Sistemas Operativos

Escuela de Ingeniería Civil Informática

• Administración de Procesos

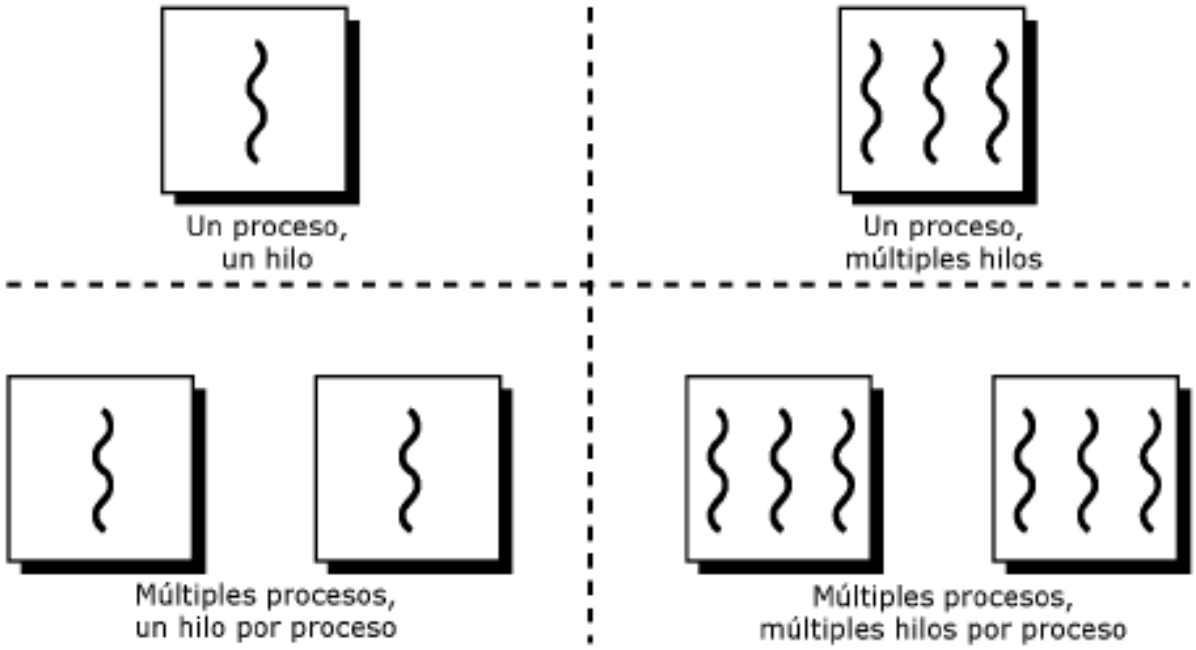
Trabajar con Hilos

Luis Gajardo - *lgajardo@ubiobio.cl*

UNIVERSIDAD DEL BÍO-BÍO

¿QUÉ ES UN PROCESO LIVIANO?

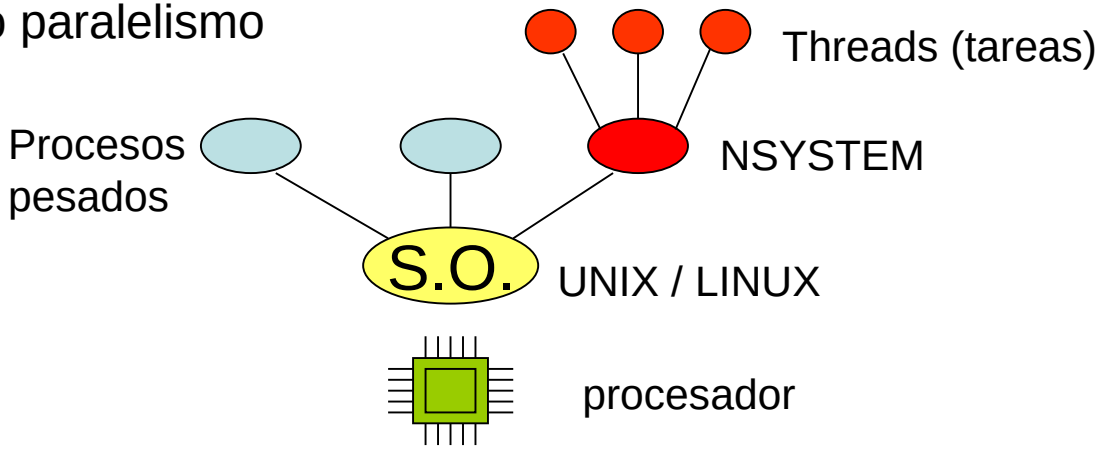
- Se llaman procesos livianos, hilos o threads.
- Los hilos comparten el mismo espacio de memoria.



