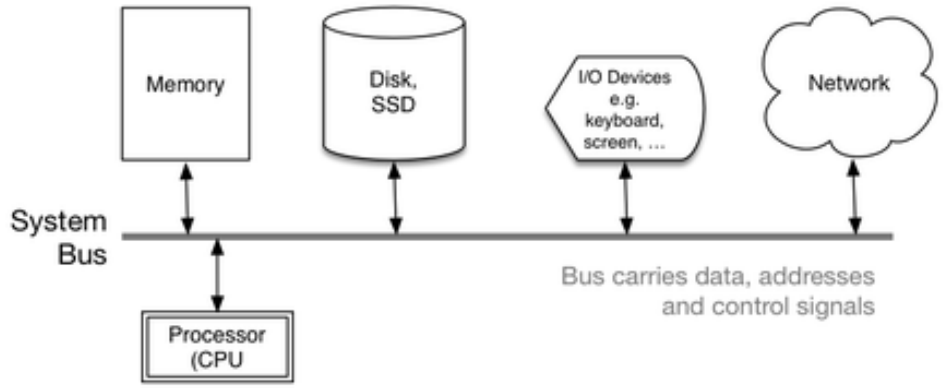
Computer Systems, C Program Life-Cycle

Tuesday, 24 July 2018 9:23 PM

All information for this course will come from Jas Shepard's notes

Computer Systems

Here is a component view of a typical modern computer system:



Processor

Modern processors provide:

- Control, arithmetic, logic, bit operators
- Relatively small sets of simple instructions
- Small amounts of very fast storage (registers)
- Small no. of control registers (e.g. PC)
- Fast fetch-decode-execute cycles (nanoseconds***/ns***)
- Access to system bus to communicate the other components

all integrated on a single chip (without considering multi-core CPUs in this course

Storage

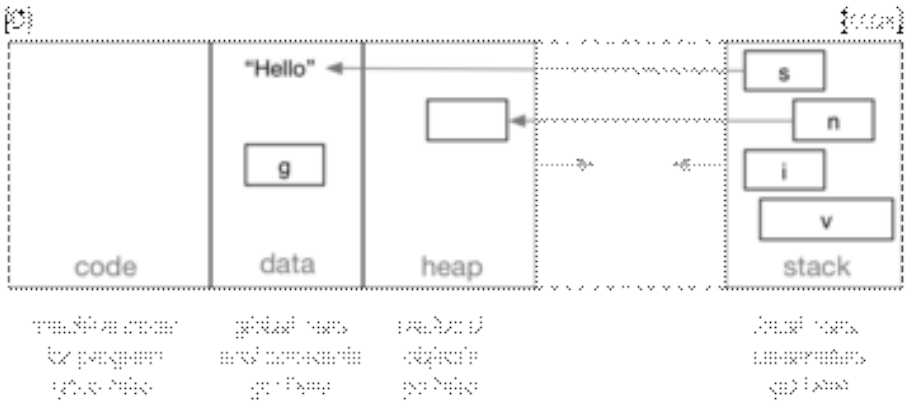
(Main) memory:

- Consists of very large random-addressable arrays of bytes
- Can fetch single bytes into CPU registers
- Can fetch multi-byte chunks into CPU (e.g. 4-byte int)
- Typically has access time of 70 nanoseconds and size 64GB

Disk storage:

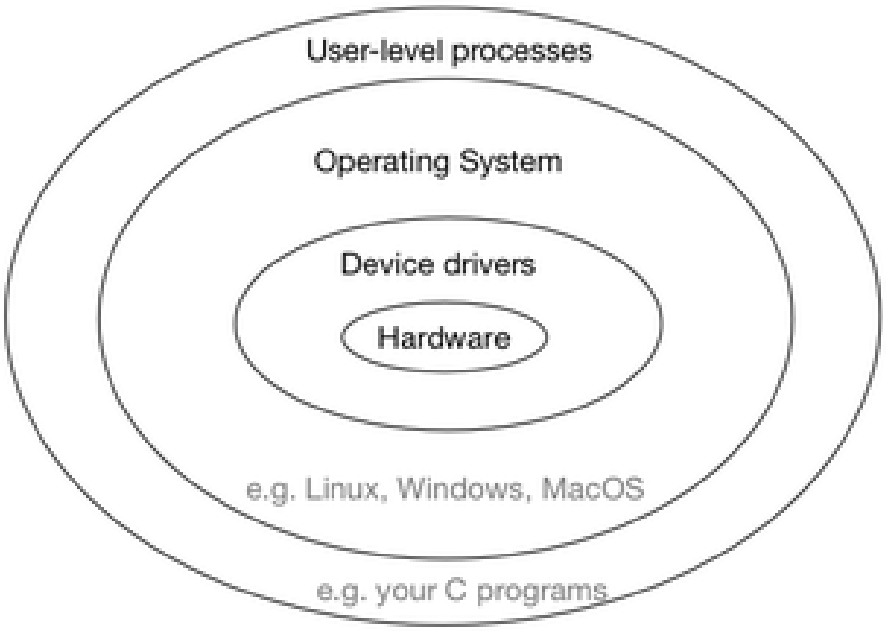
- Consists of very very large block-oriented storage
- Often on spinning disk, fetching 512-4KB per request
- Typically has access time of 30 milliseconds, and size 8TB
- Nowadays, we has SSD, which has access time of 0.8 milliseconds, size 1GB

Run-time memory usage depends on the language processor. A typical C compiler uses the memory like this:



Computer System Layers

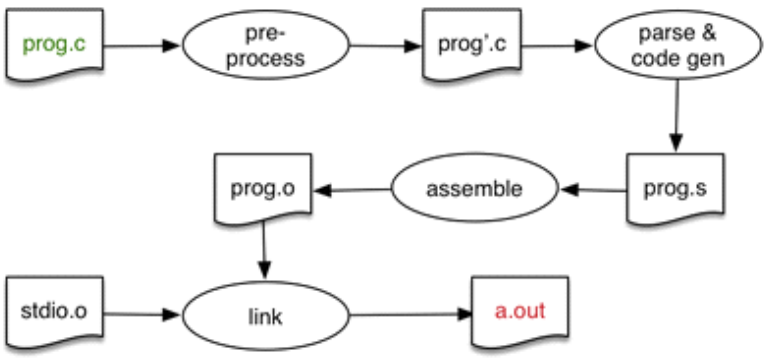
A diagrammatical view of software layers in a typical computer system:



C Program life-cycle

Our C program starts a text - well-designed, readable and maintainable. Ultimately they are executed on a CPU as machine code, efficiently producing correct results, handling error conditions robustly and utilising services of underlying operating systems.

Our C source code is mapped to machine code like this:



1. A pre-process takes our `prog.c` and modifies it into a `prog'.c` (a different .c program), before it is even compiled. (All `#includes` get expanded in place, all `#defines` get mapped into our code)
   To get `prog'.c` run **`gcc -E prog.c`**
2. Modified program is compiled to `prog.s`. This includes parsing, making sure it looks like a valid C program and generating some code (assembly language code is initially generated)
   To get `prog.s` run **`gcc -S prog'.c`**
3. Assembly language code is not really machine code because it still has some symbolic stuff, so it is them assembled to `prog.o` (machine code that requires some bits to be filled out)
   To get `prog.o` run **`gcc -c prog.s`**
4. Link fills in the bits needed to form `a.out`
   To get `prog.o` run **`gcc -o prog.o`**

In general: initial C code -> pre-process modifies initial code -> parsed to assembly code -> turned into machine code -> linked with libraries -> executable

```
If you only want some of the stages of compilation, you can use -x (or filename suffixes) to
tell gcc where to start, and one of the options -c, -S, or -E to say where gcc is to stop.
Note that some combinations (for example, -x cpp-output -E) instruct gcc to do nothing at all.

-c  Compile or assemble the source files, but do not link.  The linking stage simply is not
    done.  The ultimate output is in the form of an object file for each source file.

    By default, the object file name for a source file is made by replacing the suffix .c, .i,
    .s, etc., with .o.

    Unrecognized input files, not requiring compilation or assembly, are ignored.

-S  Stop after the stage of compilation proper; do not assemble.  The output is in the form of
    an assembler code file for each non-assembler input file specified.

    By default, the assembler file name for a source file is made by replacing the suffix .c,
    .i, etc., with .s.

    Input files that don't require compilation are ignored.

-E  Stop after the preprocessing stage; do not run the compiler proper.  The output is in the
    form of preprocessed source code, which is sent to the standard output.

    Input files that don't require preprocessing are ignored.

-o file
    Place output in file file.  This applies to whatever sort of output is being produced,
    whether it be an executable file, an object file, an assembler file or preprocessed C code.

    If -o is not specified, the default is to put an executable file in a.out, the object file
    for source.suffix in source.o, its assembler file in source.s, a precompiled header file in
    source.suffix.gch, and all preprocessed C source on standard output.
```

History of compilers

Milestones in C history

- cc ... original compiler by Dennis Ricthie (1971)
- gcc ... open source compiler by Richard Stallman et al (1987)
- gcc 2.0 ... added C++ compilation (1992)
- clang ... gcc replacement by Apple et al (2007)
- dcc ... Python wrapper on clang by Andrew Taylor (2012?)
    - augments error messages to be more helpful to novices
    - incorporates useful run-time checking to help debugging

C Review

Tuesday, 24 July 2018 9:31 PM

Assumed knowledge:

- design an algorithmic solution
- describe your solution in C code, using ...
  - variables, assignment, tests (==, !, <=, &&, etc)
  - `if, while, for, break, scanf(), printf()`
  - functions, return, prototypes, *.h, *.c
  - arrays, files, structs, pointers, `malloc()`, `free()`

Not-assumed knowledge:

- Linked lists
- ADTs
- Sorting
- Recursion
- Bit operations

Type Definitions

Reminder: you can give a name to a C type definition e.g.

```
typedef int Integer;
typedef long long int BigInt;
typedef unsigned char Byte;
typedef struct { int x; int y; } Coord;
```

and then use the name instead of the type definition e.g.

```
Byte  byte;
Coord here, there;
```

Assignment as Expression

Assignments can be treated as an expression returning a value

```
x = 5           // returns 5
```

It can be useful as in here:

```
x = y = z = 0   // equivalent to x = (y = (z = 0))
```

And it can be dangerous

```
if (x = 0)      // "test" always fails
```

Assignment-as-expression is often used to simplify loops. e.g.

```
while ((ch = getchar()) != EOF) {
    // do something with ch
}
```

rather than

```
ch = getchar();
while (ch != EOF) {
    // do something with ch
    ch = getchar();
}
```

We often ignore the value returned by an assignment. Some C functions return a result which is often ignored

```
int x = 123, y = 42;
printf("%d %d\n", x, y);
// printf() returns the number of characters printed.
// which is 7 (123 42\n)
```

Return values of common C functions

`fscanf()`	On success, returns of the no. input items successfully ***matched*** and ***assigned***; this can be fewer than provided for, or even zero, if there is an early matching failure.	The value EOF is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. EOF is also returned if a read error occurs.
`fprintf()`	On success, returns the no. of characters printed (excluding the null byte used to end output to string bytes).	If an error is encountered, a negative value is returned.
`fgetc()`	Returns the character written as an `unsigned char` cast to an `int`	Returns EOF on end of file or error.
`fputc()`	Returns the character written as an `unsigned char` cast to an `int`	Returns EOF on end of file or error.
`fgets()`	Returns string on success	Returns NULL on error or when end of file occurs while no characters have been read
`fputs()`	Returns a non-negative number on success	Returns EOF on error

Fine Control

`break` and `continue` provide fine loop control

- `break` exits the innermost enclosing loop
- `continue` goes directly to the next iteration

Both are not strictly needed in programming because the same effect can be achieved using ifs inside the loop. Using them is sometimes considered "bad style" but, when they are used carefully, they can make code easier to understand.

```
for (i = 0; i < N; i++) {
   if (a[i] == 0) break;
   if (a[i] < 0) continue;
   sum += a[i];
}
/*  vs */
for (i = 0; i < N && a[i] != 0; i++) {
   if (a[i] > 0) sum += a[i];
}
```

Fine Function Control

`return` returns the result of a function. Good style says "have a single return at end" However ...

- a function can contain several returns
- each can capture a specific condition to complete function

When used carefully, it can make functions easier to understand.

Fine Program Control

Good style says "Program returns at end of main()". However, exceptional conditions may require early exit. `exit()` terminates program with return code. It can be used from anywhere in the program (even deeply nested functions). `assert()` can also be used to trigger program exit

```
assert(Condition);
if (!Condition) exit(1);
```

Return values not related to function semantics are typically success/failure statuses. For example `main()` returns zero (success) and non-zero (error). Ignored return error statuses are generally extremely unlikely to occur or they are not fatal if the failure occurs.
If there is an error it will set the value of `errno` to the number of the last error. If you want to find out what the error is, `perror()` prints the error based on the value errno is set to
Note: `errno` is a global variable which is available in every C program you write and `perror()` is a function

Stacks, Queues, Priority Queues

Tuesday, 24 July 2018 11:17 PM

Computer systems frequently make use of:

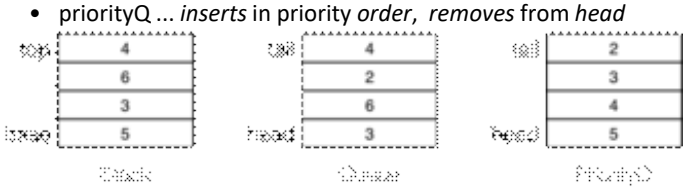
- *stacks* ... last-in-first-out lists
  - e.g. used to represent function local variables
- *queues* ... first-in-first-out lists
  - e.g. used to ensure fair access to a resource
- *priority queues* ... highest-priority-out lists
  - e.g. used to represent weighted lists of processes

All of these data structures have:

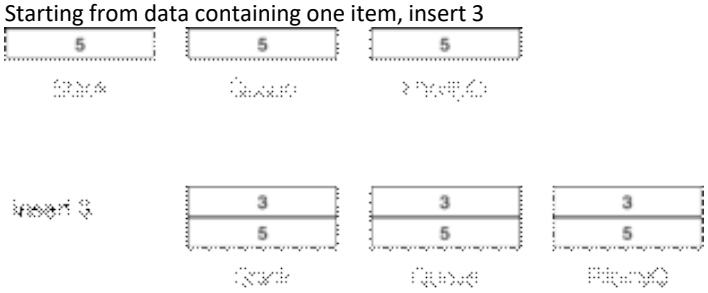
- A ~~sequence~~ of items
- An operation to *insert* an item into the sequence
- An operation to *remove* an item from the sequence

Data structures differ in

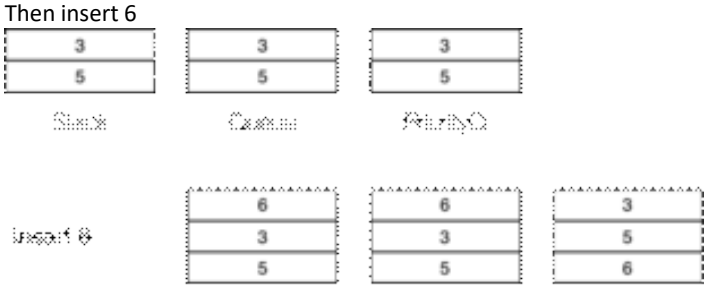
- stack ... *insert* adds on *top*, *removes* from *top*
- queue ... *insert* adds at *tail*, *removes* from *head*
- priorityQ ... *inserts* in priority *order*, *removes* from *head*



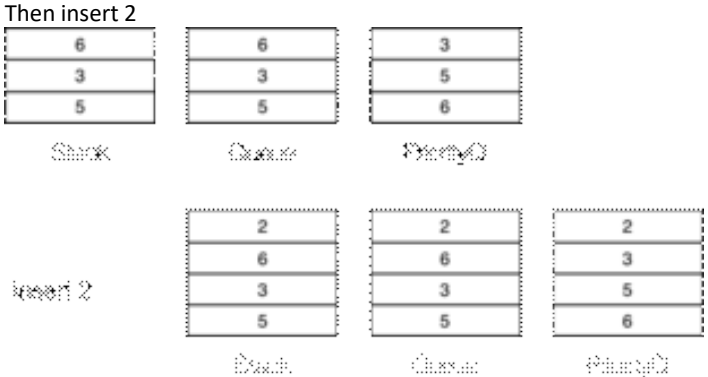
Starting from data containing one item, insert 3



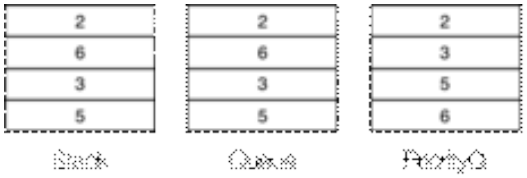
Then insert 6



Then insert 2



Then remove an item



Stack Data Structure

Stack: Last-in, First-out (LIFO) protocol

- insert operation called `push()`
- remove operation called `pop()`
- has a top (last item added) and a size

Stack interface:

```
typedef struct _stack { ... } Stack;
void initStack(Stack *s);
int  pushStack(Stack *s, Item val);
Item popStack(Stack *s);
int  isEmptyStack(Stack s);
...
```

Example Stack Client

A program to use a stack of `int` values

```
#include <stdio.h>
#include "Stack.h"
int main(void)
{
     Stack myStack;
     int   x;

     initStack(&myStack);
     while (scanf("%d",&x) == 1) {
          if (!pushStack(&myStack, x)) break;
     }
     while (!isEmptyStack(myStack)) {
          printf("%d\n", popStack(&myStack));
     }
        return 0;
}
```

Queue Data Structure

Queue: First-in, First-out (FIFO) protocol

- insert operation called enter (or enqueue())
- remove operation called leave (or dequeue())
- has a head (first item added), a tail (last item added), and a size

Queue interface:

```
typedef struct _queue { ... } Queue;
void initQueue(Queue *q);
int  enterQueue(Queue *q, Item val);
Item leaveQueue(Queue *q);
int  isEmptyQueue(Queue q);
...
```

Priority Queue Data Structure

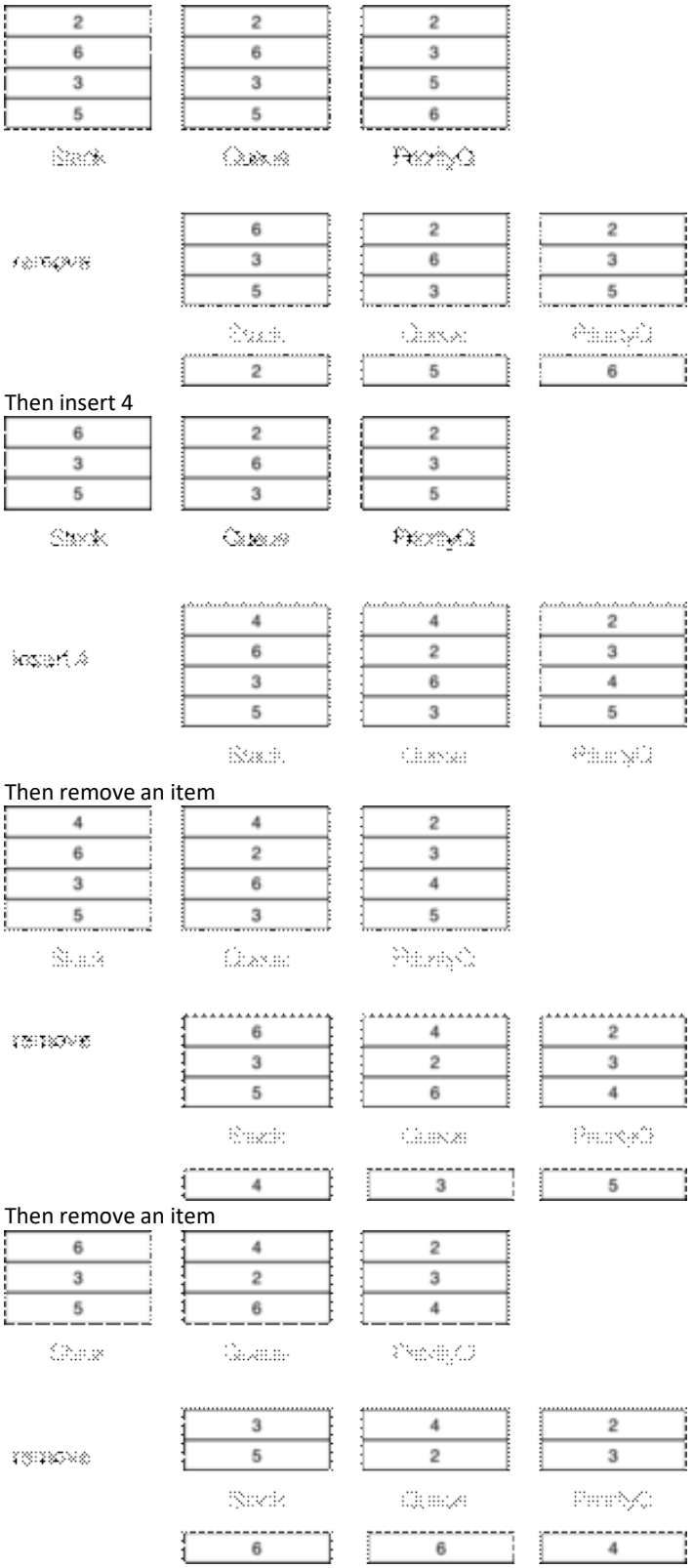
Queue: Highest-priority-out protocol

- insert operation called enter (or enqueue())
- remove operation called leave (or dequeue())
- has a head (next for removal), a tail (lowest-priority), and a size

Priority Queue interface:

```
typedef struct _priorityq { ... } PriorityQ;
void initPriorityQ(PriorityQ *q);
int  enterPriorityQ(PriorityQ *q, Item val);
Item leavePriorityQ(PriorityQ *q);
int  isEmptyPriorityQ(PriorityQ q);
...
```

```
int  enterPriorityQ(PriorityQ *q, Item val);
Item leavePriorityQ(PriorityQ *q);
int  isEmptyPriorityQ(PriorityQ q);
...
```



Bit Manipulation

Tuesday, 24 July 2018 11:43 PM

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.

C provides a set of operators that act bit-by-bit on pairs of bytes.
E.g. (10101010 & 11110000) yields 10100000 (bitwise AND)

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

Binary Constants

C does not have a way of directly writing binary numbers. Numbers can be written in **decimal**, **hexadecimal** and **octal**.
In hexadecimal, each digit represents 4 bits.

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

In octal, each digit represents 3 bits.

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

Bitwise AND

The & operator takes ***two values*** (1, 2, 4, 8 bytes), and treats them as a sequence of bits. It performs logical **AND** on each corresponding pair of bits. The result contains the same number of bits as input.
Example:

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

It is often used for checking whether a bit is set.

Bitwise OR

The | operator takes ***two values*** (1, 2, 4, 8 bytes), and treats them as a sequence of bits. It performs logical **OR** on each corresponding pair of bits. The result contains the same number of bits as input.
Example:

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

It is often used for ensuring that a bit is set

Bitwise XOR

The ^ operator takes ***two values*** (1,2,4,8 bytes), and treats them as sequence of bits. It performs logical **XOR** on each corresponding pair of bits. The result contains same number of bits as inputs.
Example:

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

It is often used in generating random numbers, and building adder circuits.

Bitwise NEG

The ~ operator takes ***a single value*** (1,2,4,8 bytes), and treats them as sequence of bits. It performs logical **negation** of each bits. The result contains same number of bits as inputs.
Example:

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

Used for e.g. creating useful bit patterns

Left Shift

The << operator takes a single value (1, 2, 4, 8 bytes)

Right Shift

The >> operator takes a single value (1, 2, 4, 8 bytes)

Left Shift

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

Example:

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

** if it is a signed quantity, whatever (signed) bit is at the top (the first bit) replaces left-end bit

Makefiles and Multi-module C Programs

Tuesday, 31 July 2018 11:14 PM

All large systems written in C are built as a large number of files, which are compiled separately and then combined to form an executable.

While developing a large system, you edit one .c file at a time, recompile just that .c file, then combine all .o files again. If you edit several .c files, you need to keep track of which ones you changed, then compile all the modified files and then compile all .o files again.

Within a large software system, ***dependencies*** exist between files. For example `x.o` depends on `x.c` and `y.h`. ***Actions*** state how to produce *targets* from *sources*. e.g. you can create `x.o` from `x.c` and `y.h` using gcc. ***Rules*** combine dependencies and actions. If `x.c` or `y.h` changes, you rebuild `x.o` with `gcc`.

A `Makefile` contains definitions and rules. The `make` command uses a `Makefile` to rebuild a system. If you write `make` without a target on the command-line, `make` will try to make the first target. If any of the *actions* fail the while `make` stops.

Example `Makefile`:

```
bm : bm.o Stack.o
    gcc -o bm bm.o Stack.o
bm.o : bm.c Stack.h
    gcc -c -Wall -Werror bm.c
Stack.o : Stack.c Stack.h
    gcc -c Wall -Werror Stack.c
```

```
CC = gc
CFLAGS = -Wall -Werror
bm : bm.o Stack.o
    gcc -o bm bm.o Stack.o
bm.o : bm.c Stack.h

Stack.o : Stack.c Stack.h
```

Legend: target, source, action

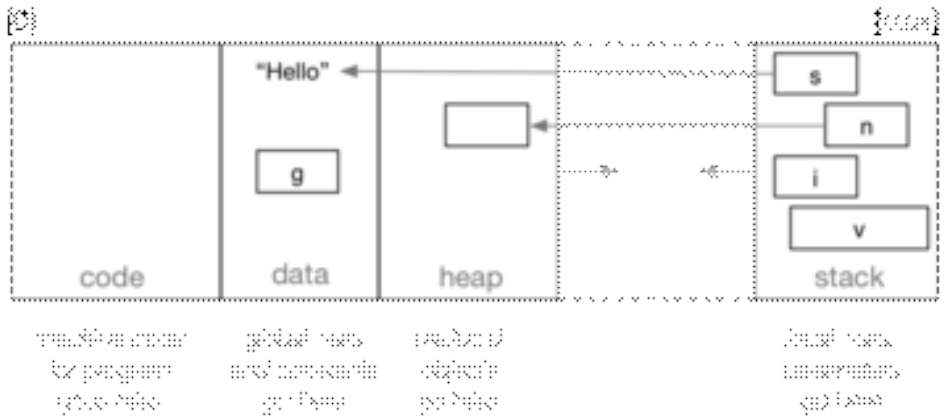
Memory

Tuesday, 24 July 2018 11:43 PM

The C View of Data

A C program sees data as a collection of *variables*. Each variable has a number of properties (e.g. name, type, size). Variables are examples of *computational objects*. Each computational object has:

- A **location** in memory
- A **value** (ultimately just bit-string)
- A **name** (unless created by `malloc()`)
- A **type**, which determines:
  - Its size
  - How to interpret its value
  - What operations apply to the value
- A **scope** (where it is visible within the program)
- A **lifetime** (during which part of program execution it exists)



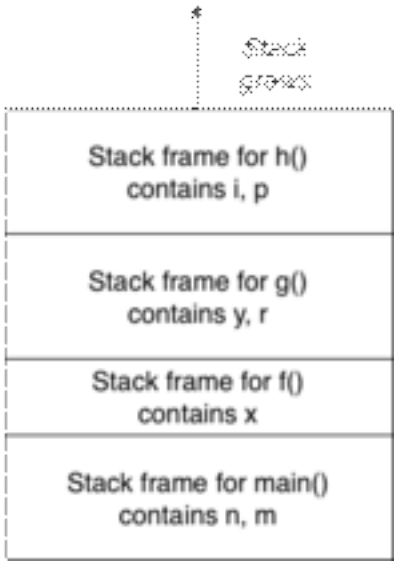
The ***global data*** area contains global variables, static variables, string constants. These objects persist for the entire duration of program execution.

The ***heap*** contains objects created by `malloc`, `calloc`,… These objects persist in the head until they are explicitly `free`'d.

The ***stack*** contains a small region (frame) for each active function. The frames contain local variables and parameters. A frame is created when a function is called and the frame is removed when the function returns.

Here is an example of runtime stack during a call to function h()

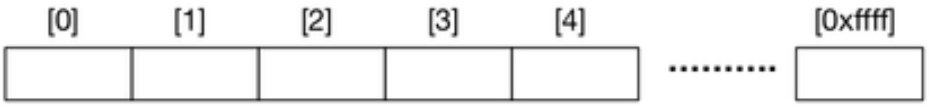
```
int main() {
    int n, m;
    n = 5;   m = f(n);
}
int f(int x) {
    return g(x);
}
int g(int y) {
    int r = 4 * h(y);
    return r;
}
int h(int z) {
    int i, p = 1;
    for (i=1; i<=z; i++)
        p = p * i;
    return p;
}
```



The Physical View of Data

Memory is essentially an indexed array of bytes.

Each byte contains 8 bits



Memory is a very large array of bytes

Indexes are "memory addresses" (a.k.a pointers) and data can be fetched in chunks of 1, 2, 4, 8 bytes.

Memory

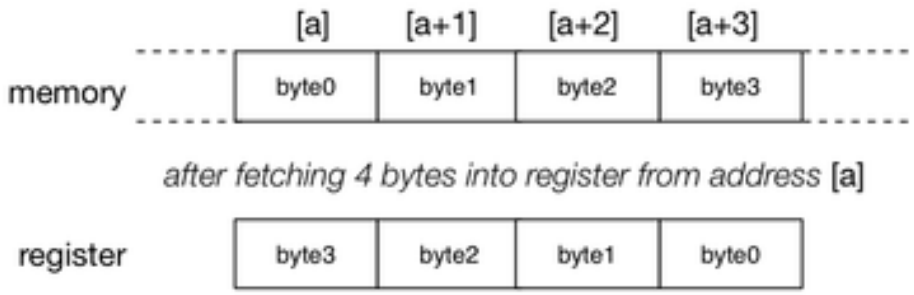
Memory is a semiconductor-based technology. It is also called RAM, main memory, primary storage.
Some distinguishing features about memory:

- It is relatively large (e.g. 2^{28} bytes)
- Any byte can be fetched with the same cost.
- The cost of fetching 1, 2, 4, 8 bytes is small (within nanoseconds)

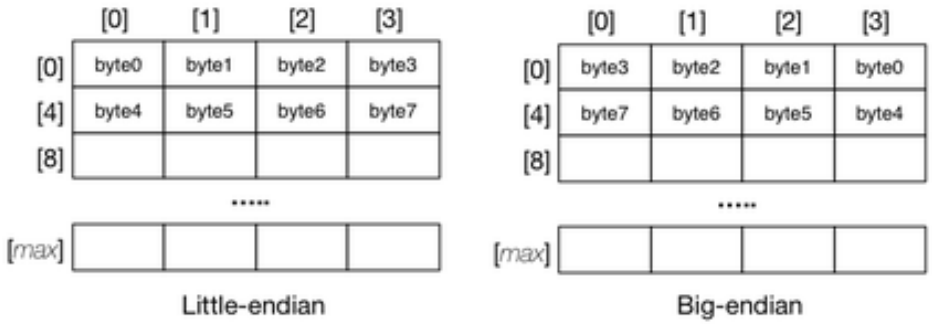
Two properties related to data persistence:

- Volatile (e.g. DRAM) .. Data is lost when powered off
- Non-volatile (e.g. EEPROM) .. Data stays when powered off

When addressing objects in memory, any byte address can be used to fetch a 1-byte object. The byte address for *N*-byte object must be divisible by *N*.
Data is fetched into *N*-byte CPU registers for use.
Data bytes in registers may be in different order to memory, e.g.



Memory can be categorised a *big-endian* or *little-endian*.



Loading a 4-byte int from address 0 gives

Data Representation

Tuesday, 24 July 2018 11:44 PM

Ultimately, memory allows you to load bit-strings of size 1, 2, 4, 8 bytes from *N*-byte boundary addresses into registers in the CPU.
What you are presented with is a string of 8, 16, 32, 64 bits. We need to ***interpret*** this bit-string as a meaningful value. Data representations provide a way of assigning meaning to bit-strings.

