

# Algoritmos de Planificación del Procesador

Victor Tortolero, 24.569.609

Sistemas Operativos, FACYT

15 de abril de 2016

# Primera version del algoritmo de Dekker

- Primer algoritmo en resolver la exclusión mutua.
- Aplica la exclusión mutua de manera correcta, y la garantiza.
- Usa variables para controlar que hilo se ejecutara.
- Revisa constantemente si la sección critica esta disponible (spinlock, busy waiting), lo que malgasta el tiempo del procesador.
- Los procesos lentos atrasan a los rápidos.

# Primera version del algoritmo de Dekker

```
1  int numero_de_proceso = 1;
2  iniciar_procesos(); // inicializa y corre ambos procesos
3
4  proceso p1:
5  void main() {
6      while(!no_terminado){
7          while(numero_de_proceso == 2); // seccion de entrada de la exclusion mutua
8          // seccion critica
9          numero_de_proceso = 2; // seccion de salida de la exclusion mutua
10         // codigo restante
11     }
12 } // fin del proceso p1
13
14 proceso p2:
15 void main() {
16     while(!no_terminado){
17         while(numero_de_proceso == 1); // seccion de entrada de la exclusion mutua
18         // seccion critica
19         numero_de_proceso = 1; // seccion de salida de la exclusion mutua
20         // codigo restante
21     }
22 } // fin del proceso p2
```

## Segunda version del algoritmo de Dekker

- No garantiza la exclusión mutua.
- Ambos procesos pueden entrar al mismo tiempo en su sección critica

# Segunda version del algoritmo de Dekker

```
1  bool dentro_de_p1 = false;
2  bool dentro_de_p2 = false;
3  iniciar_procesos(); // inicializa y corre ambos procesos
4
5  proceso p1:
6  void main(){
7      while(!no_terminado){
8          while(dentro_de_p2); // seccion de entrada de la exclusion mutua
9          dentro_de_p1 = true; // seccion de entrada
10         // seccion critica
11         dentro_de_p2 = false; // seccion de salida de la exclusion mutua
12         // codigo restante
13     }
14 } // fin del proceso p1
15
16 proceso p2:
17 void main(){
18     while(!no_terminado){
19         while(dentro_de_p1); // seccion de entrada de la exclusion mutua
20         dentro_de_p2 = true; // seccion de entrada
21         // seccion critica
22         dentro_de_p1 = false; // seccion de salida de la exclusion mutua
23         // codigo restante
24     }
25 } // fin del proceso p2
```

# Tercera version del algoritmo de Dekker

- Garantiza la exclusión mutua.
- Es posible que ocurra un deadlock.

# Tercera version del algoritmo de Dekker

```
1  bool p1_quiere_entrar = false;
2  bool p2_quiere_entrar = false;
3  iniciar_procesos(); // inicializa y corre ambos procesos
4
5  proceso p1:
6  void main(){
7      while(!no_terminado){
8          p1_quiere_entrar = true; // seccion de entrada de la exclusion mutua
9          while(p2_quiere_entrar); // seccion de entrada
10         // seccion critica
11         p1_quiere_entrar = false; // seccion de salida de la exclusion mutua
12         // codigo restante
13     }
14 } // fin del proceso p1
15
16 proceso p2:
17 void main(){
18     while(!no_terminado){
19         p2_quiere_entrar = true; // seccion de entrada de la exclusion mutua
20         while(p1_quiere_entrar); // seccion de entrada
21         // seccion critica
22         p2_quiere_entrar = false; // seccion de salida de la exclusion mutua
23         // codigo restante
24     }
25 } // fin del proceso p2
```

# Cuarta version del algoritmo de Dekker

- Es posible posponer un proceso de manera indefinida.
- Apaga las banderas por cortos periodos de tiempo para tomar control.



