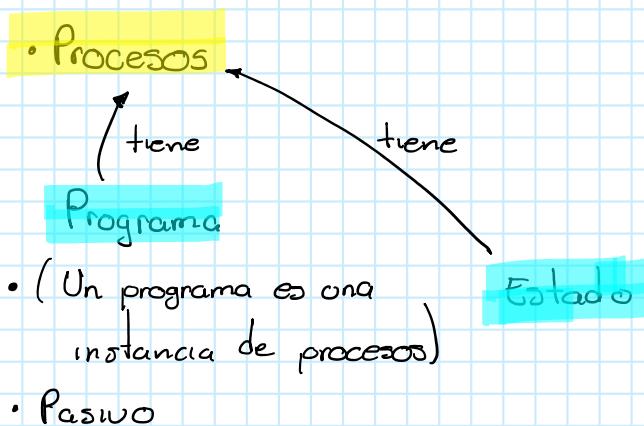


Procesos e hilos

Tuesday, 18 February 2020 6:43 PM



* Muchos procesos pueden tener un programa

* Cambios de contexto para cambiar de proceso inmediatamente

La ilusión de la concurrencia

- Un sistema actual nos da la *ilusión* de ejecución simultánea de muchos procesos
- La realidad: Casi todos están suspendidos, esperando que los active el planificador
- En un momento dado sólo puede estarse ejecutando un número de procesos igual o menor al número de CPUs que tenga el sistema.
- Esa ilusión tiene grandes costos... Especialmente pensando con suficiente velocidad

Gunnar Wolf • sistop@bgwolf.org • <http://sistop.gmwolf.org>
Sistemas Operativos (SSO) • Facultad de Ingeniería UNIRIA

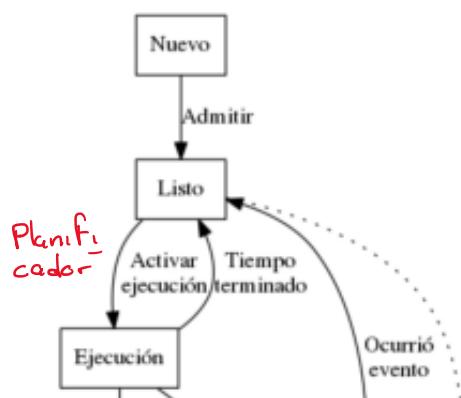


Gunnar Wolf

Administración de procesos: Procesos e hilos

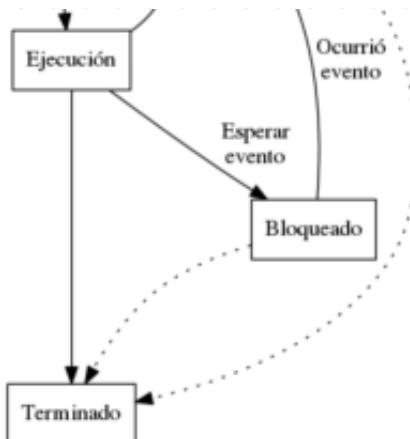
Proceso

- En qué estado puede estar un proceso



- Función "fork" python 3

$p1t = fork()$
 $p1t = 0$ $p1d = (int > 0)$



$p_{\text{int}} = 0$

X

$p_{\text{id}} = (\text{int} > 0)$

padre

// Si no lo recoge
el padre

El hijo se
vuelve zombie

El bloque de control del proceso (PCB) (1)

¿Qué información debe mantener el sistema acerca de cada proceso?

O, ¿Cuánto cuesta un cambio de contexto?

Estado del proceso El estado actual del proceso

Contador de programa Cuál es la siguiente instrucción a ser ejecutada

Registros del CPU Información específica del estado del CPU

Información de planificación (scheduling) La prioridad del proceso, la cola en que está agendado, y demás información que puede ayudar al sistema operativo a agendar al proceso



Administración de procesos: Procesos e hilos

Concepto y estados de un proceso

- El proceso
- Hilos
- Los hilos para el sistema operativo
- Concurrencia

El bloque de control del proceso (PCB) (2)

Información de administración de memoria Las tablas de mapeo de memoria (páginas o segmentos, dependiendo del sistema operativo).

