Administración de procesos: Primitivas de sincronización

Gunnar Wolf

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Concurrencia

- No tenemos que preocuparnos cuando todos los datos que maneja un hilo son *locales*
- Al utilizar *variables globales* o recursos externos, debemos recordar que el planificador puede interrumpir el flujo *en cualquier momento*
- No tenemos garantía del ordenamiento que obtendremos

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Los problemas de la concurrencia (1)

```
class EjemploHilos
  def initialize
    @x = 0
  end
  def f1
    sleep 0.1
    print '+'
    @x += 3
  end
  def f2
    sleep 0.1
    print '*'
    @x *= 2
  end
  def run
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1
    print ' %d '  % @x
  end
end
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Los problemas de la concurrencia (1)

```
class EjemploHilos
  def initialize
    @x = 0
  end
  def f1
    sleep 0.1
    print '+'
    @x += 3
  end
  def f2
    sleep 0.1
    print '*'
    @x *= 2
  end
  def run
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1
    print '%d ' % @x
  end
end
```

```
>> e = EjemploHilos.new;10.times{e.run}
0 *+3 *+9 *+21 +*48 *+99 +*204 *+411 +*828 *+1659

>> e = EjemploHilos.new;10.times{e.run}
+0 *+6 *+*18 42 +*+90 **186 +375 +**756 ++1515 *3036
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Los problemas de la concurrencia (2)

- No son dos hilos compitiendo por el acceso a la variable
  - Son tres
  - El *jefe* también entra en la competencia a la hora de imprimir
- A veces, el órden de la ejecución es (¿parece ser?) `(@x *2) + 3`, a veces `(@x + 3) * 2`
  - A veces la impresión ocurre en otro órden: `+**756` o `++1515`
- Esto porque tenemos una *condición de carrera* en el acceso a la variable compartida

Condición de carrera (Race condition)

- Error de programación
- Implica a dos procesos (o hilos)
- Fallan al comunicarse su estado mutuo
- Lleva a *resultados inconsistentes*
  - Problema muy común
  - Difícil de depurar
- Ocurre por no considerar la *no atomicidad* de una operación
- Categoría importante de fallos de seguridad

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Operación atómica

- Operación que tenemos la garantía que se ejecutará *o no* como *una sóla unidad de ejecución*
- *No implica* que el sistema no le retirará el flujo de ejecución
  - *El efecto de que se le retire el flujo* no llevará a comportamiento inconsistente.
  - Requiere sincronización *explícita* entre los procesos que la realicen

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Sección crítica

Es el área de código que:

- Realiza el acceso (¿modificación? ¿lectura?) de datos compartidos
- Requiere *ser protegida de accesos simultáneos*
- Dicha protección tiene que ser implementada *siempre, y manualmente* por el programador
  - Identificarlas requiere inteligencia
- Debe ser protegida *empleando mecanismos atómicos*
  - Si no, el problema podría aminorarse — Pero no prevenirse
  - ¡Cuidado con los accesos casi-simultáneos!

Sección crítica

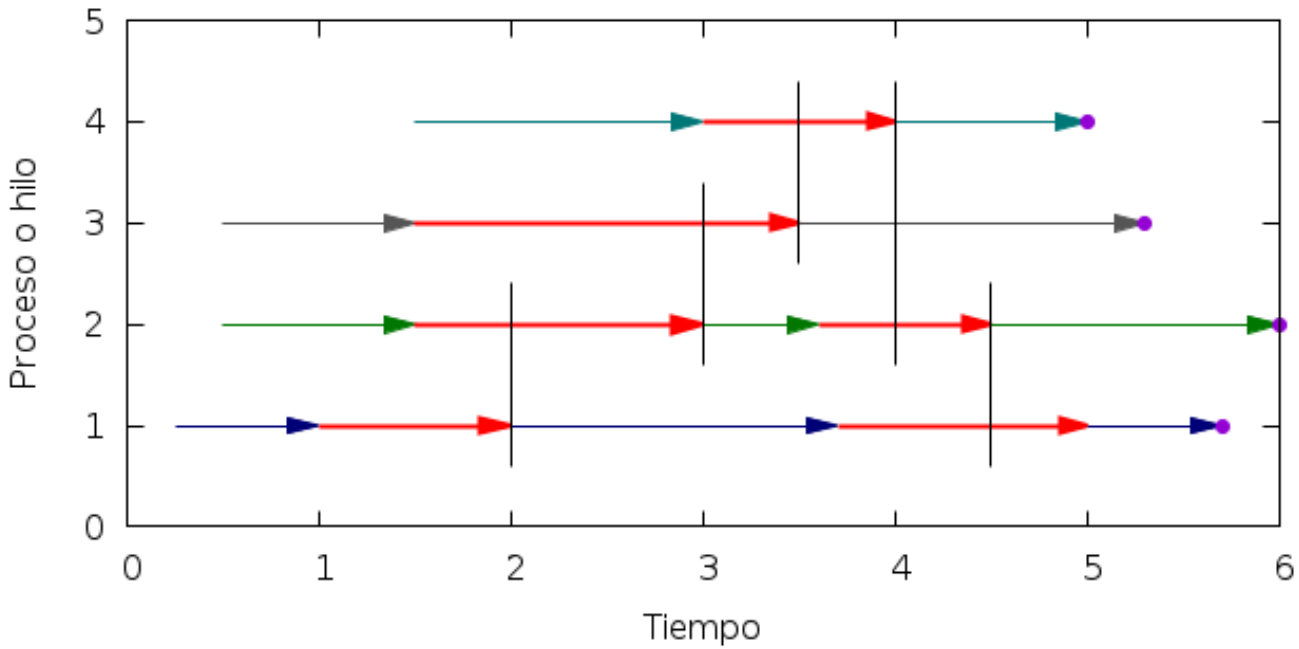


Figura: Sincronización: La exclusión de una sección crítica común a varios procesos se protege por medio de regiones de exclusión mutua

Bloqueo mutuo

Algunos autores lo presentan como *interbloqueo*.
En inglés, *deadlock*

- Dos o más procesos poseen determinados recursos
- Cada uno de ellos queda detenido esperando a alguno de los que tiene otro

Bloqueo mutuo

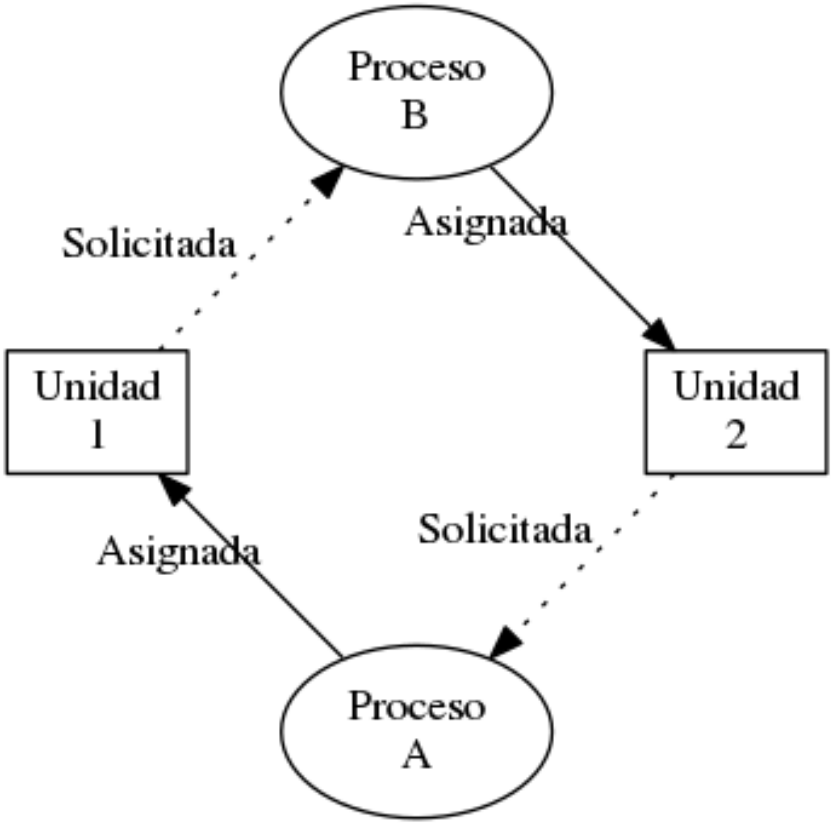


Figura: Esquema clásico de un bloqueo mutuo simple: Los procesos *A* y *B* esperan mutuamente para el acceso a las unidades *1* y *2*.

