Philosophers

Let’s go see how philosophers DIE👻

21.11.21
Yongjule

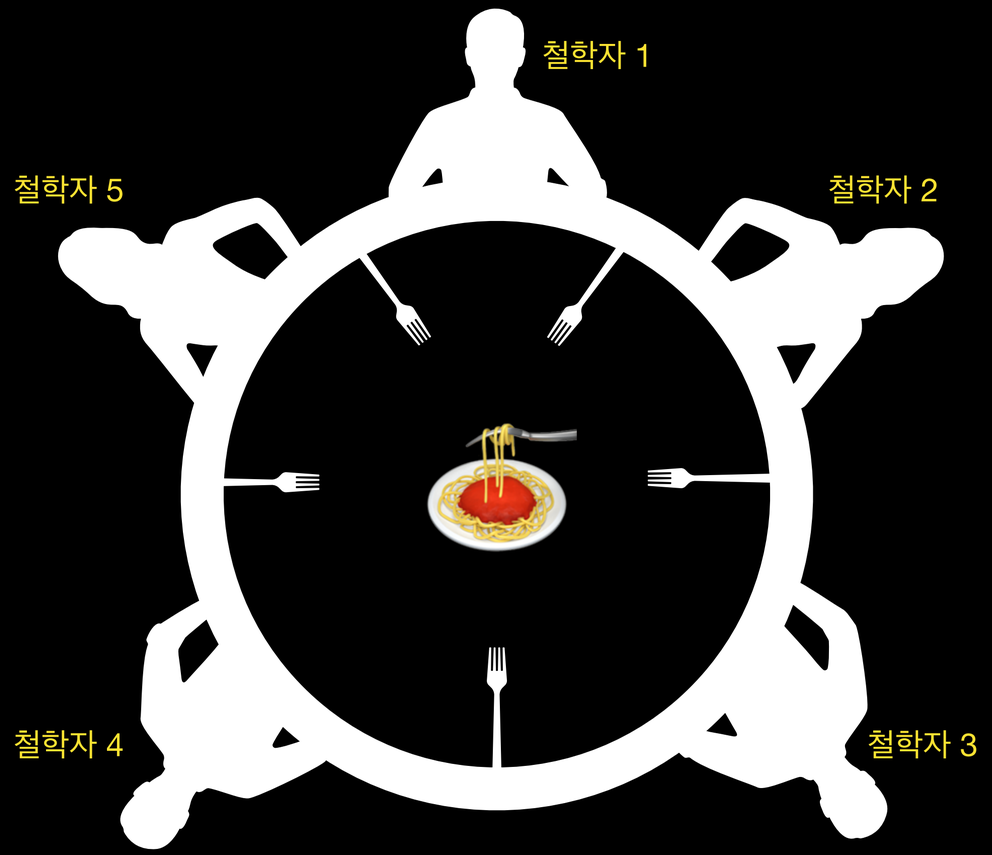
Philosophers

About philosophers

- 철학자를 thread로 생성
- 각 철학자는 식사 -> 수면 -> 생각 -> 식사를 반복
- 일정 시간 이상 굶으면 철학자는 사망
- 사망시 프로그램 종료
- (optional) 모든 철학자가 식사를 n번 이상 하면 종료

Philosophers

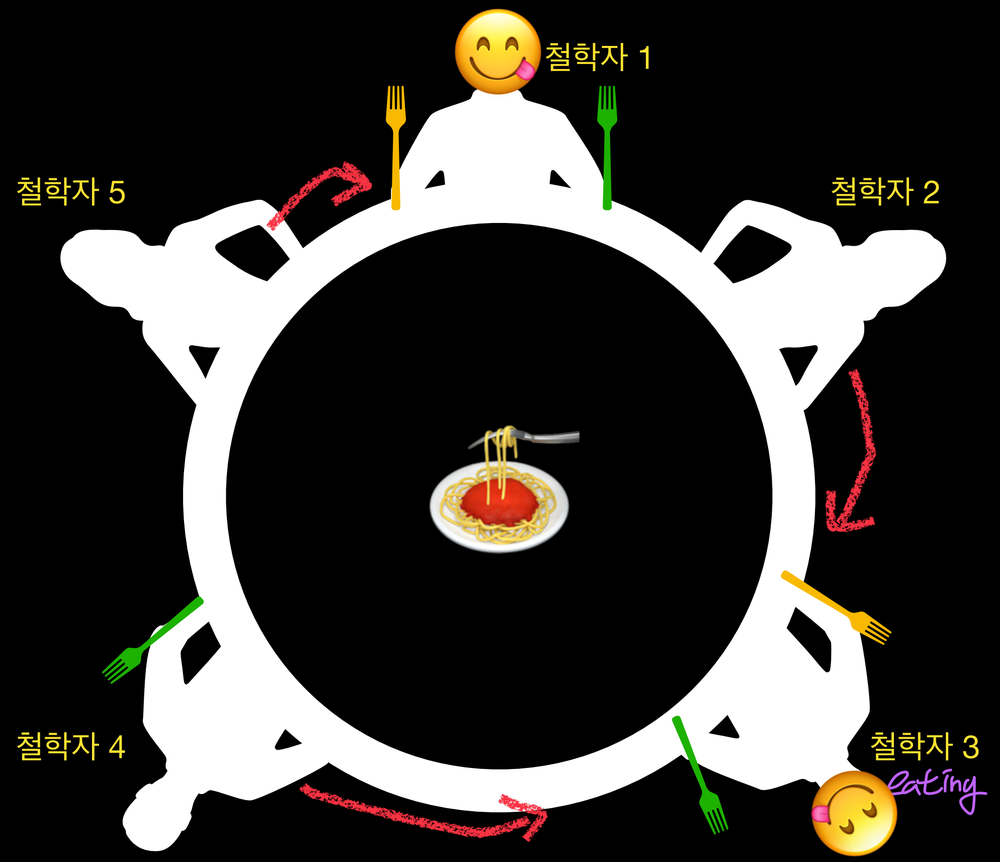
About philosophers



- 철학자는 원형으로 앉아있음
- 철학자 사이에 포크 하나씩
- 두개의 포크를 들어야 먹을 수 있음
- 사용중인 포크는 뺏지 못함(non-preemptive)

