Departamento de Arquitectura

Sistemas Operativos
Práctico 4

Curso 2024

Ejercicio 1 Se desea controlar el proceso de llenado y tapado de envases de la Compañía Algarrobo Cola. Existe una sola cinta transportadora y una máquina para cada acción. Se dispone de los siguientes procedimientos:

- **avanzo_cinta()** : Avanza la cinta un paso (muy pequeño).
- **llenar_botella()** : Llena la botella de líquido (finaliza cuando está llena).
- **pongo_tapa()** : Comienza la acción de tapado (finaliza cuando la tapa queda colocada).

Y de las siguientes funciones:

- **puedo_llenar()** : Devuelve TRUE si hay un envase bajo la estación de llenado.
- **puedo_tapar()** : Ídem para la estación de tapado.

Se pide:

(a) Resolver utilizando semáforos.

(b) Resolver utilizando monitores.

Nota: Las botellas vienen a intervalos irregulares y no siempre se está en situación de accionar las dos máquinas a la vez. Se desea conseguir los algoritmos más sencillos posibles que resuelvan correctamente el problema.

Solución: Para modelar este problema debemos tener algunos cuidados:

- No avanzar la cinta mientras se está llenando o tapando.
- No llenar si no hay botella debajo.
- No tapar si no hay botella debajo.
- No tapar ni llenar dos veces (sin avanzar la cinta).

Una parte fundamental antes de comenzar a modelarlo con alguna de las primitivas, es identificar donde podrían existir problemas de concurrencia. En particular podríamos intentar definir cuál es la zona crítica. En este problema, el recuso compartido es claro: la cinta. Por esta razón, podemos comenzar estableciendo a la misma como la zona crítica.

Parte a) Cuando trabajamos con esta primitiva, la idea es modelar el problema con procesos sincronozados mediante semáforos previamente inicializados. Para este problema podemos, en principio, diseñar dos procesos, uno por cada una de las tareas que debemos realizar: `Tapar` y `Llenar`.

Si queremos realizar una solución con dos hilos (uno para cada procedimiento) existen al menos dos enfoques, dependiendo a quien encarguemos el manejo de la zona crítica.

El más directo es realizar una solución simétrica. Esto quiere decir que tanto el hilo encargado de `Tapar` como el de `Llenar` van a manejar la cinta por igual y solo se van a diferencial en las tareas propias de cada uno.

Cuidados generales para la implementación con semáforos:

- Todas las variables globales deben estar mutoexcluidas con un semáforo con el objetivo de mantener su consistencia.
- Todo P tiene que tener su correspondiente V. Esto nos evita quedar en deadlock.
- Los semáforos deben estar correctamente inicializados (error común).

Con esto en mente, definimos la idea general de la solución:

- Siempre que se pueda tapar/llenar, se hace.
- Se puede mover la cinta tanto del proceso `Tapar` como `Llenar`, pero se debe asegurar que solo uno lo haga cada vez.
- Se debe, de alguna manera, conocer el "estado" del otro proceso antes de mover la cinta. Esto lo podemos modelar con una variable global que represente simplemente si el otro proceso ya está listo para mover la cinta o no.
- Si esta variable de estado le indica al proceso que es el primero en terminar la tarea que estaba realizando, entonces este deberá esperar a que el otro proceso termine su tarea y mueva la cinta antes de continuar (con esto se evita que la cinta sea movida en medio de una acción como tapar o llenar). Esta "espera" la modelamos con un semáforo, con el objetivo de evitar una espera activa. La idea es que el primero en terminar "se duerma" esperando a que el otro proceso termine de realizar su tarea y avance la cinta. Luego de mover la cinta, es claro que el que realizó esta tarea, debe "despertar" al otro proceso antes de continuar con el flujo usual. De esta manera, logramos sincronizar los procesos.

