DISCLAIMER

What is reentrant and example

In the context of the Linux kernel, "reentrant" refers to code that can be safely interrupted and then resumed without causing unexpected behavior or data corruption. Reentrant code is designed to handle multiple simultaneous invocations, typically by using local variables or data structures that are not shared between invocations.

The concept of reentrant code is particularly important in the kernel, as it is a multi-threaded environment where multiple processes and interrupt handlers can run concurrently. Non-reentrant code could lead to race conditions, data corruption, or other unpredictable outcomes when multiple invocations attempt to access shared resources simultaneously.

To understand reentrant code better, let's consider an example. Suppose we have a kernel function that calculates the factorial of a given number. A non-reentrant implementation might use a static variable to store the intermediate result, like this:

```
int factorial(int n) {
  static int result = 1;

  if (n == 0 || n == 1)
    return result;

  result *= n;
  return factorial(n - 1);
}
```

In this non-reentrant implementation, the result variable is shared among all invocations of the factorial() function. If two invocations of factorial() were running simultaneously, they would both read and modify the same result variable, leading to incorrect results.

To make the code reentrant, we can modify it to use a local variable to store the intermediate result, like this:

```
int factorial(int n) {
  int result = 1;

  if (n == 0 || n == 1)
    return result;

  result *= n;
  return factorial(n - 1);
}
```

In this reentrant implementation, each invocation of the factorial() function has its own independent result variable. Therefore, multiple invocations can run concurrently without interfering with each other's computations.

By designing code to be reentrant, the Linux kernel ensures that different processes, threads, or interrupt handlers can safely call the same functions without causing conflicts or unexpected behavior. This is crucial for maintaining the stability, reliability, and security of the operating system.