main:
    li   $t0, 17       # x = 17;
    la   $t1, x
    sw   $t0, 0($t1)   # $t0 will store in the value of $t1, an address in memory

    li   $t0, 25       # y = 25;
    la   $t1, y
    sw   $t0, 0($t1)

    la   $t0, x
    lw   $t1, 0($t0)
    la   $t0, y
    lw   $t2, 0($t0)
    add  $t3, $t1, $t2 # z = x + y
    la   $t0, z
    sw   $t3, 0($t0)

    la   $t0, z
    lw   $a0, 0($t0)
    li   $v0, 1        # printf("%d", z);
    syscall

    li   $a0, '\n'     # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0        # return 0
    jr   $ra

.data
x:  .space 4           # x, y and z are 32 bit variables
y:  .space 4           # we do not declare what is x, y or z
z:  .space 4
```

Add Variables in Memory (initialized)

In C:

```c
#include <stdio.h>

int x = 17, y = 25, z;
int main(void) {
    z = x + y;
    return 0;
}
```

In MIPS:

```
main:
    la   $t0, x
    lw   $t1, 0($t0)

    la   $t0, y
    lw   $t2, 0($t0)

    add  $t3, $t1, $t2 # z = x + y
    la   $t0, z
    sw   $t3, 0($t0)

    la   $t0, z
    lw   $a0, 0($t0)
    li   $v0, 1       # printf("%d", z);
    syscall

    li   $a0, '\n'    # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0       # return 0
    jr   $ra

.data
x:  .word 17          # we declare what is x and y here
y:  .word 25
z:  .space 4
```

Add Variables in Memory (array)

In C:

```c
#include <stdio.h>

int x[] = {17,25,0};
int main(void) {
    x[2] = x[0] + x[1];
    printf("%d", x[2]);
    printf("\n");
    return 0;
}
```

In MIPS:

```
main:
    la   $t0, x
    lw   $t1, 0($t0)
    lw   $t2, 4($t0)   # each array element is 4 bytes
    add  $t3, $t1, $t2 # z = x + y
    sw   $t3, 8($t0)

    lw   $a0, 8($t0)
    li   $v0, 1        # printf("%d", z);
    syscall

    li   $a0, '\n'     # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0        # return 0
    jr   $ra

.data
# int x[] = {17,25,0}
x:  .word 17,25,0
```

Store Value in Array Element (each element is 4 bytes)

In C:

```
#include <stdio.h>

int x[10];

int main(void) {
    x[3] = 17;
}
```

In MIPS:

```
main:
    li   $t0, 3
    mul  $t0, $t0, 4     # $t0 = 3*4 = 12 (for 3rd array element), each element is 4 bytes
    la   $t1, x
    add  $t2, $t1, $t0   # add 12 to start of array to go to the 3rd element
    li   $t3, 17
    sw   $t3, ($t2)      # store word: $t3 = 17 at the value of $t2 (an address) in memory
    # ...
.data
x:  .space 40            # how many bytes in the array
```

Store Value in Array Element (each element is 2 bytes)

In C:

```
#include <stdio.h>

int16_t x[30];

int main(void) {
    x[13] = 23;
}
```

In MIPS:

```
main:
    li   $t0, 13
    mul  $t0, $t0, 2     # $t0 = 13*4 = 52 (for 13th array element), each element is 2 bytes
    la   $t1, x
    add  $t2, $t1, $t0   # add 52 to start of array to go to the 3rd element
    li   $t3, 23
    sh   $t3, ($t2)      # store half: $t3 = 23 at the value of $t2 (an address) in memory
    # ...
.data
x:  .space 60            # how many bytes in the array
```

Printing Array

In C:

```
#include <stdio.h>

int numbers[5] = {3, 9, 27, 81, 243};

int main(void) {
    int i = 0;
    while (i < 5) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

In Simplified C:

```
#include <stdio.h>

int numbers[5] = {3, 9, 27, 81, 243};

int main(void) {
    int i = 0;
loop:
    if (i >= 5) goto end;
        printf("%d", numbers[i]);
        printf("%c", '\n');
        i++;
    goto loop;
end:
    return 0;
}
```

In MIPS:

```
main:
    li   $t0, 0          # int i = 0;
loop:
    bge  $t0, 5, end     # if (i >= 5) goto end;
    la   $t1, numbers    #    int j = numbers[i];
    mul  $t2, $t0, 4
    add  $t3, $t2, $t1
    lw   $a0, 0($t3)      #    printf("%d", j);
    li   $v0, 1
    syscall
    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1     #   i++
    j    loop            # goto loop
end:

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                 # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

Printing Array with Pointers

In C:

```
#include <stdio.h>

int numbers[5] = {3, 9, 27, 81, 243};

int main(void) {
    int *p = &numbers[0];       // pointers store the address
    int *q = &numbers[4];
    while (p <= q) {
        printf("%d\n", *p);
        p++;                    // adding 1 to a pointer is defined in C
    }                           // the pointer moves to the next element
    return 0;
}
```

In Simplified C:

```c
#include <stdio.h>

int numbers[5] = { 3, 9, 27, 81, 243};

int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
loop:
    if (p > q) goto end;
        int j = *p;
        printf("%d", j);
        printf("%c", '\n');
        p++;
    goto loop;
end:
    return 0;
}
```

In MIPS:

```
main:
    la   $t0, numbers     # int *p = &numbers[0];
    la   $t0, numbers     # int *q = &numbers[4]; (not necessary)
    addi $t1, $t0, 16     # add 16 bytes to go to index 4 element
loop:
    bgt  $t0, $t1, end    # if (p > q) goto end;
    lw   $a0, 0($t0)      # int j = *p;
    li   $v0, 1
    syscall
    li   $a0, '\n'         #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 4      # p++
    j    loop             # goto loop
end:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                  # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

In MIPS (faster version – not important though, as that's not the goal of this course for MIPS):

```
main:
    la   $s0, numbers     # int *p = &numbers[0];
    addi $s1, $s0, 16     # int *q = &numbers[4];
loop:
    lw   $a0, ($s0)       # printf("%d", *p);
    li   $v0, 1
    syscall
    li   $a0, '\n'        # printf("%c", '\n');
    li   $v0, 11
    syscall
    addi $s0, $s0, 4      # p++
    ble  $s0, $s1, loop   # if (p <= q) goto loop;

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                  # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

Read 10 Numbers into an Array and then Print them

In C:

```
#include <stdio.h>

int numbers[10] = { 0 };

int main(void) {
    int i;

    i = 0;
    while (i < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[i]);
        i++;
    }

    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

In MIPS:

```
main:

    li   $s0, 0          # i = 0
loop0:
    bge  $s0, 10, end0   # while (i < 10) {

    la   $a0, string0    #   printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5          #   scanf("%d", &numbers[i]);
    syscall              #

    mul  $t1, $s0, 4     #   calculate &numbers[i]
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #
    sw   $v0, ($t3)      #   store entered number in array

    addi $s0, $s0, 1     #   i++;
    j    loop0           # }
end0:

    li   $s0, 0          # i = 0
loop1:
    bge  $s0, 10, end1   # while (i < 10) {

    mul  $t1, $s0, 4     #   calculate &numbers[i]
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #
    lw   $a0, ($t3)      #   load numbers[i] into $a0
    li   $v0, 1          #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'       #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $s0, $s0, 1     #   i++
    j    loop1           # }
end1:

    li   $v0, 0          # return 0
    jr   $ra

.data

numbers:                 # int numbers[10];
     .word 0 0 0 0 0 0 0 0 0 0

string0:
    .asciiz "Enter a number: "
```

Read 10 Numbers into an Array and then Print them in Reverse

In C:

```c
#include <stdio.h>

int numbers[10];

int main() {
    int count;

    count = 0;
    while (count < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[count]);
        count++;
    }

    printf("Reverse order:\n");
    count = 9;
    while (count >= 0) {
        printf("%d\n", numbers[count]);
        count--;
    }

    return 0;
}
```

In MIPS:

```
main:
    li   $s0, 0           # count = 0

read:
    bge  $s0, 10, print   # while (count < 10) {
    la   $a0, string0     # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5           #   scanf("%d", &numbers[count]);
    syscall               #
    mul  $t1, $s0, 4      #   calculate &numbers[count]
    la   $t2, numbers     #
    add  $t1, $t1, $t2    #
    sw   $v0, ($t1)       #   store entered number in array

    addi $s0, $s0, 1      #   count++;
    j    read             # }

print:
    la   $a0, string1     # printf("Reverse order:\n");
    li   $v0, 4
    syscall
```

```
    li   $s0, 9           # count = 9;
next:
    blt  $s0, 0, end1     # while (count >= 0) {

    mul  $t1, $s0, 4      #   printf("%d", numbers[count])
    la   $t2, numbers     #   calculate &numbers[count]
    add  $t1, $t1, $t2    #
    lw   $a0, ($t1)       #   load numbers[count] into $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'         #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $s0, $s0, -1     #   count--;
    j    next             # }
end1:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                 # int numbers[10];
     .word 0 0 0 0 0 0 0 0 0 0

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

Read 10 Numbers into an Array and then Print them in Reverse

In C:

```
#include <stdio.h>

int numbers[10];
int main() {
    int count;
    count = 0;
    while (count < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[count]);
        count++;
    }

    printf("Reverse order:\n");
    count = 9;
    while (count >= 0) {
        printf("%d\n", numbers[count]);
        count--;
    }

    return 0;
}
```

In MIPS:

```
main:
    li   $s0, 0           # count = 0

read:
    bge  $s0, 10, print   # while (count < 10) {
    la   $a0, string0     # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5           #   scanf("%d", &numbers[count]);
    syscall               #
    mul  $t1, $s0, 4      #   calculate &numbers[count]
    la   $t2, numbers     #
    add  $t1, $t1, $t2    #
    sw   $v0, ($t1)       #   store entered number in array

    addi $s0, $s0, 1      #   count++;
    j    read             # }

print:
    la   $a0, string1     # printf("Reverse order:\n");
    li   $v0, 4
    syscall

    li   $s0, 9           # count = 9;
next:
    blt  $s0, 0, end1     # while (count >= 0) {

    mul  $t1, $s0, 4      #   printf("%d", numbers[count])
    la   $t2, numbers     #   calculate &numbers[count]
    add  $t1, $t1, $t2    #
    lw   $a0, ($t1)       #   load numbers[count] into $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'        #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $s0, $s0, -1     #   count--;
    j    next             # }
end1:

    li   $v0, 0           # return 0
    jr   $ra

.data

numbers:                # int numbers[10];
     .word 0 0 0 0 0 0 0 0 0 0

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

Read 10 Numbers into an Array and then Print them in Scaled by 42

In C:

```c
#include <stdio.h>

int
main() {
    int i;
    int numbers[10];

    i = 0;
    while (i < 10) {
        printf("Enter a number: ");
        scanf("%d", &numbers[i]);
        i++;
    }
    i = 0;
    while (i < 10) {
        numbers[i] *= 42;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
    return 0;
}
```

In MIPS:

```
main:
    li   $s0, 0           # i = 0

loop0:
    bge  $s0, 10, end0  # while (i < 10) {
    la   $a0, string0   # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5         # scanf("%d", &numbers[i]);
    syscall             #
    mul  $s1, $s0, 4    # calculate &numbers[i]
    la   $s2, numbers   #
    add  $s1, $s1, $s2  #
    sw   $v0, ($s1)     # store entered number in array

    addi $s0, $s0, 1    # i++;
    j    loop0
end0:
    li   $s0, 0         # i = 0
```

```
loop1:
    bge  $s0, 10, end1  # while (i < 10) {

    mul  $s1, $s0, 4    #
    la   $s2, numbers   # calculate &numbers[i]
    add  $s1, $s1, $s2  #
    lw   $t0, ($s1)     # load numbers[i] into $t0
    mul  $t0, $t0, 42   # numbers[i] *= 42;
    sw   $t0, ($s1)     # store scaled number in array

    addi $s0, $s0, 1    # i++;
    j    loop1
end1:
    li   $s0, 0

loop2:
    bge  $s0, 10, done  # while (i < 10) {

    mul  $s1, $s0, 4    # printf("%d", numbers[i])
    la   $s2, numbers   # calculate &numbers[i]
    add  $s1, $s1, $s2  #
    lw   $a0, ($s1)     # load numbers[i] into $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $s0, $s0, 1    # i++
    j    loop2

done:
    li    $v0, 0        # return 0
    jr    $ra

.data

numbers:
    .space 40           # int numbers[10];

string0:
    .asciiz "Enter a number: "
string1:
    .asciiz "Reverse order:\n"
```

Data Structures and MIPS

- C data structures and their MIPS representations:
  - `char` … as byte in memory, or register
  - `int` … as bytes in memory, or register
  - `double` … as 8 bytes in memory, or $f? register
  - `arrays` … sequence of bytes in memory, elements accessed by index (calculated on MIPS)
  - `structs` … sequence of bytes in memory, accessed by fields (constant offsets on MIPS)
- A `char`, `int` or `double`
  - can be stored in register if local variable and no pointer to it
  - otherwise stored on stack if local variable
  - stored in data segment if global variable

How do addresses work in 2D Arrays?

- We know that in 1D arrays, the size of each element of the array is the size of the type we set the array to
- So in a 1D array, the addresses of the elements in memory are next to each other
- In a 2D array, it is quite similar:

Consider this code: `2d_array_element_address.c`

```c
#include <stdio.h>

#define X 3
#define Y 4

int main(void) {
    int array[X][Y];

    for (int x = 0; x < X; x++) {
        for (int y = 0; y < Y; y++) {
            array[x][y] = x + y;
        }
    }

    for (int x = 0; x < X; x++) {
        for (int y = 0; y < Y; y++) {
            printf("%d ", array[x][y]);
        }
        printf("\n");
    }

    printf("sizeof array[2][3] = %lu\n", sizeof array[2][3]);
    printf("sizeof array[1] = %lu\n", sizeof array[1]);
    printf("sizeof array = %lu\n", sizeof array);

    printf("&array=%p\n", &array);
    for (int x = 0; x < X; x++) {
        printf("&array[%d]=%p\n", x, &array[x]);
        for (int y = 0; y < Y; y++) {
            printf("&array[%d][%d]=%p\n", x, y, &array[x][y]);
        }
    }
}
```

It prints:

```
0 1 2 3
1 2 3 4
2 3 4 5
sizeof array[2][3] = 4
sizeof array[1] = 16
sizeof array = 48
&array=0x7ffc7cafc740
&array[0]=0x7ffc7cafc740
&array[0][0]=0x7ffc7cafc740
&array[0][1]=0x7ffc7cafc744
&array[0][2]=0x7ffc7cafc748
&array[0][3]=0x7ffc7cafc74c
&array[1]=0x7ffc7cafc750
&array[1][0]=0x7ffc7cafc750
&array[1][1]=0x7ffc7cafc754
&array[1][2]=0x7ffc7cafc758
&array[1][3]=0x7ffc7cafc75c
&array[2]=0x7ffc7cafc760
&array[2][0]=0x7ffc7cafc760
&array[2][1]=0x7ffc7cafc764
&array[2][2]=0x7ffc7cafc768
&array[2][3]=0x7ffc7cafc76c
```

- They are all next to each other in memory! WOW!
- So instead of using array indexing, we can emulate it here:

```c
#include <stdio.h>
#include <stdint.h>


uint32_t array[3][4] = {{10, 11, 12, 13}, {14, 15, 16, 17}, {18, 19, 20, 21}};

int main(void) {
    // use a typecast to assign array address to integer variable i
    uint64_t i = (uint64_t)&array;

    // i += (2 * 16) + 2 * 4. For array[1+2][1+2]
    i += (2 * sizeof array[0]) + 2 * sizeof array[0][0];

    // use a typecast to assign  i to a pointer vaiable
    uint32_t *y = (uint32_t *)i;

    printf("*y = %d\n", *y); // prints 20
}
```

Print a 2D Array

In C:

```c
#include <stdio.h>

int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};

int main(void) {
    int i = 0;
    while (i < 3) {
        int j = 0;
        while (j < 5) {
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        }
        printf("%c", '\n');
        i++;
    }
    return 0;
}
```

In Simple C:

```c
#include <stdio.h>

int numbers[3][5] = {{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};

int main(void) {
    int i = 0;
loop1:
    if (i >= 3) goto end1;
        int j = 0;
    loop2:
        if (j >= 5) goto end2;
            printf("%d", numbers[i][j]);
            printf("%c", ' ');
            j++;
        goto loop2;
    end2:
        printf("%c", '\n');
        i++;
    goto loop1;
end1:
    return 0;
}
```

In MIPS:

```
main:
    li   $s0, 0          # int i = 0; (ROW)
loop1:
    bge  $s0, 3, end1    # if (i >= 3) goto end1;
    li   $s1, 0          #    int j = 0; (COLUMN)
loop2:
    bge  $s1, 5, end2    #    if (j >= 5) goto end2;
    la   $t0, numbers    #        printf("%d", numbers[i][j]);
    mul  $t1, $s0, 20    # 5*4 = 20 (so we can jump across rows)
    add  $t2, $t1, $t0
    mul  $t3, $s1, 4     # (so we can jump across columns)
    add  $t4, $t3, $t2
    lw   $a0, ($t4)
    li   $v0, 1
    syscall
    li   $a0, ' '        #       printf("%c", ' ');
    li   $v0, 11
    syscall
    addi $s1, $s1, 1     #       j++;
    j    loop2           #    goto loop2;
end2:
    li   $a0, '\n'       #    printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $s0, $s0, 1     #   i++
    j    loop1           # goto loop1
end1:

    li   $v0, 0          # return 0
    jr   $ra

.data
# int numbers[3][5] =
{{3,9,27,81,243},{4,16,64,256,1024},{5,25,125,625,3125}};
numbers:
     .word  3, 9, 27, 81, 243, 4, 16, 64, 256, 1024, 5, 25, 125, 625, 3125
```

Unaligned Access

- Sometimes we may get errors due to unaligned access when we do `sb`, `sh` or `sw`!
- This means, for addresses, that we are trying to store an address that is not aligned on a 4-byte boundary
- For example: unalign.c

```
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uint8_t bytes[32]
    uint32_t *i = (unint32_t *)&bytes[12];
    // illegal store - not aligned on a 4-byte boundary
    *i = 0x03040506;
    printf("%d\n", bytes[12]); // prints 6 if little endian machine
}                              // cause its still actually a 1 byte space
```

Will it Align?

```
main:
    li   $t0, 1

    sb   $t0, v1  # will succeed because no alignment needed
    sh   $t0, v1  # will fail because v1 is not 2-byte aligned
    sw   $t0, v1  # will fail because v1 is not 4-byte aligned

    sh   $t0, v2  # will succeeed because v2 is 2-byte aligned
    sw   $t0, v2  # will fail because v2 is not 4-byte aligned

    sh   $t0, v3  # will succeeed because v3 is 2-byte aligned
    sw   $t0, v3  # will fail because v3 is not 4-byte aligned

    sh   $t0, v4  # will succeeed because v4 is 2-byte aligned
    sw   $t0, v4  # will succeeed because v4 is 4-byte aligned

    sw   $t0, v5  # will succeeed because v5 is 4-byte aligned

    sw   $t0, v6  # will succeeed because v6 is 4-byte aligned

    li   $v0, 0
    jr   $ra   # return

    .data
    # data will be aligned on a 4-byte boundary
    # most likely on at least a 128-byte boundary
    # but safer to just add a .align directive
    .align 2
    .space 1
v1: .space 1
v2: .space 4
v3: .space 2
v4: .space 4
    .space 1
    .align 2 # ensure e is on a 4 (2**2) byte boundary
v5: .space 4
    .space 1
v6: .word 0  # word directive aligns on 4 byte boundary
```

- Notice here that .align 2 is used. This allows that a space is on a 2^2 byte boundary

Structs in C

- C struct definitions effectively define a new type. This is a simple struct:

```c
#include <stdio.h>
#include <stdint.h>

struct details {
    uint16_t  postcode;
    char      first_name[7];
    uint32_t  zid;
};

struct details student = {2052, "Alice", 5123456};

int main(void) {
    printf("%d", student.zid);
    putchar(' ');
    printf("%s", student.first_name);
    putchar(' ');
    printf("%d", student.postcode);
    putchar('\n');
    return 0;
}
```

- The way that the fields of a struct are organised in memory is done to ensure each field and their type is aligned! THIS IS AUTOMATICALLY DONE IN C, BUT NOT IN MIPS!
- So we can get gaps between the fields in memory and sometimes not
- Here is an example where there is padding to allow for alignment:

```c
#include <stdio.h>
#include <stdint.h>

struct s1 {
    uint32_t   i0;
    uint32_t   i1;
    uint32_t   i2;
    uint32_t   i3;
};

struct s2 {
    uint8_t    b;
    uint64_t   l;
};

int main(void) {
    struct s1 v1;

    printf("&v1      = %p\n", &v1);
    printf("&(v1.i0) = %p\n", &(v1.i0));
    printf("&(v1.i1) = %p\n", &(v1.i1));
    printf("&(v1.i2) = %p\n", &(v1.i2));
    printf("&(v1.i3) = %p\n", &(v1.i3));

    printf("\nThis shows struct padding\n");

    struct s2 v2;
    printf("&v2      = %p\n", &v2);
    printf("&(v2.b)  = %p\n", &(v2.b));
    printf("&(v2.l)  = %p\n", &(v2.l));
}
```

