Tema 2.1: Introducción a la gestión de Procesos.

- **Proceso:** Programa en ejecución, cada ejecución da lugar un proceso nuevo. Cada proceso tiene un identificador único llamado pid(process id), que nos permite obtener mas información sobre el proceso, como el contexto, directorio, padre, etc.
  - Esta formado por: Código del programa(instrucciones) y los datos asociados a la propia ejecución del programa.
  - **Partes**: Pila(se va ampliando hacia abajo), memoria dinámica, datos(se amplia hacia arriba) y texto(el código). Cada proceso tiene esta estructura y son independientes entre ellos.
- **Ciclo de vida de un proceso:**
  - **Listo**: Cuando se ejecuta un programa se hace el exec, lo primero que hace es comprobar sus propiedades y si se tiene recursos para ejecutar ese proceso nuevo. Si se tienen recursos se creará ese proceso nuevo y se le da un pid propio, el pid debe estar por debajo del limite máximo de procesos o no se podrá crear. No se reutilizan pid's, habría que reiniciar.
  - **En ejecución :** Cuando esta listo, se pasa a memoria y se ejecuta, si es para algo lento lo pasaremos a bloqueado, para que espere a recibir los datos que necesite, y pasara a listo, si el proceso ha superado su tiempo asignado o si cede su CPU. Este ultimo lo controla el planificador. El proceso puede terminar con exit, que liberará los recursos o si no, esperará un tiempo antes de liberarlo.
  - **Bloqueado**: Cuando el proceso esta esperando ha recibir datos, si lo recibe pasa a listo, pero si no recibe los datos en ningún momento se quedará en bloqueado.
- **Modelo de colas simplificado: Un procesador**.
  - Los procesos están en una cola, y según las unidades de procesado que necesita va a una parte u otra, hay una para procesos rápidos y otras para procesos mas pesados. Si un proceso en su franja de tiempo no ha terminado, se vuelve a poner en la cola. El que elige donde va cada proceso es el dispatcher. (Función como la cola de un supermercado)
- **Información de un proceso**: Toda la información que permite la correcta ejecución del proceso.
  - **Tres categorías**:
    - Información almacenada en el procesador.
    - Información almacenada en memoria.
    - Información adicional gestionada por el sistema operativo.
  - El pid se almacena en la tabla de procesos (TP). Cada proceso tiene su BCP, donde se almacena su entorno(no cambia), mapa de memoria(texto y datos) y el contexto/ estado(puede cambiar).
  - En el estado/contexto se incluye los valores de los registros accesibles y no accesibles para el usuario del procesador. Cuando se cambia de contexto, se salvaguarda el estado del procesador del saliente y se restaura el estado del proceso del entrante.
  - **Memoria virtual**: Se engaña al sistema, ya que se necesitan recursos que no tiene la maquina y puede que tenga que matar a otro proceso para ir haciendo hueco. Mem. dinam.
  - **Modelo de imagen de memoria con Regiones multiples:**
    - El numero de regiones es fijo, y cada region es de tamaño variables.
    - Las regiones son: Pila, Datos y Texto.
    - Con memoria virtual el hueco entre pila y datos no consume recursos físicos.
  - Pero hay modelos de imagen de memoria con una sola region y también con regiones no fijas, que son opciones mas avanzadas como Windows.
  - **El sistema operativo** mantiene información adicional sobre los procesos, en la Tabla de Procesos. Cada una de las entradas de la TP es un BCP (Bloque de Control de Procesos), que mantiene casi toda la información sobre el proceso.
    - **Contenido del BCP:**
      - Información de identificación:
        - Estado del proceso.
        - Evento por el que espera, si es que está bloqueado.
        - Prioridad del proceso.