Solución simétrica con 2 hilos:

```
program Algarrobo_Cola

semaphore: mutex_var, cinta
bool: primero

def Tapar ():
  while True:
    if puedo_tapar():
      pongo_tapa()

    #La variable "primero" es global. Para
    #mantener la consistencia su uso debe estar
    #mutoexcluido por un semaforo.
    P(mutex_var)

    if primero:
      primero = False
      V(mutex_var) #Libero el mutex antes de dormirme para que
                #el otro proceso pueda trabajar con la variable.
                #(Y no entrar en deadlock)
      P(cinta) #Temine mi trabajo, espero a que muevan la cinta.
      V(mutex_var)
      #Si entre en el if, el otro proceso entro en el else,
      #por lo tanto, completo el V que falta.
    else:
      avanzo_cinta()
      primero = True #Ya avance, vuelvo la variable al estado inicial.
```

```
        V(cinta) #Despierto al otro proceso.

    def Llenar():
      while True:
        if puedo_llenar():
          llenar_botella()

        P(mutex_var)

        if primero:
          primero = False
          V(mutex_var)
          P(cinta)
          V(mutex_var)
        else:
          avanzo_cinta()
          primero = True
          V(cinta)

    def main():
      primero = True
      init(mutex_var,1)  #init en 1 porque lo usamos para mutoexcluir
      init(cinta,0) #init en 0 porque lo usamos para sincronizar
      cobegin #Lanzo los procesos de manera concurrente
        Tapar()
        Llenar()
      coend
```

Luego de analizar la solución presentada debería hacerse la siguiente pregunta: ¿Por qué el proceso que entra al flujo por el `else` no libera el mutex que él tomó? (es decir, no ejecuta `V(mutex_var)`). Para entender esto analicemos el siguiente escenario:

- Supongamos que ahora el mutex se libera de la siguiente manera

```
if primero:
  primero = False
  V(mutex_var)
  P(cinta)
else:
  avanzo_cinta()
  primero = True
  V(mutex_var)
  V(cinta)
```

- Ahora suopongamos que el proceso encargado del llenado, realiza la siguiente ejecución:
  1. Ejecuta la función de llenado
  2. Gana el mutex `mutex_var`
  3. Ejecuta el código del if hasta `V(mutex_var)`
  4. Le quitan el procesador antes de la siguiente instrucción
- Por otra parte el proceso encargado de tapar ejecuta de la siguiente manera:
  1. Hace una pasada completa por el código
  2. Vuelve a comenzar por la condición del while
  3. Toma el mutex que él mismo liberó en el paso 1

4. Evalúa la condición del if y le da True pues el mismo puso primero en True
5. Pone primero en False, libera el mutex, hace `P(cinta)` y no se queda trancado pues el mismo hizo `V(cinta)` en el paso 1
6. Vuelve a entrar por el while
7. Vuelve a llenar (sin mover la cinta)
8. Evalúa nuevamente la condición del if y le da False pues el mismo puso primero en False
9. Avanza la cinta. Hace nuevamente `V(cinta)` y repite lo mismo indefinidamente sin que el otro proceso participe

La manera de evitar este problema, es asegurando que se ejecute el `P(cinta)` antes de que el proceso que ejecutó `V(cinta)` vuelva a entrar en el flujo del condicional. Como se mostró en la solución, obligando a liberar el mutex tomado por el proceso que hizo el `V(cinta)` luego del `P(cinta)` podemos evitar esta situación, ya que para la segunda vuelta el proceso no podrá ganar el mutex hasta que el otro se lo libere.

Observar que otro enfoque posible para resolver el problema del tapado y llenado con dos hilos, es pensar una solución asimétrica donde solo uno de los procesos se encargue de mover la cinta.

Por último, se muestra una manera de pensar el problema, delegando la tarea de manejar el acceso al recurso compartido a un hilo independiente que realiza la tarea de "controlador".

Solución con 3 hilos:

```
program Algarrobo_Cola
semaphore: s1, s2, s3
boolean: primero

def Tapar ():
  while True:
    if puedo_tapar():
      pongo_tapa()
    V(s1)
    P(s2)

def Llenar():
  while True:
    if puedo_llenar():
      llenar_botella()
    V(s1)
    P(s3)

def Cinta():
  while True:
    P(s1)
    P(s1)
    avanzo_cinta()
    V(s2)
    V(s3)

def main():
  init(s1,0)
  init(s2,0)
  init(s3,0)
  cobegin
    Tapar()
    Llenar()
    Cinta()
  coend
```

En esta solución, el procedimiento encargado de avanzar la cinta queda trancado por dos semáforos que son liberados a medida que los procedimientos de llenado y tapado finalizan su tarea.

parte b) Los monitores, a diferencia de los semáforos, son una herramienta de sincronización más estructurada. Estos aseguran que solo un proceso a la vez esté activo en los procedimientos del monitor. Es decir, los procedimientos que pongamos dentro del monitor están mutoexcluidos.

