Week 4(Threads)

Q1. What is the difference between Fork and Vfork.
Ans: Both **fork()** and **vfork()** are the **system calls** that creates a new process that is identical to the process that invoked fork() or vfork(). Using **fork()** allows the execution of parent and child process simultaneously. The other way, **vfork()** suspends the execution of parent process until child process completes its execution.

The primary difference between the fork() and vfork() system call is that the child process created using fork has separate address space as that of the parent process. On the other hand, child process created using vfork has to share the address space of its parent process.

Q2. Describe the action taken by kernel to context-switch between processes.
Ans: Actions taken by a kernel to context-switch between processes are -

- The OS must save the PC and user stack pointer of the currently executing process, in response to a clock interrupt and transfers control to the kernel clock interrupt handler
- Saving the rest of the registers, as well as other machine state, such as the state of the floating point registers, in the process PCB is done by the clock interrupt handler.
- The scheduler to determine the next process to execute is invoked the OS.
- Then the state of the next process from its PCB is retrieved by OS and restores the registers. The restore operation takes the processor back to the state in which the previous process was previously interrupted, executing in user code with user-mode privileges.

Many architecture-specific operations, including flushing data and instruction caches also must be performed by Context switches.

Q3. Consider the following statements about user level threads and kernel level threads. Which one of the following statement is FALSE?
(A) Context switch time is longer for kernel level threads than for user level threads.
(B) User level threads do not need any hardware support.
(C) Related kernel level threads can be scheduled on different processors in a multi-processor system.
(D) Blocking one kernel level thread blocks all related threads.

Q4. What are the similarities and differences between process and threads?
Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.

- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread are design to assist one other.
- Note that processes might or might not assist one another because processes may originate from different users.

Q5. Compare the actions performed by kernel to context switch among the threads and the processes?

Action of Kernel to Context Switch Among Threads

The threads share a lot of resources with other peer threads belonging to the same process. So a context switch among threads for the same process is easy. It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplished this task.

Action of kernel to Context Switch Among Processes

Context switches among processes are expensive. Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the fol owing information:

The process state.
The program counter, PC.
The values of the dif erent registers.
The CPU scheduling information for the process. Memory management information regarding the process. Possible accounting information for this process.
I/O status information of the process.

When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.

Q 6. When a multi-threaded process receives a signal, to what thread should that signal be delivered?

There are four major options:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals in a process.

- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
- UNIX provides two separate system calls, kill(pid, signal) and pthread_kill(tid, signal), for delivering signals to processes or specific threads respectively.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

Q 7. In the following program the parent send the message "Hello my child" to the child. Modify this program so that the parent creates a second pipe for the child to send the message "Hello my parent" to the parent. Thus the program output should be:

Hello my child
Hello my parent

```
int main(void)
{
      pid_t pid;
      int fd1[2];

      char buf[100];
      pipe(fd1);

      pid = fork();
      if (pid > 0) {
            close(fd1[0]);
            write(fd1[1], "Hello my child\n", 12);
            close(fd1[1]);


      }
      else {
            close(fd1[1]);
            read(fd1[0], buf, 100);
            printf("%s\n",buf);
            close(fd1[0]);
      }
}
```

Solution:

```
int main(void)
{
      pid_t pid;
      int fd1[2];
      int fd2[2];

        char buf[100];

      pipe(fd1);
      pipe(fd2);

      pid = fork();


      if (pid > 0) {
            close(fd1[0]);
            write(fd1[1], "Hello my child\n", 12);
            close(fd1[1]);
```

```
            close(fd2[1]);
            read(fd2[0], buf, 100);
            printf("%s\n",buf);
            close(fd2[0]);
        }
        else {
            close(fd1[1]);
            read(fd1[0], buf, 100);
            printf("%s\n",buf);
            close(fd1[0]);

            close(fd2[0]);
            write(fd2[1], "Hello my parent\n", 13);
            close(fd2[1]);
        }
}
```

Q 8. What is the output of the following program?

```
 int main()
{
      int* var = (int *) malloc(sizeof(int));
      *var = 10;

      pid_t pid = fork();

      if (pid == 0) {
         (*var)++;
         printf("Hello, I  am the child, var=%d\n", *var);
         exit(0);
      }
      wait(NULL);
      printf("Hello, I am the parent, var=%d\n", *var);
}
```

Solution:

Hello, I am the child, var=11
Hello, I am the parent, var=11

Q 9. In Linux thread, clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags. Write the meaning of each flag?

flag	**Meaning**
CLONE_FS	
CLONE_VM	
CLONE_SIGHAND	
CLONE_FILES	

Ans:

flag	**Meaning**
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared

CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

Q 10. What is Process Contention Scope(PCS) and System Contention Scope(SCS). Write their advantages.

1. **Process Contention Scope (PCS) –**
   The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
2. **System Contention Scope (SCS) –**
   The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.

Advantages of PCS over SCS :

- If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.
- PCS is cheaper than SCS.
- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.
- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.
- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces.