Computación Concurrente

Mecanismos de sincronización

22 octubre 2021

Locks y turnos

Método de Peterson

```
int bandera1, bandera2;
int quien;

/* Proceso 1: */
...
bandera1=1;
quien=2;
if ( bandera2 && (quien==2)) {
esperar();
}

/* Sección crítica */

cuenta = cuenta + 1;
bandera1=0;
```

```
/* Proceso 2: */
...
bandera2=1;
quien=1;
if ( bandera1 && quien==1) {
esperar();
}
/* Sección crítica */
cuenta = cuenta + 1;
bandera2=0;
```

Problemas con candados y turnos

Al desarrollar la solución de Peterson en el control de la actualización de una variable compartida se utilizan:

- Banderas
- Turnos

En este método si un proceso que detecta que otro proceso actualizó el turno entonces lo deja pasar, esto conduce a la **espera activa** (*spinlocks*).

En la espera activa un proceso puede consumir mucho tiempo de procesador solo al esperar que otro proceso cambie una bandera.

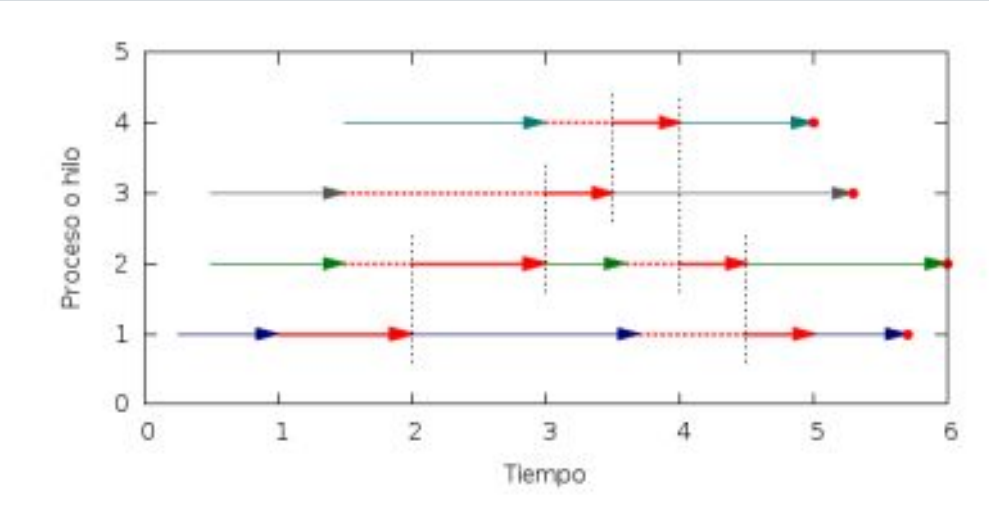
Una forma de mitigar la espera activa es **forzar el cambio de contexto** mediante primitivas del lenguaje.

Mutex

Regiones de exclusión mutua

Uno de los métodos para evitar la espera activa se conoce como **mutex** que propiamente se refiere a la creación de una **región de exclusión mutua** (*mutual exclusion*) por medio de un candado.

En este mecanismo asegura que cierta región del código será ejecutada como si fuera una **operación atómica.**



Regiones de exclusión mutua

El *mutex* es un mecanismo de prevención que mantiene en espera cualquier tarea que quiera entrar a la sección crítica protegida por el mutex.

El *mutex* retiene a la tarea hasta que la tarea que está ejecutando la sección crítica salga de ella.

Si no hay una tarea en la sección crítica uno solo de las tareas que esperan podrá ingresar

Para que un *mutex* sea efectivo tiene que ser implementado mediante una primitiva a bajo nivel implicando al planificador.

Regiones de exclusión mutua

La función **Lock()** de python utiliza un mutex delimitado por el ámbito de su implementación.

Un área de exclusión mutua debe:

1. **Ser mínima:** Debe ser tan corta como sea posible para evitar que otros hilos queden bloqueados fuera del área crítica.
2. **Ser completa:** Se debe analizar bien cuál es el área a proteger y no arriesgarse en proteger de menos.

Un *mutex* es una herramienta sencilla y pieza básica de sincronización de tareas.

Lo fundamental para emplearlos es identificar las regiones críticas del código y proteger el acceso con un mecanismo apto que garantice **atomicidad**.

Speed up y eficiencia

Rendimiento

Es inmediato pensar que si se tiene una algoritmo **a** que es ejecutado por medio de **p** diferentes hilos o procesos, el algoritmo concurrente correrá p veces más rápido que la versión secuencial.

Si asumimos:

- **Ts:** Tiempo que ejecuta el algoritmo secuencial
- **Tc**: Tiempo que ejecuta el algoritmo concurrente

Lo que podríamos esperar es que el **Tc=Ts/p**

Si esto sucede se dice que el programa concurrente tiene **speed up** lineal

Sin embargo esto no es del todo posible debido a múltiples factores:

Rendimiento

Factores que impiden speed up lineal:

- Memoria compartida
- Condiciones de carrera
- Secciones críticas
- Utilización de **mutex**
- Sobrecarga en la ejecución (**overhead**)

Es común que la sobrecarga aumente cuando se incrementa el número de hilos/procesos.

Cada vez más hilos van a competir por acceso a una sección crítica.

Comúnmente, se incrementará el intercambio de datos.

