Operating Systems

Condition Variable and Semaphore (Chapter 30 ~ 31)

Dr. Young-Woo Kwon

30. Condition Variables

Condition Variables

- There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- Example:
  - A parent thread might wish to check *whether a child thread has completed.*
  - This is often called a `join()`.

Condition Variables

A Parent Waiting For Its Child

```c
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;
}
```

What we would like to see here is:

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

Parent waiting fore child: Spin-based Approach

```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;
}
```

- This is hugely <u>inefficient</u> as the parent spins and **wastes CPU time**.

How to wait for a condition

- Condition variable
  - Queue of threads
  - **Waiting** on the condition
    - <u>An explicit queue</u> that threads can put themselves on when some state of execution is not as desired.
  - **Signaling** on the condition
    - Some other thread, *when it changes it state*, can wake one of those waiting threads and allow them to continue.

Definition and Routines

- Declare condition variable

```
pthread cond t c;
```

  - Proper initialization is required.

Three variables for CV:
condition variable **c**
state variable **m**
lock **l**;

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

- Operation (the POSIX calls)
  - The wait() call takes a mutex as a parameter.
    - The wait() call release the lock and put the calling thread to sleep.
    - When the thread wakes up, it must re-acquire the lock.

Parent waiting for Child: Use a condition variable

```
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);
}
```

//3 variables

//State variable

//Signaling...

//State variable

//Waiting...

Parent waiting for Child: Use a condition variable

```
(cont.)
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;
}
```

//Thread creation

//Join() can be implemented like this

Parent waiting for Child: Use a condition variable

- **Parent:**
  - Creates the child thread and continues running itself.
  - Calls into `thr_join()` to wait for the child thread to complete.
    - Acquires the lock.
    - Checks if the child is done.
    - Puts itself to sleep by calling `wait()`.
    - Releases the lock.
- **Child:**
  - Prints the message “child”.
  - Calls `thr_exit()` to wake up the parent thread.
    - Acquire the lock.
    - Sets the state variable `done`.
    - Signals the parent thus waking it.
    - Release the lock

The importance of the state variable `done`

```
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);
}
```

`//Execute first`

`//Then, waits for the child`

`thr_exit()` and `thr_join()` without variable `done`

– Imagine the case where the *child runs immediately*.
- The child will signal, but there is <u>no thread asleep</u> on the condition.
- When the parent runs, it will call wait and be stuck.
- No thread will ever wake it.

Another poor implementation

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

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

//Signaling to the parent

//Before wait() is called, ctx occurs

//Then, waits for the client

– The issue here is a subtle **race condition**.

- The parent calls `thr_join()`.
  – The parent checks the value of `done`.
  – It will see that it is 0 and try to go to sleep.
  – *Just before* it calls wait to go to sleep, the parent is <u>interrupted</u> and the child runs.
- The child changes the state variable `done` to 1 and signals.
  – But no thread is waiting and thus no thread is woken.
  – When the parent runs again, it sleeps forever.

The Producer / Consumer Problem

- **Producer**
  - **Produce** data items
  - Wish to place data items in a buffer
- **Consumer**
  - **Get** data items from the buffer and then **consume** them in some way
- Example: Multi-threaded web server
  - ***A producer puts HTTP requests*** in to a work queue
  - ***Consumer threads take requests*** out of this queue and process them

Bounded buffer

- A bounded buffer is used when you pipe the output of one program into another.
  - Example: `grep foo file.txt | wc -l`
    - The `grep` process is the producer.
    - The wc process is the consumer.
    - Between them is an in-kernel **bounded buffer.**
  - Bounded buffer is Shared resource → **Synchronized access** is required.

The Put and Get Routines (Version 1)

```
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;
}
```

- Only put data into the buffer when `count` is zero.
  - i.e., when the buffer is *empty*.
- Only get data from the buffer when `count` is one.
  - i.e., when the buffer is *full*.

