



# Condition Variables, Classic Sync Problem

*CS 571: Operating Systems (Spring 2020)*

Lecture 4

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpaci-Dusseau.


Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

# Condition Variables

# Condition Variables

A parent waiting for its child

```
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 }
```



# Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

# Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

What's the problem of this approach?

# Condition Variables (CV)

- Definition:
  - An explicit queue that threads can put themselves when some **condition** is not as desired (by **waiting** on the condition)
  - Other thread can wake one of those waiting threads to allow them to continue (by **signaling** on the condition)
- Pthread CV

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

# CV-based Approach

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

```
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;  
}
```

# Broken Implementation 1

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

```
1 void thr_exit() {  
2     Pthread_mutex_lock(&m);  
3     Pthread_cond_signal(&c);  
4     Pthread_mutex_unlock(&m);  
5 }  
6  
7 void thr_join() {  
8     Pthread_mutex_lock(&m);  
9     Pthread_cond_wait(&c, &m);  
10    Pthread_mutex_unlock(&m);  
11 }
```

```
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;  
}
```

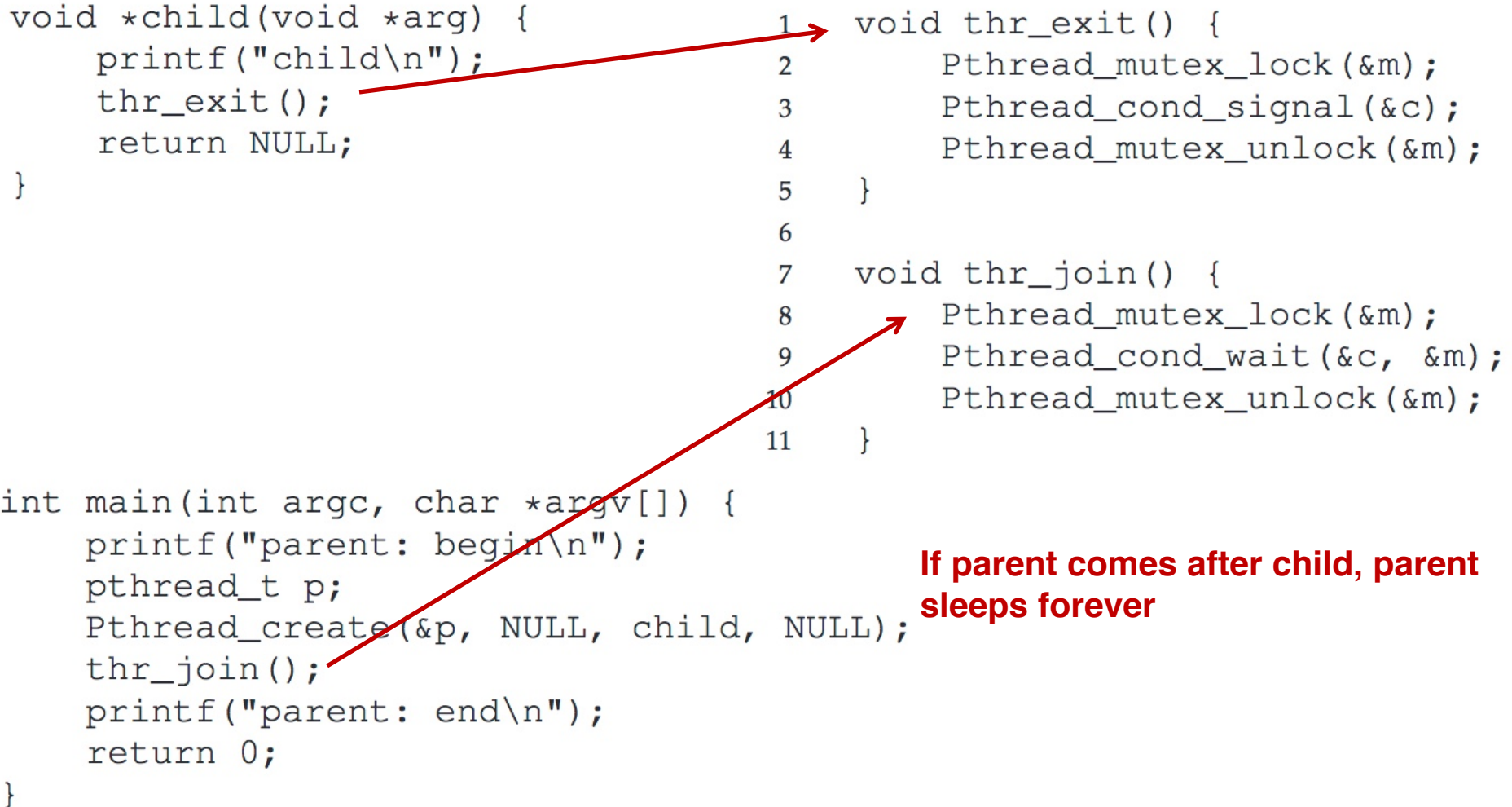


# Broken Implementation 1

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

1 void thr_exit() {
2     Pthread_mutex_lock(&m);
3     Pthread_cond_signal(&c);
4     Pthread_mutex_unlock(&m);
5 }
6
7 void thr_join() {
8     Pthread_mutex_lock(&m);
9     Pthread_cond_wait(&c, &m);
10    Pthread_mutex_unlock(&m);
11 }

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;
}
```



**If parent comes after child, parent sleeps forever**

# Broken Implementation 1

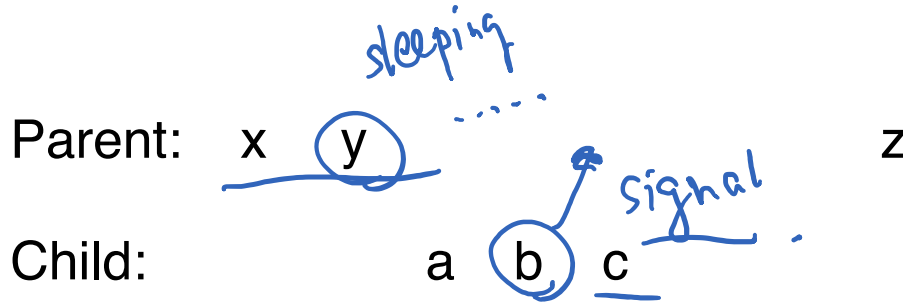
*time*



```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);         // b  
    Mutex_unlock(&m);        // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);       // y  
    Mutex_unlock(&m);        // z  
}
```

# Broken Implementation 1



```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);         // b  
