Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
14) C: Additional features

E. Benoist & C. Grothoff

Fall Term 2018-19

▸ Computer Science Division

Additional Features of C

- Macros
- The C preprocessor
- Fork / Processus
- Threads
- Conclusion

Macros

Macros

- `#define` **can be used to define symbolic constant**
  - Numbers that should be changed easily
  - Strings that are stored centrally
- **Can also be used to define macros**
  - Macros perform textual substitution, they are *not* functions
- **Macros syntactically resemble functions**
  - They *may* contain instructions
- **But are also very different**
  - Code is inserted at precompilation
  - There is no “call”
  - Neither a return
  - Text is just: “inserted”!

Macros

- The value is replaced in the source
- `area` is replaced by `length * width` during precompilation

Example

```
#include <stdio.h>
#define area length * width
int main(int argc, char* argv[]){
  int length, width;
  printf("length: ");
  scanf("%d", &length);
  printf("width: ");
  scanf("%d", &width);
  // area is replaced by the value "length * width"
  printf("Area = %d \n",area);
}
```

Macros

More crazy example

```
#include <stdio.h>
#define BEGIN { printf ("Begin!\n");
#define END printf ("End!\n"); }
int main(int argc, char* argv[]){
  if (argc > 2)
  BEGIN
  printf("Middle\n");
  END
}
```

Logging

```
#include <stdio.h>
#define LOG printf ("At line %u:%s\n", __LINE__, __FILE__);
int main(int argc, char* argv[]){
  LOG
  printf("Middle\n");
  LOG
}
```

Macros with arguments

- **One can give arguments to a macro**
  - They are remplaced within the text
  - The result looks like a function
  - Remember: the text is just “replaced”

```
#define area(x,y) x * y
...
printf("Area␣=␣%d␣\n",area(length,width));
/* is equivalent to:
printf("Area = %d \n", length * width);
*/
```

Caveats

```c
#define square(x) x * x

int getv() {
  int i;
  scanf ("%d", &i);
  return i;
}
printf("Square:␣%d\n", square (getv()));
```

More caveats

```
#define square(x) x * x

printf("Square␣of␣6:␣%d\n", square (3 + 3));
```

Solution

```
#define square(x) (x) * (x)

printf("Square␣of␣6:␣%d\n", square (3 + 3));
```

More caveats

```
#define printpos(x) if (x > 0) printf ("↘
→Positive!\n");

int main () {
  int i;
  if (1 != scanf ("%d", &i)) return 1;
  if (0 == i % 2))
    printpos (i);
  else
    printf ("number␣is␣odd\n");
  return 0;
}
```

Common solution

```
#define printpos(x) do { if (x > 0) printf ("↘
→Positive!\n"); } while (0)

int main () {
  int i;
  if (1 != scanf ("%d", &i)) return 1;
  if (0 == i % 2))
    printpos (i);
  else
    printf ("number␣is␣odd\n");
  return 0;
}
```

Macros multiline

- **Macros can need more than one line**
  - Each line must be terminated by \
  - It acts like one single line
  - A macro can represent a compound statement
- **Example**

```
#define print(n) int i; for(i=0;i<n;i++){\
  printf("%d x %d = %d\n",i,i,i*i);\
  }
...
  print(width);
```

The C preprocessor

The C preprocessor

- **Processes directives**
  - Called as first step of C compilation
- **Directives**
  - `#include` and `#define` are directives
  - `#if`, `#elif` (i.e. else if), `#else`, and `#endif`
    this is evaluated BEFORE the compilation, the code not selected is simply not compiled
    Very efficient, the code does not exist for the compiler!
  - `#ifdef` (if defined), `#ifndef` (if not defined) `#undef` (undefine)

C Preprocessor

- **Used to remove Debug information**
  - While debugging phase: DEBUG is 1
  - Debug code is compiled and executed
  - After debugging: DEBUG is 0
  - The debug code will not be present in production executable
- **Example**
  - in the header of the file

```
#define DEBUG 1
```

  - Inside the function

```
#if DEBUG
printf("new car\n");
printf("Address in memory %lX",p_car);
printf("Size in memory %lu\n",sizeof(*p_car));
#endif
```

C Preprocessor

- **Used to insert file only once**
  - `.h` files may be included many times
  - The code should be inserted only once
  - We add a constant and test if it is defined
  - precompilation directive: `#ifndef`
- **Example:** `car.h`

```
#ifndef STRUCT_CAR
#define STRUCT_CAR
typedef struct{
  char id_number[30];
  char brand[80];
  char type[80];
  double price;
} car;
#endif
```

C Preprocessor (Cont.)

- **One should change all the examples**
- **Example:** `myLib.h` **(from the libraries exercise)**

```
#ifndef MY_MATH_LIB
#define MY_MATH_LIB

#define EULER 2.71828
double exponential(long);
long logarithm(long);

long square(long);
long gcd(long, long);
#endif
```

C Preprocessor (Cont.)

