Course Introduction

Computer Systems

1. Processor
2. Storage
3. C Program Lifecycle

C Revisited

1. Abstract Data Types
    a. Implementation in C
2. Collections
3. Polymorphism
4. Stacks and Queues

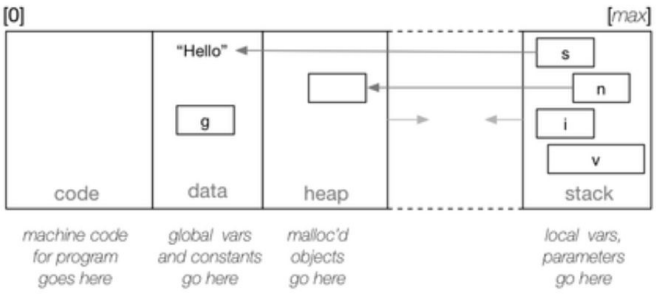
Bit Manipulation

1. Binary, Decimal, Octal, Hexadecimal
2. Bitwise Operators
    a. AND (&) OR (|) XOR (^) NEG (~) LS (<<) RS (>>)
3. File Permissions
    a. -rwxrwxrwx (owner | group | everyone else)

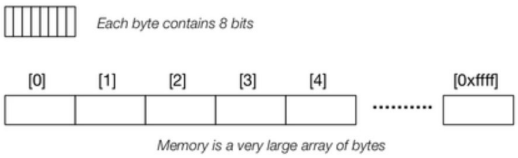
Memory and Data

Memory and Data

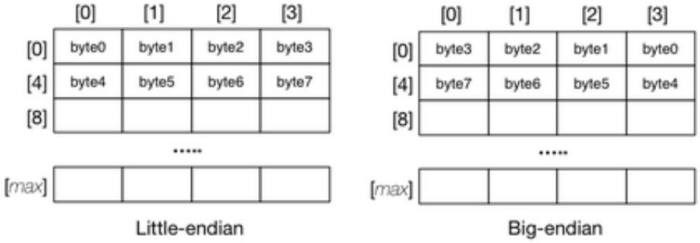
1. C sees data as a collection of variables, each of which have:
    a. Location, value, name, type (size, how to interpret, operations), scope, lifetime
2. Physical view of memory:



Memory = indexed array of bytes



Indexes are "memory addresses" (a.k.a. pointers)



Data Representation

1. Character Data

a. ASCII
   i. 7-bit values, using lower 7-bits of a byte (the top bit is always 0)
   ii. Has values in range 0x00 to 0x7F (0-127)
      - control characters (0..31) ... e.g. `'\0'`, `'\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'`

b. UTF-8 (Unicode)
   i. 8-bit values, with ability to extend to multi-byte values

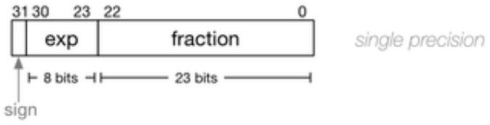
#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	**0xxxxxxx**	-	-	-
2	11	**110xxxxx**	**10xxxxxx**	-	-
3	16	**1110xxxx**	**10xxxxxx**	**10xxxxxx**	-
4	21	**11110xxx**	**10xxxxxx**	**10xxxxxx**	**10xxxxxx**

ch	unicode	bits	simple binary	UTF-8 binary
$	U+0024	7	010 0100	00100100
¢	U+00A2	11	000 1010 0010	11000010 10100010
€	U+20AC	16	0010 0000 1010 1100	11100010 10000010 10101100
𐍈	U+10348	21	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000

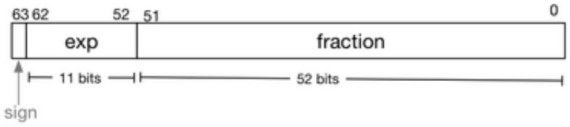
2. Numeric Data

a. Integer (subset of mathematical integers)
   i. Decimal (0..9), Hexadecimal (0..F), Octal (0..7)
   ii. Unsigned - 32 bits [0..2^32 - 1]
   iii. Signed - 32 bits [-2^31..2^31 - 1]
   iv. Short (16 bits), Long (64 bits)
   v. Two's Complement
      1. NEG all bits, then add 1

b. Floating Point (subset of the mathematical real numbers)
   i. Float (32-bit quantity)
   ii. Double (64-bit quantity)



single precision



double precision

Component	Min Value	Max Value
exponent	00000000 = -127	11111111 = 128
fraction	00...00 = 0	11...11 = $2^{-1}+2^{-2}+...+2^{-24}$

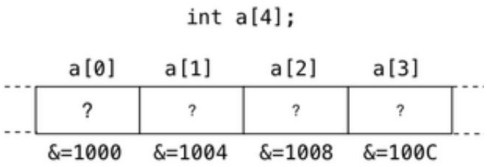
Component	Min Value	Max Value
exponent	00000000000 = -2047	11111111111 = 2048
fraction	00...00 = 0	11...11 = $2^{-1}+2^{-2}+...+2^{-51}$

3. Pointers

a. Represent memory addresses (typically 32-bits)
   i. Data pointers reference addresses in data/heap/stack regions
   ii. Function pointers reference addresses in code region
b. Can be manipulated by pointer arithmetic
   i. For any pointer T *p, p++ increases p by sizeof(T)

4. Arrays

a. Have N elements, each of type T, laid out adjacent in memory

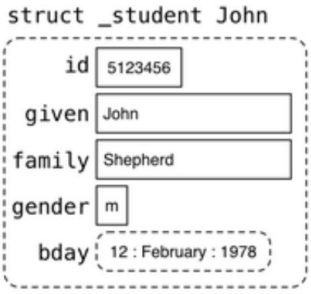


5. Structs

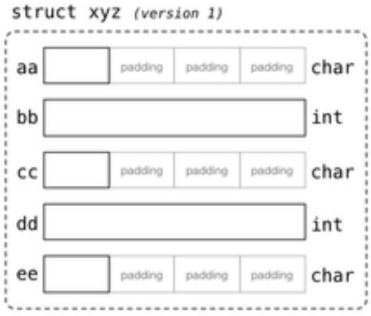
a. Have a number of components, where each component has a Name and a Type
   i. Internal layout of struct components is determined by compiler

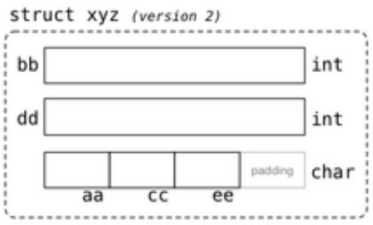
```
typedef struct … Date;

struct _student {
   int id;
   char given[50];
   char family[50];
   char gender;
   Date bday;
};
```

Defines a new data type called `struct _student`



b. Padding (to ensure alignment)



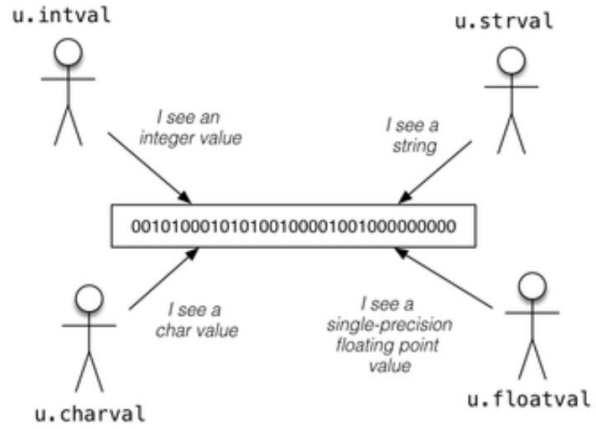


c. Variable-length Structs
   i. Structs can contain pointers to dynamic objects, but also “embed” a dynamic object
d. Bit-wise Structs
   i. For fine-grained control over layout of fields in struct

```
struct _bit_fields {
   unsigned int first_bit    : 1,
                next_7_bits  : 7,
                last_24_bits : 24;
};
```

6. Unions

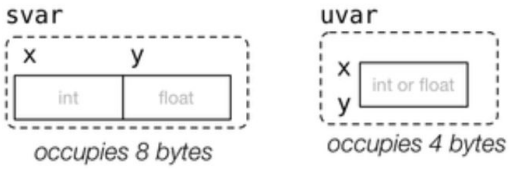
a. Specify multiple interpretations for a single piece of memory
   i. sizeof(Union) is the size of the largest member
   ii. &uvar.Member1 == &uvar.Member2 == &uvar.Member3



