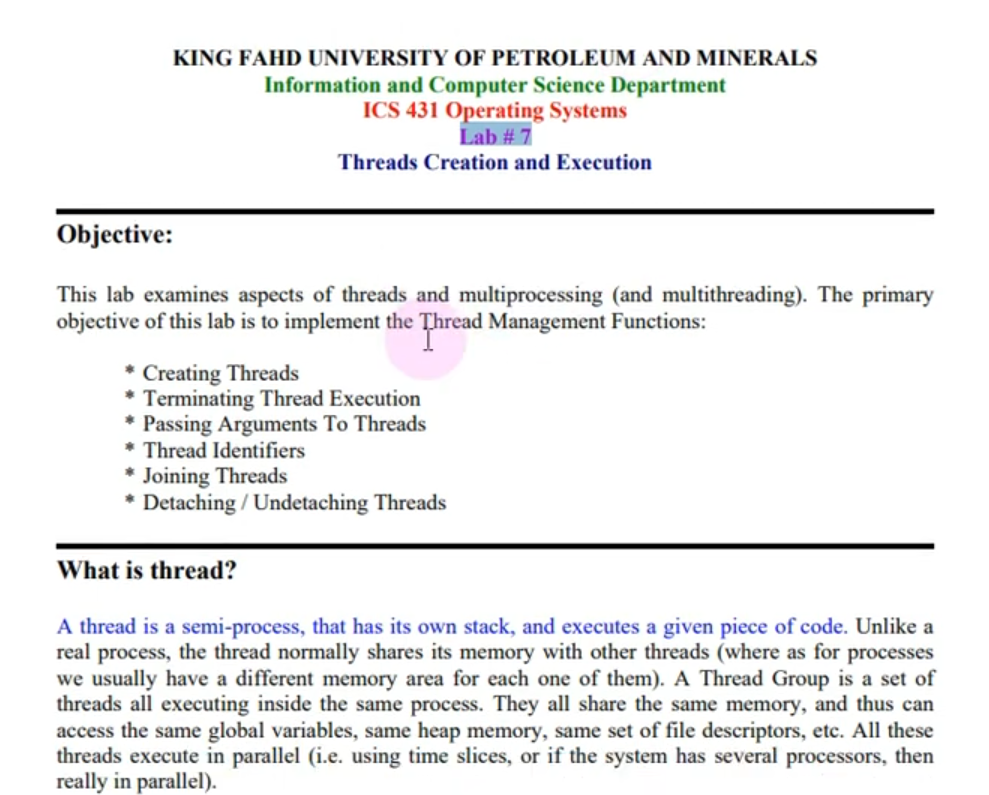
**OS – Sheet 5:**

* Threads.
* **Threads:**



***Questions:*** *What’s the difference between distributing tasks between (Parent & Children) and (Process & Threads)?*

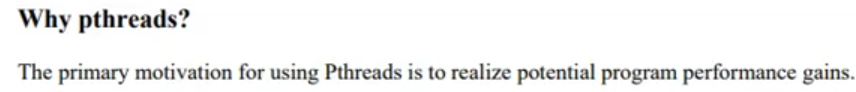
**1. Processes (Parent-Child Concept)**

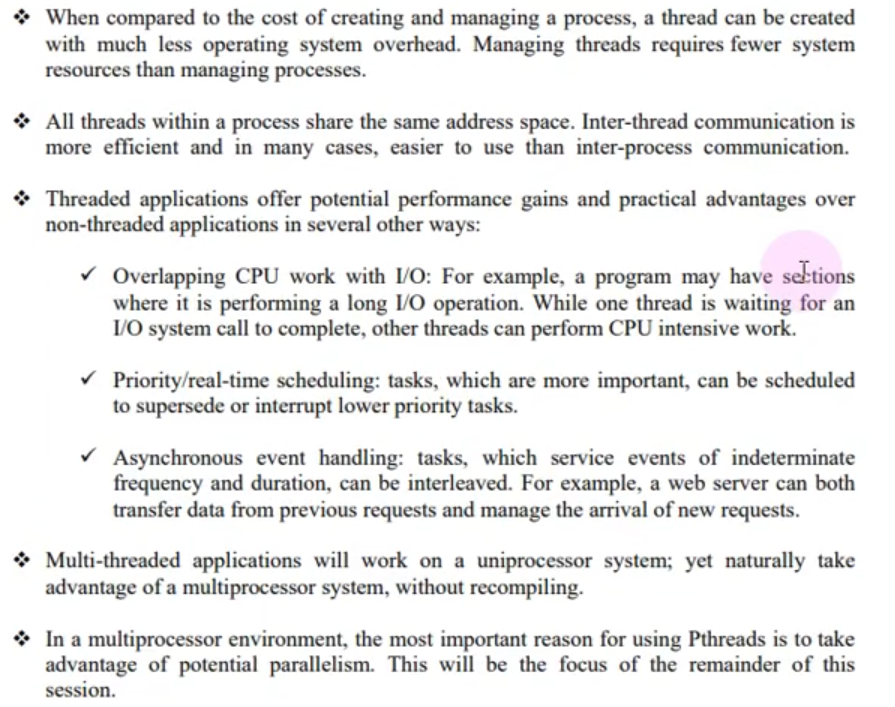
* **Definition**: Processes are independent execution units with their own memory space, file descriptors, and resources.
* **Communication**:
  + Inter-process communication (IPC) is required to share data (e.g., pipes, shared memory, sockets, or message queues).
  + Communication is typically slower due to the overhead context switching and memory separation.
* **Resource Sharing**:
  + Processes don’t share memory directly; each process has its own address space.
  + The parent process may allocate shared resources explicitly (e.g., shared files or memory regions).
* **Fault Isolation**:
  + A crash in a child process does not directly affect the parent or sibling processes.
  + Processes are isolated and independent, making the system more robust.
* **Usage**:
  + Suitable for tasks that require strong isolation or when processes run independently (e.g., web servers spawning child processes for client requests).
