Tema 3.1: Procesos concurrentes y problema en la comunicación y la sincronización.

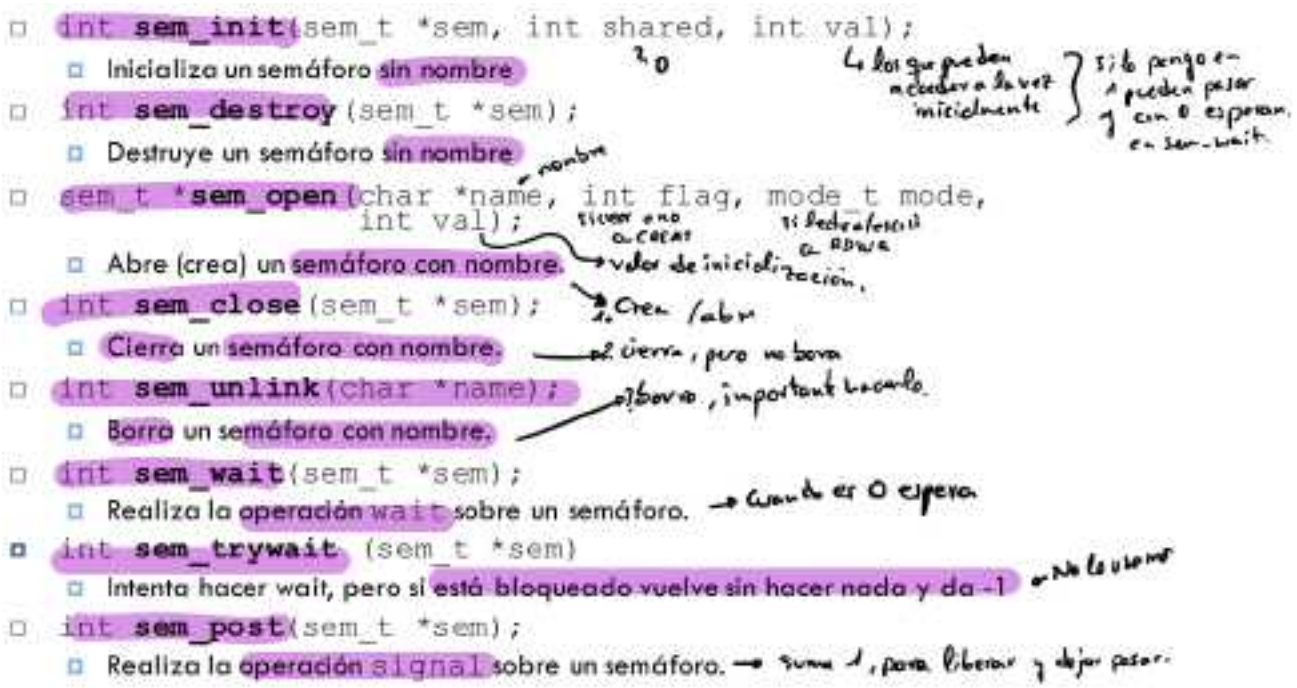
- **Procesos concurrentes:** Dos procesos ejecutándose en el mismo momento, es decir, se ejecutan de manera que su intervalos de ejecución se solapan. No que uno se paso a otro.
  - **Tipos de concurrencias:**
    - **Concurrencia aparente:** Hay mas procesos que procesadores, los procesadores se multiplexan en el tiempo. Pseudoparalelismo.
    - **Concurrencia real:** Cada proceso se ejecuta en un procesador, se produce una ejecución en paralelo. Paralelismo real.
  - **Modelos de programación concurrente:**
    - **Único procesador:** El sistema operativo se encarga de repartir el tiempo entre los procesos, incluso expulsándolos. Es inherente.
    - **Multiprocesador**: Combinan paralelismo real con pseudoparalelismo. Ya que normalmente hay mas procesos que procesadores.
    - **Sistema distribuido:** Varios computadores conectados por red.
  - **Ventajas de ejecución concurrente:**
    - Facilita la programación, se puede estructurar en procesos separados.
    - Acelera la ejecución de cálculos, división de cálculos en procesos ejecutados en paralelo.
    - Mejora la interactividad de las aplicaciones, separar las tareas de procesamientos de las tareas de atención de usuarios.
    - Mejora el aprovechamiento de la CPU, aprovechan las fases de E/S de una aplicación.
  - **Tipos de procesos concurrentes:**
    - **Independientes**: Se ejecutan concurrentemente pero sin ninguna relación.
      - No necesitan comunicarse, ni sincronizarse .
    - **Cooperantes**: Se ejecutan concurrentemente con alguna interacción entre ellos.
      - Pueden comunicarse entre si y pueden sincronizarse.
  - **Interacciones entre procesos:**
    - **Accesos a recursos compartidos**: Procesos que comparten recursos y que compiten por un recurso.
    - **Comunicación**: Procesos que intercambian información.
    - **Sincronización**: Un proceso debe esperar a un evento en otro proceso.
- **Condiciones de carrera**: Cuando varios procesos trabajan sobre el mismo recurso a la vez y el resultado que obtenemos es cada vez distinto, y nunca se obtiene el resultado correcto. Este problema se debe a que dependen de la velocidad de cada proceso.
  - El funcionamiento y resultado de un proceso debe ser independiente de su velocidad relativa de ejecución con respecto a otros procesos. Para garantizar que el orden de ejecución no afecte al resultado, se utilizan mecanismos como la Exclusión Mutua.
  - **Exclusión Mutua:** Cuando un proceso usa un recurso compartido nadie mas lo puede usar, de esta manera nos aseguramos de que no acceden dos procesos a los mismos datos de forma simultánea. Solamente un proceso puede estar simultáneamente en la sección critica de un recurso.
    - **Sección critica**: Es el segmento de código que manipula un recurso.
    - **Problemas de la sección critica**:
      - **Interbloqueos**: Se produce cuando admitimos exclusión mutua para mas de un recurso, de esta manera los dos pueden quedarse bloqueados y no avanzan. Cuando esta en una sección critica y solicita entrar en otra, y el proceso que esta en esa otra esta esperando a entrar en la sección critica del primero.
      - **Inanición**: Un proceso queda indefinidamente bloqueado, a la espera de entrar. No puede acceder a la sección critica porque llegan otros.