Character Data

Character data has several possible representations (encodings)
The two most common are:

- ASCII (ISO 646)
  - 7-bit values, using lower 7-bits of a byte (top bit always zero)
  - can encode roman alphabet, digits, punctuation, control chars
- UTF-8 (Unicode)
  - 8-bit values, with ability to extend to multi-byte values
  - can encode all human languages plus other symbols (e.g. √ ∑ ∀ ∃ or emojis)

ASCII Character Encoding

ASCII uses values in the range 0x00 to 0x7F (0..127)
The characters are partitioned into sequential groups

- 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'

In C, we can map between `char` and ASCII code by e.g. `((int) 'a')`
The sequential nature of groups allow for e.g. `(ch - '0')`

Hexadecimal ASCII char table (from `man 7 ascii`)

```
00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

```
0x0a = '\n',   0x20 = ' ',   0x09 = '\t',   but note no EOF
```

Unicode

Unicode is a widely-used standard for expressing "writing systems" (not all writing systems use a small set of discrete symbols). It is 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 only need 7-bits) so more compact character encodings have been developed (e.g. UTF-8).

UTF-8 uses a variable-length encoding as follows:

#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 ...

UTF-8 examples:

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

Unicode strings can be manipulated in C (e.g. "안녕하세요")
Like other C strings, they are terminated by a 0 byte (i.e. `'\0'`)

Unicode constants in C strings ...
The following two notations work in some contexts

- **\u***HexDigits* ... insert Unicode code value
- **\x***2HexDigits* ... insert individual bytes

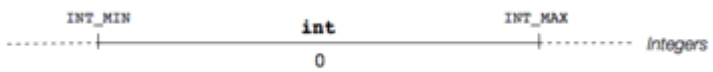
Examples:

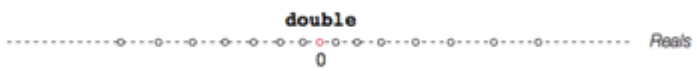
```
u"abc\u21ABdef"
"abc\xE2\x86\xABdef"
```

The red sequences produce 3 bytes and 1 Unicode symbol.

Numeric Data

Numerical data comes in two major forms: ***integers***, a subset (range) of the mathematical integers and ***floating point***, a subset of mathematical real numbers.





Integer Constants

There are three ways to write integer constants in C.

- ***Signed decimal*** (0..9) e.g. 42
- ***Unsigned hexadecimal*** (0..F) e.g. 0x2A
- ***Signed octal*** (0..7) e.g. 052

Some variations are

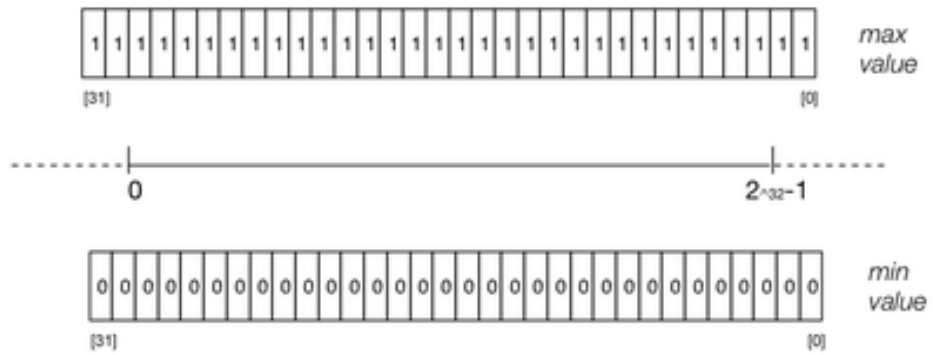
- Unsigned int value (typically 32 bits) e.g. 123U
- Long int value (typically 64 bits) e.g. 123L
- Short in value (typically 16 bits) e.g. 123S

Invalid constants lie outside the range for their type. e.g.

```
4294967296,   -1U,   666666S,   078
```

Unsigned Integers

The **unsigned int** data type is commonly 32 bits, storing values in the range $0..2^{32}-1$



The value is interpreted as a binary number.

`01001101` = $2^6 + 2^3 + 2^2 + 2^0$ = 64 + 8 + 4 + 1 = 77

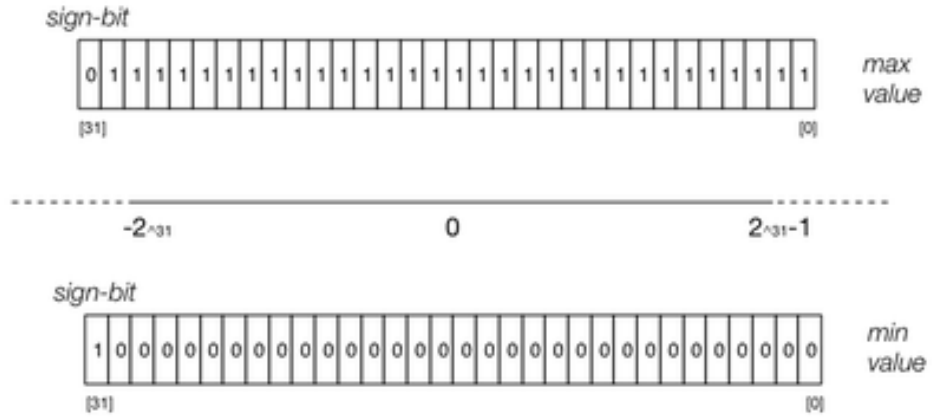
Addition is bitwise with carry

```
  00000001     00000001     01001101     11111111
+ 00000010   + 00000011   + 00001011   + 00000001
  --------     --------     --------     --------
  00000011     00000100     01011000     00000000
```

Most machines will also flag the *overflow* in the fourth example.

Signed Integers

The **int** data type is commonly 32 bits, storing values in the range $-2^{31}..2^{31}-1$



There are several possible representations for negative values:

- ***Signed magnitude*** - the first bit is the sign, the rest are magnitude
  e.g. 5 is `00000101` -> -5 is `10000101`
  A problem with signed magnitude is that there are ***two zeros***; `00000000` and `10000000`, one positive and one negative. Another problem is $x + -x \neq 0$ (mostly) with simple addition.
- ***Ones complement*** - for -N by ***inverting*** all bits in N
  e.g. 5 is 00000101 -> -5 is 11111010
  A problem with ones complement is that there are ***two zeros***; `00000000` and `11111111`, one positive and one negative. At least $x + -x$ is equal to one of the zeroes with simple addition
- ***Twos complement*** - form -N by ***inverting*** N and adding 1
  e.g. 5 is 00000101 -> -5 is 11111011
  Twos complement only has ***one*** representation for ***zero*** (`00000000`) . Also
  $-(-x) = x$ and $x + -x = 0$ in all cases with simple addition. It always produces and *overflow* bit, but this can be ignored

In all representations, *+ve numbers have 0 as the leftmost bit.*

Pointers

Pointers represent memory addresses/locations. The number of bits depends on the memory size, but it is typically 32 bits. Data pointers reference addresses in ***data***/***heap***/***stack*** regions. Function

pointers reference addresses in ***code*** region.
There are many kinds of pointers, one for each data type, but
`sizeof(int *) = sizeof(char *) = sizeof(double *) = sizeof(struct X *)`

Pointer *values* must be appropriate for data types. e.g.

- `(char *)` ... can reference any byte address
- `(int *)` ... must have *addr*`%4 == 0`
- `(double *)` ... must have *addr*`%8 == 0`

Pointers can *move* from object to object by pointer arithmetic. For any pointer `T *p;` `p++` means `p = p + 1` and it increases p by `sizeof(T)`.

Examples (assuming 16-bit pointers):

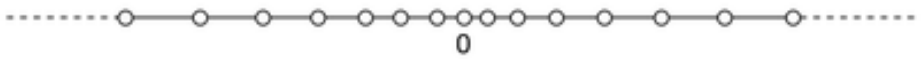
```
char   *p = 0x6060;  p++;  assert(p == 0x6061)
int    *q = 0x6060;  q++;  assert(q == 0x6064)
double *r = 0x6060;  r++;  assert(r == 0x6068)
```

This is a common (and efficient) paradigm for scanning a string.

```
char *s = "a string";
char *c;
// print a string, char-by-char
for (c = s; *c != '\0'; c++) {
   printf("%c", *c);
}
```

Floating Point Numbers

Floating point numbers model a (tiny) subset of $\mathbb{R}$. Many real values don't have exact representations (such as 1/3), so results of calculations may contain small inaccuracies.
Precision categorises how close values are to their exact numbers. Numbers close to zero have higher precision (more accurate) while numbers further from zero have lower precision (less accurate).



C has two floating point types:

- ***`float`***: typically 32-bit quantity (lower precision, narrower range)
- ***`double`***: typically 64-bit quantity (higher precision, wider range)

Some literal floating point values: `3.14159, 1.0/3, 1.0e-9`
To display them via `printf`

```
printf("%W.Pf", (float)2.17828)
printf("%W.Plf", (double)2.17828)
```

W gives total width (blank padded), *P* gives #digits after dec point

```
printf("%10.4lf", (double)2.718281828459);
// displays _ _ _ _2.7183
printf("%20.20lf", (double)4.0/7);
// displays 0.57142857142857139685
```

IEEE 754 standard ...

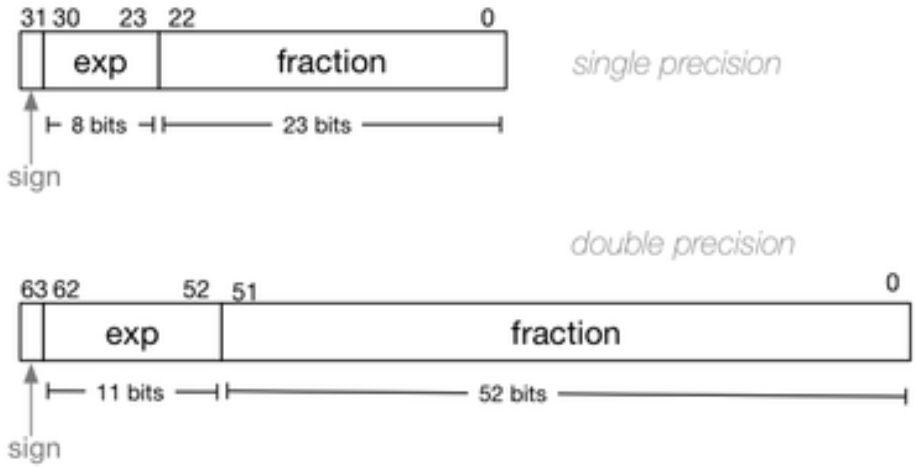
- scientific notation with *fraction F* and *exponent E*
- numbers have form $F \times 2^E$, where both *F* and *E* can be -ve
- INFINITY = representation for ∞ and -∞ (e.g. 1.0/0)
- NAN = representation for invalid value NaN (e.g. sqrt(-1.0))
- 32-bit single-precision, 64-bit double precision

The fraction part is *normalised* (i.e. 1.2345×10^2 rather than 123.45)
Example of normalising in binary:

- 1010.1011 is normalized as 1.0101011×2^{011}

- 1010.1011 = 10 + 11/16 = 10.6875
- 1.0101011×2^{011} = (1 + 43/128) * 2^3 = 1.3359375 * 8 = 10.6875

The internal structure of floating point values looks like this:



It is more complex than `int` because its format *1.dddd e dd*
Details of the internal structure:

- The fraction part is always 1.*bbbbbbbb*; don't store 1
- The exponent is offset relative to a baseline $-2^{b-1}-1$, b = #exponent bits

The range of values for 32-bit single precision float is:

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

`00...00` = 24 zero bits, `11...11` = 24 one bits
The ranges of values for 64-bit double-precision float is:

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

`00...00` = 52 zero bits, `11...11` = 52 one bits

Example (single-precision):

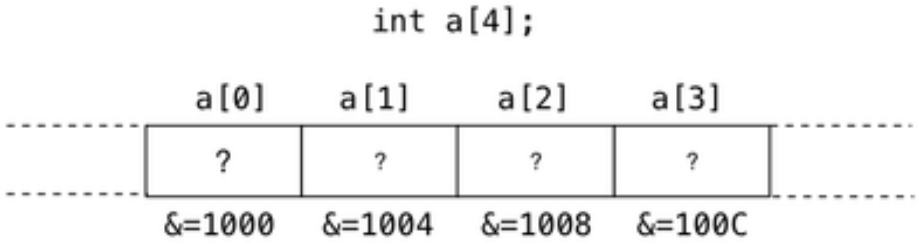
```
150.75 = 10010110.11
     // normalise fraction, compute exponent
        = 1.001011011 × 2^7
     // determine sign bit
     // map fraction to 24 bits
     // map exponent relative to baseline
        = 01000001100010110110000000000000
```

where red is sign bit, green is exponent, blue is fraction
Note:

- the baseline (aka bias) is 127, the exponent is 2^7
- so, in the exponent, we store 127+7 = 134 = 10000110

Arrays

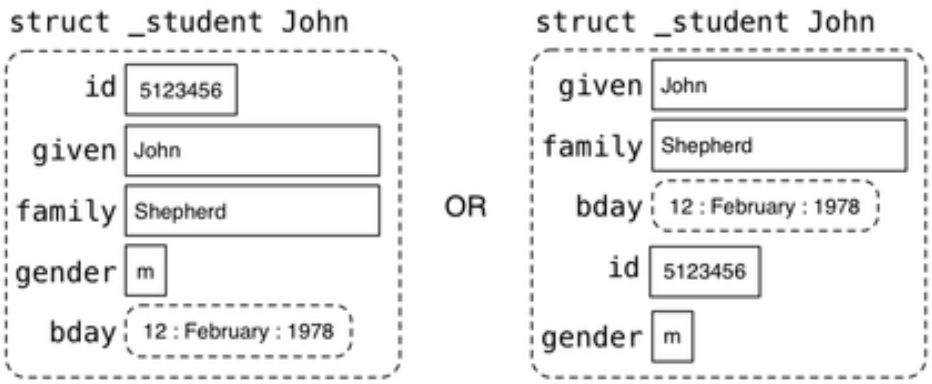
Arrays are defined to have N elements, each of type T. The elements are laid out in adjacent memory.



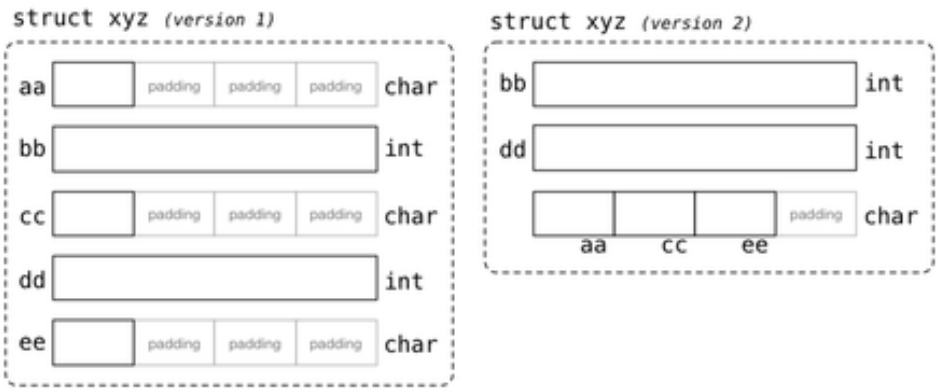
Assuming an array declaration like *Type v[N];*
The individual elements are accessed via indices `0..N-1`. The total amount of space allocated to the array is *N* × `sizeof(`*Type*`)`.
The name of array gives the address of the first element (e.g. `v = &v[0]`) and it can be treated as a pointer to element type *Type*.
Array indexing can be treated a `v[i]≅ *(v+i)`. If you have a pointer to the first element, you can use it like an array. Strings are just arrays of char with a `'\0'` terminator.
Constant strings have `'\0'` automatically added. String buffers must allow for elements to hold `'\0'`.

Structs

Structs are defined to have a number of components. Each component has a name and type. The internal layout of `struct` components are determined by the compiler.



Each name maps to a byte offset within the struct. e.g. in first example id = offset 0, given = offset 4, family = offset 54, etc.

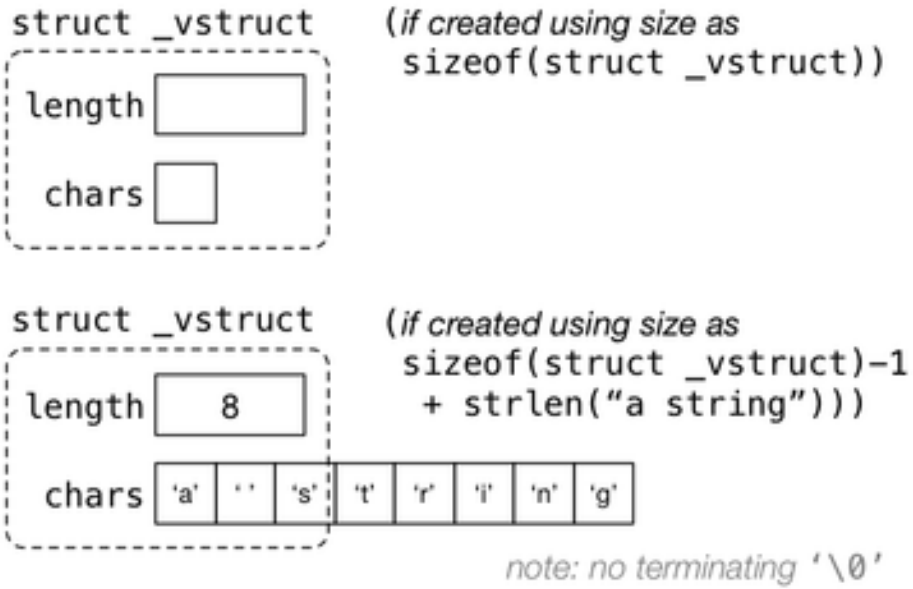


To ensure alignment, internal *padding* may be needed. Padding wastes space, so fields are often re-ordered to minimise waste.

Variable-length Structs

`Structs` can contain pointers to dynamic objects, but we can also *embed* one dynamic object in a malloc'd `struct`; define the dynamic object as the last component, then `malloc()` more space than the struct required to make the final component as large as required.

The amount of memory allocated to struct is determined dynamically:



Bit-wise Structs

C allows programmers to specify `structs` bit-wise to give them a fine control over the layout of fields in structs.

Bit-field structs specify unnamed ***components*** using standard types and named individual ***bit fields*** in each component.

Example:

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

Has one component and three bit fields within that component.

There are two ways of declaring bit fields:

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

OR

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

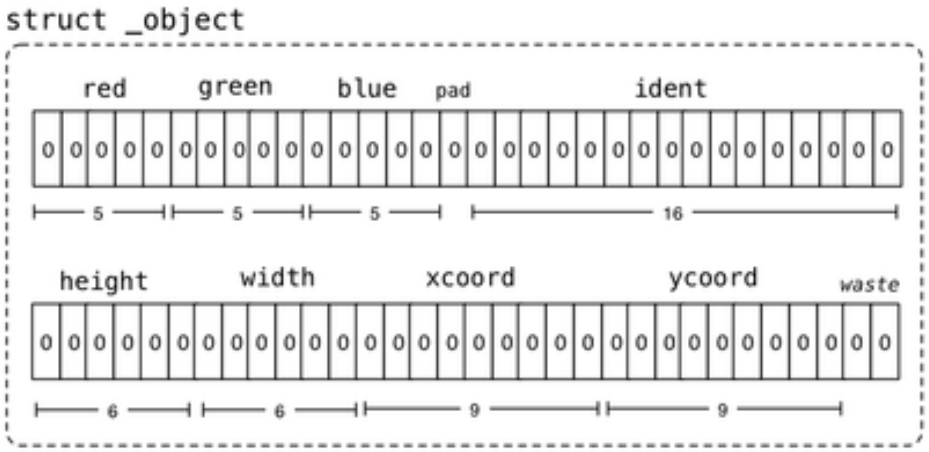
In both cases, `sizeof(struct _bit_fields)` is 4 bytes. The first way makes it cleared that a single `unsigned int` is used.

Another examples (graphical objects):

```
struct _object { // compromised of two 32-bit words
   unsigned int  red    : 5,  // 5 bits for red
                 green  : 5,  // 5 bits for green
                 blue   : 5,  // 5 bits for blue
                 pad    : 1,  // 1 bits to pad to short
                 ident  : 16; // 16 bits for object ID
   unsigned int  height : 6,  // 6 bits for object height
```

```
                width  : 6,  // 6 bits for object width
                xcoord : 9,  // 9 bits for object x-coordinate
                ycoord : 9;  // 9 bits for object y-coordinate
};
struct _object oval;
...
oval.red = 4; oval.blue = 31; oval.green = 15;
oval.height = 5; oval.width = 15;
```

The graphics object would be stored in memory as:



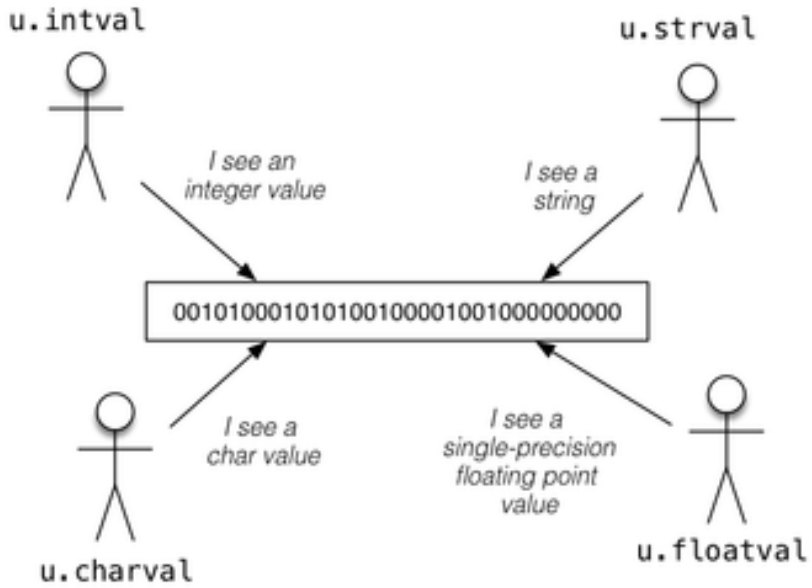
Bit-fields provide an alternative to bit operators and masks.

```
typedef unsigned int uint;
typedef uint privs;
#define OWNER_READ  (1 << 8)
#define OWNER_WRITE (1 << 7)
#define OWNER_EXEC  (1 << 6)
...
#define OTHER_WRITE (1 << 1)
#define OTHER_EXEC  (1 << 0)
unsigned int myPrivs;
myPrivs |= OWNER_EXEC;
myPrivs &= ~OTHER_WRITE;
open = myPrivs & OTHER_READ;
```

```
struct _privs {
unsigned int
    owner_read  : 1,
    owner_write : 1,
    owner_exec  : 1,
    ...
    other_write : 1,
    other_exec  : 1;
} myPrivs;
myPrivs.owner_exec = 1;
myPrivs.other_write = 0;
open = myPrivs.other_read;
```

Unions

Unions allow programmers to specify multiple interpretations for a single piece of memory.



Example of defining a union type (cf. struct):

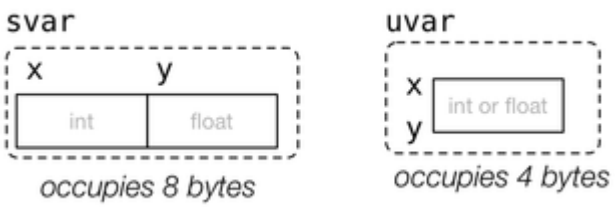
```
union _alltypes {
    int   intval;
    char  strval[4];
    char  charval;
    float floatval;
};
union _alltypes myUnion;
```

myUnion is a single 4-byte memory object
Programmers can specify how to interpret bits using field names.
In the example above, all components are coincidentally the same size (4 bytes)

Difference between a struct and a union

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

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



The general syntax for defining union types and variables is:

```
union Tag {
   Type1  Member1;
   Type2  Member2;
   Type3  Member3;
   ...
} uvar;
```

Type can be any C type; *Member* names must be distinct.
sizeof(*Union*) is the size of the largest member
&uvar.*Member*$_1$ == &uvar.*Member*$_2$ == &uvar.*Member*$_3$ ...

Common use of union types: *generic* variables

```
#define IS_INT   1
```

```
#define IS_FLOAT 2
#define IS_STR   3
struct _generic {
   int   vartype;
   union { int ival; char *sval; float fval; };
};
struct _generic myVar;
// treat myVar as an integer
myVar.vartype = IS_INT;
myVar.ival    = 42;
printf("%d\n", myVar.ival);
// now treat myVal as a float
myVar.vartype = IS_FLOAT;
myVar.fval    = 3.14159;
printf("%0.5f\n", myVar.fval);
```

Enumerated Types

Enumerated types allow programmers to define a set of distinct named values.

```
typedef enum { RED, YELLOW, BLUE } PrimaryColours;
typedef enum { LOCAL, INTL } StudentType;
```

The names are assigned consecutive `int` values, starting from `0`
Above `PrimaryColors` type is equivalent to

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

Variables of type `enum...` are effectively `unsigned ints`.

Instruction Set Architecture

Monday, 6 August 2018 7:36 PM

CPU Architecture

A typical modern CPU has:

- A set of data registers
- A set of control registers (including PC)
- An arithmetic-logic unit (ALU)
- Access to random access memory (RAM) - it can push stuff to memory or fetch things from memory
- A set of simple instructions to:
  - Transfer data between memory and registers
  - Push values thorough ALU to compute results
  - Make tests and transfer control of execution (*conditional execution*)

Different types of processors have different configurations of the above (e.g. different no. of registers, different sized registers, different instructions.

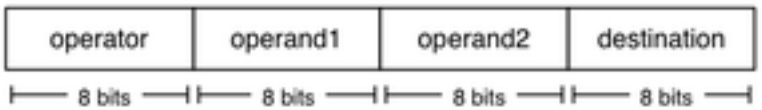
Instruction Sets

There are two broad families of instruction set architectures: 10mins

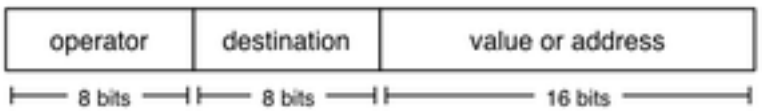
- RISC (reduced instruction set computer) - a small(ish) set of simple, general instructions, separate computation & data transfer instructions leading to simpler processor hardware (e.g. MIPS, RISC, Alpha, SPARC, PowerPC, ARM, …)
  Advantage: easy to build hardware, instructions execute faster
- CISC (complex instruction set computer) - a large(r) set of powerful instructions, where each instruction has multiple actions (comp + store). There is more circuitry to decode/process instructions (e.g. PDP, VAX, Z80, Motorola 68xxx, Intel x86, …)

Machine-level instructions typically have 1-2 32-bit words per instruction. They partition bits in each word into operator and operands. The no. of bits for each depends on the number of instructions and no. of register and other things.

Example of instruction word formats:



<- typical of an **add** operation

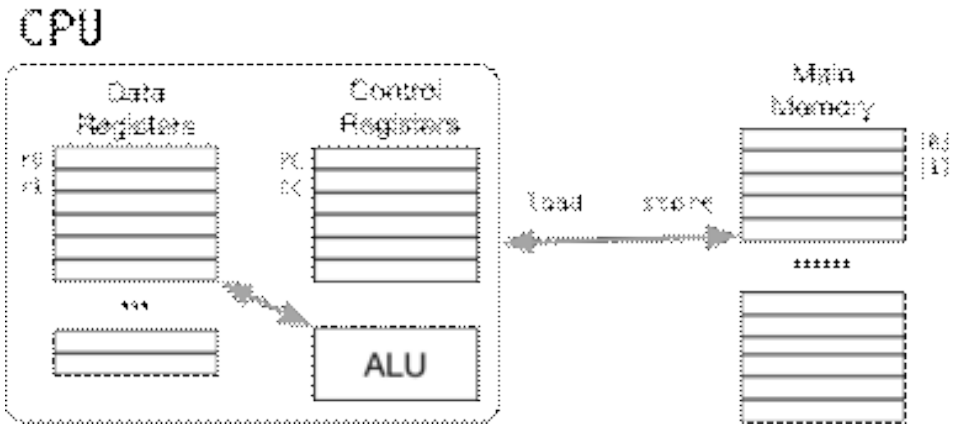


<- typical of a **store** operations

Operands and *destination* are typically registers.

Common kinds of instructions (*not from any real machine*) are:

- **load** *Register*, *MemoryAddress* - copy a value stored in memory at the given address into the named register.
- **loadc** *Register*, *ConstantValue* - copy the value into the named register
- **store** *Register*, *MemoryAddress* - copy the value stored in named register into memory at the given address
- **jump** *MemoryAddress* - transfer execution of program to instruction at address. Changes the value of PC (*Program Counter*)
- **jumpif** *Register*, *MemoryAddress* - transfer execution of program if e.g. register holds zero value
- **add** $Register_1$, $Register_2$, $Register_3$ (similarly for **sub**, **mul**, **div**) - $Register_3 = Register_1 + Register_2$
- **and** $Register_1$, $Register_2$, $Register_3$ (similarly for **or**, **xor**) - $Register_3 = Register_1 \& Register_2$
- **neg** $Register_1$, $Register_2$ - $Register_2 = \sim Register_1$
- **shiftl** $Register_1$, *Value*, $Register_2$ (similarly for **shiftr**) - $Register_2 = Register_1 << Value$
- **syscall** *Value* - invoke a system service; which service determined by *Value*



Fetch-Execute Cycle

All CPUs have program execution logic like:

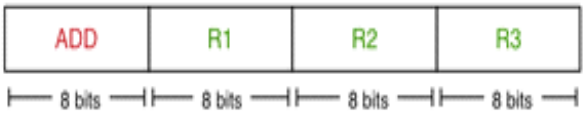
```
while (1)
{
   instruction = memory[PC]
   PC++ // move to the next instruction
   if (instruction == HALT)
      break
   else
      execute(instruction)
}
```

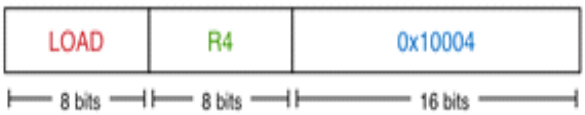
PC = Program Counter, a CPU register which keeps track of execution

Note: some instructions may modify PC further (e.g. JUMP)

Executing an instruction involves:

- Determining what the operator is
- Determining which registers, if any, are involved
- Determining which memory location, if any, is involved
- Carrying out the operation with the relevant operands
- Storing the result, if any, in the appropriate register





Assembly Language (MIPS)

Monday, 6 August 2018 8:37 PM

Assembly Language

Instructions are simply bit patterns within a 32-bit bit-string.
They could describe machine programs as a sequence of hex digits, e.g.

