**DEFINICIONES IMPORTANTES**

No hay un orden preestablecido en la ejecución ⇒ no determinismo

(ejecuciones con la misma “entrada” puede generar diferentes “salidas”)

No **determinismo**, en ejecución concurrentes con la misma entrada se pueden obtener resultados diferentes *depende del orden en que se van ejecutando las cosas*.

Algunas **ventajas de la programacion concurrente**

- La velocidad de ejecución que se puede alcanzar.

- Mejor utilización de la CPU de cada procesador, de los cores.

- Explotación de la concurrencia inherente a la mayoría de los problemas reales, soluciones más sencilla de pensar/ver.

Un Programa concurrente va a estar formado por varios procesos, este programa es para resolver un único problema, ese programa va a estar compuesto por varios procesos, distintos comportamientos, pueden ser independientes los procesos cada uno resuelve su parte sin interactuar, pero vemos cuando estos procesos cooperan o compiten, entonces los procesos puede ser de tres formas, **independientes, cooperar o competir**

**Cooperar** c/u ejecutar una parte y después de alguna manera esas partes se integran para obtener el resultado final.

**Compiten** pelean para utilizar cierto recurso compartido(venta de pasaje de avion, usuarios compiten para acc a cierto pasaje)

Esto no quiere decir que un proceso todo el tiempo compite, ni todo el tiempo coopera ni es independiente, esto puede ir variando.

Ejemplos abuelitas.

**Independiente** lo que menos nos interesa en la materia, tres abuelas, cada una de las abuelas va a tejer su propio pullover y agujas.

**Competencias**: cierto recurso compartido, comunes a los procesos o subconjunto de procesos que tienen que trabajar con ello, si ese recurso podría ser usado por más de un proceso a la vez no habría problemas, cuando puede utilizarlo uno solo compite por el recurso.

Ahora dos abuelas, compiten por el ovillo de lana uno blanco y uno gris, cuando quieren usar el mismo, ver en qué caso utiliza cual, compiten por el uso de los ovillos, vamos a tener que secuencializar, termine la otra y se lo dé a la abuela.

Problemas:

**DeadLock** → ninguno suelta su recurso hasta no obtener el otro y se produce deadlock(punto muerto), abrazo mortal porque ninguno proceso va a poder continuar.

**Inanición →** alguno de los procesos nunca logre conseguir el recurso porque el otro gana, cuando más procesos hay es más probable que esto se dé.

**Cooperación →** dividimos el problema en partes y luego lo unimos para formar lo que sea el resultado. Hay muchas decisiones que tomar, usamos 4 , o 5 tres que procesen y una que una? el que terminó antes? el más rápido? Lo mejor posible en cuanto al tiempo de ejecución.

Los procesos se comportan de las tres maneras, pueden intercalarse entre ellas.

Otro enfoque: un sóla máquina dedica una parte del tiempo a cada

componente del objeto ⇒ Concurrencia sin paralelismo de hardware ⇒

Menor speedup.

Dificultades ⇒

Distribución de carga de trabajo.

Necesidad de compartir recursos.

Necesidad de esperarse en puntos clave.

Necesidad de comunicarse.

Necesidad de recuperar el “estado” de cada proceso alretomarlo.

**CONCURRENCIA Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.**

**HILOS Y PROCESOS**  
Al tener el ED de un mismo proceso compartido tiene sus inconvenientes, porque podrá haber dos hilos modificar el valor de una variable al mismo tiempo → **interferencia**.

Necesito algún mecanismo que me permita evitar las interferencias, sino resultado erróneos, distinto a no determinismo→ dos result dif. La concu puede estar provista por el lenguaje como en Java, o provista por el SO, ciertas librerías y a través de eso el SO se encarga de generar la concurrencia.   
 **COMUNICACIÓN**  
**En algún momento puede necesitar cierto dato que tiene otro de los procesos.** Comunicación entre procesos, modo en que se organizan y transmiten los datos, requiere especificar ciertos protocolos, como progresar y que se haga de forma correcta.

En la clasificación en dos grandes grupos,

Los procesos se COMUNICAN: • Por Memoria Compartida.

• Por Pasaje de Mensajes.

• **Por Memoria Compartida:** nos permite diferenciar la forma en que se programa y se trabaja, necesito si o si algún lugar de memoria donde todos los proc pueda acceder, dejando los datos en alguna dirección donde otro pueda tomarlo, todo a través de esa

memoria compartida, una físicamente y además los procesos acceder al mismo ED. Cuidado con el uso dos procesos quieren modificar la misma variable por ejemplo. Necesito que la herramienta me brinde algún bloqueo o liberar el acceso a una posición de memoria. Lo que se usa semáforo, nos habilitará o no el paso. Una máquina que tenga una memoria compartida, que permita que los procesos puedan acceder al mismo espacio de direcciones.

• **Por Pasaje de Mensajes.** En las unidades de procesadores separadas, no existen memoria compartida, la comunicación enviando mensajes a través de una red, arquitectura distribuida, procesos separados, también utilizado en memoria compartida y esos canales se simulan por espacio de memoria pero los procesos no pueden acceder al mismo espacio de direcciones, cada uno tiene su ED, el único que conoce eso es el dueño de los datos y para compartir esa info lo hace a través de mensaje. Canales lógicos o físicos que me permiten transmitir, tengo que poder definir esos canales, sentencias primitivas para los datos que se envían a través de esos canales, a quien le envió, de quien recibo información.

