CS211: Programing For Operating Systems

Robert Davidson

Contents

Useful headers

- `stdio.h` : Standard input/output library
- `stdlib.h` : Standard library
- `string.h` : String manipulation functions (`strncpy`, `strcat`, `strcmp`, `strstr`, `strlen`)
- `unistd.h` : Standard library for system calls (`fork`, `getpid`, `getppid`)

0.1 Intro to C

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.

0.2 Hello World

Listing 1: Hello World in C

```
1    #include <stdio.h>
2    int main(){
3        printf("Hello World\n");
4        return 0;
5    }
6
```

- **Line 1** : `#include <stdio.h>` is a preprocessor directive that tells the compiler to include the standard input/output library. This library contains the `printf` function.
- In C almost every line it either preprocessor directive, variable declaration, or a function call.
- C uses curly braces to delimit blocks of code and semicolons to terminate statements.
- **Line 4**: In our case, we assume main is called by the Operating System, so return 0 is used to indicate that the program has run successfully.

0.3 Variables

In C all variables must be declared before they are used. The declaration should have a type; telling the compiler what sort of data the variable will hold. The types of variables are:

- **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)

We can Also declare arrays as follows:

Listing 2: Declaring Arrays

```
1    int arr[5]; // Array of 5 integers
2    char name[10]; // Array of 10 characters
```

To access the first element of arr we can do `arr[0]`

0.4 An Example

Listing 3: Example of Variables

```
1    int d=-101;
2    float f=1.23456;
3    char c='a';
4    printf("Values of d, f, c are: %d, %f, %c\n", d, f ,c );
```

Explanation: In this case, `%d` is a placeholder for an integer, `%f` is a placeholder for a float, and `%c` is a placeholder for a character.

0.5 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

0.6 Control Structure

Listing 4: If-Else

```
1     int a = 10;
2     if(a > 10){
3             printf("a is greater than 10\n");
4         }else if(a == 10){
5             printf("a is equal to 10\n");
6         }else{
7     printf("a is less than 10\n");
8     }
9
```

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

Listing 5: Complex If-Else

```
1     if(a > 10 && a < 20){
2         printf("a is between 10 and 20\n");
3     }
4
```

0.6.1 For loop

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

Listing 6: Print numbers from 0 to 9

```
1 for(int i = 0; i < 10; i++){
2     printf("i is %d\n", i);
3 }
```

0.6.2 While loop

`while(expression){...}`

Listing 7: Print numbers from 0 to 9

```
1 int i = 0;
2 while(i < 10){
3     printf("i is %d\n", i);
4     i++;
5 }
```

0.6.3 Do While loop

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

Listing 8: Print numbers from 0 to 9

```
1 int i = 0;
2 do{
3     printf("i is %d\n", i);
4     i++;
5 }while(i < 10);
```

0.7 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

0.8 Input

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

Listing 9: Reading an integer

```
1     int number;
2     char letter;
3     printf("Enter a number and a char: ");
4     scanf("%d %c", &number, &letter);
5
6     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 10: Check for no input

```
1     int number;
2     printf("Enter a number between 1 and 30: ");
3     scanf("%d", &number);
4
5     while ((number<1) || (number>30))
6     {
7         printf("Invalid number. Please enter a number between 1 and 30: ");
8         scanf("%d", &number);
9     }
```

0.9 Functions

Definition 0.1: Protype and Definition

A **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.

0.10 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:

```
1 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.

Concept 0.1: Pointers

- if `i` is a variable, then `&i` is a pointer to the location in memory where the value of `i` is stored. If a memory address is stored in the variable `p`, then `*p` is the value stored at that address.
- If a memory address is stored in the variable `p`, then `*p` is the value stored at that address.

0.11 Characters

Concept 0.2: Character representation

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

Printing characters:

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

Reading characters:

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

0.12 Strings

Concept 0.3: String representation

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.

0.12.1 String Functions

Useful functions defined in `string.h`:

```
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.

```
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.

```
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

```
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.

```
int strlen(char *str)
```

Returns the length of the string `str` (not including the null character).

1.11 Strings (Continued)

0.12.2 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.

0.12.3 String Input

- `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.

0.13 Arrays

0.14 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)`

0.15 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 `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]`.

0.16 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()`)

0.17 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.

0.18 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)

0.19 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

0.20 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()`

0.21 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

0.22 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.

0.23 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.

0.24 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.

Concept 0.4: Every process has

- **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.

Concept 0.5: Operations OS must perform on 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.)

