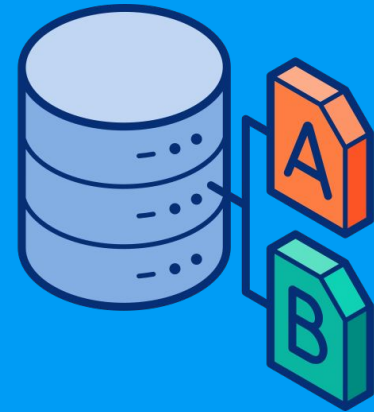


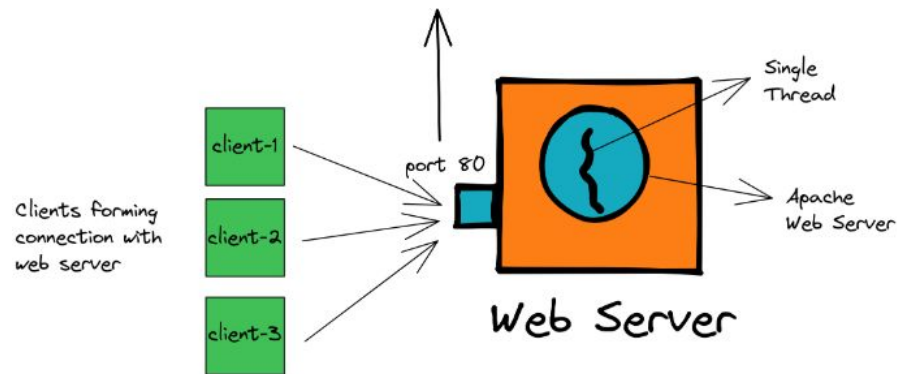
# Concurrencia - Laboratorio #03

Brandon Duque Garcia  
Jonathan A Granda Orrego

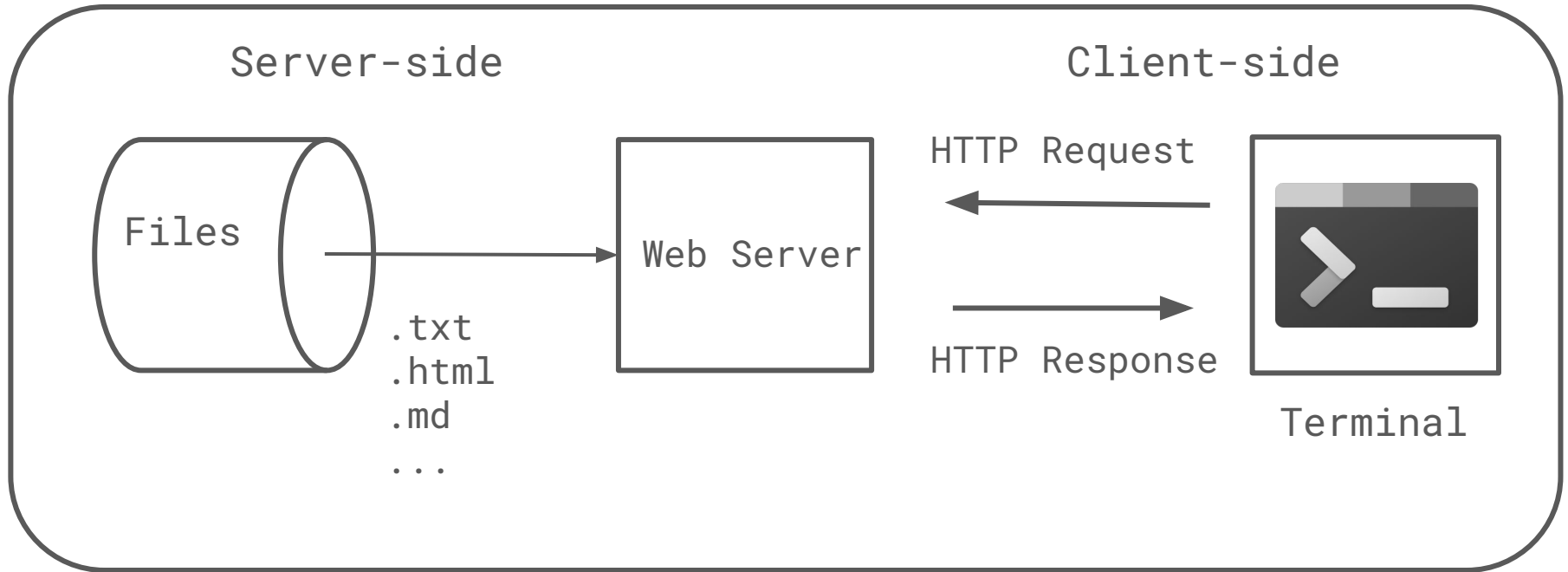


# Implementación de un Web Server Concurrente

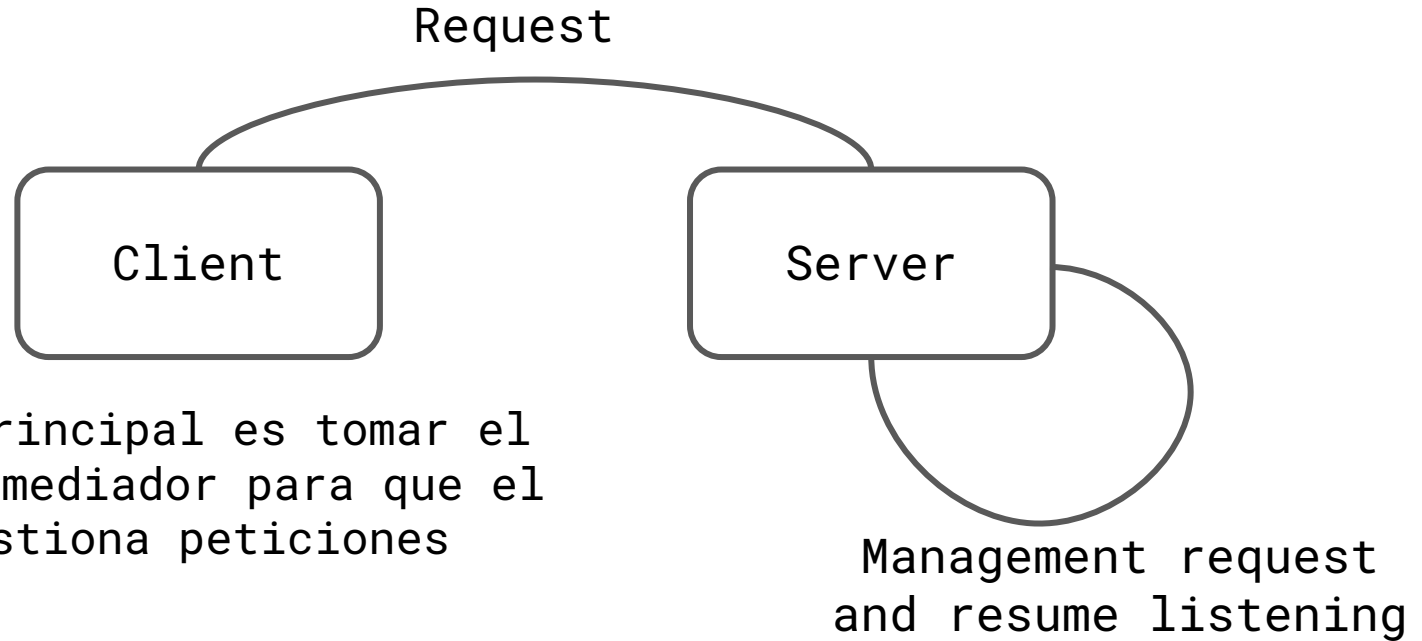
Se desarrollara un servidor web concurrente, partiendo de un código de un servidor no concurrente, opera con un solo hilo, y se tiene como trabajo hacer que el servidor sea multihilo para poder manejar múltiples peticiones al mismo tiempo



