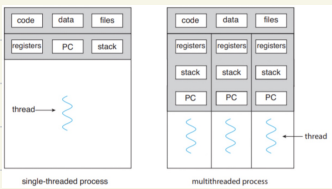
Thread

: a single execution sequence that represents a separately

Thread



- thread-specific resources
  - ✓ local stack of frames : 해당 Process의 stack frame 유지. 각자 따로 관리!
  - ✓ registers : 실제 레지스터(HW)를 각각 할당 받는게 아니라, 약간의 register는 context 개념 같이 레지스터가 있다고 볼 수 있음
  - ✓ scheduling property (priority)
  - ✓ thread-specific data … library supporting
    : 해당 process에서 파생된 thread들이 share 하는 전역 변수. 각 thread 마다 다른 element의 접근자를 가진다.

- Threads share code (text), data (global variable), heap, file 등
  - changes to shared value ⇒ should be viewed by others
  - ✓ shared among threads

  R/W to shared memory requires synchronization

강점 fork() 의 비해서 pthread_create()가 좋은 이유?

① light weight … HW resource를 share 하기 때문에 훨씬 light weight.

② responsiveness … process: block = whole block … no CPU usage
thread: block = use other thread!

③ Scalability … one processor threads를 multi-core systems에 parallel 하게 할 수 있음 ⇒ throughput ↑

POSIX Thread API (Pthreads)

Portable **O**perating **S**ystem **I**nterface

Pthread creation

```c
#include <pthread.h>
int
pthread_create (
    pthread_t    *thread_handle,
    const pthread_attr_t    *attribute,
    void *    (*thread_function)(void *),
    void    *arg);
```

- `*thread_handle` ← thread_id 저장
- `(*thread_function)(void *)` ← threaded execution sequence
- `*arg` ← argument of thread function

thread attribute

: thread specific resource를 설정하는 것

- stack size
- detach state : process가 terminate 되고 thread는 어떻게 처리할지
  - PTHREAD_CREATE_DETACHED : reclaim storage at termination
  - PTHREAD_CREATE_JOINABLE : retain storage 〃
- Scheduling policy
  - no priority → SCHED_OTHER: (RR manner)
  - yes priority (선착순, 그냥 지정 priority) → SCHED_FIFO, SCHED_RR:
    priority 순으로 실행하기
- scheduling parameters - only priority
- inherit scheduling policy : parent thread inherit / overwrite
  PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
- thread scheduling scope
  PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS

create 할때 속성을 create 해야하는거!

POSIX Thread API (Pthreads) (Cont.)

Pthread termination

handling child exit : Pthread creator procedure must handle child thread (created) termination

```
int
pthread_join (
    pthread_t thread,
    void **ptr);
```

creator calls!

] 한번 child thread 씩! thread_id, exit_status

How to terminate

- natural exit after thread function returns
- 강제적인 exit → pthread_exit()
- 다른 애가 exit 시킴 → pthread_cancel()
  - cancel signal 받아야지 cancel 진행되도!
    cancel point에서 cancel 됨
  - cancelled thread는 clean-up handler call 함
- Parent thread가 exit되면 child도 exit됨

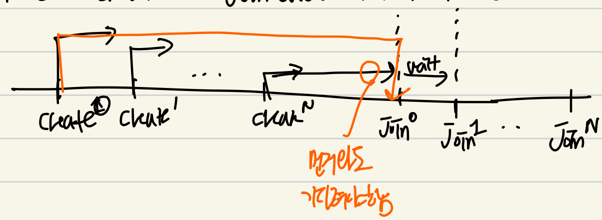
언제 terminate 되는지

Multi-threading example

Parallelization 하려고 하는거에서 data-parallel이 제일 쉬운 접근 방식

↳ operation 같고, data(input) 다름!

① thread create & join (wait termination)



② thread function execution

→ reentrancy 꼭 고려해야함 (re-enter 해도 consistency 유지)

: functionA()가 호출되는 도중에 interrupt가 발생해서 중간에 다른 procedure가 functionA()를 호출해도, **기존에 호출된 function에 영향 X**

↳ 그러려면 해당 함수 안에서 사용하는 전역변수에 영향이 없어야함

그러려면 reentrant function은 static/global 변수 사용 X

→ shared variable 에서 주의 해야함

: 왜냐면, shared variable에 대한 다른 thread들의 write는 false sharing 발생 확률 개높음!!!!!

보통 update 하는 variable 또는 code line 만큼 크지 않고, 만약 하나의 Array 안에서 각 thread가 element에 접근한다면 false sharing ⇒ performance 최악

✓ good performance

- shared variable에 대해 false sharing 안일어나게 local variable로 접근한다면, degree of concurrency 만큼 speed up 상승 할듯!?

