COMP1521 21T2 — Integers

https://www.cse.unsw.edu.au/~cs1521/21T2/

10 types of students

There are only 10 types of students ...

- those that understand binary
- those that don't understand binary

Decimal Representation

- Can interpret decimal number `4705` as:
  $4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$
- The *base* or *radix* is 10 ... digits `0` – `9`
- Place values:

...	1000	100	10	1
...	10^3	10^2	10^1	10^0

- Write number as 4705_{10}
  - Note use of subscript to denote base

Representation in Other Bases

- base 10 is an arbitrary choice
- can use any base
- e.g. could use base 7
- Place values:

...	343	49	7	1
...	7^3	7^2	7^1	7^0

- Write number as 1216_7 and interpret as:
  $1 \times 7^3 + 2 \times 7^2 + 1 \times 7^1 + 6 \times 7^0 == 454_{10}$

Binary Representation

- Modern computing uses binary numbers
  - because digital devices can easily produce high or low level voltages which can represent 1 or 0.
- The *base* or *radix* is 2
  Digits `0` and `1`
- Place values:

$\cdots$	8	4	2	1
$\cdots$	2^3	2^2	2^1	2^0

- Write number as 1011_2 and interpret as:
  $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 == 11_{10}$

Hexadecimal Representation

- Binary numbers hard for humans to read — too many digits!
- Conversion to decimal awkward and hides bit values
- Solution: write numbers in hexadecimal!
- The *base* or *radix* is 16 ... digits `0`, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Place values:

...	4096	256	16	1
...	16^3	16^2	16^1	16^0

- Write number as $3AF1_{16}$ and interpret as:
  $3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 == 15089_{10}$
- in C, **`0x`** prefix denotes hexadecimal, e.g. **`0x3AF1`**

Octal & Binary C constants

- Octal (based 8) representation used to be popular for binary numbers
- Similar advantages to hexadecimal
- in C a leading **`0`** denotes octal, e.g. **`07563`**
- 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); // might compile and print 42
```

Binary Constants

In hexadecimal, each digit represents 4 bits

```
    0100 1000 1111 1010 1011 1100 1001 0111
0x     4    8    F    A    B    C    9    7
```

In octal, each digit represents 3 bits

```
    01 001 000 111 110 101 011 110 010 010 111
0    1   1   0   7   6   5   3   6   2   2   7
```

In binary, each digit represents 1 bit

```
    0b01001000111110101011110010010111
```

Binary to Hexadecimal

- Example: Convert 1011111000101001_2 to Hex:

- Example: Convert 10111101011100_2 to Hex:

Hexadecimal to Binary

- Reverse the previous process ...
- Convert each hex digit into equivalent 4-bit binary representation
- Example: Convert $AD5_{16}$ to Binary:

Representing Negative Integers

- modern computers almost always use two's complement to represent integers
- positive integers and zero represented in obvious way
- negative integers represented in clever way to make arithmetic in silicon fast/simpler
- for an n-bit binary number 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

- Some simple code to examine all 8 bit twos complement bit patterns.

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

source code for 8_bit_twos_complement.c

```
$ dcc 8_bit_twos_complement.c print_bits.c -o 8_bit_twos_complement
```

source code for print_bits.c source code for print_bits.h

Code example: printing all 8 bit twos complement bit patterns

```
$ ./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
```

Code example: printing bits of int

```
int a = 0;
printf(”Enter an int: ”);
scanf(”%d”, &a);
// sizeof returns number of bytes, a byte has 8 bits
int n_bits = 8 * sizeof a;
print_bits(a, n_bits);
printf(”\n”);
```

source code for print_bits_of_int.c

```
$ dcc print_bits_of_int.c print_bits.c -o print_bits_of_int
$ ./print_bits_of_int
Enter an int: 42
00000000000000000000000000101010
$ ./print_bits_of_int
Enter an int: -42
11111111111111111111111111010110
```

Code example: printing bits of int

```
$ ./print_bits_of_int
Enter an int: 0
00000000000000000000000000000000
$ ./print_bits_of_int
Enter an int: 1
00000000000000000000000000000001
$ ./print_bits_of_int
Enter an int: -1
11111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: 2147483647
01111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: -2147483648
10000000000000000000000000000000
$
```

Bits in Bytes in Words

- Many hardware operations works with bytes: 1 byte == 8 bits
- C's **`sizeof`** gives you number of bytes used for variable or type
- **`sizeof`** ***variable*** - returns number of bytes to store ***variable***
- **`sizeof (`*`type`*`)`** - returns number of bytes to store ***type***
- On CSE servers, C types have these sizes
  - `char` = 1 byte = 8 bits, 42 is 00101010
  - `short` = 2 bytes = 16 bits, 42 is 0000000000101010
  - `int` = 4 bytes = 32 bits, 42 is 00000000000000000000000000101010
  - `double` = 8 bytes = 64 bits, 42 = ?
- above are common sizes but not universal on a small embedded CPU `sizeof (int)` might be 2 (bytes)

Code example: `integer_types.c` - exploring integer types

We can use **`sizeof`** and **`limits.h`** to explore the range of values which can be represented by standard C integer types **on our machine**...

```
$ dcc integer_types.c -o integer_types
$ ./integer_types
              Type Bytes Bits
              char     1    8
       signed char     1    8
     unsigned char     1    8
             short     2   16
    unsigned short     2   16
               int     4   32
      unsigned int     4   32
              long     8   64
     unsigned long     8   64
         long long     8   64
unsigned long long     8   64
```

```
              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           4294967295
              long   -9223372036854775808  9223372036854775807
     unsigned long                      0 18446744073709551615
         long long   -9223372036854775808  9223372036854775807
unsigned long long                      0 18446744073709551615
```

source code for integer_types.c

stdint.h - integer types with guaranteed sizes

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

- to get below integer types (and more) with guaranteed sizes
- we will use these heavily in COMP1521

```c
                // 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
```

source code for stdint.c

Code example: char_bug.c

Common C bug:

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

Typically `stdio.h` contains:

```
#define EOF -1
```

- most platforms: char is signed (-128..127)
  - loop will incorrectly exit for a byte containing 0xFF
- rare platforms: char is unsigned (0..255)
  - loop will never exit

source code for char_bug.c

COMP1521 21T2 — Bitwise Operators

https://www.cse.unsw.edu.au/~cs1521/21T2/

Recap of yesterday's lecture

Why do we care about data representation?

- Information = Data + Representation
  - Without the data, there's obviously no information at all
  - But without knowing the exact representation, who knows what the data could even mean?

Data ambiguity example

What does `0b10100011` mean?

- Does it mean -93? (signed 1-byte integer)
- Does it mean 163? (unsigned 1-byte integer)
- Does it mean something else?

What does `0b01110011_01110010_01101001_00000000` mean?

- Does it mean 1936877824? ([un]signed 4-byte int)
- Does it mean ~1.9205×10^{31}? (IEEE-754 single-precision floating point)
- … or could it mean "`sri`"? (null-terminated ascii string)

A common UNIX data representation

Consider file permissions in the Unix file system.

Each file has three sets of “flags” defining its permissions:

```
$ ls -l foo.c
-rwxrw-r--  1 sri  group  486 4 May 12:34 foo.c
```

In this example:

- `rwx` gives permissions for the owner of the file
- `rw-` gives permissions for group members
- `r--` gives permissions for everyone else

How can we represent this information *efficiently*?

A common UNIX data representation

We could use:

```
// 10 * 1 byte = 10 bytes
char permissions[10] = "rwxrw-r--";
```

Or possibly:

```
// 9 * 4 bytes = 36 bytes
int permissions[9] = {1, 1, 1,  1, 1, 0,  1, 0, 0};
```

Stop and think - can we make a more efficient representation?

A common UNIX data representation

Since each permission is only a `true` or `false` boolean value, we can take advantage of this and use only a single bit for each permission.

This allows us to represent the entire data in just 2 bytes!

```
//                              rwxrw-r--
unsigned short permissions = 0b111110100;
```

This is *much* more efficient, but how are we able to work with individual bits in C?

Bitwise Operators

Sometimes we want to work with individual bits inside a larger value.

Fortunately, everything in C really is just 1's and 0's under the hood!

- eg. the number 42 is `0b00101010`
- eg. the ascii character `'#'` is `0b00100011`
- eg. the floating point `3.14` is `0b01000000010010001111010111000011`

C provides special operators to read/write individual bits:

- & = bitwise AND
- | = bitwise OR
- ~ = bitwise NOT
- ^ = bitwise XOR
- « = left shift
- » = right shift

Bitwise AND: &

The **&** operator

- takes two values (1,2,4,8 bytes), treats as 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
```

Used for e.g. checking whether a bit is set

Checking for Odd Numbers

The obvious way to check for odd numbers in C

```
int isOdd(int n) {
    return n % 2 == 1;
}
```

We can use **&** to achieve the same thing:

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

Bitwise OR: `|`

The `|` operator

- takes two values (1,2,4,8 bytes), treats as 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
```

Used for e.g. ensuring that a bit is set

Bitwise NEG: ~

The ~ operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- performs logical negation of each bit
- result contains same number of bits as input

Example:

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

Used for e.g. creating useful bit patterns

Bitwise XOR: ^

The **^** operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical XOR 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
```

Used in e.g. generating hashes, graphic operation, cryptography

Left Shift: «

The « operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer *x*
- moves (shifts) each bit *x* positions to the left
- left-end bit vanishes; right-end bit replaced by zero
- result contains same number of bits as input

Example:

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

Right Shift: »

The **»** operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- also takes a small positive integer *x*
- moves (shifts) each bit *x* positions to the right
- right-end bit vanishes; left-end bit replaced by zero(*)
- result contains same number of bits as input

Example:

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

- shifts involving negative values are not portable (implementation defined)
- common source of bugs in COMP1521 and elsewhere
- always use unsigned values/variables to be safe/portable.

bitwise.c: showing results of bitwise operation

```
$ dcc bitwise.c print_bits.c -o bitwise
$ ./bitwise
Enter a: 23032
Enter b: 12345
Enter c: 3
     a = 0101100111111000 = 0x59f8 = 23032
     b = 0011000000111001 = 0x3039 = 12345
    ~a = 1010011000000111 = 0xa607 = 42503
 a & b = 0001000000111000 = 0x1038 = 4152
 a | b = 0111100111111001 = 0x79f9 = 31225
 a ^ b = 0110100111000001 = 0x69c1 = 27073
a >> c = 0000101100111111 = 0x0b3f = 2879
a << c = 1100111111000000 = 0xcfc0 = 53184
```

source code for bitwise.c
source code for print_bits.c source code for print_bits.h

bitwise.c: code

```
uint16_t a = 0;
printf(”Enter a: ”);
scanf(”%hd”, &a);
uint16_t b = 0;
printf(”Enter b: ”);
scanf(”%hd”, &b);
printf(”Enter c: ”);
int c = 0;
scanf(”%d”, &c);
print_bits_hex(”     a = ”, a);
print_bits_hex(”     b = ”, b);
print_bits_hex(”    ~a = ”, ~a);
print_bits_hex(” a & b = ”, a & b);
print_bits_hex(” a | b = ”, a | b);
print_bits_hex(” a ^ b = ”, a ^ b);
print_bits_hex(”a >> c = ”, a >> c);
print_bits_hex(”a << c = ”, a << c);
```

source code for bitwise.c

`shift_as_multiply.c`: using shift to multiply by 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
$
```

shift_as_multiply.c: using shift to multiply by 2^n

```c
int n = strtol(argv[1], NULL, 0);
uint32_t power_of_two;
int n_bits = 8 * sizeof power_of_two;
if (n >= n_bits) {
    fprintf(stderr, "n is too large\n");
    return 1;
}
power_of_two = 1;
power_of_two = power_of_two << n;
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");
```

source code for shift_as_multiply.c

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:
00000000000001111111111111111111
$ ./set_low_bits 29
The bottom 29 bits of 536870911 are ones:
00011111111111111111111111111111
```

set_low_bits.c: using « and − to set low *n* bits

```
int n = strtol(argv[1], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof mask;
assert(n >= 0 && n < n_bits);
mask = 1;
mask = mask << n;
mask = mask - 1;
printf("The bottom %d bits of %u are ones:\n", n, mask);
print_bits(mask, n_bits);
printf("\n");
```

source code for set_low_bits.c

set_bit_range.c: using « and − to set a range of 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
```

set_bit_range.c: using « and – to set a range of bits

```
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;
```

```
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");
```

source code for set_bit_range.c

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
```

extract_bit_range.c: extracting a range of bits

```c
int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
// get a value with the bits outside the range low_bit..high_bit set to zero
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");
```

source code for extract_bit_range.c

print_bits.c: extracting the n-th bit of a value

```c
void print_bits(uint64_t value, int how_many_bits) {
    // print bits from most significant to least significant
    for (int i = how_many_bits - 1; i >= 0; i--) {
        int bit = get_nth_bit(value, i);
        printf("%d", bit);
    }
}
```

```c
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;
}
```

source code for print_bits.c

print_int_in_hex.c: print an integer in hexadecimal

- write C to print an integer in hexadecimal instead of using:

```
printf("%x", n)
```

```
$ 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
$
```

source code for print_int_in_hex.c

print_int_in_hex.c: main

```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;
}
```

source code for print_int_in_hex.c

print_int_in_hex.c: print_hex - extracting digit

```
// sizeof returns 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 * (n_hex_digits - which_digit - 1);
    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')
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    putchar(hex_digit_ascii);
}
```

source code for print_int_in_hex.c

int_to_hex_string.c: convert int to a string of hex digits

- Write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function `snprintf` to do this.

```
$ 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
$
```

source code for int_to_hex_string.c

int_to_hex_string.c: main

```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;
}
```

source code for int_to_hex_string.c

int_to_hex_string.c: convert int to a string of hex digits

```c
// sizeof returns 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 * (n_hex_digits - which_digit - 1);
    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;
```

source code for int_to_hex_string.c

hex_string_to_int.c: convert hex digit string to int

- As an exercise write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function `strtol` to do this.

```
$ 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
$
```

source code for hex_string_to_int.c

hex_string_to_int.c: main

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

source code for hex_string_to_int.c

hex_string_to_int.c: convert array of hex digits to int

```
uint32_t hex_string_to_int(char *hex_string) {
    uint32_t value = 0;
    for (int which_digit = 0; hex_string[which_digit] != 0; which_digit++) {
        int ascii_hex_digit = hex_string[which_digit];
        int digit_as_int = hex_digit_to_int(ascii_hex_digit);
        value = value << 4;
        value = value | digit_as_int;
    }
    return value;
}
```

source code for hex_string_to_int.c

hex_string_to_int.c: convert single hex digit to int

```c
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 integer
        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');
    }
    fprintf(stderr, "Bad digit '%c'\n", ascii_digit);
    exit(1);
}
```

source code for hex_string_to_int.c

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
```

source code for shift_bug.c

xor.c: fun with xor

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

source code for xor.c

xor.c: fun with xor

```
$ 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
$
```

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

```
#define FIRE_TYPE       0x0001
#define FIGHTING_TYPE   0x0002
#define WATER_TYPE      0x0004
#define FLYING_TYPE     0x0008
#define POISON_TYPE     0x0010
#define ELECTRIC_TYPE   0x0020
#define GROUND_TYPE     0x0040
#define PSYCHIC_TYPE    0x0080
#define ROCK_TYPE       0x0100
#define ICE_TYPE        0x0200
#define BUG_TYPE        0x0400
#define DRAGON_TYPE     0x0800
#define GHOST_TYPE      0x1000
#define DARK_TYPE       0x2000
#define STEEL_TYPE      0x4000
#define FAIRY_TYPE      0x8000
```

source code for pokemon.c

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

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

```c
uint16_t our_pokemon = BUG_TYPE | POISON_TYPE | FAIRY_TYPE;
```

```c
// 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
}
```

source code for pokemon.c

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

```
// 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
}
```

source code for pokemon.c

```
$ 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 = 0x100010116
b = 0000000000000000000000000000001000000000000000000000000000111000 = 0x200000038
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
```

bitset.c: main

```
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));
```

source code for bitset.c

`bitset.c`: common set operations

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

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

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

```c
set set_member(int x, set a) {
    assert(x >= 0 && x < MAX_SET_MEMBER);
    return a & ((set)1 << x);
}
```

bitset.c: counting set members

```c
int set_cardinality(set a) {
    int n_members = 0;
    while (a != 0) {
        n_members += a & 1;
        a >>= 1;
    }
    return n_members;
}
```

bitset.c: set input

```
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;
}
```

bitset.c: set output

```
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");
}
```

Exercise: Bitwise Operations

Given the following variable declarations:

```
// a signed 8-bit value
unsigned char x = 0x55;
unsigned char y = 0xAA;
```

What is the value of each of the following expressions:

- `(x & y)  (x ^ y)`
- `(x « 1)  (y « 1)`
- `(x » 1)  (y » 1)`

Exercise: Bit-manipulation

Assuming 8-bit quantities and writing answers as 8-bit bit-strings:

What are the values of the following:

- `25, 65, ~0, ~~1, 0xFF, ~0xFF`
- `(01010101 & 10101010), (01010101 | 10101010)`
- `(x & ~x), (x | ~x)`

How can we achieve each of the following:

- ensure that the 3rd bit from the RHS is set to 1
- ensure that the 3rd bit from the RHS is set to 0

COMP1521 21T2 — Floating-Point Numbers

https://www.cse.unsw.edu.au/~cs1521/21T2/

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 maybe only 80 bits used)
- Floating point constants, e.g : **`3.14159`** **`1.0e-9`** are **`double`**
- Reminder: division of 2 ints in C yields an int.
  - but division of double and int in C yields a double.

Floating Point Number - Output

```
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 5
printf("%10.1lf\n", d);  // prints        0.6
```

source code for float_output.c

Floating Point Numbers

- if we represent floating point numbers with a fixed small number of bits
  - there are only a finite number of bit patterns
  - can only represent a finite subset of reals
- almost all real values will have no exact representation
- value of arithmetic operations may be real with no exactly representation
- we must use closest value which can be exactly represented
- this approximation introduces an error into our calculations
- often, does not matter
- sometimes ... can be disasterous

Fixed Point Representation

- can have fractional numbers in other bases, e.g.: $110.101_2 == 6.625_{10}$
- could represent fractional numbers similarly to integers by assuming decimal point is in ***fixed*** position
- for example with 32 bits:
  - 16 bits could be used for integer part
  - 16 bits could be used for the fraction
  - equivalent to storing values as integers after multiplying (***scaling***) by 2^{16}
  - major limitation is only small range of values can be represented
    - minimum $2_{-16} \approx 0.000015$
    - maximum $2_{15} \approx 32768$
- usable for some problems, but not ideal
- used on small embedded processors without silicon floating point

floating_types.c - print characteristics of floating point types