# Esquema

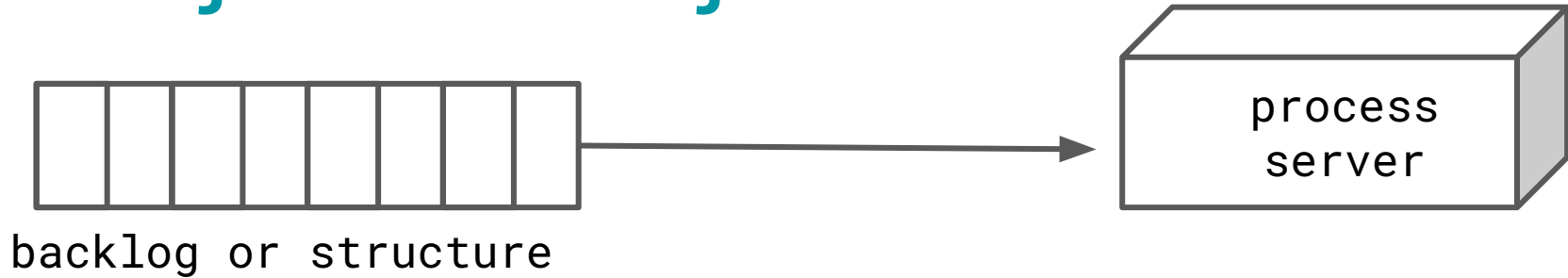


# Flujo de trabajo

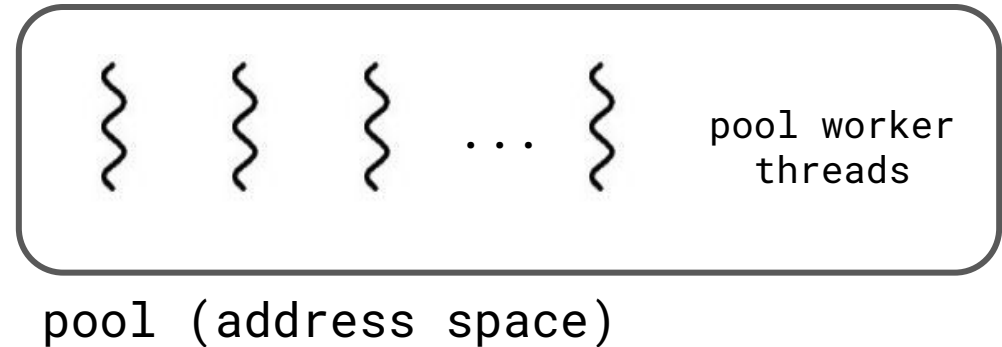


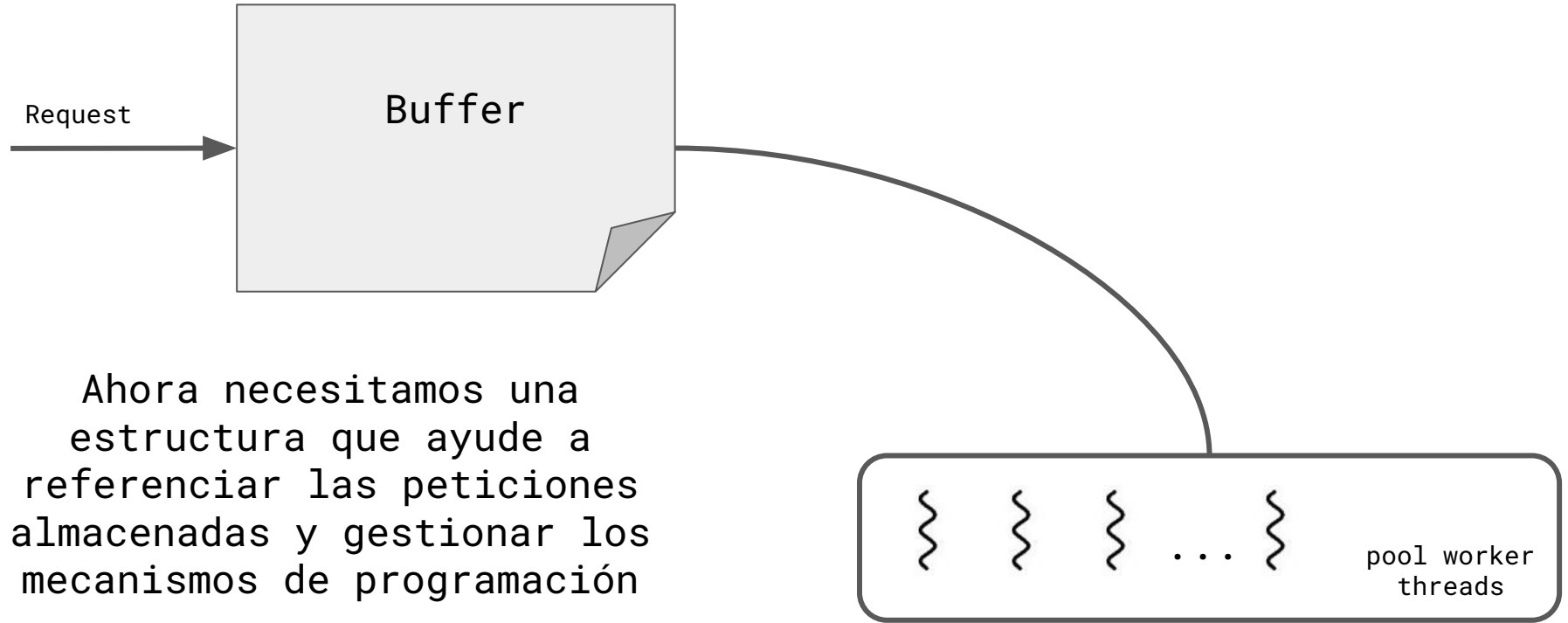
El esquema principal es tomar el cliente como mediador para que el server gestiona peticiones

# Flujo de trabajo



Queremos manejar un espacio de memoria para manipular los hilos que procesan la respuesta para el cliente





# Implementación

La implementación consiste en definir un fichero adicional que contenga los archivos `thread_pool.c` y su interfaz `thread_pool.h`. Es importante definir cómo se gestionará el búfer para el manejo de las peticiones dentro de los hilos.

Se realizó una modificación en `wserver.c` para permitir la gestión de los hilos y el manejo del búfer desde allí, así como la incorporación de funciones que permiten escalar las peticiones de forma periódica, tomada de la implementación mostrada en el archivo `spin.c`.

