# Sistemas Operativos
# Sincronización de procesos

Amilcar Meneses Viveros
ameneses@computacion.cs.cinvestav.mx

# Presentación

- Introducción
- Problema de sección crítica
- Hardware de sincronización
- Semáforos
- Problemas clásicos de sincronización
- Regiones críticas
- Monitores

# Introducción

- Problemas con procesos cooperativos puede establecerse a través de hilos o procesos pesados
- El problema productor-consumidor puede no ejecutarse correctamente

  Problema: variables compartidas

# Problema de sección crítica

- Región de código que hace referencia, o accesa, a variables compartidas
- *n* procesos compiten por utilizar datos compartidos
- La sección crítica DEBE ser mútuamente excluyente
- Un proceso debe solicitar permiso para ingresar a la sección crítica

# Problema de sección crítica

- La solución a la sección crítica debe cumplir:
  - Exclusión mutua
  - Progreso
  - Espera limitada

```
while(TRUE)
{

 sección de ingreso
   sección crítica
 sección de salida
   sección restante
}
```

# Problema de sección crítica
## Soluciones para dos procesos

```
while(TRUE)
{

   while (turno != i) ;

     sección crítica

   turno = j;

     sección restante

}
```

Algoritmo 1

# Problema de sección crítica
## Soluciones para dos procesos

```
int indicador[2];
indicador[0]=indicador[1]=FALSE;
while(TRUE)
{
   indicador[i]=TRUE;
   while (indicador[j]) ;

     sección crítica

   indicador[i]=FALSE;

     sección restante

}
```

Algoritmo 2

# Problema de sección crítica
## Soluciones para dos procesos

```
int indicador[2];
while(TRUE)
{
   indicador[i]=TRUE;
   turno=j;
   while (indicador[j] &&
          turno==j) ;

     sección crítica

   indicador[i]=FALSE;

     sección restante
}
```

Algoritmo 3

# Problema de sección crítica
## Soluciones para múltiples procesos

```
int numero[N];  // inicializado a 0
BOOL turno[N];  // inicializado a FALSE
while(TRUE)
{
   turno[i]=TRUE;
   numero[i]=MAX(numero,N)+1;
   turno[i]=FALSE;
   for (j=0; j<N; j++) {
      while(turno[j]==TRUE);
      while(numero[j]!=0 &&
      EV(numero[j],j)<EV(numero[i],i));
   }

     sección crítica

   indicador[i]=0;

     sección restante
}
```

Algoritmo 4: (bakery Algorithm)

# Hardware de sincronización

- Garantizar la ejecución atómica cuando se modifica una variable compartida (no es factible en entornos multiprocesadores)
- El HW garantiza la ejecución atómica de las secciones de ingreso y de egreso

# Hardware de sincronización

```
BOOL evalua_y_asigna(BOOL &objetivo) {
      BOOL rv=target;
      target=TRUE;
      return rv;
}
```

Función `evalua_y_asigna` atómica

# Hardware de sincronización

```
while(TRUE)
{

   while (evalua_y_asigna(cerradura)) ;

     sección crítica

   cerradura = false;

     sección restante

}
```

# Hardware de sincronización

```
void swap(BOOL &a, BOOL &b) {
      BOOL tmp=a;
      a=b;
      b=tmp;
}
```

Función *swap* (intercambio) atómica

# Hardware de sincronización

```
BOOL lock=TRUE;  // dato compartido


while(TRUE)
{

   key = TRUE;
   while (key==TRUE) swap(lock,key);

     sección crítica

   lock = FALSE;

     sección restante

}
```

# Semáforos

- Herramienta de sincronización que no requiere de una espera "de alto costo"
- Semaforo S - variable entera
- Dos operaciones atómicas asociadas
  - wait(S) o down(S):
  -
    ```
    while (S<=0) no_op();
    S--;
    ```
  - signal(S) o up(S):

    ```
    S++;
    ```

# Semáforos

