서버 아키텍처 이해를 위한

Process, Thread, IO Model #1

Why?

- 복잡도와 요구사항은 늘어나고 일정은 짧아짐
- 처리해야할 클라이언트도 증가
- 일정 규모 이상의 서비스를 제공하려면 아키텍처에 대한 이해 필요
- 아키텍처 개선은 프로세스/쓰레드 같은 low level 지식 필요

세미나 목표

- 왜 CGI 는 느리고 mod_php, mod_python 은 이보다 빠를까?
- 왜 특정 DBMS 는 프로세스 구조를 채택했을까
- 왜 apache 는 server pool 구조일까
- 왜 DB Connection pool 이 web 서비스에 꼭 필요한가?

발표자

- 정광섭 (lesstif@gmail.com)
- http://lesstif.com, https:/github.com/lesstif
- "쉽게 배우는 라라벨 5 프로그래밍" 저자
  - http://wikibook.co.kr/laravel-5-programming/
- "견고한 웹 서비스 만들기" 문서 프로젝트 진행중
  - https://github.com/lesstif/web-service-hardening

Program - 정의

- 지정한 작업을 수행하도록 작성후 컴파일 또는 인터프리팅
- OS 에 의해 실행

Program - 정의

```c
#include <stdio.h>

static char* name="KwangSeob Jeong";
static char* addr;

const char* msg = "Hello, World!";
int main()
{
    int i = 0;

    char* comp = malloc(sizeof(char) * 100);

    printf("%s %d\n", msg, i);

    return 0;
}
```

- gcc –o a.out a.c
- 생성된 실행 파일 a.out 이 프로그램

Process 정의

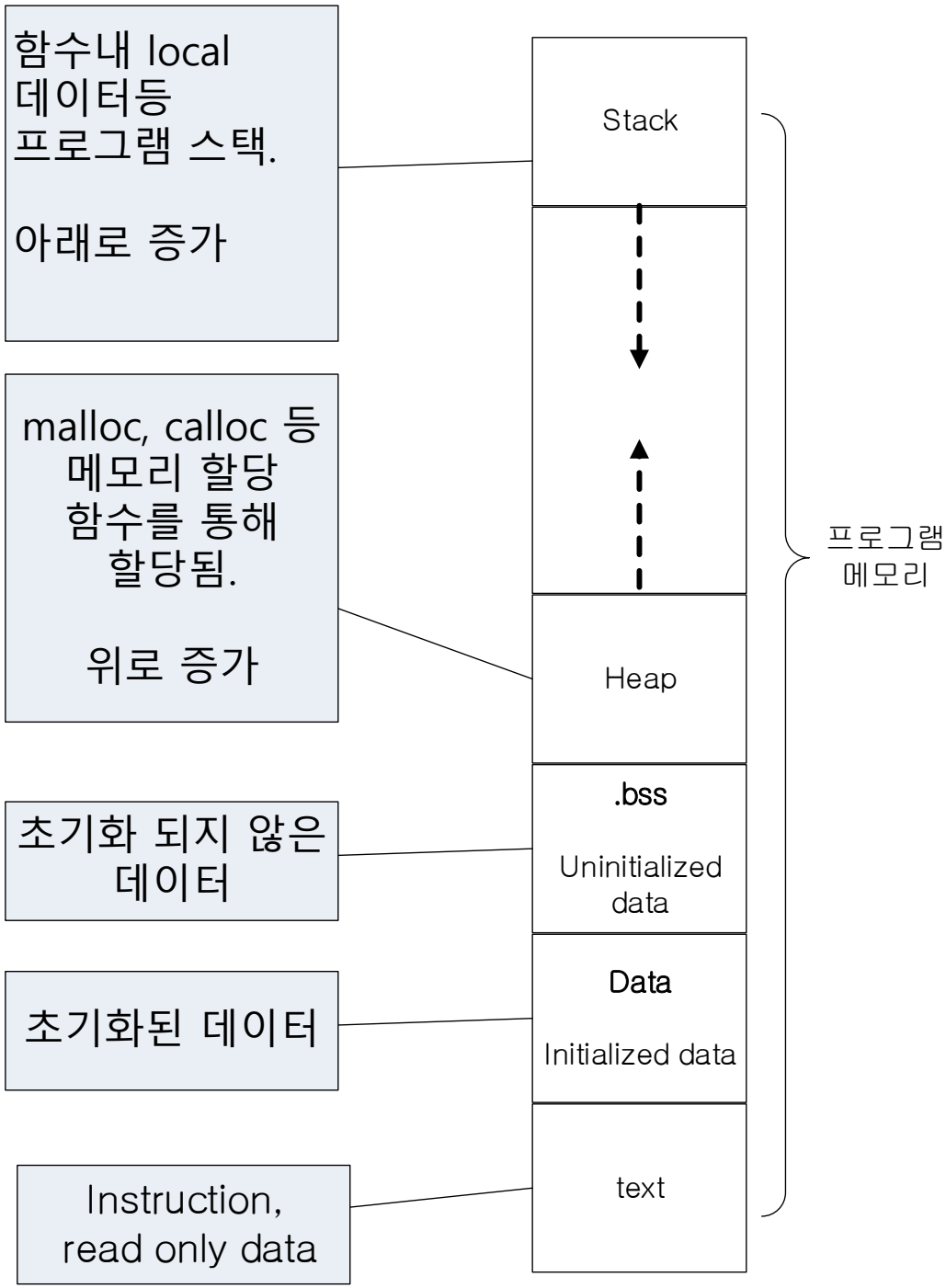
- 프로그램을 실행하여 구동된 상태
- 모든 프로세스는 부모 프로세스를 가짐
- OS 에 의해 실행되며 프로세스마다 아래와 같은 리소스를 각각 보유함

