Sistemas Operativos
Práctico 5

Curso 2024

Ejercicio 1 Se desea modelar utilizando mailboxes el siguiente problema:

El taller mecánico “Boxes” tiene capacidad para 20 vehículos en sus instalaciones. La entrada de los usuarios a las instalaciones debe ser estrictamente en orden de llegada (la solución no debe permitir bajo ningún concepto “colados”).
El usuario debe ser atendido por el primer mecánico libre de los 5 con que cuenta el taller. El mecánico, una vez terminado el arreglo, le indicará a la caja el monto a cobrar por el arreglo.
El cliente recibirá de la caja el monto a pagar.

Se dispone de las siguientes funciones:

- **arreglar_auto():integer**
  Invocada por un mecánico para arreglar el auto. Retorna el costo del arreglo.
- **pagar_arreglo(Monto)**
  Invocada por el usuario para pagar el monto que le fue indicado por la caja.

Nota:

- Se prohíbe expresamente el busy-waiting.
- Se debe explicitar la semántica de las primitivas de mailbox utilizadas.
- Tener cuidado de que el socio pague el importe correcto del arreglo a la caja.

Solución: Utilizaremos Mailboxes infinitos, send no bloqueante, receive bloqueante.

Comenzando con la solución del problema nos encontramos con que existen tres participantes en la realidad:

- clientes
- mecanicos
- caja

La idea a alto nivel es que el proceso sea:

1. El cliente llega y avisa que tiene un auto para arreglar
2. Un mecánico toma el pedido, arregla el auto y avisa a la caja el costo del arreglo
3. La caja comunica al cliente el monto y queda a la espera de que este pague

La interacción entre el cliente y el mecánico se podría solucionar simplemente con un mailbox, pero si observamos el resto de las interacciones, es claro que vamos a necesitar que los clientes puedan ser identificados de alguna manera. Esto se debe a que el costo del arreglo es único para cada auto y también a que diferentes arreglos pueden requerir diferente cantidad de tiempo.

Para lograr dar un identificador a cada cliente, utilizaremos una técnica de tickets, entregando nosotros un número a cada cliente con el cual será atendido. Tomando ventaja de que el taller tiene una capacidad máxima (20) haremos un mailbox donde los que vayan llegando retirarán su identificador.

```
taller: mailbox (integer);
```

Notar que para que esto funcione debemos "cargar" los número a ser tomados por los clientes. Esto lo haremos al inicio de la ejecución, enviando los números del 1 al 20 como mensajes a ese mailbox (se vé más adelante).

Una vez solucionado eso, podemos hacer que el cliente pida que se arregle su auto enviando su identificador. Creamos un mailbox donde los clientes enviarán sus identificadores y los mecánicos irán tomando los trabajos a medida que estén disponibles. Esto nos asegura un comportamiento FIFO por hacerlo a través del mailbox.

```
mec_cli: mailbox (integer);
```

Siguiendo nuestra estructura de solución de alto nivel, debemos proveer una vía de comunicación entre los mecánicos y la caja, para que esta pueda luego cobrar a los clientes. Los mecánicos deben informar el costo y además enviar el identificador del cliente al que le van a cobrar, aquí tomaremos ventaja de que podemos definir el tipo de dato que trata cada mailbox y crearemos uno que trbajará con tuplas formadas por dos enteros (monto e id)

```
mec_caja: mailbox (integer, interger);
```

Resta entonces declarar las comunicaciones entre la caja y los clientes. Aquí es donde realmente se vuelve útil tener un identificador y saber que está dentro de cierto rango de enteros

$$1, 20$$

, porque declararemos un array de 20 mailboxes donde cada uno será el canal con el cliente que tenga ese identificador.

```
caja_cli: array [1..20] of mailbox (integer);
```

Por último necesitamos que el cliente informe que ya realizó el pago, para que la caja pueda liberarse, esto lo haremos con un mailbox donde no se enviará información sino que la utilizaremos solamente para sincronizar esta transacción.

```
cli_caja: mailbox (null);
```

Ahora empezamos a construir los procesos que harán uso de lo antes declarado. Primero tenemos al cliente:

```
Procedure cliente ()
var     lugar, monto: integer;
begin
        lugar = receive(taller);
        send(mec_cli, lugar);
        monto = receive(caja_cli[lugar]);
        pagar_arreglo(monto);
        send(cli_caja, null);
        send(taller, lugar);
end
```

Notar que al final de su ejecución, el cliente es quien "devuelve" su identificador, enviándolo nuevamente al mailbox taller. Gracias a esto los siguientes clientes podrán seguir siendo atendidos. Si nos olvidásemos de esto, sólo se arreglarían los primeros 20 autos y luego los clientes no tendrían forma de obtener identificadores.