- **Condiciones para la exclusión mutua:**
  - Solamente se permite que un proceso este en la sección critica de un recurso.
  - Tiene que ser justo y dejar a todos entrar, no postergarlo indefinidamente.
  - Si una sección critica no esta siendo accedido, se podrá acceder sin demora.
  - No puede depender de la velocidad relativa de los procesos.
  - El tiempo dentro de una sección critica es finito.
- **Mecanismos de sincronización**: Cualquier mecanismo que solucione el problema de la sección critica debe proporcionar sincronización entre procesos. Primero se debe solicitar permiso para entrar en la sección critica y cuando termine lo debe indicar.
  Alternativas:
  - **Desactivar interrupciones**, en el caso de los procesadores monoprocesador, de esta manera no ejecuta el otro proceso para acceder a la sección critica.
  - **Instrucción maquina**, con una variable, test and set o swap. Pero implican espera activa, en un bucle o similar, y ademas son posibles inanición e interbloqueos.
  - **Solución de Peterson:** Solo valida para 2 procesos. Asume que Load y Stores son atómicas e no interrumpibles. Ambos procesos comparte 2 variables, una que es el turno y otra que es flag un array de 2 booleanos. Turno indica quien entra y flag controla si un proceso está listo para entrar.
  - **Semáforos (Dijkstra):** Sincronización de procesos mediante un mecanismo de señalización, un semáforo. Un semáforo se asocia a una sección critica.
  - **Las posibles operaciones son:**
    - Primero, iniciación a un valor, que no debe ser negativo.
    - semWait, que reduce el contador del semáforo. Espera a que se desbloquee, y cuando entra reduce el contador para bloquearlo.
      - Se bloquea cuando el contador<0.
    - semSignal, que incrementa el contador del semáforo. Se hace al salir, para que otro pueda entrar.
      - Se desbloquea cuando el contador<=0
  - **Código**:
    - **Declarar**: sem_t semáforo;
    - **Inicio**: sem_init(&semáforo, 0,1);
    - **Entrada/Pedir acceso**: sem_wait(&semáforo)
    - **Salida/Indicar abandono**: sem_post(&semáforo);
    - **Fin**: sem_destroy(&semáforo);
- **El problema del productor-consumidor:** Proceso/Thread que va escribiendo algo en un buffer y otro lo va leyendo. Se puede hace con semáforos, primero uno se encarga de controlar el acceso a ese espacio intermedio y otro se encarga de dar paso a cada uno u otro. El segundo lo empieza en 0, para que uno lo primero que haga es esperar y mientras otro hace un camino hasta aumentar ese numero y dar paso al otro. Los dos modifican la zona de datos compartida.
  - El proceso productor, produce elementos de información.
  - El proceso consumidor, consume elementos de información.
  - Hay un espacio de almacenamiento intermedio, que es el buffer.
- **El problema de los lectores-escritores:** Se plantea cuando se tiene un área de almacenamiento compartida.
  - Múltiples procesos leen información.
  - Múltiples procesos escriben información.
  - **Condiciones**: Se le da prioridad una de las dos funciones, escritura o lectura.
    - Cualquier numero de lectores pueden leer de la zona de datos concurrentes, pero si se esta escribiendo no puede leer nadie y solo escribir uno.
    - Solamente un escritor puede modificar la información a la vez.
    - Durante una escritura ningún lector puede realizar una consulta.

- **Lectores con prioridad:** Si hay alguien leyendo pueden entrar mas, pero solo se podrá escribir si no hay nadie leyendo. Este puede provocar inanición para escritores.
- **Escritores con prioridad:** Si se esta escribiendo no se admiten lectores.

• 3.2 Hilos y mecanismo de comunicación y sincronización.

- **Mecanismos de comunicación**: Permiten la transferencia de información entre dos procesos.
  - Archivos, Tuberías, variables en memoria compartida y Paso de mensajes.
- **Mecanismos de sincronización**: Permiten forzar a un proceso a detener su ejecución hasta que ocurra un evento. Al recibir una señal se desbloquea. La operacion debe ser atómica.
  - Señales, Tuberías, Semáforos, Mutex y variables condicionales y Paso de mensajes.
- **Semaforos POSIX:** Mecanismo de sincronización para procesos y threads.
  - **Semáforos con nombre**: Se utiliza para distintos procesos, deben conocer el nombre.
  - **Semáforos sin nombre**: Solo puede ser utilizado por el proceso que lo crea, por lo tanto sirve para threads de un proceso, o procesos que tengan memoria compartida.



  - **Estructura (ya creado el semáforo):**
    - sem_wait(s); //Entrada, resta 1
    - sección critica, codigo que usa la sección.
    - sem_post(s); //Salida, suma 1
- **Diferencias con otros problemas:**
  - **Lectores-escritores:** Pueden leer y escribir múltiples procesos, leer simultaneamente cuantos quieran, pero escribir solo puede hacerlo uno a la vez y sin leer otros.
  - **Exclusión mutua:** Solo permite a un proceso acceder a la vez a la informacion.
  - **Productor-consumidor:** Los dos procesos modifican la zona de datos compartida.

- **Mutex (Mutual Exclusion) y variables condicionales**: Mecanismo de sincronización indicado para procesos ligeros, threads. Es un semáforo binario, 1 o 0, con dos operaciones:
  - **lock(m)** Bloquea el mutex, y si la lo esta suspende el proceso.
  - **unlock(m)** Desbloquea el mutex, pero solo uno de ellos.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t * attr);
```

- Inicializa un mutex.

```
int pthread_mutex_destroy(pthread_mutex_t  *mutex) ;
```

- Destruye un mutex.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Intenta obtener el mutex. Bloquea al proceso ligero si el mutex se encuentra adquirido por otro proceso ligero.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Desbloquea el mutex.

- Variables condicionales: Variables de sincronización asociadas a un mutex, tiene dos operaciones, que conviene ejecutarlas entre lock y unlock:
  - **wait** Bloquea el thread y le expulsa del mutex, lo libera para el otro.
  - **signal** Desbloquea a uno o varios procesos suspendidos en la variable condicional.

```
int pthread_cond_init(pthread_cond_t*cond,
                      pthread_condattr_t*attr);
```

- Inicializa una variable condicional.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Destruye un variable condicional.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Se reactivan uno o más de los procesos ligeros que están suspendidos en la variable condicional cond.
- No tiene efecto si no hay ningún proceso ligero esperando (diferente a los semáforos).

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Todos los threads suspendidos en la variable condicional cond se reactivan.
- No tiene efecto si no hay ningún proceso ligero esperando.

```
int pthread_cond_wait(pthread_cond_t*cond,
                      pthread_mutex_t*mutex);
```

- Suspende al proceso ligero hasta que otro proceso señaliza la variable condicional cond.
- Automáticamente se libera el mutex. Cuando se despierta el proceso ligero vuelve a competir por el mutex.

  - **Uso de mutex y variables condicionales:**

- Proceso ligero A

```
lock(mutex);  /* acceso al recurso */
comprobar las estructuras de datos;
while (recurso ocupado)
        wait(condition, mutex);
marcar el recurso como ocupado;
unlock(mutex);
```

- Proceso ligero B

```
lock(mutex);  /* acceso al recurso */
marcar el recurso como libre;
signal(condition, mutex);
unlock(mutex);
```

- Importante utilizar while

- **3.3 Desarrollo de servidores concurrentes. (Mirar diapositivas para el codigo)**
  - **Servidores de peticiones:** Se aplican en muchos contextos. Un servidor recibe peticiones que debe procesar.
    - **Estructura**:
      - **Recepción de peticion**: Cada peticiones requiere un cierto tiempo en operaciones de entrada/salida.
      - **Procesamiento de las peticion**: Tiempo de procesamiento en CPU.
      - **Envio de respuesta**: Tiempo de entrada/salida para contestar.
    - Una solucion es ejecutar la recepción, procesamiento y envío, en un único proceso. Es muy lento, si llegan dos peticiones al mismo tiempo o mientras una se procesa se pierden.
  - **Solución basada en procesos**: Cada vez que llega una peticion se crea un proceso hijo.
    - El proceso hijo realiza el procesamiento de la peticion.
    - El proceso padre pasa a esperar la siguiente peticion.
    - El problema es tener que arrancar un proceso por cada peticion que llega, y terminarlo por cada petición. Consume demasiados recursos.
  - **Solución basada en hilos bajo demanda**: Cada vez que se recibe una peticion se crea un hilo.
    - Un hilo receptor encargado de recibir las peticiones.
    - Cada vez que llega una peticion se crea un hilo y se le pasa una copia la peticion al hilo recién creado.
    - La creacion y terminación de hilos tiene un coste menor que la de procesos, pero sigue siendo un coste.
    - Hay que controlar la condición de carrera.
  - **Solución basada en pool de hilos**: Se tiene un numero fijo de hilos creados desde el principio para ejecutar un servicio.
    - Cada vez que llega una peticion se pone en un cola de peticiones pendientes.
    - Todos los hilos esperan a que haya alguna peticion en la cola y la retiran para procesarla.