```
Semaforo mutex=1;  // dato compartido


while(TRUE)
{

   wait(mutex);

     sección crítica

   signal(mutex);

     sección restante

}
```

# Implantación de Semáforos

- Definir la estructura

```
typedef struct _semaforo {
   int valor;
   struct proceso *L;
} semaforo;
```

- Asumir dos operaciones

1. `block`
2. `wakeup(P)`

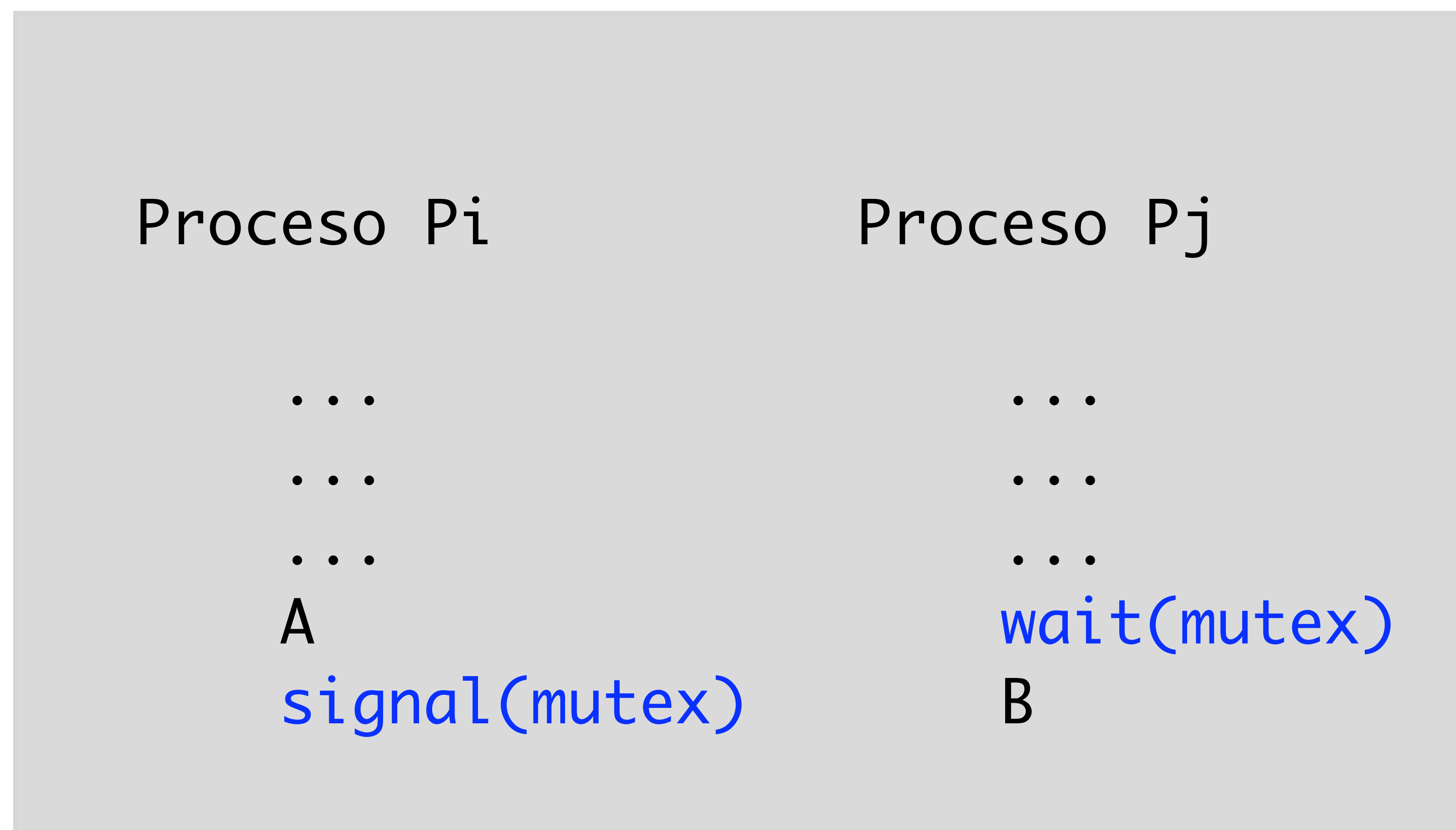
# Implantación de Semáforos

- Operaciones del semáforo quedan como:

```
wait(semaforo &S) {
   S.valor--;
   if (S.valor<0) {
      agrega_proceso a S.L;
      block;
   }
}

signal(semaforo &S) {
   S.valor++;
    if (S.valor<=0) {
      remueve un proceso P de S.L;
      wakeup(S);
   }
}
```

# Semáforos como herramienta de sincronización




Ejecutar la sentencia B en Pj, después de ejecutar la sentencia A en Pi

# Problema:
# Candado mortal

```
Proceso Pi          Proceso Pj

 wait(A);            wait(B);
 wait(B);            wait(A);
 ...                 ...
 ...                 ...
 signal(B);          signal(A);
 signal(A);          signal(B);
```

# Problemas clásicos de sincronización

- Productor - consumidor
- Escritores y lectores
- Filósofos pensantes

# Problema productor-consumidor

- Un proceso se encarga de generar datos en un buffer compartido
- Una serie de procesos consumidores toman datos del buffer compartido

# Problema productor-consumidor

```
// buffer compartido
element buffer[N];

// semaforos, tambien compartidos
semaforo vacio, lleno, mutex;

// inicializacion de semaforos
mutex.valor = 1;
vacio.valor = N;
lleno.valor = 0;
```

# Problema productor-consumidor

```
element sigp; // variable local
while(TRUE) {
  sigp = produce_nuevo_elemento();

  wait(vacios);
  wait(mutex);
    agrega_buffer(buffer, sigp);
  signal(mutex);
  signal(lleno);
}
```

Proceso productor

# Problema productor-consumidor

```
element sigc; // variable local
while(TRUE) {
  wait(lleno);
  wait(mutex);
    sigc=toma_elemento(buffer);
  signal(mutex);
  signal(vacio);

  consume_elemento(sigc);
}
```

Proceso consumidor

# Problema lectores-escritores

- Varios procesos comparten un dato
- Algunos procesos modifican el dato
- Algunos procesos leen el dato

# Problema lectores-escritores

```
// dato compartido
dato dato_compartido;

// semaforos, tambien compartidos
semaforo mutex, s_dato;

// contador de procesos lectores
int nlectores = 0;

// inicializacion de semaforos
mutex.valor  = 1;
s_dato.valor = 1;
```

# Problema lectores-escritores

```
dato nuevo_valor; // variable local

while(TRUE) {
  nuevo_valor = genera_nuevo_valor();

