CS2850 Operating System Lab

Week 1: Introduction

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

References

System Programming

Key features of C

Compilers

Standard library

Control flow: `if`, `for`, `while` ...

References

Brian W. Kernighan, Dennis Ritchie: The C Programming LangaugePrentice-Hall 1978 ISBN 0-13-110163-3

Randal Bryant, David O'Hallaron: Computer Systems: A Programmer's Perspective C Pearson Education Limited, 3rd edition, 2016 ISBN-13: 9781292101767.

The GNU C Library Reference Manual

The Operating System (OS)

The OS is a layer of software that

- provides a better, simpler, cleaner, model of the computer and
- helps the user handle resources: processors, disks, printers, keyboard, display, ...

Two popular OS are UNIX and Windows.

Why C-programming?

C is a general-purpose programming language. You can write almost anything in C.

C is not tied to any OS. Your programs will work on any machine.

UNIX is largely written in C.

A relatively low-level language

C does not include

- x operators acting on composite objects, e.g. strings of characters, array,s or lists,
- x Dynamical memory allocation facilities,
- x READ or WRITE statements (you need to *call* dedicated functions),

C is ‘easy’

“... keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in a small place and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language” [1]

[1]from Brian W. Kernighan, Dennis Ritchie: The C Programming Langauge

ANSI C

C is machine-independent.

The program below works on computers with different OS.

```
#include <stdio.h>
int main() {
  printf("hello, world\n");
}
```

To run it, you need the *system-dependent* executable, `a.out`,

```
00000000: 01111111 01000101 01001100 01000110 00000010 00000001  .ELF..
00000006: 00000001 00000000 00000000 00000000 00000000 00000000  ......
0000000c: 00000000 00000000 00000000 00000000 00000011 00000000  ......
00000012: 00111110 00000000 00000001 00000000 00000000 00000000  >.....
....
```

Compilation under UNIX

The OS produces `a.out` from the provided C code[2].

To compile `hello.c`, use the shell command

```
gcc -Wall -Werror -Wpedantic hello.c
```

A useful sanity check of your program is run by entering[3]

```
valgrind ./a.out
```

[2] Use the additional option `-o yourExecutable` to change the executable name.

[3] `valgrind` is a powerful debugging tool for Linux programs.

The Standard Library

ANSI C is based on a *established library* of functions.

You need standard library functions to

- read or write files,
- allocate memory,
- handle strings,
- ....

They are mostly written in C and may contain *a few* non-portable OS details, e.g. system call syntax.

Headers

Write `# include <...>` to make a piece of the library accessible to your program.

`<stdio.h>`: input and output.

`<stdlib.h>`: memory allocation, process control.

`<unistd.h>`: system calls.

`<string.h>`: string-handling.

`<errno.h>`: error reporting.

`<math.h>`: common mathematical functions

Control flow

The control flow fixes the order in which instructions are executed

The most used control-flow statements are

- sequential instructions: "`;`" (default line-by-line execution)
- grouping symbols: `{ ... }`
- selection commands: `if-else`, `switch`, ...
- repetition tools: `for`, `while`, ...

Example

A C program that prints "`hello, world`" several times

```
#include <stdio.h>
#define N 5
int main() {
   int i;
   for (i = 0; i < N; i++) {
      printf("%d) hello, world\n", i + 1);
    }
}
```

The output is

```
1) hello, world
2) hello, world
3) hello, world
4) hello, world
5) hello, world
```

Notes

All variables need to be declared before using them.

`printf` [4] can print

- simple strings: `printf("hello, world\n");`
- variable values: `printf("%d \n", i);` where `%d` specifies that `i` should be printed as an `int`
- a mix of string and values:
  `printf("%d) hello, world \n", i);`

Add `/* .... */` and `// ...` to comment out multiple or single lines.

[4] Defined in `stdio.h`

CS2850 Operating System Lab

Week 2: Types, Variables, Functions

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Programs

C programs consist of functions and variables

A function is a set of statements that specify the operations to be performed

A variable is a location in storage with three attributes:

- an identifier or *name*,
- a storage class determining the variable *lifetime*, and
- a type determining the *meaning* of the value found in the identified storage

Declaration

Functions and variables should always be declared before they are used, i.e. you need to write

```
storage_class variable_type  variable_name;
```

The lifetime of a variable is specified by *where* it is declared (and by its *storage class*).

The declaration of a variable does not initialize it automatically.[1]

[1] But you can separate their declaration and definition.

Example

```c
#include <stdio.h>
void printInt(int j);
int main() {
  int i;
  i = 1;
  printInt(i);
}
void printInt(int k) {
  printf("i=%d\n", k);
}
```

Types, names, storage classes

The variable *type* specifies how the stored bytes should be *interpreted*.

Variable names are sequences of letters and digits used for referring an object, i.e. `i` or `printInt`.

There are two important storage classes[2]:

- automatic: the variable is local to the block and reinitialized every time the function is called and
- static: the variable keeps its value across different function calls.

[2] By default, *local* variables are automatic, *global* variables are static.

Functions

A function is defined by writing

```
return_type  function_name(arguments) {statements};
```

where

- `return_type` is the *type* of the return value,
- `function_name` is the function name (used to *call* it,
- `arguments` denotes the *name and type* of the function parameters, and
- `statements` is the set of *operations* to be performed.

Function calls

A function is called by writing its name

```
f(argument_value_1, argument_value_2, ... );
```

where `argument_value_I` is the value of parameter `I` (at the right place).

In the function declaration, parameter names are *arbitrary*. Only their *type* is strictly required (see the code above for an example).

Types

There are a few basic data types:

- `char` a single character, e.g a
- `int` an integer, e.g. -1234
- `unsigned int` a positive integer, e.g. 12345
- `float` a single-precision floating point, e.g. $23,923$, and
- `void`, an empty set of values [3].

The operator `sizeof()` returns the (architecture dependent) number of bytes of any given basic type.

[3] `void` is not a regular type, e.g. you cannot declare a `void` variable.

Example

Run this program to check the size of the basic C types on your system

```
#include <stdio.h>
int main() {
  printf("sizeof(char)=%lu\n", sizeof(char));
  printf("sizeof(int)=%lu\n", sizeof(int));
  printf("sizeof(unsigned int)=%lu\n", sizeof(unsigned
      int));
  printf("sizeof(float)=%lu\n", sizeof(float));
}
```

More on `int` and `char`

Integer constants can be used to initialize a variable and given in binary (e.g. `int a = 0b01010101;`), decimal (`int a = 85;`), hexadecimal (`int a = 0x55;`) or octal (`int a = 0125;`).

Character constants are characters enclosed in single quotes, (e.g. `char a = 'x';` or `char a = '\n'`).

The numerical value of a `char` is the ASCII integer code corresponding to it, e.g. check the output of

```
printf("ASCII(r)=%d\n", 'r');
```

Printable characters, e.g. `r`, $, or 3, are always positive but plain `char`'s can be signed or unsigned (depending on the machine).

Example

```
#include <stdio.h>
int main() {
  char a = -123;
  printf("character=%c\n", (char) a);
  printf("signed int=%d\n", (int) a);
  printf("signed int(octal)=%o\n", (int) a);
  printf("unsigend int=%u\n", (unsigned int) a);
  printf("unsigend int(octal)=%o\n", (unsigned int) a);
}
```

Type casting specifiers as `(unisigned int)` force a *specified interpretation* of the following variable.

While -123 does not correspond to a *valid character*, the *stored bytes* (printed in octal) can be interpreted in different ways.

Operators

Arithmetic operators: `+`, `-`, `*`, `/`, and `%`.

Relational operators: `==`, `!=`, `>`, `>=`, `<`, and `<=`.

Logical operators: `&&` (and), and `||`(or).

The negation operator, "`!`", converts a non-zero operand into a 0, and a zero operand into 1 i.e. `if (!operand)` and `if(operand == 0)` are *equivalent*.

`%` is the modulus operator, e.g. $1\%2 = 1$, $2\%1 = 0$, $3\%2 = 1$, and $2\%3 = 2$.

Arithmetics

Operators may cause conversion of the value of an operand from one type to another.

For example, integer division may or may not truncate any fractional part, e.g. $3/2 = 1$ but $3/2. = 1.5$ and $3./2 = 1.5$.

The full list of arithmetic conversion rules is in Appendix A6.5 of The C Programming Langauge.

Non-basic types

There is a *conceptually infinite* class of derived types, which are built from the basic types in various ways.

The most important *derived types* are

- strings: null-terminated lists of `char`'s,
- arrays: lists of objects of a given type,
- functions: sets of statements returning objects of a given type,
- pointers: memory addresses of objects of a given type, and
- structures: general composite objects containing objects of different types.

The size of composite objects

The structure of composite objects, e.g. the number of entries of an array, should be declared before they are used and *cannot be changed at run time*.

The size in bytes of composite objects can be obtained using `sizeof`

```
printf("sizeof(char[10])=%lu\n", sizeof(char[10]));
printf("10 * sizeof(char)=%lu\n", 10 * sizeof(char));
```

Strings (1)

The program in the next slide,

i) *declares and initialises* a string (constant),

ii) *prints* the string using `printf` and the format specifier `%s`,

iii) prints a specific *character* of the string and non-initialised value *outside* the string.

Strings (2)

```c
#include <stdio.h>
int main() {
  char *s = "hello, world\n";
  printf("s=%s", s);
  printf("s[7]=%c\n", s[7]);
  printf("s[100]=%c\n", s[100]);
}
```

Strings (3)

The program *does not crash* if you try to access uninitialised entries of `s` but the value stored there is *unpredictable*

The format specifier `%s` allows you to use `printf` for printing `s` with as a unit.

The program *knows where the string terminates* because strings are null-terminated lists of `char`'s, i.e. the last `char` of a string is *always* a `\0`.

