SINCRONIZACIÓN DE PROCESOS

Sincronización de Procesos

• En un sistema existen muchos más procesos que recursos disponibles

• El acceso a los recursos debe regularse

• Se presentan en el sistema lo que se denomina Condiciones de Competencia (*Race Condition*)

Definiciones

- *Concurrencia*

Propiedad de los sistemas en la cual dos o más procesos se ejecutan simultáneamente, y potencialmente interactúan entre sí.

- *Race Condition*

Dos o más procesos acceden y manipulan los mismos datos de manera concurrente, y el estado de estos datos depende del orden de ejecución y como acceden a los datos dichos procesos

Instituto Politécnico Superior

Sincronización de Procesos

- Ejemplo de ***Race Condition***

variable **cant**

```
Process P1() {
  var i;
  i = cant;     (1)
  i = i + 1;    (2)
  cant = i;     (3)
}
```

```
Process P2() {
  var j;
  j = cant;     (1)
  j = j + 1;    (2)
  cant = j;     (3)
}
```

1 2 3 1 2 3	1 1 2 3 2 3	1 1 2 2 3 3	1 2 1 3 2 3
1 2 3 1 2 3	1 2 1 3 2 3	1 1 2 2 3 3	1 2 1 2 3 3
1 1 2 2 3 3	1 2 1 2 3 3	1 1 2 3 2 3	1 2 1 2 3 3
1 1 2 3 2 3	1 2 1 2 3 3	1 1 2 3 2 3	1 1 2 2 3 3

- **2** ejecuciones correctas entre **16** posibles
  Un **12,5%** de probabilidad de ejecución correcta

Secciones o Regiones Críticas

- Proveer un mecanismo para prohibir que más de un proceso lea o escriba simultáneamente en los datos compartidos

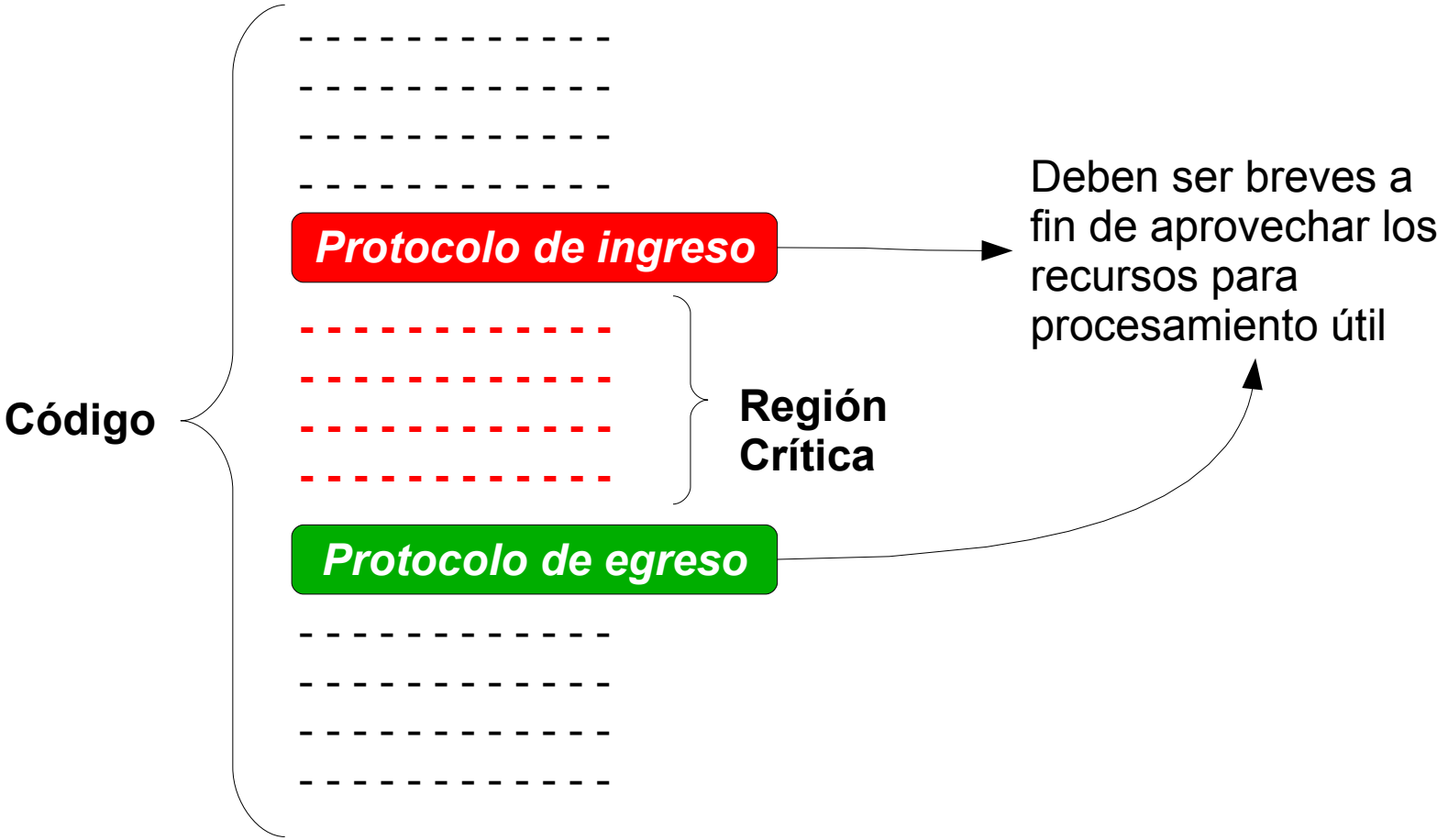
- Ejecutar el bloque de instrucciones que accede a los datos compartidos, la *región crítica*, como si fuese una única instrucción (indivisible, *atómica*)

