

Interprocess Communication (IPC)

Desarrollo de Sistemas en Red



Proceso

- Una **instancia** de un programa en ejecución.
- El kernel es responsable de **administrar y compartir** los recursos de la computadora entre todos los procesos en ejecución:
 - **Recursos limitados:** Se le asigna una cantidad y se ajusta durante la vida del proceso con base a la demanda del recurso tanto del proceso mismo como del sistema. Ej. Memoria RAM.
 - **Recursos ilimitados:** Se deben compartir equitativamente entre todos los procesos. Ej. Tiempo de CPU y ancho de banda

Diferencia entre programa y proceso

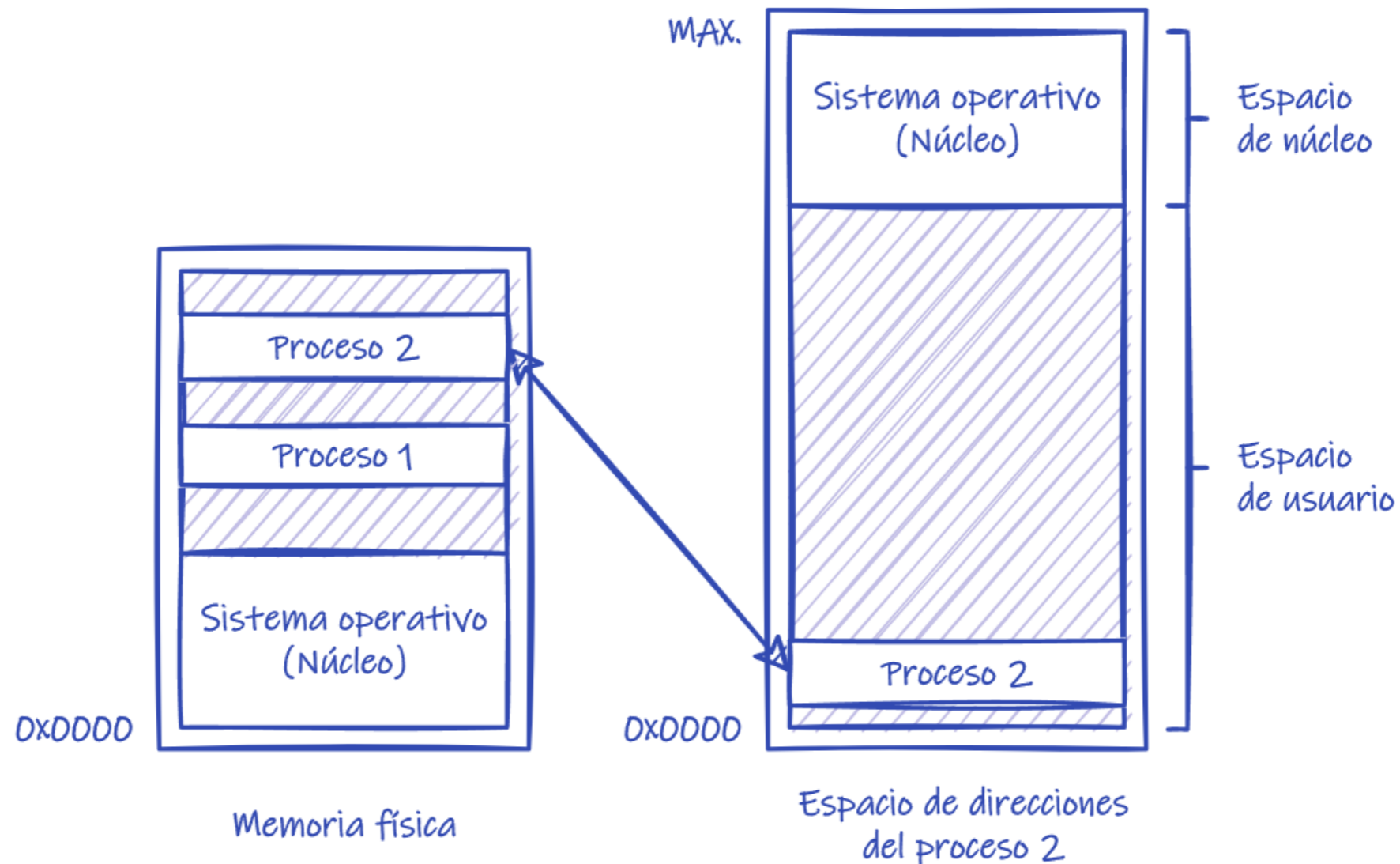
Aspecto	Programa	Proceso
Definición	Conjunto de instrucciones escritas en un lenguaje de programación.	Ejecución de un programa en un sistema operativo.
Naturaleza	Es un ente pasivo, un archivo almacenado en disco.	Es un ente activo, en ejecución en la memoria.
Estado	No cambia mientras no se modifique el archivo.	Cambia dinámicamente (puede estar listo, en ejecución, en espera, terminado).
Almacenamiento	Se guarda como archivo en disco (ejemplo: .exe, .py, .class).	Reside en memoria principal y tiene asignados recursos del sistema.
Recursos	No utiliza recursos hasta ser ejecutado.	Consume CPU, memoria, archivos abiertos, dispositivos de E/S, etc.
Identificación	No tiene identificador único en el sistema.	Tiene un identificador único: PID (Process ID).
Multiplicidad	Un mismo programa puede ser ejecutado varias veces.	Cada ejecución corresponde a un proceso distinto, aunque provenga del mismo programa.

Actividades del Kernel al ejecutar un programa

1. Carga el código del programa en la memoria virtual
2. Asigna espacio para las variables del programa; y
3. Configura estructuras de datos para registrar información sobre el proceso
 - ID, estatus de terminación, ID de usuario e ID de grupo

Modelo de programación de Unix

- Múltiples procesos ejecutándose
- Proceso con su propio espacio de direcciones



- Si un proceso requiere alguna entidad, el sistema operativo lo trata como un recurso:
 - Archivos, memoria, procesador, disco, etc.
- Un proceso se suspende hasta que se le asigna el recurso que necesita

Diseño de la memoria de un proceso

Stack: Se usa para almacenar variables locales e información de enlace a funciones

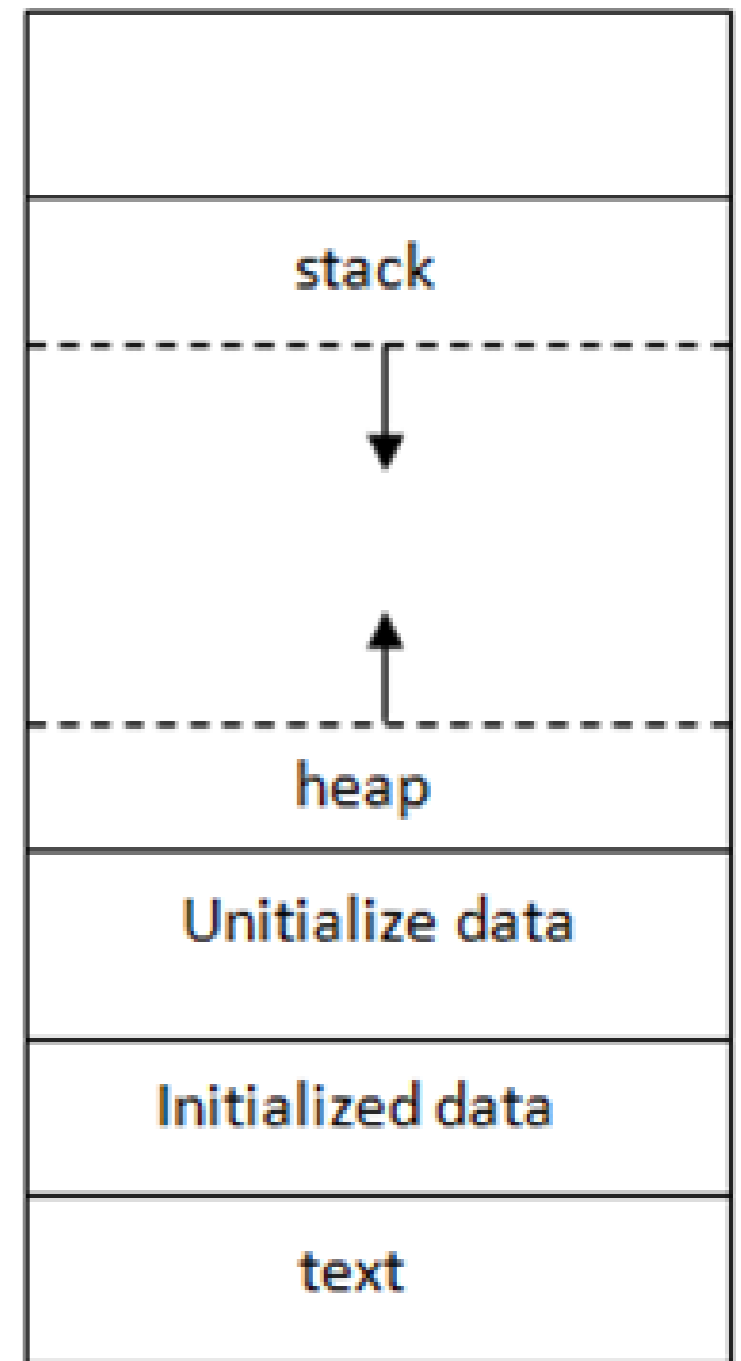
Heap: Área de memoria en donde los programas pueden asignar dinámicamente memoria extra.

