

Nomenclatura útil en UNIX (comandos, banderillas etc)

dpkg -i <<nombre del paquete/archivo>> (Instala paquetes)

su - (hace usuario root *administrador*)

cat (Concatena lista de archivos)

~/ (Retorna a Home)

pwd (print working directory (indica ruta actual))

cd (change directory (autocompletear nombre de ruta con tab))

ls (Listar contenidos de una carpeta)

banderilla -l (large, lista larga)

banderilla -la (muestra todo, incluyendo ocultos)

cd .. (retorna a carpeta padre)

cd - (retorna a la carpeta anterior)

cd (retorna a home)

En explorador de archivos: CTRL+H muestra archivos ocultos

apt (trabaja con repositorios remotos)

update (actualiza base de datos local)

touch (modifica fechas/propiedades de los archivos, sino existe, lo crea)

echo (Lo que reciba por argumentos lo tira en la salida estándar)

echo \$ (\$ significa variable)

\W ruta en la que estoy

\w toda la ruta

al final de la línea en terminal:

\$ (usuario sin privilegios)

(usuario administrador)

rc (archivo de configuración)

ampersan al final significa: corra un comando en segundo plano.

comando&

>> (hace append de un archivo)

GIT

git diff (Indica las diferencias entre dos archivos en git cuando haya un modified)

git rebase (buscar en google porque no sé aún que hace).

el gitignore ignora recursivamente a los archivos de las subcarpetas (desde la raíz a las siguientes)

¿Cómo crear un Tag?

Varias maneras

1. \$ git tag <tag name>

e.g. \$ git tag Tarea01

Asocia el tag al último commit que ha hecho.

2. \$ git tag -a <tag name> -m "Comment"

e.g. \$ git tag -a Tarea01 -m "Entrega para revisión de Tarea01 / Proyecto1.1"

Asocia el tag al último commit que ha hecho y deja un comentario explicativo.

3. \$ git tag <tag name> 3eeaf60a

e.g. \$ git tag Tarea01 3eeaf60a

Asocia el tag al commit con ese identificador (puede usar la versión corta o larga del identificador).

¿Cómo ver del Tag?

Varias maneras

1. \$ git tag

2. \$ git show <tag name>

3. \$ git tag -l <wildcard>

e.g. \$ git tag -l "Tarea*"

¿Cómo hacer push del Tag?

Varias maneras

1. \$ git push <tag name>

e.g. \$ git push Tarea01

2. \$ git push --tags hace push de todos los tags

C

cc nombreArchivo (genera un archivo a.out)

cc nombreArchivo -o nombreEjecutable

banderilla -g (empotra el código fuente, para poder depurar)

banderilla -Wall -W (habilita muchos más warnings)

banderilla -Wall -Wextra (habilita muchos más warnings)

echo \$PATH (variable ambiente PATH (tiene una lista de carpetas separadas por dos puntos))

which cc (da la ruta donde se encuentra el archivo cc.)

-O

banderilla que optimiza según el nivel (1,2,3)

-DNDEBUG

macro que se traga todos los assert (los ignora)

este es el que se le entrega al cliente. (sin -g)

Linter

herramienta de análisis estático de código

primero en existir: para C.

Lint significa pelusa

Un linter se encarga de revisar cosas que el compilador no (como índices fuera de rango)

Ejemplo de linter: `cppcheck`

Análisis estático de código (código fuente).

Código dinámico (ejecutable)

Generan diagnósticos (errores o warnings)

`cpplint` (para el estilo de código) creado por Google, tiene 69 reglas y cada una tiene razonamiento.

Comando útil: `filter` (habilita y deshabilita reglas)

Otro ejemplo: `vera++`

Otro, para Javascript: `ESlint` (261 reglas) es muy pro.

Los linter basados en estilo de código son importantes para mantener una base de código común.

A nivel de proyectos en la industria es muy importante.

En las empresas antes de hacer commit evalúa el código por hooks (revisan el estilo de código).

Además del estilo de código, hay otros linters para probar si es bueno el código.

Makefile:

Archivo con reglas para correr comandos // para construir cosas.

se crea: `touch Makefile`

El destino es lo que se quiere construir

Destino usualmente es el ejecutable

pre-requisitos es de lo que depende el destino. Se actualiza con las recetas.

si no tiene "all":

 si ejecuta solo make, se hará la primer regla

 si no, deberá ejecutar make seguida del nombre de la regla para ejecutarla.

```
hello > Makefile
1 all: hello lint
2
3 hello: hello.c
4 |    cc -g -Wall -Wextra hello.c -o hello
5
6 lint: hello.c
7 |    cpplint --filter=-readability/casting hello.c
```

Instrucciones especiales para make: empiezan por punto y en mayúscula.

.PHONY

hace que la regla que se le indique por prerequisitos no sean de tipo objeto, sino que sean etiquetas que necesitan siempre ejecutar su receta.

Se puede usar la regla clean para borrar los ejecutables.
banderilla -f fuerza la eliminación sin preguntar antes y si no existe no dice nada.

Makefile se utiliza para cualquier lenguaje de programación.

alias: gu git pull rebase
alias gu='git pull -v --rebase'
-v es la banderilla que da un reporte de qué cambió.

Analizador dinámico de código (corre el ejecutable, recibe el ejecutable).

===== indica el número de proceso

Valgrind (programa que corre nuestro programa como si Valgrind fuera un SO), está disponible para UNIX. Es un analizador dinámico que tiene subherramientas.
Lleva contabilidad de lo requerido (tomando datos) y genera un reporte.

HPC = High Performance Computing.

Área de la computación que se enfoca en acelerar las cosas (mejorar el desempeño).

comando:

valgrind ./ejecutable

--

comando extra: time ./ejecutable (dice el tiempo que tarda en ejecutarse el programa).

--

valgrind --tool=nombreDeHerramienta. ./ejecutable

memcheck: herramienta que lleva contabilidad de los accesos a memoria (acceder a memoria, puntero, crear memoria dinámica, fugas de memoria etc).

comando: valgrind --tool=memcheck ./hello

allocs : alojamiento en memoria dinámica

frees: liberación de memoria dinámica

pegar: --leak-check=full al código de ejecución de valgrind para obtener un análisis más grande.

ejemplo: valgrind --tool=memcheck --leak-check=full ./ejecutable

Si le decimos que corra en **quiet** lo único que produce es la salida del programa (si hubiera error igual reporta el error).

valgrind --quiet --tool=memcheck ./ejecutable

el contrario de quiet es -v (nos da mucha más información, sirve para cuando hay un error)

valgrind -v --tool=memcheck ./ejecutable

podríamos agregar otro -v y generaría más información aún.

herramienta helgrind

(su contabilidad tiene que ver con condiciones de carrera o buenas prácticas con concurrencia).

comando: valgrind --tool=helgrind ./hello

archivos en .S son archivos de ensamblador.

Reglas de desarrollo: le interesa el linking (pruebas de memoria etc)

De abajo hacia arriba se lee la terminal de helgrind (es una pila).

La biblioteca estandar de C no tiene nada de concurrencia.

Pero se extiende con POSIX.

Posix modifica funciones como printf para evitar que se contamine con otras.

es decir que no genera condiciones de carrera (las colisiones se evitan). valgrind no detecta eso, porque dice que es una posible condición de carrera (la gente de valgrind lo conoce y debería arreglarlo).

Pruebas de caja negra

Son pruebas para analizar si las salidas son las esperadas.

Cuando se hace testing pensar en casos complicados y comunes.

endoffile: -1 binario

en la terminal se indica con CTRL + D.

con "<" el bash redirecciona la entrada estándar (lee desde lo que le pasemos como si lo introdujeramos en la terminal)

ej: ./ejecutable < carpeta/archivo.txt

al ejecutable se le pasará el contenido del txt.

con ">" el bash redirecciona la salida estándar

el 2> redirecciona el error estándar.

diff archivo archivo

si no genera nada => los archivos son iguales.

Otra forma de hacerlo: si uso less. less es un programa que recibiría la salida que genere mi programa.

ej: ./ejecutable < archivo | less.

less permite recorrer o buscar

también se puede pasar por word cout (wc). wc cuenta cuántas líneas tiene y cuántos caracteres tiene.

también se puede pasar a diff. (diff comparará lo que viene en la entrada estándar con lo que se le envíe).

comando: ./ejecutable < archivo | diff - archivoAComparar.

//quiero que archivo sea ejecutable

comando: chmod +x archivo.extensión

PX:

es un shell script que hace cosas con casos de prueba

los lugares donde ponemos ejecutables son en la variable PATH (en la carpeta bin)

comando: echo \$PATH

comandos para instalarlo:

```
wget http://jeisson.ecci.ucr.ac.cr/tmp/px -O ~/bin/px
```

```
chmod +x ~/bin/px
```

```
export PATH=$PATH:~/bin
```

enlace simbólico/acceso directo con "ln -s archivoOrigen archivoDestino"

archivoDestino es el mismo archivo (es la ruta para llegar a archivoOrigen es como un puntero si se edita uno se edita la otra).

se ejecuta: "px"

Luego utilizamos "px test"

icdiff para ver más bonito el diff (con color)

sudo apt install icdiff

Doxxygen:

sudo apt install doxygen graphviz

graphviz: biblioteca que crea diagramas (en caso de herencia o inclusiones)

crear doxyfile:

doxygen -g -s

-s para hacerlo sin comentarios.

se configura el doxyfile y luego se ejecuta para generar el archivo

se ejecuta con: doxygen

index.html es el archivo que interesa

Buenas prácticas de programación:

indentación

identificadores significativos

programación defensiva: 3 formas

// diseño por contratos: documenta diciendo que el usuario debe hacer tal cosa sí o sí.

// programación defensiva: con assert()

// assert es una macro que siempre tiene efecto, cuando se compila el release, el compilador los ignora (son solo para programadores)

//assert funciona así: recibe un supuesto que tiene que cumplirse, si no se cumple se aborta el programa (es un fallo del programador)

//es buena práctica tener asserts

// usar if(condición) reporta error es otra forma de hacer programación defensiva (esa es para cliente y dev).

No usar constantes mágicas... mejor:

const size_t NOMBRECONSTANTE = valor;

en otras versiones de C más viejas:

#define NOMBRE_CONSTANTE valor

Subdividir subrutinas

Que cada una haga un trabajo en concreto.

Variables inicializadas. (ninguna sin inicializar)

Entre más local la variable, mejor.

Si no se modifica la variable invoque con "const"

Entre más local y constante, mejor.

Taller C++ a C

Computación como ciencia o ingeniería:

Depende del autor y filosofía pero somos más ingeniería que ciencia

1. Resuelve problemas con máquinas (software)
2. Crea constructos computacionales (nuevo conocimiento)
3. Preserva la disciplina. (en la docencia)

Resolver cualquier problema con un constructo: (almacenamiento, procesamiento, comunicación).

Cambio en la forma de hacer las cosas

En C se utiliza el paradigma de programación procedural

Es el paradigma más exitoso

Almacenamiento maneja conceptos (value, variable, indirection, record, array)

Procesamiento (read/write, condition, repeat, call, define)

Comunicación (file)

En C no hay clases u objetos, hay registros y subrutinas.

struct (registro en forma genérica)

_t para decir que es un tipo.

Implementación en .c y especificación en .h

En C no hay referencias

.h es público (atributos y subrutinas)

registro opaco: concepto de C de atributos privados (llevando struct al .c) (debe igualmente estar definido en el .h) es buena práctica hacerlos privados.

Para crear un registro opaco debemos hacerlo en una función que lo cree y lo destruye.

Cuando es init: la interfaz es pública.

Cuando es create: la interfaz es opaca (privada). retorna un puntero.

No hay objetos: no hay operadores sobrecargados.

comparar strings: con strcmp de string.h

retorna entero:

<0 (segundo arg mayor alfabéticamente)

>0 (primer arg mayor alfabéticamente)

==0 (son iguales)

break es válido cuando se utiliza de manera correcta. (cuando no haya que liberarse memoria a posterior).

para usar bool se debe incluir la biblioteca <stdbool.h>

C no tiene plantillas

const a la izquierda del tipo define que la variable no se puede cambiar

const a la derecha del * define que el puntero no va a cambiar de lo que apunta.

El cursor del std siempre va hacia adelante, no puede regresar

Sem1-b

Repasamos los paradigmas y conceptos generales de recursión.

- Procedimental (tenemos a Fortran, C, ensamblador) (completo) (diseño con algoritmos, pseudocódigo y diagramas de flujo)
- Funcional (completo) (diseño con matemática) (es bueno para corrección ya que la permite, va a funcionar todo el tiempo, pero cuesta muchos recursos computacionales, es muy poderoso)
- POO (incompleto) (diseño por UML) (modularizado las bases del código, creando interfaces de usuario)
- Concurrente (incompleto) (es enorme, casi un metaparadigma) (no tiene una forma de diseñar, se utiliza pseudocódigo, UML con marcas de concurrencia y un modelo matemático (Petri)).

Incompleto: agrega conceptos a otro paradigma (depende sobre cual estoy (otro paradigma))

El compilador redunda el código para hacerlo procedural (lenguaje máquina)

Concurrente sobre procedural: problemas pero más control y eficiencia

Concurrente sobre funcional: menos problema, control y eficiencia.

Cuando diseño, lo hago en base de un paradigma.

Paradigma concurrente:

(parallelismo - cambios de contextos)

Serial/secuencial:

```
task1()  
task2()
```

La tarea 2 no inicia hasta que la tarea 1 no haya terminado.

En la vida real: si uno hace algo por partes se puede corromper.

Ejemplo: años escolares en la escuela.

Concurrente:

Todo lo que no es serial/secuencial.

Por lo tanto es un área muy grande.

Ejemplo: llevar 2 cursos en un semestre. (Se intercalan entre un curso y otro).

“No tengo que esperar a que termine totalmente un curso para iniciar el otro.”

Cambio de contextos: es una forma de concurrencia que alterna trozos de ejecución de distintas tareas.

Paralelo:

Es una forma de concurrencia.

Puede tener una parte que no es ni serial ni concurrente.

En el curso se ve la forma en que está completamente en concurrencia.

No intercalo pedacitos de una tarea con pedacitos de otra.

Las dos tareas se efectúan al mismo tiempo.

En hardware: 2 procesadores en simultáneo.

En la vida real: ver una película mientras respiran, comen etc.

Cuando en hardware se pueden hacer 2 cosas al mismo tiempo.

Distribuido

Nomenclatura del eje X (en imagen del profe):

Estado/Recursos/Memoria.

Cosas que tiene un computador: Procesamiento-almacenamiento y comunicación.

Ejecutante: ejecuta un código (hilo, proceso). Puede ser por ejemplo un acelerador gráfico. (depende de del hardware)

Mono-hilo: alguien que ejecuta un código en una CPU. Puede ser serial. Puede ser muy lenta. (ejemplo: diseñando de manera serial)

Paradigma por Eventos: realiza la tarea/el diseño/proyecto pero con escucha (se concentra pero escucha).

Gracias a esto atiende a los eventos.

La tarea1 sigue recibiendo requerimientos

Picadillo.

Gasta recursos porque debe recordar qué hacía cuando cambia de eventos.

Evento: algo que ocurre en cualquier momento.

Call-back: reacción ante el evento. es muy complicada.

async-await: código concurrente que funciona por detrás, es para agilizar el trabajo.

Funciona para desarrollador web.

Con varios ejecutantes:pc

Proyecto grande.

Objetivo de eficiencia muy grande.

Con multi-hilos.

Con recursos compartidos (computadora, escritorio, aire, escucha, pizarra etc).

No tiene intervención de terceros.

Es muy natural.

Imagen: arriba es imperativa abajo es más declarativo.

Pthreads es imperativa: permite controlar los hilos de ejecución. Todo el control de la máquina (muy procedural)

OpenMP: para ciertos casos específicos. Es un poco más funcional (menos procedural).

JavaScript es muy muy muy concurrente.

Hilo: concepto del SO.

Procesador: hardware que permite ejecutar hilos.

Recursos distribuidos:

se contrapone a compartido.

Los ejecutantes ya no comparten los recursos sin intervención de un tercero /intercomunicador.

El tercero será el SO. En conceptos de materia

Metáfora de ejemplo: sucursal en santiago de Chile - Correo - Sucursal en Madrid.

Hardware:

Clúster: distribución simétrica. El hardware será lo más homogéneo posible. Conjunto de máquinas conectadas a la red (muy buena) con las mismas especificaciones. Esto acelera el paralelismo. Son muy caros (millones de \$). Tiene relación con HPC (high performance computing).

Malla: distribución asimétrica: Los recursos no son homogéneos/uniformes.

“el programa tiene que correr por hardware distinto” ejemplo: Zoom.

No tengo tanta facilidad, porque tengo el problema de la heterogeneidad del hardware.

Este mundillo genera mucho dinero y es muy popular (+ usuarios).

Significado de !!!: se pueden serializar los hilos (por medio de un mutex). Es un modelo muy caro, no sale bien, poco eficiente, no es natural. Ejemplo: contrata 5 informáticos y solo uno trabaja. (Modelo AyA).

Diferencia de los recursos distribuidos (procesos) ante los recursos compartidos (hilos/eventos):

En el mundo de recursos compartidos es muy complicado escalar, ya que es muy muy caro.

En el mundo de los recursos distribuidos es más sencillo escalar (es natural), la comunicación es cara.

Caja aceleración por hardware. Distribución heterogénea:

Dispositivos especializados para un proceso.

Objetivo: la optimización.

La optimización está diseñada casi que por artesanía (se debe moldear al problema para darle solución)

Mayores velocidades alcanzadas.

CUDA: Como Pthreads.

OpenACC: independiente, él controla.

Dos segmentos de código (CPU - Acelerador) en un solo programa.

La optimización reduce el ámbito (cantidad de hardware donde puede correr).

Para qué sirve el paradigma concurrente?

1. Incremento del desempeño (paralelismo de datos | HPC)

Duración reducida de un programa.

Se recurren a las formas que hay en la imagen.

2. Separación de asuntos (concurrency de tareas)

Ejemplo de especialización de humanos (informáticos limpiando pisos, leyes etc).

Se ve mal un diseño donde

Ejemplo de reproductor de video (un hilo para interfaz gráfica).

Ocupo que diferentes ejecutantes realicen diferentes tareas y que se comuniquen entre ellos.

Sem2-a

Tarea:

Arreglos más fáciles de paralelizar

No optimizar por ahora (aún así que sea eficiente)

Concentrarse en la corrección.

Estilo empresarial.

casos de prueba son buenos hacerlos antes de diseñar

Ciclo de resolución de problemas

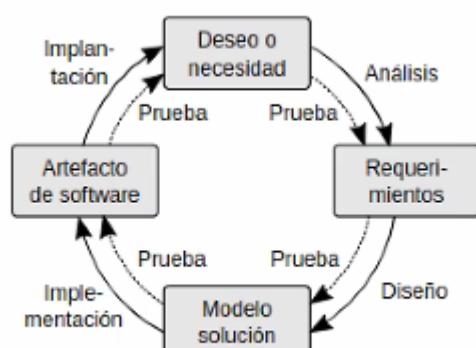


Figura 1. Proceso propuesto de resolución de problemas

Herramientas para programar: paradigma...

Cada uno tiene sus fuerzas y fines.

trucazo con bshrc:

source ~/sjr/bin/bashrc

subirlo a dropbox para tener el mismo en varias compus.

El pseudocódigo debe escribirse según el paradigma (no según el lenguaje).

En el procedural tiene 8 instrucciones

No se puede escribir a la libre el pseudocódigo y no puede ser declarativo ya que el procedural es imperativo.

Objetivo lograble: se le da a una persona que pueda implementarlo.

instrucciones del paradigma: ir a jeisson.ucr.ac.cr -> progra2 video del 5 al 9 y el 12.

Las instrucciones del paradigma son:

Los hilos cuando se crea un ejecutante es como contratar a un empleado.

Ej: restaurante (cargos).

Se necesita separar asuntos.

Tienen tareas.

Diferencia entre procedimiento y función:

Subrutina

Función:

NO tiene efecto colateral
genera lo mismo siempre
Siempre tiene valor de retorno.
Paradigma de programación funcional

Predicado:

Es una pregunta.

Procedimiento:

Sí tiene efecto colateral (cambia estado de la máquina) ej: imprime una variable, mueve mouse...

Vuelve a un estado distinto

Paradigma de programación procedimental

// pido ayuda a otro ejecutante / delegue una orden / sin parentesis: crea otro hilo secundario, le asigna esa tarea.

```
create_thread(greet)  
create_thread(cook)
```

// lo hará en concurrente, dependiendo del hardware incluso lo puede hacer en paralelo.

diseñamos en hello.pseudo (agnóstico al paradigma procedimental)-
traducimos el código a C.

expresiones regulares

regex101: web

comando: man pthread create (manual de posix, viene en el SO).

POSIX: acuerdo de el sistema UNIX que tiene lineamientos que seguir.

Es muy popular.

Linux no es UNIX, la mayoría de HPC (99.9%) es en linux.

forma estándar de UNIX adoptado por otras para crear hilos de ejecución.

A los identificadores de nombres se le ponen guiones bajos. _

El SO es el que tiene contabilidad de los hilos, él crea y destruye los hilos, no son infinitos (infinita memoria etc).

```
void * (*start_routine) (void*)
```

subrutina

es la tarea que quiero que el hilo se ponga a ejecutar.

C es una programa de una pasada: lee el código una sola vez

Las subrutinas tienen que estar declaradas (al menos la firma) antes de ser invocadas.
Si solo se pone la firma, el que los une es el linker después.

ld es el linker. Es una herramienta del SO.

No compilar con -g cuando haremos un release (le damos todo al usuario).

-g deshabilita optimizaciones.

banderilla -l, hace referencia al linker.

si usamos directamente -pthread, hace referencia al preprocesador, al compilador y al linker.

Herramientas para encontrar el origen de un error.

Compilador

Lint (pelusa en español)

revisa errores sintácticos, lógicos, de convención de estilos (diseño/modelo).

da pistas y da luz sobre donde puede haber un problema.

Sem2-b

Sanitizers de Google (son 4 o 5)

-fsanitize=address

Son instrumentalización de código (agrega código), el compilador agrega más código por mí.

Asan reporta: accesos inválidos, fugas de memoria.

El que más se le parece es memcheck de valgrind.

Otro sanitizer:

Memory sanitizer

Detecta memoria no inicializada (ej: usar una variable no inicializada)

Clang es un compilador más moderno. (Universidad de Illinois, Apple, empresas...)

CC es más viejito.

Debe correrse con clang

Requiere mucha memoria para detectar lo que no se ha inicializado. (aumenta el tamaño del ejecutable).

Es mejor inicializar siempre las variables.

Otro:

ThreadSanitizer:

Sí lo soporta cc

Reporta hilos,

Se diferencia con valgrind porque valgrind solo ocupo un ejecutable (no importa el lenguaje)

Los sanitizers ocupan el código fuente para instrumentalizarlo.

Valgrind es más todo terreno.

Existen falsos positivos....

Entre más herramientas, más opiniones y mejor para uno.

undefined behavior:

Me dice si una aplicación se va a comportar de diferente manera en diferentes situaciones.

Intenta oler con heurísticas si el programa correrá mal o distinto en otro dispositivo

Resultado de las herramientas:

Valgrind con memcheck nos da error de memoria

tsan nos dice que tenemos un thread leak (pedir hilo y nunca lo libera).

Ojo: el hilo1 es el secundario, el primero es el hilo0.

Se soluciona con un join (subrutina que se encarga de liberar los recursos).

Join es como un delete.

<unistd.h>
biblioteca estándar de unix
tiene sleep | usleep(1) microsegundos |

la concurrencia genera indeterminismo (no puedo predecir el orden en que se ejecutan las instrucciones concurrentes), puede verse como el azar.

Nunca use microSleep / sleep en el código del cliente.

No correr código instrumentalizado con valgrind (sino genera muchos errores).

Concepto de hilo de ejecución:

Hacer rastreo de memoria (generar evidencia de lo que pasa en el programa) procesamiento, memoria y comunicación. Esto nos hará entender mejor.

El compilador traduce (ej: de c a código ejecutable).

Cuando se ejecutan pasan demasiadas cosas.

El SO genera un proceso, sabiendo/asignando recursos necesita esa unidad/ejecutable.

Secciones de memoria (antes llamados segmentos)

El segmento es más limitado, por eso se cambió el nombre.

Hay al menos 4 segmentos que el SO carga para un ejecutable:

1) Segmento de código: instrucciones del programa se cargan en la memoria principal.

Todo byte tiene una dirección de memoria.

Subrutinas: arreglo de instrucciones consecutivas y pueden ejecutarse.

2) Segmento de datos: también viene del ejecutable. Están las variables alojadas de manera estática. (static provoca que la variable se cree en el segmento de datos y no en la ejecución de la subrutina). Hay globales y locales. Todas las variables en el segmento de datos DEBEN estar inicializadas (sino el compilador las inicializa en 0).

En el mundo concurrente las variables globales son NOCIVAS. no se pueden declarar variables globales porque produce muchos errores.

Tener cuidado con las bibliotecas porque esta puede incluir variables globales. (tiene que saberlo).

Cuando una subrutina usa variables globales le diremos: reentrante.

La documentación de la librería debe decir si es threadsafe o no.

Ejemplo de variable global de una biblioteca: stderr de stdio.

Las variables globales siempre existen, antes y después que la ejecución de las subrutinas

3) El SO crea un hilo de ejecución para ponerlo a ejecutar cosas. (el main de un programa por ejemplo). en C++: runtime library, en C: crt0, (el lenguaje crea abstracciones sobre hardware).

La de C no es tan grande, la de C++ sí. (polimorfismo, try catch...).

La biblioteca runtime son lo que ocupan los programas hechos en ese lenguaje para ejecutarse y tener info sobre el hardware, viene en el SO.

En linux la biblioteca viene ya instalada porque el compilador y C están relacionados demasiado de linux. C es un lenguaje súper importante.

El hilo principal:

100 significa que inicie en esa línea del código de segmento.

StackFrame: lugar de una memoria donde coloco los valores de las variables locales.

primer espacio: para parámetros.

demás espacios: para variables.

El ámbito de la variable local es pequeño.

Las constantes (const) no importan si son globales o locales (el ámbito es super pequeño).

Variables lo más locales posibles.

Para C una subrutina es una dirección de memoria (un puntero) ej: greet.

Crear un hilo secundario es el mismo procedimiento que se hizo para crear el principal (es lo mismo). Un SO tiene un scheduler que se encarga de los hilos de ejecución.

Ocupa la subrutina que tiene que invocar (una dirección de memoria). ProgramCounter contiene la línea de memoria en donde empieza. Además ocupa una pila (estructura de datos).

Cuando el hilo es creado el scheduler le dice al SO que está listo (creado el nodo lo pone en la cola) cuando es el primero del heap, será llamado a ejecutarse.

Condición de carrera: dos hilos luchando por la misma variable (mutable) y genera contaminación.

palabra mágica: espera. (en el mundo de la concurrencia).

El SO tiene una cantidad de colas de esperas enormes.

Tres formas de acabar/terminar un hilo: return, }, pthread exit (no recomendada). el nombre join.

Hilo de ejecución:

Se puede definir como un arreglo de valores que el SO carga en los registros de una CPU y permiten ejecutar código en ella.

metáfora: hilo de pensamiento, discurso. Es diferente al otro. Tiene que ver con una tarea.

No es hilo de tejido, sino de continuidad de discurso por ejemplo.

Pregunta: qué pasa con los demás hilos cuando un hilo tiene un error?

Sem3-a

cp para copiar

cp -a para copiar recursivamente

mv para renombrar

git mv para renombrar y que git no note un cambio

makefile tiene variable

La forma de usar una variable es un dolar. (\$).

Se puede escribir entre llaves \${NOMBRE}.

Letra mayúscula en variables por convención. (funciones en minúscula)

El ejecutable se llamará igual que la carpeta que lo contiene

pwd: print working directory, da la ruta donde me encuentro

basename: obtiene el nombre de la carpeta actual

back ticks: reciben un comando cualquiera, captura la salida y ejecuta lo de dentro.

no se pueden anidar: entonces usamos \$(nombre).

basename `nombre`

o

basename \$(nombre)

make tiene funciones

shell hace que tome el comando que sigue y lo interprete en/de la terminal.

duda: cambiar nombre de carpeta rápido.

Make tiene otras variables que siempre define (variables automáticas).

Son bastante poderosas y versátiles (lo malo es que es intelectable).

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

usaremos:

\$@ | Lo que nos indica es el target

\$< | Indica el nombre del prerequisito.

CFLAGS: banderas que le pasamos al compilador de C

argc | indica argument count

argv | indica argument vector

Utilizaremos := como símbolo de asignación para evitar = en el pseudocódigo

En C:

los arreglos están muy relacionados a la máquina

Los argumentos, para que acepte tab o espacios en blanco utilizamos comillas simples ' texto' o comillas dobles. (serán tomados como parte del argumento). La comilla simple no interpola variables (es literal). La comilla doble sí interpola variables (si le pone una variable, dará su valor).

Los bash son muy útiles, tienen ciclos, variables y demás funciones.

comentar con pre-procesador:

#if 0

#endif

strtol(variable, ...)

casting de str a long

thread_number es conocido como rank. (varios hilos "enumerados/rankeados")

No se debe entregar / terminar un código con warnings.

Depurador

gdb ./ejecutable

r | run

bt | back trace (dice que hay en la pila)

Tener cuidado con el inicio del arreglo, no asumir que envían más de un argumento.

El argumento 0 es el nombre del programa

El ultimo argumento es el argument counter.

Los errores en código generan que baje el prestigio del programador, además de crear espacios o vulnerabilidades del programa (pueden atacarlo).

los assert son errores para el equipo de desarrollo (no para clientes). ej: cuando ejecutamos una subrutina y no le damos un valor necesario.

para compilar en modo release (sin código fuente, para cliente):

-O | optimice

-DNDEBUG | No debuguee

El segmento de pila tiene límites

en linux: 8megas

Cuando invado memoria de otros procesos el SO da la advertencia (mientras esté en mi stack segment no indica error solo que se puede ensuciar lo que esté allí.)

arreglos automáticos de tamaño variable (variable length arrays):

en C. No debe estar en la pila porque es dependiente del usuario.

size_t : es un typedef que va a ajustarse a la cantidad de bits de la arquitectura. (sin signo). Se utiliza mucho con las direcciones de memoria.

Heap segment (memoria dinámica):

Se administra con malloc, calloc y realloc

malloc: se le pasa por parámetro el tamaño de bytes que ocupa y retorna un void* (una dirección de memoria). Le puedo hacer casting.

Programación defensiva es super importante.

Por cada malloc/calloc debe haber free. (liberar memoria).

Es importante liberar memoria apenas se deje de utilizar

La concurrencia es inherentemente indeterminista:

(nunca se sabe su orden). No debería de ser serial.

Un join implica espera.

Es importante entender cuando un programa es indeterminado o serial.

Es peor un programa concurrente serializado que un serial.

Determinismo no es que se imprima en orden o no, se detecta cuando no sé quien ocupa cpu primero (no sé quién lo hace antes que otros).

El programa puede tener cálculos concurrentes e indeterministas pero la impresión puede ser necesariamente serializada.

Condiciones de carrera:

Con Valgrind/helgrind pueden haber falsos positivos con impresiones. (problemas que no son realmente problemas).

problema con long: depende de la arquitectura y no sé a qué va a cambiar (diferencia con size_t).

Encabezados definidos:

stdint

inttypes

inttypes define macros.

reinterpret_cast no está disponible en c

Cuántos cpus hay disponible en el sistema:

en unix:

sysconfig

incluir <unistd.h> (biblioteca estándar de unix).

Sem3-b

Es buena práctica crear códigos de retorno para los errores. (Es mejor si se definen como constantes)

Si corto una línea se ve bien dejar un operador en la siguiente para que quede más claro que la está cortando.

El valor de retorno se atrapa con \$? (es una variable automática).

echo \$?

Puedo escribir scripts en la línea de comandos:

./ejecutable && echo OK

No importa la convención de estilo de programación, lo que interesa es ser consistentes.