  wait(s_dato);
   dato_compartido = nuevo_valor;
  signal(s_dato);
}
```

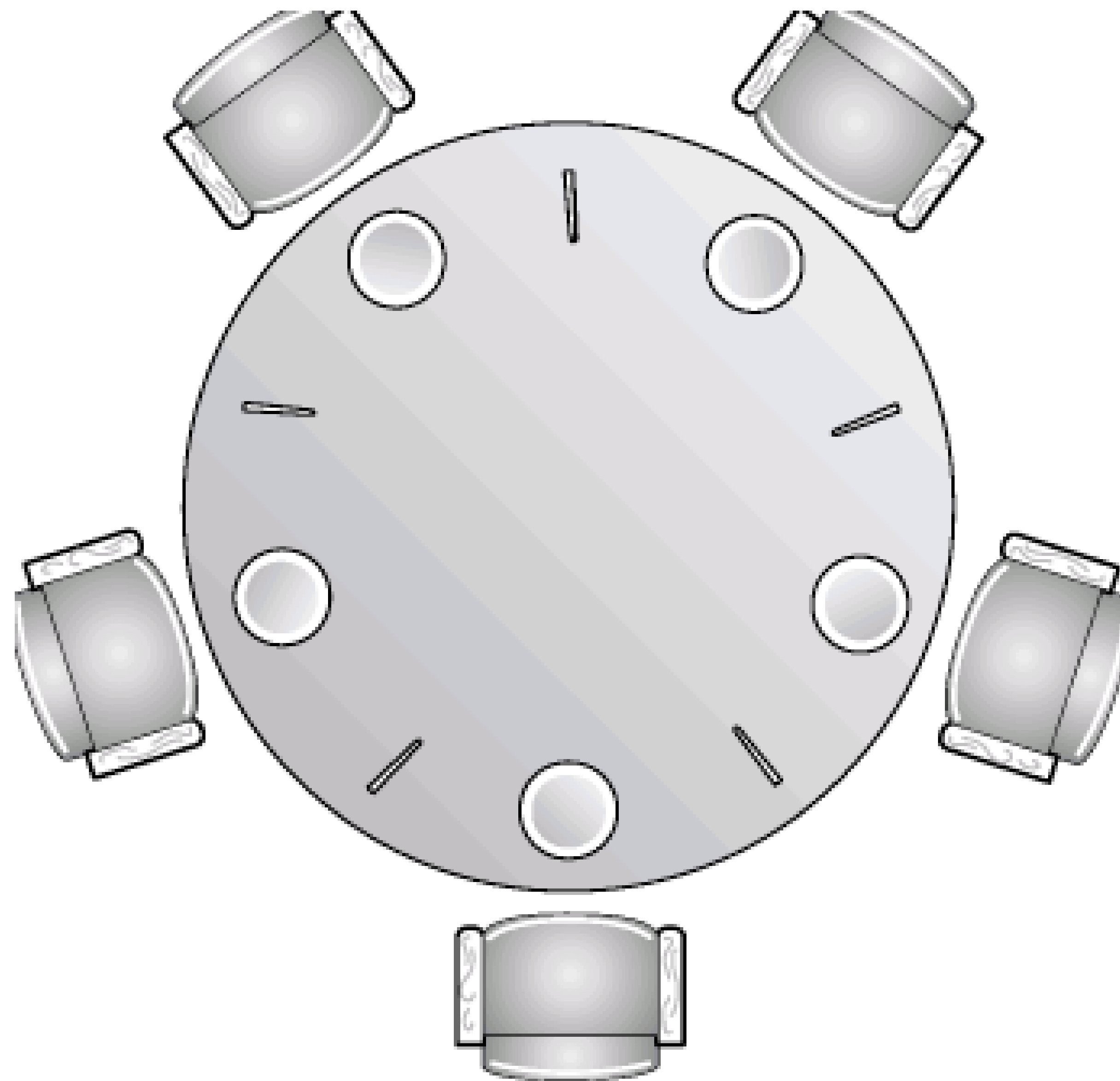
Proceso escritor

# Problema lectores-escritores

```
while(TRUE) {
  wait(mutex);
  nlectores=nlectores+1;
  if (nlectores==1) wait(s_dato);
  signal(mutex);
  lee_dato_compartido();
  wait(mutex);
  nlectores=nlectores-1;
  if (nlectores==0) signal(s_dato);
  signal(mutex);
  usa_dato();
}
```

Proceso lector

# Problema filósofos pensantes



# Problema filósofos pensantes

```
semaforo palillo[5];

while(TRUE) {
  wait(palillo[i]);
  wait(palillo[(i+1)%5]);
   comer();
  signal(palillo[i]);
  signal(palillo[(i+1)%5]);
   pensar();
}
```

Proceso escritor

# Regiones críticas

- Semaforos: exclusión mutua explícita
- Mal uso de semáforos (codificación):

1. inversion de wait(), signal()
2. repeticion de wait()
3. olvido de una operacion wait(), signal()

# Monitores

- Exclusión mutua implícita
- Declaración de un conjunto de funciones dentro de una región que garantiza la exclusión mútua

# Monitores

```
type nombre-monitor monitor {
  declaración de variables
  void funcion1();
  void funcion2();
  ...
   {
        código de inicialización
    }
}
```