    Mutex_unlock(&m);        // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);       // y  
    Mutex_unlock(&m);        // z
```

# Broken Implementation 1

Parent: x y z

Child: a b c

**GOOD!**

```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);          // b  
    Mutex_unlock(&m);        // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);        // y  
    Mutex_unlock(&m);        // z  
}
```

# Broken Implementation 1

```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);          // b  
    Mutex_unlock(&m);        // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);        // y  
    Mutex_unlock(&m);        // z  
}
```

# Broken Implementation 1

Parent:                    x    y    ..... 22 .....  
                          *signal*    *lost.*  
Child:      a  b  c  

```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);          // b  
    Mutex_unlock(&m);         // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);        // y  
    Mutex_unlock(&m);         // z  
}
```

# Broken Implementation 1

Parent:                    x   y   ... *sleeeeeeeeeep forever* ...


Child:     a   b   c

```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    Cond_signal(&c);          // b  
    Mutex_unlock(&m);        // c  
}
```

```
void thread_join() {  
    Mutex_lock(&m);           // x  
    Cond_wait(&c, &m);        // y  
    Mutex_unlock(&m);        // z  
}
```

# Broken Implementation 2

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

```
1 void thr_exit() {  
2     done = 1;  
3     Pthread_cond_signal(&c);  
4 }  
5  
6 void thr_join() {  
7     if (done == 0)   
8         Pthread_cond_wait(&c);  
9 }
```

```
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;  
}
```



# Broken Implementation 2

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

```
1 void thr_exit() {  
2     done = 1;  
3     Pthread_cond_signal(&c);  
4 }  
5  
6 void thr_join() {  
7     if (done == 0)  
8         Pthread_cond_wait(&c);  
9 }
```

```
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;  
}
```

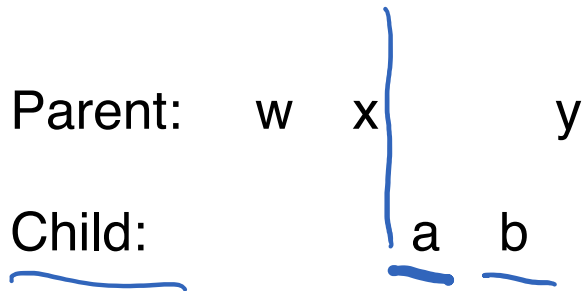
**No mutual exclusion, hence child may signal before parent calls `cond_wait()`. In this case, parent sleeps forever!**

# Broken Implementation 2

