Programación Concurrente

Varios procedimientos son ejecutados durante periodos de tiempo traslapados en lugar de ejecutarse secuencialmente.

Hardware en el computo paralelo

* **Multiprocesador**: varios procesadores físicos (CPUs) conectados por memoria o red.
* **Multinucleo**: varios procesadores físicos dentro del mismo chip.
* **Multihilo**: simula varios procesadores lógicos dentro de un único procesador físico.

***Proceso***: instancia de un programa en ejecución. (Cada proceso tiene su PCB)

***Hilo***: secuencia mas pequeña de instrucciones programadas que pueden ser manejadas independientemente por un planificador.

**Procesos** (PCB)

Un proceso consta al menos de:

1. El código del programa
2. Los datos del programa
3. Una pila de ejecución
4. El PC indicando la próxima instrucción
5. Un conjunto de registros de propósito general con los valores actuales.
6. Un conjunto de recursos del SO (memoria, archivos abiertos, etc.)

Lo importante son los procesos, no los programas.

**Procesos Concurrentes**

* Pueden ser ejecutados en un solo núcleo intercalando los pasos de ejecución de cada proceso mediante segmentos de tiempo 🡪 ***Multitarea Apropiativa***.
* Solo se ejecuta un proceso a la vez y si no se termina en el tiempo asignado, se pausa.

Estados de Transiciones

1. Nuevo
2. Ejecutándose
3. Preparado
4. Bloqueado
5. Terminado

Cambio de contexto

* Cuando un proceso esta ejecutándose, su PC, puntero a pila, están cargados en la CPU.
* Cuando se detiene su ejecución, salva los valores (contexto) en el PCB de ese proceso.
* La acción de conmutar la CPU de un proceso a otro, se denomina ***cambio de contexto***.

Memoria Compartida

* Los componentes concurrentes se comunican mediante la alteración del contenido de ubicaciones de memoria compartida.
* Requiere una forma de bloquea (Mutexes, semáforos, monitores) para la sincronización entre procesos e hilos.

Peligros de la Ejecución Concurrente

* ***Condición de carrera***: varios procesos acceden al mismo tiempo y cambian el estado de un recurso compartido.
* ***Deadlock*** (una persona con arco y otra persona con la flecha)
* ***Livelock*** (dos personas en un pasillo que se bloquean mutuamente y se mueven para dejar pasar el otro, manteniendo el bloqueo).
* ***Starvation***(filósofos chinos) se le deniega siempre el uso a un recurso.

**Conceptos importantes**

* Exclusión Mutua: asegurar que solo un proceso acceda a un recurso mientras lo este usando.
  + Monitor
  + Semáforos
  + Mutex

Paso de mensajes

* Los componentes concurrentes se comunican mediante el intercambio de mensajes.
* El intercambio de mensajes se puede efectuar *asíncronamente o síncrono*
  + Síncrono: el que envía el mensaje se bloquea hasta que recibe el mensaje.
  + Asíncrono: puede ser confiable o no confiable (**send** and **pray**).

Interacción mediante paso de mensajes

* Los mensajes recibidos son solicitudes y son puestos en una cola para ser manejados uno por uno.
* El proceso que envía no para de trabajar mientras espera una respuesta a su mensaje y sigue atendiendo a los demás en la fila.

Peligros

* No elimina las *condiciones de carrera*, pero si las disminuye.
* Tampoco elimina *Deadlocks*, cuando un proceso se queda en espera de la respuesta de un mensaje.

Procesos y Concurrencia en Erlang

Las funciones son asíncronas, por lo que no se espera a que se obtenga un resultado para continuar con el resto de las instrucciones.

En Erlang, la Concurrencia se implementa mediante la ***creación y comunicación de procesos***.

* Proceso: unidad de cómputo separada, es una función en ejecución, que se ejecuta concurrentemente con otros procesos en el sistema.
* Los procesos de Erlang no comparten memoria (datos) con otros procesos.
* Los procesos se comunican mediante el paso de mensajes (modelo de programación concurrente)

Procesos

En Erlang, los procesos pertenecen al lenguaje y no al SO.

* Spawn: crear procesos
* Send: mandar mensajes
* Receive: recibir mensajes

**Creación de Procesos**

Spawn/1 o spawn/3, crean un nuevo proceso concurrente y regresa su identificador.