¿QUÉ UTILIZAREMOS PARA APRENDER HILOS?

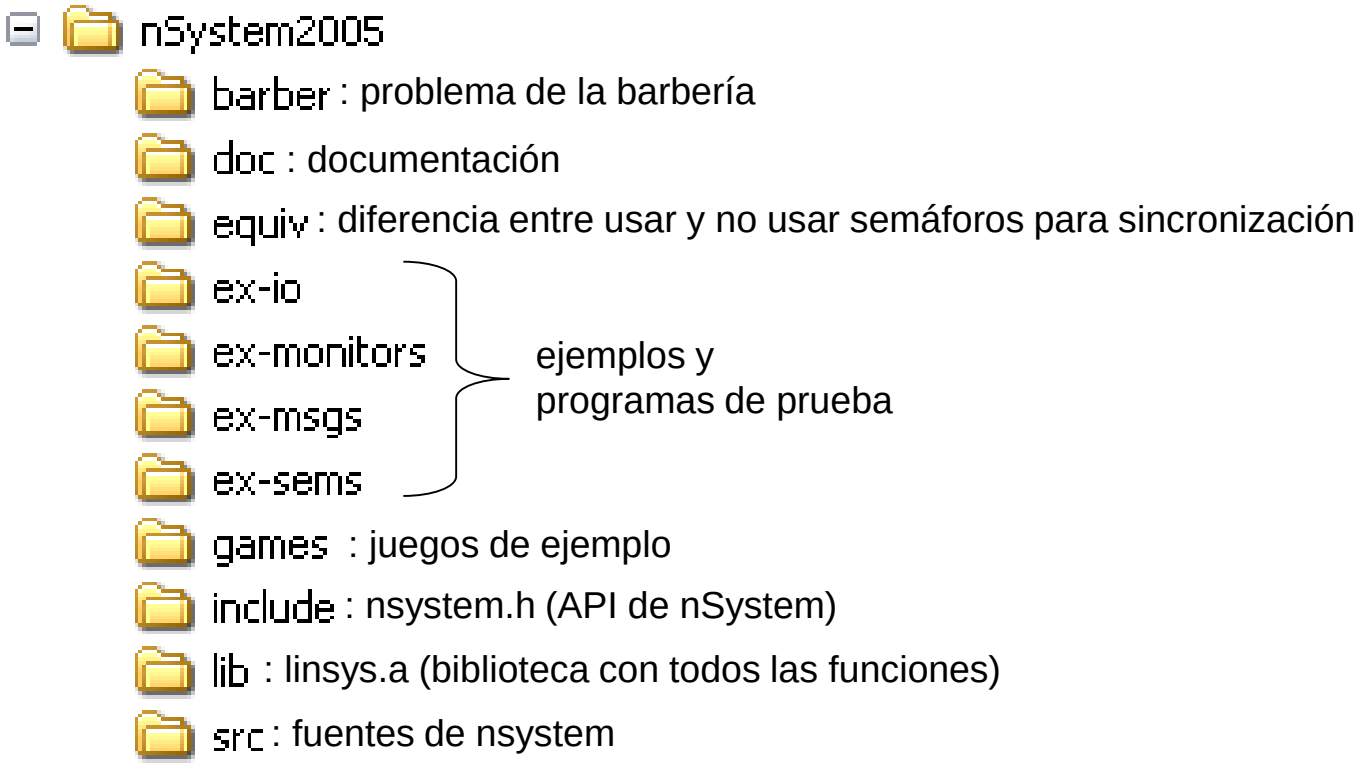
nSystem

- Realizaremos una disección a este pseudo S.O. diseñado con fines docentes.
- Características:
  - Procesos livianos: tareas (tasks)
  - Preemptive y non-preemptive
  - Corre en 1 proceso UNIX
  - No hay verdadero paralelismo