* **Examples**:
  + Forking a child process in Unix/Linux using fork ().
  + Running a database as a separate process.

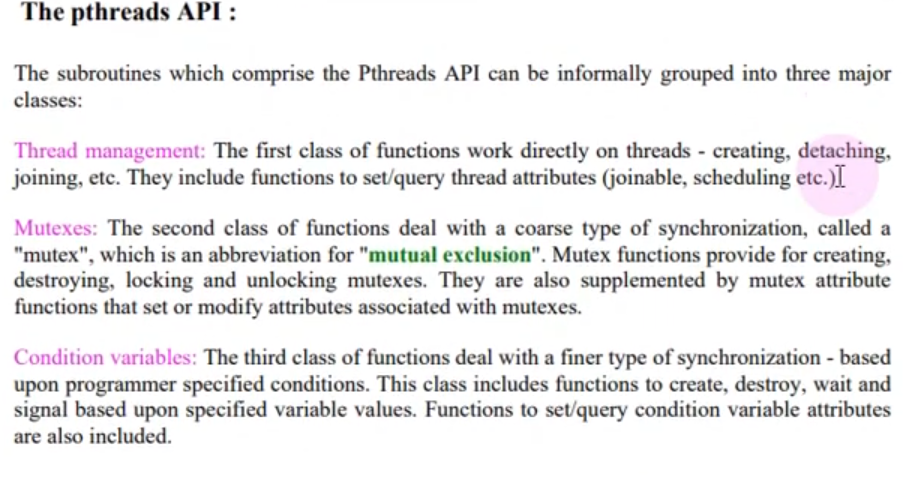
**2. Threads**

* **Definition**: Threads are lightweight execution units within the same process. All threads of a process share the same memory space, file descriptors, and other resources.
* **Communication**:
  + Threads share the same memory space, so data sharing is straightforward (e.g., using global variables or shared objects).
  + No need for IPC mechanisms, leading to faster communication.
* **Resource Sharing**:
  + Threads within a process share the same address space and resources.
  + This makes creating and managing threads less resource-intensive compared to processes.
* **Fault Isolation**:
  + A crash in a thread can bring down the entire process, including all other threads.
  + Threads lack isolation, which can lead to issues like race conditions and deadlocks.
* **Usage**:
  + Suitable for tasks that require frequent communication or shared state (e.g., real-time data processing, GUI applications, or multi-threaded servers).
* **Examples**:
  + Multithreading in C using pthread library.
  + Threads in Java or Python.

