10 types of students

$1 \times 2^1 + 0 \times 2^0 = 2$

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

Binary Representation

Computer uses base 2.

- In a similar way, can interpret binary number 1011 as:
  $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10}$
- The *base* or *radix* is 2
  Digits 0 and 1
- Place values:

···	8	4	2	1
···	2^3	2^2	2^1	2^0

1011 % 10
= 1
1 x 2

- Write number as 1011_2
  $(= 11_{10})$

Hexadecimal Representation

- Can interpret hexadecimal number 3AF1 as:
  $3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0$
- The *base* or *radix* is 16
  Digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Place values:

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

- Write number as $3AF1_{16}$
  $(= 15089_{10})$

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:

Bits in Bytes in Words

Values that we normally treat as atomic can be viewed as bits, e.g.

- `char` = 1 byte = 8 bits (a is 01100001)
- `short` = 2 bytes = 16 bits (42 is 0000000000101010)
- `int` = 4 bytes = 32 bits (42 is 0000000000...0000101010)
- `double` = 8 bytes = 64 bits

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

C provides a set of operators that act bit-by-bit on pairs of bytes.

E.g. `(10101010 & 11110000) == 10100000` (bitwise AND)

C bitwise operators: `& | ^ ~ << >>`

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
```

Only 2 '1's with result '1'.

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

Checking for Odd Numbers

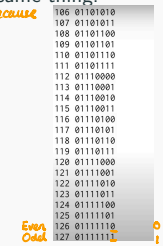
The obvious way to check for odd numbers in C

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

We can use & to achieve the same thing:

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

Because



No special/extra benefits to use this.
Just need to aware this

Even: 0 & 1 → 0 → Not odd.
Odd: 1 & 1 → 1 → Odd.

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
```

Any '1' will result '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
```

Opposite.

Used for e.g. creating useful bit patterns

Bitwise Operations in C

- everything is ultimately a string of bits
- e.g. `unsigned char` = 8-bit value
- e.g. literal bit-string `0b01110001`
- e.g. literal hexadecimal `0x71`
- `&` = bitwise AND
- `|` = bitwise OR
- `~` = bitwise NEG

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
```

Same number ⇒ 0
Different number ⇒ 1.

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
- and 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
- and 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
```

Beware: shifts involving negative values are not portable (implementation defined) - use unsigned values to be safe/portable.

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

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

```c
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
- sometime 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
printf("double      %2lu bytes  min=%-12g  max=%g\n", sizeof d, DBL
printf("long double %2lu bytes  min=%-12Lg  max=%Lg\n", sizeof l, L
```

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

In the lecture, he focus in float = single precision (binary 32). because double can be understood once float is understood.

- 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 x 2^3
- $1.0101011 \times 2^{011} = (1 + 43/128) \times 2^3 = 1.3359375 \times 8 = 10.6875$

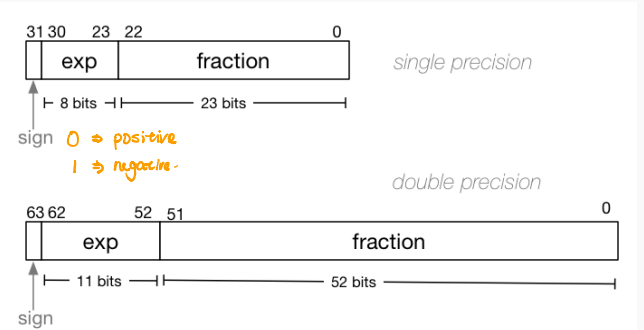
The normalised fraction part always has 1 before the decimal point. Assume to be there

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

why we need to minus.

IEEE-754 Single Precision example: -0.125

```
$ ./explain_float_representation -0.125
-0.125 is represented as a float (IEEE-754 single-precision) by the
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:
11000010111000000010000000000000
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

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

```c
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

```
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

```
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
```

Why Study Assembler?

Useful to know assembly language because ...

- sometimes you are *required* to use it (e.g. device handlers)
- 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

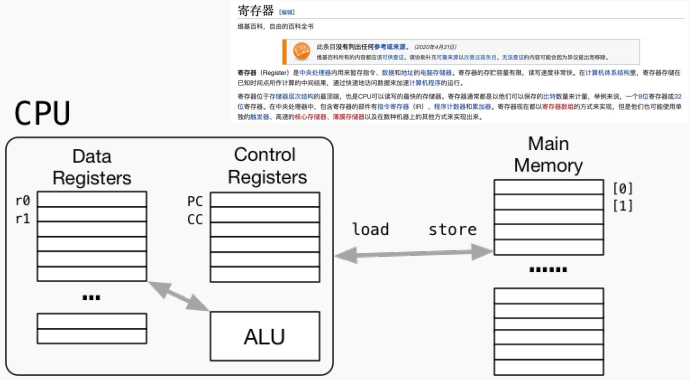
CPU Architecture

A typical modern CPU has

- a set of data registers
- a set of control registers (incl PC)
- an arithmetic-logic unit (ALU)
- access to memory (RAM)
- a set of simple instructions
  - transfer data between memory and registers
  - push values through the ALU to compute results
  - make tests and transfer control of execution

Different types of processors have different configurations of the above

CPU Architecture



Fetch-Execute Cycle

- typical CPU program execution pseudo-code:

```
uint32_t pc = STARTING_ADDRESS;
while (1) {
   uint32_t instruction = memory[pc];
   pc++;  // move to next instr
   if (instruction == HALT) {
      break;
   } else {
      execute(instruction);
   }
}
```

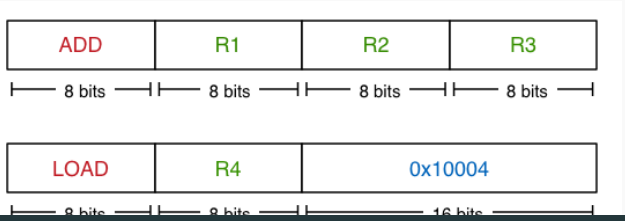
- `pc` = Program Counter, a CPU register which tracks execution
  - note some instructions modify `pc` (branches and jumps)

Fetch-Execute Cycle

Executing an `instruction` involves

- determine what the *operator* is
- determine which *registers*, if any, are involved
- determine which *memory location*, if any, 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, TVs
- but being out-competed by arm (in phones, ...)

We consider the MIPS32 version of the MIPS family

- `qtspim` ... provides a GUI front-end, useful for debugging
- `spim` ... command-line based version, useful for testing
- `xspim` ... GUI front-end, useful for debugging, only in CSE labs

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

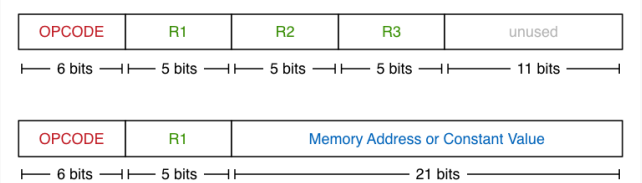
- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register

MIPS Instructions

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 (32-bits on the MIPS)
- Could write machine code program just by specifiying the bit-pattern e.g as a sequence of hex digits:

0x3c041001
0x34020004
0x0000000c
0x03e00008

- unreadable and 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 string
  - refer to registers using either numbers or names
  - allow names (labels) to be 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 in
                    # registers Hi,Lo
seq  $t7,$t1,$t2    # reg[t7] = (reg[t1] == reg[t2])
j    label          # PC = label
beq  $t1,$t2,label  # PC = label if reg[t1]==reg[t2]
nop                 # do nothing
```


MIPS Architecture

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, cannot be written
- 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 conventions.
- Conventions allow compiled code from different sources to be combined (linked).
- Most of these conventions are irrelevant when you are writing small MIPS assembly code programs.
- But use registers 8..25 for holding values.

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

Notes:

- floating point registers can be used as 32 32-bit register or 16 64-bit registers
- for 64-bit use even numbered registers

Data and Addresses

All operations refer to data, either

- in a register
- in memory
- literally (i.e. constant)

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, ...

To refer to literals, use C-like constants:

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


Describing MIPS Assembler 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)} \

- {`Reg[]`{*R*}{`[]`}} = `contents of register` {*R*}.
- *Data transfer is denoted by* {`<-`}.