Estado de E/S Listado de dispositivos y archivos asignados que el proceso tiene abiertos en un momento dado.

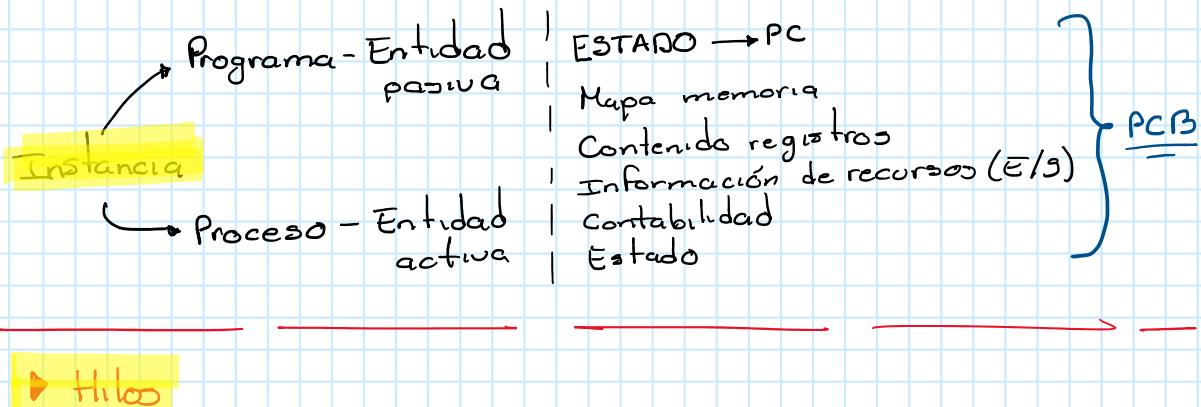
Información de contabilidad Información de la utilización de recursos que ha tenido este proceso

- Tiempo de usuario y de sistema
- Uso acumulado de memoria y dispositivos
- etc.



Hilos

Thursday, 20 February 2020 5:55 PM



- Si mi proceso no tiene hilos, entonces tiene un hilo.
- Si mi sistema operativo no conoce de hilos
 - Cada hilo es un proceso

* Bloque de Control de Procesos (PCB)

- La cantidad de información implicada en un cambio de contexto es muy grande
- Desperdicio burocrático de recursos
- Respuesta → Procesos ligeros (LWP)
 - Diversos hilos de ejecución dentro de un mismo proceso.

La diferencia desde dentro

- Entre diferentes procesos, cada uno tiene ilusión de exclusividad virtual sobre la computadora
 - Espacio de direccionamiento de memoria exclusivo
 - Descriptores de archivos y dispositivos independientes
- Los hilos son y deben ser conscientes de su coexistencia
 - Comparten memoria, descriptores de archivos y dispositivos
 - Pueden tener variables locales y globales





Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

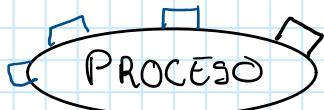
¿Qué tanto se aligera el PCB?

- Estado del proceso
- Cada hilo se ejecuta de forma aparentemente secuencial
 - Tiene su propio *contador de programa*
 - Y su propia pila de llamadas (*stack*)
- Conjunto de variables locales
 - Puede ser implementado de forma muy sencilla
 - p.ej. con arreglos indexados por ID de hilo
- Información de planeación *interna*
- Pueden llevar información de contabilidad
- **No requieren:**
 - *Registros del CPU* ← *sacar y meter CPU en el mapa de memoria*
 - Estado de E/S



← sacar y meter
CPU en el mapa
de memoria

//Analogía



— Maneras para levantar
un proceso → hilos

Patrones de trabajo: Jefe / trabajador

- Un hilo tiene una tarea distinta de todos los demás
- **El hilo jefe genera o recopila tareas**
- **Los hilos trabajadores efectúan el trabajo.**
- **Modelo común para procesos servidor, GUIs que procesan eventos** ← *Programación Orientada a Eventos*
- El jefe puede llevar la contabilidad de los trabajos realizados



Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Patrones de trabajo: Jefe / trabajador

