CC4302 Sistemas Operativos – Control 2 – Semestre Primavera 2016 – Prof.: Luis Mateu

Pregunta 1

Parte a.- (4 puntos) El siguiente código es una implementación incorrecta e ineficiente de un sistema de subastas para múltiples threads:

```
typedef enum {ADJUD, RECHAZ, SINRES} Resol;
typedef struct {
  PriQueue c; // cola de prioridades
  int n;      // número de unidades
} *Subasta;
typedef struct {
  double oferta; // dinero ofrecido
  Resol resol;   // estado de la oferta
} Req;
Subasta nuevaSubasta(int n) {
  Subasta s= malloc(sizeof(*s));
  s->n= n;
  s->c= MakePriQueue();
  return s;
}
Resol ofrecer(Subasta s, double oferta) {
  Req r= {oferta, SINRES};
  PriPut(s->c, &r, oferta);
  if (PriSize(s->c)>s->n) {
    Req *pr= PriGet(s->c);
    pr->resol= RECHAZ; // se rechaza
  }
  while (r.resol == SINRES)
    ;
  return r.resol;
}
double adjudicar(Subasta s, int *punid) {
  double recaud= 0.0;
  *punid= 0;
  while (!EmptyPriQueue(s->c)) {
    Req *pr= PriGet(s->c);
    pr->resol= ADJUD; // se adjudica
    (*punid)++;
    recaud += pr->oferta;
  }
  s->n -= *punid;
  return recaud;
}
```

Este código por sí solo debe ser suficiente para que Ud. comprenda qué es lo que debe hacer el sistema de subastas. Estúdielo detenidamente y descubrirá lo siguiente. La implementación funciona correctamente si las operaciones no se invocan simultáneamente. Un propietario crea su subasta invocando *nuevaSubasta*. En *n* indica el número de unidades idénticas del producto que subastará. Múltiples threads invocan concurrentemente *ofrecer* para hacer una oferta por una unidad de la subasta *s*. El dinero ofrecido se indica en *oferta*. La función *ofrecer* espera hasta que se resuelva si se adjudicó o no la unidad. Esto ocurre cuando se llega a los *n* threads que ya hicieron una oferta mayor (por simplicidad suponga que todas las ofertas son distintas) en cuyo caso se retorna *RECHAZ*; o cuando el propietario llama a *adjudicar* y en ese caso *ofrecer* retorna *ADJUD*. La función *adjudicar* retorna el total recaudado y entrega el número de unidades que se subastaron en **punid* (menor que *n* si no hay suficientes oferentes).

Observe que cuando llega un oferente se usa *PriPut* para encolar un objeto *r* de tipo *Req* en la cola *c* de la subasta. Su prioridad es la oferta. Si se llega a *n+1* oferentes, se extrae con *PriGet* la oferta menor y se rechaza.

Reprograme el código anterior para una máquina con 8 cores físicos sin un núcleo de sistema operativo, y por lo tanto no hay scheduler de procesos. Los 8 cores ejecutan código en paralelo y comparten la memoria. Los spin-locks son la única herramienta disponible para sincronizar los distintos cores. Observe que no hay threads, los oferentes son los distintos cores. Dado que no hay una cola de procesos *ready*, para esperar no le queda otra que hacerlo mediante un spin-lock. No se permiten otras formas de busy-waiting. Ud. sí dispone de *malloc*, del tipo *PriQueue* y sus operaciones.

Parte b.- (1 punto) Si tuviese que programar desde cero un núcleo de sistema operativo para máquinas mono-core, ¿elegiría un núcleo clásico o un núcleo moderno? Explique la razón. ¿Por qué cambiaría su decisión si se tratase de máquinas multi-core?

Parte c.- (1 punto) Un programador plantea eliminar el busy-waiting de los spin-locks de la siguiente manera: cuando se pida un spin-lock que está cerrado, entonces el programador propone suspender el thread que lo pide y retomar otro thread. ¿Cómo le respondería Ud.?

Pregunta 2

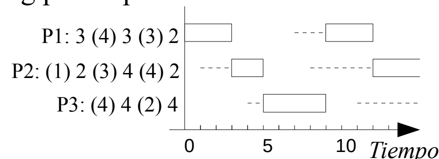
I. (4 puntos) Implemente la siguiente API de forma nativa en nSystem:

```
nSubasta nNuevaSubasta(int n);
Resol nOfrecer(nSubasta s, double oferta);
double nAdjudicar(nSubasta s, int *punid);
```

Debe reimplementar la misma funcionalidad de la pregunta 1 usando las funciones de bajo nivel de nSystem (*START_CRITICAL*, *Resume*, *PutTask*, etc.). Ud. no puede hacer busy-waiting. Sí dispone del tipo *PriQueue*. Sea eficiente evitando cambios de contexto innecesarios.

II. (1 punto) ¿Por qué se inhiben las interrupciones cuando se usan spin-locks para garantizar la exclusión mutua en un núcleo moderno para multi-core?

III. (1 punto) El diagrama parcial muestra las decisiones de scheduling para 3 procesos.



Para cada proceso se indica la duración de la ráfaga y la duración de su estado de espera entre paréntesis. En el diagrama la línea punteada indica que el estado del proceso es READY. Si el proceso está en estado de espera el espacio aparece en blanco. ¿De qué estrategia de scheduling se trata? Rehaga el diagrama completo considerando ahora la estrategia *shortest job first* non preemptive. Considere que el predictor de duración para la próxima ráfaga es la duración de la última ráfaga.