Sistemas Operativos
Pauta Control 1
(2 horas)
Abril de 2009

Pregunta 1

Hemos definido un Sistema Operativo como un administrador de recursos que protege el hardware de los usuarios.

Sin embargo, un proveedor de S.O. argumenta que no es su culpa que el sistema se caiga, cuando existe una aplicación mal programada.

Discuta la afirmación con argumentos técnicos y según la plataforma de hardware que se esté usando. ¿Es posible hacer un S.O. en que un programa con errores o mal intencionado jamás pueda botarlo? ¿En cualquier hardware?

Es posible hacer un S.O. que nunca se caiga por errores de programas usuario, pero solo si contamos con una plataforma de hardware que nos provea: modo usuario/supervisor, traducción de direcciones de memoria y protección de las instrucciones de E/S. De esa forma, a través de interrupciones, el S.O. puede administrar toda la operación protegiendo al sistema de caídas.

En una plataforma que no provea esas funciones por hardware, no es posible hacer un S.O. que garantice esas propiedades y efectivamente una aplicación mal programada o mal intencionada puede botar al sistema completo.

Pregunta 2

Se quiere sincronizar un "paso cordillerano" de una sola vía, donde threads vienen por ambos lados (entrada 1 y entrada 2) y demoran un tiempo en atravesarlo. La sincronización es parecida a la que proveen los "bandereros" en la realidad, organizando el tráfico en una dirección por un rato y en otra después. La solución actual es la siguiente (usando monitores del nSystem):

```
struct Paso {
    nMonitor mon;
    nCondition vacio; /* paso vacio */
    int hay1, hay2; /* cuantos autos vienen desde 1 y desde 2 */
};
/* lo invoca auto que quiere entrar por 1 */
void entry1(struct Paso *paso) {
   nEnter(paso->mon);
   while(paso->hay2)
      nWaitCondition(paso->vacio);
   paso->hay1++;
   nExit(paso->mon);
}
/* lo invoca auto que sale, que entró por 1 */
void exit1(struct Paso *paso) {
```

```
   nEnter(paso->mon);
   if(--(paso->hay1)) nSignalCondition(paso->vacio);
}
/* entry2 y exit2 son simétricas a entry1 y exit1 */
```

Esta solución es injusta y puede generar hambruna. Explique por qué.

Si el flujo de autos en una dirección es constante y suficiente para mantener siempre ocupado el paso, los autos de la otra dirección nunca podrán pasar.

Luego, modifique la solución para implementar el "algoritmo del banderero": comienza dejando pasar desde un lado. Mientras llegan autos en ese lado, los va dejando pasar. Si llega gente desde el otro lado, y no hay nadie en el paso, cambia la bandera verde y los deja pasar. Pero, si hay gente en el paso y hay gente esperando al otro lado, solo deja pasar 5 seguidos en la misma dirección, luego bloquea esa entrada, espera que todos salgan, y finalmente deja pasar desde el otro lado.

Implemente el algoritmo del banderero modificando esta solución. Explique si resuelve la hambruna o no.

Esta solución es justa:

```
struct Paso {
    nMonitor mon;
    nCondition vacio; /* paso vacio */
    int hay1, hay2; /* cuantos autos vienen desde 1 y desde 2 */
    int esp1, esp2; /* esperando en 1 y en 2 */
    int turno;      /* 1: le toca a 1, 2: le toca a 2 */
    int pasaron;    /* cuantos han pasado en este turno */
};
/* lo invoca auto que quiere entrar por 1 */
void entry1(struct Paso *paso) {
   nEnter(paso->mon);
   paso->esp1++;
   while(paso->hay2 || (paso->turno == 2 && paso->esp2 > 0))
      nWaitCondition(paso->vacio);
   paso->esp1--;
   paso->hay1++;
   if(paso->turno == 1 && ++paso->pasaron >= 5) {
      paso->turno = 2;
      paso->pasaron = 0;
   }
   nExit(paso->mon);
}
/* lo invoca auto que sale, que entró por 1 */
void exit1(struct Paso *paso) {
   nEnter(paso->mon);
   if(--(paso->hay1)) nSignalConditionAll(paso->vacio);
   nExit(paso->mon);
}
/* entry2 y exit2 son simétricas a entry1 y exit1 */
```

Pregunta 3

La implementación de un semáforo en nSystem es:

```
void nWaitSem(nSem sem)
{
  START_CRITICAL();
  if (sem->count>0)
    sem->count--;
  else
  {
    current_task->status= WAIT_SEM;
    PutTask(sem->queue, current_task);
    ResumeNextReadyTask();
  }
  END_CRITICAL();
}
void nSignalSem(nSem sem)
{
  START_CRITICAL();
   if (EmptyQueue(sem->queue))
    sem->count++;
   else
   {
    nTask wait_task= GetTask(sem->queue);
    wait_task->status= READY;
  /* wait_task ahora pasa al estado ready y queremos que
   * tome la CPU.  La siguiente secuencia es la unica forma
   * de transferir la CPU directamente a otra tarea.
   */
    PushTask(ready_queue, current_task); /* Sigue estando ready */
    PushTask(ready_queue, wait_task);
    ResumeNextReadyTask(); /* wait_task es la primera en la cola! */
   /* Frecuentemente, esta tarea retomara la CPU cuando wait_task
    * pierda la CPU.
    */
  }
 END_CRITICAL();
}
```

Escriba una solución correcta que no use `PushTask()`, si no solamente `PutTask()` (que agrega al final de la cola) y en que Signal no use `ResumeNextReadyTask()` directamente.

```
void nWaitSem(nSem sem)
{
  START_CRITICAL();
  while(sem->count == 0)
  {
    current_task->status= WAIT_SEM;
```

```
    PutTask(sem->queue, current_task);
    ResumeNextReadyTask();
  }
  sem->count--;
  END_CRITICAL();
}
void nSignalSem(nSem sem)
{
  START_CRITICAL();
   if (EmptyQueue(sem->queue))
    sem->count++;
   else
   {
    nTask wait_task= GetTask(sem->queue);
    wait_task->status= READY;
    PutTask(ready_queue, wait_task);
   }
 END_CRITICAL();
}
```