Concept 0.6: States of a 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

0.25 Process Creation

A parent creates a child process. The child process can create its own child, 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, none or some of its resources with the child process (memory space, open files, etc.)

0.26 Process Identifcation Number

ll 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 `fork()` , 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

0.26.1 `unistd.h` functions

```
pid_t fork(void);
```

- It takes no arguments
- It 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.

0.27 Process Programming

Often we don't want the parent to continue running while the child is running, the results may be non-deterministic. 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 or its memory space may be overlaid with a new program/set of instructions.

Often when a child 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.

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.

Concept 0.7: Processes are terminated 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 exiting and the OS does not allow subproc to continue if the parent is terminated

0.28 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.

Concept 0.8: Types of Signals

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

The kill function can take two arguments:

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

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.

0.29 Inter Process Communication

Inter process communication (IPC) is a mechanism that allows processes to communicate with each other. This is important because processes may need to share data or synchronize their actions

Concept 0.9: Types of Process Communication

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

Concept 0.10: Why we allow IPC

- **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.

Definition 0.2: Producer Consumer Model

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

Concept 0.11: Shared Buffer Types

- **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.

Concept 0.12: How we implement shared buffers

- **Physically** by a set of shared variables (shared memory addresses), the implementation of this is the responsibility of the programmer.
- **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**.

0.29.1 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

0.29.2 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.

0.30 `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**.

0.31 Theads

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 0.13: 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.

0.32 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.

Definition 0.3: 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.

Definition 0.4: 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 0.14: 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.

0.33 Two types of threads - User and Kernel

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

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

0.34 Sheduling Algorithms

"Several process/jobs are available to be run. In what order should they be executed?

0.35 Metrics to Compare Algoritms

Concept 0.15: Metrics to Compare Algorithms

- **Turnaround time** - the total time taken to execute a process. It is the sum of the waiting time and the burst time.
- **Waiting time** - the total time a process has been waiting in the ready queue.
- **Response time** - the time taken from when a request was submitted until the first response is produced.

Algorithm	Advantages	Disadvantages
FIFO (First In First Out)	Simple to implement, only need a FIFO queue. It is non-preemptive	Has a long average weight time, and suffers from the **convoy effect** (short jobs wait for long jobs)
SJF (Shortest Job First)	Optimal - minimum average waiting time for a given set of processes. Is non pre-eemptive.	Cannot be emplyed in a realisting setting because we would be required to know the length of next CPU Burst before it happens
STCF (Shortest Time to Completion First)	"More optimal" than SJF when processes arrive at different times	Needs preempting
RR (Round Robin)	Fair, each process gets a fair share of the CPU. It is preemptive and simple to implement	Average waiting time is high, and it is not optimal for short jobs

0.36 RR - Round Robin

Definition 0.5: Round Robin

A preemptive version of FIFO. Each process is assigned a time slice (quantum) and is executed for that time. If it does not finish, it is put back in the queue and the next process is executed.

Example 0.1

Proc	Arrival Time	Burst Time
P_1	0	20
P_2	0	15
P_3	4	10
P_4	6	5

1. First-Come, First-Served (FCFS)

Order: $P_1 \to P_2 \to P_3 \to P_4$

- P_1: starts at 0, ends at 20
- P_2: starts at 20, ends at 35
- P_3: starts at 35, ends at 45
- P_4: starts at 45, ends at 50

$$\text{TAT} = \text{Completion Time} - \text{Arrival Time}$$
$$\text{WT} = \text{TAT} - \text{Burst Time}$$
$$\text{RT} = \text{Start Time} - \text{Arrival Time}$$

Process	CT	TAT	WT	RT
P_1	20	20	0	0
P_2	35	35	20	20
P_3	45	41	31	31
P_4	50	44	39	39

$$\text{Average TAT} = \frac{20+35+41+44}{4} = 35$$
$$\text{Average WT} = \frac{0+20+31+39}{4} = 22.5$$
$$\text{Average RT} = \frac{0+20+31+39}{4} = 22.5$$

2. Shortest Job First (SJF, Non-preemptive)

- At time 0: P_1, P_2 ready $\to$ pick P_2
- At time 15: P_1, P_3 ready $\to$ pick P_3
- At time 25: P_1, P_4 ready $\to$ pick P_4
- At time 30: P_1 runs

- P_2: start 0, end 15
- P_3: start 15, end 25
- P_4: start 25, end 30
- P_1: start 30, end 50