ORGANIZACIÓN DE NSYSTEM

- La estructura de directorio de nSystem es la siguiente:



PRIMEROS PASOS CON NSYSTEM

- ¿Cómo es un programa en NSYSTEM?

```
#include <nsystem.h>

int nMain(int argc, char **argv) {
  nprintf("Hola Mundo \n");
  return 0;
}
```

hola.c

- ¿Cómo se instala NSYSTEM en Linux?
  - Se descomprime el archivo a una carpeta en la ubicación deseada
  - Se configuran las rutas para que el compilador encuentre:
    - En csh → `setenv NSYSTEM <directorio raiz de NSYSTEM>`
    - En sh → `NSYSTEM <directorio raiz de NSYSTEM>`
      `export NSYSTEM`

PRIMEROS PASOS CON NSYSTEM

- ¿Cómo se compila en NSYSTEM?

```
$ gcc -c -ggdb -I$NSYSTEM/include hola.c
$ ls
  hola.c  hola.o
$ gcc -ggdb hola.o $NSYSTEM/lib/libnsys.a -o hola
$ ./hola
  Hola Mundo
$
```

API DE NSYSTEM

- ***nEmitTask***
  Emite o lanza una nueva tarea que ejecuta el procedimiento “proc”.
  Acepta un máximo de 6 parámetros (enteros o punteros) p1, p2, p3, ….
  Retorna un descriptor de tarea.

- Prototipo:
  *nTask nEmitTask(int (*proc)(), int p1, int p2, … int p6)*

- Ejemplo:

```
int loop() {
  for(;;);
  return 0;
}
```

```
int nMain(int argc, char *argv) {
  nTask t= nEmitTask(loop);
  for(;;);
  return 0;
}
```

API DE NSYSTEM

- ***nExitTask***
  Termina la ejecución de la tarea que la invoca. “rc” es el código de retorno de la tarea.

- Prototipo:
  `void nExitTask(int rc);`

- Ejemplo:
```
int i;
for(i=0; i<100000; i++);
nExitTask(1);
return 0;
```

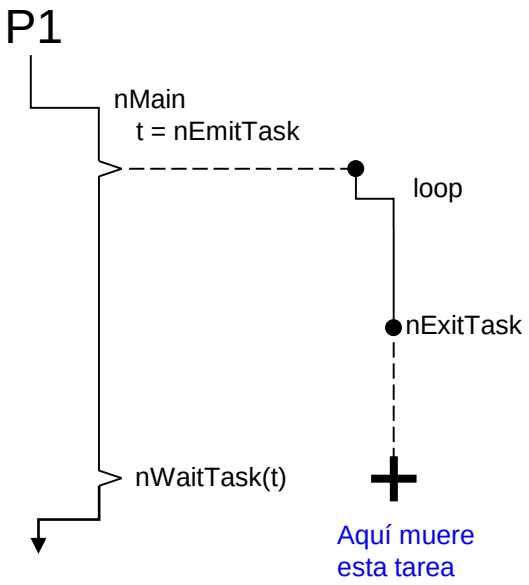
- ***nWaitTask***
  Espera que termine t y libera todos los recursos que t ocupa. Entrega el código de retorno dado a nExitTask.

