Algoritmos y programación 3 - cátedra Fontela

Concurrencia

Eugenio Yolis - Noviembre 2009

Concurrencia

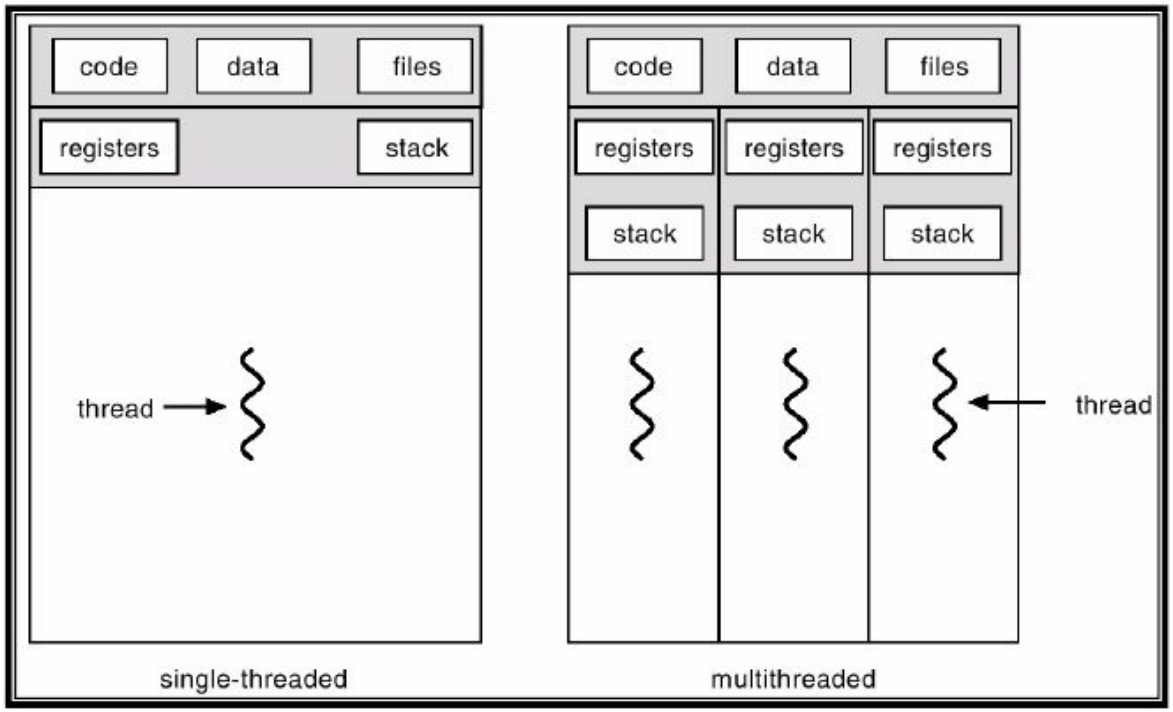
- Varias ejecuciones corriendo en paralelo
- El sistema operativo reparte el tiempo del procesador
- Caso simple: varios procesos en paralelo
  - Ejemplo: abro mi aplicación más de 1 vez
  - Los recursos compartidos son solo externos (archivos, etc)
- Caso complejo: multithreading
  - Varios hilos en el mismo proceso
  - Recursos compartidos externos
  - Pero tambien internos! (clases, objetos, atributos, etc)

Mas de 1 thread: para qué?

- Básicamente, para optimizar tiempos
- Tiempo de respuesta al usuario
  - Hago cosas “en background” mientras el usuario sigue trabajando (ejemplo, eclipse compila todo el tiempo sin que yo tenga que dejar de escribir)
- Tiempos de ejecución de mi aplicación
  - Red, discos rígidos, etc, son lentos
  - Puedo tener distintos threads esperando (ejemplo, abro varias ventanas del explorador de internet para cargar páginas en paralelo)

Threads (hilos)

- Por lo menos tenemos un hilo de ejecución, pero puede haber más
- Cada hilo tiene su stack (variables locales),
  - Pero comparten el heap (objetos en memoria)



Threads en Java

- Varias formas
  - La recomendada:
    - Una clase que implementa Runnable (tiene un metodo “run”)
    - Crear un objeto de esa clase
    - Crea un thread, y pedirle que corra a ese objeto

```
class HelloRunnable implements Runnable {
   public void run() {
       System.out.println(“Hello Runnable”);
   }
}

Thread t1 = new Thread(new HelloRunnable());
t1.start();
```

Ejemplo 1

- Vemos lo siguiente:
  - Cada thread tiene sus variables locales, pero comparten el heap
  - Thread.sleep(int) // duerme el thread
  - t.join() // espera a que termine t
  - Todos los metodos que esperan, pueden fallar
    - Porque nos pueden “interrumpir” en el medio
- No es tan facil programar con threads!
  - El resultado, es el esperado?

Arreglando el ejemplo 1

- “Debuggear” es complicado
  - Hay más de 1 thread corriendo
- Debemos hacer “join” con los plazos fijos tambien

Ejemplo 2

- Vemos lo siguiente:
  - Cuando el procesador reparte su tiempo, ejecuta solo un poco de cada thread por vez
  - No necesariamente ejecuta cada metodo entero
  - No conocemos el algoritmo que usa para cortar
- La ejecucion no es determinista!
  - No falla siempre igual…

Arreglando el Ejemplo 2

- Debemos sincronizar el acceso a las cosas!
  - “synchronized” nos asegura que 2 threads no van a ejecutar a la vez ese código
- No alcanza con agregar “synchronized” a los metodos de cuenta!
  - Porque cuando lo usamos, primero pedimos el saldo y despues modificamos
  - Debemos sincronizar todo el acceso al objeto
- A veces, agregar más “println()” puede alterar el resultado
  - Porque al hacer entrada/salida, probablemente la JVM cambie de thread

Más sobre sincronización

- No es simple
- Sincronizar de más puede hacer que perdamos la ventaja de correr con varios threads
- Sincronizar de menos produce errores dificiles de detectar (como ya vimos en los ejemplos)
- Otros problemas
  - Deadlocks (cuando 2 threads se esperan mutuamente)

Más sobre la API

- t.interrupt(); // interrumpe al thread
- t.isAlive(); // pregunta si sigue corriendo
- t. join(); // espero a que ese thread termine
- Thread.sleep(int); // espero esa cantidad de milisecs
- objeto.wait(); // me quedo esperando
- objeto.notify(); // despierto a uno de los que esperan
- objeto.notifyAll(); // despierto a todos que esperan

Preguntas?