Node of Operational System

Week 1

Practice Problems

1. Consider the following pseudocode. Show how the computation-steps are executed on a von Neumann computer(code and figure shown below)

```
foo(){
    readIO a;
    readIO b;
    c = a + b;
    store c;
    d = a - b;
    print d;
}
```

> 1. *Control Unit understands that data needs to be provided by user*
> 2. *Data is read from input device and brought to the GPU*
> 3. *Control Unit understands that data needs to be provided by user*
> 4. *Data is read from input device and brought to the GPU*
> 5. *Controller commands ALU to compute addition*
> 6. *Controller copies data from ALU to Memory*
> 7. *Controller commands ALU to comute substraction*
> 8. *Controller sends data from ALU to display*

2. With an example, explain the advantages of having registers inside CPU.

> *Registers imporve processing speed. Without registers, Controller need to transfer data between CPU and Main Memory which will take a long time*

3. How are pointer variables related to the memory of a computer?

> *Compiler allocates memory for the variables. Each variable has a unique address. A pointer is a variable that contains the address of a variable*

4. How are pointers and arrays related in C?

> *The pointer of an array will point to the first element of the array*

5. What deose the following function perform?

```
void foo(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

> *Switch the values in `px` and `py`*

6. Consider little-endian representation of data. What will be the output of this program?

```c
int main(){
    int a[] = {5, 10, 15, 20};
    char *p;
    p = (char *) a;

    int i;
    for(i = 0; i < 5; i++)
        printf("%d", *(p+i));

    return 0;
}
```

> ***Little endian**: "little byte at the little position."*
> *Least byte is stored at the first address*
> *`int a[] = {5, 10, 15, 20};`*
> *Each int takes 4 bytes*
> *| 5 | 0 | 0 | 0 | 10 | 0 | 0 |*
> *A char pointer points to Bytes one-by-one*
> *`char *p = (char*)a;`*
> *So, the program will print the bytes*
> *`5 0 0 0 10`*

- How do you know if your computer is little or big endian?
  - You can pint the bytes one by on
    In a big endian computer, int a = 6 will be stored as
    | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 10 |
    so, the printed bytes will be `0,0,0,5`

7. What will be the output of this program?

```c
int main(){
    float arr[5] = {12.5, 10.0, 13.5, 90.5, 0.5};
    float *ptr1 = &arr[0];
    float *ptr2 = ptr1 + 3;
    printf("%f\n", *ptr2);
    printf("%d\n", ptr2-ptr1);
    return 0;
}
```

> *`90.5 3`*

8. Consider little-endian representation of data. What will be the output of this program?

```c
int main(){
    int a;
    char *x;
    x = (char *) &a;
    a = 512;
    x[0] = 1;
    x[1] = 2;
    printf("%d\n", a);
```

```
    return 0;
}
```

> *a = 512 = 22^8 + 0*
> *In little endian*
> *| 0 | 2 | 0 | 0 |*
> *A byte is 8 bits. So, a maximum value a byte can have is 2^8-1 = 255*
> *x is char pointer*
> *| 1 | 2 | 0 | 0 |*
> *So, the new value of a = 1 + 22^8 = 513*

9. What will be the output of this program?

```
int main(){
    int a[5] = {1,2,3,4,5};
    int *ptr = (int*)(a+1);
    printf("%d %d", *(a+1), *(ptr-1));
    return 0;
}
```

> *2 1*

10. How is the string "Hello World!" stored in the memory? How many bytes does this string consume in C?

> *A string in C is `\0` terminated*
> *So, consider an extera byte for that*
> *13 bytes*

11. What does the following function do with the two input string pointers?

```
void foo(cahr *s, char *t){
    int i = 0;
    while((s[i] = t[i]) != '\0')
        i++;
}
```

> *foo() copies a string t into s*

12. What does the following function do with the two input string pointers?

```
void foo(char *s, char *t){
    while((*s = *t) != '\0'){
        s++; t++;
    }
}
```

> *foo() copies a string t into s*

13. What does the following function do with the two input string pointers?

```
void foo(char *s, char *t){
    while ((*s++ = *t++) != '\0');
}
```

> *foo() copies a string t into s*

14. Write a C program that prints the elements of a 2D matrix in column-major order using a pointer.

```
int *p = &a[0][0]
int i,j;
for(i = 0; j < COL; i++){
    for(j = 0; j<Row; j++){
        printf("%d\n", *(p+j*COL));
    }
    p++;
}
return 0
```

15. Write a C program that prints the elements of a 2D matrix in row-major order using a pointer.

