Overview of Computer Architecture

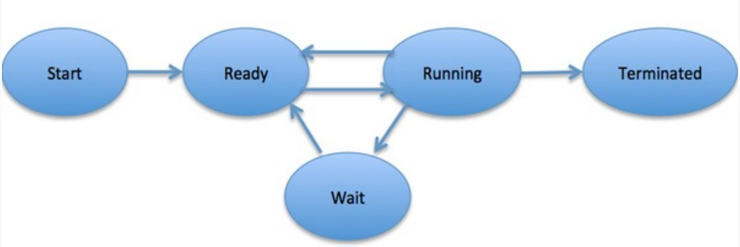
High Level Languages

Carl Henrik Ek - carlhenrik.ek@bristol.ac.uk

December 9, 2019

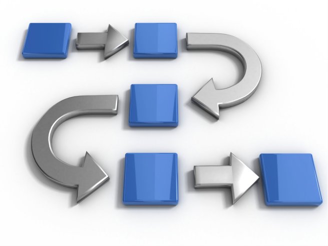
http://carlhenrik.com

Process

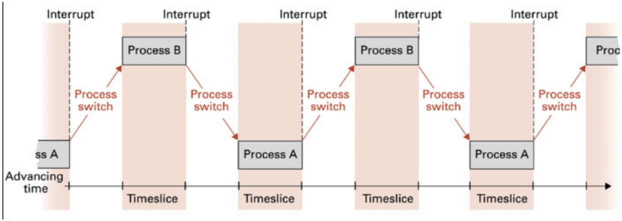


Scheduler

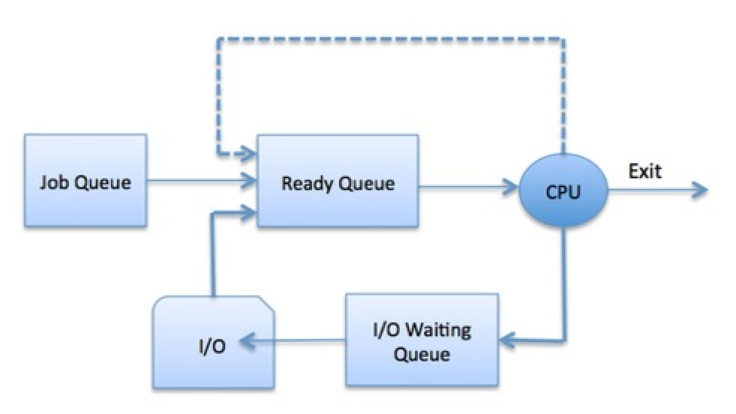
- Program is static
- Execution is not
- Process state
    - Program Counter
    - Registers
    - (Memory)
- Context switch



Scheduler


Interrupt
Interrupt
Interrupt
Interrupt
Interrupt
ss A
Process B
Process A
Process B
Process A
Proc
Process switch
Process switch
Process switch
Process switch
Process switch
Advancing time
Timeslice
Timeslice
Timeslice
Timeslice

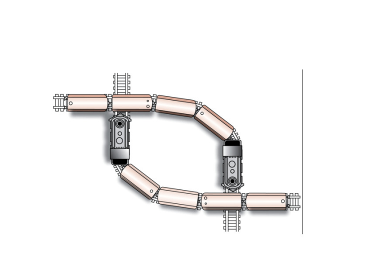
Scheduler



- How to structure execution?

