Documentación Taller POSIX Sincronización

Guillermo Andrés Aponte Cárdenas

Daniel Felipe Ramirez Vargas

Pontificia Universidad Javeriana

Sistemas Operativos

Bogotá, 2025

Marco Teórico

Intercomunicación de Procesos

(Silberschatz et al., 2020) “Los procesos que se ejecutan simultáneamente en el sistema operativo pueden ser independientes o cooperantes. Un proceso es independiente si no comparte datos con otros procesos que se ejecutan en el sistema. Un proceso coopera si puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema. Claramente, cualquier proceso que comparte datos con otros procesos es un proceso cooperante.”

Proceso

(Silberschatz et al., 2020) “Un proceso es un programa en ejecución. Un proceso necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos de E/S, para realizar su tarea. Estos recursos suelen asignarse al proceso mientras se ejecuta.

Un proceso es la unidad de trabajo en la mayoría de los sistemas. Los sistemas constan de un conjunto de procesos: los procesos del sistema operativo ejecutan el código del sistema y los procesos de usuario ejecutan el código del usuario. Todos estos procesos pueden ejecutarse simultáneamente.

Los sistemas operativos modernos admiten procesos con múltiples hilos de control. En sistemas con múltiples núcleos de procesamiento de hardware, estos hilos pueden ejecutarse en paralelo.

Uno de los aspectos más importantes de un sistema operativo es cómo programa los hilos en los núcleos de procesamiento disponibles. Los programadores disponen de varias opciones para diseñar programadores de CPU.”

Semáforos

(*sem_overview(7) - Linux manual page*, s. f.) “Los semáforos POSIX permiten que los procesos e hilos sincronicen sus acciones.

Un semáforo es un entero cuyo valor nunca puede ser inferior a cero. Se pueden realizar dos operaciones con los semáforos: incrementar su valor en uno (sem_post(3)) y decrementarlo en uno (sem_wait(3)). Si el valor de un semáforo es cero, la operación sem_wait(3) se bloqueará hasta que el valor sea mayor que cero.

Los semáforos POSIX se presentan en dos formas: semáforos con nombre y semáforos sin nombre.”

Semáforos con Nombre

(*sem_overview(7) - Linux manual page*, s. f.) “Un semáforo con nombre se identifica mediante un nombre con el formato /nombre; es decir, una cadena terminada en nulo de hasta NAME_MAX-4 (es decir, 251) caracteres que consta de una barra inicial seguida de uno o más caracteres, ninguno de los cuales es una barra. Dos procesos pueden operar sobre el mismo semáforo con nombre pasando el mismo nombre a sem_open()”.

Memoria Compartida POSIX

(*shm_overview(7) - Linux manual page*, s. f.) “

La API de memoria compartida POSIX permite que los procesos comuniquen información compartiendo una región de memoria”

Función sem_open()

(*sem_open(3) - Linux manual page*, s. f.) “sem_open() crea un nuevo semáforo POSIX o abre uno existente.”

Función sem_post()

(*sem_post(3) - Linux manual page*, s. f.) “`sem_post()` incrementa (desbloquea) el semáforo al que apunta `sem`.

Si el valor del semáforo pasa a ser mayor que cero, entonces otro proceso o hilo bloqueado en una llamada a `sem_wait(3)` se reactivará y procederá a bloquear el semáforo.”

Función sem_wait()

(*sem_wait(3) - Linux manual page*, s. f.) “`sem_wait()` decrementa (bloquea) el semáforo al que apunta `sem`. Si el valor del semáforo es mayor que cero, entonces el decremento se realiza y la función retorna inmediatamente. Si el semáforo actualmente tiene el valor cero, la llamada se bloquea hasta que sea posible realizar el decremento (es decir, el valor del semáforo supere cero) o un manejador de señales interrumpa la llamada.”

