Mensajes

(versión preliminar)

Ejemplo: productor/consumidor con buffer de tamaño 0

```
void produce(Item *p_it);
void consume(Item *p_it);

int nMain() {
  nTask cons= nEmitTask(consProc);
  nTask prod= nEmitTask(prodProc, cons);
  nWaitTask(cons);
}

int prodProc(nTask dest) {
  for (;;) {
    Item it;
    produce(&it);
    nSend(dest, &it);
  }
}

int consProc() {
  for (;;) {
    nTask t;
    Item it= *(Item*)nReceive(&t, -1);
    nReply(t, 0); /* se puede porque el item quedo almacenado localmente */
    consume(&it);
  }
}
```

Cuidado: esta implementación del consumidor sería incorrecta:

```
int consProc() {
  for (;;) {
    nTask t;
    Item *p_it= (Item*)nReceive(&t, -1);
    nReply(t, 0); /* incorrecto! el productor podría destruir *pit */
    consume(p_it);
  }
}
```

El problema está en que el puntero pit apunta hacia la variable it en el productor. Al responder el mensaje con nReply, el nSend en el productor se desbloquea y termina el bloque de la iteración. Con ello se destruye la variable it. Por lo tanto, cuando el consumidor usa *p_it, la variable a la que apunta pudo haber sido destruida.

¿Entonces por qué no responder el mensaje después de consume? De esta forma, el consumidor quedaría como:

```
int consProc() {
  for (;;) {
```

```
    nTask t;
    Item *p_it= (Item*)nReceive(&t, -1);
    consume(p_it);
    nReply(t, 0);
  }
}
```

La implementación sería correcta, pero no habrá paralelismo entre productor y consumidor. Cuando el productor envía el ítem al consumidor, se quedaría bloqueado en el nSend hasta que se haya consumido el ítem, y no podría producir al mismo tiempo que se consume. Entonces producir y consumir ocurrirían secuencialmente y se crearía inútilmente un thread para el consumidor.

Buffer de 1 ítem

Para implementar un buffer de 1 ítemes se crea una tarea intermedia que lo único que hace es recibir un ítem, almacenarlo localmente y luego reenviarlo al consumidor:

```
int bufProc(nTask dest) {
  for (;;) {
    nTask t;
    Item it= *(Item*)nReceive(&t, -1);
    nReply(t, 0); /* se puede porque el item quedo almacenado localmente */
    nSend(dest, &it);
  }
}
```

El nMain se cambia por:

```
int nMain() {
  nTask cons= nEmitTask(consProc);
  nTask buf= nEmitTask(bufProc, cons);
  nTask prod= nEmitTask(prodProc, buf);
  nWaitTask(cons); /* Nunca termina: solo para evitar que nMain termine */
}
```

Ejercicio: cambie solo nMain para implementar un buffer de 2 ítemes.

Equivalencia entre herramientas de sincronización

Las tres herramientas de sincronización vistas en el curso (semáforos, monitores y mensajes) son equivalentes porque cualquier problema que se resuelva con una de ellas se puede resolver con cualquiera de las otras 2. Esto es trivialmente cierto porque cualquier herramienta de sincronización se puede implementar a partir de otra herramienta. Para probar esto mostraremos que (i) los monitores se pueden implementar a partir de semáforos, (ii) los semáforos se pueden implementar a partir de mensajes, y (iii) los mensajes se pueden implementar a partir de monitores.

Monitores a partir de semáforos

```
typedef struct node {
  nSem wait;
  struct node *next;
} Node;

typedef struct {
  nSem mutex;
  Node *head;
```

```
} Monitor;

Monitor *makeMonitor() {
  Monitor *mon= (Monitor*)nMalloc(sizeof(*m));
  mon->mutex= nMakeSem(1);
  mon->head= NULL;
  return mon;
}

void monEnter(Monitor *mon) {
  nSemWait(mon->mutex);
}

void monExit(Monitor *mon) {
  nSemSignal(mon->mutex);
}

void monWait(Monitor *mon) {
  Node node;
  node.wait= nMakeSem(0);
  node.next= mon->head;
  mon->head= &node;
  nSemSignal(mon->mutex);
  nSemWait(node.wait);   /* se bloquea a la espera del notify */
  nDestroySem(node.wait);
  nSemWait(mon->mutex);  /* se bloquea a la espera del monitor */
}

void monNotifyAll(Monitor *mon) {
  Node *node;
  for (node= mon->head; node!=NULL; node= node->next)
    nSignalSem(node->wait);
  mon->head= NULL;
}
```