Datos: Aquí se almacenan las variables globales y estáticas utilizadas por el programa.

Texto: Las instrucciones del programa para ser consumidas por el CPU. Solo lectura.

High address

Low address



- Una forma para que los procesos se comuniquen es leyendo y escribiendo información en archivos de disco.
- Para muchas aplicaciones, esto es demasiado lento e inflexible.
- Linux proporciona un amplio conjunto de mecanismos para la comunicación entre procesos (IPC)

- **Señales:** Se utilizan para indicar que ha ocurrido un evento
- **Pipes:** Se pueden usar para transferir datos entre procesos
- **Sockets:** Se pueden usar para transferir datos de un proceso a otro, ya sea en la misma computadora host o en diferentes hosts conectados por una red
- **Bloqueo de archivos:** Permite que un proceso bloquee regiones de un archivo para evitar que otros procesos lean o actualicen el contenido del archivo

- **Colas de mensajes:** Se utilizan para intercambiar mensajes (paquetes de datos) entre procesos
- **Semáforos:** Se utilizan para sincronizar las acciones de los procesos; y
- **Memoria compartida:** Permite que dos o más procesos compartan memoria
 - Cuando un proceso cambia el contenido de la memoria compartida, todos los demás procesos pueden ver los cambios de inmediato

Señales

- Se emplean en diferentes contextos además de IPC
- Se les conoce como *interrupciones por software*
- La llegada de una señal informa a un proceso que se ha producido algún evento o condición excepcional
- Hay varios tipos de señales, cada una identifica un evento o condición diferente

- Cada tipo de señal se identifica por un número entero diferente
- Las señales son enviadas a un proceso por: el kernel, por otro proceso o por el proceso mismo

Ejemplo: El kernel puede enviar una señal a un proceso cuando ocurre uno de los siguientes:

- a) El usuario escribió el carácter de interrupción (^-C) en el teclado
- b) Uno de los hijos del proceso ha finalizado
- c) Un temporizador establecido por el proceso ha expirado; o
- d) El proceso intentó acceder a una dirección de memoria no válida

- Dentro del shell, el comando kill se usa para enviar una señal a un proceso
- La llamada al sistema kill() proporciona la misma facilidad dentro de los programas

Cuando un proceso recibe una señal, toma una de las siguientes acciones:

- Ignora la señal
- Se termina por la señal; o
- Se suspende hasta que se reciba una señal

Para la mayoría de los tipos de señal, en lugar de aceptar la acción de señal predeterminada, un programa puede:

- Elegir ignorar la señal; o
- Establecer un manejador de señales

Un **manejador de señales** es una función definida por el programador que se invoca automáticamente cuando la señal se entrega al proceso

- Realiza alguna acción apropiada a la condición que generó la señal
- En el tiempo entre que se genera y entrega una señal se dice que ésta está pendiente
- Se entrega una señal pendiente tan pronto como el proceso de recepción esté programado para ejecutarse o inmediatamente si ya se está ejecutando

Hilos o subprocessos

- En UNIX, cada proceso puede tener múltiples hilos de ejecución.
- Se puede ver a los hilos como un conjunto de procesos que comparten la misma memoria virtual.
- Cada subprocesso:
 - Ejecuta el mismo código y comparte la misma área de datos y *heap*
 - Tiene su propia pila con variables locales e información de enlace de llamada a funciones.

- Los hilos pueden comunicarse entre sí a través de las variables globales que comparten
- La API de subprocessos proporciona **variables de condición y mutexes:**
 - Primitivas que permiten a los hilos comunicarse y sincronizar sus acciones, en particular, su uso de variables compartidas
- Los hilos también pueden comunicarse entre sí utilizando IPC y los mecanismos de sincronización ya descritos

- Las ventajas de utilizar hilos son que:
 - Facilitan el intercambio de datos (a través de variables globales) entre hilos cooperantes; y
 - Algunos algoritmos se trasladan más naturalmente a una implementación multihilo que a una implementación multiproceso
- Una aplicación multihilo puede aprovechar de forma transparente las posibilidades de procesamiento paralelo en hardware multiprocesador

Programación de sistemas en red

Desarrollo de Sistemas en Red



Introducción

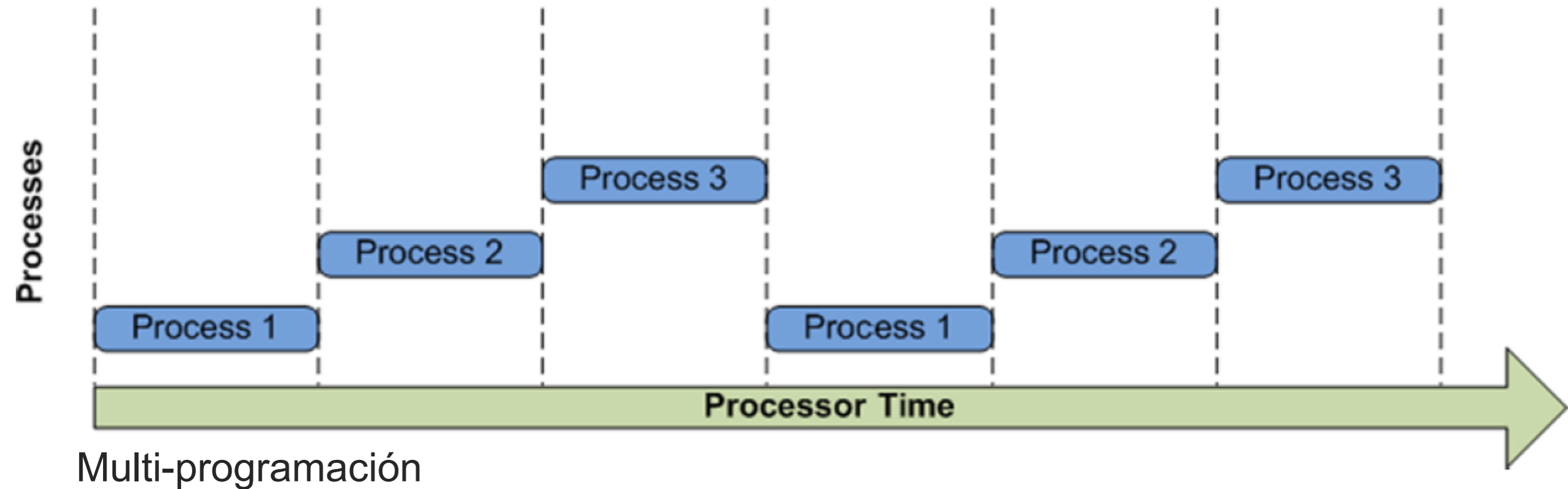
En la programación de sistemas en red se utilizan:

- Lenguajes de alto nivel
- Conjunto de principios y técnicas

Los conceptos básicos y técnicas permanecen sin importar el lenguaje de programación empleado y representan la base para construir aplicaciones

Concepto	Definición	Características principales	Ejemplo
Uni-programación	Ejecución de un solo programa de usuario a la vez.	- CPU trabaja en un programa hasta terminarlo.	MS-DOS, primeras computadoras.
		- Si espera E/S, la CPU queda inactiva.	
		- No requiere protección entre procesos.	
Multiprogramación	Varios programas están cargados en memoria y el SO selecciona cuál se ejecuta.	- El CPU cambia entre procesos cuando uno entra en espera.	UNIX, Linux, Windows.
		- Requiere mecanismos de protección de memoria.	
		- Aumenta el aprovechamiento del CPU.	
Multitarea	Extensión de la multiprogramación que da la sensación de ejecución simultánea de varios programas.	- El CPU alterna rápidamente entre tareas (time sharing).	Windows 10, macOS, Linux.
		- Los usuarios perciben que varios programas se ejecutan al mismo tiempo.	
		- Necesita planificación y control del tiempo.	