```
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_MA
```

source code for floating_types.c

```
$ ./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
```

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*
- numbers have form $sign\ fraction \times 2^{exponent}$, where $sign$ is +/-
- ***fraction*** always has 1 digit before decimal point (***normalized***)
  - as a consequence only 1 representation for any value
- ***exponent*** is stored as positive number by adding constant value (***bias***)
- numbers close to zero have higher precision (more accurate)

Floating Point Numbers

Example of normalising the fraction part in binary:

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

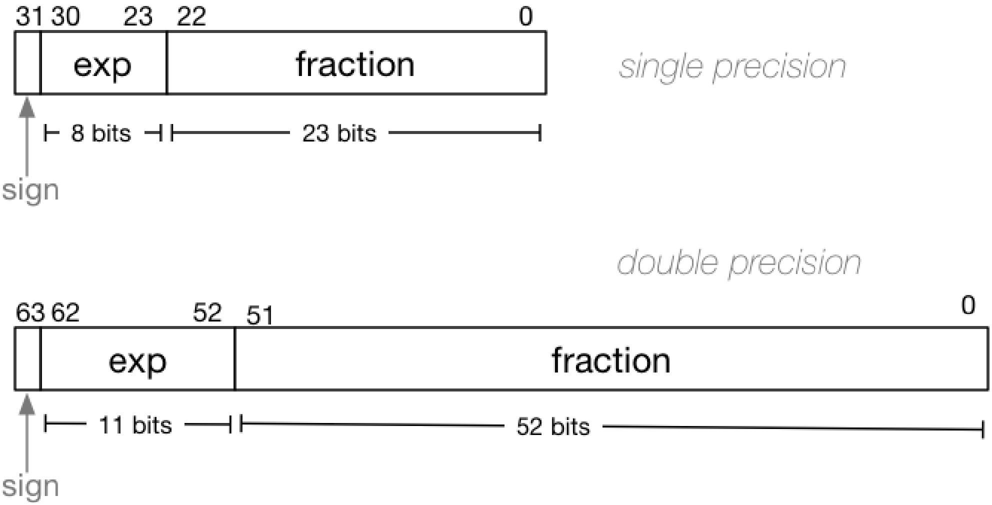
The normalised fraction part always has 1 before the decimal point.

Example of determining the exponent in binary:

- if exponent is 8-bits, then the bias = $2^{8-1} - 1$ = 127
- valid bit patterns for exponent `00000001` .. `11111110`
- correspond to B exponent values -126 .. 127

Floating Point Numbers

Internal structure of floating point values



IEEE-754 Single Precision example: **0.15625**

```
0.15625 is represented in IEEE-754 single-precision by these bits:
00111110001000000000000000000000
sign | exponent | fraction
   0 | 01111100 | 01000000000000000000000
sign bit = 0
sign = +
raw exponent    = 01111100 binary
                = 124 decimal
actual exponent = 124 - exponent_bias
                = 124 - 127
                = -3
number = +1.01000000000000000000000 binary * 2**-3
       = 1.25 decimal * 2**-3
       = 1.25 * 0.125
       = 0.15625
```

source code for explain_float_representation.c

IEEE-754 Single Precision example: **-0.125**

```
$ ./explain_float_representation -0.125
-0.125 is represented as a float (IEEE-754 single-precision) by these bits:
10111110000000000000000000000000
sign | exponent | fraction
   1 | 01111100 | 00000000000000000000000
sign bit = 1
sign = -
raw exponent    = 01111100 binary
                = 124 decimal
actual exponent = 124 - exponent_bias
                = 124 - 127
                = -3
number = -1.00000000000000000000000 binary * 2**-3
       = -1 decimal * 2**-3
       = -1 * 0.125
       = -0.125
```

IEEE-754 Single Precision example: **150.75**

```
$ ./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 = +
raw exponent    = 10000110 binary
                = 134 decimal
actual exponent = 134 - exponent_bias
                = 134 - 127
                = 7
number = +1.00101101100000000000000 binary * 2**7
       = 1.17773 decimal * 2**7
       = 1.17773 * 128
       = 150.75
```

IEEE-754 Single Precision example: **-96.125**

```
$ ./explain_float_representation -96.125
-96.125 is represented in IEEE-754 single-precision by these bits:
11000010110000000100000000000000
sign | exponent | fraction
   1 | 10000101 | 10000000100000000000000
sign bit = 1
sign = -
raw exponent    = 10000101 binary
                = 133 decimal
actual exponent = 133 - exponent_bias
                = 133 - 127
                = 6
number = -1.10000000100000000000000 binary * 2**6
       = -1.50195 decimal * 2**6
       = -1.50195 * 64
       = -96.125
```

IEEE-754 Single Precision exploring bit patterns #1

```
$ ./explain_float_representation 00111101110011001100110011001101
sign bit = 0
sign = +
raw exponent    = 01111011 binary
                = 123 decimal
actual exponent = 123 - exponent_bias
                = 123 - 127
                = -4
number = +1.10011001100110011001101 binary * 2**-4
       = 1.6 decimal * 2**-4
       = 1.6 * 0.0625
       = 0.1
```

`infinity.c`: exploring infinity

- IEEE 754 has a representation for +/- infinity
- propagates sensibly through calculations

```
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)
```

source code for infinity.c

nan.c: handling errors robustly

- C (IEEE-754) has a representation for invalid results:
  - NaN (not a number)
- ensures errors propagates sensibly through calculations

```c
double x = 0.0/0.0;
printf("%lf\n", x); //prints nan
printf("%lf\n", x - 1); // prints nan
printf("%d\n", x == x); // prints 0 (false)
printf("%d\n", isnan(x)); // prints 1 (true)
```

source code for nan.c

IEEE-754 Single Precision example: **inf**

```
$ ./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
```

IEEE-754 Single Precision exploring bit patterns #2

```
$ ./explain_float_representation 01111111110000000000000000000000
sign bit = 0
sign = +
raw exponent    = 11111111 binary
                = 255 decimal
number = NaN
```

source code for explain_float_representation.c

Consequences of most reals not having exact representations

```c
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
    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
```

source code for double_imprecision.c

- do not use **==** and **!=** with floating point values
- instead check if values are close

Consequences of most reals not having exact representations

```c
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
```

source code for double_catastrophe.c

Another reason not to use == with floating point values

```
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");
}
```

source code for double_not_always.c

Another reason not to use == with floating point values

```
$ 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
```

because closest possible representation for d + 1 is also closest possible representation for d

source code for double_not_always.c

Consequences of most reals not having exact representations

```c
double d = 9007199254740992;
// loop never terminates
while (d < 9007199254740999) {
    printf("%lf\n", d); // always prints 9007199254740992.000000
    // 9007199254740993 can not be represented as a double
    // closest double is 9007199254740992.0
    // so 9007199254740992.0 + 1 = 9007199254740992.0
    d = d + 1;
}
```

source code for double_disaster.c

- 9007199254740993 is $2^{53} + 1$
  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

Exercise: Floating point → Decimal

Convert the following floating point numbers to decimal.

Assume that they are in IEEE 754 single-precision format.

```
0 10000000 11000000000000000000000

1 01111110 10000000000000000000000
```

COMP1521 21T2 — MIPS Basics

https://www.cse.unsw.edu.au/~cs1521/21T2/

Why Study Assembler?

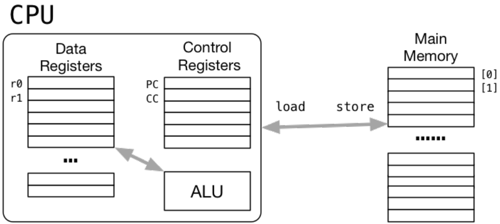
Useful to know assembly language because ...

- sometimes you are *required* to use it:
  - e.g., low-level system operations, device drivers
- improves your understanding of how compiled programs execute
  - very helpful when debugging
  - understand performance issues better
- performance tweaking ... squeezing out last pico-second
  - re-write that performance critical code in assembler!

CPU Components

A typical modern CPU has

- a set of *data* registers
- a set of *control* registers (including 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

CPU Architecture Families Used in Game Consoles

Year	Console	Architecture	Chip	MHz
1995	PS1	MIPS	R3000A	34
1996	N64	MIPS	R4300	93
2000	PS2	MIPS	Emotion Engine	300
2001	xbox	x86	Celeron	733
2001	GameCube	Power	PPC750	486
2006	xbox360	Power	Xenon (3 cores)	3200
2006	PS3	Power	Cell BE (9 cores)	3200
2006	Wii	Power	PPC Broadway	730
2013	PS4	x86	AMD Jaguar (8 cores)	1800
2013	xbone	x86	AMD Jaguar (8 cores)	2000
2017	Switch	ARM	NVidia TX1	1000
2020	PS5	x86	AMD Zen 2 (8 cores)	3500
2020	xboxs	x86	AMD Zen 2 (8 cores)	3700

Fetch-Execute Cycle

- typical CPU program execution pseudo-code:

```
word pc = START_ADDRESS;
while (1) {
    word instruction = memory[pc];
    pc++; // move to next instr
    if (instruction == HALT)
        break;
    else
        execute(instruction);
}
```

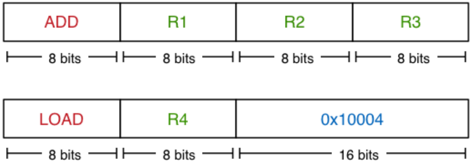
- **pc** = Program Counter, a special CPU register which tracks execution
    - note some instructions modify **pc** (branches and jumps)

Fetch-Execute Cycle

Executing an instruction involves:

- determine what the *operator* is
- determine if/which *register(s)* are involved
- determine if/which *memory location* is involved
- carry out the operation with the relevant operands
- store result, if any, in appropriate register

Example instruction encodings
(not from a real machine):



MIPS Architecture

MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to PlayStations, ...
- still popular in some embedded fields: e.g., modems/routers, TVs
- but being out-competed by ARM and, more recently, RISC-V

We consider the MIPS32 version of the MIPS family, running on SPIM

- `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

Executables and source: *http://spimsimulator.sourceforge.net/*

Source code for browsing under */home/cs1521/spim*

MIPS Instructions

MIPS has several classes of instructions:

- *load and store* ... transfer data between registers and memory
- *computational* ... perform arithmetic/logical operations
- *jump and branch* ... transfer control of program execution
- *coprocessor* ... standard interface to various co-processors
- *special* ... miscellaneous tasks (e.g. syscall)

And several *addressing modes* for each instruction:

- between memory and register — **direct**, **indirect**
- constant to register — **immediate**
- register + register + destination register

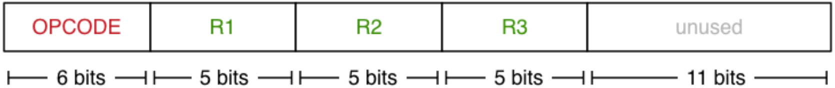
MIPS Instructions

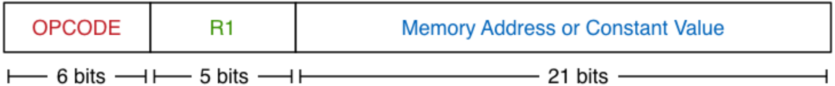
Instructions are simply bit patterns.

MIPS instructions are 32-bits long, and specify ...

- an **operation** (e.g. load, store, add, branch, ...)
- one or more **operands** (e.g. registers, memory addresses, constants)

Some possible instruction formats





Assembly Language

Instructions are simply bit patterns — on MIPS, 32 bits long.

- Could write machine code program just by specifying bit-patterns
  e.g as a sequence of hex digits:

  ```
  0x3c041001  0x34020004  0x0000000c  0x03e00008
  ```

  - unreadable! difficult to maintain!
- adding/removing instructions changes bit pattern for other instructions
- changing variable layout in memory changes bit pattern for instructions

Solution: **assembly language**, a symbolic way of specifying machine code

- write instructions using names rather than bit-strings
- refer to registers using either numbers or names
- allow names (labels) associated with memory addresses

Examples MIPS Assembler

```
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 across 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 Architecture: Registers

MIPS CPU has

- 32 general purpose registers (32-bit)
- 16/32 floating-point registers (for float/double)
- *PC* ... 32-bit register (always aligned on 4-byte boundary)
- *Hi*, *Lo* ... for storing results of multiplication and division

Registers can be referred to as `$0`...`$31`, or by symbolic names

Some registers have special uses; e.g.,

- register `$0` always has value 0, discards all written values
- registers `$1`, `$26`, `$27` reserved for use by system

More details on following slides ...

MIPS Architecture: 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)

MIPS Architecture: Integer Registers ... Usage Convention

- Except for registers 0 and 31, these uses are *only* programmers conventions
  - no difference between registers 1..30 in the silicon
- *Conventions* allow compiled code from different sources to be combined (linked).
- Some of these conventions are irrelevant when writing tiny assembly programs ... follow them anyway
- for general use, keep to registers $t0..$t9, $s0..$t7
- use other registers only for conventional purpose
  - e.g. only use $a0..$a3 for arguments
- *never* use registers 1, 26, 27 ($at, $k0, $k1)

MIPS Architecture: Floating-Point Registers

Reg	Notes
`$f0..$f2`	hold return value of functions which return floating-point results
`$f4..$f10`	temporary registers; not preserved across function calls
`$f12..$f14`	used for first two double-precision function arguments
`$f16..$f18`	temporary registers; used for expression evaluation
`$f20..$f30`	saved registers; value is preserved across function calls

Floating-point registers come in pairs:

- either use all 32 as 32-bit registers,
- or use only even-numbered registers for 16 64-bit registers

COMP1521 will not explore floating point on the MIPS

Data and Addresses

All operations refer to data, either

- in a register
- in memory
- a constant which is embedded in the instruction itself

Computation operations refer to registers or constants.

Only load/store instructions refer to memory.

To access registers, you can also use `$`*name*

e.g. `$zero` == `$0`, `$t0` == `$8`, `$fp` == `$30`, ...

The syntax for constant value is C-like:

```
1  3  -1  -2  12345  0x1  0xFFFFFFFF
"a string"  'a'  'b'  '1'  '\n'  '\0'
```

Describing MIPS Assembly Operations

Registers are denoted:

R_d	destination register	where result goes
R_s	source register #1	where data comes from
R_t	source register #2	where data comes from

For example:

$$\text{add}\quad \$R_d, \$R_s, \$R_t \qquad \Longrightarrow \qquad R_d := R_s + R_t$$

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.
- see also: **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 instructions(s).

Extra Arithmetic Instructions

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	`0000000000000000ddddd00000001010`
mfhi r_d	r_d = hi	`0000000000000000ddddd00000001001`

- **mult** provides multiply with 64-bit result
- little use of these instructions in COMP1521 except challenge exercises
- 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	$r_d = r_s \mid r_t$	`000000ssssstttttddddd00000100101`
xor r_d, r_s, r_t	$r_d = r_s$ ^ r_t	`000000ssssstttttddddd00000100110`
nor r_d, r_s, r_t	$r_d = \sim(r_s \mid r_t)$	`000000ssssstttttddddd00000100111`
andi r_t, r_s, I	$r_t = r_s$ & I	`001100ssssstttttIIIIIIIIIIIIIIII`
ori r_t, r_s, I	$r_t = r_s \mid$ I	`001101ssssstttttIIIIIIIIIIIIIIII`
xori r_t, r_s, I	$r_t = r_s$ ^ I	`001110ssssstttttIIIIIIIIIIIIIIII`
not r_d, r_s	$r_d = \sim r_s$	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	$r_d = r_t \ll r_s$	`000000ssssstttttddddd00000000100`
srlv r_d, r_t, r_s	$r_d = r_t \gg r_s$	`000000ssssstttttddddd00000000110`
srav r_d, r_t, r_s	$r_d = r_t \gg r_s$	`000000ssssstttttddddd00000000111`
sll r_d, r_t, I	$r_d = r_t \ll$ I	`00000000000tttttdddddIIIII000000`
srl r_d, r_t, I	$r_d = r_t \gg$ I	`00000000000tttttdddddIIIII000010`
sra r_d, r_t, I	$r_d = r_t \gg$ I	`00000000000tttttdddddIIIII000011`

- **srl** and **srlv** shift zeros into most-significant bit
  - this matches shift in C of **unsigned** value
- **sra** and **srav** propagate most-significant bit
  - this ensure shifting a negative number divides by 2
- spim provides **rol** and **ror** pseudo-instructions which rotate bits
  - real instructions on some MIPS versions
  - 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

```
# 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

Pseudo-Instructions

```
move $a1, $v0

li   $t5, 42

li   $s1, 0xdeadbeef


la   $t3, label
```

Real Instructions

```
addu $a1, $0, $v0

ori  $t5, $0, 42

lui  $at, 0xdead
ori  $s1, $at, 0xbeef

lui  $at, label[31..16]
ori  $t3, $at, label[15..0]
```

MIPS vs SPIM

MIPS is a machine architecture, including instruction set

SPIM is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides (primitive) debugging capabilities
    - single-step, breakpoints, view registers/memory, ...
- provides mechanism to interact with operating system (syscall)

Also provides extra instructions, mapped to MIPS core set:

- provide convenient/mnemonic ways to do common operations
- e.g. `move $s0, $v0` rather than `addu $s0, $v0, $0`

Using SPIM

Three ways to execute MIPS code with SPIM...

`spim` ... command line tool

- load programs using `-file` option
- interact using stdin/stdout via terminal

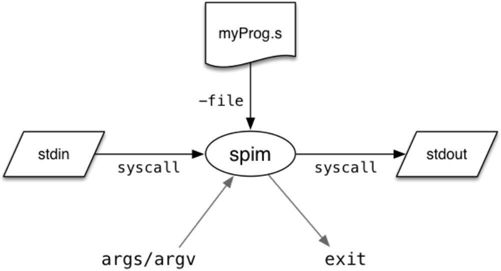
`qtspim` ... GUI environment

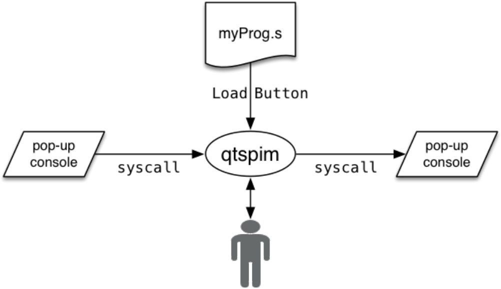
- load programs via a load button
- interact via a pop-up stdin/stdout terminal

`xspim` ... GUI environment

- similar to `qtspim`, but not as pretty
- requires X Windows

Using SPIM


myProg.s
-file
stdin
syscall
spim
syscall
stdout
args/argv
exit


myProg.s
Load Button
pop-up console
syscall
qtspim
syscall
pop-up console

Using SPIM Interactively