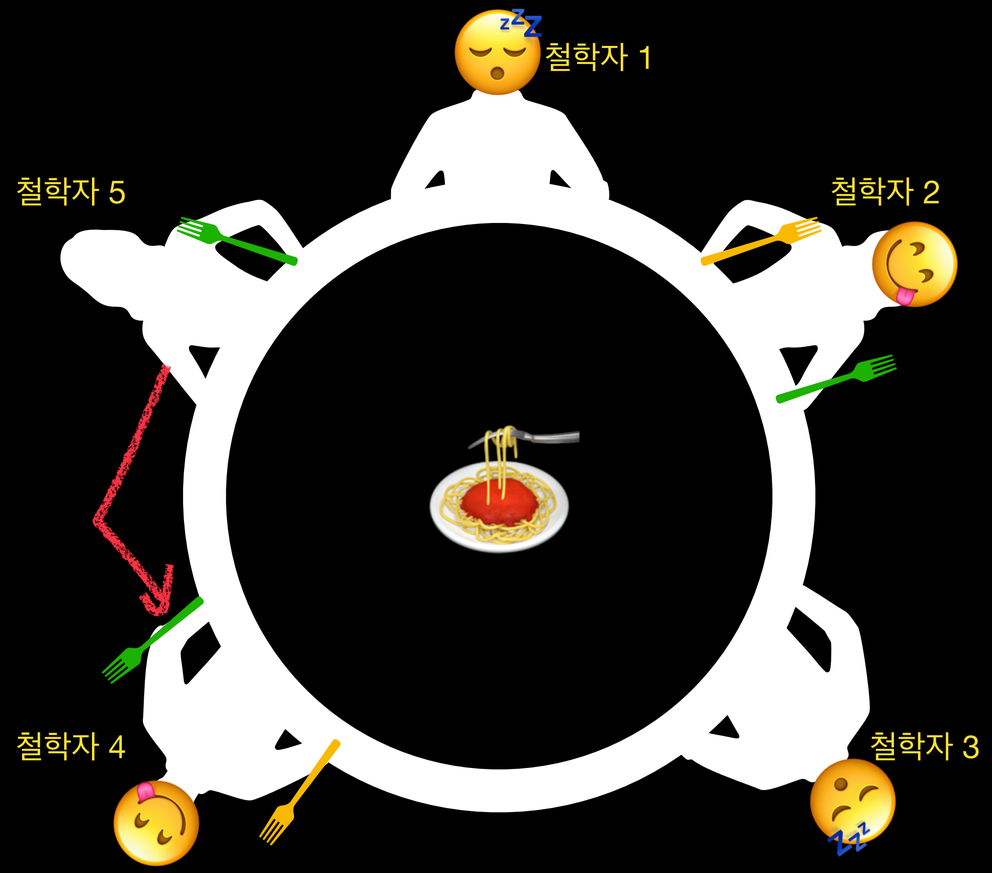
Philosophers

Eating



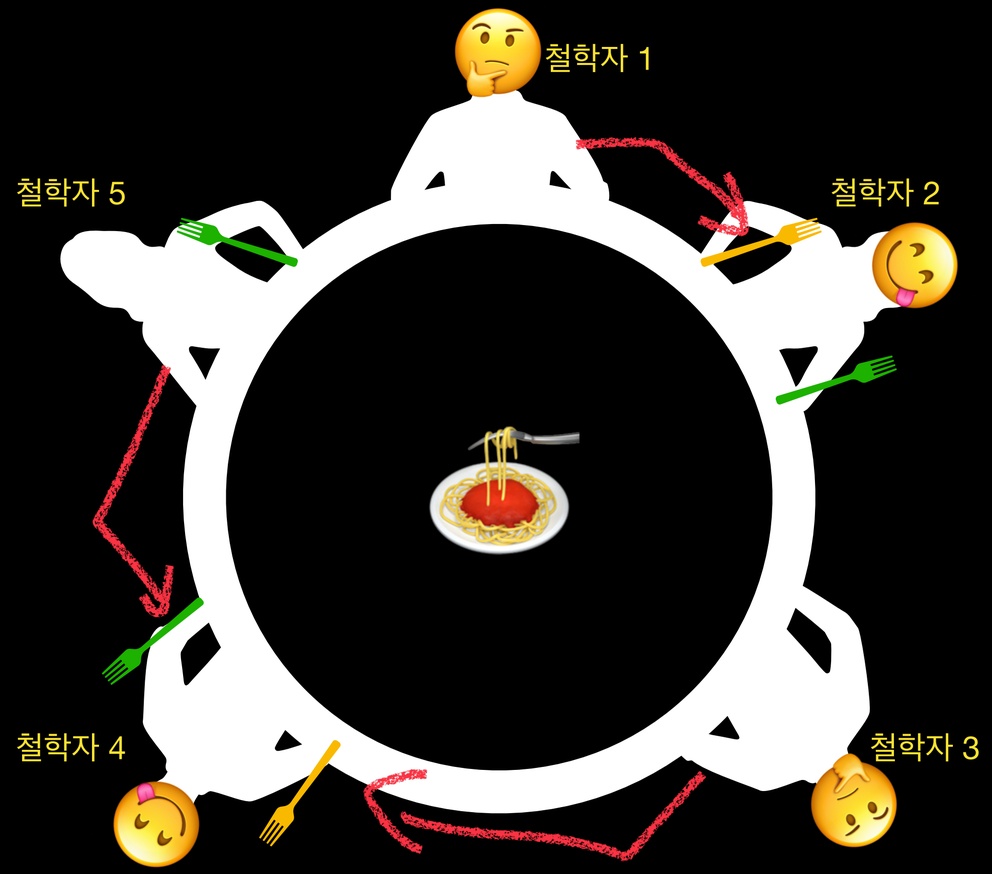
Philosophers

Sleeping



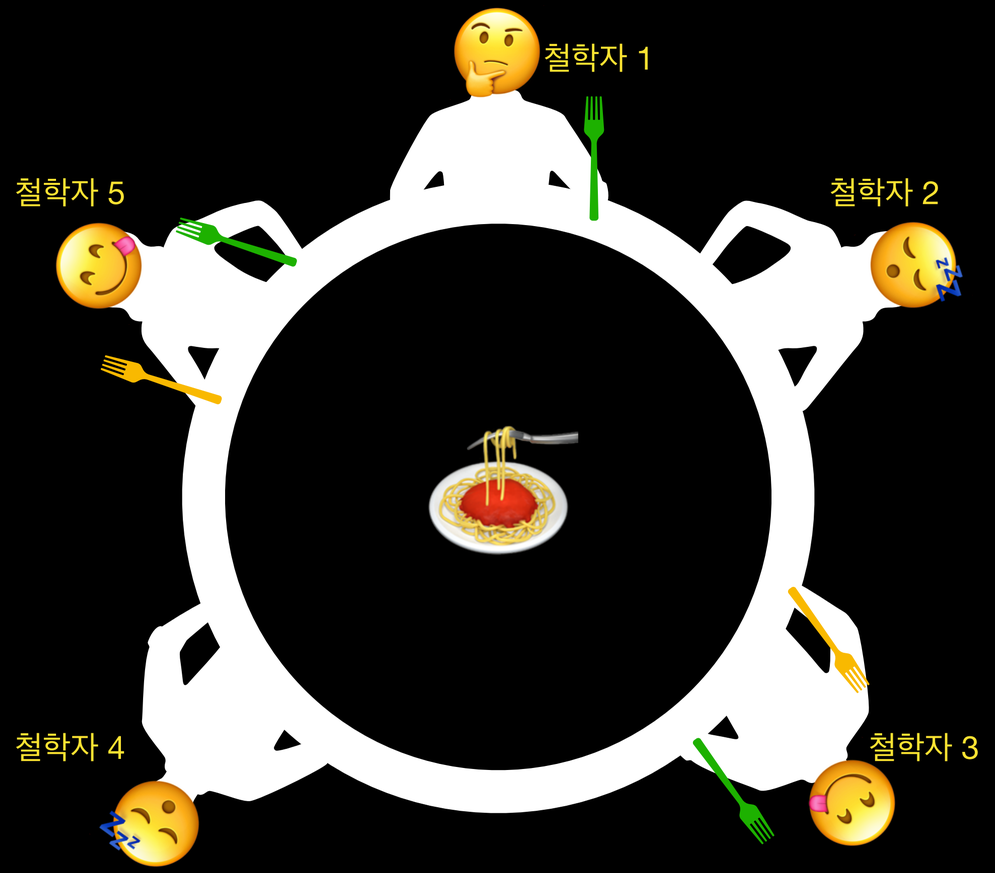
Philosophers

Wake up (time_to_sleep < time_to_eat)

