Procesos

Un *proceso* es un *programa en ejecución*. Este programa se ejecuta en un procesador, tiene su código, datos, pila, un contador de programa, un puntero a la pila, otros registros y descriptores de entrada/salida (como el fd entregado por *open*).

Un sistema operativo es *multiproceso* cuando puede ejecutar varios procesos concurrentemente, aunque no necesariamente en paralelo. En este caso el procesador asignado a cada proceso es virtual y el núcleo multiplexa el o los procesadores reales disponibles (*cores*) en tajadas de tiempo que se otorgan por turnos a los distintos procesos.

Cuando el computador es *multiprocesador* (también se dice *multicore*) y el sistema operativo está diseñado para explotar todos los procesadores disponibles, entonces los procesos se pueden ejecutar efectivamente en paralelo.

Clasificación de procesos

Procesos pesados versus procesos livianos

Los procesos que implementa un sistema operativo se clasifican según el grado en que comparten la memoria (ver figura):

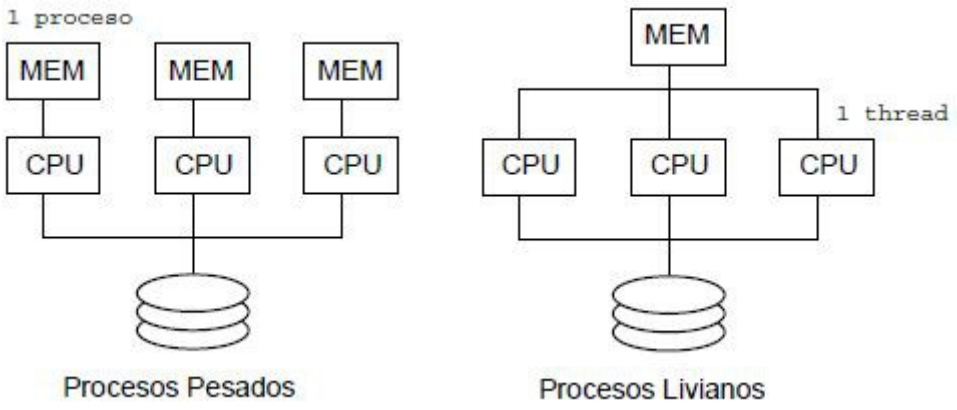

Figure: Procesos pesados y livianos

- Procesos Pesados (proceso Unix): Los procesos no comparten ninguna porción de la memoria. Cada proceso se ejecuta en su propio procesador virtual con CPU y memoria. Todos los procesos sí comparten el mismo espacio de almacenamiento permanente (el disco).

- Procesos Livianos o *threads*: Los threads comparten toda la memoria y el espacio de almacenamiento permanente.

El primer tipo de procesos se dice *pesado* porque el costo de implementación en tiempo de CPU y memoria es mucho más elevado que el de los procesos *livianos*. Además la implementación de procesos pesados requiere de una MMU o Unidad de Manejo de la Memoria. Esta componente de hardware del procesador se encarga de la traducción de direcciones virtuales a reales. La implementación en software de esta traducción sería demasiado costosa en tiempo de CPU, puesto que para garantizar una verdadera protección habría que recurrir a un intérprete del lenguaje de máquina.

Unix estándar sólo ofrece procesos pesados, pero como veremos existen extensiones que implementan procesos livianos para Unix. Un ejemplo de sistema de procesos livianos es el que implementaba el sistema operativo de los antiguos computadores Commodore Amiga (1985), que no tenía la MMU necesaria para implementar procesos pesados.

La ventaja de los procesos pesados es que garantizan protección. Si un proceso falla, los demás procesos continúan sin problemas. En cambio si un thread falla, esto causa la falla de todos los demás threads que comparten el mismo procesador.

La ventaja de los threads es que pueden comunicarse eficientemente a través de la memoria que comparten. Si se necesita que un thread comunique información a otro thread basta que le envíe un puntero a esa información. En cambio los procesos pesados necesitan enviar toda la información a otros procesos pesados usando *pipes*, *sockets*, mensajes o archivos en disco, lo que resulta ser más costoso que enviar tan solo un puntero.

Preemption versus non-preemption

Los procesos también se pueden clasificar según quién tiene el control para transferir el procesador multiplexado de un proceso a otro:

- Procesos *preemptive* (con adelantamiento): Es el núcleo el que decide en qué instante se transfiere el procesador real de un proceso al otro. Es decir un proceso puede perder el control del procesador en cualquier instante.

