Assignment 2

CSCU9V4 – Systems

Spring 2022

Student ID: 2925399

Contents

Task 1: Operating Systems

Processes and their life cycle

The role of processes in a computer

A process is a series of actions taken to achieve a particular end.

A computer performs operations sequentially based on instructions given to it by a program with the end goal of executing the entire set of instructions. The act of executing all the instructions in a computer program is a computer process: a program in execution. We can say that a process is a dynamic instance of a computer program.

The life cycle of a computer process

The different phases a computer goes through when performing a process, i.e., running a program, are known as the process life cycle.

The states of a computer process life cycle

A process can have one of five states at a time:

- Start state, which is the initial state when a process is first started/created.
- Ready state, when the process is ready to be assigned to the processor by the operating system so that it can run.
- Running state, when the process has been assigned to the processor by the operating system scheduler and processor executes its instructions.
- Waiting state, when/if the process is waiting for a resource, such as user input or for a file to become available.
- Terminated or Exit state, when the process finishes its execution or is terminated by the operating system: it is moved to the terminated state where it waits to be removed from main memory.

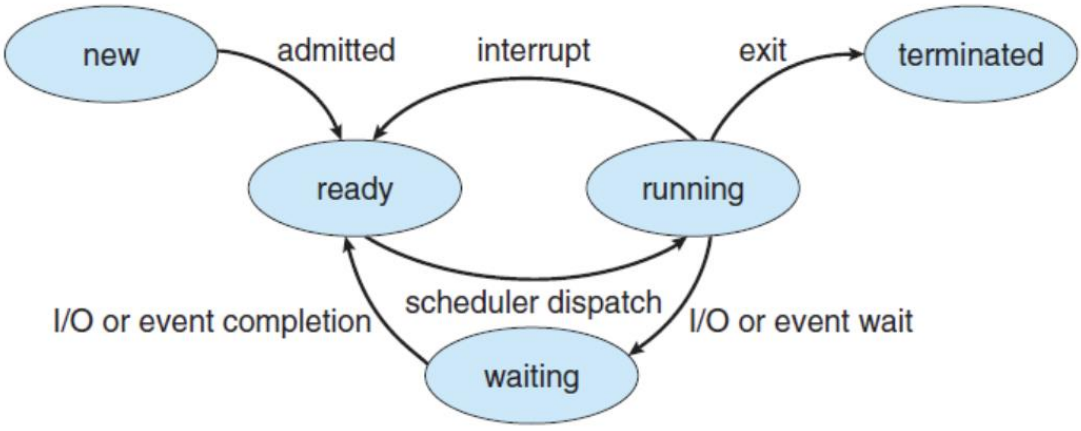


Figure 1: process life cycle [1]

The process scheduler

Process schedulers are operating system software whose role it is to schedule processes for the processor to execute. They select which jobs, work that needs to be done, to load into the system and decide which processes, the way work can be done, to run.

A process scheduler uses scheduling algorithms to schedule different processes to be assigned to the processor.

These algorithms are either non-pre-emptive when processes are not interrupted once they enter the running state and until their life cycle is complete or pre-emptive when processes can be interrupted before their completion.

CPU scheduling is the method by which threads, processes, or data flows are given access to system resources.

The Round Robin scheduling algorithm

Round-robin scheduling is a simple pre-emptive scheduling algorithm.

Processes are received and placed into a request queue according to their arrival time.

Time slices of equal duration called quanta are assigned to each process in circular order, handling all processes without priority.

A time quantum (TQ) is the amount of time share given to each process to execute, interrupting the job if it is not completed within the fixed time.

Burst time (BT) refers to the time required in milli-seconds by a process for its execution.