Process	CT	TAT	WT	RT
P_2	15	15	0	0
P_3	25	21	11	11
P_4	30	24	19	19
P_1	50	50	30	30

$$\text{Average TAT} = \frac{15+21+24+50}{4} = 27.5$$
$$\text{Average WT} = \frac{0+11+19+30}{4} = 15$$
$$\text{Average RT} = \frac{0+11+19+30}{4} = 15$$

3. Round Robin (RR), Time Quantum = 10

Execution sequence:

- 0-10: P_1 (rem 10)
- 10-20: P_2 (rem 5)
- 20-30: P_3 (done)
- 30-35: P_4 (done)
- 35-45: P_1 (done)
- 45-50: P_2 (done)

Response times:

- P_1: 0
- P_2: 10
- P_3: 20
- P_4: 30

Process	CT	TAT	WT	RT
P_1	45	45	25	0
P_2	50	50	35	10
P_3	30	26	16	20
P_4	35	29	24	30

$$\text{Average TAT} = \frac{45+50+26+29}{4} = 37.5$$
$$\text{Average WT} = \frac{25+35+16+24}{4} = 25$$
$$\text{Average RT} = \frac{0+10+20+30}{4} = 15$$

4. Shortest Time to Completion First (STCF)

Execution order:

- 0-4: P_2 runs (rem 11)
- 4-6: P_3 runs (rem 8)
- 6-11: P_4 runs (done)
- 11-21: P_3 runs (done)
- 21-35: P_2 resumes (done)
- 35-55: P_1 runs (done)

Response times:

- P_1: 21
- P_2: 0
- P_3: 4
- P_4: 6

Process	CT	TAT	WT	RT
P_1	55	55	35	21
P_2	35	35	20	0
P_3	21	17	7	4
P_4	11	5	0	6

$$\text{Average TAT} = \frac{55+35+17+5}{4} = 28$$
$$\text{Average WT} = \frac{35+20+7+0}{4} = 15.5$$
$$\text{Average RT} = \frac{21+0+4+6}{4} = 7.75$$

0.37 Concurrency

A cooperating process is one that can affect or be affected by another process that is executing on the system.

Threads are prime examples of this: we can think of them as a single process with multiple points of execution. They share program code and, crucially, data.

In this section, we consider the problems that occur then one or more threads try to access the same data, and we look at potential solutions.

A classic data inconsistency problem is the so-called "Race Condition",

A Race Condition (also called a data race) is one where the result depends on the order in which instructions are executed.

For a single-thread process, this is predetermined.

But for multi-threaded processes, we do not have control over the order in which individual threads execute their instructions.

Example 0.2

Two cooperating process called P_1 and P_2 share the variable count. At various times during execution either may increment or decrement count.

The machine usually implements an increment as follows:

- 1. Load the value of count into a register: R1 = count
- 2. Add 1 to the contents of the register: R1 = R1 + 1
- 3. Overwrite the contents of count with the contents of the register: count = R1.

A decrement would be implemented as

- 1. load the value of count into a register: R2 = count
- 2. subtract 1 from the contents of the reg: R2 = R2 - 1
- 3. save the contents of the register as count: count = R2

Suppose the value of count is 5. If P1 executes an increment and P2 executes a decrement, then the value of count should still be 5. Unless the individual operations happen in the following order...

P_1 executes	$\text{REG}_1 = \text{count}$	$\text{REG}_1 = 5$
P_1 executes	$\text{REG}_1 = \text{REG}_1 + 1$	$\text{REG}_1 = 6$
P_2 executes	$\text{REG}_2 = \text{count}$	$\text{REG}_2 = 5$
P_2 executes	$\text{REG}_2 = \text{REG}_2 - 1$	$\text{REG}_2 = 4$
P_1 executes	$\text{count} = \text{REG}_1$	$\text{count} = 6$
P_2 executes	$\text{count} = \text{REG}_2$	$\text{count} = 4$

We arrive at the wrong state because we allowed both threads to manipulate the variable count at the same time. Since the outcome depends on the order in which each operation takes place, we have a race condition.

0.38 Critical Sections

Definition 0.6: A

ritical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

The example given above shows that multiple threads executing the same code can result in a race condition, that is an example of a critical section.
To resolve this, we would like to enforce mutual exclusion: This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

One possible solution is to make the operation "atomic" (or indivisible). This is, the critical section is executed as though it were a single operation, and so impossible to interrupt.

In a realistic setting, that is not possible for all race conditions. But, as we will see, the use of some

atomic operations can help us solve the larger problem, by creating locks.