- Procesos *non-preemptive* (sin adelantamiento): Es el proceso el que decide en que momento se puede transferir el procesador real a otro proceso.

Los procesos de Unix son preemptive. Ejemplos de procesos non preemptive son los que antiguamente ofrecía Windows hasta la versión 3.X o MacOS hasta la versión 6.X. En Windows 3.X si una aplicación entraba en un ciclo infinito, no había forma de quitarle el procesador real, la única salida era el relanzamiento del sistema operativo completo. Lo mismo ocurría en MacOS 6.X. Los procesos necesitaban ser non-preemptive para asegurar el funcionamiento adecuado de las aplicaciones que databan de cuando estos mismos sistemas operativos eran mono-aplicación.

El nano System

Dado que en Unix un proceso pesado corre en un procesador virtual es posible implementar un sistema de threads que comparten el mismo espacio de direcciones virtuales en un proceso Unix. Un ejemplo de estos sistemas es el nano System (nSystem de ahora en adelante).

En vez de usar la palabra thread -que es difícil de pronunciar en castellano- usaremos como sinónimo la palabra tarea. Las tareas de nSystem se implementan multiplexando el tiempo de CPU del proceso Unix en tajadas de tiempo que se otorgan por turnos a cada tarea, de la misma forma en que Unix multiplexa el tiempo del procesador real para otorgarlo a cada proceso pesado.

De esta forma varios procesos Unix puede tener cada uno un enjambre de tareas que comparten el mismo espacio de direcciones, pero tareas pertenecientes a procesos distintos *no comparten la memoria*.

Gestión de tareas en nSystem

Los siguientes procedimientos de nSystem permiten crear, terminar y esperar tareas:

- `nTask nEmitTask( int (*proc)(...), ...)`: Emite o lanza una nueva tarea que ejecuta el procedimiento `proc`. Acepta un máximo de 6 parámetros (enteros o punteros) que son traspasados directamente a `proc`. Retorna un descriptor de la tarea lanzada.

- `void nExitTask(int rc)`: Termina la ejecución de la tarea que lo invoca, `rc` es el código de retorno de la tarea.

- `int nWaitTask(nTask task)`: Espera a que una tarea termine, entrega el código de retorno dado a `nExitTask`.

- `void nExitSystem(int rc)`: Termina la ejecución de todas las tareas. Es decir, termina el proceso Unix con código de retorno `rc`.

En la siguiente figura se observa que si una tarea A espera otra tarea B antes de que B haya terminado, A se bloquea hasta que B invoque `nExitTask`. De la misma forma, si B termina antes de que otra tarea invoque `nWaitTask`, B no termina -permanece bloqueada- hasta que alguna tarea lo haga. Por esta razón se necesita que toda tarea T lanzada con nEmitTask tenga un y solo un nWaitTask(T) asociado. Es el nWaitTask(T) el que libera el descriptor de tarea que representa T. La existencia de un segundo nWaitTask(T) sería un error porque el descriptor T ya no existiría. La situación es similar a la de *malloc* y *free* en el lenguaje C. Todo puntero P entregado por malloc debe tener un y solo un free(P) asociado.

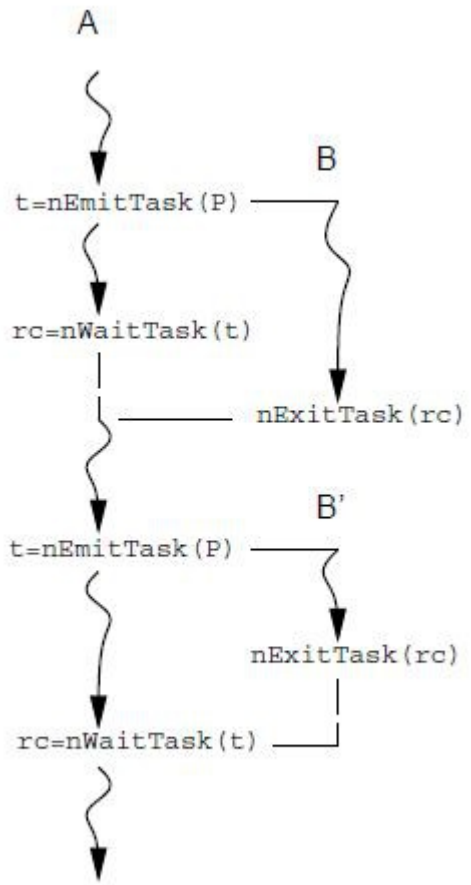

Figura: Creación, espera y término de tareas en nSystem

A modo de ejemplo veremos como se calcula un número de Fibonacci usando tareas de nSystem. Pero antes

veamos la versión secuencial:

```
int fib(int n) {
  if (n<=1)
    return 1;
  else
    return fib(n-1)+fib(n-2);
}
```

