Sistemas Operativos

Escuela de Ingeniería Civil Informática

- Administración de Procesos
  Sincronización con Semáforos

UNIVERSIDAD DEL BÍO-BÍO

RESUMEN LOCKS

- Los Locks permiten eliminar la **espera activa** (o **busy waiting**) en los procesos.
- Las operaciones ***lock()*** y ***unlock()*** son parte del código del sistema operativo (kernel).
- *lock()* se activa cuando el proceso encuentra un código en condición de ocupado, entonces el proceso se bloquea (debe esperar).
- En este caso, el sistema operativo escoge otro proceso para asignarle el procesador.
- *unlock()* únicamente libera el lock y el proceso que lo invoca puede continuar su ejecución.
- Este mecanismo permite definir secciones críticas en un proceso y retorna el control al sistema operativo cuando un proceso no puede continuar.

COOPERACIÓN DE PROCESOS

- Hay ocasiones en que varios procesos trabajan directamente juntos para completar una tarea común.

- Algunos ejemplos famosos son el del productor-consumidor, filósofos comensales y lectores escritores.

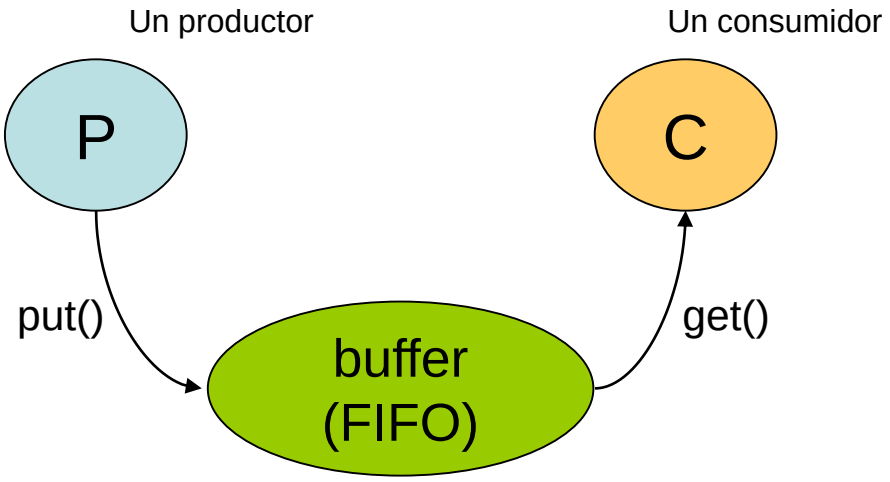
- Cada caso requiere tanto exclusión mutua como sincronización y todos se implementan utilizando un nuevo mecanismo llamado **semáforo**.

PROBLEMA: PRODUCTOR CONSUMIDOR

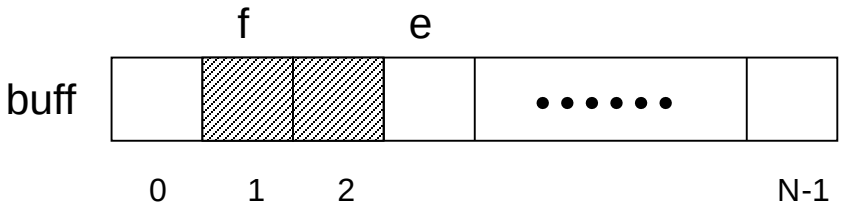
- Se tienen dos procesos: P (produtor) y C (consumidor):
  - Si buffer está lleno → P espera
  - Si buffer está vacío → C espera

P:
```
for(;;) {
  Item it= produce();
  put(it);
}
```

C:
```
for(;;) {
  Item it= get();
  consume(it);
}
```



- Implementación de put() y get()



- f indica cual es el primer casillero lleno
- e indica cual es el primer casillero vacío
- c cantidad de items en buff

PROBLEMA: PRODUCTOR CONSUMIDOR

- Implementación de put() y get():

```
#define N 100
Item buff[N];
int e=0, f=0, c=0;

void put(Item it) {
  while(c>=N);
  buff[e]= it;
  e= (e+1)%N;
  c++;
}

Item get() {
  Item it;
  while(c<=0);
  it= buff[f];
  f= (f+1)%N;
  c--;
  return it;
}
```

Sección crítica

¿Qué problema tiene este código?

¿Funciona con varios productores?

¡¡EVITAR!!