Bloqueo mutuo

- El sistema operativo puede seguir procesando normalmente
  - Pero ninguno de los procesos involucrados puede avanzar
  - ¿Única salida? Que el administrador del sistema interrumpa a alguno de los procesos
    - . . . Implica probable pérdida de información

Inanición

En inglés, *resource starvation*

- Situación en que uno o más procesos están atravesando exitosamente una sección crítica
  - Pero el flujo no permite que otro proceso, posiblemente de otra clase, entre a dicha sección
- El sistema continúa siendo productivo, pero uno o más de los procesos puede estar detenido por un tiempo arbitrariamente largo.

Primer acercamiento: *Reservas de autobús*

¡Inseguro! ¿Qué hizo el programador bien? ¿qué hizo mal?

```
use threads::shared;
my ($proximo_asiento :shared, $capacidad :shared, $bloq
    :shared);
$capacidad = 40;
sub asigna_asiento {
  while ($bloq) { sleep 0.1; }
  $bloq = 1;
  if ($proximo_asiento < $capacidad) {
    $asignado = $proximo_asiento;
    $proximo_asiento += 1;
    print "Asiento asignado: $asignado\n";
  } else {
    print "No hay asientos disponibles\n";
    $bloq = 0; return 1; # Indicando error / falla
  }
  $bloq = 0; return 0;
}
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¿Por qué es inseguro el ejemplo anterior?

Líneas 5 y 6:

- Espera activa (*spinlock*): Desperdicio de recursos
  - Aunque esta espera activa lleva dentro un `sleep`, sigue siendo espera activa.
  - Eso hace que el código sea *poco considerado* — No que sea inseguro
- ¿Quién protege a `$bloq` de modificaciones no-atómicas?

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Las secciones críticas deben protegerse a otro nivel

- Las primitivas que empleemos para sincronización *deben ser atómicas*
- La única forma de asegurar su atomicidad es *implementándolas a un nivel más bajo* que el del código que deben proteger
  - (Al menos) el proceso debe implementar la protección entre hilos
  - (Al menos) el sistema operativo debe implementar la protección entre procesos

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Mismo ejemplo, empleando un candado (*mutex*)

```
use threads::shared;
my ($proximo_asiento :shared, $capacidad :shared);
$capacidad = 40;
sub asigna_asiento {
  lock($proximo_asiento);
  if ($proximo_asiento < $capacidad) {
    $asignado = $proximo_asiento;
    $proximo_asiento += 1;
    print "Asiento asignado: $asignado\n";
  } else {
    print "No hay asientos disponibles\n";
    return 1;
  }
  return 0;
}
```

La implementación comunicación entre hilos en Perl implementa un *mutex* a través de la función `lock()`, como *atributo* sobre una variable, con *ámbito léxico*

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Requisitos para las *primitivas*

- Implementadas a un nivel más bajo que el código que protegen
  - Desde el sistema operativo
  - Desde bibliotecas de sistema
  - Desde la máquina virtual (p.ej. JVM)
- ¡No las implementes tú mismo!
  - Parecen conceptos simples... Pero no lo son
  - Utilicemos el conocimiento acumulado de medio siglo

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Candados (Mutex)

- Contracción de *Mutual Exclusion*, exclusión mutua
- Un mecanismo que asegura que la región protegida del código se ejecutará como si fuera atómica
  - *No garantiza que el planificador no interrumpa* — Eso rompería el multiprocesamiento preventivo.
  - Requiere que *cada hilo o proceso* implemente (¡y respete!) al mutex
- Mantiene en espera a los procesos adicionales que quieran emplearlo
  - Sin garantizar ordenamiento

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Candados (Mutex)

- Contracción de *Mutual Exclusion*, exclusión mutua
- Un mecanismo que asegura que la región protegida del código se ejecutará como si fuera atómica
  - *No garantiza que el planificador no interrumpa* — Eso rompería el multiprocesamiento preventivo.
  - Requiere que *cada hilo o proceso* implemente (¡y respete!) al mutex
- Mantiene en espera a los procesos adicionales que quieran emplearlo
  - Sin garantizar ordenamiento
- Ejemplo: La llave del baño en un entorno de oficina mediana

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Semáforos

- Propuestos por Edsger Dijkstra (1965)
- Estructuras de datos simples para la sincronización y (muy limitada) comunicación entre procesos
  - ¡Increíblemente versátiles para lo limitado de su interfaz!
- Se han publicado muchos patrones basados en su interfaz, modelando interacciones muy complejas

¡Ojo! No piensen en semáforos viales, verde/amarillo/rojo (eso sería un simple *mutex*). Piensen en semáforos de tren.

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Semáforos de tren

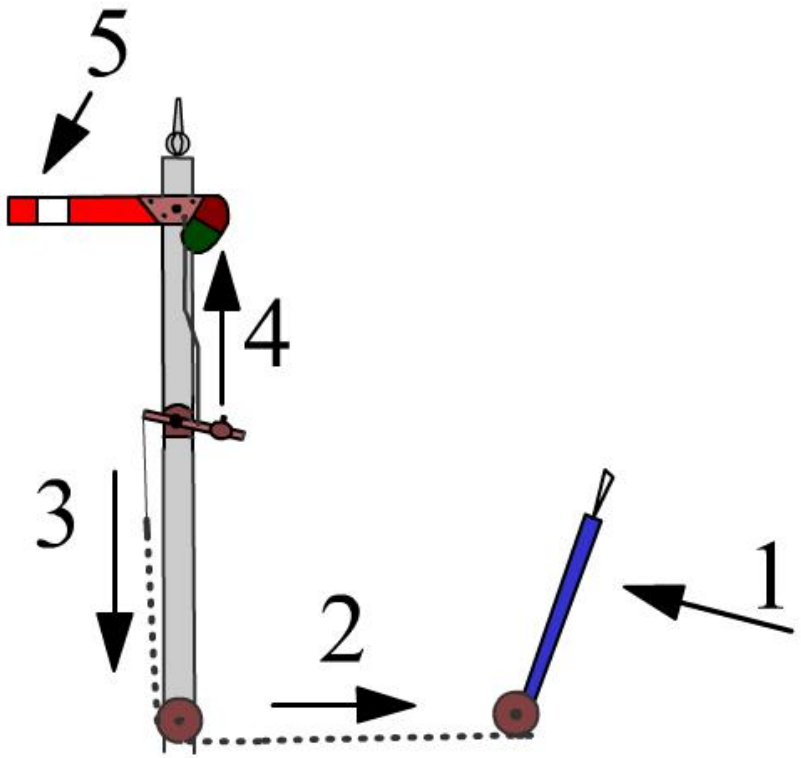


Figura: Semáforo de tren (Fuente: Wikipedia, *Señalización ferroviaria argentina*)

Las tres operaciones de los semáforos

Inicializar Puede inicializarse a cualquier valor entero. Una vez inicializado, el valor *ya no puede ser leído*.

Decrementar Disminuye en 1 el valor del semáforo. Si el resultado es negativo, el hilo *se bloquea* y no puede continuar hasta que *otro hilo* incremente al semáforo.
Puede denominarse `wait`, `down`, `acquire`, `P` (*proberen te verlagen*, *intentar decrementar*)

Incrementar Incrementa en 1 el valor del semáforo. Si hay hilos esperando, uno de ellos es despertado.
Puede denominarse `signal`, `up`, `release`, `post` o `V` (*verhogen*, *incrementar*).

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Los semáforos en C (POSIX pthreads)

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- `pshared` indica si se compartirá entre procesos o sólo entre hilos (por optimización de estructuras)
- `sem_trywait` extiende la interfaz de Dijkstra: Verifica si el semáforo puede ser decrementado, pero en vez de bloquearse, regresa al invocante un error
  - El proceso *debe tener la lógica* para no proceder a la sección crítica

Variables de Condición

Extensión sobre el comportamiento de un mutex permitiéndole una mayor ïnteligencia". Siempre operan *junto* con un mutex.

`wait()` Libera el candado y se bloquea hasta recibir una *notificación*. Una vez despertado, *re-adquiere* el candado.

`signal()` Despierta a un hilo esperando a esta condición (si lo hay). No libera al candado.

`broadcast` Notifica a todos los hilos que estén esperando a esta condición

`timedwait(timeout)` Como `wait()`, pero se despierta (regresando error) pasado el tiempo indicado si no recibió notificación.

Variables de Condición en C (POSIX pthreads)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
    *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
    *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Ejemplo con variables de condición

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
/* (...) */
/*  Un hilo espera hasta que x sea mayor que y */
pthread_mutex_lock(&mut);
while (x <= y) {
        pthread_cond_wait(&cond, &mut);
}
/* (Realiza el trabajo...) */
pthread_mutex_unlock(&mut);
/* (...) */
/* Cuando otro hilo modifica a x o y, notifica */
/* a todos los hilos que est n esperando */
pthread_mutex_lock(&mut);
x = x + 1;
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

http://www.sourceware.org/pthreads-win32/manual/pthread_cond_

Ejemplo de espera limitada con VCs