Short Term Scheduler

- How to structure execution?
- Deadlock



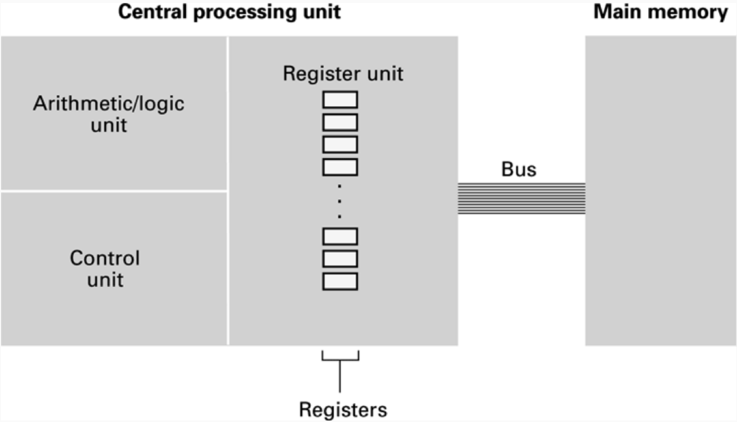
Short Term Scheduler

- How to structure execution?
- Deadlock
- Starvation

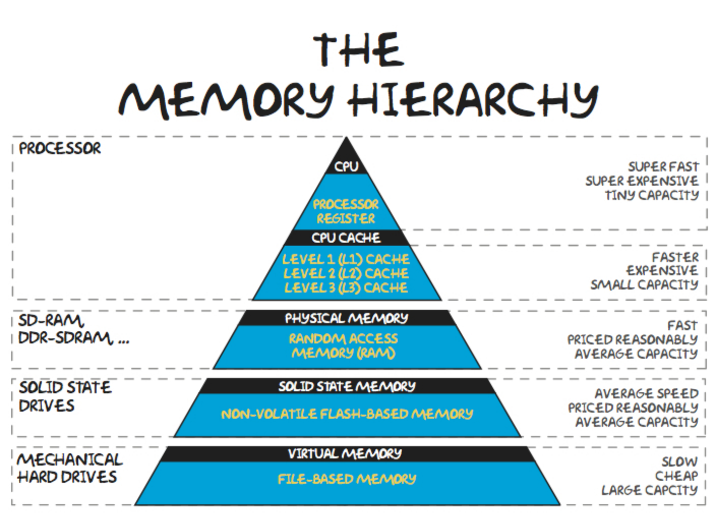
Scheduling

- Maximise Throughput
- Minimise Latency
- Minimise Overhead
- Responsiveness
- Real-time (deadline)

CPU


Central processing unit
Main memory
Arithmetic/logic unit
Register unit
Bus
Control unit
Registers

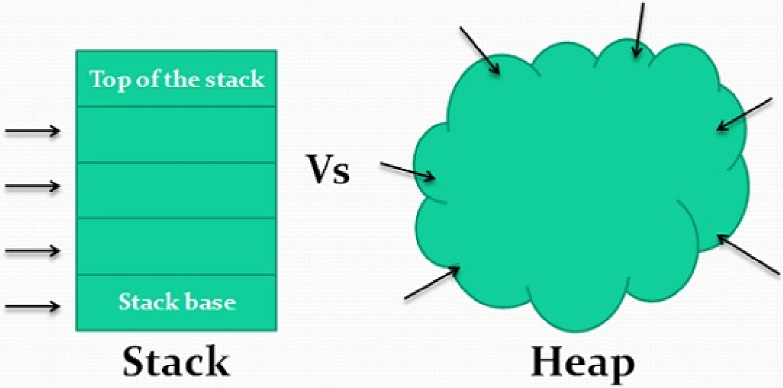
Memory Hierarchy



Memory

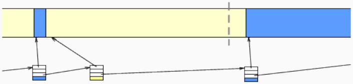
- Bus can address memory
- Tasks
  - Allocation (distribution)
  - Protection
- Fragmentation
- Overhead

Memory: Allocation

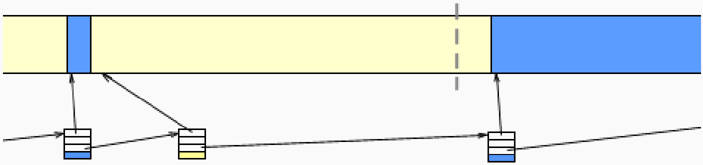


Memory: Allocation

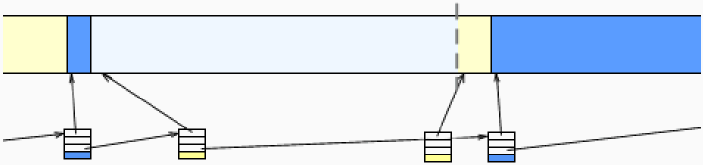
- Allocation
  - First fit
  - Best fit
  - Worst fit
- Sorting for quicker allocation



