C Programming

Lab 1

Marin Iuliana

marin.iulliana25@gmail.com

C Programming

Introduction

Memory allocation

File system management

Process management

Process communication

INTRODUCTION

- C was invented to write an operating system called UNIX.
- Today C is the most widely used and popular System Programming Language.
- Today's most popular Linux OS and RDBMS MySQL have been written in C.

Installing C programming for Linux

- To compile C programs, you need packages containing a compiler, libraries, and man pages. In Linux systems, these packages may be missing:
  - build-essential
  - manpages-dev
- If they are not installed by default on your Linux system, you can use

```
student@sde:~$ sudo apt-get install build-essential manpages-dev
```

Compile a C program on Linux

- In what follows, we will consider the compiler to obtain one or more source files in an executable file.
- We go to the /home/student/sde/tp01/simple-gcc directory where we find the simple_hello.c file.

simple_hello.c

```c
#include <stdio.h>

int main(void)
{
        printf("Hello world!\n");
        return 0;
}
```

```
student@sde$ pwd
/home/student/
student@sde$ cd sde/tp01/simple-gcc
student@sde$ ls
Makefile  hello.c  simple_hello.c  utils.h
errors.c  help.c   utils.c         warnings.c
student@sde$ gcc simple_hello.c
student@sde$ ls
Makefile  errors.c  help.c          utils.c  warnings.c
a.out     hello.c   simple_hello.c  utils.h
student@sde$ ./a.out
Hello world!
```

MEMORY ALLOCATION

- *Malloc* allocates a block of size bytes of memory, returning a pointer to the beginning of the block.

```
#include <stdio.h>

int main ()
{
  int i,n;
  int * pData;
  printf ("Amount of numbers to be entered: ");
  scanf ("%d",&i);
  pData = (int*) malloc (i*sizeof(int));
  if (pData==NULL) exit (1);
  for (n=0;n<i;n++)
  {
    printf ("Enter number #%d: ",n);
    scanf ("%d",&pData[n]);
  }
  printf ("You have entered: ");
  for (n=0;n<i;n++) printf ("%d ",pData[n]);
  free (pData);
  return 0;
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   char name[100];
   char *description;

   strcpy(name, "Ana");

   /* allocate memory dynamically */
   description = malloc( 200 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   }
   else {
      strcpy( description, "Ana is at UPB");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

Resizing and Releasing Memory

- When you are not in need of memory anymore then you should release that memory by calling the function **free()**.
- Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

   char name[100];
   char *description;

   strcpy(name, "Ana");

   /* allocate memory dynamically */
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcpy( description, "Ana is at UPB.");
   }

   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcat( description, "She is in CJ");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );

   /* release memory using free() function */
   free(description);
}
```

File system management

- A file represents a sequence of bytes, regardless of it being a text file or a binary file.
- You can use the **fopen( )** function to create a new file or to open an existing file.

FILE *fopen(const char * filename, const char * mode);

- To close a file, use the fclose( ) function. The prototype is: int fclose(FILE *fp);
- The function to write individual characters to a stream: int fputc(int c, FILE *fp);
- The function to read a single character from a file: int fgetc(FILE * fp);

File opening modes

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Example

- Creates a new file **test.txt** in /tmp directory and writes two lines using two different functions.

```c
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

Example

- Read strings from a file, but it stops reading after encountering the first space character.

```
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

Process management

- A *process* is defined as an instance of a program that is currently running.
- A processor system can still execute multiple processes giving the appearance of a multi-processor machine.
- When a program is called, a process is created and a process ID is issued. The process ID is given by the function getpid() defined in <unistd.h>.
- The prototype for pid( ) is given by

#include <unistd.h>

pid_t getpid(void);

- The *ps* command lists all the current processes.

Multi-process programming

- Multi-process means that each task has its own address space.
- More task isolation and independence compared to multi-threading
- Useful choice for multi-tasking application where tasks have significant requirements in terms of resources
  - Tasks requiring “long” processing times
  - Tasks processing big data structures
  - Tasks featuring high I/O activity (networking, disk accesses, …)

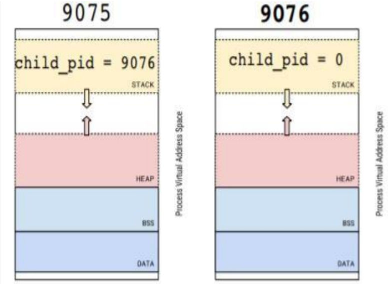
Fork

- Creates a new process duplicating the calling process.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
  pid_t child_pid;
  printf("Main process id = %d (parent PID = %d)\n",
    (int) getpid(), (int)  getppid());

  child_pid = fork();
  if (child_pid != 0)
        printf("Parent: child's process id = %d\n", child_pid);
  else
        printf("Child:  my process id = %d\n", (int) getpid());
  return 0;
}
```




