Prueba de Oposición

AY2 - Sistemas

Teodoro Freund

20 de Septiembre del 2017

16:00

Contexto

- Materia: Sistemas Operativos

Contexto

- Materia: Sistemas Operativos
- Práctica 3: Sincronización entre procesos

Contexto

- Materia: Sistemas Operativos
- Práctica 3: Sincronización entre procesos
- Primera Parte:
  - Procesos y API del SO
  - Scheduling

Contexto

- Materia: Sistemas Operativos
- Práctica 3: Sincronización entre procesos
- Primera Parte:
  - Procesos y API del SO
  - Scheduling
  - **Sincronización entre procesos**

Contexto

- Materia: Sistemas Operativos
- Práctica 3: Sincronización entre procesos
- Primera Parte:
  - Procesos y API del SO
  - Scheduling
  - **Sincronización entre procesos**
- Segunda Parte:
  - Administración de Memoria
  - Administración de Entrada/Salida
  - Protección y Seguridad
  - Sistemas de Archivos
  - Sistemas Distribuidos
  - Programación Concurrente

El Ejercicio 21: La cena de los antropófagos

Una tribu de antropófagos cena de una gran cacerola que puede contener M porciones de misionero asado. Cuando un antropófago quiere comer se sirve de la cacerola, excepto que esté vacía. Si la cacerola está vacía, el antropófago despierta al cocinero y espera hasta que éste rellene la cacerola.
Pensar que, sin sincronización el antropófago hace:

```
while(true){
   obtener_porcion();
   comer();
}
```

La idea es que el antropófago no pueda comer si la cacerola está vacía y que el cocinero sólo trabaje si está vacía la cacerola.

¿Por qué?

- Es un buen caso de ejemplo de la primitiva menos usada: condición de variable.
- Si bien entra dentro del marco Productor-Consumidor, tiene un giro sobre el mismo.
- Da para discutir distintas implementaciones y sus ventajas y desventajas.
- Por todo esto, es un buen ejercicio para dar en clase.

Pensemos partes de la solución

- El cocinero espera a que la cacerola esté vacía

Pensemos partes de la solución

- El cocinero espera a que la cacerola esté vacía
- El antropófago, a que esté llena

Pensemos partes de la solución

- El cocinero espera a que la cacerola esté vacía
- El antropófago, a que esté llena
- Esas condiciones podrían ser señalizadas de alguna manera en vez de preguntadas por dichos hilos

Pensemos partes de la solución

- El cocinero espera a que la cacerola esté vacía
- El antropófago, a que esté llena
- Esas condiciones podrían ser señalizadas de alguna manera en vez de preguntadas por dichos hilos
- Necesitamos exclusión mútua sobre la 'cacerola'

Primera Versión [1]

```
//Variables globales
int porciones = 0;
mutex = semaforo(1);
vacia = semaforo(0);
lleno = semaforo(0);


void* cocinero(void* s){
  while(true){
    vacia.wait();
    porciones += llenarCacerola();
    llena.signal();
  }
}
```

```
void* antropofago(void* s){
  while(true){
    mutex.wait();
    if (porciones == 0){
      vacia.signal();
      lleno.wait();
    }
    porciones--;
    obtener_porcion();
    mutex.signal();
    comer();
  }
}
```

[1] Sacada de una clase de IIT

Pros y contras de esta solución

Pros:

- Resuelve el problema

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- Mantiene exclusión mútua sobre `obtener_porcion()`

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- ~~Mantiene exclusión mútua sobre `obtener_porcion()`~~

Contras:

- Mantiene exclusión mútua sobre `obtener_porcion()`

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- ~~Mantiene exclusión mútua sobre `obtener_porcion()`~~

Contras:

- Mantiene exclusión mútua sobre `obtener_porcion()`
- Innumerables llamadas al sistema operativo

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- ~~Mantiene exclusión mútua sobre `obtener_porcion()`~~

Contras:

- Mantiene exclusión mútua sobre `obtener_porcion()`
- Innumerables llamadas al sistema operativo
- La abstracción está lejos de ser ideal

Segunda versión

```
//Variables globales
int porciones = 0;
vacia = condicion();
llena = condicion();
condMutex = semaforo(1);

void* cocinero(void* s){
  while(true){
    condMutex.lock();
    while(porciones > 0){
      vacia.wait(condMutex);
    }
    porciones += llenarCacerola();
    llena.broadcast();
    condMutex.unlock();
  }
}
```

```
void* antropofago(void* s){
  while(true){
    condMutex.lock();
    while(porciones <= 0){
      vacia.broadcast();
      llena.wait(condMutex);
    }
    int porcion = porciones--;
    condMutex.unlock();
    obtener_porcion();
    comer();
  }
}
```

Pros y contras de esta solución

Pros:

- Resuelve el problema

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- No mantiene exclusión mútua sobre `obtener_porcion()`

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- No mantiene exclusión mútua sobre `obtener_porcion()`
- Las primitivas parecen abstraer mejor la situación

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- No mantiene exclusión mútua sobre `obtener_porcion()`
- Las primitivas parecen abstraer mejor la situación

Contras:

- Innumerables llamadas al sistema operativo

Pros y contras de esta solución

Pros:

- Resuelve el problema
- No hay busy-wait
- No mantiene exclusión mútua sobre `obtener_porcion()`
- Las primitivas parecen abstraer mejor la situación

Contras:

- Innumerables llamadas al sistema operativo ¿Esto es algo malo?

Tercera versión, spinning

```
//Variables globales
int porciones = 0;
sLock = spinLock();

void* cocinero(void* s){
  while(true){
    while(porciones > 0){}
    sLock.lock();
    porciones += llenarCacerola();
    sLock.unlock();
  }
}
```

```
void* antropofago(void* s){
  while(true){
    while (porciones == 0){}
    sLock.lock();
    if (porciones >= 0){
      porciones--;
      sLock.unlock();
      obtener_porcion();
      comer();
    }else{
      sLock.unlock();
    }
  }
}
```