```
Address    Content
0x100000   0x3c041001
0x100004   0x34020004
0x100008   0x0000000c
0x10000C   0x03e00008
```

They often call "*assembly language*" as "*assembler*", which is a slight notational abuse, because "*assembler*" also refers to a program that translates assembly language to machine code.

Assembly language provides a symbolic way of giving machine code a way to:

- writes instructions using mnemonics rather than hex or binary codes,
- refer to registers using either numbers or names,
- associate names to memory addresses.

The style of expression is significantly different to e.g. C as you need to use fine-grained control of memory usage, are required to manipulate data in registers, and control structures programmed via explicit jumps.

MIPS Architecture

MIPS is a well-known and relatively simple architecture. It was very popular in a range of computing devices in the 1990's (e.g. Silicon Graphics, NEC, Nintendo64, PlayStation, supercomputers).
We will consider the MIPS32 version of the MIPS family using two variants of the open-source SPIM emulator:

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

MIPS vs SPIM

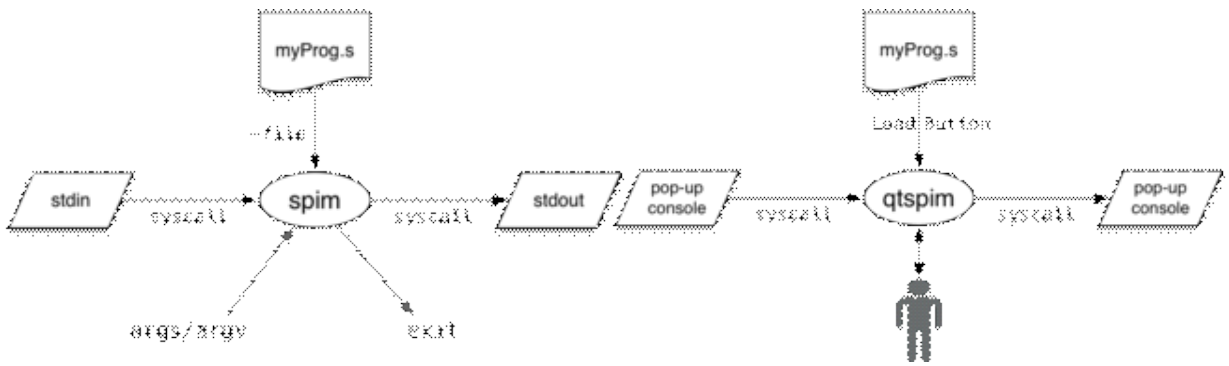
MIPS is a machine architecture, including instruction set.

SPIM is an *emulator* for the MIPS instruction set. It reads text files containing instructions and directives, then convert this into machine code and loads it into "memory". SPIM provides debugging capabilities (single-step, breakpoints, view registers/memory), and provides a mechanism to interact with the operating system (`syscall`).
It also provides extra instructions, mapped to MIPS core set and provides convenient/mnemonic was to do common operations (e.g. `move $s0, $v0` rather than `addu $s0, $0, $v0`)

Using SPIM

There are 3 ways to execute MIPS code with SPIM:

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



MIPS Machine Architecture

MIPS CPU has:

- 32 × 32-bit general purpose registers
- 16 × 64-bit double-precision registers
- 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

Registers and their usage:

Reg	Name	Notes
$0	zero	the value 0, not changeable

$1	$at	assembler temporary; used to implement pseudo-ops
$2	$v0	value from expression evaluation or function return
$3	$v1	value from expression evaluation or function return
$4	$a0	first argument to a function/subroutine, if needed
$5	$a1	second argument to a function/subroutine, if needed
$6	$a2	third argument to a function/subroutine, if needed
$7	$a3	fourth argument to a function/subroutine, if needed
$8..$15	$t0..$t7	temporary; must be saved by caller to subroutine; *subroutine can overwrite*
$16..$23	$s0..$s7	safe function variable; *must not be overwritten by called subroutine*
$24..$25	$t8..$t9	temporary; must be saved by caller to subroutine; subroutine can overwrite
$26..$27	$k0..$k1	for kernel use; may change unexpectedly
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	return address of most recent caller

"*subroutine*" = older low-level name for "*function*"
Note: all operators in MIPS are binary; they can only take two arguments

Floating point register usage:

Reg	Notes
$f0..$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

Notes:
- registers come in pairs of 2 × 32-bits
- only even registers are addressed for double-precision

MIPS Assembly Language

MIPS assembly language programs contain:
- comments - are introduced by **#**
- labels (named pieces of memory) - are appended with **:**
- directives - are a 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

.data	Indicates what you want to be stored in the data part of memory
.text	Indicates where you want to store in the code part of memory

Example MIPS assembler program:

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

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

    .text          # the code segment
    .globl main
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)
```

Color coding: label, directive, comment

MIPS programs assume the following memory layout

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***

General structure of MIPS programs

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

    .data     # variable declarations follow this line
              # …

    .text     # instructions follow this line
    .globl main
main:          # indicates start of code
               # (i.e. first user instruction to execute)
               # ...

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

Another MIPS assembler program:

```
    .data
a:  .word 42            # int a = 42;
b:  .space 4            # int b;

    .text
    .globl main
main:
    lw   $t0, a         # reg[t0] = a
    li   $t1, 8         # reg[t1] = 8
    add  $t0, $t0, $t1  # reg[t0] = reg[t0]+reg[t1]
    li   $t2, 666       # reg[t2] = 666
    mult $t0, $t2       # (Lo,Hi) = reg[t0]*reg[t2]
    mflo $t0            # reg[t0] = Lo
    sw   $t0, b         # b = reg[t0]
    ...
```

k_text	0x80000000	kernel code; read-only; only accessible kernel mode
k_data	0x90000000	kernel data; read/write; only accessible kernel mode

Note: no heap, but the data segment can be expanded

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 are 32-bits long an specify an operation (e.g. *load*, *store*, *add*, *branch*, ...), and one or more operands (*registers, memory addresses, constants).*
Some possible instruction formats:

Addressing Modes

Memory addresses can be given by:

- Symbolic name (label) (effectively a constant)
- Indirectly via a register (effectively pointer dereferencing

```
prog:
a:    lw    $t0, var     # address via name
b:    lw    $t0, ($s0)   # indirect addressing
c:    lw    $t0, 4($s0)  # indexed addressing
d:    lw    $t0, vec($s1)# 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`

Addressing modes in MIPS

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)

Examples of load/store and addressing:

```
    .data
vec: .space   16      # int vec[4]; 16 bytes of storage

    .text
__start:
    la  $t0, vec      # reg[t0] = &vec
    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] = 42
```

Operand Sizes

MIPS instructions can manipulate different-sized operands; single bytes, two bytes (***halfword***), four bytes (***word***)
Many instructions also have variants for signed and unsigned.
This leads to many operation codes for a (conceptually) single operation. e.g.

LB	load one byte from specified address
LBU	load unsigned byte from specified address
LH	load two bytes from specified address
LHU	load unsigned 2-bytes from specified address
LW	load four bytes (one word) from specified address
LA	load the specified address

All of the above specify a destination register.

MIPS Instruction Set

The MIPS processor implements a base of set instructions e.g. (lw, sw, add, sub, and, or, sll, slt, beq, jr, jal,...). These instruction are augmented by a set of pseudo-instructions e.g. (move, rem, la, li, blt, ...). Each pseudo-instruction maps to one or more base instructions.

Pseudo-instruction	Base instruction(s)
`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: the use of the $at register is for intermediate results

In describing instructions:

Syntax	Semantics

Examples of testing and branching instructions:

```
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)
```

After each branch instruction, execution continues at new PC location

Special jump instructions for invoking functions

```
jal  label        # make a subroutine call
                  # save PC in $ra, set PC to &label
```

Note: the use of the $at register is for intermediate results

In describing instructions:

Syntax	Semantics
$Reg	as source, the content of the register, reg[Reg]
$Reg	as destination, value is stored in register, reg[Reg] = value
Label	references the associated address (in C terms, &Label)
Addr	any expression that yields an address (e.g. Label($Reg))
Addr	as source, the content of memory cell memory[Addr]
Addr	as destination, value is stored in memory[Addr] = value

Effectively …

- treat registers as unsigned int reg[32]
- treat memory as unsigned char mem[2^{32}]

Examples of data movement instructions:

```
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][32:16] = const
```

Examples of bit manipulation instructions:

```
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]
```

Examples of arithmetic instructions:

```
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
```

MIPS Programming

Writing directly in MIPS assembler is difficult (almost impossible).
A strategy for producing likely correct MIPS code is to:

- develop the solution in C
- map this solution to "simplified" C
- translate each simplified C statement to MIPS instructions

Simplified C

- ***does not*** have `while`, `switch`, complex expressions
- ***does*** have simple `if`, `goto`, one-operator expressions
- ***does not*** have function calls and auto local variables
- ***does*** have jump-and-remember-where-you-came-from

An example of translating C to MIPS:

```
C             Simplified C   MIPS Assembler
------------  ------------   --------------
int x = 5;    int x = 5;     x:  .word 5
int y = 3;    int y = 3;     y:  .word 3
int z;        int z;         z:  .space 4
              int t;         ...
                             lw  $t0, x
                             lw  $t1, y
z = 5*(x+y);   t = x+y;      add $t0, $t0, $t1
                             li  $t1, 5
               t = 5*t;      mul $t0, $t0, $t1
               z = t;        sw  $t0, z
```

Simplified C makes extensive use of:

Special jump instructions for invoking functions

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

```
main:
    # set params
    # $a0, $a1, …
    jal func
    # main continues
    …

func:
    …
    # set return $v0
    jr $ra
```

SPIM interacts with stdin/stdout via `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")

Directives (instructions to assemble, not MIPS instructions):

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

    .globl        # make symbol available globally

a:  .space 18     # unchar a[18]; or uint a[4];
    .align 2      # align next object on 2^2-byte address

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 t[3] = {'a', 'b', 'c'};
```

Beware: registers are shared by all parts of the code.

One function can overwrite a value set by another function.
For example:

```
int x;   // first global variable
int y;   // second global variable
int main(void)          int f(int n)
{                       {
   x = 5;                  y = 1;
   y = f(x);               for (x = 1; x <= n; x++)
   printf("...",x,y);         y = y * x;
   return 0;               return y;
}                       }
```

After the function, `x == 6` and `y == 120`. It is a sheer coincidence that y has the correct value.

We need to be careful managing registers. Follow the conventions implied by register names and preserve values that need to be saved across function calls.

Within a function, you manage register usage as you like, by typically making use of `$t?` registers. When making a function call, you transfer control to a separate piece of code, which may change the value of any non-preserved register.
`$s?` registers must be preserved by a function, while **`$a?`, `$v?`, `$t?`** registers may be modified by the function.

- labels - the symbolic name for C statements
- goto - a way to transfer control to a labelled statement

Example:

```
Standard C              Simplified C
------------------      ------------------
i = 0; n = 0;           i = 0; n = 0;
while (i < 5) {         loop:
   n = n + i;             if (i >= 5) goto end;
   i++;                   n = n + i;
}                         i++;
                          goto loop;
                        end:
```

Rendering C in MIPS

C provides expression evaluation and assignment, e.g.

```
x = (1 + y*y) / 2;    z = 1.0 / 2; ...
```

MIPS provides register-register operations, e.g.

```
move Rd,Rs,    li Rd,Const,    add, div, and, ...
```

C provides a range of control structures

```
sequence (;), if, while, for, break, continue, ...
```

MIPS provides testing/branching instructions

```
seq, slti, sltu, ..., beq, bgtz, bgezal, ..., j, jr, jal, ...
```

We need to render C's structures in terms of testing/branching.

Sequence is easy S_1 ; S_2 → mips(S_1) mips(S_2)

Here is a simple example of assignment and sequence:

```
int x;          x: .space 4
int y;          y: .space 4
x = 2;             li   $t0, 2
                   sw   $t0, x
y = x;             lw   $t0, x
                   sw   $t0, y
y = x+3;           lw   $t0, x
                   addi $t0, $t0, 3
                   sw   $t0, y
```

Arithmetic Expressions

Expression evaluation involves describing the process as a sequence of binary operations, and managing data flow between the operations.

Example:

```
# x = (1 + y*y) / 2
# assume x and y exist as labels in .data
  lw   $t0, y          # t0 = y
  mul  $t0, $t0, $t0   # t0 = t0*t0
  addi $t0, $t0, 1     # t0 = t0+1
  li   $t1, 2          # t1 = 2
  div  $t0, $t1        # Lo = t0/t1 (int div)
  mflo $t0             # t0 = Lo
  sw   $t0, x          # x = t0
```

It is useful to minimise the number of registers involved in the evaluation

Conditional Statements

```
Standard C              Simplified C            MIPS Assembler
----------------------- ----------------------- -----------------------
                        if_stat:                if_stat:
if (Cond)                 t0 = (Cond)             t0 = evaluate (Cond)
   { Statements1 }        if (t0 == 0)            beqz $t0, else_part
else                        goto else_part;       execute Statements1
 { Statements2 }          Statements1             j    end_if
                          goto end_if;           else_part:
                        else_part:                execute Statements2
                          Statements2           end_if:
                        end_if:
```

Example of if-then-else:

```
int x;          x is $t0
int y;          y is $t1
char z;         z is $a0
x = getInt();   li   $v0, 5
                syscall
                move $t0, $v0
y = getInt();   li   $v0, 5
                syscall
                move $t1, $v0
if (x == y)     bne  $t0, $t1, setN
   z = 'Y';   setY:
                li   $a0, 'Y'
                j    print
else          setN:
   z = 'N';     li   $a0, 'N'
                j    print
              print:
putChar(z);     li   $v0, 11
                syscall
```

We can also make switch statements by converting it into an if statement. An alternative way to implement switch would be:

```
                                jump_tab:
                                   .word c1, c2, c2, c2, c3
                                switch:
                                    t0 = evaluate Expr
                                    if (t0 < 1 || t0 > 5)
switch (Expr) {                        jump to default
case 1:                             dest = jump_tab[(t0-1)*4]
   Statements1 ; break;             jump to dest
case 2:                         c1: execute Statements1
case 3:                             jump to end_switch
case 4:                         c2: execute Statements2
   Statements2 ; break;             jump to end_switch
case 5:                         c3: execute Statements3
   Statements3 ; break;             jump to end_switch
default:                        default:
   Statements4 ; break;             execute Statements4
}                               end_switch:
```

This works best for small, dense ranges of case values (e.g. 1 to 10)

```
                print:
putChar(z);       li   $v0, 11
                  syscall
```

```
                                  end_switch:
```

This works best for small, dense ranges of case values (e.g. 1 to 10)

Boolean Expressions

Boolean expressions in C are short circuit

```
(Cond1 && Cond2 && ... && Condn)
```

It evaluates by

- Evaluating $Cond_1$; if 0 then return 0 for whole expression
- Evaluating $Cond_2$; if 0 then return 0 for whole expression
- ...
- Evaluating $Cond_n$; if 0 then return 0 for whole expression
- Otherwise, return 1

Similarly for disjunctions

```
(Cond1 || Cond2 || ... || Condn)
```

It evaluates by

- Evaluating $Cond_1$; if !0 then return 1 for whole expression
- Evaluating $Cond_2$; if !0 then return 1 for whole expression
- ...
- Evaluating $Cond_n$; if !0 then return 1 for whole expression
- Otherwise, return 1

In C, any non-zero value is treated as true; MIPS tends to use 1 for true
C99 standard defines return value for booleans expressions as 0 or 1

Iteration Statements

Iteration - treat `for` loops as an alternative to `while` loops

```
                    top_while:
while (Cond) {         t0 = evaluate Cond
   Statements;         beqz $t0,end_while
}                      execute Statements
                       j    top_while
                    end_while:
```

Example of iteration over an array:

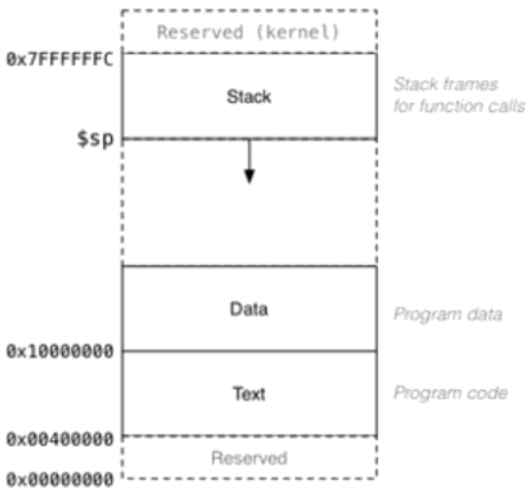
```
int sum, i;              sum: .word 4       # use reg for i
int a[5] = {1,3,5,7,9};  a:   .word 1,3,5,7,9
...                           ...
sum = 0;                      li   $t0, 0   # i = 0
                              li   $t1, 0   # sum = 0
                              li   $t2, 4   # max index
for (i = 0; i < N; i++)  for: bgt  $t0, $t2, end_for
                              move $t3, $t0
                              mul  $t3, $t3, 4
   sum += a[i];               add  $t1, $t1, a($t3)
printf("%d",sum);             addi $t0, $t0, 1   # i++
                              j    for
                         end_for: sw   $t1, sum
                              move $a0, $t1
                              li   $v0, 1
                              syscall       # printf
```

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

Data associated with function calls is placed on the MIPS stack.



Each function allocates a small section of the stack (a *frame*)

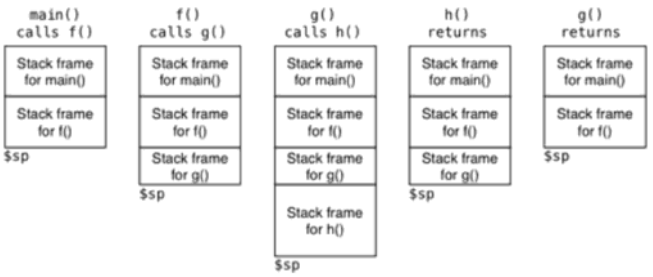
- used for: saved registers, local variables, parameters to callees
- created in the function *prologue* (pushed)
- removed in the function *epilogue* (popped)

Why we use a stack:

- function `f()` calls `g()` which calls `h()`
- `h()` runs, then finishes and returns to `g()`
- `g()` continues, then finishes and returns to `f()`

i.e. last-called, exits-first (last-in, first-out) behaviour

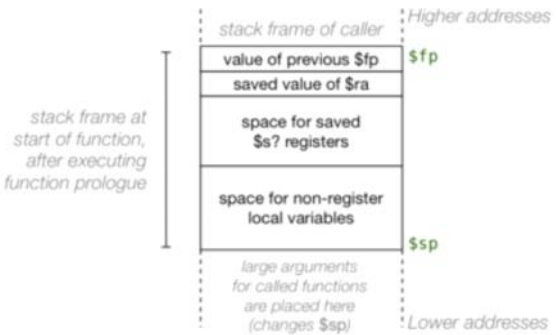
How stack changes as functions are called and returned:



Register usage conventions when `f()` calls `g()`:

- caller saved registers (saved by `f()`)
  - `f()` tells `g()` "If there is anything I want to preserve in these registers, I have already saved it before calling you"
  - `g()` tells `f()` "Don't assume that these registers will be unchanged when I return to you"
  - e.g. `$t0 .. $t9,  $a0 .. $a3,  $ra`
- callee saved registers (saved by `g()`)
  - `f()` tells `g()` "I assume the values of these registers will be unchanged when you return"
  - `g()` tells `f()` "If I need to use these registers, I will save them first and restore them before returning"
  - e.g. `$s0 .. $s7,  $sp,  $fp`

Contents of a typical stack frame:



MIPS Branch Delay Slots

The real MIPS architecture is "pipelined" to improve efficiency. One instruction can start before the previous one finished. For branching instructions (e.g. `jal`), instructions following branch is executed before the branch completes. To prevent problems, use nop immediately after branch.

A problem scenario, and its solution:

Implementation of `print(compute(42))`

```
li   $a0, 42          li   $a0, 42
```

```
jal  compute          jal  compute
move $a0, $v0         nop
jal  print            move $a0,$v0
                      jal  print
```

Since SPIM is not pipelined, the nop is not required.

Why do we need $fp and $sp?

During execution of a function `$sp` can change (e.g. pushing parameters, adding local variables). You may need to reference local variables on the stack. It is useful if they can be defined by an offset relative to fixed point. `$fp` provides a fixed point during function code execution.

```
int f(int x) {
   int y = 0;                  // y created in prologue
   for (int i = 0; i < x; i++) // i created in for-loop
      y += i                   //     which changes $sp
   return y;
}
```

Function Calling Protocol

Before one function calls another, it needs to

- place 64-bit double arguments in `$f12` and `$f14`
- place 32-bit arguments in the `$a0..$a3`
- if there are more than 4 arguments, or arguments larger than 32-bits ...
  - push value of all such arguments onto stack
- save any non-`$s?` registers that need to be preserved
  - push value of all such registers onto stack
- `jal` address of function (usually given by a label)

Pushing value of e.g. $t0 onto stack means:

```
addi $sp, $sp, -4
sw   $t0, ($sp)
```

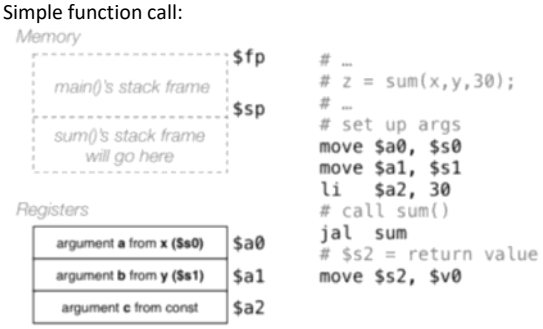
Note: when `jal`

- on MIPS hardware `$ra = PC + 8`
- on SPIM emulator `$ra = PC + 4`

Example: simple function call

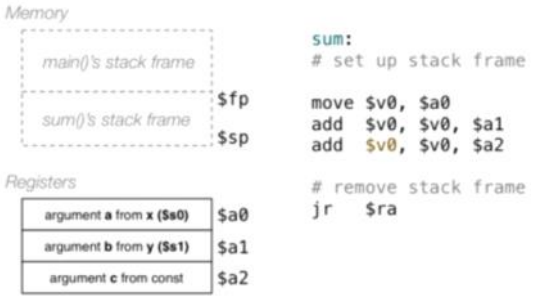
```
int main()
{
   // x is $s0, y is $s1, z is $s2
   int x = 5; int y = 7; int z;
   ...
   z = sum(x,y,30);
   ...
}
int sum(int a, int b, int c)
{
   return a+b+c;
}
```

Simple function call:




```
# …
# z = sum(x,y,30);
# …
# set up args
move $a0, $s0
move $a1, $s1
li   $a2, 30
# call sum()
jal  sum
# $s2 = return value
move $s2, $v0
```

Execution of `sum()` function:




```
sum:
# set up stack frame

move $v0, $a0
add  $v0, $v0, $a1
add  $v0, $v0, $a2

# remove stack frame
jr   $ra
```

Structure of Functions

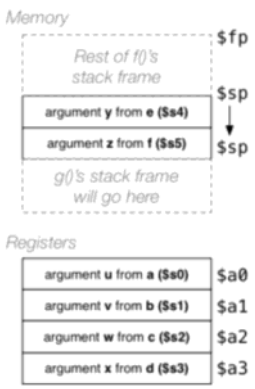
Functions in MIPS have the following general structure:

```
# start of function
FuncName:
   # function prologue
```

Example: function f() calls function g(a, b, c, d, e, f)

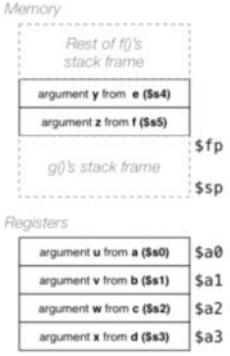
```
int f(...)
{
   // variables happen to be stored
   // in registers $s0, $s1, ..., $s5
   int a,b,c,d,e,f;
   ...
   a = g(a,b,c,d,e,f);
   ...
}
int g(int u,v,w,x,y,z)
{
   return u+v+w*w*x*y*z;
}
```

Function call in MIPS:




```
# … rest of f()'s code
# a = g(a,b,c,d,e,f)
# …
# set up args
move $a0, $s0
move $a1, $s1
move $a2, $s2
move $a3, $s3
sw   $s4, -4($sp)
sw   $s5, -8($sp)
addi $sp, $sp, -8
# call g()
jal  g
# $s0 = return value
move $s0, $v0
```

Execution of g() function:




```
g:
# set up stack frame

move $v0, $a2       # w
mul  $v0, $v0, $a2  # * w
mul  $v0, $v0, $a3  # * x
lw   $t0, 8($fp)
mul  $v0, $v0, $t0  # * y
lw   $t0, 4($fp)
mul  $v0, $v0, $t0  # * z
add  $v0, $v0, $a0  # + u
add  $v0, $v0, $a1  # + v

# remove stack frame
jr   $ra
```

Computing u+v+w*w*x*y*z

```
#   set up stack frame ($fp, $sp)
#   save relevant registers (incl. $ra)
...
# function body
#   perform computation using $a0, etc.
#   leaving result in $v0
...
# function epilogue
#   restore saved registers (esp. $ra)
#   clean up stack frame ($fp, $sp)
jr  $ra
```

Aim of prologue: create environment for function to execute in.

Function Prologue

Before a function starts working, it needs to ...

- create a stack frame for itself (change $fp and $sp)
- save the return address ($ra) in the stack frame
- save any $s? registers that it plans to change

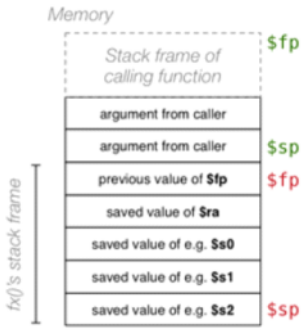
We can determine the initial size of the stack frame via

- 4 bytes for saved $fp + 4 bytes for saved $ra
- + 4 bytes for each saved $s?

Changing $fp and $sp ...

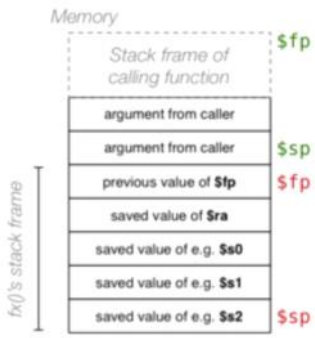
- new $fp = old $sp - 4
- new $sp = old $sp - size of frame (in bytes)

Example of function fx(), which uses $s0, $s1, $s2



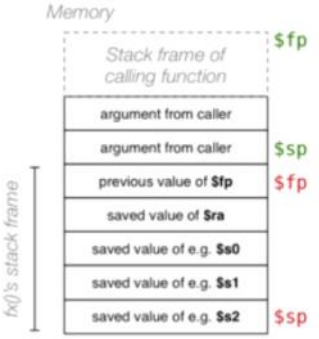

```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
sw   $ra, -8($sp)
sw   $s0, -12($sp)
sw   $s1, -16($sp)
sw   $s2, -20($sp)
la   $fp, -4($sp)
add  $sp, $sp, -20
# rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```

Alternatively... (a more explicit push)




```
# start of fx() function
fx:
# fx()'s prologue
addi $sp, $sp, -4
sw   $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw   $ra, ($sp)
addi $sp, $sp, -4
sw   $s0, ($sp)
addi $sp, $sp, -4
sw   $s1, ($sp)
addi $sp, $sp, -4
sw   $s2, ($sp)
# rest of fx()'s code

initial values: $fp,$sp   final values: $fp,$sp
```

Alternatively...(relative to new $fp)




```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
la   $fp, -4($sp)
sw   $ra, -4($fp)
sw   $s0, -8($fp)
sw   $s1, -12($fp)
sw   $s2, -16($fp)
add  $sp, $sp, -20
# rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```

Function Epilogue

Before a function returns, it needs to ...

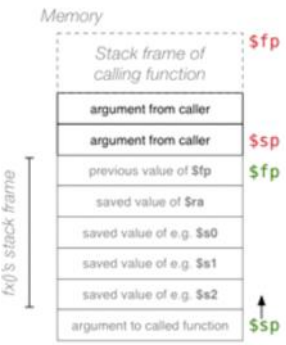
- place the return value in $v0 (and maybe $v1)
- pop any pushed arguments off the stack
- restore the values of any saved $s? registers
- restore the saved value of $ra (return address)
- remove its stack frame (change $fp and $sp)
- return to the calling function (jr $ra)

Locations of saved values computed relative to $fp

Changing $fp and $sp ...

- new $sp = old $fp + 4
- new $fp = memory[old $fp]

Example of function fx(), which uses $s0, $s1, $s2




```
# end of fx() function
lw   $v0, return value
# fx()'s epilogue
lw   $s2, -16($fp)
lw   $s1, -12($fp)
lw   $s0, -8($fp)
lw   $ra, -4($fp)
# remove stack frame
la   $sp, 4($fp)
lw   $fp, ($fp)
# return
jr   $ra

initial values: $fp,$sp
final values: $fp,$sp
```

Data Structures and MIPS

C data structures and their MIPS representations:

- char - as ***byte*** in memory, or low-order byte in register
- int - as ***word*** in memory, or whole register
- double - as ***two-words*** in memory, or $f? register
- arrays - sequence of memory bytes/words, accessed by index
- structs - chunk of memory, accessed by fields/offsets
- linked structures - struct containing address of another struct

A char, int or double

- could be implemented in register if used in small scope
- could be implemented on stack if local to function
- could be implemented in .data if need longer persistence

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
```

Dynamic allocation (i):

- variables local to a function

Prefer to put local vars in registers, but if cannot ...

- use space allocated on stack during function prologue
- referenced during function relative to $fp
- space reclaimed from stack in function epilogue

Example:

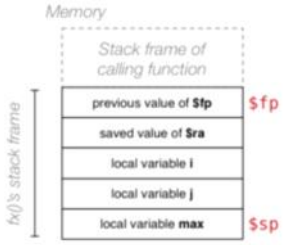
```
int fx(int a[])
{
```

```
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"
```

Example of local variables on the stack:




```
fx:
# fx()'s prologue
# save $fp, $ra; fix $fp, $sp
# allocate space for i, j, max
add  $sp, $sp, -12
…
li   $t0, 1          # i = 1;
sw   $t0, -8($fp)
li   $t0, 2          # j = 2;
sw   $t0, -12($fp)
…
# fx()'s epilogue
add  $sp, $sp, 12
lw   $ra, ($sp)
…
```

SPIM does not provide `malloc()`/`free()` functions, but it provides `syscall 9` to extend `.data`. Before syscall, set `$a0` to the number of bytes requested. After syscall, `$v0` holds the start address of allocated memory.
Example:

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

You **cannot** access allocated data by name; You need to retain address. There is no way to free allocated data, and there is no way to align data appropriately.

1-D Arrays in MIPS

1-d arrays in MIPS can be named/initialised as noted below:

```
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}
```

We can access elements via index or cursor (pointer). Either approach needs to account for size of elements.

Arrays are passed to functions via pointer to first element. You **must** also pass the array size, since it is not available elsewhere.

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

Scanning across an array of N elements using ***index***

```
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
#    printf("%d\n", vec[i]);
   li   $s0, 0              # i = 0
   li   $s1, 10             # no. of elements
   li   $s2, 4              # sizeof each element
loop:
   bge  $s0, $s1, end_loop  # if (i >= 10) break
   mul  $t0, $s0, $s2       # index->byte offset
   lw   $a0, vec($t0)       # a0 = vec[i]
   jal  print               # print a0
   addi $s0, $s0, 1         # i++
   j    loop
end_loop:
```

This assumes the existence of a `print()` function.

2-d Arrays in MIPS

2-d arrays can be represented in two ways:

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

- space reclaimed from stack in function epilogue

Example:

```
int fx(int a[])
{
   int i, j, max;
   i = 1; j = 2; max = i+j;
   ...
}
```

Dynamic allocation (ii):

- uninitialised block of memory allocated at run-time

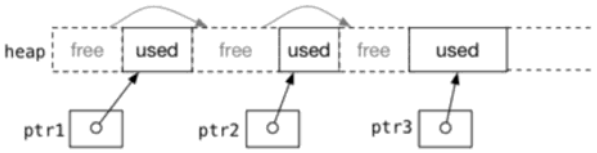
```
int  *ptr = malloc(sizeof(int));
char *str = malloc(20*sizeof(char));
int  *vec = malloc(20*sizeof(int));

*ptr = 5;
strcpy(str, "a string");
vec[0] = 1;
vec[1] = 6;
```

- initialised block of memory allocated at run-time

```
int *vec = calloc(20, sizeof(int));
// vec[i] == 0, for i in 0..19
```

Implementing C-like `malloc()` and `free()` in MIPS requires

- a complete implementation of C's heap management, i.e.
- a large region of memory to manage (`syscall 9`)
- ability to mark chunks of this region as "in use" (with size)
- ability to maintain list of free chunks
- ability to merge free chunks to prevent fragmentation



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:
```

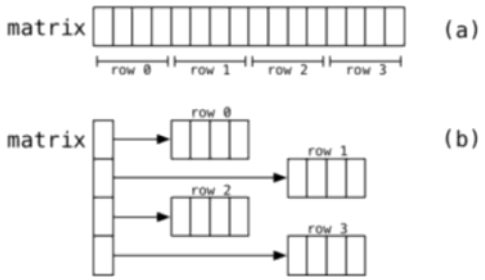
Assumes the existence of a `print()` function.

Arrays that are local to functions are allocated space on the stack

```
fun:                          int fun(int x)
                              {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)               // push a[] onto stack
   addi $sp, $sp, -40            int a[10];
   move $s0, $sp                 int *s0 = a;

   ... compute ...               // compute using s0
                                 // to access a[]
   addi $sp, $sp, 40             // pop a[] off stack
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra                   }
```

Computing sum of all elements for strategy (a) `int matrix[4][4]`

```
   li  $s0, 0            # sum = 0
   li  $s1, 4            # s1 = 4 (and size of int)
```



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

```
matrix: .space 64
row0:   .space 16
row1:   .space 16
row2:   .space 16
row3:   .space 16
matrix: .word row0, row1, row2, row3
```

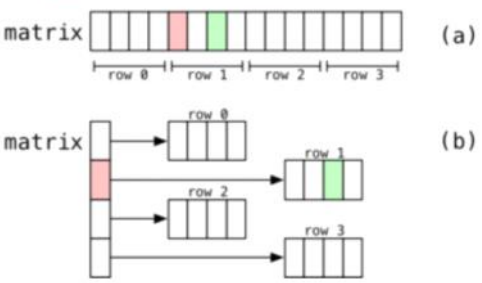
Now consider summing all elements

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

Accessing elements:

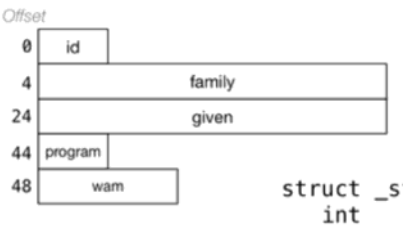
```
x = matrix[1][2];
```

Find start of row 1, then add offset 2 within row



Structs in MIPS

C `structs` hold a collection of values accessed by name




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

C struct definitions effectively define a new type.

```
// new type called struct _student
struct _student {...};
// new type called Student
typedef struct _student Student;
```

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;
```

Accessing structure components is by offset, not name

```
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;
```

Computing sum of all elements for strategy (a) `int matrix[4][4]`

```
   li  $s0, 0             # sum = 0
   li  $s1, 4             # s1 = 4 (and size of int)
   li  $s2, 0             # i = 0
   li  $s3, 16            # size of rows in bytes
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, $s3     # off = 4*4*i + 4*j
   mul  $t1, $s3, $s1     #  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:
```

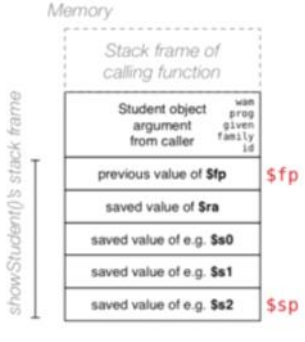
Computing sum of all elements for strategy (b) int matrix[4][4]

```
   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, $s1     # off = 4*i
   lw   $s4, matrix($t0)  # row = &matrix[i][0]
loop2:
   beq  $s3, $s1, end2    # if (j >= 4) break
   mul  $t0, $s3, $s1     # 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:
```

C can pass whole structure to functions, e.g.

```
# Student stu; ...
# // set values in stu struct
# showStudent(stu);
.data
stu: .space 56
   .text
   ...
   la   $t0, stu
   addi $sp, $sp, -56    # push Student object onto stack
   lw   $t1, 0($t0)      # allocate space and copy all
   sw   $t1, 0($sp)      # values in Student object
   lw   $t1, 4($t0)      # onto stack
   sw   $t1, 4($sp)
   ...
   lw   $t1, 52($t0)     # and once whole object copied
   sw   $t1, 52($sp)
   jal  showStudent      # invoke showStudent()
   ...
```

Accessing `struct` within function ...




```
# start of function
showStudent:
# prologue
   …
# function body
   …
   # t0 = s.id
   lw   $t0, 4($fp)
   # t1 = s.prog
   lw   $t1, 48($fp)
   # print s.family
   la   $a0, 8($fp)
   li   $v0, 4
   syscall
   …
```

Can also pass a pointer to a struct

```
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 are allocated space on the stack

```
fun:                          int fun(int x)
                              {
   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;

   ... compute ...               // compute using t0
                                 // 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                   }
```

Compiling C to MIPS

Using simplified C as an intermediate language make things easier for a human to produce MIPS code. However, it does not provide an automatic way of translating. This is provided by a *compiler* (e.g. `dcc`).

What does the compiler need to do to convert C to MIPS?

- convert `#include` and `#define`
- *parse* code to check syntactically valid
- manage a list of *symbols* used in program
- decide how to represent data structures
- allocate local variables to registers or stack
- map control structures to MIPS instructions

C Pre-processor

A C pre-processor maps C → C, performing various ***substitutions***

- **`#include`** *`File`* - replace `#include` by contents of file.
  e.g. "`name.h`" uses named *`File`*`.h`,
  `<name.h>` uses *`File`*`.h` in `/usr/include`
- **`#define`** *`Name Constant`* - replace all occurrences of symbol *`Name`* with *`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)`

Symbol Table Management

Compiler keeps track of names

- scope, lifetime, locally/externally defined
- disambiguates e.g. x in `main()` vs x in `fun()`
- resolves symbols to specific locations (data/stack/registers)
- external symbols may remain unresolved until linking
- however, need to have a type for each external symbol

Example:

```
double fun(double x, int n);

int main(void) {
   int i;  double res;
   scanf("%d", &i);
   res = fun((float)i, 5);
   return 0;
}
```

Local Variables

We have two choices for local variables. They can be:

- on the stack:
  + persist for whole function
  - lw/sw needed in MIPS
- in a register:

Can also pass a pointer to a struct

```
# Student stu;
# // set values in stu struct
# changeWAM(&stu, float newWAM);
.data
stu: .space 56
wam: .space 4
   .text
   ...
   la   $a0, stu
   lw   $a1, wam
   jal  changeWAM
   ...
```

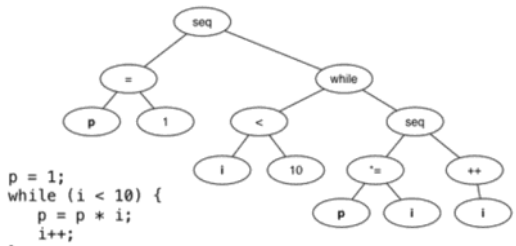
Clearly a more efficient way to pass a large struct
Also, required if the function needs to update the original struct

More C pre-processor substitutions

```
Before cpp                  After cpp
x = 5;                      x = 5;
#if 0                       x = x + 2;
x = x + 1;                  printf("x=%d\n",x);
#else                       x = x * 2;
x = x + 2;
#endif

                            Assuming ...
#ifdef DEBUG                #define DEBUG 1
printf("x=%d\n",x);         or
#endif                      gcc -DDEBUG=1 ...
x = x * 2;
```

C Parser

A C Parser Understands the syntax of C language. It attempts to convert a C program into ***parse tree***.




```
p = 1;
while (i < 10) {
   p = p * i;
   i++;
}
```

Mapping Control Structures

Mapping control structures use templates, e.g.

```
while (Cond) { Stat1; Stat2; ... }
loop:
   MIPS code to check Cond1; result in $t0
   beqz $t0, end_loop
   MIPS code for Stat1
   MIPS code for Stat2
   MIPS code for ...
   j    loop
end_loop:
```

Template for if...else if... else

```
if (Cond1) Stat1 else if (Cond2) Stat2 else Stat3
if:
   MIPS code to check Cond1; result in $t0
   beqz $t0, else1
   MIPScode for Stat1
   j    end_if
else1:
   MIPS code to check Cond2; result in $t0
   beqz $t0, else2
   MIPS code for Stat2
   j    end_if
```

- on the stack:
  + persist for whole function
  - lw/sw needed in MIPS
- in a register:
  + efficient
  - not many registers, useful if the variable is used in small scope

  If the variable needs to persist across function calls, use a `$s?` register.

  If it is used in very localised scope, you can use a `$t?` register.

Example:

```
int sum(List L)
{
   if (L == NULL) return 0;
   int first = L->value;     // must be in $s?
   int rest = sum(L->next);  // can be in $t?
   return first + rest;
}
```

Expression Evaluation

Expression evaluation uses temporary (`$t?`) registers. Complex expressions don't generally need > 3-4 registers.

Example:

```
x = ((y+3) * (z-2) * x) / 4;
lw   $t0, y
addi $t0, $t0, 3
lw   $t1, z
addi $t1, $t1, -2
mul  $t0, $t0, $t1
lw   $t1, x
mul  $t0, $t0, $t1
li   $t1, 4
div  $t0, $t0, $t1
```

Complex boolean expressions are handled by short-circuit evaluation.

```
   MIPS code to check Cond2; result in $t0
   beqz $t0, else2
   MIPS code for Stat2
   j    end_if
else2:
   MIPS code for Stat3
end_if:
```

Argc and Argv

The real MIPS machine has no idea about argc and argv

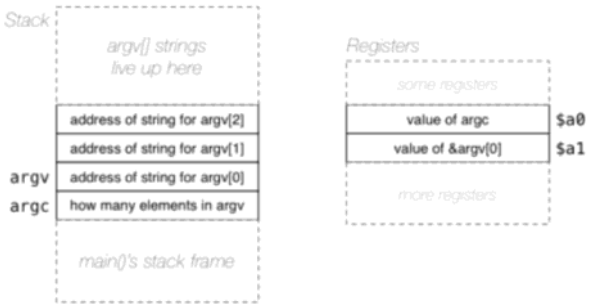
SPIM runs under Linux, and needs to interact with environment

So, the initialisation code (that invokes `main`) sets them up:

```
lw    $a0 0($sp)   # argc
addiu $a1 $sp 4    # argv
...
jal   main
nop
li    $v0 10
syscall            # syscall 10 (exit)
```

Note: we are ignoring `envp` (environment pointer).

What the main program receives:



Code to print the program's name (argv[0]):

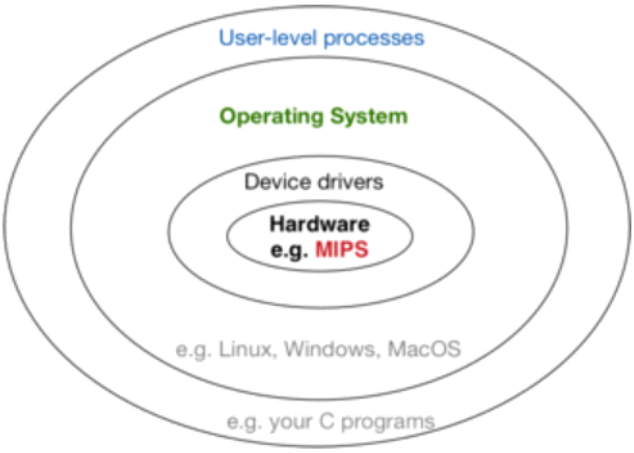
```
# assume argc is $s0, argv is $s1
addi   $t0, $s1, 0    # &argv[0]
lw     $a0, ($t0)     # argv[]
li     $v0, 4
syscall
```

Code to print the first cmd-line arg (argv[1])

```
# assume argc is $s0, argv is $s1
addi   $t0, $s1, 4    # &argv[1]
lw     $a0, ($t0)     # argv[]
li     $v0, 4
syscall
```

Computer Systems Architecture

Tuesday, 21 August 2018 7:59 PM



Evolution of Operating Systems (OSs)

1940's (e.g. ENIAC)

- no OS ... one program at a time, manually loaded
- programs had to take account of details of machine/devices

1950's (e.g. Whirlwind)

- batch processing ... load several programs at once, run in sequence
- programs had to take account of details of machine/devices

1960's (e.g. IBM360)

- computers proliferate ... programmers want to transport code
- having to cope with different config on each machine was tedious
- solution: layer of software between raw machine and user programs

Nice example of using abstraction to enhance code portability

1970's (e.g. PDP-11)

- computers become smaller and faster (but still fridge-size)
- complexity of 1960's OSs drove Bell Labs researchers to
    - develop a small OS core, written mostly in HLL
    - with a set of simple tools and ways of combining them
    - led to the Unix programming environment
- writing OS core in HLL made it portable
    - same OS environment provided on many different machines

Other less-portable OSs came and went ...

1980's - present

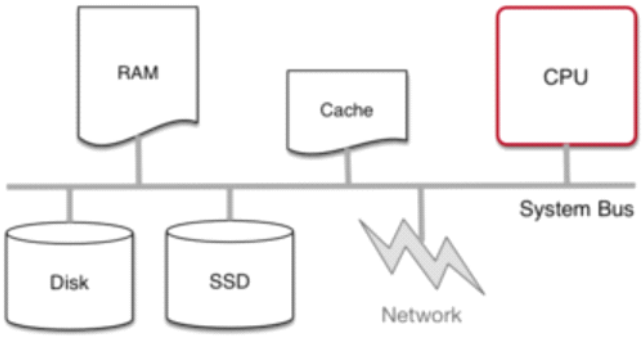
- Unix and variants ported to wide variety of architectures
    - BSD, SysV, Linux, OS X, Android all based on Unix approach
- developments in hardware/software led to more OS services
    - databases, O-O programming, GUI interfaces, games
    - networking, multi-CPU systems, mobile devices, etc.

Today: very large installed base of Unix systems

- embedded systems, phones, workstations, servers, ...

Operating Systems

A modern computer has devices connected via a system bus:



Operating systems (OSs) provide an abstraction layer on top of hardware. i.e. the same view is available regardless of the underlying hardware.

Some characteristics of OSs are that they:

- have ***privileged access*** to the raw machine
- ***manage*** use of machine ***resources*** (CPU, disk, memory, etc.)
- ***provide uniform interface*** to access machine-level operations
- arrange for ***controlled execution*** of user programs
- provide ***multi-tasking*** and (pseudo) ***parallelism***

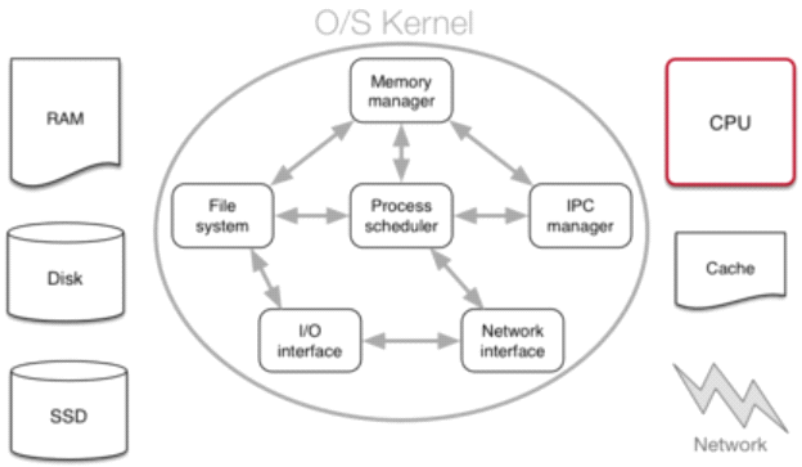
Some different flavours of OSs are:

- ***Batch*** (e.g. Eniac, early IBM OSs) - computational jobs run one-at-a-time via a queue
- ***Multi-user*** (e.g. Multics, Unix/Linux, OSX, Windows) - multiple jobs (appear to) run in parallel
- ***Embedded*** (e.g. Android, iOS, ...) - a small(ish), cut-down OS embedded in a device
- ***Real-time*** (e.g. RTLinux, DuinOS, ...) - specialised OS with time guarantees on job completion

Some abstractions provided by modern OSs include:

- Users - who can access (login to) the system
- Access rights - what the users are allowed to do
- File system - how data is organised on storage devices
- Input/output - transferring data to/from devices
- Processes - active "computational entities" on the system
- Communication - how processes interact
- Networking - how the system talks to other systems

These core OS functions form the operating system ***kernel***.



Execution modes are critical to OS development.
CPUs can typically run in two modes:

- Privileged mode - this allows full access to all machine operations and memory regions. This is the mode in which the OS runs. It has access to all resources in the computer.
- Non-privileged (user) mode - this allows a limited (but still Turing complete) set of operations and access to only part of the memory. The operations available can be executed by any application/user.

System Calls

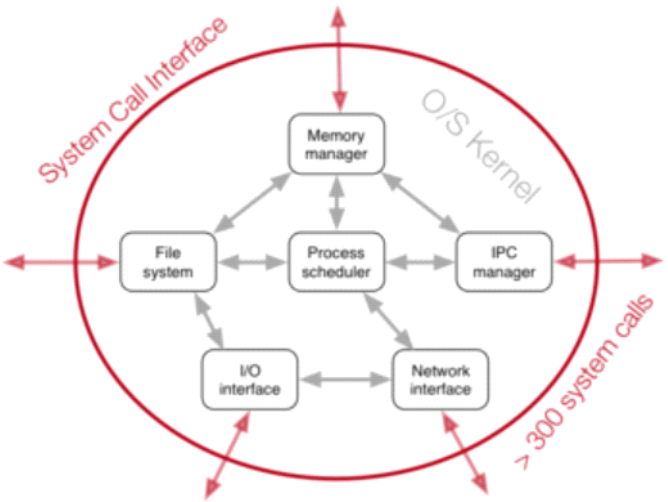
SPIM has no OS, but provides a simple set of "system calls". These system calls are primarily for I/O (read/write) on various types. They are also used for memory allocation and process exit.

A ***system call*** is a way in which a program requests a service from the kernel of the OS it is operating on. It is a way for programs to interact with the operating system. It provides an interface between a process and an operating system to allow user-level processes to request services of the OS. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

An OS like Unix/Linux provides 100's of system calls:

- Process management (e.g. `fork(), exec(), _exit(), ...`)
- File management (e.g. `open(), read(), fstat(), ...`)
- Device management (e.g. `ioctl(), ...`)
- Information maintenance (e.g. `settimeofday(), getuid(), ...`)
- Communication (e.g. `pipe(), connect(), send(), ...`)

User programs invoke sys calls through an Application Program Interface (`API`) (`POSIX + Linux`)



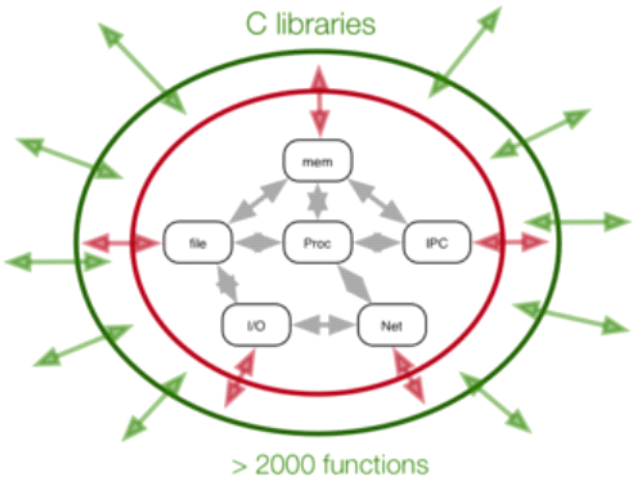
Libraries

User programs can request services via system calls, but system calls provide relatively low-level operations.

To simplify programming, ***libraries*** are provided. Libraries are collections of useful functions that interact with hardware through system calls. The functions are referenced within user programs such as C functions. They are defined by `#include <xxx.h>` in C code and are integrated with user code at "link time".

Some examples of libraries:

- `stdio.h` - text-oriented, formatted input/output
- `stdlib.h` - wide range of functions e.g. `rand(), malloc()`

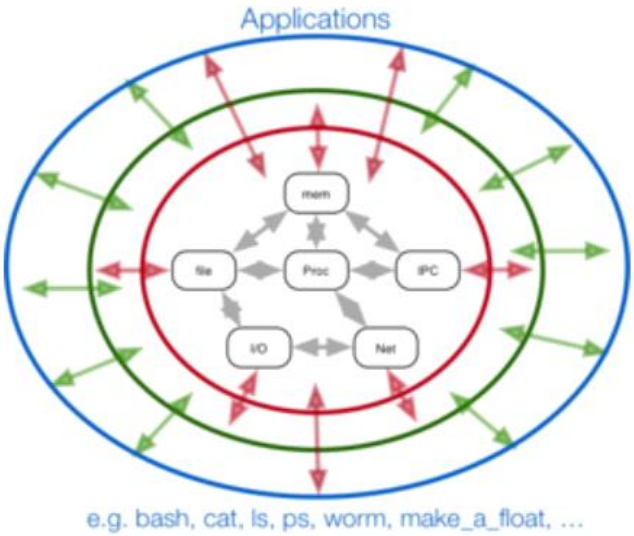


Applications

Applications are user-level programs, which perform some useful task(s). They can be supplied with the system (e.g. `ls`, `vim`, `gcc`), or they can be implemented by users (e.g. `dcc`, `check`, `Webcms3`).

Applications live in `/bin, /usr/bin` etc. via a PATH.
Applications are generally built using libraries, but they may also make direct use of system calls.

Unix was unusual in having a command interpreter (`bash`) that runs as a user-level process (not privileged) but can invoke other user-level processes.



System calls are invoked:

- directly, through a library of system calls, which are documented in the Unix Programmers Manual section 2 (e.g. `man 2 open`)
- indirectly, through functions in C libraries, which are documented in the Unix Programmers Manual section 3 (e.g. `man 3 fopen`)

Example of system call library vs C library

- file descriptors, `open(), close(), read(), write()` (via `#include <unistd.h>`)
- file pointers `(FILE*), fopen(), fclose, scanf(), printf()` (via `#include <stdio.h>`)

System calls attempt to perform actions, but they may fail.
User programs can detect this inseveral ways;

- checking the return value of a sys call function (-1 typically flags an error)
- checking the global variable `errno`

C programs need to check and handle errors themselves. Unlike other languages, C provides no exception handling.
Note: successful system calls generally return 0, unless the system call has a result value (e.g. no. of bytes read)

An action in response to a failed system call is often e.g.

```
fprintf(stderr, "Can't do %s", Something);
exit(1);
```

We can give more precise feedback via library functions, e.g.

```
void perror(char *Message)
```

if *Message* is not NULL, it will write to `stderr`.
It will write a standard message corresponding to `errno`.
The we could then do `exit(errno);` to send precise error to the shell.

Linux library function to make it easy to report errors and exit. e.g.

```
error(Status, ErrNum, Format, Expressions, ...)
```

which prints an error message using the program name, *Format* and *Expressions*
If *Status* is non-zero, we invoke `exit(Status)` after printing message.
If *ErrNum* is non-zero, it will also print a standard system error message.
Example:

```
error(1, errno, "Can't do %s", Something);
vs
```

```
char errmsg[BUFSIZ];
sprintf(errmsg, "Can't do %s", Something);
perror(errmsg);
exit(1);
```

Roadmap

The following sections consider OS modules:

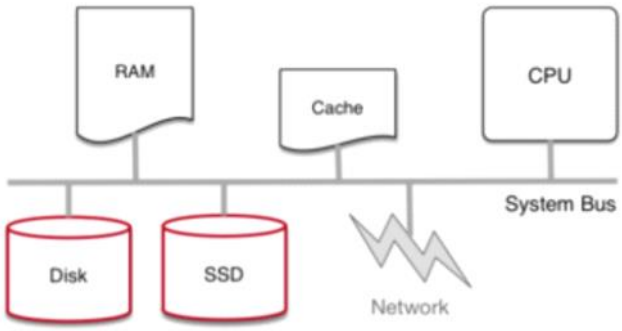
- ***Storage management*** - disk storage, filesystems, Unix file system interface
- ***Memory management*** - RAM, physical/virtual/process memory space, page faults
- ***Process management*** - creating processes, process hierarchy, signals
- ***Device management*** - control registers, data transfer, interrupts

Operating System Components

Wednesday, 22 August 2018 4:11 PM

Storage Management

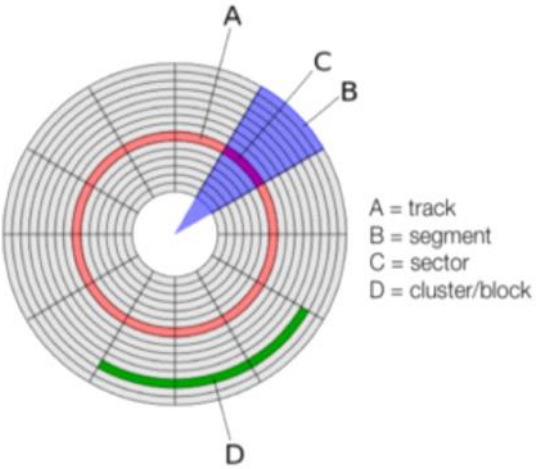
Computer Systems have *bulk, persistent* storage devices.



Disks

Hard Disk Drives (HDDs) have the following characteristics:

- magnetic medium, arranged in concentric *tracks*
- usually multiple surfaces for storing data
- rotates at very high speed (e.g. 7200rpm)
- each track divided into fixed size *sectors*
  - outer tracks have more sectors than inner tracks
- read/write head which can read/write one sector at-a-time



To access data on disks (one sector at-a-time), the head moves to a track and waits for a sector to rotate underneath it.
Then it reads/writes data from/to the disk surface.

Disk access costs:

- seek time (move to track) in range 0ms .. 12ms
- rotational latency (wait for sector) in range 2ms .. 4ms
- data transfer ... typically 200 MB/s (relatively small)
- Overall: tens of milliseconds for a single access

Some typical (and historical) measures of HDDs:

- Capacity - 4TB; available storage is 3.5TB
- Size - 2.5/3cm; volume ≈ 34cm^3; weight = 60g
- Speed - ~20ms to find a sector; <1ms to read a sector

Compared to 1958:

- Capacity - 3MB; available storage is 2.5MB
- Size - 1.5m; volume ≈ 2m^3; weight = 900kg
- Speed - ~600ms to find a sector; 5ms to read a sector

Disks are ***block-oriented*** devices. All data transfers read/write a fixed number of bytes, even if only one byte is required. A typical transfer size is 512 byte to 4096 bytes.

The OS views data on disk as logical blocks, each block being compromised of one or more sectors. The sectors are arranged to minimise head movement/latency

SSDs

Solid State Drives (SSDs) are fully-electronic devices that can store data persistently.

- don't have mechanical delays that HDDs do (R/W much faster)
- have better thermal performance than HDDs (e.g. no fans needed)
- controller more complex/sophisticated than HDD controller
- relatively expensive per byte of storage (compared to HDDs)
- limited number of read/writes on each block before failure
- data "leaks" over time (over years ... but worse than HDDs)
- writing requires erase-then-overwrite
- susceptible to elector-magnetic fluctuations (e.g. power-outs)

Useful comparison between HDD and SSD on Wikipedia

OSs can view SSDs like HDDs; as block oriented devices but much faster than HDDs (e.g. 0.1ms), but have a limited number (10^5) of erases on each lock before failure. They find it easier to access logical blocks of SSDs (it can access multiple SSD blocks) and they have less problems than HDDs.

Many SSDs are packaged with similar form-factor to HDD; drop-in, high-speed replacement for HDD

Operating Systems and Storage

Operating systems manage data stored on bulk storage devices. This provides an abstract view of devices, hiding details of device control and data transfer.

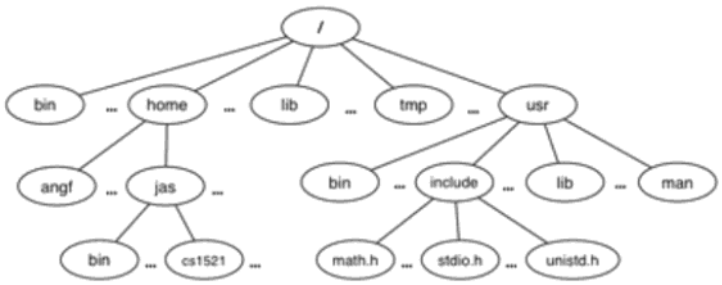
File Systems

File systems provide a mechanism for managing *stored data*:

- typically on a disk device (or, nowadays, on SSD)
- allocating chunks of space on the device to *files*
  - where a file is viewed as a sequence of bytes
- allowing access to files by *name* and with *access rights*
- arranging access to files via *directories* (folders)
- maintaining information about files/directories (*meta-data*)
- dealing with damage on the storage device ("bad blocks")

Unix/Linux File System

The Unix/Linux file system is tree-structured



Processes have a notion of their location within the file system; the ***current working directory*** (CWD)

The file system is used to access various types of objects; files, directories (folders), devices, processes, sockets, etc. These objects are referenced via a ***path*** (…/x/y/z/…). The paths can be:

- ***absolute*** - a full path from root. e.g. `/usr/include/stdio.h, /home/jas/cs1521/`
- ***relative*** - a path which stars from CWD. e.g. `../../another/path/prog.c, …/a.out, a.out`

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

- **`off_t`** - offsets within files. They are typically long and signed to allow backward references
- **`size_t`** - the number of bytes in some object. They are unsigned, since objects can't have negative size.
- **`ssize_t`** - the size of read/written blocks. Like size_t, but signed to allow for error values
- **`struct stat`** - a file system object metadata. It stores information about a file, but does not store the contents of the file. It requires `ino_t, dev_t, time_t, uid_t, ...`

Metadata for file system objects is stored in ***inodes.*** It stores:

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

Each file system volume has a table of inodes in a known location. If you delete a file, the inode will still exist as long as there are references to the deleted file.

Note: an inode does not contain the name of the file

Access to a file by name requires a *directory,* where a directory is effectively a list of (name, inode) pairs

Accessing to files by *name* goes like this:

1. open the directory and scan for *name*

2. if *name* is not found, "No such file or directory"
3. if found as (*name*,ino), access inode table `inodes[ino]`
4. collect file metadata and:
    a. check file access permissions have been given to the current user/group
        i. if they don't have required access, "Permission denied"
    b. collect information about file's location and size
    c. update access timestamp
5. 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, and they are* accessed via a *file descriptor* (index into a system table).

Some common operations include:

- `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
- `mount()` - place a filesystem on a device

`open()/close()`

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

- `open()` attempts 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`

Could be unsuccessful if *`FileDesc`* is not an open file descriptor

An aside: removing an object e.g. via `rm` removes the object's entry from a directory, but the inode and data persist until all processes that are accessing the object `close()` their handle, and all references to the inode from other directories are removed. After this, the inode and the blocks on storage device are recycled.

Example using `open()` and `close()`

```
// count lines
#include <unistd.h>
#include <fcntl.h>
#include <error.h>
#include <errno.h>

char buf[1000];  int n, nl = 0;
int fd = open("myFile", O_RDONLY);
if (fd < 0)
   error(errno,errno,"Can't open %s","myFile");

while ((n = read(fd, buf, 1000)) > 0) {
   for (int i = 0; i < n; i++)
```

```
      if (buf[i] = '\n') nl++;
}
close(fd);
…
```

read()/write()

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

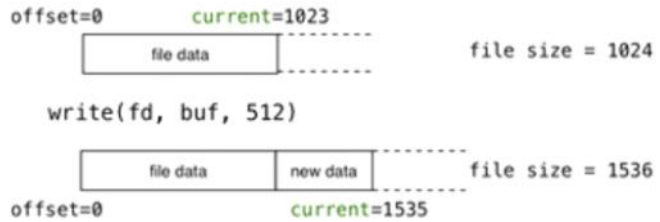
- attempt to read *Count* bytes from *FileDesc* into *Buffer*
- if "successful", return number of bytes actually read (*NRead*)
- if currently positioned at end of file, return 0
- if unsuccessful, return -1 and set errno
- does not check whether *Buffer* contains enough space
- advances the file offset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

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



Functions from stdio.h tend to be char-oriented.
File-descriptor-based system calls deal with byte sequences. The bytes can be interpreted as char, int, struct, etc, so, many kinds of objects can be read() or write() **
This allows programmers to manipulate files of data items, e.g. list of double values read from sensor device, and a collection of Student records.

** you cannot save/restore pointer values using write()/read() because they refer to memory addresses within a process instance and a different process instance might already have used those addresses

Files of *records* can be produced by either,

- write()ing chunks of bytes from struct objects or,
- printing formatted text representation of struct data.

The latter approach is a form of *serialisation.*
For the write() approach, there is no need to worry about formatting issues. It writes entire structures, even if the string buffers is half empty. You can lseek() to i^{th} struct via *i*sizeof(StructType)*

For the printing approach:

- produces files that are human-readable
- only uses as many bytes as required from string buffers
- can access structures only sequentially (unless using padding)

Example of write()ing records vs printf()ing records

```
typedef struct _student {
    int id; char name[99]; float wam;
} Student;
int infd, outfd;
FILE *inf, *outf;
```

```
Student stu;

write(outfd, &stu, sizeof(struct _student));
  vs
fprintf(outf, "%d:%s:%f\n", stu.id, stu.name, stu.wam);

read(infd, &stu, sizeof(Student));
  vs
fscanf(inf, "%d:[^:]:%f\n", &(stu.id), &(stu.name), &(stu.wam));
```

seek()

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

Sets 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)
`lseek()` returns how far into the file you are.

Example:

```
// move around the file
#include <unistd.h>
...
lseek(fd, 0, SEEK_SET);    // go to start of file
lseek(fd, 0, SEEK_END);    // go to end of file
lseek(fd, -20, SEEK_CUR);  // go back 20 bytes
lseek(fd, 100, SEEK_SET);  // go to 100 bytes from the start
```

stat()

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

Stores meta-data associated with *FileName* into *StatBuf*
The 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

It returns `-1` and sets `errno` if meta-data not accessible.
If *FileName* is a symbolic link, `stat()` returns the information about the referenced file.
Note: `stat` is short for status

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

Is the same as `stat()` but it gets data via an open file descriptor.

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

Is the same as `stat()` but it doesn't follow symbolic links

Links

File system *links* allow multiple paths to access the same data.

Hard links are multiple directory entries referencing the same inode. That is, you have two separate names referring to the same object, where it can be hard to distinguish which is the original. The two entries must be on the same filesystem.

Symbolic links (symlinks) are a file containing the path name of another file. In a sense, it is an *alias* for a file. Opening the symlink opens the file being referenced.
To set up a symbolic link:

```
ln -s fileName referencedFileName
```

Example:

```
-rw-r----- 2 cs1521 46 Sep 10 22:28 fileA
-rw-r----- 2 cs1521 46 Sep 10 22:28 fileB
lrwxrwxrwx 1 cs1521  5 Sep 10 22:29 fileC -> fileA
```

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
};
```

The st_mode is a bit-string containing some of:

```
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)`

Creates 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

It returns `0` if it is successful, and 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.

Note also: `fflush()` forces `stdio` buffers to be copied to kernel buffers.

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

A 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) and it 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,...)`
(use disk5 to hold the /usr file system as read-only ext3-type)

File System Summary

Operating systems provide a *file system* as an abstraction over physical storage devices (e.g. disks), by:

- 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 meta-data associated with files (size, location, ...)

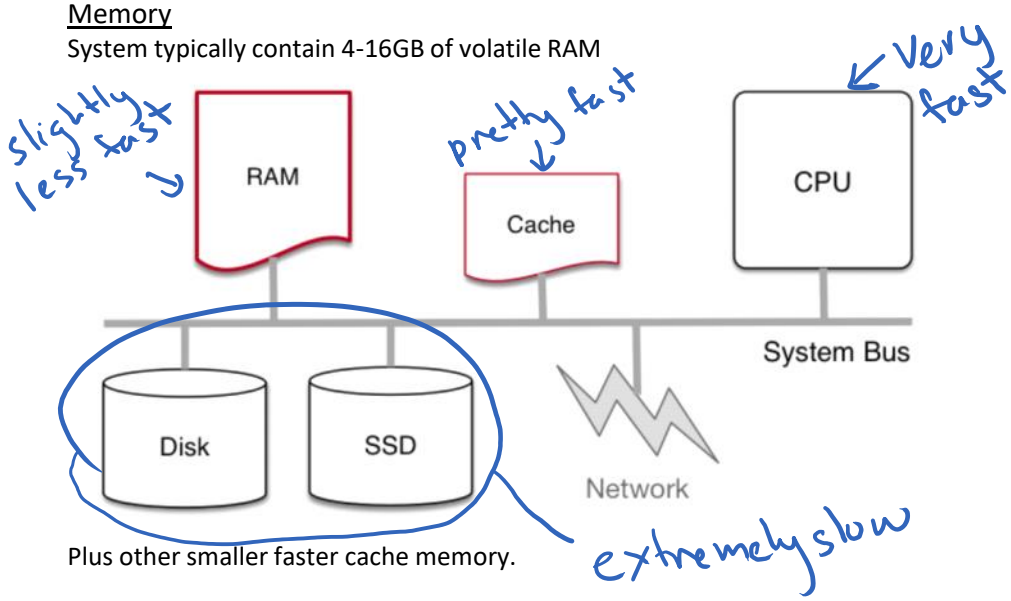
Operating systems also manage other resources

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

Memory Management

Memory

System typically contain 4-16GB of volatile RAM



Plus other smaller faster cache memory.

Processes

A process is an active computation, consisting of :

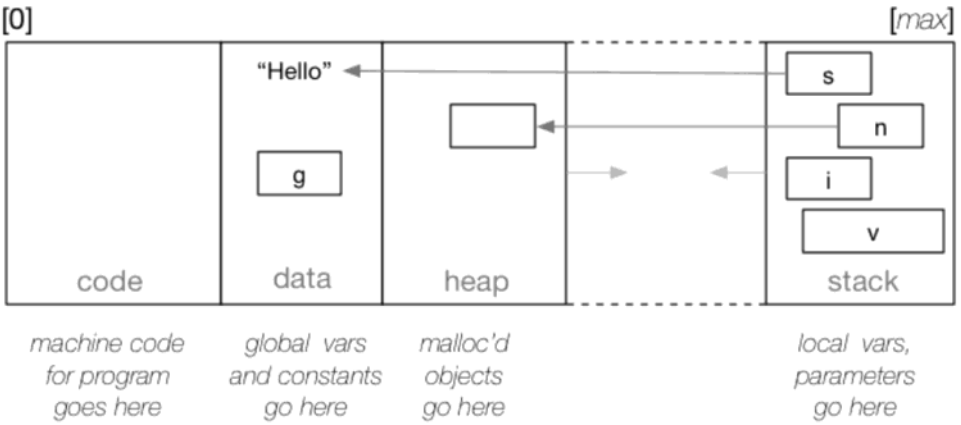
- RAM: code segment (read-only), data segment (read/write)
- Registers: program counter (PC) and other registers.
- Other management information, which we will discuss later

Altogether they comprises the current execution state of the program.

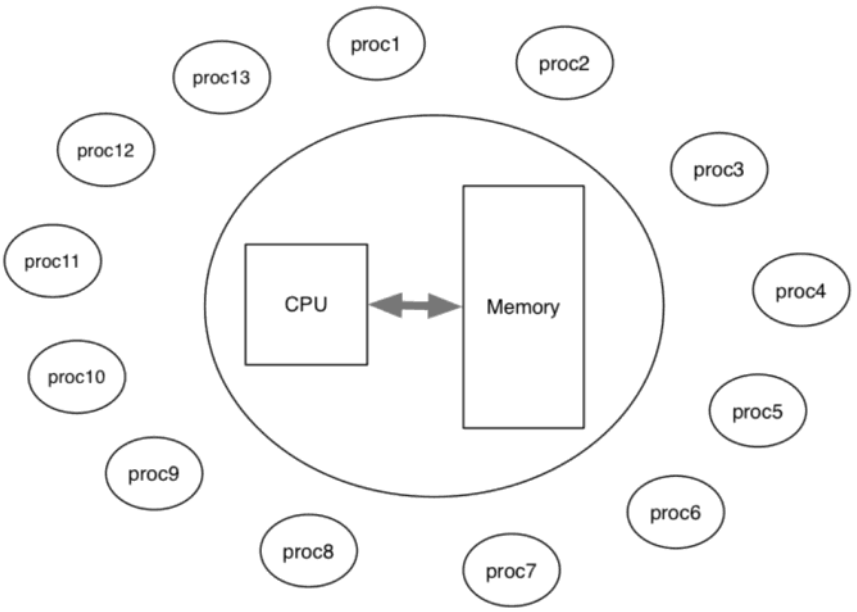
Multiple processes can be *active* simultaneously (in the sense that they have not completed their computation). Typically they are not all loaded in RAM at once. Processes can be suspended (waiting).
Restoring a process: load code, data, registers
Each process wants a view that it is the only one on the system.

Memory Management

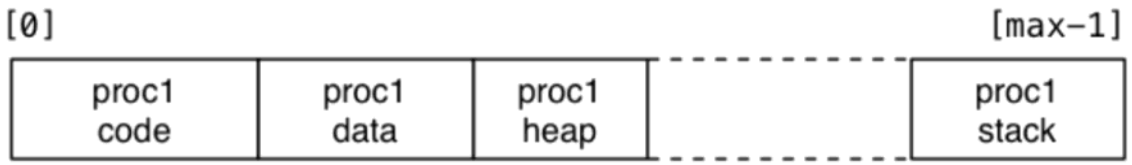
Operating systems provide a view of memory for individual processes



On a system with, say, 1 CPU, 1 memory and 100's of processes



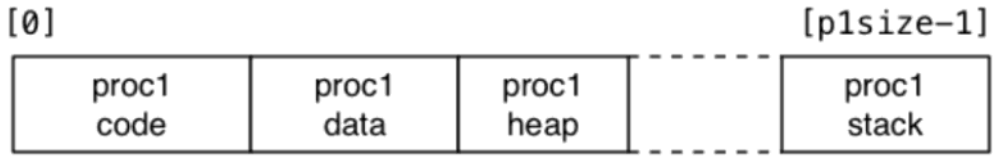
In the good-ol-days, one process/computation was done at a time.
The process can use the entire memory, and the addresses within the process code are absolute.

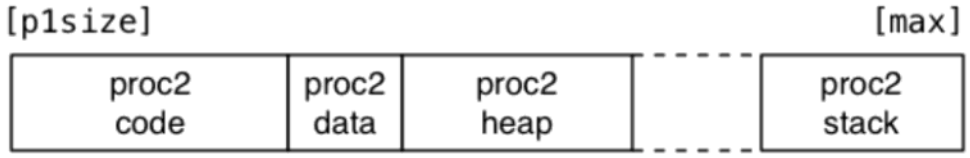


Or, if the process did not need the entire memory

It is easy to implement by initialising `$sp` to `psize-4`

When two processes loaded into memory at once, the addresses in `proc1` are absolute, and the addresses in `proc2` need to be interpreted relative to `p1size.`



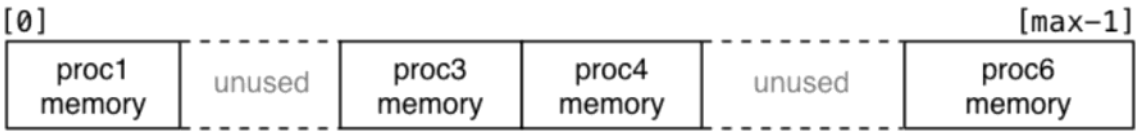


When `proc1` is loaded, copy code+data memory, starting at address 0. then set the stack pointer to `p1size-4`.

How to sort out `proc2` addresses? (which assumes start `addr = 0`)

- "fix" them when process code+data is loaded. That is, replace each address in code by `addr+p1size`
- "map" addresses during execution. That is, after a memory address is computed in machine code, increment it by `p1size` before accessing memory. This requires extra hardware (holding `proc2`'s start address)

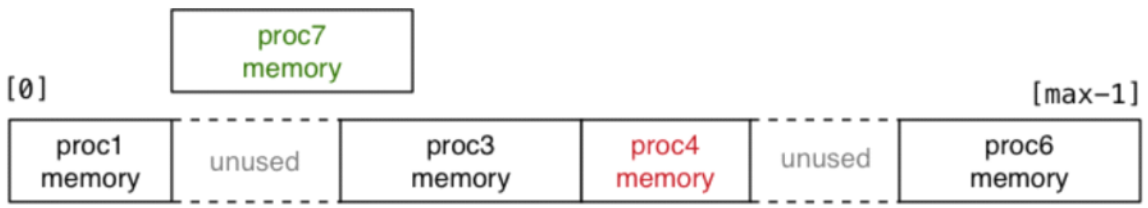
Consider a scenario with multiple processes loaded in memory:



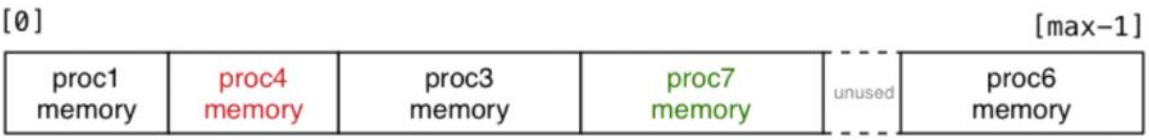
If we do on-the-fly address mapping, we need to remember the *`base`* address for each process (process table). When the process (re)starts, load *`base`* into mapping hardware. Interpret every address *`addr`* in the program as *`base+addr`*.

Each process sees its own address space as [`0..p1size-1`].
The process can be loaded anywhere in memory without change

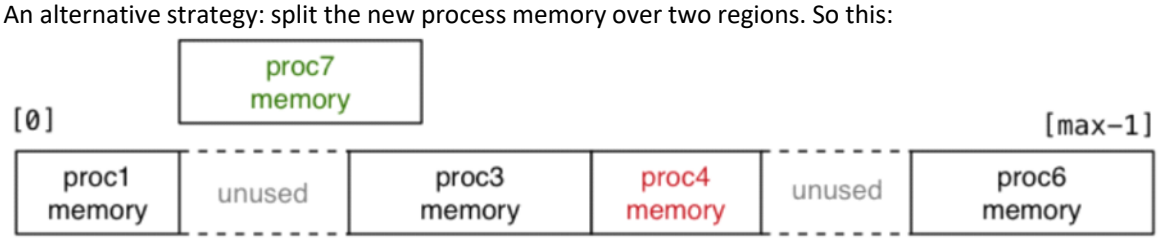
Consider the same scenario, but we want to add a new process.



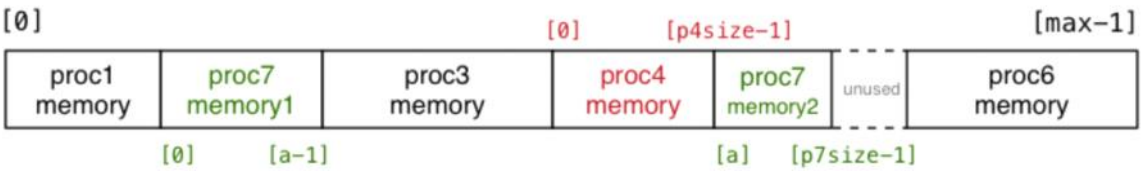
The new process doesn't fit in any of the unused slots.
We could move some processes to make a single large slot.



An alternative strategy: split the new process memory over two regions. So this:



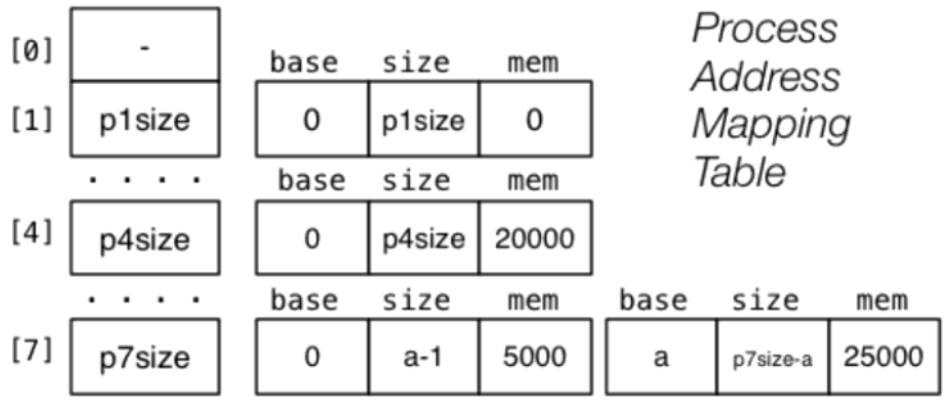
becomes:



Implications for splitting process memory across physical memory

- each chunk of process address space has its own *base*
- each chunk of process address space has its own *size*
- each chunk of process address space has its own *memory* location

We need a table of process/address information to manage this, e.g.



Note: base refers to a base within the process address space, not the offset within the physical memory of where the chunk starts

Under this scheme, address mapping calculation is complicated

```
Address processToPhysical(pid, addr)
{
   Chunk chunks[] = getChunkInfo(pid);
   for (int i = 0; i < nChunks(pid); i++) {
      Chunk *c = &chunks[i];
      if (addr >= c->base && addr < c->base+c->size)
         break;
   }
   uint offset = addr - c->base;
   return c->mem + offset;
}
```

The above mapping *must* be done in hardware to be efficient.

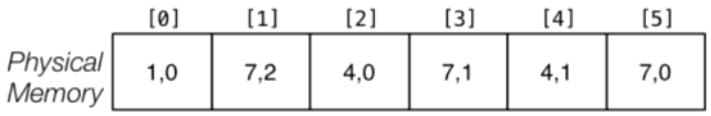
Page-based address mapping:

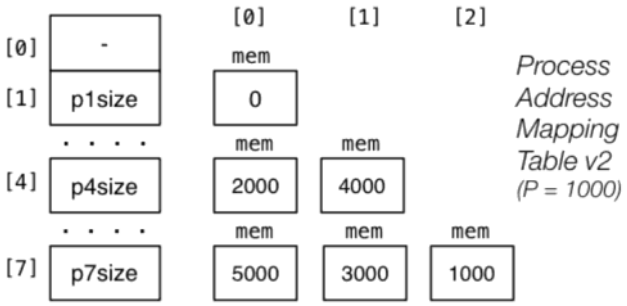
Address mapping would be simpler if all chunks were same size, so partition process address space into fixed-sized chunks. We call each chunk of address space a *page*. All pages are the same size *P* (*PageSize*), so process memory is spread across $\lceil \frac{ProcSize}{P} \rceil$ pages. Page i has addresses *A* in range $i*P \leq A < (i+1)*P$

This also leads to a simpler address mapping table:

- each process has an array of page entries
- each page entry contains start address of one chunk
- can compute index of relevant page entry by $(\frac{A}{P})$
- can compute offset within page by (*A % P*)

An example of simple mapping table:





Memory Management Review

Reminder: process addresses ↔ physical addresses

- process has (virtual) address space `0..N-1` bytes
- memory has (physical) address space `0..M-1` bytes
- both address spaces partitioned in P byte pages
- process address space contains K = ⌈N/P⌉ *pages*
- memory address space has L = ⌈M/P⌉ *frames*

Mapping:

- takes virtual address (`Vaddr`) in process address space
- returns physical address (`Paddr`) in memory address space

Address Mapping

Mapping from process address to physical address:

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

Computation of `pageno`, `offset` is efficient if `Pagesize == 2`^{n}

Note that we assume `PageInfo` entries with more information

Exercise 3: Process→Physical Address Mapping

Consider process **p7** from previous slide has three pages: p0 @ 5000, p1 @ 3000, p2 @ 1000

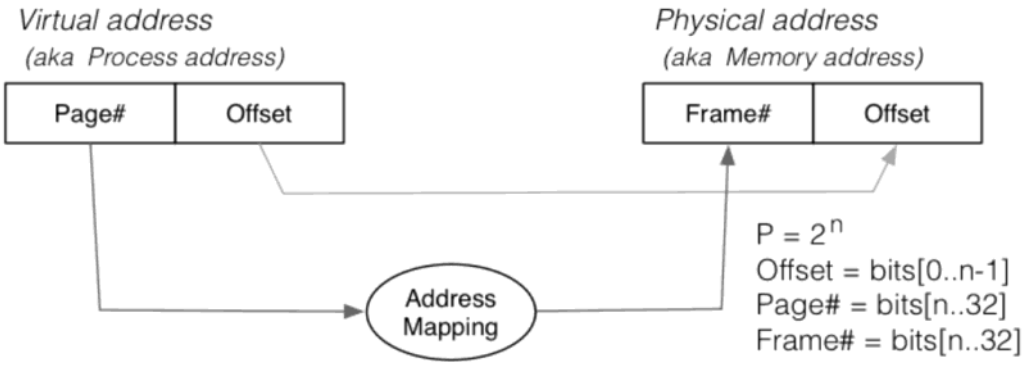
Page size is 1000

For each of the following process addresses, where is it located in physical memory?

a. **0x0000** = 0 5000
b. **0x0080** = 128 5128
c. **0x0400** = 1024 3024
d. **0x0888** = 2184 1184
e. **0x1000** = 4096 cannot be calculated

Page table entries typically do not store physical address. To save space, they just store frame number F and compute the physical address via (`P * F + Offset`)

If `P == 2`^{n}, then address mapping becomes



We can calculate the physical address more efficently when using bit manipulation:
Let P = 2^{12} , so the last 12 bits of a 32bit address would be the offset.
Assume that `PageInfo = (status,frameNo)`

```
Address processToPhysical(pid, Vaddr)
{
   uint offset = Vaddr & 0x0FF;
   unit pageno = Vaddr >> 12;
   PageInfo mapping[] = getPageInfo(pid);
   uint frameno = mapping[pageno].frame;
   uint Paddr = (frameno << 12) | offset;
   return Paddr;
}
```

Virtual Memory

A side-effect of this type of virtual→physical address mapping is that we:

- 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 and on-demand loading of process pages into physical memory is called ***virtual memory***.

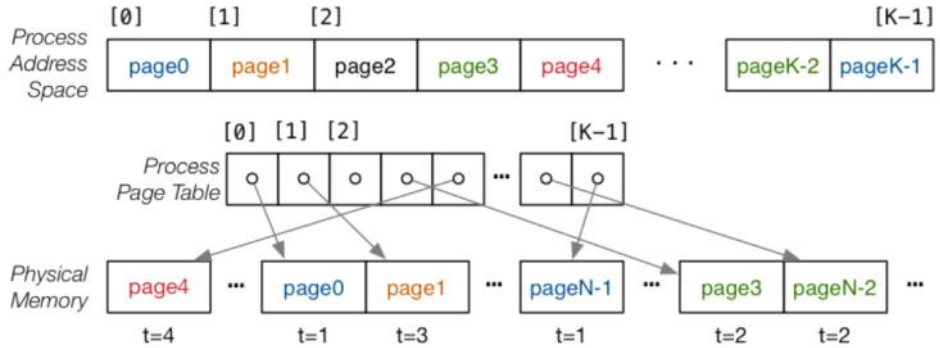
Pages/frames are typically 512B .. 8KB in size
In a 4GB memory, would have ≅ 4 million × 1KB frames
Each frame can hold one page of process address space
Leads to a memory layout like this (with *L* total pages of physical memory):

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

How to arrange mapping process address → physical address?
Consider a per-process page table, e.g.

- each page table entry (PTE) contains
  - page status - Loaded, IsModified, NotLoaded
  - frame number of page (if Loaded)
  - ... maybe others ... (e.g. last accessed time)
- we need $\lceil \frac{ProcSize}{PageSize} \rceil$ entries in this table

Example of page table for one process:



Timestamps show when the pages were loaded.

Virtual address to physical address mapping (more detail):

```
typedef struct {char status, uint frameNo, ...} PageData;

PageData *AllPageTables[maxProc];
    // one entry for each process

Address processToPhysical(pid, Vaddr)
{
   PageData *PageTable = AllPageTables[pid];
   uint pageno = PageNumberFrom(Vaddr);
   uint offset = OffsetFrom(Vaddr);
   if (PageTable[pageno].status != Loaded) {
      // load page into free frame
      // set PageTable[pageno]
   }
   uint frame = PageTable[pageno].frameNo;
   return frame * P + offset;
}
```

An Aside: Working Sets

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?

From observations of running programs …

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

Known as the ***working set*** model (cf *locality of reference*)
We only need to hold, at a given time, the process's working set of pages
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

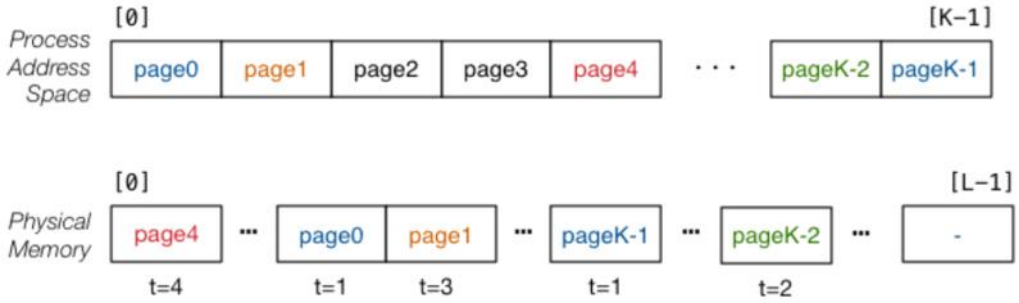
Virtual Memory continued…

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
- global data is also initially loaded from here
- dynamic (heap, stack) data is created in memory

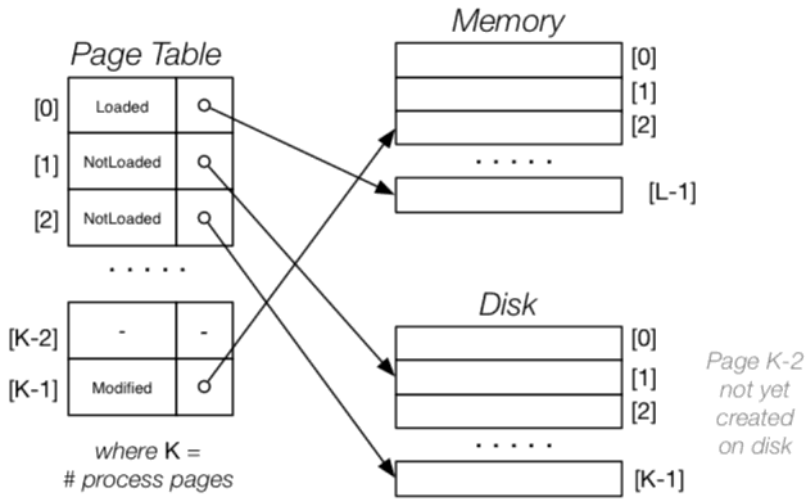
Consider a process whose address space exceeds physical memory.
The pages of dynamic data not currently in use may need to be removed temporarily from memory (see later), thus it would also be saved on disk and restored from disk.
We can imagine that a process's address space exists on disk for the duration 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, so we need to ensure minimal reading from/writing to disk.

Per-process page table, allowing for some pages to be not loaded



Recall the address mapping process with per-process page tables.

```
Address processToPhysical(pid, Vaddr)
{
   PageData *PageTable = AllPageTables[pid];
   uint pageno = PageNumberFrom(Vaddr);
   uint offset = OffsetFrom(Vaddr);
   if (PageTable[pageno].status != Loaded) {
      // load page into free frame
      // set PageTable[pageno]
   }
   uint frame = PageTable[pageno].frameNo;
   return frame * P + offset;
}
```

What to do if the page is not loaded?

Page Faults

Requesting a non-loaded page generates a ***page fault***.
One approach to handling a page fault is to find a free (unused) page frame in memory and use that:

```
   // load page into free frame
   else {
      frameno    = getFreeFrame();
      p->frameNo = frameno;
      p->status  = Loaded;
   }
```

This assumes that we have a way of quickly identifying free frames
This is commonly handled via a *free list.*
Reminder: frames allocated to a process become *free* when process exits

What happens if there are currently no free page frames?
What does `getFreeFrame()` do?
Possibilities:

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

Suspending requires the process manager to

- maintain a (priority) queue of processes waiting for pages
- dequeue and schedule the first process on queue when page freed

Will discuss process queues further in next section.

Page Replacement

What happens when a page is replaced?
If it's been modified since loading, it is saved to disk ** (in the disk-based virtual memory space of the running process). Then we grab its frame number and give it to the requestor.

How to decide which frame should be replaced?

- define a "usefulness" measure for each frame
- grab the frame with lowest usefulness (e.g. priority queue?)

** we need a flag to indicate whether a page is modified

```
#define NotLoaded   0x00000000
#define Loaded      0x00000001
#define IsModified  0x00000002
```

Factors to consider in deciding which page to replace

- 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)

The 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).
A useful heuristic is ***LRU replacement*** - a page not used recently may not be needed again soon

Factors for choosing best page to replace are *heuristic*
What happens if ...

- we replace a page which is soon used again
- this causes us to replace another page
- and the second page is soon used again .....

This is known as ***thrashing***, where we are constantly swapping pages in and out of memory.
The working set model plus LRU helps avoid *thrashing:*
A recently used page is likely to be used again soon, while a not recently used page is unlikely to be used again soon.