```
add $6, $7, $8  #  Reg[6] <- Reg[7] + Reg[8]
```

Setting Register

`li` R_d , *imm*	*l*oad *i*mmediate
	Reg[R_d] <- imm
`la` R_d , *addr*	*l*oad *a*ddress
	Reg[R_d] <- addr
`move` R_d , R_s	*move* data reg-to-reg
	Reg[R_d] <- Reg[R_s]

Remember R_d , R_s

↑

Destination.

Setting Register

- The {*li*} *(load immediate) instruction is used to set a register to a constant value, e.g*

```
li $8,  42    # $8 = 42
li $24, 0x2a  # $24 = 42
li $15, '*'   # $15 = 42
```

- The {*move*} *instruction is used to set a register to the same value as another register, e.g*

```
move $8, $9  # assign to $8 value in $9
```

- Note destination is first register.

Setting A Register to An Address

- Note the {*la*} *(load address) instruction is used to set a register to a labelled memory address.*

```
la $8, 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  $7, vec
li  $7, 0x10000100
```

- In both cases the constant is encoded as part of the instruction.
- Neither {*la*} *or {li} access memory - there are very different to the {lw} instruction.*

Pseudo Instructions

- Both *la* and *li* are pseudo instructions provide by the assembler for user convenience.
- The assembler translates these pseudo-instructions into instructions actually implemented by the processor.
- For example, `li` $7, 15 might be translated to addi $7, $0, 15
- If the constant is large the assembler will need to need translate a li/la instruction to two actual instructions.

Arithmetic Instructions

add R_d, R_s, R_t	*add*ition
	Reg[R_d] <- Reg[R_s]+Reg[R_t]
add R_d, R_s, *imm*	*add*ition
	Reg[R_d] <- Reg[R_s]+imm
sub R_d, R_s, R_t	*sub*traction
	Reg[R_d] <- Reg[R_s]-Reg[R_t]
mul R_d, R_s, R_t	*mul*tiplication
	Reg[R_d] <- Reg[R_s] * Reg[R_t]
div R_d, R_s, R_t	*div*ision
	Reg[R_d] <- Reg[R_s] / Reg[R_t]
rem R_d, R_s, R_t	*rem*ainder
	Reg[R_d] <- Reg[R_s] % Reg[R_t]
neg R_d, R_s	*neg*ate
	Reg[R_d] <- - Reg[R_s]

All arithmetic is signed (2's-complement).

Logic Instructions

and R_d, R_s, R_t	logical *and*
	Reg[R_d] <- Reg[R_s] & Reg[R_t]
and R_d, R_s, *imm*	logical *and*
	Reg[R_d] <- Reg[R_s] & imm
or R_d, R_s, R_t	logical *or*
	Reg[R_d] <- Reg[R_s] | Reg[R_t]
not R_d, R_s	logical *not*
	Reg[R_d] <- Reg[R_s]
xor R_d, R_s, R_t	logical *xor*
	Reg[R_d] <- Reg[R_s] ^ Reg[R_t]

All of these instructions can use *imm* instead of R_t.

Bit Manipulation Instructions

sll R_d, R_s, R_t	*s*hift *l*eft *l*ogical
	Reg[R_d] <- Reg[R_s]<<Reg[R_t]
sll R_d, R_s, *imm*	*s*hift *l*eft *l*ogical
	Reg[R_d] <- Reg[R_s]<< imm
srl R_d, R_s, *imm*	*s*hift *r*ight *l*ogical
	Reg[R_d] <- Reg[R_s]>> imm
sra R_d, R_s, *imm*	*s*hift *r*ight *a*rithmetic
	Reg[R_d] <- Reg[R_s]>> imm
rol R_d, R_s, *imm*	*r*otate *l*eft
	Reg[R_d] <- rot(Reg[R_s] imm left)
ror R_d, R_s, *imm*	*r*otate *r*ight
	Reg[R_d] <- rot(Reg[R_s] imm right)

All of these instructions can use R_t instead of *imm*.

Jump Instructions

Jumps control flow of program execution.

j *label* jump to location
PC <- & label
jal *label* jump and link

jr R_s jump via register
PC <- Reg[R_s]
jalr R_s jump and link via reg

Branch Instructions

Branches combine testing and jumping.

beq R_s, R_t, *label*	branch on equal if (Reg[R_s]==Reg[R_t]) PC <- label
bne R_s, R_t, *label*	branch on not equal if (Reg[R_s]!=Reg[R_t]) PC <- label
blt R_s, R_t, *label*	branch on less than if (Reg[R_s]<Reg[R_t]) PC <- label
ble R_s, R_t, *label*	branch on less or equal if (Reg[R_s]<=Reg[R_t]) PC <- label
bgt R_s, R_t, *label*	branch on greater than if (Reg[R_s]>Reg[R_t]) PC <- label
bge R_s, R_t, *label*	branch on greater or equal if (Reg[R_s]>=Reg[R_t]) PC <- label

MIPS Instruction Set

Implementation of pseudo-instructions:

```
What you write          Machine code produced
----------------------  ----------------------

li   $t5, const         ori  $t5, $0, const

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

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

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

Note: use of $at register for intermediate results

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 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,$0,$v0`

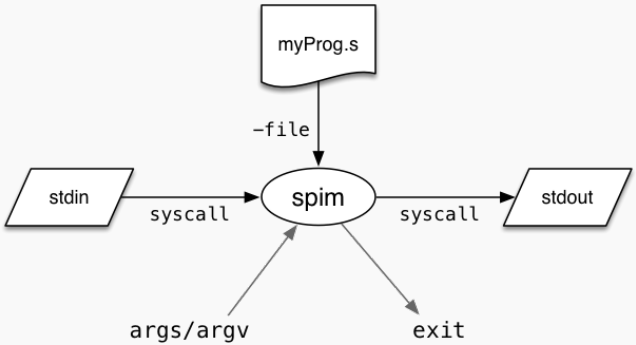
Using SPIM

Three ways to execute MIPS code with SPIM

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

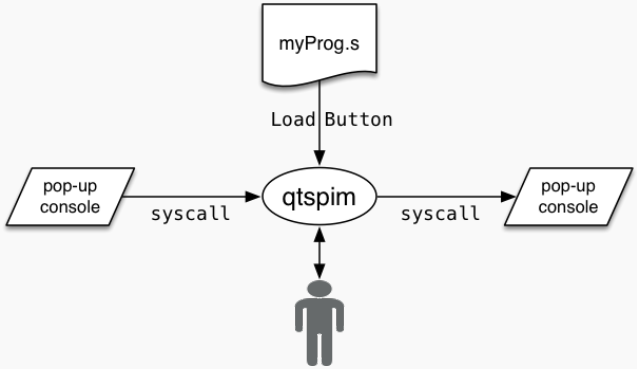
Using SPIM

Command-line tool:



Using SPIM

GUI tool:



Using Spim Interactively

```
$ 1521 spim
Loaded: /home/cs1521/share/spim/exceptions.s
(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

The SPIM interpreter provides I/O and memory allocation via the syscall instruction.

Service	n	Arguments	Result
printf("%d")	1	int in $a0	-
printf("%f")	2	float in $f12	-
printf("%lf")	3	double in $f12	-
printf("%s")	4	$a0 = string	-
scanf("%d")	5	-	int in $v0
scanf("%f")	6	-	float in $f0
scanf("%lf")	7	-	double in $f0
fgets	8	buffer address in $a0	
		length in $a1	-
sbrk	9	nbytes in $a0	address in $v0
printf("%c")	11	char in $a0	-
scanf("%c")	12	-	char in $v0
exit(status)	17	status in $a0	-

MIPS (SPIM) memory layout

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

MIPS Assembly Language

MIPS assembly language programs contain

- comments ... introduced by #
- labels ... appended with :
- directives ... symbol beginning with .
- assembly language instructions

Programmers need to specify

- data objects that live in the data region
- functions (instruction sequences) that live in the code/text region

Each instruction or directive appears on its own line

Example MIPS assembler program

```
# hello.s ... print "Hello, MIPS"

main:
    la $a0, msg   # load the argument string
    li $v0, 4     # load the system call (print)
    syscall       # print the string
    jr $ra        # return to caller (__start)

    .data         # the data segment
msg: .asciiz "Hello, MIPS\n"
```

Structure of Simple MIPS programs

```
# Prog.s ... comment giving description of function
# Author ...

main:          # indicates start of code
               # (i.e. first user instruction to execute)
               # ...

     .data     # variable declarations follow this line
               # ...

