CC41B Sistemas Operativos

Control 2 – Semestre Primavera 2010

Prof.: Luis Mateu

Pregunta 1

Se desea agregar a nSystem un sistema de mensajes *asíncronos*. Esto significa que el emisor de un mensaje no se bloquea a la espera de una repuesta. Este sigue trabajando y puede emitir nuevos mensajes. Los mensajes se encolan en la tarea destinataria hasta que ésta los obtenga. Concretamente las operaciones que Ud. debe programar son:

- `void nPost(nTask t, void *msg)`: Emite el mensaje `msg` con la tarea `t` como destinataria. Este procedimiento encola `msg` en `t` y retorna de inmediato.
- `void* nGetMsg( )`: Lo invoca una tarea para recibir un mensaje pendiente extraído de su cola de mensajes. Los mensajes se entregan en orden FIFO. Si no hay mensajes encolados, la tarea se bloquea a la espera de que otra tarea invoque `nPost`.

Implemente esta API usando los procedimientos de bajo nivel de nSystem (`START_CRITICAL`, `Resume`, `PutTask`, etc.). Ud. *no puede usar* otros mecanismos de sincronización ya disponibles en nSystem, como semáforos, monitores, mensajes, etc. No olvide indicar las modificaciones que Ud. haga al descriptor de tareas de nSystem.

Nótese que no hay límite a la cantidad de mensajes que se pueden emitir. Es responsabilidad del usuario de este sistema de mensajes: (i) evitar agotar la memoria emitiendo mensajes indiscriminadamente sin que sean recibidos, y (ii) evitar liberar la memoria del mensaje antes de que el receptor termine de usar el mensaje.

Pregunta 2

Se le pide a Ud. implementar un sistema de mensajes similar al de nSystem (pero sin *time-outs*) para un multiprocesador que no posee un núcleo de sistema operativo y por ende no existe un *scheduler* de procesos (no hay `Resume`) y no hay herramientas de sincronización avanzadas (como monitores o semáforos). Por lo tanto cuando por ejemplo Ud. envía un mensaje con `send`, no tiene otra alternativa que esperar la respuesta usando un *spin-lock*, la única herramienta de sincronización disponible. A cada procesador (o core) se le asocia un único *port* que sirve como casilla para que ese procesador reciba mensajes. Concretamente las operaciones que Ud. debe implementar son:

- `void initPort(Port p)`: Inicializa el port `p`.
- `void send(Port p, void *msg)`: envía el mensaje `msg` al procesador asociado con el port `p`. Se espera hasta que el mensaje sea respondido con `reply`.
- `void* receive(Port p)`: lo invoca solamente el procesador asociado con el port `p` para recibir un mensaje. Si no hay mensajes pendientes se espera hasta recibir uno. Si hay uno o más mensajes pendientes se recibe solo uno, pero puede ser cualquiera.
- `void reply(Port p)`: lo invoca solamente el procesador asociado con el port `p` para responder el último mensaje recibido con `receive`.

Use la siguiente estructura para representar un Port:

```
typedef struct {
  int rec_sl;    /* spin lock para esperar en receive */
  int send_sl;   /* spin lock para esperar en send */
  int reply_sl; /* spin lock para esperar el reply */
  ...            /* Otros campos que Ud. necesite */
} *Port;
```

A modo de ejemplo de uso, el código de más abajo es una implementación de N productores/1 solo consumidor usando este sistema de mensajes. El *buffer* resultante puede almacenar un solo item.

Productores (varios procesadores)	*Consumidor* (un solo procesador que recibe a través del port `p`)

`for (;;) {`
`  Item it= produce();`
`  send(p, &it);`
`}`
	`for (;;) {`
	`  void *msg= receive(p);`
	`  Item it= *(Item*)msg;`
	`  reply(p);`
	`  consume(it);`
	`}`

Nota: Si Ud. no logra programar una solución correcta, entregue su mejor solución y haga un diagrama de threads mostrando una situación de inconsistencia. Esto será mejor evaluado que una solución incorrecta que se entrega como correcta.