La siguiente función realiza el mismo cálculo pero usando 2 tareas de nSystem:

```
int pfib(int n) {
  if (n<=1)
    return 1;
  else {
    nTask task1= nEmitTask(fib, n-1);
    nTask task2= nEmitTask(fib, n-2);
    return nWaitTask(task1)+nWaitTask(task2);
  }
}
```

La ejecución de `pfib(6)` crea dos tareas que calculan *concurrentemente* `fib(5)` y `fib(4)`. Este ejemplo tiene solo fines pedagógicos porque el tiempo de ejecución no será menor que el tiempo de la versión secuencial. Esto se debe a que la implementación actual de nSystem no ejecuta en paralelo estas 2 tareas. Aún en una máquina dual-core solo se usa un solo core. De hecho el tiempo de ejecución será algo superior debido al sobrecosto de creación de las tareas. Si queremos disminuir ese sobrecosto podemos reusar la tarea original para hacer uno de los cálculos:

```
int pfib(int n) {
  if (n<=1)
    return 1;
  else {
    nTask task= nEmitTask(fib, n-1);
    return nWaitTask(task)+fib(n-2);
  }
}
```

Consideremos ahora la hipotética existencia de una implementación multicore de nSystem, es decir en donde las tareas sí se ejecutan en paralelo. Supongamos que se dispone de 16 cores. Entonces uno estaría tentado de aprovechar todos los cores disponibles de la siguiente manera:

```
int pfib(int n) {
  if (n<=1)
    return 1;
  else {
    nTask task= nEmitTask(pfib, n-1);
    return nWaitTask(task)+pfib(n-2);
  }
}
```

Sin embargo, esta versión sería varios órdenes de magnitud *más ineficiente* que cualquiera de las versiones anteriores porque crea demasiadas tareas. De hecho crea una tarea por cada suma que se necesite. El problema está en que el sobrecosto de crear una tarea es altísimo comparado con realizar una sola suma (varias decenas de microsegundos versus una fracción de nanosegundo). En buenas cuentas no vale la pena crear una tarea cuando el trabajo útil que ésta realizará toma menos tiempo que el de crear esa misma tarea.

Por lo tanto en los problemas de paralelización siempre se debe contemplar un mecanismo que limite la cantidad de tareas de tal forma que no se supere en exceso la cantidad de cores disponibles o al menos el trabajo útil de cada tarea sea muy superior al tiempo de creación de la tarea.

Ejemplo: factorial paralelo

Considerando una implementación multicore de nSystem, se necesita paralelizar el cálculo del factorial de un entero, creando a lo más P tareas. El problema consiste en calcular el producto de los números entre 1 y n:

$$1 * 2 * 3 * \ldots * n/2 * (n/2+1) * \ldots * n$$

Por lo tanto lo más simple es usar dividir para reinar y calcular el producto de los enteros entre *1* y *n/2* y multiplicarlo por el producto de los enteros entre *(n/2+1)* y *n*. La función *producto* realizará este trabajo, entregando el resultado en el contenido del puntero que recibe como último argumento:

```
#define P 16   /* a lo más 16 tareas */

double fact(int n) {
  double prod;
  producto(1, n, &prod, P);
  return prod;
}
```

La función producto no puede simplemente retornar el resultado porque usaremos esta función como parámetro de *nEmitTask*, el que requiere una función que retorne un entero. La estructura de producto es similar a la de *pfib* en el problema anterior.

```
int producto(int i, int j, double *res, int p) {
  if (p==1 || j-i<20) { /* Ver explicacion */
    int k;
    double prod= 1.0;
    for (k= i; k<= j; k++)
      prod *= k;
    *res= prod;
  }
  else {
    double prod1, prod2;
    nTask t= nEmitTask(producto, i, (i+j)/2, &prod1, p/2);
    producto((i+j)/2+1, j, &prod2, p-p/2);
    nWaitTask(t);
    *res= prod1 * prod2;
  }
  return 0; /* se necesita para que el compilador no reclame */
}
```

El primer *if* determina si el producto se realizará secuencialmente o en paralelo. Como hay que crear a lo más *p* tareas, a una de las llamadas recursivas de producto le pedimos que cree la mitad de *p* tareas y a la otra llamada lo que queda para llegar a *p*. La recursividad se detiene por un lado cuando *p* llega a 1, pero también se detiene si el trabajo pedido es demasiado pequeño como para crear múltiples tareas y esto es cuando la distancia entre *j* e *i* es inferior a 20.