Arrays (1)

The program in the next slide,

i) *declares* an array of 10 `int`, i.e. allocate the memory space to store 10 `int`,

ii) *loads* random integers to its entries, through component-wise assignments, and

iii) prints the vector components with `printf` and the format specifier `%d`.

Arrays (2)

```
#include <stdio.h>
#include <stdlib.h>
void loadVector(int a[], int size) {
  for (int i=0; i<size; i++) a[i] = ((float) 10 * rand
      ())/RAND_MAX;
}
void printVector(int a[], int size) {
  for (int i=0; i<size; i++) printf("a[%d]=%d\n",i,a[i
      ]);
}
int main() {
  int size = 10;
  int a[size];
  loadVector(a, size);
  printVector(a, size);
  printf("a[100]=%d\n", a[100]);
}
```

Arrays (3)

If you try to access uninitialized entries of `a` but the value stored there is *unpredictable*.

The program *let you access* uninitialized memory regions, i.e. `a[100]`.

You need a customized function to load and print `a` as a single unit.

`loadVector` and `printVector` do not know the length of the input because `a` is passed as the address of `a[0]`.

CS2850 Operating System Lab

Week 3: Memory basics, Pointers, Arrays

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

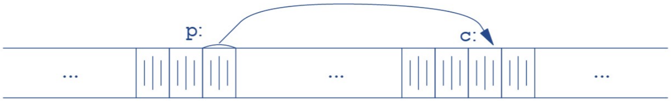
Memory basics

The memory is a *large array* of memory cells (bytes).

The array contains the entire state of your programs: variables, constants, data, and machine code.

Data are stored at specific memory addresses.

Pointers are *variables* for storing the memory address of other variables.



Two useful operators

```
int i = 1;
int *ip = &i;
```

The *address operator*, `&`, returns the *address of* `i`.

The *dereferencing* operator, `*`, returns the value stored at a given address.

`ip` is a pointer variable storing the address of `i`.

Pointers at work

```
int i = 1;
int *ip, *iq;
ip = &i;
*ip = *ip + 1;
iq = ip;
```

You can use `*ip` instead of `i` in any context, e.g. `*ip = *ip + 1;` adds 1 to `i`.

You can *initialize* or *redirect* pointers by copying the content of another pointer of the same type to it, e.g. `iq = ip;`.

Types and pointers

Pointers store the address of specific data type, e.g. `int *ip;` says that `ip` is a *pointer to integers*, i.e. `*ip` is a `int`.

All pointers have the same size, 8 bytes.

The *generic pointer type*, `void *`, can be cast to `void *` and back.

Example

This program returns the content of a `void*` address that is *casted* to a pointer to `int`.

```
int main() {
        int i = 1;
        int *ip = &i;
        void *iq;
        iq = (void *)ip;
        return *(int *)iq;
}
```

To see the return value on the terminal, run

```
gcc -Wall -Werror -Wpedantic pointers.c
./a.out
echo $?
```

Declaring arrays

The following declaration allocates 10 *consecutive* blocks of 4 bytes named `a[0]`, `a[1]`, . . . , `a[9]`,

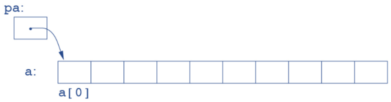
```
int a[10];
```

As `a[0]`, `a[1]`, . . . are *all* integers the program only needs to know the address of the first element, i.e. the *pointer and type to the first element*.

Pointers and arrays

The following defines a pointer to the first element of a,

```
int a[10];
int *pa = &a[0];
```



Pointers and arrays are *closely related*: the value of a (without brackets) is the *address of its first element*.

Strings

```
char *s = "hello world";
```

Strings are *null-terminated* arrays of `char`, i.e. their last `char` is `'\0'`.

The null-termination lets the program find the *end of the string*.

There are *no C operators for processing strings as units*. But you can use `printf("s=%s\n", s);` to print `s` or as because they are null-terminated.

String constants and character arrays are different

```
char *s = "string constant";
char as[20] = "character array";
```

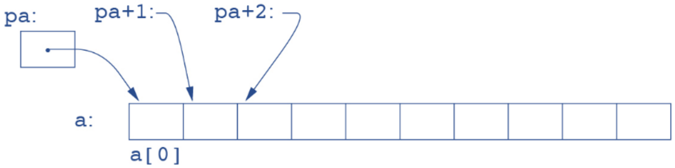
`s` is a pointer *to a constant* (un-modifiable) piece of memory.

`as` is the address of a 20-byte *character array* (you can write on it).

Address arithmetics

Let `a` be an array of 10 `int` and `pa` a pointer to `int`.

After writing `pa = a;`, `pa + 1` points to `a[1]` and `pa + 4` points to `a[4]`[1]



`a[i]` and `*(a+i)` refer to the same object (the content of `a[i]`).

[1] The value of a (without brackets) is the *address of its first element*

Note

Portions of `s` (or `as`) can be accessed by specifying the address of a single character within them.

The following lines print the substrings `"constant"` and `"array"`,

```
char *s = "string constant";
char as[20] = "character array";
printf("%s", &s[7]);
printf("%s", s + 7);
printf("%s", &as[10]);
printf("%s", as + 10);
```

Pointers and functions

Arguments are passed to functions by value.

Functions cannot modify a variable *defined in the calling function*.

To *save the changes* you need to

- define a function with pointer arguments, e.g.

  ```
  void f(int *a) {*a = 5;}
  ```

- pass a pointer when you call `f`.

  ```
  int a = 3;
  f(&a);
  printf("a=%d\n", a);
  ```

Pointers to pointers

Pointers and can store the address of other pointers.

You can use an array of pointers to `char` to store a *list of strings*,

```
char *sa[10];
sa[0] = "hello";
sa[1] = ", ";
sa[2] = "world";
sa[3] = "!";
sa[4] = NULL;
int i = 0;
while (*(sa + i)) {
    printf("sa[%d]=%s\n", i, sa[i]);
    i++;
}
```

Command-line arguments

C programs accept command-line arguments through a strings array called argv.

```
int main(int argc, char **argv) {
    int i = 1;
    while (i < argc) {
            printf("argv[%d]=%s\n", i, argv[i]);
            i++;
    }
}
```

The output is as before,

```
cim-ts-node-02$ ./a.out hello ,  world !
argv[1]=hello
argv[2]=,
argv[3]=world
argv[4]=!
```

CS2850 Operating System Lab

Week 4: processes and IO

nicolo colombo

`nicolo.colombo@rhul.ac.uk`

Office Bedford 2-21

outline

System calls

File system

Processes

`fork`

System calls

The operating system provides its services through a set *system calls*.

Accessing to the OS is mainly needed for

- input/output,
- file handling,
- memory allocation,
- . . .

A portable interface

Different OS's may offer similar services in *different ways*.

In a C program, system calls are architecture-independent, e.g. `printf` performs a system call and its syntax is machine independent.

The portable interface is a set of *standard library functions*.

These routines *depend on the host OS*, i.e. they are written in terms of the facilities of a *specific* OS.

System calls from a C program

From a C program, you can make

- Task-dedicated UNIX system calls, e.g.

  ```
  int n_read = read(int fd, char *buf, int n);
  ```

- Direct system calls, through a general function declared in `unistd.h`. (for system calls that have no *wrapper* in the standard library), e.g.

  ```
  syscall(3, ... );
  ```

  where 3 is the number of the *read* system call.

syscall

`syscall` is a flexible function with a variable number of parameters,

```
long int sys_return = syscall(long int SYS_NO, ... );
```

- `SYS_NO` is the system call number, which identifies each kind of system call,
- the remaining *parameters* depend on the system call and the architecture,
- the return value is the return value of the specific system call.

`syscall` is useful for system calls that have no wrapper in the C library.

Example

The following program prints `hello, world` on screen.

```
#include <unistd.h>
int main() {
  syscall(1, 1, "hello world \n", 20);
}
```

What is the meaning of the first two arguments? What happens if you change the last argument to 6?

Low-level input-output

You can use `syscall` to implement low-level IO operations.

```
n_read = syscall(SYS_read, fd, buf, n)
n_written = syscall(SYS_write, fd, buf, n)
```

where `SYS_read` and `SYS_write` are the numbers associated with the *read* and *write* system calls in `sys/syscall.h`.

Primitive input-output

Reading and *writing* is easier with

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

- `n_read` or `n_written` is the number of bytes actually transferred (if an error occurs, `n_read` $<$ `n`, idem `n_write`),
- `fd` is a *file descriptor* (see below),
- `char *buf` is a character array,
- `n` is the number of bytes to be transferred.

`read` and `write` appear in *higher-level* functions that read or write, e.g. `getchar` or `printf`.

Example (1)

This program copies its input into its output.

```
#include <unistd.h>
int main() {
  char buf[20];
  int n;
  while((n = read(STDIN_FILENO, buf, 20))> 0)
    write(STDOUT_FILENO, buf, n);
}
```

The file descriptors `STDIN_FILENO` and `STDOUT_FILENO` are defined in `unistd.h` and refer to *standard input* (the keyboard) and *standard output* (the terminal).

Example (2)

This program is equivalent to the previous one.

```
#include <unistd.h>
int main() {
  char buf[20];
  int n;
  while((n = syscall(0, 0, buf, 20))> 0)
    syscall(1, 1, buf, n);
}
```

Read more about `read` and `write` on this page and more about `syscall` on this page of the The GNU C Library Reference Manual.

syscall codes

Run the following program to print on screen the syscall codes for a few IO and process control operations.

```
#include <stdio.h>
#include <sys/syscall.h>
int main() {
  printf("SYS_read=%d\n",SYS_read);
  printf("SYS_write=%d\n",SYS_write);
  printf("SYS_open=%d\n",SYS_open);
  printf("SYS_close=%d\n",SYS_close);
  printf("SYS_fork=%d\n",SYS_fork);
  printf("SYS_getpid=%d\n",SYS_getpid);
}
```

Why do you need to include stdio.h here? Why can you use file-IO facilities to read from and write on the terminal?