La carpeta build guarda archivos temporales y binarios salidos del código fuente

opción -c:

le indica al compilador que se detenga en el momento en que genera el .o (no termina de generar el ejecutable).

sdt=gnu11

solo voy a utilizar lo que es estandar gnu

queremos habilitar las extensiones que gnu, clang y cc tienen hasta el 2011

-I

carpetas donde el preprocesador va a buscar los .h

ejemplo: -Icarpeta

-I. // es la carpeta actual

-MMD:

optimización: compila el .c , genere el .o y otro .d que demuestra dependencias (de qué .c depende un .h). Sirve para que el make solo compile de los que dependía un archivo que esté desactualizado.

-Im:

biblioteca l(library) m (math)

LIBS lee solo en linker

FLAGS lee para los tres: precompilador, linker

Cambiamos makefile para incluir pthreads en FLAGS

Dos propósitos:

conurrencia de tareas(separación de asuntos: crear hilos y dar a cada uno una tarea distinta.) /un hilo por tarea. Lo hacen de forma concurrente, no le sacan el jugo a todos los procesadores.

parallelismo de datos (incremento del desempeño un for creando una n cantidad de hilos que hacen la misma tarea). /

Importante: la subrutina que crea el hilo, lo destruye. (contrato y despido).

Ejemplo 3: memoria privada

donde cada hilo imprime "hola" con el numero de hilos creado hasta el momento.

Concepto: familia/equipo de hilos.

thread team: varios hilos trabajando en lo mismo (tarea dada desde un for), que saben su numero y cuantos hay creados hasta ese momento. (parallelismo de datos)

Las variables globales frenan la concurrencia (es no reentrante), complica las cosas. No escala, no se puede trabajar en equipo.

Una clase de C++ se parece a un struct de C. (es un registro)

arreglo: región continua de memoria que almacena valores del mismo tipo.

El registro es una región continua de la memoria que puede alojar campos de distinto tipo. (en inglés se denota como record(de memoria) o register(de cpu))

Rust es como un C que restringe su uso.

Registro: tres tipos

Estructura:

Union: almacena el tipo más grande.

Clases:

azúcar sintáctico: notación más compacta que no provee ninguna característica nueva.

Las clases las descompone el compilador en registros y subrutinas.

rank es igual a thread_number.

calloc(): recibe la cantidad de elementos y el tamaño de uno de los elementos. Llama a malloc y luego inicializa la memoria en 0s.

calloc es más lento que malloc porque inicializa. // new []() en c++

malloc recibe la cantidad de bytes que quiero reservar, mantiene la basura ahí. (no inicializa). // new[] en c++

struct hay dos formas de trabajarlos, como c++ y como c.

La memoria debo borrarla al orden inverso de la creación.

creo A.

creo B.

borro B.

borro A.

Podemos utilizar aritmética de punteros o aritmética de indexación. Es lo mismo.

Los hilos tienen memoria privada (obligatoriamente), propia.

bash: si pongo A && B : "si A está bien ejecute B"

bash: si pongo A && B : "ejecute A y ejecute B"

ejercicio 4 de tarea

Memoria compartida entre los hilos.

se comparte un dato, no se duplica.

Se utiliza mucho a nivel de hilos.

ejercicio 5:

En los datos compartidos tengo el dato que tiene todo el equipo no es un duplicado ni una copia.

declarativo significa lo que quiero.

podemos compartir una variable con todos los hilos que se creen.

igualmente utilizamos un struct para decir qué cosas compartiremos
desde el struct privado ponemos un puntero hacia el dato compartido de todos.

Sem4-a

Quiz1 temas:

- Paradigmas de programación concurrente y distribuida
- Hardware y software concurrente
- Concurrency de tareas y paralelismo de datos
- Concurrency imperativa con POSIX Threads
- Concepto de hilo de ejecución
- Rastreo de memoria

tiempo de pared:

tiempo real

tiempo de cpu:

cantidad de tiempo que estuvo mi programa consumiendo cpu

porque estuvo en el cpu (user)

o pidió datos al SO (sys)

diferencia entre tiempo de pared y cpu

el de cpu se multiplica la cantidad de tiempo por la cantidad de hilos.

El programa time sirve para pruebas pequeñas, él mide la ejecución de todo el programa e incluso de otro código de terceros, la carga del IDE, del lenguaje etc... Por eso para el curso o cosas científicas no funciona, a veces se quiere medir solo una parte pequeña.

Medición precisa con subrutina `clock_gettime()`, tiene resolución de nanosegundos.

La resolución es la unidad más pequeña que puede dar un reloj.

Debian no es SO de tiempo real => no funciona `clock_realtime`.

Hay sistemas operativos de tiempo real que tienen más restricciones para mantener todo funcionando en todo momento. (paciente)

reloj monotonico:

comienza desde un punto y el resto es un incremento respecto a ese punto de referencia.

`CLOCK_MONOTONIC`

`struct timespec start_time`

`clock_gettime`

%.⁹ presición de 9 decimales

`%f` flotante

`%ld` flotante doble precisión.

punto fijo y punto flotante

aritmetica de precision fija: son mucho más rápidas que las de punto flotante.

aritmetica de precision flotante:

concepto "promueve": ve un casting y pasa el otro a ese tipo para realizar la operación. ej:
`double(num) / num2.` como num2 tiene cast entonces el compilador también ve, i.e:
promueve a num2 como double.

la división(/) es muy cara a nivel de flotantes

(`* 1e-9 == (/ 1000000000.0)`). El de la izquierda es mejor.

$$\sigma^2 = \alpha\beta^2$$

Ejemplo6: imprimir concurrentemente en orden.

con una espera activa.

un while que espera a un next_thread para comparar si ya tocó el turno según el thread number

Hay que tener cuidado con la espera activa (busy-wait). Puede ocupar todos los núcleos.

SO: yo le voy a dar a cada hilo (lo cargo en la cpu) un time_slice de 50microsegundos. A los 50microsegundos saca al que ya se le acabó el time_slice y mete a otro, por democracia. Esto se verá más a profundo en el curso de SO.

La CPU es un recurso compartido caro.

Metáfora: baño de la casa, voy a esperar en el baño aunque no tenga ganas.

Eso no es eficiente ni buena manera de hacerlo.

La espera activa es muy severa (maliciosa/nocivo), no es ético a nivel de programación. Tiene consecuencias terribles.

La espera es el mecanismo fundamental del control de la concurrencia.

La espera buena es la pasiva/inactiva! no la activa.

Una espera pasiva NO se da en el CPU. Le pedimos al SO que me tome y me ponga en la cola de espera (no la del CPU).

No existe un detectador de espera activa.

En el ejemplo 6 no se soluciona lo de la espera activa aún.

Ejemplo 8

Se quiere modificar la espera activa para que el hilo llegué e informe en la salida estándar en qué posición llegó. (hay determinismo en la carrera de hilos pero no con el orden de hilos).

en pseudocódigo el "my o local" se utilizan mucho. Lo que quiere decir es que es privado y/o local.

Condición de carrera definición: (race_condition / data race):

Modificación concurrente de memoria compartida

Para que se dé una condición de carrera tiene que haber (las 3):

1. Modificación de memoria, es decir: variable y operador de asignación
2. Concurrencia ie: está dentro de una subrutina/instrucción concurrente (dos hilos pueden entrar).
3. Memoria compartida: dos hilos lo pueden modificar ya que no es memoria privada.

si solo es lectura no cumple la 1. (modificación)

si uno lee y otro modifica sí cumple la 1. El proceso se corrompe.

Hay veces donde no es fácil detectar una condición de carrera. Un ejemplo pueden ser dos hilos que intentan modificar un atributo de una clase.

La condición de carrera es un enemigo en el curso y no es ético a nivel profesional. Las herramientas (tsan) no siempre detecta el error.

La condición de carreras se podrían evitar quitando una de las 3: serializando por ejemplo. Para eso se requieren mecanismos de control de concurrencia.

Mecanismo: mutex (exclusión mutua): solo un hilo ejecuta las instrucciones protegidas con el mutex en la región crítica.

Se verá la concurrencia con la metáfora de vías de carretera(cpus) y carros (hilos).

por un puente angosto entra el carro(hilo/ownership) que va a ejecutar la instrucción que está en la región crítica. El resto de hilos NO ejecutan dicha opción.

Cada hilo tiene su cola de espera en el Sistema Operativo.

Las esperas malas son la activa (nociva) y los sleep (tonta porque no sabe cuando va a suceder).

Las demás esperas son buenas.

Crear mutex: es como crear un hilo.

create_mutex() // le pido al SO que me cree un mutex inicializado en verdadero.

Funciona como un booleano (puedo entrar/verdadero? sí o no).

se deben usar buenos identificadores para los mutex.

usando una convención:

shared can_access_position = create_mutex()

el mutex casi siempre es compartido para que los hilos puedan esperar por el mismo "puente angosto". El mutex no necesita ser protegido por nadie. EL SO lo ejecuta con instrucciones de procesadores para hacer exclusión mutua (a nivel de hardware).

wait (lock) y signal (unlock)

Mutex no asegura orden, solo exclusión mutua.

Sem4-b

Los mutex serializan el código.

Entre más pequeña sea la región serializada, mejor.

Se debe tener cuidado ya que los mutex crean colas de espera y eso requiere memoria, si son muchos mutex se puede quedar sin campo y cae el programa.

lock (pone mutex en false)

instrucción

unlock (pone mutex en true)

Mientras hace lock, pone los hilos que no tienen permiso en una cola de espera en la memoria (espera pasiva/inactiva).

Se pueden poner corchetes entre el unlock y el lock para indicar que está dentro de un mutex.

```
lock
{
instrucciones
}
unlock
```

Tener en cuenta de que cuando el unlock se ejecuta, el código de abajo sigue siendo concurrente (no asegura orden de impresión por ejemplo).

Un trabajo pesado serializado junto a un print se ve mal.

Si inicializo un mutex debo destruirlo por seguridad de la memoria en el SO aunque las herramientas no marquen error.

Tener en cuenta que un hilo posterior puede cambiar un valor dentro de un mutex mientras que otro (anterior) está imprimiendo ese valor también. Aquí se da un caso de condición de carrera ya que T1 imprimirá el valor COMPARTIDO que T2 está modificando paralelamente.

Se puede solucionar usando memoria privada (una variable con la copia de la variable compartida).

```
125 // lock(can_access_position)
126 pthread_mutex_lock(&shared_data->can_access_position);
127 // race condition/data race/condición de carrera:
128 // modificación concurrente de memoria compartida
129 // position := position + 1
130 ++shared_data->position;
131 // my_position := position
132 uint64_t my_position = shared_data->position;
133 // unlock(can_access_position)
134 pthread_mutex_unlock(&shared_data->can_access_position);
135
136 // print "Hello from secondary thread"
137 printf("Thread %" PRIu64 "/" PRIu64 ": I arrived at position %" PRIu64 "\n"
138     , private_data->thread_number, shared_data->thread_count, my_position);
139
140 return NULL;
141 } // end procedure
142
```

Otro error: (2 regiones críticas usando el mismo mutex): no hay indeterminismo pero implica que un hilo podría ejecutar después de lo que se quería (un hilo posterior editó su dato)

```

● 121 assert(data);
● 122 private_data_t* private_data = (private_data_t*) data;
● 123 shared_data_t* shared_data = private_data->shared_data;
124
125 // lock(can_access_position)
126 pthread_mutex_lock(&shared_data->can_access_position);
{
128     // race condition/data race/condición de carrera:
129     // modificación concurrente de memoria compartida
130     // position := position + 1
131     ++shared_data->position;
132     // unlock(can_access_position)
133 }
134 pthread_mutex_unlock(&shared_data->can_access_position);
135
136 pthread_mutex_lock(&shared_data->can_access_position);
{
138     // print "Hello from secondary thread"
139     printf("Thread %" PRIu64 "/" PRIu64 ": I arrived at position %" PRIu64 "\n"
140         , private_data->thread_number, shared_data->thread_count, shared_data->position);
141 }
142 pthread_mutex_unlock(&shared_data->can_access_position);
143
● 144 return NULL;
145 } // end procedure
146

```

Annotations in red:

- Line 125: A red circle with a question mark is above the code.
- Line 132: A red arrow labeled '1' points to the unlock operation.
- Line 136: A red arrow labeled '2' points to the second mutex lock.
- Line 139: A red arrow labeled '3' points to the printf statement.
- Line 140: A red arrow labeled '3' points to the printf statement.
- Line 142: A red arrow labeled '3' points to the third mutex unlock.

Se soluciona creando otro mutex (usando 2 distintos)



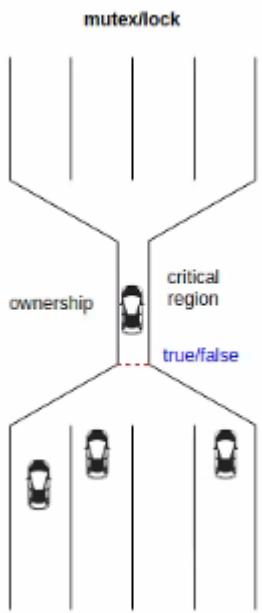
Puede haber un carro en mutex1 y otro en mutex2.

Entre más pequeño (menos tiempo) es mejor para la región crítica.

Tener cuidado con la colas de esperas (hacer unlock de los lock para que los hilos puedan seguir corriendo) (bello durmiente: ningún hilo ocupa CPU ya que está en colas de espera). No es un bloqueo mútuo (uno espera recursos del otro), ya que todos los hilos están esperando por otro hilo que ya terminó. Podría llamarse “espera indefinida” porque está indefinidamente esperando.

Este problema ocurre cuando un proceso se ocupa de un recurso y nunca lo suelta para que otro lo ocupe.

Metáfora del mutex:



Anuncios:

Varios lunes sin clases => lunes 13 sí hay clases (2 grupos) de 9 a 12am

Los quices serán los miércoles de 11am a 11:30am. Repo a la 1

Ejemplo 09:

Concepto de semáforos:

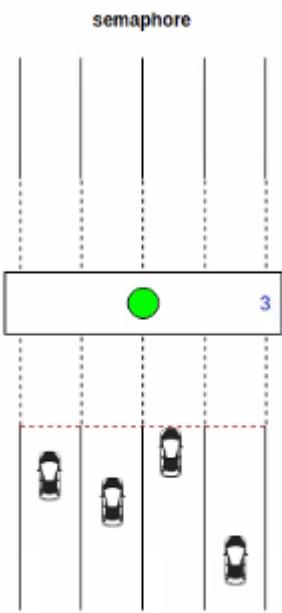
“generalización de mutex”

Dijkstra: un semáforo es un entero con signo con tres características:

1. Cuando se crea, se inicializa como un entero cualquiera. Luego solo se puede incrementar (signal) o decrementar(wait) en 1. No se puede leer el valor actual del semáforo (no hay get).
2. Cuando un hilo decrementa un semáforo, si el resultado es negativo (cuantos están esperando), el hilo es bloqueado y no puede continuar hasta que otro hilo incremente el semáforo.
- 3.

trylock va a funcionar entre un semáforo (trywait) y un mutex. Incumplen la distribución de Dijkstra, genera casos extraños a nivel de programación aunque lo facilitan.

Metáfora del semáforo:



semaphore

A diferencia del mutex, el semáforo permite concurrencia ya que no genera exclusión mutua. Además, el semáforo no tiene un ownership (pertenencia). Todos los hilos pueden incrementar el semáforo.

Los mutex de linux no tiene ownership. (pequeó dato)

Imprimir en orden sin espera activa requiriendo a n colas:
creando n semáforos para cada hilo.

Sem5-a (lunes 13 de setiembre)

Ejemplo 9. Hola mundo ordenado con semáforos.

El mutex implica serialización, el semáforo no siempre (depende de su número de serialización).

En Unix, los hilos tienen dos partes: hilo de usuario e hilos del SO. depende del SO la forma en que se implemente.

concepto futex (fast userspace mutex):

En linux se puede utilizar como un futex (otro carro puede prender o apagar el semáforo)
En el curso lo veremos solo como mutex(el ownership es el único que lo puede cambiar).

Cuatro formas al menos, de declarar un semáforo.

```

4 | declare sems as array[0:4] of semaphore
5 | declare sems as an array of 4 semaphores initialized in 3
6 | declare sems as semaphore[4](3)
7 | sems := semaphore[4](3)

```

Para los nombres de los semáforos pueden haber varios nombres

Para el arreglo, usaremos como si fuera un booleano que responde: verdadero o falso.
“can_greet”

```
5 | | can_greet[thread_number] := create_semaphore(not thread_number)
```

inicializa el index 0 en 1 y los demás en 0.

not 0 == 1 true.

not >0 == 0 false

tr	!tn
0	1
1	0
2	0
3	0
-4	0

wait decrementa los hilos.

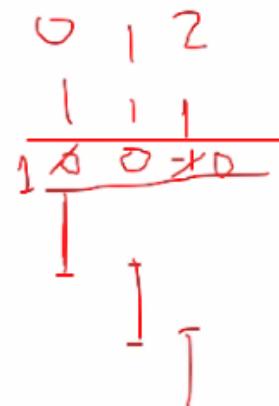
signal incrementa el hilo que envío.

Si el resultado del wait queda en 0 o positivo, puede continuar ejecutando código

Lógica modular (línea 14):

```
hello_order_semaphore.pseudo M • Untitled-1 •
design > hello_order_semaphore.pseudo
1 procedure main(argc, argv[])
2     shared thread_count := integer(argv[1])
3     shared can_greet as array of thread_count semaphores
4     for thread_number := 0 to thread_count do
5         can_greet[thread_number] := create_semaphore(not thread_number)
6         create_thread(greet, thread_number) // thread team
7     end for
8     print "Hello from main thread"
9 end procedure
10
11 procedure greet(thread_number)
12     wait(can_greet[thread_number])
13     print "Hello from secondary thread", thread_number, " of ", thread_count
14     signal(can_greet[(thread_number + 1) mod thread_count])
15 end procedure
```

2 11
3 x 3



Nos deja la situación inicial (1 0 0).

Usando el operador modulo como: valor % cantidad_de_valores.

Así cuando valor % valor2 sea 0, se devuelve al index 0 sin generar acceso a memoria no deseado.

Implementación del ejercicio 9:

```
#include <semaphore.h>
```

hay dos tipos de semáforo: de memoria y de disco (nombrados).

Los de memoria son como el mutex, tiene funciones como init, destroy.

De disco (nombrados). están las funciones como open, close y unlink (para destruir el archivo). post: es un incremento (signal). wait. trywait (trato de incrementar el semáforo y si no puede retorna un código error). timewait (espera a que el semáforo se ponga en verde)

pero si después de un tiempo (en parámetro) no se ha puesto en verde, retorna código error).

El semáforo es mucho más versátil que los mutex porque permite manejar archivos con memoria compartida del proceso.

En linux trabajaremos con semáforos de memoria.

en mac no se utilizan los de memoria porque están deprecated, se usan los de disco.

tipo de dato del semáforo: sem_t

```
13 // thread_shared_data_t
14 typedef struct shared_data {
15 | sem_t* can_greet;           declara semáforo en memoria dinámica porque no se
saben la cantidad de elementos del semáforo en tiempo de compilación sino en tiempo de
ejecución.
```

Para inicializar punteros en C, se utiliza NULL.

int pshared: parámetro que recibe el init del semáforo. Si mando 0, el semáforo NO se comparte con otros procesos, si se envía algo distinto a 0, se compartiría con otros procesos. Para el ejemplo 9 lo pondremos en 0 (no se comparte).

No se tiene la posibilidad de crear tantos hilos, hay un límite, la creación de hilos tiene un límite.

Se debe tener cuidado, si tiene muchos mecanismos de control de concurrencia, quizás no se tiene el mejor diseño de concurrencia. Se debe tener cuidado con los recursos de la máquina.

Ejemplo 10. Hola mundo ordenado (seguridad condicional)

Muy útil cuando se quiere hacer un trabajo concurrente controlando el timing de ejecución, producir reporte ordenado y válido.

Tenemos dos opciones:

1. controlar la concurrencia (ejemplos anteriores), “en qué momento puede imprimir”. Eso en cierto modo frena la concurrencia y consume muchos recursos.
2. Evitar el control de concurrencia (NO todo el problema de concurrencia lo permite, usualmente es para paralelismo de datos (separación de asuntos no)). Se basa en que los hilos se ejecuten de manera paralela pero en recursos distintos para que un hilo no se meta con otros. (NO existe un término formalmente, se le conoce como Conditionally safe). Los hilos pueden acceder a diferentes objetos de la memoria pero evitando condiciones de carrera. La estructura más sencilla para trabajarla son los arreglos porque está indexado, se pueden usar rangos de trabajo.

Para la opción 2 se ocupa una estructura de datos, se crea un arreglo.

Se almacenan en el arreglo los saludos y se utiliza otro for para recorrer el arreglo e imprimir esos saludos.

```

-- 
13 procedure greet(thread_number)
14   greets[thread_number] := format("Hello from secondary thread", thread_number
15   , " of ", thread_count)
16 end procedure
17

```

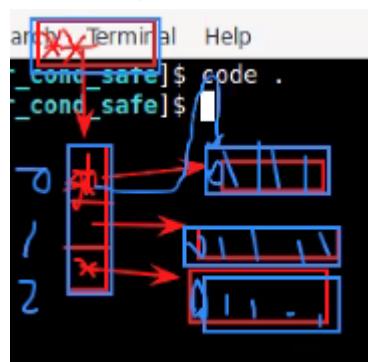
Aunque cumpla las tres condiciones de carrera, cada hilo se encarga de un solo espacio de memoria, por lo que se divide la memoria en partes privadas.

Implementación del ejemplo 10:

Crear un arreglo tipo:

a * nombre;

con el tamaño de los hilos a crear (hay que hacerlo dinámico porque no conocemos la cantidad)



`sprintf(buffer, "impresion")`

hace que la impresión vaya hacia la cadena de caracteres.

Lo que tiene el conditionally safe es que requiere de buen manejo de memoria pero es muy útil.

Sem5-b (jueves 16/09)

Quiz2: el miércoles 22 (materia hasta el lunes 13)

NUEVO TEMA: productor-consumidor

fue un problema inicial simplificado hecho por Dijkstra. Uno puede inferir una solución que soluciona otros muchos problemas.

1.10. Productor-consumidor

El problema del productor-consumidor en computación es un problema clásico de control de concurrencia, propuesto por Edsger Dijkstra. Se suele presentar de forma semi-concreta, lo que ayudar a identificar otros contextos donde su solución se puede aplicar. Hay dos variantes, una de buffer acotado y otra de buffer no acotado. De ambas se puede generalizar un patrón de software que puede enriquecerse con otros paradigmas de programación como el genérico y orientado a objetos.

ejemplo:

cadena de producción de un reproductor de video

recibe bytes genera matriz, recibe matriz, decodifica matriz, recibe decodificación y produce pixeles.

1.10.1. Problema de buffer acotado

El problema del productor-consumidor lo tienen dos entes concurrentes, uno que produce algo que el segundo consume. Ambos tienen acceso a un espacio finito (acotado) donde se alojan los productos, comúnmente llamado buffer y de naturaleza circular. Dado que los entes son concurrentes, uno está agregando productos al buffer mientras el otro está al mismo tiempo consumiéndolos (o quitándolos) del buffer.

El problema surge cuando ambos entes actúan a diferente velocidad. Si el consumidor es más rápido que el productor, podría consumir productos que aún no se han colocado en el buffer. Si el productor es más rápido que el consumidor, lo alcanza y sobrescribe productos que el consumidor aún no ha procesado.

El problema se resuelve cuando el consumidor procesa todos los productos y en el mismo orden en que el productor los genera. Al igual que ocurre con seres humanos, la solución implica imponer espera. Si el consumidor es más veloz, vaciará el buffer y deberá esperar a que el productor genere el próximo producto. Si el productor es más veloz y satura el buffer, deberá esperar a que el consumidor libere al menos un espacio para poder continuar produciendo.

create_threads sirve para crear más hilos. Para paralelismo de datos.

```
create_thread(produce, args)
create_threads[10, consume, 29, 20183, "ji"]
```

rand:

```
34 | end for
35 end procedure
36
37 function random_between(min, max):
38 | return min + rand() % (max - min)
39 end function
40
```

$$\begin{array}{r} \text{r} \times 40 \\ \hline 0 \dots 39 \\ \hline 10 \\ \hline 99 \end{array}$$

[10, 50]

2 semáforos, 1 por procedimiento:

```

design > prod_cons_bound.pseudo
● 00:17:22
1 procedure main(argc, argv[]):
2   if argc = 7 then
3     shared buffer_capacity := integer(argv[1])
4     shared buffer as array of buffer_capacity of float
5     shared rounds := integer(argv[2])
6     shared producer_min_delay := integer(argv[3])
7     shared producer_max_delay := integer(argv[4])
8     shared consumer_min_delay := integer(argv[5])
9     shared consumer_max_delay := integer(argv[6])
10
11   create_threads(1, produce)
12   create_threads(1, consume)
13 end if
14 end procedure
15
16 procedure produce:
17   declare count := 0
18   for round := 1 to rounds do
19     for index := 0 to buffer_capacity do
20       delay(random_between(producer_min_delay, producer_max_delay))
21       buffer[index] := ++count
22       print("Produced ", buffer[index])
23     end for
24   end for
25 end procedure
26
27 procedure consume:
28   for round := 1 to rounds do
29     for index := 0 to buffer_capacity do
30       value := buffer[index]
31       delay(random_between(consumer_min_delay, consumer_max_delay))
32       print("Consumed ", value)
33     end for
34   end for
35 end procedure

```

Pro (on)

4	0
3	1
2	2

4	3	1
0	4	-1
2	3	

a diferencia de un procedimiento, la función no genera consecuencias.
rand por ejemplo es muy procedimental, no muy funcional.

En la actividad 10:

Hay condición de carrera y hay que imponer un orden.

Primera opción para arreglarlo:

Conditionally safety.

Cada productor trabaja con su propia memoria.

Pero necesitamos que los dos comparten el buffer por lo que no es factible para el problema.

Segunda opción:

Control de concurrencia:

1. mutex: provocaría exclusión mútua (solo un hilo ejecutando a la vez las regiones críticas).

En la vida real uno no se detiene porque otro se detenga.

Mutex "debería de ser la última carta a jugar"

2. semáforo: permite orden, concurrencia sin serializar

Solución:

2 semáforos, uno para el productor y otro para el consumidor.

Para darle el nombre: preguntar si puede hacer algo.

wait: esperar si el semáforo está negativo y también decremento

signal: aumenta un valor inicial.

```
// @see `man feature_test_macros`
#define _DEFAULT_SOURCE
```

```
enum {
```

```

ERR_NOMEM_SHARED = EXIT_FAILURE + 1,
ERR_NOMEM_BUFFER,
ERR_NO_ARGS,
ERR_BUFFER_CAPACITY,
ERR_ROUND_COUNT,
ERR_MIN_PROD_DELAY,
ERR_MAX_PROD_DELAY,
ERR_MIN_CONS_DELAY,
ERR_MAX_CONS_DELAY,
ERR_CREATE_THREAD,
};


```

define constantes, aumenta en 1 la anterior.

sleep: resolución de segundos
el que duerme es el hilo que ejecuta el sleep.

micro sleep (usleep): resolución de microsleep.

No es permitido usar sleeps en las tareas y proyectos.

Sem6-a y Sem 6-b

1.10.2. Problema de buffer no acotado

Dijkstra publicó una segunda versión del problema del productor-consumidor que elimina las limitaciones del *buffer* acotado.

En esta versión, el *buffer* tiene memoria infinita, hay una cantidad arbitraria de productores, y una cantidad arbitraria de consumidores. Al igual que el anterior, el problema se resuelve si todos los consumidores procesan *todos* los productos *en el mismo orden* en que los productores los generan.

Rastreo de memoria:

```

22
23 procedure produce:
24   while next_unit < unit_count do
25     next_unit := next_unit + 1
26     declare my_unit := next_unit
27     delay(random between(producer_min_delay, producer_max_delay))
28     enqueue(queue, my_unit)
29     print("Produced ", my_unit)
30   end while
31 end procedure
32
33 procedure consume:
34   while consumed_count < unit_count do
35     consumed_count := consumed_count + 1
36     declare my_unit := dequeue(queue)
37     print("\tConsuming ", my_unit)
38     delay(random_between(consumer_min_delay, consumer_max_delay))
39   end while
40 end procedure
41
42 function random_between(min, max):
43   return min + rand() % (max - min)
44 end function
45
46 modificacion concurrente memoria_compartida
47


```

	A	B	C	D
1	unit_count		3	
2	queue			
3	next_unit		2	
4	consumed_coun		1	
5				
6				
7	producer 0	producer 1	consumer 0	consumer 1
8	24 while 0<3			
9	25 next_unit := 1	24 while 0<3		
10	26 my_unit := 1	25 next_unit := 1	34 while 0<3	
11	27 delay(0)	26 my_unit := 2	35 consumd:=1	34 while 1<3
12	28 enqueue(1)	27 delay(0)	36 my_unit := de	35 consumd:=2
13				
14				
15				
16				
17				
18				
19				
20				
21				

Cambio en makefile:

agrega args

run: ejecuta el ejecutable con los argumentos puestos en args

Por qué usamos cola en lugar de un arreglo dinámico?

porque el manejo de memoria dinámico para un arreglo es muy caro en eficiencia en cambio con la cola es solo un reajuste de punteros lo que se necesita.

Además, si se ocupa mucha memoria, el arreglo hace que la memoria quede muuy fragmentada (pide pedazos grandes y "realoca"), la cola solo pide pedacitos pequeños.

team: hilos de un tipo haciendo un trabajo (paralelismo de datos)

analizamos el pseudo porque se dan condiciones de carrera e hicimos un rastreo de memoria

condición de carrera: depende de quién llegue primero.

un enqueue y un dequeue requieren modificación de punteros (no es atómica), es decir: condición de carrera si no tienen control de concurrencia.

"modificación concurrente de memoria_compartida"

tres problemas de condición de carrera en el pseudo:

- next_unit := next_unit + 1
- enqueue(queue, my_unit) con declare my_unit := dequeue(queue)
- variable entera que controla la concurrencia. (la protegeremos con un mutex, protegiendo tanto la modificación como el while que lo contiene)

modificación: constantes

memoria_compartida: (variables privadas | seguridad condicional)

concurrente: (semáforos | exclusión mutua-mutex)

en exclusión mutua se busca que tarde lo mínimo de tiempo

mutex: 1) lock unlock | 2) wait signal | release...

thread safe / monitor: contenedores donde todas las operaciones tienen protección de concurrencia con un mutex.

No pueden coexistir: control de flujo con control de concurrencia.

produce1 sale de la necesidad de separar el control de concurrencia del de flujo. sacamos lo que ocupaba el control de concurrencia y utilizar un resultado (variable local) lo que ocupaba la concurrencia. se puede modularizar para evitar redundancia.

contras:

produce1: redundancia de trabajo

produce2: break

Corregidas las condiciones de carrera por la cola y la de la variable entera pero:
que pasa si los productores no han producido y los consumidores sacan de una cola vacía?

usando semáforos se solucionaría? quizás

pero no tendría sentido hacer un "can produce" con un buffer no acotado ya que es infinito. más bien usaremos un can_consume que usarán tanto los productores como los consumidores.

El orden del control de concurrencia es muy importante.

Cuidado con las esperas activas (hilos esperando en CPU (apariencia de ciclo con condición (while if...)))

punteros a funciones:

void*(*subroutine)(void*)

usamos mutex para proteger a la cola. Ok
Pero que pasa si hay muchas colas? creo un mutex por cada mutex? no
creamos una cola thread safe. sus subrutinas son ya protegidas por exclusión mutua. (serializando las
funciones de la cola).
en java se conoce como parallel_contenedor. (es un mutex que protege a un contenedor. el patrón utilizado se
conoce como monitor).

isEmpty

thread-safe:
significa que no genera condiciones de carrera

cuando tengo un mutex (como en is_empty), se tiene garantía de exclusión mutua. sin embargo no puedo
suponer que la realidad que yo hago cuando verifico is_empty es la misma luego de preguntarla (otro mutex
pudo haber sido liberado). (ya no tiene sentido). el is_empty no garantiza que la cola luego de liberar el mutex
siguiente sea vacía o no.

ejemplo:

```
while (!queue_is_empty(queue)) { // thread-safe
    pthread_mutex_lock(&queue->can_access_queue); // pido de nuevo el mutex pero no sé si la realidad de
antes se mantiene. (NO sé si otro hilo ya lo eliminó)
```

consistencia de acceso a la cola

una subrutina unsafe no puede invocar una thread-safe. Para que ellas puedan ser llamadas de forma segura
dentro de un mutex

create, init y destroy no pueden ser thread-safe porque ocupan un mutex que no ha sido creado o que está
siendo eliminado

No se puede invocar en un ambiente concurrente (solo debe invocarlo una vez).