# End of program; leave a blank line to make SPIM happy
```


Assembler Directives

Directives {(instructions to assembler, not MIPS instructions)}

```
    .text       # following instructions placed in text
    .data       # following objects placed in data

    .globl      # make symbol available globally

a:  .space 18   # uchar a[18];  or  uint a[4];
    .align 2    # align next object on 2-byte addr 2^2 = 4
                ↖ the power of.
i:  .word 2     # unsigned int i = 2;
v:  .word 1,3,5 # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6 # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3 # unsigned char b[3] = {1,2,3};
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'};
```


Encoding MIPS Instructions as 32 bit Numbers

Assembler	Encoding
$ 7 = $ 8 + $ 0	
add $ d, $ s, $ t	000000 sssss ttttt ddddd00000100000
add $ 7, $ 8, $ 0	000000 00111 01000 0000000000100000
	0x01e80020 == 31981600
$ 5 = $ 1 - $ 3	
sub $ d, $ s, $ t	000000 sssss ttttt ddddd00000100010
sub $ 5, $ 1, $ 3	000000 00001 00011 0010100000100010
	0x00232822 == 2304034
$ 2 = $ 2 + 1	
addi $ d, $ s, C	001000 sssss ddddd CCCCCCCCCCCCCCCC
addi $ 2, $ 2, 1	001000 00010 00010 0000000000000001
	0x20420001 == 541196289

All instructions are variants of 3 patterns of bits.

Which simplifies silicon and human decoding (ass1!).

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 poor 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 the solution in C
- map to "simplified" C
- translate each simplified C statement 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

Example:

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 control structures by goto

add: C to simplified C

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

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


add: simplified C to MIPS

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

```
main:
    li   $t0, 17
    li   $t1, 25
    add  $t2, $t1, $t0

    move $a0, $t2
    li   $v0, 1
    syscall

    li   $a0, '\n'
    li   $v0, 11
    syscall

    jr   $ra
```


while: C to simplified C

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

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


while: simplified C to MIPS

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

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

if: C to simplified C

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

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

Note: you can't use else as a {label} in C}

if: simplified C to MIPS

```c
if (i >= 0) 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
  sub $t1, $t1, $t0
  goto end1
else1:
  add $t1, $t1, $t0
end1:
```


if/and: C to simplified C

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

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


if/and: simplified C to MIPS

```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
  goto end1
else1:
  add $t1, $t1, $t0
end1:
```

odd-even: C to simplified C

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

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


Example Printing First 10 Integers

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


Example Printing First 10 Integers

Convert to goto and simple C statements and decide where variables will be stored.

```c
int main(void) {
    int i;    // in register $t0
    i = 0;
loop:
    if (i >= 10)
        goto end;
    i++;
    printf("%d", i);
    printf("%c", '\n');
    goto loop;
end:
    return 0;
}
```


Example Printing First 10 Integers

```
main:                   # int main(void) {
                        # int i; // in register $t0
    li    $t0, 0        # i = 0;
loop:                   # loop:
    bge  $t0, 10 end    # if (i >= 10) goto end;
    add  $t0, $t0 1     #   i++;
    move $a0, $t0       #   printf("%d" i);
    li   $v0, 1
    syscall
    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall
    b    loop           # goto loop;
end:
    jr   $ra            # return
```

Assembler Directives

Directives al(instructions to assembler, not MIPS instructions)}

```
    .text       # following instructions placed in text
    .data       # following objects placed in data
    .globl      # make symbol available globally

a:  .space 18   # uchar a[18];  or  uint a[4];
    .align 2    # align next object on 2-byte addr

i:  .word 2     # unsigned int i = 2;
v:  .word 1,3,5 # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6 # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3 # unsigned char b[3] = {1,2,3};
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'};
```


SPIM memory layout

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

Setting A Register to An Address

- Note the `la` (load address) instruction is used to set a register to a labelled memory address.

  ```
  la $8, 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  $7, vec
  li  $7, 0x10000100
  ```

- In both cases the constant is encoded as part of the instruction(s).
- Neither `la` or `li` access memory - there are very different to the `lw` instruction.

Accessing Memory

- only load/store instructions move data between memory and CPU.
- 1, 2 and 4-bytes (8, 16 and 32 bit) quantities can be moved.
  - two bytes ("halfword"), four bytes ("word")
- Two operands: the register which supplies/receives the value and the memory address.
- for 1 and 2-byte operations the low (least significant) bits of the register are used.
- LB`` &LH assume bytes/halfword contain a 8-bit/16-bit signed integer
- unsigned equivalents LHU and LBU assume integr is unsigned

Accessing Memory

assembly	meaning	bit pattern
`lb` r_t, I(r_b)	r_t = `mem[`r_b+I`]`	100000bbbbbttttt0000000000000000
`lh` r_t, I(r_b)	r_t = `mem[`r_b+I`]` |	100001bbbbbttttt0000000000000000
	`mem[`r_b+I+1`]` << 8	
`lw` r_t, I(r_b)	r_t = `mem[`r_b+I`]` |	100011bbbbbttttt0000000000000000
	`mem[`r_b+I+1`]` << 8 |	
	`mem[`r_b+I+2`]` << 16 |	
	`mem[`r_b+I+3`]` << 24	
`sb` r_t, I(r_b)	`mem[`r_b+I`]` = r_t & 0xff	101000bbbbbttttt0000000000000000
`sh` r_t, I(r_b)	`mem[`r_b+I`]` = r_t & 0xff	101001bbbbbttttt0000000000000000
	`mem[`r_b+I+1`]` = r_t >> 8 & 0xff	
`sw` r_t, I(r_b)	`mem[`r_b+I`]` = r_t & 0xff	101011bbbbbttttt0000000000000000
	`mem[`r_b+I+1`]` = r_t >> 8 & 0xff	
	`mem[`r_b+I+2`]` = r_t >> 16 & 0xff	
	`mem[`r_b+I+3`]` = r_t >> 24 & 0xff	

Memory Addressing Modes

Ways of specifying memory addresses:

Format	Address referred to
(reg)	contents of register e.g. ($8) ($fp) ($5)
imm	immediate (= constant) e.g. 0 0x80000000 123456789
imm(reg)	immediate + contents of register e.g. 0($fp) 0x80000000($9)
sym	address of symbol (= name) e.g. main endloop exit
sym(reg)	address of symbol + reg contents e.g. main($fp) array($9)
sym +/- imm	address of symbol +/- immediate e.g. main+8 endloop-4
sym +/- imm *(reg)*	sym address +/- (imm + reg contents) e.g. main+8($8) array+0($18)

Addressing Modes

Examples of load/store and addressing:

```
main:
    la  $t0, vec       # reg[t0] = &vec[0]
    li  $t1, 5         # reg[t1] = 5
    sw  $t1, ($t0)     # vec[0] = reg[t1]
    li  $t1, 13        # reg[t1] = 13
    sw  $t1, 4($t0)    # vec[1] = reg[t1]
    li  $t1, -7        # reg[t1] = -7
    sw  $t1, 8($t0)    # vec[2] = reg[t1]
    li  $t2, 12        # reg[t2] = 12
    li  $t1, 42        # reg[t1] = 42
    sw  $t1, vec($t2)  # vec[3] = reg[t1]
    jr   $ra           # return
    .data              # 16 bytes of storage
vec: .space   16       # int vec[4];
```


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

Global/Static Variables

- global/static variables need 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
```

initialized to 0 by default, other directives allow initialization 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

```
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

C

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

MIPS

```
main:
    li   $t0, 17
    sw   $t0, x
    li   $t0, 25
    sw   $t0, y
    lw   $t0, x
    lw   $t1, y
    add  $t2, $t1, $t0
    sw   $t2, z
.data
x:  .space 4
y:  .space 4
z:  .space 4
```


store value in array element - v1

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   $t3, ($t2)
.data
x:  .space 40
```

store value in array element - v2

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
    sw   $t3, ($t2)
.data
x:  .space 60
```


1-d Arrays in MIPS

Can be named/initialised as noted above:

```
vec:  .space 40
# could be either int vec[10] or char vec[40]

nums: .word 1, 3, 5, 7, 9
# int nums[6] = {1,3,5,7,9}
```

Can access elements via index or cursor (pointer)

- either approach needs to account for size of elements

Arrays passed to functions via pointer to first element

- must also pass array size, since not available elsewhere

{See `sumOf()` exercise for an example of passing an array to a function }

Printing 1-d Arrays in MIPS - v1

C

```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
vec: .word 0,1,2,3,4
```


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   $a0, ($s0)
   li   $v0, 1
   syscall
   addi $s0, $s0, 4
   b    loop
end:
   .data
vec: .word 0,1,2,3,4
```

1-d Arrays in MIPS

Scanning across an array of `N` elements using cursor

```
# int vec[10] = {...};
# int *cur, *end = &vec[10];
# for (cur = vec; cur < end; cur++)
#    printf("%d\n", *cur);}}
```

```
   la   $s0, vec              # cur = &vec[0]
   la   $s1, vec+40           # end = &vec[10]
loop:
   bge  $s0, $s1, end_loop    # if (cur >= end) break
   lw   $a0, ($s0)            # a0 = *cur
   jal  print                 # print a0
   addi $s0, $s0, 4           # cur++
   j    loop
end_loop:
```

