Sistemas Operativos

Escuela de Ingeniería Civil Informática

- Administración de Procesos
  Sincronización con Monitores

UNIVERSIDAD DEL BÍO-BÍO

MONITORES

- Los semáforos son una herramienta general y eficiente para sincronizar procesos, pero siguen siendo de bajo nivel.

- Las soluciones mediante semáforos no son ni muy limpias ni muy claras, y siempre están sujetas a errores como la omisión o mala ubicación de una operación wait o signal.

- Para facilitar la escritura de programas correctos, **Brinch Hansen** propuso una primitiva de sincronización de alto nivel, llamada **monitor**.

- Los monitores son una herramienta de sincronización más estructurada que encapsula variables compartidas junto con los procedimientos para acceder a ellas.

- Para modificar las estructuras de datos internas del monitor es preciso utilizar los procedimientos definidos en él, no hay acceso directo a estas estructuras desde fuera del monitor.

MONITORES ESTILO HANSEN

- Los monitores originales de Java usan este mismo estilo y nSystem los ofrece con la siguiente API:

```
nMonitor nMakeMonitor( );
void nEnter(nMonitor m);
void nExit(nMonitor m);
void nWait(nMonitor m);
void nNotifyAll(nMonitor m);
void nNotify(nMonitor m);
void nDestroyMonitor(nMonitor m);
```

PRODUCTOR/CONSUMIDOR CON MONITORES

- La siguiente son 2 soluciones al problema del Productor/Consumidor.
- La solución con monitores es casi una transformación directa y mecánica desde la solución incorrecta.

Solución incorrecta	*Solución correcta con monitores*

`Item buf[N];`
`int nextempty= 0, nextfull= 0;`
`int count= 0;`
	`Item buf[N];`
	`int nextempty= 0, nextfull= 0;`
	`int count= 0;`
	`nMonitor m;` `/* = nMakeMonitor() */`

`Item get() {`
`  Item x;`
 	
`  while (count==0)`
`  ; /* busy-waiting */`
`  x= buf[nextfull];`
`  nextfull= (nextfull+1)%N;`
`  count--;`
 	
`}`
	`Item get() {`
	`  Item x;`
	`  nEnter(m);`
	`  while (count==0)`
	`  nWait(m);`
	`  x= buf[nextfull];`
	`  nextfull= (nextfull+1)%N;`
	`  count--;`
	`  nNotifyAll(m);`
	`  nExit(m);`
	`}`

PRODUCTOR/CONSUMIDOR CON MONITORES

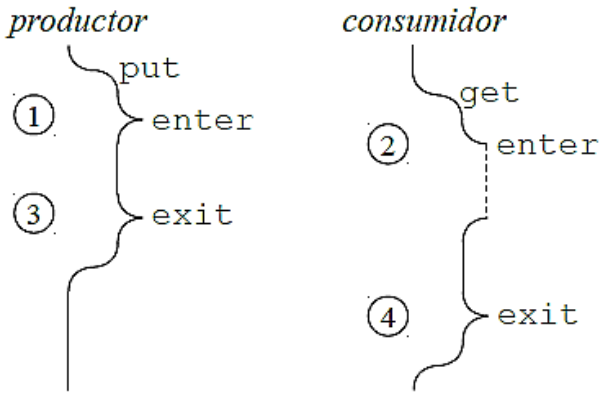
```
void put(Item x) {

  while (count==N)
    ; /* busy-waiting */
  buf[nextempty]= x;
  nextempty= (nextempty+1)%N;
  count++;


}
```