Process - 리소스

- Process ID, process group ID, user ID, and group ID
- Environment, Working directory
- Stack, Heap
- File descriptors
- Signal handler
- Shared libraries
- Inter-process communication tools (message queues, pipes, semaphores, shared memory).

메모리 구조

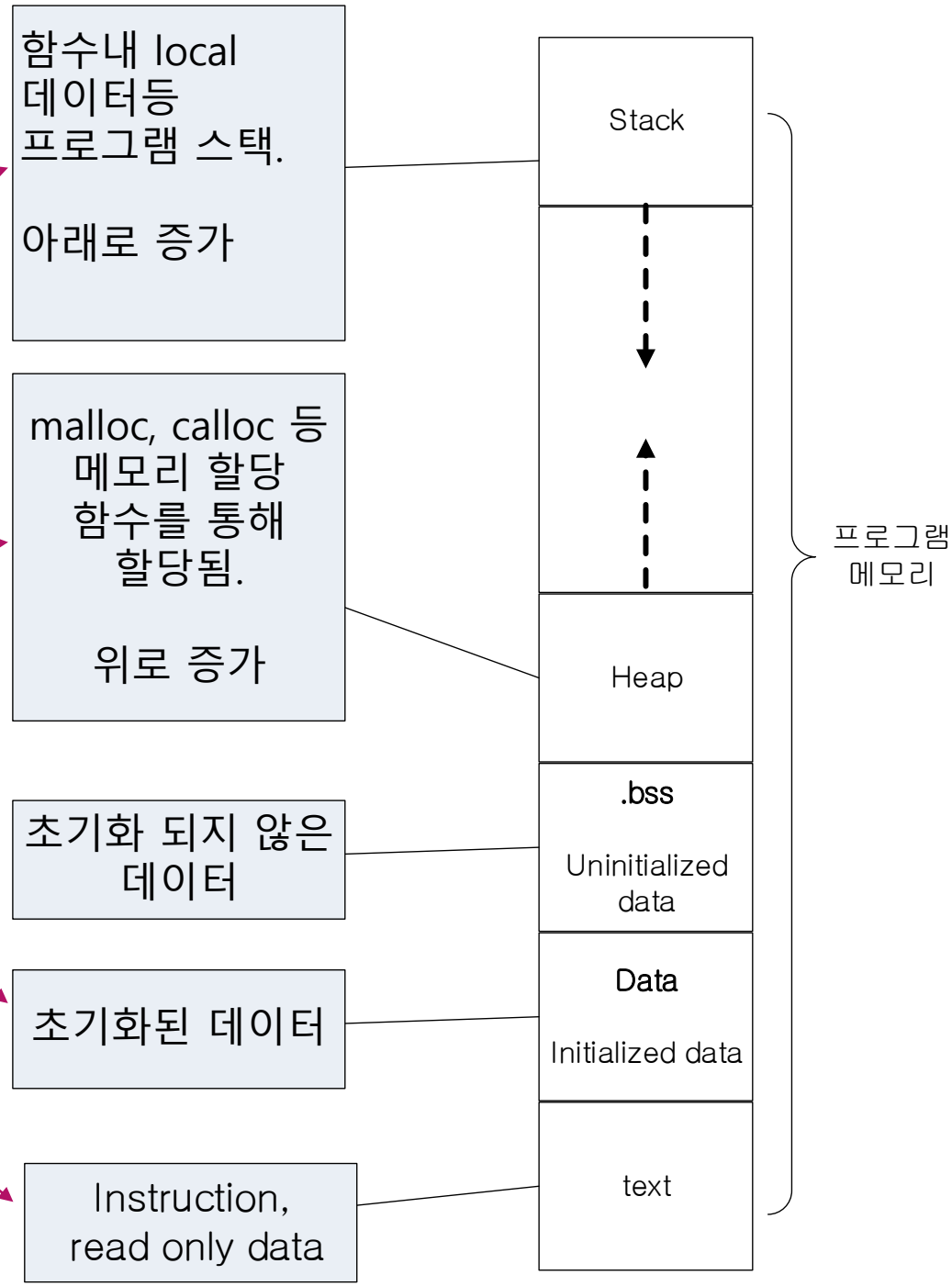
https://en.wikipedia.org/wiki/Data_segment