LRU is one replacement strategy. Others include:
First-in-first-out (FIFO)

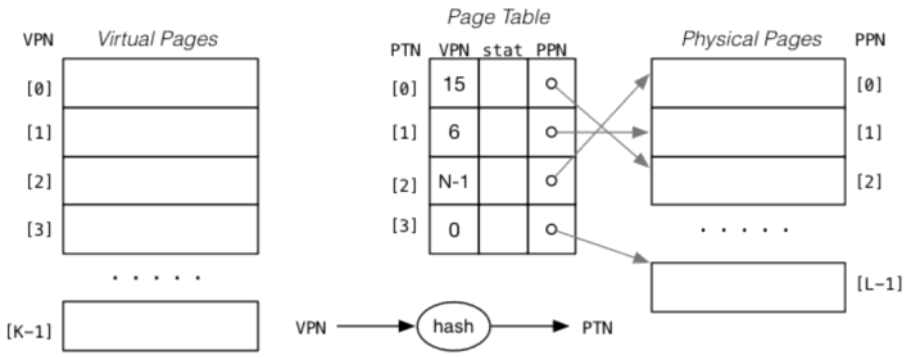
- page frames are entered into a queue when loaded
- page replacement uses the frame from the front of the queue

Clock sweep

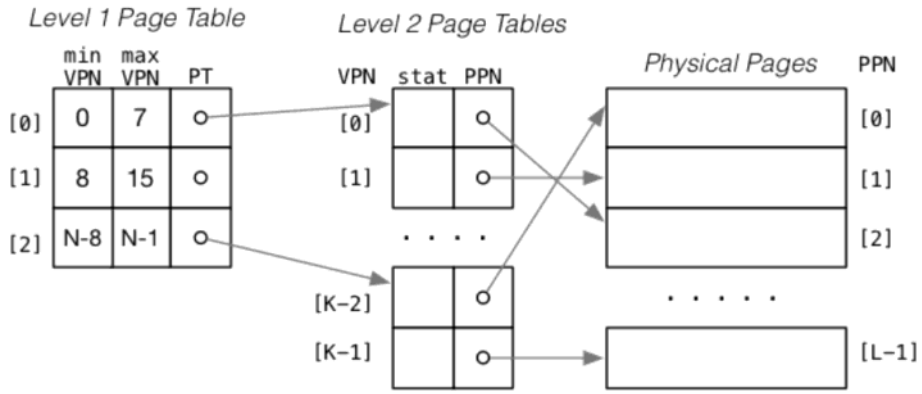
- uses a reference bit for each frame, updated when page is used
- maintains a circular list of allocated frames
- uses a "clock hand" which iterates over page frame list
    - skipping and resetting reference bit in all referenced pages
- page replacement uses first-found unreferenced frame

More Virtual Memory

Page tables (PTs) revisited:
A virtual address space with K pages needs K PT entries. Since K may be large, we do not want to store whole PT, especially since working set tells us $n \ll K$ needed at once.
One possibility: PT with $n < K$ entries and hashing

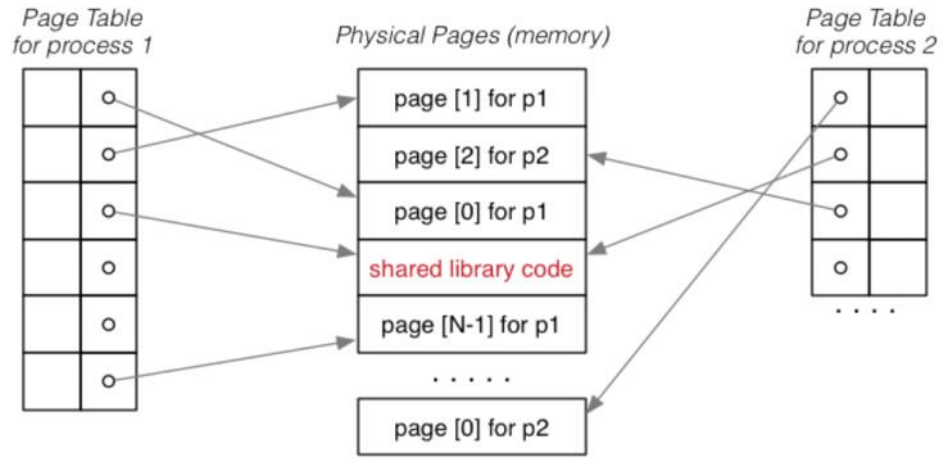


An alternative strategy: multi-level page tables



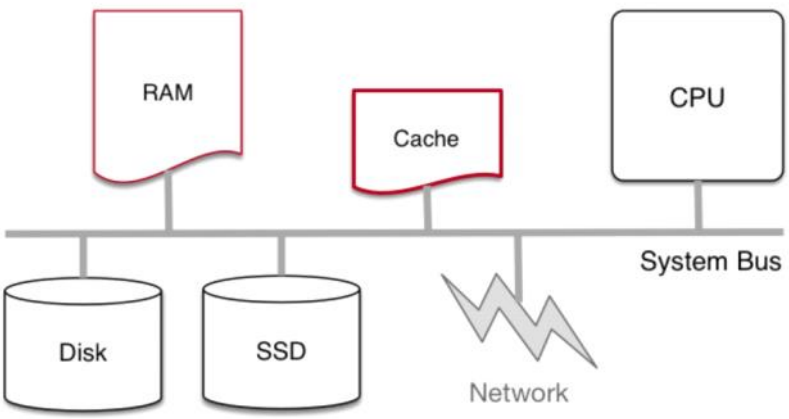
This is effective because not all pages in virtual address space are required (e.g. the pages between the top of the heap and the bottom of the stack)

Virtual memory also allows sharing of read-only pages (e.g. library code) as several processes include the same frame in virtual address space.



Cache Memory

Cache memory is small*, fast memory*, which is close to the CPU.
*small = MB, fast = 5×RAM

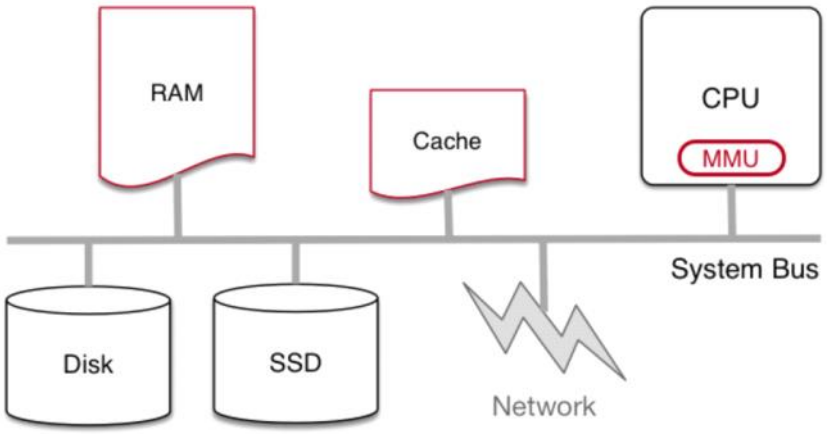


Cache memory holds the parts of RAM that are (hopefully) heavily used. It transfers data to/from RAM in blocks (***cache blocks***). Memory reference hardware first looks in cache memory. If the required address is

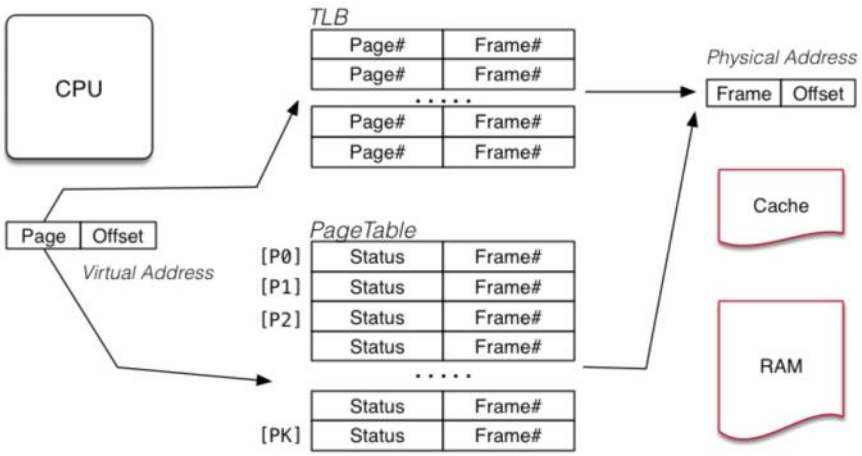
there, it use its contents. If not, it gets it from RAM and puts it in cache, which may possibly replace an existing cache block. The replacement strategies have similar issues to virtual memory.

Memory Management Hardware

Address translation is very important/frequent. There are specialised hardware (MMU) to do it efficiently. This is sometimes located on CPU chip, other times it may be separate from the CPU chip.



A ***translation lookaside buffer*** (TLB) is a lookup table containing (virtual, physical) address pairs.



Process Management

Processes

A ***process*** is an instance of an executing program.
Multiple processes are "active" simultaneously. Multi-tasking is provided on all modern operating systems.
The OS provides each process with:

- ***control-flow independence*** - each process executes as if it is the only process running on the machine
- ***private address space*** - each process has its own address space (`M` bytes, addressed `0..M-1`)

Process management is a critical operating system functionality.

Control-flow independence is where each process has the mindset "*I am the only process, and I run until I finish*". In reality, there are multiple processes running on the machine. Each process uses the CPU until it is pre-empted or exits. Then another processes uses the CPU until it too is pre-empted. Eventually, the first process will get another run on the CPU. This provides the overall impression that the three programs are running simultaneously.

What can cause a process to be pre-empted?

- It runs "long enough" and the OS replaces it by a waiting process.
- It attempts to perform a long-duration task like i/o

On pre-emption, the process's entire dynamic state must be saved (incl PC). The process is flagged as temporarily suspended and it is placed on a process (priority) queue for re-start.
On resuming, the state is restored and the process starts at the saved PC.
This provides the overall impression that the process ran until it finished all of its computation.

Process Management

So how does the OS manage multiple simultaneous processes?
For each process, it maintains context (or state). ***Context*** is the collection of information for one process.

Static information: program code and data
Dynamic state: heap, stack, registers, program counter
OS-supplied state: environment variable, stdin, stdout

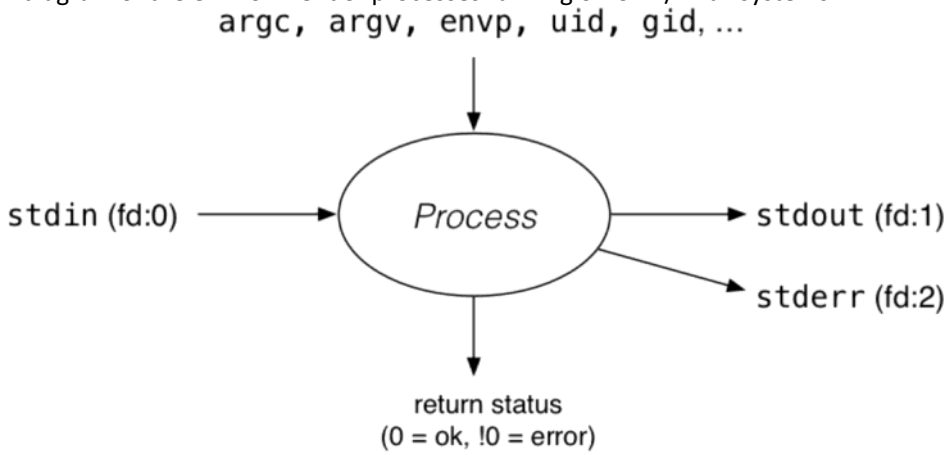
At pre-emption, the OS performs a context switch. A ***context switch*** is when you save the context for one process and restore the context of another process. This is handles by the OS in privileged mode.

A non-static process context is held in a process control block. A ***process control block*** is where the OS records information about each process:

Identifier:	Unique process ID
Status:	Running, ready, suspended, exited. If suspended, the event being waited for.
State:	Registers (including PC)
Privileges	Owner, group
Memory management information:	(reference to) page table
Accounting	CPU time used, amount of I/O done
I/O:	Open file descriptors

The operating system maintains a table of PCBs, one for each currently active process (indexed by process ID). The operating system ***scheduler*** maintains a queue of runnable processes, which are ordered based on information in the PCBs. When the current process is pre-empted or suspends, the scheduler saves the state of process, updates PCB entry and selects the next process to run, and re-starts it.

A diagram of the environment of processes running on Unix/Linux systems



Unix provides a range of tools for manipulating process:

Commands:

- **`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

Information associate with processes (PCB):

- `pid` - process id, unique among current processes
- `ruid`, `euid` - real and effective user id (real user id is the user id that started the process, effective user id is how the system will treat the process, as to who it thinks is running the process)
- `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 signal
- Pointer to process page table
- Process state - running, suspended, asleep, etc.

Every process in Unix/Linux is allocated a process ID. The ID is a positive integer unique among the currently executing processes with type `pid_t` (defined in `<unistd.h>`).
Process 0 is the idle process. Process 1 is `init` (for starting/stopping the system). It is created when the system starts up. If you stop process 1, then your system stops. Low-numbered processes are typically system-related, while regular processes have `PID` in range `300..MaxPid` (e.g. 2^{16}).

Process 0 is not a real process (it's a kernel artefact). It exists to ensure that there is always at least one process to run. On older Unix systems, process 0 was called ***sched*.**

Each process has a parent process, typically the process that creates the current process. A process may have children process, which are any processes that it created.
Process 1 is created at system startup. If a process' parent dies, it is inherited by process 1.

Processes are also collected into ***process groups***. Each group is associated with a unique PGID with type `pid_t`. a child process belongs to the process group of its parent. A process can create its own process group, or it can move to another process group.

Groups allows the:

- OS to keep track of group processes working together,
- distribution of signal to a set of related processes
- management of processes for job control (`ctrl-Z`)
- management of processes within pipelines

System Calls (and Failure)

Reminder:
System calls are requests for the OS to do something. e.g. create a new process, send a signal, read some data etc. Sometimes the request cannot be completed. e.g. invalid PID, or file descriptor, resources exhausted, etc. In such cases the system call returns `-1`, and the value of the global variable `errno` is set. In many/most cases, a failed system call is a fatal error.

How to deal with failed system call?
Generally, we print an error and terminate the process. A useful strategy: a wrapper function with the same arguments/returns as system call catches and reports the error. It only every returns with a valid result. However, this is not always appropriate. e.g. the failure of `open()` is best handled by the caller.

`perror()` also prints an error message based on the value of `errno`, which tells why the function failed.

Example: a wrapper function for `read()`

```
size_t Read(int fd, void *buf, size_t nbytes) {
    ssize_t nread =  read (fd, buf, nbytes);
    if (nread < 0) {
        perror("read() failed");
        exit(i)
    }
    return (size_t)nread;
}
```

`Read()` is used like `read()` but it only gets non-negative returns.

Processes belong to ***process groups***. A signal can be sent to all processes in a process group.

The process information is split across a process table entre and user structure.
The ***process table*** is a kernel data structure describing all processes. It is memory-resident since it is very heavily used and contains information described above for PCB. The content of PCB entry is critical for the scheduler.

A ***user structure*** is kernel data structure describing run-time state. It holds information, which is not needed when the process is swapped out. e.g. the execution state (registers, signal handlers, file descriptors, …).

Process-related System Calls
Unix/Linux system calls:

- **`fork()`** - create a new child process (copy of current process)
- **`_exit()`** - terminate an executing process (without cleaning up)
- **`execve()`** - convert one process into another
- **`getpid()`** - get process ID
- **`getpgid()`** - get process group ID
- **`wait()`** - wait for state change in child process
- **`kill()`** - send a signal to a process

`pid_t fork(void)`
`fork()` requires `#include <unistd.h>`. It creates new process by duplicating the calling process. The new process is the ***child***, calling process is the ***parent***. The child has a different process ID (`pid`) to the parent. In the child, `fork()` returns 0, while in the parent, `fork()` returns the `pid` of the child. If the system call fails, `fork()` returns -1. The child inherits copies of parent's address space and open fd's Typically, the child `pid` is a small increment over the parent `pid`.`fork()` will fail if the OS is unable to create any more processes, or if the user reaches the limit of how many processes they can create.

Minimal example for `fork()`:

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid < 0)
        perror("fork() failed");
    else if (pid == 0)
        printf("I am the parent.\n");
    else
        printf("I am the child.\n");
    return 0;
}
```

We can write a wrapper function for fork(), where if fork() fails, write a message and exit.
Use the function header:

```
pid_t fork() {...}
```

Getting information about a process ...
`pid_t getpid()`
This requires `#include <sys/types.h>` and it returns the process ID of the current process
`pid_t getppid()`
This requires `#include <sys/types.h>` and it returns the parent process ID of the current process
For more details: `man 2 getpid`

Processes belong to *process* groups. A signal can be sent to all processes in a process group.
`pid_t getpgid(pid_t pid)`
`getpgid()` 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)`
`setpgid()` sets the process group ID of a specified process.

Both `getpgid()` and `setpgid()` return -1 and set `errno` on failure.
For more information: `man 2 getpgid`

Minimal example for `getpid()`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
   pid_t pid;
   pid = fork();
   if (pid != 0)
      printf("I am the parent (%d)\n", getpid());
   else
      printf("I am the child (%d)\n", getpid());
   return 0;
}
```

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

This pauses the current process until process `pid` changes state, where state changes include finishing, stopping, re-starting. This ensures that the child resources are released on exit.
Some 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

`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

`pid_t wait(int *status)`

This is equivalent to `waitpid(-1, &status, 0)`
It pauses until one of the child processes terminates.
For more information: `man 2 waitpid`

Minimal example for `wait()`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void)
{
   pid_t pid;
   pid = fork();
   if (pid != 0) {
      wait(NULL);
      printf("I am the parent.\n");
   }
   else
      printf("I am the child.\n");
   return 0;
}
```

`int kill(pid_t` *`ProcID`*`, int` *`SigID`*`)`

`kill()` requires `#include <signal.h>`. It sends signal *`SigID`* to process *`ProcID`* various signals (POSIX) e.g.

- `SIGHUP` - hangup detected on controlling terminal/process
- `SIGINT` - interrupt from keyboard (ctrl-C)
- `SIGKILL` - kill signal (e.g. `kill -9`)

- SIGILL - illegal instruction
- SIGFPE - floating point exception (e.g. divide by zero)
- SIGSEGV - invalid memory reference
- SIGPIPE - broken pipe (no processes reading from pipe)

If it is successful, it returns 0; on error, it returns -1 and set errno

Minimal example for kill():

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
int main(void)
{
   pid_t pid;
   pid = fork();
   if (pid != 0) {
      printf("I am the parent.\n");
      kill(pid, SIGKILL);
   }
   else
      printf("I am the child.\n");
   return 0;
}
```

void _exit(int status)

This terminates the current process and closes any open file descriptors. A SIGCHLD signal is sent to the parent. It returns status to parent (via wait()). Any child processes are inherited by scheduler (pid=1). Termination may be delayed waiting for i/o to complete.
On final exit, process's process table and page table entries are removed

void exit(int status)

This terminates the current process. It triggers any functions registered as atexit(), flushes stdio buffers; closes open FILE *'s then behaves like _exit()

Related function: **void abort(void)**
abort() generates SIGABRT signal (normally terminates process). It closes and flushes stdio streams and is used by the assert() macro.

When a process finishes, it sends SIGCHLD signal to parent.
A ***zombie process*** is a process which has exited but its signal is not handled. All processes become zombie until SIGCHLD is handled. The parent may be delayed (e.g. slow i/o), but usually resolves quickly. A bug in parent that ignores SIGCHLD creates long-term zombies.
Note that zombies occupy a slot in the process table

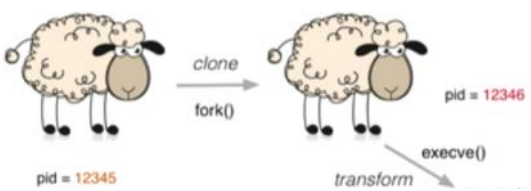
An ***orphan process*** is a process whose parent has exited. When the parent exits, the orphan is assigned pid=1 as its parent. pid=1 always handles SIGCHLD when a process exits.

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

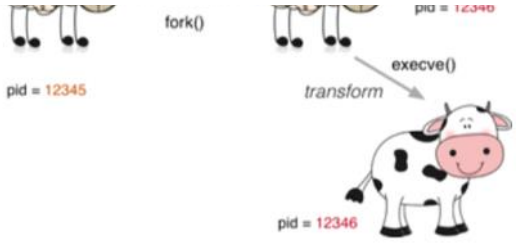
This replaces the current process by executing *Path* object. *Path* must be an executable, binary or script (starting with #!). It passes arrays of strings to new process. Both arrays are terminated by a NULL pointer element. *envp*[] contains strings of the form key=value.
Much of the state of the original process is lost, e.g. new virtual address space is created, signal handlers reset. The new process inherits open file descriptors from original process.
On error, returns -1 and sets errno. If successful, it does not return.

How Unix creates processes:
On Unix, processes create new different processes via:

```
pid_t pid = fork();
if (pid != 0)
   // parent...
   wait(NULL); // wait for child to complete
else {
   // child
```



```
   // parent...
   wait(NULL); // wait for child to complete
else {
   // child...
   char *cmd = "/x/y/z"; // name of executable
   char **args;
   ... // set up command-line argument
   char **env;
   ... // set up environment variables
   execve(cmd, args, env); // child is transformed
}
```



Process Control Flow

When a process is executing:

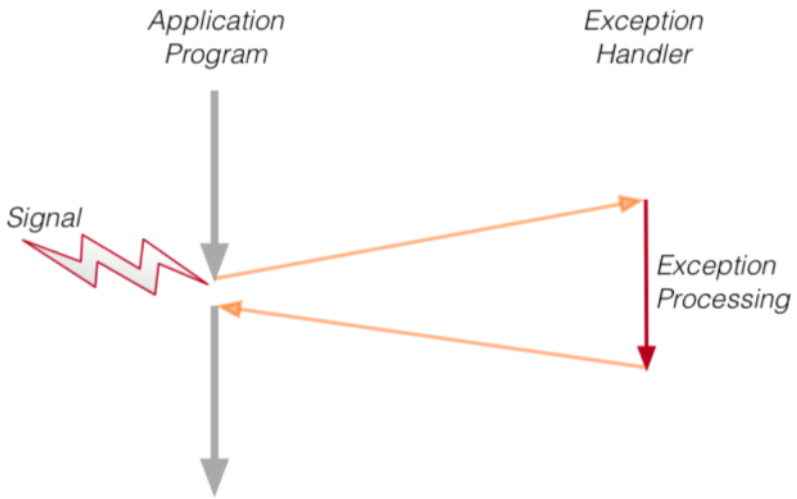
```
fetches instructions from memory[PC]; PC++.
decodes and executes the instruction.
    // if is a jump-type instruction, PC = new address
    // if is a regular instruction, carry out operation
Repeat above.
```

This type of regular control flow is produced by the process. Regular control flow can be interrupted. This is known as ***exceptional control flow***.

Exceptional events are "unexpected" condition occurring during the program execution, which require some form of immediate action (or maybe just quit).
There are two types of exceptional events:

- Exceptions - from conditions within an executing program. These are often fatal to the continued execution of the program.
- Interrupts - from events external to the program. These often require some action and then execution can continue.

Effects of exceptions/interrupts on control flow



System calls typically operate via exceptions (traps). A process makes system call (e.g. `fork()`) and this generates an exception which

- transfers control to system call handler (in privileged mode)
- carries out system-level operations (e.g. modify process table)
- then returns control to the process (in user mode)

The effect is like a normal function call (`jal ... jr`). The critical difference is that we transfer control to the system space (privileged mode).

Signals

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. division by zero)
- From a devices (e.g. disk read completes)

Processes can define how they want to handle signals:

- Using the **`signal()`** library function (a simple method for handling signals)
- Using the **`sigaction()`** system call (a powerful method for handling signals)

Signals from internal process activity, e.g.

- `SIGILL` - illegal instruction (terminate by default)
- `SIGABRT` - generated by `abort()` (dump core by default)
- `SIGFPE` - floating point exception (dump core by default)
- `SIGSEGV` - invalid memory reference (dump core by default)

Signals from external process events, e.g.

- `SIGINT` - interrupt from keyboard (terminate by default)
- `SIGPIPE` - broken pipe (terminate by default)
- `SIGCHLD` - child process stopped or died (ignored by default)
- `SIGTSTP` - stop typed at tty (ctrl-Z) (stop by default)

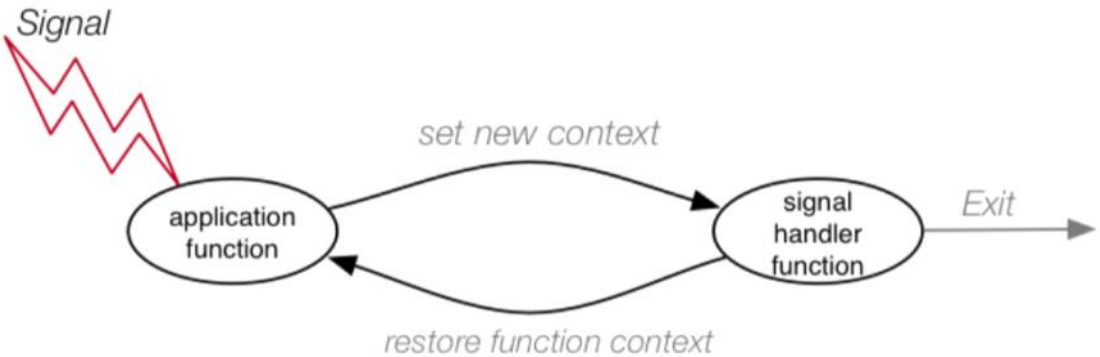
Processes can choose to ignore most signals. If signals are not ignored, they can be handled in several default ways

- Term - terminate the process
- Core - terminate the process, dump core
- Stop - stop the process
- Cont - continue the process if currently stopped

Or you can write your own *signal handler*.
See `man 7 signal` for details of signals and default handling.

Signal Handlers

A ***signal handler*** is a function invoked in response to a signal. It knows which signal it was invoked by and need to ensure that the invoking signal (at least) is blocked. It carries out an appropriate action and may return.



`SigHnd signal(int SigID, SigHnd Handler)`

This defines how to handle a particular signal. It requires `<signal.h>` (library function, not syscall).
`SigID` is one of the OS-defined signals. e.g. `SIGHUP, SIGCHLD, SIGSEGV`, … but not `SIGKILL, SIGSTOP.` A ***handler*** can be one of:

- `SIG_IGN` ... ignore signals of type *`SigID`*
- `SIG_DFL` ... use default handler for *`SigID`*
- a user-defined function to handle *`SigID`* signals

Note: **`typedef void (*SigHnd)(int);`**
It returns previous value of signal handler, or SIG_ERR.

How to define and install a signal handler function:

```
void myHandler(int sigID)
{
   // we don't normally put printf() in a signal handler
   printf("Caught SIGINT\n");
   exit(0);
}
int main(int argc, char **argv)
{
   if (signal(SIGINT, myHandler) == SIG_ERR) {
```

```
      printf("Can't set signal handler\n");
      exit(1);
   }
   sleep(3); // wait for signal
   printf("No signal received\n");
   return 0;
}
```

`int sigaction(int sigID, struct sigaction *newAct, struct sigaction *oldAct)`
`sigID` is one of the OS-defined signals e.g. `SIGHUP, SIGCHLD, SIGSEGV, ...` but not `SIGKILL, SIGSTOP`
`newAct` defines how signal should be handled.
`oldAct` saves a copy of how signal was handled.
if *`newAct`*`.sa_handler == SIG_IGN`, the signal is ignored.
if *`newAct`*`.sa_handler == SIG_DFL`, the default handler is used.
The function returns `0` on success and on error, it returns `-1` and sets `errno`.
For much more information: `man 2 sigaction`

Details on **`struct sigaction`**:

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

Details on **`siginfo_t`**:

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

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

Aside:
The `sleep(N)` functions pauses a process until `N` seconds have passed, or a signal is received by the process. It returns the number of un-slept seconds. i.e. if stopped after `M` seconds, it return `N-M`.

Interrupts

Interrupts are signals which cause normal process execution to be suspended. An ***interrupt handler*** then carries out tasks related to the interrupt and control is then returned to the original process.

Example (input/output):
A process requests data from a disk (this will take 100ms). If the process can do other work not related to the requested data, it will keep executing. When data is fetched from the disk, the process is interrupted. The handler places data in a buffer for access by the process and in-memory computation resumes.

Example (process pre-emption):
A process runs for a while (normal control flow). It receives a timer signal from the OS. The process is suspended and its state is saved. The process is eventually added to a *runnable* queue. When it is removed from the front of the queue and its state is restored, the process continues from where it was suspended.
This is different from the input/output example as the interrupt has no effect on the process state.

Interrupts are frequently associated with input/output. In-memory computations are very fast (ns) while input/output operations are very slow (ms). We can't afford for a process to wait for i/o so it is suspended and placed on a "waiting for i/o" queue. When i/o is complete, it moves the process to a runnable queue. Eventually the process resumes with new data available.

Exceptions

Above exceptions are low-level, system ones.

An alternative notion of ***exceptions*** is that it is an unexpected condition which arises during computation. It is unexpected but not unanticipated.
Examples:

```
if ((p = malloc(sizeof(Type))) == NULL)
   ...
if (scanf("%d", &n) != 1)
   ...
avg = (n != 0) sum/n : 0;
```

Such exceptions require handling in context of computation.
Example:
A function `create()` makes a data structure using multiple `malloc()`s. Part-way through function, one `malloc()` fails. If we abandon `create()`, we need to clean up previous `malloc()`s. If we are terminating the entire process, there is no need to clean up.

Many programming languages have special mechanisms for this

```
try { SomeCode } catch { HandleFailuresInCode }
```

C does not have generic exception handling; roll your own.

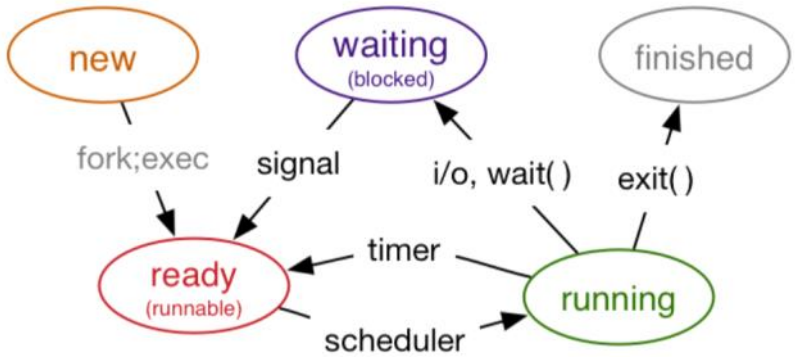
Multi-tasking

Multi-tasking is when multiple processes are "active" at the same time. The process are not necessarily *executing* simultaneously (although this could happen if there are multiple CPUs). It is more likely that we have a mixture of processes; some are *blocked* waiting on a signal (e.g. i/o completion), some are *runnable* (ready to execute), one is running (on each CPU).

The aim is to give the appearance of multiple simultaneous processes by switching processes after one runs for a defined *time slice*. After the timer counts down, the current process is *pre-empted*. A new process is selected to run by the system *scheduler*.

Process States

How the process state changes during execution:



Scheduling

Scheduling is selecting which process should run next. The processes are organised into *priority queue(s)*, where the "highest" priority process is always at the head of the queue. Priority is determined by multiple factors. e.g.

- system processes having higher priority than user processes
- longer-running processes might have lower priority
- memory-intensive processes might have lower priority
- processes can also suggest their own priority

The Linux process scheduler is priority-based (priority queue of processes). Linux process priorities are values in range `-20..139` Lower values represent higher priority. Some factors in determining priority include:

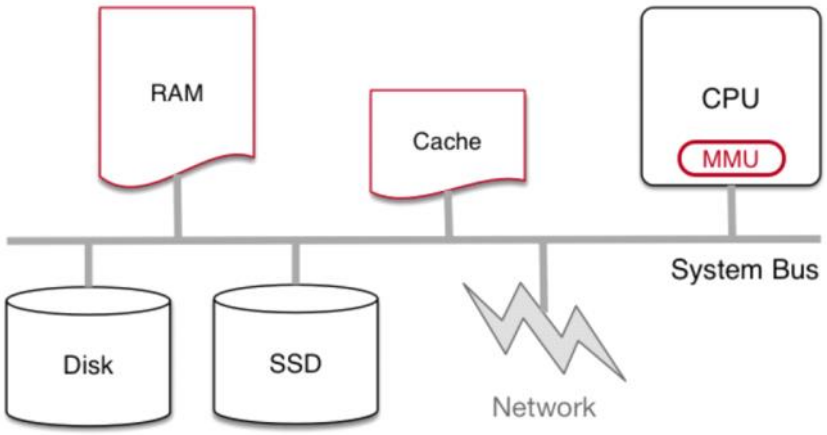
- user processes have lower priority than system processes
- processes have *nice*ness value -20 (highest) .. +20 (lowest)
- recent CPU time usage by the process

The scheduler chooses highest priority process from runnable.

An abstract view of the OS scheduler:

```
onTimerInterrupt()
{
   save state of currently executing process
   newPID = dequeue(runnableProcesses)
   setup state of newPID
   - make process's Page Table active
   - load pages in working set
   - load process's registers
   transfer control to newPID
   - leave kernel mode
   - set PC to saved PC from process
}
```

Device Management



I/O Devices

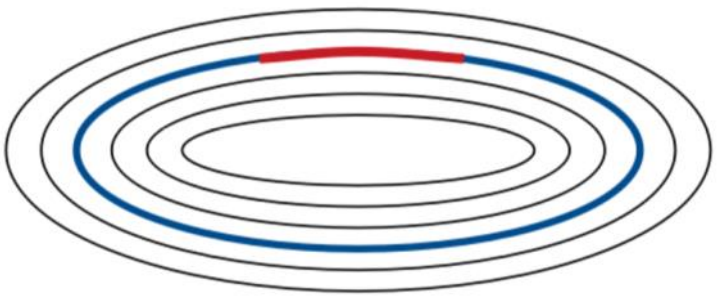
Input/output (I/O) devices allow programs to communicate with "the outside" world. They have significantly different characteristics to memory-based data.

