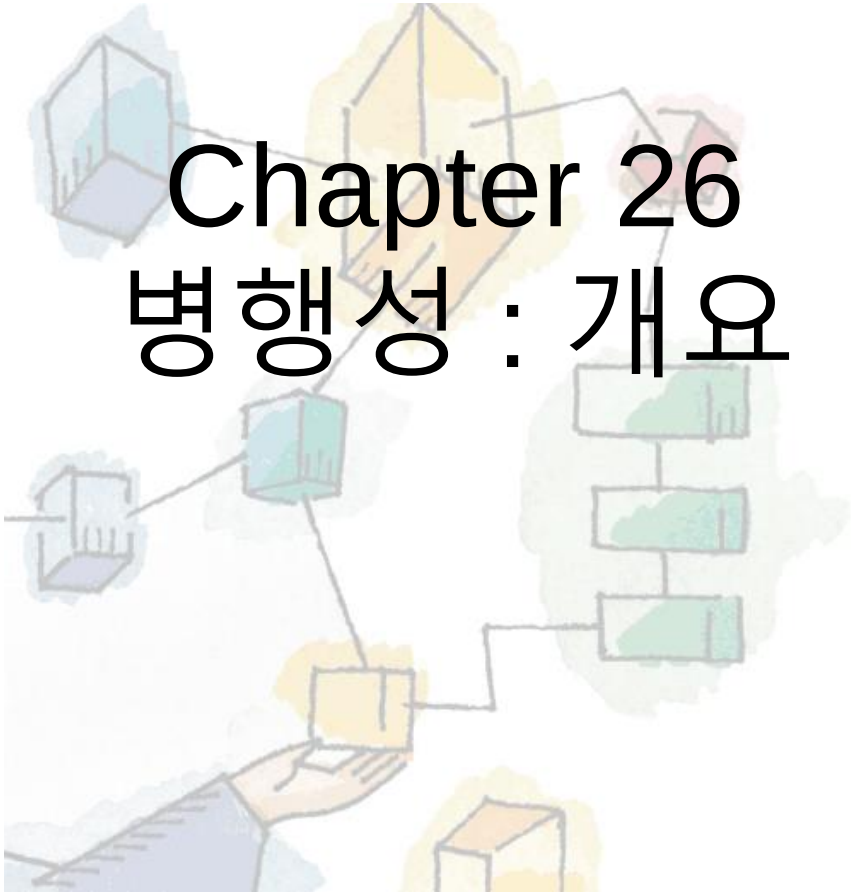
Chapter-18

운영 체제

정내훈

2023년 가을학기
게임공학과
한국공학대학교



Chapter 26
병행성 : 개요

쓰레드

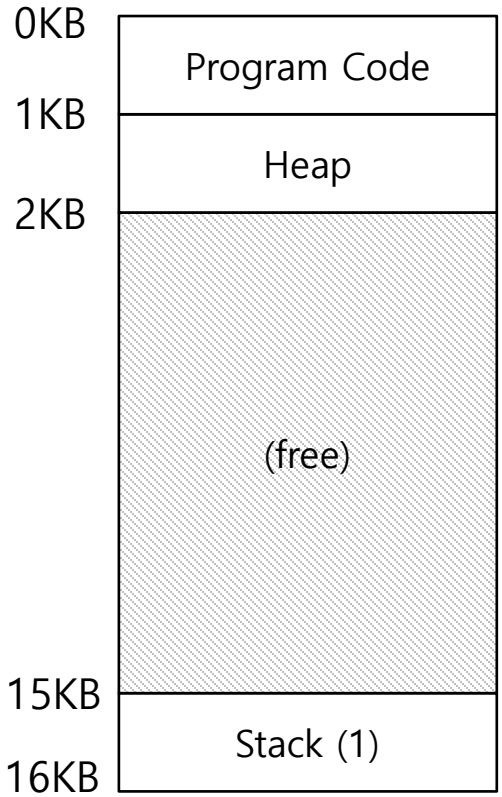
- 프로세스에 도입된 새로운 개념
  - 실행을 표현
  - 프로세스의 구성요소
  - 프로세스가 자체적으로 **멀티코어**를 활용하는 방법
- 멀티쓰레드 프로그램
  - 멀티쓰레드 프로그램은 여러 개의 실행 지점을 갖는다.
  - 여러 개의 PC (Program Counter)
  - 같은 주소 공간을 공유한다.

쓰레드 사이의 문맥교환

- 각 쓰레드는 자신만의 PC와 레지스터 집합을 갖는다.
  - 각 쓰레드는 자신의 쓰레드 제어 블록(**thread control blocks, TCB)**을 갖는다.
- T1 쓰레드에서 T2쓰레드로 교환할 때.
  - T1의 레지스터 집합이 저장된다.
  - T2의 레지스터 집합이 로딩된다.
  - 주소 공간은 그대로 유지된다.