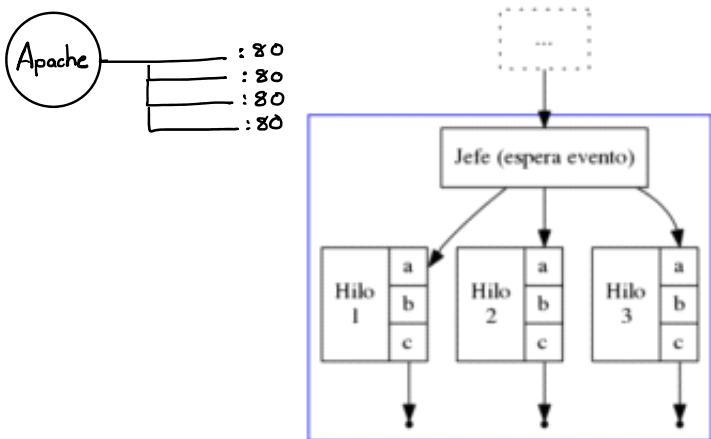


Figura: Patrón de hilos jefe/trabajador



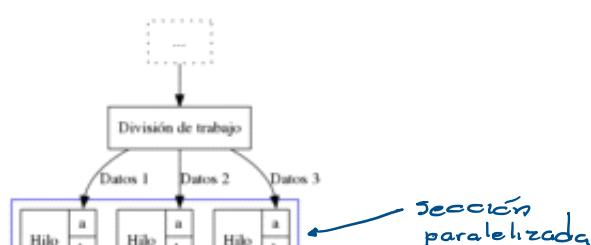
Patrones de trabajo: Equipo de trabajo

- A partir de hilos idénticos
- Realizarán las mismas tareas sobre diferentes datos
- Muy frecuentemente utilizado para cálculos matemáticos (p.ej. criptografía, render).
- Puede combinarse con jefe/trabajador para generar previsualizaciones (la tarea se realiza progresivamente)



Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Patrones de trabajo: Equipo de trabajo



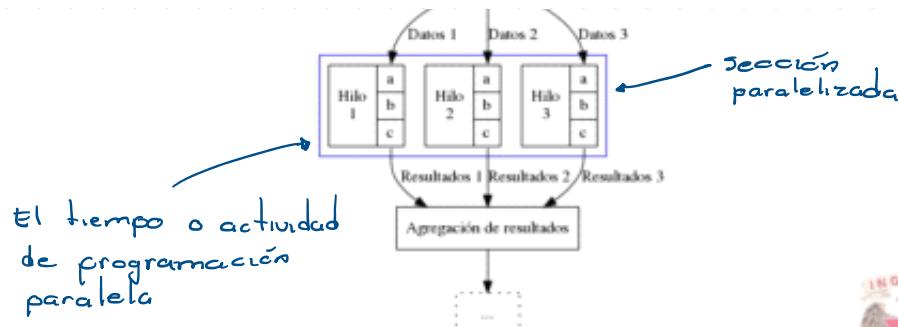


Figura: Patrón de hilos *Equipo de trabajo*



• sistop@gwolf.org • <http://sistop.gwolf.org>
jerátivos (1554) • Facultad de Ingeniería UNAM

Patrones de trabajo: Línea de ensamblado

- Una tarea larga que puede dividirse en pasos
 - Cada hilo se enfoca en una sola tarea
 - Pasa los datos a otro hilo conforme va terminando
 - Ayuda a mantener las rutinas simples de comprender
 - Permite continuar procesando datos cuando hay hilos esperando E/S



卷之三

Gunnar Wolf

Administración de procesos: Procesos e hilos

Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Patrones de trabajo: Línea de ensamblado

* Para mango de buffers

- * En hardware
- * En software

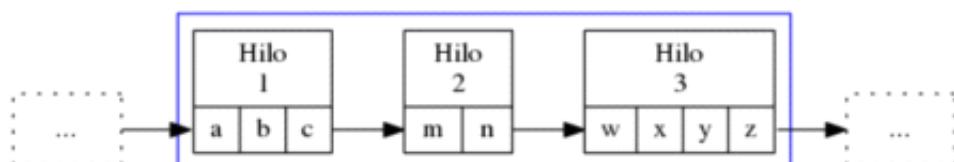


Figura: Patrón de hilos *Línea de ensamblado*

Gunnar Wolf • sistop@gwolf.org • <http://gwolf.org>
Sistemas Operativos (1554) • Facultad de In