{Assumes the existence of a `print()` function to do `printf("\%d n",x)`}

2-d Arrays in MIPS

Representations of `int matrix[4][4]` ...

```
matrix: .space 64
```

Now consider summing all elements

```
int i, j, sum = 0;
for (i = 0; i < 4; i++) {
   for (j = 0; j < 4; j++) {
      sum += matrix[i][j];
   }
}
```


2-d Arrays in MIPS

Computing sum of all elements in `int matrix[6][5]` in C

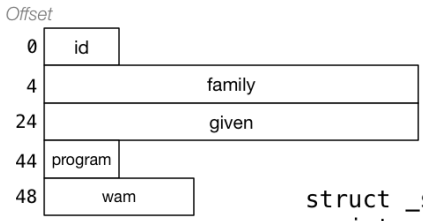
```
int row, col, sum = 0;

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


2-d Arrays in MIPS

Computing sum of all elements `int matrix[6][5]`

```
   li  $s0, 0              # sum = 0
   li  $s1, 6              # s1 = #rows
   li  $s2, 0              # row = 0
   li  $s3, 5              # s3 = #cols
   li  $s4, 0              # col = 0 // redundant
   li  $s5, 4              # intsize = sizeof(int)
   mul $s6, $s3, $s5       # rowsize = #cols*intsize
loop1:
   bge  $s2, $s1, end1     # if (row >= 6) break
   li   $s4, 0             # col = 0
loop2:
   bge  $s4, $s3, end2     # if (col >= 5) break
   mul  $t0, $s2, $s6      # t0 = row*rowsize
   mul  $t1, $s4, $s5      # t1 = col*intsize
   add  $t0, $t0, $t1      # offset = t0+t1
   lw   $t0, matrix($t0)   # t0 = *(matrix+offset)
```

Structs in MIPS




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


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;
```


Structs in MIPS

Accessing structure components is by offset, not name

```
stu1: .space 56         # student_t stu1;
stu2: .space 56         # student_t stu2;
# stu is $s1            # student_t *stu;

li  $t0  5012345
sw  $t0, stu1+0         # stu1.id = 5012345;
li  $t0, 3778
sw  $t0, stu1+44        # 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;
```

MIPS Functions

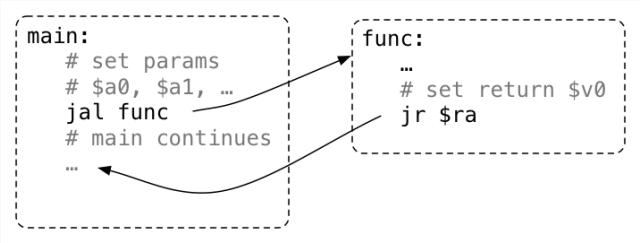
When we call a function:

- 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

Function Calls

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 function parameters passed in **$a0 $a1 $a2 $a3**
- if there are more 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

```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

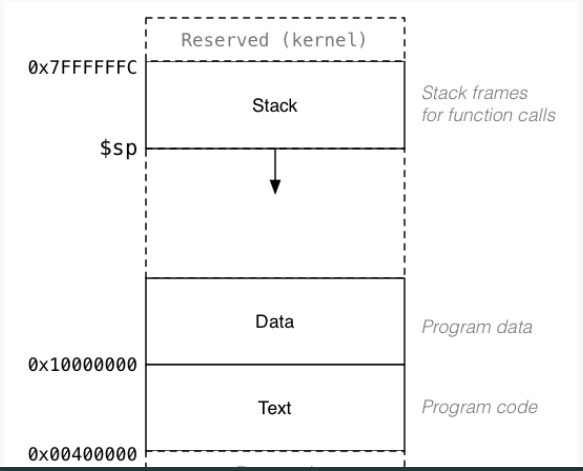
Simple Function Call Example - broken MIPS

```
    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 call
    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

Stack - Where it is in Memory

Data associated with a function call placed on the stack:



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
- 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`)

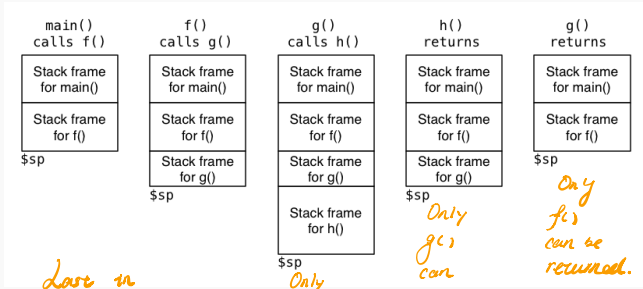
```
add  $sp, $sp, 40    # move stack pointer back
```


Stack - Using stack to Save/Restore 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
```


Stack - Growing & Shrinking

How stack changes as functions are called and return:



Function calling another function - How to Do It

- 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

Can I use counter?

- *a*0..**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 u
    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
    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
    jr   $ra            # return from two
```


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
    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
    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
```

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 whe
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

```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 prioirity

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

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 kernal
- 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

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.

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

```c
// 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

Using open system call to copy a file

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

Unix-like Files

- 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/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

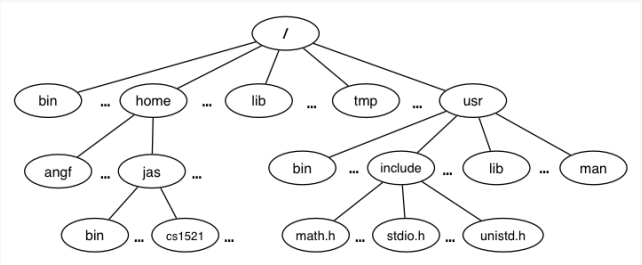
  e.g. `/usr/include/stdio.h`, `/cs1521/public_html/`
- every process (running process) has an associated *absolute* pathname called the *current working directory* (CWD)
- shell command `pwd` prints CWD
- *relative* pathname do not start with a leading `/` e.g. `../../another/path/prog.c`, `./a.out`, `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/../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
  - but only if you follow symbolic links

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-numbe*r, 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

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
```


File Operations: Overview

Unix presents a uniform interface to file system objects

- functions/syscalls 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 common operations:

- **`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

libc Types for File System Operations

Unix defines a range of file-system-related types:

- **`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`, ...

open - libc

```
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`**

close - libc

`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

read

`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`

write - libc

`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 libc

```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 fi
//   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

lseek

`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("The 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("The 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("The 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("The 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("The 100th byte of the file is 0x%02x\n", bytes[0]);
```

source code for lseek.c

stat

```
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

```
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 */
};
```


ist_mode 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

```
struct stat s;
if (stat(pathname, &s) != 0) {
    perror(pathname);
    exit(1);
}
printf("s.st_ino =  %10ld # Inode number\n", s.st_ino);
printf("s.st_mode = %10o # File mode \n", s.st_mode);
printf("s.st_nlink =%10ld # Link count \n", (long)s.st_nlink);
printf("s.st_uid =  %10u # Owner uid\n", s.st_uid);
printf("s.st_gid =  %10u # Group gid\n", s.st_gid);
printf("s.st_size = %10ld # File size (bytes)\n", (long)s.st_size);
printf("s.st_mtime =%10ld # Modification time (seconds since 1/1/7C
```

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

Example:

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


Other useful Linux (POSIX) functions

```
chdir(char *path)  // change current working directory

getcwd(char *buf, size_t size) // get current working directory

rename(char *oldpath, char *newpath) // rename a file/directory

link(char *oldpath, char *newpath) // create hard link to a file

symlink(char *target, char *linkpath) // create a symbolic link

unlink(char *pathname) // remove a file/directory/...

chmod(char *pathname, mode_t mode) // change permission of file/...
```


stdio.h

`stdio.h` functions more portable more convenient than open/read/write/... use them by default

- `stdio.h` equivalent to open is **fopen**

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

- **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

`int fclose(FILE *stream)`

- `stdio.h` equivalent to `close`

stdio - read and writing

```
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
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
// write array of bytes
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream)
```


stdio.h - convenience functions for stdin/stdout

As we often read/write to stdin/stdout `stdio.h` provides convenience functions which only read/write stdin/stdout

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

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

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

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


stdio.h - other operations on

```
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** relavtive to file start, current position or file end

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

- flush any buffered data on writing stream

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

- equivalent to `close`

@hello_fputc.c @hello_fputs.c @hello_fwrite.c @hello_fprintf.c @cat_fgetc.c @cat_fgets.c @cat_fwrite.c @cp_fgetc.c @cp_fwrite.c @create_file_fopen.c @create_append_truncate_fopen.c

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**

@myio_unbuffered.c @myio_input_buffered.c @myio_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