```
int *p = &a[0][0]
    int i;
    for(i = 0; j < ROW*COL; i++){
        printf("%d\n", *(p+i));
    }
```

Week 2

Practice Questions

1. This question is about memory layout of a typical C program. What are the segmentas of a C program?

> *Text or code segments, Data segments, stack segment and Heap segment*

2. Describe the differences between local, heap, global and static variables.

> *Stack segment is used to store all local or automatic variables*
> *heap segment is used to store dynamically allocated variables are stored*
> *Data segments contain initialized and uninitialized global and static variables respectively*

3. With an example, describe how stack frames are created and destroyed during function calls.

> *Scope is the function. They are allocated in the stack-frame of the function. After the function call, the stack-frame is released*

4. Describe the consequences of returning a pointer to a local variable from a function call. Give an example.

> *The local variable is released after function call. So the return pointer is not existed*

5. Describe the differences between pass-by-value and pass-by-pointer

> *Pass-by-value: function gets a local copy of variable, so it will only happen within function*
> *Pass-by-pointer: function gets a local copy of variable which contains the address, so it will update the memory location where variable is stored*

6. What is wrong with this program?

```c
int *max(int *a, int *b){
    int temp;
    if(*a > *b)
        temp = *a;
    else
        temp = *b;
    return &temp;
}
int main(){
    int a = 4, b = 5;
    int *c;
    c = max(&a, &b);
    printf("Max value = %d",*c);
    return 0;
}
```

> *The variable which return in `max()` is local variable. It will be released after calling the function.*
> *`c` points to `temp` which stored in the stack-frame(becomes invalid after `max()`) of `max()`*

7. This question is about dynamic memory allocation. Write C language syntax for dynamically allocating an integer array of length LENGTH using malloca(). What happens when malloc() fails to allocate memory?

```c
int main(){
    int LENGTH, i;
    int *p;
    printf("Provide array size:");
    scanf("%d", &LENGTH);

    if((p = (int*) malloc(LENGTH * sizeof(int))) == NULL){
        printf("Allocation failed");
        exit(-1);
    }
    printf("Provide %d integers\n", LENGTH);
    for(i = 0; i<LENGTH; i++)
        scanf("%d", p+i);

    free(p);
    return 0;
}
```

8. The following unfinished C program declares an array of four pointers in line 5. Using nested for loops with counters i and j, complete the unfinsihed program to construct the two-dimensional matrix

```
00 01 02 03
04 05 06 07
08 09 10 11
12 13 14 15
```

such that p[i] points to the i-th row of the matrix. Do not forget to free the allocated memory in the end.

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
    int i, j;
    int *p[4];

    for (i = 0; i<4; i++){
        if((p[i] = (int*) malloc(4*sizeof(int))) == NULL){
            printf("Allocation error");
            exit(-1);
        }
    }

    for (i = 0; i<4; i++){
        for(j = 0; j<4; j++){
            p[i][j] = i*4+j;
        }
    }

    for (i = 0; i<4; i++){
        for(j = 0; j<4; j++){
            printf("%d \t", p[i][j]);
        }
    }

    for (i = 0; i<4; i++){
        free(p[i]);
    }

    return 0;
}
```

9. What is wrong with these function calls?

```c
int *foo1(){
    int x = 20;
    return &x;
}
int *foo2(){
    int *p;
    *p = 20;
    return p;
}
```

> *`x` and `p` are the local variable which will be invalid after calling the function*

10. In the following code snippet, a structure object is passed to a function foo() by value

```
struct VectorPair{
    int a[512];
    int b[512];
};

VectorPair foo(VectorPair VP1){
    VectorPair VP2;
    int i;
    for(i=0; i<512; i++){
        VP2.a[i] = VP1.a[i] + VP1.b[i];
        VP2.b[i] = VP1.a[i] - VP1.b[i];
    }
    return VP2;
}
```

Rewrite the functionfoo so that objects are passeed using C style pointers;

```
struct VectorPair{
    int a[512];
    int b[512];
};

VectorPair *foo(VectorPair *VP1, VectorPair *VP2){
    int i;
    for(i=0; i<512; i++){
        *VP2.a[i] = *VP1.a[i] + *VP1.b[i];
        *VP2.b[i] = *VP1.a[i] - *VP1.b[i];
    }
    return *VP2;
}
```

11. Consider the following three C functions:

```
int *g1(void)
{
    int x = 10;
    return (&x);
}
int *g2(void)
{
    int *px;
    *px = 10;
    return px;
}
int *g3(void)
{
    int *px;
```

```
    px = (int *)malloc(sizeof(int));
    *px = 10;
    return px;
}
```

which of the above three functions are likely to cause problems with pointers?

