UNIVERSIDAD ORT
Uruguay

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1

-

Sistemas Operativos

Autores:

Joaquín Costa - 330734

Ezequiel Rivero - 302985

Tomás Dotta - 326829

Tutor:

Anuar Maubrigades
Enrique Latorres

2025

UNIVERSIDAD ORT
Uruguay

UNIVERSIDAD ORT
Uruguay

Ejercicio 1

El ejercicio consiste en desarrollar un sistema de gestión para la empresa Citadel, que fabrica pinturas para miniaturas. El sistema debe permitir la gestión de usuarios, ingreso y venta de productos y filtrado por tipo. En Bash.

Decisiones de diseño

- Almacenamiento:
    - Usuarios en users.txt (formato: usuario contraseña).
    - Productos en productos.csv (formato: código,tipo,modelo,descripción,cantidad,precio).
- Archivos Temporales: se usa un archivo temporal session.tmp para saber si hay usuario logueado.
- Restricciones: se controlan duplicados, campos vacíos/no válidos y stock en ventas (tuvimos el problema de que se pueden ejecutar comandos de bash cuando se espera un input en consola).

Decisiones por sección

- **“ $? ”** Se usa en la sección crearUser y otras funciones como login.
  ¿Qué hace **$?** devuelve el código de salida del último comando ejecutado:
  0 = éxito . Distinto de 0 = error o no coincidencia.
- **“-s”** session.tmp Se usa en la función validarLogin. **“-s”** verifica si el archivo existe y no está vacío.
- **“while IFS= read -r line; do ... awk ... xargs”** Se usa en ingProd, vendProd y filterProd.
    - While IFS= read -r line lee el archivo línea por línea.
    - awk -F',' '{print $N}' extrae la columna N del CSV.
    - xargs limpia espacios sobrantes (trim).

      Lo usamos para obtener datos de los productos ya que awk funciona para columnas de un csv

UNIVERSIDAD ORT
Uruguay

Uso de IA en el Ejercicio 1

awk para actualizar stock:

```
241     col=5
242     valor=$((cantidad - cant))
243     # linea hecha con IA
244     awk -v r="$num" -v c="$col" -v v="$valor" 'BEGIN{FS=OFS=","} NR==r{$c=v}1' \
245     productos.csv > productos.csv.tmp && mv productos.csv.tmp productos.csv
246
```

En productos.csv

- -v r="$num": pasa a awk el número de fila que queremos editar .
- -v c="$col": pasa el número de columna a editar (5) que es el de la cantidad.
- -v v="$valor": pasa el nuevo valor (stock actualizado).
- BEGIN{FS=OFS=","}: fija el separador de campos de entrada y salida a la coma (CSV).
- NR==r{$c=v}1:
    - NR==r: cuando el número de registro (línea actual) es la fila r
    - {$c=v}: asigna a la columna c el valor v .
    - 1: significa “imprimir la línea”. Como no se filtra nada, imprime todas; con la edición aplicada en la fila r.

UNIVERSIDAD ORT
Uruguay

Ejercicio 2

El ejercicio 2 plantea una variante del clásico problema de lectores y escritores, aplicada a un panel de información de un aeropuerto. En este escenario interactúan 100 pasajeros (lectores) y 5 oficinistas (escritores).
Los pasajeros consultan el cartel, mientras que los oficinistas realizan actualizaciones críticas que deben ejecutarse sin interferencias. El objetivo es diseñar un mecanismo de sincronización que garantice que:

- Los pasajeros pueden leer el cartel siempre que no exista una modificación en curso.
- Los oficinistas sólo puede n modificar el cartel cuando no haya lectores activos.
- Cada oficinista realiza al menos 3 actualizaciones, con tiempos aleatorios de espera (máx. 5 s para oficinistas y 3 s para pasajeros).

Esto asegura que ningún pasajero lea el panel mientras está siendo actualizado.

Decisiones de diseño

Para lograr una correcta sincronización entre lectores y escritores se decidió el uso de un sistema de tickets (funcionamiento semejante a una cola).
Dentro del código podemos encontrar dentro del sistema de tickets los siguientes elementos:

Variables globales

```
int next_ticket = 0;
int turno_actual = 0;
pthread_mutex_t ticket_mutex = PTHREAD_MUTEX_INITIALIZER;
```

representando **next_ticket** el número del próximo turno a otorgar y **turno_actual** el número del turno que debe ejecutarse en ese momento. Acompañado de **ticket_mutex**
El cual sirve para asegurar la exclusión mutua de los hilos a la hora de modificar el siguiente turno (**next_ticket**).

