Apellidos: *Nombre:*

Concurrencia y Paralelismo

Grado en Ingeniería Informática

Junio 2021

1. *Gestión de Peticiones [2.5 puntos]*

   En un servidor web se gestionan las peticiones usando un modelo de productores/consumidores donde un dispatcher recibe las peticiones y las inserta en una cola, y varios threads worker las procesan.

```
struct request_queue {
  queue requests;
  pthread_mutex_t queue_lock;
  pthread_cond_t queue_full, queue_empty;
};

// Funciones ya implementadas en otros módulos del sistema
int elements(queue *q);
request *remove(queue *q);
void insert(queue *q, request *r);
int buffer_size(queue *q);
void register_worker(pthread_t id);

void *dispatcher(void *ptr) {
  struct request_queue *req = ptr;
  while(1) {
    request r = accept_connection();

    pthread_mutex_lock(&req->queue_lock);
    while(elements(&req->requests)==buffer_size(&req->requests)) // Esperar por sitio
      pthread_cond_wait(&req->queue_full, &req->queue_lock);

    insert(&req->requests, r);

    if(elements(&req->requests)==1) pthread_cond_broadcast(&req->queue_empty);
    pthread_mutex_unlock(&req->buffer_lock);
  }
}

void *worker(void *ptr) {
  struct request_queue *req = ptr;
  while(1) {
    request r;

    pthread_mutex_lock(&req->queue_lock);
    while(elements(&req->requests)==0)
      pthread_cond_wait(&req->queue_empty, &req->queue_lock);

    r = remove(&req->requests);

    if(elements(&req->requests)==buffer_size(&req->requests)-1)
      pthread_cond_broadcast(&req->queue_full);
    pthread_mutex_unlock(&req->queue_lock);

    serve_request(r);
  }
}
```

   Para garantizar que el sistema es capaz se adapta a la carga de trabajo en cada momento, modifique la implementación de tal forma que:

   a) Cuando un worker se encuentre la cola de peticiones llena debe iniciar un nuevo thread worker para aumentar la capacidad del sistema. Para registrar el nuevo worker en el sistema llame a la función `register_worker` pasando el id del nuevo thread.

   b) Para no tener demasiados procesos cuando no haya peticiones, solo debería haber un worker esperando. Si un worker ve la cola de peticiones vacías cuando ya hay otro worker esperando debería terminar y no esperar.

Solución

```
struct request_queue {
  queue requests;
  pthread_mutex_t queue_lock;
  pthread_cond_t queue_full, queue_empty;
  int waiting_workers;
};

int elements(queue *q);
request *remove(queue *q);
void insert(queue *q, request *r);
int buffer_size(queue *q);
void register_worker(pthread_t id);

void *dispatcher(void *ptr) {
  struct request_queue *req = ptr;
  while(1) {
    request r = accept_connection();

    pthread_mutex_lock(&req->queue_lock);
    while(elements(&req->request)==buffer_size(&req->requests))  // Esperar por sitio
      pthread_cond_wait(&req->queue_full, &req->queue_lock);

    insert(&req->requests, r);

    if(req->waiting_workers) {
        req->waiting_workers--;
        pthread_cond_signal(&req->queue_empty);
    }
    pthread_mutex_unlock(req->buffer_lock);
  }
}

void *worker(void *ptr) {
  struct request_queue *req = ptr;
  while(1) {
    request r;

    pthread_mutex_lock(&req->queue_lock);
    while(elements(&req->requests)==0) {
      if(req->waiting_workers > 0) {
        pthread_mutex_unlock(&req->queue_lock);
        return NULL;
      }
      req->waiting_workers++;
      pthread_cond_wait(&req->queue_empty, &req->queue_lock);
    }

    r = remove(&req->requests);

    if(elements(&req->requests)==buffer_size(&req->requests)-1) {
      pthread_t new_worker_id;
      pthread_cond_broadcast(&req->queue_full);
      pthread_create(&new_worker_id, NULL, worker, req);
      register_worker(new_worker_id);
    }
    pthread_mutex_unlock(&req->queue_lock);

    serve_request(r);
  }
}
```

2. *Barreras [2.5 puntos]*

a) **Thread principal espera por todos**

Dado un `thread` principal y varios threads auxiliares se quiere que el `thread` principal espere a que todos los otros threads lleguen a un determinado punto. El `thread` principal crea una nueva barrera con `barrier_create()`. El número de threads auxiliares que se van a sincronizar se pasa como argumento.

Cuando el thread principal quiere esperar porque todos los threads lleguen a un punto determinado llamará a la función `barrier_server()`. El resto de threads, al llegar al punto de sincronización llaman a la función `barrier_client()`.