- Prototipo: *`int nWaitTask(nTask task);`*

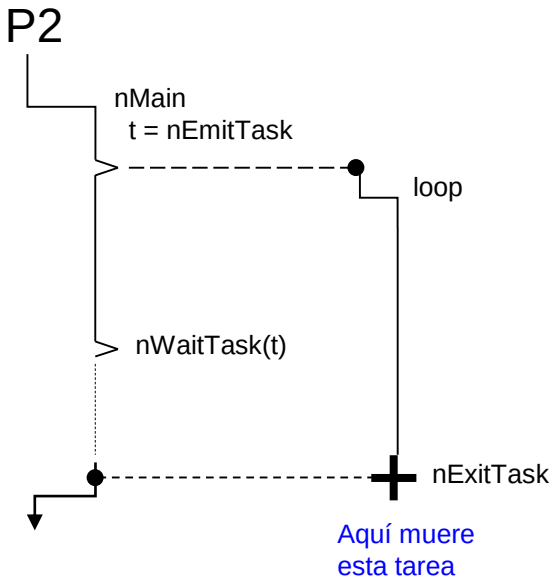
- Ejemplo:
```
int i, rc;
for(i=0; i<=200000; i++);
rc= nWaitTask(t);
...
```

DIAGRAMA DE TAREAS

- Ejemplo: se tienen dos procesos en ejecución, P1 y P2:



Primer caso



Segundo caso

PROGRAMA EJEMPLO EN NSYSTEM

- Ejercicio: Fibonacci

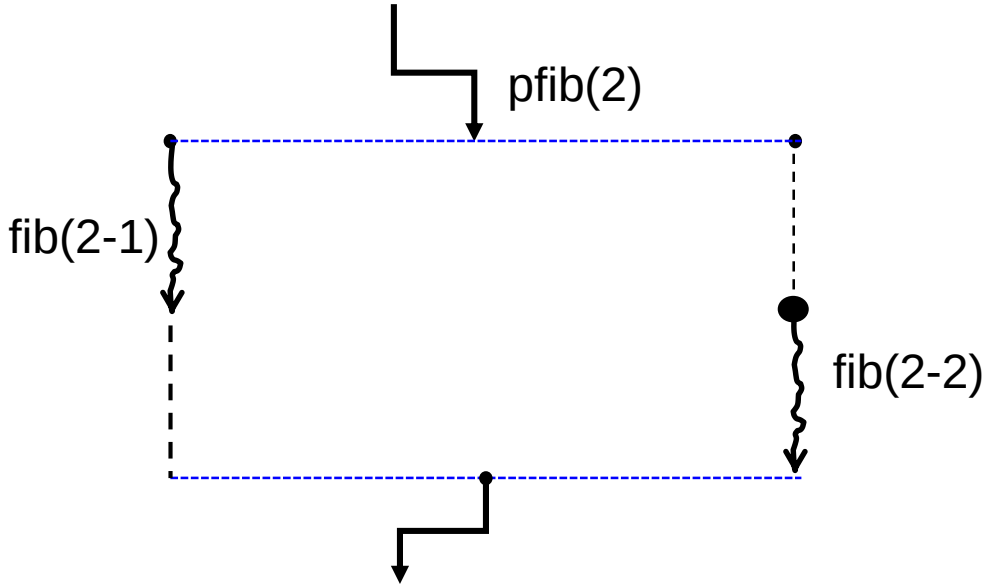
```
int fib(int n) {
  if (n<=1)
     return 1;
  else
     return fib(n-1) + fib(n-2);
}
```

- Solución concurrente 1 para Fibonacci:

```
int pfib(int n) {
  if (n<=1)
     return 1; /*equivalente a nExitTask(1) */
  else {
     nTask t1= nEmitTask(fib, n-1);
     nTask t2= nEmitTask(fib, n-2);
     return nWaitTask(t1) + nWaitTask(t2);
  }
}
```

¿Qué hace realmente este código para pfib(2) ?