- **Example:** `exercise2library.h` **(from the libraries exercise)**

```
#ifndef MY_MATH_LIB2
#define MY_MATH_LIB2

/* the maximal number of factors of a 64bit ↘
→signed number is 62) */
#define PRIMEFACTORLEN 65
void initArray(long [],int ,long);
void primeFactors(long , long []);
void printArray(long []);
double average(long arr[], int size);
double stdDev(long arr[], int size);
double square(double x);

#endif
```

Concatenation

```
int ab;
#define FOO(x) a##x

use (FOO(b));
```

Fork / Processes

Processes

- **A Unix process is an instance of the execution of a program**
  - Has a stackpointer, memory reserved
- **In C, one can generate a new process**
  - By cloning the actual process
  - Function `fork()`
- **Function `fork()`**
  - used to create a new process by duplicating the existing process
  - The existing process from which this function is called becomes the parent process
  - the newly created process becomes the child process.

Fork

- **The child is a duplicate copy of the parent but there are some exceptions to it.**
  - The child has a unique PID like any other process running in the operating system.
  - The child has a parent process ID which is same as the PID of the process that created it.
  - Resource utilization and CPU time counters are reset to zero in child process.
  - Set of pending signals in child is empty.
  - Child does not inherit any timers from its parent
- **Return**
  - If the fork() function is successful then it returns twice.
  - Once it returns in the child process with return value '0' and
  - then it returns in the parent process with child's PID as return value.
  - Same code is executed in parent and in child:
    fork returns twice (once in parent and once in child).

Fork Example

```c
/* Source  http://www.thegeekstuff.com/2012/05/c-fork-function/ */
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
int var_glb; /* A global variable*/
int main(void)
{
    pid_t childPID;
    int var_lcl = 0;
    int i,j,k;
    childPID = fork();
    if(childPID >= 0) // fork was successful
    {
        if(childPID == 0) // child process
        {
            var_lcl++;
            var_glb++;
            printf("\n Child Process :: var_lcl = [%d], var_glb[%d]\n",
            var_lcl, var_glb);
        }
        else //Parent process
        {
            var_lcl = 10;
            var_glb = 20;
            printf("\n Parent process :: var_lcl = [%d], var_glb[%d]\n",
            var_lcl, var_glb);
        }
    }
```

Fork Example (Cont)

```
    else // fork failed
    {
        printf("\n␣Fork␣failed,␣quitting!!!!!!\↘
        →n");
        return 1;
    }
    return 0;
}
/* Output:
bie1@bie1-VirtualBox:~/svn/examples/cAdditional↘
→/parallelism$ ./fork1
 Parent process :: var_lcl = [10], var_glb[20]
 Child Process :: var_lcl = [1], var_glb[1]
*/
```

Fork Example 2

```
...
#define MAXLOOP 10
#define LOOP 1000000
int var_glb; /* A global variable*/
int main(void){
    pid_t childPID;
    int i,j,k;
    childPID = fork();
    if(childPID >= 0) { // fork was successful
        if(childPID == 0) { // child process
            for(i=0;i<MAXLOOP;i++){
              for(j=0;j<LOOP;j++){ k+=1; }
              printf("Child Process: %d \n",i);
            }
        }
        else //Parent process
        {
            for(i=0;i<MAXLOOP;i++){
              for(j=0;j<LOOP;j++){
                k+=1;
              }
              printf("Parent Process: %d\n",i);
            }
        }
    }
    else // fork failed
    {
        printf("\n Fork failed, quitting!!!!!!\n");
        return 1;
    }
    return 0;
}
```

Output

```
bie1@bie1-VirtualBox:~/svn/examples/cAdditional↘
→/parallelism$ ./fork1

Child Process: 0
Parent Process: 0
Parent Process: 1
Child Process: 1
Child Process: 2
Child Process: 3
Parent Process: 2
Parent Process: 3
Child Process: 4
Parent Process: 4
Parent Process: 5
Parent Process: 6
Child Process: 5
Child Process: 6
Child Process: 7
Parent Process: 7
```

Starting a different program

Example:

```
execlp ("/usr/bin/ls", "ls", "-al", "/", ↘
→NULL);
```

The `exec` function family:

```
int execl(const char *path, const char *↘
→arg, ...  /* (char  *) NULL */);
int execlp(const char *file, const char *↘
→arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *↘
→arg, ... /* (char *) NULL, char * const ↘
→envp[] */);
int execv(const char *path, char *const ↘
→argv[]);
int execvp(const char *file, char *const ↘
→argv[]);
int execvpe(const char *file, char *const ↘
→argv[], char *const envp[]);
```

Threads

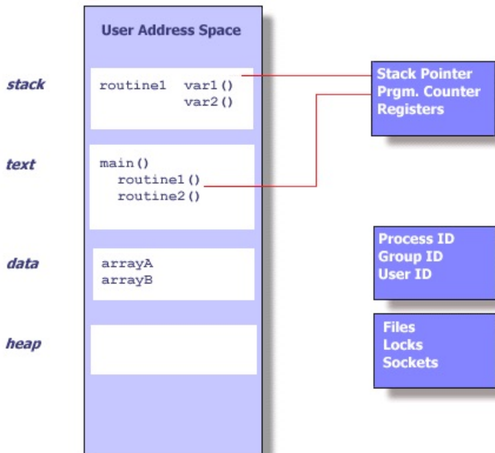
Threads

- A process can have multiple threads of execution
- Each thread is executed concurrently
- Threads may share the same CPU core: preemptive scheduling
- Mapping of threads to cores depends on the hardware and the the OS
- Threads may be used to better use multicore system
- Each thread has its own stack