*// Estructura del buffer*

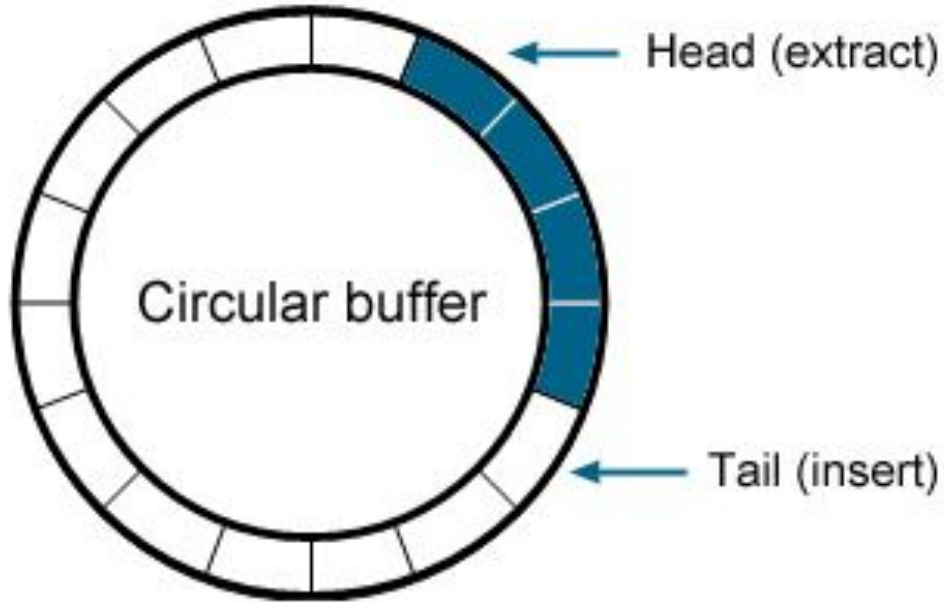
```
typedef struct buffer
{
    int *elementos;
    int indice;
    int back_indice;
    int count;
    int capacidad;

    pthread_mutex_t lock;
    pthread_cond_t produce;
    pthread_cond_t consume;
} buffer_t;
```



Esta estructura es la base para implementar un **buffer circular concurrente** donde múltiples hilos pueden producir y consumir datos, define una estructura (**struct**) de buffer circular o cola para la sincronización entre hilos (threads), y aplicar productor-consumidor.





```
int indice;
```

Índice donde el **productor** va a insertar el siguiente elemento

```
int back_indice;
```

Índice donde el **consumidor** va a extraer el siguiente elemento.

# Campos de sincronización

`pthread_mutex_t lock;`

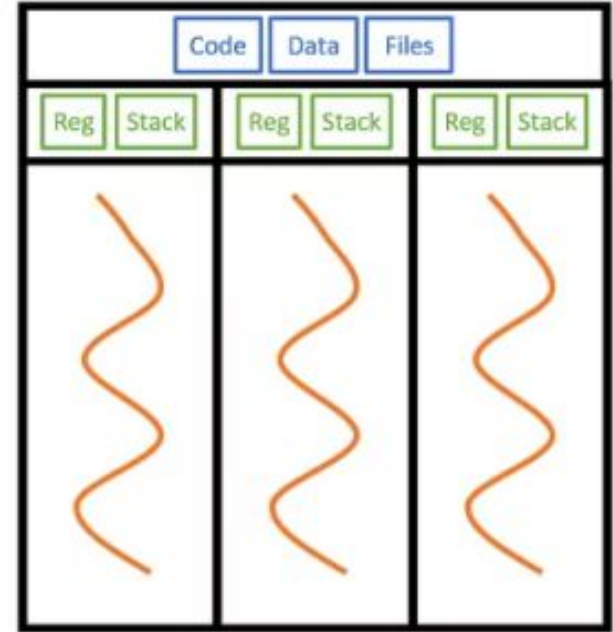
Mutex (candado) usado para proteger el acceso concurrente al buffer. Asegura que solo un hilo a la vez pueda modificar el buffer.

`pthread_cond_t produce;`

Variable de condición para el **productor**. Si el buffer está lleno, el productor espera en esta condición.

`pthread_cond_t consume;`

Variable de condición para el **consumidor**. Si el buffer está vacío, el consumidor espera en esta condición.



# Operaciones del buffer

```
void inicializar(buffer_t *buffer, int capacidad);
```

```
void destruir(buffer_t *buffer);
```

```
void insertar(buffer_t *buffer, int item);
```

```
int borrar(buffer_t *buffer);
```



Prepara el buffer para que pueda ser utilizado por los hilos productores y consumidores, crea el espacio en memoria para almacenar los datos, inicializa las variables y configura los mecanismos para trabajar **threads sin interferirse**.