El siguiente es el proceso mecánico, que simplemente tomará el pedido, arreglará el auto e informará a la caja:

```
Procedure mecanico (num: integer)
var monto, lugar: integer;
begin
        while true do
        begin
                lugar = receive(mec_cli);
                monto = arreglar_auto();
                send(mec_caja, (monto, lugar));
        end
end
```

Por último resta especificar el comportamiento de la caja, el cual es en la misma línea de lo explicado anteriormente. Recibe del mecánico el monto y el id del cliente y manda al mailbox ubicado en la posición correspondiente del arreglo el monto a pagar. Luego simplemente espera que el cliente indique que terminó con la transacción:

```
Procedure caja ()
var monto, lugar: integer;
begin
        while true do
        begin
                (monto, lugar) = receive (mec_caja);
                send (caja_cli[lugar], monto);
                receive(cli_caja);
        end
end
```

Para terminar con el ejercicio, debemos declarar la inicialización de los procesos dentro de un cobegin coend, pero dada nuestra solución antes debemos enviar al mailbox taller los números del 1 al 20 que oficiarán de identificadores. Teniendo en cuenta estas dos tareas, el main de nuestro código quedará de la siguiente manera:

```
begin
        for i:= 1 to 20 do
                send(taller, i);
        end
        cobegin
                mecanico(1);
                mecanico(2);
                mecanico(3);
                mecanico(4);
                mecanico(5);
                caja();
                cliente();
                ...
```

```
                cliente();
        coend
end
```

Nota: Puede resultar atractivo (sobre todo cuando la cantidad de procesos concurrentes es mayor) cambiar el código dentro del cobegin-coend por uno o varios for dentro de los cuales se ejecuten los procesos iguales. Esta solución no es correcta ya que dentro del cobegin-coend el for es tratado como un único proceso que es ejecutado de forma serial por tanto **es necesario** colocar los procesos tal como se encuentran en la solución.

Ejercicio 2 (Presentación ejercicio 7) Sea un consultorio médico, el cual posee una sala de espera en la cual pueden haber como máximo 10 pacientes. El médico atiende a una sola persona de la sala de espera por vez, teniendo prioridad los niños. Cuando el médico termina de atender al paciente deja que el próximo entre. Si no hay ninguno lee durante cinco minutos y vuelve a ver si hay alguien. Si no lo hay vuelve a leer y así sucesivamente.

Se dispone de las siguientes funciones:

- **atender()**
  Invocada por el doctor para atender un paciente.
- **leer_diario()**
  Invocada por el doctor para leer el diario.

Se pide: Representar mediante tareas de ADA los pacientes, el doctor y la sala de espera.

Nota: El paciente debe solicitar permiso al portero para entrar a la sala y al médico para entrar al consultorio.

Solución:

Lo primero que debemos hacer es, una vez más, diagramar a alto nivel las interacciones que se darán en la realidad que queremos modelar. De esta manera esperamos encontrar las dificultades a sortear y ordenar un poco la letra del ejercicio.

En esta letra hay cuatro entidades:

- Doctor
- Niños
- Adultos
- Sala de espera

Notar que la sala de espera será representada por un portero.

