

Lecture Note 5. Concurrency: Semaphore and Deadlock

April 12, 2020

Jongmoo Choi

Dept. of software
Dankook University

<http://embedded.dankook.ac.kr/~choijm>

(This slide is made by Jongmoo Choi. Please let him know when you want to distribute this slide)

Contents

From Chap 30~32 of the OSTEP

Chap 30. Condition Variables

Chap 31. Semaphores

Chap 32. Common Concurrency Problems

Chap 33. Event-based Concurrency

Chap 34. Summary



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

```
1  void *child(void *arg) {
2      printf("child\n");
3      // XXX how to indicate we are done?
4      return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8      printf("parent: begin\n");
9      pthread_t c;
10     Pthread_create(&c, NULL, child, NULL); // create child
11     // XXX how to wait for child?
12     printf("parent: end\n");
13     return 0;
14 }
```

Figure 30.1: A Parent Waiting For Its Child

Chap. 30 Condition Variables

Feasible solution 1: busy waiting with a variable

```
1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     while (done == 0) // (비교문제)
14     ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

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

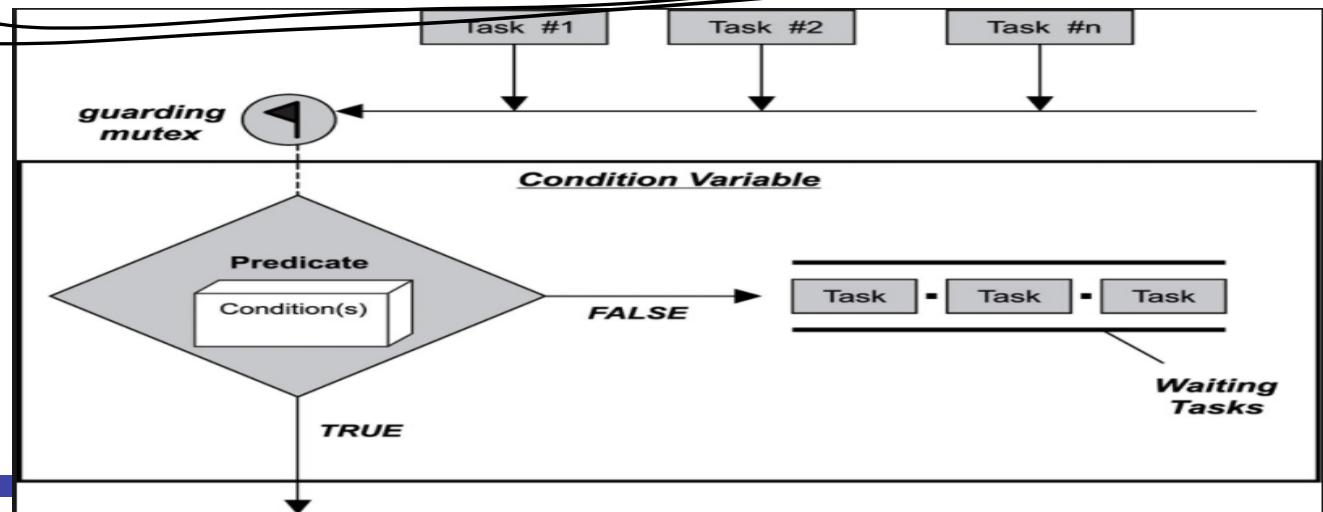
30.1 Definition and Routines

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

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

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

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

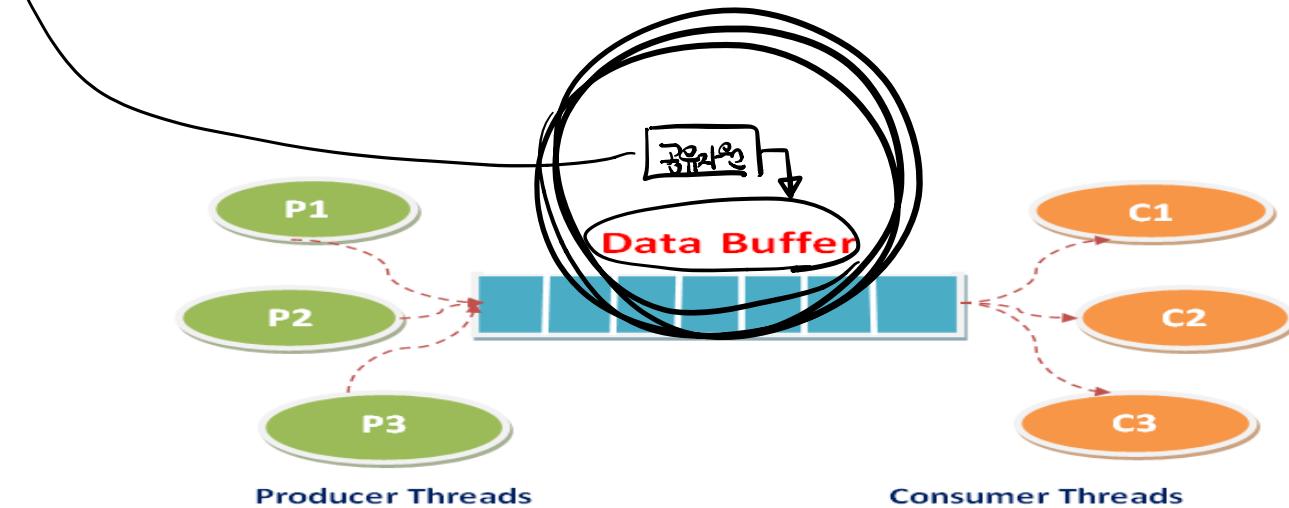
e.g. DB server, streaming server, pipe, cache, ...

- ✓ Issue

Mutual exclusion

Empty case: no data

Full case: no available buffer



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) Later relax later
- ✓ Producer/Consumer: `producer()`, `consumer()`

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0); 비어있어야지, 버퍼를 써야지.
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1); 한번 써놨으니깐 꺼내야지.
12     count = 0;
13     return buffer;
14 }
```