Processor Time Allocation



Paradigma multi-tarea

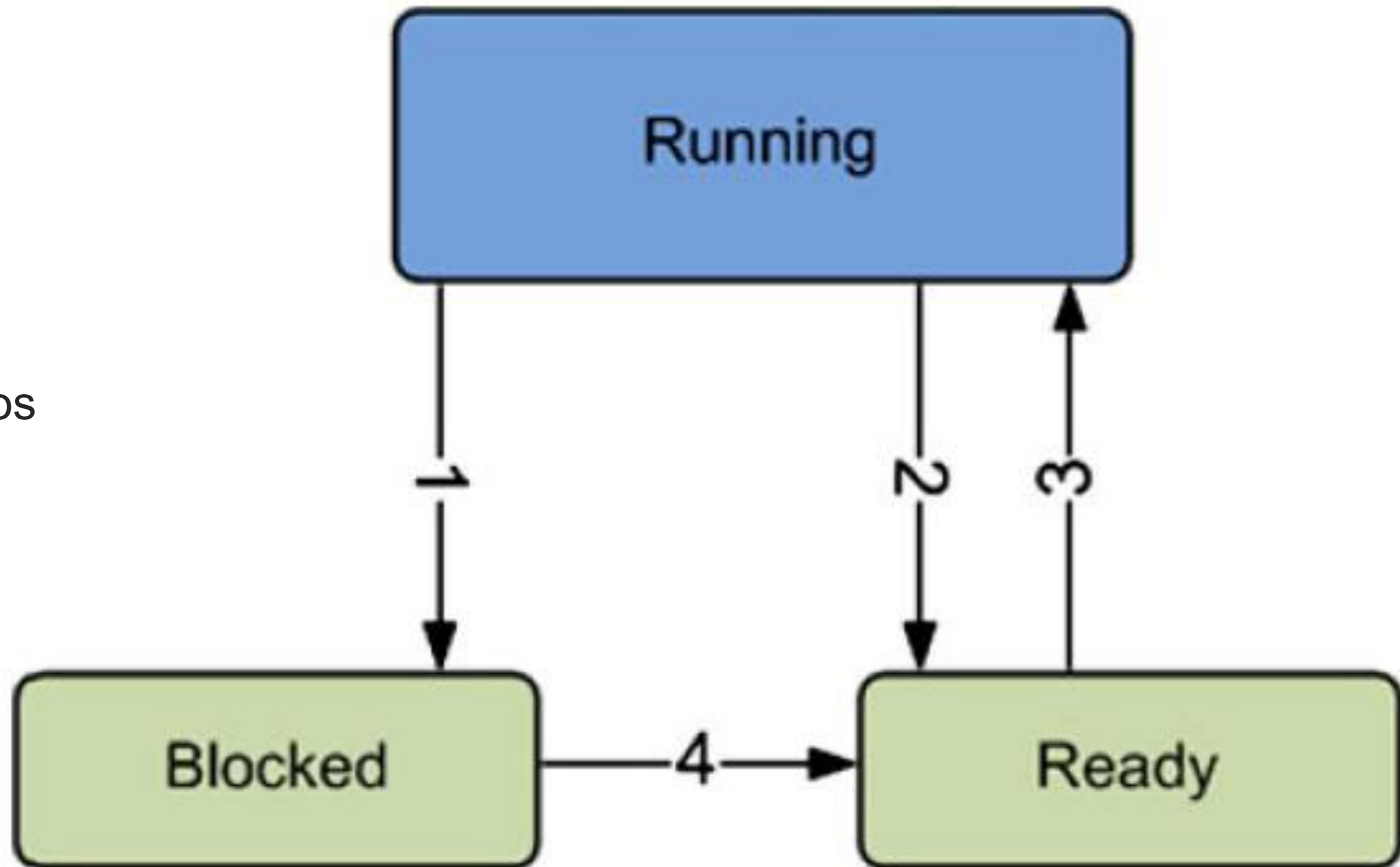
Contexto de un proceso

- Estado del proceso
- Imagen del código máquina ejecutable correspondiente al programa
- Memoria asignada
- Descriptores de los recursos utilizados y atributos de seguridad (propietario y permisos)
- Estado del procesador como el contenido de los registros y el direccionamiento de la memoria física

Procesos y estados

- El **estado** del proceso representa el estado con respecto al uso de tiempo de CPU y puede ser uno de los siguientes tres:
 - **En ejecución:** El proceso utiliza la CPU
 - **Bloqueado:** El proceso no se puede ejecutar hasta que ocurra algún evento externo. La CPU podría estar libre durante este periodo si ninguno de los procesos existentes está en posición de ejecutarse
 - **Listo (Ejecutable):** El proceso está listo para ejecutarse (no tiene que esperar a que ocurra ningún evento), pero el S.O. lo detiene temporalmente para permitir que otros procesos se ejecuten

Procesos



Transición de estados
de un proceso

Procesos y estados (2)

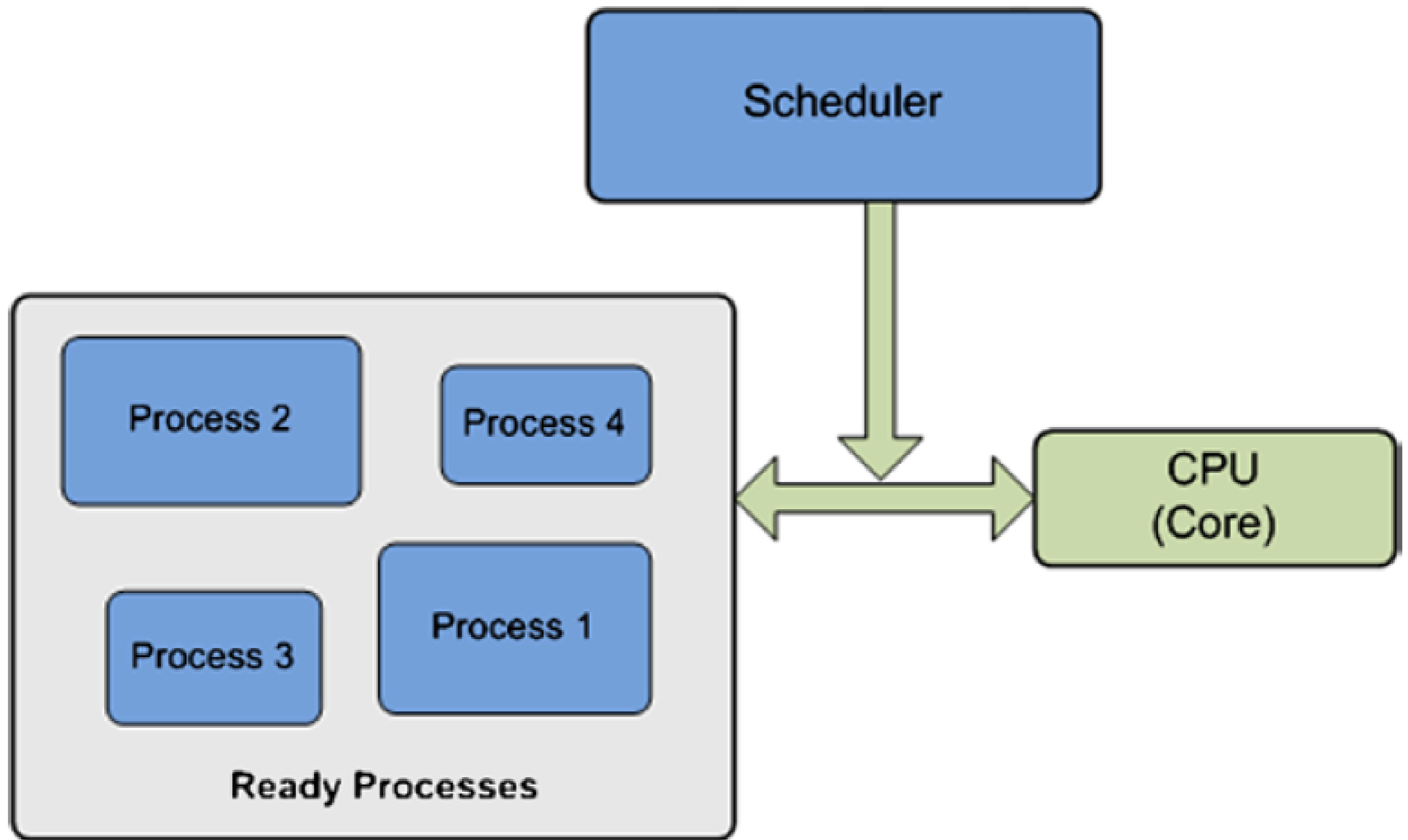
- Cuando múltiples procesos están en el estado de LISTO, el S.O. debe determinar cuál de ellos ejecutar primero:
 - De manera voluntaria (*non-preemptive scheduling*)
 - De manera forzada (*preemptive scheduling*)