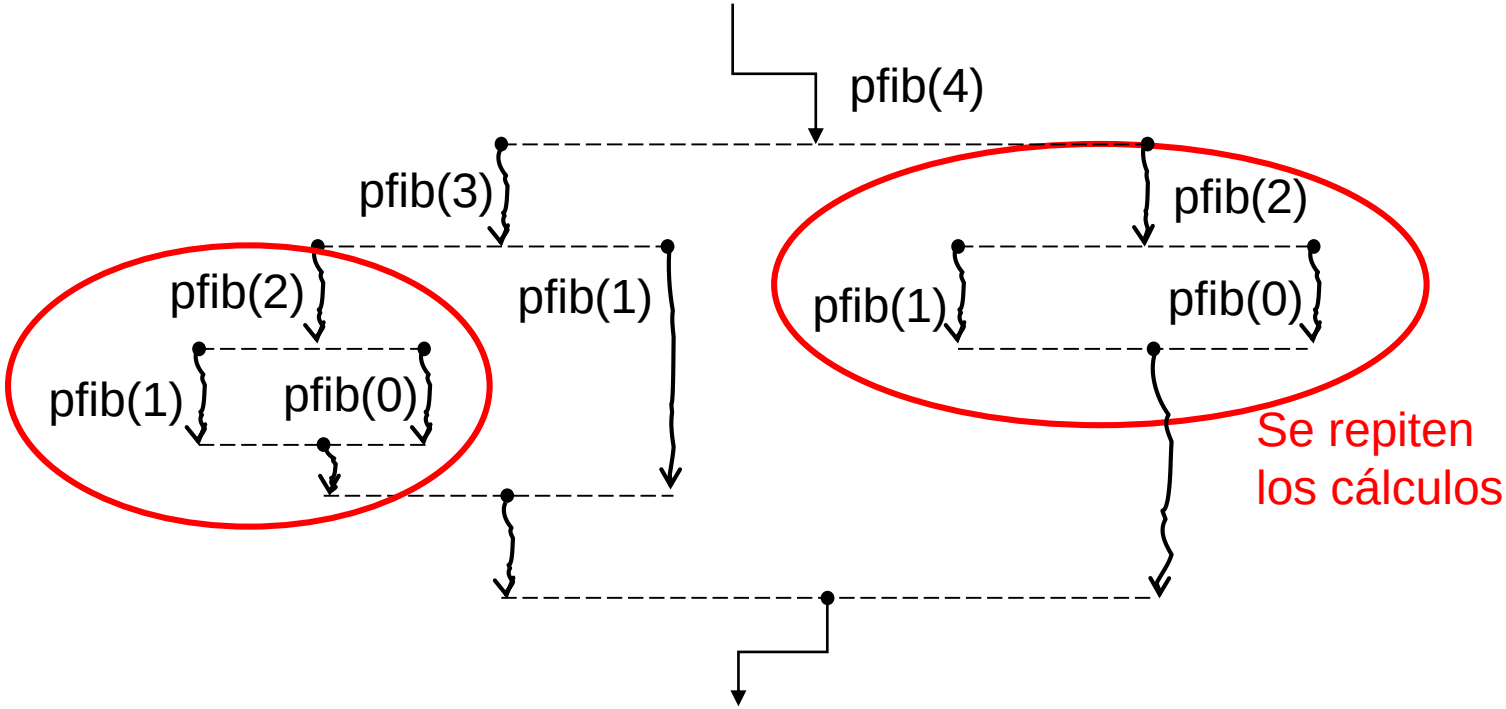
PROGRAMA EJEMPLO EN NSYSTEM



- Ejercicio: Hacer el diagrama de tareas para pfib(4), ahora cambiar fib por pfib en la llamada a nEmitTask anterior.

```
nTask t1= nEmitTask(pfib, n-1);
nTask t2= nEmitTask(pfib, n-2);
```

PROGRAMA EJEMPLO EN NSYSTEM



¿Qué se puede deducir?

En esta solución se crea un n° exponencial de tareas, por lo que es mala. El costo de crear una tarea es más caro que un cálculo de una suma y por ende en total es más caro.

PROGRAMA EJEMPLO EN NSYSTEM

- Solución concurrente 2 para Fibonacci:

```
int pfib(int n) {
  if (n<=1)
     return 1;
  else {
     nTask t1= nEmitTask(pfib, n-1);
     int x= pfib(n-2);
     return nWaitTask(t1) + x;
  }
}
```

Tampoco es buena, el mismo problema anterior: se generan demasiados procesos.