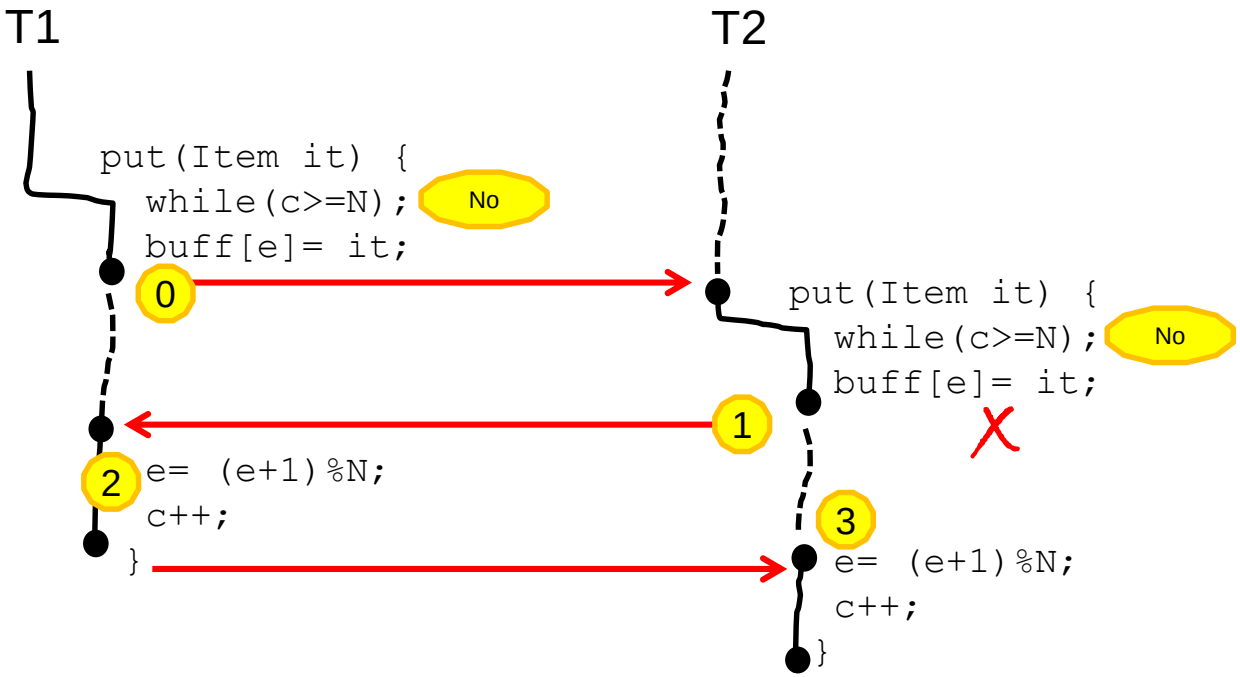
Espera Activa o Busy waiting
El procesador trabaja en algo no muy productivo.

PROBLEMA: PRODUCTOR CONSUMIDOR

- Problema con varios productores:

  Propuesto: Dibuje el diagrama de tareas que muestra el data-race con 2 productores simultáneos.

```
int e=0, f=0, c=0; N=5;
```



SINCRONIZACIÓN CON SEMÁFOROS

- Los semáforos sirven para sincronizar procesos livianos.
- Se puede imaginar un semáforo como un distribuidor de tickets (sino hay tickets disponibles entonces hay que esperar).
- Observación: los locks son semáforos binarios.
- Operaciones:
  - **makeSem(n):** crea un semáforo, tipo **Sem** con n tickets
  - **wait(s):** saca un ticket y espera sino hay disponibles (nunca hay una cantidad negativa).
  - **signal(s):** aporta un ticket al semáforo (nunca espera)
- ¿cómo se implementan secciones críticas con semáforos?

```
S = makeSem(1);
wait(s);
/*sección crítica*/
Signal(s)
```

SOLUCIÓN AL PRODUCTOR CONSUMIDOR

- Utilizar semáforos con contador para solucionar el problema:

```
#define N 100
Item buff[N];
int e=0, f=0;
...
Sem full= makeSem(0);
Sem empty= makeSem(N);
...
void put(Item it) {
  wait(empty);
  buff[e]= it;
  e= (e+1)%N;
  signal(full);
}

Item get() {
  Item it;
  wait(full);
  it= buff[f];
  f= (f+1)%N;
  signal(empty);
  return it;
}
```

Asociado a los datos

Asociado a los casilleros vacíos

Mientras no hay casilleros vacíos, esperar

Aporta un dato (se insertó un dato en el buffer)

Esta solución funciona con un productor y un consumidor simultáneos

¿Funciona con varios productores o consumidores simultáneos?

SOLUCIÓN AL PRODUCTOR CONSUMIDOR

- Productor/Consumidor para varios consumidores o productores simultáneos:

La solución consiste en evitar que dos consumidores saquen el mismo item dos veces y dos productores utilicen la misma casilla simultáneamente.

Usar secciones críticas diferentes para cada función de modo que un productor pueda operar concurrentemente con un consumidor.

```
#define N 100
Item buff[N];
int e=0, f=0, c=0;
...
Sem full= makeSem(0);
Sem empty= makeSem(N);
Sem semP= makeSem(1);
Sem semC= makeSem(1);
...
```

```
void put(Item it) {
  wait(empty);
  wait(semP);
  buff[e]= it;
  e= (e+1)%N;
  signal(semP);
  signal(full);
}

Item get() {
  Item it;
  wait(full);
  wait(semC);
  it= buff[f];
  f= (f+1)%N;
  signal(semC);
  signal(empty);
  return it;
}
```