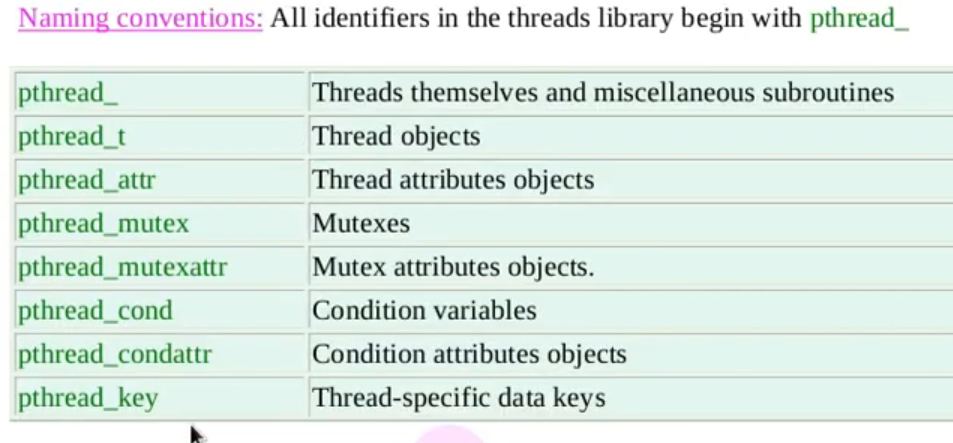
**POSIX**: it refers to Portable Operating System Interfaces



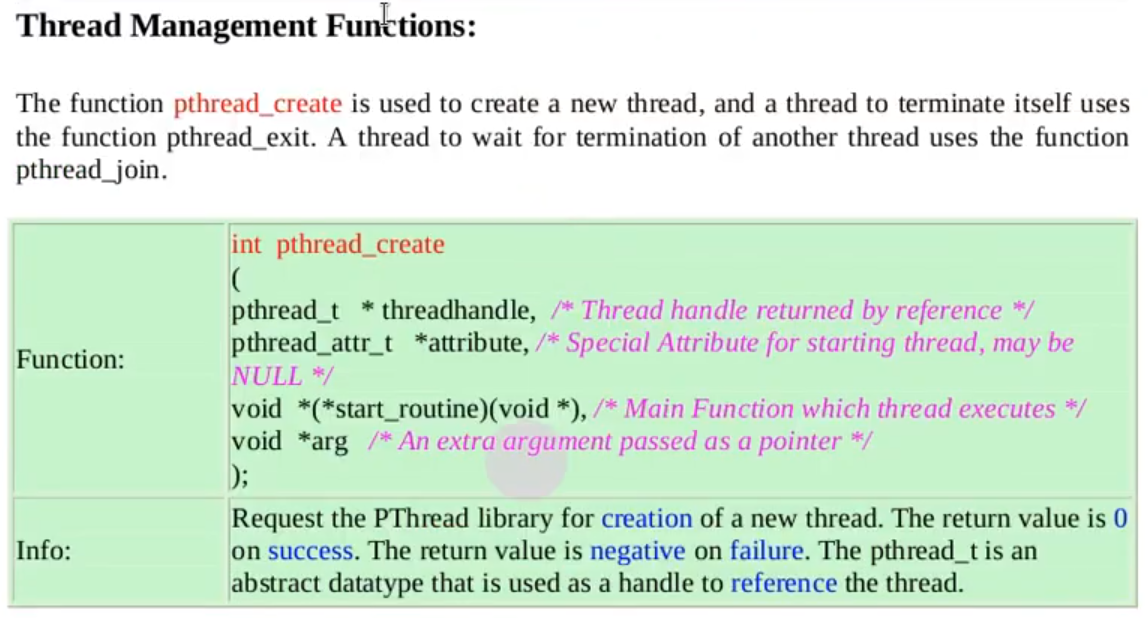


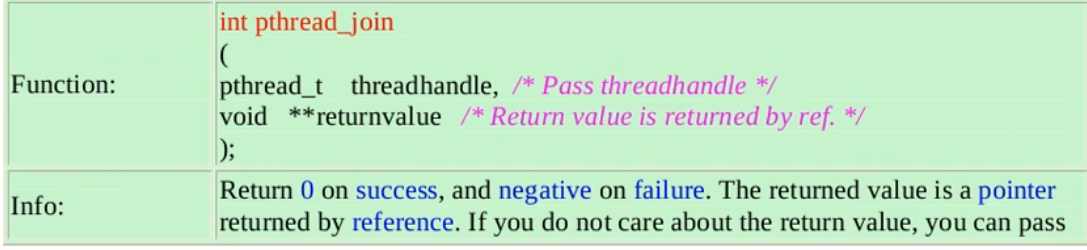


* ***First, keywords:***



* ***Second, functions:***







* To know more about any function of the previous 3 functions: go to CLI and type ***man pthread\_create***.
* ***Very important note:*** join & exit are tied together which means if you use one of them then logically you must use the other one why?
  + ***Exit:*** it’s like ‘return’ keyword in the function which means it’s used to return the result of the thread function ***inside the thread function***, and if you don’t want to return something from this thread then you pass ‘NULL’ to the function exit.
  + ***Join:*** It’s used when you want to wait for a particular thread and return its result ***outside the thread function*** and if want to ignore the output of the thread and just wait for it then the second parameter in the join will be NULL.