ENTONCES … EL PATRÓN A UTILIZAR ES:

Para evitar este problema el trabajo realizado por una tarea debe tomar más tiempo que el costo de crear una tarea. → Acotar el n° de tareas a crear

```
ConcProcRecursivo(...) {
  if (...)
     SecProcRec(...);
  else {
     nTask t= nEmitTask(...);
     ...
  }
}
```

```
int pfib(int n) {
  if (n<=20)
     return fib(n);
  else {
     nTask t1= nEmitTask(pfib, n-1);
     return nWaitTask(t1);
  }
}
```

PROGRAMA EJEMPLO EN NSYSTEM

- Ejercicio Factorial:

```
int fact(int n) {
  double r= 1.0;
  pmul(1, n, &r);
  return r;
}
```

$$fact(n) = \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot n/2}_{\text{tarea 1}} \cdot \underbrace{(n/2+1) \cdot \ldots \cdot n}_{\text{tarea 2}}$$

```
int pmul(int i, int j, double *res) {
  if ((j-i)<=14) {
     for (k= i; k<= j; k++)
        *res = *res * k;
  }
  else {
     double r1, r2;
     nTask t1= nEmitTask(pmul, i, (i+j)/2, &r1);
     nTask t2= nEmitTask(pmul, ((i+j)/2)+1, j, &r2);
     nWaitTask(t1);
     nWaitTask(t2);
     *res= r1 * r2;
  }
  return 0;
}
```

Si el valor a calcular es menor que un cierto margen (<=14)

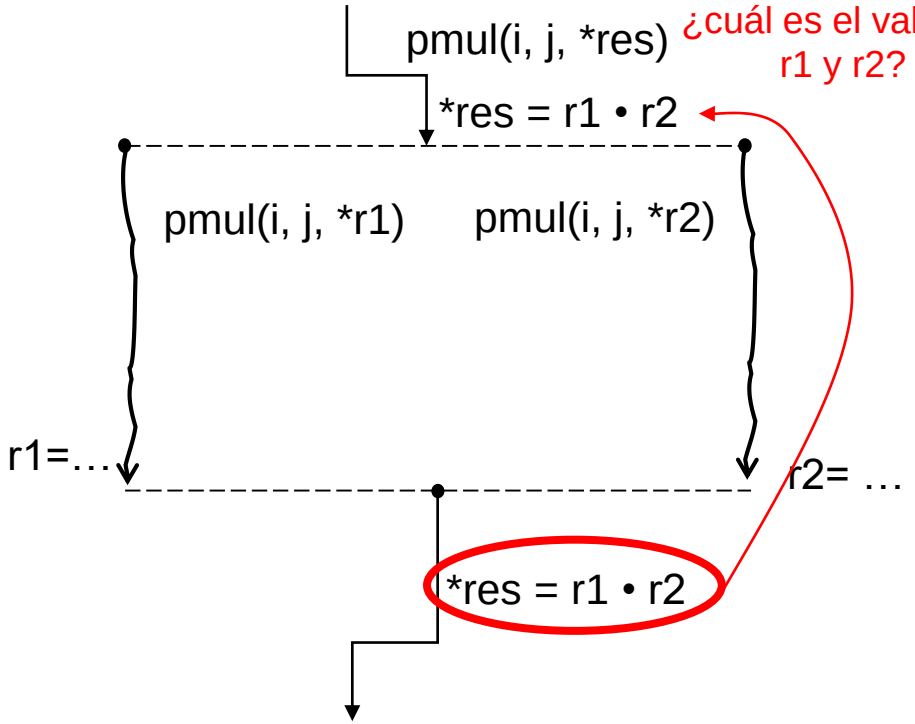
entonces el cálculo se realiza de forma secuencial (for…)