Prints:

```
&v1      = 0x7ffc5c968f20    // each address going up by 4 bytes
&(v1.i0) = 0x7ffc5c968f20
&(v1.i1) = 0x7ffc5c968f24
&(v1.i2) = 0x7ffc5c968f28
&(v1.i3) = 0x7ffc5c968f2c

This shows struct padding
&v2      = 0x7ffc5c968f40
&(v2.b)  = 0x7ffc5c968f40    // has 8 byte alignment since v2.l is 8 bytes
&(v2.l)  = 0x7ffc5c968f48    // and must be aligned
```

- Here is an example where padding does and does not occur:

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

void print_bytes(void *v, int n);

struct s1 {
    uint8_t    c1;
    uint32_t   l1;
    uint8_t    c2;
    uint32_t   l2;
    uint8_t    c3;
    uint32_t   l3;
    uint8_t    c4;
    uint32_t   l4;
};

struct s2 {
    uint32_t   l1;
    uint32_t   l2;
    uint32_t   l3;
    uint32_t   l4;
    uint8_t    c1;
    uint8_t    c2;
    uint8_t    c3;
    uint8_t    c4;
};

int main(void) {
    struct s1 v1;
    struct s2 v2;

    printf("sizeof v1 = %lu\n", sizeof v1);
    printf("sizeof v2 = %lu\n", sizeof v2);

    printf("alignment rules mean struct s1 is padded\n");

    printf("&(v1.c1) = %p\n", &(v1.c1));
    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.c2) = %p\n", &(v1.c2));
    printf("&(v1.l2) = %p\n", &(v1.l2));

    printf("struct s2 is not padded\n");

    printf("&(v1.l1) = %p\n", &(v1.l1));
    printf("&(v1.l2) = %p\n", &(v1.l2));
    printf("&(v1.l3) = %p\n", &(v1.l3));
    printf("&(v1.l4) = %p\n", &(v1.l4));
    printf("&(v2.c1) = %p\n", &(v2.c1));
    printf("&(v2.c2) = %p\n", &(v2.c2));
    printf("&(v2.c3) = %p\n", &(v2.c3));
    printf("&(v2.c4) = %p\n", &(v2.c4));

}
```

Prints:

```
sizeof v1 = 32    // they are different sizes!
sizeof v2 = 20

alignment rules mean struct s1 is padded
&(v1.c1) = 0x7ffcf9fd4000     // divisible by 1
&(v1.l1) = 0x7ffcf9fd4004     // this is the next address divisible by 4
&(v1.c2) = 0x7ffcf9fd4008     // divisible by 1
&(v1.l2) = 0x7ffcf9fd400c     // this is the next address divisible by 4

struct s2 is not padded
&(v1.l1) = 0x7ffcf9fd4004     // divisible by 4
&(v1.l2) = 0x7ffcf9fd400c     // divisible by 4
&(v1.l3) = 0x7ffcf9fd4014     // divisible by 4
&(v1.l4) = 0x7ffcf9fd401c     // divisible by 4
&(v2.c1) = 0x7ffcf9fd4050     // this is the next address divisible by 1
&(v2.c2) = 0x7ffcf9fd4051     // divisible by 1
&(v2.c3) = 0x7ffcf9fd4052     // divisible by 1
&(v2.c4) = 0x7ffcf9fd4053     // divisible by 1
```

Structs in MIPS

- There is no automatic padding in MIPS!
- Do it manually, we also have to take notice of the offset
- Here is a real example of MIPS code where there is no padding

```
DETAILS_POSTCODE    = 0
DETAILS_FIRST_NAME  = 2
DETAILS_ZID         = 4

main:
    la   $t0, student           # printf("%d", student.zid);
    add  $t1, $t0, DETAILS_ZID
    lw   $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%s", student.first_name);
    add  $a0, $t0, DETAILS_FIRST_NAME
    li   $v0, 4
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    add  $t1, $t0, DETAILS_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, '\n'              #   putchar('\n');
    li   $v0, 11
    syscall
```

```
    li   $v0, 0                # return 0
    jr   $ra

.data

student:                # struct details student = {2052, "Alice", 5123456};
     .half 2052
     .asciiz "Andrew"
     .word 5123456
```

- .half is 2 bytes
- The string can be anywhere – has no alignment (7 bytes, don't for \0)
- .word is 4 bytes and must be aligned – so this WILL NOT WORK

- To fix this:

```
DETAILS_POSTCODE   = 0
DETAILS_FIRST_NAME = 2
DETAILS_ZID        = 12

main:
    la   $t0, student           # printf("%d", student.zid);
    add  $t1, $t0, DETAILS_ZID
    lw   $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%s", student.first_name);
    add  $a0, $t0, DETAILS_FIRST_NAME
    li   $v0, 4
    syscall

    li   $a0, ' '               #   putchar(' ');
    li   $v0, 11
    syscall

    la   $t0, student           # printf("%d", student.postcode);
    add  $t1, $t0, DETAILS_POSTCODE
    lhu  $a0, ($t1)
    li   $v0, 1
    syscall

    li   $a0, '\n'              #   putchar('\n');
    li   $v0, 11
    syscall

    li   $v0, 0                 # return 0
    jr   $ra

.data

student:                # struct details student = {2052, "Alice", 5123456};
     .half 2052
     .asciiz "Andrew"
     .space 3           # struct padding to ensure zid field is ona 4-byte boundary
     .word 5123456
```

- There is a `.space 3` in `.data` and `DETAILS_ZID` is now 12

- 2 + 7 (don't forget the \0) + 3 = 12 so `.word` is 12 bytes after the start of the struct memory address

man Guide

- man man
- man -k printf
- man 1 printf – executable programs or shell commands
- man 3 printf – library calls (functions within program libraries)

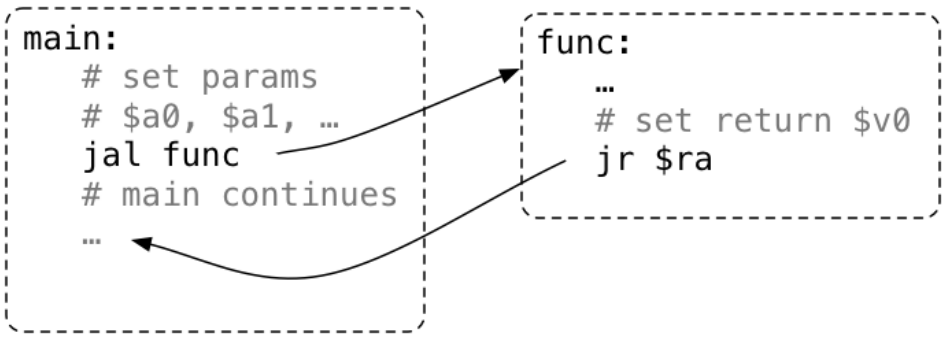
MIPS Functions

The role of Functions

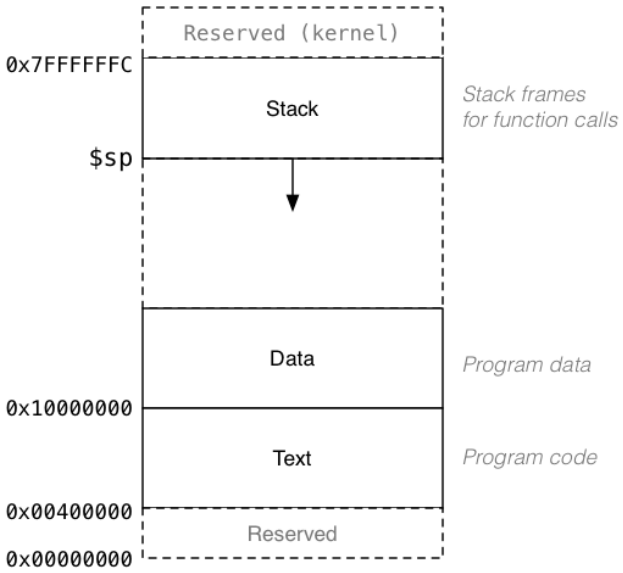
- When we call a function:
  - the arguments are evaluated and set up for a function
  - control is transferred to the code of the function
  - local variables are created
  - the function code is executed in this environment
  - the return value is set up
  - control transfers back to where the function was called from
  - the caller receives the return value

Functions in MIPS – How it works and common conventions

- Load arguments into `$a0`, `$a1`, `$a2`, `$a3`
- `jal function` sets `$ra` to the next line in code after it (e.g. PC + 4), and jumps to the function
- Function puts return value in `$v0`
- Return to caller using `jr $ra`



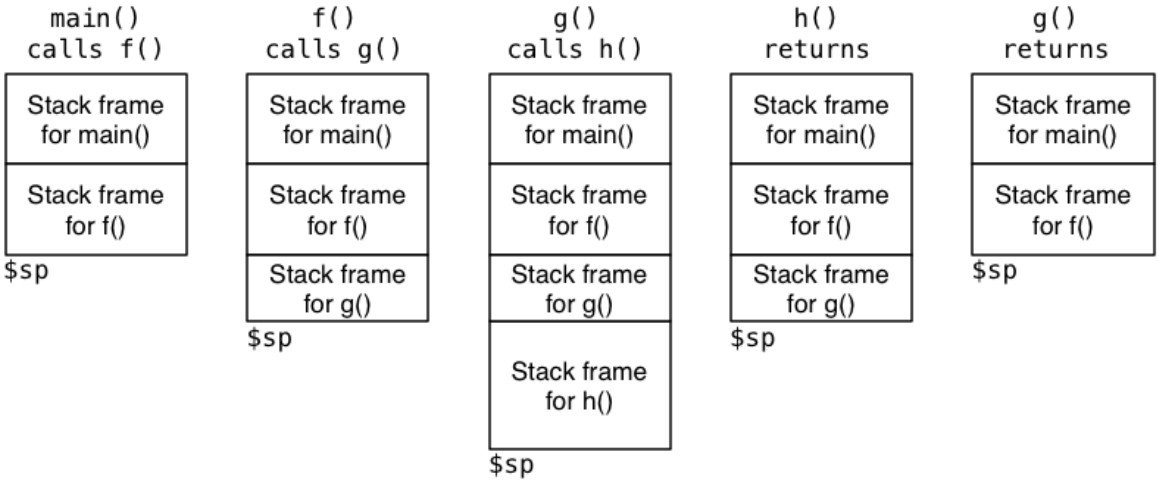
- If function changes `$sp`, `$fp`, `$s0..$s8` it restores their value – essentially, callers assume it is unchanged by call (`jl`)
- A function can also destroy the value of other registers, e.g. `$t0..$t9` – essentially, callers assume it is changed by call (`jl`)



How Stack works

- $sp (stack pointer) is initialised by operating system
- Always 4-byte aligned (divisible by 4)
- Points a currently used (4-byte) word
- Grows downward
- A function can do this to allocated 40 bytes in the stack:
  ```
  sub $sp, $sp, 40 # move stack pointer down
  ```
- A function can change $sp, but MUST change it back to its original value once that function is finished
- So if you allocated 40 bytes, before returning (jr $ra) do:
  ```
  add $sp, $sp, 40 # move stack pointer back
  ```

How stack changes as functions are called and returned



Stack – Where is it in memory

- Data associated with a function call placed on a stack
- Remember the nature of stack: FILO (first in – last out)
- Down: closer to memory address `0x0000000`

- Up: closer to memory address `0xFFFFFFFF`

How is using the stack useful?

- A function that calls another function MUST preserve `$ra`
- Functions in functions will cause `$ra` to change due to `jal`, which is why we store them on the stack for easy access when we need them!

Example: Function with a No Parameters or Return Value

In C:

```c
#include <stdio.h>

void f(void);

int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    return 0;
}

void f(void) {
    printf("in function f\n");
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4    # move stack pointer down to make room
    sw   $ra, 0($sp)     # save $ra on $stack

    la   $a0, string0    # printf("calling function f\n");
    li   $v0, 4
    syscall

    jal  f               # set $ra to following address

    la   $a0, string1    # printf("back from function f\n");
    li   $v0, 4
    syscall

    lw   $ra, 0($sp)     # recover $ra from $stack
    addi $sp, $sp, 4     # move stack pointer back to what it was
```

```
    li   $v0, 0          # return 0 from function main
    jr   $ra             #


f:
    la   $a0, string2    # printf("in function f\n");
    li   $v0, 4
    syscall
    jr   $ra             # return from function f


    .data
string0:
    .asciiz "calling function f\n"
string1:
    .asciiz "back from function f\n"
string2:
    .asciiz "in function f\n"
```

Example: Function with a Return Value but No Parameters

- By convention, function return value is passed back in $v0

In C:

```
#include <stdio.h>

int answer(void);

int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}

int answer(void) {
    return 42;
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4 # move stack pointer down to make room
    sw   $ra, 0($sp)  # save $ra on $stack

    jal  answer       # call answer, return value will be in $v0

    move $a0, $v0     # printf("%d", a);
    li   $v0, 1
    syscall

    li   $a0, '\n'    # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)  # recover $ra from $stack
```

```
    addi $sp, $sp, 4  # move stack pointer back up to what it was when main
called
    jr   $ra          #


answer: # code for function answer
    li   $v0, 42      #
    jr   $ra          # return from answer
```

Example: Function with a Return Value and Parameters

- By convention, first 4 function parameters are passed in `$a0 $a1 $a2 $a3`
- If there is more, they are passed onto the stack but we don't go through that in COMP1521

In C:

```c
#include <stdio.h>

int sum_product(int a, int b);
int product(int x, int y);

int main(void) {
    int z = sum_product(10, 12);
    printf("%d\n", z);
    return 0;
}

int sum_product(int a, int b) {
    int p = product(6, 7);
    return p + a + b;
}

int product(int x, int y) {
    return x * y;
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4   # move stack pointer down to make room
    sw   $ra, 0($sp)    # save $ra on $stack

    li   $a0, 10         # sum_product(10, 12);
    li   $a1, 12
    jal  sum_product

    move $a0, $v0       # printf("%d", z);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)    # recover $ra from $stack
    addi $sp, $sp, 4    # move stack pointer back up to what it was when main
called

    li   $v0, 0         # return 0 from function main
    jr   $ra            # return from function main
```

```
sum_product:
    addi $sp, $sp, -12  # move stack pointer down to make room
    sw   $ra, 8($sp)    # save $ra on $stack
                        # has address of move $a0, $v0
    sw   $a1, 4($sp)    # save $a1 on $stack
    sw   $a0, 0($sp)    # save $a0 on $stack

    li   $a0, 6         # product(6, 7);
    li   $a1, 7
    jal  product

    lw   $a1, 4($sp)    # restore $a1 from $stack
    lw   $a0, 0($sp)    # restore $a0 from $stack

    add  $v0, $v0, $a0  # add a and b to value returned in $v0
    add  $v0, $v0, $a1  # and put result in $v0 to be returned

    lw   $ra, 8($sp)    # restore $ra from $stack
                        # which has address of move $a0, $v0
    addi $sp, $sp, 12   # move stack pointer back up to what it was when main
called

    jr   $ra            # return from sum_product


product:                # product doesn't call other functions
                        # so it doesn't need to save any registers
    mul  $v0, $a0, $a1  # return argument * argument 2
    jr   $ra            # $ra here is address of lw   $a1, 4($sp)
```

Example: two_powerful

In C:

```c
#include <stdio.h>

void two(int i);

int main(void) {
    two(1);
}

void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4   # move stack pointer down to make room
    sw   $ra, 0($sp)    # save $ra on $stack

    li   $a0, 1         # two(1);
    jal  two

    lw   $ra, 0($sp)    # recover $ra from $stack
    addi $sp, $sp, 4    # move stack pointer back up to what it was when main
called

    jr   $ra            # return from function main

two:
    addi $sp, $sp, -8   # move stack pointer down to make room
    sw   $ra, 4($sp)    # save $ra on $stack
    sw   $a0, 0($sp)    # save $a0 on $stack

    bge  $a0, 1000000, print
    mul  $a0, $a0, 2    # restore $a0 from $stack
    jal  two

print:
    lw   $a0, 0($sp)    # restore $a0 from $stack
    li   $v0, 1         # printf("%d");
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 4($sp)    # restore $ra from $stack
    addi $sp, $sp, 8    # move stack pointer back up to what it was when main
called

    jr   $ra            # return from two
```

Example: squares - store first 10 squares into an array which is a local variable then print them from array. THIS USES THE STACK TO STORE THE ARRAY ELEMENTS!

In C:

```c
#include <stdio.h>

int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
    i = 0;
    while (i < 10) {
        printf("%d", squares[i]);
        printf("%c",'\n');
        i++;
    }
    return 0;
}
```

In MIPS:

```
main:
    addi $sp, $sp, -40  # move stack pointer down to make room
                        # to store array numbers on stack
    li   $t0, 0         # i = 0
loop0:
    bge  $t0, 10, end0  # while (i < 10) {

    mul  $t1, $t0, 4    #    calculate &numbers[i]
    add  $t2, $t1, $sp  #
    mul  $t3, $t0, $t0  #    calculate i * i
    sw   $t3, ($t2)     #    store in array

    addi $t0, $t0, 1    #    i++;
    j    loop0          # }
end0:
    li   $t0, 0         # i = 0
loop1:
    bge  $t0, 10, end1  # while (i < 10) {

    mul  $t1, $t0, 4
    add  $t2, $t1, $sp  #   calculate &numbers[i]
    lw   $a0, ($t2)     #   load numbers[i] into $a0
    li   $v0, 1         #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'      #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1    #   i++
    j    loop1          # }
end1:

    addi $sp, $sp, 40   # move stack pointer back up to what it was when main called
    li   $v0, 0         # return 0 from function main
    jr   $ra            #
```

Example: pointer

In C:

```c
#include <stdio.h>

int answer = 42;

int main(void) {
    int i;
    int *p;

    p = &answer;
    i = *p;
    printf("%d\n", i); // prints 42
    *p = 27;
    printf("%d\n", answer); // prints 27

    return 0;
}
```

In MIPS:

```
main:
    la   $t0, answer     # p = &answer;

    lw   $t1, ($t0)      # i = *p;

    move $a0, $t1        # printf("%d\n", i);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $t2, 27         # *p = 27;
    sw   $t2, ($t0)      #

    lw   $a0, answer     # printf("%d\n", answer);
    li   $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li   $v0, 0          # return 0 from function main
    jr   $ra             #

    .data
answer:
    .word 42             # int answer = 42;
```

Example: strlen_array

In C:

```c
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

In Simple C:

```c
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
loop:
    if (s[length] == 0) goto end;
       length++;
    goto loop;
end:
    return length;
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4    # move stack pointer down to make room
    sw   $ra, 0($sp)     # save $ra on $stack

    la   $a0, string     # my_strlen("Hello");
    jal  my_strlen

    move $a0, $v0        # printf("%d", i);
    li   $v0, 1
    syscall
```

```
    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)    # recover $ra from $stack
    addi $sp, $sp, 4    # move stack pointer back up to what it was when main
called

    li   $v0, 0         # return 0 from function main
    jr   $ra            #


my_strlen:              # length in t0, s in $a0
    li   $t0, 0
loop:                   # while (s[length] != 0) {
    add  $t1, $a0, $t0  #   calculate &s[length]
    lb   $t2, 0($t1)    #   load s[length] into $t2
    beq  $t2, 0, end    #
    addi $t0, $t0, 1    #   length++;
    j    loop           # }
end:
    move $v0, $t0       # return length
    jr   $ra

    .data
string:
    .asciiz "Hello"
```

Example: strlen_pointer

In C:

```c
#include <stdio.h>

int my_strlen(char *s);

int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (*s != 0) {
        length++;
        s++;
    }
    return length;
}
```

In MIPS:

```
main:
    addi $sp, $sp, -4   # move stack pointer down to make room
    sw   $ra, 0($sp)    # save $ra on $stack

    la   $a0, string    # my_strlen("Hello");
    jal  my_strlen

    move $a0, $v0       # printf("%d", i);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    lw   $ra, 0($sp)    # recover $ra from $stack
    addi $sp, $sp, 4    # move stack pointer back up to what it was when main
called

    jr   $ra            # return from function main

my_strlen:              # length in t0, s in $a0
    li   $t0, 0
loop:                   #
    lb   $t1, 0($a0)    # load *s into $t1
    beq  $t1, 0, end    #
    addi $t0, $t0, 1    # length++
    addi $a0, $a0, 1    # s++
    j    loop           #
end:
    move $v0, $t0       # return length
    jr   $ra

    .data
string:
    .asciiz "Hello Andrew"
```

Frame Pointers

- Frame pointer, $fp, is a second register pointing to stack
- By convention set to point at start of stack frame
- Provides a fixed point during function code execution
- Useful for functions which grow stack (change $sp) during execution
- Makes it easier for debuggers to forensically analyse stack, such as if you want to print stack backtrace after error
- Frame pointer is optional in COMP1521 and generally
- Often omitted when fast execution or small code is a priority

Example: Frame Pointer Use

In C:

```c
#include <stdio.h>

void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}
```

In MIPS:

```
f:
    addi $sp, $sp, -12  # move stack pointer down to make room
    sw   $fp, 8($sp)    # save $fp on $stack
    sw   $ra, 4($sp)    # save $ra on $stack
    sw   $a0, 0($sp)    # save $a0 on $stack
    addi $fp, $sp, 12   # have frame pointer at start of stack frame

    li   $v0, 5         # scanf("%d", &length);
    syscall

    mul  $v0, $v0, 4    # calculate array size
    sub  $sp, $sp, $v0  # move stack_pointer down to hold array

    # ... more code ...

    lw   $ra, -8($fp)   # restore $ra from stack
    move $sp, $fp       # move stack pointer backup  to what it was when main
called
    lw   $fp, -4($fp)   # restore $fp from $stack
    jr   $ra            # return
```

Invalid C

- Accessing memory where you are not supposed to is BAD!
- There are lots of security issues with this

Example: stack_inspect.c

```c
#include <stdio.h>
#include <stdint.h>

/*
$ clang stack_inspect.c
$ a.out
 0: Address 0x7ffe1766c304 contains 3           <- a[0]
 1: Address 0x7ffe1766c308 contains 5           <- x
 2: Address 0x7ffe1766c30c contains 2a          <- b
 3: Address 0x7ffe1766c310 contains 1766c330    <- f  frame pointer (64 bit)
 4: Address 0x7ffe1766c314 contains 7ffe
 5: Address 0x7ffe1766c318 contains 40120c      <- f return address
 6: Address 0x7ffe1766c31c contains 0
 7: Address 0x7ffe1766c320 contains 22
 8: Address 0x7ffe1766c324 contains 25
 9: Address 0x7ffe1766c328 contains 9           <- a
10: Address 0x7ffe1766c32c contains 0
11: Address 0x7ffe1766c330 contains 401220      <- main return address
12: Address 0x7ffe1766c334 contains 0
13: Address 0x7ffe1766c338 contains c7aca09b    <- main frame pointer (64 bit)
14: Address 0x7ffe1766c33c contains 7ff3
15: Address 0x7ffe1766c340 contains 0
*/

void f(int b) {
    int x = 5;
    uint32_t a[1] = { 3 };

    for (int i = 0; i < 16; i++)
        printf("%2d: Address %p contains %x\n", i, &a[i], a[0 + i]);
}

int main(void) {
    int a = 9;
    printf("function main is at address %p\n", &main);
    printf("function f is at address %p\n", &f);
    f(42);
    return 0;
}
```

Example: invalid0.c

```
Run at CSE like this


$ clang invalid0.c -o invalid0
$ ./invalid0
42 77 77 77 77 77 77 77 77 77
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int a[10];
    int b[10];
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("b[0] is at address %p\n", &b[0]);
    printf("b[9] is at address %p\n", &b[9]);

    for (int i = 0; i < 10; i++) {
        a[i] = 77;
    }

    // loop writes to b[10] .. b[12] which don't exist -
    // with gcc 7.3 on x86_64/Linux
    // b[12] is stored where a[0] is stored
    // with gcc 7 on CSE lab machines
    // b[10] is stored where a[0] is stored

    for (int i = 0; i <= 12; i++) {
        b[i] = 42;
    }

    // prints 42 77 77 77 77 77 77 77 77 77 on x86_64/Linux
    // prints 42 42 42 77 77 77 77 77 77 77 at CSE
    for (int i = 0; i < 10; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
```

Example: invalid1.c

```
Run at CSE like this


$ clang invalid1.c -o invalid1
$ ./invalid1
i is at address 0x7ffe2c01cd58
a[0] is at address 0x7ffe2c01cd30
a[9] is at address 0x7ffe2c01cd54
a[10] would be stored at address 0x7ffe2c01cd58


doesn't terminate
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i;
    int a[10];
    printf("i is at address %p\n", &i);
    printf("a[0] is at address %p\n", &a[0]);
    printf("a[9] is at address %p\n", &a[9]);
    printf("a[10] would be stored at address %p\n", &a[10]);

    // loop writes to a[10] .. a[11] which don't exist -
    // but with gcc 7 on x86_64/Linux
    // i would be stored where a[11] is stored

    for (i = 0; i <= 11; i++) {
        a[i] = 0;
    }

    return 0;
}
```

Example: invalid2.c

```
Run at CSE like this


$ clang -Wno-everything  invalid2.c -o invalid2
$ ./invalid2
answer=42
```

```c
#include <stdio.h>

void f(int x);

int main(void) {
    int answer = 36;
    printf("answer is stored at address %p\n", &answer);

    f(5);
    printf("answer=%d\n", answer); // prints 42 not 36

    return 0;
}

void f(int x) {
    int a[10];

    // a[18] doesn't exist
    // with clang at CSE variable answer in main
    // happens to be where a[19] would be

    printf("a[18] would be stored at address %p\n", &a[18]);

    a[18] = 42;
}
```

Example: invalid3.c

```
Run at CSE like this


$ clang invalid3.c -o invalid3
$ ./invalid3

I will never be printed.
argc was 1
$
```

```c
#include <stdio.h>
#include <stdlib.h>

void f(void);

int main(int argc, char *argv[]) {
    f();

    if (argc > 0) {
        printf("I will always be printed.\n");
    }

    if (argc <= 0) {
        printf("I will never be printed.\n");
    }

    printf("argc was %d\n", argc);
    return 0;
}

void f() {
    int a[10];

    // function f has it return address on the stack
    // the call of function f from main should return to
    // the next statement which is:  if (argc > 0)
    //
    // with clang at CSE  f's return address is stored where a[12] would be
    //
    // so changing a[12] changes where the function returns
    //
    // adding 12 to a[12] happens to cause it to return several statements
later
    // at the printf("I will never be printed.\n");

    a[12] += 12;
}
```

Example: invalid4.c

```
Run at CSE like this


$ clang invalid4.c -o invalid4
$ ./invalid4
authenticated is at address 0xff94bf44
password is at address 0xff94bf3c

Enter your password: 123456789

Welcome. You are authorized.
$
```

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int authenticated = 0;
    char password[8];

    printf("authenticated is at address %p\n", &authenticated);
    printf("password[8] would be at address %p\n", &password[8]);

    printf("Enter your password: ");
    int i = 0;
    int ch = getchar();
    while (ch != '\n' && ch != EOF) {
        password[i] = ch;
        ch = getchar();
        i = i + 1;
    }
    password[i] = '\0';

    if (strcmp(password, "secret") == 0) {
        authenticated = 1;
    }

    // a password longer than 8 characters will overflow the array password
    // the variable authenticated is at the address where
    // where password[8] would be and gets overwritten
    //
    // This allows access without knowing the correct password

    if (authenticated) {
        printf("Welcome. You are authorized.\n");
    } else {
        printf("Welcome. You are unauthorized.  Your death will now be
implemented.\n");
        printf("Welcome. You will experience a tingling sensation and then
death. \n");
        printf("Remain calm while your life is extracted.\n");
    }

    return 0;
}
```

Files

What is an Operating System (OS)?

- Sits between the user and the hardware
- Acts as a virtual machine to each user
- This virtual machine is MUCH simpler than real machine and can be consistent across different hardware
- Can coordinate/share access to resources between users
- Can provide privileges/security

What does the Operating System need from the hardware?

- OS must provide a privileged mode:
    - Can access all memory/hardware
    - OS (kernel) runs in privileged mode
    - Allows transfer to running code on a non-privileged mode
- OS must also provide a non-privileged mode:
    - Prevents access to hardware
    - Limits access to memory
    - Provides mechanism to make requests to OS
- OS requests are called system calls
    - System calls transfer execution back to kernel code in privileged mode

What is a System Call?

- Allow programs to make hardware requests
- System call transfers execution to OS code in privileged mode
    - Includes arguments specifying details of request being made
    - OS checks if operation is valid and permitted
    - OS carries out operation
    - Transfers execution back to user code in non-privileged mode
- Different OS, have different system calls
- Operations provided by system calls include:
    - read/write bytes to a file
    - request more memory
    - create a process (run a program)
    - terminate a process
    - send or receive information via a network

System Call in SPIM

- SPIM is a virtual machine that can execute MIPS program (essentially a tiny OS)
- Small number of SPIM system calls for I/O and memory allocation
- Access via `syscall`
- MIPS programs running on real hardware also use `syscall`
- SPIM system calls are designed for students writing tiny programs
    - e.g. SPIM system call 1 – prints an integer
- System calls on real operating systems more general
    - might be: write n bytes
- Library system calls are more general e.g. `printf`

Example: Hello World (DIRECT syscall)

```c
#include <unistd.h>

int main(void) {
    char bytes[16] = "Hello, Andrew!\n";

    // argument 1 to syscall is  system call number, 1 == write
    // remaining arguments are specific to each system call

    // write system call takes 3 arguments:
    //   1) file descriptor, 1 == stdout
    //   2) memory address of first byte to write
    //   3) number of bytes to write

    syscall(1, 1, bytes, 15); // prints Hello, Andrew! on stdout

    return 0;
}
```

Example: Read and write system calls to copy stdin to stdout (DIRECT syscall)

```c
#include <unistd.h>

int main(void) {
    // copy stdin to stdout with read & write syscalls
    while (1) {
        char bytes[4096];

        // system call number 0 == read
        // read system call takes 3 arguments:
        //   1) file descriptor, 1 == stdin
        //   2) memory address to put bytes read
        //   3) maximum number of bytes read
        // returns number of bytes actually read

        long bytes_read = syscall(0, 0, bytes, 4096);

        if (bytes_read <= 0) {
            break;
        }

        syscall(1, 1, bytes, bytes_read); // prints bytes to stdout
    }

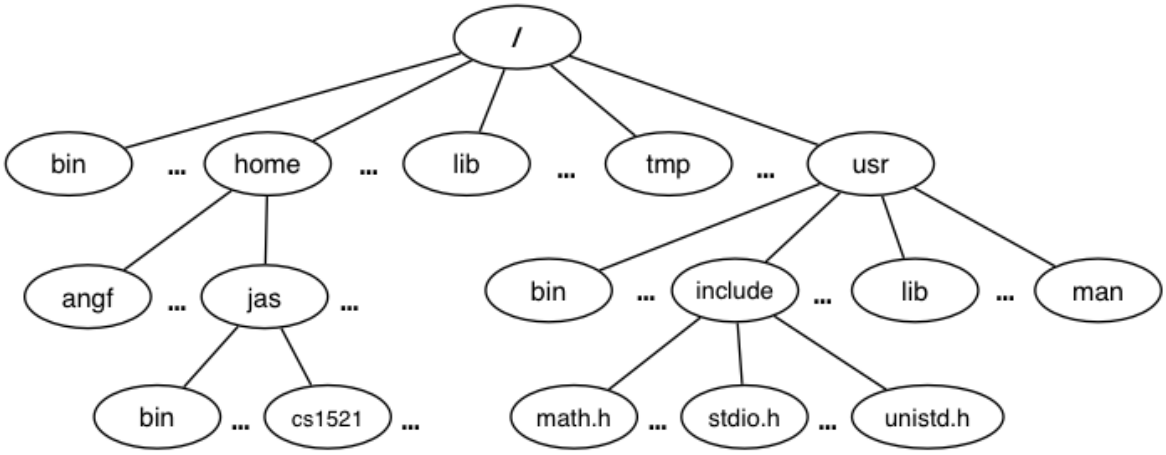
    return 0;
}
```

NOTE: WE WILL NOT BE CODING LIKE THIS!

What are Files and Directories?

- File systems manage persistent (existing forever) stored data e.g. on a magnetic disk or SSD
- On Unix-like systems:
  - a **file** is sequence (array) of zero or more bytes
  - no meaning for bytes associate with file
    - file metadata doesn't record that it is e.g. ASCII or MP4
    - they are just bytes – nothing special
  - a **directory** is an object containing zero or more files or directories
- File systems maintain metadata for files and directories, e.g. permissions and can access:
  - files
  - directories (folders)
  - storage devices (disks, SSD, etc)
  - peripherals (keyboard, mouse, USB, etc)
  - system information
  - inter-process communication
- System calls provide operations to manipulate files
- libc provides a low-level API to manipulate files
- `stdio.h` provides more portable, higher-level API to manipulate files

- File systems are like a tree, if you follow symbolic links it is a graph



Unix-like Files and Directories

- Unix-like filenames are sequences of 1 or more bytes
  - Name can't have 0x00 (ASCII '\0' – to terminate files) and 0x2F (ASCII '/' – to separate pathnames
  - Max filename length is typically 255
- Can't use `.` (current directory) or `..` (parent directory) as file names
- Some programs (shell, ls) treat filenames starting with `.` specially

- Unix philosophy is: EVERYTHING IS A FILE

Unix/Linux Pathnames

- Files and directories are accessed via pathnames, e.g:
  `/home/z5555555/lab07/main.c`
- Absolute pathnames start with a leading / and give full path from root
  `e.g. /usr/include/stdio.h, /cs1521/public_html/`
- Every process (running program) has an associated absolute pathname called the current working directory (CWD)
- Shell command `pwd` prints CWD
- Relative pathname do not start with a leading /
  `e.g. ../../another/path/prog.c, ./a.out, main.c`

File Metadata

- Metadata for file system objects is stored in inodes, which hold:
  - Location of file contents in file systems
  - File type (regular file, directory, …)
  - File size in bytes
  - File ownership
  - File access permissions – who can read, write, execute the file
  - Time stamps – time of creation/access/update
- File systems add much complexity to improve performance. E.g. may store very small files in an inode itself
- Do: `ls -l file.c` for metadata of file.c

```
$ ls -l hello
-rw-r--r-- 2 andrewt 13 Oct 23 16:18 hello $

^ (first 3 is what owner can do, next 3 is what people in the same
group can do, last 3 is what anyone can do – [r]ead, [w]rite, [a]ppend)
          ^ (bytes of file)
            ^ (owner)
                    ^ (last accessed)
```

File Inodes

- Unix-lke file systems effectively have an array of inodes
- Each inode has an inode-number (or i-number) – its index in this array
- Directories are effectively a list of (name, i-number) pairs
- i-numbers effectively identify files in a file system (like zID), we don't do much with the number itself

```
$ ls -i file.c
109988273 file.c
$
```

File Access behind-the-scenes

1. Open directory and scan for name of file
2. If not found, “No such file or directory”
3. If found as (name, i-number), access inode table inodes[inumber]
4. Check file metadata and …
    a. Check file access permissions (if don’t have required access: “Permission Denied”
    b. Collect info about file’s size and location
    c. Update timestamp
5. Use data in inode to access file contents

Hard Links and Symbolic Links

- File system links allow multiple paths to access the same file
- Hard Links:
    - Multiple names referencing the same file (inode)
    - Two entries must be on the same file system
    - All hard links to a file have equal status
    - File is destroyed when last hard link is removed
    - Can not create an extra hard link to directories
- Symbolic Links (symlinks):
    - Point to another path name
    - Accessing the symlink (by default) accesses the file being pointed to
    - Can point to a directory
    - Can point to a pathname on other filesystems

```
$ echo 'Hello Andrew' >hello
$ ln hello hola # create hard link
$ ln -s hello selamat # create symbolic link
$ ls -l hello hola selamat
-rw-r--r-- 2 andrewt 13 Oct 23 16:18 hello
-rw-r--r-- 2 andrewt 13 Oct 23 16:18 hola
lrwxrwxrwx 1 andrewt 5 Oct 23 16:20 selamat -> hello
$ cat hello
Hello Andrew
$ cat hola
Hello Andrew
$ cat selamat
Hello Andrew
```

System Calls to Manipulate Files

- Unix presents a uniform interface to file system objects
- System calls manipulate a stream of bytes
- Accessed via a file descriptor
    - which are small integers
    - index to per-process operating system table (array)
    - Definition from StackOverflow:
      "In simple words, when you open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 100 files opened in your OS then there will be 100 entries in OS (somewhere in kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. So it is just an integer number that uniquely represents an opened file in operating system. If your process opens 10 files then your Process table will have 10 entries for file descriptors."
- Important system calls:
    - `open()` – open a file system object, return a file descriptor
    - `close()` – stop using a file descriptor
    - `read()` – read some bytes into a buffer from a file descriptor
    - `write()` – write some bytes from a buffer to a file descriptor
    - `lseek()` – move to a specified offset within a file
    - `stat()` – get file system object metadata

Example: Using system call directly to create a file

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    // cp <file1> <file2> with syscalls, no error handling!
    // system call number 2 == open, takes 3 arguments:
    //   1) address of zero-terminated string containing file pathname
    //   2) bitmap indicating whether to write, read, ... file
    //      0x41 == write to file creating if necessary
    //   3) permissions if file will be newly created
    //      0644 == readable to everyone, writeable by owner

    long read_file_descriptor = syscall(2, argv[1], 0, 0);
    long write_file_descriptor = syscall(2, argv[2], 0x41, 0644);

    while (1) {
        char bytes[4096];
        long bytes_read = syscall(0, read_file_descriptor, bytes, 4096);
        if (bytes_read <= 0) {
            break;
        }
        syscall(1, write_file_descriptor, bytes, bytes_read);
    }

    return 0;
}
```

C Library Wrappers for System Calls

- There are C library functions on Unix-like systems corresponding to each system call
  - e.g. open, read, write, close
  - THE SYSCALL FUNCTION IS NOT USED IN NORMAL CODING
  - NOTE: These functions AREN'T PORTABLE – absent from many platforms/implementations
- POSIX standardises some of these functions
  - POSIX (The Portable Operating System Interface) is a family of standards set by IEEE Computer Society for maintaining compatibility between operating systems
- SO… it is better to use functions from the standard C library, available everywhere such as in `stdio.h`
- But sometimes you may need to use lower level functions

OPEN SYSTEM CALL

```
int open(char *pathname, int flags)
```

- Open file at `pathname`, according to `flags`
- `flags` is a bit-mask define in `<fcntl.h>`
  - `O_RDONLY` — open for reading
  - `O_WRONLY` — open for writing
  - `O_APPEND` — append on each write
  - `O_RDWR` — open object for reading and writing
  - `O_CREAT` — create file if doesn't exist
  - `O_TRUNC` — truncate to size 0
- Flags can be combined e.g. `(O_WRONLY|O_CREAT)`
- If successful, return file descriptor
- If fail, return `-1` and set `errno`

CLOSE SYSTEM CALL

```
int close(int fd)
```

- Release open file descriptor `fd`
- If successful return `0`
- If fail, return -1 and set `errno` (this may be due to `fd` is already closed or `fd` is not an open file descriptor)

- NOTE: If you `rm` a file, you:
  - Remove the file's entry from a directory
  - But the inode and data persists until:
    - All references to the inode from other directories are removed
    - All processes accessing the file `close()` their file descriptor
  - After this, the inode and the space used for file contents is recycled

READ SYSTEM CALL

```
ssize_t read(int fd, void *buf, size_t count)
```

- Read bytes from stream identified by `fd` to `buf`
- If successful, number of bytes read is returned
- If 0 returns, there is no more bytes to read
- If error return -1 and `errno` set to reason
- Repeated calls to read the entire file

WRITE SYSTEM CALL

```
ssize_t write(int fd, const void *buf, size_t count)
```

- Write bytes from `buf` into stream identified by `fd`
- If successful, number of bytes written is returned
- If error return -1 and `errno` set to reason
- Repeated calls to write to the entire file

LSEEK SYSTEM CALL

```
off_t lseek(int fd, off_t offset, int whence)
```

- Changes the 'current position' in the file of `fd`
- offset is in units of bytes, and can be positive or negative
- whence can be one of …
    - SEEK_SET — set file position to Offset from start of file
    - SEEK_CUR — set file position to Offset from current position
    - SEEK_END — set file position to Offset from end of file
- Seeking beyond the end of a file leaves a gap which read's as 0
- Seeking back beyond the start of file sets position to start of file

STAT SYSTEM CALL

```
int stat(const char *pathname, struct stat *statbuf)
```

- Returns metadata associated with pathname in statbuf (look at example!)
- Metadata has:
    - inode number
    - type (file, directory, symbolic link, device)
    - size of file in bytes (if it is a file)
    - permissions (read, write, execute)
    - times of last access/modification/status-change

returns -1 and sets `errno` if metadata not accessible

Extra Types for File System Operations

- These help for describing variables! They are just different names for other types you know

```
#include <sys/types.h>
#include <sys/stat.h>
```

- `off_t` – offsets within files
    - typically `int64_t` – signed to allow backward references
- `size_t` – number of bytes in some object
    - typically `uint64_t` – unsigned since objects can't have a negative size
- `ssize_t` – sizes of read/written bytes
    - like `*size_t`, but signed to allow for error values
- `struct stat` – file system object metadata
    - stores information about file, not its contents
    - requires other types: `ino_t`, `dev_t`, `time_t`, `uid_t`

Example: Hello world with libc

```c
#include <unistd.h>

int main(void) {
    char bytes[16] = "Hello, Andrew!\n";

    // write takes 3 arguments:
    //   1) file descriptor, 1 == stdout
    //   2) memory address of first byte to write
    //   3) number of bytes to write

    write(1, bytes, 15); // prints Hello, Andrew! on stdout

    return 0;
}
```