# Operaciones del buffer

`void inicializar(buffer_t *buffer, int capacidad);`

`void destruir(buffer_t *buffer);` 

`void insertar(buffer_t *buffer, int item);`

`int borrar(buffer_t *buffer);`

En este caso se liberan los recursos que fueron reservados en el método anterior, borrando la memoria, quitando los Mutex, y limpiando los datos

# Operaciones del buffer

```
void inicializar(buffer_t *buffer, int capacidad);
```

```
void destruir(buffer_t *buffer);
```

```
void insertar(buffer_t *buffer, int item);
```

```
int borrar(buffer_t *buffer);
```

Agregar un nuevo dato (request) al buffer, si hay espacio disponible, un thread productor debe guardar en el buffer. Se valida si esta lleno con una función auxiliar, para poner en espera si el buffer está lleno.

```
static inline bool is_buffer_full(const buffer_t *buffer)
{
    return buffer->count == buffer->capacidad;
}
```

# Operaciones del buffer

```
void inicializar(buffer_t *buffer, int capacidad);
```

```
void destruir(buffer_t *buffer);
```

```
void insertar(buffer_t *buffer, int item);
```

```
int borrar(buffer_t *buffer);
```

Extrae un dato(request) del buffer, si hay al menos uno disponible, un **thread consumidor** desea tomar una request del buffer.

Se valida si esta vacío con una función auxiliar, para poner en espera hasta que resulte una request.

```
static inline bool is_buffer_empty(const buffer_t *buffer)
{
    return buffer->count == 0;
}
```

# Modificaciones en wserver.c

```
pthread_t *thread_pool = malloc(sizeof(pthread_t) * num_threads);
```

```
...
```

```
pthread_create(&thread_pool[i], NULL, routine_thread, NULL);
```

En lugar de que el servidor atienda una solicitud a la vez (como en el primer código), ahora puede manejar **múltiples solicitudes simultáneamente** mediante un **grupo de hilos**. Cada hilo en el *thread pool* atiende conexiones en paralelo.

# Modificaciones en wserver.c

```
static buffer_t request_buffer;  
inicializar(&request_buffer, cap_buffer);  
  
...  
insertar(&request_buffer, conn_fd);  
  
...  
int conn_fd = borrar(&request_buffer);
```

Añadir la estructura **buffer\_t** para almacenar conexiones entrantes. Donde los hilos consumidores de la rutina (**routine\_thread()**) extraen las conexiones del buffer y las procesan, mientras que el hilo principal acepta las conexiones y las agrega al buffer.



# Modificaciones en wserver.c

Agregamos la función `get_seconds()`, la cual tiene como finalidad **obtener el tiempo actual** (en microsegundos y segundos) al momento de ejecutarse, esto nos servirá para más adelante poder realizar pruebas **más claras** de la concurrencia.

```
double get_seconds()
{
    struct timeval t;
    int rc = gettimeofday(&t, NULL);
    assert(rc == 0);
    return (double)((double)t.tv_sec +
(double)t.tv_usec / 1e6);
}
```

# Modificaciones en wserver.c

Cada hilo ejecuta esta función en bucle. Espera una conexión desde el buffer, la maneja y la cierra. Para poder trabajar varios hilos de manera paralela, aumentando el rendimiento del servidor.

La espera artificial de 3 segundos (`while ((get_seconds() - time1) < 3)`) simula carga o retardo, para los requests.

```
void *routine_thread(void *arg)
{
    while (1)
    {
        int conn_fd = borrar(&request_buffer);
        double time1 = get_seconds();
        while ((get_seconds() - time1) < 3)
        {
            sleep(1);
        }
        request_handle(conn_fd);
        close_or_die(conn_fd);
    }
    return NULL;
}
```