```
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

Converting Unicode Codepoints to UTF-8

```
    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.

virtual_memory # 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 (pid)
- positive integer - type `pid_t` defined in `unistd.h`
- process 0 is effectively part of the operating system
- process 1 (***init***) - used to boot the system
- some parts of operating system may run as processes
- low-numbered processes are typically system-related process started at boot-time

Process Parents

Each process has a *parent process*

- initially it is the process that created it
- if a process' parent terminates, its parent becomes process 1

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

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

Aside: Zom

Zombie Proces

Photo credit: Kenny Louie, Flickr.com

Aside: zombie Processes

- a process can't 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 is assigned pid=1 (init) as its parent
  - init should always handles `SIGCHLD` when process exits

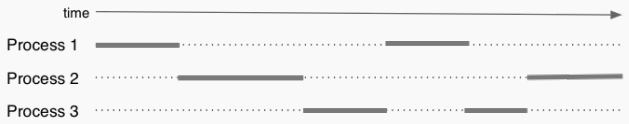
Multi-Tasking

On a typical modern operating system

- multiple processes are active "simultaneously" (*multi-tasking*)
- operating systems provides a virtual machine to each processs
  - 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

- each process uses the CPU until *pre-empted* or exits
- then another process uses the CPU until it too is pre-empted
- eventually, the first process will get another run on the CPU



Overall impression: three programs running simultaneously

Processes

What can cause a process to be pre-empted?

- it runs "long enough" and the OS replaces it by a waiting process
- it needs to wait for input, or output or . . .

On pre-emption ..

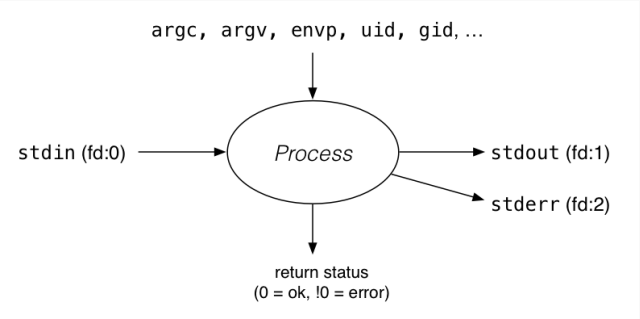
- the process's entire state must be saved
- the new process's state must be restored
- this change is called a ***context switch***
- ***context switches*** are expensive

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

Unix/Linux Processes

Environment for processes running on Unix/Linux systems



Process-related Unix/Linux Functions/System Calls

- `posix_spawn()` ... create a new process, see also
  - `clone()` ... duplicate current process
    address space can be shared to implement threads
    only use clone if posix_spawn can't do what you want
- `fork()` ... duplicate current process - do not use in new code
- `execvp()` ... replace current process
- `system()` `popen()` ... create a new process via a shell (unsafe)
- `exit()` ... terminate current process, see also
  - `_exit()` ... terminate current process immediately
    atexit functions not be called: stdio buffers not flushed
- `getpid()` ... get process ID
- `getpgid()` ... get process group ID
- `waitpid()` ... wait for state change in child process

posix_spawn() - run a new process

```
#include <spawn.h>

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

- creates new process, running program at **path**
- **argv** specifies argv of new program
- **envp** specifies environment of new program
- ***pid** set to process id of new program
- **file_actions** specifies file actions to be performed before running program
  - can be used to re-direct stdin or stdout to file or pipe
  - advanced topic
- **attrp** specifies attributes for new process

Simple 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)
    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

```
#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*
- child has a different process ID (pid) to the parent
- in the child, `fork()` returns 0
- in the parent, `fork()` returns the pid of the child
- if the system call fails, `fork()` returns -1
- child inherits copies of parent's address space and open file descriptors
- do not use in new code use **posix_spawn** instead
  - **fork** appears simple but prone to subtle bugs

Simple example of using fork()

```
// 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[]);
```

- relaces current process by executing `file`
  - `file` must be an executable: binary or script starting with `#!`
- `argv` specifies argv of new program
- most of the current process is reset
  - e.g. new virtual address space is created, signal handlers reset
- new process inherits open file descriptors from original process
- on error, returns -1 and sets `errno`
- if successful, does not return

Simple example of using exec()

```
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
$
```


Simple example using fork() and exec() to run /bin/date

```
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

```
#include <stdlib.h>
int system(const char *command);
```

- runs `command` via `/bin/sh`
- waits for `command` to finish and returns exit status
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only

```
// 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

running ls -ld via posix_spawn

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

source code for lsld_spawn.c

running ls -ld via posix_spawn

```
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid");
    exit(1);
}
// exit with whatever status ls exited with
return exit_status;
```

source code for lsld_spawn.c

running ls -ld via system

```
char *ls = "/bin/ls -ld";
int command_length = strlen(ls);
for (int i = 1; i < argc; i++) {
    command_length += strlen(argv[i]) + 1;
}
// create command as string
char command[command_length + 1];
strcpy(command, ls);
for (int i = 1; i <= argc; i++) {
    strcat(command, " ");
    strcat(command, argv[i]);
}
int exit_status = system(command);
```

source code for lsld_system.c

getpid & getppid

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

pid_t getpid(void);
pid_t getppid(void);
```

- **getpid** returns the process ID of the current process
- **getppid** returns process ID of the the parent of current process

waitpid

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

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

- **waitpid** pauses 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 the specified process

```
pid_t wait(int *status)
```

- equivalent to `waitpid(-1, &status, 0)`
- pauses until one of the child processes terminates

waitpid

More on `waitpid(pid, &status, 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`

linux/environment variables

- when linux/unix program are passed ***environment variables***
- ***environment variables*** are array of strings of form `name=value`
- array is NULL-terminated
- access via global variable **environ**
- many C implementation also provide as 3rd parameter to main:

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

- most program use **getenv** & **setenv** to access environment variables
- can access environment variables directly, eg:

```
// 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

accessing an environment variable with getenv

```
#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

setting an environment variables with setenv

```c
#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

```c
// 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

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)

Also **`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 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 (and other) file descriptors can be used with stdio via **fdopen**

popen() - convenient but unsafe way to set up pipe

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- runs **command** via **/bin/sh**
- if **type** is "w" pipe to stdin of **command** created
- if **type** is "r" pipe from stdout of **command** created
- **FILE *** stream returned - get then use **fgetc**/**fputc** etc
- **NULL** returned if error
- close stream with **pclose** (not **fclose**)
  - **pclose** waits for **command** and returns exit status
- convenient but brittle and highly vulnerable to security exploits
- use for quick debugging and throw-away programs only

popen() - capturing output from a process

```
// 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 or
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

popen() - sending input to a process

```
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 progran
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to
    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 & 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

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 `SIGKILL` and `SIGSTOP` 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.

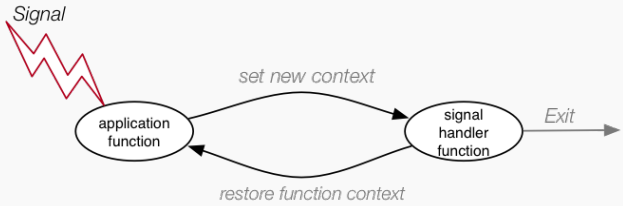
- `SIGILL` ... illegal instruction (***Term*** by default)
- `SIGABRT` ... generated by `abort()` (***Core*** by default)
- `SIGFPE` ... floating point exception (***Core*** by default)
- `SIGSEGV` ... invalid memory reference (***Core*** by default)

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



Installing A Signal Handler - the Old Way

```
#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

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` ...

- `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)

Signal Handlers

Details on `siginfo_t` ...

- `si_signo` ... signal being handled
- `si_errno` ... any errno value associated with signal
- `si_pid` ... process ID of sending process
- `si_uid` ... user ID of owner of sending process
- `si_status` ... exit value for process termination
- etc. etc. etc.

For more details: `bits/types/siginfo_t.h` (system-dependent)

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(), SIG
    // 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

Waiting For an Event - The Smart 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(), SIG
    // suspend execution for 1 hour
    sleep(3600);
}
```

source code for wait_for_signal.c

sending signals with kill()

```c
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