연관된 쓰레드의 스택

- 쓰레드 마다 하나의 스택.

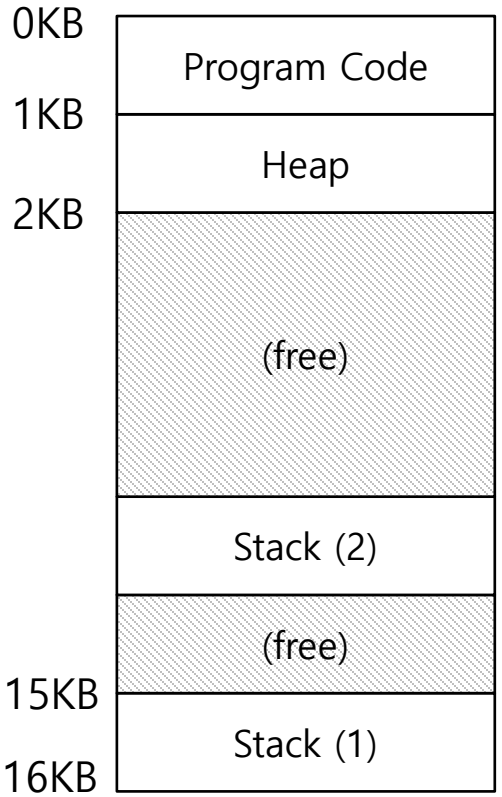


A Single-Threaded Address Space

The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures (it grows downward)

(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.



Two threaded Address Space

쓰레드 생성

- 쓰레드 생성 API는?
- C++11의 표준 API가 존재 한다.
- 이전에는 OS마다 서로 다른 API를 사용했어야 했다.

예제: 쓰레드 생성

```
#include <iostream>
#include <thread>
using namespace std;

void mythread(const char *mess)
{
   cout << mess << endl;
}

int main()
{
   cout << "main: begin₩n";
   auto p1 = thread{ mythread, "A " };
   auto p2 = thread{ mythread, "B " };
   p1.join();
   p2.join();
   cout << "main: end₩n";
}
```

- 문제
  - "A "가 먼저 출력될지 "B "가 먼저 출력될지 알 수 없다.

문제 : 데이터의 공유

- 멀티쓰레드 프로그래밍은 하나의 작업을 여러 개로 나누어 여러 개의 쓰레드에서 동시 병렬적으로 수행하는 것
  - 쓰레드 사이의 정보 교환이 필수이다.
    - 전역변수를 사용한 데이터 공유를 사용한다.

예제: 데이터 공유

```
#include <iostream>
#include <thread>
using namespace std;

volatile int counter = 0;
int loops = 100000000;

void mythread(int count)
{
   for (int i=0;i<count;++i)
        counter++;
}

int main()
{
   thread p1, p2;
   p1 = thread{ mythread, loops / 2 };
   p2 = thread{ mythread, loops / 2 };
   p1.join();
   p2.join();
   cout <<  “counter =  “ << counter << endl;
}
```

문제 : 경쟁 조건 (Race Condition)

- 앞의 프로그램의 실행 결과가 예상과 다르게 나온다.

- 왜?

- 쓰레드 사이의 작업 순서 정리가 없이 마구잡이로 실행되었기 때문.
- 이러한 현상을 경쟁 조건, 경쟁 상태, Race Condition 또는 Data Race라 부른다.

경쟁 조건

- 두 스레드에서 counter = counter + 1 (default is 50)
  - 결과 값이 52가 되어야 한다.

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov eax, [counter]		105	50	50
	add eax, 1		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov eax, [counter]	105	50	50
		add eax, 1	108	51	50
		mov [counter], eax	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov [counter], eax		113	51	**51**

임계 영역

- 임계 영역(critical section)
- 공유 변수에 접근하는 코드로 여러 스레드에서 동시에 수행되면 안되는 부분
  - 임계 영역을 여러 스레드가 동시에 수행하면 경쟁 조건이 발생한다.
  - 임계 영역은 **원자적(Atomic)**으로 수행해야 한다. (상호 배제, **mutual exclusion**)

락(lock)

- 이러한 임계 영역들이 하나의 원자적 명령어인 것처럼 실행되게 하는 함수. (여러 개의 명령어를 원자적으로 실행).

```
1    mutex mylock;
2     . . .
3     mylock.lock();
4     balance = balance + 1;
5     mylock.unlock();
```

Critical section

27. 막간 : 쓰레드 API

쓰레드 생성

- 어떻게 쓰레드를 생성하고 다루는가?

```
#include <thread>

std::thread thread_object { thread_function, arg1, arg2, arg3… };
```

  - thread: 쓰레드 클래스.
  - thread_function : 생성된 쓰레드가 실행할 함수.
  - arg: thread_function에 넘길 매개변수, 없어도 됨
    - thread_function의 선언에 사용된 매개변수와 일치해야 함..