Threads vs. Processes

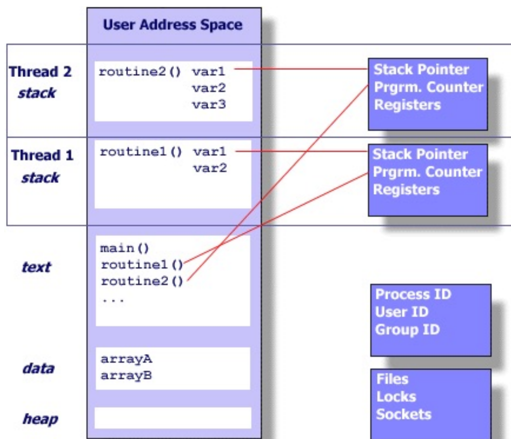
- Processes do not share their address space
- Threads of the same process share the address space.
- Context switching between threads is faster than context switching between processes
- The interaction between two processes requires files or other OS functions called inter process communication (IPC)
- Threads can communicate since they share mutable memory

Memory in a Unix process[1]



[1] Source: `https://computing.llnl.gov/tutorials/pthreads/`

Threads within a Unix process[2]



[2] Source: `https://computing.llnl.gov/tutorials/pthreads/`

Create a thread[3]

- **Define thread reference variables**
  - The variable type `pthread_t` is a means of referencing threads.
  - There needs to be a `pthread_t` variable in existence for every thread being created.
  - Something like `pthread_t thread0;` will do the trick.
- **Create an entry point for the thread**
  - When creating a thread using pthreads, you need to point it to a function for it to start execution.
  - The function must return void * and take a single void * argument.
  - For example, if you want the function to take an integer argument, you will need to pass the address of the integer and dereference it later.

[3] Source: `http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/`

Create a thread (Cont.)

- **Create the thread**
  - we can create the thread using `pthread_create`.
  - This method takes four arguments:
    - a pointer to the `pthread_t` variable,
    - any extra attributes (don't worry about this for now - just set it to `NULL`),
    - a pointer to the function to call (ie: the name of the entry point)
    - and the pointer being passed as the argument to the function.
- **Join everything back up**
  - In the end: join the thread back using `pthread_join` function
  - Parameters: the thread id, and the return value pointer (set it to `NULL`).

Example of use of a thread

```c
#include <pthread.h>
#include <stdio.h>
void *inc_x(void *x_void_ptr){
  int *x_ptr = (int *)x_void_ptr;
  while(++(*x_ptr) < 100);
  printf("x increment finished\n");
  return NULL;
}
int main(){
  int x = 0, y = 0;
  printf("x: %d, y: %d\n", x, y);
  pthread_t inc_x_thread;
  if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
    fprintf(stderr, "Error creating thread\n");
    return 1;
  }
  while(++y < 100);
  printf("y increment finished\n");
  if(pthread_join(inc_x_thread, NULL)) {
    fprintf(stderr, "Error joining thread\n");
    return 2;
  }
  printf("x: %d, y: %d\n", x, y);
  return 0;
}
```

Threads: Syntax

- `pthread_create(&thread_id, NULL, &doSomeThing, NULL);`
  - Creates a new threads
  - The new thread executes the function `doSomeThing`
  - the `thread_id` is set by the function (argument passed by address).
- **Threads share global and static variables (but not automatic variables)**
  - Automatic variables are on the stack, that is different for each thread
  - static and global variables are in the bss or text memory

Example of thread 2

```c
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
void* doSomeThing(void *arg)
{   unsigned long i = 0;
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0])){
        printf("\n First thread processing\n"); }
    else{     printf("\n Second thread processing\n");   }
    for(i=0; i<(0xFFFFFFFF);i++);
    return NULL;
}
int main(void)
{   int i = 0;
    int err;
    while(i < 2)
    {   err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread:[%s]", strerror(err));
        else
            printf("\n Thread created successfully\n");
        i++;
    }
    sleep(5);
    return 0;
}
```

MUTual EXclusion (Mutex)

```c
#include<pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_attr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_settype (&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init (&mutex, &attr);
pthread_mutex_lock (&mutex);
pthread_mutex_unlock (&mutex);
pthread_mutex_destroy (&mutex);
```

Conclusion

Conclusion

- **A lot of different topics**
  - Macros: powerful text substitution, but tricky
  - C Prepocessor directives: Executed before compilation, commonly used to remove code from the executable.
  - Fork vs. Thread: Processes can contain many threads.

Bibliography

- This course corresponds to chapter *14* course book: **Schaum's OuTlines, Programming with C** (second edition), *Byron Gottfied*, Mc Graw-Hill, 1996
- Fork:`http://www.thegeekstuff.com/2012/05/c-fork-function/`
- Threads: `https://computing.llnl.gov/tutorials/pthreads/` `http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/`