Difference between a `struct` and a `union`

```
struct _s {
   int   x;
   float y;
} svar;
```

```
union _u {
   int   x;
   float y;
} uvar;
```



7. Enumerated Types

a. Define a set of distinct named values, assigned consecutive int values, starting from 0

```
typedef enum { RED, YELLOW, BLUE } PrimaryColours;
```

Is equivalent to:

```
#define RED    0
#define YELLOW 1
#define BLUE   2
```

Instruction Set Architectures

Instruction Set Architectures

1. CPU Architecture
    a. Modern CPU has:
        i. A set of data registers
        ii. A set of control registers
        iii. An arithmetic-logic unit
        iv. Access to random access memory (RAM)
        v. A set of simple instructions
            1. Transfer data between memory and registers
            2. Push values through the ALU to compute results
            3. Make tests and transfer control of execution
2. Instruction Sets
    a. RISC (reduced instruction set computer)
        i. Small set of simple, general instructions
        ii. Separate computation & data transfer instructions
        iii. E.g. MIPS, RISC, Alpha, SPARC, PowerPC, ARM
    b. CICS (complex instruction set computer)
        i. Larger set of power instructions
        ii. Each instruction has multiple actions (compute + store)
        iii. More circuitry to decode/process instructions
        iv. E.g. Intel x86, PDP, VAX, Z80
3. Machine-level Instructions
    a. Typically 32-bit words per instruction
    b. Partition bits in each word into operator and operands
    c. Operands and destination are typically registers

4. Fetch-Execute Cycle
    a. All CPUs have program execution logic like where PC = Program Counter
    b. Executing an instruction involves:
        i. Determining the operator, registers and memory location
        ii. Carrying out the operation
        iii. Storing the result

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

Assembly Language

1. Assembler
    a. Symbolic language for writing machine code, using mnemonics rather than hex codes
    b. Reference registers using either numbers or names
    c. Can associate names to memory addresses

MIPS

1. 32 x 32-bit general purpose registers
2. PC = 32-bit register (always aligned on 4-byte boundary)
3. HI, LO for storing results of multiplication and division
4. 32-bit registers:

Reg	Name	Notes
$0	zero	the value 0, not changeable
$1	$at	**a**ssembler **t**emporary; reserved for assembler use
$2	$v0	**v**alue from expression evaluation or function return
$3	$v1	**v**alue from expression evaluation or function return
$4	$a0	first **a**rgument to a function/subroutine, if needed
$5	$a1	second **a**rgument to a function/subroutine, if needed
$6	$a2	third **a**rgument to a function/subroutine, if needed
$7	$a3	fourth **a**rgument to a function/subroutine, if needed
$8..$15	$t0..$t7	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
$16..$23	$s0..$s7	**s**afe function variable; must not be overwritten by called subroutine
$24..$25	$t8..$t9	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
$26..$27	$k0..$k1	for **k**ernel use; may change unexpectedly
$28	$gp	**g**lobal **p**ointer
$29	$sp	**s**tack **p**ointer
$30	$fp	**f**rame **p**ointer
$31	$ra	**r**eturn **a**ddress of most recent caller

5. Floating point registers (only the even register is addressed)

$f0..$f2	hold floating-point function 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

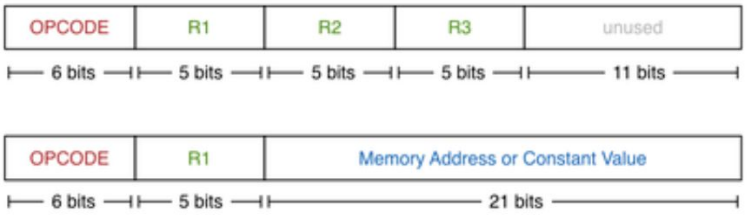
MIPS Conventions

1. # Comments
2. Labels :
3. .directives

Region	Address	Notes
text	0x00400000	contains only instructions; read-only; cannot expand
data	0x10000000	data objects; readable/writeable; can be expanded
stack	0x7fffefff	grows down from that address; readable/writeable

MIPS Instructions

1. Classes of Instructions
    a. Load and store (transfer data between registers and memory)
    b. Computational (perform arithmetic/logical operations)
    c. Jump and branch (transfer control of program execution)
    d. Coprocessor (standard interface to various co-processors)
    e. Special (miscellaneous tasks, e.g. syscall)
2. Addressing Modes
    a. Direct/Indirect (between memory and register)
    b. Immediate (constant to register)
    c. Register + register + destination register
3. Possible instruction formats:



Common Instructions

1. Data Movement

```
la   $t1,label     # reg[t1] = &label
lw   $t1,label     # reg[t1] = memory[&label]
sw   $t3,label     # memory[&label] = reg[t3]
                   # &label must be 4-byte aligned
lb   $t2,label     # reg[t2] = memory[&label]
sb   $t4,label     # memory[&label] = reg[t4]
move $t2,$t3       # reg[t2] = reg[t3]
lui  $t2,const     # reg[t2][31:16] = const
```

2. Bit Manipulation

```
and  $t0,$t1,$t2   # reg[t0] = reg[t1] & reg[t2]
and  $t0,$t1,Imm   # reg[t0] = reg[t1] & Imm[t2]
                   # Imm is a constant (immediate)
or   $t0,$t1,$t2   # reg[t0] = reg[t1] | reg[t2]
xor  $t0,$t1,$t2   # reg[t0] = reg[t1] ^ reg[t2]
neg  $t0,$t1       # reg[t0] = ~ reg[t1]
```

3. Arithmetic

```
add  $t0,$t1,$t2  # reg[t0] = reg[t1] + reg[t2]
                  #   add as signed (2's complement) ints
sub  $t2,$t3,$t4  # reg[t2] = reg[t3] + reg[t4]
addi $t2,$t3, 5   # reg[t2] = reg[t3] + 5
                  #   "add immediate" (no sub immediate)
addu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   add as unsigned integers
subu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   subtract as unsigned integers
mult $t3,$t4      # (Hi,Lo) = reg[t3] * reg[t4]
                  #   store 64-bit result in registers Hi,Lo
div  $t5,$t6      # Lo = reg[t5] / reg[t6] (integer quotient)
                  # Hi = reg[t5] % reg[t6] (remainder)
mfhi $t0          # reg[t0] = reg[Hi]
mflo $t1          # reg[t1] = reg[Lo]
                  # used to get result of MULT or DIV
```

4. Testing and Branching

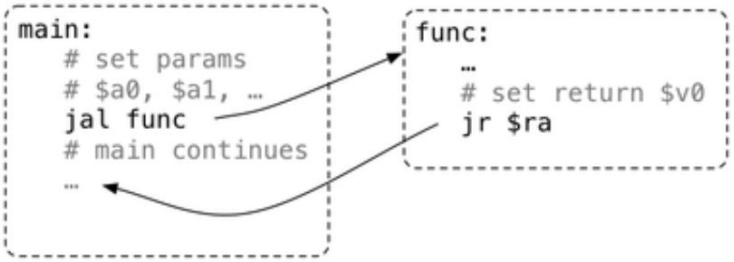
```
seq  $t7,$t1,$t2    # reg[t7] = 1 if (reg[t1]==reg[t2])
                    # reg[t7] = 0 otherwise (signed)
slt  $t7,$t1,$t2    # reg[t7] = 1 if (reg[t1] < reg[t2])
                    # reg[t7] = 0 otherwise (signed)
slti $t7,$t1,Imm    # reg[t7] = 1 if (reg[t1] < Imm)
                    # reg[t7] = 0 otherwise (signed)

j    label          # PC = &label
jr   $t4            # PC = reg[t4]
beq  $t1,$t2,label  # PC = &label if (reg[t1] == reg[t2])
bne  $t1,$t2,label  # PC = &label if (reg[t1] != reg[t2])
bgt  $t1,$t2,label  # PC = &label if (reg[t1] > reg[t2])
bltz $t2,label      # PC = &label if (reg[t2] < 0)
bnez $t3,label      # PC = &label if (reg[t3] != 0)
```

5. Jumping

```
jal  label          # make a subroutine call
                    # save PC in $ra, set PC to &label
                    # use $a0,$a1 as params, $v0 as return
```

6. Syscalls

Service	Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = char *	
read_int	5		integer in `$v0`
read_float	6		float in `$f0`
read_double	7		double in `$f0`
read_string	8	`$a0` = buffer, `$a1` = length	string in buffer (including "\n\0")



7. Directives

```
    .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 2^2-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'};
```