Here we have an example [2] of the round robin scheduling algorithm:

Process	AT	BT
P1	0	5
P2	1	3
P3	3	6
P4	5	1
P5	6	4

TQ = 3

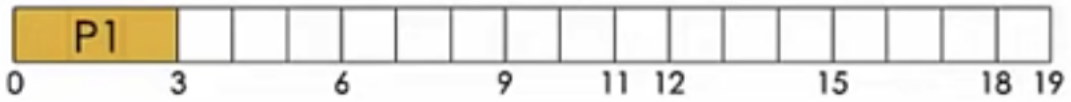
Figure 2: processes that need to be handled and given time quantum [2].

Process	AT	BT
P1	0	~~5~~ 2
P2	1	3
P3	3	6
P4	5	1
P5	6	4

Request queue:

~~P1~~

TQ = 3



AT = Arrival Time

BT = Burst Time

Figure 3 [2]: First, process P1 arrives and is the first to be put in the request queue. P1 is executed from 0 to 3 because the time quantum is 3, even though the burst time is 5. The remaining time left for P1 will be 2 seconds now.

Process	AT	BT
P1	0	~~5~~ 2
P2	1	~~3~~ 0
P3	3	6
P4	5	1
P5	6	4

Request queue:

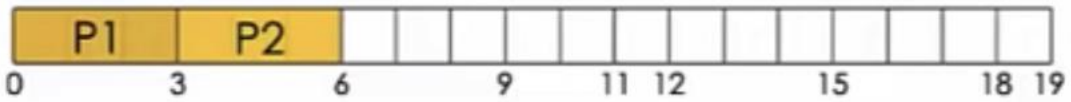
~~P1~~

~~P2~~

P3

P1

TQ = 3



AT = Arrival Time

BT = Burst Time

Figure 4 [2]: Column (AT) stands for arrival time. P2 and P3 arrive after 3 time units and are added to the request queue. P1

has not been completed fully and still has some remaining burst time left. P1 is put back in the request queue.
Next in the queue is P2 which executes for 3 seconds, from 3 to 6. The remaining time left for P2 is 0, so the process is completed and does not go back into the queue.