Producer/Consumer Threads (Version 1)

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

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

- **Producer** puts an integer into the shared buffer loops number of times.
- **Consumer** gets the data out of that shared buffer.

Producer/Consumer: Single CV and If Statement

- A single condition variable `cond` and associated lock `mutex`

```
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
    }
}
```

//Wait until the buffer is empty

//Signals to the consumer

Producer/Consumer: Single CV and If Statement

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

//Wait until the buffer is filled

//Signals to the producer

– p1-p3: A producer waits for the buffer to be empty.

– c1-c3: A consumer waits for the buffer to be full.

– With just *a single producer* and *a single consumer*, the code works.

If we have more than one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

	TC1_State		TC2_State		TP_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	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	sneaks in …
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	**Oh oh! No data**

//Wait

//Switched to Tc2 by OS

//Woken up by TP

//Wait

Thread Trace: Broken Solution (Version 1)

- The problem arises for a simple reason:
  - After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer *changed by* T_{c2}.
  - There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
    - **Virtually every system ever built employs *Mesa semantics*.**
  - Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken.

Producer/Consumer: Single CV and While

- Consumer T_{c1} wakes up and re-checks the state of the shared variable.
  - If the buffer is empty, the consumer simply goes back to sleep.

```
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
    }
}
```

//Check the buffer again

Producer/Consumer: Single CV and While

```
(Cont.)
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);
    }
}
```

//Check the buffer again

- A simple rule to remember with condition variables is to **always use while loops**.
- **However, this code still has a bug (*next page*).**

Thread Trace: Broken Solution (Version 2)

	TC1_State		TC2_State		TP_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	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	grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke**

//Wait

//Scheduled

//Wait

//Signal to TC1

//Wait

//Woken by TP

//Signal to ????

Thread Trace: Broken Solution (Version 2) (Cont.)

	State		State		State	Count	Comment
…	…	…	…	…	…	…	(*cont.*)
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 …**

```
//Woken by TC1
```

- **A consumer should not wake other consumers, only producers, and vice-versa.**

The Single Buffer Producer/Consumer Solution

- Use **two condition variables** and while
  - **Producer** threads wait on the condition `empty`, and signals `fill`.
  - **Consumer** threads wait on `fill` and signal `empty`.

```
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);
    }
}

```

//Wait for empty buffer

//Signal to consumers

The single Buffer Producer/Consumer Solution

```
(Cont.)
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);
    }
}
```

//Wait for full buffer

//Signal to the producer

The Final Producer/Consumer Solution

- More **concurrency** and **efficiency** → **Add more buffer slots**.
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

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

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

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

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
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
    }
}

```

The Final Producer/Consumer Solution (Cont.)

```
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);
    }
}
```

The Final Working Solution (Cont.)

– p2: **A producer** only sleeps if all buffers are currently filled.

– c2: **A consumer** only sleeps if all buffers are currently empty.

Covering Conditions

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

```
// 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);
}
```

//Condition re-check

//Need to wake up the other thread?

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads.**
    - <u>Cost</u>: too many threads might be woken.
    - Threads that shouldn’t be awake will simply wake up, re-check the condition, and then go back to sleep.

31. Semaphore

Semaphore: A definition

- An object with an integer value
  - We can manipulate with two routines; `sem_wait()` and `sem_post()`.
  - Initialization

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

- Declare a semaphore `s` and initialize it to the value **1**
- The second argument, 0, indicates that the semaphore is **shared between *threads in the same process*.**

Semaphore: Interact with semaphore

- `sem_wait()`

//Decrement semaphore value by 1

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

  - If the value of the semaphore is *one* or *higher* when called `sem_wait()`, **return right away**.
  - If the value of the semaphore < 0, it will cause the caller to <u>suspend execution</u> waiting for a subsequent post.
  - When **negative**, the value of the semaphore is **equal to the number of waiting threads**.

Semaphore: Interact with semaphore (Cont.)

- `sem_post()`

//Increment semaphore value by 1

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

  - Simply **increments** the value of the semaphore.
  - If there is a thread waiting to be woken, **wakes** one of them up.

Binary Semaphores (Locks)

- What should **X** be?
  - The initial value should be **1**.