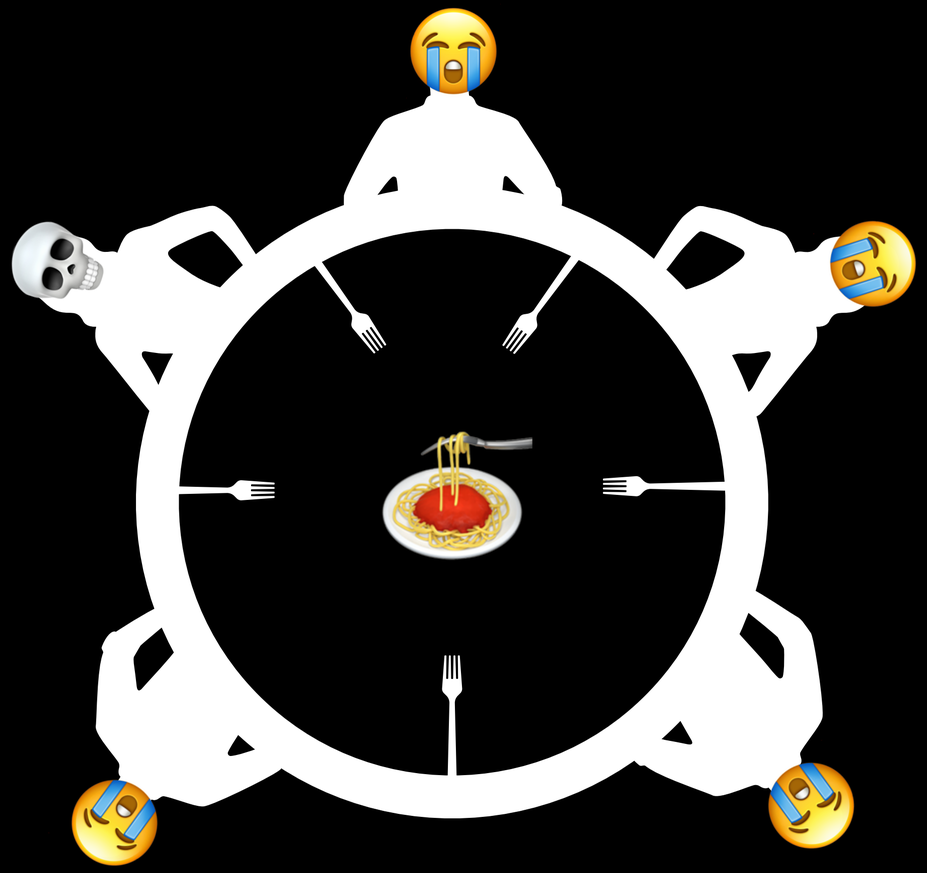


Philosophers

Cycle



Die (if time_to_die < time_to_eat + time_to_sleep)



Thread

Thread

About thread

Thread

- component of process - philosophers

TMI

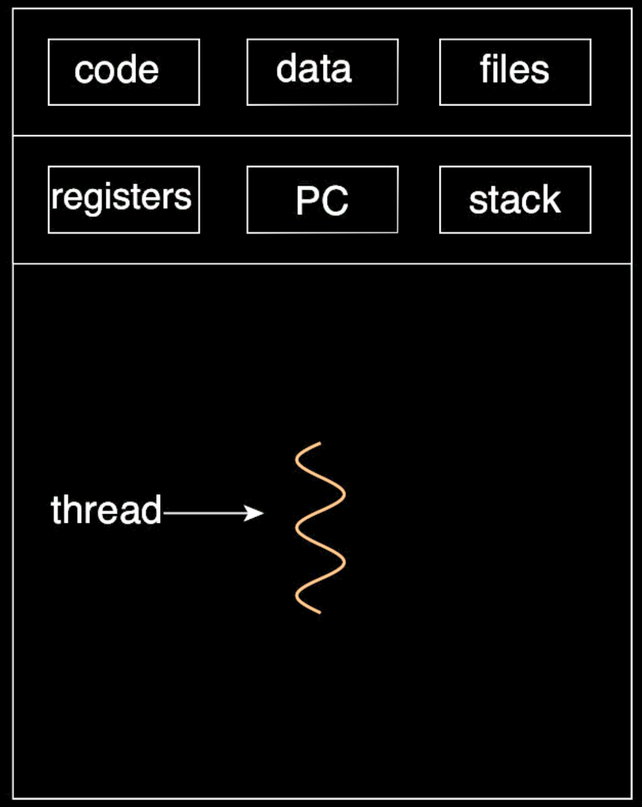
- hardware thread vs software thread, kernel thread user thread LWP
- basic unit of CPU utilization (from 공룡책)
- 프로세스 내에서 실행되는 흐름의 단위

“locus of control within an instruction sequence”

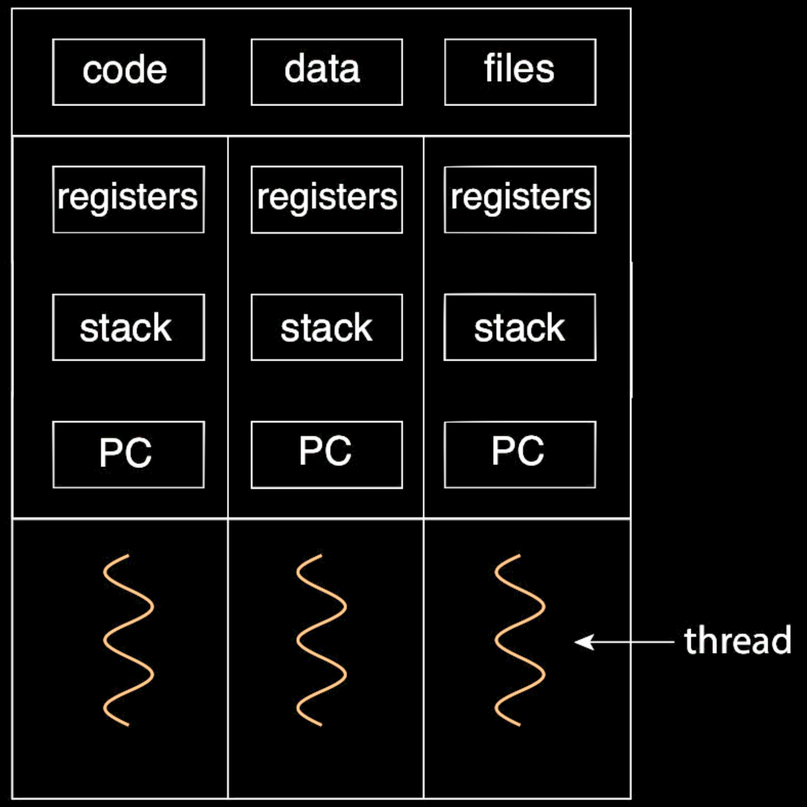
- Traffic Control in a multiplexed Computer System, by Jerome Howard Saltzer (1966), suggested by Vyssotsky