> *`g1()` returns pointer to local object*
> *`g2()` has a bigger problem. Nomemory is allocated for `px`. Program will cause `segment fault`*
> *`g3()` looks fine. It allocates heap memory and then returns a pointer to that memory.*

12. Does this program leak memory?

```
int main(){
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 6;
    printf("%d", *p);
    return(0);
}
```

> *Obvious, leaks memory. No `free()` is called*

13. Does this program function correctly? Why or why not?

```
void foo(int*a){
    a = (int*)malloc(sizeof(int));
}
int main(){
    int *p;
    fun(p);
    *p = 6;
    printf("%d", *p);
    return(0);
}
```

> *No `free()` is called*

14. In C, a pointer to a pointer variable is declared using **. How many bytes are allocated, deallocated and leaked in this program.

```
int main(){
    int *p1, **p2;
    p1 = malloc(sizeof(int));
    *p1 = 5;
    p2 = malloc(sizeof(int*));
    *p2 = p1;
    free(p1);
    return 0;
}
```

> *There are memory leaks. No `free()` is called for `p2`*

15. What is the `double free` problem?

> *Double free errors occur when `free()` is called more than onece with the same memory address as an argument*
> *When `free()` is called twice with the same argument, the program's memory management data structures become corruptes*
> *Every time free() is called, the address is added in the list of available memory blocks*
> *The last address added to the list can by the next malloc() call*
> *Since the freed address is present twice in the list, next two memory allocations will have the same address*

16. Type casting in C refers to changing a variable of one data type into another. With examples, describe advantages and pitfall of typecasting in C.

> *Advantages*
>
> - *Type casting in C programming makes the program very lightweight*
> - *Type representation and hierarchies are some features we can take advantage of with the help of typecasting*
> - *Type casting helps programmers to convert one data type to another data type*
>
> *pitfall*
>
> - *the type casting may not fit for all the types*

17. Describe the difference between `little` and `big` endian representations

> ***little endian**: Least byte is stored at the first address*
> ***Big endian**: Least byte is stored at the last address*

18. How many bytes are there in `unsigned int` a = `0x12345678` ; Print the bytes one-by-one

```c
int main(){
    unsigned int a = 0x12345678;
    char *p;
    p = (char *) a;

    int i;
    for(i = 0; i < 8; i++)
        printf("%d", *(p+i));

    return 0;
}
```

19. You have two 32-bit integer variables.

```c
unsigned int a = 0x12345678;
unsigned int b;
```

Copy `a` into `b` byte-by-byte such that in the end of the copy operation, we have a and b equal

```
int main(){
    unsigned int a = 0x12345678;
    unsigned int b;
    char *p = &a;
    char *q = &b;
    p = (char *) a;
    q = (char *) b;

    int i;
    for(i = 0; i < 8; i++)
        q[i] = p[i] ;

    return 0;
}
```

20. What is the memory hierarchy of a computer? What are the advantages of using a `memory hierarchy` ?

> *A computer system is made of different parts which are register, cache memory, main memory, hard disk, and magnetic tape It can balance access time and cost*

21. What is the locality of reference?

> *Locality of reference is the tendency of a processor to acces the same set of memory locations repetively over a short period of time*

22. With a diagram, show the memory layout of a 2D matrix.

> *As same as 2d array*

23. The following program negates a 2D array

```
int sum_array(int a[N][M]){
    int i,j;
    for(i = 0; i<M;i++){
        for(j = 0; j<n; j++){
            a[j][i] = -a[j][i];
        }
    }
}
```

Why does this program cannot exploit memory hierarchy?
Rewrite the program such that it can exploit the memory hierarchy and thus become faster.

> *The 2D array is stored in Row-major order but this program is using in column-major order which means the 2D array will not be load to cache because it doesn't fit spatial locality*

Week 3

Practice Problems

1. During Week 2 you have seen how to implement a linked list in C. How can you make a linked-list more efficient by exploiting the memory hierarchy? For example, finding an element in the list becomes faster. There can be other operations as well.

   > *Store the elements of linked list in continuous addresses which will fit spatial locality and load the linked list to the cache*

2. What is a function pointer in C? Give an example where function pointer can be useful.

   > ***int** (*foo)(**int**);*

3. Are they the same or different?

   ```
   int (*foo)(int);
   int *foo(int);
   ```

   > *`int (*foo)(int)` is the declaration of the function pointer*
   > *`int *foo(int)` is the function returns pointer of type int*

4. The following C program verifies a password provided by a user. If the provided password matches with the stored password, then the program prints the contents of a secret function. Otherwise the program terminates.
   The program is running in a server and you have access to the program through a terminal. The program asks you to provide a 6-letter password. Your goal is to get inside the secret function.
   You are aware of the source code, but you do not know what the secret password is. Describe a way to cheat the password verification scheme. [Hint: see buffer overflow]

