

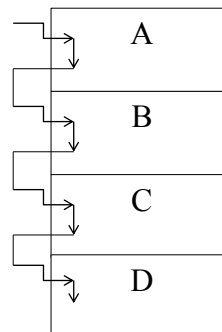
## Tema 2: Procesos e hilos

- Procesos
- Hilos
- Comunicación entre procesos
- Programación concurrente
  - Exclusión mutua con espera activa.
  - Exclusión mutua sin espera activa.
  - Problemas clásicos de comunicación entre procesos.
- 5 Comunicación entre procesos en UNIX (IPC's)

1

## Procesos

- Proceso = programa en ejecución
- Sistema de tiempo compartido:
  - Se ejecuta un proceso unos cuantos mls, luego otro ...
  - pseudoparalelismo
- Multiprogramación
- Velocidad no uniforme
  - No reproducible
- No suposiciones de tiempo



2

## Creación de procesos

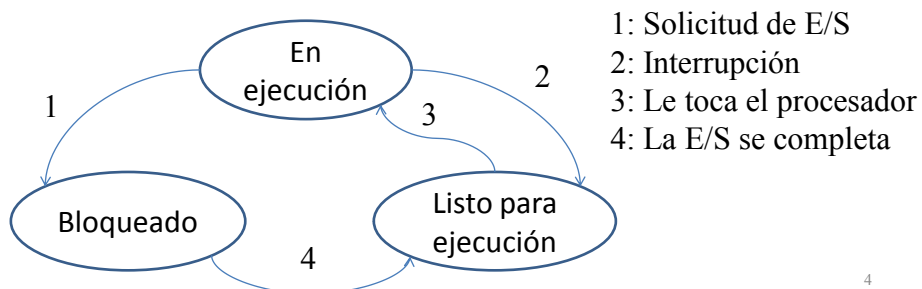
- En todo S.O hay mecanismos para crear procesos
  - FORK
    - Crea una copia exacta del proceso
    - Una vez creado el padre y el hijo se ejecutan en paralelo
  - CALL
    - Carga un fichero binario en memoria y se ejecuta como hijo
    - El padre se suspende hasta que el hijo es ejecutado (no en paralelo)
  - EXEC
    - El proceso cambia su imagen de memoria para pasar a ejecutar otro programa (no se crea proceso)

```
if ( fork() == 0 )
{
  /* proceso hijo */
}
else
{
  /* proceso padre */
}
```

3

## Estado de los procesos

- Necesidad de interacción entre los procesos
  - `cat f1 f2 f3 | grep palabra`
- El proceso se puede bloquear o el S.O puede decidir parar un proceso para ejecutar otro



4

## Planificador (Scheduler)

- Parte del S.O que decide qué proceso se ejecuta.
- Debe decidir:
  - De ejecución a bloqueado (Solicitud de E/S)
  - De ejecución a listo (Interrupción)
  - De bloqueado a listo (La E/S se completa)
  - Cuando termina el proceso en ejecución

5

## Tabla de procesos

- Hay una entrada por cada proceso
- Se indica el estado del proceso (bloqueado/listo/en ejecución)
- Toda la información para poder pasar de listo a ejecución
  - Contexto
- Cambio de contexto
  - Restauración de todo el contexto para poder seguir la ejecución

6

## Funcionamiento del planificador ligado a las interrupciones

- Llega una interrupción del disco
  - correspondiente a una operación de E/S de P1
  - mientras se está ejecutando P3
- El HW
  - salva automáticamente el PC y otros datos en la pila
  - Carga el nuevo PC para realizar el salto a la dir de memoria en el vector de interrupciones
- El procedimiento de servicio
  - Salva todos los registros (contexto) en la tabla de procesos (para P3)
  - P3 pasa de en ejecución a listo
  - P1 pasa de bloqueado a listo
  - Se ejecuta el código de la interrupción
  - Se llama al planificador que decide qué proceso se ejecutará a continuación
  - Se llama al despachador

7

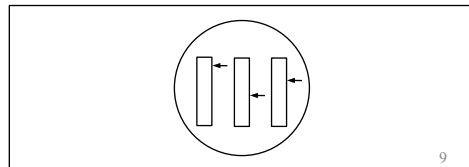
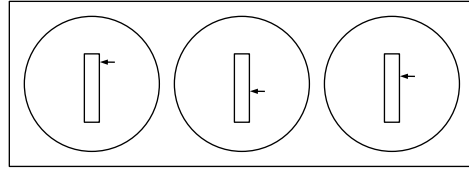
## Despachador (dispatcher)