- Cuando un proceso ejecuta su RC ningún otro proceso puede acceder a la suya ***Exclusión Mutua***

- Deben diseñarse protocolos que impidan o bloqueen el acceso a una RC mientras otro se encuentra en la suya

- Evitar condiciones de competencia → garantizar la exclusión mutua

- Más información: http://en.wikipedia.org/wiki/Critical_section


Código
Protocolo de ingreso
Región
Crítica
Protocolo de egreso
Deben ser breves a
fin de aprovechar los
recursos para
procesamiento útil

Secciones o Regiones Críticas

Condiciones o requisitos para obtener una solución al problema de las RC

- *Exclusión mutua*
  Si un proceso está ejecutando código de la RC, ningún otro proceso podrá hacerlo en la suya

- *Bloqueo*
  Ningún proceso en ejecución fuera de su RC puede bloquear a otros procesos

- *Progreso*
  Si ningún proceso está ejecutando dentro de la RC, la decisión de qué proceso entra en la sección se hará sobre los procesos que desean entrar

- *Espera acotada o limitada*
  Debe haber un límite en el número de veces que se permite que los demás procesos entren a ejecutar código de la RC después de que un proceso haya efectuado una solicitud de entrada y antes de que se conceda la suya

Secciones o Regiones Críticas – Métodos para lograr la exclusión mutua

• Desactivación de interrupciones
- Hacer que se deactiven las interrupciones al ingresar un proceso a su RC y habilitarlas luego al egresar de la misma
- En sistemas monoprocesador es más sencillo de lograr
- Al no existir interrupciones se evitan las syscalls que provocan cambios de contexto

✗ *Contra*
No es correcto para los procesos de usuario, se les está otorgando un poder excesivo ¿si tiene algún bug durante la ejecución de la RC?

✔ Pro
Útil dentro del núcleo, a veces se realiza esto a fin de evitar que se vean perjudicadas tareas propias del sistema operativo

Secciones o Regiones Críticas – Métodos para lograr la exclusión mutua

- **Variables de cerradura**

```
flag = false
```

```
Process P1 () {
   for ( ; ; ) {
        while ( flag == true ) ;   ①
        flag = true;               ②
        /* RC */
        flag = false;
    }
}
```

```
Process P2 () {
   for ( ; ; ) {
        while ( flag == true ) ;   ①
        flag = true;               ②
        /* RC */
        flag = false;
    }
}
```

✔ Sencillo

✘ No funciona !! Secuencia: ① ① ② ②

Secciones o Regiones Críticas – Métodos para lograr la exclusión mutua

- **Alternancia estricta**

```
turno = false
```

```
Process P1 () {
   for ( ; ; ) {
        while ( turno != false ) ;
        /* RC */
        turno = true;
    }
}
```

```
Process P2 () {
   for ( ; ; ) {
        while ( turno != true ) ;
        /* RC */
        turno = false;
    }
}
```

✔ Funciona

✗ **P1** no puede ingresar nuevamente a su RC si **P2** no lo hace antes
Espera ocupada (*spin lock*) - Comentar

Secciones o Regiones Críticas – Métodos para lograr la exclusión mutua

- Dos banderas

```
flag1 = false
flag2 = false
```

```
void bloqueo (bool *mi_flag, bool *su_flag) {
   mi_flag = true;
   while ( su_flag ) ;
}
```

```
void desbloqueo (bool *mi_flag) {
   mi_flag = false;
}
```

```
Process P1 () {
   for ( ; ; ) {
      bloqueo( &flag1, &flag2 );
      /* RC */
      desbloqueo( &flag1 );
    }
}
```

```
Process P2 () {
   for ( ; ; ) {
      bloqueo( &flag2, &flag1 );
      /* RC */
      desbloqueo( &flag2 );
    }
}
```

