

# **CS211: Programing For Operating Systems**

Robert Davidson

# Contents

0.1	Intro to C . . . . .	5
0.2	Hello World . . . . .	5
0.3	Variables . . . . .	5
0.4	An Example . . . . .	5
0.5	Operators . . . . .	6
0.6	Control Structure . . . . .	6
0.6.1	For loop . . . . .	6
0.6.2	While loop . . . . .	7
0.6.3	Do While loop . . . . .	7
0.7	Output . . . . .	7
0.8	Input . . . . .	8
0.9	Functions . . . . .	8
0.10	Call-by-Value and Pointers . . . . .	8
0.11	Characters . . . . .	9
0.12	Strings . . . . .	9
0.12.1	String Functions . . . . .	9
0.12.2	String Output . . . . .	10
0.12.3	String Input . . . . .	10
0.13	Arrays . . . . .	10
0.14	Arrays . . . . .	10
0.15	Explanation of Array Element Access in C . . . . .	11
0.16	Files . . . . .	11
0.17	Declaring a file identifier: . . . . .	11
0.18	Opening a file: . . . . .	11
0.19	Closing a File . . . . .	11
0.20	Reading from a file . . . . .	12
0.21	Navigating a file . . . . .	12
0.22	Writing to a file . . . . .	12
0.23	Issues concerning the use of files in C, we should be aware of: . . . . .	13
0.24	The Process . . . . .	13
0.25	Process Creation . . . . .	14
0.26	Process Identification Number . . . . .	14
0.26.1	unistd.h functions . . . . .	14
0.27	Process Programming . . . . .	15
0.28	Signals . . . . .	15
0.29	Inter Process Communication . . . . .	16
0.29.1	Direct Communication . . . . .	17
0.29.2	Indirect Communication . . . . .	17
0.30	pipe() . . . . .	17
0.31	Threads . . . . .	17
0.32	Single Point of Exuction vs Multiple Points of Execution . . . . .	18
0.33	Two types of threads - User and Kernel . . . . .	18
0.34	Sheduling Algorithms . . . . .	18
0.35	Metrics to Compare Algoritms . . . . .	19
0.36	RR - Round Robin . . . . .	19
0.37	Concurrency . . . . .	21
0.38	Critical Sections . . . . .	21
0.39	Semaphores . . . . .	22
0.40	Deadlock and Starvation . . . . .	23
0.41	Resource Allocation Graphs . . . . .	23
0.42	The Dining Philosophers . . . . .	24
0.43	Deadlock Detection . . . . .	25
0.44	Deadlock Avoidance . . . . .	25
0.45	Memory Management . . . . .	25
0.45.1	Summary of main ideas . . . . .	25
0.46	Memory and Storage Management . . . . .	26

0.47	Contiguous Memory Allocation . . . . .	26
0.48	Paging . . . . .	27
0.49	Basic idea . . . . .	27
0.50	Address Translation Scheme . . . . .	27
0.51	Basic ideas . . . . .	28
0.52	What happens if there is no free frame? . . . . .	28
0.52.1	Page replacement . . . . .	28
0.53	Exercise . . . . .	29
0.54	Memory management in C . . . . .	29
0.55	Dynamic memory allocation . . . . .	31
<b>1</b>	<b>Module Review</b>	<b>32</b>

## 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 is either a 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 operators, && 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
<code>\a</code>	Produces a beep or flash
<code>\b</code>	Moves cursor to last column of previous line
<code>\f</code>	Moves cursor to start of next page
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\v</code>	Vertical tab
<code>\\</code>	Prints a backslash
<code>\'</code>	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
<code>%c</code>	Single character (char)
<code>%d</code> or <code>%i</code>	Decimal integer (int)
<code>%e</code> or <code>%E</code>	Floating-point (scientific notation)
<code>%f</code>	Floating-point value (float)
<code>%g</code> or <code>%G</code>	Same as <code>%e/%E</code> or <code>%f</code> , whichever is shorter
<code>%s</code>	String (char array)
<code>%u</code>	Unsigned int
<code>%x</code>	Hexadecimal integer
<code>%p</code>	Pointer (memory address)
<code>%%</code>	Prints the <code>%</code> character

