# Sistemas Operativos 2/2020
# Laboratorio 3

Profesores:
Cristóbal Acosta (cristobal.acosta@usach.cl)
Fernando Rannou (fernando.rannou@usach.cl)
Ayudantes:
Benjamín Muñoz (benjamin.munoz.t@usach.cl)
Marcela Rivera (marcela.rivera.c@usach.cl)

## I. Objetivos Generales

Este laboratorio tiene como objetivo aplicar los conceptos de concurrencia de *pthread*, tales como hebras, semáforos y barreras. La aplicación debe ser escrita en el lenguaje de programación C sobre sistema operativo Linux.

## II. Objetivos Específicos

1. Conocer y usar las funcionalidades de `getopt()` como método de recepción de parámetros de entradas.
2. Validar los parámetros recibidos con `getopt()`
3. Construir funciones de lectura y escritura de archivos .csv usando `fopen()`, `fread()`, y `fwrite()`.
4. Construir una función que permita la búsqueda de puntos en un radio.
5. Practicar técnicas de documentación de programas.
6. Conocer y practicar uso de makefile para compilación de programas.
7. Utilizar recursos de *pthread* para sincronizar las hebras y proteger los recursos compartidos.

## III. Conceptos

### III.A. Concurrencia y Sincronización

Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades.

De no existir una sincronización, es posible sufrir una corrupción en los recursos compartidos u obtener soluciones incorrectas.

### III.B. Sección Crítica

Porción de código que se ejecuta de forma concurrente y podría generar conflicto en la consistencia de datos debido al uso de variables globales.

### III.C. Mutex

Provee exclusión mutua, permitiendo que sólo una hebra a la vez ejecute la sección crítica.

### III.D. Hebras

Los hilos POSIX, usualmente denominados pthreads, son un modelo de ejecución que existe independientemente de un lenguaje, además es un modelo de ejecución en paralelo. Éstos permiten que un programa controle múltiples flujos de trabajo que se superponen en el tiempo.

Para poder utilizar hebras, es necesario incluir la librería **pthread.h**. Por otro lado, dentro de la función main, se debe instanciar la variable de referencia a las hebras, para esto se utiliza el tipo de dato **pthread_t** acompañado del nombre de variable.

Luego el código que ejecutarán las hebras se debe construir en una función de la forma:

```
void * function (void * params)
```

La cual recibe parámetros del tipo void, por lo cual es necesario castear el o los parámetros de entrada para así poder utilizarlos sin problemas.

Algunas funciones para manejar las hebras son:

- **pthread_create:** función que crea una hebra. Recibe como parámetros de entrada:
  - La variable de referencia a la hebra que desea crear.
  - Los atributos de éste, los cuales no es obligación de modificar, por lo que en caso de no querer hacerlo, simplemente se deja NULL.
  - El nombre de la función que la hebra ejecutará (la cual debe cumplir con la descripción antes mencionada)
  - Por último, los parámetros de entrada (de la función que se ejecutará) previamente casteados.

  Un ejemplo sería:

```
while(i < numeroHebras)
{
    pthread_create(&hilo[i], NULL, escalaGris, (void *) &structHebra[i].id);
    i++;
}
```

- **pthread_join:** función donde la hebra que la ejecuta, espera por las hebras que se ingresan por parámetro de entrada.

  Un ejemplo sería:

```
while(i < numeroHebras)
{
    pthread_join(hilo[i],NULL);
    i++;
}
```

- **pthread_mutex_init:** función que inicializa un mutex, pasando por parámetros la referencia al mutex, y los atributos con que se inicializa.

  Para inicializar la estructura se utiliza:

```
while(i < numeroHebras)
{
    pthread_mutex_init(&MutexAcumulador, NULL);
    i++;
}
```

Donde **MutexAcumulador**, es una variable (de tipo `pthread_mutex_t`) que representa un mutex, el cual permitirá implementar exclusión mutua a una sección crítica que será ejecutada por varias hebras.