Streams

A file is a *stream of data*, i.e. a sequence of bytes.

What happens when you open a file?

1. the kernel checks your right to do so (Does the file exist? Do you have permission to access it?),
2. the kernel returns to the program a small non-negative integer called a file descriptor, and
3. the kernel keeps track of all information about the open file, e.g. a file position (offset from the beginning of a file).

File references

Direct system calls, e.g. `read`, `write`, and `syscall`, refers files through their file descriptor.

Higher-level IO facilities defined in the standard library refer to files through their file handle.

A file handle is an object of type `struct` containing

- a pointer to a buffer,
- the number of characters left in the buffer,
- a pointer to the next position in the buffer,
- the file descriptor,
- various flags, e.g. describing the read/write *mode*,
- ...

Unix IO

In a C program, the easiest way to open a file is to call

```
FILE *fopen(char *name, char *mode);
```

- `FILE` is the file handle-type.
- The `name` parameter is the file name (a string constant).

  The `mode` parameter controls how the file is to be opened (`"r"` for *read-access*, `"w"` for *write-access* and `"a"` for *append access*).

File descriptors and file handles

File handles, i.e. `struct` of type `FILE`, are composite objects and contain all the information you need to access an open file.

In most cases, file handles are *passed to functions* by passing a pointer to `FILE`.

File descriptors are simple integers. You can obtain the file descriptor of an open file from the file handle of an open file with

```
int fd = fileno(FILE *fh);
```

Normally, IO functions defined in `stdio.h`, e.g. `fopen` and `fileno`, refer to open files through file handles.

Lower-level file handling

The *lower-level* versions of `fopen` are defined in `fcntl.h` and return a file descriptor.

```
int open(char *name, int flag, int perms)
int creat(char *name, int perms)
```

See The GNU C Library Reference Manual for more details about the arguments `flags`, a parameter that controls how the file is opened, and `perms`, a number encoding the user's permissions.

`stdin` and `stdout`

In UNIX, all IO operations are done by reading or writing files all peripheral devices, even the keyboard and the screen, are files in the file system.

Printing on the terminal or reading from the keyboard is a special case.

All processes automatically open three files called `stdin` (*standard input*), `stdout` (*standard output*), and `stderr` (*standard error*), with fixed file descriptors 0, 1, and 2.

`stdin`, `stdout`, and `stderr` are file handles defined in `stdio.h`.

High-level reading and writing

In a C program, the easiest way to read and write on file is to use

```
int fscanf(FILE *f, char *format, ...)
int fprintf(FILE *f, char *format, ...)
```

Example: the following two calls are equivalent:

```
fprintf(stdout, "hello world");
printf("hello world);
```

Read more about `fscanf` and `fprintf` on the The GNU C Library Reference Manual.

Processes

A process is the OS abstraction of a program in execution.

Each program runs in the context of some process.

The context includes all state information needed to run the process: program code, data, stack, program counter, environment variables, ...

Concurrent processes

You always have the *illusion* that your program is the only one running on the system.

This happens because a process has

- an *independent* control flow and
- a *private* address space, i.e. the memory associated with a process cannot be read or written by any other process.

Multitasking is needed for running multiple *concurrent* processes in an apparently simultaneous way.

Process handling

Unix provides the basic system calls for manipulating processes from C programs.

A process has a unique process identifier (PID) obtained by calling

```
pit_t getpid(void);
```

defined in `unistd.h`.

The type of the return value (a positive integer), `pid_t`, is defined in `types.h`.

You can print the process identifier of your running program at any time by writing

```
printf("pid=%d\n", getpid());
```

States of a process

A process can be in three states:

i) *running*: executing on the CPU,

ii) *stopped*: the execution is suspended (until it receives a signal), or

iii) *terminated*: the process is stopped permanently

Child processes

A process *can create a new running process* by calling

```
pid = fork();
```

which is defined in `unistd.h` and returns twice:

1. `pid` $=0$ in the child process, and
2. `pid` $=$ `pid_child`, where `pid_child` is the *pid of the generated child process* in the parent process.

`fork` returns -1 if the process creation failed.

Process termination

A process can be terminated by writing (in `main`)

- `return i;`

  which *returning the integer* `i`,

- `exit();`

  which has no return value.

Note: you need to include `stdlib.h` to use `exit`.

Child and parent are quasi-identical

`fork` creates a child process that is almost identical to its parent.

The child gets a *separate copy* of the parent's (user level) address space: code, data, heap, user stack.

The child gets copies of all parent's open file descriptors, e.g. the child can read and write any files that are *open in the parent when* `fork` *is called*.

This is why they can both print on `stdout`.

Parent and child are distinct

The parent and the child have different PID's.

`fork` is called once by the parent, but it returns twice, once in the parent and once in the child.

The parent and the child processes run concurrently changes made after calling `fork` are private i.e not reflected in the other process.

Read more about `fork` on The GNU C Library Reference Manual.

Waiting for a child process to terminate

Unix provides a few more system calls for manipulating processes from C programs.

A parent process waits for a child to terminate by calling the

```
pid_t wait(int *status)
```

which is defined in `sys/wait.h`.

The parameter `status` can be used to encode some *information about the termination of the child process*, [1], e.g. its return value.

The return value is *the PID of the child that terminates* or -1 if the process has no children

[1]Set `&status` to `NULL` if you do not need this information

Suspending a process

A process can be suspended by calling

```
unisgned int sleep(unsigned int secs)
```

which is defined in `unistd.h`.

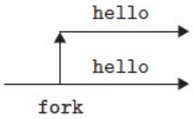
`secs` is the number of seconds the process will sleep.

The return value is the number of seconds left to sleep (if it returns prematurely).

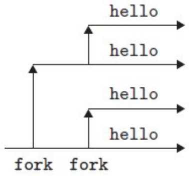
We suggest you read more about `wait` and `sleep` on The GNU C Library Reference Manual.

Examples (1)

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

int main(){
        fork();
        printf("hello\n");
}
```



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

int main(){
        fork();
        fork();
        printf("hello\n");
}
```



Examples (2)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  if (!fork())
    printf("the PID of the child is %d\n", getpid());
  else{
    printf("the return value of wait(NULL) is %d\n",
        wait(NULL));
    printf("the PID of the parent is %d\n", getpid());
  }
}
```

The program above prints

```
the PID of the child is 3384654
the return value of wait(NULL) is 3384654
the PID of the parent is 3384653
```

CS2850 Operating System Lab

Week 5: the UNIX shell

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

How the shell works.

The UNIX shell

Wild cards

Bash programming

How the shell works

A *shell* is a user-level process for interacting with your machine.

The process *starts* when you open the terminal, e.g. when you log in using `puTTY`.

The shell prints a prompt and waits for you to *type a command* in the terminal and *press* `enter`.

The shell executes the insturction and prints a new prompt when it is ready to *accept a new command*.

Command execution

When you run a command the shell creates a new child process.

The child process executes the corresponding program, e.g. `ls` is a program name. [1]

The shell waits for the child process to terminate.

After the shell has reaped the child process, it prints a prompt again and waits for the next input.

[1] The shell assumes the first word of the input is an executable file, like `./a.out`.

Example

The user moves to `Desktop/operatingSystems/`, looks at the directory content, and prints their location.

```
cd Desktop/operatingSystems/
ls
CS2850Labs  intros
pwd
/home/cim/staff/ugqm002/Desktop/operatingSystems/
ls -l CS2850Labs/
total 72
-rwx------ 1 ugqm002 staff 16832 Sep 21 17:56 a.out
drwx------ 2 ugqm002 staff  8192 Oct  4 08:45 example
drwx------ 2 ugqm002 staff  8192 Sep 23 13:01 week1
drwx------ 3 ugqm002 staff  8192 Sep 30 17:23 week2
drwx------ 2 ugqm002 staff  8192 Oct  7 18:08 week3
```

`fork` and `execve`

Child processes are created with `fork` and used to invoke a loader (`execve`) that

1. replaces the child's virtual memory with the info needed *to execute the command*, e.g. the code of the command entered by the user,
2. runs the specified program (within the child process), and
3. terminate the child process.

Example

This program works like a shell and prints `hello, world!` on the screen.

```
#include <unistd.h>
#include <wait.h>
int main() {
  char *argv[2];
  argv[0] = "a.out";
  argv[1] = NULL;
  if (fork() == 0) execv(argv[0], argv);
  wait(NULL);
}
```

`a.out` is the executable of the following code

```
#include <stdio.h>
int main() {
  printf("hello, world!\n");
}
```

The UNIX shell

The first Unix shell was developed for Unix by Steven R. Bourne in 1974.

The *Bourne shell* is now the standard shell in UNIX systems.

Each process created by a Unix shell has three open files:

1. standard input (`fd` = `STDIN_FILENO` = 0),
2. standard output (`fd` = `STDOUT_FILENO` = 1), and
3. standard error (`fd` = `STDERR_FILENO` = 2).

Extra features

IO redirection operators, < and >.

Example: the following lines *write the output* of `ls -l` on `outupt.txt` and print on the screen the first lines of `output.txt`.

```
ls > output.txt
head < output.txt
```

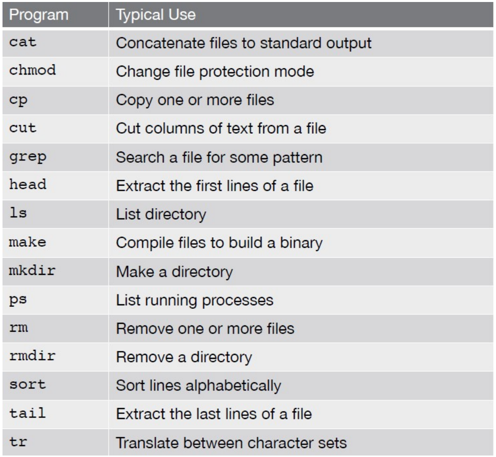
A pipe command, |, to send a command output to another command.[2]

Example: the following line prints only the files and directories modified on October 25th.

```
more output.txt | grep "Oct 25"
```

[2]Without generating extra files.

Shell commands



Program	Typical Use
`cat`	Concatenate files to standard output
`chmod`	Change file protection mode
`cp`	Copy one or more files
`cut`	Cut columns of text from a file
`grep`	Search a file for some pattern
`head`	Extract the first lines of a file
`ls`	List directory
`make`	Compile files to build a binary
`mkdir`	Make a directory
`ps`	List running processes
`rm`	Remove one or more files
`rmdir`	Remove a directory
`sort`	Sort lines alphabetically
`tail`	Extract the last lines of a file
`tr`	Translate between character sets

ls-l

The full information about the files in a directory is obtained by running

```
ls -l directoryPath
```

and includes the *location*, *type*, *size*, the owner and owner's group, their *permissions*, and the last time the file was *modified*, e.g. [3]

```
ls -l
total 96
-rwx------ 1 ugqm002 staff 16840 Oct 20 11:08 a.out
-rw------- 1 ugqm002 staff    63 Oct 20 12:11 helloWorld.c
-rw------- 1 ugqm002 staff   496 Oct 20 12:27 output.txt
drwx------ 2 ugqm002 staff  8192 Sep 10 17:18 w1
```

[3] Directories have a `'d'` as first character.

Permissions

The permissions of files are encoded in 10-character strings, specifying the permissions of the *user*, the user *group*, and the *rest of the world* to *read*, *write*, and *execute* the file (in the order).

The 10 characters below show that the user can read, write, and execute, the user group can read and execute, and the rest of the world can only execute. e.g.

```
-rwxr-x--x
```

chmod

You can set or change the permission bits with

```
chmod [who] operator [permissions] file_name
```

where `who` $\in \{\mathtt{u}, \mathtt{g}, \mathtt{o}, \mathtt{a}\}$, `operator` $\in \{+, -, =\}$, and `permission` $\in \{\mathtt{r}, \mathtt{w}, \mathtt{x}\}$.

Example: If you want to make a file executable to yourself you need

```
chmod u+x file_name
```

Check Section 1.4 of Linux and Shell Programming for using `chmod` in absolute mode, where the operation string with an octal number, e.g. 0100 means *“the owner can execute”*.

Shell variables

The shell uses environment variables to set a working environment when you log in.

You can inspect *all* environment variables by running `env`, e.g.

```
env
SHELL=/bin/bash
LANGUAGE=en_GB:en
PWD=/home/cim/staff/ugqm002/Desktop/operatingSystems
    /2021/labs/week5
...
```

You can print a list of selected variables, `VARNAME1`, `VARNAME2`, . . . , by running

```
echo $VARNAME1 $VARNAME2 ...
```

Setting variable values

You can *set* the value of `VARNAME` to a specific `value` using

```
VARNAME=value
```

This changes the variable only for the current process[4]

You can clear a variable with

```
unset VARNAME
```

[4]Use `export VARNAME` to affects its subprocesses (but not its parent process).

Example

Try the following commands to understand how this works

```
ONE=1; export ONE; TWO=2
./changeVariables.sh
1

1
3
echo $ONE
1
echo $TWO
2
```

`changeVariables.sh` is a file containing the following lines

```
#!/bin/bash
echo $ONE; echo $TWO
ONE=1; TWO=3; export TWO
echo $ONE; echo $TWO
```

Wild cards

The shell has a set of pattern-matching meta-characters to match strings based on patterns:

- `*` matches any string including a null string, e.g. `ls *.c`,
- `?` matches any one character, e.g. `ls ??Script.sh`,
- `[...]` matches any characters enclosed in the square brackets, e.g `ls *[S]*`, and
- `[!...]` matches any characters other than the characters following the exclamation mark, e.g. `ls *.[!s]*`

The resulting general expressions can be used for running shell commands.[5]

[5]The shell expands all wild cards before executing the commands.

Example

The following bash lines combine shell commands and regular expressions

```
ls -l *.* > output.txt
grep '\.c' output.txt
```

`ls -l *.* > output.txt` writes the info of all items with name matching `item_name.ext_name` on `output.txt`

`grep '\.c' output.txt` picks the lines corresponding to all `.c` files.

See more about `grep` (global regular expression print) in Chapter 8 of Linux and Shell Programming.

Shell scripts

You can compose a list of commands for the shell to execute.

A shell script is a file holding *a program built with shell commands*.

Scripts can use some of the *usual constructs* to make simple decisions and loops, i.e. the shell is a programming language.

Example

```
#!/bin/bash
i=start; echo $i > output.txt
for i in 1 2 3 4 5 6 7 8
do
  echo $i "hello world" >> output.txt
done
more output.txt
```

The script above prints the following lines.

```
./myScript.sh
start
1 hello world
2 hello world
...
7 hello world
8 hello world
```

Script variables

Local variables can be declared, modified, and printed on the terminal, e.g.

```
VARNAME=value
$VARNAME=newValue
echo "VARNAME=" $VARNAME
```

Special variables can be used to process the program input, e.g.

$#: number of parameters passed to the script,

$0: script name,

$1: first parameter,

$2: second parameter

...

$?: exit value of last command.

Tests

Let `VAR` be a *string* and `N` an *integer*, e.g. `VAR=one` and `N=1`.

You can make tests through

```
if [ $VAR = test_string ]
if [ $N -eq test_value ]
```

Note: all *empty spaces* are strictly required.

You can modify numerical variables using `expr`, e.g.

```
N=`expr $N + $N + 1`
```

Loops

You can use `for`-loops and `while`-loops as in many other languages, e.g.

```
#!/bin/bash
VAR=0
echo "VAR=" $VAR
for i in 1 2 3; do
        VAR=`expr $VAR + $i`
done
echo "VAR=" $VAR
while [ $VAR -ge 0 ]; do
        VAR=`expr $VAR - 1`
done
echo "VAR=" $VAR
```

Example

Run this script with 2 inputs, $\texttt{string} \in \{\texttt{plus}, \texttt{minus}\}$ and $\texttt{value} \in \mathbb{N}$

```
#!/bin/bash
VAR=$1; N=$2
echo "VAR=" $VAR ", N=" $N
if [ $VAR  =  plus ]; then
  N=`expr $N + 1`; VAR=good
else
  if [ $N -ge 0 ]; then
    N=`expr -1`
  fi
  N=`expr $N - 1`; VAR=...
fi
echo "VAR=" $VAR ", N=" $N
if [ $N -lt 0 ]; then
  echo "wait..."
  while [ $N -le 0 ]; do
    N=`expr $N + 1`; echo "N=" $N
  done
  VAR=good
fi
echo "VAR=" $VAR ", N=" $N
```

CS2850 Operating System Lab

Week 6: Dynamic memory

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

outline

Processes

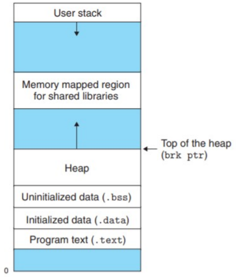
A process is a *running program*.

Each process has a private address space, i.e. a set of memory locations *labeled* from 0 to some maximum.

The address space contains the program

- *executable file*,
- *data*,
- *stack*, and
- *heap*.

Virtual address space


User stack
Memory mapped region for shared libraries
Top of the heap (brk ptr)
Heap
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0

The process *virtual* address space is a list of memory locations.

The program reads from and writes to both the heap and the stack.

The stack

The stack is the region *on the top* of the virtual address space used to implement *function calls*.

It *expands and contracts* when the program runs, e.g.

- it grows each time a function is called, e.g. to allocate *local variables*, and
- it contracts when a function returns as all local variables are *discarded*.

The heap

The heap is a *collection* of various-sized blocks, each block being a slot of virtual memory that is either allocated or free.

The code and data areas are *just below* the run-time heap.

The heap *expands and contracts* dynamically at run time as dynamic memory is allocated.

The kernel maintains a variable that points to the *top of the heap*.

Example

Run this program to check that *static* and *dynamic* objects are allocated in different address space regions.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        char a = 'a';
        char b[10];
        char *aDyn=malloc(sizeof(char));
        *aDyn = 'A';
        char *bDyn=malloc(10 * sizeof(char));
        for (int i=0; i<10; i++) {
                *(b + i) = 'b';
                *(bDyn + i) = 'B';
        }
        printf("a=%c and &a=%p\n", a, (void *) &a);
        printf("b[0]=%c and &b[0]=%p\n", *b, (void *) b);
        printf("*aDyn=%c and &p[0]=%p\n", *aDyn, (void *) aDyn);
        printf("bDyn[0]=%c and &bDyn[0]=%p\n", *bDyn, (void *) bDyn);
        free(aDyn);
        free(bDyn);
}
```

Static memory allocation

In C, the amount of memory used to allocate variables in the stack is determined at compile time.

Stack variables are declared by writing

```
char a;
int b[10];
```

The size of a stack variable can not be changed *at runtime*. In other words, the memory allocation is static.

Dynamic memory allocation

The standard library provides functions for allocating memory in the run-time heap.

Heap memory blocks are allocated dynamically and it is possible to change the size of the corresponding variables.

To change the size of a heap-allocated variable, you need to call a library function that *finds* a new suitable slot and *frees* the old one.

Memory allocator

The library function for allocating memory in the heap is

```
void *malloc(size_t size)
```

`size_t` is the size of the block to be allocated.

For example, you can allocate a *memory region* for hosting an array of 10 integers with

```
int *a;
a = malloc(10 * sizeof(int))
```

`malloc`

`malloc` is defined `stdlib.h` and

- returns a `void` pointer [1], if the allocation is successful, and `NULL` otherwise, and
- creates a new un-typed and uninitialised block of memory of size `size`.

After allocating the right amount of memory you can cast the `void` pointer to any required type.

