Paralelización con Threads

Basadas en las clases del relator

Rodrigo Arenas A., Luis Mateu B. & Lucas Torrealba A.

14 de marzo de 2024

Contenidos

1. Thread (procesos livianos)
2. Computadores multi-core
3. Programas con múltiples threads
4. Funciones para manipular threads
5. Ejemplos
6. Paralelización

Procesos en paralelo

- Procesos Pesados (visto en PSS)
  - No comparten memoria
  - Sirven para transferir data
  - Costosos de instanciar
  - Costosos para comunicarse entre ellos
- Procesos Livianos:
  - Llamados **threads**, traducido al español como *hilos de ejecución* o *hebras*
  - Pueden compartir memoria
  - Son baratos de instanciar

Computadores multi-core

- ¿Qué es un *core*? Es un núcleo de ejecución. (Componente de *hardware* capaz de ejecutar un thread)
- Hasta el año ∼2005 los computadores personales eran solo *mono-core*. Después se comenzaron a utilizar 2 *cores*, 4 *cores*, 8 *cores*...
- Un computador *quad-core* puede ejecutar hasta 4 *threads* en paralelo. Idealmente se puede hacer el mismo trabajo en la cuarta parte del tiempo.
- La cantidad de *cores* no limita la cantidad de *threads* que podemos crear. En un programa podemos crear muchos mas threads que los *cores* disponibles.
- El sistema operativo (S.O.) atribuye los *cores* a los *threads* activos otorgándoles tajadas de tiempo de ejecución. Estas tajadas son lo suficientemente pequeñas para que parezca que los threads se están ejecutando en paralelo, pero **no es así**.

Programas con múltiples threads

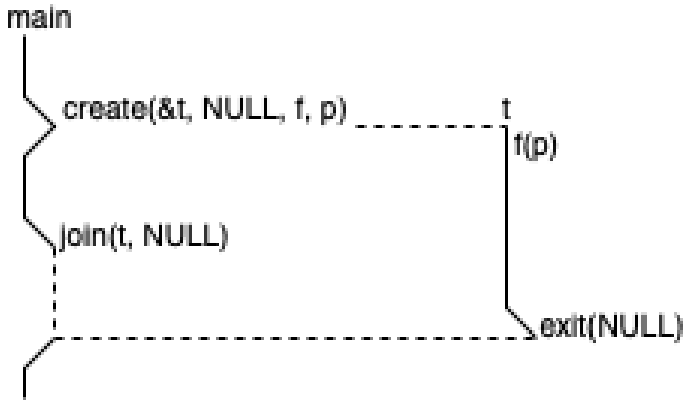
- En estos programas tenemos un *thread* principal que ejecuta la función *main*. Este *thread* puede crear un(os) nuevo thread(s) que se ejecuta en paralelo con el programa principal.
- Ventajas:
  - Los *threads* se ejecutan en paralelo.
- Desventajas:
  - Programación es mas compleja.
  - Nuevos errores, ocurren aleatoriamente y son muy difíciles de depurar.
- Antes de recurrir a los threads múltiples es más rentable investigar si hay algoritmos más eficientes, por ejemplo, usar un algoritmo $O(n \log n)$ en vez de $O(n^2)$. Si se agotan las alternativas, se recurre a los *threads*.

Funciones para manipular threads

Se utilizarán los *pthreads* que es un estándar POSIX. Para utilizar las siguientes funciones debe agregar `#include<pthread.h>`

- Creación: `int pthread_create(pthread_t *thread, const pthread_attr *attr, void *(*start_routine)(void *), void *arg)`
- Término:
  - Cuando `start_routine` retorna
  - Cuando desde cualquier función que se esté ejecutando en este thread, invoque a: `void phtread_exit(void *ret)`
- Esperar a que un *thread* termine: `int pthread_join(pthread_t t, void **pret)`
- Todo *thread* debe ser enterrado una y solo una vez, de otro modo el *thread* se convierte en un **zombie**.

Ejemplo:

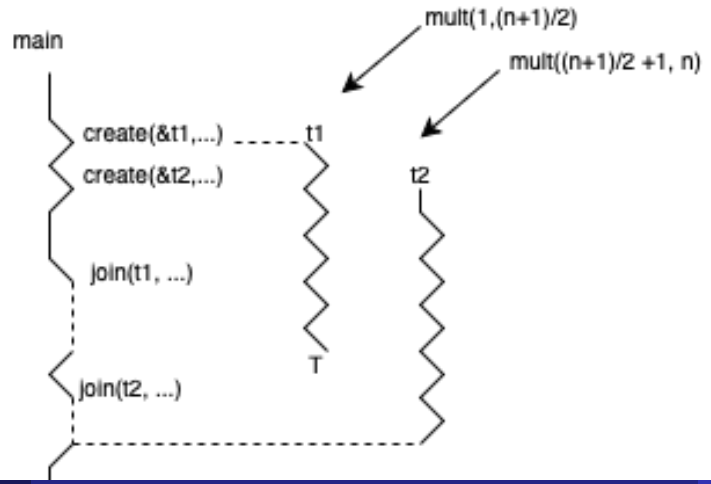


Ejemplo: Factorial

Se calculará el factorial en una computador dual core. Primero, se observará el cálculo de factorial versión secuencial

```
1  double mult(int i, int j){
2      double p = 1.0;
3      for (int k = i; k <= j; k++)
4          p *= k;
5      return p;
6  }
7
8  double factorial(int n){
9      return mult(1, n)
10 }
```

