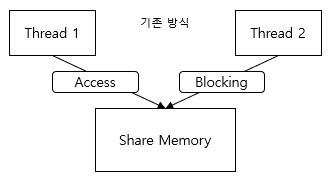
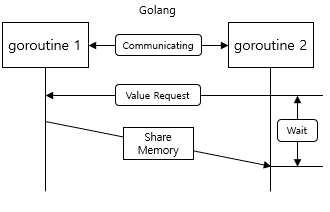
**Golang 동시성 모델**

**CSP**(Communicating Sequential Processes) 모델

**공유 메모리 접근**





* 언어 차원에서의 순차적 통신 행위를 할 수 있도록 하는 여러 기능들을 제공한다.
* ‘Buffered channel’을 지원하여 이것에 의해 전송이 완료 될 때까지 channel 간에 어떤 ‘lock’이나’sync’을 맞출 필요가 없도록 한다.
* Sync/lock(Mutex) 또한 지원한다

**Goroutine**

* ‘Thread’와 비슷한 개념으로 경량화된 Thread라고도 한다.
* 동일한 주소 공간에서 다른 Goroutine과 동시에 실행할 수 있는 기능
* 모든 동기화 및 메모리 관리는 기본적으로 Go 언어에 의해 수행된다.
* 그들이 진짜 Thread가 아닌 이유는 기본적으로 병행적으로 행동하지 않으며 ‘Multiplexing’ & ‘Sync’로 인해 동시적으로 동작이 가능하기 때문이다.

**Channel**

* 동시성을 실현하는 또 다른 핵심 개념
* Goroutine간의 ‘통신’을 담당한다.
* 버퍼링 된 채널을 이용하면 goroutine이 대기하기 전에 더 많은 값을 대기열에 넣을 수 있다.

**Golang 스케줄링**

**필요성**

OS와 Golang 스케줄러가 어떻게 작동하는지에 대한 이해를 가지고 멀티쓰레드 소프트웨어를 올바르게 설계하기 위해 Golang 스케줄러의 설계 및 동작에 대한 이해가 필요하다.

**OS 스케줄러**

* 스케줄러는 실행되는 하드웨어의 상태와 설정(다중 프로세서 및 코어, CPU 캐시 및 NUMA)을 고려해야 한다.
* 프로그램은 순차적으로 실행해야 하는 일련의 기계 명령어일 뿐, 할당된 명령어 세트를 순차적으로 실행하는 것은 운영체제의 ‘Thread’이다.
* 모든 프로그램은 프로세스를 만들고 각 프로세스는 초기 Thread가 제공된다. Thread는 서로 다른 Thread와 독립적으로 실행되며 스케줄 결정은 프로세스 레벨이 아닌 Thread 레벨에서 수행된다.
* Thread는 더 많은 Thread를 생성할 수 있으며 동시(하나의 코어에서 실행) 또는 병렬(여러 코어에서 실행)로 실행될 수 있다.
* 운영체제 스케줄러는 실행할 수 있는 Thread가 존재하는 한 코어가 쉬지 못하게 해야 한다.

**Thread Status**

* 대기, 실행가능, 실행 세가지 상태가 있다.
* **대기** : 하드웨어(디스크, 네트워크), 운영체제(system call), 동기화 호출(atomic, mutex) 등이 원인, 성능저하의 근본 원인
* **실행가능** : 기계 명령어를 실행할 수 있다. 여러 쓰레드가 코어 사용시간을 요구할 경우, 쓰레드는 오랜 시간을 기다려야 한다. 또한 개별 쓰레드의 사용시간은 단축되어 쓰레드가 사용시간을 두고 경쟁한다. 이는 성능저하를 유발한다.
* **실행 :** 쓰레드가 코어에 할당되어 기계 명령어를 실행중인 상태

**작업의 유형**

* **CPU-Bound** : 쓰레드를 대기 상태로 절대로 변경시키지 않는 작업으로 작업은 지속적으로 연산한다.
* **IO-Bound** : 쓰레드를 대기 상태로 변경시키는 작업으로 네트워크를 통해 리소스에 접근하거나 시스템 호출을 운영체제로 보내는 작업이다. 데이터베이스에 접근해야 하는 쓰레드는 IO-Bound가 되며 쓰레드를 대기하게 하는 동기 이벤트(atomic, mutex)도 포함된다.

**Context Switch**

* 코어에서 쓰레드를 교환하는 행위
* 실행 쓰레드를 코어에서 끌어와서 실행가능 쓰레드로 대체할 때 발생한다.
* 실행 큐에서 선택된 쓰레드가 실행 상태로 이동
* 끌어온 쓰레드는 실행가능 상태, 또는 대기 상태로 이동할 수 있다.
* 컨텍스트 스위치는 시간이 걸리기 때문에 비용이 많이 든다.
* IO-Bound 중점의 프로그램에서는 장점
* CPU-Bound 중점의 프로그램에서는 단점