```
struct timeval now;
struct timespec timeout;
int retcode;
pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT) {
        retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT) {
        /* Expirado el tiempo estipulado - Falla. */
} else {
        /* Trabaja con x y y */
}
pthread_mutex_unlock(&mut);
```

Problemática con mutexes, semáforos y VCs

- No sólo hay que encontrar el mecanismo correcto para proteger nuestras secciones críticas
  - hay que *implementarlo correctamente*
  - La semántica de paso de mensajes por esta vía puede ser confusa
- Un encapsulamiento *más claro* puede reducir problemas
- Puede haber procesos que compitan por recursos *de forma hostil*

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Competencia hostil por recursos

Qué pasa si en vez de esto:

```
sem_wait(semaforo);
seccion_critica();
sem_post(semaforo);
```

Tenemos esto:

```
while (sem_trywait(semaforo) != 0) {}
seccion_critica();
sem_post(semaforo);
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

La estupidez humana puede ser infinita

```
/* Crucemos los dedos... */
/* A fin de cuentas, corremos con baja frecuencia! */
seccion_critica();
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Monitores

- Estructuras abstractas (*ADTs* u *objetos*) provistas por el lenguaje o entorno de desarrollo
- *Encapsulan* tanto a los datos *como a las funciones que los pueden manipular*
- Impiden el acceso directo a las funciones potencialmente peligrosas
- *Exponen* una serie de *métodos públicos*
  - Y pueden implementar *métodos privados*
- Al no presentar una interfaz que puedan *subvertir*, aseguran que todo el código que asegura el *acceso concurrente seguro* es empleado
- Pueden ser presentados como *bibliotecas*

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Ejemplo: Sincronización en Java

Java facilita que una clase estándar se convierta en un monitor como una propiedad de la declaración de método, y lo implementa directamente en la JVM. (Silberschatz):

```
public class SimpleClass {
  // . . .
  public synchronized void safeMethod() {
    /* Implementation of safeMethod() */
  }
}
```

La JVM implementa:

- Mutexes a través de la declaración `synchronized`
- *variables de condición*
- Una semántica parecida (¡no idéntica!) a la de semáforos con `var.wait()` y `var.signal()`

#+END_CENTER

Soluciones en hardware

- Decimos una y otra vez que *la concurrencia está aquí para quedarse*
- El hardware especializado para cualquier cosa (interrupciones, MMU, punto flotante, etc.) es siempre caro, hasta que baja de precio
- ¿No podría el hardware ayudarnos a implementar operaciones atómicas?

Veamos algunas estrategias

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Inhabilitación de interrupciones

Efectivamente evita que las secciones críticas sean interrumpidas, pero. . .

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Inhabilitación de interrupciones

Efectivamente evita que las secciones críticas sean interrumpidas, pero...

- Inútil cuando hay multiprocesamiento real
  - A menos que detenga también la ejecución en los demás CPUs
- *Matar moscas a cañonazos*
- Inhabilita el multiproceso preventivo
  - Demasiado peligroso → Bastaría un error en la sección crítica de *cualquier proceso* para que se congelara el sistema entero

Instrucciones atómicas: `test_and_set` (1)

Siguiendo una implementación *en hardware* correspondiente a:

```
boolean test_and_set(int i) {
    if (i == 0) {
        i = 1;
        return true;
    } else return false;
}
void free_lock(int i) {
    if (i == 1) i = 0;
}
```

Instrucciones atómicas: `test_and_set` (2)

Bastaría con:

```
enter_region:
  tsl reg, flag       ; Test and Set Lock; 'flag' es la variable
                      ; compartida, es cargada al registro 'reg'
                      ; y, atomicamente, convertida en 1.
  cmp reg, #0         ; Era la bandera igual a 0 al entrar?
  jnz enter_region    ; En caso de que no fuera 0 al ejecutar
                      ; tsl, vuelve a 'enter_region'
  ret                 ; Termina la rutina. 'flag' era cero al
                      ; entrar. 'tsl' fue exitoso y 'flag' queda
                      ; no-cero. Tenemos acceso exclusivo al
                      ; recurso protegido.
leave_region:
  move flag, #0       ; Guarda 0 en flag, liberando el recurso
  ret                 ; Regresa al invocante.
```

¿Qué problema le vemos?

Problemas con `test_and_set`

- Espera activa
  - Se utiliza sólo en código no interrumpible (p.ej. gestor de interrupciones en el núcleo)
- Código no portable
- Imposible de implementar en arquitecturas RISC limpias
  - Doble acceso a memoria en una sóla instrucción
  - . . . Muy caro de implementar en arquitecturas CISC
- Susceptible a problemas de coherencia de cache

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Memoria transaccional

- Idea base: Semántica de bases de datos en el acceso a memoria
- Permite agrupar varias operaciones en una sóla *transacción*
  - Una vez terminada, *confirmar* (*commit*) todos los cambios
  - O, en caso de error, *rechazarlos* (*rollback*)
- Si algún otro proceso modifica alguna de las localidades en cuestión, el proceso se *rechaza*
- Toda lectura de memoria antes de *confirmar* entrega los datos previos

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Memoria transaccional: Ejemplo de semántica

