System Programming & OS 실습
10. Concurrency

이성현, 최민국
Dankook University
{leesh812, mgchoi}@dankook.ac.kr

Index

- Lock
  - Practice 3: Sloppy Counter
  - Practice 4: Accuracy

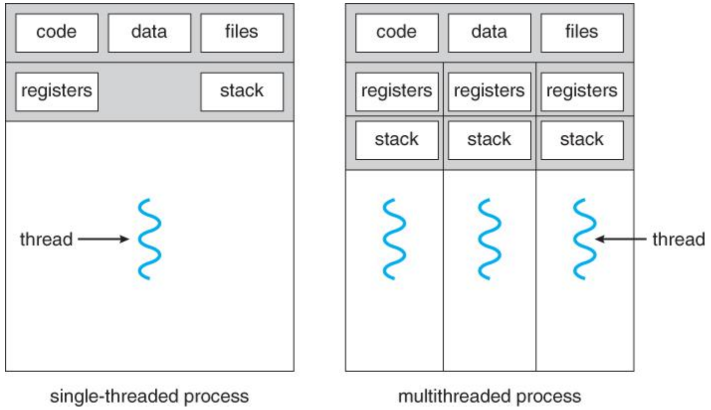
- Semaphore
  - Practice 5: Counting semaphore

- Deadlock
  - Practice 6: Dining philosophers problem
  - Practice 7: Circular Wait
  - Practice 8: No Preemption

Thread

- Multi-threaded program
  - Thread: flow of control
  - Process: one flow of control + resources (address space, files)
  - Multi-threaded program (or process): multiple flow of controls + resources (address space, files)
    - Multiple threads share address space
    - Multiple Processes do not share their address space

- Concurrency
  - Shared data → race condition → may generate wrong results
  - Concurrency: enforce to access shared data in a synchronized way



(Source: A. Silberschatz, “Operating system Concept”)

Thread

- Benefit of Thread
  - Fast creation
  - Parallelism
  - Can overlap processing with waiting
  - Data sharing

- Thread management
  - Several stacks in an address space
  - Scheduling entity

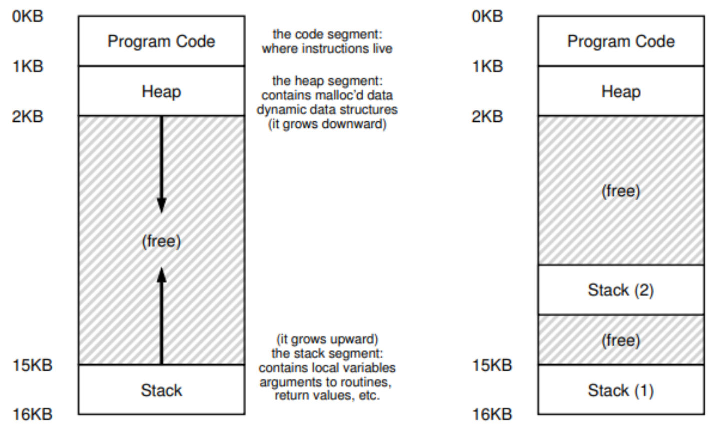
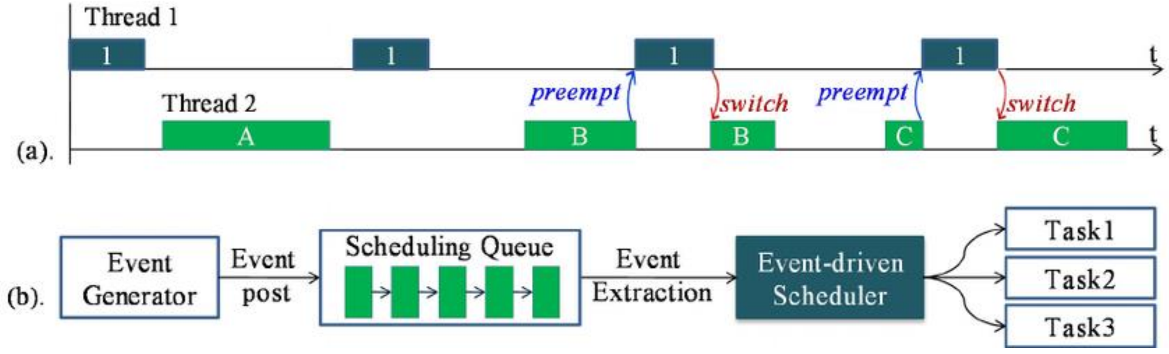


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**



Thread Scheduling

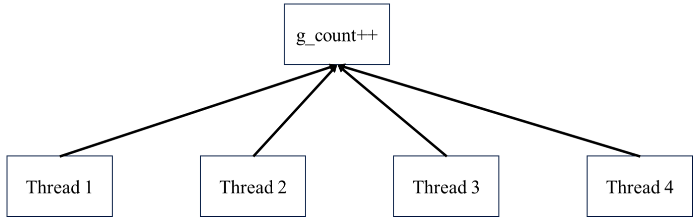
Thread

- #include <pthread.h>
- int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
  void *(*start_routine)(void *), void *restrict arg);
  - similar to fork(), thread exits when the passed function reach the end.
  - arg1) thread structure to interact with this thread,
  - arg2) attribute of the thread such as priority and stack size, in most case it is NULL (use default)
  - arg3) function pointer for start routine
  - arg4) arguments
- int pthread_join(pthread_t thread, void **retval);
  - similar to wait(), for synchronization
  - arg1) thread structure, which is initialized by the thread creation routine
  - arg2) a pointer to the return value (NULL means “don’t care”)

Pratice

- Practice 1

> gcc thread.c -lpthread -o thread.out (컴파일)
> ./thread.out 4 1000 (실행1)
> ./thread.out 4 10000 (실행2)



