CS211: Programing For Operating Systems

Robert Davidson

Contents

1 Intro to C

C is a compiled language, not an interpretive language. Meaning we need a program called a compiler to convert the code into machine code. The compiler is called **gcc**

It is a **very small language and relies heavily on libraries**. The compiler must be told in advance how these functions should be used. So before the compilation process, the **preprocessor** is run to include the function prototypes The compiler then compiles the code into an object file.

1.1 Variables

In C, variables must be declared before they're used. Declarion should have a type to tell compiler what data the variable will hold

- **int** : Integer (1, 2, 3, 4, 5, ...)
- **float** : Floating-point number (7 decimal digits)
- **double** : Double-precision floating-point number (15 decimal digits)
- **char** : Character (a, b, c, ...)
- **void** : No type (used for functions that do not return a value)

1.2 Operators

Operator	Description	Example
+	Addition	`a + b`
-	Subtraction	`a - b`
*	Multiplication	`a * b`
/	Division	`a / b`
%	Modulus	`a % b`

Table 1: Arithmetic Operators

Operator	Description	Example
=	Assignment	`a = b`
+=	Add and assign	`a += b`
-=	Subtract and assign	`a -= b`
*=	Multiply and assign	`a *= b`
/=	Divide and assign	`a /= b`
%=	Modulus and assign	`a %= b`
++	Increment	`a++`
–	Decrement	`a-`

Table 2: Assignment and Arithmetic Assignment Operators

Operator	Description	Example
==	Equal	`a == b`
!=	Not Equal	`a != b`
>	Greater	`a > b`
<	Less	`a < b`
>=	Greater or Equal	`a >= b`
<=	Less or Equal	`a <= b`

Table 3: Relational Operators

Operator	Description	Example
&&	Logical AND	`a && b`
||	Logical OR	`a || b`
!	Logical NOT	`!a`

Table 4: Logical Operators

1.3 Control Flow

1.4 If Else

Listing 1: If-Else

```c
int a = 10;
if(a > 10){
        printf("a is greater than 10\n");
    }else if(a == 10){
        printf("a is equal to 10\n");
    }else{
printf("a is less than 10\n");
}
```

Logical opeators, `&&` and `||` can be used to make more complex conditions.

Listing 2: Complex If-Else

```c
if(a > 10 && a < 20){
    printf("a is between 10 and 20\n");
}
```

For loop

```
for(initial val; continuation condition; increment/decrement){...}
```

Listing 3: Print numbers from 0 to 9

```c
for(int i = 0; i < 10; i++){
    printf("i is %d\n", i);
}
```

While loop

```
while(expression){...}
```

Listing 4: Print numbers from 0 to 9

```c
int i = 0;
while(i < 10){
    printf("i is %d\n", i);
    i++;
}
```

Do While loop

```
do{...}while(expression);
```

Listing 5: Print numbers from 0 to 9

```c
int i = 0;
do{
    printf("i is %d\n", i);
    i++;
}while(i < 10);
```

1.5 Output

`printf()` is used to print formatted output to the screen. It is a variadic function, meaning it can take any number of arguments. The first argument is a format string, followed by the values to be printed.

The format string may contain a number of escape characters, represented by a backslash. Some of the most common escape characters are:

Sequence	Description
`\a`	Produces a beep or flash
`\b`	Moves cursor to last column of previous line
`\f`	Moves cursor to start of next page
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\\`	Prints a backslash
`\'`	Prints a single quote

A conversion character is a letter that follows a `%` and tells `printf()` to display the value stored in the corresponding variable. Some of the most common conversion characters are:

Specifier	Description
`%c`	Single character (char)
`%d` or `%i`	Decimal integer (int)
`%e` or `%E`	Floating-point (scientific notation)
`%f`	Floating-point value (float)
`%g` or `%G`	Same as %e/%E or %f, whichever is shorter
`%s`	String (char array)
`%u`	Unsigned int
`%x`	Hexadecimal integer
`%p`	Pointer (memory address)
`%%`	Prints the % character

1.6 Input

`scanf()` reads input from standat input, format it, as directed by a conversion character and store the address of a specified variable.