# Cuarta version del algoritmo de Dekker

```
1  bool p1_quiere_entrar = false;
2  bool p2_quiere_entrar = false;
3  iniciar_procesos(); // inicializa y corre ambos procesos
4  proceso p1:
5  void main() {
6      while(!no_terminado){
7          p1_quiere_entrar = true; // seccion de entrada de la exclusion mutua
8          while(p2_quiere_entrar){ // seccion de entrada
9              p1_quiere_entrar = false; // seccion de entrada
10             // esperar por una cantidad de tiempo pequena aleatoria
11             p1_quiere_entrar = true;
12         }
13         // seccion critica
14         p1_quiere_entrar = false; // seccion de salida
15         // codigo restante
16     }
17 } // fin del proceso p1
18 proceso p2:
19 void main() {
20     while(!no_terminado){
21         p2_quiere_entrar = true; // seccion de entrada de la exclusion mutua
22         while(p1_quiere_entrar){ // seccion de entrada
23             p2_quiere_entrar = false; // seccion de entrada
24             // esperar por una cantidad de tiempo pequena aleatoria
25             p2_quiere_entrar = true;
26         }
27         // seccion critica
28         p2_quiere_entrar = false; // seccion de salida
29         // codigo restante
30     }
31 } // fin del proceso p2
```

# Quinta version del algoritmo de Dekker

- Garantiza la exclusión mutua.
- Marca procesos como preferidos para determinar el uso de las secciones criticas.
- El estatus de “Preferido” se turna entre los procesos.
- Evita deadlock's, y el posponer un proceso de manera indefinida.

# Quinta version del algoritmo de Dekker I

```
1  int proceso_preferido = 1;
2  bool p1_quiere_entrar = false;
3  bool p2_quiere_entrar = false;
4  iniciar_procesos(); // inicializa y corre ambos procesos
5
6  proceso p1:
7  void main() {
8      while(!no_terminado){
9          p1_quiere_entrar = true;
10         while(p2_quiere_entrar){
11             if(proceso_preferido == 2){
12                 p1_quiere_entrar = false;
13                 while(proceso_preferido == 2); // busy wait
14                 p1_quiere_entrar = true;
15             }
16         }
17         // seccion critica
18         proceso_preferido = 2;
19         p1_quiere_entrar = false; // seccion de entrada
20         // codigo restante
21     }
22 } // fin del proceso p1
```

# Quinta version del algoritmo de Dekker II

```
23
24 proceso p2:
25 void main() {
26     while(!no_terminado){
27         p2_quiere_entrar = true;
28         while(p1_quiere_entrar){
29             if(proceso_preferido == 1){
30                 p2_quiere_entrar = false;
31                 while(proceso_preferido == 1); // busy wait
32                 p2_quiere_entrar = true;
33             }
34         }
35         // seccion critica
36         proceso_preferido = 1;
37         p2_quiere_entrar = false;
38         // codigo restante
39     }
40 } // fin del proceso p2
```

---

# Algoritmo de Peterson

- Garantiza la exclusión mutua.
- Cada proceso tendrá su turno.
- Requiere que los 2 procesos compartan 2 variables: **int** turn;  
**boolean** flag[2];

# Quinta version del algoritmo de Dekker I

---

```
1  do{
2      flag[i] = true;
3      turn = j;
4      while(flag[j] && turn == j);
5      // seccion critica
6      flag[i] = false;
7      // codigo restante
8  }while(true);
```

---

# Test and Set

- Operación Atómica.
- Retorna el valor del lock, y lo cambia a verdadero.
- Si el valor retornado es **falso**, obtenemos el lock. Si es **verdadero**, esta ocupado por otro proceso.

---

```
1  boolean test_and_set(boolean *target) {  
2      boolean rv = *target;  
3      *target = true;  
4      return rv;  
5  }
```

---

# Compare and Swap

- Operación Atómica.
- Retorna el valor original de value.
- Cambia el valor de la variable value si es igual al valor de expected.