[1] A pointer to unspecified data

How not to write your program

"ISO C99 introduced the capability to have array dimensions be expressions that are computed as the array is being allocated, and recent versions of `gcc` *support most of the conventions for variable-sized arrays in ISO C99.*[2]

In this course, you should ignore these extensions and always use `malloc` to allocate memory dynamically.

[2]from Section 3.8.5 of Computer Systems: A Programmer's Perspective C

`realloc` and `free`

Other memory-allocation functions defined in `stdlib.h` are

- `void *realloc(void *p, size_t size)`, which changes the size of the object pointed to by `p` to a new given size, `size`, without changing its content (up to `size`), and
- `void free(void *p)`, which de-allocates the *heap space* pointed to by `p`.

The argument of `free` must be a pointer to *previously allocated dynamic memory*. For example, the following statements cause an *execution error*.

```
int i = 1;
free(i);
```

Free the memory

It is good practice to *free any dynamically allocated memory that is no longer needed* because

- memory usage keeps growing with every new allocation and
- it may help avoid memory leaks, i.e. allocated memory slots that are no longer referenced by a pointer.

Suggestion: Always check if your program leaks memory by running

```
valgrind ./a.out
```

Example

This program produces a memory leak of 40 bytes because the corresponding pointer is *moved elsewhere*.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int i = 1;
  int *a =malloc(10 * sizeof(int));
  for (int j=0; j<10; j++)
    *(a+j)=j;
  a = &i;
  printf("a[9]=%d\n", *(a+9));
  //free(a);
}
```

Example

Why does running the program (without the final call to `free`) produce a *strange* output?

```
./a.out
a[9]=-1701673536
```

If you *un-comment* the last line makes the program crash and you get

```
./a.out
a[9]=1133786560
free(): invalid pointer
Aborted (core dumped)
```

valgrind

Running the program with valgrind may help you spot the issue as you get

```
valgrind ./a.out
==1570619== Memcheck, a memory error detector
...
==1570619== Command: ./a.out
a[9]=0
==1570619== HEAP SUMMARY:
==1570619==     in use at exit: 40 bytes in 1 blocks
==1570619==   total heap usage: 2 allocs, 1 frees, 1,064 bytes allocated
==1570619== LEAK SUMMARY:
==1570619==    definitely lost: 40 bytes in 1 blocks
==1570619==    indirectly lost: 0 bytes in 0 blocks
==1570619==      possibly lost: 0 bytes in 0 blocks
==1570619==    still reachable: 0 bytes in 0 blocks
==1570619==         suppressed: 0 bytes in 0 blocks
...
==1570619== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

references

Read more about the `malloc` family on

- this page of the online C manual,
- Section 8.7 of The C Programming Langauge, or
- Section 9.9 of Computer Systems: A Programmer's Perspective C.

CS2850 Operating System Lab

Week 7: Linked Lists

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Structures

Structures help *organize heterogeneous* data.

A structure is a *collection* of variables of *possibly different types*.
The variables are called structure members.

A structure has a single *type-name*, called the structure tag.

Defining a `struct`

Structure are defined *outside* `main` by specifying the type of each struct member, e.g.

```
struct structureTag{
   int member1Name;
   char *member2Name;
};
```

Objects of type `struct structureTag` can be declared *inside* `main` as usual, e.g.

```
int main() {
  int i;
  struct structureTag myStruct;
  ...
}
```

Accessing the `struct` members

There are *two equivalent ways* of accessing struct members, e.g.

```
myStruct.member1Name = 1;
*(myStruct.member2Name) = 'h';
```

or

```
(&myStruct)->member1Name = 1
*((&myStruct)->member2Name) = 'h'
```

Note: the *arrow notation* applies to the structure address.

Global operations

You can copy or assign structures *as a unit*, e.g.

```
struct structureTag myStruct2 = myStruct;
```

You can get the address of the entire `struct` with &, e.g.

```
struct structureTag *pointerToMyStruct= &myStruct;
```

You can compute the size of a structure with `sizeof`, e.g.

```
int sizeOfMyStruct = sizeof(struct structureTag);
```

Illegal operations

You can *not* compare two structures, e.g. you can *not* write

```
if(myStruct1 == myStruct2) {...}
```

You can *not* perform arithmetic operations between structures, e.g. you can *not* write

```
myStruct1 = myStruct1 + myStruct2
```

Why not?

Structures are composite objects and handling them *as a unit* is not always allowed.

Note: all this *does not apply* to single members, e.g. you can have

```
if(myStruct1.member2Name == (&myStruct2)->member2Name) {
    myStruct1.member1Name = myStruct2.member1Name + 1;
    ((&myStruct2)->member1)++;
}
```

Example (1)

```c
#include <stdio.h>
#define MAX 100
struct word{
  int length;
  char s[MAX];
};
int main() {
  struct word helloWorld;
  char *s = "hello, world!";
  helloWorld.length = 0;
  while (*(s + helloWorld.length) != '\0') {
    *(helloWorld.s + helloWorld.length) = *(s + helloWorld.length);
    (&helloWorld)->length++;
  }
  *(helloWorld.s + helloWorld.length) = '\0';
  struct word myStruct = helloWorld;
  printf("myStruct.s=%s\n", myStruct.s);
  printf("myStruct.length=%d\n", myStruct.length);
  printf("sizeof(myStruct)=%lu\n", sizeof(myStruct));
}
```

Example (2)

The program above defines, initializes, copies and prints the content of a struct.

The output is

```
myStruct.s=hello, world!
myStruct.length=13
sizeof(myStruct)=104
```

Why do you need to initialize `myStruct.s` “*element by element*”?

Size of a struct

The size in bytes of structure may depend on the order you declare the struct members, e.g. in

```
#include <stdio.h>
#define MAX 4
struct word1{
  char c;
  int v[MAX];
  char c2;
};
struct word2{
  int v[MAX];
  char c;
  char c2;
};
int main() {
  struct word1 w1;
  struct word2 w2;
  printf("sizeof(w1)=%lu\n", sizeof(w1));
  printf("sizeof(w2)=%lu\n", sizeof(w2));
}
```

Linked lists of structures

Linked lists of structures are also called self-referential structures.

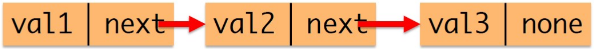
Each node in the list is a *pre-defined* object of type `struct` with *at least* two members:

1. the *value* of the node and
2. a *reference* to the next node in the list.

Implementation

The *reference to the next node* is a pointer to a `struct` object of the *same type*, e.g.

```
struct node {
    int val;
    struct node *next;
};
```



The last element in the list is a null pointer, `NULL`, so you can know when you have reached the end of the list.

Dynamical lists

Linked lists are useful if you *do not know* the number of nodes in advance.

The idea is to create new nodes when new data points *arrive*.

Creating a new node requires allocating a new memory slot of size `sizeof(struct node)`.

The total amount of memory needed is *unknown* at compile time, i.e. you need to allocate nodes dynamically (using `malloc`).

The first node

Dynamic linked lists are built iteratively by linking each new node to the *head of the list*.

A new node is declared and initialized as

```
struct node *pNode = malloc(sizeof(node));
pNode->val = someValue;
pNode->next = NULL;
```

The head

A pointer, `head`, is needed to keep track of *where is the list head*.

Declare it as

```
struct node *head = NULL;
```

and use it to store the address of the list head.

When the list consists of a single node you set

```
head = pNode
```

New nodes

As a new value, e.g. `newValue` = 10, arrives you need to

- allocate memory for a new node

```
struct node *cur = malloc(sizeof(struct node));
```

- store the new value in the node

```
cur->val = newValue;
```

- link the new node to the head and move the head to the new node

```
cur->next = head;
head = cur;
```

Creating the list

Initialize a list for storing a *variable number* of integers (and a 0 in the first node).

```
//headers
struct node{
        int val;
        struct node *next;
};
int main() {
  int n = getchar() - '0';
  struct node *head = NULL;
  struct node *cur = malloc(sizeof(struct node));
  cur->val = 0;
  cur->next = head;
  head = cur;
}
```

Iterations

Iteratively create `n` new nodes and store the first `n` integers into them.

```
for (int i=0; i < n; i++) {
  struct node *cur = malloc(sizeof(struct node));
  cur->val = (i+1);
  cur->next = head;
  head = cur;
}
```

Printing the list

Print information about the nodes (from the last node, i.e. the head, to the first).

```
struct node *cur = head;
for (int i=0; i < n; i++) {
    printf("address node %d = %p\n", n-i, (void *) cur);
    printf("value node %d = %d\n", n-i, cur->val);
    printf("reference node %d = %p\n", n-i, (void *) cur
        ->next);
    printf("-----------------------\n");
    if (cur) cur=cur->next;
}
```

Freeing the list

Free the list nodes (from the last node, i.e. the head to the first).

```
for (int i=0; i < n; i++) {
  struct node *cur = head;
  head  = cur->next;
  free(cur);
 }
```

Freeing the nodes is important to avoid memory leaks.

Other linked lists

A doubly-linked list is a list where each node has two references, one to the *previous* and one to the *next* node.

See C example 7.3 for an example of a doubly-linked list.

A circular list is a *simply-linked* list where the *last* node is connected to the *first*.

See C example 7.4 for an example of a circular list.

References

More about structures can be found in the online C manual or Chapter 6 The C Programming Langauge.

See also the code in C example 7.1 on Moodle.

More about linked lists can be found in Section 6.5 of The C Programming Langaugeor Section 10.2 of Cormen et al.'s Introduction to Algorithms.

See also the code in C example 7.2, C example 7.3, and C example 7.4 on Moodle.

CS2850 Operating System Lab

Week 8: Pipes

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Processes

A *process* is the operating system's abstraction of a running program.

When multiple processes run concurrently, each process appears to have *exclusive* use of the hardware.

The process context is the *state information* the process needs to run.

The OS keeps the context associated with concurrent processes separate.

Inter-process communication

Can two processes communicate?