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

← shared variable (*s)

re-entrancy 보장

잦은 update는 local variable로!

마지막 최종에 한번 적용!

← 만약, false sharing 없이 update 하려면!? 2D로 만들어서 [[ ],[ ],[...]] size = cache line (padding)

Synchronization Primitives

multiple threads 들이 shared variable(critical section)의 write 할때 값이 깨지는 현상 방지

Common case : 보통 shared variable update는 3 instructions로 이루어짐

① load from memory (data section) to register
② update register
③ store register value to memory (data section)

과정의 사이에서 interrupt 발생, inconsistency ↑

· 예) `int x=0;` (global)
`Inc(){ x=x+1 }`

⇒

thread 1
load x
add x
store x ①

thread 2
load x
add x
store x ②

synchronized

undesired value ⇒

thread 1
load x
add x 값=1

thread 2
load x
add x 값=2
store x ②

thread 2
store x 2

Synchronization Primitives (Cont.)

critical section : global variable이 write 되는 code 부분

⇒ Should be executed by only one thread at a time

mutex lock : to achieve mutual exclusion in critical section when using Pthread

How to use

⓪ initiate mutex lock

① request lock before executing critical section

② enter critical section when lock is acquired

③ release lock when leaving critical section

(request lock, release lock → atomic operation supported by HW)

type (init 할때 설정됨) : lock owner가 또다시 same lock에 대해서 request 하면!?

- Normal : thread deadlocks
- recursive : 무한히는 안종 계속 lock request 가능
  - 접근 할때마다 +1 하고, 나중에 unlock 할때마다 -1 하기
  - lock이 0이 되면 relinquish lock!

  … 그제서야 다른 애들도 lock 가질 수 있는거임

(semaphore랑 다름)

```
void foo()
{
    pthread_mutex_lock(&m);
    …
    bar();
    pthread_mutex_unlock(&m);
}

void bar()
{
    pthread_mutex_lock(&m); //
    …
    pthread_mutex_unlock(&m);
}
```

thread 1	thread 2
thread 1이 lock 갖고있음	접근 불가능함
thread 1은 또 request 날리고. resume! no lock

thread 1만 접근 할수있는데 lock 잡기 위해 기다리는 거임!

Producer-Consumer using Mutex Locks

Constraints	when task queue can hold only 1 task (task queue → bounded buffer)
Producer	: task queue가 다 차있으면 produce 중지
Consumer	: task queue가 비어있으면 consume 중지
Busy-waiting version	: 기본적인 mutex_lock은 unavailable lock에 대해 접근하면 block 되어서

Overheads of Locking

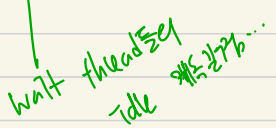
- Locks enforce serialization
  : threads must execute critical section one at a time

- large critical section can degrade performance
  - blocked time 길어지는것 responsiveness 떨어짐 ...

  ⇒ 최대한 critical section 코드를 줄이는 것이 좋다!
  최대한 Compact 하게 !

- waiting (block) 되어있는 동안, computation 진행하도록 해줄것도 방법!
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex_lock)`
    → if lock is unavailable, do something else and not waiting !
    → 문제점: 대부분 lock 불가능!!!

- locker owner 만이 release 할수 있는 mutex exclusive 방법이다
  - lock : lock available / unavailable
  - semaphore : semaphore 가 0 or not
  - Conditional variable : waits condition ... signal() / wait()



Conditional Variable

waiting on some condition for shared state (predicate)

APIs

- wait()
  - automatically release lock and yield processor
    - thread 상태를 ready로 바꿈!
  - reacquire lock when woken up by signal()
- signal()
  - wake up waiting thread if any (can be no-effect)) priority 있는 경우
- broadcast()
  - wake up all waiting threads) priority 없는 경우

memoryless ☆

: Past의 사실이 current action에 영향을 안줌.
즉, 함수를 호출하면 오직 1개의 action만 존재하는 → * lock이나 semaphore의 경우, shared variable의 값에 따라서 action 달라짐

- only internal state ... queue of waiting threads
- no memory of earlier signal()/broadcast() ... 상관없이 무조건 wait ☆
- signal()/broadcast() has no side effect when empty waiting queue

spurious approach 오히려 정확해서 좋음

Predicate ☆

: signal()에 의해서만 wake up 하는게 정상인데, 안그럴 수도 있으니까 (error prevention/avoid)
while(predicate unsatisfied)로 항상 double check 해주기

thread1

```
action() {
    ...
    mutex_lock(&lock);
    while (predicate == 0)  // test predicate
        cond_wait(&cond, &lock);
    mutex_unlock(&lock);
    // perform action
    ...
}
```