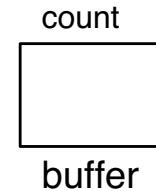


Figure 30.4: The Put And Get Routines (Version 1)

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     int i;
11     while (1) {
12         int tmp = get(); 꺼내고, 출력해야지.
13         printf("%d\n", tmp);
14     }
15 }
```

Annotations on the right side of the code:

- A red oval encloses the `put(i);` line in the producer function, with the handwritten note "제작업, 버퍼에 넣어라" (Put into the buffer).
- A red oval encloses the `get()` call in the consumer function, with the handwritten note "꺼내면 꺼내면 된다." (If you take it out, you can take it out).
- A red oval encloses the `printf` statement in the consumer function, with the handwritten note "제작업, 출력해야지" (Output it).

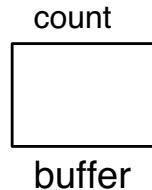
Figure 30.5: Producer/Consumer Threads (Version 1)

30.2 Producer/Consumer (Bounded Buffer)

Problem

Solution 1: Now consider sharing

- ✓ Mutual exclusion: mutex
- ✓ Ordering: condition variable



```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);                                // p1
8          if (count == 1)                                            // p2
9              Pthread_cond_wait(&cond, &mutex);                      // p3
10         put(i);                                                 // p4
11         Pthread_cond_signal(&cond);                                // p5
12         Pthread_mutex_unlock(&mutex);                             // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);                                // c1
20         if (count == 0)                                            // c2
21             Pthread_cond_wait(&cond, &mutex);                      // c3
22         int tmp = get();                                         // c4
23         Pthread_cond_signal(&cond);                                // c5
24         Pthread_mutex_unlock(&mutex);                             // c6
25         printf("%d\n", tmp);
26     }
27 }
```

정답인가요?

Is it correct?

Figure 30.6: Producer/Consumer: Single CV And If Statement

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	
	Sleep		Ready		Ready	0	
	Sleep		Ready		Ready	0	
	Sleep		Ready		Ready	0	
	Ready		Ready		Running	0	
	Ready		Ready		Running	0	
	Ready		Ready		Running	1	
	Ready		Ready		Running	1	
	Ready		Ready		Running	1	
	Ready		Ready		Running	1	
	Ready		Ready		Running	1	
	Ready		Ready		Sleep	1	
	Ready	c1	Running		Sleep	1	Buffer full; sleep T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data T_p awoken
	Ready	c5	Running		Ready	0	
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data 

Figure 30.7: Thread Trace: Broken Solution (Version 1)

```
16 void *consumer(void *arg) {  
17     int i;  
18     for (i = 0; i < loops; i++) {  
19         Pthread_mutex_lock(&mutex); // c1  
20         if (count == 0) // c2  
21             Pthread_cond_wait(&cond, &mutex); // c3  
22         int tmp = get(); // c4  
23         Pthread_cond_signal(&cond); // c5  
24         Pthread_mutex_unlock(&mutex); // c6  
25         printf("%d\n", tmp);  
26     }  
27 }
```

```
4 void *producer(void *arg) {  
5     int i;  
6     for (i = 0; i < loops; i++) {  
7         Pthread_mutex_lock(&mutex); // p1  
8         if (count == 1) // p2  
9             Pthread_cond_wait(&cond, &mutex); // p3  
10        put(i); // p4  
11        Pthread_cond_signal(&cond); // p5  
12        Pthread_mutex_unlock(&mutex); // p6  
13    }  
14 }
```

30.2 Producer/Consumer (Bounded Buffer) Problem

Solution 2

- ✓ while instead of if

```
1  cond_t cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex); // p1
8          while (count == 1) // p2
9              Pthread_cond_wait(&cond, &mutex); // p3
10         put(i); // p4
11         Pthread_cond_signal(&cond); // p5
12         Pthread_mutex_unlock(&mutex); // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex); // c1
20         while (count == 0) // c2
21             Pthread_cond_wait(&cond, &mutex); // c3
22         int tmp = get(); // c4
23         Pthread_cond_signal(&cond); // c5
24         Pthread_mutex_unlock(&mutex); // c6
25         printf("%d\n", tmp);
26     }
27 }
```

(올바른 코드 예제입니다.)

정确한 예제입니다.
정确한 예제입니다.

Now, is it correct?

Figure 30.8: Producer/Consumer: Single CV And While

30.2 Producer/Consumer (Bounded Buffer)

Producer

Solution 2 (cont')

- ✓ Signal to P, but wake up C2

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

Figure 30.9: Thread Trace: Broken Solution (Version 2)

Producer

```

16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0);                  // c2
21         Pthread_cond_wait(&cond, &mutex);    // c3
22         int tmp = get();                  // c4
23         Pthread_cond_signal(&cond);        // c5
24         Pthread_mutex_unlock(&mutex);      // c6
25         printf("%d\n", tmp);
26     }
27 }
28
29 void *producer(void *arg) {
30     int i;
31     for (i = 0; i < loops; i++) {
32         Pthread_mutex_lock(&mutex);           // p1
33         while (count == 1);                  // p2
34         Pthread_cond_wait(&cond, &mutex);    // p3
35         put(i);                          // p4
36         Pthread_cond_signal(&cond);        // p5
37         Pthread_mutex_unlock(&mutex);      // p6
38     }
39 }

```

30.2 Producer/Consumer (Bounded Buffer) Problem

Solution 3 (final)

✓ Two condition variables

Indicate explicitly which thread I want to send my signal.

→ 각각의 스레드에게
별도로 신호를.