UNIVERSIDAD ORT
Uruguay

Funciones

```
void esperar_turno(int* mi_turno) {
    pthread_mutex_lock(&ticket_mutex);
    *mi_turno = next_ticket++;
    pthread_mutex_unlock(&ticket_mutex);

    while (turno_actual != *mi_turno) {
        sched_yield();
    }
}

void liberar_turno() {
    __sync_fetch_and_add(&turno_actual, 1);
}
```

Se tiene 2 funciones para lograr el intercalamiento al estilo de una cola:
esperar_turno:
La cual recibe como parámetro el turno actual, el cual será el turno del hilo en ejecución, y aumenta al ejecutarse en uno al siguiente turno para avanzar así al siguiente pasajero u oficinista siempre y cuando este la variable libre para su modificación con **ticket_mutex**. Además siempre que el **turno_actual** global no sea el mio voy a esperar a que llegue mi turno con **sched_yield()**(es una llamada al sistema que indica al planificador del sistema operativo que el hilo actual está dispuesto a ceder voluntariamente el procesador).
liberar_turno:
Que lo único que hace es **__sync_fetch_and_add(&turno_actual, 1)** una función la cual hace los siguientes pasos:

1. Lee el valor actual de **turno_actual**.
2. Suma 1.
3. Escribe el valor actualizado.
4. Devuelve el valor anterior.

Esto lo que logra es avanzar el turno actual en 1 para poder seguir con los demás.

Hilos:

Se tienen los hilos escritor y lector los cuales su única función es esperar su turno, ejecutar su tarea la cual puede ser modificar o mirar el cartel y liberar su turno.

UNIVERSIDAD ORT
Uruguay

Main:

El main se encarga de preparar y coordinar la creación de todos los hilos del sistema, logrando dividir su funcionamiento en 4 etapas principales:

1. **Inicialización de los datos**
   Se inicializan los arreglos con los identificadores de los lectores y escritores, además de un arreglo con todos los hilos del sistema y otro de roles. El cual cuenta con 100 posiciones de ‘L’ y 15 de ‘E’. Simulando las instancias de Lectores y Escritores respectivamente, siendo 15 escritores porque cada uno de los 5 hace 3 modificaciones quedando así en 15 instancias.

2. **Mezcla aleatoria de roles**
   Se realiza una mezcla de los roles para simular una situación real en la que cada lector y escritor llega en un momento aleatorio sin un orden fijo predecible, Dejando de esta forma el orden final de ejecución al sistema de tickets.

3. **Creación de hilos**
   se recorre el arreglo de roles y se crea un hilo lector o escritor dependiendo la letra que salga de roles. a cada hilo se le inicializa con un id numérico único.

4. **Espera de los hilos**
   Para finalizar se espera la liquidación de cada uno de los hilos ejecutados en el programa.

UNIVERSIDAD ORT
Uruguay

Ejercicio 3

El ejercicio 3 se basa en la creación de una simulación de tiempos en las que 100 “vacas” van pasando por una zona de vacunación o ordeñe, y luego suben a uno de dos camiones de capacidad de 50 vacas.

Decisiones de diseño

Para lograr el objetivo principal del ejercicio, el manejo de hilos y procesos, se decidió solo hacer una procedure main, pero que cada tarea sea una task, las salas y el pasillo son tasks que reciben parámetros según la vaca y la acción, estas solo devuelven que la vaca está realizando su debida acción, luego la task vaca genera un numero random para decidir qué acción va a hacer, y se ejecuta con su id (si es posible y no redundante) .

UNIVERSIDAD ORT
Uruguay

Decisiones de Código

Subtype y range

Define un subtipo restringido de Integer con un rango válido.