Thread

About thread for philosophers



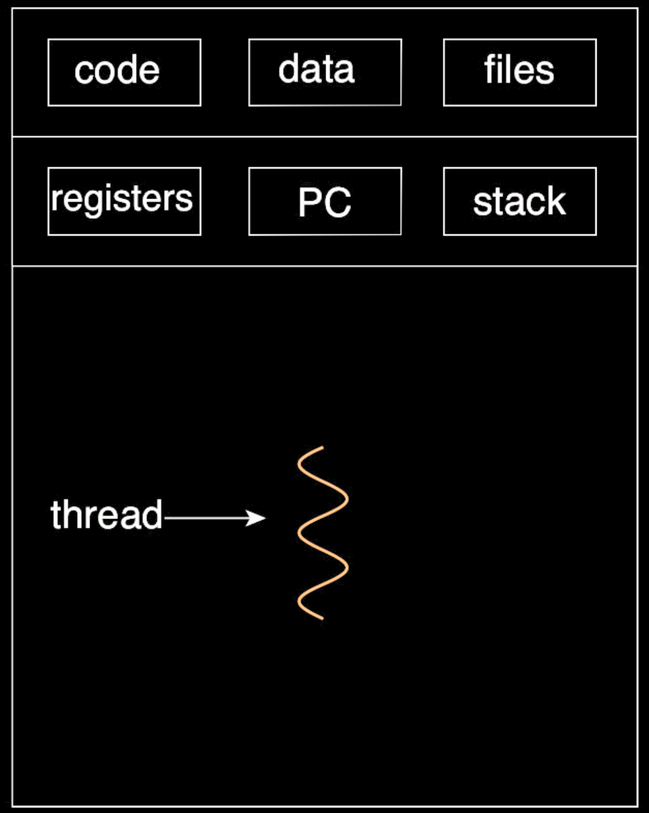
single-threaded process



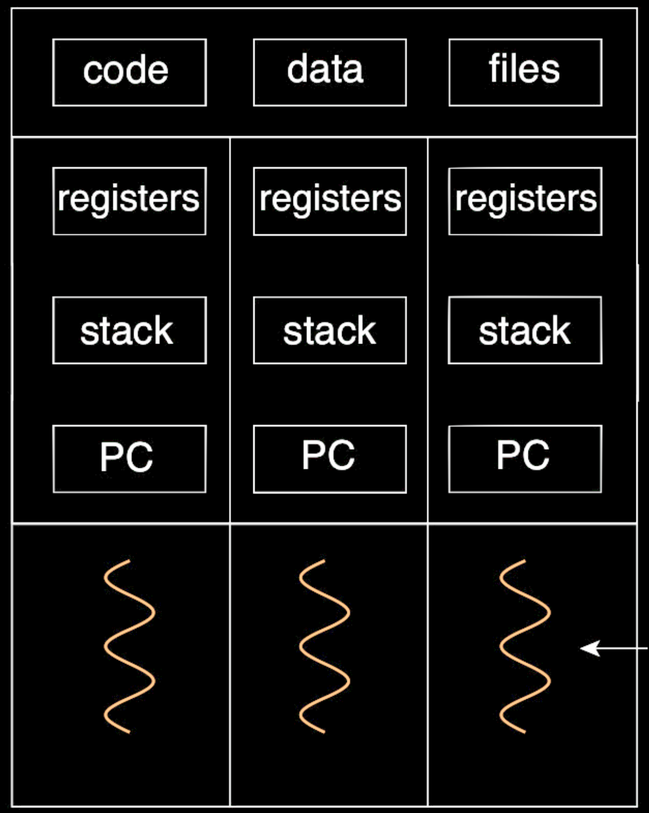
multithreaded process

Thread

About thread for philosophers



single-threaded process



multithreaded process

- 모든 프로세스는 최소 하나의 쓰레드를 가짐.
- 그 쓰레드로 프로세스가 실행되는것!
- 여러 개의 쓰레드를 만들어 동시에 실행 할 수 있음.
- 이 쓰레드들이 철학자가 되어야 함.

Thread

Thread synchronization

- 공유된 메모리를 read만 한다면 문제가 생기지 않음.
- 한 thread라도 변수를 변경하고자 하면 synchronize 문제가 생김
- 적절한 도구 (Mutex)를 사용하여 synchronize를 해주어야 함.