- **pthread_mutex_lock:** entrega una solución a la sección crítica. Ésta recibe como parámetros la variable que se desea bloquear para el resto de hebras. Un ejemplo de uso sería:

```
pthread_mutex_lock(&MutexAcumulador);
```

  Cabe destacar que la primera hebra en ejecutar **pthread_mutex_lock** podrá ingresar a la sección crítica, el resto de hebras quedarán bloqueadas a la espera de que se libere el mutex.

- **pthread_mutex_unlock:** permite liberar una sección crítica. Ésta recibe como parámetro de entrada, la variable que se desea desbloquear. Su implementación es la siguiente:

```
pthread_mutex_unlock(&MutexAcumulador);
```

### III.E. Monitores

Los monitores fueron construidos como una forma de subsanar las desventajas de la implementación de semáforos, los cuales no es sencillo ver el efecto global de sus operaciones (wait y signal) sobre los semáforos y sección de código que afectan. El monitor provee una funcionalidad equivalente a la de los semáforos pero es más fácil de controlar.

Un monitor soporta la sincronización mediante el uso de variables de condición que están contenidas dentro del monitor y son accesibles sólo desde el monitor; se manipulan mediante dos funciones:

- **cwait(c):** Suspende la ejecución del proceso llamante en la condición c. El monitor queda disponible para ser usado por otro proceso.

- **csignal(c):** Retoma la ejecución de algún proceso bloqueado por un cwait en la misma condición. Si hay varios procesos, elige uno de ellos; si no hay ninguno, no hace nada.

Aunque un proceso puede entrar en el monitor invocando cualquiera de sus procedimientos, puede entenderse que el monitor tiene un único punto de entrada que es el protegido, de ahí que sólo un proceso pueda estar en el monitor a la vez. Otros procesos que intenten entrar en el monitor se unirán a una cola de procesos bloqueados esperando por la disponibilidad del monitor.

## IV. Enunciado

### IV.A. Hoyos negros, discos proto planetarios y ALMA

Hace un año aproximadamente los científicos han logrado reconstruir la primera imagen del horizonte de eventos de un hoyo negro. Esta frontera marca el lugar desde donde el cual ya nada puede salir del hoyo negro, ni materia ni luz. Es por eso que la imagen muestra la luz, radiación fuera de la frontera y dentro de la frontera sólo se ve oscuridad. La figura 1 muestra la primera imagen de un hoyo negro:

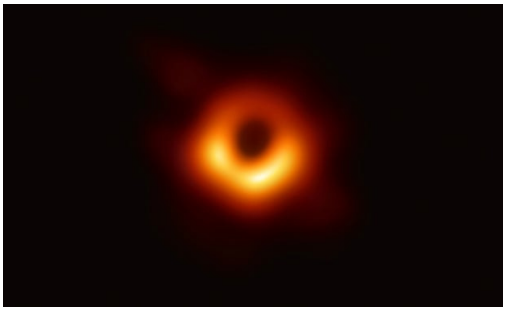

**Figure 1. Primera imagen de un hoyo negro.**

Esta imagen es el resultado de un proceso computacional llamado síntesis o reconstrucción de imágenes interferométricas. Es decir, las antenas no capturan la imagen que nosotros vemos, sino detectan señales de radio, las cuales son transformadas por un programa computacional en la imagen visual.

En estricto rigor los datos recolectados por las antenas corresponden a muestras del espacio Fourier de la imagen. Estas muestras están repartidas en forma no uniforme e irregular en el plano Fourier. La figura 2 muestra un típico patrón de muestreo del plano Fourier. La imagen de la derecha muestra la imagen reconstruida a partir de los datos. Esta imagen corresponde a un disco protoplanetario llamado HLTau.