- Se encarga de iniciar el proceso seleccionado
- Para ello:
  - Cambia el contexto
  - Cambia a modo usuario
  - Salta a la posición adecuada del programa de usuario

8

## Hilos

- Proceso
  - Ejecución de un programa
- Hilo = Thread
  - Subproceso dentro de un proceso (proceso ligero)
- Cada hilo tiene independiente:
  - Contador de programa (PC)
  - Pila (Stack)
  - Estado
- Los hilos de un proceso comparten:
  - Espacio de direcciones
  - Variables globales
  - Ficheros abiertos
  - Semáforos



9

## Ventajas e inconvenientes

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>• Ventajas           <ul style="list-style-type: none"> <li>– Comunicación más fácil</li> <li>– Mayor paralelismo con ejecución secuencial y llamadas al sistema bloqueantes (síncronas)</li> <li>– Cambio de contexto más simple</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>• Inconvenientes           <ul style="list-style-type: none"> <li>– No hay protección de un hilo frente a otro</li> </ul> </li> </ul> |
|---|--|

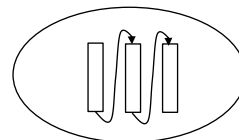
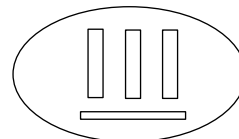
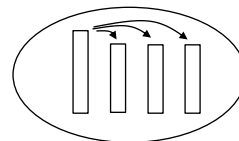
10

## Ejemplo

- Servidor de fichero (usa caché para acceder al disco)
  - N procesos: difícil compartir la caché
  - 1 proceso sin múltiples hilos
    - Llamadas al sistema bloqueantes (sin paralelismo)
    - Llamadas al sistema no bloqueantes (difícil de programar)
  - 1 proceso con N hilos
    - Cada hilo procesa una petición de forma secuencial y no importa que se bloquee
    - La caché estaría en variables globales o memoria dinámica
- Cliente navegador web
  - Cada hilo puede abrir una conexión para ir descargando los datos

## Organización

- Se pueden usar de múltiples formas
  - Hilo servidor
  - Equipo
  - Entubamiento (pipe-lining)



## Implementación

- Paquete de hilos:
  - Conjunto de primitivas accesibles al usuario relacionadas con los hilos
- Gestión de los hilos
  - Hilos estáticos
  - Hilos dinámicos
- Implementación del paquete de hilos
  - En el espacio de usuario
    - Más simple
      - Creación y destrucción de hilos
      - Cambio de hilo
    - Permite portar código
  - En el Kernel
    - Más paralelismo (no bloquea)

## Comunicación entre procesos

- Los procesos que cooperan entre sí se tienen que comunicar
  - Normalmente leen o escriben en un área común (en memoria o disco)
- Ej: Cuando un proceso quiere imprimir un fichero:
  - Escribe el nombre en un directorio especial llamado directorio de spooling
  - El demonio impresor mira periódicamente el directorio para ver si hay algún fichero para imprimir (lo imprime y luego borra el nombre)

# A y B quieren imprimir (fA y fB)

Directorio de spooling

4	f1
5	f2
6	f3
7	

ent = 7  
sal = 4

- El proceso A y B deciden imprimir
- A lee ent (7)
- Llega la interrupción del reloj y el planificador cede el paso a B
- B lee ent (7)
- Pone fB en el directorio de spooling
- Y actualiza ent a 8
- Cuando A se siga ejecutando pondrá fA en la entrada 7, borrando lo que dejó B
- El demonio impresor no ve ninguna anomalía, pero fB nunca se imprime

Directorio de spooling

4	f1
5	f2
6	f3
7	fB

ent = 8  
sal = 4

4	f1
5	f2
6	f3
7	<del>fB</del> fA

15

## Condiciones de carrera

- Son las situaciones en las que 2 o más procesos leen o escriben en una zona común (área compartida) y el resultado final depende de los instantes de ejecución de cada uno
- Para evitar las condiciones de carrera se impide que haya más de un proceso leyendo o escribiendo simultáneamente los datos compartidos