Example: Copy from stdin to stout with libc

```c
#include <unistd.h>

int main(void) {
    while (1) {
        char bytes[4096];

        // system call number 0 == read
        // read system call takes 3 arguments:
        //   1) file descriptor, 0 == stdin
        //   2) memory address to put bytes read
        //   3) maximum number of bytes read
        // returns number of bytes actually read

        ssize_t bytes_read = read(0, bytes, 4096);

        if (bytes_read <= 0) {
            break;
        }

        write(1, bytes, bytes_read); // prints bytes to stdout
    }

    return 0;
}
```

Example: Copy two files with libc

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    // open takes 3 arguments:
    //   1) address of zero-terminated string containing pathname of file to
open
    //   2) bitmap indicating whether to write, read, ... file
    //   3) permissions if file will be newly created
    //      0644 == readable to everyone, writeable by owner

    int read_file_descriptor = open(argv[1], O_RDONLY);
    int write_file_descriptor = open(argv[2], O_WRONLY | O_CREAT, 0644);

    while (1) {
        char bytes[4096];
        ssize_t bytes_read = read(read_file_descriptor, bytes, 4096);
        if (bytes_read <= 0) {
            break;
        }
        write(write_file_descriptor, bytes, bytes_read);
    }

    return 0;
}
```

Example: lseek to read the last byte then the first byte of a file with libc

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <source file>\n", argv[0]);
        return 1;
    }

    int read_file_descriptor = open(argv[1], O_RDONLY);
    char bytes[1];
    // move to a position 1 byte from end of file
    // then read 1 byte
    lseek(read_file_descriptor, -1, SEEK_END);
    read(read_file_descriptor, bytes, 1);
    printf("last byte of the file is 0x%02x\n", bytes[0]);

    // move to a position 0 bytes from start of file
    // then read 1 byte
    lseek(read_file_descriptor, 0, SEEK_SET);
    read(read_file_descriptor, bytes, 1);
    printf("first byte of the file is 0x%02x\n", bytes[0]);
```

```c
    // move to a position 41 bytes from start of file
    // then read 1 byte
    lseek(read_file_descriptor, 41, SEEK_SET);
    read(read_file_descriptor, bytes, 1);
    printf("42nd byte of the file is 0x%02x\n", bytes[0]);

    // move to a position 58 bytes from current position
    // then read 1 byte
    lseek(read_file_descriptor, 58, SEEK_CUR);
    read(read_file_descriptor, bytes, 1);
    printf("100th byte of the file is 0x%02x\n", bytes[0]);

    return 0;
}
```

stdio.h – C Standard Library I/O Functions

- stdio.h is part of standard C library
- Available in every C implementation that can do I/O
- stdio.h functions are portable, convenient and efficient
- use them by default for file operations
- on Unix-like systems they will call open/read/write/… but with buffering for efficiency (so they send multiple bytes at a time for efficiency, NOT by sending each byte one by one)

stdio.h operations

```c
FILE *fopen(const char *pathname, const char *mode)
                        ^ the new file name          ^ r and/or w and/or a
```

- `stdio.h` equivalent to open
- `mode` is a string of 1 or more characters, for what you want to do with the file, including:
  - r – open text file for reading
  - w – open text file for writing and truncated to 0 length (will make a file if it doesn't exist)
  - a – open text file for appending (will make a file if it doesn't exist)
- returns a `FILE *` pointer
- `FILE` is an opaque struct – we cannot access its fields
- If fails, leaves a reason it failed in global variable. Use `perror(const char *s)` to print it

```c
int fclose(FILE *stream)
```

- `stdio.h` equivalent to close
- Causes any buffers for stream to be emptied and the file to be closed

```c
int fflush(FILE *stream);
```

- Flush any buffered data on output stream
- Since `stdio.h` sends data with buffering, this will send it NOW!
- Does not close the file

```c
int fseek(FILE *stream, long offset, int whence);
```

- `stdio.h` equivalent to lseek
- Changes the 'current position' in the file of `stream`
- offset is in units of bytes, and can be positive or negative
- whence can be one of …
  - SEEK_SET — set file position to Offset from start of file
  - SEEK_CUR — set file position to Offset from current position
  - SEEK_END — set file position to Offset from end of file

```
int fgetc(FILE *stream)
```

- Reads a byte from the stream

```
int fputc(int c, FILE *stream)
```

- Writes a byte to the stream
- c is the byte, which is an int type (not char – char is 0 -> 255) since EOF has a value of -1

```
char *fputs(char *s, FILE *stream)
```

- Writes a string to the stream
- s is the string
- Stops printing when it reaches '\0'

```
char *fgets(char *s, int size, FILE *stream)
```

- Reads a line from the stream of a certain size

```
int fscanf(FILE *stream, const char *format, ...)
```

- Formatted input

```
int fprintf(FILE *stream, const char *format, ...)
```

- Formatted output
- Stops printing when it reaches '\0'

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Read an array of bytes (fgetc + loop is often better/more common to use)
- ptr is where the data will be after reading it
- size is the total number of bytes to be read
- nmemb is the number of times a record will be read
- stream where the records will be read from

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

- Write an array of bytes (fputc + loop is often better/more common to use)
- ptr is where the data will be from
- size is the total number of bytes to be read
- nmemb is the number of times a record will be read
- stream is where the records will be written to

```
long ftell(FILE *stream);
```

- Returns your position in the stream

```
int fstat(int fd, struct stat *statbuf)
```

- Returns metadata associated with pathname in statbuf (look at example!)
- Metadata has:
    - inode number
    - type (file, directory, symbolic link, device)
    - size of file in bytes (if it is a file)
    - permissions (read, write, execute)
    - times of last access/modification/status-change
- returns -1 and sets `errno` if metadata not accessible

Definition of struct stat

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
};
```

st_mode field of struct stat

- Bitwise of these values (written in octal here)

```
S_IFLNK 0120000 symbolic link
S_IFREG 0100000 regular file
S_IFBLK 0060000 block device
S_IFDIR 0040000 directory
S_IFCHR 0020000 character device
S_IFIFO 0010000 FIFO
S_IRUSR 0000400 owner has read permission
S_IWUSR 0000200 owner has write permission
S_IXUSR 0000100 owner has execute permission
S_IRGRP 0000040 group has read permission
S_IWGRP 0000020 group has write permission
S_IXGRP 0000010 group has execute permission
S_IROTH 0000004 others have read permission
S_IWOTH 0000002 others have write permission
S_IXOTH 0000001 others have execute permission
```

```
int mkdir(const char *pathname, mode_t mode)
```

- Create a new directory called pathname with permissions on mode
- If pathname is a/b/c/d
    - All directories a, b and c must exist
    - Directory c must be writtable to the caller
    - Directory d must not already exist
- If fail, returns -1 and sets `errno`
- If success returns 0

File Permissions

- Three types:
    - Read – to get bytes of file
    - Write – to change the bytes of a file
    - Execute – to execute the file
- Often represented as bits of an octal digit
- For three groups:
    - Owner – the file owner
    - Group – users in the group of the file
    - Other – for everyone

Other useful Linux (POSIX) functions

```
chmod(char *pathname, mode_t mode)    // change permission of file/...
unlink(char *pathname)                // remove a file/directory/...
rename(char *oldpath, char *newpath)  // rename a file/directory
chdir(char *path)                     // change current working directory
getcwd(char *buf, size_t size)        // get current working directory
link(char *oldpath, char *newpath)    // create hard link to a file
symlink(char *target, char *linkpath) // create a symbolic link
```

Example: Using fputc to output bytes

```c
char bytes[] = "Hello, stdio!\n"; // 15 bytes
// write 14 bytes so we don't write (terminating) 0 byte
for (int i = 0; i < (sizeof bytes) - 1; i++) {
    fputc(bytes[i], stdout);
}
// or as we know bytes is 0-terminated
for (int i = 0; bytes[i] != '\0'; i++) {
    fputc(bytes[i], stdout);
}
// or if you prefer pointers (probably not the best way as not very readable)
for (char *p = &bytes[0]; *p != '\0'; p++) {
    fputc(*p, stdout);
}
```

Example: Using fputs, fwrite and fprintf to output bytes

```c
char bytes[] = "Hello, stdio!\n"; // 15 bytes

// fputs relies on bytes being 0-terminated
fputs(bytes, stdout);
// write 14 1 byte items
fwrite(bytes, 1, (sizeof bytes) - 1, stdout);
// %s relies on bytes being 0-terminated
fprintf(stdout, "%s", bytes);
```

Example: Using fgetc and fputc to copy stdin to stdout

```c
#include <stdio.h>

int main(void) {
    // c can not be char (common bug)
    // fgetc returns 0..255 and EOF (usually -1)
    int c;

    // return  bytes from the stream (stdin) one at a time
    while ((c = fgetc(stdin)) != EOF) {
        fputc(c, stdout); // write the byte to standard output
    }

    return 0;
}
```

Example: Using fgets and fputs to copy stdin to stdout

```c
#include <stdio.h>

int main(void) {
    // return bytes from the stream (stdin) line at a time
    // BUFSIZ is defined in stdio.h - its an efficient value to use
    // but any value would work


    char line[BUFSIZ];
    while (fgets(line, sizeof line, stdin) != NULL) {
        fputs(line, stdout);
    }
    //
    // NOTE: fgets returns a null-terminated string
    //       in other words a 0 byte marks the end of the bytes read
    //                        ^ (ASCII: NULL)
    //
    // fgets can not be used to read bytes which are 0
    // fputs takes a null-terminated string
    // so fputs can not be used to write bytes which are 0
    // hence you can't use fgets/fputs for binary data e.g. jpgs
    // (so it's probs better most of the time to use fgetc/fputc + loop)

    return 0;
}
```

Example: Using fread and fwrite to copy stdin to stdout

```c
#include <stdio.h>

int main(void) {
    while (1) {
        char bytes[4096];

        ssize_t bytes_read = fread(bytes, sizeof bytes, 1, stdin);

        if (bytes_read <= 0) {
            break;
        }

        fwrite(bytes, bytes_read, 1, stdout);
    }

    return 0;
}
```

Example: Creating a file

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    FILE *output_stream = fopen("hello.txt", "w");
    if (output_stream == NULL) {
        perror("hello.txt");
        return 1;
    }

    fputs("Hello, Andrew!\n", output_stream);

    // fclose will flush data to file
    // best to close file ASAP
    // but doesn't matter as file automatically closed on exit
    fclose(output_stream);

    return 0;
}
```

Example: Using fgetc to copy a file

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);
        return 1;
    }

    FILE *input_stream = fopen(argv[1], "r");
    if (input_stream == NULL) {
        perror(argv[1]);  // prints why the open failed
        return 1;
    }

    FILE *output_stream = fopen(argv[2], "w");
    if (output_stream == NULL) {
        perror(argv[2]);
        return 1;
    }

    int c; // not char!
    while ((c = fgetc(input_stream)) != EOF) {
        fputc(c, output_stream);
    }


    fclose(input_stream);  // optional as close occurs
    fclose(output_stream); // automatically on exit

    return 0;
}
```

Example: Using fwrite to copy a file

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source file> <destination file>\n", argv[0]);
        return 1;
    }

    FILE *input_stream = fopen(argv[1], "rb");
    if (input_stream == NULL) {
        perror(argv[1]);  // prints why the open failed
        return 1;
    }

    FILE *output_stream = fopen(argv[2], "wb");
    if (output_stream == NULL) {
        perror(argv[2]);
        return 1;
    }

    // this will be slightly faster than an a fgetc/fputc loop
    while (1) {
        char bytes[BUFSIZ];
        size_t bytes_read = fread(bytes, sizeof bytes, 1, input_stream);
        if (bytes_read <= 0) {
            break;
        }
        fwrite(bytes, bytes_read, 1, output_stream);
    }

    fclose(input_stream);  // optional as close occurs
    fclose(output_stream); // automatically on exit

    return 0;
}
```

Example: Using fseek to read the last byte then the first byte of a file

Example: Using fseek to read bytes in the middle of a file

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <source file>\n", argv[0]);
        return 1;
    }

    FILE *input_stream = fopen(argv[1], "rb");

    // move to a position 1 byte from end of file
    // then read 1 byte
    fseek(input_stream, -1, SEEK_END);
    printf("last byte of the file is 0x%02x\n", fgetc(input_stream));

    // move to a position 0 bytes from start of file
    // then read 1 byte
    fseek(input_stream, 0, SEEK_SET);
    printf("first byte of the file is 0x%02x\n", fgetc(input_stream));

    // move to a position 41 bytes from start of file
    // then read 1 byte
    fseek(input_stream, 41, SEEK_SET);
    printf("42nd byte of the file is 0x%02x\n", fgetc(input_stream));

    // move to a position 58 bytes from current position
    // then read 1 byte
    fseek(input_stream, 58, SEEK_CUR);
    printf("100th byte of the file is 0x%02x\n", fgetc(input_stream));

    return 0;
}
```

Example: Using fseek to read change a random file it – this is called fuzzing

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <source file>\n", argv[0]);
        return 1;
    }


    FILE *f = fopen(argv[1], "r+");    // open for reading and writing

    fseek(f, 0, SEEK_END);             // move to end of file

    long n_bytes = ftell(f);           // get number of bytes in file

    srandom(time(NULL));               // initialize random number
                                       // generator with current time

    long target_byte = random() % n_bytes; // pick a random byte

    fseek(f, target_byte, SEEK_SET);  // move to byte

    int byte = fgetc(f);               // read byte


    int bit = random() % 8;            // pick a random bit

    int new_byte = byte ^ (1 << bit); // flip the bit

    fseek(f, -1, SEEK_CUR);            // move back to same position


    fputc(new_byte, f);                // write the byte

    fclose(f);

    printf("Changed byte %ld of %s from %02x to %02x\n",target_byte, argv[1],
byte, new_byte);
    return 0;
}
```

Example: Create a gigantic sparse file – 16TB (advanced topic)

```c
#include <stdio.h>
int main(void) {
    FILE *f = fopen("sparse_file.txt", "w");
    fprintf(f, "Hello, Andrew!\n");
    fseek(f, 16L * 1000 * 1000 * 1000 * 1000, SEEK_CUR);
    fprintf(f, "Goodbye, Andrew!\n");
    fclose(f);
    return 0;
}
```

Almost all the 16 TB are zeros, which the file system doesn't actually store – search sparse file!

Convenient Functions for stdin/stdout

```
int getchar()                   // fgetc(stdin)
int putchar(int c)              // fputc(c, stdin)

int puts(char *s)               // fputs(s,stdout)

int scanf(char *format, ...)    // fscanf(stdin, format, ...)
int printf(char *format, ...)   // fprintf(stdout, format, ...)

char *gets(char *s);             // NEVER USE
```

stdio.h – I/O to strings

- `int snprintf(char *str, size_t size, const char *format, ...);`
  - Like printf, but output goes to char array str (so you don't need to make a loop + getchar)
  - Handy for creating strings passed to other functions
- `int sscanf(const char *str, const char *format, ...);`
  - Like scanf, but input comes from char array str
- `int sprintf(char *str, const char *format, ...); // DO NOT USE`
  - DO NOT USE THIS! As it can overflow str

Example: stat.c

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

void stat_file(char *pathname);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        stat_file(argv[arg]);
    }
    return 0;
}

void stat_file(char *pathname) {
    printf("stat(\"%s\", &s)\n", pathname);

    struct stat s;
    if (stat(pathname, &s) != 0) {
        perror(pathname);
        exit(1);
    }

    printf("ino =  %10ld # Inode number\n", s.st_ino);
    printf("mode = %10o # File mode \n", s.st_mode);
    printf("nlink =%10ld # Link count \n", (long)s.st_nlink);
    printf("uid =  %10u # Owner uid\n", s.st_uid);
    printf("gid =  %10u # Group gid\n", s.st_gid);
    printf("size = %10ld # File size (bytes)\n", (long)s.st_size);

    printf("mtime =%10ld # Modification time (seconds since 1/1/70)\n",
           (long)s.st_mtime);
}
```

Example: Create a directory

```
$ dcc mkdir.c
$ ./a.out new_dir
$ ls -ld new_dir
drwxr-xr-x 2 z5555555 z5555555 60 Oct 29 16:28 new_dir
$
```

```c
#include <stdio.h>
#include <sys/stat.h>

// create the directories specified as command-line arguments
int main(int argc, char *argv[]) {

    for (int arg = 1; arg < argc; arg++) {
        if (mkdir(argv[arg], 0755) != 0) { // 0755 is the permissions
            perror(argv[arg]);  // prints why the mkdir failed
            return 1;
        }
    }

    return 0;
}
```

Example: Changing File Permissions

```
$ dcc chmod.c
$ ls -l chmod.c
-rw-r--r-- 1 z5555555 z5555555 746 Nov  4 08:20 chmod.c
$ ./a.out 600 chmod.c
$ ls -l chmod.c
-rw------- 1 z5555555 z5555555 787 Nov  4 08:22 chmod.c
$ ./a.out 755 chmod.c
chmod.c 755
$ ls -l chmod.c
-rwxr-xr-x 1 z5555555 z5555555 787 Nov  4 08:22 chmod.c
$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

// change permissions of the specified files
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <mode> <files>\n", argv[0]);
        return 1;
    }

    char *end;
    // first argument is mode in octal
    mode_t mode = strtol(argv[1], &end, 8);

    // check first argument was a valid octal number
    if (argv[1][0] == '\0' || end[0] != '\0') {
        fprintf(stderr, "%s: invalid mode: %s\n", argv[0], argv[1]);
        return 1;
    }

    for (int arg = 2; arg < argc; arg++) {
        if (chmod(argv[arg], mode) != 0) {
            perror(argv[arg]);  // prints why the chmod failed
            return 1;
        }
    }

    return 0;
}
```

Example: Removing Files

```
$ dcc rm.c
$ ./a.out rm.c
$ ls -l rm.c
ls: cannot access 'rm.c': No such file or directory
$
```

```c
#include <stdio.h>
#include <unistd.h>

// remove the specified files
int main(int argc, char *argv[]) {

    for (int arg = 1; arg < argc; arg++) {
        if (unlink(argv[arg]) != 0) {
                // if unlink returns 0 it means it failed
            perror(argv[arg]);  // prints why the unlink failed
            return 1;
        }
    }

    return 0;
}
```

Example: Renaming a File

```
$ dcc rename.c
$ ./a.out rename.c renamed.c
$ ls -l  renamed.c
renamed.c
$
```

```c
#include <stdio.h>

// rename the specified file
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <old-filename> <new-filename>\n",
                argv[0]);
        return 1;
    }
    char *old_filename = argv[1];
    char *new_filename = argv[2];
    if (rename(old_filename, new_filename) != 0) {
        // returning 0 means success, not returning 0 is error
        fprintf(stderr, "%s rename %s %s:", argv[0], old_filename,
                new_filename);
        perror("");
        return 1;
    }

    return 0;
}
```

Example: cd-ing up one directory at a time

```c
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <string.h>

int main(void) {
    // use repeated chdir("..") to climb to root of the file system
    char pathname[PATH_MAX];
    while (1) {
        if (getcwd(pathname, sizeof pathname) == NULL) {
                // returning NULL means error (USE MAN SO U DON'T FORGET!)
            perror("getcwd");
            return 1;
        }
        printf("getcwd() returned %s\n", pathname);

        if (strcmp(pathname, "/") == 0) {
            return 0;
        }

        if (chdir("..") != 0) {
            perror("chdir");
            return 1;
        }
    }
    return 0;
}
```

Example: Making a 1000-deep directory

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <limits.h>


int main(int argc, char *argv[]) {

    for (int i = 0; i < 1000; i++) {
        char dirname[256];
        snprintf(dirname, sizeof dirname, "d%d", i);

        if (mkdir(dirname, 0755) != 0) {
            perror(dirname);
            return 1;
        }
        if (chdir(dirname) != 0) {
            perror(dirname);
            return 1;
        }

        char pathname[1000000];
        if (getcwd(pathname, sizeof pathname) == NULL) {
            perror("getcwd");
            return 1;
        }
        printf("\nCurrent directory now: %s\n", pathname);
    }

    return 0;
}
```

Example: Creating 100 hard links to a file

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <limits.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char pathname[256] = "hello.txt";

    // create a target file
    FILE *f1;
    if ((f1 = fopen(pathname, "w")) == NULL) {
        perror(pathname);
        return 1;
    }
    fprintf(f1, "Hello Andrew!\n");
    fclose(f1);

    for (int i = 0; i < 1000; i++) {


        printf("Verifying '%s' contains: ", pathname);


        // print contents of the file
        FILE *f2;
        if ((f2 = fopen(pathname, "r")) == NULL) {
            perror(pathname);
            return 1;
        }
        int c;
        while ((c = fgetc(f2)) != EOF) {
            fputc(c, stdout);
        }
        fclose(f2);