Thread

Unsynchronized threads

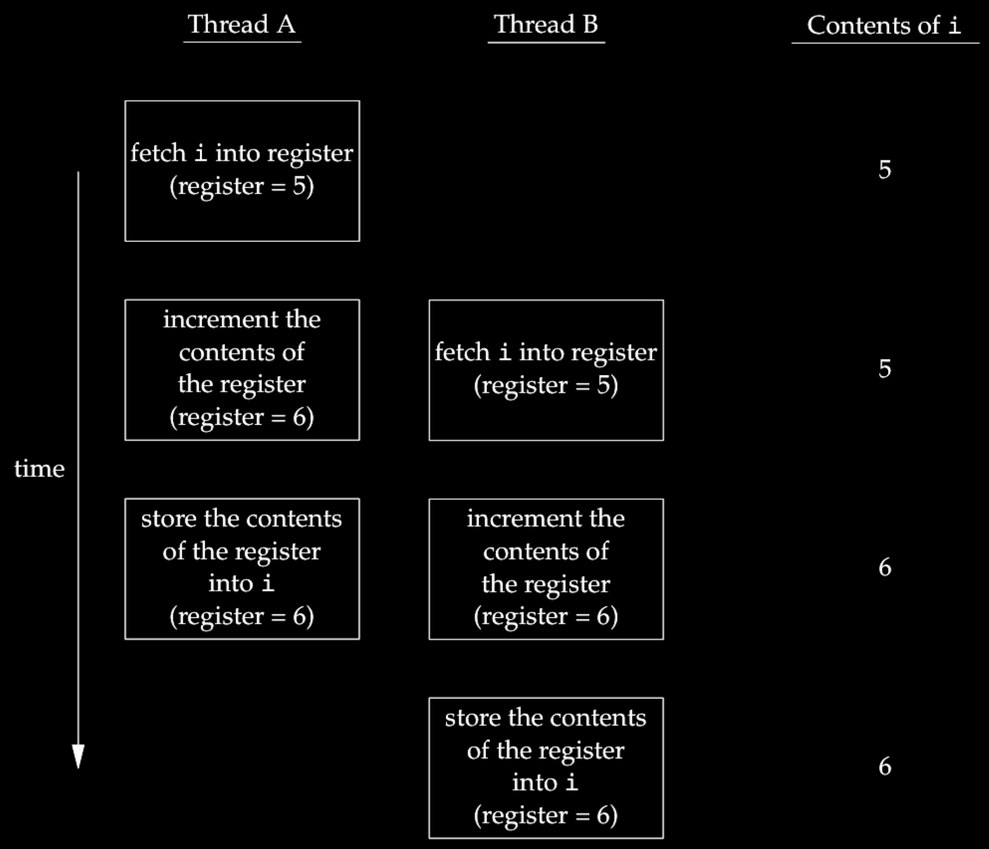


Figure 11.9 Two unsynchronized threads incrementing the same variable

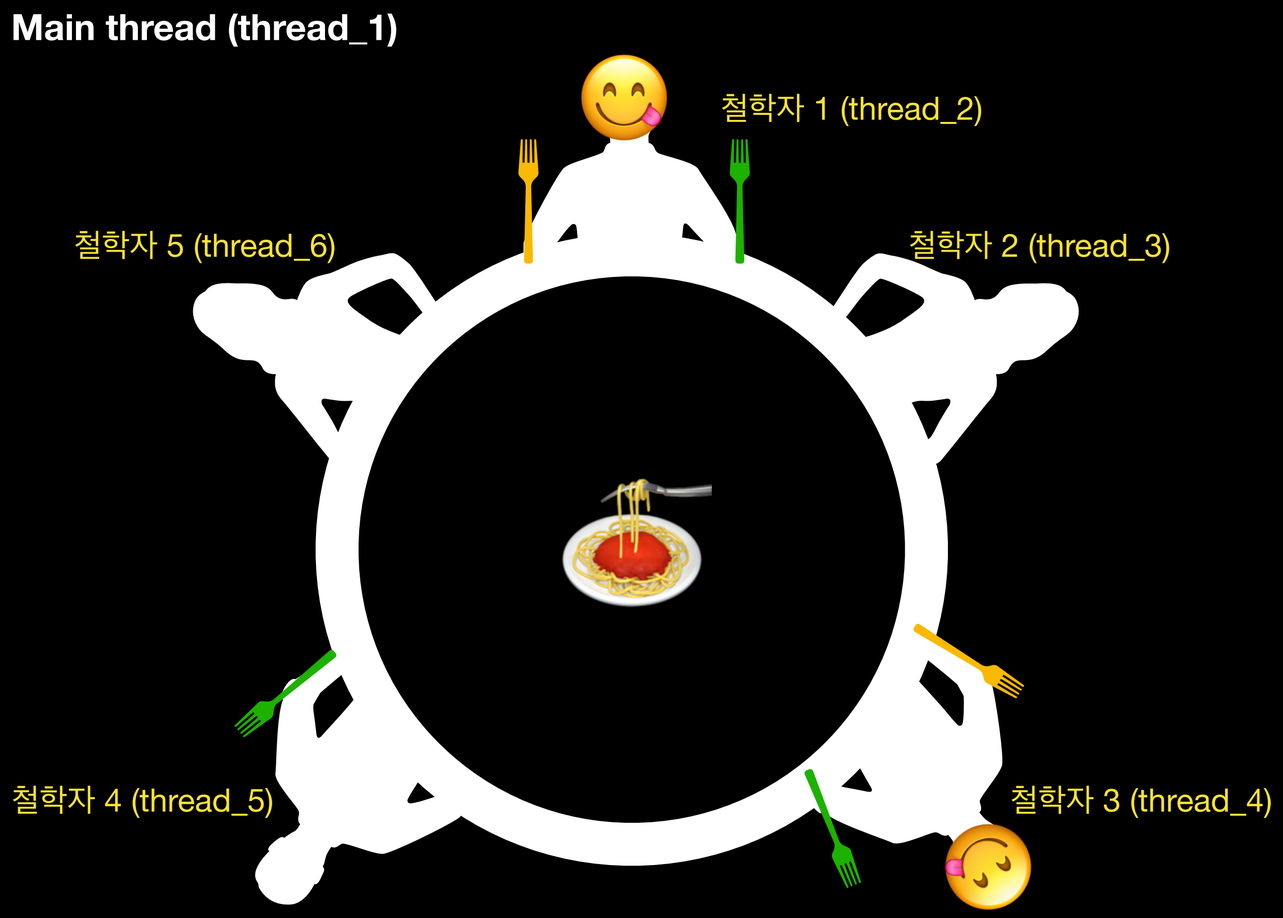
변수를 증가시키는 것은

1. 메모리의 register에서 값을 읽음
2. register의 값을 증가시킴
3. 새로운 값을 메모리에 씀

이러한 과정을 거치며, 이 사이에 다른 thread가 개입하면 의도치 않은 값이 나옴

Thread

Threads in Philosophers



Mutex

Mutex

Mutual exclusive

- synchronize를 위한 tool 중 하나
- 특정 영역을 mutual exclusive 하게 만들어서 하나의 쓰레드만 접근 할 수 있게 만듦.
- 이 특정 영역을 critical section이라 부르며, critical section에 들어가기 전에 lock, 나오면서 unlock 해주는 방식
- deadlock, starving 문제 발생

Mutex

Mutual exclusive

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

- 철학자의 포크가 뮤텍스가 됨.
- critical section은 포크를 들고 밥을 먹는 과정
- 출력하는 과정도 critical section이 되어 mutex를 이용해서 synchronize 해야함.