```
$ 1521 spim
(spim) load "myprogram.s"
(spim) step 6
[0x00400000] 0x8fa40000  lw $4, 0($29)
[0x00400004] 0x27a50004  addiu $5, $29, 4
[0x00400008] 0x24a60004  addiu $6, $5, 4
[0x0040000c] 0x00041080  sll $2, $4, 2
[0x00400010] 0x00c23021  addu $6, $6, $2
[0x00400014] 0x0c100009  jal 0x00400024 [main]
(spim) print_all_regs hex
....
                 General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000 ...
R1  (at) = 10010000  R9  (t1) = 00000000  R17 (s1) = 00000000 ...
```

System Calls

Our programs can't really do anything … we usually rely on system services to do things for us.
`syscall` lets us make *system calls* for these services.

SPIM provides a set of system calls for I/O and memory allocation.
$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	`$a0`: line, `$a1`: length	
`exit(0)`	10	status in `$a0`	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

System Calls ... Little 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`	

System calls 13...16 support file I/O: open, read, write, close.

Encoding MIPS Instructions as 32 bit Numbers

Assembler	Encoding
`add $a3, $t0, $zero`	
`add $d, $s, $t`	`000000 sssss ttttt ddddd00000100000`
`add $7, $8, $0`	`000000 00111 01000 00000000000100000`
	`0x01e80020` (decimal 31981600)
`sub $a1, $at, $v1`	
`sub $d, $s, $t`	`000000 sssss ttttt ddddd00000100010`
`sub $5, $1, $3`	`000000 00001 00011 00101000000100010`
	`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)

all instructions are variants of a small number of bit patterns

... register numbers always in same place

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
- instruction sequences that live in the code/text region

Each instruction or directive appears on its own line.

Our First MIPS program

C

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

MIPS

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

source code for i_love_mips.s source code for i_love_mips.c

COMP1521 21T2 — MIPS Control

https://www.cse.unsw.edu.au/~cs1521/21T2/

Jump Instructions

assembler	meaning	bit pattern
`j` *label*	pc = pc & 0xF0000000 | (X«2)	`000010XXXXXXXXXXXXXXXXXXXXXXXXXX`
`jal` *label*	r_{31} = pc + 4; pc = pc & 0xF0000000 | (X«2)	`000011XXXXXXXXXXXXXXXXXXXXXXXXXX`
`jr` r_s	pc = r_s	`000000sssss000000000000000001000`
`jalr` r_s	r_{31} = pc + 4; pc = r_s	`000000sssss000000000000000001001`

- jump instruction **unconditionally** transfer execution to a new location
- `spim` will calculate correct value for *X* from location of *label* in code
- **`jal`** & **`jalr`** set r_{31} (`$ra`) to address of the next instruction
  - used for function calls
  - return can then be implemented with `jr $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	`000100ssssstttttIIIIIIIIIIIIIIII`
bne r_s,r_t,*label*	if (r_s != r_t) pc += I«2	`000101ssssstttttIIIIIIIIIIIIIIII`
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 instruction **conditionally** transfer execution to a new location
- `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

Example Translation of Branch Pseudo-instructions

Pseudo-Instructions

```
bge  $t1, $t2, label


blt  $t1, $t2, label
```

Real Instructions

```
slt  $at, $t1, $t2
beq  $at, $0, label

slt  $at, $t1, $t2
bne  $at, $0, label
```

goto in C

The **goto** statement allows transfer of control to any labelled point with a function. 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:
```

goto in C

- **goto** statements can result in very difficult to read programs.
- **goto** statements can also result in slower programs.
- In general, use of **goto** is considered **bad** programming style.
- Do not use **goto** without very good reason.
- kernel & embedded programmers sometimes use goto.

MIPS Programming

Writing correct assembler directly is hard.

Recommended strategy:

- develop a solution in C
- map down to “simplified” C
- translate simplified C statements to MIPS instructions

Simplified C

- does *not* have `while`, compound `if`, complex expressions
- *does* have simple `if`, `goto`, one-operator expressions

Simplified C makes extensive use of

- *labels* ... symbolic name for C statement
- *goto* ... transfer control to labelled statement

Mapping C into **MIPS**

Things to do:

- allocate variables to registers/memory
- place literals in data segment
- transform C program to:
  - break expression evaluation into steps
  - replace most control structures by `goto`

Adding Two Numbers — C to Simple C

C

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

source code for add.c

Simplified C

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

source code for add.simple.c

Adding Two Numbers — Simple C to MIPS

Simplified C

```c
int x, y, z;
x = 17;
y = 25;
z = x + y;
printf("%d", z);
printf("\n");
```

MIPS

```
# add 17 and 25  and print result
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
    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
```

source code for add.s

Loops — `while` from C to Simplified C

Standard C

```
i = 0;
n = 0;
while (i < 5) {

   n = n + i;
   i++;
}
```

Simplified C

```
    i = 0;
    n = 0;
loop:
    if (i >= 5) goto end;
    n = n + i;
    i++;
    goto loop;
end:
```

Loops — `while` from Simplified C to MIPS

Simplified C

```
    i = 0;
    n = 0;
loop:
    if (i >= 5) goto end;
    n = n + i;
    i++;
    goto loop;
end:
```

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:
```

Conditionals — `if` from C to Simplified C

Standard C

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

} else {
    n = n + i;
}
```

Simplified C

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

note: `else` is not a valid label name in C

Conditionals — if from Simplified C to MIPS

Simplified C

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

MIPS

```
    # assuming i in $t0,
    # assuming n in $t1...

    bge $t0, 0, else1
    sub $t1, $t1, $t0
    goto end1
else1:
    add $t1, $t1, $t0
end1:
```

Conditionals — if and &&: from C to Simplified C

Standard C

```
if (i < 0 && n >= 42) {

    n = n - i;

} else {
    n = n + i;
}
```

Simplified C

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

Conditionals — `if` and `&&`: from Simplified C to MIPS

Simplified C

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

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: from C to simplified C

Standard C

```
if (i < 0 || n >= 42) {




    n = n - i;

} else {
    n = n + i;
}
```

Simplified C

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

Printing First 10 Integers: C to simplified C

C

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

source code for print10.c

Simplified C

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

source code for print10.simple.c

Printing First 10 Integers: MIPS

```
# print integers 1..10 one per line
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
```

source code for print10.s

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;
}
```

source code for odd_even.c

Simplified 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;
}
```

source code for odd_even.simple.c

Odd or Even: MIPS

```
# read a number and print whether its odd or even
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
```

source code for odd_even.s

Odd or Even: MIPS

```
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"
```

source code for odd_even.s

Sum 100 Squares: C to simplified C

C

```
int main(void) {
    int sum = 0;
    for (int i = 0; i <= 100; i++) {
        sum += i * i;
    }
    printf("%d\n", sum);
    return 0;
}
```

source code for sum_100_squares.c

Simplified 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;
}
```

source code for sum_100_squares.simple.c

Sum 100 Squares: MIPS

```
# calculate 1*1 + 2*2 + ... + 99 * 99 + 100 * 100
# sum in $t0, i in $t1, square in $t2
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:
```

source code for sum_100_squares.s

Sum 100 Squares: MIPS

```
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
```

source code for sum_100_squares.s

COMP1521 21T2 — MIPS Data

https://www.cse.unsw.edu.au/~cs1521/21T2/

The Memory Subsystem

- memory subsystem typically provides capability to load or store **bytes**
- each byte has unique **address**, think of:
  - memory as implementing a gigantic array of bytes
  - and the address is the array index
- on the MIPS32 machine, all addresses are 32-bit
- most general purpose computers now use 64-bit addresses (and there are 64-bit MIPS)
- typically, a small (1,2,4,8,...) group of bytes can be loaded/stored in single operations
- general purpose computers typically have complex *cache systems* to improve memory performance (not covered in this course)
- operating systems on general purpose computers typically provide **virtual memory** (covered later in this course)

Accessing Memory on the MIPS

- addresses are 32 bit (but there are 64-bit MIPS CPUs)
- only load/store instructions access memory on the MIPS
- 1 byte (8-bit) loaded/stored with **`lb`**/**`sb`**
- 2 bytes (16-bit) called a **half-word**, loaded/stored with **`lh`**/**`sh`**
- 4 bytes (32-bits) called a **word**, loaded/stored with **`lw`**/**`sw`**
- memory address used for load/store instructions is sum of a specified register and a 16-bit constant (often 0) which is part of the instruction
- for **`sb`** & **`sh`** operations low (least significant) bits of source register are used.
- **`lb`**/**`lh`** assume byte/halfword contains a 8-bit/16-bit **signed** integer
  - high 24/16-bits of destination register set to 1 if 8-bit/16-bit integer negative
- unsigned equivalents **`lbu`** & **`lhu`** assume integer is **unsigned**
  - high 24/16-bits of destination register always 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] | **mem**[r_s+I+1] « 8	100001ssssstttttIIIIIIIIIIIIIIII
lw r_t, I(r_s)	r_t = **mem**[r_s+I] | **mem**[r_s+I+1] « 8 | **mem**[r_s+I+2] « 16 | **mem**[r_s+I+3] « 24	100011ssssstttttIIIIIIIIIIIIIIII
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 **mem**[r_s+I+1] = r_t » 8 & 0xff	101001ssssstttttIIIIIIIIIIIIIIII
sw r_t, I(r_s)	**mem**[r_s+I] = r_t & 0xff **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	101011ssssstttttIIIIIIIIIIIIIIII

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
    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
```

source code for load_store_no_label.s

Assembler Directives

SPIM has directive to initialise memory, and to associate labels with addresses.

```
     .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 addr
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'};
```

Code example: storing and loading a value

```
# simple example of load & storing a byte
main:
    li   $t0, 42
    la   $t1, x
    sb   $t0, 0($t1)  # store 42 in byte at address labelled x
    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
```

source code for load_store.s

Testing Endian-ness

C

```c
uint8_t b;
uint32_t u;
u = 0x03040506;
// load first byte of u
b = *(uint8_t *)&u;
// prints 6 if little-endian
// and 3 if big-endian
printf("%d\n", b);
```

source code for endian.c

MIPS

```
    li   $t0, 0x03040506
    la   $t1, u
    sw   $t0, 0($t1) # u = 0x03040506;
    lb   $a0, 0($t1) # b = *(uint8_t *)&u
    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
```

source code for endian.s

Setting A Register to An Address

- Note 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 is run, so this differs only syntatctically from the **`li`** instruction.
- 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 **`lw`** etc

Specifying Addresses: Some SPIM short-cuts

- SPIM allows the constant which is part of load & store instructions can be omitted in the common case it is 0.

```
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 effectively pseudo-instructions.
- You can use these short cuts but won't help you much
- Most assemblers have similar short cuts for convenience

SPIM Memory Layout

Region	Address	Notes
`.text`	**`0x00400000`**..	instructions only; read-only; cannot expand
`.data`	**`0x10000000`**..	data objects; read/write; can be expanded
`.stack`	..**`0x7fffffef`**	this address and below; read/write
`.ktext`	**`0x80000000`**..	kernel code; read-only; only accessible in kernel mode
`.kdata`	**`0x90000000`**..	kernel data; only accessible in kernel mode

Global/Static Variables

Global and static variables need an appropriate number of bytes allocated in `.data` segment, using **`.space`**:

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

Initialised to 0 by default ... other directives allow initialisation to other values:

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

add: local variables in registers

C

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

MIPS

```
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li    $t0, 17
    li    $t1, 25
    add   $t2, $t1, $t0

    // ...
```

add variables in memory (uninitialized)

C

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

MIPS (.data)

```
.data
x:  .space 4
y:  .space 4
z:  .space 4
```

MIPS (.text)

```
main:
    li   $t0, 17         # x = 17;
    la   $t1, x
    sw   $t0, 0($t1)
    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)
```

source code for add_memory.s

add variables in memory (initialized)

C

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

MIPS .data

```
.data
x:   .word 17
y:   .word 25
z:   .space 4
```

MIPS .text

```
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
```

source code for add_memory_initialized.s

add variables in memory (array)

C

```c
int x[] = {17,25,0};
int main(void) {
    x[2] = x[0] + x[1];
}
```

MIPS .data

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

MIPS .text

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

source code for add_memory_array.s

store value in array element — example 1

C

```c
int x[10];

int main(void) {
    // sizeof x[0] == 4
    x[3] = 17;
}
```

MIPS

```
main:
    li   $t0, 3

    # each array element is 4 bytes
    mul  $t0, $t0, 4
    la   $t1, x
    add  $t2, $t1, $t0
    li   $t3, 17
    sw   $t3, 0($t2)
.data
x:  .space 40
```

store value in array element - example 2

C

```c
#include <stdint.h>

int16_t x[30];

int main(void) {
    // sizeof x[0] == 2
    x[13] = 23;
}
```

MIPS

```
main:
    li   $t0, 13

    # each array element is 2 bytes
    mul  $t0, $t0, 2
    la   $t1, x
    add  $t2, $t1, $t0
    li   $t3, 23
    sh   $t3, 0($t2)
.data
x:  .space 60
```

Printing Array: C to simplified C

C

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

source code for print5.c

Simplified C

```c
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;
}
```

source code for print5.simple.c

Printing Array: MIPS

```
# print array of ints
# i in $t0
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:
```

source code for print5.s

Printing Array: MIPS (continued)

```
end:
    li   $v0, 0          # return 0
    jr   $ra
.data
numbers:                 # int numbers[10] = { 3, 9, 27, 81, 243};
     .word 3, 9, 27, 81, 243
```

source code for print5.s

Printing Array with Pointers: C to simplified C

C

```c
int main(void) {
    int *p = &numbers[0];
    int *q = &numbers[4];
    while (p <= q) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

source code for pointer5.c

Simplified C

```c
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;
}
```

source code for pointer5.simple.c

Printing Array with Pointers: MIPS

```
# p in $t0, q in $t1
main:
    la   $t0, numbers    # int *p = &numbers[0];
    la   $t0, numbers    # int *q = &numbers[4];
    addi $t1, $t0, 16    #
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:
```

source code for pointer5.s

Printing 1-d Arrays in MIPS - v1

C

```
int vec[5]={0,1,2,3,4};
 // ...
 int i = 0
 while (i < 5) {
  printf("%d", vec[i]);
  i++;
 }
 // ....
```

- i in $s0

MIPS

```
   # ...

   li   $s0, 0
loop:
   bge  $s0, 5, end
   la   $t0, vec
   mul  $t1, $s0, 4
   add  $t2, $t1, $t0
   lw   $a0, ($t2)
   li   $v0, 1
   syscall
   addi $s0, $s0, 1
   b    loop
end:

   # ...
   .data
```

Example C with unaligned accesses

```c
uint8_t bytes[32];
uint32_t *i = (int *)bytes[1];
// illegal store - not aligned on a 4-byte boundary
*i = 0x03040506;
printf("%d\n", bytes[1]);
```

source code for unalign.c

Example MIPS with unaligned accesses

```
    .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
```

source code for unalign.s

Example MIPS with unaligned accesses

```
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
```

source code for unalign.s

Data Structures and MIPS

C data structures and their MIPS representations:

- `char` ... as byte in memory, or register
- `int` ... as 4 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

Printing 1-d Array in MIPS - v2

C

```c
int vec[5]={0,1,2,3,4};
 // ...
 int *p = &vec[0];
 int *end = &vec[4];
 while (p <= end) {
  int y = *p;
  printf("%d", y);
  p++;
 }
 // ....
```

- p in $s0
- end in $s1

MIPS

```
    li   $s0, vec
    la   $t0, vec
    add  $s1, $t0, 16
loop:
    bgt  $s0, $s1, end
    lw   $a0, 0($s0)
    li   $v0, 1
    syscall
    addi $s0, $s0, 4
    b    loop
end:
    .data
vec: .word 0,1,2,3,4
```

Computing sum of 2-d Array : C

Assume we have a 2d-array:

```
int32_t matrix[6][5];
```

We can sum its value like this in C

```
int row, col, sum = 0;
// row-by-row
for (row = 0; row < 6; row++) {
   // col-by-col within row
   for (col = 0; col < 5; row++) {
      sum += matrix[row][col];
   }
}
```

MIPS directives for an equivalent 2d-array

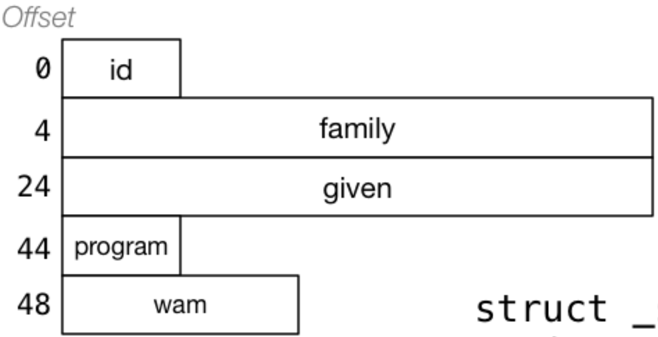
`mips     .data matrix: .space 120 # 6 * 5 == 30 array elements each 4 bytesmips`
.text