De manera voluntaria (non-preemptive scheduling)

- Implica que los procesos ceden el tiempo del procesador para permitir la ejecución de otros procesos
- Generalmente ocurre cuando el proceso en ejecución tiene que cambiar al estado BLOQUEADO, mientras espera un evento externo.

De manera forzada (preemptive scheduling)

- El proceso en ejecución cambia al estado LISTO por la fuerza para permitir que se ejecuten otros procesos
- Se puede realizar de acuerdo con una política

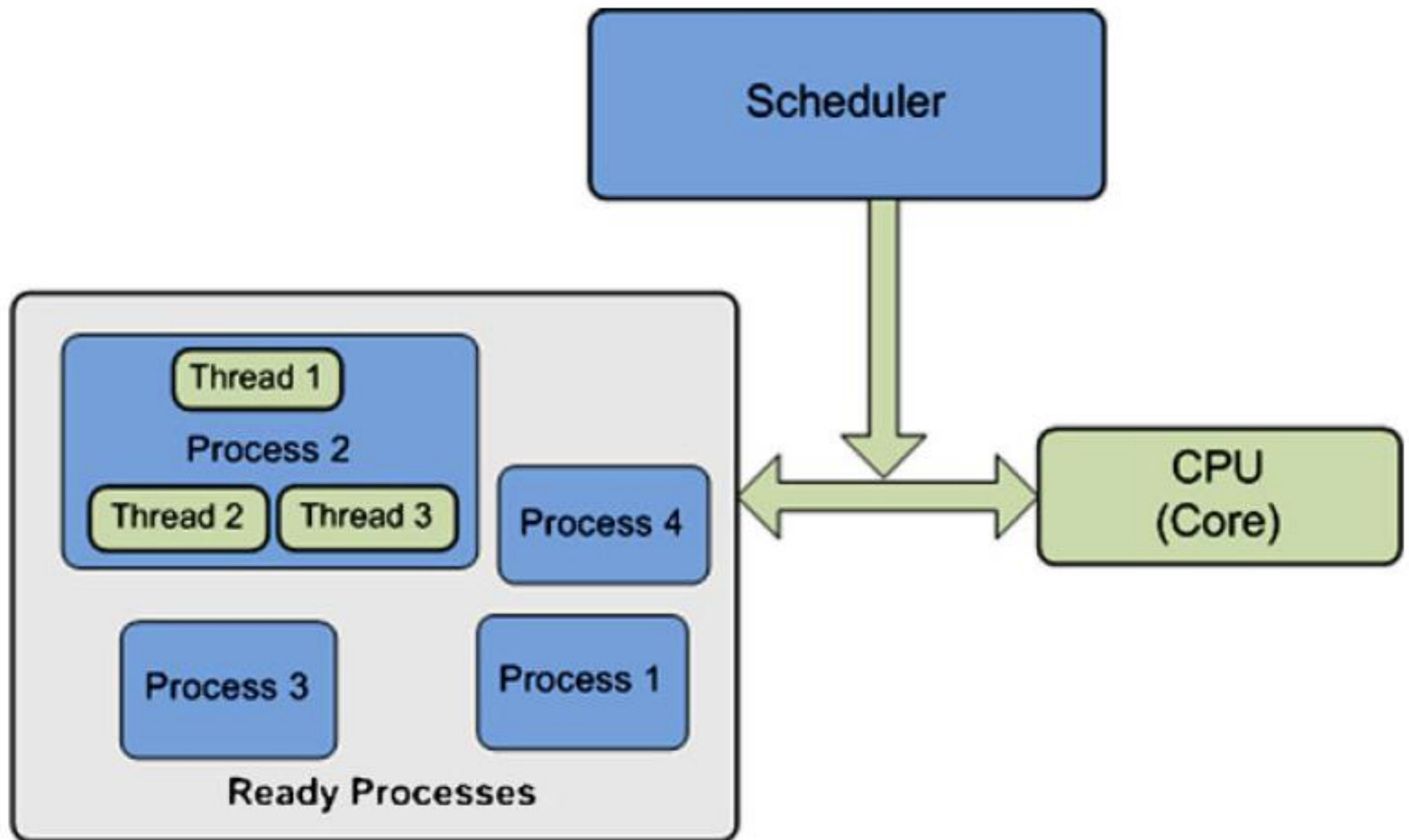


Procesos

Planificación

Hilos

- Un **hilo** es una secuencia de un programa que realiza ciertas tareas y se ejecuta dentro de un proceso
- Los hilos se ven como **procesos ligeros** (subprocesos)
 - Tienen su propia pila, pero
 - Comparten memoria, datos, y descriptores de recursos con otros hilos dentro del mismo proceso
 - Se les pueden asignar diferentes prioridades según su rol dentro del proceso de la aplicación

