题目1：使用fork函数，设计并实现WebServer以支持多进程并发处理众多客户端的请求。

要使用fork函数支持多进程并发处理客户端请求，可以在main函数中创建一个循环，为每个传入的连接fork一个新进程。以下是修改后的代码：

```
#include <sys/wait.h>
// ... (其余的代码)
int main(int argc, char **argv)
{
  // ... (现有的代码)
  for (hit = 1;; hit++)
  {
    length = sizeof(cli_addr);
    if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr, &length)) < 0)
      logger(ERROR, "system call", "accept", 0);
    // Fork一个新的进程
    if ((pid = fork()) < 0)
    {
      logger(ERROR, "system call", "fork", 0);
    }
    else if (pid == 0) // 子进程
    {
      close(listenfd); // 子进程不需要监听器
      // 处理请求
      web(socketfd, hit);
      exit(0); // 子进程处理完请求后退出
    }
    else // 父进程
    {
      close(socketfd); // 父进程不需要已接受的套接字
    }
  }
}

```

题目2：使用信号量、共享内存等系统接口函数，来统计每个子进程的消耗时间以及所有子进程消耗时间之和。

在下面这个修改后的代码中，每个子进程在处理完请求后，会记录开始和结束时间，并将消耗时间加入到共享内存中，使用POSIX信号量保护共享内存的写操作，以防止多个子进程同时写入。同时，父进程在最后也会统计所有子进程的总消耗时间。

```
#include <semaphore.h>
#include <fcntl.h>
#include <sys/mman.h>

// 全局变量
sem_t *semaphore;
long long *total_time; // 共享内存

// ... (其余的代码)

int main(int argc, char **argv)
```

```
{
  // ... (现有的代码)

  // 初始化POSIX信号量
  semaphore = sem_open("/my_semaphore", O_CREAT | O_EXCL, 0644, 1);
  if (semaphore == SEM_FAILED)
  {
    perror("sem_open");
    exit(EXIT_FAILURE);
  }

  // 初始化共享内存
  int shm_fd = shm_open("/my_shared_memory", O_CREAT | O_RDWR, 0644);
  if (shm_fd == -1)
  {
    perror("shm_open");
    exit(EXIT_FAILURE);
  }
  ftruncate(shm_fd, sizeof(long long));
  total_time = (long long *)mmap(NULL, sizeof(long long), PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
  if (total_time == MAP_FAILED)
  {
    perror("mmap");
    exit(EXIT_FAILURE);
  }

  // ... (现有的代码)

  for (hit = 1;; hit++)
  {
    // ... (现有的代码)

    // Fork一个新的进程
    if ((pid = fork()) < 0)
    {
      logger(ERROR, "system call", "fork", 0);
    }
    else if (pid == 0) // 子进程
    {
      close(listenfd); // 子进程不需要监听器

      // 记录开始时间
      long long start_time = current_timestamp();

      // 处理请求
      web(socketfd, hit);

      // 记录结束时间
      long long end_time = current_timestamp();

      // 计算消耗时间
      long long elapsed_time = end_time - start_time;

      // 打印每个子进程的消耗时间
      printf("Child process %d: %lld microseconds\n", getpid(), elapsed_time);

      // 使用POSIX信号量保护共享内存的写操作
```

```
      sem_wait(semaphore);
      *total_time += elapsed_time;
      sem_post(semaphore);

      exit(0); // 子进程处理完请求后退出
    }
    else // 父进程
    {
      close(socketfd); // 父进程不需要已接受的套接字
      waitpid(-1, NULL, WNOHANG); // 非阻塞等待回收僵尸进程
    }
  }

  // ... (现有的代码)
  // 清理POSIX信号量
  sem_close(semaphore);
  sem_unlink("/my_semaphore");

  // 清理共享内存
  munmap(total_time, sizeof(long long));
  shm_unlink("/my_shared_memory");

  return 0;
}

```

```
➜  OS-project git:(main) ✗ ./webserver 8080 ./nwebdir
Child process 1573545: 1001372 microseconds
Child process 1573546: 1000324 microseconds
Child process 1573552: 1000591 microseconds
Child process 1573553: 1000602 microseconds
^C
Total processing time for all child processes: 4002889 microseconds
```

- 问题记录：/usr/bin/ld: webserver.o: in function `main': webserver.c:(.text+0x7b7): undefined reference tosem_open' /usr/bin/ld: webserver.c:(.text+0x7f6): undefined reference to `shm_open' /usr/bin/ld
- 解决方案：

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -pthread -lrt
```

- 问题记录： 如何通过Ctrl+C来退出服务，并在退出时输出所有子进程的消耗时间之和。
- 可以使用信号处理机制来捕获Ctrl+C（SIGINT）信号，并在信号处理函数中设置退出标志。然后，可以在主循环中检查该退出标志，以便在Ctrl+C被触发时退出服务并输出所有子进程的消耗时间之和。

题目3：使用http_load来测试当前设计的多进程WebServer服务性能

```
➜  http_load-12mar2006 git:(main) ✗ ./http_load -parallel 10 -fetches 1000 urls.
txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.160759 seconds
260 mean bytes/connection
6220.49 fetches/sec, 1.61733e+06 bytes/sec
msecs/connect: 0.080291 mean, 4.136 max, 0.013 min
msecs/first-response: 1.47173 mean, 12.292 max, 0.139 min
HTTP response codes:
  code 200 -- 1000
```

其比单进程Web服务性能提高的原因：

1. **并行处理请求：** 多进程允许同时处理多个请求，而不是一个接一个地处理。这可以显著提高并发性能，尤其是在有大量同时到达的请求时。
2. **系统资源的更好利用：** 多进程可以更好地利用多核系统的资源。每个进程都可以在一个独立的CPU核上执行，从而提高整体性能。
3. **防止阻塞：** 单进程模型可能会因为一个长时间运行的请求而阻塞其他请求的处理。而使用多进程，每个请求都在独立的进程中执行，一个进程的阻塞不会影响其他进程。

在多进程Web服务器的情况下，潜在的瓶颈和优化机会：

1. **进程间通信开销：** 在多进程模型中，进程间通信（IPC）可能引入一些开销。这包括通过共享内存传递数据、使用信号量进行同步等。你可以考虑优化进程间通信的方式，或者尽量减少它们的频率。
2. **文件系统访问：** 如果Web服务器在处理请求时频繁地访问文件系统，文件系统的性能可能成为瓶颈。使用缓存机制、减少对磁盘的访问次数，以及使用更高性能的文件系统都是可能的优化点。
3. **共享资源竞争：** 如果多个进程竞争共享资源（如共享内存区域），可能会引起性能问题。使用适当的同步机制，如互斥锁，来避免竞争条件。