```
void thread_exit() {  
    done = 1;           // a  
    Cond_signal(&c);    // b  
}
```

```
void thread_join() {  
    Mutex_lock(&m);     // w  
    if (done == 0)     // x  
        Cond_wait(&c, &m); // y  
    Mutex_unlock(&m);  // z  
}
```

# Broken Implementation 2



```
void thread_exit() {  
    done = 1;           // a  
    Cond_signal(&c);    // b  
}
```

```
void thread_join() {  
    Mutex_lock(&m);     // w  
    if (done == 0)     // x  
        Cond_wait(&c, &m); // y  
    Mutex_unlock(&m);  // z  
}
```

# Broken Implementation 2

Parent: w x y ... *sleeeeeeeeeeep forever* ...

Child: a b

```
void thread_exit() {  
    done = 1;           // a  
    Cond_signal(&c);    // b  
}
```

```
void thread_join() {  
    Mutex_lock(&m);     // w  
    if (done == 0)     // x  
        Cond_wait(&c, &m); // y  
    Mutex_unlock(&m);  // z  
}
```

# Broken Implementation 2

Parent: w x y ... *sleeeeeeeeeeep forever* ...

Child: a b

```
void thread_exit() {  
    done = 1;           // a  
    Cond_signal(&c);    // b  
}  
  
void thread_join() {  
    Mutex_lock(&m);     // w  
    if (done == 0)     // x  
        Cond_wait(&c, &m); // y  
    Mutex_unlock(&m);  // z  
}
```

How to fix?

# Broken Implementation 2

Parent: w x y ... *sleeeeeeeeeeep forever* ...

Child: a b

*child*  
↓

```
void thread_exit() {  
    done = 1;           // a  
    Cond_signal(&c);    // b  
}  
  
void thread_join() {  
    Mutex_lock(&m);  
    while if (done == 0)   
        Cond_wait(&c, &m);  
    Mutex_unlock(&m);  
}  
  
// w  
// x  
// y  
// z
```

*Mutex\_lock(&m);*

*Mutex\_unlock(&m);*

# Trap 1 When Using CV



# Trap 1 When Using CV





# Trap 1 When Using CV



# Trap 1 When Using CV



Only one thread gets a signal

# Trap 2 When Using CV

Condition Variable

# Trap 2 When Using CV



# Trap 2 When Using CV

Condition Variable

# Trap 2 When Using CV



# Trap 2 When Using CV



# Trap 2 When Using CV



Signal lost if nobody waiting at that time



# Guarantee

Upon signal, there has to be **at least one** thread waiting;  
If there are threads waiting, **at least one** thread will wake



FIFO

```
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 }
```

## CV-based Parent-wait-for-child Approach

```
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 } Y.Cheng
```

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

```
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
```

## CV-based Parent-wait-for-child Approach

```
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
```

## Good Rule of Thumb

Always do **1. wait** and **2. signal** while **holding the lock**

```
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 }
```

**Why:** To prevent lost signal

```
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 } Y.Cheng
```

# Classical Problems of Synchronization

- Producer-consumer problem
  - CV-based version
- Readers-writers problem
- Dining-philosophers problem

# CV-based Producer-Consumer Implementation 1

## Single CV and if statement

```
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);
    }
}
```

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

Put and Get routines  
Single buffer

# CV-based Producer-Consumer Implementation 1

## Single CV and if statement

```
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);
    }
}
```

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

## Put and Get routines Single buffer

What's the problem of this approach?

# CV-based Producer-Consumer Implementation 1

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

← C1 running

```
Pthread_mutex_lock(&mutex); // c1
if (count == 0) // c2
    Pthread_cond_wait(&cond, &mutex); // c3
```

```
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
    }
}
```

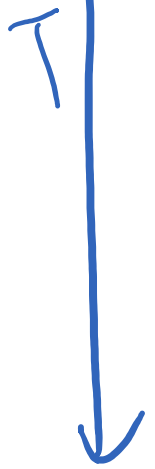
C1

C2

P

empty

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	<u>Nothing to get</u>



# CV-based Producer-Consumer Implementation 1

 P running

```
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);
    }
}
```

```
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
    }
}
```

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
c1	Running		Ready		Ready	0	Nothing to get
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
						0	



# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

```
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
    }
}
```

← P running

$T_{c1}$	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

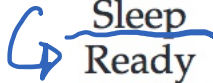
# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

```
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
    }
}
```

← P running

$T_{c1}$	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



# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