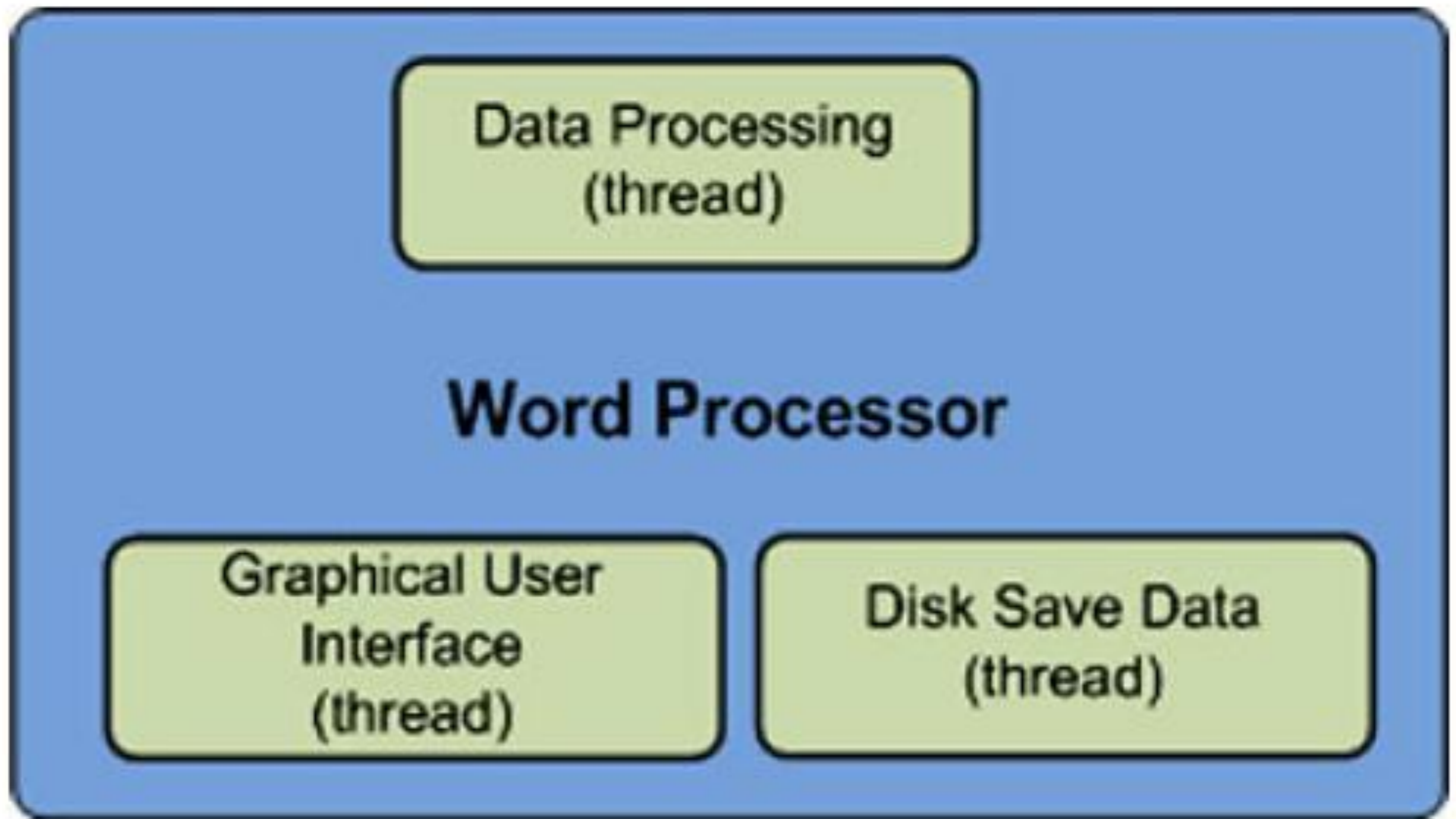


Procesos

Hilos y procesos

Multithreading

- Proporciona otro nivel de paralelismo para la ejecución de tareas, con menos sobrecarga
- Cuando existen varios hilos, se pueden realizar diferentes tareas en paralelo utilizando datos y recursos comunes
- El cambio de contexto entre hilos es menos complejo y más rápido que entre procesos. La programación multi-hilos es eficiente al emular el paralelismo
- La eficacia de la conmutación de hilos se determina por el hecho de que los hilos poseen menos recursos que deben guardarse antes de cambiar de contexto que los procesos



Hilos

Multithreading

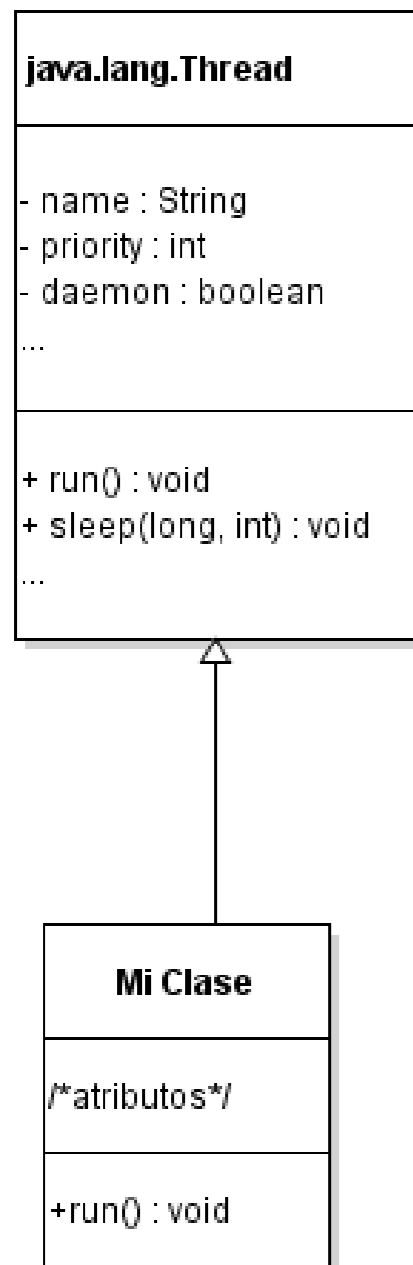
- A pesar de los beneficios, el número de hilos debe mantenerse al mínimo
- No sobrecargar al sistema con cambios de contexto innecesarios
- Los hilos deben emplearse cuando se obtienen beneficios claros del uso del procesamiento paralelo
- Involucrar *multithreading* en una serie de tareas altamente secuenciales solo agrega complejidad a una solución simple

- Las prioridades se deben gestionar de acuerdo con el propósito y los requisitos de la aplicación
- Un hilo que realiza tareas críticas en el tiempo debe tener mayor prioridad
- Ej. A la GUI se le debe asignar una prioridad más alta para mantener la interacción con el usuario dentro de los requisitos en tiempo

Multithreading en Java

- Java proporciona soporte para la programación Multithreading
- Ofrece dos formas para usar hilos:
 - Crear una clase que extienda la **clase Thread**
 - Crear una clase que implemente la **interfaz Runnable**
- Igualmente eficientes pero se prefiere la segunda ya que:
 - Proporciona una separación más clara entre el comportamiento del hilo y el hilo mismo
 - Permite una mejor reutilización

Ejemplo: Heredando de Thread



```
public class H1 extends Thread{
```

```
    @Override
```

```
    public void run() {
```

```
        System.out.println("subproceso esta corriendo...");
```

```
    }
```

```
    public static void main(String[] args) {
```

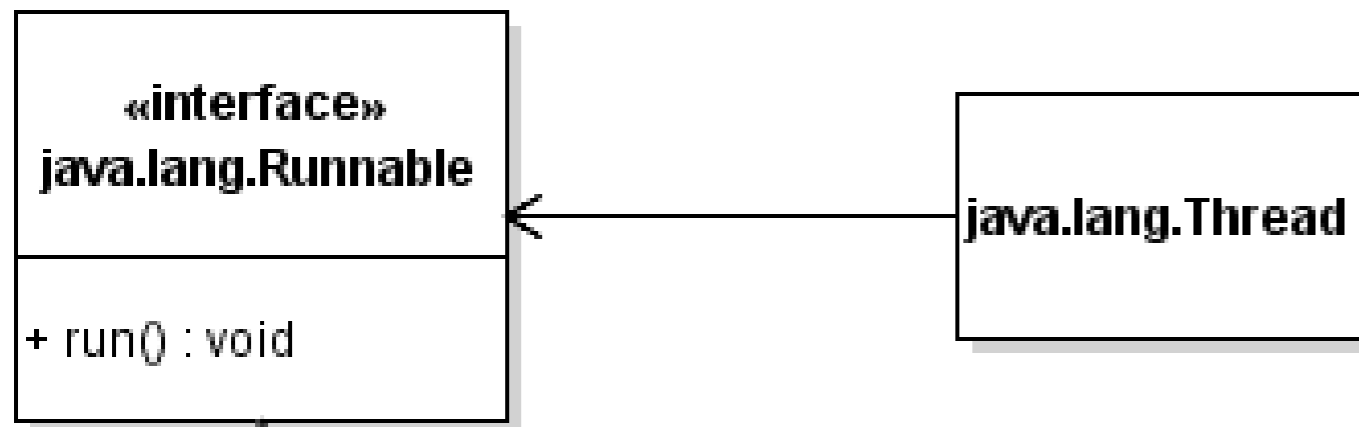
```
        H1 mih1 = new H1();
```

```
        mih1.start();
```

```
    }
```

```
}
```

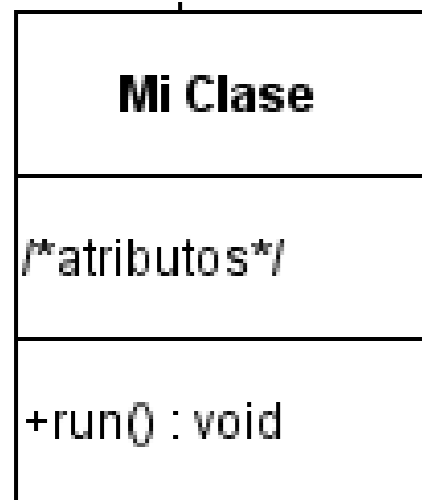
Ejemplo Implementando Interface Runnable



```
public class H2 implements Runnable{

    @Override
    public void run() {
        System.out.println("Subproceso esta corriendo...");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new H2());
        t.start();
    }
}
```



Clase Thread

- Hay varios métodos importantes que se usan con hilos:
 - `void start()` - Hace que el hilo inicie su ejecución (JVM llama al método `run()` del subproceso)
 - `void run()` - Ejecuta la tarea del hilo. Si el hilo se construyó a partir de otro objeto ejecutable, llama automáticamente al método `run()` de ese objeto
 - `void setName(String name)` y `String getName()` - Cambia y recupera el nombre del hilo cuando se le llama
 - `int getPriority()` y `void setPriority(int)` - Recupera y establece la prioridad del hilo. Los valores posibles están entre 1 y 10