Mutex

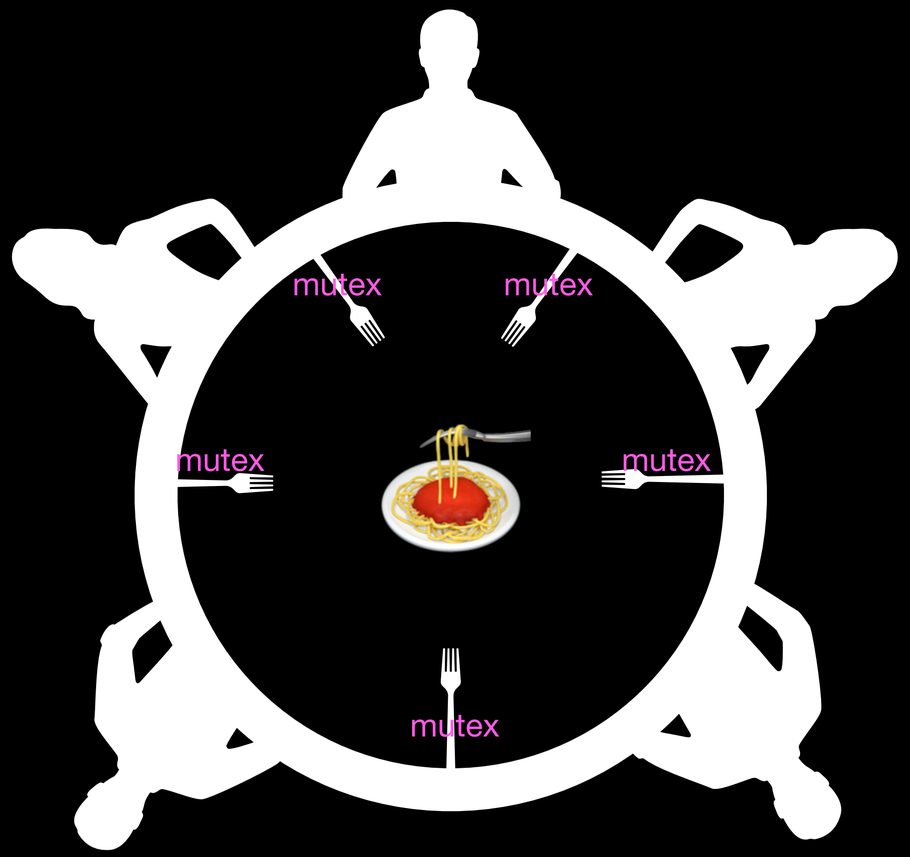
Mutual exclusive

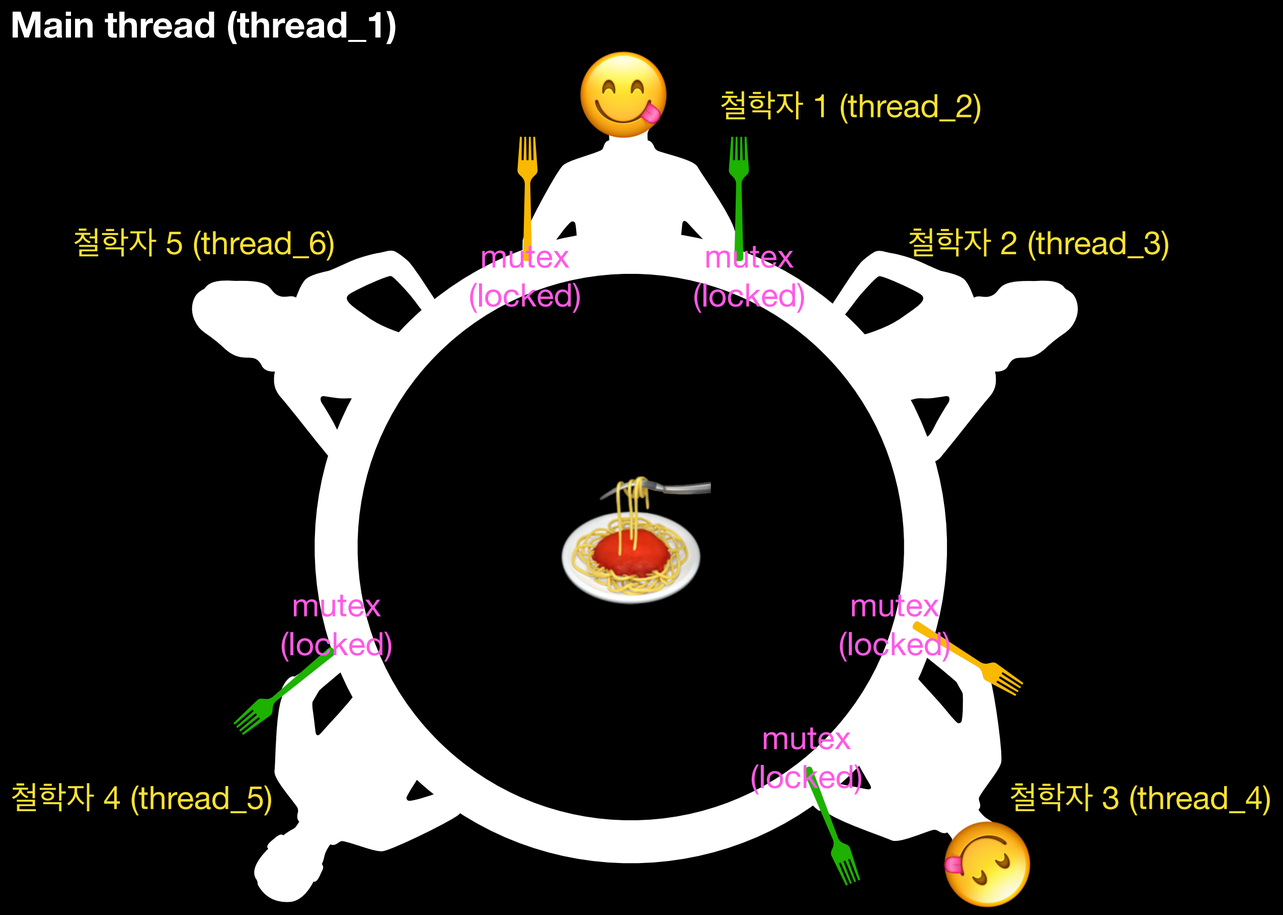
```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

- Mutex 없이 단순 while문으로 가능하지만, 이는 spin-lock 개념으로 자원 낭비가 너무 심함!
- Mutex는 systemcall을 활용하여 철학자 같은 경우에서 더 효율적으로 자원을 사용할 수 있음.

Thread

Threads in Philosophers





Mutex

Inside of Mutex

- 그럼 mutex안의 instructions도 값이 꼬일 수 있지 않을까?
- atomic(non-interruptable) 개념을 사용하여 하드웨어 측면에서 지원
- spin-lock와 확실한 성능 차이를 보임

Mutex

Problem in Mutex : Starving

- Starving : 동시성을 가진 프로그램에서 특정 작업이 지속적으로 자원 할당을 받지 못하고 계속 기다리는 것.
- 철학자에선 die가 이에 해당함.
- 3명의 철학자가 있다면, 1->2->1->2->1… 이런 식으로 밥을 먹어서 3번이 굶어 죽을 수 있음.

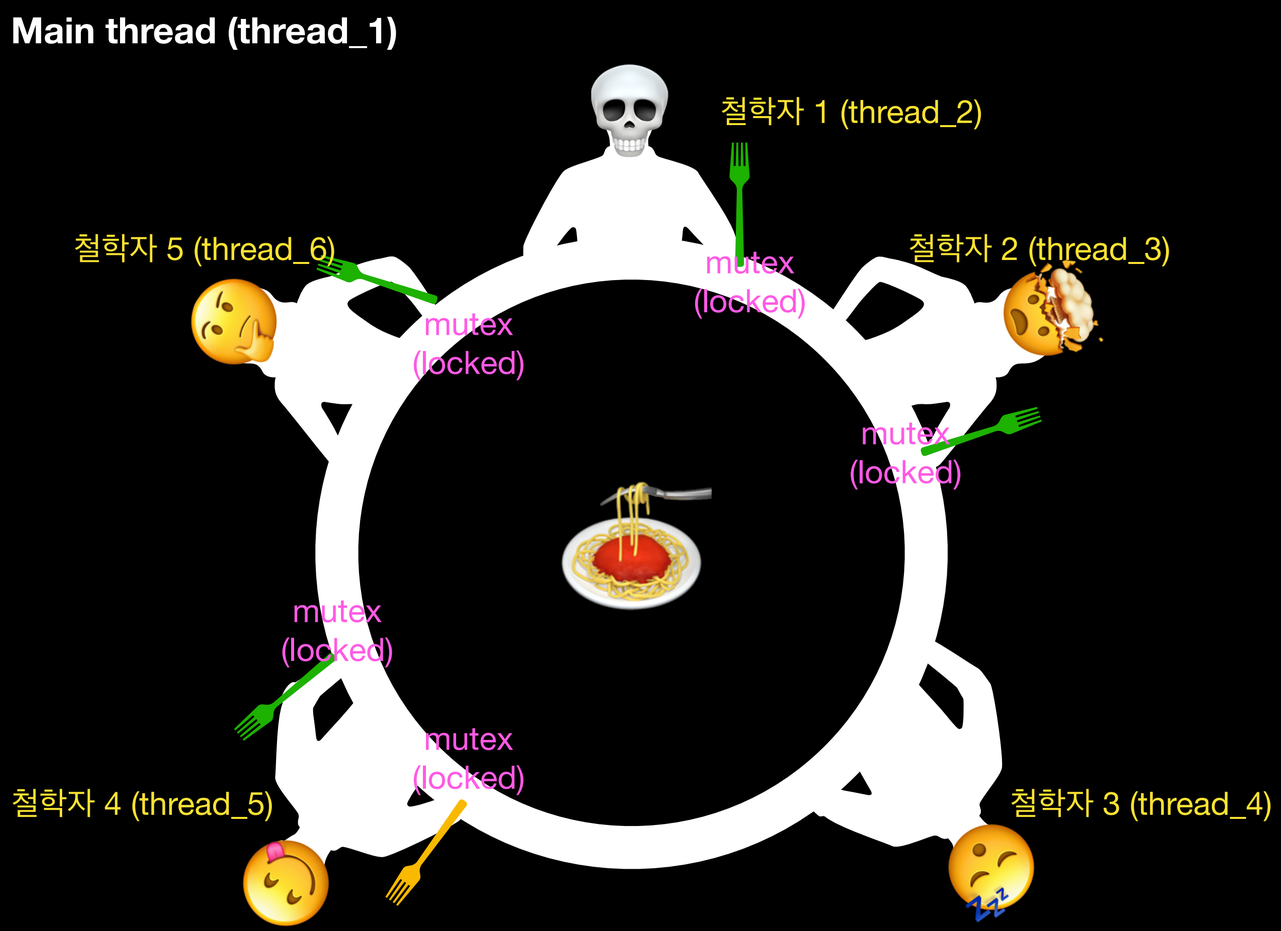
Mutex

Problem in Mutex : Deadlock

- Deadlock : 각각의 멤버가 다음 작업을 수행하기 전 서로를 기다리게 되어서 아무런 작업도 수행하지 못하게 되는것
- 동시성을 갖는 프로그램에서 같은 공유자원에 접근하는걸 막는 과정에서 발생
- Philosophers 과제에선 mutex로 인해 발생
  - 한 철학자(thread)가 포크(mutex)를 든 상태로 끝난 경우
  - 모든 철학자(threads)가 한 손에만 포크(mutex)를 들고 있는 경우