To connect the context of two processes you need an IPC *system call*.

Example: IPC occurs when a parent process *waits* for a child to terminate (by calling `wait`).

The child sends a *termination* signal to the parent that *can be decoded* through the macro `WEXISTATUS`

example

IPC between a child and its parent.

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <stdlib.h>
int main() {
  int status, pid;
  int N = getchar()- '0';
  int *vOut = malloc(sizeof(int) * N);
  int *vPid = malloc(sizeof(int) * N);
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if (!(pid=fork())) return rand() % (1 + j);
    *(vPid + j) = wait(&status);
    *(vOut + j) = WEXITSTATUS(status);
  }
  for (int k = 0; k < N; k++) printf("%d-pid=%d returns %d \n",
        k, *(vPid + k), *(vOut+k));
  free(vOut);
  free(vPid);
}
```

More general IPC

There exist different types of IPC:

- *Signals*: simple messages that are not used to transfer data,
- *Anonymous pipes*: unidirectional data channels between *related processes*,
- *Named pipes*: more flexible pipes that are treated like a file,
- *Shared memory*: a memory block that can be accessed by multiple concurrent processes,
- ...

See Section 6 of the Linux online manual or Wikipedia page on IPC for more details about UNIX and general IPC.

Linux channels

A channel is an IPC model for message passing.

It has a shareable reference that allows more processes to access it.

Processes can access the channel to write a message *in* it or read a message *from* it.

Channels normally have two distinct references, one for *reading* or *writing*.

They behave like one-way tunnels with two *gates*: an *entry* on one side and an *exit* on the other.

Writing and reading

Processes connected through a channel know the reference of (at least) one end of the channel so that

- one of them can *push* data into the channel and
- one of them can *pull* data out of the channel

i.e. reading and writing happen at different ends.

Written data *travel* through the tunnel before they are read and arrive in the order they are sent (FIFO).

Data written into the channel are *buffered* by the OS until they are read from the other end.

Implementation

In C, you can implement 3 types of channels:

- *half-duplex UNIX pipes*: communication between related processes, e.g. a parent and a child or the children of the same parent,
- *FIFOs named pipes*: communication between two independent processes,
- *sockets*: communication between different computers,
- ...

UNIX half-duplex (*anonymous*) pipes

Anonymous pipes are the eldest of the IPC tools.

A half-duplex pipe connects processes that *share a related ancestor*, i.e.

- a parent and child or
- children of the same parent.

Defining pipes requires kernel-level operations, i.e. a pipe is created by making a *system call*, e.g. by calling `pipe()` in a C program.

2-way pipes can be created by opening two pipes.

Example

Pipes are used in the UNIX shell to connect the *output* of one process to the *input* of another one, e.g. in

```
ls  sort
```

Intuitively, you can imagine data flow through the channel from the *left* to the *right* both the `ls` and `sort` processes are *children of the same process*, i.e. the shell.

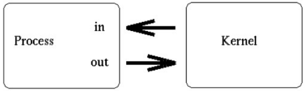
Creating a pipe: step 1

The system call `pipe()` creates two file descriptors associated with the *reading* and *writing* ends:

- `fd0` for *reading*,
- `fd1` for *writing*.

The new pipe initially connects a process to itself.

Data *written to* and *read from* the pipe travels through the kernel.



Step 1: code

```c
#include <stdio.h>
#include <unistd.h>
#define MAXCHARS 100
int main() {
  int fd[2];
  pipe(fd);
  printf("fd[0]=%d, fd[1]=%d\n", fd[0], fd[1]);
  char buf[MAXCHARS];
  *buf = '\0';
  printf("buf = %s\n", buf);
  int nbytes = write(fd[1], "hello world", 12);//write to fd[1]
  printf("%d written to fd[1]\n", nbytes);
  nbytes = read(fd[0], buf, nbytes);
  printf("%d read from fd[0]\n", nbytes);// and read from fd[0]
  printf("buf = %s\n", buf);
  return 0;
}
```

Step 1: output

The output of the program is

```
fd[0]=3, fd[1]=4
buf =
12 bytes written to fd[1]
12 bytes read from fd[0]
buf = hello world
```

File descriptors

File descriptors, e.g. `fd[0]` and `fd[1]`, are used by *low-level* I/O functions as `read` and `write`. [1]

The *standard library* functions refer to files through file *handles* or streams, i.e. *structures* that contain more information about the file.

Given a file descriptor, you can obtain the corresponding stream and *vice versa* through.

```
FILE *streamPointer = fdopen (f0, openMode)
int f1 = fileno(streamPointer)
```

where `f0` and `f1` are (equal) integers, `openMode` a string, e.g. `"w"` or `"r"`, and `streamPointer` a pointer to a `FILE` structure.

[1]See Slides 4 or Section 13.4 of the GNU online manual for more details about low-level I/O

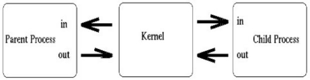
Creating a pipe: step 2

What happens if the process generates a child process *after creating* the pipe?

As `fork` makes an *exact copy* of the parent process, the pipe file descriptors are copied from the parent into the child,

Both processes have then access to the pipe and can use it to *communicate*.

]



Step 2: code

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int copyString(char *out, char *in);
int main() {
  int fd[2], nBytes;
  pipe(fd);
  char out[100], in[100], author[100];
  if (!fork()) {//child process
    sleep(1);
    nBytes=copyString(in, "world");
    copyString(author, "child");
  }
  else { //parent process
    nBytes=copyString(in, "hello");
    copyString(author, "parent");
  }
  nBytes = write(fd[1], in, nBytes);//write to fd[1]
  printf("the %s writes to fd[1]: %s (%d)\n", author, in, nBytes);
  wait(NULL);
  nBytes = read(fd[0], out, nBytes);
  printf("the %s reads from fd[0]: %s (%d)\n",author, out, nBytes);
}
```

Step 2: copyString and program output

copyString is defined by

```
int copyString(char *out, char *in) {
  int k = 0;
  while (*(in + k) != '\0') {
    *(out + k) = *(in + k);
    k++;
  }
  *(out +k) = '\0';
  return k;
}
```

The program produces the following output

```
the parent writes to fd[1]: hello (5)
the child writes to fd[1]: world (5)
the child reads from fd[0]: hello (5)
the parent reads from fd[0]: world (5)
```

Problems

Why do we get such an output?

The child process inherits both file descriptors, `fd[0]` and `fd[1]` both processes can write to and read from the channel.

While the child is sleeping the parent writes to the channel.

The order of the messages is kept in the channel (channels are FIFO containers) so messages written by the parent and the child may get mixed.

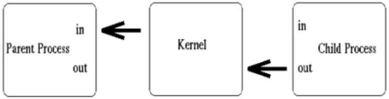
Creating a pipe: step 3

You must decide *who writes* and *who reads* i.e. in which direction data will travel.

The processes should *close* the unused end of the pipe, i.e.

- close `fd[0]` if the process will be *writing* and
- close `fd[1]` if the process will be *reading*

Example: The child *writes* and the parent *reads*.



Pseudo-code

createPipe.c creates a message-passing system between a parent process and its child in a few steps.

- The parent creates a half-duplex pipe. (3)
- The parent generates a child process by calling fork.(5)
- Both processes close the *unused* ends of the pipe (6 and 15).
- The child's sends messages to the parent by calling writeMessages. (9)
- The parent starts reading the child messages and prints them on stdout by calling readMessages.(19)
- The child sends a signal, e.g. a message containing the *keyword* quit, to tell the parent that the *last message* has been sent. (10)
- The child and the parents close the channel. (11 and 20)

createPipe.c (main)

```c
int main() {
  int fd[2];
  pipe(fd);
  char  process[MAXCHARS];
  if (!fork()) {
    close(fd[0]);
    copyString(process, "the child");
    writeMessage(fd[1], "hello", process);
    writeMessage(fd[1], "world", process);
    writeMessage(fd[1], "stop", process);
    close(fd[1]);
    return 0;
  }
  else {
    close(fd[1]);
    copyString(process, "the parent");
    int end = 0;
    while(end == 0)
      end = readMessage(fd[0], process);
    close(fd[0]);
  }
}
```

createPipe.c (Headers and subroutines declaration)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXCHARS 100
int readMessage(int fd, char *reader);
int writeMessage(int fd, char *buf, char *author);
int compareString(char *s1, char *s2);
int copyString(char *in, char *out);
int stringLength(char *s);
```

createPipe.c (Read and write functions)

```c
int writeMessage(int fd, char *buf, char *author) {
  int nbytes = stringLength(buf);
  printf("%s writes to fd[1]: %s (%d) \n", author, buf, nbytes);
  nbytes = write(fd, buf, nbytes + 1);
  return 0;
}
int readMessage(int fd, char *reader) {
  char c = ' ';
  int i = 0;
  char buf[MAXCHARS];
  while (c != '\0') {
    read(fd, &c, 1);
    buf[i++] = c;
  }
  buf[i] = '\0';
  if (compareString(buf, "stop") == 0) return -1;
  printf("%s read from fd[0]: %s (%d) \n", reader, buf, stringLength(buf));
  return 0;
}
```

Step 3: code (string functions)

```
int copyString(char *out, char *in) {
  int k = 0;
  while (*(in + k) != '\0') {
    *(out + k) = *(in + k);
    k++;
  }
  *(out + k) = '\0';
  return k;
}
int stringLength(char *s) {
  int i = 0;
  while (*(s + i) != '\0')
    i++;
  return i;
}
int compareString(char *s1, char *s2) {
  int i = 0;
  while (*(s1 + i) != '\0' && *(s2 + i) != '\0')
    if (*(s1 + i) != *(s2 + i)) return -1;
    else i++;
  if (*(s1 + i) == *(s2 + i)) return 0;
  else return -1;
}
```

Step 3: `createPipe.c` (output)