Hilos de usuario (o hilos verdes)

Hilos de usuario (o hilos verdes)

- Los hilos pueden implementarse 100 % con las facilidades del proceso
 - Caso extremo: Programas multihilos en sistemas operativos mínimos (o directo sobre el hardware)
- Mayor portabilidad
- A través de alguna biblioteca *del lenguaje/entorno de programación*
- Típicamente multitarea *interna cooperativa*



Gunnar Wolf

Administración de procesos: Procesos e hilos

Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Pros y contras de los hilos de usuario

Ganamos:

- Espacio de memoria compartido sin intervención del sistema operativo
- Mayor rapidez para realizar trabajos cooperativos o multiagentes

Perdemos:

- Cualquier llamada al sistema interrumpe a todos los hilos
- No aprovechan el multiprocesamiento



Hilos nativos (o hilos de kernel)

- A través de bibliotecas de sistema que *informan* al sistema
- El núcleo se encarga de la multitarea *preventiva* entre los hilos
- El proceso puede pedir al sistema un mayor número de

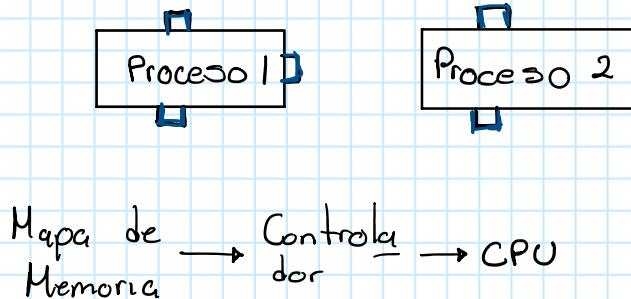
- El núcleo se encarga de la multitarea *preventiva* entre los hilos
- El proceso *puede* pedir al sistema un mayor número de procesadores
 - Logrando ejecución verdaderamente paralela
- El sistema *puede* priorizar de diferente manera a un proceso multihilo



Hilos 2

Tuesday, 25 February 2020 5:41 PM

→ Cambio de contexto



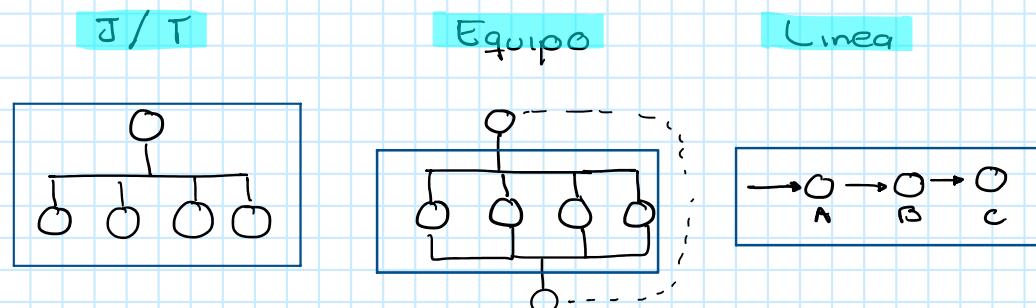
- Multihilos → La forma de
- Multiprocesos → entender el mundo será distinto
- Si se comparte información → Multihilos
- * Código reentrant → Código para aplicarle hilos

• Cambios de contexto
↳ Cambian entre procesos

- Hilos

- ↳ Tienen parte del programa
- ↳ No tienen protección de memoria

Patrones de Hilos



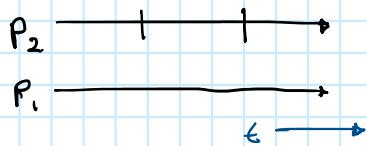
* Es común que el jefe lance procesos y supervise

- Parte paralela del patrón

Concurrencia

- Nos referimos a 2 o más eventos cuyo orden no es determinista

- Nos referimos a 2 o más eventos cuyo orden no es determinista



- No podemos predecir el orden relativo en el que ocurrirán

¿Cuándo hablamos de concurrencia?