   ```c
   #include<stdio.h>
   #include<stdlib.h>
   int secret_function(){
       printf("Inside secret function!\n");
       return 0;
   }
   int password_verify(){
       // Assume password is of length 6
       char received_password[7];
       char password_stored[7]; // one extra for \0
       FILE *fp;
       // Program reads password from file
       fp = fopen("secret_file", "r");
       fscanf(fp, "%s", password_stored);
       fclose(fp);
       // Program receives user-input
       printf("Enter 6 letter password: ");
       scanf("%s", received_password);
       // Verify password char-by-char
       int i;
       for(i=0; i<6; i++){
           if(received_password[i] != password_stored[i]){
               printf("Password not matched\n");
   ```

```
            exit(-1);
        }
    }
    printf("Password matched! Welcome!\n");
    secret_function();
    return 0;
}
int main(){
    password_verify();
    return 0;
}
```

> *`received_password` and `password_stored` are stored in continuous addresses and `scanf` doesn't check the length of input. So we can input `aaaaaaaaaaaaaa` to overwrite the `password_stored`. So that they are the same in value now*

Week 4

Practice Questions

1. Why are modern computers multi-core?

   > *Multiple cores allow PCs to run multiple processes at the same time with greater ease, increasing your performance when multitasking or under the demands of powerful apps and programs.*

2. What is the cache coherence problem? Describe a protocal to resolve this problem.

   > *The cache coherence problem is the challenge of keeping multiple local caches synchronized when one of the processors updates its local copy of data which is shared among multiple caches.*
   > ***MSI protocol** is a simple cache coherence protocol. In this protocol, each cache line is labeled with a state:*
   > ***M**: cache block has been modified. **S**: Other caches may be sharing this block.*
   > ***I**: cache block is invalid*

3. Give an example where we must have a concurrent program instead of a sequenctial program.

   > *In factory, the center controller should handle different works at the same time*

4. Describe how to add two matrices using concurrent threads.

   > *Using a set of threads to calculate different column of the result matrix*

5. Describe how to transpose a matrix using concurrent threads.

   > *Using N(N is the number of columnes) threads to transpose Nth column and row*

6. Can you perform the above matrix transpose operation `in-place`, i.e., without creating or allocating any nticeable extra memory?

> *Using N(N is the number of columnes) threads to transpose Nth column and row(operating in the same matrix)*

7. Describe how to compute matrix multiplication using concurrent threads.

> *Split into submatrices. Compute submatrix multiplications in parallel using threads*

8. Is it possible to compute (and speedup) factorial computation using parallel threads?

> *Computation of factorial can be split into multiple small groups*

9. Give an example of a computational problem where it is not possible to perform parallel computation.

> *n-body problem in physics*

10. Explain why does this program compute unreliably?

```c
void *functionC();
int counter =0;
main(){
    int rc1,rc2;
    pthread_t thread1,thread2;// Two threads execute functionC()
    pthread_create(&thread1,NULL,&functionC,NULL);
    pthread_create(&thread2,NULL,&functionC,NULL);
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    return 0;
}
void *functionC(){
    counter++;
    printf("Counter value: %d\n", counter);
}
```

> *Two threads share the same variable `counter`. It will cause race condition.*

11. Give an example where sharing a pointer to a stack variable by several concurrent threads leads to unreliable program execution.

```c
void *functionC();
int *counter = 0;
main(){
    int rc1,rc2;
    pthread_t thread1,thread2;// Two threads execute functionC()
    pthread_create(&thread1,NULL,&functionC,NULL);
    pthread_create(&thread2,NULL,&functionC,NULL);
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    return 0;
}
void *functionC(){
    *counter++;
```

```
    printf("Counter value: %d\n", *counter);
}
```

Week 6

Additional Exercises

1. How does the distinction between kernel mode and user mode function as a rudimentary form of protection(security) system

   > *All CPU instructions which might affect other users or access devices directly may be only executed in kernel mode. Hence the operating system can prevent user 1 from executing instructions which might affect user 2*

2. Which of the following instructions should be privileged

   1. Set value of timer

      > *Yes, every process need to use timer (so it will affect a lot)*

   2. Read the clock

      > *No, write need privileged but read doesn't*

   3. Clear memory

      > *No, if it clears the `OWN` memory don't need privileged, otherwise, yes*

   4. Issue a trap instruction

      > *Yes, except the one used for issuing a system call, must not be privilieged*

   5. Turn off interrupts

      > *Yes, without interrupts the system may out of control*

   6. Modify entries in device-status table

      > *Yes*

   7. Switch from user to kernel mode

      > *Yes, except the ones used in connection with system calls, should be unprivilieged*

   8. Access I/O device

      > *Yes*

3. Why is it important not to have memory leaks in the kernel?

   > *The kernel never stops and has no grabage collector. Hence any memory lost to a memory leak will not be reclaimed(□□)*

4. What is the effect of calling a routine which may send the current process to sleep in interrupt mode?

   > *The interrupt may occur while any process is executing. If the current process is sent to sleep, this process might not be woken up and therefore will never continue.*

