UNIVERSIDAD PRIVADA DE SANTA CRUZ DE LA SIERRA

SI322 - A - Laboratorio de Sistemas Operativos II

Docente: Mgs. Heber Tito Zúñiga Morales

Segundo Parcial - Ejercicios con Semáforos

INTEGRANTES:

DYLAN JOEL URIBE BLATNIK
WERNER HOLTERS ZABALA
KENNETH MAURICIO FLORES
ANTONIO MIGUEL NATUSCH ZARCO

Índice

Antecedentes

La computadora como herramienta indispensable de trabajo para la creación de soluciones informáticas tiene muchas similitudes con sus creadores. Primeramente, tiene componentes físicos -denominados “hardware” en su conjunto- que permite, mediante fenómenos físicos, realizar las acciones que se le son pertinentes según la capacidad que posean. Estos componentes físicos, sin embargo, no son capaces de hacer nada por sí mismos si es que no existe un porqué que actúe sobre ellos. En el caso de los seres humanos, lo denominamos pensamiento -producto del raciocinio-, y en el de las computadoras, una instrucción -producto de interacciones propias de sus componentes lógicos o “software” en su conjunto-.

Al igual que el ser humano que posee un sistema nervioso el cual se encarga de, entre otras cosas, enviar impulsos eléctricos que se encargan de activar los componentes físicos visibles, el computador posee un elemento importante que se encarga de administrar todos los recursos físicos mediante cálculos aritmético-lógicos denominado sistema operativo.

Iglesias (2023) define a este sistema como “un conjunto de programas de propósito específico que se ejecutan en un sistema informático y que le permiten funcionar correctamente. Estos programas realizan tareas básicas como reconocer las entradas del teclado, mantener un registro de los archivos y carpetas almacenados, enviar información a la pantalla y controlar los dispositivos periféricos.”

A su vez, describe que los sistemas operativos tienen varios cometidos, entre ellos:

- Se encargan de la gestión de los recursos del sistema, como por ejemplo el procesador, la memoria y los dispositivos de entrada y salida, asignando dichos recursos a las distintas aplicaciones informáticas, garantizando que éstas no interfieran entre sí.
- Abstraen el hardware del ordenador, evitando que usuarios y programadores tengan que conocer sus detalles y proporcionando una interfaz sencilla y coherente para ejecutar aplicaciones, ocultando la complejidad del hardware subyacente.
- Proporcionan seguridad y protección a las aplicaciones, ofreciendo mecanismos para proteger los recursos y datos del sistema de accesos no autorizados y software malicioso.
- Proporcionan utilidades y servicios que facilitan a los usuarios la interacción con el sistema y la ejecución de aplicaciones.

Referente al primer punto, Iglesias nos indica cuán importante es la gestión de procesos, el cual es el tema central de la materia hacia la cual va destinado este documento hasta la fecha de publicación del mismo.

Sobre esto, Iglesias dice: “La gestión de procesos es una parte fundamental de los sistemas operativos. El objetivo de la gestión de procesos es crear, ejecutar y coordinar procesos. Un proceso tiene su propio espacio de direcciones, es decir, su propia área de memoria para su código y datos, así como otros recursos (por ejemplo, archivos, dispositivos a los que se accede, etc.). Cada proceso se identifica por un identificador de proceso único y, como se ha mencionado anteriormente, puede tener uno o más hilos, es decir, un proceso puede describirse como un contenedor para uno o más hilos. Un hilo es una unidad ligera de ejecución dentro de un proceso. Pueden existir varios hilos dentro del mismo proceso y compartir el mismo espacio de memoria y recursos del sistema, por lo que los hilos de un proceso pueden comunicarse y compartir datos directamente accediendo a la memoria compartida, así como utilizando primitivas de sincronización como mutexes o semáforos, que se presentarán más adelante.”

Para visualizar este comportamiento, se dotó a los estudiantes de la materia distintos ejercicios los cuales tienen como objetivo probar la comprensión y manejo de estos conceptos de una forma didáctica y comprensiva. En el caso del presente trabajo, se realizó un par de ejercicios los cuales lidian con semáforos.

Un semáforo, según Iglesias, es “una variable entera que se utiliza para gestionar el acceso a un determinado recurso o para controlar el flujo de ejecución de hilos o procesos. Existen dos tipos de semáforos: semáforos binarios y semáforos de conteo”.

Además, se decidió implementar la solución a los ejercicios utilizando el lenguaje de programación C++.

C++ es un lenguaje de programación que viene como sucesor del lenguaje de programación C; C siendo un lenguaje de programación procedural que le permite al programador manejar el hardware y gestionar la memoria de forma manual.

Lo que brinda C++ a la mesa es la capacidad de realizar programación orientada a objetos, la cual es de vital importancia a la hora de realizar trabajos colaborativos que requieran el manejo de conceptos algo complejos.

Marco teórico

Computadora: “Máquina electrónica capaz de realizar un tratamiento automático de la información y de resolver con gran rapidez problemas matemáticos y lógicos mediante programas informáticos” (RAE)

Sistema operativo: “Conjunto de programas de propósito específico que se ejecutan en un sistema informático y que le permiten funcionar correctamente.” (Iglesias)

Proceso: “Instancia de un programa informático que está siendo ejecutada por el hardware del ordenador” (Iglesias)

Hilo: “Unidad ligera de ejecución dentro de un proceso” (Iglesias)

Mutex: “Primitiva de sincronización que permite que sólo un hilo o proceso acceda a la vez a un recurso compartido, como por ejemplo la memoria o un fichero. Se utiliza para prevenir condiciones de carrera, donde múltiples hilos o procesos pueden intentar acceder y modificar el recurso compartido simultáneamente, conduciendo a resultados impredecibles o incorrectos.” (Iglesias)

C: “Lenguaje de programación procedimental conocido por su simplicidad y capacidades de bajo nivel. Adecuado para programación de sistemas y sistemas embebidos debido a su acceso directo al hardware y manejo de memoria manual.” (Kholmatov & Mubiyna)

C++: “Lenguaje de programación multiparadigma que se construye en base a la fundación de C. Soporta tanto programación procedimental como orientada a objetos. C++ introduce clases y objetos, habilitando la encapsulación, herencia y polimorfismo.“ (Kholmatov & Mubiyna)

Semáforo: “Variable entera que se utiliza para gestionar el acceso a un determinado recurso o para controlar el flujo de ejecución de hilos o procesos.” (Iglesias)

Semáforo binario: “Tipo de semáforo utilizado para implementar mutex. Tienen dos estados: bloqueado y desbloqueado, permitiendo que sólo un hilo o proceso acceda a un recurso compartido a la vez. Cuando un hilo o proceso quiere acceder al recurso, intenta adquirir el semáforo binario. Si el semáforo está bloqueado, el hilo o proceso se bloquea hasta que el semáforo se desbloquee.” (Iglesias)

Clase: “Plantilla para el objetivo de la creación de objetos de datos según un modelo predefinido.” (Wikipedia)

Planteamiento del Problema

10. En la fábrica de bicicletas MountanenBike, tenemos tres operarios que denominaremos OP1, OP2 y OP3. OP1 monta ruedas, OP2 monta el cuadro de las bicicletas, y OP3, el manillar. Un cuarto operario, el Montador, se encarga de tomar dos ruedas, un cuadro y un manillar, y arma la bicicleta. Sincronizar las acciones de los tres operarios y el Montador utilizando semáforos, en los siguientes casos:

a) Los operarios no tienen ningún espacio para almacenar los componentes producidos pero sí tienen espacio para fabricar de a uno, y el Montador no podrá tomar ninguna pieza si esta no ha sido fabricada previamente por el correspondiente operario.

b) Los operarios no tienen ningún espacio para almacenar los componentes producidos pero sí tienen espacio para fabricar de a uno, y en el caso de OP1 tiene espacio para fabricar las dos ruedas y el Montador no podrá tomar ninguna pieza si esta no ha sido fabricada previamente por el correspondiente operario.Restricción: el OP1 sólo produce de a una rueda por vez.

11. El problema de los fumadores de cigarrillos (Patil 1971). Considere un sistema con tres procesos fumadores y un proceso agente. Cada fumador está continuamente armando y fumando cigarrillos. Sin embargo, para armar un cigarrillo, el fumador necesita tres ingredientes: tabaco, papel y fósforos. Uno de los procesos fumadores tiene papel, otro tiene el tabaco y el tercero los fósforos. El agente tiene una cantidad infinita de los tres materiales. El agente coloca dos de los ingredientes sobre la mesa. El fumador que tiene el ingrediente restante arma un cigarrillo y se lo fuma, avisando al agente cuando termina. Entonces, el agente coloca dos de los tres ingredientes y se repite el ciclo. Escriba un programa para sincronizar al agente y los fumadores utilizando semáforos.

Herramientas para la Solución

Librerías Usadas

`<thread>`

La librería **thread** introduce la capacidad de trabajar con hilos (threads) en C++. Permite crear y gestionar hilos de ejecución, facilitando el desarrollo de aplicaciones concurrentes y paralelas.

`std::thread:` Representa un hilo de ejecución. Se puede utilizar para crear un nuevo hilo y asociarlo con una función o método.

`std::this_thread::sleep_for:` Hace que el hilo actual duerma durante un período de tiempo específico.

`std::this_thread::yield:` Cede el control del hilo actual, permitiendo que otros hilos se ejecuten.

`<semaphore>`

La librería **semaphore** proporciona la funcionalidad de semáforos, que es otro mecanismo de sincronización. Los semáforos controlan el acceso a un recurso compartido permitiendo que un número determinado de hilos accedan al recurso al mismo tiempo, a diferencia de los mutexes, que solo permiten uno.

`std::mutex:` El mutex estándar básico que permite bloquear y desbloquear secciones de código.

`std::counting_semaphore:` Un semáforo que permite que más de un hilo acceda a un recurso a la vez, hasta un número máximo predefinido.

`std::binary_semaphore:` Un semáforo binario, que es similar a un mutex, pero con más flexibilidad, permitiendo adquirir y liberar acceso más de una vez.

`<mutex>`

La librería **mutex** proporciona la funcionalidad de bloqueo de exclusión mutua (mutual exclusion), que es clave en la programación concurrente. Los mutexes se utilizan para proteger secciones críticas del código, asegurando que solo un hilo pueda acceder a ellas en un momento dado, evitando condiciones de carrera.

`std::mutex:` El mutex estándar básico que permite bloquear y desbloquear secciones de código.

`std::atomic:` Atomic se encarga de operaciones que no pueden ser divididas o también no pueden detener por otros hilos esto mejora el orden y la interacción entre los hilos.

Entornos de Desarrollo

IDE

Se utilizará Clion como IDE principal para el desarrollo del proyecto, incorporando el estándar C++ 20 debido a que cuenta con la clase semáforo dentro.

Sistema de Control de Versiones

El repositorio del proyecto estará alojado en GitHub donde se utilizará Git como sistema de control de versiones a modo de una oportuna colaboración efectiva.

Construcción y Automatización

Se utilizará CMake para gestionar la construcción del proyecto. Las configuraciones sobre la construcción y la automatización para el proyecto se encuentran en el archivo CMakeLists.txt.

Explicación Detallada

A continuación se hará una explicación detallada del código en base a la estructura usada para la realización del proyecto.

La estructura usada fue la siguiente:

- src - Aquí se encontrarán todas las implementaciones de los archivos de cabecera para las clases junto al archivo principal del programa o main.
- include - Aquí se encontrarán todas las definiciones de las clases a usar, los archivos de cabecera.
- docs - Aquí se encontrarán documentos varios que puedan ser de utilidad para el proyecto, como ser explicaciones de código, el presente documento, etc.

main.cpp: En este archivo se encuentra la función principal que ejecuta el programa, la cual presenta un menú interactivo con distintas opciones para resolver los problemas: el inciso A y el inciso B del problema 10, así como el problema 11, detallados anteriormente en el documento. La estructura del archivo es la siguiente:

- Primero, se declara una variable de tipo string llamada answer, que se encargará de recibir la entrada del usuario para determinar qué problema desea resolver y cuándo desea finalizar la ejecución del programa.
- La entrada será válida dentro de un bucle while donde se tendrá las siguientes opciones:
    - 1: Para resolver el inciso A del problema 10
    - 2: Para resolver el inciso B del problema 10
    - 3: Para resolver el problema 11
    - e: Para terminar la ejecución del programa

Así mismo, se han implementado diversas validaciones para asegurar un uso correcto del programa y evitar problemas al interactuar con el menú. Algunas de estas validaciones son las siguientes:

- No se podrá colocar espacios vacíos o espacios en blanco
- No se podrá colocar más de un carácter en el input. Ej:
    - Válido: ‘e’
    - Inválido: ‘es’
- Se asegura que si el usuario teclea ‘E’ de forma que esté en mayúscula no se interrumpa el programa, ya que el carácter será transformado a minúscula.

Se proporciona retroalimentación mediante mensajes de error para entradas no válidas, mejorando la experiencia del usuario. Las únicas opciones válidas son: ‘1’, ‘2’, ‘3’ y ‘e’ o ‘E’.

Menu.cpp: Siguiendo con el programa, nos adentramos a las distintas opciones que tenemos, este archivo contiene las implementaciones de las funciones del menú para resolver los

problemas de sincronización que fueron propuestos. Se utilizan distintas clases, como Worker, Mounter, Agent, y Smoker, para modelar las interacciones entre los operarios y el montador en el contexto de la fabricación de bicicletas y el armado de cigarrillos.

- **Problema10ExerciseA:** La función Menu::Problem10ExerciseA() comienza creando instancias de las clases Worker y Mounter, representando a los operarios y al montador, respectivamente. Luego, se crean hilos para ejecutar las tareas de cada operario y del montador de manera concurrente.

    1. **Creación de Hilos:**
        - Se crean dos hilos (t1 y t2) que llaman al método MakeWheel del operario wk1 para producir dos ruedas. Esto refleja la necesidad de dos ruedas para armar una bicicleta.
        - Se crea un hilo (t3) para que el operario wk2 construya el cuadro de la bicicleta mediante el método MakeFrame.
        - Un hilo adicional (t4) se encarga del operario wk3, que fabrica el manillar utilizando el método MakeHandlebar.
        - Finalmente, se crea un hilo (t5) para el montador (mounter), que ensamblará la bicicleta con las piezas producidas.

    2. **Sincronización**:
        - Después de iniciar todos los hilos, el código utiliza join() para esperar que cada uno de ellos complete su ejecución. Esto asegura que el programa no continúe hasta que todas las piezas necesarias estén listas, garantizando la sincronización adecuada entre la producción de los componentes y el montaje final.

- **Problema10ExerciseB:** La función Menu::Problem10ExerciseB() comienza creando instancias de las clases Worker y Mounter, representando a los operarios y al montador, respectivamente. Al igual que en el inciso A, se crean hilos para ejecutar las tareas de cada operario y del montador de manera concurrente.

    1. **Creación de Hilos:**
        - Se crean dos hilos (t1 y t2) que invocan el método MakeWheel del operario wk1 para producir dos ruedas. Esto es fundamental para que el montador pueda armar la bicicleta, dado que se requieren dos ruedas.
        - Se crea un hilo (t3) para el operario wk2, que se encarga de fabricar el cuadro de la bicicleta mediante el método MakeFrame.
        - Un hilo adicional (t4) se encarga del operario wk3, quien produce el manillar utilizando el método MakeHandlebar.
        - Finalmente, se crea un hilo (t5) para el montador (mounter), que ahora invoca el método MakeBicycleTwoWheels, adaptado para ensamblar la bicicleta con las dos ruedas producidas por el operario.

2. **Sincronización:**
    - Al igual que en el inciso A, se utilizan las llamadas a join() para esperar que todos los hilos finalicen su ejecución. Esto garantiza que el programa no progrese hasta que todas las piezas necesarias estén listas, asegurando la sincronización adecuada entre la producción de los componentes y el ensamblaje final.

Worker.cpp: El archivo worker.cpp contiene la implementación de la clase Worker, que modela el comportamiento de los operarios en la fabricación de bicicletas. Cada operario se encarga de producir componentes específicos: ruedas, cuadros y manillares. Se utilizan hilos y semáforos para manejar la concurrencia y la sincronización entre los diferentes operarios y el montador.

- **Problema 11:** La función Menu::Problem11() comienza creando instancias de la clase Agent y de tres fumadores (smk1, smk2, smk3), cada uno representando a un fumador con un ingrediente diferente: tabaco, papel y fósforos.

    1. **Creación de Hilos:**
        - Se crea un hilo (agent_thread) que ejecuta el método PutTwoIngredients del agente (ag). Este hilo es responsable de colocar dos de los tres ingredientes sobre la mesa para que los fumadores puedan usarlos.
        - Se crean tres hilos adicionales (tobacco_thread, paper_thread y matches_thread) para cada uno de los métodos PutTobacco, PutPaper y PutMatches, que permiten a los fumadores poner su ingrediente en la mesa. Estos hilos están asociados con los respectivos fumadores (smk1, smk2, smk3).
    2. **Sincronización y Control de Ejecución:**
        - Después de iniciar todos los hilos, se utiliza join() en el hilo del agente para esperar a que complete su tarea de colocar los ingredientes.
        - Luego, se verifica la variable continue_smoking, que determina si los fumadores deben seguir trabajando. Si es falsa, se separan los hilos de los fumadores mediante detach(), permitiendo que se ejecuten independientemente, ya que no se necesitan más interacciones. Esto implica que los fumadores no podrán seguir armando cigarrillos.
        - Si continue_smoking es verdadera, se utiliza join() en los hilos de los fumadores para esperar a que terminen sus tareas, asegurando que el programa no continúe hasta que todos los ingredientes hayan sido colocados y utilizados.

Agent.cpp: El archivo agent.cpp contiene la implementación de la clase Agent, que modela el comportamiento de los agentes, el cual se encarga de la creación de cigarros. El agente se

encargará de proveer dos materiales de los tres mencionados tabaco, papel o fósforos, estos van a ser propuestos de forma aleatoria en la creación de cada nuevo proceso en los hilos para formar el cigarro estos tendrán un máximo de 5 iteraciones, en el mismo se encuentra la función para volver a repetir el proceso mediante una interrogante “Desea continuar?”.

Smoker.cpp: El archivo smoker.cpp contiene la implementación de la clase Smoker, que modela el comportamiento de los fumadores, los cuales se encarga fumar y adicionar el recursos faltante. El smoker se encargará de proveer el último recurso faltante y luego armara el cigarro y lo fumara éste será identificado como un nuevo proceso.

Funcionalidades Problema 10 A

```
void Worker::MakeWheel() {
    std::this_thread::sleep_for( rtime: std::chrono::milliseconds ( rep: 1000));
    {
        std::lock_guard<std::mutex> lock( [&] print_mutex);
        std::cout<<"OP 1: Rueda montada.\n";
    }
    wheel_semaphore.release();
}
```

Esta función simula la fabricación de una rueda. Primero, utiliza std::this_thread::sleep_for para simular un tiempo de producción de 1 segundo.

A continuación, un lock_guard se utiliza para bloquear el print_mutex, asegurando que el mensaje sobre la rueda montada se imprima sin interferencias de otros hilos.

Finalmente, se libera el semáforo wheel_semaphore, indicando que la rueda ha sido fabricada y está disponible para el montador.

```
void Worker::MakeFrame() {
    std::this_thread::sleep_for( rtime: std::chrono::milliseconds ( rep: 2000));
    {
        std::lock_guard<std::mutex> lock( [&] print_mutex);
        std::cout<<"OP 2: Cuadro montado.\n";
    }
    frame_semaphore.release();
}
```

Esta función simula la fabricación de un cuadro. Primero, utiliza std::this_thread::sleep_for para simular un tiempo de producción de 2 segundos.

A continuación, un lock_guard se utiliza para bloquear el print_mutex, asegurando que el mensaje sobre la rueda montada se imprima sin interferencias de otros hilos.

Finalmente, se libera el semáforo frame_semaphore, indicando que el cuadro ha sido fabricado.

```
void Worker::MakeHandlebar() {
    std::this_thread::sleep_for( rtime: std::chrono::milliseconds ( rep: 1500));
    {
        std::lock_guard<std::mutex> lock( [&] print_mutex);
        std::cout<<"OP 3: Manillar montado.\n";
    }
    handlebar_semaphore.release();
}
```

Esta función se encarga de fabricar un manillar, con un tiempo de producción simulado de 1.5 segundos.

Al igual que en las funciones anteriores, se utiliza un lock_guard para controlar el acceso a la consola y evitar impresiones entremezcladas.

Al finalizar, se libera el semáforo handlebar_semaphore, indicando que el manillar está disponible.

Mounter.cpp: contiene la implementación de la clase Mounter, que se encarga del ensamblaje de las bicicletas. Esta clase gestiona la recolección de los componentes producidos por los operarios, como las ruedas, el cuadro y el manillar, utilizando semáforos para asegurar que cada componente esté disponible antes de proceder con el ensamblaje.

```
void Mounter::CollectWheels() {
        wheel_semaphore.acquire();
        wheel_semaphore.acquire();
        {
            std::lock_guard<std::mutex> lock( [&] print_mutex);
            std::cout << "Montador: Ambas ruedas recolectadas.\n";
        }
}
```

Esta función se encarga de recolectar las dos ruedas necesarias para el ensamblaje de la bicicleta.

Utilizamos el semáforo wheel_semaphore para adquirir ambas ruedas.

Se emplea un lock_guard para proteger la impresión en consola, asegurando que el

mensaje sobre la recolección de las ruedas no se interrumpa.

Al finalizar, imprime un mensaje indicando que ambas ruedas han sido recolectadas.

```
void Mounter::MakeBicycle() {

    wheel_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Primera rueda recolectada.\n";
    }
    wheel_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Segunda rueda recolectada.\n";
    }
    frame_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Cuadro recolectado.\n";
    }
    handlebar_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Manillar recolectado.\n";
    }

    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Todas las piezas listas, bicicleta ensamblada.\n";
    }
}
```

Esta función se encarga de ensamblar la bicicleta.

Primero, adquiere cada componente necesario utilizando los semáforos correspondientes para las ruedas, el cuadro y el manillar.

Para cada adquisición, se utiliza un lock_guard para proteger las impresiones en consola, asegurando que los mensajes de recolección sean claros y no se mezclen.

Una vez que todos los componentes han sido recolectados, se imprime un mensaje final indicando que la bicicleta ha sido ensamblada.

Funcionalidades Problema 10 B

```
void Mounter::MakeBicycleTwoWheels() {
    CollectWheels();

    frame_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Cuadro recolectado.\n";
    }
    handlebar_semaphore.acquire();
    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Manillar recolectado.\n";
    }

    {
        std::lock_guard<std::mutex> lock([&]print_mutex);
        std::cout<<"Montador: Todas las piezas listas, bicicleta ensamblada.\n";
    }
}
```

Esta función es similar a MakeBicycle(), pero primero llama a CollectWheels() para asegurarse de que ambas ruedas estén listas antes de continuar.

Después de recolectar las ruedas, adquiere el cuadro y el manillar utilizando los semáforos correspondientes, protegiendo las impresiones en consola con lock_guard.

Al final, imprime un mensaje indicando que todas las piezas están listas y que la bicicleta ha sido ensamblada.

Funcionalidades Problema 11

```
void Agent::PutTwoIngredients () {
    const int MAX_ITERATIONS = 5;
    int iteration_count = 0;
    while (true) {
        continue_smoking= true;
        agent_ready.acquire();  // Espera a que el fumador termine de fumar

        int choice = rand() % 3;

        if (choice == 0) {
            std::cout << "\nAgente coloca papel y fosforos en la mesa." << std::endl;
            smoker_with_tobacco.release();
        } else if (choice == 1) {
            std::cout << "\nAgente coloca tabaco y fosforos en la mesa." << std::endl;
            smoker_with_paper.release();
        } else {
            std::cout << "\nAgente coloca tabaco y papel en la mesa." << std::endl;
            smoker_with_matches.release();
        }

        iteration_count++;

        if(iteration_count % MAX_ITERATIONS == 0 && !AskToContinue()) {
            std::cout << "\n saliendo del ciclo y volviendo al menú..." << std::endl;
            continue_smoking = false;
            break;
        }
    }
```

Esta función se encarga de colocar dos de los materiales necesarios en la mesa.

Primero se declaran dos variables una que estable el limite de iteraciones y otro que funcionará como contador de iteraciones.

Entrado en el bucle, se estable por defecto que se quiere continuar armando cigarrillos para posteriormente asegurar con acquire que antes se ha terminado de fumar.

El número aleatorio generado representará los dos primeros materiales que colocará el mismo incrementando el contador de iteraciones.

Además de que cuando coloca los dos materiales, lo libera con release permitiendo a fumador entrar en acción.

```cpp
bool Agent::AskToContinue() {
    std::string choice;

    while (true) {
        std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 1));
        std::cout << "\nDesea continuar? (y/n): ";
        std::getline( &: std::cin,  &: choice);

        if (choice.empty() || choice == " ") {
            std::cerr << "Se introdujo un espacio en blanco o carácter vacío, introduzca una opción.\n";
            continue;
        }

        switch (tolower( C: choice[0])) {
            case 'y':
                return true;
            case 'n':
                return false;
            default:
                std::cerr << "Entrada no válida. Por favor, ingrese 'y' o 'n'." << std::endl;
        }
    }
}
```

Esta función pregunta si desea continuar haciendo cigarrillos.

sleep for y chrono permiten simular el tiempo transcurrido en este caso 1 segundo.

Se valida que la entrada se correcta y si se elige 'y' se continuará creando mas cigarrillos, caso contrario que si se elige 'n' entra a la condición de la anterior función por tanto se rompe el bucle.

```cpp
void Smoker::PutTobacco() {
    while (true) {
        smoker_with_tobacco.acquire();
        std::cout << "Fumador con tabaco arma y fuma el cigarrillo." << std::endl;
        std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 1));
        agent_ready.release();
    }
}
```

Esta función se encarga de complementar con tabaco caso de que el agente haya colocado sobre la mesa papel y fosforos.

Se asegura que el agente haya liberado su acción, luego se simula un tiempo de armado con sleep_fpr y chrono y finalmente se libera la acción.

```
void Smoker::PutPaper() {
    while (true) {
        smoker_with_paper.acquire();
        std::cout << "Fumador con papel arma y fuma el cigarrillo." << std::endl;
        std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 1));
        agent_ready.release();
    }
}
```

Esta función se encarga de complementar con papel caso de que el agente haya colocado sobre la mesa tabaco y fosforos.

Se asegura que el agente haya liberado su acción, luego se simula un tiempo de armado con sleep_for y chrono y finalmente se libera la acción.

```
void Smoker::PutMatches() {
    while (true) {
        smoker_with_matches.acquire();
        std::cout << "Fumador con fosforos arma y fuma el cigarrillo." << std::endl;
        std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 1));
        agent_ready.release();
    }
}
```

Esta función se encarga de complementar con fosforos caso de que el agente haya colocado sobre la mesa tabaco y papel.

Se asegura que el agente haya liberado su acción, luego se simula un tiempo de armado con sleep_for y chrono y finalmente se libera la acción.

Resultados

```
===== Problemas con semáforos en C++ =====
Elija una opción:
1. Problema 10 inciso A
2. Problema 10 inciso B
3. Problema 11
e. Salir
e
Hasta luego.
```

Resultado Problema 10 A

```
OP 1: Rueda montada.
OP 1: Rueda montada.
Montador: Primera rueda recolectada.
Montador: Segunda rueda recolectada.
OP 3: Manillar montado.
OP 2: Cuadro montado.
Montador: Cuadro recolectado.
Montador: Manillar recolectado.
Montador: Todas las piezas listas, bicicleta ensamblada.
```

Resultado Problema 10 B

```
OP 1: Rueda montada.
OP 1: Rueda montada.
Montador: Ambas ruedas recolectadas.
OP 3: Manillar montado.
OP 2: Cuadro montado.
Montador: Cuadro recolectado.
Montador: Manillar recolectado.
Montador: Todas las piezas listas, bicicleta ensamblada.
```

Resultado Problema 11

```
Agente coloca tabaco y papel en la mesa.
Fumador con fosforos arma y fuma el cigarrillo.

Agente coloca tabaco y papel en la mesa.
Fumador con fosforos arma y fuma el cigarrillo.

Agente coloca tabaco y fosforos en la mesa.
Fumador con papel arma y fuma el cigarrillo.

Agente coloca tabaco y fosforos en la mesa.
Fumador con papel arma y fuma el cigarrillo.

Agente coloca tabaco y papel en la mesa.
Fumador con fosforos arma y fuma el cigarrillo.

Desea continuar? (y/n):
```

```
Desea continuar? (y/n):n

 saliendo del ciclo y volviendo al men...
```

Referencias

Iglesias, M. J. F (2023). Conceptos básicos de sistemas operativos.

Kholmatov, A., & Mubiyna, A. (2023). C AND C++ PROGRAMMING LANGUAGES CAPABILITIES AND DIFFERENCES. Galaxy International Interdisciplinary Research Journal, 11(11), 35-40.

Clase (informática). (2023). Wikipedia, La enciclopedia libre.