**SINCRONIZACIÓN**

La sincronización es la posesión de información acerca de otro proceso para

coordinar actividades. Los procesos se sincronizan:

• **Por exclusión mutua**. Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo solo por un proceso a la vez, y hasta que no termine no cambia. Ejemplo: Una variable compartida que tiene que ser modificada, el uso de una impresora. Si el programa tiene secciones críticas que pueden compartir más de un

proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse

en la misma sección crítica al mismo tiempo.

**• Por condición.** Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada. Ejemplo: elemento que hay en una cola, no puedo ir y sacarlo espero que este vacía para recién hacer el pop y sacar el elemento.

*Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores).*

Problema concurrente que debe utilizar ambos, la utilización de memoria compartida, al cual todos los proc pueden acceder y es limitada, procesos productores y consumidores, no puedo dejar elementos hasta que alguien lo haya dejado, los consumidores tampoco pueden tomarlo hasta que al menos haya un dato en el buffer, se dan ambos tipos de sincronización, el consumidor se bloquea hasta que hay algo y es consumido, se bloquea hasta que haya espacio para dejar un elemento, esto es por condición.

Si hay más de un productor, los productores van a tener que sincronizarse entre ellos por exclusión mutua para que de los productores uno solo pueda acceder.

Del otro lado tenemos la sincronización por exclusión mutua, si hay más de un productor para dejar un elemento pero un único elemento, por exclusión mutua van a tener que sincronizarse.  
 **INTERFERENCIA**

Un proceso toma una acción que invalida las suposiciones de otro hechas por otro proceso.

**PRIORIDAD Y GRANULARIDAD**

Un proceso que tiene mayor **prioridad** puede causar la suspensión(preemption) de otro proceso concurrente.

Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

La **granularidad** de una aplicación está dada por la relación entre el cómputo y la

comunicación.

Relación y adaptación a la arquitectura.

Grano fino cantidad de cómputo, y comunicación, poco cómputo y comunicar y grano grueso código grande y luego comunicado, bloque de trabajo con poca comunicación.

**MANEJO DE LOS RECURSOS**

Uno de los temas principales de la programación concurrente es la administración de recursos compartidos:

• Esto incluye la **asignación** de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.

• Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (**fairness**).

• Dos situaciones NO deseadas en los programas concurrentes son la **inanición** de un proceso (no logra acceder a los recursos compartidos) y el **overloading** de un proceso (la carga asignada excede su capacidad de procesamiento).

• Otro problema importante que se debe evitar es el **deadlock**.  
Dos (o más) procesos pueden entrar en deadlock, si por error de

programación ambos se quedan esperando que el otro libere un

recurso compartido. La ausencia de deadlock es una propiedad

necesaria en los procesos concurrentes.

**• 4 propiedades necesarias y suficientes para que exista deadlock son:**

* Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
* Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.
* No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
* Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

**• Requerimientos de un lenguaje de programación concurrente:**

* Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
* Mecanismos de sincronización.
* Mecanismos de comunicación entre los procesos.

**PROBLEMAS DE LOS PROGRAMAS CONCURRENTES**

* Los procesos no son independientes y comparten recursos. La necesidad de

utilizar mecanismos de exclusión mutua y sincronización agrega complejidad

a los programas.

* Los procesos iniciados dentro de un programa concurrente pueden NO estar

“vivos”. Esta pérdida de la propiedad de liveness puede indicar deadlocks o

una mala distribución de recursos.

* Hay un **no determinismo** implícito en el interleaving de procesos concurrentes.

Esto significa que dos ejecuciones del mismo programa no necesariamente

son idénticas ⇒ dificultad para la interpretación y debug.

* Posible reducción de performance por overhead de context switch,

comunicación,sincronización.

* Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos

secuenciales.

* Necesidad de adaptar el software concurrente al hardware paralelo para

mejora real en el rendimiento.  
  
La performance del context switch, cuando saco un proceso tengo que de alguna manera guardar el estado para cuando vuelvo guardar su estado todo este context switch lleva su tiempo.  
  
 **CONCURRENCIA**  
Concepto de software no restringido a una arquitectura particular de hardware ni a un

número determinado de procesadores.

Especificar la concurrencia implica especificar los procesos concurrentes, su comunicación y su sincronización.  
 **PARALELISMO**

Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal

de reducir el tiempo de ejecución.  
  
**Clases de instrucciones**

**Asignación**

Asignación simple x = e

Sentencia de asignación compuesta   
 x:=x+1;  
 y:=y-1;  
 a[2]=6 ,   
 aa[2,6]= a[4]  
  
Llamado a funciones x=f(y)+g(6) - 7

swap: v1:=:v2 → intercambiar los valores de dos variables

skip: termina inmediatamente y no tiene efecto sobre ninguna variable de programa → si utilizamos, no hace nada no utiliza procesador no hace cambios.   
  
Sentencia **co**:

**co** S1 // ..... // Sn oc → Ejecuta las Si tareas concurrentemente.

La ejecución del co termina cuando todas las tareas terminaron.

Cuantificadores.

co [i=1 to n]{ a[i]=0; b[i]=0 } oc → Crea n tareas concurrentes.

**Process:** otra forma de representar concurrencia

processA {sentencias} → proceso único independiente.

Cuantificadores.

process B [i=1 to n] {sentencias} → n procesos independientes.

Diferencia: **process** ejecuta en background, mientras el código que contiene un **co** espera a que el proceso creado por la sentencia co termine antes de ejecutar la siguiente sentencia.  
  