Según la letra, y simplificando a los pacientes (niños y adultos por igual) podríamos definir que el flujo normal es:

1. Un **paciente** llega y avisa al portero
2. El **portero** chequea que haya lugar y lo deja o no entrar a la sala

3. El **paciente** pide al médico para ser atendido
4. El **médico** atiende al paciente (recordar que debe respetar cierta prioridad)
5. El **paciente** avisa al portero que se retira y lo hace

Siguiendo esa estructura, parece razonable comenzar a definir las diferentes tareas. Empezaremos por los pacientes, que son los que tienen el comportamiento más simple. Como no tenemos algun tipo de primitiva para distinguir el tipo de los pacientes, definiremos una tarea para adultos y otra para niños, ya que es importante diferenciar los pedidos de atención a la hora de comunicarse con doctor para poder asegurar la prioridad. De esta manera, aunque no hayamos implementado la tarea doctor aún, podemos asumir que contará con una ENTRY para cada tipo de paciente. Las llamaremos **adulto** y **niño**. También usaremos las entry **entrar** y **salir** para comunicarnos con la task portero.

```
task type NIÑO;
task body NIÑO is
begin
  portero.entrar;
  doctor.niño;
  portero.salir;
end NIÑO;

task type ADULTO;
task body ADULTO is
begin
  portero.entrar;
  doctor.adulto;
  portero.salir;
end ADULTO;
```

Notar que las declaramos como **task type** porque habrá más de una instancia de este tipo.

Siguiendo el orden de complejidad, implementaremos ahora la tarea **portero**, que se encargará de que la restricción de que sólo puede haber 10 personas en la sala de espera se cumpla. Como ya dijimos anteriormente, este contará con dos entry (**entrar** y **salir**). Este tipo de tarea es ampliamente utilizada cuando queremos restringir el acceso a un recurso. Lo que hará es llevar un contador que aumentará cada vez que alguien entre a la sala y disminuirá cuando salgan. Para mantener la cantidad de personas dentro menor a 10, simplemente dejará de aceptar encuentros en la entry **entrar** cuando este contador sea igual a 10.

```
task PORTERO is
  entry entrar;
  entry salir;
end PORTERO;

task body PORTERO is
  n:integer;
begin
  n := 0;
  loop
    select
    when (n < 10) =>
      accept entrar;
```

```
        n := n+1;
    or
      accept salir;
        n := n-1;
    end select;
  end loop;
end PORTERO;
```

El comportamiento más complejo es el de la tarea doctor. Aquí debemos implementar una prioridad que distinga entre niños y adultos y además manejar el caso en el cual no hay nadie en la sala de espera y el doctor se pone a leer. Para definir la prioridad, haremos uso de la herramienta **COUNT** que nos brinda ADA, con esta podemos crear una guarda que dependa de la cantidad de gente que esté esperando un encuentro en alguna de las ENTRY de la task. En este caso, sólo aceptaremos encuentros para atender adultos, cuando no hay nadie esperando por un encuentro en niño (es decir, no hay ningún niño sin atender). También colocaremos un **else** en nuestro select, esto se debe a que en el caso de que cuando el doctor "consulte" la sala, si no hay nadie esperando no queremos que se bloquee hasta que alguien pida un encuentro, sino que queremos que se ponga a leer el diario. Teniendo estas consideraciones podemos implementar la tarea doctor de la siguiente manera:

```
task DOCTOR is
  entry niño;
  entry adulto;
end DOCTOR;

task body DOCTOR is
begin
  loop
    select
      accept niño do
        atender();
      end niño;
    or when niño'count = 0  =>
      accept adulto do
        atender();
      end adulto;
    else
      leer_diario();
    end select;
  end loop;
end DOCTOR;
```

Ejercicio 3 (Presentación ejercicio 9) Se desea modelar en ADA una heladería. En la heladería hay dos vendedores y cinco empleados que arman los helados. Los cucuruchos más grandes solo pueden ser armados por los empleados más experientes por lo que, dependiendo del helado solicitado por el cliente, no necesariamente podrá ser atendido por cualquier empleado.

Cuando llega un cliente le hace el pedido al vendedor y este consulta **simultáneamente** a los cinco empleados para ver si alguno está libre y es capaz de armar el helado solicitado (asumiremos que se prepara un helado por cliente). El primer empleado que conteste afirmativamente será el encargado de armar el helado solicitado y entregárselo al cliente quien esperará hasta que el pedido esté pronto. A los otros empleados disponibles (en caso que los haya) se les indicará que el pedido ya fue atendido

por otro de ellos. En caso de que todos estén ocupados se le indica al cliente que no se le puede vender en este momento y se retira.