## 0.8 Input

`scanf()` reads input from standard 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 successful 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 reading 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: Prototype and Definition

**A Prototype :** 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 it is important to distinguish 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 distinguishing features of C is that 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 *strncpy(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 characters 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 and returns an integer:

- 0 if the strings are equal
- A negative integer if `str1` comes first alphabetically
- A positive 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 the 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  $3 \times 4$  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 creates 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 means if `a[] []` is declared as a two-dimensional  $3 \times 4$  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 identifier:

```
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 used for file opening. It takes two arguments: the name of a file and the mode it will operate in. A file pointer is returned. The mode can be:

- `r` : read (open an existing file for reading)
- `w` : write (overwrite 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 not 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 `str`. The new 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 read 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 indicator 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 `place` 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 concerning 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 temporary 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 Identification 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 receives 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 `execlp()` 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 usually:
  - 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 asynchronous 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 perform other tasks.

### 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 perform a specific action, when it receives 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 each other

### Concept 0.10: Why we allow IPC

- **Information sharing** - two procs might require access 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 executed concurrently 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 implemented by the OS.. The IPC provides a logical communication link via a message passing facility with two fundamental operations : **send message** and **receive message**.



### 0.29.1 Direct Communication

Each process must nominate the process they want to communicate it. Pairs of communicating processes must know each others ID's in order to establish 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 procs

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 each other before 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 system. Each with a unique ID.

For procs to communicate 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 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 subprocess 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 multiple PC's.

#### Concept 0.13: Why threads are useful

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

But, with subprocesses 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 Execution 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 access 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 **pthread**s (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 Algorithms

### 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 <b>convoy effect</b> (short jobs wait for long jobs)
SJF (Shortest Job First)	Optimal - minimum average waiting time for a given set of processes. Is non pre-emptive.	Cannot be employed 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:  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

- $P_1$ : 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

TAT = Completion Time – Arrival Time

WT = TAT – Burst Time

RT = Start Time – 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  $\rightarrow$  pick  $P_2$
- At time 15:  $P_1, P_3$  ready  $\rightarrow$  pick  $P_3$
- At time 25:  $P_1, P_4$  ready  $\rightarrow$  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:  $R1 = \text{count}$
- 2. Add 1 to the contents of the register:  $R1 = R1 + 1$
- 3. Overwrite the contents of count with the contents of the register:  $\text{count} = R1$ .

A decrement would be implemented as

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

Suppose the value of count is 5. If  $P_1$  executes an increment and  $P_2$  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 critical 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:

- **Interrupt 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 being 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

### Definition 0.7

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.

#### 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 resource 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

### Definition 0.8

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

- 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  $P_1, P_2, \dots, P_n$  of waiting processes such that
  - $P_1$  is waiting for a resource that is held by  $P_2$ ,
  - $P_2$  is waiting for a resource that is held by  $P_3, \dots$ ,
  - $P_{n-1}$ , is waiting for a resource that is held by  $P_n$  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.

### Conditions for Deadlock:

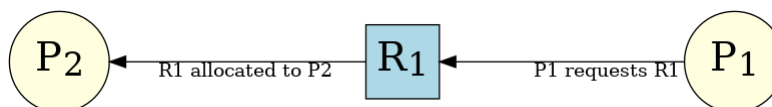
- If there are no cycles, there is no deadlock
- If there is deadlock, there must be a cycle
- If there is a cycle, there may be deadlock
- If each resource has only one instance, and there is a cycle, then there is a deadlock

The resource allocation graph has two sets of vertices:

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

And two sets of edges:

- from  $P_j \rightarrow R_k$  if process  $j$  has 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,  $R_1$  and  $R_2$ , and  $n = 2$  processes,  $P_1$  and  $P_2$ .

- $P_1$  has been allocated  $R_1$ , and is requesting  $R_2$ .
- $P_2$  has been allocated  $R_2$ , and is requesting  $R_1$ .

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

### Example 0.4

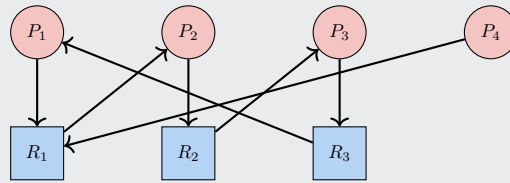
A system has  $m = 2$  resources,  $R_1$  and  $R_2$ , and  $n = 2$  processes,  $P_1$  and  $P_2$ .

- $P_1$  is requesting  $R_1$ .
- $P_2$  has been allocated  $R_2$ , and is requesting  $R_1$ .

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

### Example 0.5

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



## 0.42 The Dining Philosophers

### Definition 0.9

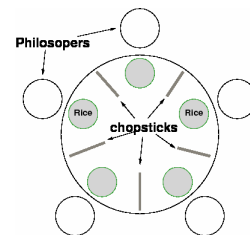
Starvation occurs when a particular process is always waiting for a particular semaphore to become available.

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 manage 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,  $P_0$ ,  $P_1$ , . . . ,  $P_4$ .

	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

## 0.45 Memory Management

### 0.45.1 Summary of main ideas

- "Storage" is how/where long-term data is maintained, particularly while a program is not running.

- During execution, a program (now, a process) has data stored in main memory, also known as "primary storage".
- Data belonging to a process is stored in "actual" memory: integrated circuits physically located on the device.
- The OS presents this location to the device as a logical address.
- The OS can use (backing) storage as temporary main memory, using "swapping".

## 0.46 Memory and Storage Management

### Definition 0.10

oughly, "storage" refers to how long-term data is maintained, particularly while a program is not running. Traditionally, this was on disks (drives), more recently it is on networked (cloud) devices.

During execution, a program (now, a process) has data stored in main memory. Management of both main memory and long-term storage require a high degree of sophistication on the part of the operating system.

A program must be brought into (main) memory and placed within a process for it to be executed. An Input queue is maintained, i.e., a collection of jobs on the disk that are waiting to be brought into memory for execution. User programs go through several steps before being executed, in particular they must be allocated space in memory.

Furthermore, recalling CPU Scheduling: the OS attempts to maximise CPU usage and responsiveness by switching between processes. This means that memory must be allocated to a (possibly large) number of processes at any given time.

**This leads to several complications:**

1. Since there are many processes running, the OS must have a fair way of allocating memory to all running processes.
2. As we've seen with fork()'ed processes, two processes can think they have access to the same memory address. So some virtualization is needed.
3. Most computers have different types of "memory", such as cache, RAM, and backing storage. The OS presents a single coherent view of all this to processes.
4. Physical memory is not necessarily large enough to accommodate all the processes, and so secondary storage must be used as "over-flow" memory.

A logical address is generated by the CPU; The OS uses this address. It is also referred to as a virtual address. A physical address is that what is used by the memory unit, i.e., the one loaded into the memory address register.

The OS takes care of mapping between the logical addresses and the physical addresses.

The virtual address is unique within a process. That is, no two variables belonging to a process can be stored at the same virtual address. But two different processes could use the same virtual address. You can check the virtual address used by a C program as follows:

```
int x;
fork(); // Now there are 2 processes.
x = getpid();
printf("[%d] x stores the value %d at %p\n",
       getpid(), x, &x);
```

The output I get is:

```
x is stores 21106 at 0x7fff1650fdc4
x is stores 21107 at 0x7fff1650fdc4
```

## 0.47 Contiguous Memory Allocation

In a multiprogramming environment, memory space is occupied by the operating system and a collection of user processes.

In order to maximise the degree of multiprogramming on the system, the OS will load as many processes into memory as there is space for. When new processes arrive in the input queue, it picks the first one and loads it into memory unless there is not enough room to accommodate it ("Banker's Algorithm"). In this case, it may search through to processes in the Input queue and load the first one for which there is a large enough space to accommodate.

When user processes terminate, memory "holes" are created. The operating system must allocate one of these holes to a new process that is starting. If the hole is too large, only part of it is allocated and a new smaller hole is created. Also, when a process terminates, if its space is adjacent to a free hole, they are amalgamated to create a larger one.

The scheduler then checks if the new hole is large enough to accommodate the next job in the Input queue.

Analogy: books in a book-case (where we can't change the position of any book on the case)

If this procedure of splitting and amalgamating of holes continues for a while, we end up with blocks of available memory of various sizes scattered throughout memory. This is called Fragmentation and is to be avoided.

Strategies/Algorithms for allocating memory to processes is a crucial part of memory management.

### Three different strategies

1. First Fit (FF): allocate the first hole that is large enough to accommodate the new process. This is the fastest method, but may cause the most fragmentation-i.e., the most small "pockets" of unused non-contiguous memory.
2. Best Fit (BF): search all available holes and allocate the smallest hole that is big enough to accommodate the new process. This is slower than best-fit, but leads to the smallest "pocket" sizes.
3. Worst Fit (WF): search all available holes and assign part of the largest available hole. This is the slowest method but may leave a smaller number of larger holes than either first or best fit.

Memory is partitioned into contiguous segments and allocated to different processes. The methods for contiguous memory allocation described above can lead to:

- External fragmentation: total memory space exists to satisfy a request, but it is not contiguous. That is, memory is available but is broken up into "pockets" between partitions that are too small to be used.
- Internal fragmentation: Suppose there is a free hole of size 12,100 bytes and we require a partition of size 12,080 bytes. The OS may require more than 20 bytes to keep track of the unused portion, so it can be more economical to allocate all 12,100 bytes even though this is slightly larger than requested memory; this is called "internal fragmentation".

## 0.48 Paging

### 0.49 Basic idea

- Logical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available ("chop up the processes").
- Divide physical memory into fixed-sized blocks called frames. All frames on the system are of the same size, usually some power of 2, determined by the hardware ("chop up the space").
- Logical memory into blocks called pages.
- Frames and Pages are of the same size.
- The OS keeps track of all free frames in memory.
- To run a program that will use  $n$  pages, we need to find  $n$  free frames and load the program.
- Page Table maintained to translate logical to physical addresses.

## 0.50 Address Translation Scheme

An address generated by CPU is divided into:

- Page number ( $p$ ), used as an index into a page table which contains the base address of each page in physical memory.
- Page offset ( $d$ ), combined with the base address to define the physical memory address that is sent to the memory unit.

## 0.51 Basic ideas

- Bring a page into memory only when it is needed (lazy paging).
- Pages belonging to a process may be memory resident or not.
- The system needs some mechanism for distinguishing between resident and non-resident pages. Therefore the page table associates a valid/invalid bit with each page.
- $1 \Rightarrow$  valid/resident     $0 \Rightarrow$  invalid/not in physical memory.
- If a process wishes to access a valid page (a "HIT") it can do so.
- If a process wishes to access an invalid page (a "MISS") then the paging hardware generates a Page Fault.

If there is a reference to an invalid page, reference will trap to OS.

- Reference to nonexisting page  $\rightarrow$  abort
  - Just not in memory  $\rightarrow$  must bring page into memory.
1. Reference is made to invalid page
  2. Page fault generated - call to Operating system.
  3. Locate empty frame in physical memory.
  4. Swap page into frame.
  5. Reset tables, flip validation bit (change to 1)
  6. Restart user process.

Of course, this assumes that there is an empty frame in memory...

## 0.52 What happens if there is no free frame?

### 0.52.1 Page replacement

Find some page in memory that is not in use, and swap it out. We need an algorithm for selecting such a page. The algorithm will be evaluated on the basis of the generation of a minimum number of page faults.

Page fault handling is slow, so the performance of the whole system depends heavily on having an efficient Page Replacement algorithm - one with the lowest possible page fault rate.

To evaluate a page replacement algorithm, we will consider:

- An example where a program has the following stream of virtual pages (also called the "page reference string")

$$\{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\}$$

This means it first references page #1, then page #2, then page #3,... finally page #5. ...

- $F$ , the number of frames on the system
- the algorithm.

For these we shall compute the Frame Hit Rate:

$$\frac{\text{number of Hits}}{\text{Length of page reference string}}$$

The greater the hit rate, the better.

When a free frame is needed, the victim is the page that has been in memory the longest. Suppose that  $F = 3$ , and that at the start no pages are in memory. Then we get

1	2	3	4	1	2	5	1	2	3	4	5
MISS	MISS	MISS	MISS	MISS							
1	2	3									
	1	2									
		1									

If we wanted to improve the hit rate, we could try increasing the number of frames. So, let's redo the previous example, but with  $F = 4$ .

1	2	3	4	1	2	5	1	2	3	4	5
MISS	MISS	MISS	MISS								
1	2	3	4								
	1	2	3								
		1	2								
			1								

This is called B elady's Anomaly: sometimes increasing the number of frames can increase the number of page faults!

Optimal algorithm requires the least number of page faults and does not suffer for Belady's Anomaly: Replace page that will not be used for longest period of time. However, one needs to know what pages will be referenced in the future. So this is difficult to implement.

LRU: The Least Recently Used (LRU) algorithm is similar to FIFO but, rather than removing the frame that has been in memory the longest, we remove the one that has not been referenced for the longest time.

### 0.53 Exercise

Re-do the examples from Slides 26 and 27. How many Page faults are generated?

### 0.54 Memory management in C

We'll finish with a short note about how a process can request memory from the OS. We'll need to recall, from Week 4 (Part 2):

- A Pointer is a special variable that has as its value a memory location.
- Declaring a pointer to an integer is done by:

```
int *p;
```

- The variable  $p$  can contain an address.
- Access the contents of that address with  $*p$ .

In this context, the  $*$  symbol is called a dereferencer.

It is reasonable to think of  $&$  as the inverse of the operator  $*$ . This is because  $&i$  means "the address of the variable stored in  $i$ ", while  $*p$  means "the value stored at the address  $p$ ".

If  $i$  is an integer with value 23 then

```
*(&i)
```

will evaluate as 23. However,

```
&(*i)
```

is illegal. Why?

If we declare  $p$  as a pointer to an integer and set

```
p = &i;
```

then

```
&(*p)
```

will evaluate as the address of  $i$ .

Question: Is  $*(&p)$  legal? If so, what will it evaluate as?

When an array of integers is declared, e.g.,

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

what actually happens is:

- The system declares a pointer to an integer called *a*.
- The pointer is to *a*[0], the "base address" of the array.
- Space is allocated to the addresses pointed to by *a*, *a* + 1, ..., *a* + 9.

It is not true that arrays and pointers are exactly the same. If we declare:

```
int *p, a[10];
```

the value of the pointer *p* can be reassigned anytime we like, but the value of *a* is fixed.

However, because of this, we use pointers when we wish to pass arrays as functions.

Here is an example of passing an array as an argument to a function.

Given an integer array *a*, we'll calculate  $\sum_{i=0}^n a_i$ .

```
int sum_pointer(int *p, int n);
int sum_array(int a[], int n);

int main()
{
    int a[4] = {1, 99, 40, 60};
    printf("a[0]+a[1]+a[2]=%d\n", sum_array(a,3));
    printf("a[1]+a[2]+a[3]=%d\n", sum_pointer(a+1,3));
    return(0);
}

int sum_pointer(int *p, int n)
{
    int i, sum=0;
    for (i=0; i<n; i++)
        sum+=*(p+i);
    return(sum);
}