메모리 구조

https://en.wikipedia.org/wiki/Data_segment

```c
#include <stdio.h>

static char* name="KwangSeob Jeong";
static char* addr;

const char* msg = "Hello, World!";
int main()
{
    int i = 0;

    char* comp = malloc(sizeof(char) * 100);

    printf("%s %d\n", msg, i);

    return 0;
}
```



Process - 생성 - #1

```c
#include <stdio.h>

int main()
{
    pid_t pid = fork();

    if (pid == 0) // child process
    {
        printf("i'm child\n");
    }
    else if (pid > 0) // parent
    {
        printf("i'm parent\n");
    }
    else
    {
        printf("fork failed\n");
    }
}
```

- *fork* system call로 프로세스 생성
- 두 번 리턴하는 신기한 시스템 콜

Process - 생성 - #2

- *fork* 시 두 프로세스간 메모리는 격리됨
- *fork* 된 자식은 부모 프로세스의 메모리 및 데이타를 그대로 복사
- OS 는 실제로는 COW(Copy on write) 기법을 사용하므로 부모의 메모리를 바로 복사하진 않음

Process - 생성 - #3

- 그래도 *fork* 는 많은 부하가 발생하는 시스템 콜
- 프로세스 덩치가 크고 사용하는 메모리가 많을수록 (Ex: 오라클 DBMS) *fork* 호출 비용이 비쌈

Process - 단점

- 생성 비용이 많이 발생
- 프로세스간 통신이 어려움
  - IPC(Inter Process Communication), Socket, Pipe 등을 사용하여 메시지 전달
  - ipcs 명령으로 현재 시스템의 IPC 현황 알 수 있음.
- 한 프로세스 내에서 동시에 여러 로직을 수행할수 없음(지금은 멀티 코어 시대)
- Process 간 Context switching 비용이 비쌈

Process 관리 FAQ - #1

- 모든 프로세스는 부모가 있다고 했는데 만약 부모 프로세스가 죽으면 ?