Memory-based data is fast (ns) random access via (virtual) address. It transfers data in units of bytes, halfwords, words.

Device data is much slower (ms) access, random or sequential. It often transfers data in *blocks* (e.g. 128B, 512B, 4KB, ...).

Hard Disk characteristics:

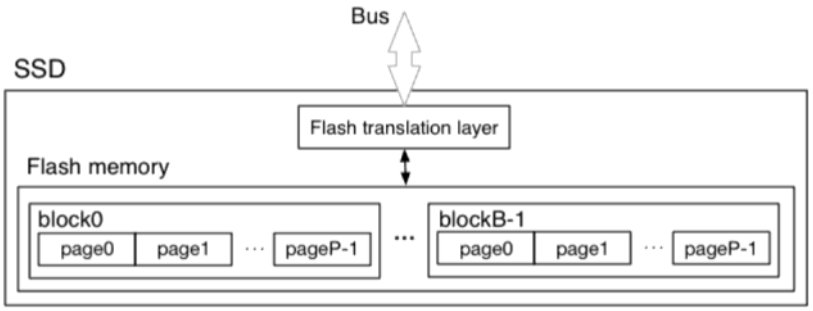
- address specified by track and sector (s/t ≅ 200, s.size 512B)
- access time (move to track + wait for sector + read block)



Typical cost: 10ms seek + 5ms latency + 0.1ms transfer

Solid State Disk (SSD) characteristics:

- high capacity (GB), high cost, reading faster than writing
- pages 512B..4KB, blocks 32..128 pages, R/W page-at-a-time
- pages updated by erase then write, limit on #updates
- avg read time 11×10^{-6} sec, avg write time 15×10^{-6} sec



Example: network transfer

- destination specified by IP address, packet size < 1KB
- transfer time includes
  - d_T transmission delay - time to push data packet onto "the wire"
  - d_P propagation delay - time for packet to travel along "the wire"
  - d_C processing delay - time to check header, re-route to next node
  - d_Q queueing delay - time waiting on node before transmission
  - need to calculate for N hops, so $\cong N(d_T + d_P + d_C + d_Q)$

Typical transfer time: 0.5ms (local ethernet), 200ms (internet), ...
Can check transmission times using the `ping` command

Other types of devices ...

- keyboard - byte-by-byte input, often line-buffered
- screen - pixel-array output, typically via GPU
- mouse - transmit X,Y movement and button presses
- camera - convert video signal, frame-by-frame, to digital stream
- microphone - convert analog audio signal to digital stream



Device Drivers

Each type of device has its own unique access protocol

- special control and data registers
- locations (buffers) for data to be read/written

Device drivers are chunks of code to control an i/o device. They are often written in assembly and are core components of the operating system.
A typical protocol to manipulate devices involves

- sending a request for operation (e.g. read, write, get status)
- receiving an interrupt when request is completed

For more details: see COMP2121 or ELEC2142

Memory-mapped I/O

Operating systems defines special memory addresses. The user programs perform i/o by getting/putting data into memory. Virtual memory addresses are associated with data buffers of i/o devices and control registers of i/o devices.

Some advantages of memory-mapped input/output:

- uses existing memory access logic circuits ⇒ less hardware
- can use full range of CPU operations on device memory
  (cf. having limited set of special instructions to manipulate i/o devices)

Example of using memory-mapped I/O.
Assume memory address of disk device is 0x80000200
write:

```
li  $t0, CodeForWrite
sw  $t0, 0x80000200
li  $t0, LogicalBlockAddress
```

```
sw  $t0, 0x80000200
li  $t0, MemAddressOfData
sw  $t0, 0x80000200
```
Transfers data from memory onto the disk at specified block.
Disclaimer: Above code is not real MIPS/SPIM code. Illustrative only.

Devices on Unix/Linux

Unix treats devices uniformly as byte-streams (like files).
Devices can be accessed via the file system under /dev, e.g.

- **`/dev/diskN`** - (part of) a hard drive
- **`/dev/ttyN`** - a terminal device
- **`/dev/ptyN`** - a pseudo-terminal device

Other interesting "devices" in /dev

- **`/dev/mem`** - the physical memory (mostly protected)
- **`/dev/null`** - data sink or empty source
- **`/dev/random`** - stream of pseudo-random numbers

There are two standard types of "device files":
Character devices (aka character special files) - they provide *unbuffered* direct access to hardware devices. Programmers interact with device by writing individual bytes. They do not necessarily provide byte-by-byte hardware i/o (e.g. disks).
Block devices (aka block special files) - they provide *buffered* access to hardware devices. Programmers interact with device by writing chunks of bytes. Data is transferred to the device via operating system buffers.

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

Devices can be manipulated by open(), read(), write() ...
`int open(char *PathName, int Flags)`
This attempts to open device `PathName` in mode *Flags*
`Flags` can specify caching, async i/o, close on `exec()`, etc.
It returns the file descriptor if ok, -1 (plus `errno`) if error
`ssize_t read(int FDesc, void *Buf, size_t Nbytes)`
This attempts to read `Nbytes` of data into `Buf` from `FDesc`
It returns # bytes actually read, 0 at EOF, -1 (plus `errno`) if error
`ssize_t write(int FDesc, void *Buf, size_t Nbytes)`
This attempts to write `Nbytes` of data from `Buf` to `FDesc`
It returns # bytes actually written, -1 (plus `errno`) if error

Buffered I/O

Devices are typically very slow compared to CPU speed
Using a `read()` from a device for each byte is inefficient
The OS uses a collection of buffers to hold data from devices. Data is supplied to user programs normally from buffer
Assumption: many buffers, holding the current data working set

Note that `read()` accesses data via the OS buffers. It is slow because of the context switch each time it is called.

The standard i/o library (`stdio.h`) provides buffered i/o

- from tty-like devices, generally line buffered
- from disk-like devices (files), read/written in BUFSIZ chunks
    - via OS buffers, so may not need disk access each time

- buffering hidden from user, who sees getchar(), fgets(), etc.
- buffer is allocated by fopen(), removed by fclose()

The FILE object (normally manipulated via a FILE * pointer)

- contains the buffer, malloc'd on fopen()
- contains current position within the buffer and file
- etc. etc.

Concurrency/Parallelism

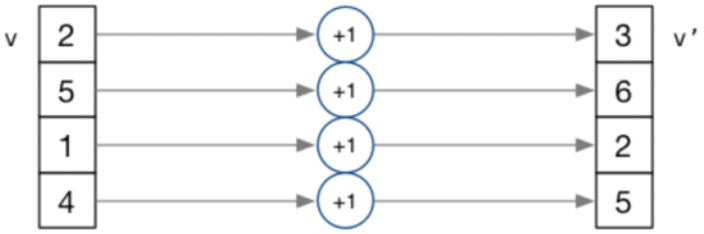
Tuesday, 11 September 2018 10:52 PM

Parallelism is when multiple computations are executed simultaneously.
- e.g. multiple CPUs, one process on each CPU (MIMD)
- e.g. data vector, one processor computes on each element (SIMD)
- e.g. map-reduce: computation spread across multiple hosts

Concurrency is when multiple processes are running (pseudo) simultaneously.
- e.g. single CPU, alternating between processes (time-slicing)

They are primarily concerned with concurrency and ***concurrency control***.
Both parallelism and concurrency need to deal with ***synchronisation***.

An example of SIMD parallel processing (e.g. GPU)
- multiple identical processors
- each processor is given one element of a data structure from main memory
- each processor is performing same computation on that element
- results copied back to main memory data structure



It is not totally independent: need to *synchronise* on completion

Map-reduce is a popular programming model for manipulating very large data sets on a large network of nodes (local or distributed).

The map step filters data and distributes it to nodes. Data is distributed as *(key, value)* pairs. Each node receives a set of pairs with common *key(s)*.
The nodes then perform calculation on the received data items. The ***reduce*** step computes the final result by combining outputs (calculation results) from the nodes.
Note: we also need a way to determine when all calculations are completed.

Creating Concurrency

One method for creating concurrent tasks is `fork()`
`fork()` creates a new (child) process.
The child executes concurrently with the parent. It runs in its own address spaces (which is initially copied from the parent) and it inherits other state information from its parent too (e.g. open file descriptors).

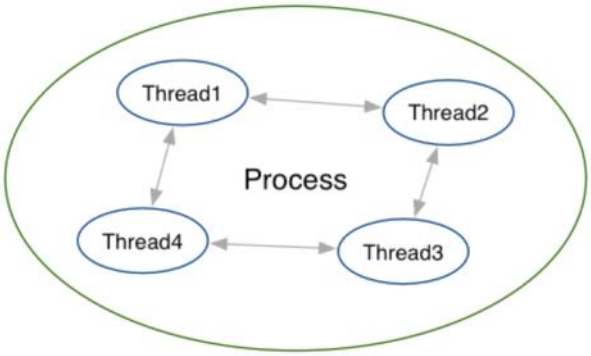
Processes have some disadvantages.
- Process switching requires kernel intervention
- Each has a significant amount of state
- Process interaction requires system-level mechanisms

An alternative method for concurrent tasks is ***threads***.
The difference between threads and processes

Processes	Threads
• are independent of each other • have their own state • each have their own address space • communicate via IPC mechanisms (see later) • context-switching between processes is expensive	• exist within a (parent) process • share parent process state • within a process share one address space • can communicate via shared memory • context-switching between threads is (relatively) cheap

Linux/Unix Threads

pThreads = POSIX implementation of threads
It requires `#include <pthread.h>`
Provides (opaque) `pthread_t` data type
Functions on threads: create, identify, send signals, exit,…




```
int pthread_create(pthread_t *Thread,
                   pthread_attr_t *Attr,
                   void *(*Func)(void *),
                   void *Arg)
```

This creates a new thread with specified **Attr**ibutes.
Thread information is stored in ***Thread**
The thread starts by executing **Func()** with **Arg**
It returns `0` if OK, `-1` otherwise and sets `errno`
In some ways, it is analogous to `fork()`

`pthread_t pthread_self(void)`
`pthred_self()` returns **`pthread_t`** for the current thread
In some ways, it is analogous to `getpid()`

`int pthread_equal(pthread_t t1, pthread_t t2);`
This compares two threads.
It returns a non-zero if same thread, otherwise it returns `0`

`int pthread_join(pthread_t T, void **value_ptr)`
This suspends execution until thread **T** terminates.
`pthread_exit()` value is placed in **`*value_ptr`**
if **T** has already exited, it does not wait

`void pthread_exit(void *value_ptr)`
This terminates the execution of a thread, and also does some cleaning up.
It stores a return value in **`*value_ptr`**

Concurrency

The alternative to concurrency is sequential execution.
Each process runs to completion before another one starts. This has low throughput (the amount of material or items passing through a system/process); so it is not acceptable on multi-user systems.

Concurrency increases system throughput. e.g. if one process is delayed, others can run; if we have multiple CPUs, use all of them at one.

```
0011 1111 1111 1111
3fff
```

If processes are completely independent, each process runs and completes its task without any effect on the computation of other processes. In reality, processes are often not independent. Multiple processes are often accessing a shared resource and one process may be synchronising with another for some computation.

Effects of poorly-controlled concurrency:

- ***nondeterminism*** - same code, which on different runs provides different results
  - e.g. output on shared resource is jumbled
  - e.g. input from shared resource is unpredictable
- ***deadlock*** - when a group of processes end up waiting for each other
- ***starvation*** - when one process keeps missing access to resource

Therefore we need ***concurrency control*** methods.

Example of problematic concurrency - bank withdrawal:

```
// check balance and return amount withdrawn
1. int withdraw(Account acct, int howMuch)
2. {
3.    if (acct.balance < howMuch) {
4.       return 0; // can't withdraw
5.    else {
6.       acct.balance -= howMuch;
7.       return howMuch;
8.    }
9. }
```

Scenario: two processes, one account A, initial balance `$500`

- each process attempts to `withdraw(A, $300)`

Restatement of program:

```
1. int withdraw(Account acct, int howmuch) {
```

```
2.    if (acct.balance < howMuch) return 0;
3.    acct.balance -= howMuch; return howMuch;
4. }
```

Possible outcome of scenario:

- process 1 executes up to line 3, then swapped out
- process 2 executes up to line 3, then swapped out
- process 1 continues and reduces balance by $300
- process 2 continues and reduces balance by $300

Observed: each process gets $300; account balance is -$100
Expected: one process gets $300; other fails; balance is $200

Concurrency Control

Concurrency control aims to

- provide correct sequencing of interactions between processes
- coordinate semantically-valid access to shared resources

There are two broad classes of concurrency control schemes:

- ***shared memory*** based (e.g. semaphores)
- ***message passing*** based (e.g. send/receive)

Both schemes require programming support available via special library functions, or new language constructs.

Shared memory approach:

- Uses shared variable, manipulated ***atomically*** (guaranteed isolation from interrupts, signals, concurrent processes and threads)
- Blacks if access unavailable, decrements once available

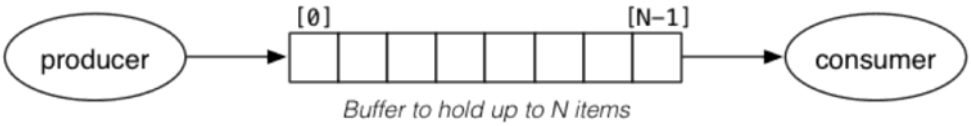
Message passing approach:

- Processes communicate by sending/receiving messages
- The receiver can block waiting for message to arrive
- Sender *may* block waiting for message to be received
    - Synchronous message passing: sender waits for ACK or receipt
    - Asynchronous message passing: sender transmits and continues

Producer-Consumer Problem

This is a classic example for concurrency control issues.
We have:

- A buffer with slots for N items
- A process that produces new items and puts them in the buffer
- A process that consumes items form the buffer
- A mechanism for a process to pause itself
- A mechanism for signalling a process to wake up



Data objects shared by two processes:

```
#define N ??
Item buffer[N];  // buffer with slots for N items
int  nItems = 0; // #items currently in buffer
int  head = 0, tail = 0;
```

Functions on buffer (simplified view):

```
putItemIntoBuffer(item) {
   tail = (tail+1) % N;
   buffer[tail] = item;
}
```

```
Item getitemFromBuffer() {
   Item res = buffer[head];
   head = (head + 1) % N;
   return res;
}
```

Producer process (P):

```
producer()
{
   Item item;
   for (;;) {
      item = produceItem();
      // wait if buffer is currently full
      if (nItems == N) pause();
      putItemIntoBuffer(item);
      nItems++;
      // tell consumer item now available
      if (nItems == 1) wakeup(consumer);
   }
}
```

Consumer process (C):

```
consumer()
{
   Item item;
```

```
   for (;;) {
      // wait if nothing to consume
      if (nItems == 0) pause();
      item = getItemFromBuffer()
      nItems--;
      // free slot available in buffer
      // wake up produce in case sleeping
      if (nItems == N-1) wakeup(producer);
      consumeItem(item);
   }
}
```

A possible scenario (assumes signals only reach paused processes)

- C checks `nItems`, finds `zero`, decides to pause
- just before pausing, C is timed-out (different to paused)
- P creates item, puts it in currently empty buffer
- because buffer now has one item, P signals C
- because C is not paused, signal is lost
- C is resumed after time-out and pauses
- P resumes and adds more items
- eventually buffer fills and P pauses
- each process is paused, waiting for signal from the other

This situation is *deadlock* ... How to fix?

Semaphores

Semaphore operations:

- **`init(`*`Sem, InitValue`*`)`**
  set the initial value of semaphore *`Sem`*
- **`wait(`*`Sem`*`)`**
  if current value of *Sem* > 0, decrement *Sem* and continue
  otherwise, block and wait until *Sem* > 0, then decrement
- **`signal(`*`Sem`*`)`**
  increment value of *Sem*, and continue

Note: all of them happen *atomically*

Needs fair release of blocked processes, otherwise ⇒ starvation

This can be achieved via a FIFO queue (which is fair, but maybe not optimal)

Using semaphores for the produce-consumer problem:

- Semaphores are updated atomically
- So they can't time out `if (nItems == 0) pause()`

```
semaphore nFilled;  init(nFilled, 0); // no. of filled slots
semaphore nEmpty;   init(nEmpty, N);  // no. of empty slots
semaphore mutex;    init(mutex, 1);
```

The `mutex` semaphore ensures that

- only one process at a time manipulates the buffer
- allows multiple producers/consumers to interact correctly

Produce process:

```
producer() // process
{
   Item item;
   for (;;) {
      item = produceItem();
      wait(nEmpty);   // pause if buffer full
      wait(mutex);    // get exclusive access
      putItemIntoBuffer(item);
      signal(mutex);  // release exclusive access
      signal(nFilled);
   }
}
```

Consumer process:

```
consumer() // process
{
   Item item;
   for (;;) {
      wait(nFilled);  // pause if buffer empty
      wait(mutex);    // get exclusive access
      item = getItemFromBuffer()
      signal(mutex);  // release exclusive access
      signal(nEmpty);
      consumeItem(item);
   }
}
```

Semaphores on Linux/Unix ...

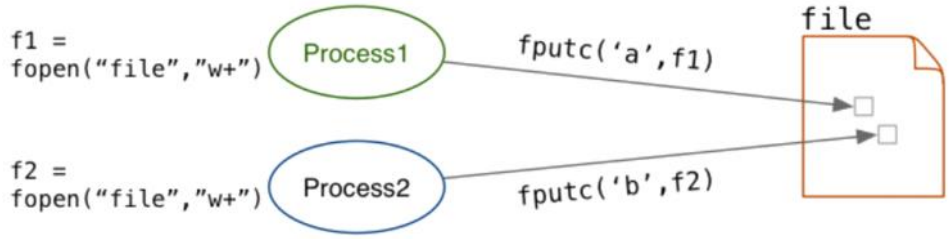
- **`#include <semaphore.h>`**, giving **`sem_t`**
- **`int sem_init(sem_t *`*`Sem`*`, int `*`Shared`*`, uint `*`Value`*`)`**
  create a semaphore object, and set initial value
- **`int sem_wait(sem_t *Sem)`** (i.e. `wait()`)
  try to decrement; block if *`Sem`* `== 0`

- **`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`*

Interacting Processes
Process can interact via
- signals: `kill()`, `wait()`, signal handlers
- accessing the same resource (e.g. writing onto the same file)
- pipes: stdout of process A goes into stdin of process B
- message queues: passing data between each other
- sockets: client-server style interaction

Uncontrolled interaction is a problem: nondeterministic

An example of problematic process interaction:
We have two processes writing to the same file "simultaneously". The order of output depends on the actions of an (opaque) scheduler.



We could control the access to the file via semaphores.
An alternative (less general) mechanism is ***file locking***.

File Locking

`int flock(int FileDesc, int Operation)`
This controls access to shared files (**note**: files not file descriptors)
Some 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 there is a lock available.
It 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 the file is not locked, or has other shared locks, it is OK
- If the file has an exclusive lock, it is blocked

If a process tries to acquire and ***exclusive lock***:
- If the file is not locked, it is OK
- If there are any locks (shared or exclusive) on the file, it is blocked

If using a non-blocking lock:
- `flock()` returns `0` if the lock was acquired
- `flock()` returns `-1` if the process would've been blocked

Pipes

Pipes are a common style of process interaction (communication).
The producer process writes to a byte stream (cf. stdout) and the consumer process reads from the same byte stream.

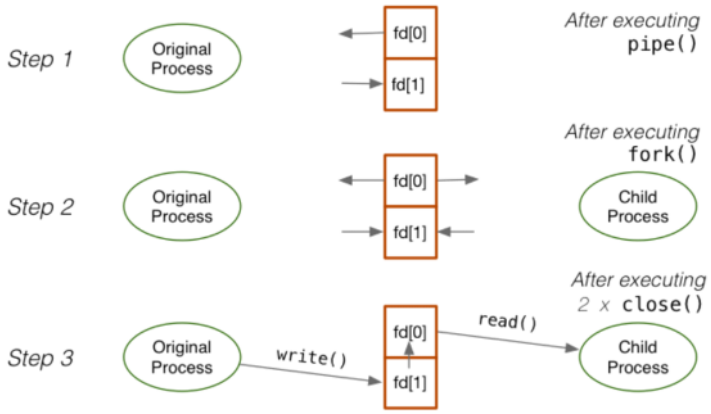
A ***pipe*** provides buffered i/o between the producer and consumer.
The producer blocks when the buffer is full; the consumer blocks when the buffer is empty.

Pipes are bidirectional unless processes close one file descriptor.

`int pipe(int fd[2])`
`pipe()` opens two file descriptors (to be shared by processes)
`fd[0]` is opened for reading; `fd[1]` is opened for writing
It returns `0` if OK, otherwise return `-1` and sets `errno`

Creating the pipe would then be followed by
- `fork()` to create a child process
- both processes have copies of `fd[]`
- one can write to `fd[1]`, the other can read from `fd[0]`

Creating a pipe



Example: setting up a pipe

```
int main(void)
{
   int fd[2], pid;  char buffer[10];
   assert(pipe(fd) == 0);
   pid = fork();
   assert(pid >= 0);
   if (pid != 0) {
      close(fd[0]);
      write(fd[1], "123456789", 10);
   }
   else {
      close(fd[1]);
      read(fd[0], buffer, 10);
      printf("got \"%s\"\n", buffer);
   }
   return 0;
}
```

A common pattern in pipe usage is that:

- You set up a pipe between parent and child
- `exec()` child to become the new process talking to the parent

Because this is so common, there is a library function available for it.

`FILE *popen(char *Cmd, char *Mode)`

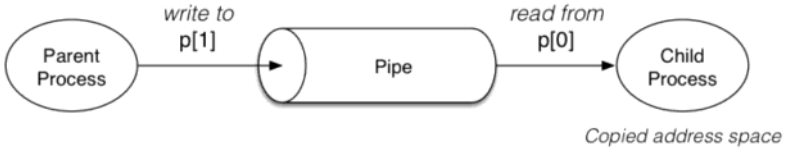
This is analogous to fopen, except first arg is a command
Cmd is passed to shell for interpretation
It returns `FILE*` which be read/written depending on *Mode*
It returns `NULL` if it can't establish a pipe or *Cmd* is invalid

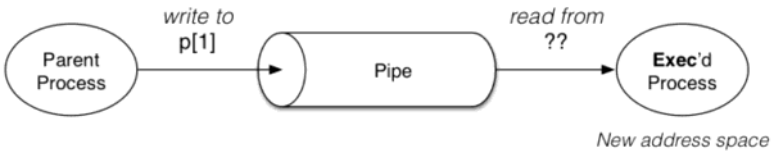
Example of `popen()`

```
int main(void)
{
    FILE *p = popen("ls -l", "r");
    assert(p != NULL);
    char line[200], a[20],b[20],c[20],d[20];
    long int tot = 0, size;
    while (fgets(line,199,p) != NULL) {
        sscanf(line, "%s %s %s %s %ld",
                       a, b, c, d, &size);
        fputs(line, stdout);
        tot += size;
    }
    printf("Total: %ld\n", tot);
}
```

Our first example of piping works because both parent and child process share an address space. They can both access `p[]` and they both have the same set of file descriptors. When we `execve()` in the child, the transformed process has its own address space. This address space does not contain `p[]`. It is possible for the transformed child to access the pipe since `popen()` reads/write to stdout/stdin despite being in a different address space.

The two scenarios are:





So how do we get a pipe to connect to two different processes, where the processes do not share an address space?
Use the stdin and stdout file descriptors! Both processes have *stdin* (fd:0) and *stdout* (fd:1) and the fd's are retained even across an `execve()`.

If we want the parent to send data to an exec'd child:

- Connect `stdout` and `p[1]` in the parent process
- Connect `stdin` and `p[0]` in the child process

If we want an exec'd child to send data to a parent:

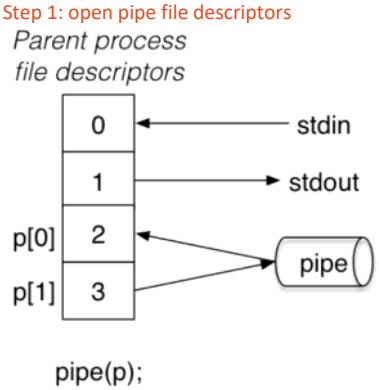
- Connect `stdin` and `p[0]` in the parent process
- Connect `stdout` and `p[1]` in the child process

The connection is accomplished via the `dup2()` function.
`dup2(fd1, fd2)` copies file descriptor `fd1` onto `fd2`. recall that file descriptors are just indexes into a table of structures. If `fd2` was already open, it is closed before the copy.

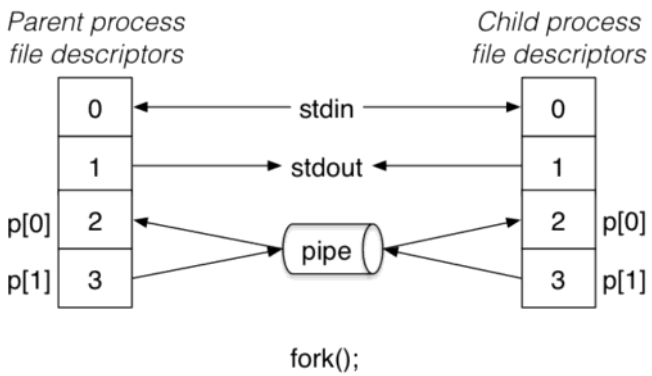
Note: there is a related function called `dup()`. **`dup(fd1)`** copies `fd1` onto the "next free" fd; next free being the lowest numbered unused file descriptor.
Note: `dup2(fd1, 0)` is like `close(0); dup(fd1);`

Pipe between two `exec`'d processes (adapted from University of Illinois CS241)
Scenario: Child sending data to parent

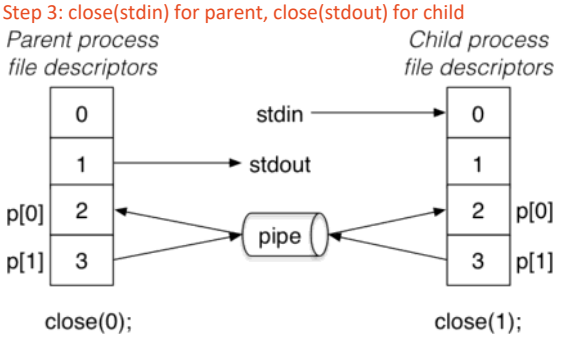
Step 1: open pipe file descriptors



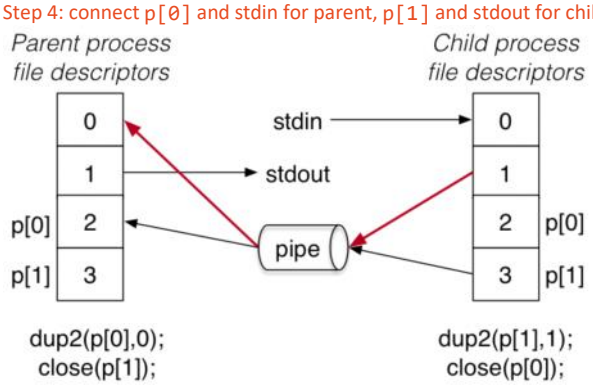
Step 2: fork your process



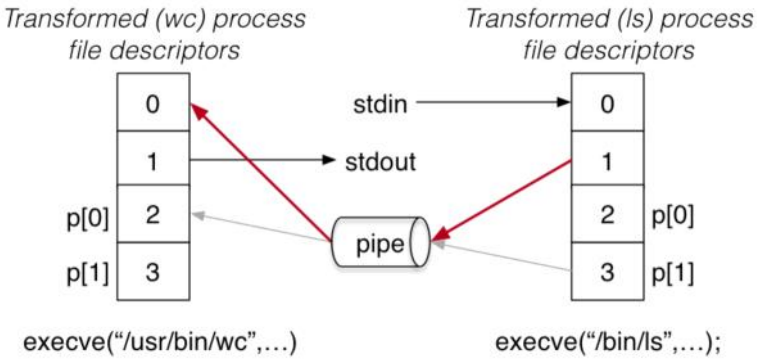
Step 3: close(stdin) for parent, close(stdout) for child



Step 4: connect `p[0]` and stdin for parent, `p[1]` and stdout for child



Step 5:

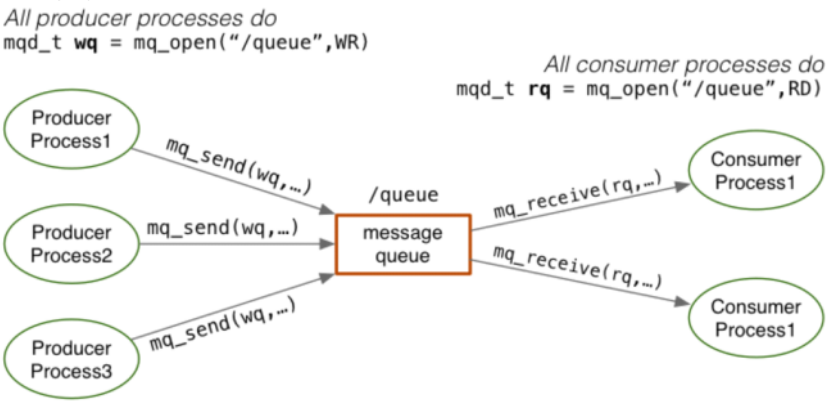


Message Queues

Pipes operate between two processes on the same host.
Processes initially come from parent/child pairs (via `fork`). The connection is established via *shared file descriptors*.

A ***message queue*** (MQ) provides a mechanism for unrelated processes to pass information along a buffered channel shared by many processes.
Processes connect to message queues by name. Message queue names look like "`/SomeCharacters`" (no slash chars)

Message queue architecture



Requires `#include <mqueue.h>`, giving `mqd_t`
`mqd_t mq_open(char *Name, int Flags)`
This create a new message queue, or opens an existing one
`Flags` are like those for `open()` (e.g. O_RDONLY)

`int mq_close(mqd_t *MQ)`
This finishes accessing message queue *MQ*, but the message queue continues to exist (cf. fclose())

For more details `man 7 mq_overview`

`int mq_send(mqd_t MQ, char *Msg, int Size, uint Prio)`
This adds the message *Msg* to message queue *MQ*
Prio gives the priority of the message (which determines the order of messages on MQ)
If *MQ* is full ...

- blocks until MQ space available
- if O_NONBLOCK is set, the `mq_send()` fails and returns error

also, `mq_timedsend()` which

- waits for specified time if MQ full
- it fails if still no space on MQ after timeout

`int mq_receive(mqd_t MQ, char *Msg, int Size, uint *Prio)`
This removes the highest priority message from queue *MQ*
If **Prio* is not NULL, receives message priority
If *MQ* is empty ...

- blocks until a message is added to *MQ*
- if O_NONBLOCK is set, fails and returns error

If several processes blocked on `mq_receive()` the oldest and highest priority process receives the message.