### » EXCLUSIÓN MUTUA

- Si un procesos está accediendo al área compartida ningún otro proceso puede acceder
- **SECCIÓN CRÍTICA:** parte del programa que accede a memoria compartida

16



## Las soluciones deben cumplir:

- Que nunca haya dos procesos que estén dentro de sus regiones críticas
- Que no se haga suposición a priori de la velocidad de los procesos
- Qué ningún proceso fuera de su región crítica pueda bloquear a otros
- Que ningún proceso tenga que esperar un tiempo arbitrariamente grande para entrar en su RC

17

## Programación concurrente

- Distintos mecanismos para que no se produzcan condiciones de carrera:
  - Exclusión mutua con espera activa
  - Exclusión mutua sin espera activa

18

## Exclusión mutua con espera activa

- Prohibición de las interrupciones
- Variables cerrojo
- Alternancia estricta
- Solución de Peterson
- La instrucción TSL

19

## Prohibición de las interrupciones

- Se prohíben todas las interrupciones en la RC y se permiten al salir
- Como la asignación del procesador se realiza como consecuencia de una interrupción, no se cambia de proceso hasta que no se sale de la RC
- No recomendable
  - En procesos de usuario
  - En sistemas con más de un procesador
- Si en el kernel del S.O con un solo procesador

20

## Variable cerrojo

- Se tiene una variable cerrojo compartida
  - 0: No hay ningún proceso en la RC
  - 1: Hay un proceso en la RC
- Valor inicial 0
- Un proceso que quiere entrar en la RC
  - Mira el valor del cerrojo
  - Si es 0, lo pone a 1 y entra en la RC
  - Si ya es 1, espera a que sea 0
- Al salir se pone a 0 el cerrojo

21

## Variable cerrojo

```
while (cerrojo == 1);
cerrojo=1;
region_critica();
cerrojo=0;
```

- cerrojo a 0
- P0 pone cerrojo a 1 y entra en RC
- P1 quiere entrar - Espera
- P0 sale de la RC - cerrojo a 0
- P1 pone cerrojo a 1 y entra en la RC

**Espera activa:** comprobación continua de una variable hasta que adquiera un determinado valor  
Problema: Desperdicia tiempo de CPU

22

## Problema con variable cerrojo

```

→ while (cerrojo == 1);
   cerrojo=1;
   region_critica();
   cerrojo=0;

```

- cerrojo a 0
- P0 se dispone a poner cerrojo a 1
- Llega una interrupción y el planificador pasa a P1
- P1 pone cerrojo a 1 y entra en la RC
- ???

23

## Alternancia estricta

- Variable turno entera que lleva la cuenta de a quién le toca el turno de entrar en la RC y utilizar la memoria compartida

```

while (TRUE){
    while(turno!=0);
    region_critica();
    turno=1;
    region_no_critica();
}

```

```

while (TRUE){
    while(turno!=1);
    region_critica();
    turno=0;
    region_no_critica();
}

```

24

## Alternancia estricta

- P0 entra en la RC y P1 quiere entrar

```

→ while (TRUE){
→   while(turno!=0);
→   region_critica();
   turno=1;
   region_no_critica();
}

```

```

while (TRUE){
  while(turno!=1);
  region_critica();
  turno=0;
  region_no_critica();
}

```

Proceso	turno
	0
P0 entra en RC	

25

## Alternancia estricta

- P0 entra en la RC y P1 quiere entrar

```

→ while (TRUE){
→   while(turno!=0);
→   region_critica();
   turno=1;
   region_no_critica();
}

```

```

→ while (TRUE){
→   while(turno!=1);
→   region_critica();
   turno=0;
   region_no_critica();
}

```

Proceso	turno
	0
P0 entra en RC	
P1 espera	

26

# Alternancia estricta

- P0 sale de la RC

```
➡ while (TRUE){
➡   while(turno!=0);
➡   region_critica();
➡   turno=1;
➡   region_no_critica();
}
```

```
➡ while (TRUE){
➡   while(turno!=1);
➡   region_critica();
➡   turno=0;
➡   region_no_critica();
}
```