# Modificaciones en wserver.c

```
while ((c = getopt(argc, argv, "d:p:t:b:")) != -1)
```

```
switch (c){
```

```
.  
.   
.
```

```
case 't':
```

```
    num_threads = atoi(optarg);
```

```
    break;
```

```
case 'b':
```

```
    cap_buffer = atoi(optarg);
```

```
    break;
```

```
default:
```

```
    fprintf(stderr, "usage: wserver [-d basedir] [-p port] [-t # threads] [-b buffers-capacity]\n");
```

```
    exit(1);
```

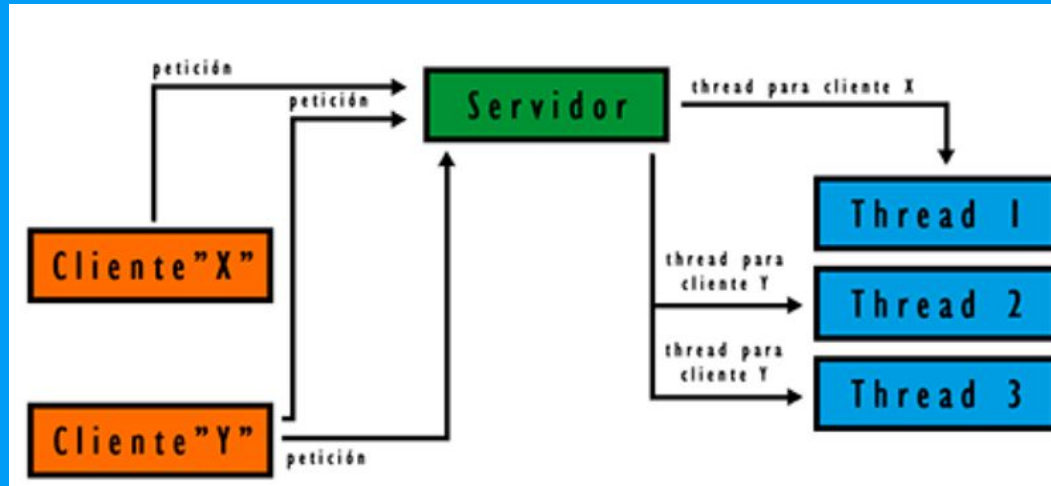
```
}
```

Añadimos los argumentos requeridos tales como el **número de hilos** (-t) y la **capacidad del buffer** (-b), esto para optimizar la funcionalidad multi-hilo de nuestro servidor web.

-t → número de hilos del pool.

-b → capacidad del buffer.

# Funcionamiento



# Funcionamiento

Aca podemos **ver** el servidor web **corriendo**, lo ejecutamos con los nuevos argumentos de **hilos y buffer**, en 2 cada valor.

```
42 |     sprintf(content, "%s<p>I spun for %.2f seconds</p>\r\n", content, t2 - t1);
    |     ^~~~~~
spin.c:42:5: warning: 'sprintf' argument 3 overlaps destination object 'content' [-Wrestrict]
spin.c:39:10: note: destination object referenced by 'restrict'-qualified argument 1 was declared here
39 |     char content[MAXBUF];
    |     ^~~~~~

(bryan@AMG) - [~/Documents/Lab3/concurrency_web_server/concurrency_web_server]
$ ./wserver -p 8080 -t 2 -b 2
```

# Funcionamiento

Acá vamos a **ejecutar** el **servidor cliente**, realizando **varias (7)** **peticiones** a nuestro servidor web, en este caso usaremos el archivo `index.html` en cada petición.

```
index.html:Zone.Identifier  LICENSE          pool_request.o  request.o  wclient.c  wserver.o
io_helper.c                Makefile        README.md      spin.c     wclient.o
io_helper.h                pool_request.c  request.c      spin.cgi   wserver

(bran@AMG) - [~/Documents/Lab3/concurrency_web_server/concurrency_web_server]
$ ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient
localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 808
0 /index.html & ./w^Client localhost 8080 /index.html & ./wclient localhost 8080 /index.html
```