```
do {
    begin_transaction();
    var1 = var2 * var3;
    var3 = var2 - var1;
    var2 = var1 / var2;
} while (! commit_transaction());
```

Ejemplo poco eficiente, elegido meramente por claridad: Efectúa múltiples cálculos dentro de una espera activa efectiva

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Memoria transaccional en software (STM)

- Implementaciones como la mencionada disponibles en varios lenguajes
- Computacionalmente caras
  - Invariablemente *mucho* más lentas
  - Y menos eficientes en espacio
  - . . . Pero en hardware tampoco serían mucho más baratas — y sí tendrían restricciones (en tamaño o cantidad de transacciones simultáneas)
- Puede llevar a inconsistencias si implican *cualquier estructura* fuera del control de la transacción (archivos, dispositivos, IPC)
- Construcción poderosa (¡y cómoda!)

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¿Patrones?

- A continuación veremos ocho *patrones* de sincronización basados en semáforos.
  - Señalizar (*signal*), *Rendezvous*, Mutex, Multiplex, Torniquete (*turnstile*), Apagador (*switch*), Barrera (*barrier*), Cola (*queue*)
  - Claro está, hay más. Estos son sólo los básicos.

Sí, pero...

¿Qué son los patrones? ¿De qué me sirven? ¿Cómo los puedo usar?

Sí, ¡Patrones!

- Estructuras de programación que *tienden a aparecer* como respuestas naturales
  - En el código de *varios* buenos programadores
- Conviene conocerlos y tenerlos en mente para reconocer las *situaciones* en que los podamos emplear

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Señalizar

- Un hilo debe informar a otro que cierta condición está cumplida
- Ejemplo: Un hilo prepara una conexión en red mientras el otro prepara los datos a enviar
  - No podemos arriesgarnos a comenzar la transmisión hasta que la conexión esté lista

```
from threading import Semaphore, Thread
semaf = Semaphore(0)
Thread(target=prepara_conexion, args=[semaf]).start()
Thread(target=envia_datos, args=[semaf]).start()
```

```
def prepara_conexion(semaf):
  crea_conexion()
  semaf.release()
```

```
def envia_datos(semaf):
  calcula_datos()
  semaf.acquire()
  envia_por_red()
```

Rendezvous

- Nombre tomado del francés para *preséntense* (utilizado ampliamente en inglés para *tiempo y lugar de encuentro*)
- Dos hilos deben esperarse mutuamente en cierto punto para continuar en conjunto
  - Empleamos dos semáforos
- Por ejemplo, en un GUI:
  - Un hilo prepara la interfaz gráfica y actualiza sus eventos
  - Otro hilo efectúa cálculos para mostrar
  - Queremos mostrar la simulación desde el principio, no debe iniciar el cálculo antes de que haya una interfaz mostrada
  - No queremos que la interfaz se presente en blanco

Rendezvous

```
from threading import Semaphore, Thread
guiListo = Semaphore(0)
calculoListo = Semaphore(0)
Thread(target=gui, args=[]).start()
Thread(target=calculo, args=[]).start()
```

```
def calculo():
  inicializa_datos()
  calculoListo.release()
  guiListo.acquire()
  procesa_calculo()
```

```
def gui():
  inicializa_gui()
  guiListo.release()
  calculoListo.acquire()
  recibe_eventos()
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Mutex

Un mutex puede implementarse con un semáforo inicializado a 1:

```
mutex = Semaphore(1)
# ...Inicializamos estado y lanzamos hilos
mutex.acquire()
# Estamos en la region de exclusion mutua
x = x + 1
mutex.release()
# Continua la ejecucion paralela
```

- Varios hilos pueden pasar por este código, tranquilos de que la región crítica será accesada por sólo uno a la vez
- El mismo mutex puede proteger a *diferentes secciones críticas* (p.ej. distintos puntos donde se usa el mismo recurso)

Multiplex

- Un mutex que permita a *no más de cierta cantidad de hilos* empleando determinado recurso
- Para implementar un *multiplex*, basta inicializar el semáforo de nuestro mutex a un valor superior:

```
import threading; import time; import random
multiplex = threading.Semaphore(5)

def operacion(id, semaf):
  semaf.acquire()
  print "Id %d en la secc. crit." % id
  time.sleep(random.random())
  semaf.release()
  print "Terminando el id %d" % id

for hilo in range(10):
  threading.Thread(target=operacion,
      args=[hilo,multiplex]).start()
```

Torniquete

- Garantiza que un grupo de hilos o procesos pasan por un punto determinado *de uno en uno*
- Ayuda a controlar contención, es empleado como parte de construcciones posteriores

```
torniquete = Semaphore(0)
# (...)
if alguna_condicion():
  torniquete.release()
# (...)
torniquete.acquire()
torniquete.release()
```

- Esperamos primero a una *señalización* que permita que los procesos comiencen a fluir
- La sucesión rápida `acquire()` / `release()` permite que los procesos fluyan uno a uno

Apagador

- Principalmente emplados en situación de *exclusión categórica*
  - Categorías de procesos, no procesos individuales, que deben excluirse mutuamente de secciones críticas
- Metáfora empleada:
  - La zona de exclusión mutua es un *cuarto*
  - Los procesos que quieren entrar deben verificar *si está prendida la luz*
- Implementación ejemplo a continuación (problema de *lectores-escritores*)

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Barrera

- Generalización de *rendezvous* para manejar a varios hilos (no sólo dos)
- El papel de cada uno de estos hilos puede ser el mismo, puede ser distinto
- Requiere de una variable adicional para mantener registro de su *estado*
  - Esta variable adicional es *compartida* entre los hilos, y debe ser *protegida por un mutex*

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Barrera: Código ejemplo

```
from threading import
    Semaphore, Thread
cuenta = 0
mutex = Semaphore(1)
barrera = Semaphore(0)
for i in range(10):
  Thread(target=vamos,args=[i]).start()
```

```
def vamos(id):
  global cuenta, mutex, barrera
  inicializa(id)
  mutex.acquire()
  cuenta = cuenta + 1
  if cuenta == 10:
    barrera.release()
  mutex.release()
  barrera.acquire()
  barrera.release()
  procesa(id)
```

- Todos los hilos se inicializan por separado (`inicializa()`)
- Ningún hilo inicia hasta que todos estén listos
- *Pasar la barrera* en este caso equivale a *habilitar un torniquete*

Barreras: Implementación en pthreads

- Las barreras son una construcción tan común que las encontramos "prefabricadas"
- Definición en los hilos POSIX (pthreads):

```
int pthread_barrier_init(pthread_barrier_t  *barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned count);
int pthread_barrier_wait(pthread_barrier_t  *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Cola

- Tenemos que asegurarnos que procesos de dos distintas categorías procedan siempre *en pares*
- Patrón conocido también como *baile de salón*:
  - Para que una pareja baile, tiene que haber un *líder* y un *seguidor*
  - Cuando llega al salón un *líder*, revisa si hay algún *seguidor* esperando
    - Si lo hay, bailan
    - Si no, espera a que llegue uno
  - El seguidor emplea la misma lógica.

Cola

```
colaLideres = Semaphore(0)
colaSeguidores = Semaphore(0)
# (...)
```

```
def lider():
  colaSeguidores.release()
  colaLideres.acquire()
  baila()
```

```
def seguidor():
  colaLideres.release()
  colaSeguidores.acquire()
  baila()
```

Cola

```
colaLideres = Semaphore(0)
colaSeguidores = Semaphore(0)
# (...)
```

```
def lider():
  colaSeguidores.release()
  colaLideres.acquire()
  baila()
```

```
def seguidor():
  colaLideres.release()
  colaSeguidores.acquire()
  baila()
```

- Nuevamente, estamos viendo un *rendezvous*
  - Pero es entre dos categorías, no entre dos hilos específicos
- El patrón puede refinarse mucho, esta es la implementación básica
  - Asegurarse que sólo una pareja baile a la vez
  - Asegurarse que bailen en el órden en que llegaron

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¿De qué se tratan estos problemas clásicos?

- Manera fácil de recordar para hablar de situaciones comunes en la vida real
- Forma de demostrar las ventajas/desventajas de una construcción de sincronización frente a otras
- Ampliamente utilizados en la literatura de la materia
- Ayudan a comprender la complejidad del manejo de los patrones, aparentemente simples

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Problema productor-consumidor: Planteamiento

- División de tareas tipo *línea de ensamblado*
  - Un grupo de procesos va *produciendo* ciertas estructuras
  - Otro grupo va *consumiéndolas*
- Emplean un buffer de acceso compartido para comunicarse dichas estructuras
  - Agregar o retirar un elemento del buffer *debe hacerse de forma atómica*
  - Si un consumidor está listo y el buffer está vacío, debe bloquearse (¡no espera activa!)
- Refinamientos posteriores
  - Implementación con un buffer no-infinito (¿buffer circular?):
- Vida real: Cola de trabajos para impresión, *Pipes* (tuberías) entre procesos

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Productor-consumidor: Implementación ingenua