```
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
    }
}
```

**← P running**

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
c1	Running		Ready		Ready	0	Nothing to get
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	
	Ready		Ready	p5	Running	1	Buffer now full $T_{c1}$ awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	

# CV-based Producer-Consumer Implementation 1

```


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);
    }
}

```

```

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
    }
}

```

 **P running**

$T_{c1}$	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	
	Ready		Ready	p5	Running	1	
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	

Buffer now full  
 $T_{c1}$  awoken

# CV-based Producer-Consumer Implementation 1

```

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);
    }
}

```



C1 runnable

```

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
    }
}

```

$T_{c1}$	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	<u>p3</u>	Sleep	1	Buffer full; sleep

# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

← C2 running

```
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
    }
}
```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c2</sub> sneaks in ...

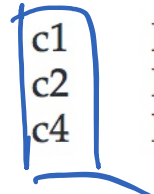
# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

 **C2 running**

```
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
    }
}
```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c2</sub> sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data



# CV-based Producer-Consumer Implementation 1

```

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);
    }
}

```

← C2 running

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c2</sub> sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T <sub>p</sub> awoken



# CV-based Producer-Consumer Implementation 1

```

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);
    }
}

```

← C2 running

```

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
    }
}

```

$T_{c1}$	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	

# CV-based Producer-Consumer Implementation 1

```
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);
    }
}
```

← C1 running

```
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
    }
}
```

<u>T<sub>c1</sub></u>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c2</sub> sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T <sub>p</sub> awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

# CV-based Producer-Consumer Implementation 2



## Single CV and while

```
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 }
```

# CV-based Producer-Consumer Implementation 2

## Single CV and while

```
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 }
```

What's the problem of this approach?

C1 running



```

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);
  }
}

```

```

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
  }
}

```

<u>T<sub>c1</sub></u>	State	T <sub>c2</sub>	State	T <sub>p</sub>	State	Count	Comment
c1	Running		Ready		Ready	0	Nothing to get
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	

cond\_wait() {  
 unlock() // sleep ...  
 // woken ...  
 lock(-)  
 }

← C2 running

```

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);
    }
}

```

```

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
    }
}

```

$T_{c1}$	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	<u>Sleep</u>		Ready	0	<u>Nothing to get</u>

P running



```

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);
  }
}

```

```

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
  }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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

P running



```

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);
  }
}

```

```

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
  }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> awoken



P sleeping



```

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);
    }
}

```

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)

```

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);
    }
}

```

← C1 running

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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

```

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);
    }
}

```

← C1 running

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c1</sub> grabs data

```

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);
    }
}

```

← C1 running

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c1</sub> grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T <sub>c2</sub>



C1 sleeping

```

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);
    }
}

```

```

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
    }
}

```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c1</sub> grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T <sub>c2</sub>
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get



← C2 running

```
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);
    }
}
```

```
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
    }
}
```

T <sub>c1</sub>	State	T <sub>c2</sub>	State	T <sub>p</sub>	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 <sub>c1</sub> 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 <sub>c1</sub> grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T <sub>c2</sub>
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...

## C2 sleeping

```

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);
    }
}

```

```

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
    }
}

```

$T_{c1}$	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...

# CV-based Producer-Consumer Implementation 3

```
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 }
```

Two CVs and while



# CV-based Producer-Consumer Implementation 3

```
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 }
```

## Two CVs and while

Using **two CVs** to distinguish two types of threads; in order to properly signal which thread should wake up

- Producer waits on **empty**
- Consumer waits on **full**

# Readers-Writers Problem

# Readers-Writers Problem

- A data object (e.g. a file) is to be shared among several concurrent processes/threads
- A writer process/thread must have exclusive access to the data object
- **Multiple reader** processes/threads may access the shared data simultaneously without a problem

# Reader-Writer Lock

RWlock.