- Dos o más hilos del mismo proceso
- Dos o más procesos en la misma computadora
- Dos o más procesos en computadoras separadas conectadas por red
- Dos o más procesos que ocurran sin conexión alguna y posteriormente requieran sincronización



La sincronización implica involucramiento del programador

El sistema operativo brinda la *ilusión* a cada proceso de estar ejecutando en una computadora dedicada, pero...

- El proceso puede depender de datos obtenidos en fuentes externas
- Puede ser necesario esperar a que otro proceso haya pasado cierto punto en su ejecución
 - ... Que tengamos ya los resultados parciales de un cómputo paralelo
 - ... O que no más de m o menos de n procesos estén en un determinado punto
 - ... O notificar a otro proceso de nuestro avance, o...



La sincronización como comunicación entre procesos (IPC)

Parte donde _____

Parte donde _____
podemos
encontrarla

- En esta sección veremos ejemplos de *primitivas de sincronización*
- Todas ellas son modalidades de *comunicación entre procesos* (IPC)
- Nos centraremos principalmente en los *semáforos*



Gunnar Wolf Administración de procesos: Procesos e hilos

Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Problemas clásicos para ir pensando

- Problema productor-consumidor
- Problema lectores-escritores
- La cena de los filósofos
- Los fumadores compulsivos

... Y hay ≈ 20 problemas más explicados en *The little book of semaphores*.

¡Revísenlo!



Problema productor-consumidor

- Pensemos un entorno multihilo como una línea de ensamblado
 - Algunos hilos *producen* ciertas estructuras (p.ej. eventos en un GUI)
 - Otros hilos las *consumen* (procesan los eventos)
- Los eventos se *apilan* en un buffer disponible para todos los hilos
- ¿Cómo podemos asegurar que dos hilos no modifiquen al buffer al mismo tiempo?
- ¿Cómo podemos evitar que los consumidores hagan ~~espeña~~ *espera activa*?

while (banda.size = 0)

pass

Espera activa

Problema productor-consumidor

- Pensemos un entorno multihilo como una línea de ensamblado
 - Algunos hilos *producen* ciertas estructuras (p.ej. eventos en un GUI)
 - Otros hilos las *consumen* (procesan los eventos)
- Los eventos se *apilan* en un buffer disponible para todos los hilos
- ¿Cómo podemos asegurar que dos hilos no modifiquen al buffer al mismo tiempo?
- ¿Cómo podemos evitar que los consumidores hagan *espera activa*?

while (banda.size = 0)
 pass
} Espera activa



Gunnar Wolf

Administración de procesos: Procesos e hilos

Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Problema lectores-escritores

→ Necesita que no haya un lector activo

- Muchos procesos *lectores* pueden usar simultáneamente una estructura
 - Si algún proceso *escritor* la requiere, debemos evitar que cualquier *lector* esté activo
- Requisitos de sincronización:
 - Sin límite en la cantidad de *lectores activos*
 - Los *escritores* deben tener acceso exclusivo a la sección crítica
 - Evitar *inanición* de escritores por exceso de lectores



La cena de los filósofos (Edsger Dijkstra, 1965)

- Una mesa redonda
- Tazón de arroz al centro
- Cinco platos, cinco filósofos, cinco palillos chinos
- Los filósofos piensan hasta tener hambre.
 - Cuando tienen hambre, buscan comer
 - No hablan entre sí
- Sólo un filósofo puede sostener un palillo a la vez
- ¿Qué puede salir mal?
 - ¿Cómo evitarlo con sólo primitivas de sincronización?