```
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);
```

Value of Semaphore	Thread 0	Thread 1
1		
1	call sema_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() returns	Running		Ready
0	(crit set: begin)	Running		
0	*Interrupt; Switch → T1*	Ready		
0		Ready	call sem_wait()	
-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	wait(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Decrement S (no waiting)

Context switching to T1

Increment S

Decrement S (Waiting because s is -1)

Sleep

Proceed to the critical section

Semaphores as Condition Variables

```
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;
}
```

Semaphore variable must be 0

A Parent Waiting For Its Child

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

The execution result

– What should **x** be?

- The value of semaphore should be set to is **0**.

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*Switch→Child*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait() retruns`	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`.

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer Problem

- **Producer**: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it.
- **Consumer**: `get()` interface
  - Wait for a buffer to become *filled* before using it.

```c
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;
}
```

The Producer/Consumer Problem

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&empty);          // line P1
            put(i);                    // line P2
            sem_post(&full);           // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
            sem_wait(&full);           // line C1
            tmp = get();               // line C2
            sem_post(&empty);          // line C3
            printf("%d\n", tmp);
    }
}
…
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer Problem

```
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
    // …
}
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- Imagine that `MAX` is greater than 1 .
  - If there are multiple producers, race condition can happen at line *f1*.
  - It means that the old data there is overwritten.
- We've forgotten here is **mutual exclusion**.
  - The filling of a buffer and incrementing of the index into the buffer is a critical section.

A Solution: 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)
    }
}

```

Mutex using semaphore

A Solution: Adding Mutual Exclusion

```c
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);
    }
}
```

Mutex using semaphore

A Solution: Adding Mutual Exclusion (Cont.)

- Imagine two thread: one producer and one consumer.
  - The consumer **acquire** the `mutex` (line c0).
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1).
  - The consumer is **blocked** and **yield** the CPU.
    - The consumer still holds the mutex!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0).
  - The producer is now **stuck** waiting too.

→ a classic deadlock.

A Working Solution

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

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
            sem_wait(&empty);         // line p1
            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
    }
}

```

Change the order

Adding Mutual Exclusion (Correctly)

A Working Solution

```
 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
     sem_init(&full, 0, 0);    // To check if the buffer is full
     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
     // …
}
             …
```

Change the order

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- Imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
  - **insert:**
    - Change the state of the list
    - A traditional critical section makes sense.
  - **lookup:**
    - Simply *read* the data structure.
    - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.

This special type of lock is known as a reader-write lock.

A Reader-Writer Locks

- Only **a single writer** can acquire the lock.
- Once a reader has acquired a read lock,
  - **More readers** will be allowed to acquire the read lock too.
  - A writer will have to wait until all readers are finished.

```
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);
}
```

A Reader-Writer Locks

```
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);
 }
```

Allow write lock only for the reader

If there are multiple readers, immediately returns

```
  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);
 }
```

A Reader-Writer Locks (Cont.)