sino de manera concurrente

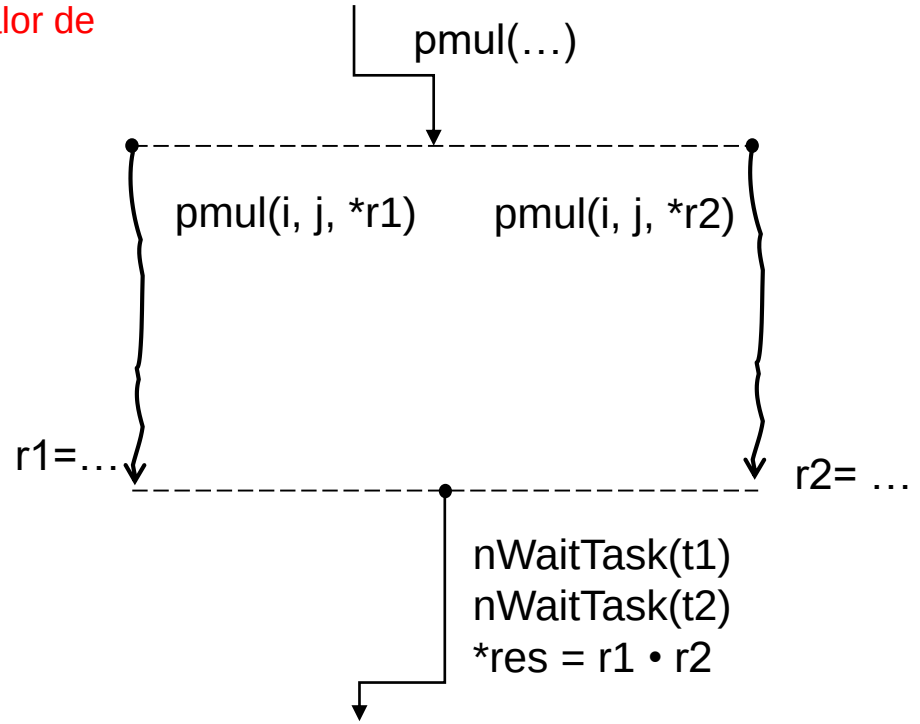
Código en el que hay que tener cuidado!!!

PROBLEMAS DE CONCURRENCIA

- Sin nWaitTask(t) puede ocurrir lo siguiente:



Solución incorrecta



Solución correcta

PROBLEMAS DE CONCURRENCIA

Otra solución para factorial concurrente:

n=16

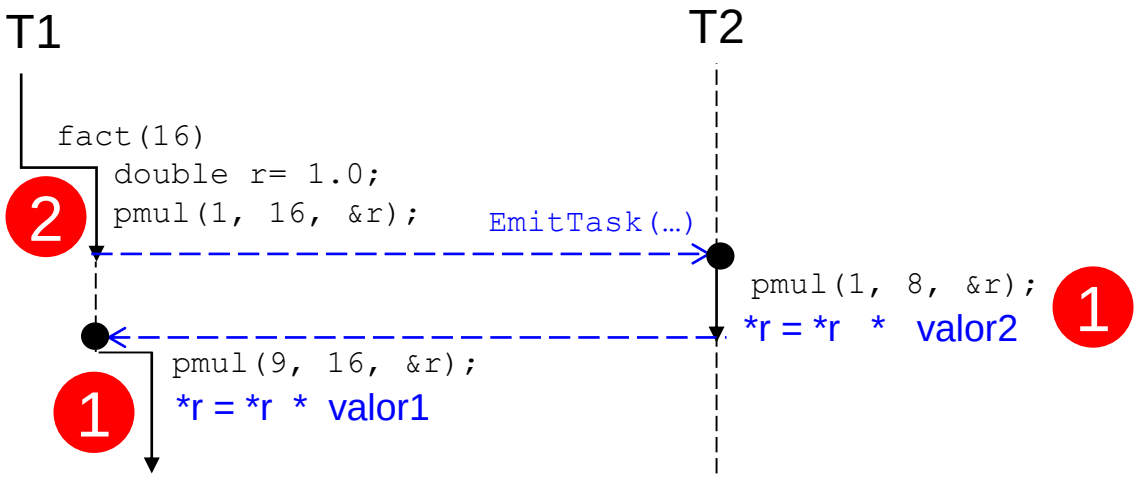
```
int fact(int n) {
  double r= 1.0;
  pmul(1, n, &r);
  return r;
}
```

```
int pmul(int i, int j, double *res) {
  if ((j-i)<=14) {
    for (k= i; k<= j; k++)
      *res= *res * k;
  }
  else {
     nTask t2= nEmitTask(pmul, i, (i+j)/2, &res);
     pmul(((i+j)/2)+1, j, &res);
     nWaitTask(t2);
  }
  return 0;
}
```