Cada cierto tiempo llega un supervisor de bromatología el cual verifica que los helados no estén contaminados. Mientras el supervisor trabaja no se podrá armar ningún helado (los empleados pueden terminar de armar los que ya habían empezado antes de que llegue el supervisor). El supervisor tiene prioridad sobre los clientes. Es válido rechazar clientes nuevos mientras el inspector está trabajando.

Se dispone de los siguientes procedimientos auxiliares:

- `elegir_vendedor(): {1,2}` que ejecutado por un cliente le indica el número de vendedor con quien comunicarse.
- `que_helado(): pedido` que ejecutado por un cliente retorna un pedido con el helado que desea el cliente
- `comer_helado(helado)` ejecutado por el cliente para comer el helado
- `puedo_armar_helado(pedido): boolean` ejecutado por el empleado para ver si puede armar el helado
- `armar_helado(pedido): helado` que ejecutado por un empleado arma el helado solicitado
- `verificar_helados()` ejecutado por el inspector para verificar el estado de los helados en el local
- `otras_tareas_inspector()` ejecutado por el inspector cuando no esta inspeccionando

Se pide: Implementar en ADA las tareas vendedor, empleado, cliente e inspector. Se puede usar hasta una tarea auxiliar.

Solución: Para comenzar a solucionar el ejercicio, debemos entender las interacciones entre las tareas (y evaluar si vamos a necesitar una tarea auxiliar). Intentando ordenar la letra y diagramar las comunicaciones podemos decir que a alto nivel, en el caso "feliz" donde el pedido es realizable y sin tener en cuenta al inspector, el comportamiento sería:

1. El cliente llega, elige su helado y su vendedor.
2. El cliente solicita el helado al vendedor.
3. El vendedor chequea si hay algún empleado que pueda hacer el pedido.
4. El vendedor avisa al cliente qué empleado lo realizará.
5. El empleado arma el helado y se lo da al cliente.

En este caso iremos manejando los casos alternativos a medida que se presentan en el diagramado de la soución, porque el manejo de prioridades en ADA es relativamente sencillo, utilizando la primitiva COUNT'.

La primer tarea a implementar será la de el cliente. Esta decisión fue tomada en base a que este es el que inicia la interacción.

Primero la creamos, sabiendo que según nuestro diagrama, este no necesita ninguna Entry:

```
task type cliente is
end cliente;
```

El cliente deberá comunicarse con alguno de los vendedores (el indicado por la respuesta de **elegir_vendedor()**). Aquí tenemos que tomar nuestra primer decisión: ¿Cómo le hablaremos a cada uno de los vendedores por separado?

Usamos la solución clásica en ada, donde colocaremos a los vendedores dentro de un arreglo, indexados por su identificador (más adelante veremos como se crea este arrego y como indicamos los identificadores a los vendedores).

A su vez, en la comunicación con el vendedor deben suceder tres cosas: (1) el cliente debe indicarle qué helado quiere, (2) el vendedor debe responderle si existe la posibilidad de hacerlo y en caso de que se vaya a hacer (3) indicarle el identificador del empleado que lo hará (esto último se debe a que de aquí en más el cliente esperará el helado por parte del empleado para dejar al vendedor libre).

Definiendo el comportamiento del cliente, agregamos la línea donde enviamos el tipo de helado al vendedor elegido y esperamos en las variables **empleado** y **ok** lo que este responda.

```
task body cliente is
var
        empleado: integer;
        h: helado;
        ok: boolean;

begin
vendedores[elegir_vendedor()].pedir_helado(que_helado(), empleado, ok);
```

Lo siguiente que debemos hacer es decidir en función de lo que se haya cargado en la variable **ok** si esperaremos el helado o no.

Una idea común es diagramar la entrega del helado por parte del empleado como un encuetro que el cliente acepta. El problema con esta solución es que el empleado debería saber cómo llamar directamente a los clientes, pero estos no tienen forma de ser indentificados (es más fácil dar un identificador fijo a cada uno de los empleados). Por esto el vendedor nos debe indicar en la parte anterior a qué empleado consultar y es el cliente quien llama al encuentro, que el empleado estará esperando cuando termine el helado.