```
import threading
buffer = []
threading.Thread(target=productor,args=[]).start()
threading.Thread(target=consumidor,args=[]).start()

def productor():
  while True:
    event = genera_evento()
    buffer.append(event)

def consumidor():
  while True:
    event = buffer.pop()
    procesa(event)
```

¿Qué problema vemos?
¿Qué estructuras neceistan protección?
(¿Qué estructuras no?)

Productor-consumidor: Estructuras a emplear

Vamos a emplear dos semáforos:

- Un mutex sencillo (`mutex`)
- Un semáforo (`elementos`) representando el estado del sistema

  `elementos > 0` Cuántos eventos tenemos pendientes por procesar

  `elementos < 0` Cuántos consumidores están listos y esperando un evento

Productor-consumidor: Implementación

```
import threading
mutex = threading.Semaphore(1)
elementos = threading.Semaphore(0)
buffer = []
class Evento:
  def __init__(self):
    print "Generando evento"
  def process(self):
    print "Procesando evento"
threading.Thread(target=productor, args=[]).start()
threading.Thread(target=consumidor, args=[]).start()
```

```
def productor():
  while True:
    event = Evento()
    mutex.acquire()
    buffer.append(event)
    mutex.release()
    elementos.release()
```

```
def consumidor():
  while True:
    elementos.acquire()
    mutex.acquire()
    event = buffer.pop()
    mutex.release()
    event.process()
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Problema lectores-escritores: Planteamiento

- Una estructura de datos puede ser accesada simultáneamente por muchos *procesos lectores*
- Si un proceso requiere *modificarla*, debe asegurar que:
  - Ningún proceso lector esté empleándola
  - Ningún otro proceso escritor esté empleándola
  - Los escritores deben tener *acceso exclusivo* a la sección crítica
- Refinamiento: Debemos evitar que un *influjo constante de lectores* nos deje en situación de *inanición*

Lectores-escritores: Primer acercamiento

- El problema es de *exclusión mutua categórica*
- Empleamos un patrón *apagador*
  - Los escritores entran al *cuarto* sólo con la luz *apagada*
- Mutex para el indicador del número de lectores

```
import threading
lectores = 0
mutex = threading.Semaphore(1)
cuarto_vacio =
    threading.Semaphore(1)

def escritor():
  global lectores
  cuarto_vacio.acquire()
  escribe()
  cuarto_vacio.release()
#+end_src python
#+latex: \end{column}
    \begin{column}{0.5\textwidth}
```

Lectores-escritores: Notas

- La misma estructura puede usarse siguiendo diferentes patrones
  - `escritor()` usa `cuarto_vacio` como un mutex, `lector()` lo usa como un apagador
  - Porque las características (requisitos) de cada categoría son distintas
- Susceptible a la inanición
  - Si tenemos alta concurrencia de lectores, un escritor puede quedarse esperando para siempre
  - Podemos agregar un *torniquete* evitando que lectores adicionales se *cuelen* si hay un escritor esperando

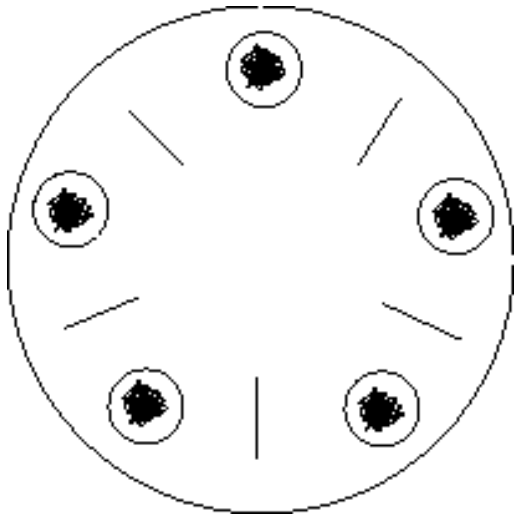
Lectores-escritores sin inanición

```
import threading
lectores = 0
mutex = threading.Semaphore(1)
cuarto_vacio =
    threading.Semaphore(1)
torniquete =
    threading.Semaphore(1)

def escritor():
  torniquete.acquire()
  cuarto_vacio.acquire()
  escribe()
  cuarto_vacio.release()
  torniquete.release()
```

```
def lector():
  global lectores
  torniquete.acquire()
  torniquete.release()

  mutex.acquire()
  lectores = lectores + 1
  if lectores == 1:
    cuarto_vacio.acquire()
  mutex.release()

  lee()

  mutex.acquire()
  lectores = lectores - 1
  if lectores == 0:
    cuarto_vacio.release()
  mutex.release()
```

La cena de los filósofos: Planteamiento (1)

- Hay cinco filósofos sentados a la mesa
  - Al centro de la mesa hay un tazón de arroz
  - Cada filósofo tiene un plato, un palillo a la derecha, y un palillo a la izquierda
  - El palillo lo *comparten* con el filósofo de junto



Imagenes ilustrativas: Ted P. Baker

La cena de los filósofos: Planteamiento (2)

- Cada filósofo *sólo* sabe hacer dos cosas: Pensar y comer
- Los filósofos *piensan* hasta que les da hambre
  - Una vez que tiene hambre, un filósofo levanta un palillo, luego levanta el otro, y come
  - Cuando se sacia, pone en la mesa un palillo, y luego el otro
- ¿Qué problemas pueden presentarse?

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Cena de filósofos: Bloqueo mutuo

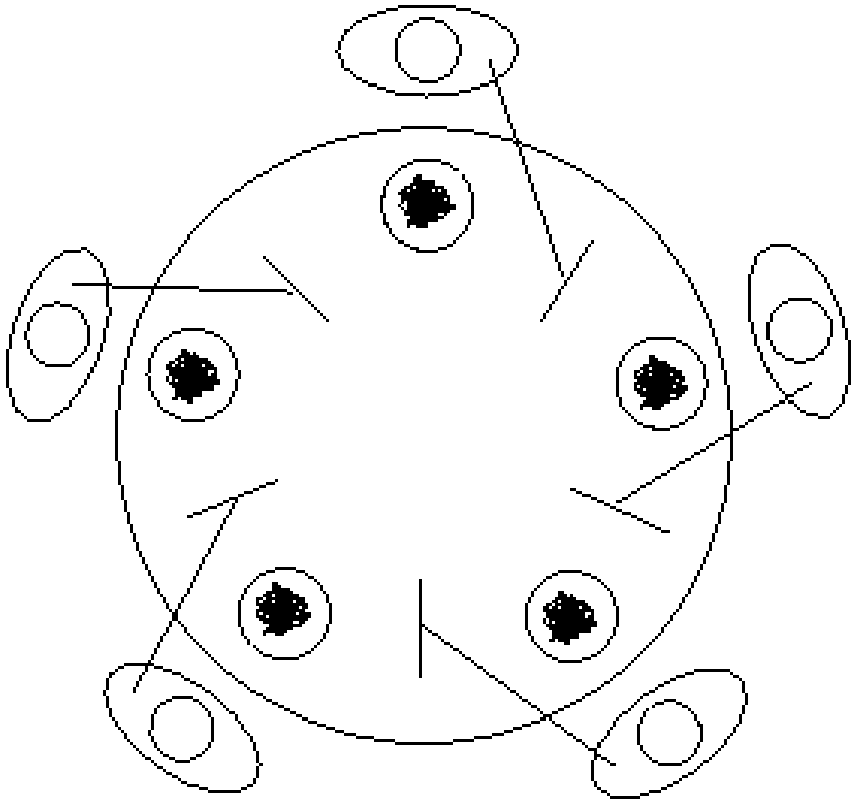


Figura: Cuando todos los filósofos intentan levantar uno de los palillos se produce un *bloqueo mutuo*