예: 쓰레드 생성

```
#include <thread>

void thread_function(int a, int b) {
        cout << a << ", " << b << endl;
}

int main() {
        int a = 10;
        int b = 20;        …
        std::thread p {thread_function, a, b);
}
```

쓰레드 종료

```
void thread::join();
```

- thread클래스의 멤버함수
- 쓰레드 객체의 종료를 기다린다.
  - busy waiting없이 기다린다.

예 : 쓰레드 종료 기다리기

```
#include <iostream>
#include <thread>
using namespace std;

struct myret_t {
    int x;
    int y;
};

void thread_func(int a, int b, myret_t *r)
{
    cout << a << ",  " << b << endl;
    r->x = 1;
    r->y = 2;
}

int main()
{
        myret_t m;
        int a = 10;
        int b = 20;
        thread p {thread_func, a, b, &m };
        p.join();
        cout << "returned: " << m.x << ", " << m.y << endl;
}
```

락(Locks)

- 임계 영역에 상호 배제를 제공한다.
  - 락 객체를 사용한다

```
#include <mutex>
std::mutex mylock;
```

    - 객체 생성시 자동으로 초기화 한다.
  - 인터페이스

```
void mutex::lock();
void mutex::unlock();
```

락(Locks)

- 임계 영역에 상호 배제를 제공한다.
  - 사용법

```
std::mutex mylock;
mylock.lock();
x = x + 1; // or whatever your critical section is
mylock.unlock();
```

  - lock을 갖고 있는 쓰레드가 없을 경우 -> lock을 얻고 임계 영역에 진입.
  - 다른 쓰레드가 lock을 갖고 있을 경우 -> lock을 얻을 때 까지 리턴하지 않음.

Locks (Cont.)

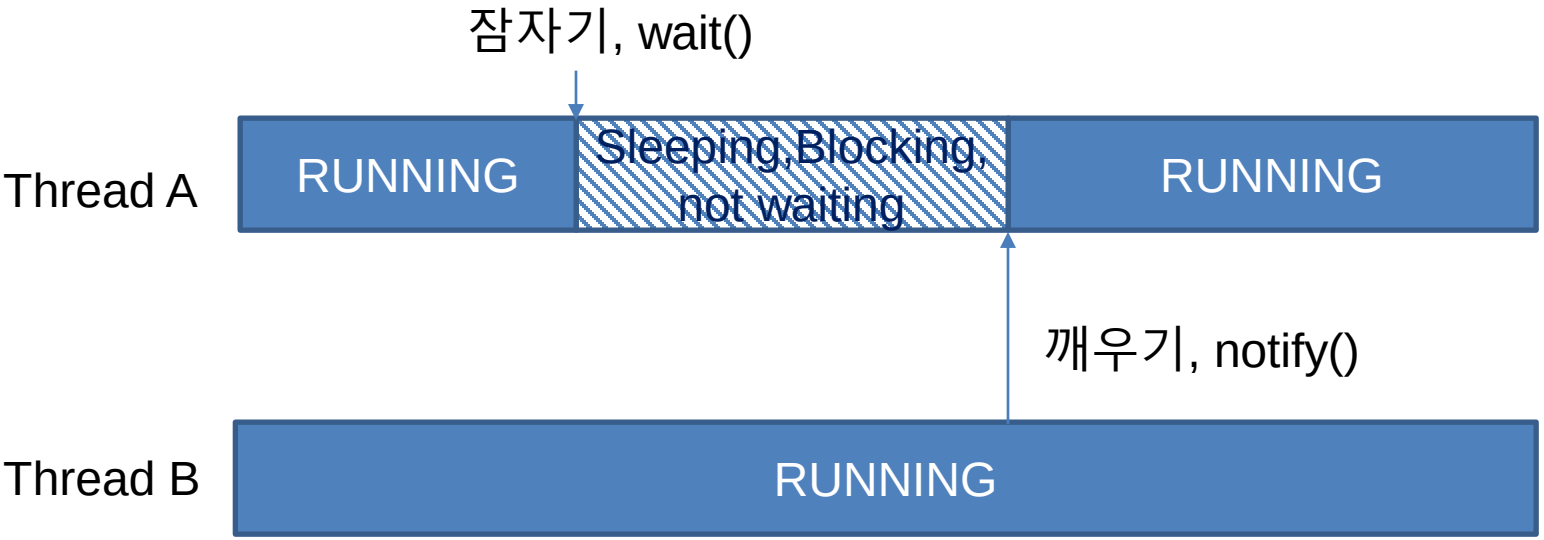
- 다른 사용법
  - try_lock: 이미 락의 획득에 실패하면 false를 리턴, 락 획득에 성공하면 true를 리턴.

```
bool mutex::try_lock()
```

    - 락을 가지고 있는 쓰레드가 없다고 해서 무조건 성공하는 것은 아니다.
    - 락을 이미 가지고 있는 쓰레드에서 호출했을 경우 어떻게 될지 알 수 없다.