  
**ATOMICIDAD DE GRANO FINO**  
**Estado** de un programa concurrente, el valor que tienen las variables compartidas de cada proceso en cierto momento, si uno corta la ejecución de los proceso y guarda esos valores ese sería el estado de un programa.

Cuando un proceso tiene que ejecutar un programa concurrente lo que ejecuta es una serie de sentencias → formadas por acciones atómicas. Cada proceso ejecuta una acción atómica cada una, lo que hace es una transformación de estado indivisible, los estados intermedios que se pueden llevar a cabo mientras se está llevando a cabo esa acción, no se ve por los otros , por ende no puede haber una interferencia.

Una acción atómica hace una transformación de estado **indivisibles** (estados intermedios invisibles para otros procesos).

• Ejecución de un programa concurrente → intercalado (interleaving) de las acciones atómicas ejecutadas por procesos individuales.

• Historia de un programa concurrente (trace): ejecución de un programa concurrente con un interleaving particular. En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.

• Interacción → determina cuáles historias son correctas.  
  
**La sincronización por condición permite restringir las historias de un programa concurrente para asegurar el orden temporal necesario.**  
Atomicidad de grano fino   
Si una expresión e en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.

• Si una asignación x = e en un proceso no referencia ninguna variable

alterada por otro proceso, la ejecución de la asignación será atómica.

**Referencia crítica en una expresión ⇒ referencia a una variable que es modificada por otro proceso.**

**Asumamos que toda referencia crítica es a una variable simple leída y escrita**

**atómicamente.  
  
Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez  
  
Fairness: que todos tengan chances de avanzar con su ejecución**

**Acciones atómicas es elegibles si es la próxima que se va a ejecutar atómicamente. Entre las cuales vamos a ver cual se va a ejecutar en el momento, para formar parte de la historia que se está armando.   
  
Una política de scheduling determina cual será la próxima en ejecutarse.  
  
Fairness** incond, toda acc atom incond(sin await, o si pero sin booleana), elegible en algun momento se va a ejecutar.

Clase 3: (VERLA DE VUELTA)

Problema de la Sección Crítica: implementación de acciones atómicas en

software (locks). Protocolo de entrada y salida, y que los procesos se ejecuten con exclusión mutua.

Barrera: punto de sincronización que todos los procesos deben alcanzar para

que cualquier proceso pueda continuar.  
  
Busy waiting, chequea repetidamente hasta que una condición es verdadera,

ventaja, se puede ejec con instru de cualquier procesador, desventaja ineficiente cuando se habla de programas que tienen mas de un programa ejecutando sobre una unidad de procesamiento, retrasando la ejec de otro proceso para volver v. Si no compartimos unidad de procesamiento esta desventjaa no se presenta.

**Problema de la sección critica** consiste en poder armar acc atom por SW, protocolo de entrada y de salida, podemos usarla para exclusión mutua,   
  
Propiedades que debe cumplir, la soluc a la sección crítica, primero Exclusion mutua → a lo sumo un proceso a la vez debe estar ejec la secc.

Ausencia de deadlock, al menos uno si o si debe poder ejecutar.

Ausencia de demora innecesaria, si ninguno quiere ejec la secc critc, no debe ser impedido de entrar.

Eventual entrada, un proceso que quiere entrar a la secc critica, siempre debe poder hacerlo.

Sentencias await por exclusión mutua cumple esto.

• Si S es skip, y B cumple ASV, 〈await (B);〉 puede implementarse por medio de ⇒ while (not B) skip;  
Me quedo iterando hasta que logro continuar.   
  
Correcto, pero ineficiente: un proceso está spinning continuamente saliendo y entrando a SC hasta que otro altere una variable referenciada en B.

**Clase 4: Semáforos  
  
V: señala la ocurrencia de un evento, increm de forma atom el valor.**

**P: se usa para demorar un proceso hasta que ocurra un evento.**

sem s; → SI o si se deben inicializar en la declaración

sem mutex=1

sem fork[5]=([5] 1); → inicializo 5 sem en 1.

semaforo general,

P(s): < await (s>0) s=s-1; >

V(s): < s= s+1; >

Semáforo binario: 0 o 1, condición que el valor del semáforo sea menor que uno.

solo se puede inicializar con 0 o 1.   
  
NO vamos a respetar ningun orden!!!   
  
Seccion critica: Exclusion mutua

en vez de marcar si esta ocupada o no, decimo si está libre o no.

free solo puede tomar 0 o 1.   
sem free=1

process SC[i=1 to n]  
{

while(true){

P(free)

seccion critica;

V(free)  
seccion no critica

}

}

si inicializo free=0?  
  
Todos los procesos se van a qeudar demorados, nadie ingresa y llega a ejecutar.

**Semáforo de inicialización¡¡BARRERA!!  
Inicializados en 0, porque el evento esperado no ocurrió**   
Barrera entre dos procesos, avisar que llego hasta esa barrera.

Señala que llego, con P, sobre el semáforo del otro proceso.   
  
Cuidado con el orden de P y V, ambos bloqueados, no llegan a ejecutar porque el semáforo quedan bloqueados, intentan pasar p que pueden pasarla cuando tienen mayor que 0,

primero hacemos el v indicando que llegamos hasta la barreda y luego nos quedamos esperando a que llegue el p.