```
$ gcc example1.c -o fork_ex1
$ ./fork_ex1
```

```
Main process id = 9075 (parent PID = 32146)
Parent: child's process id = 9076
Child:  my process id = 9076
```

Multi-Process Programming

- Two processes writing to the standard output

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
    putchar( *str++ );   // Write a char and increment the pointer
    fflush( stdout );    // Print out immediately (no buffering)
    usleep(50);
  }
}
int main() {
  if( fork() == 0 )
    char_at_a_time( ".............." );
  else
    char_at_a_time( "||||||||||||||" );
}
```

```
$ gcc forkme_sync1.cpp -o forkme
$ ./forkme

|.|.|.|.|.|.|..||..|.|.|.|
```

Forking a process with synchronization

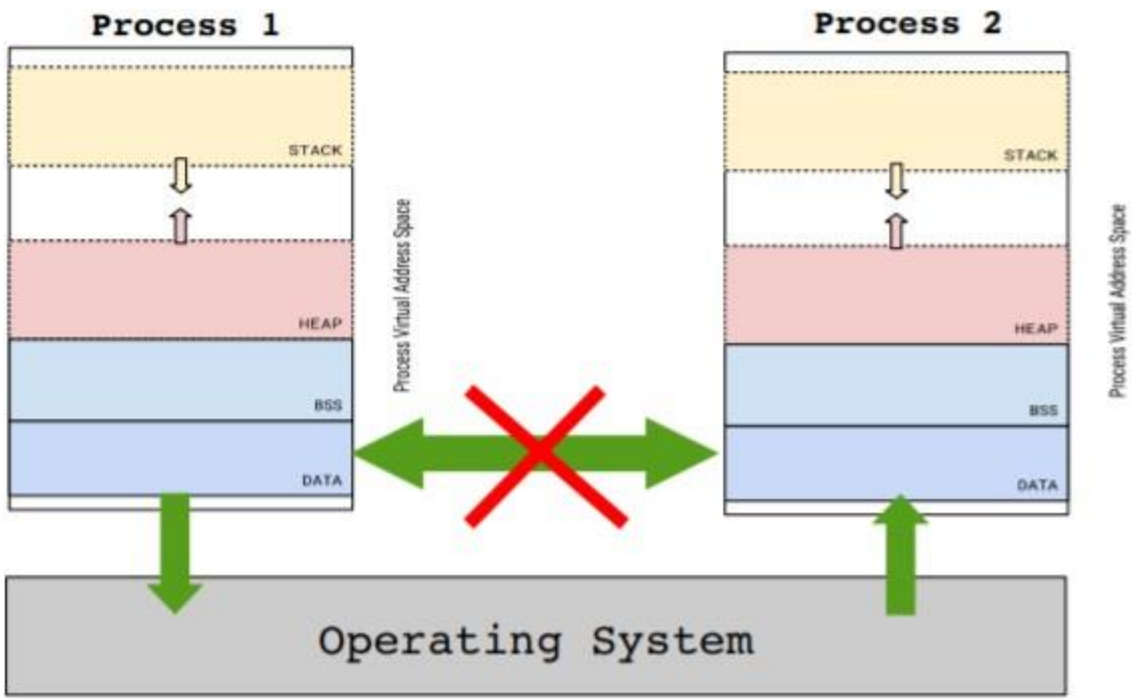
- Synchronization using wait(). The parent process block itself until a status change has occurred in one of the child processes.
- If a child terminates, without wait() performed, it remains in a "zombie" state.
- If a parent process terminates, then its "zombie" children (if any) are adopted by the init process and init automatically performs a wait to remove the zombies.

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
    putchar( *str++ );   // Write a char and increment the pointer
    fflush( stdout );    // Print out immediately (no buffering)
    usleep(50);
  }
}
int main() {
  if( fork() == 0 )
    char_at_a_time( "............." );
  else {
    wait( NULL );
    char_at_a_time( "|||||||||||||" );
}
```

Inter-process Communication

- Operating systems provide system calls on top of which communication mechanisms and APIs are built.



Common POSIX signals

POSIX signals	Portable number	Default action	Description
SIGABRT	6	Terminate	Process abort signal
SIGALRM	14		Alarm clock
SIGCHLD	N/A	Ignore	Child process terminated, stopped or continued
SIGINT	2	Terminate	Terminal interrupt
SIGKILL	9	Terminate	Kill the process
SIGPIPE	N/A	Terminate	Write on a pipe with no one to read it
SIGSEV	N/A	Terminate	Invalid memory reference
SIGUSR1	N/A	Terminate	User-defined signal 1
SIGUSR2	N/A	Terminate	User-defined signal 2
...	...	...	...

USER-DEFINED SIGNAL HANDLING

- Include <signal.h> header file
- Declare a data structure of type sigaction
- Clear the sigaction data structure and then set sa_handler field to point to the handler() function
- Register the signal handler for signal SIGUSR1 by calling the sigaction() function.