- 유닉스는 init daemon(PID: 1) 이라는 훌륭한 고아 원장님이 계심.
- 이 분이 고아 프로세스(orphan process)를 입양
- Kernel 은 부팅시 init 프로세스를 제일 먼저 생성

Process 관리 FAQ - #2

- 좀비 프로세스는 무엇인가요?

- 커널은 자식 프로세스가 종료하면 부모에게 SIGCHLD 라는 시그널을 송신
- 부모는 *wait* 시스템 콜을 호출하여 자식의 종료 상태를 얻어가야 함
- *wait* 호출을 하지 않으면 process table 에 정보가 남아 있고 이 상태가 좀비 프로세스
- zombie, 또는 defunct process 라고도 함

A1. cgi보다 mod_php가 빠른 이유

- cgi 방식으로 php 스크립트를 구동할 경우
  - httpd -> fork -> exec() 로 php 실행
  - php 스크립트 해석때마다 fork 로 httpd 복제
- mod_php 방식
  - php 엔진이 httpd 에 모듈로 포함. fork 불필요

Thread 정의

- 한 프로세스 내에서 동시에 실행될 수 있는 단위
- 기존 프로세스는 싱글 쓰레드
- 여러 개의 thread 를 생성하여 처리하는 것을 Multi Thread라 통칭

Thread 장점

- 프로세스의 메모리 공유(Data, Text, Heap) 하므로 생성 부하 적음
- 쓰레드간 통신이 쉬움(프로세스내 변수로 접근)
- 동시에 여러 로직 수행 가능(멀티코어 시대에 적합)
- 가벼운 context switching

Thread 단점

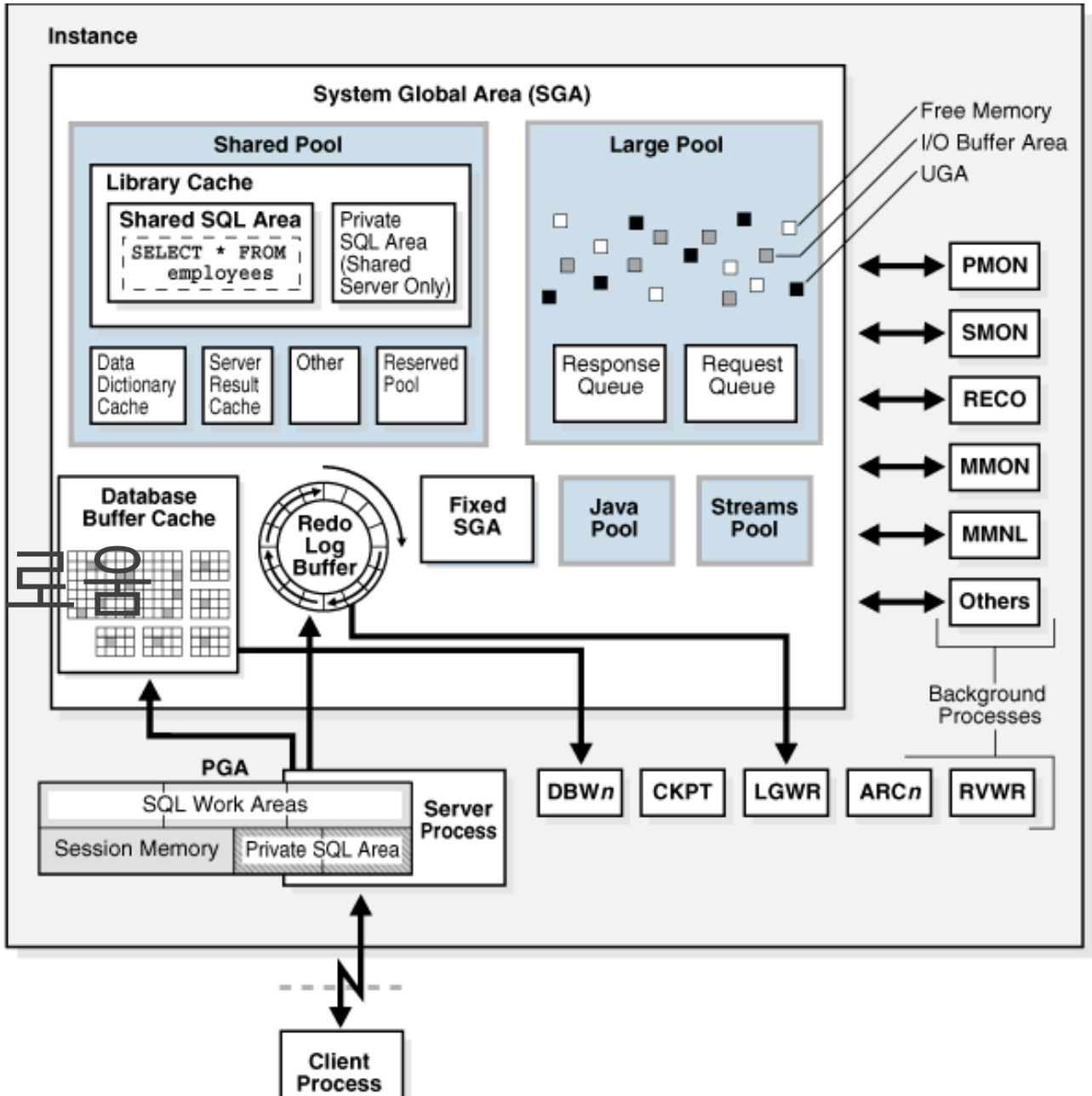
- 통신이 쉽고 동시에 실행될 수 있으므로 동시성 문제 발생 -> 동기화 기법 필요
- 특정 쓰레드에서 잘못된 동작시(널 포인터 접근등) 나머지 쓰레드로 장애 전파(프로세스 자체를 종료시킴)
- 실행 순서를 보장하지 않으므로 기존 비즈니스 로직 변경이 필요할 수도 있음
- 디버깅이 어려움