Proceso	turno
	0
P0 entra en RC	
P1 espera	
P0	1
P0 en R no C	

27

# Alternancia estricta

- P1 entra en RC

```
➡ while (TRUE){
➡   while(turno!=0);
➡   region_critica();
➡   turno=1;
➡   region_no_critica();
}
```

```
➡ while (TRUE){
➡   while(turno!=1);
➡   region_critica();
➡   turno=0;
➡   region_no_critica();
}
```

Proceso	turno
	0
P0 entra en RC	
P1 espera	
P0	1
P0 en R no C	

Proceso	turno
P1 en RC	

28

# Alternancia estricta

- P0 quiere entrar en la RC

```
→ while (TRUE){
→   while(turno!=0);
→   region_critica();
→   turno=1;
→   region_no_critica();
}
```

Proceso	turno
	0
P0 entra en RC	
P1 espera	
P0	1
P0 en R no C	

```
→ while (TRUE){
→   while(turno!=1);
→   region_critica();
→   turno=0;
→   region_no_critica();
}
```

Proceso	turno
P1 en RC	
P0 espera	

29

# Alternancia estricta

- Si P0 más rápido que P1

```
→ while (TRUE){
→   while(turno!=0);
→   region_critica();
→   turno=1;
→   region_no_critica();
}
```

Proceso	turno
	0
P0 entra en RC	
P1 espera	
P0	1
P0 en R no C	

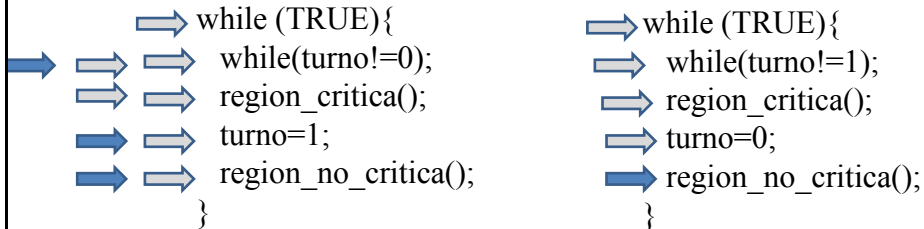
```
→ while (TRUE){
→   while(turno!=1);
→   region_critica();
→   turno=0;
→   region_no_critica();
}
```

Proceso	turno
P1 en RC	
P0 espera	
P1	0
P1 en R no C	
P0 en RC	

30

## Alternancia estricta

- Si P0 más rápido que P1



- Problema: Un proceso que no está en su RC puede bloquear a otro proceso que quiere entrar en su RC.
  - Esta solución obliga a la alternancia estricta

## Solución de Peterson

```

#define FALSE 0
#define TRUE 1
#define N 2
int turno=0;
int interesado[N]; /* iniciado a FALSE */

entrar_en_region(int proceso)
{
    int el_otro;
E1  el_otro=1-proceso;
E2  interesado[proceso]=TRUE;
E3  turno=proceso;
E4  while(turno==proceso && interesado[el_otro]==TRUE);
}

salir_de_region(int proceso)
{
S1  interesado[proceso]=FALSE;
}

```

El proceso:

```

while(TRUE) {
1   entrar_en_region(proceso); /*0 / 1 */
2   region_critica();
3   salir_de_region(proceso);
4   region_no_critica();
}

```



## Solución de Peterson

```

#define FALSE 0
#define TRUE 1
#define N 2
int turno=0;
int interesado[N]; /* iniciado a FALSE */

entrar_en_region(int proceso)
{
    int el_otro;
E1    el_otro=1-proceso;
E2    interesado[proceso]=TRUE;
E3    turno=proceso;
E4    while(turno==proceso && interesado[el_otro]==TRUE);
    salir_de_region(int proceso)
    {
S1    interesado[proceso]=FALSE;
    }

```

El proceso:

```

while(TRUE) {
1    entrar_en_region(proceso); /*0 / 1 */
2    region_critica();
3    salir_de_region(proceso);
4    region_no_critica();
}

```

33

## Solución de Peterson

```

#define FALSE 0
#define TRUE 1
#define N 2
int turno=0;
int interesado[N]; /* iniciado a FALSE */

entrar_en_region(int proceso)
{
    int el_otro;
E1    el_otro=1-proceso;
E2    interesado[proceso]=TRUE;
E3    turno=proceso;
E4    while(turno==proceso && interesado[el_otro]==TRUE);
    salir_de_region(int proceso)
    {
S1    interesado[proceso]=FALSE;
    }