Listing 6: Reading an integer

```c
int number;
char letter;
printf("Enter a number and a char: ");
scanf("%d %c", &number, &letter);

printf("You entered: %d and %c\n", number, letter);
```

- The scan `scanf()` returns an integer equal to the number of successfull conversions made.
- There is related function `fscanf()` that reads from a file. `scanf()` is really just a wrapper for `fscanf()` that treats the keyboard as a file.
- There are other useful functions for readint the standard input stream: `getchar()` and `gets()`.

Listing 7: Check for no input

```c
int number;
printf("Enter a number between 1 and 30: ");
scanf("%d", &number);

while ((number<1) || (number>30))
{
    printf("Invalid number. Please enter a number between 1 and 30: ");
    scanf("%d", &number);
}
```

2 Functions

2.1 Protype and Definition

Protype : A function prototype is a declaration of a function that tells the compiler what the function looks like. It includes the function name, return type, and parameter types. The prototype must be declared before the function is called.

Definition : A function definition is the actual implementation of the function. It includes the function name, return type, parameter types, and the body of the function. The definition must be declared after the function is called.

2.2 Call-by-value and Pointers

In C is it important to distuinguish between a variable and the value stored in it. A variable has a location in memory. The value of the variable is stored in that location. For example:

```c
int i = 10;
```

tells the system to allocate a location in memory to store the value 10. The variable `i` is a pointer to that location in memory. One of the distuinguihing features of C is it we can manipulate the memory address of the variable almost as easily as we can manipulate the value stored in it.

Pointers

- if `i` is a variable, then `&i` is a pointer to the location in memory where the value of `i` is stored.
- The declaration `int *p;` creates a variable `p` that can store the memory address of an integer. The `*` indicates that `p` is a pointer to an integer.
- If a memory address is stored in the variable `p`, then `*p` is the value stored at that address.

3 Characters and Strings

In C, a character is just an unsigned integer. Each character is represented by an integer between 0 and 127.

3.1 Reading and writing characters

Printing characters:

- `printf("&c", c)`
- `putchar(c)`

Reading characters:

- `scanf("&c", c)`
- `c = getchar()`

3.2 Strings

C does not have a string data type. Instead it uses arrays of type char to represent strings. For example if we make a declaration: `char greeting[20]="Hello. How are you?";` the system stores each character as an element of the array greeting.

3.3 String functions

Useful functions defined in `string.h`:

srncpy()
`char *strncp(char *dest, char *src, int n)`
Copies at most `n` characters from `src` to `dest`. The advantage of this is that we don't copy more chatavyers to `dest` than it can hold (prevents buffer overflow). `strncpy()` does not append a null character to the end of the string, so we need to do that manually.

strcat()
`char *strcat(char *dest, char *src)`
Concatenates the string `src` to the end of the string `dest`. The destination string must be large enough to hold the concatenated result. `strcat()` appends a null character to the end of the string.

strcmp()
`int strcmp(char *str1, char *str2)`
Compares two strings annd returns an intger:

- 0 if the strings are equal
- A negative integer if `str1` comes first alphabetically
- A negative integer if `str2` comes first alphabetically

strstr()
`char *strstr(char *haystack, char *needle)`
Searches for the first occurrence of the string `needle` in the string `haystack`. If found, it returns a pointer to the first occurrence of `needle` in `haystack`. If not found, it returns NULL.

strlen()
`int strlen(char *str)`
Returns the length of the string `str` (not including the null character).

3.3.1 String Output

We know how to use `printf()` to print strings:

- `printf("%s%s\n, "Good Morning ", name);`
- `printf("%s%8s\n", "Good Morning ", name);`

The second example, the field width specifier is given. This causes the string to be padded so it takes up a total of 8 spaces. If the string is shorter than 8 characters, it is padded with spaces on the left. If the string is longer than 8 characters, it is printed as is.

String Input
Input is a more complicated issue, there are three basis methods:

- `scanf("%s", name);` reads the next "word" from th input stream and stores it in the array `name`. A "word" is defined as a sequence of characters separated without a space, tab, or newline. The string is terminated with a null character.
- `getchar(name);` to get more control of the input in a loop
- `gets(string);` reads a line input and stores it all except the newline character. The string is terminated with a null character. `gets()` is not safe to use because it does not check the length of the input string. If the input string is longer than the array, it will cause a buffer overflow and overwrite other data in memory.
- `fgets(string, n, stdin);` reads in a line of text from the keyboard (standard input stream) and stores at most `n` characters in the array `string`. The string is terminated with a null character. If the input string is longer than `n`, it will be truncated and the rest will be discarded. `fgets()` is safer to use than `gets()` because it checks the length of the input string.

4 Arrays

To declare a 3×4 matrix of floats, we write `float a[3][4];`. So:

$$\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] & a[0][3] \\ a[1][0] & a[1][1] & a[1][2] & a[1][3] \\ a[2][0] & a[2][1] & a[2][2] & a[2][3] \end{bmatrix}$$

In general, an $n \times m$ array is declared as `float a[n][m];`. The first index is the row number and the second index is the column number. The first element of the array is `a[0][0]` and the last element is `a[n-1][m-1]`.

If a has the line `int [a][4];` then the system creats three arrays, each of length four. More precisely, it:

- declares 3 pointers to type `int`: `a[0]`, `a[1]`, and `a[2]`
- space for storing an integer is allocated to each of addresses `a[0], a[0]+1, a[0]+2, a[0]+3, a[1], a[1]+1`, ..., `a[2]+3`

This mean is if `a[][]` is declared as a two-dimensional 3×4 array, then the following are equivalent:

- `a[1][2]`
- `*(a[1]+2)`
- `*(*(a+1)+2)`
- `*(&a[0][0]+4+2)`

4.1 Explanation of Array Element Access in C

`a[1][2]`

This is the standard way to access a two-dimensional array element. It directly fetches the element in the second row (index 1) and the third column (index 2).

`*(a[1]+2)`

- `a[1]` yields the second row, which decays to a pointer to its first element (i.e., equivalent to `&a[1][0]`).
- Adding 2 moves the pointer two elements forward in that row.
- The dereference operator `*` then accesses the element at that position, which is `a[1][2]`.

`*(*(a+1)+2)`

- `a+1` moves the pointer from the first row to the second row.
- `*(a+1)` dereferences that pointer to yield the address of the first element of the second row (again, equivalent to `a[1]`).
- Adding 2 moves to the third element in that row, and the outer `*` fetches its value—again, `a[1][2]`.

`*(&a[0][0]+4+2)`

- `&a[0][0]` gets the address of the very first element of the array.
- Since the array is stored in contiguous memory, pointer arithmetic treats it as a flat sequence. Adding 4+2 (i.e., 6) moves the pointer to the 7th element in that sequence.
- If the layout of the array is such that the element `a[1][2]` is the 7th element (this is true, for example, if the row length is at least 3), then dereferencing this pointer retrieves `a[1][2]`.

5 Files

Taking an input from a file is not much different than taking input from a keyboard. All we do is:

- Declare an identifier of type `FILE` to hold the file pointer.
- Open the file (`fopen()`)
- Read the file
- Close the file (`fclose()`)

5.1 Declaring a file indentifier:

```
FILE *datafile;
```

The datafile is now a pointer we can associate with a file. The `FILE` type is defined in the `stdio.h` library.

5.2 Opening a file:

```
fileptr = fopen(char *filename, char *mode);
```

The `fopen()` is a function that is use for file opening. It takes two arguments: the name of a file and the mode it will operrate in. A file pointer is returned. The mode can be:

- `r` : read (open an existing file for reading)
- `w` : write (overrwite the file or create a new one)
- `a` : append (add to the end of the file)

5.3 Closing a File

```
fclose(fileptr);
```

Once a file is "closed" we can no longer read from it or write to it, unless we open it again. If we don't do this, the file will still be closed when the program terminates. But until then, no other program on the same node (computer) can work with the file, and it might but be fully written to storage

5.4 Reading from a file

```
fgets(char *str, int n, FILE *fileptr);
```

reads in a line of text from the fileptr stream and stores at most n characters in array sting. The mew line character is stored. If the string can't be read, because we have reached the end of the file, then NULL is returned.

```
fgetc(FILE *fileptr);
```

reads the next character in the file and stores it in the char variable c. If the end of the file has been reached, EOF is returned.

```
fscanf(FILE *fileptr, char *format, ...);
```

reads formatted input from the file. The format string is similar to the one used in `printf()`

5.5 Navigating a file

Each time a character is reader from the input stream, a counter associated with the stream is incremented.

```
rewind(FILE *fileptr);
```

sets the indicator to the start of the file

```
ftell(FILE *fileptr);
```

is used to check the current value of the file position indivator in the form of a long int.

```
fseek(FILE *fileptr, long offset, int place);
```

the value of offset is the amount the indicator will be changed by, while placce is one of:

- `SEEK_SET(0)` : the start of the file
- `SEEK_CUR(1)` : the current position in the file
- `SEEK_END(2)` : the end of the file

5.6 Writing to a file

To write to a file, we declare a file pointer:

```
FILE *outfile;
```

and open a new file in write mode:

```
outfile = fopen("myList.txt", "w");
```

to write to the file, we use one of:

- `fprintf(outfile, char *format, ...);` : works like `printf()` except its first argument is a file pointer
- `fputs(char *str, FILE *fileptr);` : writes the string str to the file pointed to by fileptr, without its trailing '\0' character. The string is not formatted.
- `fputc(int c, FILE *fileptr);` : writes the character c to the file pointed to by fileptr. The character is not formatted.

5.7 Issues concerining the use of files in C, we should be aware of:

- There are 6 modes a file can have: `r, w, a, r+, w+, a+`
- To open a binary file, also add a `b` to the mode. For example, `rb` opens a file for reading in binary mode.
- `freopen()` attaches a new file to an existing file pointer. It is used to redirect the standard input or output to a file. For example, `freopen("myList.txt", "w", stdout);` redirects the standard output to the file `myList.txt`. This means that any output that would normally go to the screen will now go to the file.
- `tmpfile()` open a tempory file in binary read/write and is automatically deleted when closed or when the program terminates
- `fflush(fileptr);` flushes the output buffer of the file pointer. This means that any data that has been buffered but not yet written to the file will be written to the file.
- `remove("myList.txt");` deletes the file `myList.txt` from the disk.
- `rename("myList.txt", "myList2.txt");` renames the file `myList.txt` to `myList2.txt`. If the new name already exists, it will be overwritten.
- `int feof(FILE *fileptr);` returns a non-zero value if the end of the file has been reached. It is used to check if we have reached the end of the file while reading it.

6 The Process

"A process is a running program."

The operating system gives the impression many programs are running at the same time, but in reality, only one program is running at a time. This is made possible by abstracting the the concept of a running program as a process.

Every process consists of

- The process text - the program code
- The program counter - the address of the next instruction to be executed
- The process stack - temporary data, local variables, function parameters, return addresses
- The data section - global variables

A process is not just a program - if two users run the same program at the same time, they create different processes. Each process has its own memory space, so they do not interfere with each other. A program is a passive entity, while a process is an active / dynamic entity.

A set of operations OS must apply to a process

- **Create** a new process (open a new program)
- **Terminate / Destory** a process (close a program)
- **Wait** or pause a process until some event occurs (e.g. waiting for user input)
- **Supsend and Resume** similar to wait but more explicit
- **Status** report info about a process (e.g. how much memory it is using, how long it has been running, etc.)

The state of a process

The state of a process is defined by the current activity of that process.

- **New** - the process is being created
- **Running** - instructions being executed
- **Block / Waiting** - waiting for some event to occur (e.g. I/O operation)
- **Ready** - waiting to be assigned to a processor
- **Terminated** - the process has finished executing

6.1 Process Creation

A parent createas a child process, which create other child processes, forming a tree of processes. After a parent creates a child process it may:

- Execute concurrently with the child process
- Wait until the child process terminates to continue

The parent may share all, some or none of the resources with the child process (resources include memory space, open files, etc.).

6.1.1 Process Indentification Number (PID)

All processes have a unique identifier called a PID. If we create a child process in C, using a `fork()` a new process is created:

- The new process runs concurrently with the parent, unless we instruct it to `wait()`.
- The subproc (child) is given a copy of the parents memory space
- At the time of creation, the two processes are identical, except the `fork()` returns te child process' PID to the parent and 0 to the child.

In order to use this function, we must include the `unistd.h` library, which includes:

- `fork()` - creates a new process
- `getpid()` - returns the PID of the calling process
- `getppid()` - returns the PID of the parent process

Listing 8: Forking a process

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

int main(void)
{
    pid_t pid1, mypid;
    pid1 = fork();
    mypid = getpid();
    printf("I am %d\t", mypid);
    printf("Fork returned %d\n", pid1);
    return 0;
}