```
$ gcc example4.cpp -o sig_example
$ ./sig_example
Running process... (PID=16151)
```

```
$ kill -SIGUSR1 16151
```

```
SIGUSR1 was raised 1 times
```

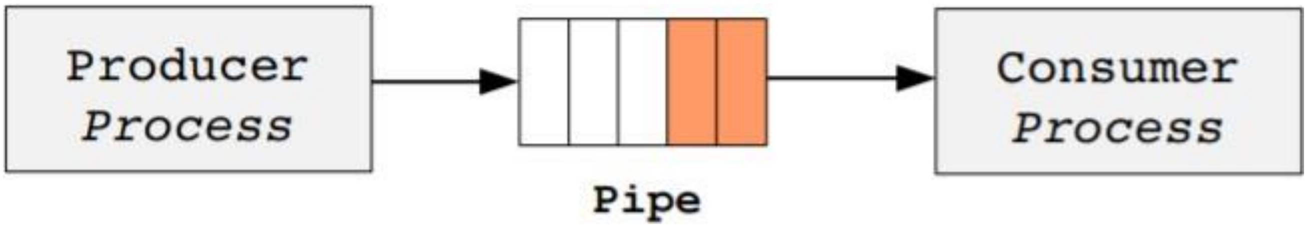
```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
sig_atomic_t sigusr1_count = 0;

void handler (int signal_number) {
  ++sigusr1_count;
}

int main() {
  struct sigaction sa;
  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);
  fprintf(stderr, "Running process... (PID=%d)\n", (int) getpid());
  /* Do some lengthy stuff here.  */
  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

UNNAMED PIPES

- Based on the producer/consumer pattern
- A producer write, a consumer read
- Data are written/read in a First-In First-Out (FIFO) fashion



- In Linux, the operating system guarantees that only one process per time can access the pipe
- Data written by the producer (sender) are stored into a buffer by the operating system until a consumer (receiver) read it

Unnamed pipe messaging

- Create a pipe with pipe() call and initialize the array of file descriptors "fds"
- Fork a child process that will behave as consumer
- Close the write end of the pipe file descriptors array
- Open the read end of the pipe file descriptors array
- Call the reader() function to read data from the pipe
- Parent process acts as producer
- Close the read end of the pipe file descriptors array
- Open the write end of the pipe file descriptors array
- Call the writer() function to write 3 times "Hello, world."

Unnamed pipe messaging (1/2)

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */
void writer(const char * message, int count, FILE * stream) {
  for(; count > 0; --count) {
    fprintf(stream, "%s\n", message);
    fflush(stream);
    sleep(1);
  }
}
void reader(FILE * stream) {
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while(!feof(stream) && !ferror(stream)
        && fgets(buffer, sizeof(buffer), stream) != NULL)
    fputs(buffer, stdout);
}
```

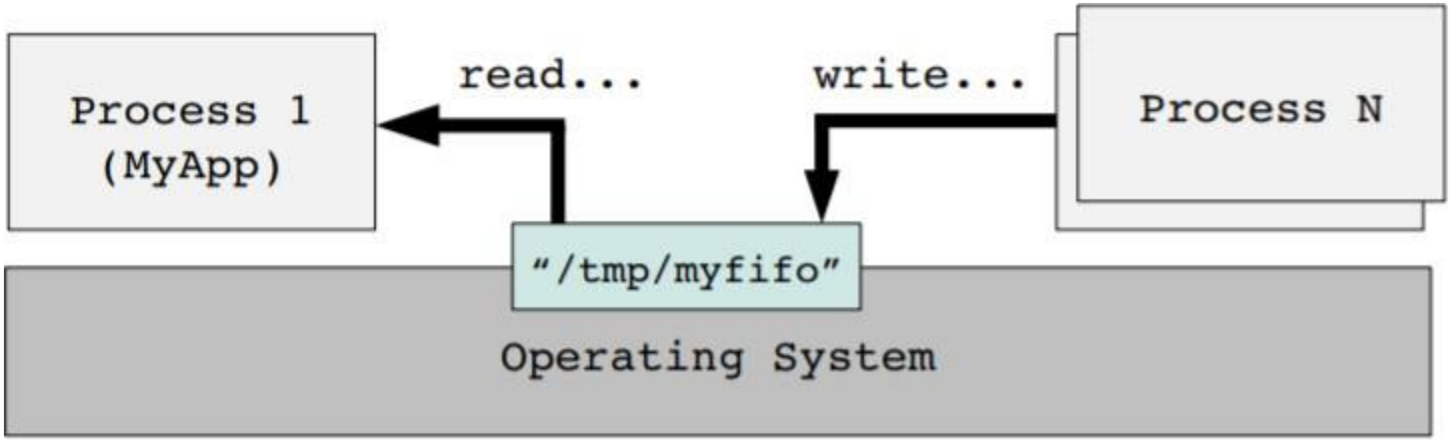
UNNAMED PIPE MESSAGING (2/2)