* The importance of each one of the 3 functions:
  + **Create:** from its name it’s obvious it’s used to create threads.
  + **Join:** it’s used to wait for a thread to terminate. It plays a crucial role in managing multithreaded programs, ensuring proper synchronization and resource cleanup, it’s mainly used in 3 tasks:
    - **Synchronizing Threads**: pthread\_join allows the main thread (or another thread) to wait for a specific thread to finish its execution. Without it, the program might exit before a thread completes, leading to incomplete work or other issues.
    - **Resource Cleanup**: When a thread terminates, its resources remain allocated until another thread calls pthread\_join. By calling pthread\_join, the resources used by the terminated thread (such as memory, file descriptors, etc.) are properly freed. If you do not call pthread\_join, and you keep creating new threads, your program might run out of resources due to these "zombie threads."
    - **Retrieving Return Values**: If a thread returns a value via pthread\_exit or a return statement, pthread\_join can be used to retrieve that value. This can be helpful when a thread performs some computation, and you need the result.
      * + **Exit: it’**s used to terminate a thread in a controlled manner, ensuring that resources are properly released and return values are passed to other threads if needed. It is an essential function for managing the lifecycle of threads in a multithreaded program, it’s mainly used in these tasks:
    - **Graceful Termination**:
* pthread\_exit allows a thread to exit safely without abruptly terminating the entire process (which would happen if you used exit). When a thread calls pthread\_exit, only that specific thread terminates, while other threads in the process can continue executing.
  + - **Returning Values to pthread\_join**:
* When a thread terminates using pthread\_exit, it can return a value to the thread that called pthread\_join. This is useful for passing results from a thread back to the main program or another thread.

For example, if a thread performs a computation, it can return the result via pthread\_exit, and another thread can retrieve that result using pthread\_join.

* + - **Resource Cleanup for Detached Threads**:
* For detached threads, pthread\_exit ensures that resources associated with the thread (such as memory and thread-local storage) are cleaned up when the thread terminates. Unlike a joinable thread, a detached thread doesn't need to be "joined," so pthread\_exit is critical in preventing resource leaks.
* **Notice:** Your code might be working without using ‘join’ or ‘exit’ **because** the main thread happens to stay alive long enough for the worker threads to complete, but this is not guaranteed behavior. On other systems or with more complex workloads, the absence of pthread\_join or pthread\_exit can lead to unpredictable results, resource leaks, and incomplete thread execution, and to know the detailed info ‘why’ read their importance.

And now the final code must be like that:

A computer code with white text

Description automatically generated

Now when compiling this code, it’s not like any other compilation for any C code.

Instead, you need to add this statement -***l pthread*** to the compilation statement which means link this code with other threads.



A computer code with white text

Description automatically generated

* ***Notice 1:*** the thread function takes an argument from type ***void pointer*** to be able to receive any value from the main function, and pointers make it safer than direct values.
* ***Notice 2:*** inside the function you can make casting like he did in ‘***int \****’.
* ***Notice 3:*** in the previous example, we made the arguments as ordinary pointers but this is not right because, arguments between threads must be stored in heap which means these arguments must be created using heap, maybe we neglected that in this example because we have only one thread but once you have multiple threads you must make their arguments stored in heap, and to do so like we studied in memory lesson, we must create these pointers using dynamic allocation using the keyword ‘***malloc***’ and of course don’t forget to free them after usage using the keyword ‘***free***’ like that:

**A computer screen with text on it

Description automatically generated A screenshot of a computer

Description automatically generated**

Why Heap?

**1. Thread Lifespan:**

The thread's lifespan might extend beyond the scope of the function that created it. If you pass arguments as local (stack-allocated) variables, those variables will be destroyed once the function returns, potentially leading to undefined behavior if the thread tries to access them.

**2. Concurrent Access:**

If multiple threads share the same stack-allocated variable (such as a local variable), this can lead to race conditions. By allocating arguments on the heap, each thread gets its own independent copy, which avoids this issue.

**3. Flexibility in Data Size:**

The heap allows you to dynamically allocate memory based on the size needed at runtime. This can be particularly useful if the size of the data passed to the thread isn't known at compile time. Heap memory is not limited to the stack's relatively small size, so you can handle large amounts of data without worrying about stack overflow.