Output:
I am 7791. Fork returned 0
I am 7790. Fork returned 7791
```

`fork()`

The proptype for `fork()` is:

`pid_t fork(void);`

- It takes no arguments
- It really returns an `int`
- It returns -1 if the fork failed
- Otherwise, it returns the PID of the newly created child process to the parent process
- It returns 0 to the child process
- The child process is distinct from the parent (it gets its own copy of the parent's memory space)
- Both parent and child process run concurrently
- Starting from the `fork()` call, the parent and child process execute the same instruction set.

`pid_t getpid(void);`

Returns the value of the processes own PID

`pid_t getppid(void);`

Returns the value of the parent process' PID Since the parent and subproc (child) have copies of the same memory space and instruction set, `getpid()` and `getppid()` are useful for working out which is which.

6.1.2 Process Programing

Often we don't want the parent to continue running while the child is running, the results may be non-deterministic. For example:

Listing 9: Unpredictable a process

```
int main(void)
{
    int i;
    fork();
    srand(getpid()); // different seed for each process
    printf("Watch me (%d) count to 10: ", getpid());
    for (i=1; i<=10; i++)
    {
        sleep(rand()%2); // sleep for 0 or 1 seconds
        printf("%3d...", i);
        fflush(stdout);
    }
    printf("\n");
    return 0;
}

Output:
Watch me (11695) count to 10: 1... 2... 3...Watch me (11696) count
to 10: 1... 4... 2... 5... 6... 3... 7... 4... 8... 5...
9... 6... 10... 7... 8... 9... 10...
```

`pid_t wait(int *wstatus);`

It returns the PID of the child process the parent was waiting for and its status. Acall to the wait() function suspends the execution of the parent process until such time as the child process completes (or, at least, signals to the parent – more about that later)

Listing 10: Wait for a process

```
#include <sys/wait.h>
int main(void)
{
    pid_t pid1 = fork();
    srand(getpid()); // different seed for each process
    if (pid1 != 0) // Parent follows this path
        wait(NULL);
    printf("Watch me (%d) count to 10: ", getpid());
    for (int i = 1; i <= 10; i++) {
        sleep(rand() % 2); // sleep for 0 or 1 seconds
        printf("%3d...", i);
        fflush(stdout);
    }
    printf("\n");
    return 0;
}
```

Recall, a sub-proc will share the parents memory only in a sense that it recieves a copy. The child process can mimic the parents execution as in the examples above or its memory space may be overlaid with a new program/set of instructions.
Often when a cild process is created it is overlaid with another program. In C, this can be done with the exelclp() function. In the following example, the sub-procs memory space is overlaid the program text of the ls command. Again we will use the `wait()` function.

Listing 11: Overlaying a process using execlp

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

int main(void)
{
    pid_t pid1 = fork();
    if (pid1 == 0) {
        // child
        printf("This is process %d\t", getpid());
        printf("Here is a directory listing:\n");
        execlp("ls", "ls", NULL);
    } else {
        // parent
        int p = wait(NULL);
        printf("This is process %d\t", getpid());
        printf("Wait returned %d, so subproc %d is done.\n", p, pid1);
    }
    return 0;
}
```

The OS is responsible for de-allocating the resources of a process that has finished. It may also be responsible for terminating a process that is not responding. Process termination occurs when:

- The proc executes its last instruction and asks the operating system to delete it (`exit()`). At that time it will usualy:
  - Output data from subproc to parent (via `wait()`)
  - Have its resources de-allocated by the OS
- Parent terminates a child process because:
  - Sub-proc has exceeded allocated resources
  - The task assigned to the sub-proc is no longer needed
- The parent is existing and the Os does not allow subproc to continue if the parent is terminated

On unix system, when a process terminates, its subprocs are reparented (adopted) by the init process. The mechanism for this is through "signals" and the `kill()` function. The `kill()` function is used to send a signal to a process.

6.2 Signals

The `kill()` system call is an example of a signal - a form of communication from one process to another. These provide a facility for asynchronus event handling. Note - when the subproc sends the kill() signal to the parent, the subproc also terminates.
The `kill()` function can send other signals, but most (such as SIGABRT, SIGILL, SIGQUIT, SIGTERM) are just variants of SIGKILL. However there are signals that preform other taks:

- **SIGSTOP** - stops a process
- **SIGCONT** - continues a stopped process
- **SIGUSR1** - user defined signal 1
- **SIGUSR2** - user defined signal 2

```c
int kill(pid_t pid, int sig);
```

The `kill()` function takes two arguments: the PID of the process to send the signal to and the signal to send. The function returns 0 on success and -1 on failure. The `kill()` function can be used to send any signal to any process, including itself.
With the `signal()` function, we can send a signal that tells the process to preform a specific asction, when it recieves a `SIGUSR1` or `SIGUSR2` signal.