컨디션 변수

- **컨디션 변수(Condition variables, 조건 변수)** 쓰레드 사이에 신호를 주고 받을 때 유용하게 사용.
  - 멈춰 있는 쓰레드를 깨울 때 사용



컨디션 변수

- **컨디션 변수(Condition variables, 조건 변수)** 쓰레드 사이에 신호를 주고 받을 때 유용하게 사용.
  - 신호가 올 때 까지 기다려야 할 때 대기상태에서 기다리게 해서 CPU 낭비를 줄여 줌
  - 커널 호출 필요 (thread의 상태를 바꾸므로)
- 컨디션 변수 객체 필요.

```
#include <condition_variable>
std::condition_variable cv;
```

컨디션 변수(계속)

```
mutex  cv_m;
conditional_variable cv;
```

- wait를 호출하고 대기하는 쓰레드:

```
cv_m.lock();
while (false == initialized)
        cv.wait(cv_m);
cv_m.unlock();
```

  - wait호출은 호출한 쓰레드가 대기 상태가 되면서 lock을 반납한다.
  - 하지만 wait가 깨어나서 복귀할 때 lock을 다시 획득한다.

- 쓰레드를 깨우는 코드 :

```
cv_m.lock();
initialized = true;
cv.notify();
cv_m.unlock();
```

컨디션 변수(Cont.)

- 대기하는 쓰레드는 while 루프에서 조건을 반복 검사한다. 단순히 if 으로 한번 검사하지 않는다.

```
cv_m.lock();
while (initialized == 0)
        cv.wait(cv_m);
cv_m.unlock();
```

  - 재검사를 하지 않으면, 대기하는 쓰레드는 컨디션이 변경되지 않았음에도 변경되었다고 판단할 수 있기 때문이다.

컨디션 변수 (계속)

- 이런 식의 프로그래밍은 금지.
  - wait하는 쓰레드:

```
while(initialized == 0); // spin
```

  - 깨우는 쓰레드:

```
initialized = 1;
```

  - 성능이 좋지 않다. → CPU 낭비 (Busy Waiting)
  - 에러 발생 : 컴파일러 에러, CPU 에러

컨디션 변수 (계속)

- 샘플

```
#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

using namespace std;
using namespace chrono;

condition_variable cv;
mutex cv_m;
int i = 0;

void waits()
{
    unique_lock<mutex> lk(cv_m);
    cout << "Waiting... ₩n";
    while (1 != i) cv.wait(lk);
    cout << "...finished waiting. i == 1₩n";
}
```

컨디션 변수 (계속)

- 샘플

```
void signals()
{
    this_thread::sleep_for(1s);
    cout << "Notifying...₩n";
    cv.notify_all();
    this_thread::sleep_for(1s);
    i = 1;
    cout << "Notifying again...₩n";
    cv.notify_all();
}
int main()
{
    thread t1(waits), t2(waits), t3(waits), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    system("pause");
}
```

컨디션 변수 (계속)

- 샘플
  - `1s` : <chrono>에서 사용 1초를 뜻함. (`1m` = 1밀리 세컨드)
  - `this_thread::sleep_for(시간)` : 시간동안 쓰레드 자신을 대기상태로 놓는다.
  - `conditional_variable::wait(unique_lock)`
    - unique_lock을 해제하고 대기상태가 된다. 대기상태가 풀릴 때 unique_lock의 획득을 먼저 한다.
  - `conditional_variable::notify()`
    - 대기상태의 쓰레드를 하나 깨운다.
  - `conditional_variable::notify_all()`
    - 모든 대기상태의 쓰레드를 깨운다.

컨디션 변수 (계속)

- 샘플
  - unique_lock<락_타입> 유니크락_객체(락_객체) : 락_타입의 락_객채의 락을 획득하고 비 프로그래밍 블록이 끝날 때 해제하며, 이 관리를 유니크락_객체가 한다.
    - 해제가 자동으로 되므로, 프로그래밍 오류를 줄이기 위해 사용

```
mutex cv_m;

cv_m.lock();
// 임계 영역
cv_m.unlock();
```


```
mutex cv_m;

{
   unique_lock<mutex> lk(cv_m);
   // 임계 영역
}
```

컨디션 변수 (계속)

- 너무 어려워 하지 말자. 뒤에 가서 차근 차근 다시 다룬다.

11월 1일 예비군

- 휴강
- 12월 3째 주 보강예정

- (어린이날, 추석, 예비군, 전국 선거일)