29/01/24

TEMA 1:

Concurrencia ≡ Paralelismo Potencial.

30/01/24

Instrucciones atómicas → Leer / Escribir

Usuario → Preferencia por paralelismo

SO → Preferencia por secuencial.

Ingeniero → tiene que tener en cuenta punto de vista de usuario y SO

Máxima concurrencia

secciones críticas → parte del código donde tenemos que limitar la concurrencia ↑↑↑ IDENTIFICAR (↑↑↑ mayor problema.)

Un proceso solo puede ejecutar 1 programa.

Hilo → proceso ligero. Comparte código y memoria

05/02/2024

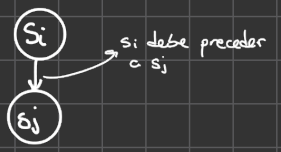
Grafos de precedencia:

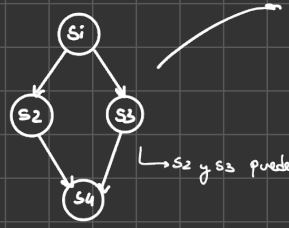
Si → Cjto sentencias secuenciales

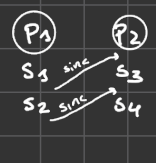
Identificar procesos

Obj => Máx. Concurrencia

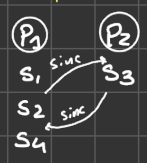
Mínimo nº de procesos concurrentes







Mejor opción ↑



TEMA 2

- Exclusión mutua → Si un proceso está en su sección crítica, el otro no puede ejecutar su sección crítica

Los procesos entran a la sección crítica CUANDO QUIEREN → Pueden no entrar nunca.

- Un proceso no puede decidir por sí mismo que va a entrar en sección crítica.

TEMA 3 -

> Indivisible.

wait y signal se ejecutan como instrucciones atómicas.

S(1, 1)

↳ valor inicial.

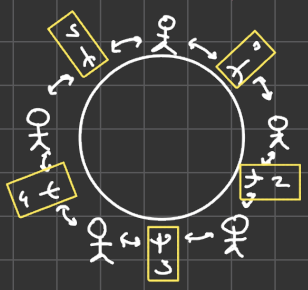
↳ Valor máximo

26/02/2024.

Semáforos de paso: Semáforo binario con variable interna inicializada a 0.
No hacer si no sabemos si hay otro proceso esperando paso.

Problema de los Filósofos:

No se puede arreglar un problema por casuística.



PROBLEMA

1. Cinco procesos Filósofo 1, ... , Filósofo 5.
2. Escribir un programa DIFERENTE para cada proceso.

Prog Filo 1 ... Prog Filo 2

3. Identificar secciones críticas.
4. Herramienta sincronización para limitar la concurrencia <= Código de sincronización

Filósofo 1:
```
Pensar();
Coger_tenedor(1);
Coger_tenedor(2);
comer()
liberar_tenedores();
```

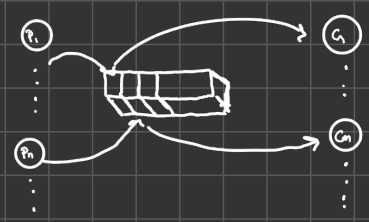
Filósofo 2:
```
Pensar()
coger_tenedor(2)
coger_tenedor(1)
comer()
liberar_tenedores();
```

Filósofo 5:
```
pensar();
wait(sem_comedor)
wait(sem_tenedor[5])
coger_tenedor(5);
wait(sem_tenedor[4])
coger_tenedor(4);
Si los 5 cogen un tenedor, hay interbloqueo. Tenemos que limitar la concurrencia a 4 procesos
Comer()
liberar_tenedores()
Signal(sem_tenedor[5]) signal(sem_tenedor[4])
Signal(sem_tenedor);
```

Semaforo sem_tenedor[5] = {1,1,1,1,1} → 5 secciones críticas. Solucionamos con semaforos binarios inicializados a 1.
Semaforo sem_comedor = {4,4,4,4} → Limitamos a 4 procesos concurrentemente.

```
Filósofo_i
Pensar();
wait(sem_comedor);
wait(sem_tenedor[i]);
wait(sem_tenedor[i-1]);
comer();
Signal(sem_tenedor[i-1]);
Signal(sem_tenedor[i]);
Signal(sem_comedor);
```

Problema del productor - consumidor.



1. P_1 ... P_n | C_1 ... C_n

```
e = producir
intro_elemento(e)
```

```
e.retirar_elem();
consumir(e);
```

REDUCIMOS EL PROBLEMA

- Solo 2 productores

```
e = producir_elem();
if(lleno) wait(sem_lleno)
intro_elemento(e);
Signal(sem_elemento);
```

```
e = producir_elem();
intro_elemento(e);
```

- Solo 2 consumidores → wait(sem_elementos)

```
e.retirar_elem()
consumir(e);
if(prod_esperando) Signal(sem_lleno)
Signal(sem_huecos)
```

```
e.retirar_elem()
consumir(e);
```

```
int prod_esp = 0
int num_huecos = N;
Semaforo sem_buffer(1,1)
sem_lleno(1,0)
sem_hueco(1,0)
sem_aux(1,1)
```

sem_huecos(N,N)
sem_elementos(N,0)

04/03/2024.

Si el buffer es ilimitado y el buffer siempre tiene procesos → semaforo de exclusion mutua

El productor solo puede introducir si el buffer está vacío. Si está lleno depende de cualquier consumidor => sincronización =>
=> semáforo de paso.

```
lleno() {                                   ← wait(sem_aux)
  if (num_huecos == 0) {
    Prod_esperando ++
    wait (Sem_lleno)
    Prod_esperando --
  else
    num_huecos --;
}                                           → signal (sem_aux)
```

↓ sustituir todo

opción 2: wait (sem_huecos)

```
Productor() {
  e = producir();
  wait (sem_huecos);
  wait (sem_buffer);
  introducir (e);
  signal (sem_buffer)
  signal (sem_elementos).
}
```

```
Consumidor() {
  wait (sem_elementos);
  wait (sem_buffer);
  e = retirar_elemento();
  signal (sem_buffer)
  signal (sem_elemento);
  consumir (e);
}
```