```
1  cond_t  empty,  fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == 1)
9              Pthread_cond_wait(&empty,  &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill,  &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n",  tmp);
26     }
27 }
```

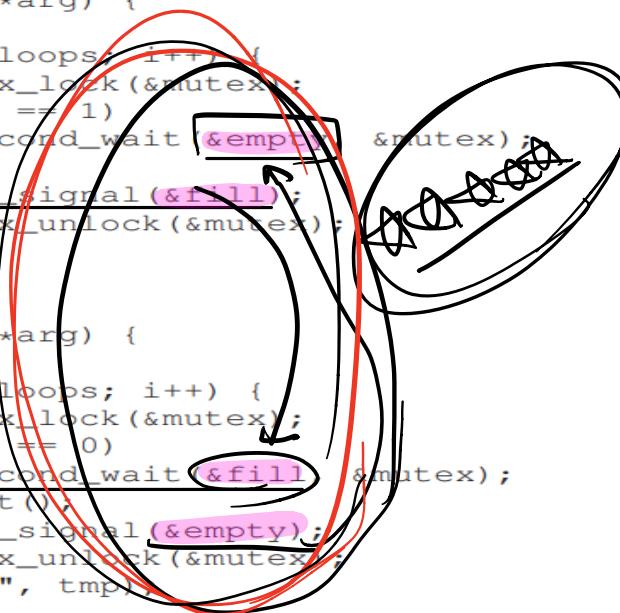


Figure 30.10: Producer/Consumer: Two CVs And While

30.2 Producer/Consumer (Bounded Buffer)

Problem

Multiple buffers cases: final solution

```
1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

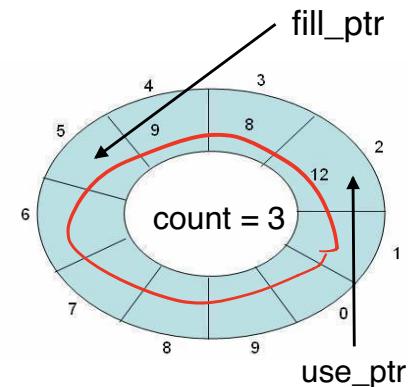


Figure 30.11: The Correct Put And Get Routines

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 30.12: 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()~~

→ context switch overhead
broadcast 1회에만 각각으로 번갈아감

```
1 // how many bytes of the heap are free?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // need lock and condition too
5 cond_t c;
6 mutex_t m;
7
8 void *
9 allocate(int size) {
10     Pthread_mutex_lock(&m);
11     while (bytesLeft < size)
12         Pthread_cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     Pthread_mutex_unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     Pthread_mutex_lock(&m);
```

Simple question to take attendance: 1) Explain that this pthread_cond_broadcast() can solve the problem discussed in page 12. 2) Also, explain why we use two condition variables shown in page 13 instead of pthread_cond_broadcast(). (until 6 PM, April 29th)

Chap 31. Semaphores

Semaphore

- ✓ Well-known structure for concurrency control

Can be used as both a lock and a condition variable

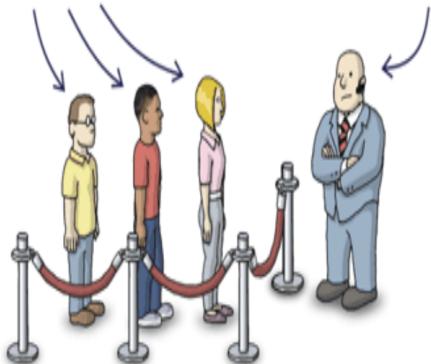
Binary semaphore, Counting semaphore

Can be employed by various concurrency problems including
and 3) dining philosophers

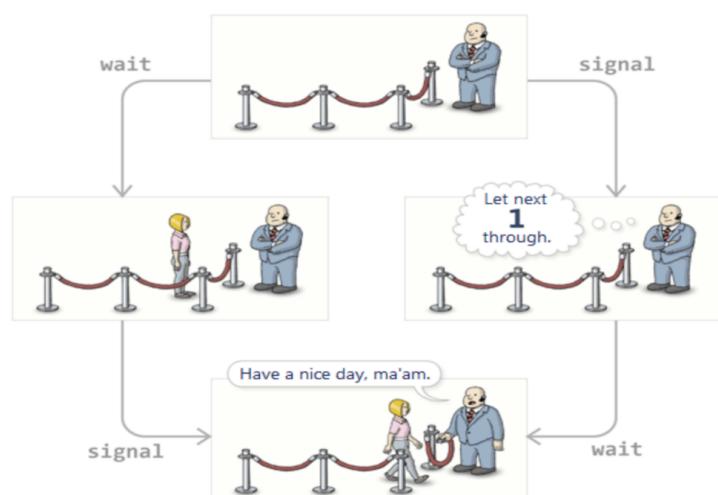
1) producer/consumer, 2) reader/writer

- ✓ Invented by the famous Edsger Dijkstra

These people represent waiting threads.
They aren't running on any CPU core.



The bouncer represents a semaphore.
He won't allow threads to proceed
until instructed to do so.



(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 $S \geq 0$) 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()

waiter
only
initial value

↑
Semaphore init 840109
initial value 551501 324762

```
1 #include <semaphore.h>
2 sem_t s;
3 sem_init(&s, 0, 1); // 1st time
```

Figure 31.1: Initializing A Semaphore

```
1 int sem_wait(sem_t *s) {
2     decrement the value of semaphore s by one
3     wait if value of semaphore s is negative
4 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }
```

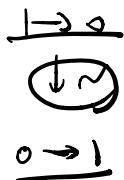
Figure 31.2: Semaphore: Definitions Of Wait And Post

31.2 Binary Semaphores (Locks)

Using a semaphore as a lock

```
1 sem_t m;
2 sem_init (&m, 0, 1); // initialize semaphore to X; what should X be?
3
4 sem_wait (&m); * Ho
5 // critical section here
6 sem_post (&m);
```

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



스레드는 세부작업의 흐름을 관리함.

