Lecture Note 5. Concurrency: Semaphore and Deadlock

April 15, 2025

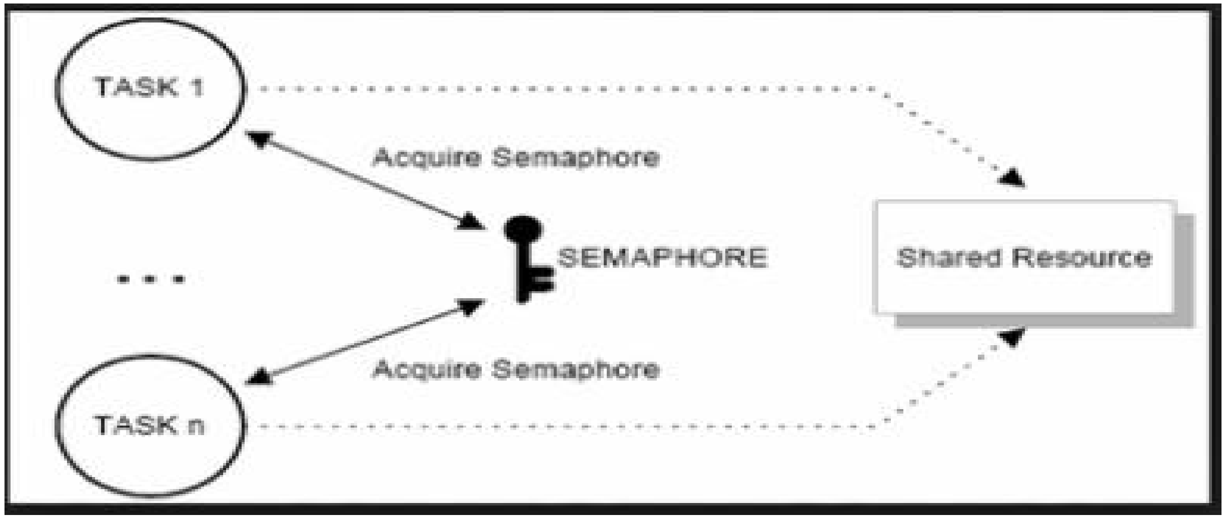
Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

(본 교재는 **2025**년도 과학기술정보통신부 및 정보통신기획평가원의 '**SW**중심대학사업' 지원을 받아 제작 되었습니다.)

Contents

- From Chap 30~34 of the OSTEP
- Chap 30. Condition Variables
- Chap 31. Semaphores
- Chap 32. Common Concurrency Problems
- Chap 33. Event-based Concurrency
- Chap 34. Summary



(Source: https://www.crocus.co.kr/1261)

Chap. 30 Condition Variables

- Locks
  - ✓ Mainly focusing on mutual exclusion
- Condition variables
  - ✓ Focusing on synchronization (not only mutual exclusion but also ordering)
  - ✓ Specifically, used for checking whether a condition is true
    - E.g.: 1) whether a child has completed. 2) whether a buffer is filled

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Figure 30.1

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

- Feasible solution 1: busy waiting with a variable

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

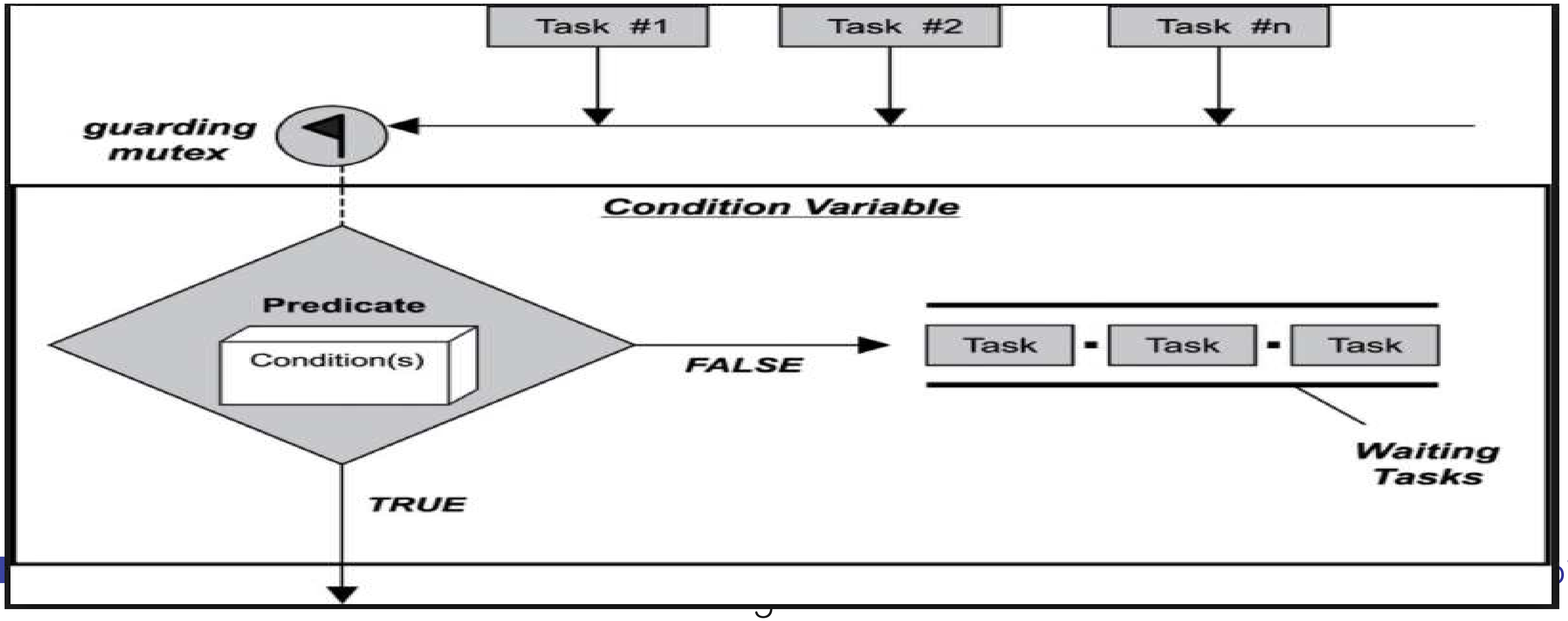
Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

- ✓ Generally work, but inefficient (waste CPU time), sometimes can be incorrect (e.g. multiple children case)
- ✓ What we would like here instead is some way to put the parent to sleep until the condition we are waiting for (e.g., the child is done executing) comes true.

30.1 Definition and Routines

- Feasible solution 2: condition variable
  - ✓ An explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired
  - ✓ Some other thread, when it changes state, can then wake one (or more) of those waiting threads and thus allow them to continue.
  - ✓ pthread APIs

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```



30.1 Definition and Routines

- Feasible solution 2: condition variable
  - ✓ Condition variable example

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Figure 30.4: **Parent Waiting: No State Variable**

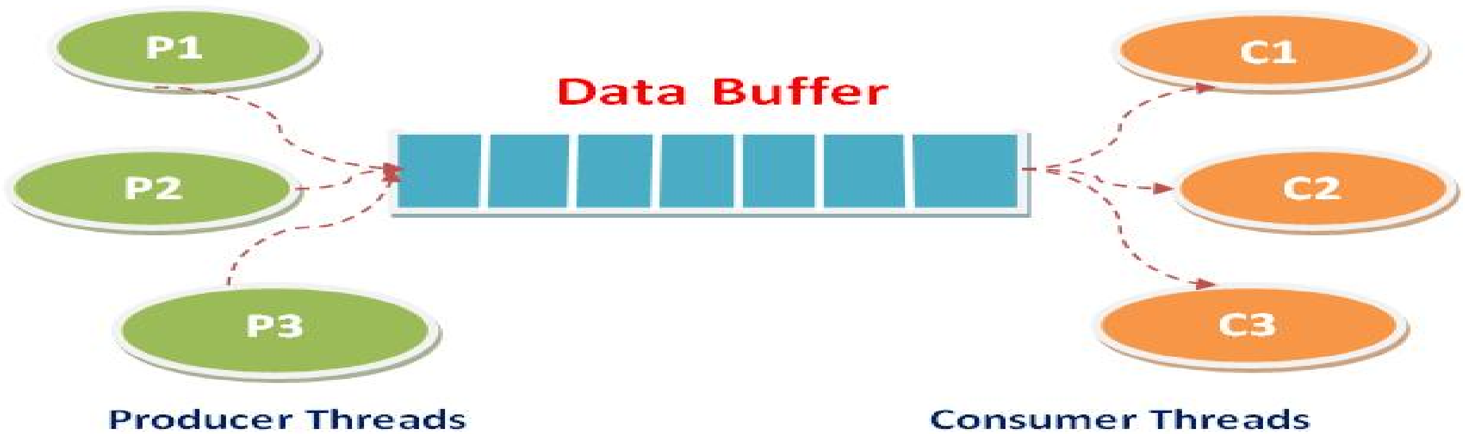
```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Figure 30.5: **Parent Waiting: No Lock**

  - ✓ Note: 1) Need done(state variable)?, 2) Need lock? 3) how about if instead of while in join(), 4) wait(): unlock/lock implicitly

30.2 Producer/Consumer (Bounded Buffer) Problem

- The famous Producer/Consumer problem (also known as bounded buffer problem)
  - ✓ Scenario
    - Producers generate data items and place them in a buffer
    - Consumers grab the items from the buffer and consume them
    - e.g. DB server, streaming server, pipe, cache, …
  - ✓ Issues
    - Mutual exclusion
    - Empty case: no data (need condition check)
    - Full case: no available buffer (need condition check)

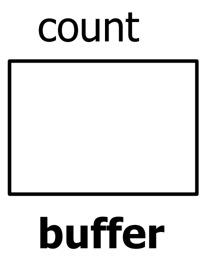


30.2 Producer/Consumer (Bounded Buffer) Problem

- Basic structure: without considering sharing
  - ✓ Shared buffer: put(), get() interfaces
    - Assumption: space for only one item (single buffer) ➔ relax later
  - ✓ Producer/Consumer: producer(), consumer()

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

Figure 30.6: The Put And Get Routines (v1)




```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

Figure 30.7: Producer/Consumer Threads (v1)

30.2 Producer/Consumer (Bounded Buffer) Problem

- Solution 1: Now consider sharing
  - ✓ Mutual exclusion: mutex
  - ✓ Ordering: condition variable

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // p1
        if (count == 1)                          // p2
            Pthread_cond_wait(&cond, &mutex);    // p3
        put(i);                                  // p4
        Pthread_cond_signal(&cond);              // p5
        Pthread_mutex_unlock(&mutex);            // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // c1
        if (count == 0)                          // c2
            Pthread_cond_wait(&cond, &mutex);    // c3
        int tmp = get();                         // c4
        Pthread_cond_signal(&cond);              // c5
        Pthread_mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.8: **Producer/Consumer: Single CV And If Statement**

☛ Is it correct?

30.2 Producer/Consumer (Bounded Buffer) Problem

- Solution 1 (cont')
  - ✓ Wake up C1, but run C2

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Figure 30.9: Thread Trace: Broken Solution (v1)

30.2 Producer/Consumer (Bounded Buffer) Problem

- Solution 2
  - ✓ while instead of if

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        while (count == 1)                    // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        while (count == 0)                    // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.10: **Producer/Consumer: Single CV And While**

☛ Now, is it correct?

30.2 Producer/Consumer (Bounded Buffer) Problem

- Solution 2 (cont')
  - ✓ Signal to P, but wake up C2

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Figure 30.11: Thread Trace: Broken Solution (v2)

30.2 Producer/Consumer (Bounded Buffer) Problem

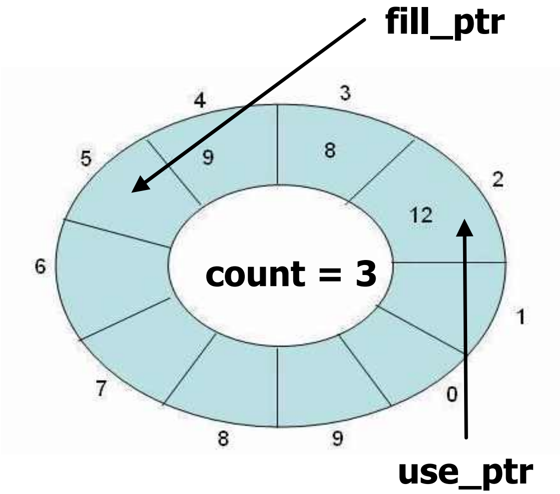
- Solution 3 (final)
  - ✓ Two condition variables
    - Indicate explicitly which thread I want to send my signal.

```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: Producer/Consumer: Two CVs And While

30.2 Producer/Consumer (Bounded Buffer) Problem

- Multiple buffers cases: final solution




```c
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.13: The Correct Put And Get Routines

```c
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // p1
        while (count == MAX)                        // p2
            Pthread_cond_wait(&empty, &mutex);      // p3
        put(i);                                     // p4
        Pthread_cond_signal(&fill);                 // p5
        Pthread_mutex_unlock(&mutex);               // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                 // c1
        while (count == 0)                          // c2
            Pthread_cond_wait(&fill, &mutex);       // c3
        int tmp = get();                            // c4
        Pthread_cond_signal(&empty);                // c5
        Pthread_mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.14: The Correct Producer/Consumer Synchronization

30.3 pthread_cond_broadcast: Covering Conditions

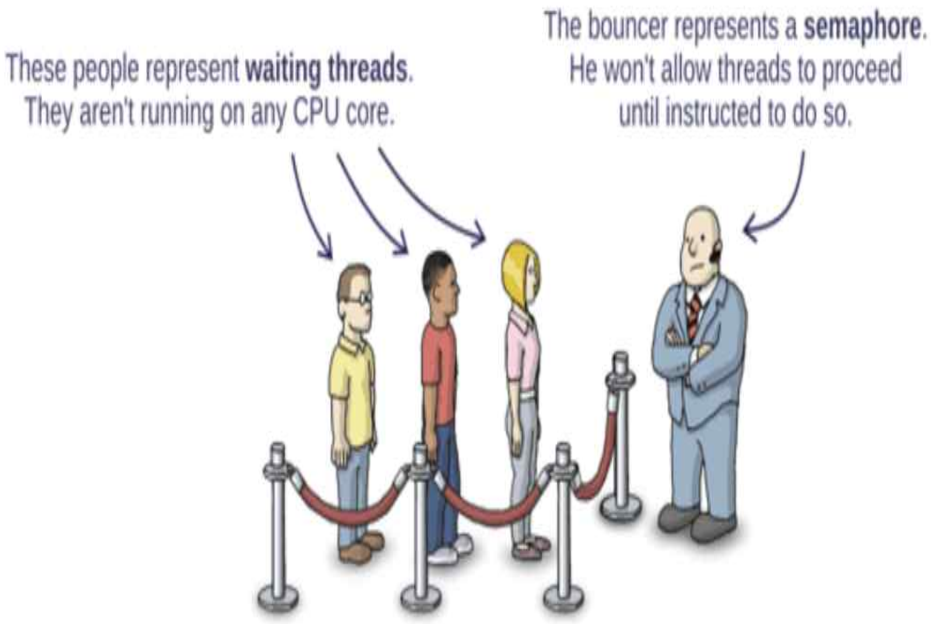
- Memory allocation library for multi-thread env.
  - ✓ Issue: which one to wake up?
    - E.g.) no free space, T1 asks 100B, T2 asks 10B, Both sleep ➔ T3 free 50B ➔ T2 wakeup: okay, T1 wakeup: sleep again, but T2 also sleeps
  - ✓ pthread_cond_broadcast() instead of pthread_cond_signal()

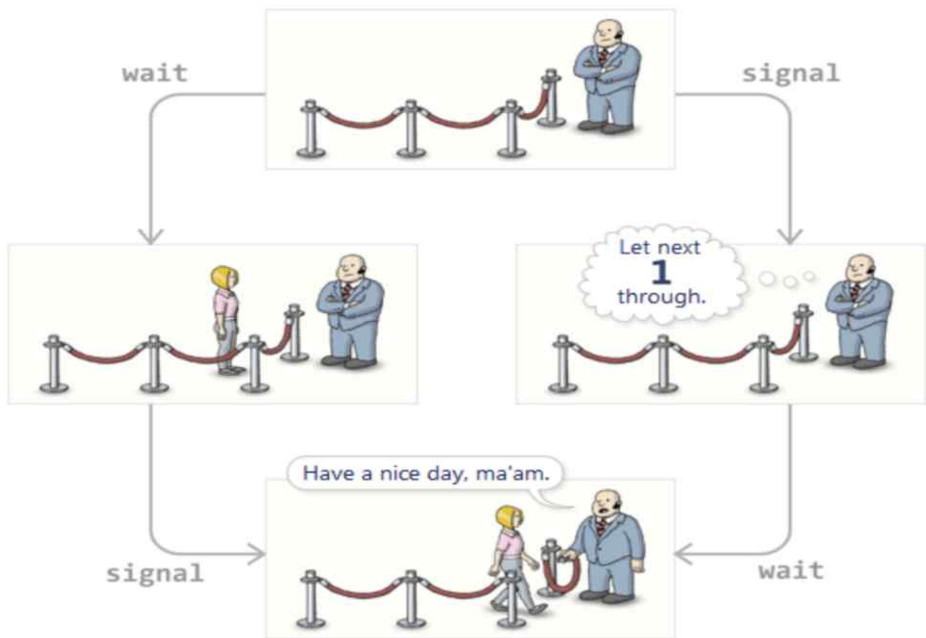
```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Figure 30.15: Covering Conditions: An Example

Chap 31. Semaphores

- 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
  - ✓ Invented by the famous Edsger Dijkstra





(Source: http://preshing.com/20150316/semaphores-are-surprisingly-versatile/)

31.1 Semaphores: A Definition

- 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**

31.2 Binary Semaphores (Locks)

- Using a semaphore as a lock

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore 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)**

- ✓ Running example
  - Can support the mutual exclusion
  - Note that the value of the semaphore, when negative, is equal to the number of waiting threads

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit sect: begin)	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem<0)→sleep	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	(crit sect: end)	Running		Sleeping
-1	call sem_post()	Running		Sleeping
0	increment sem	Running		Sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	sem_wait() returns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

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

31.3 Semaphores for Ordering

- Using a semaphore as a conditional variable
  - ✓ Initial semaphore value: 0 (note: it is initialized as 1 for mutex)

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 31.6: A Parent Waiting For Its Child

☛ Compare semaphore (this page) with condition variable (page 6) → No "Done" variable

31.4 Producer/Consumer (Bounded Buffer) Problem

- Using a semaphore for the producer/consumer problem
  - ✓ mutex: binary semaphore, full/empty: counting semaphore

```
int buffer[MAX];
int fill  = 0;
int use   = 0;

void put(int value) {
    buffer[fill] = value;       // Line F1
    fill = (fill + 1) % MAX;    // Line F2
}

int get() {
    int tmp = buffer[use];      // Line G1
    use = (use + 1) % MAX;      // Line G2
    return tmp;
}
```

Figure 31.9: T

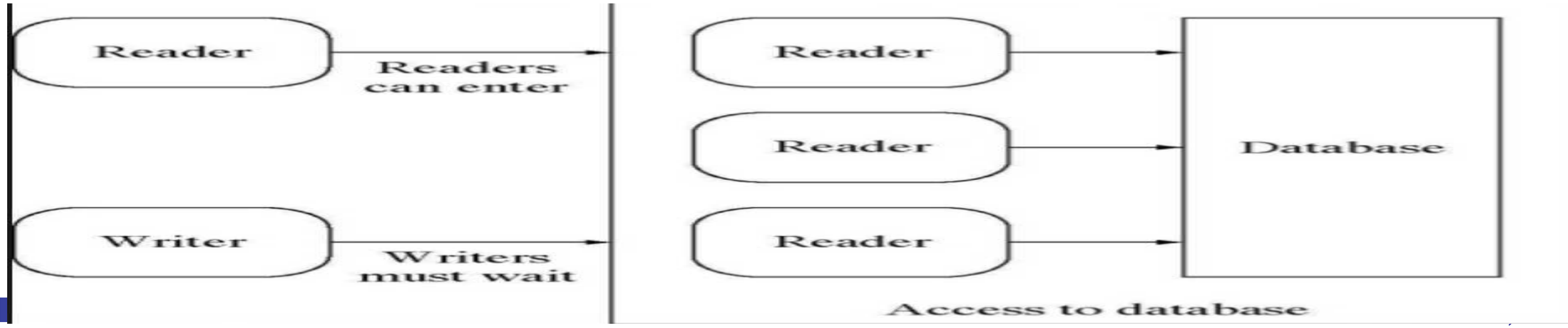
```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++)
        sem_wait(&empty);
        sem_wait(&mutex);           // Line P1.5 (MOVED MUTEX HERE...)
        put(i);                     // Line P2
        sem_post(&mutex);           // Line P2.5 (... AND HERE)
        sem_post(&full);            // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // Line C1
        sem_wait(&mutex);           // Line C1.5 (MOVED MUTEX HERE...)
        int tmp = get();            // Line C2
        sem_post(&mutex);           // Line C2.5 (... AND HERE)
        sem_post(&empty);           // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // ...
}
```

Figure 31.12: Adding Mutual Exclusion (Correctly)

☛ Summary of two versions (semaphore in page 20 vs condition variable in page 14)
- 1) No count variable (owing to counting semaphore)
- 2) ordering ➔ mutex vs mutex ➔ ordering (See Figure 31.11 in OSTEP)

31.5 Reader-Writer Locks

- Producer/Consumer vs. Reader/Writer
  - ✓ Producer/Consumer: need mutual exclusion (e.g. list insert/delete)
  - ✓ Reader/Writer: need mutual exclusion, but allow multiple readers (e.g. tree lookup and insert)
    - Specific comparison
      - A Producer or Consumer in Critical Section ➔ next Producer or Consumer must wait
      - A writer in Critical Section ➔ 1) next writer or 2) next reader must wait
      - A reader in Critical Section ➔ 1) next writer must wait, 2) but next reader can enter (better performance)
    - Issue (related to starvation)
      - Readers in Critical Section + a writer is waiting ➔ a reader arrives : wait or allowed (depending on either writer preference or reader preference)



31.5 Reader-Writer Locks

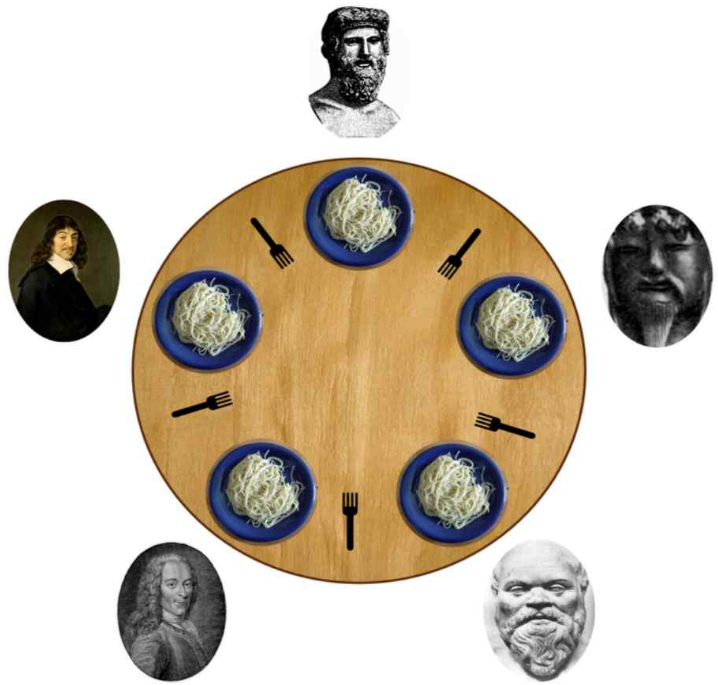
- Implementation for reader/writer
  - ✓ lock: for mutual exclusion on readers
  - ✓ writelock: to allow a write or multiple readers
    - This implementation prefers to readers (writers can starve in this version)

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases writelock
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figure 31.13: A Simple Reader-Writer Lock

31.6 The Dining Philosophers

- 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



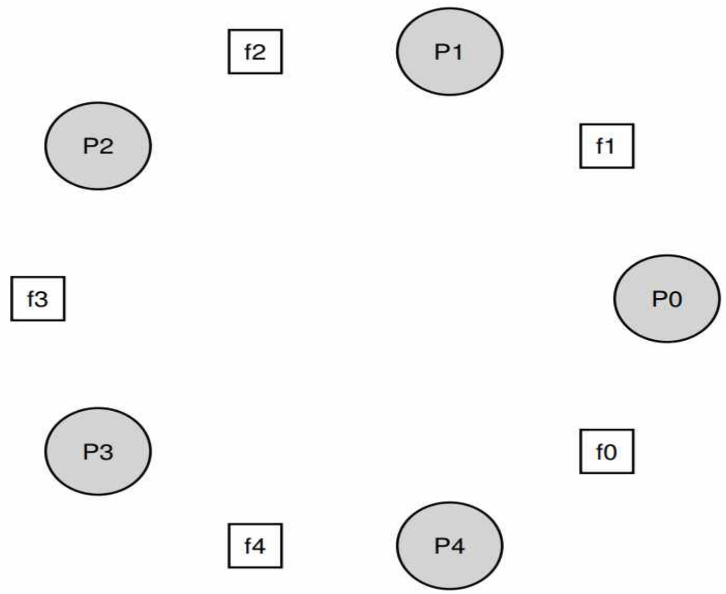


Figure 31.14: **The Dining Philosophers**

31.6 The Dining Philosophers

- Solution
  - ✓ Basic loop for each philosopher
  - ✓ Now question is how to implement getforks() and putforks()
    - Using five semaphores: sem_t forks[5]
    - Obtain semaphore before acquire a fork

```
while (1) {
    think();
    getforks();
    eat();
    putforks();
}
```

(Basic loop)

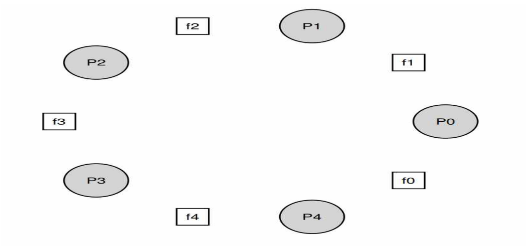


Figure 31.14: The Dining Philosophers

```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

Figure 31.15: The **get_forks()** And **put_forks()** Routines

(First solution)

  - ✓ Cause Deadlock
    - All philosophers obtain their left fork, while waiting their right one
    - How to avoid this issue?

- New Solutions
  - ✓ 1) break dependency (break ordering)
  - ✓ 2) set limit
  - ✓ 3) employ transaction (e.g. the Monitor)
  - ✓ 4) more resource
  - ✓ 5) teach philosophers (idea from a student)

```
while (1) {
   think();
   getforks();
   eat();
   putforks();
}
```

(Basic loop)

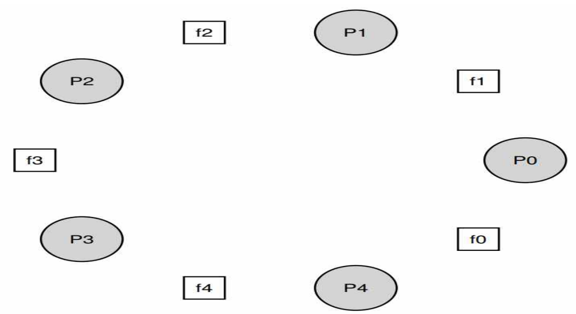


Figure 31.14: The Dining Philosophers

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

Figure 31.16: Breaking The Dependency In get_forks()

(New solution)

Chap 32. Common Concurrency Problems

- Concurrency
  - ✓ Pros: can enhance throughput via processing in parallel
  - ✓ Cons: may cause several troublesome concurrency bugs (a.k.a. timing bugs)

- 32.1 What Types of Concurrency Bugs Exist?

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

  - ✓ Total bugs: 105
    - Deadlock bugs: 31
    - Non-deadlock bugs : 74
  - ✓ Differ among applications

32.2 Non-Deadlock Bugs

- Two major types of non-deadlock bugs
  - ✓ Atomicity-Violation Bugs (From MySQL sources)

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

  - ✓ Order-Violation Bugs

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}

Thread 2::
void mMain(...) {
     ...
     mState = mThread->State;
     ...
}
```

- Solution to Atomicity-Violation Bugs

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

- Solution to Order-Violation Bugs

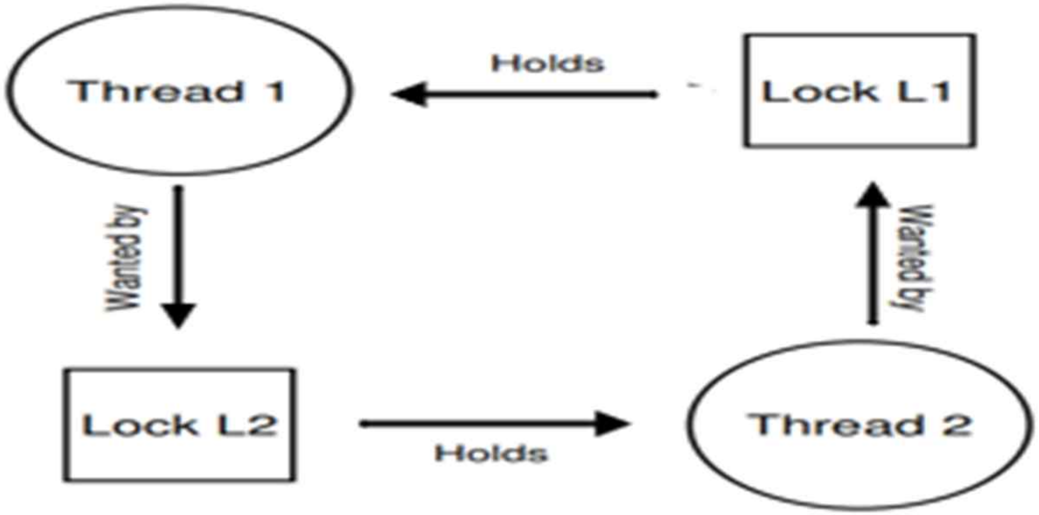
```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
int mtInit             = 0;

Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);

   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    ...
}
```

32.3 Deadlock Bugs

- 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);
```

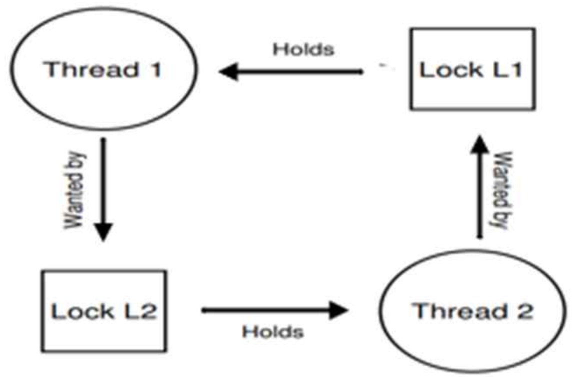
    - E.g.) When a thread (say T1) is holding a lock (L1) and waiting for another one (L2); unfortunately, the thread (T2) that holds lock L2 is waiting for L1 to be released.



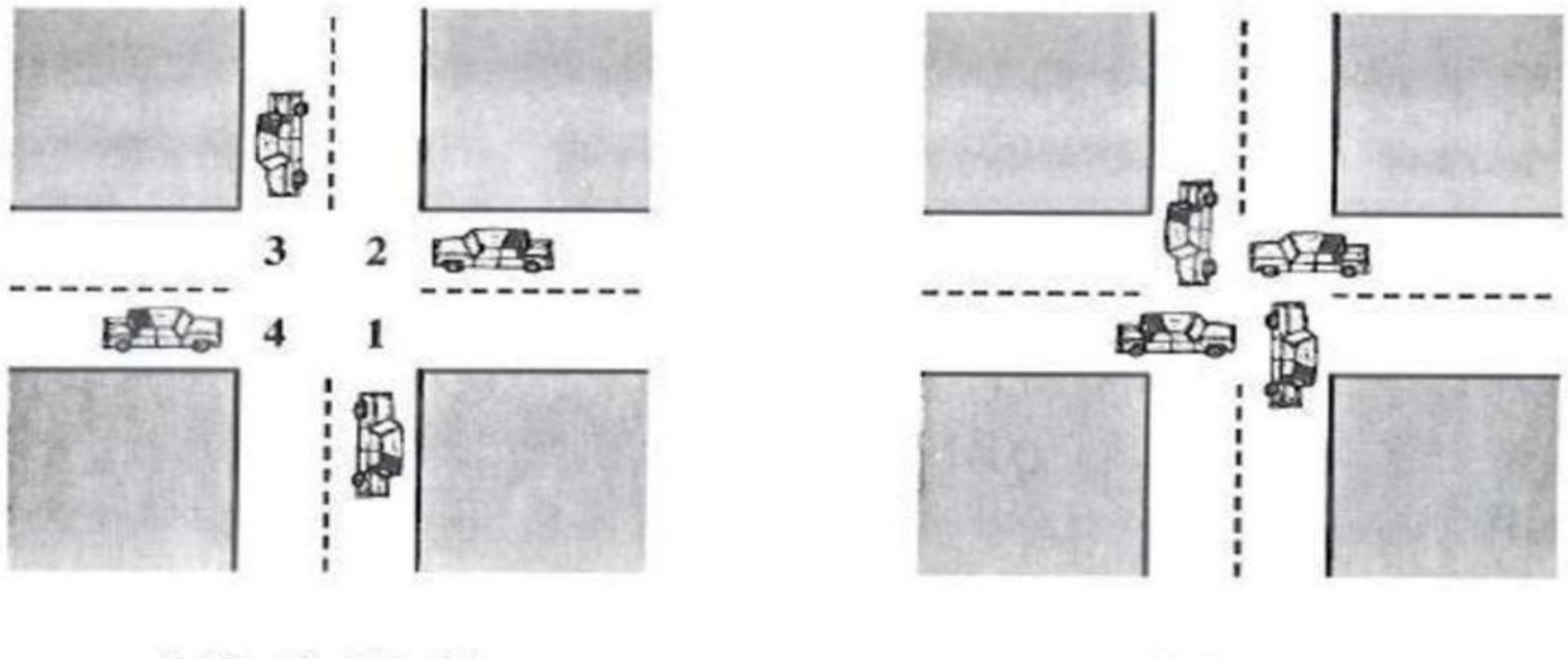
(Deadlock Dependency Graph)

32.3 Deadlock Bugs

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



(Deadlock Dependency Graph)



(a) Deadlock Possible

(b) Deadlock

(Source: Google image)

32.3 Deadlock Bugs

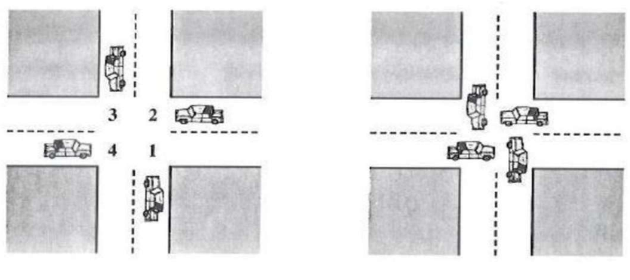
- How to handle Deadlock: three strategies
  - ✓ 1. Deadlock Prevention
  - ✓ 2. Deadlock Avoidance via Scheduling
  - ✓ 3. Deadlock Detection and Recovery

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

(Source: "Operating systems: Internals and Design Principle" by W. Stalling)

32.3 Deadlock Bugs

- Deadlock prevention
  - ✓ This strategy seeks to prevent one of the 4 Deadlock conditions
  - ✓ 1. Hold-and-wait
    - Acquire all locks at once, atomically
  - ✓ 2. 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
  - ✓ 3. Circular Wait
    - A total ordering on lock acquisition
    - E.g.) The comment at the top of the source code in Linux: *"i_mutex" before i_mmap_mutex"*



(a) Deadlock Possible (b) Deadlock

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

(Acquire all locks atomically)

```
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)

- Deadlock prevention (cont')
  - ✓ 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

- List management (39 page in LN4)

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  n->next  = head;
  head     = n;
}
```

Using Lock

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  pthread_mutex_lock(listlock);    // begin critical section
  n->next  = head;
  head     = n;
  pthread_mutex_unlock(listlock); // end critical section
}
```

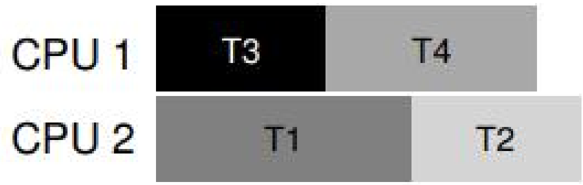
Lock free

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  do {
    n->next = head;
  } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

- Lock free: applicable only some specific cases vs Lock: general

- 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

- Deadlock Avoidance via Scheduling (cont')
  - ✓ Famous algorithm: Banker's algorithm
    - E.g.) Multiple processes with single resource case (also applicable to multiple resources case)

	Has	Max
A	0	5
B	0	6
C	0	3
D	0	7

Initial State: Free =10

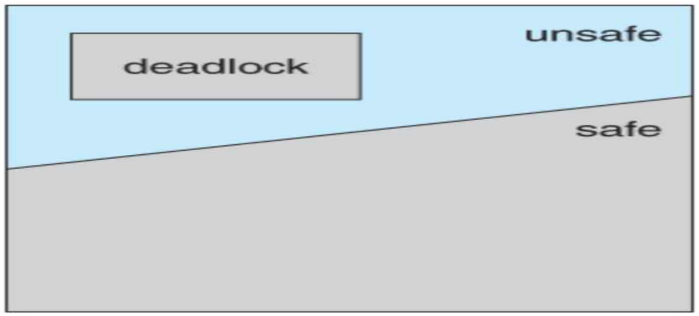
	Has	Max
A	2	5
B	0	6
C	1	3
D	5	7

State 1: Free =2

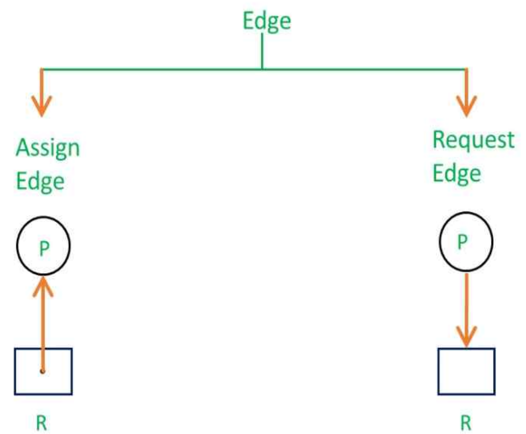
	Has	Max
A	2	5
B	1	6
C	1	3
D	5	7

State 2: Free =1

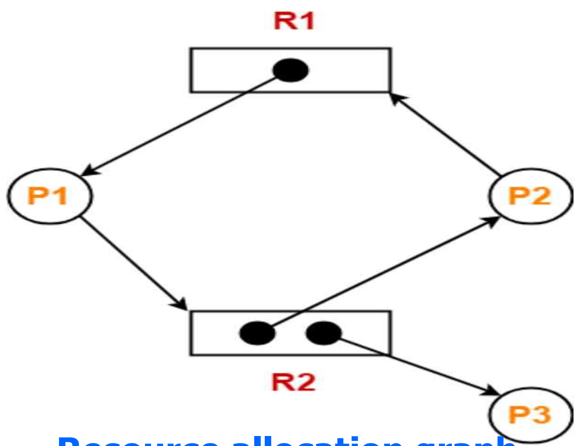
- Safe and unsafe state
  - Try to stay in safe state while allocating resources



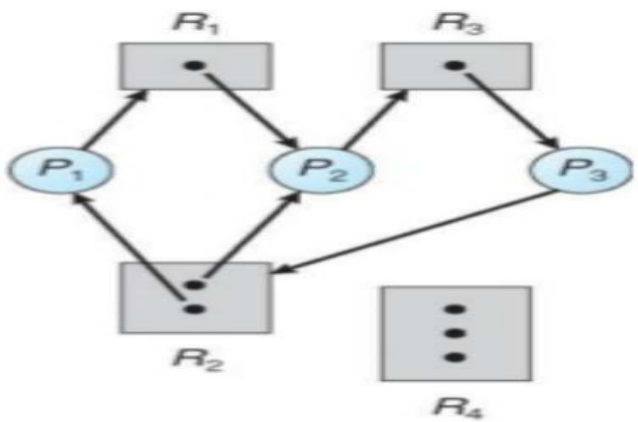
- Deadlock Detection and Recovery
  - ✓ Allow deadlocks to occasionally occur, and then take a detection and recovery action
    - E.g.) If an OS froze once a year, you would just reboot it (but failure is a norm in a Cloud/Bigdata platform)
    - Many DB systems employ active deadlock detection approach
  - ✓ How to detect?
    - Periodically, build resource allocation graph, checking in for cycles
  - ✓ How to recovery?
    - Select a victim (youngest or least locks)



Meaning of Node and Edge in Resource allocation graph



Resource allocation graph Example without Deadlock



Resource allocation graph Example with Deadlock

(Source: https://www.slideshare.net/AbhinawRai/deadlock-51330115)

33 Summary Dialogue on Currency

Professor: *Indeed it is. I am always amazed that when concurrent execution is involved, just a few lines of code can become nearly impossible to understand.*

Student: *Me too! It's kind of embarrassing, as a Computer Scientist, not to be able to make sense of five lines of code.*

Professor: *Oh, don't feel too badly. If you look through the first papers on concurrent algorithms, they are sometimes wrong! And the authors often professors!*

Student: *(gasps) Professors can be ... umm... wrong?*

Professor: *Yes, it is true. Though don't tell anybody — it's one of our trade secrets.*

Student: *I am sworn to secrecy. But if concurrent code is so hard to think about, and so hard to get right, how are we supposed to write correct concurrent code?*

Professor: *Well that is the real question, isn't it? I think it starts with a few simple things. First, keep it simple! Avoid complex interactions between threads, and use well-known and tried-and-true ways to manage thread interactions.*

Student: *Like simple locking, and maybe a producer-consumer queue?*

Professor: *Exactly! Those are common paradigms, and you should be able to produce the working solutions given what you've learned. Second, only use concurrency when absolutely needed; avoid it if at all possible. There is nothing worse than premature optimization of a program.*

Student: *I see — why add threads if you don't need them?*

Professor: *Exactly. Third, if you really need parallelism, seek it in other simplified forms. For example, the Map-Reduce method for writing parallel data analysis code is an excellent example of achieving parallelism without having to handle any of the horrific complexities of locks, condition variables, and the other nasty things we've talked about.*

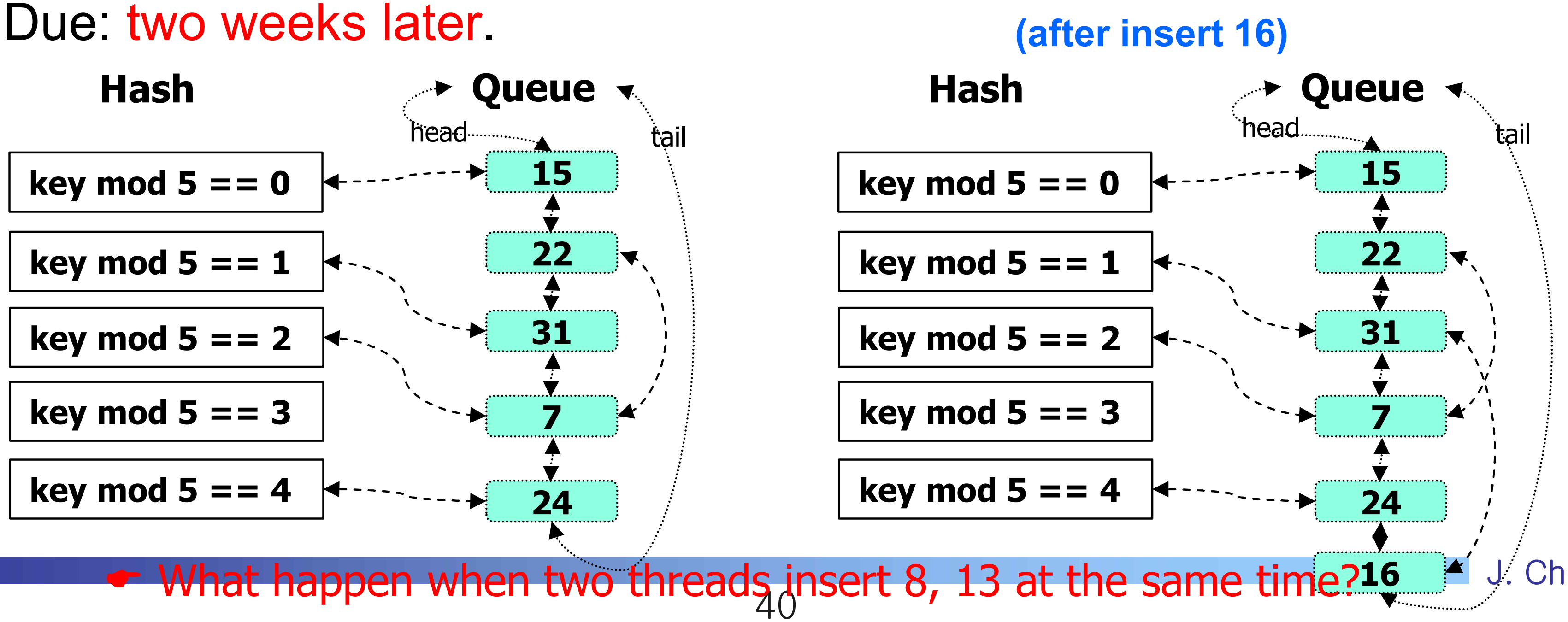
Summary

- Concurrency method
  - ✓ Lock, Condition variable, Semaphore, …
- Well-known concurrency problems
  - ✓ The Producer/Consumer problem
  - ✓ The Reader/Writer problem
  - ✓ The Dining philosopher problem
- Concurrency bugs
  - ✓ Non-Deadlock bugs
  - ✓ Deadlock bugs
- Deadlock approach
  - ✓ Prevention strategy
  - ✓ Avoidance strategy
  - ✓ Detection and Recovery strategy

> **TIP: DON'T ALWAYS DO IT PERFECTLY (TOM WEST'S LAW)**
> Tom West, famous as the subject of the classic computer-industry book *Soul of a New Machine* [K81], says famously: "Not everything worth doing is worth doing well", which is a terrific engineering maxim. If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small. If, on the other hand, you are building a space shuttle, and the cost of something going wrong is the space shuttle blowing up, well, perhaps you should ignore this piece of advice.

Lab 2: Concurrent Data Structure

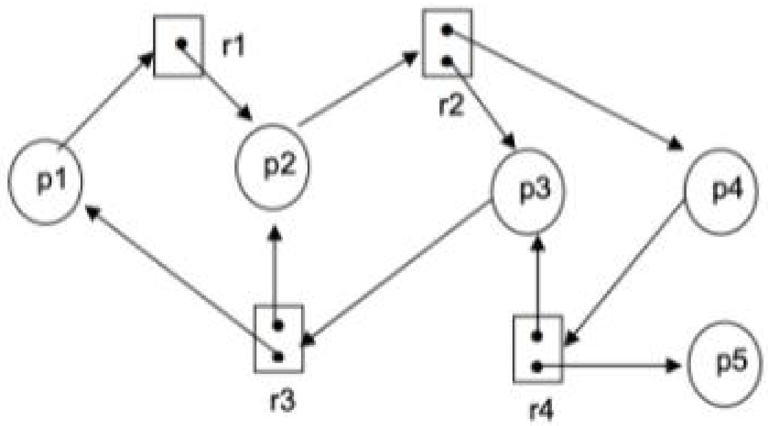
- What to do?
  - ✓ Goal
    - Make a concurrent data structure (for example Queue or Hash or Skip list, …)
    - See Lab. 2 in https://github.com/DKU-EmbeddedSystem-Lab/2025_DKU_OS
  - ✓ How to submit?
    - 1) Report (Sections: Goal, Design, Result, Discussion), 2) Source code (with Makefile) ➔ upload at Google Form or email to TA(yeojinoh@dankook.ac.kr)
  - ✓ Requirement
    - Three comparisons: 1) with/without locks, 2) fined-grained/coarse grained lock, 3) Performance under different number of threads
  - ✓ Due: two weeks later.



☛ What happen when two threads insert 8, 13 at the same time?

Quiz for this Lecture

- Quiz
  - ✓ 1. Explain the three issues that we need to consider for the producer/consumer problem.
  - ✓ 2. Describe whether the program in Figure 30.8 is correct or not? If incorrect, discuss why?
  - ✓ 3. Explain the meaning of semaphore value in Figure 31.5. Is it possible that this value becomes -2?
  - ✓ 4. Discuss the differences between the producer/consumer and reader/writer problem (at lease 2 differences).
  - ✓ 5. Is there a deadlock in the below right resource allocation graph?



(Source: www.chegg.com/homework-help/questions-and-answers/)

Appendix 1

- 31.4 Producer/Consumer (Bounded Buffer) Problem
  - ✓ Second attempt: Adding mutual exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // Line P0 (NEW LINE)
        sem_wait(&empty);           // Line P1
        put(i);                     // Line P2
        sem_post(&full);            // Line P3
        sem_post(&mutex);           // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // Line C0 (NEW LINE)
        sem_wait(&full);            // Line C1
        int tmp = get();            // Line C2
        sem_post(&empty);           // Line C3
        sem_post(&mutex);           // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
    // ...
}
```

☛ Is it correct?

Figure 31.11: Adding Mutual Exclusion (Incorrectly)

Appendix 1

- 31.7 How to Implement Semaphores
  - ✓ Using mutex and condition variable

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Figure 31.16: Implementing Zemaphores With Locks And CVs

- 본 교재는 2025년도 과학기술정보통신부 및 정보통신기획평가원의 'SW중심대학사업' 지원을 받아 제작 되었습니다.
- 본 결과물의 내용을 전재할 수 없으며, 인용(재사용)할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원이 지원한 'SW중심대학'의 결과물이라는 출처를 밝혀야 합니다.

IITP 정보통신기획평가원

디지털인재양성단 SW인재팀