So now we know we would like to execute a series of instructions atomically. But, in general, on a multiprocessor system, we can't. But what we can do is create a lock which we put around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction. For the lock approach to work, the following 3 conditions must be satisfied:

- **Mutual Exclusion**: If process T_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Fairness/Progress:** If there are some processes that wish to enter the critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.
- **Performance/Bounded Waiting:** after a process has made a request to enter its critical section and before that request is granted, then must be a bound (i.e., a limit) on the number of times other processes are allowed to enter their critical sections.

Approaches to this: Interupt Suspension Suppose a process is in its critical section. If it cannot be preempted then data consistency should be maintained. On a single processor system, the problem could be solved by disabling interrupts while a shared variable is begin modified. However, such a method is not feasible on a multiproc system: large over-heads would be incurred informing all processes that interrupts are dis-allowed. **Automatic Instructions** The processor has facilities to swap the contents of two words (in memory), or test and change the contents of a word automatically – i.e., as a single instruction. There are other hardware and software solutions to synchronization problems. The most important, perhaps, is a tool known as a semaphore

0.39 Semaphores

a Semaphore S is an integer variable that can only be accessed via one of two operations: Test/sem wait P(S), and Increment/sem post V (S).

Listing 11: Test or Sem_wait

```
1  P(S)
2  while (S <= 0) {
3      wait();
4  }
5  S--;
6
```

Listing 12: Test/Sem_wait

```
1  V(S)
2  S++;
3
```

These operations must be indivisible (or "atomic"). This is, when one process (or thread) modifies a semaphore value, no other process can modify it at the same time.

There are two types of semaphore:
1. Binary semaphores (locks): These are used to control access to a single resource, such as a memory location. If the resources is available then $S = 1$. Otherwise $S = 0$. When a process wants to access it,

- It calls the function $P(S)$
- Enters the critical section
- Calls $V(S)$ to release the resource when it exists the critical section

2. General (or counting) semaphores These are used to control access to a pool consisting of a finite number of identical resources. Say there are 5 units available. The S is initialised to 5. Whenever a process requests the resource, it calls $P(S)$ and decrements the value of S. If S reaches 0 then the next process that requests that resource must wait until another frees it by running $V(S)$.

0.40 Deadlock and Starvation

Deadlock is when two or more procs are waiting indefinitely for an event that can only be caused by one of the waiting processes. E.g., all are stuck in the wait() loop of the P() function.

Deadlock can arise if four conditions hold simultaneously 1

- Mutual exclusion: only one process can have access to a particular resource at any given time.
- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.

- No preemption: a resource can only be released voluntarily by the process holding it, after that process has completed its task.
- Circular wait: there exists a set P1, P2, . . . , Pn of waiting processes such that
  - P1 is waiting for a resource that is held by P2,
  - P2 is waiting for a resource that is held by P3, ..., ∘
  - Pn-1, is waiting for a resource that is held by Pn which in turn is waiting for P_1

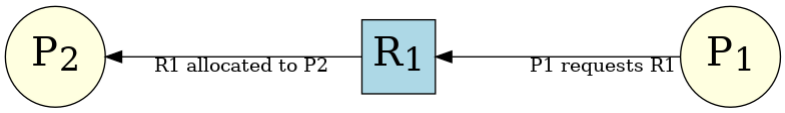
0.41 Resource Allocation Graphs

Deadlocks may be described in using a directed graph called a resource allocation graph. This graph has two sets of vertices:

- **Processes** - $P_1, P_2, \ldots, P_n$
- **Resources** - $R_1, R_2, \ldots, R_m$

And Edges:

- from $P_j \rightarrow R_k$ if process j as requested resource k but not yet been allocated to it
- from $R_k \rightarrow P_j$ if a resource k has been allocated to process j and not yet released



Example 0.3

A system has m = 2 resources, R1 and R2, and n = 2 processes, P1 and P2.

- P1 has been allocated R1, and is requesting R2.
- P2 has been allocated R2, and is requesting R1.

Draw the resource allocation graph for the scenario. Does the system reach deadlock?

Example 0.4

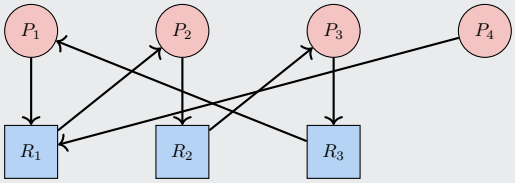
A system has m = 2 resources, R1 and R2, and n = 2 processes, P1 and P2.