Usamos los semaforos generales, simulando los binarios. Semáforo binario unico, dividido en n partes, que la suma de entre 0 y 1, me perm manejar la em pero la competencia no se da al mismo tiempo entre todos los procesos, alternar el uso o acceso a la seccion critica, servir para acceso alternando distintos tipos de procesos, ejemplo concreto

productores y consumidores, buffer unitario, depositar y si esta lleno se demora, hasta q se vacía, y del lado del consumidor esperar a q haya algo en el buffer.  
  
alternar el acceso al buffer entre os cons y prod, tipo de proceso. Entre todos van a competir a ver quien entra. Dos semaforos, vacio=1 lleno=0

semafoto binario dividido, o los dos son cero o uno y uno.

contar cantidad de unidades libres, rec de múltiples unidades.

consumidor retira los mensajes que fueron depositados ahi y snon retirados , repres por un vector, puntero libre y ocupado, como si fuese una cola circular, desp de pos 0.

Procesos productor consumidor con buffer compartido

typeT buf[n] int ocupado=0 libre=0

sem vacio=n ; → cant de pos del buffer libre

sem lleno= 0 → cual tiene elem ocupados  
  
Process productor{

while(true){

producir mensaje datos

P(vacio); //cuadno haya un elem vacio al menos uno libre, decrem el valor

buff[libre]= datos; //deposito el elm en la pos del buffer dada por la variable libre

libre := libre+1 mod n; //inc libre, si libre en la ultima la sig es 0, cola cirsular.

V(lleno)#depositar //avisar que ahora quedo una pos mas en ese buffer.

}

Process consumidor

{

while(true)

{ P(lleno)

resultado=buff[ocupado]

ocupado = ocupado +1 mod n;

V(vacio) #retirar

consumir mensaje resultado

}  
  
el problema de esta solucuiionm es cuando hay mas de un productor o mas de un consumidor.

si tengo mas de una pos libre, entonces podria darse qe sobreescriban el eleme de el otro, solo sirve para uno solo.

El proceso de depositar o retirar un elemento lo haran dentro de una seccion critica. Un produc o un consumidor al mismo tiempo, la em

dentro de la parte donde deposito o retiro, dentro de secciones criticas..

mutexD para depositar

MutexR para retirar en 1 porqe son para em.   
  
Process productor{

while(true){

producir mensaje datos  
 **P(mutexD)**

P(vacio); //cuadno haya un elem vacio al menos uno libre, decrem el valor

buff[libre]= datos; //deposito el elm en la pos del buffer dada por la variable libre

libre := libre+1 mod n; //inc libre, si libre en la ultima la sig es 0, cola cirsular.

**V(mutexD)**  
 V(lleno)#depositar //avisar que ahora quedo una pos mas en ese buffer.

}

Process consumidor

{

while(true)

{ P(lleno)  
**P(mutexR)**

resultado=buff[ocupado]

ocupado = ocupado +1 mod n;  
**V(mutexR)**

V(vacio) #retirar

consumir mensaje resultado

}  
  
**V(mutexR), P(mutexD) para hacer als secciones criticas.**Varios procesos que compiten por varios recursos compartidos, Procesos P y recursos R, un proceso que debe adquirir los bloqueos sobre todos los recursos, necesito bloquear algunos procesos para poder usar los procesos, cada proceso necesita un subconj superpuesto con otro, pero ningun de P, necesita al mismo conj de R. Podría generarse deadlock, acc a R y que lo tenga otro.

Exclusion mutua selectiva, filosofos 5, tenedores 5, pensando y comiendo, para poder comer adquirir sus dos tenedores, cada tenedor independiente, compiten entre si con el que tienen al lado, derecha u izquierda. Arreglo de semaforos para pder acceder a esos tenedores, arr manejar uso de c u de los tenedores, llamar a p sobre el sem q le corresponde a un tenedor, y v a otro, el problema es que se podría generar dead lock, ocupado por un filosofo particular dif, todos los filosofos van a tener un tenedor y se bloqueo todo el sistema.

Carac para que se cumpla la ausencia de deadlock.

Los tres primeros, toman a la derecha, mientras que otro el 4, toma al revés el derecho y luego el izq, **ROMPEMOS LA POSIBILIDAD DE QUE HAYA DEADLOCK .**

**Exclusión mutua selectiva**Passing de botton

dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los escritores debe ser exclusivo para evitar interferencia entre transacciones. Los lectores pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando

Ambos necesitan acceso exclusivo a la BD. Los escritores como los editores

sem rw = 1;

process Lector [i = 1 to M]

{ while(true)

{ …

P(rw);

lee la BD;

V(rw);

}

}  
  
Así podría haber mas de un lector,   
  
entonces…

process Lector [i = 1 to M]

{ while(true)

{ …

< nr = nr + 1; if (nr == 1) P(rw); >

lee la BD;

< nr = nr - 1; if (nr == 0) V(rw); >

}

}

esto hay que implementarlo, porq es de grano grueso. Agregar un semaforo para poder acceder a la excl mutua. MutexR

int nr = 0; # número de lectores activos   
sem rw = 1; # bloquea el acceso a la BD   
sem mutexR= 1; # bloquea el acceso de los lectores a nr   
  
process Escritor [j = 1 to N]

{ while(true)

{ …

P(rw);

escribe la BD;

V(rw);

}

}

process Lector [i = 1 to M]

{ while(true)

{ …

**P(mutexR);**

nr = nr + 1;

if (nr == 1) P(rw);

**V(mutexR);**

lee la BD;

**P(mutexR);**

nr = nr - 1;

if (nr == 0) V(rw);

**V(mutexR);**

}

}  
  
**Passing the baton**

Varaibles con cant de editores y escritores.  
  
cuando no se puede solo aplicar un semaforo

Las condiciones de espera son diferentes y ademas superpuestas  
No son idependientes,   
  
**Passing the baton: técnica general para implementar sentencias await.   
Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar. Cuando el proceso llega a un SIGNAL (sale de la SC), pasa el baton (control) a otro proceso. Si ningún proceso está esperando por el baton (es decir esperando entrar a la SC) el baton se libera para que lo tome el próximo proceso que trata de entrar**  
*SC el uso de las variables, no es la bd el recurso compartido, sino que el acceso.*

La **sincronización** se expresa con sentencias atómicas de la forma:

La sinc sentencias atom incondicionales, exclusión mutua.  
F1 : 〈 Si 〉 o   
Condicionales  
F2 : 〈 await (Bj ) Sj 〉

Puede hacerse con semáforos binarios divididos (SBS).  
  
e semáforo binario inicialmente 1 (controla la entrada a sentencias

atómicas).

Utilizamos un semáforo bj → distintas cond de los distintos procesos. y un contador dj cada uno con guarda diferente

Bj

; todosinicialmente 0.  
  
bj se usa para demorar procesos esperando que Bj sea true. → un semaforo para demorar

dj es un contador del número de procesos demorados sobre bj. → para contar los demorados, en 0.

e y los bj se usan para formar un SBS: a lo sumo uno a la vez es 1, y cada

camino de ejecución empieza con un P y termina con un único V.   
  
CONJ de variables compartidas qe administran el acceso.   
  
cada acceso a la sc, empieza en un unico p y termina en un unico v.

Como convertiriams lso await cond e incond,

F1 : P(e); Si ; SIGNAL;  
  
acceder a la sc, para trabajar con atomicidad, esperando a recibir el baton, P(e), ejecutamos, luego vemos q hay q hacer, liberando el baton o pasandolo a otro.

F2 : P(e);

if (not Bj) {dj = dj + 1; V(e); P(bj); }

Sj

;

SIGNAL  
  
Una vez que accedemos, lo primero es chequear si la cond es verdadera o no. Si es true work, sino me demoro y tiene efecto passing de baton, me duermo en el sem correspo en el sem bj, entonces increm dj avisando q va a haber otro proc mas, P(bj) antes de dormirme debo liberar la cs, con el V(e), si la cond no se da, inc de los de espera, dj, libero y P.

Soy el unico que podrá liberar.  
  
Que significa el signal, el liberar

SIGNAL: if (B1 and d1 > 0){d1 = d1 - 1; V(b1)}

...

(Bn and dn

> 0) {dn = dn - 1; V(bn)}

else V(e);

Si hay algun proces esperando, por una cond que se volvió verdadera, despierto a alguno de los procesos, le paso el baton, uso excl de la sc, tendría q chequear para cada una de las codn de espera, si es true y si hay alguien esperando, si hubiera una o mas verdaderas se elige una y despertar a uno de los procesos que está esperando por esa condición, decrement la canti de procesos q está esperando, V(b1) sobre el sem, si huebiera q pasar el baton,

Si ninguna se da, falsa o por esa cond no hya nadie, lo q hago es liberar totalmente el baton para q el primero q llega pueda llegar y ejecutar.

No puedo dormirme antes de liberar porq se bloquea todo, tmp incr el contador desp de dormirnos.

Una vez q pasamos el if, conj de sentencias asco en el await, incr en 1 la variable nr, libero el baton, //escribo en bd,

incod en forma atom decrem la cant de escrituras.

**Liberación del botón o pasaje del baton a otro proceso!**

**En este caso el passing baton estoy despertando a cualquiera de los lectores o cualesquiera de los lectores.**

**No respeta el orden de los procesos.**

**Como manejar la locación de recurso y scheduling**quien y en que momento se puede usar cierto recurso, esos proc compiten por un recurso, una sola o muchas, cualquiera de los dos sacoss puede darse. Cada recurso va a estar libre o en uso, unico proceso a la vez, el proceso para trabajar necesita adquirir un subconj por ejem, esos procesos van a tener dos operaciones luego lo utliza y luego retira. Esperar a que tal cond se de y cuando termine liberarlo, dejar disponibles las unidades.  
No vamos a determinar cual es el tipo de proceso, sino cual tiene q pasar, despertar un usuario en aprticular, al que algo, uno SOLO no alguno o cualquiera, despertamos a un proceso en particular.

Passin the baton, como convertirlas a una sol en semáforos, podemos usarlo para hacer la solicitud de recursos y la liberación.   
  
shortest job next, algun proceso puede no se lo asignen porqe tienen q usar mucho tiempo el recurso. Bajar el tiempo de estimación, internamente para poder ejecutarlo. SJN. UNico recursos, bool libre, cuadno hacemos la solic id del proceso y tiempo estimado de uso, rec libre y cuando esta libre ponerlo en falso. A quien despierto, y cómo se va a dormir?   
  
Cómo sería la implementación de la solicitud del recurso e implementación. El conj de tiempo id,

**Semáforo Privado** si hay un único proceso que puede dormirse sobre él, cuando tengo que despertar a alguien en particular y no a cualquiera.  
  
La forma en que se insertan en la estructura ordenada. Una cola donde lo que vamos insertando es el id, modelo practico para cualquier orden, que hay q modificar si en vez de un unico rec tengo varios?

No voy a tener varaible libre, voy a tener un entero que me indica cuantos rec libres hay. Y alguno almacenado con el nombre de los rec libres.

**MONITORES - CLASE 5**  
Modulos de programa, que pueden ser implementados tan eficientemente como los semaforos, es un mecanismo de abstracción de datos, un monitor va a encapsular las caract que vamos a representar con esos monitores, conjunto de operacipones.

variables y operaciones  
ademas compuesto por procesos, cada proceso sus variables locales y puede interactuar con los monitores, cualquier cosa compartida dentro del monitor, resolver de forma explicita uno de los tipos de sinc,   
  
Procesos activos

En algún momento van a ahacer llamados a los monitores para hacer uso del recurso compartido, esos monitores pasivos, cuando un proceso lo llama ahi so ejecuta la invocacion al monitor, si dos procesos necesitan interactuar entre ellos, a través de la invoc a los proce q estan en algun monitor, con una variable y un proceso, y otro proceso darle un valor y tomar un valor, aunqe sea solo tranmitir el datos.   
Ventajas de los monitores, libera de la exc mutua, nos queda la sinc por condicion, al estar separados los procesos y los recursos, al monitor no le improta mucho y tmp como esta implem, en todo caso la forma en qu trabaje ese procedimiento.   
  
El monitor dentro

variables o datos del recurso, tb para el manejo de la sinc entre los procesos dentro del monitor,   
  
Carac nos va a liberar nos va a liberar, de alguno de los dos tipos de sinc, los proc dentro de un mismo mon no se ejec de forma concu, hasta q un proced no ose duerma o no termine su ejec hasta ese momento ningun otro va a poder ejecutar un llamado a ese proceso. Como hay un unico procedimiento dentro de ese monitor al mismo tiempo, no existe la posib de que estan dos al mismo tiempo. La excl mutua se da implicitamente.

LA por condición, dormimos un proceso para liberar un monitor y dejar que otro proceso continue con un procedimiento que se habia dormido en algun momento.

Procesos activos, en algun momento llamado a los monitores para llamar a los recursos compartidos, si dos procesos necesitaran interactuar, no queda ota q a trvés de la invoc a los proced para comaprtir o implem un dato tendre que tener un proced para q el primer proceso deje algo en la variable y otro lo tome.   
  
en comparacion con un tad, es que un monitor agrupa la repres de un rec conmpartido, compartido por mas de un proceso que ejec concurrentemente, van a tener una interfaz y un cuerpo.

operaciones, procedimientos, la interfaz específica las operaciones, son los procedimientos que exporta ese monitor, que pueden ser llamados por distintos procesos para trabajar con cierto recurso compartido, cuerpo el monitor, conjuntos de variables del monitor se denominan **PERMANENTES**, cuando un monitor hace un llamado a un proceso, en la siguiente invocación a alguno de los procedimientos la variable va a seguir siendo la misma, CONSERVA EL VALOR A TRAVÉS DE LAS DISTINTAS INVOCACIONES.   
  
En la interfaz se tiene lo necesario para que el proceso sepa, no van a ser funciones, son procedimientos, param de E/S.

nombreMonitor.op1(param)  
Dentro del cuerpo del monitor los procesos van a poder acc a las var permanentes, a las locales, con sus inicializaciones y a dif de las permanentes, las variables van a ser nuevas, no se mantiene de un llamado a otro.

Cuando se diseña el monitor, no sabemos el orden en el que son llamados lso procedimientos, lo haremos de manera tal que sea como sea que se ejecute este funcione.  
  
cuerpo del monitor, no vamos a hacer la interfaz.

nombre, variables permanentes

cod de incializacion → varialbes q no son simples

NO WHILE TRUE sino queda inutilizado en el cod de inic.

proc op1(par formales){

cuerpo de op1

}

proc opn(par formales){

cuerpo de opn

}

un programa puede tener muchos monitores, un proced en un monitor y otro en otro, la em implícita se da en uno monitor mismo, puedo tener muchos monitores que hagan lo mismo.   
Podrai declarar una arreglo de monitores, que tenga un cuantificador y autom genero esa cant de monitores, si hago un llamado digo a cual quiero acceder,

**Ejemplo de uso de monitores.**

5 empleados que hacen algún prod continuamente. Hay un proc coodinador que cada cierto tiempo debe contar la cantidad total de prod hechos.

Contador dentro del monitor.

Dos proc, por los empl. y otro cood para contar.

monitor TOTAL {

int cant = 0;

procedure incrementar()

{ cant = cant+1; //dentro del monitor

}

procedure verificar(R: out int)

{ R = cant;

}

}

process empleado[id: 0..4] {

while (true)

{ ……

TOTAL.incrementar();

……

}

}  
  
 process coordinador{

int c;

while (true)

{ ……

TOTAL.verificar(c);

……

}

}

La sincronización por condición es programada explícitamente con **variables**

**condición** → cond cv; //internam asoc una cola de procesos dormidos o demorados, el manejo de la cola no es visible, no a mano sacar o poner elem, ni ver la cantidad.

El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador.  
 Operaciones sobre las variables condición,variables dentro del monitor, donde esta declarada la variable condicion:

**• wait(cv)** → el proceso se demora al final de la cola de cv y deja el acceso

exclusivo al monitor. //se queda dormido en la cola, deja el acceso exclusivo del monitor, libera la cola, se duerme en una var cond, para demorar al proc hasta que alguien lo despierte. //duerme o demora, al final de la cola, cola fija. Desp puede cambiar en las sig ops.

• **signal(cv)** → despierta al proceso que está al frente de la cola interna en la var condicion(si hay alguno) y lo saca de ella. El proceso despertado recién podrá ejecutarse cuando requiera el acceso exclusivo al monitor.

• **signal\_all(cv)** → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.

cuadno el proceso despertado puede acceder? depende de la disciplina de señalizacion,

Signal and continue → usamos esta, signal despierta al proc q estaba dormido pero cont usando hasta que termina o el se duerme, el proceso despetadotendra que competir.

Signal and wait, el despertado cont ejec, mientras que el otro pasa a competir por el monitor.

TEORIA NO PRACTICA.

• empty(cv) → retorna true si la cola controlada por cv está vacía. //si hay agun proc o no dormido en la variable

• wait(cv, rank) → el proceso se demora en la cola de cv en orden ascendente de//ademas de codn, segundo param una cierta prioridad, demora el proceso pero de acuerdo al orden dado por rank de menor a mayor, signal no despeirta al primero despierta al de menor valor.

acuerdo al parámetro rank y deja el acceso exclusivo al monitor.

• minrank(cv) → función que retorna el mínimo ranking de demora

permite ver cual es la prioridad del primer proceso que esté dormido.

TOP DE LAS OPER SOBRE LAS COLAS.

¿COMO ES EL COMP DE LOS PROC DE A CUERDO A SIGNAL ADN WAIT, SIGNAL AND CONTINUE?

l and Continued: el proceso que hace el signal continua usando el

monitor, y el proceso despertado pasa a competir por acceder nuevamente

al monitor para continuar con su ejecución (en la instrucción que

lógicamente le sigue al wait).

• Signal and Wait: el proceso que hace el signal pasa a competir por

acceder nuevamente al monitor, mientras que el proceso despertado pasa a

ejecutar dentro del monitor a partir de instrucción que lógicamente le

sigue al wait.

signal cuando no habia ningun proc dormido no tiene ningun efecto,

wait, siempre se duerme vs p depende del valor, solo si es 0

signal si no hay naide no esfecto, sino el primero,vs v increm el semáforo para que un proceso dormido o que hara p continue.

no sigue ningun orden para despertarlos.

**Como simular los semaforos**inic en 1,  
int s=1  
cond pos; //los proc q intentan hacer p se quden dormidos.

cuando hay muchos procesos, cualquiera de los qeu esta esperando afuera puede pasar, hay que chequear la cond cada vez que el proceso es despertado, sino vuelve a ser 0 y no podia continuar, usamos un while, mientrsaas 0 el proc se duerme.

Que dif hay con los semaforos? → en esta soluc, se mantiene un cierto orden en el paso de la op P, no se cumple un orden total, otro proceso puede ganarle el acceso y pase antes el que estaba por ser despertado, se mantinee un cierto orden, en vez de unar un signal usaria un signal all, entocnes todos pasan a competir y todos quieren entrar, solucion apra que sea igual a los semaforos que usabamos, no eficiente o mejor.

Si quisiera hacerla mas eficiente, lograr uqe todos pasen.. la op P en el orden que lo intentaron, los procesos demorados en P mejor q la hagan en el orden q pidieron, orden entre ellos y adeams otro no pueda ganarles a los recien desperados, **passingb the conditions**

==0 me demoro, sin no.

mas de un proceso dormido, nuo llama a v q es el que increm el contador, si yo marco ese cambio en S, dejo con valor 1 a variable s, passin de condition, pero sin modificar el valor de s, s==0, ninguno q intente p va a poder continuar,   
  
1ero cuando el proc llega y quiere hacer P, s==0 me duermo, sino decremento, no hay naide dormido, ni en su secc critica, decrem y lo dejo en 0 para uqe nadie pueda continuar, si no hay nadie dormido inc en 1, despertar al primero pero sin cambiar el valor para q no pase otro. si hay uno dormido, despierto, dejo el sem en 0, si el uqe gana el acc, es otro chequea con el if y el valor del sem es 0, entocnes 2 se duerme en la var cond, cuando deja el monitor accede el proc 1 y puede avanzar,

**siempre que tengo que respetar el orden, tengo que usar la tecnica passing the conditions.**   
  
Se puede reemplazar el empty que no se usa en la practica, con un contador, espera en P ++, en v --.  
  
**Alocacion SJN**

admin el acceso al recu compartido, wait con prioridad

tb passing the condition, y wait con prioridad. primero el que tiene que usarlo por menos tiempo.

| monitor Shortest\_Job\_Next  { bool libre = true;  cond turno;  procedure request(int tiempo)  { if (libre) libre = false;  else wait (turno, tiempo);  };  procedure release ()  { if (empty(turno))libre = true  else signal(turno);  };  } | Se usa wait con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso.   Se usa empty para determinar si hay procesos demorado  Cuando el recurso es liberado, si hay procesos demorados se despierta al que tiene mínimo rank.  Wait no se pone en un loop pues la decisión de cuándo puede continuar un proceso la hace el proceso que libera el recurso. |
| --- | --- |

la consulta de la condicion siempre con el if.   
  
como resuelvo esto sjn pero con las herram de la practica. No empty sobre la variable condicion y no wait con prioridad, orden de llagada.  
  
utilizar variables condiciones privadas arreglo turno, orden de los procesos, cola explitaa espera para dormir proc.   
inserto\_ordenado(par id tiempo) //si o si antes de dormirlo, sino no se.

wait(turno[id) //duerme uno

si esta vacia la cola espera libero, no hay nadie esperando

else , sacar el primero (id), signal sobre la variable cond de ese proceso.