Addressing Modes

```
prog:
a:    lw    $t0, var      # address via name
b:    lw    $t0, ($s0)    # indirect addressing
c:    lw    $t0, 4($s0)   # indexed addressing
```

If `$s0` contains `0x10000000` and `&var` = `0x100000008`

- computed address for `a:` is `0x100000008`
- computed address for `b:` is `0x100000000`
- computed address for `c:` is `0x100000004`

Format	Address computation
(register)	address = *register = contents of register
k	address = k
k(register)	address = k + *register
symbol	address = &symbol = address of symbol
symbol ± k	address = &symbol ± k
symbol ± k(register)	address = &symbol ± (k + *register)

where *k* is a literal constant value (e.g. `4` or `0x10000000`)

INSTRUCTION SET GIVEN IN EXAM

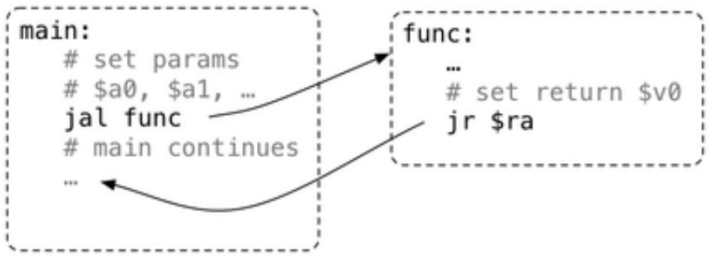
Operand Sizes

1. MIPS instructions can manipulate different-sized operands:
    a. Single bytes, two bytes, four bytes, signed, unsigned

Function/Subroutine Calls

Simple function-call protocol:

- load argument values into `$a0`, `$a1`, ...
- invoke `jal`: loads PC into `$ra`, jumps to function
- function puts return values in `$v0`, `$v1`
- returns to caller using `jr $ra`



More detail on the function call protocol (assume function `f()`):

- on entry, save the value of `$ra`
- on entry, save the value of any `$s?` registers modified by `f()`
- use the values in `$a?` as input parameters (e.g. `f(2,5)`)
- ... perform the required computation ...
- set the values of `$v0` and `$v1` as returned values
- on exit, restore the saved values of `$s?` registers
- on exit, restore the value of `$ra`

Control Structures

1. If-else and switch and while statements

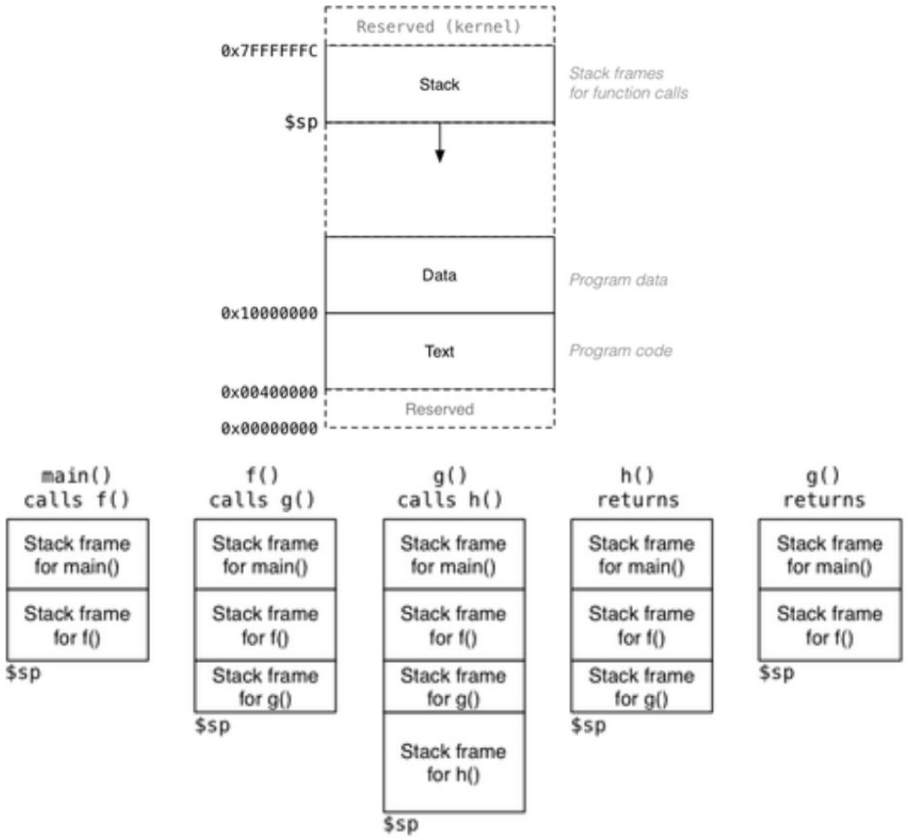
```
if_stat:
  t0 = evaluate Cond
  beqz $t0, else_part
  execute Statements1
  j     end_if
else_part:
  execute Statements2
end_if:
```

```
jump_tab:
   .word c1, c2, c2, c2, c3
switch:
    t0 = evaluate Expr
    if (t0 < 1 || t0 > 5)
       jump to default
    dest = jump_tab[(t0-1)*4]
    jump to dest
c1: execute Statements1
    jump to end_switch
c2: execute Statements2
    jump to end_switch
c3: execute Statements3
    jump to end_switch
default:
    execute Statements4
end_switch:
```

```
top_while:
    t0 = evaluate Cond
    beqz $t0,end_while
    execute Statements
    j     top_while
end_while:
```

Function/Subroutine calls in Detail

1. Function calls handled using the MIPS stack
    a. Each function allocates a small section of the stack (a frame)
        i. Used for saving registers, local variables and parameters to callees
        ii. Created in the function prologue (pushed) and removed in function epilogue (popped)
    b. LIFO behaviour



2. Structure of Functions
    a. Prologue:
        i. Create a stack frame for itself (change $fp and $sp)
            1. New $fp = old $sp - 4
            2. New $sp = old $sp - size of frame
        ii. Save the return address in the stack frame
        iii. Save any $s? Registers that it plans to change
    b. Epilogue
        i. Place the return value in $v0
        ii. Pop any pushed arguments off the stack
        iii. Restore the saved values of any $s? registers

iv. Restore the saved value of $ra (return address)
v. Remove its stack frame (change $fp and $sp)
vi. Return to the calling function

```
# start of function
FuncName:
   # function prologue
   # sets up stack frame
   # saves relevant registers
   ...
   # function body
   # performs computation
   # leaving result in $v0
   ...
   # function epilogue
   # restores registers
   # cleans up stack frame
   jr  $ra
```

Syscall/Variable Storage

Service	Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = char *	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0 = buffer, $a1 = length	string in buffer (including "\n\0")

```
    .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 2^2-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'};
```

Static vs Dynamic Allocation

Static allocation:

- uninitialised memory allocated at compile/assemble-time, e.g.

```
int  val;              val: .space 4
char str[20];          str: .space 20
int  vec[20];          vec: .space 80
```

- initialised memory allocated at compile/assemble-time, e.g.

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

Dynamic Allocation

- Variables local to a function stored in registers or on the stack

SPIM doesn't provide `malloc()`/`free()` functions

- but provides a `syscall` to extend `.data`
- before `syscall`, set `$a0` to the number of bytes requested
- after `syscall`, `$v0` holds start address of allocated chunk

Example:

```
li   $a0, 20    # $v0 = malloc(20)
li   $v0, 9
syscall
move $s0, $v0   # $s0 = $v0
```

Cannot access allocated data by name; need to retain address.