```
the child writes to fd[1]: hello (5)
the child writes to fd[1]: world (5)
the child writes to fd[1]: stop (4)
the parent read from fd[0]: hello (5)
the parent read from fd[0]: world (5)
```

Note: The implementation of `stringLength` makes the program print the actual size of a word, excluding the null-termination character.

Would it be possible for the parent to read `hello` *before the child writes* `world`?

CS2850 Operating System Lab

Week 9: Threads

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

Outline

Threads

Computers are busy machines.

Threads help them perform several tasks at the same time.

A single process can solve *different related problems* by starting multiple threads.

Threads share the same *global* memory, e.g. data, but have their own stack (*automatic variables*).

Address space

A process memory contains

- process-wide resources, e.g. the program instructions and global data, and
- execution state information, e.g. the program counters and stack.

All threads can access process-wide resources but have private execution state information.

POSIX threads

There are differences between thread packages. Here we focus on Pthreads (for *POSIX* threads).

The Portable Operating System Interface (POSIX) is a set of *types and functions* that you can call from a C program (under UNIX).

Pthreads let you *divide* a program in *sub-tasks* and execute them sequentially or in parallel.

Pthreads in a C program

How to *create* multiple threads in C?

In a C program, you create new threads with a POSIX function called `pthread_create`.

You also need to

- define the task assigned to each thread (normally a subroutine of your program) and
- call a function that merges the output of threads when they return.

All threads are equal

There is *no hierarchy* between running threads, e.g. no parent-child distinctions.

Each thread executes independently: the actual execution order is *unpredictable*.

You synchronize them *explicitly* by waiting until they return.

Handling threads with POSIX *functions* requires to

- *include* the header `pthread.h` and
- *compile* your program with a `-pthread` flag.

Creating a Pthread

You create a new thread with[1]

```
int pthread_create(pthread_t *thread, const
    pthread_attr_t *attr, void *subroutine, void *arg)
```

- thread points to a *buffer* that stores the thread identifier,
- attr points to a struct that specifies various features of the new thread (write NULL for the default),
- subroutine points to the subroutine executed by the new thread, and
- arg points to the parameter of start_routine.

[1]See more about pthread_create on this page of the Linux online manual on Pthreads.

Terminating a Pthread

You wait for a thread to terminate and see its return value with

```
int pthread_join(pthread_t thread, void **retval)
```

- `thread` is the thread identifier and
- `retval` is the location where the exit status of the target thread is copied.

A thread terminates when it reaches a `return` statement or calls

```
void pthread_exit(void *retval)
```

Synchronization

You can ensure that one *event* (in one thread) happens before another event (in a concurrent thread) by synchronizing the two threads.

For example, you can wait for one thread to *return* by calling `pthread_join` in the main program.

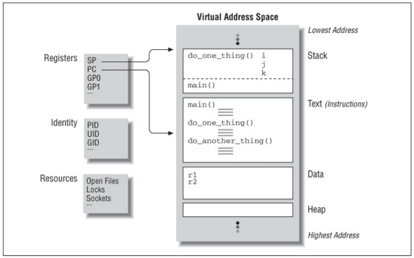
See more about thread synchronization in Chapter 3 of *PThreads Programming* by Farrel et al..

Threads or processes?

In the following examples, we implement the same program using

- no threads,
- a single thread,
- two threads, and
- two child processes.

A process with no threads



- Text contains the program instructions.
- Data are the global data needed to run the program.
- Heap is for dynamic memory allocation.
- Stack is for storing automatic variables.
- Registers, Identity, and Resources contain the info needed to execute.

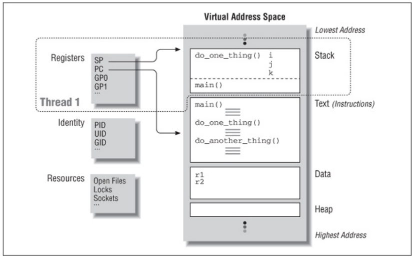
Example: `main` (simple version)

```c
#include <stdio.h>
void sayGoodbye(int n1, int n2);
int brian(int n), dennis(int n);
int main(){
  int nBrian = 3, nDennis = 2;
  while(nBrian + nDennis > 0) {
    if (nBrian) nBrian = brian(nBrian);
    if (nDennis) nDennis = dennis(nDennis);
    sayGoodbye(nBrian, nDennis);
  }
}
int brian(int n) {
  for (int i = 0; i < n; i++) printf("%d-hi Dennis!\n", i + 1);
  return n - 1;
}
int dennis(int n) {
  for (int i = 0; i < n; i++) printf("%d-hi Brian!\n", i + 1);
  return n -1;
}
void sayGoodbye(int n1, int n2) {
    printf("%d-goodbye!\n", n1 + n2);
}
```

Output

```
 ./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

A process with a single Pthreads



Stack and Registers are now part of the thread.

Each *active thread* has a reserved stack frame and specific machine registers.

Example: main (Pthread compatible)

pthread_create takes three free arguments

```
int pthread_create(pthread_t *tName, NULL, void *f(void
    *), void *arg)
```

void *f(void *) is (a pointer to) a function of type void which takes a pointer to void as a single argument.

The function arguments should be passed through a single pointer to void.

You need to rewrite brian and dennis as

```
void *brian(void *n)
void *dennis(void *n)
```

New version of brian and dennis

To meet the requirement of `pthread_create` you need a structure holding the original integer argument,

```
struct arg {
  int n;
};
```

In the first line of `f`, cast the `void` pointer to be a pointer to a variable of type `struct arg`[2]

```
  struct arg *p = n;
```

In main, `n` becomes `(&argName)->n`, where `argName` is an object of type `struct arg`.

[2]To access the original arguments with `p->n`.

New version of brian and dennis

You obtain

```
void *brian(void *nIn) {
  struct arg *nStack = nIn;
  for (int i=0; i<nStack->n; i++)
    printf("%d-hi Dennis!\n", i + 1);
  nStack->n = nStack->n - 1;
  return NULL;
}

void *dennis(void *nIn) {
  struct arg *nStack = nIn;
  for (int i=0; i<nStack->n; i++)
    printf("%d-hi Brian!\n", i + 1);
  nStack->n = nStack->n - 1;
  return NULL;
}
```

New version of brian and dennis

The new version of `main` is

```
#include <stdio.h>

struct arg{
  int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  struct arg nBrian = {3}, nDennis = {2};
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) brian(&nBrian);
    if (nDennis.n) dennis(&nDennis);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

`sayGoodbye` can be left unchanged.

Example (1 running Pthread)

```
#include <stdio.h>
#include <pthread.h>

struct arg{
  int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  pthread_t t1;
  struct arg nBrian = {3}, nDennis = {2};
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) pthread_create(&t1, NULL, brian, &nBrian);
    pthread_join(t1, NULL);
    if (nDennis.n) pthread_create(&t1, NULL, dennis, &nDennis);
    pthread_join(t1, NULL);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

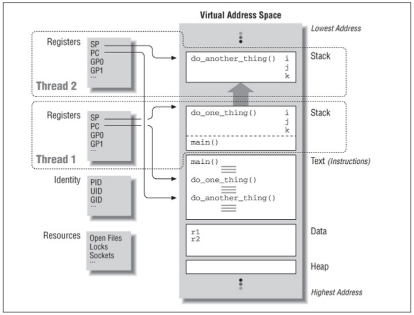
All auxiliary functions are defined as above.

Output (1 running Pthread)

The execution is as before

```
 ./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

A process with two threads



Both threads have *their private copy* of the machine registers and can use variables or file descriptors in the *process-wide areas*.

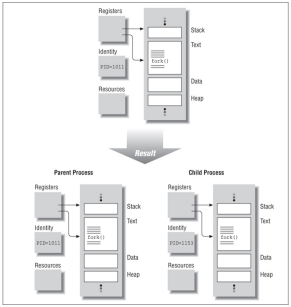
The two threads execute at different locations.

Example (2 running Pthreads)

```
#include <stdio.h>
#include <pthread.h>

struct arg{
        int n;
};

void *brian(void *n), *dennis(void *n), sayGoodbye(int n1, int n2);

int main(){
  struct arg nBrian = {3}, nDennis = {2};
  pthread_t t1, t2;
  while(nBrian.n + nDennis.n > 0) {
    if (nBrian.n) pthread_create(&t1, NULL, brian, &nBrian);
    if (nDennis.n) pthread_create(&t2, NULL, dennis, &nDennis);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

As `t1` starts before `t2` and the task performed by `brian` is *computationally cheap*, the output on `stdout` is unchanged.

Two processes vs. two threads



The address space of two concurrent processes created by `fork` is *completely private* (no shared areas).

The two processes can interact via a channel (anonymous pipe).

Example: `main` (2 running child processes, slowBrian)

```
int main() {
  int status;
  while(nBrian.n + nDennis.n > 0) {
    int pidBrian = fork();
    if (!pidBrian) {
      if (nBrian.n) brianSlow(&nBrian);
      return nBrian.n;
    }
    int pidDennis = fork();
    if (!pidDennis) {
      if (nDennis.n) dennis(&nDennis);
      return nDennis.n;
    }
    waitpid(pidBrian, &status, 0);
    nBrian.n = WEXITSTATUS(status);
    waitpid(pidDennis, &status, 0);
    nDennis.n = WEXITSTATUS(status);
    sayGoodbye(nBrian.n, nDennis.n);
  }
}
```

Include `unistd.h` and `sys/wait.h` to use `fork` and `wait`.