Programming Exercise

1. Write a program which creates a new file and writes the numbers from 1 to 100 into it

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
int main(){
    int fileDescriptor;
    int buffer[100];
    int i;
    int res;
    // 0 USER GROUP OTHER
    // Read(R): 4     Write(W): 2    Execute(X):1
    fileDescriptor = open("myFile.txt", O_RDWR | O_CREADT, 0644);
    if(fileDescriptor < 0){
        // print error message
        perror("Opening failed");
        exit(1);
    }
    for (i = 0; i < 100; i++){
        buffer[i] = i + 1;
    }
    res = write(fileDescriptor, buffer, sizeof(int) * 100);
    if(res < 0){
        perror("writing failed");
        close(fileDescriptor);
        exit(1);
    }
    printf ("%d bytes written\n", res);
    res = close(fileDescriptor);
    if(res < 0){
        perror("closing failed");
        exit(1);
    }
    return 0;
}
```

Week 7

Additional Exercises

1. MCQS

    1. The mapping of a logical address to a physical address is done in hardware by `A`. `A) memory management unit(MMU)`
    B) memory address register

C) relocation register
D) dynamic loading register

> *A register is not sufficient for this mapping. So we need a hardware which is MMU*

2. In a dynamically linked library, `D`. A) loading is postponed(□□) until execution time
   B) system language libraries are treated like any other object module
   C) more disk space is used than in a statically linked library
   `D) a stub is included in the image for each library-routine reference`

   > *The stub checks whether the library code is already in main memory and loads it if not*

3. An address generated by a CPU is referred to as a `B`. A) physical address
   `B) logical address`
   C) post relocation register address
   D) memory management unit(MMU) generated address

   > *The logical addresses are translated by the MMU into physical address*

4. Consider a logical address with a page size of 8 KB. How many bits must be used to represent the page offset in the logical address? `C`
   A) 10
   B) 8
   `C) 13`
   D) 12

   > *2^13 = 8192*

5. Consider a logical address with 18 bits used to represent an entry in a conventional page table. How many entries are in the conventional page table? `A`
   `A) 262144`
   B) 1024
   C) 1048576
   D) 18

   > *We need 2^18 = 262144 entries*

6. What is the context switch time, associated with swapping, if a disk drive with a transfer rate of 2MB/s is used to swap out part of a program this is 200KB in size? Assume that no seeks are necessary and that the average latency is 15ms. `B`
   A) 300
   `B) 115`
   C) 155
   D) None of the above

   > *The answer is 115 ms - 100ms for the data transfer and 15ms latency*

2. [Part 1] Consider a paging system with the page table stored in memory.

   1. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

      > *400 nanoseconds - we need to acces main memory twice*

2. Suppose we have associative registers where finding a page-table entry (if present) takes zero time, and which contains 75 of all page-table references. What will be the effective memory reference time in this case

   > *200 ns + 0.25*200 ns = 250 ns*

3. Suppose the time to find an entry in associative registers in 2 ns. What will be the access time?

   > *250 ns + 0.75 * 2 ns = 251.5 ns*

3. [Virtual Memory]. Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at 4. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening?
   A) CPU utilization 13 percent; disk utilization 97 percent

   > *Thrashing: the CPU is `swapping pages` to and from disk.*

   B) CPU utilization 87 percent; disk utilization 3 percent

   > *At least one `CPU-intensive` programming is running.*

   C) CPU utilization 13 percent; disk utilization 3 percent

   > *`Underused` system with spare capacity.*

Programming Exercise

1. Extend the device driver such that writing to the device sets the counter to the value written, which is an integer. You will have to decide whether you want to write integers as strings or as numbers to the kern

```
// Int
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
    if (len < sizeof(int)) {
    printk(KERN_INFO "Integer expected\n");
    return -EINVAL;
    }

    if (copy_from_user(&counter, buff, sizeof(int))) {
    printk(KERN_ALERT "Copying from user failed.\n");
    return -EPERM;
    }
    printk(KERN_INFO "New counter value is %d\n", counter);
    return len;
}