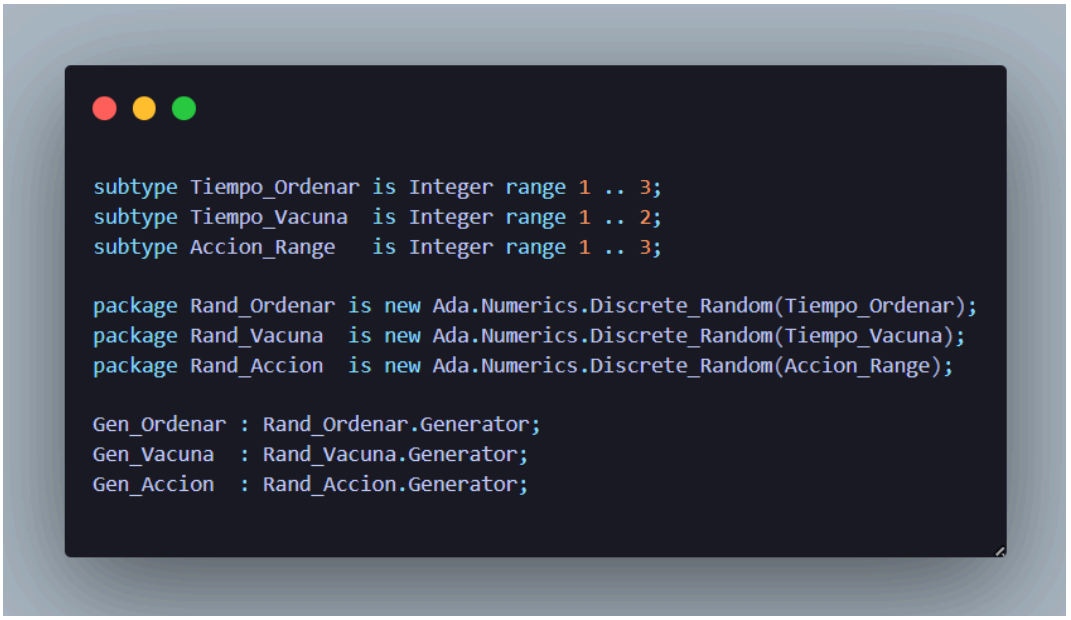
Esto fuerza que cualquier valor de ese subtipo esté entre X e Y,lo usamos para instanciar generadores aleatorios con límites, por ejemplo el tiempo de una vaca en un tambo está entre 1 y 5.

Instancias de paquetes genéricos

“Discrete_Random” es un paquete genérico que necesita un tipo discreto.
is new crea una instancia especializada para generar números aleatorios en el rango del subtipo.

Generadores

Cada paquete instanciado define un tipo Generator.
Se usa Random(Gen) para obtener un valor aleatorio dentro del rango y Reset(Gen) para inicializar la semilla.



```
subtype Tiempo_Ordenar is Integer range 1 .. 3;
subtype Tiempo_Vacuna  is Integer range 1 .. 2;
subtype Accion_Range   is Integer range 1 .. 3;

package Rand_Ordenar is new Ada.Numerics.Discrete_Random(Tiempo_Ordenar);
package Rand_Vacuna  is new Ada.Numerics.Discrete_Random(Tiempo_Vacuna);
package Rand_Accion  is new Ada.Numerics.Discrete_Random(Accion_Range);

Gen_Ordenar : Rand_Ordenar.Generator;
Gen_Vacuna  : Rand_Vacuna.Generator;
Gen_Accion  : Rand_Accion.Generator;
```

UNIVERSIDAD ORT
Uruguay

task , entry e accept

```
task body Sala is
      Cant : Integer := 0;
   begin
      loop
         accept Operar(ID : Integer; Accion : String; Exito : out Boolean) do
            if Accion = "Entrar" then
               if Cant < 15 then
                  Cant := Cant + 1;
                  Exito := True;
                  Put("Vaca "); Put(ID); Put_Line(" entra a la sala");
               else
                  Exito := False;
               end if;
            else
               Cant := Cant - 1;
               Exito := True;
               Put("Vaca "); Put(ID); Put_Line(" sale de la sala");
            end if;
         end Operar;
      end loop;
   end Sala;

   task Mangas is
      entry Operar(ID : Integer; Accion : String; Exito : out Boolean);
   end Mangas;
```

task mangas is entry Operar()

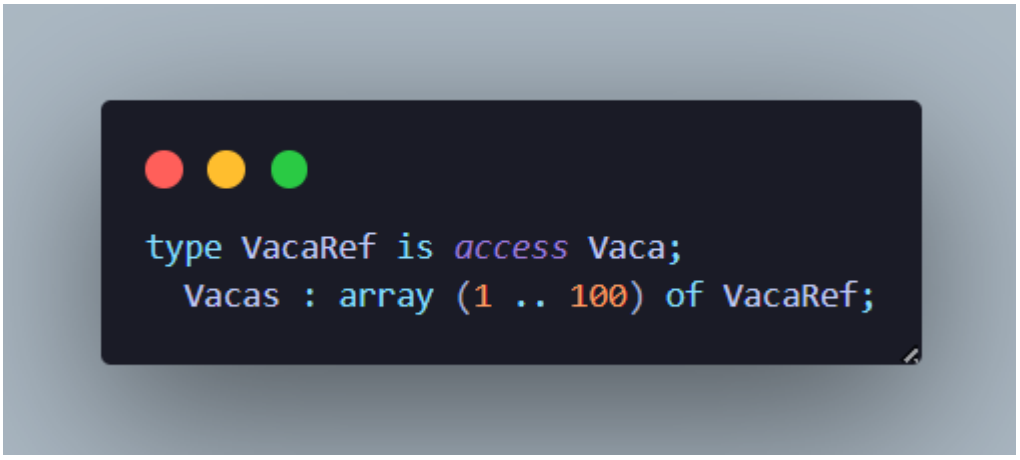
Declara una tarea concurrente con una entrada (entry) que otros pueden invocar. Las entradas son puntos de sincronización para comunicación entre tareas, permitiendo pasar parámetros y recibir resultados.

