

# SINCRONIZACIÓN Y COMUNICACIÓN

---

Karim Guevara Puente de la Vega

2017

# Mecanismos de sincronización



Exclusión mutua con  
espera activa



SLEEP y WAKEUP



Semáforos



Monitores



Transferencia de  
mensajes



# Introducción

## ❑ Problemas:

- Soluciones a las secciones críticas, proveen espera activa (busy waiting).
- Difícil de generalizarlas para problemas de sincronización más complejos o distintos a la de la SC.

## ❑ Solución:

- Dijkstra [1965]:
  - SC como “Recurso de acceso exclusivo”. El proceso que consigan el permiso lo adquieren, el resto queda a la espera.

# Semáforo

- ❑ Es una variable entera, que contabiliza el número de señales de despertar (wakeup) guardadas.
  - Si es 0, no se tienen señales de despertar.
  - Si es positivo, hay una o más señales de despertar pendientes.
- ❑ Evita la espera activa

# Definición

- ❑ **Mutex** – Es un tipo abstracto de datos (TAD), con las siguientes operaciones:
  - Una **inicialización**
  - **Bajar** - P (down, wait): una generalización de SLEEP.
  - **Subir** - V (up, signal): una generalización de WAKEUP.

# Definición: P (proberen)

- ❑ **S** es la variable entera asociada al semáforo.
- ❑ **P**: verifica el valor de **S**
  - Si **S > 0** → decrementa el valor de **S** (gasta una señal de despertar).
  - Sino → el proceso que lo invoca se pone a **esperar** sin poder completar la operación **P**.
    - No realiza espera activa

```
P(S) : [ while ( S <= 0 ) { esperar } ;  
        S = S - 1; ]
```

# Definición: V (verhogen)

- ❑ **V incrementa** el valor del semáforo direccionado **S**.
  - Si hay varios procesos esperando el semáforo **S** (sin completar **P**), se escoge uno para permitir que complete **P**.
  - Después de una operación **V** en un semáforo que tiene procesos esperando, el semáforo seguirá en 0.
    - Habrá menos procesos esperando completar **P**

$V(S) : [ S = S + 1; ]$

# Importante

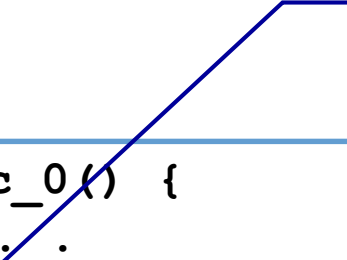
- **P** y **V** corresponde a los protocolos de acceso y salida de la región crítica.
- Las operaciones de verificación, modificación, y la de dormirse, así como la de incremento y de despertar un proceso son indivisibles: **atómicas**.



# Inicialización del semáforo

- ❑ Semáforo tiene un contador/variable interna.
- ❑ Su valor inicial indica el número máximo de procesos que pueden utilizar el recurso asociado simultáneamente.
- ❑ Si el recurso es SC, se requiere un semáforo **Binario**
  - Debe ser inicializado en 1.
  - Asegura que sólo un proceso este en SC.
  - Protocolo de entrada: *P(down)*
  - Protocolo de salida: *V(up)*

# Solución a la SC – 2 procesos




```
while ( S <= 0 ) { esperar } ;  
S = S - 1;
```

```
Proc_0() {  
    . . .  
    P(s) ;  
    balance=balance+cantidad;  
    V(s) ;  
    . . .  
}
```

```
semaforo s=1;  
fork ( proc_0);  
fork ( proc_1);
```

```
Proc_1() {  
    . . .  
    P(s) ;  
    balance=balance-cantidad;  
    V(s) ;  
    . . .  
}
```



```
S = S + 1;
```

# Solución SC – N procesos

## ❑ Semáforo MUTEX

Dato compartido:

```
semaforo mutex; //inicialmente mutex=1
```

Proceso  $P_i$ :

```
do {
```

```
    P(mutex) ;
```

```
    sección crítica
```

```
    V(mutex) ;
```

```
    sección no crítica
```

```
}while(1)
```