//String
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{

    char *kbuff;
```

```
    printk(KERN_INFO "Buffer length = %ld\n", len);
    kbuff = kmalloc(len+1, GFP_KERNEL);
    if (kbuff == NULL) {
    printk(KERN_ALERT "Cannot allocate kernel memory\n");
    return -ENOMEM;
    }
    if (copy_from_user(kbuff, buff, len)) {
    printk(KERN_ALERT "Copying from user failed.\n");
    kfree(kbuff);
    return -EPERM;
    }
    kbuff[len] = '\0'; // prevent buffer overflow
    if (kstrtol(kbuff, 10, &counter)) {
    printk(KERN_INFO "Not an integer: %s\n", kbuff);
    kfree(kbuff);
    return -EINVAL;
    }
    kfree(kbuff);
    return len;
}
```

Week 8

Additional Exercises

1. Is it a good idea to have access to peripheral(□□) devices as part of a critical section

   > *No - the access operation can take some time, and may even sleep.*

2. Are race conditions possible if there is only one process runing at the same time?

   > *Yes - because the scheduler might re-empt one process and run another one.*

3. Can the C-compiler find race conditions?

   > *No - race conditions cannot be prevented by checking the syntax of a program.*

4. Is it possible for testing to show the absence of race conditions?

   > *No - as it is impossible to create all posible sequences in which programs can be executed.*

5. A program has no deadlock when only function 1 or function 2, but never function 1 and function 2, are executed. Can you guarantee that the program still has no deadlock when function and function 2 are both executed?

   > *No - deadlock is a `global property`. You will need to check function1 and function2 together to rule out deadlocks*

Week 9

Additional Exercises

1. The __ of a process contains temporary data such as function parameters, return addresses, and local variables. `D`
   A) text section
   B) data section
   C) program counter
   `D) stack`

   > *The text section contains the program code The data section contains the heap The program counter contains the address of the next insturction which is to be executed*

2. The list of processes waiting for a particular I/O device is called a(n) __? `B`
   A) standby queue
   `B) device queue`
   C) ready queue
   D) interrupt queue

   > ***Job queue**: set of all processes in the system*
   > ***Ready queue**: set of all processes residing in main memory, ready an waiting to execute*
   > ***Device queues**: set of processes waiting for an I/O device*

3. The __ refers to the number of processes in memory. `B`
   A) process counter
   `B) degree of multiprogramming`
   C) CPU scheduler

   > ***process counter**: a register in a computer processor that contains the address(location) of the instruction beging executed at the current time.*
   > ***degree of mutiprogramming**: The maximum number of processes that a single-processor system can accommodate efficiently.*
   > ***CPU scheduler**: it will decide which of the ready, in-memory processes is to be executed(allocated a CPU) after an interrupt or system call.*

4. When a child process is created, which of the following is a possibility in terms of the execution or address spce of the child process? `D`
   A) The child process runs concurrently with the parent.
   B) The child process has new program loaded into it
   C) The child is a duplicate of the parent
   `D) All of the above`

   > *All possibilities exist and are useful*

5. A __ saves the state of the currently running process and restores the state of the next process to run. `C`
   A) save-and-restore
   B) state switch
   `C) context switch`
   D) none of the above

   > ***context switch** is the process of storing the the state of a process or thread, so that it can be restored and resume execution at a later point*

6. A process may transition to the Ready state by which of the following actions? `D`
   A) Completion of an I/O event

B) Awaiting its turn on the CPU
C) Newly-admitted process
`D) All of the above`
) All transitions exist and are useful

7. A process that has terminated, but whose parent has not yet called wait(), is known as a __ process. `A`
   `A) zombie`
   B) orphan
   C) terminated
   D) init

   > ***zombie process**: Parent do not call `wait()` for exit(), so its entries are still in the process table*
   > ***orphan process**: Parent completes before the complement of the child process*

8. The __ process is assigned as the parent to orphan process. `B`
   A) zombie
   `B) init`
   C) main
   D) renderer

   > *In newer UNIX systems this can also be the systemd process*

9. When a process creates a new process using the fork() operation, which of the following state is shared between the parent process and the child process? `C`
   A) Stack
   B) Heap
   `C) Shared memory segments`

   > *Stack and Heap are copied when the fork system call is executed*

10. Which of the following scheduling algorithms could result in starvation? `B,D`
    A) First-come, first-served
    `B) Shortest job first`
    C) Round robin
    `D) Priority`

    > *First-come, first-served cannot result in starvation if pre-emption is possible*

11. The total number of child processes created by the following code(excluding Parent) is __. `B`
    A) n
    `B) 2^n-1`
    C) 2^n
    D) 2^(n+1)-1

    > *Each iteration doubles the number of processes created*

12. True/False

    1. All processes in UNIX first translate to a zombie process unpon termination. `True`
    2. The difference between a program and a process is that a program is an active entity while a process is a passive entity. `False`
    3. The exec() system call creates a new process. `False`

Week 10

Additional Exercises

1. Discuss how performance optimisations for file systems might result in difficulties in maintaining the consistency of the systems in the event of a computer crashes.

   > *One very useful optimisation is to `store frequently` used directory entries in main memory and write them only periodically to the disk.*
   > *One of those frequently used directories is `the root` of the file system. If the computer crashes before these directory entries have been written back to disk, the information on the disk is inconsistent*

2. In what situations would using main memory for storing the content of a disk(called a RAM disk) be more useful than using it as a disk cache.

   > *If the disk only contains temporary information which is not required after a restart, it doesen't matter if the information is lost when the system crashes. Therefore if enough RAM is available, it is very efficient to store the whole disk in RAM and only operate in RAM. This is ture (e.g. for /tmp on a UNIX system)*

3. Some systems automatically open a file when it is references for the first time, and close the file when job terminates. Discuss the advantages and the disadvantages of this scheme. Compare it to the more traditional one, hwere the user has to open and close the file explicitly.

   > *Opening and closing a file saves explicit system calls in user space and therefore makes programming easier.*
   > *However, it is less efficient, as the overhead of making the device accessible and inaccessible is incurred for every read and write-operation.*
   > *In addition, the kernel can maintain internal state(ie the current position in a file) when separate open and close-calls are used. If they are not used, the user has to maintain this internal state.*

4. Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

   > *If the same storage is used for a new file, then inconsistencies might arise. If a file with the same absolute path name is re-created, it will be used as the target for the link. One possible solution is the way symbolic links are implemented in UNIX: A link points to a file identified by a filename, not an area on disk. If the file the link points to does not exist, all operations on this link fail. If a new file with the name the link points to is created, it is automatically used.*

Mock Exam

Question 1

1. Will there be any memory leakage in the following program? Explain your answer.