Clase Thread (2)

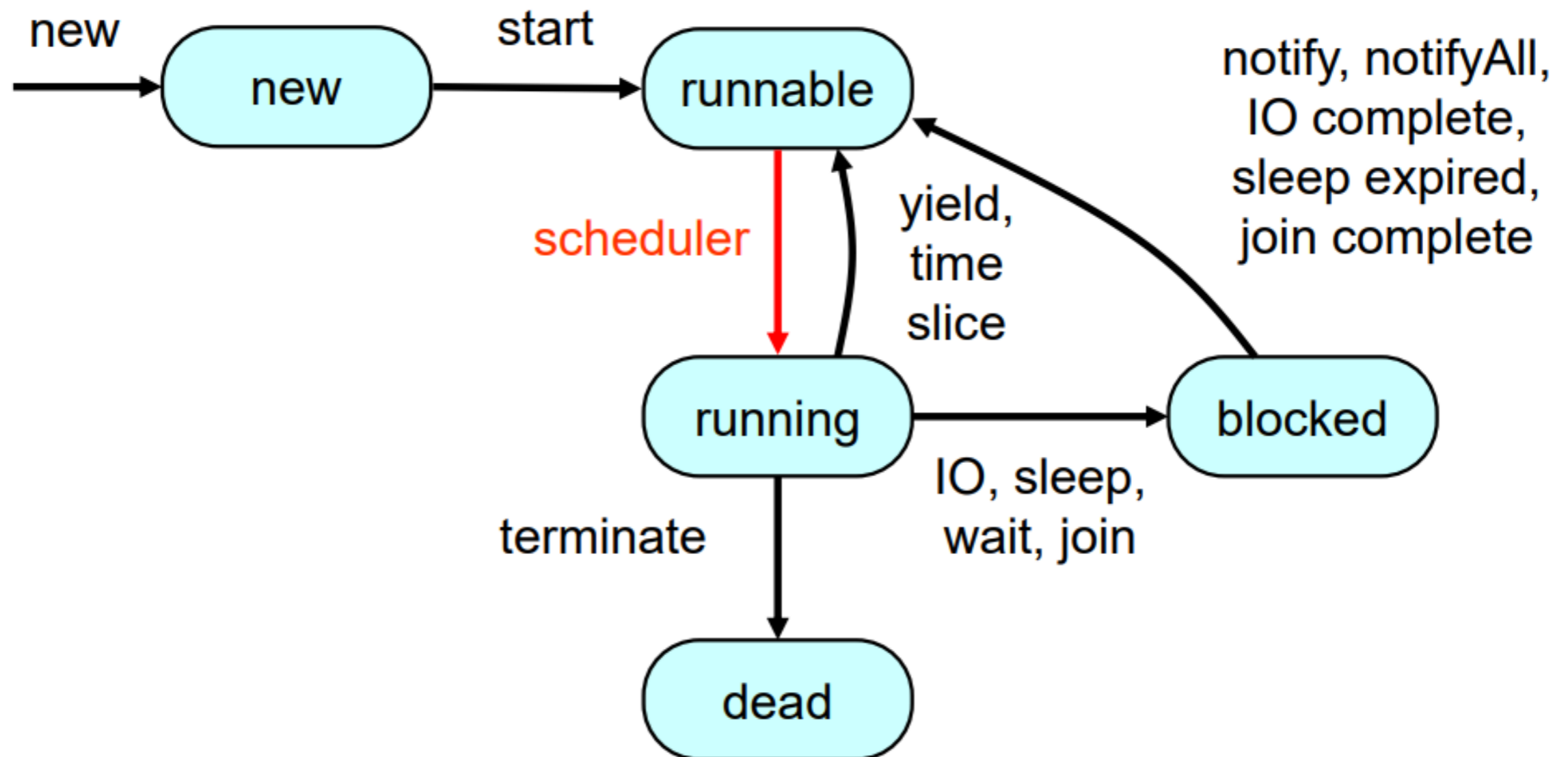
- `static void sleep(long)` y `static void sleep(long,long)` - El hilo deja de ejecutarse durante la cantidad especificada de milisegundos y milisegundos + nanosegundos, respectivamente
- `static void yield()` - Pausa el hilo temporalmente y permite que se ejecuten otros
- `void join(long millisec)` - Invocado usualmente por el hilo padre causando que éste se bloquee hasta que el hilo hijo termina o pasa el número especificado de milisegundos
- `boolean isAlive ()` - Devuelve true si el hilo está activo.

Interfaz Runnable

- El único método de la interfaz Runnable es void run() y requiere ser implementado

Ciclo de vida de un hilo

- Diagrama de estado



Comunicación inter-hilos

Comunicación inter-hilos (entre hilos)

- Se refiere a la comunicación que puede realizarse de hilo a hilo en el mismo proceso (sin considerar otros procesos)
- Los mecanismos más importantes son:
 - Memoria compartida
 - Buzones (o colas de mensajes)
 - Funciones especiales del lenguaje de programación

Memoria compartida

- Todos los hilos del mismo proceso comparten el mismo espacio en memoria, por lo que no es necesario hacer nada especial
- Método propenso a sufrir problemas de sincronización

Buzones

- Similar a la cola de mensajes de los procesos
- Cada hilo tiene asociada un buzón donde otros hilos pueden enviarle mensajes
- Los buzones normalmente están sincronizados
- Los lenguajes de programación proveen un API para hacer uso de los buzones

Funciones especiales del lenguaje de programación

- Los lenguajes de programación tienen diversas funciones especiales para realizar comunicación entre hilos
- Dichas funciones se implementan sobre los 2 mecanismos antes mencionados
- Algunas funciones comunes son:
 - Detener
 - Despertar
 - Dormir
 - Iniciar
 - Matar

Exclusión mutua y sincronización

- El tener estructuras compartidas entre hilos y/o procesos plantea un problema
- Si dos procesos están compartiendo un recurso, ambos pueden modificarlo
- Si no se comunican de forma adecuada, puede ser que un proceso sea interrumpido, justo cuando modificó el recurso y no le dio tiempo de avisarle al otro proceso
- Sin saberlo, el otro proceso modificará el recurso dando como resultado datos incorrectos

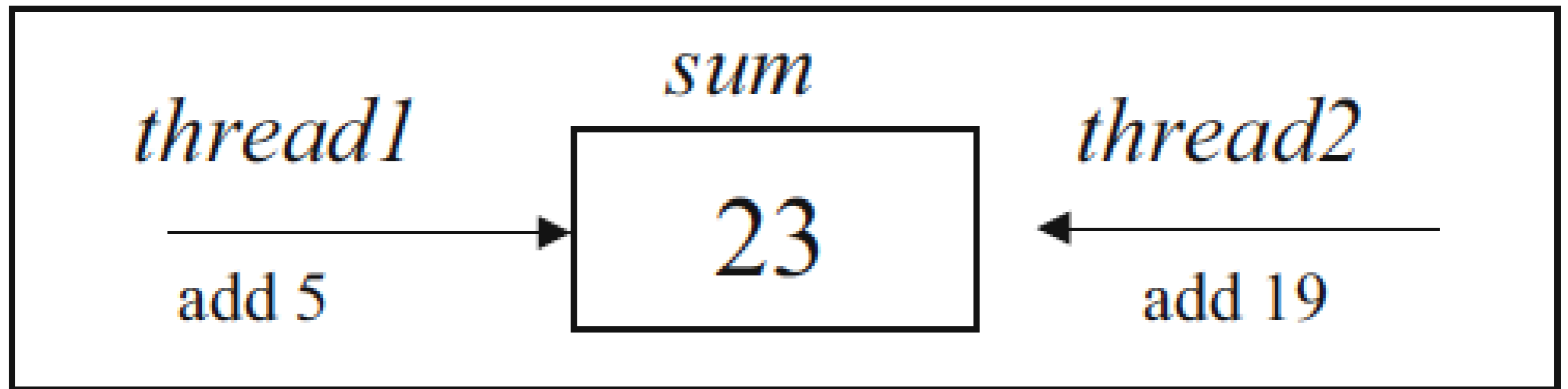
Condiciones de carrera

- Se dan cuando dos procesos están compartiendo un recurso y el resultado final depende de cual se ejecuto primero
- El sistema operativo debe proporcionar mecanismos adecuados que resuelvan los problemas que origina la competencia por los recursos compartidos