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

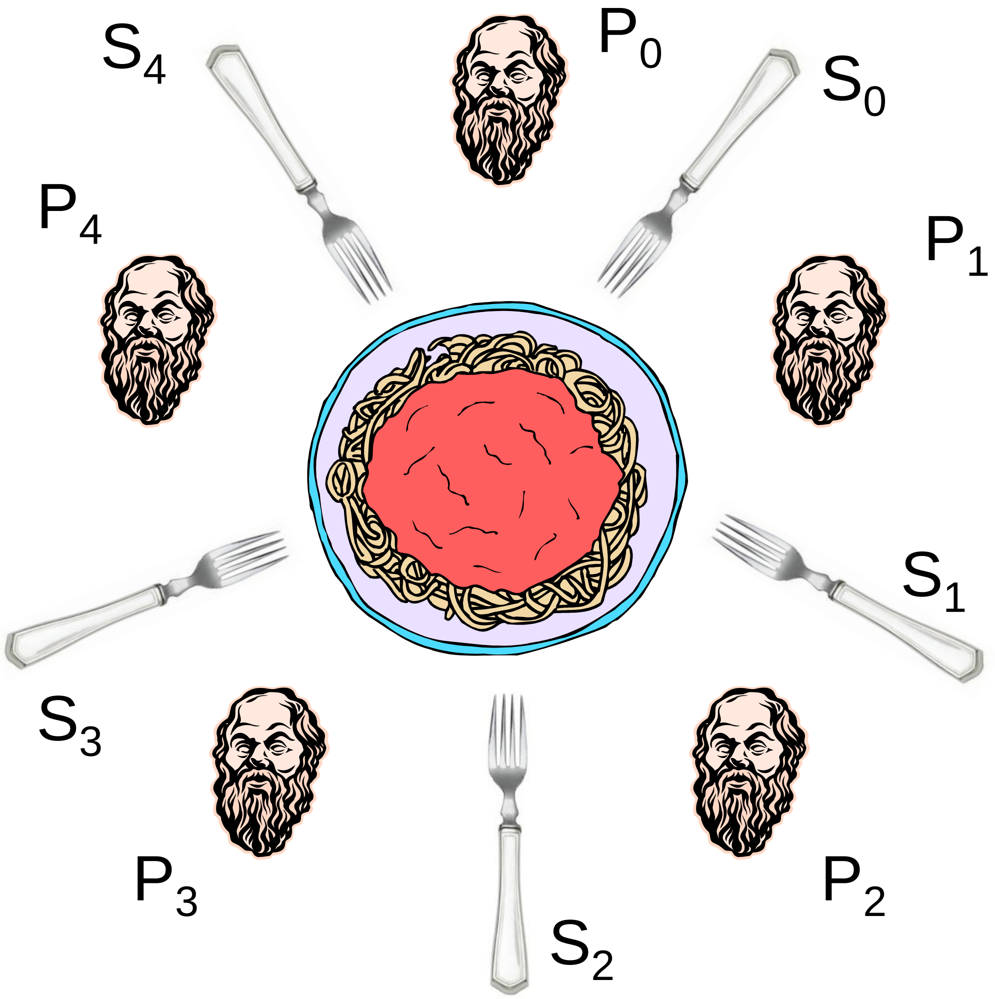
```
void rwlock_release_writelock(rwlock_t *rw) {
      sem_post(&rw->writelock);
 }
```

A Reader-Writer Locks (Cont.)

- The reader-writer locks have fairness problem.
  - It would be relatively easy for reader to **starve writer**.
  - To avoid this, prevent readers from entering the lock once a writer is waiting

Dining Philosophers (Dijkstra)

- A classic
- 5 Philosophers, 1 bowl of spaghetti
- Philosophers (threads) think & eat ad infinitum
  - Need left & right fork to eat (!?)
- Want solution that prevents starvation & does not delay hungry philosophers unnecessarily



The Dining Philosophers (Cont.)

- Key challenge
  - There is **no deadlock**.
  - **No** philosopher **starves** and never gets to eat.
  - **Concurrency** is high.

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

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

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

Helper functions (Downey's solutions)

- Philosopher `p` wishes to refer to the for on their left → call `left(p)`.
- Philosopher `p` wishes to refer to the for on their right → call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

The `getforks()` and `putforks()` Routines (Broken Solution)

  - **Deadlock** occur!
    - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
    - Each will be stuck ***holding one fork* and waiting for another, *forever***.

A Solution: Breaking The Dependency

- Change **how forks are acquired**.
  - Let's assume that philosopher 4 acquire the forks in a *different order*.

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

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken**.