```c
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

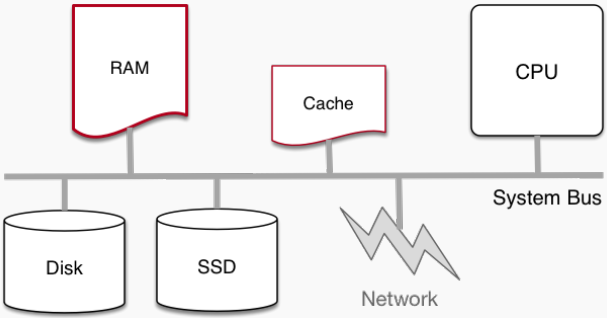
example - catching an internal error with a signal handler

```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

Memory

Systems typically contain 4-16GB of volatile RAM



Plus a hierarchy of smaller cache memory -on or off the CPU chip.

Single Process Resident in RAM without Operating System

- Many small embedded 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

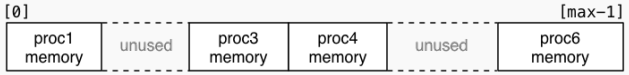
- If multiple processes to be resident in RAM O/S can swap execution between them quickly.
- RAM belonging to other processes & kernel must be protected
- Hardware support can limit process accesses to particular **segment** (region) of RAM.
- BUT program may be loaded anywhere in RAM to run
- Breaks 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

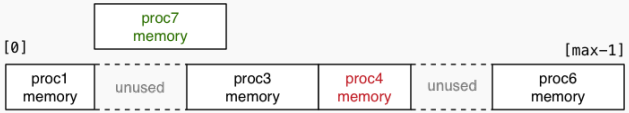
Consider a scenario with multiple processes loaded in memory:



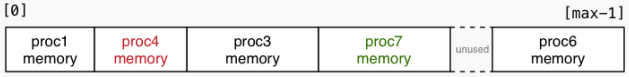
- Every process is in a contiguous section of RAM, starting at address **base** finishing at address **limit**.
- Each process sees its own address space as [0 .. size - 1]
- Process can be loaded anywhere in memory without change.
- Process accessing memory address **a** is translated to **a + base**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



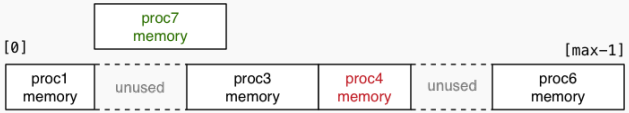
- The new process doesn't fit in any of the unused slots (fragmentation).
- Could move some process to make a single large slot



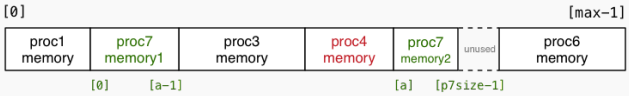
- Slow if RAM heavily used.
- Does not allow sharing or loading on demand.
- Limits process address space to size of RAM.
- Little used in modern computing.

Virtual Memory with Multiple Memory Segments Per Process

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

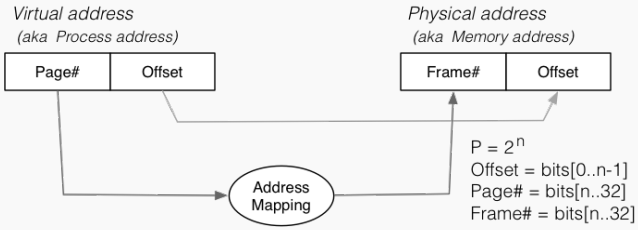
With pages, translating virtual to physical address is simpler making hardware support difficult:

```
// 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 → 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 ≈ 1 million × 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

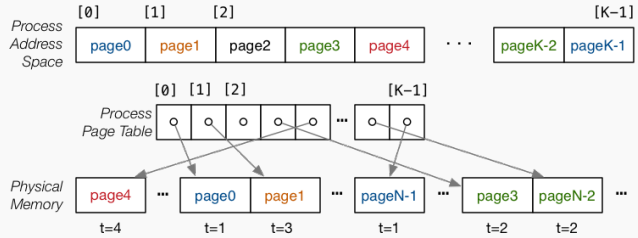
Page Tables

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

```
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?

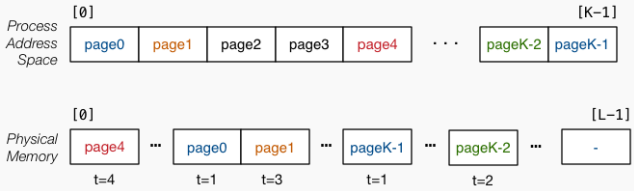
- code is loaded from the executable file stored on disk into read-only pages
- some data (e.g. C strings) also loaded into read-only pages
- initialised data (C global/static variables) also loaded from executable file
- pages for uninitialised data (heap, stack) are zero-ed
  - prevents information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

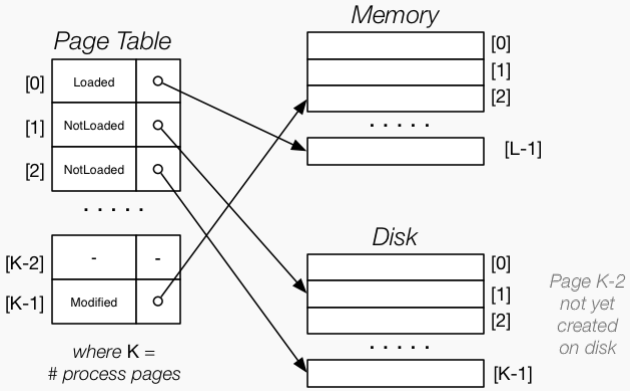
- exists on disk for the duration of the process's execution
- and only some parts of it are in memory at any given time



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`

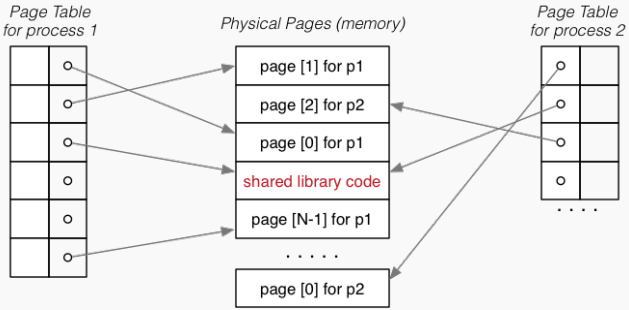
`5, 0, 0, 0, 5, 1, 1, 5, 1, 5, 2, 2, 3, 0, 0, 5`

Assume that all PTEs and frames are initially empty/unused

Virtual Memory - Read-only Pages

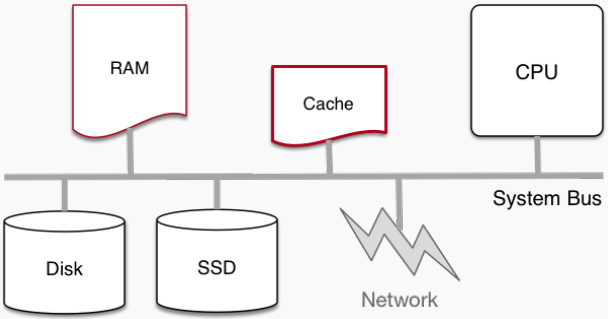
Virtual memory allows sharing of read-only pages (e.g. library code)

- several processes include same frame in virtual address space



Cache Memory

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



Small = MB, Fast = 5 × *RAM*

Cache Memory

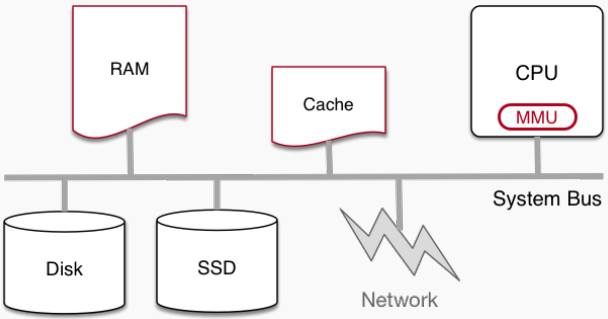
Cache memory

- 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

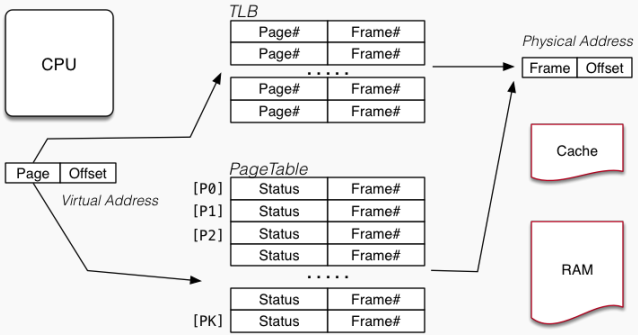
- provide specialised hardware (MMU) to do it efficiently
- sometimes located on CPU chip, sometimes separate



Memory Management Hardware

TLB = translation lookaside buffer

- lookup table containing (virtual,physical) address pairs



Concurrency/Parallelism

Concurrency multiple computations in overlapping time periods; does not have to be simultaneous

Parallelism multiple computations executing simultaneously

Parallel computation occurs at different level:

- spread across computers (e.g., with MapReduce)
- multiple cores of a CPU executing different instructions (MIMD)
- multiple cores of a CPU executing same instruction (SIMD)
  - e.g. GPU rendering pixels

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

Parallel Computing Across Many Computers

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 (*key*, *value*) pairs
- each node receives a set of pairs with common *key*(s)

Nodes then perform calculation on received data items

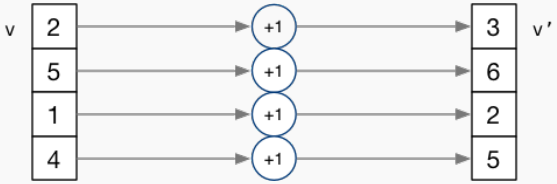
The *reduce* step computes the final result

- by combining outputs (calculation results) from the nodes

Also needs a way to determine when all calculations completed

Parallelism Across a 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

Example: GPU rendering pixels or neural network

Parallelism Across Processes

One method for creating parallelism:

Use `posix_spawn()` to create multiple processes, each does part of job.

- child executes concurrently with parent
- runs in its own address space
- inherits some state information from parent, e.g. open fd's

Processes have some disadvantages

- process switching expensive
- each require a significant amount of state (RAM)
- communication between processes limited and/or slow

One big advantage - separate address spaces make processes more robust.

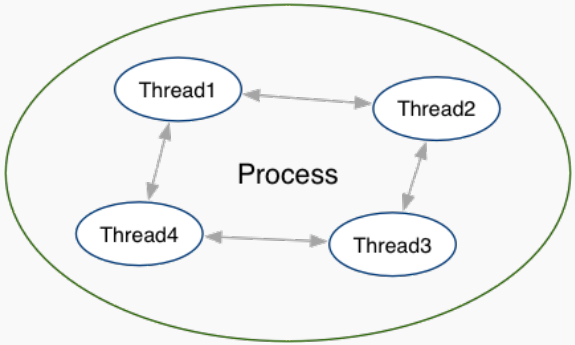
Parallelism within Processes

threads - mechanism for parallelism within process.

- threads allow simultaneous execution within process
- each thread has its own execution state
- threads within a process have same address space:
  - threads share code (functions)
  - threads share global & static variables
  - threads share heap (`malloc`)
- but separate stack for each thread
  - local variables not shared
- threads share file descriptor
- threads share signals

POSIX threads (pThreads)