Para la versión paralela queremos utilizar 2 threads para el cálculo del factorial, como se muestra en el siguiente diagrama:



Ejemplo: Factorial dual core

¿Podemos utilizar la función `mult` ya creada de la siguiente forma: `create(&t1, NULL, mult, ...)`?

- **No**, el problema es que `mult` no tiene el encabezado que necesita la función `create`. El compilador arrojaría error si colocamos `mult` como parámetro en esa posición.
- Para solucionar este problema, crearemos la función `void *mult_thread(void *ptr)` que si tiene el encabezado que pide `create`, es decir, recibe un puntero opaco y retorna un puntero opaco.
- El parámetro `ptr` es una dirección de una estructura que contiene los valores `i`, `j`. Es por eso que definimos la estructura:

```
1 typedef struct{
2     int i, j;
3     double res;
4 } Args
```

Note que añadimos un campo `res` que será utilizado para guardar el resultado.

Ejemplo: Factorial dual core

```c
void *mult_thread(void *ptr){
    // note que no podemos hacer ptr->i
    Args *a = ptr
    a -> res = mult(a->i, a->j);
    return NULL;
}
```

```c
typedef struct{
    int i, j;
    double res;
} Args
```

Ahora programaremos la función main que es la encargada de crear y enterrar los 2 *threads* para calcular el factorial.

```c
int main(int argc, char **argv){
    int n=atoi(argv[1]);
    int h = (n+1)/2;
    Args a1 = {1, h, 0.};
    Args a2 = {h+1, n, 0.};
    pthread_t t1, t2;
    create(&t1, NULL, mult_thread, &a1);
    create(&t2, NULL, mult_thread, &a2);
    join(t1, NULL);
    join(t2, NULL);
    printf(" %g\n", a1.res*a2.res);
    return 0;
}
```

Errores frecuentes: No llamar a join

```c
void *mult_thread(void *ptr){
    // note que no podemos hacer ptr->i
    Args *a = ptr
    a -> res = mult(a->i, a->j);
    return NULL;
}
```

```c
typedef struct{
    int i, j;
    double res;
} Args
```

```c
int main(int argc, char **argv){
    int n=atoi(argv[1]);
    int h = (n+1)/2;
    Args a1 = {1, h, 0.};
    Args a2 = {h+1, n, 0.};
    pthread_t t1, t2;
    create(&t1, NULL, mult_thread, &a1);
    create(&t2, NULL, mult_thread, &a2);
    //join(t1, NULL);
    //join(t2, NULL);
    printf(" %g\n", a1.res*a2.res);
    return 0;
}
```

¿El resultado sigue correcto?

Errores frecuentes: threads que no trabajan en paralelo

```c
void *mult_thread(void *ptr){
    // note que no podemos hacer ptr->i
    Args *a = ptr
    a -> res = mult(a->i, a->j);
    return NULL;
}
```

```c
typedef struct{
    int i, j;
    double res;
} Args
```

```c
int main(int argc, char **argv){
    int n=atoi(argv[1]);
    int h = (n+1)/2;
    Args a1 = {1, h, 0.};
    Args a2 = {h+1, n, 0.};
    pthread_t t1, t2;
    create(&t1, NULL, mult_thread, &a1);
    join(t1, NULL); // enterramos el thread 1 antes de llamar al thread 2
    create(&t2, NULL, mult_thread, &a2);
    join(t2, NULL);
    printf(" %g\n", a1.res*a2.res);
    return 0;
}
```

¿El resultado sigue correcto?

Errores frecuentes: threads que realizan muy poco trabajo

```
1 void *mult_thread(void *ptr){
2     // note que no podemos hacer ptr->i
3     Args *a = ptr
4     a -> res = mult(a->i, a->j);
5     return NULL;
6 }
```

```
1 typedef struct{
2     int i, j;
3     double res;
4 } Args
```

```
1 int main(int argc, char **argv){
2     int n=atoi(argv[1]);
3     int h = (n+1)/2;
4     Args a1 = {1, h, 0.};
5     Args a2 = {h+1, n, 0.};
6     pthread_t t1, t2;
7     create(&t1, NULL, mult_thread, &a1);
8     create(&t2, NULL, mult_thread, &a2);
9     join(t1, NULL);
10    join(t2, NULL);
11    printf(" %g\n", a1.res*a2.res);
12    return 0;
13 }
```

¿Cuan eficiente es esta solución? ¿Es más eficiente que la versión secuencial?