“‘

Computing sum of 2-d Array : MIPS

Computing sum of 2-d Array : MIPS

```
        li   $s0, 0            # sum = 0
        li   $s2, 0            # row = 0
loop1:  bge  $s2, 6, end1      # if (row >= 6) break
        li   $s4, 0            # col = 0
loop2:  bge  $s4, 5, end2      # if (col >= 5) break
        la   $t0, matrix
        mul  $t1, $s2, 20      # t1 = row*rowsize
        mul  $t2, $s4, 4       # t2 = col*intsize
        add  $t3, $t0, $t1     # offset = t0+t1
        add  $t4, $t3, $t2     # offset = t0+t1
        lw   $t5, 0($t4)       # t0 = *(matrix+offset)
        add  $s0, $s0, $t5     # sum += t0
        addi $s4, $s4, 1       # col++
        j    loop2
end2:   addi $s2, $s2, 1       # row++
        j    loop1
end1:
```




```c
struct _student {
    int     id;
    char    family[20];
    char    given[20];
    int     program;
    double  wam;
};
```

Implementing Structs in MIPS

C **struct** definitions effectively define a new type.

```
// new type called "struct student"
struct student {...};

// new type called student_t
typedef struct student student_t;
```

Instances of structures can be created by allocating space:

```
                  # sizeof(Student) == 56
stu1:             # student_t stu1;
     .space 56
stu2:             # student_t stu2;
     .space 56
stu:
     .space 4     # student_t *stu;
```

Implementing Structs in MIPS

Accessing structure components is by offset, not name

```
li   $t0  5012345
la   $t1, stu1
sw   $t0, 0($t1)        # stu1.id = 5012345;
li   $t0, 3778
sw   $t0, 44($t1)       # stu1.program = 3778;

la   $s1, stu2          # stu = &stu2;
li   $t0, 3707
sw   $t0, 44($s1)       # stu->program = 3707;
li   $t0, 5034567
sw   $t0, 0($s1)        # stu->id = 5034567;
```

COMP1521 21T2 — MIPS Functions

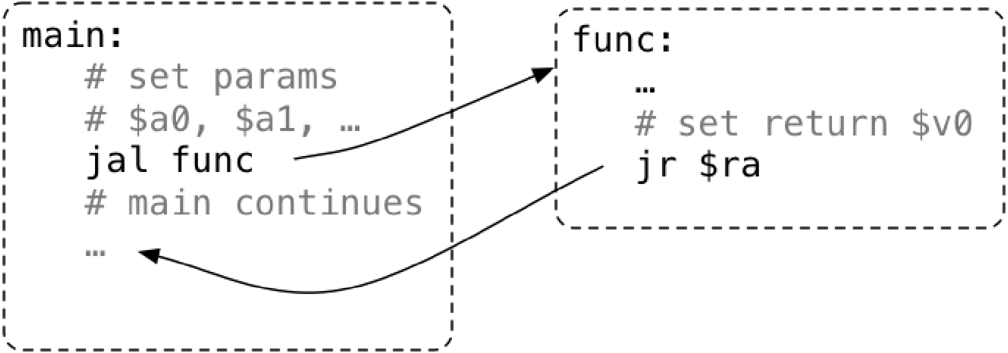
https://www.cse.unsw.edu.au/~cs1521/21T2/

When we call a function:

- the arguments are evaluated and set up for function
- control is transferred to the code for 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

Simple view of function calls:

- load argument values into **`$a0`**, **`$a1`**, **`$a2`**, **`$a3`**.
- **`jal function`** set **`$ra`** to PC+4 and jumps to function
- function puts return value in **`$v0`**
- returns to caller using **`jr $ra`**



Function with No Parameters or Return Value

- **jal hello** sets **$ra** to address of following instruction, and transfers execution to **hello**
- **`jr $ra`** transfers execution to the address in **$ra**

```c
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```
main:
    ...
    jal  hello
    ...
hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra
    .data
string:
    .asciiz "hi\n"
```

Function with a Return Value but No Parameters

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

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

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

```
main:
    ...
    jal  answer
    move $a0, $v0
    li   $v0, 1
    syscall
    ...
answer:
    li $v0, 42
    jr $ra
```

Function with a Return Value and Parameters

By convention, first 4 parameters are passed in **$a0**, **$a1**, **$a2**, **$a3**; if there are more (or more complex) parameters, they are passed on the stack

```c
int main(void) {
    int a = product(6, 7);
    printf("%d\n", a);
    return 0;
}

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

```
main:
    ...
    li   $a0, 6
    li   $a1, 7
    jal  product
    move $a0, $v0
    li   $v0, 1
    syscall
    ...
product:
    mul  $v0, $a0, $a1
    jr   $ra
```

Function calling another function ... DO NOT DO THIS

A function that calls another function *must* save **`$ra`**.

The **`jr $ra`** in `main` below **will fail**, because **`jal hello`** changed **`$ra`**

```c
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```
main:
    jal  hello
    li $v0, 0
    jr $ra # THIS WILL FAIL
hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra
    .data
string: .asciiz "hi\n"
```

Simple Function Call Example - C

```
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");
}
```

source code for call_return.c

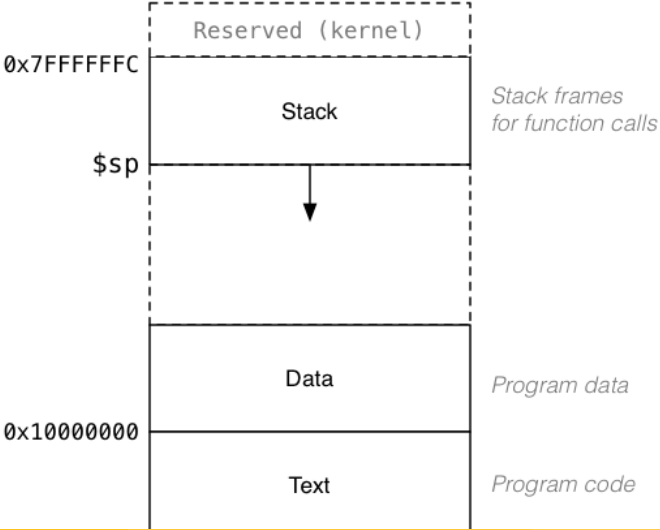
Simple Function Call Example - broken MIPS

```
    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
    li   $v0, 0         # fails because $ra changes since main called
    jr   $ra            # return from function main
f:
    la   $a0, string2   # printf("in function f\n");
    li   $v0, 4
    syscall
    jr   $ra            # return from function f
    .data
```

source code for call_return.broken.s

The Stack: Where it is in Memory

Data associated with a function call placed on the stack:



The Stack: Allocating Space

- **$sp** (stack pointer) initialized by operating system
- always 4-byte aligned (divisible by 4)
- points at currently used (4-byte) word
- grows downward (towards smaller addresses)
- a function can do this to allocate 40 bytes:

```
sub  $sp, $sp, 40    # move stack pointer down
```

- a function **must** leave $sp at original value
- so if you allocated 40 bytes, before return (**jr $ra**)

```
add  $sp, $sp, 40    # move stack pointer back
```

The Stack: Saving and Restoring Registers

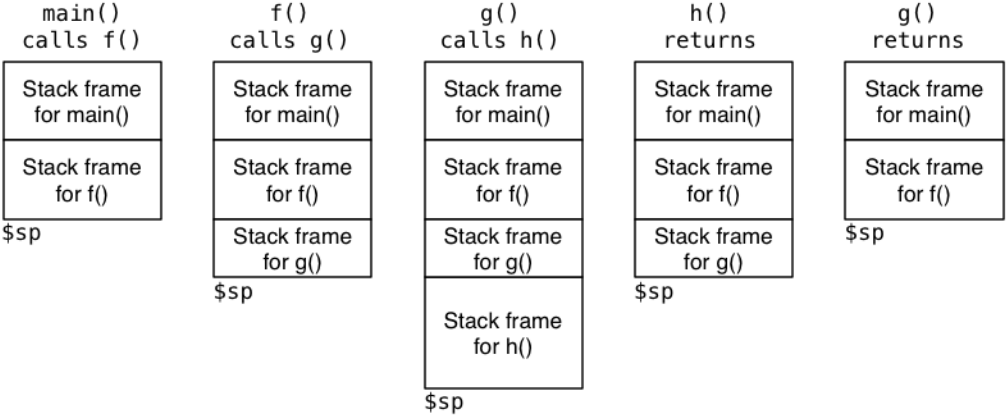
```
f:
    sub  $sp, $sp, 12    # allocate 12 bytes
    sw   $ra, 8($sp)     # save $ra on $stack
    sw   $s1, 4($sp)     # save $s1 on $stack
    sw   $s0, 0($sp)     # save $s0 on $stack

    ...

    lw   $s0, 0($sp)     # restore $s0 from $stack
    lw   $s1, 4($sp)     # restore $s1 from $stack
    lw   $ra, 8($sp)     # restore $ra from $stack
    add  $sp, $sp, 12    # move stack pointer back
    jr   $ra             # return
```

The Stack: Growing & Shrinking

How stack changes as functions are called and return:



Function calling another function ... how to do it right

A function that calls another function must save **$ra**.

```
main:
    sub   $sp, $sp, 4     # move stack pointer down
                          # to allocate 4 bytes
    sw    $ra, 0($sp)     # save $ra on $stack

    jal   hello           # call hello

    lw    $ra, 0($sp)     # recover $ra from $stack
    add   $sp, $sp, 4     # move stack pointer back up
                          # to what it was when main called
    li    $v0, 0          # return 0
    jr    $ra             #
```

Simple Function Call Example - correct MIPS

```
    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
```

source code for call_return.s

MIPS Register usage conventions

- $a0..$**a3** contain first 4 arguments
- **$v0** contains return value
- **$ra** contains return address
- if function changes **`$sp`**, **`$fp`**, **`$s0`**..**`$s8`** it restores their value
- callers assume **`$sp`**, **`$fp`**, **`$s0`**..**`$s8`** unchanged by call (**`jal`**)
- a function may destroy the value of other registers e.g. **`$t0`**..**`$t9`**
- callers must assume value in e.g. **`$t0`**..**`$t9`** changed by call (**`jal`**)

MIPS Register usage conventions (not covered in COMP1521)

- floating point registers used to pass/return float/doubles
- similar conventions for saving floating point registers
- stack used to pass arguments after first 4
- stack used to pass arguments which do not fit in register
- stack used to return values which do not fit in register
- for example C argument or return value can be a struct, which is any number of bytes

Example - Returning a Value - C

```c
int answer(void);
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}
int answer(void) {
    return 42;
}
```

source code for return_answer.c

Example - Returning a Value - 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
```

source code for return_answer.s

Example - Argument & Return - C

```c
void two(int i);
int main(void) {
    two(1);
}
void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

source code for two_powerful.c

Example - Argument & Return - MIPS (main)

```
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 calle
    jr   $ra            # return from function main
```

source code for two_powerful.s

Example - Argument & Return - MIPS (two)

```
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 calle
    jr   $ra            # return from two
```

source code for two_powerful.s

Example - More complex Calls - C

```c
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;
}
```

source code for more_calls.c

Example - more complex Calls - MIPS (main)

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

source code for more_calls.s

Example - more complex Calls - MIPS (sum_product)

```
sum_product:
    addi $sp, $sp, -12  # move stack pointer down to make room
    sw   $ra, 8($sp)    # save $ra on $stack
    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
    addi $sp, $sp, 12   # move stack pointer back up to what it was when main calle
    jr   $ra            # return from sum_product
```

source code for more_calls.s

Example - more complex Calls - MIPS (product)

- a function which doesn't call other functions is called a leaf function
- its code *can* be simpler...

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

source code for more_calls.c

```
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            #
```

source code for more_calls.s

Example - strlen using array - C

C

```
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;
}
```

source code for strlen_array.c

Simple C

```
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;
}
```

source code for strlen_array.simple.c

Example - pointer - C

```
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;
}
```

source code for pointer.c

Example - pointer - 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            #
```

Example - strlen using pointer - C

```c
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;
}
```

source code for strlen_array.c

Example - strlen using pointer - MIPS (my_strlen)

```
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             #
```

source code for strlen_array.s

Storing A Local Variables On the Stack

- some local (function) variables must be stored on stack
- e.g. variables such as arrays and structs

```c
int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
```

source code for squares.c

```
main:
    sub   $sp, $sp, 40
    li    $t0, 0
loop0:
    mul   $t1, $t0, 4
    add   $t2, $t1, $sp
    mul   $t3, $t0, $t0
    sw    $t3, ($t2)
    add   $t0, $t0, 1
    b     loop0
end0:
```

source code for squares.s

Example - strlen using pointer - C

```
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;
}
```

source code for strlen_array.c

What is a Frame Pointer

- 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 analyze stack
  - e.g 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 a priority

Example of Growing Stack Breaking Function Return

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

source code for frame_pointer.c

```
f:
    sub   $sp, $sp, 4
    sw    $ra, 0($sp)
    li    $v0, 5
    syscall
    # allocate space for
    # array on stack
    mul   $t0, $v0, 4
    sub   $sp, $sp, $t0
    # ... more code ...
    # breaks because $sp
    # has changed
    lw    $ra, 0($sp)
    add   $sp, $sp, 4
    jr    $ra
```

source code for frame_pointer.broken.s

Example of Frame Pointer Use

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

source code for frame_pointer.c

```
f:
    sub   $sp, $sp, 8
    sw    $fp, 4($sp)
    sw    $ra, 0($sp)
    add   $fp, $sp, 8
    li    $v0, 5
    syscall
    mul   $t0, $v0, 4
    sub   $sp, $sp, $t0
    # ... more code ...
    lw    $ra, -4($fp)
    move $sp, $fp
    lw    $fp, 0($fp)
    jr    $ra
```

source code for frame_pointer.s

COMP1521 21T2 — Processes

https://www.cse.unsw.edu.au/~cs1521/21T2/

Processes

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.

On Unix/Linux:

- each process had a unique process ID, or PID:
  a positive integer, type **`pid_t`**, defined in `<unistd.h>`
- PID 1: ***init***, used to boot the system.
- low-numbered processes usually system-related, started at boot
  - ... but PIDs are recycled, so this isn't *always* true
- some parts of the operating system may appear to run as processes
  - many *nix-like systems use PID 0 for the operating system

Process Parents

Each process has a *parent process*.

- initially, the process that created it;
- if a process' parent terminates, its parent becomes *init* (PID 1)

Unix provides a range of commandss for manipulating processes, e.g.:

- `sh` ... creating processes via object-file name
- `ps` ... showing process information
- `w` ... showing per-user process information
- `top` ... showing high-cpu-usage process information
- `kill` ... sending a signal to a process

Aside: Zombie Process

Zombie Process?

Photo credit: Kenny Louie, Flickr.com

Aside: Zombie Processes

A process cannot terminate until its parent is notified.

- if `exit()` called, operating system sends `SIGCHLD` signal to parent
- `exit()` will not return until parent handles `SIGCHLD`

Zombie process = exiting process waiting for parent to handle `SIGCHLD`

- all processes become zombies until `SIGCHLD` handled
- bug in parent that ignores `SIGCHLD` creates long-term zombie processes
  - wastes some operating system resources

Orphan process = a process whose parent has exited

- when parent exits, orphan assigned PID 1 (*init*) as its parent
- *init* should always handles `SIGCHLD` when process exits

Multi-Tasking

On a typical modern operating system...

- multiple processes are active “simultaneously” (*multi-tasking*)
- operating systems provides a virtual machine to each process:
  - each process executes as if the only process running on the machine
  - e.g. each process has its own address space (N bytes, addressed 0..N-1)

When there are multiple processes running on the machine,

- a process uses the CPU, until it is *preempted* or exits;
- then, another process uses the CPU, until it too is preempted.
- eventually, the first process will get another run on the CPU.

Multi-tasking

Overall impression: three programs running simultaneously.
(In practice, these time divisions are imperceptibly small!)

Preemption — When? How?

What can cause a process to be preempted?

- it ran “long enough”, and the OS replaces it by a waiting process
- it needs to wait for input, output, or other some other operation

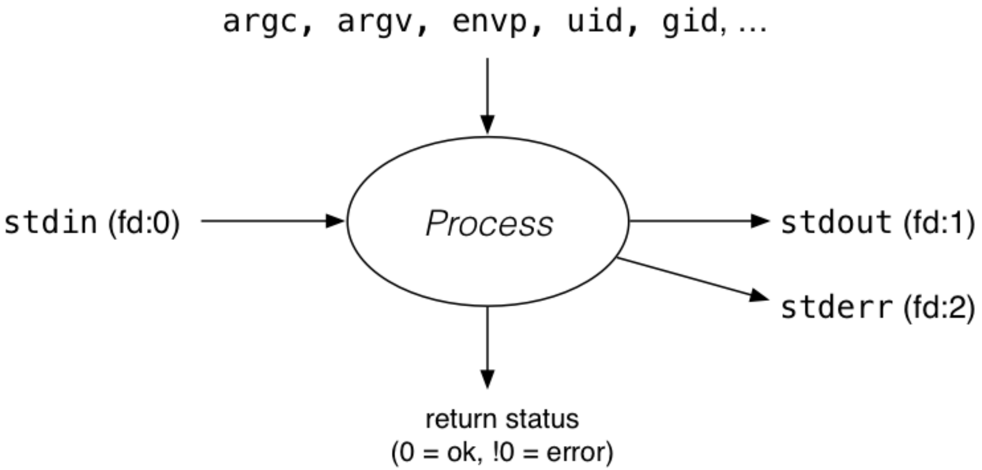
On preemption…

- the process’s entire state is saved
- the new process’s state is restored
- this change is called a ***context switch***
- context switches are *very* expensive!

Which process runs next? The ***scheduler** answers this.
The operating system’s process scheduling attempts to:

- fairly sharing the CPU(s) among competing processes,
- minimize response delays (lagginess) for interactive users,
- meet other real-time requirements (e.g. self-driving car),
- minimize number of expensive context switches

Environment for processes running on Unix/Linux systems



Process-related Unix/Linux Functions/System Calls

Process information:

- `getpid()` ... get process ID
- `getppid()` ... get parent process ID
- `getpgid()` ... get process group ID

Creating processes:

- `posix_spawn()` ... create a new process.
- `fork()` ... duplicate current process. (do not use in new code)
- `vfork()` ... duplicate current process. (do not use in new code)
- `execvp()` ... replace current process.
- `system()`, `popen()` ... create a new process via a shell (*unsafe*)

Destroying processes:

- `exit()` ... terminate current process, see also
  - `_exit()` ... terminate *immediately*
    atexit functions not called, stdio buffers not flushed
- `waitpid()` ... wait for state change in child process

posix_spawn() — Run a new process

```c
#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[]);
```

Creates a new process.