```
// POSIX threads widely supported in Unix-like
// and other systems (Windows).  Provides functions
// to create/synchronize/destroy/... threads

#include <pthread.h>
```



Create A POSIX Thread

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

- creates a new thread with attributes specied in **`attr`**
  - `attr` can be `NULL`
- thread info stored in **`*thread`**
- thread starts by executing **`start_routine(arg)`**
- returns 0 if OK, -1 otherwise and sets **`errno`**
- analogous to **`posix_spawn()`**

Wait for A POSIX Thread

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

- wait until **`thread`** terminates
- **`thread`** return (or **`pthread_exit()`**) value is placed in **`*retval`**
- if **`thread`** has already exited, does not wait
- if main returns or exit called, all threads terminated
- programs typically need towait for all threads before main returns/exit called
- analogous to **`waitpid`**

Terminate A POSIX Thread

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

- terminate execution of thread (and free resources)
- **retval** is returned (see **pthread_join**)
- if **thread** has already exited, does not wait
- analagous to **exit**

Simple example - Creating Two threads

```c
#include <pthread.h>
// this function is called to start thread execution
// it can be given any pointer as argument (int *) in this example
void *run_thread(void *argument) {
    int *p = argument;
    for (int i = 0; i < 10; i++) {
        printf("Hello this is thread #%d: i=%d\n",  *p, i);
    }
    // a thread finishes when the function returns or  thread_exit
    // a pointer of any type can be returned
    // this can be obtained via thread_join's 2nd argument
    return NULL;
}
```

source code for two_threads.c

Simple example - Creating Two threads

```c
//create two threads performing almost the same task
pthread_t thread_id1;
int thread_number1 = 1;
pthread_create(&thread_id1, NULL, run_thread, &thread_number1);
int thread_number2 = 2;
pthread_t thread_id2;
pthread_create(&thread_id2, NULL, run_thread, &thread_number2);
// wait for the 2 threads to finish
pthread_join(thread_id1, NULL);
pthread_join(thread_id2, NULL);
```

source code for two_threads.c

CLassic Bug - Sharing a Variable Between Threads

```c
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);
```

source code for two_threads_broken.c

- variable **thread_number** will probably have changed in main before thread 1 starts executing
- so thread 1 will probably print **Hello this is thread 2**

Simple example - Creating Many threads

```
int n_threads = strtol(argv[1], NULL, 0);
assert(n_threads > 0 && n_threads < 100);
pthread_t thread_id[n_threads];
int argument[n_threads];
for (int i = 0; i < n_threads; i++) {
    argument[i] = i;
    pthread_create(&thread_id[i], NULL, run_thread, &argument[i]);
}
// wait for the threads to finish
for (int i = 0; i < n_threads;i++) {
    pthread_join(thread_id[i], NULL);
}
```

source code for n_threads.c

Simple example - Dividing a task between threads

```
struct job {
    long start;
    long finish;
    double   sum;
};
void *run_thread(void *argument) {
    struct job *j = argument;
    long start = j->start;
    long finish = j->finish;
    double sum = 0;
    for (long i = start; i < finish; i++) {
        sum += i;
    }
    j->sum = sum;
```

source code for thread_sum.c

Simple example - Dividing a task between threads

```
printf("Creating %d threads to sum the first %lu integers\n",
       n_threads, integers_to_sum);
printf("Each thread will sum %lu integers\n", integers_per_thread);
pthread_t thread_id[n_threads];
struct job jobs[n_threads];
for (int i = 0; i < n_threads; i++) {
    jobs[i].start = i * integers_per_thread;
    jobs[i].finish = jobs[i].start + integers_per_thread;
    if (jobs[i].finish > integers_to_sum) {
        jobs[i].finish = integers_to_sum;
    }
    // create a thread which will sum integers_per_thread integers
    pthread_create(&thread_id[i], NULL, run_thread, &jobs[i]);
}
```

source code for thread_sum.c

Simple example - Dividing a task between threads

```
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

Simple example - Dividing a task between threads

```
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

Simple example - Dividing a task between threads

- on a AMD Ryzen 3900x which can run *24* threads simultaneously
- 1 thread takes `6.9` seconds to sum first 10000000000 integers
- 2 threads takes `3.6` seconds to sum first 10000000000 integers
- 4 threads takes `1.8` seconds to sum first 10000000000 integers
- 12 threads takes `0.6` seconds to sum first 10000000000 integers
- 24 threads takes `0.3` seconds to sum first 10000000000 integers
- 50 threads takes `0.3` seconds to sum first 10000000000 integers
- 500 threads takes `0.3` seconds to sum first 10000000000 integers

Example - Unsafe Access to Global Variable

```
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 - Unsafe Access to Global Variable

```
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 Variable: Race Condition

Incrementing a global variable is not an atomic (indivisible) operation.

```
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 Variable: Race Condition

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

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

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

One increment is lost.

Note threads don't share registers or stack (local variable).

They do share global variables.

Global Variable: Race Condition

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

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

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


Exclude Other Threads from Code

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

- only one thread can enter a *critical section*
- establishes mutual exclusion — *mutex*
- call **`pthread_mutex_lock`** before
- call **`pthread_mutex_unlock`** after
- only 1 thread can execute in protected code
- for example:

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

Example - Protecting Access to Global Variable with a Mutex

```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
        bank_account = bank_account + 1;
        pthread_mutex_unlock(&bank_account_lock);
    }
    return NULL;
}
```

source code for bank_account_mutex.c

Semaphores

Semaphores are special variables which provide a more general synchronisation mechanism than mutexes.

```c
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- `sem_init` initialises `sem` to `value`
- `sem_wait` - classically called `P()`
  - if `sem` > 0, decrement `sem` and continue
  - otherwise, wait until `sem` > 0
- `sem_post` - classically called `V()`
  - increment `sem` and continue

Allow n threads access to a resource

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

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


Protecting Access to Global Variable with a Semaphore

```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
        // 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_semaphore.c

Protecting Access to Global Variable with a Semaphore

```
// 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);
```

source code for bank_account_semaphore.c

File Locking

`int flock(int FileDesc, int Operation)`

Similar to mutexes for a file.

- controls access to shared files (**note**: files not fds)
- possible operations
  - `LOCK_SH` ... acquire shared lock
  - `LOCK_EX` ... acquire exclusive lock
  - `LOCK_UN` ... unlock
  - `LOCK_NB` ... operation fails rather than blocking
- in blocking mode, `flock()` does not return until lock available
- only works correctly if all processes accessing file use locks
- return value: 0 in success, -1 on failure

File Locking

If a process tries to acquire a *shared lock* ...