Ejercicios:

1. Modifique el programa de modo que se lance un solo nuevo *thread*. Use el *thread* principal para hacer la mitad del cálculo.

2. Modifique el programa anterior, de manera que se reciba como segundo parámetro el número de threads que se deben usar para calcular el factorial. Ejemplo de uso:
   `$ ./fact 300 8`

3. Revise el funcionamiento para n pequeños.

Ejemplo: Multiplicación de Matrices

```
1  typedef double **Matriz;
2
3  Matriz nuevaMatriz(int n, int m){
4      double **filas = malloc(n*sizeof(double *));
5      for (int i=0; i<n; i++)
6          filas[i] = malloc(m*sizeof(double));
7      return filas;
8
9
10 //podemos crear matrices
11 Matriz mat = nuevaMatriz(5,6);
12 // y modificar los valores
13 mat[1][2] = 5.0
```

Además recordemos que la multiplicación de matrices viene dada por la fórmula:

$$C = A \times B$$

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

Ejemplo: Multiplicación de Matrices - Secuencial

Asumiremos que las matrices son cuadradas, es decir, son de tamaño $n \times n$:

```
1  // multiplica la matriz a con b y deja el resultado en la matriz c
2  void mult(Matriz a, Matriz b, Matriz c, int n){
3      for(int i=0; i<n; i++{
4          for(int j=0; j<n; j++){
5              double sum = 0.0;
6              for (int k=0; k<n; k++){
7                  sum += a[i][k]*b[k][j];
8              c[i][j] = sum
9          }
10     }
11 }
```

Ejemplo: Multiplicación de Matrices - Paralela

¿Cómo calcular en paralelo usando *p threads*?

- En nuestra matriz resultante de la multiplicación tendremos n filas y n columnas. La idea es que los p *threads* se dividan el trabajo de calcular las n filas.
  - Si p es divisor de n, entonces a cada *thread* le corresponde $\frac{n}{p}$ filas que calcular.
  - En caso contrario, asumiremos que tiene resto $k > 0$, luego $n \mod p \equiv k$. En este caso, tendremos que k *threads* calcularán $\lfloor \frac{n}{p} \rfloor + 1$ filas, mientras que los $p - k$ *threads* restantes, calcularán $\lfloor \frac{n}{p} \rfloor$ filas.

Ejemplo: Multiplicación de Matrices - Paralela

Reutilizaremos la función mult pero esta vez le agregaremos dos parámetros que denotarán el inicio y el fin de la fila que estamos calculando.

```
1  // multiplica la matriz a con b y deja el resultado en la matriz c
2  void mult(Matriz a, Matriz b, Matriz c, int n, int ini, int fin){
3      for(int i=ini; i<fin; i++{ // solo se modifica esta linea
4          for(int j=0; j<n; j++){
5              double sum = 0.0;
6              for (int k=0; k<n; k++){
7                  sum += a[i][k]*b[k][j];
8              c[i][j] = sum
9          }
10     }
11 }
```

Además, como necesitamos invocar pthread_create, necesitamos la estructura Args para entregarle los parámetros.

```
1  typedef struct{
2      Matriz a,b,c;
3      int n, ini, fin;
4  } Args;
```

Ejemplo: Multiplicación de Matrices - Paralela

```
1 typedef struct{
2     Matriz a,b,c;
3     int n, ini, fin;
4 } Args;
```

Además, necesitamos una función que tenga el encabezado apropiado para pasarle a pthread_create:

```
1 void *mult_thread(void *ptr){
2     Args *arg = ptr;
3     mult (arg -> a, arg-> b, arg->c, arg->n, arg->ini, arg->fin);
4     return NULL;
5 }
```

Ejemplo: Multiplicación de Matrices - Paralela

```
1 void *mult_thread(void *ptr){
2     Args *arg = ptr;
3     mult (arg -> a, arg-> b, arg->c, arg->n,
        ↪ arg->ini, arg->fin);
4     return NULL;
5 }
```

```
1 typedef struct{
2     Matriz a,b,c;
3     int n, ini, fin;
4 } Args;
```

```
1  void par_mult(Matriz a, Matriz b, Matriz c, int n, int p){
2      pthread_t t[p];
3      Args array[p];
4      int i = 0; int k = n % p
5      for(int s=0; s<p; s++){
6          Args *arg = &array[s];
7          arg->a = a; arg->b =b; arg->c = c;
8          arg->n = n; arg->ini = i;
9          i += n/p;
10         if (s<k) i++;
11         arg->fin = i;
12         pthread_create(&t[1], NULL, mult_thread, arg)
13     }
14     for (int s = 0; s < p; s++){
15         pthread_join(t[s], NULL)
16     }
17 }
```

Paralelización

En los problemas recién vistos los *threads* eran completamente independientes, es decir, no compartían información. Pero a veces es necesario que los *threads* compartan información e incluso modifiquen la información compartida.
Para solucionar los posibles problemas que esto puede ocasionar tenemos **secciones críticas**... Próxima clase!