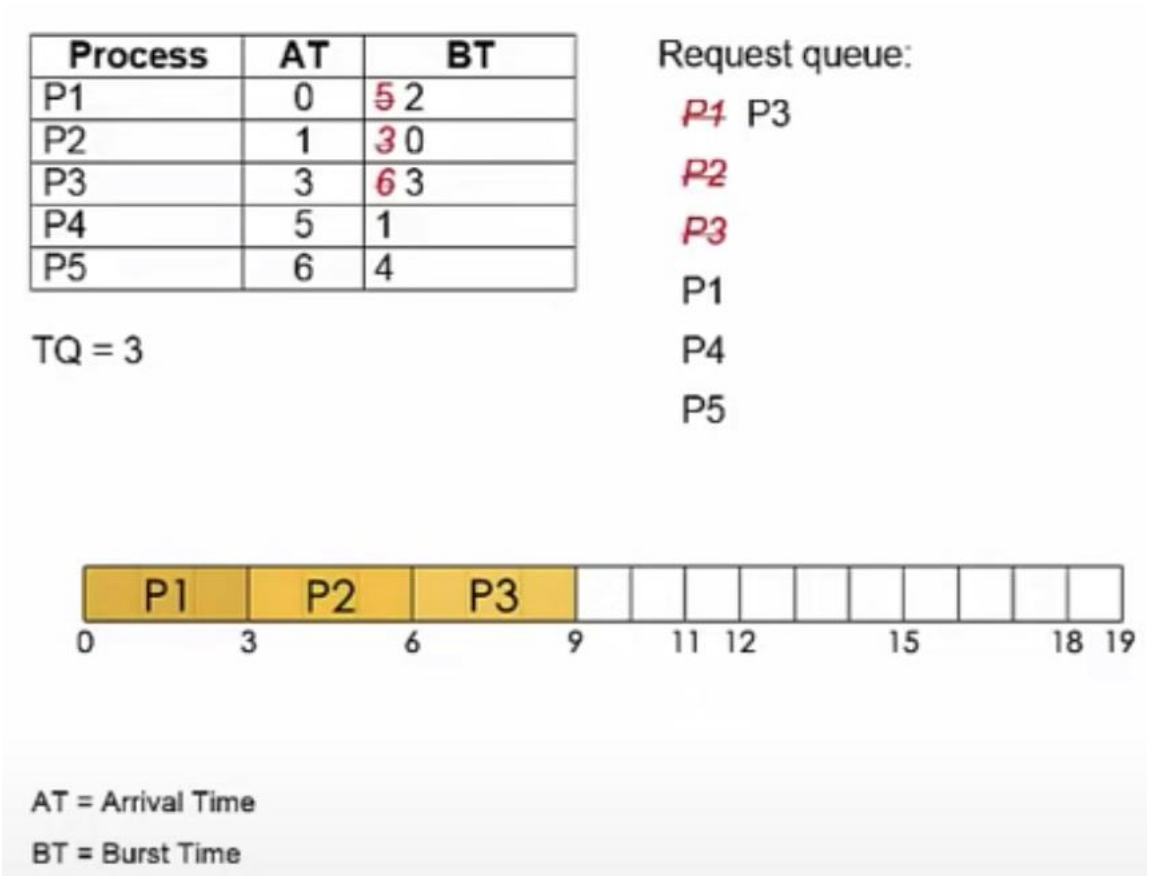


Figure 5 [2]: 6 seconds have passed P4 and P5 arrive and are added to the request queue.
P3 is next and is executed from 6 to 9 time units, which leaves P3 with a remaining execution time of 3. P3 has not been completed fully so it is put back in queue.

Process	AT	BT
P1	0	~~5~~ ~~2~~ 0
P2	1	~~3~~ 0
P3	3	~~6~~ 3
P4	5	1
P5	6	4

TQ = 3

Request queue:

~~P1~~ P3

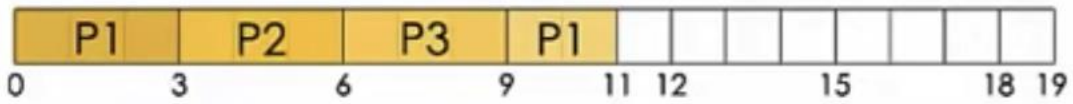
~~P2~~

~~P3~~

~~P1~~

P4

P5



AT = Arrival Time

BT = Burst Time

Figure 6 [2]: Next in queue is P1, so it is executed again. This time for 2 seconds to finish its remaining burst time, from 9 to 11, completing process P1.

Process	AT	BT
P1	0	~~5~~ ~~2~~ 0
P2	1	~~3~~ 0
P3	3	~~6~~ 3
P4	5	~~1~~ 0
P5	6	4

TQ = 3

Request queue:

~~P1~~ P3

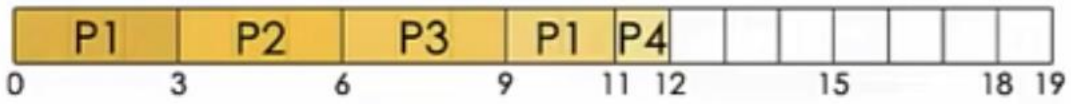
~~P2~~

~~P3~~

~~P1~~

~~P4~~

P5



AT = Arrival Time

BT = Burst Time

Figure 7 [2]: Next in queue is P4, it executes for 1 second because P4 has a burst time of 1. P4 is from time unit 11 to 12 and is complete.

Process	AT	BT
P1	0	~~5~~ ~~2~~ 0
P2	1	~~3~~ 0
P3	3	~~6~~ 3
P4	5	~~1~~ 0
P5	6	~~4~~ 1

TQ = 3

Request queue:

~~P1~~ P3

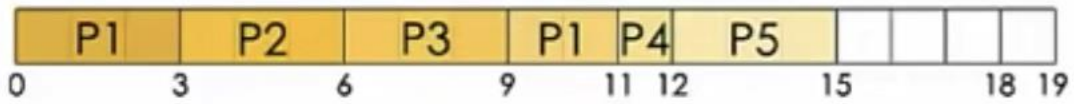
~~P2~~ P5

~~P3~~

~~P1~~

~~P4~~

~~P5~~



AT = Arrival Time

BT = Burst Time

Figure 8 [2]: Up next is P5, it executes for 3 seconds, from 12 to 15. This leaves P5 with a remaining burst time of 1 so it goes back in the queue.

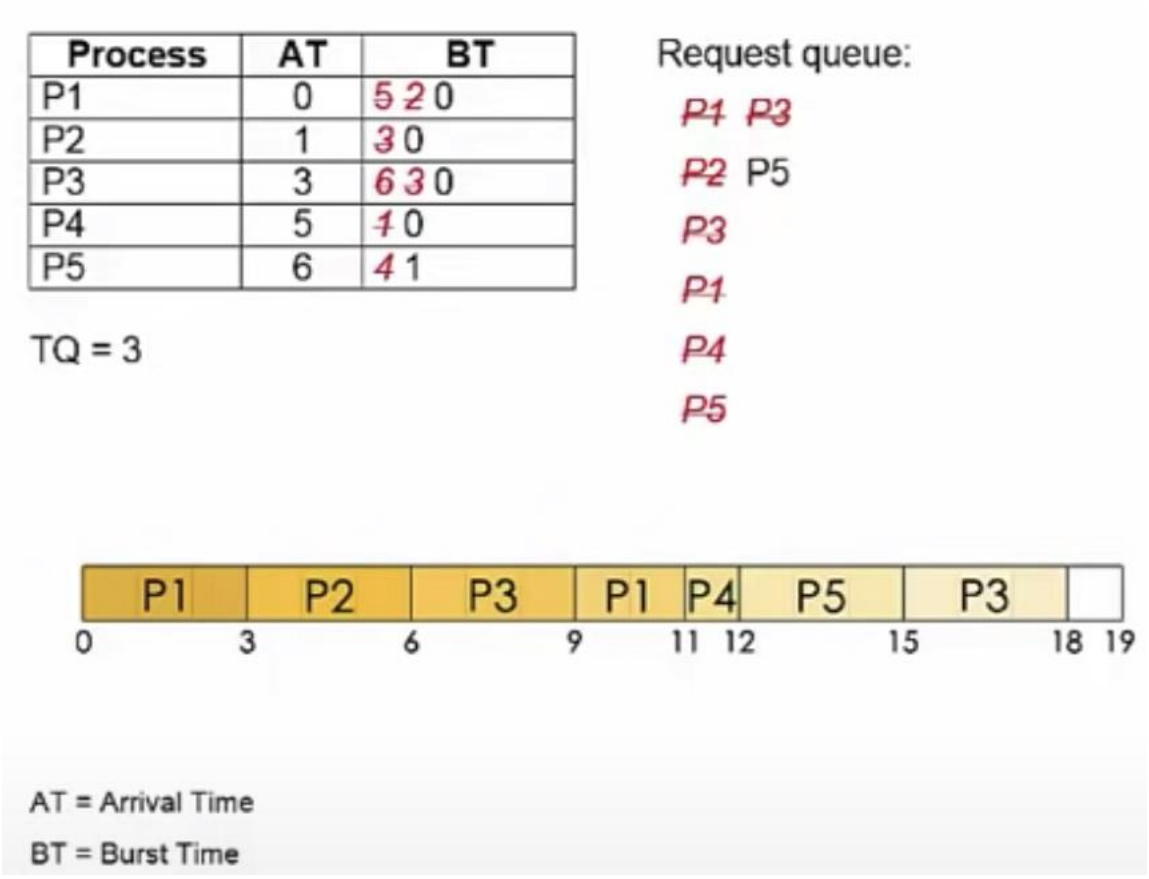


Figure 9 [2]: Next in queue is P3 again, it executes for 3 seconds from 15 to 18. P3 has 3 seconds burst time remaining, the process is completed.