✗ Espera ocupada
Deadlock

- **Secciones o Regiones Críticas – Métodos para lograr la exclusión mutua**

- **Dos banderas - Solución?**

```
flag1 = false
flag2 = false
```

```
void bloqueo (bool *mi_flag, bool *su_flag) {
   mi_flag = true;
   while ( su_flag ) {
      mi_flag = false; mi_flag = true;
   }
}
```

```
void desbloqueo (bool *mi_flag) {
   mi_flag = false;
}
```

```
Process P1 () {
   for ( ; ; ) {
      bloqueo ( &flag1, &flag2 );
      /* RC */
      desbloqueo( &flag1 );
    }
}
```

```
Process P2 () {
   for ( ; ; ) {
      bloqueo ( &flag2, &flag1 );
      /* RC */
      desbloqueo( &flag2 );
    }
}
```

Puede darse el deadlock si se presenta una sincronización completa

Algoritmo de Peterson

- Formulado por Gary L. Peterson en 1981 - www.math.jmu.edu/~peterson

```
flag1 = false
flag2 = false
turno = 0

Process 0:                          Process 1:
  flag1 = true                        flag2 = true
  turno = 1                           turno = 0
  while( flag2 && turno == 1 ) ;      while( flag1 && turno == 0 ) ;
  /* RC */                            /* RC */
  flag1 = false                       flag2 = false
```

Algoritmo de Dekker

- Formulado por el matemático alemán Theodorus Dekker en la década del 60
- Considerada la primer solución correcta a la exclusión mutua evitando la alternancia estricta

```
flag1 = false
flag2 = false
turno = 0

Process 0:
   flag1 = true
   while (flag2 == true) {
      if (turno != 0) {
         flag1 = false
         while (turno != 0) ;
         flag1 = true
      }
   }
   /* RC */
   ...
   turno = 1
   flag1 = false
```

```
Process 1:
   flag2 = true
   while (flag1 == true) {
      if (turno != 1) {
         flag2 = false
         while (turno != 1) ;
         flag2 = true
      }
   }
   /* RC */
   ...
   turno = 0
   flag2 = false
```

Soluciones para múltiples procesos

• Se pretende que en estos algoritmos
- Exista exclusión mutua
- los *n* procesos ingresen a su RC
- los procesos fuera de su RC no inhiban a otro que desee ingresar

• Historia
- Dijkstra (1965) – Algoritmo no acotado (en espera de un proceso por acceder a su RC) – No cumplía *Espera limitada*
- Knuth (1966) – Algoritmo con cota 2^n
- Eisemberg-McGuire (1972) – Cota n-1
- Lamport (1974) - Cota n-1 (más simple que el anterior)
  - Conocido como ***Algoritmo de la Panadería***

Algoritmo de la Panadería (Bakery Algorithm)

- Basados en el modo de atención de los comercios
- Analogía: proceso ↔ cliente
- Cada cliente recibe un número, el que tiene en número más bajo es atendido
- El algoritmo no garantiza que dichos números sean únicos
- Cuando un proceso desea entrar a su RC debe chequear si es su turno para hacerlo.
  Debe chequear el número del resto de los procesos para asegurarse de que posee el menor.
  En el caso de que otro proceso posea el mismo número, el que posea el menor id (el PID) entrará a la RC primero
- Nomenclatura

```
(a, b) < (c,d)
```

significa

```
((a < c) || ((a==c) && (b<d))
```

Algoritmo de la Panadería (Bakery Algorithm)

```
// Variables globales
Entering: array [1..N] of bool = {false};
Number: array [1..N] of int = {0};

lock(int i) {
   Entering[i] = true;  // Deseo acceder
   Number[i] = 1 + max(Number[1], ..., Number[N]); // Obtengo un número
   Entering[i] = false; // Ya tengo un número

   for (j = 1; j <= N; j++) {
      // Esperar hasta que el proceso j reciba su número
      while (Entering[j]) ; /* nada */

      // Esperar hasta que todos los procesos con números menores o el mismo número,
      // pero con prioridad mayor, termine su trabajo
      while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) ; /* nada */

   }
}
```

Entering es true si el proceso **i** desea ingresar a la RC, usa **Number[i]** para hacerlo

Number[j] !=0, proceso j desea entrar a RC
Obtiene acceso a RC si se sale del for (todo while se falsea)

```
unlock(int i) {
   Number[i] = 0; //Sin inteción de acceso a RC
}
```

```
Process(int i) {
  while (true) {
     lock(i);
     /* RC */
     unlock(i);
     /* código corriente */
  }
}
```