Process 장점

- 프로세스 모델이라고 나쁜 것은 아님
- 안정적인 장점이 있음(한 프로세스가 죽어도 전파 안 됨)
- DBMS 같이 죽으면 안 되는 프로그램의 경우 제품에 따라 멀티 프로레스로 동작

A2 – DBMS 가 멀티 프로세스 구조인 이유

- 오라클 DBMS 는 안정성을 위해 Multi Process 구조 채택
- 프로세스간 통신과 데이터 공유는 Shared Memory로 처리(SGA 영역)
- 이때문에 오라클 설치가 번거로움 (Kernel 의 IPC 설정 필요)



쉬어가기 - 2천만 달러 버그 #1

- 때는 93년 SUN Micro Systems
- 개발중인 Solaris 의 비동기 IO library 에서 랜덤하게 버그 발생
- 이로 인해 Sparc machine 출시가 6개월 지연되어 2천만 달러의 손해
- 최우선 순위로 버그로 등록하고 회사 역량을 총동원해 디버깅

쉬어가기 - 2천만 달러 버그 #2

- 원인은 *x==2;* 코드
- 대입문에 실수로 = 를 하나 더 사용
- C 언어 표준은 조건문이 아니더라도 비교 연산자 사용은 문법 에러가 아님
- 비슷한 문제로 if (x = 2) 가 있음(조건문내 대입 연산자)
- 당시의 컴파일러는 경고만 냈거나 또는 경고없이 컴파일(요샌 컴파일 에러)

쉬어가기 - 2천만 달러 버그 #3

- 레거시에서 다음 같은 코드를 보았을 때 반응
  - "왜 가독성 떨어지게 상수를 왼쪽에 썼냐" (X)
  - 버그를 방지하기 위한 선임자의 고민 결과(O)
- 상수를 왼쪽에 쓰면 실수로 대입 연산자 사용시 문법 에러

```
1  void func1(int x)
2  {
3      if (3 == x) { // 조건문에서 대입 실수 방지
4          // do something
5      } else if (4 == x) {
6          //
7      }
8  }
9
```

쉬어가기 -2천만 달러 버그 #4

- 프로그래밍 언어는 2 가지 종류라고 함
  - 욕 먹는 언어와 아무도 사용하지 않는 언어
- 언어마다 현실적 문제때문에 저런 불합리한 면을 갖고 있음
- 이를 피해서 사용하는 것도 중요한 코딩 테크닉(Ex: PHP 의 === 연산자)

Thread 만들기

- 유닉스에서 멀티 쓰레드 만들기는 매우 매우 쉬움 (성능을 고려하지 않는다면)
- 쓰레드로 만들 함수를 POSIX 표준 함수인 pthread_create 의 파라미터로 넘겨주면 끝!