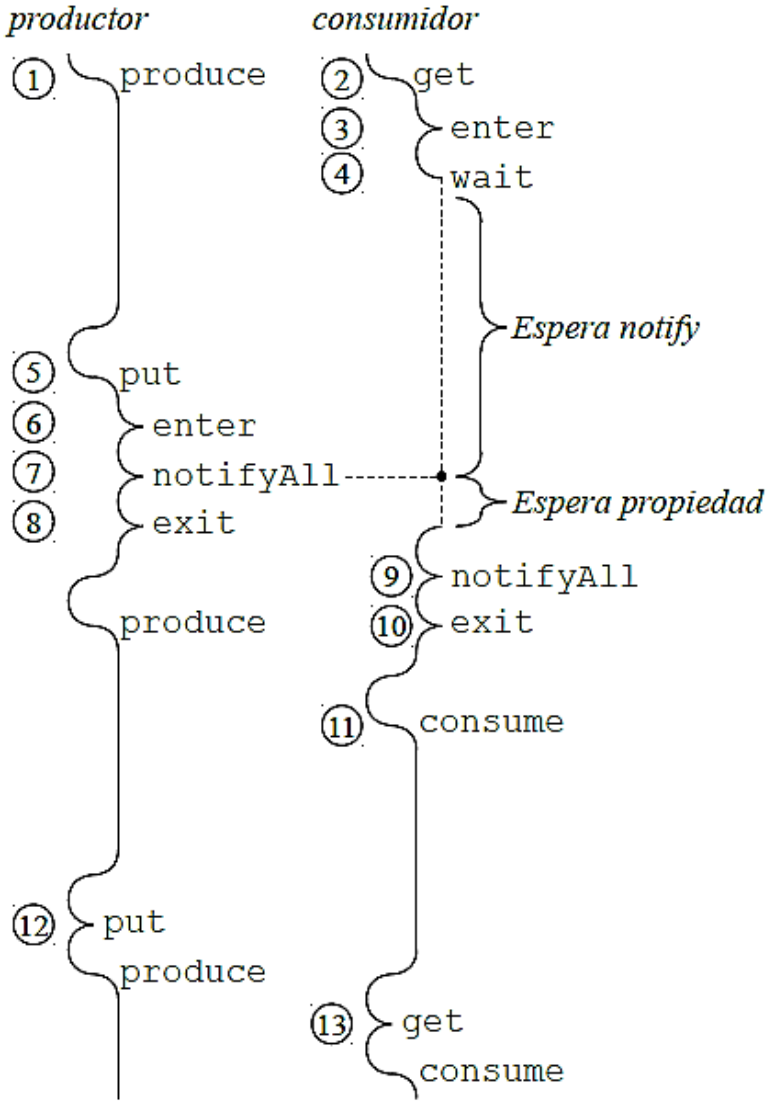
```
void put(Item x) {
  nEnter(m);
  while (count==N)
    nWait(m);
  buf[nextempty]= x;
  nextempty= (nextempty+1)%N;
  count++;
  nNotifyAll(m);
  nExit(m);
}
```

- El funcionamiento de nEnter y nExit se muestra en el siguiente diagrama:



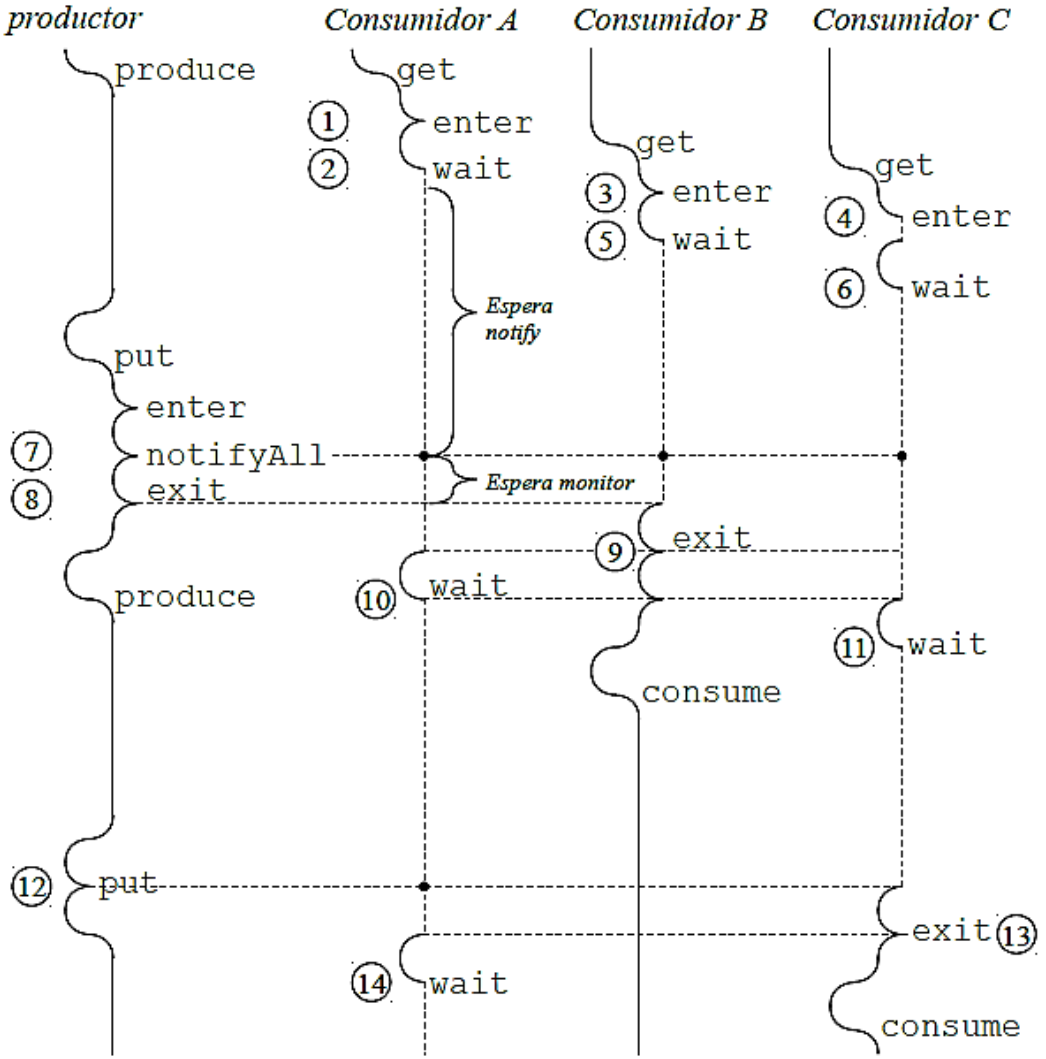
PRODUCTOR/CONSUMIDOR CON MONITORES

- Este diagrama muestra lo que ocurre cuando el buffer está vacío.



PRODUCTOR/CONSUMIDOR CON MONITORES

- Los monitores no especifican el orden en que se despiertan los threads después de una notificación.
- Esto funciona también así en los monitores de Java y lo pthreads de linux.
- Se debe utilizar while en vez de if en una llamada a nWait.
- El siguiente diagrama presenta esta situación.