```

El proceso:

```

while(TRUE) {
1    entrar_en_region(proceso); /*0 / 1 */
2    region_critica();
3    salir_de_region(proceso);
4    region_no_critica();
}

```

34

# La instrucción TSL

- TSL- TEST AND SET LOCK
- Comprobación y asignación a cerrojo
- Cerrojo:
  - 0 si no hay ningún proceso en la RC
  - 1 si hay
- Lee el contenido de una dirección de memoria y carga un valor distinto de 0
- El HW garantiza que estas dos operaciones son indivisibles
  - Se bloquea la ruta de comunicación con la memoria para que ningún otro procesador pueda acceder a ella hasta que termine

35

# La instrucción TSL

```
entrar_en_region:
    tsl registro,indicador /* almacena indicador en registro y pone 1 */
    cmp registro,#0
    jnz entrar_en_region
    ret
```

```
salir_de_region:
    mv indicador, #0
    ret
```

CASO 1: P0 entra y P1 espera

	registro	indicador	
		0	
P0 (entrar_en_region)	0	1	entra en RC espera
P1 (entrar_en_region)	1	1	
P0 (salir_de_region)		0	
P1 (entrar_en_region)	0	1	entra en RC

36

# La instrucción TSL

entrar\_en\_region:

→ tsl registro,indicador /\* almacena indicador en registro y pone 1 \*/

cmp registro,#0

jnz entrar\_en\_region

ret

salir\_de\_region:

mv indicador, #0

ret

CASO 2: P0 y P1 quieren entrar

	registro	indicador	
		0	
P0 (entrar_en_region)	0	1	
P1 (entrar_en_region)	1	1	
P0 (entrar_en_region)	?????		

llega int. espera

37

## Problema de soluciones con espera activa

- En sistemas de planificación por prioridad
  - H de más prioridad que L
  - H bloqueado por E/S
  - L puede estar en su RC
  - Cuando H pase a listo -> se ejecuta
  - Si H quiere entrar en RC
    - Tiene que esperar que L salga de su RC
  - L no termina ya que es de menor prioridad

38

## Métodos SIN espera activa

- Métodos que bloquean la ejecución del proceso cuando no puede entrar en la RC.
- No malgastan tiempo de procesador
- Métodos:
  - Dormir y despertar
  - Semáforos
  - Monitores
  - Paso de mensajes

39

## Dormir y despertar

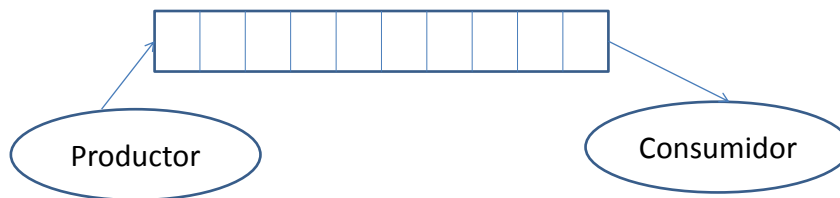
- Dormir (sleep): llamada al sistema que bloquea al llamador
  - Suspende su ejecución hasta que otro proceso lo despierta
- Despertar (wakeup): llamada al sistema que despierta al proceso que se da como parámetro
  - Si está bloqueado -> lo desbloquea
  - En caso contrario -> no hace nada
- Problema del productor/consumidor

40

## Dormir y despertar

### Problema del productor/consumidor

- 2 procesos que comparten un buffer finito
- El productor deja información en el buffer
- El consumidor lo retira
  - Si el productor quiere dejar algo en el buffer y está lleno se duerme, se despierta cuando el consumidor haya retirado elementos
  - Si el consumidor quiere retirar un elemento pero el buffer está vacío se duerme, se despierta cuando el productor deja elementos