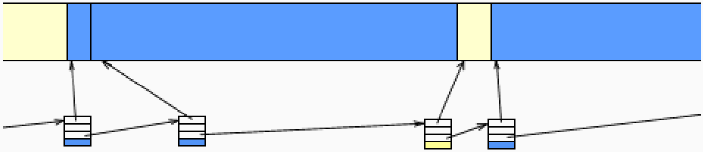
Memory: Allocation



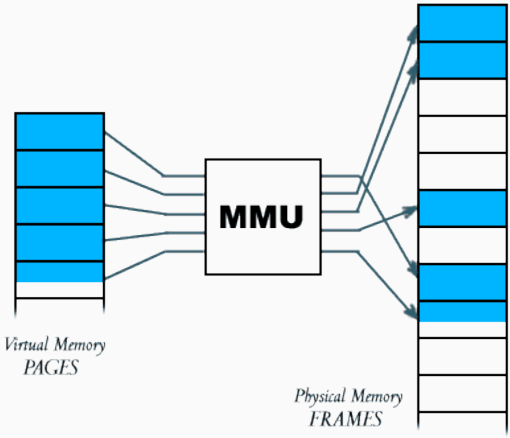
Memory: Allocation



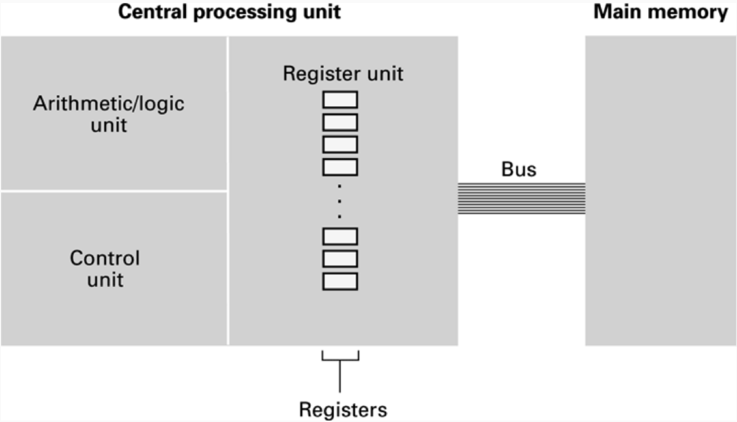
Memory: Allocation



Memory Management Unit



CPU


Central processing unit
Main memory
Arithmetic/logic unit
Register unit
Control unit
Bus
Registers

Summary

- Memory management is very hard
- Trade-off between overhead and efficiency
- Want to avoid fragmentation of memory
- Allocation/Protection

High Level Languages

Code

```
int main(void)
{
  int a = 0;
  for (int i=0;i<200;i++)
    {
      a +=i;
    }
  return a;
}
```

Code

```
_start:
        push    {fp}
        add     fp, sp, #0
        sub     sp, sp, #12
        ......
        ldr     r2, [fp, #-8]
        ldr     r3, [fp, #-12]
        add     r3, r2, r3
        str     r3, [fp, #-8]
        ......
        cmp     r3, #199
        ble     826c <main+0x20>
        ......
        pop     {fp}
        bx      lr
```

High Level Languages

- Assembly code has a *one-to-one* mapping to machine code
- High-level language code does not
- How can we write in an abstract language and convert this to machine code?

High Level Languages

1. Lexer/Tokeniser
2. Parser
3. Translator
4. Optimiser
5. Code Generator

Lexer

Code

```
int main(int argc, char **argv)
```

Lexer

Input sequence of characters

Output sequence of tokens with

- `type` (`KEYWORD`, `WORD`, `LPAREN`, ..)
- `value` ([`WORD` $\rightarrow$"main"], [`LPAREN` $\rightarrow$"("]
- debugging info (file, line, position)

Operation recognise tokens with state machines, can catch tokens that doesn't exists