Cena de filósofos: Inanición

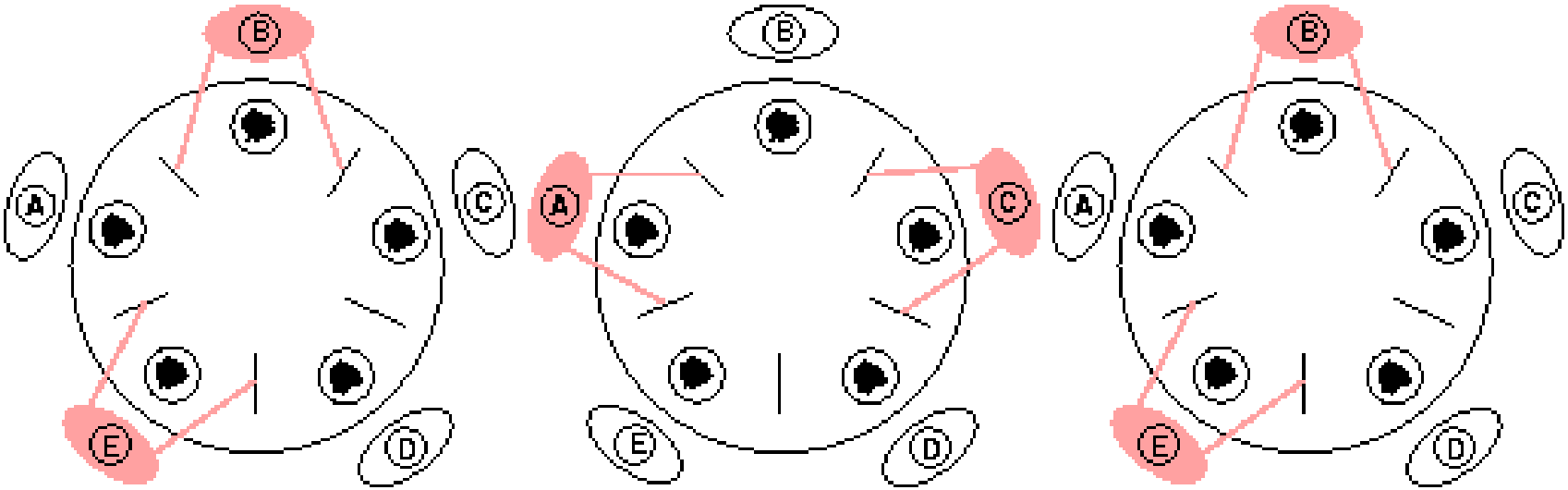


Figura: Una rápida sucesión de C y E lleva a la inanición de D

Cena de filósofos: Primer acercamiento

- Para la resolución de este problema, representaremos a los palillos como un arreglo de semáforos
  - Esto asegura que presenten la semántica de exclusión mutua: *levantar un palillo* es una operación atómica
- Cada filósofo sabe cuál es su ID (numérico, 0 a n; ejemplo con $n = 5$)
  - Los palillos de i son `palillos[i]` y `palillos[(i+1) % n]`

Cena de filósofos: Primer acercamiento

```
import threading
num = 5
palillos = [threading.Semaphore(1) for i in range(num)]
filosofos = [threading.Thread(target=filosofo,
    args=[i]).start() for i in range(num)]
```

```
def filosofo(id):
  while True:
    piensa(id)
    levanta_palillos(id)
    come(id)
    suelta_palillos(id)
def levanta_palillos(id):
  palillos[(id+1) %
      num].acquire()
  print "%d - Tengo el palillo
      derecho" % id
  palillos[id].acquire()
  print "%d - Tengo ambos
      palillos" % id
```

```
def suelta_palillos(id):
  palillos[(id+1) %
      num].release()
  palillos[id].release()
  print "%d - Sigamos
      pensando..." % id
def piensa(id):
  # (...)
  print "%d - Tengo hambre..."
      % id
def come(id):
  print "%d - A comer!" % id
  # (...)
```

Cena de filósofos: Semáforos para comunicación

- Sujeto a bloqueos mutuos
  - Eventualmente, terminarán todos suspendidos con el palillo derecho en la mano
- ¿Ideas para evitar el bloqueo mutuo?

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Cena de filósofos: Filósofos zurdos

¿Y si hacemos que los filósofos pares sean diestros y los impares sean zurdos?

```
def levanta_palillos(id):
  if (id % 2 == 0): # Zurdo
    palillo1 = palillos[id]
    palillo2 = palillos[(id+1) % num]
  else: # Diestro
    palillo1 = palillos[(id+1) % num]
    palillo2 = palillos[id]
  palillo1.acquire()
  print "%d - Tengo el primer palillo" % id
  palillo2.acquire()
  print "%d - Tengo ambos palillos" % id
```

Cena de filósofos: Basta con uno

De hecho, basta con que uno de los filósofos sea zurdo para que no haya bloqueo mutuo

```
def levanta_palillos(id):
 if id == 0: # Zurdo
   palillos[id].acquire()
   print "%d - Tengo el palillo izquierdo" % id
   palillos[(id+1) % num].acquire()
 else: # Diestro
   palillos[(id+1) % num].acquire()
   print "%d - Tengo el palillo derecho" % id
   palillos[id].acquire()
 print "%d - Tengo ambos palillos" % id
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Cena de filósofos: Monitor y VCs (C)

Implementación de Ted P. Baker (Florida State University) basada en Tanenbaum

```
/* Implementacion para cinco filosofos */
pthread_cond_t  CV[NTHREADS];    /* Variable por filosofo */
pthread_mutex_t M;               /* Mutex para el monitor */
int             state[NTHREADS]; /* Estado de cada filosofo */

void init () {
    int i;
    pthread_mutex_init(&M, NULL);
    for (i = 0; i < 5; i++) {
        pthread_cond_init(&CV[i], NULL);
        state[i] = PENSANDO;
    }
}

void come(int i) {
    printf("El filosofo %d esta comiendo\n", i);
}
```

Cena de filósofos: Monitor y VCs (C)

```
void toma_palillos (int i) {
    pthread_mutex_lock(&M)
    state[i] = HAMBRIENTO;
    actualiza(i);
    while (state[i] == HAMBRIENTO)
        pthread_cond_wait(&CV[i], &M);
    pthread_mutex_unlock(&M);
}