- `path`: path to the process to run
- `argv`: arguments to pass to new program
- `envp`: environment to pass to new program
- `pid`: returns process id of new program
- `file_actions`: specifies *file actions* to be performed before running program
  - can be used to redirect *stdin*, *stdout* to file or pipe
- `attrp`: specifies attributes for new process
  - not used/covered in COMP1521

Example: using posix_spawn() to run /bin/date

```
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);
```

source code for spawn.c

fork() — clone yourself

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

pid_t fork(void);
```

Creates new process by duplicating the calling process.

- new process is the *child*, calling process is the *parent*

Both child and parent return from `fork()` call... how do we tell them apart?

- in the child, `fork()` returns 0
- in the parent, `fork()` returns the pid of the child
- if the system call failed, `fork()` returns -1

Child inherits copies of parent's address space, open file descriptors, ...

Do not use in new code! Use *`posix_spawn()`* instead. *`fork()`* appears simple, but is prone to subtle bugs

Example: using fork()

```c
// 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);
}
```

source code for fork.c

```
$ dcc fork.c
$ a.out
I am the parent because fork() returned 2884551.
I am the child because fork() returned 0.
$
```

execvp() - replace yourself

```
#include <unistd.h>

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

Replace the program in the currently-executing process.

- `file`: an executable — either a binary, or script starting with `#!`
- `argv`: arguments to pass to 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 ... where would it return *to*?

Example: using exec()

```c
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");
```

source code for exec.c

```
$ dcc exec.c
$ a.out
good-bye cruel world
$
```

Example: Using fork() and exec() to run /bin/date

```c
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;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }
    printf("/bin/date exit status was %d\n", exit_status);
}
```

source code for fork_exec.c

system() — convenient but unsafe way to run another program

```c
#include <stdlib.h>

int system(const char *command);
```

Runs **command** via **/bin/sh**.

Waits for **command** to finish and returns exit status

Convenient ... but **extremely dangerous** —
very brittle; highly vulnerable to security exploits

- use for quick debugging and throw-away programs only

```c
// 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;
```

source code for system.c

Example: Running `ls -ld` via `posix_spawn()`

```
char *ls_argv[2];
ls_argv[0] = "/bin/ls";
ls_argv[1] = "-ld";
pid_t pid;
extern char **environ;
if (posix_spawn(&pid, "/bin/ls", NULL, NULL, ls_argv, environ) != 0) {
    perror("spawn"); exit(1);
}
```

Example: Running `ls -ld` via `system()`

```
system("ls -ld");
```

getpid(), getppid() — get process IDs

```c
#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 current process' parent.

waitpid() — wait for a process to change state

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

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

- **waitpid** pauses current process until process pid changes state
    - where state changes include finishing, stopping, re-starting, ...
- ensures that child resources are released on exit
- 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 specified process

```
pid_t wait(int *wstatus);
```

- equivalent to `waitpid(-1, &status, 0)`
- pauses until any child processes terminates.

waitpid() — wait for a process to change state

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

status is set to hold info about pid.

- e.g., exit status if pid 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

For more information, man 2 waitpid.

Environment Variables

- When run, a program is passed a set of ***environment variables***
  an array of strings of the form **name=value**, terminated with NULL.
- access via global variable **environ**
  - many C implementation also provide as 3rd parameter to main:

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

  - but in practice this is extremely hard to get right

```c
// print all environment variables
extern char **environ;
for (int i = 0; environ[i] != NULL; i++) {
    printf("%s\n", environ[i]);
}
```

source code for environ.c

- Most programs instead use **getenv()** and **setenv()** to access environment variables

getenv() — get an environment variable

```
#include <stdlib.h>

char *getenv(const char *name);
```

- search environment variable array for **name=value**
- returns **value**
- returns **NULL** if **name** not in environment variable array

```
// print value of environment variable STATUS
char *value = getenv("STATUS");
printf("Environment variable 'STATUS' has value '%s'\n", value);
```

source code for get_status.c

setenv() — set an environment variable

```
#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 changed if **overwrite** is non-zero

```
// 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,
    getenv_argv, environ) != 0) {
    perror("spawn");
    exit(1);
}
```

source code for set_status.c

Example: Changing behaviour with an environment variable

```c
pid_t pid;
char *date_argv[] = { "/bin/date", NULL };
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);
```

source code for spawn_environment.c

exit() — terminate yourself

```c
#include <stdlib.h>

void exit(int status);
```

- triggers any functions registered as atexit()
- flushes stdio buffers; closes open FILE *'s
- terminates current process
- a SIGCHLD signal is sent to parent
- returns status to parent (via waitpid())
- any child processes are inherited by init (pid 1)

```c
void _exit(int status);
```

- terminates current process without triggering functions registered as atexit()
- stdio buffers not flushed

pipe() — stream bytes between processes

```c
#include <unistd.h>

int pipe(int pipefd[2]);
```

A **pipe** is a unidirectional byte stream provided by the operating system.

- **`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[1]`**

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 they are not using
  - 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 file descriptors can be used with stdio via **fdopen**.

popen() — a 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 status

Convenient, but brittle and highly vulnerable to security exploits ...
use for quick debugging and throw-away programs only

Example: capturing process output with popen()

```c
// 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
```

source code for read_popen.c

Example: sending input to a process with popen()

```
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");
    if (p == NULL) {
        perror("");
        return 1;
    }
    fprintf(p, "plz date me\n");
    pclose(p); // returns command exit status
    return 0;
}
```

source code for write_popen.c

posix_spawn and pipes (advanced topic)

```
int posix_spawn_file_actions_destroy(
    posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_init(
    posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
    posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
    posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);
```

- functions to combine file operations with posix_spawn process creation
- awkward to understand and use — but robust

Example: capturing output from a process: source code for spawn_read_pipe.c

Example: sending input to a process: source code for spawn_write_pipe.c

COMP1521 21T2 — Signals

https://www.cse.unsw.edu.au/~cs1521/21T2/

Linux/Unix Signals

- signal are simple form of interprocess-communication
- signals can be generated from a variety of sources
  - from another process via `kill()`
  - from the operating system (e.g. timer)
  - from within the process (e.g. system call)
  - from a fault in the process (e.g. div-by-zero)
- processes can define how they want to handle signals
  - using the `signal()` library function (simple)
  - using the `sigaction()` system call (powerful)
- signal `SIGKILL` always terminates receiving processes
- only owner of a processes can send signal to it

Signal Handling

Default handling of signal can be:

- ***Term*** ... terminate the process
- ***Ign*** ... ignored; the signal does nothing
- ***Core*** ... terminate the process and dump memory image to file named core
- ***Stop*** ... pause the process
- ***Cont*** ... continue the process (if paused)

Processes can choose to ignore a signal.

Processes can set a custom *signal handler* for signal.

... except for `SIGKILL` and `SIGSTOP`, which cannot be caught, blocked, or ignored.

See `man 7 signal` for details of signals and default handling.

Operating System-Generated Signals

Signals from internal process activity, e.g.

- `SIGILL` ... illegal instruction (***Term*** by default)
- `SIGABRT` ... generated by `abort()` (***Core*** by default)
- `SIGFPE` ... floating point exception (***Core*** by default)
- `SIGSEGV` ... invalid memory reference (***Core*** by default)

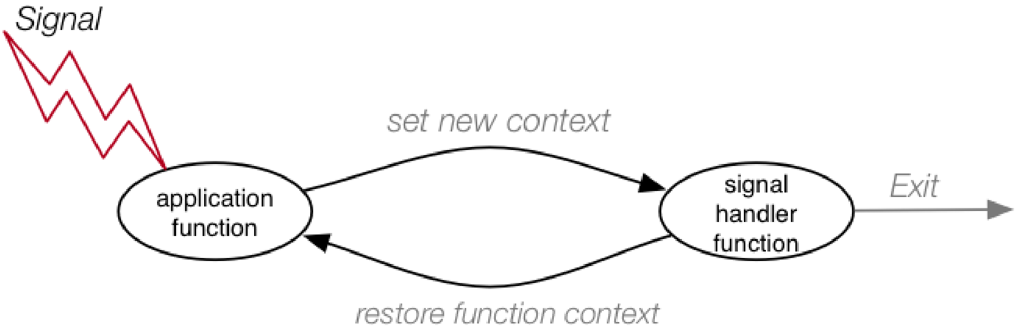
Signals from external process events, e.g.

- `SIGHUP` ... hangup detected on controlling terminal/process
- `SIGINT` ... interrupt from keyboard (ctrl-c) (***Term*** by default)
- `SIGPIPE` ... broken pipe (***Term*** by default)
- `SIGCHLD` ... child process stopped or died (***Ign*** by default)
- `SIGTSTP` ... stop typed at tty (ctrl-z) (***Stop*** by default)

Signal Handlers

Signal Handler = a function invoked in response to a signal

- knows which signal it was invoked by
- needs to ensure that invoking signal (at least) is blocked
- carries out appropriate action; may return



signal() — installing a signal handler, the old way

```c
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- old way to create signal handler - do not use in new code
- set how to handle a signal **signum** (e.g. SIGINT)
- **handler** can be one of ...
  - SIG_IGN ... ignore signal **signum**
  - SIG_DFL ... use default handler for **signum**
  - a user-defined function for **signum** signals
    - function type must be void (int)
- returns previous value of signal handler, or SIG_ERR

sigaction() — installing a signal handler, the new way

```
#include <signal.h>

int sigaction (
    int signum,
    const struct sigaction *act,
    struct sigaction *oldact);
```

- set how to handle a signal **signum** (e.g. SIGINT)
- **act** defines how signal should be handled
- **oldact** saves a copy of how signal was handled
- if `act->sa_handler == SIG_IGN`, signal is ignored
- if `act->sa_handler == SIG_DFL`, default handler is used
- on success, returns 0; on error, returns -1 and sets `errno`

For much more information: `man 2 sigaction`

Signal Handlers

Details on `struct sigaction` ...

```
struct sigaction {
    void (*sa_handler) (int);
    void (*sa_sigaction) (int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    /* ... */
};
```

- `void (*sa_handler)(int)`
  - pointer to a handler function, or `SIG_IGN` or `SIG_DFL`
- `void (*sa_sigaction)(int, siginfo_t *, void *)`
  - pointer to handler function; used if `SA_SIGINFO` flag is set
  - allows more context info to be passed to handler
- `sigset_t sa_mask`
  - a mask, where each bit specifies a signal to be blocked
- `int sa_flags`
  - flags to modify how signal is treated
    (e.g., don't block signal in its own handler)

Details on `siginfo_t` ...

```
typedef struct {
    int       si_signo;  /* signal number of signal being handled */
    int       si_code;   /* signal code - more information about why */
    pid_t     si_pid;    /* process ID of sending process */
    uid_t     si_uid;    /* user ID of owner of sending process */
    void     *si_addr;   /* address of faulting instruction */
    int       si_status; /* exit value for process termination */
    /* ... */
} siginfo_t;
```

System-dependent; these are (a subset of) mandated fields.

Waiting for an event ... the dumb way

```c
#include <signal.h>
void signal_handler(int signum) {
    printf("signal number %d received\n", signum);
}
int main(void) {
    struct sigaction action = {.sa_handler = signal_handler};
    sigaction(SIGUSR1, &action, NULL);
    printf("I am process %d waiting for signal %d\n", getpid(), SIGUSR1);
    // loop waiting for signal
    // bad consumes CPU/electricity/battery
    // sleep would be better
    while (1) {
    }
}
```

source code for busy_wait_for_signal.c

Waiting for an event ... the smart way

```c
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

- sleep() suspended the caller for **seconds** of real-time
- efficient way to wait for an event such as an signal
- allows operating system to run other processes

Example: waiting for an event

```c
#include <signal.h>
void signal_handler(int signum) {
    printf("signal number %d received\n", signum);
}
int main(void) {
    struct sigaction action = {.sa_handler = signal_handler};
    sigaction(SIGUSR1, &action, NULL);
    printf("I am process %d waiting for signal %d\n", getpid(), SIGUSR1);
    // suspend execution for 1 hour
    sleep(3600);
}
```

source code for wait_for_signal.c

kill() — sending signals

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

int kill(pid_t pid, int sig);
```

- send signal number **sig** to process number **pid**
- if successful, return 0; on error, return -1 and set errno

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <signal> <pid>\n", argv[0]);
        return 1;
    }
    int signal = atoi(argv[1]);
    int pid = atoi(argv[2]);
    kill(pid, signal);
}
```

source code for send_signal.c

Example: ignoring a signal

```c
#include <signal.h>
int main(void) {
    // catch SIGINT which is sent if user types cntrl-d
    struct sigaction action = {.sa_handler = SIG_IGN};
    sigaction(SIGINT, &action, NULL);
    while (1) {
        printf("Can't interrupt me, I'm ignoring ctrl-C\n");
        sleep(1);
    }
}
```

source code for ignore_control_c.c

Example: a simple signal handler

```c
#include <signal.h>
void ha_ha(int signum) {
    printf("Ha Ha!\n"); // I/O can be unsafe in a signal handler
}
int main(void) {
    // catch SIGINT which is sent if user types cntrl-d
    struct sigaction action = {.sa_handler = ha_ha};
    sigaction(SIGINT, &action, NULL);
    while (1) {
        printf("Can't interrupt me, I'm ignoring ctrl-C\n");
        sleep(1);
    }
}
```

source code for laugh_at_control_c.c

Example: another simple signal handler

```c
#include <signal.h>
int signal_received = 0;
void stop(int signum) {
    signal_received = 1;
}
int main(void) {
    // catch SIGINT which is sent if user types cntrl-C
    struct sigaction action = {.sa_handler = stop};
    sigaction(SIGINT, &action, NULL);
    while (!signal_received) {
        printf("Type ctrl-c to stop me\n");
        sleep(1);
    }
    printf("Good bye\n");
}
```

source code for stop_with_control_c.c

Example: catching an internal error with a signal handler

```c
#include <signal.h>
#include <stdlib.h>
void report_signal(int signum) {
    printf("Signal %d received\n", signum);
    printf("Please send help\n");
    exit(0);
}
int main(int argc, char *argv[]) {
    struct sigaction action = {.sa_handler = report_signal};
    sigaction(SIGFPE, &action, NULL);
    // this will produce a divide by zero
    // if there are no command-line arguments
    // which will cause program to receive SIGFPE
    printf("%d\n", 42/(argc - 1));
    printf("Good bye\n");
}
```

source code for catch_error.c

COMP1521 21T2 — Files

https://www.cse.unsw.edu.au/~cs1521/21T2/

Operating system - What Does it Do.

- Operating system sits between the user and the hardware
- Operating system effectively provides a virtual machine to each user
- This virtual machine is much simpler and more convenient than real machine
- The virtual machine interface can be consistent across different hardware.
    - program can portably access hardware across different hardware configurations
    - linux available for almost all suitable hardware
- can coordinate/share access to resources between users
- can provide privileges/security

Operating Systems - What Does it Need from Hardware.

- needs hardware to provide a **privileged** mode which:
  - allows access to all hardware/memory
  - Operating System (kernel) runs in **privileged** mode
  - allows transfer to running code a **non-privileged** mode
- needs hardware to provide a **non-privileged** mode which:
  - prevents access to hardware
  - limits access to memory
  - provides mechanism to make requests to operating system
- operating system requests are called **system calls**
  - system calls transfers execution back to kernel code in **privileged** mode

System Call - What is It

- system call allow programs to request hardware operations
- system call transfers execution to OS code in **privileged** mode
  - includes arguments specifying details of request being made
  - OS checks operation is valid & permitted
  - OS carries out operation
  - transfers execution back to user code in **non-privileged** mode
- different operating system have different system calls
  - e.g Linux provides completley different system calls to Windows
- Linux provides 400+ system calls
- Operations likely to be provide by system calls:
  - 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 provides a virtual machine which can execute MIPS programs
- SPIM also provides a tiny operating system
- small number of SPIM system calls for I/O and memory allocation
- access is via the **`syscall`** instruction
- MIPS programs running on real hardware also use **`syscall`**
  - on linux **`syscall`**, will pass execution to linux kernel
- SPIM system calls are designed for students writing tiny programs
  - e.g SPIM system call 1 - print an integer
- system calls on real operating systems more general
  - e.g. system call might be write n bytes
- in real operating system library systems calls more general
  - library functions like **`printf`** provide convenient operations

Hello Systems Calls!

```c
// hello world implemented with a direct syscall
#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;
}
```

source code for hello_syscalls.c

Using read & write system calls to copy stdin to stdout

```
// 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
}
```

source code for cat_syscalls.c

What Really are Files and Directories?

- ***file systems*** manage persistent stored data e.g. on magnetic disk or SSD
- On Unix-like systems:
  - a ***file*** is sequence (array) of zero or more bytes.
  - no meaning for bytes associated with file
    - file metadata doesn't record that it is e.g. ASCII, MP4, JPG, ...
    - Unix-like files are just bytes
  - a ***directory*** is an object containing zero or more files or directories.
- file systems maintain metadata for files & directories, e.g. permissions
- 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.