Función sem_close()

(*sem_close(3) - Linux manual page*, s. f.) “sem_close() cierra el semáforo especificado por sem, permitiendo que se liberen todos los recursos que el sistema haya asignado al proceso que realiza la llamada para este semáforo.”

Función sem_unlink()

(*sem_unlink(3) - Linux manual page*, s. f.) “sem_unlink() elimina el semáforo especificado por su nombre. El nombre del semáforo se elimina inmediatamente. El semáforo se destruye una vez que todos los demás procesos que lo tienen abierto lo cierran.”

Función shm_open()

(*shm_open(3) - Linux manual page*, s. f.) “shm_open() crea y abre un nuevo objeto de memoria compartida POSIX, o abre uno existente.

Un objeto de memoria compartida POSIX es, en efecto, un identificador que pueden usar procesos no relacionados para mapear con mmap(2) la misma región de memoria compartida.”

Función truncate()

(*truncate(2) - Linux manual page*, s. f.) “Las funciones `truncate()` y `ftruncate()` truncan el archivo regular cuyo nombre se especifica en la ruta o al que hace referencia el descriptor de archivo (fd), a un tamaño de exactamente longitud en bytes. Si el archivo era mayor que este tamaño, los datos adicionales se pierden. Si el archivo era menor, se extiende yla parte extendida se lee como bytes nulos ('\0').”

Función mmap()

(*mmap(2) - Linux manual page*, s. f.) "La función mmap() crea una nueva asignación en el espacio de direcciones virtuales del proceso que la invoca. La dirección inicial de la nueva asignación se especifica en addr. El argumento length especifica la longitud de la asignación (que debe ser mayor que 0)."

Función munmap()

(*mmap(2) - Linux manual page*, s. f.) "La llamada al sistema munmap() elimina las asignaciones para el rango de direcciones especificado.

Esto provoca que cualquier referencia posterior a direcciones dentro de dicho rango genere referencias de memoria no válidas. La región también se desasigna automáticamente cuando finaliza el proceso.

Por otro lado, cerrar el descriptor de archivo no desasigna la región."

Función unlink()

(*unlink(2) - Linux manual page*, s. f.) "La función `unlink()` elimina un nombre del sistema de archivos. Si ese nombre era el último enlace a un archivo y ningún proceso lo tiene abierto, el archivo se elimina y el espacio que utilizaba queda disponible para su reutilización. Si el nombre era el último enlace a un archivo, pero algún proceso aún lo tiene abierto, el archivo permanecerá en el sistema hasta que se cierre el último descriptor de archivo que lo referencia."

Mutex

(AI-FutureSchool, 2025) “Un mutex, o mutua esclusión, es un mecanismo de sincronización utilizado en programación concurrente para gestionar el acceso a recursos compartidos entre múltiples hilos de ejecución. Su principal objetivo es prevenir condiciones de carrera, donde dos o más hilos intentan acceder y modificar un recurso al mismo tiempo, lo que puede resultar en comportamientos inesperados y errores en el programa.”

Cond

(*z/OS*, 2021) “Se bloquea en una variable de condición. Debe llamarse con *el mutex* bloqueado por el hilo que realiza la llamada; de lo contrario, se producirá un comportamiento indefinido. Un mutex se bloquea mediante pthread_mutex_lock().

cond es una variable de condición que comparten los hilos. Para cambiarla, un hilo debe mantener el *mutex* asociado a la variable de condición. La función pthread_cond_wait() libera este *mutex* antes de suspender el hilo y lo obtiene de nuevo antes de regresar.”

Función sprintf()

(*IBM i*, 2025) “La función sprintf() formatea y almacena una serie de caracteres y valores en el *almacenamiento intermedio*de la matriz. Cualquier *lista-argumentos* se convierte y se coloca de acuerdo con la especificación de formato correspondiente en la *serie-formato*.

La *serie-formato* consta de caracteres ordinarios y tiene el mismo formato y función que el argumento *serie-formato* para la función printf().”