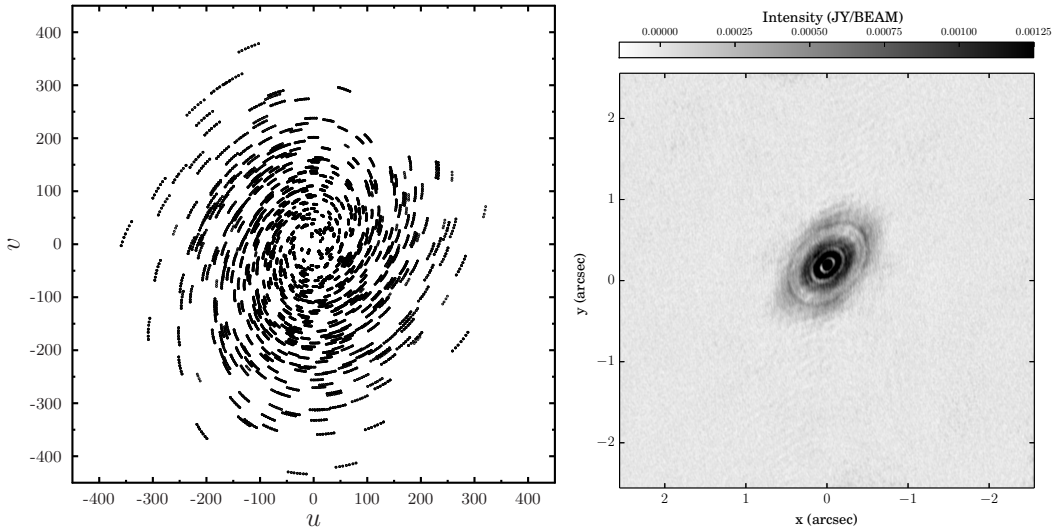


**Figure 2. Plano $(u, v)$ y imagen sintetizada de HLTau.**

En cada punto $(u, v)$ se mide una señal llamada Visibilidad. Cada visibilidad es un número complejo, es decir posee una parte real y otra imaginaria. La imagen por otro lado es solo real.

Usaremos la siguiente notación para describir los datos:

- $V(u, v)$ es la visibilidad en la coordenada $(u, v)$ del plano Fourier
- $V(u, v).r$ es la parte real de la visibilidad
- $V(u, v).i$ es la parte imaginaria de la visibilidad
- $V(u, v).w$ es el ruido en la coordenada $(u, v)$

La siguiente lista es un ejemplo de una pequeña muestra de visibilidades

```
46.75563,-160.447845,-0.014992,0.005196,0.005011
119.08387,-30.927526,0.016286,-0.001052,0.005888
-132.906616,58.374054,-0.009442,-0.001208,0.003696
-180.271179,-43.749226,-0.011654,0.001075,0.003039
-30.299767,-126.668739,0.015222,-0.004145,0.007097
-18.289482,28.76403,0.025931,0.001565,0.004362
```

La primera columna es la posición en el eje $u$; la segunda es la posición en el eje $v$. la tercera columna es el valor real de la visibilidad, la cuarta, el valor imaginario y finalmente, la quinta es el ruido. Estos datos corresponden a datos reales de un disco proto planetario captadas por el observatorio ALMA.

El número de visibilidades depende de cada captura de datos y del telescopio usado, pero puede variar entre varios cientos de miles de puntos hasta cientos de millones. Por lo tanto, el procesamiento de las visibilidades es una tarea computacionalmente costosa y generalmente se usa paralelismo para acelerar los procesos. En este lab, simularemos este proceso usando varias hebras que cooperan para hacer cálculo a las visibilidades. Por ahora, no haremos síntesis de imágenes.

### IV.B. Cálculo de propiedades

En ciertas aplicaciones, antes de sintetizar la imagen, es necesario y útil extraer características del plano Fourier. Uno de estos preprocesamiento, agrupa las visibilidades en anillos concéntricos con centro en el $(0, 0)$ y radios crecientes. Por ejemplo, suponga que definimos un radio $r$, el cual divide las visibilidades en dos anillos, aquellas visibilidades cuya distancia al centro es menor o igual a $R$, y aquellas cuya distancia al centro es mayor a $R$. La distancia de una visibilidad al centro es simplemente:

$$d(u, v) = (u^2 + v^2)^{1/2}$$

Si definieramos dos radios $R_1$ y $R_2$ ($R_1 < R_2$), las visibilidades se dividirían en tres discos: $0 \leq d(u, v) < R_1$, $R_1 \leq d(u, v) < R2$, y $R_2 \leq d(u, v)$.