Unix-like Files & Directories

- Unix-like filenames are sequences of 1 or more bytes.
  - filenames can contain any byte except **`0x00`** and **`0x2F`**
  - **`0x00`** bytes (ASCII '\0') used to terminate filenames
  - **`0x2F`** bytes (ASCII '/') used to separate components of pathnames.
  - maximum filename length, depends on file system, typically 255
- Two filenames can not be used - they have a special meaning:
  - **`.`** current directory
  - **`..`** parent directory
- Some programs (shell, ls) treat filenames starting with **`.`** specially.
- Unix-like directories are sets of files or directories

Unix/Linux Pathnames

- Files & directories accessed via pathnames, e.g: `/home/z5555555/lab07/main.c`
- *absolute* pathnames start with a leading **/** and give full path from root

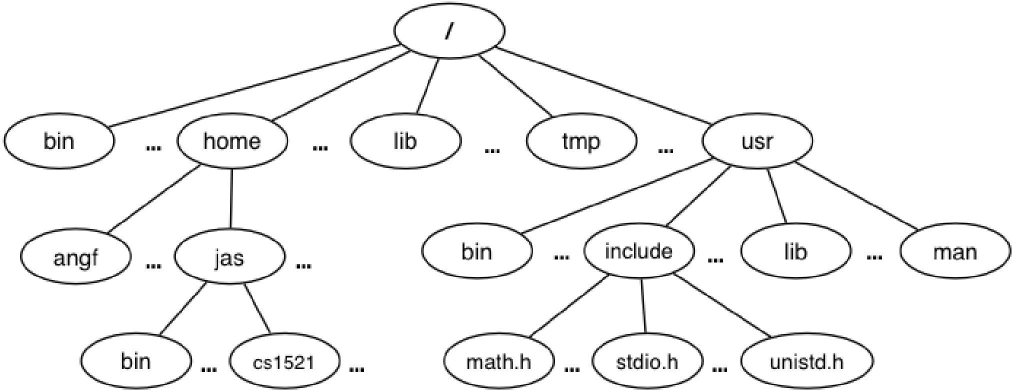
  e.g. `/usr/include/stdio.h`, `/cs1521/public_html/`
- every process (running program) has an associated *absolute* pathname called the *current working directory* (CWD)
- shell command pwd prints CWD
- *relative* pathname do not start with a leading **/** e.g. `../../another/path/prog.c`, `./a.out`, `main.c`
- *relative* pathnames appended to CWD of process using them
- Assume process CWD is `/home/z5555555/lab07/`
  `main.c` translated to absolute path `/home/z5555555/lab07/main.c`
  `../a.out` translated to absolute path `/home/z5555555/lab07/../a.out`
  which is equivalent to absolute path `/home/z5555555/a.out`

Everything is a File

- Originally files only managed data stored on a magnetic disk.
- Unix philosophy is: *Everything is a File*.
- File system can be used to access:
  - files
  - directories (folders)
  - storage devices (disks, SSD, ...)
  - peripherals (keyboard, mouse, USB, ...)
  - system information
  - inter-process communication
  - ...

Unix/Linux File System

Unix/Linux file system is tree-like



- We think of file-system as a *tree*
- But beware if you follow symbolic links it is a *graph*.
  - and you may infinitely loop attempting to traverse a file system

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 byte
- file ownership
- file access permissions - who can read, write, execute the file
- timestamps - time of creation/access/update

Note: file systems add much complexity to improve performance

- e.g. very small files might be stored in an inode itself

File Inodes

- unix-like file systems effectively have an array of inodes
- every inode has a ***inode-number*** (or ***i-number***)- its index in this array
- directories are effectively a list of (name, inode-number) pairs
- inode-number uniquely identify files within a filesystem
  - just a zid uniquely identifies a student within UNSW
- **ls -i** prints *inode-number*, e.g.:

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

File Access: Behind the Scenes

Access to files by name proceeds (roughly) as...

- open directory and scan for *name*
- if not found, "No such file or directory"
- if found as (*name*,`inumber`), access inode table `inodes[inumber]`
- collect file metadata and...
  - check file access permissions given current user/group
    - if don't have required access, "Permission denied"
  - collect information about file's location and size
  - update access timestamp
- use data in inode to access file contents

Hard Links & Symbolic Links

File system *links* allow multiple paths to access the same file

Hard links

- multiple names referencing the same file (inode)
- the two entries must be on the same filesystem
- all hard links to a file have equal status
- file destroyed when last hard link removed
- can not create a (extra) hard link to directories

Symbolic links (symlinks)

- point to another path name
- acessing the symlink (by default) accesses the file being pointed to
- symbolic link can point to a directory
- symbolic link can point to a pathname on another filesystems
- symbolic links don't have permissions (just a pointer)

Hard Links & Symbolic Links

```
$ 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 objects as a *stream of bytes*
- accessed via a *file descriptor*
    - file descriptors are small integers
    - index to a per-process operating system table (array)

Some important system calls:

- **`open`**`()` — open a file system object, returning 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

Using system call directly to create a file

```
// 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);
}
```

source code for cp_syscalls.c

C Library Wrappers for System Calls

- On Unix-like systems there are C library functions corresponding to each system call,
  - e.g. `open`, `read`, `write`, `close`
  - the **`syscall`** function is not used in normal coding
- These functions are not portable - absent from many platforms/implementations
- POSIX standardizes some of these functions
  - some non-Unix systems provide implementations of these functions
- better to use functions from standard C library, available everywhere
  - e.g fopen, fgets, fputc from **`stdio.h`**
  - on unix-like systems these will call `open`, `read`, `write`,
- but sometimes need to use lower level functions

Extra Types for File System Operations

Unix-like (POSIX) systems add some extra file-system-related C types in these include files:

```
#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 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`, ...

C library wrapper for open system call

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

- open file at **`pathname`**, according to **`flags`**
- **`flags`** is a bit-mask defined 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 (small non-negative `int`)
- if unsuccessful, return **`-1`** and set **`errno`**

C library wrapper for close system call

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

- release open file descriptor **`fd`**
- if successful, return `0`
- if unsuccessful, return `-1` and set **`errno`**
  - could be unsuccessful if **`fd`** is not an open file descriptor

    e.g. if **fd** has already been closed

An aside: removing a file e.g. via `rm`

- removes the file's entry from a directory
- but the inode and data persist 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

C library wrapper for read system call

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

- read (up to) **count** bytes from **fd** into **buf**
  - **buf** should point to array of at least **count** bytes
  - read does (can) not check **buf** points to enough space
- if successful, number of bytes actually read is returned
- 0 returned, if no more bytes to read
- -1 returned if error and `errno` set to reason
- next call to **read** will return next bytes from file
- repeated calls to reads will yield entire contents of file
  - associated with a file descriptor is "current position" in file
  - can also modify this position with `lseek`

C library wrapper for write system call

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

- attempt to write **count** bytes from *buf* into

  stream identified by file descriptor **`fd`**
- if successful, number of bytes actually written is returned
- if unsuccessful, return −1 and set **`errno`**
- does (can) not check **buf** points to **count** bytes of data
- next call to **write** will follow bytes already written
- file often created by repeated calls to write
  - associated with a file descriptor is “current position” in file
  - can also modify this position with **`lseek`**

Hello write!

```c
// hello world implemented with libc
#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;
}
```

source code for hello_libc.c

Using read & write to copy stdin to stdout

```c
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
    ssize_t bytes_read = read(0, bytes, 4096);
    if (bytes_read <= 0) {
        break;
    }
    write(1, bytes, bytes_read); // prints bytes to stdout
}
```

source code for cat_libc.c

Using open to copy a file

```c
// 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);
}
```

source code for cp_libc.c

C library wrapper for lseek system call

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

- change the 'current position' in the file of **fd**
- **offset** is in units of bytes, and can be 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 end of file leaves a gap which reads as 0's
- seeking back beyond start of file sets position to start of file

Example: `lseek(fd, 0, SEEK_END);` (move to end of file)

Using lseek to read the last byte then the first byte of a file

```c
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]);
```

source code for lseek.c

Using lseek to read bytes in the middle of a file

```
printf("first byte of the file is 0x%02x\n", bytes[0]);
// 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]);
```

source code for lseek.c

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 & efficient
- use them by default for file operations
- on Unix-like systems they will call open/read/write/...
  - but with buffering for efficiency

stdio.h - fopen/fclose

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

- `stdio.h` equivalent to `open`
- **mode** is string of 1 or more characters including:
  - **r** open text file for reading.
  - **w** open text file for writing truncated to 0 zero length if it exists created if does not exist
  - **a** open text file for writing writes append to it if it exists created if does not exist
- fopen returns a **FILE *** pointer
- **FILE** is an opaque struct - we can not access fields

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

- `stdio.h` equivalent to `close`

stdio.h - read and writing

```c
int fgetc(FILE *stream)             // read a byte
int fputc(int c, FILE *stream)      // write a byte

char *fputs(char *s, FILE *stream) // write a string

char *fgets(char *s, int size, FILE *stream) // read a line

// formatted input
int fscanf(FILE *stream, const char *format, ...)
// formatted output
int fprintf(FILE *stream, const char *format, ...)

// read array of bytes (fgetc + loop often better)
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
// write array of bytes (fputc + loop often better)
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream)
```

stdio.h - 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
for (char *p = &bytes[0]; *p != '\0'; p++) {
    fputc(*p, stdout);
}
```

source code for hello_stdio.c

stdio.h - using fputs, fwrite & fprintf to output bytes

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

```c
// 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);
```

source code for hello_stdio.c

stdio.h - using fgetc to copy stdin to stdout

```c
// 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
}
```

source code for cat_fgetc.c

stdio.h - using fgets to copy stdin to stdout

```c
// 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
//
// 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 fget/fputs for binary data e.g. jpgs
```

source code for cat_fgets.c

stdio.h - using fwrite to copy stdin to stdout

```
while (1) {
    char bytes[4096];
    ssize_t bytes_read = fread(bytes, 1, sizeof bytes, stdin);
    if (bytes_read <= 0) {
        break;
    }
    fwrite(bytes, 1, bytes_read, stdout);
}
```

source code for cat_fwrite.c

stdio.h - creating a file

```
// create file "hello.txt" containing 1 line: Hello, Andrew
#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;
    }
    fprintf(output_stream, "Hello, Andrew!\n");
    // fclose will flush data to file
    // best to close file ASAP
    // but doesn't matter as file autoamtically closed on exit
    fclose(output_stream);
    return 0;
}
```

source code for create_file_fopen.c

stdio.h - using fgetc to copy a file

```
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;
}
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
```

source code for cp_fgetc.c

stdio.h - using fwrite to copy a file

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

```
FILE *output_stream = fopen(argv[2], "wb");
```

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

source code for cp_fwrite.c

stdio.h - other operations

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

- **fseek** is stdio equivalent to lseek
- like `lseek` **offset** can be postive or negative
- like `lseek` **whence** can be `SEEK_SET`, `SEEK_CUR` or `SEEK_END` making **offset** relative to file start, current position or file end

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

- flush any buffered data on output stream

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

```
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));
```

source code for fseek.c

- NOTE: important error checking is missing above

Using fseek to read bytes in the middle of a file

```
// 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));
```

source code for fseek.c

- NOTE: important error checking is missing above

Using fseek to change a random file bit

```
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);
```

source code for fuzz.c

- random changes to search for errors/vulnerabilities called fuzzing

Using fseek to create a gigantic sparse file (advanced topic)

```c
// Create a 16 terabyte sparse file
// https://en.wikipedia.org/wiki/Sparse_file
// error checking omitted for clarity
#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;
}
```

source code for create_gigantic_file.c

- almost all the 16Tb are zeros which the file system doesn't actually store

stdio.h - convenience functions for stdin/stdout

- as we often read/write to stdin/stdout `stdio.h` provides convenience functions, we can use:

```
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

`stdio.h` provides useful functions which operate on strings

```
int snprintf(char *str, size_t size, const char *format, ...);
```

- like printf, but output goes to char array **str**
- handy for creating strings passed to other functions
- do not use unsafe related function: ‘sprintf

```
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
```

- like **snprintf** but dangerous because can overflow **str**

C library wrapper for stat system call

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

- retaurns metadata associated with **pathname** in **statbuf**
- metadata returned includes:
  - 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

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

- same as `stat()` but gets data via an open file descriptor

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

- same as `stat()` but doesn't follow symbolic links

definition of struct stat

```c
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

`st_mode` is a bitwise-or of these values (& others):

```
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
```

Using stat

```c
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);
```

source code for stat.c

mkdir

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

- create a new directory called **pathname** with permissions **mode**
- if **pathname** is e.g. `a/b/c/d`
  - all of the directories a, b and c must exist
  - directory c must be writeable to the caller
  - directory d must not already exist
- the new directory contains two initial entries
  - `.` is a reference to itself
  - `..` is a reference to its parent directory
- returns 0 if successful, returns -1 and sets `errno` otherwise

for example:

```
mkdir("newDir", 0755);
```

Example of using mkdir to create directories

```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) {
            perror(argv[arg]);  // prints why the mkdir failed
            return 1;
        }
    }
    return 0;
}
```

source code for mkdir.c

Other useful Linux (POSIX) functions

```c
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
```

file permissions

- file permissions are separated into three types:
  - **read * - permission to get bytes of file
  - **write* - permission to change bytes of file
  - **execute* - permission to execute file
- read/write/execute often represented as bits of an octal digit
- file permissions are specified for 3 groups of users:
  - **owner** - permissions for the file owner
  - **group** - permissions for users in the group of the file
  - **other** - permissions for any other user

changing file permssions

```c
// 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;
    }
}
```

source code for chmod.c

removing files

```c
int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        if (unlink(argv[arg]) != 0) {
            perror(argv[arg]);  // prints why the unlink failed
            return 1;
        }
    }
    return 0;
}
```

source code for rm.c

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

renaming a 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) {
        fprintf(stderr, "%s rename %s %s:", argv[0], old_filename,
                new_filename);
        perror("");
        return 1;
    }
    return 0;
}
```

source code for rename.c

cd-ing up one directory at a time

```c
// use repeated chdir("..") to climb to root of the file system
char pathname[PATH_MAX];
while (1) {
    if (getcwd(pathname, sizeof pathname) == NULL) {
        perror("getcwd");
        return 1;
    }
    printf("getcwd() returned %s\n", pathname);
    if (strcmp(pathname, "/") == 0) {
        return 0;
    }
    if (chdir("..") != 0) {
        perror("chdir");
        return 1;
    }
}
```

source code for getcwd.c

making a 1000-deep directory

```c
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);
}
```

source code for nest_directories.c

creating 1000 hard links to a file (creating the file)

```
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);
```

source code for many_links.c

creating 1000 hard links to a file (checking the file)

```
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);
```

source code for many_links.c

creating 1000 hard links to a file (creating a 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;
}
```

source code for many_links.c

POSIX functions to access directory contents (advanced)

```c
#include <sys/types.h>
#include <dirent.h>

// open a directory stream for directory name
DIR *opendir(const char *name);

// return a pointer to next directory entry
struct dirent *readdir(DIR *dirp);

// close a directory stream
int closedir(DIR *dirp);
```

Using opendir/readdir to print directory contents (advanced)

```c
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);
}
```

source code for list_directory.c

writing an array as binary data (using fwrite)

```c
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);
```

source code for write_array.c

reading an array as binary data (using fread)

```
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");
```

source code for read_array.c

writing a pointer as binary data (using fwrite)

```c
int array[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
int *p = &array[5];
FILE *f = fopen("array.save", "w");
```

```c
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);
```

source code for write_pointer.c

reading a pointer as binary data (using fread)

```
int array[10];
int *p;
FILE *f = fopen("array.save", "r");
```

```
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);
```

source code for read_pointer.c

I/O Performance & Buffering - Copying One Byte Per Time

```c
int read_file_descriptor = open(argv[1], O_RDONLY);
int write_file_descriptor = open(argv[2], O_WRONLY | O_CREAT, 0644);
// copy bytes 1 at a time
while (1) {
    char bytes[1];
    ssize_t bytes_read = read(read_file_descriptor, bytes, 1);
    if (bytes_read <= 0) {
        break;
    }
    write(write_file_descriptor, bytes, 1);
}
```

source code for cp_libc_one_byte.c

- similar to earlier example source code for cp_libc.c but one byte at time

I/O Performance & Buffering - Copying One Byte Per Time

```
$ clang -O3 cp_libc_one_byte.c -o cp_libc_one_byte
$ dd bs=1M count=10 </dev/urandom >random_file
10485760 bytes (10 MB, 10 MiB) copied, 0.183075 s, 57.3 MB/s
$ time ./cp_libc_one_byte random_file random_file_copy
real  0m5.262s
user  0m0.432s
sys   0m4.826s
```

- much slower than previous version which copies 4096 bytes at a time

```
$ clang -O3 cp_libc.c -o cp_libc
$  time ./cp_libc random_file random_file_copy
real  0m0.008s
user  0m0.001s
sys   0m0.007s
```

- main reason - system calls are expensive

I/O Performance & Buffering - stdio Copying 1 Byte Per Time

```c
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;
}
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
```

source code for cp_fgetc.c

I/O Performance & Buffering - stdio Copying 1 Byte Per Time

```
$ clang -O3 cp_fgetc.c -o cp_fgetc
$ time ./cp_fgetc random_file random_file_copy
real   0m0.059s
user   0m0.042s
sys    0m0.009s
```

- at the user level copies 1 byte at time using `fgetc`/`fputc`
- much faster that coping 1 byte at time using `read`/`write`
- little slower than coping 4096 bytes at time using `read`/`write`
- how?

I/O Performance & Buffering - stdio buffering

- assume stdio buffering size (`BUFSIZ`) is `4096` (typical)
- stdio **buffers** 1 byte `fgetc`/`fputc` into 4096 bytes `read`/`write`
- first `fgetc` reads 4096 bytes into an array (input ***buffer***)
  - next 4095 `fgetc` calls get byte from array
- first 4095 `fputc` put bytes into another array (output ***buffer***)
  - next 4095 `fgetc` get byte from array
- output buffer* emptied by **`exit`** or `main` returning
- data in output buffer
- program can force empty of output buffer with **`fflush`** call

reimplementing stdio.h no buffering - struct

```
// re-implementation of stdio functions fopen, fgetc, fputc, fclose
// with no buffering and *zero* error handling for clarity
#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;
```

source code for cp_unbuffered.c

reimplementing stdio.h no buffering - my_fopen

```
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;
}
```

source code for cp_unbuffered.c

reimplementing stdio.h no buffering - - my_fgetc

```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;
    }
}
```

source code for cp_unbuffered.c

reimplementing stdio.h no buffering - - my_fputc

```c
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;
    }
}
```

source code for cp_unbuffered.c

reimplementing stdio.h no buffering - - my_fclose