```c
int main () {
  FILE * stream;
  /* Create pipe place the two ends pipe file descriptors in fds */
  int fds[2];

  pipe(fds);
  pid_t pid = fork();
  if(pid == (pid_t) 0) {     /* Child process (consumer) */
    close(fds[1]);           /* Close the copy of the fds write end */
    stream = fdopen(fds[0], "r");
    reader(stream);
    close(fds[0]);
  }
  else {                     /* Parent process (producer)  */
    close(fds[0]);           /* Close the copy of the fds read end */
    stream = fdopen(fds[1], "w");
    writer("Hello, world.", 3, stream);
    close(fds[1]);
  }
  return 0;
}
```

NAMMED PIPES (FIFO)

- Pipe-based mechanism accessible through file-system
- The pipe appears as a special FIFO file
- The pipe must be opened on both ends (reading and writing)
- OS passes data between processes without performing real I/O
- Suitable for unrelated processes communication



FIFO

- The writer
  - Creates the named pipe (mkfifo)
  - Open the named pipe as a normal file in read/write mode (open)
  - Write as many bytes as the size of the data structure
    - The reader must be in execution (otherwise data are sent to no nobody)
  - Close the file (close) and then release the named pipe (unlink)
- The reader
  - Open the named pipe as a normal file in read only mode (open)
  - The read() function blocks waiting for bytes coming from the writer process
  - Close the file (close) and then release the named pipe (unlink)

FIFO_WRITER.C

```
int main () {
  struct datatype data;
  char * myfifo = "/tmp/myfifo";
  if (mkfifo(myfifo, S_IRUSR | S_IWUSR) != 0)
      perror("Cannot create fifo. Already existing?");

  int fd = open(myfifo, O_RDWR);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }
  int nb = write(fd, &data, sizeof(struct datatype));
  if (nb == 0)
      fprintf(stderr, "Write error\n");

  close(fd);
  unlink(myfifo);
  return 0;
}
```

FIFO_READER.C

```
int main () {
  struct datatype data;
  char * myfifo = "/tmp/myfifo";

  int fd = open(myfifo, O_RDONLY);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }

  read(fd, &data, sizeof(struct datatype));
  …

  close(fd);
  unlink(myfifo);
  return 0;
}
```

MESSAGE READER

- The (a priori known) named pipe location is opened as a regular file (open) to read and write
- Write permission is required to flush data from pipe as they are read
- Blocking read() calls are performed to fetching data from the pipe
- The length of the text string not known a priori '#' is used as special END character
- Close (close) and release the pipe (unlink) when terminate

```
int main () {
  char data = ' ';
  char * myfifo = "/tmp/myfifo";

  int fd = open(myfifo, O_RDWR);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }
  while (data != '#') {
    while (read(fd, &data, 1) && (data != '#'))
        fprintf(stderr, "%c", data);
  }
  close(fd);
  unlink(myfifo);
  return 0;
}
```

```
$ gcc example7.cpp -o ex_npipe
$ ./ex_npipe
Hello!
My name is
Joe
Communication closed
```

```
$ echo "Hello!" > /tmp/myfifo
$ echo "My name is" > /tmp/myfifo
$ echo "Joe" > /tmp/myfifo
$ echo "#" > /tmp/myfifo
```

Exercises

1) Consider the case when several persons play the wireless phone game. They send a message from the first person until the last one and the message gets distorted. Simulate this process according to your own distortion rules.
2) Some of the persons do not want to be understood by others who are around them, so they start to use the chicken language by adding a ‘p’ letter before and after a vowel. Simulate the new game and display the results.