```c
int main()
{
    int *A = (int *) malloc(sizeof(int));
    scanf("%d", A);
    int *B;
    B = A;
    free(B);
    return 0;
}
```

> *The program will not leak memory*
> □□□□□ *(2 marks)*
> *The allocated memory-block is pointed by A and later by both A and B. Hence, both A and B contain the same address. free(B) is the same as free(A) and hence there will be no memory leak.*
> □□□□□□*A -> □□A□B□□□□□□□ -> □□free(B)□□free(A) -> □□□□□□□□ (4 marks)*

2. A programmer has written the following function with the aim to return a pointer to an array of 10 random integers (int) to a caller function. There is a serious problem with this code. Expain what is wrong, why it is a problem, and how it can be fixed. Use this to write a correct version of the function without changing the function-signature. Assume that the caller function of `randomArray` is responsible for freeing any memory occupied by the array.

```c
int* randomArray(void)
{
    int array[10], i;
    for (i = 0; i < 10; i++)
        array[i] = rand();
    return &array[0];
}
```

> *The array is allocated in the stack frame of randomArray(), hence it is a local object. The array object is only in scope during the excution of the function. At the completion of a randomArray() function call, the stack frame will be deallocated and the local array object will be out of scope.*
> □□□□
> *If the function returns a pointer to an out-of-scope array object, then the pointed data will not be reliable and hence the program output will not be reliable*
> □□□□ *(3 marks)*
> *Instead of creating a local array object in the stack fram, the function should allocate the array in the heap and then return a pointer to the heap-based array.*
> □□□□

```c
int* randomArray(void)
{
    int *array, i;
    array = (int *) malloc (10*sizeof(int));
    for (i = 0; i < 10; ++i)
        array[i] = rand();
```

```
    return array;
}
```

□□□□ *(4 marks)*

3. Consider the following two C functions `sum2Darray1` and `sum2Darray2`. Both of them compute the sum of all the elements of an input 2-demesional matrix. Which one of them will be able to exploit memory hierarchy and thus achieve faster computation time? Explain your answer.

```
int sum2Darray1(int a[N][M])
{
    int i, j, sum = 0;
    for(i=0;i<m;i++)
        for(j=0;j<N;j++)
            sum = sum + a[j][i];
    return sum;
}

int sum2Darray2(int a[N][M])
{
    int i, j, sum = 0;
    for(i=0;i<m;i++)
        for(j=0;j<N;j++)
            sum = sum + a[i][j];
    return sum;
}
```

Modern computers with memory hierarchy try to speed up computation by applying the principles of temporal and spatial locality. Thus, when a program tries to read one int object from the main memory, other adjacent int objects are also brought to the cache.
□□□□□*Memory hierarchy*□*temporal,spatial locality (2 marks)*
The function `sum2Darray1` *reads the matrix elements along the columns. Whereas* `sum2Darray2` *reads the matrix elements along the rows. Since, the C programming language stores a 2D matrix in the row-major order,* `sum2Darray2` *offers better spatial locality compared to* `sum2Darray1`*, and thus offers better performance.*
□□ *(5 marks)*
Hence, Strategy2 will be more efficient
□□

Question 2

1. The question is about Main and Virtual Memory. Provide a brief answer.

    1. Why does a processor have a set of registers in addition to a large main memory?

        Registers are normally at the top of the memory hierarchy, and provide the fastest way to access data.

    2. A scheduler controls the degree of multi-programming in an Operaing System. The schedular can send a process to which states(s)?

> *Ready, waiting*

3. Does adding more frames during Page Replacement always lead to improved performance?

   > *No. Adding more frames may decrease performance. Belady's Anomaly*

4. A system is running with following measure behaviour: CPU utilization 10%; Paging disk 95% Other I/O devices 3%. Explain which of the following actions will improve CPU utilization and why?

   1. Install more main memory
   2. Install a faster disk
   3. Changing the degree of multi-programming

   > *All three actions will improve CPU utilization.*
   > *-With more main memory, more pages can stay in main memory and less paging to or from the disk is required.*
   > *-Faster disk means faster response and more throughput, so the CPU will get data more quickly.*
   > *-Decreasing the degree of multi-programming will help.(Less context switches)*

2. Briefly describe what the possible consequences are of a buffer overflow in the kernel

   > *Any answer that would correctly describe that arbitrary data may be overwritten, hence any process or even the kernel may be corrupted and crash.*

3. Consider the following piece of kernel code. The intention is that whenever data is written to a proc-file, this data is written to a device. The device provides two functions: `start transfer` starts the transfer of count bytes to the device and returns immediately, and `transfer finished`, which is called by the device when the data transfer is finished. The function `kernelWrite` should return the number of bytes transferred to the device.