Función pthread_create()

(*pthread_create(3) - Linux manual page*, s. f.) “La función pthread_create() inicia un nuevo hilo en el proceso que la llama.”

Función pthread_join()

(*pthread_join(3) - Linux manual page*, s. f.) “La función pthread_join() espera a que finalice el hilo especificado por thread. Si dicho hilo ya ha finalizado, entonces pthread_join() devuelve el control inmediatamente. El hilo especificado por thread debe ser unible para la unión de hilos.”

Función pthread_cancel()

(*pthread_cancel(3) - Linux manual page*, s. f.) “La función pthread_cancel() envía una solicitud de cancelación al hilo.

El hilo de destino reacciona a la solicitud de cancelación, y cuándo lo hace, dependiendo de dos atributos que están bajo su control: su estado de cancelabilidad y su tipo.”

Función pthread_exit()

(*AIX*, 2024) “La subrutina pthread_exit termina la hebra de llamada de forma segura y almacena un estado de terminación para cualquier hebra que pueda unirse a la hebra de llamada. El estado de terminación es siempre un puntero vacío; puede hacer referencia a cualquier tipo de datos.”

Función pthread_mutex_lock()

(*AIX*, 2025) “El objeto de mútex al que hace referencia el parámetro mutex se bloquea llamando a pthread_mutex_lock. Si el mutex ya está bloqueado, la hebra de llamada se bloquea hasta que el mutex esté disponible. Esta operación se devuelve con el objeto mutex al que hace referencia el parámetro mutex en el estado bloqueado con la hebra de llamada como su propietario.”

Función pthread_mutex_unlock()

(*Z/OS*, 2023) “Libera un objeto mutex. Si uno o más hilos están esperando para bloquear el mutex, pthread_mutex_unlock() hace que uno de esos hilos regrese de pthread_mutex_lock() con el objeto mutex adquirido. Si ningún hilo está esperando el mutex, el mutex se desbloquea sin propietario actual”

Función pthread_cond_wait()

(*pthread_cond_wait*, s. f.) “Las funciones pthread_cond_wait() y pthread_cond_timedwait() se utilizan para bloquearse en una variable de condición. Se llaman con el mutex bloqueado por el hilo que realiza la llamada; de lo contrario, se producirá un comportamiento indefinido.

Estas funciones liberan el mutex de forma atómica y hacen que el hilo que realiza la llamada se bloquee en la variable de condición cond ; atómicamente aquí significa "atómicamente con respecto al acceso de otro hilo al mutex y luego a la variable de condición". Es decir, si otro hilo puede adquirir el mutex después de que el hilo que está a punto de bloquearse lo haya liberado, entonces una llamada posterior

a pthread_cond_signal() o pthread_cond_broadcast() en ese hilo se comporta como si se hubiera emitido después de que el hilo que está a punto de bloquearse se haya bloqueado".

Función pthread_cond_signal()

(*pthread_cond_signal*, s. f.) "Se utiliza para desbloquear hilos bloqueados en una variable de condición.

La llamada a *pthread_cond_signal()* desbloquea al menos uno de los hilos que están bloqueados en la variable de condición especificada *cond* (si algún hilo está bloqueado en *cond*).

Si más de un hilo está bloqueado en una variable de condición, la política de planificación determina el orden en que se desbloquean los hilos. Cuando cada hilo desbloqueado como resultado de una llamada a ` pthread_cond_signal() `
o `pthread_cond_broadcast()` regresa de su llamada
a `pthread_cond_wait()` o `pthread_cond_timedwait()`"

Memoría Dinámica

(*Manejo de Punteros y Memoria Dinámica | Sistemas Operativos - UTN FRBA*, s. f.) "La memoria dinámica es memoria que se reserva en *tiempo de ejecución* y se aloja en el heap del proceso (los datos apuntados por los punteros). En C, el programador deberá reservar dicha memoria para su uso y también tendrá la responsabilidad de liberarla cuando no la utilice más.