La primer pregunta a responder cuando comenzamos a modelar problemas con monitores es ¿qué pondremos dentro del monitor? Aprovechando la ventaja que nos brindan los monitores (mutoexclusión gratis), y habiendo identificado previamente a la cinta como el recurso compartido, tiene sentido que el manejo de la cinta esté dentro del monitor.

Con esto en mente podemos etructurar la solución de la siguiente manera:

- Definimos un monitor para controlar la cinta.
- Dentro del monitor para controlar la cinta debemos definir un procedimiento que siga una lógica similar mostrada en la parte a) donde aseguremos que solo el segundo en terminar su tarea mueva la cinta. Además, mientras esto sucede el primero en terminar debe esperar "dormido".
- Fuera del monitor tendremos dos procedimientos simétricos que van a realizar el tapado y llenado de las botellas, además de hacer uso del monitor para mover la cinta.

Siguiendo tenemos la siguiente solución al problema:

```
monitor ControlarCinta
  boolean: soy_primero
  condition: espero_otro

  def Continuar():
    if soy_primero: #si soy el primero, cambio el estado y me duermo.
      soy_primero = False
      espero_otro.wait()
    else: #si soy el ultimo avanzo la cinta y despierto al que esperaba.
      avanzo_cinta()
      soy_primero = True
      espero_otro.signal()

  def main():
    soy_primero = True

program AlgarroboCola
  def Tapar():
    while True:
      if puedo_tapar():
        pongo_tapa()
      Cinta.Continuar

  def Llenar():
    while True:
      if puedo_llenar():
        llenar_botella()
      Cinta.Continuar

  def main():
    cobegin
      Tapar()
      Llenar()
    coend
```

Ejercicio 2 Una tribu de N caníbales come de una gran marmita común con capacidad para 6 comensales simultáneos. Cuando un comensal quiere comer, come de la marmita, a menos que no haya suficiente comida para él. Si no hay suficiente comida en la marmita, el caníbal despierta al cocinero y espera a que el cocinero haya rellenado la marmita con la carne de los misioneros capturados (no debe haber notificaciones repetidas). Para rellenar la marmita el cocinero debe esperar a que todos los comensales que se encuentran actualmente comiendo terminen. El cocinero, por su parte, vuelve a dormir cuando ha rellenado la marmita. Consideraciones:

- No podrán entrar nuevos comensales a la marmita cuando el cocinero está rellenando o esperando para rellenar.
- Se supone que la marmita llena dispone de comida para más de seis caníbales.

Se dispone de las siguientes funciones:

- **Hay_suficiente_comida():boolean**
  Esta función es ejecutada por los comensales y retorna si hay suficiente comida en la marmita. No puede ser ejecutada por dos o más comensales a la vez.
- **Rellenar()**
  Esta función es ejecutada por el cocinero.
- **Comer()**
  Es ejecutado por los comensales.
- **Ocio()**
  Ejecutada por los caníbales cuando no están comiendo.

Se pide: Implementar los procedimientos caníbal y cocinero utilizando monitores. Especifique la semántica de los mismos en caso de ser necesario.

Solución: Para comenzar a modelar esta solución con monitores debemos definir cuantos monitores vamos a utilizar y que tareas necesitamos en ellos.

La zona crítica que debemos mutoexcluir en este caso, es la Marmita, por lo que tiene sentido utilizar un Monitor para controlar las acciones sobre la misma.

Los controles que debemos realizar son:

- no permitir que haya más de seis caníbales comiendo de la marmita.
- que el cocinero y los caníbales no estén “utilizando” la Marmita a la vez. Es decir, que no se llene la Marmita mientras haya caníbales comiendo.
- el cocinero debe ser despertado por el primer canibal que detecta que no hay suficiente comida.