        // create link
        char new_pathname[256];
        snprintf(new_pathname, sizeof new_pathname,
                 "hello_%d.txt", i);

        printf("Creating a link %s -> %s\n",
               new_pathname, pathname);
        if (link(pathname, new_pathname) != 0) {
            perror(pathname);
            return 1;
        }
    }

    return 0;
}
```

Example: Create symbolic links to a file

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <limits.h>
#include <string.h>


int main(int argc, char *argv[]) {
    char pathname[256] = "hello.txt";

    // create target file
    FILE *f1;
    if ((f1 = fopen(pathname, "w")) == NULL) {
        perror(pathname);
        return 1;
    }
    fprintf(f1, "Hello Andrew!\n");
    fclose(f1);

    for (int i = 0; i < 1000;i++) {
        printf("Verifying '%s' contains: ", pathname);
        FILE *f2;
        if ((f2 = fopen(pathname, "r")) == NULL) {
            perror(pathname);
            return 1;
        }
        int c;
        while ((c = fgetc(f2)) != EOF) {
            fputc(c, stdout);
        }
        fclose(f2);

        char new_pathname[256];
        snprintf(new_pathname, sizeof new_pathname, "hello_%d.txt", i);

        printf("Creating a symbolic link %s -> %s\n", new_pathname, pathname);
        if (symlink(pathname, new_pathname) != 0) {
            perror(pathname);
            return 1;
        }

        strcpy(pathname, new_pathname);

    }

    return 0;
}
```

Example: List directory

```c
#include <stdio.h>
#include <dirent.h>

// list the contents of directories specified as command-line arguments
int main(int argc, char *argv[]) {

    for (int arg = 1; arg < argc; arg++) {
        DIR *dirp = opendir(argv[arg]);
        if (dirp == NULL) {
            perror(argv[arg]);  // prints why the open failed
            return 1;
        }

        struct dirent *de;

        while ((de = readdir(dirp)) != NULL) {
            printf("%ld %s\n", de->d_ino, de->d_name);
        }

        closedir(dirp);
    }
    return 0;
}
```

Example: Writing an array as binary data (using fwrite)

```
$ dcc write_array.c -o write_array
$ dcc read_array.c -o read_array
$ ./write_array
$ ls -l array.save
-rw-r--r-- 1 z5555555 z5555555 40 Oct 30 21:46 array.save
$ ./read_array
10 11 12 13 14 15 16 17 18 19
$
```

```c
#include <stdio.h>

int main(void) {
    int array[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };

    FILE *f = fopen("array.save", "w");
    if (f == NULL) {
        perror("array.save");
        return 1;
    }
    // assuming int are 4 bytes, this will
    // write 40 bytes of array to "array.save"
    if (fwrite(array, 1, sizeof array, f) != sizeof array) {
        perror("array.save");
        return 1;
    }
    fclose(f);
    return 0;
}
```

Example: Writing an array as binary data (using fread)

```c
#include <stdio.h>

int main(void) {
    int array[10];

    FILE *f = fopen("array.save", "r");
    if (f == NULL) {
        perror("array.save");
        return 1;
    }
    // read array: NOT-PORTABLE: depends on size of int and byte-order
    if (fread(array, 1, sizeof array, f) != sizeof array) {
        perror("array.save");
        return 1;
    }
    fclose(f);

    for (int i = 0; i < 10; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Example: Writing a pointer as binary data (using fwrite)

```
$ dcc write_pointer.c -o write_pointer
$ dcc read_pointer.c -o read_pointer
$ ./write_pointer
p         = 0x410234
&array[5] = 0x410234
array[5]  = 15
*p        = 15
$ ls -l array_pointer.save
-rw-r--r-- 1 z5555555 z5555555 48 Oct 30 21:46 array.save
$ ./read_pointer
10 11 12 13 14 15 16 17 18 19
p         = 0x410234
&array[5] = 0x4163f4
array[5]  = 15
*p        = -1203175425
$
```

```c
#include <stdio.h>

int main(void) {
    int array[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
    int *p = &array[5];

    FILE *f = fopen("array.save", "w");
    if (f == NULL) {
        perror("array.save");
        return 1;
    }

    if (fwrite(array, 1, sizeof array, f) != sizeof array) {
        perror("array.save");
        return 1;
    }

    if (fwrite(&p, 1, sizeof p, f) != sizeof p) {
        perror("array.save");
        return 1;
    }

    fclose(f);

    printf("p         = %p\n", p);
    printf("&array[5] = %p\n", &array[5]);
    printf("array[5]  = %d\n", array[5]);
    printf("*p        = %d\n", *p);
    return 0;
}
```

Example: Reading a pointer as binary data (using fread)

```c
#include <stdio.h>

int main(void) {
    int array[10];
    int *p;

    FILE *f = fopen("array.save", "r");
    if (f == NULL) {
        perror("array.save");
        return 1;
    }

    if (fread(array, 1, sizeof array, f) != sizeof array) {
        perror("array.save");
        return 1;
    }
    // BROKEN - address of array has almost certainly  changed
    // BROKEN - so address p needs to point has changed
    if (fread(&p, 1, sizeof p, f) != sizeof p) {
        perror("array.save");
        return 1;
    }

    fclose(f);
```

```
    // print array
    for (int i = 0; i < 10; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    printf("p         = %p\n", p);
    printf("&array[5] = %p\n", &array[5]);
    printf("array[5]  = %d\n", array[5]);
    printf("*p        = %d\n", *p);

    return 0;
}
```

Example: Reimplementing stdio.h – my_fopen, my_fgetc, my_fputc, my_fclose (UNBUFFERED)

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

#define MY_EOF -1

// struct to hold data for a stream
typedef struct my_file {
    int fd;
} my_file_t;

my_file_t *my_fopen(char *file, char *mode) {
    int fd = -1;
    if (mode[0] == 'r') {
        fd = open(file, O_RDONLY);
    } else if (mode[0] == 'w') {
        fd = open(file, O_WRONLY | O_CREAT, 0666);
    } else if (mode[0] == 'a') {
        fd = open(file, O_WRONLY | O_APPEND);
    }

    if (fd == -1) {
        return NULL;
    }

    my_file_t *f = malloc(sizeof *f);
    f->fd = fd;
    return f;
}
```

```c
int my_fgetc(my_file_t *f) {
    uint8_t byte;
    int bytes_read = read(f->fd, &byte, 1);
    if (bytes_read == 1) {
        return byte;
    } else {
        return MY_EOF;
    }
}

int my_fputc(int c, my_file_t *f) {
    uint8_t byte = c;
    if (write(f->fd, &byte, 1) == 1) {
        return byte;
    } else {
        return MY_EOF;
    }
}

int my_fclose(my_file_t *f) {
    int result = close(f->fd);
    free(f);
    return result;
}

int main(int argc, char *argv[]) {
    my_file_t *input_stream = my_fopen(argv[1], "r");
    if (input_stream == NULL) {
        perror(argv[1]);
        return 1;
    }

    my_file_t *output_stream = my_fopen(argv[2], "w");
    if (output_stream == NULL) {
        perror(argv[2]);
        return 1;
    }

    int c;
    while ((c = my_fgetc(input_stream)) != MY_EOF) {
        my_fputc(c, output_stream);
    }

    return 0;
}
```

Example: Reimplementing stdio.h – my_fopen, my_fgetc, my_fputc, my_fclose (BUFFERED - input)

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

// how equivalents for  EOF & BUFSIZ from stdio.h
#define MY_EOF -1
#define MY_BUFSIZ 512

// struct to hold data for a stream
typedef struct my_file {
    int     fd;
    int     n_buffered_bytes;
    int     next_byte;
    uint8_t buffer[MY_BUFSIZ];
} my_file_t;

my_file_t *my_fopen(char *file, char *mode) {
    int fd = -1;
    if (mode[0] == 'r') {
        fd = open(file, O_RDONLY);
    } else if (mode[0] == 'w') {
        fd = open(file, O_WRONLY | O_CREAT, 0666);
    } else if (mode[0] == 'a') {
        fd = open(file, O_WRONLY | O_APPEND);
    }

    if (fd == -1) {
        return NULL;
    }

    my_file_t *f = malloc(sizeof *f);
    f->fd = fd;
    f->next_byte = 0;
    f->n_buffered_bytes = 0;
    return f;
}

int my_fgetc(my_file_t *f) {
    if (f->next_byte == f->n_buffered_bytes) {
        // buffer is empty so fill it with a read
        int bytes_read = read(f->fd, f->buffer, sizeof f->buffer);
        if (bytes_read <= 0) {
            return MY_EOF;
        }
        f->n_buffered_bytes = bytes_read;
        f->next_byte = 0;
    }

    // return 1 byte from the buffer
    int byte = f->buffer[f->next_byte];
```

```
    f->next_byte++;
    return byte;
}

int my_fputc(int c, my_file_t *f) {
    uint8_t byte = c;
    if (write(f->fd, &byte, 1) == 1) {
        return byte;
    } else {
        return MY_EOF;
    }
}

int my_fclose(my_file_t *f) {
    int result = close(f->fd);
    free(f);
    return result;
}

int main(int argc, char *argv[]) {
    my_file_t *input_stream = my_fopen(argv[1], "r");
    if (input_stream == NULL) {
        perror(argv[1]);
        return 1;
    }

    my_file_t *output_stream = my_fopen(argv[2], "w");
    if (output_stream == NULL) {
        perror(argv[2]);
        return 1;
    }

    int c;
    while ((c = my_fgetc(input_stream)) != MY_EOF) {
        my_fputc(c, output_stream);
    }

    my_fclose(input_stream);
    my_fclose(output_stream);

    return 0;
}
```

Example: Reimplementing stdio.h – my_fopen, my_fgetc, my_fputc, my_fclose (BUFFERED - output)

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

// how equivalents for  EOF & BUFSIZ from stdio.h
#define MY_EOF -1
#define MY_BUFSIZ 512

// struct to hold data for a stream
typedef struct my_file {
    int     fd;
    int     is_output_stream;
    int     n_buffered_bytes;
    int     next_byte;
    uint8_t buffer[MY_BUFSIZ];
} my_file_t;

my_file_t *my_fopen(char *file, char *mode) {
    int fd = -1;
    if (mode[0] == 'r') {
        fd = open(file, O_RDONLY);
    } else if (mode[0] == 'w') {
        fd = open(file, O_WRONLY | O_CREAT, 0666);
    } else if (mode[0] == 'a') {
        fd = open(file, O_WRONLY | O_APPEND);
    }

    if (fd == -1) {
        return NULL;
    }

    my_file_t *f = malloc(sizeof *f);
    f->fd = fd;
    f->is_output_stream = mode[0] != 'r';
    f->next_byte = 0;
    f->n_buffered_bytes = 0;
    return f;
}

int my_fgetc(my_file_t *f) {
    if (f->next_byte == f->n_buffered_bytes) {
        // buffer is empty so fill it with a read
        int bytes_read = read(f->fd, f->buffer, sizeof f->buffer);
        if (bytes_read <= 0) {
            return MY_EOF;
        }
        f->n_buffered_bytes = bytes_read;
        f->next_byte = 0;
    }
```

```
    // return 1 byte from the buffer
    int byte = f->buffer[f->next_byte];
    f->next_byte++;
    return byte;
}

int my_fputc(int c, my_file_t *f) {
    if (f->n_buffered_bytes == sizeof f->buffer) {
        // buffer is full so empty it with a write
        write(f->fd, f->buffer, sizeof f->buffer); // no error checking
        f->n_buffered_bytes = 0;
    }

    // add byte byte to buffer to be written later
    f->buffer[f->n_buffered_bytes] = c;
    f->n_buffered_bytes++;
    return 1;
}

int my_fclose(my_file_t *f) {
    // don't keave unwritten bytes
    if (f->is_output_stream && f->n_buffered_bytes > 0) {
        write(f->fd, f->buffer, f->n_buffered_bytes); // no error checking
    }

    int result = close(f->fd);
    free(f);
    return result;
}

int main(int argc, char *argv[]) {
    my_file_t *input_stream = my_fopen(argv[1], "r");
    if (input_stream == NULL) {
        perror(argv[1]);
        return 1;
    }

    my_file_t *output_stream = my_fopen(argv[2], "w");
    if (output_stream == NULL) {
        perror(argv[2]);
        return 1;
    }

    int c;
    while ((c = my_fgetc(input_stream)) != MY_EOF) {
        my_fputc(c, output_stream);
    }

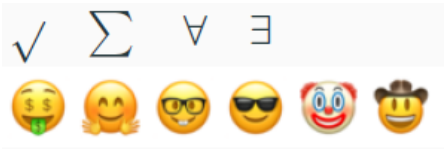
    my_fclose(input_stream);
    my_fclose(output_stream);

    return 0;
}
```

Unicode

Character Data

- Huge number of character representations (encodings) exist, these are the two you need to know:
    - ASCII (ISO 646)
        - single byte values, ony low 7-bit used, top bit always 0
        - can encode roman alphabet a-z, A-Z, digits 0-9, punctutation, control chars
        - complete alphabet for English, Bahasa
        - no diacritics, e.g. ç, so missing a little of alphabet for other latin languages, e.g.: German, French, Spanish, Swedish, Tagalog, Swahili
        - characters for most of world's languages completely missing
    - UTF-8 (Unicode)
        - contains all ASCII (single-byte) values
        - also has 2-4 byte values, top bit always 1 for bytes of multi-byte values
        - contains symbols for essentially all human languages plus other symbols, e.g.:



ASCII Character Encoding

- Uses values in the range 0x00 to 0x7F (0..127)
- Characters partitioned into sequential groups:
    - control characters (0..31) e.g. `'\n'`
    - punctuation chars (32..47,91..96,123..126)
    - digits (48..57) i.e. `'0' .. '9'`
    - uppercase alphabetic (65..95) i.e. `'A' .. 'Z'`
    - lowercase alphabetic (97..122) i.e. `'a' .. 'z'`
- Sequential nature of groups allow ordination e.g.
  `'3' - '0' == 3 and 'J' - 'A' == 10`
- See `man 7 ascii`

Unicode

- Is a widely-used standard for expressing "writing systems"
- Basically, a 32-but representation of a wide range of symbols
    - Around 140K for 140 different languages
- But using 32-bits for every symbol is expensive, so…
    - standard roman alphabet + punctuation only needs 7 bits
    - several Unicode encodings have been developed, but UTF-8 is most common and dominates web-use

UTF-8 Encoding

- Format for 1-byte, 2-byte, 3-byte and 4-byte codes:

#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

- The 127 1-byte codes are compatible with ASCII
- The 2048 2-byte codes include most Latin-script alphabet
- 65536 3-byte codes include most Asian languages
- The 2097152 4-byte codes include symbols and emojis
- Examples:

ch	code-point	unicode binary	UTF-8 encoding
$	U+0024	**0100100**	0**0100100**
¢	U+00A2	**00010100010**	11**000010** 10**100010**
€	U+20AC	**0010000010101100**	1110**0010** 10**000010** 10**101100**

Example: Printing UTF-8 in C

```c
#include <stdio.h>

int main(void) {
    printf("The unicode code point U+1F600 encodes in UTF-8\n");
    printf("as 4 bytes: 0xF0 0x9F 0x98 0x80\n");
    printf("We can output the 4 bytes like this: \xF0\x9F\x98\x80\n");
    printf("Or like this: ");
    putchar(0xF0);
    putchar(0x9F);
    putchar(0x98);
    putchar(0x80);
    putchar('\n');
}
```

Example: Converting Unicode Codepoints to UTF-8

```c
#include <stdio.h>
#include <stdint.h>

void print_utf8_encoding(uint32_t code_point) {
    uint8_t encoding[5] = {0};

    // this basically splits the code_point to a byte each, depending if its
    // 1-byte, 2-byte, etc
    if (code_point < 0x80) { // includes ASCI
        encoding[0] = code_point;
    } else if (code_point < 0x800) { // latin script
        encoding[0] = 0xC0 | (code_point >> 6);
        encoding[1] = 0x80 | (code_point & 0x3f);
    } else if (code_point < 0x10000) { // Asian languages
        encoding[0] = 0xE0 | (code_point >> 12);
        encoding[1] = 0x80 | ((code_point >> 6) & 0x3f);
        encoding[2] = 0x80 | (code_point  & 0x3f);
    } else if (code_point < 0x200000) { // other symbols, emojis
        encoding[0] = 0xF0 | (code_point >> 18);
        encoding[1] = 0x80 | ((code_point >> 12) & 0x3f);
        encoding[2] = 0x80 | ((code_point >> 6)  & 0x3f);
        encoding[3] = 0x80 | (code_point  & 0x3f);
    }

    printf("U+%x  UTF-8: ", code_point);
    for (uint8_t *s = encoding; *s != 0; s++) {
        printf("0x%02x ", *s);
    }
    printf(" %s\n", encoding);
}

int main(void) {
    print_utf8_encoding(0x42);
    print_utf8_encoding(0x00A2);
    print_utf8_encoding(0x10be);
    print_utf8_encoding(0x1F600);
}
```

Summary of UTF-8 Properties

- Compact! But it is NOT minimal though; encoding allows you to resync immediately if bytes lost from a stream
- ASCII us a subset of UTF-8 – complete backwards compatibility
- 0x2F (ASCII /) and 0x00 can not appear in multi-byte characters
- All other UTF-8 bytes > 127 (0x7F)
- C programs will treat UTF-8 similarly to ASCII
- BEWARE: Number of bytes in UTF-8 string != number of characters like in ASCII

Processes

What is a Process?

- 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 (virtual) memory
    - information about open files, sockets, etc.
        - A socket is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. Unlike pipes sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network.
- On Unix/Linux:
    - Each process has a unique process ID (pid), positive integer, type: `pid_t` defined in `unistd.h`
    - Process 0 is effectively part of the operating system
    - Process 1 (`init`) is used to boot the system – the first process! (initialisation)
    - Some parts of operating system may run as processes
    - Low-numbered processes are typically system-related processes started at boot-time

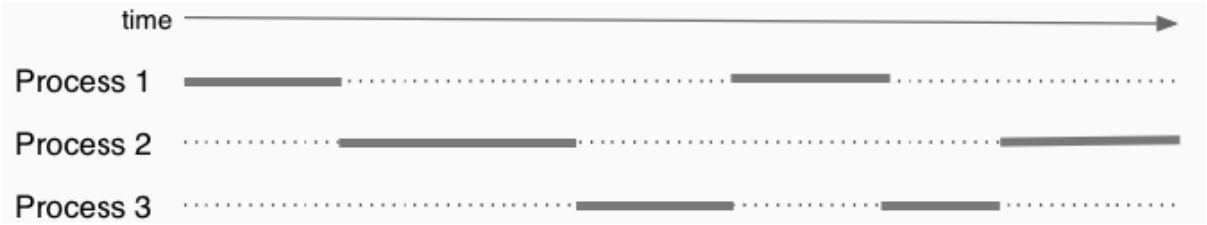
Process Parents

- Each process has a parent process
- It is the one that originally created it
- If a process terminates, then the parent becomes process 1
- Unix commands for manipulating processes:
    - `sh`: for creating processes via object-file name
    - `ps`: show process information
    - `w`: show per-user process information
    - `top`: show high-cpu-usage process information
    - `kill`: send a signal to process

- A process does not terminate until the parent knows and handles the situation!
- If exit() is called, the OS sends SIGCHILD signal to parent
- exit() won't return, or occur (terminate) until the parent handles the signal
- So they are currently zombies
- Once the parent handles it, exit() returns
- Sometimes bugs in the parent could ignore SIGCHILD, creating a long-term zombie process – wasting system resources
- Orphan process is when a parent has finished/terminated, but the processs (orphan process) keeps running. Hence, the orphan is assigned PID = 1 (init) as the parent. Init will handle SIGCHILD when process exists!

Multi-Tasking

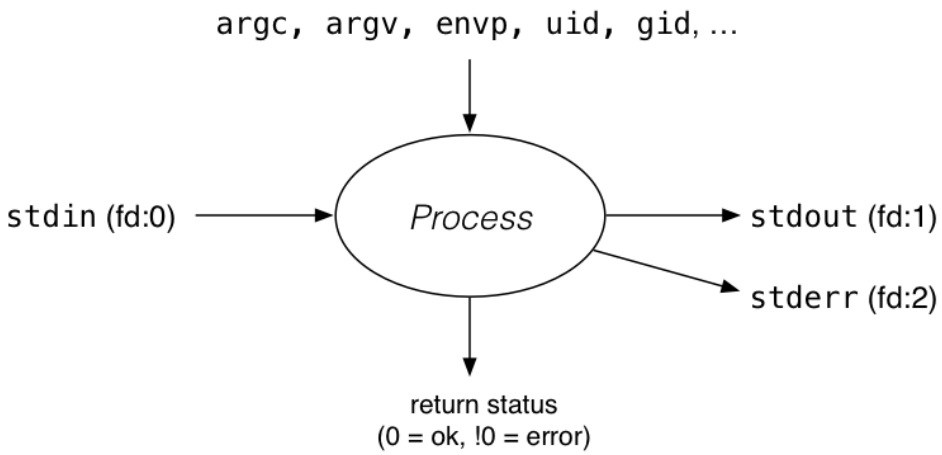
- On a typical modern operating system:
  - multiple processes are active “simultaneously” (multi-tasking)
  - the OS provides a virtual machine to each process
    - each process executes as if it’s the only process on the machine
    - it even has its own address space
- When there are multiple processes on the machine
  - each process uses the CPU until pre-empted or exits
    - In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such changes of the executed task are known as context switches.
  - then another process uses the CPU until it too is pre-empted
  - eventually the first process will get another run on the CPU



  - the overall impression is that everything runs simultaneously
- What can cause a process to be pre-empted?
  - it runs “long enough” and the OS replaces it by a waiting process
  - it needs to wait for input, or output
- On pre-emption, a context switch occurs and ITS EXPENSIVE!
  - The process’s entire state has to be saved
  - The new process’s stat must be restored
- So there has to be a balance:
  - Fairly sharing the CPU(s) among competing processes
  - Minimise response delays for the user
  - Meet other real time requirements (e.g. self-driving car)
  - Minimise number of expensive context switches

Unix/Linux Processes

- The environment for processes running on Unix/Linux systems



posix-spawn() – Run a new process

```
#include <spawn.h>

int 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[]);
```

- Creating new process, running program at `path`
- `argv` specifies the argv of new program
- `envp` specifies environment of new program
- `*pid` set to process id of new program
- `file_actions` specifies file actions to be performed before running program
- `attrp` specifies attributes for new process

Example: Using posix_spawn() to run /bin/date

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int main(void) {

    pid_t pid;
    extern char **environ;
    char *date_argv[] = {"/bin/date", "--utc", NULL};

    // spawn "/bin/date" as a separate process
    if (posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ) != 0) {
        perror("spawn");
        exit(1);
    }

    // wait for spawned processes to finish
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }

    printf("/bin/date exit status was %d\n", exit_status);
    return 0;
}
```

fork() – clone yourself

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

- Creates a new process by duplicating the calling process
- New process is the child, calling process is the parent
- The child has a different PID to parent
- In the child, fork() returns 0
- In the parent, fork() returns the PID of the child
- If the syscall call fails, fork() returns -1
- Child inherit copies of paren't saddress space and open file descriptors
- Do not use in new code use posix_spawn instead
- fork() is simple buy prone to subtle bugs

Example: fork()

```
$ dcc fork.c
$ a.out

I am the parent because fork() returned 2884551.