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

Figure 31.5: Thread Trace: Two Threads Using A Semaphore

31.3 Semaphores for Ordering

Using a semaphore as a condition variable

- ✓ Initial Semaphore value 0 (note: it is initialized as 1 for mutex)

```
1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL; // wake parent
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     pthread_create(&c, NULL, child, NULL);
16     sem_wait(s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

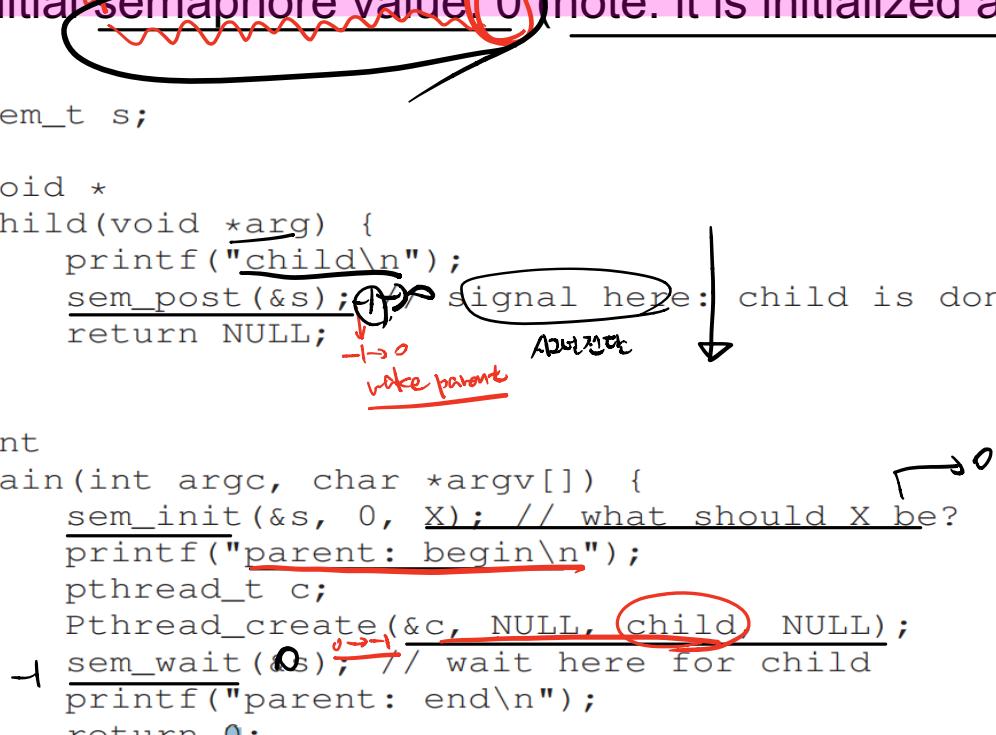


Figure 31.6: A Parent Waiting For Its Child

- ✓ 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

```
1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4
5  void put(int value) {
6      buffer[fill] = value; // Line F1
7      fill = (fill + 1) % MAX; // Line F2
8  }
9
10 int get() {
11     int tmp = buffer[use]; // Line G1
12     use = (use + 1) % MAX; // Line G2
13     return tmp;
14 }
```

Figure 31.9: 1

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         sem_wait(&empty);
9         sem_wait(&mutex);
10        put(i);
11        sem_post(&mutex);
12        sem_post(&full);
13    }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         sem_wait(&full);
20         sem_wait(&mutex);
21         int tmp = get();
22         sem_post(&mutex);
23         sem_post(&empty);
24         printf("%d\n", tmp);
25     }
26 }
27
28 int main(int argc, char *argv[]) {
29     ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0); // ... and 0 are full
32     sem_init(&mutex, 0, 1); // mutex=1 because it is a lock
33     ...
34 }
```

Figure 31.12: Adding Mutual Exclusion (Correctly)

Figure 31.9: 1

Summary of two versions (semaphore in page 20 vs condition variable in page 14)

- 1) No count variable (owing to counting semaphore)
- 2) ordering \sqsubset mutex vs mutex \sqsubset ordering (See page 40)

```
// Line P1.5 (MOVED MUTEX HERE...)
// Line P2
// Line P2.5 (... AND HERE)
// Line P3
```

```
// Line C1
// Line C1.5 (MOVED MUTEX HERE...)
// Line C2
// Line C2.5 (... AND HERE)
// Line C3
```

31.5 Reader-Writer Locks

Producer/Consumer vs. Reader/Writer

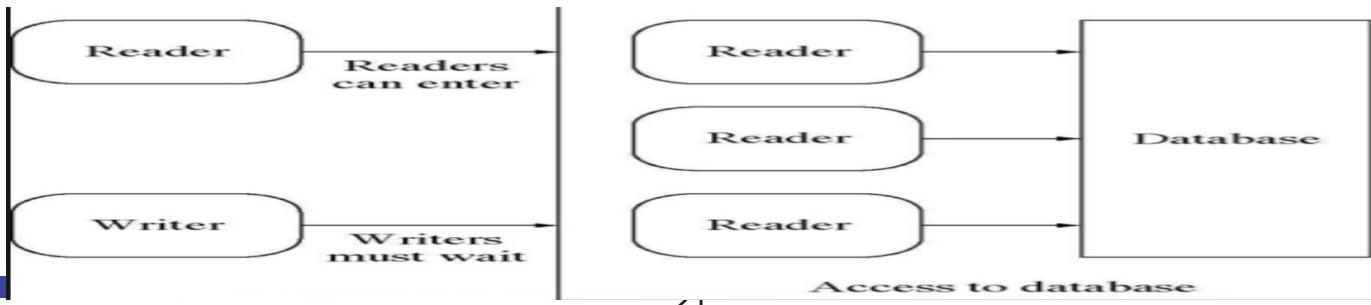
- ✓ Producer/Consumer: need mutual exclusion (e.g. list insert/delete)
- ✓ Reader/Writer: need mutual exclusion, but allow multiple readers (e.g. tree lookup and insert)