```c
int my_fclose(my_file_t *f) {
    int result = close(f->fd);
    free(f);
    return result;
}
```

source code for cp_unbuffered.c

reimplementing stdio.h - buffering (advanced topic)

- reimplementing stdio with input buffering

source code for cp_input_buffered.c

- and output buffering

source code for cp_output_buffered.c

File System Summary

Operating systems provide a *file system*

- as an abstraction over physical storage devices (e.g. disks)
- providing named access to chunks of related data (files)
- providing access (sequential/random) to the contents of files
- allowing files to be arranged in a hierarchy of directories
- providing control over access to files and directories
- managing other metadata associated with files (size, location, ...)

Operating systems also manage other resources

- memory, processes, processor time, i/o devices, networking, ...

Character Data

Huge number of character representations (encodings) exist
you need know only two:

- ASCII (ISO 646)
    - single byte values, only low 7-bit used, top bit always 0
    - can encode roman alphabet a-zA-Z, digits 0-9 , punctuation, 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, Italian, 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) ... **`'0'`**..**`'9'`**
  - upper case alphabetic (65..90) ... **`'A'`**..**`'Z'`**
  - lower case alphabetic (97..122) ... **`'a'`**..**`'z'`**
- Sequential nature of groups allow ordination e.g.
  `'3' - '0' == 3   'J' - 'A' == 10`
- See **`man 7 ascii`**

Unicode

- Widely-used standard for expressing “writing systems”
  - not all writing systems use a small set of discrete symbols
- Basically, a 32-bit representation of a wide range of symbols
  - around 140K symbols, covering 140 different languages
- Using 32-bits for *every* symbol would be too expensive
  - e.g. standard roman alphabet + punctuation needs only 7-bits
  - Several Unicode encodings have been developed
  - UTF-8 most widely used encoding, dominates web-use
  - designed by Ken Thompson on napkin in New Jersey diner

UTF-8 Encoding

#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 alphabets
- The 65536 3-byte codes include most Asian languages
- The 2097152 4-byte codes include symbols and emojis and ...

ch	*code-point*	*unicode binary*	*UTF-8 encoding*
$	U+0024	**0100100**	0**0100100**
¢	U+00A2	**00010100010**	110**00010** 10**100010**
€	U+20AC	**0010000010101100**	1110**0010** 10**000010** 10**101100**

Printing UTF-8 in a C program

```
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');
```

source code for hello.c

Converting Unicode Codepoints to UTF-8

```c
uint8_t encoding[5] = {0};
if (code_point < 0x80) {
    encoding[0] = code_point;
} else if (code_point < 0x800) {
    encoding[0] = 0xC0 | (code_point >> 6);
    encoding[1] = 0x80 | (code_point & 0x3f);
} else if (code_point < 0x10000) {
    encoding[0] = 0xE0 | (code_point >> 12);
    encoding[1] = 0x80 | ((code_point >> 6) & 0x3f);
    encoding[2] = 0x80 | (code_point  & 0x3f);
} else if (code_point < 0x200000) {
    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);
}
```

source code for utf8_encode.c

```
    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);
}
```

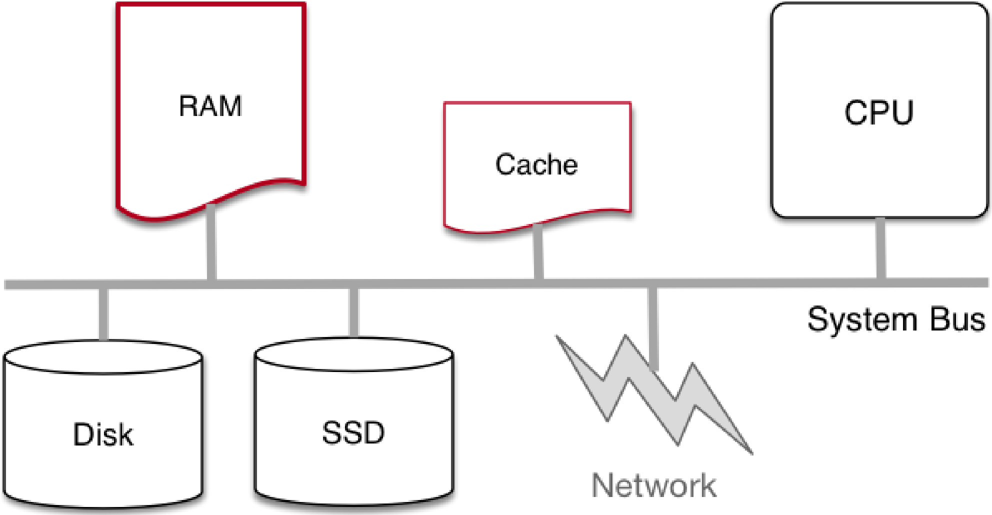
source code for utf8_encode.c

Summary of UTF-8 Properties

- Compact, but not minimal encoding; encoding allows you to resync immediately if bytes lost from a stream.
- ASCII is a subset of UTF-8 - complete backwards compatibility!
- All other UTF-8 bytes > 127 (0x7f)
  - no byte of multi-byte UTF-8 encoding is valid ASCII.
- No byte of multi-byte UTF-8 encoding is 0
  - can still use store UTF-8 in null-terminated strings.
- 0x2F (ASCII /) and 0x00 can not appear in multi-byte characters
  - hence can use UTF-8 for Linux/Unix filenames
- C programs can treat UTF-8 similarly to ASCII.
- Beware: number of bytes in UTF-8 string != number of characters.

Memory

Systems typically contain 4-16GB of volatile RAM



Single Process Resident in RAM without Operating System

- Many small embedded system run without 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 tricky.
- Widely used for simple micro-controllers.
- Parallelism and exploiting multiple-core CPUs problematic

Single Process Resident in RAM with Operating System

- Operating system 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 .
- Operating system must write out all memory of old process to disk and read all memory of new process from disk.
- OK for some uses, but inefficient in general.
- Little used in modern computing.

Multi Processes Resident in RAM without Virtual Memory

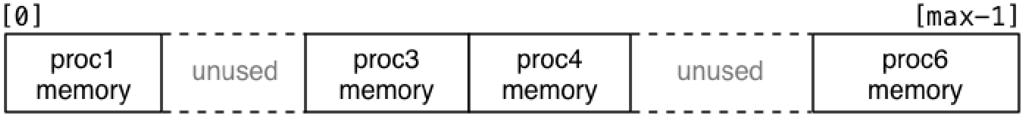
- If multiple processes to be resident in RAM O/S can swap execution between them quickly.
- RAM belonging to other processes & kernel must be protected
- Hardware support can limit process accesses to particular **segment** (region) of RAM.
- BUT program may be loaded anywhere in RAM to run
- Breaks instructions which use absolute addresses, e.g.: **`lw`**, **`sw`**, **`jr`**
- Either programs can't use absolute memory addresses (relocatable code)
- Or code has to be modified (relocated) before it is run - not possible for all code!
- Major limitation - much better if programs can assume always have same address space
- Little 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 an physical (actual) RAM address.
- Convenient for programming/compilers - each process has same virtual view of RAM.
- Can have multiple processes be in RAM, allowing fast switching
- Can load part of processes into RAM on demand.
- Provides a mechanism to share memory betwen processes.
- Address to fetch every instruction to be executed must be translated.
- Address for load/store instructions (e.g. **`lw`**, **`sw`**) must be translated .
- Translation needs to be really fast so largely implemented in hardware (silicon).

Virtual Memory with One Memory Segment Per Process

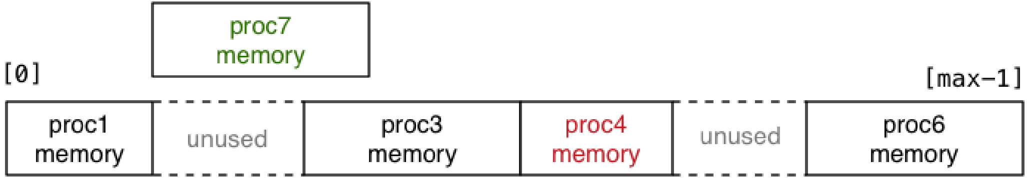
Consider a scenario with multiple processes loaded in memory:



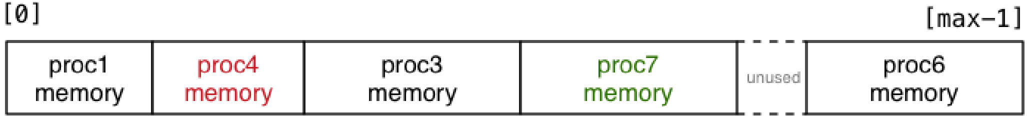
- Every process is in a contiguous section of RAM, starting at address **base** finishing at address **limit**.
- Each process sees its own address space as [0 .. size - 1]
- Process can be loaded anywhere in memory without change.
- Process accessing memory address **a** is translated to **a + base**
- and checked that **a + base** is **< limit** to ensure process only access its memory
- Easy to implement in hardware.

Virtual Memory with One Memory Segment Per Process

Consider the same scenario, but now we want to add a new process

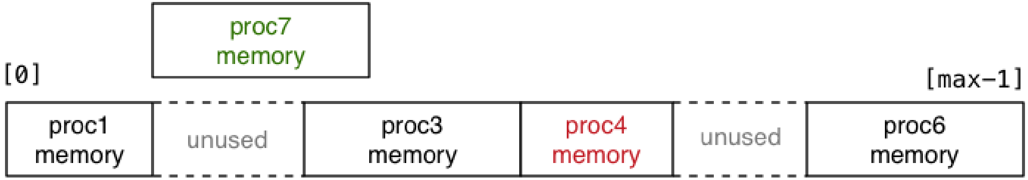


- The new process doesn't fit in any of the unused slots (fragmentation).
- Could move some process to make a single large slot

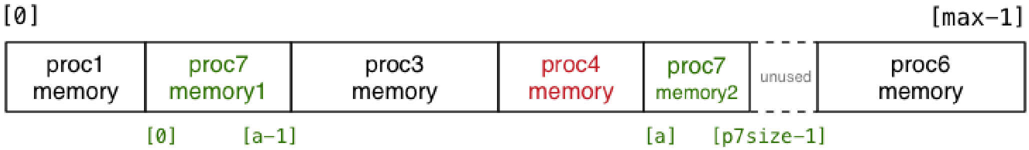


- Slow if RAM heavily used.
- Does not allow sharing or loading on demand.
- Limits process address space to size of RAM.

Idea: split process memory over multiple parts of physical memory.



becomes



Virtual Memory with Multiple Memory Segments Per Process

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***
- translation of addresses can be implemented with an array
- 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***

Virtual Memory with Pages

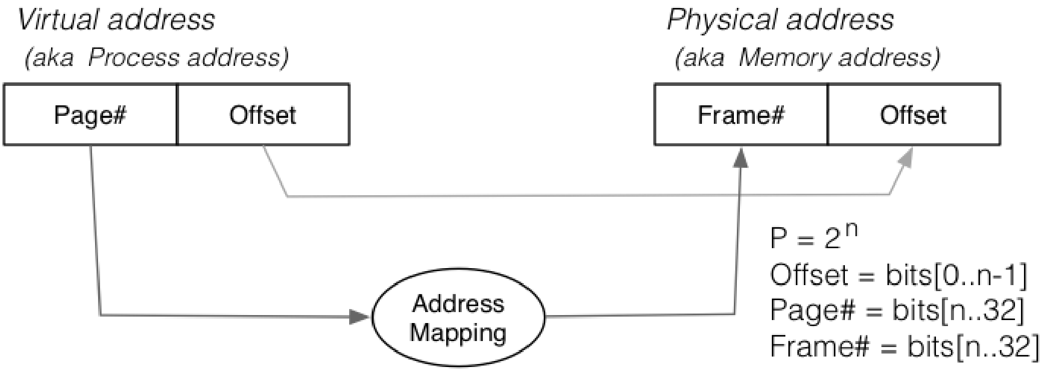
With pages, translating virtual to physical address is simpler making hardware support difficult:

```
// 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
}
```

- Calculation of *page_number* and *offset* can be faster/simpler bit operations if *PAGE_SIZE* $== 2^n$, e.g. 4096, 8192, 16384
- Note **PageInfo** entries will have more information about the page ...

Address Mapping

If $P == 2^n$, then address mapping becomes



Virtual Memory with pages - Lazy Loading

A side-effect of this type of virtual $\rightarrow$ physical address mapping

- don't need to load all of process's pages up-front
- start with a small memory "footprint" (e.g. `main` + stack top)
- load new process address pages into memory *as needed*
- grow up to the size of the (available) physical memory

The strategy of ...

- dividing 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 .. 256KB in size

With 4GB memory, would have $\approx$ 1 million $\times$ 4KB frames

Each frame can hold one page of process address space

Leads to a memory layout like this (with L total pages of physical memory):

When a process completes, all of its frames are released for re-use

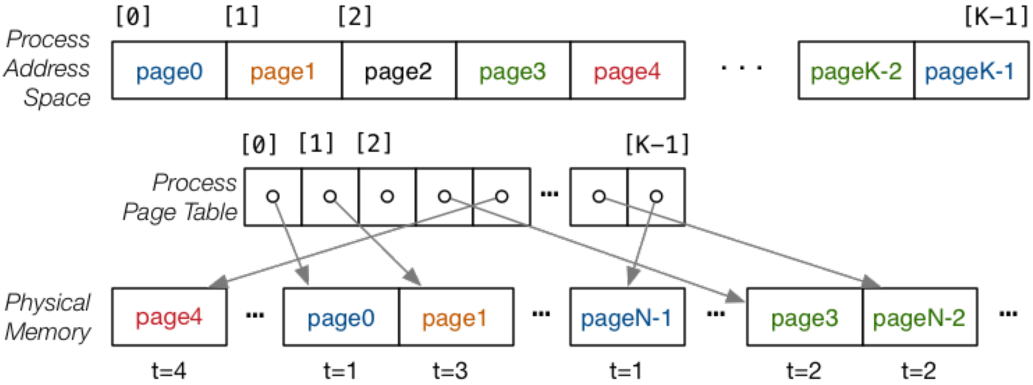
Page Tables

Consider a possible per-process page table, e.g.

- each page table entry (PTE) might contain
  - page status ... *not_loaded*, *loaded*, *modified*
  - frame number of page (if *loaded*)
  - ... maybe others ... (e.g. last accessed time)
- we need $\lceil ProcSize/PageSize \rceil$ entries in this table

Example Page Table

Example of page table for one process:



Timestamps show when page was loaded.

Virtual Memory - Loading Pages

```c
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
}
```

Virtual Memory - Loading Pages

Consider a new process commencing execution ...

- initially has zero pages loaded
- load page containing code for `main()`
- load page for `main()`'s stack frame
- load other pages when process references address within page

Do we ever need to load all process pages at once?

Virtual Memory - Working Sets

From observations of running programs ...

- in any given window of time, process typically access only a small subset of their pages
- often called *locality of reference*
- subset of pages called the *working set*

Implications:

- if each process has a relatively small working set,
  can hold pages for many active processes in memory at same time
- if only need to hold some of process's pages in memory,
  process address space can be larger than physical memory

Virtual Memory - Loading Pages

We say that we ”load” pages into physical memory

But where are they loaded from?

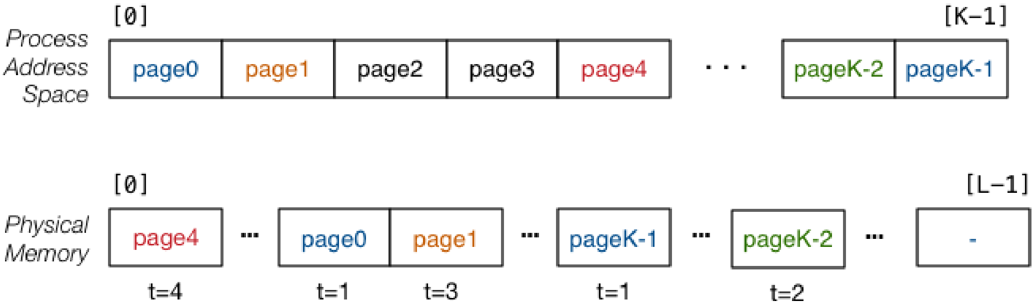
- code is loaded from the executable file stored on disk into read-only pages
- some data (e.g. C strings) also loaded into read-only pages
- initialised data (C global/static variables) also loaded from executable file
- pages for uninitialised data (heap, stack) are zero-ed
  - prevents information leaking from other processes
  - results in uninitialised local (stack) variables often containing 0

Consider a process whose address space exceeds physical memory

Virtual Memory - Loading Pages

We can imagine that a process's address space ...

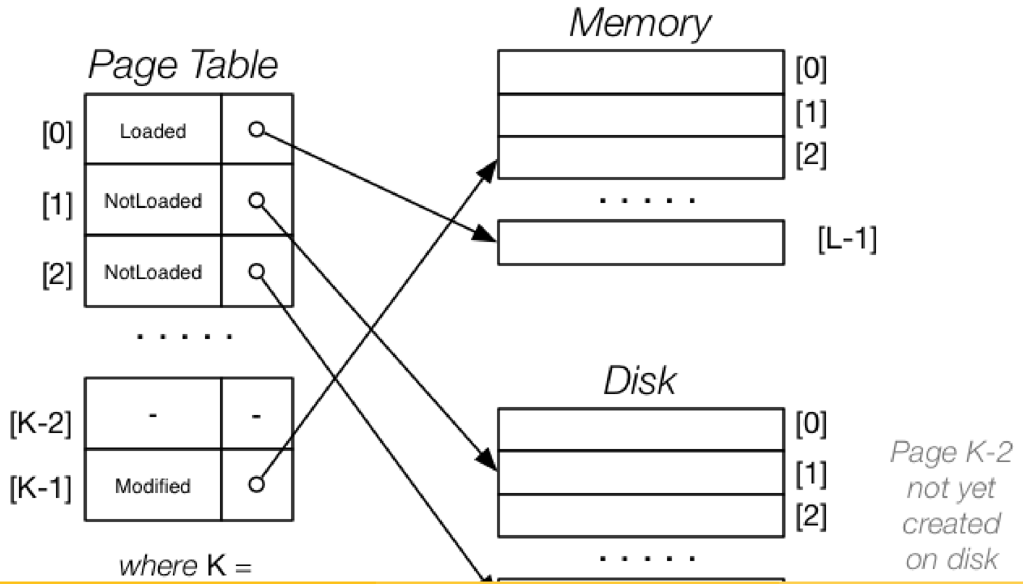
- exists on disk for the duration of the process's execution
- and only some parts of it are in memory at any given time



Transferring pages between disk↔memory is **very** expensive

- need to ensure minimal reading from / writing to disk

Page table with some pages not loaded



Virtual Memory - Handling Page Faults

An access to a page which is not-loaded in RAM is called a ***page fault***.

Where do we load it in RAM?

First need to check for a free frame

- need a way of quickly identifying free frames
- commonly handled via a free list

What if there are currently no free page frames, possibilities:

- *suspend* the requesting process until a page is freed
- *replace* one of the currently loaded/used pages

Suspending requires the operating system 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

Page Replacement

If no free pages we need to choose a page to evict:

- best page is one that won't be used again by its process
- 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 only one process (see later)

OS can't predict whether a page will be required again by its process

But we do know whether it has been used recently (if we record this)

One good heuristic - replace Least Recently Ued (LRU) page.

- page not used recently probably not needed again soon

Exercise: Page Replacement

Show how the page frames and page tables change when

- there are 4 page frames in memory
- the process has 6 pages in its virtual address space
- a LRU page replacement strategy is used

For each of the following sequences of virtual page accesses