Una diferencia importante es que el tamaño de la memoria dinámica se puede ir modificando durante la ejecución del programa. ¿Qué quiere decir ésto? Que, por ejemplo, podrías ir agrandando/achicando una determinada estructura (por ejemplo, un array) a medida que lo necesitas.

Algunas ventajas que ofrece la memoria dinámica frente a la memoria estática es que podemos reservar espacio para variables de tamaño no conocido hasta el momento de la ejecución (por ejemplo, para listas o arrays de tamaños variables), o bloques de memoria que, mientras mantengamos alguna referencia a él, pueden sobrevivir al bloque de código que lo creó."

Función malloc()

(*malloc(3) - Linux manual page*, s. f.) "La función malloc() asigna size bytes y devuelve un puntero a la memoria asignada. La memoria no se inicializa. Si size es 0, entonces malloc() devuelve un valor de puntero único que posteriormente se puede pasar correctamente a free(). (Consulte "Comportamiento no portátil" para obtener más información sobre problemas de portabilidad)."

Función free()

(*malloc(3) - Linux manual page*, s. f.) "La función `free()` libera el espacio de memoria al que apunta `p`, que debe haber sido devuelto por una llamada previa a `malloc()` o funciones relacionadas. De lo contrario, o si `p` ya se ha liberado, se produce un comportamiento indefinido. Si `p` es `NULL`, no se realiza ninguna operación."

Actividad 1

En la Actividad 1 se implementó un programa del tipo Productor–Consumidor utilizando IPC POSIX, específicamente, Named Semaphores y memoria compartida. El objetivo fue permitir que dos procesos completamente independientes, un productor y un consumidor, pudieran intercambiar datos sin necesidad de tener una relación padre-hijo como en fork(). La solución empleó un buffer almacenado en un objeto de memoria compartida, mientras que dos semáforos (/vacio y /lleno) sincronizaron el flujo de producción y consumo asegurando que no se perdieran datos y que no se produjeran condiciones de carrera. El productor creó los semáforos y la memoria compartida, produjo diez elementos en el buffer con un ciclo for y se utilizó sem_wait y sem_post para coordinar el acceso. El consumidor, por su parte, abrió esos mismos recursos y consumió los elementos del buffer para finalmente, que ambos procesos liberaran y eliminaron los semáforos y la memoria compartida del sistema.

Esta actividad, nos permitió comprender de forma práctica cómo los procesos independientes pueden usar IPC. La experiencia reforzó saberes fundamentales de sincronización, gestión de recursos y diseño de arquitecturas de comunicación entre procesos.

Aspectos importantes:

- Uso de named semaphores para IPC
- Memoria compartida POSIX con shm y mmap
- Manejo y liberación de recursos (shm_unlink, sem_unlink, etc)

Actividad 2

En esta actividad se crearon diez hilos productores encargados de escribir mensajes en un búfer de cadenas y un hilo spooler que imprimió esos mensajes en orden. Los hilos compartieron el mismo espacio de memoria del proceso, por lo que fue necesario usar exclusión mutua y espera condicional para evitar condiciones de carrera. El mutex buf_mutex protegió toda la sección crítica, mientras que las variables de condición buf_cond y spool_cond permitieron coordinar cuándo producir y cuándo imprimir, evitando el uso de espera indefinida. Finalmente, se gestiona la cancelación del hilo spooler una vez no quedaban líneas por imprimir. Esta actividad nos permitió aplicar los principios de sincronización dentro de un único proceso con múltiples hilos, reforzando conceptos asociados a mutex, esperas condicionales y comunicaciones internas al proceso. Este ejercicio fortaleció habilidades clave para construir software concurrente eficiente y seguro, especialmente en entornos donde múltiples hilos deben coordinarse sobre la misma memoria evitando condiciones de carrera y deadlocks.