```
int total˙transferred = 0;
/* total number of bytes transferred since module loaded */
int transferred = 0;
/* bytes transferred to device in single transaction */

/* called by device when transfer finished */
/* called in interrupt mode */
void transfer˙finished(int count) {
   transferred = transferred + count;
   /* wakeup waiting process */
}

/* called every time data is transferred to kern
,as a result of writing to proc-file */
int kernelWrite(char * buffer, int count) {
   /* buffer is pointer to user space */
   start˙transfer(buffer, count);
   /* go to sleep until woken up in transfer finish */
```

```
    transferred = transferred + count;
    return transferred;
}
void init module(void) {
    /* set up proc-structure - code omitted */
    proc-¿write˙proc = kernelWrite;
}
```

This kernel code compiles correctly but does not work as intended. Identify these errors and suggest remedies. If you think critical sections are required, it is sufficient to indicate begin and end of a critical section, and whether you would use semaphores or spinlocks to protect the critical section.

> *The errors and fixes are:*
>
> - *The variable `transferred` is increased twice, whereas the variable `total_transferred` is not increased at all. Each variable must be increased once within a critical section. The variable `transferred` should be declared within the `kernelWrite` function and initialised to 0. If a critical section happens wthin `transfer_finished`, spinlocks must be used to protect this critical section. If a critical section happens within `kernelWrite`, semaphores should be used.*
> - *The data must be copied from `buffer`, which is in user space, to a buffer in kernel space eg. via `copy_from_user`.*
> - *The function `kernelWrite` needs to return the number of bytes transferred. In the proposed solution line 20 happens to be OK but this needs to be checked for other solutions.*
>
> *Other solutions are OK as long as they address the problems.*

Question 3

1. Four processes running on a single-core processor (Table 1). Among all the Scheduling Algorithms, briefly explain which one you will prefer and which one you would like to avoid for the given scenario?

Process	Type	Arrival time	Burst time
P1	CPU Bound	0	50
P2	I/O Bound	1	3
P3	I/O Bound	2	4
P4	CPU Bound	3	20

> *We should avoid First Come, First Served (FCFS) Scheduling because processes with short burst time may have to wait for processes with long burst time. We can use either Shortest Job First (SJF) Scheduling, Shortest remaining time first (SRTF), Round Robin (RR).*

2. Predict all possible outputs that the following C program will print to the console and briefly explain your answer. What will be the state of parent process? Briefly explain the behaviour of the program if we comment out the line number 16.

```
#include ¡sys/types.h¿
#include ¡sys/wait.h¿
#include ¡unistd.h¿
#include ¡stdio.h¿
#include ¡stdlib.h¿
int main() {
    pid˙t wpid, child˙pid;
    int status = 0;
    int i;
    for(i = 0; i ¡ 2; i++) {
        if ((child˙pid = fork()) == 0) {
            printf(”process “n”);
        exit(0);
        }
    }
    while ((wpid = wait(&status)) ¿ 0);
    return 0;
}
```

> *If the fork() succeeds, Line 12 will be printed twice. If the fork() fails, then no output will be generated. Parent process will be in wait state. If we comment out line 16, the parent will not wait for the child process to complete (Zombie process).*

3. Your computer system uses Round Robin scheduler and is not very responsive and so you decide to change the scheduling time quantum from 50 msec to 1 msec. Now the performance is even worse. Why is this happening?

> *Any correct answer that would describe that extra context switches are taking place*

4. Consider a concurrent system with two processes A and B (Figure 1). Assume y is a shared variable with value of 5. Describe how a race condition is possible and provide a solution to prevent the race condition from occurring.

> *The assignment operation for y can be interrupted while execution leading to values different than actual value 5 after calling the method increment and decrement in a sequence.*
> *Proposed solution : Make shared variables y and z atomic. Use any lock technique (mutex, semaphores, hardware lock etc).*