(annotations: acquire lock, wait condition, release lock)

thread2

```
signal() {
    ...
    mutex_lock(&lock);
    predicate = 1;  // set predicate
    cond_signal(&cond);   — wake up 1 waiting thread
    mutex_unlock(&lock);
    ...
}
```

Spurious wakeup을 막기 위해서! (가짜의)

* waiting 하는 입장에서 waking up 되면 signal/broadcast/malicious 등이 온거 알 길이 없음! 그래서 predicate re-check 해야함

Conditional Variable (Cont.)

- use **predicate** and **lock** to enter critical section

- How to make wait with predicate and lock
  1. thread lock a mutex
  2. Test predicate defined on shared variable ... 필요한 signal 전!
     → If(predicate == false) ... wait on condition variable
     ... unlock associated mutex
     (allow other can hold it)

- How to make signal with predicate and lock
  1. thread lock a mutex
  2. Set predicate true
  3. signal conditional variable
     → signal (wake up 1 thread) / broadcast (wake up all)
  4. return lock

생산자와 소비자 이론!

*
```
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *wtime);
```
abort wait if time exceeded

: signal 받기전에 wtime 이 expire 되면 error message return!

Producer-Consumer using Cond. Variable

```
void *producer(void *producer_thread_data) {
    int inserted; task_t *t;
    while (!done()) {
        t = create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (work_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                &task_queue_cond_lock);
        consumer_work = t;
        work_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

true predicate
⇒ work-available == 0

releases mutex on wait

⇒ predicate is false

wait and release

consume

⇒ signal to consumer

update predicate

reacquires mutex when woken

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (work_available == 0)
            pthread_cond_wait(&cond_queue_full,
                &task_queue_cond_lock);
        my_task = consumer_work;
        work_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

releases mutex on wait

⇒ predicate is false

true predicate
⇒ work-available == 1

reacquires mutex when woken

… 같이 while 없는 이유는, correct signal 인지 판단하려고 함

Reader-Writer problem

Acquire read lock	concurrent reader 가능 • other thread가 read lock을 잡고 있어도 critical section 접근가능 (Read니까!) • condition wait on write lock on data or queued write locks writer의 starvation 막기 위해서!
Acquire write lock	only exclusive writer • condition wait on multiple threads request a write lock

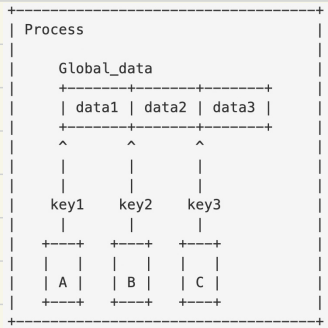
Thread-Specific Data

Thread는 Heap, Data Segment를 공유하기 때문에 global variable = shared by others
= others can access it.

- associate some data with a thread
  - pass data as argument to each call thread makes
  - store data in shared variable indexed by tid
    → false sharing problem 발생가능
  - using thread-specific keys
    ↳ 예외, global variable은 shared by all threads 라서!
    특정 thread가 접근한 data를 만드는건 관리 불가능

* fork()는 private global variable 가능

Thread-Specific keys



: global variable 처럼 TSD는 key를 갖고 있어야만 접근가능
해당 key는 thread-specific 함! 즉, 1 thread나 2 child 간에 공유가능!

- library가 internal state 저장
- client는 key 값을 몰라도 됨 → Abstraction

```
int pthread_key_create(pthread_key_t *key, void (*destroy)(void *))
int pthread_setspecific(pthread_key_t key, const void *value)
void *pthread_getspecific(pthread_key_t key)
```

associate (key, value)

key를 가진, 해당 key에 해당하는 value 받기

값저장용

heap은 shared by threads

```
#include <pthread.h>
static pthread_key_t profiler_state;
initialize_profiler_state() {
  ...
  pthread_key_create(&profiler_state,
                     (void *) free_profile);
  ...
}
void free_profile(profile *my_profile) {
  free(my_profile);
}
```

opaque handle used to locate thread-specific data

destructor for key value

```
void init_thread_profile(...) {
  profile *my_profile = (profile *) malloc(...);
  pthread_setspecific(profiler_state, (void *) my_profile);
  ...
}
void update_thread_profile(...) {
  profile *my_profile = (profile *)
            pthread_getspecific(profiler_state);
  // update profile
}
```

only this thread만 접근가능