Actividad 12 [prod_cons_pattern]

La solución al problema del productor-consumidor es aplicable a una cantidad significativa de contextos.
Generalice el código de su solución para que pueda ser reutilizado. Utilice los mecanismos de reutilización de
código de su lenguaje de programación, por ejemplo, subrutinas y metaprogramación en C, o programación
orientada a objetos polimórfica y plantillas en C++.

Cree una simulación de una red de productores, consumidores, y repartidores. Haga que los consumidores
sean dueños de las colas de trabajo, de tal forma que haya una relación 1:1 entre una cola y un consumidor.

Agregue a su código el concepto de repartidor. Un repartidor es tanto un productor como un consumidor (hilo
de ejecución) cuya funcionalidad es

Patrón de diseño: sirve para solucionar problemas en varios contextos. Si una persona conoce el patrón, le
será más fácil entender nuestro código y hasta lo podrá mejorar, haciéndolo más compacto.

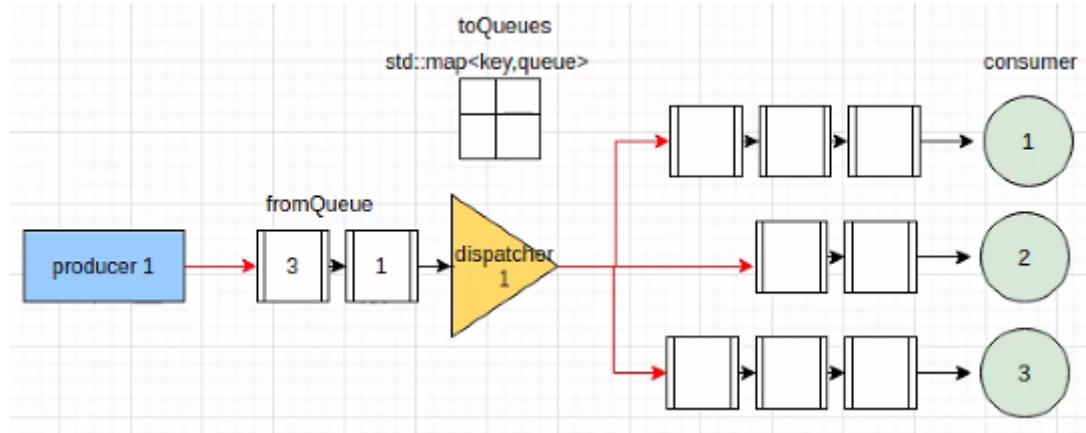
Código a utilizar en el proyecto1.

Código asociado a redes de computadoras. (hacer un enrutador: especie de servidor que tiene un nivel de
conurrencia muy grande).

Mala programación: redundancia de código.

El código que veremos se utiliza para aprovechar la reutilización de código y evitar redundancia.

para el diseño usaremos dibujos informales (simulando una cadena de producción)



Proyectos hechos en C++.

Componentes genéricos: sirven para repartir cosas, en este ejemplo representa un repartidor.

El productor genera números al azar para darle a los consumidores (1,2 o 3).

El repartidor reparte según el número elegido por el productor.

el productor es un hilo, el repartidor es un hilo, entre ellos media un buffer (ej:una cola) (no pueden hablar directamente, ocupan un buffer). En este ejemplo usaremos una cola no acotada aunque es más real una acotada.

Cuando el productor quiere que el repartidor pare, hay que poner mensajes por la cola.

En ese diseño se cumple los patrones de consumidor-productor

arranca con cola vacía, se duerme cuando no hayan elementos y se despiertan cuando hayan elementos.

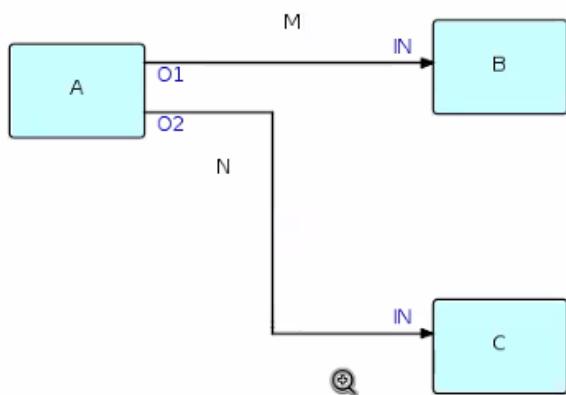
La cola es threadsafe y hace dormir a los repartidores

ensamblador. métafora de la cadena de producción... los tres son ensambladores.

paradigma de flujo de datos (es muy útil, no se ve en la carrera), tiene sub-paradigmas.

El que haremos es muy imperativo con 4 componentes.

La forma de programar en ese paradigma es muy visual



sinpe móvil, pago con tarjeta etc son ejemplos de flujo de datos. (uno le dice a la máquina el qué hacer y ella se preocupa del cómo (declarativo)).

resumen: flujo de datos es un paradigma declarativo en el que se diseña por dibujos, hay componentes (hilos) que se comunican entre sí por un buffer.

En el proyecto se diseña con código UML y dibujos parecidos a los de arriba.

Makefile:

con macro

-DSIMULATION

-DWEB SERVER

defina eso en tiempo de compilación. equivale a:

#define SIMULATION

```
Makefile
1 # C/C++ Makefile v2.2.0 2021-Aug-22 Jeisson Hidalgo ECCI-UCR CC-BY 4.0
2
3 # Compiler and tool flags
4 CC=cc
5 XC=g++
6 DEFS=-DSIMULATION
7 FLAGS=$(strip -Wall -Wextra -pthread $(DEFS))
8 FLAGC=$(FLAGS) -std=gnu11
9 FLAGX=$(FLAGS) -std=gnu++11
10 LIBS=
11 LINTF=-build/header_guard,-build/include_subdir
12 LINTC=$(LINTF),-readability/casting

main.cpp .../simulation < main.cpp .../webapp

src > simulation > main.cpp > ...
1 /// @copyright 2020 ECCI, Universidad de Costa Rica. All rights reserved
2 /// This code is released under the GNU Public License version 3
3 /// @author Jeisson Hidalgo-Céspedes <jeisson.hidalgo@ucr.ac.cr>
4
5 #ifdef SIMULATION
6
7 #include "ProducerConsumerTest.hpp"
8
9 int main(int argc, char* argv[]) {
10     return ProducerConsumerTest().start(argc, argv);
11 }
12
13 #endif // SIMULATION
14
```

es para ver si se ejecuta la simulación o el webserver. (usa ifdef y endif)

Modelo vista controlador (MVC)

tenemos código que se encarga de la lógica del dominio (modelo)

modelo: calculan la factorización prima de un número (ejemolo) trabajan con el almacenamiento (sacan datos, hacen lógica y producen datos.) No hablan con nadie.

Ej: IDE que guarda, carga archivos.

Puedo generar una vista o dos o más del mismo modelo (ej: archivo abierto en dos ventanas de VSC).

Se pueden generar EVENTOS en la vista. La vista puede hablar al modelo (el modelo no le responde).

Eventos: mecanismo de comunicación genérico.

vistas: pueden ser gráficas de usuario, de web, línea de comandos, en servidor etc. Cambian de acuerdo a la interfaz del usuario, los modelos NO cambian.

El controlador arranca y pone a trabajar al modelo y a la vista. Los controladores se hablan entre ellos.

Es un patrón muy utilizado.

servidor:

objeto controlador

método start, stop

deme status (corriendo, detenido)

restart (stop y start)

tiene bitácora.

validar/reajustar configuración

extensión: h ¿es C o C++?

Es mejor desambiguar si es con C, poner archivo.**hpp**

Makefile distingue entre ambos así que hay que tener cuidado.

Es importante dejar una marca para cada variable, si es parte de una clase: usar this
si es de la std, poner std. etc... (para que no parezca que son variables globales, sino que se sepa de donde salen.)

productor ocupa la cantidad/tamaño del paquete, el delay y la cantidad de consumidores.

repartidor con su propia cola.

los consumidores (vector de std)

pista: la forma en que se detiene es con la 3er forma del consumer que vimos.

Sem7b

explicación del src para el proyecto

el controlador crea las colas para cada consumidor

El productor tiene una sola cola, al igual que los consumidores

Los repartidores tienen al menos dos colas.

main llama al controlador, controlador:

1. crear todos los hilos
2. crear colas e intercomunicar hilos por medio de colas
3. arrancar maquinaria

productor:

producertest:

crea productor (ProducerTest que es un productor de mensajes de red (hereda))

run lo que hace es un ciclo para producir n cantidad de paquetes de red

Plantilla Producer:

Plantilla para hacer clases

puede producir enteros, sonidos etc (es genérico)

Hereda de un hilo de ejecución (productor es automáticamente un hilo de ejecución) como thread es abstracta, producir también lo será.

explicit: lo que dice es que el compilador no haga conversiones implícitas. (no pase de cola a productores)

conversión implícita ejemplo: de entero a fraccional (7 a 7/1)

inline void (invocación en línea): el compilador decide (hace lo que le da la gana)... trata de optimizar de forma que se salte la invocación y lo reemplaza con solo una línea. (setProducingQueue)

método produce: pone el dato en una cola (en la de producción).

clase Thread:

es una clase abstracta

trata de encapsular (wrapper) la creación de hilos

facilita la concurrencia con objetos (utiliza POO para crear hilos, ya no es procedimental)

deficiencia: solo invocan una subrutina con su constructor (el constructor recibe esa subrutina)

NetworkMessage:

struct con una fuente ,un cuerpo y un destino

un mensaje de red puede ser una condición de parada (en el proyecto decidimos los valores de los atributos para determinar comparando si es el final o no de una cola) ej: source: 0, dest:0, body: null.

Clase Cola:

encapsulador de una cola genérica de C++

es threadsafe (con mutex para proteger las inserciones y extracciones y semáforo para que el consumidor espere si la cola está vacía para poder consumir)

de buffer no acotado

DISABLE_COPY:

hace delete de algunos métodos como constructor por copia para que las colas no se puedan copiar los objetos de esa clase (en este caso la cola).

tiene los métodos de c++ de una cola pero protegidos con control de concurrencia.

hace falta cambiar los nombres push y pop por enqueue y dequeue además de más métodos como clear.

Clase Semaphore:

usa sem_t

usa compilación condicional para saber si estamos en mac (usaremos semáforos con nombre (de disco(sem_t*)), porque los semáforos de memoria(sem_t) están deprecated)

el semáforo binario no tiene ownership (puede ser aumentado o decrementado por varios)

Consumidor:

consumerTest:

es un consumidor de mensajes de red

override: le indica al compilador que el método sobreescribe un método virtual. Se utiliza con métodos virtuales como una buena práctica.

Clase Consumer (abstracta por derivar de thread)

consumeForever: saca infinitamente de la cola hasta que llega a la condición de parada (rompe el ciclo). La cola es threadSafe.

Repartidor

DispatcherTest (deriva de Dispatcher).

Dispatcher:

Arreglo asociativo: como una arreglo normal pero en lugar de usar índices como números, puede asociar una clave de cualquier tipo a un contenido (imagen - sonido - hilera etc).

```
src > prodcons > Dispatcher.hpp > Dispatcher<KeyType, DataType> > fromQueue
18 template <typename KeyType, typename DataType>
19 class Dispatcher : public Consumer<DataType> {
20     /// Objects of this class cannot be copied
21     DISABLE_COPY(Dispatcher);
22
23 protected:
24     /// Alias to the inherited queue for a more meaningful identifier
25     Queue<DataType*>*& fromQueue = Consumer<DataType>::consumingQueue;
26
27     /// This thread will distribute elements to the following queues
28     std::map<KeyType, Queue<DataType*>> toQueues;
```

In 27 y 28

hace un arreglo asociativo que redirecciona con enteros a otras colas.
extractKey

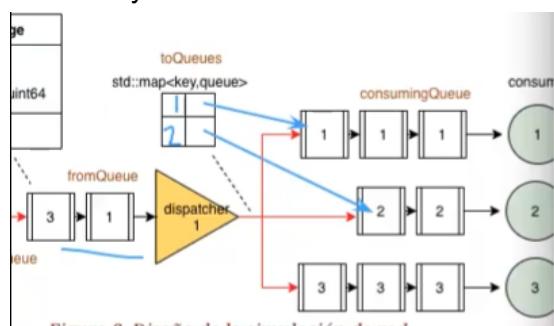


Figura 3. Diseño de la simulación de red

```

40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
}
/// Register a map. When the data to be consumed has this key, it will be
/// redirected to the given queue
inline void registerRedirect(const KeyType& key, Queue<DataType>*& toQueue) {
    this->toQueues[key] = toQueue;
}

/// Override this method to process any data extracted from the queue
void consume(DataType data) override {
    const KeyType& key = this->extractKey(data);
    const auto& itr = this->toQueues.find(key);
    if (itr == this->toQueues.end()) {
        throw std::runtime_error("dispatcher: queue's key not found");
    }
    itr->second->push(data);
}

```

reb serial que puede ser paralelizado mediante una cadena de productores y consumidores. Este es el objetivo del proyecto01

Proyecto 11 [network_simul_packet_loss]

ed para introducir pérdida de paquetes. Reciba un parámetro

```

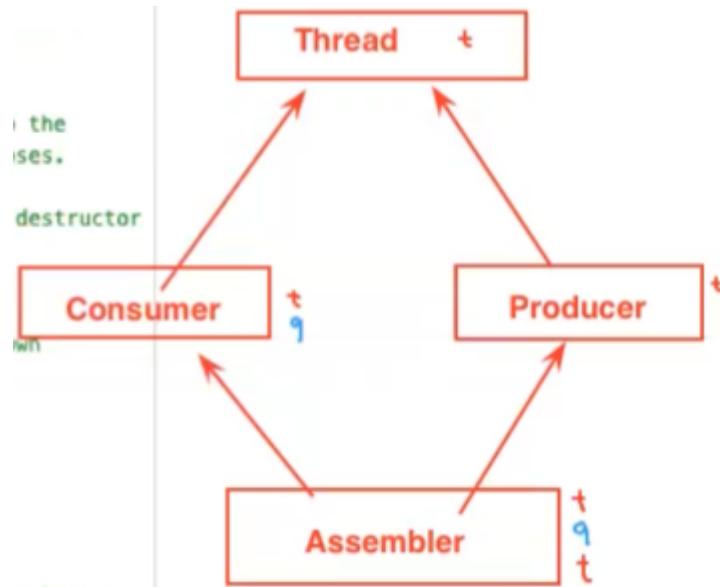
ProducerTest.cpp 1  ConsumerTest.cpp  DispatcherTest.hpp  ProducerConsumerTest.cpp 5
src > simulation > < ProducerConsumerTest.cpp > start(int, char *[])
51 // Producer push network messages to the dispatcher queue
52 this->producer->setProducingQueue(this->dispatcher->getConsumingQueue());
53 // Dispatcher delivers to each consumer, and they should be registered
54 for ( size_t index = 0; index < this->consumerCount; ++index ) {
55     this->dispatcher->registerRedirect(index + 1
56         , this->consumers[index]->getConsumingQueue());
57 }

```

Cuando el dispatcher termina de consumir pone la condición de parada en todas las colas

Assembler:

clase abstracta
consume de una cola y produce en otra cola
es más sencillo que el dispatcher
hereda del consumidor y del productor
 pierde algunos datos al azar
considera el problema del diamante (curiosidad de C++)



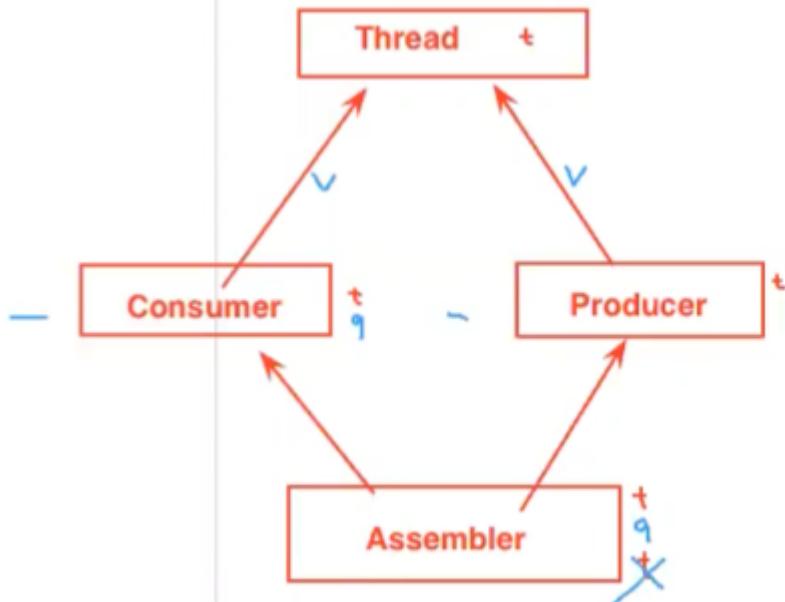
diamante formado entre thread y assembler

problema con los atributos.

atributo: thread. puntero a hilo de ejecución

c++ lo duplica. (producer y consumer tienen una copia de thread. además consumer tiene una cola => assembler como hereda de ambos tendrá 2 threads lo cual no tiene sentido y una cola). C++ tiene herencia múltiple (Java no la permite).

Lo que se hace es usar un virtual Thread para que C++ detecte que el abuelo tiene un puntero a thread, que los hijos también pero el nieto solo debe tener uno (evita redundancia):



problema solucionado.

Bifurcación concurrente en el curso

concurrencia de tareas: separación de asuntos (forma natural de trabajar, una tarea por trabajador)

paralelismo de datos: incremento del desempeño

... ocurren más en ciencia. con HPC (high performance computing)

Optimización: no se debe hacer una optimización arriesgada/casera. La optimización es una etapa opcional. Se devuelve al modelo de la solución. tiene dos desventajas: el código será más complejo y más específico (para una máquina, depende del contexto). No se debe sobreoptimizar (quitar la generalidad y meter mucha complejidad).

Método sugerido para optimizar (siempre y cuando hayan recursos, tiempo y clientes para hacerlo):

1. medir rendimiento del código antes de realizar las modificaciones
2. Analizar el código para detectar las regiones críticas a optimizar (profiling)
profiling: análisis de recursos en código con fines de rendimiento.
3. Hacer las modificaciones que se cree incrementarán el rendimiento en las regiones críticas.
4. Asegurarse de que las modificaciones sean correctas (correr el programa de pruebas)
5. Medir el rendimiento del código después de realizar las modificaciones y comparar para determinar si hubo ganancia. con métricas.
6. Indiferentemente de si se incrementó o no el rendimiento, documentar las lecciones aprendidas, ya que servirán para otros desarrolladores que intenten optimizar la misma sección de código.
7. Si es realmente requerido incrementar aún más el rendimiento, repetir desde el paso 2.

...

Optimizaciones tiene cara de artesanía (muchas hipótesis para ver si funciona en mejor forma).

En profiling casi siempre se busca optimizar lo que tenga repetición (ciclos o recursividad)

Descomposición y mapeo (una optimización más)

Descomposición:

tomar/identificar unidades de trabajo (tareas o datos) que se pueden realizar de forma independiente para poder ejecutarla de forma paralela.

Una vez definidas, las tareas serán: unidades indivisibles.

Es seguido por el mapeo, el cual consiste en repartir esas unidades de trabajo a los ejecutantes.

granularidad fina: se definen muchas tareas pequeñas (incrementa el grado de concurrencia y la interacción entre tareas (ejecutantes).

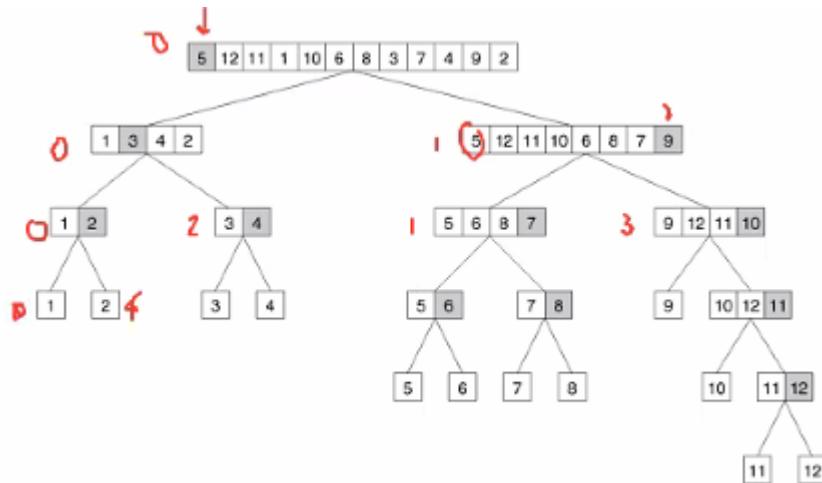
Granularidad gruesa: pocas tareas grandes. Disminuye la interacción entre ejecutantes pero reduce el grado de concurrencia. (se vuelve casi serial).

No hay mejor ni peor granularidad, depende del problema.

Formas de descomponer: depende de la naturaleza del problema. (todos tienen repetición)

Algunas técnicas (existen muchas pero nos enfocaremos en una)

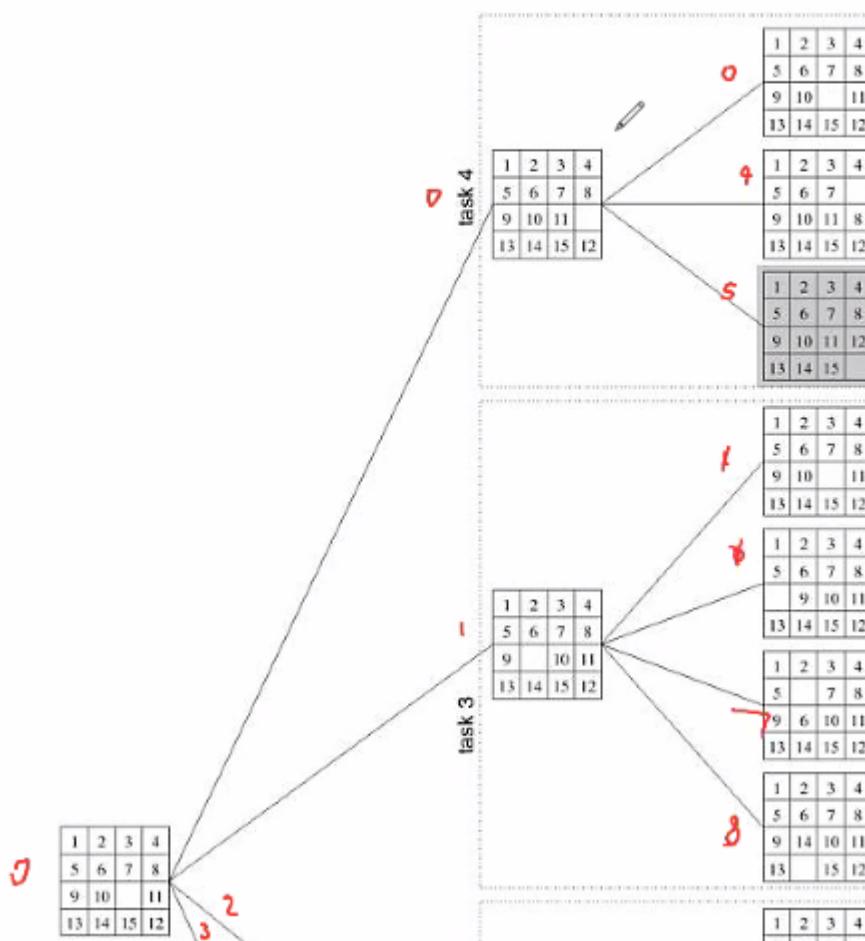
Desc. recursiva: una tarea que empiezo a hacer y me beneficio de que otro me ayude con ese pedazo de la tarea. (divide y vencerás). ej: quickSort



Desc. de datos: tenemos datos (no tanto tarea) en cualquier conjunto de datos, que podemos repartir entre ejecutantes. Típicamente obtiene un incremento del desempeño sistemático.

Desc. exploratoria: puede ser recursiva o iterativa. Se anda buscando una solución o varias...

Figure 3.18. The states generated by an instance of the 15-puzzle problem.



ejemplo: _____

No se sabe en qué momento se encontrará la solución. Puede ser en el nivel [0, 1 ,2 ...a] del árbol. A veces corre lento o a veces rápido, depende de cuando se encuentre la solución.

se podría implementar una variable compartida para saber si ya se encontró una solución (protegida por mutex para evitar condición de carrera). cada hilo antes de subdividir el problema consulta por la variable.

Desc. especulativa: le asignamos un hilo a lo que creemos que puede llegar a ser (una probabilidad) solución y a lo que no, no.

ej: búsqueda en google que especula que el usuario dará click a la primer página y por eso la carga en el disco para que si da click, se ingrese más rápido.

Las cuatro técnicas se pueden combinar y hasta existen más.

Mapeo:

Actividad que sigue inmediatamente a la descomposición.

Se le asignan las aplicaciones indivisibles a cada hilo.

Algoritmos (-distribuciones- no utilizada mucho) de mapeo:

2 tipos:

estático (cuando se pueden hacer las asignaciones de las unidades de trabajo antes de realizar el trabajo, disminuye la interacción, son fáciles de implementar y son baratos de coste, solo ocupo saber cuánto trabajo debo realizar (conozco el conjunto de datos))

dinámico (no sé cuánto trabajo habrá que hacer, se sabrá hasta el tiempo de ejecución) ambos tienen pros y contras

Es malo que haya mucha interacción entre hilos porque se vuelve más lento.

Práctica de algoritmos de mapeo en google sheets: [Hoja de cálculo sin título - Hojas de cálculo de Google](#)
3 casos se quieren paralelizar. Los datos están escritos en la celda celeste e indican la cantidad de unidades de tiempo que tome hacer el trabajo. Arriba están los índices (trabajo repartido en un arreglo).

el arreglo tiene 22 elementos.

la duración de trabajo varía para cada caso

caso1: muy juntos

caso2: exponencial (2^n)

caso3: azar

Mapeo por bloque:

Bloques continuos asignados. Su diseño es funcional, se resuelve por un modelo matemático. Se puede hacer desbalanceado o balanceado.

Fórmula para balancear el mapeo por bloque:

D: cantidad de datos

w: cantidad de workers (trabajadores-ejecutantes)

i: número de hilo

$$\text{start}(i, D, w) = i \left\lfloor \frac{D}{w} \right\rfloor + \min(i, \underline{\text{mod}}(D, w))$$

$$\text{finish}(i, D, w) = \text{start}(i + 1, D, w)$$

login: Mon Oct 4 08:46:52 on ttys000
jhc — bash — 177x34

i	s	f	$[s, f]$	$\text{mod}(s, w)$
0	0	4	[0, 4[4
1	4	8	[4, 8[4
2	8	12	[8, 12[4
3	12	16	[12, 16[4
4	16	19	[16, 19[3
5	19			

$s(i, 19, 5) = 3i + \min(i, 4)$

$19 \mid 5$

$\frac{19}{4} = 4 \text{ resto } 3$

somos tan rápidos como el más lento del equipo)

Mapeo estático cíclico:

Como se reparte una baraja de naipes (toda ella) a varios jugadores (una carta a la vez).

Tiene más forma de algoritmo que de matemática.

El algoritmo de mapeo que uno escoja tiene un impacto enorme sobre el rendimiento, se deben implementar de acuerdo a un análisis previo, para saber cuál nos conviene.

Mapeo bloque cíclico / mapeo cíclico con bloques:

contiene ventajas y desventajas de los anteriores.

es el mismo mapeo cíclico solo que “en lugar de repartir una carta a la vez, reparto n cartas a la vez”

Es decir, designo el tamaño de bloque.

Mapeo dinámico:

de tipo dinámico, ya no es estático.

Se sabe qué le toca a quién hasta en tiempo de ejecución.

El primer ejecutante que termine su trabajo, solicita la próxima unidad de trabajo disponible.

No se sabe a quién le tocará la próxima unidad si dos ejecutantes la solicitan.

A nivel de quiz se le asignará al de índice menor.

Relacionado con el patrón productor-consumidor.

Métricas

Incremento de velocidad (speedup)

relación entre lo que duraba antes y lo que dura después.

$$S = \frac{T_{\text{before}}}{T_{\text{after}}} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

No tiene unidades, sino que se cuentan por “veces” “cuantas veces es más veloz el algoritmo de ahora respecto al de antes”

Punto de referencia (comparando un dato consigo mismo): es 1. $1/1 = 1$.

problema del incremento de velocidad: el aumento de los recursos los cuales tienen un costo.

Por eso....

La eficiencia considera el incremento de velocidad y el coste de los recursos.

Eficiencia:

$$E = \frac{\text{speedup}}{\text{workers}} = \frac{T_{\text{serial}}}{T_{\text{parallel}} \cdot w}$$

Los recursos serán los hilos ejecutantes que usemos.

entre 0 y 1, donde 1 es el tope o el máximo grado de eficiencia. Se utilizan los recursos al máximo (eso es muy eficiente)

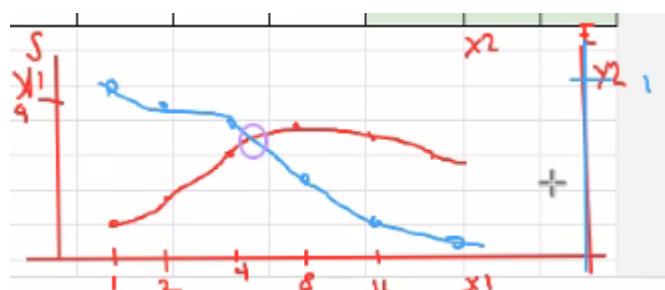
El uso de gráficas con estas métricas son muy útiles para exponer.

en un eje: tiempos

otro: speedup

otro: eficiencia

otro: cantidad de hilos



En los mapeos estáticos, los valores del arreglo no afectan los efectos del mapeo

análisis

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	
1	Caso 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	0	1	2	3	max	speed! efficier		
2	Serial	1	2	1	3	2	1	1	2	1	3	1	2	2	3	2	2	1	1	2	2	2	2	2	39			39	1.000	1.000		
3	Bloque	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3	10	10	10	9	10	3.900	0.975	
4	Cíclico	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	1	9	12	7	11	12	3.250	0.813	
5	BloqueCicli	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	2	9	11	12	7	12	3.250	0.813	
6	Dinámico	0	1	2	3	0	2	1	2	0	1	3	0	2	3	0	1	2	2	3	0	1	2	10	10	10	9	10	3.900	0.975		
7																																
8	Caso 2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	0	1	2	3	max	speed! efficier		
9	Serial	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8199	1633	327	6553	1311	2621	5242	1048	2097	4194303				4194303	1.000	1.000		
10	Bloque	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3	63	4032	126976	4063232	4063232	1.032	0.258	
11	Cíclico	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	1118481	2236962	279620	559240	2236962	1.875	0.469		
12	BloqueCicli	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	197379	789516	3158064	49344	3158064	1.328	0.332		
13	Dinámico	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	1118481	2236962	279620	559240	2236962	1.875	0.469		
14																																
15	Caso 3	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	0	1	2	3	max	speed! efficier		
16	Serial	42	97	73	75	13	14	8	10	22	63	23	77	14	35	69	24	7	25	22	42	21	50	826				826	1.000	1.000		
17	Bloque	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	314	203	149	160	314	1.000	0.250		
18	Cíclico	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	119	284	195	228	284	1.106	0.276		
19	BloqueCicli	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	256	312	147	111	312	1.008	0.252		
20	Dinámico	0	1	2	3	0	0	0	2	3	0	2	1	3	2	3	0	2	2	0	2	1	3	186	195	215	230	230	1.365	0.341		
21																																
22	Dinamico 1	1	2	3	4	6	7	8	Dinamico 2			Dinamico 3			186																	
23	0	1	3	4	6	8	10	0	1	17				0	42	55	69	77	140	164	186											
24	1	2	3	6	8	10		1	2	34				1	97	174	195															
25	2	1	2	4	6	7	8	10	2	4				2	73	83	106	141	148	173	215											
26	3	3	4	7	9				3	8				3	75	97	111	180	230													

análisis del caso 2:

Conclusiones:

los mapeos estáticos son sumamente sensibles al orden de las unidades de trabajo (cualquier de las 3) ellos mapean por posiciones, sin saber el tiempo. Un orden puede hacer que las cosas sean maravillosas o refeas.

