Misselaineous Topics

Outline

- Endianness
- const Keyword
- Comma operator
- Static Functions
- NAN and Infinity
- Threads and Processes
- Interprocessor Communications

Endianness

- The order in which bytes of a multi-byte variable are arranged when stored in memory
  - can be extended to transmission
- Common formats:
  - Big-endian:
    most significant byte (contains most significant bit) is stored first (at lowest address)
  - Little-endian:
    least significant byte (contains least significant bit) is stored first (at lowest address)
- Bits Endianness:
  - Order of bits within a byte. Affects bit fields but not bitwise operations
- Can be checked in several ways

const Keyword

- Pointer to a const: `const <type> * <ptr>`
  - defines a pointer `ptr` that points to types of `type`
  - `const` modifier means that the values stored in the pointee cannot be changed

```
int i = 100;    const int* pi = &i;
     /* *pi = 200; <- won't compile */                  pi++;
```

- const pointer
  - you can change the value of the pointee
  - but you can't make the pointer points to a different variable or memory location

```
 int i = 100;     int* const pi = &i;
      *pi = 200;                                                    /*
pi++; <- won't compile */
```

- const Pointer to a const
  - a constant pointer to a constant variable
  - you can NOT change neither where the pointer points nor the value of the pointee

```
int i = 100;    const int* const pi = &i;
/* *pi = 200; <- won't compile */            /* pi++; <- won't compile */
```

const Keyword

- What is the meaning of:
  - ```
    /* pointer to const function -- has no meaning */
    int (const *func)(int);
    ```
  - ```
    /* const pointer to function. Allowed, must be initialized.*/
    int (*const func)(int) = ff;
    ```
  - ```
    /* pointer to function returning pointer to const */
    void const *(*func)(int);
    ```
  - ```
    /* const pointer to function returning pointer to const. */
    void const *(*const func)(int) = ff.
    ```

The comma Operator

- The comma operator:
  - combines the two expressions either side of it into one
  - evaluating them both in left-to-right order
  - the value of the right-hand side is returned as the value of the whole expression
  - (expr1, expr2) is like { expr1; expr2; } but you can use the result of expr2
  - Not recommended, and it is easy to abuse

Example:

```
#include <stdio.h>
int main(){
  int x, y;
  x = 1, 2;
  y = (3,4);
  printf( "%d %d\n", x, y );
}
```

Common in for statements:

```
for (low = 0, high = 100; low < high; low++, high--) {
  /* do something with low and high and put new values in newlow and newhigh */
```

Static Functions

- When a function’s definition prefixed with static keyword it is called a static function
- Have no effect if the program consists of one source file
- A static function is not visible outside of its translation unit:
  - the object file it is compiled into
  - making a function static limits its scope
  - think of a static function as being "private" to its *.c file (and its included files)
- Useful scenario:
  - a program of several files that you include in your main file
  - two of them have a function that is only used internally for convenience called add(int a, b)
  - the compiler would easily create object files for those two modules
  - but the linker will throw an error, because it finds two functions with the same name and it does not know which one it should use (even if there's nothing to link, because they aren't used somewhere else but in it's own file)

NAN and INFINITY

- IEEE 754 FP numbers can represent +∞, -∞, and NaN (not a number).
- Not supported in C89
- In C99:

```
int main(){
  double x = 0/0.;
  double y = 1/0.;
  double z = -1./0.;
  printf("%lf\n", x);
  printf("%lf\n", y);
  printf("%lf\n", z);
  printf("It is %sNAN\n", x!=x?"":"not ");
  printf("It is %sPositive Infinity\n", y==1/0.?"":"not ");
  printf("It is %sNegative Infinity\n", z==-1/0.?"":"not ");
  return 0;
}
```

- Some implementations of math.h defines constants INFINITY and NAN

Threads and processes

- Processes
  - Multiple simultaneous programs
  - Independent memory space
  - Independent open file-descriptors

  In linux/unix process can be forked to sub processes each is a clone of the original the continues execution either from the forking point or from the beginning of the program.

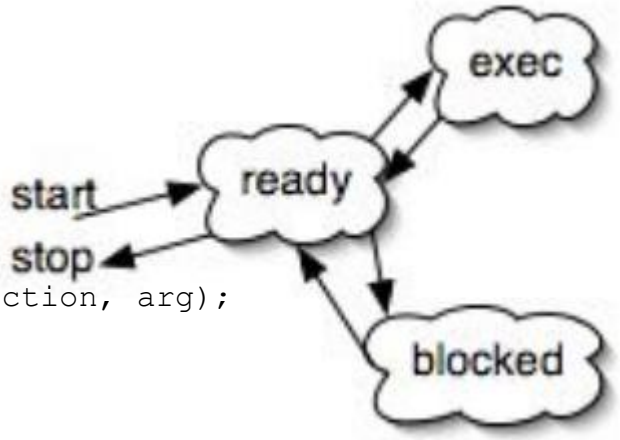
- Threads
  - Multiple simultaneous functions
  - Shared memory space
  - Shared open file-descriptors

  Examples:

  - Web browser tabs (not in google chrome)
  - GUI

Threading

- Shared memory:
  - One copy of the heap
  - One copy of the code
  - Multiple stacks
- Life cycle:
  - `#include <pthread.h>`
  - Define a worker function:
    - `void *foo(void *args){ }`
  - Initialize pthread attr t
    - `pthread_attr t attr;`
    - `pthread_attr init(attr);`
  - Create a thread
    - `pthread_t thread;`
    - `pthread create(&thread, &attr, worker function, arg);`
  - Exit current thread
    - `pthread_exit(status)`



Threading: Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
void *print_hello(void *threadid){
  long tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t = 0; t < NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(threads + t, NULL, print_hello, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  return 0;
}
```

Threading: Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
void* print_hello(void* threadid){
  long tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t = 0; t < NUM_THREADS; t++) {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(threads + t, NULL, print_hello, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  /* wait for all threads to complete */
  for (t = 0; t < NUM_THREADS; t++)
    pthread_join(threads[t], NULL);
  return 0;
}
```

Interprocess Communication

- Each process has its own address space
  - individual processes cannot communicate through program memory unlike threads
- Interprocess communication:
  - Linux/Unix and Windows provide several ways to allow communications:
    - Signals
    - Pipes
    - Sockets
    - RPC
    - clipboard
    - shared memory (linux) and File mapping (windows)
  - Linux/Unix provides
    - FIFO queues
    - semaphores
  - Windows provides:
    - DDE, COM and DCOM
    - Data copy
    - Mailslot