I am the child because fork() returned 0.
$
```

- The code after fork() runs twice, one for the child and the other for the parent

```
#include <stdio.h>
#include <unistd.h>

int main(void) {

    // fork creates 2 identical copies of program
    // only return value is different

    pid_t pid = fork();

    if (pid == -1) {
         perror("fork");  // print why the fork failed
    } else if (pid == 0) {
        printf("I am the child because fork() returned %d.\n", pid);
    } else {
        printf("I am the parent because fork() returned %d.\n", pid);
    }

    return 0;
}
```

execvp() – Replace yourself

```
#include <unistd.h>

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

- Replaces current process by executing file
  - file must be an executable: binary or script starting with #!
- argv specifies argv of new program
- most of the current process is reset
  - e.g. new virtual address space is created, signal handlers reset
- new process inherits open file descriptors from original process
- on error, returns -1 and sets errno
- if successful, does not return
- exec, execv, execvp, execl (vector, vector pointer, list, etc)

Example: Using exec()

```
$ dcc exec.c
$ a.out
good-bye cruel world
$
```

```c
#include <stdio.h>
#include <unistd.h>

// simple example of program replacing itself with exec
int main(void) {
    char *echo_argv[] = {"/bin/echo","good-bye","cruel","world",NULL};
    execv("/bin/echo", echo_argv);

    // if we get here there has been an error
    perror("execv");
    return 1;
}
```

Example: Combining fork() and exec() to run /bin/date

- simple example of classic fork/exec run date --utc to print current UTC
- use posix_spawn instead

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid = fork();

    if (pid == -1) {
         perror("fork"); // print why fork failed
    } else if (pid == 0) { // child

        char *date_argv[] = {"/bin/date", "--utc", NULL};

        execv("/bin/date", date_argv);

        perror("execvpe"); // print why exec failed

    } else { // parent

        int exit_status; // we gotta wait for child to finish first
        if (waitpid(pid, &exit_status, 0) == -1) {
            perror("waitpid");
            exit(1);
        }
        printf("/bin/date exit status was %d\n", exit_status);
    }

    return 0;
}
```

system() – convenient but unsafe way to run anther program

```c
#include <stdlib.h>
int system(const char *command);
```

- runs commands via /bin/sh
- waits for command to finish and returns exit status
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only

Example: system.c

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    // system passes string to a shell for evaluation
    // brittle and highly vulnerable to security exploits
    // system is suitable for quick debugging and throw-away programs only
    // run date --utc to print current UTC
    int exit_status = system("/bin/date --utc");
    printf("/bin/date exit status was %d\n", exit_status);
    return 0;
}
```

Example: Running ls -ld via posix_spawn

```c
#include <stdio.h>
#include <stdlib.h>
#include <spawn.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    // to fit all arguments
    char *ls_argv[argc + 2];
    ls_argv[0] = "/bin/ls";
    ls_argv[1] = "-ld";
    for (int i = 1; i <= argc; i++) {
        ls_argv[i + 1] = argv[i];
    }

    // remember this occurs twice for the parent and the child
    pid_t pid;
    extern char **environ;
    if (posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_argv, environ) != 0) {
        perror("spawn");
        exit(1);
    }

    // we gotta wait for the child to finish!
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }

    // exit with whatever status ls exited with
    return exit_status;
}
```

Example: Running ls -ld via system

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *ls = "/bin/ls -ld";
    int command_length = strlen(ls);
    for (int i = 1; i < argc; i++) {
        command_length += strlen(argv[i]) + 1;
    }
    // create command as string
    char command[command_length + 1];
    strcpy(command, ls);
    for (int i = 1; i <= argc; i++) {
        strcat(command, " "); // actually useful, strcat!
        // adds " " to end of command string, then returns a pointer to command
        strcat(command, argv[i]);
    }
    int exit_status = system(command);
    return exit_status;
}
```

getpid and getppid

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- getpid returns the process ID of the current process
- getppid returns the process ID of the parent of the current process

waitpid

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *wstatus, int options);
pid_t wait(int *wstatus);
```

- waitpid pauses the current process until process pid changes state
    - where state changes include finishing, stopping, re-starting, …
- ensures that child resources are released on exist
- special values for pid …
    - if `pid = -1`, wait on any child process
    - if `pid = 0`, wait on any child in process group
    - if `pid > 0`, wait on the specified process

`pid_t wait(int *status)`

- equivalent to `waitpid(-1, &status, 0)`
- pauses until one of the child processes terminates

`waitpid(pid, &status, options)`

- status is set to hold info about `pid`
- e.g. exit status if `pid` is terminated
- macros allow precise determination of state change (e.g. `WIFEXITED(status)`, `WCOREDUMP(status)`)
- options provide variations in `waitpid()` behaviour
    - default: wait for child process to terminate
    - `WNOHANG`: return immediately if no child has exited
    - `WCONTINUED`: return if a stopped child has been restarted
- More info: `man 2 waitpid`

Linux/environment variables

- When linux/unix program are passed environment variables
- Environment variables are array of strings of from name = value
- Array is NULL-terminated
- Access via global variable environ
- Many C implementation also provide as 3rd parameter to main:
- Most program use getenv and setenv to access environmental variables
- Can access environment variables directly, e.g.

```c
#include <stdio.h>

int main(void) {
    // print all environment variables
    extern char **environ;

    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

Some output:

```
cadPATH=/usr/local/cad/bin:/usr/local/cad/other/bin:/usr/local/cad/ucb/bin
MANPATH=/import/ravel/5/z5308693/man:/usr/local/man:/usr/local/X11/man:/us
r/man:/usr/share/man
TERM_PROGRAM=vscode
localPATH=/usr/local/bin
gnuMANPATH=/usr/local/gnu/man
TERM=xterm-256color
SHELL=/usr/local/bin/bash
AMD_ENTRYPOINT=vs/server/remoteExtensionHostProcess
SSH_CLIENT=115.64.1.151 63286 22
TERM_PROGRAM_VERSION=1.51.1
etcPATH=/usr/local/etc:/usr/etc:/etc:/sbin:/usr/sbin
USER=z5308693
```

Accessing an environment variable with getenv

```
#include <stdlib.h>
char *getenv(const char *name);
```

- Search environment variable array from `name = value`
- Returns `value`
- Returns `NULL` if `name` is not in environment variable array

```
$ dcc get_status.c -o get_status
$ STATUS=ok ./get_status

Environment variable 'STATUS' has value 'ok'
$
```

```c
#include <stdio.h>
#include <stdlib.h>

// simple example of accessing an environment variable
int main(void) {
    // print value of environment variable STATUS
    char *value = getenv("STATUS");
    printf("Environment variable 'STATUS' has value '%s'\n", value);
    return 0;
}
```

Setting an environment variables with setenv

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

- Adds `name = value` to environment variable array
- If `name` in array, value change if `overwrite` is non-zero

```
$ dcc set_status.c -o set_status
$ dcc get_status.c -o get_status
$ ./set_status

Environment variable 'STATUS' has value 'great'
$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <spawn.h>
#include <sys/wait.h>
// simple example of setting an environment variable
int main(void) {
    // set environment variable STATUS
    setenv("STATUS", "great", 1);
    char *getenv_argv[] = {"./get_status", NULL};
    pid_t pid;
    extern char **environ;
    if (posix_spawn(&pid, "./get_status", NULL, NULL, // will do this command! And its arguments!
        getenv_argv, environ) != 0) {
        perror("spawn");
        exit(1);
    }
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }

    // exit with whatever status s exited with
    return exit_status;
}
```

Examples: Changing behaviour with an environment variable

```c
#include <stdio.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid;

    char *date_argv[] = { "/bin/date", NULL }; // remember the array is NULL-terminated
    char *date_environment[] = { "TZ=Australia/Perth", NULL };
    // print time in Perth
    if (posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv,
                    date_environment) != 0) {
        perror("spawn");
        return 1;
    }

    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        return 1;
    }

    printf("/bin/date exit status was %d\n", exit_status);
    return 0;
}
```

exit() – terminate yourself

```c
#include <stdlib.h>

void exit(int status);
```

- Triggers any functions registered as `atexit()`, so any functions pointed by `atexit()` will run
- Flushes `stdio` buffers; closes open `FILE *`'s
- Terminates current process
- A SIGCHILD signal is sent to parent
- Returns status to parent (via `waitpid()`)
- Any child processes are inherited by `init`

`_exit(int status)`

- Terminates current process without triggering functions registered as `atexit()`
- `stdio` buffers are not flushed

pipe() – stream bytes between processes

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- a pipe is a one way byte stream provided by the OS
- can connect two processes together!
- pipefd[0] – set to file descriptor of read end of pipe
- pipefd[1] – set to file descriptor of write end of pipe
- bytes written to pipefd[1] will be read from pipefd[0] (wrong in lecture slide)
- child processes (by default) inherit file descriptors including for pipe
- parent can send/receive bytes (not both) to child via pipe
- parent and child should both close the pipe file descriptor if they aren't using it! (look at examples)
    - e.g. if bytes being written (sent) parent to child
        - parent should close read end pipefd[0]
        - child should close write end pipefd[1]
- pipe (and other) file descriptors can be used with stdio via fdopen

popen() – convenient but unsafe way to set up pipe

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- runs `command` via `/bin/sh`
- if `type` is "w" pipe to stdin of `command` created
- if `type` is "r" pipe from stdout of `command` created
- `FILE *` stream returned – get then use `fgetc`/`fputc` etc
- `NULL` returned if error
- Close stream with `pclose` (not `fclose`)
    - `pclose` waits for `command` and returns exit name
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only

Example: Capture output from a process

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    // popen passes string to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    // popen is suitable for quick debugging and throw-away programs only
    FILE *p = popen("/bin/date --utc", "r");
    if (p == NULL) {
        perror("");
        return 1;
    }
    char line[256];
    if (fgets(line, sizeof line, p) == NULL) {
        fprintf(stderr, "no output from date\n");
        return 1;
    }
    printf("output captured from /bin/date was: '%s'\n", line);

    pclose(p); // returns command exit status
    return 0;
}
```

Example: Sending input to a process

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {

    // popen passes command to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    // popen is suitable for quick debugging and throw-away programs only
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to upper case

    FILE *p = popen("tr a-z A-Z", "w"); // we wanna write input into this process
    if (p == NULL) {
        perror("");
        return 1;
    }

    fprintf(p, "plz date me\n");

    pclose(p); // returns command exit status
    return 0;
}
```

Example: Using a pipe with posix_spawn to capture output

```c
#include <stdio.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int main(void) {
    // create a pipe
    int pipe_file_descriptors[2];
    if (pipe(pipe_file_descriptors) == -1) {
        perror("pipe");
        return 1;
    }

    // create a list of file actions to be carried out on spawned process
    posix_spawn_file_actions_t actions;
    if (posix_spawn_file_actions_init(&actions) != 0) {
        perror("posix_spawn_file_actions_init");
        return 1;
    }

    // tell spawned process to close unused read end of pipe
    // without this - spawned process would not receive EOF
    // when read end of the pipe is closed below,
    if (posix_spawn_file_actions_addclose(&actions, pipe_file_descriptors[0])
!= 0) {
        perror("posix_spawn_file_actions_init");
        return 1;
    }
```

```
    // tell spawned process to replace file descriptor 1 (stdout)
    // with write end of the pipe
    if (posix_spawn_file_actions_adddup2(&actions, pipe_file_descriptors[1],
1) != 0) {
        perror("posix_spawn_file_actions_adddup2");
        return 1;
    }

    pid_t pid;
    extern char **environ;
    char *date_argv[] = {"/bin/date", "--utc", NULL};
    if (posix_spawn(&pid, "/bin/date", &actions, NULL, date_argv, environ) !=
0) {
        perror("spawn");
        return 1;
    }

    // close unused write end of pipe
    // in some case processes will deadlock without this
    // not in this case, but still good practice
    close(pipe_file_descriptors[1]);

    // create a stdio stream from read end of pipe
    FILE *f = fdopen(pipe_file_descriptors[0], "r");
    if (f == NULL) {
        perror("fdopen");
        return 1;
    }

    // read a line from read-end of pipe
    char line[256];
    if (fgets(line, sizeof line, f) == NULL) {
        fprintf(stderr, "no output from date\n");
        return 1;
    }

    printf("output captured from /bin/date was: '%s'\n", line);

    // close read-end of the pipe
    // spawned process will now receive EOF if attempts to read input
    fclose(f);

    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        return 1;
    }
    printf("/bin/date exit status was %d\n", exit_status);

    // free the list of file actions
    posix_spawn_file_actions_destroy(&actions);

    return 0;
```

```
}
```

Example: Using a pipe with posix_spawn to send input to spawned process

```c
#include <stdio.h>
#include <unistd.h>
#include <spawn.h>
#include <sys/wait.h>

int main(void) {
    // create a pipe
    int pipe_file_descriptors[2];
    if (pipe(pipe_file_descriptors) == -1) {
        perror("pipe");
        return 1;
    }

    // create a list of file actions to be carried out on spawned process
    posix_spawn_file_actions_t actions;
    if (posix_spawn_file_actions_init(&actions) != 0) {
        perror("posix_spawn_file_actions_init");
        return 1;
    }

    // tell spawned process to close unused write end of pipe
    // without this - spawned process will not receive EOF
    // when write end of the pipe is closed below,
    // because spawned process also has the write-end open
    // deadlock will result
    if (posix_spawn_file_actions_addclose(&actions, pipe_file_descriptors[1])
!= 0) {
        perror("posix_spawn_file_actions_init");
        return 1;
    }

    // tell spawned process to replace file descriptor 0 (stdin)
    // with read end of the pipe
    if (posix_spawn_file_actions_adddup2(&actions, pipe_file_descriptors[0],
0) != 0) {
        perror("posix_spawn_file_actions_adddup2");
        return 1;
    }


    // create a process running /usr/bin/sort
    // sort reads lines from stdin and prints them in sorted order
    char *sort_argv[] = {"sort", NULL};
    pid_t pid;
    extern char **environ;
    if (posix_spawn(&pid, "/usr/bin/sort", &actions, NULL, sort_argv, environ)
!= 0) {
        perror("spawn");
        return 1;
    }

    // close unused read end of pipe
    close(pipe_file_descriptors[0]);
```

```
    // create a stdio stream from write-end of pipe
    FILE *f = fdopen(pipe_file_descriptors[1], "w");
    if (f == NULL) {
        perror("fdopen");
        return 1;
    }

    // send some input to the /usr/bin/sort process
    //sort with will print the lines to stdout in sorted order
    fprintf(f, "sort\nwords\nplease\nthese\n");

    // close write-end of the pipe
    // without this sort will hang waiting for more input
    fclose(f);

    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        return 1;
    }
    printf("/usr/bin/sort exit status was %d\n", exit_status);

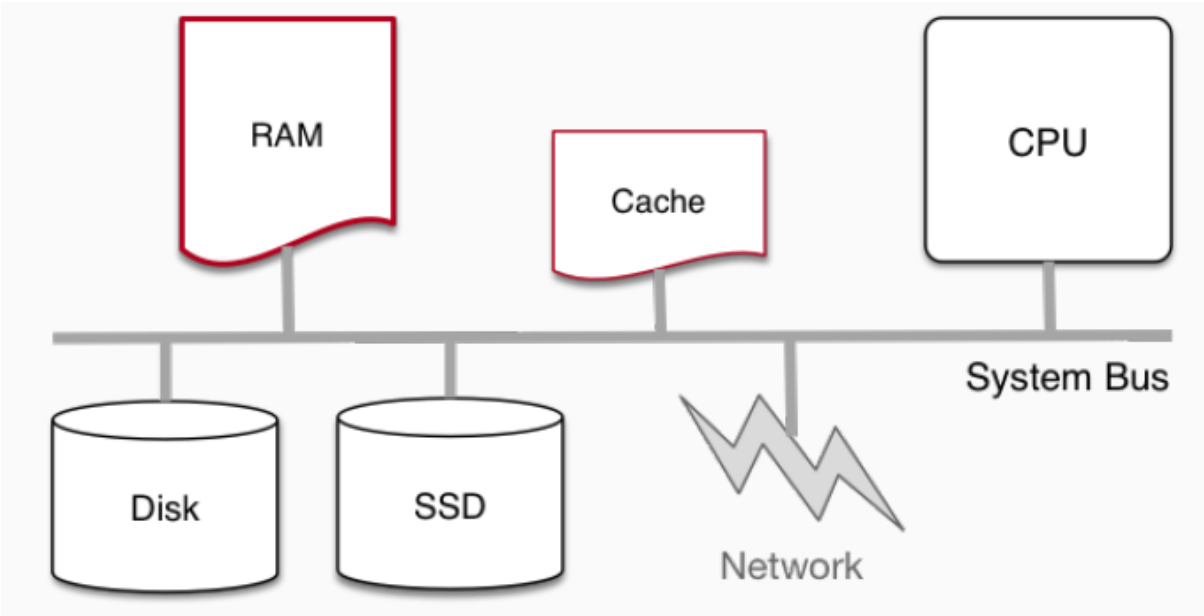
    // free the list of file actions
    posix_spawn_file_actions_destroy(&actions);

    return 0;
}
```

Virtual Memory

Memory

- Systems typically contain 4-16GB of volatile RAM



- Plus a hierarchy of smaller cache memory – on or off the CPU chip

Single Process Resident in RAM without Operating System

- Many small embedded systems run without an operating system
- Single program running, probably written in C
- Devices (sensors, switches, …) often wired at particular address
- E.g. can set motor speed by storing byte at 0x100400
- Program accesses (any) RAM directly
- Development and debugging is tricky (no printf or IDE – debug using an LED or multimeter)
- Widely used for simple micro-controllers
- Parallelism and exploiting multiple-core CPUs problematic

Single Process Resident in RAM with Operating System

- OS need (simple) hardware support
- Part of RAM (kernel space) must be accessible only in a privileged mode
- System call enables privileged mode and passes execution to operating system code in kernel space
- Privileged mode disabled when system call returns
- Privileged mode could be implemented by a bit in a special register
- If only one process resident in RAM at any time – switching between processes is slow
- OS must write out the memory of the old process to the disk and read the memory of the new process from the disk
- It's inefficient
- Example: SPIM

Multi Processes Resident in RAM without Virtual memory

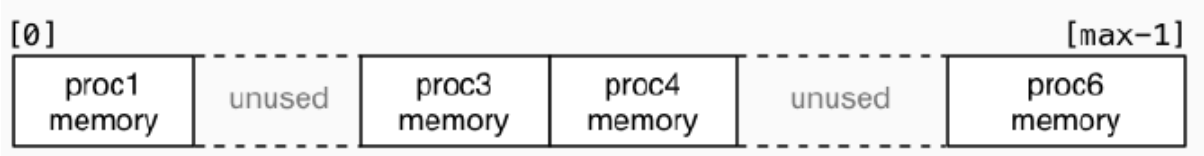
- If multiple processes to be resident in RAM, O/S can swap execution between them quickly
- RAM belonging to other processes and kernel must be protected
- Hardware support can limit access to a region of RAM
- But program can run anywhere on the RAM
- However, this breaks instructions which use absolute addresses: lw, sw, jr
- Either programs can't use absolute memory address (relocatable code) or code has to be recompiled before it's run – not possible for all code!
- This is a major limitation – it's much better if the program can assume it always has the same address space
- This is not very used in modern computing

Virtual Memory

- BIG IDEA – disconnect address processes use from actual RAM address
- Operating system translates (virtual) address a process uses to a physical (actual) RAM address
- So convenient for programming/compilers:
    - each process has the same virtual view of RAM
    - can have multiple processes in RAM, allowing fast switching
    - can load part of processes into RAM on demand
    - provides a mechanism to share memory between processes
- However,
    - address to fetch every instruction executed must be translated
    - address for load/store instructions must be translated
    - translation needs to be really fast so largely implemented in hardware (silicon)

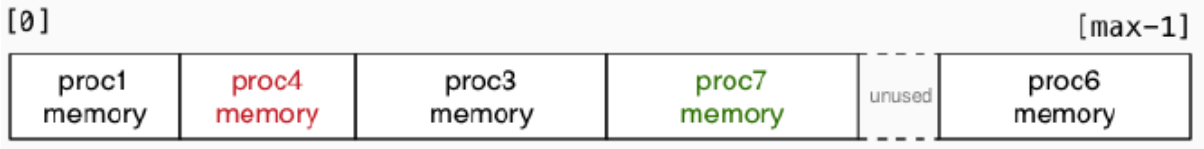
Virtual memory with One Memory Segment Per Process

- Consider a scenario with multiple processes loaded in memory:



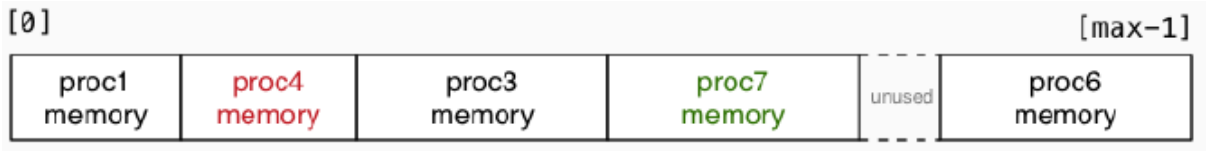
- Each process is in a contiguous region of RAM, starting at address base, finishing at address limit
- Each process sees its own address space as [0 .. size – 1]
- Process accessing memory address a is translated to a + base and is checked so that a + base < limit so that the process only accesses its own memory
- This is simple to implement in hardware!

- Consider when we want to add a new process and it can't fit:



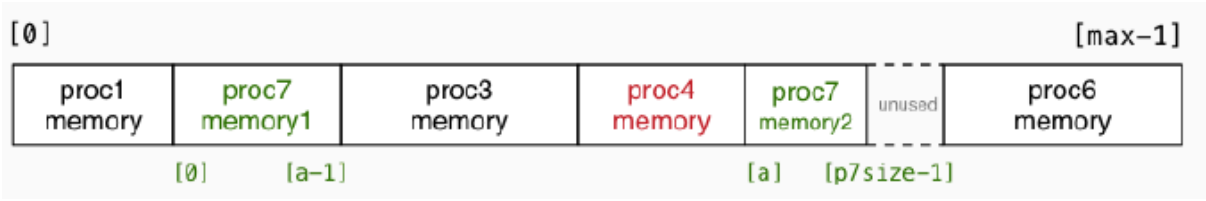
- The new process doesn't fit in any of the unused slots

- So we can move some process to make a single large slot



- But this is:
  - SLOW if RAM heavily used (everything has to be rearranged)
  - Doesn't allow sharing or lading on demand
  - Limits process address space to size of RAM (that unused space can never be used)
  - Little used in modern computing

- Better way would be to split process memory over multiple parts of physical memory



- But with arbitrary sized memory segments, translating virtual to physical address is complicated making hardware support difficult:

```
// translate virtual_address to physical RAM address
uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
    uint32_t n_segments;
    Segment *segments = get_segments(process_id, &n_segments);
    for (int i = 0; i < n_segments; i++) {
        Segment *c = &segments[i];
        if (virtual_addr >= c->base &&
            virtual_addr < c->base + c->size) {
            uint32_t offset = virtual_addr - c->base;
            return c->mem + offset;
        }
    }
// handle illegal memory access
}
```

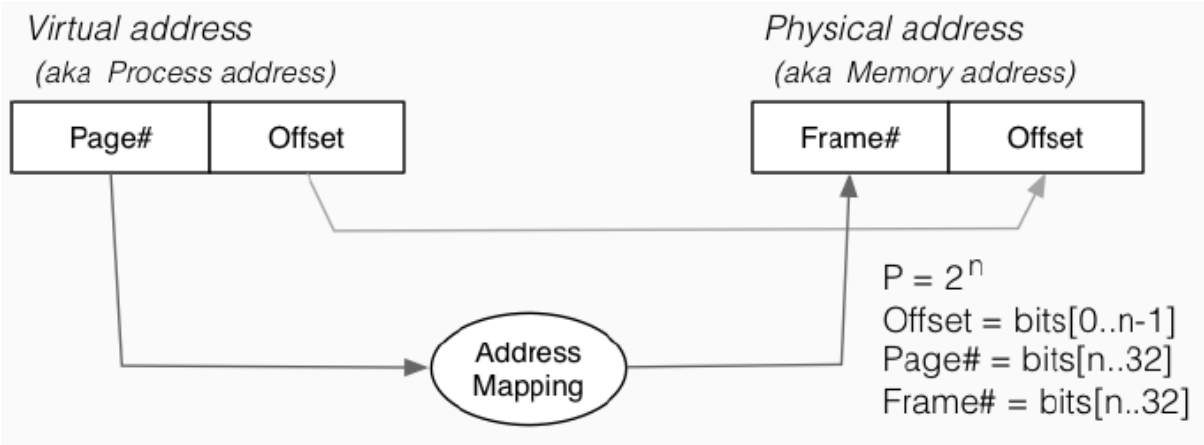
Virtual Memory with Pages

- Address mapping would be simpler if all segments were same size
  - call each segment of address space a **page**
  - make all pages the same size **P**
  - page **I** holds addresses: **I*P … (I + 1)*P**
  - each process has an array called the page table
  - each array element contains the physical address in RAM of that page
  - for virtual address **V**, **page_table[V / P]** contains physical address of page
  - the address will at be at offset **V % P** in both pages
  - so physical address for **V** is: **page_table[V / P] + V % P**

- With pages, translating virtual to physical address is simpler making hardware support difficult:

```
// translate virtual_address to physical RAM address
uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
    uint32_t pt_size;
    PageInfo *page_table = get_page_table(process_id, &pt_size);
    page_number = virtual_addr / PAGE_SIZE;
    if (page_number < pt_size) {
        uint32_t offset = virtual_addr % PAGE_SIZE;
        return PAGE_SIZE * page_table[page_number].frame + offset;
    }
    // handle illegal memory access
}
```

- Calculating of `page_number` and offset can be faster/simpler bit operatings if `PAGE_SIZE==`2^n
- Note `PageInfo` entries will have more information about the page
- If `P ==` 2^n, then address maping becomes:



- If `P = 4096 =` 2^{12}
- Bottom 12 bits stay the same in translation
- Top 2 bits is translated by array look up

Virtual Memory with Pages – Lazy Loading

- A side-effect of this type of virtual -> physical mapping
    - don't need to load all of processes pages up-from
    - start with a small memory "footprint" (e.g. main + stack tip) – so you only need to load these first, nothing else
    - load new process address pages into memory *as needed*
    - grow up to the size of (available) physical memory
- The strategy of …
    - diving process memory space into fixed-size pages
    - on-demand loading of process pages into physical memory

  is what is generally meant by *virtual memory*

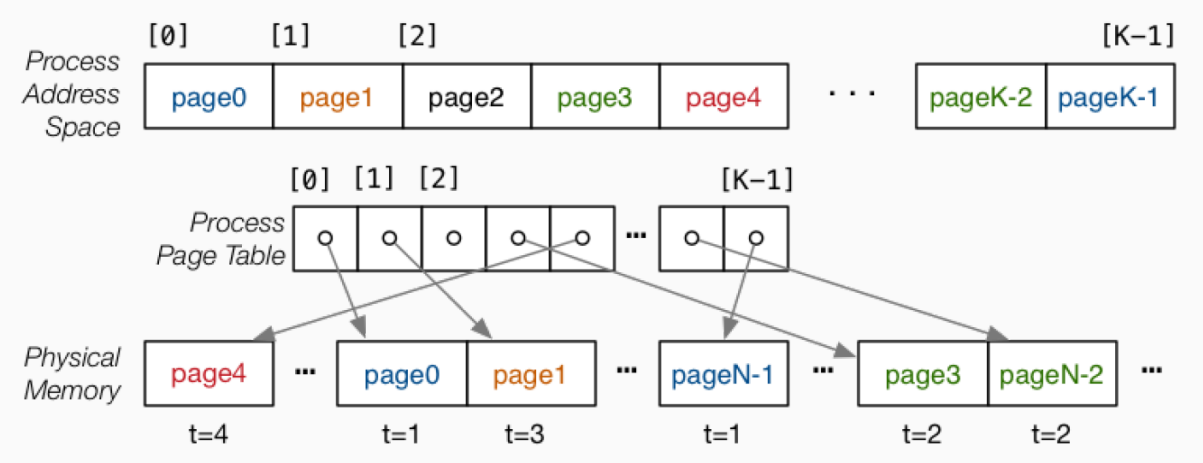
Virtual Memory

- Pages/frames are typically 4KB … 256 KB in size
- A frame is a page, but in physical memory
- Each frame can hold one page of process address space
- Leads to a memory layout like this (with L total frames of physical memory):

- When a process is complete, all its frames are released for reuse

Page Tables

- Each process has a page table, which maps to the real physical memory
- Each array element contains the physical address in RAM of that page
- Each page table entry might contain:
  - page status: not_loaded, loaded, modified
  - frame number of page (if loaded)
  - maybe others e.g. last accessed time
- We need [ProcSize/PageSize] entries in this table
- Example of page table for one process, the timestamps show when page was loaded:



Loading Pages

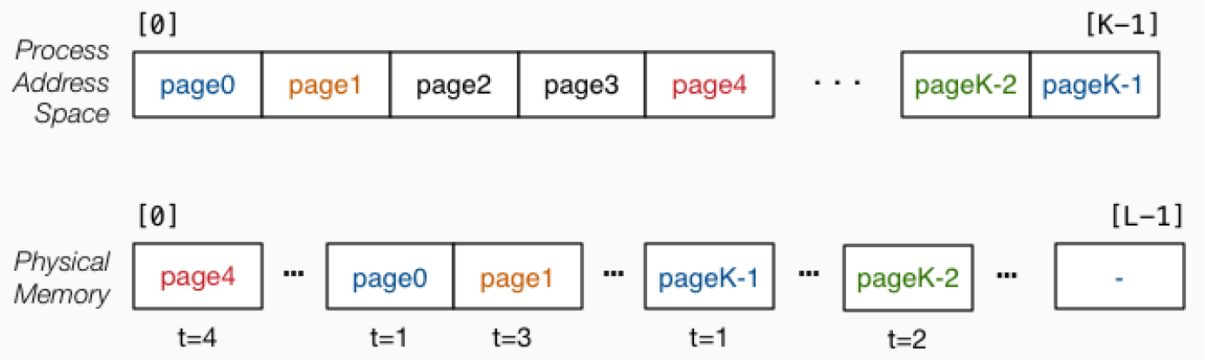
- Consider a new process commencing execution:
  - It initially has 0 pages loaded
  - Load page for main()
  - Load page for main()'s stack frame
  - Load other pages when process references address within page
- So we have a working set:
  - In any given window of time, process typically accesses a small subset of their pages, this is called locality of reference
  - The subset of pages are called the working set
  - Hence, since each process has a small working set, the RAM can hold more active processes at the same time

    - Thus, the processes address space can be larger than in physical memory (look in diagram for definitions of process address space)

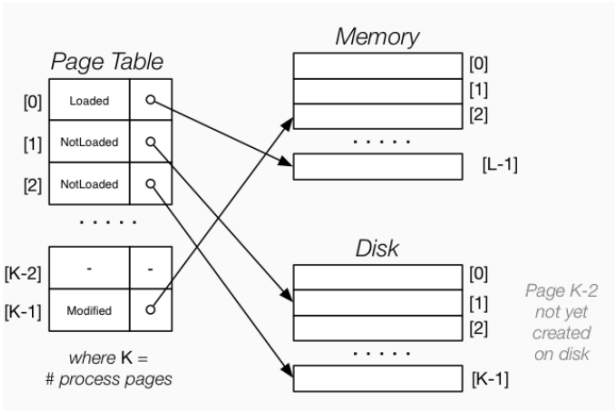
```
typedef struct {int status, int frame, ...} PageInfo;
uint32_t translate(uint32_t process_id, uint32_t virtual_addr) {
    uint32_t pt_size;
    PageInfo *page_table = get_page_table(process_id, &pt_size);
    page_number = virtual_addr / PAGE_SIZE;
    if (page_number < pt_size) {
        if (page_table[page_number].status != LOADED) {
            // page fault - need to load page into free frame
            page_table[page_number].frame = ???
            page_table[page_number].status = LOADED;
            }
        uint32_t offset = virtual_addr % PAGE_SIZE;
        return PAGE_SIZE * page_table[page_number].frame + offset;
    }
    // handle illegal memory access
}
```

- Where are pages loaded from?
    - Code is loaded from the executable file stored on disk into read-only pages
    - Some data (e.g. C strings) also loaded into read-only pages
    - Initialised data (C global/static variables) also loaded from executable file
    - Pages for uninitialized data (head, stack) are zero-ed
        - Prevents information leaking from other processes
        - Results in uninitialized local (stack) variables often containing 0
- Executable file: e.g. a.out
- Consider a process whose address space exceeds physical memory

- We can imagine that a process's address space …
    - exists on disk for the duration of the process's execution
    - and only some parts of it are in memory at any given time



- Transferring pages between disk and memory is very expensive!
- Need to ensure minimal reading from / writing to disk

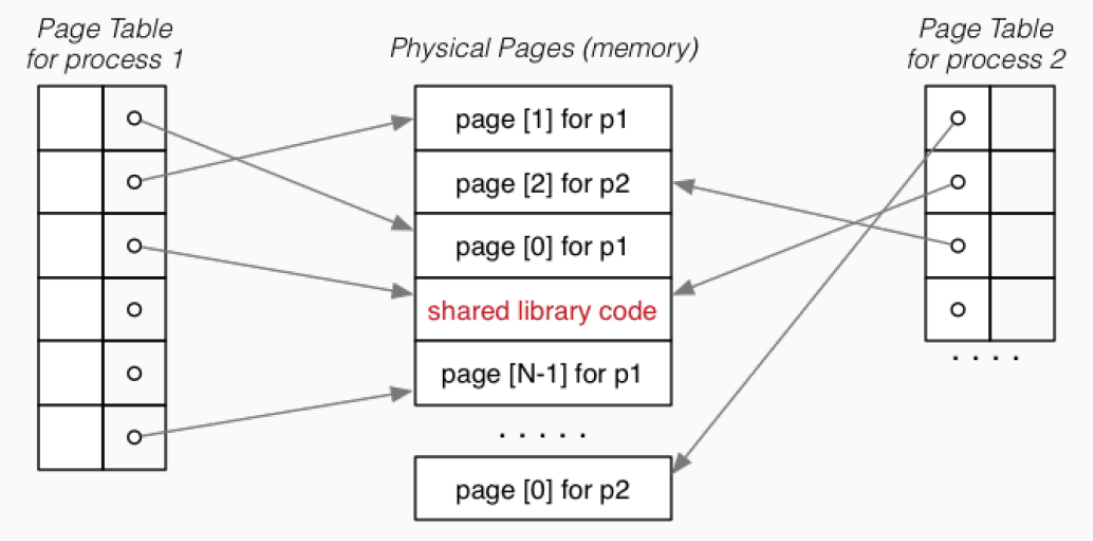


Page Faults

- An access to a page which is non-loaded in RAM is called a page fault
- Where do we load it?
    - If there is a free frame:
        - We load it there
    - If there are no free frames
        - Either suspend process until a page is freed or
        - Replace one of the currently loaded/used pages
- Suspending requires the OS to:
    - Mark the process as unable to run until page available
    - Switch to running another process
    - Mark the process as able to run when page available
- When we replace one of the currently loaded/used pages due to no free frames, we need to choose which to evict:
    - best page is one that won't be used by system again
    - prefer pages that are read only (no need to write to disk)
    - prefer pages that are unmodified (no need to write to disk)
    - prefer pages that are used by one process (shared by more than one process)
- The OS can't predict which page will be required again by the process but a good heuristic is to replace the Least Recently Used (LRU) page – it's probably not needed again soon
- Week 10 Lab Exercise 3 to this in action!

Read-only Pages

- Virtual memory allows for sharing read-only pages (e.g. library code)
- Several processes include same frame in virtual address space

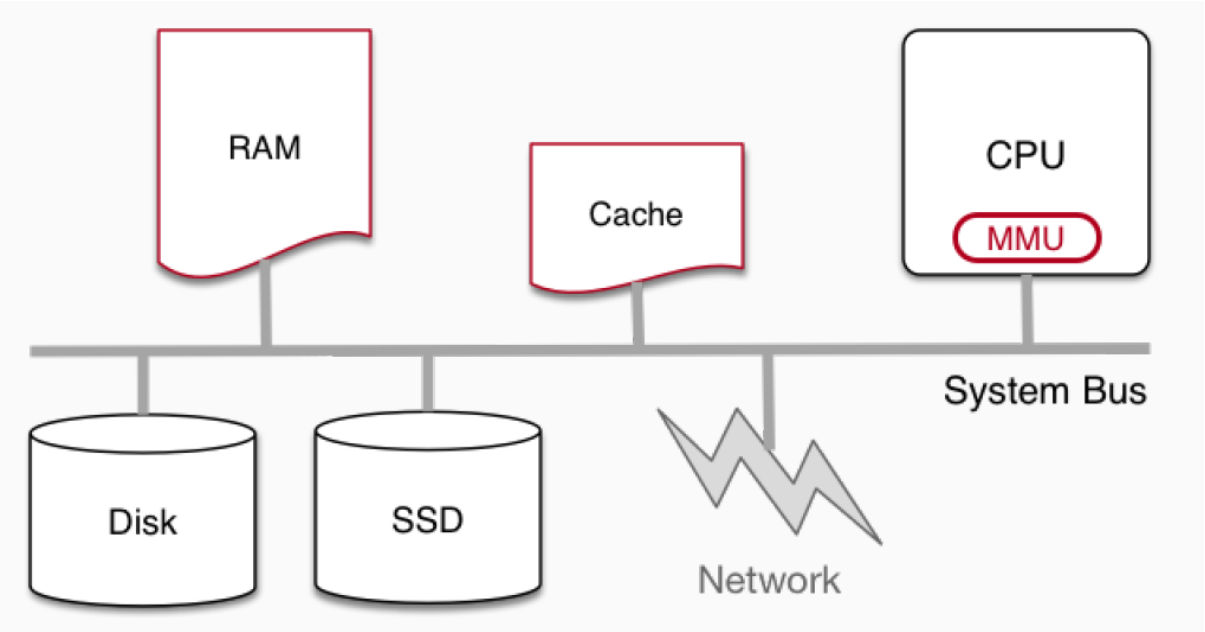


Cache Memory

- Cache memory holds parts of the RAM that are used frequently
- That are located in the CPU and are way faster than RAM
- But only stores a very small amount of data
- Transfers data to/from RAM in blocks (cache blocks)
- Memory reference hardware first looks in cache:
    - if required address is there, use its contents
    - if not, get it from RAM and put it in cache
    - possible replacing an existing cache block
- Replacement strategies have similar issue to virtual memory!

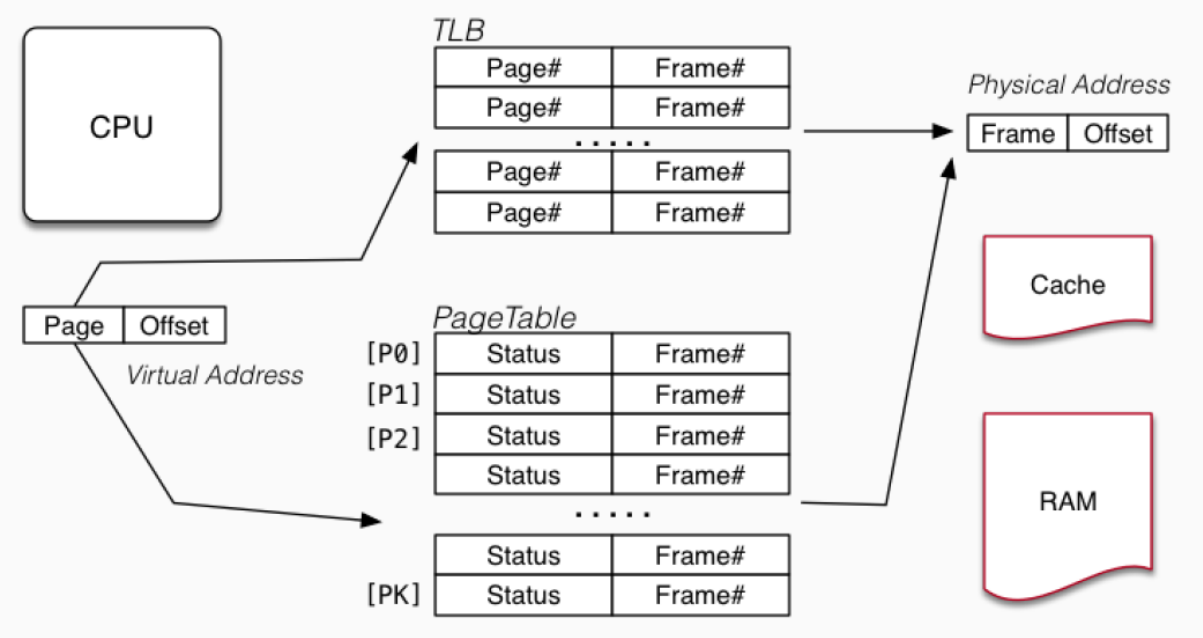
Memory Management Hardware

- Address translation is important/frequent
- Provides specialised hardware (MMU) to do it efficiently
- Sometimes location on CPU chip, sometimes sperate



- TLB = translation lookaside buffer

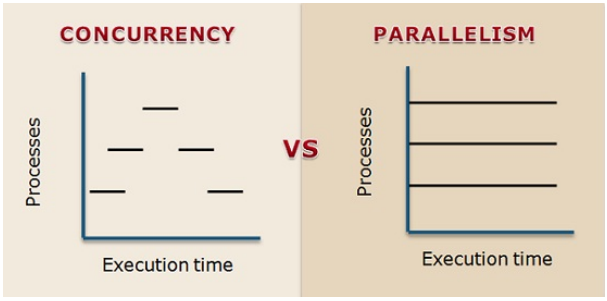
- Lookup table containing (virtual, physical) address pairs



Threads

Concurrency/Parallelism

- **Concurrency**: multiple computations in overlapping time periods; does not have to be simultaneous
- **Parallelism**: multiple computations executing simultaneously
    - can occur across computers (e.g. with MapReduce)
    - or multiple cores of a CPU executing different instructions (MIMD)
    - or multiple cores of a CPU executing same instruction (SIMD) e.g. GPU rendering pixels
- Both parallelism and concurrency need to deal with synchronisation

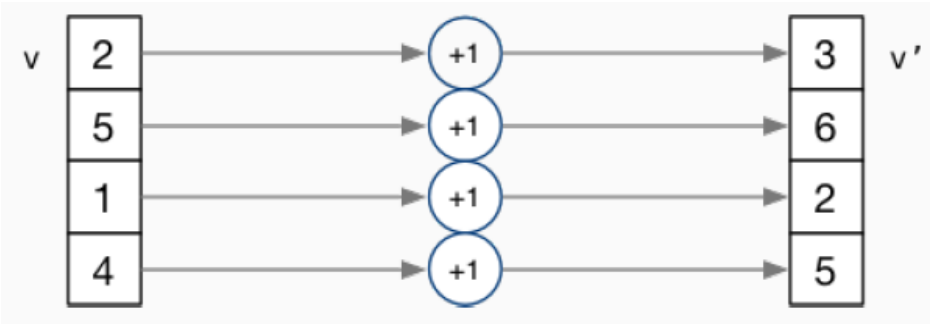


Parallel Computing Across Many Computers

- E.g MapReduce is a popular programming model for:
    - manipulating large data sets
    - on a large network for computers (local or distributed)
- The *map* step filters data and distributes it to nodes
    - Data distributed as (key, value) pairs
    - Each nod receives a set of pairs with common *key*(s)
- Nodes then perform calculation on received data items
- The *reduce* step computes the final result
    - by computing outputs (calculation results) from the nodes
- Also needs a way to determine when all calculations completed

Parallelism Across an Array

- Multiple identical processors
- Each given one element of an array from main memory
- Each performing same computation on that element (SIMD)
- Results copied back to array in main memory



- But not totally independent: need to synchronise on completion
- E.g. GPU rendering pixels or neural network

Parallelism Across Processes

- One method for creating parallelism:
- Use posix_spawn() to create multiple processes, each does part of job
    - child executes concurrently with parent
    - runs in its own address space
    - inherits some state information from parent, e.g. open fd's
- Processes have some disadvantages
    - process switching expensive
    - each require a significant amount of state (RAM)
    - communication between processes limited and/or slow
- One big advantaged – separate address space make processes more robust

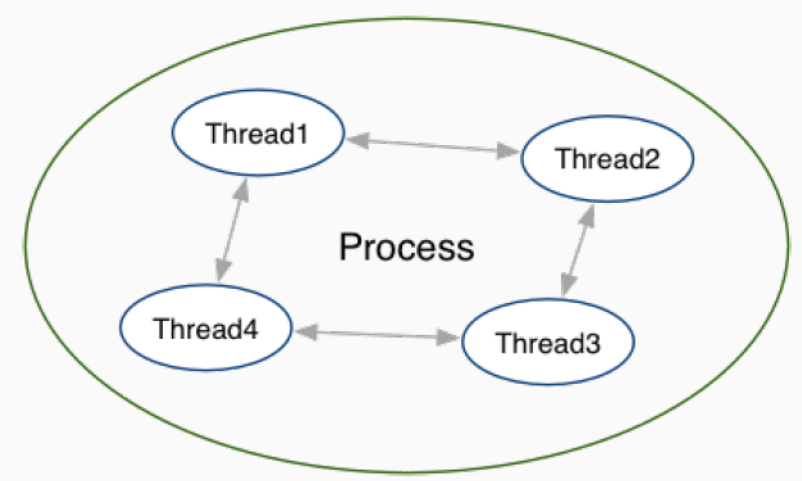
Parallelism within Processes

- **Threads** – mechanism for parallelism within process
- Threads allow simultaneous execution within process
- Each thread has its own execution state
- Threads within a process have same address space:
    - threads share code (functions)
    - threads share global and static variables
    - threads share heap (malloc)
- But separate stack for each thread
    - local variables are not shared
- Threads share file descriptor
- Threads share signals

POSIX threads (pThreads)