int sum_array(int a[], int n)
{
    int i, sum=0;
    for (i=0; i<n; i++)
        sum+=a[i];
    return(sum);
}
```

Before studying dynamic memory allocation, we will work out how much storage is required by integers, floats, characters, and even pointers. This is because we need to tell the system how many bytes are required to store an array. Although we can hard-code this into our program, we prefer not to since:

- We might make a mistake.
- Some systems store data types differently from others (standards for the C language specify the minimum number of bytes used, but this can be exceeded).
- There are more complex structures which have variable memory requirements.

So we use the `sizeof()` operator. (Note: you've already seen this when using the read and write functions for pipes).

The `sizeof()` operator returns the number of bytes for a particular data type.

It can take data types (e.g., float, char) or variable names as its argument.

It returns an unsigned integer.

```
int x=-123, *p; char name[6]="CS211";
printf("A char takes\t\t%3lu bytes;\n", sizeof(char));
printf("A float uses\t\t%3lu bytes;\n", sizeof(float));
printf("A double uses\t\t%3lu bytes;\n", sizeof(double));
```

```
printf("x is requires\t\t%3lu bytes;\n", sizeof(x));
printf("A pointer needs\t\t%3lu bytes;\n", sizeof(p));
printf("Array %s is stored in %3lu bytes;\n", name, sizeof(name));
printf("enum MONTH takes\t\t%3lu bytes;\n", sizeof(MONTH));
printf("struct Date takes\t\t%3lu bytes.\n", sizeof(Date));
```

The output I get is:

A char takes	1 bytes;
A float uses	4 bytes;
A double uses	8 bytes;
x is requires	4 bytes;
A pointer needs	8 bytes;
Array CS211 is stored in	6 bytes;
enum MONTH takes	4 bytes;
struct Date takes	12 bytes.

## 0.55 Dynamic memory allocation

Having to declare the size of an array in a function header is a huge restriction. Either we have to modify the program every time we change the size of the array, or we have to define the array to be as big as possible.

This is wasteful of time and resources. To minimize the amount of coding we have to do, and to use resources well, we simply declare an appropriate variable with type:

- Pointer to int for an integer vector: **int \*v**

Next we must ask the system to reserve some memory. There are four important commands:

- **sizeof()** (see above)
- **calloc(n, sizeof(x))**: Continuous Memory Allocation. It will reserve enough space to  $n$  variables each with the same size as  $x$ . It sets them all to zero.
- **malloc(n\*sizeof(int))**: Memory Allocation. As above, but it doesn't do any initialization.
- **free(ptr)**: deallocate the space that begins at the address stored in **ptr**.

The headers for these functions are in **stdlib.h**.

The important thing is that we can call **calloc()**, **malloc()**, and **free()** any time we like. This is what is dynamic about it.

Both **malloc()** and **calloc()** return pointers to the base of the memory they allocated. However, because they are not for specific pointer types (e.g., pointer to int or pointer to char) they return "void pointers".

These 'void pointers' are then automatically promoted to the relevant pointer type, so the following is correct:

```
int c;
float x;
c = malloc(7*sizeof(char));
x = calloc(10, sizeof(float));
```

In the example below, the size of the vector  $v$  is not fixed until the program is run. Note that the sum function will work for any sized array.

```
int main(void)
{
    int *v, n, i, ans;
    printf("How many elements are there in v?: ");
    scanf("%d", &n);
    v = (int *)calloc(n, sizeof(int));
    for (i=0; i < n; i++)
    {
        printf("Enter v[%d]: ", i);
        scanf("%d", &v[i]);
    }
}
```