El problema de la sección crítica

Se propone la siguiente variación *incorrecta* de la solución anterior. La idea es utilizar una sola variable declarada en *fact* para ir acumulando el producto. En la función *producto* no se declaran nuevas variables como *prod*, *prod1* o *prod2*:

```
int producto(int i, int j, double *res, int p) { /* **Versión incorrecta** */
  if (p==1 || j-i<20) {
    int k;
    double prod= 1.0;
```

```
    for (k= i; k<= j; k++)
      *res = *res * k; /* ver explicación */
  }
  else {
    nTask t= nEmitTask(producto, i, (i+j)/2, res, p/2);
    producto((i+j)/2+1, j, res, p-p/2);
    nWaitTask(t);
  }
  return 0; /* se necesita para que el compilador no reclame */
}
```

Esta solución es errada porque 2 tareas pueden llegar a ejecutar en paralelo la asignación a **res*, de una manera tal que pueden llevar la variable **res* a un valor inconsistente. Por ejemplo, supongamos por simplicidad que *res vale inicialmente 1 y que en una tarea *k* vale 2 y en la otra 3. El valor esperado para **res* es 6, pero como se observa en la siguiente ejecución representada mediante una tabla, el valor final de *k* es distinto. En la tabla las filas están ordenadas cronológicamente. Observe que **res* es compartido por las tareas, mientras que *r1* y *r2* son registros del procesador (virtual), que no son compartidos con los registros de T2 (*r1'* y *r2'*).

T1	*valor*	*T2*	*valor*
:		:	
r1= *res	1	:	
r2= r1 * k	2	r1'= *res	1
*res= r2;	2	r2'= r1' * k'	3
		*res= r2'	3

El valor de k termina en 3 y no en 6 como debería ser. En programación concurrente este error se denomina *datarace*. Este nombre es una metáfora de un *signal race* que se aplica en el dominio de la electrónica. Una *sección crítica* es un conjunto de código que sólo puede ser ejecutada por una tarea a la vez para que funcione correctamente. De otra forma podría producirse un *datarace*. Es decir, se necesita que las tareas se ejecuten en *exclusión mutua* en la sección crítica. Para lograr la exclusión mutua se requiere usar herramientas de sincronización como semáforos, monitores, *mutex* o *locks*, que estudiaremos durante el curso. Por lo pronto en el próximo capítulo usaremos semáforos para garantizar la exclusión mutua.

Otros procedimientos de nSystem

Durante el lanzamiento de nSystem se realizan algunas inicializaciones y luego se invoca el procedimiento `nMain`, el que debe ser suministrado por el programador. Éste procedimiento contiene el código correspondiente a la primera tarea del programador. Desde ahí se pueden crear todas las tareas que sean necesarias para realizar el trabajo. El encabezado para este procedimiento debe ser:

```
int nMain(int argc, char *argv[]) {
   ...
}
```

El retorno de `nMain` hace que todas las tareas pendientes terminen y por lo tanto es equivalente a llamar `nExitSystem`. Es importante por lo tanto evitar que `nMain` se termine antes de tiempo.

Los siguientes procedimientos son parte de nSystem y pueden ser invocados desde cualquier tarea de nSystem.

Parámetros para las tareas:

- `void nSetStackSize(int new_size)`: Define el tamaño de la pila para las tareas que se emitan a continuación.

- `void nSetTimeSlice(int slice)`: Tamaño de la tajada de tiempo para la administración *Round-Robin* (preemptive) (el `slice` está en miliseg). Degenera en *FCFS* (non-preemptive) si `slice` es cero, lo que es muy útil para depurar programas. El valor por omisión de la tajada de tiempo es cero.

- `void nSetTaskName(char *format, <args>, ... )`: Asigna el nombre de la tarea que la invoca. El formato y los parámetros que recibe son análogos a los de `printf`.

Entrada y Salida:

Los siguientes procedimientos son equivalentes a `open`, `close`, `read` y `write` en UNIX. Sus parámetros son los mismos. Los "nano" procedimientos sólo bloquean la tarea que los invoca, el resto de las tareas puede continuar.

- `int nOpen(char *path, int flags, int mode)`: Abre un archivo.

- `int nClose(int fd)` : Cierra un archivo.

- `int nRead(int fd, char *buf, int nbyte)` : Lee de un archivo.

- `int nWrite(int fd, char *buf, int nbyte)` : Escribe en un archivo.

Servicios Misceláneos:

- `nTask nCurrentTask(void)` : Entrega el identificador de la tarea que la invoca.

- `int nGetTime(void)` : Entrega la ``hora'' en miliseg.

- `void *nMalloc(int size)`: Es un `malloc` non-preemptive.

- `void nFree(void *ptr)`: Es un `free` non-preemptive.

- `void nFatalError(char *procname, char *format, ... )`: Escribe salida formateada en la salida estándar de errores y termina la ejecución (del proceso Unix). El formato y los parámetros que recibe son análogos a los de `printf`.

- `void nPrintf(char *format, ... )`: Es un `printf` sólo bloqueante para la tarea que lo invoca.

- `void nFprintf(int fd, char *format, ... )`: Es ``como'' un `fprintf`, sólo bloqueante para la tarea que lo invoca, pero recibe un fd, no un `FILE *`.

La API completa de nSystem se encuentra en el archivo *nSystem.h* ubicado en el directorio *include* de la distribución de nSystem.

Importante: las funciones de Unix, como *malloc*, *printf*, *random*, etc. pueden producir *dataraces* o *deadlocks* en nSystem si se ejecutan concurrentemente desde 2 tareas. Por ello evite usarlas. Si necesita una función de

Unix que nSystem no ofrece, use un semáforo para garantizar la exclusión mutua durante su invocación.

Semáforos

Un *semáforo* es un distribuidor de tickets y sirve para acotar el número de tareas que pasan por una determinada instrucción. Para ello un semáforo **S** dispone de una cierta cantidad de *tickets* a repartir.

Las operaciones que acepta un semáforo son:

- Wait(S): Pide un ticket al semáforo. Si el semáforo no tiene tickets disponibles, el proceso se bloquea hasta que otro proceso aporte tickets a ese mismo semáforo.

- Signal(S): Aporta un ticket al semáforo. Si había algún proceso esperando un ticket, ese proceso se desbloquea. Si había más procesos esperando tickets, se desbloquea el primero que llegó pidiendo tickets.

En nSystem los semáforos se crean con:

```
nSem sem= nMakeSem(inittickets);
```

En donde `inittickets` es la cantidad inicial de tickets. Las operaciones Wait y Signal se llaman `nWaitSem` y `nSignalSem` respectivamente. Además un semáforo se destruye con `nDestroySem`.

Uno de los usos de los semáforos es garantizar exclusión mutua en secciones críticas. En la sección anterior vimos una versión incorrecta de la función producto ya que puede producir *dataraces*. A continuación usamos un semáforo para asegurar la exclusión mutua:

```
double fact(int n) {
  double prod;
  nSem mutex= nMakeSem(1);
  producto(1, n, &prod, P, mutex);
  return prod;
}

int producto(int i, int j, double *res, int p, nSem mutex) {
  if (p==1 || j-i<20) {
    int k;
    double prod= 1.0;
    for (k= i; k<= j; k++) {
      nWaitSem(mutex);
      *res = *res * k; /* ver explicación */
      nSignalSem(mutex);
    }
  }
  else {
    nTask t= nEmitTask(producto, i, (i+j)/2, res, p/2, mutex);
    producto((i+j)/2+1, j, res, p-p/2, mutex);
    nWaitTask(t);
  }
  return 0; /* se necesita para que el compilador no reclame */
}
```

El semáforo se llama mutex con el objeto de indicar que es usado para asegurar la exclusión mutua. Inicialmente contiene un solo ticket. La primera tarea que llega a la sección crítica -es decir la asignación de **pres*- pide el único ticket, de modo que ninguna otra tarea puede entrar a la sección crítica. Al salir la tarea devuelve el ticket para que otra tarea pueda entrar a la sección crítica.

Con esta modificación la solución es correcta, pero no es eficiente. El problema es que la sincronización tiene un sobrecosto asociado. Pedir un semáforo es caro, especialmente cuando hay contención, es decir cuando es frecuente que las tareas tengan que bloquearse a la espera del ticket. Por esta razón la primera solución que se vió para el cálculo del factorial es la mejor. En general son más eficientes las soluciones que evitan compartir datos en lo posible, porque así no se requiere sincronizar. Pero recuerde que nunca se debe sacrificar la correctitud de una solución por ganar un poco de eficiencia.