Output (2 running child processes, slow Brian)

The output is the same as before

```
 ./a.out
1-hi Dennis!
1-hi Brian!
2-hi Brian!
2-hi Dennis!
3-hi Dennis!
3-goodbye!
1-hi Dennis!
1-hi Brian!
2-hi Dennis!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

Another example of synchronization: Mutexes

Variables of type `pthread_mutex_t` allow you to synchronize the access to some *area of your code*.

The protected area is mutually exclusive. While one thread is in the protected area, other threads cannot

- *execute* the protected code and
- *lock or unlock* the Mutex.

Mutex-protected variables

To ue a Mutex to regulate the *updates of shared variable*, `x`,

1. create and initialize a mutex variable, `m`, using
`pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
(`m` should be declared outside `main`).

2. in the thread subroutine, *lock and unlock* the mutex before updating `x`, e.g. write

```
pthread_mutex_lock(&m)
x= ...
pthread_mutex_unlock(&m)
```

Example: (2 running threads with thread-safe subroutines)

The *thread-safe* version of `slowBrian` and `dennis` is

```
void *brianSlow(void *n) {
  struct arg *n1 = n;
  int t = 0;
  pthread_mutex_lock(&m); //start of the mutual exclusive area
  for (int i=0; i<(n1->n); i++) {
    printf("%d-hi Dennis!\n", i + 1);
    while (t < 100000) t++;
  }
  pthread_mutex_unlock(&m);//end of the mutual exclusive area
  n1->n = n1->n - 1;
  return NULL;
}

void *dennis(void *n) {
  struct arg *n1 = n;
  pthread_mutex_lock(&m);//start of the mutual exclusive area
  for (int i=0; i<(n1->n); i++) printf("%d-hi Brian!\n", i + 1);
  pthread_mutex_unlock(&m);//end of the mutual exclusive area
  (n1->n) = (n1->n) -1;
  return NULL;
}
```

Output (2 running threads, thread-safe subroutines)

```
./a.out
1-hi Dennis!
2-hi Dennis!
3-hi Dennis!
1-hi Brian!
2-hi Brian!
3-goodbye!
1-hi Dennis!
2-hi Dennis!
1-hi Brian!
1-goodbye!
1-hi Dennis!
0-goodbye!
```

CS2850 Operating System Lab

Week 10: Review of pointers, strings, and process control

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

outline

Pointers (1)

Pointers are variables that store the memory address of other variables.

Address operator: `&i` is the address of `i`.

Dereferencing operator: `*ip` is the value of the variable stored at the address `ip`.

You can print on `stdout` the *unpredictable value* of `ip`, i.e. the address stored in `ip`, with

```
printf("%p",( void *) ip)
```

Pointers (2)

The type of a pointer is the type of variable stored at the pointed address.

The data type of a variable specifies how to interpret the data stored in the variable.

A pointer of a given type cannot store the address of variables of a different type, e.g.

```
int i = 1;
char *pc = &i
```

is illegal and produces compilation errors.

Example: `int`, `char`, `int*`, and `char*` (1)

```
#include <stdio.h>
int main() {
  int i = 10;
  int *pi = &i;
  char c = 'a';
  char *pc = &c;
  printf("sizeof(pi)=%lu\n", sizeof(pi));
  printf("sizeof(pc)=%lu\n", sizeof(pc));
  printf("pi=%p and  &i = %p\n", (void *) pi, (void *) &
      i);
  printf("pc=%p and &c = %p\n", (void *) pc, (void *) &c
      );
  printf("*pi=%d and i = %d\n", *pi, i);
  printf("*pc=%c and c = %d\n", *pc, c);
}
```

Output

```
./a.out
sizeof(pi)=8
sizeof(pc)=8
pi=0x7ffd17e6b974 and  &i = 0x7ffd17e6b974
pc=0x7ffd17e6b973 and &c = 0x7ffd17e6b973
*pi=10 and i = 10
*pc=a and c = 97
```

Strings (1)

String constants are *null-terminated* arrays of `char`, e.g.

```
char *s = "hello world"
```

`s` is a pointer to `char` that stores the address of the first element of the string `"hello world"`.

The last `char` of `s` is the null character `'\0'`.

Strings (2)

String *constants* and character arrays, e.g. `s` and `as` defined by

```
char *s = "hello world";
char as[12] = "hello world";
```

are *not equivalent*.

`s` is a pointer (a variable) and can be *assigned to a different address*.

`as` is an array and always refers to *fixed allocated* storage.

Strings (3)

`s[7]` and `as[7]` both return `’w’` but

`s[7] = ’x’;`

causes a `Segmentation Fault`, while

`as[7] = ’x’;`

is legal and sets the 8th character of `as` to `’x’`, i.e. `’o’` becomes `’x’`.

Strings (4)

You can use `printf("%s", s)` and `printf("%s", as)` to print the strings `s` and `as` up to their null-termination character.

In C, there are *no built-in operators* for processing an entire string as a unit, e.g.

```
char as[12];
as = 'hi world';
```

is *illegal*, but you can *re-assign* `s` to point to another string constant, e.g.

```
char *s;
s = 'hi world';
```

Strings (6)

You can print a portion of `s` or `as` by specifying the address of a single character, e.g.

```
s = "hi world";
printf("(s + 3) = %s", (s + 3));
```

prints `(s + 3) = world`.

You can cut a character array by setting one element to `'\0'`, e.g.

```
char as[10] = "hi world";
*(as + 2) = '\0';
printf("as = %s and (as + 3) = %s\n", s, (s + 3));
```

prints `as = hi and (as + 3) = world`.

Example: char* and char[13]

```
#include <stdio.h>
int main() {
  char *s = "hello world";
  char sa[13] = "hello world";
  //s[7] = 'x'; is illegal
  sa[7] = 'x';//is legal
  printf("s[7] = %c\n",s[7]);
  printf("sa[7] = %c\n",sa[7]);
  printf("(s + 1) = %s\n",s + 1) ;
  printf("(sa + 1) = %s\n",sa + 1);
  s = "hi world";
  printf("s = %s\n",s);
  //sa = s; is illegal
  s = &sa[6];//is legal and equivalent to s = sa + 6;
  printf("s = &sa[6] => s=%s\n",s);
  printf("(sa + 6) = %p and s = %p\n",(void *) (sa + 6),
       (void *)s);
}
```

Output

```
./a.out
s[7] = o
sa[7] = x
(s + 1) = ello world
(sa + 1) = ello wxrld
s = hi world
s = &sa[6] => s=wxrld
(sa + 6) = 0x7ffd23e608c1 and s = 0x7ffd23e608c1
```

Example: pointer arithmetic

```c
#include <stdio.h>
int main() {
  char as[100] = "one two three four five";
  char vs[100]= "one two three four five";
  char *s = "one two three four five";
  for (int i = 0; i < 13; i++) s++;
  as[13] = '\0';
  printf("as = %s\n", as);
  printf("vs = %s\n", vs);
  printf("s = %s\n", s);
}
```

Output:

```
as = one two three
vs = one two three four five
s =  four five
```

Pointers and functions (1)

The de-referencing operator can be used in function declarations, e.g.

```
double f(char *c);
```

says that `f` returns a `double` and the argument of `f` is a pointer to `char`.

Arguments are passed to functions by value. There is *no direct way* for the called function to modify a variable in the calling function.

To change the value of a variable in the calling function, pass the address of that variable.

Pointers and functions (2)

Arrays are represented by *the location of the initial element*.

When you pass an array to a function, you will see the changes in `main`, e.g.

```
#include <stdio.h>
void initialise(int *a) {
  int i;
  for (i = 0; i < 10; i++) *(a + i) = i;
}
int main() {
  int i, a[10];
  initialise(a);
  for (i = 0; i < 10; i++) printf("%d ", *(a  + i));
}
```

prints `0 1 2 3 4 5 6 7 8 9`.

Example: swapping int

```
#include <stdio.h>
void swap(int *pi, int *pj);
void swapWrong(int i, int j);
int main() {
  int x = 1, y = 2, *px = &x,  *py = &y;
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
  printf("calling swapWrong ... \n");
  swapWrong(x, y);
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
  printf("calling swap ... \n");
  swap(&x, &y);
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
}
```

Output:

```
./a.out
x = 1, y = 2, px = 0x7fff73496720, and py = 0x7fff73496724
calling swapWrong ...
x = 1, y = 2, px = 0x7fff73496720, and py = 0x7fff73496724
calling swap ...
x = 2, y = 1, px = 0x7fff73496720, and py = 0x7fff73496724
```

Auxiliary functions

A function to swap the *content of two addresses*.

```
void swap(int *pi, int *pj) {
  int temp = *pi;
  *pi = *pj;
  *pj = temp;
}
```

A function to swap two *stack variables*.[1]

```
void swapWrong(int i, int j) {
  int temp = i;
  i = j;
  j = temp;
}
```

[1] All changes are discarded when the function returns.

Process control (1)

`unistd.h` and `sys/wait.h` contains the functions you need to control a process from a C program.

`int getpid()` returns the process identifier (PID) of the running process.

`int fork()` creates a child process that is a copy of the running process.

`void exit()` or `return` terminate the current process.

`int wait(int * status)`: to wait for the *first* child that terminates (call it several times if you have generated more than one child). [2]

[2] See more about process control in Section 26 of the GNU online manual.

Process control (2)

The OS assigns PIDs in increasing order but their values are unpredictable.

In C, `fork` is *called once but returns twice*.

The return value is -1 if the process creation failed, 0 in the child process, and the child's PID in the parent process.

Process control (3)

When `fork` returns, the child address space is a copy of the parent address space, but the parent and the child have different PIDs.

The return values of `fork` allow you to make conditional statements.

The address spaces of the child and the parent are *private*, i.e. the parent cannot see the changes made by the child, and vice versa.

The parent and the child run concurrently. The execution order is unpredictable.

Example: private but identical address spaces

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  int j = 0;
  printf("PID = %d, j = %d, &j = %p \n", getpid(), j, (void *) &j);
  int pid  = fork();
  if (!pid) j = 1;
  else j = 2;
  printf("PID = %d, j = %d, &j = %p \n", getpid(), j, (void *) &j);
}
```

Output:

```
./a.out
PID = 2789388, j = 0, &j = 0x7fff5ea69010
PID = 2789388, j = 2, &j = 0x7fff5ea69010
PID = 2789389, j = 1, &j = 0x7fff5ea69010
```