```

    ans = sum(v, n);
    printf("The sum of the entries of v is %d\n", ans);
    free(v);
    return(0);
}

```

## 1 Module Review

The topics we have covered (not necessarily in order) are:

1. What is an OS?
2. Computer History: from batch systems to distributed systems.
3. Programming with processes: fork(), getpid(), ...
4. Interprocess communication with pipe(), kill(), signal()
5. Threads, including pthreads
6. Process Scheduling Algorithms.
7. Concurrency; race conditions; Critical Sections; locks; Semaphores;
8. The dining Philosophers problem
9. Resource allocation; Banker's algorithm.
10. Memory management; Contiguous allocation;
11. Pages and frames; Page replacement algorithms.

In C Programming, we had:

1. Basic structure. Comments. Header files.
2. Key-words and operators.
3. Selection statements: if else, etc. Logic operators.
4. Loops: for, while, do... while;
5. Input (printf()) and output scanf(); conversion characters.
6. Functions, including argument lists, return values, void, call-by-value; call-by-reference.
7. Pointers
8. Multidimensional arrays; strings, and all the str\* functions.
9. Files: fopen(), fclose(), reading and writing; navigating
10. User-defined types enum, struct, typedef
11. Dynamic memory allocation: calloc(), malloc(), free()

## Q2 (f)

Which of the following CPU scheduling algorithms are preemptive? Select all that apply.

1. First Come First Served (FCFS),
2. Shortest Job First (SJF),
3. Shortest Time To Completion First (STCF),
4. Round Robin (RR).

The table below shows the CPU burst times (in seconds) of four processes submitted in the given order, all at time  $t = 0$ .

Process	$P_1$	$P_2$	$P_3$	$P_4$
Burst Time	20	10	15	5

Sketch Gantt charts for how these processes are scheduled for each of the following algorithms.

1. First-Come-First-Served (FCFS),
2. Shortest-Job-First (SFJ), and
3. Round Robin with a time quantum of  $q = 5$  seconds

You may assume that context switching is instantaneous. Calculate the average turn-around and response times for each algorithm.



**FCFS**

**SJF**

**RR**

### Q3 (f)

What is a Resource Allocation Graph? Explain the relation between cycles in the graph and system deadlock in the case where there is a single instance of each resource.

Consider the two Resource Allocation Graphs in Figure ?? . In both scenarios presented there are four processes,  $P_1, P_2, P_3$  and  $P_4$ , and three resources,  $R_1, R_2$ , and  $R_3$ , of which there is one instance each.

1. Determine if it represents a deadlocked state.
2. If it is deadlocked, explain why.
3. If it is not deadlocked, explain how the resource allocation would proceed so that, eventually, all processes received their desired resource allocation. (You may assume that if a process has its desired resource allocation it will eventually terminate and release that allocation).

Suppose a system has three running processes, each of which requires three resources of a particular type. Furthermore the system has five instances of that resource. Explain, with the aid of a resource allocation graph, how the system could reach deadlock.

Suppose a system has two free memory partitions, of size 400 k and 600 k, in that order.

Four jobs requiring (contiguous) memory space of various sizes are submitted at the same time, and in the following order:

Process	Order submitted	Size
$P_1$	1st	320 k
$P_2$	2nd	200 k
$P_3$	3rd	280 k
$P_4$	4th	200 k

1. Would all these processes be allocated memory if the First-Fit (FF) scheme is employed? If not, which one(s) would be omitted?
2. Would all these processes be allocated memory if the Worst-Fit (WF) scheme is employed? If not, which one(s) would be omitted?

### First Fit

Holes: 400k, 600k. Requests: 320k, 200k, 280k, 200k

Holes: 400k, 600k. Requests: 320k, 200k, 280k, 200k

Suppose a system has  $F = 4$  available frames, and executes a process that has the following "page reference string"

$$\{1, 2, 3, 1, 4, 5, 1, 4, 2, 4, 3, 4\}$$

1. Calculate the Frame Hit rate using the First-In-First-Out algorithm.
2. Calculate the Frame Hit rate using the Least Recently Used algorithm.
3. What would be Frame Hit rate be for both these algorithms if we increase the number of frames (i.e.,  $F \geq 5$ )?

$$\{1, 2, 3, 1, 4, 5, 1, 4, 2, 4, 3, 4\}$$

### Five or more frames

$$\{1, 2, 3, 1, 4, 5, 1, 4, 2, 4, 3, 4\}$$