void suelta_palillos (int i) {
    state[i] = PENSANDO;
    actualiza((i + 4) % 5);
    actualiza((i + 1) % 5);
    pthread_mutex_unlock(&M);
}
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Cena de filósofos: Monitor y VCs (C)

```
/* No incluimos 'actualiza' en los encabezados (funcion
    interna) */
int actualiza (int i) {
    if ((state[(i + 4) % 5] != COMIENDO) &&
        (state[i] == HAMBRIENTO) &&
        (state[(i + 1) % 5] != COMIENDO)) {
        state[i] = COMIENDO;
        pthread_cond_signal(&CV[i]);
    }
    return 0;
}
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Cena de filósofos: Monitor y VCs (C)

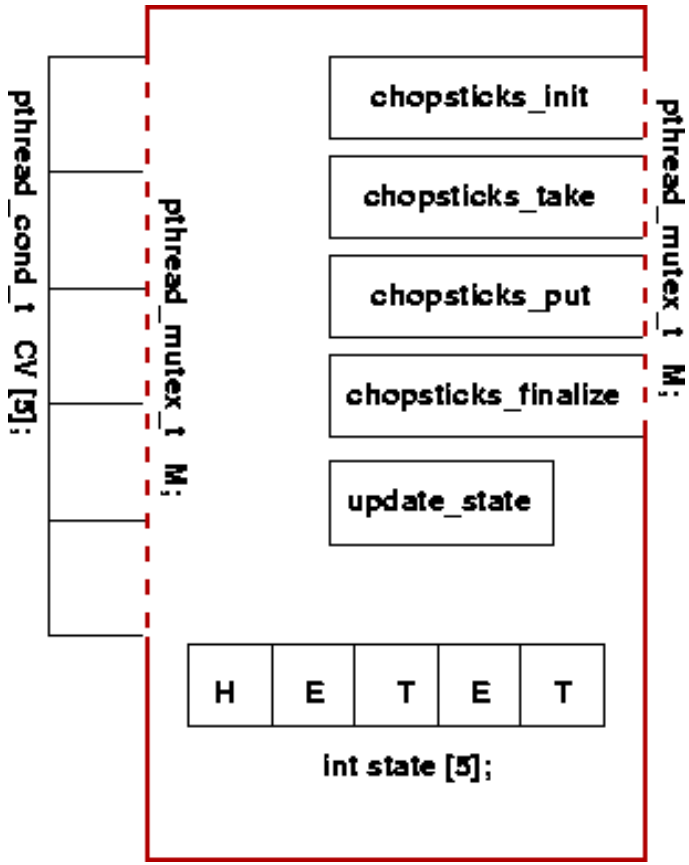


Figura: Representación del monitor

Cena de filósofos: ¿Y la inanición?

- La inanición es un problema mucho más dificil de tratar que el bloqueo mutuo
- El algoritmo presentado por Tanenbaum (1987) buscaba prevenir la inanición, pero Gingras (1990) demostró que aún éste es vulnerable a ciertos patrones
- Gingras propone un algoritmo libre de inanición, pero demanda de estructuras adicionales que rompen el planteamiento del problema original

Artículo de Gingras (1990): Dining philosophers revisited
(descargable desde la red de la UNAM — No RIU)

Fumadores compulsivos: Planteamiento

- Para fumar hacen falta tres ingredientes
  - Tabaco, papel, cerillos
- Hay tres fumadores compulsivos / en cadena
  - Cada uno tiene una cantidad ilimitada de *un* ingrediente
  - No cooperan, pero no se estorban: No comparten, pero no acaparan
- Un *agente* de tiempo en tiempo consigue insumos *en pares*
  - Cuando no hay nada en la *mesa*, puede poner una dosis de *dos* ingredientes
  - El agente no habla con los fumadores. Sólo deja los ingredientes cuando uno de ellos termina de fumar.
  - Requisito planteado: *No podemos modificar la lógica del agente*

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Fumadores compulsivos: Primer implementación

Estructuras comunes:

```
import random
import threading
ingredientes = ['tabaco', 'papel', 'cerillo']
semaforos = {}
semaforo_agente = threading.Semaphore(1)
for i in ingredientes:
  semaforos[i] = threading.Semaphore(0)

threading.Thread(target=agente, args=[]).start()
fumadores = [threading.Thread(target=fumador, args=[i]).start()
    for i in ingredientes]

def fuma(ingr):
  print 'Fumador con %s echando humo...' % ingr
```

Fumadores compulsivos: Primer implementación

```
def agente():
  while True:
    semaforo_agente.acquire()
    mis_ingr = ingredientes[:]
    mis_ingr.remove(random.choice(mis_ingr))
    for i in mis_ingr:
      print "Proveyendo %s" % i
      semaforos[i].release()

def fumador(ingr):
  mis_semaf = []
  for i in semaforos.keys():
    if i != ingr:
      mis_semaf.append(semaforos[i])
  while True:
    for i in mis_semaf:
      i.acquire()
    fuma(ingr)
    semaforo_agente.release()
```

Fumadores compulsivos: Bloqueo mutuo

- Tenemos un semáforo por ingrediente
- Pero no podemos asegurar el ordenamiento
- Al aparecer *un* ingrediente en la mesa, cualquiera de los dos fumadores que lo requiera lo va a tomar
  - La mitad de las veces, el segundo ingrediente que *aparezca* no le servirá
  - Otro de los fumadores tomará este segundo ingrediente
  - ¡Bloqueo!

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Fumadores compulsivos: Empleando intermediarios

- No podemos modificar al agente, pero sí a los fumadores
- Podemos usar *intermediarios*
  - Uno por cada ingrediente
  - Con comunicación entre sí
  - Cuando toman una decisión, despiertan al fumador en cuestión
- Agregamos algunas variables globales para representar a los intermediarios y la comunicación entre ellos

```
que_tengo = {}
semaforos_interm = {}
for i in ingredientes:
  que_tengo[i] = False
  semaforos_interm[i] = threading.Semaphore(0)
interm_mutex = threading.Semaphore(1)
intermediarios = [threading.Thread(target=intermediario,
    args=[i]).start() for i in ingredientes]
```

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Fumadores compulsivos: Empleando intermediarios

El código del fumador resulta mucho más simple:

```
def fumador(ingr):
  while True:
    semaforos_interm[ingr].acquire()
    fuma(ingr)
    semaforo_agente.release()
```

Sólo debe esperar a que su intermediario lo despierte.
Sigue siendo su responsabilidad *notificar al agente* para que éste continúe.

Fumadores compulsivos: Empleando intermediarios

¿Qué y cómo se comunican entre sí los intermediarios?

```
def intermediario(ingr):
  otros_ingr = ingredientes[:]
  otros_ingr.remove(ingr)
  while True:
    semaforos[ingr].acquire()
    interm_mutex.acquire()
    for i in otros_ingr:
      if que_tengo[i]:
        que_tengo[i] = False
        semaforos_interm[i].release()
        break
      que_tengo[i] = True
    interm_mutex.release()
```

Tip: Intentar analizar el flujo en paralelo de los tres intermediarios

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

. . . ¿Listos para resolver un ejercicio de tarea?

Implementa hilos que controlen el siguiente sistema:

- La cochera de casa es de portón de apertura automática
- La puerta toma *algún tiempo* en abrir y en cerrar
- El coche toma *algún tiempo* en entrar y salir.
  - Si hay peatones o vehículos cruzando frente a la puerta, tienes que esperar a que terminen de pasar.
- La batería del control está muy débil
  - Cuando llegas o te vas tienes que pedirle a otro actor que abra y cierre por tí.
- Si el coche es golpeado por la puerta, se abolla y hay que llevarlo al taller. No quieres que eso ocurra.

¿Listos para resolver un ejercicio de tarea?

Implementa un sistema que tome en cuenta las restricciones en cuestión. Para su calificación, importa que:

- El problema se resuelva correctamente
- El código sea válido y legal en tu lenguaje favorito
  - Puedes asumir que la semántica del control de hilos, señales y primitivas de sincronización es similar a la expuesta en clase, no hace falta implementar eso a detalle.

Esta tarea pueden hacerla individualmente o en equipos de dos personas; en caso de que elijan hacerlo así, les pido que ambos suban el mismo archivo a EDUCAFI, y que ambos indiquen quiénes son los autores.

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Índice

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

Volviendo al problema de la concurrencia

A modo de corolario, intentemos resolver el problema inicial...

```
class EjemploHilos
  def initialize
    @x = 0
  end
  def f1
    sleep 0.1
    print '+'
    @x += 3
  end
  def f2
    sleep 0.1
    print '*'
    @x *= 2
  end
  def run
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1
    print '%d ' % @x
  end
end
```

```
>> e = EjemploHilos.new;10.times{e.run}
0 *+3 *+9 *+21 +*48 *+99 +*204 *+411 +*828 *+1659
>> e = EjemploHilos.new;10.times{e.run}
+0 *+6 *+*18 42 +*+90 **186 +375 +**756 ++1515 *3036
```

Devolviendo la predictibilidad

- Tenemos un programa *explícitamente hecho para fallar*
  - Con muchos *vicios* en su código
- Ilustra cómo los hilos pueden enredarse entre sí
- . . . ¿Cómo desenmarañar la madeja?

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¿Bastará con proteger con mutex?