El mapeo dinámico se comporta siempre como el mejor.

La ventaja de los estáticos es que disminuye la interacción entre los hilos. Además el dinámico requiere más recursos (control de concurrencia).

Si no conozco la naturaleza de los datos, uso dinámico.

Si conozco la naturaleza de los datos (un patrón en los datos) entonces estudio e implemento el mapeo estático más eficiente.

No hay mapeo perfecto, depende del problema.

Hacer actividad 36 (mapping_simulation)

Cómo se implementan los mapeos?

estático por bloque:

```
procedure block(data, data_count, thread_count)
```

```
    declare my_start := start(thread_number, data_count, thread_count)
```

```
    declare my_finish := start(thread_number +1, data_count, thread_count)
```

```
    for (index := my_start; index < my_finish; ++index)
```

```
        process(data[index])
```

```
    end for
```

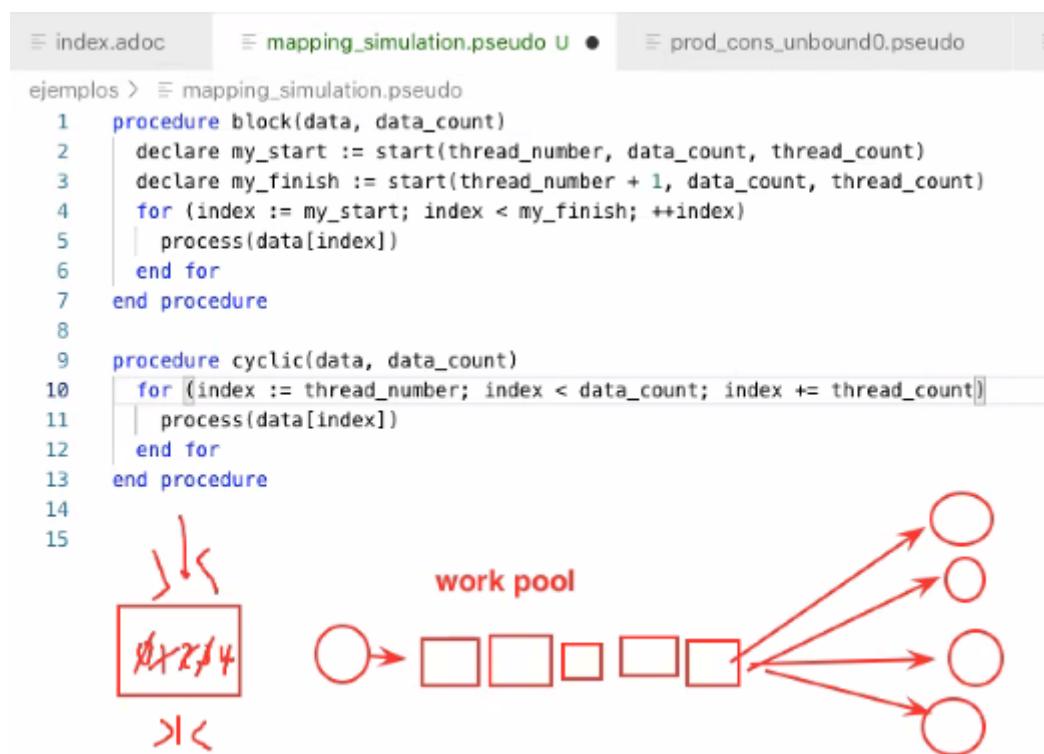
```
end procedure
```

estático cíclico:

```
procedure cyclic(data, data_count, thread_count)
    for (index := thread_number; index < data_count; index += thread_count)
        process(data[index])
    end for
end procedure
```

dinámico con 2 maneras:

- 1) usando contadores. (contador protegido por mutex). El hilo que se desocupa se asigna la siguiente unidad.
- 2) work pool (buffer - cola): pongo las unidades a trabajar ahí.
Cola protegida por un semáforo y un mutex. Con un productor y con consumidores.
Patrón productor consumidor.



sem 8-b

mapeo dinámico aleatorio.

Las unidades se asignan de forma aleatoria a los ejecutantes para que las trabajen.

El repartidor elige de forma aleatoria a qué cola enviar la unidad.

Con grandes cúmulos de datos, el azar suele ser muy bueno

Métrica de eficiencia

1.0 es una cota, ya que no se puede tener tanta eficiencia (todos los hilos trabajando de inicio a fin de forma paralela, no habría serialización).

0 a 1.0 | 0 es nada eficiente.

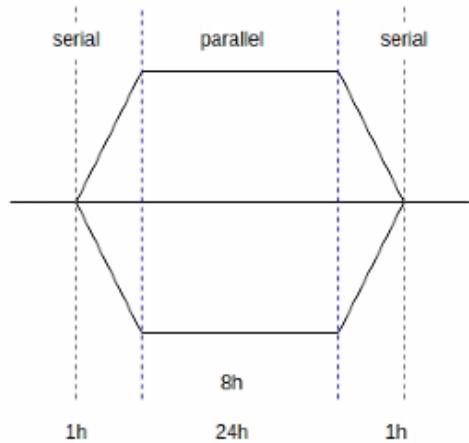


Figura 8. Gráfico de tiempo de un programa con tres fases: serial inicial, paralela, y serial final

se reduce de 26 horas totales a 10 horas si tres hilos trabajan paralelamente.

Lo mínimo que puede durar es 2 horas (no se puede achicar la parte serial)

La ley de Amdahl establece que el máximo speedup que un programa puede alcanzar por paralelización está acotado por la porción del programa que se mantiene o solo puede ser serial. La ley de Amdahl se puede utilizar a favor para acotar la duración y saber en cuánto tiempo se puede resolver un problema con una cantidad adecuada de recursos.

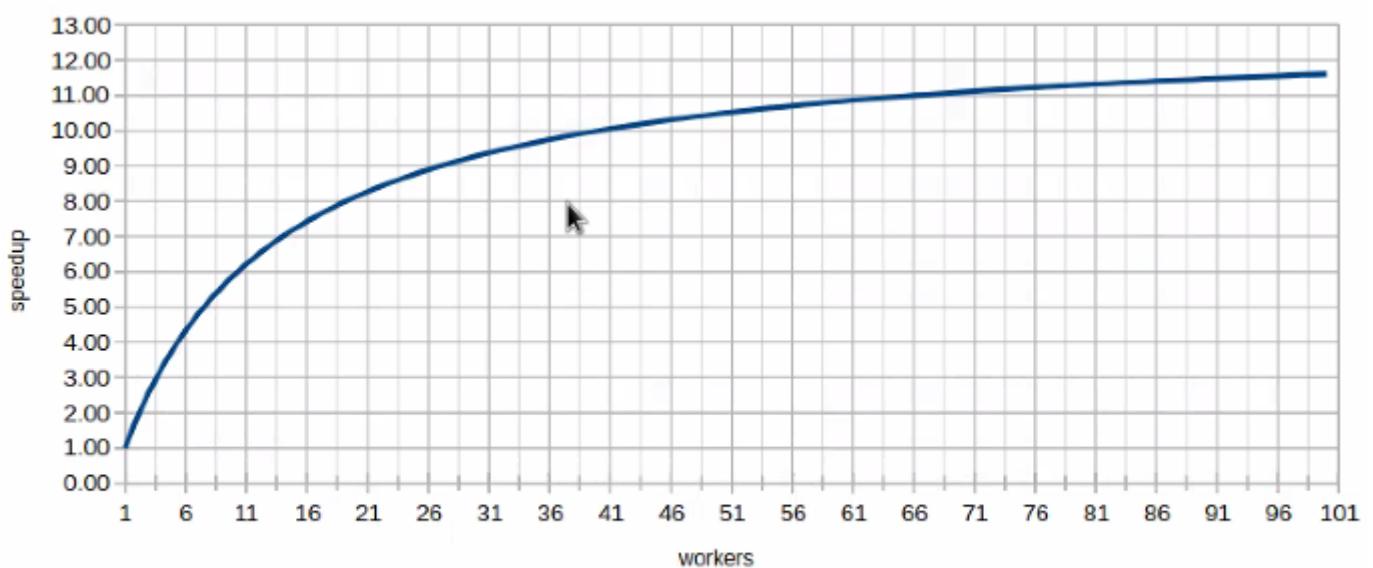
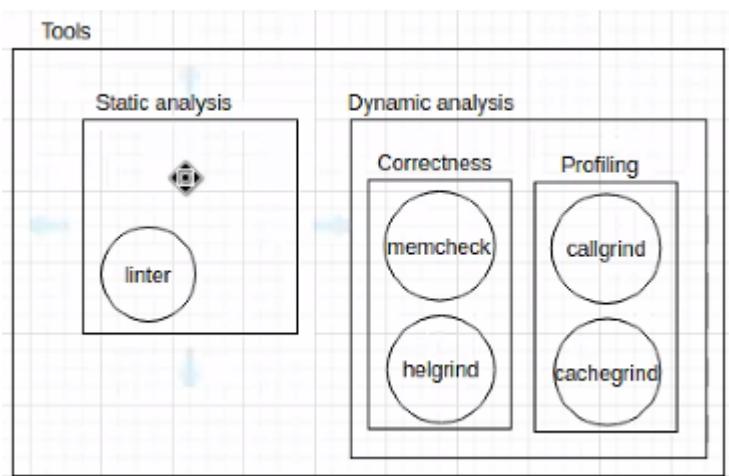


Figura 9. El incremento de velocidad está acotado por la porción serial de un programa

Se agradece la ley de Amdahl, porque ayuda a conocer el fenómeno y prevenir una sobrecompra de computadoras sin sentido por ejemplo.

Profiling

pasa que el programa es lento y grande (25Millones de líneas por ejemplo), se quiere optimizar... cómo se hace para optimizarlo? con herramientas de profiling
pertenece al análisis dinámico de código



las estáticas trabajan con el código fuente.

(compilador que genera warning por ejemplo)

Las dinámicas tienen como entrada al ejecutable (ejemplo: sanitizers y de corrección).

Las de profiling son dinámicas. ejemplo la de google chrome (lightning house). Valgrind tiene a callgrind y a cachegrind.

Para hacer profiling debemos escoger una herramienta

Optimizar lo ya optimizado es aún más difícil, a nivel en que uno avanza en niveles (de programación y abstracción) también.

Recordar que la optimización es un arte, a veces es más fácil optimizar la primera versión.

Visualización en computación: hacer visible lo que no lo es a simple vista. ejemplo: graphic equalizer que permite ver las potencias del sonido. Otro ejemplo: un radar, otro: gráfico del mundo de como se expande el covid.

La visualización utiliza la graficación.

Usualmente el ciclo que más funciona paralelizar es el más externo. Cuando se busca optimizar para paralelizar, se buscan ciclos. La herramienta de callgrind sirve para eso, para hallar la instrucción que más consume procesamiento de todo el código (self). Después de allí se recomienda entender el código y hacer un caso ejemplo a mano para saber cómo lo podría descomponer.

Cluster: red de computadoras homogéneas optimizadas para correr un programa lo más rápido posible.

Nodo máster/login: computadora cualquiera (su objetivo es que la gente se conecte a él y preparar los procesos que van a correr, pero él no corre nada.).

Máquinas esclavas: son las que corren los programas.

comando:

lscpu // para saber la info una máquina

Cultura de clúster: se debe ser solidario
prohibido ejecutar programa en clúster

ssh compute-0-N

la 1 no funciona porque está dañada. están: 0 2 3

nohup: no hang up: hace que el programa corra de manera protegida (el programa corre en segundo plano, independiente de mi salida).

top: dice los top procesos que consumen el cpu.

htop:

ps -eu

para matar un proceso: kill #proceso
si no muere: kill -9 #proceso
el -9 lo mata forzosamente.

Si corremos una tarea serial, lo haremos en el compute 0.

Si corremos concurrentemente, lo hacemos en compute 2 y compute 3 con cuidado de que no haya otro estudiante

Sem10-a

Proyecto pensado para trabajar en paralelo con cada integrante del equipo.

Volvemos a concurrencia de tareas:

objetivo: hacer las cosas bien, que cada ejecutante haga lo que deba de hacer y no genere errores.

Contrario al paralelismo de datos, que busca acelerar y optimizar.

Recomendación: libro pequeño de semáforos [LittleBookOfSemaphores.pdf \(greenteapress.com\)](http://greenteapress.com/ LittleBookOfSemaphores.pdf)

Primitivas:

pseudo para resolver problemas de sincronización y código de Petri.

Hay muchas formas de fórmulas soluciones a problemas de sincronización.

TLA es un formalismo matemático para problemas discretos (para gente de matemática). Leslie Lamport (LAtex).

Algoritmos (pseudo)

Ventaja: Es muy fácil de escribir, leer y entender porque es muy preciso, se puede enfatizar en el problema concurrente.

Desventaja: no hay un estándar de como escribir. 2) Cada autor hace lo que le gusta, hay tendencias por ejemplo hacia la matemática o hacia C por ejemplo. 3) No es verificable automáticamente, es decir, entre más se acerque al lenguaje natural, más ambigüo será y no hay software que indique si es válido y correcto.

Redes de Petri

Es un modelo formal y matemático

taskc: tareas de concurrencia.

Teniendo las rutas claras, puedo listarlas e implementar la solución al problema de concurrencia; después verificar si mi solución soluciona todas esas rutas de ejecución. Sino, debo hacer una mejor solución.

Redes de Petri

Es un área de investigación, un modelo formal de Allan Petri basado en matemática; las redes están hechas para diseñar o modelar sistemas (puede ser concurrencia)

Sistema: conjunto de componentes que recibe entradas y produce salidas. Es como una función recursiva donde se conecta la salida de un sistema a la entrada de otro. Los sistemas guardan como memoria de lo que ha ocurrido antes si así se quiere (estados).

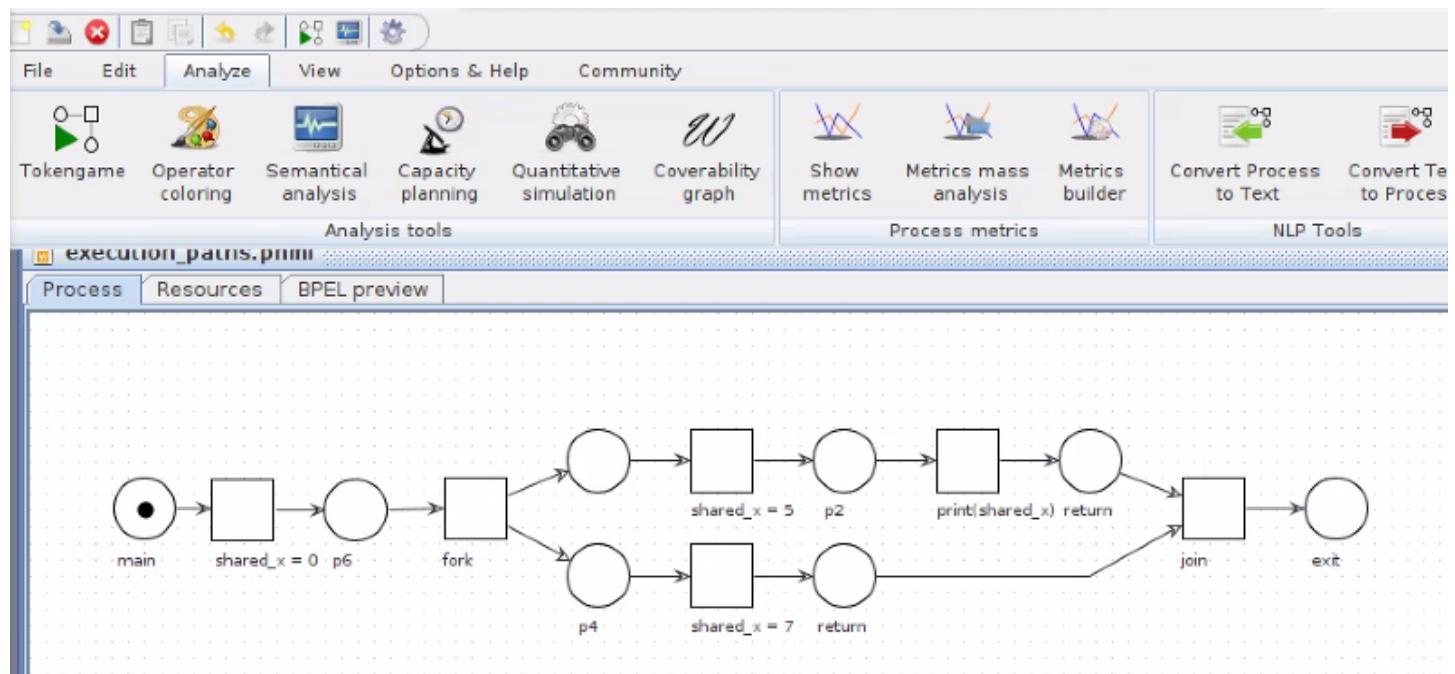
Las redes de petri modelan con estados (puntitos)

Además tienen concurrencia (la ejecución de un sistema es naturalmente concurrente) donde cada estado es independiente de otro que podría estar en otro sistema (estado distribuido). Naturalmente existe concurrencia entre humanos, por lo cual las redes de petri es muy útil ya que modela sistemas.

No hay interpretación universal (nosotros le damos la semántica)
 objetos: place (lugares)
 puntos: estados (relacionado a place, el programa llega a ese estado)
 rectángulos: transiciones (operaciones en el programa)

Ejemplo2 (actividad 13 for count)

Se utiliza el paralelismo de datos, nos preguntamos si hay casos que puedan romper mi solución.
 Las redes de Petri no es tan práctica en este temas porque tendríamos que hacer redundancia (creación de hilos).



Patrones básicos

Patrones que se crean con mecanismo de control de concurrencia. (semáforos y mutex)

1. aviso (notificar - signaling)

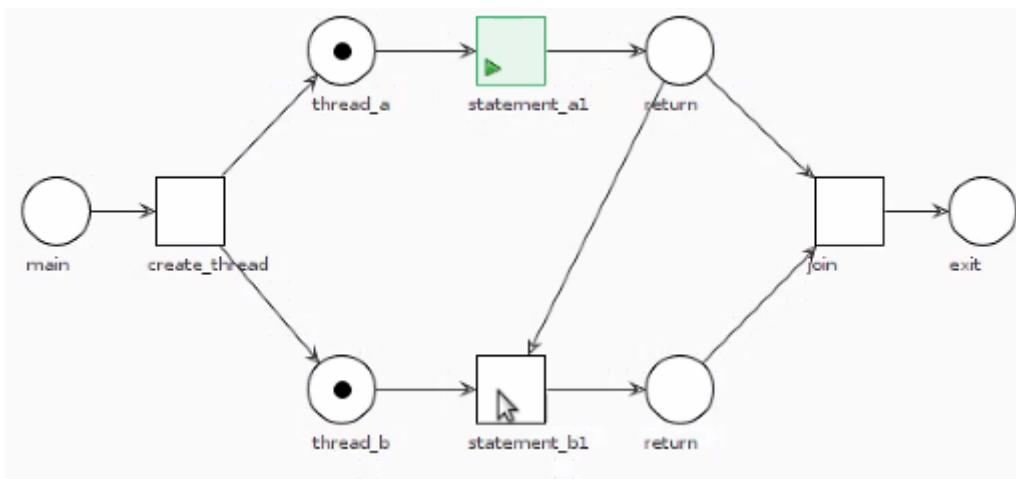
Un hilo habla directo con otro hilo (sin un buffer), es distinto al patrón productor consumidor.

Pongo un semáforo donde tengo que esperar (wait en 0). y envío el aviso desde donde debo hacerlo (signal).

`wait(1)` sirve para exclusión mutua. nunca debe superar el valor 1.

SEM10-B para practicar:

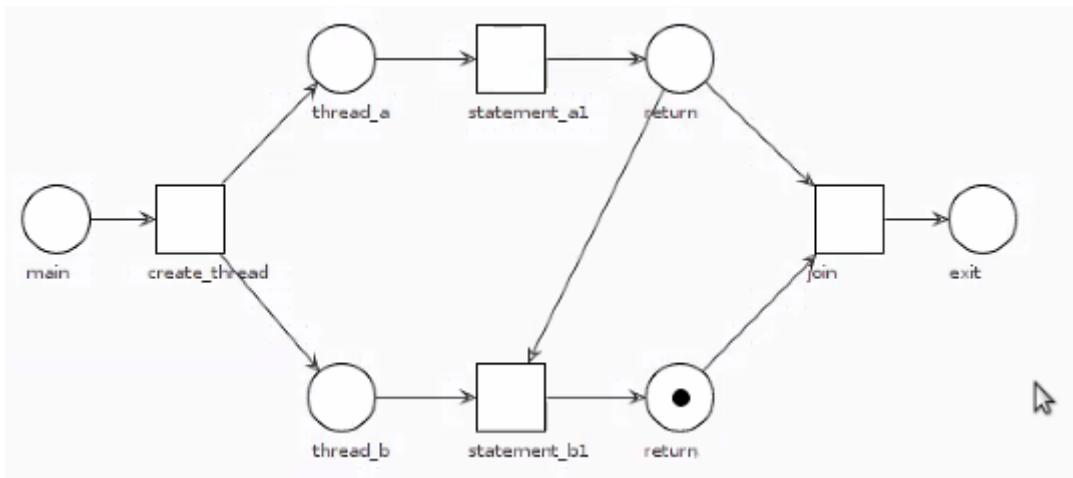
actividad 16 y 17



notar que no se puede ejecutar b1 porque tiene que esperar (Depende) de a1.

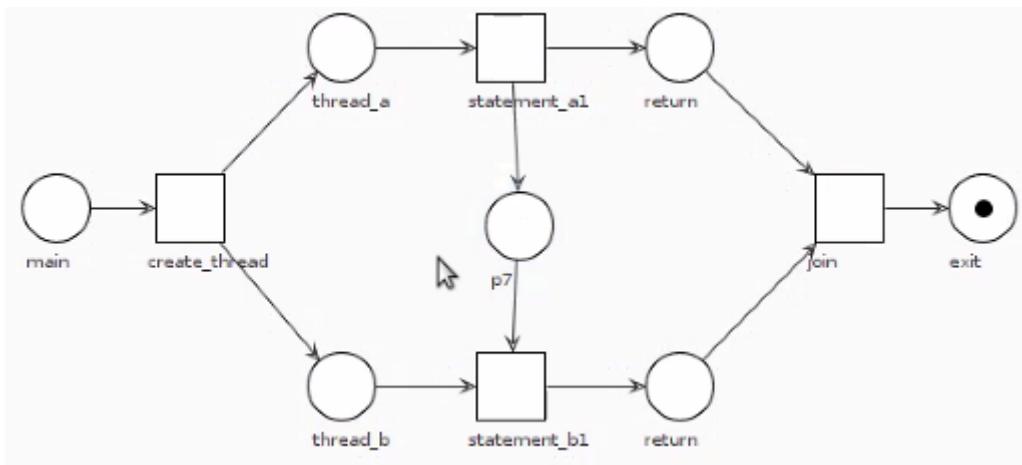
si hay dos salidas en transición entonces se da una bifurcación...

si hay dos salidas en un place (redondo) entonces se unifican (cuidado)...



se da un error (bello durmiente) porque solo hay un token en vez de dos y el join no se podría hacer.

Se arregla haciendo un nuevo estado:



rendezvous

son dos avisos cruzados (son dos signaling)

ejemplo: a1 y b1 se deben ejecutar antes que a2 y b2

signaling.pseudo ×

gnaling > ≡ signaling.pseudo

```
1 procedure main:  
2     shared a1_ready := create_semaphore(0) // can  
3     create_thread(thread_a)  
4     create_thread(thread_b)  
5 end procedure  
6  
7 procedure thread_a:  
8     statement a1  
9     signal(a1_ready)  
10    end procedure  
11  
12 procedure thread_b:  
13     wait(a1_ready)  
14     statement b1  
15    end procedure  
16
```

```
1 procedure main:  
2     shared a1_ready := create_semaphore(0) ✓  
3     shared b1_ready := create_semaphore(0) ✓  
4     create_thread(thread_a)  
5     create_thread(thread_b)  
6 end procedure  
7  
8 procedure thread_a:  
9     statement a1  
10    signal(a1_ready)  
11    wait(b1_ready)  
12    statement a2  
13 end procedure  
14  
15 procedure thread_b:  
16     statement b1  
17     signal(b1_ready)  
18     wait(a1_ready)  
19     statement b2  
20 end procedure  
21
```

rendezvous.pseudo U ×

rendezvous > ≡ rendezvous.pseudo

```
1 procedure main:  
2     shared a1_ready := create_semaphore(0)  
3     shared b1_ready := create_semaphore(0)  
4     create_thread(thread_a)  
5     create_thread(thread_b)  
6 end procedure  
7  
8 procedure thread_a:  
9     statement a1  
10    signal(a1_ready)  
11    wait(b1_ready)  
12    statement a2  
13 end procedure  
14  
15 procedure thread_b:  
16     statement b1  
17     signal(b1_ready)  
18     wait(a1_ready)  
19     statement b2  
20 end procedure
```

el hilo **a** avisa al **b** y el **b** avisa al **a**

```

1  procedure main:
2    shared a1_ready := create_semaphore(0) -|
3    shared b1_ready := create_semaphore(0) -|
4    create_thread(thread_a)
5    create_thread(thread_b) | M 22
6  end procedure
7
8  procedure thread_a:
9    statement a1
10   wait(b1_ready) 22
11   signal(a1_ready)
12   statement a2
13 end procedure
14
15 procedure thread_b:
16   statement b1
17   wait(a1_ready) 22
18   signal(b1_ready)
19   statement b2
20 end procedure
21

```

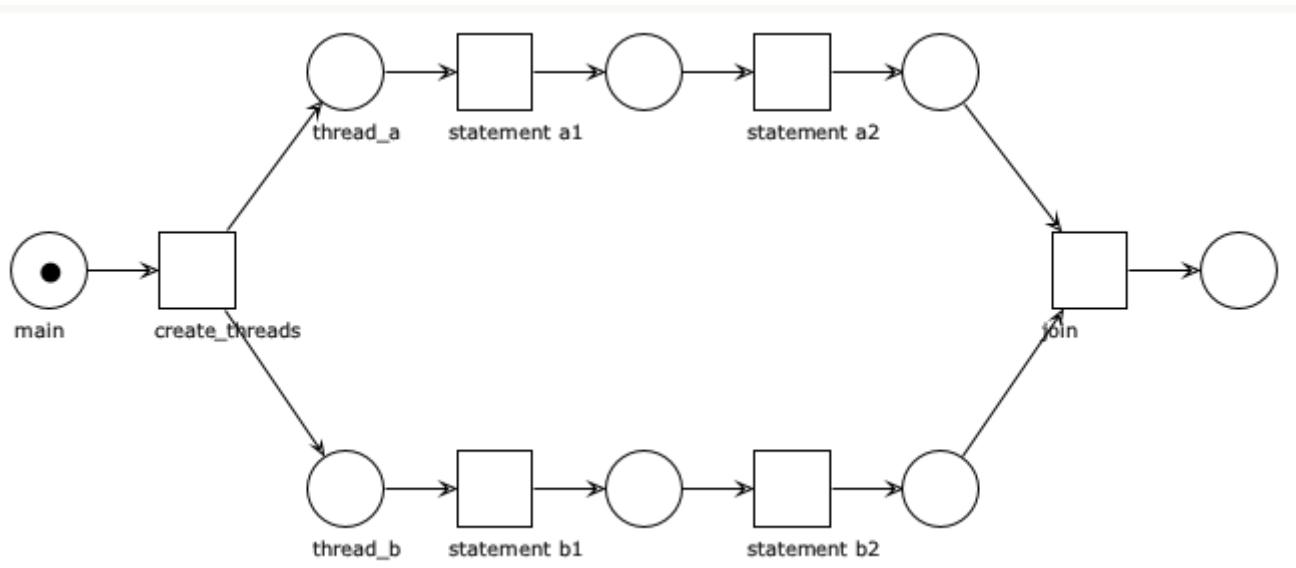
se da bloqueo mutuo (deadlock)

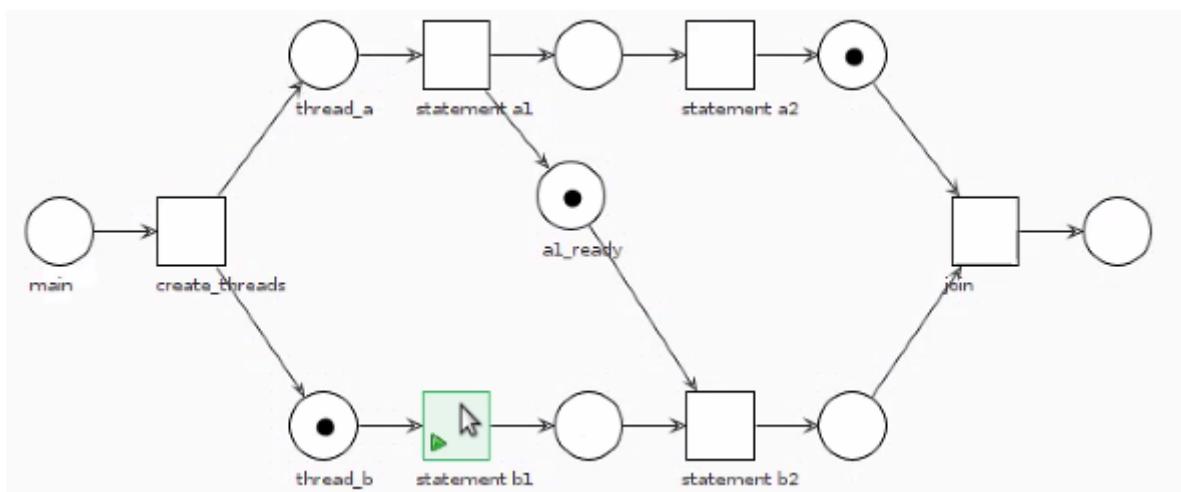
un hilo espera un recurso de otro hilo y ese hilo espera del primero.

Generalizado: cualquier orden de ejecutantes donde se pueda dibujar un grafo, donde hay una lista circular de dependencias.

Bello durmiente: espera indefinida (error más general que deadlock), subconjunto: deadlock.

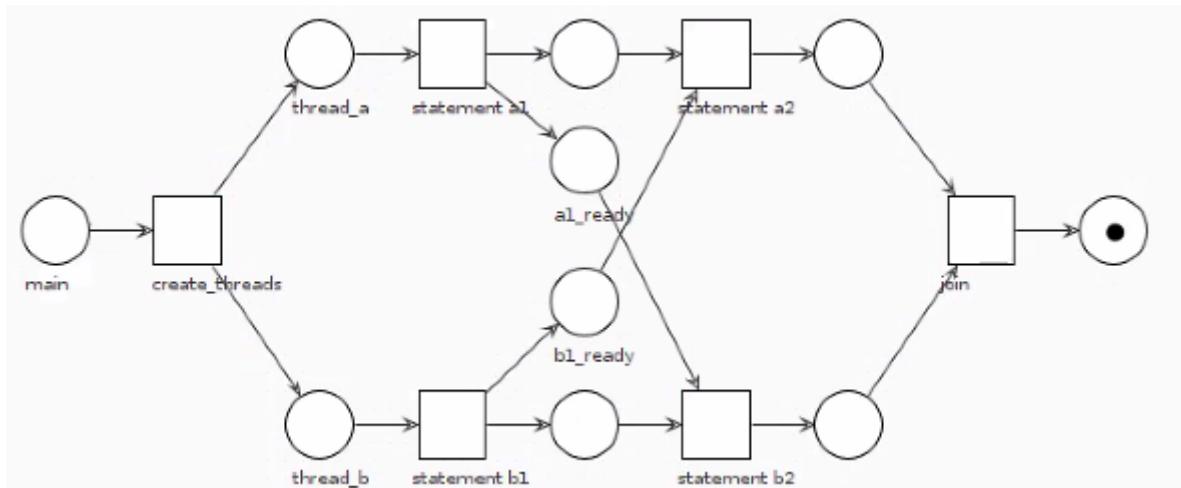
redes de petri





no debería poder

ejecutar a2 sin ejecutar b1



solución

doble señal cruzada

Un semáforo inicializado en 0 se utiliza para avisar.