Se debe tener presente que el acceso a monitor es **individual**, por lo que hay que tener especial cuidado con las operaciones que realizamos dentro del mismo. Con esto presente, definimos los procedimientos que vamos a utilizar en el monitor:

- Un procedimiento para dejar ingresar a comer. Este procedimiento lo utilizarán los caníbales y deberá, en escencia, controlar que haya menos de 6 caníbales comiendo y suficiente comida. Si ya hay 6 caníbales, deberá “esperar” dentro del monitor. Esta espera la modelamos con una variable de tipo *condition*. Si no hay suficiente comida deberá avisar al cocinero, para modelar esto es claro que utilizaremos otra variable de tipo *condition*.

- Un procedimiento para que los caníbales avisen cuando terminaron de comer. En este procedimiento se actualizará la cantidad de caníbales utilizando la marmita y se dejará entrar a otros cuando sea necesario.
- Análogamente podemos utilizar dos procedimientos para la entrada y salida del cocinero.

Notar que con este enfoque los procesos que modean a los caníbales y al cocinero **deben** realizar sus tareas **fuera** del monitor.

```
Monitor Marmita
  integer: cant_canibales # controlo canibales en la marmita
  boolean: rellenar # flag para indicar cuando se debe rellenar
  condition: cnd_cocinero, cnd_canibal, cnd_relleno

  def comer():
    #si no hay lugar o no hay suficiente comida, me duermo
    if cant_canibales  == 6 or rellenar:
      cnd_canibal.wait()

    #si hay lugar pero no comida, levanto la flag
    #si ademas no hay nadie comiendo, despierto al cocinero
    #luego me duermo hasta que rellenen la marmita
    if not hay_suficiente_comida():
      rellenar = True
      if cant_canibales = 0 :
        cnd_cocinero.signal() {*}
      cnd_relleno.wait()

    #si llegue hasta aqui es porque hay lugar y comida
    #actualizo el contador y si hay mas lugar despierto
    #a alguien (si habia, de lo contrario el signal queda
    #sin efecto)
    cant_canibales++
    if cant_canibales < 6:
      cnd_canibal.signal()

  def termine_comer():
    #si termine de comer actualizo el contador
    cant_canibales--

    #si no hay que rellenar y estoy en las condiciones
    #despierto al cocinero, sino hay que rellenar y hay lugar
    #hago signal
    if cant_canibales == 0 and rellenar then
      cnd_cocinero.signal()
    if not rellenar then
      cond_canibal.signal()

  def cocinero_inicio():
    if not rellenar or cant_canibales > 0:
      cnd_cocinero.wait()

  def cocinero_fin():
    integer: i
    rellenar = False
    cnd_relleno.signal()
```

```
    def main():
      cant_canibales = 0
      rellenar = False
end

def canibal ():
  while True:
    #pide para comer
    marmita.comer()
    #cuando sale del monitor ya puede comer
    comer()
    #avisa que termino de comer
    marmita.termine_comer()
    ocio()

def cocinero ():
  while True:
    marmita.cocinero_inicio()
    rellenar()
    marmita.cocinero_fin()
```

(*) Este signal no causa problemas pues el cocinero dejará el monitor inmediatamente al recibir el signal y entonces el caníbal entrará nuevamente a hacer el wait. Podría haber problemas si el cocinero no dejara el monitor luego del signal e hiciera un signal al caníbal ahí mismo pues podria pasar que el caníbal no hubiera llegado a hacer el wait y el signal se perdería.

Para el cocinero, podríamos tener un solo procedimiento en el monitor:

```
def cocinero_inicio():
  cnd_cocinero.wait()
  rellenar()
  rellenar = False
  cnd_canibal.signal()
```

Y el procedimiento cocinero sería:

```
def cocinero ():
  while True:
    marmita.cocinero_inicio()
```

Pero en este caso tendríamos problemas con (*) si el monitor se libera al hacer signal. No daría tiempo a que el canibal haga wait.