Para este lab, las propiedades que se calcularán por disco son

1. Media real : $\frac{1}{N} \sum V(u, v).r$, donde $N$ es el número de visibilidades en el disco
2. Media imaginaria : $\frac{1}{N} \sum V(u, v).i$, donde $N$ es el número de visibilidades en el disco
3. Potencia: $\sum((V(u, v).r)^2 + (V(u, v).i)^2)^{1/2}$
4. Ruido total: $\sum V(u, v).w$

## V. El programa del lab

### V.A. Lógica de solución

En este laboratorio crearemos un conjunto de procesos que calculen las propiedades antes descritas. Usted debe organizar su solución de la siguiente manera.

1. El proceso principal recibirá argumentos por línea de comando el número de radios en los que se debe dividir el plano de Fourier. Además recibirá el ancho $\Delta_r$ de cada intervalo, tal que $R_1 = \Delta_r$, $R_2 = 2\Delta_r$, etc. Este proceso también recibe como argumento el nombre del archivo de entrada, el nombre del archivo de salida y el tamaño del búffer de cada monitor.
2. El proceso principal debe crear tantos monitores y hebras como número de discos hayan sido especificados (una hebra por cada disco). Además de mutex, variables de condición y cualquier herramienta de pthreads que sea necesaria para proveer exclusión mutua y sincronización.
3. El proceso principal (hebra padre) debe instanciar una estructura común en la cual todas las hebras deberán escribir los resultados de las propiedades.
4. El proceso principal leerá línea a línea el archivo de entrada con formato .csv y determinará el disco al que pertenece la visibilidad, y utilizando un monitor, pondrá a disposición de cada hebra los datos que le correspondan, en función del tamaño de búffer ingresado como parámetro.
5. Cada hebra deberá recoger los datos que le correspondan cuando el búffer se encuentre lleno, además debe calcular parcialmente las propiedades.
6. Cuando el proceso principal termine de leer y enviar todas las visibilidades, debe indicar que en caso de no encontrarse llenos los búffer, como ya se acabaron los datos, entonces las hebras de todas formas pueden recoger lo que haya en los búffer (evita deadlock).
7. Una vez que se hayan procesado todos los datos, cada hebra debe escribir en la estructura común los resultados finales de las propiedades.
8. El proceso principal lee los resultados de la estructura común y escribe el archivo de salida final.

Para implementar esta lógica, se debe tener un programa. Este será `lab2.c` el cual será el proceso padre y por lo tanto se encargará de las funciones que le corresponde (previamente descritas), como por ejemplo leer el archivo y crear hebras, para luego distribuir los datos según el radio entre cada hebra. Esto se hará implementando un monitor por cada disco y llenando un búffer también por cada disco con

todos los datos correspondientes. Por ejemplo, si el búffer es igual a 10, entonces se deben recolectar 10 datos correspondientes al disco 1 para que recién en ese momento la hebra correspondiente al disco 1 pueda recoger dichos datos y calcular parcialmente las propiedades pedidas.

Por otro lado, las hebras deben recoger los datos correspondientes a su disco según el tamaño del búffer y mediante acumuladores calcular parcialmente las propiedades. Las hebras además esperarán un mensaje de término, para que no se produzca deadlock en caso de que nunca se llegue a completar el búffer del monitor, el cual indicará que ya pueden calcular las propiedades finales. Las propiedades finales deben ser guardadas en una estructura común (por lo tanto debe protegerse con exclusión mutua) y finalmente el proceso principal debe leer los resultados en esta estructura común y escribirlos en un archivo de salida.