Aspectos importantes:

- Uso de multihilo en un único proceso
- Uso de mutexes como mecanismo de exclusión mutua
- Uso de variables de condición para coordinación de hilos
- Cancelación de hilos

BONO

El Bono abordó el problema de calcular el máximo valor de un vector de enteros utilizando paralelismo con pthread POSIX. En lugar de sincronización compleja, este ejercicio se centra en dividir el trabajo entre varios hilos, cada uno operando sobre un segmento independiente del vector. Cada hilo calculó un máximo parcial, y al finalizar todos, el hilo principal realizó una reducción para obtener el máximo global. No se emplearon mutexes ni variables de condición porque no había memoria compartida que pudiera causar conflictos de escritura ya que cada hilo operó únicamente en su propio rango definido en la llamada a su función. El main gestionó la lectura del archivo, la reserva dinámica de memoria, la creación de hilos y la espera de estos. Esta última actividad demostró de forma práctica cómo aprovechar el paralelismo con hilos POSIX para acelerar tareas computacionales intensivas (como el taller de rendimiento). La experiencia reforzó conceptos fundamentales de segmentación de datos, creación y gestión de hilos, y reducción de resultados parciales. El ejercicio también evidenció la

importancia de validar parámetros y de diseñar estructuras de datos adaptadas al paralelismo. Esta actividad consolidó los saberes adquiridos sobre concurrencia y paralelismo, permitiendo apreciar cómo los hilos pueden mejorar el rendimiento cuando las tareas se distribuyen adecuadamente.

Aspectos importantes:

- Segmentación de datos (división del vector)
- Ejecución paralela sin necesidad de sincronización
- Uso de funciones para gestionar el comportamiento de los hilos (create, join, exit, etc)
- Gestión de memoria

Bibliografía

AI-FutureSchool. (2025, enero 4). *Mutex en programación: Conceptos y mejores prácticas*. AI-FutureSchool. https://www.ai-futureschool.com/es/informatica/mutex-sincronizacion-en-programacion.php

AIX. (2024, diciembre 5). https://www.ibm.com/docs/es/aix/7.2.0?topic=p-pthread-exit-subroutine

AIX. (2025, julio 11). https://www.ibm.com/docs/es/aix/7.3.0?topic=p-pthread-mutex-lock-pthread-mutex-trylock-pthread-mutex-unlock-subroutine

IBM i. (2025, septiembre 29). https://www.ibm.com/docs/es/i/7.5.0?topic=functions-sprintf-print-formatted-data-buffer

malloc(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/free.3.html

Manejo de Punteros y Memoria Dinámica | Sistemas Operativos—UTN FRBA. (s. f.). Recuperado 13 de noviembre de 2025, de https://docs.utnso.com.ar/guias/programacion/punteros

mmap(2)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/munmap.2.html

pthread_cancel(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://www.man7.org/linux/man-pages/man3/pthread_cancel.3.html

Pthread_cond_signal. (s. f.). Recuperado 13 de noviembre de 2025, de https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_signal.html

Pthread_cond_wait. (s. f.). Recuperado 13 de noviembre de 2025, de https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_wait.html

pthread_create(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/pthread_create.3.html

pthread_join(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/pthread_join.3.html

sem_close(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_close.3.html

sem_open(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_open.3.html

sem_overview(7)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man7/sem_overview.7.html

sem_post(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_post.3.html

sem_unlink(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_unlink.3.html

sem_wait(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/sem_wait.3.html

shm_open(3)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man3/shm_open.3.html

shm_overview(7)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man7/shm_overview.7.html

Silberschatz, A., Galvin, P., & Gagne, G. (2020). *Operating System Concepts* (Tenth). Wiley.

truncate(2)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/ftruncate.2.html

unlink(2)—Linux manual page. (s. f.). Recuperado 13 de noviembre de 2025, de https://man7.org/linux/man-pages/man2/unlink.2.html

Z/OS. (2021, junio 25). https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-pthread-cond-wait-wait-condition-variable

Z/OS. (2023, abril 28). https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-pthread-mutex-unlock-unlock-mutex-object