Listing 12: Signal handling

```c
void signal_handler1(int sig);

int main(void)
{
    pid_t subproc_pid = fork(); // New process will send signal to parent

    if (subproc_pid != 0) {
        // parent
        signal(SIGUSR1, signal_handler1); // What to do if I get signaled
        sleep(100); // Wait potentially a long time
        printf("%d got woken \n", getpid());
        kill(subproc_pid, SIGKILL); // tell watcher to terminate.
    } else {
        // child
        printf("I am the sub-proc (%d)\n", getpid());
        printf("Should %d signal parent (%d)? ('y'/'n')\n",
               getpid(), getppid());
        if (getchar() == 'y') {
            printf("Sending siguser 1 to parent\n");
            kill(getppid(), SIGUSR1);
        }
    }
    return 0;
}

void signal_handler1(int sig) {
    printf("Proc %d called signal_handler1() with signal %d\n",
           getpid(), sig);
}
```

What the process does after it has finished calling the signal hander function depends on the situation. In the example above, it stopped sleeping. However if it had been making a blocking call, such as taking input from the keyboard, or reading from a pipe, it returns to doing that.

6.3 IPC - Inter Process Communication

Def. **Process Communication Types**

Two processes that are executing on the same computer may be either:

- **Independent** : Cannot affect or be affected by each other
- **Co-operating** : Can be affected by the execution of eachother

We allow processes communication so we can faciliate:

- **Information sharing** - two procs might require acess to the same file
- **Modularity** - Different procs might be dedicated to different system functions
- **Convenience** - a user might be running an editor, spell checker and printer all for the same file
- **Computational Speed** - on a multiprocessing system tasks are sub-divided and executred conrurrently on different processors.

Communication between cooperative may be described in terms of the **producer consumer model**

Def. **Producer Consumer Model**

One process (editor) has information it wants to produce for consumption by another (printer). This may be done by a **mutual buffer**. Producer write to the buffer and consumer reads from it.

Concept: **Shared Buffer Types**

Shared buffers may be:

- **Un-bounded** : no size limit placed on the buffer. The producer may continue to produce and write to the buffer as long as it wants to.
- **Bounded** : strict size limit on the buffer. If full, producer must wait until consumer removes some data.

This may be implemnted **Physically** by a set of shared variables (shared memory addresses), the implementation of this is the responsibility of the programmer. It also may be inplemented **Logically** by Message passing using an Inter Process Communication. Such system is inplemented by the OS..

The IPC prodives a logical communcation link via a message passing facilitiy with two funadmental operations : **send message** and **recieve message**. Two standard forms of IPC are **Direct** and **Indirect** communication. Furthermore, IPC may also be **Symmetric** or **Asymmetric**.

6.3.1 Direct Communication

Each process must nomiate the process they want to communicate it. Pairs of cumminicting processes must know each others ID's in order to etablish a link. With this system a link is established by the two processes automatically, there are exactly two processes in the link and there can be at most one link between two procss

This is an example of **Symmetric addressing**. In the **Asymmetric addressing** model, the recipient listens out for any messages addressed to it (like making a phone call).

Disadvantages

- Processes must have details of eachother befoe they can communicate
- One link between processes only
- Only two processes can communicate at a time

6.3.2 Indirect Communication

The alternative is to have a number of ports (mailboxes) in the sytem. Each with a unique ID.

For procs to communicae they must know the name of the mailbox. More than two procs can share a mailbox . Two procs may share more than one mailbox

Disadvantages

- Mailbox is owned by only one proc. Several procs can write to it, but only one can read from it.
- Procs are owned only by the OS. Permissions are then granted to processes to create and delete mailbox and to send and receive messages.

6.4 `pipe()`

A means of communication, based around the `pipe()`, `write()`, `read()` functions that allows one process to send data to another. The data can be anything, but we will use examples of sending integers. This is an example of **Symmetric direct communication**.