- if file not locked or other shared locks, OK
- if file has exclusive lock, blocked

If a process tries to acquire an *exclusive lock* ...

- if file is not locked, OK
- if any locks (shared or exclusive) on file, blocked

If using a non-blocking lock

- `flock()` returns 0 if lock was acquired
- `flock()` returns -1 if process would have been blocked

Concurrent Programming is Complex

Concurrency is *complex* with many issues beyond this course:

Data races thread behaviour depends on unpredictable ordering; can produce difficult bugs or security vulnerabilities

Deadlock threads stopped because they are wait on each other

Livelock threads running without making progress

Starvation threads never getting to run

Example - deadlock accessing two resources

```c
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 - deadlock accessing two resources

```c
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 - deadlock accessing two resources

```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

Course Goals

At the end of COMP1521, we hope that you ...

- understand the structure of computer systems
- can describe how computers/programs work at a low-level
- are better able to reason about and debug your C programs

Major topics ...

- components of modern computer systems
- how C programs execute (at the machine level)
- how to write (MIPS) assembly language
- Unix/Linux system-level programming

Syllabus/Topics

- bit-level operations
- representation of integers & doubles
- the basic components of a (MIPS) CPU
- representation of programs as (MIPS) machine code
- how to write programs in (MIPS) assembler
- how C programs are implemented as (MIPS) instructions
- systems programming, including:
  - file operations
  - processes
- representation of characters as Unicode
- introduction to virtual memory
- introduction to threads/concurrency

Assessment

- 15% Labs
- 10% Weekly Programming Tests
- 15% Assignment 1 — due week 7
- 15% Assignment 2 — due week 10
- 45% Final Exam

Above marks may be scaled to ensure an appropriate distribution
To pass you must:

- score 50/100 overall
- score 18/45 on final exam

For example:

55/100 overall, 17/45 on final exam ⇒ **55 UF** not 55 PS

Labs and Tests

- 9 labs - weeks 1-5,7-10
  - max lab mark 2 marks with challenge exercises
  - max lab mark ~1.6 marks without challenge exercises
  - 9 labs marks summed and capped at 15
  - you could get 99% for lab mark without challenge exercises
  - most people will get 12+/15
- 8 tests - weeks 3...10:
  - max test mark 1.7
  - best 6 of 8 test marks summed and capped at to give mark out of 10.
  - most people will get 7+/10

20T2 COMP1521 Final Exam

- The 20T2 COMP1521 Final Exam is available at:
  https://cgi.cse.unsw.edu.au/~cs1521/20T3/exam/practice/questions
- You can complete it as a practice exam
- Autotests available
- Sample answers will be released noon Wednesday 25th
- 20T3 exam will use similar format similar for at least some questions

Exam

- Run under same conditions as Weekly Tests
- Except 3 hours and some questions may not be coding
- Saturday 28 November 13:00 — 16:00
- Exam will be released on class web site at 12:50
- Announcements before & during exam will be sent to your UNSW email
- Questions during exam can be sent to **`cs1521.exam@cse.unsw.edu.au`**
- We may also announce a 2nd place you can ask questions during exam.

Exam Conditions

- You are not permitted to communicate (email, phone, message, talk, ...) to anyone but COMP1521 staff during exam
- You are not permitted to get help from anyone but COMP1521 staff during the exam.
- This is a closed book exam: you are not permitted to access papers, books, files on your computer or the internet
- You are permitted to access the exam web pages on the class web site
- You are permitted to access the online language cheatsheets & documentation on the class web site
- Deliberate violation of exam conditions will be treated as serious misconduct

Exam Format

- 10-15 questions
- Each question answered in a separate file.
- Some questions will ask you to write C.
- Some questions will ask you to write MIPS.
- Other languages not permitted (e.g., Python, C++, Java, Rust, ...)
- Some questions may not involve coding
- Answers will be submitted with give.
- Questions not equal difficulty
- Questions may not be worth equal marks

Non-coding Questions

- Answers must be an specified file, e.g. `q1.txt`
- Question may specify format of file
  - e.g. 5 integers one per line
  - follow this format `EXACTLY`
- Question must give you an initial file to complete
- File will be submitted with give.

Coding Questions

For question that require you to write C or MIPS ...

- Questions will usually include examples.
- You may or may not be given starting code.
- You may or may not be given test data or other files
- 1 or more autotests may be available on submission.
- `Passing autotests does not guarantee any marks.` Do your own testing.
- There may be no submission tests for some questions.
- It is not sufficient to match any supplied examples.

Marking of Coding Questions

- Answers will be run through automatic marking software.
  - Please follow the input/output format shown exactly.
  - Please make your program behave exactly as specified.
- All answers are hand marked, guided by automarking.
  - No marks awarded for style or comments ...
  - But use decent formatting so the marker can read the program!
  - Comments only necessary to tell the marker something.
- Minor errors will result in only a small penalty.
  - e.g. an answer correct except for a missing semi-colon would receive almost full marks.
- No marks will given unless an answer contains a substantial part of a solution ($> 33\%$).
- No marks just for starting a question and writing some code

Special Exam Conditions

- any extra time specified in your ELS exam conditions is allowed in this exam
- all student see the same exam questions text
- the text shows the standard exam deadline, any extra time is additional to it
- give should be configured to allow your extra time
- give should show a deadline **including** your extra time
- email `cs1521@cse.unsw.edu.au` immediately during exam if you have have concerns regarding ELS conditions
- if ELS conditions prevent you taking exam, alternative will be supp in January

Exam On Same Day

- a small number of students have exams on the same day
  - MARK1012 & COMP9517
- these exams are timetabled as all day exams
- special consideraions have advised they will not give special consideration for clashes with all day exams

Problems During Exam

- if a problem occurs during the exam, e.g.internet failure
- please document the problem as possible, e.g. take screenshot
- email `cs1521.exam@cse.unsw.edu.au`
- if the problem is of short duration we may be able to give you extra time
- otherwise you'll need to apply for special consideration

Special Consideration — 'Fit to Sit'

- By starting the exam, you are saying `"I am well enough to sit it"`.
- If unwell befor exam, see a doctor, apply for Special Consideration.
- If you become unwell during the exam
  - email `cs1521.exam@cse.unsw.edu.au`
  - if you can't continue the exam you will need to see a doctor and apply for Special Consideration.

What to study

- Important Areas to Focus Your Study On
  - anything covered in a standard lab exercise
  - anything covered in a weekly test
  - anything covered by the assignments
- Less Important Areas
  - challenge lab exercises
  - topics not covered in labs, tests or assignments
  - may still be questions on these topic but not many
- Even Less Important Areas
  - creating pipes and other complex aspects of creating processes
  - complex signal handling
  - semaphores & file locking
  - might or might not be a question on these topic

Provisional Results

- Exam marks will be made available via class database when marking is complete.
- I'll send email announcing this.
- Marking will probably take 12 days.
- Your will receive marks for individual exam questions.
- You will be opportunity to have marking reviewed.
- Final results will appear on myUNSW.

Supplementary Assessment

- UNSW supplementary exams are run centrally.
- Supplementary exams are for students who miss original exam due to illness/misadventure.
- If this is you — apply for special consideration.
- Lecturers & schools can not offer supps.
- Students with borderline results are not offered supps.
  . . . except potential graduands
- Supp exams centrally timetabled for January 11-15, 2021
- Similar format to final exam

What did you like?

One aim of COMP1521 is to give a taste of many topics:

- Liked MIPS/assembly?
  ⇒ COMP3222, COMP3211 . . .
- Curious about programming languages?
  ⇒ COMP3131, COMP3141, COMP3161, . . .
- Liked Operating Systems?
  ⇒ COMP3231/3891, COMP9242, . . .
- Liked Concurrency?
  ⇒ COMP3151, . . .
- Liked Networking?
  ⇒ COMP3331, COMP4336, COMP4337, . . .
- Liked Unix shell?
  ⇒ COMP2041

Course Offerings

COMP1531 Software Engineering Fundamentals

- 2021: term1, term3

COMP2511 Object-oriented Programming

- 2021: term2, term3

COMP2521 Data Structures and Algorithms

- 2021: term1, term2, term3

COMP3231 Operating Systems

- 2021: term1

COMP1521- The Bad

- MIPS and its relation to C needs to be better explained
- Not enough time to cover (so) many things
- Tuts and Labs need to integrate better.
- Labs a lot of work - but you learnt a lot
- Assignments a **lot** of work - but you learnt a lot

COMP1521 - The Good

- Most labs exercises (do you agree??)
- Weekly Tests (do you agree??)
- Tutors
- Discourse
- Students

And that's all . . .

- Good Luck
- I hope what you've learnt in this course will be useful.
- I hope you get the mark you deserve.