Specific comparison

- A Producer or Consumer in Critical Section \Rightarrow next Producer or Consumer must wait
- A writer in Critical Section \Rightarrow 1) next writer or 2) next reader must wait
- A reader in Critical Section \Rightarrow 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 \Rightarrow 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

The below implementation prefer readers (writers can starve)

```
1  typedef struct _rwlock_t {
2      sem_t lock;           // binary semaphore (basic lock)
3      sem_t writelock;     // used to allow ONE writer or MANY readers
4      int readers;         // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

Figure 31.13: A Simple Reader-Writer Lock

31.6 The Dining Philosophers

Problem definition

Σ

- 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 \square shared resource \square concurrency

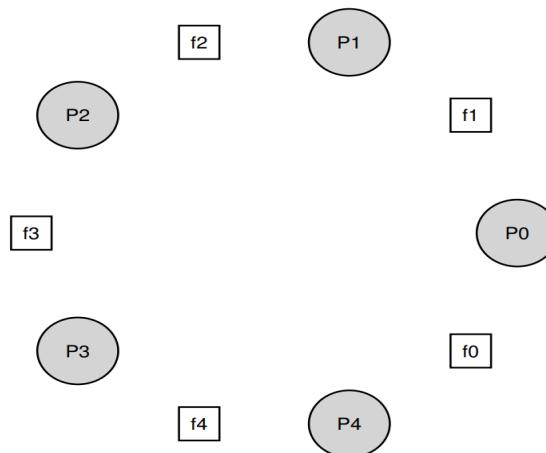
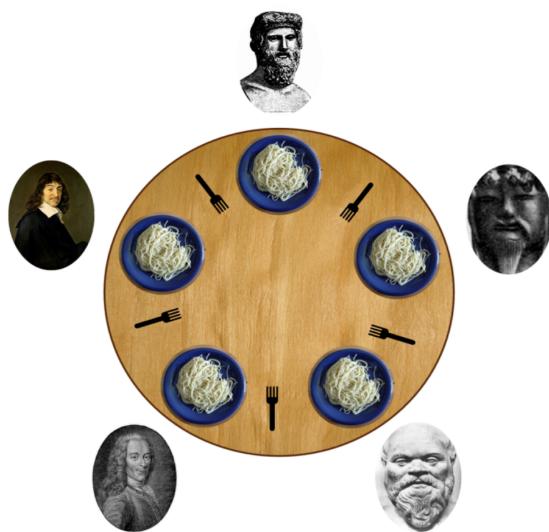


Figure 31.14: The Dining Philosophers

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]`

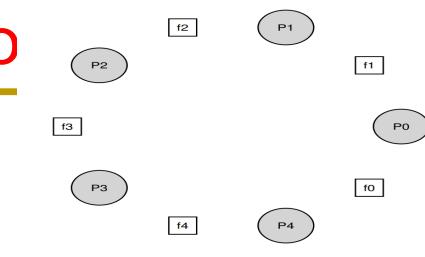
Obtain semaphore before acquire a fork

- ✓ Cause Deadlock

All philosophers obtain their left fork, while waiting their right one

How to avoid this issue?

Figure 31.14: The Dining Philosophers



```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; }

(Lower level interfaces)

1 void getforks() {
2     sem_wait(forks[left(p)]);
3     sem_wait(forks[right(p)]);
4 }

5

6 void putforks() {
7     sem_post(forks[left(p)]);
8     sem_post(forks[right(p)]);
9 }

(Interfaces used by philosophers)
```

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

(Basic loop)

Figure 31.15: The `getforks()` And `putforks()`

# 31.6 The Dining Philosophers

## New Solutions

### Break ordering (dependency)

```
1 void getforks () {
2 if (p == 4) {
3 sem_wait (forks [right (p)]);
4 sem_wait (forks [left (p)]);
5 } else {
6 sem_wait (forks [left (p)]);
7 sem_wait (forks [right (p)]);
8 }
9 }
```

### Using mutual exclusion (similar approach)

Get both forks altogether or not

### More resources

### Teach philosophers (from a student)

...

Figure 31.14: The Dining Philosophers

t at the same



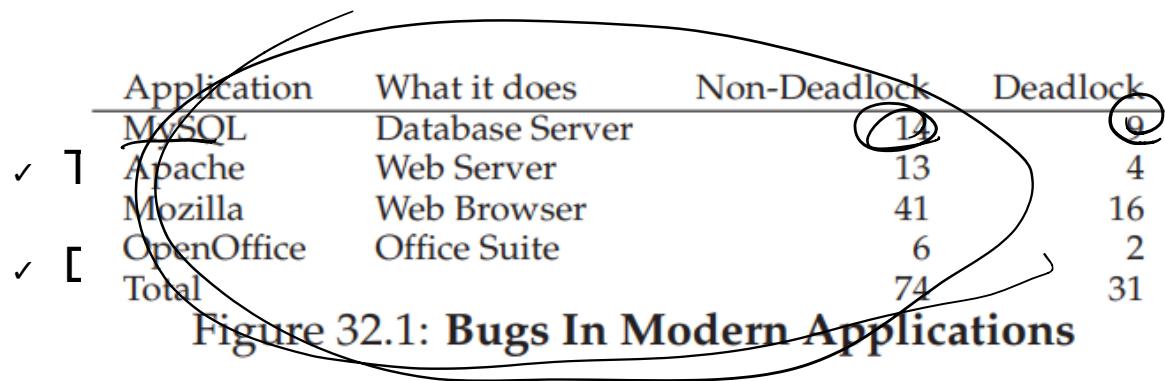
- Simple question to take attendance: In page 20, assume that MAX = 2. 1) Explain the values of two counting semaphores (empty and full) when the scheduling order of threads are “p1, p2, p3, p4, p5, c1, c2”. 2) Discuss the meaning of the empty semaphore value in the scenario of question 1). (until 6 PM, April 30th)

# 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?



## 32.2 Non-Deadlock Bugs

Two major types of non-deadlock bugs

✗ Atomicity-Violation Bugs (From MySQL sources)