Gunnar Wolf

Administración de procesos: Procesos e hilos

Concepto y estados de un proceso
El proceso
Hilos
Los hilos para el sistema operativo
Concurrencia

Los fumadores compulsivos (Suhas Patil, 1971)

Los fumadores compulsivos (Suhas Patil, 1971)

- Tres fumadores empedernidos
 - Con cantidades ilimitadas de uno de tres ingredientes cada uno: Tabaco, papel, cerillos
- Un agente que consigue ingredientes independientemente
- ¿Cómo asegurarse de que los recursos se utilizan siempre, *tan pronto como estén disponibles*?
 - Sin que el agente sepa quién tiene qué ingrediente



Los problemas de la concurrencia (1)

```
def f2
    sleep 0.1
    print '*'
    @x *= 2
end
def f1
    sleep 0.1
    print '+'
    @x += 3
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

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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 orden de la ejecución es (*¿parece ser?*) ($@x * 2$)
+ 3, a veces ($@x + 3$) * 2
 - A veces la impresión ocurre en otro orden: ***756 o ++1515
- Esto porque tenemos una **condición de carrera** en el acceso a la variable compartida



Condición de carrera (Race condition)

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 Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

Operación atómica

- Operación que tenemos la garantía que se ejecutará o no como *una sola 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

INGENIERIA

Sección crítica

Es el área de código que:

- Realiza el acceso (¿modificación? ¿lectura?) de datos compartidos
- Requiere ser *protugida 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!



INGENIERIA



Sección crítica

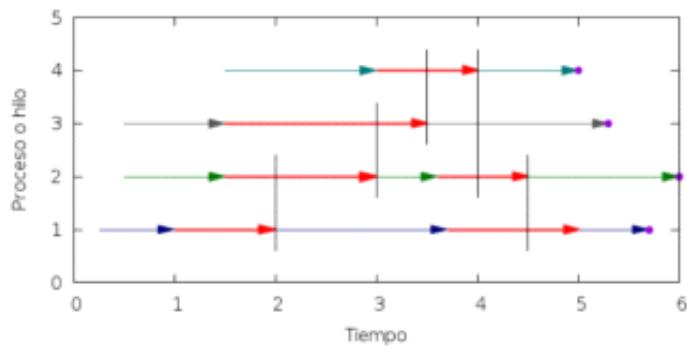


Figura: Sincronización: La exclusión de una sección crítica común a varios procesos se protege por medio de regiones de exclusión mutua



- * Sólo un proceso avanza cuando estamos en una sección crítica

Concurrencia

Thursday, 27 February 2020 5:43 PM

Condición de Carrera

- Rivalidad por el acceso de un espacio compartido
 - Sección crítica
 - ↳ Disfraz de "operación atómica"
 - ↳ Proteger ante simultaneidad

Bloqueo mutuo

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

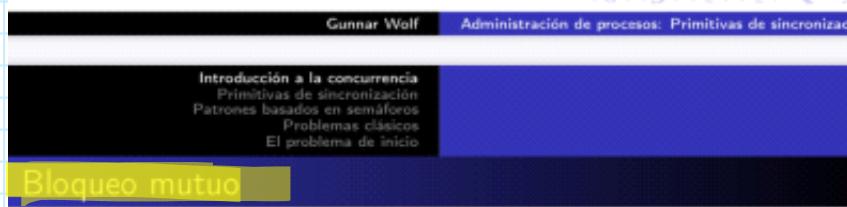
- Dos o más procesos poseen determinados recursos
 - Cada uno de ellos queda detenido esperando a alguno de los que tiene otro

* Perdida de información

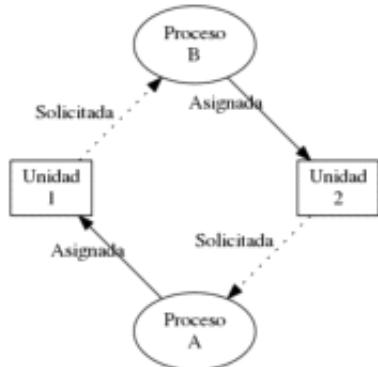


A → x
B → y

DEADLOCK



Bloqueo mutuo



Iniciación → No tener la oportunidad de acceso

espera activa → Desperdicio de memoria



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

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.



Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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;
}
```

→ No es
atómico



¿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 Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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



Solución Perl

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

► Primitivas de Sincronización

► Primitivas de Sincronización

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

• Hay forma de asegurar los hilos

- Alg. Dekker
- Solución Peterson



Gunnar Wolf Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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



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



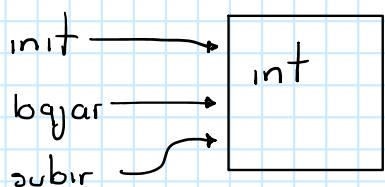
Gunnar Wolf Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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.