Notese que los threads clientes no tienen que esperar a que todos los threads clientes lleguen al punto, simplemente indican al thread principal que ya han llegado.

```
struct barrier {
        ....
};

struct barrier *barrier_create(int num_threads) {
        ....
}

int barrier_server(struct barrier *barrier) {
        ....
}

int barrier_client(struct barrier *barrier) {
        ....
}
```

b) **Todos los threads esperan por todos**

Modifique el código del apartado anterior para que todos los threads esperen por todos. Es decir, cuando un thread llega a la barrera, indica al thread principal que ya ha llegado, y espera a que el thread principal le diga que ya han llegado todos. Tenemos que asegurarnos que ningún thread sale de `barrier_client()` hasta que no hayan llegado todos a esa función.

c) **Función thread_destroy()**

Necesitamos liberar la `struct barrier` que creamos con `barrier_create()`. Eso lo haremos con la función `barrier_destroy()` que la llamará el thread principal después de llamar a `barrier_server()`. Nótese que tiene que asegurarse que todos los threads clientes ya no estén usado la estructura.

```
int barrier_destroy(struct barrier *barrier) {
        ....
}
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*fun) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Solución

a) Apartado 1

```
struct barrier {
        int num_threads;
        int counter;
        pthread_mutex_t m;
        pthread_cond_t server;
};

struct barrier *barrier_create(int num_threads)
{
struct barrier *b = malloc(sizeof(struct barrier));

        if (!b)
         return NULL;
        b->num_threads = num_threads;
        b->counter = 0;

        if (pthread_mutex_init(&b->m, NULL)) {
         free(b);
                return NULL;
        }
        if (pthread_cond_init(&b->server, NULL)) {
                pthread_mutex_destroy(&b->m);
         free(b);
                return NULL;
        }
        return b;
}

int barrier_server(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
while (barrier->counter < barrier->num_threads) {
printf("barrier counter %d\n", barrier->counter);
         pthread_cond_wait(&barrier->server, &barrier->m);
         }
pthread_mutex_unlock(&barrier->m);
        return 0;
}

int barrier_client(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
        barrier->counter++;
pthread_mutex_unlock(&barrier->m);
        pthread_cond_signal(&barrier->server);
        return 0;
}
```

b) Apartado 2

```
struct barrier {
        int num_threads;
        int counter;
        pthread_mutex_t m;
        pthread_cond_t server;
        pthread_cond_t clients;
};

struct barrier *barrier_create(int num_threads)
{
struct barrier *b = malloc(sizeof(struct barrier));

        if (!b)
         return NULL;
        b->num_threads = num_threads;
        b->counter = 0;

        if (pthread_mutex_init(&b->m, NULL)) {
         free(b);
                return NULL;
```

```
        }
        if (pthread_cond_init(&b->server, NULL)) {
                pthread_mutex_destroy(&b->m);
         free(b);
                return NULL;
        }
        if (pthread_cond_init(&b->clients, NULL)) {
           pthread_cond_destroy(&b->server);
           pthread_mutex_destroy(&b->m);
         free(b);
                return NULL;
        }
        return b;
}

int barrier_server(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
while (barrier->counter < barrier->num_threads) {
printf("barrier counter %d\n", barrier->counter);
         pthread_cond_wait(&barrier->server, &barrier->m);
        }
        pthread_cond_broadcast(&barrier->clients);
pthread_mutex_unlock(&barrier->m);
        return 0;
}

int barrier_client(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
        barrier->counter++;
        pthread_cond_signal(&barrier->server);
        pthread_cond_wait(&barrier->clients, &barrier->m);
pthread_mutex_unlock(&barrier->m);
        return 0;
}
```

---

c) Apartado 3

```
struct barrier {
        int num_threads;
        int counter;
        pthread_mutex_t m;
        pthread_condition_t server;
        pthread_condition_t clients;
        pthread_condition_t destroy;
};

struct barrier *barrier_create(int num_threads)
{
struct barrier *b = malloc(sizeof(struct barrier));

        if (!b)
         return NULL;
        b->num_threads = num_threads;
        b->counter = 0;

        if (pthread_mutex_init(&b->m)) {
         free(b);
                return NULL;
        }
        if (pthread_cond_init(&b->server)) {
                pthread_mutex_destroy(&b->m);
         free(b);
                return NULL;
        }
        if (pthread_cond_init(&b->clients)) {
           pthread_cond_destroy(&b->server);
           pthread_mutex_destroy(&b->m);
         free(b);
                return NULL;
        }
        if (pthread_cond_init(&b->destroy)) {
           pthread_cond_destroy(&b->clients);
           pthread_cond_destroy(&b->server);
           pthread_mutex_destroy(&b->m);
```

```
         free(b);
                return NULL;
        }
        return b;
}

int barrier_server(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
while (barrier->counter < barrier->num_threads) {
         pthread_cond_wait(&barrier->server, &barrier->m);
        }
        pthread_cond_broadcast(&barrier->clients);
pthread_mutex_unlock(&barrier->m);
        return 0;
}

int barrier_client(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
        barrier->counter++;
        pthread_cond_signal(&barrier->server);
        pthread_cond_wait(&barrier->client, &barrier->m);
        barrier->counter--;
        pthread_cond_signal(&barrier->destroy);
pthread_mutex_unlock(&barrier->m);
        return 0;
}

int barrier_destroy(struct barrier *barrier)
{
pthread_mutex_lock(&barrier->m);
while (barrier->counter > 0) {
         pthread_cond_wait(&barrier->destroy, &barrier->m);
        }
pthread_mutex_unlock(&barrier->m);
        pthread_cond_destroy&barrier->server);
        pthread_mutex_destroy&barrier->clients);
        pthread_mutex_destroy&barrier->destroy);
        free(barrier);
        return 0;
}
```