```
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 }
```

# Reader-Writer Lock

```
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 }
```

# Reader-Writer Lock

```
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 }
```

Initially, # readers is 0  
binary sem lock set to 1  
writelock set to 1

# Reader-Writer Lock

```
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 }
```

Initially, # readers is 0  
binary sem lock set to 1  
writelock set to 1

# Reader-Writer Lock

```
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 }
```

Initially, # readers is 0  
binary sem lock set to 1  
writelock set to 1



# Reader-Writer Lock

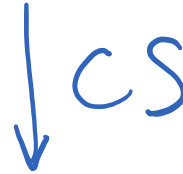
```
1 typedef struct _rwlock_t {
2     sem_t lock; // binary semaphore (basic lock)
3     sem_t writelock; // used to allow ONE writer or MANY readers
4     int readers; // count of readers reading in critical section
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 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 }
```

Initially, # readers is 0  
binary sem lock set to 1  
writelock set to 1

**Writer cannot  
be in CS when  
readers are!**

# Readers-Writers Problem: Writer Thread

```
rwlock_acquire_writelock(rw);  
    ...  
    write is performed  
    ...  
rwlock_release_writelock(rw);
```



# Readers-Writers Problem: Reader Thread

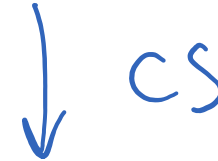
```
rwlock_acquire_readlock(rw)
```

...

```
read is performed
```

...

```
rwlock_release_readlock(rw)
```



**Well, is this solution Okay?**


# Readers-Writers Problem: Reader Thread

```
rwlock_acquire_readlock(rw)
    ...
    read is performed
    ...
rwlock_release_readlock(rw)
```

**Well, is this solution Okay?**

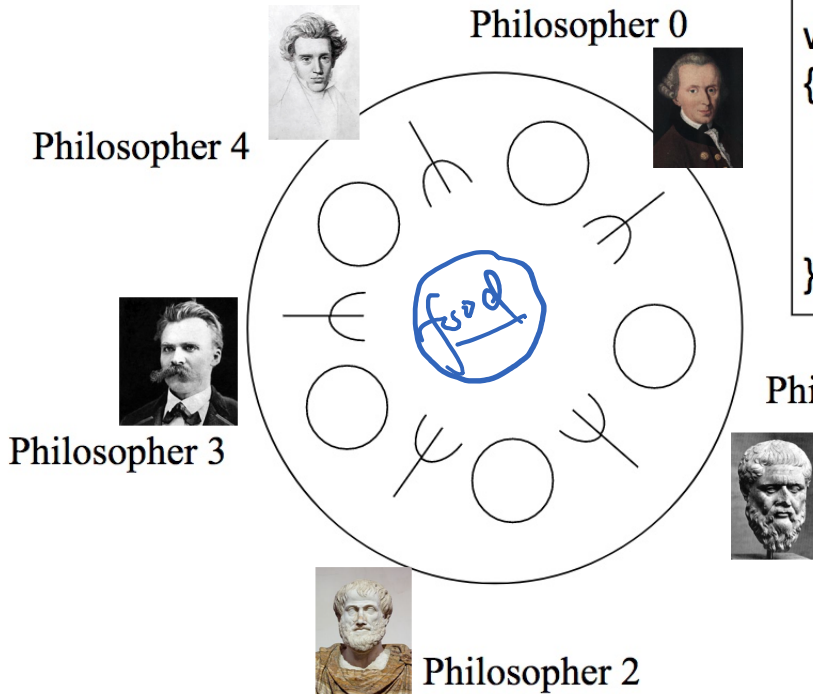
A: Technically it works. But **starvation** may happen

# Starvation

- A process/thread that is forced to wait **indefinitely** in a synchronization program is said to be subject to **starvation**
  - In some execution scenarios, that process does not make any progress
  - **Deadlocks imply starvation, but the reverse is not true** 

# Dining-Philosophers Problem

# Dining-Philosophers Problem



```
while(food available)
{pick up 2 adj. forks;
 eat;
 put down forks;
 think awhile;
}
```

Philosopher 1

- 5 philosophers share a common circular table. There are 5 forks (or chopsticks) and food (in the middle). When a philosopher gets hungry, he tries to pick up the closest forks
- A philosopher may pick up only one fork at a time, and cannot pick up a fork already in use. When done, he puts down both of his forks, one after the other

Shared data

```
sem_t forks[5];
```

Initially all semaphore values are 1

# Dining-Philosophers Problem

- The basic loop of a philosopher

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

??

Critical section

??



# The Helper Functions

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

sem\_t forks[5]

- Each fork initialized to 1

```
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 }
```

**Is this solution correct?**

# Simplest Example of A Deadlock

$T_0$   $[f_0, f_1]$   $T_1$

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])
```

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])  
  
sem_wait(fork[1])
```

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])  
  
sem_wait(fork[1])  
  
sem_wait(fork[0])
```

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])  
  
sem_wait(fork[1])  
  
sem_wait(fork[0])  
wait...  
  
sem_wait(fork[1])
```

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

Thread 0