Problemas comunes de la comunicación entre procesos y entre hilos

- Garantizarle a un proceso o hilo que utiliza un recurso compartido, que otro que lo uso no lo haya modificado durante su ejecución
- **Evitar las condiciones de carrera**, impidiendo que dos o más procesos utilicen al mismo tiempo el recurso que están compartiendo

- Hacer que uno excluya al otro en el uso del recurso:
 - Esto recibe el nombre de **exclusión mutua**
 - Asegurar que cuando un proceso utiliza un recurso que es compartido, los otros no puedan usarlo



Dos hilos intentando actualizar una variable

47 (=23 + 5 + 19)

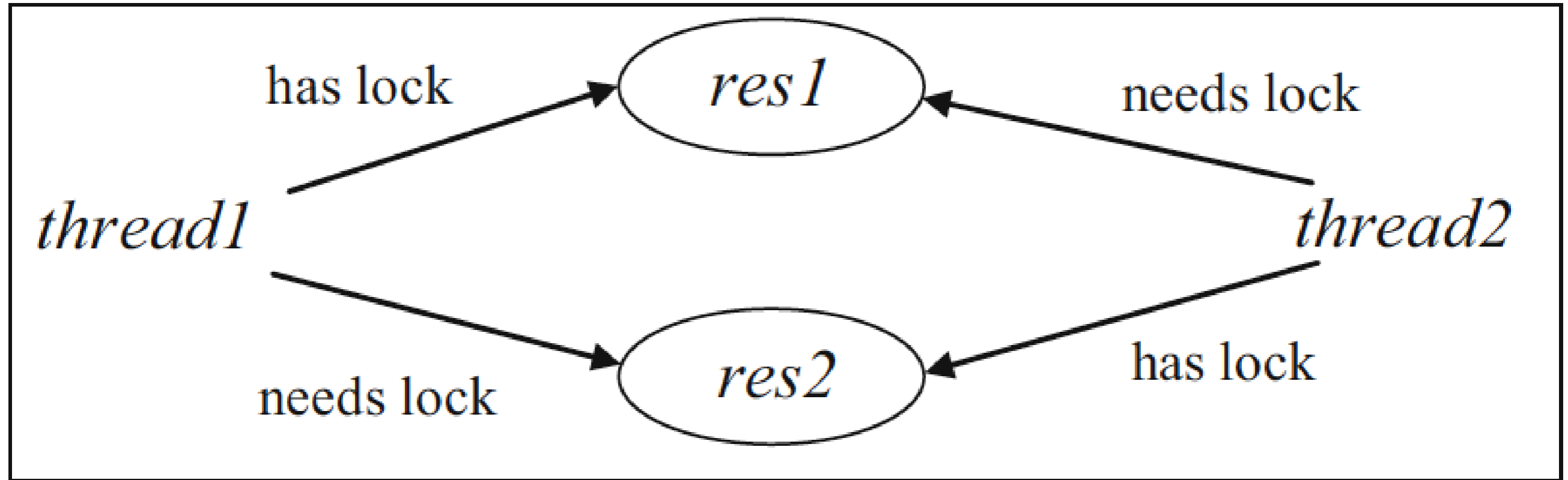
28 (=23 + 5)

42 (=23 + 19)

Problemas comunes de la comunicación entre procesos (2)

- **Interbloqueo (*deadlock*):**
 - Si dos procesos requieren el uso de dos o más recursos compartidos
 - El sistema operativo le asigna a cada proceso un recurso
 - Si uno de los procesos necesita a su vez del recurso que tiene el otro proceso para continuar (y viceversa) se produce un bloqueo (*deadlock* o abrazo de la muerte)

- Ninguno de los dos procesos puede continuar
- No se produce ningún error, simplemente los procesos (o hilos) se quedan bloqueados
- Los recursos nunca se liberan
- Normalmente son un error del programador



Problemas comunes de la comunicación entre procesos

Deadlock

Problemas comunes de la comunicación entre procesos y entre hilos (3)

- **Inanición:**

- Un proceso permanece bloqueado, esperando un recurso que necesita, pero por algún motivo externo al proceso solicitante, nunca logra que el recurso le sea asignado.
- En general porque dos o más procesos se pasan mutuamente el control de dicho recurso por lo que siempre estará ocupado.

Problemas comunes de la comunicación entre procesos y entre hilos (4)

- **Garantizar la coherencia o integridad de los datos:**
 - Si un proceso modifica el contenido de un recurso, los demás procesos deben ser enterados de esto de forma oportuna, para evitar resultados erróneos
 - Solo un proceso a la vez deberá poder escribir contenido en el recurso compartido

Problemas comunes de la comunicación entre procesos y entre hilos (5)

- **Evitar que más de un proceso este en su región crítica al mismo tiempo:**
 - La región crítica es la parte del proceso donde se tiene acceso al recurso compartido
 - Si se logra sincronizar el acceso al región crítica, se evitara las condiciones de carrera

Solución a las condiciones de carrera

- Una solución efectiva debe cumplir con las siguientes **condiciones**:
 - Dos procesos no pueden estar en sus regiones críticas al mismo tiempo
 - No pueden hacerse suposiciones acerca de la velocidad de la CPU ni su número
 - Ningún proceso que se este ejecutando fuera de su región crítica puede bloquear a otros procesos
 - Ningún proceso deberá esperar infinitamente por el recurso compartido

Sincronización (Java)

- La sincronización se refiere a no permitir que más de un hilo se ejecute en ciertas condiciones
 - En java se utiliza la palabra reservada `synchronized`
 - Hay dos niveles:
 - **Sincronizar método:** se sincroniza todo el código del método (como si todo el código fuera región crítica), el objeto de referencia utilizado para sincronizar (monitor, lock) es el propio objeto donde se encuentra el método
 - **Sincronizar bloque:** se bloquea utilizando un objeto de referencia (monitor o lock), solo se sincroniza el código en el bloque

Sincronización de hilos

- El bloqueo se logra colocando la palabra reservada `synchronized` como modificador en la definición del método o bloque de código que realiza la actualización
- Todos los demás hilos que intenten invocar este método deben esperar
- Para mejorar la eficacia de los hilos y ayudar a evitar un interbloqueo, se utilizan los siguientes métodos:
 - `wait()`
 - `notify()`
 - `notifyAll()`

Sincronización de hilos (2)

- Si un hilo que ejecuta un método sincronizado determina que no puede continuar, puede ponerse en estado de espera llamando al método `wait()`
- Esto libera el bloqueo en el objeto compartido y permite que otros hilos obtengan el bloqueo
- Una llamada a `wait()` puede dar lugar a una `InterruptedException`

Sincronización de hilos (3)

- Cuando un método sincronizado se completa, se puede hacer una llamada a `notify()`, lo que “despertará” a un hilo que está en espera
- Como no hay forma de especificar qué hilo se debe despertar, esto solo es apropiado si solo hay un hilo en espera

Sincronización de hilos (4)

- Si todos los hilos que esperan un bloqueo en un objeto deben ser despertados, entonces usamos `notifyAll()`
- Los métodos `wait()`, `notify()` y `notifyAll()` solo pueden invocarse cuando el hilo actual tiene un bloqueo en el objeto
 - Desde dentro de un método sincronizado o desde un método que ha sido llamado por un método sincronizado
- Si se llama a alguno de estos métodos desde otro lugar, se lanza una excepción `IllegalMonitorStateException`

Administración adecuada de hilos

- En muchos lenguajes de programación las operaciones para matar abruptamente hilos no son recomendables (pueden afectar al proceso)
- Si se desea contar con la funcionalidad, se recomienda implementarla manualmente
- La implementación es dependiente del problema
- Así mismo, para pausar un hilo se recomienda hacerlo dentro del método run, ya que si el hilo se ejecuta dentro de un ambiente gráfico es posible que no se obtenga el resultado esperado

Problema del productor consumidor

- Ejemplo del uso de estas primitivas es la solución al problema clásico del productor y consumidor.
- En el caso más simple, dos procesos comparten un área de memoria (buffer) la cual tiene un tamaño finito, por supuesto.
- Uno de ellos es el productor, debido a que “produce algo”, puede ser información, autos en una línea de producción, etc. y cada vez que los produce los deposita en el buffer.
- El otro proceso es el consumidor, ya que “consume” los productos que hay en el buffer. Es decir los saca de ahí.