Pseudo-code showing structure of a MQ server program:

```
main() {
   // set up message queue
   attr = (0,#msgs,MsgSize)
   mode = O_CREAT|O_RDWR
   mq = mq_open("/queue", mode, Perms, &attr)
   while (1) {
      // ask for request ... wait
      mq_receive(mq, InBuf, Size, NULL)
      ... Determine response ... put in OutBuf
      mq_send(mq, OutBuf, strlen(OutBuf), 0)
```

```
   }
}
```

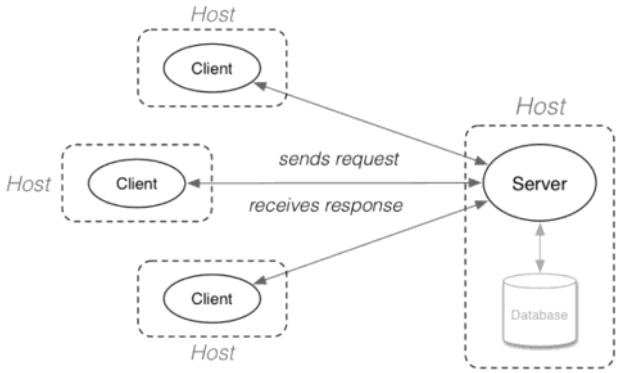
Pseudo-code showing structure of a MQ client program:

```
main() {
   // set up message queue
   attr = (0,#msgs,MsgSize)
   mode = O_RDWR
   mq = mq_open("/queue", mode, Perms, &attr)
   // read request from user
   while (fgets(Request, Size, stdin) {
      mq_send(mq, Request, strlen(OutBuf), 0)
      mq_receive(mq, InBuf, Size, NULL)
      ... get response ... act accordingly ...
   }
}
```

Networked IPC

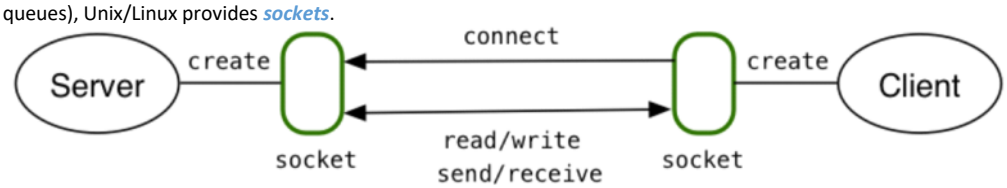
Inter-process communication (IPC) mechanisms considered so far assume that both processes are on the same host.
How do we implement systems on the network, where the processes exist on different machines?

A popular network-based software architecture:



Sockets

To implement systems that work across the network (e.g. web servers, networked databases, networked message queues), Unix/Linux provides ***sockets***.



Sockets are an end-point of an IPC channel.
They are commonly used to construct client-server systems either locally (Unix domain) or network wide (Internet domain).
The server creates a socket,

- then binds it to an address (local or network)
- and listens for connections from clients.

The client creates a socket,

- then connects to the server using a known address,
- writes to the server via a socket (i.e. sends requests),
- and reads from the server via the socket (i.e. receives responses)

`int socket(int Domain, int Type, int Protocol)`
This requires **`#include <sys/socket.h>`**
It creates a socket, using:
`Domain` - the communications domain

- `AF_LOCAL` - on the local host (Unix domain)
- `AF_INET` - over the network (Internet domain)

`Type` - the semantics of communication

- `SOCK_STREAM` - sequenced, reliable communications stream
- `SOCK_DGRAM` - connectionless, unreliable packet transfer

`Protocol` - communication protocol

- there are many protocols, which exist (see /etc/protocols), e.g. IP, TCP, UDP, ...

It returns a socket descriptor (small int) or `-1` on error

`int bind(int Sockfd, SockAddr *Addr, socklen_t AddrLen)`
This associates an open socket with an address.
For Unix Domain, the address is a pathname in the file system
For Internet Domain, the address is IP address + port number

`int listen(int Sockfd, int Backlog)`
This wait for connections on socket ***`Sockfd`***.

It allows at most *BackLog* connections to be queued up.

SockAddr = struct sockaddr_in

- **sin_family** - domain: AF_UNIX or AF_INET
- **sin_port** - port number: 80, 443, etc.
- **sin_addr** - structure containing host address
- **sin_zero[8]** - padding

Example:

```
struct sockaddr_in web_server;
server = gethostbyname("www.cse.unsw.edu.au");
web_server.sin_family = AF_INET;
web_server.sin_addr.s_addr = server;
web_server.sin_port = htons(80);
```

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)

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

Pseudo-code showing structure of a simple client program:

```
main() {
  s = socket(Domain, Type, Protocol)
  serverAddr = {Family,HostName,Port}
  connect(s, &serverAddr, Size)
  write(s, Message, MsgLength)
  read(s, Response, MaxLength)
  close(s)
}
```

(See http://www.linuxhowtos.org/C_C++/socket.htm)

Pseudo-code showing structure of a server program:

```
main() {
   s = socket(Domain, Type, Protocol)
   serverAddr = {Family,HostName,Port}
   bind(s, serverAddr, Size)
   listen(s, QueueLen)
   while (1) {
      int ss = accept(s, &clientAddr, &Size)
      if (fork() != 0)
         close(ss)
      else {

         close(s)
         handleConnection(ss)
         exit(0)
      }
   }
}
```

Network Architecture

Tuesday, 18 September 2018 5:11 PM

Networks

Networks are interconnected collections of computers.
Types of networks:

- ***Local area networks*** (LAN) - within an organisation/physical location
- ***Wide area network*** (WAN) - geographically dispersed
- ***Internet*** - a global set of interconnected WANs

Why do we need networks?
Previously we needed networks to transfer data and send text-based emails. Nowadays we networks for communication and sharing resources (e.g. printers, large storage devices)

What are the basic requirements for a network?
To get data from machine A to machine B, where A And B may be separated by 100's of networks and devices.

How do we achieve this (using a postal service analogy)?
We need:

- A unique address for destination
- To identify a route (the first post office)
- To process at intermediate nodes (other post offices)
- To follow certain protocols (envelopes, stamp fees)

Overview of Network Communication

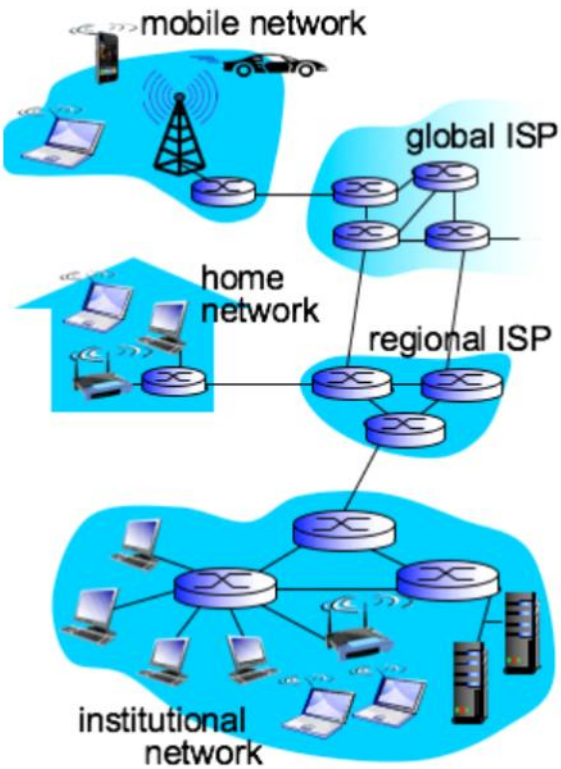
How a file is sent over the network:

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

The Internet

Components of the Internet

- Millions of ***connected devices***
  - e.g. PC, server, laptop, smartphone
  - host is the end system, running network apps
- ***Communication links***
  - e.g. fibre, copper, radio, satellite
  - bandwidth is the rate of transmission
- ***Packet switches***
  - e.g. routers, network switches
  - compute the next hop and forward packets



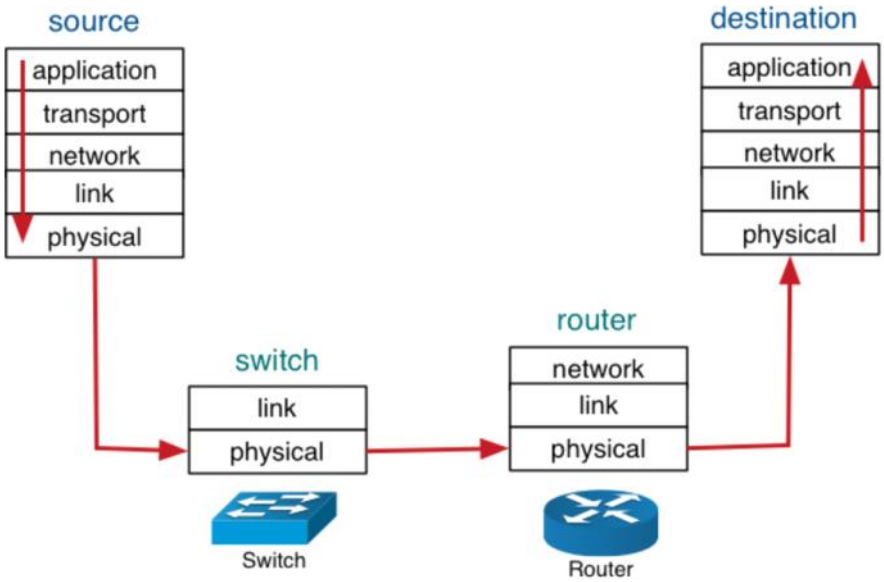
Internet communication are based on a 5-layer "stack":

- ***Physical layer:*** bits on (wires or fibre optics or radio)
- ***Link layer:*** ethernet, MAC addressing, CSMA etc.
- ***Network layer:*** routing protocols, IP
- ***Transport layer:*** process-process data transfer, TCP/UDP
- ***Application layer:*** DNS, HTTP, email, Skype, torrents, FTP etc.

A typical packet encapsulates data from all lower layers.

Each layer encapsulates on aspect of network transport. It provides a layered ***reference model*** for discussion. Separating network transportation in to layers eases maintenance and updating as changing the implementation of one layer does not affect other layers.

Path of data through network layers:



Protocols

Network protocols govern all communication activity on the network.
Protocols provide ***communication rules***; a format and order of messages sent/received, the actions taken on message transmission/retrieval.

Protocols are defined in all of the layers, e.g.

- Link layer: PPP (point-to-point protocol)
- Network layer: IP (internet protocol)
- Transport layer: TCP (transmission control), UDP (user datagram)
- Application layer: HTTP, FTP, SSH, POP, SMTP

Higher-level layers typically have a wider variety of protocols.

Network Application Layer

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

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking

Client-Server Architecture

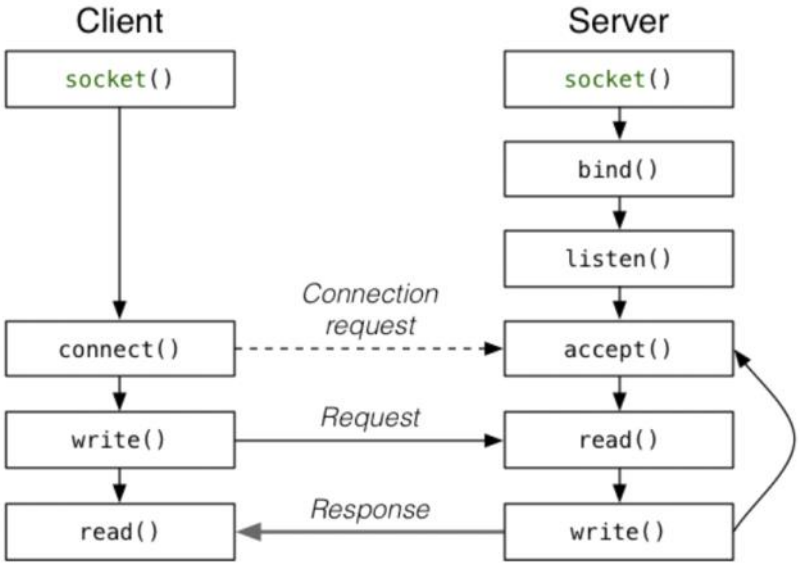
Client-server architecture is a common way of structuring network communication.

The server is a data provider. It is a process that waits for a requests. It is always-on host, with a permanent IP address and is possibly using data centres/multiple CPUs for scaling.

The client is a data consumer. It sends requests to the sever and collects a response. It may be intermittently connected and may have a dynamic IP address. It does not communicate directly with other clients.

Peer-to-peer (P2P) systems run both client and server processes on each host.

Client-server systems are frequently implemented with sockets.



Addressing

Server processes must have a unique Internet-wide *address*. Part of the address is the *IP address* of the host machine, the other part is the *port number*, where the server listens.
Example: `128.119.245.12:80` is the address of web server on gaia.cs.umass.edu

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

IP Addresses are unique identifiers for a host on the network. It is given as a 32-bit identifier (dotted quad), e.g. `129.94.242.20`.
A special case is `127.0.0.1` , which is a loopback address referring to the ***local host***.
IP addresses are assigned by the system administrator entering into local registry (for "permanent" addresses). They are assigned dynamically, by getting a temporary address from DHCP server.

Note: the world is running out of 32-bit IP addresses.
Why? Internet of Things - *every* networked device needs an IP address.
IPv6 uses 128-bit addresses e.g. `2001:388:c:4193:129:94:242:20`
To compare:
IPv4 has 4×10^9 distinct addresses, while IPv6 has 3×10^{38} distinct addresses.

Application-layer protocols

Each application-layer protocol defines:

- ***Types*** of messages - different types of requests and responses
- Message ***syntax*** - what fields are in messages how fields are delineated
- Message ***semantics*** - meaning of information in fields
- Processing ***rules*** - when and how processes respond to messages

Protocols can be open (e.g. HTTP) or proprietary (e.g. Skype).

The HTTP Protocol

HTTP (HyperText Transfer Protocol) is an extremely important protocol (as it drives the Web).
Message types: URLs (requests) and Web pages (responses)
Message syntax: headers + data (see details later).

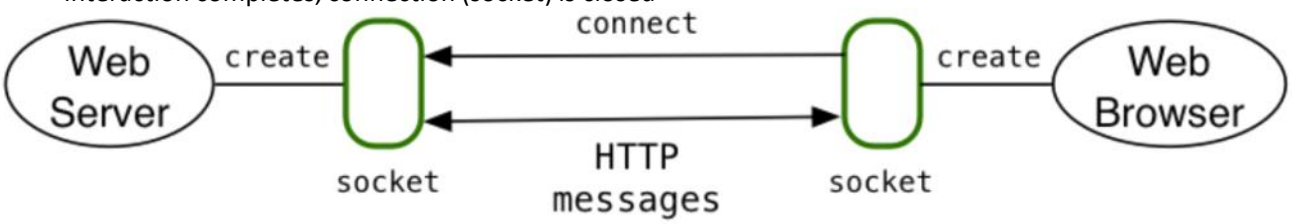
URLs are the primary type of request.

Web pages are the primary type of response. They contain HTML an may contain references to other types of objects. All web objects are addressable via a URL.

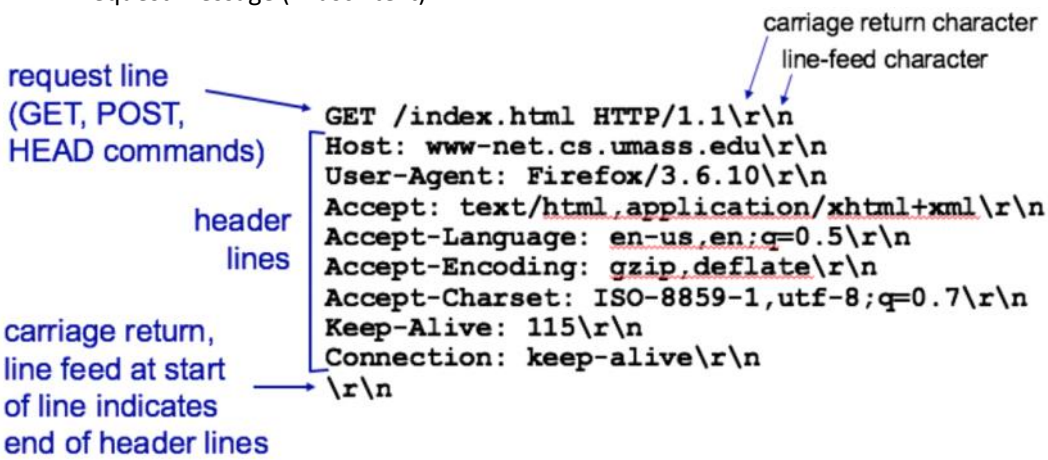
HTTP is an application-layer protocol for the web. In the client-server model:
The client is the ***web browser***. It sends HTTP requests,
receives HTTP responses and shows the response as a webpage.
The server is the ***web server***. It receives HTTP requests and sends HTTP responses.

Transport layer view of HTTP application later:
Using TCP,

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



A HTTP request message (in ascii text)

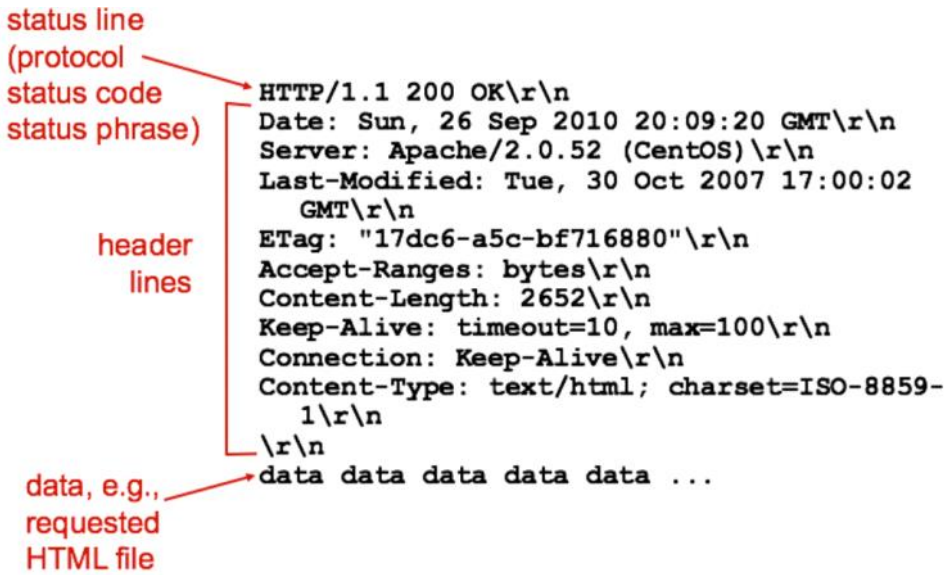


A URL can also include a ***query string***:

The first line of a HTTP request contains (a method, path, protocol) e.g.

There is no need to mention the host, since connection is already established.
GET requests data from resource specified by a path. A query string is included in the path.
POST submits data to be processed by a specified resource. The query string is included in the body.
HEAD is the same as get, but it only returns the header (not data).

A HTTP response message (in ascii text):



Response status codes in the first line of a HTTP response

- **`202 OK`** - successful request
- **`301 Moved Permanently`** - requested object moved. It returns the new URL for client to use in future requests
- **`400 Bad Request`** - request cannot be processed. Possible reasons: bad request syntax, request size too large, ...
- **`403 Forbidden`** - valid request cannot be processed. Possible reasons: user does not have permission for operation
- **`404 Not Found`** - path does not exist on server
- **`500 Internal Server Error`** - server cannot complete request. Possible reasons: server side script fails, database not accessible, ...

Server Addresses (DNS)

Network requests typically use server ***names***. e.g `http://www.cse.unsw.edu.au/~cs1521/17s2/`

Setting up a TCP connection needs an IP address, not a name.
The ***Domain Name System*** provides name to IP address mapping. It can access this on Unix/Linux via the `host` command. e.g.

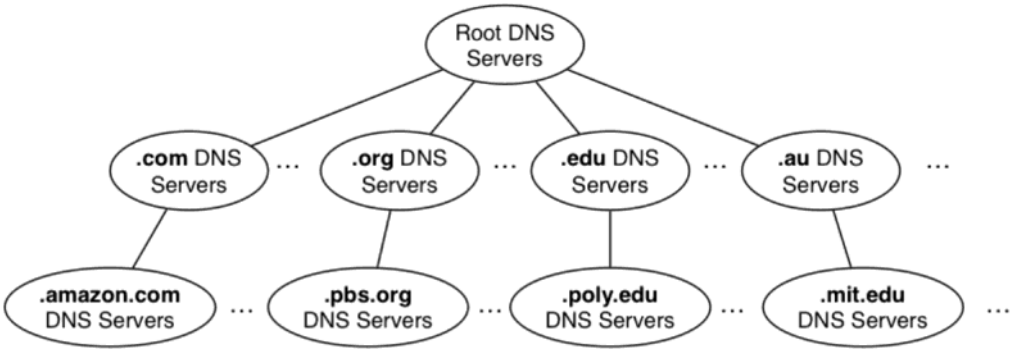
```
$ host www.cse.unsw.edu.au
www.cse.unsw.edu.au has address 129.94.242.51
$ host a.b.c.com
Host a.b.c.com not found: 3(NXDOMAIN)
```

Note: this assumes that you have a network connection

In real life, we often have one object referenced by many names. e.g. a person can be referenced by name, SSN, TFN, passport.
On the Internet, each ***host*** has one or more symbolic names and a unique IP address.
e.g. symbolic: `www.cse.unsw.edu.au`, IP: `129.94.242.51`

Note: a given IP address may be reachable via several names, and a given name may map to several IP addresses (e.g. for load distribution).

The ***Domain Name System*** (DNS) is effectively a distributed database of name to IP address mappings. It is implemented across a hierarchy of ***name servers***. Name servers cooperate to resolve names to IP addresses. This is an extremely important core function on the Internet. We don't have a centralised DNS because it becomes a central point of failure and increases traffic volume. A distant database can result in lag and maintenance of a large database require lots of effort.



Example: finding the IP address of `www.amazon.com` in the system above:

1. Contact a root DNS sever to find the `.com` DNS server
2. Contact the `.com` DNS server to get the `amazon.com` DNS server
3. Contact the `amazon.com` DNS server to the IP address of their web server.

There are two types of name resolution.

- ***Iterated query*** - work is done by the client.
    1. client contact a name server X.
    2. gets a response:
        i. "I don't know, but ask name server Y",
        ii. or "Here is the IP address"
    3. repeats the above steps until name is resolved.
- ***Recursive query*** - work is done by the name servers.
    1. Client contacts server name X
    2. X contact server Y, Y contacts Z, etc.
    3. Query propagates until name is resolved.

How are the various DNS servers structured/managed?

- ***Top-level domain*** (TLD) name servers - `.com`, `.org`, `.edu` and all country-level domains (e.g. `.uk`)
    - Network solutions maintain servers for `.com`
    - AusResgistry maintains servers for `.au`
- ***Authoritative*** name servers - maintains mappings from names to IP addresses within an organisation. All hosts within the organisation are registered here.
- ***Local*** (default) name servers - maintains a cache of name to IP address mappings. It is the starting point for DNS queries. If the query is not cached it is forwarded to TLD server.

Network Architecture

Recall our five layer model:

- ***physical*** *layer* ... bits on wires <- lowest (least abstract) level
- ***link*** *layer* ... e.g. ethernet, MAC
- ***network*** *layer* ... e.g. IP
- ***transport*** *layer* ... e.g. TCP/UDP
- ***application*** *layer* ... e.g. HTTP, email <- highest (most abstract) level

Packets at each level incorporate headers from lower levels.
Software at each level uses headers appropriate for its purpose

Transport Layer

The transport layer deals with:

- ***data integrity***
    - some apps (e.g. file transfer) require 100% reliable transfer
    - other apps (e.g. audio streaming) can tolerate some loss
- ***timing***
    - some apps (e.g. networked games) require low transmission delay
- ***throughput***
    - some apps (e.g. multimedia) require minimum throughput
    - other apps ("elastic apps") can use whatever is available
- ***security***
    - some apps (e.g. web services) require encrypted transmission

Properties of some common apps ...

- file transfer: no loss, elastic, not time sensitive
- email: no loss, elastic, not time sensitive
- web/http: no loss, elastic, not time sensitive
- audio: loss-tolerant, 5Kbps-1Mbps, few ms delay ok
- video: loss-tolerant, 10Kbps-5Mbps, few ms delay ok
- games: loss-tolerant, 5Kbps-5Mbps, few secs delay ok
- texting: no loss, elastic, few ms delay ok

Transport layer protocols provide:

- Logical communication between processes on different hosts
- Transport protocols run within end-point processes
    - Sender: splits messages into segments, passes segments to network layer
    - Receiver: reassembles segments into messages, passes to application layer

There are two main transport layer protocols on the Internet:

- TCP - reliable, connection-oriented protocol, byte-stream
- UDP - unreliable, simple, connectionless protocol, segments

TCP (Transmission Control Protocol) provides:

- ***Reliable transport***: data flow between sender/receiver
- ***Flow control***: making sure the sender does not overwhelm the receiver
- ***Congestion control***: slow down the send if network is congested
- ***Connection-oriented***: setup required between client/server

It does not provide: timing/throughput guarantees, security

UDP (User Datagram Protocol) provides fast (for sender), unreliable data transfer.
It does not provide: reliability, flow control, timing/throughput guarantees, security

TCP is typically layered on top of IP protocol. ***IP*** is an unreliable network layer protocol. TCP provides a reliable stream of data on top of IP.

How TCP works:

1. Set up a connection between sender and receiver
2. Sender transmit a pipeline of segments
3. Expect acknowledgement for each segment
4. Retransmissions triggered by timeouts and duplicate acknowledgements
5. Receiver manages receipt and collation of segments

TCP uses ***checksum*** for error detection. The sender and receiver treat each data segment as a sequence of ints. The sender computes the sum of ints and stores it in the header, and the receiver computes the sum of ints and compares it to the checksum in the header. If the sums do not match, the packet is most likely damaged.

UDP have a number of advantages (over TCP):

- Small segment headers means no connection set up costs
- UDP senders can transmit signals as fast as they like
- Segments are handled independently of each other

This is effective for low-latency apps that can tolerate lost/damaged packets

Some applications that use UDP are: DNS, TFTP, RTSP

Network Layer

The transport layer provides a way for app processes to communicate. The network layer provides communication between hosts, where the hosts are specified by IP addresses.
Basic functions of the network layer (Internet layer):

- For outgoing packets - select the next-hop host, pass packets to link later to transmit to host
- For incoming packets - if the packet has reached its destination, extract payload and pass to transport layer. If it has not reached its destination, treat it as an outgoing packet.
- For all packets/transmissions: error detection, diagnostics
- May also split "oversize" segments into smaller packets.

Network Layer Protocol

IP (Internet Protocol) is a network layer protocol that provides host addressing and routing of packets. It provides spilling and reassembly of large packets.
Routing is one of its most important functions. Each host maintains a ***routing table*** (which maps address to next-hop). It uses ***subnets*** to reduce table size. All hosts in a subnet have a common prefix (e.g. CSE `129.94.2XX.XXX`). All IP addresses with a common prefix are sent to the same host (gateway). The routing table is maintained dynamically. The host transmits 'active' signal to each other periodically.

A simplified routing algorithm (for IP forwarding):

```
Inputs: D = destination IP address
        N = network prefix (of IP address)

if (N matches a directly connected network address)
   send packet over link to D
else if (routing table contains a route for N)
   send packet to next-hop address given in routing table
else if (default route exists in routing table)
   send packet to the default route
else
   can't find route; transmit error message to sender
```

Link Layer

The link layer takes packets from the network layer and transmits them. Every host on the network has a network layer implementation, which is implemented as a combination of hardware/software. Each host contains a ***network interface card*** (NIC) connected to the system bus and an i/o device.

Services provided by the link layer include:

- ***Flow control*** - pacing between adjacent sending and receiving node
- ***Error detection*** - detects transmission errors; flags error to network layer
- ***Error correction*** - can identify and correct single bit errors

If an error is corrected, there is no need for retransmission. If it is not correctable, retransmission is requested.

Ethernet

Ethernet is an example of link layer implementation. Ethernet is a cable physically connection multiple hosts. Data is broadcast onto the cable and tagged with the receiver MAC address. The devices recognise their own data using MAC addresses.
Note: MAC (Media Access Control) addresses are stored in the NIC (Network Interface Card)

Ethernet is a shared broadcast medium, and so we can have interference and collisions. ***Interference*** is when two different packets are broadcast at the same time. ***Collision*** is when a node received two or more signals at the same time.

Multiple access protocols handle this, but it cannot also use ethernet. Example of multiple access protocols:

- ***Channel partitioning*** - partitions channels based on time-slices/frequency-bands etc. It allocates one partition to each node for exclusive use.
- ***Random access*** - allows collisions and needs a mechanism to recover from collisions.
- ***Taking turns*** - nodes take turns sending packets. Nodes with more packets to send get longer turns

A random access transmission control protocol is CSMA (Carrier-Sense Multiple Access)

1. NIC receives packet from network layer and creates frame
2. if NIC senses channel idle, start frame transmission
3. if channel busy, wait until channel idle and go to step 2
4. if entire frame transmitted without interference, go to step 8
5. if NIC detects interference while transmitting, abort transmission
6. after abort, choose "random" delay time (longer if more collisions)
7. after waiting, go to step 2
8. mission accomplished (frame transmitted)

Wireless Sensor Network Security (Non-Examinable)

Sunday, 4 November 2018 10:13 PM

What is *wireless sensor network* (WSN)?

- wireless network consisting of spatially-distributed autonomous devices
- using sensors to cooperatively monitor physical or environmental conditions
    - such as temperature, sound, vibration, pressure, motion or pollutants
- one or multiple base stations; devices/base may be mobile or static

Reasons for popularity:

- low cost
- ability to solve and interact with real world problems and challenges

Security of Wireless Sensor Networks

- operate in *unattended* and *hostile* environments
- potentially, interact with sensitive data

WSNs operate under *resource constraints*

Applications

- Environmental/Habitat monitoring
- Acoustic detection
- Seismic Detection
- Military surveillance
- Inventory tracking
- Medical monitoring
- Smart spaces
- Process Monitoring

Main aspects to discuss ...

1. obstacles to sensor network security
2. requirements of a secure wireless sensor network
3. attacks
4. defensive measures

Obstacles to Sensor Network Security

- Difficult to apply existing approaches to WSN Security
- Very Limited Resources
    - Limited Memory and Storage Space
    - Power Limitation
- Unreliable Communication
    - Unreliable Transfer (Connectionless routing, Packet loss/Error handling)
    - Conflicts (Packet collision)
    - Latency (Multihop routing/network congestion/node processing)
- Unattended Operation
    - Exposure to Physical Attacks (Bad weather etc)
    - Managed Remotely (Physical tampering)
    - No Central Management point

Security Requirements of a Wireless Sensor Network

- Data Confidentiality
    - Military Sensitive Data
    - Encryption
- Data Integrity
- Data Freshness
    - Important because of shared key strategies
- Availability
- Self Organization
- Authentication

WSN Attacks

Types of Attacks:

- Denial of service attack
- The Sybil attack
  - Malicious device taking on multiple identities (Voting systems)
- Traffic Analysis Attack
- Node Replication Attacks
- Attacks against Privacy
  - Monitor and Eavesdropping, Traffic Analysis, Camouflage
- Physical Attacks

Denial of service (DoS) attack

- any event that diminishes or eliminates network's capacity to perform its expected function

Constraints

- Computational Overhead in WSN
- Critical applications

Types of DoS

- Intermittent Jamming
- Constant Jamming
- Link Layer Attacks (Collision)

Sybil Attack

- Malicious device illegitimately taking on multiple identities

Effective against:

- Routing algorithms, Data Aggregation, Voting
- Fair resource allocation and foiling misbehavior detection

Example: Sensor Network Voting Scheme

- Multiple votes registered using multiple identities

Node Replication Attacks

- Add another node to sensor network by replicating node ID

Can lead to:

- packet corruption
- incorrect packet routing

Insert a node at strategic points

- to manipulate a specific segment of the network

Physical Attacks

- Destruction, tampering with circuitry or modification of programming

Causes:

- nodes operate in hostile outdoor environments
- small form factor
- unattended nature of deployment

Other kinds of attack:

- Modification attack
  - Malicious node modifies data in transit between source and destination
- Selective forwarding attack
  - Malicious node intends to selectively drop some packets
- Black-hole attack
  - Malicious node intend to refuse routing and drop all packets through

Defence strategies ...

- Encryption
- Multipath Routing
- Overhearing
- Concealed data aggregation
- More!!!