No way to free allocated data, and no way to align data appropriately

Arrays in MIPS

1D Arrays

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

nums: .word 1, 3, 5, 7, 9
# int nums[6] = {1,2,3,5,7,9}
```

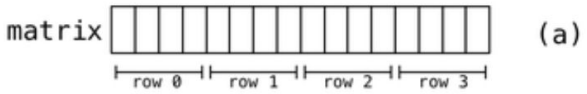
Scanning across an array:

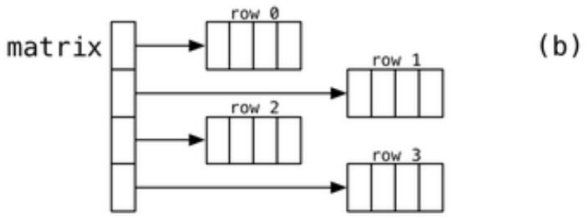
```
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
#    printf("%d\n", vec[i]);

   li   $s0, 0              # i = 0
   li   $s1, 10
loop:
   bge  $s0, $s1, end_loop  # if (i >= 10) break
   li   $t0, 4
   mul  $t0, $s0, $t0       # index -> byte offset
   lw   $a0, vec($t0)       # a0 = vec[i]
   jal  print               # print a0
   addi $s0, $s0, 1         # i++
   j    loop
end_loop:
```

2D Arrays

Represented by either a) or b)

```
int matrix[4][4];
```






```
# for strategy (a)
matrix: .space 64
# for strategy (b)
row0:   .space 16
row1:   .space 16
row2:   .space 16
row3:   .space 16
matrix: .word row0, row1, row2, row3
```

Summing all elements of a) and b)

```
   li  $s0, 0              # sum = 0
   li  $s1, 4              # s1 = 4
   li  $s2, 0              # i = 0
loop1:
   beq  $s2, $s1, end1     # if (i >= 4) break
   li   $s3, 0             # j = 0
loop2:
   beq  $s3, $s1, end2     # if (j >= 4) break
   mul  $t0, $s2, 16       # off = 4*4*i + 4*j
   mul  $t1, $s3, 4        #  matrix[i][j] is
   add  $t0, $t0, $t1      #  done as *(matrix+off)
   lw   $t0, matrix($t0)   # t0 = matrix[i][j]
   add  $s0, $s0, $t0      # sum += t0
   addi $s3, $s3, 1        # j++
   j    loop2
end2:
   addi $s2, $s2, 1        # i++
   j    loop1
end1:
```

```
   li  $s0, 0              # sum = 0
   li  $s1, 4              # s1 = 4 (sizeof(int))
   li  $s2, 0              # i = 0
loop1:
   beq  $s2, $s1, end1     # if (i >= 4) break
   li   $s3, 0             # j = 0
   mul  $t0, $s2, 4        # off = 4*i
   lw   $s4, matrix($t0)   # row = &matrix[i][0]
loop2:
   beq  $s3, $s1, end2     # if (j >= 4) break
   mul  $t0, $s3, 4        # off = 4*j
   add  $t0, $t0, $s4      # int *p = &row[j]
   lw   $t0, ($t0)         # t0 = *p
   add  $s0, $s0, $t0      # sum += t0
   addi $s3, $s3, 1        # j++
   j    loop2
end2:
   addi $s2, $s2, 1        # i++
   j    loop1
end1:
```

Structs in MIPS

Accessing struct components is by offset, not name like in C

Instances of structures can be created by allocating space:

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

Structs that are local to functions can be stored on the stack

```
fun:                            int fun(int x)
   # prologue                   {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)                 // push onto stack
   addi $sp, $sp, -56              Student st;
   move $t0, $sp                   Student *t0 = &st;
   # function body
   ... compute ...                 // compute using t0
   # epilogue                      // to access struct

   addi $sp, $sp, 56              // pop st off stack
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra                    }
```

Linked Structures in MIPS

MIPS doesn't have a "malloc" function to dynamically allocate the structs of a linked list but can do:

```
...                     # $s0 represents Node *first
li   $a0, 8             # sizeof(Node) == 8
jal  malloc
move $s0, $v0           # s0 = malloc(sizeof(Node))
li   $t0, 1
sw   $t0, 0($s0)        # s0->value = 1
li   $a0, 8             # required: $a0 not persistent
jal  malloc
mv   $t1, $v0           # s1 = malloc(sizeof(Node))
sw   $t1, 4($s0)        # s0->next = s1
li   $t0, 2
sw   $t0, 0($t1)        # s1->value = 2
sw   $0, 4($t1)         # s1->next = NULL
...
```

Compiling C to MIPS

C → MIPS

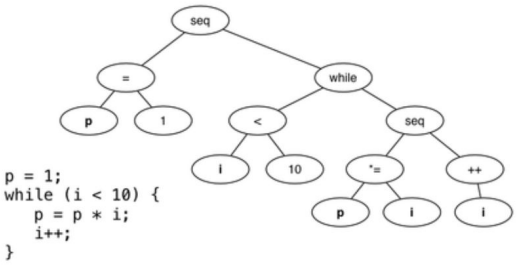
1. Convert #include and #define

   `#include` *File*
   - replace `#include` by contents of file
   - `"name.h"` ... uses named *File*`.h`
   - `<name.h>` ... uses *File*`.h` in `/usr/include`

   `#define` *Name Constant*
   - replace all occurences of symbol *Name* by *Constant*
   - e.g `#define MAX 5`
     `char array[MAX]` → `char array[5]`

   `#define` *Name*`(`*Params*`)` *Expression*
   - replace *Name*`(`*Params*`)` by *SubstitutedExpression*
   - e.g. `#define max(x,y) ((x > y) ? x : y)`
     `a = max(b,c)` → `a = ((b > c) ? b : c)`

2. Parse code to check syntax
   a. C Parser understands C language, attempts to convert C program into “parse tree”



3. Manage a list of symbols used in program
   a. Compiler keeps track of names
      i. Scope, lifetime, local/external definition
      ii. Disambiguates (x in main() vs x in fun())
      iii. Resolves symbols to specific locations
4. Decide how to represent data structures
5. Allocate local variables to registers or stack
   a. Stack - persists for whole function
   b. Register - efficient, good for variables with a small scope
   c. Temporary registers - used for expression evaluation
6. Map control structures to MIPS instructions
   a. Uses templates for typical control structures (if, while, for etc.)

Computer Systems Architecture

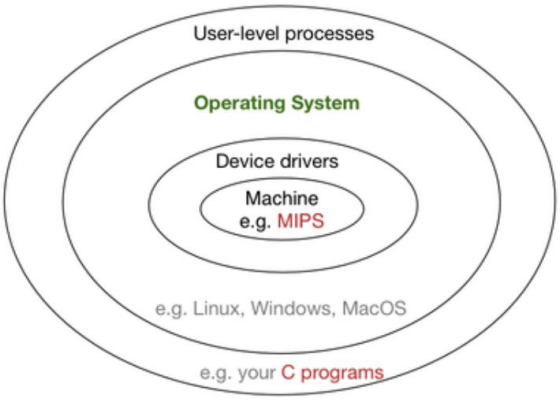
Operating Systems

- Have privileged access to the raw machine
- Manage machine resources (CPU, disk, memory)
- Provide interface to access machine-level operations
- Arrange the execution of user programs
- Provide multi-tasking and (pseudo) parallelism

Abstractions:

- Users/privileges
- File system
- Processes
- Communication

Core OS functions form the kernel, which runs in privileged mode



System Calls

Unix/Linux provides 100s of system calls which a user can call through an API (POSIX + Linux)

- Process Mgmt (fork, exec, exit…)
- File Mgmt (open, read, fstat…)
- Device Mgmt (ioctl…)
- Information maintenance (getuid, settimeofday…)
- Communication (pipe, connect, send…)

Invocation

- Directly, through a library of system calls
  - E.g. man 2 open
- Indirectly, through functions in the C libraries
  - E.g. man 3 fopen

System Call Operation

- Sys calls attempt to perform actions, but may fail
  - Return value of -1 typically means an error has occurred
  - Global variable **errno** may contain the error
- C programs need to check and handle errors themselves and can use **error(status, errnum, format, expr,...)**

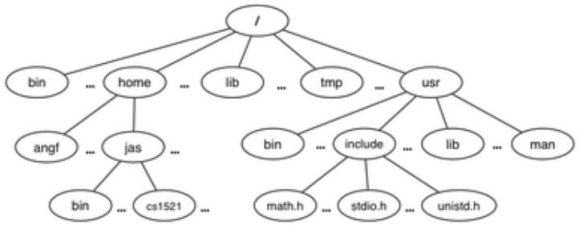
```
in = open(argv[1],O_RDONLY);
if (in < 0)
   error(errno, errno, "Can't open %s", argv[1]);
close(in);
```

File Systems

Provide a mechanism for managing stored data:

- Typically on a disk
- Allocating chunks of space on the device to files
    - A file = sequence of bytes
- Allowing access to files by name and with access rights
- Arranging access to files via directories
- Maintaining info about files/directories (metadata)
- Dealing with damage on the storage device

Unix/Linux file system is tree-structured



Paths

The file system is used to access various types of objects which are referenced via a path

- Files, directories, devices, processes, sockets

A path can be:

1. Absolute
    a. Full path from room /usr/...
2. Relative
    a. Path starts from CWD
        i. CWD = current working directory
        ii. Processes have a notion of their location within the file system

File-system-related Types

1. **off_t**
    a. Offsets within files
2. **size_t**
    a. Number of bytes in some object
3. **ssize_t**
    a. Sizes of read/written blocks
    b. Same as size_t but signed to allow for error values
4. **struct stat**
    a. File system object metadata
    b. Stores info about file

File Metadata

Metadata for system files stored in **inodes**

- Physical location on storage device of file data, file type (file/directory), file size (bytes/blocks), ownership, access permissions, timestamps (create/access/update)

Accessing a File by Name

- open directory and scan for *name*
- if not found, "No such file or directory"
- if found as (*name*,`ino`), access inode table `inodes[ino]`
- 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 physical location to access device and read/write file's data

File System Operations

Unix presents a uniform interface to file system objects

- Functions/syscalls manipulate objects as a stream of bytes
- Accessed via a file descriptor

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 meta-data about a file system object
`mkdir()` ... create a new directory
`fsync()` ... synchronise file data in memory with data on disk
`mount()` ... place a filesystem on a device

`int open(char *`*Path*`, int` *Flags*`)`

- attempt to open an object at *Path*, according to *Flags*
- flags (defined in `<fcntl.h>`)
  - `O_RDONLY` ... open object for reading
  - `O_WRONLY` ... open object for writing
  - `O_APPEND` ... open object for writing at end
  - `O_RDWR` ... open object for reading and writing
  - `O_CREAT` ... create object if doesn't exist
- flags can be combined e.g. `(O_WRONLY|O_CREAT)`
- if successful, return file descriptor (small +ve `int`)
- if unsuccessful, return -1 and set `errno`

`int close(int` *FileDesc*`)`

- attempt to release an open file descriptor
- if this is the last reference to object, release its resources
- if successful, return 0
- if unsuccessful, return -1 and set `errno`

`ssize_t read(int` *FileDesc*`, void *`*Buffer*`, size_t` *Count*`)`

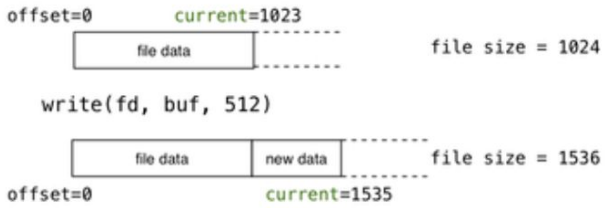
- attempt to read *Count* bytes from *FileDesc* into *Buffer*
- if "successful", return number of bytes actually read (*NRead*)
- if currently positioned at end of file, return 0
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* contains enough space
- advances the file offset by *NRead*
- does not treat `'\n'` as special

Once a file is `open()`'d ...

- the "current position" in the file is maintained as part of the fd entry
- the "current position" is modified by `read()`, `write()` and `lseek()`

`ssize_t write(int` *FileDesc*`, void *`*Buffer*`, size_t` *Count*`)`

- attempt to write *Count* bytes from *Buffer* onto *FileDesc*
- if "successful", return number of bytes actually written (*NWritten*)
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* has *Count* bytes of data
- advances the file offset by *NWritten* bytes



`off_t lseek(int` *FileDesc*`, off_t` *Offset*`, int` *Whence*`)`

- set the "current position" of the *FileDesc*
- *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)

`int stat(char *`*FileName*`, struct stat *`*StatBuf*`)`

- stores meta-data associated with *FileName* into *StatBuf*
- information includes
  - inode number, file type + access mode, owner, group
  - size in bytes, storage block size, allocated blocks
  - time of last access/modification/status-change
- returns -1 and sets `errno` if meta-data not accessible

`int fstat(int` *FileDesc*`, struct stat *`*StatBuf*`)`

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

`int stat(char *`*FileName*`, struct stat *`*StatBuf*`)`

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

File Stat Structure