41

## Dormir y despertar

### Problema del productor/consumidor

```
#define N 100
int cuenta=0;
productor()
{
    while(TRUE){
        producir_elemento();
        if (cuenta==N) dormir();
        dejar_elemento();
        cuenta=cuenta+1;
        if (cuenta==1) despertar(consumidor);
    }
}

Consumidor()
{
    while(TRUE){
        if (cuenta==0) dormir();
        retirar_elemento();
        cuenta=cuenta-1;
        if(cuenta==N-1) despertar(productor);
        consumir_elemento();
    }
}
```

cuenta: nº de elementos en el buffer (variable compartida)

42


## Dormir y despertar

### Problema del productor/consumidor

```
#define N 100
int cuenta=0;
productor()
{
    while(TRUE){
        producir_elemento();
        if (cuenta==N) dormir();
        dejar_elemento();
        cuenta=cuenta+1;
        if (cuenta==1) despertar(consumidor);
    }
}
```

```
Consumidor()
{
    while(TRUE){
        if (cuenta==0) dormir();
        retirar_elemento();
        cuenta=cuenta-1;
        if(cuenta==N-1) despertar(productor);
        consumir_elemento();
    }
}
```

**Problema**



43


## Dormir y despertar

### Problema del productor/consumidor

```
#define N 100
int cuenta=0;
productor()
{
    while(TRUE){
        producir_elemento();
        if (cuenta==N) dormir();
        dejar_elemento();
        cuenta=cuenta+1;
        if (cuenta==1) despertar(consumidor);
    }
}
```

```
Consumidor()
{
    while(TRUE){
        if (cuenta==0) dormir();
        retirar_elemento();
        cuenta=cuenta-1;
        if(cuenta==N-1) despertar(productor);
        consumir_elemento();
    }
}
```

**Problema**



Problema: Se puede producir una condición de carrera ya que el acceso a la variable cuenta no está controlada

Posible solución: Indicar el nº de señales despertar que no se han tenido en cuenta.

44

## Semáforos

- Semáforo: variable entera
- Operaciones básicas sobre un semáforo:
  - BAJAR (wait): Comprueba el valor del semáforo
    - Si semaforo > 0 : decrementar el valor y continuar la ejecución
    - Si semaforo = 0 : el proceso se pone a dormir
  - Acción atómica
  - SUBIR (signal)
    - Si hay procesos esperando: desbloquea a uno
    - En caso contrario: Incrementa el valor del semáforo
  - Los procesos no se bloquean
  - Acción atómica

45

## Semáforos

### Problema del productor/consumidor

```
#define TRUE 1
#define N 100
typedef int semaforo;
semaforo mutex=1;
semaforo vacio=N;
semaforo lleno=0;
```

```
productor()
{
    int elemento;
    while(TRUE){
        producir_elemento(&elemento);
        bajar(&vacio);
        bajar(&mutex);
        dejar_elemento(elemento);
        subir(&mutex);
        subir(&lleno);
    }
}
```

vacio: n° de posiciones vacías en el buffer  
 lleno: n° de posiciones llenas en el buffer  
 mutex: semáforo binario para exclusión mutua

```
consumidor()
{
    int elemento;
    while(TRUE){
        bajar(&lleno);
        bajar(&mutex);
        retirar_elemento(&elemento);
        subir(&mutex);
        subir(&vacio);
        consumir_elemento(elemento);
    }
}
```

46

# Semáforos

## Problema del productor/consumidor

```
#define TRUE 1
#define N 100
typedef int semaforo;
semaforo mutex=1;
semaforo vacio=N;
semaforo lleno=0;
```

Caso 1: El buffer está vacío y el consumidor quiere consumir

```
productor()
{
    int elemento;
    while(TRUE){
        producir_elemento(&elemento);
        bajar(&vacio);
        bajar(&mutex);
        dejar_elemento(elemento);
        subir(&mutex);
        subir(&lleno);
    }
}
```

```
consumidor()
{
    int elemento;
    while(TRUE){
        bajar(&lleno);
        bajar(&mutex);
        retirar_elemento(&elemento);
        subir(&mutex);
        subir(&vacio);
        consumir_elemento(elemento);
    }
}
```

47

# Semáforos

## Problema del productor/consumidor

```
#define TRUE 1
#define N 100
typedef int semaforo;
semaforo mutex=1;
semaforo vacio=N;
semaforo lleno=0;
```