```
1 Thread 1::
2 if (thd->proc_info) {
3 ...
4 fputs(thd->proc_info, ...);
5 ...
6 }
7
8 Thread 2::
9 thd->proc_info = NULL;
```

✓ Ord

```
1 Thread 1::
2 void init() {
3 ...
4 mThread = PR_CreateThread(mMain, ...);
5 ...
6 }
7
8 Thread 2::
9 void mMain(...) {
10 ...
11 mState = mThread->State;
12 ...
13 }
```

## 32.2 Non-Deadlock Bugs

### Solution to Atomicity-Violation Bugs

```
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1:::
4 pthread_mutex_lock(&proc_info_lock);
5 if (thd->proc_info) {
6 ...
7 fputs(thd->proc_info, ...);
8 ...
9 }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2:::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```



## 32.2 Non-Deadlock Bugs

2)

### Solution to Order-Violation Bugs,

```
1 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3 int mtInit = 0;
4
5 Thread 1::
6 void init() {
7
8 ...
9
10 // signal that the thread has been created...
11 pthread_mutex_lock(&mtLock);
12 mtInit = 1;
13 pthread_cond_signal(&mtCond);
14 pthread_mutex_unlock(&mtLock);
15
16 }
17
18 Thread 2::
19 void mMain(...) {
20
21 ...
22 // wait for the thread to be initialized...
23 pthread_mutex_lock(&mtLock);
24 while (mtInit == 0)
25 pthread_cond_wait(&mtCond, &mtLock);
26 pthread_mutex_unlock(&mtLock);
27
28 mState = mThread->State;
29 }
```

⚠ `pthread_cond_wait()`: unlock and lock mutex implicitly before and after sleep (see page 6)

## 32.3 Deadlock Bugs

# Deadlock

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

Thread 1: `pthread_mutex_lock(L1);`  
Thread 2: `pthread_mutex_lock(L2);`  
E.g. `pthread_mutex_lock(L2);` `pthread_mutex_lock(L1);` ~~immediately, the~~

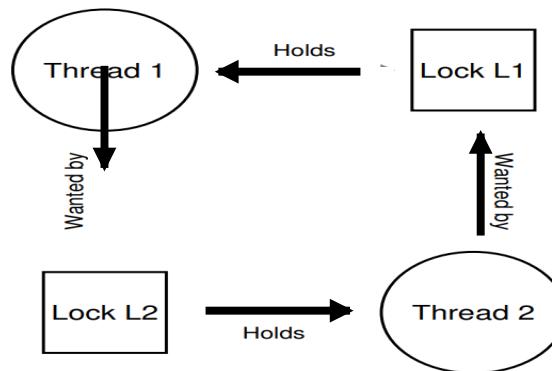
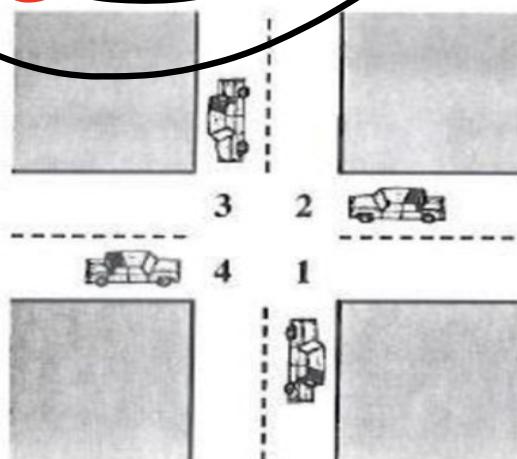


Figure 32.2: The Deadlock Dependency Graph

## 32.3 Deadlock Bugs

### Deadlock: 4 Conditions

- ✓ Mutual exclusion
- ✓ Hold-and-Wait
- ✓ No preemption for resource
- ✓ Circular wait



(a) Deadlock Possible

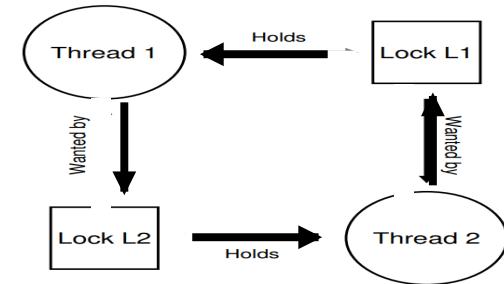
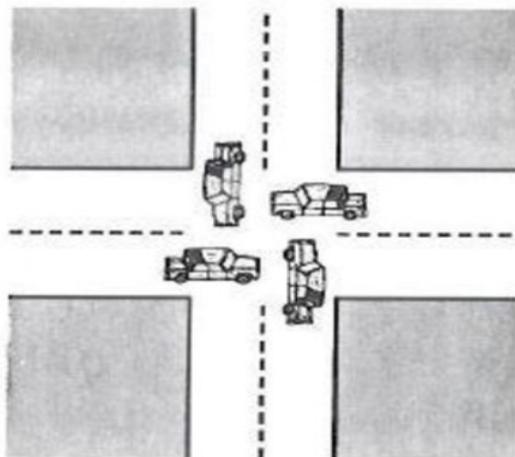


Figure 32.2: The Deadlock Dependency Graph



(b) Deadlock

## 32.3 Deadlock Bugs

How to handle Deadlock: three strategies

- ✓ 1. Deadlock Prevention
- ✓ 2. Deadlock Avoidance via **Scheduling**
- ✓ 3. Deadlock Detection and Recovery



|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

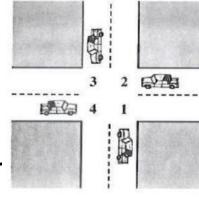
(Source: "Operating systems: Internals and Design Principle" by W. Stallings)