El problema del productor/consumidor

En una categoría importante de problemas se puede identificar la necesidad de producir ítemes para luego consumirlos. Podemos modelar la solución de estos problemas por medio del siguiente código:

```
for(;;) { /* Ciclo infinito */
  Item x= produce();
  consume(x);
}
```

En este problema la producción de un ítem es completamente independiente de su consumo y por lo tanto se puede efectuar en paralelo. Para paralelizar ese problema se necesitan 2 tareas, una tarea productora y una tarea consumidora, más un depósito de ítemes que llamaremos *buffer*. La tarea productora itera produciendo ítemes y los deposita en el buffer, mientras que la tarea consumidora los extrae para luego consumirlos. Este es el código de ambas tareas.

Productor	***Consumidor***

`for (;;) {`
`  Item x= produce();`
`  put(x); /* deposita un ítem */`
`}`
	`for (;;) {`
	`  Item x= get();`
	`  consume(x); /* extrae un ítem */`
	`}`

Restricciones con respecto al buffer:

- *get* debe entregar los ítemes en el orden en que fueron depositados en el buffer con *put*
- si el buffer está vacío, *get* debe esperar hasta que se deposite un ítem
- el buffer tiene una capacidad máxima para almacenar *N* ítemes
- si el buffer está lleno, *put* debe esperar hasta que se extraiga un ítem

Solución incorrecta

El buffer se mantiene en un arreglo de `N` elementos de tipo `Item`. La variable `nextempty` indica el índice en el arreglo en donde hay que depositar el próximo ítem producido. La variable `nextfull` indica el índice en el arreglo de donde hay que extraer el próximo ítem por consumir. Esta estructura se aprecia en la siguiente figura. El buffer es circular, es decir cuando se depositó un ítem al final del arreglo, el próximo ítem se depositará al comienzo.

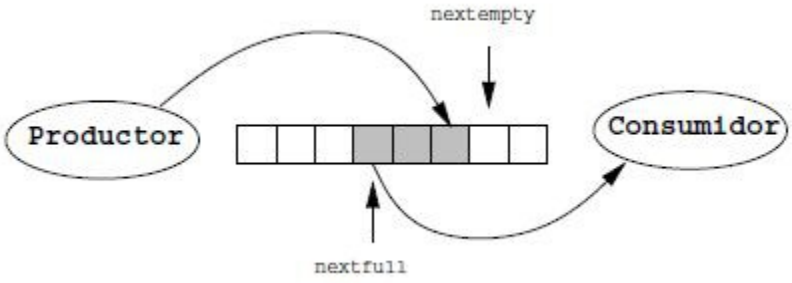

Figura: El problema del productor/consumidor

El siguiente código es una implementación *incorrecta* de put y get.

```
                Item buf[N];
                int nextempty= 0, nextfull= 0, count= 0;
```

```
void put(Item x) {
  while (count==N)
    ; /* busy-waiting */
  buf[nextempty]= x;
  nextempty= (nextempty+1)%N;
  count++;
}
```

```
Item get() {
  Item x;
  while (count==0)
    ; /* busy-waiting */
  x= buf[nextfull];
  nextfull= (nextfull+1)%N;
  count--;
}
```

Esta es una mala solución por varios motivos. Primero, y el más importante, porque sufre de dataraces y por lo tanto es incorrecta. Es fácil mostrar con un diagrama de tareas que el incremento/decremento de *count* puede llevarlo a un valor inconsistente. Observe que el acceso a *nextempty* y *nextfull* no produce dataraces porque estas variables no son compartidas por la tarea productora y la tarea consumidora (aunque sí habría dataraces cuando hay varios productores o varios consumidores). Por el mismo motivo el acceso al arreglo tampoco produce dataraces.

El segundo motivo es que esta es una solución ineficiente debido al *busy-waiting*. En estos ciclos se ocupa el 100% de la CPU. En un sistema monocore, si el consumidor ocupa el 100% de la única CPU en el ciclo de busy-waiting, entonces menos oportunidad va a tener el productor de acceder a la CPU para producir un ítem. Un objetivo importante de este curso es que los alumnos aprendan a resolver problemas de sincronización sin recurrir al busy-waiting y por lo tanto su uso en controles o tareas no entrega puntaje alguno. Más tarde veremos que para implementar las herramientas de sincronización como semáforos y monitores sí se requiere usar busy-waiting, pero se hace de un modo que la espera esté acotada en el tiempo. Por lo mismo, a veces será necesario resolver un problema de control usando busy-waiting, pero esto estará indicado en el enunciado.