Speed up

Por tanto el speed up podemos calcularlo para un programa concurrente (o paralelo) de la siguiente manera:

$$S = Ts/Tc$$

Un speed up lineal devolvería S=p, lo cual no es usual.

A medida que **p** se incrementa podría esperarse que **S** disminuya cada vez más en cierta proporción respecto al speed up lineal.

Otra forma de expresar lo anterior es considerar que la relación **S/p** será cada vez más pequeña en medida que p aumenta.

La relación **E=S/p** es conocida como eficiencia **E** del programa concurrente.

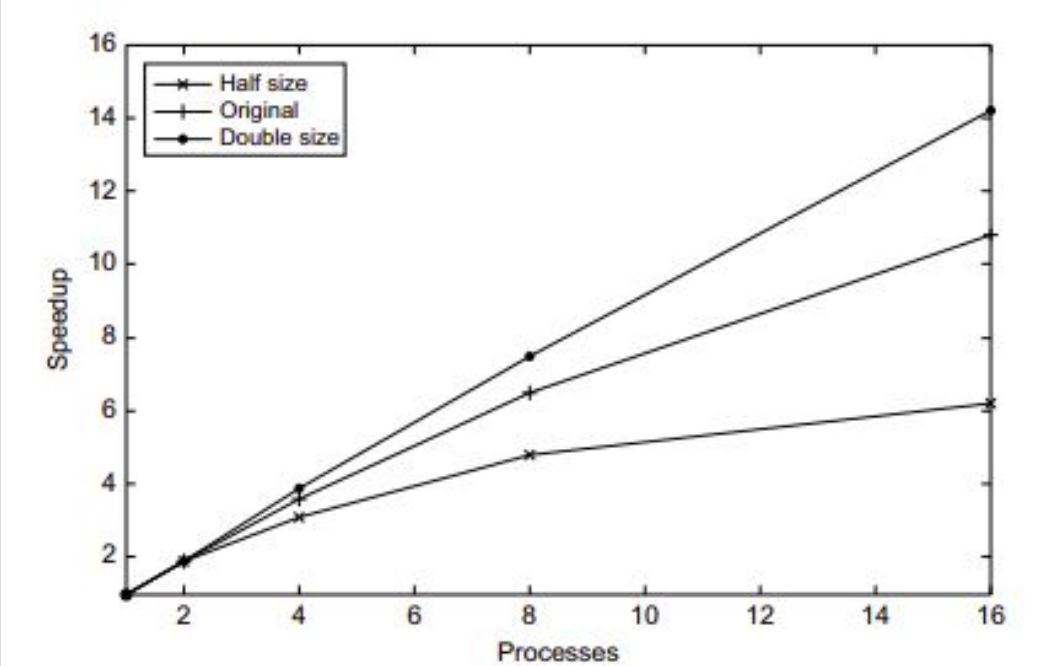
Cálculo Speed up y eficiencia

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

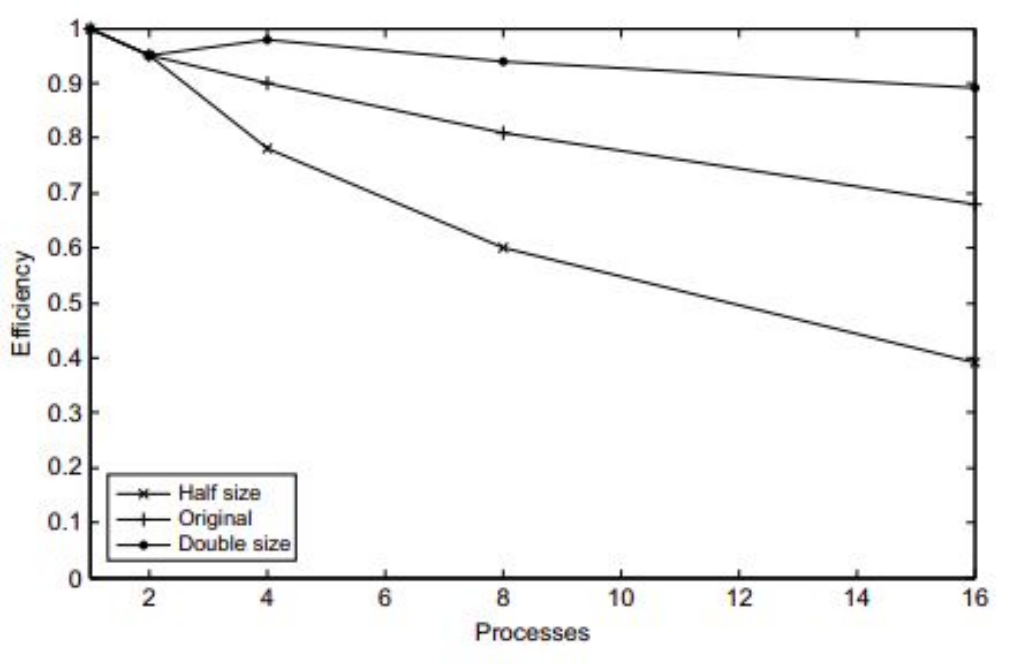
Cálculo de speedup y eficiencia para diferente número de hilos/procesos y tamaño de entrada del programa concurrente

Gráfico de Speed up



Speed up para un programa concurrente con diferente tamaño en la entrada

Gráfico de eficiencia



Eficiencia para un programa concurrente con diferente tamaño en la entrada

Semáforos

oscar.esquivel@iimas.unam.mx