Mutex

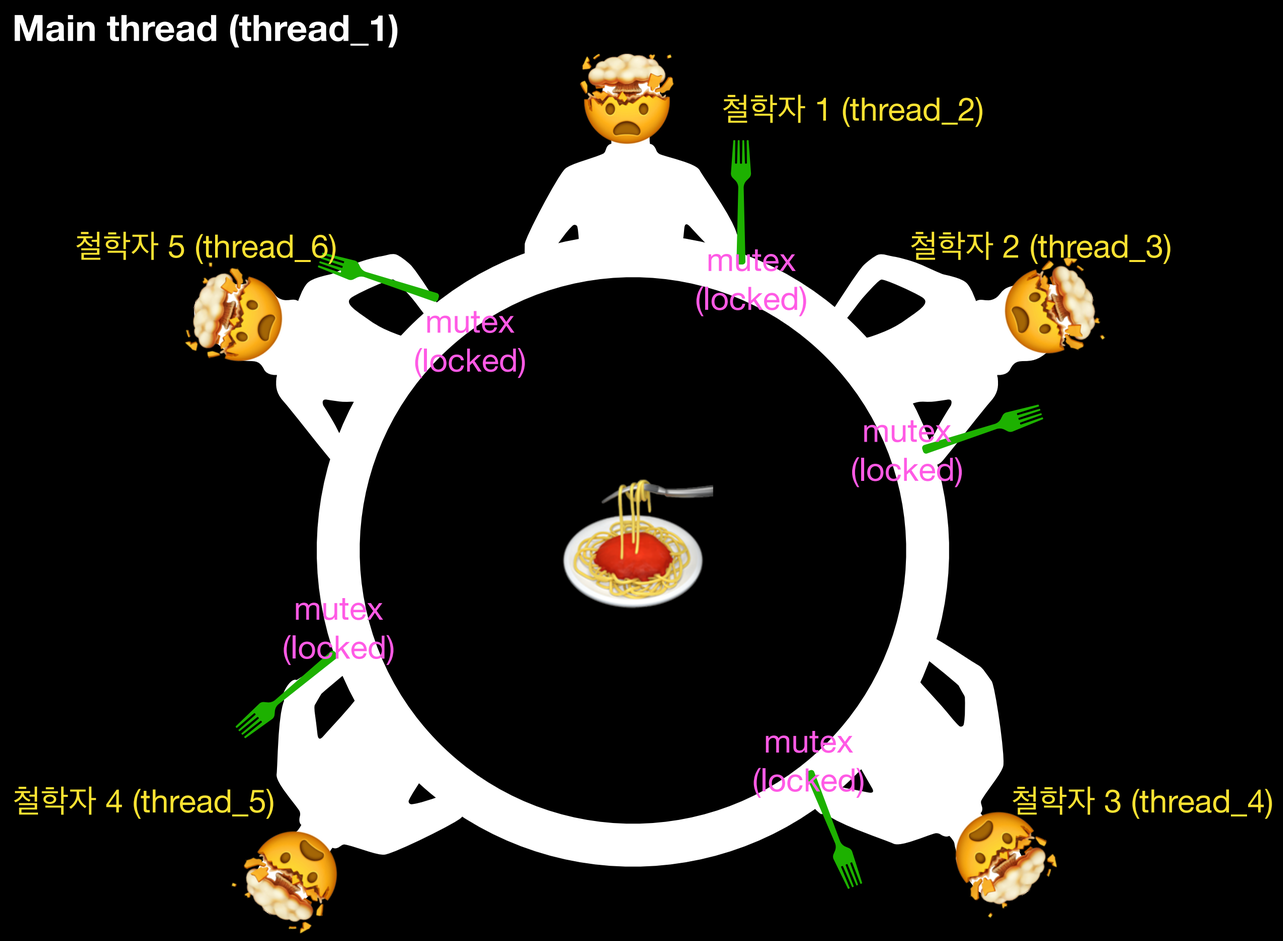
Deadlock in philosophers



- 1번 철학자가 포크를 들고 죽음(lock이 풀리지 않음)
- 2번 철학자는 밥을 먹을 차례인데, 1번 철학자의 포크가 unlock되지 않아 mutex_lock에서 계속 기다림.
- 시뮬레이션이 지속되어서 다른 철학자가 죽어도 끝나지 않음.
- 항상 쓰레드가 종료될 때 mutex를 모두 unlock 해줘야함!!

Mutex

Deadlock in philosophers

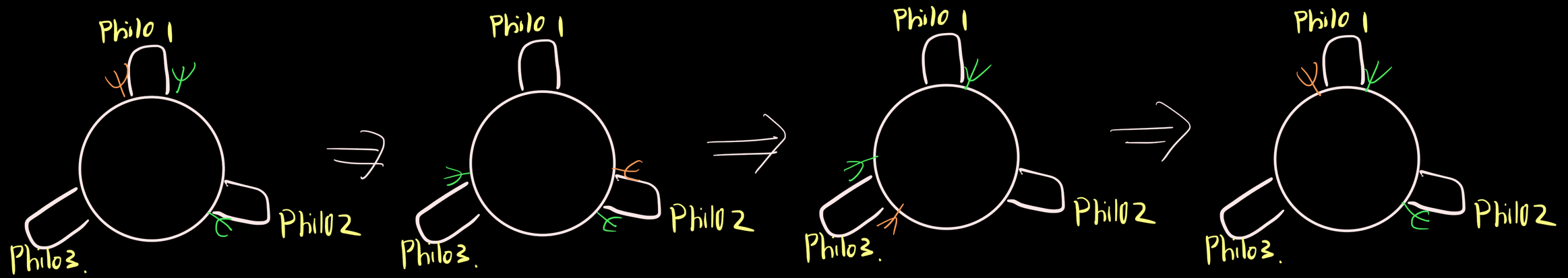


- 모든 철학자가 한 손에만 포크를 들고있음.
- mutex_lock 속에서 다음 포크를 들기 위하여 기다리고 있음.
- 기다림은 영영 끝나지 않고,,, 죽지도 못함.

Mutex

Remedy for deadlock and starvation

- 항상 왼쪽 포크를 먼저 들고, (mutex_lock(left)) 오른쪽 포크를 들으면 (mutex_lock(right)) starving 문제가 어느정도 해결 됨.



Mutex

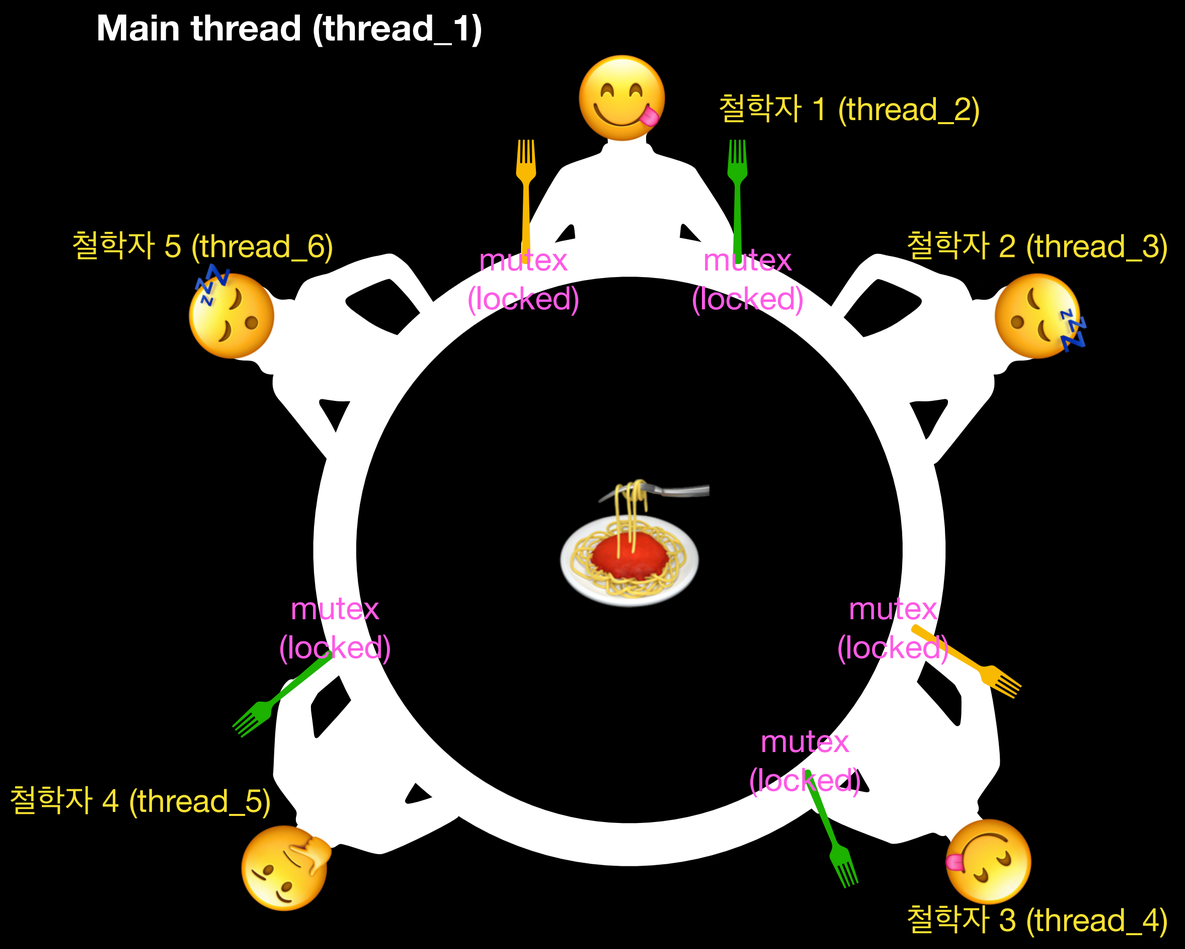
Remedy for deadlock and starvation

- 하지만, 모두가 왼손에 포크를 들게 되는 상황은 발생할 수 있음.
- 짝수번째 철학자를 생성할때 usleep를 이용하여 약간의 시간차를 줘서 해결
  - 상수로 usleelp를 하면 철학자가 매우 많아졌을때, 문제가 생김
  - deadlock의 가능성이 완전히 사라지는건 아님.
  - fork 배열을 이용하여 while문으로 양쪽 다 가용할때 mutex를 걸 수 있지만(spin-lock), 완벽한 해결책도 아니며 성능 저하가 심해짐.

Philosopher simulation

Philosopher simulation