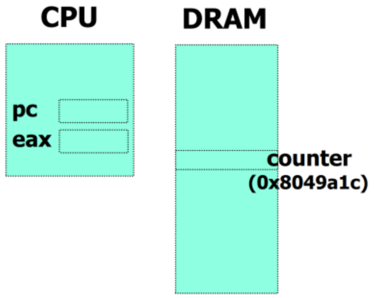

```
embedded@embedded:~/thread_test$ ./thread.out 4 10000
Thread number 0: 10000
Thread number 2: 20000
Thread number 1: 30000
Thread number 3: 40000
Complete
embedded@embedded:~/thread_test$ ./thread.out 4 100000
Thread number 0: 99991
Thread number 2: 218531
Thread number 3: 279583
Thread number 1: 379583
Complete
```

```
for (i = 0; i < 1e7; i++) {
    counter = counter + 1;
}
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

			(after instruction)		
OS	Thread 1	Thread 2	PC	eax	counter
	before critical section		100	0	50
	mov 8049a1c,%eax		105	**50**	50
	add $0x1,%eax		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		mov 8049a1c,%eax	105	**50**	50
		add $0x1,%eax	108	**51**	50
		mov %eax,8049a1c	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	mov %eax,8049a1c		113	51	**51**

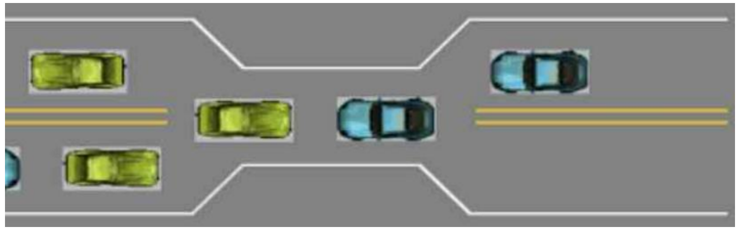
Figure 26.7: **The Problem: Up Close and Personal**



Thread Problem

- Reason
  - Numerous threads access shared data(critical section) at the same time
    → race condition
  - Uncontrolled scheduling
    → Results are different at each execution depending on scheduling order

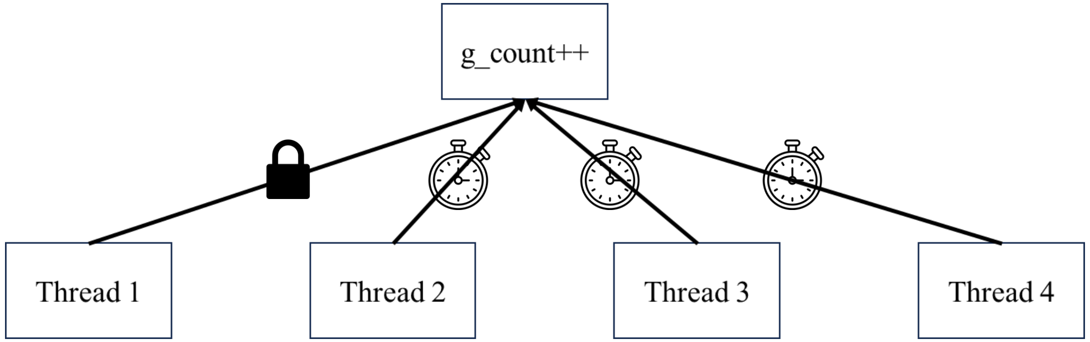
- Solution
  - Controlled scheduling: Do all or nothing (indivisible) → atomicity
  - The code that can result in the race condition → critical section
  - Allow only one thread in the critical section → mutual exclusion

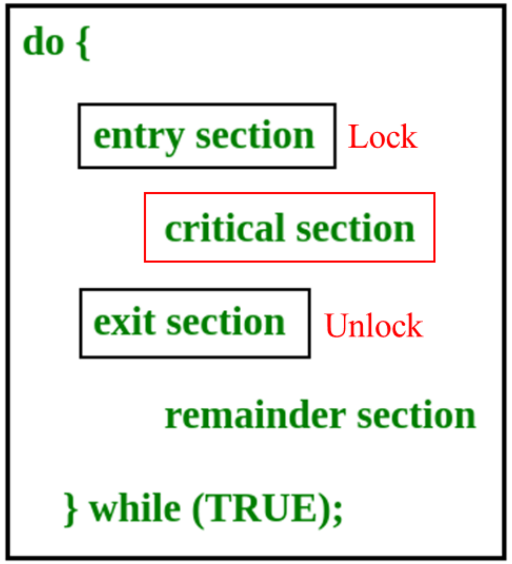


Thread Problem

- Mutual exclusion API (mutex_***)
  - #include <pthread.h>

  - pthread_mutex_t lock;
  - int pthread_mutex_init(pthread_mutex_t *restrict mutex,
  - const pthread_mutexattr_t *restrict attr);
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);



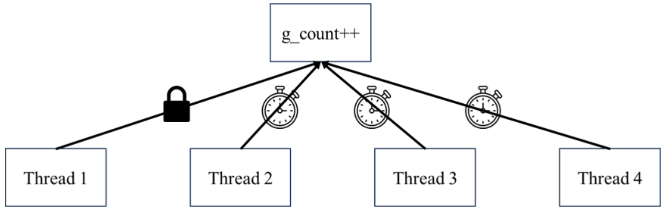


Practice 2

- Practice 2
  - \> gcc -o thread_lock.out thread_lock.c -lpthread (컴파일)
  - \> ./thread.out 4 10000 (실행1)
  - \> ./thread_lock.out 4 10000 (실행2)

```
embedded@embedded:~/thread_test$ ./thread.out 4 100000
Thread number 0: 99991
Thread number 2: 218531
Thread number 3: 279583
Thread number 1: 379583
Complete
```

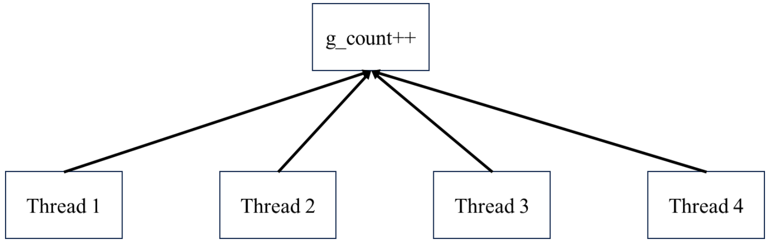
```
embedded@embedded:~/thread_test$ ./thread_lock.out 4 100000
Thread number 1: 235328
Thread number 2: 379740
Thread number 3: 380224
Thread number 0: 400000
Complete
```



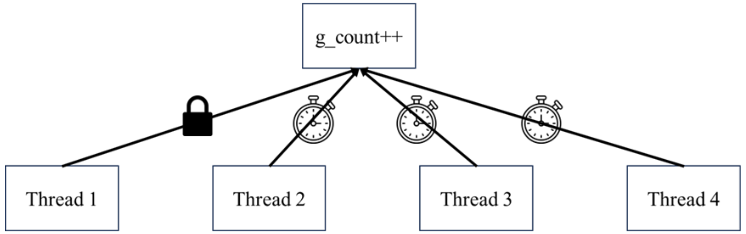

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Practice 3: Sloppy Counter

Practice 1: Multi-threading without lock



Practice 2: Multi-threading with lock



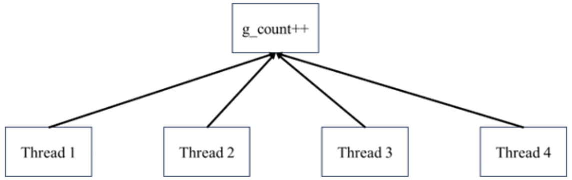
Which one would be faster?

> time ./thread.out 4 10000000

> time ./thread_lock.out 4 10000000

Practice 3: Sloppy Counter

Practice 1: Multi-threading without lock

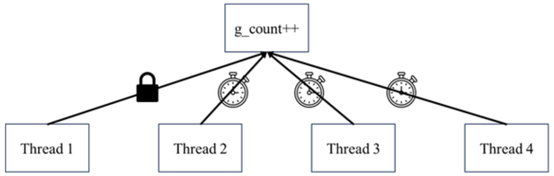


Which one would be faster?

> time ./thread.out 4 10000000

```
mingu@server:~/TABA_OS_2023/thread_practice$ time ./thread.out 4 10000000
Thread number 2: 10509904
Thread number 1: 12437601
Thread number 0: 12486358
Thread number 3: 13135124
Complete

real    0m0.176s
user    0m0.665s
sys     0m0.000s
```

Practice 2: Multi-threading with lock



> time ./thread_lock.out 4 10000000

```
mingu@server:~/TABA_OS_2023/thread_practice$ time ./thread_lock.out 4 10000000
Thread number 2: 35600970
Thread number 0: 38511449
Thread number 1: 39049911
Thread number 3: 40000000
Complete

real    0m2.478s
user    0m3.725s
sys     0m4.959s
```

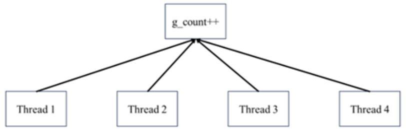
0.176s

14x slower

2.478s

Practice 3: Sloppy Counter

Practice 1: Multi-threading without lock

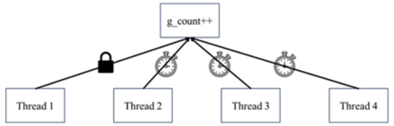


Which one would be faster?

> time ./thread.out 4 10000000