# Funcionamiento

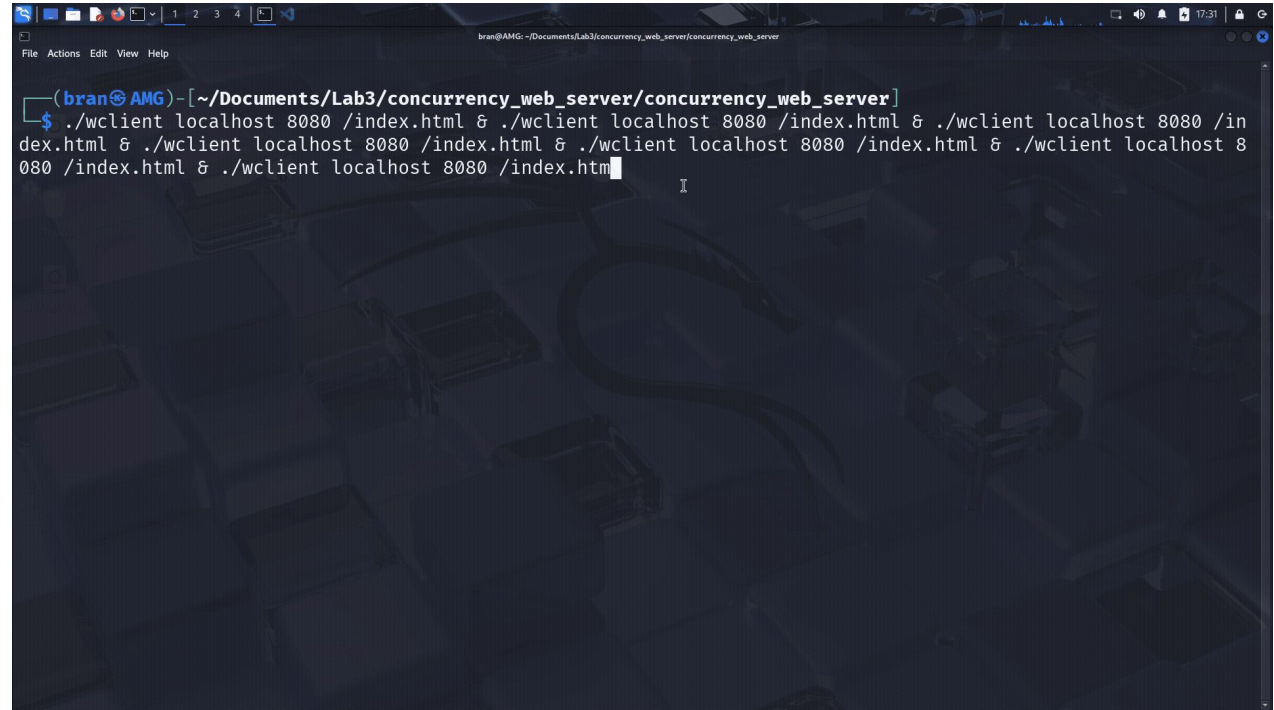
Acá podemos **observar** todas las **peticiones recibidas** en nuestro servidor web, de igual manera vemos que sigue a la **espera** de nuevas peticiones.

```
(bran@AMG) - [~/Documents/Lab3/concurrency_web_server/concurrency_web_server]
$ ./wserver -p 8080 -t 2 -b 2
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1
method:GET uri:/index.html version:HTTP/1.1

```

# Funcionamiento

Acá se puede observar cómo responde el server a la petición que realiza el cliente, y mostrando la escala de tiempo cada 2 peticiones

A terminal window with a dark background and a cityscape pattern. The prompt is 'bran@AMG: ~/Documents/Lab3/concurrency\_web\_server/concurrency\_web\_server'. The command entered is: `$.wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html`. The cursor is at the end of the command.

```
bran@AMG: ~/Documents/Lab3/concurrency_web_server/concurrency_web_server
$.wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html & ./wclient localhost 8080 /index.html
```



# Anexos