        - Información de planificación.
      - Estado del procesador:
        - Archivos abiertos.
        - Puertos de comunicación usado
        - Temporizadores
      - Información de control del proceso:
        - Punteros para estructurar los procesos en cola.
    - **Contenido fuera del BCP**:
      - El BCP no almacena toda la información del un proceso, se decide que no almacenar según la eficiencia, si se puede almacenar bien o no en la tabla, y si hay que compartir información , si se comparte algún dato no estará en el BCP.
      - Se usan punteros, que se almacenan en tablas de paginas, y desde el BCP se puede acceder a esa tabla. Se usa este método porque tienen tamaño variables y el compartir datos debe ser externo.
    - La tabla de punteros de posición de los ficheros, que almacena los ficheros abierto por un proceso, se almacena fuera del BCP y si se almacena dentro no podrán ser compartidos.
  - **El servicio fork,** hace que el proceso padre e hijo compartan el contexto, y solo se hará una copia cuando el hijo haga una modificación, mientras comparte el espacio con el padre.
  - **El servicio exec,** lee un ejecutable, y cambia el mapa de memoria y el contexto, al proceso que lo está ejecutando.
  - **El servicio exit,** finaliza la ejecución del proceso, cierra los descriptores, liberan todos lo recursos y libera el BCP. Los 3 están explicados en el Tema 1.
- **Tipos de sistemas operativos:**
  - **Multiproceso**: Utiliza el paralelismo, varios procesos funcionan a la vez.
    - Multiusuario y Monousuario.
  - **Monoproceso**: Concurrencia, funciona un proceso a la vez, pero se van compaginando y da la sensación de paralelismo
    - Monousuario.
- **Multitarea**: Alternancia en los procesos de fases de E/S y de procesamiento, la memoria almacena varios procesos.
  - **Ventajas**:
    - Facilita la programación, se divide un programa en procesos. Modularidad.
    - Permite el servicio interactivo simultáneo de varios usuarios.
    - Aprovecha los tiempos de espera de E/S de los procesos.
    - Aumenta el uso de CPU.
  - **Grado de multiprogramación:** Numero de procesos activos, cuantos más procesos el rendimiento de la maquina disminuye. Los sistemas con memoria virtual dividen el espacio direccionable de los procesos en paginas y el de memoria física en marcos de pagina. El numero de paginas en memoria principal es el Conjunto Residente.
    - Cuando aumenta el grado de multiprogramacion, disminuye el tamaño del conjunto residente de cada proceso, para que quepan todos.
      - **Con poca memoria física:** Se produce hiperpaginacion cuando alcanzo alto porcentaje de uso, para soluciónalo hay que ampliar la memoria principal.
      - **Con mucha memoria física**: Se alcanza alto porcentaje de utilización de CPU con menos procesos de los que caben en memoria, para solucionarlo se puede mejorar el procesador o añadir más procesadores.
- **Cambio de contexto**: Cuando se pasa de ejecutar un proceso a otro. Se produce cuando el sistema operativo asigna el procesador a un nuevo proceso.
  - **Tipos de cambios de contexto:**
    - **Cambio de contexto voluntario(CCV):** El proceso realiza una llamada al sistema o produce un excepción que implica esperar por un evento, así deja funcionar a otro.
    - **Cambio de contexto involuntario(CCI)**: El SO quita de la CPU al proceso, puede darse porque el proceso haya agotado su rodaja de ejecución.
  - **Acciones**:

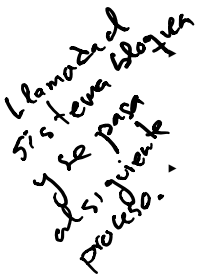
        - Guarda el estado del procesador en el BCP del proceso en ejecución.
        - Restaurar el estado del nuevo proceso en el procesador.
    - Generación de ejecutables: MIRAR LIBRO Y DIAPOSITIVA.

Tema 2.2: Planificación de procesos.