Semáforos a partir de mensajes

```
typedef struct {
  nTask qtask, server;
} Sem;

typedef enum {SEMWAIT, SEMSIGNAL, SEMDESTROY} SemOp;

typedef struct {
  SemOp op;
  nTask emitter;
} Req;

int semQueueProc(), semServerProc(int ini_tickets);

Sem *makeSem(int ini_tickets) {
  Sem *s= (Sem*)nMalloc(sizeof(*s));
  s->server= nEmitTask(semServerProc, ini_tickets);
  s->qtask= nEmitTask(semQueueProc, s->server);
  return s;
}

void destroySem(Sem *s) {
```

```
  Req req= {SEMDESTROY, nCurrentTask()};
  nSend(s->qtask, &req);
  nWaitTask(s->server);
  nWaitTask(s->qtask);
}

void semWait(Sem *s) {
  Req req= {SEMWAIT, nCurrentTask()};
  nSend(s->qtask, &op);
}

void semSignal(Sem *s) {
  Req req= {SEMSIGNAL, nCurrentTask()};
  nSend(s->server, &op);
}

int semQueueProc(nTask server) {
  int finish= FALSE;
  do {
    Req *req= (Req*)nReceive(NULL, -1);
    if (req->op==SEMDESTROY)
      finish= TRUE;
    nSend(server, req);
  } while (!finish);
  return 0;
}

int semServerProc(int tickets) {
  int finish= FALSE;
  do {
    nTask t;
    Req *req= (Req*)nReceive(&t, -1);
    if (req->op==SEMDESTROY)
      finish= TRUE;
    else if (req->op==SEMSIGNAL)
      tickets++;
    else if (req->op==SEMWAIT) {
      if (tickets>0)
        tickets--;
      else {
        nTask signaltask;
        Req *sigrec= (Req*)nReceive(&signaltask, -1);
        if (sigrec!=SEMSIGNAL)
          nFatalError("semServerProc", "se esperaba un SEMSIGNAL\n");
        nReply(signaltask, 0);
      }
      nReply(req->emitter, 0);
    }
    nReply(t, 0);
  } while (!finish);
  return 0;
}
```

En este código, la tarea qtask hace las veces de cola de solicitudes de tickets. En efecto, observe en el código del servidor que cuando llega una solicitud de ticket (SEMWAIT), el servidor se queda esperando solo una notificación de depósito de ticket (SEMSIGNAL) y no se pone en el caso de que llegue otra solicitud de ticket. Esto se puede hacer porque al no responder el mensaje *req* proveniente de *qtask*, esa tarea no puede reenviar un nuevo mensaje, y cualquier solicitud de ticket tiene que pasar por *qtask*. La pregunta entonces es ¿qué pasa con esas nuevas solicitudes de ticket? En realidad quedan encoladas en la cola de mensajes de *qtask* y serán procesados uno a uno a medida que *qtask* pueda recibirlos.

En este caso no es muy difícil escribir una versión que no use *qtask*. Todas las solicitudes de ticket deberán ser recibidas directamente por el servidor. Si no hay tickets disponibles esas solicitudes se deben encolar explícitamente en alguna estructura de datos. Haga el ejercicio usando el tipo FifoQueue incluido en nSystem.

Mensajes a partir de monitores

Ejercicio propuesto.

Lectores/escritores con mensajes

```
typedef enum { ENTERREAD, EXITREAD, ENTERWRITE, EXITWRITE } RWOp;

typedef struct {
  RWOp op;
  nTask emitter;
} Req;

static nTask qtask; /* = nEmitTask(queueProc); */
static server;      /* = nEmitTask(serverProc); */

static void sendOp(Op op, nTask t) {
  Req req= { Op, nCurrentTask() };
  nSend(t, &req);
}

void enterRead() { sendOp(ENTERREAD, qtask); }
void exitRead() { sendOp(EXITREAD, server); }
void enterWrite() { sendOp(ENTERWRITE, qtask); }
void exitWrite() { sendOp(EXITREAD, server); }

static int queueProc() {
  for (;;) {
    Req *req= (Req)nReceive(NULL, -1);
    nSend(server, req);
} }

static int serverProc() {
  int readers= 0;
  nTask aux;
  for (;;) {
    /* Se autorizan lecturas hasta que llegue una solicitud de escritura */
    Req *req= (Req*)nReceive(NULL, -1); /* Estado WAIT */
    while (req->op!=ENTERWRITE) {
      if (req->op==EXITREAD)
        readers--;
      else { /* req->op==ENTERREAD */
        readers++;
        nReply(req->emitter, 0);  /* se autoriza una lectura */
      }
      nReply(qtask, 0);
      req= (Req*)nReceive(NULL, -1); /* Estado READ1 */
    }
    /* la escritura no ha sido autorizada aún */
    while (readers>0) { /* se espera a que salgan los lectores */
      nReceive(&aux, -1); /* Estado READ2: tiene que llegar un EXITREAD */
      nReply(aux, 0);
      readers--;
    }
    nReply(req->emitter, 0); /* se autoriza la escritura */
    nReceive(&aux, -1); /* Estado WRITE: tiene que llegar un EXITWRITE */
```

```
      nReply(aux, 0);
  } }
```

Aquí nuevamente qtask hace las veces de cola de solicitudes de entrada. Haga el ejercicio de no usar qtask y manejar explícitamente una cola de solicitudes de entrada en el servidor y va a ver que resulta bastante más complicado que la versión de más arriba.

Ejercicio: buffer de tamaño N

Usando una metodología similar a la anterior programe un buffer de tamaño N para el problema del productor/consumidor. Use 2 procesos "queue": uno para los *get* y el segundo para los *put*. Haga el diagrama de estados para el servidor.