- Use `pipe(fd)` to create a pipe. Where `fd` is an array of two integers.
- To put a message into the pripe, write to `name[1]`
- To read a message from the pipe, read from `name[0]`

Listing 13: Pipe example

```c
#include <unistd.h>  // For pipe(), fork(), read(), write()
#include <stdio.h>   // For printf()

int main(void)
{
    int childpid, parentpid, pipefd[2];
    pipe(pipefd);    // Create a pipe with two file descriptors:
                     // pipefd[0] for reading, pipefd[1] for writing
    childpid = fork(); // Create child process (returns child PID to parent, 0 to child)

    if (childpid != 0) // This is the parent process (fork returned child's PID)
    {
        parentpid = getpid();  // Get parent's process ID
        printf("I'm the Parent proc (%d); my child is: %d\n",
               getpid(), childpid);
        write(pipefd[1], &parentpid, 4); // Write parent's PID to the pipe. Child will be able to read this value
    }
    else // This is the child process (fork returned 0)
    {
        read(pipefd[0], &parentpid, 4); //Read into parentpid. This blocks until data is available
        printf("I'm the Child proc (%d); what I read from pipe: %d\n",
               getpid(), parentpid);
    }

    return 0;
}
```

6.5 Intro to threads

So far, we've assumed that every process has its own program counter (PC), which tells the OS where it is in its instruction set (think line of program currently executing). So even when we duplicate using `fork()` the new subproc has its own PC, even though they have the same value as the parents PC.

However, modern OS's allow for threads: single process that can have mulitple PC's.

***Concept:* Why threads are useful**

- The process may need to preform lots of operations at the same time (on different cores) on the same data - i.e. adding vectors
- Processes have to preform different operation that run at different speeds, such as adding vectors (fast) and waiting for user input (slow)

But, with subprocs made with `fork()` memory is not really shared, so the advantage of being able to do things at once is lost by the need to communicate.

6.5.1 Single Point of Exuction vs Multiple Points of Execution

Instead of our classic view of a single point of execution withing a program, a multi-threaded program has more than one point of execution. Each thread is very much like a seperate process, except for **one difference:** they share the same address and space and thus can acess the same data.

***Def.* Thread**

A thread (or lightweight process) is a basic unit of CPU utilization. Rather than a process being dedicated to one sequential list of tasks it may be broken into threads. These consists a set of distinct:

- **Program Counter** - the next instruction to be executed
- **Register set** - operands for CPU instructions
- **Stack** - temporary variables, etc.

***Def.* Task**

A task is a collection of threads that belong to the same process and share code section, data section, and operating-system resources.

***Concept:* Why use threads over processes**

- **Parallization** - Each thread can execute on a different processor core.
- **Responsiveness** - A part of a process may continue working, even if another part is blocked, i.e. waiting for an I/O operation
- **Resource Sharing** - For exampl we can have several hundred threads running at the same time. If they were all processes, we would have to allocate memory for each one. This is not the case with threads, as they share the same address space.
- **Economy** - Thread creation is faster than process creation and context switching is faster
- **Efficiency** - Threads belong to the same proc share common memory space and do not need support from the OS to communicate.

6.5.2 Two types of threads - User and Kernel

Thread Type	User Threads	Kernel Threads
Implementation	Implemented by a thread library at compiler/library level above the OS kernel	Created, managed and scheduled by the operating system kernel
Advantages	Quick to create and destroy since system calls are not required	Do not suffer from blocking problems; if one thread blocks, the kernel can schedule another thread from the same application
Disadvantages	If the kernel is not threaded and one thread makes a blocking system call, the entire process will be blocked	Slower to manipulate compared to user threads due to system call overhead

Table 5: Comparison of User and Kernel Threads

6.6 Constasting subprocs and threads

These programs will have a globl int `GlobalVar`, initalised as 0. First we'll see that subprocces that increment GlobalVar do not change the parents value.

Listing 14: Subprocs and GlobalVar

```c
    /* forkmem.c: forks don't share memory */
#include <stdio.h>
#include <assert.h>
#include <pthread.h> // not used here, but needed in Example 2
#include <unistd.h>

int val = 0; // Global variable; will check if really shared

int main(void)
{
    printf("--------------------------------------\n");
    printf(" An example showing that fork()'ed \n");
    printf(" child process has a copy of its parent's \n");
    printf("START:\tThis is parent (PID=%d); val=%d\n",
           getpid(), val);

    int pid = fork();
    if (pid == 0) { // child
        val++;
        printf("\tThis is child (PID=%d); val=%d\n",
               getpid(), val);
    } else { // parent
        sleep(1); // make sure the subprocess does its thing first
        printf("END:\tThis is parent (PID=%d); val=%d\n",
               getpid(), val);
    }

    return 0;

    Output:
    "An example showing that fork()ed subprocess has a copy of its parents memory, but it is not shared.
    START: This is parent (PID=24875); val=0
    This is child (PID=24876); val=1
    END: This is parent (PID=24875); val=0"
}
```