Por último, esta solución podría terminar en un ciclo infinito dependiendo de las optimizaciones que efectúe el compilador. Esto se debe a que los compiladores tratan de llevar las variables a registros porque que el acceso a estos últimos es más eficiente que el acceso a memoria. El problema es que el compilador podría colocar el valor de *count* en un registro que no es compartido entre las tareas. Esto haría que el registro nunca sea actualizado aún cuando la otra tarea modifique la variable *count* en memoria, gatillando el ciclo infinito. En todo caso esto se puede resolver fácilmente colocando el atributo volatile a la variable count:

```
volatile int count= 0;
```

El atributo le prohibe al compilador mantener *count* en un registro. En cada iteración del ciclo de busy-waiting se leerá su valor de la memoria, con lo que tarde o temprano leerá el nuevo valor que almacenó la otra tarea.

Solución correcta

La primera solución correcta que veremos de este problema usa dos *semáforos*. Un semáforo mantendrá un ticket por cada casilla vacía en el buffer. Cuando el productor deposita un ítem resta un ticket con Wait, dado que una casilla vacía en el buffer se habrá lleando. Si el buffer está lleno, el productor se bloqueará ya que no haya más tickets en el semáforo. Cuando el consumidor extrae un elemento agrega un ticket con Signal, porque ahora hay una nueva casilla vacía.

El segundo semáforo mantiene un ticket por cada casilla llena en el buffer. El productor agrega tickets con Signal y el consumidor quita tickets -si hay disponibles- con Wait.

```
Item buf[N];
int nextempty= 0, nextfull= 0, count= 0;
nSem empty; /* = nMakeSem(N); */
nSem full; /* = nMakeSem(0); */
```

```
void put(Item x) {
  nWaitSem(empty);
  buf[nextempty]= x;
  nextempty= (nextempty+1)%N;
  nSignalSem(full);
}
```

```
Item get() {
  nWaitSem(full);
  x= buf[nextfull];
  nextfull= (nextfull+1)%N;
  nSignalSem(empty);
}
```

Observe que en C está prohibido inicializar una variable global con un valor que no es una constante. Por eso en el código se colocó en un comentario la inicialización con la llamada a nMakeSem, que necesita la respectiva variable global (normalmente en el nMain).

Uno de los aspectos interesantes del problema de productor/consumidor es que las soluciones son simétricas como se aprecia en el código de más arriba. Observe que las modificaciones de las variables `nextempty` y `nextfull` no son secciones críticas, puesto que cada variable es modificada por una sola tarea. Sin embargo si generalizamos el problema a varios productores y consumidores, estas modificaciones sí serán secciones críticas.

Ejercicio: Modifique esta solución para que funcione en el caso de varios productores y consumidores. Para ello use uno o 2 semáforos adicionales para garantizar la exclusión mutua del acceso a `nextempty` y `nextfull`.

El problema de la cena de filósofos

Este problema también es conocido como el restaurant chino. Consiste en 5 filósofos que van a un restaurant chino a comer arroz. El dueño del restaurant es un sabio chino y disfruta planteándole problemas desafiantes a los filósofos. Por ello los sienta en un mesa circular con 5 sillas pero solo 5 palitos. Cada filósofo ocupa una de la sillas y se dedica a comer y a pensar. Para pensar no necesita ningún palito, pero para comer necesita los 2 palitos más cercanos a su silla.

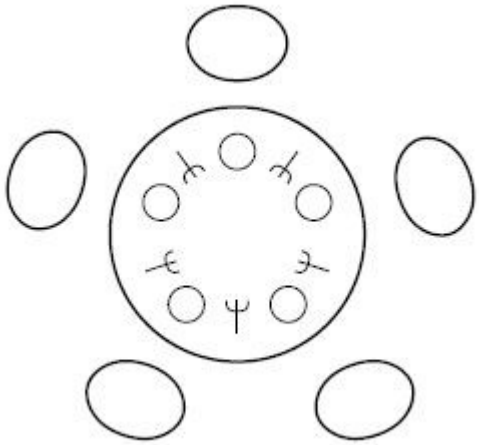

Figura: El problema de los filósofos comiendo.

Los filósofos se enumeran de 0 a 4 al igual que los palitos. El filósofo *i* necesita los palitos *i* e *(i+1)%5* para poder comer.