```
struct stat {
   dev_t     st_dev;     // ID of device containing file
   ino_t     st_ino;     // inode number
   mode_t    st_mode;    // file type + permissions
   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; // blocksize for file system I/O
   blkcnt_t  st_blocks;  // number of 512B blocks allocated
   time_t    st_atime;   // time of last access
   time_t    st_mtime;   // time of last modification

     time_t    st_ctime;   // time of last status change
 };
```

Hard link - multiple directory entries referencing the same inode
Symbolic link (symlink) - a file containing the path name of another file
st_mode can be:

```
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   FIFO

S_IRUSR    0000400   owner has read permission
S_IWUSR    0000200   owner has write permission
S_IXUSR    0000100   owner has execute permission

S_IRGRP    0000040   group has read permission
S_IWGRP    0000020   group has write permission
S_IXGRP    0000010   group has execute permission

S_IROTH    0000004   others have read permission
S_IWOTH    0000002   others have write permission
S_IXOTH    0000001   others have execute permission
```

```
int mkdir(char *PathName, mode_t Mode)
```

- create a new directory called *PathName* with mode *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);`

```
int fsync(int FileDesc)
```

- ensure that data associated with *FileDesc* is written to storage

Unix/Linux makes heavy use of buffering

- data "written" to a file is initially stored in memory buffers
- eventually, it makes its way onto permanent storage device
- `fsync()` forces this to happen *now*

Writing to permanent storage is typically an expensive operation

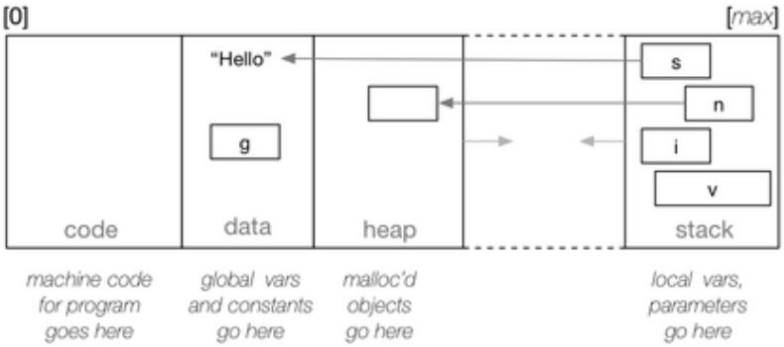
- `fsync()` is normally called just once at process exit

```
int mount(char *Source, char *Target, char *FileSysType,
                        unsigned long Flags, void *data)
```

- file systems normally exist on permanent storage devices
- `mount` attaches a file system to a specific location in the file hierarchy
- *Source* is often a storage device (e.g. `/dev/disk`)
- *Source* contains a file system (inode table, data chunks)
- *Target* (aka *mount point*) is a path in the file hierarchy
- *FileSysType* specifies a particular layout/drivers
- *Flags* specify various properties of the filesys (e.g. read-only)

Example: `mount("/dev/disk5","/usr","ext3",MS_RDONLY,...)`

Memory Management

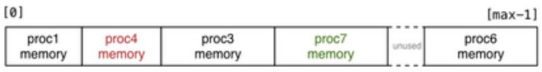


Splitting Process v Creating Space for Process

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

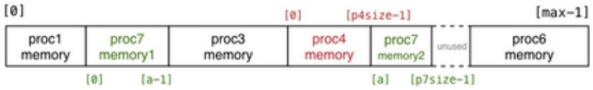
The new process doesn't fit in any of the unused slots

Could move some process to make a single large slot



Alternative strategy: split new process memory over two regions

becomes

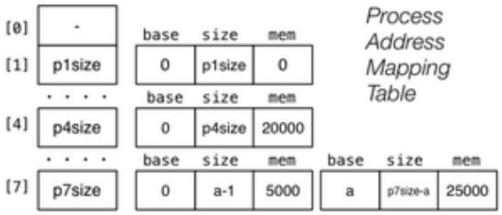


Splitting process memory:

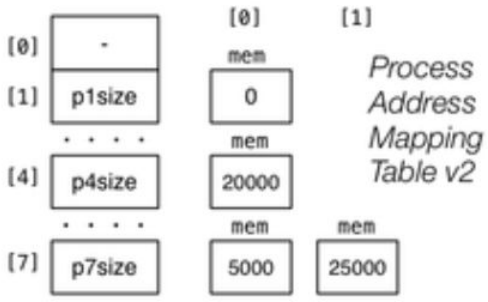
Each chunk of process address space has its own:

- base, size, memory location

Address mapping using split process:



If chunk size were uniform, address mapping would be simplified:



And mapping from process address → physical address is simplified:

```
Address processToPhysical(pid, addr)
{
   PageInfo pages[] = getPageInfo(pid);
   uint pageno = addr / PageSize;  // int div
   uint offset = addr % PageSize;
   return pages[pageno].mem + offset;
}
```

Virtual Memory

A side-effect of process → physical address mapping

- Don't need to load all of processes pages upfront
- Start with a small memory 'footprint'
- Load new process address pages into memory as needed
- Grow up to the size of the (available) physical memory

Virtual memory is the strategy of:

- Dividing process memory space into fixed-size pages
- On-demand loading of process pages into physical memory
- Process addresses = virtual addresses

Page Frames

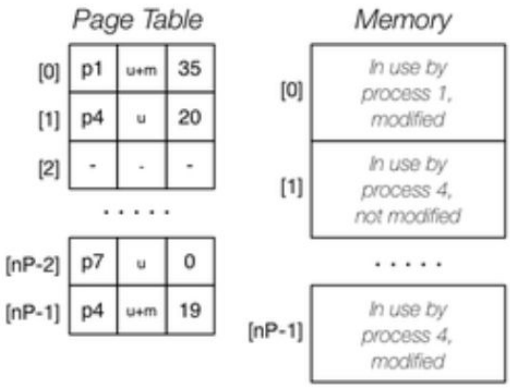
Page frames are page-sized regions of memory, typically 512B..8KB in size

- Each page frame contains a small region of a process' address space
- Memory layout is similar to this:

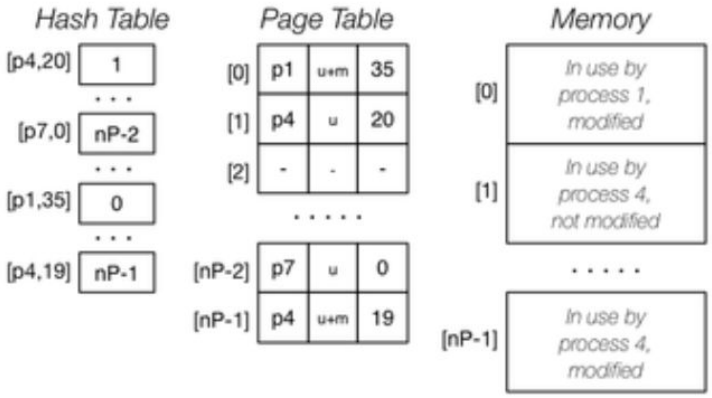
- When a process completes, all of its page frames are released for reuse

Virtual Memory Management

- Page table management which records:
  - Which process the page is allocated to
  - Whether the page is currently in use and modified
  - Which chunk it represents within the process' address space

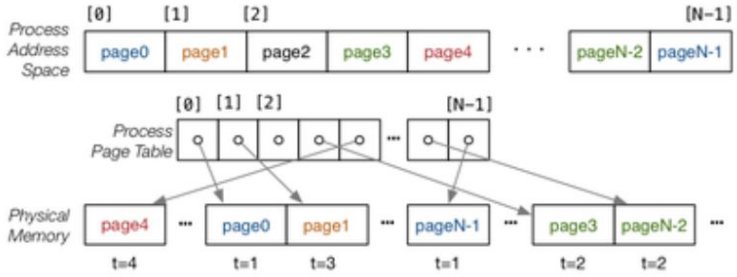


- However, since we need to search the table to find a page, hashing is a better option:



Summary:

- Process views its (virtual) address space as [0..max addr]
- Memory manager partitions it over fixed-size **pages**
- Process pages are loaded into memory when references
- Process **page table** gives mapping virtual → physical pages



Working Sets

In any given window of time, a process is likely to access only a small subset of its pages

- This gives that only the process' working set of pages needs to be held

Implications:

1. Can hold pages for many active processes in memory at same time (given smaller working sets)
2. Process address space can be larger than physical memory

Page Loading/Faults/Replacement

Pages are loaded from:

- Code is loaded from the executable file stored on disk
- Global data is also initially loaded from here
- Dynamic (head, stack) data is created in memory

Transferring pages between disk ←→ physical memory is very expensive

- Minimise reading/writing to disk

```
Address processToPhysical(pid, addr)
{
   PageData *ProcPageTable = PageTables[pid];
   int pageno = addr / PageSize;
   int offset = addr % PageSize;
   int frameno;  // which page in memory
   if (loaded(ProcPageTable[pageno].status))
      frameno = ProcPageTable[pageno].memPage;
   else
      // load page into a free frame → frameno ...
   return frameno * PageSize + offset;
}
```

Page Faults

- Requesting a non-loaded page is a page-fault
- Process
    - Find a free (unused) page frame in memory
    - If it can't find one:
        - Suspend the requesting process until a page is freed
            - Requires process manager to maintain priority queue of processes waiting for pages
        - Replace one of the currently loaded/used pages
            - If replaced file has been modified → save it to disk
            - Take its frame number and give it to the new process

Page Replacement

- Replace the page that:
    - Won't be used again by its process, preferably read-only, unmodified and used by only one process
- Common replacement strategies:
    - LRU - least recently used
    - FIFO - first in first out (front of queue)
    - Clock sweep - maintains circular list of allocated frames and replaces first-found frame in clock sweep

Processes and Signals

Processes

A process is an instance of an executing program

- Process is identified by a unique process ID (PID)
- Process has state:
  - Static - program code and data
  - Dynamic - heap, stack, registers, program counter
  - OS-supplied - environment variables, stdin, stdout
- Processes have:
  - Control-flow independence (process executes as though it's only process on the machine)
    - However, in reality, multiple processes are running on the machine simultaneously and each stops when it is pre-empted or exits
    - Pre-empted
      - If the program runs long enough and the OS replaces it by a waiting process, or if the process attempts to perform a long-duration task
      - The state is saved, process flagged as temporarily suspended and placed on PQ for restart
      - Once resumed, state is restored and the process starts at saved PC

  - Private address space

Process-related Commands and System calls

Information associated with processes:

- `pid` ...process id, unique among current processes
- `ruid`, `euid` ... real and effective user id
- `rgid`, `egid` ... real and effective group id
- current working directory
- accumulated execution time (user/kernel)
- user file descriptor table
- information on how to react to signals
- pointer to process page table
- process state ... running, suspended, asleep, etc.

This data is split across a kernel process table entry and a user area

Commands:

- **`sh`** ... for creating processes via object-file name
- **`ps`**, **`w`**, **`top`** ... show process information
- **`kill`** ... send a signal to a process

System calls:

- **`fork()`** ... create a new child process (copy of current process)
- **`execve()`** ... convert one process into another by executing object
- **`wait()`** ... wait for state change in child process
- **`kill()`** ... send a signal to a process
- **`_exit()`** ... terminate an executing process (after clean up)

`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 fd's
- child does not inherit copies of pending signals, memory locks, ...

Typically, the child pid is a small increment over the parent pid

---

Every process is allocated a process ID (PID)

- This is a positive integer, unique among currently executing processes
- Type is **pid_t** (defined in <unistd.h>)
- Process 0 is the scheduler
- Process 1 is init (for starting/stopping the system)
- Regular processes have PID in range (300..max PID) where max PID can equal 2^16
- Processes are also collected into process groups (PGID)
    - Groups allow distribution of signals to a set of related processes

`pid_t getpid()`

- returns the process ID of the current process

`pid_t getppid()`

- returns the parent process ID of the current process

`pid_t getpgid(pid_t pid)`

- returns the process group ID of specified process
- if `pid` is zero, use get PGID of current process

`int setpgid(pid_t pid, pid_t pgid)`

- set the process group ID of specified process

All return -1 and set `errno` on failure.

`pid_t waitpid(pid_t pid, int *status, int options)`

- pause 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 for any child in process group
    - if `pid` > 0, wait on the specified process
- information about child process state is stored in `status`
- `options` alters behaviour of `wait` (e.g. don't block)

`pid_t wait(int *status)`

- equivalent to `wait(-1, &status, 0)`

`int execve(char *Path, char *Argv[], char *Envp[])`

- replaces current process by executing `name` object
    - *`Path`* must be an executable, binary or script (starting with `#!`)
- passes arrays of strings to new process
    - both arrays terminated by a `NULL` pointer element
    - *`envp[]`* contains strings of the form `key=value`
- much of the state of the original process is lost, e.g.
    - a 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

`int kill(pid_t ProcID, int SigID)`

- send signal *SigID* to process *ProcID*
- various signals (POSIX) e.g.
  - SIGHUP ... hangup detected controlling terminal/process
  - SIGINT ... interrupt from keyboard
  - SIGILL ... illegal instruction
  - SIGFPE ... floating point exception (e.g. divide by zero)
  - SIGKILL ... kill signal (e.g. `kill -9`)
  - SIGSEGV ... invalid memory reference
  - SIGPIPE ... broken pipe (no processes reading from pipe)
- on error, returns -1 and sets `errno`
- if successful, returns 0

Signals

Signals can be generated from multiple sources:

- From a program via kill()
- From the OS
- From a device (I/O)

Processes can define how they want to handle signals:

- using `signal(int SigID, sighandler_t Handler)`
- *Handler* can be SIG_IGN, SIG_DFL or a function
- *SigID* is one of the OS-defined signals

Interrupts

Interrupts are signals which:

- Cause normal process execution to be suspended

Handlers:

- Carry out tasks related to the interrupt and hands control back to the original process

Multi-tasking & Scheduling

Multi-tasking = multiple processes are “active” at the same time

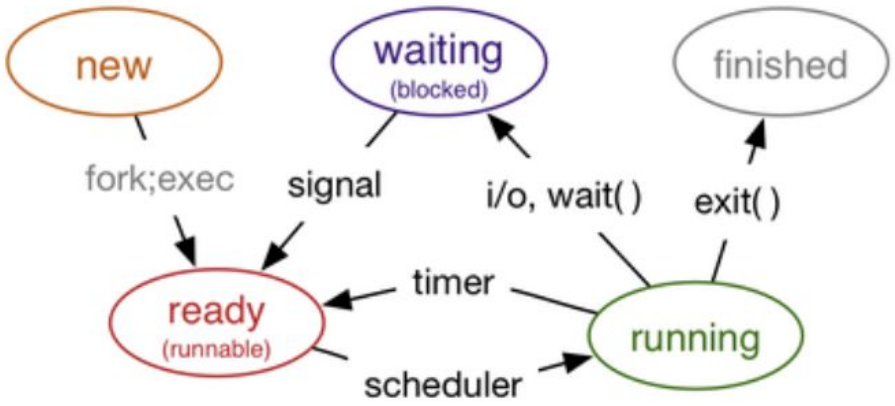
- Processes are not necessarily executing simultaneously (unless multiple CPUs)
- Usually:
  - One process is running (per CPU)
  - Some are runnable and ready to execute
  - Some are blocked waiting on a signal
- Switches between processes after each runs for a defined “time slice”

Scheduling = selecting which process should run next

- Processes are organised into priority queue(s)
  - Highest priority at head of queue
- Priority determined by many factors:
  - System process > user processes
  - Longer-running processes might have lower priority
  - Memory-intensive processes might have lower priority
  - Processes suggest their own priority
- Priority ranges from highest 0 → 139 lowest
- Abstract view of scheduler:

```
onTimerInterrupt()
{
   save state of currently executing process
   newPID = dequeue(runnableProcesses)
   setup state of newPID (e.g. load pages)
   transfer control to newPID (i.e. set PC)
}
```

- Process control block stores process information
    - ID data, state data, control data



I/O Devices

Input/Output devices:

- Allow programs to communicate with the “outside world”
- Much slower access time than memory-based data
- Transfer data in blocks (e.g. 512B or 4KB)

Examples include:

- Hard disks (address specified by track and sector)
- Network transfer (address specified by IP address)
- Keyboard, screen, mouse, camera, microphone

Unix treats devices as byte-streams (like files)

- Devices can be accessed via the file system under /dev

Memory Mapped I/O

- OS defines special memory address
- User programs perform I/O by getting/putting data into this memory address
- Virtual memory addresses are associated with I/O device’s data buffers and control registers
- Benefits:
    - Uses existing memory access logic circuits (less hardware)
    - Can use full range of CPU operations on device memory

Device Files

- Character devices
    - Provide direct access to hardware devices
    - Programmers write individual bytes
- Block devices
    - Provide buffered access to hardware devices
    - Programmers write chunks of bytes which is transferred to device via OS buffers

Buffered I/O

- Using a read() from a device is very inefficient
    - Instead, the OS uses a collection of buffers to hold data from a device, which can then be fed to user programs from the buffers without accessing the device each time

Device Drivers

A device driver is a code chunk to control an I/O device, and each type of device has its own unique access protocol

- Consists of:
    - Special control and data registers
    - Locations (buffers) for data to be read/written
- Often written in assembler
- Typical protocol:
    - Send request for operation
    - Receive interrupt when request is completed

I/O Related Commands

`int ioctl(int FileDesc, int Request, void *Arg)`

- manipulates parameters of special files (behind open *`FileDesc`*)
- *`Request`* is a device-specific request code,
- *`Arg`* is either an integer modifier or pointer to data block
- requires `#include <sys/ioctl.h>`, returns 0 if ok, -1 if error

Example: SCSI disk driver

- `HDIO_GETGEO` ... get disk info in (heads,sectors,cylinders,...)
- `BLKGETSIZE` ... get device size in sectors
- in both cases, *`Arg`* is a pointer to an appropriate object

`int open(char *PathName, int Flags)`

- attempts to open file/device *`PathName`* in mode *`Flags`*
- *`Flags`* can specify caching, async i/o, close on `exec()`, etc.
- returns file descriptor if ok, -1 (plus `errno`) if error

`ssize_t read(int FileDesc, void *Buf, size_t Nbytes)`

- attempts to read *`Nbytes`* of data into *`Buf`* from *`FileDesc`*
- returns # bytes actually read, 0 at EOF, -1 (plus `errno`) if error

---

`ssize_t write(int FileDesc, void *Buf, size_t Nbytes)`

- attempts to write *`Nbytes`* of data from *`Buf`* to *`FileDesc`*
- returns # bytes actually written, -1 (plus `errno`) if error

Exceptions

Exceptions are:

- Unexpected conditions occurring during program execution which require some action
- Two types:
    - Internal Faults
        - Often fatal to continued execution of program
        - Examples:

```
SIGILL ... illegal instruction
SIGABRT ... generated by abort()
SIGFPE ... floating point exception
SIGSEGV ... invalid memory reference
```

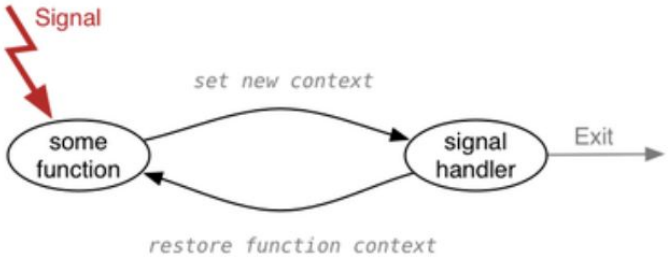
    - External Faults
        - Often require some action and then execution can continue