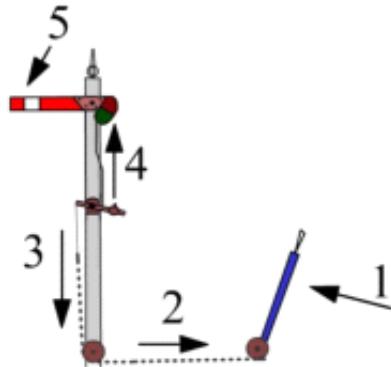


Figura: Semáforo de tren (Fuente: Wikipedia, Señalización ferroviaria argentina)

Gunnar Wolf Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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*).

→ Sección crítica
: Dormir

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*



Gunner Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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);
```



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

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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()



Critica

Soluciones en hardware

Embebido → hardware dedicado a algo específico

- 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



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





Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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;  
}
```



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 sola instrucción
 - ... Muy caro de implementar en arquitecturas CISC
- Susceptible a problemas de coherencia de cache



Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

Memoria transaccional

- Idea base: Semántica de bases de datos en el acceso a memoria
- Permite agrupar varias operaciones en una sola *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*

- 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



Patrones basados en Semáforos

Tuesday, 3 March 2020 5:40 PM

Reaso

Condición de carrera - Operación Atómica - Conurrencia - Bloqueo - Inanición Mutuo

↓
Espera circular eterna por un conjunto de procesos a un conjunto de recursos

↓
Competencia entre procesos penaliza do a uno con espera excesiva

↓
Matar procesos

Sincronización

① Primitivas: (Viene de abajo)

→ las da el S.O.

② Mutex - Candado

- int / lock / unlock

③ Semáforos

- Variables de condición
- Memoria transaccional ↗ SW ↘ HW
- test and set

► Señalizar

- Semáforo en 0, está cerrado
 - El primero que lo intente adquirir se va a dormir

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=preparaConexion, args=[semaf]).start()
Thread(target=enviaDatos, args=[semaf]).start()
```

```
def preparaConexion(semaf):
    creaConexion()
    semaf.release()
```

```
def enviaDatos(semaf):
    calculaDatos()
    semaf.acquire()
    enviaPorRed()
```



```
crea_conexion()  
semaf.release()
```

```
semaf.acquire()  
envia_por_red()
```



Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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



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)



Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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()
```



Terciugusto

```
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()
```

Umo de los
hilos hará esta parte

- 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

Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

Apagador

- Principalmente empleados 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)



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

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

Barrera: Código ejemplo

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

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

} Torniquete

- 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);
```



Gunnar Wolf

Administración de procesos: Primitivas de sincronización

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()
```

2 clases de hilos

= Muy similar
a Rendezvous

Para hacer solo
una pareja bailar



HUTEX

Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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 orden en que llegaron



Problema Clásicos

Thursday, 5 March 2020 5:45 PM

Abstract

① Un hilo controla a otro

$h1.\text{hagaAlgol} \Rightarrow h1.\text{release} \rightarrow h2.\text{acquire}$

Rendezvous

h1

h2

$s2.\text{release} \rightarrow s1.\text{release}$
 $s1.\text{acquire} \rightarrow s2.\text{acquire}$

Barrera / Torniquete

torniquete:

Barrera:

mutex.acquire

$c = c + 1$

$\text{if } c == \text{total}$

$b.\text{release}()$

mutex.release()

$b.\text{acquire}$

$b.\text{release}$

$\text{total} = 5$

$c = 0$

m b

1	0
---	---

► Problemas Clásicos

NOTA

- Lenguajes de script utilizan GIL
- Antes Linux tenía algo similar denominada BKL

log al principio
log
← por un
MUTEX

Protegido

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



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(0)
        procesa(event)
  
```

Buffer = []

mutex.acquire()

mutex.release()

señal.realiza

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



mut. acquire

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=[1]).start()
  
```

* Esto podría ser
un multiplex

```

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()

```



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*

Lector → Entró / Leo / Salgo

Escritor → Entró / Escribe / Salgo



Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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()
    escribir()
    cuarto_vacio.release()
    #end_sic python
    # latex: \end{column}

```



def lector

* Cosa |

```

def lector():
    global lectores
    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()

```

* Caso de inanición

¿Cuándo entrará

al salón si siempre está ocupado?

Lectores-escritores sin inanición

```

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

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()

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

```

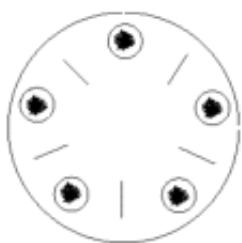
Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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



La cena de los filósofos: Planteamiento (2)

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?

Navigation icons

Gunnar Wolf

Administración de procesos: Primitivas de sincronización

Introducción a la concurrencia
 Primitivas de sincronización
 Patrones basados en semáforos
Problemas clásicos
 El problema de inicio

Cena de filósofos: Bloqueo mutuo

A circular diagram representing a table where five philosophers are seated. Each philosopher is depicted as a head with two arms. In front of each philosopher is a pair of chopsticks (represented by small circles with crosses). The chopsticks are positioned such that each philosopher has one chopstick in each hand, illustrating a state where no two adjacent philosophers have both chopsticks, thus preventing them from both picking up their chopsticks simultaneously.

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

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

Cena de filósofos: Inanición

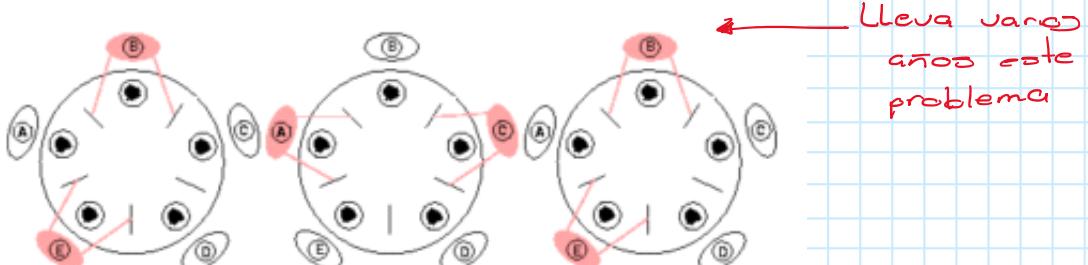


Figura: Una rápida sucesión de C y E lleva a la inanición de D



Gunnar Wolf Administración de procesos: Primitivas de sincronización

- 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 $\text{palillos}[i]$ y $\text{palillos}[(i+1) \% n]$



Concurrencia

Tuesday, 10 March 2020 6:22 PM

Volviendo al problema de la concurrencia

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

```
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

class EjemploHilos
  def initialize
    @x = 0
  end
  def f1
    sleep 0.1
    print '+'
    @x += 3
  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 Administración de procesos: Primitivas de sincronización
```



Introducción a la concurrencia
Primitivas de sincronización
Patrones basados en semáforos
Problemas clásicos
El problema de inicio

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?

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 != 0)
    @x += 3; @estado += 1
    @cv.broadcast
    @mut.unlock
  end
  def f1
    sleep 0.1; @mut.lock
    @cv.wait(@mut) while
      (@estado != 0)
    @x *= 2; @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
```





Introducción a la concurrencia
 Primitivas de sincronización
 Patrones basados en semáforos
 Problemas clásicos
 El problema de inicio

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 azúcar *sintáctico*

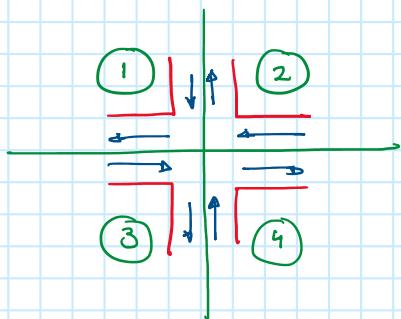
Ventaja Menor acoplamiento entre funciones

- Ordenamiento determinado por @estado
- Podría configurarse en un sólo punto — **Ejemplo**



► Problema / Cruce caminos

- Hay un cruce de caminos



* Cada sección crítica
 (Evitar choques) se prevendrá
 con un **MUTEX**