Pid = **spawn**(función)

Pid = **spawn**(modulo, función, listaArgs)

* Los **pid** son usados para intercambio de mensajes.
* La llamada no espera a que la función se evalúe (regresa inmediatamente).
* El proceso termina automáticamente cuando la función termina de ejecutarse.
* El valor de retorno del proceso se pierde.

Envío de mensajes

Un mensaje se envía a otro proceso mediante el primitivo **!** (**send**), como:

Pid ¡ mensaje.

* El **send** regresa el mensaje que envías.
* El envío de mensaje es **asíncrono**
  + El que envía el mensaje continúa con lo que estaba haciendo, **no espera.**
  + El sistema no informa al que envía si el mensaje se entregó.
* El mensaje puede ser cualquier término Erlang válido.
* El valor de retorno de **!** es el mensaje que envía, es *asociativo por la derecha*.
  + Pid1 ¡ Pid2 ¡ Mensaje
  + Es equivalente a Pid1 ¡ (Pid2 ¡ Mensaje)



Si el receptor no ha terminado, todos los mensajes le son entregados en el mismo orden

en el que se le envían.

Recepción de Mensajes

El primitivo **receive** se usa para recibir mensajes

receive

Patron 1 when Guardia1 –> Acciones 1;

Patron 2 when Guardia2 -> Acciones 2;

. . .

end

* Cada proceso tiene su propio buzón de correo.
* Todos los mensajes que se le envían a un proceso se almacenan en su buzón en el orden en la que se reciben.
* **Receive** regresa el resultado de la ultima acción ejecutada.

El **Receive** termina de ejecutarse cuando un mensaje es recibido, el sistema trata de empatarlo secuencialmente.

* Si un mensaje empata, se elimina del buzón y se evalúan las acciones relacionadas.
* Si no empata con ningún patrón, permanece en el buzón para su procesamiento posterior y el proceso continua con el siguiente mensaje en su buzón.
* El proceso que evalúa un **Receive** es suspendido hasta que un mensaje sea empatado.
* Los mensajes que arriban a un proceso no pueden bloquear otros mensajes para ese proceso.
* El buzón se puede llenar con mensajes que no empatan con los patrones.
* Es responsabilidad del programador asegurarse que el buzón no se llene.

Mensajes de procesos específicos

Cuando se quiere recibir mensajes de un proceso específico, el que envía debe mandar su propio **pid** en el mensaje.

Tiempos de espera

Recordemos que **Receive** supende la ejecución de la función mientras el proceso no empate con alguno de los mensajes.

**Receive** puede incluir tiempos de espera (*timeouts*) para no bloquear al proceso para siempre si no recibe un mensaje.

receive

After T -> true

end.

Las acciones de espera se evalúan si no se selecciona un mensaje antes que termine el tiempo de espera.

Ejemplo: suspender un proceso T milisegundos

Ejemplo: detectar dobles clicks

receive

{mouse, click} ->

receive {mouse, click} -> doublé\_click

After

Double\_click\_interval() -> single\_click

end

end

Tiempos especiales de espera

1. Infinity, especifica una espera que nunca ocurrirá.
2. 0 especifica que la espera se acaba inmediatamente
   1. Antes, el sistema trata todos los mensajes actualmente en el buzón.

Recordar: Los mensajes se procesan en el orden que se reciben.

Registro de Procesos

* El PID de un proceso se requiere para mandarle un mensaje
  + Esto es muy seguro, pero inconveniente porque el procsoe le tiene que enviar su PID a todos los otros procesos que se quieran comunicar con el.
  + Erlang tiene un método para publicar los PIDs para que cualquier proceso en el sistema les pueda enviar mensajes => **Registo de procesos.**
* Primitivos:
  + Register(Alias, Pid) – registra el proceso Pid con el nombre Alias (un atomo).
  + Unregister(Alias) – remueve cualquier registro con el nombre Alias.
  + Whereis(Alias) -> Pid | undefined – determina si el nombre Alias esta registrado.
  + Registered() – regresa una lista con todos los procesos registrados en el sistema.

Ejemplo de Registro de procesos

A picture containing diagram, text

Description automatically generated

A picture containing text

Description automatically generated