```
SIGINT ... interrupt from keyboard   (handled by Terminate)
SIGPIPE ... broken pipe   (handled by Terminate)
SIGCHLD ... child process stopped or died   (handled by Ingore)
SIGTSTP ... stop typed at tty (control-Z)   (handled by Stop)
```

- Many exceptions manifest themselves via signals
    - Term - terminate the process
    - Ign - ignore the signal
    - Core - terminate the process, dump core
    - Stop - stop the process
    - Cont - continue the process if currently stopped
- See "**man 7 signal**" for details of signals and default handling

Signal Handling

A function invoked in response to a signal



Interacting Processes

Processes can interact in several ways:

- Accessing the same resource
- Sending signals (via kill())
- Pipes (process A | process B)
- Sockets (message passing)

Interaction needs to be controlled to avoid unpredictable results

File Locking

`flock(int FileDesc, int Operation)`

- 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
- `flock()` only *attempts* to acquire a lock
  - if it can't acquire the lock now, it is blocked (suspended)
  - when it can acuqire the lock, `flock()` returns
- only works correctly if all processes accessing file use locks
- return value: 0 in success, -1 on failure

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

- if file not locked or 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

Concurrency

Concurrency is multiple processes running (pseudo) simultaneously, rather than sequential execution

- Increases system throughput
- If one process is delayed, others can run
- If we have multiple CPUs, can use all at once

If concurrency goes wrong:

- Nondeterminism (same code, different runs, different results)
- Deadlock (a group of processes end up waiting for each other)
- Starvation (one process keeps missing access to resource)

Concurrency Control

Concurrency control aims to:

- Provide correct sequencing of interactions between processes
- Coordinate access to shared resources

Two broad classes of concurrency control schemes:

- Shared memory based (e.g. semaphores)
    - Uses shared variable, manipulated atomically
    - Blocks if access unavailable, decrements once available
- Message passing based (e.g. send/receive)
    - Processes communicate by sending/receiving messages
    - Receiver can block waiting for message to arrive
    - Sender may block waiting for message to be received
        - Synchronous - sender waits for acknowledge of receipt
        - Asynchronous - sender transmits and continues

Semaphores

`#include <semaphore.h>`, giving `sem_t`

`int sem_init(sem_t *Sem, int Shared, uint Value)`
- create a semaphore object, and set initial value

`int sem_wait(sem_t *Sem)` (i.e. wait())
- try to decrement, block if `Sem == 0`
- has variants that don't block, but return error if can't decrement

`int sem_post(sem_t *Sem)` (i.e. signal())
- increment the value of semaphore `Sem`

`int sem_destroy(sem_t *Sem)`
- free all memory associated with semaphore `Sem`

Message Passing

`#include <mqueue.h>`, giving `mqd_t`

`mqd_t mq_open(char *Name, int Flags)`
- create a new message queue, or open existing one

`int mq_send(mqd_t *MQ, char *Msg, int Size, uint Prio)`
- adds message `Msg` to message queue `MQ`
- `Prio` gives priority; blocks if `MQ` is full

`int mq_receive(mqd_t *MQ, char *Msg, int Size, uint *Prio)`
- removes oldest message with priority `*Prio` from queue `MQ`
- blocks if `MQ` is empty; can run non-blocking

`int mq_close(mqd_t *MQ)`
- finish accessing message queue `MQ`

Networks

A network is an interconnected collection of computers used mostly for communication/transfer of data

- Network types include:
    - Local area networks - within an organisation/physical location
    - Wide area networks (WAN) - geographically dispersed
    - Internet - global set of interconnected WANs

How a network communicates files:

- File data divided into packets by source device
    - Packets are small fixed-size chunks of data, with headers
- Passed across physical link (wire, radio, optic fibre)
- Pass through multiple nodes (routers, switches)
    - Each node decides where to send it next
- Packets reach destination device
- Re-ordering, error-checking and buffering occur
- File received by receiving process/user

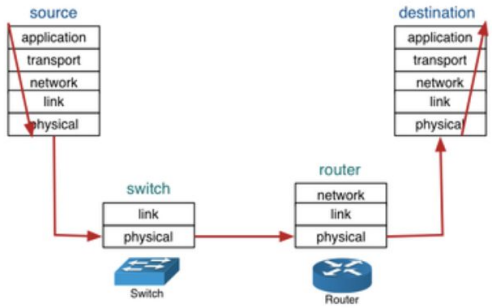
The Internet

Components:

- Connected devices
    - PC, server, phone
    - Host = end system, running network apps
- Communication links
    - Fibre, copper, radio, satellite
    - Bandwidth = transmission rate
- Packet switches
    - Routers, network switches
    - Responsible for computing the next hop, forward packets

5-layer Stack

- Physical layer - bits on wires/optics/radio
- Link layer - ethernet, addressing
- Network layer - routing protocols, IP
- Transport layer - process-process data transfer
    - Deals with data integrity, timing, throughput and security
- Application layer - DNS, HTTP, email etc.



Protocols

Network protocols govern all communication activity on the network

- Format and order of messages sent/received
- Actions taken on message transmission/receipt

Each application layer protocol defines:

- Types of messages - possible requests/responses
- Message syntax - what fields are in messages
- Message semantics - meaning of information in fields
- Processing rules - when and how processes respond to messages

TCP (Transmission Control Protocol)

- Provides a logical communication between processes on different hosts
    - Reliable transport, flow control, congestion control and is connection-oriented

Application Layer

The application layer directly supports the appswe interact with, e.g.

- Email, web, text, login, games, etc.

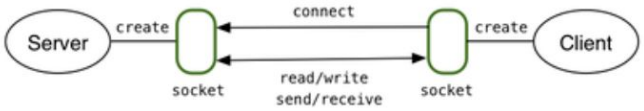
Client-Server Architecture

- Server is a data provider
    - Waits for requests
    - Is always on with permanent IP address
- Client is a data consumer
    - Sends requests to server, collects response
    - May be intermittently connected with dynamic IP address
    - Does not communicate directly with other clients

Unix Sockets

An endpoint of an inter-process communication channel

- Commonly used to construct client-server systems
- Server creates a socket, binds to an address, listens for connections
- Client creates a socket, connects to the server, reads/writes



`int socket(int Domain, int Type, int Protocol)`

- requires `#include <sys/socket.h>`, sockets are `int`s (like fds)
- creates a socket, using ...
    - ***`Domain`*** ... communications domain
        - `AF_LOCAL` ... on the local host (Unix domain)
        - `AF_INET` ... over the network (Internet domain)
    - ***`Type`*** ... semantics of communication
        - `SOCK_STREAM` ... sequenced, reliable communications stream
        - `SOCK_DGRAM` ... connectionless, unreliable packet transfer
    - ***`Protocol`*** ... communication protocol
        - many exist (see `/etc/protocols`), e.g. IP, TCP, UDP, ...
- returns a socket descriptor or -1 on error

`int bind(int Sockfd, SockAddr *Addr, socklen_t AddrLen)`

- associates an open socket with an address
- for Unix Domain, address is a pathname in the file system
- for Internet Domain, address is IP address + port number

`int listen(int Sockfd, int Backlog)`

- wait for connections on socket *`Sockfd`*
- allow at most *`Backlog`* connections to be queued up

`SockAddr` **`= struct sockaddr_in`**

- C struct containing components of "visible" socket address

```
int accept(int Sockfd, SockAddr *Addr, socklen_t *AddrLen)
```

- *Sockfd* has been created, bound and is listening
- blocks until a connection request is received

- sets up a connection between client/server after `connect()`
- places information about the requestor in *Addr*
- returns a new socket descriptor, or -1 on error

```
int connect(int Sockfd, SockAddr *Addr, socklen_t AddrLen)
```

- connects the socket *Sockfd* to address *Addr*
- assumes that *Addr* contains a process listening appropriately
- returns 0 on success, or -1 on error

Addressing

Server processes must have a unique Internet-wide address

- Part of address is IP address of host machine
- Other part of address is port number where server listens

Some standard port numbers

- 22 ... ssh (Secure Shell)
- 25 ... smtp (Simple Mail Transfer Protocol)
- 53 ... dns (Domain Name System)
- 80 ... http (Web server)
- 389 ... ldap (Lightweight Directory Access Protocol)
- 443 ... https (Web server (encrypted))

- 5432 ... PostgreSQL database server

IP Addresses

- Unique identifier for host on network
- Given as a 32-bit identifier (soon to be 128-bit addresses as we are running out of unique IP addresses for each device)

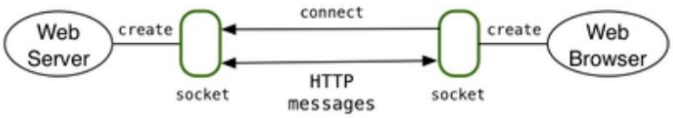
HTTP Protocol

HTTP = HyperText Transfer Protocol

- Drives the web
- Message types: URL (request) and web pages (response)
- Message syntax: headers + data

Transport layer view of HTTP application layer:

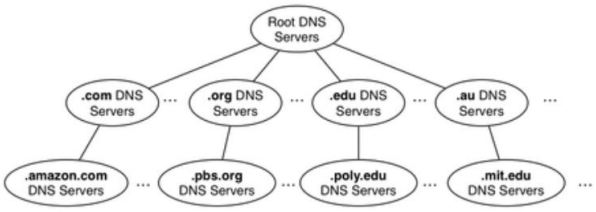
- using TCP
- client initiates TCP connection (socket) to server, port 80
- server accepts TCP connection from client
- client sends HTTP request messages (e.g. GET)
- server responds with HTTP messages (e.g. HTML)
- interaction completes, connection (socket) closed



Server Addresses (DNS)

Network requests typically use server names

- DNS = Domain Name System which provides name → IP address mapping



Link Layer

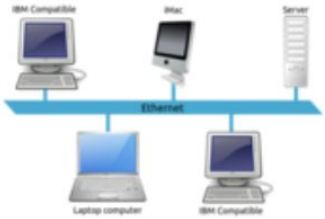
Link layer takes packets from network layer and transmits them

- Provides:
  - Flow control
  - Error detection
  - Error correction

Ethernet

An example of a link layer implementation

- Ethernet is a cable physically connecting multiple hosts
- Data broadcast onto cable, tagged with receiver MAC address
- Devices recognise their own data using MAC address



MAC address = Media Access Control address, stored in NIC