Thread 동시성 – Mutex #1

- 멀티 쓰레드는 동시성 문제가 있다면서요 ?

- 동기화가 필요한 임계 영역(critical section)을 수행시 Mutex(mutual exclusion) 로 한 쓰레드만 진입하도록 처리
- POSIX 표준 함수인 *pthread_mutex_lock* 을 사용하면 mutex 를 획득한 쓰레드만 임계 구역 진입 가능

Thread 동시성 – Mutex #2

- 그럼 mutex 를 획득하지 못한 쓰레드는요?

- 나머지 쓰레드는 mutex 를 얻을 때 까지 blocking 됨
- 이때문에 잘못 만든 멀티 쓰레드 프로그램은 싱글 쓰레드보다 느릴 수 있음
  - 과다한 임계 영역 생성시 실제로는 싱글 쓰레드와 비슷해짐
    - mutex 동기화로 인한 부하(mutex는 비싼 함수)

Thread 동시성 – Mutex #3

- 그럼 mutex 호출시 blocking 방지는?

- *pthread_mutex_trylock* 사용시 이미mutex 획득 쓰레드가 있을 경우 바로 리턴(Non-Blocking)

- *mutex*가 필요하면 계속 시도해야 하므로 CPU 시간 소요

```
1  for (loop=0; loop<LOOPCONSTANT; ++loop) {
2      int rc = pthread_mutex_trylock(&mutex);
3      if (rc == EBUSY) { // 이미 다른 쓰레드가 mutex
4        sleep(1);
5        continue;
6      }
7
8      // mutex 획득, 동기화가 필요한 작업 수행
9      doSomething();
10
11     rc = pthread_mutex_unlock(&mutex);
12     printf("pthread_mutex_unlock()\n", rc);
13   }
```

Thread 동시성 – Cond Var

- 그럼 쓰레드는 딴 일 하다가 mutex가 해제되면 통보받음 되겠네요?

- mutex_lock, trylock 로 대신 mutex 해제시 알림 받는 함수가 POSIX에 정의됨
- POSIX는 이를 Condition variable 이라 명명
- mutex 대기 쓰레드는 phread_cond_wait 호출
- 사용이 끝난 쓰레드는 pthread_cond_signal로 신호 전송

Thread 동시성 – Cond Var

- 그럼 쓰레드는 딴 일 하다가 mutex가 해제되면 통보받음 되겠네요?

- mutex_lock, trylock 로 대신 mutex 해제시 알림 받는 함수가 POSIX에 정의됨
- POSIX는 이를 Condition variable 이라 명명
- mutex 대기 쓰레드는 phread_cond_wait 호출
- 사용이 끝난 쓰레드는 pthread_cond_signal로 신호 전송

Thread 동시성 - 중지와 재시작

- 실행중인 thread 를 외부에서 suspend, resume, stop, kill 하는 것은 심각한 문제 발생 가능
- 만약 suspend 시킨 쓰레드가 mutex 나 lock 을 갖고 있으면 deadlock 발생
- 이때문에 Java나 .NET 은 직접 쓰레드의 동작을 제어하는 메소드는 deprecated

Thread 동시성 - 정리

- mutex와 condition variable 은 운영체제, 언어, 프레임워크를 막론하고 멀티 쓰레드 환경에서 공통으로 사용하는 동기화 기법 (용어는 다를 수 있음)
-