- P1 is requesting R1.
- P2 has been allocated R2, and is requesting R1.

Draw the resource allocation graph for the scenario. Does the system reach deadlock?

Example 0.5

Consider the Resource Allocation Graph below. Does it represents a deadlocked state?



- If there are no cycles, there is no deadlock
- If there is deadlock, there must be a cycle
- If there is a cycle, there may be deadlock
- If each resource has only one instance, and there is a cycle, then

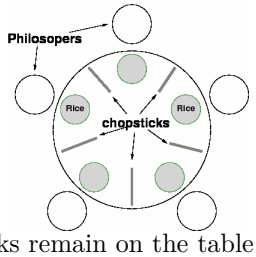
0.42 The Dining Philosophers

Starvation occurs when a particular process is always waiting for a particular semaphore to become available.

The ideas of Deadlock and Starvation are exhibited in the classic synchronization problem: The Dining Philosophers Problem.

- There are five philosophers seated at a round table.
- Each has a bowl of rice in front of them and a chopstick to their left and right.
- They spend their day alternating between eating and thinking.
- However there are only five chopsticks...

If a philosopher is hungry, they will try to pick up the chopstick to the left and then the chopstick to the right. If they manages to do this they will eat for a while before putting down both and thinking for a while. However, if they pick up one, they will not let go of it until they can pick up the second and eat.



Suppose each of the picks up the chopstick to their left. No chopsticks remain on the table so we reach a state of deadlock. The challenge is to find a solution so that.

- Deadlock does not occur
- Neither does starvation - where one philosopher doesn't eat

Recall the "Dining Philosophers Problem" as a model for process synchronisation. For each of the following statements, state if it is true or false, and explain your answer.

- If there are five diners, and five chopsticks, and all diners pick up a chopstick at the same time, the system will be in deadlock.
- If there are only four diners, and five chopsticks, the system cannot reach a deadlocked state.
- If there are six diners, and five chopsticks, the system cannot reach a deadlocked state.

In general operating systems take one of three approaches to deal with deadlock:

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.

In Case 1, there are two possibilities:

- Prevention: we ensure at least one of the four necessary conditions
- Deadlock avoidance: where the OS uses a priori information about

In Case 2, the OS must have mechanisms for first detecting deadlock and then dealing with it. Recall that a Semaphore, S, is an integer variable that can only be accessed via one of two operations:

- Test/sem_wait P(S), and
- Increment/sem_post, V(S)

It is "special" in the sense that the two operations on it can't be interrupted. In Lab 6, we will implement a solution to a Race Condition when we have multiple fork'ed subprocesses, problem by implementing a semaphore via a pipe. However, POSIX systems already come with a semaphore solution when working with pthreads. To use it:

- sem_t S; declares S to be a semaphore.
- sem_init(&S, 0, 1);
- sem_wait(&S) checks if it is available and, if it is, grabs it.
- sem_post(&S) releases it, causing awaiting thread to wake up and try taking it.

0.43 Deadlock Detection

Detection of deadlock is an important, but difficult problem. Using ideas like the Resource Allocation Graph, the system's needs can be represented mathematically, and then a deadlock state checked for

Idea Suppose that a system has n processes, and a total of m resources that it can allocate. Resources can only be requested or released one at a time. Then the system is Deadlock free if the following conditions hold:

- each process requires at least 1 resource and at most m.
- the sum of all their requirements is less than m + n.

0.44 Deadlock Avoidance

Banker's Algorithm
Only allocate resources to a process if you can allocate all the resources it requests. In particular we need to know:

- The number of resources the system has.
- The number currently allocated to each process;
- The maximum that any process might request.

With this information, it should be possible to ensure that a circular wait condition does not hold. To understand this, we need to concepts of safe states and safe sequences.

- A resources-allocation state is the number of available and allocated resources, and the maximum demands of processes.
- A state is safe if the system can allocate resources to each process (eventually), and still avoid deadlock.
- A safe sequence is a sequence of processes such that their resource requests can be granted, in order, with no process having to wait indefinitely.

Example
Suppose we have a system with

- three resource types, A, B and C. There are in total 10 instances of
- 5 processes, P0 , P1, . . . , P4.

	Total Reqs			**Current Alloc**		
	A	B	C	A	B	C
P_0	7	5	3	0	1	0
P_1	3	2	2	2	0	0
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

Table 5: Resource Allocation Table