Agregamos entonces las siguientes líneas:

```
if ok then
        empleados[empleado].entregar_helado(h);
        comer_helado(h);
end if
end
```

Y con eso terminamos la implementación de la tarea Cliente.

La siguiente que implementaremos será la tarea **vendedor**, siguiendo con el flujo de nuestro diagrama.

Lo primero que haremos es definir sus entry y sus variables, las cuales iremos describiendo y justificando a medida que sean necesarias:

```
task type vendedor is
entry pedir_helado(p: IN pedido, empleado: OUT integer, ok: OUT boolean);
entry respuesta_empleado(resp: IN boolean, id: IN integer, ok: OUT boolean);
entry obtener_id(id: IN integer);
end
```

```
task body vendendor is
var
        mi_id, cant, i, id_c: integer;
        resp, primero: boolean;
```

Lo primero que debe hacer el vendedor, es conocer su identificador, de esta manera podrá mandarselo a todos los que necesiten luego responderle algún tipo de mensaje.

Para resolver esto, creamos la entry **obtener_id** mediante la que recibiremos nuestro identificador (que se corresponde con nuestro indice en el arreglo de vendedores). Apenas iniciamos la ejecución esperamos este encuentro y cargamos la variable **mi_id**.

```
begin
accept obtener_id(id: IN integer) do
        mi_id := id;
end accept
```

Luego de obtener su identificador, el vendedor puede comenzar su loop de trabajo.

En este se encontrará la parte más compleja de la solución, donde las interacciones pueden ser un poco mas complicadas.

El vendedor esperará que un cliente le pida un helado y cuando eso suceda, deberá buscar un empleado libre capaz de hacerlo.

El funcionamiento, dentro de la entry **pedir_helado** será el siguiente:

1. Intento avisar a todos los empleados del pedido que recibí y registro a cuántos efectivamente logré contactar.
2. Chequeo las respuestas de los empleados, hasta que encuentre uno que pueda hacerlo. A este le indico que será él quien lo prepare (a los que no pueden hacerlo, les indico que no lo prepararán).

Para solucionar el item 1 recorreremos el arreglo de empleados, buscando un encuentro de tipo **puede_preparar** con cada uno de ellos. Como sabemos que puede haber empleados que no estén en ese momento aceptando dicho encuentro, utilizaremos la estructura de **select-else**, donde si le encuentro solicitado en el select no es aceptado inmediatamente el programa pasa al else. **Notar que esta estructura no soporta or, como su equivalente cuando aceptamos entries.**

Implementando lo descrito anteriormente, agregamos a la solución:

```
loop
cant := 5;
accept pedir_helado(p: IN pedido, empleado: OUT integer, ok: OUT boolean)
do
        for i:= 1 to 5 do
        select
                cocineros[i].puede_preparar(mi_id, p);
        else
                cant := cant - 1;
        end select
        end for
        ok := false;
        if cant > 0 then
                while (cant > 0) and (not ok) do
                        accept respuesta_empleado(resp: IN boolean,
```

```
                                id_c: IN integer, ok_empleado: OUT boolean) do
                                        if resp then
                                                ok_empleado := true;
                                                ok := true;
                                                empleado := id_c;
                                        else
                                                ok_empleado := false;
                                        end
                                end accept
                                cant := cant - 1;
                        end while
                end if
        end accept
```

Una vez termiando el encuentro con el cliente, todavía pueden haber empleados esparando para responder si pueden realizar el pedido solicitado. A estos les respondemos fuera de la parte anterior, para evitar que el cliente espere más de lo debido.

El código para esto queda:

```
        while (cant > 0) do
                accept respuesta_empleado(resp: IN boolean, id_c: IN integer,
                 ok_empleado: OUT boolean) do
                        ok_empleado := false;
                end accept
                cant := cant - 1;
        end while
        end loop
        end
```

Pasamos ahora a la implementación de los empleados. Aquí es donde debemos tener en cuenta por primera vez a los inspectores, ya que estos tiene prioridad sobre los empleados para acceder a los helados.