State Machines

Code

```
int a(int *b){return 0;}
int a) int int *b( {return -1;}
```

- Both of these are valid to the `lexer`
- But only the top is valid `C` code

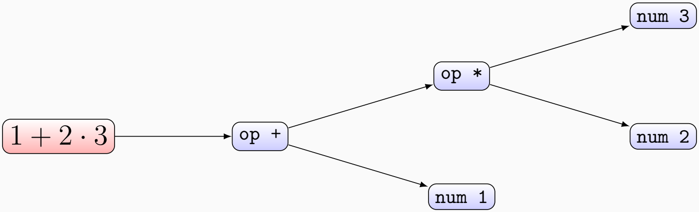
Parser

Input sequence of tokens

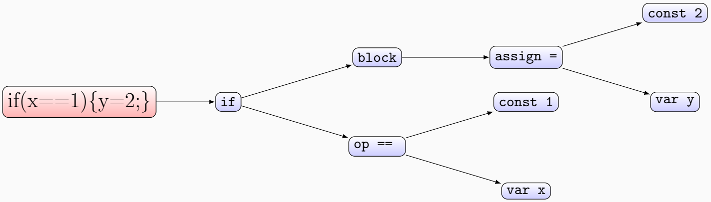
Output syntax tree

Operations very language dependent

Syntax Trees



Syntax Trees

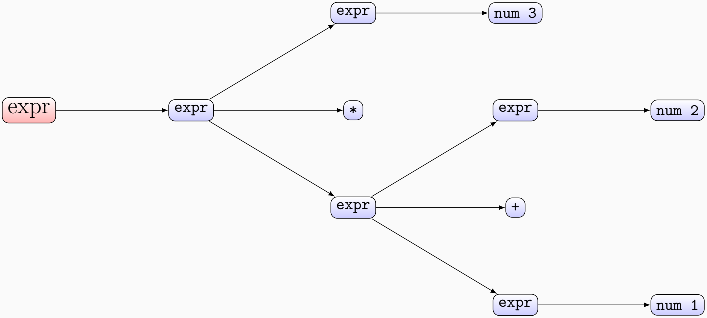


Grammars

$$1 + 2 \cdot 3$$

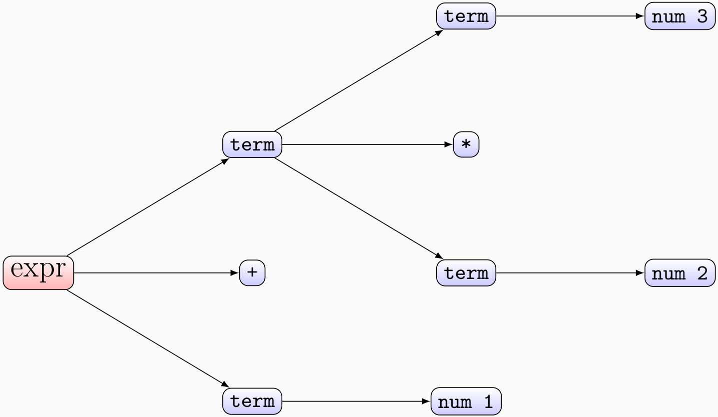
- How can we parse this sentence?
- How can we evaluate this?

Grammars



`expr:  num | expr '+' expr | expr'*'expr | '('expr')'`

Grammars



- `expr: term '+' term | term | num`
- `term: term '*' term | num | '(' expr ')'`

C Grammar

stmt expr';' | cond | block | ..

cond if'('expr')'stmt

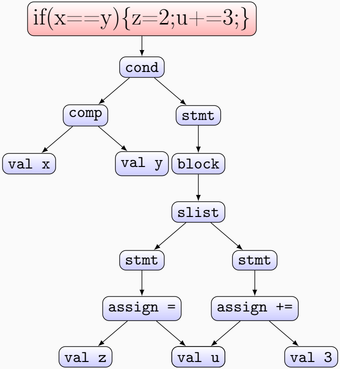
block '{'slist'}'

slist stmt | slist stmt

expr exrp assign expr | expr comp expr | val

assign '' | '+' | '-=' | ...

comp '=' | '!' | '>'



Why do languages look the way they do?

$$1 + 2 \cdot 3$$

$$1\ 2\ 3\ \cdot\ +$$

$$2\ 3\ \cdot\ 1\ +$$

- "Normal notation requires involved syntax tree"
- Reverse Polish Notation directly written as tree

Grammars

- We can check if the code matches the defined grammar of the language
- If we have the associated information of where in the file the token comes from we can potentially output error