Problema del productor consumidor (2)

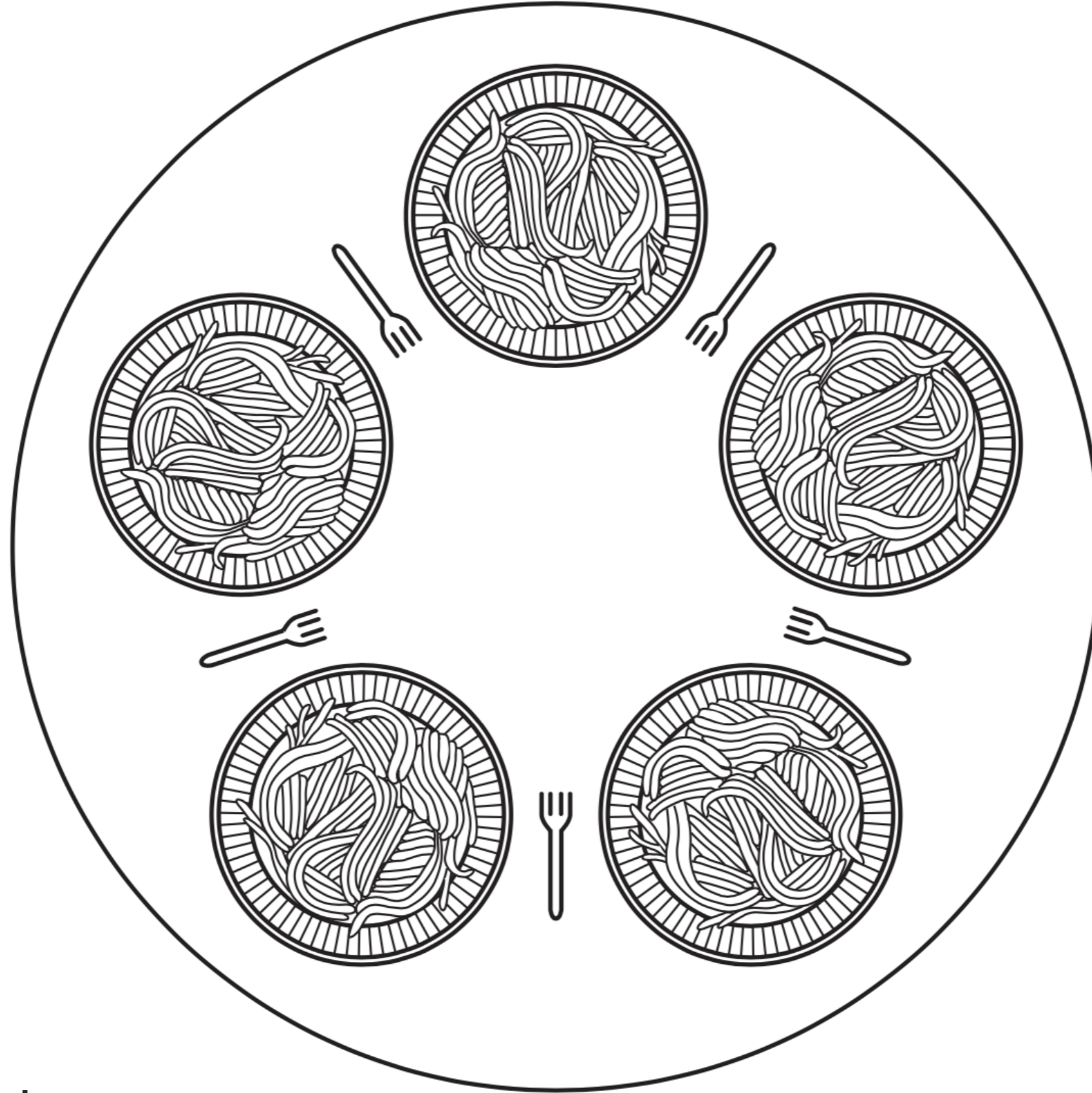
- Dado que tanto el tamaño del buffer es finito, puede llegar un momento en que el productor ya no tenga espacio para guardar sus “productos”.
- La solución es que deje de producir hasta que haya espacio, porque el consumidor ha consumido algo más. Es decir debe desactivarse.
- El consumidor también puede encontrarse con el caso de que cuando quiera “consumir” un producto, ya no haya en el buffer .
- En ese caso, deberá desactivarse hasta que el productor ponga más “productos” en el buffer.

Problema del productor consumidor (3)

- Este problema presenta condiciones de competencia como las descritas anteriormente .
- Para resolverlo es necesario saber cuántos “productos” caben en el buffer
- Con esto se podrá llevar el control del mismo para saber cuándo desactivar al productor o al consumidor.
- Se implementa una solución en Java

Filósofos comelones

- En una mesa redonda se sientan 5 filósofos, al centro de la mesa hay un tazón mágico de espagueti (el espagueti nunca se termina)
- Lo único que hacen los filósofos es comer y pensar (no pueden hacer las dos cosas al mismo tiempo)
- Para poder comer, cada filósofo necesita de 2 tenedores: un tenedor izquierdo y uno derecho. En total hay 5 tenedores en la mesa
- Como es una mesa redonda, algunos de los tenedores son compartidos. Por ejemplo: el tenedor izquierdo del filósofo 1 es el derecho del filósofo 5, mientras que el tenedor derecho del filósofo 1 es el tener izquierdo del filósofo 2



Filósofos comelones

Mesa y tenedores

Filósofos comelones (2)

- Los recursos compartidos son los tenedores (2 por filósofo)
- Si dos filósofos adyacentes desean comer al mismo tiempo puede generarse una condición de carrera (se compite por los tenedores)
- Es necesario asegurar que eventualmente todos los filósofos que desean comer lo hagan
- Es necesario evitar *dead-locks*

Posible solución

- Cada filósofo es un hilo diferente
- En cada hilo puede haber 4 métodos:
 - **Vivir** (run): es un ciclo infinito, los filósofos piensan por un tiempo aleatorio, luego se disponen a comer, tratan de tomar primero el tenedor izquierdo y luego el derecho, si tienen los dos tenedores comen durante un tiempo aleatorio (sino esperan, hacen wait), dejan de comer (liberan los tenedores) y repiten el ciclo indefinidamente.
 - **Pensar**: se imprime un mensaje que indica que el filósofo está pensando. El hilo se duerme por un tiempo aleatorio.

- **Tomar tenedor:** se indica si se quiere tomar el tenedor izquierdo o derecho. Si alguno de los tenedores esta ocupado el hilo se bloquea (se hace un wait)
- **Comer:** se imprime un mensaje indicando que el filósofo está comiendo. El hilo se duerme por un tiempo aleatorio. Al terminar de comer el hilo libera los tenedores y despierta a los hilos adyacentes que pudieron haberse bloqueado

Filósofos comelones (3)

- Para evitar problemas de inanición se puede utilizar una aproximación de alternancia estricta.
- Cada tenedor es compartido por 2 hilos, cada tenedor puede ser una variable numérica, el valor del tenedor indica cuál de los dos hilos tiene el control del tenedor en un momento dado (ninguno lo tiene al inicio del proceso).
- Un hilo solo puede utilizar el tenedor si es su turno.
- Si el turno de un hilo termina (ya comió) le da el turno al otro hilo.

- Para evitar dead-lock:
 - Si se sigue la aproximación de alternancia estricta antes mencionado, se caerá en un dead-lock desde el principio del proceso.
 - Para evitarlo basta con que uno de los filósofos no tome primero el tenedor izquierdo sino el derecho.

Ejemplo salida

Filosofo 3 toma su tenedor derecho...

Filosofo 3 esta comiendo...

Filosofo 3 deja de comer...

Filosofo 3 esta pensando...

Filosofo 1 se prepara para comer...

Filosofo 1 toma su tenedor izquierdo...

Filosofo 1 toma su tenedor derecho...

Filosofo 1 esta comiendo...

Filosofo 4 se prepara para comer...

Filosofo 4 toma su tenedor izquierdo...

Filosofo 4 toma su tenedor derecho...

Filosofo 4 esta comiendo...

Filosofo 3 se prepara para comer...

Filosofo 3 toma su tenedor izquierdo...•