Clase Auxiliar 4

Prof: L. Mateu
Aux: Javier Bustos

28 de agosto de 2006

1. Impresora dormilona

Este problema salió en el exámen del año 1998. El enunciado dice:

Suponga que en `nSystem` se dispone de una impresora compartida por todas las tareas. Dos o más tareas pueden usar simultáneamente la impresora, pero en este caso las líneas de la impresora saldrían entremezcladas. En una aplicación se desea evitar el entremezclado de líneas haciendo que cada tarea solicite el acceso exclusivo a la impresora antes de ocuparla y además notifique cuando termina de utilizarla. Por lo tanto una tarea tendrá la siguiente forma:

```
tarea()
{
...
obtenerImpresora();
... /* utilizar impresora */
devolverImpresora();
...
}
```

Además, por razones de ahorro de electricidad, se necesita que la impresora se coloque en modo de bajo consumo cada vez que transcurran 5 minutos sin ser utilizada por ninguna tarea.

Para colocar la impresora en modo de bajo consumo invoque el procedimiento `modoBajoConsumo()`. Para volver a usar la impresora cuando está en modo de bajo consumo, invoque el procedimiento `modoUsoNormal()`.

Implemente los procedimientos **obtenerImpresora()**, **devolverImpresora()** y un procedimiento **inicializarImpresora()** que se invoca al inicio de **nMain**.

El uso de $busy - waiting$ o consultas periódicas para ver si se han cumplido los 5 minutos es equivalente a no responder la pregunta.

2. Equivalencias entre mecanismos de sincronización

Implemente una versión simplificada de mensajes usando monitores, la API es la siguiente:

```
typedef struct { ... } Port;
void send(Port *p, void *msg)
/* se bloquea hasta que se haga receive
 (reply automatico) */
void *receive(Port *p)
```

3. Soluciones

3.1. P1

```
enum {PEDIR, DEVOLVER}
nTask impresora;
void obtenerImpresora()
{
int msg = PEDIR;
```

```
nSend(impresora,&msg);
}
void devolverImpresora()
{
int msg = DEVOLVER;
nSend(impresora,&msg);
}

// nEmitTask(inicializarImpresora())

void inicializarImpresora()
{
nTask t;
int *msg;
fifoqueue printerQueue;
while (true)
    {
    if (!nEmpty(printerQueue)) nReply(get(printerQueue),1);
    else
        {
        msg = (int *) nReceive(&t,5*60);  // Recibimos el pedido de  impresora
        if (msg == NULL) // Nos vamos al modo ahorro de energia
            {
            modoBajoConsumo();
            msg = (int *) nReceive(&t,-1); // Hasta que alguien nos  despierte
            }
     nReply(t,1);
        }
    do {
         msg = (int *) nReceive(&t,-1);
         if (*msg == PEDIR) put(printerQueue,t);
         } while(*msg == PEDIR);
     /*
    Solo a quien le dimos la impresora puede hacer un DEVOLVER,
    y es el que esta en  la variable t
    */
    nReply(t,0);
    }
}
```

3.2. P2

```
typedef struct {
  int num, visor; /* = 0 en el inicio */
  nMonitor mon;
  void *msg;
} Port;

void send(Port *p, void *msg) {
  int minum= obtenerNumero(p);
  depositarMensaje(p, msg, minum);
  esperarReceive(p, minum);
}

int obtenerNumero(p) {
  int minum;
  nEnter(p->mon);
  minum= p->num++;
  nNotifyAll(p->mon);
  nExit(p->mon); /* sacar en version optimizada */
  return minum;
}
```

```
void depositarMensaje(Port *p, void *msg, int minum) {
  nEnter(p->mon);  /* sacar en version optimizada */
  while (minum!=p->visor) /* Asegurarse que sea mi turno */
    nWait(p->mon);
  p->msg= msg;
  nExit(p->mon);  /* sacar en version optimizada */
}

void esperarReceive(Port *p, int minum) {
  nEnter(p->mon);  /* sacar en version optimizada */
  while (minum==p->visor) /* esperar a que el receptor saque el msg */
    nWait(p->mon);
  /* nNotifyAll: no es necesario porque no cambia el estado */
  nExit(p->mon);
}

void *receive(Port *p) {
  void *msg;

  nEnter(p->mon);

  while (p->num==p->visor) /* asegurar que haya un emisor */
    nWait(p->mon);

  msg= p->msg; /* sacar el mensaje */
  p->visor++;  /* ahora le toca el turno a otro */
  nNotifyAll(p->mon);
  nExit(p->mon);

  return msg;
}
```