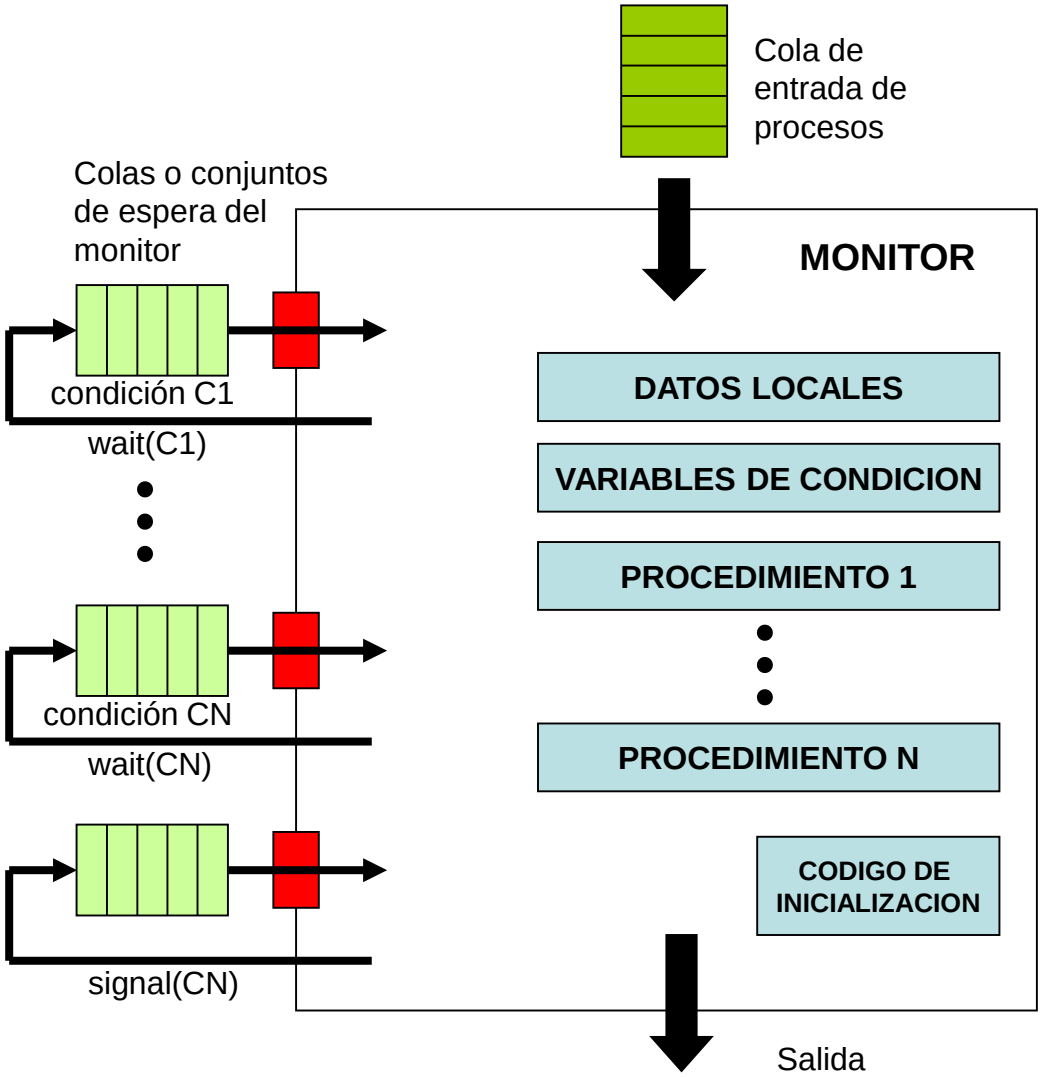
MONITORES ESTILO HOARE

- Para optimizar la solución del problema del Productor/Consumidor se debe utilizar los monitores creados por **C.A.R. Hoare**.

- Hoare introduce un nuevo tipo de datos: las condiciones, cuya API en nSystem es la siguiente:

```
nCondition nMakeCondition(nMonitor mon);
void nDestroyCondition(nCondition cond);
void nWaitCondition(nCondition cond);
void nSignalCondition(nCondition cond);
```

ESTRUCTURA DE UN MONITOR HOARE



Se asegura la exclusión mutua dentro del monitor

SOLUCIÓN EFICIENTE PROD./CONS.

- Solución eficiente al problema del Productor/Consumidor:

```
Item buf[N];
int nextempty= 0, nextfull= 0;
int count= 0;
nMonitor m;          /* = nMakeMonitor() */
nCondition no_empty; /* = nMakeCondition(m); */
nCondition no_full;  /* = nMakeCondition(m); */
```

```
Item get() {
  Item x;
  nEnter(m);
  while (count==0)
    nWaitCondition(no_empty); /*A */
  x= buf[nextfull];
  nextfull= (nextfull+1)%N;
  count--;
  nSignalCondition(no_full); /*B */
  nExit(m);
}
```

Sección crítica

SOLUCIÓN EFICIENTE PROD./CONS.

```
void put(Item x) {
  nEnter(m);
  while (count==N)
    nWaitCondition(no_full);   /* C */
  buf[nextempty]= x;
  nextempty= (nextempty+1)%N;
  count++;
  nSignalCondition(no_empty); /* D */
  nExit(m);
}
```

Sección crítica

PATRÓN DE USO DE MONITORES

- La mayoría de los problemas de sincronización se resuelve con el siguiente patrón:

```
nEnter(m);
while ( … )
  nWait(m);
... consulta/actualización de la sincronización ...
nNotifyAll(m);
nExit(m);
```