Process	AT	BT
P1	0	~~5~~ ~~2~~ 0
P2	1	~~3~~ 0
P3	3	~~6~~ ~~3~~ 0
P4	5	~~1~~ 0
P5	6	~~4~~ ~~1~~ 0

TQ = 3

Request queue:

~~P1~~ ~~P3~~

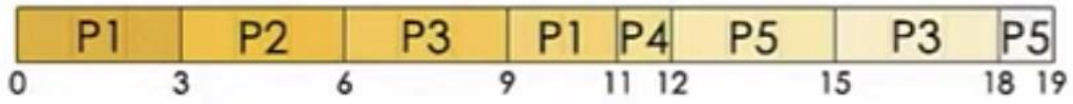
~~P2~~ ~~P5~~

~~P3~~

~~P1~~

~~P4~~

~~P5~~



AT = Arrival Time

BT = Burst Time

Figure 10 [2]: Finally, next in queue is P5 with a remaining burst time of 1 second, P5 executes for 1 second, from 18 to 19.

Task 2: Numbers

a. minmax.c program

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

int main (void){
    int number_of_ints, counter;
    double min = __DBL_MAX__, max = __DBL_MIN__, avg, sum;

    printf("Insert the number \"n\" of floating point numbers to calculate the
minimum, maximum, and average of:\t");
    scanf("%d", &number_of_ints);

    for (counter = 0; counter < number_of_ints; counter++){
        printf("Enter %d of %d: \t", (counter+1), number_of_ints);
        double num;
        scanf("%lf", &num);
        sum += num;
        if(num>max){
            max = num;
        }
        if(num<min){
            min=num;
        }
    }

    avg = sum/number_of_ints;

    printf("Maximum number is\t%.3lf\nMinimum number is\t%.3lf\nAverage
is\t%.3lf\n", max, min, avg);

    return 0;
}
```

```
Insert the number "n" of floating point numbers to calculate the minimum, maximum, and average of:      3
Enter 1 of 3:   1.4323453
Enter 2 of 3:   2.3
Enter 3 of 3:   2345
Maximum number is       2345.000
Minimum number is       1.432
Average is      782.911
```

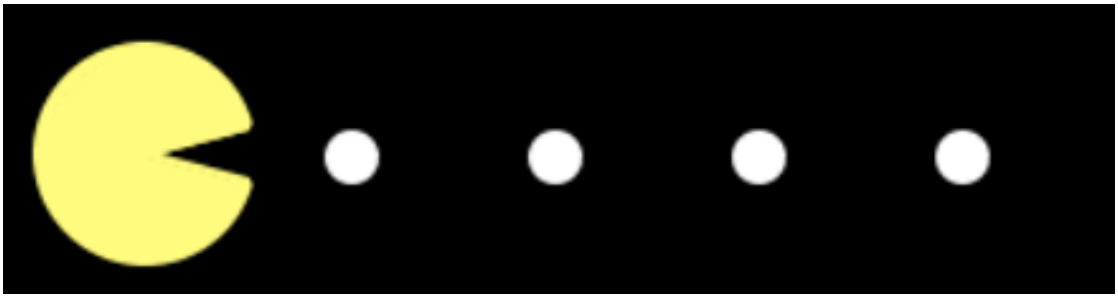
b. how the scanf() function works

[3] [4] [5] [6] [7] [7] [7, 8, 9, 10]The scanf() function is included with the header file stdio.h, which contains C functions for performing input and output.

When we run a C program it treats devices as files. The keyboard file is called Standard Input. We use the file pointer stdin to access Standard Input for reading.
The stdin file pointer is not a memory address, rather it is an Input/Output stream: a sequence of data elements made available over time.

The scanf() function allows us to read this stream of input from the keyboard and scan it according to a format we provide in what is known as the format string.

It can be helpful to visualise this stream of data as Pac-Man:



Each dot is a character from the input stream, and scanf() reads one dot at a time, just like Pac-Man.

The format string specifies the type of data to read from the input stream through what are known as conversion specifications, which begin with the % character: scanf() matches groups of characters from the input stream with conversion specifications in the format string.
For each conversion specification scanf() tries to locate an item of appropriate type in the input data, skipping blank space if necessary.

For example, in the following piece of code:

```
long long upc = 0;

printf("Insert a 12 digit UPC: ");
scanf("%lld", &upc);
```

The first scanf() argument in double quotes holds the format string. It contains one conversion specifier of type long long integer (lld). So scanf() will look for to match groups of characters from the input stream that are of type long long integer.
The matched data from the input stream can be assigned to a variable in the C program via a pointer to said variable: the & unary operator returns the address of a variable in memory. We can observe this in the code above where &upc tells scanf() to assign the matched group of characters to the variable upc.

Let's look at an example where things don't work as expected.
We want the program to accept fractions with a space in between numbers, e.g. 2_/_3.

```
Insert 2 fractions in the X / Y format and I will display the sum!
Insert fraction 1 "X / Y" :     2/4
Insert fraction 2 "X / Y" :     2 / 3
The sum of the fractions:       2 / 4   +       2 / 1   =       10 / 4
```

Something isn't working as anticipated. Our second fraction is not being read correctly when we leave a space in between numbers.

```
printf("Insert 2 fractions in the X / Y format and I will display the sum!\n");
printf("Insert fraction 1 \"X / Y\" :\t");
scanf("%d/%d", &numerator1, &denominator1);
printf("Insert fraction 2 \"X / Y\" :\t");
scanf("%d/%d", &numerator2, &denominator2);

int lowest_common_multiple = denominator1 * denominator2;
int new_numerator1 = (lowest_common_multiple / denominator1) * numerator1;
int new_numerator2 = (lowest_common_multiple / denominator2) * numerator2;
int numerator_sum = new_numerator1 + new_numerator2;

printf("The sum of the fractions:\t%d / %d\t+\t%d / %d\t=\t%d / %d",numerator1, d
```

The problem resides in our scanf function calls.

It is very important to remember a space after conversion specifications in the scanf format string if you want to read other input symbols from the input stream, like forward slashes (/).

Otherwise scanf() will try to match the symbol after the conversion specification in the format string with the next symbol it finds in the input stream. If it doesn't find the same symbol scanf() will terminate without reading a value for the following inputs!

In this case, if the user inputs a space instead of a forward slash (/) after having inputted the numerator scanf() terminates without reading a value for the denominator.

```
printf("Insert fraction 1 \"X / Y\" :\t");
scanf("%d /%d", &numerator1, &denominator1);
printf("Insert fraction 2 \"X / Y\" :\t");
scanf("%d /%d", &numerator2, &denominator2);
```

Adding a space after the first conversion specifier tells scanf() to eat, or skip, any white space until the next symbol: (/) is found in the input stream.

The reason we don’t need to insert another (_) white space before our denominator is because the %d conversion specifier automatically eats any preceding white spaces.

```
Insert 2 fractions in the X / Y format and I will display the sum!
Insert fraction 1 "X / Y" :     2/4
Insert fraction 2 "X / Y" :     2 / 3
The sum of the fractions:       2 / 4   +       2 / 3   =       14 / 12
```

Now the program works as intended: it accepts fractions with a white space in between numbers.

Task 3: Reading the code

```
for (int i = 0;  i < (int) (sizeof(a) / sizeof(a[0]));  i++)
      a[i] = 0;
```

This is a for loop: a block of code that runs multiple times until a given condition returns false.

For loops provide the following functionality:

- Initialise the iterator variable using an initial value
- Check if the iterator has reached its final value
- Increases the iterator

The first statement within the parentheses is the initialisation statement. It is executed first and only once.

```
int i = 0;
```

This step allows us to declare and initialize the loop control variable i with a value of zero. For loops require an iterator variable *i* that counts the number of times the body of a for loop has been executed.

The second statement within the parentheses is the test expression or condition. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute, and the flow of the program jumps to the next statement after the for loop.