```
// POSIX threads widely supported in Unix-like
// and other systems (Windows). Provides functions
// to create/synchronize/destroy/... threads

#include <pthread.h>
```



Create a POSIX Thread

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

- Creates a new thread with attributes specified in `attr`
  - `attr` can be `NULL`
- Thread info stored in `*thread`
- Thread starts by executing `start_routine(arg)`
- Returns 0 if OK, -1 otherwise and sets `errno`
- Analogous to `posix_spawn()`

Wait for a POSIX Thread

```
int pthread_join(pthread_t thread, void **retval)
```

- Wait until thread finishes
- thread return (or `pthread_exit()`) value is placed in `*retval`
- If thread has already exited, does not wait
- If main returns or exit called, all threads terminated
- Programs typically need `towait` for all threads before main returns/exit called
- Analogous to `waitpid`

Terminate a POSIX Thread

- Terminate execution of thread (and free resources)
- `retval` is returned (see `pthread_join`)
- Analogous to `exit`

Example: Create two threads

```
simple example which launches two threads of execution


$ gcc -pthread two_threads.c -o two_threads
$ ./two_threads|more

Hello this is thread #1 i=0

Hello this is thread #1 i=1

Hello this is thread #1 i=2

Hello this is thread #1 i=3

Hello this is thread #1 i=4

Hello this is thread #2 i=0

Hello this is thread #2 i=1
...
```

```c
#include <stdio.h>
#include <pthread.h>

// this function is called to start thread execution
// it can be given any pointer as argument (int *) in this example

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 10; i++) {
        printf("Hello this is thread #%d: i=%d\n",  *p, i);
    }

    // a thread finishes when the function returns or  thread_exit is called
    // a pointer of any type can be returned
    // this can be obtained via thread_join's 2nd argument
    return NULL;
}

int main(void) {
    //create two threads performing almost the same task

    pthread_t thread_id1;
    int thread_number1 = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number1);

    int thread_number2 = 2;
    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, run_thread, &thread_number2);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

Example: Classic Bug – Sharing a variable between threads

```
simple example which launches two threads of execution
 but demonstrates the perils of accessing non-local variables
 from a thread


$ gcc -pthread two_threads_broken.c -o two_threads_broken
$ ./two_threads_broken|more

Hello this is thread 2: i=0

Hello this is thread 2: i=1

Hello this is thread 2: i=2

Hello this is thread 2: i=3

Hello this is thread 2: i=4

Hello this is thread 2: i=5

Hello this is thread 2: i=6
```

```
Hello this is thread 2: i=7

Hello this is thread 2: i=8

Hello this is thread 2: i=9

Hello this is thread 2: i=0

Hello this is thread 2: i=1

Hello this is thread 2: i=2

Hello this is thread 2: i=3

Hello this is thread 2: i=4

Hello this is thread 2: i=5

Hello this is thread 2: i=6

Hello this is thread 2: i=7

Hello this is thread 2: i=8

Hello this is thread 2: i=9
$...
```

```c
#include <stdio.h>
#include <pthread.h>

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 10; i++) {

        // variable thread_number will probably have changed in main
        // before execution reaches here
        printf("Hello this is thread %d: i=%d\n",  *p, i);

    }

    return NULL;
}

int main(void) {
    pthread_t thread_id1;
    int thread_number = 1;
    pthread_create(&thread_id1, NULL, run_thread, &thread_number);

    thread_number = 2;
    pthread_t thread_id2; // same thread_number
    pthread_create(&thread_id2, NULL, run_thread, &thread_number);

    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);
    return 0;
}
```

Example: Creating many threads

```
simple example of running an arbitrary number of threads
 for example:


$ gcc -pthread n_threads.c -o n_threads
$ ./n_threads 10

Hello this is thread 0: i=0

Hello this is thread 0: i=1

Hello this is thread 0: i=2

Hello this is thread 0: i=3

Hello this is thread 0: i=4

Hello this is thread 0: i=5

Hello this is thread 0: i=6

Hello this is thread 0: i=7
...
```

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

void *run_thread(void *argument) {
    int *p = argument;

    for (int i = 0; i < 42; i++) {
        printf("Hello this is thread %d: i=%d\n",  *p, i);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <n-threads>\n", argv[0]);
        return 1;
    }
    int n_threads = strtol(argv[1], NULL, 0);
    assert(n_threads > 0 && n_threads < 100);

    pthread_t thread_id[n_threads];
    int argument[n_threads];

    for (int i = 0; i < n_threads; i++) {
        argument[i] = i;
        pthread_create(&thread_id[i], NULL, run_thread, &argument[i]);
    }

    // wait for the threads to finish
    for (int i = 0; i < n_threads;i++) {
        pthread_join(thread_id[i], NULL);
    }

    return 0;
}
```

Example: Dividing a task between threads

```
simple example of dividing a task between n-threads


 compile like this:


$ gcc -O3 -pthread thread_sum.c -o thread_sum


 one thread takes 10 seconds


$ time ./thread_sum 1 10000000000

Creating 1 threads to sum the first 10000000000 integers

Each thread will sum 10000000000 integers

Thread summing integers          0 to 10000000000 finished sum is 49999999990067863552



Combined sum of integers 0 to 10000000000 is 49999999990067863552


real    0m11.924s
user    0m11.919s
sys 0m0.004s
$
# Four threads runs 4x as fast on a machine with 4 cores

$ time ./thread_sum 4 10000000000
Creating 4 threads to sum the first 10000000000 integers
Each thread will sum 2500000000 integers
Thread summing integers 2500000000 to  5000000000 finished sum is
9374999997502005248
Thread summing integers 7500000000 to 10000000000 finished sum is
21874999997502087168
Thread summing integers 5000000000 to  7500000000 finished sum is
15624999997500696576
Thread summing integers          0 to  2500000000 finished sum is
3124999997567081472

Combined sum of integers 0 to 10000000000 is 49999999990071869440

real    0m3.154s
user    0m12.563s
sys 0m0.004s
$


Note result is inexact because we use values can't be exactly represented as
double
and exact value printed depends on how many threads we use - becayse we break
up
the computation differently depending on number of threads

*/
```

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

struct job {
    long start;
    long finish;
    double   sum;
};

void *run_thread(void *argument) {
    struct job *j = argument;
    long start = j->start;
    long finish = j->finish;
    double sum = 0;

    for (long i = start; i < finish; i++) {
        sum += i;
    }

    j->sum = sum;

    printf("Thread summing integers %10lu to %11lu finished sum is %20.0f\n", start, finish, sum);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n-threads> <n-integers-to-sum>\n", argv[0]);
        return 1;
    }

    int n_threads = strtol(argv[1], NULL, 0);
    assert(n_threads > 0 && n_threads < 1000);
    long integers_to_sum = strtol(argv[2], NULL, 0);
    assert(integers_to_sum > 0);

    long integers_per_thread = (integers_to_sum - 1)/n_threads + 1;

    printf("Creating %d threads to sum the first %lu integers\n",
           n_threads, integers_to_sum);
    printf("Each thread will sum %lu integers\n", integers_per_thread);

    pthread_t thread_id[n_threads];
    struct job jobs[n_threads];

    for (int i = 0; i < n_threads; i++) {
        jobs[i].start = i * integers_per_thread;
        jobs[i].finish = jobs[i].start + integers_per_thread;
        if (jobs[i].finish > integers_to_sum) {
            jobs[i].finish = integers_to_sum;
        }

        // create a thread which will sum integers_per_thread integers
        pthread_create(&thread_id[i], NULL, run_thread, &jobs[i]);
    }

    // wait for each threads to finish
    // then add its individual sum to the overall sum
    double overall_sum = 0;
    for (int i = 0; i < n_threads;i++) {
        pthread_join(thread_id[i], NULL);
        overall_sum += jobs[i].sum;
    }

    //
    printf("\nCombined sum of integers 0 to %lu is %.0f\n",
           integers_to_sum, overall_sum);
    return 0;
}
```

Example: Unsafe Access to Global Variable

```
simple example demonstrating unsafe access to a global variable from threads


$ gcc -O3 -pthread bank_account_broken.c -o bank_account_broken
$ ./bank_account_broken

Andrew's bank account has $108829
$
```

```c
#define _POSIX_C_SOURCE 199309L

#include <stdio.h>
#include <pthread.h>
#include <time.h>

int bank_account = 0;

// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {

        // execution may switch threads in middle of assignment
        // between load of variable value
        // and store of new variable value
        // changes other thread makes to variable will be lost
        nanosleep(&(struct timespec){.tv_nsec = 1}, NULL);
        bank_account = bank_account + 1;
    }

    return NULL;
}

int main(void) {
    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will probably be much less than $200000
    printf("Andrew's bank account has $%d\n", bank_account);
    return 0;
}
```

Global Variable: Race Condition

- Incrementing a global variable is not an atomic (indivisible) option

In C:

```
int bank_account;
void *thread(void *a) {
// ...
bank_account++;
// ...
}
```

In MIPS:

```
la $t0, bank_account
lw $t1, ($t0)
addi $t1, $t1, 1
sw $t1, ($t0)
.data
bank_account: .word 0
```

- If bank_account == 42 and two threads increment simultaneously

```
la $t0, bank_account
lw $t1, ($t0)
# $t1 == 42
addi $t1, $t1, 1
# $t1 == 43
sw $t1, ($t0)
# bank_account == 43
```

```
la $t0, bank_account
lw $t1, ($t0)
# $t1 == 42
addi $t1, $t1, 1
# $t1 == 43
sw $t1, ($t0)
# bank_account == 43
```

- One increment is lost at store word!
- Note threads don't share registers or stack (local variable). They do share global variables!

- If bank_account == 100 and two thread change simultaneously

```
la $t0, bank_account
lw $t1, ($t0)
# $t1 == 100
addi $t1, $t1, 100
# $t1 == 200
sw $t1, ($t0)
# bank_account == ?
```

```
la $t0, bank_account
lw $t1, ($t0)
# $t1 == 100
addi $t1, $t1, -50
# $t1 == 50
sw $t1, ($t0)
# bank_account == ?
```

- Will the result be 50 or 200?

Exclude Other Threads from Code

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Only one thread can enter a *critical section*
- Estables mutal exclusion – *mutex*
- Call `pthread_mutex_lock` before and call `pthread_mutex_unlock` after
- Only 1 thread can execute in protected code
- For example:

```
pthread_mutex_lock(&bank_account_lock);
andrews_bank_account += 1000000;
pthread_mutex_unlock(&bank_account_lock);
```

Example: Protecting Access to Global Variable with a Mutex

```
simple example demonstrating safe access to a global variable from threads
 using a mutex (mutual exclusion) lock


$ gcc -O3 -pthread bank_account_mutex.c -o bank_account_mutex
$ ./bank_account_mutex

Andrew's bank account has $200000
$
```

```c
#include <stdio.h>
#include <pthread.h>

int bank_account = 0;
pthread_mutex_t bank_account_lock = PTHREAD_MUTEX_INITIALIZER;

// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {

        pthread_mutex_lock(&bank_account_lock);
        // only one thread can execute this section of code at any time
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
    return NULL;
}
int main(void) {
    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will always be $200000
    printf("Andrew's bank account has $%d\n", bank_account);
    return 0;
}
```

Semaphores

- Semaphores are special variables which provide a more general synchronisation mechanism that mutexes

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- `sem_init` initialises `sem` to `value`
- `sem_wait` – classically called `P()`
    - if `sem > 0`, decrement `sem` and continue
    - otherwise, wait until `sem > 0`
- `sem_post` – classically called `V()`
    - increment sem and continue

Example: Semaphores – allow n threads to a resource

```
#include <semaphore.h>
sem_t sem;
sem_init(&sem, 0, n);

sem_wait(&sem);
// only n threads can be in executing
// in here simultaneously
sem_post(&sem);
```

Example: Protecting Access to Global Variable with a Semaphore

```
simple example demonstrating ensuring safe access to a global variable from
threads
 using a semaphore


$ gcc -O3 -pthread bank_account_semphore.c -o bank_account_semphore
$ ./bank_account_semphore

Andrew's bank account has $200000
$
```

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int bank_account = 0;

sem_t bank_account_semaphore;

// add $1 to Andrew's bank account 100,000 times
void *add_100000(void *argument) {

    for (int i = 0; i < 100000; i++) {

        // decrement bank_account_semaphore if > 0
        // otherwise wait until > 0
        sem_wait(&bank_account_semaphore);

        // only one thread can execute this section of code at any time
        // because  bank_account_semaphore was initialized to 1

        bank_account = bank_account + 1;

        // increment bank_account_semaphore
        sem_post(&bank_account_semaphore);
    }

    return NULL;
}

int main(void) {
    // initialize bank_account_semaphore to 1
    sem_init(&bank_account_semaphore, 0, 1);

    //create two threads performing  the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, add_100000, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, add_100000, NULL);

    // wait for the 2 threads to finish
    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    // will always be $200000
    printf("Andrew's bank account has $%d\n", bank_account);

    sem_destroy(&bank_account_semaphore);
    return 0;
}
```

File Locking

```
int flock(int FileDesc, int Operation)
```

- Similar to mutexes for a file
- Controls access to shared files (note: files not fds)
- Possible operations
    - LOCK_SH … acquire shared lock
    - LOCK_EX … acquire exclusive lock
    - LOCK_UN … unlock
    - LOCK_NB … operation fails rather than blocking
- In blocking mode, `flock()` does not return until lock available
- Only works correctly if all processes accessing file use locks
- Return value: 0 in success, -1 on failure

- If a process tries to acquire a shared lock …
    - if file not locked or shared locks, OK
    - if file has exclusive lock, blocked
- If a process trues to acquire an exclusive lock …
    - if file is not locked, OK
    - if any locks (shared or exclusive) on file, blocked
- If using a non-blocking lock
    - `flock()` returns 0 if lock was acquired
    - `flock()` returns -1 if process would have been blocked

Concurrent Programming is Complex

- Concurrency is complex with many issues beyond this course:
    - Data races: thread behaviour depends on unpredictable ordering; can produce difficult bugs or security vulnerabilities
    - Deadlock: threads stopped because they are wait on each other
    - Livelock: threads running without making progress
    - Starvation: threads never getting to run

Example: Deadlock accessing two resources

```c
#include <stdio.h>
#include <pthread.h>

int andrews_bank_account1 = 100;
pthread_mutex_t bank_account1_lock = PTHREAD_MUTEX_INITIALIZER;

int andrews_bank_account2 = 200;
pthread_mutex_t bank_account2_lock = PTHREAD_MUTEX_INITIALIZER;

// swap values between Andrew's two bank account 100,000 times
void *swap1(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&bank_account1_lock);
        pthread_mutex_lock(&bank_account2_lock);

        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;

        pthread_mutex_unlock(&bank_account2_lock);
        pthread_mutex_unlock(&bank_account1_lock);

    }

    return NULL;
}

// swap values between Andrew's two bank account 100,000 times
void *swap2(void *argument) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&bank_account2_lock);
        pthread_mutex_lock(&bank_account1_lock);

        int tmp = andrews_bank_account1;
        andrews_bank_account1 = andrews_bank_account2;
        andrews_bank_account2 = tmp;

        pthread_mutex_unlock(&bank_account1_lock);
        pthread_mutex_unlock(&bank_account2_lock);

    }

    return NULL;
}

int main(void) {
    //create two threads performing almost the same task

    pthread_t thread_id1;
    pthread_create(&thread_id1, NULL, swap1, NULL);

    pthread_t thread_id2;
    pthread_create(&thread_id2, NULL, swap2, NULL);

    // threads will probably never finish
    // deadlock will likely likely occur
    // with one thread holding  bank_account1_lock
    // and waiting for bank_account2_lock to unlock
    // and the other  thread holding  bank_account2_lock
    // and waiting for bank_account1_lock to unlock,
    // could swap them to solve this but this is a simple example of dead lock


    pthread_join(thread_id1, NULL);
    pthread_join(thread_id2, NULL);

    return 0;
}
```