Caso 2: El buffer se llena y el productor quiere dejar elemento

```
productor()
{
    int elemento;
    while(TRUE){
        producir_elemento(&elemento);
        bajar(&vacio);
        bajar(&mutex);
        dejar_elemento(elemento);
        subir(&mutex);
        subir(&lleno);
    }
}
```

```
consumidor()
{
    int elemento;
    while(TRUE){
        bajar(&lleno);
        bajar(&mutex);
        retirar_elemento(&elemento);
        subir(&mutex);
        subir(&vacio);
        consumir_elemento(elemento);
    }
}
```

48



# Semáforos

## Problema del productor/consumidor

```
#define TRUE 1
#define N 100
typedef int semaforo;
semaforo mutex=1;
semaforo vacio=N;
semaforo lleno=0;
```

Caso 3: El productor está dejando un elemento y el consumidor quiere retirar

```
productor()
{
    int elemento;
    while(TRUE){
        producir_elemento(&elemento);
        bajar(&vacio);
        bajar(&mutex);
        → dejar_elemento(elemento);
        subir(&mutex);
        subir(&lleno);
    }
}
```

```
consumidor()
{
    int elemento;
    while(TRUE){
        bajar(&lleno);
        bajar(&mutex);
        retirar_elemento(&elemento);
        subir(&mutex);
        subir(&vacio);
        consumir_elemento(elemento);
    }
}
```

49

## Usos de los semáforos

- Para exclusión mutua
  - Semáforo binario
- Para sincronización

50

## Semáforos

```
#define TRUE 1
#define N 100
typedef int semaforo;
semaforo mutex=1;
semaforo vacio=N;
semaforo lleno=0;
```

```
productor()
{
    int elemento;
    while(TRUE){
        producir_elemento(&elemento);
        bajar(&vacio);
        bajar(&mutex);
        dejar_elemento(elemento);
        subir(&mutex);
        subir(&lleno);
    }
}
```

Problema: no es fácil comunicar procesos mediante semáforos

```
consumidor()
{
    int elemento;
    while(TRUE){
        bajar(&lleno);
        bajar(&mutex);
        retirar_elemento(&elemento);
        subir(&mutex);
        subir(&vacio);
        consumir_elemento(elemento);
    }
}
```

51

Si mutex estuviera arriba, el productor se quedaría bloqueado en bajar vacio y el mutex estaría a 0 por lo que el consumidor nunca podría producir.

## Otro uso de los semáforos

- Para abstraer las interrupciones
  - 1 semáforo por cada dispositivo de E/S con valor inicial 0
  - Al arrancar una operación de E/S se ejecuta BAJAR por lo que el proceso se queda bloqueado
  - El manejador de interrupciones ejecuta SUBIR para desbloquear al proceso que pidió la E/S

52

Los dispositivos E/S suelen funcionar en forma asíncrona, aunque desde el punto de vista del usuario es como si fuera síncrono. Es más fácil programar primitivas síncronas, que bloquean al proceso.

## Monitores

- Mecanismo de más alto nivel que los semáforos
  - Propuestos por Brinch Hansen y Hoare
- Monitor:
  - Conjunto de procedimientos (funciones), variables y estructuras de datos
  - Los procesos pueden llamar a los procedimientos del monitor, pero no pueden acceder a las estructuras de datos internas al monitor (encapsulación)

```
monitor ejemplo
  declaración de variables
  definición de procedimientos
end monitor
```

53

## Monitores: exclusión mutua

- Sólo un proceso puede estar activo dentro de un monitor en un instante dado
- Si otro proceso intentase entrar queda bloqueado
- Se programa las RC's como procedimientos dentro del monitor
- El compilador traduce estos procedimientos de forma distinta:
  - Primero se comprueba si hay algún otro proceso dentro del monitor
    - Si hay algún otro proceso, el proceso se suspende
    - En caso contrario, el proceso entra
  - Se implementan por medio de semáforos

54

## Monitores: sincronización

- Variables condición
  - Operaciones:
    - ESPERAR: el proceso se bloquea y permite entrar a otro proceso en el monitor
    - DARPASO: desbloquea a un proceso que se haya quedado esperando en la variable condición
      - Hay que evitar que dos procesos estén activos dentro del monitor
        - » Solución: un proceso que ejecuta DARPASO debe salir inmediatamente del monitor
  - Las variables condición no son contadores.