```
i < (int) (sizeof(a) / sizeof(a[0]));
```

The expression:

```
(sizeof(a) / sizeof(a[0]))
```

Determines the length of array *a* by dividing the size of array *a* by the size of one of its elements.
The *sizeof()* operator computes the size of its operand by returning the amount of memory allocated to that data type.
The / operator divides the numerator to the left of / by the de-numerator to the right of /.
Here *a* is an array and *a[0]* is the first element of array *a*.
The syntax before the expression is a type cast:

```
(int)
```

Type casting converts one data type into another.
Here the length of array *a* is cast to the integer data type so that it can be compared to the loop control variable *i* which is also of type integer.
The start of the test expression:

```
i <
```

Checks if the control variable *i* is less than the length of the array using the relational operator <.
If this condition is true, the body of the for loop is executed.

The body of this for loop contains:

```
a[i] = 0;
```

This statement accesses the *i-th* element of the array *a* and sets its value to zero.

The final statement in the parentheses is the update statement or increment. After the body of the for loop executes, the increment statement updates any loop control variables.

```
i++
```

Here the control variable *i* is incremented by one.

The test expression or condition is now evaluated again. If it is true, the loop executes, and the process repeats itself:

- Body of loop
- Increment step
- Check condition for validity

After the condition becomes false, the loop terminates.

In conclusion, this loop cycles through the elements of array *a* and changes the value of all of *a*'s elements to zero.

Task 4: Functions and array

a. maxArray.c program & maxptr(int a[]) function

```c
#include <stdio.h>

int* maxptr(int a[]){

    //a variable to hold the max value
    int max = a[0];
    //declaring a pointer (variable that stores an address rather than a value) that will point to the maximum element of the array
    int* max_pointer = &a[0]; //Pointing to first element in array which we set as max value. If this isn't done then an array of length 1 would not have its max value pointed to by max pointer and the function would not operate as desired.
    //a for loop to cycle through the elements of the array
    int i;
    for (i=1; i<10; i++){ //index i can start at 1 because element at index 0 is already set to max value!
        //if this element is bigger than the previous biggest element, make this the biggest element.
        if (a[i] > max){
            max = a[i];
            //storing the address of the maximum element of the array in the pointer
            max_pointer = &a[i];
        }
    }
    //replacing the maximum element in the array with -1 using the pointer
    *max_pointer = -1;
    //returning the pointer to the element of the array that used to be the maximum
    return max_pointer;
}

int main (void){
    //defines an array of 10 positive integers
    int nums[10] = {1,2,3,4,50,6,7,8,9,10};

    //prints the array of 10 integers
    printf("array of positive integers:\n");
    for (int j = 0; j < 10; j++){
        printf("\tn%d = %d ;", (j+1), nums[j]);
    }

    //calls the maxptr() passing the array as parameter
    int* max_p = maxptr(nums);

    //print the returned pointer
    printf("\nPrinting the pointer:\t%p", max_p);

    //prints the array after the execution of maxptr()
    printf("\nthe same array after execution of maxptr():\n");
    for (int j = 0; j < 10; j++){
        printf("\tn%d = %d ;", (j+1), nums[j]);
    }
```

```
array of positive integers:
    n1 = 1 ;    n2 = 2 ;    n3 = 3 ;    n4 = 4 ;    n5 = 50 ;   n6 = 6 ;    n7 = 7 ;    n8 = 8 ;    n9 = 9 ;    n10 = 10 ;
Printing the pointer:   0x7ff7b469f570
the same array after execution of maxptr():
    n1 = 1 ;    n2 = 2 ;    n3 = 3 ;    n4 = 4 ;    n5 = -1 ;   n6 = 6 ;    n7 = 7 ;    n8 = 8 ;    n9 = 9 ;    n10 = 10 ;
```

b. maxptr() function workings

The *maxptr()* function:

- takes an array of positive integers as a parameter,
- finds the maximum element of the array,
- returns a pointer to the maximum element of the array, and
- replaces the maximum element in the array with -1 (using the pointer).