***Now check these 3 codes:***

[***Pthread management code***](Thread%20management%20codes/Threads%20Management%20code.c) ***.c***

[***Process division to threads 1***](Thread%20management%20codes/Dividing%20process%20into%20threads%20code%201.c) ***.c***

[***Process division to threads 2***](Thread%20management%20codes/Dividing%20process%20into%20threads%20code%202.c) ***.c***

A close-up of a text

Description automatically generated

A white text with black text

Description automatically generated

It’s used to define a set of properties or attributes to the thread or to get them from it, and we have different types of attributes that we can give to out thread like:

A computer screen with white text

Description automatically generated

***Pthread attributes characteristics:***

A black screen shot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

Code: [thread-attribute.c](Threads%20attribute.c)

Paraphrase of some characteristics:

* ***Set-detatch-state:*** it has 2 choices:
  + 1. **PTHREAD\_CREATE\_JOINABLE ‘***the default***’**
* **Meaning**: The thread can be joined later using pthread\_join.
* **Behavior**:
  + The thread remains "joinable" after it finishes execution, which means the parent thread (or another thread) can wait for it to complete using pthread\_join.
  + The thread's resources are not automatically freed after it finishes; they are cleaned up only after pthread\_join is called.
* **Use Case**:
  + Use this when you need to retrieve the thread's exit status or ensure the thread has completed its work before continuing.

**Paraphrased**: The thread is designed to allow other threads to "join" it later, meaning they can synchronize with it and clean up its resources manually.

**2. PTHREAD\_CREATE\_DETACHED**

* **Meaning**: The thread is detached, meaning it automatically cleans up its resources once it finishes.
* **Behavior**:
  + The thread runs independently and cannot be joined.
  + Its resources (stack, memory, etc.) are automatically reclaimed by the system after it terminates.
* **Use Case**:
  + Use this for threads that run independently and don’t require the parent or other threads to wait for their completion or collect their status.

**Paraphrased**: The thread is set up to handle its cleanup automatically, so no other thread needs to manage or wait for it.

* **pthread\_attr\_setschedpolicy**

*Purpose:*

This function sets the **scheduling policy** attribute in a thread attributes object (pthread\_attr\_t). The scheduling policy determines how the thread scheduler decides which thread to run next when multiple threads are competing for CPU time.

*Syntax*

A black background with pink and white text

Description automatically generated

* **attr**: A pointer to the thread attributes object.
* **policy**: The scheduling policy to set, chosen from predefined constants.

*Options for policy*

1. **SCHED\_FIFO (First In, First Out)**:
   * Threads are scheduled in a fixed order based on their priority.
   * Once a thread starts running, it runs until it voluntarily yields, blocks, or finishes.
   * Suitable for real-time systems where strict priority handling is required.
   * **Paraphrased**: Threads are queued by priority and run to completion unless interrupted.
2. **SCHED\_RR (Round Robin)**:
   * Threads with the same priority are scheduled in a circular order, each getting a fixed time slice.
   * Offers time-sharing behavior for threads of equal priority.
   * **Paraphrased**: Threads take turns running for a fixed time if they have the same priority.
3. **SCHED\_OTHER**:
   * Default scheduling policy for non-real-time threads.
   * Scheduling behavior depends on the system’s implementation, often a dynamic time-sharing algorithm.
   * **Paraphrased**: The system uses its default policy for general-purpose thread scheduling.

*Return Value*

* **0**: Success.
* **Non-zero**: An error occurred (e.g., invalid scheduling policy or insufficient privileges).
* **pthread\_attr\_setschedparam**

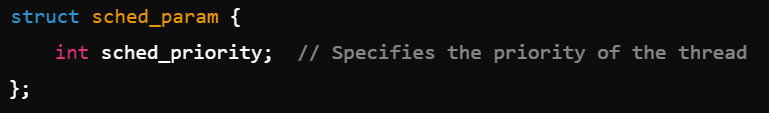
*Purpose*

This function sets the **scheduling parameters**, such as priority, in a thread attributes object (pthread\_attr\_t). Scheduling parameters are applied within the context of the selected scheduling policy.

*Syntax*



* **attr**: A pointer to the thread attributes object.
* **param**: A pointer to a sched\_param structure that holds the scheduling parameters.



*sched\_priority:*

* + Specifies the priority level of the thread.
  + The valid range depends on the scheduling policy and system configuration.
  + Higher values generally indicate higher priorities.

*Interaction with pthread\_attr\_setschedpolicy*

* The **sched\_priority** must be compatible with the policy set by pthread\_attr\_setschedpolicy.
  + For example, with **SCHED\_OTHER**, the priority might be ignored.
  + With **SCHED\_FIFO** and **SCHED\_RR**, priority directly affects execution order.

*Return Value*

* **0**: Success.
* **Non-zero**: An error occurred (e.g., invalid priority or attributes object).