Sistemas Operativos

Escuela de Ingeniería Civil en Informática
Universidad de Valparaíso, Chile

http://informatica.uv.cl

Universidad de Valparaíso CHILE

Sincronización

Ejemplo

Cada thread lee las líneas de un archivo determinado y agrega cada una a una estructura compartida

```
std::vector<std::string> g_textoConsolidado;

void fooOperation(std::string pathFile)
{
     std::ifstream inputFile(pathFile);

     for(std::string line; getline(inputFile, line); ){

          g_textoConsolidado.push_back(pathFile + ": " + line);

     }

     inputFile.close();

}
```

Ejemplo

```
$ ./example data/data01.txt data/data02.txt
double free or corruption (out)
Aborted (core dumped)
$
```

Ejecución paralela

```
$ taskset -c 0 ./example data/data01.txt data/data02.txt
[...]líneas mezcladas [...]
$
```

Ejecución concurrente

Concepto involucrado en este ejemplo: **Race Condition**

Error clásico en programación paralela/concurrente.
Resultado: estados inconsistentes.
Programa difícil de depurar.
Ocurre por no considerar la **no atomicidad** de una operación

Conceptos claves

Operación Atómica

Operación que se ejecuta como una sólo unidad de ejecución.
Una planificación apropiativa de esta operación no origina un estado inconsistente.

Exclusión Mutua

Garantía que si un proceso utiliza un recurso compartido, el resto NO puede utilizarlo.

Sección Crítica

Parte del código en la que se tiene acceso al recurso compartido

Características de las Soluciones

Universidad de Valparaíso CHILE

Dos procesos no deben encontrarse al mismo tiempo dentro de sus secciones críticas.

No se deben hacer hipótesis sobre la velocidad o el número de CPU.

Ningún proceso que esté ejecución fuera de su sección crítica puede bloquear a otros procesos.

Ningún proceso debe esperar eternamente para entrar a su sección crítica.

Algunas soluciones

Variables mutex

MUTual **EX**clusion

Mecanismo que asegura que la sección protegida del código se ejecutará cómo si fuera atómica

El planificador puede expulsar el proceso.

Cada proceso debe implementar el mutex.

Mantiene en espera a los procesos adicionales que quieran emplearlo (no garantiza orden)

MUTEX en Threads Posix

```
pthread_mutex_t M;
pthread_mutex_init(&M, NULL);
pthread_mutex_lock(&M);
pthread_mutex_unlock(&M);
```

Variables mutex

Universidad de Valparaíso CHILE

MUTEX en Threads Posix

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Ver documentación en:

http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_mutex_lock.html
http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_mutex_init.html

Variables mutex

Universidad de Valparaíso CHILE

MUTEX en Threads Posix

```
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
//S.C
pthread_mutex_unlock(&mutex);
```

Es equivalente a

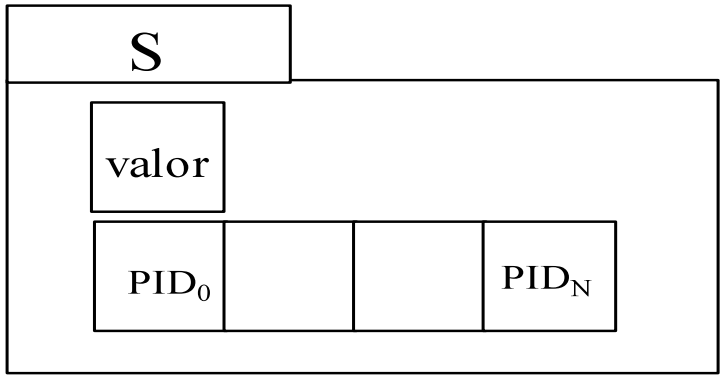
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);
//S.C
pthread_mutex_unlock(&mutex);
```

Variables semáforos

Universidad de Valparaíso CHILE

Edsger Dijkstra (1965)

```
Typedef struct{
    int value;
    struct proces *L;
} semaphore;
```




```
void wait(semaphore S) {
    S.value--;
    If(S.value<0){
      agregar_proceso(S.L, this);
      block(this);
    }
}
```

```
void signal(semaphore S) {
    S.value++;
    If(S.value<=0){
        P = sacar_proceso(S.L);
        wakeup(P);
    }
}
```

Operaciones básicas de semáforos

Inicializar

Puede inicializarse a cualquier valor entero no negativo

Decremento

`(wait, down,P)`

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.

Incrementar

`(signal, up, V)`

Incrementa en 1 el valor del semáforo. Si hay hilos esperando, uno de ellos es despertado.

Patrones de Diseño con Semáforos

Ejemplos

Universidad de Valparaíso CHILE

Mutex

```
semaphore mutex;
mutex.value=1;
```

(Ejemplo: Procesos con zona de exclusión mutua)

Multiplex

```
semaphore mutiplex;
multiplex.value=N; //N > 0
```

(Ejemplo: Productor/ consumidor)

Ejemplos

Universidad de Valparaíso CHILE

Señalización

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

```
1  def prepara_conexion(semaf):
2    crea_conexion()
3    semaf.release()
```

```
1  def envia_datos(semaf):
2    calcula_datos()
3    semaf.acquire()
4    envia_por_red()
```

Barrera

```
semaphore barrera;
barrera.value=0;
```

```
1  import random
2  from time import sleep
3  from threading import Semaphore
4  # n = Numero de hilos
5  n = random.randint(1,10)
6  cuenta = 0
7  mutex = Semaphore(1)
8  barrera = Semaphore(0)
```

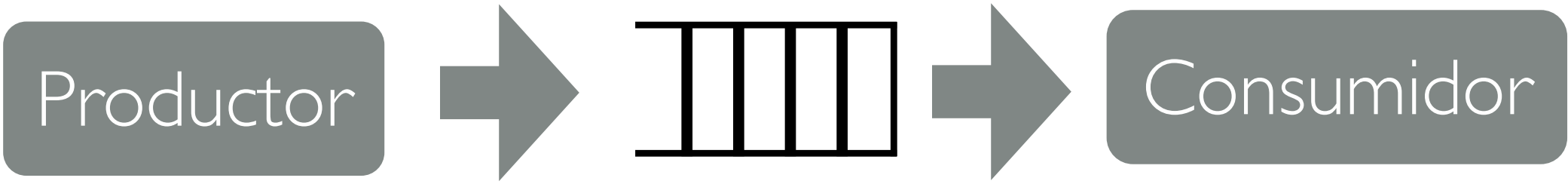
```
1  def vamos():
2    global cuenta
3    inicializa_estado()
4    mutex.acquire()
5    cuenta = cuenta + 1
6    mutex.release()
7    if cuenta == n:
8      barrera.release()
9    barrera.acquire()
10   barrera.release()
11   procesamiento()
```

Problemas Clásicos

Problema Productor-Consumidor

Universidad de Valparaíso CHILE

Transferencia de datos



Crea datos a cierta velocidad **λ**

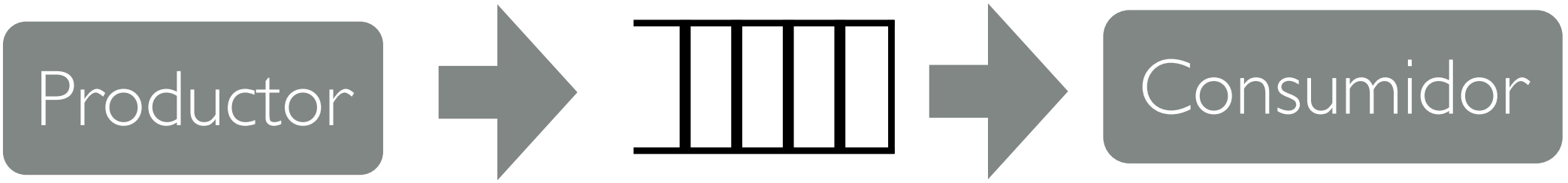
$\lambda \neq \mu$

Consume datos a cierta velocidad **μ**

Agrega datos cuando hay espacio en el búfer

Saca datos cuando hay por lo menos UN dato en el búfer

Problema Productor-Consumidor



Semaphore vacios.value=N;
Semaphore llenos.value=0;
Semaphore mutex = 1;

```
void productor(void){
  while(1){
    producir elemento;
    wait(vacios)
    wait(mutex)
    Almacenar elemento en el buffer
    signal(mutex)
    signal(llenos)
  }
}
```

```
void consumidor(void){
  while(1){
    wait(llenos)
    wait(mutex)
    Extraer elemento del buffer
    signal(mutex)
    signal(vacios)
    Procesar elemento
  }
}
```

Lectores-Escritores

Escenario:

Una estructura puede ser leído o modificada por diferentes procesos, denominados “Escritores” y “Lectores.

Mientras la estructura es modificada, los **lectores** no pueden acceder a ella.

Varios lectores pueden leer la estructura, en forma secuencial.

Un **escritor** no puede entrar a la estructura mientras ésta este siendo utilizada por otro proceso.

Lectores-Escritores

Escritor

```
wait(wrt);
        ...
    writing is performed
        ...
signal(wrt);
```

Lector

```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(wrt);
signal(mutex);
   ...
   reading is performed
   ...
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex);
```

wrt: mutex para los escritores

mutex: mutex para **readcount**

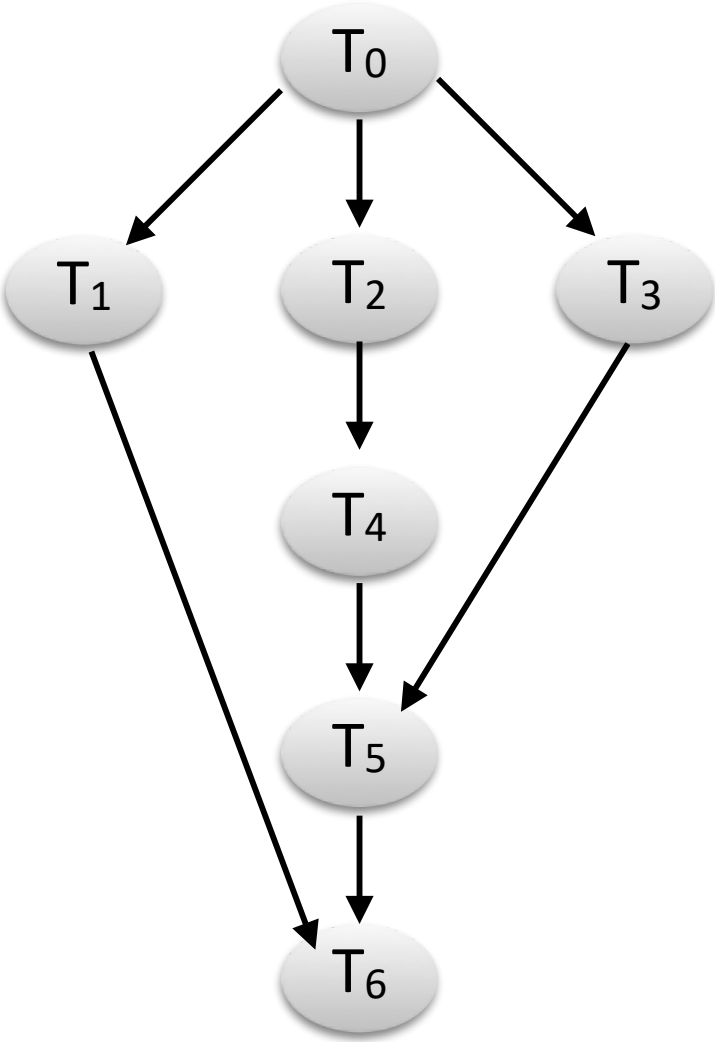
readcount: nº de lectores

Implementación de sincronización de tareas

Idea

Todas las tareas se inician al mismo tiempo

Algunas inician inmediatamente, otras deben esperar el término de la tarea que la precede.



Implementación de sincronización de tareas

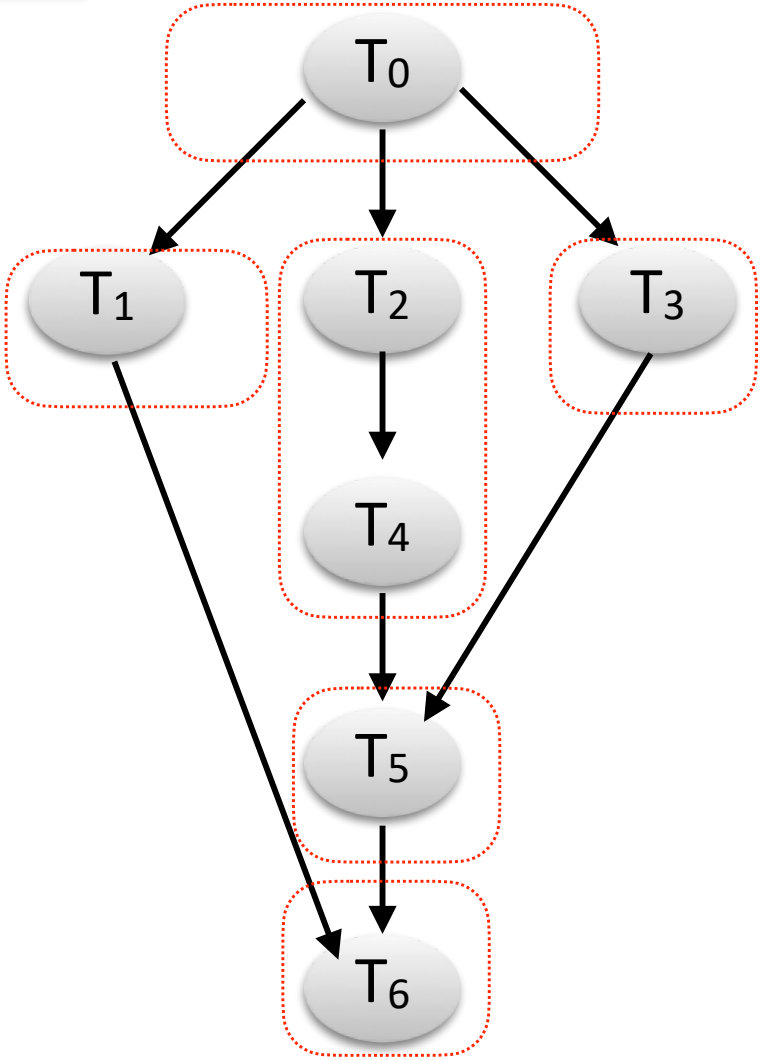
Idea

Todas las tareas se inician al mismo tiempo

Algunas inician inmediatamente, otras deben esperar el término de la tarea que la precede.

Agrupar las tareas que pueden realizar en paralelo y las tienen dependencias con otras.

Cada grupo se puede implementar como un thread que se ejecuta en forma paralela



Implementación de sincronización de tareas

Implementación con semáforos binarios y threads

(tipo barrera)

Cada arista es un semáforo

`semaphore s[8]`

`s[i].value = 0 ∀i`

Thread1

```
start(T0);
s1.signal();
s2.signal();
s3.signal();
```

Thread2

```
s1.wait()
start(T1);
s4.signal()
```

Thread3

```
s2.wait()
start(T2);
start(T4);
s5.signal();
```

Thread4

```
s3.wait()
start(T3);
s6.signal();
```

Thread5

```
s5.wait();
s6.wait();
start(T5);
s7.signal()
```

Thread6

```
s4.wait();
s7.wait();
start(T6);
```