Crearemos la task con las entries ya antes utilizadas y le sumaremos dos para que los inspectores puedan entrar y salir.

```
        task type empleado is
                entry puede_preparar(vendedor: IN integer, p: IN pedido);
                entry entregar_helado(hel: OUT helado);
                entry llega_inspector();
                entry sale_inspector();
                entry obtener_id(id: IN integer);
        end
        task body empleado is
        var
        vend, mi_id: integer;
        ok: boolean;
        p: pedido;
        h: helado;
```

De la misma manera que los vendedores, los empleados deben saber su identificador, ya que lo usan para responder a los vendedores cuando pueden (o no) preparar un pedido. Usaremos la misma técnica para sortear esta dificultad:

```
begin
accept obtener_id(id: IN integer) do
        mi_id := id;
end accept
```

Primero consideremos la solución si no existiesen los inspectores. En ese caso, el empleado debe:

1. Esperar un encuentro por parte del vendedor (**puede_preparar**) que le indicará el pedido y el id del vendedor al cual debe responder.
2. Responder a dicho vendedor con el resultado de la función **puedo_armar_helado**.
3. Si el vendedor indica que realice el pedido, armarlo y esperar que el cliente me lo pida por **entregar_helado**

Esto se vería de la siguiente manera:

```
accept puede_preparar(vendedor: IN integer, ped: IN pedido) do
        vend := vendedor;
        p := ped;
end accept;
vendedores[vend].respuesta_empleado(puedo_armar_helado(p), mi_id, ok);
if ok then
        h = armar_helado(p);
        accept entregar_helado(hel: OUT helado) do
                hel := h;
        end accept;
end if;
```

Habiendo solucionado la interacción de todo el caso, ahora solo nos resta agregar la posibilidad de que en cualquier momento un inspector quiera entrar.

El primer acercamiento a esto puede ser utilizar un **select-or** donde al mismo tiempo pueda venir un pedido para preparar o ingresar un inspector. Esta solución es correcta, pero no implementa la prioridad que tienen los inspectores sobre los pedidos.

Para representar la prioridad, pondremos una guarda en el accept de **puede_preparar**, que indicará que sólo aceptaremos este encuentro cuando no haya nadie solicitando uno del tipo **llega_inspector**. Esto se realiza con la primitiva 'COUNT.

En el caso de que si haya un inspector, simplemente aceptaremos que entre y aceptaremos que salga, dejando el trabajo de avisar a **todos** los empleados en manos del inspector.

De esta manera, el código del loop en el empleado queda de la siguiente manera:

```
loop
select
        when llega_inspector'count = 0 =>
                accept puede_preparar(vendedor: IN integer,
                 ped: IN pedido) do
                        vend := vendedor;
                        p := ped;
                end accept;
                vendedores[vend].respuesta_empleado(puedo_armar_helado(p),
                 mi_id, ok);
                if ok then
```

```
                        h = armar_helado(p);
                        accept entregar_helado(hel: OUT helado) do
                        hel := h;
                        end accept;
                end if;
        or
                accept llega_inspector();
                accept sale_inspector();
        end select
        end loop
        end
```

Por último nos resta implementar la tarea inspector. Esta es relativamente simple, ya que solo debe avisar a todos los empleados que quiere entrar (ellos le darán prioridad), verificar los helados y volver a avisar a todos que terminó:

```
        task inspector is
        end

        task body inspector is
        begin
        loop
                otras_tareas_inspector();
                for i := 1 to 5 do
                        empleados[i].llega_inspector();
                end for
                verificar_helados();
                for i := 1 to 5 do
                        empleados[i].sale_inspector();
                end for
        end loop
        end
```

Para terminar con nuestra solución, debemos crear los arreglos de las tareas empleados y vendedores y enviarle a cada una de ellas su identificador.

La primer parte es simple:

```
        empleados: array[1..5] of empleado;
        vendedores: array[1..2] of vendedor;
```

Luego, en el programa principal, enviaremos a cada uno de los vendedores su index en el arreglo y lo mismo para los empleados (con la ayuda de un loop)

```
        // programa principal
        var i: integer;

        begin
        vendedores[1].obtener_id(1);
        vendedores[2].obtener_id(2);

        for i := 1 to 5 do
                empleados[i].obtener_id(i);
        end for
        end
```