```
sem_wait(fork[0])  
sem_wait(fork[1])  
sem_signal(fork[0])  
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])  
  
sem_wait(fork[1])  
  
sem_wait(fork[0])  
wait...  
  
sem_wait(fork[1])  
wait...
```

Thread 1

```
sem_wait(fork[1])  
sem_wait(fork[0])  
sem_signal(fork[1])  
sem_signal(fork[0])
```

# Simplest Example of A Deadlock

**Q: Would the previous 5DP implementation cause exactly the same form of a deadlock as shown below?**

Thread 0

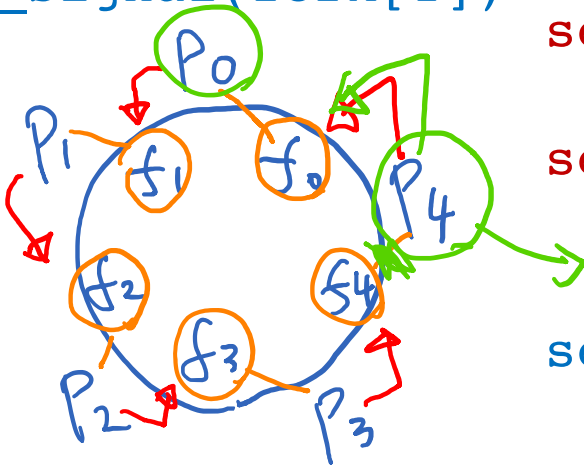
```
sem_wait(fork[0])
sem_wait(fork[1])
sem_signal(fork[0])
sem_signal(fork[1])
```

Interleaving

```
sem_wait(fork[0])
sem_wait(fork[1])
sem_wait(fork[1])
sem_wait(fork[0])
wait...
sem_wait(fork[1])
wait...
```

Thread 1

```
sem_wait(fork[1])
sem_wait(fork[0])
sem_signal(fork[1])
sem_signal(fork[0])
```






# Review: Conditions for Deadlocks

- Mutually exclusive access of shared resources
  - Binary semaphore `fork[0]` and `fork[1]`

# Review: Conditions for Deadlocks

- Mutually exclusive access of shared resources
  - Binary semaphore `fork[0]` and `fork[1]`
- Circular waiting
  - Thread 0 waits for Thread 1 to signal(`fork[1]`) and
  - Thread 1 waits for Thread 0 to signal(`fork[0]`)

# Review: Conditions for Deadlocks

- Mutually exclusive access of shared resources
    - Binary semaphore `fork[0]` and `fork[1]`
  - Circular waiting
    - Thread 0 waits for Thread 1 to `signal(fork[1])` and
    - Thread 1 waits for Thread 0 to `signal(fork[0])`
  - Hold and wait
    - Holding either `fork[0]` or `fork[1]` while waiting on the other
- 

# Review: Conditions for Deadlocks

- Mutually exclusive access of shared resources
  - Binary semaphore `fork[0]` and `fork[1]`
- Circular waiting
  - Thread 0 waits for Thread 1 to signal(`fork[1]`) and
  - Thread 1 waits for Thread 0 to signal(`fork[0]`)
- Hold and wait
  - Holding either `fork[0]` or `fork[1]` while waiting on the other
- No preemption
  - Neither `fork[0]` and `fork[1]` can be removed from their respective holding threads

# Why 5DP is Interesting?

- How to eat with your fellows without causing deadlocks
  - Circular arguments (the **circular wait condition**)
  - Not giving up on firmly held things (**no preemption**)
  - Infinite patience with half-baked schemes (**hold some & wait for more**)
- Why starvation exists and what we can do about it?

# Why 5DP is Interesting?

- ~~How to eat with your fellows without causing deadlocks~~ — **How to mess with your fellows!**
  - Circular arguments (the **circular wait condition**)
  - Not giving up on firmly held things (**no preemption**)
  - Infinite patience with half-baked schemes (**hold some & wait for more**)
- Why starvation exists and what we can do about it?

# Dijkstra's Solution:

## Break the Circular Wait Condition

- Change how forks are acquired by at least one of the philosophers
- Assume P0 – P4, 4 is the highest number

```
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 }
```

# Again, Starvation

- Subtle difference between deadlock and starvation
  - Once a set of processes are in a deadlock, there is **no future execution sequence** that can get them out of it!
  - In starvation, there does exist hope – some execution order may be favorable to the starving process although no guarantee it would ever occur
- Rollback and retry are prone to starvation
- Continuous arrival of higher priority process is another common starvation situation



# **Building a Semaphore w/ CV Worksheet**