El archivo que debe escribir el proceso principal con las propiedades calculadas por las hebras, debe tener el siguiente formato:


**Figure 3. Ejemplo archivo de salida.**

Cabe destacar que se espera un resultado en cuanto al cálculo de propiedades igual a la experiencia anterior.
En donde el nombre del archivo es el nombre entregado como argumento (en este caso: propiedades.txt), además se incluyen todas las propiedades que cada hijo debe calcular en su disco asignado según el ancho y el número de discos también ingresados como argumentos (en este caso: 3 discos). El formato debe indicar cada disco de forma ordenada y luego mostrar sus propiedades indicando el nombre de estas y su resultado.

El programa se ejecutará usando los siguientes argumentos (ejemplo):

```
$ ./lab2 -i visibilidades.csv -o propiedades.txt -d ancho -n ndiscos -s tamañobuffer -b
```

- **-i**: nombre de archivo con visibilidades, el cual tiene como formato uv_values_i.csv con i = 0,1,2,....
- **-o**: nombre de archivo de salida
- **-n**: cantidad de discos
- **-d**: ancho de cada disco
- **-s**: tamaño del buffer de cada monitor
- **-b**: bandera o flag que permite indicar si se quiere ver por consola la cantidad de visibilidades encontradas por cada proceso hijo. Es decir, si se indica el flag, entonces se muestra la salida por consola.

Ejemplo de compilación y ejecución:

```
>> gcc lab2.c -o lab2
>> ./lab2 -i uvplaneco65.csv -o propiedades.txt -d 100 -n 4 -s 10 -b
```

En el primer caso, se ejecuta el programa entregando como nombre de archivo de entrada `uvplaneco65.csv`, además el proceso principal deberá crear 4 hebras y cada hebra recogerá los datos que le corresponden y le entrega el proceso principal cuando se llene el búffer de tamaño 10 (es decir, puede guardar 10 filas de datos).

Las visibilidades se repartirán entre los siguientes rangos: $[0, 100)$, $[100, 200)$, $[200, 300)$, $[300, )$. Finalmente, debido a que el primer caso considera el flag -b, cada hebra debe identificarse y mostrar la cantidad de visibilidades que procesó. Ejemplo:

```
Soy la hebra 1, procesé 10000 visibilidades
Soy la hebra 2, procesé 14000 visibilidades
...
```

El segundo caso, funciona igual al anterior, siendo la única diferencia que las hebras no deben mostrar ningún tipo de salida por consola.

Como requerimientos no funcionales, se exige lo siguiente:

- Debe funcionar en sistemas operativos con kernel Linux.
- Debe ser implementado en lenguaje de programación C.
- Se debe utilizar un archivo Makefile para su compilación.
- Realizar el programa utilizando buenas prácticas, dado que este laboratorio no contiene manual de usuario ni informe, es necesario que todo esté debidamente comentado.
- Que el programa principal esté desacoplado, es decir, que se desarrollen las funciones correspondientes en otro archivo .c para mayor entendimiento de la ejecución.
- Se deben verificar y validar los parámetros de entrada.

## VI. Entregables

El laboratorio es en parejas y se descontará 1 punto por día de atraso. Cabe destacar que no entregar un laboratorio, implica la reprobación de la asignatura. FInalmente, debe subir en un archivo comprimido a usachvirtual los siguientes entregables:

- `Makefile`: Archivo para make que compila el programa.
- `archivos .c y .h` Se debe tener como mínimo un archivo .c principal con el main del programa. Se debe tener mínimo un archivo .h que tenga cabeceras de funciones, estructuras o datos globales. Se deben comentar todas las funciones de la forma:

```
//Entradas: explicar que se recibe
//Funcionamiento: explicar que hace
//Salidas: explicar que se retorna
```

- Trabajos con códigos que hayan sido copiados de un trabajo de otro grupo serán calificados con la nota mínima.

El archivo comprimido debe llamarse: RUTESTUDIANTE1_RUTESTUDIANTE2.zip
Ejemplo: 19689333k_186593220.zip
NOTA: los laboratorios son en parejas, las cuales deben ser de la misma sección. De lo contrario no se revisarán laboratorios.

## VII. Fecha de entrega

Domingo 3 de Enero