- **Creación de procesos en UNIX:** Se distingue entre crear procesos y ejecutar nuevos programas.
    - Para crear un proceso se hace mediante fork(), que no cambia la imagen de memoria y mantiene los valores del padre, crea una copia, y los cambios en el hijo no le afectan. Copia el BCP del padre y su Mapa de memoria (incluyendo pilas).
        - Un inconveniente es que muchos datos podrían compartirse.
    - Para ejecutar un programa se hace mediante exec(), se hace sobre procesos hijo, sustituye su imagen en memoria por la del programa. Mientras el padre puede esperar al hijo con wait o seguir creando hijos.
        - Un inconveniente es que fork ha copiado todo lo del padre, para que luego lo deshecha el exec.
    - Para solucionar ambos inconvenientes se usa COW(Copy-on-Write), los datos se marcan de manera que si se intentan modificar se realiza una copia para cada proceso.
- **Terminación de procesos:** Todos los recursos asignados son liberados, excepto el valor del exit que se mantiene en el BCP esperando el wait del padre y si el padre ha muerto (el hijo se queda zombie) lo hereda el init y este hace el wait. Entonces se elimina el BCP, tras el wait. Puede terminar de 2 manera:
    - **Voluntariamente**: Llamada al sistema exit().
    - **Involuntariamente**: Excepciones, abortado por el usuario(ctrl+c) u otro proceso(kill).
- **Expulsión a disco(swap):** Cuando hay demasiados procesos y baja el rendimiento, se pasan algunos procesos a disco con swap que libera memoria, y cuando se quieran seguir ejecutando se pasa de nuevo a memoria. Los nuevos estados son: Bloqueado y suspendido, y Listo y suspendido.
- **Niveles de planificación:**
    - Planificación a corto plazo: Selecciona el siguiente proceso a ejecutar.
    - Planificación a medio plazo: Elegir que proceso se retira o añade a memoria principal.
    - Planificación a largo plazo: Realiza el control de admisión de procesos a ejecutar.
- **Tipos de planificación:**
    - **No apropiativo:** El proceso conserva el uso de CPU, no se puede expulsar hasta que termine.
    - **Apropiativo:** El sistema puede expulsar a un proceso de CPU, se puede expulsar aunque no haya terminado para ejecutar otro proceso.
- **Momentos de decidir la planificación:**
    - Cuando un proceso se bloquea, mientras espera el evento se pasa a ejecutar otro proceso.
    - Cuando se produce una interrupción, como la del reloj o de E/S.
    - Cuando termina un proceso.
- **Cola de procesos:** Los procesos listos para ejecutar se mantiene en una cola, que puede ser única, por tipo de proceso o por prioridad. Cuando se saca un proceso que no ha terminado se pone al final de la cola. Si entra a ejecutar uno nuevo se pone antes que el que esta terminando.
- **Algoritmos de planificación:**
    - Se pueden orientar a:
        - **Utilización de CPU:** maximizar el tiempo de uso.
        - **Productividad:** maximizar el numero de procesos que terminan por unidad de tiempo.
        - **Tiempo de retorno(Tq=Tf-Ti):** Minimizar el tiempo que un proceso pasa desde que entra hasta que termina.
        - **Tiempo de servicio(Ts):** Tiempo que se ha dedicado a tareas productivas.
        - **Tiempo de espera(Te):** Tiempo que un proceso ha estado en la cola de espera.
        - **Tiempo de retorno normalizado(Tn=Tq/Ts):** Tanto por uno que un proceso ha estado en espera.
    - **Asignación First to Come First to Serve (FCFS):** Es FIFO, primero en llegar primero en

El cambio por bloque afecta a todos los tipos.

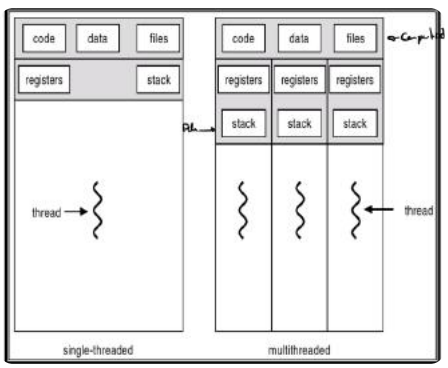
salir. Es no apropiativo, no se puede expulsar un proceso. Penaliza a los cortos, que tienen que esperar a que terminen todos lo previos.



- **Asignación Shortest Job First:** Primero los procesos mas cortos, menor tiempo de servicio. Es no apropiativo, no se puede expulsar un proceso. Se debe conocer la duración de antemano. El problema es que si solo llegan procesos cortos lo largos no se ejecutan.
- **Cíclico o Round-Robin:** Mantiene una cola FIFO con los procesos listos para ejecutar. A cada proceso se le asigna una rodaja de tiempo de procesador, cuando la agota pasa a la cola y se ejecuta el siguiente en la cola. Se ejecutan un poco todos, hasta que terminan. Un proceso puede volver a lista por que acabe su tiempo o pase a bloqueado, esperando un evento. Es apropiativo, por eso se puede expulsar un proceso para poner otro a ejecutar.
  - Hay que intentar que tampoco haya demasiados cambios de contexto, ya que estos añaden tiempo.
- **Asignación por prioridades:** Cada proceso tiene una prioridad asignada, y se seleccionan primero los procesos con mayor prioridad.

Tema 2.4: Hilos y Procesos.

- Hilos de ejecución de un proceso de forma concurrente. Los threads de un proceso se guardan en su BCP en la tabla de threads, Threads Table, donde se almacena su información y contenido. Se considera la unidad básica de utilización de la CPU.
- **Crear hilos permite que:**
  - Compartan memoria entre ellos, pero cada uno necesita una pila para almacenar sus datos y un minicontexto propio.
  - Aprovechen mejor la rodaja de tiempo que tienen asignada, al hacer menos cambios de contexto, que los que implica crear y terminar un proceso hijo.
  - Un thread esta ligado a un proceso, no es independiente por lo que si muere el proceso mueren los hilos, no como pasa con los procesos hijos.



- **Contenido de un hilo:**
  - Identificador del thread.
  - Contador de programa.
  - Conjunto de registros.
  - Pila.
- **Comparten entre hilos:**
  - Mapa de memoria.
  - Ficheros abiertos.
  - Señales, semáforos y temporizadores.
- **Beneficios:**
  - Mayor interactividad al separar las tareas en hilos.
  - Comparten la mayor parte de los recursos.
  - Cuesta menos tiempo crear un hilo que un proceso.
  - En arquitecturas multiproceso permite hacer ejecución en paralelo asignando hilos distintos a distintos procesadores.
- **Soporte de hilos:**
  - **Espacio de usuario, ULT**, User Level Threads.
    - Están implementado en forma de biblioteca de funciones.
    - El kernel no tiene conocimiento sobre los threads, solo sobre el proceso creador.
    - Es más rápido, pero un bloqueo bloquea todo el proceso.
    - Un hilo de usuario se puede pasar al kernel para que este lo gestione, y deja de depender del proceso y aunque muera el proceso no muere el hilo. Cambia su id.
  - **Espacio de núcleo, KLT**, Kernel Level Threads.
    - Son de los que se ocupa el kernel, de crearlos, planificarlo y destruirlos.
    - Un poco mas lentos al hacerlo por medio del kernel y no directamente.
    - Los bloqueos solo bloquean el thread implicado, no a todo.
    - Se pueden ejecutar varios procesos a la vez, pero hay un limite de threads.
- **Modelos de multiples hilos:**
  - **Muchos a uno:** Corresponde mucho hilos de usuario con un único hilo del núcleo. Una

llamada bloqueante, bloquea todos lo hilos.
    - **Uno a uno:** Hace corresponder un hilo del kernel a cada hilo de usuario. La mayoría de implementaciones restringen al numero de hilos que se pueden crear.
    - **Muchos a muchos:** Multiplexa los threads de usuario en un numero determinado de threads en el kernel. El núcleo se complica mucho.
- **Aspectos de diseño:**
    - Cuando se hace fork de un proceso con hilos, duplica el proceso con todos sus hilos.
    - exec no es adecuado, sustituirá la imagen del proceso, que quitara todo incluido los hilos.
    - **Otra version:** Cuando hace fork, duplica el proceso solo con el hilo que hace fork.
    - Esta versión es mas eficiente si va ha hacer un exec.
  - **Cancelación de hilos:** Un hilo notifica a otro de que deben terminar.
    - **Cancelación asíncrona:** Se fuerza la terminación instantánea del hilo.
        - Puede ocasionar problemas con los recursos asignados.
    - **Cancelación diferida:** El hilo comprueba periódicamente si debe terminar. Es preferible.
  - Para aplicaciones que reciben peticiones y las procesan se pueden usar hilos.
    - Se hace **Thread Pool (Conjunto de Hilos)** que consiste en crear con anterioridad lo hilos para evitar el tiempo de creación (retardos) y se dejan en espera, y se establece un limite, para intentar evitar que un avalancha de peticiones agoten los recursos del sistema. Cuando se reciben mas peticiones de las que se pueden tratar se ponen en cola y cuando la cola se ha llegando se hace denegación de servicios, que no la deja entrar mientras no vayan acabando las peticiones.
- **Hilos en pthreads:**
  - **Creación de hilos:**
    - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg)
        - Devuelve un entero que indica como ha ido la ejecución.
        - *thread es el puntero donde se almacenará el ThreadId, el manejador del hilo.
        - *attr Estructura de atributos.
        - *(*func)(void *) función con el código a ejecutar.
        - *arg parámetro del hilo.
    - Estructura pthead_attr_t: Son los atributos asociado a cada hilo.
        - Controlan:
            - Si el hilo es independiente o dependiente, de kernel o de biblioteca.
            - Tamaño de la pila privada.
            - Localización de la pila.
            - Política de planificación, solo posible si es de kernel.
        - Inicializar: pthread_attr_init(pthread_attr_t *attr)
        - Destruir: pthread_attr_destroy(pthread_attr_t *attr)
        - Por defecto el atributo de independencia esta en PTHREAD_CREATE_JOINABLE, lo que quiere decir que espera un join y no se liberan los recursos. La otra opción cuando acaba el sistema operativo libera los recursos. PT..._DETACHED.
        - El resto en diapositiva 28 de Tema 2.3.
    - pthread_t pthread_self(void) Devuelve el identificador del thread.
    - int pthread_join(pthread_t thread, void **value) Es el wait de los hilos.
        - thread manejador del hilo a esperar.
        - **value Valor de terminación del hilo.
    - int pthread_exit(void *value) Finaliza su ejecución y devuelve value que puede ser de cualquier tipo. Aunque no hagan join, no se quedan hilos zombies.
    - **EVITAR usar variables globales, fork y exec.**
- **Planificación de hilos:** Se basa en el modelo de prioridades y no utiliza el modelo de segmentación por segmentos de tiempo. Un thread continuará ejecutándose en la CPU hasta pasar a un estado que no le permita seguir en ejecución. Para alternar entre thread, se debe

asegurar que el thread permita la ejecución de otros threads.
  - PTHREAD_SCOPE_PROCESS Planificación PCS, del proceso del user, de biblioteca.
  - PTHREAD_SCOPE_SYSTEM Planificación SCS, del kernel.

Tema 2.4 Señales

- **Señales**: Son un mecanismo para comunicar eventos a los procesos, para que estos puedan reaccionar. Son excepciones asíncronas, se procesan de inmediato. Posibles reacciones a un respuesta:
    - Ignorar la señal, SIG_IGN.
    - Invocar la rutina de tratamiento por defecto, si el proceso no esperaba ninguna lo general es que sea muerte.
    - Invocar a un rutina de tratamiento propia.
        - Por ejemplo, un proceso padre recibe la señal SGCHLD cuando su proceso hijo termina, esa señal la envía el SO, no el hijo directamente.
  - Son interrupciones al proceso, si esta ejecutando normal la rutina se lleva a cabo en el siguiente ciclo de reloj y si se captura cuando termine continuara por donde estaba. Pero si esta en un bucle se ejecuta en ese momento, ya que no se sabe cuando terminara el bucle.
  - **Tratamiento**: El SO las transmite al proceso, pero el proceso debe estar preparado para recibirlo o en general muere. El proceso debe especificar con sigaction que señal espera y que rutina de tratamiento seguir.
  - **Enviar señal a un proceso**: **int kill(pid_t pid, int sig);** No es para matar el proceso, aunque si el proceso no la espera le matará.
    - Para enviar una señal a sí mismo **int raise(int sig);**
    - CTRL-C SIGINT
    - CTRL-Z SIGSTOP
  - **Esperar a una señal:** Con el servicio **int pause(void)** el proceso espera hasta que le llegue una señal cualquiera que le reanuda, se utilizar para ejecutar en orden.
    - No se puede especificar un plazo de desbloqueo.
    - No se puede indicar el tipo de señal que se espera.
    - No se desbloquea si recibe una señal ignorada.
  - **Sleep(unsigned int sec)** Suspende un proceso hasta que vence un plazo o se recibe una señal.
  - **Especificar acción:** Se utiliza sigaction para especificar la accion a realizar como tratamiento de la señal sig. OJO, cuando se ha completado la rutina rearmar el sigaction.
    - **int sigaction(int sig, struct sigaction *act, struct sigaction *oact);**
        - Preferible ante signal(,)
        - Sig es la señal para la que actúa.
        - En la estructura de act, en sa_handler se pone que rutina seguir.
        - En oact esta la configuración.
    - **Estructura sigaction{void (*sa_handler)(); sigset_t sa_mask; int sa_flags;};**
        - Sa_handler Es el manejador, lo que hace, puede ser SIG_DFL por defecto, SIG_IGN ignora la señal o un dirección de una función.
        - Sa_mask Mascara de señales a ignorar durante el manejador.
        - Sa_flags Opciones.
  - **Lista de señales importantes**: Están incluidas en signal.h, que son propias del sistema operativo.
    - SIGILL Instrucción ilegal.
    - SIGALRM Cuando termina un temporizador.
    - SIGKILL Mata al proceso.
    - SIGSGEV Violación de segmento de memoria.
    - SIGUSR1 y SIGUSR2 Reservadas par el uso del programador.

- **Conjunto de señales:**
  - int sigemptyset(sigset_t * set);
    - Crea un conjunto vacío de señales.
  - int sigfillset(sigset_t * set);
    - Crea un conjunto lleno con todas la señales posibles.
  - int sigaddset(sigset_t * set, int signo);
    - Añade una señal a un conjunto de señales.
  - int sigdelset(sigset_t * set, int signo);
    - Borra una señal de un conjunto de señales.
  - int sigismember(sigset_t * set, int signo);
    - Comprueba si una señal pertenece a un conjunto.

- **Temporizadores:** El sistema operativo mantiene un temporizador por proceso, en su BCP.
  - Es el sistema operativo quien actualiza todos los temporizadores.
  - Cuando llega a cero el SO pasa al proceso la señal SIGALRM, para que se ejecute la rutina de tratamiento.
  - **Establecer un temporizador:** int alarm(unsigned int sec) puede cambiarse sec por otra unidad de tiempo. Si el parámetro es 0, desactiva el temporizador.
  - **Diferencia de sleep y alarm:**
    - **alarm**, es una llamada al sistema y del proceso se duerme exactamente el tiempo indicado, lo tiene medido.
    - **sleep**, no es una llamada al sistema y como mínimo duerme el tiempo indicado.
- **Excepciones**: Cuando el hardware detecta condiciones especiales; fallo de pagina, escritura a pagina de solo lectura, desbordamiento de pila, violación de segmento, syscall, ...
  - Transfiere control al SO para su tratamiento, que:
    - Salva el contexto proceso.
    - Ejecuta la rutina si es necesario.
    - Envía una señal al proceso indicando la excepción.
  - Sirven para optimizar el rendimiento, ya que ahorran código de comprobaciones.
  - Muchos lenguajes usan mecanismos llamados Exceptions para controlar errores.
- **Entorno de un proceso**: Se hereda del padre y contiene los siguientes datos:
  - Vector de argumentos del comando del programa
  - Vector de entorno, una lista de variables que el padre pasa al hijo. <nombre, valor>
    - Es una forma flexible de comunicar ambos procesos y determinar aspectos de la ejecución del hijo en modo usuario. Partícula a aspectos a nivel de cada proceso.
  - **Variables de entorno:** Mecanismo de paso de información a un proceso.
  - **Localización:** El entorno se coloca en la pila del proceso al iniciarlo. Y se recibe como tercer parámetro de main la dirección de la tabla de variables de entorno.

Llamadas de entorno

- `char * getenv(const char * var);`
  - Obtiene el valor de una variable de entorno.
- `int setenv(const char * var, const char * val, int overwrite);`
  - Modifica o añade una variable de entorno.
- `int putenv(const char * par);`
  - Modifica o añade una asignación var=valor

Entorno de un proceso en Windows

- `DWORD GetEnvironmentVariable(LPCTSTR lpszName, LPTSTR lpszValue, DWORD valueLenght);`
  - Devuelve el valor de una variable de entorno.
- `BOOL SetEnvironmentVariable(LPCTSTR lpszName, LPTSTR lpszValue);`
  - Modifica o crea una variable de entorno.
- `LPVOID GetEnvironmentStrings();`
  - Obtiene un puntero a la tabla de variables de entorno.

Temporizadores en Windows

- `UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);`
  - Activa un temporizador y ejecuta la funció lpTimerFunc cuando venza el tiempo.
  - La función debe cumplir con:
    - VOID TimerFunc(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dwTime);
- `BOOL KillTimer(HWND hWnd, UINT uIdEvent);`
  - Desactiva un temporizador.
- `VOID Sleep(DWORD dwMilliseconds);`
  - Hace que el hilo actual se su suspenda durante un cierto tiempo.