Server Pool - #1

- 웹 서버등은 구동된 동안 지속적으로 서비스 제공 필요
- 클라이언트 서비스가 끝났다고 서버를 종료하는 것은 낭비
- 일정 갯수 이상의 서버를 미리 생성하여 부하를 줄이기 위한 아키텍처가 Pool
- Thread 방식일 경우 Thread pool, 프로세스는 Server Pool 이라 통칭

Server Pool - #2

- apache httpd, php-fpm 등은 서버 풀로 동작
- 서버 풀은 보통 다음 설정 제공(Ex: 아파치)
  - 최초 시작 프로세스 갯수(StartServers)
  - 클라이언트 급증에 대비하기 위한 최대 idle 프로세스 갯수(MaxSpareServers)
  - 클라이언트가 줄었을 때 기 생성된 프로세스를 줄이기 위한 최소 idle 갯수(MinSpareServers)

DB Connection Pool - #1

- 웹 서비스 특성상 많은 DBMS 쿼리 발생
- DBMS 연결은 많은 비용이 발생(connection 시간, DBMS 서버 프로세스 생성 필요등)
- 많은 WAS는 이런 문제 해결을 위해 사전에 DB Connection Pool 을 생성
- app 가 요청시 Pool에서 커넥션을 가져오고 app 가 반납하면 pool 에 추가

PHP DB Connection Pool

- PHP는 검증된 DB Connection Pool 이 없는 것 같음(SQLRelay ?)
- MySQL이 가볍고 connect/close 비용이 저렴한 편이므로 필요성이 적은 듯함
- 기업 환경에서 사용하려면 다양한 DBMS 에서 동작하는 pool 기능 필요

DB Pool Manager를 만든다면?

- DB Pool 내 자원 반출/회수 작업은 임계 영역. Mutex 나 lock 필요
- Pool 내 리소스(DB Connection)가 유효한지의 검사 책임은? (Pool Manager or App ?)
  - 기업내 DBMS 가 별도 네트웍일 경우 사용하지 않은 연결은 방화벽이 끊을수 있음
  - 자바의 특정 db pool 구현은 주기적으로 dummy 쿼리를 날리는 기능 보유

DB Pool Manager를 만든다면?

- Pool 내 커넥션이 늘어날 경우 반출/회수 속도 향상을 위해 다중 Pool 사용
- Pool내 30 개의 커넥션이 있는 것보다 3 개 Pool 내 각각 10 개 커넥션이 처리 속도 빠름

Multi Thread safety #1

- PHP의 Zend 엔진은 ZTS와 None ZTS 2개 종류가 있던데 차이는?

- ZTS(Zend Thread Safety) 는 이름 그대로 멀티 쓰레드에서도 안전한 PHP 엔진
- 여러 개의 쓰레드에서 동시에 호출되고 병렬로 실행되어도 안전해야 쓰레드 안전(Thread safety)라고 할수 있음
- 김선영님 블로그(http://sunyzero.tistory.com/97)

Multi Thread safety #2

- 내 코드가 쓰레드에서 안전하려면 별도의 수정이 필요한가요?

- 쓰레드 안전하려면 몇 가지 주의 사항이 있음
  - 원자성(atomic) 필요, static, global 변수 사용 X
- POSIX.1-2008 표준에 정의된 함수는 모두 쓰레드 안전하지만 아래 링크에 있는 함수는 예외 (http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_09_01)
- 컴파일시 특수 옵션 필요(gcc 의 경우 -pthreads)

재진입성(Reentrant)

- 쓰레드 안전 + 시그널 같은 비동기 호출에서도 안전한 함수
- 여러 쓰레드가 재귀 호출로 사용해도 안전
- 재진입성을 가지려면 함수를 특별하게 설계해야 함
- 재진입성 함수는 이름뒤에 _r 이 붙음
- asctime_r, ctime_r 등

참고 자료 & QNA

아래 도서는 제가 읽은지 오래되서 세부 내용은 가물가물하지만 목록 정리해 봅니다.

- Advanced Programming in the Unix Environment
- Unix network programing
- Expert C Programming: Deep C Secrets
- Programming with POSIX Threads
- 김선영님 블로그(http://sunyzero.tistory.com/)

- Q&A

감사합니다.