On Linux, Mac and related systems, in C we can create threads called `pthreads` (POSIX threads) which are defined in the `pthread.h` header. We will use two functions:

- `pthread_create()` - used for creating thread and takes four arguments.
  - ID of thread process
  - Thread attributes (we'll ignore and use NULL)
  - A function called by thread when created
  - Argument to pass to the function.
- `pthread_join()` - tells parent to wait for a thread to finish

Listing 15: Threads and GlobalVar

```c
    /* pthreadmem.c: threads share memory */
    #include <stdio.h>
```

```
#include <assert.h>
#include <pthread.h> // Where the pthread functions are defined
#include <unistd.h>

int val = 0; // Global variable; will check if really shared

void *mythread(void *arg) // This is the function the thread will call
{
    val++;
    printf("\tThread with (PID=%d); val=%d\n",
        getpid(), val);
    return(NULL);
}

int main(void)
{
    printf("------------------------------------------\n");
    printf(" An example showing that a pthread \n");
    printf(" shares its parent's memory \n");
    printf("START:\tParent with (PID=%d); val=%d\n",
        getpid(), val);

    pthread_t p1; // declare the thread's ID
    pthread_create(&p1, NULL, mythread, "N"); // create the thread
    pthread_join(p1, NULL); // wait for it to finish

    printf("END:\tParent with (PID=%d); val=%d\n",
        getpid(), val);
    return 0;

    Output:
    "An example showing that a pthread shares its parent's memory
    START: Parent with (PID=25149); val=0
    Thread with (PID=25149); val=1
    END: Parent with (PID=25149); val=1"
}
```

7 Scheduling Processes

CPU scheduling is the basis of multipogrammed operating systems.

***Concept:* Underling concepts of CPU scheduling**

- **Goal**: Maximum CPU utilization, there should be something running at any given time.
- **Based around**: CPU burst or I/O burst Cycle. A running process alternates between executing instructions (CPU burst) and waiting for I/O operations to complete (I/O burst).
- **Depends on**: the burst distribution, given a proc predominantly concerned with computation or I/O. Also a proc usually begins with a long CPU burst and alomst always ends with a CPU burst.

The CPU scheduler selects from among the processes in memory that are ready to execute and allocates CPU to one of them. This occurs when a process:

- Switches from running → waiting state (non preemptive)
- Terminates (non preemptive)
- Switches from running → ready state (preemptive)
- Switches from waiting → ready state (preemptive)

If a scheduler uses only non-premptive methods, once a process is allocated to the CPU it will reamin there until it terminates or changes states.

For premptive methods, the scheduler must have an algorithm for deicing when control of CPU must be given to another process.

7.1 The Dispatcher

The dispatcher is a module that gives control of the CPU process selected by the short-term scheduler. It is responsible for:

- Switching context
- Switching to user mode

- Jumping to the proper location in the user program to restart that program

This gives rise to dispatch latency, the time it takes for the dispatcher to stop one process and start another. This is a very small amount of time, but it is important to keep it as low as possible.

7.2 Scheduling Algorithms

- First Come First Served (FCFS) - the process that arrives first is executed first. This is a non-preemptive algorithm.
- Shortest Job First (SJF) - the process with the shortest CPU burst time is executed first. This is a non-preemptive algorithm.
- Shortest Time To Completion (STC) - the process with the shortest time to completion is executed first. This is a preemptive algorithm.
- Round Robin (RR) - each process is given a fixed time slice (quantum) to execute. If the process does not finish within the time slice, it is preempted and the next process is executed. This is a preemptive algorithm.

We'll consider a few examples when describing these, and assume each proc has a single CPU burst, measured in milliseconds. We would like to compare and determine which is best, but there are many choices of what quallifies as best. Here are a few metrics.

- Turnaround time - the time that elapses between when proc arrives in system and when it is completed.
- Wait time - time between when a proc arrives and when it comples, that it is spent doing nothing.
- Reponse time - time between when proc arrives and when it first starts executing.