```
int* max_p = maxptr(nums);
```

This is the line of code in the maxArray.c program that calls the function *maxptr()*. Here a variable named *max_p* of type pointer with base type integer is declared and initialised with the return value of the function *maxptr()* when passed the array of positive integers *nums*.

The *maxptr()* function has return type pointer of base type integer and a single parameter: an array of integers *a*.

```
int* maxptr(int a[]){
```

When *maxptr()* is called the first thing it does is declare two variables:

1. A variable named *max* of type integer which is initialised with a value of *a[0]*, the first element of the array.
   This variable will hold the maximum element of the array.
2. A variable named *max_pointer* of type pointer.
   This pointer will point to the maximum element of the array.

The next operation performed by the *maxptr()* is identifying the greatest element in the array using a for loop to cycle through the elements of the array.

If an element *a[i]* is bigger than the current *max* element, *max* is set to the value of the current element and *max_pointer* is set to point to the memory address of *a[i]*, the new *max* element of the array *a*.

Once the loop terminates the *max* element of the array has been identified and *max_pointer* is pointing to it.

At this point dereferencing is used to replace the value of the maximum element in the array with the value *-1*.

```
*max_pointer = -1;
```

Dereferencing is the act of referring to where a pointer points, instead of the memory address of a pointer. Dereferencing a pointer is achieved using the asterisk operator (*).

References

[1] Zitoc, “Process Life Cycle,” [Online]. Available: https://zitoc.com/process-life-cycle/.

[2] ouchouchbaby, “Round Robin Algorithm Tutorial (CPU Scheduling),” 18 March 2015. [Online]. Available: https://www.youtube.com/watch?v=aWlQYllBZDs. [Accessed April 2022].

[3] Y. Peng, “c-scanf-function,” [Online]. Available: https://www.demo2s.com/c/c-scanf-function.html.

[4] G. Samaras, “Caution when reading char with scanf (C),” [Online]. Available: https://gsamaras.wordpress.com/code/caution-when-reading-char-with-scanf-c/.

[5] tutorialspoint, “C - Input and Output,” [Online]. Available: https://www.tutorialspoint.com/cprogramming/c_input_output.htm.

[6] tutorialspoint, “C library function - scanf,” [Online]. Available: https://www.tutorialspoint.com/c_standard_library/c_function_scanf.htm.

[7] D. D. Bhowmik and D. A. Bracciali, *Formatted Input/Output.*

[8] D. D. Bhowmik and D. A. Bracciali, *Introduction: C Part II.*

[9] unwind, “What is stdin in C language?,” [Online]. Available: https://stackoverflow.com/questions/37719915/what-is-stdin-in-c-language.

[10] Wikipedia, “Stream (computing),” [Online]. Available: https://en.wikipedia.org/wiki/Stream_(computing).

[11] edureka, “How To Best Implement For Loop In C?,” [Online]. Available: https://www.edureka.co/blog/for-loop-in-c/#ForLoopinC.

[12] programiz, “C for loop,” [Online]. Available: https://www.programiz.com/c-programming/c-for-loop.

[13] learn-c.org, “learn-c.org,” [Online]. Available: https://www.learn-c.org/en/For_loops.

[14] tutorialspoint, “for loop in C,” [Online]. Available: https://www.tutorialspoint.com/cprogramming/c_for_loop.htm.

[15] learn-c, “Pointers,” [Online]. Available: https://www.learn-c.org/en/Pointers.

[16] tutorialspoint, “Operating System - Processes,” [Online]. Available: https://www.tutorialspoint.com/operating_system/os_processes.htm.

[17] tutorialspoint, “Operating System Scheduling algorithms,” [Online]. Available: https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm.

[18] D. D. Bhowmik and D. A. Bracciali, *Arrays.*