```
mingu@server:~/TABA_OS_2023/thread_practice$ time ./thread.out 4 10000000
Thread number 2: 10509904
Thread number 1: 12437601
Thread number 0: 12486358
Thread number 3: 13135124
Complete

real    0m0.176s
user    0m0.665s
sys     0m0.000s
```

Practice 2: Multi-threading with lock



> time ./thread_lock.out 4 10000000

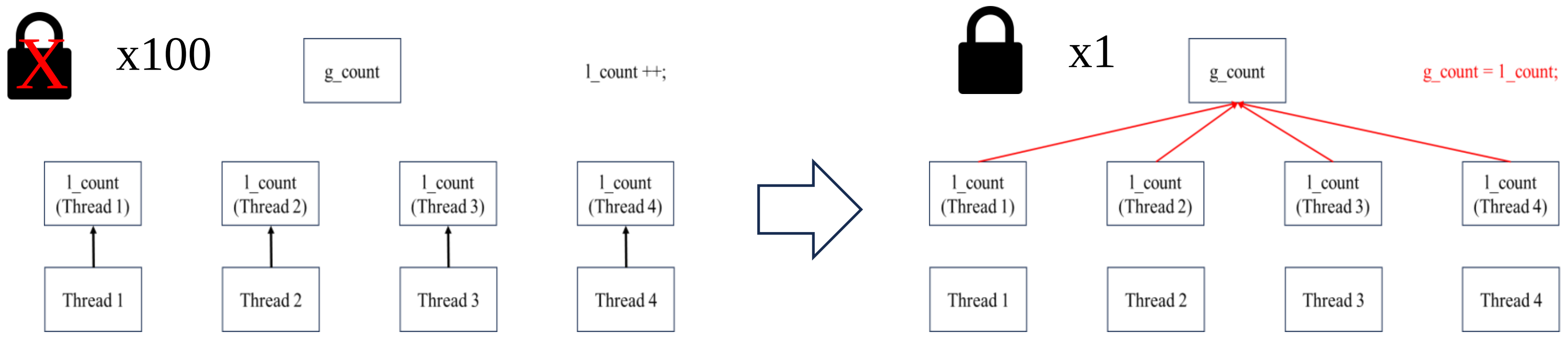
```
mingu@server:~/TABA_OS_2023/thread_practice$ time ./thread_lock.out 4 10000000
Thread number 2: 35600970
Thread number 0: 38511449
Thread number 1: 39049911
Thread number 3: 40000000
Complete

real    0m2.478s
user    0m3.725s
sys     0m4.959s
```

0.176s 14x slower 2.478s

Execution time becomes too long due to **lock contention** …
Do we have to lock **g_count** every time?
Or wouldn't it be okay to lock **g_count** just a few times?

Practice 3: Sloppy Counter

- Sloppy counter
  - a.k.a Scalable counter or Approximate counter
  - Quite higher performance
  - A single global counter + Several local counters
    - Usually, one per CPU core
    - Update local counter → periodically update global counter
    - Less contention → Scalable

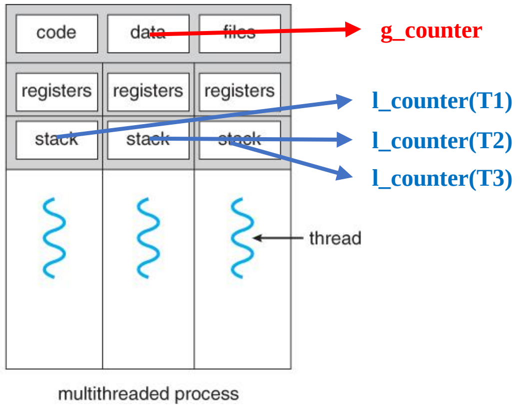


Practice 3: Sloppy Counter

- Sloppy counter
  - A single global counter + Several local counters
    - Update local counter → periodically update global counter

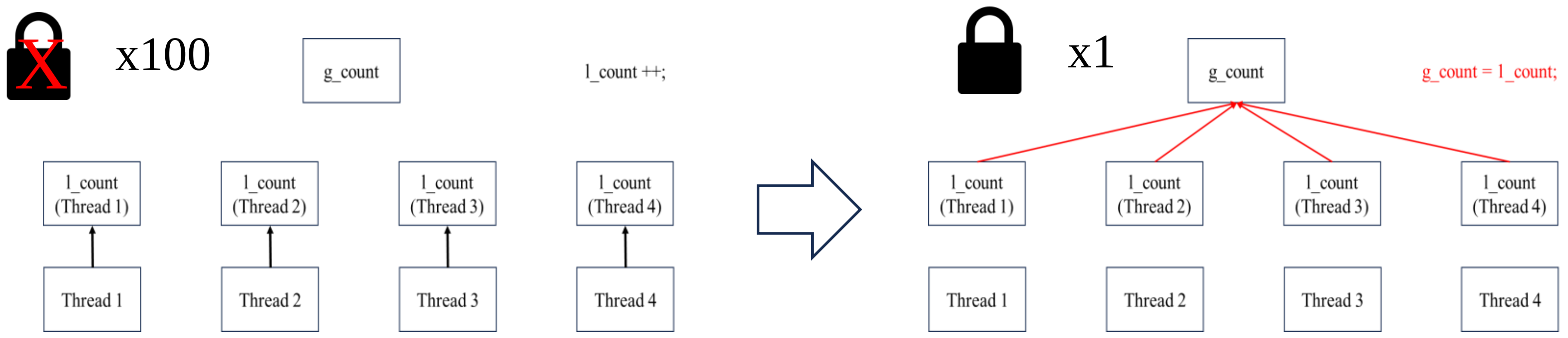
Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.3: **Tracing the Approximate Counters**



Practice 3: Prepare

- Practice 2 command for prepare
  - > cp thread_lock.c thread_sloppy.c (파일 복사)
  - > vim thread_ sloppy.c (코드 작성)



Practice 3: Code


```c
#include <pthread.h>
#include <stdint.h>

int g_count = 0; // counter (critical section)
int g_nthd = 0; // num of threads
int g_worker_loop_cnt = 0;
int sloppy = 0;
pthread_mutex_t lock;

static void *work(void* tno); // thread routine

int main(int argc, char *argv[]){
    pthread_t *thd_arr; // thread array
    int thd_cnt; // thread count

    if (argc < /*fill the blanks */){
        fprintf(stderr, "%s parameter : nthread, worker_loop_cnt, 
sloppy\n", argv[0]);
        exit(-1);
    }

    // get num of threads and worker loop count
    g_nthd = atoi(argv[1]);
    g_worker_loop_cnt = atoi(argv[2]);
    sloppy = /*fill the blanks */;

    // ...
```

```c
static void *work(void* cnt){
    int thd_cnt = (int)(intptr_t)cnt;
    int i, j;
    int l_count = 0;

    for(i = 0; i < /*fill the blanks */; i++){
        for(j = 0; j < /*fill the blanks */ ; j++){
            l_count++;
        }
        pthread_mutex_lock(&lock);
        /*fill the blanks */
        /*fill the blanks */

        /*fill the blanks */
    }

    printf("Thread number %d: %d \n", thd_cnt, g_count);
    return NULL;
}
```

Practice 3: Run

- Practice 3 command 2
  - gcc -o thread_sloppy.out thread_sloppy.c -lpthread (컴파일)

> time ./thread_sloppy.out 1 100000 1000
> time ./thread_sloppy.out 2 100000 1000
> time ./thread_sloppy.out 3 100000 1000
> time ./thread_sloppy.out 4 100000 1000

> time ./thread_lock.out 1 100000
> time ./thread_lock.out 2 100000
> time ./thread_lock.out 3 100000
> time ./thread_lock.out 4 100000

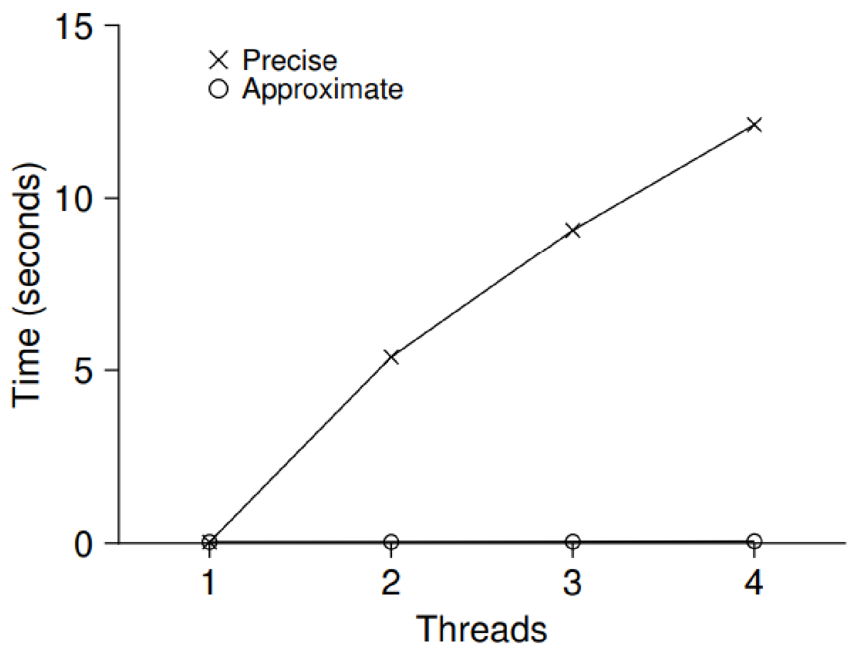


Figure 29.5: **Performance of Traditional vs. Approximate Counters**

Practice 3: Run

- Practice 3 command 3
  - gcc -o thread_sloppy.out thread_sloppy.c -lpthread (컴파일)

```
> time ./thread_sloppy.out 4 100000 1
> time ./thread_sloppy.out 4 100000 4
> time ./thread_sloppy.out 4 100000 16
> time ./thread_sloppy.out 4 100000 64
> time ./thread_sloppy.out 4 100000 256
> time ./thread_sloppy.out 4 100000 1024
```

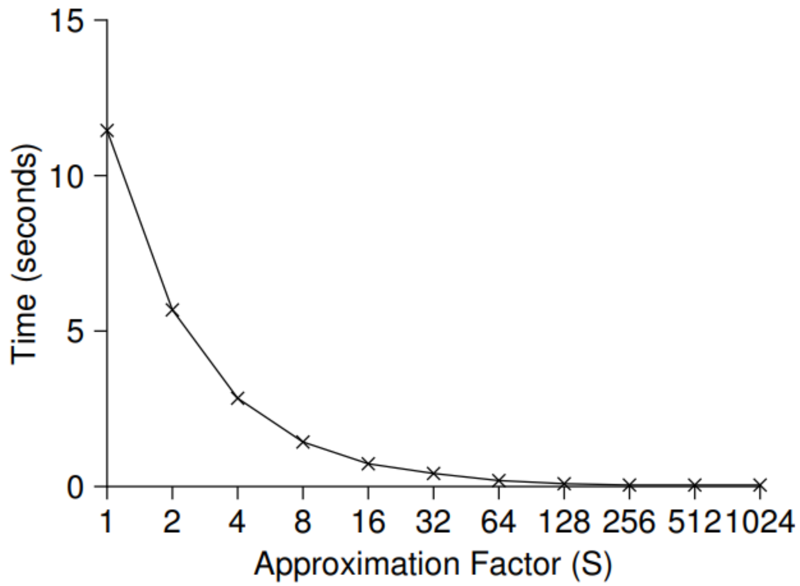


Figure 29.6: **Scaling Approximate Counters**

Practice 4: Accuracy

- Bigger the S, shorter the time
  - Then large S(update frequency) is sliver bullet?

  - Are there any other considerations?
    - Accuracy of counter during increment

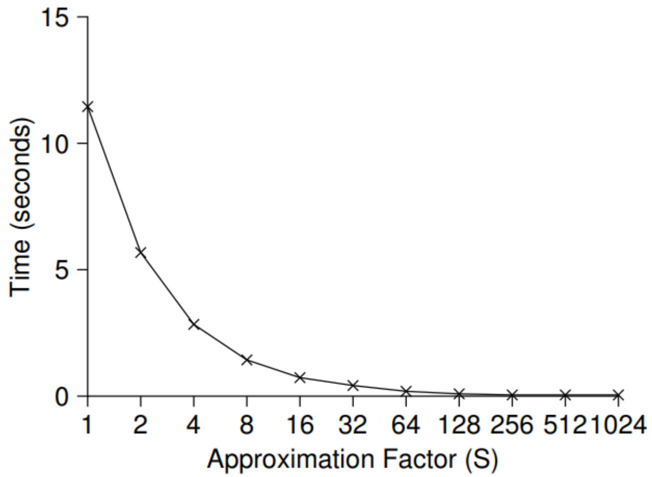


Figure 29.6: **Scaling Approximate Counters**

Practice 4: Prepare

- Practice 4 command for prepare

  > cp thread_sloppy.c thread_ accuracy.c   (파일 복사)

  > vim thread_ accuracy.c   (코드 작성)

Practice 4: Code


```
int main(int argc, char *argv[]){
    // ...
    for(thd_cnt=0; thd_cnt < g_nthd; thd_cnt++){
        // create thread
        assert(pthread_create(&thd_arr[thd_cnt], NULL,
               work, (void*) (intptr_t) thd_cnt) == 0);
        pthread_mutex_lock(&lock);
        printf("After create %d: %d \n", thd_cnt, g_count);
        pthread_mutex_unlock(&lock);
    }

    for(thd_cnt=0; thd_cnt < g_nthd; thd_cnt++){
        // join thread
        pthread_mutex_lock(&lock);
        printf("Before join %d: %d \n", thd_cnt, g_count);
        pthread_mutex_unlock(&lock);
        assert(pthread_join(thd_arr[thd_cnt], NULL) == 0);
    }
    printf("Complete\n");
}
```

Practice 4: Run

- Practice 4 command 2

  > gcc -o thread_accuracy.out thread_ accuracy.c -lpthread (컴파일)

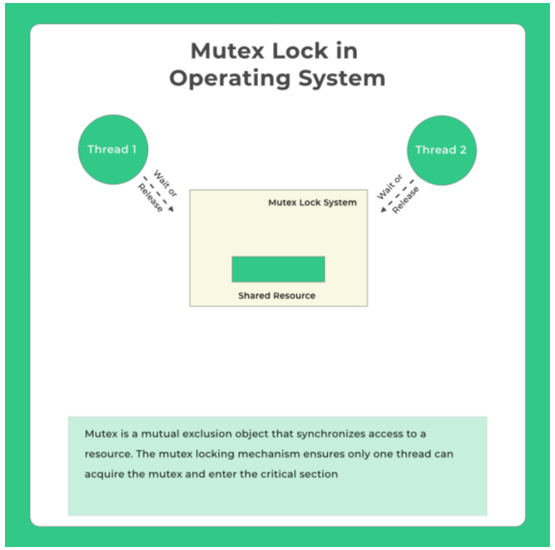
  > ./thread_accuracy.out 2 100000 1
  > ./thread_ accuracy.out 2 100000 10
  > ./thread_ accuracy.out 2 100000 100
  > ./thread_ accuracy.out 2 100000 1000

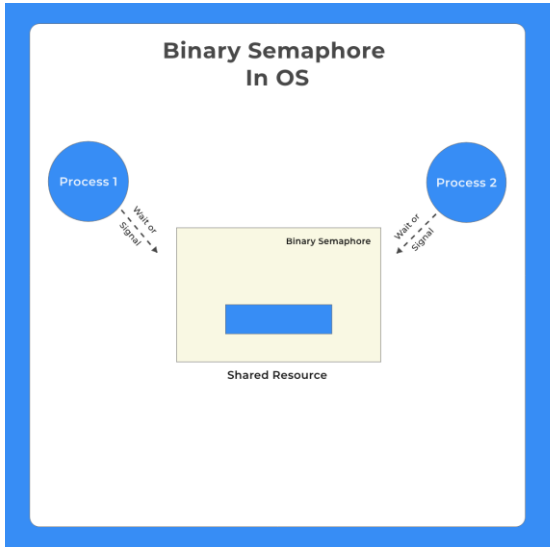
```
mingu@server:~/TABA_OS_2023/thread_practice$ ./thread_accuracy.out 2 100000 1
After create 0: 0
After create 1: 906
Before join 0: 1025
Thread number 1: 160644
Thread number 0: 200000
Before join 1: 200000
Complete
mingu@server:~/TABA_OS_2023/thread_practice$ ./thread_accuracy.out 2 100000 10
After create 0: 0
After create 1: 2840
Before join 0: 3810
Thread number 0: 157120
Before join 1: 186960
Thread number 1: 200000
Complete
```

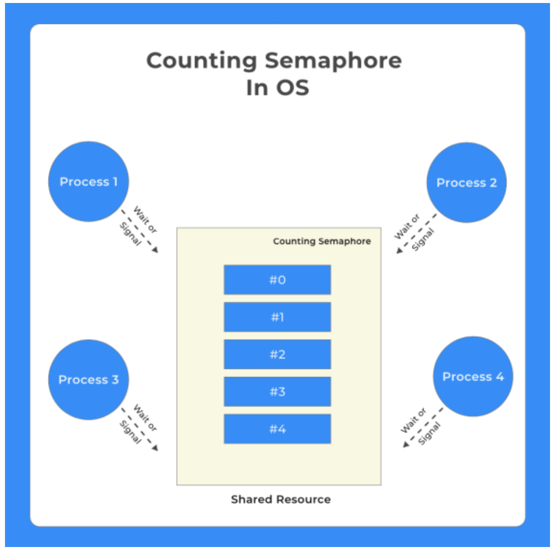
```
mingu@server:~/TABA_OS_2023/thread_practice$ ./thread_accuracy.out 2 100000 100
After create 0: 0
After create 1: 7200
Before join 0: 8200
Thread number 0: 184200
Before join 1: 196100
Thread number 1: 200000
Complete
mingu@server:~/TABA_OS_2023/thread_practice$ ./thread_accuracy.out 2 100000 1000
After create 0: 0
After create 1: 6000
Before join 0: 7000
Thread number 0: 187000
Before join 1: 199000
Thread number 1: 200000
Complete
```

Semaphore

- Semaphore
  - Well-known structure for concurrency control
    - Can be used as both a lock and a condition variable
    - Binary semaphore, Counting semaphore
    - Can be employed by various concurrency problems including
      1) producer/consumer, 2) reader/writer and 3) dining philosophers







Semaphore

- Semaphore definition
  - An object with an integer value manipulated by three routines
    - sem_init(semaphore, p_shared, initial_value)
    - sem_wait(): also called as P(), down() …

      Decrease the value of the semaphore (S). Then, either return right away (when S >= 0) or cause the caller to suspend execution waiting for a subsequent post (when S < 0)
    - sem_post(): also called as V(), up(), sem_signal() …

      Increment the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up
    - Others: sem_trywait(), sem_timewait(), sem_destroy()

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Figure 31.1: **Initializing A Semaphore**

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

Figure 31.2: **Semaphore: Definitions Of Wait And Post**

Semaphore

- Using a semaphore as a lock
  - Binary semaphore

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Figure 31.3: **A Binary Semaphore (That Is, A Lock)**

  - Counting semaphore

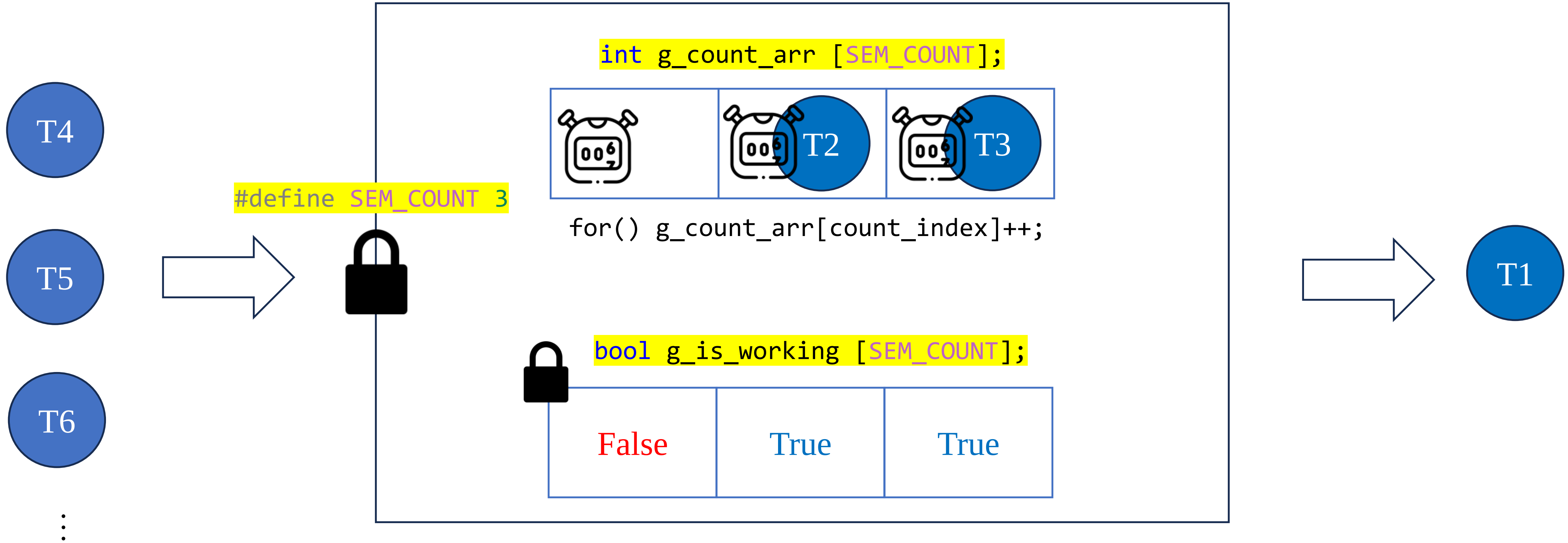
Semaphore

- Using a semaphore as a lock
  - Running example

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call sem_wait()	Run		Ready
0	sem_wait() returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call sem_wait()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0)→sleep	Sleep
-1		Run	*Switch→T0*	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post() returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	sem_wait() returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post()	Run
1		Ready	sem_post() returns	Run

Figure 31.5: **Thread Trace: Two Threads Using A Semaphore**

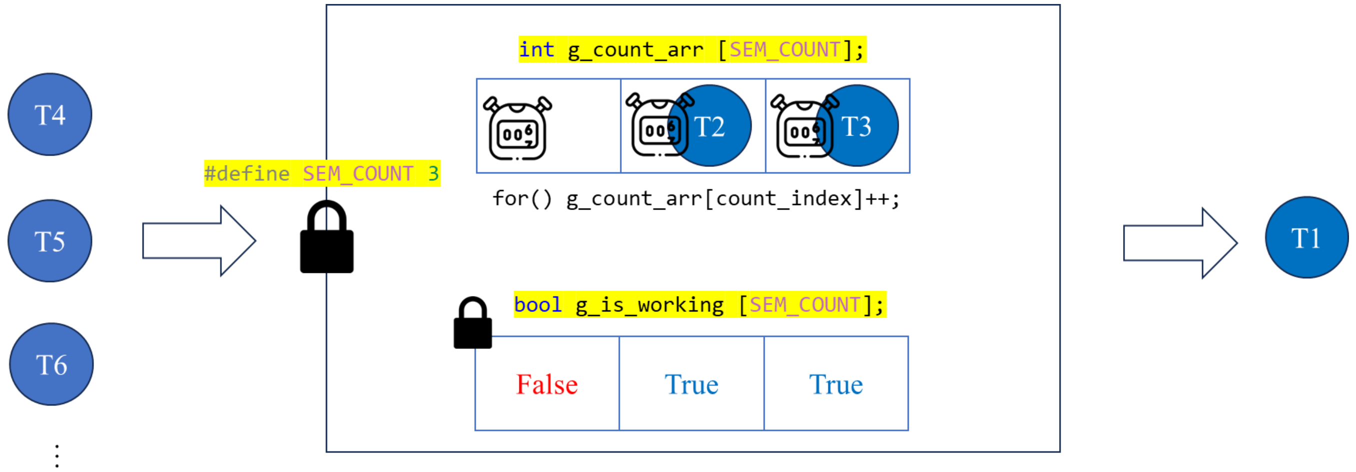
Practice 5



int g_count_arr [SEM_COUNT];
T2
T3
#define SEM_COUNT 3
for() g_count_arr[count_index]++;
T4
T5
T6
T1
bool g_is_working [SEM_COUNT];
False
True
True

Practice 5: Prepare

- Practice 5 command for prepare
  - > cp thread.c semaphore.c (파일 복사)
  - > vim semaphore.c (코드 작성)



Practice 5: Code


```c
#define SEM_COUNT 3

int g_count = 0; // counter (critical section)
int g_count_arr [SEM_COUNT]; // counter array (critical section)
bool g_is_working [SEM_COUNT];  // check if g_count_arr is working

sem_t b_semaphore; // semaphore for alloc counter index
sem_t c_semaphore; // semaphore for checking # of worker

int main(int argc, char *argv[]){
    // ...
    thd_arr = malloc(sizeof(pthread_t) * g_nthd);

    sem_init(&b_semaphore, /*fill the blanks */); // init sem
    sem_init(&c_semaphore, /*fill the blanks */); // init sem

    // ...

    for(thd_cnt=0; thd_cnt<g_nthd; thd_cnt++) {
        assert(pthread_join(thd_arr[thd_cnt], NULL) == 0);
    }

    // check result of counter array
    printf("Counter array: ");
    for (int i = 0; i < SEM_COUNT; i++){
        printf("%d\t", g_count_arr[i]);
    }
    printf("\n");
}
```

```c
static void *work (void* cnt){
    int thd_cnt = (int)(intptr_t)cnt;
    int count_index = -1; // index for count arr

    sem_wait(/*fill the blanks */);

    sem_wait(/*fill the blanks */);
    for (int i=0; i < SEM_COUNT; i++){
        if(g_is_working[i] == false){ // check if counter is not working
            g_is_working[i] = /*fill the blanks */; // this counter will be used
            count_index = /*fill the blanks */;     // remember counter index
            break;
        }
    }
    sem_post(/*fill the blanks */);

    if(count_index == -1) {
        fprintf(stderr, "Thread number %d: count_index < 0", thd_cnt);
        exit(-1);
    }

    for(int i = 0; i < g_worker_loop_cnt; i++)
        g_count_arr[count_index]++;

    sem_wait(/*fill the blanks */);
    g_is_working[count_index] = /*fill the blanks */; // free counter
    sem_post(/*fill the blanks */);

    sem_post(/*fill the blanks */);

    return NULL;
}
```

Practice 5: Run

- Command

  > gcc -o semaphore.out semaphore.c -lpthread   (컴파일)

  > ./semaphore.out 10 10
  > ./semaphore.out 10 100000

```
mingu@server:~/TABA_OS_2023/thread_practice$ gcc -o semaphore.out semaphore.c -lpthread
mingu@server:~/TABA_OS_2023/thread_practice$ ./semaphore.out 10 10
Count array: 100        0       0
mingu@server:~/TABA_OS_2023/thread_practice$ ./semaphore.out 10 100000
Count array: 400000     300000  300000
mingu@server:~/TABA_OS_2023/thread_practice$ 
```

Deadlock

- Deadlock
  - A situation where two or more threads wait for events that never occur

```
Thread 1:                     Thread 2:
pthread_mutex_lock(L1);       pthread_mutex_lock(L2);
pthread_mutex_lock(L2);       pthread_mutex_lock(L1);
```

  - E.g.) When a thread (say Thread 1) is holding a lock (L1) and waiting for another one (L2); unfortunately, the thread (Thread 2) that holds lock L2 is waiting for L1 to be released.

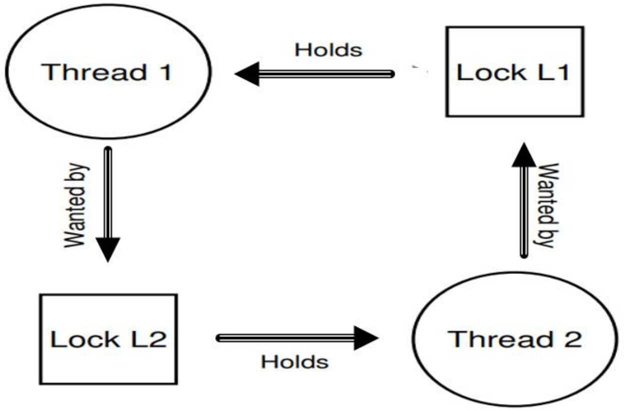


Figure 32.2: **The Deadlock Dependency Graph**

Deadlock

- Deadlock: 4 Conditions
  - Mutual exclusion
  - Hold-and-Wait
  - No preemption for resource
  - Circular wait

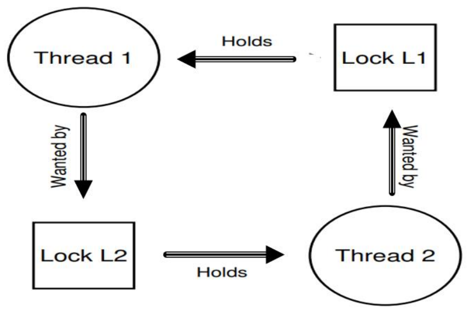
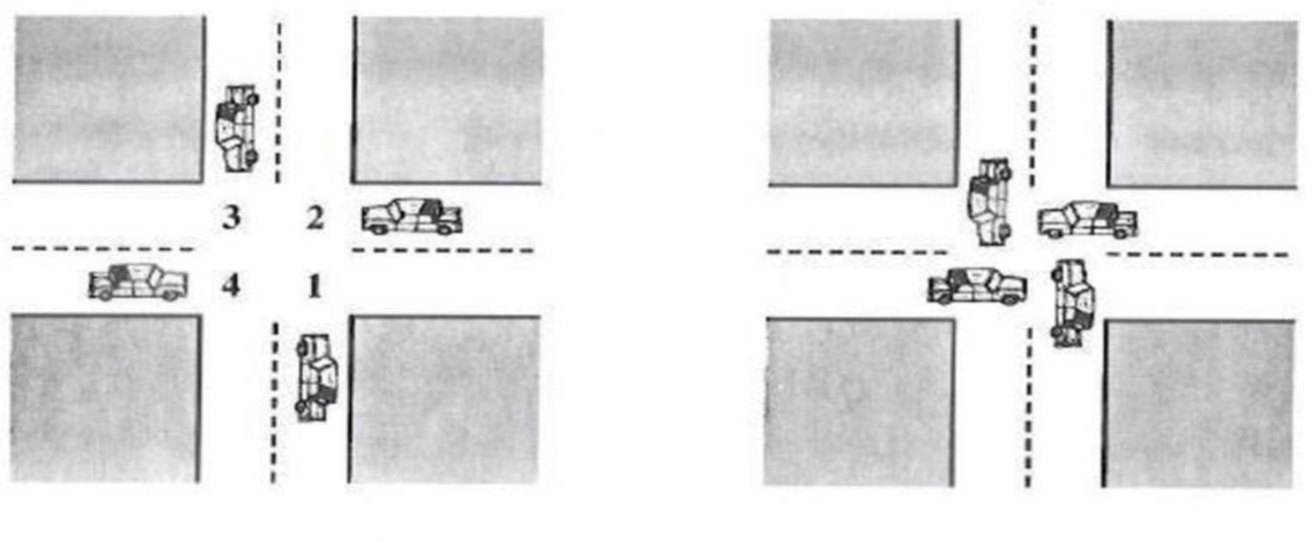


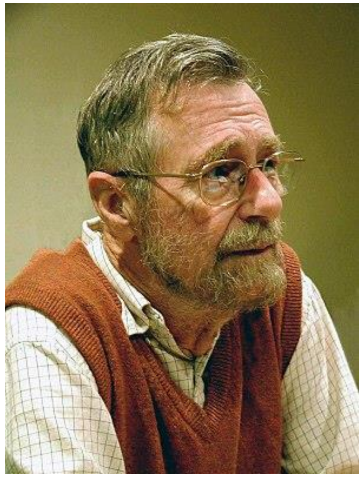
Figure 32.2: The Deadlock Dependency Graph



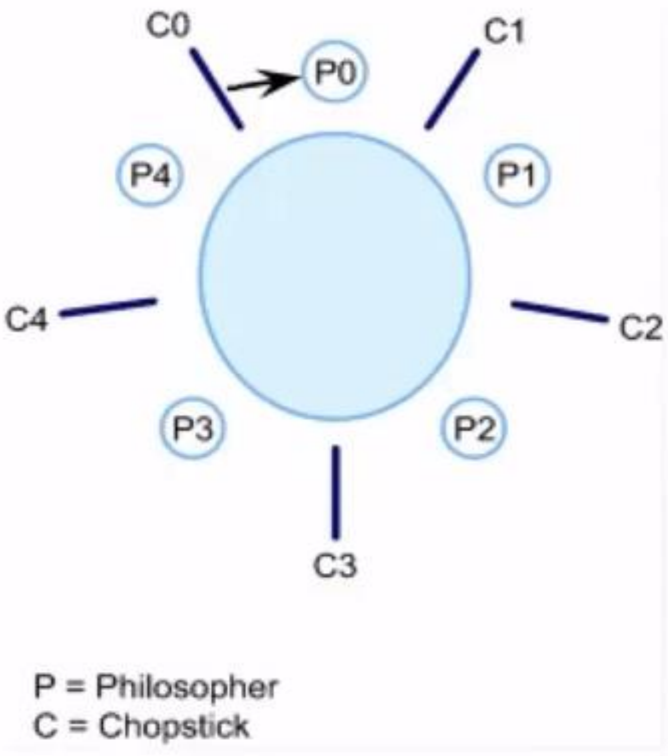
(a) Deadlock Possible

(b) Deadlock

Dining philosophers problem



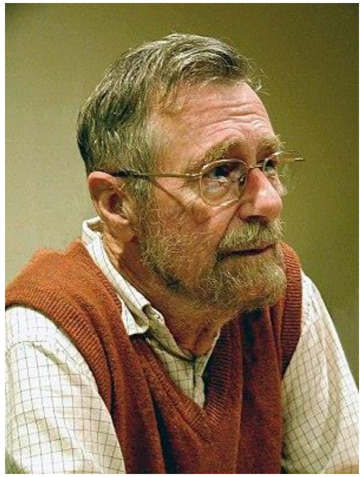
Edsger Wybe Dijkstra



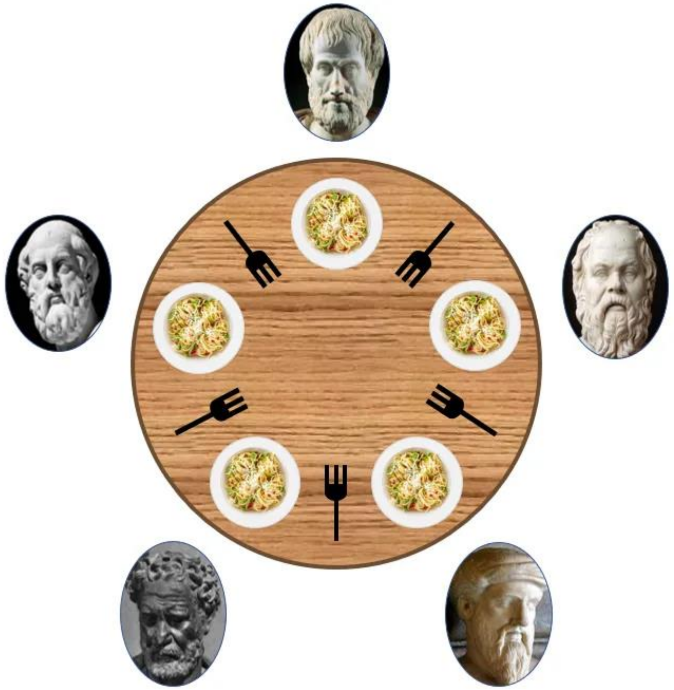
Problem Definition

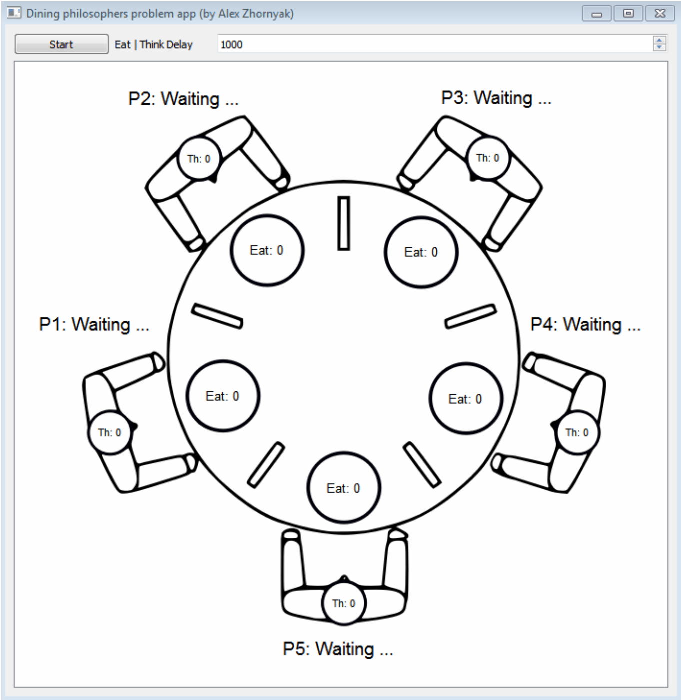
- There are five “philosophers” sitting around a table.
- Between each pair of philosophers is a single fork (thus, five total)
- The philosophers each have times for thinking or for eating
- In order to eat, a philosopher needs two forks, both the one on their left and the one on their right → shared resource → concurrency

Dining philosophers problem



Edsger Wybe Dijkstra





P6: Dining philosophers problem

- Prepare

```
> cd ~
> mkdir philosophers
> cd philosophers
> vim philosophers.c
```

P6: Dining philosophers problem

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define NUM_PHILS 5
pthread_mutex_t forks[NUM_PHILS];

void init();
int leftOf(int i);
int rightOf(int i);
void* philosopher(void* param);
void think(int id);
void eat(int id);
void get_forks(int id);
void put_forks(int id);

int main(){
    pthread_t *thd_arr; // thread array
    thd_arr = malloc(sizeof(pthread_t) * NUM_PHILS);

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_mutex_init(/*fill blanks*/, NULL);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_create(/*fill blanks*/, NULL,
                       philosopher, (void*) &i);
        usleep((1 + rand() % 50) * 10000);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_join(/*fill blanks*/, NULL);
    }
    return 0;
}
```

```c
int leftOf(int i){
    return (i) % NUM_PHILS;
}

int rightOf(int i){
    return (i + 1) % NUM_PHILS;
}

void* philosopher(void* param){
    int id = *((int *) param);
    while(1){
        think(id);
        get_forks(id);
        eat(id);
        put_forks(id);
    }
}

void think(int id){
    printf("%d: Now, I'm thinking...\n", id);
}

void eat(int id){
    printf("%d: Now, I'm eating...\n", id);
}
```

```c
void get_forks(int id){
    pthread_mutex_lock(/*fill blanks*/);
    pthread_mutex_lock(/*fill blanks*/);
}

void put_forks(int id){
    pthread_mutex_unlock(/*fill blanks*/);
    pthread_mutex_unlock(/*fill blanks*/);
}
```

P6: Dining philosophers problem

- Run

  > gcc –o philosophers.out philosophers.c -lpthread

  > ./philosophers.out

```
mingu@server:~/TABA_OS_2023/philosophers$ gcc -o philosophers philosophers.c -lpthread
mingu@server:~/TABA_OS_2023/philosophers$ ./philosophers
1: Now, I'm thinking...
3: Now, I'm thinking...
0: Now, I'm thinking...
2: Now, I'm eating...
2: Now, I'm thinking...
2: Now, I'm eating...
2: Now, I'm thinking...
1: Now, I'm eating...
1: Now, I'm thinking...
0: Now, I'm eating...
3: Now, I'm eating...
3: Now, I'm thinking...
4: Now, I'm thinking...
4: Now, I'm eating...
0: Now, I'm thinking...
2: Now, I'm eating...
4: Now, I'm thinking...
1: Now, I'm eating...
2: Now, I'm thinking...
0: Now, I'm eating...
1: Now, I'm thinking...
0: Now, I'm thinking...
```

Deadlock

Deadlock

- How to handle Deadlock: three strategies
  - 1. Deadlock Prevention
  - 2. Deadlock Avoidance via Scheduling
  - 3. Deadlock Detection and Recovery

Deadlock

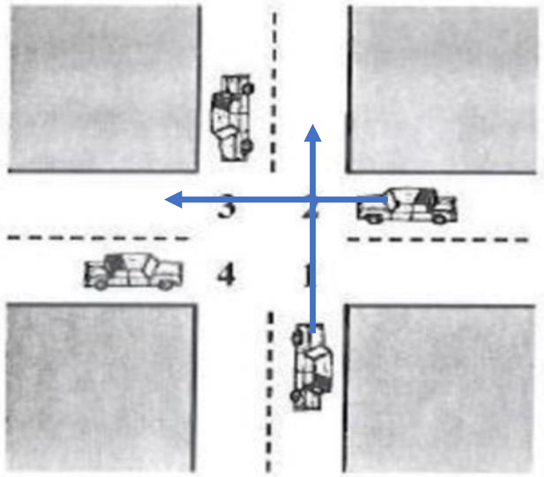
- Deadlock prevention
  - This strategy seeks to prevent one of the 4 Deadlock conditions
    - 1. Hold-and-wait
      - Acquire all locks at once, atomically

```
pthread_mutex_lock(prevention);   // begin lock acquistion
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

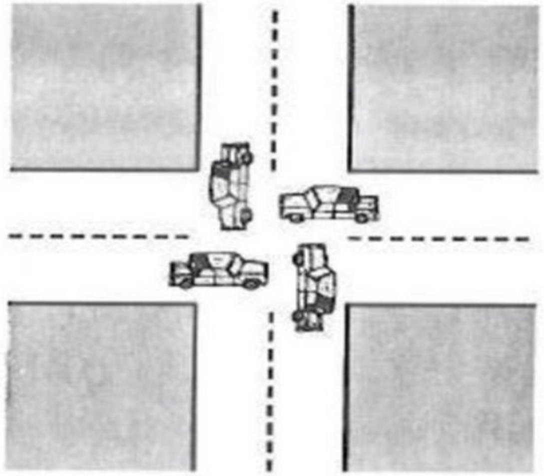
Acquire all locks atomically (Super-Lock)

Deadlock

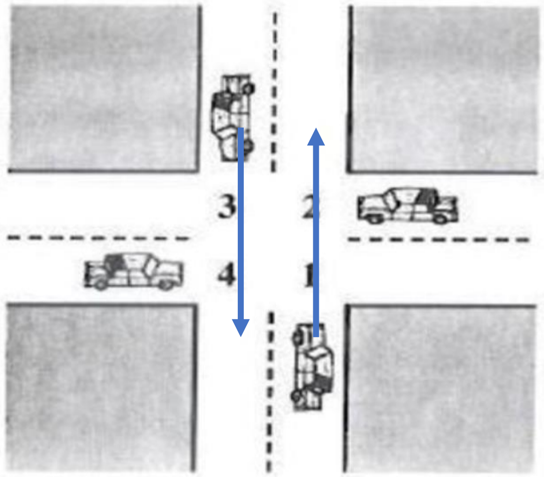
- Deadlock prevention
  - This strategy seeks to prevent one of the 4 Deadlock conditions
    - 2. Circular Wait
      - A total ordering on lock acquisition



(a) Deadlock Possible



(b) Deadlock



(a) Deadlock Possible

P7: Circular Wait

- Prepare

  > cp philosophers.c cp philosophers_circular.c

  > vim philosophers_circular.c

P7: Circular Wait

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define NUM_PHILS 5
pthread_mutex_t forks[NUM_PHILS];

void init();
int leftOf(int i);
int rightOf(int i);
void* philosopher(void* param);
void think(int id);
void eat(int id);
void get_forks(int id);
void put_forks(int id);

int main(){
    pthread_t *thd_arr; // thread array
    thd_arr = malloc(sizeof(pthread_t) * NUM_PHILS);

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_mutex_init(/*fill blanks*/, NULL);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_create(/*fill blanks*/, NULL,
                       philosopher, (void*) &i);
        usleep((1 + rand() % 50) * 10000);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_join(/*fill blanks*/, NULL);
    }
    return 0;
}
```

```c
int leftOf(int i){
    return (i) % NUM_PHILS;
}

int rightOf(int i){
    return (i + 1) % NUM_PHILS;
}

void* philosopher(void* param){
    int id = *((int *) param);
    while(1){
        think(id);
        get_forks(id);
        eat(id);
        put_forks(id);
    }
}

void think(int id){
    printf("%d: Now, I'm thinking...\n", id);
}

void eat(int id){
    printf("%d: Now, I'm eating...\n", id);
}
```

```c
void get_forks(int id){
    if (/*fill blanks*/){
        /*fill blanks*/
        /*fill blanks*/
    } else{
    pthread_mutex_lock(/*fill blanks*/);
    pthread_mutex_lock(/*fill blanks*/);
    }
}

void put_forks(int id){
    pthread_mutex_unlock(/*fill blanks*/);
    pthread_mutex_unlock(/*fill blanks*/);
}
```

P7: Circular Wait

- Run

  > gcc –o philosophers_circular.out philosophers_circular.c -lpthread

  > ./ philosophers_circular.out

Deadlock

- Deadlock prevention
  - This strategy seeks to prevent one of the 4 Deadlock conditions
    - 3. No Preemption
      - Release lock if it can not hold another lock
      - Concern: 1) may cause Livelock, 2) sometimes require undo
        - • Two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks → add random delay

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

Release lock if it can not hold another lock

P8: No Preemption

- Prepare

  > cp philosophers.c cp philosophers_no_preempthion.c

  > vim philosophers_no_preempthion.c

P8: No Preemption


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define NUM_PHILS 5
pthread_mutex_t forks[NUM_PHILS];

void init();
int leftOf(int i);
int rightOf(int i);
void* philosopher(void* param);
void think(int id);
void eat(int id);
void get_forks(int id);
void put_forks(int id);

int main(){
    pthread_t *thd_arr; // thread array
    thd_arr = malloc(sizeof(pthread_t) * NUM_PHILS);

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_mutex_init(/*fill blanks*/, NULL);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_create(/*fill blanks*/, NULL,
                        philosopher, (void*) &i);
        usleep((1 + rand() % 50) * 10000);
    }

    for(int i = 0; i < NUM_PHILS; i++){
        pthread_join(/*fill blanks*/, NULL);
    }
    return 0;
}
```

```c
int leftOf(int i){
    return (i) % NUM_PHILS;
}

int rightOf(int i){
    return (i + 1) % NUM_PHILS;
}

void* philosopher(void* param){
    int id = *((int *) param);
    while(1){
        think(id);
        get_forks(id);
        eat(id);
        put_forks(id);
    }
}

void think(int id){
    printf("%d: Now, I'm thinking...\n", id);
}

void eat(int id){
    printf("%d: Now, I'm eating...\n", id);
}
```

```c
void get_forks(int id){
    while (1){
        /*fill blanks*/
        if (/*fill blanks*/) != 0){
            /*fill blanks*/
        } else{
            /*fill blanks*/
        }
    }
}

void put_forks(int id){
    pthread_mutex_unlock(/*fill blanks*/);
    pthread_mutex_unlock(/*fill blanks*/);
}
```

P8: No Preemption

- Run

  > gcc –o philosophers_no_preemption.out philosophers_no_preemption.c -lpthread

  > ./ philosophers_no_preemption.out

Deadlock

- Deadlock prevention
  - This strategy seeks to prevent one of the 4 Deadlock conditions
    - 4. Mutual Exclusion:
      - "lock free" approach: no lock but support mutual exclusion
        - Using powerful hardware instructions, we can build data structures in a manner that does not require explicit locking
      - Atomic integer operation with compare-and-swap (chapter 28.9 in LN 4)

```
void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}
```

Using Lock

```
void AtomicIncrement(int *value, int amount) {
  do {
    int old = *value;
  } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Lock free

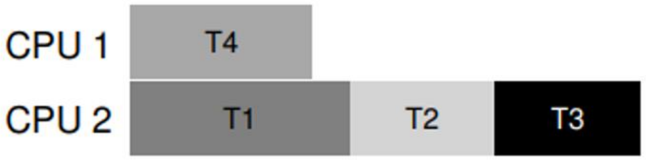
Deadlock

- Deadlock Avoidance via Scheduling
  - Instead of prevention, try to avoid by scheduling threads in a way as to guarantee no deadlock can occur.
    - E.g.) two CPUs, four threads, T1 wants to use L1 and L2, T2 also wants both, T3 wants L1 only, T4 wants nothing

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

    - E.g. 2) more contention (negative for load balancing)

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no



    - No deadlock, but under-utilization → A conservative approach