# Algunos problemas: Mutex

```
proc_0( ) {  
    ...  
    P(s);  
    balance=balance+cantidad;  
    V(s);  
    ...  
}
```

```
semaforo s=0;  
fork(proc_0);  
fork(proc_1);
```

```
proc_1( ) {  
    ...  
    P(s);  
    balance=balance-cantidad;  
    V(s);  
    ...  
}
```

- Inicializar el semáforo equivocadamente
  - **Abrazo mortal a la entrada de la SC**

# Algunos problemas: Mutex

```
proc_0( ) {  
    ...  
    P(s) ;  
    balance=balance+cantidad;  
    V(s) ;  
    ...  
}
```

```
semaforo s=1;  
fork(proc_0);  
fork(proc_1);
```

```
proc_1( ) {  
    ...  
    V(s) ;  
    balance=balance-cantidad;  
    P(s) ;  
    ...  
}
```

- Intercambiar **P** y **V**
  - No hay exclusión mutua

# Algunos problemas: Mutex

```
proc_0( ) {  
    ...  
    P(mutex1);  
    <borrar elemento>;  
    <computo intermedio>;  
    P(mutex2);  
    <actualizar longitud>  
    V(mutex1);  
    V(mutex2);  
    ...  
}  
  
semaforo mutex1=mutex2=1;  
fork(proc_0);  
fork(proc_1);
```

```
proc_1( ) {  
    ...  
    P(mutex2);  
    <actualizar longitud>  
    <computo intermedio>;  
    P(mutex1);  
    <añadir elemento>;  
    V(mutex2);  
    V(mutex1) ...  
}
```

- Anidamiento inadecuado del semáforo
  - **Abrazo mortal**

# Semáforos y la sincronización

- ❑ Los semáforos sirven de una forma general para lograr la sincronización de procesos.
- ❑ Garantizar que ciertas sucesiones de eventos ocurran o no ocurran
  - Usar **B** en  $P_j$  sólo después que se use **A** en  $P_i$

```
Proceso_0( ) {  
    ...  
    < Uso Recurso_A >;  
    V(flag);  
    ...  
}  
semaforo flag=0;
```

```
Proceso_1( ) {  
    ...  
    P(flag);  
    < Uso Recurso_B >;  
    ...  
}
```

# Semáforos y la sincronización

- ❑ **Proceso\_B** no debería ejecutar *leer()* hasta que el **Proceso\_A** no acabe de *escribir()* sobre la variable **x**, y del mismo modo para la variable **y**.

```
shared double x,y:
```

```
Proceso_A( ) {  
    while (TRUE) {  
        <computo A1>;  
        escribir(x);  
        <computo A2>;  
        leer(y);  
    }  
}
```

```
Proceso_B( ) {  
    while (TRUE) {  
        leer(x);  
        <computo B1>;  
        escribir(y);  
        <computo A2>;  
    }  
}
```



# Semáforos con valor inicial > 1

- Estos semáforos pueden ser utilizados para contabilizar recursos y/o eventos
  - Sistema en el cual se puede permitir acceso simultáneo a recursos a lo más a N procesos

```
shared semaforo mutex = N;
```

```
Pi:
```

```
...
```

```
P(mutex);
```

```
uso del recurso;
```

```
V(mutex);
```

```
...
```

# Productor -Consumidor

```
Productor( ) {  
    int item,pos;  
    while(TRUE) {  
        producir_Item(item);  
        P(vacio);  
        P(mutex);  
        pos=obtener(vacio);  
        V(mutex);  
        copiarBufer(item,pos);  
        V(lleno);  
    }  
}
```

```
Consumidor( ) {  
    int item,pos;  
    while(TRUE) {  
        P(lleno);  
        P(mutex);  
        pos=obtener(lleno);  
        V(mutex);  
        copiarBufer(pos,item);  
        V(vacio);  
        consumir_item(item);  
    }  
}
```

```
semaforo mutex =1, lleno = 0, vacio = N;  
int bufer[N];  
fork(productor); fork(consumidor);
```

# Implementación de semáforos

- **Usando interrupciones**- deshabilita las interrupciones por un corto periodo de tiempo (operaciones **P** y **V**, no para entrar a la SC)

```
class semaforo {  
    int valor;  
    stack bloqueados;  
public:  
    semaforo(int v=1) {  
        valor=v;  
    }  
    P();  
    V();  
};
```

```
P() {  
    deshabilitarInterrupciones();  
    while (valor<=0) {  
        bloqueados.poner();  
        habilitarInterrupciones();  
        deshabilitarInterrupciones();  
    }  
    valor--;  
    habilitarInterrupciones();  
}
```

```
V() {  
    deshabilitarInterrupciones();  
    valor++;  
    bloqueados.sacar();  
    habilitarInterrupciones();  
}
```

# PROBLEMAS DE SINCRONIZACIÓN

---

# Lectores & Escritores

- ❑ Modela el acceso a un recurso compartido por varios procesos concurrentes.
  - Lectores: consultan.
  - Escritores: modifican.
- ❑ Puede haber varios lectores simultáneamente.
- ❑ Si un proceso está actualizando en la base de datos, ningún otro podrá tener acceso a ella, ni siquiera los lectores.

# Lectores & Escritores v.1

```
Lector() {  
    while ( TRUE ) {  
        ...  
        P(mutex);  
        Lectores ++ ;  
        if (Lectores == 1)  
            P(bloqueoEscribir);  
        V(mutex);  
        acceder (recurso);  
        P(mutex);  
        Lectores - - ;  
        if (Lectores == 0)  
            V(bloqueoEscribir);  
        V(mutex);  
    }  
}
```

```
Escritor() {  
    while( TRUE ) {  
        ...  
        P(bloqueoEscribir);  
        acceder (recurso);  
        V (bloqueoEscribir);  
    }  
}
```

```
tipoRecurso *recurso;  
int Lectores = 0;  
semaforo mutex = 1;  
semaforo bloqueoEscribir = 1;  
  
fork(lector);  
fork(escritor);
```