Un semáforo inicializado en 1 y cuyo valor nunca supera 1 se utiliza para exclusión mútua.

Actividad 16 [sem_mutex]

```

1  procedure main:
2    shared can_access_count := create_semaphore(1) 0 ✓ 0 1
3    shared count := 0
4    create_thread(thread_a)
5    create_thread(thread_b)
6  end procedure
7
8  procedure thread_a:
9    wait(can_access_count) ✓
10   count := count + 1 /
11   signal(can_access_count) /
12 end procedure
13
14 procedure thread_b:
15   wait(can_access_count)
16   count := count + 1 ✓
17   signal(can_access_count) /
18 end procedure

```

Diferencia entre mutex y semáforo inicializado en 1:

el semáforo en 1 se usa cuando se quiere que cualquiera pueda incrementar o decrementar

el mutex tiene dueño (alguien en el puente angosto es el único que puede incrementar o decrementar)

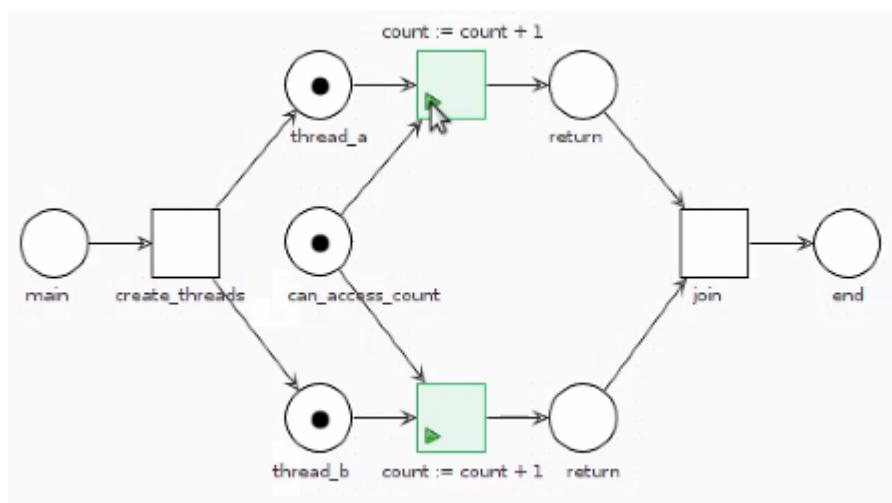
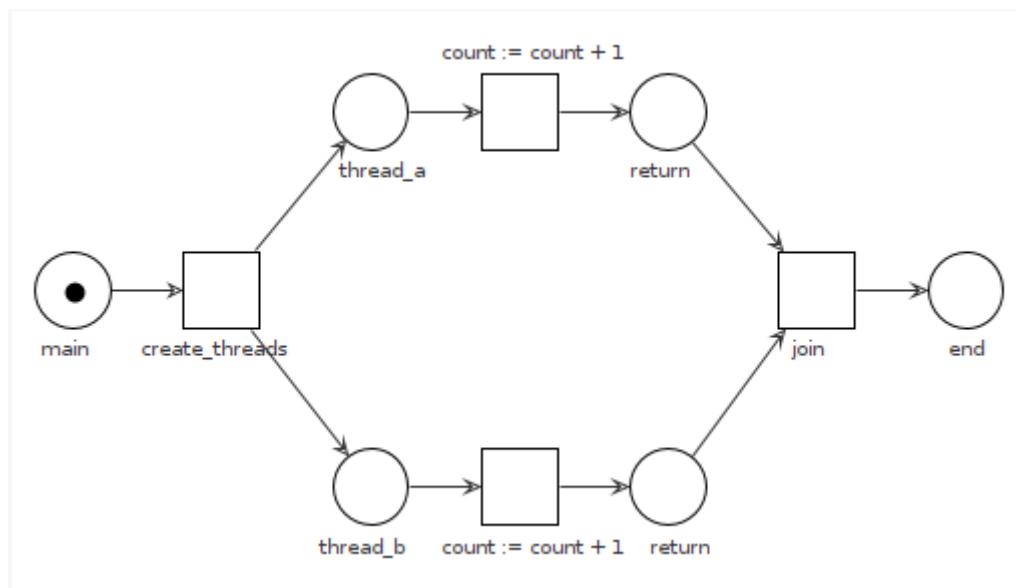
En linux tanto el semáforo como mutex trabajan sobre futex (en linux no tiene dueño) sin embargo, cualquier otro programa podría decir que: usted está incrementando el mutex sin ser el dueño.

mutex | binary semaphore |

m s({1}) s(...)
owner

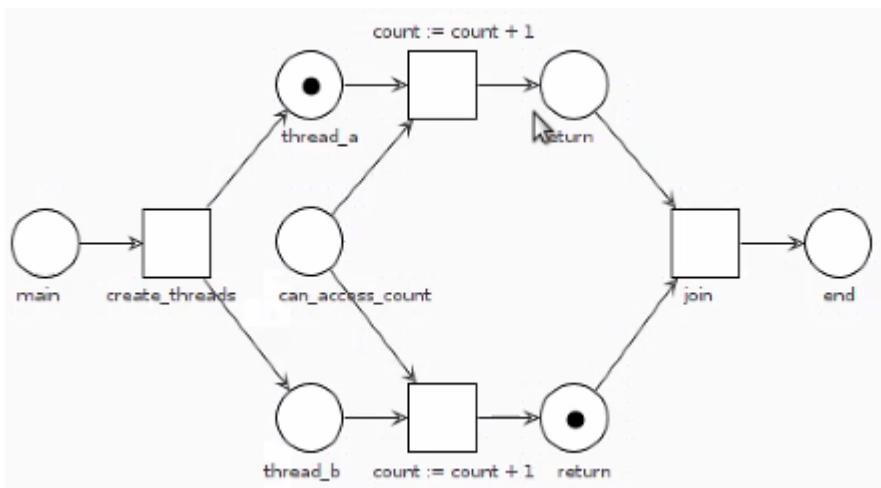
el pequeño libro de los semáforos utiliza solo el semáforo binario.

Corrija la red de Petri para que los incrementos se realicen con exclusión mutua.

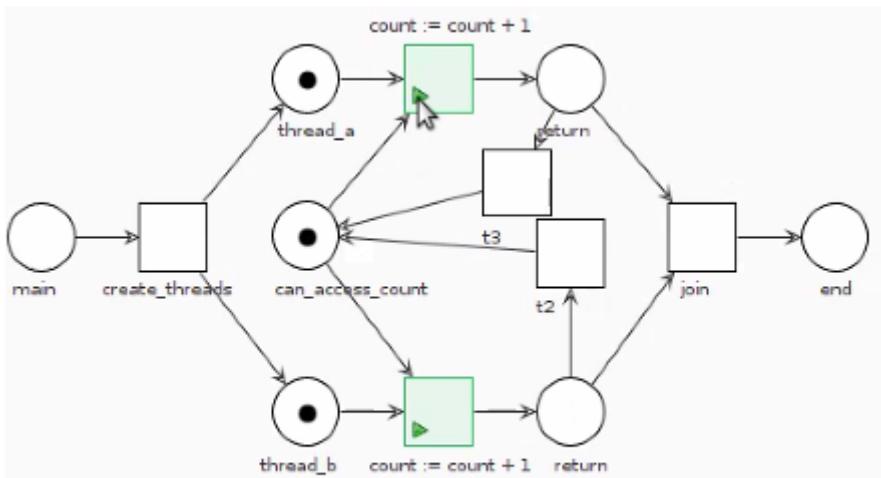


pero en realidad no

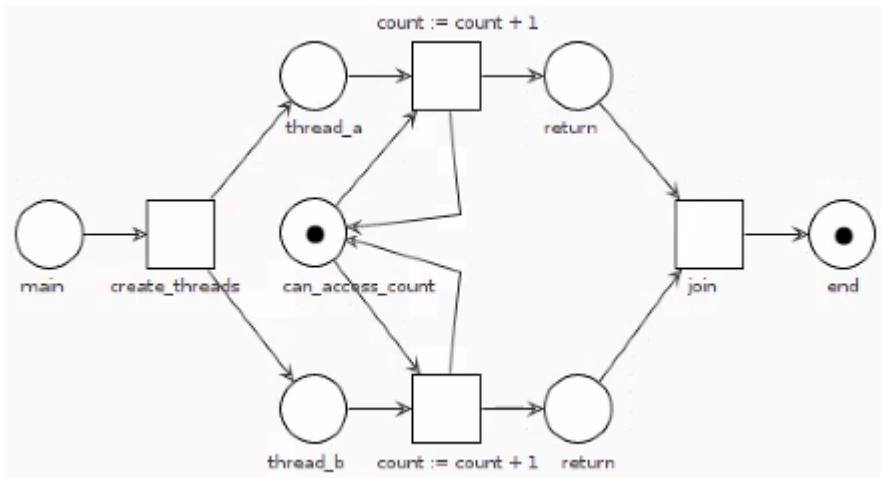
parece que hay condición de carrera



como es un place, unifica y deja sin
toquen a alguno de los dos... HACEN FALTA LOS SIGNALS



desde el place

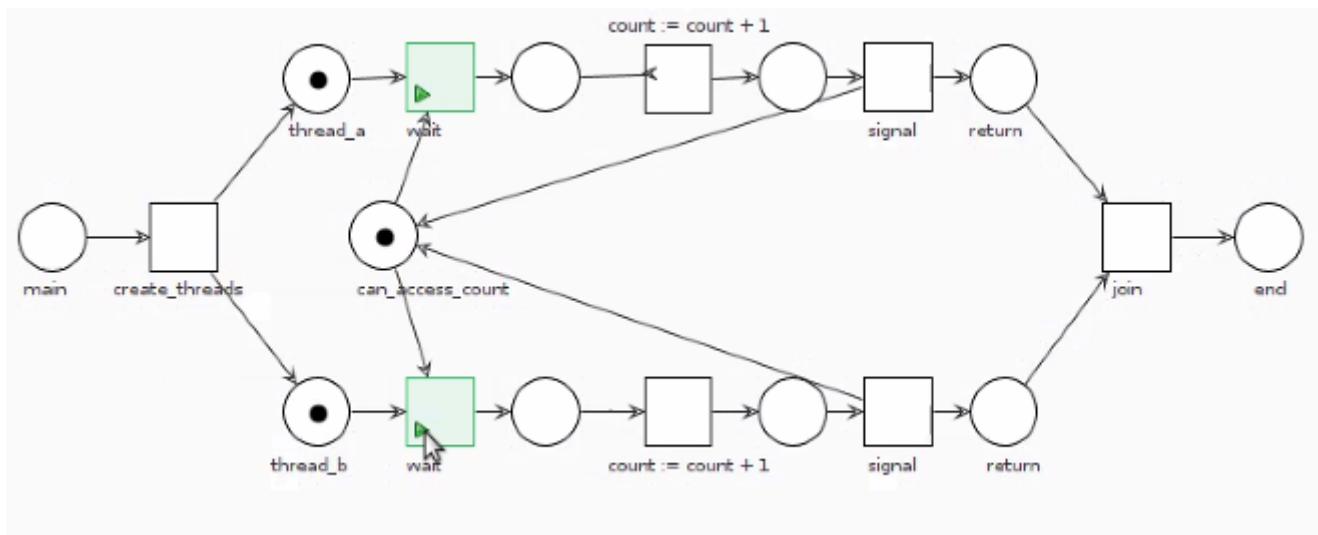


condición de carrera.

no funciona porque no debo bifurcar

desde el place

es funcional pero parece que genera



esta

solución es más real (amplia pero clara).

Actividad 17 [sem_mutex_sym]

```

1  procedure main:
2      shared can_access_count := create_semaphore(1) 0 -1 -2 -1 0
3      shared count := 0
4      shared constant thread_count = read_integer()
5      create_threads(thread_count, secondary)
6  end procedure
7
8  procedure secondary:
9      // Critical section
10     wait(can_access_count) 2
11     count := count + 1 -
12     signal(can_access_count)
13 end procedure
14

```

Patrón multiplex: varios hilos creados pero no todos deben trabajar. Ej: muchas ventanas en el banco pero no todos los funcionarios deben trabajar. W hilos creados, n hilos deben trabajar.

Permita un límite superior de n threads ejecutando la sección crítica. A este patrón se le llama multiplex y es útil para problemas donde se tienen distintos entes trabajando al mismo tiempo pero con máximo de capacidad.

Por ejemplo, la cantidad de cajeros atendiendo en las ventanillas de un banco o de clientes patinando en la pista del salón de patines. En este último caso, si la capacidad de la pista se agota, algunos clientes tendrán que esperar afuera, y apenas un cliente sale de la pista, otro podrá ingresar de inmediato. Use el código siguiente para reflejar este comportamiento.

```

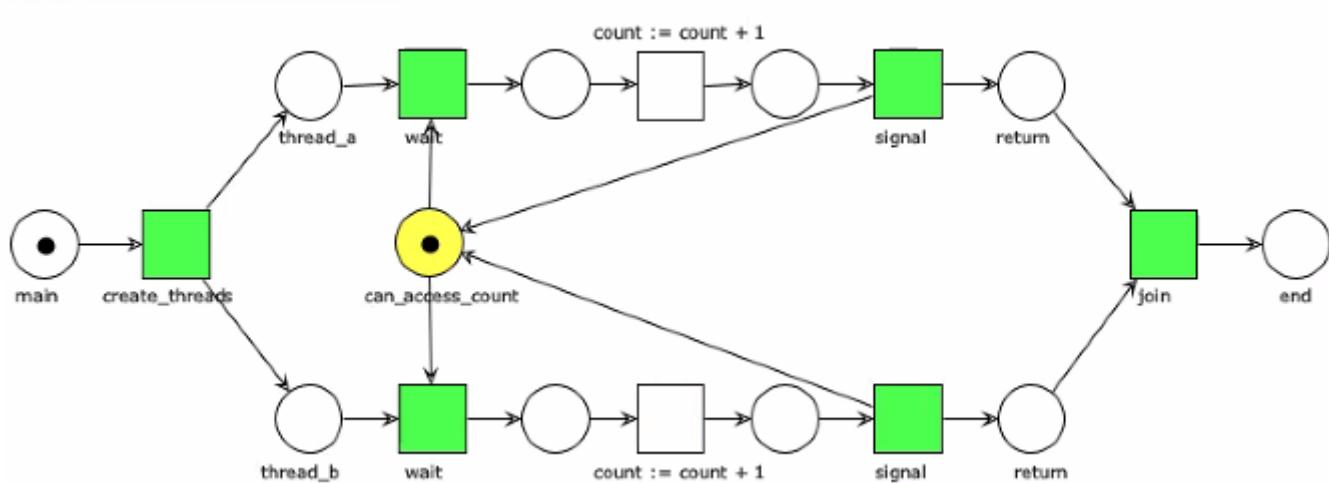
1 procedure main:
2     shared constant skater_count = read_integer() 20
3     shared constant skater_capacity = read_integer() 10
4     shared can_skate := create_semaphore(skater_capacity) 50 49 48 10 -1
5     create_threads(thread_count, secondary)
6 end procedure
7
8 procedure secondary:
9     // Bounded waiting
10    wait(can_skate) 5
11    skate() 5
12    signal(can_skate)
13 end procedure
14

```

-2 -150
-141

muchos hilos pero solo 50 pueden entrar

Sem11a



exclusión mutua en redes de petri

Patrón importante: La Barrera

Lo importante es hacer lo correcto, no importa tanto el incremento del desempeño.

1 , 2 , 3.... cuando llega a -2 se hace un if para saber si ya llegó el último

```

14 procedure secondary:
15     Statement A 0 0 0
16     // Adapt rendezvous solution here
17     wait(can_access_count)
18     count := count + 1 1 2 3
19     if count = thread_count then
20         for index := 0 to thread_count do
21             signal(barrier)
22         end for
23     end if
24     signal(can_access_count)
25     wait(barrier) -1 -2 -1 0 1 0
26     // Statement B can be only executed until
27     Statement B 0 0 0
28 end procedure
29

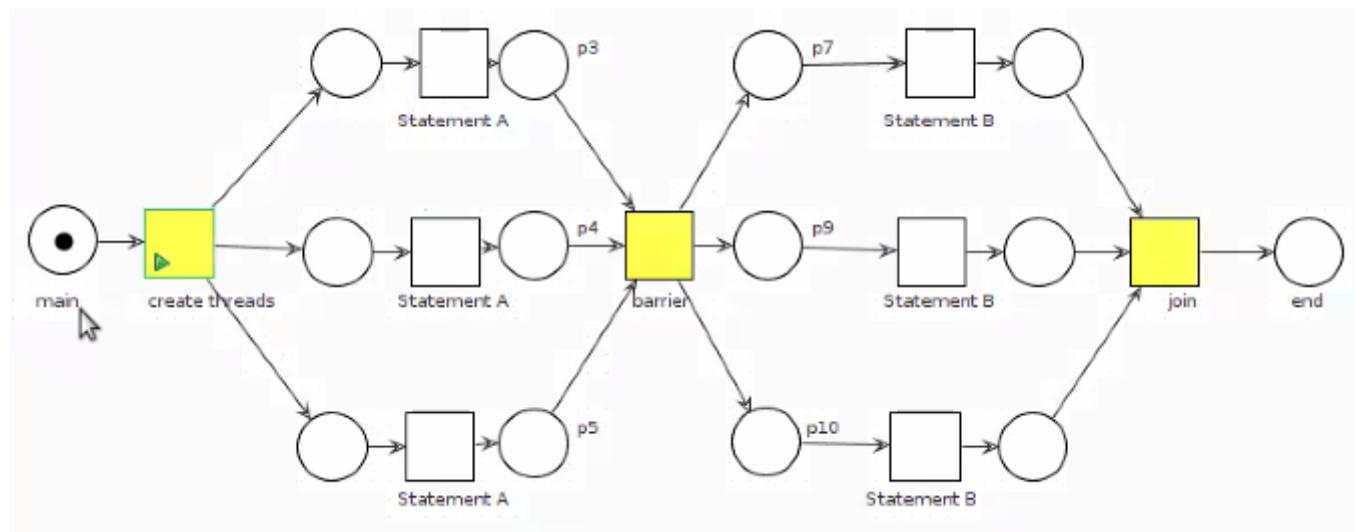
```

ejemplo con 3 hilos

es un punto de encuentro (rendezvous) donde hasta que no llegue el último, nadie avanza

Barrera en red de Petri:

El join que hemos venido haciendo en Petri es un tipo de barrera, se puede replicar ese comportamiento.

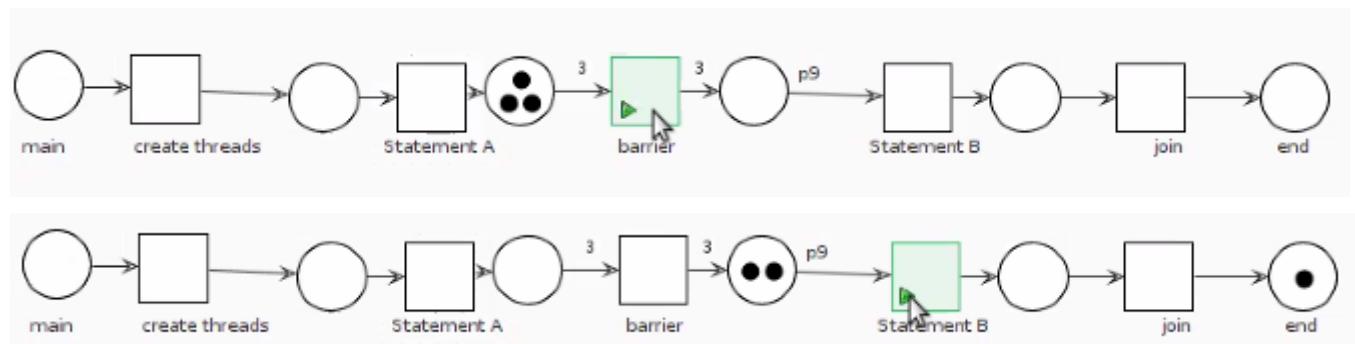


barrier es el rendezvous (la barrera).

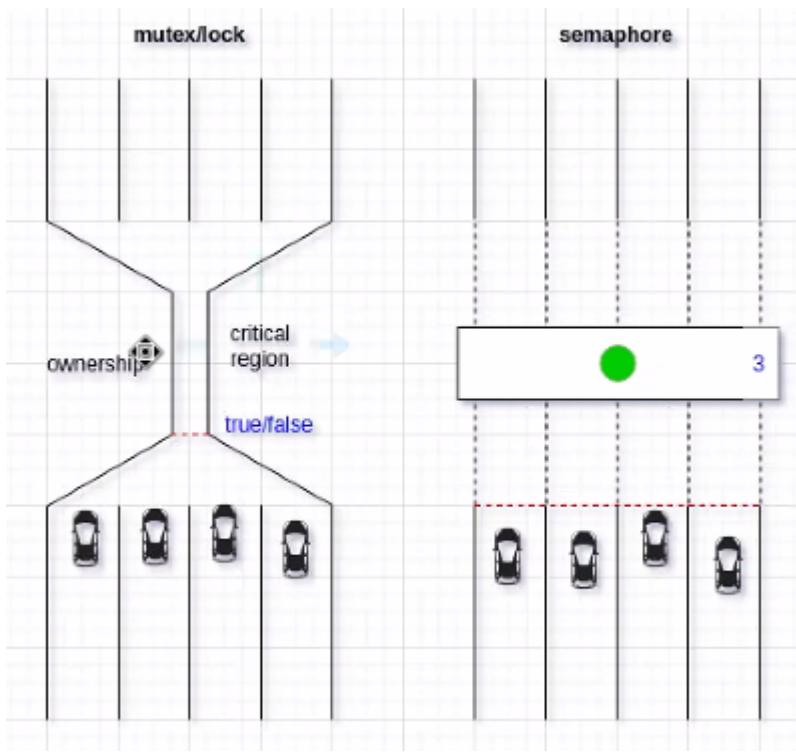
Otra forma de representar barreras en redes de petri (más sencilla):

peso de arco en **PLACE**: cuantos tokens tienen que llegar para poder pasar. (arc weight).

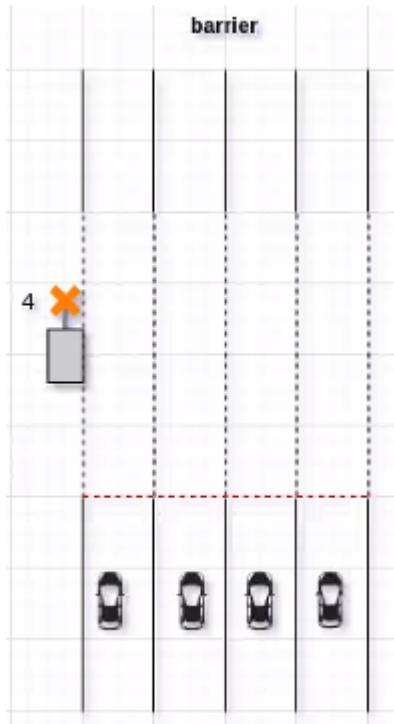
peso de arco en **Transición**: cuantos tokens quiere que salgan de ella. (arc weight).



Metáfora de la barrera

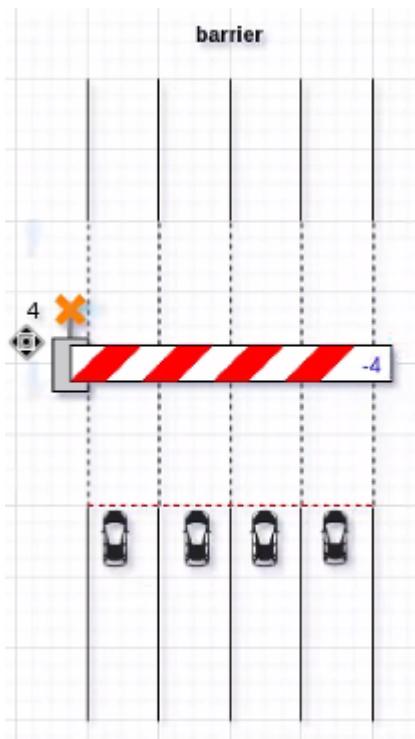


recordar metáfora del mutex y semáforo

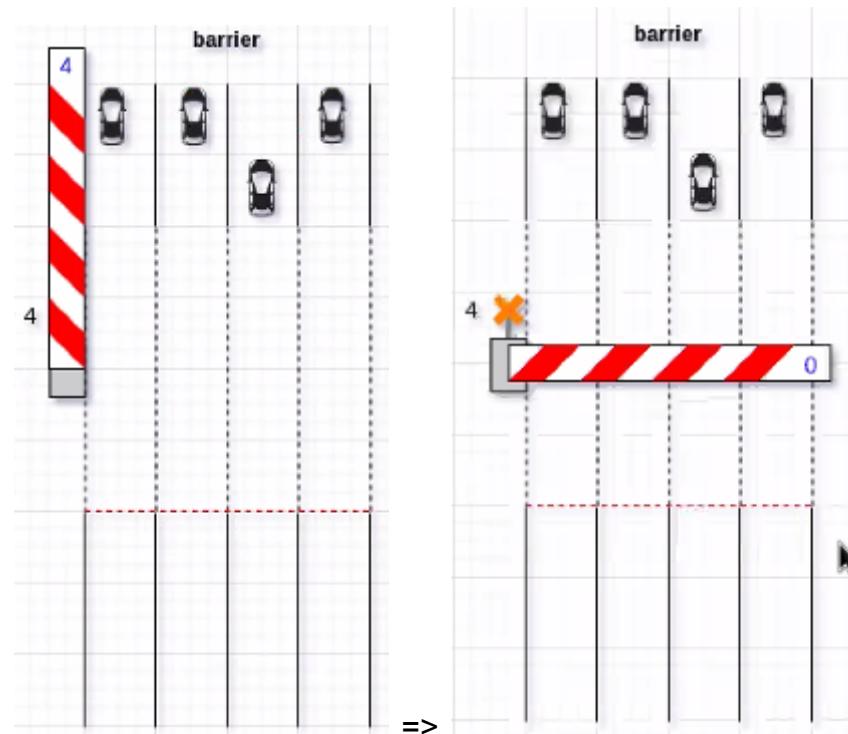


metáfora de la barrera (otro control de concurrencia): como una aguja de carros

Tiene dos enteros asociados: (límite de la barrera (la barrera se levanta cuando hayan llegados tantos carros en valor absoluto como el límite dado), cuántos hilos están esperando por ella)



ejemplo: hay 4 hilos esperando, el límite es 4 entonces la barrera se levanta. Los carros avanzarán en orden indeterminado (depende del SO).



vuelve a caer (es reutilizable)

Las barreras **NO** garantizan el orden. Solo garantizan que los hilos ejecutan una función (todos) antes de ejecutar otra.

El valor de inicialización es constante una vez creada la barrera.

proxima clase: 40 minutos de lección, 40 minutos en video.

Problema de las barreras cuando están dentro de un ciclo:

No se ejecuta en bloques como se espera... Se podría ejecutar la segunda de B antes que hayan ejecutado todos B por primera vez.

Solución: crear otra barrera entre B y A. (puede poner antes de A o después de B). Entre más barreras más orden y control de concurrencia. En ciclos suele ser usual tener 2 o 3 barreras.

Las barreras son el mecanismo de concurrencia más usadas en paralelismo de datos (tiene que ver con ciclos), se usa el 99% de las veces).

```
14 procedure secondary:  
15     while true do  
16         Statement A  
17         // Adapt rendezvous solution here  
18         wait(can_access_count)  
19         count := count + 1  
20         if count = thread_count then  
21             for index := 0 to thread_count do  
22                 signal(barrier)  
23             end for  
24         end if  
25         signal(can_access_count)  
26         wait(barrier)  
27         // Statement B can be only executed until all threads have run Statement A  
28         Statement B  
29     end while  
30 end procedure
```

(i) The Marketplace has extensions that can help you with your code files

barrera después de

B o antes que A.

Otra forma de hacerlo:

Hacer un torniquete (dos sub-barreras).



Patrón torniquete (trompo) **metáfora:**
el chofer del bus bloquea el trompo... los puede dejar pasar cuando quiera cobrar (desbloquea)...
pasa una persona y esa persona le da el paso al siguiente. Detiene a todos y apenas pasa uno: deja
pasar a todos.

Es más elegante que hacer el for de la barrera.

decremente el semáforo y luego incremente.

Si pasa un hilo deja pasar al que viene detrás de él

```

14 procedure secondary:
15   while true do
16     Statement A
17     // Adapt rendezvous solution here
18     wait(can_access_count)
19     count := count + 1 1 2 3
20     if count = thread_count then
21       signal(barrier)
22     end if
23     signal(can_access_count)
24   Z 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
25   signal(barrier)
26   // Statement B can be only executed
27   Statement B
28 end while
29 end procedure

```

wait y signal de la 24 y 25.

Usar dos barreras en un solo ciclo:

```

14 procedure secondary:
15   while true do
16     Statement A
17
18     // Adapt rendezvous solution here
19     wait(can_access_count)
20     count := count + 1
21     if count = thread_count then
22       signal(barrier)
23       count := 0
24     end if
25     signal(can_access_count)
26     wait(barrier)
27     signal(barrier)
28
29     // Statement B can be only executed until :
30     Statement B
31
32     // Adapt rendezvous solution here
33     wait(can_access_count)
34     count := count + 1
35     if count = thread_count then
36       signal(barrier)
37       count := 0
38     end if
39     signal(can_access_count)
40     wait(barrier)
41     signal(barrier)
42
43 end while

```

se debe tener 2 semáforos (una barrera es

independiente de la otra)...

Hacer que funcione de manera artesanal para ese ciclo:

Haciendo una super barrera de 2 fases (con dos torniquetes-tornillos-trompos-turnstile).
dos trompos que están conectados turnstile1 y turnstile2.

```
15 procedure secondary:  
16   while true do  
17     Statement A 0 0 0 0  
18  
19   // Adapt rendezvous solution here  
20   ⚡ wait(can_access_count)  
21     count := count + 1 1 2 2 1  
22     if count = thread_count then  
23       signal(turnstile1)  
24     end if  
25   ⚡ signal(can_access_count)  
26   ⚡ wait(turnstile1) 1 - 2 1 0 1 0 1  
27   ⚡ signal(turnstile1) 1  
28  
29   // Statement B can be only executed until  
30   Statement B 0 0 0  
31  
32   // Adapt rendezvous solution here  
33   ⚡ wait(can_access_count)  
34     count := count - 1 2 1 0  
35     if count = 0 then  
36       signal(turnstile2)  
37     end if  
38   ⚡ signal(can_access_count)  
39   ⚡ wait(turnstile2) - 1 - 2 - 1 0  
40   ⚡ signal(turnstile2)  
41  
42   end while  
43 end procedure
```

problema: el turnstile termina en 1 entonces 0

podría hacer la ejecución B antes que todos los demás....