# 32.3 Deadlock Bugs

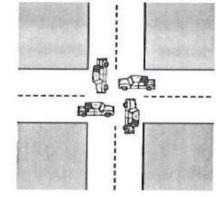
## Deadlock prevention

- ✓ 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 to acquire both locks
- ✗ 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*”

*Livelock* vs. *deadlock*



(a) Deadlock Possible



(b) Deadlock

```
1 pthread_mutex_lock(prevention); // begin lock acquisition
2 pthread_mutex_lock(L1);
3 pthread_mutex_lock(L2);
4 ...
5 pthread_mutex_unlock(prevention); // end
```

(Acquire all locks atomically)

```
1 top:
2 pthread_mutex_lock(L1);
3 if (pthread_mutex_trylock(L2) != 0) {
4 pthread_mutex_unlock(L1);
5 goto top;
6 }
```

(Release lock if it can not hold another lock)

# 32.3 Deadlock Bugs

## 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

```
1 void AtomicIncrement(int *value, int amount) {
2 do {
3 int old = *value;
4 } while (CompareAndSwap(value, old, old + amount) == 0);
5 }
```

Lock free

```
1 void insert(int value) {
2 node_t *n = malloc(sizeof(node_t));
3 assert(n != NULL);
4 n->value = value;
5 n->next = head;
6 head = n;
7 }
```

Using Lock

```
1 void insert(int value) {
2 node_t *n = malloc(sizeof(node_t));
3 assert(n != NULL);
4 n->value = value;
5 pthread_mutex_lock(listlock); // begin critical section
6 n->next = head;
7 head = n;
8 pthread_mutex_unlock(listlock); // end critical section
9 }
```

Lock free

```
1 void insert(int value) {
2 node_t *n = malloc(sizeof(node_t));
3 assert(n != NULL);
4 n->value = value;
5 do {
6 n->next = head;
7 } while (CompareAndSwap(&head, n->next, n) == 0);
8 }
```

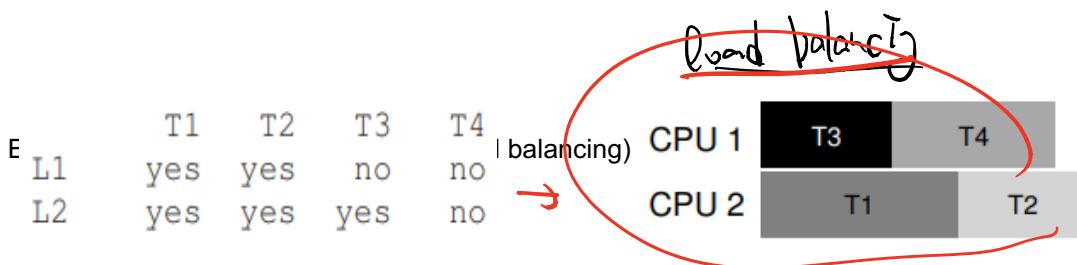
Lock free: applicable only some specific cases vs Lock: general

# 32.3 Deadlock Bugs

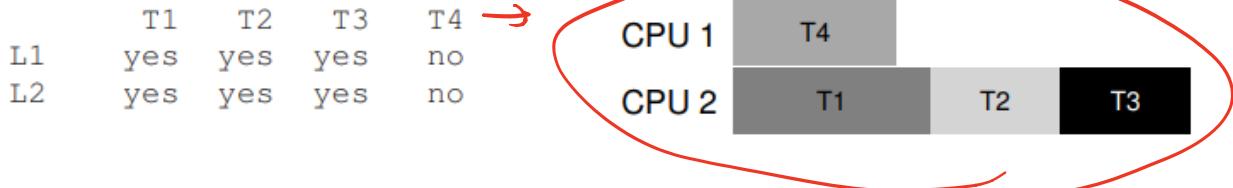
## Deadlock Avoidance via Scheduling

- 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



No deadlock, but under-utilization A conservative approach



## 32.3 Deadlock Bugs

### Deadlock Avoidance via Scheduling (cont')

- ✓ Famous algorithm: Banker's algorithm

E.g.) Multiple processes with single resource case (also applicable to multiple resources case)