```
class EjemploHilos
  def initialize
    @x = 0
    @mut = Mutex.new
  end
  def run
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1
    @mut.lock
    print ' %d '  % @x
    @mut.unlock
  end
  def f1
    sleep 0.1
    @mut.lock
    print '+'
    @x += 3
    @mut.unlock
  end
  def f2
    sleep 0.1
    @mut.lock
    print '*'
    @x *= 2
    @mut.unlock
  end
end
```

¿Será con esto suficiente? ¿Tendremos resultados consistentes?

¿Para qué intento proteger con el mutex al `print` en `run`?

Un mutex no es suficiente

```
>>  e = EjemploHilos.new;10.times{e.run}
0 **++3 *12 **+*27 +57 +*120 *+243 +*492 +*+993 *1986 +=> 10
>> * e = EjemploHilos.new;10.times{e.run}
0 +**6 +15 +*36 +*+78 *162 +*330 +*666 +**1338 +2679 => 10
>> *
```

- Nuestro problema no *sólo* venía del acceso concurrente a `@x`
- Sino que al ordenamiento relativo
  - No importa que entren a la vez
  - $(a + b) \times c \neq a + (b \times c)$
- ¿Cómo podemos asegurar una invocación ordenada *sin perder el paralelismo*?
  - Las tres funciones (`f1`, `f2` y `run`) incluyen un `sleep(0.1)`
  - Que ese `sleep` reperesente nuestro código *paralelizable*

Solución 1: Esperar a la finalización de los hilos

Del mismo modo que podemos lanzar un hilo nuevo (en Ruby, `Thread.new {...}`; en Python, `threading.Thread(...).start()`), podemos esperar a que termine, con `join`

```
class EjemploHilos
  def initialize
    @x = 0
  end
  def run
    t1 = Thread.new {f1}
    t2 = Thread.new {f2(t1)}
    sleep 0.1
    t2.join
    print ' %d '  % @x
  end
  def f1
    sleep 0.1
    print '+'
    @x += 3
  end
  def f2(t)
    sleep 0.1
    t.join
    print '*'
    @x *= 2
  end
end
```

Esperar a la finalización

```
>> e = EjemploHilos.new;10.times{e.run}
+*6 +*18 +*42 +*90 +*186 +*378 +*762 +*1530 +*3066 +*6138 => 10
>> e = EjemploHilos.new;10.times{e.run}
+*6 +*18 +*42 +*90 +*186 +*378 +*762 +*1530 +*3066 +*6138 => 10
```

- Código resultante *claro*, aunque más rígido
  - ¿Qué tendríamos que modificar para que la multiplicación ocurra *antes de* la suma?
- *Mayor acoplamiento* entre las tres funciones
  - `f2` debe recibir una referencia al hilo 1 (`t`)
  - Cada función debe *comprender* su papel en la *línea de ensamblaje*
- ¿Aprovecho efectivamente el paralelismo?
  - ¿Se ejecutan *a la vez* las secciones no críticas? (los `sleep 0.1`)

Modificando el flujo

```
def run
  t1 = Thread.new {f1}
  t2 = Thread.new {f2(t1)}
  sleep 0.1
  t2.join
  print '%d ' % @x
end
def f1
  sleep 0.1
  print '+'; @x += 3
end
def f2(t)
  sleep 0.1
  t.join
  print '*'; @x *= 2
end
```

→

```
def run
  t1 = Thread.new {f2}
  t2 = Thread.new {f1(t1)}
  sleep 0.1
  t2.join
  print '%d ' % @x
end
def f1(t)
  sleep 0.1
  t.join
  print '+'; @x += 3
end
def f2
  sleep 0.1
  print '*'; @x *= 2
end
```

```
e = EjemploHilos.new;10.times{e.run}
*+3 *+9 *+21 *+45 *+93 *+189 *+381 *+765 *+1533 *+3069 => 10
```

Solución 2: Variables de condición

```
class EjemploHilos
  require 'thread'
  def initialize
    @x = 0; @estado = 0
    @mut = Mutex.new
    @cv = ConditionVariable.new
  end
  def run
    @estado = 0
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado != 2)
    puts ' %d '  % @x
    @mut.unlock
  end
  def f1
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado != 0)
    @x += 3; @estado += 1
    @cv.broadcast
    @mut.unlock
  end
  def f2
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado != 1)
    @x *= 2; @estado += 1
    @cv.broadcast
    @mut.unlock
  end
end
```

Sincronización a través de CV

Desventaja Código resultante *más complejo* (más elementos a considerar), aunque más flexible

- Ejemplo de complejidad: No *me* funcionó tras un tiempo razonable :-P
- Podría mejorarse con un poco de *azucar sintáctico*

Ventaja Menor acoplamiento entre funciones

- Ordenamiento determinado por `@estado`
- Podría configurarse en un sólo punto — Ejemplo a continuación

Solución 3: CVs, ordenamiento centralizado

```
class EjemploHilos
  require 'thread'
  def initialize
    @orden = {:f1 => 0, :f2 =>
        1, :run => 2}
    @x = 0; @estado = 0
    @mut = Mutex.new
    @cv = ConditionVariable.new
  end
  def run
    @estado = 0
    t1 = Thread.new {f1}
    t2 = Thread.new {f2}
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado !=
        @orden[:run])
    puts ' %d ' % @x
    @mut.unlock
  end
  def f1
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado !=
        @orden[:f1])
    @x += 3; @estado += 1
    @cv.broadcast
    @mut.unlock
  end
  def f2
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
        (@estado !=
        @orden[:f2])
    @x *= 2; @estado += 1
    @cv.broadcast
    @mut.unlock
  end
end
```

¿Y cómo es que *el intérprete* detecta...?

- Por varios semestres(!) *no pude lograr* que el código que les presenté funcionara
  - El intérprete *detecta un bloqueo mutuo*
  - Indica, sin duda, un error del programador

```
>> EjemploHilos.new.run
fatal: deadlock detected
        from /usr/lib/ruby/1.9.1/thread.rb:71:in
            'sleep'
        from /usr/lib/ruby/1.9.1/thread.rb:71:in
            'wait'
        (...)
```

  - Y, sí, derivaba de que llamaba a `@cv.broadcast(@mut)` (argumento de más)
- Pero... ¿Cómo se dio cuenta?
  - ¿Cómo puede Ruby *prevenir* este bloqueo y *notificar al programador*? ¿¡Antes de la primer iteración!?

¡No se lo pierda!

- ¡Muy pronto!
- ¡A la misma hora! ¡Por el mismo canal!

Bloqueos mutuos

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¡No se lo pierda!

- ¡Muy pronto!
- ¡A la misma hora! ¡Por el mismo canal!

Bloqueos mutuos

¡Prevención!

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¡No se lo pierda!

- ¡Muy pronto!
- ¡A la misma hora! ¡Por el mismo canal!

Bloqueos mutuos

¡Prevención!

¡Evasión!

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¡No se lo pierda!

- ¡Muy pronto!
- ¡A la misma hora! ¡Por el mismo canal!

Bloqueos mutuos

¡Prevención!
¡Evasión!
¡Detección y recuperación!

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

¡No se lo pierda!

- ¡Muy pronto!
- ¡A la misma hora! ¡Por el mismo canal!

Bloqueos mutuos

¡Prevención!
¡Evasión!
¡Detección y recuperación!

...Y la triste realidad

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM