Thread Creation and PID Management in GNU/Linux: `pthread-pid.c` Example

The `pthread-pid.c` program demonstrates thread creation in GNU/Linux and the retrieval of the thread's PID. Since both `main()` and `fonction_thread()` run continuously, we can closely observe their execution. This program must be compiled with the `-lpthread` library.

Code Example:

```c
#include <unistd.h>  // for sleep
#include <pthread.h> // pthread_create, pthread_join, pthread_exit
#include <stdio.h>

void *fonction(void *arg)
{
    printf("Thread child PID = %d\n", (int)getpid());

    while(1);  // Runs forever
    return NULL;
}

int main()
{
    pthread_t thread;

    pthread_create(&thread, NULL, fonction, NULL);

    printf("Main PID = %d\n", (int)getpid());

    while(1);  // Runs forever

    return 0;
}
```

Compilation and Execution: To compile and run `pthread-pid.c` in the background:

```
MonPc > gcc -o pthread-pid thread-pid.c -lpthread
MonPc > pthread-pid &
[1] 24133
MonPc > Main PID = 24133
Thread child PID = 24136
```

Displaying Threads with `ps`:

```
MonPc > ps x
PID     TTY   STAT TIME COMMAND
23928   pts/2 S    0:00 -tcsh
24133   pts/2 R    0:53 pthread-pid
24135   pts/2 S    0:00 pthread-pid
24136   pts/2 R    0:53 pthread-pid
24194   pts/2 R    0:00 ps x
```

---

1. Scheduler Activations

Scheduler activations aim to mimic the performance of kernel threads while preserving the lightweight nature of user threads.

- **Key Concept**: If a thread is blocked on a system call or a page fault, another ready thread should be executed.
- If a thread is blocked by another thread, there is no need for kernel intervention, avoiding performance loss caused by transitions between kernel and user space.

2. Scheduler Activations (Principle)

- The kernel assigns a number of virtual processors to each process. Then, the execution system allocates these virtual CPUs to threads.
- Virtual processors may or may not correspond to actual CPUs in the case of multiprocessor systems.
- A process initially starts with one virtual processor but can request more or return them if not needed. The kernel can also reclaim processors when necessary.

3. Scheduler Activations (Operation)

When the kernel detects a thread blockage, it notifies the execution system by sending the thread ID and a description of the error.

- **Mechanism**: The kernel must activate the execution system at a known starting address, a mechanism known as **upcall**.
- Upon activation, the execution system marks the thread as blocked and selects another ready thread for execution.
- When the kernel determines that the blocked thread is ready to resume, it performs another upcall to notify the system that the thread can resume execution.

4. Scheduler Activations (Interrupt Handling)

In the case of a hardware interrupt, the concerned processor switches to kernel mode, and the thread is blocked. There are two possible cases:

1. **Unrelated Interrupt**: If the interrupt concerns another process (e.g., I/O for another process), the thread is restored to its state prior to the interruption.
2. **Related Interrupt**: If the process is involved (e.g., a page arrives for one of its threads), the thread remains blocked. The execution system takes control of the processor and chooses to either resume the interrupted thread or switch to another ready thread.

Summary

In GNU/Linux, thread management and process scheduling involve intricate interactions between user-space threads and kernel-level mechanisms. With the use of virtual processors and techniques such as upcalls, systems can efficiently manage thread execution while minimizing unnecessary kernel involvement. However, various scenarios, such as thread blockage and hardware interrupts, require careful coordination to ensure smooth thread execution. The `pthread-pid.c` example provides a practical demonstration of thread creation and PID management in a multithreaded environment.