↑ happens during

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

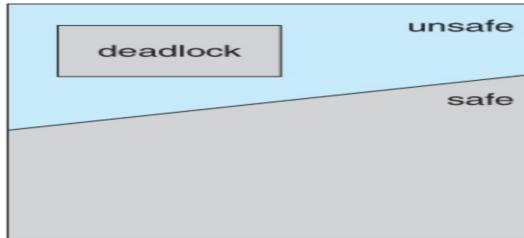
Initial State: Free = 10

|   | 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   |

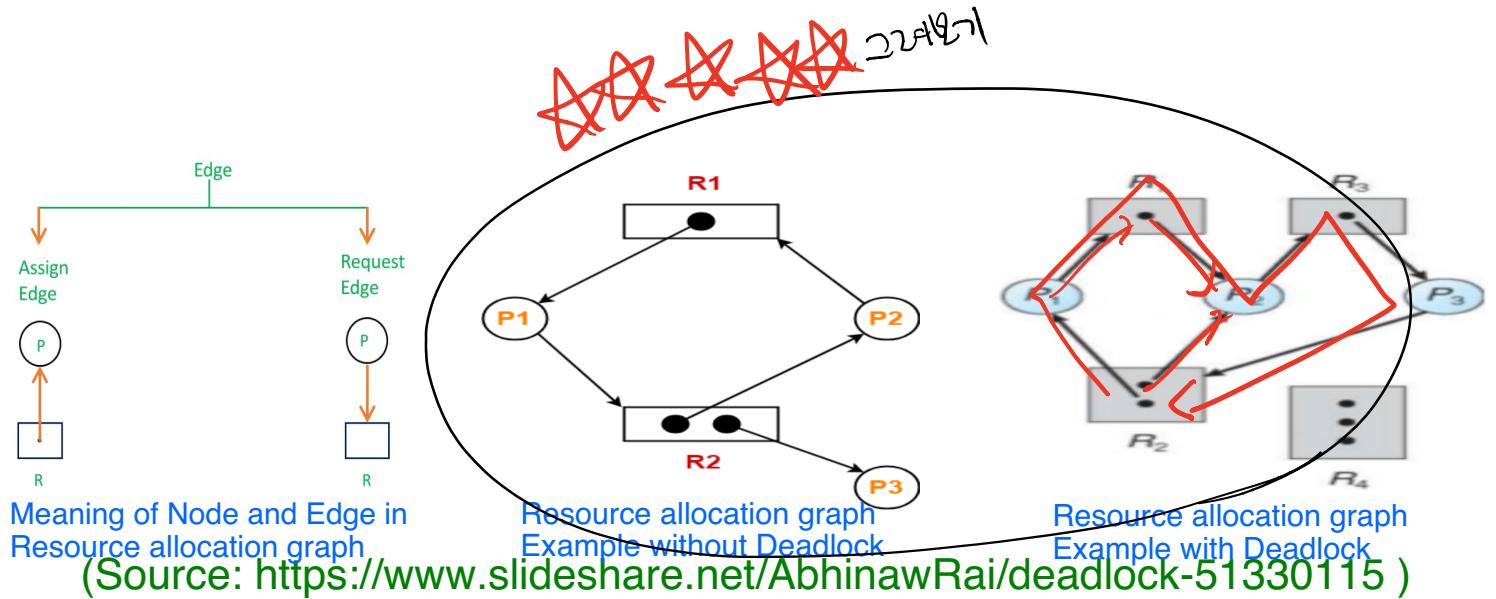
State 2: Free = 1



# 32.3 Deadlock Bugs

## 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)



## 32.4 Summary

---

### Concurrency method

- ✓ Lock, Condition variable, Semaphore, ...

### Well-known concurrency problems

- ✓ The Producer/Consumer problem
- ✓ The Reader/Writer problem
- ✓ The Dining philosopher problem

### Currency bugs

- ✓ Non-Deadlock bugs
- ✓ Deadlock bugs

### Deadlock approach

- ✓ Prevention
- ✓ Avoidance
- ✓ Detection and Recovery

#### 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.

- Simple question to take attendance: 1) Explain how the program, shown in page 40, might cause deadlock (explain a scenario that causes deadlock), 2) Discuss this deadlock scenario with the resource allocation graph. (hint: compare it with the program shown in page 20). (until 6 PM, May 7th)

# Appendix

## 31.4 Producer/Consumer (Bounded Buffer) Problem

- ✓ First attempt

```
1 int buffer[MAX];
2 int fill = 0;
3 int use = 0;
4
5 void put(int value) {
6 buffer[fill] = value; // Line F1
7 fill = (fill + 1) % MAX; // Line F2
8 }
9
10 int get() {
11 int tmp = buffer[use]; // Line G1
12 use = (use + 1) % MAX; // Line G2
13 return tmp;
14 }
```

Figure 31.9: The Put And Get Routines

```
1 sem_t empty;
2 sem_t full;
3
4 void *producer(void *arg) {
5 int i;
6 for (i = 0; i < loops; i++) {
7 sem_wait(&empty); // Line P1
8 put(i); // Line P2
9 sem_post(&full); // Line P3
10 }
11 }
12
13 void *consumer(void *arg) {
14 int i, tmp = 0;
15 while (tmp != -1) {
16 sem_wait(&full); // Line C1
17 tmp = get(); // Line C2
18 sem_post(&empty); // Line C3
19 printf("%d\n", tmp);
20 }
21 }
22
23 int main(int argc, char *argv[]) {
24 // ...
25 sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26 sem_init(&full, 0, 0); // ... and 0 are full
27 // ...
28 }
```

⌚ What do you think about this? Correct?

Figure 31.10: Adding The Full And Empty Conditions

# Appendix

## 31.4 Producer/Consumer (Bounded Buffer) Problem

- ✓ Second attempt: Adding mutual exclusion

```
1 sem_t empty;
2 sem_t full;
3 sem_t mutex;
4
5 void *producer(void *arg) {
6 int i;
7 for (i = 0; i < loops; i++) {
8 sem_wait(&mutex);
9 sem_wait(&empty);
10 put(i);
11 sem_post(&full);
12 sem_post(&mutex);
13 }
14 }
15
16 void *consumer(void *arg) {
17 int i;
18 for (i = 0; i < loops; i++) {
19 sem_wait(&mutex);
20 sem_wait(&full);
21 int tmp = get();
22 sem_post(&empty);
23 sem_post(&mutex);
24 printf("%d\n", tmp);
25 }
26 }
27
28 int main(int argc, char *argv[]) {
29 ...
30 sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31 sem_init(&full, 0, 0); // ... and 0 are full
32 sem_init(&mutex, 0, 1); // mutex=1 because it is a lock (NEW LINE)
33 ...
34 }
```

❓ How about now? Is it correct?

Figure 31.11: Adding Mutual Exclusion (Incorrectly)

## 31.7 How to Implement Semaphores

- ✓ Using mutex and condition variable

```
1 typedef struct __Zem_t {
2 int value;
3 pthread_cond_t cond;
4 pthread_mutex_t lock;
5 } Zem_t;
6
7 // only one thread can call this
8 void Zem_init(Zem_t *s, int value) {
9 s->value = value;
10 Cond_init(&s->cond);
11 Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15 Mutex_lock(&s->lock);
16 while (s->value <= 0)
17 Cond_wait(&s->cond, &s->lock);
18 s->value--;
19 Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {
23 Mutex_lock(&s->lock);
24 s->value++;
25 Cond_signal(&s->cond);
26 Mutex_unlock(&s->lock);
27 }
```

Figure 31.16: Implementing Zemaphores With Locks And CVs