# Lectores & Escritores v.2

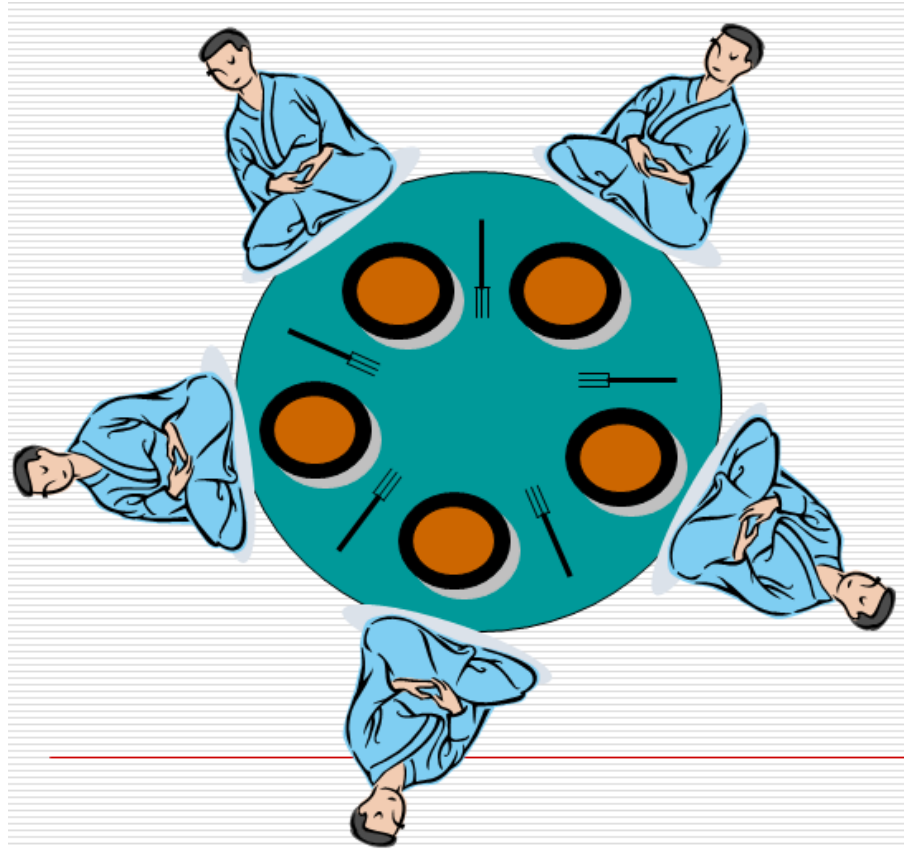
```
Lector() {  
    while( TRUE ) {  
        ...  
        P(bloqueoLeer);  
        P(mutex1);  
        Lectores ++ ;  
        if(Lectores == 1)  
            P (bloqueoEscribir);  
        V(mutex1);  
        V(bloqueoLeer);  
        acceder (recurso);  
        P(mutex1);  
        nLectores--;  
        if(nLectores == 0)  
            V(bloqueoEscribir);  
        V(mutex1);  
    }  
}
```

```
Escritor() {  
    while( TRUE ) {  
        ...  
        P(mutex2);  
        nEscritores++;  
        if(nEscritores == 1)  
            P(bloqueoLeer);  
        V(mutex2);  
        P(bloqueoEscribir);  
        acceder (recurso);  
        V(bloqueoEscribir);  
        P(mutex2);  
        nEscritores--;  
        if(nEscritores == 0)  
            V(bloqueoLeer);  
        V(mutex2);  
    }  
}
```

```
tipoRecurso *recurso;  
int nLectores=0, nEscritores=0;  
semaforo mutex1=1, mutex2=1;  
semaforo bloqueoLeer=1, bloqueoEscribir=1;
```

# Cena de los filósofos

- ❑ Periodos alternantes de **comer** y **pensar**
- ❑ Si tiene hambre, trata de adquirir sus tenedores, **uno a la vez**.
- ❑ Si logra adquirirlos, come por un rato, y luego seguirá pensando.





# Cena de los filósofos

```
#define N 5
void filosofo(int i) {
    while(TRUE) {
        pensar();
        tomar_tenedor (i);
        tomar_tenedor ((i+1) % N);
        comer();
        dejar_tenedor (i);
        dejar_tenedor ((i+1) % N);
    }
}
```

```
#define N 5
void filosofo (int i) {
    while (TRUE) {
        pensar();
        P(mutex);
        tomar_tenedor (i);
        tomar_tenedor ((i+1) % N);
        comer();
        dejar_tenedor (i);
        dejar_tenedor ((i+1) % N);
        V(mutex);
    }
}

semaforo mutex=1;
```

# Cena de los filósofos

```
#define N 5
#define IZQ (i+N-1)%N
#define DER (i+1)%N
#define PENSANDO 0
#define HAMBRE 1
#define COMIENDO 2
int estado[N];
semaforo mutex=1;
semaforo s[N];
//inicializados en 0
```

```
void filosofo(int i){
    while(TRUE) {
        pensar();
        tomar_tenedor (i);
        comer();
        dejar_tenedor (i);
    }
}
```

```
void dejar_tenedor(int i){
    P(mutex);
    estado[i]=PENSANDO;
    probar(IZQ);
    probar(DER);
    V(mutex);
}
```

```
void tomar_tenedor(int i){
    P(mutex);
    estado[i]=HAMBRE;
    probar(i);
    V(mutex);
    P(s[i]);
}
```

```
void probar(i){
    if (estado[i]==HAMBRE && estado[IZQ]!=COMIENDO && estado[DER]!=COMIENDO){
        estado[i]=COMIENDO;
        V(s[i]);
    }
}
```

# Conclusión

- ❑ Ya se tiene una solución a la espera activa
- ❑ Pero...
  - Es responsabilidad del programador el uso adecuado de las primitivas de sincronización:
    - Bajar para entrar a la sección crítica
    - Subir para salir de ella
  - El sistema falla:
    - Si inadvertidamente no se sigue estrictamente el protocolos de entrada y salida de la región o
    - No se inicializa adecuadamente el semáforo

# Mecanismos de sincronización



Exclusión mutua con  
espera activa



SLEEP y WAKEUP



Semáforos



Monitores



Transferencia de  
mensajes



# Introducción

## ❑ Problema:

- Semáforos, están sujetas a errores, como la omisión o mala ubicación de una operación P o V.

## ❑ Solución:

- 1974 (Hoore) y 1975 (B.Hanse) :
  - Proponen una primitiva de sincronización de más alto nivel:  
MONITOR

# Monitor

- ❑ **Módulo Monitor-** Colección de procedimientos, variables y estructuras de datos, agrupados en una especie de módulo.
  - **Monitor-** TDA para el cual un proceso/hilo puede estar ejecutando cualquiera de sus procedimientos miembro en cualquier momento.
  - Los únicos que acceden a las estructuras internas del monitor son los procedimientos declarados en el monitor.
  - La ejecución de la función miembro se trata como una sección crítica

# Monitor

- ❑ Procesos
  - Invocan a proced. del monitor.
  - No pueden acceder directamente a las variables del monitor
- ❑ Procedimientos del monitor:
  - No pueden acceder a variables externas
  - Pero si a las permanentes (i y c), a las locales al procedimiento, y a los argumentos del procedimiento (x).
- ❑ Inicializar las variables permanentes del monitor antes que ningún procedimiento sea ejecutado.

```
monitor ejemplo {  
private:  
    int i = 0;  
    condition c ;  
public:  
    productor (x) {  
        ...  
        ...  
    };  
    consumer (x) {;  
        ...  
        ...  
    };  
};
```

# Implementación de la Exclusión Mutua

**“Solamente un proceso puede estar activo a la vez dentro de un monitor”.**

- ❑ Los monitores son una construcción de los lenguajes de programación
  - Los compiladores saben que las llamadas a los procedimientos de un monitor se manejan de forma distinta a las llamadas a los procedimientos convencionales.
- ❑ La implementación de la exclusión mutua en las entradas del monitor es labor del compilador



# Implementación de la Exclusión Mutua



```
monitor unTAD {  
  private:  
    semaforo mutex = 1;  
    < estructuras de datos del TAD >  
    ...  
  public:  
    proced_i(...) {  
      P(mutex);  
      <procesamiento para proced_i>  
      V(mutex);  
    };  
    . . .  
}
```



```
monitor balCompartido {  
private:  
    int balance;  
public:  
    aportar(int cantidad){  
        balance=balance+cantidad;  
    }  
    retirar(int cantidad){  
        balance=balance-cantidad;  
    }  
};
```

```
Proc_0(){  
    ...  
    balCompartido.aportar(cantidad;  
    ...  
}  
  
Proc_1() {  
    ...  
    balCompartido.retirar(cantidad);  
    ...  
}  
  
fork(proc_0);  
fork(proc_1);
```

# Variables de condición

- ❑ Si un proceso descubre que no puede seguir, hasta que otro proceso cambie el estado de la información protegida por el monitor.....

```
Monitor lectorEscritor_1 {  
    int nLectores = 0;  
    int nEscritores = 0;  
    boolean ocupado = FALSE;  
public:  
    iniciarLectura() {  
        while (nEscritores != 0) ;  
        nLectores ++;  
    }  
    finalizarLectura() {  
        nLectores -- ;  
    }  
    ...  
}
```

```
Monitor lectorEscritor_1 {  
    ...  
public:  
    ...  
    iniciarEscritura () {  
        nEscritores ++;  
        while(ocupado || nLectores >0);  
        ocupado = TRUE;  
    }  
    finalizarEscritura () {  
        nEscritores--;  
        ocupado = FALSE;  
    }  
}
```

# Variables de condición

- ❑ Solución: permitir que el proceso en espera ceda temporalmente el monitor, **se bloquee**
- ❑ Se incluyen las **variables de condición**:
  - Estructura de datos, global a todos los procedimientos del monitor.
  - Puede ser modificado por:
    - **wait()**: proceso invocador se suspende
    - **signal()**: reanuda exactamente un proceso si hay uno suspendido actualmente
    - **queue()**: es TRUE si hay un hilo/proceso suspendido sobre la variable de condición, y FALSE en otro caso

# Variables de condición

- ❑ No son contadores, no acumulan señales para su uso futuro (semáforos).
- ❑ Si una variable de condición recibe una señal y ningún proceso está esperando esta variable, la señal se perderá (wait antes de signal).
- ❑ ¿Qué sucede después de signal?
  - **Hoare**: si P1 esta esperando la señal, y P0 la ejecuta, éste último cede el monitor a P1 para continuar su ejecución.
  - **Hansen**: P0 ejecuta la señal y continua ejecutándose, luego P1 intenta ejecutarse comprobando la condición.
    - Menos cambios de contexto.

# Lectores & Escritores

```
monitor lectorEscritor_2 {  
    int nLectores = 0;  
    boolean ocupado = FALSE;  
    condition okLeer,  
        okEscribir;  
public:  
    iniciaLectura( ){  
        if (ocupado || (okEscribir.queue()))  
            okLeer.wait( );  
        nLectores++;  
        okLeer.signal( );  
    }  
    finalizaLectura( ){  
        nLectores-- ;  
        if (nLectores == 0)  
            okEscribir.signal( );  
    }  
    ...  
};
```

```
...  
iniciaEscritura(){  
    if (nLectores!=0 || ocupado)  
        okEscribir.wait()  
        ocupado = TRUE;  
}  
finalizaEscritura(){  
    ocupado = FALSE;  
    if (okEscribir.queue())  
        okEscribir.signal();  
    else  
        okLeer.signal();  
}  
};
```

# Lectores & Escritores

```
monitor lectorEscritor_2 {  
    int nLectores = 0;  
    boolean ocupado = FALSE;  
    condition okLeer, okEscribir;  
public:  
    iniciaLectura( ){  
        if (ocupado || (okEscribir.queue()))  
            okLeer.wait( );  
        nLectores++;  
        okLeer.signal( );  
    }  
    finalizaLectura( ){  
        nLectores-- ;  
        if (nLectores == 0)  
            okEscribir.signal( );  
    }  
    ...  
};
```

```
Proceso Lector(){  
    while(TRUE){  
        ...  
        lectorEscritor_2.iniciaLectura();  
        acceder_recurso();  
        lectorEscritor_2.finalizaLectura();  
        ...  
    }  
}
```

# Lectores & Escritores

```
monitor lectorEscritor_2 {  
    ...  
public:  
    iniciaEscritura(){  
        if (nLectores!=0 || ocupado)  
            okEscribir.wait()  
        ocupado = TRUE;  
    }  
    finalizaEscritura(){  
        ocupado = FALSE;  
        if (okEscribir.queue())  
            okEscribir.signal();  
        else  
            okLeer.signal();  
    }  
};
```

```
Proceso Escritor(){  
    while(TRUE ){  
        ...  
        lectorEscritor_2.iniciaEscritura();  
        acceder_recurso();  
        lectorEscritor_2.finalizaEscritura();  
        ...  
    }  
}
```



# Ventajas vs. desventajas

- + Monitores hacen que la programación paralela sea menos propensa a errores que los semáforos.
- + Exclusión mutua es automática.
- Monitores son concepto del lenguaje de programación.
  - El compilador debe de reconocerlos y gestionar la exclusión mutua.

# Mecanismos de sincronización



Exclusión mutua con  
espera activa



SLEEP y WAKEUP



Semáforos



Monitores



Transferencia de  
mensajes



# Transferencia de mensajes

- ❑ Investigar acerca de la transferencia de mensajes entre procesos como mecanismo de comunicación y sincronización