Solución:

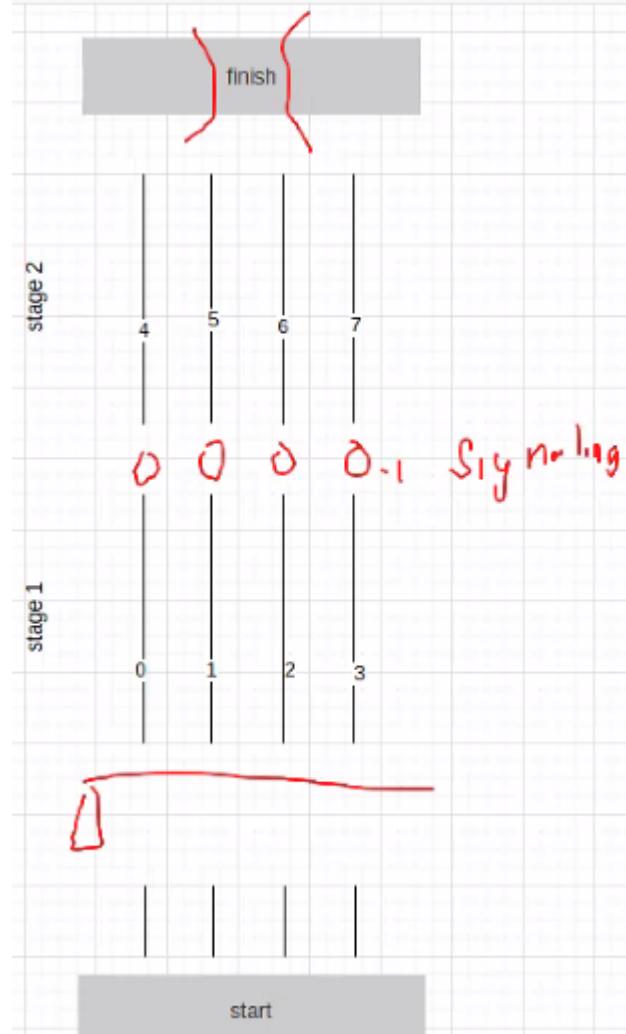
```

15 procedure secondary:
16   while true do
17     Statement A
18
19     // Adapt rendezvous solution here
20     wait(can_access_count)
21     count := count + 1
22     if count = thread_count then
23       | wait(turnstile2)
24       | signal(turnstile1)
25     end if
26     signal(can_access_count)
27     wait(turnstile1)
28     signal(turnstile1)           I
29
30   // Statement B can be only executed
31   Statement B
32
33   // Adapt rendezvous solution here
34   wait(can_access_count)
35   count := count - 1
36   if count = 0 then
37     | wait(turnstile1)
38     | signal(turnstile2)
39   end if
40   signal(can_access_count)
41   wait(turnstile2)
42   signal(turnstile2)
43
44 end while
45 end procedure

```

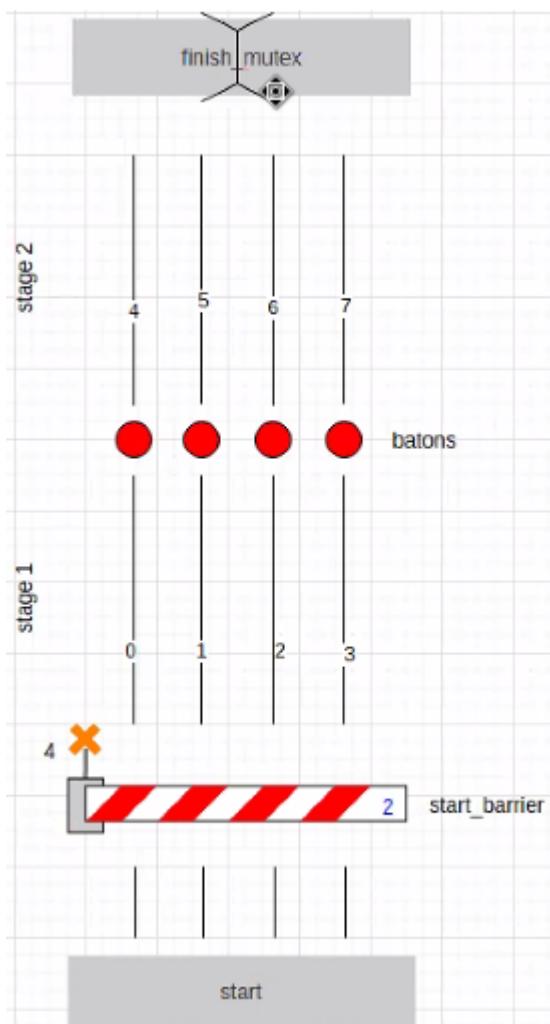
In 23 y In 37... el ultimo del trompo1 bloquea al trompo 2 y viceversa... el trompo1 se inicializa en 0 y el trompo2 se inicializa en 1 para que funcione el primer caso.

ejemplo: Carrera de relevos (barrera)



diseño con dibujos (no es tan formal):
mutex.

barrera, signaling y



barrera de inicio, estafeta (baton) y mutex al como la meta

En el pseudocódigo es bueno poner comentarios a alto nivel.

```

25  procedure run_stage2(team_number):
26      // Wait until the peer relay the baton
ejemplo: 27      wait(baton[team_number])

```

Sem11-b

Concurrencia declarativa (OpenMP)

restrict: marca que ponen los desarrolladores para indicar que solo se va acceder por un lugar(un puntero) a una dirección de memoria.

git stash:

toma los cambios que no están en commits y los guarda en una pila

git stash pop: saca de la pila los cambios (los retorna al código).

pthread es imperativo (contrapuesto a declarativo)

imperativo: se tiene control, se crea tanto código como se desee; desventaja: tener que controlar muchos detalles al intentar arreglar algo (mucho tiempo y esfuerzo).

Concurrencia compartida declarativa (OpenMP)

cedo control para con menos instrucciones, haga lo mismo; la computadora decide más cosas.

Enfoque: decir qué quiero lo que se haga y la compu se encarga (se crea más rápido, con menos líneas).

La computación tiende a ir hacia lo más declarativo.

OpenMP: desarrollada por empresas y para el paralelismo de datos. No sirve para semáforos, para mutex o barreras sí. Es una especificación para C, C++ y Fortran (importante para HPC incremento del desempeño). Documento: [Specifications - OpenMP](#) (estándar, acuerdo)

Implementado en compiladores que se apegan a esa especificación. reference guide es un resumen o índice de lo que ofrece la tecnología: [OpenMPRefCard-5.1-web.pdf](#)

zona importante: azul (biblioteca) y amarillo (directivas)

si ponemos algo con `#` el que lo recibe es el preprocesador... Entonces, las directivas son las instrucciones para el preprocesador (las que comienzan con #).

pragma:

es una directiva... son parámetros que enviamos a **un compilador en particular**, es decir, una forma de comunicarse con un compilador. ejemplo: desactivar un warning.

qué es pragma con OpenMP.

Se debe habilitar para el compilador: con `-fopenmp` (para gcc y compiladores con la misma interfaz como clang)

es con `-f` porque está instrumentalizando el código (inyectando código)... esto es declarativo.

pragma (directiva) omp (cual directiva) parallel (instrucción a ejecutar)

Cuando se usan bloques, se deben poner así:

```
{  
    instrucción  
}
```

Es un **bloque estructurado** porque tiene restricciones, **no puede tener return** y algunas veces sin **break**.

OpenMP lo que hace cuando instrumentaliza crea una región paralela que consta de la parte que está dentro del bloque estructurado.

Queremos que (**parallel**) cree hilos. Utiliza por defecto la cantidad de CPUS disponible donde se está corriendo el código. Luego de la región paralela, hace join y continúa ejecutando el hilo principal después del bloque. Nota: el hilo principal es el encargado de crear y destruir (join) los hilos secundarios.

```
1 #include <iostream>  
2  
3 int main() {  
4     #pragma omp parallel  
5     {  
6         std::cout << "Hello from main thread\n";  
7         std::cout << "jiji\n";  
8     }  
9 }
```

Dentro del bloque pueden haber subrutinas con parámetros

Puede haber indeterminismo a la hora de imprimir con un << porque tiene un mutex, pero varios hilos pueden estar accediendo (aunque no es condición de carrera) e imprimir algo antes o después de lo que se espera, se arregla poniendo un mutex propio:

```
1 #include <iostream>
2
3 int main() {
4     #pragma omp parallel
5     {
6         #pragma omp critical
7         {
8             std::cout << "Hello from secondary thread" << std::endl;
9         }
10    }
11    std::cout << "jiji\n";
12 }
13
```

critical crea un mutex.

Si utilizo más mutex se utiliza el mismo (uno para todos)

```
1 #include <iostream>
2
3 int main() {
4     #pragma omp parallel
5     {
6         #pragma omp critical()
7         std::cout << "Hello from secondary thread" << std::endl;
8
9         #pragma omp critical
10        std::cout << "Hello from secondary thread" << std::endl;
11
12         #pragma omp critical
13         std::cout << "Hello from secondary thread" << std::endl;
14     }
15 }
```

pero se puede enviar una cláusula con el nombre del mutex, lo mismo para la barrera:

```
3 int main() {
4     #pragma omp parallel
5     {
6         #pragma omp critical(mutex1)
7         std::cout << "Hello from secondary thread" << std::endl;
8
9         #pragma omp critical(mutex2)
10        std::cout << "Hello from secondary thread" << std::endl;
11
12         #pragma omp critical(mutex1)
13         std::cout << "Hello from secondary thread" << std::endl;
14     }
15 }
```

Problema con openMP: doble instrumentalización por los **sanitizers**... tiene problemas con valgrind. Se debe recompilar.

Con la parte de funciones de biblioteca (parte azul-gris) podemos obtener valores, como el rank, thread number etc...

OpenMPRefCard-5.1-web.pdf

the value of the first element of the <i>nthreads-var</i> ICV of the current task to <i>num_threads</i> .	void omp_set_num_threads (int num_threads);	int omp_get_level (void);
Fortran	C/C++	Fortran
void omp_set_num_threads (int num_threads);	subroutine omp_set_num_threads (nested) logical <i>nested</i>	integer function omp_get_level ()
Fortran	C/C++	Fortran
omp_get_num_threads [3.2.2] [3.2.2]	• • omp_get_nested [3.2.10] [3.2.11]	omp_get_ancestor_thread_num [3.2.18] [3.2.19]
Returns the number of threads in the current team. The binding region for an omp_get_num_threads region is the innermost enclosing parallel region. If called from the sequential part of a program, this routine returns 1.	Returns whether nested parallelism is enabled or disabled. ICV: <i>max-active-levels-var</i>	Returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.
Fortran	C/C++	Fortran
int omp_get_num_threads (void);	int omp_get_nested (void);	int omp_get_ancestor_thread_num (int level);
Fortran	C/C++	Fortran
integer function omp_get_num_threads ()	logical function omp_get_nested ()	integer function omp_get_ancestor_thread_num (level) integer level
omp_get_max_threads [3.2.3] [3.2.3]	omp_set_schedule [3.2.11] [3.2.12]	omp_get_team_size [3.2.19] [3.2.20]
Returns an upper bound on the number of threads that could be used to form a new team if a parallel construct without a num_threads clause were encountered after execution returns from this routine.	Affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the <i>run-sched-var</i> ICV.	Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.
Fortran	C/C++	Fortran
int omp_get_max_threads (void);	void omp_set_schedule(omp_sched_t kind, int chunk_size);	int omp_get_team_size (int level);
Fortran	Fortran	Fortran
integer function omp_get_max_threads ()	subroutine omp_set_schedule (kind, chunk_size) integer (kind=omp_sched_kind) kind integer chunk_size	integer function omp_get_team_size (level) integer level
omp_get_thread_num [3.2.4] [3.2.4]	See omp_get_schedule for kind.	omp_get_active_level [3.2.20] [3.2.21]
Returns the thread number of the calling thread, within the current team.		Returns the number of active, nested parallel regions on the device enclosing the task containing the call. ICV: <i>active-level-var</i>
Fortran	C/C++	Fortran
int omp_get_thread_num (void);	void omp_get_schedule (omp_sched_t *kind, int *chunk_size);	int omp_get_active_level (void);
Fortran	Fortran	Fortran
integer function omp_get_thread_num ()	subroutine omp_get_schedule (kind, chunk_size) integer (kind=omp_sched_kind) kind integer chunk_size	integer function omp_get_active_level ()
omp_in_parallel [3.2.5] [3.2.6]	kind for omp_set_schedule and omp_get_schedule is an implementation-defined schedule or:	Thread affinity routines
Returns true if the <i>active-levels-var</i> ICV is greater than zero; otherwise it returns false.	omp_sched_static omp_sched_dynamic omp_sched_guided omp_sched_auto	omp_get_proc_bind [3.3.1] [3.2.23]
5:36 / 11:29	Use + or operators (C/C++) or the + operator (Fortran) to combine the <i>kinds</i> with the modifier.	Returns the thread affinity policy to be used for the subsequent nested proc_bind clause.
See omp_in_parallel function .		omp_proc_bind

omp_get_num_threads me dice la cantidad que otorgué para la zona paralela.

Para que no haya errores, se debe incluir la biblioteca: **omp.h** para C++ y C.

puedo preguntar por el max_threads disponibles en máquina (CPUS)

Sem11-b

```

16 #pragma omp parallel num_threads(2)
17 {
18     {
19         if (omp_get_thread_num() == 0) {
20             produce();
21         } else {
22             consume();
23         }
24     }
25 }
```

Si ocupo semáforos, podría utilizar los de algunas librerías de C++, los del SO etc...

Recordar que openMP ayuda a paralelizar (paralelismo de datos, aumento del desempeño) lo que está relacionado a ciclos o recursividad.

parallel for tiene que estar seguido de un for.

los hilos se reparten las iteraciones (uno hace una parte, otro otra y así).

```
15 #pragma omp parallel for num_threads(thread_count)
16 for (int iteration = 0; iteration < iteration_count; ++iteration) {
17     #pragma omp critical
18     {
19         std::cout << omp_get_thread_num() << "/" << omp_get_num_threads()
20         << ": iteration " << iteration << "/" << iteration_count <<
21         std::endl;
22 }
```

ejemplo de parallel for, cada hilo hace una parte:

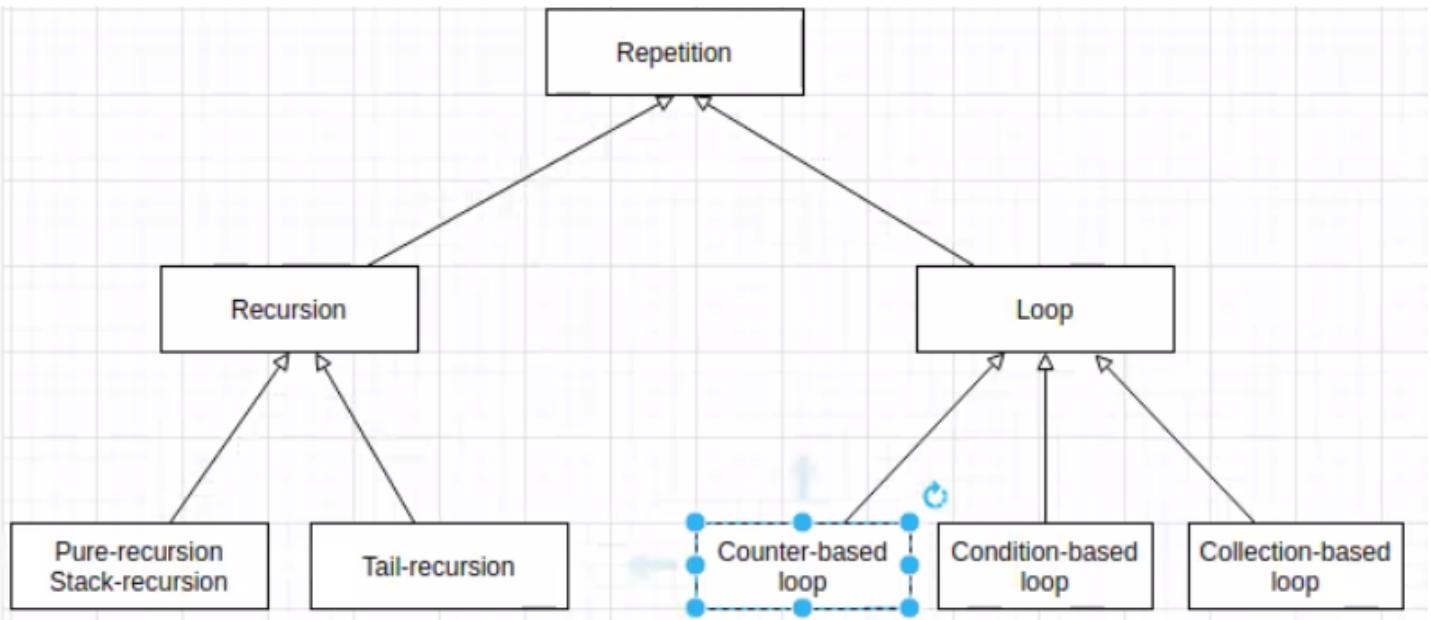
```
0/3: iteration 0/10
0/3: iteration 1/10
1/3: iteration 4/10
1/3: iteration 5/10
1/3: iteration 6/10
0/3: iteration 2/10
0/3: iteration 3/10
2/3: iteration 7/10
2/3: iteration 8/10
2/3: iteration 9/10
```

... se debe ser cuidadoso con el uso del código ya que una palabra podría cambiar muchas cosas.

Restricciones del parallel for:

tiene que ponerse antes de un ciclo for (no puede ser con while u otro)

tiene que ser un ciclo por contador:



tres tipos de ciclos (en mundo imperativo de la programa):

- **ciclo por contador:** se sabe cuántas iteraciones van a ocurrir de antemano
- **ciclo condicionado:** NO se sabe cuántas iteraciones van a ocurrir de antemano
- **ciclo por colección:** itera a lo largo de una estructura de datos (se puede saber o no)

OpenMP requiere que sea un **ciclo por contador**

Ocupa saber la cantidad de iteraciones porque utiliza un mapeo estático (por bloque) y por eso exige un ciclo por contador.
de nuevo, no pueden haber break (porque debe ser un bloque estructurado).

default(None) // cláusula

dice: si los desarrolladores no dicen qué irán en datos compartidos o privados, usted asúmalo (por void* data a la subrutina que ejecuta el hilo).

ej:

```
11 int iteration_count = thread_count;
12 if (argc >= 3) {
13     iteration_count = atoi(argv[2]);
14 }
15
16 #pragma omp parallel for num_threads(thread_count) default(None)
17 for ...:
18     pthread_create(..., omp_sub1, ...);
19
20 // for (int iteration = 0; iteration < iteration_count; ++iteration) {
21 //     #pragma omp critical(stdout)
22 //     std::cout << omp_get_thread_num() << '/' << omp_get_num_threads()
23 //         << ": iteration " << iteration << '/' << iteration_count << std::endl;
24 // }
25 }
26
27 void* omp_sub1(void* data) {
28     for (int iteration = 0; iteration < iteration_count; ++iteration) {
29         #pragma omp critical(stdout)
30         std::cout << omp_get_thread_num() << '/' << omp_get_num_threads()
31             << ": iteration " << iteration << '/' << iteration_count << std::endl;
32     }
33 }
```

iteration count no se le está pasando...

puedo decidir si se lo paso como privado o como compartido

```

1 #include <omp.h>
2
3 #include <iostream>
4
5 int main(int argc, char* argv[]) {
6     int thread_count = omp_get_max_threads();
7     if (argc >= 2) {
8         thread_count = atoi(argv[1]);
9     }
10
11    int iteration_count = thread_count;
12    if (argc >= 3) {
13        iteration_count = atoi(argv[2]);
14    }
15
16    #pragma omp parallel for num threads(thread count) default(private) shared()
17    for (int iteration = 0; iteration < iteration_count; ++iteration) {
18        #pragma omp critical(stdout)
19        std::cout << omp_get_thread_num() << '/' << omp_get_num_threads()
20        | << ": iteration " << iteration << '/' << iteration_count << std::endl;
21    }
22
23 }
```

pasamos `iteration_count` como `shared()` para no hacer muchas copias de un solo dato `cout` debemos pasarlo compartido porque dentro de él hay un archivo que tienen cosas compartidos (ningún archivo se deja copiar en general).

Es mejor nunca dejar libremente el `default(private)`, se debe utilizar siempre y decidir si es privada o compartida. Importancia: permite análisis de datos para evitar condición de carrera en pocas líneas de código.

```

16    #pragma omp parallel for num threads(thread count) default(private) shared(iteration_count, std::cout)
17    for (int iteration = 0; iteration < iteration_count; ++iteration) {
18        #pragma omp critical(stdout)
19        std::cout << omp_get_thread_num() << '/' << omp_get_num_threads()
20        | << ": iteration " << iteration << '/' << ++iteration_count << std::endl;
21    }
```

por ejemplo si `iteration_count` se modificara, se puede saber rápidamente qué se puede cambiar para evitar la condición de carrera.

`cout` ya está protegido por mutex internamente.

caracter de escape: \

hace que se ignore lo que sigue, por ejemplo cambio de línea
hace que el linter no pegue gritos por ejemplo

...

```

15
16     while true {
17         #pragma omp parallel for num_threads(thread_count) \
18             default(None) shared(iteration_count, std::cout)
19         for (int iteration = 0; iteration < iteration_count; ++iteration) {
20             #pragma omp critical(stdout)
21             std::cout << "stage 1: " << omp_get_thread_num() << '/'
22             << omp_get_num_threads() << ": iteration " << iteration << '/'
23             << iteration_count << std::endl;
24         }
25
26         std::cout << std::endl;
27
28         #pragma omp parallel for num_threads(thread_count) \
29             default(None) shared(iteration_count, std::cout)
30         for (int iteration = 0; iteration < iteration_count; ++iteration) {
31             #pragma omp critical(stdout)
32             std::cout << "stage 2: " << omp_get_thread_num() << '/'
33             << omp_get_num_threads() << ": iteration " << iteration << '/'
34             << iteration_count << std::endl;
35         }
36
37         std::cout << std::endl;
38
39         #pragma omp parallel for num_threads(thread_count) \
40             default(None) shared(iteration_count, std::cout)
41         for (int iteration = 0; iteration < iteration_count; ++iteration) {
42             #pragma omp critical(stdout)
43             std::cout << "stage 3: " << omp_get_thread_num() << '/'
44             << omp_get_num_threads() << ": iteration " << iteration << '/'
45             << iteration_count << std::endl;
46         }
47     }
48 }
49

```

si hacemos varias regiones paralelas, los hilos se crean y destruyen y eso es mucho gasto de recursos.

Para optimizar y gastar menos recursos:

crear antes del primer ciclo y destruir luego del tercer ciclo

omp for // significa: reparta las iteraciones.

```

several_for.cpp ⑥ ×
several_for > src > several_for.cpp > main(int, char *[])
13     iteration_count = atoi(argv[2]);
14 }
15
16 #pragma omp parallel num_threads(thread_count) \
17     default(none) shared(iteration_count, std::cout)
18
19 #pragma omp for
20 for (int iteration = 0; iteration < iteration_count; ++iteration) {
21     #pragma omp critical(stdout)
22     std::cout << "stage 1: " << omp_get_thread_num() << '/'
23     | << omp_get_num_threads() << ": iteration " << iteration << '/'
24     | << iteration_count << std::endl;
25 }
26
27 std::cout << std::endl;
28
29 #pragma omp for
30 for (int iteration = 0; iteration < iteration_count; ++iteration) {
31     #pragma omp critical(stdout)
32     std::cout << "stage 2: " << omp_get_thread_num() << '/'
33     | << omp_get_num_threads() << ": iteration " << iteration << '/'
34     | << iteration_count << std::endl;
35 }
36
37 std::cout << std::endl;
38
39 #pragma omp for
40 for (int iteration = 0; iteration < iteration_count; ++iteration) {
41     #pragma omp critical(stdout)
42     std::cout << "stage 3: " << omp_get_thread_num() << '/'
43     | << omp_get_num_threads() << ": iteration " << iteration << '/'
44     | << iteration_count << std::endl;
45 }
46
47 }
48

```

al final de una región paralela hay un join.

al final de los for (de omp for), hay una barrera implícita (se podría desactivar pero es muy útil) por omisión.

`#pragma omp barrier //` declara una barrera implícita

se puede marcar una instrucción con un `#pragma omp single`

`single //` dentro de la región paralela tengo el `thread_count...` cuando digo `single` significa que **1 de los hilos** ejecuta el código de bloque que le continúa

```

25 #pragma omp single
26 std::cout << omp_get_thread_num() << std::endl;
27 // #pragma omp barrier

```

ejemplo de `single...` escoge al azar al hilo que lo ejecutará. Los demás hilos esperarán hasta que el `single` imprima, porque **tiene una barrera implícita** (se puede desactivar pero igual es útil).

`gprof` toma el tiempo de todas las subrutinas de nuestro programa.

`perf` toma el tiempo de nuestro programa... pero podría guardar datos y hasta generar gráficos.

Descomposición y mapeo en OpenMP (schedule)

hay descomposición (se reparten las unidades de descomposición, yo las decido, la que pongo en la zona paralela)

hay mapeo (schedule) // schedule en SO es el que se encarga de tomar los hilos y ponerlos en las colas, quitarlos, darle CPUs etc.

tipos de mapeo o schedule

static / dynamic / guided

por defecto el schedule toma el static (por bloque).

schedule(static)

para pasarle otro:

#pragma omp for schedule(dynamic)

guided es también dinámico, pero crea bloques y los reparte, donde cada bloque se vuelve cada vez más pequeño. Cada vez que termina un chunk, solicita otro chunk

auto elige uno por defecto (el static).

runtime escoge uno en tiempo de ejecución, con variables de ambiente.

gcc y clang toman distinto al mapeo guided. gcc lo implementa más ingenuo, clang lo hace de manera más inteligente.

```
[doc03:schedule]$ bin/schedule 3 12
static   0 0 0 0 1 1 1 1 2 2 2 2
dynamic  0 2 1 0 0 0 2 0 2 2 0 2
guided   0 0 0 0 | 1 1 1 | 0 0 | 2 | 2 | 1 |
```

schedule

el guiado (guided) en gcc:

cuando un hilo se desocupa toma el siguiente bloque, en cada iteración le resta uno y lo divide entre el número de hilos

```
[doc03:schedule]$ bin/schedule 3 24
static   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
dynamic  1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 1 0 2 0 1
guided   8 0 0 0 0 0 0 0 | 2 2 2 2 2 2 | 1 1 1 1 | 0 0 | 2 2 | 1 | 2 |
[doc03:schedule]$
```

16/3 10/3 8/3 8/3

unidadesRestantes/thread_count

24/3 → 16/3 → 12/3 → 8/3 ...

Empieza con bloques grandes y conforme se avanza, se van haciendo más pequeños los bloques.

Ventaja: involucra ambas granularidades (fina [muchas tareas mucha interacción] y gruesa [grandes tareas pero pocas y baja interacción concurrente]) para que sea un caso “promedio” de ambas. Es decir, el mapeo guiado combina lo bueno de ambos “mundos”.

el mapeo tiene mucho grado de importancia en la eficiencia del paralelismo.

```
[doc03:schedule]$ bin/schedule 3 10
static    0 0 0 0 1 1 1 2 2 2
dynamic   2 1 1 2 2 1 2 1 2 1
guided    0 0 0 0 2 2 1 1 2 2
[doc03:schedule]$ bin/schedule 3 10 1
static,N  0 1 2 0 1 2 0 1 2 0
dynamic,N 2 0 1 0 0 2 0 2 2 0
guided,N  1 1 1 1 2 2 1 1 1 1
[doc03:schedule]$
```

guiado y dinámico por defecto el bloque es 1 pero con N cambia para el estático el bloque por defecto es cíclico

en el guiado el parámetro N significa que reparte bloques grandes y se va decrementando teniendo como mínimo a N (bloque de tamaño N como mínimo para el guiado)

Solo cambia el mínimo si no quedan más hilos (se da el residuo == 1).

runtime:

uno decide en tiempo de ejecución cuál es el mapeo. el instrumentalizador analiza el código para saber que tipo de mapeo usar.

variables ambientes:

variable HOME: echo \$HOME

variable PS1: echo \$PS1

para saber cuales variables hay: env

para definir un mapeo: OMP_SCHEDULE=static

OMP_SCHEDULE=static, 4

donde 4 es el tamaño de bloque (N).

```
runtime 1 2 0 1 0 2 1 0 2 1
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 1 0 2 1 2 0 1 2 1 1
[jhc@doc03:runtime_schedule]$ OMP_SCHEDULE=static
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 2 0 1 0 1 0 1 0 1 0
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 2 1 0 1 0 2 1 0 2
[jhc@doc03:runtime_schedule]$ export OMP_SCHEDULE=static
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 0 0 0 1 1 1 2 2 2
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 0 0 0 1 1 1 2 2 2
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 0 0 0 1 1 1 2 2 2
[jhc@doc03:runtime_schedule]$ export OMP_SCHEDULE=static,1
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 1 2 0 1 2 0 1 2 0
[jhc@doc03:runtime_schedule]$ export OMP_SCHEDULE=static,4
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 10
runtime 0 0 0 0 1 1 1 1 2 2 2
[jhc@doc03:runtime_schedule]$ bin/runtime_schedule 3 30
runtime 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 1 1 1 1 2 2 2 2 0 0 0 0 1 1
[jhc@doc03:runtime_schedule]$ export OMP_SCHEDULE=static,4
```

export para que sea general el cambio.

ejemplo.


```
#pragma omp atomic  
total_sum += my_partial_sum;
```

mutex: constructo del SO.

--

reducciones: cosas pequeñas hacen cosas grandes.
mapeo (map reduce), operaciones aritméticas...
se deriva de la programación funcional
era ineficiente hasta que existieron los multithreading.

operación reduce:
tomo un conjunto de valores y agarramos una única salida.

En **openMP:**

operación: variables donde se aplica esa reducción
reduction(+:var)
reduction(^:var)
reduction(BUSCARPorOpValidas:var)

las variables pasadas por parámetro de reduction son siempre compartidas
notación compacta, variables locales en vez de conditionally safe

--

Concurrencia compartida:

hilos que comparten el procesador, reloj del sistema etc... es fácil la comunicación.
desventaja: escalabilidad

Concurrencia distribuida

los hilos no comparten los recursos sin intervención del sistema operativo(s).

desventaja: comunicación

muchas máquinas, pueden ser simétricas (clúster) o asimétrica (malla). El clúster requiere de mucho dinero.

Tecnología MPI:

Interfaz de paso de mensajes (message passing interface)

medio declarativa media imperativa

Simétrica (clúster) y asimétrica

Sem13-a

MPI: es una especificación (su producto es un documento), todo lo de afuera que diga ser parte se tiene que apegar a ella. No viene con el compilador (como openMP).
para trabajar en clúster (tecnología de paso de mensajes)... para procesos (recursos distribuidos).
Páginas: MPI forum (está el estándar, página oficial), 2 caps importante: 3 (comunicación punto a punto (un proceso habla con otro)) y cap 6 (comunicación colectiva: un proceso habla con uno o varios, o varios con varios.)

Implementaciones de MPI:

- MPICH
- OpenMPI

MPICH ha quedado más rezagado.... OpenMPI está más actualizado.
En el curso utilizaremos los dos para poder manejar cualquier clúster.

Es una biblioteca y tiene herramientas

OpenMPI: sudo apt install openmpi-bin

mpiexec --help

mpiexec -np 1 date // corre date con 1 proceso

mpiexec date // corre date con 4 procesos

ejemplo de uso:

```
[B95690@arenal ~]$ mpiexec -np 10 hostname
```

aplicación: usar una instrucción para ejecutar algo muchas veces

archivo de configuración:

ls -la /etc/skel

```
-rw-r--r-- 1 root root 56 sep 10 15:44 hosts-mpich  
-rw-r--r-- 1 root root 80 sep 10 15:45 hosts-openmpi
```

MPI puede distribuir procesos sin tener que loguearse (ejemplo del clúster de arenal) y funciona para funciones fuera de MPI (como hostname, date etc)

-f para enviar de forma distribuida

el que me crea como proceso es el mpiexec (ambiente de ejecución - comunicador), además los intercomunica.

ejecute el comando `` ```con un equipo (world) de n procesos.

“es un mundo de procesos”

Por eso está tanto la palabra Comm, porque mpiexec es el comunicador al que se le preguntan cosas (como el rank).

Con MPI se podrían crear varios equipos (worlds) de procesos. El mundo por defecto no es el mundo 0... MPI lo define en una constante:MPI_COMM_WORLD

carpetas common:

son carpetas que tienen archivos en común

mpicc invoca al compilador por defecto (gcc) y le pasa los parámetros:

mpicc --version

sale la versión de cc (gcc).

En makefile con include (reutilización de código):

include ../../common/Makefile

CC=mpicc
XC=mpic++

pseudo apegado a MPI:

4.1. Distribución simétrica (MPI)

Message Passing Interface (MPI) es una tecnología de distribución homogénea creado por un grupo de académicos con el fin de facilitar el paralelismo de aplicaciones científicas, que se convirtió en un estándar de facto. Existen otras tecnologías como Charm++ de más alto nivel.