caso 1

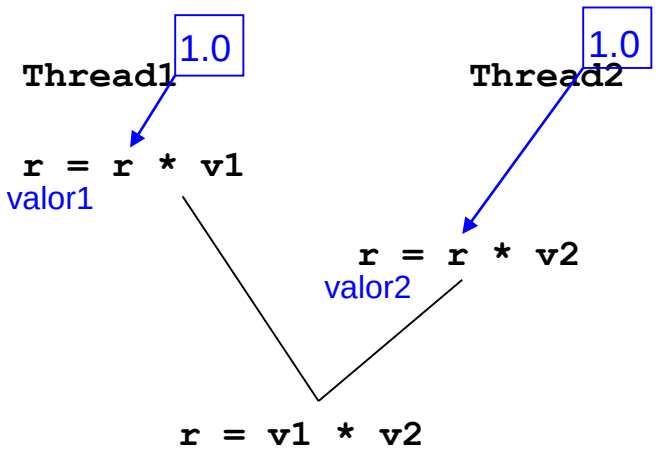
caso 2

Cálculo en secuencial de 1 • 2 • 3 • … • 14

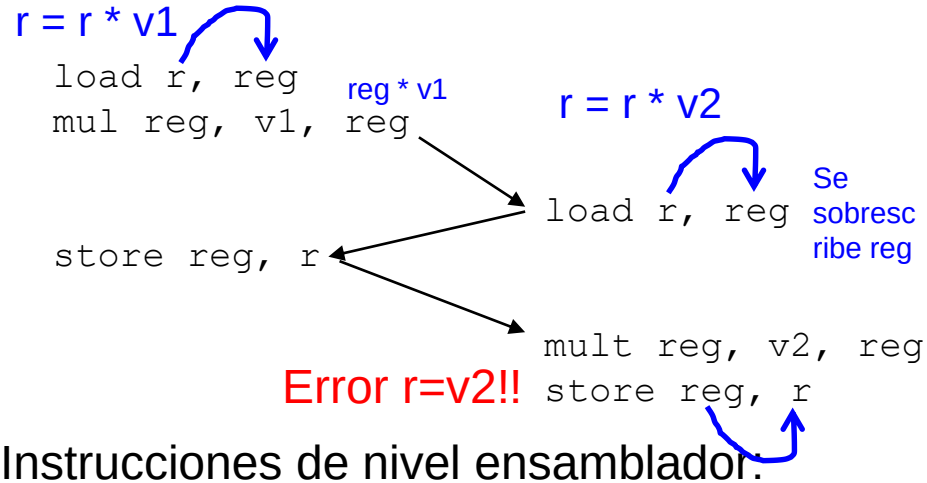


PROBLEMAS DE CONCURRENCIA

Caso safe (seguro)



Caso conflictivo



Instrucciones de nivel ensamblador:

load r, reg	traer el valor de r desde la memoria y dejarlo en reg
mul reg, v1, reg	multiplicar reg por v1 y dejar el resultado en reg
store reg, r	almacenar el valor de reg en la variable r

PROBLEMAS DE CONCURRENCIA

Conclusión:

Para que este código funcione se debe garantizar la **exclusión mútua** de las tareas al momento de ejecutar

```
*res = *res * k
```

De lo contrario se produce un **data race**.

El trozo de código, dónde encontramos la(s) instrucción(es) “delicada(s)” se denomina **sección crítica**.

Solución: Utilizar un mecanismo de sincronización de tareas.