**Técnicas de Sincronización   
Buffer Limitado: Sincronización por Condición Básica**

**vector de pos q pueden ser ocuapdas, punteros pos libreu ocupada, cantiad=0 cant de elem en el buffer.**depositar, si no esta lleno, cant ==n , duermo en condición para los productores,

retirar chequeo si hay elem, sino me duermo, me despi y vuelvo a cehquear, cuando salgo del while lo pongo en el param de salidaa, dec la canti, un elem menos, y por si hay algun prod dormido los despierto.

monitor Shortest\_Job\_Next

{ bool libre = true;

cond turno[N];

cola espera;

procedure request (int id, int tiempo)

{ if (libre) libre = false

else { insertar\_ordenado(espera, id, tiempo);

wait (turno[id]);

};

};

procedure release ()

{ if (empty(espera)) libre = true

else { sacar(espera, id);

signal(turno[id]);

};

};

}

**Técnicas de Sincronización   
Lectores y escritores: Passing the Condition  
1 moni con 4 proced**

cantidad de lectores, y cant de lectores.

pedido\_leer{

nr>0 wait se duermen // cheuqeamos la cond) porque puede ser qeu otro logro pasar y uso.

sale del while es proqe no hay otro.

libera leer

dec lectores--

if=0, signal ok\_escribir  
  
escri escribir

si hay lectores o escritores, wait(ok\_escribir) // se duermen,

cuando pasa no hay nadie en la bd, inc la cant de escrit

libera escribir

nw=nw-1

signal(ok\_escri)

signalall(ok\_leer) //esto le da un poco de prioridad al lector ..

MISMO PROBLEM con PASSING THE CONDITION   
 **Tecnicas de sincronizacion**

**Covering conditions**

Diseño de un reloj lógico

monitor Timer

{ int hora\_actual = 0;

cond chequear;

procedure demorar(int intervalo)

{ int hora\_de\_despertar;

hora\_de\_despertar=hora\_actual+intervalo;

while (hora\_de\_despertar>hora\_actual)

wait(chequear);

}

procedure tick( )

{ hora\_actual = hora\_actual + 1;

signal\_all(chequear);

}

}  
  
NO nos conviene despetar a todos!

**Wait con prioridad**

no despertar para que vueñva a dormirse, solo cuando tenga que irse.

usar un wait con prioridad, los voy durmiendo en una variable condicion de acuerdo a su tiempo. cuando me despertaron es porque ya tengo que irme. tick inc en 1 y despierta a todos aquellos donde su hora ya llego, si el rank es menro a la actual lo teine q despertar, hasta encontrar el primero, donde frena y llamara de vuelta al prox.

mismo problema para la practica!! porque no puedo usar wait y minrank.

ordenada de acuerdo a quien minimice la hora

inc en uno el contador, se fija de la cola el valore de la hora que tiene que despertarse, mientras aux < despertar

saco el proceso y despierto haciendo el signal en esa posicion.

cleitnes y peluquero, es a traves del monitor : tres proced,   
  
peluqiero cuadno ya paso al peloquero, para cortar, le cortaa, luego corte terminado, se queda dorm en una variable cliente .   
  
cuando hay que hacer un encuentro entre dos, no siempre se el orden, controlar si el otro llego para ver si nos dormimos o no. chequear si no ha llegado el peloquero.  
 **Scheduling de disco**

direc fisica, cilindro, num de pista y desplazamiento.   
minimizar el mov de la cabeza de lect o esc, a quien vamos a dejar leer o escribir primero?   
  
Cscan, se atienden como lec o esc al que este mas cercano y todo en una sola direccion, de mas chico al mas grande y en el mismo sentido,

scheduling de disco usando cscan,   
  
monitor de scheduler Disco

{ int pos=-1 v\_act=0 → vuelta actual

v\_prox=1 → pendientes de la vuelta sig

cond scan

pos == -1 //libre

pos = cilindo al que accede

elseif //usado y mayor el valor de posicion,

entocnes wait/dormir en la varaible cond asoc con la vuelta actual.

else usaro y menor// ya pasamos ese cilindro

wait pero asoc a la provima vuelta

proc liberar

a cual despierto, si hay alguno pendiente en de la vuelta acrual, cilindro mas cerca a la pos, lo actualizo y despierdo ese proceso.

si es vacia tegn oque ver con la vuelta proxima, sino despierto al qu esta

}

si accede directamente al disco y salta el cheduler ya no sirve, intentar mejorar la solucin, usar un monitor intermediario entre los procesos usuario y el disk driver. El monitor envia los pedidos al disk driver en el orden de preferencia deseado.

monitor Interfaz\_al\_Disco

{ variables permanentes para estado, scheduling y transferencia de datos.

procedure **usar\_disco**(int cil, parámetros de transferencia y resultados)

{ esperar turno para usar el manejador

almacenar parámetros de transferencia en variables permanentes

esperar que se complete la transferencia

recuperar resultados desde las variables permanentes

}

procedure **buscar\_proximo\_pedido**(algunType &resultados)

{ seleccionar próximo pedido

esperar a que se almacenen los parámetros de transferencia

setear resultados a los parámetros de transferencia

}

procedure transferencia\_terminada(algunType resultados)

{ almacenar los resultados en variables permanentes

esperar a que **resultados** sean recuperados por el cliente

}

}