accept

La tarea se bloquea hasta que otra invoque la entrada.
Se ejecuta el bloque do ... end, esto es lo que nos asegura la sincronización

UNIVERSIDAD ORT
Uruguay

Tipos acceso y creación dinámica



Access define un puntero seguro a una tarea y **new Vaca(I)** crea dinámicamenteuna tarea concurrente con el parámetro ID desde el 1 al 100.

Esto fue extraído de la pagina link, para resolver el problema de que se ejecutaban los tasks secuencialmente. ya que trabajamos con el objeto vaca, para poder usar estrategias con las que estamos más familiarizados.

Control de Concurrencias

Las salas, mangas y camiones se manejan con variables internas (Cant, C1, C2, etc) y accept.
Si no hay espacio, la entrada devuelve Exito=False y la tarea reintenta tras un delay.

UNIVERSIDAD ORT
Uruguay

Ejercicio 4

El ejercicio 4 se centra en la virtualización de un repositorio web, permitiendo que el mismo soporte flujo de distintas redes conectadas, así como almacenar la información de las mismas. Esto se logra utilizando docker y las diferentes herramientas que este brinda junto a otros agentes externos.

Contamos con 3 archivos clave, **docker-compose** y dos **Dockerfile**, uno dentro de la carpeta de **nginx** y el siguiente dentro de la carpeta **web.**

Instancias web

Aquí creamos una imagen de node, y ejecutamos el comando básico para contar con las implementaciones del servicio, que es npm install. A su vez al utilizar la etiqueta 'alpine' indicamos que deseamos la última versión disponible. Por último indicamos que el puerto 5000 será dónde se ejecute node, de esta forma logramos que todos los servicios están respondiendo y leyendo al mismo puerto. Para inicializar el servicio se utiliza el comando node y luego se llama al archivo del servidor (server.js).

```
FROM node:alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 5000
CMD ["node", "server.js"]
```

Instancia Nginx

Ya contábamos con un archivo nginx.conf brindado por la rúbrica, valga mencionar que el mismo fue modificado dado que daba fallas ya que faltaban instancias en el mismo.
Respecto al archivo de docker, el mismo únicamente descarga la imagen web de nginx, de tipo alpine, e intercambia el archivo nginx.conf de la imagen web por el creado en el repositorio local. La lógica de distribución balanceada entre los diferentes servidores

declarados se encuentra dentro de este archivo, al utilizar la expresión “”upstream loadbalancer”.

Es así que teniendo las dos partes por separado de la aplicación, falta el ensamblaje que será realizado en el archivo docker-compose.

Docker-Compose

Este documento inicializa los **servicios necesarios para el correcto funcionamiento de la web**, este cuenta con varios apartados tanto de backend como de frontend. Aclaración global para todos los contenedores utilizados, la etiqueta “restar: always” permite que cada vez que sucede un error inesperado, o algún otro problema el contenedor se reinicie. Por otro lado, la etiqueta “hostname” únicamente renombra el container con un nombre específico.

Primero iniciamos el dockerfile que se encontraba en la carpeta web. Iniciando así la aplicación. Indicamos que va a ser hosteado el servicio utilizando redis, el puerto del mismo, y el puerto en el que se encontrará el sitio web. Al final, en networks, aclaramos que la web tendrá acceso tanto a la capa de front-end como a la de back-end.

Segundo, inicializamos nuestra base de datos (redis-db), que será realizada con redis, por eso mismo creamos un contenedor utilizando la imágen web de redis e indicamos que vamos a utilizar el contenedor de redis para almacenar datos. A su vez vinculamos este contenedor al backend, ya que almacenará datos.

Por último, se crea el contenedor correspondiente al nginx, en el frontend, y lo inicializamos basándonos en el Dockerfile dentro de la carpeta nginx. Luego abrimos públicamente un puerto, el cual será el de acceso, que es el 8080. Este puerto redirecciona al usuario al puerto 80, manejando por nginx, y este nuevamente lo redirecciona al 5000, que es en donde se está ejecutando la app. Esto último es logrado al implementar la cláusula de “depends on”.

Al final del documento se indica el volumen en el que se almacenarán los datos, llamado “redis_data” y se indican las diferentes capas del servicio. Para inicializar la aplicación únicamente debemos de correr el siguiente comando y con eso estará todo.