```
0, 5, 0, 0, 5, 1, 5, 1, 2, 4, 3, 3, 4, 2, 5, 3, 2
```

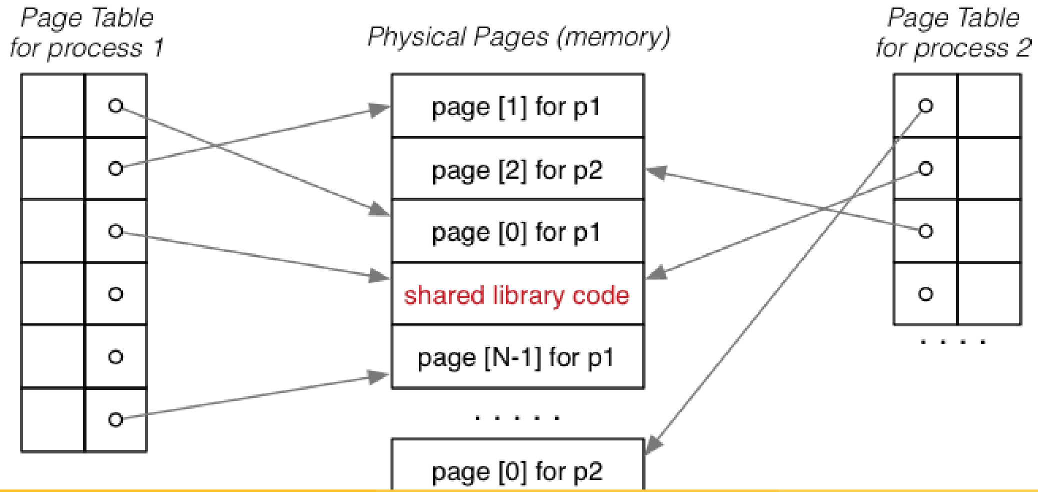
```
5, 0, 0, 0, 5, 1, 1, 5, 1, 5, 2, 2, 3, 0, 0, 5
```

Assume that all PTEs and frames are initially empty/unused

Virtual Memory - Read-only Pages

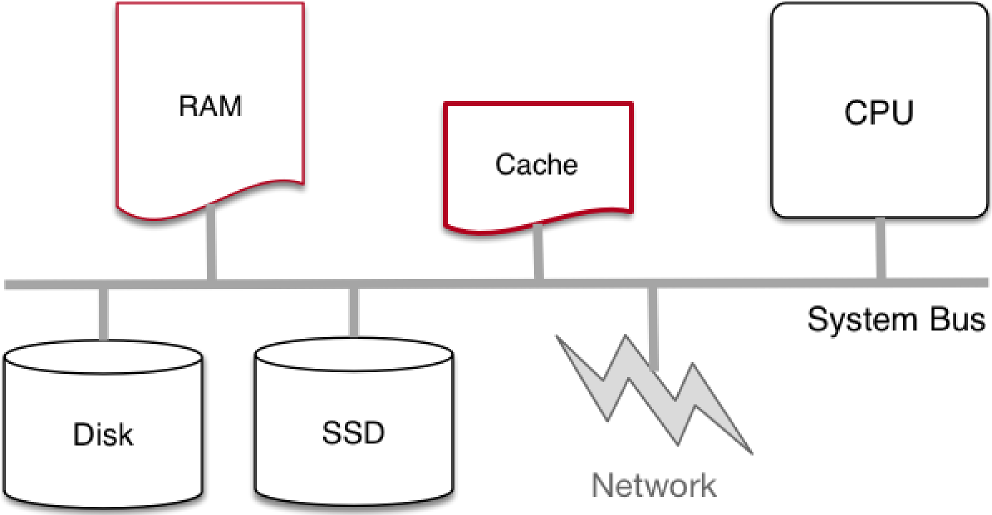
Virtual memory allows sharing of read-only pages (e.g. library code)

- several processes include same frame in virtual address space



Cache Memory

Cache memory = small*, fast memory* close to CPU



Cache Memory

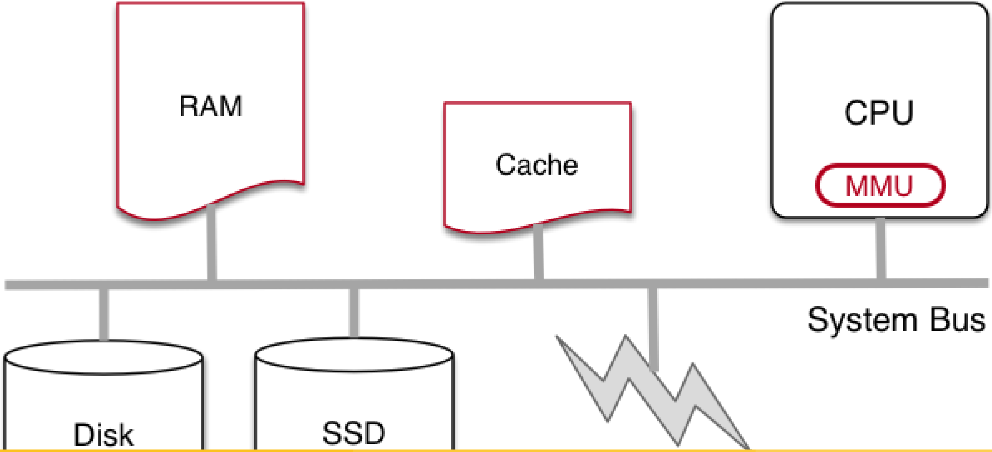
Cache memory

- holds parts of RAM that are (hopefully) heavily used
- 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 in cache
  - possibly replacing an existing cache block
- replacement strategies have similar issues to virtual memory

Memory Management Hardware

Address translation is very important/frequent

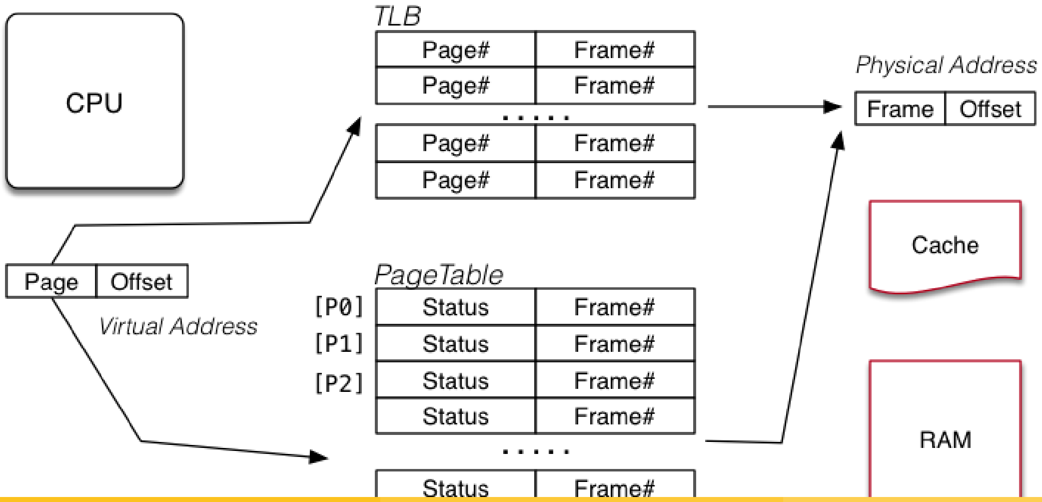
- provide specialised hardware (MMU) to do it efficiently
- sometimes located on CPU chip, sometimes separate



Memory Management Hardware

TLB = translation lookaside buffer

- lookup table containing (virtual,physical) address pairs



COMP1521 21T2 — Concurrency, Parallelism, Threads

https://www.cse.unsw.edu.au/~cs1521/21T2/

Concurrency? Parallelism?

Concurrency:
multiple computations in overlapping time periods …
does *not* have to be simultaneous

Parallelism:
multiple computations executing *simultaneously*

Parallel computations occur at different levels:

- SIMD: Single Instruction, Multiple Data ("vector processing"):
  - multiple cores of a CPU executing (parts of) same instruction
  - e.g., GPUs rendering pixels
- MIMD: Multiple Instruction, Multiple Data ("multiprocessing")
  - multiple cores of a CPU executing different instructions
- distributed: spread across computers
  - e.g., with MapReduce

Both parallelism and concurrency need to deal with *synchronisation*.

Distributed Parallel Computing: Parallelism Across Many Computers

Example: Map-Reduce is a popular programming model for

- manipulating *very large* data sets
- on a large network of computers — local or distributed

The *map* step filters data and distributes it to nodes

- data distributed as $(\text{key}, \text{value})$ pairs
- each node receives a set of pairs with common key

Nodes then perform calculation on received data items.

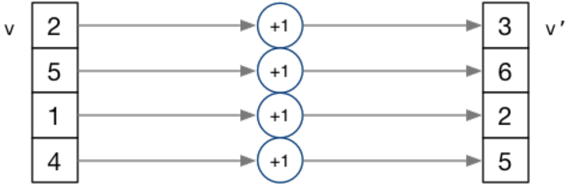
The *reduce* step computes the final result

- by combining outputs (calculation results) from the nodes

(This also needs a way to determine when all calculations completed.)

Data Parallel Computing: Parallelism Across An Array

- multiple, identical processors
- each given one element of a data structure from main memory
- each performing same computation on that element: SIMD
- results copied back to data structure in main memory



But not totally independent: need to *synchronise* on completion

Common use-case for GPUs, neural network processors, etc.

Parallelism Across Processes

One method for creating parallelism:
create multiple processes, each doing part of a 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 is *expensive*
- each require a *significant* amount of state — memory usage
- communication between processes potentially limited and/or slow

But one big advantage:

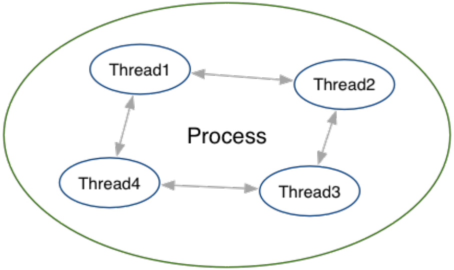
- separate address spaces make processes more robust.

(You're probably using a process-parallel program right now!)

Threads: Parallelism within Processes

Threads allow us parallelism *within* a process.

- Threads allow *simultaneous* execution.
- Each thread has its own execution state (TCB).
- Threads within a process *share* address space:
    - threads share code: functions
    - threads share global/static variables
    - threads share heap: `malloc`
- But a *separate* stack for each thread:
    - local variables *not* shared
- Threads in a process share file descriptors, signals.



Threading with POSIX Threads (pthreads)

POSIX Threads is a widely-supported threading model.
supported in most *nix-like operating systems, and beyond

Describes an API/model for managing threads (and synchronisation).

```
#include <pthread.h>
```

More recently, ISO C:2011 has adopted a pthreads-like model...
less well-supported generally, but very, very similar.

pthread_create(3): create a new thread

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

- Starts a new thread running the specified `thread_main(arg)`.
- Information about newly-created thread stored in `thread`.
- Thread has attributes specified in `attr` (possibly `NULL`).
- Returns 0 if OK, -1 otherwise and sets **`errno`**
- analogous to ***posix_spawn(3)***

pthread_join(3): wait for, and join with, a terminated thread

```
int pthread_join (pthread_t thread, void **retval);
```

- waits until `thread` terminates
  - if `thread` already exited, does not wait
- thread return/exit value placed in `*retval`
- if `main` returns, or *exit(3)* called, *all* threads terminated
  - program typically needs to wait for all threads before exiting
- analogous to ***waitpid(3)***

pthread_exit(3): terminate calling thread

```c
void pthread_exit (void *retval);
```

- terminates the execution of the current thread (and frees its resources)
- `retval` returned — see *pthread_join(3)*
- analagous to ***exit(3)***

Example: two_threads.c — creating two threads (i)

```c
#include <pthread.h>
#include <stdio.h>
// This function is called to start thread execution.
// It can be given any pointer as an argument.
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 either the thread's start function
    // returns, or the thread calls `pthread_exit(3)'.
    // A thread can return a pointer of any type --- that pointer
    // can be fetched via `pthread_join(3)'
    return NULL;
}
```

source code for two_threads.c

Example: two_threads.c — creating two threads (ii)

```c
int main (void)
{
    // Create two threads running the same task, but different inputs.
    pthread_t thread_id1;
    int thread_number1 = 1;
    pthread_create (&thread_id1, NULL, run_thread, &thread_number1);
    pthread_t thread_id2;
    int thread_number2 = 2;
    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;
}
```

source code for two_threads.c

Example: n_threads.c — creating many threads

```
    int n_threads = strtol (argv[1], NULL, 0);
    assert (0 < n_threads && 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;
}
```

source code for n_threads.c

Example: thread_sum.c — dividing a task between threads (i)

```c
struct job {
    long start, 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;
```

source code for thread_sum.c

Example: thread_sum.c — dividing a task between threads (ii)

```
printf (
    "Creating %d threads to sum the first %lu integers\n"
    "Each thread will sum %lu integers\n",
    n_threads, integers_to_sum, 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]);
}
```

source code for thread_sum.c

Example: thread_sum.c — dividing a task between threads (iii)

```c
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);
```

source code for thread_sum.c

`thread_sum.c` performance

Summing the first 1e+10 (10,000,000,000) integers, with N threads, on some different machines...

host	1	2	4	12	24	50	500
ceyx	6.9	3.6	1.8	0.6	0.3	0.3	0.3
lisbon	7.6	3.9	2.0	0.8	0.7	0.7	0.7

ceyx: AMD Ryzen 3900X (12c/24t), 3.8 GHz
lisbon: AMD Ryzen 4750U (8c/16t), 4.1 GHz

Example: two_threads_broken.c — shared mutable state gonna hurt you

```c
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;
    pthread_create (&thread_id2, NULL, run_thread, &thread_number);
    pthread_join (thread_id1, NULL);
    pthread_join (thread_id2, NULL);
    return 0;
}
```

source code for two_threads_broken.c

- variable `thread_number` will probably change in `main`, *before* thread 1 starts executing…
- $\Longrightarrow$ thread 1 will probably print **`Hello this is thread 2`** … ?!

Example: bank_account_broken.c — unsafe access to global variables (i)

```c
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;
}
```

source code for bank_account_broken.c

Example: bank_account_broken.c — unsafe access to global variables (ii)

```c
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;
}
```

source code for bank_account_broken.c

Global Variables and Race Conditions

Incrementing a global variable is not an *atomic* operation.

- (*atomic*, from Greek — “indivisible”)

```c
int bank_account;

void *thread(void *a) {
    // ...
    bank_account++;
    // ...
}
```

```
la   $t0, bank_account
lw   $t1, ($t0)
addi $t1, $t1, 1
sw   $t1, ($t0)
.data
bank_account:  .word 0
```

Global Variables and Race Condition

If, initially, `bank_account = 42`, and two threads increment simultaneously...

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

```
la      $t0, bank_account
# {| bank_account = 42 |}
lw      $t1, ($t0)
# {| $t1 = 42 |}
addi    $t1, $t1, 1
# {| $t1 = 43 |}
sw      $t1, ($t0)
# {| bank_account = 43 |}
```

Oops! We lost an increment.

Threads do not share registers or stack (local variables)...
but they *do* share global variables.

Global Variable: Race Condition

If, initially, `bank_account = 100`, and two threads change it simultaneously...

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, 100
# {| $t1 = 200 |}
sw      $t1, ($t0)
# {| bank_account = ...? |}
```

```
la      $t0, bank_account
# {| bank_account = 100 |}
lw      $t1, ($t0)
# {| $t1 = 100 |}
addi    $t1, $t1, -50
# {| $t1 = 50 |}
sw      $t1, ($t0)
# {| bank_account = 50 or 200 |}
```

This is a *critical section*.

We don't want two processes in the critical section — we must establish *mutual exclusion*.

pthread_mutex_lock(3), *pthread_mutex_unlock(3)*: Mutual Exclusion

```c
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

We associate a resource with a *mutex*.
For a particular mutex, only one thread can be running between `_lock` and `_unlock`.
Other threads attempting to `_lock` will block.
(Other threads attempting to `_trylock` will fail.)

For example:

```c
    pthread_mutex_lock (&bank_account_lock);
    andrews_bank_account += 1000000;
    pthread_mutex_unlock (&bank_account_lock);
```

Example: bank_account_mutex.c — guard a global with a mutex

```c
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;
}
```

source code for bank_account_mutex.c

Semaphores

Semaphores are a more general synchronisation mechanism than 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(3)* initialises `sem` to `value`.
- *sem_wait(3)* — classically **P**
  - if sem > 0, then sem $:=$ sem $- 1$ and continue...
  - otherwise, **wait** until sem > 0
- *sem_post(3)* — classically **V**, also *signal*
  - sem $:=$ sem $+ 1$ and continue...

Example: Allow *n* threads to access a resource

```c
#include <semaphore.h>
sem_t sem;
sem_init (&sem, 0, n);
```

```c
sem_wait (&sem);
// only n threads can be executing here simultaneously
sem_post (&sem);
```

Example: bank_account_sem.c: guard a global with a semaphore (i)

```c
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;
}
```

source code for bank_account_sem.c

Example: bank_account_sem.c: guard a global with a semaphore (ii)

```c
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;
}
```

source code for bank_account_sem.c

Concurrent Programming is Complex

Concurrency is *really 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: bank_account_deadlock.c — deadlock with two resources (i)

```c
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;
}
```

source code for bank_account_deadlock.c

Example: bank_account_deadlock.c — deadlock with two resources (ii)

```
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;
}
```

source code for bank_account_deadlock.c

Example: bank_account_deadlock.c — deadlock with two resources (iii)

```c
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
    // and the other  thread holding  bank_account2_lock
    // and waiting for bank_account1_lock
    pthread_join (thread_id1, NULL);
    pthread_join (thread_id2, NULL);
    return 0;
}
```

source code for bank_account_deadlock.c