---

```
1  int compare_and_swap(int *value, int expected, int new_value) {  
2      int temp = *value;  
3  
4      if(*value == expected)  
5          *value = new_value;  
6  
7      return temp;  
8  }
```

---



# Problemas Clasicos: Productor y Consumidor

- Los productores insertan elementos en el buffer. Los consumidores los extraen.
- No se pueden insertar elementos en el buffer si esta lleno. No se pueden extraer si esta vacío.
- **Semaphore:** (binario)mutex = 1, (contador)empty = n, (contador)full = 0;

## Proceso Productor

---

```
do{
    /*Produce elemento */
    wait(empty);
    wait(mutex);
    /* Inserta elemento en el buffer*/
    signal(mutex);
    signal(full);
}while(true);
```

---

## Proceso Consumidor

---

```
do{
    wait(full);
    wait(mutex);
    /* Obtiene elemento del buffer */
    signal(mutex);
    signal(empty);
    /* Consume elemento */
}while(true);
```

---

# Problemas Clasicos: Lectores y Escritores

- Una base de datos que debe ser compartida por **lectores** y **escritores**.
- Si dos lectores acceden de manera simultanea, no hay problemas.
- Si un escritor y otro proceso (lector o escritor), acceden simultáneamente, puede traer problemas.

## Proceso Lector

---

```
do{
    wait(turn);
    signal(turn);
    wait(mutex);
    read_count++;
    if(read_count == 1) wait(rw_mutex);
    signal(mutex);
    /* Se realiza la lectura */
    wait(mutex);
    read_count--;
    if(read_count == 0) signal(rw_mutex);
    signal(mutex);
}while(true);
```

---

Semaphore: (binario)rw\_mutex = 1,

(binario)mutex = 1; Int read\_count = 0;

## Proceso Escritor

---

```
do{
    wait(turn);
    wait(rw_mutex);
    /* Se realiza la escritura */
    signal(turn);
    signal(rw_mutex);
}while(true);
```

---

# Problemas Clasicos: Barbero Dormilon

- El barbero duerme si no hay clientes.
- Si no hay sillas disponibles, el cliente se va. En caso contrario se sienta en una.
- Si el cliente llega y el barbero esta dormido, lo despierta.

## Proceso Cliente

---

```
do{
    wait(mutex);
    if(sillas_libres == 0){
        signal(mutex);
        break;
    }
    sillas_libres--;
    signal(mutex);
    wait(sillon);
    wait(mutex);
    sillas_libres++;
    signal(mutex);
    signal(barbero);
    /* cortarse el pelo */
    wait(finalizo_corte);
    signal(sillon);
}while(true);
```

---

Int: sillas\_libres = N; Semaphore: sillon = 1,  
barbero = 0, fin = 0, mutex = 1, turn = 1;

## Proceso Barbero

---

```
do{
    wait(barbero);
    /* cortar pelo */
    signal(fin);
}while(true);
```

---

# Problemas Clasicos: Filósofos Comensales





- Cuando un filosofo tiene hambre, intenta acceder a los palillos de su izquierda y derecha.
- Un filosofo necesita ambos palillos para comer.
- Un filósofo no puede quitarle un palillo a otro filósofo.
- **Semaphore:** palillos[5] = 0, sirviente = 1;

---

```
do{  
    /* Pensar */  
    wait(sirviente);  
    wait(palillos[i]);  
    wait(palillos[(i + 1) % 5]);  
    /* Comer */  
    signal(palillos[i]);  
    signal(palillos[(i + 1) % 5]);  
    signal(sirviente);  
}while(true);
```

---

# References I

-  Abraham Silberschatz & Peter Baer Galvin & Greg Gagne, *Fundamentos de sistemas operativos, 7ma edición*, McGraw Hill, 2005.
-  David Vallejo Fernández & Carlos González Morcillo & Javier A. Albusac Jiménez, *Programación concurrente y tiempo real, tercera edición*, 2016.
-  *Presentacion de la university of limerick*, Disponible en <http://garryowen.csisdms.ul.ie/~cs4023/resources/oth6.pdf>.
-  Jonathan Walpole, *Introduction to operating systems - interprocesses communication & synchronization*.