Cree una carpeta `mpi/` en su repositorio de control de versiones. Para el pseudocódigo se utilizará la siguiente convención de paso de mensajes:

```
// Global constants
process_number    // Number of this process (integer)
process_count     // Number of processes in the team this process belongs to
process_hostname  // Name of the computer where this process is running (text)
any_process       // For accepting messages from any process
any_tag           // For accepting messages classified with any tag

// Send a variable or array (a set of bytes) to the destination process
send(message, count, to_process, tag = 0)
// Receive a set of bytes into a variable from a source process
receive(&message, count, from_process, tag = 0)
// Send a variable or array from the source process to the rest
broadcast(&message, count, from_process)
// Reduce the data value from all processes to a single result value in
// destination process applying the given operation
reduce(data, &result, count, operation, to_process)
// Reduce the data value from all processes to a single result value that will
// be available in all processes
reduce_for_all(data, count, &result, operation)
// Distributes an array into subsets for each process using block mapping
scatter(entire_array, entire_count, &subset_array, &subset_count, from_process)
// Updates the entire array from the subsets that were assigned to each process
gather(&entire_array, entire_count, subset_array, subset_count, to_process)
```

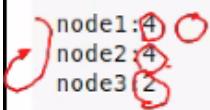
en vez de equipo de hilos, son procesos... podríamos crearlo en una compu y otros en otra.
mpi funciona para C y C++.

```
bin/hello
mpiexec bin/hello
mpiexec -np 1 bin/hello
```

La depuración, búsqueda de errores, fallos en el mundo distribuido es muy difícil... lo que se hacen son bitácoras y monitorio (logging (mandamos a una máquina de bitácoras la info), monitoring (filtra y reduce la info en reportes críticos en un momento) + visualization (aprovechar los pixeles para ilustrar la bitácora, como gráficas))

El clúster usa mpich por defecto

mpich usa un mapeo cíclico con bloques cuando asigna proceso a las máquinas



Es común que en los clústers de computadoras se ofrezcan archivos preconfigurados como los anteriores o preferiblemente se usen sistemas de colas. Por ejemplo, para el clúster de Arenal de la ECCI, se ofrecen copias en la carpeta `/etc/skel/`, la cual se copia cuando se crean las cuentas de usuario, por lo que probablemente disponga de ellos en su carpeta de usuario.

0 -> 12

~np 13

Ambas tecnologías reparten los procesos usando un mapeo cíclico, y los números en el archivo de `hosts` corresponden a tamaños de bloque. Por ejemplo, si se crean 13 procesos se asignarán los primeros 4 al `node1`, los siguientes 4 al `node2`, los siguientes 2 al `node3`, y los restantes 3 al `node1` como se ve a continuación:

node1:	0	1	2	3	4	5	6	7	8	9	10	11	12
node2:													
node3:													

cantidad apropiada del tamaño de bloque: 1

```
[arenal:hello]$ cat hosts-mpich
compute-0-0:8
compute-0-1:8
compute-0-2:8
compute-0-3:8
[arenal:hello]$ nano hosts-mpich
[arenal:hello]$ cat hosts-mpich
compute-0-0:1
#compute-0-1:1
compute-0-2:1
compute-0-3:1
[arenal:hello]$ cat hosts-mpich
```

editar el hosts-mpich para deshabilitar la compu 1.

comando module

module --help

module list: dice cuántos módulos tengo cargados por defecto.

module purge (elimina los que esté en la lista)

module avail (dice qué módulos están disponibles)

```
[arenal:hello]$ module list
Currently Loaded Modulefiles:
 1) /share/apps/modulefiles/mpich
[arenal:hello]$ module purge
[arenal:hello]$ module list
No Modulefiles Currently Loaded.
[arenal:hello]$ mpieexec --help
bash: mpieexec: command not found...
[arenal:hello]$ module avail

----- /usr/share/Modules/modulefiles -----
dot           module-info    null          opt-python    rocks-openmpi_ib
module-git    modules       opt-perl      rocks-openmpi  use.own

----- /etc/modulefiles -----
mpi/openmpi-x86_64

----- /share/apps/tools/modulefiles -----
compilers/gcc-9.4.0   development/python-3.9.7 mpi/mpich-3.2.1      mpi/openmpi-4.1.1
[arenal:hello]$
```

module load (carga un módulo)

module add (carga un módulo)

```
[arenal:hello]$ bin/hello
Hello from main thread of process 0 of 1 on arenal.ecci.ucr.ac.cr
[arenal:hello]$ mpieexec bin/hello
Hello from main thread of process 0 of 4 on arenal.ecci.ucr.ac.cr
Hello from main thread of process 1 of 4 on arenal.ecci.ucr.ac.cr
Hello from main thread of process 2 of 4 on arenal.ecci.ucr.ac.cr
Hello from main thread of process 3 of 4 on arenal.ecci.ucr.ac.cr
[arenal:hello]$ mpieexec -np 3 bin/hello
Hello from main thread of process 0 of 3 on arenal.ecci.ucr.ac.cr
Hello from main thread of process 1 of 3 on arenal.ecci.ucr.ac.cr
Hello from main thread of process 2 of 3 on arenal.ecci.ucr.ac.cr
[arenal:hello]$ nano hosts-openmpi
[arenal:hello]$ mpieexec -np 3 -f hosts-openmpi bin/hello
mpieexec: Error: unknown option "-f"
Type 'mpieexec --help' for usage.
[arenal:hello]$
```

openmpi no reconoce el -f

utiliza más bien un hostfile

```
[arenal:hello]$ mpieexec -np 3 --hostfile hosts-openmpi bin/hello
Hello from main thread of process 0 of 3 on compute-0-0.local
Hello from main thread of process 1 of 3 on compute-0-2.local
Hello from main thread of process 2 of 3 on compute-0-3.local
[arenal:hello]$ mpieexec -np 5 --hostfile hosts-openmpi bin/hello
```

el parámetro oversubscribe dice: si son más procesos (slots) que máquinas entonces empieza a repartir (sobrecargar) los nodos. (mpich lo hacía por defecto). OpenMPI ordena la salida (es más fácil de estudiar que en mpich).

OpenMPI escoge un mapeo estático por bloque. MPICH utiliza un mapeo cíclico por bloque.

remote SSH (extensión de VSC)

para editar a distancia

sudo apt install sshfs

instala un paquete de sistema de archivos ssh

necesita un url y una carpeta vacía (para montar)

Host *

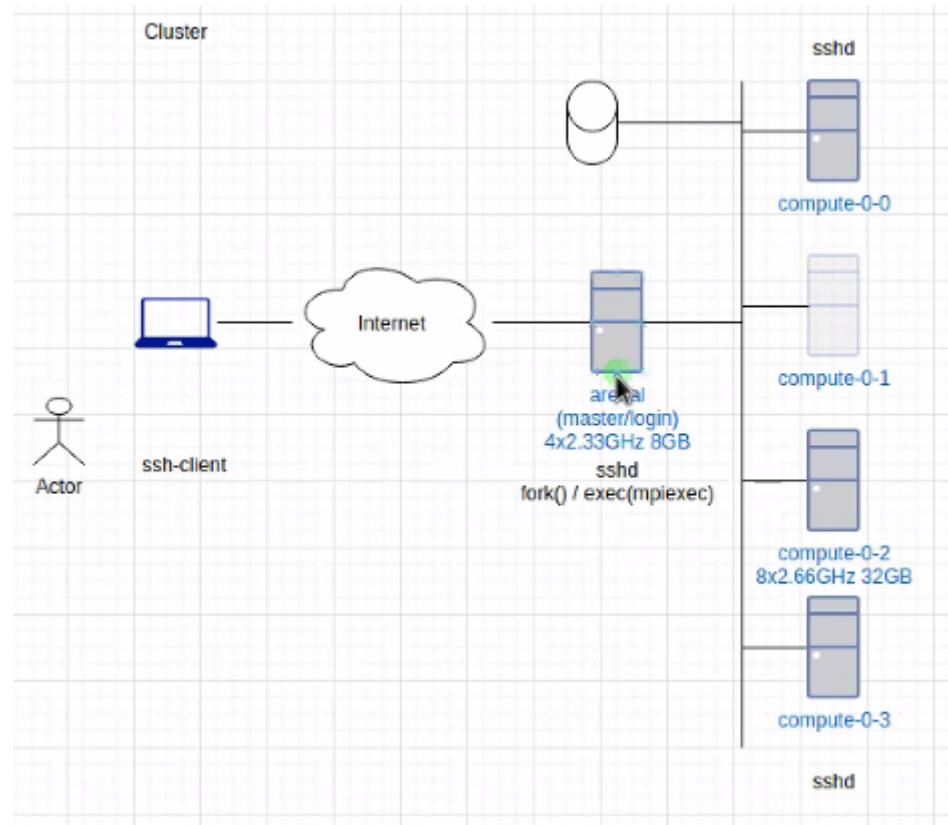
ServerAliveInterval 240

mantiene viva una conexión con los servidores remotos

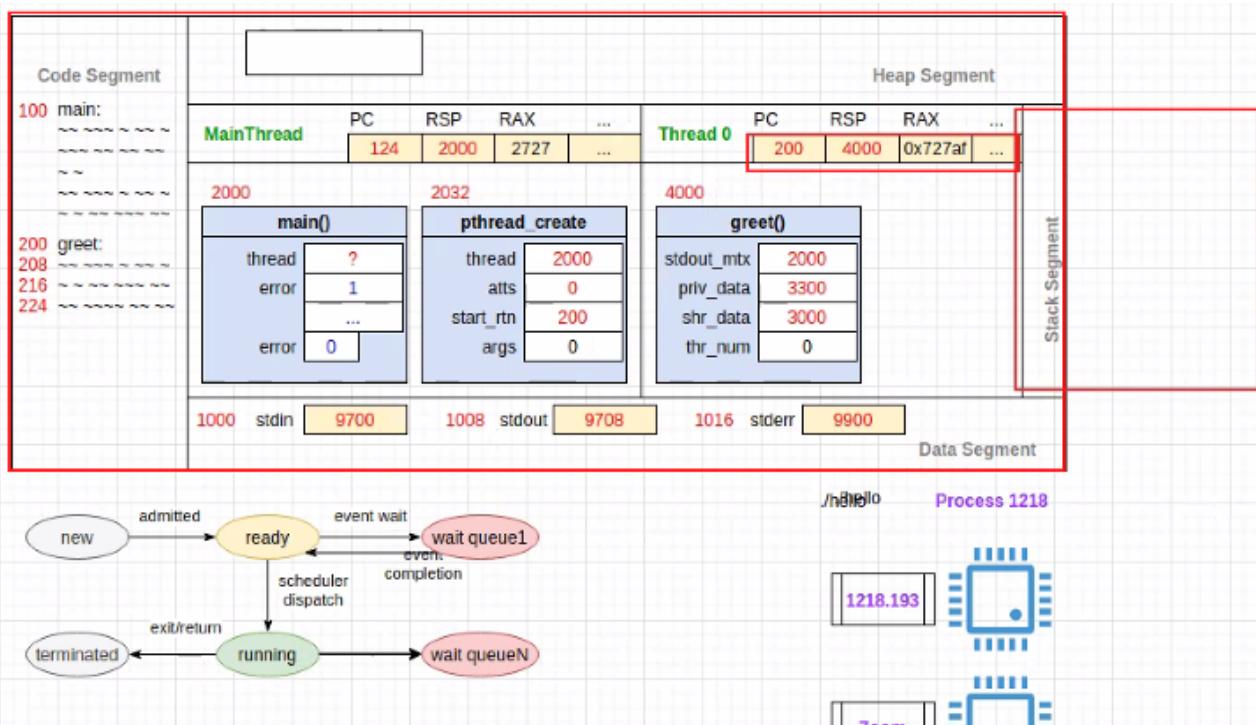
Sem13-b

Ejemplo hello_hybrid

Entender como funciona mpi por debajo sirve para entender servidores web, bases de datos etc

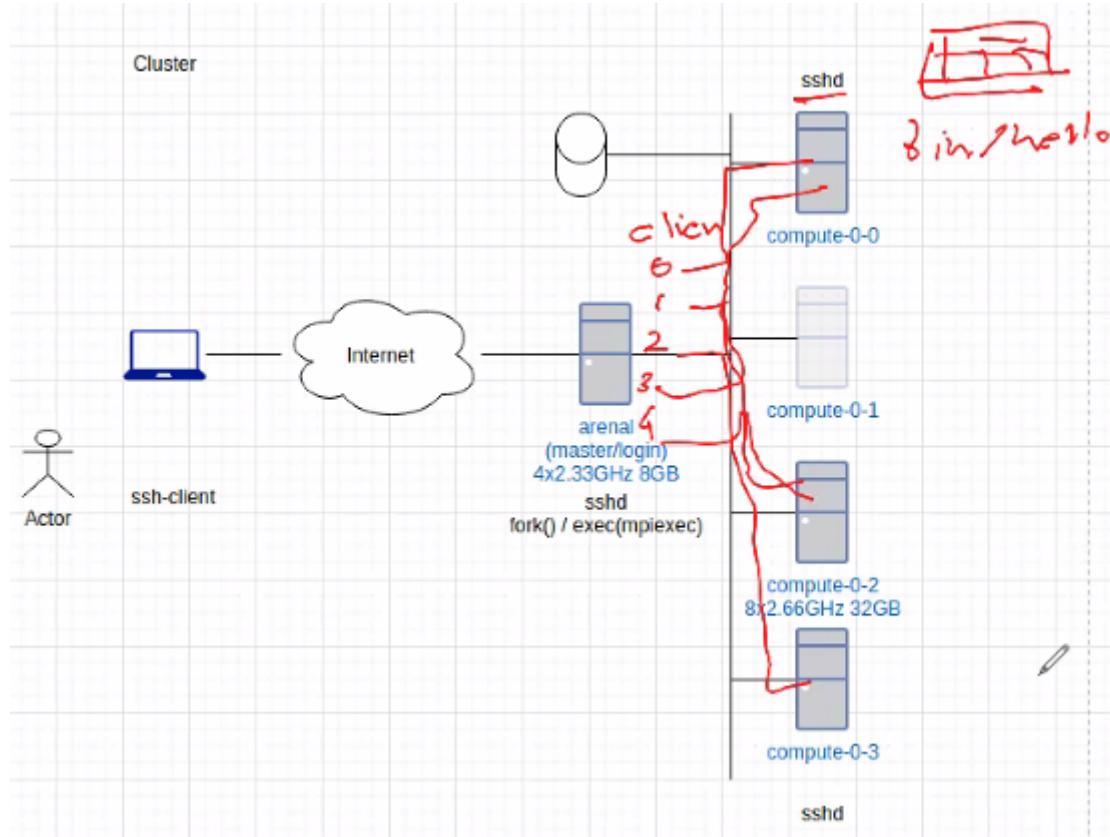


1. se conecta a un servidor remoto con ssh, se bifurca mediante el sshd mediante un fork() que duplica todo el proceso.



exec (mpiexec) lo que hace es que ahora que hay dos procesos, reemplace el segundo (hijo) por otro ejecutable.

2. mpi utiliza la biblioteca ssh, en linux ssh tiene esa biblioteca. Cuando se pide: mpiexec corra n procesos, lo que hace es crear n clientes (hilo) de ssh que se va a hablar con el servidor (sshd) mediante un socket de red y le pide que corra un ejecutable (el que le dan por la terminal).



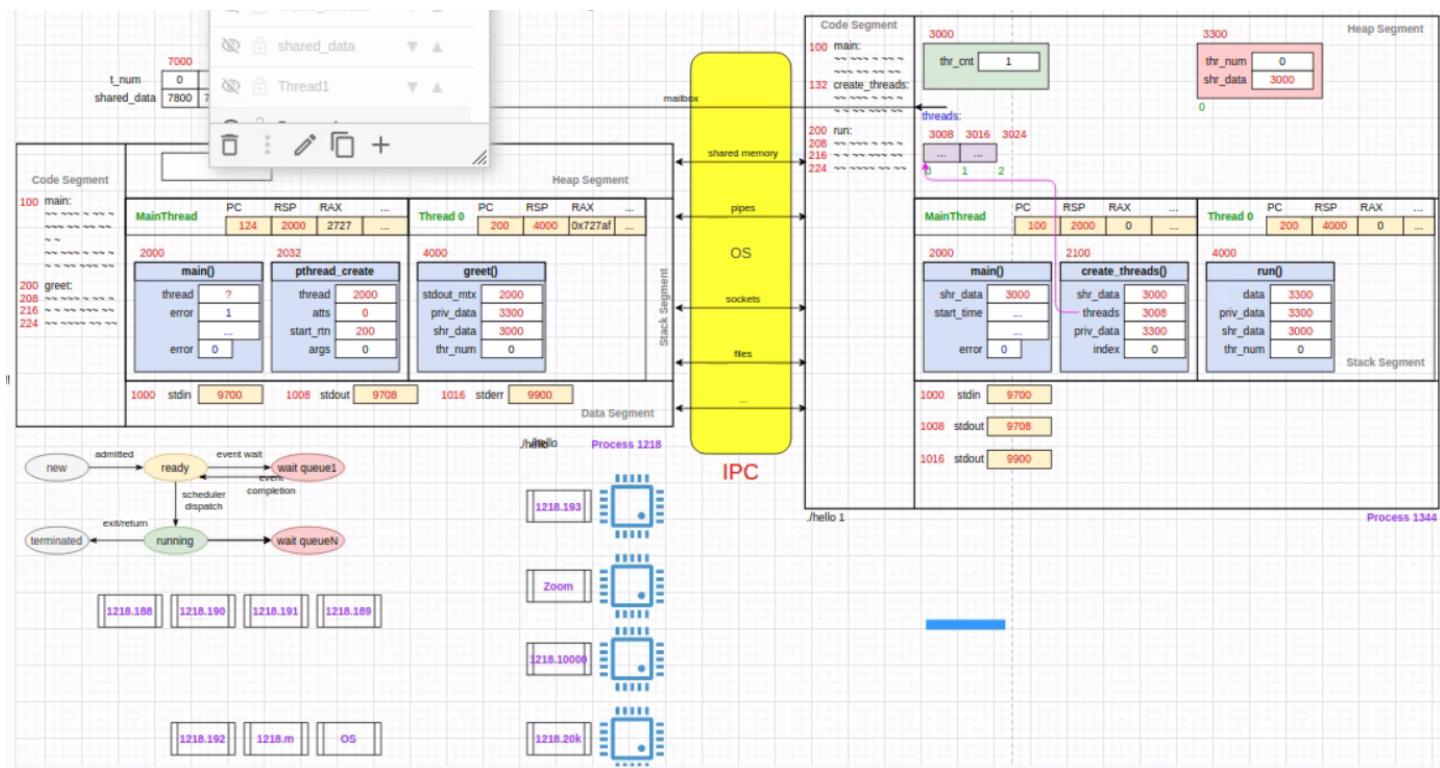
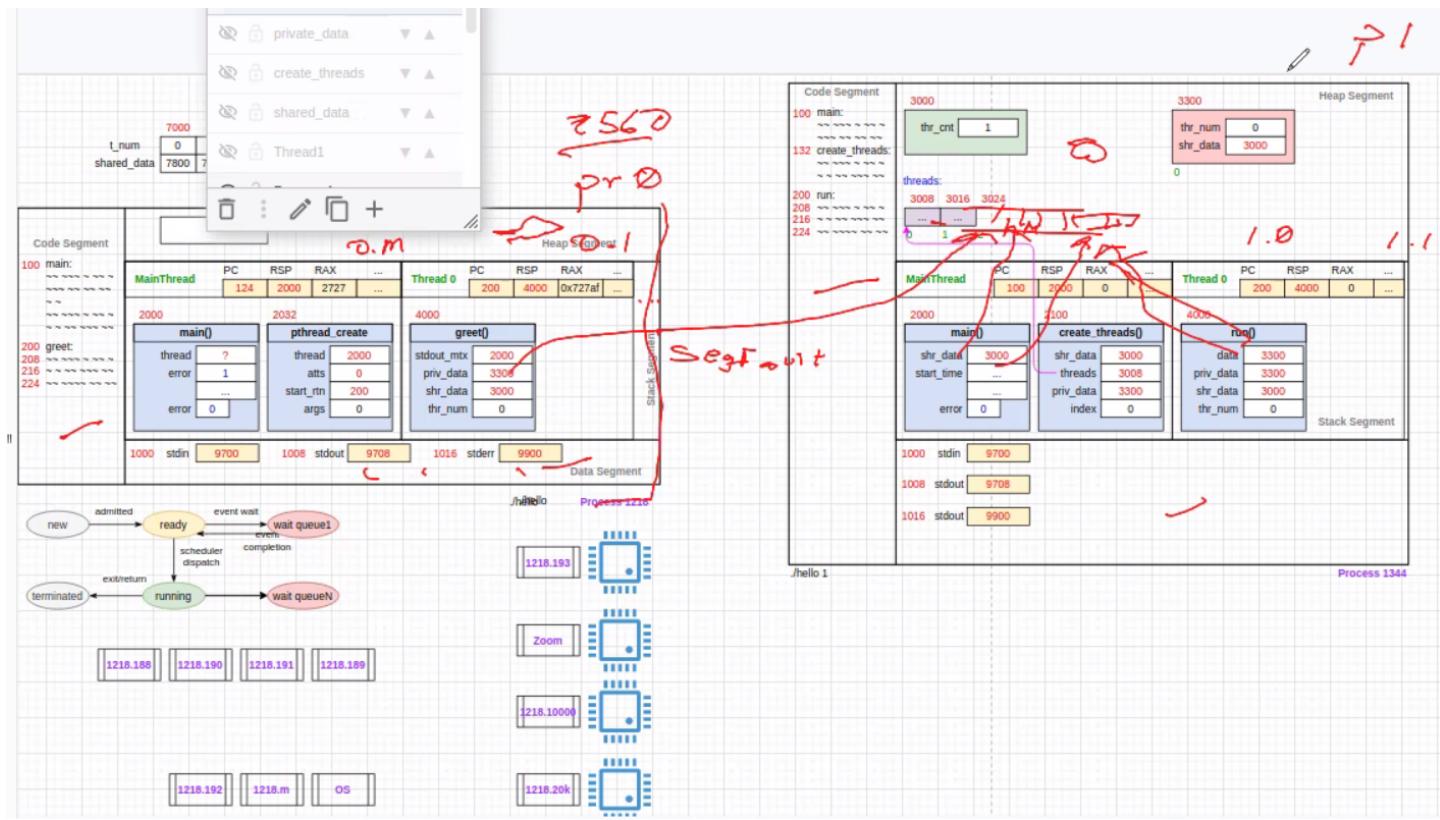
El clúster está configurado sin password para que confíe en el nodo master.

network file system (nfs) : sistema de archivos en disco duro aparte o junto (con muchos discos duros redundantes, por si uno falla, incluso trabajando en paralelo.). Los ejecutables se guardan en el nfs y se consultan lento porque es por medio de red (no directamente del disco duro).

nota:

mpich: cliquico por bloque

mpi: estático por bloque



Entre procesos no pueden acceder a la memoria de otro porque se da un segmentation fault... solo pueden ser accedidos por medio del SO (lo que lo hace más lento).

Solución: el SO crea una memoria compartida para procesos en la memoria principal (si los procesos están en la misma compu).

Por medio de pipes (la entrada es la salida de otro).

Sockets (permite comunicación entre compus que están en otros lugares (es más lento pero es el más usual)

Archivos (lento, puede generar condiciones de carrera porque el archivo está en un disco duro que puede acceder cualquier hilo).

Cada hilo puede acceder sin intervención del SO solo si es en el mismo proceso.

siguiendo con el hello_hybrid:

```
1 procedure main(argc, argv[])
2     // create thread_count as result of converting argv[1] to integer
3     shared const thread_count = integer(argv[1])
4     create_threads(thread_number, greet)
5     print "Hello from main thread of process ", process_number, " of " \
6           , process_count, " on ", process_hostname
7 end procedure
8
9 procedure greet(thread_number)
10    print "\tHello from thread ", thread_number, " of ", thread_count, \
11        " of process ", process_number, " of ", process_count, " on " \
12        , process_hostname
13 end procedure
14
```

pseudocódigo híbrido usando pthreads(imperativo) y distribución(por las constantes).

pseudocódigo híbrido usando openMP(declarativo) y distribución(por las constantes de proceso).
Basta con rotular que el ciclo es parallel

```
1 procedure main(argc, argv[])
2     shared const thread_count = integer(argv[1])
3     print "Hello from main thread of process ", process_number, " of " \
4           , process_count, " on ", process_hostname
5
6     parallel do
7         print "\tHello from thread ", thread_number, " of ", thread_count, \
8             " of process ", process_number, " of ", process_count, " on " \
9             , process_hostname
10    end parallel
11
12 end procedure
```

este último incluso puede servir para pthreads.

Salida estándar ordenada / desordenada:

OpenMPI utiliza un mecanismo de control (guarda en buffer lo que viene de cada hilo)

MPICH recibe bytes y los lanza como vengan

Siguiente ejemplo:

hybrid_distr_arg

distribuir el rango...

Los mapeos en openMP están implementados cuando uno hace un for

En MPI no hay implementación de la descomposición ni del mapeo (el programador decide); MPI tiene dos instrucciones : scatter y gather que hacen un mapeo por bloque pero tiene una interfaz muy incómoda y queda mejor hacerlo manualmente.

3.4.1. Mapeo por bloque

El **mapeo estático por bloque** asigna rangos continuos de trabajo a cada trabajador. Es el mapeo que potencialmente puede disminuir más fallos de caché o *false sharing* si se trabaja con memoria continua. El bloque de un trabajador i está dado por el rango de índices enteros $[start, finish]$, donde $start$ y $finish$ son funciones dadas por

$$start(i, D, w) = i \left\lfloor \frac{D}{w} \right\rfloor + \min(i, \text{mod}(D, w))$$

$$finish(i, D, w) = start(i + 1, D, w)$$

donde i es el número del trabajador (iniciando en 0), D es la cantidad de datos, y w es la cantidad total de trabajadores.

¿Qué pasa cuando el mapeo por bloque no empieza en 0?

D: range. rango de valores (cantidad de datos) que tengo que repartir.

i: rank

w: workers

Se le suma el begin (funciona como constante de una función lineal).

nota: para openMP, en los datos compartidos no se deben pasar los datos const porque por defecto son compartidos.

mapeo dinámico con MPI para las tareas.

send-receive

tecnología de paso de mensajes a través de una red, tiene muchos usos: control de concurrencia por ejemplo.

MPI posee muchas familias de envío de mensajes.

MPI_Recv implica bloqueo (espera).

ejemplo send_recv_ord_sem

send() envía bytes hacia un proceso. Es una subrutina de E/S sincrónica (**synchronous io | half blocking io**). Retorna cuando la otra máquina reciba el primer byte.

receive() se espera que lleguen **todos** los bytes de la otra máquina. Es la que más bloquea de todas las subrutinas receive. Es (**synchronous io | full blocking io**).

lógica modular con números negativos: sumar la cantidad de valores inicialmente no afecta el resultado y soluciona el error.

```

1 procedure main:
2   declare const previous_process = (process_count + process_number - 1) \
3     % process_count
4   declare const next_process = (process_number + 1) % process_count
5   declare can_print := true
6
7   if process_number > 0 then
8     | receive(&can_print, 1, previous_process)
9   end if
10  print "Hello from main thread of process ", process_number, " of " \
11    , process_count, " on ", process_hostname
12  send(can_print, 1, next_process)
13 end procedure

```

status es opcional, nos dice si hubo error y quién lo envió.

Sem14-a

receive (espera)

send (avisa)

signaling

quitar openMPI y poner Mpich:

sudo apt remove openmpi-bin; sudo apt autoremove; sudo apt update; sudo apt install mpich

La red es propensa a errores.

excepciones de C++

std::exception

throw std::runtime_error("no pude recibir un mensaje")

.what() da el texto que genera el throw.

se puede definir una macro para reducir el código:

```
#define fail(msg) throw std::runtime_error(msg)
```

El paso de mensajes en MPI puede ser orientado a arreglos (vectorial) siempre y cuando estén serializados en la memoria. También permite separar asuntos de acuerdo al número de proceso (rank).

se pone un buffer con x capacidad para recibir n bytes.

```
receive(&message, message_capacity, source) la cantidad recibida se desconoce
```

Almacenar en buffer en C:

```
char buffer[210]
sprintf(buffer, "Hello from %zu on %s", sdafd, asdfjasf)
```

en C++:

```
#include <iostream>
```

```

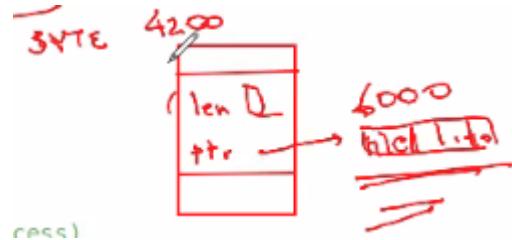
std::stringstream buffer;
buffer << "Hello from main thread of process " << process_number
<< " of " << process_count << " on " << process_hostname << std::endl;

```

str:

con parámetro: mete lo nuevo
sin parámetro: saca lo que tenía

No se envían objetos por la red, sino los buffer



con data() retorna un puntero a los caracteres

message.data() de message.length() caracteres de tipo MPI_CHAR que se lo envía al proceso 0 por el comunicador en común.

```

const std::string& message = buffer.str();
if (MPI_Send(message.data(), message.length(), MPI_CHAR, /*target*/ 0
, /*tag*/ 0, MPI_COMM_WORLD) != MPI_SUCCESS) {
    fail("could not send message");
}

```

```

if (process_number == 0) {
    for (int source = 0; source < process_count; ++source) {
        std::vector<char> message2(MESSAGE_CAPACITY);
        if (MPI_Recv(&message2[0], MESSAGE_CAPACITY, MPI_CHAR, source
        , /*tag*/ 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) != MPI_SUCCESS ) {
            fail("could not receive message");
        }
}

```

El programa falla porque la tecnología no permite enviarse mensaje a sí mismo

falla porque el proceso 0 espera un mensaje de sí mismo, mpich y mpi lo hacen con **espera activa**.

```

if process_number != 0 then
    send(message, length(message), 0)
else
    for source := 0 to process_count do
        receive(&message, message_capacity, source)
        print(source, " sent ", message)
    end
end if
end procedure

```

excluyendo el 0.

msg (message) match:

forma de hacer coincidir los mensajes (qué sent con qué receive)

process match ocurre con 3 parámetros:

- tipo de dato, proceso origen y tag (proceso destino).

```

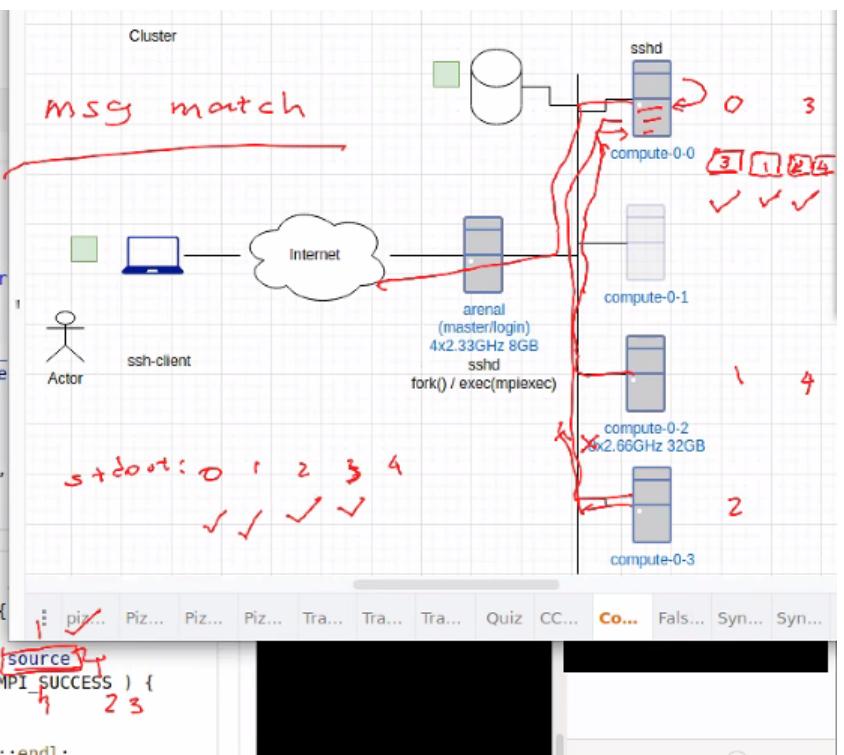
for source := 1 to process_count do
    receive(&message, message_capacity, source)
    print(source, " sent ", message)
end

v_ord_itm.cpp 2, U x
ord_itm > src > C: send recv_ord_itm.cpp > G: greet(int, int, const char *)
    error = EXIT_FAILURE;
}
return error;

id greet(int process_number, int process_count, const char*
std::stringstream buffer;
buffer << "Hello from main thread of process " << process_
<< " of " << process_count << " on " << process_hostname

if (process_number != 0) {
    const std::string& message = buffer.str();
    if (MPI_Send(message.data(), message.length(), MPI_CHAR,
        /*tag*/ 0, MPI_COMM_WORLD) != MPI_SUCCESS) {
        fail("could not send message");
    }
} else {
    std::cout << process_number << " said " << buffer.str()
    for (int source = 0; source < process_count; ++source) {
        std::vector<char> message(MESSAGE_CAPACITY);
        if (MPI_Recv(&message[0], MESSAGE_CAPACITY, MPI_CHAR, source,
            /*tag*/ 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) != MPI_SUCCESS) {
            fail("could not receive message");
        }
    }
}

```



Si no coincide el source, el proceso queda en espera activa.

para recibir sin importar quién envíe (indeterminista) pongo por parámetro de receive “any_process” : MPI_ANY_SOURCE en lugar del source (for de 0 a process_count). También existe any_tag...