La restricción más importante en este problema es que un palito no puede ser ocupado simultáneamente por 2 filósofos. En términos computacionales esto representa un datarace. La siguiente es una solución *incorrecta* porque viola esta retricción, pero sirve para modelar el comportamiento de los filósofos por medio de tareas de nSystem.

```
void filosofo(int i) { /* solución incorrecta */
  for(;;) {
    comer(i, (i+1)%5);
    Pensar();
} }
```

En una solución correcta, cuando un filósofo necesita un palito, debe esperar a que este se desocupe. Si logró tomar uno de los palitos, el filósofo puede mantenerlo mientras espera el segundo palito, como también puede soltarlo para que su otro vecino pueda comer.

Este es el más clásico de los problemas de sincronización debido a que es muy usual encontrar problemas equivalentes en los sistemas concurrentes y distribuidos. Lamentablemente no siempre es sencillo detectar su presencia, por lo que también es causa de muchas de las caídas de estos sistemas. Usualmente el problema también se explica con tenedores para comer un plato de tallarines en vez de arroz. Personalmente me gusta más explicarlo con 2 palitos para comer arroz porque nadie come tallarines con 2 tenedores.

Segunda solución incorrecta

Cada filósofo es una tarea. El uso de cada palito es una sección crítica, ya que dos filósofos no pueden estar usando el mismo palito en un instante dado. El acceso a los palitos se controla en un arreglo de 5 semáforos con un ticket cada uno. Para poder comer, el filósofo `i` debe pedir el ticket del semáforo `i` y el del semáforo `(i+1)%5`.

Inicialmente se tiene:

```
nSem palitos[5]; /* = nMakeSem(1); x 5 */

void filosofo(int i) {
  for(;;) {
    nWaitSem(palitos[i]);
```

```
    nWaitSem(palitos[(i+1)%5]);
    comer(i, (i+1)%5);
    nSignalSem(palitos[i]);
    nSignalSem(palitos[(i+1)%5]);
    pensar();
} }
```

Observe que en esta solución, cuando un filósofo tomó un palito no lo suelta a pesar de que tenga que esperar por el próximo palito.

Esta solución es incorrecta porque los filósofos pueden llegar a una situación de bloqueo eterno, o en inglés *dead-lock*. Esta situación se produce cuando cada filósofo está en poder de un palito y espera a que el otro se desocupe, lo que nunca ocurrirá. En estas condiciones todos los filósofos morirán de hambre.

Primera solución correcta

Se tiene especial cuidado al tomar los palitos, de modo que siempre se tome primero el palito con menor índice en el arreglo de semáforos. El código para tomar los palitos queda:

```
j= min(i,(i+1)%5);
k= max(i,(i+1)%5);
nWaitSem(palitos[j]);
nWaitSem(palitos[k]);
```

Con este orden es imposible la situación de *dead-lock* en que cada proceso tenga un palito y espere el otro, puesto que el último palito (índice 4) siempre se pide en segundo lugar. Por lo tanto se garantiza que al menos uno de los filósofos podrá tomar este palito y comer. En el peor de los casos tendrán que comer secuencialmente.

Segunda solución

Permitir un máximo de 4 filósofos en la mesa. Para lograr esto se necesita un nuevo semáforo (`sala`) que parte inicialmente con 4 tickets.

```
Void filosofo(int i) {
  for(;;) {
    nWaitSem(sala);
    nWaitSem(palitos[i]);
    nWaitSem(palitos[(i+1)%5]);
    Comer(i, (i+1)%5);
    nSignalSem(palitos[i]);
    nSignalSem(palitos[(i+1)%5]);
    nSignalSem(sala);
    pensar();
} }
```

Con esta restricción también se garantiza que al menos un filósofo podrá comer. Al resto tarde o temprano le llegará su turno.

El problema de esta solución es que solo se aplica al problema de la cena de filósofos. No sirve en el caso general en donde los palitos representan las distintas estructuras de datos que comparten muchos threads, y en donde una transacción puede requerir acceder a 2, 3 o más estructuras de datos para poder realizarse. En cambio la primera solución sí se adapta igualmente bien al problema más general.

Errores frecuentes en el uso de semáforos

El error más trivial es olvidar depositar el ticket de vuelta en una sección crítica. Esto obviamente se traduce en un deadlock. A veces se deposita el ticket pero en el semáforo equivocado. Pero el error más sutil consiste en incluir la operación que interesa paralelizar en la sección crítica. Por ejemplo, llamar a pensar antes de soltar los palitos. En ese caso se estaría reduciendo inútilmente el paralelismo.

Ejercicio:

- Resuelva el problema 1 del control 1 del semestre Otoño 2013 de sistemas operativos, publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c1-131.pdf.

- Resuelva el problema 1 del control 1 del semestre Otoño 2012 de sistemas operativos, publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c1-121.pdf.