55

## Monitores problema del productor/consumidor

```

monitor ProductorConsumidor
  condition lleno,vacio;
  integer cuenta;

  procedure depositar;
  begin
    if cuenta=N then esperar(lleno);
    depositar_elemento;
    cuenta:=cuenta+1;
    if cuenta=1 then darpaso(vacio);
  end;

  procedure retirar;
  begin
    if cuenta = 0 then esperar(vacio);
    retirar_elemento;
    cuenta:=cuenta-1;
    if cuenta= N-1 then darpaso(lleno);
  end;

begin
  cuenta:= 0;
end monitor;
procedure productor;
begin
  while true do
    begin
      producir_elemento;
      ProductorConsumidor.depositar
    end
  end;
end;
procedure consumidor;
begin
  while true do
    begin
      ProductorConsumidor.retirar
      consumir_elemento;
    end
  end;
end;

```

56

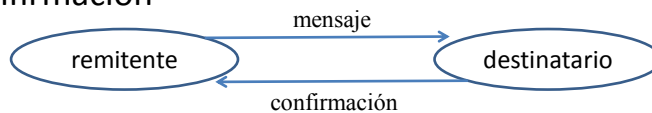
## Paso de mensajes

- Para comunicar y sincronizar procesos
- 2 primitivas:
  - ENVIAR (send)
    - enviar(destino,&mensaje)
  - RECIBIR (receive)
    - recibir(origen,&mensaje)
    - Bloquea al proceso mientras no se reciba un mensaje

57

## Problemas adicionales

- Pérdida de mensajes
  - Para evitarlo el destinatario devuelve una confirmación



- Nombrado de procesos
  - proceso@maquina / proceso@maquina.dominio
- Autenticación
  - cifrado

58

## Paso de mensajes

### Problema del productor/consumidor

```
#define N 100

productor()
{
    int elemento;
    mensaje m;
    while(TRUE){
        producir_elemento(&elemento);
        recibir(consumidor,&m);
        formar_mensaje(&m,elemento);
        enviar(consumidor,&m);
    }
}

Consumidor()
{
    int elemento,i;
    mensaje m;
    for(i=0; i<N;i++)
        enviar(productor,&m);
    while(TRUE){
        recibir(productor,&m);
        extraer_elemento(&m,&elemento);
        enviar(productor,&m);
        consumir_elemento(elemento);
    }
}
```

59

## Paso de mensajes

### Buzones

- Buzón = buffer
  - Si se utilizan buzones:
    - El proceso que envía se queda bloqueado si el buzón está lleno, hasta que el proceso que receptor recibe el mensaje
  - Si no se utilizan buzones – Rendezvous (cita)
    - El proceso que envía se queda bloqueado hasta que el otro reciba
    - El proceso que recibe se queda bloqueado hasta que el otro envía
      - Obliga a sincronizar al remitente y al destinatario en cada paso

60

## Equivalencia entre primitivas

- Se pueden utilizar monitores, mensajes o semáforos para implementar cualquiera de las otras técnicas
- Por ejemplo:
  - si en un sistema tenemos monitores pero no semáforos, podemos implementar los semáforos a partir de los monitores y al revés.

61

## Equivalencia entre primitivas: Mensajes para implementar semáforos

- Sincronizador
  - Por cada semáforo:
    - Contador y lista de procesos bloqueados
- Un proceso para realizar un SUBIR o BAJAR
  - Envía un mensaje indicando la Operación y el semáforo al proceso sincronizador
  - Ejecuta un recibir para obtener respuesta del sincronizador
- El sincronizador para realizar las operaciones:
 

<ul style="list-style-type: none"> <li>– BAJAR:</li> </ul> <p>Si contador &gt; 0          Decrementar contador          Enviar mensaje al proceso</p> <p>Si contador = 0          Añadir proceso a la lista de bloqueados          No enviar mensaje (bloquea)</p>	<ul style="list-style-type: none"> <li>– SUBIR:</li> </ul> <p>Enviar mensaje al proceso          Si procesos bloqueados (lista no vacía)          Eliminar un proceso de la lista          Enviar mensaje a este proceso (desbloquea)</p> <p>En caso contrario          Incrementar el contador</p>
--	---

62

## Problemas clásicos de comunicación entre procesos

- Los filósofos comensales
- Lectores/escritores

63