De esa forma el intermediario recibe los datos en el orden que lleguen.

Familia de send:

hay muchos send

Send:

- I nonblock starts copying to buffer
- B blocking buffered
- Ib nonblocking buffered
- SSend blocking arrived/standard (Send)
- R ready send/receive before
- Ir nonblock ready

Recv blocking standard

I nonblock standard

M blocking matched

Im nonblocking matched

Varían en si permite controlar los buffers y si yo me bloqueo a la hora de hacer un envío o no.

I: por eventos (no bloquea, inmediatamente retorna).

B: controlo los buffers (buffers propios)

Send/SSend: es el que utilizamos, bloquea hasta que recibe el primer byte (retorna ACK (aviso))

Rsend: bloquea hasta que llegue el último byte (full blocking)

IR:

Receive:

Recv: es el que utilizamos, bloquea hasta que llegue el último byte

I:

M:

Im:

La importancia es saber que dependiendo de cuál usemos, podemos sacar provecho para tener más velocidad (gigas de genomas por ejemplo)

La entrada la lee mpiexec, mpiexec la captura y envía al proceso 0 (los demás procesos no leen de la entrada).

Solución:

```
if process_number = 0 then
    declare value := 0
    while can_read(stdin) do
        append(values, value)
    end while
end if
```

La lectura solo la hace el proceso 0.

Ahora, el proceso 0 debe enviar esa información a los demás procesos.

típicamente la red envía paquetes de 1500 bytes
enviar paquetes muy pequeños puede ser muy ineficiente.

no usar variables const porque la ocupo tener en memoria para poder enviar o recibir.

MPI asegura que los datos sean recibidos en el mismo orden en que fueron enviados (send serializado).

```
value_count := count(values)
for target := 1 to process_count do
    send(&value_count, 1, target)
    send(&values, count(values), target)
end
else
    receive(&value_count, 1, 0)
    receive(&values, capacity??, 0)
end if
```

se recibe primero el value_count que values siempre. Lo

que es indeterminista es el target (proceso) al que le llegue primero.

para vector:

resize cambia la capacidad y el count

reserv solo cambia la capacidad

static_assert() es un assert que se da en tiempo de compilación.

`static_assert(sizeof(value_count) == sizeof(uint64_t));` programación defensiva para un futuro (por si el size_t en otra máquina es distinto de 64 bits).

Los sistemas distribuidos no comparten recursos (memoria, cpu, relojes) entonces cuesta calcular el tiempo de ejecución de cada proceso. Se usa un protocolo de tiempo para intentar triangular el tiempo pero no hay garantía de que esté en el mismo tiempo.

El tiempo se puede usar para referencia para sí mismo pero no es válido para otros procesos.

En MPI:

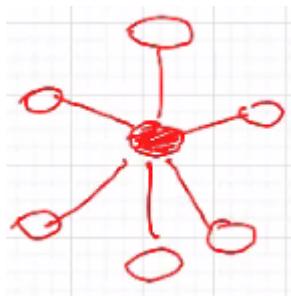
la barrera se levanta cuando todos los miembros lleguen a ella.
signaling con send y receive.

En redes:

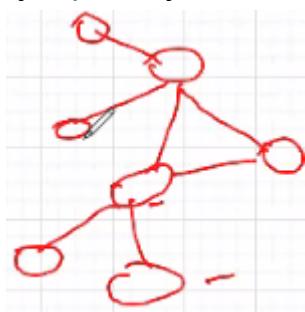
centralizado o descentralizado

centralizado: modelo cliente / servidor -> se tiene alguien que centraliza (todos los miembros se comunican con uno solo).

desventaja: el del centro se puede atarrear
es fácil programar.



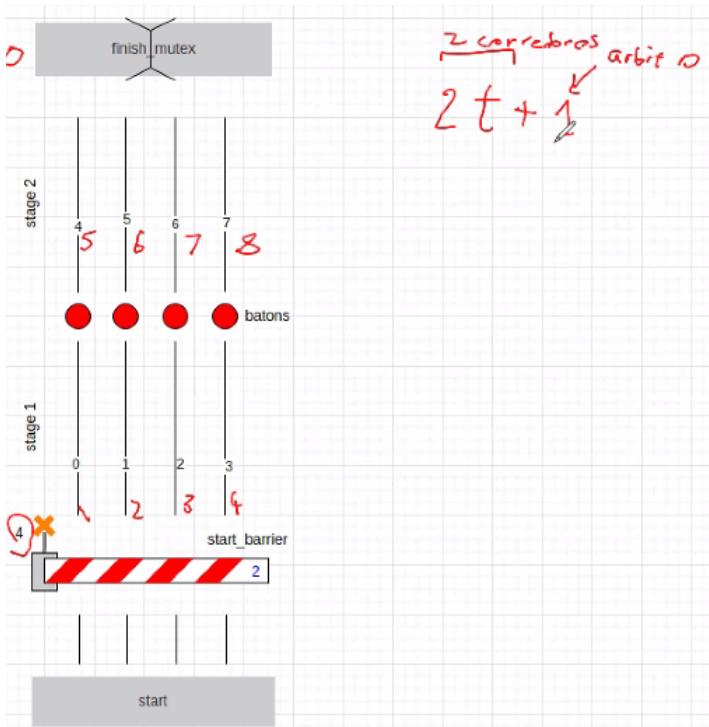
descentralizado: peer2peer -> se conecta cualquiera con cualquiera y toman el rol que quieran.
ejemplo: bajar un torrent.



es más complejo de programar.

en el curso se usarán redes centralizadas.

proceso 0 tiende a hacer las cosas diferentes.



actividad 50

2 corredores por equipo; 4 equipos. 1 árbitro (línea de meta, proceso 0).

separación de asuntos entre procesos (mundo distribuido):

```

1 procedure main(argc, argv[]):
2   if argc = 3 then
3     if process_count >= 3 and process_count % 2 = 1 then
4       declare constant team_count = div(process_count, 2)
5       declare constant stage1_delay = integer(argv[1])
6       declare constant stage2_delay = integer(argv[2])
7
8       if process_number = 0 then
9         referee()
10      else if process_number <= team_count then
11        run_stage1(stage1_delay, process_number, team_count)
12      else
13        run_stage2(stage2_delay, process_number, team_count)
14      end if
15    else
16      print "error: process count must be odd and greater than 3" condicionando por
número de proceso

```

problema con la barrera: no se puede inicializar, mpi la inicializa en el número de procesos

MPI tiene un reloj: MPI_Wtime

send y receive son de comunicación punto a punto

Comunicación colectiva:

más conveniente y eficiente que punto a punto ya que es natural y gratis.

con broadcast: de 1 a varios, envía y recibe (no se declara dos veces para el mismo valor enviado/recibido, con una vez es suficiente).

Es más compacto que send y receive

Sem15a

continuamos con broadcast

Reducciones en mundo distribuido con MPI

con -I se hacen includes, puede incluir toda una carpeta con makefile

reduce: toma un arreglo de valores

de varios procesos puedo tomar varios valores en una misma posición de un arreglo y lo pongo en la misma posición de otro arreglo en un proceso.

```
// Reduce the data value from all processes to a single result value in  
// destination process applying the given operation  
reduce(data, &result, count, operation, to_process)
```

reducciones viene del paradigma funcional

programación de orden superior permite que las funciones sean valores (utilizadas como variables).

map reduce

map recibe un vector, lo modifica en las mismas posiciones

reduce agarre un vector y aplica una función a todo el vector y lo guarda en un único valor.

Collective operations

The collective combination operations (e.g., MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER, and MPI_SCAN) take a combination operation. This operation is of type MPI_Op in C and of type INTEGER in Fortran. The predefined operations are

MPI_MAX

return the maximum

MPI_MIN

return the minimum

MPI_SUM

return the sum

MPI_PROD

return the product

MPI_LAND

return the logical and

MPI_BAND

return the bitwise and

MPI_LOR

return the logical or

MPI_BOR

return the bitwise or

MPI_LXOR

return the logical exclusive or

MPI_BXOR

return the bitwise exclusive or

MPI_MINLOC

return the minimum and the location (actually, the value of the second element of the structure where the minimum of the first is found)

MPI_MAXLOC

return the maximum and the location

MPI_REPLACE

replace b with a

MPI_NO_OP

perform no operation



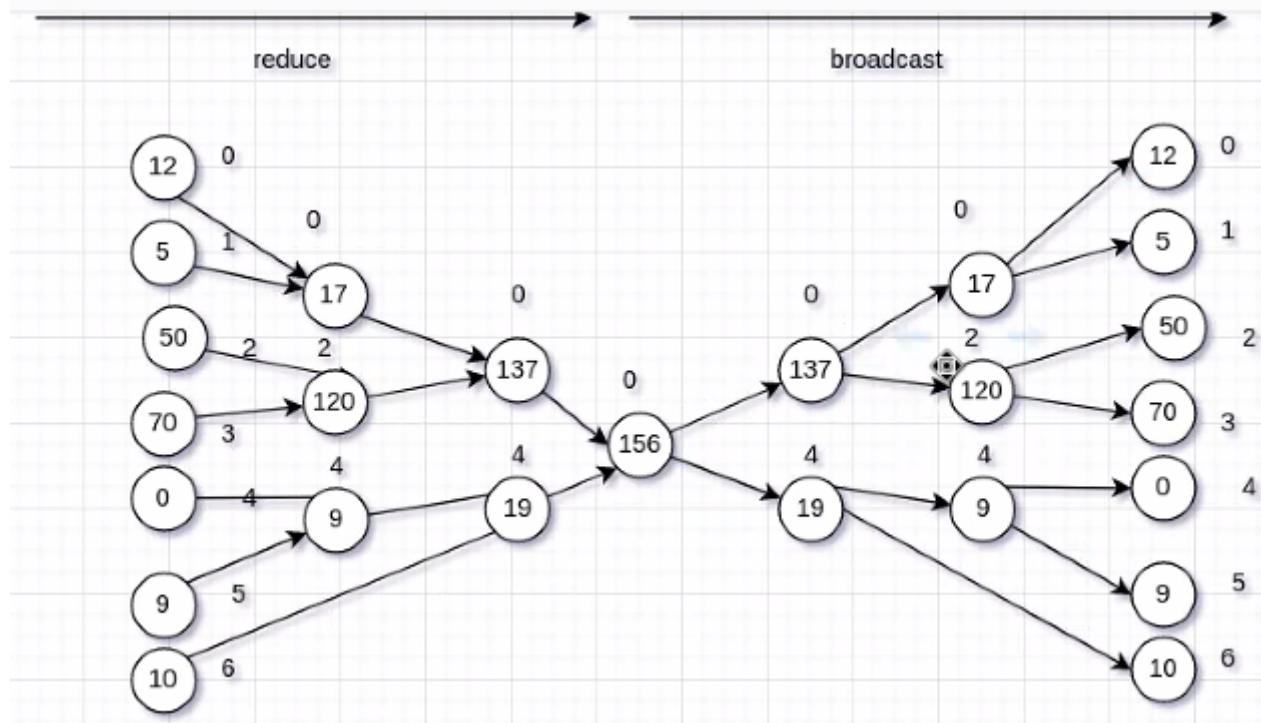
resta y división no están porque no tienen sentido

```

// Update distributed statistics from processes' lucky numbers
if (MPI_Reduce(/*input*/ &my_lucky_number, /*output*/ &all_min, /*count*/ 1
    , MPI_INT, MPI_MIN, /*root*/ 0, MPI_COMM_WORLD) != MPI_SUCCESS) {
    fail("error: could not reduce min");
}
if (MPI_Reduce(/*input*/ &my_lucky_number, /*output*/ &all_max, /*count*/ 1
    , MPI_INT, MPI_MAX, /*root*/ 0, MPI_COMM_WORLD) != MPI_SUCCESS) {
    fail("error: could not reduce max");
}
if (MPI_Reduce(/*input*/ &my_lucky_number, /*output*/ &all_sum, /*count*/ 1
    , MPI_INT, MPI_SUM, /*root*/ 0, MPI_COMM_WORLD) != MPI_SUCCESS) {
    fail("error: could not reduce sum");
}

```

MPI implementa el reduce con mensajes punto a punto.



un reduce seguido de un broadcast es muy común (all_reduce).

```

// Reduce the data value from all processes to a single result value that will
// be available in all processes
all_reduce(data, &result, count, operation)

```

todos los procesos quedan con el dato del reduce

single: solo un thread ejecuta

embudo: el hilo que llama a init thread es el que hace invocaciones de MPI

serializado: utiliza un mutex para atender cada llamado a función de MPI

múltiple: permite que se invoquen las funciones sin restricciones

C no está pensado para multithreading, MPI_FILE da un empujón para la entrada y salida en C.

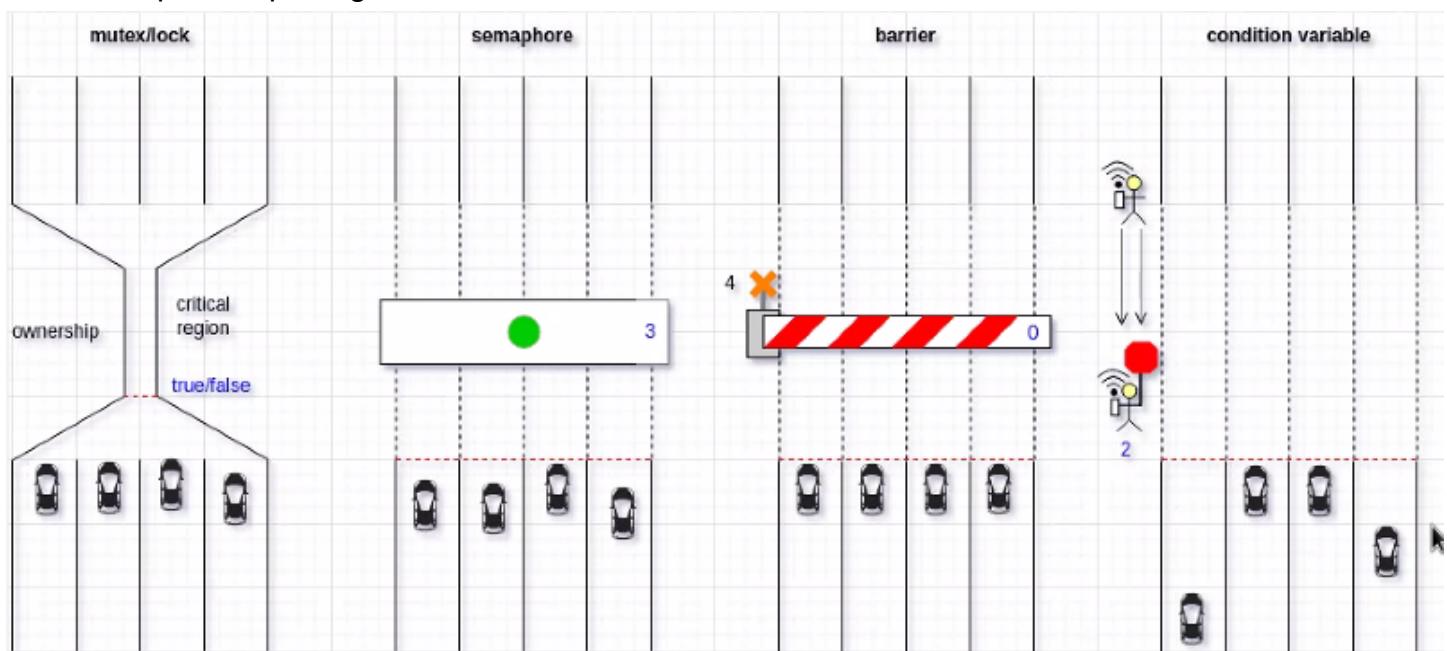
Volviendo a concurrencia de tareas

Misterio (variable de condición)

variable de condición: "mecanismo de control de concurrencia (como mutex, semáforo y barrera)

No es tan atómico.

Metáfora: "policía que regula la cantidad de carros"



La persona sabe cuántos hay en el alto esperando.

Un intercomunicador le indica qué hacer con los hilos que llegaron.

dos opciones: signal (pasa uno) y broadcast (pasan todos)

si hay carros en el alto, pasan, si no habían y se hizo un signal o broadcast no pasa nada y se vuelve a cerrar/ poner en alto.

requiere un mutex (estar en la región crítica)

implementa una barrera (como el torniquete)

wait en variable de condición:

Sem15b

filósofos comensales:

solución: que uno sea distinto (con uno zurdo o derecho basta)

solución2: poner cuota (multiplex) con una cantidad de comensales que pueden comer concurrentemente. (sencillo)

solución3: usando variables protegidas por mutex con los estados en que se encuentran los filósofos.

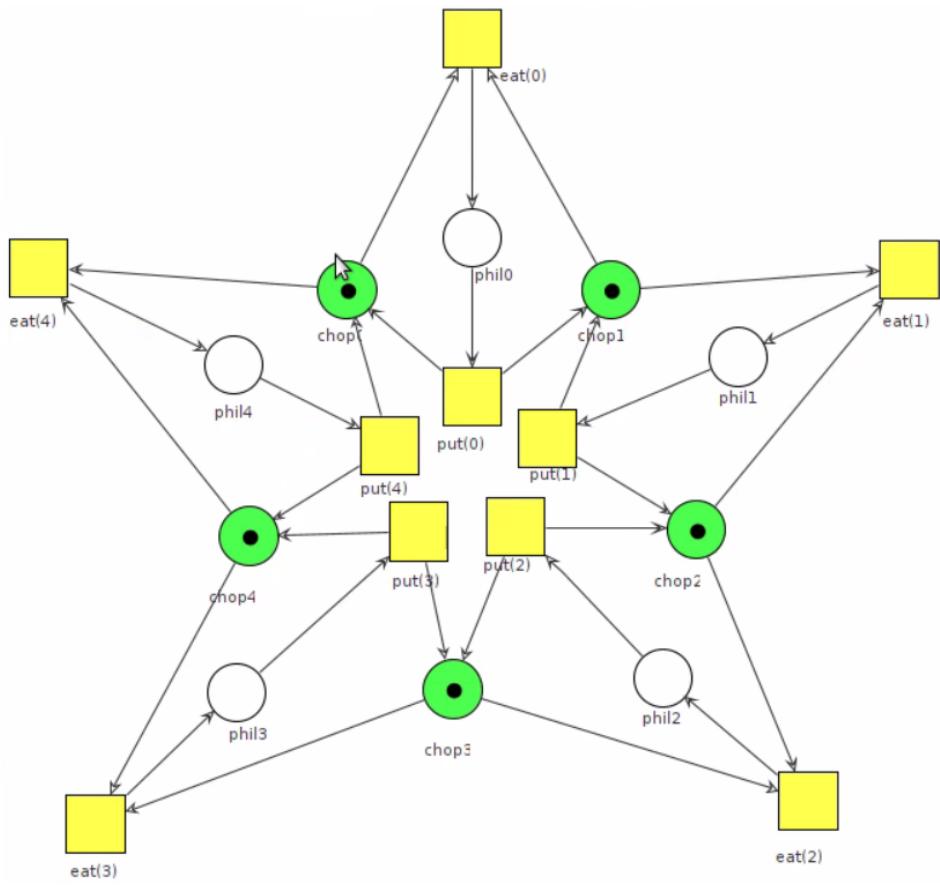
distribuidos (en lugar de hilos son procesos) los filósofos sí hablan entre ellos (no son silenciosos)

solución4:

con modelo centralizado: se decide un árbitro/mesero que tiene el control central de todo; él decide mediante un sistema de control. (sencilla)

solución5: descentralizada: entre los filósofos se hablan (no ocupa un árbitro)

red de petri:



cuando un filósofo tiene el token: está comiendo. No hay bloqueo mutuo pero sí inanición (no está completa - sol3).

ejemplo de lectores y escritores (candado de escritura y lectura):

Exclusión mutua categórica: categorizamos a los hilos(solo se permite la concurrencia a una categoría de estos, no a todos)

t Selection View Go Run Terminal Help

≡ dining_philosophers_quota.pseudo ≡ readers_writers.pseudo U x

readers_writers > ≡ readers_writers.pseudo

```

1  procedure main:
2    shared can_access_medium := create_semaphore(1) 0 -1 -2 -1 0
3    shared can_access_reading := create_semaphore(1) 0 -1 -2 -1 0 | 0
4    shared reading_count := 0 1 2 3 X 0
5
6    while true:
7      case read_char() of:
8        'R': create_thread(reader)
9        'W': create_thread(writer)
10       EOF: return
11     end case
12   end while
13 end procedure
14
15 procedure reader:
16   wait(can_access_reading)
17   reading_count := reading_count + 1
18   if reading_count = 1 then
19     | wait(can_access_medium)
20   end if
21   signal(can_access_reading)
22   read() X 2 3
23   wait(can_access_reading) 2
24   reading_count := reading_count - 1
25   if reading_count = 0 then
26     | signal(can_access_medium)
27   end if
28   signal(can_access_reading)
29 end procedure
30
31 procedure writer: 4
32   wait(can_access_medium) 4 ✓
33   write()
34   signal(can_access_medium)
35 end procedure

```

Diagrama de semáforos:

- Canal de acceso a la lectura (can_access_reading): 0 → -1 → -2 → -1 → 0 | 0 → 1 → 0 → 1 → 0 → 1
- Canal de acceso a la escritura (can_access_medium): 0 → -1 → -2 → -1 → 0 | 0 → 1 → 0 → 1 → 0 → 1

Diagrama de conteo de lectores:

0	1	2	3
X	0	1	1

por puede modificarlo a la inversa, por cada W que se lea de la función read() al mismo considerarse una exclusión mutua en la categoría de hilos, pero excluyendo el lector.

solución utilizando torniquete y mutex para lograr lo buscado (pero no resuelta la inanición)

patrón (candados de lectura y escritura en pthreads) consiste en categorizar la exclusión mutua

pthread_rwlock: el unlock es igual al del mutex

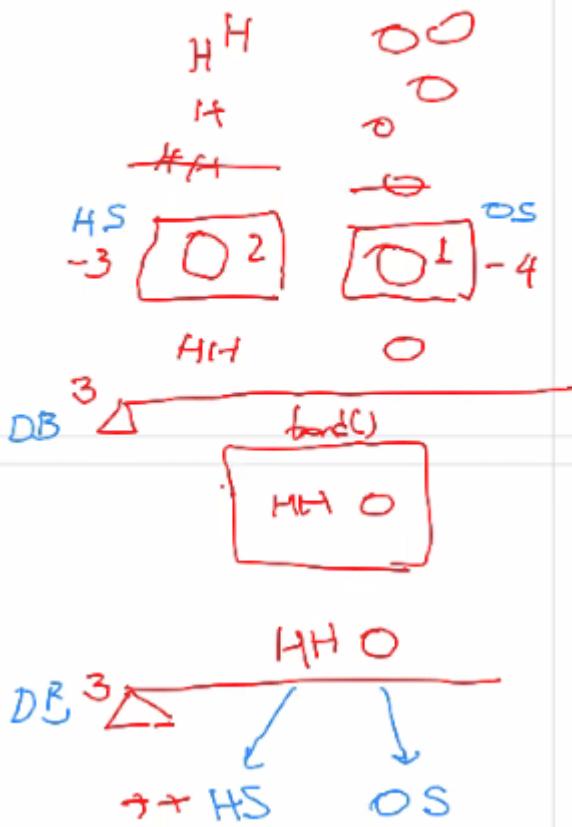
tenemos dos lock: para leer rdlock (no bloquea si hay varios leyendo) y para escribir wlock (igual que el lock del mutex)

cuando casi todo es lectura es mejor el candado de lectura y escritura, sino es mejor usar mutex (exclusión mutua pura).

```

readers_writers.pseudo U      C readers_writers_rwlock.c U      build_h2o.pseudo U ×      ...
build_h2o > ⌂ build_h2o.pseudo
1 procedure main:
2   while true do
3     case read_char() of
4       'H': create_thread(hydrogen)
5       'O': create_thread(oxygen)
6       EOF: return
7     end case
8   end while
9 end procedure
10
11 procedure hydrogen:
12   bond()
13 end procedure
14
15 procedure oxygen:
16   bond()
17 end procedure
18

```



Sem16b

El falso compartido (false sharing)
tema de paralelismo de datos

3.6. Falso compartido (*false sharing*)

Actividad 33. Arreglo falso compartido [*false_sharing_array*]

Estudie y ejecute el programa dado en sus cuatro modos, usando los argumentos de línea de comandos 0 a 3. Todos los modos realizan la misma tarea de actualizar dos elementos de un arreglo cien millones de veces. En los modos 0 y 1 los dos elementos se actualizan de forma serial. En los modos 2 y 3 dos hilos concurrentemente actualizan los elementos, como se muestra en la siguiente tabla.

0	1	2	...	999	1000
M	M				
M					m
0	1				
0					l

Modo	Tipo	Elemento 1	Elemento 2
0	Sequencial	El <u>primero</u> del arreglo	El <u>segundo</u> del arreglo
1	Sequencial	El <u>primero</u> del arreglo	El <u>último</u> del arreglo
2	Concurrente	El <u>primero</u> del arreglo	El <u>segundo</u> del arreglo
3	Concurrente	El <u>primero</u> del arreglo	El <u>último</u> del arreglo

200.000.000 de modificaciones en cualquier modo
No hay condiciones de carrera, cada hilo tiene su índice

mode	ms
0	1987
1	1994
2	1386
3	1001

por qué el modo 3 dura menos que el 2 si es lo mismo, modifican de la misma forma dos celdas, solo que en un modo están más alejadas entre sí.

Por la coherencia de caché

False sharing: como si estuvieran compartiendo el elemento, es por hardware, que genera una espera a nivel de hardware.

```

for (size_t iteration = 0; iteration < ITERATION_COUNT; ++it)
    array[my_index] = array[my_index] + iteration % ELEMENT_SIZE;
}

return NULL;
}

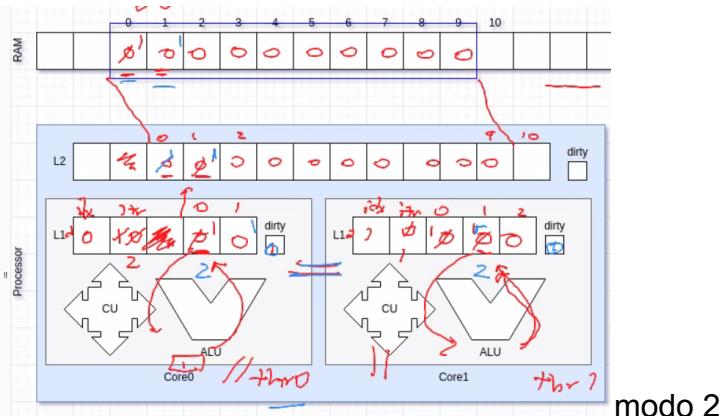
void run_sequential(size_t index0, size_t index1) {
    update_element((void*) index0);
    update_element((void*) index1);
}

void run_concurrent(size_t index0, size_t index1) {
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, update_element, (void*) index0);
    pthread_create(&thread1, NULL, update_element, (void*) index1);
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
}

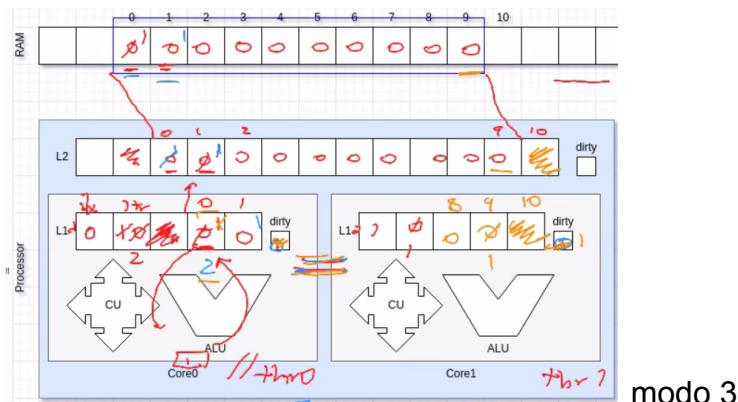
int main(int argc, char* argv[]) {
    const int mode = argc >= 2 ? atoi(argv[1]) : 0;
    array = (int*) calloc(ELEMENT_COUNT, sizeof(int));
    assert(array);

    struct timespec start_time, finish_time;
    clock_gettime(CLOCK_REALTIME, &start_time);
}

```



Para el modo 3 no se bloquea porque no se invalidan/bloquean el uno al otro.



para saber si estamos siendo afectados por cache: perf stat

```

Performance counter stats for 'bin/false_sharing_array 0':

    187,731      cache-references
    95,843      cache-misses          #  51.053 % of all cache refs
 7,350,849,482      cycles
 3,202,813,659      instructions        #  0.44  insn per cycle
   400,646,819      branches
    63      faults
     0      migrations

2.018759408 seconds time elapsed

2.016792000 seconds user
0.000000000 seconds sys

```

mode 0

```

Performance counter stats for 'bin/false_sharing_array 1':

    187,481      cache-references
    89,066      cache-misses          #  47.507 % of all cache refs
 7,281,235,155      cycles
 3,202,767,350      instructions        #  0.44  insn per cycle
   400,630,412      branches
    63      faults
    0      migrations

2.008112979 seconds time elapsed

2.007935000 seconds user
0.000000000 seconds sys

```

mode 1

```

Performance counter stats for 'bin/false_sharing_array 2':

    102,966,615      cache-references
    159,044      cache-misses          #  0.154 % of all cache refs
 10,068,478,851      cycles
 3,204,412,675      instructions        #  0.32  insn per cycle
   400,972,026      branches
    67      faults
     6      migrations

1.390265220 seconds time elapsed

2.757885000 seconds user
0.012008000 seconds sys

```

mode 2

```

Performance counter stats for 'bin/false_sharing_array 3':

    277,864      cache-references
    126,675      cache-misses          #  45.589 % of all cache refs
 7,304,816,379      cycles
 3,203,374,432      instructions        #  0.44  insn per cycle
   400,739,293      branches
    68      faults
     0      migrations

1.004185923 seconds time elapsed

1.993856000 seconds user
0.004003000 seconds sys

```

mode 3 (menos miss)

valgrind, perf pueden hacer gráficos

Lidiar con false sharing:

No podemos actuar sobre el hardware pero podríamos decidir que los hilos trabajen en dos memorias distintas para que no genere bloqueos a la hora de modificar el dirty bit. Con otro tipo de mapeo (cíclico es el peor caso), bloque es relativamente bueno dependiendo del tamaño de caché (optimización). Dinámico es muy variado...

Por bloque reduce la probabilidad de que se dé false sharing.

Con una variable privada para evitar el bloqueo.

Reentrant:

Se define en función de invocar a la subrutina dos veces en un mismo hilo.
si una invocación afecta a otra NO es reentrante.

thread-safe:

cuando no tiene condición de carrera.

findoc