Grammars

- We can check if the code matches the defined grammar of the language
- If we have the associated information of where in the file the token comes from we can potentially output error
- *Why are GCCs error messages often not at the right line?*

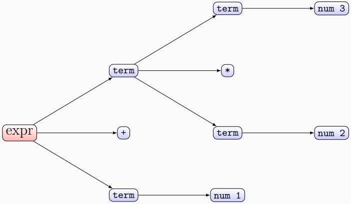
Translator

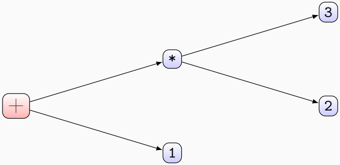
Input Syntax Tree

Output An independent representation (IR)

Operations symbol tables, tree transformations, semantic analysis

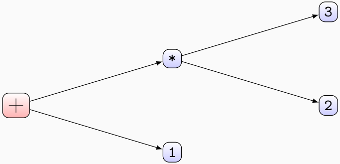
Tree Transformation





Semantic Analysis

- `eval(n) = n`
- `eval(+) = eval(a) + eval(b)`
- `eval(*) = eval(a) * eval(b)`



Evaluation

`eval(add(num 1)(mul(num 2)(num 3))) = eval(num 1) + eval(mul(num 2)(num 3)) = 1 + (eval(num 2) * eval(num 3)) = 1 + (2 * 3)`

Syntax and Semantics

Syntax structure of code

- "The circle square"

Semantics meaning of code

- "The circle is square"

Syntax

Code

```
int main(void)
{
  a = 3;
  int a
  int b = 1;

  return -1;
}
```

Symbol Tables

Code

```c
int main(void)
{
  int a;
  a = 3;
  int b = 1;

  return -1;
}
```

Symbol Tables

Code

```
int x; /*a declaration - goes into the symbol table*/
x = 1; /*a definition - produces machine code*/
```

`C` requires you to declare names (functions, variables, etc) before you use them.

Code

```c
void f()
{
  char x = 2;
  if (x)
    {
      int x = 3;
      fprintf(stdout, "%d\n", x);
    }
}
int main(void)
{
  long x = 1;
  f();
}
```

Scopes

- Why would you want nested variable declarations?
- How to implement nested variable declarations?
- Why would you want to separate declarations and definitions?

Typing

In C `x++` means,

add 1 to `x` if `x` is an integer

add `sizeof(T)` **to** `x` if `x` is a `T*`

an error if `x` is a `struct`

`x+y` needs different instructions in the implementation dependent on type

Different types of typing

Static Typing variables are established ad compile time and cannot change.

Dynamic Typing the type can change during run-time

Dynamic Typing

Code

```
x = 2.0
print(type(x))
x = 'apa'
print(type(x))
```

Asymmetries

$$x = y$$

$$y = x$$

L and R values

- L and R values interact through assignment operators (=)
- L-values refers to memory locations that identifies an object
- R-values refers to values that are stored in memory. We cannot assign a value to a r-value

L - Values

- The name of a variable of any type

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-inderiction (*) that does not refer to a pointer

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indeririction (*) that does not refer to a pointer
- An l-value expression in parentheses

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-inderiction `(*)` that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indericiton (*) that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object
- The result of indirection through a pointer (not function pointer)

L - Values

- The name of a variable of any type
- A subscript that does not evaluate to a pointer
- A unary-indericiton `(*)` that does not refer to a pointer
- An l-value expression in parentheses
- A `const` object
- The result of indirection through a pointer (not function pointer)
- Member access in a struct ->

Examples

Code

```
int a = 1, b;
int *p, *q;

b = a;
*p = 1;
q = p+5;
a + 1 = b;
p = &a;
&a = p;
```

Translator

- Transform syntax tree
- Create Symbol Table
  - Deal with scopes
  - Deal with types
- Normally outputs an intermediate hardware independent representation

High Level Languages

1. Lexer/Tokeniser
2. Parser
3. Translator
4. Optimiser
5. Code Generator

Summary

Summary

- High level languages are really tricky things
- Think about beginning to write `GCC`
- We will continue with the reminder on Friday