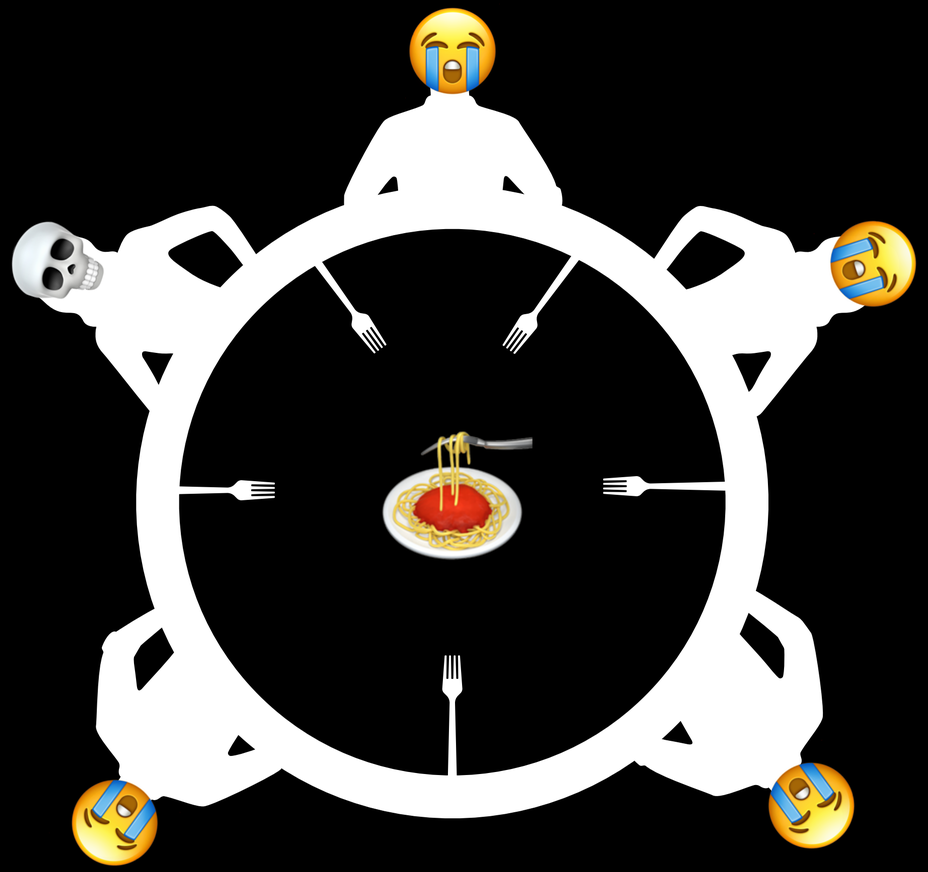
In simulation



- 적절한 시간동안 먹고, 자고, 생각하는 작업을 반복하여야 함.
- 시간을 usleep만 이용하면, 지연시간이 커짐.
  - “usleep == 최소한 이 시간만큼 대기”
- gettimeofday를 while문으로 반복하면, 시스템콜이 반복되어서 지연 시간이 커짐.
- usleep와 gettimeofday를 적절히 활용…!

Philosopher simulation

when philosopher die…



- 한 철학자가 죽으면, 더이상 아무것도 출력하지 않고 프로그램이 종료되어야함.
- 종료 시 mutex는 다 정상적으로 unlock 되어야 하고, 사용한 자원은 모두 반환 되어야함.
- 쓰레드 자원 반환

  - pthread_join vs pthread_detach

Philosopher simulation

when philosopher die…

pthread_join	pthread_detach
쓰레드가 종료 될 때 자원을 반환하게 해주는 함수.	
해당 thread를 기다림	호출한 시점에 해당 thread가 분리됨 (독립적)
thread 생성한 이후에만 호출 가능	thread_create전에 옵션을 줄 수 있고, 생성 후 detach를 시킬 수 있음.

Philosopher simulation

when philosopher die…

- 시뮬레이션 종료 후 쓰레드 자원을 반환하고, 메모리를 해제해주는 작업이 필요.
- detach를 사용하게 되면 main thread가 먼저 종료 될 수 있고, 다른 쓰레드가 사용중인 자원의 메모리를 해제시켜 비정상종료를 일으킬 수 있음.
- join을 이용하여 각 쓰레드가 종료되기를 기다린 후 메모리를 해제해주면 됨!
- 어떤 철학자가 죽었을 때, 모든 철학자들의 Mutex를 잘 unlock 해줘야함.